Studies in Computational Intelligence, Volume 358

Editor-in-Chief
Prof. Janusz Kacprzyk
Systems Research Institute
Polish Academy of Sciences
ul. Newelska 6
01-447 Warsaw
Poland
E-mail: kacprzyk@ibspan.waw.pl

Further volumes of this series can be found on our homepage: springer.com

Vol. 336. Paolo Remagnino, Dorothy N. Monekosso, and Lakhmi C. Jain (Eds.)
Innovations in Defence Support Systems – 3, 2011
ISBN 978-3-642-18277-8

Vol. 337. Sheryl Brahnam and Lakhmi C. Jain (Eds.)
Advanced Computational Intelligence Paradigms in Healthcare 6, 2011
ISBN 978-3-642-17823-8

Vol. 338. Lakhmi C. Jain, Eugene V. Aidman, and Canicious Abeynayake (Eds.)
Innovations in Defence Support Systems – 2, 2011
ISBN 978-3-642-17763-7

Vol. 339. Halina Kwasnicka, Lakhmi C. Jain (Eds.)
Innovations in Intelligent Image Analysis, 2010
ISBN 978-3-642-17933-4

Vol. 340. Heinrich Hussmann, Gerrit Meixner, and Detlef Zuehlke (Eds.)
Model-Driven Development of Advanced User Interfaces, 2011
ISBN 978-3-642-14561-2

Vol. 341. Stéphane Doncieux, Nicolas Bredeche, and Jean-Baptiste Mouret(Eds.)
New Horizons in Evolutionary Robotics, 2011
ISBN 978-3-642-18271-6

Vol. 342. Federico Montesino Pouzols, Diego R. Lopez, and Angel Barriga Barros
Mining and Control of Network Traffic by Computational Intelligence, 2011
ISBN 978-3-642-18083-5

Vol. 343. Kurosh Madani, António Dourado Correia, Agostinho Rosa, and Joaquim Filipe (Eds.)
Computational Intelligence, 2011
ISBN 978-3-642-20205-6

Vol. 344. Atilla Elçi, Mamadou Tadiou Koné, and Mehmet A. Orgun (Eds.)
Semantic Agent Systems, 2011
ISBN 978-3-642-18307-2

Vol. 345. Shi Yu, Léon-Charles Tranchevent, Bart De Moor, and Yves Moreau
Kernel-based Data Fusion for Machine Learning, 2011
ISBN 978-3-642-19405-4

Vol. 346. Weisi Lin, Dacheng Tao, Janusz Kacprzyk, Zhu Li, Ebroul Izquierdo, and Haohong Wang (Eds.)
Multimedia Analysis, Processing and Communications, 2011
ISBN 978-3-642-19550-1

Vol. 347. Sven Helmer, Alexandra Poulovassilis, and Fatos Xhafa
Reasoning in Event-Based Distributed Systems, 2011
ISBN 978-3-642-19723-9

Vol. 348. Beniamino Murgante, Giuseppe Borruso, and Alessandra Lapucci (Eds.)
Geocomputation, Sustainability and Environmental Planning, 2011
ISBN 978-3-642-19732-1

Vol. 349. Vitor R. Carvalho
Modeling Intention in Email, 2011
ISBN 978-3-642-19955-4

Vol. 350. Thanasis Daradoumis, Santi Caballé, Angel A. Juan, and Fatos Xhafa (Eds.)
Technology-Enhanced Systems and Tools for Collaborative Learning Scaffolding, 2011
ISBN 978-3-642-19813-7

Vol. 351. Ngoc Thanh Nguyen, Bogdan Trawiński, and Jason J. Jung (Eds.)
New Challenges for Intelligent Information and Database Systems, 2011
ISBN 978-3-642-19952-3

Vol. 352. Nik Bessis and Fatos Xhafa (Eds.)
Next Generation Data Technologies for Collective Computational Intelligence, 2011
ISBN 978-3-642-20343-5

Vol. 353. Igor Aizenberg
Complex-Valued Neural Networks with Multi-Valued Neurons, 2011
ISBN 978-3-642-20352-7

Vol. 354. Ljupco Kocarev and Shiguo Lian (Eds.)
Chaos-Based Cryptography, 2011
ISBN 978-3-642-20541-5

Vol. 355. Yan Meng and Yaochu Jin (Eds.)
Bio-Inspired Self-Organizing Robotic Systems, 2011
ISBN 978-3-642-20759-4

Vol. 356. Slawomir Koziel and Xin-She Yang (Eds.)
Computational Optimization, Methods and Algorithms, 2011
ISBN 978-3-642-20858-4

Vol. 357. Nadia Nedjah, Leandro Santos Coelho, Viviana Cocco Mariani, and Luiza de Macedo Mourelle (Eds.)
Innovative Computing Methods and Their Applications to Engineering Problems, 2011
ISBN 978-3-642-20957-4

Vol. 358. Norbert Jankowski, Włodzisław Duch, and Krzysztof Grąbczewski (Eds.)
Meta-Learning in Computational Intelligence, 2011
ISBN 978-3-642-20979-6

Norbert Jankowski, Włodzisław Duch,
and Krzysztof Grąbczewski (Eds.)

Meta-Learning in Computational Intelligence

Springer

Editors

Dr. Norbert Jankowski
Department of Informatics
Nicolaus Copernicus University
ul. Grudziądzka 5
87-100 Toruń
Poland
Email: norbert@is.umk.pl

Dr. Krzysztof Grąbczewski
Department of Informatics
Nicolaus Copernicus University
ul. Grudziądzka 5
87-100 Toruń
Poland
Email: kg@is.umk.pl

Professor Włodzisław Duch
Department of Informatics
Nicolaus Copernicus University
ul. Grudziądzka 5
87-100 Toruń
Poland
Email: wduch@is.umk.pl

ISBN 978-3-642-20979-6 e-ISBN 978-3-642-20980-2

DOI 10.1007/978-3-642-20980-2

Studies in Computational Intelligence ISSN 1860-949X

Library of Congress Control Number: 2011929217

Typeset & *Cover Design:* Scientific Publishing Services Pvt. Ltd., Chennai, India.

Printed on acid-free paper

9 8 7 6 5 4 3 2 1

springer.com

Preface

In the early days of pattern recognition and statistical data analysis life was rather simple: datasets were relatively small, collected from well-designed experiments, analyzed using a few methods that had good theoretical background. Explosive growth of the use of computers led to the creation of huge amounts of data of all kinds, coming from business, finance, scientific experiments, medical evaluations, Web-related data, multimedia, various imaging techniques, sensor networks, and many others sources. Human experts are not capable of deep exploration of large amounts of data, especially when an expertize in several different areas is necessary for this purpose.

The need for scalable algorithms applicable for massive data sets, discovering novel patterns in data, contributed to the growing interest in data mining and knowledge discovery. It led to the development of new machine learning techniques, including the use of inspirations from nature to develop evolutionary, neural and fuzzy algorithms for data analysis and understanding. Computational intelligence became a field in its own, combining all these methods, and the number of algorithms available for data analysis was rapidly growing. These algorithms started to be collected in larger data mining packages, with a free package called Weka starting the trend, and many others that soon followed. These packages added better user interfaces, environments to build data flow schemes by connecting various modules and test various algorithms. However, the number of modules at each stage: pre-processing, data acquisition, feature selection and construction, instance selection, classification, association and approximation methods, optimization techniques, pattern discovery, clusterization, visualization and post-processing became unmanageably large. A large data mining package allows for billions of combinations, making the process of knowledge discovery increasingly difficult. Gone are the days when the life of a data miner was simple and a background course in multivariate statistics was all that was needed to do the job.

Paradoxically, the abundance of methods did not simplify the process of data analysis. Despite the "no free lunch" theorem the work in computational intelligence has been focused largely on universal algorithms, that should beat all others in wide range of situations. Large packages allow for construction of more specialized algorithms, properly biased for a given data. Constructing a successful chain of data transformations to create optimal decision models should not consist in picking arbitrarily some components and gluing them together. On the other hand, testing all possible combinations is not feasible, because the number of combinations of data transformations at each stage is too large, and any drastic reduction of the set of components that prevents combinatorial explosion reduces the chance of finding interesting models. Human experts can help, but

even for trivial data experts tend to create complex systems that miss simple solutions. For complex data methodology that guarantees finding appropriate combination of CI transformation does not exist. Some of these methods are redundant, generating almost identical solutions. Advanced data mining framework in hand of an amateur is like a scalpel in hand of a Parkinson patient trying to perform neurosurgery on his colleague. It can only result in a brain salad surgery. Even the best experts do not know weak and strong points of all methods available.

This is where *meta-learning* comes to rescue. The term "meta-learning" still means different things to different people, but in general it covers a broad range of approaches directed at *learning how to learn*. Humans are adapted to the niche of adaptability, not to a particular environment, and this is where the superiority of human intelligence lies. Learning of each method requires various specialized selection and adaptation procedures. Meta-learning, operating in the space of available data transformations and optimization techniques, should learn from experience of solving different problems, make inferences about transformations useful in different contexts, and finally help to construct interesting learning algorithm that can uncover various aspects knowledge hidden in data. In general meta-learning should do what a good expert does.

The last decade have brought many interesting ideas within the scope of meta-learning, making this field a hot topic. Various approaches that address meta-learning challenge at different levels of abstraction include:

- flexible frameworks for data analysis, facilitating systematic composition of transformations and optimization methods to build data models;
- methods that analyze families of algorithms within a given (usually narrow) framework;
- methods that support general context-sensitive analysis of the learning processes;
- methods for meta-knowledge acquisition from human experts;
- methods that build interesting models using meta-knowledge, verify these models and provide information to improve or correct meta-knowledge;
- theoretical approaches and principles that help to understand learning methods.

These methods are often focused on a narrow subtask of improving particular aspects of gaining, building, using, verifying and correcting meta-knowledge. The problem is so demanding that every little nugget of meta-knowledge is precious. Hence all tools that are capable of contributing to the final goal are valuable, including knowledge generated from experiments with many learning algorithms and also knowledge provided by human experts. Meta-learning should eventually simplify the work of experts and non-experts, whose job will be to formulate interesting questions and interpret the results, rather than to fight with tedious technicalities required to create models, as it is done now.

In this book the reader will find 10 chapters summarizing research in many directions relevant to meta-learning. The first chapter, "Universal meta-learning architecture and algorithms", written by Norbert Jankowski and Krzysztof Grąbczewski, is concerned with the most important problem, how should the meta-learning search be organized. In this chapter authors propose sophisticated representation of meta-search space in a form of specialized graph of learning machines generators. The process of meta-learning is advised by dedicated mechanism of test tasks ordered by approximation of task complexity, which is guided by complexity evaluators. The proposed framework is capable of using several types of meta-knowledge (for example knowledge based on optimization procedures or attractiveness of learning machines).

In the second chapter Kate Smith-Miles and Rafiqul Islam write about "Meta-learning of instance selection for data summarization". Selection of instances or prototypes is important especially for very large datasets as well as for understanding the data. Selection is directed here by dedicated meta-learning algorithm based on meta-features extracted especially for this task, although they should also be useful for other problems.

Chapter three, by Damien François, Vincent Wertz and Michel Verleysen is focused on the problem of automatic choice of the metric in similarity-based approaches realized with a help of meta-learning. This is one of the specific aspects of data mining.

Chapter four, "Meta-learning Architectures: Collecting, Organizing and Exploiting Meta-knowledge", by Joaquin Vanschoren is a nice *rendez-vous* through different meta-learning algorithms, discussing their strategy of learning. It presents information about different ways of collecting and using of meta-knowledge. Authors also compare different architectures of meta-learning systems and their information flows, presenting in the end an interesting proposal for extended meta-learning system based on *consolidation* of different approaches.

Chapter five, "Computational intelligence for meta-learning: a promising avenue of research" by Ciro Castiello and Anna Maria Fanelli, presents a system based on neuro-fuzzy hybridization. Theoretical discussion of various aspects of meta-learning architectures is given from the point of view of such hybrid systems. This chapter goes beyond selection of learning machines, focusing on deeper analysis of the learning model behavior. Applying neuro-fuzzy approach to standard- and meta-levels of learning an integrated fuzzy logic system is constructed.

In chapter six, Pavel Kordík and Jan Černý discuss self-organization of supervised models within their Fully Automated Knowledge Extraction framework. Diverse local expert models are combined within a cascade-like ensemble using a group of adaptive models evaluation algorithm based on specialized evolutionary optimization and statistical ensemble techniques, including bagging, boosting and stacking.

In chapter seven Ricardo B. C. Prudêncio, Marcilio C. P. de Souto and Teresa B. Ludermir describe how to select machine learning algorithms using the ranking approach. This chapter presents application of meta-learning to the prediction of time series and to clustering. In this case meta-learning is based mainly on extraction of various meta-features that characterize the dataset.

Chapter eight, by Talib S. Hussain, is more theoretical, introducing "A Meta-Model Perspective and Attribute Grammar Approach to Facilitating the Development of Novel Neural Network Models". This chapter presents an interesting direction for systematic creation of complex neural networks that integrate multiple diverse models.

In chapter nine Melanie Hilario, Phong Nguyen, Huyen Do and Adam Woznica write about "Ontology-Based Meta-Mining of Knowledge Discovery Workflows". This is one of the more advanced meta-learning frameworks, going beyond systems based on meta-features that use correlations between data and performance of base learning machines. Data mining ontology is used for optimization process, and a workflow representation is used to compose complex learning machines defined by direct acyclic graphs.

In the final chapter Włodzisław Duch, Tomasz Maszczyk and Marek Grochowski focus on creation of optimal features spaces in which meta-learning may take place. In their approach parallel chains of transformations extract useful information granules that are used as additional hidden features. Resulting algorithms facilitate deep learning by composition of transformations, and enable understanding of data structures by visualization of data distribution after transformations, and by creating various logical rules based on the new features.

We hope that this book will help to find valuable information about the new trends in meta-learning and will inspire the readers to further research in the field of meta-learning methods.

Contents

Universal Meta-Learning Architecture and Algorithms

Norbert Jankowski and Krzysztof Grąbczewski

Department of Informatics
Nicolaus Copernicus University
Toruń, Poland
{norbert,kg}@is.umk.pl
http://www.is.umk.pl/

Abstract. There are hundreds of algorithms within data mining. Some of them are used to transform data, some to build classifiers, others for prediction, etc. Nobody knows well all these algorithms and nobody can know all the arcana of their behavior in all possible applications. How to find the best combination of transformation and final machine which solves given problem?

The solution is to use configurable and efficient meta-learning to solve data mining problems. Below, a general and flexible meta-learning system is presented. It can be used to solve different problems with computational intelligence, basing on learning from data.

The main ideas of our meta-learning algorithms lie in complexity controlled loop, searching for most adequate models and in using special functional specification of search spaces (the meta-learning spaces) combined with flexible way of defining the goal of meta-searching.

Keywords: Meta-Learning, Data Mining, Learning Machines, Computational Intelligence, Data Mining System Architecture, Computational Intelligence System Architecture.

1 Introduction

Recent decades have brought large amount of data, eligible for automated analysis that could result in valuable descriptions, classifiers, approximators, visualizations or other forms of models. The Computational Intelligence (CI) community has formulated many algorithms for data transformation and for solving classification, approximation and other optimization problems [1]. The algorithms may be combined in many ways, so that the tasks of finding optimal solutions are very hard and require sophisticated tools. Nontriviality of model selection is evident when browsing the results of NIPS 2003 Challenge in Feature Selection [2,3], WCCI Performance Prediction Challenge [4] in 2006 or other similar contests.

Most real life learning problems can be reasonably solved only by complex models, revealing good cooperation between different kinds of learning machines.

N. Jankowski et al. (Eds.): Meta-Learning in Computational Intelligence, SCI 358, pp. 1–76.
springerlink.com

To perform successful learning from data in an automated manner, we need to exploit meta-knowledge i.e. the knowledge about how to build an efficient learning machine providing an accurate solution to the problem being solved.

One of the approaches to meta-learning develops methods of decision committees construction, different stacking strategies, also performing nontrivial analysis of member models to draw committee conclusions [5,6,7,8,9]. Another group of meta-learning enterprises [10,11,12,13] base on data characterization techniques (characteristics of data like number of features/vectors/classes, features variances, information measures on features, also from decision trees etc.) or on *landmarking* (machines are ranked on the basis of simple machines performances before starting the more power consuming ones) and try to learn the relation between such data descriptions and accuracy of different learning methods. Although the projects are really interesting, they still suffer from many limitations and may be extended in a number of ways. The whole space of possible and interesting models is not browsed so thoroughly, thereby some types of solutions can not be found with this kind of approaches.

In gating neural networks [14] authors use neural networks to predict performance of proposed *local experts* (machines preceeded by transformations) and decide about final decision (the best combination learned by regression) of the whole system. Another application of meta-learning to optimization problems, by building relations between elements which characterize the problem and algorithms performance, can be found in [15].

We do not believe that on the basis of some, not very sophisticated or expensive, description of the data, it is possible to predict the structure and configuration of the most successful learner. Thus, in our approach the term *meta-learning* encompasses the whole complex process of model construction including adjustment of training parameters for different parts of the model hierarchy, construction of hierarchies, combining miscellaneous data transformation methods and other adaptive processes, performing model validation and complexity analysis, etc. So in fact, our approach to meta-learning is a search process, however not a naive search throughout the whole space of possible models, but a search driven by heuristics protecting from spending time on learning processes of poor promise and from the danger of combinatorial explosion.

This article presents many aspects of our meta-learning approach. In Section 2 we present some basic assumptions and general ideas of our efforts. Section 3 presents the main ideas of the computational framework we have developed to make deeper meta-level analysis possible. Next, Section 4 describes the Meta Parameter Search Machine, which supports simple searches within the space of possible models. Section 5 is the most important part which describe main parts of meta-learning algorithm (definition of configuration of meta-learning, elements of scheme of main algorithm presented in Section 2, complexity control engine). Section 7 presents example application of proposed meta-learning algorithm for variety of benchmark data streams.

2 General Meta-Learning Framework

First the difference between learning and meta-learning should be pointed out. Both the learning and meta-learning are considering in the context of learning from data, which is common around computational intelligence problems. Learning process of a *learning machine* $\mathcal{L}$ is a function $\mathcal{A}(\mathcal{L})$:

$$\mathcal{A}(\mathcal{L}) : \mathcal{K}_{\mathcal{L}} \times \mathcal{D} \rightarrow \mathcal{M}, \tag{1}$$

where $\mathcal{K}_{\mathcal{L}}$ represents the space of configuration parameters of given learning machine $\mathcal{L}$, $\mathcal{D}$ defines the space of data streams (typically a single data table, sometimes composed by few independent data inputs), which provide the learning material, and $\mathcal{M}$ defines the space of goal models. Models should play a role (assumed by $\mathcal{L}$) like classifier, feature selector, feature extractor, approximator, prototype selector, etc.

Indeed, learning should be seen as the process in which some *free parameters* of the machine M are adjusted or determined according to a *strategy* (algorithm) of the learning machine $\mathcal{L}$. After the learning process of $\mathcal{L}$, the model M should be ready to use as a classifier, data transformer, etc. depending on the goal of $\mathcal{L}$.

From such point of view meta-learning is another or rather specific learning machine. In the case of meta-learning the learning phase learn how to learn, to learn as well as possible. This means that the target model of a meta-learning machine (as the output of meta-learning) is a configuration of a learning machine extracted by meta-learning algorithm. The configuration produced by meta-learning should play the goal-role (like, already mentioned, classifier, approximator, data transformer, etc.) of meta-learning task. It is important to see that meta-learning is obligated to chose machine type (it may be even very complex one) and their strict configuration. This is because different configuration may provide incomparable behavior of given learning machine. Of course such definition does not indicate, how the meta-learning should search for the best type of learning machine and their best configuration.

Almost always meta-learning algorithms learn by observation and testing of nested learning machines. Meta-learning differ in the strategy of selection, which learning machines to observe and what to observe in chosen machines to find possibly best or at least satisfactory conclusions. From the theoretical point of view meta-learning, in general, is not limited in any way except the limitation of memory and time.

We propose a unified scheme of meta-learning algorithms (MLAs) which base on learning from observations. It is depicted in Figure 1.

The initialization step is a *link* between given configuration of meta-learning (which is very important question—see Section 5.2) and the further steps.

The meta-learning algorithm, after some initialization, starts the main loop, which up to the given *stop condition*, *runs* different learning processes, *monitors* them and *concludes* from their gains. In each repetition, it defines a number of tasks which test the behavior of appropriate learning configurations (i.e. configurations of single or complex learning machines)—step *start some tasks*. In

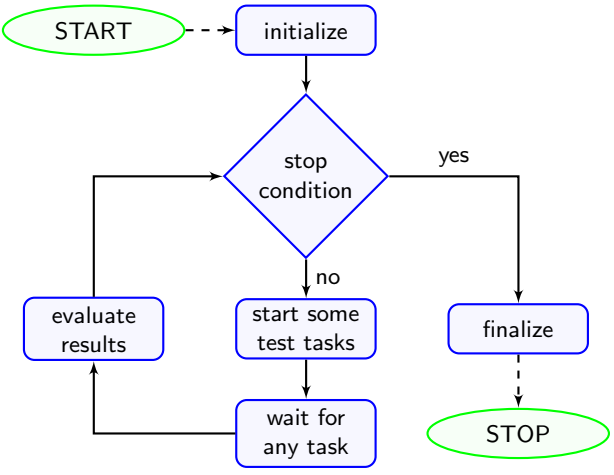


Fig. 1. General meta-learning algorithm.

other words, at this step it is decided, which (when and whether) machines are tested and how it is done (the strategy of given MLA). In the next step (*wait for any task*) the MLA waits until any test task is finished, so that the main loop may be continued. A test task may finish in a natural way (at the assumed end of the task) or due to some exception (different types of errors, broken by meta-learning because of exceeded time limit and so on). After a task is finished, its results are analyzed and *evaluated*. In this step some results may be accumulated (for example saving information about best machines) and new knowledge items created (e.g. about different machines cooperations). Such knowledge may have crucial influence on further parts of the meta-learning (tasks formulation and the control of the search through the space of learning machines). Precious conclusions may be drawn, even if a task is finished in a non-natural way.

When the *stop condition* becomes satisfied, the MLA prepares and returns the final result: the configuration of chosen learning machine or, in more general way, even a ranking of learning machines (ordered by a degree of goal satisfaction), comments on chosen learning machines and their interactions, etc.

Each of the key steps of this general meta-learning algorithm may be realized in different ways yielding different meta-learning algorithms.

It is important to see that such a general scheme is not limited to a single strategy of MLA or searching by observing task by task (MLA autonomously decides about current group of started test tasks). This scheme does not apply any limits to the learning machine search space which in general can be a non-fixed space and may evaluate in the progress of meta search, for example to produce complex substitutions of machines. This opens the gates even to directing the functional space of learning machines according to collected meta-knowledge.

Also the stop condition may be defined according to the considered problem and their limits.

This meta-scheme may be used to solve different types of problems. It is not limited only to classification or approximation problems.

First, note that finding an optimal model for given data mining problem $\mathcal{P}$ is almost always NP hard. Because of that, meta-learning algorithms should focus on finding approximation to the optimal solution independently of the problem type. Second, it would be very useful if the meta-learning could find solutions which at least are not worst than the ones that can be found by human experts in data mining in given limited amount of time. "At least" because usually meta-learning should find more attractive solutions, sometimes even of surprising structure. In general meta-learning is more open to make deeper observation of intermediate test tasks and the search procedure may be more exhaustive and consistent. Experts usually restrict their tests to a part of algorithms and to some schemes of using them. Sophisticated meta-learning may quite easily overcome such disadvantages simultaneously keeping high level of flexibility.

This is why our **general goal of the meta-learning** is *to maximize the probability of finding possibly best solution of given problem $\mathcal{P}$ within a search space in as short time as possible.*

As a consequence of such definition of the goal, the construction of meta-learning algorithm should carefully advise the order of testing tasks during the progress of the search and build meta-knowledge based on the experience from passed tests. Meta-knowledge may cover experience of so different kinds, among others: the correlations between subparts of machines in the context of performance, experience connected to the complexities of machines etc.

In our meta-learning approach, the algorithms search not only among *base* learning machines, but also produce and test different, sometimes quite complex machines like compositions of (several) transformations and classifiers (or other final e.i. decision making machines), committees of different types of machines, including complex ones (like composition of a transformer and a classifier as a single classifier inside the committee). Also, the transformations may be nested or compose chains. The compositions of complex machines may vary in their behavior and goal.

In the past, we have come up with the idea that meta-learning algorithms should favorite simple solutions and start the machines providing them before more complex ones. It means that MLAs should start with maximally plain learning machines, then they should test some plain compositions of machines (plain transformations with plain classifiers), after that more and more complex structures of learning machines (complex committees, multi-transformations etc.). But the problem is that the order of such generated tasks does not reflect real complexity of the tasks in the context of problem $\mathcal{P}$ described by data D. Let's consider two testing tasks T_1 and T_2 of computational complexities $O(mf^2)$ and $O(m^2f)$ respectively. Assume the data D is given in the form of data table and m is the number of instances and f is the number of features. In such case, it is not possible to compare time consumption of T_1 and T_2 until the final values

m and f are known. What's more, sometimes a composition of a transformation and a classifier may be indeed of smaller complexity than the classifier without transformation. It is true because when using a transformation, the data passed to the learning process of the classifier may be of smaller complexity and, as a consequence, classifier's learning is simpler and the difference between the classifier learning complexities, with and without transformation may be bigger than the cost of the transformation. This proves that real complexity is not reflected directly by structure of learning machine.

To obtain the right order in the searching queue of learning machines, a complexity measure should be used. Although the Kolmogorov complexity [16,17]

$$C_K(P) = \min_p \{l(p) \ : \text{program } p \text{ prints } P\} \tag{2}$$

is very well defined from theoretical point of view, it is unrealistic from practical side—the program p may work for a *very long* time. Levin's definition [17] introduced a term responsible for time consumption:

$$C_L(P) = \min_p \{c_L(p) \ : \text{program } p \text{ prints } P \text{ in time } t^p\} \tag{3}$$

where

$$c_L(p) = l(p) + \log(t^p). \tag{4}$$

This definition is much more realistic in practical realization because of the time limit [18,17]. Such definition of complexity (or similar, as it will be seen further in this paper) helps prepare the order according to the real complexity of test tasks.

Concluding this section, the meta-learning algorithm presented below as a special realization of the general meta-learning scheme described above, can be shortly summarized by the following items:

- The search is performed in a functional space of different learning algorithms and of different kinds of algorithms. Learning machines for the tests will be generated using specialized machines generators.
- The mail loop is controlled by checking the complexity of the test tasks. Complexity control is also useful to handle with halting problems of subsequent tasks, started by meta-learning.
- Meta-learning collects meta-knowledge basing on the intermediate test tasks. Using this knowledge the algorithm provides some correction of complexities, and changes the behavior of advanced machine generators, what has crucial role in defining the meta-space of learning machines. The knowledge may be accumulated per given problem but also may survive like for example in the case of the knowledge about complexities. Meta-learning algorithms may use meta-knowledge collected from other learning tasks.

What can be the role of meta-learning in the context of no free lunch *theorem?* Let's start from another side, from the point of view of a learning machine which has satisfactory level of validated(!) performance on the training data and smallest complexity among other machines of similar performance, such simplest solution has the highest probability of success on a test set, and it was shown in literature from several perspectives like bias-variance, minimum description length, regularization, etc [19,20,21,22,23]. From this side, in the case of classification problems, the process of meta-learning gives closer solution to the optimal Bayesian classifier than single (accidental?) learning machine.

The problem is that *no free lunch theorem* does not assume any relation of the distribution $P(X, Y)$ of the learning data D with the distribution $P(X', F(X'))$ $(X \subset X')$ of unknown target function $F(.)$ except being not contradictory at points of the data D. Within the context of given learning data D, not all targets have similarly high (or highest) probability of evidence.

The *perfect learning machine* should discover not the origin-target, but the most probable target. In other words: the goal of generalization is not to predict an unpredictable model.

3 General System Architecture

Advanced data mining, including meta-learning, is not possible without a general and versatile framework for easy and efficient management of different models, performing tests etc. One of the main keys to such a system is a unified view of *machines* and *models*. We define a machine as any process that can be configured and run to bring some results. The results of the processes constitute models. For example an MLP network algorithm [24] as the MLP machine can be configured by the network structure, initial weights, learning parameters etc. It can be run on some training data, and the result is a trained network—the MLP model created by the learning process of the MLP machine.

We deliberately avoid using the term "learning machine", since in our approach a machine can perform any process which we would, not necessarily, call a learning process, such as loading data from a disk file, data standardization or testing a classifier on external data.

A general view of a machine is presented in Figure 2. Before a machine may be created it must be completely configured and its context must be defined. Full machine configuration consists of:

- specification of inputs and outputs (how many, names and types),
- machine process parameters,
- submachines configuration (it is not depicted in Figure 2 to keep the figure clear; in further figures, starting with Figure 3, the submachines are visible as boxes placed within the parent machine).

Machine context is the information about:

- the parent machine (handled automatically by the system, when a machine orders creation of another machine) and the child index,

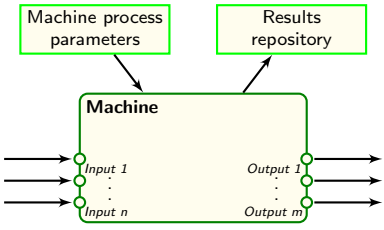


Fig. 2. The abstract view of a machine.

– input bindings i.e. the specification of other machines outputs that are to be passed as inputs to the machine to be created.

Some parts of machine configuration are fixed and do not require verbatim specification each time, a machine is created (e.g. the collection of inputs and outputs, for most machines, are always the same). Other configuration items usually have some (most common or most sensible) default values, so the machine user needs to specify only the items, that are different from the defaults.

A properly configured machine can be run (more precisely, the machine process is run to create the model). After the machine process is finished, the results may be deposited in the *results repository* and/or exhibited as *outputs*.

The inputs and outputs serve as sockets for information exchange between machines. The difference between machine inputs and configuration is that inputs come from other machines and the configuration contains the parameters of the process provided by the user. It is up to the machine author whether the machine receives any inputs and whether it has some adjustable parameters.

Similarly, machine outputs provide information about the model to other machines, and results repositories contain additional information about machine processes—the information that is not expected by other machines in the form of inputs.

The interconnections between outputs and inputs of different machines define the information flow within the project. Therefore, it is very important to properly encapsulate the CI functionality into machines. For more flexibility, each machine can create and manage a collection of submachines, performing separate, well defined parts of more complex algorithms. This facilitates creating multi-level complex machines while keeping each particular machine simple and easy to manipulate. An example of a machine with submachines is the repeater machine presented in Figure 3 The submachines are placed within the area of their parent machines.

The repeater in the example, performed two independent runs of 2-fold cross-validation (CV). It has generated two distributors (one for each CV cycle) and four test schemes (two per CV cycle). The CV distributor outputs are two training sets and two test sets—the first elements go to the inputs of the first test

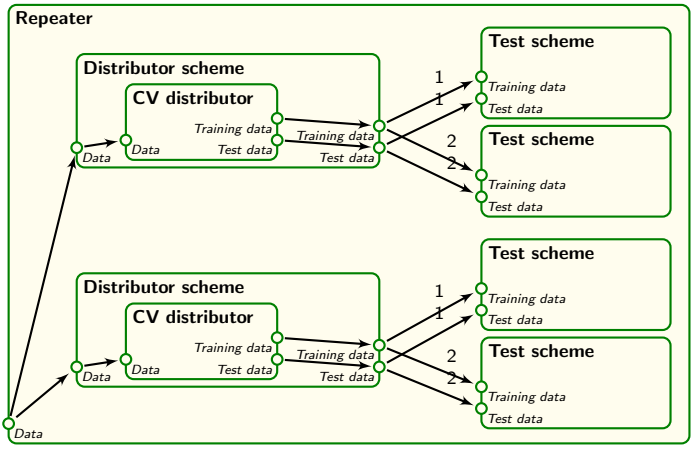


Fig. 3. Run time view of Repeater machine configured to perform twice 2-fold CV. Test schemes are simplified for clearer view—in fact each one contains a scenario to be repeated within the CV, for example the one of Figure 4.

scheme and the second elements to the second scheme. In this example the repeater machine has 6 submachines, each having further submachines.

3.1 Schemes and Machine Configuration Templates

When configuring complex machines like the repeater in Figure 3, it is important to be provided with simple tools for machine hierarchy construction. To be properly defined, the repeater needs definitions of two schemes: one defining the distributor (i.e. the machine providing training and test data for each fold) and one to specify the test performed in each fold (Figure 4 shows an example test scenario, where kNN (k-nearest neighbors [25]) and SVM (support vector

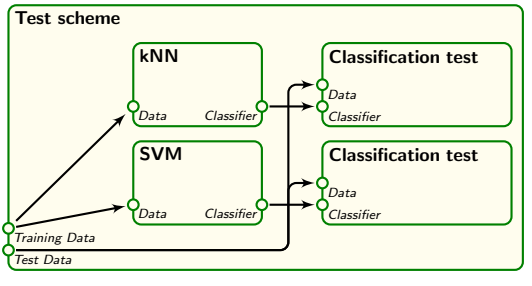


Fig. 4. A test scheme example.

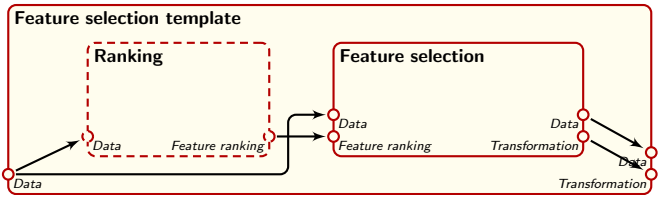


Fig. 5. Feature selection template.

machines [26,27]) classifiers are, in parallel, created, trained on the training data and tested on the test data). At run time the repeater creates and runs the distributor scheme, and then creates and runs a number of test schemes with inputs bound with subsequent data series exhibited as distributor scheme outputs.

The repeater's children are *schemes*, i.e. machines especially designated for constructing machine hierarchies. Schemes do not perform any advanced processes for themselves, but just run graphs of their children (request creation of the children and wait till all the children are ready to eventually exhibit their outputs to other machines).

A machine configuration with empty scheme (or empty schemes) as child machine configuration (scheme which contain information only about types and names of inputs and outputs) is called a *machine template* (or more precisely a *machine configuration template*). Machine templates are very advantageous in meta-learning, since they facilitate definition of general learning strategies that can be filled with different elements to test a number of similar components in similar circumstances. Empty scheme may be filled by one or more configurations. The types of empty scheme inputs and outputs defines general type of role of scheme. For example, the feature selection template, presented in Figure 5, may be very useful for testing different feature ranking methods from the point of view of their eligibility for feature selection tasks. The dashed box represents a *placeholder* (empty scheme with defined types and names of inputs and outputs) for a ranking. In case of ranking the scheme has single input with data and single output with information about ranking of features. After replacing the Ranking placeholder by a compatible configuration the whole construct can be created and run or put into another configuration for more complex experiments.

3.2 Query System

Standardization of machine results management makes the technical aspects of results analysis completely independent of the specifics of particular machines. Therefore, we have designed the *results repository*, where the information may be exposed in one of three standard ways:

– the machine itself may deposit its results to the repository (e.g. classification test machines put the value of accuracy into the repository),

– parent machines may comment their submachines (e.g. repeater machines comment their subschemes with the repetition and fold indices),
– special commentator objects may comment machines at any time (this subject is beyond the scope of this article, so we do not describe it in more detail).

The information within the repository has a form of label-value mappings.

Putting the results into the repositories is advantageous also from the perspective of memory usage. Machines can be discarded from memory when no other machine needs their outputs, while the results and comments repositories (which should be filled with moderation) stay in memory and are available for further analysis.

The information can be accessed directly (it can be called a low level access) or by running a query (definitely recommended) to collect the necessary information from a machine subtree.

Queries facilitate collection and analysis of the results of machines in the project, it is not necessary to know the internals of the particular machines. It is sufficient to know the labels of the values deposited to the repository.

A query is defined by:

– the root machine of the query search,
– a *qualifier* i.e. a filtering object—the one that decides whether an item corresponding to a machine in the tree, is added to the result series or not,
– a *labeler* i.e. the object collecting the results objects that describe a machine qualified to the result series.

Running a query means performing a search through the tree structure of submachines of the root machine and collecting a dictionary of label-value mappings (the task of the labeler) for each tree node qualified by the qualifier.

For example, consider a repeater machine producing run time hierarchy of submachines as in Figure 3 with test schemes as in Figure 4. After the repeater is finished, its parent wants to collect all the accuracies of SVM machines, so it runs the following code:

```
1 Query.Series results = Query(repeaterCapsule,
2     new Query.Qualifier.RootSubconfig(1, 3),
3     new Query.Labeler.FixedLabelList("Accuracy"));
```

The method Query takes three parameters: the first repeaterCapsule is the result of the CreateChild method which had to be called to create the repeater, the second defines the qualifier and the third—the labeler. The qualifier RootSubconfig selects the submachines, that were generated from the subconfiguration of repeater corresponding to path "1, 3". The two-element path means that the source configuration is the subconfiguration 3 of subconfiguration 1 of the repeater. The subconfiguration 1 of the repeater is the configuration of the test scheme (0-based indices are used) and its subconfiguration 3 is the SVM Classification test. So the qualifier accepts all the machines generated on the basis of the configuration Classification test taking *Classifier* input from SVM machine. These are classification

tests, so they put Accuracy to the results repository. The labeler FixedLabelList of the example, simply describes each selected machine by the object put into the results repository with label Accuracy. Intemi provides a number of qualifiers and labelers to make formulating and running miscellaneous queries easy and efficient. As a result we obtain a series of four descriptions (descriptions of four nodes) containing mappings of the label Accuracy to the floating point value of the accuracy.

In practice we are usually interested in some derivatives of the collected descriptions, not in the result series being the output of the query. For this purpose, Intemi provides a number of *series transformations* and tools for easy creation of new series transformations. The transformations get a number of series objects and return another series object. One of the basic transformations is the BasicStatistics which transforms a series into a single item series containing the information about minimum, mean, maximum values and standard deviation. More advanced predefined transformations perform results grouping, ungrouping, mapping group elements and calculate statistics for hypotheses testing including t-test, Wilcoxon test, McNemar test etc.

For the purpose of meta-learning we have encapsulated machine qualifier, labeler and final series transformation into the class of QueryDefinition. Running a query defined by the three components, in fact means collection of the results according to the qualifier and the labeler, and transforming the collected series with the transformation to obtain the final result of interest.

3.3 Task Spooling

Before a machine request is fulfilled, i.e. the requested machine is created and its process run, the request context is equipped with proper task information and the task is pushed to the *task spooler*, where it waits for its turn and for a free processing thread. The task spooler of our system is not a simple standard queue. To optimize the efficiency of task running, we have introduced a system of hierarchical priorities. Each parent machine can assign priorities to its children, so that they can be run in proper order. It prevents from starting many unrelated tasks in parallel i.e. from too large consumption of memory and computation time. As a result, the spooler has the form of tree containing nodes with priorities.

Intemi environment delegates machine creation and running machine processes to separate task management modules. Each project can subscribe to services of any number of *task managers* executed either on local or remote computers (see Figure 6). Moreover subscribing and unsubscribing to task managers may be performed at project run time, so the CPU power can be assigned dynamically. Each task manager serves the computational power to any number of projects. Task managers run a number of threads in parallel to make all the CPU power available to the projects. Each project and each task manager, presented in Figure 6, may be executed on different computer.

A task thread runs machine processes one by one. When one task is finished, the thread queries for another task to run. If a task goes into waiting mode (a machine requests some submachines and waits for them) the task manager is

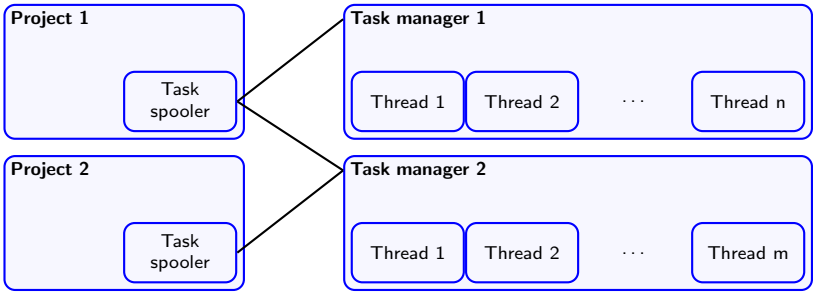


Fig. 6. Two projects and two task managers.

informed about it and starts another task thread, to keep the number of truly running processes constant.

Machine tasks may need information from another machines of the project (for example input providers or submachines). In the case of remote task managers, a *project proxy* is created to supply the necessary project machines to the remote computer. Only the necessary data is marshaled, to optimize the information flow.

Naturally, all the operations are conducted automatically by the system. The only duty of a project author is to subscribe to and unsubscribe from task manager services—each requires just a single method call.

3.4 Machine Unification and Machine Cache

In advanced data mining project, it is inevitable that a machine with the same configuration and inputs is requested twice or even more times. It would not be right, if an intelligent data analysis system were running the same adaptive process more than once and kept two equivalent models in memory. Therefore, Intemi introduced machine contexts as formal objects separate from proper machines. Different contexts may request for the same machine, and may share the machine.

Constructed machines are stored in *machine cache*, where they can be kept even after getting removed from the project. When another request for the same machine occurs, it can be restored from the cache and directly substituted instead of repeated creation and running. To achieve this, each machine request is passed to the machine cache, where the new machine configuration and context are compared to those, deposited in the cache. If unification is successful, the machine cache provides the machine for substitution.

Another unification possibility occurs between the requests pushed to the task spooler. Intemi controls unification also at this level preventing from building the same machine twice.

An illustrative example of machine unification advantages can be a project testing different feature ranking methods. Table 1 shows feature rankings obtained

Table 1. Feature rankings for UCI Wisconsin breast cancer data.

Ranking method	Feature ranking
F-score	6 3 2 7 1 8 4 5 9
Correlation coefficient	3 2 6 7 1 8 4 5 9
Information theory	2 3 6 7 5 8 1 4 9
SVM	6 1 3 7 9 4 8 5 2
Decision tree (DT), Gini	2 6 8 1 5 4 7 3 9
DT, information gain	2 6 1 7 3 4 8 5 9
DT, information gain ratio	2 6 1 5 7 4 3 8 9
DT, SSV	2 6 1 8 7 4 5 3 9

for Wisconsin breast cancer data from the UCI repository with eight different methods: three based on indices estimating feature's eligibility for target prediction (F-score, correlation coefficient and entropy based mutual information index), one based on internals of trained SVM model and four based on decision trees using different split criteria (Gini index, information gain, information gain ratio and SSV [28]). To test a classifier on all sets of top-ranked features for each of the eight rankings, we would need to perform 72 tests, if we did not control subsets identity. An analysis of the 72 subsets brings a conclusion that there are only 37 different sets of top-ranked features, so we can avoid 35 repeated calculations.

In simple approaches such repetitions can be easily avoided by proper machine process implementation, but in complex projects, it would be very difficult to foresee all the redundancies (especially in very complicated meta-learning projects), so Intemi resolves the problem at the system level by means of machine unification.

To read more about Intemi please see [29,30,31,32,33].

4 Parameter Search Machine

One of the very important features of meta-learning is the facility of optimization of given machine configuration parameters, for example, the number of features to be selected from a data table. Parameter optimization may be embedded in the algorithm of given learning machine or the optimization may be performed outside of the learning machine. In some cases, from computational point of view, it is more advantageous to realize embedded optimization—by implementing optimization inside given learning process. However it is rather rare and in most cases the optimization must not be realized in the embedded form without loss of complexity and time of optimization.

As presented in previous sections, in so general purpose system, the machine devoted to optimize parameters of machine configurations must be also very general and ready to realize different search strategies and must be open to extensions by new search strategies in feature. The new search strategies are

realized as new modules for optimization machine and extend the possibilities of searching in new directions.

Presented *parameters search machine* (PSM) can optimize any elements of machine configuration[1]. The optimization process may optimize any elements of configuration and any element of subconfigurations (including subsubconfigurations etc.). Also subelements of objects in (sub-)configurations can be optimized. This is important, because so often, machines are very complex and their configurations are complex too, then the elements of optimization process are sometimes deeply nested in complex structures. Thus, a mechanism of pointing such elements is mandatory in flexible optimization definition. The search and optimization strategies are realized as separate modules which realize appropriate functionality. Because of that, the search strategies are ready to provide optimization of any kind of elements (of configurations) even the abstract (amorphic) structures can be optimized. Such structures may be completely unknown for PSM, but given search strategy knows what to do with objects of given structure. Search strategies provides optimization of a single or a set of configuration elements during the optimization process. It is important to notice that in so general system, sometimes even changing simple scalar parameter, the behavior of machine may change very significantly.

The PSM uses test procedures to estimate the objective functions as the quality test to help the search strategy undertake the next steps. The definition of such test must be realized in very open way to enable using of PSM to optimize machines of different kinds and in different ways.

Such general behavior of MPS was obtained by flexible definition of configuration of MPS combined with general optimization procedure presented below. Let's start the description of configuration of MPS. The MPS configuration consists of (not all elements are obligatory):

Test Template: It determines the type of test measuring influence of the parameters being optimized to the quality of the resulting model. The test template may be defined in many different ways and may be defined for different types of problems. The most typical test used in such case for the classification problems is the cross-validation test of chosen classifier (sometimes classifier is defined as complex machine). This is a mandatory part of configuration.

Path to the final machine: Variable PathToFinalConfigurationInTemplate in the code presented below, defines a place (a path) of subconfiguration in the test template, which will become the final configuration machine of MPS. For example, in the case of optimizing a classifier this path will point to the classifier in the test template and after optimization process based on configuration pointed by this path in final configuration, the final configuration of classifier will be extracted, and finally the MPS will *play the role* of the classifier (which means that this classifier will be an output of the MPS on finish). This parameter is not obligatory. If this parameter is not defined in

[1] It is possible to optimize single or several parameters during optimization, sequentially or in parallel, depending on used search strategy.

configuration of MPS, the MPS will return just the final (optimized) configuration as the result of the optimization procedure.

Query definition: For each machine configuration created in the optimization procedure (see below) the quality test must be computed to advise further process of optimization and final choice of configuration parameters. The query is realized exactly as it was presented in Section 3.2. The test template (for example cross-validation) will produce several submachines with some results like accuracy, which describe the quality of each subsequent test. The query definition specifies which machines (the machine qualifier) have to be asked for results and which labels (the machine labeler) provides interesting values to calculate the final quantity. As a result, series of appropriate values are constructed. The last element of query definition defines how to calculate the result-quantity (single real value) on the basis of previously obtained series of values. Query definition is obligatory in MPS configuration.

Scenario or ConfigPathToGetScenario: These are two alternative ways to define the scenario (i.e. the strategy) of parameter(-s) search and optimization. Either of them must be defined. If the scenario is defined, then it is a direct scenario is defined directly. If the path is defined, it points the configuration which is expected to support auto-reading of the default scenario for this type of configuration.

The *scenario* defined within the configuration of PSM determines the course of the optimization process. Our system contains a number of predefined scenarios and new ones can easily be implemented. The main, obligatory functionalities of the scenarios are:

SetOptimizationItems: each scenario must specify which element(-s) will be optimized. The items will be adjusted and observed in the optimization procedure. This functionality is used at the configuration time of the scenario, not in the optimization time.

NextConfiguration: subsequent calls return the configurations to be tested. The PSM, inside the main loop, calls it to generate a sequence of configurations. Each generated configuration is additionally commented by the scenario to enable further meta-reasoning or just to inform about the divergence between subsequent configurations. The method NextConfiguration returns a boolean value indicating whether a new configuration was provided or the scenario stopped the process of providing next configurations (compare line 8 of the code of MPS shown below).

RegisterObjective: scenarios may use the test results computed for generated configurations when generating next configurations. In such cases, for each configuration provided by the scenario, the MPS, after the learning process of such task, runs the test procedure, and the value of the quality test is passed back to the scenario (compare above comments on Query definition and line code 12) to inform the *optimization strategy* about the progress.

High flexibility of the elements of MPS, described above, enable creation of optimization algorithm in relatively simple way, as presented below.

```
4  function MetaParamSearch(TestTemplate, Scenario,
5      PathToFinalConfigurationInTemplate, QueryDefinition);
6      Scenario.Init(TestTemplate);
7      ListOfChangeInfo = {};
8      while (Scenario.NextConfiguration(config, changes))
9      {
10         t = StartTestTask(config);
11         qt = computeQualityTest(t, QueryDefinition);
12         Scenario.RegisterObjective(config, qt);
13         ListOfChangeInfo.Add(<qt, changes>);
14         RemoveTask(t);
15     }
16     if (defined PathToFinalConfigurationInTemplate)
17     {
18         confM = config.GetSubconfiguration(
19             PathToFinalConfigurationInTemplate);
20         c = CreateChildMachine(confM);
21         SetOutput(c);
22     }
23     return <config, ListOfChangeInfo>;
24 end
```

In line 6 the scenario is initialized with the configuration to be optimized. It is important to note that the starting point of the MPS is not a default configuration of given machine type but strict configuration (testing template with strict configuration of *adjustable* machine) which may be a product of another optimization. As a consequence, starting the same MPS machine with different optimized configurations may finish in different final configurations—for example tuning of the SVM with linear kernel or tuning of the SVM with Gaussian kernel finish in completely different states.

Every configuration change is *commented* by the chosen scenario. This is useful in further analysis of optimization results—see line 7.

The main loop of MPS (line 8) works as long as any new configuration is provided by the scenario. When NextConfiguration returns **false**, the variable config holds the final configuration of the optimization process and this configuration is returned by the last line of the MPS code. After a new configuration is provided by the scenario, a test task is started (see line 10) to perform the test procedure, for example the cross-validation test of a classifier. After the test procedure, the QueryDefinition is used to compute the quality test. The quality test may be any type of attractiveness measure (attractiveness ≡ reciprocal of error). For example the accuracy or negation of the mean squared error. The resulting quality is sent to the scenario, to inform it about the quality of the configuration adjustments (line 12). Additionally, the resulting quality value and the comments on the scenario's adjustments are added to the queue ListOfChangeInfo.

Finally the test task is removed and the loop continues.

When the PathToFinalConfigurationInTemplate is configured, the MPS builds the machine resulting from the winner configuration—see code lines 16–22. The type of the machine is not restricted in any way—it depends only on the configuration pointed by PathToFinalConfigurationInTemplate.

The MPS algorithm finishes by returning the final (the best) configuration and the comments about the optimization process.

4.1 Examples of Scenarios

Intemi provides a number of predefined scenarios. At any time, the system may be extended by new scenarios. Below, we present some simple, not nested scenarios and then, more complex, nested ones.

The most typical scenario, used for optimization of many learning machines' parameters, is the *StepScenario* based on quite simple idea to optimize single chosen element of a configuration. Such element is defined (not only in the case of that scenario) by two paths which constitute the scenario configuration. The first is the subconfiguration path, which goal is to define in which (sub-)configuration the optimization parameter lies (at any depth). When the path is empty, it points the main configuration. The second path is the property path, pointing (in configuration pointed by first path) to the property (or sub-sub-. . . property) which is the element to be optimized. The property path may not be empty (something must be pointed!). The search strategy is very simple, it generates configurations with the element set to values from a sequence defined by start value, step value and the number of steps. Step-type of StepScenario may be linear, logarithmic or exponential. This is very convenient because of different behaviors of configuration elements. This scenario may be used with real or integer value type. It may optimize, for example, the number of selected features (for feature selection machine) or the SVM's C parameter or the width of the Gaussian kernels.

The *SetScenario* is powerful in the cases, when the optimized parameter is not continuously valued. It can be used with every enumerable type (including real or integer). Because this scenario is type independent, the examples of using it are quite diverse. In the case of k nearest neighbors, the metric parameter may be optimized by setting the configuration of metric to each of the elements of a set of metrics, for example the set of Euclidean, Manhattan and Chebyshev metrics. In cases where a few fixed values should be checked as the values of given configuration element, the step scenario may be configured to the fixed set of values, for example: 2, 3, 5, 8, 13, 21. The determination of the optimization parameter is done in the same way as in the case of StepScenario: by two paths which point the subconfiguration and the subproperty.

SetAndSetSubScenario is, in some way, a more general version of the previous scenario. It also checks a set of enumerable values attached to given configuration element, but additionally it is able to start sub-scenario declared for all or selected values from the mentioned enumerable set. Configuration of this scenario consists of a set of pairs: $< value_k, scenario_k >$. For each pair, an element

of configuration is set to $value_1$ and the subscenario is started to optimize another element of configuration by $scenario_1$, if only $scenario_1$ is not null. The scenario can be used, for example, to chose metric (between some metric from a set as before) and, in the case of Minkovsky metric, additionally tune (via nested scenario) the parameter which represents the power in the Minkovsky metric.

Another very useful scenario is the *StackedScenario* which facilitates building a sequence or a combination of scenarios. This scenario may be used in one two modes. The first mode, the *sequence mode*, enables starting scenarios one by one, according to the order in the sequence. In the *grid mode* each value generated by the first scenario is tried against each value provided by second scenario and so on. For example, assume that we need to optimize the C parameter of the SVM and also the width of the Gaussian kernel. Using the sequence mode it is possible to optimize the C first, and after that, the width or in the reversed order. Also, it is possible to optimize the C first, then the width and then again the C, and so on. In grid mode, every pair of parameters will be checked (every C against every width). In general, the grid may be composed of more than two scenarios, then such scenario will try each triple, each four and so on. Note that the stacked scenario may be composed of any scenarios, including stacked scenario or set scenario, however too complex optimizations are not recommended because of growing complexity.

SimpleMaximumScenario may be used to look for maximum, assuming the single maximum problem (or looking for local minimum), observing the quality test returned to the scenario. When using this scenario it is recommended to put a limit on the number of steps beside the limit on progress of optimization. Of course this scenario may be extended in several simple ways to more sophisticated versions.

4.2 Auto-scenario

If the optimization of given machine or its single chosen parameter is to be performed typically it is done, in most cases, in the same way. This suggests that it is not necessary (and not recommended) to rediscover the method of parameters tuning in each optimization of given machine. The auto-scenarios is the idea to "dedicate" the behavior of optimization for configuration parameters and whole learning machines. In the presented system it is done using the scenario-comment attributes which define the default way of optimization. Scenario-comments are used with configuration elements and also with the whole configurations of learning machines. The difference is that, for the whole machine, it is responsible for optimization of the whole machine configuration in possibly best way, while for single configuration elements it is responsible to optimize the pointed element of configuration without any change of other elements.

Such idea of auto-scenarios is very convenient, because it simplifies the process of optimization without loss of quality. Nobody is obliged to know the optimal way of parameter optimization for all machines. The scenario comments compose a brilliant base of meta-knowledge created by experts—using auto-scenarios, meta-learning does not have to rediscover the optimal way of optimization of

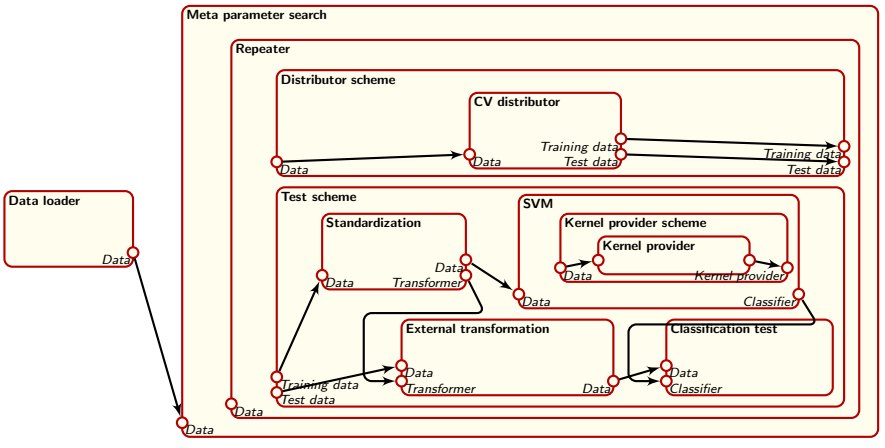


Fig. 7. A meta parameter search project configuration.

each particular machine. In meta-learning auto-scenarios are used together with machine generators to auto-generate optimization strategies of chosen machine configuration. It is also possible to define a few default optimization ways for one machine. This enables using a simpler and a deeper optimization scenarios and depending on the circumstances, one may be chosen before the other. Auto-scenarios can be attached to any machines, both the simple ones and the complex hierarchical ones.

4.3 Parameter Search and Machine Unification

As mentioned in the preceding section, machine unification is especially advantageous in meta-learning. Even one of the simplest meta-learning approaches, a simple meta parameter search, is a good example. Imagine a project configuration depicted in Figure 7, where the MPS machine is designed to repeat 5 times 2-fold CV of the test template scenario for different values of the SVM C and kernel σ parameters. Testing the C parameter within the set $\{2^{-12}, 2^{-10}, \ldots, 2^{2}\}$ and σ within $\{2^{-1}, 2^{1}, \ldots, 2^{11}\}$ we need to perform the whole 5×2 CV process 8×7 times. As enumerated in Table 2, such a project contains (logically) 4538 machines. Thanks to the unification system, only 1928 different machines are created saving both time and memory. The savings are possible, because we perform exactly the same CV many times, so the data sets can be shared and also the SVM machine is built many times with different C parameters and the same kernel σ, which facilitates sharing the kernel tables by quite large number of SVM machines.

Table 2. Numbers of machines that exist in the project logically and physically.

Machine	logical count	physical count
Data loader	1	1
Meta parameter search	1	1
Repeater	56	56
Distributor scheme	280	5
CV distributor	280	5
Test scheme	560	560
Standardization	560	10
External transformation	560	10
SVM	560	560
Kernel provider scheme	560	80
Kernel provider	560	80
Classification test	560	560
Sum	4538	1928

5 Meta-Learning Algorithm Elements

This section provides description of all the parts of the presented meta-learning algorithm. First, machine configuration generators, which represent and define the functional form of meta-learning search space, are described. Next, a clear distinction between the configuration of the algorithm and the main part of the algorithm is stated. After that, other elements of the main part of the algorithm will be presented.

5.1 Machine Configuration Generators and Generators Flow

In the simplest way, the machine space browsed by meta-learning may be realized as a set of machine configurations. However such solution is not flexible and strongly limits the meta-learning possibilities to fixed space. To overcome this disadvantage the *machine configuration generators* (MCG) are introduced.

The main goal of MCGs is to provide/produce machine configurations. Each meta-learning may use several machine configuration generators nested in a **generators flow** (a graph of generators). Each MCG may base on different meta-knowledge, may reveal different behavior, which in particular may even change in time (during the meta-learning progress). The simplest generators flow is presented in Figure 8.

Each generator provides machine configurations through its output. Generators may also have one or more inputs, which can be connected to outputs of other generators, similarly to machines and their inputs and outputs. If the output of generator A is connected to an input number 3 of generator B, then every machine configuration generated by the generator A will be provided to the input 3 of the generator B. If a generator output is attached to more inputs, then each of the inputs will receive the output machine configurations.

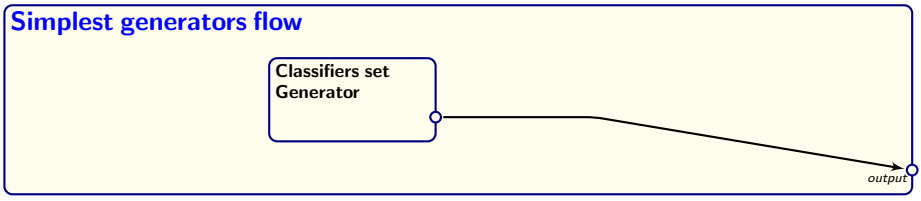


Fig. 8. Example of simplest generator flow.

The inputs-outputs connections between generators, compose a generators flow graph, which must be a directed acyclic graph. A cycle would result in infinite number of machine configurations. Some of the outputs of generators can be selected as the output of the generators flow—the providers of configurations to the meta-learning. In the run time of the meta-learning algorithm, the configurations returned by the generators flow, are transported to a special heap, before the configured machines are tested.

The streams of configurations provided by generators may be classified as fixed or non-fixed. *Fixed* means that the generator depends only on its inputs and configuration. The non-fixed generators depend also on the learning progress (see the advanced generators below).

Another important feature of generators is that, when a machine configuration is provided by a generator, information about the *origin* of configuration and some comments about it are attached to the machine configuration. This is important for further meta-reasoning. It may be useful to know information on how the configuration was created in further meta-reasoning.

In general, generators behavior is not limited except the fact, that they should provide a finite series of configurations. Below we describe some generators examples.

Set-based Generators. The simplest and very useful machine generator is based on the idea to provide just an arbitrary sequence of machine configurations (thus the name *set-based generator*). Usually, it is convenient to have a few set-base generators in single generators flow—a single set per a group of machines of similar functionality like feature rankings or classifiers. An example of set-base generator providing classifier machine configurations was presented in Figure 8.

Template-based Generators. Template-based generators are used to provide complex configurations based on given machine configuration template (described in Section 3.1) and generators connected to them. For example, if a meta-learner is to search through combinations of different data transformers and kNN classifier, it can easily do it with a template-based generator. The combinations may be defined by a machine template with a placeholder for data transformer and fixed kNN classifier. Such a template-based generator may be

connected to a set-based generator with a sequence of data transformations, which would make the template-based generator provide a sequence of complex configurations resulting from replacing the placeholder with subsequent data transformation configurations. Please, note that, in the example, the machine template is to play the role of a classifier. Because of that, we can use the Transform and Classify machine template shown in Figure 9 on the left. This machine learning process starts with learning the data transformation first and then the classifier.

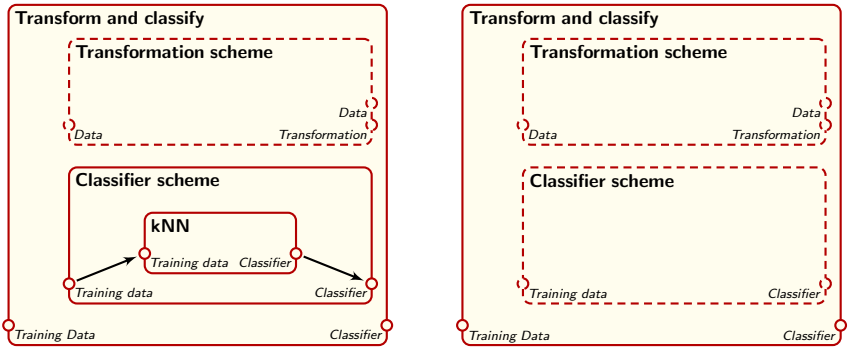


Fig. 9. A Transform and classify machine configuration template. LEFT: placeholder for transformer and fixed classifier (kNN). RIGHT: two placeholders, one for the transformer and one for the classifier

The generator's template may contain more than one placeholder. In such a case the generator needs more than one input. The number of inputs must be equal to the number of placeholders. The role of a placeholder is defined by its inputs and outputs declarations. So, it may be a classifier, approximator, data transformer, ranking, committee, etc. Of course the placeholder may be filled with complex machine configuration too.

Replacing the kNN from the previous example by a classifier placeholder (compare Figure 9 on the right), we obtain a template that may be used to configure a generator with two inputs. One designated for a stream of transformers, and the other one for a stream of classifiers.

The template-base generator can be configured in one of two modes: *one-to-one* or *all-to-all*. In the case of the example considered above, mode 'one-to-one' makes the template-based generator get one transformer and one classifier from appropriate streams and put them into the two placeholders to provide a result configuration. The generator repeats this process as long as both streams are not empty. In 'all-to-all' mode the template-based generator combines each transformer from the stream of transformers with each classifier from the classifiers stream to produce result configurations.

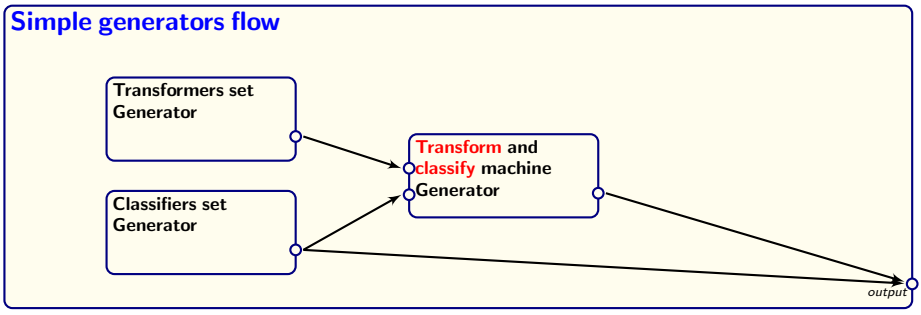


Fig. 10. A simple generator flow.

Figure 10 presents a simple example of using two set-based generators and one template-based generator. The set-based generators provide transformers and classifiers to the two inputs of the template-based generator, which puts the configurations coming to its inputs to the template providing fully specified configurations. Different mixtures of transformations and classifiers are provided as the output, depending on the mode of the generator: one-to-one or all-to-all. Please, note that the generator's output gets configuration for both the set-based classifiers generator and the template-based generator, so it will provide both the classifiers taken directly from the proper set and the classifiers preceded by the declared transformers.

Another interesting example of a template-based generator is an instance using a template of ParamSearch machine configuration (the ParamSearch machine is described in Section 4). The template containing test scheme with a placeholder for a classifier is very useful here. When the input of the template-based generator is connected to a stream of classifiers, the classifiers will fill the placeholder, producing new configurations of the ParamSearch machine. Configuring the ParamSearch machine to use the *auto-scenario* option makes the meta-learner receive configurations of ParamSearch to realize the auto-scenario for given classifier. It means that such a generator will provide configurations for selected classifiers to auto-optimize their parameters.

An more complex generator flow using the generators presented above, including template-based generator with ParamSearch machine is presented in Figure 11. This generator flow contains three set-based generators, which provide classifiers, transformers and ranking configurations to other (template-based) generators. Please, see that the classifier generator sends its output configurations directly to the generator flow output and additionally to three template-based generators: two combining transformations with classifiers and the ParamSearch generator (MPS for Classifiers Generator). It means that each classifier configuration will be send to the meta-learning heap and additionally to other generators

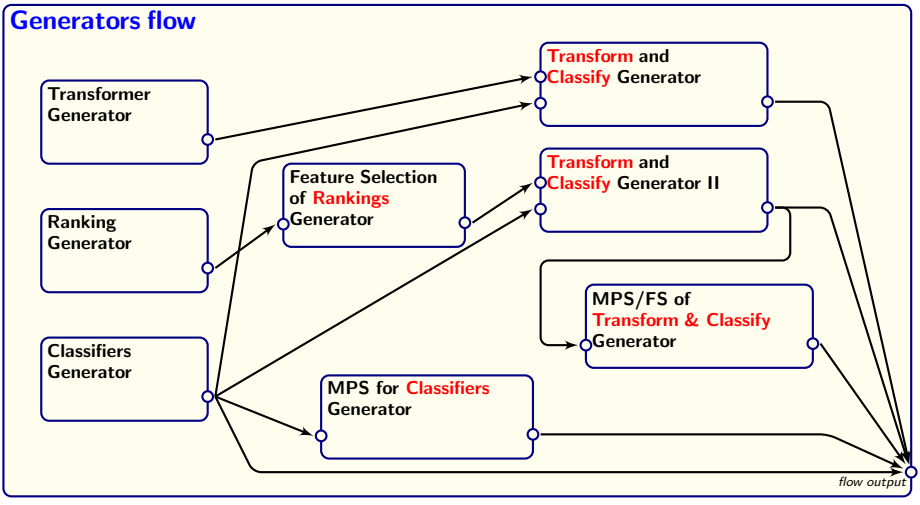


Fig. 11. Example of generators flow.

to be nested in other configurations (generated by the Transform and Classify and the MPS generators).

The two Transform and Classify generators combine different transformations with the classifiers obtained from the Classifiers Generator. The configurations of transformation machines are received by proper inputs. It is easy to see, that the first Transform and Classify generator uses the transformations output by the Transformer Generator while the second one receives configurations from another template-based generator which generated feature selection configurations with the use of different ranking algorithms received through the output-input connection with the Ranking Generator. The combinations generated by the two Transform and Classify generators are also sent to the meta-learning heap through the output of the generators flow.

Additionally, the Transform and Classify Generator II sends its output configurations to the ParamSearch generator (MPS/FS of Transform & Classify generator). This generator produces ParamSearch configurations, where the number of features is optimized for configurations produced by the Transform and Classify Generator II. The output of the ParamSearch generator is passed to the output of the generators flow too.

In such a scheme, a variety of configurations may be obtained in a simple way. The control of template-based generators is exactly convergent with the needs of meta-search processes.

There are no a priori limits on the number of generators and connections used in generator flows. Each generator flow may use any number of any kind of generators. In some cases it is fruitful to separate groups of classifiers in independent set-based generators to simplify connection paths in the generators

flow graph. The same conclusion is valid also for transformations which grouped into separate sets may facilitate more flexibility, for example when some machines have to be preceded by a discretization transformation or should be used with (or without) proper filter transformations.

Advanced Generators. Advanced generators are currently under development. Their main advantage, over the generators mentioned above, is that they can make use of additional meta-knowledge. A meta-knowledge, including the experts' knowledge, may be embedded in a generator for more intelligent filling of placeholders in the case of template-based generators. For example, generators may *know* that given classifier needs only continuous or discrete features.

Advanced generators are informed each time a test task is finished and analyzed. The generators may read the results computed within the test task and the current machine configuration ranking (compare code line 68 in Section 5.5). The strategy enables providing machine configurations derived from an observation of the meta-search progress. Advanced generators can learn from meta-knowledge, half-independently of the heart of the meta-learning algorithm, and build their own specialized meta-knowledge. It is very important because the heart of meta-learning algorithm can not (and should not) be responsible for specific aspects of different kind of machines or other *localized* type of knowledge.

5.2 Configuring Meta-Learning Problem

It is crucial to see the meta-learning not as a wizard of everything but as a wizard of consistently defined meta-learning task. The meta-learning does not discover the goal itself. The meta-learning does not discover the constraints on the search space. Although the search space may be adjusted by meta-learning. To define a meta-learning algorithm that can be used for different kinds of problems and that can be adjusted not by reimplementing but through configuration changes, it must be designed very flexibly.

Stating of Functional Searching Space. A fundamental aspect of every meta-learning is determination of the search space of learning machines. The simplest solution is just a set of configurations of learning machines. Although it is acceptable that the configurations can be modified within the meta-learning process, a single set of configurations still offers strongly limited possibilities. A functional form of configurations of learning machines is much more flexible. This feature is realized by the idea of *machine configuration generators* and the *machine generators flow* described in Section 5.1. Using the idea of machine configuration generators and their flows, defining the search space for meta-learning is much more powerful and intuitive. One of the very important features is, that generators may describe non-fixed spaces of configurations. This means that the space may continuously evolve during the meta-search procedure. The graph of generators may be seen as continuously working factory of machine configurations. Defining meta-search space by means of machines generators fulfill all or almost all the needs of applications to miscellaneous types of problems.

Defining the Goal of Meta-learning. Another very important aspect of configuration of a meta-learner is the possibility of strict and flexible definition of the goal. In the context of our general view of MLA (please look back at Figure 1), we focus on the step *start some test tasks*. Creation of test tasks is not learning a machine according to the chosen configuration, but preparation of the *test procedure* of the chosen configuration. From technical point of view, it is nothing else than building another complex machine configuration, performing the test procedure i.e. validating the chosen machine configuration. This is why the definition of the test procedure may be encapsulated in a functional form of *machine configuration template* (compare Section 3.1) with nested placeholder for chosen configuration to be validated. The meta-learning takes the test machine template, fills the placeholder with the chosen machine configuration and starts the test task. For example, when the MLA is to solve a classification problem, a classifier test must be defined as the *test machine template*—in particular it may be the commonly used repeated cross-validation test.

After normal finish of a test task the MLA needs to *evaluate the results* i.e. to determine a quantity estimating, in some way, the quality of the test results. When the test task is finished, the MLA has the possibility to explore it. The test task has a form of a tree of machines (compare Figure 3). In our approach, to compute the *quality of the results* it is necessary to define:

- which nodes of the machine tree preserve the values with the source information for computing the *quality-test*,
- which values at given nodes are the source of required information,
- how to use the values collected from the tree to measure the quality of given test.

And this is exactly the same information type as those presented in Section 3.2, where to define a *query* we had to define: the *root machine* (the root of the tree to search—the test machine in this case), *qualifier* which selects the nodes of the tree containing the information, the *labeler* which describes the selected nodes with labels corresponding to the desired values and the *series transformation*, which determines the final value of the quality estimate. The example of a query, presented in Section 3.2, clearly shows that it is a simple and minimal mean to compute qualities of given test tasks. Thanks to such a form of the quality test definition, the MLA can be simply adapted to very different types of quality tests, independently from the problem type.

Defining the Stop Condition. Another important part of the MLA configuration is the specification, when to stop. The stop condition can be defined in several ways, for example:

- stop after given *real time limit*,
- stop when given *CPU time limit* is achieved,
- stop after given *real time limit* or no important progress in the quality test is observed for given amount of time (defined by progress threshold and progress tail length).

Of course the meta-learning may be stopped in other ways—by implementing another stop-function. An interesting extension may be intelligent stop functions, i.e. functions, which make use of meta-knowledge in their decisions.

Defining the Attractiveness Module. If the attractiveness module is defined, then the corrections to the complexity of machines will be applied, to give additional control of the order of tests in the machine space exploration. It is described in more detail in Section 6.1.

Initial Meta-knowledge. A meta-knowledge is almost always passed to the MLA by means of the configuration items such as the machine configuration generators, the attractiveness module or the complexity computation module. It is very important include appropriate meta-information at the initialization stage of the search process, since it may have very important influence on the results of the meta-learning search.

Summing up the above section, beside the attractiveness module, all the configuration elements, explained above, compose the minimal set of information to realize the meta-learning search. It is impossible to search in undefined space or search without strictly defined goal. All the parts of the configuration can be defined in very flexible way. At the same time, defining the search space by the machine configuration generators, specifying the goal by the query system, determining the stop condition, are also quite simple and may be easily extended for more sophisticated tasks.

5.3 Initialization of Meta-Learning Algorithm

The general scheme of Figure 1, can be written in a simple meta-code as:

```
25 procedure Meta_learning;
26     initialization;
27     while(stopCondition != true)
28     {
29         start tasks if possible
30         wait for finished task or break delayed task
31         analyze finished tasks
32     }
33     finalize;
34 end
```

At the initialization step, the meta-learning is set up in accordance to the configuration. Among others, the goal (testing machine template and query definition) and search space and the stop-condition of meta-learning are defined. The machine search space is determined by machine configuration generators embedded in a generator flow. See lines 36–41 in the code below.

The machinesRanking serves as the ranking of tested machines according to the quality test results obtained by applying the queryDefinition to the test

task. machinesRanking keeps information about the quality of each configuration, and about their origin (it is necessary to understand how it was evolving). The heap named machinesHeap (line 44) will keep machine configurations organized according the complexity i.e. will decide about the order of test tasks starting. The methods of complexity calculation are explained in Section 6. The machineGeneratorForTestTemplate is constructed as a machine generator based on the testing template. The goal of using this generator is to nest each machine configuration provided by the generators flow inside the testing machine template (see line 46) which is then passed through the machineHeap to start, and later to test the quality of provided machine. This is why machineGeneratorForTestTemplate has to be connected to the machinesHeap (line 48) and in previous line machine generators flow made connection to the machineGeneratorForTestTemplate.

```
35 procedure Initialization;
36     read configuration of ML and
37     set machineGeneratorsFlow
38     set testingTemplate
39     set queryDefinition
40     set stopCondition
41     set attractivenessModule
42     machinesRanking = {};
43     priority = 0;
44     machinesHeap = {};
45     machineGeneratorForTestTemplate =
46         MachineGenerator(testingTemplate);
47     machineGeneratorsFlow.ConnectTo(machineGeneratorForTestTemplate);
48     machineGeneratorForTestTemplate.ConnectTo(machinesHeap);
49 end
```

5.4 Test Tasks Starting

Candidate machine configurations are passed through a heap structure (machineHeap), from which they come out in appropriate order, reflecting machine complexity. The heap and complexity issues are addressed in detail, in Section 6. According to the order decided within the heap, procedure *start_tasks_if_possible*, sketched below, starts the simplest machines first and than more and more complex ones.

```
50 procedure startTasksIfPossible;
51     while (¬ machinesHeap.Empty() ∧ ¬ mlTaskSpooler.Full())
52     {
53         <mc, cmplx> = machinesHeap.ExtractMinimum();
54         timeLimit = τ · cmplx.time / cmplx.q
55         mlTaskSpooler.Add(mc, limiter(timeLimit), priority−−);
56     }
57 end
```

Tasks are taken from the machinesHeap, so when it is empty, no task can be started. Additionally, the task-spooler of meta-learning must not be full. MLAs use the task spooler described in Section 3.3 but via additional spooler which controls the width of the beam of tasks waiting for run.

If the conditions to start a task (line 51 of the code) are satisfied, then a pair of machine configuration mc and its corresponding complexity description cmplx is extracted from machinesHeap (see line 53).

The complexity is approximated for given configuration (see Section 6 for details). Since it is only an approximation, the meta-learning algorithm must be ready for cases when this approximation is not accurate or even the test task is not going to finish (according to the halting problem or problems with convergence of learning). To bypass the halting problem and the problem of (the possibility of) inaccurate approximation, each test task has its own time limit for running. After the assigned time limit the task is aborted. In line 54 of the code, the time limit is set up according to predicted time consumption (cmplx.time) of the test task and current reliability of the machine (cmplx.q). The initial value of the reliability is the same (equal to 1) for all the machines, and when a test task uses more time than the assigned time limit, the reliability is decreased (it can be seen in the code and its discussion presented in Section 5.5). τ is a constant (in our experiments equal to 2) to protect against too early test task braking.

The time is calculated in universal seconds, to make time measurements independent of the type of computer on which the task is computed. The time in universal seconds is obtained by multiplication of the real CPU time by a factor reflecting the comparison of the CPU power with a reference computer. It is especially important when a cluster of different computers is used.

Each test task is assigned a priority level. The MLAs use the priorities to favor the tasks that were started earlier (to make them finish earlier). Therefore the tasks of the smallest complexity are finished as first. Another reason of using the priority (see code line 55) is to inform the task spooler (compare Section 3.3) to favor not only the queued task but also each child machine. This is very important, because it saves memory resources. In the case of really complex machines which MLAs have to deal with, it is crucial.

The *while* loop in line 51 saturates the task spooler. This concept works in harmony with the priority system, yielding a rational usage of memory and CPU resources.

It is a good place to point out, that even if the generators flow provides a test task which has already been provided earlier, it will not be calculated the next time. Thanks to the unification engine, described in Section 3.4, the machine cache keeps all the computed solutions (if not directly in RAM memory, then in a disc cache). Of course, it does not mean that the generators flow should not care for the diversity of the test tasks configurations. The machine cache is also important to save time when any sub-task is requested repeatedly.

5.5 Analysis of Finished Tasks

After starting apropriate number of tasks, the MLA is waiting for a task to finish (compare the first code in Section 5.3). A task may finish normally (including termination by an exception) or halted by time-limiter (because of exceeding the time limit).

```
58 procedure analyzeFinishedTasks;
59     foreach (t in mlTaskSpooler.finishedTasks)
60     {
61         mc = t.configuration;
62         if (t.status = finished_normally)
63         {
64             qt = computeQualityTest(t, queryDefinition);
65             machinesRanking.Add(qt, mc);
66             if(attractivenessModule is defined)
67                 attractivenessModule.Analyze(t, qt, machinesRanking);
68             machineGeneratorsFlow.Analyze(t, qt, machinesRanking);
69         }
70         else // task broken by limiter
71         {
72             cmplx = mc.cmplx;
73             cmplx.q = cmplx.q / 4;
74             machinesHeap.Quarantine(mc);
75         }
76         mlTaskSpooler.RemoveTask(t);
77     }
78 end
```

The procedure runs in a loop, to serve all the finished tasks as soon as possible (also those finished while serving other tasks.

When the task is finished normally, the quality test is computed basing on the test task results (see line 64) extracted from the project with the query defined by queryDefinition. As a result a quantity qt is obtained. The machine information is added to the machines ranking (machinesRank) as a pair of quality test qt and machine configuration mc.

Later, if the attractiveness module is defined (see lines 66–67), it gets the possibility to analyze the new results and, in consequence, may change the attractiveness part of machines complexities. In this way, the MLA may change the complexity of machines already deposited in the heap (machinesHeap) and the heap is internally reorganized according to the new complexities (compare Eq. 7). Attractiveness modules may learn and organize meta-knowledge basing on the results from finished tasks.

Next, the generators flow is called (line 68) to analyze the new results. The flow passes the call to each internal generator to let the whole hierarchy analyze the

results. Generators also may learn by observation of results to provide new, more interesting machine configurations (only in the case of advanced generators).

When a task is halted by time-limiter, the task is moved to the *quarantine* for a period not counted in time directly but determined by the complexities. Instead of constructing a separate structure responsible for the functionality of a quarantine, the quarantine is realized by two naturally cooperating elements: the machines heap and the reliability term of the complexity formula (see Eq. 7). First, the reliability of the test task is corrected—see code line 73, and after that, the test task is resend to the machine heap as to quarantine—line 74. The role of quarantine is very important and the costs of using the quarantine are, fortunately, not too big. MLAs restart only these test task for which the complexity was badly approximated. To better see the costs, assume that the time complexity of a test task was completely badly approximated and the real universal time used by this task is t. In the above scheme of the quarantine, the MLA will spend, for this task, a time not greater than $t+t+\frac{1}{4}t+\frac{1}{16}t+\ldots=\frac{7}{3}t$. So the maximum overhead is $\frac{4}{3}t$, however it is the worst case—the case where we halt the process just before it would be finished (hence the two t's in the sum). The best case gives only $\frac{1}{3}t$ overhead which is almost completely insignificant. The overhead is not a serious hamper, especially, when we take to the account that the MLA with the quarantine is not affected by the halting-problem of test-task. Moreover, the cost estimation is pessimistic also from another point of view: thanks to the unification mechanism, each subsequent restart of the test may reuse significant number of submachines run before, so in practice, the time overhead is usually minimal.

5.6 Meta-Learning Results Interpretation

The final machine ranking returned by meta-learning may be interpreted in several ways. The first machine configuration in the ranking is the best one according to the measure represented by queryDefinition.

But the runners-up may not be significantly worse. Using statistical tests, like McNemmar test, a set of machines of insignificant quality differences, can be caught. Next, the final solution may be chosen according to another criterion, for example, the simplest one from the group of the best solutions or the one with the smallest variance etc.

The results may be explored in many different ways as well. For example, one may be interested in finding solutions using alternative (to the best machine) feature sets or using the smallest number of features or instances (in the case of similarity based machines), etc. Sometimes, comprehensive models may be preferred, like decision trees, if only they work sufficiently well.

In some cases it may be recommended to repeat some tests, but with little "deeper" settings analysis, before the final decision is made.

During the results ranking analysis, the commentary parts[2] of the results can also be used as a fruitful information to support the choice.

6 Machine Complexity Evaluation

Since the complexity has to determine the order of performing test tasks, its computation is extremely important. The complexity can not be computed from learning machines, but from configurations of learning machines and descriptions of their inputs, because the information about complexity is needed before the machine is ready for use. In most cases, there is no direct analytical way of computing the machine complexity on the basis of its configuration. Therefore, we introduce an approximation framework for automated complexity approximation.

6.1 Complexity in the Context of Machines

The Kolmogorov complexity definition is not very useful in real tasks especially in computational intelligence problems. The problem of finding a minimal program is unsolvable—the search space of programs is unlimited and the time of program execution is unlimited. In the case of Levin's definition (Eq. 3) it is possible to realize the Levin Universal Search (LUS) [18,17] but the problem is that this algorithm is NP-hard. This means that, in practice, it is impossible to find an exact solution to the optimization problem.

The strategy of meta-learning is different than the one of LUS. Meta-learning uses the functional definition of the search space, which is not infinite, in the finite meta-learning process. This means that the search space is, indeed, strongly limited. The generators flow is assumed to generate machine configurations which are "rational" from the point of view of given problem $\mathcal{P}$. Such solution restricts the space to the most interesting algorithms and makes it strictly dependent on the configuration of the MLA.

In our approach to meta-learning, the complexity controls the order of testing machine configurations collected in machine heap. Ordering programs only on the basis of their length (as it was defined in Kolmogorov Eq. 2) is not rational. The problem of using Levin's additional term of time, in real applications, is that it is not rigorous enough in respecting time. For example, a program running 1024 times longer than another one may have just a little bigger complexity (just +10) when compared to the rest (the length). This is why we use the following definition, with some additions described later:

$$c_a(p) = l(p) + t^p / \log(t^p). \tag{5}$$

Naturally, we use an approximation of the complexity of a machine, because the actual complexity is not known before the real test task is finished. The

[2] Parts with information about derivation of configuration and other comments on quality test.

approximation methodology is described in Section 6. Because of this approximation and because of the halting problem (we never know whether given test task will finish) an additional penalty term is added to the above definition:

$$c_b(p) = [l(p) + t^p / \log(t^p)]/q(p), \tag{6}$$

where $q(p)$ is a function term responsible for reflecting an estimate of reliability of p. At start the MLAs use $q(p) = 1$ (generally $q(p) \leq 1$) in the estimations, but in the case when the estimated time (as a part of the complexity) is not enough to finish the program p (given test task in this case), the program p is aborted and the reliability is decreased. The aborted test task is moved to a *quarantine* according to the new value of complexity reflecting the change of the reliability term. This mechanism prevents from running test tasks for unpredictably long time of execution or even infinite time. Otherwise the MLA would be very brittle and susceptible to running tasks consuming unlimited CPU resources. More details on this are presented in Section 5.4.

Another extension of the complexity measure is possible thanks to the fact that MLAs are able to collect meta-knowledge during learning. The meta-knowledge may influence the order of test tasks waiting in the machine heap and machine configurations which will be provided during the process. The optimal way of doing this, is adding a new term to the $c_b(p)$ to shift the start time of given test in appropriate direction:

$$c_m(p) = [l(p) + t^p / \log(t^p)]/[q(p) \cdot a(p)]. \tag{7}$$

$a(p)$ reflects the attractiveness of the test task p.

Complexities of What Machines are We Interested in? As described in Section 5.3, the generators flow provides machine configurations to machineGeneratorForTestTemplate and after nesting the configurations inside the test template, the whole test configurations are sent to the machinesHeap. The machinesHeap uses the complexity of the machine of given configuration, as the priority key. It is not accidental, that the machine configuration which comes to the machinesHeap is the configuration of the whole test machine (where the proposed machine configuration is nested). This complexity really well reflects complete behavior of the machine: a part of the complexity formula reflects the complexity of learning of given machine and the rest reflects the complexity of computing the test (for example classification or approximation test). The costs of learning are very important, because trivially, without learning there is no model. The complexity of the testing part is also very important, because it reflects the simplicity of further use of the model. Some machines learn quickly and require more effort to make use of their outputs (like kNN classifiers), while others learn for a long time and after that may be very efficiently exploited (like many neural networks). Therefore, the test procedure should be as similar to the whole life cycle of a machine as possible (and of course as trustful as possible).

To understand the needs of complexity computing we need to go back to the task of learning. To provide a learning machine, regardless of whether it is a

simple one, a complex machine or a machine constructed to help in the process of analysis of other machines, its configuration and inputs must be specified (compare Section 3). Complexity computation must reflect the information from configuration and inputs. The recursive nature of configurations, together with input–output connections, may compose quite complex information flow. Sometimes, the inputs of submachines become known just before they are started, i.e. after the learning of other machines[3] is finished. This is one of the most important reasons why determination of complexity, in contrary to actual learning processes, must base on *meta-inputs*, not on exact inputs (which remain unknown). Assume a simple scene, in which a classifier TC is built from a data transformer T and a classifier C (compare Figure 9). It would be impossible to compute complexity of the classifier C basing on its inputs, because one of the inputs is taken from the output of the transformer T, which will not be known before the learning process of T is finished. Complexity computation may not be limited to a part of TC machine or wait until some machines are ready. To make complexity computation possible we use proper *meta-inputs* descriptions. Meta-inputs are counterparts of inputs in the "meta-world". Meta-inputs contain descriptions (as *informative* as possible) of inputs which "explain" or "comment" every useful aspect of each input which could be helpful in determination of the complexity.

Because machine inputs are outputs of other machines, the space of meta-inputs and the space of meta-outputs are the same.

To facilitate recurrent determination of complexity—which is obligatory because of basing on a recurrent definition of machine configuration and recurrent structure of real machines—the functions, which compute complexity, must also provide meta-outputs, because such meta-outputs will play crucial role in computation of complexities of machines which read the outputs through their inputs.

In conclusion, a function computing the complexity for machine $\mathcal{L}$ should be a transformation

$$\mathcal{D}_{\mathcal{L}} \ : \ \mathcal{K}_{\mathcal{L}} \times \mathcal{M}_{+} \rightarrow R^2 \times \mathcal{M}_{+}, \tag{8}$$

where the domain is composed by the configurations space $\mathcal{K}_{\mathcal{L}}$ and the space of meta-inputs $\mathcal{M}_{+}$, and the results are the time complexity, the memory complexity and appropriate meta-outputs. It is important to see the similarity with the definition of learning (Eq. 1), because computation of complexity is a derivative of the behavior of machine learning process.

The problem is not as easy as the form of the function in Eq. 8. Finding the right function for given learning machine $\mathcal{L}$ may be impossible. This is caused by unpredictable influence of some configuration elements and of some inputs (meta-inputs) to the machine complexity. Configuration elements are not always as simple as scalar values. In some cases configuration elements are represented by functions or by subconfigurations. Similar problem concerns meta-inputs. In many cases, meta-inputs can not be represented by simple chain of scalar values. Often, meta-inputs need their own complexity determination tool to reflect

[3] Machines which provide necessary outputs.

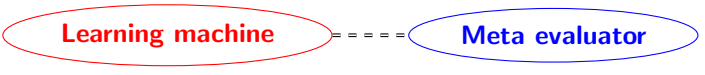
their functional form. For example, a committee of machines, which plays a role of a classifier, will use other classifiers (inputs) as "slave" machines. It means that the committee will use classifiers' outputs, and the complexity of using the outputs depends on the outputs, not on the committee itself. This shows that sometimes, the behavior of meta-inputs/outputs is not trivial and proper complexity determination requires another encapsulation.

6.2 Meta Evaluators

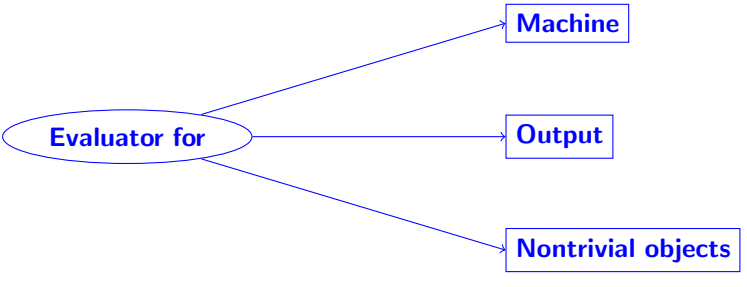
To enable so high level of generality, the concept of *meta-evaluators* has been introduced. The general *goal of meta-evaluator* is

- to evaluate and exhibit *appropriate aspects* of complexity representation basing on some meta-descriptions like meta-inputs or configuration[4].
- to exhibit a functional description of complexity aspects (comments) useful for further reuse by other meta evaluators[5].

To enable complexity computation, every learning machine gets its own meta evaluator.



Because of recurrent nature of machine (and machine configuration) and because of nontriviality of inputs behavior (which sometimes have complex functional form), meta evaluators are constructed not only for machines, but also for outputs and other elements with "nontrivial influence" on machine complexity.



Each evaluator will need *adaptation*, which can be seen as an initialization and can be compared to the learning of machine. In such meaning the process $D_{\mathcal{L}}$ (Eq. 8) will be the typical adaptation of evaluator devoted on the machine $\mathcal{L}$. It means that before using given evaluator, it has to be adapted. Then, evaluator can be used to calculate aspects of complexity devoted for given evaluator (compare, presented below, typical evaluators type and their functionality).

[4] In case of a machine to exhibit complexity of time and memory.
[5] In case of a machine the meta-outputs are exhibited to provide complexity information source for their inputs readers.

It is sometimes necessary to estimate complexity on the basis of machine configuration and real inputs (not meta-inputs as in Eq. 8). In such case, we would need an adaptation of machine evaluator in the form:

$$\mathcal{D}'_{\mathcal{L}} \ : \ \mathcal{K}_{\mathcal{L}} \times \mathcal{I}_{+} \rightarrow R^2 \times \mathcal{M}_{+}, \tag{9}$$

where $\mathcal{I}_{+}$ is the space of machine $\mathcal{L}$ inputs. Such approach would require construction of two evaluators for each machine: for the forms presented in Eq. 8 and Eq. 9. But it is possible to resign from the Eq. 9 form. The solution is to design output evaluators and their adaptation as:

$$\mathcal{D}_o \ : \ \mathcal{I}_1 \rightarrow \mathcal{M}_1, \tag{10}$$

where $\mathcal{I}_1$ is a space of (single) output and $\mathcal{M}_1$ is the space of meta-output. And now we can seen that meta-input (or meta-output) is nothing else than special case of evaluator, the output evaluator.

Using output evaluators, the "known" inputs can be transformed to meta-inputs ($\mathcal{K}_{\mathcal{L}} \times \mathcal{I}_{+} \rightarrow \mathcal{K}_{\mathcal{L}} \times \mathcal{M}_{+}$), and after that, the machine evaluator of the form of Eq. 8 can be used. This finally reduces the needs of adaptation in the form of Eq. 9.

Sometimes, machine complexity depends on nontrivial elements (as it was already mentioned), typically some parts of the configuration. Then, the behavior of machine changes according to changes of nontrivial part of the machine configuration. For example, configurations of machines like kNN or SVM are parameterized by metric. The complexity of metric (the time needed to calculate single distance between two instances) does not depend on the kNN or SVM machine, but on the metric function. Separate evaluators for such nontrivial objects, simplify creation of machine evaluators, which may use subevaluators for the nontrivial objects. Every evaluator may order creation of any number of (nested) evaluators. Adaptation of evaluators for nontrivial objects may be seen as:

$$\mathcal{D}_{obj} \ : \ \mathcal{OBJ} \rightarrow \mathcal{M}_{obj}, \tag{11}$$

where $\mathcal{OBJ}$ is the space of nontrivial objects and $\mathcal{M}_{obj}$ is their evaluators space (which is subspace of all evaluators, of course).

The adaptation process is the major functionality of each evaluator and depends on the type of the evaluator and parameters of the adaptation function. Adaptation is realized by EvaluatorBase method:

```
EvaluatorBase(object[] data);
```

In general, the goal of this method is to use the data as the "source of information" for given evaluator.

The data are different in type, goal and other aspects, depending on the type of evaluator (compare Eq. 8, 10, and 11):

- if an evaluator is defined for a machine, then the data may be a real machine or a configuration and meta-inputs,

– evaluators constructed for outputs, in the data get a real output,
– in other cases, data depend on the needs of particular evaluators.

When evaluators may be defined in analytical way (quite rare cases), the evaluators need only to be adapted via EvaluatorBase. In other cases, the *approximation framework* is used to construct evaluators (see Figure 12 and Section 6.3). The precess of creation of evaluators is presented in Section 6.3.

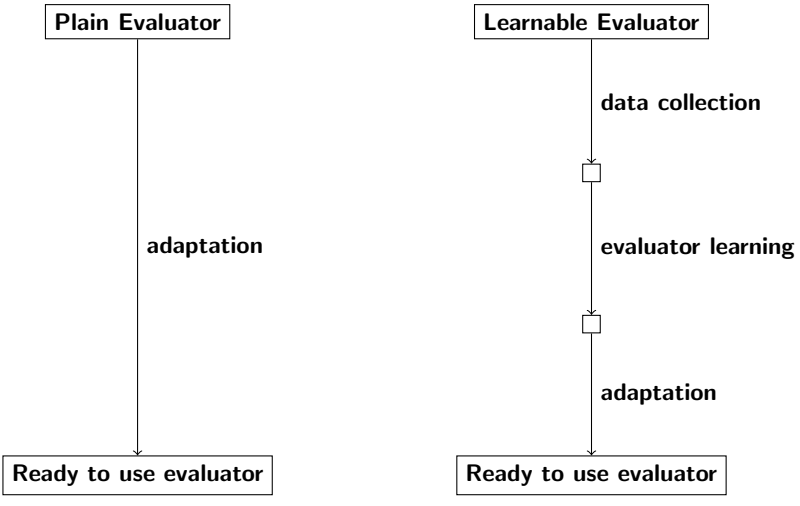


Fig. 12. Creation of ready to use evaluator for plain evaluator and evaluator constructed with approximation framework.

Further functionality of meta evaluators depends on their types. Some examples are presented in the following subsections.

Machine Evaluator. In the case of any machine evaluator, the additional functionality consists of:

Declarations of output descriptions:
: If given machine provides outputs, then also the output evaluators, devoted to this machine type, must provide meta-descriptions of the outputs. The descriptions of outputs are meta evaluators of appropriate kind (for example meta-classifier, meta-transformer, meta-data etc.). Output description may be the machine evaluator itself or a subevaluator produced by the machine evaluator or the evaluator provided by one of submachine evaluators constructed by the machine evaluator (machine may create submachines, evaluator may create evaluators of submachines basing on their configuration and meta-inputs).

Time & Memory:
: The complexities defined by Eq. 3, 5–7 make use of *program* length and time. Here, the two quantities must be provided by each machine evaluator to enable proper computation of time and memory complexity.

Child Evaluators:
: for advanced analysis of complex machines complexities, it is useful to have access to meta evaluators of submachines. Child evaluators are designed to provide this functionality.

Classifier Evaluator. Evaluator of a classifier output, has to provide the time complexity of classification of an instance:

```
real ClassifyCmplx(DataEvaluator dtm);
```

Apart from the learning time of given classifier, the time consumed by the instance classification routine is also very important in calculation of complexities. To estimate time requirements of a classifier test machine, one needs to estimate time requirements of the calls to the machine classification function. The final time estimation depends on the classifier and on the data being classified. The responsibility to compute the time complexity of the classification function, belongs to the meta classifier side (the evaluator of the classifier). Consider a classification committee: to classify data, it needs to call a sequence of classifiers to get the classification decisions of the committee members. The complexity of such classification, in most natural way, is a sum of the costs of classification using the sequence of classifiers, plus a (small) overhead which reflects the scrutiny of the committee members' results to make the final decision. Again, the time complexity of data classification is crucial to estimate the complexity and must be computable.

Approximator Evaluator. Evaluator of an approximation machine has exactly the same functionality as the one of a classifier, except that approximation time is considered in place of classification time:

```
real ApproximationCmplx(DataEvaluator dtm);
```

Data Transformer Evaluator. Evaluator of a data transformer has to provide two estimation aspects. The first one is similar to the functionality of the evaluators described above. Here it represents the time complexity of transformation of data instances. The second requirement is to provide a meta-description of data after transformation: the data evaluator. It is of highest importance—the quality of this meta-transformation of data-evaluator is transfered to the quality of further complexity calculations.

Metric Evaluator. The machines that use metrics, usually allow to set the metric at the configuration stage (e.g. kNN or SVM). As parameters of machine

configurations, metrics have nontrivial influence on the complexity of the machine while not being separate learning machines. The most reasonable way to enable complexity computation, in such cases, is to reflect the metric-dependence inside the evaluators (one evaluator per one metric). The meta-evaluators for metrics provide the functionality of time complexity of distance computation and are used by the evaluators of proper machines or outputs:

```
real DistanceTimeCmplx();
```

Data Evaluators. Another evaluators of crucial meaning are data evaluators. Their goal is to provide information about data structure and statistics. Data evaluator has to be as informative as possible, to facilitate accurate complexity determination by other evaluators. In the context of data tables, the data evaluators should provide information like the number of instances, the number of features, descriptions of features (ordered/unordered, number of values, etc.), descriptions of targets, statistical information per feature, statistical information per data and others that may provide useful information to compute complexities of different machines learning from the data.

Other Evaluators. The number of different types of meta evaluators is not determined. Above, only a few examples are presented of many instances available in the system. During future expansion of the system, as the number of machine types grows, the number of evaluators will also increase.

6.3 Learning Evaluators

Defining manually the functions to compute time and memory complexities inside evaluators for each machine (as well as other complexity quantities for evaluators of other types) is very hard or even impossible. Often, analytical equation is not known, and even if it is known or determinable, there is still a problem with conversion of the analytical formula or the knowledge about the dependencies into estimation of real time measured in universal seconds.

In any case, it is possible to build approximators for elements of evaluators which estimate complexity or help in further computation of such complexity. We have defined an *approximation framework* for this purpose. The framework is defined in very general way and enables building evaluators using approximators for different elements like learning time, size of the model, etc. Additionally, every evaluator that uses the approximation framework, may define special functions for estimation of complexity (MethodForApprox). This is useful for example to estimate time of instance classification etc. It was constructed to fulfill needs of different kinds of evaluators.

The complexity control of task starting in meta-learning does not require very accurate information about tasks complexities. It is enough to know, whether a task needs a few times more of time or memory than another task. The differences of several percent are completely out of interest here. Assuming such

level of accuracy of complexity computation, we do not loose much, because meta-learning is devoted to start many test tasks and small deviations from the optimal test task order are irrelevant. Moreover, although for some algorithms the approximation of complexity is not so accurate, the quarantine (see Section 5.5) prevents from capturing too much resources by a single long-lasting task.

Using the approximation framework, meta evaluator can learn as many aspects of machine behavior as necessary. Evaluator using approximation framework can estimate an unlimited set of quantities that may be useful for determination of complexities of some elements or some quantities for further computation of complexities. Typically, a single evaluator using the approximation framework creates several approximators. For example, evaluator of each machine has to provide time and memory complexities. The evaluator will realize it with two approximators. Additionally, in the case, when machine corresponding to given evaluator is also a classifier, the classification time may be learned as well, within the same framework (another dedicated approximator may be constructed). The approximators are constructed, learned and used (called to approximate) automatically, according to appropriate declarations in the evaluators, as it will be seen later (in the examples of evaluators). There is no manual intervention needed in the approximator building process.

Of course, before an evaluator is used by a meta-learning process, all its approximators must be trained. The learning of all evaluators may be done once, before the first start of meta-learning tasks. Typically, learned evaluators reside in an *evaluators project* which is loaded before the MLA starts its job. If the system is extended by a new learning machine and a corresponding evaluator, the evaluator (if it uses the approximation framework) has to learn and also will reside in the evaluators project for further use. This means that every evaluator using the approximation engine, has to be trained just once.

Before learning of evaluator approximation models, appropriate data tables must be collected (as learning data). This process will be described later. First we will present the evaluator functionality extension, facilitating usage of the approximation framework.

Approximation Framework of Meta Evaluators. The construction of a *learnable* evaluator (an evaluator making use of approximation framework) differs from construction of a plain evaluator (compare Figure 12).

The approximation framework enables to construct series of *approximators* for single evaluators. The *approximators* are functions approximating a real value on the basis of a vector of real values. They are learned from examples, so before the learning process, the learning data have to be collected for each approximator.

Figure 13 presents the general idea of creating the approximators for an evaluator. To collect the learning data, proper information is extracted form observations of "machine behavior". To do this an "environment" for machine monitoring must be defined. The environment configuration is sequentially adjusted, realized and observed (compare the data collection loop in the figure). Observations bring the subsequent instances of the training data (corresponding to current state of the environment and expected approximation values).

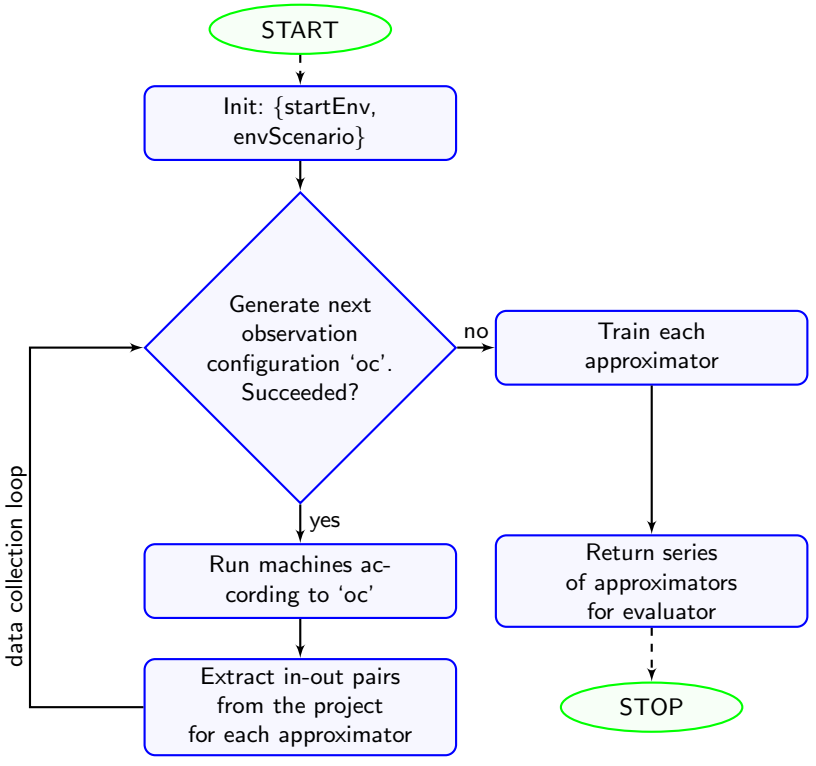


Fig. 13. Process of building approximators for single evaluator.

Changes of the environment facilitate observing the machine in different circumstances and gathering diverse data describing machine behavior in different contexts.

The environment changes are determined by initial representation of the environment (the input variable startEnv) and specialized scenario (compare Section 4), which defines how to modify the environment to get a sequence of machine observation configurations i.e. configurations of the machine being examined nest in a more complex machine structure. Generated machine observation configurations should be as realistic as possible—the information flow similar to expected applications of the machine, allows to better approximate desired complexity functions. Each time, a next configuration 'oc' is constructed, machines are created and run according to 'oc', and when the whole project is ready, the learning data are collected. Full control of data acquisition is possible thanks to proper methods implemented by the evaluators. The method EvaluatorBase is used to prepare the evaluator for analysis of the environment being observed, while GetMethodsForApprox declares additional approximation

tasks and ApproximatorDataIn, ApproximatorDataOut prepare subsequent input–output vectors from the observed environment.

When generation of new machine observation configurations in the data collection loop fails, the data collection stage is finished. Now each approximator can be learned from the collected data and after that the evaluator may use them for complexity prediction. Before the predicition, the evaluator is provided with appropriate configuration and/or meta-inputs (depending on the evaluator type)—this adaptation is performed by the EvaluatorBase method (see Eq. 8, 10, 11).

Note that the learning processes of evaluators are conducted without any advise from the user (compare Section 6.3).

From the description given above, we may conclude that the approximation framework gives two types of functionality:

- to define information necessary to collect training data for learning approximators,
- to use learned approximators.

The latter enables using approximators (after they are built within the approximation framework) by calling the function:

```
real[] Approximate(int level);
```

The input vector for given approximator is obtained by ApproximatorDataIn, described in detail below. Note that ApproximatorDataIn is used to provide input vectors to approximate and to collect learning input vectors. It is possible thanks to the extended EvaluatorBase.

The following code presents an example of using the Approximate. It is used to calculate time complexity of classification of data basing on their meta-input (dtm is an evaluator of the data).

```
79 function ClassifyCmplx(Meta.Data.DataTable_Meta dtm)
80     return Approximate(classifyCmplxLevel)[0] * dtm.DataMetaParams.InstanceCount;
81 end
```

Line 80 calls the approximator with index 0, from level classifyCmplxLevel (0-th element in the output vector is responsible for time complexity; the idea of levels and layers is addressed below).

Extended EvaluatorBase. There are two roles of the EvaluatorBase functionality:

- To prepare the adaptation process (as it was presented in Section 6.2 and in Eq. 8, 10 and 11), which has to provide necessary variables/quantities to become ready to use complexity *items* inside the evaluator. Also to prepare elements needed by ApproximatorDataIn to build the input vectors for particular approximators,
- To provide elements necessary for data collection for approximator learning basing on observed environments. EvaluatorBase has to provide elements, necessary and sufficient to build learning in-out pairs with ApproximatorDataIn, ApproximatorDataOut and GetMethodsForApprox.

The approximators can be constructed to estimate:

- time and memory complexity of machine,
- time and memory complexity of particular machine methods (declared for the approximation by GetMethodsForApprox),
- other quantities.

To embed the above three types of approximation, the approximator learners are placed in separate *levels* in three *layers*:

Level	**Layer**
level 1 ... level k	1 — Approximators of other quantities
level $k+1$ ... level $n-1$	2 — Approximators for specially defined methods
level n	3 — Approximators of time & memory complexity of machine

The order of levels reflects the order of data collection (in each iteration of data collection loop) and of further learning of approximators after data collection.

Using of approximators from the first layer may be helpful for composing input vectors for the next two layers. This functionality is used only for advanced evaluators (not too common, but sometimes very helpful) and will not be described here in more detail.

For each of the three layers, another set of evaluator methods is used to prepare the in-out learning pairs. In case of the first layer, two functions are used:

real[] ApproximatorDataIn(**int** level) — provides the input vector for level *level*,
real[] ApproximatorDataOut(**int** level) — provides the output vector for level *level*.

For the purpose of the second layer, we need:

real[] ApproximatorDataIn(**int** level) — provides the input vector for complexity of the corresponding method,
MethodForApprox[] GetMethodsForApprox() — provides table of methods being subjects to complexity checking.

To define approximation of the *machine* layer (the third one), there is only a need for

real[] ApproximatorDataIn(**int** level).

Functions of each layer, have the same general goal: to collect a single input-output pair of data for learning appropriation basing on the information extracted by EvaluatorBase from the observed environment.

Collection of the learning data for the first layer is easy to interpret: the function ApproximatorDataIn composes the input vector and ApproximatorDataOut composes the output vector.

The number of levels in the second layer is auto-defined by the number of methods returned by the GetMethodsForApprox method. When empty sequence is returned, the layer is not used by the evaluator. Otherwise, for each of the returned methods, the approximation framework, automatically approximates the time of execution and sizes of all the returned (by the method) objects. Therefore, only the input part of the learning data (ApproximatorDataIn) must be provided by the evaluator—the output is determined automatically by the system. Each approximation method MethodForApprox is defined as function:

```
object[] MethodForApprox(int repetitionNr);
```

The parameter repetitionNr is used by the approximation framework to ask for a number of repetitions of the test performed by the method (to eliminate the problem of time measurement for very quick tests). For example, let's see the code shown below, where a classifier is called to classify repetitionNr instances. The aim of the function is to measure time complexity of the classification routine.

```
82 function ClassifyTimeChecking(int repetitionNr)
83     IDataSet ds = mi.OpenInput("Dataset") as IDataSet;
84     IDataSet ds2 = RandomChainOfInstances(ds, repetitionNr);
85     IClassifier c = mi.OpenOutput("Classifier") as IClassifier;
86     IOneFeatureData d = c.Classify(ds2);
87     return null;
88 end
```

It is important to realize that the call of ClassifyTimeChecking is preceded by a call to EvaluatorBase, which sets up the mi to facilitate opening appropriate inputs (dataset and classifier).

The last layer is designed for learning time complexity and memory complexity of the machine. This layer is used only for machine evaluators. In this case, the approximation framework automatically tests the learning time and memory usage. These quantities compose the output vectors, so that definition of ApproximatorDataOut is not used here (as in the case of the second layer). The input vector is obtained, as for both previous layers, by calling the ApproximatorDataIn function, which is the only requirement for this layer.

Very important is the role of EvaluatorBase, responsible for two types of adaptation—during data collection and during complexity estimation by evaluators. The method has to provide all the information necessary to collect proper data with ApproximatorDataIn and ApproximatorDataOut methods, and to realize the tests of each associated MethodForApprox.

The input vector for each level may be different and should be as useful as possible, to simplify the process of approximator learning. Any useful element of machine description which can help to learn the characteristics of complexity should

be placed inside the input vector. The same ApproximatorDataIn method is called also before the evaluator estimates the complexity of a machine. After the valuator adapts to given machine configuration and meta-inputs, ApproximatorDataIn prepares data for all the approximators to predict their targets and final complexities are estimated.

Environments for approximators learning. As shown in Figure 13 and described above, to build input data tables, necessary for training the approximators, the machine is observed in changing environment. Each change in the environment results in a single input–output pair of data for each of approximators. Therefore, to construct successful complexity evaluators, apart from specification of the necessary approximators, one needs to define the environment and the way of its manipulation.

To share the ways of handling environments, some groups of common properties are defined and each evaluator has to assign itself to one of the groups or to define a new group and assign to it. For example, to learn estimation of machine complexities, the machine should be trained using different configurations with different input data to explore the space of significantly different situations of the machine training and exploitation of its model.

The machine observation configurations, generated in the data collection loop are determined by the following items:

IConfiguration ApproximationConfig — defines the initial configuration of the machine closest environment for observations, to be nested in the ApproximationGroupTemplate defined for the group (see below). ApproximationConfig is needed because not always, it is enough to learn and observe a single machine. Sometimes it is necessary to precede the learning of the machine by a special procedure (some necessary data transformation, etc.). However sometimes the machine may be used directly (then the property is just a configuration instance of the machine).

int[] Path — determines the placement of the machine, being observed, in the environment defined above. For machines, it points the machine. For outputs, it points the machine which provides the output.

IScenario Scenario — defines the scenario (see Section 4), which goal is to provide different configurations derived from the ApproximationConfig to explore the space of machine observation configurations. For example, in the case of the kNN machine, the scenario may browse through different values of the number of neighbors k and different metric definitions.

MachineGroup Group; — encapsulates a few functionalities, which extend the space of observed configurations. The groups of functionalities are shared between evaluators of similar type, which simplifies the process of defining evaluators. Each group is characterized by:

- string[] DataFileNames; — defines file names of learning data which will be used to observe behavior of the learning process.
- IConfiguration ApproximationGroupTemplate; — defines the procedure of using given type of machines. For example, it may consist of two elements

in a scheme: a data multiplier which constructs learning data as a random sequence of instances and features from a dataset provided as an input, and a placeholder for a classifier (an empty scheme to be replaced by a functional classification machine).

int[] Path; — points the placeholder within the ApproximationGroupTemplate, to be filled with the observed machine, generated by Scenario used on ApproximationConfig (compare with the Path described above).

IScenario GroupScenario; — the scenario of configuration changes (see Section 4) to the ApproximationGroupTemplate. The environment is subject to changes of a data file DataFileNames and configuration changes defined by the GroupScenario. For example, this scenario may cooperate with a machine randomizing the learning data within the ApproximationGroupTemplate as it was already mentioned.

All the functionalities described above, used together, provide very flexible approximation framework. Evaluators can be created and functionally-tuned, according to the needs, supplying important help in to successful complexity computation.

The functions discussed above, are used in the meta-code of the next section, to present some aspects of the proposed meta-learning algorithm.

Creation and Learning of Evaluators. After presenting the idea of the approximation framework for evaluators, here, we present the algorithms constructing evaluators, in more detail.

Before any meta-learning algorithm is run, the function CreateEvaluatorsDictionary builds a dictionary of evaluators which are constructed by the function CreateEvaluator. In fact, CreateEvaluatorsDictionary creates the evaluators project, which is used inside any meta-learning task.

```
89 function CreateEvaluatorsDictionary(Type[] allMachineTypes);
90     foreach (machineType in allMachineTypes)
91         evalDict[machineType] = CreateEvaluator(machineType);
92     return evalDict;
93 end
```

Creation of an evaluator starts with the creation of an instance (object) of given class corresponding to the type of given learning machine (machineType, see code line 95).

```
94 function CreateEvaluator(Type machineType);
95     eval = getEvalutorInstanceFor(machineType);
96     if(eval is ApproximableEvaluator)
97     {
98         sequenceOfDatasets = CreateDataTablesForApprox(machineType);
99         listOfApprox = {};
100        for (level=1 to eval.LevelsCount)
101        {
```

```
102                 <TRS, TES> = GetTrainTestDataTablesFor(level);
103                 approxTab = TrainApproximatorTab(<TRS, TES>);
104                 listOfApprox.Append(approxTab);
105             }
106             eval.Approximators = listOfApprox;
107         }
108         return eval;
109     end
```

If the evaluator does not use the approximation framework, then it is ready (without learning) and may be called to estimate complexities. Otherwise, learning of appropriate approximators is performed. Line 98 calls a function (described later) which creates a sequence of learning data tables, according to the meta-description of the evaluator.

Next lines (100–105), for each level, prepare data tables and start learning of a vector of approximators. The vector of approximators is appended to the list listOfApprox and finally assigned to the evaluator (in line 106).

The function CreateDataTablesForApprox plays a crucial role in the complexity approximation framework, as it constructs learning data tables for the approximators. To start with, it needs an instance (object) of the evaluator (line 112) and the meta-description of its requirements related to the approximation framework.

```
110 function CreateDataTablesForApprox(Type machineType);
111     sequenceOfDatasets = {};
112     eval = getEvalutorInstanceFor(machineType);
113     machineGroup = eval.Group;
114     foreach (DataFileName in machineGroup.DataFileNames)
115     {
116         dataMachine = CreateDataLoader(DataFileName);
117         groupScenario = machineGroup.GroupScenario;
118         groupScenario.Init(machineGroup.ApproximationGroupTemplate);
119         foreach (gconfig in groupScenario)
120         {
121             scenario = eval.Scenario;
122             scenario.Init(eval.ApproximationConfig);
123             foreach (sconfig in scenario)
124             {
125                 c = PlaceConfigInTemplate(gconfig, sconfig, machineGroup.Path);
126                 t = StartTask(c, dataset);
127                 m = t.GetSubmachine(machineGroup.Path + eval.Path);
128                 eval.EvaluatorBase(m);
129                 for (level=1 to eval.InnerApproximatorLevels) // layer 1 (other quantities)
130                 {
131                     dtIn[level].AddVector(eval.ApproximatorDataIn(level));
132                     dtOut[level].AddVector(eval.ApproximatorDataOut(level));
133                 }
134                 foreach (meth in eval.GetMethodsForApprox()) // layer 2 (methods)
135                 {
136                     level++;
137                     <time, objectsTab> = CheckMethod(meth, t);
138                     objectsSizes = CheckSizes(objectsTab);
```

```
139                         dtIn[level].AddVector(eval.ApproximatorDataIn(level));
140                         dtOut[level].AddVector(<time, objectsSizes>);
141                     }
142                     if (eval is assigned to a machine) // layer 3 (machine)
143                     {
144                         level++;
145                         dtIn[level].AddVector(eval.ApproximatorDataIn(level));
146                         dtOut[level].AddVector(<t.time, t.memorySize>);
147                     }
148                 }
149             }
150         }
151         sequenceOfDatasets = resplit(<dtIn, dtOut>);
152         return sequenceOfDatasets;
153     end
```

The observations are performed for each dataset (the loop in line 114), for each group configuration gconfig generated by the group scenario (the loop in line 119), for each machine configuration sconfig generated by the scenario (the loop in line 123). The scenarios are initialized before they provide the configurations (see lines 118 and 122, and Section 4).

In line 125, the configuration sconfig is placed within the group configuration gconfig in the location defined by machineGroup.Path. This composes a configuration c which defines the observation task t (next line). The following line extracts a link to the observed machine m from the observation task.

Next, basing on the link m, the function EvaluatorBase is called to provide information on the observed machine to ApproximationData(In/Out).

Then, new instances are added to the learning data tables for approximators of subsequent levels (first those of the first layer, then the methods layer and finally the machine layer).

As described before, the method ApproximationDataIn is called for each layer, while ApproximationDataOut just for the levels of the first layer—the system automatically prepares the output data for both methods layer (the outputs are time and proper object sizes) and machine layer (the outputs are time and memory complexities).

At the end (line 151), the input and target parts are transformed to appropriate data tables to be returned by the function.

6.4 Example of Evaluators

To better see the practice of evaluators, we present key aspects of evaluators for some particular machines.

Evaluator for K Nearest Neighbors Machine. This evaluator is relatively simple. Nevertheless, the requires functionality must be defined (according to the description of the preceding sections).

EvaluatorBase

In the case of kNN, in this function, the evaluator saves the kNN configuration and the evaluator of the input data. Another goal is to prepare the output description of the classifier's evaluator.

```
154 function EvaluatorBase(object[] data);
155     Config = GetConfiguration(data);
156     Outputs_Meta inputsMeta = GetOutput_MetaFrom(data);
157     DataEvaluator = inputsMeta["Dataset"][0];
158     DeclareOutputDescription("Classifier", this);
159 end
```

Time

kNN machines do not learn, so the time of learning is 0.

```
160 function Time()
161     return 0;
162 end
```

Memory

The model uses as much memory as the input data, i.e. DataEvaluator.MemoryCmplx():

```
163 function Memory()
164     return DataEvaluator.MemoryCmplx();
165 end
```

ClassifyCmplx

This function approximates the complexity of classification using the approximator:

```
166 function ClassifyCmplx(DataEvaluator dtm);
167     return Approximate(classifyCmplxLevel)[0] * dtm.InstanceCount;
168 end;
```

classifyCmplxLevel points appropriate approximators level.

ApproximatorDataIn

The method provides training data items for approximation of the classification complexity (line 172) and of the machine learning process complexity (line 175):

```
169 function ApproximatorDataIn(int level)
170     switch (level)
171     {
172         case classifyCmplxLevel:
173             return { Config.K,
174                 DataEvaluator.InstanceCount * Metric_Meta.DistanceTimeCmplx };
175         case machineLevel:
176             return { Config.K, Metric_Meta.DistanceTimeCmplx,
177                 DataEvaluator.InstanceCount, DataEvaluator.FeatureCount };
178     }
179 end
```

ApproximationConfig
: For kNN, it is just the configuration of kNN machine.

```
180 function ApproximationConfig()
181     return new kNNConfig();
182 end
```

Scenario
: The scenario manipulates the "k" (the numbers of neighbors) and the metric.

Path
: The kNN configuration is not nested in another machine configuration (it constitutes the ApproximationConfig itself), so the path does not need to point any internal configuration, hence is empty.

```
183 function Path()
184     return null;
185 end
```

GetMethodsForApprox
: Returns the function ClassifyTimeChecking devoted to computing the time of classification with the kNN model:

```
186 function GetMethodsForApprox()
187     return new MethodForApprox[]{ ClassifyTimeChecking };
188 end
```

The kNN evaluator is assigned to a machine group prepared for classifiers. The group definition includes:

ApproximationGroupTemplate
: This template is a scheme configuration with two subconfigurations. The first is the RandomSubset machine, which provides data sets consisting of different numbers of randomly selected instances and features taken from some source data set. The second subconfiguration is the placeholder for a classifier. At runtime, the placeholder is filled with proper classifier (in this case with the kNN configuration). The classifier gets data input from the RandomSubset machine output.

GroupScenario
: It randomizes the configuration of RandomSubset machine to obtain more observations for learning the approximation targets.

Example of Evaluator for Boosting Machine. Boosting is an example of machine using many submachines. Intemi implementation of boosting machine, repeatedly creates triples of submachines consisting of data distributor machine, classifier machine and a test of the classifier. All of the submachines have their own influence on the complexity of the boosting machine.

EvaluatorBase

Boosting evaluator requires more complex EvaluatorBase then the one of kNN:

```
189 function EvaluatorBase(params object[] data)
190     Config = GetConfiguration(data);
191     Outputs_Meta inputsMeta = GetOutput_MetaFrom(data);
192     DataEvaluator = inputsMeta["Dataset"][0];
193     Outputs_Meta d = { {"Dataset", DataEvaluator} };
194     boostDistributor = EvaluatorEngine.EvaluateMachine(
195         BoostingDistributor, d);
196     classifier = EvaluatorEngine.EvaluateMachine(
197         Config.Subconfigurations[0], d);
198     Outputs_Meta d2 = new Outputs_Meta();
199     d2.Add("Dataset", DataEvaluator);
200     d2.Add("Classifier", classifier.GetOutputDescription("Classifier"));
201     classTest = EvaluatorEngine.EvaluateMachine(
202         new Intemi.Testing.ClassTestConfig(), d2);
203     DeclareOutputDescription("Classifier", this);
204 end
```

Lines 190 and 192 determine the configuration of boosting machine and the data evaluator, similarly as for the evaluator of kNN. In line 194 the data distributor evaluator is created using the EvaluatorEngine, which enables creation of evaluators by other evaluators.

The classifier evaluator (see line 196) is constructed in a similar way to the data evaluator. Here, the classifier configuration is extracted from the structure of subconfigurations, because it may be any classifier defined as the first subconfigration of the boosting configuration.

Line 201 constructs the evaluator of a classification test. The evaluator gets meta-inputs descriptions of the data evaluator (DataEvaluator) and the classifier evaluator (classifier).

Because boosting is a classifier, the last line of the code of EvaluatorBase declares output meta-description of the classifier.

Time

The time of boosting machine training is the sum of time amounts necessary to build the sequence of distributors, classifiers and test machines plus the time of the boosting-only part of learning:

```
205 function Time
206   return JustThisMachineTime + Config.NrOfClassifiers *
207       (boostDistributor.Time + classifier.Time + classTest.Time);
208 end
```

Memory

Calculation of the occupied memory is analoguous to that of time consumption:

```
209 function Memory
210   return JustThisMachineMemory + Config.NrOfClassifiers *
211       (boostDistributor.Memory + classifier.Memory + classTest.Memory);
212 end
```

ClassifyCmplx

The costs of classifying by boosting models are nearly equal to the sum of classifying given data (dtm evaluator) by each of the subclassifiers:

```
213 function ClassifyCmplx(DataEvaluator dtm)
214     subclass = classifier.GetOutputDescription("Classifier");
215   return Config.NrOfClassifiers *
216       subclass.ClassifyCmplx(dtm) * 1.1;
217 end
```

ApproximatorDataIn

The input data for the approximators is quite easy to determine. Boosting complexity (excluding learning of submachines) depend mostly on the number of submachines (the cost of creation not of the learning) and on the size of data. Thus:

```
218 function ApproximatorDataIn(int level)
219   return { Config.NrOfClassifiers, DataEvaluator.InstanceCount };
220 end
```

ApproximationConfig

The tested configuration is just a boosting configuration with proper classifier configuration inside (here, the naive Bayes classifier):

```
221 function ApproximationConfig
222   BoostingConfig c;
223   c.ClassfierTemplate = NBCConfig();
224   return c;
225 end
```

The evaluator of boosting will work properly not only with naive Bayes, because in code line 196, appropriate evaluator for inner classifier is constructed (at the time of complexity estimation, it is the evaluator of the classifier defined in the configuration).

Scenario

The scenario, simply builds configurations with different numbers of submachines.

```
226 function Scenario()
227     return new Meta.ParamSearch.StepScenario_I(null,
228         new string[] { "NrOfClassifiers" },
229         Meta.ParamSearch.StepScenario_I.StepTypes.Linear, 10, 10, 3);
230 end
```

Path and GetMethodsForApprox
: These properties are exactly the same as in the evaluator of the kNN machine.

Boosting machine is a classifier, so its evaluator is also attached to the group devoted to classifiers. Therefore, the group items are the same as in the case of kNN.

7 Meta-Learning in Action

Presented meta-learning algorithm, or rather meta-learning system, may be used in variety of ways. Generators flow may be defined as a simple graph, but usually, for advanced problems, it is quite nontrivial graph, which in effect produces many test configurations. The goal of meta-learning which reflects the problem type may also be defined in several ways, according to the needs. Similarly, the stop criterion should reflect the preferences about the conditions of regarding the meta-search as finished.

To present meta-learning in action, we have used a few well known problems from the UCI Machine Learning repository [34]. All the benchmarks, presented below, are classification problems. All the following results are computed using the same configuration of meta-learning (obviously except the specification of the benchmark dataset).

First, we have to present the meta-learning configuration, according to the information presented in Sections 5.2 and 5.3. The configuration consists of several elements: the meta-learning test template, query test, stop criterion and the generators flow.

Meta-learning Test Template
The test template exhibits the goal of the problem. Since, the chosen benchmarks are classification problems, we may use cross-validation as the strategy for estimation of classifiers capabilities. The repeater machine may be used as the test configuration with distributor set up to the CV-distributor and the inner test scheme containing a placeholder for classifier and a classification test machine configuration, which will test each classifier machine and provide results for further analysis. Such a repeater machine configuration template is presented in Figure 14. When used as the ML test template, it will be repeatedly converted to different feasible configurations by replacing the classifier placeholder inside the template with classifiers configurations generated by the generators flow.

Query test
To test a classifier quality, the accuracies calculated by the classification test machines may be averaged and the mean value may be used as the quality measure.

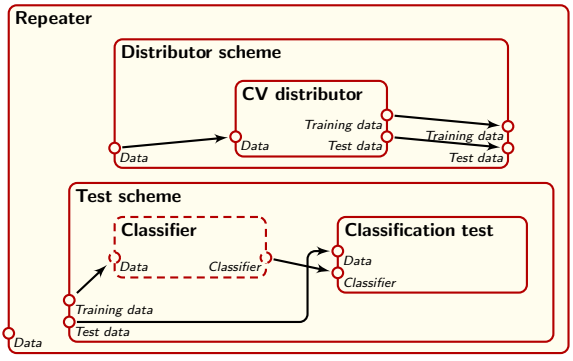


Fig. 14. Meta-learning test template: Repeater machine configuration for cross-validation test with placeholder for classifier.

Stop criterion
In the tests, the stop criterion was defined to become true when all the configurations provided by the generators flow are tested.

Generators flow
The generators flow used for this analysis of meta-learning is rather simple, to give the opportunity to observe the behavior of the algorithm. It is not the best choice for solving classification problems, in general, but lets us better see the very interesting details of its cooperation with the complexity control mechanism. To find more sophisticated configuration machines, more complex generators graph should be used. Anyways, it will be seen that using even so basic generators flow, the results ranked high by the MLA, can be very good. The generators flow used in experiments is presented in Figure 15. Very similar generators flow was explained in detail in Section 5.1.

To know what exactly will be generated by this generators flow, the configurations (the sets) of classifiers generator and rankings generator must be specified. Here, we use the following:

Classifier set:

kNN (Euclidean) — k Nearest Neighbors with Euclidean metric,
kNN [MetricMachine (EuclideanOUO)] — kNN with Euclidean metric for ordered features and Hamming metric for unordered ones,
kNN [MetricMachine (Mahalanobis)] — kNN with Mahalanobis metric,
NBC — Naive Bayes Classifier
SVMClassifier — Support Vector Machine with Gaussian kernel
LinearSVMClassifier — SVM with linear kernel

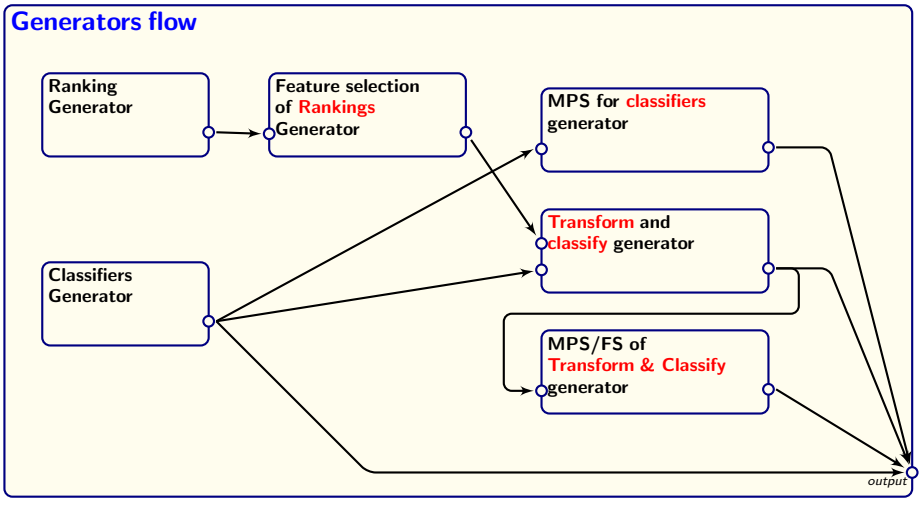


Fig. 15. Generators flow used in tests.

[ExpectedClass, kNN [MetricMachine (EuclideanOUO)]] — first, the ExpectedClass[6] machine transforms the original dataset, then the transformed data become the learning data for kNN,
[LVQ, kNN (Euclidean)] — first, Learning Vector Quantization algorithm [35] is used to select prototypes, then kNN uses them as its training data (neighbor candidates),
Boosting (10x) [NBC] — boosting algorithm with 10 NBCs.

Ranking set:

RankingCC — correlation coefficient based feature ranking,
RankingFScore — Fisher-score based feature ranking.

The base classifiers and ranking algorithms, together with the generators flow presented in Figure 15, produce 54 configurations, that are nested (one by one) within the meta-learning test-scheme and sent to the meta-learning heap for complexity controlled run.

All the configurations provided by the generators flow are presented in Table 3. The square brackets, used there, denote submachine relation. A machine name standing before the brackets is the name of the parent machine, and the machines in the brackets are the submachines. When more than one name is embraced with

[6] ExpectedClass is a transformation machine, which outputs a dataset consisting of one "super-prototype" per class. The super-prototype for each class is calculated as vector of the means (for ordered features) or expected values (for unordered features) for given class. Followed by a kNN machine, it composes a very simple classifier, even more "naive" than the Naive Bayes Classifier, though sometimes quite successful.

the brackets (comma-separated names), the machines are placed within a scheme machine. Parentheses embrace significant parts of machine configurations.

To make the notation easier to read, we explain some entries of the table. The notation does not present the input–output interconnections, so it does not allow to reconstruct the full scenario in detail, but shows machine structure, which is sufficient, here, and significantly reduces the occupied space.

The following notation:

[[[RankingCC], FeatureSelection], [kNN (Euclidean)], TransformAndClassify]

means that a feature selection machine selects features from the top of a correlation coefficient based ranking, and next, the dataset composed of the feature selection is an input for a kNN with Euclidean metric—the combination of feature selection and kNN classifier is controlled by a TransformAndClassify machine.

Notation:

[[[RankingCC], FeatureSelection], [LVQ, kNN (Euclidean)], TransformAndClassify]

means nearly the same as the previous example, except the fact that between the feature selection machine and the kNN is placed an LVQ machine as the instance selection machine.

The following notation represents the ParamSearch machine which optimizes parameters of a kNN machine:

ParamSearch [kNN (Euclidean)]

In the case of

ParamSearch [LVQ, kNN (Euclidean)]

both LVQ and kNN parameters are optimized by the ParamSearch machine.

However in the case of notation

ParamSearch [[[RankingCC], FeatureSelection], kNN (Euclidean)]

only the number of chosen features is optimized because this configuration is provided by the MPS/FS of Transform & Classify Generator (see Figure 15), where the ParamSearch configuration is set up to optimize only the parameters of feature selection machine. Of course, it is possible to optimize all the parameters of all submachines, but this is not the goal of the example and, moreover, the optimization of too many parameters may provide to very complex machines (sometimes uncomputable in a rational time).

Table 3. Machine configurations produced by the generators flow of Figure 15 and the enumerated sets of classifiers and rankings.

1	kNN (Euclidean)
2	kNN [MetricMachine (EuclideanOUO)]
3	kNN [MetricMachine (Mahalanobis)]
4	NBC
5	SVMClassifier [KernelProvider]
6	LinearSVMClassifier [LinearKernelProvider]
7	[ExpectedClass, kNN [MetricMachine (EuclideanOUO)]]
8	[LVQ, kNN (Euclidean)]
9	Boosting (10x) [NBC]
10	[[[RankingCC], FeatureSelection], [kNN (Euclidean)], TransformAndClassify]
11	[[[RankingCC], FeatureSelection], [kNN [MetricMachine (EuclideanOUO)]], TransformAndClassify]
12	[[[RankingCC], FeatureSelection], [kNN [MetricMachine (Mahalanobis)]], TransformAndClassify]
13	[[[RankingCC], FeatureSelection], [NBC], TransformAndClassify]
14	[[[RankingCC], FeatureSelection], [SVMClassifier [KernelProvider]], TransformAndClassify]
15	[[[RankingCC], FeatureSelection], [LinearSVMClassifier [LinearKernelProvider]], TransformAndClassify]
16	[[[RankingCC], FeatureSelection], [ExpectedClass, kNN [MetricMachine (EuclideanOUO)]], TransformAndClassify]
17	[[[RankingCC], FeatureSelection], [LVQ, kNN (Euclidean)], TransformAndClassify]
18	[[[RankingCC], FeatureSelection], [Boosting (10x) [NBC]], TransformAndClassify]
19	[[[RankingFScore], FeatureSelection], [kNN (Euclidean)], TransformAndClassify]
20	[[[RankingFScore], FeatureSelection], [kNN [MetricMachine (EuclideanOUO)]], TransformAndClassify]
21	[[[RankingFScore], FeatureSelection], [kNN [MetricMachine (Mahalanobis)]], TransformAndClassify]
22	[[[RankingFScore], FeatureSelection], [NBC], TransformAndClassify]
23	[[[RankingFScore], FeatureSelection], [SVMClassifier [KernelProvider]], TransformAndClassify]
24	[[[RankingFScore], FeatureSelection], [LinearSVMClassifier [LinearKernelProvider]], TransformAndClassify]
25	[[[RankingFScore], FeatureSelection], [ExpectedClass, kNN [MetricMachine (EuclideanOUO)]], TransformAndClassify]
26	[[[RankingFScore], FeatureSelection], [LVQ, kNN (Euclidean)], TransformAndClassify]
27	[[[RankingFScore], FeatureSelection], [Boosting (10x) [NBC]], TransformAndClassify]
28	ParamSearch [kNN (Euclidean)]
29	ParamSearch [kNN [MetricMachine (EuclideanOUO)]]
30	ParamSearch [kNN [MetricMachine (Mahalanobis)]]

Table 3. (*continued*)

31	ParamSearch [NBC]
32	ParamSearch [SVMClassifier [KernelProvider]]
33	ParamSearch [LinearSVMClassifier [LinearKernelProvider]]
34	ParamSearch [ExpectedClass, kNN [MetricMachine (EuclideanOUO)]]
35	ParamSearch [LVQ, kNN (Euclidean)]
36	ParamSearch [Boosting (10x) [NBC]]
37	ParamSearch [[[RankingCC], FeatureSelection], [kNN (Euclidean)], TransformAndClassify]
38	ParamSearch [[[RankingCC], FeatureSelection], [kNN [MetricMachine (EuclideanOUO)]], TransformAndClassify]
39	ParamSearch [[[RankingCC], FeatureSelection], [kNN [MetricMachine (Mahalanobis)]], TransformAndClassify]
40	ParamSearch [[[RankingCC], FeatureSelection], [NBC], TransformAndClassify]
41	ParamSearch [[[RankingCC], FeatureSelection], [SVMClassifier [KernelProvider]], TransformAndClassify]
42	ParamSearch [[[RankingCC], FeatureSelection], [LinearSVMClassifier [LinearKernelProvider]], TransformAndClassify]
43	ParamSearch [[[RankingCC], FeatureSelection], [ExpectedClass, kNN [MetricMachine (EuclideanOUO)]], TransformAndClassify]
44	ParamSearch [[[RankingCC], FeatureSelection], [LVQ, kNN (Euclidean)], TransformAndClassify]
45	ParamSearch [[[RankingCC], FeatureSelection], [Boosting (10x) [NBC]], TransformAndClassify]
46	ParamSearch [[[RankingFScore], FeatureSelection], [kNN (Euclidean)], TransformAndClassify]
47	ParamSearch [[[RankingFScore], FeatureSelection], [kNN [MetricMachine (EuclideanOUO)]], TransformAndClassify]
48	ParamSearch [[[RankingFScore], FeatureSelection], [kNN [MetricMachine (Mahalanobis)]], TransformAndClassify]
49	ParamSearch [[[RankingFScore], FeatureSelection], [NBC], TransformAndClassify]
50	ParamSearch [[[RankingFScore], FeatureSelection], [SVMClassifier [KernelProvider]], TransformAndClassify]
51	ParamSearch [[[RankingFScore], FeatureSelection], [LinearSVMClassifier [LinearKernelProvider]], TransformAndClassify]
52	ParamSearch [[[RankingFScore], FeatureSelection], [ExpectedClass, kNN [MetricMachine (EuclideanOUO)]], TransformAndClassify]
53	ParamSearch [[[RankingFScore], FeatureSelection], [LVQ, kNN (Euclidean)], TransformAndClassify]
54	ParamSearch [[[RankingFScore], FeatureSelection], [Boosting (10x) [NBC]], TransformAndClassify]

Data benchmarks
Table 4 summarizes the properties of data benchmarks (from the UCI repository) used in the tests.

Table 4. Benchmark data used for the tests.

Dataset	# classes	# instances	# features	# ordered f.
appendicitis	2	106	7	7
german-numeric	2	1000	24	24
glass	6	214	9	9
mushroom	2	8124	22	0
splice	3	3190	60	0
thyroid-all	3	7200	21	6

Table 5 presents exact complexities (see Eq. 6 for each test machine configuration obtained for the vowel data. The table has three columns: the first one contains the task id which corresponds to the order of configurations providing by the generators flow (the same as the ids in Table 3), the second column is the task configuration description, and the third column shows the task complexity. The rows are sorted according to the complexity.

The results obtained for the benchmarks are presented in the form of diagrams. The diagrams are very specific and present many properties of the meta-learning algorithm. The diagrams present information about the times of starting, stopping and breaking of each task, about complexities (global, time and memory) of each test task, about the order of the test tasks (according to their complexities, compare Table 3) and about the accuracy of each tested machine.

In the middle of the diagram—see the first diagram in Figure 16—there is a column with task ids (the same ids as in tables 3 and 5). But the row order in diagram reflects the complexities of test task, not the order of machine creation. It means that at the top, the most complex tasks are placed and at the bottom the task of the smallest complexities. For example, in Figure 16, at the bottom, we can see task ids 4 and 31 which correspond to the Naive Bayes Classifier and the ParamSearch [NBC] classifier. At the top, task ids 54 and 45 are the most complex ParamSearch test tasks of this benchmark.

On the right side of the *Task id* column, there is a plot presenting starting, stopping and breaking times of each test task. As it was presented in Section 5.4 the tasks are started according to the approximation of their complexities, and when a given task does not reach the time limit (which correspond to the time complexity—see Section 6.1) it finishes normally, otherwise, the task is broken and restarted according to the modified complexity (see Section 6.1). For an example of restarted task please look at Figure 16, at the topmost task-id 54—there are two horizontal bars corresponding to the two periods of the task run. The break means that the task was started, broken because of exceeded allocated time and restarted when the tasks of larger complexities got their turn. The breaks occur for the tasks, for which the complexity prediction was too

Table 5. Complexities of the tasks produced by the generators flow for vowel data.

4	NBC	4.77E+006
31	ParamSearch [NBC]	4.99E+006
13	[[[RankingCC], FeatureSelection], [NBC], TransformAndClassify]	5.25E+006
22	[[[RankingFScore], FeatureSelection], [NBC], TransformAndClassify]	5.26E+006
7	[ExpectedClass, kNN [MetricMachine (EuclideanOUO)]]	5.29E+006
1	kNN (Euclidean)	5.78E+006
16	[[[RankingCC], FeatureSelection], [ExpectedClass, kNN [MetricMachine (EuclideanOUO)]], TransformAndClassify]	5.81E+006
25	[[[RankingFScore], FeatureSelection], [ExpectedClass, kNN [MetricMachine (EuclideanOUO)]], TransformAndClassify]	5.81E+006
10	[[[RankingCC], FeatureSelection], [kNN (Euclidean)], TransformAndClassify]	5.84E+006
19	[[[RankingFScore], FeatureSelection], [kNN (Euclidean)], TransformAndClassify]	5.84E+006
2	kNN [MetricMachine (EuclideanOUO)]	7.82E+006
11	[[[RankingCC], FeatureSelection], [kNN [MetricMachine (EuclideanOUO)]], TransformAndClassify]	8.09E+006
20	[[[RankingFScore], FeatureSelection], [kNN [MetricMachine (EuclideanOUO)]], TransformAndClassify]	8.09E+006
17	[[[RankingCC], FeatureSelection], [LVQ, kNN (Euclidean)], TransformAndClassify]	8.18E+006
26	[[[RankingFScore], FeatureSelection], [LVQ, kNN (Euclidean)], TransformAndClassify]	8.18E+006
12	[[[RankingCC], FeatureSelection], [kNN [MetricMachine (Mahalanobis)]], TransformAndClassify]	9.60E+006
21	[[[RankingFScore], FeatureSelection], [kNN [MetricMachine (Mahalanobis)]], TransformAndClassify]	9.60E+006
3	kNN [MetricMachine (Mahalanobis)]	9.70E+006
8	[LVQ, kNN (Euclidean)]	1.00E+007
6	LinearSVMClassifier [LinearKernelProvider]	1.19E+007
33	ParamSearch [LinearSVMClassifier [LinearKernelProvider]]	1.21E+007
15	[[[RankingCC], FeatureSelection], [LinearSVMClassifier [LinearKernelProvider]], TransformAndClassify]	1.46E+007
24	[[[RankingFScore], FeatureSelection], [LinearSVMClassifier [LinearKernelProvider]], TransformAndClassify]	1.46E+007
14	[[[RankingCC], FeatureSelection], [SVMClassifier [KernelProvider]], TransformAndClassify]	1.72E+007
23	[[[RankingFScore], FeatureSelection], [SVMClassifier [KernelProvider]], TransformAndClassify]	1.72E+007
5	SVMClassifier [KernelProvider]	1.82E+007
18	[[[RankingCC], FeatureSelection], [Boosting (10x) [NBC]], TransformAndClassify]	4.20E+007

Table 5. (*continued*)

27	[[[RankingFScore], FeatureSelection], [Boosting (10x) [NBC]], TransformAndClassify]	4.20E+007
9	Boosting (10x) [NBC]	4.31E+007
36	ParamSearch [Boosting (10x) [NBC]]	4.33E+007
34	ParamSearch [ExpectedClass, kNN [MetricMachine (EuclideanOUO)]]	5.27E+007
29	ParamSearch [kNN [MetricMachine (EuclideanOUO)]]	6.84E+007
30	ParamSearch [kNN [MetricMachine (Mahalanobis)]]	8.09E+007
40	ParamSearch [[[RankingCC], FeatureSelection], [NBC], TransformAndClassify]	1.63E+008
49	ParamSearch [[[RankingFScore], FeatureSelection], [NBC], TransformAndClassify]	1.63E+008
37	ParamSearch [[[RankingCC], FeatureSelection], [kNN (Euclidean)], TransformAndClassify]	1.78E+008
46	ParamSearch [[[RankingFScore], FeatureSelection], [kNN (Euclidean)], TransformAndClassify]	1.78E+008
43	ParamSearch [[[RankingCC], FeatureSelection], [ExpectedClass, kNN [MetricMachine (EuclideanOUO)]], TransformAndClassify]	1.79E+008
52	ParamSearch [[[RankingFScore], FeatureSelection], [ExpectedClass, kNN [MetricMachine (EuclideanOUO)]], TransformAndClassify]	1.79E+008
38	ParamSearch [[[RankingCC], FeatureSelection], [kNN [MetricMachine (EuclideanOUO)]], TransformAndClassify]	2.24E+008
47	ParamSearch [[[RankingFScore], FeatureSelection], [kNN [MetricMachine (EuclideanOUO)]], TransformAndClassify]	2.24E+008
44	ParamSearch [[[RankingCC], FeatureSelection], [LVQ, kNN (Euclidean)], TransformAndClassify]	2.39E+008
53	ParamSearch [[[RankingFScore], FeatureSelection], [LVQ, kNN (Euclidean)], TransformAndClassify]	2.39E+008
39	ParamSearch [[[RankingCC], FeatureSelection], [kNN [MetricMachine (Mahalanobis)]], TransformAndClassify]	2.54E+008
48	ParamSearch [[[RankingFScore], FeatureSelection], [kNN [MetricMachine (Mahalanobis)]], TransformAndClassify]	2.54E+008
42	ParamSearch [[[RankingCC], FeatureSelection], [LinearSVMClassifier [LinearKernelProvider]], TransformAndClassify]	3.65E+008
51	ParamSearch [[[RankingFScore], FeatureSelection], [LinearSVMClassifier [LinearKernelProvider]], TransformAndClassify]	3.65E+008
41	ParamSearch [[[RankingCC], FeatureSelection], [SVMClassifier [KernelProvider]], TransformAndClassify]	4.36E+008
50	ParamSearch [[[RankingFScore], FeatureSelection], [SVMClassifier [KernelProvider]], TransformAndClassify]	4.36E+008
28	ParamSearch [kNN (Euclidean)]	4.52E+008
32	ParamSearch [SVMClassifier [KernelProvider]]	8.04E+008
35	ParamSearch [LVQ, kNN (Euclidean)]	9.46E+008
45	ParamSearch [[[RankingCC], FeatureSelection], [Boosting (10x) [NBC]], TransformAndClassify]	1.30E+009
54	ParamSearch [[[RankingFScore], FeatureSelection], [Boosting (10x) [NBC]], TransformAndClassify]	1.30E+009

optimistic. A survey of different diagrams (in Figure 16–23) easily brings the conclusion that the amount of inaccurately predicted time complexity is quite small (there are quite few broken bars). Note that, when a task is broken, its subtasks, that have already been computed are not recalculated during the test-task restart (due to the machine unification mechanism and machine cache). At the bottom, the *Time line* axis can be seen. The scope of the time is the interval $[0, 1]$ to show the times relative to the start and the end of the whole MLA computations. To make the diagrams clearer, the tests were performed on a single CPU, so only one task was running at a time and we can not see any two bars overlapping in time. If we ran the projects on more than one CPU, a number of bars would be "active" at almost each time, which would make reading the plots more difficult.

The simplest tasks are started first. They can be seen at the bottom of the plot. Their bars are very short, because they required relatively short time to be calculated. The higher in the diagram (i.e. the larger predicted complexity), the longer bars can be seen. It confirms the adequacy of the complexity estimation framework, because the relations between the predictions correspond very good to the relations between real time consumed by the tasks. When browsing other diagrams a similar behavior can be observed—the simple tasks are started at the beginning and then, the more and more complex ones.

On the left side of the *Task-id* column, the accuracies of classification test tasks and their approximated complexities are presented. At the bottom, there is the the *Accuracy* axis with interval from 0 (on the right) to 1 (on the left side). Each test task has its own gray bar starting at 0 and finished exactly at the point corresponding to the accuracy. So the accuracies of all the tasks are easily visible and comparable. Longer bars show higher accuracies. However remember that the experiments were not tuned to obtain the best accuracies possible, but to illustrate the behavior of the complexity controlled meta-learning and the generators flows.

The leftmost column of the diagram presents ranks of the test tasks (the ranking of the accuracies). In the case of the vowel data, the machine of the best performance is the kNN machine (the task id is 1 and the accuracy rank is 1 too) ex equo with kNN [MetricMachine (EuclideanOUO)] (task id 2). The second rank was achieved by kNN with Mahalanobis metric which is a more complex task.

Between the columns of *Task-id* and the accuracy-ranks, on top of the gray bars corresponding to the accuracies, some thin solid lines can be seen. The lines start at the right side as the accuracy bars and go to the right according to proper magnitudes. For each task, the three lines correspond to the total complexity (the upper line), the memory complexity (the middle line) and the time complexity (the lower line)[7]. All three complexities are the approximated complexities (seeEq. 6 and 7). Approximated complexities presented on the left side of the diagram can be easily compared visually to the time-schedule obtained in the real time on the right side of the diagram. Longer lines mean higher complexities. The longest line is spread to maximum width. The others are proportionally shorter. So the complexity lines at the top of the diagram are

[7] In the case of time complexity the $t/\log t$ is plotted, not the time t itself.

long while the lines at the bottom are almost invisible. It can be seen that sometimes the time complexity of a task is smaller while the total complexity is larger and vice versa. For example see tasks 42 and 48 again in Figure 16.

The meta-learning illustration diagrams (Figures 16–23) clearly show that the behavior of different machines changes between benchmarks. Even the standard deviation of accuracies is highly diverse. When looking at accuracies within some test, groups of machine of similar accuracy may be seen, however for other benchmark, within the same group of machines the accuracies are very variant. Of course the complexity of a test task for given configuration may change significantly from benchmark to benchmark. However it can be seen that in the case of benchmarks of similar properties, the permutations of task ids in the diagrams are partially similar (e.g. see the bottoms of Figures 21 and 23).

The **most important feature** of the presented MLA is that it facilitates finding accurate solutions in the order of increasing complexity. Simple solutions are started before the complex ones, to maximize the probability that an accurate solution is found as soon as possible. It is confirmed by the diagrams in Figures 16–23. Thanks to this property, in the case of a strong stop-condition (significant restriction on the running time) we are able to find really good solution(-s) because of starting test tasks in proper order. Even if some tasks get broken and restarted, it is not a serious hindrance to the main goal of algorithm.

For a few of the benchmarks, very simple and accurate models were found just at the beginning of the meta-learning process. Please see Figure 16 task ids 1 and 2, Figure 17 task ids 1 and 2, Figure 19 task ids 1 and 2, Figure 22 task ids 4 and 31, Figure 23 task id 19. The machines of the first four diagrams, are all single machines of relatively low complexities. But not only single machines may be of small complexity. The most accurate machine (of the 54 machines being analysed) for the thyroid data is the combination of feature selection based on F-score with kNN machine (task id 19). Even nontrivial combinations of machines (complex structures) may provide low time and memory complexity while single machine do not guarantee small computational complexity. In the case of very huge datasets (with huge number of instances and features) almost no single algorithm works successfully in rational time. However classifiers (or approximators) preceded by not too complex data transformation algorithms (like feature selection or instance selection) may be calculated in quite short time. The transformations may reduce the costs of classifier learning and testing, resulting in significant decrease of the overall time/memory consumption.

In some of the benchmarks (see Figures 18, 20 and 21) the most accurate machine configurations were not of as small complexity as in the cases mentioned above. For the german-numeric benchmark, the best machines are SVM's with linear and Gaussian kernels (task ids 6, 33[8] and 5). The winner machines, for

[8] Note that 33 means ParamSearch [LinearSVMClassifier [LinearKernelProvider]] where a linear SVM is nested within a ParamSearch, but the auto-scenario for linear SVM is empty, which means that ParamSearch machine does not optimize anything and indeed it is equivalent to the linear SVM itself. The small difference is a result of additional memory costs for ParamSearch encapsulation.

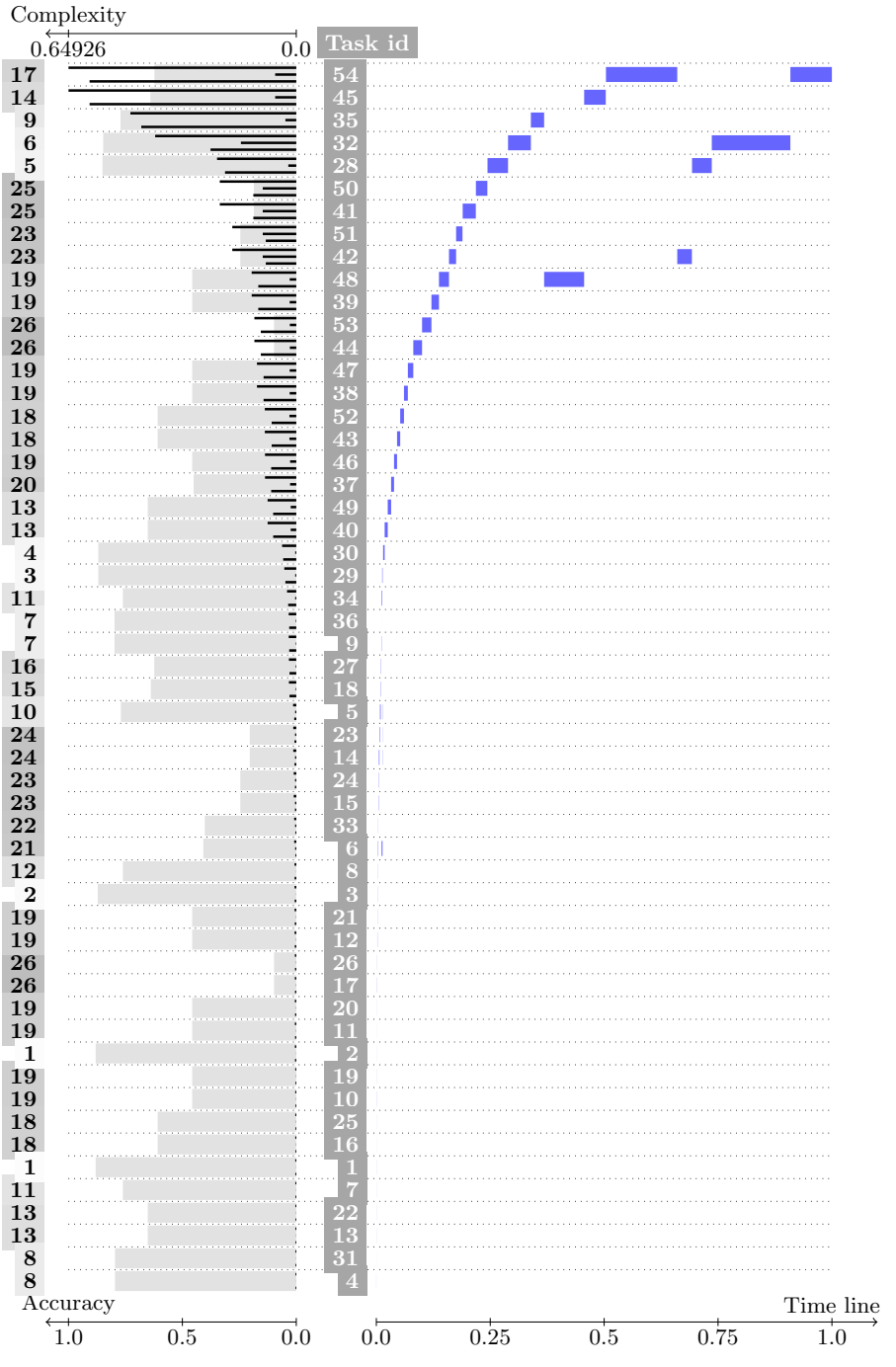

Complexity
0.64926
0.0
Task id
17 14 9 6 5 25 25 23 23 19 19 26 26 19 19 18 18 19 20 13 13 4 3 11 7 7 16 15 10 24 24 23 23 22 21 12 2 19 19 26 26 19 19 1 19 19 18 18 1 11 13 13 8 8
54 45 35 32 28 50 41 51 42 48 39 53 44 47 38 52 43 46 37 49 40 30 29 34 36 9 27 18 5 23 14 24 15 33 6 8 3 21 12 26 17 20 11 2 19 10 25 16 1 7 22 13 31 4
Accuracy
1.0 0.5 0.0
Time line
0.0 0.25 0.5 0.75 1.0

Fig. 16. vowel

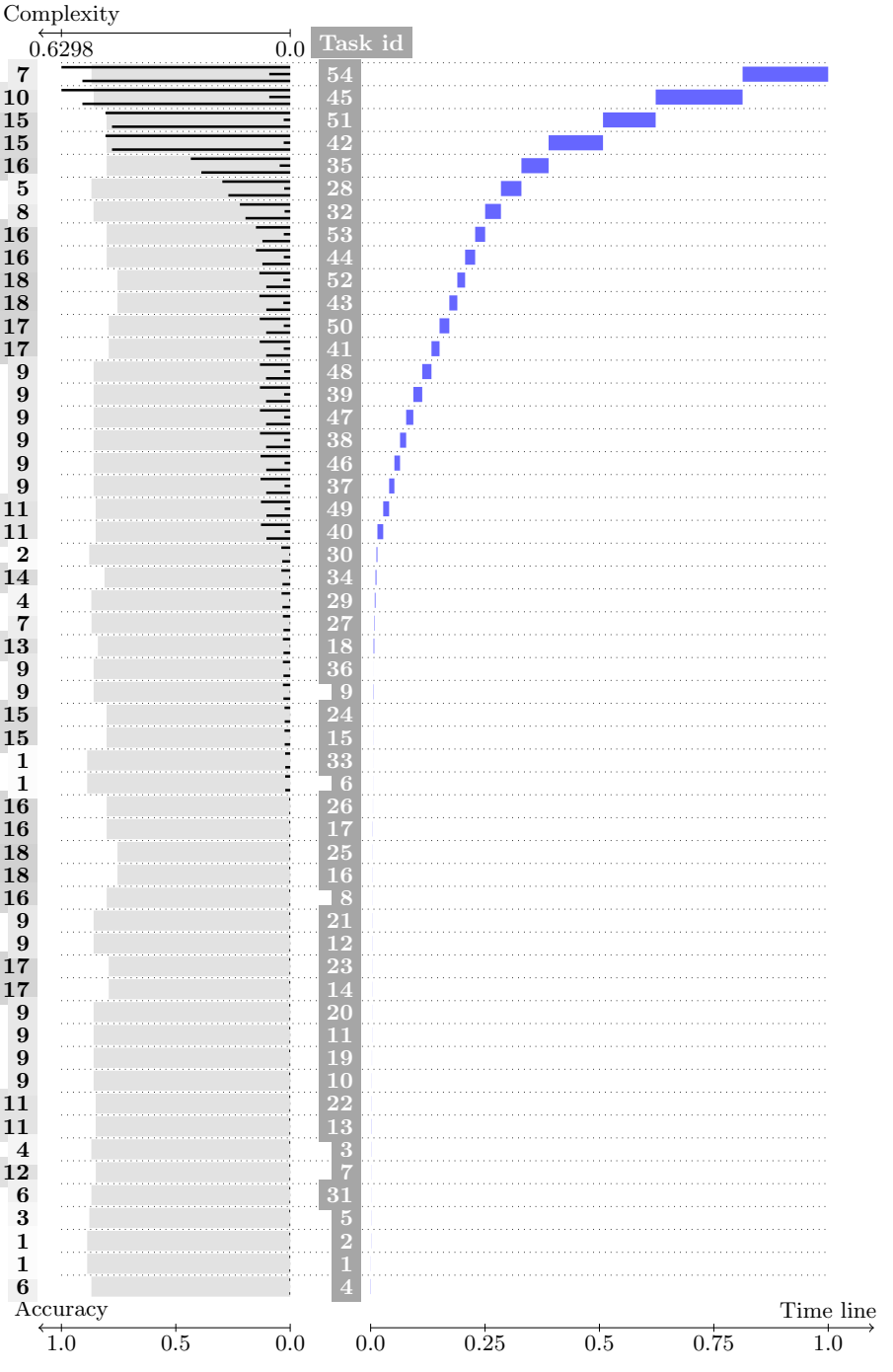


Fig. 17. appendicitis

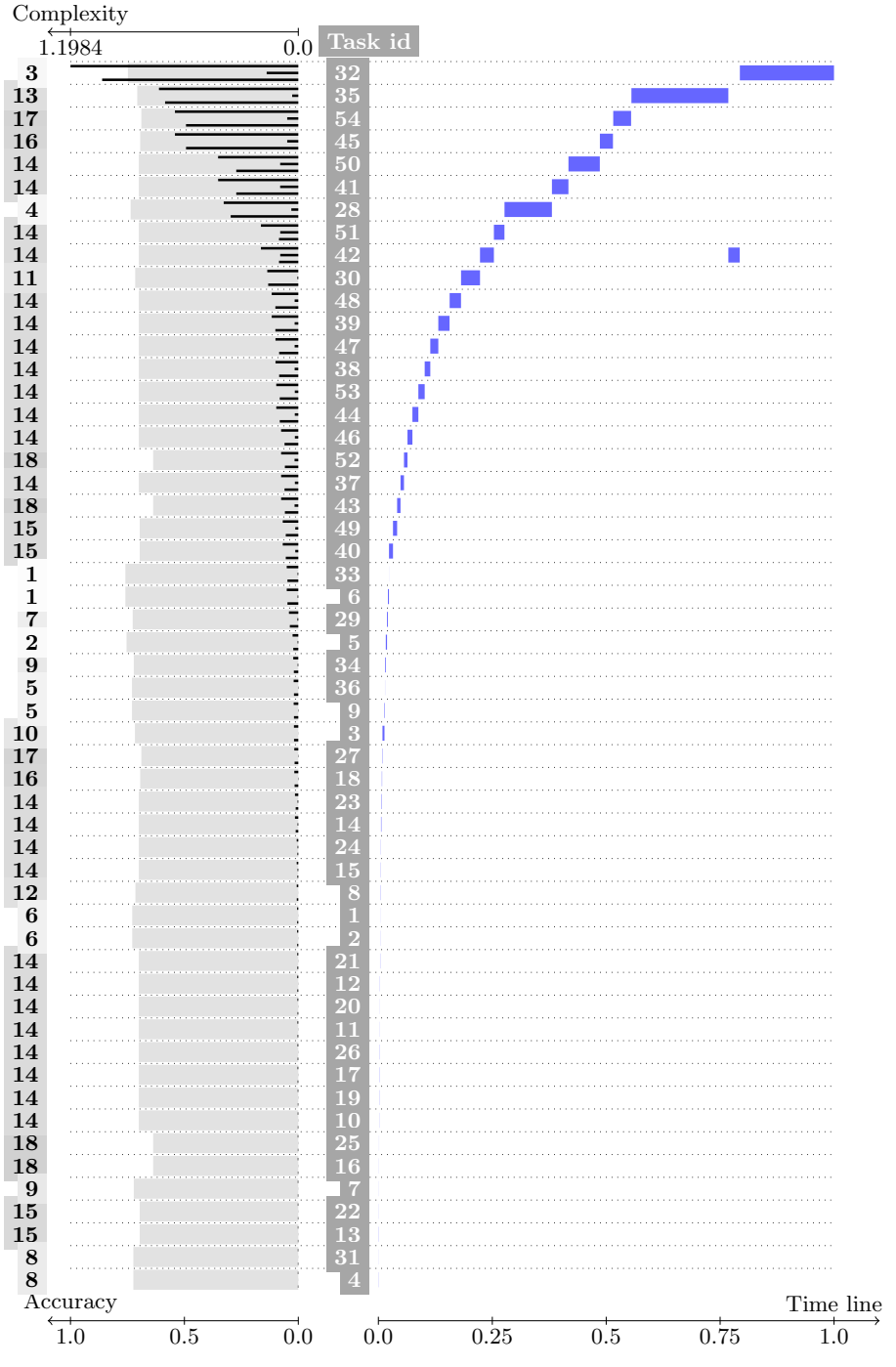


Fig. 18. german-numeric

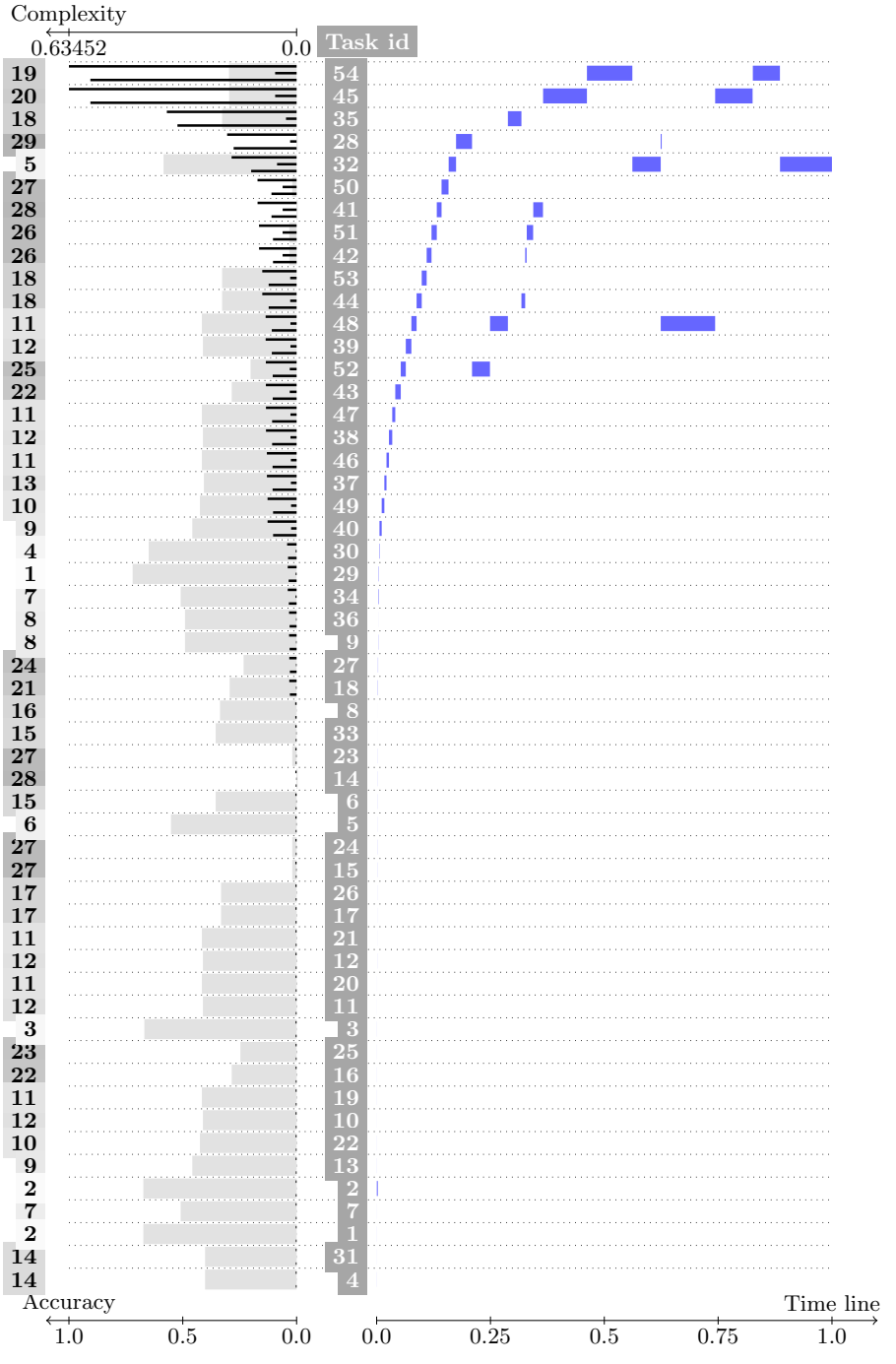

Complexity
0.63452
0.0
Task id
19 54
20 45
18 35
29 28
5 32
27 50
28 41
26 51
26 42
18 53
18 44
11 48
12 39
25 52
22 43
11 47
12 38
11 46
13 37
10 49
9 40
4 30
1 29
7 34
8 36
8 9
24 27
21 18
16 8
15 33
27 23
28 14
15 6
6 5
27 24
27 15
17 26
17 17
11 21
12 12
11 20
12 11
3 3
23 25
22 16
11 19
12 10
10 22
9 13
2 2
7 7
2 1
14 31
14 4
Accuracy
1.0
0.5
0.0
Time line
0.0
0.25
0.5
0.75
1.0

Fig. 19. glass

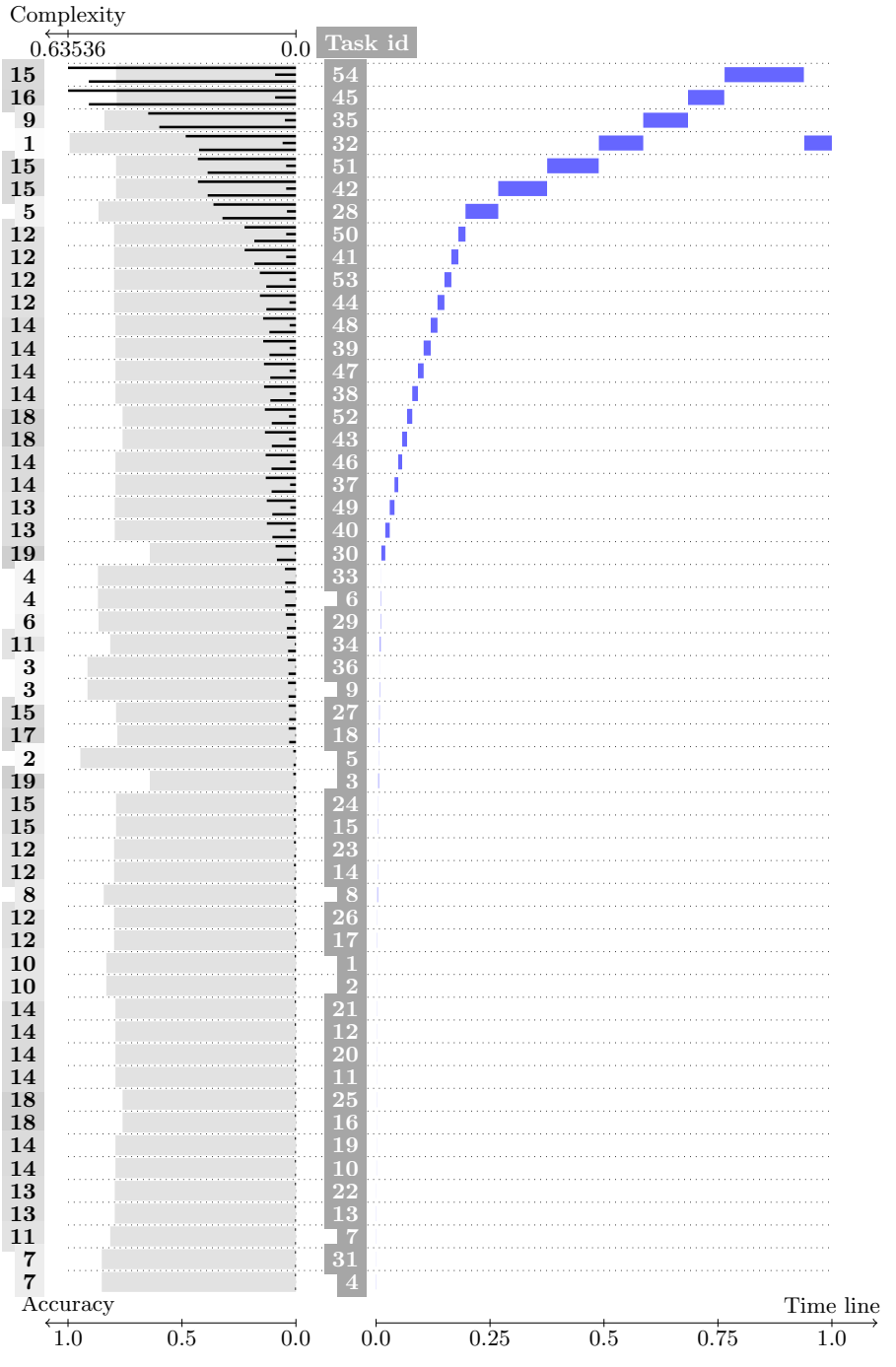


Fig. 20. ionosphere-ALL

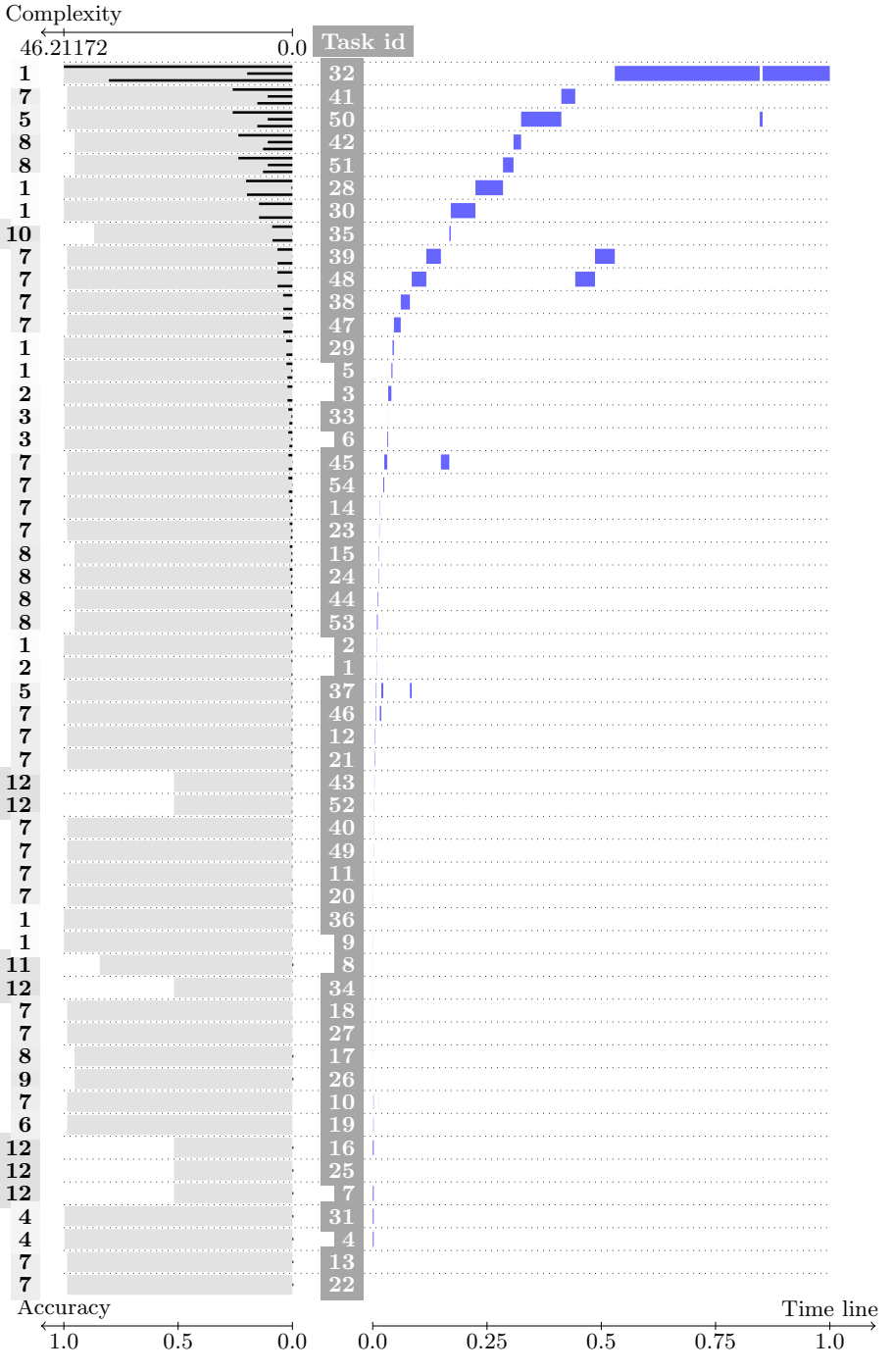


Fig. 21. mushroom

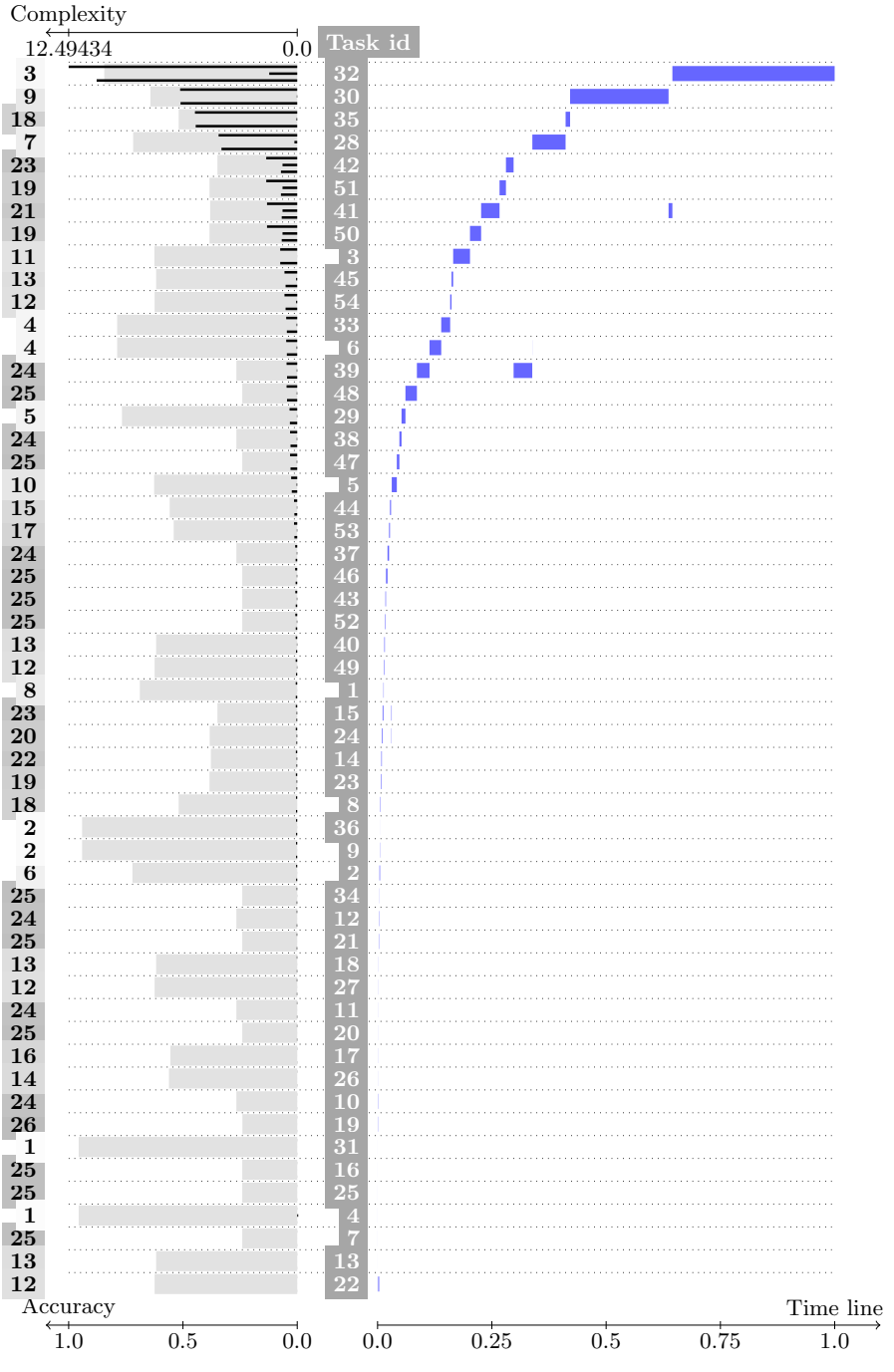


Fig. 22. splice

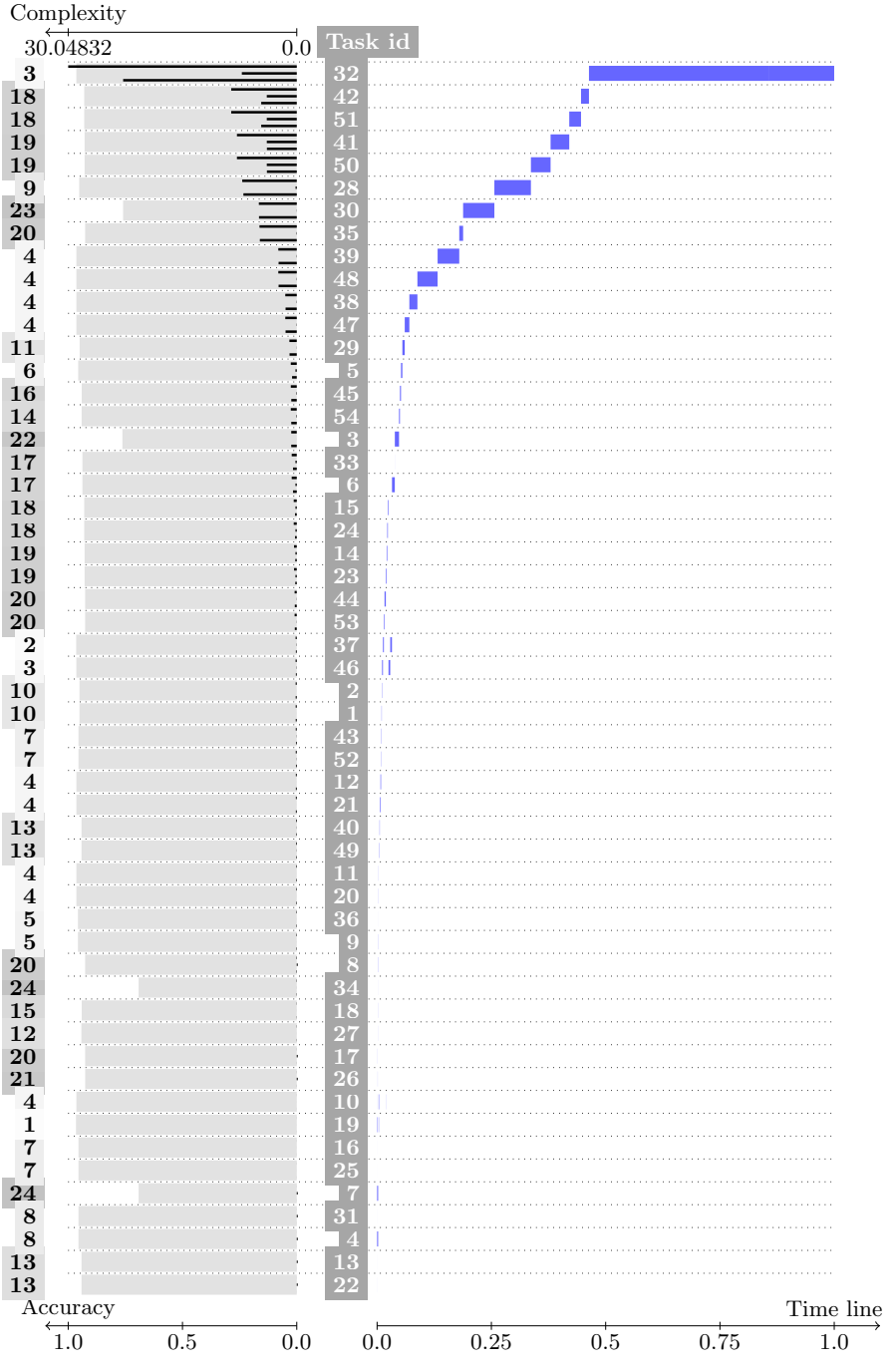


Fig. 23. thyroid-all

this benchmark, are of average complexity and are placed in the middle of the diagram. For the ionosphere benchmark, the most accurate machine is the SVM with Gaussian kernel but nested in a ParamSearch machine tuning the SVM parameters. This is one of the most complex test tasks. The MLA running on the mushroom data, has found several alternative configurations of very good performance: the simplest is a boosting of naive bayes classifier (task id 9), the second is the kNN [MetricMachine (EuclideanOUO)], followed by SVM (with Gaussian kernel), ParamSearch [kNN [MetricMachine (EuclideanOUO)]], other two configuration of kNN and ParamSearch [SVMClassifier [KernelProvider]].

Naturally, in most cases, more optimal machine configuration may be found, when using more sophisticated configuration generators and larger sets of classifiers and data transformations (for example adding decision trees, instance selection methods, feature aggregation, etc.) and performing deeper parameter search.

Note that the approximated complexity time is not in perfect compatibility with real time presented on the right side of the diagrams. The differences are due to not only the approximation inaccuracy, but also the machine unifications and some deviations in real CPU time consumption which sometimes is different even for two runs of the same task (probably it is caused by the .Net kernel, for example by garbage collection which, from time to time, must use the CPU to perform its own tasks).

Without repeating the experiments, one can think of the results obtained with the stop criterion set to a time-limit constraint. For example, assume that the time limit was set to 1/5 of the time really used by given MLA run. In such a case, some of the solutions described above would not be reached, but still, for several datasets the optimal solutions would be obtained and for other benchmarks, some slightly worse solutions would be the winners. This feature is crucial, because in real life problems the time is always limited and we are interested in finding as good solutions as possible within the time limits.

8 Future and Current Research

The next step toward more sophisticated meta-learning is to design advanced machine configuration generators, able to use and gather meta-knowledge of different kinds. We have already started some efforts in this direction. For example, we are working on using meta-knowledge to advise composition of complex machines and to get rid of ineffective machine combinations. Meta-knowledge will also help produce and advise new data transformations and check their influence on the results of the meta-search. Advanced generators can learn and collect experience from information about the most important events of the meta-search (starting, stopping and breaking test tasks). Using a number of specialized machine generators will facilitate composition of a variety of machine configurations, while enabling *smart* control over the generators results, by means of the part of complexity definition, responsible for machine attractiveness.

9 Summary

The meta-learning algorithm presented in the paper opens new gates of computational intelligence. The algorithm may be used to solve problems of various types, in the area of computational intelligence. Defining the goal is every flexible and may fulfill different requirements.

Also the search space, browsed during meta-learning is defined by means of a general mechanism of generators flow, which enables defining different combinations of base machines in a flexible way. As a result, the meta-learning searches among simple machines, compositions of transformers and classifiers or approximators, and also among more complex structures. It means that we look for more and more optimal combinations of transformations and final classifiers or approximators. What's more, this meta-learning is able to find independent (more optimal) transformations for different classifiers and then, use the complex models in committees.

The criterion of choosing the best model may be defined up to the needs, thanks to the query system. The focus may be put on accuracy, balanced accuracy or some result of a more complex statistical tests.

The most important job is made by the complexity control module which organizes the order of test task analysis in the loop of meta-learning. In most cases, the complexities are learned by approximation techniques. This approach may and be used to any type of machines in the system. Even the machines that will be added in future, may work as well with the scheme of complexity control. The biggest advantage of complexity based test task order is its independence of particular problem and used generators flow. Without such a mechanism, meta-learning is condemned to a serious danger of yielding no results because of starting a long lasting test task, which can not be finished in available time.

There is no problem to search for solutions among different complex machine structures exploiting feature selection algorithms, instance selection algorithms, other data transformation methods, classification machines, approximation machines, machine committees etc. MLAs do not need to know much about the nature of different machine components, so as to be able to run the tasks from the simplest to the most time and memory consuming. They can not *loose* simple and accurate solution, even when they are given little time for the search.

Proposed methodology allows to collect meta-knowledge and use it in further work (of the same MLA or other MLAs). Complexity estimation may be augmented, in a variety of ways, by defining corrections based on the knowledge gained during meta-learning.

Presented MLAs are able to autonomously and effectively search through functional model spaces for close to optimal solutions which sometimes are simple and sometimes really complex. They present very universal and powerful tools for solving really non-trivial problems in computational intelligence.

References

1. Jankowski, N., Grąbczewski, K.: Learning machines. In: Guyon, I., Gunn, S., Nikravesh, M., Zadeh, L. (eds.) Feature extraction, foundations and applications, pp. 29–64. Springer, Heidelberg (2006)
2. Guyon, I.: Nips 2003 workshop on feature extraction, `http://www.clopinet.com/isabelle/Projects/NIPS2003` (2003)
3. Guyon, I., Gunn, S., Nikravesh, M., Zadeh, L.: Feature extraction, foundations and applications. Springer, Heidelberg (2006)
4. Guyon, I.: Performance prediction challenge (2006), `http://www.modelselect.inf.ethz.ch`
5. Chan, P., Stolfo, S.J.: On the accuracy of meta-learning for scalable data mining. Journal of Intelligent Information Systems 8, 5–28 (1996)
6. Prodromidis, A., Chan, P.: Meta-learning in distributed data mining systems: Issues and approaches. In: Kargupta, H., Chan, P. (eds.) Book on Advances of Distributed Data Mining. AAAI Press, Menlo Park (2000)
7. Todorovski, L., Dzeroski, S.: Combining classifiers with meta decision trees. Machine Learning Journal 50, 223–249 (2003)
8. Duch, W., Itert, L.: Committees of undemocratic competent models. In: Kaynak, O., Alpaydın, E., Oja, E., Xu, L. (eds.) ICANN 2003 and ICONIP 2003. LNCS, vol. 2714, pp. 33–36. Springer, Heidelberg (2003)
9. Jankowski, N., Grąbczewski, K.: Heterogenous committees with competence analysis. In: Nedjah, N., Mourelle, L., Vellasco, M., Abraham, A., Köppen, M. (eds.) Fifth International conference on Hybrid Intelligent Systems, Brasil, Rio de Janeiro, pp. 417–422. IEEE, Computer Society, Los Alamitos (2005)
10. Pfahringer, B., Bensusan, H., Giraud-Carrier, C.: Meta-learning by landmarking various learning algorithms. In: Proceedings of the Seventeenth International Conference on Machine Learning, pp. 743–750. Morgan Kaufmann, San Francisco (2000)
11. Brazdil, P., Soares, C., da Costa, J.P.: Ranking learning algorithms: Using IBL and meta-learning on accuracy and time results. Machine Learning 50, 251–277 (2003)
12. Bensusan, H., Giraud-Carrier, C., Kennedy, C.J.: A higher-order approach to meta-learning. In: Cussens, J., Frisch, A. (eds.) Proceedings of the Work-in-Progress Track at the 10th International Conference on Inductive Logic Programming, pp. 33–42 (2000)
13. Peng, Y., Falch, P., Soares, C., Brazdil, P.: Improved dataset characterisation for meta-learning. In: The 5th International Conference on Discovery Science, pp. 141–152. Springer, Luebeck (2002)
14. Kadlec, P., Gabrys, B.: Learnt topology gating artificial neural networks. In: IEEE World Congress on Computational Intelligence, pp. 2605–2612. IEEE Press, Los Alamitos (2008)
15. Smith-Miles, K.A.: Towards insightful algorithm selection for optimization using meta-learning concepts. In: IEEE World Congress on Computational Intelligence, pp. 4117–4123. IEEE Press, Los Alamitos (2008)
16. Kolmogorov, A.N.: Three approaches to the quantitative definition of information. Prob. Inf. Trans. 1, 1–7 (1965)
17. Li, M., Vitányi, P.: An Introduction to Kolmogorov Complexity and Its Applications. In: Text and Monographs in Computer Science. Springer, Heidelberg (1993)
18. Jankowski, N.: Applications of Levin's universal optimal search algorithm. In: Kącki, E. (ed.) System Modeling Control 1995, vol. 3, pp. 34–40. Polish Society of Medical Informatics, Lódź (1995)

19. Bishop, C.M.: Neural Networks for Pattern Recognition. Oxford University Press, Oxford (1995)
20. Duda, R.O., Hart, P.E., Stork, D.G.: Pattern Classification, 2nd edn. Wiley, Chichester (2001)
21. Hastie, T., Tibshirani, R., Friedman, J.: The Elements of Statistical Learning: Data Mining, Inference, and Prediction. Springer Series in Statistics. Springer, Heidelberg (2001)
22. Rissanen, J.: Modeling by shortest data description. Automatica 14, 445–471 (1978)
23. Mitchell, T.: Machine learning. McGraw-Hill, New York (1997)
24. Werbose, P.J.: Beyond regression: New tools for prediction and analysis in the bahavioral sciences. PhD thesis. Harvard Univeristy, Cambridge, MA (1974)
25. Cover, T.M., Hart, P.E.: Nearest neighbor pattern classification. Institute of Electrical and Electronics Engineers Transactions on Information Theory 13, 21–27 (1967)
26. Boser, B.E., Guyon, I.M., Vapnik, V.: A training algorithm for optimal margin classifiers. In: Haussler, D. (ed.) Proceedings of the 5th Annual ACM Workshop on Computational Learning Theory. ACM Press, Pittsburgh (1992)
27. Vapnik, V.: Statistical Learning Theory. Wiley, Chichester (1998)
28. Grąbczewski, K., Duch, W.: The Separability of Split Value criterion. In: Proceedings of the 5th Conference on Neural Networks and Their Applications, Zakopane, Poland, pp. 201–208 (2000)
29. Grąbczewski, K., Jankowski, N.: Saving time and memory in computational intelligence system with machine unification and task spooling. Knowledge-Based Systems, 30 (2011) (in print)
30. Jankowski, N., Grąbczewski, K.: Increasing efficiency of data mining systems by machine unification and double machine cache. In: Rutkowski, L., Scherer, R., Tadeusiewicz, R., Zadeh, L.A., Zurada, J.M. (eds.) ICAISC 2010. LNCS, vol. 6113, pp. 380–387. Springer, Heidelberg (2010)
31. Grąbczewski, K., Jankowski, N.: Task management in advanced computational intelligence system. In: Rutkowski, L., Scherer, R., Tadeusiewicz, R., Zadeh, L.A., Zurada, J.M. (eds.) ICAISC 2010. LNCS, vol. 6113, pp. 331–338. Springer, Heidelberg (2010)
32. Jankowski, N., Grąbczewski, K.: Gained knowledge exchange and analysis for meta-learning. In: Proceedings of International Conference on Machine Learning and Cybernetics, Hong Kong, China, pp. 795–802. IEEE Press, Los Alamitos (2007)
33. Grąbczewski, K., Jankowski, N.: Meta-learning architecture for knowledge representation and management in computational intelligence. International Journal of Information Technology and Intelligent Computing 2, 27 (2007)
34. Merz, C.J., Murphy, P.M.: UCI repository of machine learning databases (1998), `http://www.ics.uci.edu/~simmlearn/MLRepository.html`
35. Kohonen, T.: Learning vector quantization for pattern recognition. Technical Report TKK-F-A601, Helsinki University of Technology, Espoo, Finland (1986)

Meta-Learning of Instance Selection for Data Summarization

Kate A. Smith-Miles[1] and Rafiqul M.D. Islam[2]

[1] School of Mathematical Sciences, Monash University, VIC 3800, Australia
kate.smith-miles@sci.monash.edu.au
[2] School of Information Technology, Deakin University, Burwood VIC 3125, Australia
rafiq@deakin.edu.au

Abstract. The goal of instance selection is to identify which instances (examples, patterns) in a large dataset should be selected as representatives of the entire dataset, without significant loss of information. When a machine learning method is applied to the reduced dataset, the accuracy of the model should not be significantly worse than if the same method were applied to the entire dataset. The reducibility of any dataset, and hence the success of instance selection methods, surely depends on the characteristics of the dataset. However the relationship between data characteristics and the reducibility achieved by instance selection methods has not been extensively tested. This chapter adopts a meta-learning approach, via an empirical study of 112 classification datasets, to explore the relationship between data characteristics and the success of a naïve instance selection method. The approach can be readily extended to explore how the data characteristics influence the performance of many more sophisticated instance selection methods.

Keywords: instance selection, meta-learning, data summarization, data reduction, classification, data characteristics, clustering.

1 Introduction

Classification problems in the real world, such as classifying likely taxation fraud in the large-scale databases of national taxation agencies, frequently operate on a completely different scale to the classification problems that most researchers use to develop and test their algorithms, such as the UCI Repository [1]. When dealing with such large-scale datasets, it is often a practical necessity to seek to reduce the size of the dataset, acknowledging that in many cases the patterns that are in the data would still exist if a representative subset of instances were selected. Further, if the right instances are selected, the reduced dataset can often be less noisy than the original dataset, producing superior generalization performance of classifiers trained on the reduced dataset. The goal of *instance selection* is to select such a representative subset of instances, enabling the size of the new dataset to be significantly reduced without

N. Jankowski et al. (Eds.): Meta-Learning in Computational Intelligence, SCI 358, pp. 77–95.
springerlink.com

compromising the accuracy of a particular classification algorithm, and cleaning it of noisy data that may otherwise affect the performance of classification algorithms [2].

There have been many instance selection methods proposed over the last four decades or so [2-4], the broad classes of which will be reviewed in Section 2. With the existence of so many different approaches and algorithms for instance selection though, it is natural to wonder which method is likely to perform best for a given dataset. The No-Free-Lunch Theorem [5] suggests that it is unlikely that there is a single method that will perform best on all datasets regardless of their characteristics or properties. Indeed, the comparison on instance selection algorithms performed by Grochowski and Jankowski [6] confirms that even the average performance of instance selection methods across a group of 6 UCI classification problems varies considerably, and also depends on which classification algorithm is applied to the reduced dataset. Reinartz [4] provides a brief summary of some experiments performed to explore how the performance of various simple sampling methods varied with changing data characteristics. Each dataset is characterized by coarse and qualitative descriptions such as whether the number of instances and the number of attributes are few or many, whether there are many classes, and whether there are more qualitative or quantitative attributes. Such characteristics are not quantified precisely. Each sampling method is scored against these characteristics based on whether they performed well or poorly on datasets with those characteristics. Reinartz [4] acknowledged the need for more extended studies to "understand the relation between different instance selection techniques and to come up with reliable heuristics and guidelines … given a specific data mining environment". The data mining environment naturally needs to consider both the nature of the dataset (its characteristics or features) and the particular algorithms that are to be applied to the reduced dataset.

Reinartz's identification of this open issue prompts the approach taken in this chapter. We ask the same question about suitability of various instance selection methods, but place it in the context of the Algorithm Selection Problem [7, 8], adopting a meta-learning approach. Just as meta-learning research has helped to identify which classification algorithm should be used for a certain dataset [9-12], we can extend the concepts here to explore the suitability of instance selection algorithms. In fact, adopting a meta-learning approach to assist with the design of many elements of the data mining process is a useful endeavor [13-15]. We present a methodology that can be used to learn the relationship between the characteristics of the data and the performance of instance selection methods. We explore the statistical properties of a classification datasets that enable a significant reduction in the size of the training data to be achieved without compromising classification accuracy.

The aim of this chapter is thus to demonstrate how a meta-learning approach can be used to identify which datasets lend themselves to reducibility. We consider an extensive set of 112 classification problems from the UCI repository [1], and characterize each dataset with a clear set of statistical metrics. Using a naïve instance selection method together with a Naïve Bayes classifier, we evaluate the reducibility that can be achieved for each dataset, by finding the smallest subset of the training data that enables the accuracy to be not significantly worse than the accuracy obtained

with a Naïve Bayes classifier on the original training data. The relationship between the data characteristics and the performance of this naïve instance selection approach is then explored using both supervised and unsupervised learning methods. The proposed methodology can be readily extended to consider other instance selection methods from the major classes shown in Figure 1, and other classifiers.

The remainder of this chapter is as follows: Section 2 discusses the instance selection problem in more detail, and reviews some of the existing approaches. The naïve method used in this chapter is presented in the context of these other methods. Section 3 describes how a meta-learning framework can be developed for algorithm selection, and to gain insights into the relationship between problem features and characteristics and the suitability of various algorithms. Section 4 presents the methodology used in this study, including a description of the datasets used, and how their features are measured. Section 5 presents the experimental results, where the relationships in the meta-data are learned using both supervised and unsupervised learning approaches. Section 6 concludes the chapter with a summary of the insights generated by this meta-learning process, and identifies opportunities to extend these ideas for future research.

2 Instance Selection Methods

Jankowski and Grochowski [3] classify instance selection approaches into three main types: i) noise filters [16-18] are decremental algorithms which remove instances whose class labels do not agree with the majority of their neighbours; ii) condensation algorithms [19-23] are typically incremental algorithms that add instances from the training data to a new dataset if they add new information, but not if they have the same class label as their neighbours; iii) prototype construction methods [24-26] do not focus on selecting which instances of the training data to include in the reduced dataset, but create new instances which are representative of the whole dataset via data squashing [27] or clustering methods [28, 29].

According to Reinartz's unifying view on instance selection [4] the first two types of instance selection methods (noise filters and condensation algorithms) can also be considered prototype selection methods (deciding which instances in the training data to include in the reduced dataset, using either incremental or decremental methods), while the third type are basically prototype construction approaches which seek to find new instances that can represent the whole dataset more compactly. Added to the group of prototype selection methods are those based on random sampling [30, 31] which randomly select instances at first and then identify instances to swap based on goodness measures.

Figure 1 provides a taxonomic summary of the related literature and the various approaches to instance selection.

There are other instance selection methods which combine elements of clustering and prototype selection. Leader sampling [4] identifies prototypes (leaders) based on clustering, and these prototypes represent a set of instances that are close enough to the leader (within a similarity threshold). New leaders are identified to represent any instances which are not close enough to a leader.

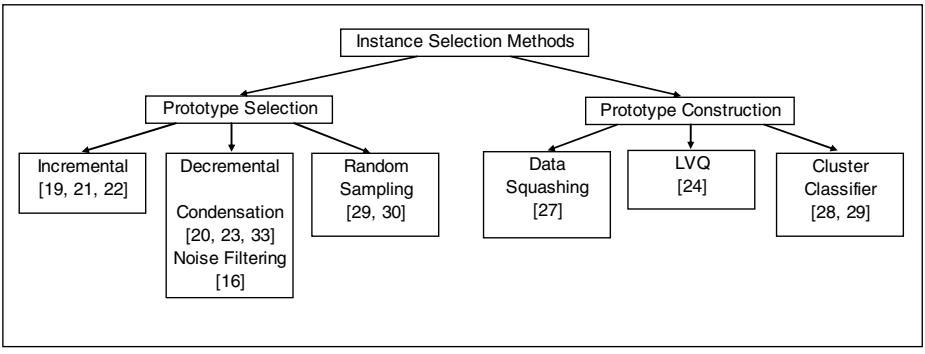


Fig. 1. Taxonomic summary of various instance selection approaches

We adopt in this chapter another approach, related to leader sampling, but quite simpler. Prototype points (leaders) are identified through the k-means clustering algorithm [32]. The prototypes are not used for constructing new instances, but form the basis of a prototype selection process. From each cluster we select the closest (100-β)% of the cluster size measured as the Euclidean distance from the cluster centroid (leader). This is a form of stratified sampling based on the similarity of the instances, rather than the class labels, and thus is quite naïve since apriori knowledge about class probabilities is not being used. Of course, this strategy means that it is not being used as a noise filter based on class membership, and so it is closer to a condensation algorithm. The data reduction achieved is β%. We vary the value of β to explore the effectiveness of a classification algorithm on the reduced dataset, compared to the performance of the classification algorithm on the original dataset. Naturally, any of the instance selection methods discussed in this section could have been selected, but for the sake of demonstrating the meta-learning methodology, we have elected to focus on this naïve instance selection method working in partnership with a Naïve Bayes classifier. There is no doubt that many of the more sophisticated instance selection methods would yield improved accuracy for the classifier, but the point here is to explore how the performance on a given instance selection method varies with instance characteristics. The methodology is broadly applicable and extendable to other instance selection methods and classification algorithms.

3 Meta-Learning about Algorithm Performance

The No Free Lunch theorem [5] warns us against expecting a single algorithm to perform well on all classes of problems, regardless of their structure and characteristics. Instead we are likely to achieve better results, on average, across many different classes of problem, if we tailor the selection of an algorithm to the characteristics of the problem instance. This challenge is known as the Algorithm Selection Problem.

As early as 1976, Rice [7] proposed a framework for tackling the problem, consisting of four essential components:

- the problem space P represents the set of problems within a problem class;
- the feature space F contains measurable characteristics of the problems generated by a computational feature extraction process applied to P;
- the algorithm space A is the set of all considered algorithms for tackling the problem;
- the performance space Y represents the mapping of each algorithm to a set of performance metrics.

In addition, we need to find a mechanism for generating the mapping from feature space to algorithm space. The framework of Rice [7] is summarized in Figure 2. The Algorithm Selection Problem can be formally stated as: For a given problem $x \in P$, with features $f(x) \in F$, find the selection mapping $S(f(x))$ into algorithm space A, such that the selected algorithm $\alpha^* \in A$ maximizes the performance mapping $y(\alpha^*, x) \in Y$. The collection of data describing {P, A, Y, F} is known as the *meta-data*.

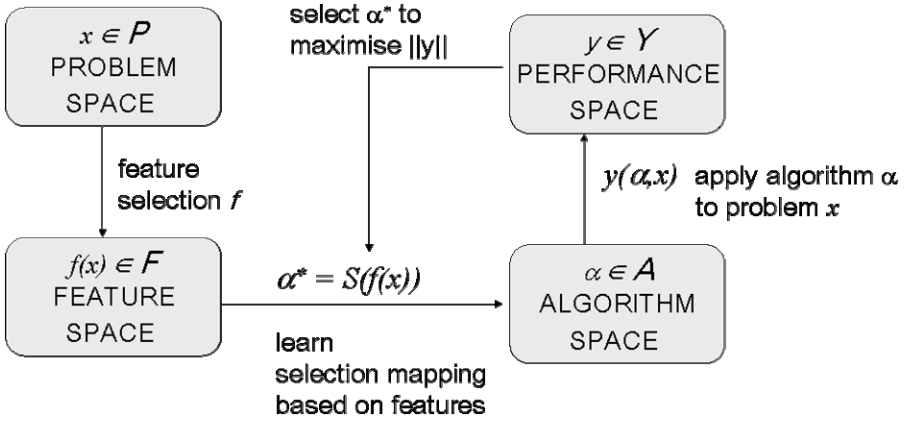


Fig. 2. Schematic diagram of Rice's [1976] Algorithm Selection Problem model.

There have been many studies in the broad area of algorithm performance prediction, which is strongly related to algorithm selection in the sense that supervised learning or regression models are used to predict the performance ranking of a set of algorithms, given a set of features of the problems. Unfortunately, a variety of terms have been used by various communities, and a recent survey paper [8] seeks to unify the efforts found in the machine learning, artificial intelligence, operations

research, statistics, and other communities, based on the common framework of Rice's model.

Of most relevance to this chapter is the algorithm selection work done by the machine learning community over the last 15 years or so. The term meta-learning [9] has been used, since the algorithms they are focused on are learning algorithms, with the aim of learning about learning algorithm performance. The meta-data used by meta-learning researchers, particularly during the two large European projects of STATLOG and METAL, can be defined using Rice's notation as: a set of classification problems (P), trained using typical machine learning classifiers such as decision trees, neural networks, or support vector machines (A), where supervised learning methods (S) are used to learn the relationship between the statistical and information theoretic measures of the classification problem (F) and the classification accuracy (Y). Studies of this nature include [10-12] to name only three of the many papers published over the last 15 years.

Clearly, Rice's framework of learning from meta-data generalizes to any domain where we have sufficient meta-data: where we have a choice of algorithms, with the performance of the algorithms clearly measurable, and a collection of datasets whose characteristics (features) can be measured in some way. The extension of meta-learning ideas beyond classification algorithm selection to broader goals, such as assisting with the selection of the optimal data mining process, is a useful pursuit [13-15]. In fact, when we refer to algorithm selection, we can readily substitute the word 'algorithm' with 'parameter' or 'process', and apply the same meta-learning ideas at a different scale.

Beyond the goal of performance prediction also lies the ideal of greater insight into algorithm performance, and very few studies have focused on methodologies for acquiring such insights. Instead the focus has been on selecting the best algorithm for a given problem (dataset), without consideration for what implications this has for algorithm design or insight into algorithm behaviour. Even if the algorithm set consist of only one algorithm (and therefore the algorithm selection problem is not relevant), there is value to be found in gathering meta-data about its performance across a range of problems, and seeking to learn under what conditions this single algorithm performs well or poorly. The goal of this chapter is to demonstrate that knowledge discovery processes can be applied to a rich set of meta-data to develop, not just performance predictions, but visual explorations of the meta-data and learned rules, with the goal of learning more about the dependencies of algorithm performance on problem structure and data characteristics.

4 Methodology

In this section we describe the experimental meta-data used for learning the relationships between classification problem features and the reducibility achieved by the naïve instance selection method coupled with a Naïve Bayes classifier. We also provide a description of the machine learning algorithms applied to the meta-data to produce rules and visualizations of these relationships.

4.1 Generating Meta-Data

Using the notation of Rice [7], the meta-data used in this study comprises a set of 112 classification problems (P) selected from the UCI Repository [1] (see Appendix A); the set of algorithms (A) in this study comprises the combination of the naïve instance selection method described in Section 2 implemented with various values of β, with each one followed by the Naïve Bayes classifier; the performance metric (Y) is the maximum percentage reduction in data size possible while maintaining a classification accuracy that is not statistically significantly worse than the accuracy obtained on the original (complete) dataset; and the set of features (F) used to characterize the classification problems comprises a set of statistical metrics described below.

Statistical Features

Each data set can be described by simple, distance and distribution-based statistical measures [34]. Let $X_{ik;j}$ be the value of the jth attribute (column) in the kth instance (row) of dataset i. These three types of measures characterise the data set in different ways. Firstly, the simple classical statistical measures identify the data characteristics based on attribute to attribute comparisons (i.e. comparisons between columns of the dataset). These include various measures of the centre of the data for each variable (geometric, harmonic and trim means, median) and spread (standard deviations, interquartile range, range) as well as skewness, kurtosis, and maximum and minimum eigenvalues, correlation coefficient, and z-score. Then, the distance based measures identify the data characteristics based on instance to instance comparisons (between rows of the data set). These include Euclidean, Mahalanobis and city-block distances. Finally, the density-based measures consider the relationships between single data points and the statistical properties of the entire data matrix to identify the data set features. Distributions of pdf and cdf based on χ^2, normal, binomial, discrete uniform, exponential, F, gamma, geometric, hypergeometric, lognormal, Poisson, Rayleigh and student t-test are all considered. All of these statistical measures, with complete formulae, have been summarised in [34]. The simple statistical measures are calculated within each column, and then averaged over all columns to obtain global measures of the data set. Likewise, the distance measures are averaged over all pair wise comparisons, and the density based measures are averaged across the entire matrix. For each dataset i, a total of 29 measures are calculated (11 statistical, 3 distance-based, 15 density-based). The data set feature matrix is then assembled with the columns comprising the 29 features, and the rows comprising the 112 datasets. A subset of 14 of these features were used in the analysis, and are shown in Table 1.

Algorithm Implementation

The algorithm used in this experimental study is a combination of the naïve instance selection method to build a reduced dataset coupled with the Naïve Bayes classifier. We start by reserving half of the available data for testing purposes. Based on the

training data (50%), we use the k-means algorithm to generate k clusters in feature space. For each cluster, we select only a subset of instances around the cluster centroid to create a reduced dataset. The reduced data set is used to train a Naïve Bayes classifier to predict class membership, and then the model is tested on the reserved 50% test set for accuracy. Experiments are conducted to determine, how much reduction can be achieved without degradation of the model accuracy compared to using all of the full training data. The details of the algorithm are as follows:

Step 1: Randomly divide original data set (containing N instances) into 50% for training (D_t) and 50% for testing or evaluation (D_e). Let the size of each dataset be denoted by $N_t=N_e=0.5N$;

Step 2: Cluster the training data (D_t) using the k-means algorithm, for a given value of k, selecting $1\leq k\leq N_t$. Cluster j contains N_j instances, for $1\leq j\leq k$;

Step 3: Identify each of the k cluster centroids as a "leader", and build a new dataset by selecting for each cluster the closest η_j instances (in Euclidean space) around leader j, where $\eta_j = N_j*(100-\beta)\%$, for a selected value of β. The new reduced dataset $D_r(\beta)$ is $\beta\%$ smaller than the original training data D_t;

Step 4: The reduced dataset $D_r(\beta)$ is used to train a Naïve Bayes classifier (using the default parameter settings in Weka [35])

Step 5: The evaluation dataset D_e (50% of the original data) is applied to the classifier, and the accuracy of the classification is recorded as $\gamma_r(\beta)$;

Step 6: Repeat Steps 3-5 for varying values of β ($0\leq\beta\leq100$ in steps of 10), recording the accuracy of the classifier when using the entire dataset D_t (when $\beta=0$) as γ;

Step 7: Identify the value of β, denoted by β^*, when the difference between γ and $\gamma_r(\beta)$ first becomes statistically significant (using a t-test with p=0.05). Clearly, this value of $\beta*$ depends on the value of k, but also on the features of the dataset.

Step 8: Repeat Steps 2-7 for varying values of k[1].

Based on an initial small study of randomly selected UCI repository problems, we found this approach to be quite robust to the value of k, with all problems following a similar contour when plotting how the test set accuracy varied with β (see Figure 3). Regardless of number of clusters (k-value), the point at which data reduction was no longer possible seems similar, but as expected, the rate of reduction in accuracy is faster for small numbers of clusters if the sampling around the cluster centre is more limited. The pattern shown in Figure 3 was similar for several randomly chosen datasets. Since we are only interested in identifying the smallest subset that retains the original accuracy of the model trained on all of the data, we arbitrarily set the number of clusters as 10% of the size of the training data, so that $k=0.1N_t$. Naturally, different values of k can be tested within this methodology though. Figure 4 shows the performance of several of the randomly selected subset of datasets, using $k=0.1N_t$. It is clear that even fixing the value of k based on the size of the dataset still reveals much differentiation in performance of the algorithm, and different optimal $\beta*$ values for

[1] This step could be eliminated with the use of a clustering method that does not require the number of clusters to be specified by the user, or using methods that seek to identify the number of natural clusters in a dataset [36].

each dataset due to the different characteristics of the datasets. It is the relationship between these characteristics and the kind of performance results we observe in Figure 4, extended across all 112 problems, that we now seek to learn based on the meta-data.

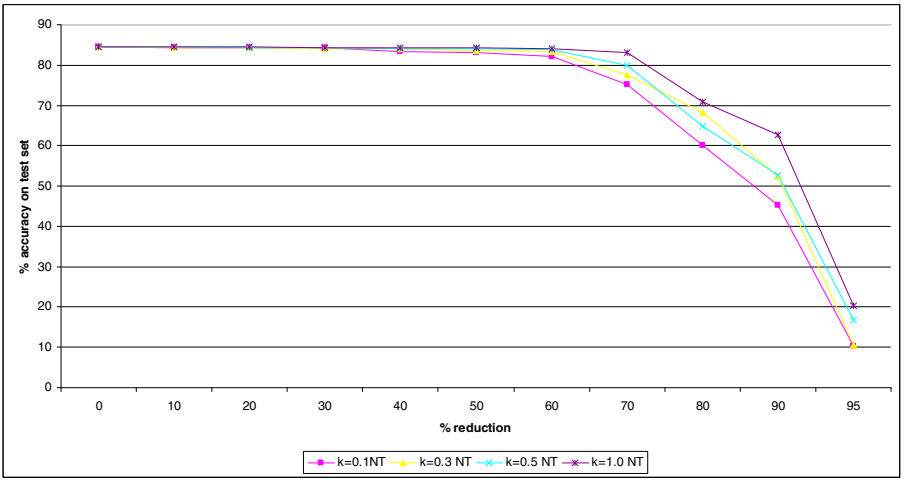


Fig. 3. Effect of the number of clusters on the algorithm performance.

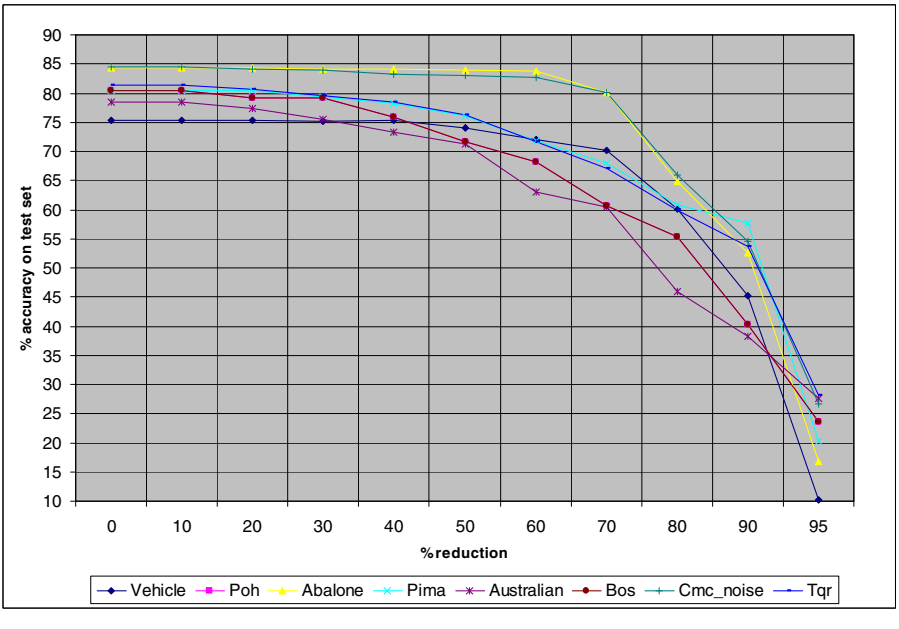


Fig. 4. Algorithm performance on a subset of problems using $k=0.1N_t$

4.2 Knowledge Discovery on the Meta-Data

When exploring any data-set to discover knowledge, there are two broad approaches. The first is supervised learning (aka directed knowledge discovery) which uses training instances – sets of independent attributes (inputs) and dependent attributes (outputs or classes in the case of classification problems) - to learn a predictive model which is then generalized for new instances to predict the dependent attribute (output) based only on the independent attributes (inputs). This approach is useful for building models to predict which algorithm or heuristic is likely to perform best given only the feature vector as inputs. The second broad approach to knowledge discovery is unsupervised learning (aka undirected knowledge discovery) which uses only the independent attributes to find similarities and differences between the structure of the instances, from which we may then be able to infer relationships between these structures and the dependent attribute (class membership). This second approach is useful for our goal of seeking greater insight into *why* certain algorithms (combinations of instance selection methods and classification algorithms) might be better suited to certain datasets, rather than just building predictive models of algorithm performance.

In this section we briefly summarise the methods we have used for knowledge discovery on the meta-data.

Correlation and Regression Analysis

In order to determine which of the 29 features [34] are most likely to be predictive of reducibility, we first perform a correlation analysis. Any feature with a correlation greater than 0.25 or less than -0.25 is selected in a reduced set of 14 features. These are: the number of instances; the harmonic mean; inter-quartile range; kurtosis; sample correlation coefficient; z score; Mahalanobis distance; probability density function for various distributions (Chi-squared, Normal, Binomial, Exponential, F, Rayleigh, and Student t distributions).

This subset of 14 features was then used as inputs to a multiple regression model, to establish a baseline predictive model, with the output being the percentage reducibility achieved for each dataset (β^*). An R^2 (coefficient of determination calculated as the ratio of the sum of squared residuals to the total sum of squares, subtracted from 1) value of 0.87 was achieved. The residual errors of the regression model can be added as an additional feature to characterize each dataset, similar to the concept of landmarking [37] whereby the performance of simple methods is used as a feature to predict the performance of more sophisticated algorithms. However, this feature should not be used as an input to any predictive model of algorithm performance, since it relies on knowing the actual reducibility of the algorithm in order to generate the residual error. This final feature can only be used for exploring the relationships in the data, rather than developing predictive models. The final set of 15 features available to the knowledge discovery process are shown in Table 1.

Table 1. 15 features used in meta-data, showing correlation with β*

Feature Description	Abbreviation	Correlation with β*
Number of Instances	N	0.8
Harmonic Mean	HM	-0.25
Inter-quartile Range	IQR	0.33
Kurtosis	KURT	0.34
Correlation Coefficient	CC	0.35
Z Score	Z	-0.25
Mahalanobis Distance	MD	0.25
Chi-squared pdf	C2pdf	-0.32
Normal pdf	Npdf	0.32
Binomial pdf	Bpdf	-0.26
Exponential pdf	Epdf	-0.29
F pdf	Fpdf	0.32
Rayleigh pdf	Rpdf	0.37
Student t pdf	Tpdf	0.33
Regression Residual Error	ResErr	0.49

Neural Networks

As a supervised learning method [38], neural networks can be used to learn to predict the data reduction capability (β*) of a dataset using a certain algorithm (instance selection method and classifier). In the case of multiple competing algorithms, the neural network can be used to predict the relative performance of the algorithms, thus solving the Algorithm Selection Problem [7] via supervised learning. A training dataset is randomly extracted (80% of the 112 problems) and used to build a non-linear model of the relationships between the input set (features F) and the output (metric Y). Once the model has been learned, its generalisation on an unseen test set (the remaining 20% of the datasets) is evaluated and recorded as percentage accuracy in predicting the performance of the algorithm. This process is repeated five times for different random extractions of the training and test sets, to ensure that the results were not simply an artifact of the random number seed. This process is known as five-fold cross validation, and the reported results show the average accuracy on the test set across these five folds.

For our experimental results, the neural network implementation within the Weka machine learning platform [35] was used with 14 input nodes (excluding ResErr), 18 hidden nodes, and a single output node. The transfer function for the hidden nodes was a sigmoidal function, and the neural network was trained with the backpropagation (BP) learning algorithm with learning rate = 0.3, momentum = 0.2. The BP algorithm stops when the number of epochs (complete presentation of all examples) reaches a maximum training time of 500 epochs or the error on the test set does not decrease after a threshold of 20 epochs.

Decision Tree

A decision tree [39] is also a supervised learning method, which uses the training data to successively partition the data, based on one feature at a time, into classes. The goal is to

find features on which to split the data so that the class membership at lower leaves of the resulting tree is as "pure" as possible. In other words, we strive for leaves that are comprised almost entirely of one class only. The rules describing each class can then be read up the tree by noting the features and their splitting points. Five-fold cross validation is also used in our experiments to ensure the generalisation of the rules.

In order to apply a decision (classification) tree to our problem, we first discretize the β^* values into three categories: LOW corresponding to a β^* value under 20% reducibility, HIGH corresponding to a β^* value greater than 40%, and MEDIUM corresponding to a β^* value between 20% and 40%. These bins were determined based on an examination of the frequency distribution to ensure a reasonable distribution of the datasets based on their relative reducibility. The J4.8 decision tree algorithm, implemented in Weka [35], was used for our experimental results based on the 14 features (excluding ResErr), with a minimum leaf size of 10 datasets. The generated decision tree is pruned using subtree raising with confidence factor = 0.25.

Self-Organizing Maps

Self-Organizing Maps (SOMs) are the most well-known unsupervised neural network approach to clustering. Their advantage over traditional clustering techniques such as the k-means algorithm lies in the improved visualization capabilities resulting from the two-dimensional map of the clusters. Often patterns in a high dimensional input space have a very complicated structure, but this structure is made more transparent and simple when they are clustered in a lower dimensional feature space. Kohonen [40] developed SOMs as a way of automatically detecting strong features in data sets. SOMs find a mapping from the high dimensional input space to low dimensional feature space, so any clusters that form become visible in this reduced dimensionality. The architecture of the SOM is a multi-dimensional input vector connected via weights to a 2-dimensional array of neurons. When an input pattern is presented to the SOM, each neuron calculates how similar the input is to its weights. The neuron whose weights are most similar (minimal distance in input space) is declared the winner of the competition for the input pattern, and the weights of the winning neuron, and its neighbours, are strengthened to reflect the outcome. The final set of weights embeds the location of cluster centres, and is used to recognize to which cluster a new input vector is closest.

For our experiments we randomly split the 112 problems into training data (80%) and test data (20%). We use the Viscovery SOMine software (www.eudaptics.com) to cluster the instances based only on the 14 features (exclusing ResErr) as inputs. A map of 2000 nodes is trained for 41 cycles, with the neighbourhood size diminishing linearly at each cycle. After the clustering of the training data, the distributions of β^* and ResErr values are examined within each cluster, and knowledge about the relationships between problem structure and algorithm performance is inferred and evaluated on the test data.

5 Experimental Evaluation

The neural network results demonstrate that the relationships between the features of the datasets and the reducibility of each dataset using the selected algorithm can

indeed be learned to a very high accuracy. Based on the five-fold cross validation testing procedure, an R squared value of 0.941 was obtained. These prediction results outperform the original regression model's R squared value of 0.87. While the neural network can be expected to learn the relationships in the data more powerfully, due to its nonlinearity, its limitation is the lack of insight and explanation of those relationships.

The decision tree results produced classification accuracy, on five-fold cross-validation, of 87.5%, with most of the errors due to misclassification of some HIGH datasets as MEDIUM. The arbitrariness of the discretization bins may be contributing to this performance. Figure 5 shows the resulting decision tree rules from the best tree, with the confidence for each rule shown in brackets. These rules suggest that, based on the meta-data, if the number of instances in a dataset is too small (below 768 instances) then the reducibility of the dataset using the chosen algorithm (naïve instance selection method combined with Naïve Bayes' classifier) is likely to be low. This makes sense given the way the algorithm works, particularly the chosen value of k in the k-means clustering algorithm being a fraction (10%) of the number of training instances available. For datasets with a higher kurtosis and higher Normal pdf value the reducibility of the datasets in the meta-data tends to be higher, since more of the variance is due to infrequent extreme deviations which can be eliminated without as much impact.

```
If (N >=768) Then
   If (KU >=1.04) Then
      If (Npdf >=12.6) Then HIGH (100%)
      Else MEDIUM (83%)
   Else MEDIUM(89%)
Else LOW (96%)
```

Fig. 5. Pseudo-code for the decision tree rule system, showing the accuracy of each rule

The advantage of the Self-organizing Map is its visualization capabilities, enabling the exploration of visual correlations and patterns between features and clusters of datasets. After training the SOM based on the 14 input features (excluding ResErr and β^* values), the converged map shows 7 clusters, each of which contains similar datasets defined by Euclidean distance in feature space (see Figure 6). Essentially, the 14-dimensional input vectors have been projected onto a two-dimensional plane, with topology-preserving properties. The clusters can be inspected to see which datasets are most similar, and a statistical analysis of the features within each cluster can be performed to understand what the datasets within each cluster have in common.

The distribution of the input features, and additional information including the distribution of β^* values, can be visually explored using the maps shown in Figure 7 (not all features have been included). A k-nearest neighbour algorithm (with k=7) is used to distribute additional data instances (from the test set) or extra variables (β^* values) across the map.

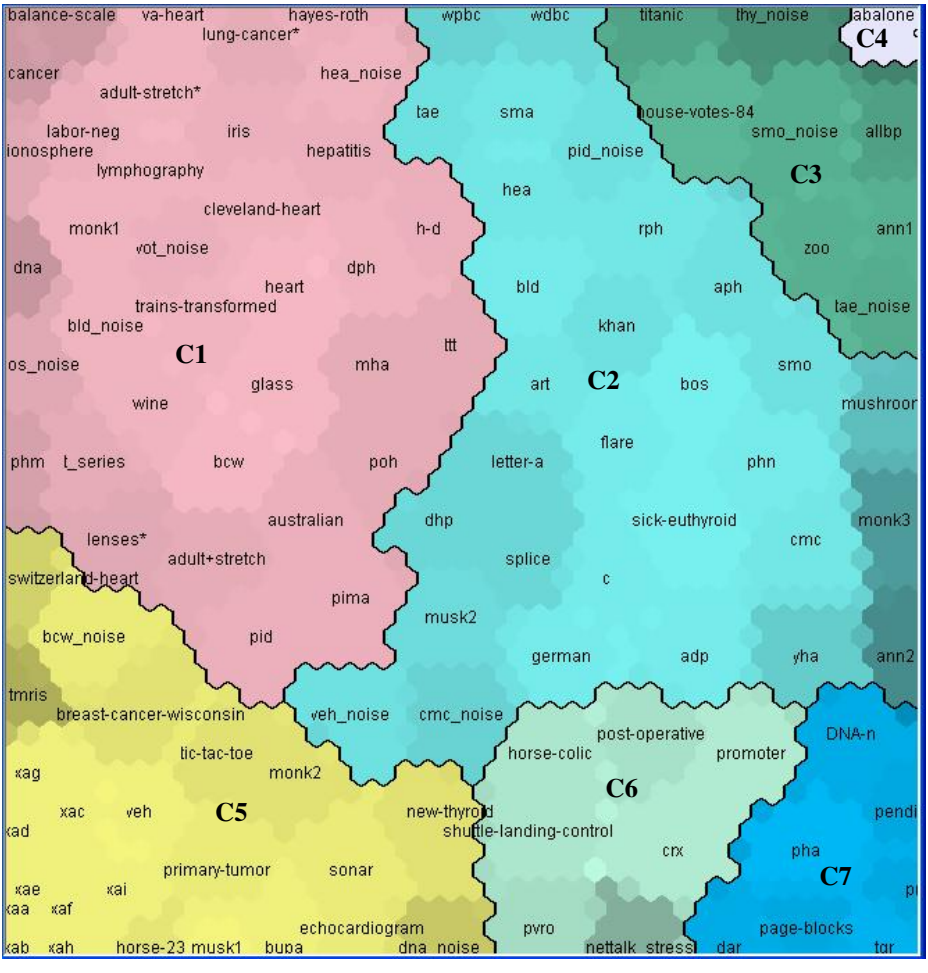


Fig. 6. Self-Organizing Map showing 7 clusters, with classification datasets labeled within each cluster (map generated based on training data, then labeled with training and test data).

In addition to supporting the kind of rules generated by the decision tree (see Figure 5), this kind of visual exploration can be used to identify the group of datasets that are not very reducible using the chosen algorithm (clusters 1, 5 and 6), and to observe that it is not just the number of instances that creates this performance (although all analysis performed so far including the correlation analysis supports the number of instances as a primary determinant of the reducibility of the chosen algorithm). In fact there is significant variation within cluster 2 of the reducibility of the datasets, despite most datasets in cluster 2 being of similar size. The residual errors for these datasets is high suggesting that these datasets were not well modeled by the regression model, and may have some unique properties. The success of the SOM to assist with developing insights relies very heavily on the quality of the

features selected to explore the relative differences between the datasets. A visual exploration of the distribution of the 14 features across these clusters suggests that none of the existing features can, on their own, explain well the behaviour of the datasets in the middle of map. The rules shown in Figure 5 are supported by the observation of cluster 3, 4, and 7. The benefit of the SOM is that if new features are derived, their values can easily be superimposed across the map to explore if they help to explain the performance of certain datasets.

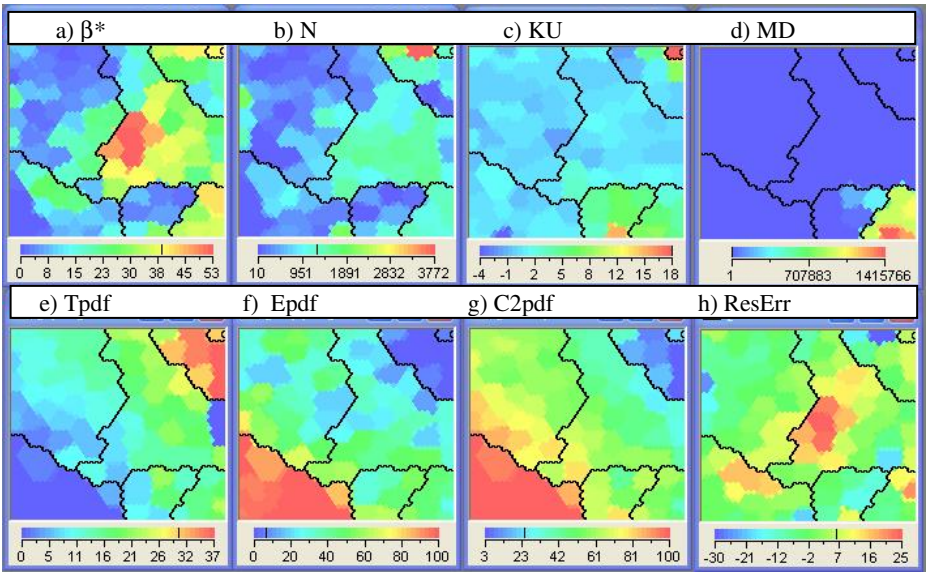


Fig. 7. The distribution of β* values (Fig. 6a) and several key features (Fig. 7b-7h) across the clusters. The colour scale shows each feature at its minimum value as blue, and maximum value as red.

6 Conclusions and Future Research

In this chapter we have generalized the meta-learning process to consider what can be learned about important data pre-processing steps such as instance selection. The performance of a classifier depends strongly on which instances are selected as representatives of the dataset. While we have not utilized any large-scale datasets in this study (our aim has been to utilize well benchmarked UCI repository datasets), the motivation for focusing on instance selection from the wide array of data pre-processing steps we could have selected is due to its importance for many real world data mining tasks.

We have couched the task of instance selection from within the framework of Rice's Algorithm Selection Problem [7]. How do we know which instance selection method is likely to be best suited to a given dataset? How much reducibility can we expect from an instance selection method, given the features of the dataset? A meta-learning approach for tackling these questions has been proposed, whereby meta-data

is generated containing experimental results of many problems (P), described by a set of features (F), tackled with algorithms (A), with the performance evaluated by metrics (Y). This meta-data set of {P, A, Y, F} can then be explored using both supervised and unsupervised learning methods to develop both predictive models of algorithm performance, as well as insights into dependence of algorithm performance on problem features.

This methodology has been illustrated using only one algorithm, but can readily be extended to compare the relative performance of many algorithms. Here, an algorithm is a combination of an instance selection method followed by a classification algorithm. Given the vast number of algorithms available for both of these tasks, the combination creates a huge number of possible algorithms that can be evaluated. In this experimental study, we have illustrated the methodology using a naïve instance selected method based on clustering, and a Naïve Bayes classifier. The performance metric is the level of reducibility this algorithm managed to achieve on a dataset. Certainly the conclusions we can draw from the experimental study only relate to this particular algorithm, and it is clear that more sophisticated algorithms should be able to achieve higher reducibility than the method utilized in this chapter. If we were to adopt this methodology to consider a wider range of algorithms, then the performance metric becomes a ranking for each algorithm based on performance, as has been done in the meta-learning community [9-11].

It is clear from the experimental results that the selected features enabled the performance of the algorithm to be predicted to a high accuracy, and that rules can be generated to explain the conditions under which the algorithm performs well or poorly. The conclusions we can draw, however, are extremely limited by the chosen features. We have utilized a subset of the features used in previous meta-learning studies of classification problems, but in future research it is advisable that these features be re-examined. The Self-organising map visualizations revealed that while some of the algorithm performance can be explained by the decision tree rules, and visual observations were able to confirm these rules, there are clusters of datasets whose performance was not readily explained by the selected features. The construction of features to suitably characterize a dataset, and capture the diversity of problem difficulty that may enable the relative strengths and weaknesses of a range of algorithms to be exposed, remains one of the main challenges in meta-learning.

References

1. Blake, C., Merz, C.J.: UCI Repository of Machine Learning Databases. University of California, Irvine (2002), http://www.ics.uci.edu/_mlearn/MLRepository.html
2. Liu, H., Motoda, H.: On Issues of Instance Selection. Data Mining and Knowledge Discovery 6, 115–130 (2002)
3. Jankowski, N., Grochowski, M.: Comparison of Instances Selection Algorithms I. Algorithms Survey. In: Rutkowski, L., Siekmann, J.H., Tadeusiewicz, R., Zadeh, L.A. (eds.) ICAISC 2004. LNCS (LNAI), vol. 3070, pp. 598–603. Springer, Heidelberg (2004)
4. Reinartz, T.: A Unifying View on Instance Selection. Data Mining and Knowledge Discovery 6, 191–210 (2002)

5. Wolpert, D.H., Macready, W.G.: No Free Lunch Theorems for Optimization. IEEE T. Evolut. Comput. 1, 67 (1997)
6. Grochowski, M., Jankowski, N.: Comparison of Instance Selection Algorithms II. Results and Comments. In: Rutkowski, L., Siekmann, J.H., Tadeusiewicz, R., Zadeh, L.A. (eds.) ICAISC 2004. LNCS (LNAI), vol. 3070, pp. 580–585. Springer, Heidelberg (2004)
7. Rice, J.R.: The Algorithm Selection Problem. Adv. Comp. 15, 65–118 (1976)
8. Smith-Miles, K.A.: Cross-Disciplinary Perspectives On Meta-Learning For Algorithm Selection. ACM Computing Surveys 41(1), article 6 (2008)
9. Vilalta, R., Drissi, Y.: A Perspective View and Survey of Meta-Learning. Artif. Intell. Rev. 18, 77–95 (2002)
10. Michie, D., Spiegelhalter, D.J., Taylor, C.C. (eds.): Machine Learning, Neural and Statistical Classification. Ellis Horwood, New York (1994)
11. Brazdil, P., Soares, C., Costa, J.: Ranking Learning Algorithms: Using IBL and Meta-Learning on Accuracy and Time Results. Mach. Learn. 50, 251–277 (2003)
12. Ali, S., Smith, K.: On Learning Algorithm Selection for Classification. Appl. Soft Comp. 6, 119–138 (2006)
13. Prodromidis, A.L., Chan, P., Stolfo, S.J.: Meta-learning in distributed data mining systems: issues and approaches. In: Kargupta, H., Chan, P. (eds.) Advances of Distributed Data Mining. AAAI Press, Menlo Park (2000)
14. Bernstein, A., Provost, F., Hill, S.: Toward intelligent assistance for a data mining process: an ontology-based approach for cost-sensitive classification. IEEE Transactions on Knowledge and Data Engineering 17, 503–518 (2005)
15. Charest, M., Delisle, S., Cervantes, O., Shen, Y.: Bridging the gap between data mining and decision support: A case-based reasoning and ontology approach. Intelligent Data Analysis 12, 211–236 (2008)
16. Wilson, D.: Asymptotic properties of nearest neighbor rules using edited data. IEEE Transactions on Systems, Man, and Cybernetics 2, 408–421 (1972)
17. Tomek, I.: An experiment with the edited nearest-neighbor rule. IEEE Transactions on Systems, Man, and Cybernetics 6, 448–452 (1976)
18. Jankowski, N.: Data regularization. In: Rutkowski, L., Tadeusiewicz, R. (eds.) Neural Networks and Soft Computing, Zakopane, Poland, pp. 209–214 (2000)
19. Hart, P.E.: The condensed nearest neighbor rule. IEEE Transactions on Information Theory 14, 515–516 (1968)
20. Gates, G.: The reduced nearest neighbor rule. IEEE Transactions on Information Theory 18, 431–433 (1972)
21. Aha, D.W., Kibler, D., Albert, M.K.: Instance-based learning algorithms. Machine Learning 6, 37–66 (1991)
22. Brighton, H., Mellish, C.: Advances in instance selection for instance-based learning algorithms. Data Mining and Knowledge Discovery 6, 153–172 (2002)
23. Wilson, D.R., Martinez, T.R.: Reduction techniques for instance-based learning algorithms. Machine Learning 38, 257–286 (2000)
24. Kohonen, T.: Learning Vector Quantization. Neural Networks 1, 303 (1988)
25. Skalak, D.B.: Prototype and feature selection by sampling and random mutation hill climbing algorithms. In: International Conference on Machine Learning, pp. 293–301 (1994)
26. Sen, S., Knight, L.: A Genetic Prototype Learner. In: Mellish, C.S. (ed.) Proceedings of the 14th International Joint Conference on Artificial Intelligence, Montreal, Quebec, Canada, August 20-25, vol. I, pp. 725–731. Morgan Kaufmann, San Mateo (1995)

27. Madigan, D., Raghavan, N., DuMouchel, W., Nason, M., Posse, C., Ridgeway, G.: Likelihood-Based Data Squashing: A Modeling Approach to Instance Construction. Data Mining and Knowledge Discovery 6, 173–190 (2002)
28. Li, B., Chi, M., Fan, J., Xue, X.: Support Cluster Machine. In: 25th International Conference on Machine Learning. Morgan Kaufmann, San Francisco (2007)
29. Evans, R.: Clustering for Classification: Using Standard Clustering Methods to Summarise Datasets with Minimal Loss of Classification Accuracy. VDM Verlag (2008)
30. Li, X.: Data reduction via Adaptive Sampling. Communications in Information and Systems 2, 5–38 (2002)
31. Domingo, C., Gavaldà, R., Watanabe, O.: Adaptive Sampling Methods for Scaling Up Knowledge Discovery Algorithms. Data Mining and Knowledge Discovery 6, 131–152 (2002)
32. Hartigan, J.A.: Clustering Algorithms. John Wiley & Sons, Inc., New York (1975)
33. Marchiori, E.: Hit Miss Networks with Applications to Instance Selection. Journal of Machine Learning Research 9, 997–1017 (2008)
34. Ali, S., Smith-Miles, K.A.: A meta-learning approach to automatic kernel selection for support vector machines. Neurocomputing 70, 173–186 (2006)
35. Witten, I.H., Frank, E.: Data Mining: Practical Machine Learning Tools and Techniques, 2nd edn. Morgan Kaufmann, San Francisco (2005)
36. Hathaway, R.J., Bezdek, J.C.: Visual cluster validity for prototype generator clustering models. Pattern Recognition Letters 24, 1563–1569 (2003)
37. Pfahringer, B., Bensusan, H., Giraud-Carrier, C.G.: Meta-Learning by Landmarking Various Learning Algorithms. In: Proc. ICML, pp. 74–750 (2000)
38. Smith, K.A.: Neural Networks for Prediction and Classification. In: Wang, J(ed.), Encyclopaedia of Data Warehousing And Mining, vol. 2, pp. 864–869. Information Science Publishing, Hershey PA (2006)
39. Quinlan, J.R.: C4.5: Programs for Machine Learning. Morgan Kaufmann, San Francisco (1993)
40. Kohonen, T.: Self-Organized Formation of Topologically Correct Feature Maps. Biol. Cyber. 43, 59–69 (1982)

Appendix A: 112 datasets used in experimental study (from [34])

Dataset name	# samples	# attributes	# classes	Dataset Name	# samples	# attributes	# classes
abalone	1253	8	3	Musk2	1154	15	2
adp	1351	11	3	nettalk_stress	1141	7	5
adult-stretch	20	4	2	new-thyroid	215	5	3
allbp	840	6	3	Page-blocks	1149	10	5
ann1	1131	6	3	pendigits-8	1399	16	2
ann2	1028	6	3	Pha	1070	9	5
aph	909	18	2	Phm	1351	11	3
art	1051	12	2	Phn	1500	9	2
australian	690	14	2	Pid	532	7	2
balance-scale	625	4	3	pid_noise	532	15	2
bcw	699	9	2	Pima	768	8	2
bcw_noise	683	18	2	Poh	527	11	2
bld	345	6	2	post-operative	90	8	3
bld_noise	345	15	2	primary-tumor	339	17	2
bos	910	13	3	Pro	1257	12	2
bos_noise	910	25	3	promoter	106	57	2
breast-cancer	286	6	2	Pvro	590	18	2
breast-cancer-wisconsin	699	9	2	Rph	1093	8	2
bupa	345	6	2	shuttle	1450	9	5
c	1500	15	2	shuttle-landing-control	15	6	2
cleveland-heart	303	13	5	sick-euthyroid	1582	15	2
cmc	1473	9	3	Sma	409	7	4
cmc_noise	1473	15	3	Smo	1429	8	3
crx	490	15	2	smo_noise	1299	15	3
dar	1378	9	5	Sonar	208	60	2
dhp	1500	7	2	splice	1589	60	3
dna	2000	60	3	switzerland-heart	123	8	5
dna_noise	2000	80	3	t_series	62	2	2
DNA-n	1275	60	3	Tae	151	5	3
dph	590	10	2	tae_noise	151	10	3
echocardiogram	131	7	2	Thy	1887	21	3
flare	1389	10	2	thy_noise	3772	35	3
german	1000	24	2	tic-tac-toe	958	9	2
glass	214	10	6	titanic	2201	3	2
hayes-roth	160	5	3	Tmris	100	3	2
h-d	303	13	2	Tqr	1107	11	2
hea	270	13	2	trains-transformed	10	16	2
hea_noise	270	20	2	Ttt	958	9	2
heart	270	13	2	va-heart	200	8	4
hepatitis	155	19	2	Veh	846	18	4
horse-23	368	22	2	veh_noise	761	30	4
house-votes-84	435	16	2	vot_noise	391	30	2
hypothyroid	1265	25	2	wdbc	569	30	2
ionosphere	351	33	2	Wine	178	13	3
iris	150	4	3	wpbc	199	33	2
khan	1063	5	2	Xaa	94	18	4
labor-neg	40	16	2	Xab	94	18	4
lenses	24	5	3	Xac	94	18	4
letter-a	1334	16	2	Xad	94	18	4
lymphography	148	18	8	Xae	94	18	4
mha	1269	8	4	Xaf	94	18	4
monk1	556	6	2	Xag	94	18	4
monk2	601	6	2	Xah	94	18	4
monk3	554	6	2	Xai	94	18	4
mushroom	1137	11	2	Yha	1601	9	2
musk1	476	166	2	Zoo	101	16	7

Choosing the Metric: A Simple Model Approach

Damien François[1], Vincent Wertz[1], and Michel Verleysen[2]

[1] Université catholique de Louvain, Machine Learning Group,
ICTEAM/INMA, av. G. Lemaître 4, B-1358 Louvain-la-Neuve, Belgium
damien.francois@uclouvain.be, vincent.wertz@uclouvain.be
[2] Université catholique de Louvain, Machine Learning Group
ICTEAM/ELEN, Pl. du Levant, 3, B-1358 Louvain-la-Neuve, Belgium
michel.verleysen@uclouvain.be

Abstract. One the earliest challenges a practitioner is faced with when using distance-based tools lies in the choice of the distance, for which there often is very few information to rely on. This chapter proposes to find a compromise between an a priori unoptimized choice (e.g. the Euclidean distance) and a fully-optimized, but computationally expensive, choice made by means of some resampling method. The compromise is found by choosing distance definition according to the results obtained with a very simple regression model – that is one which has few or no meta-parameters – and then use that distance in some other, more elaborate regression model. The rationale behind this heuristic is that the similarity measure which best reflects the notion of similarity with respect to the application should be the optimal one whatever model is used for classification or regression. This idea is tested against nine datasets and five prediction models. The results show that this approach is a reasonable compromise between the default choice and a fully-optimized choice of the metric.

1 Introduction

Many regression or classification models are build on some similarity (or dissimilarity) measure which is used to compare how similar/dissimilar two data elements are. The most typical example is the nearest neighbor prediction model which uses the distance between data elements (or cases, instances) in the data space to determine the response value of a fresh data element based on an average of the target value among its neighbors – the elements which are closest to it.

Other tools transform the distance, which is intrinsically a dissimilarity measure, into a similarity measure, most often through some decreasing function of the distance. Support vector machines [1], but also Radial-Basis Function Networks [2], and many Lazy-Learning methods [3] are based on this principle. Other kernels, like polynomial kernels, are not based on distances, but they can be interpreted as well a similarity measures.

One the earliest challenges the practitioner is faced with when using such tools in the choice of the distance. Sometimes, the choice of the distance is obvious ; with 2-dimensional, noise-free, data, the Euclidean distance is unquestionable. For other types of data, specific distances can be designed incorporating prior knowledge [4].

N. Jankowski et al. (Eds.): Meta-Learning in Computational Intelligence, SCI 358, pp. 97–115.
springerlink.com

Sometimes the choice is much less obvious, especially when data show specific features like a mix of continuous and categorical attributes, or when the data are highly-dimensional, with strongly redundant attributes, or affected by non-Gaussian noise. The distance metric should then be considered as a meta parameter to the model, which should be chosen precisely [5]. Although the Euclidean distance might appear in many cases as the optimal one, when the data are complex, there is a real need to check that other distance do, or do not, perform better.

One of the following two approaches is consequently often adopted. Either the practitioner uses the basic defaults – the Euclidean distance – even if it may not be the optimal choice. Or, the practitioner adds the distance definition to the cross-validation loop used to optimize other (meta-)parameters like the regularization parameters in support vector machines, or the number of centroids in radial-basis function networks, or the number of neighbors in lazy learning methods. The former approach is definitely not optimal, while the latter is very CPU intensive.

The idea proposed in this chapter is to find a compromise between both approaches by optimizing the choice of the distance with a very simple regression model – that is one which has few or no meta-parameters – and then use that distance in some other, more elaborate regression model.

The rationale behind this idea is that the similarity measure which best reflects the notion of similarity with respect to the application is the optimal one whatever model is used for regression. This rationale is very similar to the one followed in filters methods for feature selection [6] and landmarking methods for dataset characterization in meta learning [7].

The main objective of this contribution is to show that the optimal metric for the simpler model (the one nearest-neighbor regression model) is highly correlated with the optimal metric for more elaborate models like support vector machines or neural networks, consequently showing that optimizing the metric on simpler models to then use it in more elaborate ones actually make sense.

The remaining of this chapter is organized as follows. Section 2 introduces the notion of distance and provides some definitions. Section 3 describes the proposed approach, while the results are reported in Section 4.

2 The Notion of Distance

Let $\mathscr{X}$ be a set of data elements living in some space Ω. To compute the distance between any two points of $\mathscr{X}$, we need to define a distance function, or metric, d over Ω.

2.1 Definitions

A distance function d over the set Ω is defined on $\Omega \times \Omega$ and takes values in $\mathbb{R}^+$, the set of non-negative reals. When evaluated between two identical points, it must return zero, and must be symmetric and obey the triangle inequality.

The most well-know distance definition is the Euclidean distance, defined over $\mathbb{R}^p$, with x_i denoting the ith component of vector x, as

$$d(x,y) = \sqrt{\sum_i (x_i - y_i)^2} \tag{1}$$

It corresponds to the usual notion of distance we use in the real (three-dimensional) world.

The notion of distance is intrinsically linked to the notion of similarity referred to when interpreting prediction models. While distances are often used for comparison in prediction models like the k-nearest neighbors model, similarities are rather used when the value is to be used for weighting in a sum, like in Support Vector Machines (SVM) for instance. Distant objects are seen as different while objects that are close one to another are deemed similar. A similarity function is thus a non-negative, bounded, decreasing function of the distance.

One of the most used similarity function in kernel methods is the Gaussian kernel:

$$s(x,y) = e^{-\left(\frac{d(x,y)}{2\sigma}\right)^2}, \tag{2}$$

while other measure are also popular in lazy learning, such as

$$\frac{1}{1+d(x,y)}. \tag{3}$$

In this chapter we will nevertheless focus exclusively on the on the Gaussian kernel because it is used in prediction and classification models from statistics, machine learning, pattern recognition, fuzzy logic, etc [1] .

2.2 Distance on Other Types of Data

The Euclidean distance is used when data can be expressed in a vector space, that is basically when data are numerical, or when numeric features can be extracted form them. Many other distance definition have been developed for non-numerical data, like the now classical edit distance on strings [8], and the more recent commute distance on graph nodes [9].

Edit distance (Levenshtein) on strings. Character strings are sequences of characters representing words in a written language, or in any other formal language based on a finite alphabet, like DNA strings described by the four letters A C T and G. The idea of the edit distance is to define how similar two strings are by the number of fundamental operations needed to transform one string into the other. Fundamental operations are classically defined as: insertion, deletion, and substitution of a single character. The distance is computed as the minimal number of such operations. The computations are carried on using a dynamic computing algorithm requiring order the product of both string lengths operations.

Commute time distance between nodes in a graph. Graph data structures are often used to represent communication networks (roads, phone calls, etc.) and preference relationships (social networks, movie preferences, etc.). This latter kind of graphs is made of nodes of two different types, often called users and items. The distance between for instance a user and an item is computed as how easy it is to go from one to another by 'walking' on the graph, moving from one node to another by following the edges. The distance between two items measures how many users have used both at the same time, while the distance between two users measures how many common items they have used, and the distance between an item and a user measures how likely a user is to be interested in that item.

Many other distance definition have been used in specific contexts, involving specific data types (images, sounds, etc.). A comprehensive review can be found in [4]. Although this chapter focusses on numerical, vectorial, data, the ideas can be transposed for non numerical data as far as several distance metrics can be defined and no hint is available about which one of them is to be used.

2.3 Extensions of the Euclidean Distance

While the Euclidean distance has been used extensively since the early days of multivariate data analysis, the last decade has shown the advent of a new kind of data. While typical data used to be small and clean, data are now large and dirty. Data are described by a large number of attributes, contains outliers, are redundant (collinear), polluted by non-Gaussian noise, etc. This shift in data has led to investigation of extensions to the Euclidean distance, in two main directions: weighted distances on the one end, and Minkowski and fractional distances on the other end.

2.3.1 Weighted Distances

Weighted Euclidean distances are a family of distances where each component of the feature vector is weighted to give it more or less importance relatively to the other components. Without loss of generality, the Euclidean distance can be rewritten as

$$d(x,y) = \sqrt{(x-y)^t A (x-y)} \tag{4}$$

where A is the $p \times p$ identity matrix. This form allows an immediate extension by allowing matrix A to be any semi-definite (with non-negative eigen-values) matrix. If A is diagonal, this corresponds to giving each dimension a relative weight. Using a full matrix A allows taking into account interaction between variables (dimensions).

Although this kind of extension is very interesting, it can be thought of as a preprocessing of the data rather than a true questioning of the relevance of the Euclidean distance. Indeed, using the Mahalanobis distance, defined using the inverse of the covariance matrix of the data as matrix A, would be equivalent as using the Euclidean distance on the same data preprocessed with Principal Component Analysis.

In this chapter, we will rather focus on the other type of extension to the Euclidean norm.

2.3.2 Minkowski and Fractional Distances

The other way of extending the Euclidean norm is by allowing the exponent in the definition to take any positive value. This leads to the Minkowski family of distances, and to fractional distances.

The Euclidean distance can be written, with $l = 2$, as

$$d(x,y) = \left(\sum_i |x_i - y_i|^l \right)^{\frac{1}{l}}. \tag{5}$$

By allowing l to take any positive integer value, we define the Minkowski family of norms, from which the most known are the Euclidean, of course, but also the Manhattan distance ($l = 1$) and the Chebyshev distance (by letting $l \to \text{inf}$), which is computed as

$$d(x,y) = \max_i |x_i - y_i| \tag{6}$$

Allowing l to take any positive value leads to the so-called fractional norms, which have been investigated in various contexts [10]. They will be denoted l-norms in the following.

2.4 Distances in Prediction Models

Regression models are built from data having a continuous label, called a response value, or a target. The objective is to build a function (or model) that associates a data element with its response value. The model is built using data for which this label is known, it is then applied to new data – for which the label is unknown – to obtain a prediction of the correspondig response value.

Many prediction models fall in a category which we could label as 'geometrical' models because they rely on the definition of a metric over the data space. In such models, the estimation of the value to predict is carried out by computing a weighted average of the response values associated to known data lying in the neighborhoods of the new value.

In such models, the predicted value $\hat{y}$ for a new datum x is of the form:

$$\hat{y} = \sum_i w_i \cdot s(C_i, x), \tag{7}$$

where $s(.,.)$ denotes a similarity measure, such as (2) or (3) for instance, and the C_i are vectors living in the same space as the data.

Different choices for $s(.,.)$, C_i and w_i lead to different models structures. For instance, by letting C_i be the original data points and w_i their associated values, and by defining $s(.,.)$ to be one if the distance between both arguments is less than a threshold, and zero otherwise, we obtain a nearest-neighbor prediction model [3].

By letting C_i be centroids obtained by a clustering of the data and w_i be optimised, we obtain a radial-basis function network [11]. Such model can also be obtained by letting C_i be data points choosen according to a forward procedure and the w_i be set by Orthogonal Least Squares [12].

By letting C_i be some data points chosen as the results of a penalized quadratic optimization problem, and w_i be the response value of those specific points, called support vectors, we recognize the support vector machine [1].

The least squares support vector machine [13], as well as the kernelized version of the nearest neighbor prediction model [14] and many lazy-learning algorithm [3] can also be expressed in a way very similar to Equation 7.

3 The Distance as a Meta Parameter

The Euclidean distance is often the default choice, mainly because of historical reasons. When data was low-dimensional, described by two or three attributes, or sometimes four, and the noise was close to white noise, nicely Gaussian, the Euclidean distance was without a doubt the optimal choice.

Nowadays, as data become higher- and higher-dimensional, described by hundreds, or thousands of attributes, various other noise scheme appear than the white noise, and the Euclidean distance is more and more questioned as a relevant measure of similarity [15].

3.1 Optimizing the Choice of the Distance

Still, the general practice is often to consider the Euclidean distance only. The optimal way of choosing among alternative distance definitions would be to consider adding an outer loop to the usual cross-validation of the hyper parameters of the prediction or classification model in which they appear. Considering most nonlinear models depend on at least two hyper parameters, which must be optimized through some resampling method, this would of course multiply the time needed in the optimization of the models by the number of distinct distance functions that are considered, making this approach computationally too heavy in practice.

3.2 The General Idea: An a Priori Choice of the Distance

The former approach for choosing the distance could be thought of as an instance of a wrapper methodology for metric selection, by similarity with such approach in a feature selection context [16]. Note that feature selection can be thought of as a way of changing the metric so that only few dimensions are taken into account in the distance definition, or, in other words, a case of weighted distance where the weights are binary.

Feature selection offers another methodology, often referred to as filter [17], where the idea is to choose the features according to some method before any prediction model is built. Filters are often based on statistical or entropy-related criteria such as correlation, Mutual Information [6], or the non-parametric noise variance estimator Gamma Test [18] . Both the gamma test and the most efficient estimator of mutual information in higher dimensional-spaces are based on nearest neighbors [19].

In the field of meta learning for data mining, the concept of landmarking [7] is similar in the sense that the idea is to characterize a dataset by the performances of simple models on the data, from which information is ultimately extracted to find, beforehand,

the optimal model that would perform best on that particular dataset. Besides the linear model, the simple nearest neighbor model, is often used.

As the idea in this chapter is to choose the optimal distance, we suggest to use the performances of a nearest neighbor model (k-NN) to choose the most relevant metric in a reasonable time.

3.3 The Criterion: The Performances of a k-NN

The performances of a k-nearest neighbor regression model can be estimated using the Leave-One-Out normalized Median Squared Error of the model on the data, as explained hereafter.

The leave-one-out procedure is used so as to avoid potential biases due to over fitting, especially in the case of low values of k. Allowing $k = 1$, without leave-one-out procedure, would indeed lead to perfect (and useless) prediction in all cases if no leave-one-out procedure is assumed.

Normalizing, in the present context, means dividing the median squared error by the variance of the response value so that the resulting figure is normalized by the range of the target. A value of zero means perfect prediction while a value larger than one means a completely useless model performing no better than a constant model. The value can be interpreted as a percentage of the variance of the target which can be explained by the model.

The rationale behind the use of the median squared error rather than the more common mean squared error is to avoid problems due to potential outliers in the data. Outliers often lead to normalized mean squared errors reaching values higher than one in such context when outliers have not been identified yet.

3.4 The Proposed Method: Randomized Subsamples with Optimized Number of Neighbors

The nearest-neighbor prediction model depends on one parameter, namely the number of neighbors k to be considered in the evaluation of the predicted response value. One option would be to choose a reasonable default value beforehand, sufficiently high so that the variance of the estimation is low enough. Nevertheless, the structure of the nearest neighbor model is such that getting the results for one particular value of k is nearly as costly as getting them all up to some bound K. Therefore, it is suggested to take the performance associated to one particular distance definition as the maximum over all values $1 \leq k \leq K$ of the number of nearest neighbors, and taking K to be a reasonable value depending on the total size of the dataset.

Even though the nearest neighbor approach requires virtually no computer time in building the model, since it belongs to the so called lazy learners, the time needed for prediction can be rather large when the dataset size grows. In such cases, an indexing structure can be of some help, like the kd-tree. However these trees have shown their limits on high-dimensional data [20]. Approximate nearest neighbors, or randomized nearest neighbors can the be used to reduce the computational time. Here, we suggest to consider a random subset of the observations, or cases, whose size is kept small enough to allow reasonable computation times.

Because of that random subsampling, and to robustify the method against the variance of the nearest neighbor model and of the leave-one-out procedure, it is suggested to draw several subsamples and to repeat the evaluation of each distance definition several times. The suggested policy is to choose the distance function that outperforms the other, if such exists, or to default to the Euclidean distance in case of ties.

Finally, we suggest considering a limited subset among all possible Minkowski and fractional distances, namly: the 1/2-norm distance, the Manhattan distance, the Euclidean distance and the Chebyshev distance. A larger number of values could of course be tested, but our experience is that values larger than 2 often lead to results comparable to the Chebyshev distance, while values lower than 1/2 lead either to results close to those of the 1/2-norm, or to severe numerical instabilities [15].

4 Experiments

The suggested approach is tested against nine datasets and five prediction models based on a distance definition.

4.1 Datasets

The datasets were chosen so as to encompass a rather large variety of data. They are all concerned with regression problems. In the experiments, no domain knowledge has been used to pre process the data ; this of course may harm the results, but it allows fair comparison.

Friedman. This artificial dataset is generated from five random $X_1 \ldots X_5$ variables uniformly distributed over $[0,1]$. The target is computed as

$$Y = 10sin(X_1X_2) + 20(X_3 - 0.5)^2 + 10X_4 + 5X_5 + \varepsilon.$$

where ε is a random Gaussian noise whose variance is chosen to obtain a Signal to Noise Ratio (SNR) of 10. The sample size is 200. This data was originally used by Friedman to illustrate the MARS algorithm [21]. It originally contained five more variables carrying solely noise. These have been ignored here as distance-based models are highly sensitive to noise-variables and the objective here is to assess the relevance of the method rather than the relevance of the models.

Tecator. Tecator is a set of meat near-infrared spectra, discretized from 850 nm to 1050 nm into 100 values. The objective is to predict the amount of fat in the meat sample and the sample size is 215. Variables in this dataset are highly correlated, leading to highly redundant features. This dataset is available from `http://lib.stat.cmu.edu/datasets/tecator`. This data are known to be rather linear [22].

Housing. The Housing dataset comprises 7 real values describing some demographic information about a small area in the suburbs of Houston, in the USA [23]. The target

is the median house price in that area. The sample size is 506. This dataset is available from the UCI repository. This dataset originally comprises additional categorical variables which are not being considered here.

Forest. The Forest dataset is available from the UCI repository [23]. The aim is to estimate the size of the burned area in forest fires in Portugal, from meteorological and spatial data. The dimensionality of the dataset is 13 while the sample size is 517.

Hardware. The Computer Hardware data set comprises nine variables describing computer hardware[23] ; the target is a measure of the performance of the hardware in terms of some benchmark. Most of the values are integer ; the sample size is 209. This dataset is available from the UCI repository.

Concrete. The Concrete Compressive strength dataset is a highly nonlinear function of nine attributes describing concrete composition and age. It comprises 1030 instances and is available from the UCI repository [23].

Housingburst. This dataset is the same as the Housing dataset except that it has been artificially altered with burst noise, that is noise which affects only some of the values, but alters them dramatically. Such noise scheme can be related to outliers and to multiplicative noise in signal processing. Input errors in human-encoded data often also follow that noise scheme (a misplaced comma leading to a 10-fold error in the value, etc.) This type of data has been found to be better handled by fractional metrics.

Tecatorburst. This dataset is the same as the Tecator dataset except that is was given the same treatment as Housingburst to artificially favor fractional distance.

Delve. This dataset is actually a subset of the Census-House dataset[1] where only continuous variables have been considered, because introducing binary or categorical variables in a distance function requires treatments that are outside the scope of this study.

4.2 Models

All the models used are based on the Gaussian kernel. The prediction function they always follow the form of (7). The differences between those models lie in the algorithm used to determine the parameters. Here follows a list of the mnemonics for each model along with a short description and references for a more complete description.

LLWKNN. This model is taken from the field of Lazy Learning [3]. It is similar to a nearest-neighbor model except that the influence of each neighbor is weighted by the similarity between that neighbor and the new data for which an estimation of the label must be found. It has two meta parameters : the number of neighbors and the width of the kernel used as similarity measure.

[1] http://www.cs.toronto.edu/ delve/data/datasets.html

KKNN. This is the kernelized version of the nearest-neighbor [14]. The distance between the data elements is computed in the induces feature space rather than in the data space. The meta parameters of this model are the same as for the previous one.

RBFNOrr. This model corresponds to the radial-basis function networks trained using Orr's orthogonal least squares with forward selection of the centers. Centers here are taken amongst the elements of the training set. This model has two meta parameters; the width of the Gaussian functions, and a regularization parameter [12].

LSSVM. Least Squares-Support Vector Machines [13] are defined in a way very similar to ridge regression, but the learning scheme approaches the one of the support vector machine. As for the previous model, two meta parameters must be optimized; the width of the kernel and a regularization parameter.

SVM. Support vector machines are built by solving a quadratic programming problem which identifies support vectors among the data [1]. The Gaussian kernel is used here. SVM's rely on two parameters; the width of the kernel and a regularization parameter expressing an upper bound on the number of support vectors allowed in the model. Support vector machines for regression furthermore have a third parameter which is the width of the so called ε-insensitive tube where the prediction error is considered to be null. In this work, that parameter is chosen a priori according to a simple heuristic based on the variance of the target.

It is important to note that kernels based on a distance other than the Euclidean norm are not necessarily Mercer kernels [1] and therefore should be used in SVM-like methods with caution. However, is has become an accepted practice to use non-Mercer kernels in support vector machines. The convexity of the problem is lost, but sufficiently good models are often built despite this drawback [24].

4.3 Methodology

The methodology as follows. First, the proposed methodology is followed to estimate the distance relevance. Four choices for the distances are considered : the 1/2-norm distance, the Manhattan distance, the Euclidean distance and the Max-norm distance. The number of neighbors is optimized in the range 1..21, and the number of repetitions is 30, while the number of cases kept in each repetition is 100.

Then, the performances of the selected models are evaluated. Those models are optimized using a 10-fold cross-validation procedure, and results on an independent test set are reported (normalized median squared error). This whole procedure is repeated ten times with random splitting into training and test set.

Finally, the correspondence between the results of the nearest neighbor model and the selected models is investigated.

4.4 Results for the Nearest Neighbor Model

The results of the nearest neighbors model are given in Figure 1. All 30 runs are summarized in a box plot. The horizontal axis represents the distance, from left to right:

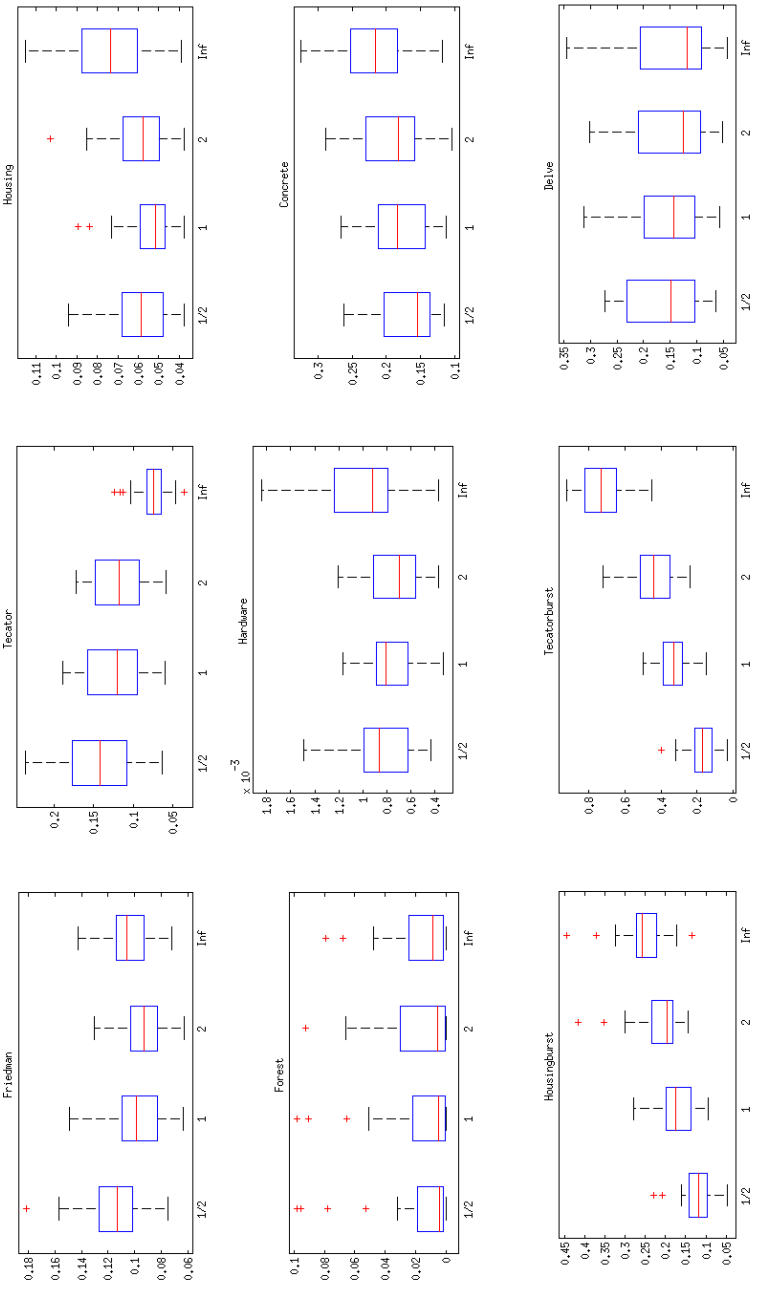


Fig. 1. Box plot of the Normalized Median Squared Error of the nearest neighbor for estimating distance function relevance. The horizontal axis represents the exponent of the distance; in order, 1/2-norm, Manhattan distance, Euclidean distance and Chebyshev distance

1/2-norm, Manhattan distance, Euclidean distance and Chebyshev distance. The boxes represent the inter-quartile range, the horizontal line inside the box is the median, and the tails represent the fifth and 95th percentile respectively. The plusses represent single outliers. Table 1 provides with the mean and standard deviation of those distributions.

For Friedman and Hardware, the visual inspection of the plots seems to favor the Euclidean distance, at least in terms of median results, although the difference, might not be statistically significant.

As far as dataset Tecator is concerned, the Chebyshev metric seems to be the favorite choice. The same conclusion can be seen for Delve, although this is much less obvious. For both Housingburst and Tecatorburst, the 1/2-norm seems to be preferable. The same conclusion can be drawn about Concrete while it is not as clear as for the latter.

The results for Housing seem to point out the Manhattan norm as most relevant; it nevertheless does not outperform the 1/2-norm and the Euclidean norm significantly, while for the Forest dataset, all distances seem to perform equally. For those datasets, defaulting to the Euclidean distance seems also reasonable.

Table 1. Mean, (and standard deviation) of the Normalized Median Squared error of the k-nearest neighbor model with four different distance definitions, over 30 runs.

Dataset	1/2-norm	1-norm	2-norm	∞-norm
Friedman	0.1148 (0.023)	0.0973 (0.019)	0.0928 (0.013)	0.1049 (0.017)
Tecator	0.1464 (0.043)	0.1263 (0.036)	0.1198 (0.033)	0.0753 (0.020)
Housing	0.0588 (0.104)	0.0543 (0.012)	0.0597 (0.014)	0.0734 (0.020)
Forest	0.0176 (0.027)	0.0179 (0.026)	0.0161 (0.022)	0.0167 (0.020)
Hardware	0.0008 (0.0002)	0.0007 (0.0002)	0.0007 (0.0002)	0.0010 (0.0004)
Concrete	0.1724 (0.042)	0.1804 (0.044)	0.1904 (0.046)	0.2158 (0.046)
Housingburst	0.1199 (0.038)	0.1721 (0.044)	0.2164 (0.058)	0.2532 (0.058)
Tecatorburst	0.1729 (0.080)	0.3322 (0.089)	0.4435 (0.125)	0.7319 (0.127)
Delve	0.1575 (0.064)	0.1554 (0.063)	0.1488 (0.070)	0.1523 (0.081)

4.5 Results for the Selected Prediction Models

The results of 10 runs of the random splitting are shown in Figure 2. Most of the time, the results of the different models are comparable. No model outperforms all other over all datasets. Sometimes, however, one model performs worse than the others. It is the case for instance with the RBFNOrr model on Tecator and Hardware. For Friedman and Concrete, the KKNN performs really badly with the 1/2-norm while for Delve, the SVM performs worse with the Euclidean norm than any other model on those data. Note that some models deliver poor performances (around 0.6 and above) with specific distances measures ; in those cases, the choice of the correct distance measure is crucial.

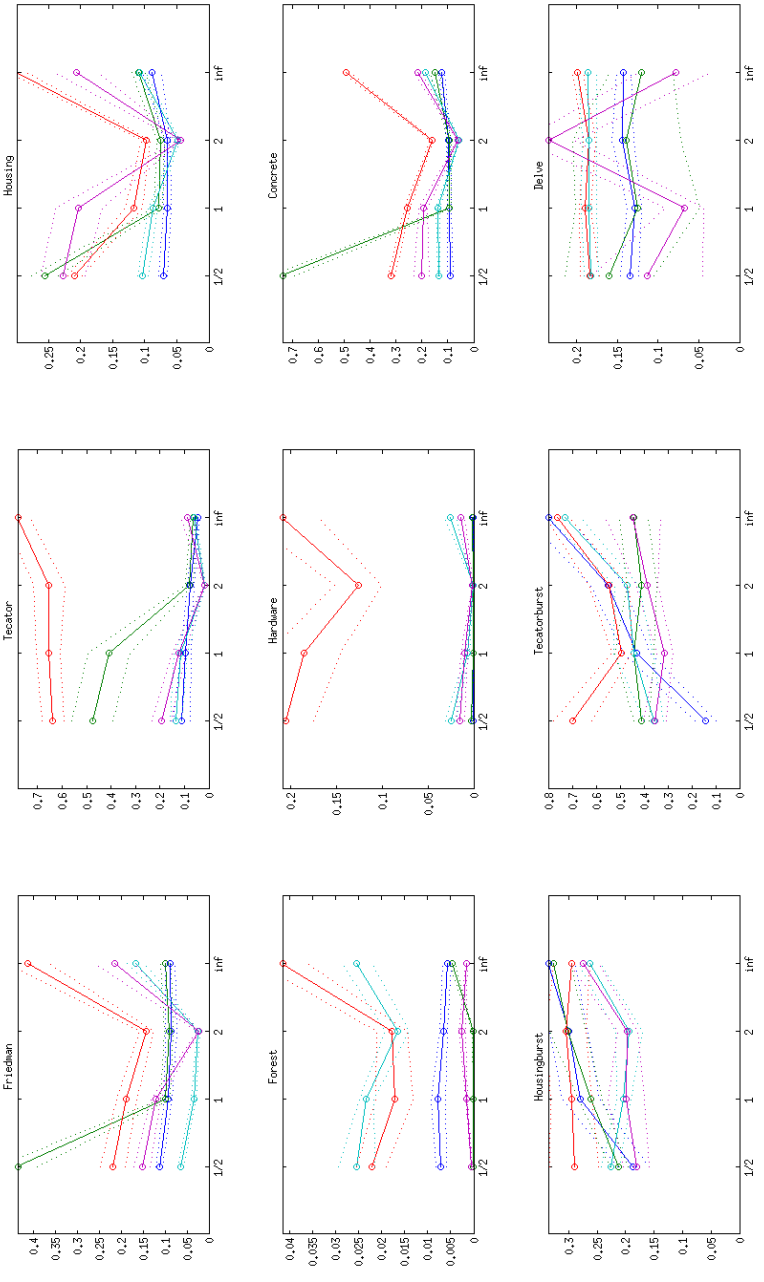


Fig. 2. Mean (plain) and standard deviation (dotted around the mean) of the Median squared error in test of all models for four distance definitions, over 10 runs. Blue: LLNKWW, Green: KKNN, Red: RBFNOrr, Cyan: LSSVM, Pink: SVM

Friedman dataset. The results for the Friedman dataset are given Table 2. The results show that for most models, the Euclidean distance is the optimal one, although the Manhattan distance performs nearly as well as the Euclidean one.

Table 2. Results for the selected models on the Friedman dataset.

Model	1/2-norm	1-norm	2-norm	∞-norm
LLWKNN	0.11228 (0.008)	0.09209 (0.009)	0.08555 (0.013)	0.08838 (0.010)
KKNN	0.43597 (0.044)	0.09950 (0.011)	0.09144 (0.009)	0.09821 (0.012)
RBFNOrr	0.21901 (0.029)	0.18788 (0.023)	0.14174 (0.016)	0.41353 (0.047)
LSSVM	0.06422 (0.007)	0.03438 (0.006)	0.02466 (0.003)	0.16650 (0.020)
SVM	0.15259 (0.018)	0.12033 (0.011)	0.02298 (0.003)	0.21535 (0.045)

Tecator dataset. The results are presented in Table 3. Except for the RBFNOrr model, the Chebyshev distance performs well. It is outperformed by the Euclidean distance with the LSSVM and SVM, but not by far.

Table 3. Results for the selected models on the Tecator dataset. Both the 1/2 and the ∞ norms were found most relevant for this dataset by the nearest neighbour approach.

Model	1/2-norm	1-norm	2-norm	∞-norm
LLWKNN	:0.11121 (0.035)	0.09388 (0.021)	0.07496 (0.014)	0.04611 (0.011)
KKNN	0.47489 (0.084)	0.40867 (0.085)	0.08167 (0.016)	0.06504 (0.012)
RBFNORR	0.63766 (0.045)	0.65640 (0.048)	0.65381 (0.064)	0.78148 (0.049)
LSSVM	0.13403 (0.025)	0.11551 (0.027)	0.01635 (0.003)	0.05884 (0.007)
SVM	0.19250 (0.039)	0.12456 (0.028)	0.01939 (0.002)	0.08696 (0.028)

Housing dataset. As shown in Table 4, the Euclidean distance performs best in most of the cases. For some models, the Manhattan distance performs equally to the Euclidean one.

Table 4. Results for the selected models on the Housing dataset. The Euclidean distance was found to be the most relevant for this dataset by the nearest neighbour approach.

Model	1/2-norm	1-norm	2-norm	∞-norm
LLWKNN	0.07142 (0.006)	0.06442 (0.006)	0.06501 (0.004)	0.08863 (0.014)
KKNN	0.25540 (0.021)	0.07839 (0.007)	0.07490 (0.006)	0.10903 (0.011)
RBFNORR	0.20883 (0.011)	0.11742 (0.013)	0.09826 (0.006)	0.29778 (0.014)
LSSVM	0.10279 (0.008)	0.08691 (0.007)	0.04797 (0.003)	0.10732 (0.008)
SVM	0.22720 (0.035)	0.20255 (0.036)	0.04456 (0.004)	0.20638 (0.034)

Forest dataset. The results on this dataset are very good in general. All distance definitions provide similar results except for the Chebyshev distance with the LSSVM and the RBFNOrr.

Table 5. Results for the selected models on the Forest dataset.

Model	1/2-norm	1-norm	2-norm	∞-norm
LLWKNN	0.00711 (0.001)	0.00772 (0.002)	0.00656 (0.001)	0.00580 (0.001)
KKNN	0.00008 (0.000)	0.00007 (0.000)	0.00017 (0.000)	0.00470 (0.001)
RBFNORR	0.02217 (0.003)	0.01724 (0.004)	0.01772 (0.003)	0.04156 (0.005)
LSSVM	0.02546 (0.004)	0.02334 (0.002)	0.01654 (0.003)	0.02535 (0.003)
SVM	0.00043 (0.000)	0.00160 (0.000)	0.00251 (0.001)	0.00161 (0.000)

Hardware dataset. The results are given in Table 6. The euclidean norm appears to be the optimal choice for all models. The other distances perform nearly as well though. The model RBFNOrr performs really worse than the others ; for him the Euclidean distance is clearly the most relevant one.

Table 6. Results for the selected models on the Hardware dataset.

Model	1/2-norm	1-norm	2-norm	∞-norm
LLWKNN	0.00066 (0.000)	0.00064 (0.000)	0.00071 (0.000)	0.00093 (0.000)
KKNN	0.00243 (0.001)	0.00083 (0.000)	0.00075 (0.000)	0.00129 (0.000)
RBFNORR	0.20447 (0.030)	0.18521 (0.039)	0.12617 (0.025)	0.20857 (0.039)
LSSVM	0.02503 (0.006)	0.00685 (0.002)	0.00040 (0.000)	0.02558 (0.006)
SVM	0.01465 (0.005)	0.01041 (0.004)	0.00117 (0.000)	0.01444 (0.004)

Concrete dataset. On this dataset, all models agree that the Euclidean norm gives the best results, although for some models, those are not significantly better than those obtained with the 1/2-norm. See Table 7 for detailed results.

Table 7. Results for the selected models on the Concrete dataset.

Model	1/2-norm	1-norm	2-norm	∞-norm
LLWKNN	0.09023 (0.011)	0.09468 (0.008)	0.09721 (0.008)	0.12322 (0.010)
KKNN	0.73774 (0.045)	0.09357 (0.007)	0.09453 (0.008)	0.15029 (0.019)
RBFNORR	0.31822 (0.013)	0.25667 (0.010)	0.16228 (0.006)	0.49396 (0.015)
LSSVM	0.13310 (0.004)	0.13886 (0.008)	0.05614 (0.002)	0.18750 (0.012)
SVM	0.20152 (0.030)	0.19531 (0.017)	0.06034 (0.004)	0.21536 (0.016)

Housingburst dataset. For all models, best results are obtained using the 1/2-norm. Although all differences might not be statistically significant, there seem to be a clear

tendency to decrease the results when the exponent in the distance definition increases (Table 8).

Table 8. Results for the selected models on the Housingburst dataset.

Model	1/2-norm	1-norm	2-norm	∞-norm
LLWKNN	0.18671 (0.028)	0.27868 (0.032)	0.29941 (0.033)	0.33561 (0.047)
KKNN	0.21145 (0.015)	0.26135 (0.021)	0.30078 (0.032)	0.32621 (0.030)
RBFNORR	0.29017 (0.043)	0.29494 (0.036)	0.30390 (0.035)	0.29473 (0.032)
LSSVM	0.22618 (0.016)	0.20318 (0.012)	0.19373 (0.023)	0.26322 (0.019)
SVM	0.18102 (0.023)	0.19860 (0.032)	0.19655 (0.018)	0.27359 (0.026)

Tecatorburst. Although less convincingly than with the Housingburst dataset, the results tend to show a decrease in performances as the exponent increases. The sole RBFNOrr behaves differently, as for this model, the 1/2-norm is the worst performing one (Table 9).

Table 9. Results for the selected models on the Tecatorburst dataset.

Model	1/2-norm	1-norm	2-norm	∞-norm
LLWKNN	0.14339 (0.044)	0.42972 (0.056)	0.55222 (0.070)	0.80076 (0.088)
KKNN	0.41087 (0.033)	0.44195 (0.082)	0.40912 (0.062)	0.44132 (0.061)
RBFNORR	0.69896 (0.082)	0.49272 (0.043)	0.54690 (0.083)	0.76273 (0.047)
LSSVM	0.35721 (0.040)	0.44292 (0.082)	0.47213 (0.050)	0.73222 (0.067)
SVM	0.35293 (0.048)	0.31233 (0.034)	0.38794 (0.047)	0.44586 (0.115)

Delve. For each model, except for the LSSVM, all distance measures perform equally good. It is worth noting that for the LSSVM model, both the Manhattan metric and the Chebyshev distance perform better than any other combination of model and distance definition. By contrast, the LSSVM with the Euclidean distance perform worse than all others. (Table 10).

Table 10. Results for the selected models on the Delve dataset.

Model	1/2-norm	1-norm	2-norm	∞-norm
LLWKNN	0.13492 (0.011)	0.12886 (0.009)	0.14356 (0.014)	0.14207 (0.009)
KKNN	0.15991 (0.054)	0.12507 (0.077)	0.13853 (0.066)	0.12023 (0.039)
RBFNORR	0.18417 (0.012)	0.18917 (0.007)	0.18426 (0.007)	0.19852 (0.007)
LSSVM	0.18191 (0.009)	0.18458 (0.009)	0.18430 (0.009)	0.18631 (0.011)
SVM	0.11346 (0.068)	0.06793 (0.024)	0.23471 (0.032)	0.07827 (0.044)

4.6 Summary

Following the suggested policy (Cfr Sec 3.4), the distance functions were chosen for each dataset as follows:

Dataset	1/2-norm	1-norm	2-norm	∞-norm
Friedman			⋆	
Tecator				⋆
Housing			⋆	
Forest			⋆	
Hardware			⋆	
Concrete			⋆	
Housingburst	⋆			
Tecatorburst	⋆			
Delve				⋆

To better grasp the relevance of those choices, we compute the differences in Normalized Median Squared Error between that choice and the optimal choice over all distance definitions. The larger the difference, the worse the choice is. If the difference is zero, it simply means that the optimal choice was made. The differences are computed for all experiments ; 9 datasets, 5 models, 10 repetitions, for a total of 450 experiments. They are then sorted and plotted, to obtain a curve resembling a lift curve. The larger the area under the curve, the better are the results. Figure 3 shows such curve for the suggested policy, but also for a simpler policy that would choose the Euclidean distance by default, and a random choice policy.

The proposed policy achieves better results than defaulting to the Euclidean norm. It is worth mentioning that the computational costs for obtaining the results of the nearest neighbor model were neglectable compared to the time needed for optimizing one single prediction model.

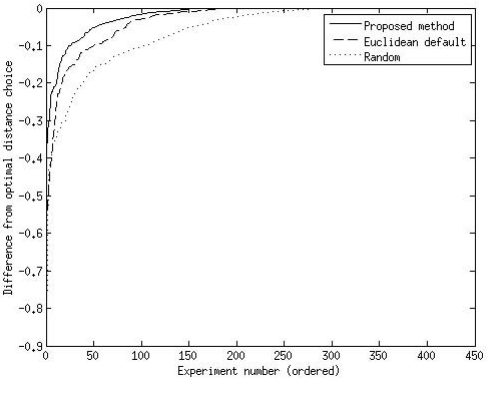


Fig. 3. Ordered differences between the chosen distance and the optimal one according to the suggested policy, compared with the default Euclidean distance and a random choice.

5 Conclusions and Perspectives

The choice of an optimal metric is a choice more and more important to make in modeling, especially when the practitioner is facing complex data. When no prior information is available to choose the most relevant distance measure, the choice can be optimized by resorting to resampling methods, adding a layer of complexity to the usual cross-validation loops needed to learn complex models which depend on some meta-parameters, or, most often, it is defaulted to the Euclidean distance.

The approach which was developed in this chapter aims at finding a compromise between both extremes, using ideas similar to the ones developed in filters methods for feature selection and landmarking approaches to meta learning. The idea is to assess each candidate distance metric using the performances of a simple nearest-neighbor model priori to building a more elaborate model based on distances (support vector machines with Gaussian kernels, radial basis function networks, and other lazy learning algorithms.) Ties are resolved by defaulting to the Euclidean distance.

The experiments show that, although this approach does not allow finding the optimal metric for all datasets and all models, it proves being a reasonable heuristic providing, at a reasonable cost, hints and information about which distance metric to use. The experiments furthermore show that the choice of the optimal metric is rather constant across distance-based models, with maybe the exception of the support vector machine whose learning algorithm may suffer drawbacks from not using the Euclidean distance because not all metrics lead to Mercer kernels.

In most cases, when the optimal metric was not found using the suggested approach, the results obtained using the suggested metric were close enough to the optimal ones to justify relying on the the method.

Although the approach was tested in this study only for Minkowski and fractional metrics, it could be extended for other choices of the metric, and future work will consist in developing a similar approach for the choice of the kernel in kernel-based methods, a problem which is very closely related to the choice of the distance measure.

References

1. Scholkopf, B., Smola, A.J.: Learning with Kernels: Support Vector Machines, Regularization, Optimization, and Beyond. MIT Press, Cambridge (2001)
2. Park, J., Sandberg, I.W.: Universal approximation using radial basis function networks. Neural Computations 3, 246–257 (1991)
3. Aha, D., Kibler, D., Albert, M.: Instance-based learning algorithms. Machine Learning 6, 37–66 (1991)
4. Deza, M.-M., Deza, E.: Dictionary of Distances. Elsevier Science, Amsterdam (2006)
5. François, D.: High-dimensional data analysis: from optimal metrics to feature selection. VDM Verlag Dr. Muller (2008)
6. Battiti, R.: Using the mutual information for selecting features in supervised neural net learning. IEEE Transactions on Neural Networks 5, 537–550 (1994)
7. Pfahringer, B., Bensusan, H., Giraud-Carrier, C.: Meta-learning by landmarking various learning algorithms. In: Proceedings of the Seventeenth International Conference on Machine Learning, ICML 2000, pp. 743–750. Morgan Kaufmann, San Francisco (2000)

8. Navarro, G.: A guided tour to approximate string matching. ACM Computing Surveys 33(1), 31–88 (2001)
9. Yen, L., Saerens, M., Mantrach, A., Shimbo, M.: A family of dissimilarity measures between nodes generalizing both the shortest-path and the commute-time distances. In: Proceeding of the 14th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD 2008, pp. 785–793. ACM, New York (2008)
10. François, D., Wertz, V., Verleysen, M.: The concentration of fractional distances. IEEE Transactions on Knowledge and Data Engineering 19(7), 873–886 (2007)
11. Moody, J.E., Darken, C.: Fast learning in networks of locally-tuned processing units. Neural Computation 1, 281–294 (1989)
12. Orr, M.J.L.: Regularisation in the selection of radial basis function centres. Neural Computation 7(3), 606–623 (1995)
13. Suykens, J., Van Gestel, T., De Brabanter, J., De Moor, B., Vandewalle, J.: Least Squares Support Vector Machines. World Scientific, Singapore (2002)
14. Yu, K., Ji, L., Zhang, X.: Kernel nearest-neighbor algorithm. Neural Processing Letters 15(2), 147–156 (2002)
15. Aggarwal, C.C., Hinneburg, A., Keim, D.A.: On the surprising behavior of distance metrics in high dimensional space. In: Van den Bussche, J., Vianu, V. (eds.) ICDT 2001. LNCS, vol. 1973, pp. 420–434. Springer, Heidelberg (2000)
16. Kohavi, R., John, G.H.: Wrappers for feature subset selection. Artificial Intelligence 97(1-2), 273–324 (1997)
17. Guyon, I., Elisseeff, A.: An introduction to variable and feature selection. Journal of Machine Learning Research 3, 1157–1182 (2003)
18. Stefánsson, A., Koncar, N., Jones, A.J.: A note on the gamma test. Neural Computing & Applications 5(3), 131–133 (1997)
19. Reyhani, N., Hao, J., Ji, Y., Lendasse, A.: Mutual information and gamma test for input selection. In: European Symposium on Artificial Neural Networks, ESANN 2005, Bruges, Belgium, April 27-29, pp. 503–508 (2005)
20. Berchtold, S., Bohm, C., Keim, D.A., Kriegel, H.-P.: A cost model for nearest neighbor search in high-dimensional data space. In: 16th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS), Tucson, Arizona, USA, May 12-14, pp. 78–86. ACM Press, New York (1997)
21. Friedman, J.: Multivariate adaptive regression splines (with discussion). Annals of Statistics 9(1), 1–141 (1991)
22. Borggaard, C., Thodberg, H.H.: Optimal minimal neural interpretation of spectra. Analytical Chemistry 64, 545–551 (1992)
23. Asuncion, A., Newman, D.J.: UCI machine learning repository. School of Information and Computer Sciences. University of California, Irvine (2007)
24. Ong, C.S., Mary, X., Canu, S., Smola, A.J.: Learning with non-positive kernels. In: Proceedings of the Twenty-First International Conference on Machine Learning, ICML 2004, p. 81. ACM Press, New York (2004)

Meta-Learning Architectures: Collecting, Organizing and Exploiting Meta-Knowledge

Joaquin Vanschoren

Department of Computer Science, K.U. Leuven, Leuven, Belgium

1 Introduction

While a valid intellectual challenge in its own right, meta-learning finds its real *raison d'être* in the practical support it offers Data Mining practitioners [20]. Indeed, the whole point of understanding how to learn in any given situation is to go out in the real world and learn as much as possible, from any source of data we encounter! However, almost any type of raw data will initially be very hard to learn from, and about 80% of the effort in discovering useful patterns lies in the clever preprocessing of data [47]. Thus, for machine learning to become a tool we can instantly apply in any given situation, or at least to get proper guidance when applying it, we need to build extended meta-learning systems that encompass the entire knowledge discovery process, from raw data to finished models, and that keep learning, keep accumulating meta-knowledge, every time they are presented with new problems.

The algorithm selection problem is thus widened into a *workflow creation* problem, in which an entire stream of different processes needs to be proposed to the end user. This entails that our collection of meta-knowledge must also be extended to characterize all those different processes.

In this chapter, we provide a survey of the various architectures that have been developed, or simply proposed, to build such extended meta-learning systems. They all consist of integrated repositories of meta-knowledge on the KD process and leverage that information to propose useful workflows. Our main observation is that most of these systems are very different, and were seemingly developed independently from each other, without really capitalizing on the benefits of prior systems. By bringing these different architectures together and highlighting their strengths and weaknesses, we aim to reuse what we have learned, and we draw a roadmap towards a new generation of KD support systems.

Despite their differences, we can classify the KD systems in this chapter in the following groups, based on the way they leverage the obtained meta-data:

Expert systems: In expert systems, experts are asked to express their reasoning when tackling a certain problem. This knowledge is then converted into a set of explicit rules, to be automatically triggered to guide future problems.

Meta-models: The goal here is to automatically predict the usefulness of workflows based on prior experience. They contain a meta-model that is updated

N. Jankowski et al. (Eds.): Meta-Learning in Computational Intelligence, SCI 358, pp. 117–155.
springerlink.com

as more experience becomes available. It is the logical extension of meta-learning to the KD process.

Case-based reasoning: In general terms, case-based reasoning (CBR) is the process of solving new problems based on the solutions of similar past problems. This is very similar to the way most humans with some KD experience would tackle the problem: remember similar prior cases and adapt what you did then to the new situation. To mimic this approach, a knowledge base is populated by previously successful workflows, annotated with meta-data. When a new problem presents itself, the system will retrieve the most similar cases, which can then be altered by the user to better fit her needs.

Planning: All possible actions in a KD workflow are described as operations with preconditions and effects, and an AI planner is used to find the most interesting plans (workflows).

Querying: In this case, meta-data is gathered and organized in such a way that users can ask any kind of question about the utility or general behavior of KD processes in a predefined query language, which will then be answered by the system based on the available meta-data. They open up the available meta-data to help users make informed decisions.

Orthogonal to this distinction, we can also characterize the various systems by the *type of meta-knowledge* they store, although in some cases, a combination of these sources is employed.

Expert knowledge: Rules, models, heuristics or entire KD workflows are entered beforehand by experts, based on their own experience with certain learning approaches.

Experiments: Here, the meta-data is purely empirical. They provide objective assessments of the performance of workflows or individual processes on certain problems.

Workflows: Workflows are descriptions of the entire KD process, involving linear or graph-like sequences of all the employed preprocessing, transformation, modeling and postprocessing steps. They are often annotated with simple properties or qualitative assessments by users.

Ontologies: An *ontology* is a formal representation of a set of concepts within a domain and the relationships between those concepts [9]. They provide a fixed vocabulary of data mining concepts, such as algorithms and their components, and describe how they relate to each other, e.g. what the role is of a specific component in an algorithm. They can be used to create unambiguous descriptions of meta-knowledge which can then be interpreted by many different systems to reason about the stored information.

While we cover a wide range of approaches, the landscape of all meta-learning and KD solutions is quite extensive. However, most of these simply facilitate access to KD techniques, sometimes offering wizard-like interfaces with some expert advice, but they essentially leave the intelligent selection of techniques as an exercise to the user. Here, we focus on those systems that introduce new ways of leveraging meta-knowledge to offer intelligent KD support. We also skip

KD systems that feature some type of knowledge base to monitor the current workflow composition, but that do not use that knowledge to advise on future problems, such as the INLEN system [44,33,34]. This overview partially overlaps with previous, more concise overviews of meta-learning systems for KD [7,20]. Here, however, we provide a more in-depth discussion of their architectures, focussing on those aspects that can be reused to design future KD support systems.

In the remainder of this chapter, we will split our discussion in two: past and future. The past consists of all previously proposed solutions, even though some of these are still in active development and may be extended further. This will cover the following five sections, each corresponding to one of the five approaches outlined above. For each system, we consecutively discuss its architecture, the employed meta-knowledge, any meta-learning that is involved and finally its benefits and drawbacks. Finally, in Section 7, we provide a short summary of all approaches, before looking towards the future: we outline a platform for the development of future KD support systems aimed at bringing the best aspects of prior systems together.

2 Expert Systems

2.1 Consultant-2

One of the first inroads into systematically gathering and using meta-knowledge about machine learning algorithms was Consultant-2 [14,57]: an expert system developed to provide end-user guidance for the MLT machine learning toolbox [46,37]. Although the system did not learn by itself from previous algorithm runs, it did identify a number of important meta-features of the data and the produced models, and used these to express rules about the applicability of algorithms.

Architecture. A high-level overview of the architecture of Consultant-2 is shown in Figure 1. It was centered around a knowledge base that stored about 250 rules, hand-crafted by machine learning experts. The system interacted with the user through question-answer sessions in which the user was asked to provide information about the given data (e.g. the number of classes or whether it could be expected to be noisy) and the desired output (e.g. rules or a decision tree). Based on that information, the system then used the stored rules to calculate a score for each algorithm. The user could also go back on previous answers to see how that would have influenced the ranking. When the user selected an algorithm, the system would automatically run it, after which it would engage in a new question-answer session to assess whether the user was satisfied with the results. If not, the system would generate a list with possible parameter recommendations, again scored according to the stored heuristic rules.

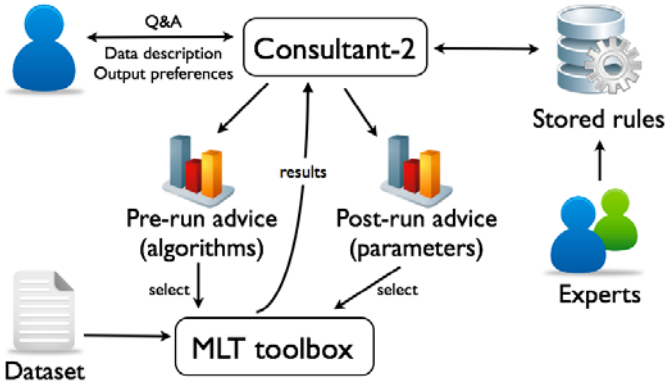


Fig. 1. The architecture of Consultant-2. Derived from Craw et al. [14]

Meta-knowledge. The rules were always of the form $if(A_n)$ *then* B, in which A_n was a conjunction of properties of either the *task description* or the *produced models*, and B expressed some kind of action the system could perform. First, the task description included qualitative measures entered by the user, such as the task type, any available background knowledge and which output models were preferable, as well as quantitative measures such as the number of classes and the amount of noise. Second, the produced models where characterized by the amount of time and memory needed to build them and model-specific features such as the average path length and number of leaf nodes in decision trees, the number and average center distance of clusters and the number and significance of rules.[1] The resulting actions B could either adjust the scores for specific algorithms, propose ranges for certain parameter values, or transform the data (e.g. discretization) so as to suit the selected algorithm. Illustrations of these rules can be found in Sleeman at al. [57].

Meta-learning. There is no real meta-learning process in Consultant-2, it just applied its predefined 'model' of the KD process. This process was divided into a number of smaller steps (selecting an algorithm, transforming the data, selecting parameters,...), each associated with a number of rules. It then cycles through that process, asking questions, executing the corresponding rules, and triggering the corresponding actions, until the user is satisfied with the result.

Discussion. The expert advice in Consultant-2 has proven to be successful in a number of applications, and a new version has been proposed to also provide guidance for data preprocessing. Still, to the best of our knowledge, Consultant-3 has never been implemented. An obvious drawback of this approach is the fact that the heuristic rules are hand-crafted. This means that for every new

[1] These were used mostly to advise on the perspective algorithm's parameters.

algorithm, new rules have to be defined that differentiate it from all the existing algorithms. One must also keep in mind that MLT only had a limited set of 10 algorithms covering a wide range of tasks (e.g. classification, regression and clustering), making it quite feasible to select the correct algorithm based on the syntactic properties of the input data and the preferred output model. With the tens or hundreds of methods available today on classification alone, it might not be so straightforward to make a similar differentiation. Still, the idea of a database of meta-rules is valuable, though instead of manually defining them, they should be learned and refined automatically based on past algorithm runs. Such a system would also automatically adapt as new algorithms are added.

3 Meta-Models

3.1 STABB and VBMS

The groundwork for many meta-learning systems were laid by two early precursors. STABB [61] was a system that basically tried to automatically match a learner's bias to the given data. It used an algorithm called LEX with a specific grammar. When an algorithm could not completely match the hidden concept, its bias (the grammar) or the structure of the dataset was changed until it could. VBMS [52] was a very simple meta-learning system that tried to select the best of three learning algorithms using only two meta-features: the number of training instances and the number of features.

3.2 The Data Mining Advisor (DMA)

The Data Mining Advisor (DMA) [19] is a web-based algorithm recommendation system that automatically generates rankings of classification algorithms according to user-specified objectives. It was developed as part of the METAL project [43].

Foundations: StatLog. Much of the theoretical underpinning of the DMA was provided by the StatLog project [45], which was aimed to provide a large-scale, objective assessment of the strengths and weaknesses of the various classification approaches existing at the time. Its methodology is shown in Figure 2.

First, a wide range of datasets are characterized with a wide range of newly defined meta-features. Next, the algorithms are evaluated on these datasets. For each dataset, all algorithms whose error rate fell within a certain (algorithm-dependent) margin of that of the best algorithms were labeled *applicable*, while the others were labeled *non-applicable*. Finally, decision trees were built for each algorithm, predicting when it can be expected to be useful on a new dataset. The resulting rules, forming a rule base for algorithm selection, can be found in Michie et al. [45].

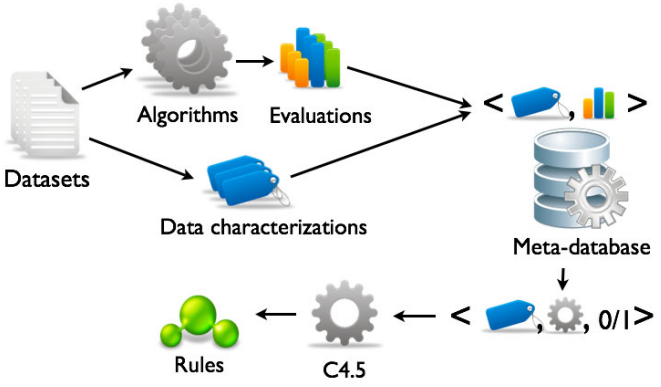


Fig. 2. The StatLog approach.

Architecture. The architecture of the DMA is shown in Figure 3. First, the tool is trained by generating the necessary meta-data, just as in StatLog. However, instead of predicting whether an algorithm was applicable or not, it *ranked* all of them. New datasets can be uploaded via a web-interface, its meta-features automatically computed (below the dotted line in Figure 3). Bearing privacy issues in mind, the user can choose to keep the dataset, the data characterizations or both hidden from other users of the system. Next, the DMA will use the stored meta-data to predict the expected ranking of all algorithms on the new dataset. Finally, as a convenience to the user, one can also run a number of algorithms on the new dataset, after which the system returns their true ranking and performance. Figure 4 shows an example of the rankings generated by DMA. The meaning of the 'Predicted Score' is explained in Section 3.2.

Meta-knowledge. While the METAL project discovered various new kinds of meta-features, these were not used in the DMA tool. In fact, only a set of 7 numerical StatLog-style characteristics was selected[2] for predicting the relative performance of the algorithms. This is most likely due to the employed meta-learner, which is very sensitive to the curse of dimensionality.

At its inception, the DMA tool was initialized with 67 datasets, mostly from the UCI repository. Since then, an additional 83 tasks were uploaded by users [19]. Furthermore, it operates on a set of 10 classification algorithms, shown in Figure 4, but only with default parameter settings, and evaluates them based on their predictive accuracy and speed.

Meta-learning. DMA uses a k-nearest neighbors (kNN) approach as its meta-learner, with k=3 [8]. This is motivated by the desire to continuously add new meta-data without having to rebuild the meta-model every time.

[2] Some of them were slightly modified. For instance, instead of using the absolute number of symbolic attributes, DMA uses the *ratio* of symbolic attributes.

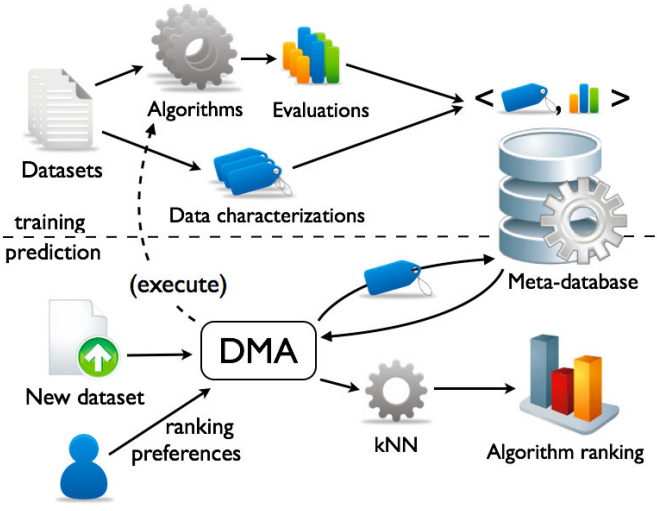


Fig. 3. The architecture of DMA. Adapted from Brazdil at al. [7].

Ranking table

Dowload results | register now! | learn more about the online advisor

Predicted Rank	Algorithm	Predicted Score	Status	Run	Accuracy	Time	True Rank	True Score
1.	c50rules	1.031	finished	--	0.2830000	?	6	1.003
2.	lindiscr	1.03	finished	--	0.2340000	?	1	1.048
3.	c50tree	1.026	--		--	--	--	--
4.	ltree	1.023	--		--	--	--	--
5.	clemMLP	1.017	finished	--	0.2680000	?	5	1.006
6.	c50boost	1.017	--		--	--	--	--
7.	ripper	1.009	--		--	--	--	--
8.	mlcnb	1	finished	--	0.2430000	?	2	1.036
9.	clemRBFN	0.948	--		--	--	--	--
10.	mlcib1	0.913	finished	--	0.3180000	?	10	0.938

(a) Emphasis on Accuracy

Ranking table

Dowload results | register now! | learn more about the online advisor

Predicted Rank	Algorithm	Predicted Score	Status	Run	Accuracy	Time	True Rank	True Score
1.	lindiscr	1.153	finished	--	0.2340000	?	1	1.154
2.	c50tree	1.144	finished	--	0.2940000	?	2	1.101
3.	c50rules	1.07	finished	--	0.2830000	?	4	1.058
4.	ltree	1.061	--		--	--	--	--
5.	mlcnb	1.051	--		--	--	--	--
6.	c50boost	1.009	--		--	--	--	--
7.	ripper	1.002	--		--	--	--	--
8.	clemMLP	0.909	--		--	--	--	--
9.	mlcib1	0.903	finished	--	0.3180000	?	8	0.931
10.	clemRBFN	0.842	--		--	--	--	--

(b) Emphasis on Training Time

Fig. 4. An illustration of the output of the DMA. The first ranking favors accurate algorithms, the second ranking favors fast ones. Taken from Giraud-Carrier [19].

A multi-criteria evaluation measure is defined (the *adjusted ratio of ratios*) which allows the user to add emphasis either on the predictive accuracy or the training time. This is done by defining the weight *AccD* of the training time ratio. This allows the user to trade *AccD*% accuracy for a 10-time increase/decrease in training time. The DMA interface offers 3 options: *AccD*=0.1 (focus on accuracy), *AccD*=10 (focus on speed) or *AccD*=1 (equal importance). The user can also opt not to use this measure and use the *efficiency measure* of Data Envelopment Analysis (DEA) instead [19]. The system then calculates a kind of average over the 3 most similar datasets for each algorithm (the 'predicted score' in Figure 4) according to which the ranking is established.

Discussion. The DMA approach brings the benefits of meta-learning to a larger audience by automating most of the underlying steps and allowing the user to state some preferences. Moreover, it is able to continuously improve its predictions as it is given more problems (which are used to generate more meta-data). Unfortunately, it has a number of limitations that affect the practical usefulness of the returned rankings. First, it offers no advice about which preprocessing steps to perform or which parameter values might be useful, which both have a profound impact on the relative performance of learning algorithms. Furthermore, while the multi-criteria ranking is very useful, the system only records two basic evaluation metrics, which might be insufficient for some users. Finally, using kNN means no interpretable models are being built, making it a purely predictive system. Several alternatives to kNN have been proposed [50,3] which may be useful in future incarnations of this type of system.

3.3 NOEMON

NOEMON [31,32], shown in Figure 5, follows the approach of Aha [1] to compare algorithms two by two. Starting from a meta-database similar to the one used in

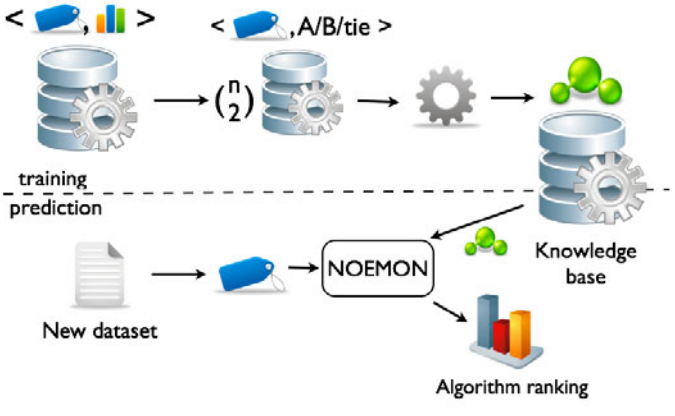


Fig. 5. NOEMON's architecture. Based on Kalousis and Theoharis [32].

DMA, but with histogram-representations of attribute-dependent features, the performance results on every combination of two algorithms are extracted. Next, the performance values are replaced with a statistical significance tests indicating when one algorithm significantly outperforms the other and the number of data meta-features is reduced using automatic feature selection. This data is then fed to a decision tree learner to build a model predicting when one algorithm will be superior or when they will tie on a new dataset. Finally, all such pairwise models are stored in a knowledge base.

At prediction-time, the system collects all models concerning a certain algorithm and counts the number of predicted wins/ties/losses against all other algorithms to produce the final score for each algorithm, which is then converted into a ranking.

4 Planning

The former two systems are still limited in that they only tackle the algorithm selection step. To generate advice on the composition of an entire workflow, we need additional meta-data on the rest of the KD processes. One straightforward type of useful meta-data consists of the preconditions that need to be fulfilled before the process can be used, and the effect it has on the data it is given. As such, we can transform the workflow creation problem into a *planning* problem, aiming to find the best plan, the best sequence of actions (process applications), that arrives at our goal - a final model.

4.1 The Intelligent Discovery Electronic Assistant (IDEA)

A first such system is the Intelligent Discovery Electronic Assistant (IDEA) [5]. It regards preprocessing, modeling and postprocessing techniques as operators, and returns all plans (sequences of operations) that are possible for the given problem. It contains an ontology describing the preconditions and effects of each operator, as well as manually defined heuristics (e.g. the speed of an algorithm), which allows it to produce a ranking of all generated plans according to the user's objectives.

Architecture. The architecture of IDEA is shown in Figure 6. First, the systems gathers information about the given task by characterizing the given dataset. Furthermore, the user is asked to provide additional metadata and to give weights to a number of heuristic functions such as model comprehensibility, accuracy and speed. Next, the planning component will use the operators that populate the ontology to generate all KD workflows that are valid in the user-described setting. These plans are then passed to a heuristic ranker, which will use the heuristics enlisted in the ontology to calculate a score congruent with the user's objectives (e.g. building a decision tree as fast as possible). Finally, this ranking is proposed to the user which may select a number of processes to be executed on the provided data. After the execution of a plan, the user is allowed to review the results and alter the given weights to obtain new rankings.

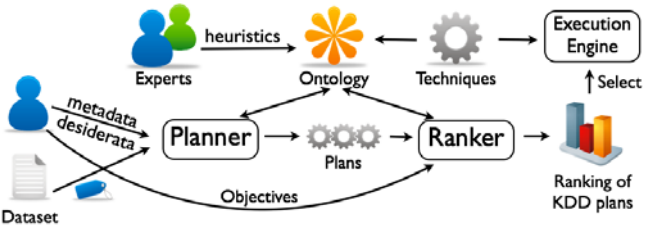


Fig. 6. The architecture of IDEA. Derived from Bernstein et al. [5].

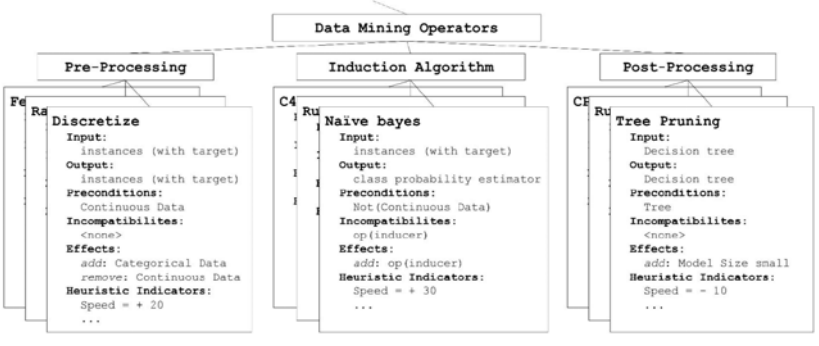


Fig. 7. Part of IDEA's ontology. Taken from Bernstein et al. [5].

For instance, the user might sacrifice speed in order to obtain a more accurate model. Finally, if useful partial workflows have been discovered, the system also allows to extend the ontology by adding them as new operators.

Meta-knowledge. IDEA's meta-knowledge is all contained in its ontology. It first divides all operators into preprocessing, induction or postprocessing operators, and then further into subgroups. For instance, induction algorithms are subdivided into classifiers, class probability estimators and regressors. Each process that populates the ontology is then described further with a list of properties, shown in Figure 7. It includes the required input (e.g. a decision tree for a tree pruner), output (e.g. a model), preconditions (e.g. 'continuous data' for a discretizer), effects (e.g. 'removes continuous data', 'adds categorical data' for a discretizer), incompatibilities (e.g. not(continuous data) for naïve Bayes) and a list of manually defined heuristic estimators (e.g. relative speed). Additionally, the ontology also contains recurring partial plans in the form of composite operators.

Meta-learning. Though there is no actual meta-learning involved, planning can be viewed as a search for the best-fitting plan given the dataset, just as learning is a search for the best hypothesis given the data. In the case of IDEA, this is

achieved by exhaustively generating all possible KD plans, hoping to discover better, novel workflows that experts never considered before. The provided meta-data constitute the initial state (e.g. 'numerical data') and the user's desiderata (e.g. 'model size small') make up the goal state.

Discussion. This approach is very useful in that it can provide both experts and less-skilled users with KD solutions without having to know the details of the intermediate steps. The fact that new operators or partial workflows can be added to the ontology can also give rise to *network externalities*, where the work or expertise of one researcher can be used by others without having to know all the details. The ontology thus acts as a central repository of KD operators, although new operators can only be added manually, possibly requiring reimplementation.

The limitations of this approach lie first of all in the hand-crafted heuristics of the operators. Still, this could be remedied by linking it to a meta-database such as used in DMA and learning the heuristics from experimental data. Secondly, the current implementation only covers a small selection of KD operators. Together with the user-defined objectives, this might constrain the search space well enough to make the exhaustive search feasible, but it is unclear whether this would still be the case if large numbers of operators are introduced. A final remark is that most of the used techniques have parameters, whose effects are not included in the planning.

4.2 Global Learning Scheme (GLS)

The 'Global Learning Scheme' (GLS) [73,71,72] takes a multi-agent system approach to KDD planning. It creates an "organized society of KDD agents", in which each agent only does a small KDD task, controlled by a central planning component.

Architecture. Figure 8 provides an overview of the system. It consists of a pool of agents, which are described very similarly to the operators in IDEA, also using an ontology to describe them formally. However, next to base-level agents, which basically envelop one DM technique each, there also exist *high-level agents*, one for each phase of the KD process, which instead point to a list of candidate agents that could be employed in that phase. The controlling 'meta-agent' (CMA) is the central controller of the system. It selects a number of agents (high-level agents at first) and sends them to the planning meta-agent (PMA). The PMA then creates a planning problem by transforming the dataset properties and user objectives into a world state description (WSD) for planning and by transforming the agents into planning operators. It then passes the problem to a STRIPS planner [17]. The returned plans are passed back to the CMA, which then launches new planning problems for each high-level agent. For instance, say the data has missing values, then the returned plan will contain a high-level missing value imputation agent: the CMA will then send a range of base-level agents to the

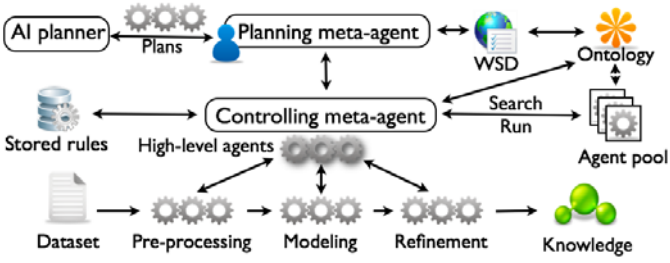


Fig. 8. The architecture of GLS. Adapted from Zhong et al. [72].

PMA to build a plan for filling in the missing values. The CMA also executes the resulting sub-plans, and calls for adaptations if they do not prove satisfactory.

Meta-knowledge. GLS's meta-knowledge is similar to that of IDEA. The ontology description contains, for each agent, the data types of in- and output, preconditions and effects. In the case of a base-level agent, it also contains a KD process, otherwise, a list of candidate sub-agents. However, the same ontology also contains descriptors for the data throughout the KD process, such as the state of the data (e.g. raw, clean, selected), whether or not it represents a model and the type of that model (e.g. regression, clustering, rule). This information is used to describe the world state description. Finally, the CMA contains some static meta-rules to guide the selection of candidate agents.

Meta-learning. In Zong et al. [72], the authors state that a meta-learning algorithm is used in the CMA to choose between several discretization agents or to combine their results. Unfortunately, details are missing. Still, even if the current system does not learn from previous runs (meaning that the CMA uses the same meta-rules every time), GLA's ability to track and adapt to changes performed by the user can definitely be regarded as a form of learning [7].

Discussion. Given the fact that covering the entire KD process may give rise to sequences of tens, maybe hundreds of individual steps, the 'divide and conquer' approach of GLS seems a promising approach. In fact, it seems to mirror the way humans approach the KD process: identify a hierarchy of smaller subproblems, combine operators to solve them and adapt the partial solutions to each other until we converge to a working workflow. Unfortunately, to the best of our knowledge, there are no thorough evaluations of the system, and it remains a work in progress. It would be interesting to replace the fixed rules of the CMA with a meta-learning component to select the most promising agents, and to use IDEA-like heuristics to rank possible plans.

4.3 The Goal-Driven Learner (GDL)

The Goal-Driven Learner[3] [62], is a quite different planning approach, mainly because it operates on a higher level of abstraction. Its task is to satisfy a predefined *goal*, given a number of knowledge bases, a group of human experts, and a collection of KD tools. Each of these (including the experts) are described in a Learning Modeling Language (LML) which describes properties such as their *vocabulary* (e.g. the attribute names in a decision tree or the expert's field of expertise), *solutions* (e.g. classes predicted by the tree or medical treatments) or the *time* it takes them to provide a solution (which is obviously higher for an expert than for a decision tree). Its goal and subgoals are also described in LML, e.g. expressing that a simple model must be built within a given timeframe. The GDL then executes a search, using the 'distance' to each of the subgoals as a heuristic and applications of the KD tools as actions. During this process, it selects useful knowledge systems (e.g. based on their vocabulary), combines them (e.g. after conversion to Prolog clauses), applies KD tools (e.g. to build a new decision tree) or petitions an expert to provide extra information (e.g. if part of the vocabulary cannot be found in the available knowledge systems). It is a very useful approach in settings were many disparate knowledge sources are available, although the LML description is probably still too coarse to produce fine-grained KDD plans.

4.4 Complexity Controlled Exploration (CCE)

Complexity Controlled Exploration [22] is also a high-level system that consists of a number of *machine generators*, which generate KD workflows, and a search algorithm based on a measure for the *complexity* of the proposed KD workflows. While not performing planning per se, many of the generators could be planners such as IDEA and GLS. Alternatively, they may simply consist of a list of prior solutions, or *meta-schemes*: partial workflows in which certain operators are left blank to be filled in with a suitable operator later on (also see 'templates' in Section 6.1).

The algorithm asks all generators to propose new KD processes, and to rank them using an adapted complexity measure [39,40] defined as $c(p) = [l(p) + log(t(p))] \cdot q(p)$, in which p is a *program* (a KD workflow), $l(p)$ the length of that program, $t(p)$ the estimated runtime of the program, and $q(p)$ the inverse of the *reliability* of the program, which reflects their usefulness in prior tasks. For instance, if a certain combination of a feature selection method and a classifier proved very useful in the past, it may obtain an improved reliability value. If the complexity cannot be estimated, it is approximated by taking the weighted average of past observations of the program on prior inputs. Note that a 'long'

[3] Goal-driven learning is also a more general AI concept, in which the aim is to use the overall goals of an intelligent system to make decisions about when learning should occur, what should be learned, and which learning strategies are appropriate in a given context.

program can still be less complex than a 'short' one, e.g. if a feature selection step heavily speeds up the runtime of a learning algorithm.

The CCE algorithm then takes the least complex solution from each generator, adds the least complex of those to a queue (until it is full), and updates the *complexity threshold*, the maximum of all proposed workflows. The workflows in the queue are then executed in parallel, but are aborted if they exceed the complexity threshold and are added, with an increased complexity value, to the 'quarantine generator', which may propose them again later on. The algorithm's stopping criterion is set by the user: it may be the best solution in a given time, below a given complexity threshold, or any custom test criterion.

The CCE turns KD workflow selection into an adaptive process in which a number of other systems can be used as generators, and in which additional meta-data can be added both in the generators (e.g. new meta-schemes) as well as in the complexity estimation procedure. Instead of proposing workflows and leaving it to the user to decide which ones to try next, it also runs them and returns the optimal solution. Its success is thus highly dependent on the availability of good generators, and on how well its complexity controlled exploration matches the choices that the user would make.

5 Case-Based Reasoning

Planning is especially useful when starting from scratch. However, if successful workflows were designed for very similar problems, we could simply reuse them.

5.1 CITRUS

CITRUS [16,68] is built as an advisory component of Clementine, a well-known KDD suite. An overview is shown in Figure 9. It contains a knowledge base of available 'processes' (KD techniques) and 'streams' (sequences of processes), entered by experts and described with pre- and postconditions. To use it, the user has to provide an abstract task description, which is appended with simple data statistics, and choose between several modi of operation. In the first option, the user simply composes the entire KDD process (stream) by linking processes in Clementine's editor, in which case CITRUS only checks the validity of the sequence. In the second option, case-based reasoning is used to propose the most similar of all known streams. In the third option, CITRUS assists the user in decomposing the task into smaller subtasks, down to the level of individual processes. While pre- and postconditions are used in this process, no planning is involved. Finally, the system also offers some level of algorithm selection assistance by eliminating those processes that violate any of the constraints.

5.2 ALT

ALT [41] is a case-based reasoning variation on the DMA approach. Next to StatLog-type meta-features, it adds a number of simple algorithm characterizations. It has a meta-database of 'cases' consisting of data meta-features, algorithm

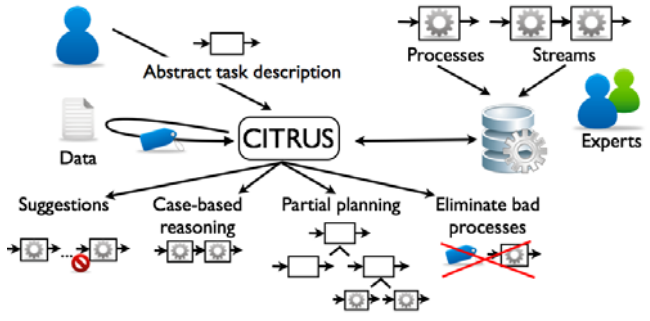


Fig. 9. The architecture of CITRUS. Derived from Wirth et al. [68].

meta-features and the performance of the algorithm. When faced with a new problem, the user can enter application restrictions beforehand (e.g. 'the algorithm should be fast and produce interpretable models'), and this information is then appended to the data meta-features of the new dataset, thus forming a new case. The system then makes a prediction based on the three most similar cases. The meta-data consists of 21 algorithms, 80 datasets, 16 StatLog data characteristics, and 4 algorithm characteristics.

The two previous approaches share a common shortcoming: the user still faces the difficult task of *adapting* it to her specific problem. The following systems try to alleviate this problem by offering additional guidance.

5.3 MiningMart

The MiningMart project [47] is designed to allow successful preprocessing workflows to be shared and reused, irrespective of how the data is stored. While most of the previously discussed systems expect a dataset in a certain format, MiningMart works directly on any SQL database of any size. The system performs the data preprocessing and transformation steps in the (local) database itself, either by firing SQL queries or by running a fixed set of efficient operators (able to run on very large databases) and storing the results in the database again. Successful workflows can be shared by describing them in an XML-based language, dubbed M4, and storing them in an online case-base. The case description includes the workflow's inherent structure as well as an ontological description of business concepts to facilitate searching for cases designed for certain goals or applications.

Architecture. The architecture of the system is shown in Figure 10. To map uniformly described preprocessing workflows to the way data is stored in specific databases, it offers three levels of abstraction, each provided with graphical editors for the end-user. We discuss them according to the viewpoint of the users who wish to reuse a previously entered case. First, they use the business ontology to search for cases tailored to their specific domain. The online interface

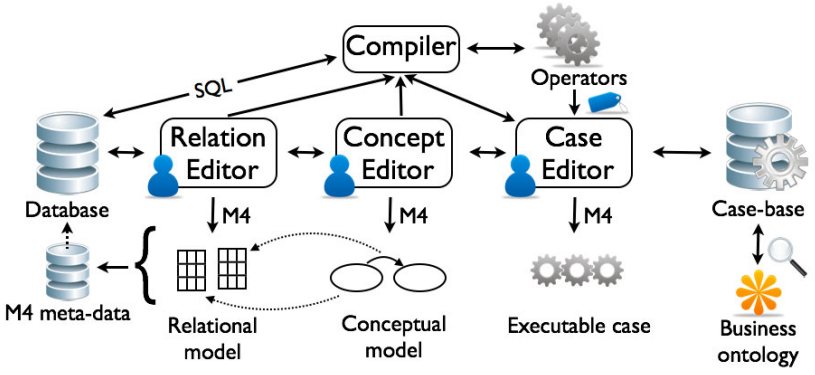


Fig. 10. MiningMart architecture. Derived from Morik and Scholz [47].

then returns a list of cases (workflows). Next, they can load a case into the *case editor* and adapt it to her specific application. This can be done on an abstract level, using abstract concepts and relations such as 'customer', 'product' and the relation 'buys'. They can each have a range of properties, such as 'name' and 'address', and can be edited in the *concept editor*. In the final phase, these concepts, relations and properties have to be mapped to tables, columns, or sets of columns in the database using the *relation editor*. All details of the entire process are expressed in the M4 language[4] and also stored in the database. Next, the *compiler* translates all preprocessing steps to SQL queries or calls to the operators used, and executes the case. The user can then adapt the case further (e.g. add a new preprocessing step or change parameter settings) to optimize its performance. When the case is finished, it can be annotated with an ontological description and uploaded to the case-base for future guidance.

Meta-knowledge. MiningMart's meta-knowledge can be divided in two groups. First, there is the fine-grained, case-specific meta-data that covers all the meta-data entered by the user into the M4 description, such as database tables, concepts, and workflows. Second, there is more general meta-knowledge encoded in the business ontology, i.e. informal annotations of each case in terms of its goals and constraints, and in the description of the operators. Operators are stored in a hierarchy consisting of 17 'concept' operators (from selecting rows to adding columns with moving windows over time series), 4 feature selection operators, and 20 feature construction operators, such as filling in missing values, scaling, discretization and some learning algorithms used for preprocessing: decision trees, k-means and SVMs. The M4 schema also captures known preconditions and assertions of the operators, similar to the preconditions and effects of

[4] Examples of M4 workflows can be downloaded from the online case-base: http://mmart.cs.uni-dortmund.de/caseBase/index.html

IDEA and GLS. Their goal is to guide the construction of valid preprocessing sequences, as was done in CITRUS. The case-base is available online [47] and currently contains 6 fully-described cases.

Meta-learning. While it is a CBR approach, there is no automatic recommendation of cases. The user can only browse the cases based on their properties manually.

Discussion. The big benefit of the MiningMart approach is that users are not required to transform the data to a fixed, often representationally limited format, but can instead adapt workflows to the way the source data is actually being stored and manipulated. Moreover, a common language to describe KD processes would be highly useful to exchange workflows between many different KD environments. Pooling these descriptions would generate a rich collection of meta-data for meta-learning and automated KD support. M4 is certainly a step forward in the development of such a language.

Compared to IDEA, MiningMart focusses much more on the preprocessing phase, while the former has a wider scope, also covering model selection and postprocessing steps. Next, while both approaches describe their operators with pre- and postconditions, MiningMart only uses this information to guide the user, not for automatic planning. MiningMart could, in principle, be extended with an IDEA-style planning component, at least if the necessary meta-data can also be extracted straight from the database.

There are also some striking similarities between MiningMart and GLS. GLS's agents correspond to MiningMart's operators and its controller (CMA) corresponds to MiningMart's compiler. Both systems use a hierarchical description of agents/operators, which are all described with pre- and postconditions. The differences lie in the scope of operators (MiningMart focuses of preprocessing while GLS wants to cover the entire KD process), the database integration (MiningMart interfaces directly with databases while GLS requires the data to be prepared first) and in storing the meta-data of the workflows for future use, which MiningMart supports, but GLS doesn't.

5.4 The Hybrid Data Mining Assistant (HDMA)

The Hybrid Data Mining Assistant[5] (HDMA) [11,12,13] also tries to provide advice for the entire knowledge discovery process using ontological (expert) knowledge, as IDEA does, but it does not provide a ranking of complete processes. Instead, it provides the user with expert advice during every step of the discovery process, showing both the approach used in similar cases and more specific advice for the given problem triggered by ontological knowledge and expert rules.

[5] This is not the official name, we only use it here to facilitate our discussion.

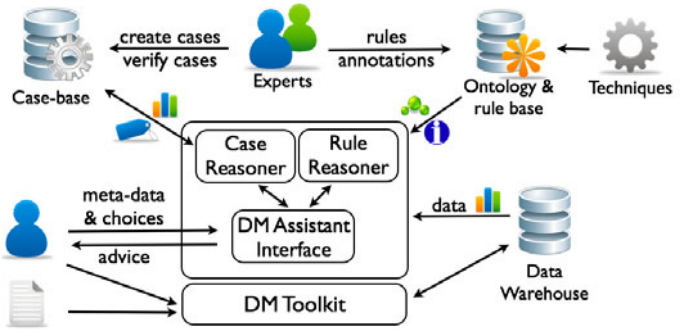


Fig. 11. The architecture of HDMA. Adapted from Charest et al. [13].

Architecture. An overview of the system is provided in Figure 11. The provided advice is based on two stores of information. The first is a repository of KDD 'cases', detailing previous workflows, while the second is an ontology of concepts and techniques used in a typical KDD scenario and a number of rules concerning those techniques. Furthermore, the system is assumed to be associated with a DM toolkit for actually running and evaluating the techniques.

The user first provides a dataset, which is characterized partly by the system, partly by the user (see below). The given problem is then compared to all the stored cases and two scores are returned for each case, one based on similarity with the current case, the other based on the 'utility' of the case, based on scores provided by previous users. After the user has selected a case, the system starts cycling through five phases of the KDD process, as identified by the CRISP-DM [10] model. At each phase the user is provided with the possible techniques that can be used in that phase, the techniques that were used in the selected case, and a number of recommendations generated by applying the stored rules in the context of the current problem. The generated advice may complement, encourage, but also advice against the techniques used in the selected case. As such, the user is guided in adapting the selected case to the current problem.

Meta-knowledge. In the case base, each case is described by 66 attributes. First, the given problem is described by 30 attributes, including StatLog-like meta-features, but also qualitative information entered by users, such as the type of business area (e.g. engineering, marketing, finance,...) or whether the data can be expected to contain outliers. Second, the proposed workflow is described by 31 attributes, such as the used preprocessing steps, how outliers were handled, which model was used, which evaluation method was employed and so on. Finally, 5 more attributes evaluate the outcome of the case, such as the level of satisfaction with the approach used in each step of the KD process. A number of seed cases were designed by experts, and additional cases can be added after evaluation.

The ontology, on the other hand, captures a taxonomy of DM concepts, most of which where elicited from CRISP-DM. A small part is shown in Figure 12.

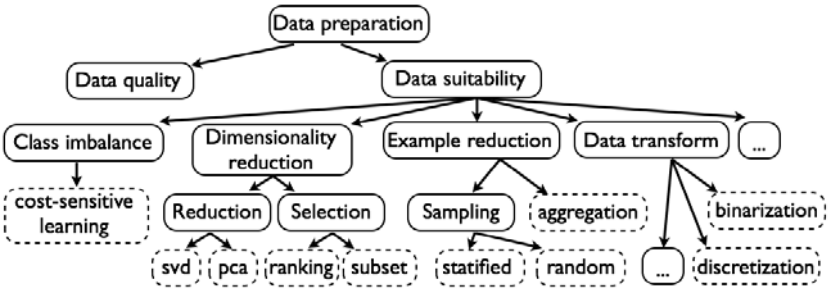


Fig. 12. Part of the HDMA ontology. Adapted from Charest et al. [13].

For instance, it shows that binarization and discretization are two data transformation functions, used to make the data more suitable to a learning algorithm, which is part of the data preparation phase of the CRISP-DM model. The dashed concepts are the ones for which exist specific KD-techniques (individuals). These individuals can be annotated with textual recommendations and heuristics, and also feature in expert rules that describe when they should be used, depending on the properties of the given problem. Some of these rules are heuristic (e.g. "use an *aggregation* technique if the *example count* is greater than 30000 and the data is of a *transactional* nature"), while others are not (e.g. "if you select the *Naive Bayes* technique, then a *nominal target* is required"). In total, the system contains 97 concepts, 58 properties, 63 individuals, 68 rules and 42 textual annotations. All knowledge is hand-crafted by experts and formalized in the OWL-DL ontology format [15] and the SWRL rule description language [28].

Meta-learning. The only meta-learning occurring in this system is contained in the case based reasoning. It uses a kNN approach to select the most similar case based on the characterization of the new problem, using a feature-weighted, global similarity measure. The weight of each data characteristic is pre-set by experts.

Discussion. HDMA is quite unique in that it leverages both declarative information, viz. concepts in the ontology and case descriptions, as well as procedural information in the form of rules. Because of the latter, it can be seen as a welcome extension of Consultant-2. It doesn't solve the problem of (semi-)automatically finding the right KD approach, but it provides practical advice to the user during every step of the KD process. Its biggest drawback is probably that it relies almost entirely on input from experts. Besides from the fact that the provided rules and heuristics may not be very accurate in some cases, this makes it hard to maintain the system. For every new technique (or just a variant of an existing one), new rules and concepts will have to be defined to allow the system to return proper advice. As with Consultant-2, some of these issues may be resolved by introducing more meta-learning into the system, e.g. advice on the proper

weights for each data characteristic in the case based reasoning step, to update the heuristics in the rules and to find new rules based on experience. Finally, the case-based advice, while useful in the first few steps, may lose its utility in the later stages of the KD process. More specifically, say the user chooses a different preprocessing step as applied in the proposed case, then the subsequent stages of that case (e.g. model selection) may lose their applicability. A possible solution here would be to update the case selection after each step, using the partial solution to update the problem description.

5.5 NExT

NExT, the Next generation Experiment Toolbox [4] is an extension of the IDEA approach to case-based reasoning and to the area of *dynamic processes*, in which there is no guarantee that the proposed workflow will actually work, or even that the atomic tasks of which it consists will execute without fault. First, it contains a knowledge base of past workflows and uses case-based reasoning to retrieve the most similar ones. More often than not, only parts of these workflows will be useful, leaving holes which need to be filled with other operators. This is where the planning component comes in: using the preconditions and effects of all operators, and the starting conditions and goals of the KD problem, it will try to find new sequences of operators to fill in those holes.

However, much more can go wrong when reusing workflows. For instance, a procedure may have a parameter setting that was perfect for the previous setting, but completely wrong for the current one. Therefore, NExT has an ontology of possible problems related to workflow execution, including resolution strategies. Calling a planner is one such strategy, other strategies may entail removing operators, or even alerting the user to fix the problem manually.

Finally, it does not start over each time the workflow breaks. It records all the data processed up to the point where the workflow breaks, tries to resolve the issue, and then continues from that point on. As such, it also provides an online log of experimentation which can be shared with other researchers willing to reproduce the workflow.

NExT has only recently been introduced and thorough evaluations are still scarce. Nevertheless, its reuse of prior workflows and semi-automatic adaption of these workflows to new problems seems a very promising.

6 Querying

A final way to assist users is to automatically answer any kind of question they may have about the applicability, general utility or general behavior of KD processes, so that they can make informed decisions when creating or altering workflows. While the previous approaches only stored a selection of meta-data necessary for the given approach, we could collect a much larger collection of meta-data of possible interest to users, and organize all this data to allow the user to write queries in a predefined query language, which will then be answered

by the system based on the available meta-data. While this approach is mostly useful for experts knowing how to ask the questions and interpret the results, frequently asked questions could be wrapped in a simpler interface for use by a wider audience.

6.1 Advanced Meta-Learning Architecture (AMLA)

One example of this approach is the Advanced Meta-Learning Architecture[6] (AMLA) [21]. It is a data mining toolbox architecture aimed at running ML algorithms more efficiently whilst inherently supporting meta-learning by collecting all meta-data from every component in the system and allowing the user to query or manipulate it.

Architecture. An overview of the system is provided in Figure 13. It encapsulates all standard KD operations with modules which can then be used as building blocks for KD processes. Each module has a set of inputs and outputs, each of which is a 'model', i.e. an actual model (e.g. a decision tree) or a dataset. It also has a special 'configuration' input which supplies parameter settings, and a special 'results' output, which produces meta-data about the process. For instance, in the case the process encapsulates a decision tree learner, the input would be training data, the configuration would state its parameters, the output would provide a decision tree model and the results would state, for instance, the number of nodes in the tree. All results are represented as name-value pairs and stored in a repository, which collects all results generated by all modules. The modules are very fine grained. For instance, there are separate modules for testing a model against a dataset (which export the performance evaluations to the repository) and for sub-components of certain techniques, such as base-learners in ensembles, kernels in SVMs and so on. Modules often used together can be combined in 'schemes', which again have their own inputs, outputs, configurations and results. Schemes can also contain unspecified modules, to be filled in when used, in which case they are called templates. There also exist 'repeater' modules which repeat certain schemes many times, for instance for cross-validation. This compositionality allows to build arbitrarily complex KD workflows, quicker implementation of variants of existing algorithms, and a more efficient execution, as modules used many times only have to be loaded once. The result repository can be queried by writing short scripts to extract certain values, or to combine or filter the results of previous queries. Finally, 'commentators' can be written to perform frequently used queries, e.g. statistical significance tests, and store their results in the repository.

Meta-knowledge. The stored meta-data consists of a large variety of name-value pairs collected from (and linked to) all previously used modules, templates and schemes. They can be of any type.

[6] This is again not the officially coined name of the system.

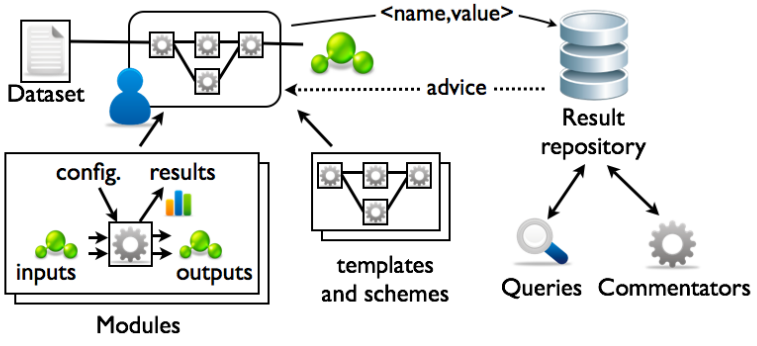


Fig. 13. The architecture of AMLA. Derived from Grabczewski and Jankowski[21].

Meta-learning. Meta-learning is done manually by programming queries. The query's constraints can involve the modules having generated the meta-data or the type of properties (the name in the name-value pairs). As such, previously obtained meta-data can be extracted and recombined to generate new meta-knowledge. Secondly, templates with missing modules can also be completed by looking up which were the most successful completed templates in similar problems.

Discussion. This is indeed a very fundamental approach to meta-learning, in the sense that it keeps track of all the meta-data generated during the design and execution of KD processes. On the other hand, each query has to be written as a small program that handles the name-value pairs, which might make it a bit harder to use, and the system is still very much under construction. For instance, at this stage of development, it is not entirely clear how the results obtained from different workflows can be compared against each other. It seems that many small queries are needed to answer such questions, and that a more structured repository might be required instead.

6.2 Experiment Databases

Experiment databases (ExpDBs) [6,65,67,66,64] provide another, although much broader, user-driven platform for the exploitation of meta-knowledge. They aim to collect the thousands of machine learning experiments that are executed every day by researchers and practitioners, and to organize them in a central repository to offer direct insight into the performance of various, state-of-the-art techniques under many different conditions. It offers an XML-based language to describe those experiments, dubbed ExpML, based on an ontology for data mining experimentation, called Exposé, and offers interfaces for DM toolboxes to automatically upload new experiments or download previous ones. The stored meta-data

can be queried extensively using SQL queries, or mined to build models of algorithm performance providing insight into algorithm behavior or to predict their performance on certain datasets. While most of the previously discussed systems are designed to be predictive, ExpDBs are mainly designed for *declarative* meta-learning, offering insights into *why* algorithms work or fail on certain datasets, although it can easily be used for predictive goals as well.

Architecture. A high-level view of the system is shown in Fig. 14. The five boxed components include the three components used to share and organize the meta-data: an ontology of domain concepts involved in running data mining experiments, a formal experiment description language (ExpML) and an experiment database to store and organize all experiments (ExpDB). In addition, two interfaces are defined: an application programming interface (API) to automatically import experiments from data mining software tools, and a query interface to browse the results of all stored experiments.

Interface. First, to facilitate the automatic exchange of data mining experiments, an application programming interface (API) is provided that builds uniform, manipulable experiment instances out of all necessary details and exports them as ExpML descriptions. The top of Fig. 14 shows some of the input details: properties (indicated by tag symbols) of the involved components, from download urls to theoretical properties, as well as the results of the experiments: the models built and their evaluations. Experiments are stored in full detail, so that they can be reproduced at any time (at least if the dataset itself is publicly available).

Software agents such as data mining workbenches (shown on the right in Fig. 14) or custom algorithm implementations can simply call methods from the API to create new experiment instances, add the used algorithms, parameters, and all other details as well as the results, and then stream the completed experiments to online ExpDBs to be stored. A multi-tier approach can also be used: a personal database can collect preliminary experiments, after which a subset can be forwarded to lab-wide or community-wide databases.

ExpML. The XML-based ExpML markup language aims to be an extensible, general language for data mining experiments, complementary to PMML[7], which allows to exchange predictive models, but not detailed experimental setups nor evaluations. Based on the Exposé ontology, the language defines various kinds of experiments, such as 'singular learner evaluations', which apply a learning algorithm with fixed parameter settings on a static dataset, and evaluate it using a specific performance estimation method (e.g. 10-fold cross validation) and a range of evaluation metrics (e.g. predictive accuracy). Experiments are described as workflows, with datasets as inputs and evaluations and/or models as outputs, and can contain sub-workflows of preprocessing techniques. Algorithms are handled as composite objects with parameters and components such as kernels, distance functions or base-learners. ExpML also differentiates between general

[7] See `http://www.dmg.org/pmml-v3-2.html`

algorithms (e.g. 'decision trees'), versioned implementations (e.g. weka.J48) and applications (weka.J48 with fixed parameters). Finally, the context of sets of experiments can also be added, including conclusions, experimental designs, and the papers in which they are used so they can be easily looked up afterwards. Further details can be found in Vanschoren et al. [66], and on the ExpDB website.

Exposé. The Exposé ontology provides a formal domain model that can be adapted and extended on a conceptual level, thus fostering collaboration between many researchers. Moreover, any conceptual extensions to the domain model can be translated consistently into updated or new ExpML definitions and database models, thus keeping them up to date with recent developments.

Exposé is built using concepts from several other data mining ontologies. First, OntoDM [49] is a general ontology for data mining which tries to relate various data mining subfields. It provides the top-level classes for Exposé, which also facilitates the extension of Exposé to other subfields covered by OntoDM. Second, EXPO [58] models scientific experiments in general, and provides the top-level classes for the parts involving experimental designs and setups. Finally, DMOP [26] models the internal structure of learning algorithms, providing detailed concepts for general algorithm definitions. Exposé unites these three ontologies and adds many more concepts regarding specific types of experiments, evaluation techniques, evaluation metrics, learning algorithms and their specific configurations in experiments.

ExpDB. The ExpDB database model, also based on Exposé, is very fine-grained, so that queries can be written about any aspect of the experimental setup, evaluation results, or properties of involved components (e.g. dataset size). When submitted to an ExpDB, the experiments are automatically stored in a well-organized way, associating them with all other stored experiments and available meta-level descriptions, thus linking empirical data with all known theoretical properties of all involved components.

All further details, including detailed design guidelines, database models, ontologies and XML definitions, can be found on the ExpDB website, which can be found at `http://expdb.cs.kuleuven.be`.

It also hosts a working implementation in MySQL which can be queried online through two query interfaces: an online interface on the homepage itself and an open-source desktop application. Both allow to launch queries written in SQL, or composed in a graphical query interface, and can show the results in tables or graphical plots. The database has been extended several times based on user requests, and is frequently being visited by over 400 users.

Meta-knowledge. The ExpML language is designed not only to upload new experiments, but also to add new definitions of any new component, such as a new algorithm or a new data characteristic. As such, the meta-knowledge contained in the database can be dynamically extended by the user to be up to date with new developments. Algorithms, datasets, preprocessing algorithms and evaluation methods are first of all described with all necessary details (e.g. version

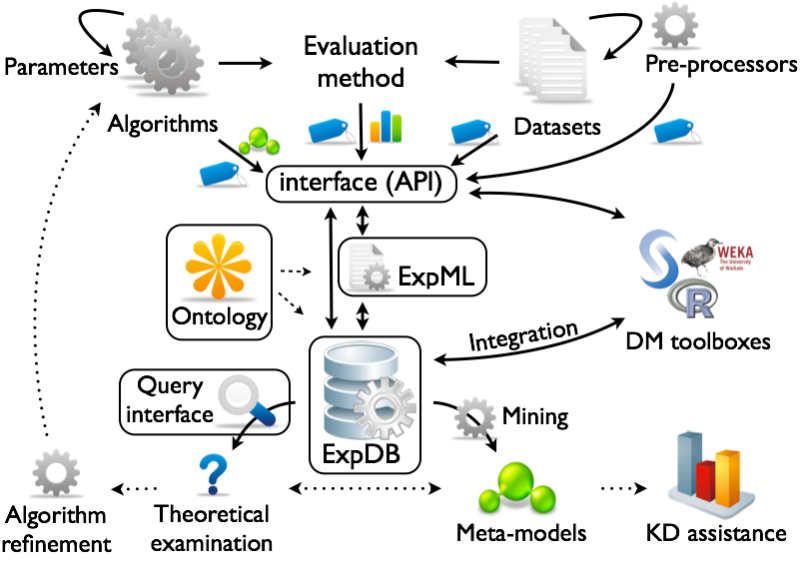


Fig. 14. The architecture of experiment databases.

numbers) to allow unique identification, as well as more informational descriptions of their use. Furthermore, they can be described with an arbitrary number of characterizations, and new ones can be added at any time, preferably with a description of how they are computed, and immediately be used to see whether they improve meta-learning predictions. Attribute-specific data characteristics can be stored as well. Parameter settings are also stored, and each parameter can be described with additional meta-data such as default values, suggested ranges and informal descriptions of how to tune them. Furthermore, an arbitrary number of performance evaluations can be added, including class-specific metrics such as AUROC, and entire contingency matrices. In principle, the produced models could also be stored, e.g. using the PMML format, although in the current version of the system only the predictions for all instances are stored[8]. At the time of writing, the database contained over 650,000 experiments on 67 classification and regression algorithms from WEKA [23], 149 different datasets from UCI [2], 2 data preprocessing techniques (feature selection and sampling), 40 data characteristics [29] and 10 algorithm characterizations [25]. In its current implementation, it covers mostly classification, and some regression tasks, naturally extending to more complex settings, such as kernel methods, ensembles or multi-target learning.

Meta-learning. The bottom half of Figure 14 shows the different ways the stored meta-data can be used. First of all, any hypothesis about learning behavior

[8] This allows computation of new evaluation metrics without rerunning experiments.

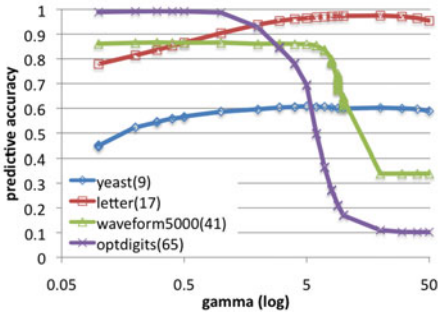


Fig. 15. The effect of parameter gamma of the RBF-kernel in SVMs.

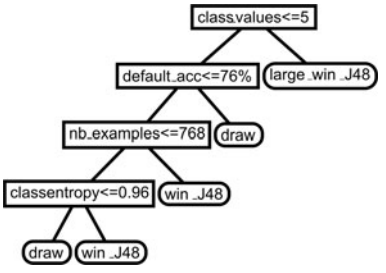


Fig. 16. A meta-decision tree predicting J48's superiority over OneR.

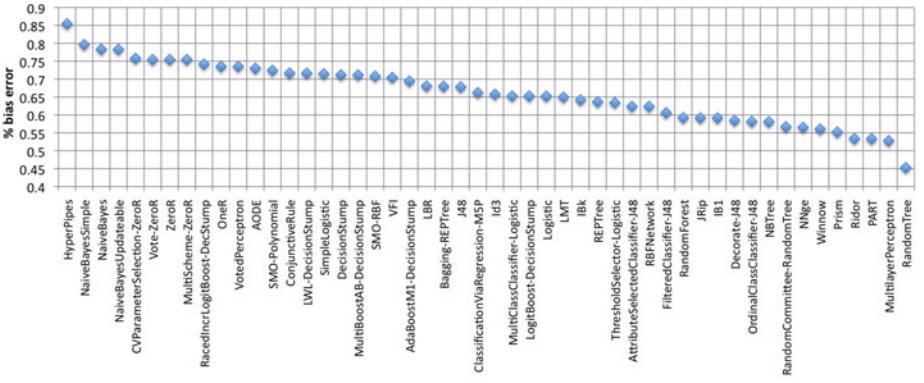


Fig. 17. The average percentage of bias-related error for each algorithm.

can be checked by translating it to an SQL query and interpreting the returned results. Adding and dropping constraints from the query provides an easy and effective means of thoroughly browsing the available meta-data. For instance, we could ask for the 'predictive accuracy' of all support vector machine 'SVM' implementations, and for the value of the 'gamma' parameter (kernel width) of the 'RBF' kernel, on the original (non-preprocessed) version of dataset 'letter'. The result, showing the effect of the gamma parameter, is shown in Fig. 15 (the curve with squares). Adding three more datasets to the query generates the other curves. Just as easily, we could have asked for the size of all datasets instead, or we could have selected a different parameter or any other kind of alteration to explore the meta-data further. SQL queries also allow for the meta-data to be rearranged or aggregated to answer very general questions directly, such as algorithm comparisons, rankings, tracking the effects of preprocessing techniques (e.g. learning curves), or building profiles of algorithms based on many experiments [6,67]. An example of the latter is shown in Fig. 17: since a large number of bias-variance decomposition experiments are stored, we can write a query that

reorganizes all the data and calculates the average percentage of bias error (as opposed to variance error) produced by each algorithm. Such algorithm profiling can be very useful for understanding the performance of learning algorithms.

As shown in Fig. 14, querying is but one of the possible meta-learning applications. First of all, insights into the behavior of algorithms could lead to algorithm refinement, thus closing the algorithm design loop. As a matter of fact, the results in Fig. 15 suggested a correlation between the useful range of the gamma parameter and the number of attributes (shown in parentheses), which led to a refinement of WEKA's SVM implementation [67]. Furthermore, instead of browsing the meta-data, we could also construct meta-models by downloading parts of the meta-data and modeling it. For instance, we could build decision trees predicting when one algorithm outperforms another, as shown in Fig. 16 for algorithms J48 and OneR, or how different parameters interact on certain datasets [63]. Finally, the meta-data can also be used to provide the necessary meta-data for other DM assistance tools, as shall be discussed in Section 7.

Discussion. For researchers developing new algorithms, or practitioners trying to apply them, there are many benefits to sharing experiments. First, they make results reproducible and therefore verifiable. Furthermore, they serve as an ultimate reference, a 'map' of all known approaches, their properties, and results on how well they fared on previous problems. And last but not least, it allows previous experiments to be readily reused, which is especially useful when benchmarking new algorithms on commonly used datasets, but also makes larger, more generalizable studies much easier to perform. As such there is a real incentive to share experiments for further reuse, especially since integration in existing DM toolboxes is relatively easy. Such integration can also be used to automatically generate new experimental runs (using, for instance, active learning principles) based on a query or simply on a lack of experiments on certain techniques. There are also numerous network effects in which advancements in learning algorithms, pre-processing techniques, meta-learning and other subfields all benefit from each other. For instance, results on a new preprocessing technique may be used directly in designing KD workflows, new data characteristics can highlight strengths and weaknesses of learning algorithms, and the large amounts of experiments can lead to better meta-learning predictions. Possible drawbacks are that, unless integrated in a toolbox, the system cannot execute algorithms or KD workflows on demand (although code or urls to executables are always stored) and that, while results received from toolboxes can be assumed to be accurate, results from individual users may have to be verified by referees.

Compared to DMA, one could say that DMA is a local, predictive system, while an ExpDB is a community-based, descriptive system. DMA also employed a database, but it was much simpler, consisting of a few large tables with many columns describing all different kinds of data characterizations. In ExpDBs, the database is necessarily very fine-grained to allow very flexible querying and to scale to millions of experiments from many different contributors. The main difference though is that, while DMA is a system designed for practical algorithm selection advice, ExpDBs offer a platform for any kind of meta-learning study

and for the development of future meta-learning systems. It is also quite complementary to MiningMart. While ExpDBs do store preprocessing workflows, MiningMart is much more developed in that area. Finally, compared to AMLA, the repository in ExpDBs is much more structured, allowing much more advanced queries in standard SQL. On the other hand, while ExpDBs can be integrated in any toolbox, AMLA offers a more controlled approach which gathers meta-data from every component in a KD workflow.

7 A New Platform for Intelligent KD Support

7.1 Summary

An overview of the previously discussed architectures is shown in Table 1. The columns represent consecutively the portion of the KD process covered, the system type, how it interacts with the user, what type of meta-information is stored, the data it has been trained on, which KD processes it considers, which evaluation metrics are covered, which meta-features are stored and which meta-learning techniques are used to induce new information, make predictions or otherwise advise the user.

As the table shows, and the systems' discussions have indicated, each system has its own strengths and weaknesses, and cover the KD process to various extents. Some algorithms, like MiningMart and DMA provide a lot of support and gather a lot of meta-information about a few, or only one KD step, while others try to cover a larger part of the entire process, but consider a smaller number of techniques or describe them with less information, usually provided by experts. All systems also expect very different things from their users. Some, especially CITRUS, GDL and AMLA, put the user (assumed to be an expert) firmly in the driver's seat, leaving every important decision to her. Others, like Consultant-2, GLS, MiningMart, HDMA and NExT allow the user to interfere in the workflow creation process, often explicitly asking for input. Finally, the meta-model systems and IDEA almost completely automate this process, offering suggestions to users which they may adopt or ignore. Note that, with the exception of Consultant-2, none of the systems performing algorithm selection also predict appropriate parameters, unless they are part of a prior workflow.

A few systems obviously learned from each other. For instance, DMA, NOEMON and ALT learned from StatLog, NExT learned from IDEA and HDMA learned from prior CBR and expert system approaches. Still, most systems are radically different from each other, and there is no strong sense of convergence to a general platform for KD workflow generation.

7.2 Desiderata

We now look forward, striving to combine the best aspects of all prior systems:

Extensibility: Every KD support system that only covers a limited number of techniques will at some point become obsolete. It is therefore important that new KD techniques can be added very easily. (GLS, AMLA, ExpDBs)

Table 1. Comparison of meta-learning architectures

	KD step[a]	type	user interface	type meta-info	data scope	KD process scope	evaluation scope	meta-feature scope	meta-learning technique
Consultant-2	■□□□	Expert system	Q&A sessions	heuristic expert rules	NA	10 algorithms from MLT	speed, model properties	simple statistics, prior knowledge	static meta-model
DMA	■□■□	meta-model	Webinterface: algo ranking	collection of experiments	67 datasets + 83 from users	10 classific. algorithms	accuracy and speed	7 modified StatLog features	kNN and ranking
NOEMON	■□■□	meta-model	Algorithm ranking	collection of decision trees	77 datasets (UCI)	3 classific. algorithms	accur., speed and memory	idem StatLog + histograms	Vote over models and ranking
IDEA	□■■■	Planning	Ranking of KD plans	ontology of KD operators with heuristics	NA	10 preproc., 6 ML algo's, 5 postproc.	accuracy, speed, model-properties	IOPE[b] + basic data properties	AI planning
GLS	□■■■	Planning	User reviews partial plans	ontology of KD operators + agent meta-rules	NA	7 preproc., 7 ML algo's, 2 postproc.	NA	NA	emergent agent behavior + AI planning
GDL	□■■■	Planning	May call on experts	LML descriptions	Knowledge bases	NA	NA	12 LML properties	Search
CCE	□■■■	Planning	Stop criterion	'reliability' values	NA	extensible	complexity	NA	Adaptive system
CITRUS	□■■■	CBR	Clementine editor	case base + process constraints	NA	Clementine processes	depends on operator	workflow description	CBR, checking constraints
ALT	□□■□	CBR	Returns 'best' algorithm	collection of cases (experiments)	80 datasets (UCI+more)	21 classific. algorithms	accuracy and speed	StatLog features + 4 algorithm feats	Case-based reasoning
MiningMart	■□□□	CBR	Editors: create or adapt cases	collection of cases + business ontology	NA (any database)	41 preprocessors	NA	business description of each workflow	none (manual CBR)
HDMA	□■■■	CBR	Step-by-step, show cases + advice	set of KD cases + set of heuristic expert rules, ontological	NA	selection of WEKA algorithms	user ratings	30 data features, 31 solution features, 5 user ratings	Case-based reasoning
NExT	□■■■	CBR + Planning	Interaction to solve workflow issues	ontology of workflow issues and solutions	NA	NA	NA	IOPE[b] + workflow description	CBR + AI planning
AMLA	□■■■	Querying	Process editor + Query lang.	Name-value pairs from any component	NA	extensible	extensible	extensible	querying + fill in templates
ExpDBs	□■■■	Querying	Query interf. + meta-data up/download	searchable repository of experim's, algo's, data,... + ontology	extensible (150+ UCI datasets)	extensible (70+ WEKA algorithms)	extensible (50+ metrics)	extensible (50+ data and algorithm features)	querying and mining (any algorithm)

[a] The boxes stand consecutively for the Data Selection, Preprocessing+transformation, Data Mining, and Postprocessing step.
[b] IOPE=Inputs, Outputs, Preconditions and Effects of KD operators.

Integration: Ideally, the system should be able to execute the workflow, instead of just offering an abstract description. Keeping extensibility in mind, it should also be able to call on some existing KD tools to execute processes, instead of reimplementing every KD process in a new environment. Indeed, as new types of algorithms are created, new data preprocessing methods are developed, new learning tasks come about, and even new evaluation procedures are introduced, it will be infeasible to re-implement this continuously expanding stream of learning approaches, or to run all the experiments necessary to learn from them. (Consultant-2, HDMA, ExpDBs)

Self-maintenance: Systems should be able to update their own meta-knowledge as new data or new techniques become available. While experts are very useful to enrich the meta-knowledge with successful models and workflows, they cannot be expected to offer all the detailed meta-data needed to learn from previous KD episodes. (DMA, NOEMON, ALT, AMLA, ExpDBs)

Common language: Effective KD support hinges on a large body of meta-knowledge. As more and more KD techniques are introduced, it becomes infeasible to locally run all the experiments needed to collect the necessary meta-data. It is therefore crucial to take a *community-based* approach, in which descriptions of KD applications and generated meta-data can be generated by, and shared with, the entire community. To achieve this, a common language should be used to make all the stored meta-data interchangeable. This could lead to a central repository for all meta-data, or a collection of different repositories interfacing with each other. (MiningMart, ExpDBs)

Ontologies: Meta-knowledge should be stored in a way that makes it machine-interpretable, so that KD support tools can use it effectively. This is reflected by the use of ontologies in many of the discussed systems. Ideally, such an ontology should also establish a common vocabulary for the concepts and relations used in KD research, so that different KD systems can interact. This vocabulary could be the basis of the proposed common description language. (IDEA, GLS, MiningMart, HDMA, NExT, ExpDBs)

Meta-data organization: The stored meta-data should also be stored in a way that facilitates manual querying by users. Indeed, we cannot foresee all the possible uses of meta-data beforehand, and should therefore enable the user to perform her own investigations. Moreover, when extending KD support tools with new ways of using meta-data, such a structured repository will drastically facilitate this extension. (AMLA, ExpDBs)

Workflow reuse and adaptation: Since most KD applications focus on a limited number of tasks, it is very likely that there exist quite a few prior successful workflows that have been designed for that task. Any intelligent KD support system should therefore be able to return similar workflows, but also offer extensive support to adapt them to the new problem. For instance, if a prior workflow can only partially be reused, new solutions should be proposed to fill in the missing pieces. (MiningMart, HDMA, NExT)

Planning: When no similar workflows exist, or when parts of workflows need to be redesigned, using the KD processes' preconditions and effects for planning

is clearly a good approach, although care should be taken that the planning space is not prohibitively large. (IDEA,GLS,NExT)

Learning: Last but not least, the system should get better as it gains more experience. This includes planning: over time, the system should be able to optimize its planning approach, e.g. by updating heuristic descriptions of the operators used, instead of generating the same plan every time it encounters the same problem. (DMA, NOEMON, GLS, all CBR approaches)

7.3 Architecture

These aspects can be combined in the KD support platform shown in Figure 18. It is based on both our analysis here and on the description of a proposed DM laboratory in Kalousis et al. [30].

A community-based approach. This platform is set in a *community-based* environment, with many people using the same KD techniques. It could cover a very general domain, such as KD as a whole, or a more specific one, such as bio-technology, which may result in more specific types of meta-data and more specific ontologies.

Notice that first of all, a common language is used to exchange meta-data between the KD assistant and the tools with which it interacts. First, on the right, there are the many DM/KD toolboxes that implement a wide variety of KD processes. They exchange workflows, descriptions of algorithms and datasets, produced models and evaluations of the implemented techniques.

On the left, there are *web services*. In the last couple of years, there has been a strong movement away from locally implemented toolboxes and datasets, and towards KD processes and databases offered as online services. These *Service Oriented Architectures* (SOAs) [18] define standard interfaces and protocols that allow developers to encapsulate tools as services that clients can access without knowledge of, or control over, their internal workings. In the case of a database, this interface may offer to answer specific queries, e.g. a database of airplane flight may return the flight schedule for a specific plane. In the case of a learning algorithm, the interface may accept a dataset (or a database implemented as a web service) and return a decision tree.

While we did not explicitly include experts as a source of meta-knowledge, we assume that they will build workflows and models using the available web services and toolboxes.

When the KD assistant interacts with these services, it will exchange workflows, descriptions of the web services (where to find them and how to interact with them), produced models and evaluation results. Given the rise of web services, XML is a very likely candidate as the modeling language for this common language. Web services interact with each other using SOAP (Simple Object Access Protocol) messages, and describe their interface in WSDL (Web Services Description Language), both of which are described in XML. XML is also used by many KD/DM toolboxes to serialize their data and produced models.

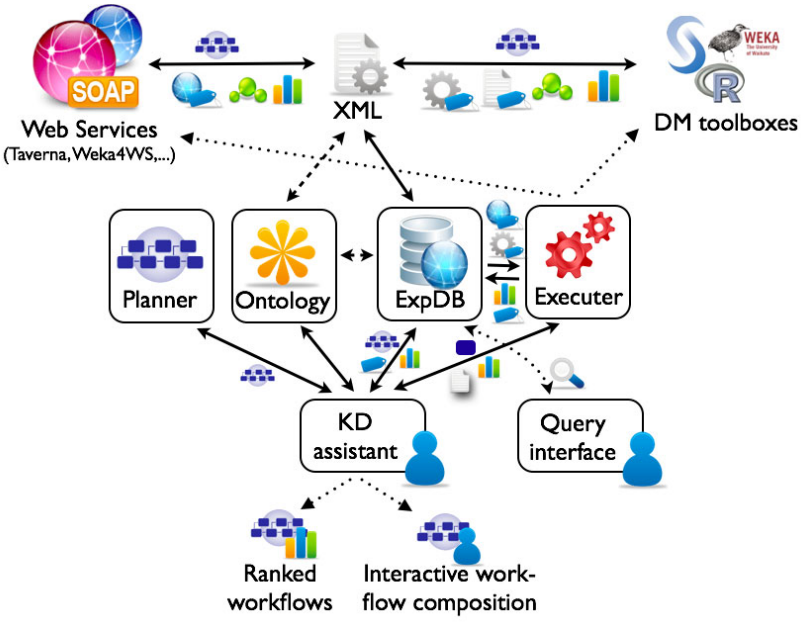


Fig. 18. A proposed platform for intelligent KD support.

A DM/KD ontology. The vocabulary used for these descriptions should be described in a common ontology. Despite many proposed ontologies, there is of yet no consensus on an ontology for DM or KD, but we hope that such an ontology will arise over the coming years. Imagine the internet without HTML, and it is obvious that a common language for KD is essential for progress in this field. The ontology should also provide detailed descriptions of KD processes, such as their place in the hierarchy of different processes, the internal structure of composite learning algorithms, preconditions and effects of all known KD operators, and the parameters that need to be tuned in order to use them. Additional information can be added to extend the ontology to engender further KD support (such as the list of KD issues and solutions in NExT).

A meta-data repository. All generated meta-data is automatically stored and organized in a central repository, such as an ExpDB. It collects all the details of all performed experiments, including the workflows used and the obtained results, thus offering a complete log of the experimentation which can be used to reproduce the submitted studies. Moreover, using the ontology, it automatically organizes all the data so it can be easily queried by expert users, allowing it to answer almost any kind of question about the properties and the performance of the used KD techniques, using the meta-data from many submitted studies. It serves as the long-term memory of the KD assistant, but also as that of the individual researcher and the community as a whole. It will be frequently polled

by the KD assistant to extract the necessary meta-data, and should contain a query interface for manual investigations as well. The database itself can be wrapped as a web service, allowing automatic interaction with other tools and web services.

Planning and execution. The KD assistant interacts with two more components: an AI planner used to solve any planning problems, and an executer component which runs the actually implemented KD algorithms in KD/DM toolboxes or web services to execute workflows, or perhaps to do other calculations such as computing meta-features if they are not implemented in the KD assistant itself. The executer polls the ExpDB to obtain the necessary descriptions and locations of the featured web services, algorithms or datasets.

As for the output generated by the KD assistant, we foresee two important types of advice (beyond manual querying), although surely many more kinds of advice are possible. The first is a ranked list of workflows produced by the KD assistant (even if this workflow only consists of a single learning algorithm). The second, possibly more useful approach is a semi-automatic interactive process in which the user actively participates during the creation of useful workflows. In this case, the KD assistant can be associated with a workflow editing tool (such as Taverna), and assist the users as they compose workflows, e.g. by checking the correctness of a workflow, by completing partial workflows using the planner, or by retrieving, adapting or repairing previously entered workflows.

7.4 Implementation

ExpDBs and ontologies. While such a KD support system may still be some way off, recently, a great deal of work has been done that brings us a lot closer to realizing it.

First of all, repositories organizing all the generated meta-data can be built using the experiment databases discussed in Section 6.2. Its ontology and XML-based language for exchanging KD experiments can also be a good starting point. However, building such ontologies and languages should be a collaborative process, so we should also build upon some other recently proposed ontologies in DM, such as OntoDM [48,49], DMOP [30,26], eProPlan [35], KDDONTO [24] and KD ontology [69,70].

Planning. Concerning planning, several approaches have been outlined that translate the ontological descriptions of KD operators to a planning description based on the standard Planning Domain Description Language (PDDL) by using an *ontological reasoner* to query their KD ontologies before starting the actual planning process [36,42,56]. Other approaches integrate a reasoning engine directly in the planner, so that the planner can directly query the ontology when needed [35,69,70]. For instance, eProPlan[35] covers the preconditions and effects of all KD operators, expressed as rules in the Semantic Web Rule Language (SWRL) [28].

Klusch et al. [36] and Liu et al. [42] use a classical STRIPS planner to produce the planning, while Sirin et al. [56] and Kietz et al. [35] propose an Hierarchical Task Network (HTN) planning approach [55], in which each *task* has a set of associated *methods*, which decompose into a sequence of (sub)tasks and/or *operators* that, when executed in that order, achieve the given task. The main task is then recursively decomposed until we obtain a sequence of applicable operators. Somewhat similar to GLS, this divide-and-conquer approach seems an effective way to reduce the planning space.

Zakova et al. [70] uses an adaptation of the heuristic Fast-Forward (FF) planning system [27]. Moreover, it allows the completed workflows to be executed on the Orange DM platform, and vice-versa: workflows composed in Orange are automatically annotated with their KD ontology so that they can be used for ontology querying and reasoning. It does this by mapping their ontology to the ontology describing the Orange operators.

Finally, Kalousis et al. [30] propose a system that will combine planning and meta-learning. It contains a kernel-based, probabilistic meta-learner which dynamically adjusts transition probabilities between DM operators, conditioned on the current application task and data, user-specified performance criteria, quality scores of workflows applied in the past to similar tasks and data, and the users profile (based on quantified results from, and qualitative feedback on, her past DM experiments). Thus, as more workflows are stored as meta-knowledge, and more is known about the users building those workflows, it will learn to build workflows better fit to the user.

Web services. The development of service oriented architectures for KD, also called *third-generation DM/KD*, has also gathered steam, helped by increasing support for building workflows of web services.

Taverna [53], for instance, is a system designed to help scientists compose executable workflows in which the components are web services, especially for biological in-silico experiments. Similarly, Triana [60] supports workflow execution in multiple environments, such as peer-to-peer (P2P) and the Grid. Discovery Net [54] and ADMIRE [38] are platforms that make it easier for algorithm designers to develop their algorithms as web services and Weka4WS [59] is a framework offering the algorithms in the WEKA toolkit as web services.

Finally, Orange4WS [51] is a *service-oriented KD* platform based on the Orange toolkit. It wraps existing web services as Orange workflow components, thus allowing to represent them, together with Orange's original components as graphical 'widgets' for manual workflow composition and execution. It also proposes a toolkit to wrap 'local' algorithms as web services.

8 Conclusions

In this chapter, we have provided a survey of the different solutions proposed to support the design of KD processes through the use of meta-knowledge and

highlighted their strengths and weaknesses. We observed that most of these systems are very different, and were seemingly developed independently from each other, without really capitalizing on the benefits of prior systems. Learning from these prior architectures, we proposed a new, community-based platform for KD support that combines their best features, and showed that recent developments have brought us close to realizing it.

Acknowledgements

This research is supported by GOA 2003/08 "Inductive Knowledge Bases" and F.W.O.-Vlaanderen G.0108.06 "Foundations of Inductive Databases for Data Mining".

References

1. Aha, D.: Generalizing from case studies: A case study. In: Proceedings of the Ninth International Conference on Machine Learning, pp. 1–10 (1992)
2. Asuncion, A., Newman, D.: Uci machine learning repository. University of California, School of Information and Computer Science (2007)
3. Bensusan, H., Giraud-Carrier, C.: Discovering task neighbourhoods through landmark learning performances. In: Zighed, D.A., Komorowski, J., Żytkow, J.M. (eds.) PKDD 2000. LNCS (LNAI), vol. 1910, pp. 325–330. Springer, Heidelberg (2000)
4. Bernstein, A., Dänzer, M.: The nExT system: Towards true dynamic adaptations of semantic web service compositions. In: Franconi, E., Kifer, M., May, W. (eds.) ESWC 2007. LNCS, vol. 4519, pp. 739–748. Springer, Heidelberg (2007)
5. Bernstein, A., Provost, F., Hill, S.: Toward intelligent assistance for a data mining process: an ontology-based approach for cost-sensitive classification. IEEE Transactions on Knowledge and Data Engineering 17(4), 503–518 (2005)
6. Blockeel, H., Vanschoren, J.: Experiment databases: Towards an improved experimental methodology in machine learning. In: Kok, J.N., Koronacki, J., Lopez de Mantaras, R., Matwin, S., Mladenič, D., Skowron, A. (eds.) PKDD 2007. LNCS (LNAI), vol. 4702, pp. 6–17. Springer, Heidelberg (2007)
7. Brazdil, P., Giraud-Carrier, C., Soares, C., Vilalta, R.: Metalearning: Applications to data mining. Springer, Heidelberg (2009)
8. Brazdil, P., Soares, C., Costa, J.P.D.: Ranking learning algorithms: Using ibl and meta-learning on accuracy and time results. Machine Learning 50, 251–277 (2003)
9. Chandrasekaran, B., Josephson, J.: What are ontologies, and why do we need them? IEEE Intelligent systems 14(1), 20–26 (1999)
10. Chapman, P., Clinton, J., Kerber, R., Khabaza, T., Reinartz, T., Shearer, C., Wirth, R.: Crisp-dm 1.0. a step-by-step data mining guide (1999), `http://www.crisp-dm.org`
11. Charest, M., Delisle, S.: Ontology-guided intelligent data mining assistance: Combining declarative and .... In: Proceedings of the 10th IASTED International Conference on Artificial Intelligence and Soft Computing, pp. 9–14 (2006)
12. Charest, M., Delisle, S., Cervantes, O., Shen, Y.: Intelligent data mining assistance via cbr and ontologies. In: Proceedings of the 17th International Conference on Database and Expert Systems Applications (DEXA 2006) (2006)

13. Charest, M., Delisle, S., Cervantes, O., Shen, Y.: Bridging the gap between data mining and decision support: A case-based reasoning and .... Intelligent Data Analysis 12, 1–26 (2008)
14. Craw, S., Sleeman, D., Graner, N., Rissakis, M.: Consultant: Providing advice for the machine learning toolbox. In: Research and Development in Expert Systems IX: Proceedings of Expert Systems 1992, pp. 5–23 (1992)
15. Dean, M., Connolly, D., van Harmelen, F., Hendler, J., Horrocks, I., Orah, L., McGuinness, D., Patel-Schneider, P.F., Stein, L.A.: Web ontology language (owl) reference version 1.0. W3C Working Draft (2003), http://www.w3.org/TR/2003/WD-owl-ref-20030331
16. Engels, R.: Planning tasks for knowledge discovery in databases; performing task-oriented user-guidance. In: Proceedings of the 2nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD 1996), pp. 170–175 (1996)
17. Fikes, R., Nilsson, N.: Strips: A new approach to the application of theorem proving to problem solving. Artificial intelligence 2, 189–208 (1971)
18. Foster, I.: Service-oriented science. science 308(5723), 814 (2005)
19. Giraud-Carrier, C.: The data mining advisor: meta-learning at the service of practitioners. In: Proceedings of the 4th International Conference on Machine Learning and Applications, pp. 113–119 (2005)
20. Giraud-Carrier, C.: Metalearning-a tutorial. Tutorial at the 2008 International Conference on Machine Learning and Applications, ICMLA 2008 (2008)
21. Grabczewski, K., Jankowski, N.: Versatile and efficient meta-learning architecture: Knowledge representation and .... In: IEEE Symposium on Computational Intelligence and Data Mining, pp. 51–58 (2007)
22. Grąbczewski, K., Jankowski, N.: Meta-learning with machine generators and complexity controlled exploration. In: Rutkowski, L., Tadeusiewicz, R., Zadeh, L.A., Zurada, J.M. (eds.) ICAISC 2008. LNCS (LNAI), vol. 5097, pp. 545–555. Springer, Heidelberg (2008)
23. Hall, M., Frank, E., Holmes, G., Pfahringer, B., Reutemann, P., Witten, I.: The weka data mining software: An update. SIGKDD Explorations 11(1), 10–18 (2009)
24. Hidalgo, M., Menasalvas, E., Eibe, S.: Definition of a metadata schema for describing data preparation tasks. In: Proceedings of the ECML/PKDD 2009 Workshop on 3rd generation Data Mining (SoKD 2009), pp. 64–75 (2009)
25. Hilario, M., Kalousis, A.: Building algorithm profiles for prior model selection in knowledge discovery systems. Engineering Intelligent Systems 8(2) (2000)
26. Hilario, M., Kalousis, A., Nguyen, P., Woznica, A.: A data mining ontology for algorithm selection and meta-mining. In: Proceedings of the ECML/PKDD 2009 Workshop on 3rd generation Data Mining (SoKD 2009), pp. 76–87 (2009)
27. Homann, J., Nebel, B.: The ff planning system: Fast plan generation through heuristic search. Journal of Artificial Intelligence Research 14, 253–302 (2001)
28. Horrocks, I., Patel-Schneider, P., Boley, H.: Swrl: A semantic web rule language combining owl and ruleml. W3C Member submission (2004), http://www.w3.org/Submissions/SWRL/
29. Kalousis, A.: Algorithm selection via meta-learning. PhD Thesis. University of Geneve (2002)
30. Kalousis, A., Bernstein, A., Hilario, M.: Meta-learning with kernels and similarity functions for planning of data mining workflows. In: ICML/COLT/UAI 2008 Planning to Learn Workshop (PlanLearn), pp. 23–28 (2008)
31. Kalousis, A., Hilario, M.: Model selection via meta-learning: a comparative study. International Journal on Artificial Intelligence Tools 10(4), 525–554 (2001)

32. Kalousis, A., Theoharis, T.: Noemon: Design, implementation and performance results of an intelligent assistant for classifier selection. Intelligent Data Analysis 3(4), 319–337 (1999)
33. Kaufman, K.: Inlen: a methodology and integrated system for knowledge discovery in databases. PhD Thesis, School of Information Technology and Engineering, George Mason University (1997)
34. Kaufman, K., Michalski, R.: Discovery planning: Multistrategy learning in data mining. In: Proceedings of the Fourth International Workshop on Multistrategy Learning, pp. 14–20 (1998)
35. Kietz, J., Serban, F., Bernstein, A., Fischer, S.: Towards cooperative planning of data mining workflows. In: Proceedings of the Third Generation Data Mining Workshop at the 2009 European Conference on Machine Learning (ECML 2009), pp. 1–12 (2009)
36. Klusch, M., Gerber, A., Schmidt, M.: Semantic web service composition planning with owls-xplan. In: Proceedings of the First International AAAI Fall Symposium on Agents and the Semantic Web (2005)
37. Kodratoff, Y., Sleeman, D., Uszynski, M., Causse, K., Craw, S.: Building a machine learning toolbox. In: Enhancing the Knowledge Engineering Process: Contributions from ESPRIT, pp. 81–108 (1992)
38. Le-Khac, N., Kechadi, M., Carthy, J.: Admire framework: Distributed data mining on data grid platforms. In: Proceedings of the 1st International Conference on Software and Data Technologies, vol. 2, pp. 67–72 (2006)
39. Levin, L.: Universal sequential search problems. Problemy Peredachi Informatsii 9(3), 115–116 (1973)
40. Li, M., Vitányi, P.: An introduction to kolmogorov complexity and its applications. In: Text and Monographs in Computer Science. Springer, Heidelberg (1993)
41. Lindner, G., Studer, R.: Ast: Support for algorithm selection with a cbr approach. In: Żytkow, J.M., Rauch, J. (eds.) PKDD 1999. LNCS (LNAI), vol. 1704, pp. 418–423. Springer, Heidelberg (1999)
42. Liu, Z., Ranganathan, A., Riabov, A.: A planning approach for message-oriented semantic web service composition. In: Proceedings of the National Conference on AI, vol. 5(2), pp. 1389–1394 (2007)
43. METAL. Metal: A meta-learning assistant for providing user support in machine learning and data mining. ESPRIT Framework IV LRT Reactive Project Nr. 26.357 (2001)
44. Michalski, R., Kerschberg, L., Kaufman, K.: Mining for knowledge in databases: The inlen architecture, initial implementation and first results. Journal of Intelligent Information Systems 1(1), 85–113 (1992)
45. Michie, D., Spiegelhalter, D., Taylor, C.: Machine learning. In: Neural and Statistical Classification. Ellis Horwood (1994)
46. MLT. Machine learning toolbox. Esprit Framework II Research Project Nr. 2154 (1993)
47. Morik, K., Scholz, M.: The miningmart approach to knowledge discovery in databases. Intelligent Technologies for Information Analysis, pp. 47–65 (2004)
48. Panov, P., Dzeroski, S., Soldatova, L.: Ontodm: An ontology of data mining. In: Proceedings of the 2008 IEEE International Conference on Data MIning Workshops, pp. 752–760 (2008)
49. Panov, P., Soldatova, L.N., Džeroski, S.: Towards an ontology of data mining investigations. In: Gama, J., Costa, V.S., Jorge, A.M., Brazdil, P.B. (eds.) DS 2009. LNCS, vol. 5808, pp. 257–271. Springer, Heidelberg (2009)

50. Pfahringer, B., Bensusan, H., Giraud-Carrier, C.: Meta-learning by landmarking various learning algorithms. In: Proceedings of the Seventeenth International Conference on Machine Learning, pp. 743–750 (2000)
51. Podpecan, V., Jursic, M., Zakova, M., Lavrac, N.: Towards a service-oriented knowledge discovery platform. In: Proceedings of the SoKD 2009 International Workshop on Third Generation Data Mining at ECML PKDD 2009, pp. 25–38 (2009)
52. Rendell, L., Seshu, R., Tcheng, D.: Layered concept learning and dynamically-variable bias management. In: Proceedings of the 10th International Joint Conference on Artificial Intelligence, pp. 308–314 (1987)
53. De Roure, D., Goble, C., Stevens, R.: The design and realisation of the myexperiment virtual research environment for social sharing of workflows. Future Generation Computer Systems 25, 561–567 (2009)
54. Rowe, A., Kalaitzopoulos, D., Osmond, M.: The discovery net system for high throughput bioinformatics. Bioinformatics 19, 225–231 (2003)
55. Sacerdoti, E.: Planning in a hierarchy of abstraction spaces. Artificial intelligence 5(2), 115–135 (1974)
56. Sirin, E., Parsia, B., Wu, D., Hendler, J., Nau, D.: Htn planning for web service composition using shop2. Journal of Web Semantics 1(4), 377–396 (2004)
57. Sleeman, D., Rissakis, M., Craw, S., Graner, N., Sharma, S.: Consultant-2: Pre-and post-processing of machine learning applications. International Journal of Human-Computer Studies 43(1), 43–63 (1995)
58. Soldatova, L., King, R.: An ontology of scientific experiments. Journal of the Royal Society Interface 3(11), 795–803 (2006)
59. Talia, D., Trunfio, P., Verta, O.: Weka4ws: a wsrf-enabled weka toolkit for distributed data mining on grids. In: Jorge, A.M., Torgo, L., Brazdil, P.B., Camacho, R., Gama, J. (eds.) PKDD 2005. LNCS (LNAI), vol. 3721, pp. 309–320. Springer, Heidelberg (2005)
60. Taylor, I., Shields, M., Wang, I., Harrison, A.: The triana workflow environment: Architecture and applications. In: Workflows for e-Science, pp. 320–339. Springer, Heidelberg (2007)
61. Utgoff, P.: Shift of bias for inductive concept learning. In: Machine learning: An artificial intelligence approach, vol. II. Morgan Kaufmann, San Francisco (1986)
62. Van Someren, M.: Towards automating goal-driven learning. In: Proceedings of the Planning to Learn Workshop at the 18th European Conference of Machine Learning (ECML 2007), pp. 42–52 (2007)
63. Vanschoren, J., Assche, A.V., Vens, C., Blockeel, H.: Meta-learning from experiment databases: An illustration. In: Proceedings of the 16th Annual Machine Learning Conference of Belgium and The Netherlands (Benelearn 2007), pp. 120–127 (2007)
64. Vanschoren, J., Blockeel, H.: A community-based platform for machine learning experimentation. In: Buntine, W., Grobelnik, M., Mladenić, D., Shawe-Taylor, J. (eds.) ECML PKDD 2009. Lecture Notes in Computer Science (LNAI), vol. 5782, pp. 750–754. Springer, Heidelberg (2009)
65. Vanschoren, J., Blockeel, H., Pfahringer, B.: Experiment databases: Creating a new platform for meta-learning research. In: Proceedings of the ICML/UAI/COLT Joint Planning to Learn Workshop (PlanLearn 2008), pp. 10–15 (2008)
66. Vanschoren, J., Blockeel, H., Pfahringer, B., Holmes, G.: Organizing the world's machine learning information. Communications in Computer and Information Science 17, 693–708 (2008)

67. Vanschoren, J., Pfahringer, B., Holmes, G.: Learning from the past with experiment databases. In: Ho, T.-B., Zhou, Z.-H. (eds.) PRICAI 2008. LNCS (LNAI), vol. 5351, pp. 485–496. Springer, Heidelberg (2008)
68. Wirth, R., Shearer, C., Grimmer, U., Reinartz, T., Schlosser, J., Breitner, C., Engels, R., Lindner, G.: Towards process-oriented tool support for knowledge discovery in databases. In: Komorowski, J., Żytkow, J.M. (eds.) PKDD 1997. LNCS, vol. 1263, pp. 243–253. Springer, Heidelberg (1997)
69. Zakova, M., Kremen, P., Zelezny, F., Lavrac, N.: Planning to learn with a knowledge discovery ontology. In: Second Planning to Learn Workshop at the Joint ICML/COLT/UAI Conference, pp. 29–34 (2008)
70. Záková, M., Podpecan, V., Zelezný, F., Lavrac, N.: Advancing data mining workflow construction: A framework and cases using the orange toolkit. In: Proceedings of the SoKD-2009 International Workshop on Third Generation Data Mining at ECML PKDD 2009, pp. 39–51 (2009)
71. Zhong, N., Liu, C., Ohsuga, S.: Dynamically organizing kdd processes. International Journal of Pattern Recognition and Artificial Intelligence 15(3), 451–473 (2001)
72. Zhong, N., Matsui, Y., Okuno, T., Liu, C.: Framework of a multi-agent kdd system. In: Yin, H., Allinson, N.M., Freeman, R., Keane, J.A., Hubbard, S. (eds.) IDEAL 2002. LNCS, vol. 2412, pp. 337–346. Springer, Heidelberg (2002)
73. Zhong, N., Ohsuga, S.: The gls discovery system: its goal, architecture and current results. In: Raś, Z.W., Zemankova, M. (eds.) ISMIS 1994. LNCS, vol. 869, pp. 233–244. Springer, Heidelberg (1994)

Computational Intelligence for Meta-Learning: A Promising Avenue of Research

Ciro Castiello and Anna Maria Fanelli

Dipartimento di Informatica, Università degli Studi di Bari
via E. Orabona, 4, 70125 – Bari, Italy
castiello@di.uniba.it, fanelli@di.uniba.it

Abstract. The common practices of machine learning appear to be frustrated by a number of theoretical results denying the possibility of any meaningful implementation of a "superior" learning algorithm. However, there exist some general assumptions that, even when overlooked, preside the activity of researchers and practitioners. A thorough reflection over such essential premises brings forward the meta-learning approach as the most suitable for escaping the long-dated riddle of induction claiming also an epistemologic soundness. Several examples of meta-learning models can be found in literature, yet the combination of computational intelligence techniques with meta-learning models still remains scarcely explored. Our contribution to this particular research line consists in the realisation of MINDFUL, a meta-learning system based on the neuro-fuzzy hybridisation. We present the MINDFUL system firstly situating it inside the general context of the meta-learning frameworks proposed in literature. Finally, a complete session of experiments is illustrated, comprising both base-level and meta-level learning activity. The appreciable experimental results underline the suitability of the MINDFUL system for managing past accumulated learning experience while facing novel tasks.

1 Introduction

The applied research in the field of artificial intelligent systems is often addressed to the empirical evaluation of learning algorithms with the aim of proving the (selective) superiority of a particular learning model. This kind of strategy is typical of a "case-study" approach, in which different learning models are evaluated on several datasets in order to identify the best algorithm for a specific problem [1, 2]. The selective pre-eminence exhibited by a learner in the context of a case-study application reflects the intrinsic character of the so-called base learning strategies, where data driven models are involved showing generalisation capabilities when applied to solve particular tasks. More precisely, base learning approaches are connected with the use of a fixed bias representing the whole hypotheses, limitations and choices at the basis of the behaviour of a learner. A fixed bias learner is therefore characterised by a restricted domain of expertise and a limited area of application. Some theoretical results established the intrinsic limitations of base-learning strategies. The "No Free Lunch"

N. Jankowski et al. (Eds.): Meta-Learning in Computational Intelligence, SCI 358, pp. 157–177.
springerlink.com

theorem states the fundamental equality in terms of performance of any pair of learners (when all possible tasks are considered) and rejects the superiority of some particular learning model beyond the "case-study" dimension. In opposition to base-learning, recent researches on meta-learning mechanisms propose to increase the possibilities of inductive learning algorithms by adapting their working mechanisms to the considered task domain. In this way, it can be possible to develop a dynamic search of the most appropriate bias for each task [3, 4, 5, 6].

In this context, the research community is ultimately showing an increasing interest for the adoption of computation intelligence techniques to foster meta-learning activity. In this chapter we propose an original meta-learning system called MINDFUL which is based on the neuro-fuzzy hybridisation: to best of our knowledge it represents one of the few examples in literature of this particular instantiation of meta-learning models. We believe that the neuro-fuzzy paradigm lends itself well to be exploited for meta-learning purposes, since it provides a formal apparatus where knowledge can be properly extracted from data and organised in structured form by means of linguistic rules. The reported results of a complete session of experiments testify the suitability of the MINDFUL system for accumulating and exploiting past experience in novel learning tasks.

The chapter is organised as follows. A preliminary discussion about the limitations of base-learning strategies is carried out in the next session together with the presentation of some theoretical results. In section 3 we provide an introduction to the meta-learning approach discussing its intrinsic potentialities. A general framework of meta-learning is presented in section 4, including some comments and a brief review of different implementations of meta-learning proposed in literature. Section 5 is devoted to the illustration of the MINDFUL system and section 6 shows in details the results of a complete experimental session articulated in base-level and meta-level activities. Section 7 closes the paper with a broad discussion, pointing out the ensemble of critical topics debated and drawing some conclusive remarks.

2 Theoretical Arguments

Induction is one of the most powerful tools in human reasoning, yet it is also a very fragile one. What can be said about the validity of induction? A number of considerations are involved in the analysis of this question, pertaining to the fields of philosophy, mathematical logics, machine learning. In this respect, the words of Hume are often quoted [7]:

> There can be no demonstrative arguments to prove, that those instances, of which we have had no experience, resemble those, of which we have had experience. [...] Thus not only our reason fails us in the discovery of the ultimate connexion of causes and effects, but even after experience has informed us of their constant conjunction, it is impossible for us to satisfy ourselves by our reason, why we should extend that experience beyond those particular instances, which have fallen under our observation.

The Scottish philosopher deprived of necessity the causality principle: the statement "A causes B" should be corrected by "in our experience A and B appeared together frequently", but although several examples have been observed, there is no reason to expect A and B joined in future circumstances. Therefore from a purely rational point of view our inductive practice has no more justification than random guessing.

Quite surprisingly, the same conclusions can be drawn from the so-called "No Free Lunch" (NFL) theorem [8, 9], popularised in the field of machine learning also as the law of conservation for generalisation performance [10], stating that the generalisation performance of any learner has zero sum when averaged on all possible learning tasks. For several years this kind of results stood as a blind alley for any researcher looking for the way of designing a "universal" learning algorithm.

The following elementary illustration (originally reported in [11]) provides an insight into the NFL theorem. Table 1 reports the input vectors consisting of three binary features (in column $\mathbf{x}$) and the values of a target function $F(\mathbf{x})$ which must be learned by means of some learning algorithm: h_1 and h_2 stand as a couple of hypotheses produced by two trivial algorithms assigning to any input the value 1 and -1 respectively, unless trained otherwise.

Table 1. Output values of a target function F and two hypotheses h_1, h_2 corresponding to 3-digits binary input vectors.

	$\mathbf{x}$	F	h_1	h_2
	000	1	1	1
$\mathcal{D}$	001	-1	-1	-1
	010	1	1	1
	011	-1	1	-1
	100	1	1	-1
	101	-1	1	-1
	110	1	1	-1
	111	1	1	-1

If we consider the set $\mathcal{D}$ as a training set, it is possible to assess the hypotheses behaviour over the remaining input vectors, thus enabling an off-training set (OTS) evaluation (which is the one adopted in the NFL scenario). With respect to the particular target function F, algorithm 1 (yielding hypothesis h_1) is better than algorithm 2 (yielding hypothesis h_2). Yet, since the target function is *a-priori unknown*, the algorithms must be compared in the overall, i.e. by averaging over all the 2^5 possible target functions consistent with $\mathcal{D}$. It is easy to observe that, in this overall evaluation, no difference emerges the in off-training set errors between the two algorithms. In fact, for each distinct target function there is another one whose output is inverted with respect to all the off-training set input vectors, therefore performances of algorithms 1 and 2 will be ultimately the same.

In sum, the NFL theorem states that, if all target functions are equally likely, any effort at selecting the "best" learning algorithm is frustrated, and averaged OTS generalisation does not draw away from random guessing (no "rational" motivation supports the choice of an inductive algorithm over others).

How to conciliate the restrictive results of the NFL theorem with the ongoing research in machine learning? Firstly, the main hypotheses of the theorem should be highlighted:

1. all the target functions (namely, all the learning tasks) are equally probable, being characterised by a uniform probability distribution;
2. performance of learning algorithms are evaluated under the OTS regime.

Admittedly, the analysis of instances which have not been encountered during training is the most relevant, however focus should be addressed to the *expected* generalisation performance of a learner, thus suggesting that some OTS instances are more likely than others. On a broader plane, a purely empirical consideration lets us assert that some learning tasks are more likely than others, in contrast with hypothesis 1. of the NFL theorem. This kind of empirical arguments have been properly formalised in [12]: the point is to make clear some assumptions that are usually taken for grant in the machine learning community.

Remark 1. The *weak assumption* of machine learning is that there exists a learning task selection process Ω inducing a non-uniform probability pr^{Ω} over the learning task space.

In other words, it is commonly assumed that some tasks are more likely than others, hence the possibility of defining a "better" algorithm.

Remark 2. The *strong assumption* of machine learning is that the probability distribution pr^{Ω} is implicitly or explicitly known (at least to a certain degree of approximation).

The strong assumption constitutes the basis for the definition of the bias of a learning algorithm.

Definition 1. *The bias constitutes the ensemble of factors influencing a learning process (apart from the instances of the task being learned). These factors include the criteria adopted to perform the search inside the space of the task solutions and the specific configuration of the learning algorithm.*

The concept of bias can be regarded as the material representation of the strong assumption and the probability distribution pr^{Ω} defines the *area of expertise* of each learning algorithm [13].

3 Introducing the Meta-Learning Approach

On the basis of the previous argumentations, it is possible to say that each learner starts from the analysis of data to induce a particular model producing its own probability distribution pr over the space of the learning tasks. The main

concern in machine learning, therefore, is to derive a learning algorithm whose corresponding model is able to produce a probability distribution pr being as near as possible the actual probability distribution pr^{Ω}.

To tackle this problem the usual way of proceeding consists in building up a particular learner from scratch in a manual designing fashion. In other words, every practitioner, when dealing with a learning task for which no existing algorithm applies sufficiently well, builds up his own learning model after a trial and error session of experiments. This is the so-called "case-study" dimension, where specific learning models are empirically evaluated on a restricted number of datasets in order to prove their "superiority" which is intrinsically selective in its essence. What does it mean in terms of the previously reported theoretical analysis? The strong assumption of machine learning implicitly plays a pivotal role since the case-study dimension pretends a thorough knowledge of the distribution pr^{Ω}, so that it can be embedded into the (fixed) bias of the proposed learning algorithm. On a broader plane, this kind of base-learning mechanism is supposed to enable an accretion of the overall learning knowledge, with each practitioner adding his contribution to the community of the machine learning researchers.

The study of meta-learning strategies has been consolidated during the last decade to set up an ensemble of approaches contrasting the "classical" base-learning mechanisms (commonly adopted in the manual design of learning algorithms). Various meanings can be found in literature coupled with the meta-learning catchword [14, 15, 16, 17]. The wholeness of references, however, agrees to consider the meta-learning approach as a research apparatus for discovering learning algorithms that improve their efficiency through experience. While base-learning focuses on the accumulated experience related to single learning tasks, meta-learning focuses on the accumulated experience related to the repeated applications of learning algorithms, thus configuring as a dynamical strategy oriented to catching the correspondence between tasks (or task domains) and classes of learners [5, 6, 18, 19].

Formally, we can state the following definition for a meta-learning process.

Definition 2. *Let $\mathcal{A}$ be a set of learning algorithms and $\mathcal{T}$ a set of tasks. Let $a_{\mathcal{A}}(t)$ be the best algorithm in $\mathcal{A}$ applicable to a specific task t, for each $t \in \mathcal{T}$, and $c(t)$ a characterisation of the chosen task t. Then a meta-learning process is an automatic mechanism that starting from the meta-training set:*

$$\{< c(t), a_{\mathcal{A}}(t) > : \ t \in \mathcal{T}\}, \tag{1}$$

induces a meta-model which is able to predict, for any new task, the best model in $\mathcal{A}$.

It should be highlighted that meta-learning goes halfway through the weak and the strong assumptions (see remarks 1, 2). In fact, even though the existence of a probability distribution pr^{Ω} is admitted, there is no pretension of knowing it. That is different from the "manual design" approach which is based on the availability of *a priori* knowledge about pr^{Ω} to be incorporated into the learning

algorithm. A meta-learning process, instead, aims at *estimating* pr^{Ω}: in this case, the assumption is that the meta-data in the meta-training set (1) are suitable to approximate (learn) pr^{Ω}.

The meta-learning assumption appears to be more plausible than the one adopted in the "case-study" formulation. Moreover, it allows to provide some kind of reply to Hume's scepticism about induction. In fact, even admitting our impossibility to establish a rational foundation for the inductive mechanisms, yet they are considered valid generalisation tools in the way they are employed to face the physical world around us. Generalising from what we have seen to whatever is yet to be encountered is the common practice in our everyday life[1] and it represents also the fundamental assumption of science as we practice it [20].

4 A General Framework for Meta-Learning

In order to discuss the main characteristics a meta-learning system should exhibit, we refer to a general framework originally introduced in [21] which has been reconsidered in recent times since it lends itself to the meta-learning formalisation [22].

Here the framework is adopted with the twofold aim of presenting the state of art on meta-learning systems proposed in literature and introducing an original meta-learning system developed in the specific context of computational intelligence.

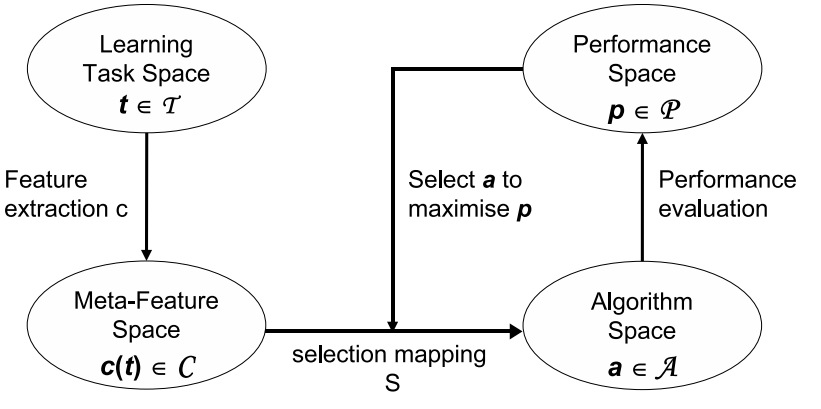


Fig. 1. A general framework for meta-learning.

As shown in figure 1, the notation adopted is coherent with the formalisation of the meta-learning process (1) expressed in definition 2. In particular, for a given

[1] To some extent, this argumentation resembles the ultimate conclusions of Hume: only habit (namely, repeated observation of regularities) is responsible for the generalisation practice [7].

learning task t inside the learning task space $\mathcal{T}$, with features $c(t)$ inside the meta-feature space $\mathcal{C}$, the framework is oriented to find the selection mapping $S(c(t))$ into the algorithm space $\mathcal{A}$, such that the selected algorithm $a \in \mathcal{A}$ maximises a performance measure p inside the performance space $\mathcal{P}$.

The definition of such a general framework paves the way for the analysis of a number of issues concerning:

1. the arrangement of the spaces $\mathcal{T}$ and $\mathcal{A}$;
2. the meta-feature extraction procedure $c(t)$;
3. the selection mapping $S(c(t))$;
4. the performance measure p.

We are going to discuss each of them in the following sections, taking the opportunity also for reviewing some meta-learning strategy proposed in literature.

4.1 About the $\mathcal{T}$ and $\mathcal{A}$ Spaces

The definition of the spaces $\mathcal{T}$ and $\mathcal{A}$ is based on the coverage of the largest number of cases and on the size of the two sets. As concerning $\mathcal{T}$, this is a virtually infinite set encompassing the whole range of learning tasks. Actually, the number of accessible, documented, real-world classification tasks is quite scarce (less than a couple hundreds) [23]. To overcome this problem some strategies have been proposed. On the one hand, the cardinality of $\mathcal{T}$ can be increased through the generation of synthetic tasks, as in [24]. On the other hand, the meta-learning process can be re-organised in order to produce a kind of incremental learning where a stream of learning tasks are faced in a life-long learning scenario to gradually tune the bias (mimicking human learning) [25, 26].

As concerning the algorithm space, the strong assumption (see remark 2) associates a single bias to each algorithm in $\mathcal{A}$. The latter, therefore, should encompass a reduced number of learning models, yet ensuring a reasonable variety of inductive biases. In this sense, the algorithm selection process is assimilated to a dynamic selection of the proper bias through the analysis of several learning tasks by different learning models. However, in a more general formulation of the meta-learning framework, the set $\mathcal{A}$ can be assimilated to the set of biases referring to a single learning model. In this way, the algorithm selection process would assume the role of a *parameter selection* function, aiming at the identification of the best parameter configuration (bias) pertaining to a single learning algorithm during its application in several different leaning tasks [27, 28].

4.2 About the Meta-Feature Extraction

The meta-features should be defined in order to describe the main properties of a specific task and should be data-driven computed. However, they should convey some different pieces of information with respect to the mere instances available for the task at hand. Moreover, an adequate set of meta-features must satisfy mainly two basic conditions. Firstly, they should prove to be useful in

Table 2. Some commonly employed meta-features.

General meta-features
Number of observations
Number of attributes
Number of output values
Dataset dimensionality ($\frac{\text{attribute}}{\text{observations}}$)
Statistical meta-features
(Mean) standard deviation
(Mean) coefficient of variation
(Mean) covariance
(Mean) linear correlation coefficient
(Mean) skewness
(Mean) kurtosis
1-D variance fraction coefficient
Information-theoretic meta-features
(Normalised) Class entropy
(Mean normalised) Attribute entropy
(Mean) Joint entropy class/attribute
(Mean) Mutual information class/attribute
Equivalent number of attributes
Noise-signal ratio

determining the relative performance of individual learning algorithms. Secondly, their computation should not be too difficult and burdensome.

Tasks characterised by similar meta-features are supposed to map to the same learning algorithms which, in turn, should exhibit similar performance results. A number of morphological measures have been proposed to characterise data related to a learning problem: a common practice has been established in focusing over general, statistical and information-theoretic measures. Meta-features have been commonly employed in several works [29, 30, 31, 32] and projects, such as StatLog and METAL. Table 2 reports a selection of meta-features; for a thorough description of those measures the interested reader is addressed to [33].

In literature can be found some other different strategies for extracting meta-features from data. In [34, 35, 36] meta-characteristics are extracted by the analysis of a decision tree obtained from the data of the learning task at hand. The properties extracted from the tree (such as nodes per feature, maximum tree depth, shape and tree imbalance) serve as a basis to explain the performance of other learning algorithms applied to the same task. In [13,37] the proposed meta-learning strategy characterises the task by means of landmarking. In this case, the recourse to meta-feature evaluation is avoided by employing simple learners, called landmarkers, whose cross-tasks performances is referred to characterise a learning domain.

4.3 About the Selection Mapping

The selection mapping S is devoted to the identification of the best algorithm to be associated to the meta-characteristics $c(t)$ extracted by a given task $t \in \mathcal{T}$. Chronologically, the very first approaches for the implementation of S required a human expert to manually build a set of meta-rules defined for associating the task domain characteristics to the learning algorithms [2, 38]. In the following, every effort has been paid to develop some automatic mechanism in order to overcome the limitations exhibited by the manual rule construction approach. The general idea consists in the application of a base-learning algorithm to the meta-training set, so that the mapping between the input and the output components in (1) can be automatically revealed. In this way, the learning experience acquired during several sessions of base-learning (and stored in the meta-training set) induces a generalisation hypothesis concerning the mapping established between different tasks and learning algorithms. The choice provided by the selection process S can be single or multiple, depending on the expected output results. In fact, a single base-learning algorithm can be chosen among a pool of candidates in $\mathcal{A}$; alternatively, a ranking of several algorithms can be provided [39, 40, 41].

4.4 About the Performance Measure

As concerning the performance measure p, predictive accuracy is the most widely adopted criterion in literature. A number of motivations support this choice, ranging from the relative easiness in computing accuracy values, to the possibility of establishing a ranking of different hypotheses in terms of predictive accuracy. However, some other factors inside the space $\mathcal{P}$ can be taken into account to assess the performance of a learning algorithm (including, for instance, expressiveness, compactness, computational complexity, comprehensibility). In [42] several alternative measures of performance are purposely discussed.

5 Computational Intelligence for Meta-Learning: An Original Proposal

In our excursus concerning different proposal of meta-learning strategies, we have not mentioned Computational Intelligence (CI) approaches. Actually, it appears that the connection between meta-learning and computation intelligence was not deeply explored in the last decade[2], yet we consider that such a marriage represents a promising avenue of research (as witnessed also by this edited book). In some previous works we have investigated the rationale behind the organisation of a meta-learning framework on the basis of hybrid integration strategies [27,44]

[2] A remarkable exception is represented by the work of Abraham [43] proposing an automatic computational framework, intended to adaptively define the architecture, connection weights and learning algorithm parameters of a neural network, with recourse to evolutionary algorithms.

and we have proposed a theoretical analysis oriented to assess the potentiality of meta-learning methods versus the common base-learning practices using the field of connectionist learning as a research laboratory, exploring the meta-learning possibilities of neural network systems [45]. Here we intend to present an original meta-learning system based on the neuro-fuzzy integration. The name of the system is MINDFUL (Meta-INDuctive neuro-FUzzy Learning) and we are going to illustrate both its working engine and a complete session of experiments.

The MINDFUL system is based on the employment of a single learning scheme: a neuro-fuzzy system plays the twofold role of base-learner (to tackle ordinary predictive learning tasks) and meta-learner (to produce some form of meta-knowledge). With reference to the general framework for meta-learning presented in section 4, here we deal with an algorithm selection process assuming the role of a parameter selection function, oriented to identify the best parameter configuration pertaining to the single learning scheme of the MINDFUL system (the set $\mathcal{A}$ actually represents the set of biases referring to the neuro-fuzzy model). In this way, we are able to characterise our approach on the following key points:

- the meta-learning strategy is translated to a more qualified level: it is not intended as simply picking a learning procedure among a pool of candidates, but it focuses on a deeper analysis of the learning model behaviour, in order to understand and possibly to improve it;
- the choice for a single learning model should be suitable to preserve the uniformity of the whole system and to reduce its complexity, even in terms of comprehensibility;
- the adoption of a neuro-fuzzy strategy, applied both at base- and meta-level, endows also the meta-learning procedure with the benefits deriving from the integration of the connectionist paradigm with fuzzy logic.

As known, the neuro-fuzzy hybridisation is one of the most productive approach in the context of CI. On the one hand, fuzzy logic provides a formal apparatus that enables a rigorous treatment of reasoning processes typical of human cognitive patterns [46]. It promotes also a form of "computing with words" where natural language can be adopted to set up a linguistic representation of knowledge [47]. On the other hand, neural networks provide a powerful means for learning and extracting knowledge from data. In this way, the neuro-fuzzy integration combines complementary aspects of machine learning, allowing fuzzy inference systems to adjust linguistic information using numerical data and neural networks to manage interpretable knowledge [48, 49, 50].

5.1 The Working Engine of the Mindful System

The neuro-fuzzy strategy constituting the kernel of the MINDFUL system adopts a working scheme similar to the ANFIS model [51] to codify knowledge in form of fuzzy rules:

$$\begin{array}{l}\text{IF } x_1 \text{ is } A_1^r \text{ AND} \ldots \text{AND } x_m \text{ is } A_m^r \\ \text{THEN } y_1 \text{ is } b_1^r \text{ AND} \ldots \text{AND } y_n \text{ is } b_n^r,\end{array} \tag{2}$$

where the index $r = 1, \ldots, R$ indicates the r-th rule among the R comprised into the rule base; A_i^r are fuzzy sets (defined over the input components x_i, $i = 1, \ldots, m$) expressed in terms of Gaussian membership functions; b_j^r are fuzzy singletons (defined over the output components y_j, $j = 1, \ldots, n$). The fuzzy inference system is comparable to the Takagi-Sugeno-Kang (TSK) method of fuzzy inference [52]: the fulfilment degree of each rule is evaluated using the product operator as the particular T-norm interpreting the AND connective. The components of the output vector inferred by the fuzzy rule base are evaluated as the weighted average values of the rule activation strengths with respect to the singletons. The fuzzy inference system is translated into a three-layer feed-forward neural network (the neuro-fuzzy network), which reflects the rule base in its parameters and topology, and the learning scheme of the MINDFUL's kernel module is articulated in two successive steps, intended to firstly initialise a knowledge structure and then to refine the obtained fuzzy rule base. During the first step, a clustering of the input data is performed by an unsupervised learning process of the neuro-fuzzy network: this clustering process is able to derive the proper number of clusters starting from a guessed number $\tilde{R}$. In this way, an initial knowledge is extracted from data and expressed in form of base of rules. The obtained knowledge is successively refined during the second step, where a supervised learning process of the neuro-fuzzy network is accomplished (based on a standard gradient descent technique), in order to attune the parameters of the fuzzy rule base to the numerical data. (Further details about the learning scheme of the neuro-fuzzy network can be retrieved in some other publication of ours [53].)

The main factors involved in the first step of learning are represented by the initial number $\tilde{R}$ of rules and the learning rates α_ω and α_ρ associated to the rival penalised learning. The learning rate η of the back-propagation algorithm can be identified as the main factor involved in the second step of learning.

The scheme of the meta-learning strategy performed by the MINDFUL system is illustrated in figure 2. The overall activity of MINDFUL is based on the previously described neuro-fuzzy model acting as base-learner (when tackling ordinary predictive learning tasks) and as meta-learner (to produce some kind of meta-knowledge). The difference is in the dataset to be handled: when dealing with base-learning tasks the dataset contains observational data related to the problem at hand. We assume that the available instances are expressed as a set of samples, each of them being a couple of input-output vectors, respectively indicated by $\mathbf{x} = (x_1, \ldots, x_m)$ and $\mathbf{y} = (y_1, \ldots, y_n)$. The (base-) knowledge derived from the base-learning activity obviously concerns only the particular task under study. The meta-learning process, instead, is based on the analysis of a meta-level dataset where information is expressed as in (1) (see section 3), where the task characterisation $c(t)$ is a vector of meta-features extracted from the base-level datasets, and the best bias $a(t) \in \mathcal{A}$ is meant as a parameter configuration of the neuro-fuzzy learning scheme. The end product of the meta-learning activity is the formulation of a (meta-) knowledge describing the correlation between each specific task and the proper parametrisation to be used while base-learning

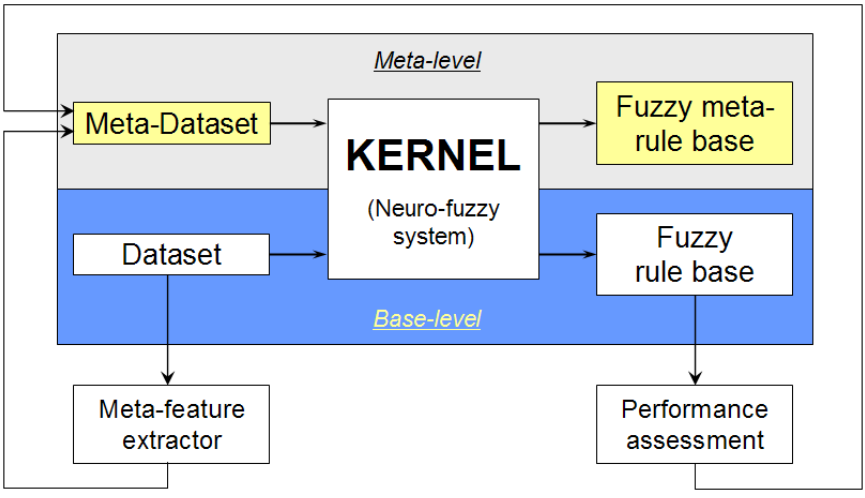


Fig. 2. The meta-learning strategy performed by the MINDFUL system

the task. Since the learning activity at base- and meta-level is performed by the same neuro-fuzzy model, both the base-knowledge and the meta-knowledge are ultimately expressed in form of fuzzy rule base, where every rule is formalised as in (2).

As shown in figure 2, the overall activity of the MINDFUL system is supported by a couple of additional components: the Meta-Feature Extractor (MFE) module, which extracts a set of features needed to characterise the base-level tasks under analysis, and the Performance Assessment (PA) module, which identifies a specific parameter configuration for each particular task. Hence, the two modules are devoted to assemble the meta-training set that will be investigated by the MINDFUL's kernel module during the meta-learning practice. In particular, the MFE module computes a subset of the meta-features reported in table 2, namely: number of attributes, number of output values, mean linear correlation coefficient, mean skewness, mean kurtosis, 1-D variance fraction coefficient, normalised class entropy, mean normalised attribute entropy, normalised mutual information class/attribute, equivalent number of attributes, noise-signal ratio (all of them have been selected as a result of a focused analysis discussed in [27, 33]). The PA module associates to each task characterisation the most suitable parameter configuration of the neuro-fuzzy learning process (in the way it can be identified while experimenting at base-learning level). In practice, the parametrisation refers to the configurations of parameters presiding over the general learning process (i.e. the previously mentioned learning rates and the initial number $\tilde{R}$ of rules). It must be highlighted that the performance criterion adopted by the PA module to identify the best parameter configuration to be associated to each task consists in a trade-off between the accuracy and the complexity of the derived knowledge (namely, the final learning error and

the number of rules of the fuzzy rule base). In other words, the MFE and the PA modules stand as the meta-feature extraction procedure c and the selection mapping S reported in figure 1, respectively.

6 Experiments with the Mindful System

For the sake of illustration we report a complete session of experiments performed by the MINDFUL system into the `Matlab`© environment. However, a more detailed experimentation (involving some preliminary testing with synthetic tasks and a thorough session of experiments performed on real-world datasets) can be found in [54].

The overall strategy consists in performing a preliminary set of base-learning experiments, useful to determine the best bias configurations in correspondence with various types of tasks. Then, a meta-training set is established, properly organising the information provided by the MFE and the PA modules. On the basis of the meta-training set, the meta-learning process is accomplished, in order to derive a fuzzy rule base embedding the extensively extracted meta-knowledge. Finally, different cases of meta-knowledge exploitation are shown to demonstrate how the meta-learning strategy can be useful in enhancing the performance of novel base-learning processes.

A total number of 10 datasets are involved in the experimental session, 9 of them directly contributed to the construction of the meta-training set. Table 3 reports a brief description for each dataset. They have been chosen to represent different classification tasks, where only numerical attributes are involved[3]. Moreover, some of them have been subjected to a pre-processing phase, to remedy particular problems, such as missing values and excessive number of input features.

Table 3. Description of the involved datasets (m denotes the number of input features, n the number of classes, K the number of instances).

Code	Dataset	m	n	K	Source
BCW	Breast-Cancer Wisconsin	9	2	683	UCI [23]
BLS	Balance Scale	4	3	625	UCI [23]
GLS	Glass Identification	9	2	214	UCI [23]
$GT0$	Geometric Task	2	2	1000	Synthetic
ION	Ionosphere	18	2	351	UCI [23]
IRS	Iris	4	3	150	UCI [23]
$MT0$	Medical Task	9	2	1000	Synthetic
PID	Pima Indians Diabetes	8	2	768	UCI [23]
SEG	Image Segmentation	18	7	2310	UCI [23]
WIN	Wine	12	3	178	UCI [23]

[3] It has been chosen to perform the base-learning activity of the MINDFUL system on classification tasks only, hence the output components of the task instances are made up of binary vectors indicating the membership of each instance to a particular class.

Table 4. The adopted bias configurations.

Bias conf.	α_ω	α_ρ	η	$\tilde{R}$
$Bias_1$	0.90	0.0001	0.85	3
$Bias_2$	0.09	0.0001	0.90	5
$Bias_3$	0.01	0.0010	0.50	7
$Bias_4$	0.01	0.0001	0.50	10
$Bias_5$	0.09	0.0100	0.25	13

Table 5. The summary of the base-learning session of experiments ($Bias_1$, and $Bias_2$ were not employed with the SEG dataset, since they are characterised by an inadequate number of rules to tackle a task with 7 output classes).

Dataset	$Bias_1$	$Bias_2$	$Bias_3$	$Bias_4$	$Bias_5$
BCW	**0.0364**	0.0408	0.0363	0.0349	0.0365
GLS	**0.0762**	0.1088	0.1190	0.0714	0.0937
$GT0$	0.2650	0.0840	**0.0470**	0.0510	0.0530
ION	0.0900	**0.0647**	0.0882	0.0677	0.0900
IRS	**0.0410**	0.0533	0.0473	0.0533	0.0533
$MT0$	0.1370	0.0820	0.0680	**0.0480**	0.0380
PID	0.3100	0.2915	**0.2590**	0.2915	0.2967
SEG	-	-	0.2658	0.2398	**0.1865**
WIN	0.1618	**0.0736**	0.0666	0.0673	0.1403

The neuro-fuzzy learning strategy has been applied to conduct the base-level activity on the basis of the datasets described in table 3 (excepting the BLS dataset, which has been kept for the subsequent phase of meta-knowledge assessment). In this way, we accumulated a learning experience that could be successively encoded into the meta-training set. Relying on the past usage of the hybrid learning strategy [53], we identified the 5 parameter configurations reported in table 4 which appeared to be quite discriminatory for the obtainable performance results (in [54] the problem of evaluating a reduced subset of parameters among infinite possibilities is discussed in details).

A 10-fold cross-validation procedure was conducted for investigating each of the 9 datasets with the kernel of the MINDFUL system, in order to recognise the configuration providing the best performance results for each classification task. It should be underlined that, although the obtained results can be considered satisfactory for almost every examined problem, the main objective of these experiments was to accumulate learning experience, instead of simply obtaining the most accurate outcomes possible.

Table 5 sums up the average classification error values achieved for each classification task during the 10-fold cross-validation sessions. The values emphasised in bold characters refer to the bias configurations yielding the best results (determined by the PA module in terms of trade-off between accuracy and complexity

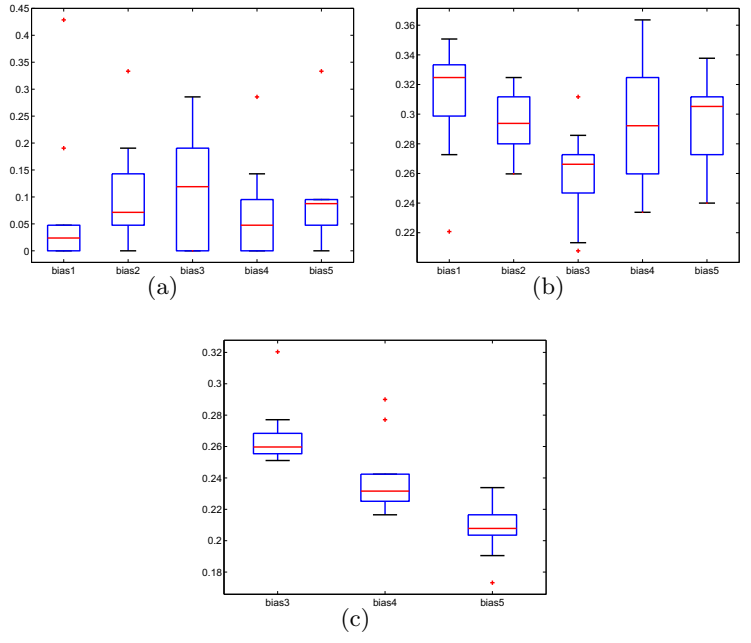


Fig. 3. Box and whiskers plots describing the variation of the classification errors, according to the selected bias configuration. (a) *GLS* dataset, (b) *PID* dataset, (c) *SEG* dataset.

of the derived fuzzy rule base). Correspondingly, figure 3 shows the 10-fold cross-validation results for three sample datasets, for each of the applied bias configurations. By analysing the figure and the table, it can be argued that $Bias_1$, $Bias_3$ and $Bias_5$ stand as the best parameter configurations to be adopted for the *GLS*, *PID* and *SEG* datasets, respectively.

The accumulated learning experience has been codified into a meta-training set by simply correlating the pieces of information deriving from the MFE and PA modules. By considering each dataset analysed during the 10 fold cross-validation procedure, the meta-training set is composed by 90 samples.

To extract meta-knowledge from the accumulated base-level experience, the same neuro-fuzzy strategy performed by the kernel of the MINDFUL system has been replied over the meta-training set. In this case, the particular task to be faced consists in a regression problem, where the output values are represented by the numerical bias parameters. During the meta-learning sessions of experiments, the hybrid learning procedure has been performed over the meta-training set by adopting the same five bias configurations described in table 4 (with the exclusion of $Bias_1$). In the end, the best parameter configuration to be applied on the meta-training set resulted to be $Bias_3$, as demonstrated by table 6, where the error values are reported in terms of Mean Squared Error, together with the final number of rules of the obtained fuzzy rule bases. In this way, the fuzzy rule base eventually embedding the meta-knowledge is composed by 6 fuzzy rules.

Table 6. Results obtained during the meta-learning experimental sessions, employing four out of the five bias configurations.

Bias configuration	Final number of rules	MSE accuracy
$Bias_2$	5	0.7772
$Bias_3$	6	0.1828
$Bias_4$	10	0.2964
$Bias_5$	11	0.2317

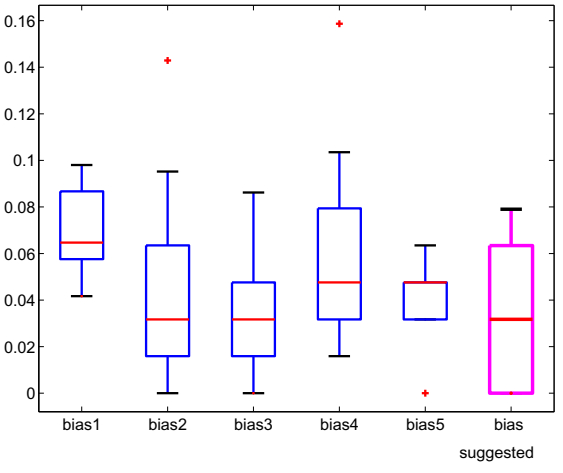


Fig. 4. Box and whiskers plot related to the experiments on the *BLS* dataset.

In order to functionally evaluate the appropriateness of the derived meta-knowledge in capturing the experience accumulated during the past sessions of experiments, we dealt with new base-level situations, supported by the bias suggestions provided by the meta-knowledge. In our simulation, we resolved to investigate the *BLS* dataset (as previously specified, it was left out from the number of datasets employed to derive the meta-training set). In detail, the meta-features extracted from the *BLS* dataset have been used as input values for the fuzzy meta-rule base to infer a configuration of bias parameters. In order to evaluate the suggested parameter setting, a 10-fold cross-validation has been executed both on the previously adopted ensemble of bias configurations, and on the newly inferred bias configuration. The results of these experimental sessions are summarised by the box and whiskers plot depicted in figure 4. As it can be easily observed, all the employed bias configurations yield satisfactory outcomes when applied to tackle the *BLS* dataset. However, the suggested bias is able to produce excelling results when compared with the other parameter settings.

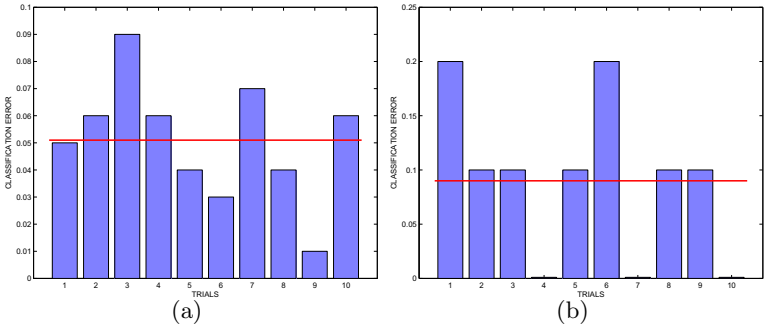


Fig. 5. Comparison of the learning results obtained at the end of the 10-fold cross-validation procedure performed over the $MT0$ dataset using (a) the complete dataset (1000 instances) and (b) the impoverished dataset set (100 instances). The horizontal lines indicate the mean error values.

For further evaluating the meta-knowledge, we planned to re-analyse some of the classification tasks already tackled during the base-learning session of experiments. In this way, it is possible to verify how the meta-knowledge may help to remedy a reduced amount of information concerning a particular task, simulated by the employment of an impoverished training set. As an example, we can consider the re-investigation of the $MT0$ dataset, where a number of only 100 instances have been used as training set (sampled from the original 1000 instances). Again, the meta-features extracted from the newly adapted dataset were useful to infer a suggested bias configuration from the fuzzy meta-rule base and a complete session of 10-fold cross-validation has been performed for the sake of evaluation. On average, the performance results obtained with the inferred bias are comparable with those produced by the previous session of base-learning experiments (proportionally to the impoverishment rate of the dataset). As demonstrated by the graphs reported in figure 5, a restrained increase of the classification error corresponds to a drastic reduction of the training set (of the order of 90% of the training samples).

7 Conclusions

The majority of artificial learning activity traditionally aims at developing learning models on the basis of a *tabula rasa* approach, where a particular task is tackled by examining a number of training examples. In this way, often the literature panorama draws attention to the realisation of intelligent systems which prove their efficiency over specific tasks, in a kind of "case-study" evaluation. Invariably, some algorithms are reported to excel others in a more or less significant way, mostly by means of empirical evaluations, instead of more appropriated formal analysis. Although the pragmatic focus on the obtained results for a problem at hand may be acceptable (or even desirable) in some engineering

contexts, it appears that a more extensive analysis is often necessary. Actually, the *tabula rasa* approach necessarily implies that each learning algorithm exhibits a selective superiority, leading to a good fit for some learning tasks and a poor fit for others. In fact, a model induced from a particular set of observations is unlikely to extend its capabilities on unseen cases, without resorting to a new set of training samples. Even, a learning model is not capable to treasure past accumulated knowledge, in order to face related tasks with a better degree of efficiency. It appears that the widest gap to fill between artificial and human learning consists in the unsatisfactory aptitude of the machines in improving their ability to learn tasks, just like common people do during their life. Moreover, from a practical perspective, an adequate disposition in retaining and exploiting past experience would be helpful also to overcome the deficiency of training samples suffered in many real world applications. The limitations of base-learning strategies can be stated even by referring to some theoretically established results: the "No Free Lunch" theorem expresses the fundamental performance equality of any chosen couple of learners (when averaged on every task), and denies the superiority of specific learning models outside the case-study dimension. In this chapter the meta-learning approach has been proposed as the most natural way of performing a dynamical bias discovering. Starting from the analysis of a general meta-learning framework, which offered the opportunity for reviewing some basic techniques in meta-learning literature, we brought forward the proposal of a particular meta-inductive neuro-fuzzy learning system, the MINDFUL system, based on the neuro-fuzzy integration. Actually, it is not so common to find in literature examples of computational intelligence techniques employed inside meta-learning scenarios. Yet, we believe that such a combination can prove to be very fruitful and a growing interest is registered toward this specific issue (as demonstrated also by the present edited publication). The results of the experimental sessions demonstrate the capability of the MINDFUL system in retaining base-learning experience and exploiting it during meta-level activity. Future work should be addressed in further assessment of the system, while studying also mechanisms for implementing some kind of life-long learning strategies based on incremental improvement of the meta-training set.

References

1. Aha, D.W.: Generalizing from case studies: a case study. In: Proceedings of the Ninth International Conference on Machine Learning, MLC 1992 (1992)
2. Brodley, C.: Addressing the selective superiority problem: automatic algorithm/model class selection. In: Proceedings of the Tenth International Conference on Machine Learning (MLC 1993), pp. 17–24 (1993)
3. Desjardins, M., Gordon, D.: Evaluation and selection of bias in machine learning. Machine Learning 20, 5–22 (1995)
4. Giraud-Carrier, C., Vilalta, R., Brazdil, P.: Introduction to the special issue on meta-learning. Machine Learning 54, 187–193 (2004)
5. Vilalta, R., Drissi, Y.: A perspective view and survey of Meta-Learning. Artificial Intelligence Review 18, 77–95 (2002)

6. Vilalta, R., Giraud-Carrier, C., Brazdil, P.: Meta-Learning: Concepts and Techniques. In: Maimon, O., Rokach, L. (eds.) Data Mining and Knowledge Discovery Handbook: A Complete Guide for Practitioners and Researchers, Springer, Heidelberg (2005)
7. Hume, D.: A Treatise of Human Nature (1740)
8. Wolpert, D.H., Macready, W.G.: No Free Lunch Theorems for Optimization. IEEE Transactions on Evolutionary Computation 1(1), 67–82 (1997)
9. Wolpert, D.H., Macready, W.G.: No Free Lunch Theorems for Search. Technical Report, Santa Fe Institute (1995)
10. Schaffer, C.: A conservation law for generalization performance. In: Proceedings of the Eleventh International Conference on Machine Learning (ICML 1994), pp. 259–265 (1994)
11. Duda, R.O., Hart, P.E., Stork, D.G.: Pattern Classification. John Wiley & Sons, Chichester (2001)
12. Giraud-Carrier, C., Provost, F.: Toward a Justification of Meta-learning: Is the No Free Lunch Theorem a Show-stopper? In: Proceedings of the ICML Workshop on Meta-Learning, pp. 9–16 (2005)
13. Bensusan, H., Giraud-Carrier, C.: Discovering Task Neighbourhoods through Landmark Learning Performances. In: Zighed, D.A., Komorowski, J., Żytkow, J.M. (eds.) PKDD 2000. LNCS (LNAI), vol. 1910, pp. 325–330. Springer, Heidelberg (2000)
14. Chan, P.K., Stolfo, S.J.: Experiments on multistrategy learning by meta-learning. In: Proc. Second International Conference Information and Knowledge Management, pp. 314–323 (1993)
15. Domingos, P.: Knowledge Discovery Via Multiple Models. Intelligent Data Analysis 2, 187–202 (1998)
16. Kalousis, A., Hilario, M.: Model Selection Via Meta-Learning: a Comparative Study. In: Proceedings of the 12th International IEEE Conference on Tools with AI. IEEE Press, Los Alamitos (2000)
17. Schweighhofer, N., Doya, K.: Meta-Learning in Reinforcement Learning. Neural Networks 16, 5–9 (2003)
18. van Someren, M.: Model class selection and construction: Beyond the procrustean approach to machine learning applications. In: Paliouras, G., Karkaletsis, V., Spyropoulos, C.D. (eds.) ACAI 1999. LNCS (LNAI), vol. 2049, pp. 196–217. Springer, Heidelberg (2001)
19. Vilalta, R., Giraud-Carrier CBrazdil, P., Soares, C.: Using Meta-Learning to Support Data Mining. International Journal of Computer Science and Applications 1, 31–45 (2004)
20. Bensusan, H.N.: Automatic Bias Learning: an inquiry into the inductive basis of induction. Ph.D. thesis,school University of Sussex (1999)
21. Rice, J.: The algorithm selection problem. Advances in Computers 15, 65–118 (1976)
22. Smith-Miles, K.A.: Cross-disciplinary Perspectives on Meta-learning for Algorithm Selection. ACM Computing Surveys (2009)
23. Asuncion, A., Newman, D.: UCI Machine Learning Repository (2007), `http://www.ics.uci.edu/~mlearn/MLRepository.html`
24. Silver, D.L.: Selective Transfer of Neural Network task knowledge, Ph.D. thesis,School University of Western Ontarioaddress London, Ontario (2000)
25. Thrun, S.: Is learning the n-th thing any easier than learning the first? In: Touretzky, D., Mozer, M., Hasselmo, M.E. (eds.) Advances in Neural Information Processing Systems, pp. 640–646. MIT Press, Cambridge (1996)

26. Thrun, S.: Lifelong learning algorithms. In: Thrun, S., Pratt, L. (eds.) Learning to learn, pp. 181–209. Kluwer Academic Publishers, Dordrecht (1998)
27. Castiello, C.: Meta-learning: a concern for epistemology and computational intelligence, Ph.D. thesis, school University of Bari, Bari, Italy (2005)
28. Soares, C., Brazdil, P., Kuba, P.: A Meta-Learning Method to Select the Kernel Width in Support Vector Regression. Machine Learning 54, 195–209 (2004)
29. Michie, D., Spiegelhalter, D., Taylor, C.: Machine learning, neural and statistical classification. Ellis Horwood Series in Artificial Intelligence (1994)
30. Gama, J., Brazdil, P.: Characterization of classification algorithms. In: Pinto-Ferreira, C., Mamede, N.J. (eds.) EPIA 1995. LNCS, vol. 990, pp. 83–102. Springer, Heidelberg (1995)
31. Linder, C., Studer, R.: AST: Support for Algorithm Selection with a CBR Approach. In: Recent Advances in Meta-Learning and Future Work, pp. 418–423 (1999)
32. Sohn, S.Y.: Meta analysis of classification algorithms for pattern recognition. JournalIEEE Transactions on Pattern Analysis and Machine Intelligence 21, 1137–1144 (1999)
33. Castiello, C., Castellano, G., Fanelli, A.M.: Meta-data: Characterization of input features for meta-learning. In: Torra, V., Narukawa, Y., Miyamoto, S. (eds.) MDAI 2005. LNCS (LNAI), vol. 3558, pp. 457–468. Springer, Heidelberg (2005)
34. Bensusan, H.: Odd Bites into Bananas Don't Make You Blind: Learning about Simplicity and Attribute Addition. In: Proceedings of the ECML Workshop on Upgrading Learning to the Meta-level: Model Selection and Data Transformation, pp. 30–42 (1998)
35. Bensusan, H., Giraud-Carrier, C., Kennedy, C.: A Higher order Approach to Meta-learning. In: Proceedings of the ECML Workshop on Meta-learning: Building Automatic Advice Strategies for Model Selection and Method Combination, pp. 109–118 (2000)
36. Peng, Y., Flach, P., Brazdil, P., Soares, C.: Improved Data Set Characterisation for Meta-learning. In: Proceedings of the Fifth International Conference on Discovery Science, pp. 141–152 (2002)
37. Pfahringer, B., Bensusan, H., Giraud-Carrier, C.: Tell me who can learn and I can tell who you are: landmarking various learning algorithms. In: Langley, P. (ed.) Proceeding of the 17th International Conference on Machine Learning (ICML2000), pp. 743–750. Morgan Kaufman, San Francisco (2000)
38. Brodley, C.: Recursive automatic bias selection for classifier construction. Machine Learning 20, 63–94 (1995)
39. Berrer, H., Paterson, I., Keller, J.: Evaluation of Machine learning Algorithm Ranking Advisors. In: Proceedings of the PKDD Workshop on Data-Mining, Decision Support, Meta-Learning and ILP: Forum for Practical Problem Presentation and Prospective Solutions, pp. 1–13 (2000)
40. Soares, C., Brazdil, P.B.: Zoomed ranking: Selection of classification algorithms based on relevant performance information. In: Zighed, D.A., Komorowski, J., Żytkow, J.M. (eds.) PKDD 2000. LNCS (LNAI), vol. 1910, pp. 126–135. Springer, Heidelberg (2000)
41. Brazdil, P., Soares, C., Pinto, J.: Ranking Learning Algorithms: Using IBL and Meta-Learning on Accuracy and Time Results. Machine Learning 50, 251–277 (2003)
42. Giraud-Carrier, C.: Beyond Predictive Accuracy: What? In: Proceedings of the EMCL 1998 Workshop on Upgrading Learning to Meta-Learning: Model Selection and Data Transformation, pp. 78–85 (1998)

43. Abraham, A.: Meta-learning evolutionary artificial neural networks. Neurocomputing Journal 56, 1–38 (2004)
44. Castiello, C., Fanelli, A.M.: Hybrid strategies and meta-learning: an inquiry into the epistemology of artificial learning. Research on Computing Science 16, 153–162 (2005)
45. Castiello, C.: Meta-Learning and Neurocomputing A New Perspective for Computational Intelligence. In: Hassanien, A.E., Abraham, A., Vasilakos, A., Pedrycz, W. (eds.) Foundations of Computational Intelligence, vol. 1, Springer, Heidelberg (2009)
46. Zadeh, L.A.: Fuzzy Sets. Infom. and Contr. 8, 338–353 (1965)
47. Zadeh, L.A., Kacprzyk, J.: Computing with Words in Information. Physica-Verlag, Heidelberg (1999)
48. Jang, J.S.R., Sun, C.T.: Neuro-Fuzzy Modeling and Control. Proceedings of the IEEE 83, 378–406 (1995)
49. Kosko, B.: Neural Networks and Fuzzy Systems: a Dynamical Systems Approach. Prentice Hall, Englewood Cliffs (1991)
50. Lin, C.T., Lee, C.S.G.: Neural Fuzzy System: a Neural-Fuzzy Synergism to Intelligent Systems. Prentice-Hall, Englewood Cliffs (1996)
51. Jang, J.R.: ANFIS: adaptive-network-based fuzzy inference system. IEEE Trans. System, Man and Cybernetics 23, 665–685 (1993)
52. Sugeno, M., Kang, G.T.: Structure identification of fuzzy model. Fuzzy Sets and Systems 28, 15–33 (1988)
53. Castellano, G., Castiello, C., Fanelli, A.M., Mencar, C.: Knowledge Discovery by a Neuro-Fuzzy Modeling Framework. Fuzzy Sets and Systems 149, 187–207 (2005)
54. Castiello, C., Castellano, G., Fanelli, A.M.: MINDFUL: a framework for Meta-INDuctive neuro-FUzzy Learning. Information Sciences 178, 3253–3274 (2008)

Self-organization of Supervised Models

Pavel Kordík and Jan Černý

Department of Computer Science, FIT,
Czech Technical University, Prague, Czech Republic
kordikp@fit.cvut.cz

1 Introduction

The cornerstone of successful data mining is to choose a suitable modelling algorithm for given data. Recent results show that the best performance can be achieved by an efficient combination of models or classifiers.

The increasing popularity of combination (ensembling, blending) of diverse models has been significantly influenced by its success in various data mining competitions [8,38].

The superior generalization performance of the ensemble can be explained by the bias-variance error decomposition [31]. Well established methods such as Bagging, Boosting or Stacking have been applied to combine the most of the existing data mining algorithms. Ensembles such as Random Forests, neural network ensembles or classifier ensembles are also widely used. Ensembling often increases the plasticity of weak learners and improves the generalization of overfitted base models.

Recent classifier combination methods are summarized in [9,39]. The related problem of regression model combination has been targeted by individual studies, but no comprehensive summarization is available. In this study, we target primary algorithms that combine (aggregate) regression models.

When we combine identical models, no improvement can be achieved. Member models have to be diverse (e.g. demonstrate diverse errors) in order to get more accurate results. In this chapter, we study the role of diversity in the ensemble and try to find a balance between the diversity and the accuracy of models within ensembles.

The motivation of our research is to develop a data mining algorithm, that provides reasonable results for a large range of different data sets which can be used in our Fully Automated Knowledge Extraction (FAKE) framework [2]. For this task, we developed the Group of Adaptive Models Evolution (GAME) algorithm [35] combining several different base models within a cascade-like ensemble. In this chapter, we examine the diversity of base models within our GAME ensembling algorithm and find that the optimal parameter setting is very data dependent. Furthermore, the GAME algorithm fails to produce ensembles of reasonable quality for complex one-dimensional data sets [16]. The reason is that data distribution to base models and the combination of their results were designed primarily for multivariate noisy data. In order to get a

N. Jankowski et al. (Eds.): Meta-Learning in Computational Intelligence, SCI 358, pp. 179–223.
springerlink.com

more versatile solution and produce accurate models for data sets with different properties, we propose an alternative algorithm, SpecGen, described later in this chapter. SpecGen combines models in a hierarchical manner, utilizing well-known ensembling techniques such as Bagging, Boosting, Stacking etc.

Firstly, we evaluated individual ensembling techniques with elementary models on several benchmarking problems. Again, we found that the performance of techniques is data dependent. Based on the results, we propose an evolutionary strategy that combine ensemble methods and base models in a hierarchical tree-like structure. This structure is evolved for every data set by means of genetic programming. Evolved hierarchical ensembles demonstrate not only versatility, but also increased accuracy for all the data sets we have been experimenting with.

Before we start to deal with ensembling techniques, elementary base models and their optimization will be described.

2 Elementary Regression Models

The simplest models that cannot be further decomposed are typically elementary functions. The functions transfer input signals $x_1, x_2, ..., x_n$ to their output y, using parameters $a_0, a_1, ..., a_n$. The parameters (also called weights) are optimized during the training - typically by a gradient-based method.

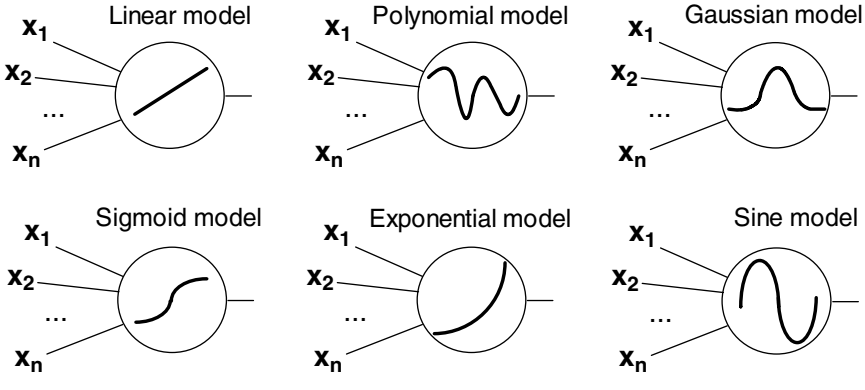


Fig. 1. Base models used in our experiments.

The base models used in our experiments are described below. In this article, we refer to base models also as neurons[1].

[1] This is not always correct, because base models do not always have a nonlinear transfer function at the output and some models are more likely an ensemble of single input neurons. However base models are elementary terminal elements within ensembles and networks and we call them neurons in this sense.

2.1 Linear Model

The linear combination is frequently used in ensemble methods, therefore we supply independent base models implementing this function.

$$y = \sum_{i=1}^{n} a_i x_i + a_0, \tag{1}$$

where n is the number of input variables, a_0 is the bias and a_i is the slope of linear function in each input dimension.

2.2 Polynomial Model

The most important parameter influencing the plasticity of the polynomial models is the maximum degree m. When this parameter equals 1, this model is equivalent to the linear model.

$$y = \sum_{i=1}^{n} \sum_{j=1}^{m} a_{ij} x_i^j + \phi \tag{2}$$

As you can see, the number of model parameters is equal to $n * m + 1$, so the maximum degree should be small for high-dimensional data.

2.3 Sigmoid Model

The sigmoid transfer function is the most popular activation function of neurons in the neural network (NN) domain. The sigmoid shape is ideal in expressing decision boundaries and it is therefore used in classifiers.

$$y = \frac{a_{n+1}}{1 + e^{-(\sum_{i=1}^{n} a_i x_i + a_0)}} + a_{n+2} \tag{3}$$

For regression purposes, we added two coefficients a_{n+1}, a_{n+2} giving the model the ability to scale and shift to data that have not been normalized.

2.4 Sine Model

The sine model can fit periodic functions well. For other data, the behaviour of the sine model will be similar to low-degree polynomials. The transfer function should be able to fit data independently in each dimension.

$$y = \sum_{i=1}^{n} a_{2n+i} \sin(a_i x_i + a_{n+i}) + a_0 \tag{4}$$

Therefore the transfer function is in fact the weighted average of one-dimensional sine functions.

2.5 Exponential Model

The exponential transfer function with scale a_{n+1} and shift a_{n+2} parameters has a limited plasticity.

$$y = a_{n+1} e^{a_{n+1}(\sum_{i=1}^{n} a_i x_i + a_0)} + a_{n+2} \tag{5}$$

2.6 Gaussian Model

The Gaussian function describes the normal distribution. Parameters adjust the shift and spread of the blob in each dimension.

$$y = e^{\sum_{i=1}^{n} -\frac{(x_i - a_i)^2}{a_{n+i}^2}} \tag{6}$$

In this version, the model can fit normalized data only. This problem can be solved by adding the shift and scale parameters (as in the Equation 5). On the other hand, these parameters increase the complexity of the parameter estimation task.

2.7 Parameter Estimation

Parameters of linear or polynomial models can be estimated directly from data in one step (linear least squares method). For example the parameters of the linear model (Equation 1) can be computed using the training data set, containing l samples in the form $x_1, x_2, \ldots, x_n; y$ (input vector; target output).

Then the model can be written as a system of linear equations:

$$\begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_l \end{bmatrix} = \begin{bmatrix} 1 & x_{11} & x_{12} & \ldots & x_{1n} \\ 1 & x_{21} & x_{22} & \ldots & x_{2n} \\ \vdots & \vdots & \vdots & & \vdots \\ 1 & x_{l1} & x_{l2} & \ldots & x_{ln} \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ \vdots \\ a_m \end{bmatrix} \tag{7}$$

The vector $\boldsymbol{a}$ of parameters can be estimated using the following matrix operations:

$$\widehat{\boldsymbol{a}} = (\mathbf{X}^T \mathbf{X})^{-1} \mathbf{X}^T Y. \tag{8}$$

For polynomial models, the matrix of inputs will be modified in the following manner: each element x_{ij} will be expanded to m elements $x_{ij}, x_{ij}^2, \ldots, x_{ij}^m$. Also, the number of parameters (length of the vector $\boldsymbol{a}$) increases to $m * n$.

To estimate parameters of other base models (sigmoid, sine, exponential and Gaussian), we use the general purpose continuous optimization algorithm described below. We also implement non-linear least squares [46] and maximum likelihood estimates [18] for individual models, and we will use this in the future.

The process of parameter optimization is depicted in Figure 4. Based on the error of the model with the initial parameters, the optimization method

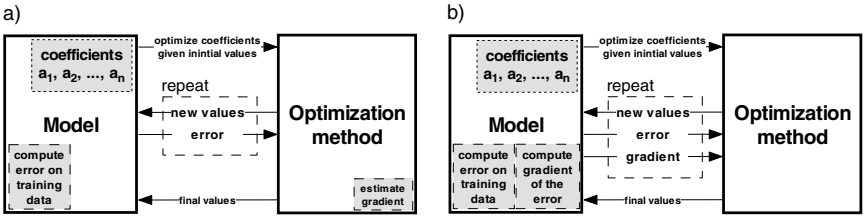


Fig. 2. Parameters $a_0, a_1, ..., a_n$ of the model can be optimized by a general purpose optimization method.

iteratively estimates new parameters and gets feedback from the model (error on the training data set).

The number of iterations (error evaluations) can be significantly reduced when global optimization methods are supplied with an analytic gradient (and possibly also Hessian) of the error function. Examples of how to compute analytic gradients of the base models can be found in [34]. In [36], we show that the most universal optimization methods for this purpose are the quasi-Newton method and the Covariance Matrix Adaptation Evolution Strategy. We also propose to combine them within a single network.

Our base models are very simple and often have a limited capacity to fit the true relationship in the data set (weak learners). Except in the case of the polynomial model with a high degree, it is not necessary to be concerned about data overfitting. On the other hand, the models are able to model only very simple relationships and their bias is high.

3 Combination of Models

3.1 Neural Networks

The combination of simple models with a sigmoid transfer function (called neurons) in a neural network dramatically reduced bias and improved the accuracy over individual models. The most popular neural network is MultiLayered Perceptron (MLP), which can be trained by means of a gradient method called back-propagation of error or using an evolutionary strategy [45].

Sigmoid models (neurons) can also be combined into a cascade network structure by means of the Cascade Correlation Algorithm [20].

The Radial Basis Function Network (RBFN) combines Gaussian models [50]. The Multilayered Iterative Algorithm (MIA GMDH) [30] or Polynomial Neural Networks (PNN) [51] combine polynomial models.

There are hundreds of possible combination techniques (learning paradigms) and many of them can model an universal function [42], when enough neurons are used. Therefore, it is necessary to prevent data overfitting (eg. by using a validation data set).

Note that a neural network would not be more efficient than a single (e.g. sigmoid) model, if it uses identical neurons. Therefore the *diversity* of neurons - in terms of their behaviour - plays a critical role in the accuracy improvement over individual neurons. Mechanisms enforcing the diversity in neural networks will be described and categorized in the next section.

3.2 Ensemble Methods

Combination (ensembling, blending) of diverse models (classifiers, regression models) is becoming main stream in data mining.

Where neural networks are algorithms combining predefined neurons into (mostly) fixed topologies in order to reduce their bias, the general ensemble methods described below can combine base models of any transfer function. Ensemble methods often improve the generalization performance of base models. The bias-variance error decomposition [31] shows that both bias and overfitting of base models are often reduced by ensemble methods. In our experiments, we use the most popular ensemble methods, a short description of which follows.

Bagging algorithm creates several training "bootstrap" subsets from the original training set. A model is created for each subset. The result for an unknown instance presented to ensemble is calculated as an average of their responses [22,10,9].

Boosting [58,9] algorithm produces a single model in the beginning. In regions where it fails to respond correctly, the instances are boosted - their weight is increased and a second model is created. The second model is more focused on instances where the first model failed. In the regions where the second model failed, weights of instances are increased again, the third model is introduced and so on. Output for an unknown instance is given by weighted average. The weight of each model is calculated from its performance (the better the model, the greater the weight).

Stacking [67,9] uses several different models trained on a single training set. The responses of these models represent meta-data which are used as inputs of a single final model which is used for calculating output for unknown instances.

Cascade Generalization [24,9] presents a sequential approach to combination of models. In Cascade Generalization a sequence of learners is created. The inputs of learner A_i consist of input base-data and also outputs of previous models A_{i-1}. Thus each model in the sequence is partially base-learner and partially meta-learner. Unknown instances pass through the sequence, and the output of the last model becomes the output of the whole ensemble.

Cascading [32,4,9] in a similar way to Boosting changes the distribution of training instances. The difference is that the distribution is changed with regard for the confidence of the previous model. Models are built in cascade. When recalling an instance, the output is produced by the first model in the cascade exceeding a confidence threshold.

Delegating [21,9] resembles Stacking, with the difference that the next models are trained on delegated instances, where previous models do not reach a minimum confidence. The number of instances in the training set decreases for the next models. For a new instance, the output is the first model with higher confidence than a given threshold.

We have also designed and implemented two *local* ensemble methods - Area Specialization and Divide Ensemble.

Area Specialization uses the same learning method as Boosting, but it can possibly use any other method. The essence of Area Specialization is in output computation for unknown instances. In short, it gives the output of that model which is best for the area where the unknown vector is. First, the distance from the unknown vector to all learning vectors is calculated and the closest N vectors are taken into the next step (N identifies the algorithm parameter called area, which determines how smooth the transitions will be between areas of output of different models). Then the best model is chosen for every learning vector selected in the first phase. Next, model weights are calculated as the difference of target value of the learning vector and the output of the best model for the unknown vector. The weight values are inverted using Gaussian function and summed up for all N learning vectors to corresponding models. Model weights are used in weighted average output.

Algorithm 1. AreaSpecialization.getOutput(unknownVector)

{Compute Euclidean distance of learning vectors to unknownVector.}
$distance[] \Leftarrow computeDistanceToLearningVectors(unknownVector)$
{Get indexes of sorted learning vectors by distance.}
$indexes \Leftarrow sort(distance)$
for $i = 0$ to $area$ **do**
  {Take i^{th} closest learning vector and find model with smallest error on that learning vector.}
  $bestModelIndex \Leftarrow getBestModelIndex(learningVectors[indexes[i]])$
  {Compute difference between target value of closest learning vector and output of model for unknownVector which is best for that learning vector.}
  $diff \Leftarrow targetOutput[indexes[i]] - model[bestModelIndex].getOutput(unknowVector)$
  {Compute model weights from how well model performs on learning vector.}
  $modelWeights[bestModelIndex] \Leftarrow +Gaussian(diff)$
end for
$modelWeights \Leftarrow normalize(modelWeights)$
$ensembleOutput \Leftarrow 0$
{Weighted average.}
for $i = 0$ to $modelsNumber$ **do**
  $ensembleOutput \Leftarrow +model[i].getOutput(uknownVector) * modelWeights[i]$
end for

Divide ensemble divides learning data into clusters (for example using k-means algorithm) and assigns one model to each cluster. Response to unknown instance is given by the model which is in charge of the cluster to which the unknown instance belongs.

There are two main advantages of this approach. Firstly, the model will probably perform better on a smaller area of learning data and has a higher chance of adapting to it. Secondly, dividing data into smaller training sets for models may cause boost in learning speed (if the learning algorithm has more than linear complexity in relation to learning vectors).

To reduce the model's unexpected behaviour near the edge of the cluster, where there are usually little or no learning vectors, we use clustering modification to enlarge all clusters by a certain amount. Function 2 describes the process. Its inputs are coordinates of cluster centroids and an array containing indexes of vectors for each cluster. Each vector is checked comparing the distance to the other centers with the distance to it's own center. If the ratio of distances is above a certain threshold, the vector is added to the cluster belonging to the other center (this means the vector can be in more than one cluster simultaneously). This feature improves model error, reduces outline values and makes transition between models (clusters) smoother.

Algorithm 2. DivideEnsemble.roughClustering(clusterCenters[], clusterIndexes[][])

```
for each vector in data do
  distance[] ⇐ computeDistanceToCenters(vector)
  {Get indexes of sorted learning vectors by distance.}
  indexes[] ⇐ sort(distance)
  closestDistance ⇐ distance[indexes[0]]
  for i = 1 to clusterCenters.length do
    currentDistance ⇐ distance[indexes[i]]
    if distance[indexes[i]] > centerDistance[indexes[0]][indexes[i]] then
      {If the distance of the vector to the other center is greater than the distance
      between centers, do not skip that vector (it means the vector is located in the
      half of the cluster that is behind its center).}
      continue
    end if
    {clusterSizeMultiplier is algorithms parameter which determines how much clus-
    ters will be resized.}
    if currentDistance/closestDistance < clusterSizeMultiplier then
      addVectorToCluster(i, vector)
    end if
  end for
end for
```

Ensembles can be used to combine any models. Base models are often collections of models themselves. Very popular are ensembles of neural networks.

3.3 Neural Network Ensembles

Research in the area of neural network ensembles [69] revealed that the generalization of neural network models (or classifiers) can be further improved by their combination.

Neural network ensemble is a learning paradigm where a collection of a finite number of neural networks is trained for the same task. It originates from Hansen and Salamons work [27], which shows that the generalization ability of a neural network system can be significantly improved through ensembling a number of neural networks, i.e., training many neural networks and then combining their predictions.

In general, a neural network ensemble is constructed in two steps, i.e., training a number of component neural networks and then combining the component outputs. As for training component neural networks, the most prevalent approaches are Bagging and Boosting [26]. Bagging is based on bootstrap sampling [28]. It generates several training sets from the original training set and then trains a component neural network from each of those training sets. Boosting generates a series of component neural networks whose training sets are determined by the performance of former ones. Training instances that are wrongly predicted by former networks will play more important roles in the training of later networks.

In [3], MLP neural network committees proved to be superior to individual MLP networks. The committee of GMDH networks generated by the Abductory Inductive Mechanism (AIM) gave better performance just when individual networks varied in complexity.

For combining the networks, several strategies can be used. The *simple ensemble* strategy computes the mean output of individual networks and *weighted ensemble* computes the weighted average. The idea of neural network stacking is to create a meta-data set containing one row for each row of the original data set. However, instead of using the original input attributes, it uses outputs of individual networks as the input attributes. The target attribute remains as in the original training set. Test instance is first evaluated by each of the base networks. Then responses are fed into a meta-level training set from which a meta-model is produced. This meta-model combines the different outputs into a final one [57].

General ensembling strategies can be applied multiple times. Some of them are designed to reduce the bias part of error, others reduce the variance part. Ensemble of ensembles can therefore combine strengths of multiple ensembling strategies.

3.4 Combination of Ensembles

Multi-boosting [66] combines ensemble learning techniques that have the capacity to effectively manage trade off between diversity and individual error. It leads to further increases in internal diversity without undue increases in individual error and results in improved accuracy. Multi-boosting combines AdaBoost, and a variant of Bagging called Wagging (weighted Bagging). It was demonstrated to attain most of Boostings superior bias reduction together with most

of Waggings superior variance reduction. Stratified gradient boosting [23] boosts stratified samples, and the experimental results indicate that Bagging increases robustness of Boosting for some data sets.

Combining Stacking and Bagging [68] also showed good results (with particular base learners on particular data sets).

The problem of these methods is that they bring improved generalization for some data sets only, and it is not clear when to use them. Basic ensemble strategies such as Bagging or Boosting should be used for overfitted base models or for weak learners. The appropriate combination of ensemble methods is the subject of a trial and error strategy.

Below, we discuss principles that may help us to build unbiased ensembles with the capacity to adapt to a particular data set.

4 Diversity and Self-organization

In this section, we give examples of how to promote *diversity* in ensembles and we discuss how ensembles can be *self-organized* in order to match required model complexity for a particular data set.

4.1 Diversity in Ensembles

As already stated above, diversity of individual models is highly important in all levels of combination. However we are far from the statement that higher diversity always means higher accuracy. The diversity can be increased by adding models with low accuracy and this will likely cause a decrease in accuracy of the combined solution. It is necessary to find a trade-off between diversity and individual error.

The diversity in the lowest level of combination can be regulated by

- dataset manipulation:
  - employed input features,
  - set of instances used for training;
- model manipulation:
  - distance of model parameters,
  - type and complexity of transfer function.

For combination of networks and ensembles, data set can be manipulated in the same way. The goal is to get a collection of networks or ensembles that exhibit diverse errors.

Randomization of decision trees [15] or GMDH MIA use a different set of input features for neurons (nodes, units).

Bagging introduces diversity by a varying set of instances used for training of individual models. In [25] bagging produces diverse models, then Principal Component Analysis (PCA) is applied to their outputs. This approach further increases the diversity and reduces noise in the network.

In [56], genetic programming is used to a design multilayered perceptron network. The article also suggests promoting diversity among individual solutions by means of the island strategy. This is consistent with [60], where fitness sharing is used to protect newly emerged diverse networks while they evolve their weights. In [13] a multiobjective evolutionary algorithm increases accuracy and diversity of individual models within an ensemble. Experimental results indicate that the evolved ensemble is more accurate than ensembles produced by strategies focused on accuracy only.

Bakker [6] clusters models in the ensemble and selects only a representative subset - which performs better than if all models are used. Here clustering allows increasing diversity in the ensemble.

The DECORATE algorithm [47] constructs artificial data samples within estimated data distribution in order to increase diversity in the ensemble. Experiments showed that DECORATE outperforms both Bagging and Boosting on a wide range of real world data sets.

A very efficient strategy is to enforce diversity of base models during training. Negative correlation network ensemble learning algorithm [43] uses a penalty term that biases the error of individual base models. The produced models are diverse and biased towards an optimal model, however their combination is close to the optima. Several later studies [12,29,19] demonstrated the usefulness of NC learning method for active promotion of diversity in ensembles.

The important question is how the diversity can be measured in the ensemble. Various diversity measures for classifiers can be found in [40,65,11]. Some of them can be adopted for regression ensembles.

Many articles [41] studied the connection of diversity and accuracy in the ensemble. In [41], several diversity measures were evaluated and found to be very similar. Experiments showed that there is no clear relationship between diversity and the averaged individual accuracy in the ensemble. The general motivation for designing diverse classifiers is correct but the problem of measuring this diversity, and thus using it effectively for building better ensembles, is still to be solved.

A very efficient and straightforward solution is to train individual models using varied architectures, instances and features in order to obtain implicit diversity.

4.2 Self-organization

Self-organization [49] is another important principle that can help us to build unbiased models of diverse data sets. The self-organized model, in our sense, is able to adapt itself to a data set complexity without any external information. Self-organizing models are constructed inductively from data. The induction means gathering small pieces of information, combining it and using already collected information in the higher abstraction level to get a complex overview of the studied object or process.

The construction process often starts from a minimal form and the model grows until a system complexity is matched.

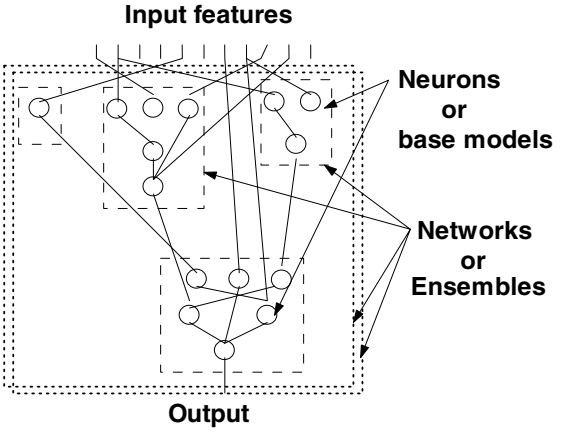


Fig. 3. An example of a self-organized model decomposing the problem and using diverse subsolutions to get a global unbiased model.

A problem is decomposed into small subproblems; combining subsolutions we get higher level solution and so on. A modelling problem can be decomposed into subproblems by scaling:

- *Input features* - the number of input features used by the model can be limited
- *Instances* - the subset of instances used by the model can be limited
- *Transfer function complexity* - complexity of transfer function can be limited
- *Parameters maximum* - the maximal values of parameters can be limited (weight decay)
- *Acceptable error* - parameter optimization can be stopped prematurely.

Figure 3 shows a hierarchical model consisting of neurons, networks and ensembles. It decomposes the problem by scaling the number of input features (maximum 1 input for neurons in the first layer of the network). Also complexity of the transfer function increases with increasing number of layer. The subset of instances used for training can be increased with the network layer. Acceptable accuracy can be varied within the ensemble layers. There are many more ways of decomposing the problem and building a self-organized model. The question is whether it is more efficient to scale all items (from the list above) at once and finish with a single network, or to scale them one-by-one finishing e.g. with an ensemble of ensembles of networks of models.

The following section describes how a self-organized network of models can be built in order to respect the principles mentioned above. We present two

different methods - the first combines models in a feedforward layered ensemble, the second combines models hierarchically using several ensembling strategies.

5 Group of Adaptive Models Evolution

The first method, called Group of Adaptive Models Evolution (GAME), was originally proposed in [34]. It combines base models in a feedforward layered manner. It uses many principles from the last chapter. It starts from a minimal form and scales the number of input features and transfer function complexity of models. At the same time it promotes the diversity of base models by bootstrapping training data, using different subsets of input features for base models that differ also in the transfer function employed.

5.1 The Pseudocode of the Algorithm

The GAME algorithm is a supervised method. For training, it requires a data set with input variables (features) and the output variable (target). The GAME algorithm, in summary, is described below:

1. Separate the validation set from the training data set (50% random subset)
2. Initialize first population of base models – input connections, transfer functions and optimization methods chosen randomly
3. Optimize parameters of models' transfer functions by assigned optimization method – error and gradients computed on the training data set
4. Compute fitness of models by summarizing their errors on the validation set
5. Select parents, apply crossover and mutation operators to generate population of offspring
6. Merge populations of parents and offspring (methods based on fitness and distance of models)
7. Go to 3), until diversity level is too low or the maximum number of epochs (generations) is reached
8. Select the best model from each niche – based on fitness and distance, freeze them to make up the layer and delete the remaining neurons
9. Until the validation error of the best model is significantly lower than the best model from the previous layer, proceed with the next layer and go to 2)
10. Mark the model with the lowest validation error as the output of the network and delete all models not connected to the output.

5.2 An Example of the GAME Algorithm on the Housing Data Set

We will demonstrate our algorithm on the Housing data set that can be obtained from the UCI repository [1]. The data set has 12 continuous input variables and one continuous output variable. In Fig. 4 you can find the description of the most important variables.

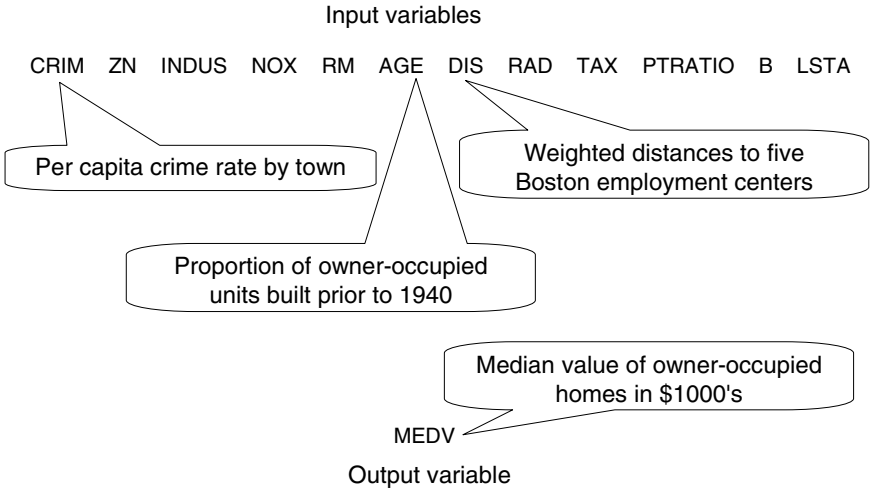


Fig. 4. Housing data set: the description of the most important variables.

Firstly, we split the data set into a subset used for training (A+B) and a test set to get an unbiased estimate of the model's error (see Fig. 22). Alternatively, we can perform k-fold crossvalidation [33].

Then we run the GAME algorithm, which separates out the validation set (B) for the fitness computation and the training set (A) for the optimization of parameters.

In the second step, the GAME algorithm initializes the population of neurons (the default number is 15) in the first layer. For instant GAME models, the preferable option is *growing complexity* (number of input connections is limited to index of layer). Under this scheme, neurons in the first layer cannot have more than one input connection, as shown in Fig. 6. The type of the transfer function is assigned randomly to neurons, together with the type of method to be used to optimize the parameters of transfer function.

The type of transfer function can be sigmoid, Gaussian, linear, exponential, sine and many others (a complete, up-to-date list of implemented transfer functions is available in the FAKEGAME application [2]), see Fig. 7. If the sigmoid transfer function is assigned to a neuron, the output of this neuron can be computed, for example as MEDV=1/(1-exp(-$\mathbf{a}_1$*CRIM+ $\mathbf{a}_0$)), where parameters $\mathbf{a}_0$ and $\mathbf{a}_1$ are to be determined.

To determine these parameters, an external optimization method is used. The optimization method is chosen from a list of available methods (Quasi-Newton, Differential Evolution, PSO, ACO etc.). Note that linear and polynomial neurons are optimized by the least squares method by default as explained in Section 2. For other neurons, analytic gradient of the error surface is supplied to enhance the convergence of general optimization methods.

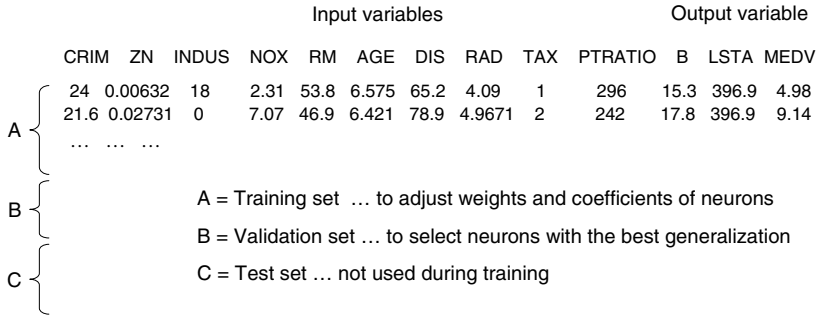


Fig. 5. Splitting the data set into the training and the test set; the validation set is separated from the training set automatically during GAME training.

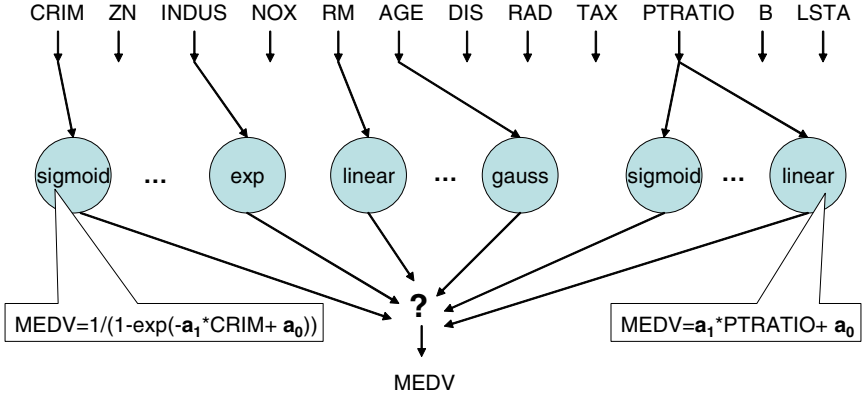


Fig. 6. Initial population of neurons in the first GAME layer.

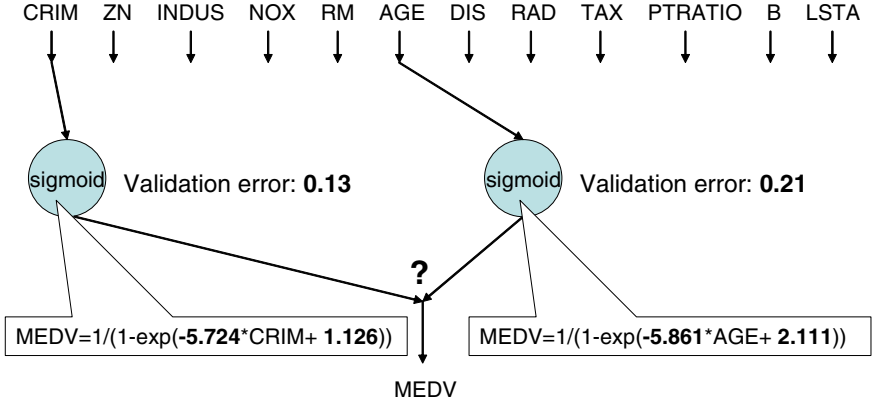


Fig. 7. Two individuals from different niches with parameters optimized on set A and validated on set B. The neuron connected to the AGE feature has a much higher validation error than neurons connected to CRIM and survives thanks to *niching*.

The fitness of each individual (neuron) is computed as the inverse of its validation error. The genetic algorithm performs selection, recombination and mutation and the next population is initialized. After several epochs, the genetic algorithm is stopped and the best neurons from individual niches are frozen in the first layer (Fig. 8).

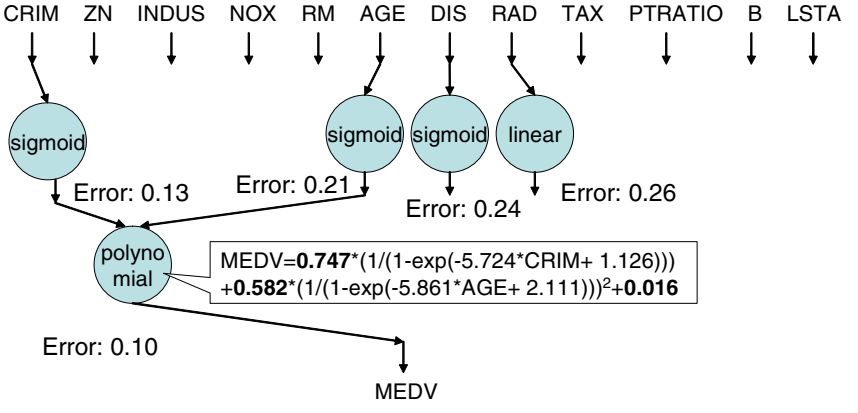


Fig. 8. In our example, the best individual evolved in the second layer combines the outputs of neurons frozen in the first layer (feature detectors).

Then the GAME algorithm proceeds with the second layer. Again, an initial population is generated with random chromosomes, evolved by means of the niching genetic algorithm, and then the *best and diverse* neurons are selected to be frozen in the second layer.

The algorithm creates layer by layer, until the validation error of the best individual decreases significantly. Fig. 9 shows the final model of the MEDV variable.

5.3 Genetic Algorithm Optimizing Connections of Base Models

In this section, we analyze the internal behaviour of the GAME algorithm in terms of diversity and accuracy of neurons. We start with the genetic algorithm employed to evolve individual neurons or base models.

The genetic algorithm is frequently used to optimize a topology of neural networks [45,59,61]. Also in GMDH related research, recent papers [52] report improving the accuracy of models by employing the genetic search to look for their optimal structure.

In the GAME algorithm, we also use genetic search to optimize the topology of models and also the parameters and shapes of transfer functions within their models. The individual in the genetic algorithm represents one particular neuron (or model) of the GAME network. Inputs of a neuron are encoded into a binary

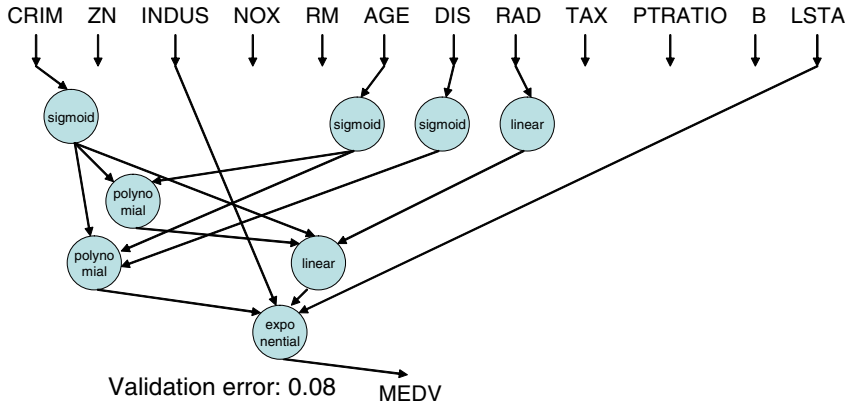


Fig. 9. The GAME model of the MEDV variable is finished when new layers do not decrease the validation error significantly.

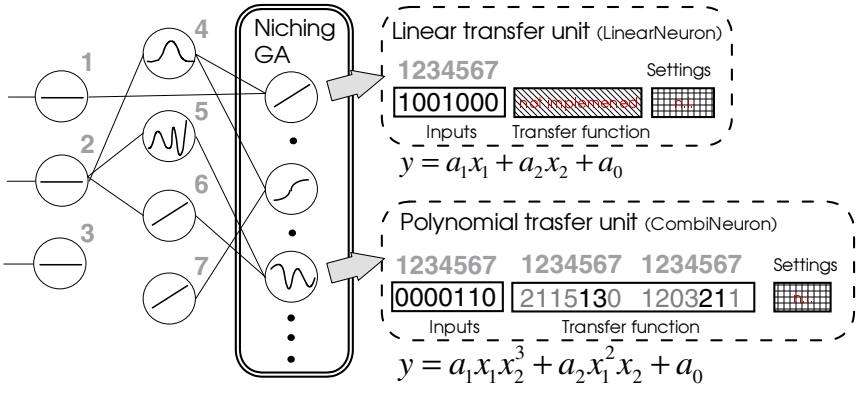


Fig. 10. If two models of different types are crossed, just the "Inputs" part of the chromosome come into play. If two Polynomial models cross over, also the second part of the chromosome is involved.

string chromosome. The transfer function can be also encoded into the chromosome (see Figure 10). The chromosome can also include configuration options such as strategies utilized during optimization of parameters. The length of the "inputs" part of the chromosome equals to the number of input variables plus the number of models from previous layers, particular neuron can be connected to. The existing connection is represented by "1" in the corresponding gene.

Note that parameters of the transfer functions $(a_0, a_1, \cdots, a_n)$ are not encoded in the chromosome (Figure 10. These parameters are set separately by optimization methods. This is crucial difference from the Topology and Weight Evolving Artificial Neural Network (TWEANN) approach [61]. The fitness of the individual (e.g. f(p1)) is inversely proportional to the error on the validation set. The application of the genetic algorithm in the GAME algorithm is depicted in the Figure 11. The left schema describes the process of single GAME layer evolution when the standard genetic algorithm is applied.

Models randomly initialized and encoded into chromosomes. Then the genetic algorithm is run. After several epochs of the evolution, individuals with the highest fitness (models connected to the most significant input) dominate the population. The best solution represented by the best individual is found. The other individuals (models) have very similar or the same chromosomes as the winning individual. This is also the reason why all models surviving in the population (after several epochs of evolution by the regular genetic algorithm) are highly correlated. The regular genetic algorithm found one best solution. We want to find also multiple suboptimal solutions (e.g. models connected to the second and the third most important input). By using less significant features we can further get more additional information than by using several best individuals connected to the most significant feature, which are in fact highly correlated

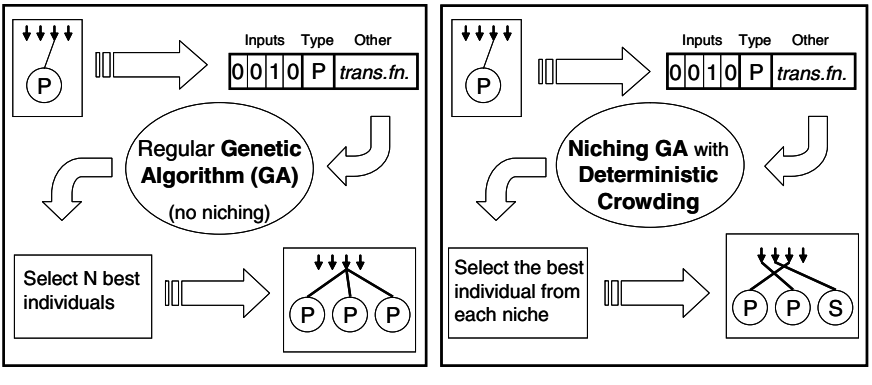


Fig. 11. GAME models in the first layer are encoded into chromosomes, then GA is applied to evolve the best performing models. After few epochs all models will be connected to the most significant input and therefore correlated. When the Niching GA is used instead of the basic variant of GA, models connected to different inputs can survive.

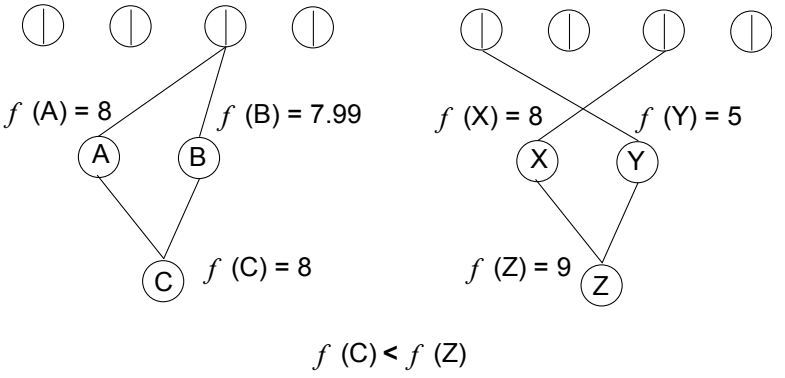


Fig. 12. Fitness of the model Z is higher than that of model C, although Z has less fit inputs.

(as shown on Figure 12.). Therefore we employ a niching method described below. It maintains diversity in the population and therefore models connected to less significant inputs are allowed to survive, too (see Figure 11 right).

5.4 Niching Methods

The major difference between the regular genetic algorithm and a niching genetic algorithm is that in the niching GA the distance among individuals is defined. The distance of two individuals can be based on the phenotypic or genotypic difference of models. In the GAME algorithm, the distance of models is computed from both differences. Figure 13 shows that the distance of models is partly

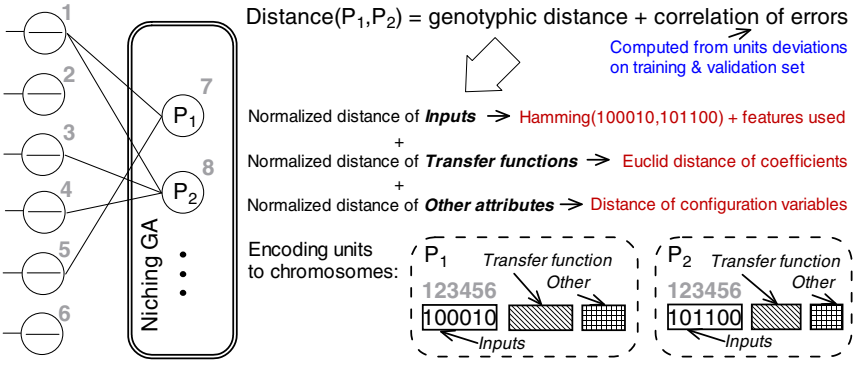


Fig. 13. The distance of two models in the GAME algorithm.

computed from the correlation of their errors and partly from their genotypic difference. The genotypic difference consists the obligatory part "difference in inputs", then some models add "difference in transfer functions" and also "difference in configurations" can be defined.

Models that survive in layers of GAME networks are chosen according to the following algorithm. A model with the lowest RMS error is selected to survive. Remaining models are sorted according to their RMSE and distance from already selected models[2]. Surviving models should have low RMS errors, high mutual distances and low correlations of errors.

Niches in GAME are formed by models with similar inputs, similar transfer functions, similar configurations and high correlation of errors.

We have experimented with the following niching genetic algorithms:

- Deterministic crowding
- Fitness sharing
- Clearing
- Probabilistic Crowding

The Deterministic Crowding (DC) [44] assumes that parents and their offsprings are similar. A parent and its less distant offspring compete to be present in the next generation. A model with higher fitness (better generalization) wins the competition.

In Fitness Sharing (FS) [44] similar individuals have to share fitness. It means, that models are penalized for being similar. The problem is to estimate the right penalty factor for given problem.

Clearing [54] preserves just the elite individual for each niche. Fitness of individuals similar (more than a threshold) to elite individuals become zero. Again, the threshold is data dependent.

Probabilistic Crowding (PC) [48] improves the DC by probabilistic selection of the winner. A model with higher fitness is more likely to be present in the next generation.

In order to decide which niching scheme is the most appropriate, one has to analyze the convergence of the genetic algorithm and monitor the diversity development.

5.5 Monitoring the Development of Diversity

At first, we have analyzed the behavior of the Deterministic Crowding (DC) algorithm. In the Figure 14, we compared the DC algorithm with the regular genetic algorithm (GA), where the distance is not taken into account. The data set used to model the output variable (Mandarin tree water consumption) has eleven input features. Models in the first layer of the GAME network are connected to a single feature. The population of 200 models in the first layer was

[2] Our experimental results shows that a reasonable ratio is 80% weight on inverted RMSE and 20% weight on distance, but the ratio is very data dependent

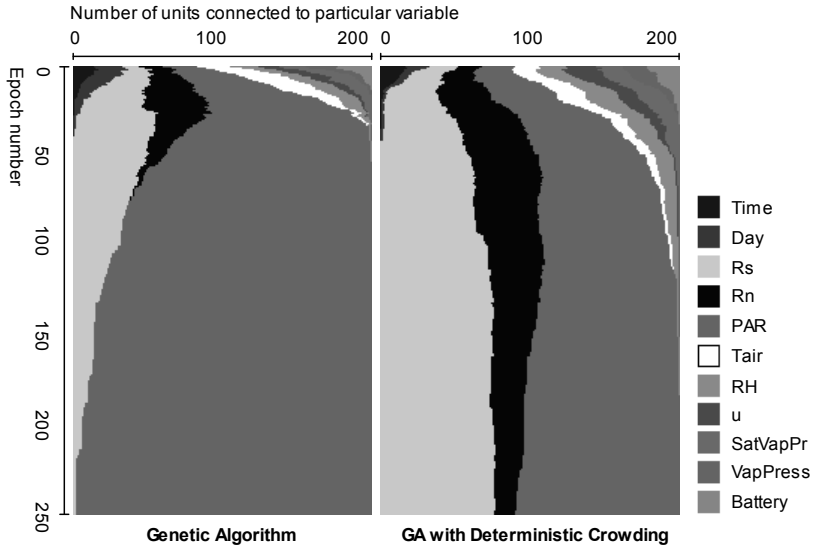


Fig. 14. The experiment shows that the regular Genetic Algorithm approaches the optimum relatively quickly. Niching preserves different models for many more iterations so we can chose the best model from each niche at the end. Niching also reduces possible premature convergence.

initialized randomly (genes are uniformly distributed - approx. the same number of models connected to each feature). After 250 epochs of the regular genetic algorithm the fittest individuals (models connected to the most significant feature) dominated the population. On the other hand the niching GA with DC maintained diversity in the population. Individuals of three niches survived. As Figure 14 shows, the difference between the niching genetic algorithm and the algorithm without the distance based selection is clear. The results for all other niching methods can be found in [53].

The number of individuals (models) in each niche is proportional to the significance of the feature, models are connected to. From each niche the fittest individual is selected and the construction goes on with the next layer. The fittest individuals in next layers of the GAME network are these connected to features which brings the maximum of additional information. Individuals connected to features that are significant, but highly correlated with features already used, will not survive. By monitoring which individuals endured in the population we can estimate the significance of each feature for the output variable modelling. This can be subsequently used for the feature ranking [55].

5.6 Monitoring the Diversity of Models

Our software enables the visual inspection of complex processes that are normally hard to control. One of these processes is the development of models diversity

during their evolution and it is displayed in the Figure 15. From left we can see the matrix of genotypic distances computed from chromosomes of individual models during the evolution of the GAME layer. Note that this distance is computed as a sum of three components: distance of inputs, transfer functions and configuration variables, where last two components are optional. The darker color of background signifies the higher distance of corresponding individuals and vice versa. The next matrix visualize distances of models based on the correlation of their errors. Darker background signifies less correlated errors. The next graph shows deviations of models output from the target value of individual training vectors. From these distances the correlation is computed. The most right graph of the Figure 15 shows the normalized RMS error of models on the training data.

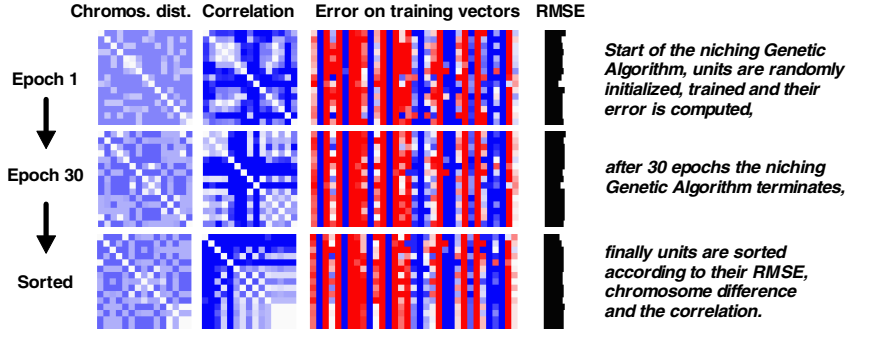


Fig. 15. During the GAME layer evolution, distances of models can be visually inspected. The first graph shows their distance based on the genotypic difference. The second graph derives distance from their correlation. Third graph shows deviations of models on individual training vectors and the most right graph displays their RMS error on the training data.

Simple visualization of distance matrices is not efficient for more models. Also clusters of similar models are not visible in the matrix visualization.

We used a projection technique [17] to map genotypic distances and correlation matrices into two dimensional scatterplots.

In the left part of Figure 16, models from the first layer are allowed to have only one input. We modelled the E-coli data set that has 8 input features therefore 8 clusters can be observed. Figure 16 shows ninth generation of the genetic algorithm and we can observe that numbers of models in the clusters are not equal. Models connected to insignificant input features have lower fitness than models connected to important features. Some models connected to the most significant feature are trained better (black crosses) than the other models (red crosses). The projection of the correlation matrix shows that there are only 5 clusters of models having similar outputs on the training data. It suggests that just 4 input features are relevant for the classification of the imU class (one class found in E-coli database). Again, we can observe the cluster with a

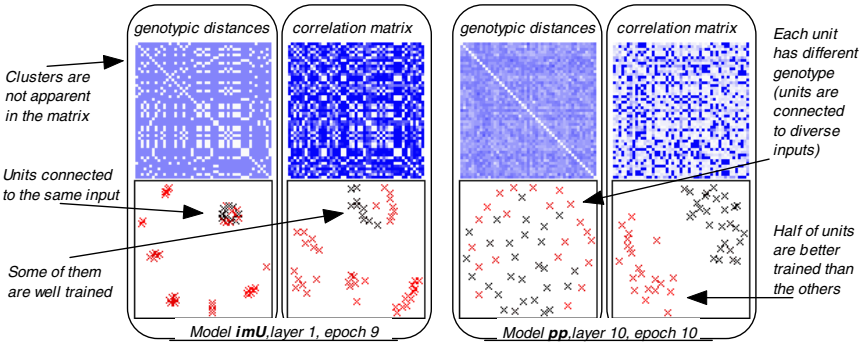


Fig. 16. For 50 models in the population, the direct visualization of the genotypic distances matrix and the correlation matrix are not informative any more (as shown on the top of this picture). The better way is to use projection into two dimensional space.

half models trained better than the other half. The right part of the Figure 16 shows genotypic distances and a correlation matrix of models in 10th layer of the GAME network. Here, number of possible inputs is very high and the probability that two models have the same set of inputs is low. As we can see from both the matrix of genotypic distances and from the projection of this matrix, models have diverse chromosomes and uniformly distributed in the space (no apparent clusters). The surprising information is carried by the correlation matrix and its projection. Again, one half of models have better fitness than the other half. The reason for this behaviour we found in the configuration of the GAME method. One half of models had linear transfer function whereas the second half of models were initialized with the sigmoid transfer function. The evolution of the transfer function was not used therefore many newly generated models were assigned the linear transfer function, that is not effective for the classification of the PP class. Visualization techniques helped us to find and remove this inefficiency.

All projections are updated as the evolution proceeds from epoch to epoch. Using this visual inspection tool, we have evaluated and tuned the distance computation used by niching genetic algorithms.

5.7 Experimental Evaluation of Niching Methods

The next goal was to evaluate if the distance metric is well defined. The results in Figures 17, 18 and 19 show that the difference between various methods of distance computation is not very significant. It is not only data set dependent, but also algorithm dependent. For some data, we got worse performance, when the distance of models is taken in account ("genetic" algorithm is superior). For some data, the best performance is achieved by the phenotypic distance ("pheno") only - it means that the distance of two models is computed as

difference of their errors. Genotypic ("geno") distance alone is seldom better than the phenotypic and combined ("combi") distance.

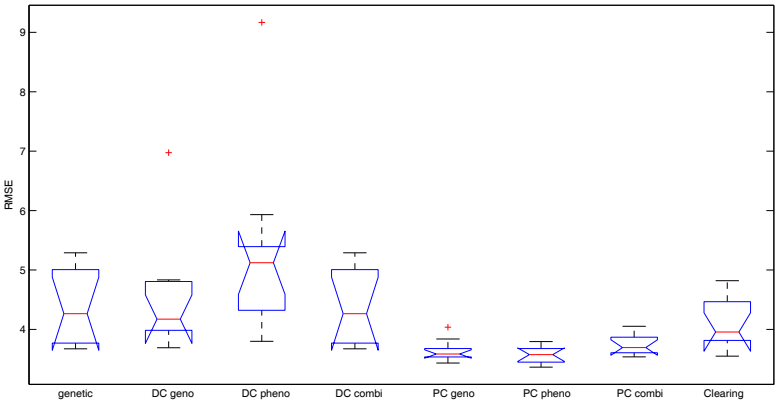


Fig. 17. The error of GAME networks with different niching algorithms optimizing connections of base models on Boston housing data.

To demonstrate the data and algorithm dependency, we include comparison of the Genetic algorithm, Deterministic (DC) and Probabilistic (PC) crowding and Clearing applied to optimize models in GAME network modelling the Boston housing data set (Figure 17). In this case, the Probabilistic crowding algorithm is superior to other algorithms in all its three variants (genotypic, phenotypic distance and their combination). For the Horse colic classification problem, one GAME network is build for each class, separating it from others. The final class is determined as the network with the highest activation. Again (see Figure 18), the models optimized by the probabilistic crowding shows slightly better performance, than other models.

We run exhaustive experiments, where we compared the performance of the GAME networks using the above described niching techniques.

Each boxplot in experiments is computed from 10 values - results of ten-fold crossvalidation of models. Therefore for regression tasks, 100 models has been built for each boxplot. For classification data, this number has to be multiplied by number of classes.

In Figure 19, the Deterministic crowding with combined distance was compared to the Genetic algorithm (selecting models to new generation based on their fitness) and to the Random selection algorithm (selecting models randomly). In this case, each boxplot was generated from 150 models (15 times 10fold CV). As you see, the results are still not very significant. For Horse colic

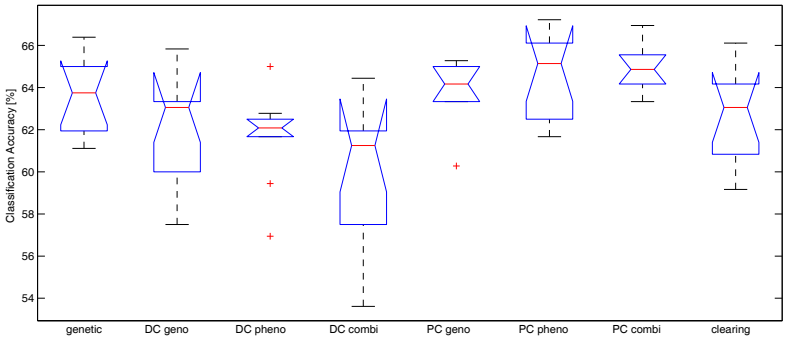


Fig. 18. Performance of GAME networks with different niching algorithms optimizing connections of base models on Horse colic data.

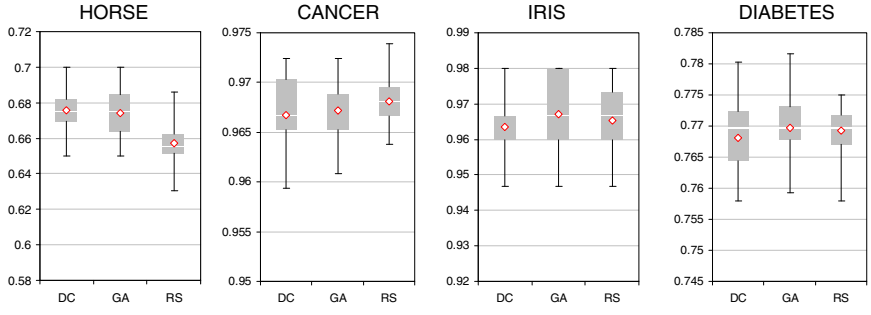


Fig. 19. The classification accuracy of GAME models on several data sets from UCI repository. Deterministic Crowding (DC), Genetic Algorithm (GA) and Random Selection (RS) was used for evolution of models in layers.

data set, the DC seems to evolve best models and RE is significantly worse. However for the Breast Cancer Wisconsin data set it is just the opposite.

By default, we recommend to use the Probabilistic Crowding strategy with the combination of genotypic and phenotypic distances.

5.8 Benchmarking GAME Algorithm

In [37] we compare the regression and classification performance of the GAME method against the performance of methods implemented in the Weka machine learning environment. For the A-EGM data set described in [37], the GAME algorithm outperformed well established methods in both classification and regression accuracy. It outperformed not only following algorithms: Linear regression

with embedded feature selection algorithm, MLP, RBFN, J48 decision trees, but also their ensembles produced by bagging and boosting.

However, although we got significant results, the comparison was conducted for single data set only and cannot be generalized. Also, for benchmarking purposes, the ensembling algorithms and the GAME algorithm should use the same base models. Recently, we have implemented all described ensembling algorithms into our FAKE GAME environment and made such experiments possible.

In the next section we present results of ensembles preformed in our new environment [fakegame].

6 Experimental Results of Ensemble Methods

At first, we look for optimal parameter settings of implemented methods, then we examine the performance of ensemble methods with elementary base models. Finally, we experiment with simple polynomial models only. The advantage is that the plasticity can be easily adjusted for polynomial models and their learning is very fast (see Section 2).

6.1 Ensemble Methods Parameters Setting

The parameter common to all ensemble methods is the number of base models. Clearly, the optimal setting of this parameter is data dependent.

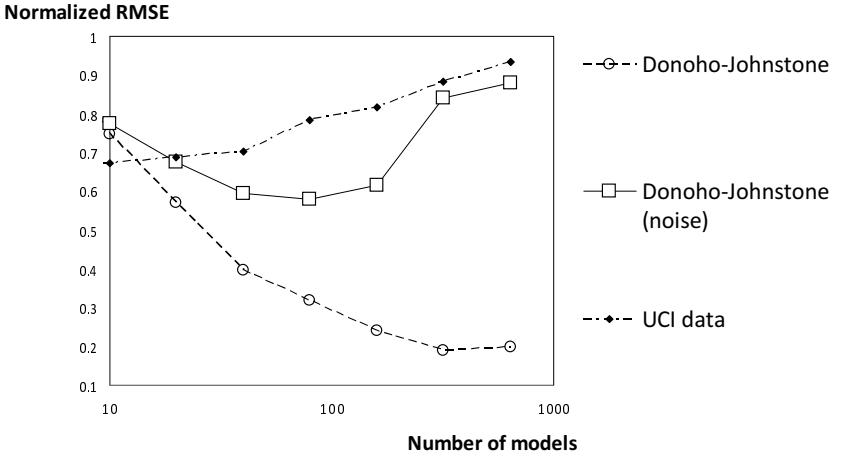


Fig. 20. The normalized error of the AS ensemble based on the number of gaussian base models.

Figure 20 shows the performance of the Area specialization ensemble for increasing number of gaussian base models. We averaged the results for all signals from the Donojo-Johnstone benchmarks (for none and medium noise extra) and for several data sets from the UCI repository (Heart, Flare, Buildings, Bosthouse). For the artificial complex signals without any noise, an increasing number of base models leads to the better accuracy. For the same signal with noise added, the overfitting leads to poor results for AS ensembles with more than hundred models (in average). Averaged results on UCI repository indicate that for chosen data sets the number of base model needed is much smaller.

Ensemble methods can have also specific parameters, such as the threshold parameter in Boosting RT. This parameter influences specialization of base models within the ensemble. For each data set, we identified its value maximizing the generalization performance of generated ensemble models. As you see in the Table 1, the universal optimal setting of the parameter does not exists. Each data set requires different threshold for model specialization. For comprehensive results regarding parameter setting of ensemble methods please refer to [64].

Table 1. For each data set, the optimal threshold of the Boosting RT ensemble can be different.

Threshold	Best performing data sets for given threshold
0	bosthouse, buildings 2
0.05	heavysine, buildings 3, flare 3
0.1	doppler, flare 2, heart
0.2	buildings 1
0.4	block
0.5	bump, flare 1

6.2 Summarized Results of Ensembles with Elementary Base Models

In this section, we present results of implemented ensemble methods with the base models described in the Section 2. We used the following benchmarking data sets:

- *Donoho-Johnstone*: Block, Bump, Doppler, Heavysine
- *Proben1*:
  - Regression: Buildings
  - Classification: Glass, Card, Cancer, Diabetes
- *UCI Machine Learning Repository*:
  - Regression: Heart, Flare
  - Classification: Iris
- *Statlib*: Bosthouse
- *Others, see [34]*: Mandarin, Antro

Results of experiments with ensembles of elementary regression models are summarized in the Table 2. We have experimented with ensembles listed in the Table 3, using all possible base models on all data sets above. Each ensembling algorithm contained 10 base models of uniform type. The type with lowest testing error is appended to the RMSE value in each cell of the Table 2.

Table 2. In each cell, there is the best performing base model type and achieved testing error for given data set. Best performing ensembles are in bold.

	RMSE, type of best performing base models						
Data\Model	**AS**	**BAG**	**BST**	**CG**	**DIV**	**ST**	**base**
Block	**1.21 G**	6.6 G	6 G	4.02 SI	3.23 SG	5.76 SI,SG	6.6 P
Bump	**1.39 P**	6.69 G	6.44 G	6.7 SG	4.8 G	6.82 G,P	6.85 G
Doppler	**1.24 G**	6.25 G	6.55 SG	5.24 SI	3.21 G	5.94 SG,SI	6.32 G
Heavysine	0.52 G	5.1 G	4.6 G	2.19 SG	**0.4 SI**	3.35 SI,SG	5.1 G
Block noise	**2.88 L**	6.92 G	6.02 G	4.9 SI	4.08 P	6.56 SI,SG	5.1 G
Bump n.	**3.07 SG**	7.14 SI	11.5 G	7.04 SG	5.41 G	7.04 G,P	7.12 G
Doppler n.	**2.87 G**	6.67 G	6.84 G	5.75 SI	4.02 P	6.37 SG,SI	6.68 G
Heavysine n.	2.51 G	5.71 SG	4.67 G	3.39 SI	**2.29 P**	4.37 SI,SG	5.75 SG
Bosthouse	3.59 SI	3.33 SG	3.47 SG	3.34 SG	4.03 SI	3.42 SG,P	**3.31 SG**
Buildings 1	**115.4 G**	121.4 P	121.3 P	119.6 SG	118.4 SG	120.8 SI	121.2 P
Buildings 2	**0.517 SI**	0.526 SI	0.527 P	0.524 SI	0.517 SI	0.52 SI	0.525 P
Buildings 3	**0.597 P**	0.624 SG	0.623 SG	0.613 SI	0.604 P	0.616 SI,SG	0.63 P
Flare 1	0.103 G	0.095 SI	0.095 G	0.095 SI	0.0976 SI	0.094 SI,SG	**0.094 SI**
Flare 2	0.034 L	0.0325 P	**0.0318 L**	0.0325 G	0.034 L	0.033 G,P	0.0327 SI
Flare 3	0.025 L	0.026 G	**0.024 L**	0.026 L	0.026 L	0.028 G,P	0.026 L
Heart	0.218 G	0.202 SG	**0.196 SG**	0.205 SI	0.237 SI	0.206 E,P	0.207 SI
Mandarin	0.102 E	0.102 SI	0.124 SI	0.103 SI	0.104 SI	**0.097 E,P**	0.103 SI
Antro age	12.83 SI	11.78 SI	**11.74 SI**	11.84 E	12.99 SG	11.77 SI	11.86 SI

For some data sets (Bosthouse, Flare 1), ensembling does not bring improved accuracy, because the input-output relationship is quite simple and can be expressed by a single function (sigmoid, sine). The Area Specialization ensemble has reasonable accuracy for several data set, but poor accuracy for data sets with simple relationships (Antro, Bosthouse). The results indicate, that the performance of ensemble algorithms is data dependent. Also, the type of the best base model differs across data sets, as expected.

6.3 Experiments with Models of Increasing Plasticity

We can use the parameter *degree* of polynomial base model to change the plasticity of the base model. In this section, we are exploring the effect of ensembling on base models with increasing plasticity.

The methodology was the following. Five base models was used in ensembles, ten fold cross validation was repeated three times. Each value in the graph,

is therefore averaged over 30 measurements (10 CV * 3 repetitions). Most of the ensembles were explained in the Section 3.2 and their abbreviations are summarized in the Table 3. The **P** is the performance of single polynomial model with given maximum degree. The **ST[P,L]** states for polynomial models stacking using the linear combination. The **EVOL[p]** is a simple ensemble evolving inputs of 5 base models for 8 generations. Up to three models (niche leaders - see Section 5.4) are selected form the final population based on their fitness. The output of the ensemble is their average, however in most cases, only single model is selected, because the population is too small to maintain stable niches. The **GAME[p]** algorithm in the following experiments is limited to 5 layers maximum, in each layer only one base model survives (after 8 generations of evolution), therefore the final ensemble can have up to 5 models (connected in cascade-like style). The **AS[DIV[P]]** ensemble is a hierarchical combination of two ensemble methods and will be explained later.

Table 3. Abbreviations of ensembles and models, their parameters.

Ensemble	Parameters
	area Area size
AS AreaSpecialization	spec Models spec.
BAG Bagging	
BST Boosting	tr Threshold
CG Cascade Generalization	
DIV Divide	mult Cluster size multiplier
ST Stacking	
Models	**Parameters**
E Exponential	
G Gaussian	
L Linear	
P Polynomial	degree Maximal degree
SG Sigmoid	
SI Sine	

High degree polynomials can highly overfit the data, making the resulting error extremely high. Therefore we have rounded maximal error to three times the standard deviation of error for a few cases, that were observed.

For linear base models ($deg = 1$ in Figure 21) most of the ensembles have the same error as the single polynomial model. The reason is obvious - linear combination of linear models is still a linear model. Two local ensembles (DIV and AS[DIV]) with lower error use specialization and reduce bias by segmenting the linear output function. The performance of local ensembles worsens with increasing polynomial degree of base models.

The relationship in this data set can be sufficiently modelled by a single polynomial function of degree four. However even polynomials with the degree

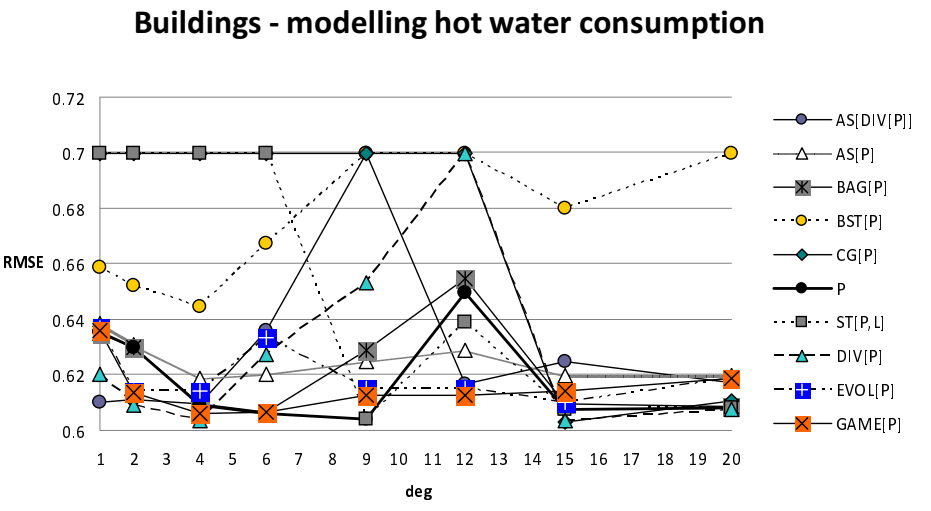


Fig. 21. Error of individual ensembles for base models with increasing polynomial degree on the medium noise Donoho-Johnstone time series (results averaged).

20 do not overfit the data and therefore the reduction of variance part of the error by ensemble methods is not observed.

The Boosting significantly worsens the accuracy of individual models. The behaviour of CG method is rather strange and we have to analyze it in more details. For higher number of polynomial degree (15+) the variance is reduced, but for less plastic models, the error is much higher than that of the single base model.

The graph on the Figure 22 shows the performance of ensembles on a data set with significantly different properties. Also, the results are significantly different.

Bagging, Boosting and Stacking have similar performance as single polynomial model and it is not worth to use them for this data set. The performance of the Area Specialization ensemble improves with increasing degree of polynomial. For the Divide ensemble, the optimal degree of base models for this data set is 4. For models with higher plasticity, the accuracy decreases. We found that the hierarchical combination [3] of the AS and DIV ensemble methods demonstrates superior results on this data set. In total, there are 25 polynomial models in this hierarchical ensemble. We compared the performance of this ensemble to AS and DIV ensembles with 25 base models and the hierarchical combination was still significantly more accurate.

[3] There are 5 models in the AS ensemble and each base model consists of the DIV ensemble of 5 polynomial models.

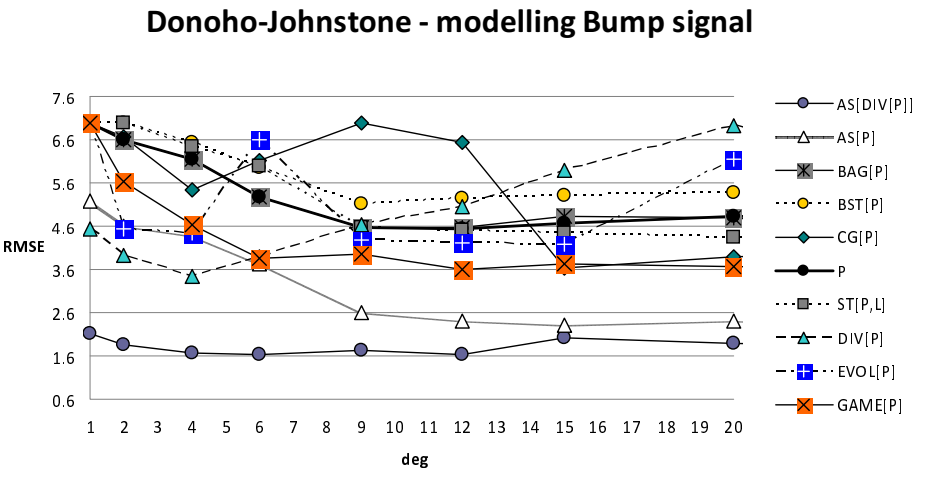


Fig. 22. Error of individual ensembles for base models with increasing plasticity on the Heart dataset from the UCI machine learning repository [1].

For both data set, the GAME algorithm performs reasonably well with all tested degrees of polynomial models. However we have experimented with many more data sets and in some cases, the performance of GAME ensembles was worse.

The conclusion from these experiments is that the performance of ensembles is data and base model dependent. We found a hierarchical combination of ensemble methods that significantly outperformed all other methods for the Donojo-Johnstone benchmarks. However this particular combination performs worse, when applied to the data set with different properties (see Figure 21).

7 Specialization-Generalization Ensemble (SpecGen)

Recently, aggregation or hierarchical combination of ensembles has also been studied [5,14,62]. In particular, gradient boosting [23] and multi-level stacking of neural networks [7] were part of the winning solution in the Netflix competition [63].

There are many more ways in which ensembles can be efficiently combined (or aggregated) into a hierarchical (tree-like) structure. Our experiments in the previous section show that the structure in which models are combined is highly data dependent. In this section, we propose to evolve such structures in order to adapt the model to the complexity of a data set.

Figure 23 shows an example of ensemble combination in a hierarchical way. Here, the full training data set is passed to a top level bagging that generates 4 bootstrap training data sets for members of the ensemble. The second bootstrap

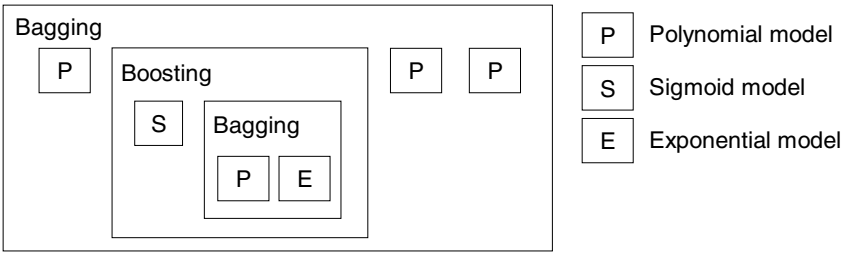


Fig. 23. An example of hierarchical combination of ensembles. Using this template, a model can be produced (see Figure 24).

training data set is used by boosting to train a sigmoid model and samples where the sigmoid model demonstrates high error are more likely to be used in the training set for the second member model of the boosting: the bagging of polynomial and exponential models. The resulting model is depicted in Figure 24. Input attributes are presented to leaf nodes of the hierarchical ensemble. The outputs of these base models are combined to produce the final output.

In general, the leaf nodes of the tree are the base models with elementary transfer function described in Section 2.

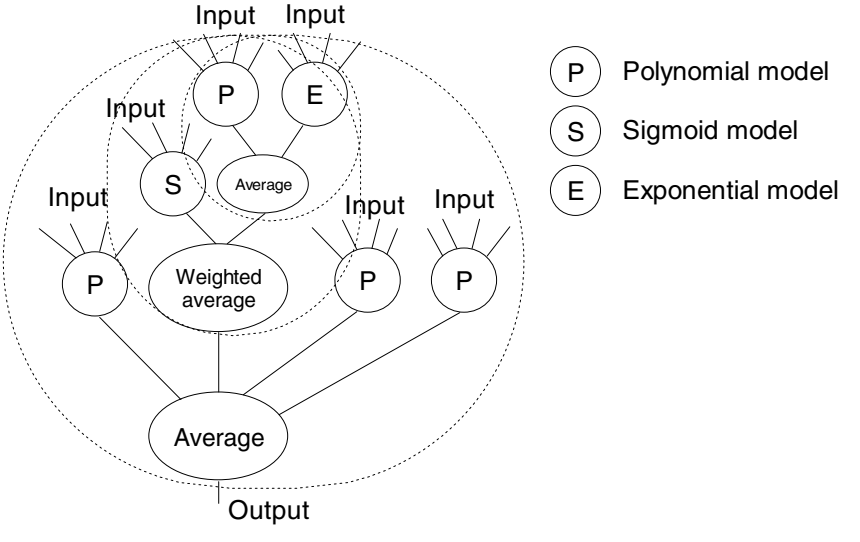


Fig. 24. The model produced by the hierarchical combination of ensembles depicted in Figure 23.

8 Optimization of the SpecGen Ensemble

SpecGen ensemble can be optimized by an algorithm specially designed to evolve tree structures of objects. It consists of two main parts: the first is the evolutionary algorithm itself, which is problem independent. The second part is called *Context* and represents problem-dependent computations (for example data preprocessing and fitness computation). These two parts will be described in detail in the next two sections.

8.1 Evolutionary Algorithm

To understand how the individual is represented in the algorithm, we need to know first how the algorithm operates without knowing anything at all about the optimized objects. It uses a template system where each template represents one elemental construction block for the algorithm. It can be a single ensemble or base model configuration or even a tree which is then regarded as an elemental block. Because the type of the object is not known, we need to create reference to the corresponding template from which the object originated. The template then has auxiliary variables such as references to templates to which it can mutate, methods which are used to change and optimize objects attributes and so on.

Next we are going to explain how the algorithm runs and evolves the configuration structures. The first thing that is done is evaluation of the current generation. This is because we need to supply the initial generation and it needs to be evaluated before any evolution operations can influence it. This is one way to guide the algorithm, by generating non-trivial solutions to initial generation. For checking individual surviving capabilities, we must know its fitness, which is represented by the inverse of the root mean squared error on validation data. There are certain situations that can occur depending on the current object fitness before it is determined whether it survives. In this stage the Context is used to compute object fitness and related operations such as returning the so-far best fitness (see Figure 25).

1. Global fitness is improved and the structure of the best solution changes. This means that we have found a new, better solution. This clears the convergency counter, which represents the number of generations without any improvement to the best solution. Also, if there is a variable in the best solution which has a value more than 50% of its maximum allowed value, that maximum in the given template is doubled. This approach is inspired by evolving from minimal form, which tries to keep state space size to a minimum and expand the parts of the space that look promising.
2. Global fitness is worsened and structure of the best solution changes. This indicates that more calculations were made to make the best solution more precise and it has turned out that its fitness is actually lower. Another solution that was already computed during evolution beforehand, had a greater value of fitness and was chosen as best. This lowers the convergency counter by half, because even if we did not find a better solution, this change is quite important to give it some time to stabilize.

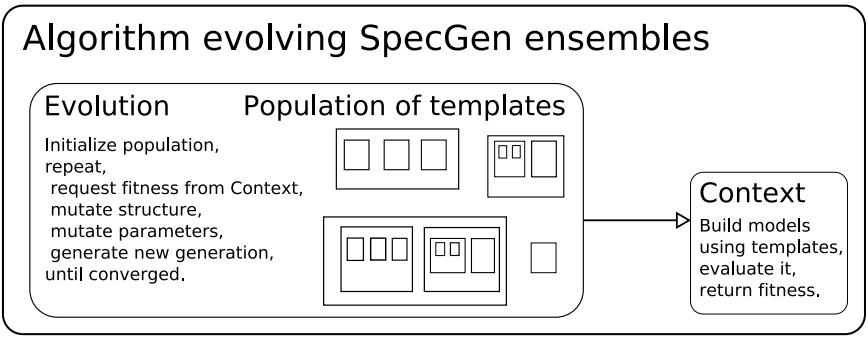


Fig. 25. The SpecGen templates are evolved from a minimal form. Structural mutation can change the type of base model or replace it by an ensemble.

3. Global fitness is improved, but the best solution stays the same. This represents a situation where the best solution is computed again to increase its precision, by which the fitness improves. The convergency counter is not affected in this situation, because we did not gain any new information regarding the best solution.

After this, a survival check is made. It determines the selection pressure in the evolution and also implements an elitism principle (best solutions are protected and cannot die). Survival capabilities of an individual is determined by three things:

- its own fitness,
- fitness of the best solution,
- its age, measured in generations.

That means that in the beginning, when the best solution is poorer, the lifespan of an average individual is quite high. On the other hand, when we have very good best solution in the end, the lifespan of poor individuals is very short. This guides the algorithm to try to improve the best solution more, rather than trying to find a new, different solution.

Next, the stop conditions are checked, which can be a fixed number of maximum generations or the convergency criterion. The convergency criterion also has another meaning in the algorithm. In employing the growth from minimal form principle, the tree depth is limited in the beginning. If the convergency starts to kick in, the depth of the tree is increased and the convergency criterion is reset. The more depth there is, the more generations without change of the best solution can pass before deeper trees are enabled. This goes on to the maximum tree depth. If the convergency criterion is met at the maximum tree depth, the evolution ends.

Lastly the evolution operations are performed. The algorithm uses the three independent types of mutations described below.

Node mutation. The node is changed to some other, randomly chosen node. Nodes to which the object can mutate can be controlled via templates. During the replacement, the following situations may happen, depending on the type of the current and new node.

- leaf → inner node - Leaf becomes descendant of the new inner node.
- inner node → leaf - Inner node is replaced with leaf. Whole subtree below inner node is deleted.
- leaf → leaf - Old leaf is replaced by new one.
- inner node → inner node - Old node is replaced by new and all successors of the old node are connected to the new node.

Successor add mutation. New leaf is added to the current successors of the inner node.

Variable mutation. The evolutionary algorithm can also optimize variable values by variable mutation. Gaussian noise is added to the variable value as its appearance in equation 9. The function *randomGaussian* returns a random value from Gaussian distribution with average 0 and standard deviation 1. If the output value is outside the allowed interval defined by *min* and *max* values, it gets trimmed to the border values of the interval. This dynamic noise size is implemented to achieve the same behaviour and to explore the interval similarly each time, irrespective on the interval size.

$$value = value + \frac{randomGaussian() * (max - min)}{2} \tag{9}$$

After the main evolution ends, we have a pretty good individual, but still there is not enough time in the main evolution to refine the individual perfectly and optimize all its variables. For this purpose (ie detailed variable optimization), a second evolution is run with all the individuals cloned from the best solution. The generation size and convergency criterion is halved to reduce the time of that evolution. This is no change in the structure of the final solution during this part of the evolution. This means that *node mutation* and *add mutation* are both set to zero. On the other hand, the attributes for variable mutations are all boosted implying that intervals of allowed values are all stretched and variable mutation probability is increased.

This second evolution increases the fitness of the final output greatly, because optimizing variables alone is relatively easy and also due to the fact that more values are involved which were not accessible in the previous evolution. But it is still a complex optimization problem because, the tree can be quite large, and every node in the tree can have several variables and the variables are not always independent of each other. This means that if, for example, we were to ran a faster gradient method for every variable, we could miss a lot of potential good solutions: firstly due to the complexity of the fitness function for a given variable, because it can have multiple local maximums and gradient methods can be easily trapped in the local extremes; and secondly due to the variable being dependent on another variable even on a different node.

8.2 Context

Context is the problem-dependant part of the evolutionary algorithm and most of the computing time is spent here. Fitness computation, which means model creation and learning is the most time-consuming operation in the whole algorithm due to that fact that it is essential to make that computation as efficient as possible. This is done by using cache. It combines the accuracy of multiple model learning and averaging output with a speed almost the same as that of the learning model only once.

When a fitness of any configuration is demanded, context checks its cache first. If there is a record that is accurate enough (it has been computed enough times and the RMSE values do not vary too much from its average RMSE) it is simply returned and no other computations are made. If the record is not present or is not accurate enough, the model is created and its RMSE is computed. The result is added onto the average and a more accurate average is returned. The exact values of deviations from average that distinguish different situations are described below.

- *Deviation less than 10%* implies stable output of the models and counts as two values towards the average computation.
- *Deviation 10-50%* is considered normal and counts as one towards average.
- *Deviation greater than 50%* shows large variation in the model output. The value counts into the average, but the number of needed computations to gain a sufficiently accurate result is increased by one.

There is also a another thing that controls the number of models and prevents creation of trees that have too many models to compute. This can happen very quickly in such hierarchical structures. For example a tree with depth 3 and a maximum of 50 models can have up to 125,000 models. If we increase the depth of the tree by one, the new tree can have up to 6 million models, which is not reasonably computable for one individual. But we cannot simply limit the number of models to a lesser number to prevent this, because in that case we would make it impossible to create simple models with lower tree depth. If we, say, limit the number of models to 5, this would prevent us from creating a solution with a depth of 1 and 50 models, which is quite easy to compute. So we would be forbidding simple solutions with our anti-complex rule, which is counter productive.

8.3 Tuning the SpecGen Evolution

We have implemented the above described algorithm and in this section we look for efficient parameter settings in order to minimize the computational expenses without harming the robustness of the algorithm at the same time.

To evaluate the functionality of the evolutionary algorithm, we prepared an artificial data sets [64] simple enough to save computational resources and complex enough not to be solved by a single base model.

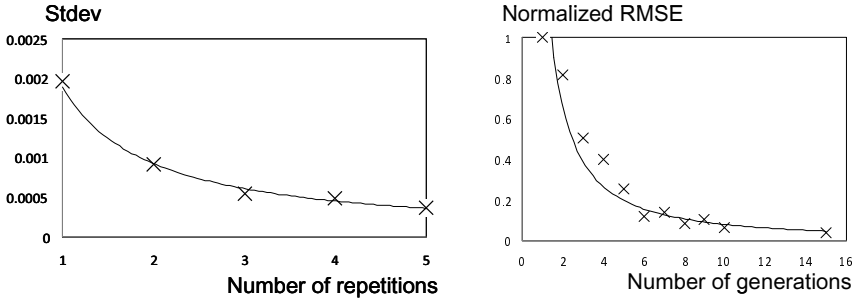


Fig. 26. Standard deviation of RMS errors indicate that the ensemble should be built three times using the same template and the average error is used as the fitness function for the template. Six is the sufficient number of generations.

The first concern is the stability of the fitness function computation. Most of our algorithms are stochastic so we can get different SpecGen ensembles for a single template. We can increase the stability and filter out the noise by building more ensembles for one template and averaging their results on the validation set. Figure 26 shows the standard deviation of fitness value for a single SpecGen ensemble based on the number of ensembles being averaged. As you can see, averaging three SpecGen ensembles stabilizes the fitness function enough.

The second concern (Figure 26 right) was the population size (number of SpecGen templates in one generation). This parameter is problem-dependent, but our experiments show that 6 individuals are sufficient for all the data sets we have been experimenting with so far.

8.4 Evaluation of Improvements

Next experiments evaluate the functionality of the evolutionary algorithm and the usefulness of the parameter evolution and growing complexity scheme.

When we disabled the selection pressure (and the elitism), the error of ensembles produced by the best template evolved increased significantly. In this case, the evolution degraded to the random search.

Table 4. The error of SpecGen ensembles produced by templates evolved with different options.

	RMSE	**Decline[%]**
All options enabled	0.29	-
Selection pressure disabled	0.44	34.1%
Parameter evolution disabled	0.49	40.8%
Growing complexity disabled	0.38	23.7 %

As you can see in the Table 4, the evolution of parameters (e.g. number of base models or degree of polynomial base models) is particulary important.

Increasing the complexity of templates during the evolution has also proved to be advantageous.

8.5 Ensembles versus Templates

During the evolution, the fitness of a template is computed by building 3 ensembles using the template and averaging their performance on the validation data. We compared the generalization performance of models produced by the best template and the ensemble best performing on the validation data discovered during the evolution. Results in the Table 5 are summarized from 20 runs of the evolution on the artificial data set. For this data set, it is better to use the best ensemble found during the evolution. However, the advantage of using the best template is that the generalization performance of models generated using the template is more stable. Experiments on more data sets are needed to evaluate whether best performing ensembles overfit the data. We also plan to build a knowledge base of templates and use it with meta-data to enhance the evolution process.

Table 5. The best ensembles versus the best templates evolved. When generating models using evolved templates, their generalization performance is worse than that of the best ensemble found during the evolution, but more stable.

	Validation data			**Testing data**		
	average	**median**	**std**	**average**	**median**	**std**
Ensembles	0.113	0.117	0.0014	0.272	0.172	0.0546
Templates	0.17	0.163	0.0023	0.295	0.234	0.0288

8.6 Benchmaring the SpecGen templates

Finally, we compare the performance of evolved hierarchical ensembles with state-of-the-art ensembles on benchmarking problems used in the previous section.

Table 7 summarizes the setup of the evolutionary algorithm for the benchmarks. The evolution was run three times for each data set and the template with lowest error on testing data was compared to the best "traditional" ensemble from the Table 2.

The overall comparison is presented in the Table 6, best templates are then listed in the separate Table 8. For data sets, where the complexity of input-output relationship is lower, the improvement is not very significant. Remarkable results were obtained for data with complex relationship (e.g. Donojo-Johnstone), where evolved hierarchical ensembles greatly reduced bias of base models.

Table 6. Errors of hierarchical ensembles produced by evolved templates are compared to best performing basic ensembles from the Table 2. The evolved templates alone are listed in the separate Table 8 due to lack of space.

	Traditional ensemble		**SpecGen**	
Data	**RMSE**	**config**	**RMSE**	**Improvement [%]**
Block	1.21	AS[10*G]	0.0015	99.9
Bump	1.39	AS[10*P]	0.678	51.2
Doppler	1.24	AS[10*G]	0.65	47.6
Heavysine	0.4	DIV[10*SI]	0.0116	97.1
Block noise	2.88	AS[10*L]	2.44	15.3
Bump n.	3.07	AS[10*SG]	2.45	20.2
Doppler n.	2.87	AS[10*G]	2.61	9.1
Heavysine n.	2.29	DIV[10*P]	2.23	2.6
Bosthouse	3.31	SG	3.23	2.4
Buildings 1	115.4	AS[G]	112.4	2.6
Buildings 2	0.517	AS[SI]	0.46	11
Buildings 3	0.597	AS[P]	0.52	12.9
Flare 1	0.941	SI	0.845	10.2
Flare 2	0.0318	BST[L]	0.0183	42.5
Flare 3	0.024	BST[L]	0.0127	47.1
Heart	0.196	BST[SG]	0.192	2
Mandarin	0.097	ST[E,P]	0.081	16.5
Antro Age	11.74	BST[SI]	11.48	2.2

Table 7. The setup of the evolutionary algorithm for overall benchmarks.

	RMSE
Maximal number of generations	200
Maximal tree depth	3
Population size	6
Training data percentage	60%
Validation data percentage	20%
Testing data percentage	20%
Maximal number of vectors	700

For some data (Antro Age, Flare 1, Building 3, Bosthouse) the evolutionary algorithm consistently produces very simple ensembles (or just elementary models) because the relationship in data can be expressed easily. On the other hand, some data require complex hierarchical structures and also for these data, the evolutionary algorithm finds the appropriate solution. Templates are varying, when the evolution is run multiple times on a single data set, but the type of utilized ensembles and base models is often the same. We plan to build a knowledge base of efficient templates for wide range of data sets represented by meta-data.

Table 8. SpecGen templates evolved for benchmarking data sets. Abbreviations of ensembles and their parameters are summarized in the Table 3.

Data	Evolved template of the SpecGen ensemble
Block	AS(area=1,spec=6.78)[145x AS(area=16,spec=4.97)[2x E], SI, DIV(mult=1)[6x P(degree=2)]]
Bump	AS(area=7,spec=5)[8x BST(tr=0)[4x E, P(degree=4)], SG, AS(area=26,spec=1)[20x AS(area=7,spec=14)[2x SG]]]
Doppler	AS(area=3,spec=20.8)[72x BAG[2x BAG[5x L, G], E]]
Heavysine	AS(area=1,spec=9.1)[178x BAG[2x G], G, BAG[2x P(degree=15)]]
Block noise	ST[12x AS(area=25,spec=1)[30x AS(area=1,spec=3.48)[2x SI]], L, AS(area=21,spec=3.64)[2x DIV(mult=3.86)[2x E, SI]]]
Bump n.	BAG[5x AS(area=6,spec=5)[56x SI]]
Doppler n.	AS(area=20,spec=1)[19x DIV(mult=1.48)[16x L, SI]]
Heavysine n.	DIV(mult=7.59)[18x DIV(mult=6.86)[2x P(degree=2), G]]
Bosthouse	DIV(mult=60)[10x SG]
Buildings 1	AS(area=10,spec=7)[5x DIV(mult=3.09)[5x E]]
Buildings 2	BST(tr=0)[6x AS(area=1,spec=28.25)[2x E, BAG[5x P(degree=2)]], L]
Buildings 3	P(degree=6)
Flare 1	ST[2x BST(tr=0.1)[5x SG], P(degree=2)]
Flare 2	BST(tr=0.28)[33x SI, BST(tr=0)[2x L, SG]]
Flare 3	AS(area=7,spec=17.98)[8x BAG[10x P(degree=5)], CG[10x SI], AS(area=1,spec=14.12)[18x L], L]
Heart	BAG[13x CG[2x P(degree=2), BAG[2x SG, P(degree=6)]]]
Mandarin	AS(area=13,spec=13.15)[7x DIV(mult=8.21)[5x G, AS(area=4,spec=11.86)[5x SG]], AS(area=3,spec=1)[2x E], G]
Antro Age	BST(tr=0.06)[2x SG]

9 Conclusion

In this chapter we focused on regression models for data mining purposes. These models are not so frequently used as classifiers, but many interesting real-world problems involve a continuous output variable. Our study starts with elementary regression models and their learning. Then we present the GAME algorithm combining these models into a layered structure. In each layer, inputs and transfer functions of base models are evolved, while their diversity is preserved. We inspect the diversity and accuracy of base models and experiment with several niching strategies. Probabilistic Crowding was identified as the best performing strategy. However, results are very data dependent and often insignificant. Our ambition to use the GAME algorithm as a universal automated model selection approach was inhibited by its poor performance when modelling complex one-dimensional signals (Donojo-Johnstone benchmarks). The reason is that the data manipulation and model combination used in GAME (Bootstrap sampling and Stacking) is inefficient for this type of problem.

We are also studying regression variants of metalearning algorithms to combine our elementary regression models. We measure their performance for several benchmarking problems and look for the best setting of their parameters and for the appropriate type of base models. The results confirmed our expectations - the performance, parameters and appropriate base model type are heavily data-dependent.

During our experiments with base models of increasing plasticity, we found that a certain hierarchical combination of ensemble techniques (metalearning algorithms) can produce models with superior results. Again, the optimal combination is data-dependent.

In the second part of the chapter, we introduce the Specialization-Generalization (SpecGen) ensembling algorithm. It combines metalearning algorithms in a hierarchical manner. For each data set, we evolve a special ensembling template, which tells us how to distribute data to base models and how to combine them. Each base model can be *specialized* to certain data vectors and its output is *generalized* through the hierarchy into the final prediction. We evolved SpecGen templates for several benchmarking problems. These templates were used to build models, and the generalization accuracy of these models outperformed the model produced by all ensembling strategies for all data sets.

Our future work is to build a knowledge base of evolved SpecGen templates for several data sets and preselect efficient combination strategies based on meta data describing data sets.

We also plan to deal with the evolution of base model connections and transfer function structure.

The proposed SpecGen algorithm and its evolutionary framework is now being applied to classifiers with similar results as for the regression models presented in this chapter.

Acknowledgements

This research benefits from the FAKE GAME framework built within the Automated Knowledge Extraction (KJB201210701) project funded by the Grant Agency of the Academy of Science of the Czech Republic (2006-2009). It is now partially supported by the grant Novel Model Ensembling Algorithms ($SGS10/307/OHK3/3T/181$) of the Czech Technical University in Prague and the research program "Transdisciplinary Research in the Area of Biomedical Engineering II" ($MSM6840770012$) sponsored by the Ministry of Education, Youth and Sports of the Czech Republic.

References

1. Uci machine learning repository (September 2006), `http://www.ics.uci.edu/~mlearn/MLSummary.html`
2. The fake game environment for the automatic knowledge extraction (November 2008), `http://www.sourceforge.net/projects/fakegame`

3. Abdel-Aal, R.: Improving electric load forecasts using network committees. Electric Power Systems Research (74), 83–94 (2005)
4. Alpaydin, E., Kaynak, C.: Cascading classifiers. Kybernetika 34, 369–374 (1998)
5. Analoui, M., Bidgoli, B.M., Rezvani, M.H.: Hierarchical classifier combination and its application in networks intrusion detection. In: International Conference on Data Mining Workshops, vol. 0, pp. 533–538 (2007)
6. Bakker, B., Heskes, T.: Clustering ensembles of neural network models. Neural Netw. 16(2), 261–269 (2003)
7. Bao, X., Bergman, L., Thompson, R.: Stacking recommendation engines with additional meta-features. In: RecSys 2009: Proceedings of the third ACM conference on Recommender systems, pp. 109–116. ACM, New York (2009)
8. Bennett, J., Lanning, S., Netflix, N.: The netflix prize. In: KDD Cup and Workshop in conjunction with KDD (2007)
9. Brazdil, P., Giraud-Carrier, C., Soares, C., Vilalta, R.: Metalearning: Applications to Data Mining. Cognitive Technologies. Springer, Heidelberg (2009)
10. Breiman, L.: Bagging predictors. Mach. Learn. 24(2), 123–140 (1996)
11. Brown, G.: Diversity in Neural Network Ensembles. PhD thesis, The University of Birmingham, School of Computer Science, Birmingham B15 2TT, United Kingdom (January 2004)
12. Brown, G., Yao, X.: On the effectiveness of negative correlation learning. In: Proceedings Of First Uk Workshop On Computational Intelligence, pp. 57–62 (2001)
13. Chandra, Arjun, Yao, Xin: Ensemble learning using multi-objective evolutionary algorithms. Journal of Mathematical Modelling and Algorithms 5(4), 417–445 (2006)
14. Costa, E.P., Lorena, A.C., Carvalho, A.C., Freitas, A.A.: Top-down hierarchical ensembles of classifiers for predicting g-protein-coupled-receptor functions. In: Bazzan, A.L.C., Craven, M., Martins, N.F. (eds.) BSB 2008. LNCS (LNBI), vol. 5167, pp. 35–46. Springer, Heidelberg (2008)
15. Dietterich, T.G.: An experimental comparison of three methods for constructing ensembles of decision trees: Bagging, boosting, and randomization. Machine Learning 40(2), 139–157 (2000)
16. Donoho, D.L.: De-noising by soft-thresholding. IEEE Trans. on Inf. Theory 41(3), 613–662 (1995)
17. Drchal, J., Šnorek, J.: Diversity visualization in evolutionary algorithms. In: Štefen, J. (ed.) Proceedings of 41th Spring International Conference MOSIS 2007, Modelling and Simulation of Systems, pp. 77–84. MARQ, Ostrava (2007)
18. Durham, G.B., Gallant, A.R.: Numerical techniques for maximum likelihood estimation of continuous-time diffusion processes. Journal Of Business And Economic Statistics 20, 297–338 (2001)
19. Eastwood, M., Gabrys, B.: The dynamics of negative correlation learning. J. VLSI Signal Process. Syst. 49(2), 251–263 (2007)
20. Fahlman, S.E., Lebiere, C.: The cascade-correlation learning architecture. Technical Report CMU-CS-90-100, Carnegie Mellon University Pittsburgh, USA (1991)
21. Ferri, C., Flach, P., Hernández-Orallo, J.: Delegating classifiers. In: ICML 2004: Proceedings of the twenty-first international conference on Machine learning, p. 37. ACM Press, New York (2004)
22. Freund, Y., Schapire, R.: A decision-theoretic generalization of on-line learning and an application to boosting. In: Proceedings of the Second European Conference on Computational Learning Theory, pp. 23–37. Springer Verlag, Heidelberg (1995)
23. Friedman, J.H.: Greedy function approximation: A gradient boosting machine. Annals of Statistics 29, 1189–1232 (2000)

24. Gama, J., Brazdil, P.: Cascade generalization. Mach. Learn. 41(3), 315–343 (2000)
25. Gelbukh, A., Reyes-Garcia, C.A. (eds.): MICAI 2006. LNCS (LNAI), vol. 4293. Springer, Heidelberg (2006)
26. Granitto, P., Verdes, P., Ceccatto, H.: Neural network ensembles: evaluation of aggregation algorithms. Artificial Intelligence 163, 139–162 (2005)
27. Hansen, L.K., Salamon, P.: Neural network ensembles. IEEE Trans. Pattern Anal. Machine Intelligence 12(10), 993–1001 (1990)
28. Islam, M. M., Yao, X., Murase, K.: A constructive algorithm for training cooperative neural network ensembles. IEEE Transitions on Neural Networks 14(4) (July 2003)
29. Islam, M.M., Yao, X., Nirjon, S.M.S., Islam, M.A., Murase, K.: Bagging and boosting negatively correlated neural networks (2008)
30. Ivakhnenko, A.G.: Polynomial theory of complex systems. IEEE Transactions on Systems, Man, and Cybernetics SMC-1(1), 364–378 (1971)
31. Jacobs, R.A.: Bias/variance analyses of mixtures-of-experts architectures. Neural Comput. 9(2), 369–383 (1997)
32. Kaynak, C., Alpaydin, E.: Multistage cascading of multiple classifiers: One man's noise is another man's data. In: Proceedings of the Seventeenth International Conference on Machine Learning ICML 2000, pp. 455–462. Morgan Kaufmann, San Francisco (2000)
33. Kohavi, R.: A study of cross-validation and bootstrap for accuracy estimation and model selection. In: Proceedings of International Joint Conference on Artificial Intelligence (1995)
34. Kordík, P.: Fully Automated Knowledge Extraction using Group of Adaptive Models Evolution. PhD thesis, Czech Technical University in Prague, FEE, Dep. of Comp. Sci. and Computers, FEE, CTU Prague, Czech Republic (September 2006)
35. Kordik, P.: Hybrid Self-Organizing Modeling Systems. In: Onwubolu, G.C. (ed.). Studies in Computational Intelligence, vol. 211, p. 290. Springer, Heidelberg (2009)
36. Kordík, P., Koutník, J., Drchal, J., Kovárík, O., Cepek, M., Snorek, M.: Meta-learning approach to neural network optimization. Neural Networks 23(4), 568–582 (2010)
37. Kordík, P., Křemen, V., Lhotská, L.: The game algorithm applied to complex fractionated atrial electrograms data set. In: 18th International Conference Proceedings Artificial Neural Networks - ICANN 2008, vol. 2, pp. 859–868. Springer, Heidelberg (2008)
38. Koren, Y.: Collaborative filtering with temporal dynamics. In: Proceedings of the 15th ACM SIGKDD international conference on Knowledge discovery and data mining KDD 2009, pp. 447–456. ACM, New York (2009)
39. Kuncheva, L.I.: Combining Pattern Classifiers: Methods and Algorithms. John Wiley and Sons, New York (2004)
40. Kuncheva, L., Whitaker, C.: Ten measures of diversity in classifier ensembles: Limits for two classifiers. In: Proc. of IEE Workshop on Intelligent Sensor Processing, pp. 1–10 (2001)
41. Kuncheva, L.I., Whitaker, C.J.: Measures of diversity in classifier ensembles and their relationship with the ensemble accuracy. Machine Learning 51, 181–207 (2003)
42. Kurkova, V.: Kolmogorov's theorem is relevant. Neural Computation 3, 617–622 (1991)
43. Liu, Y., Yao, X.: Ensemble learning via negative correlation. Neural Networks 12, 1399–1404 (1999)

44. Mahfoud, S.W.: Niching methods for genetic algorithms. Technical Report 95001. Illinois Genetic Algorithms Laboratory (IlliGaL), University of Ilinios at Urbana-Champaign (May 1995)
45. Mandischer, M.: A comparison of evolution strategies and backpropagation for neural network training. Neurocomputing (42), 87–117 (2002)
46. Marquardt, D.W.: An algorithm for least-squares estimation of nonlinear parameters. SIAM Journal on Applied Mathematics 11(2), 431–441 (1963)
47. Melville, P., Mooney, R.J.: Constructing diverse classifier ensembles using artificial training examples. In: Gottlob, G., Walsh, T. (eds.) IJCAI, pp. 505–512. Morgan Kaufmann, San Francisco (2003)
48. Mengshoel, O.J., Goldberg, D.E.: Probabilistic crowding: Deterministic crowding with probabilisitic replacement. In: Banzhaf, W., Daida, J., Eiben, A.E., Garzon, M.H., Honavar, V., Jakiela, M., Smith, R.E. (eds.) Proceedings of the Genetic and Evolutionary Computation Conference, vol. 1, pp. 409–416. Morgan Kaufmann, San Francisco (1999)
49. Muller, J.A., Lemke, F.: Self-Organising Data Mining. Berlin (2000), ISBN 3-89811-861-4
50. Nabney, I.T.: Efficient training of rbf networks for classification. Int. J. Neural Syst. 14(3), 201–208 (2004)
51. Oh, S.K., Pedrycz, W.: The design of self-organizing polynomial neural networks. Inf. Sci. 141, 237–258 (2002)
52. Oh, S.-K., Pedrycz, W., Park, B.-J.: Polynomial neural networks architecture: analysis and design. Computers and Electrical Engineering 29, 703–725 (2003)
53. Pejznoch, J.: Niching Evolutionary Algorithms in GAME. Ph.D thesis, Czech Technical University in Prague, FEE, Dep. of Comp. Sci. and Computers, FEE, CTU Prague, Czech Republic (May 2010)
54. Pétrowski, A.: A clearing procedure as a niching method for genetic algorithms. In: International Conference on Evolutionary Computation, pp. 798–803 (1996)
55. Pilný, A., Kordík, P., Šnorek, M.: Feature ranking derived from data mining process. In: Kůrková, V., Neruda, R., Koutník, J. (eds.) ICANN 2008, Part II. LNCS, vol. 5164, pp. 889–898. Springer, Heidelberg (2008)
56. Ritchie, M.D., White, B.C., Parker, J.S., Hahn, L.W., Moore, J.H.: Optimization of neural network architecture using genetic programmi ng improves detection and modeling of gene-gene interactions in studies of human diseases. BMC Bioinformatics 4(1) (July 2003)
57. Rokach, L.: Ensemble methods for classifiers. In: Maimon, O., Rokach, L. (eds.) The Data Mining and Knowledge Discovery Handbook, pp. 957–980 (2005)
58. Schapire, R.E.: The strength of weak learnability. Mach. Learn. 5(2), 197–227 (1990)
59. Sexton, R.S., Gupta, J.: Comparative evaluation of genetic algorithm and backpropagation for training neural networks. Information Sciences 129, 45–59 (2000)
60. Stanley, K.O.: Efficient evolution of neural networks through complexification. PhD thesis (2004)
61. Stanley, K.O., Miikkulainen, R.: Evolving neural networks through augmenting topologies. Evolutionary Computation Massachusetts Institute of Technology 10(2), 99–127 (2002)
62. Sung, Y.H., kyun Kim, T., Kee, S.C.: Hierarchical combination of face/non-face classifiers based on gabor wavelet and support vector machines (2009)
63. Töscher, A., Jahrer, M.: The bigchaos solution to the net ix grand prize. Technical report, commendo research & consulting (2009)

64. Černy, J.: Methods for combining models and classifiers. PhD thesis, Czech Technical University in Prague, FEE, Dep. of Comp. Sci. and Computers, FEE, CTU Prague, Czech Republic (May 2010)
65. Wang, S., Tang, K., Yao, X.: Diversity exploration and negative correlation learning on imbalanced data sets. In: Proceedings of the 2009 international joint conference on Neural Networks IJCNN 2009, pp. 1796–1803. IEEE Press, Piscataway (2009)
66. Webb, G.I., Zheng, Z.: Multi-strategy ensemble learning: Reducing error by combining ensemble learning techniques. IEEE Transactions on Knowledge and Data Engineering 16 (2004)
67. Wolpert, D.H.: Stacked generalization. Neural Networks 5, 241–259 (1992)
68. Wolpert, D.H., Macready, W.G.: Combining stacking with bagging to improve a learning algorithm. Technical report, Santa Fe Institute (1996)
69. Zhou, Z.-H., Wu, J., Tang, W.: Ensembling neural networks: Many could be better than all. Artificial Intelligence 137, 239–263 (2002)

Selecting Machine Learning Algorithms Using the Ranking Meta-Learning Approach

Ricardo B.C. Prudêncio[1], Marcilio C.P. de Souto[2], and Teresa B. Ludermir[1]

[1] Center of Informatics, Federal University of Pernambuco,
Pobox 7851 - CEP 50732-970 - Recife (PE) - Brazil
{rbcp,tbl}@cin.ufpe.br
[2] Dept. of Informatics and Applied Mathematics,
Fed. Univ. of Rio Grande do Norte, Natal, Brazil
marcilio@dimap.ufrn.br

Abstract. In this work, we present the use of Ranking Meta-Learning approaches to ranking and selecting algorithms for problems of time series forecasting and clustering of gene expression data. Given a problem (forecasting or clustering), the Meta-Learning approach provides a ranking of the candidate algorithms, according to the characteristics of the problem's dataset. The best ranked algorithm can be returned as the selected one. In order to evaluate the Ranking Meta-Learning proposal, prototypes were implemented to rank artificial neural networks models for forecasting financial and economic time series and to rank clustering algorithms in the context of cancer gene expression microarray datasets. The case studies regard experiments to measure the correlation between the suggested rankings of algorithms and the ideal rankings. The results revealed that Meta-Learning was able to suggest more adequate rankings in both domains of application considered.

1 Introduction

One of the major challenges in many domains of Computational Intelligence, Machine Learning, Data Analysis and other fields is to investigate the capabilities and limitations of the existing algorithms in order to identify when one algorithm is more adequate than another to solve particular problems [1]. Traditional approaches to selecting algorithms involve, in general, costly trial-and-error procedures, or require expert knowledge, which is not always easy to acquire [2]. Meta-Learning for algorithm selection arises in this context as an effective solution, capable of automatically predicting algorithm's performance, thus assisting users in the choice of the most adequate techniques for dealing with the problems at hand [2,3,4,5,6].

In Meta-Learning, each *meta-example* is related to a learning problem and stores: (1) the features describing the problem, called *meta-features*; and (2) the *performance information* about one or more algorithms when applied to the problem. By receiving a set of such meta-examples, another learning system (the *meta-learner*) is applied to acquire knowledge relating the performance of the candidate algorithms and the descriptive features of the problems. The acquired

N. Jankowski et al. (Eds.): Meta-Learning in Computational Intelligence, SCI 358, pp. 225–243.
springerlink.com

knowledge can then be used to predict algorithm performance for new problems not seen during the Meta-Learning process and to recommend algorithms. Different authors in Meta-Learning have developed techniques to suggest either one single algorithms or a small group of algorithms among the set of candidate ones. A more informative and flexible solution for algorithm selection is to provide a ranking of the candidate algorithms, since alternative algorithms can be eventually chosen by the users according to particular interests [7]. Ranking Meta-Learning approaches have been investigated in different case studies, mainly focused on classification and regression problems [1,6,7].

In this chapter, based on our previous works, we present the use of Meta-Learning for ranking algorithms in two different classes of problems: time series forecasting and clustering of gene expression data. Both domains are characterized by the existence of a variety of algorithms to be applied and a lack of useful guidelines to support algorithm selection. The Meta-Learning techniques originally proposed for classification and regression problems were extrapolated in our previous work to rank time series models and clustering techniques. The application of Meta-Learning in these domains was not deeply investigated yet. In a first case study, the Zoomed-Ranking approach was used to rank Artificial Neural Network models for forecasting financial and economic time series [8]. In a second case study, a Meta-Regression approach was used to rank clustering techniques for cancer gene expression [9].

The remaining of this paper is organized as follows. Section 2 presents a brief introduction on the topic of Meta-Learning. Section 3 presents the general architecture of our solution as well as some implementation issues. Section 4 brings a case study (implementation and experiments) performed in the domain of time series forecasting, followed by section 5 which presents a case study in the domain of clustering gene expression. Finally, section 6 concludes the paper with some final considerations.

2 Meta-Learning

There are different interpretations of the term *Meta-Learning* in the literature [3,6,10]. In our work, we focused on the definition of Meta-Learning as the automatic process of acquiring knowledge that relates the performance of learning algorithms to the features of the learning problems [2]. The acquired knowledge supports the task of algorithm selection, which has shown to be a difficult problem in many contexts [11].

The knowledge in Meta-Learning is commonly represented considering meta-features which describe learning problems. The meta-features are, in general, statistics describing the training dataset of the problem, such as number of training examples, number of attributes, correlation between attributes, class entropy, among others [7,12,1]. An alternative strategy to define meta-features is the *Landmarking* proposal [13]. This approach tries to relate the performance of the candidate algorithms to the performance obtained by simpler and faster designed learners, called *landmarkers*. Landmarking claims that some widely

used meta-features are very time consuming, and hence, landmarking would be an economic approach to the characterization of problems and to provide useful information for the Meta-Learning process.

Regarding the performance information (the *target* in the Meta-Learning task), each meta-example may store a class attribute which indicates the best algorithm for the problem, among a set of candidates [14,15,16,17,18]. In this strict formulation of Meta-Learning, the class label for each meta-example is defined by performing a cross-validation experiment using the available dataset. The meta-learner is simply a classifier which predicts the best algorithm based on the meta-features of the problem.

In [19], the authors used an alternative approach to defining the performance information and hence to labeling meta-examples. Initially, 20 algorithms were evaluated through cross-validation on 22 classification problems. For each algorithm, the authors generated a set of meta-examples, each one associated either to the class label *applicable* or to the class label *non-applicable*. The class label *applicable* was assigned when the classification error obtained by the algorithm fell within a pre-defined confidence interval, and *non-applicable* was assigned otherwise. Each problem was described by a set of 16 meta-features and, finally, a decision tree was induced to predict the applicability of the candidate algorithms.

In [1], the authors performed the labeling of meta-examples by deploying a clustering algorithm. Initially, the error rates of 10 algorithms were estimated for 80 classification problems. From this evaluation, they generated a matrix of dimension 80 X 10, in which each row stored the ranks obtained by the algorithms in a single problem. The matrix was given as input to a clustering algorithm, aiming to identify groups (clusters) of problems in which the algorithms obtained specific patterns of performance (e.g. a cluster in which certain algorithms achieved a considerable advantage relative to the others). The meta-examples were then associated to the class labels corresponding to the identified clusters. Hence, instead of only predicting the best algorithm or the applicability of algorithms, the meta-learner can predict more complex patterns of relative performance.

Different approaches have been proposed in order to add new functionalities in the Meta-Learning process, especially to provide *rankings* of algorithms instead of recommending a single one. In [20,21], for instance, a combination of strict meta-learners is used to recommend rankings of algorithms. In this approach, a strict meta-learner is built for each different pair (X, Y) of algorithms. Given a new learning problem, the outputs of the meta-learners are collected and then, points are credited to the algorithms according to the outputs. For instance, if 'X' is the output of meta-learner (X, Y) then the algorithm X is credited with one point. The ranking of algorithms is recommended for the new problem directly from the number of points assigned to the algorithms.

The Meta-Regression approach [22,23] tries to directly predict the accuracy (or alternatively the error) of each candidate algorithm. The meta-learner in this case may be used either to select the algorithm with the highest predicted

accuracy or to provide a ranking of algorithms based on the order of predicted accuracies. In [22], for instance, the authors obtained good results when a linear regression model was used to predict the accuracy of 8 different classification algorithms.

In the Zoomed-Ranking approach [24], the authors proposed to use instance-based learning in order to produce rankings of algorithms taking into account accuracy and execution time. In this approach, each meta-example stores the meta-features describing a learning problem, as well as the accuracy and execution time obtained by each candidate algorithm in the problem. Given a new learning problem, the Zoomed-Ranking retrieves the most similar past problem based on the similarity of meta-features. The ranking of algorithms is then recommended for the new problem by deploying a multi-criteria measure that aggregates the total accuracy and execution time obtained by the algorithms in the similar problems. More recently, the authors provided a deeper investigation of these ideas [7].

The concepts and techniques of meta-learning were mainly evaluated to select the best algorithms for classification and regression problems. In recent years, Meta-Learning has been extrapolated to other domains of application, such as in the selection of time series forecasting models [18], design of planning systems [25], combinatorial optimization [26], software engineering [27] and bioinformatics [9,28,29]. In such domains, Meta-Learning can be seen as tool for analysis of experiments performed by using a number of algorithms on a large set of problems that can be solved by these algorithms. The knowledge acquired from this analysis can be used to select algorithms for new problems. As highlighted in [5], Meta-Learning can be useful to a potentially large number of fields, since its developments can be extrapolated to learn about the behavior of algorithms on different classes of problems.

3 System Architecture and Implementation Issues

In this paper we present the use of *Ranking Meta-Learning* approaches in two different domains: time series forecasting and clustering of gene expression data [8,9]. For each domain, we implemented a specific prototype in order to perform experiments and evaluate the usefulness of the Meta-Learning solution. In this section, we introduce a general architecture of Meta-Learning systems, as well as some implementation issues that guided us in the construction of the prototypes.

3.1 General Architecture

Figure 1 shows the general architecture of systems employed for, given a dataset, ranking the candidates algorithms. As it is common in Machine Learning, the system has two phases: training and use. In the training phase, the Meta-Learner (ML) extracts knowledge from the set of meta-examples stored in the Database (DB). Such a knowledge relates characteristics of the data to the performance of the candidate algorithms. In the case studies presented, these algorithms are either time series models or clustering techniques.

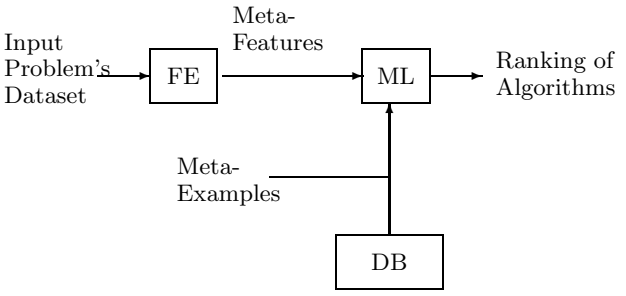


Fig. 1. System's architecture.

In the phase of use, given a new dataset, the Feature Extractor (FE) produces the values of the meta-features that describe these data. According to such values, the Meta-Learner (ML) module outputs a ranking of the available candidate algorithms. In order to do so, the ML module uses the knowledge previously provided as a result of the training phase.

The DB stores descriptions of datasets (i.e., meta-examples) used in the training phase. This set of meta-examples is semi-automatically created: (1) the choice of datasets and algorithms to be considered is a manual task; (2) the generation of the meta-features is automatically accomplished by the FE module; and (3) the performance of the candidate algorithms in each dataset is empirically obtained by directly applying each algorithm to the data and assessing the result yielded.

The ML module implements the chosen meta-learning approach to extracting knowledge (training phase) to be utilized in the choice or ranking of the candidate algorithms (use phase). As seen in Section 2, the Meta-Learning approaches implement one or more machine learning algorithms to execute such tasks. In this context, one could employ a learning technique to recommend one single algorithm from the set of candidate ones. Although this is a worthwhile approach, a more informative and flexible solution for algorithm selection is to output a ranking of the candidate algorithms to each dataset under analysis [7]. In this context, if enough resources are available, more than one algorithm could be employed with the data. Furthermore, if the one has some preference for a given subset of candidate algorithms, one can choose the algorithms that presented the best rank among the algorithms of interest.

3.2 Implementation Issues

To implement a system according to the architecture introduced in the previous section, one has to consider some important questions. Since the kind of dataset to be considered will have an impact on all the other aspects in the system's implementation, this is the first question to be addressed. Datasets are often collected from benchmarking repositories (e.g., the UCI repository in the case of classification and regression tasks) or artificially generated as performed in [30].

Then, one needs to specify which algorithms will be considered to form the set of candidate algorithms. The candidate algorithms should be selected in such a way to provide a wide range of characteristics, as well as to give some generality to the results.

The third question to be approached is which features will be employed by the FE module to describe the datasets. This decision depends on the kind of dataset being analyzed. For instance, in the context of classification problems, one can find standard sets of meta-features that have been used in the meta-learning field. This is the case of the *Data Characterization Tool*, developed within the METAL project[1]. In contrast, for time series forecasting and cluster analysis, since the application of meta-learning to these domains is relatively new, there is no such standard set of attributes. Nevertheless, one can follow some general guidelines to define them. For example, one should choose meta-features that can be reliably identified, preventing subjective analysis, such as visual inspection of plots. Subjective feature extraction is time consuming, requires expertise, and has a low degree of reliability [31]. One should also employ a manageable number of features in order to avoid a time consuming selection process.

4 Ranking Models for Time Series Forecasting

Time series forecasting has been used in several real world problems in order to eliminate losses resultant from uncertainty, as well as to support the decision-making process [32]. Several models can be used to forecast a time series. Selecting the most adequate model for a given time series, from a set of available models, may be a difficult task depending on the candidate models and the time series characteristics.

A straightforward solution to model selection is to perform an empirical evaluation (e.g. hold-out, cross-validation,...) using the available time series data, and compare the estimated performance obtained by the candidate models [33]. Despite its simplicity, this solution is costly for a large amount of series to forecast or several candidate models to evaluate [34].

A more efficient approach to selecting models is based on the development of expert systems [31], in which rules are designed to relate time series features (e.g. length, basic trend, autocorrelations...) and the candidate models performance. A landmarking work in this approach is the Rule-Based Forecasting system [31], in which an expert system with 99 rules was used to weight four forecasting methods. In the experiments performed using the expert system, the improvement in accuracy has shown to be significant. The main limitation of the expert system approach, however, is the difficulty in acquiring knowledge, since that good experts in time series forecasting are expensive and not always available [35]. This limitation may be even more drastic in the case of more complex models.

[1] http://www.cs.bris.ac.uk/~cgc/METAL

In order to minimize the above difficulty, in [18], the authors developed an original work which treats the model selection problem via Meta-Learning approaches. This solution is able not only to select the best model to forecast a given series, but also to provide more informative results, such as a ranking of the candidate models according to their performance in forecasting the given series. The viability of using Meta-Learning in the context of time series forecasting was confirmed in a number of different experiments [18,17,8,36].

In this section, we reviewed the use of a specific Meta-Learning approach, the Zoomed-Ranking (ZR), to rank Artificial Neural Networks (ANNs) forecasting models [8]. The motivation was that, although ANNs represent a powerful approach to forecasting, there is not much knowledge to guide its usage, compared to the existing knowledge that supports the use of simpler linear models [34]. Hence, the investigation of ZR for ANN model selection contributes both to the research on Meta-Learning and to research on ANNs for time series forecasting.

In order to verify the viability of Meta-Learning, a prototype was implemented following the general architecture presented in section 3. The implemented prototype was used to select the following ANN models:

1. TDNN (Time Delay Neural Network) [37]: it corresponds to a feedforward network with time delays in the connections. The input layer receives a fixed time window of the series at hand (i.e. a fixed number of past values of the series), in order to forecast future values of the series. In our work, the time window size was defined by verifying the number of statistical significant autocorrelations in the series up to the limit of 3 past values. The number of hidden neurons was defined by deploying an out-of-sample experiment [33]. In this experiment, we evaluated the TDNN with 1, 2 and 3 hidden neurons on a series sample left out and depicted the best on in terms of forecasting accuracy. The network weights were trained using the Levenberg-Marquardt algorithm [38];
2. Time-Lagged RBF (Radial Basis Function) [39]: it corresponds to a traditional RBF neural network in which the input layer receives a time window of the series at hand (as in the TDNN model). The methodology adopted to define the time window size and the number of hidden neurons was the same one adopted for the TDNN;
3. SOMTAD (SOM with Temporal Activity Diffusion) [40]: it corresponds to a SOM network that creates temporally correlated neighborhoods in the output space. We defined the input layer of the SOM as the past 3 values of the series. In the SOM training, a 10x10 bi-dimensional map was adopted with learning rate of 0.3.

The candidate ANN models were used to forecast benchmarking time series related to financial, micro and macro-economic domains, available in the Time Series Data Library (TSDL) repository[2]. In the following sections, we provide more details of the implemented prototype.

[2] http://www-personal.buseco.monash.edu.au/~hyndman/TSDL

4.1 Feature Extractor

In this case study, we used in the FE module 5 different features to describe the TSDL series:

1. Length of the time series (L): number of observations of the series;
2. Basic Trend (BT): slope of the linear regression model. As higher this feature value, higher is the global trend of the series;
3. Test of Turning Points (TP): Z_t is a turning point if $Z_{t-1} < Z_t > Z_{t+1}$ or $Z_{t-1} > Z_t < Z_{t+1}$. The presence of a very large number or a very small number of turning points indicates that the series is not generated by a purely random process;
4. Average Coefficient of Autocorrelation (AC): average of the first 5 autocorrelation coefficients. Large values of this feature suggest a strong correlation between adjacent points in the series;
5. Type of the time series ($TYPE$): it is represented by 3 categories indicating the series domain, *finances, micro-economy* and *macro-economy.*

The first four features are directly computed using the series data and TYPE in turn is an information provided by the TSDL repository.

4.2 Database

In the construction of the Database, meta-examples were generated from the empirical evaluation of the three candidate models on different time series forecasting problems. Each meta-example for the ZR is related to a time series and stores: (1) the descriptive features of the series (as defined in the FE module); and (2) the forecasting error and the execution time obtained by each candidate model, when used to forecast the series.

In this case study, the accuracy and execution time of each model were collected by performing a hold-out experiment using the available time series data. Initially, the time series data is divided in two parts: the fit period and the test period. The test period in our prototype corresponds to the last observations of the series and the fit period corresponds to the remaining data. The fit period is used to train the ANN models. The trained models are then used to generate its individual forecasts for the test period. Finally, each meta-example is then composed by: the time series features extracted for describing the fit period, the mean absolute forecasting error obtained by the models in the test period and the time execution recorded during the ANN models training.

The process described above was applied to 80 different time series, and hence, a set of 80 meta-examples was generated. We highlight that such set of meta-examples actually stores the experience obtained from empirically evaluating the ANN models to forecast a large number of different time series. A Meta-Learning approach (as the ZR) will be able to use this experience to recommend rankings of models for new series only based on the time series features, without the need of performing new empirical evaluations.

4.3 Meta-Learner

The Zoomed-Ranking (ZR) meta-learning approach was used in the Meta-Learner module in order to rank the three ANN models. This approach is composed by two distinct phases: the Zooming and the Ranking phases, described here. Given a new time series to forecast, in the Zooming phase, a number of m meta-examples are retrieved from the training set according to a distance function that measures the similarity between time series features. The distance function implemented in the prototype was the L_1-norm defined as:

$$dist(x, x_i) = \sum_{j=1}^{p} \frac{|x^j - x_i^j|}{max_l(x_l^j) - min_l(x_l^j)} \quad (1)$$

In this equation, x is the description of the input series to be forecasted , x_i is the description of the i-th series in the training set and p is the number of meta-features. We used in the implemented prototype the L_1-norm as originally proposed in the ZR approach [24].

In the Ranking phase, a ranking of models is suggested, by aggregating the forecasting error and execution time stored in the m retrieved meta-examples. This is performed by deploying the Adjust Ratio of Ratios (ARR) measure [7], as defined in the equation:

$$ARR_{k,k'}^i = \frac{\frac{S_i^{k'}}{S_i^k}}{1 + AccD * log(\frac{T_i^k}{T_i^{k'}})} \quad (2)$$

In the above equation, S_i^k and T_i^k are respectively the forecasting error and execution time obtained by the model k on series i. The metric $ARR_{k,k'}^i$ combines forecasting error and execution time, to measure the relative performance of the models k and k' in the series i. The parameter $AccD$ is defined by the user and represents the relative importance between forecasting accuracy and execution time. $AccD$ assumes values between 0 and 1. The lower is the $AccD$ parameter, the higher is the importance given to accuracy relative to execution time.

The ratio of forecasting errors $S_i^{k'}/S_i^k$ can be seen as a measure of advantage of the model k in relation to model k', that is, a measure of relative benefit of model k (the higher is $S_i^{k'}/S_i^k$, the lower is the forecasting error of model k relative to model k'). In turn, the ratio of execution times $T_i^k/T_i^{k'}$ can be seen as a measure of disadvantage of the model k in relation to model k', that is, as measure of relative cost of model k. The ARR measure uses the ratio between a benefit and a cost measure to compute the overall quality of the candidate model k related to k'.

An aspect that should be observed regarding the time ratio is the fact that this measure has a much wider range of possible values than the ratio of accuracy rate. Therefore, if simple ratios of time were used, it would dominate the ARR measure. In this way, the effect of this range could be diminished by using the log of time ratios. We highlight that the use of log of time ratios was also adopted in [24,7].

Finally, the ranking of models suggested to the input series is generated by aggregating the ARR information across the m retrieved meta-examples and K candidate models, as follows:

$$ARR_k = \frac{\sum_{k' \neq k} \sqrt[m]{\prod_{i \in Zoom} ARR^i_{k,k'}}}{K-1} \quad (3)$$

In the above equation, the $Zoom$ set represents the m retrieved meta-examples. The geometric mean in ARR is computed across the retrieved meta-examples and then the arithmetic mean across the candidate models. The ranking is suggested directly from ARR_k (the higher is the ARR_k value, the higher is the rank of model k). The geometric mean was used in order to satisfy the following property: $ARR_{k,k'} = 1/ARR_{k',k}$.

4.4 Experiments and Results

In the performed experiments, we collected 80 time series from the TSDL repository. Hence, a set of 80 meta-examples were generated by applying the procedure described in the section 4.2. This set was divided into 60 meta-examples for training and 20 meta-examples for testing the ZR approach.

The experiments were performed for different values of: (1) $AccD$ parameter (0, 0.2, 0.4 and 0.6), which controls the relative importance of accuracy and time; and (2) the parameter m (1, 3, 5, 7, 9, 11 and 13 neighbors), which defines the neighborhood size in the Zooming phase.

In order to evaluate the performance of ZR, we deployed the Spearman Ranking Correlation coefficient (SRC). Given a series i, the SRC coefficient measures the similarity between the recommended ranking of models and the ideal ranking (i.e. the correct ordering of models taking into account the ARR measure computed in the series). The SRC for a series i is computed as:

$$SRC_i = 1 - \frac{6 * \sum_{k=1}^{K} (rr_{k,i} - ir_{k,i})^2}{K^3 - K} \quad (4)$$

In the equation, $rr_{k,i}$ and $ir_{k,i}$ are respectively the rank of model k in the recommended ranking and the ideal ranking for the series i and K is the number of candidate models. SRC_i assumes values between -1 and 1. Values near to 1 indicate that the two rankings have many agreement positions and values near to -1 indicate disagreement between the rankings. In order to evaluate the rankings generated for the 20 series in the test set, we calculated the average of SRC across these series.

The ZR approach was compared to a default ranking method [18], in which the ranking is suggested by aggregating the performance information for all training meta-examples, instead of using only the most similar ones. Despite its simplicity, the default method has been used as a basis of comparison in different case studies in the literature of Meta-Learning [7,18,8].

Table 1 shows the average values of SRC across the test series, considering the ZR approach and the default ranking. As it can be seen, the rankings recommended by ZR were in average more correlated to the ideal rankings when compared to the default method. The SRC average values for the default ranking are near to zero, indicating neutrality related to the ideal rankings. In fact, the average performance of each candidate model was very similar across the 20 test series, and then there was no clear preference among the models by default. In this way, the default ranking had a quality which was similar to a random choice of models. The ZR in turn obtained SRC values from 0.45 to 0.70, for all different experimental settings, indicating positive correlation to the ideal rankings.

Table 1. Average SRC coefficient across the 20 series in the test set.

	Average SRC			
	AccD = 0.0	**AccD = 0.2**	**AccD = 0.4**	**AccD = 0.6**
m = 1	0.45	0.47	0.50	0.45
m = 3	0.47	0.50	0.52	0.50
m = 5	0.47	0.50	0.52	0.50
m = 7	0.47	0.50	0.52	0.50
m = 9	0.62	0.50	0.52	0.50
m = 11	0.67	0.70	0.67	0.50
m = 13	0.67	0.65	0.62	0.45
Default	0.02	0.05	0.07	0.05

5 Ranking Clustering Techniques for Gene Expression Data

As previously mentioned, Meta-Learning had been used mostly for ranking and selecting supervised learning algorithms. Motivated by this, we extended the use of Meta-Learning approaches for clustering algorithms. We developed our case study in the context of clustering algorithms applied to cancer gene expression data generated by microarray [9].

Cluster analysis of gene expression microarray data is of increasing interest in the field of functional genomics [41,42,43]. One of the main reasons for this is the need for molecular-based refinement of broadly defined biological classes, with implications in cancer diagnosis, prognosis and treatment. Although the selection of the clustering algorithm for the analysis of microarray datasets is a very important question, there are in the literature few guidelines or standard procedures on how these data should be analyzed [44,45].

The selection of algorithms is basically driven by the familiarity of biological experts to the algorithm rather than the features of the algorithms themselves and of the data [44]. For instance, the broad utilization of hierarchical clustering techniques is mostly a consequence of its similarity to phylogenetic methods,

which biologists are often used to. Hence, in this context, by employing a Meta-Learning approach, our aim was to provide a framework to support non-expert users in the algorithm selection task [9].

In this section, we present a case study originally proposed in [9] in which a Meta-Regression approach was used to rank seven different candidate clustering methods: single linkage (SL), complete linkage (CL), average linkage (AL), k-means (KM), mixture model clustering (M), spectral clustering (SP), and Shared Nearest Neighbors algorithm (SNN) [46,47,48]. As it will be seen, meta-examples were generated from the evaluation of these clustering methods on 32 microarray datasets of cancer gene expression. the next sections provide the details of implementation for this case study, as well as the performed experiments.

5.1 Feature Extractor

In this case study, we used a set of eight descriptive attributes (meta-features). Some of them were first proposed for the case of supervised learning tasks.

1. LgE: $\log_{10}$ of the number of examples. A raw indication of the available amount of training data.
2. LgREA: $\log_{10}$ of the ratio of the number of examples by the number of attributes. A rough indicator of the number of examples available to the number of attributes.
3. PMV: percentage of missing values. An indication of the quality of the data.
4. MN: multivariate normality, which is the proportion of T^2 [49](examples transformed via T^2) that are within 50% of a Chi-squared distribution (degree of freedom equals to the number of attributes describing the example). A rough indicator on the approximation of the data distribution to a normal distribution.
5. SK: skewness of the T^2 vector. Same as the previous item.
6. Chip: type of microarray technology used (either cDNA or Affymetrix).
7. PFA: percentage of the attributes that were kept after the application of the attribute selection filter.
8. PO: percentage of outliers. In this case, the value stands for the proportion of T^2 distant more than two standard deviations from the mean. Another indicator of the quality of the data.

5.2 Database

Meta-examples in this case study are related to cancer gene expression microarray datasets. Each meta-example has two parts: (1) the meta-features describing a gene expression dataset; and (2) a vector with the ranking of the clustering algorithms for that dataset. A meta-regressor will use a set of such meta-examples to predict the algorithms' ranks for new datasets.

In order to assign this ranking for a dataset, we executed each of the seven clustering algorithms with a given dataset to produce the respective partitions. The number of clusters was set to be equal to the true number of the classes in

the data. The known class labels was not used in any way during the clustering. As in other works, the original class labels constitute the *gold standard* against which we evaluate the clustering results [41,46,50].

For all non-deterministic algorithms, we ran the algorithm 30 times and picked the best partition. More specifically, in terms of the index to assess the success of the algorithm in recovering the gold standard partition of the dataset and building the ranking, we employed the corrected Rand index (cR) [46,50]. The maximum value of the cR is 1, indicating a perfect agreement between the partitions. A value near 0 corresponds to a partition agreement found by chance. A negative value indicates that the degree of similarity between the gold standard partition and the partition yielded by the clustering algorithm is inferior to the one found by chance.

The cluster evaluation we adopt is mainly aimed at assessing how good the investigated clustering method is at recovering known clusters from gene expression microarray data. Formally, let $U = \{u_1, \ldots, u_r, \ldots, u_R\}$ be the partition given by the clustering solution, and $V = \{v_1, \ldots, v_c, \ldots, v_C\}$ be the partition formed by an a priori information independent of partition U (the gold standard). The corrected Rand is defined as:

$$cR = \frac{\sum_i^R \sum_j^C \binom{n_{ij}}{2} - \binom{n}{2}^{-1} \sum_i^R \binom{n_{i\cdot}}{2} \sum_j^C \binom{n_{\cdot j}}{2}}{\frac{1}{2}[\sum_i^R \binom{n_{i\cdot}}{2} + \sum_j^C \binom{n_{\cdot j}}{2}] - \binom{n}{2}^{-1} \sum_i^R \binom{n_{i\cdot}}{2} \sum_j^C \binom{n_{\cdot j}}{2}}$$

where (1) n_{ij} represents the number of objects in clusters u_i and v_j; (2) $n_{i\cdot}$ indicates the number of objects in cluster u_i; (3) $n_{\cdot j}$ indicates the number of objects in cluster v_j; (4) n is the total number of objects; and (5) $\binom{a}{b}$ is the binomial coefficient $\frac{a!}{b!(a-b)!}$.

Based on the values of the cR, the ranking for the algorithms is generated as follows. The clustering algorithm that presents the highest cR come higher in the ranking (i.e., the ranking value is equal to 1). Algorithms that generate partition with the same cR receive the same ranking number, which is the mean of what they would have under ordinal rankings.

5.3 Meta-Learner

Our system generates a ranking of algorithms for each dataset given as input. In order to create a ranking of K candidates (clustering algorithms), we use K regressors, each one responsible for predicting the ranking of a specific algorithm for the dataset given as input.

For building the regressor associated to a given algorithm k, we adopt the following procedure. First, we defined a set of meta-examples. Each meta-example corresponded to a dataset, described by a set of meta-features, with one of them representing the desired output. The value of the meta-attribute representing the desired output is assigned according to the ranking of the algorithm among all the seven ones employed to cluster the dataset. Next, we applied a

supervised learning algorithm to each of the K regressors, which will be responsible for associating a dataset to a ranking.

As previously mentioned, we took into account seven clustering algorithms: SL, AL, CL, KM, M, SP and SNN. This led to construction seven regressors, $R_1, \ldots, R_7$, associated to, respectively, SL, AL, CL, KM, M, SP and SNN. For example, suppose that the outputs of the seven regressors for a new dataset are, respectively, 7, 5, 6, 1, 2, 4 and 3. Such an output means that model SL is expected to be the worst model (it is the last one in the ranking), AL is fifth best model model, CL the fourth one, KM is supposed to be better than all the others, as it is placed as first one in the ranking.

In our implementation, we employed the regression Support Vector Machine (SVM) algorithm, implemented in LIBSVM: a library for support vector machines [51]. A reason for this choice is that, in our preliminary results, SVMs showed a better accuracy than models such as artificial neural networks and k-NN.

5.4 Experiments and Results

We describe here the experiments that we developed in order to evaluate the performance of our prototype. Thirty two microarray datasets[3] (see Table 2). They are a set of benchmark microarray data presented in [52]. These datasets present different values for characteristics such as type of microarray chip (second column), number of patterns (third column), number of classes (fourth column), distribution of patterns within the classes (fifth column), dimensionality (sixth column), and dimensionality after feature selection (last column).

In terms of the datasets, it is worthwhile to point out that microarray technology is in general available in two different platforms, cDNA and Affymetrix [41,42,43]. Measurements of Affymetrix arrays are estimates on the number of RNA copies found in the cell sample, whereas cDNA microarrays values are ratios of the number of copies in relation to a control cell sample.

In order to remove uninformative genes for the case of Affymetrix arrays, we applied the following procedure. For each gene j (attribute), we computed the mean m_j. But before doing so, in order to get rid of extreme values, we discarded the 10% largest and smallest values. Based on this mean, we transform every value x^*_{ij} of example i and attribute j to:

$$y_{ij} = \log_2(x^*_{ij}/m_j)$$

After the previous transformation, we chose for further analysis genes whose expression level differed by at least l-fold, in at least c samples, from their mean expression level across samples. With few exceptions, the parameters l and c were selected in such a way as to produce a filtered dataset with around at least 10% of the original number of genes (features). It is important to point out that the data transformed with the previous equation is only used in the filtering step.

[3] http://algorithmics.molgen.mpg.de/Supplements/CompCancer/ are included in this analysis

Table 2. Dataset description

Dataset	**Chip**	n	**Nr Classes**	**Dist. Classes**	d	**Filtered** d
Alizadeh-V1	cDNA	42	2	21,21	4022	1095
Alizadeh-V2	cDNA	62	3	42,9,11	4022	2093
Armstrong-V1	Affy	72	2	24,48	12582	1081
Armstrong-V2	Affy	72	3	24,20,28	12582	2194
Bhattacharjee	Affy	203	5	139,17,6,21,20	12600	1543
Bittner	cDNA	38	2	19, 9	8067	2201
Bredel	cDNA	50	3	31,14,5	41472	1739
Chen	cDNA	180	2	104,76	22699	85
Chowdary	Affy	104	2	62,42	22283	182
Dyrskjot	Affy	40	3	9,20,11	7129	1203
Garber	cDNA	66	4	17,40,4,5	24192	4553
Golub-V1	Affy	72	2	47,25	7129	1877
Gordon	Affy	181	2	31,150	12533	1626
Khan	cDNA	83	4	29,11,18,25	6567	1069
Laiho	Affy	37	2	8,29	22883	2202
Lapoint-V1	cDNA	69	3	11,39,19	42640	1625
Lapoint-V2	cDNA	110	4	11,39,19,41	42640	2496
Liang	cDNA	37	3	28,6,3	24192	1411
Nutt-V1	Affy	50	4	14,7,14,15	12625	1377
Nutt-V2	Affy	28	2	14,14	12625	1070
Nutt-V3	Affy	22	2	7,15	12625	1152
Pomeroy-V1	Affy	34	2	25,9	7129	857
Pomeroy-V2	Affy	42	5	10,10,10,4,8	7129	1379
Ramaswamy	Affy	190	14	11,10,11,11,22,10,11 10,30,11,11,11,11,20	16063	1363
Risinger	cDNA	42	4	13,3,19,7	8872	1771
Shipp	Affy	77	2	58,19	7129	798
Singh	Affy	102	2	58,19	12600	339
Su	Affy	174	10	26,8,26,23,12,11,7,27,6,28	12533	1571
Tomlins-V1	cDNA	104	5	27,20,32,13,12	20000	2315
Tomlins-V2	cDNA	92	4	27,20,32,13	20000	1288
West	Affy	49	2	25,24	7129	1198
Yeoh-V1	Affy	248	2	43,205	12625	2526

A similar filter procedure was applied for the case of cDNA microarray, but without the need to transform the data. In the case of cDNA microarray datasets, whose attributes (genes) could present missing values, we discarded the ones with more than 10% of missing values. The attributes that are kept and still present missing values have the values replaced for the respective mean value of the attribute.

For a given dataset, in order to generate the ranking, we took into account the configuration that obtained the best corrected Rand (see the second and third paragraphs in Section 5.2). We ran the algorithms with Euclidean distance, Pearson correlation and Cosine, but always with the number of clusters equal to the real number of classes in the dataset.

We assessed the performance of the meta-learners using the leave-one-out procedure. At each step, 31 examples are employed as the training set, and the remaining example is used to test the SVMs created. This step is repeated 32 times, utilizing at each time a different test example. The quality of a suggested ranking for a given dataset is evaluated by employing the average SRC, as described in equation 4, to measure the similarity between the suggested and the ideal rankings.

The result of our approach was compared to a default ranking method, in which the average ranking is suggested for all datasets. In our case, the default ranking was: SL=6.41, AL=4.60, CL=3.84, KM=2.31, M=3.40, SP=3.07, SNN=4.36. In Table 3, we illustrate the mean and standard deviation for the Spearman coefficient for the rankings generated by our approach and for the default ranking.

Table 3. Mean of the Spearman coefficient

Method	**SRC**
Default	0.59 ± 0.37
Meta-Leaner	0.75 ± 0.21

As it can be seen, the rankings generated by our method were more correlated to the ideal ranking. In fact, according to a hypothesis test, at a significance level of 0.05, the mean of the correlation value found with our method was significantly higher than that obtained with the default ranking.

6 Conclusion

In this paper, we present the results of using Ranking Meta-Learning approaches in two different domains of application: time series forecasting and clustering of gene expression. In the performed experiments, we observed that the rankings suggested by Meta-Learning were similar to the ideal rankings of algorithms observed in the available problems.

We can point out specific contributions of our work in the fields of time series forecasting and clustering of gene expression, which are domains of particular

interest of many researchers. Different improvements can be considered in future work, such as increasing the number of meta-features (including the use of landmarking), investigating the viability of using artificial datasets in order to generate a larger database of meta-examples and performing experiments with other Meta-Learning approaches.

Finally, we highlight that Meta-Learning brings opportunities to researchers in different fields by providing general techniques that can be extrapolated to other algorithm selection tasks. Although the use of Meta-Learning in different domains has been increasing in recent years, there is still a large number of contexts in which it has not yet been investigated.

Acknowledgments. The authors would like to thank CNPq, CAPES and FACEPE (Brazilian Agencies) for their financial support.

References

1. Kalousis, A., Gama, J., Hilario, M.: On data and algorithms - understanding inductive performance. Machine Learning 54(3), 275–312 (2004)
2. Giraud-Carrier, C., Vilalta, R., Brazdil, P.: Introduction to the special issue on meta-learning. Machine Learning 54(3), 187–193 (2004)
3. Vilalta, R., Drissi, Y.: A perspective view and survey of meta-learning. Journal of Artificial Intelligence Review 18(2), 77–95 (2002)
4. Koepf, C.: Meta-Learning: Strategies, Implementations, and Evaluations for Algorithm Selection. Infix (2006)
5. Smith-Miles, K.: Cross-disciplinary perspectives on meta-learning for algorithm selection. ACM Computing Surveys 41(1), 1–25 (2008)
6. Brazdil, P., Giraud-Carrier, C., Soares, C., Vilalta, R.: Metalearning: Applications to Data Mining. In: Cognitive Technologies. Springer, Heidelberg (2009)
7. Brazdil, P., Soares, C., da Costa, J.: Ranking learning algorithms: Using IBL and meta-learning on accuracy and time results. Machine Learning 50(3), 251–277 (2003)
8. dos Santos, P., Ludermir, T.B., Prudêncio, R.B.C.: Selection of time series forecasting models based on performance information. In: 4th International Conference on Hybrid Intelligent Systems, pp. 366–371 (2004)
9. de Souto, M.C.P., Prudencio, R.B.C., Soares, R.G.F., Araujo, D.A.S., Costa, I.G., Ludermir, T.B., Schliep, A.: Ranking and selecting clustering algorithms using a meta-learning approach. In: Proceedings of the International Joint Conference on Neural Networks. IEEE Computer Society, Los Alamitos (2008)
10. Jankowski, N., Grabczewski, K.: Building meta-learning algorithms basing on search controlled by machine complexity. In: IJCNN, pp. 3601–3608 (2008)
11. Duch, W.: What is computational intelligence and where is it going? In: Duch, W., Mandziuk, J. (eds.) Challenges for Computational Intelligence. Springer Studies in Computational Intelligence, vol. 63, pp. 1–13. Springer, Heidelberg (2007)
12. Engels, R., Theusinger, C.: Using a data metric for preprocessing advice for data mining applications. In: Prade, H. (ed.) Proceedings of the 13th European Conference on Artificial Intelligence (ECAI- 1998), pp. 430–434. John Wiley & Sons, Chichester (1998)
13. Pfahringer, B., Bensusan, H., Giraud-Carrier, C.: Meta-learning by landmarking various learning algorithms. In: Proceedings of the 17th International Conference on Machine Learning, ICML 2000, pp. 743–750. Morgan Kaufmann, San Francisco (2000)

14. Aha, D.: Generalizing from case studies: A case study. In: Proceedings of the 9th International Workshop on Machine Learning, pp. 1–10. Morgan Kaufmann, San Francisco (1992)
15. Kalousis, A., Hilario, M.: Representational issues in meta-learning. In: Proceedings of the 20th International Conferente on Machine Learning, pp. 313–320 (2003)
16. Leite, R., Brazdil, P.: Predicting relative performance of classifiers from samples. In: 22nd Inter. Conf. on Machine Learning (2005)
17. Prudêncio, R.B.C., Ludermir, T.B., de Carvalho, F.A.T.: A modal symbolic classifier to select time series models. Pattern Recognition Letters 25(8), 911–921 (2004)
18. Prudêncio, R.B.C., Ludermir, T.B.: Meta-learning approaches to selecting time series models. Neurocomputing 61, 121–137 (2004)
19. Michie, D., Taylor, D.J.S. (eds.): Machine Learning, Neural and Statistical Classification. Ellis Horwood, New York (1994)
20. Kalousis, A., Theoharis, T.: Noemon: Design, implementation and performance results of an intelligent assistant for classifier selection. Intelligent Data Analysis 3, 319–337 (1999)
21. Kalousis, A., Hilario, M.: Feature selection for meta-learning. In: Cheung, D., Williams, G.J., Li, Q. (eds.) PAKDD 2001. LNCS (LNAI), vol. 2035, p. 222. Springer, Heidelberg (2001)
22. Bensusan, H., Alexandros, K.: Estimating the predictive accuracy of a classifier. In: 12th European Conf. on Machine Learning, pp. 25–36 (2001)
23. Koepf, C., Taylor, C.C., Keller, J.: Meta-analysis: Data characterisation for classification and regression on a meta-level. Proceedings of the International Symposium on Data Mining and Statistics (2000)
24. Soares, C., Brazdil, P.B.: Zoomed ranking: Selection of classification algorithms based on relevant performance information. In: Zighed, D.A., Komorowski, J., Żytkow, J.M. (eds.) PKDD 2000. LNCS (LNAI), vol. 1910, pp. 126–135. Springer, Heidelberg (2000)
25. Tsoumakas, G., Vrakas, D., Bassiliades, N., Vlahavas, I.: Lazy adaptive multicriteria planning. In: Proceedings of the 16th European Conference on Artificial Intelligence, ECAI 2004, pp. 693–697 (2004)
26. Smith-Miles, K.: Towards insightful algorithm selection for optimisation using meta-learning concepts. In: Proceedings of the IEEE International Joint Conference on Neural Networks, pp. 4118–4124 (2008)
27. Caiuta, R., Pozo, A.: Selecting software reliability models with a neural network meta classifier. In: Proceedings of the Joint International Conference on Neural Networks (2008)
28. Nascimento, A.C.A., Prudêncio, R.B.C., de Souto, M.C.P., Costa, I.G.: Mining rules for the automatic selection process of clustering methods applied to cancer gene expression data. In: Alippi, C., Polycarpou, M., Panayiotou, C., Ellinas, G. (eds.) ICANN 2009. LNCS, vol. 5769, pp. 20–29. Springer, Heidelberg (2009)
29. Souza, B., Soares, C., Carvalho, A.: Meta-learning approach to gene expression data classification. International Journal of Intelligent Computing and Cybernetics 2, 285–303 (2000)
30. Soares, C.: UCI++: Improved support for algorithm selection using datasetoids. In: Theeramunkong, T., Kijsirikul, B., Cercone, N., Ho, T.-B. (eds.) PAKDD 2009. LNCS, vol. 5476, pp. 499–506. Springer, Heidelberg (2009)
31. Adya, M., Collopy, F., Armstrong, J., Kennedy, M.: Automatic identification of time series features for rule-based forecasting. International Journal of Forecasting 17(2), 143–157 (2001)

32. Montgomery, D.C., Johnson, L.A., Gardiner, J.S.: Forecasting and Time Series Analysis. MacGraw Hill, New York (1990)
33. Tashman, L.J.: Out-of-sample tests of forecasting accuracy: An analysis and review. International Journal of Forecasting 16, 437–450 (2000)
34. Prudêncio, R.B.C., Ludermir, T.B.: Selection of models for time series prediction via meta-learning. In: Proceedings of the Second International Conference on Hybrid Systems, pp. 74–83. IOS Press, Amsterdam (2002)
35. Arinze, B.: Selecting appropriate forecasting models using rule induction. Omega-International Journal of Management Science 22, 647–658 (1994)
36. Prudêncio, R.B.C., Ludermir, T.B.: A machine learning approach to define weights for linear combination of forecasts. In: 16th International Conference on Artificial Neural Networks, pp. 274–283 (2006)
37. Lang, K.J., Hinton, G.E.: A time-delay neural network architecture for speech recognition. Technical Report CMU-DS-88-152, Dept. of Computer Science, Carnegie Mellon University, Pittsburgh (1988)
38. Levenberg, K.: A method for the solution of certain non-linear problems in least squares. Quarterly Journal of Applied Mathmatics II, 164–168 (1944)
39. Haykin, S.: Neural Networks: A Comprehensive Foundation. Macmillan College Publishing Company, New York (1994)
40. Principe, J., Euliano, N., Garania, S.: Principles and networks for self-organization in space-time. Neural Networks 15, 1069–1083 (2002)
41. Monti, S., Tamayo, P., Mesirov, J., Golub, T.: Consensus clustering: a resampling-based method for class discovery and visualization of gene expression microarray data. Machine Learning 52, 91–118 (2003)
42. Quackenbush, J.: Computational analysis of cDNA microarray data. Nature Reviews 6, 418–428 (2001)
43. Slonim, D.: From patterns to pathways: gene expression data analysis comes of age. Nature Genetics 32, 502–508 (2002)
44. D'haeseleer, P.: How does gene expression clustering work? Nature Biotechnology 23, 1499–1501 (2005)
45. de Souto, M.C., Costa, I.G., de Araujo, D.S., Ludermir, T.B., Schliep, A.: Clustering cancer gene expression data: a comparative study. BMC Bioinformatics 9, 497 (2008)
46. Jain, A.K., Dubes, R.C.: Algorithms for clustering data. Prentice Hall, Englewood Cliffs (1988)
47. Xu, R., Wunsch, D.: Survey of clustering algorithms. IEEE Transactions on Neural Networks 16, 645–678 (2005)
48. Ertoz, L., Steinbach, M., Kumar, V.: A new shared nearest neighbor clustering algorithm and its applications. In: Workshop on Clustering High Dimensional Data and its Applications, pp. 105–115 (2002)
49. Johnson, R.A., Wichern, D.W.: Applied Multivariate Statistical Analysis, 5th edn. Prentice-Hall, Englewood Cliffs (2002)
50. Milligan, G.W., Cooper, M.C.: A study of standardization of variables in cluster analysis. Journal of Classification 5, 181–204 (1988)
51. Chang, C.C., Lin, C.J.: LIBSVM: a library for support vector machines, Software (2001), `http://www.csie.ntu.edu.tw/~cjlin/libsvm`
52. de Souto, M.C.P., Costa, I.G., Araujo, D.S.A., Ludermir, T.B., Schliep, A.: Clustering cancer gene expression data - a comparative study. BMC Bioinformatics 9, 497–520 (2008)

A Meta-Model Perspective and Attribute Grammar Approach to Facilitating the Development of Novel Neural Network Models

Talib S. Hussain

Raytheon BBN Technologies, Cambridge, MA, USA

Abstract. There is a need for methods and tools that facilitate the systematic exploration of novel artificial neural network models. While significant progress has been made in developing concise artificial neural networks that implement basic models of neural activation, connectivity and plasticity, limited success has been attained in creating neural networks that integrate multiple diverse models to produce highly complex neural systems. From a problem-solving perspective, there is a need for effective methods for combining different neural-network-based learning systems in order to solve complex problems. Different models may be more appropriate for solving different subproblems, and robust, systematic methods for combining those models may lead to more powerful machine learning systems. From a neuroscience modelling perspective, there is a need for effective methods for integrating different models to produce more robust models of the brain. These needs may be met through the development of meta-model languages that represent diverse neural models and the interactions between different neural elements. A meta-model language based on attribute grammars, the Network Generating Attribute Grammar Encoding, is presented, and its capability for facilitating automated search of complex combinations of neural components from different models is discussed.

1 Introduction

Since the resurgence of the neural networks field in the mid-1980's, thousands of different neural network models have been produced. These models vary along many different dimensions, including the basic elements and properties of a neuron or synapse, properties of connections, patterns of connectivity, the degree of structural and functional modularity, temporal and spatial properties of network behaviour, mechanisms used to propagate information, and mechanisms used to adapt the network behaviour over time [1]. Despite significant progress in the field, models developed to date have tended to be relatively limited in scale and complexity. For example, neural networks for learning tend to use simple learning rules and relatively small structures with limited modularity, while neural networks for biological modeling tend to focus on detailed models of small numbers of neurons or small areas of the brain. These limitations have been largely due to the general use of manual methods for creating new models and the lack of automated methods for adapting and combining available model capabilities.

N. Jankowski et al. (Eds.): Meta-Learning in Computational Intelligence, SCI 358, pp. 245–272.
springerlink.com

While there are many software tools available for assisting researchers in the creation of neural networks, these tools have historically been of limited use in the development of new, large complex models. Neural network simulation environments [2-11] and specification languages [12-19] offer the researcher the capability to specify a wide range of neural architectures, but variations in those architectures must generally be explored in a highly manual fashion. Genetic encodings of neural networks [20-32] offer the researcher the capability to specify a variety of neural properties and to use evolutionary search to systematically explore new networks that combine those properties in novel ways, but are generally limited to a small number of structural or functional dimensions [20-28].

Most model specifications lack explicit, formal descriptions of the roles of the different neural components. In creating a model, researchers often make design decisions based on the roles that certain components may play. Certain types of neurons may be chosen to play a supervisory role for other neurons. One neural component based on a particular model may be chosen to process the input data in a certain way, such as clustering it, to allow a second neural component to perform more efficiently in a different role, such as classifying the data. A specific feature may be incorporated into a model to play an analogous role to a particular structure in the brain. While researchers understand that the neural elements are intended to perform certain roles, those roles are rarely explicitly captured in the model itself. Rather, they are buried in the prose that describes the model (if at all). Thus, when we have two different neural components that play the same or similar roles in two different network models, it requires a manual to recognize that overlap, and further manual effort to determine how to adapt one component (or interchange them) in order to explore whether one model can be improved by adopting techniques or principles from the other.

The author proposes that there is a need for representations that explicitly describe the roles of different elements of a model, and a new perspective on how to develop novel models by exploiting those roles in a systematic manner. In particular, it is proposed that there is a need for general, hierarchical methods for representing arbitrary, pertinent elements of a neural network model at sufficient levels of detail to permit exploration of model variations based on those elements. These methods should allow a single hierarchical representation to capture low-level details such as the internal functioning of a single neuron, mid-level details such as how neurons and neural components are organized structurally and functionally within a network, and high-level details such as how different network components interact with each other, what roles they play, and how they interact with their environment.

This chapter introduces a meta-model perspective on model creation, and presents the Network Generating Attribute Grammar Encoding (NGAGE) [29-32] as a meta-model representation method based on attribute grammars [33]. NGAGE supports the representation of diverse neural network models and enables the systematic exploration of new models.

2 Background

The wealth of existing neuron models and neural network architectures, and the functional and structural elements that make up those models provide a rich foundation of

material and techniques for creating new models. Many software tools and representation methods have been developed to support research on neural network modelling. These have typically fallen into one of four categories.

Single-model focus: Many software tools provide support for creating a single type of neural network model. In these tools, the researcher is limited to varying parameters of the model and little else. Many genetic encoding approaches [20-22] fall into this category. A number of techniques for the genetic encoding of neural networks have been developed with the aim of providing the capability to represent some properties of a neural network model in a manner that enables an evolutionary algorithm to automatically manipulate those properties in interesting ways. Early genetic encodings used highly static and limited representations that enabled the exploration of few neural properties and simple networks. For instance, direct encoding [20] specifies only the weight values of a network, while all other aspects of the network, including topology, are kept fixed. A parametric encoding [21, 22] may specify a wide variety of network properties, but usually these are limited in number and to high-level properties.

Specification focus: Some approaches provide the ability to create very detailed models of neural elements from first principles [9-11, 17-18, 36], or to specify a variety of elements from an underlying language of neural components [12-14]. This provides strong modelling power to the researcher to create new models, but requires significant manual effort (e.g., coding or visual coding).

Generative focus: Some tools provide a simple set of basic neural building blocks, usually a single neuron model with limited parameters, a single type of connection, and a limited set of variations on a learning rule [15]. In these tools, the researcher is free to systematically compose new networks within a limited family. These tools can be very useful in generating new solutions to problems, but typically offer few new insights into learning or biology, and often do not scale to large solutions due to limitations in the underlying neural model. Most remaining genetic encodings approaches [23-28] fall into this category. For instance, developmental rule encoding [23, 24] and cellular encoding [25] represent a network as a set of re-write rules that may be applied to an initial network configuration to produce a final topology that may range from sparse to dense and highly irregular to highly regular or modular; functional aspects of the neurons generally remain fixed. Standard tree-based, genetic programming encoding has been used to evolve the form of the learning rule from simpler functional elements, but assumes a fixed network topology [26].

Model toolbox focus: A number of tools provide the researcher with the ability to create many different types of neural networks, typically based heavily on popular neural network models in the field [3-7], or, more generally, arbitrary machine learning algorithms [8]. While some of these suffer the same limitations as single-model tools (i.e., no ability to move beyond the original models), many provide the ability to combine different models in their entirety (e.g., connect them in sequence) or to combine certain parts of models (e.g., select learning rule from available set). This permits an exploration of new models by combining and varying different elements from different models. In most such tools, however, researchers are required to manually

explore the space of model variations (e.g., by creating models via a visual interface, writing a specific script or programming).

Generally, creating new neural network models is an ad-hoc endeavour, with the focus being more on using consistent approaches to empirically verify the effectiveness of a given model than on the process of creating the model. In the area of neural networks for learning, the practices followed in creating a new, useful neural model are rarely addressed as a topic in their own right. Some researchers do use ongoing developments in computational neuroscience or theories from related fields to guide the choice of model variations [57,58], but many simply make minor changes to activation functions, structure or learning rules. Researchers on the evolution of neural networks have paid a high degree of explicit attention to representing not only the structure and functions of neural networks, but also the rules by which they may develop [20-27]. Most of these genetic encodings facilitate the programmatic exploration of model variations, but ultimately, support the search only of limited families of networks. In the area of neural networks for biological modelling, there is a greater explicit attention paid to the methods used for creating new models [49-50]. Further, advances in neuroscience quickly inform the types of model variations that are worthy of exploration. However, the process is almost exclusively manual in nature, and the intricate models developed by one researcher are generally not compatible with models developed by another researcher, nor easily shared among researchers [17,18].

Model sharing focus: Several efforts have been made to define standard description languages to address the difficulty in sharing models among researchers. The Predictive Model Markup Language (PMML) [19] is intended to specify data mining models using a wide variety of machine learning approaches so that researchers may develop models within one modeling application and use the models in another application. PMML defines the data requirements and pre-processing, model specifics, data mining schema and output post-processing used by a data mining model. Much of the specification describes how to apply a known machine learning algorithm in context. In a similar manner to a model toolbox, model-specific structural and functional details are defined based on the parameters expected by an underlying algorithm-specific module. For instance, in PMML, a multi-layer feedforward network is represented by its activation function, number of layers, connections and weights.

The Network Interchange format for Neuroscience Markup Language (NineML) is intended to specify networks of biological neural models so that different researchers may be able to replicate the results of others [17,18]. NineML is comprised of two semantic layers – an abstraction layer that provides the "the core concepts, mathematics and syntax with which model variables and state update rules are explicitly described" [18] and the user layer that "provides syntax to specify the instantiation and parameterization of a network model in biological terms". NineML is under development for the limited domain of spiking networks, and is intended to be extensible and to cover a range of modelling scales. However, since NineML is not yet available, its strengths and limitations are not known.

Thus, while there is support for the systematic exploration of relatively simple neural network models and for the manual creation of highly complex models, few researchers have considered how to support the automated search of complex, heterogenous networks. Specifically, a key issue is how to represent neural elements at

multiple levels of complexity in such a way that a search can meaningful substitute certain elements for others and/or compose certain elements with others. In other words, there is a lack of representations that identify neural elements based on a meta-level understanding of their role in the system, thereby allowing those elements with similar roles to be interchanged, and elements with complementary capabilities to be integrated together into a larger component.

3 Meta-Perspective on Model Creation

The author proposes that we need to develop and espouse a common meta-perspective on model creation. The perspective should address questions such as:

- What is the process involved in developing a new neural-based model?
- What should we consider when trying to enhance an existing model?
- What are the different ways in which neural components may interact within a model?
- How do we identify and characterize the common elements of different models?
- What are the dimensions along which we can change a given model?
- How do we formally define the meta-description of a model to capture the basic reasoning and process followed in creating that model?
- How do we relate one level of abstraction in a model to another level?

As a step towards a common meta-perspective, the following multi-level approach to understanding and describing the neural modelling process is proposed. In particular, we define six levels of specification which may be used to interpret a given neural element: Environmental, Behavioural, Developmental, Structural, Functional and Mechanical. The subtle aspect regarding such levels is to recognize that they may apply in a hierarchical fashion throughout a neural model. While all levels may not apply to all neural elements, most neural elements may be viewed on at least three levels. In using the term 'neural element', we allow it to refer to any part of a neural model that may be defined and hence manipulated by the researcher. The levels are defined below, together discussions of how they may be used as part of an explicit modeling representation.

3.1 Environmental Level

At the highest level of specification, a neural element may be regarded in terms of the environment in which it exists and with which it interacts. This environment should define the context in which the neural element operates, and should ground the motivation for the design of the neural element. Typically, for most models the environmental level is considered only in terms of the most general context (which we shall term the "external environment"). For example, a neural network for learning may be defined to solve a classification problem and a description of the inputs and outputs may be given. Moreover, that environmental context may not be explicit in the model

definition, and hence the specific relationship between the "problem" and the choice of neural elements is not captured.

The author proposes that the environmental context of a neural element should be explicitly defined for all meaningful parts of a neural model. For example, a module within a larger neural system has a context in which it is operating. This context may not be the same as the external environment. It may differ in complexity (e.g., the number of inputs may have been reduced by another module), time properties (e.g., external signals may be infrequent, but internal signals provided to the module may be very frequent), nature (e.g., external signals may be real-valued, while internal signals to the module may be binary), or stability (e.g., the external patterns may vary dramatically over time, but the internal patterns to the module may remain relatively similar).

The nature of the context in which a neural element is operating greatly informs the choices made at more detailed levels of specification. Explicitly capturing this context for arbitrary neural elements in the model allows enhanced programmatic comparison of elements. Programmatic exploration methods could then determine whether two elements can be meaningfully interchanged (e.g., because of similarities in underlying environmental assumptions, such as a given hormone concentration). Those methods could also automatically determine how a given context would need to be adapted if the assumptions were different in order to support an effective interchange (e.g., through the use of yet another neural element to modulate a hormone).

3.2 Behavioural Level

A given model operating within a given environment is designed to satisfy a certain set of goals. These may be intrinsic to the model, such as performance goals (e.g., the purpose of a learning network may be to produce an answer with minimal error, or it may be to process inputs in minimal time), or they may be extrinsic, such as a research goal (e.g., the goal of designing a complex neuron model may be to emulate particular known neural behaviour as closely as possible). Moreover, different neural elements may be incorporated into a model in order to serve different purposes and accomplish different goals. Ultimately, every neural element in a model is chosen for a reason, even if that reason is simply tradition.

When developing a model, the goal of a given element may be captured explicitly in terms of its intended role or behaviour. How does one neural component interact with or impact another? What role does it play in determining the behaviour of the larger system? At this level of specification, the researcher defines the roles for a given network and its components, and organizes those roles as appropriate to provide a general specification of the processing that will occur in the network. The behavioural level is given here as a higher level than the developmental level, though there may be cases in which a researcher may wish for the role of a neural element to change over time as the network adapts. In such a case, a robust behavioural level representation should cover all possible roles the neural element may adopt over time.

3.3 Developmental Level

Given a certain role with certain goals in a certain environment, a model must specify how all neural elements will form or adapt to achieve those goals. Even for rigid,

static network models, there are generally basic heuristics that govern the formation of and changes to the network. The developmental level is captured explicitly in most models for certain neural elements. For example, a connection pruning algorithm or a network growing rule. It is important to realize, though, that all neural elements potentially have their own approaches to formation and change over time. This becomes especially true as we begin to explore complex networks of complex and varied neurons. Simple learning rules, such a weight adaptation rule, conceptually fall in the developmental level, though they reflect a simple application of a function and a memory store.

3.4 Structural Level

A neural element may be regarded as a particular configuration of multiple simpler neural elements. These may be arranged in a particular structural and/or functional manner to produce the aggregate component. The structural level is one of the central levels of representation used when specifying neural models, and is often applied in a hierarchical manner, such as in modular networks. Note that a neural element may be specified in a highly abstract manner (e.g., a basic Hebbian neuron model [59]) or in a highly detailed structural and functional manner (e.g., a complex biological neuron model) [60-62]. A robust meta-modelling language should support either level of detail for different elements in the model.

3.5 Functional Level

Within any neural model, we may view a given neural element as performing a basic function. Thus, a neuron may have a particular activation function. A synapse may be governed by a particular set of functions. At this level, making a change to a model must address the specific types of processing that will occur, and what computations will be made. Issues of timing and transmission of information are addressed at this level. The basic functionality of a neural element may generally be specified in a highly localized manner. However, it can also be regarded in a hierarchical manner (e.g., a basis function lies within another function). Despite its local specification, the impact of a change in functionality may have far-reaching effects on the high-level behaviour of a larger neural component. The functional level is one of the common levels of representation used when specifying neural models, though it is usually used only to describe the functionality of single neurons.

3.6 Mechanical Level

At the lowest level, a neural element may be regarded as executing some basic mechanical operation. In most existing models, this operation is described as a set of mathematical equations or an algorithm, and changes to the operation may include varying specific values in an equation (e.g., constants or coefficients) or trying a new way of computing a function. At this level, the key concern of the modeller is to ensure that the operation is described correctly so that it performs the intended function specified at the higher level.

As an example of applying these levels of specification in defining a model, consider the representation of a connection in a neural network for learning.

Environmental: The connection's environment may be stable so that the neurons it is connecting never change, or it may be dynamic with its neurons changing over time. If the connection is modulated by external factors, such as a hormone or modulating connection, those elements would form part of the connection's environment.

Behavioural: A connection's role may be to transmit a particular type of signal from one neuron to another with a certain degree of attenuation based on distance, and then to be quiescent for a period of time. Or, its role may be to emulate a specific connectivity behaviour identified by neuroscientists.

Developmental: A connection may be formed at the beginning and never change. Or, it may change its transmission characteristics over time. A typical weight learning rule may be considered as part of the developmental level since it changes the properties of the connection over time.

Structural: A connection's structure may be the specific source and destination neurons of a connection, along with its weight. In another model, characteristics such as length or transmission speed may also be part of its structure.

Functional: The function used to determine a connection's transmitted signal may be specified at this level. For instance, a connection may use its weight value to modulate the signal.

Mechanical: The specific weighting equation may be specified at this level.

Many software tools and representation methods have described neural models, in varying degrees of detail, on the four lowest levels. Explicit representation of the two highest levels is rare. In particular, capturing the role of neural components is an important capability that requires attention if we are to facilitate robust automated search of diverse model variations.

4 Roles in Neural Networks

There are several lines of research from which we may draw inspiration in identifying the different types of roles that neural components may play within a neural network model. Researchers on hybrid symbolic-neural approaches [37,38], high-level descriptions of neural networks [35] and modular neural networks [43-45] have made various forays into defining and/or representing different roles that may be played by neural components within a larger system. Researchers interested in interoperability of diverse neural models [29-32, 39-42] and diverse machine learning methods [46-48] have identified additional roles that may be played by different components. These roles include:

Encode a decision rule provided by another component: In hybrid methods such as the Knowledge-Based Artificial Neural Network [38], a neural component may encode a rule provided by another component.

Transform inputs into higher-level representation: [37] describes several hybrid systems in which a neural component transforms its inputs into a higher-level

(usually symbolic) representation, which is then used by other components in the system. For example, producing the antecedent and/or consequent of a higher-level rule, or transforming a lower-level input pattern into its corresponding higher-level concept.

Store knowledge for other components: A neural component may be a memory store for other components [37,52].

Assign confidence to another module; Determine when another module learns; Specialize on subset of input patterns: In the Minos modular neural network model [43], neural components play three distinct roles. A *worker* submodule performs the role of specializing by learning certain input patterns (using back-propagation learning upon the input). An *authority* module performs the role of helping a Minos module specialize by selecting only one module to undergo training (based on the module with the lowest error). A *monitor* submodule performs the role of assigning confidence ratings to its associated worker module. During a recall event for a pattern, every module is activated upon the pattern and the authority selects the response of the Minos module with the highest self-confidence measure as the output of the network.

Combine functions of sub-components into a higher-level function or role: [33] describes a perspective that may be taken on how neural network elements may be represented hierarchically. As neural components are combined, they may come to represent something meaningful at a higher-level of abstraction. While this is somewhat obvious, it allows us to capture, for example, the notion of an 'optic nerve' at a high-level level, compared with individual axons at a low-level.

Modulate the output of other modules; Integrate outputs of modules: In a Mixture of Experts network [44], components play three different roles (see Figure 1). The *expert modules* perform a traditional learner role (using back-propagation learning upon the input). The *gating module* performs a modulating role by outputting a probability for each expert module indicating whether it represents the correct output of the network. The *integrator node* performs the role of integrating the outputs of the other modules (by computing the sum of the outputs of expert modules weighted by probabilities from the gating module).

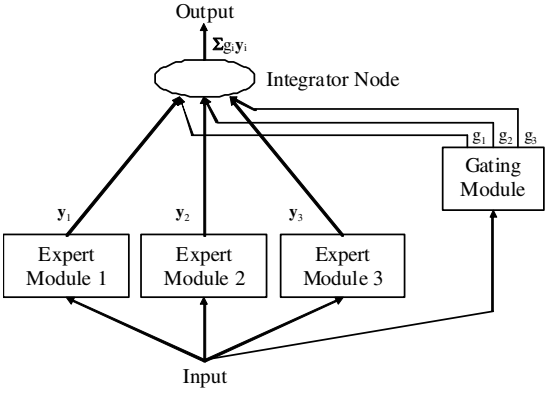


Fig. 1. Mixtures of Experts Network

Cluster data for another module; Simplify inputs for another module; Classify the outputs of another module: The counter-propagation network [45] consists of two modules connected serially. A self-organizing map module performs the roles of clustering the data (into equiprobable sections) and, usually, reducing the dimensionality of that input to simplify the learning required by the instar module. The instar module classifies each cluster by associating it with a vector representing the average desired response for that cluster.

Decompose problem; solve sub-problem: As problems get complex, it becomes important to decompose them into manageable sub-problems, and to solve those sub-problems using specialized modules. Problem decomposition is generally performed through partitioning of the input vector, specialization on a portion of the input space [43], or via explicit choices of the models used and explicit arrangement (e.g., sequencing) of those models [45]. [31] presents a generative technique in which the problem is decomposed by using a genetic search to both partition the input vector and to determine the modular arrangement for each partition.

Solve a step in a problem-solving process: [47, 48] presents an approach in which different machine learning algorithms solve certain steps within a larger problem-solving process. The results of applying a given learning algorithm at a given part of the process are made available for subsequent steps. The process may be rigidly defined (e.g., a fixed sequence of steps are followed) or may be open-ended (e.g., processing continues through iterative refinement until a solution is reached).

One component provides explicit instruction to another: Neural models often have some structures which modulate the learning of other elements. For instance, the output layer in a back-propagation network provides supervisory feedback to the hidden layer. [39] posits that neural methods may benefit from considering neural analogues of the wide variety of learning relationships that exist among intelligent creatures in nature. For instance, one neural module may learn from observing the behaviour of a different module. A module may pass on its mistakes to another module to assist it in avoiding them. Other forms of explicit learning relationships among neural components could include analogues of mentoring, mimicry, parenting, collaborative learning, and more. [46] posits that advanced learning systems may be achieved by developing new approaches that enable different machine learning components to learn from each other through the use of 'natural instruction' methods, such as instructing by demonstration, by worked example, or by analogy.

Hypothesis formation, evaluation and integration: In a framework presented by [47, 48], multiple machine learning paradigms are integrated by casting different algorithms into one of three different roles. Hypothesis formers generate hypotheses to explain some part of the problem being learned. Hypotheses evaluators test the validity of those hypotheses as best as possible according to their capabilities, so long as they are able to process a given hypothesis. Given a set of verified or partially verified hypotheses, other algorithms perform the role of integrating those hypotheses and resolving any ambiguities to produce a solution.

Role: Regulate the behaviour of other components: A variety of neural elements play the role of changing the behaviour of other neural elements. In [53], certain elements interpret the novelty of an input and, in turn, adapt the learning behaviour of other elements – learning is increased for new patterns and decreased for previously learned patterns. [54,55] explore how neural and control behaviour may be moderated using a hormone-based mechanism. In some computational neuroscience models (e.g., [56]), changes to one neural element (e.g., density and distribution of ion channels) can significantly change the behaviour of a larger neural element (e.g., change a neuron from non-spiking to spiking, and with different firing patterns).

The roles above are only some of those that neural elements may play within a complex network. Further, as we progress from the lowest levels of neural elements to the highest level of complex neural structures within a network, roles that may be played at a given level may vary from roles at other levels, even if based upon similar principles. For example, neural inhibition at the level of a small number of neurons may play a simple role (e.g., ensuring convergence in a neural map), whereas inhibition of one high-level component over another may have a different behavioural interpretation (e.g., selecting the appropriate modality to use to process the input).

5 Network Generating Attribute Grammar Encoding (NGAGE)

The Network Generating Attribute Grammar Encoding (NGAGE) [29-32] is a model representation method that explicitly attempts to broaden the scope of programmatic search of model variations. The central premise of NGAGE is that an arbitrarily rich family of neural models may be explored if we define a searchable language that can describe all key structural and functional aspects of multiple neural network models, and that can identify similar elements across different models. The basic NGAGE approach is presented, and its capabilities and techniques for supporting all six levels of specification for meta-modelling are described. In particular, the ability of NGAGE to support explicit specification of the roles of neural elements at the behavioural level is discussed.

NGAGE uses several tiers of representation and interpretation to accomplish its goals, as illustrated in Figure 2

1. An attribute grammar [33] is used to define the architectural elements - the structural, functional and learning behaviours – of a family of models in a highly modular and hierarchical manner, and the rules for creating those models from those elements.
2. A context-free parse tree generated from the attribute grammar is used to represent a particular instance of a neural architecture from the family of possible architectures the grammar defines.
3. The attributes of the productions are used to compute a complete specification of a neural network for a given parse tree.
4. The specification of the network is defined in the Generic Neural Markup Language (GNML) [32], an XML-based representation [63].

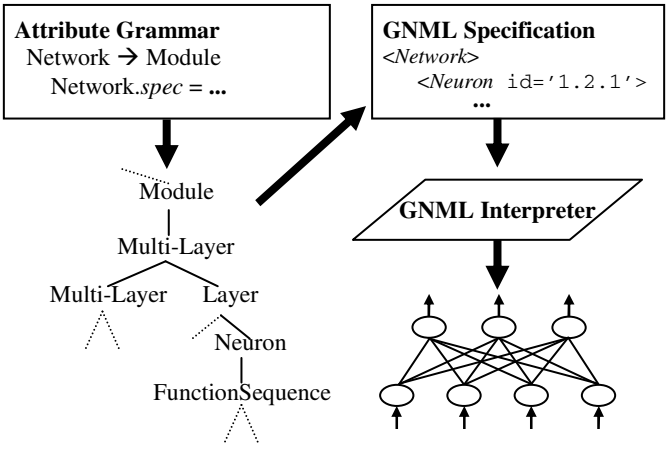


Fig. 2. NGAGE representation and interpretation method

5. The GNML specification is executed by an interpreter to produce a functional neural network.

An attribute grammar consists of a context-free grammar in which each symbol in a grammar production has a set of attributes associated with it, and each production has a set of attribute evaluation rules that specify how the attributes of its right hand symbols and/or left hand symbol are to be computed. The attributes of a given symbol may be partitioned into two disjoint sets: *inherited* and *synthesized*. For a given production, the attribute evaluation rules compute the synthesized attributes of the left-hand symbol of the production and the inherited attributes of the right-hand symbols. The use of attributes provides an attribute grammar with the formal representational power of a Turing machine [33,34].

Within NGAGE, the productions of an attribute grammar define a space of possible neural network architectures. The context-free portion of a production defines a functional, topological and/or behavioural rule for how to create part of a neural network. The attributes of a production serve several purposes. They may impose constraints, provide values, specify neural functions, define neural structures, share partial specifications, aggregate model details, Ultimately, the attributes of the root symbol will contain a complete specification of a neural network (in GNML). To enable a single grammar to specify both functional and structural neural properties, the attribute evaluation rules use string and set operations to compose GNML specifications within the attributes directly. Through the process of evaluating the attributes of the parse tree, a complete GNML specification of a single neural network is produced.

For instance, Figure 3 presents several productions within an NGAGE grammar that produce networks with different types of modules. Each production is numbered for identification, and each grammar symbol is given a meaningful name (e.g., Module). Each attribute of a symbol is also given a meaningful name and is indicated in lower-case italics and in reference to the symbol (e.g., Module.*nodes*). Each NGAGE

1. Network → { in-port } Module { out-port }
   - i Network.*spec* = finalize(Network.*inputs*, Network.*outputs*, Network.*nodes*, Network.*connections*)
   - ii. Network.*nodes* = Module.*nodes*
   - iii Network.*connections* = Module.*connections* ∪ Network.*inputs* × Module.*input_nodes* ∪ Module.*output_nodes* • Network.*outputs*
   - iv Network.*inputs* = create_set(Network.*INPUT_SIZE*, in-port.*spec*)
   - v Network.*outputs* = create_set(Network.*OUTPUT_SIZE*, out-port.*spec*)
   - vi Network.*id* = "1"
   - *a.* Module.*id* = Network.*id* + ".1"

2. $Module_1$ → sequential_full ($Module_2$, $Module_3$)
   - i $Module_1$.*nodes* = $Module_2$.*nodes* ∪ $Module_3$.*nodes*
   - ii $Module_1$.*connections* = $Module_2$.*connections* ∪ $Module_3$.*connections* ∪ $Module_2$.*output_nodes* × $Module_3$.*input_nodes*
   - iii $Module_1$.*input_nodes* = $Module_2$.*input_nodes*
   - iv $Module_1$.*output_nodes* = $Module_3$.*output_nodes*
   - *a.* $Module_2$.*id* = $Module_1$.*id* + ".1" *b.* $Module_3$.*id* = $Module_1$.*id* + ".2"

3. $Module_1$ → sequential_1to1 ($Module_2$, $Module_3$)
   - i, iii, iv, *a*, *b* – as in production 2
   - ii $Module_1$.*connections* = $Module_2$.*connections* ∪ $Module_3$.*connections* ∪ $Module_2$.*output_nodes* • $Module_3$.*input_nodes*

4. $Module_1$ → parallel ($Module_2$, $Module_3$)
   - i, *a*, *b* – as in production 2
   - ii $Module_1$.*connections* = $Module_2$.*connections* ∪ $Module_3$.*connections*
   - iii $Module_1$.*input_nodes* = $Module_2$.*input_nodes* ∪ $Module_3$.*input_nodes*
   - iv $Module_1$.*output_nodes* = $Module_2$.*output_nodes* ∪ $Module_3$.*output_nodes*

5. Module → Recurrent-Layer
   - i Module.*nodes* = Recurrent-Layer.*nodes*
   - ii Module.*input_nodes* = Recurrent-Layer.*nodes*
   - iii Module.*output_nodes* = Recurrent-Layer.*nodes*
   - iv Module.*connections* = Recurrent-Layer.*connections*
   - *a.* Recurrent-Layer.*id* = Module.*id* + ".1"

6. Module → Grid
   - i Module.*nodes* = Grid.*nodes*
   - ii Module.*input_nodes* = Grid.*input_nodes*
   - iii Module.*output_nodes* = Grid.*output_nodes*
   - iv Module.*connections* = Grid.*connections*
   - *a.* Grid.*id* = Module.*id* + ".1"

Fig. 3. NGAGE productions for heterogeneous modular networks

production consists of a context-free production (e.g., Module → Grid), attribute evaluation rules for synthesized attributes, indicated by a lower-case roman numeral (e.g., i. Module.*nodes* = Recurrent-Layer.*nodes*), and attribute evaluation rules for inherited attributes, indicated by a lower case letter (e.g., *c.* Recurrent-Layer.*id* = Module.*id* + ".1"). If the same grammar symbol appears multiple times within a production,

each instance is distinguished by a lower-case number (e.g., $Module_1$ → parallel ($Module_2$, $Module_3$)).

Attribute evaluation rules may perform various operations, such as set, mathematical, string or logical operations. A special attribute, *id*, is used to uniquely identify all elements of a network. A top-level parameter value is indicated as an attribute on the root symbol and written in capital letters (e.g., Network.*INPUT_SIZE*). An attribute may contain arbitrary levels of detail. For instance, Network.*spec* may contain a complete GNML specification of the functionality of an entire network, Network.*id* contains a single string value, and Network.*connections* may contain an arbitrarily large set of connections among neurons. Attribute evaluation functions may use helper functions (e.g., create_set(), which copies the given object to form a set of the given size with unique *ids*, or finalize() which converts the node and connection sets into a complete GNML specification).

Within an NGAGE grammar, information may flow both up the parse tree (via synthesized attributes) as well as down (via inherited attributes) when evaluating attributes. For instance, the *id* attribute passes identity information down the tree, while the *nodes* and *connections* attributes pass node and connection details up the tree to produce a complete specification at the root of both the graph formed by the connections and the functionality of each neuron. The operator '×' is used as a shorthand for fully connecting two sets of nodes, and the operator '•' is used as a shorthand for one-to-one connectivity between two sets of nodes. These operations produce GNML specifications of the connections using references to the associated nodes.

The productions in all sample grammars presented in the figures are numbered so that, when they are combined together as a single grammar, symbols representing similar elements are grouped together.

5.1 Structural Level

A key capability of an NGAGE grammar is the use of productions to specify the structure of a neural network model. A grammar may specify simple, regular structures as well as highly complex ones. For instance, Figure 3 defines networks with two types of modules connected in parallel or sequence with arbitrary hierarchical nesting and potentially complex inter-module connectivity. Figure 4 by contrast presents productions that specify a family of simple multi-layer modules with highly regular connectivity.

The attributes of a set of productions may be used to impose structural constraints that effectively limit the otherwise unbounded expansion of the context free productions. For instance, in Figure 4 the Multi-Layer.*max_layers* attribute and associated attribute evaluation rules impose a limit of 5 layers upon the multi-layer modules regardless of the depth of expansion of the Multi-Layer symbol, while the Layer productions are unbounded by their attributes and each layer may specify an arbitrary number of neurons.

7.	Module → Multi-Layer
i	Module.*nodes* = Multi-Layer.*nodes*
ii	Module.*input_nodes* = Multi-Layer.*input_nodes*
iii	Module.*output_nodes* = Multi-Layer.*output_nodes*
iv	Module.*connections* = Multi-Layer.*connections*
a	Multi-Layer.*id* = Module.*id* + ".1"
b	Multi-Layer.*max_layers* = 5
8.	Multi-Layer_1 → Multi-Layer_2 Layer
i	Multi-Layer_1.*nodes* = Multi-Layer_2.*nodes* ∪ Layer.*nodes*
ii	Multi-Layer_1.*input_nodes* = if Multi-Layer_1.*max_layers* > 1, Multi-Layer_2.*input_nodes* else Layer.*nodes*
iii	Multi-Layer_1.*output_nodes* = Layer.*nodes*
iv	Multi-Layer_1.*connections* = Multi-Layer_2.*connections* ∪ Multi-Layer_2.*output_nodes* × Layer.*nodes*
a	Multi-Layer_2.*id* = Multi-Layer_1.*id* + ".1"
b	Layer.*id* = Multi-Layer_1.*id* + ".2"
c	Multi-Layer_2.*max_layers* = max (Multi-Layer_1.*max_layers – 1, 0)*
9.	Multi-Layer → Layer
i	Multi-Layer.*nodes* = if Multi-Layer.*max_layers* > 0, Layer.*nodes* else { }
ii.	Multi-Layer.*input_nodes* = if Multi-Layer.*max_layers* > 0, Layer.*nodes* else { }
iii	Multi-Layer.*output_nodes* = if Multi-Layer.*max_layers* > 0, Layer.*nodes* else { }
iv	Multi-Layer.*connections* = { }
a	Layer.*id* = Multi-Layer.*id* + ".1"
10.	Layer_1 → Layer_2 Neuron
i	Layer_1.*nodes* = Layer_2.*nodes* ∪ Neuron.*spec*
a	Layer_2.*id* = Layer_1.*id* + ".1"
b	Neuron.*id* = Layer_1.*id* + ".2"
11.	Layer → Neuron
i	Layer.*nodes* = Neuron.*spec*
a	Neuron.*id* = Layer.*id* + ".1"

Fig. 4. NGAGE grammar productions for multi-layer module

5.2 Sharing Elements across Models

NGAGE allows components of similar type from different models to be specified using common elements where appropriate. For instance, Figure 5 shows productions that expand the Recurrent-Layer and Grid symbols from Figure 3 to define the corresponding neural modules structures. These productions re-use the grammar symbols Layer and Neuron from Figure 4, thereby enabling the three different module-types (Grid, Recurrent-Layer and Multi-Layer) to be described using the same basic building blocks.

Through the use of common symbols representing common elements, novel structures may be generated through interchanges. For instance, all expansions of the production symbol Module in Figure 3 may be regarded by a search mechanism as interchangeable, and likewise the symbol Layer.

12. Recurrent-Layer → Layer

i Recurrent-Layer.*nodes* = Layer.*nodes*

ii Recurrent-Layer.*connections* = Layer.*nodes* × Layer.*nodes* – Layer.*nodes* • Layer.*nodes*

a. Layer.*id* = Recurrent-Layer.*id* + ".1"

13. $Grid_1$ → $Grid_2$ side-corner-bottom (Neuron)

i $Grid_1$.*nodes* = $Grid_2$.*nodes* ∪ $Grid_1$.*bottom_nodes* ∪ $Grid_1$.*side_nodes* ∪ $Grid_1$.*corner_node*

ii $Grid_1$.*connections* = $Grid_2$.*connections* ∪ $Grid_1$.*bottom_nodes* • ($Grid_2$.*bottom_nodes* ∪ $Grid_2$.*corner_node*) ∪ ($Grid_2$.*bottom_nodes* ∪ $Grid_2$.*corner_node*) • $Grid_1$.*bottom_nodes* ∪ $Grid_1$.*side_nodes* • ($Grid_2$.*side_nodes* ∪ $Grid_2$.*corner_node*) ∪ ($Grid_2$.*side_nodes* ∪ $Grid_2$.*corner_node*) • $Grid_1$.*side_nodes* ∪ double_link($Grid_1$.*side_nodes* ∪ $Grid_1$.*corner_node*) ∪ double_link($Grid_1$.*bottom_nodes* ∪ $Grid_1$.*corner_node*)

iii $Grid_1$.*bottom_nodes* = create_set($Grid_2$.*size*, Neuron.*spec*)

iv $Grid_1$.*side_nodes* = create_set($Grid_2$.*size*, Neuron.*spec*)

v $Grid_1$.*corner_node* = create_set(1, Neuron.*spec*)

vi $Grid_1$.size = $Grid_2$.size + 1

a. $Grid_2$.*id* = $Grid_1$.*id* + ".1"

14. Grid → Neuron

i Grid.*nodes* = Grid.*corner_node*

ii Grid.*connections* = { }

iii Grid.*bottom_nodes* = { }

iv Grid.*side_nodes* = { }

v Grid.*corner_node* = Neuron.*spec*

vi Grid.*size* = 1

a. Neuron.*id* = Grid.*id* + ".1"

Fig. 5. Additional productions showing definition of different types of modules using common grammar symbols

5.3 Programmatic Search of Models

Through the explicit representation of the derivation of neural structure and functions, NGAGE provides support for the programmatic exploration of diverse neural architectures. Figure 6 illustrates a portion of parse tree that may be generated using the productions of Figures 3 and 4. Figure 7 illustrates the potential network for that parse tree, assuming a full expansion of all modules, where INPUT_SIZE is 3 and OUTPUT_SIZE is 4.

Because the parse trees of an attribute grammar are equivalent to a simple context-free parse tree, an evolutionary algorithm based on genetic programming may be used to automatically explore the large space of potential network variations. Genetic manipulations may include exchanging of subtrees rooted by identical symbols (e.g., exchanging Module subtrees), as well as the replacement of a subtree with a new random expansion of the subtree's root symbol. The meta-model search may be constrained to explore only certain subsets of possible architectures by limiting the symbols upon which genetic manipulations may occur [31].

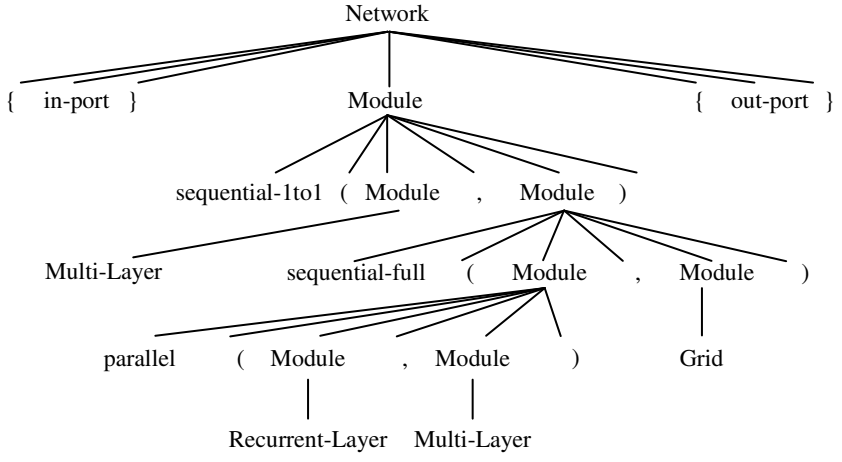


Fig. 6. Partial context-free parse tree generated from NGAGE grammar

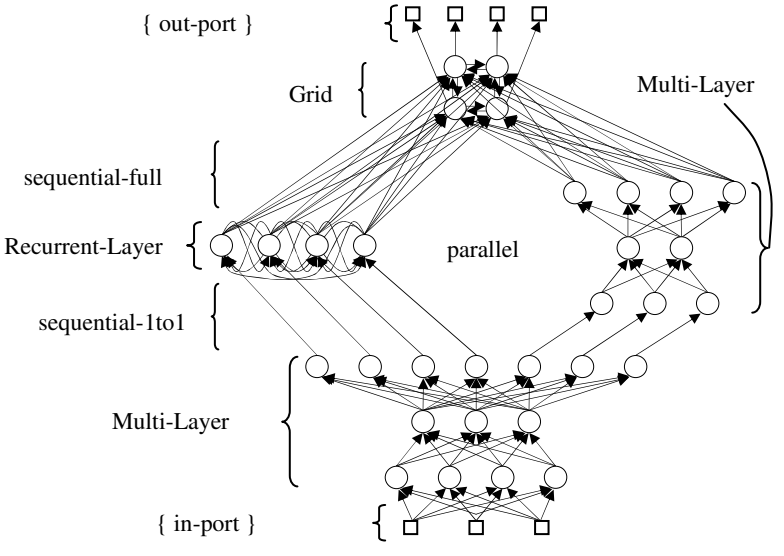


Fig. 7. Network architecture generated from parse tree

5.4 Mechanical Level

NGAGE allows the explicit specification of a wide range of neural functions. To accomplish this, it adopts a generic neuron model (see Figure 8) in which a neuron is regarded as an arbitrary sequence of (nested) functions with access to several memory stores – one for incoming connections, one for outgoing connections, and a general memory store. The neuron can receive multiple types of input signals and produce

multiple types of output signals. The GNML specification format (see Figure 9) represents the network as a list of the input and output ports of the network, followed by a list of the neurons, each with a unique identifier and a nested specification of its functionality (i.e., <NeuronSpec> block), and a list of all the connections in the network, where each connection indicates its source and target and the type of signal it transmits.

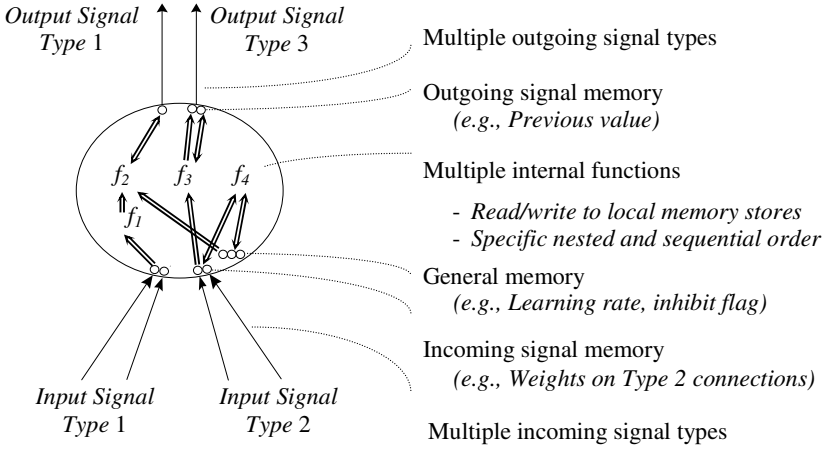


Fig. 8. Generic neuron model used in NGAGE

The GNML specification is executed by an interpreter that computes all neural functions in the order specified and cycles through the network over time. Changes to memory and to signal values are performed as specified to produce a functional neural network system.

```
<Network>
 <InPort id='i1' signalType='Activation' />
  . . .
 <OutPort id='o1' signalType='Activation, Feedback' />
 <Neuron id='1.1.1'>
         <NeuronSpec>
               <FunctionSpec> . . . </FunctionSpec>
               <FunctionSpec> . . . </FunctionSpec>
               <GeneralMemorySpec> . . . </GeneralMemorySpec>
               . . .
         </NeuronSpec>
 </Neuron>
 <Neuron id='1.1.2'>
         <NeuronSpec> . . . </NeuronSpec>
 </Neuron>
 . . .
 <Connection source='i1' target='1.1.1' signalType='Activation'/>
. . .
 <Connection source='o1' target='1.1.2' signalType='Feedback'/>
</Network>
```

Fig. 9. Sample GNML specification of a network

5.5 Functional Level

NGAGE productions compose specifications at the functional level primarily through the use of GNML templates, string substitution and concatenation. Figure 10 illustrates some sample productions that may be used to compose the specification of a neuron. Productions may include arbitrary scalar and vector operations.

15. Neuron → general-neuron (FunctionSequence)
    i Neuron.*spec* = substitute(general-neuron.*template*, "SUBFUNCTIONS", FunctionSequence.*spec*)
16. $\text{FunctionSequence}_1$ → $\text{FunctionSequence}_2$ Function
    i $\text{FunctionSequence}_1$.*spec* = $\text{FunctionSequence}_2$.*spec* + Function.*spec*
17. FunctionSequence → Function
    i FunctionSequence.*spec* = Function.*spec*
18. Function → dot-product (VectorParameter_1 , VectorParameter_2)
    i Function.*spec* = substitute(substitute(dot-product.*template*, "SUBPARAM1", VectorParameter_1.*spec*), "SUBPARAM2", VectorParameter_2.*spec*)
19. VectorParameter → memory (weight)
    i VectorParameter.*spec* = "<GeneralMemoryVector name='weight' />"
20. VectorParameter → input-signal (activation)
    i VectorParameter.*spec* = "<InputSignalVector name='currentValue' type='Activation' />"

where

general-neuron.*template* = "<*NeuronSpec*> SUBFUNCTIONS </NeuronSpec>"

dot-product.*template* =
```
"<FunctionSpec   nativeCode='dotProduct'
                 returnType='scalar'>
      <VParam>   SUBPARAM1   </VParam>
      <VParam>   SUBPARAM2   </VParam>
 </FunctionSpec>"
```

Fig. 10. Productions for determining the functionality of a neuron

5.6 Developmental Level

At the developmental level, NGAGE allows the explicit specification of learning rules. Figure 11 shows productions that replace (e.g., production 7' replaces production 7) or augment the productions of Figures 3 and 10. A specific symbol indicative of the role of the neural element (e.g., LearningRule) is used. The attributes of the productions for the module are used to ensure that the same learning rule is applied at all neurons in the module. A different set of productions could impose a different degree of regularity (in the extreme allowing every neuron to apply its own distinct learning rule).

7'. Module → uniform-learning (Multi-Layer , LearningRule)
  a Multi-Layer.*sharedLearningRule* = LearningRule.*spec*

8'. Multi-Layer$_1$ → Multi-Layer$_2$ Layer
  a Multi-Layer$_2$.*sharedLearningRule* = Multi-Layer$_1$.*sharedLearningRule*
  b Layer.*sharedLearningRule* = Multi-Layer$_1$.*sharedLearningRule*

9'. Multi-Layer → Layer
  a Layer.*sharedLearningRule* = Multi-Layer.*sharedLearningRule*

10'. Layer$_1$ → Neuron Layer$_2$
  a Neuron.*sharedLearningRule* = Layer$_1$.*sharedLearningRule*
  b Layer$_2$.*sharedLearningRule* = Layer$_1$.*sharedLearningRule*

11'. Layer → Neuron
  a Neuron.*sharedLearningRule* = Layer.*sharedLearningRule*

15'. Neuron → general-neuron (ActivationFunction)
  i Neuron.*spec* = substitute(general-neuron.*template*, "SUBFUNS",
  ActivationFunction.*spec* + Neuron.*sharedLearningRule*)

21. LearningRule → update (FunctionSequence, weight)
  i LearningRule.*spec* = substitute(substitute(update.*template*, "SUBFUNS",
  FunctionSequence.*spec*), "SUBMEM",
  "<GeneralMemoryVector name='weight' />")

22. ActivationFunction → FunctionSequence
  i ActivationFunction.*spec* = FunctionSequence.*spec*

where
update.*template* =
```
"<FunctionSpec  nativeCode='update'
                returnType='vector'>
     <VParam>   SUBFUNS </VParam>
     <VParam>   SUBMEM  </VParam>
</FunctionSpec>"
```

Fig. 11. Productions showing the specification and sharing of a learning rule

5.7 Behavioural Level

NGAGE provides the capability to define productions that capture the behaviour of a neural element, as well as the relationship among elements. Several different methods may be used to specify behaviours within an NGAGE grammar. Any or all of these methods may be used in a single grammar as needed.

Explicit symbol for a role: A grammar may use a specific symbol to denote a type of role. That symbol, in turn, may be expanded by other productions into neural structures and functions that support that role. In this method, the choice of a role is primary, and the choice of architecture to fill that role is secondary. For example, ActivationFunction and LearningRule in Figure 11 may be interpreted as symbols that identify roles rather than specific functions. Those symbols may, in principle, be expanded into a variety of functions. Figure 12 illustrates the use of a symbol (ClassifierRole or ClusteringRole) to distinguish neural architectures by the role they play. In this method, the attribute evaluation functions typically copy key structural and functional specifications of children to the parent.

23. ClassifierRole → BackPropagationModule
    - i. ClassifierRole.*nodes* = BackPropagation.*nodes*
    - ii ClassifierRole.*connections* = BackPropagation.*connections*
    - iii ClassifierRole.*input_nodes* = BackPropagation.*input_nodes*
    - iv ClassifierRole.*output_nodes* = BackPropagation.*output_nodes*

24. ClassifierRole → MixturesOfExpertsModule
    Analogous attribute evaluation rules to production 23

25. ClusteringRole → SelfOrganizingFeatureMapModule
    Analogous attribute evaluation rules to production 23

26. ClusteringRole → ARTModule
    Analogous attribute evaluation rules to production 23

Fig. 12. Use of production symbols to explicitly identify roles that may be filled by different neural architectures

Composition of behaviours: A grammar may include a set of productions that define and expand a variety of behaviours. This expansion provides a basic description of all the roles in the network and how they are related. In this method, the choice of how different roles are organized within the network is primary. These productions are largely independent of the typical developmental, structural and functional productions, and form a largely distinct high-level description of the network. In this method, attribute evaluation functions play their usual role of aggregating component specs. However, they also pass-down constraints and details that may influence how subsequent productions will specify the relevant structures and functions. Figure 13 illustrates productions that explicitly identify a sequence of roles that will be the basis

27. $Role_1$ → FilterRole feed-into $Role_2$
    - i. $Role_1$.*nodes* = FilterRole .*nodes* ∪ $Role_2$.*nodes*
    - ii $Role_1$.*connections* = FilterRole.*connections* ∪ $Role_2$.*connections*
    - iii $Role_1$.*input_nodes* = FilterRole.*input_nodes* ∪ $Role_2$.*input_nodes*
    - iv $Role_1$. *output_nodes* = FilterRole.*output_nodes* ∪ $Role_2$.*output_nodes*
    - *a* FilterRole.*nodes_to_feed_into* = $Role_2$.*input_nodes*

28. $Role_1$ → ReduceDimensionsRole feed-into $Role_2$
    Analogous attribute evaluation rules for production 27

29. $Role_1$ → ClusteringRole feed-into $Role_2$
    Analogous attribute evaluation rules for production 27

30. $Role_1$ → ClassifierRole feed-into $Role_2$
    Analogous attribute evaluation rules for production 27

31. ReduceDimensionsRole → ClusteringRole
    Analogous attribute evaluation rules for production 27 (i – iv)

Fig. 13. Productions that define roles in a network and how they are organized

32. MixtureOfExpertsModule → ProcessingModules gated-by GatingModule
  a GatingModule.*number_gates* = size(ProcessingModules.*number_experts*)
  b GatingModule.*nodes_to_influence* = ProcessingModules.*influenceable_nodes*

33. ProcessingModules → AllExperts integrated-by IntegratorModule
  i ProcessingModules.*influenceable_nodes* = IntegratorModule.*input_nodes*
  ii ProcessingModules.*number_experts* = AllExperts.*number_experts*

34. GatingModule → BackPropagationModule
  i GatingModule.*connections* = BackPropagation.*connections* ∪
    GatingModule.*output_nodes* • GatingModule.*nodes_to_influence*
  a BackPropagationModule.*num_outputs* = GatingModule.*number_gates*

35. AllExperts_1 → ExpertModule AllExperts_2
  i AllExperts_1.*number_experts* = AllExperts_2.*number_experts* + 1

36. AllExperts_1 → BackPropagationModule
  i AllExperts.*number_experts* = 1

Fig. 14. Productions that explicitly identify interactions among components

for the model. The attribute FilterRole.*nodes_to_feed_into* is used in subsequent productions to create the appropriate connections from the output elements of the structure performing the FilterRole to the input elements of the structure performing the subsequent role.

Explicit interactions among components: A production may specify how one component is intended to interact with another. These are often accomplished via descriptive terminal symbols and attribute evaluation functions that specify the appropriate structural or functional neural elements needed to support the interactions. Figure 14 illustrates simplified productions of a grammar for specifying a Mixtures of Experts network. The terminal symbols 'gated-by' and 'integrated-by' are explicit indications of the relationship between neural components. The attributes of the productions, in addition to collecting the sets of nodes and connection specifications (not shown) and constraints (e.g., *number_gates*), also specify which nodes in one component (e.g., the integrator nodes) are influenced by which nodes in another component (e.g., the gating module) so that they may be connected appropriately. Subsequent productions expand the structural and functional details for these components. Explicitly identifying influenceable nodes may be used to support other types of relationships between components. For example, if one component 'controls' or 'instructs' another.

5.8 Environmental Level

NGAGE productions may be used to explicitly specify the environment in which a neural component operates. For example, Figure 15 shows a grammar in which productions partition the input vector, the output vector or both to create sub-problems that may then be solved by distinct modules. The operation apply_filter selects a subset from a given set based on a given binary filter; the operation apply_complement selects the subset excluded by the filter. Figure 16 illustrates a potential decomposition that

may be generated from the productions of Figure 3, 4 and 15. The same partitioning approach may be used to specify the environment of nested modules within the context of the environment of the encapsulating module.

1'. Network → { in-port } Task { out-port }
i, iv, v, vi, *a* – as in Figure 5
ii. Network.*nodes* = Task.*nodes*
iii Network.*connections* = Task.*connections*
b Task.*id* = Network.*id* + ".1"
c Task.*inputs* = Network.*inputs* *d* Task.*outputs* = Network.*outputs*
37. $Task_1$ → Partition (in) $Task_2$ $Task_3$
i $Task_1$.*nodes* = $Task_2$.*nodes* ∪ $Task_3$.*nodes*
ii $Task_1$.*connections* = $Task_2$.*connections* ∪ $Task_3$.*connections*
a $Task_2$.*id* = $Task_1$.*id* + ".1" *b* $Task_3$.*id* = $Task_1$.*id* + ".2"
c $Task_2$.*inputs* = apply_filter(Partition.*filter*, $Task_1$.*inputs*)
d $Task_3$.*inputs* = apply_complement(Partition.*filter*, $Task_1$.*inputs*)
e $Task_2$.*outputs* = $Task_1$.*outputs* *f* $Task_3$.*outputs* = $Task_1$.*outputs*
38. $Task_1$ → $Task_2$ $Task_3$ $Partition_2$ (out)
i, ii, *a*, *b* as in production 37
c $Task_2$.*inputs* = $Task_1$.*inputs* *d* $Task_3$.*inputs* = $Task_1$.*inputs*
e $Task_2$.*outputs* = apply_filter(Partition.*filter*, $Task_1$.*outputs*)
f $Task_3$.*outputs* = apply_complement(Partition.*filter*, $Task_1$.*outputs*)
39. $Task_1$ → $Partition_1$ (in) $Task_2$ $Partition_2$ (out) $Partition_3$ (in) $Task_3$ $Partition_4$ (out)
i, ii, *a*, *b* as in production 37
c $Task_2$.*inputs* = apply_filter($Partition_1$.*filter*, $Task_1$.*inputs*)
d $Task_3$.*inputs* = apply_filter($Partition_3$.*filter*, $Task_1$.*inputs*)
e $Task_2$.*outputs* = apply_filter($Partition_2$.*filter*, $Task_1$.*outputs*)
f $Task_3$.*outputs* = apply_filter($Partition_4$.*filter*, $Task_1$.*outputs*)
40. Task → Module
i Task.*nodes* = Module.*nodes*
ii Task.*connections* = Module.*connections* ∪ Task.*inputs* × Module.*input_nodes* ∪ Module.*output_nodes* • Task.*outputs*
a Module.*id* = Task.*id* + ".1"
41. $Partition_1$ → 1 $Partition_2$
i $Partition_1$.*filter* = $Partition_2$.*filter* + "1"
42. $Partition_1$ → 0 $Partition_2$
i $Partition_1$.*filter* = $Partition_2$.*filter* + "0"
43. Partition → 1
i Partition.*filter* = "1"
44. Partition → 0
i Partition.*filter* = "0"

Fig. 15. Productions that define the environment for different modules

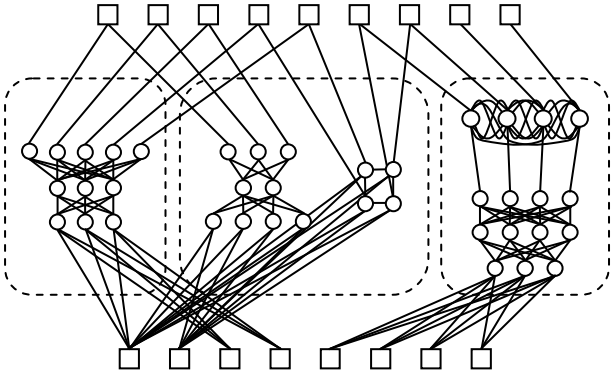


Fig. 16. Network with environment partitioning among heterogeneous modules

6 Conclusions

Current deficiencies in neural network modeling methods limit the capability of researchers to programmatically explore new models. A perspective on the meta-modelling issues that need to be addressed in the field was given, and six levels of model specification were identified. The way forward for the field will involve the creation of robust specification languages that address all six levels and represent multiple, diverse models. However, any such language will necessarily be fairly complex in order to achieve the representational power needed. The Network Generating Attribute Grammar Encoding (NGAGE) was presented as an example of a meta-modelling specification method that addresses all six levels. Further, through the separation of simple context-free productions from more detailed attribute evaluation functions, NGAGE attempts to break down the complexity of the language into manageable pieces that facilitate systematic exploration.

NGAGE provides a rich foundation for representing diverse neural network architectures and for programmatically searching through that space of possible networks. However, while grammar productions for a variety of behaviours, structures and functions have been defined, these only encompass a small number of the available models. Creating additional productions covering more models will extend the utility of NGAGE as a meta-modelling tool. Further, NGAGE is limited in current practice to a particular family of neurons, as defined by the model illustrated in Figure 9. The focus of NGAGE has been the area of neural networks for learning, and hence relatively basic attention has been paid to the internal details of neurons within a network. While the NGAGE neuron model and the associated Generic Neural Markup Language encompass the types of neurons and connections used in the many models in the area of neural networks for learning, it is lacking with respect to the area of neural networks for biological modelling. To address this issue requires a robust, multi-model specification language for biological models. The author anticipates that future languages, such as NineML, will provide such a basis and that NGAGE may be adapted to provide a meta-modelling layer above those languages.

References

[1] Parks, R.W., Levine, D.S., Long, D.L.: Fundamentals of Neural Network Modeling: Neuropsychology and Cognitive Neuroscience. The MIT Press, Cambridge (1999)

[2] http://grey.colorado.edu/emergent/index.php/Comparison_of_Neural_Network_Simulators

[3] Demuth, H., Beale, M., Hagan, M.: Neural Network Toolbox: User's Guide, Version 6. The Mathworks, Inc. (2010)

[4] Aisa, B., Mingus, B., O'Reilly, R.: The emergent neural modeling system. Neural Networks 21, 1045–1212 (2008)

[5] NeuroSolutions. v 6.0 by NeuroDimension, Inc., http://www.neurosolutions.com/products/ns/

[6] Fischer, I., Hennecke, F., Bannes, C., Zell, A.: JavaNNS Java Neural Network Simulator User Manual Version 1.1. University of Tubingen Wilhelm-Schickard-Institute for Computer Science. Department of Computer Architecture, University of Tubingen Wilhelm-Schickard-Institute for Computer Science (2001)

[7] Simbrain, Y.J.: A visual framework for neural network analysis and education. In: Lorenz, S., Egelhaaf, M. (eds.) Interactive Educational Media for the Neural and Cognitive Sciences; Brains, Minds & Media 3, bmm1411(2008)

[8] Hall, M., Frank, E., Holmes, G., Pfahringer, B., Reutemann, P., Witten, I.H.: The WEKA data mining software: An update. SIGKDD Explorations 11 (1) (2009)

[9] Beeman, D., Wang, Z., Edwards, M., Bhalla, U., Cornelis, H., Bower, J.: The GENESIS 3.0 Project: a universal graphical user interface and database for research, collaboration, and education in computational neuroscience. BMC Neuroscience 8(Suppl. 2), P4 (2007)

[10] Carnevale, N.T., Hines, M.L.: The NEURON Book. Cambridge University Press, Cambridge (2006)

[11] Dudani, N., Ray, S., George, S., Bhalla, U.S.: Multiscale modeling and interoperability in MOOSE. BMC Neuroscience 10(Suppl. 1), P54 (2010)

[12] Kock, G., Serbedzija, N.B.: Object-oriented and functional concepts in artificial neural network modelling. In: Intl. Joint Conference on Neural Networks, vol. 1, pp. 923–927 (1993)

[13] Hopp, H., Prechelt, L.: CuPit-2 - A Parallel Language for Neural Algorithms: Language Reference and Tutorial. Technical Report 4/97, Institut fur Programmstrukturen und Datenorganisation. Karlsruhe, Germany (1997)

[14] Strey, A.: EpsiloNN - A Tool for the Abstract Specification and Parallel Simulation of Neural Networks. Systems Analysis - Modelling - Simulation (SAMS) 34(4) (1999)

[15] Linden, A., Tietz, C.: Combining multiple neural network paradigms and applications using SESAME. In: Intl. Joint Conference on Neural Networks, vol. 2, pp. 528–533 (1992)

[16] Rubtsov, D., Butakov, S.: Application of XML for neural network exchange. Computer Standards and Interfaces,24(4), p. 311-322 (2002)

[17] Gorchetchnikov and INCF Multiscale Modeling Taskforce. NineML – a description language for spiking neuron network modeling: the user layer. BMC Neuroscience 11(Suppl. 1), P71 (2010)

[18] Raikov and INCF Multiscale Modeling Taskforce. NineML – a description language for spiking neuron network modeling: the abstraction layer. BMC Neuroscience, 11(Suppl. 1) P66 (2010)

[19] Data Mining Group. PMML version 4.0 (2009), http://www.dmg.org/pmml-v4-0.html

[20] Miller, G.F., Todd, P.M., Hegde, S.U.: Designing neural networks using genetic algorithms. In: Third Intl Conf. on Genetic Algorithms and Their Applications, pp. 379–384. Morgan Kaufmann, San Francisco (1989)
[21] Polani, D., Uthmann, T.: Adaptation of Kohonen feature map topologies by genetic algorithms. In: Männer, R., Manderick, B. (eds.) Parallel Problem Solving from Nature 2, pp. 421–429. Elsevier, Amsterdam (1992)
[22] Schaffer, J.D., Caruana, R.A., Eshelman, L.J.: Using genetic search to exploit the emergent behavior of neural networks. Physica D 42, 244–248 (1990)
[23] Jacob, C.: Genetic L-system programming. In: Davidor, Y., Männer, R., Schwefel, H.-P. (eds.) PPSN 1994. LNCS, vol. 866, pp. 334–343. Springer, Heidelberg (1994)
[24] Kitano, H.: Designing neural networks using genetic algorithms with graph generation system. Complex Systems 4, 461–476 (1990)
[25] Gruau, F.: Automatic definition of modular neural networks. Adaptive Behavior 3(2), 151–183 (1995)
[26] Bengio, S., Bengio, Y., Cloutier, J.: Use of genetic programming for the search of a new learning rule for neural networks. In: First Conference on Evolutionary Computation, pp. 324–327 (1994)
[27] Tsoulosa, I., Gavrilis, D., Glavas, E.: Neural network construction and training using grammatical evolution. Neurocomputing 72, 269–277 (2008)
[28] Mouret, J., Doncieux, S.: MENNAG: a modular, regular and hierarchical encoding for neural-networks based on attribute grammars. Evolutionary Intelligence 1, 187–207 (2008)
[29] Hussain, T.S., Browse, R.A.: Network generating attribute grammar encoding. In: IEEE International Joint Conference on Neural Network, pp. 431–436 (1998)
[30] Hussain, T.S., Browse, R.A.: Evolving neural networks using attribute grammars. In: 2000 IEEE Symposium on Combinations of Evolutionary Computation and Neural Networks, pp. 37-42 (2000)
[31] Hussain, T.S.: Attribute Grammar Encoding of the Structure and Behaviour of Artificial Neural Networks. Ph.D. Thesis, Queen's University (2003)
[32] Hussain, T.S.: Generic Neural Markup Language: Facilitating the Design of Theoretical Neural Network Models. In: Proceedings of the 2004 IEEE International Joint Conference on Neural Networks (IJCNN), pp. 235–242. Institute of Electrical and Electronics Engineers, Inc., Piscataway (2004)
[33] Knuth, D.E.: The semantics of context-free languages. Mathematical Systems Theory 2(2), 127–145 (1968)
[34] Deransart, P., Lorho, B., Jourdan, M.: Attribute Grammars: Definitions, Systems and Bibliography. LNCS, vol. 323. Springer, Heidelberg (1988)
[35] Vellacott, O.R.: A framework of hierarchy for neural theory. In: Second International Conference on Artificial Neural Networks, pp. 237–241 (1991)
[36] Gleeson, P., Steuber, V., Silver, R.A.: NeuroConstruct: A Tool for Modeling Networks of Neurons in 3D Space. Neuron 54(2), 219–235 (2007)
[37] Mcgarry, K., Wermter, S., Macintyre, J.: Hybrid neural systems: from simple coupling to fully integrated neural networks. Neural Computing Surveys 2, 62–93 (1999)
[38] Shavlik, J.W.: A Framework for Combining Symbolic and Neural Learning, Technical Report 1123, Computer Sciences Department, University of Wisconsin - Madison (November 1992)
[39] Hussain, T.S.: Explicit learning relationships within neural systems. In: Workshop on Achieving Functional Integration of Diverse Neural Models, held at 2005 International Joint Conference on Neural Networks, p.16 (2005)

[40] Iversen, A., Taylor, N.K., Brown, K.: Integrating Neural Network Strategies for Discrimination, Recognition and Clustering. In: Workshop on Achieving Functional Integration of Diverse Neural Models, held at 2005. Intl. Joint Conference on Neural Networks, p. 4 (2005)
[41] de Kamps, M.: Large scale brain simulations are not a technical problem. In: Workshop on Achieving Functional Integration of Diverse Neural Models, held at 2005 International Joint Conference on Neural Networks, p. 2 (2005)
[42] de Kamps, M., Baier, V., Drever, J., Dietz, M., Mosenlechner, L., van der Velde, F.: The state of MIIND. Neural Networks 21(8), 1164–1181 (2008)
[43] Smieja, F.J., Mühlenbein, H.: Reflective Modular Neural Network Systems. Technical Report: GMD #633, German National Research Centre for Computer Science (1992)
[44] Jacobs, R.A., Jordan, M.I., Nowlan, S.J., Hinton, G.E.: Adaptive mixtures of local experts. Neural Computation 3, 79–87 (1991)
[45] Hecht-Nielsen, R.: Neurocomputing. Addison-Wesley, Reading (1990)
[46] Oblinger, D.: Bootstrapped learning: Creating the electronic student that learns from natural instruction. Defense Advanced Research Projects Agency briefing (2006), http://www.darpa.mil/ipto/programs/bl/docs/AAAI_Briefing.pdf (August 19, 2010)
[47] Burstein, M., Brinn, M., Cox, M., Hussain, T., Laddaga, R., McDermott, D., McDonald, D., Tomlinson, R.: An architecture and language for the integrated learning of demonstrations. In: Workshop on Acquiring Planning Knowledge via Demonstration. held at the Twenty-Second National Conference on Artificial Intelligence (2007)
[48] Burstein, M., Laddaga, R., McDonald, D., Cox, M., Benyo, B., Robertson, P., Hussain, T., Brinn, M., McDermott, D.: POIROT: integrated learning of web service procedures. In: Cohn, A. (ed.) Proceedings of the 23rd National Conference on Artificial intelligence, vol. 3, pp. 1274–1279. AAAI Press, Chicago (2008)
[49] Koch, C., Segev, I.: Methods in Neuronal Modeling. From Ions to Networks, 2nd edn. MIT Press, Cambridge (1998)
[50] Beeman, D.: Introduction to Realistic Neural Modeling. In: Bower, J.M., Beeman, D. (eds.) Special Issue on Realistic Neural Modeling - WAM-BAMM 2005 Tutorials; Brains, Minds 1, bmm218, (2005)
[51] Jaeger, D.: Realistic single cell modeling – from experiment to simulation. In: Bower, J.M., Beeman, D. (eds.) Special Issue on Realistic Neural Modeling WAM-BAMM 2005 Tutorials; Brains, Minds 1, bmm222 (2005)
[52] Carpenter, G.A., Grossberg, S.: ART 2: Self-organization of stable category recognition codes for analog input patterns. Applied Optics 26(23), 4919–4930 (1987)
[53] Murre, J.M.J., Phaf, R.H., Wolters, G.: CALM: Categorizing and Learning Module. Neural Networks 5, 55–82 (1992)
[54] Mendao, M.: Hormonally moderated neural control. In: Hudlicka, E., Canamero, L. (eds.) Architectures for Modeling Emotion: Cross Disciplinary Foundations: Papers from the 2004 Spring Symposium, pp. 92–95. American Assoc. Artificial Intelligence, Menlo Park (2004)
[55] Liu, B., Ding, Y., Wang, J.: Intelligent network control system inspired from neuroendocrine-immune system. In: Chen, Y., Zhang, D., Deng, H., Xiao, Y. (eds.) Proceedings of the 6th International Conference on Fuzzy Systems and Knowledge Discovery, vol. 7, pp. 136–140. IEEE Press, Piscataway (2009)
[56] Århem, P., Klement, G., Blomberg, C.: Channel density regulation of firing patterns in a cortical neuron model. Biophysical Journal 90(12), 4392–4404 (2006)

[57] Mouret, J., Doncieux, S., Girard, B.: Importing the computational neuroscience toolbox into neuro-evolution-application to basal ganglia. In: Proceedings of the 12th Annual Conference on Genetic and Evolutionary Computation, pp. 587–594. ACM Press, New York (2010)
[58] Floreano, D., Epars, Y., Zufferey, J., Mattiussi, C.: Evolution of spiking neural circuits in autonomous mobile robots. Intl. Journal of Intelligent Systems 21(9), 1005–1024 (2006)
[59] Hebb, D.O.: The Organization of Behavior: A Neuropsychological Theory. Wiley, New York (1949)
[60] Grillner, S.: The motor infrastructure: from ion channels to neuronal networks. Nature Reviews Neuroscience 4, 573–586 (2003)
[61] Song, S., Miller, K.D., Abbott, L.F.: Competitive Hebbian learning through spike-timing-dependent synaptic plasticity. Nature Neuroscience 3(9), 919–926 (2000)
[62] Kempter, R., Gerstner, W., van Hemmen, J.L.: Hebbian learning and spiking neurons. Physical Review E 59(4), 4498–4514 (1999)
[63] Bray, T., Paoli, J., Sperberg-McQueen, C.M., Maler, E., Yergeau, F. (eds.): Extensible Markup Language (XML) 1.0 (5th edn.) (August 30, 2010), http://www.w3.org/TR/REC-xml/

Ontology-Based Meta-Mining of Knowledge Discovery Workflows

Melanie Hilario, Phong Nguyen, Huyen Do,
Adam Woznica, and Alexandros Kalousis

Artificial Intelligence Laboratory, University of Geneva

Abstract. This chapter describes a principled approach to meta-learning that has three distinctive features. First, whereas most previous work on meta-learning focused exclusively on the learning task, our approach applies meta-learning to the full knowledge discovery process and is thus more aptly referred to as meta-mining. Second, traditional meta-learning regards learning algorithms as black boxes and essentially correlates properties of their input (data) with the performance of their output (learned model). We propose to tear open the black box and analyse algorithms in terms of their core components, their underlying assumptions, the cost functions and optimization strategies they use, and the models and decision boundaries they generate. Third, to ground meta-mining on a declarative representation of the data mining (DM) process and its components, we built a DM ontology and knowledge base using the Web Ontology Language (OWL).

The Data Mining Optimization Ontology (DMOP, pronounced dee-mope)) provides a unified conceptual framework for analysing DM tasks, algorithms, models, datasets, workflows and performance metrics, as well as their relationships. The DM knowledge base uses concepts from DMOP to describe existing data mining algorithms and their implementations in major DM software packages. Meta-data collected from data mining experiments are also described in terms of concepts from the ontology and linked to algorithm and operator descriptions in the knowledge base; they are then stored in data mining experiment data bases to serve as training and evaluation data for the meta-miner.

These three features together lay the groundwork for what we call deep or semantic meta-mining, i.e., DM process or workflow mining that is driven simultaneously by meta-data and by the collective expertise of data miners embodied in the data mining ontology and knowledge base. In Section 1, we review the state of the art in the fields of meta-learning and data mining ontologies; at the same time, we motivate the need for ontology-based meta-mining and distinguish our approach from related work in these two areas. Section 2 gives a detailed description of DMOP, while Section 3 introduces a novel method for ontology-based discovery of generalized patterns from data mining workflows. Section 4 reports on proof-of-concept experiments conducted to gauge the efficacy of DMOP-based workflow mining, and Section 5 concludes.

N. Jankowski et al. (Eds.): Meta-Learning in Computational Intelligence, SCI 358, pp. 273–315.
springerlink.com

1 State of the Art and Motivation

The work described in this chapter draws together two research streams that have remained independent so far—meta-learning and data mining ontology construction. This section reviews the state of the art in both areas and points out the novelty of our approach with respect to each.

1.1 From Meta-learning to Meta-mining

Meta-learning is learning to learn: in computer science, it is the application of machine learning techniques to meta-data describing past learning experience in order to modify some aspect of the learning process and improve the performance of the resulting model [29,3,13,78]. Meta-learning thus defined applies specifically to learning, which is only one—albeit the central—step in the data mining (or knowledge discovery) process[1]. The quality of mined knowledge depends as much on other steps such as data cleaning, data selection, feature extraction and selection, model pruning, and model aggregation. We still lack an understanding of how the different components of the data mining process interact; there are no clear guidelines except for high-level process models such as CRISP-DM [18]. Process-related issues, such as the composition of data mining operations and the need for a methodology of data mining, are among the ten data mining challenges discussed in [80].

In response to this challenge, a number of systems have been designed to provide user support throughout the different phases of the KD process (Serban et al., 2010). Most of them rely on a planning approach and produce workflows that are valid but not necessarily optimal with respect to a given cost function such as predictive accuracy. This is the case of the planner-based intelligent discovery assistant (IDA) implemented in the e-LICO project[2]. To allow the planner to select the most promising workflows from an often huge set of candidates, an ontology-based meta-learner mines records of past data mining experiments to extract models and patterns that will suggest which DM algorithms should be used together in order to achieve the best results for a given problem, data set and cost function. The e-LICO IDA therefore self-improves as a result of *meta-mining*, loosely defined as KD process-oriented meta-learning. Meta-mining extends the meta-learning approach to the full knowledge discovery process: in the same way that meta-learning is aimed at optimizing the results of learning, meta-mining optimizes the results of data mining by taking into account the interdependencies and interactions between the different process operations, and in particular between learning and the different pre/post-processing steps. In this sense, meta-mining subsumes meta-learning and must address all the open issues regarding meta-learning.

[1] We follow current usage in treating *data mining* and *knowledge discovery* as synonyms, using the terms *learning* or *modelling* to refer to what Fayyad et al. [25] called the data mining phase of the knowledge discovery process.

[2] http://www.e-lico.org

Since it emerged as a distinct research area in machine learning, meta-learning has been cast as the problem of dynamically selecting or adjusting inductive bias [61,74,75,30,77]. There is a consensus that with no restrictions on the space of hypotheses to be explored by the learning algorithm and no preference criterion for comparing candidate hypotheses, then no inductive method can do better on average than random guessing. In short, without bias no learning is possible [51]; the so-called no-free-lunch theorem on supervised learning [79] and the conservation law for generalization performance [63] express basically the same idea. There are two types of bias: representational bias restricts the hypothesis space whereas procedural—aka search or preference—bias gives priority to certain hypotheses over others in this space. The most widely addressed meta-learning tasks—algorithm selection and model selection[3]—can be viewed as ways of selecting or adjusting these two types of bias. Algorithm selection is the choice of the appropriate algorithm for a given task, while model selection is the choice of the specific parameter settings that will produce relatively good performance for a given algorithm on a given task. Algorithm—or model class—selection amounts to adjusting representational bias and model selection to adjusting preference bias. Though there have been a number of studies on model selection [22,23,68,2], meta-learning research has focused predominantly on algorithm selection [73,67,41,43,47,69].

The algorithm selection problem has its origins outside machine learning [66]. In 1976 a seminal paper by John Rice [62] proposed a formal model comprising four components: a problem space $\mathcal{X}$ or collection of problem instances describable in terms of features defined in feature space $\mathcal{F}$, an algorithm space $\mathcal{A}$ or set of algorithms considered to address problems in $\mathcal{X}$, and a performance space $\mathcal{P}$ representing metrics of algorithm efficacy in solving a problem. Algorithm selection can then be formulated as follows: Given a problem $x \in \mathcal{X}$ characterized by $f(x) \in \mathcal{F}$, find an algorithm $\alpha \in \mathcal{A}$ via the selection mapping $S(f(x))$ such that the performance mapping $p(\alpha(x)) \in \mathcal{P}$ is maximized. A schematic diagram of the abstract model is given in Fig. 1.

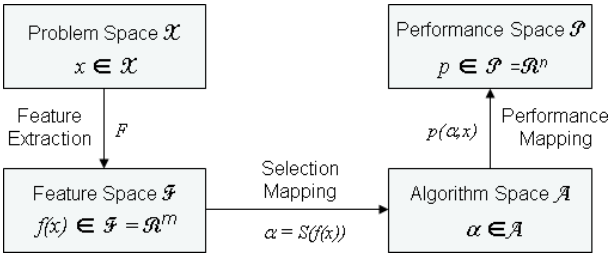


Fig. 1. Rice's model of the algorithm selection problem. Adapted from [62,66]

[3] We take algorithm/model selection to include task variants and extensions such as algorithm/model ranking and algorithm/model combination.

In Rice's model, selection mapping from problem space $\mathcal{X}$ onto algorithm space $\mathcal{A}$ is based solely on features $f \in \mathcal{F}$ over the problem instances. In machine learning terms, the choice of the appropriate induction algorithm is conditioned solely on the characteristics of the learning problem and data. Strangely, meta-learning research has independently abided by the same restriction from its inception to the present. From early meta-learning attempts [61,12] to more recent investigations, the dominant trend relies almost exclusively on meta-data describing the characteristics of base-level data sets used in learning, and the goal of meta-learning has even been defined restrictively as learning a mapping from dataset characteristics to algorithm performance [3]. Researchers have come up with an abundant harvest of such characteristics, in particular statistical and information-theoretic properties of training data [46,36,42,72,16]. A more recent research avenue, dubbed landmarking, characterizes data sets in terms of the predictive performance attained by simple learning algorithms when applied to them [57,8,27,48]; yet another approach describes data sets based on features of the models that were learned from them [8,55]. In all cases, however, the goal is to discover mappings from data set characteristics to learning algorithms viewed essentially as black boxes.

Thus far there has been no attempt to correlate dataset and algorithm characteristics, in other words to understand which aspects of a given algorithm explain its expected performance given the features of the data to be modelled. As a consequence, current meta-learners cannot generalize over algorithms as they do over data sets. To illustrate this problem, suppose that three algorithms are observed to achieve equivalent performance on a collection of datasets representing a task family. Meta-learning would yield three disjunctive rules with identical conditions and distinct recommendations. There would be no way of characterizing in more abstract terms the class of algorithms that would perform well on the given task domain. In short, no amount of meta-learning would reap fresh insights into the commonalities underlying the disconcerting variety of algorithms.

To overcome this difficulty, we propose to extend the Rice framework and pry open the black box of algorithms [37]. To be able to differentiate similar algorithms as well as detect deeper commonalities among apparently unrelated ones, we propose to characterize them in terms of components such as the model structure built, the objective functions and search strategies used, or the type of data partitions produced. This compositional approach is expected to have two far-reaching consequences. Through a systematic analysis of all the ingredients that constitute an algorithm's inductive bias, meta-learning systems (and data miners in the first instance) will be able to infer not only which algorithms work for specific data/task classes but—more importantly—why. In the long term, they should be able to operationalize the insights thus gained in order to combine algorithms purposefully and perhaps design new algorithms. This novel approach to algorithm selection is not limited to the induction phase; it should be applicable to other data and model processing tasks that require search in the space of candidate algorithms. The proposed approach will also be adapted to

model selection, i.e., finding the specific parameter setting that will allow a given algorithm to achieve acceptable performance on a given task. This will require an extensive study of the parameters involved in a given class of algorithms, their role in the learning process or their impact on the expected results (e.g., on the complexity of the learned model for induction algorithms), and their formalization in the data mining ontology.

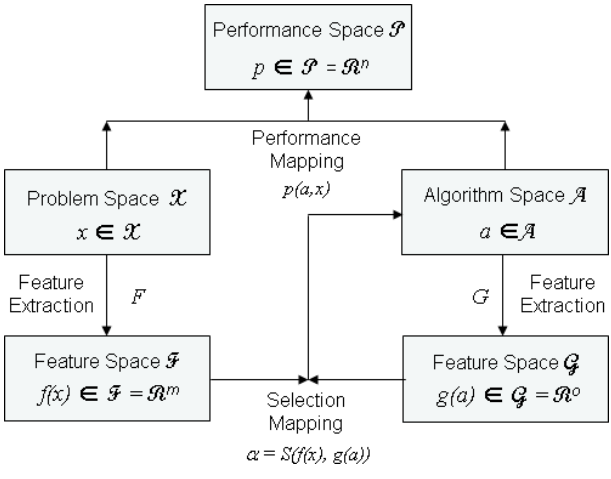


Fig. 2. Proposed model for algorithm selection

The proposed revision of Rice's model for algorithm selection is visualized in Fig. 2. It includes an additional feature space $\mathcal{G}$ representing the space of features extracted to characterize algorithms; selection mapping is now a function of both problem and algorithm features. The revised problem formulation now is: Given a problem $x \in \mathcal{X}$ characterized by $f(x) \in \mathcal{F}$ and algorithms $a \in \mathcal{A}$ characterized by $g(a) \in \mathcal{G}$, find an algorithm $\alpha \in \mathcal{A}$ via the selection mapping $S(f(x), g(a))$ such that the performance mapping $p(a(x)) \in \mathcal{P}$ is maximized.

1.2 Data Mining Ontologies

An ontology is a structure $\mathcal{O} := (\mathcal{C}, \leq_C, \mathcal{R}, \sigma, \leq_R, IR)$ consisting of a set of concepts $\mathcal{C}$ and a set of relations $\mathcal{R}$, a partial order $\leq_C$ on $\mathcal{C}$, called concept hierarchy or taxonomy, a function $\sigma : \mathcal{R} \rightarrow C \times C$ called signature, a partial order $\leq_R$ on $\mathcal{R}$ called relation hierarchy, and a set IR of inference rules expressed in a logical language $\mathcal{L}$ [39]. Before the coming of age of ontological engineering as a distinct research area, there had been early attempts at a systematic description of machine learning and data mining processes. CAMLET [71] used a rudimentary ontology of learning tasks and algorithms to support the automatic composition and revision of inductive processes. While CAMLET focused on model building, the MiningMart project [52] shifted the focus to the preprocessing phase. The

metadata used in MiningMart was condensed into a small ontology whose primary purpose was to allow for the reuse of stored data mining cases. Operator chains, mainly for preprocessing, were described at both the abstract and executable levels to facilitate maintenance of the case base and adaptation of retrieved cases. The taxonomy of data mining operators underlying the IDEA system [9] had a broader scope in the sense that it covered that preprocessing, induction and postprocessing phase of the knowledge discovery process. It had an explicit representation of operator preconditions and effects and was used by an AI-style planner to generate all valid workflows for a given application task. However, unlike CAMLET and MiningMart, where the assembled operator sequences were executed and later revised and reused, IDEA did not go beyond the simple enumeration of valid DM process plans.

The advent of ontology languages and tools for the Semantic Web gave rise to a new generation of data mining ontologies, the majority of which are aimed at the construction of workflows for knowledge discovery. Among these, DAMON [17] and GridMiner Assistant (GMA) [14] focus more specifically on the development of distributed KDD applications on the Grid. DAMON describes available data mining software, their underlying methods and associated constraints in order to enable semantic search for appropriate DM resources and tools. GMA's data mining ontology, written in OWL, is based on industry standards like the CRISP-DM process model [18] and the Predictive Model Markup Language [32]. The ontology is used to support interactive workflow design: GMA first backward-chains from the initial goal/task to compose an abstract task sequence, eliciting user preferences as needed (e.g., to select the preferred type of model). In the second phase, it forward-chains along this sequence to fill in task parameters, either by reasoning from preconditions and effects given in the ontology or by getting user input.

Other ontologies for DM workflow construction are KDDONTO [20], KD Ontology [82] and DMWF [44]. KDDONTO provides knowledge of data mining algorithms required by KDDComposer to build valid DM workflows. Given an algorithm B, the goal is to find the set of algorithms A_i whose outputs can be used as inputs to B. This is done by estimating the degree of match between the expected output of each algorithm A_i and the required input B. Semantic similarity is computed based on the length of the ontological paths between two concepts along the isA and partOf relations. However (dis)similarity is only one component of a score or cost function that takes account of other factors such as estimated performance or the relaxation of constraints on the input of B. This score induces a finer ranking on the candidate workflows and allows for the early disposal of those whose cost exceeds a given threshold.

KD Ontology [82] and a planner are tightly integrated in an automated workflow composition system that has been developed in conformance with proven standards from the semantic web, namely the Web Ontology Language for ontology modelling and the Planning Domain Definition Language (PDDL) for planning. It has a blend of interesting features not found in other related work. Contrary to IDEA and GMA which generate workflows in the form of linear

sequences, it creates workflows as directed acyclic graphs whose nodes represent implemented algorithms; however, these are abstract workflows in the sense that the algorithm parameters need to be instantiated in order to produce executable workflows. In addition, KD Ontology incorporates knowledge about a broad range of data mining algorithms, from standard propositional learners to more advanced algorithms that can handle structured and relational data, thus expanding the power and diversity of workflows that the planner is able to generate. KD Ontology has been tested on two use cases, one in genomics and the other in product engineering.

DMWF and its associated planning environment (eProPlan) [44] have been developed to provide user support in the e-LICO virtual data mining laboratory. A hierarchical task network (HTN) based planner performs a series of task decompositions starting from the initial user task, and generates alternative plans when several methods are available for a given (sub)task. Given the number of operators available to the planner (more than 600 from RapidMiner and Weka alone), the potentially infinite number of valid workflows precludes the approach of enumerating them all and leaving the final choice to the user. Hence the choice of cooperative-interactive workflow planning, in which the planner incrementally expands the current plan and periodically proposes a small number of intermediate extensions or refinements from which the user can choose. The ontology provides the basis for cooperative-interactive workflow planning through the concept of workflow templates, i.e. abstract workflows that can mix executable operators and tasks to be refined later into sub-workflows. These templates serve as the common workspace where user and system can cooperatively design workflows. Automated experimentation can help make intermediate decisions, though this is a viable alternative only when time and computational resources are abundant.

Like the other workflow building systems described above, eProPlan generates a set of correct workflows but has no way of selecting that which is most likely to produce the best results. DMWF models operator preconditions and effects but has no knowledge of the algorithms they implement or the models they are capable of generating. The solution adopted in the e-LICO virtual DM lab is to couple the workflow generator with a meta-miner whose role is to rank the workflows or select the most promising ones based on lessons learned from past data mining experience. The meta-miner relies extensively on deep knowledge of data mining algorithms' biases and capabilities modelled in DMOP (Section 2).

As mentioned above, the vast majority of existing DM ontologies are aimed at supporting workflow construction. One exception is OntoDM [53], whose declared goal is to provide a unified framework for data mining [24]. It contains definitions of the basic concepts used in data mining (e.g., DM task, algorithm, dataset, datatype), which can be combined to define more complex entities such as constraints or data mining experiments. The distinguishing feature of OntoDM is its compliance with ontological best practices defined mainly in the field of biological investigations. It uses a number of upper level ontologies such as Basic Formal Ontology (BFO), the OBO Relation Ontology (RO), and the Information Artefact Ontology (IAO). Its structure has been aligned with the Ontology

of Biological Investigations (OBI), and its development follows strict rules like avoiding multiple inheritance and limiting the number of relations. OntoDM is called a general-purpose ontology by its authors and remains to be applied to a concrete data mining use case. More recently, a similar ontology called Exposé [76] has been developed to provide the conceptual basis for a database of data mining experiments [11]. Exposé borrows OntoDM's upper level structure and DMOP's conceptualization of data mining algorithms, and completes these with a description of experiments that serves as the basis of an Experiment Markup Language. OntoDM's and Exposé's alignment with upper ontologies suggests that their primary use is to provide a controlled vocabulary for DM investigations. Among the DM ontologies that do not focus on workflow construction, DMOP is unique in its focus on the problem of optimizing the knowledge discovery process through an in-depth characterization of data and especially of DM algorithm biases and internal mechanisms.

2 An Ontology for Data Mining Optimization

2.1 Objectives and Overview

The overall goal of DMOP is to provide support for all decision-making steps that have an impact on the outcome of the knowledge discovery process. It focuses specifically on DM tasks (e.g., learning, feature extraction) whose accomplishment requires non-trivial search in the space of alternative methods. For each such task, the decision process involves two steps that can be guided by prior knowledge from the ontology: *algorithm selection* and *model selection*. While data mining practitioners can profitably consult DMOP to perform "manual" algorithm and model selection, the ontology has been designed to automate these two operations. Thus a third use of DMOP is *meta-learning*, i.e., the analysis of meta-data describing learning episodes in view of extracting patterns and rules to improve algorithm and model selection. Finally, generalizing meta-learning to the complete DM process, DMOP's most innovative objective is to support *meta-mining* or the meta-analysis of complete data mining experiments in order to extract workflow patterns that are predictive of good or bad performance. In short, DMOP charts the higher-order feature space in which meta-learning and meta-mining can take place.

The DMOP ontology's overall structure and foundational role in meta-mining are illustrated in Figure 3. DMOP provides a conceptual framework that defines the relationships among the core DM entities such as tasks, algorithms, models, workflows, experiments (Section 2.2). The hierarchy of concepts (classes), together with axioms expressing their properties, relations and restrictions, constitute the terminological box (TBox), or what we can call the ontology proper. Based on this conceptual groundwork, individuals are created as instances of one or several concepts from the TBox; these individuals, and all statements concerning their properties or their relations with other individuals, form the assertional box (ABox), also called the knowledge base.

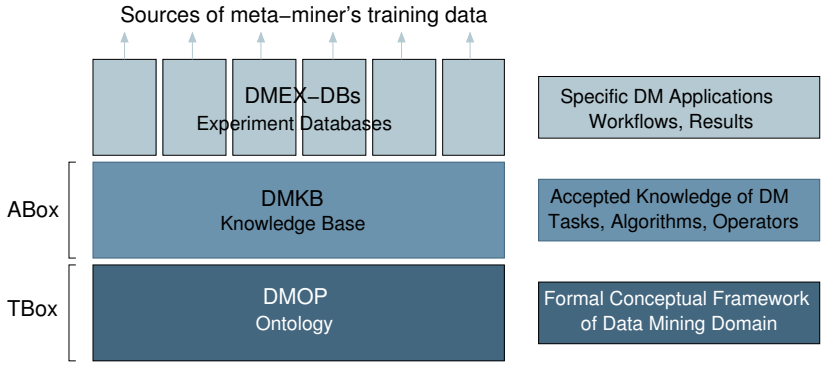


Fig. 3. The DMOP architecture

The **DM knowledge base (DMKB)** captures the DM community's collective expertise; ideally, it would be a compendium of the state of the art in data mining. DMKB builds on DMOP's taxonomy of major data mining tasks and paradigms (broad algorithm families) to classify and characterize individual algorithms that have been developed to date, together with their better known implementations. For instance, DMKB contains formal descriptions of algorithms most often used to solve classification tasks: generative approaches such as Naïve Bayes, discriminative approaches such as Logistic Regression, and discriminant function approaches such as SVMs. To distinguish individual variants of a given algorithm family (e.g. NaiveBayesNormal, NaiveBayesKernel, NaiveBayesDiscretized, MultinomialNaiveBayes, ComplementNaiveBayes), each is described giving specific values to properties defined in the DM ontology. Similarly, operators from DM packages are analysed to identify the algorithms they implement, so that all attempts to explain an operator's performance go beyond low-level programming considerations to reason on the basis of algorithm assumptions and basic components.

DM Experiment data bases (DMEX-DBs) are built using concept and property definitions from DMOP as well as concrete algorithm and operator definitions from DMKB. In contrast to DMKB, which is a compilation of commonly accepted data mining knowledge, a DMEX database is any collection of experimental data concerning a given data mining application task. It is usually domain-specific and contains ground facts about clearly defined use cases, their associated data sets, actual data mining experiments conducted to build predictive or descriptive models that address the task, and the estimated performance of these models. Thus any number of DM experimental data bases can be built with schemas based on DMOP and DMKB.

2.2 The Core Concepts

To develop the DM concept hierarchy, we start with the two endpoints of the DM process. At one end, the process receives input data relative to a given

discovery task; at the other, it outputs knowledge in the form of a descriptive or predictive model, typically accompanied by some kind of report containing the learned model's estimated performance and other meta-data. These three concepts—Data, DM-Model, DM-Report—play a central role in DMOP and have been grouped, for convenience, in a derived class called IO-Object. The major concept hierarchies of the ontology—DM-Task, DM-Algorithm, DM-Operator and DM-Workflow—are structured directly or indirectly by these three types of input/output objects.

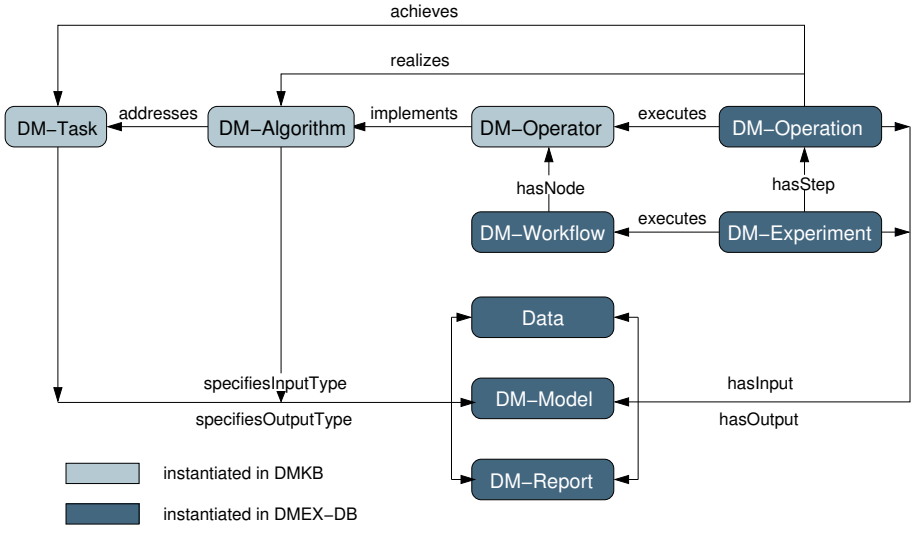


Fig. 4. The core concepts of DMOP

Tasks and algorithms as defined in DMOP are not processes that directly manipulate data or models, rather they are specifications of such processes. A DM-Task is a specification of any piece of work that is part of the DM process, essentially in terms of the input it requires and the output it is expected to produce. A DM-Algorithm is the specification of a procedure that addresses a given Task, while a DM-Operator is a program that implements a given DM-Algorithm (see Figure 4). Instances of DM-Task and DM-Algorithm do no more than specify their input/output types; only processes called DM-Operations have actual inputs and outputs. A process that executes a DM-Operator also realizes the DM-Algorithm implemented by the operator and by the same token achieves the DM-Task addressed by the algorithm. Finally, just as a DM-Workflow is a complex structure composed of DM operators, a DM-Experiment is a complex process composed of operations (or operator executions). A workflow can be represented as a directed acyclic graph in which nodes correspond to operators and edges to IO-Objects, i.e. to the data, models and meta-level reports consumed and produced by DM operations. An experiment is described by all the objects that participate in the

process: a workflow, data sets used and produced by the different data processing phases, the resulting models and meta-data quantifying their performance. Instances of DM-Algorithm and DM-Operator are described in the DMKB because they represent consensus data mining knowledge, while instances of DM-Workflow and DM-Experiment are stored in application-specific DM experiment data bases.

Data. As the critical resource that feeds the knowledge discovery process, data are a natural starting point for the development of a data mining ontology. Over the past decades many researchers have actively investigated data characteristics that might explain generalization success or failure. An initial set of such statistical and information-theoretic measures was gathered in the StatLog project [50] and extended in the Metal project with other statistical [46], landmarking-based [57,7]and model-based [55,56] characteristics. Data descriptors in DMOP are based on the Metal heritage, which we further extended with geometrical measures of data complexity [6].

Figure 5 shows the descriptors associated with the different Data subclasses. Most of these are statistical measures, such as the number of instances or the number of features of a data set, or the absolute or relative frequency of a categorical feature value. Others are information-theoretic measures (italicized in the figure) ; examples are the entropy of a categorical feature or the class entropy of a labelled dataset. Characteristics in bold font, like the max Fisher's discriminant ratio, which measures the highest discriminatory power of any single feature in the data set, or the fraction of data points estimated to be on the class boundary, are geometric indicators of data set complexity; detailed definitions

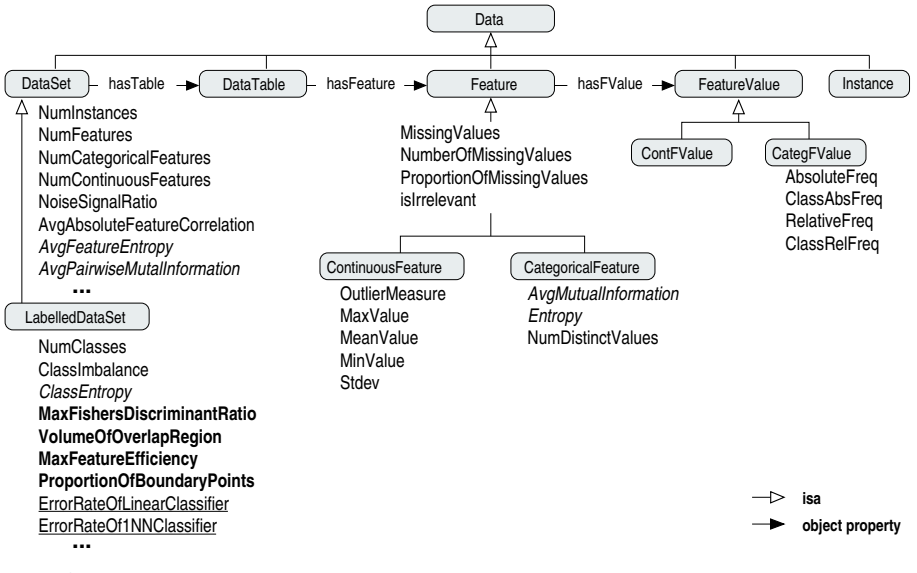


Fig. 5. Data characteristics modelled in DMOP

of these characteristics can be found in [38]. Finally, error rates such as those of a linear or a 1-NN classifier (underlined) are data characteristics based on landmarking, which was briefly described in Section 1.1.

DM Tasks. As mentioned above, DMOP places special emphasis on so-called core DM tasks—search-intensive or optimization-dependent tasks such as feature construction or learning, as opposed to utility tasks such as reading/writing a data set or sorting a vector of scalars.

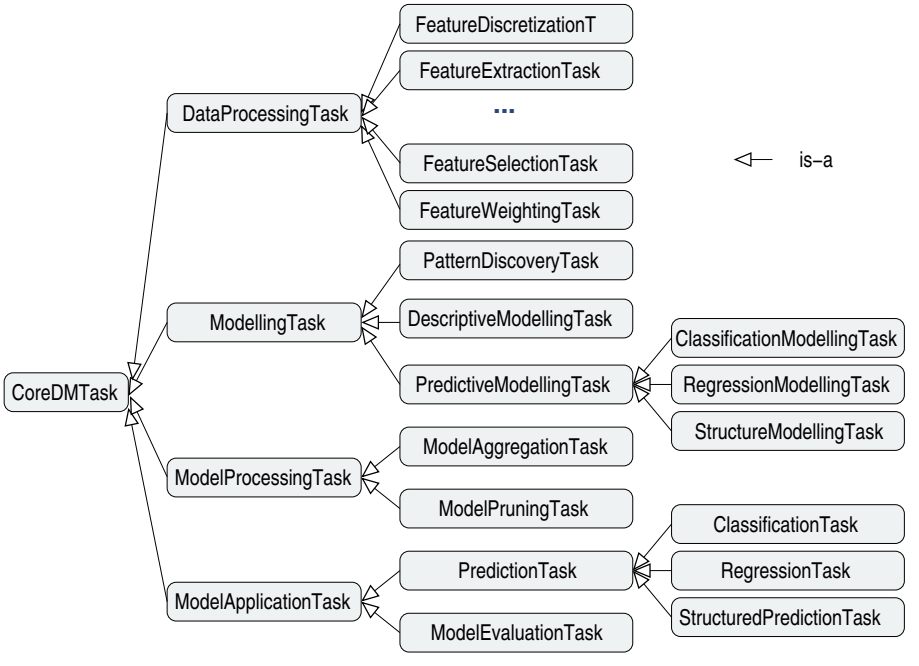


Fig. 6. The CoreDMTask Hierarchy

The CoreDMTask hierarchy (Fig. 6) comprises four major task classes defined by their inputs and outputs: data processing, modelling, model processing, and model application:

$\texttt{DataProcessingTask} \sqsubseteq \forall \texttt{specifiesInputType}.\underline{\text{Data}} \sqcap \forall \texttt{specifiesOutputType}.\underline{\text{Data}}$
$\texttt{ModellingTask} \sqsubseteq \forall \texttt{specifiesInputType}.\underline{\text{Data}} \sqcap \forall \texttt{specifiesOutputType}.\underline{\text{Model}}$
$\texttt{ModelProcessingTask} \sqsubseteq \forall \texttt{specifiesInputType}.\underline{\text{Model}} \sqcap \forall \texttt{specifiesOutputType}.\underline{\text{Model}}$
$\texttt{ModelApplicationTask} \sqsubseteq \forall \texttt{specifiesInputType}.\underline{\text{Model}} \sqcap \forall \texttt{specifiesOutputType}.\underline{\text{Report}}$

Specializations of each task are defined by specializing its input and output types. As we move down the tree in Figure 6, the descendant classes of ModellingTask

specify input and output types that are successively more specific subclasses of Data and Model respectively:

PredictiveModellingTask $\sqsubseteq$ $\forall$specifiesInputType.LabelledDataSet
$\sqcap \forall$specifiesOutputType.PredictiveModel
ClassificationModellingTask $\sqsubseteq$ $\forall$specifiesInputType.CategoricalLabelledDataSet
$\sqcap \forall$specifiesOutputType.ClassificationModel

Note the distinction between PredictiveModellingTask — the construction of a predictive model — and PredictionTask, which is the simple application of the model built through predictive modelling. The same distinction holds between their respective subclasses, e.g. classification is the application of a classifier built through classification modelling, and similarly for regression and structured prediction. This is in contrast to current usage in the literature, where the term classification, for instance, designates ambiguously the process of building or applying a classifier.

DM Algorithms. The top levels of the Algorithm hierarchy reflect those of the Task hierarchy, since each algorithm class is defined by the task it addresses, e.g. DataProcessingAlgorithm $\equiv$ Algorithm $\sqcap \exists$ addresses.DataProcessingTask. However, the Algorithm hierarchy plunges more deeply than the Task hierarchy: for each leaf class of the task hierarchy, there is an often dense subhierarchy of algorithms that specify diverse ways of addressing each task. For instance, the leaf concept ClassificationModellingTask in Figure 6 is mapped directly onto the hierarchy rooted in the concept of ClassificationModellingAlgorithm in Figure 7.

As shown in the figure, classification modelling algorithms are divided into three broad categories [10]. *Generative methods* compute the class-conditional densities $p(\mathbf{x}|C_k)$ and the priors $p(C_k)$ for each class C_k, then use Bayes' theorem to find posterior class probabilities $p(C_k|\mathbf{x})$. They can also model the joint distribution $p(\mathbf{x}, C_k)$ directly and then normalize to obtain the posteriors. In both cases, they use statistical decision theory to determine the class for each new input. Examples of generative methods are normal (linear or quadratic) discriminant analysis and Naive Bayes. *Discriminative methods* such as logistic regression compute posteriors $p(C_k|\mathbf{x})$ directly to determine class membership. *Discriminant functions* build a direct mapping $f(\mathbf{x})$ from input $\mathbf{x}$ onto a class label; neural networks and support vector classifiers (SVCs) are examples of discriminant function methods. These three Algorithm families spawn multiple levels of descendant classes that are distinguished by the type and structure of the models they generate; model structures will be discussed in Section 2.3.

In addition to these primitive classes that form a strict hierarchy (as shown in Figure 7), equivalent class definitions superpose a finer structure on the Algorithm subclasses. For instance, we can distinguish between eager and lazy learners based on whether they compress training data into ready-to-use models or simply store the training data, postponing all processing until a request for prediction is received [1]. Similarly, a classification algorithm can be classified as high-bias or high-variance based on how it tends to control the bias-variance trade-off in its learned models [28,21]. High-bias algorithms can only generate simple models

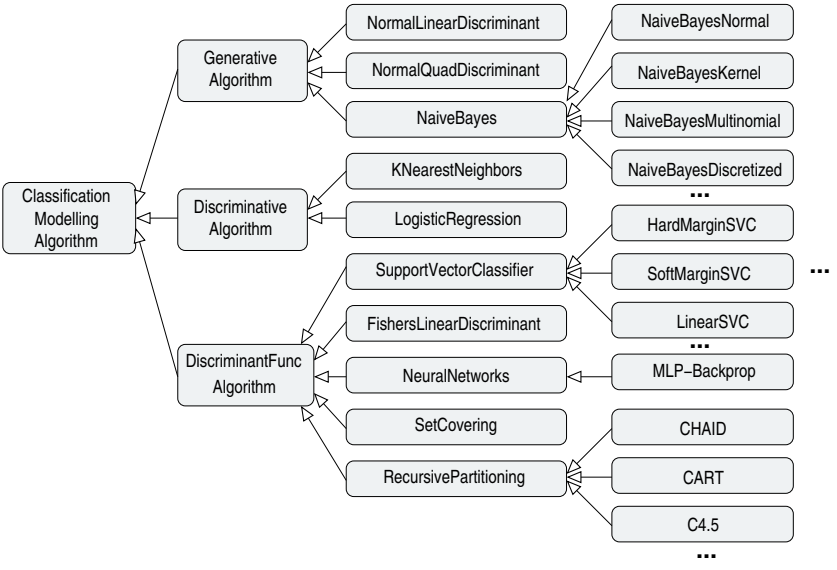


Fig. 7. The ClassificationModellingAlgorithm hierarchy. Only the primitive class hierarchy is shown.

that lack the flexibility to adapt to complex data distributions but for that reason remain stable across different training samples. High-variance algorithms span a broader range of complexity; they can generate highly complex but often unstable models: a slight change in the training sample can yield large changes in the learned models and their predictive behavior. Many other equivalent classes can be defined for modelling algorithms; as a result, algorithm instances can have multiple inheritance links (not shown in the figure) that make this concept hierarchy more of a directed acyclic graph than a simple tree structure.

2.3 Inside the Black Box: A Compositional View of DM Algorithms

As explained in Section 1, a key objective of the proposed meta-mining approach is to pry open the black box of DM algorithms in order to correlate observed behavior of learned models with both algorithm and data characteristics. This is a long-term, labor-intensive task that requires an in-depth analysis of the many data mining algorithms available. In this section, we illustrate our approach on two major data mining tasks, classification modelling and feature selection.

Classification Modelling Algorithms. Opening the black box of a modelling or learning algorithm is equivalent to explaining, or describing the sources of, its inductive bias (Section 1.1). DMOP provides a unified framework for conceptualizing a learning algorithm's inductive bias by explicitly representing: 1) its underlying assumptions; 2) its hypothesis language or so-called representational bias through a detailed conceptualization of the class of models it generates; and

3) its preference or search bias through a definition of its underlying optimization problem and the optimization strategy adopted to solve it.

Representational bias and models. As its name suggests, the keystone of a modelling algorithm is the **Model** that it was designed to produce (ModellingAlgorithm $\sqsubseteq \exists$ specifiesOutput.Model). A detailed characterization of a modelling algorithm's target model is the closest one can get to an actionable statement of its representational bias or hypothesis language. DMOP's characterization of ClassificationModel is summarized in Figure 8. To clarify how the model-related and other relevant concepts are used in describing a classification algorithm, we will use the linear soft-margin SVM classification modelling algorithm (henceforth LinSVC-A for the algorithm and LinSVC-M for the generated model) represented in Figure 9 as our running example.

A model is defined by two essential components: a model structure and a set of model parameters. The **ModelStructure** determines the three main classes of classification models (and hence of classification modelling algorithms). From the definitions given in Section 2.2, it follows that the model structure of a GenerativeModel is a JointProbabilityDistribution, while that of a DiscriminativeModel is a PosteriorProbabilityDistribution. By contrast, DiscriminantFunctionModels compute direct mappings of their inputs to a class label by summarizing the training data in a LogicalStructure (e.g., decision tree, rule set) or a MathematicalExpression (e.g., superposition of functions in neural networks, linear combination of kernels in SVMs). In LinSVC-M, where the kernel itself is linear, the linear combination of kernels is equivalent to a linear combination of features (Fig. 9).

The concept of **ModelParameter** is indissociable from that of ModelStructure. Within each model family, more specific subclasses and individual models are produced by instantiating the model structure with a set of parameters. Probabilistic — generative and discriminative — model structures are unambiguously specified by the probability distribution that generated the training data. Since this distribution can never be identified with certainty from a finite random sample, the task is often simplified by assuming a family of distributions (e.g., Gaussian in NaiveBayesNormal) or a specific functional form (e.g., the logistic function in LogisticRegression); estimating the probability distribution is then reduced to estimating the values of the distribution parameters (e.g., mean and variance of a Gaussian) or the function parameters (e.g., weights of the linear combination in the logistic function's exponent). In DMOP, the concept of ProbabilisticModelStructure has a specific property, hasDensityEstimation, that identifies the parametric or non-parametric density estimation method used to estimate the model parameters. In non-probabilistic (discriminant function) models, the nature of the model parameters varies based on the type of model structure. In logical structures, which are more or less complex expressions based on the values of individual features, the model parameters are thresholds on feature values that partition the instance space into hyperrectangular decision regions. The natural model parameters of mathematical model structures are the values of the underlying function parameters, e.g. the weights of the hidden units in a neural network or the kernel coefficients in SVMs. In LinSVC-M (Fig. 9), the

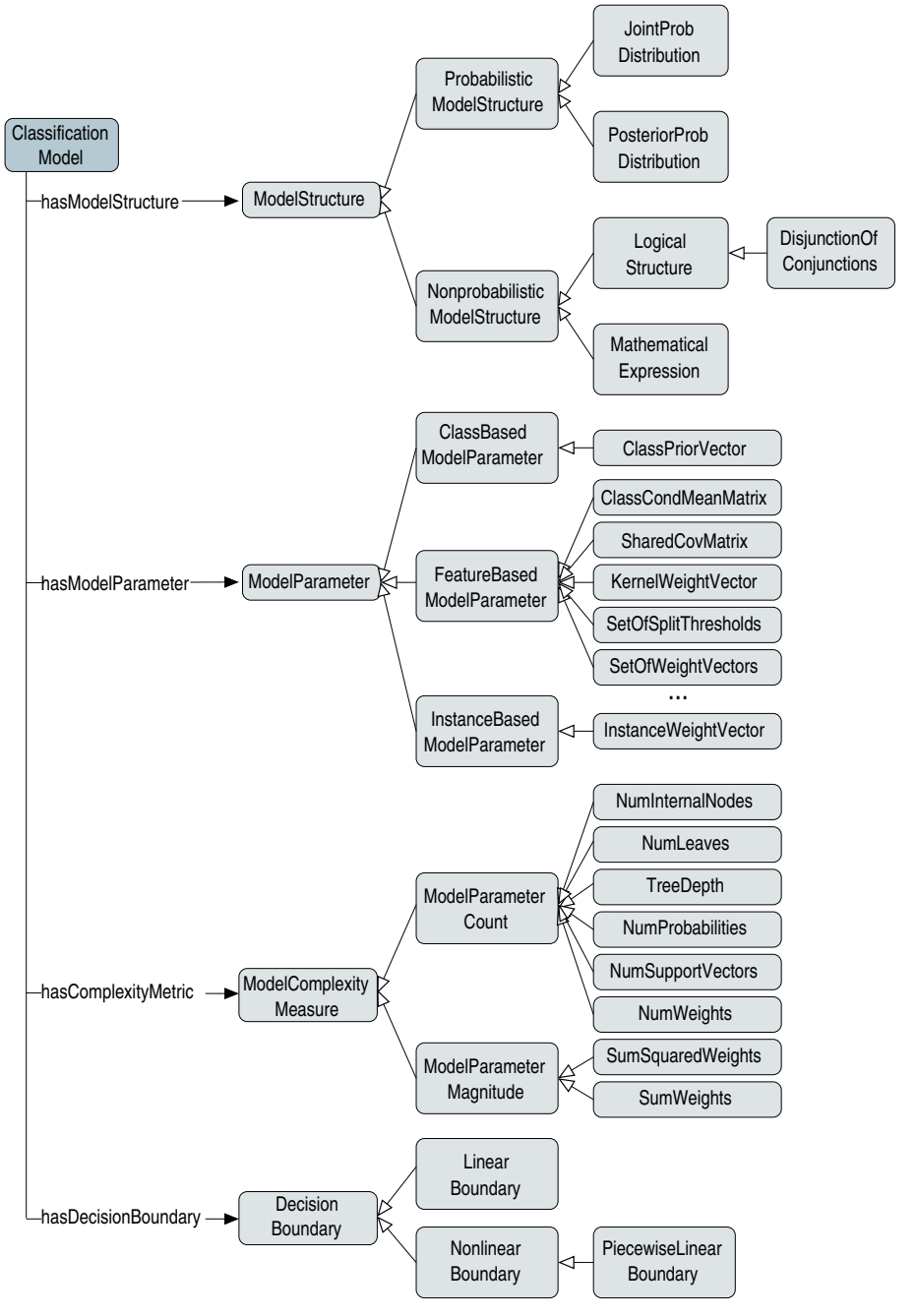


Fig. 8. DMOP's conceptualization of ClassificationModel

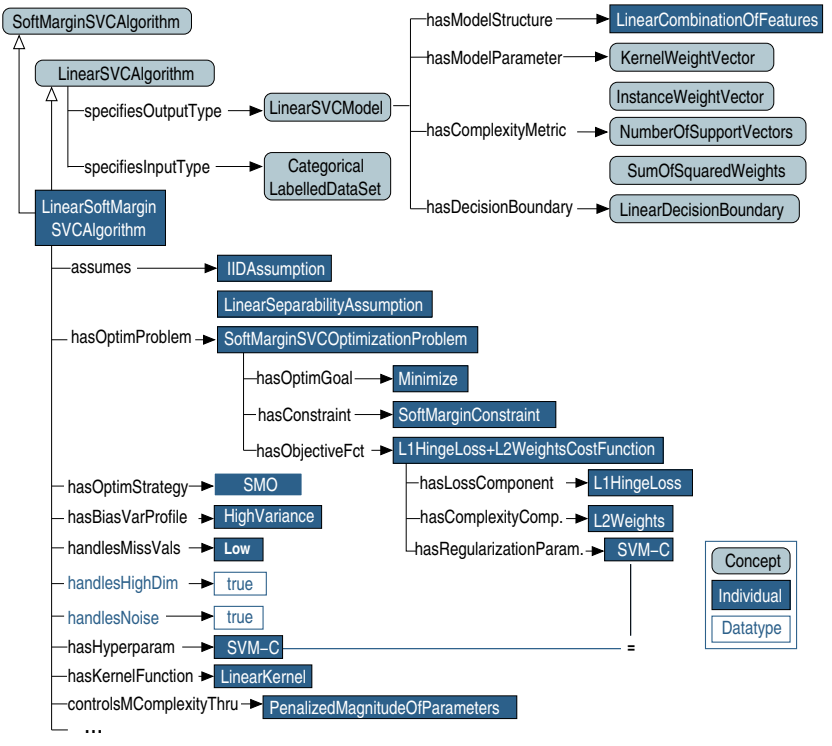


Fig. 9. Linear Soft Margin SVC Algorithm (referred to in the text as LinSVC-A) and the model it specifies (LinSVC-M in the text).

model parameters are the instance weights and the kernel weights which, as we have seen above, are those of the feature themselves.

In all cases, the number of model parameters confers on the selected model structure the degrees of freedom required to capture the characteristics of the target population. A model that has an inadequate set of parameters will underfit the data and incur systematic errors due to bias; on the other hand, a model with too many model parameters will adapt to chance variations in the sample, in short will overfit the training data and perform poorly on new data due to high variance. Selecting the right number of parameters is no other than selecting the right bias-variance tradeoff or selecting the appropriate capacity or level of complexity for a given model structure. The complexity of each learned model can be quantified using the concept of **ModelComplexityMeasure**, the most important subclass of which is ModelParameterCount. Its sibling, ModelParameterMagnitude, takes into account the view that a model's complexity is also determined by the magnitude of model parameter values [19,5]. The two complexity measures of LinSVC-M (Fig. 9) are NumberOfSupportVectors and SumOfSquaredWeights, subclasses of ModelParameterCount and ModelParameterMagnitude respectively. A final model descriptor is the type of **DecisionBoundary** that is drawn by a given model (family). DMOP's formalization of this concept distinguishes between linear and

nonlinear decision boundaries, but more work is needed to develop a more elaborate geometry of decision regions.

Preference bias and optimization strategies. Once a model structure and its set of parameters have been selected, the learning process is nothing more or less than the automated adjustment of these parameters to produce a fully specified, operational model. This is the task of the learning algorithm. The goal is to determine the set of parameter values that will maximize classification performance as gauged by some criterion. The search for the right parameter setting can be cast as an **OptimizationProblem** that consists in minimizing a cost (or objective) function, with or without a corresponding set of constraints. The cost function quantifies how close the current parameter values are to the optimum. Learning stops when the cost function is minimized. In its simplest version, the cost function is no more than a measure of error or loss (e.g. misclassification rate or sum of squared errors). However, minimizing training set error can lead to overfitting and generalization failure. For this reason many algorithms use a regularized cost function that trades off loss against model complexity. In DMOP, the generic form of the modelling CostFunction is $F = \epsilon + \lambda c$, where ϵ is a measure of loss, c is a measure of model complexity, and λ is a regularization parameter which controls the trade-off between loss and complexity. The optimization problem addressed by the LinSVC-A consists in minimizing the regularized cost function

$$\min_{\xi,\mathbf{w},b} \langle \mathbf{w}.\mathbf{w} \rangle + C \sum_{i=1}^{n} \xi_i^2$$

subject to the soft margin constraint $y_i(\langle \mathbf{w}, \Phi(\mathbf{x}_i) \rangle + b) \geq 1 - \xi_i$, with $\xi_i \geq 0$, and $i = 1, ..., n$. The ontological representation of this optimization problem is shown (labelled SoftMarginSVCOptimizationProblem) in Figure 9.

DMOP incorporates a detailed hierarchy of strategies adapted to the optimization problems encountered in modelling and in other DM tasks (Fig. 10). These **OptimizationStrategies** fall into two broad categories—continuous and discrete—depending on the type of variables that define the problem. In certain cases, optimization is straightforward. This is the case of several generative algorithms like normal linear/quadratic discriminant analysis and Naive Bayes-Normal, where the cost function is the log likelihood, and the maximum likelihood estimates of the model parameters have a closed form solution. Logistic regression, on the other hand, estimates the maximum likelihood parameters using methods such as Newton-Raphson. In the case of LinSVC-A, the variables involved in the optimization problem defined above call for a continuous optimization strategy. LinSVC-A uses Sequential Minimal Optimization (SMO), a quadratic programming method rendered necessary by the quadratic complexity component of the cost function (L_2 norm of Weights in Fig. 9).

The optimization strategy hierarchy plays an important role in DMOP because many core DM tasks other than modelling also have underlying optimization problems. In particular, discrete optimization strategies will come to the fore in feature selection methods.

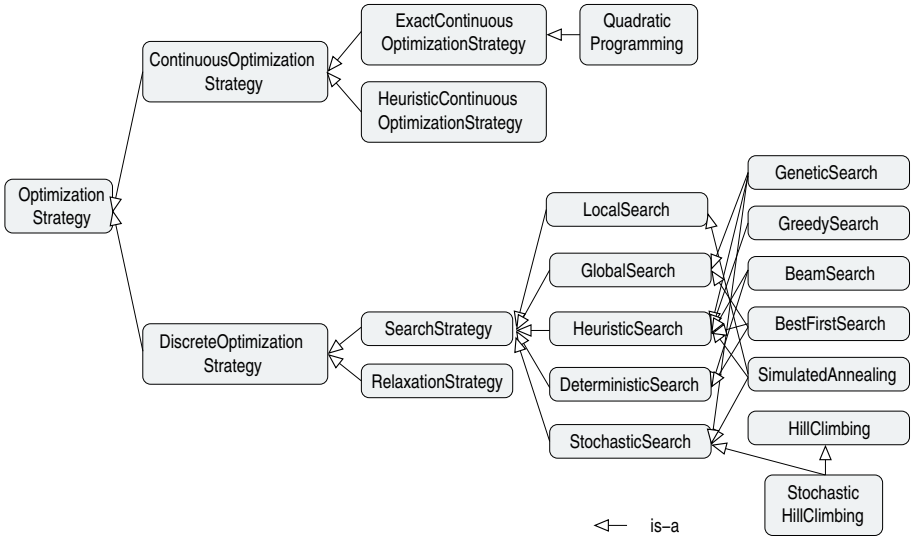


Fig. 10. The OptimizationStrategy hierarchy

Feature Selection Algorithms. Feature selection is a particular case of dimensionality reduction, which can be defined as follows: given a set of n vectors $\{\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_n\} \in \mathbb{R}^p$, find a set of lower-dimensional vectors $\{\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_n\} \in \mathbb{R}^{p'}$, $p' < p$, that maximally preserves the information in the data according to some criterion. In a classification task, for instance, the criterion could be some measure of how well the p or $p\prime$ features discriminate between the different classes. Feature selection refers to the specific case where the p' features are a subset of the original p features; dimensionality is reduced by eliminating irrelevant or redundant features. Alternatively, new features can constructed from the original ones via techniques like principal components analysis, and feature selection applied to the set of derived features; this process—feature construction followed by feature selection—is called feature extraction in [33] and in DMOP.

In feature selection, every subset of the original p-dimensional feature vector is represented by a vector $\sigma \in \{0,1\}^p$ of indicator variables, where $\sigma_i = 1$ denotes the presence and $\sigma_i = 0$ the absence of feature i. The task is to find a vector $\sigma^* \in \{0,1\}^p | \forall \sigma'$, $f(\sigma^*) \leq f(\sigma')$, where f is some measure of feature set quality. A feature selection algorithm can be described by four properties: its mode of interaction with the learning algorithm, an optimization strategy to guide search in the space of feature subsets, a feature scoring or weighting mechanism to assess the candidate subsets, and a decision strategy to make the final selection.

Interaction with the learner. Feature selection methods are commonly classified based on how they are coupled with the learning algorithm. Filter methods perform feature selection as a preprocessing step, independently of the learning method; they must then use learner-independent relevance criteria to evaluate the candidate features, either individually or as subsets of the initial feature set.

Wrapper methods wrap feature selection around the learning process and use the estimated performance of the learned model as the selection criterion; the effectiveness of the selected features depends strongly on the specific learning method used. In embedded methods, feature selection is encoded as an integral part of the learning algorithm.

Optimization strategies. Feature selection implies extensive search in the discrete space of feature subsets; there are 2^p ways of assigning values to the p-dimensional vector σ, in other words 2^p possible subsets of the initial feature set. Feature selection methods can adopt one of two optimization strategies to solve this kind of problem: SearchStrategy and RelaxationStrategy. Search strategies are based on the combinatorial approach that is a more natural approach to problems in discrete domains, while relaxation strategies relax, as it were, the discreteness constraint and reformulate the problem in a continuous space. The result is then reconverted via a decision rule into a final selection in discrete feature space. Figure 10 shows the two main types of DiscreteOptimizationStrategy. Search strategies, in particular heuristic search strategies that trade off optimality for efficiency or simple feasibility, are by far the most widely used. The subclasses of SearchStrategy are determined by the different properties of search as shown in Figure 11) : its coverage (global or local), its search direction (e.g., forward, backward), its choice policy or what Pearl calls "recovery of pursuit" [54] (irrevocable or tentative), the amount of state knowledge that guides search (blind, informed), and its level of uncertainty (deterministic, stochastic). These properties are For instance, Consistency-based feature selection [49] uses the so-called Las Vegas strategy which is an instance of StochasticHillClimbing, which combines local, greedy, stochastic search. Correlation-based feature selection [35] adopts a forward-selection variant of (non-greedy) BestFirstSearch. Representing the irrevocable choice policy of GreedySearch, C4.5's embedded feature selection algorithm and SVM-RFE [34] use GreedyForwardSelection and GreedyBackwardElimination respectively. The concept RelaxationStrategy has no descendants in the graph because after transposing the discrete problem into a continuous space, on can use any instance of ContinuousOptimizationStrategy. However, most of the feature selection algorithms that use relaxation further simplify the problem by assuming feature independence, reducing the combinatorial problem to that of weighting the p individual features and (implicitly) selecting a subset composed of the top p' features. This is the case of all so-called univariate methods, such as InfoGain, χ^2 and SymmetricalUncertainty (see Figure 12), as well as a few multivariate methods like ReliefF [45,64] and SVMOne. ReliefF solves the continuous problem similarly to univariate methods because it also weights individual features, though in a multivariate context. On the contrary, SVMOne and SVM-RFE use the continuous optimization strategy of the learner in which they are embedded — SMO, which, as we saw above is an instance of QuadraticProgramming. Finally, note the special case of SVM-RFE which actually combines the two discrete optimization strategies: it generates candidate subsets through greedy backward elimination in discrete space, then uses the SVM learner to weight the individual features in continuous space, and finally returns to discrete space by

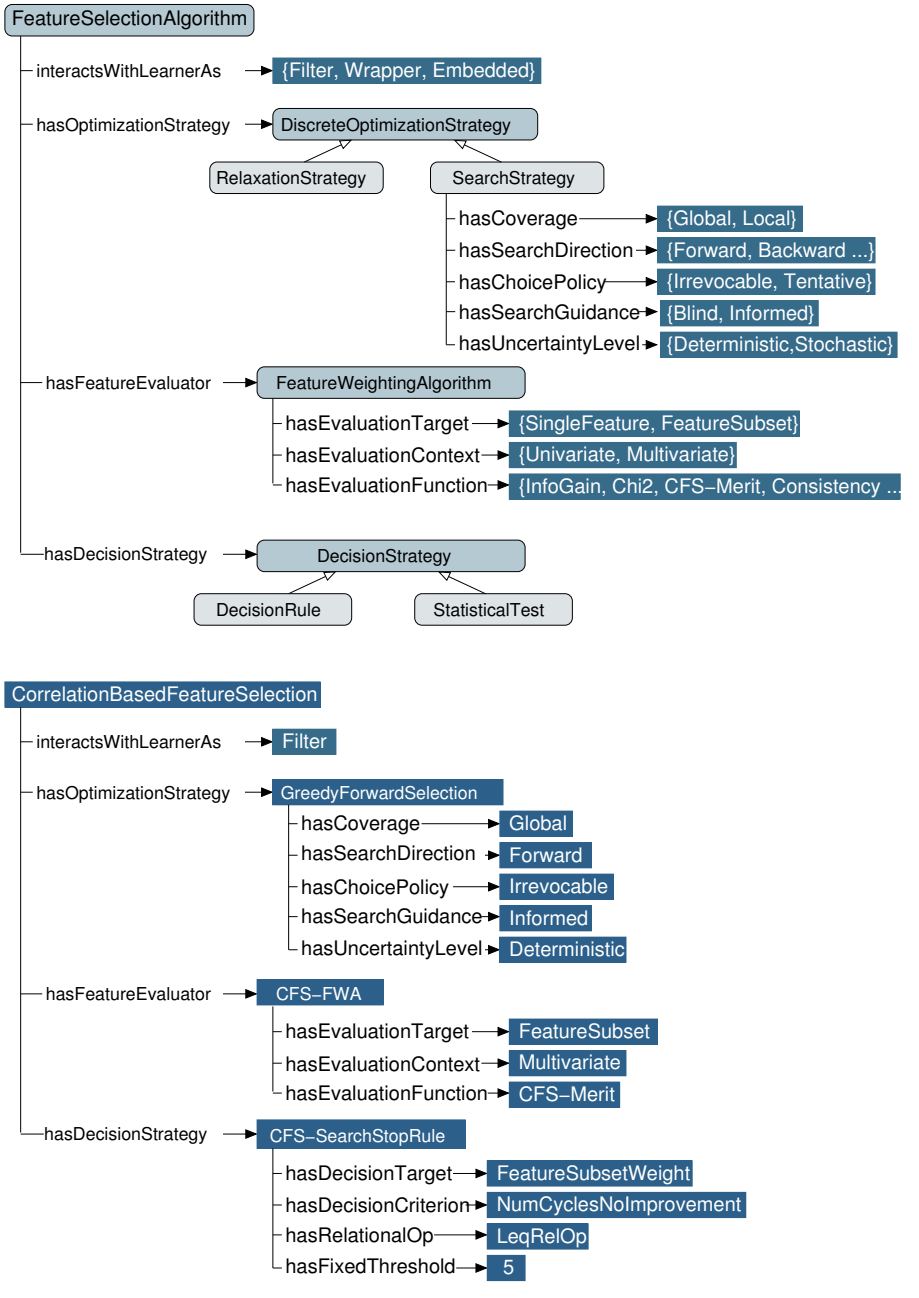


Fig. 11. Algorithms and strategies for feature selection. The upper part of the figure shows the links between the major entities involved: an optimization strategy, a feature weighting algorithm and a decision strategy. The lower part illustrates the use of these concepts in describing Correlation-based Feature Selection.

generating a new subset purged of the n features with the lowest weights. This cycle continues until there are no more features to eliminate.

Feature/subset weighting schemes. Another characteristic of a feature selection algorithm is its feature weighting scheme. Feature weighting algorithms are divided into two groups based on what is being weighted (hasEvaluationTarget property in Fig. 11): individual features or feature subsets. Single-feature weighting algorithms themselves can be classified as univariate or multivariate depending on the feature context that is brought to bear in weighting the individual feature: univariate algorithms (e.g., those that use information gain or χ^2) weight individual features in isolation from the others, while multivariate algorithms weight individual features in the context of all the others. For instance, ReliefF and SVMOne yield individual feature weights that are determined by taking all the other features into account — when computing nearest neighbors in the case of ReliefF, and in building the linear combination of features or kernels in the case of SVMOne. Finally, feature weighting algorithms are completely specified by adding the evaluation function they use – either individual feature or feature subset weighting algorithms.

Decision strategy. Once the candidate features or feature subsets have been generated and scored, a feature selection algorithm uses a decision strategy to select the fnal feature subset. This can be a statistical test that uses the resulting p-value as a basis for selection, or any kind of decision rule that sets a threshold on any quantity that describes the evaluated entities, e.g., the weights of the features or subsets, or their ranks.

Figure 12 situates a number of feature selection algorithms according to their characteristics and those of their feature weighting components.

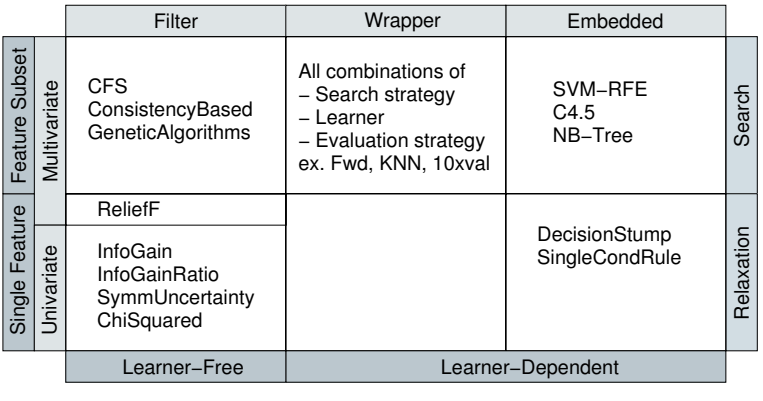


Fig. 12. Synoptic view of feature selection methods based on their interaction with the learning algorithm (learner-free=filter, learner-dependent=wrapper, embedded), the optimization strategy used (search, relaxation), and their feature weighting component's evaluation target (single feature, feature subset) and evaluation context (univariate, multivariate).

3 DMOP-Based Pattern Discovery from DM Workflows

Data mining workflows are replete with structures that are often reused. A simple example is the workflow segment where the operator Weight by Information Gain is invariably followed by Select by Weights to perform feature selection. This regularity involves individual operators, but it would be even more useful if we could detect the same basic structure had the first operator been replaced by any other that does univariate feature weighting. Similarly, bagging subworkflows should be recognizable despite the diversity of classification modelling operators used. In order to detect patterns with strong support, a frequent pattern search procedure should be capable of generalizing from specific operators to broad algorithm classes. This is one of the roles of the DMOP ontology, where we can follow the executes link from grounded operations to operators, then the implements link from operators to algorithms (Figure 4) in order to analyse the taxonomic (as in Figure 7) and non-taxonomic commonalities between algorithms. In short, prior knowledge modelled in DMOP will support the search for generalized workflow patterns, similar to the generalized sequence patterns extracted via frequent sequence mining in [70].

3.1 Workflow Representation for Generalized Pattern Mining

Workflows as hierarchical graphs. Data mining workflows are directed acyclic graphs (DAGs), in which nodes correspond to operators and edges between nodes to input/output (I/O) objects, much like the "schemes" described in [40,31]. More precisely, they are hierarchical DAGs, since nodes can represent composite operators (e.g. cross-validation) that are themselves workflows. An example hierarchical DAG representing a RapidMiner workflow is given in Figure 13. The workflow cross-validates feature selection followed by classification model building. X-Validation is a typical example of a composite operator which itself is a workflow. It has two basic blocks, a *training block* which can be any arbitrary workflow that receives as input a dataset and outputs a model, and a *testing block* which receives as input a model and a dataset, and outputs a performance measure. In this specific CV operator, the training block has three steps: computation of feature weights by the Weight by Information Gain operator, selection of a subset of features by the Select by Weights operator, and final model building by the Naive Bayes operator. The testing block consists simply of the Apply Model operator followed by the Compute Performance computation.

We now give a more formal definition of the hierarchical DAGs that we will use to describe data mining workflows. Let:

- O be the set of all available operators that can appear in a data mining workflow, e.g. classification operators, such as C4.5, SVMs, model combination operators, such as boosting, etc.

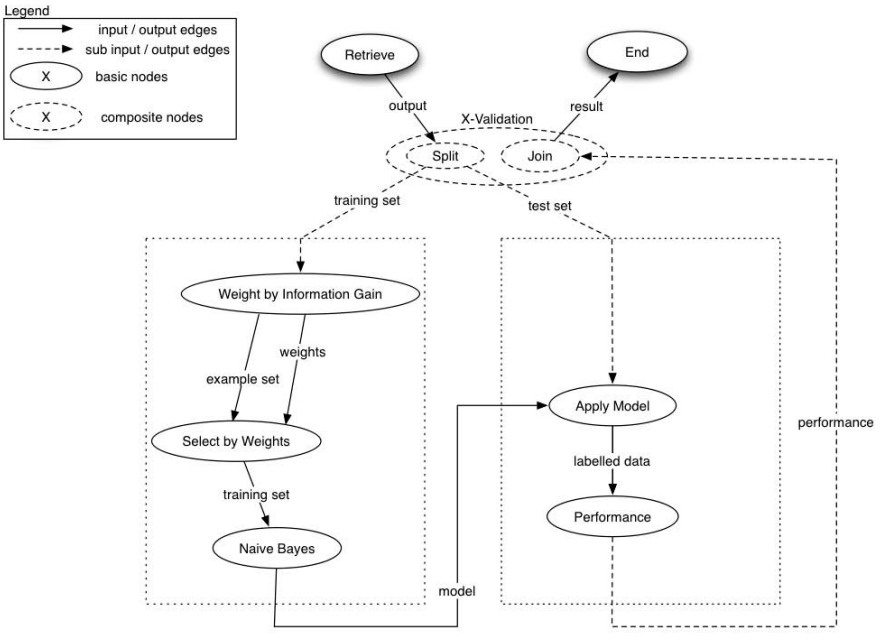


Fig. 13. A DM workflow as a hierarchical DAG

- E be the set of all available data types that can appear in a data mining workflow, e.g. the data types of the various I/O objects of some DM workflow, models, datasets, attributes, etc.
- an operator $o \in O$ be defined by its name and the data types of its inputs and outputs. 1

A hierarchical directed acyclic graph, G, that represents a data mining workflow is an ordered pair (O', E') where:

- $O' \subseteq O$ is the set of vertices or nodes that correspond to the operators used in the workflow
- $E' \subseteq E$ is the set of ordered pairs of nodes, (o_i, o_j), called directed edges, that correspond to the data types of the I/O objects, that are passed from operator o_i to operator o_j in the workflow.

E' defines the *data-flow* of the workflow and O' the *control flow*.

Workflows as parse trees. A DAG has one or more topological sorts. A *topological sort* is a permutation p of the vertices of a DAG such that an edge (o_i, o_j) indicates that o_i appears before o_j in p [65]. Thus, it is a complete ordering of the nodes of a DAG. If a topological sort has the property that all pairs of consecutive vertices in the sorted order are connected by an edge, then these edges form a directed Hamiltonian path of the DAG. In this case, the topological

order is unique. If not, then it is always possible to get the unique topological order by adding a second order such as the lexicographic order of the vertex labels. The topological sort of a DAG can be represented by a parse tree, which is a reduction of the original DAG where the edges have been fully ordered.

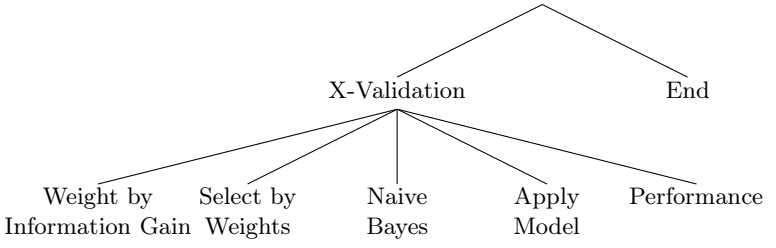


Fig. 14. The parse tree (topological sort) of the DM-workflow given in Figure 13

The parse tree of Figure 14 gives the topological sort of the DM workflow represented as a hierarchical DAG in Figure 13. As seen clearly, the parse tree is a simplification of the original graph; it represents the order of execution of the different operators and their hierarchical relation but the data-flow is lost (the edges are not labelled).

Augmenting the parse trees. Given the parse tree representation of a workflow, the next step is to augment it in view of deriving frequent patterns over generalizations of the workflow components. Generalizations will be based on concepts, relations and subsumptions modelled in DMOP. Starting from the *Operator* level, an operator $o \in O$ *implements* some algorithm $a \in A$ (Figure 4). In addition the DMOP defines a refined algorithm taxonomy, an extract of which is given in Figure 7. Note that contrary to the asserted taxonomy which is a pure tree, the inferred taxonomy can be a DAG (a concept can have multiple ancestors) [60]; consequently the subsumption order is not unique. For this reason we define a distance measure between two concepts C and D, which is related to the terminological axiom of *inclusion*, $\mathcal{C} \sqsubseteq \mathcal{D}$, as the length of the shortest path between the two concepts. This measure will be used to order the subsumptions. For the sake of clarity, we will assume a single-inheritance hierarchy in the example of the (RapidMiner) NaiveBayes operator. Given the taxonomic relations NaiveBayesNormal $\sqsubseteq$ NaiveBayesAlgorithm $\sqsubseteq$ BayesianAlgorithm $\sqsubseteq$ GenerativeAlgorithm, the reasoner infers that NaiveBayes implements someInstance of these superclasses, ordered using the distance measure described. Based on these inferences, an *augmented parse tree* is derived from an original parse tree T by inserting the ordered concept subsumptions between each node $v \in T$ and its parent node. Figure 15 shows the augmented version of the parse tree in Figure 14.

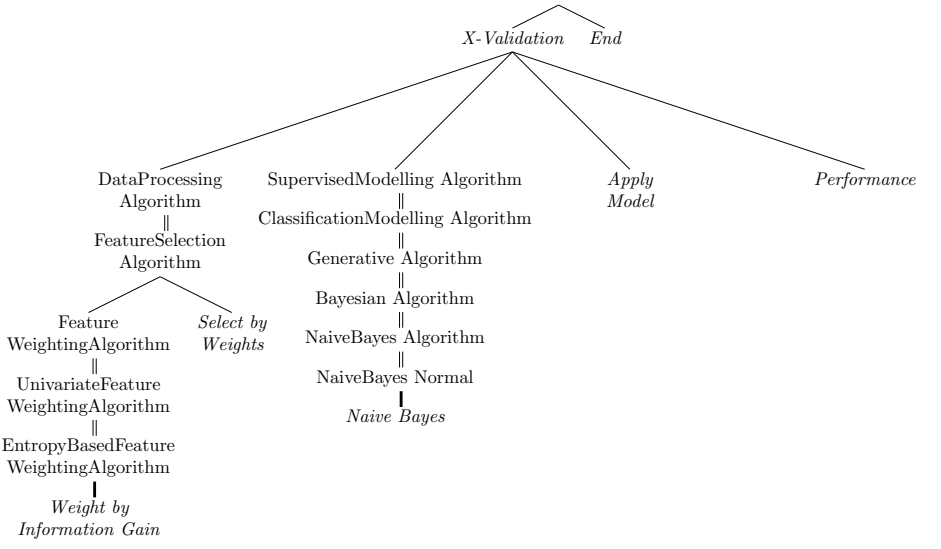


Fig. 15. An augmented parse tree. Thin edges depict task decomposition into operators (italics); a thick single line indicates that an operator implements an instance of its parent algorithm; double lines depict subsumption.

3.2 Mining Generalized Workflow Patterns

We are now ready to mine generalized patterns from DM workflows represented as (augmented) parse trees, which we now define more formally. A parse tree is a rooted k-tree [15]. A *rooted* k-tree is a set of k nodes $O' \subseteq O$ where each $o \in O'$, except one called root, has a parent denoted by $\pi(o) \in O'$. The function $l(o)$ returns the label of a node, and the operator $\prec$ denotes the order from left to right among the children of a node.

Induced subtrees. We used Zaki et al.'s TreeMiner [81] to search for frequent induced subtrees over the augmented tree representation of workflows. A tree $t' = (O_{t'}, E_{t'})$ is called an induced subtree of $t = (O_t, E_t)$, noted $t' \preceq_i t$, if and only if $O_{t'}$ preserves the direct parent-child relation of O_t. Figure 16 shows a tree T1 and two of its potential induced subtrees, T2 and T3. In the less constraining case where only an indirect ancestor-descendant relation is preserved, the subtree t is called embedded. We had no need for embedded trees: given the way augmented parse trees were built using the DMOP algorithm taxonomy, extending parent-child relationships to ancestor-descendants would only result in semantically redundant patterns with no higher support.

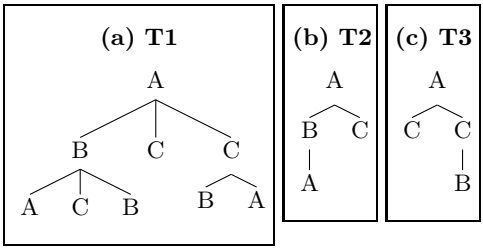


Fig. 16. A tree T1 and two of its induced subtrees ,T2 and T3

Given a database (forest) D of trees, the tree miner algorithm will produce a set $\mathcal{P}$ of induced subtrees (patterns). For a given tree $T_i \in D$ and a pattern $S \in \mathcal{P}$, if $S \preceq_i T_i$, we say that T_i *contains* S or S *occurs* in T_i. Now let $\delta_{T_i}(S)$ denote the number of occurences of the subtree $S \in \mathcal{P}$ in a tree $T_i \in D$, and let d_{T_i} be an indicator variable with $d_{T_i}(S) = 1$ if $\delta_{T_i}(S) > 0$ and $d_{T_i}(S) = 0$ if $\delta_{T_i}(S) = 0$. The *support* of the subtree S in the database D is defined as $sup(S) = \Sigma_{T_i \in D} d_{T_i}(S)$. We call the *support set* of S the set of trees $T_i \in D$ with $d_{T_i}(S) > 0$.

An example. We demonstrate frequent pattern extraction from the following workflows that do cross-validated feature selection and classification:

a) feature selection based on Information Gain, classification with Naive Bayes
b) feature selection with Relief, classification with C4.5
c) feature selection with CFS, classification with C4.5
d) wrapper feature selection with Naive Bayes, classification with Naive Bayes.

Their parse trees are given in Figure 17. Workflow a) performs univariate feature selection based on a univariate weighting algorithm. The three remaining workflows are all doing multivariate feature selection, where in b) this is done using a multivariate feature weighting algorithm, and in c) and d) using heuristic search, implemented by the *OptimizeSelection* operator, in the space of feature sets where the cost function used to guide the search is CFS and the Naive Bayes accuracy respectively.

We applied TreeMiner [81] to the augmented version of these parse trees, setting the minimum support to 2 in order to extract frequent induced subtrees. Some of the mined patterns and their support sets are shown in Figure 18.

Pattern (a) shows that in two of the four workflows, a) and b), a feature weighting algorithm is followed by the Select by Weights operator, and that this pair forms a feature selection algorithm followed by a classification modelling algorithm, nested all together in a cross-validation operator. Pattern (b) captures the fact that two workflows, b) and c), contain a multivariate feature selection followed by a decision tree algorithm, again nested inside a cross-validation.

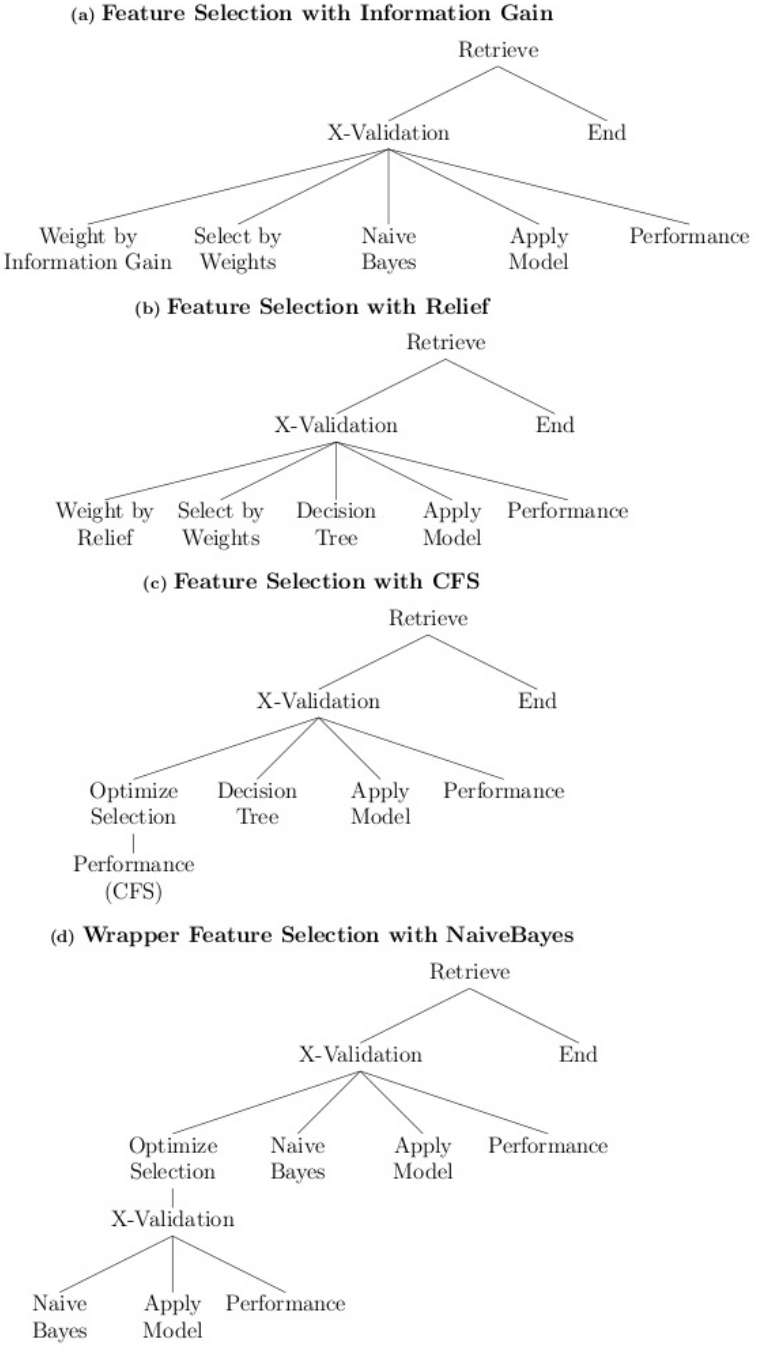


Fig. 17. Parse trees of feature selection/classification workflows

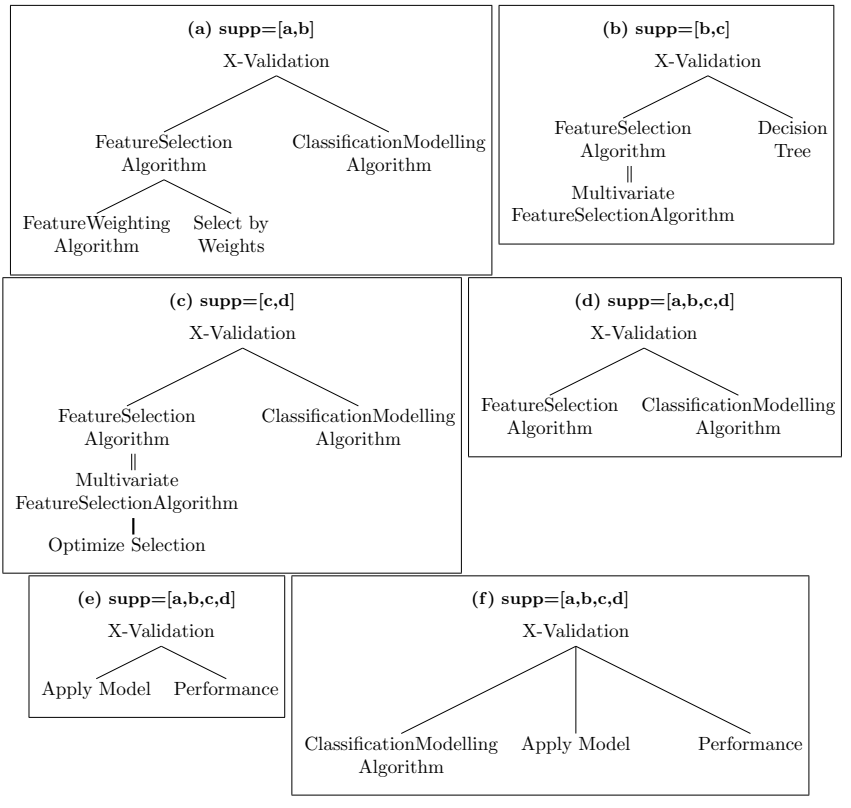


Fig. 18. Six patterns extracted from the 4 workflows of Figure 17

Pattern (c) corresponds to MultivariateFeatureSelection performed by OptimizeSelection and followed by some classification algorithm. As mentioned above, OptimizeSelection represents a heuristic search over feature sets using some search strategy and some cost function which are not specified for the moment.

Pattern (d) is a generalization of patterns (a), (b) and (c), and covers all four workflows. It simply says that a feature selection algorithm is followed by a classification modelling algorithm.

Finally, patterns (e) and (f) also cover all four workflows. Pattern (e) corresponds to the validation step where a learned model is applied to a test set using cross-validation, and some performance measure is produced. Pattern (f) is a super pattern of pattern (e) and shows that a model should first be produced by a classification modelling algorithm before it can be applied and evaluated.

4 Experiments in Workflow Meta-Mining

This section describes workflow mining experiments in which the goal is to predict the relative performance of a new workflow, whether handcrafted by the user or designed automatically, e.g. by eProPlan (page 279).

4.1 Experimental Design

As discussed in Section 1.1, standard meta-learning has been dominated by the Rice model which considers only data set characteristics to predict the performance of algorithms or algorithm families. We proposed an alternative model which takes into account both data and algorithm characteristics. In this section, we apply the revised Rice model to workflow mining: in the meta-mining view of workflows as compositions of (implemented) algorithms, workflow selection or ranking is grounded on both data and workflow characteristics.

Gathering the meta-mining data. To define meta-learning problems and gather the necessary meta-data, we need to collect results from a sizeable number domain-level data mining experiments. We gathered 65 datasets concerning microarray experiments on different types of cancer. Table 1 gives the name (prefixed by the cancer type) and the number of examples, features, and classes for each dataset. As is typical of gene profiling data, all are high-dimensional small samples, i.e. the number of features is several orders of magnitude higher than the number of examples.

The choice of a clearly circumscribed application domain for our meta-learning experiments has a clear rationale. Previous work on meta-learning typically relied on base-level experiments using UCI datasets [26] from the most diverse domains. As a result, the meta-learner groped for regularities in the intractable immensity and sparsity of the space of all possible datasets—from classical toy problems (e.g. Lenses, Tic-tac-toe) to more recent biomedical datasets (e.g. Dorothea, Arcene), where the number of dimensions is one or several orders of magnitude greater than the number of instances. Initially motivated by user rather than meta-learner considerations, so-called third-generation data mining systems [58] promoted the idea of vertical systems which focus on a specific application domain and problem, thus ensuring a more dense and coherent search space as well as the possibility of bringing domain-specific knowledge to bear in the knowledge discovery process. In this spirit, we selected gene expression-based cancer diagnosis as our problem domain, with the explicit proviso that all conclusions drawn from these experiments will apply only to datasets that stem from the same application area or at least share their essential characteristics.

We applied different data mining workflows to these datasets and estimated their performance using ten fold cross-validation. All the workflows were combinations of feature selection and classification algorithms. We used the following feature selection algorithms: Information Gain (IG), Chi-square (CHI), ReliefF (RF), and recursive feature elimination with SVM (SVMRFE); we fixed the number of selected features to ten. For classification we used the following algorithms: one-nearest-neighbor (1NN), decision tree algorithms J48 and CART, Naive Bayes

(NB), logistic regression algorithm (LR), and SVMs with linear (SVM-L) and Gaussian (SVM-RBF) kernels. For J48 the C (pruning confidence) and M (minimum number of instances per leaf) parameters were set to 0.25 and 2 respectively; for CART the M and N (number of folds for the minimal cost-complexity pruning) parameters were set to 2 and 5 respectively. The C parameter was set to 1 for both SVM-L and SVM-R, and SVM-R's γ parameter was set to 0.1. We used the implementations of these algorithms in the RapidMiner data mining suite. All the possible combinations of the four feature selection algoriths with the seven classication algorithms gave 28 different learning workflows, each applied to the 65 datasets, for a total of 1820 data mining experiments.

Table 1. The 65 microarray datasets used in the meta-mining experiments. N: number of examples, D: number of features, C: number of classes

Dataset	N	D	C	Dataset	N	D	C
adrenal_dahia	76	22283	2	leukemia_haslinger	100	12600	2
bladder_blaveri	40	5331	2	leukemia_wei	29	21481	2
bladder_dyrskjot	40	4409	3	leukemia_yagi	53	7241	2
bladder_sanchez-carbayo	157	22283	3	liver_chen	156	2621	2
breast_desmedt	198	22283	2	liver_iizuka	60	7129	2
breast_farmer	49	22215	3	liver_ye	87	8121	3
breast_gruvberger	58	3389	2	lung_barret	54	22283	2
breast_kreike	59	17291	2	lung_beer	86	7129	3
breast_ma_2	60	22575	2	lung_bhattacharjee_2	197	12600	4
breast_minn	68	22283	2	lung_bild	111	54675	2
breast_perou	65	7261	4	lung_wigle	39	1971	2
breast_sharma	60	1368	2	lymphoma_alizadeh	99	8580	2
breast_sotiriou	167	22283	3	lymphoma_booman	36	14362	2
breast_veer	97	24481	2	lymphoma_rosenwald	240	7388	3
breast_wang	286	22283	2	lymphoma_shipp	77	7129	2
breast_west	49	7129	2	medulloblastoma_macdonald	23	1098	2
cervical_wong	33	10692	2	melanoma_talantov	70	22283	3
cns_pomeroy_2	60	7129	2	mixed_chowdary	104	22281	2
colon_alon	62	2000	2	mixed_ramaswamy	76	15539	2
colon_laiho	37	22283	2	myeloma_tian	173	12625	2
colon_lin_1	55	16041	2	oral_odonnell	27	22283	2
colon_watanabe	84	54675	2	ovarian_gilks	23	36534	2
gastric_hippo	30	7127	2	ovarian_jazaeri_3	61	6445	2
glioma_freije	85	22645	2	ovarian_li_and_campbell	54	1536	2
glioma_nutt	50	12625	2	ovarian_schwartz	113	7069	5
glioma_phillips	100	22645	2	pancreas_ishikawa	49	22645	2
glioma_rickman	40	7069	2	prostate_singh	102	12600	2
head_neck_chung	47	9894	2	prostate_tomlins	83	10434	4
headneck_pyeon_2	42	54675	2	prostate_true_2	31	12783	2
leukemia_armstrong	72	12582	3	renal_williams	27	17776	2
leukemia_bullinger_2	116	7983	2	sarcoma_detwiller	54	22283	2
leukemia_golub	72	7129	2	srbct_khan	88	2308	4
leukemia_gutierrez	56	22283	4				

Predicting the performance of a candidate workflow was cast as a classification problem: given a dataset d_j, determine whether workflow wf_i will be among the top performing workflows (class *best*) or not (class *rest*). We assigned these class labels as follows. For each dataset we did a pairwise comparison of the estimated performance of the 28 workflows applied to it using a McNemar's test of statistical significance. For each workflow pair, a score of 1 was assigned to the workflow—if any—which performed significantly better than the other, which scored 0; otherwise both were assigned 0.5. The final performance rank of a workflow for a given dataset was determined by the sum of points it scored on these pairwise comparisons for that dataset. Clearly in the case of 28 workflows the maximum possible score is 27 when a workflow is significantly better than all the other workflows. If there are no significant differences then each workflow gets a score of 13.5. The class label of a workflow for a given dataset was determined based on its score; workflows whose scores were within 1.5 standard deviations of the best performance measure for that dataset were labelled *best* and the remaining workflows *rest*. Under this choice 45% of the experiments, i.e. (d_j, wf_i) pairs, were assigned the label *best* and the remaining 55% the *rest* label.

Representing the meta-data. As explained earlier in this section, we used a combination of dataset and workflow characteristics to describe the meta-learning examples.

Data descriptors. We took 6 dataset characteristics from the StatLog and METAL projects: class entropy, average feature entropy, average mutual information, noise-signal ratio, outlier measure of continuous features, and proportion of continuous features with outliers. Because our base-level datasets contained only continuous predictive features, average feature entropy and average mutual information were computed via a binary split on the range of continuous feature values, as is done in C4.5 [59]. Detailed descriptions of these data characteristics are given in [50,41].

In addition, we used 12 geometric data complexity measures from [38]. These can be grouped into three categories: (1) measures of overlaps in feature values from different classes (maximum Fisher's discriminant ratio, volume of overlap region, maximum feature efficiency); (2) measures of class separability (fraction of instances on class boundary, ratio of average intra/inter-class distance, and landmarker-type measures like error rates of 1-NN and a linear classifier on the dataset; (3) measures of geometry, topology, and density of manifolds (non-linearity of linear classifier, nonlinearity of 1NN classifier, fraction of points with retained adherence subsets, and average number of points per dimension). The definitions of these measures, their rationale and formulas, are given in [38].

Workflow descriptors. Workflow descriptors were constructed in several steps following the pattern discovery method described in Section 3:

1. We built parse trees (Section 3.1) from the 28 workflows and augmented them using concept subsumptions from the DMOP ontology (Section 3.1); we thus obtained 456 augmented parse trees such as that shown in Figure 15.

2. We applied the TreeMiner algorithm with a minimum support of 3% to the augmented parse trees, thereby extracting 3843 frequent patterns defined as induced subtrees (Section 3.2).
3. We ordered the extracted patterns in order of decreasing generality, then pruned this initial pattern set to retain only closed patterns, i.e. patterns that are maximal with respect to the subsumption ordering of an equivalence class of patterns having the same support set [4]. The final set contained 1051 closed workflow patterns similar to those in Figure 18. In a nutshell, a workflow pattern is simply a fragment of an augmented (workflow) parse tree that has a support above a predefined threshold.
4. Finally, we converted each workflow pattern into a binary feature whose value equals 1 if the given workflow contains the pattern and 0 otherwise; it is these boolean features that we call workflow descriptors. Thus each workflow was represented as a vector of 1051 boolean workflow descriptors. Essentially, what we did was propositionalize the graph structure of the DM workflows.

4.2 Experimental Results

We defined two meta-mining scenarios. Scenario A relies mainly on the dataset characteristics to predict performance, while scenario B considers both dataset and workflow characteristics.

Meta-mining scenario A. In this scenario we create one meta-mining problem per workflow, producing 28 different problems. We denote by WF this set of meta-mining problems and by WF_i the meta-mining problem associated with workflow wf_i. For each meta-mining problem WF_i, the goal is to build a model that will predict the performance of workflow wf_i on some dataset. Under this formulation each meta-mining problem consists of $|D| = 65$ learning instances, one for each of the datasets in Table 1; the features of these instances are the dataset descriptors. The class label for each dataset d_j is either *best* or *rest*, based on the score of workflow wf_i on dataset d_j as described on page 302.

An issue that arises is how to measure the error for a specific dataset, which is associated with 28 different predictions. One option is to count an error whenever the *set* of workflows that are predicted as *best* is not a subset of the truly best workflow set. This error definition is more appropriate for the task at hand, where the goal is to recommend workflows that are expected to perform best; it is less important to miss some of them (false negatives) than to recommend workflows that will actually underperform (false positives). Here we adopted the simple approach of counting an error whenever the prediction does not match the class label, regardless of the direction of the error. With this method the overall error averaged over the 65 datasets is equal to the average error over the 28 different meta-mining problems WF_i. We denote this error estimate by

$$A_{d_{algo}} = \frac{1}{|WF|} \sum_{i=1}^{|WF|} (f(x) \neq y) = \frac{1}{|D|} \sum_{i=1}^{|D|} (f(x) \neq y),$$

where $f(x)$ denotes the predicted class, y the actual class, and $algo$ the learning algorithm that was used to construct the meta-mining models. We use McNemar's test to estimate the statistical difference between the error of the meta-learner and that of the default rule, which simply predicts the majority class for each meta-mining problem WF_i. The average error of the default classifier is denoted by

$$A_{d_{def}} = \frac{1}{|WF|} \sum_{i=1}^{|WF|} (c_{maj} \neq y),$$

where c_{maj} is the majority class for problem $WF_i \in WF$ and y is the actual class or class label.

To generate the meta-mining models we used J48 with the following parameter settings: C=0.25 and M=2. Table 2 shows the error rates of the default rule ($A_{d_{def}}$) and J48 ($A_{d_{J48}}$), averaged over the 28 different meta-mining problems, which is equivalent to the error averaged over the different datasets. The average error rate of the meta-models using dataset characteristics was lower than that of the default rule by around 5%, an improvement that was shown to be statististically significant by McNemar's test.

Table 2. Average estimated errors for the 28 WF_i meta-mining problems in meta-mining scenario A. A + sign indicates that $A_{d_{J48}}$ was significantly better than $A_{d_{def}}$, an = that there was no significant difference and a - that it was significantly worse.

$A_{d_{def}}$	$A_{d_{J48}}$
45.38	40.44 (+)

Meta-mining scenario B. The main limitation of meta-mining scenario A is that it is not possible to generalize over the learning workflows. There is no way we can predict the performance of a DM workflow wf_i unless we have meta-mined a model based on training meta-data gathered through systematic experimentation with wf_i itself. To address this limitation we introduce the second meta-mining scenario which exploits *both dataset and workflow descriptions*, and provides the means to generalize not only over datasets but also over workflows.

In scenario B, we have a single meta-mining problem in which each instance corresponds to a base-level data mining experiment where some workflow wf_i is applied to a dataset d_j; the class label is either *best* or *rest,* determined with the same rule as described above. We thus have $65 \times 28 = 1820$ meta-mining instances. The description of an instance combines both dataset and workflow meta-features. This representation makes it possible to predict the performance of workflows which have not been encountered in previous DM experiments, provided they are represented with the set of workflow descriptors used in the predictive meta-model. The quality of performance predictions for such workflows clearly depends on how similar they are to the workflows based on which the meta-model was trained.

The instances of this meta-mining dataset are not independent, since they overlap both with respect to the dataset descriptions and the workflow descriptions. Despite this violation of the learning instance independence assumption, we also applied standard classification algorithms as a first approach. However, we handled performance evaluation with precaution. We first evaluated predictive performance using leave-one-dataset-out, i.e., we removed all meta-instances associated with a given dataset d_i and placed them in the test set. We built a predictive model from the remaining instances and applied it to the test instances. In this way we avoided the risk of information leakage incurred in standard leave-out-out or cross-validation, where both training and test sets are likely to contain instances (experiments) concerning the same dataset. We will denote the predictive error estimated in this manner by $B_{d_{algo}}$, where $algo$ is the classification algorithm that was used to construct the meta-mining models. The total number of models built was equal to the number of datasets. For each dataset d_j, the meta-level training set contained $64 \times 28 = 1792$ instances and the test set 28, corresponding to the application of the 28 workflows to d_j.

In addition, we investigated the possibility of predicting the performance of workflows that have not been included in the training set of the meta-mining model. The evaluation was done as follows: in addition to leave-one-dataset-out, we also performed leave-one-workflow-out, removing from the training set all instances of a given workflow. In other words, for each training-test set pair, we removed all meta-mining instances associated with a specific dataset d_j as well as all instances associated with a specific workflow wf_i. We thus did $65 \times 28 = 1820$ iterations of the train-test separation, where the training set consisted of $64 \times 27 = 1728$ instances and the test set of the single meta-mining instance$(d_i, wf_j, label)$. We denote the error thus estimated by $B_{d,wf_{algo}}$.

Table 3. $B_{d_{J48}}$ and $B_{d,wf_{J48}}$ estimated errors, meta-mining scenario B. A + sign indicates that $B_{d|d,wf_{algo}}$ was significantly better than $A_{d_{def}}$, an = that there was no significant difference and a - that it was significantly worse. Column 2 shows that the meta-miner obtains significantly better results than the default by using both dataset and workflow descriptors. Column 3 gives the results in a more stringent scenario involving workflows never encountered in previous domain-level experiments.

$A_{d_{def}}$	$B_{d_{J48}}$	$B_{d,wf_{J48}}$
45.38	38.24 (+)	42.25 (=)

Table 3 gives the estimated errors for meta-mining scenario B, in which the meta-models were also built using J48 with the same parameters as in scenario A, but this time using both dataset and workflow characteristics. McNemar's test was also used to compare their performance against the default rule. Column 2 shows the error rate using leave-one-dataset-out error estimation ($B_{d_{J48}}$), which is significantly lower than that of the default rule, but more importantly, also lower than $A_{d_{j48}}$ (Table 2), the error rate of the meta-model built using dataset characteristics alone. This provides evidence of the discriminatory power

of the frequent patterns discovered in the DM workflows and used to build the meta-models.

Column 3 shows the error rate of the meta-model built using the combined leave-one-out-dataset/leave-one-workflow-out error estimation procedure ($B_{d,wf_{J48}}$), which was meant to test the ability of the meta-model to predict the performance of completely new workflows. The estimated error rate is again lower than the baseline error, though not significantly. However, it demonstrates the point that for a new dataset, our approach can predict the performance of workflows never yet encountered in previous data mining experiments. As a matter of fact, these workflows can even contain operators that implement algorithms never seen in previous experiments, provided these algorithms have been described in DMOP.

4.3 Discussion

To gain a bit of insight into the kind of meta-model built by J48 using dataset and workflow meta-features, we reran mining scenario B on the full dataset to derive the decision tree to be deployed on new (dataset, workflow) examples. The result was a complex decision tree with 56 internal nodes (meta-feature tests), but a close analysis of a few top levels proved instructive.

The top two nodes test the dataset characteristic intraInter, or intra-interclass nearest neighbor (NN) distance [38]. This is a measure of class separability, computed as the ratio of intra (average distance between each data point and its nearest-hit, i.e. its NN from the same class) to inter (average distance between each data point and its nearest-miss, i.e. its NN from a different class):

$$intraInter = \left(\frac{1}{N}\sum_{i=1}^{N} d(x_i, nearestHit(x_i)\right) / \left(\frac{1}{N}\sum_{i=1}^{N} d(x_i, nearestMiss(x_i)\right).$$

As shown in Fig. 19, the first two tests actually split the intraInter range of values into 3 intervals. At one extreme, $intraInter > 0.99$ indicates a difficult problem where data points from different classes are almost as close as points from the same class; in this case, the meta-model predicts the class REST. At the other extreme, the distance of a data point to members of a different class is almost twice its distance to members of its class ($intraInter \leq 0.54$,); in such cases where classes are neatly separated, the prediction is BEST. Between these two extremes, other tests are needed; the next meta-feature tested is $ptsPerDim = N/D$, where N is the number of data points and D the dimensionality or number of features. The threshold of 0.0008 discriminates between extremely high-dimensional datasets which contain less than 0.0008 instances per feature, or equivalently, more than 1300 features for 1 instance (right branch) and datasets with lower D:N ratio (left branch). We omit the right branch, which grows to a depth of more than 20 levels; in the left branch, by contrast, tests on 2 workflow descriptors and 1 dataset descriptor suffice to classify the remaining instances. The workflow descriptor statBasedFeatureWeightingAlgorithm designates a class of feature weighting algorithms that weight individual features by computing statistics such as

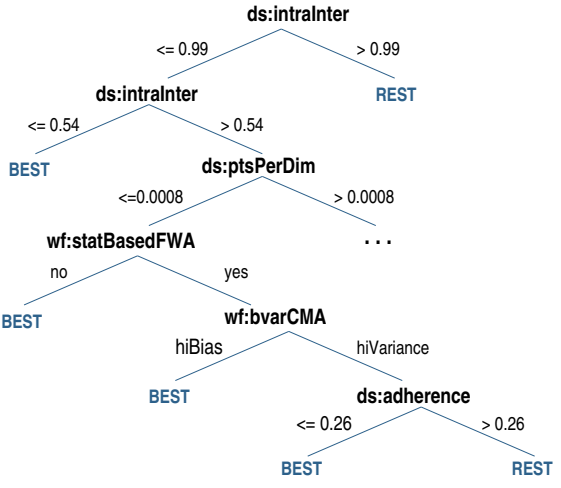


Fig. 19. Top 6 levels of the meta-mined J48 workflow performance predictor based on dataset and workflow characteristics

χ^2, F-Ratio, or Pearson's correlation coefficient. Workflows that do not use such weighting algorithms (e.g., multivariate algorithms, or univariate methods that use entropy-based weights) are classified as BEST. Among workflows that rely on such statistics to weight features, only those that also use high-bias classification modelling algorithms (e.g. linear discriminants, Naive Bayes) will be predicted to have BEST performance. High-variance algorithms will be classified as BEST only if they are applied to datasets with *adherence* < 0.26. This feature denotes the fraction of data points with maximal adherence subset retained [38]. Intuitively, an adherence subset can be imagined as a ball that is fit around each data point and allowed to grow, covering other data points and merging with other balls, until it hits a data point from a different class. With complex boundaries or highly interleaved classes, the fraction of points with retained (i.e. not merged) adherence subsets will be large. In the learned meta-decision tree, adherence should not be greater than 0.26 for high-variance classification learners to perform BEST.

To summarize, the meta-decision tree described above naturally blends data and workflow characteristics to predict the performance of a candidate workflow on a given dataset. In the vicinity of the root, J48's built-in feature selection mechanism picked up 3 descriptors of data complexity (class separability, dimensionality, and boundary complexity) and 2 workflow descriptors (use of univariate statistics-based feature scores, bias-variance profile of learning algorithm/operator used). Although data descriptors outnumber workflow descriptors in the subtree illustrated in Figure 19, the distribution is remarkably balanced over the whole tree, where 28 of the 56 internal nodes test workflow features. However, most of the workflow features used correspond to simple patterns that express a constraint on a single data mining operator. Only two nodes test

a sequence comprising a feature weighting/selection operator and a classification operator. We expect more complex patterns to appear when we feed the meta-learner with workflows from data mining experiments with multi-step data processing. Finally, as mentioned above, the right subtree below ptsPerDim (replaced by "..." in the figure), which corresponds to datasets with more than 1300 features per data point, is considerably more complex; worth noting, however, is the recurrence of the workflow pattern that contains "high-dimensionality tolerant classification modelling algorithm" in branches that lead to a BEST leaf.

5 Conclusion

In this chapter, we proposed a semantic meta-mining approach that contrasts with standard meta-learning in several respects. First, the traditional meta-learning focus on a single task or operator has been replaced by a broader perspective on the full knowledge discovery process. Next, we introduced a revised Rice model that grounds algorithm selection on both data and algorithm characteristics. We operationalized this revised model while mining workflows viewed as compositions of (implemented) algorithms, and performed workflow performance prediction based on both dataset and workflow characteristics. In two distinct meta-mining scenarios, models built using data and workflow characteristics outperformed those based on data characteristics alone, and meta-mined workflow patterns proved discriminatory even for algorithms and workflows not encountered in previous experiments. These experimental results show that the data mining semantics and expertise derived from the DMOP ontology imparts new generalization power to workflow meta-mining.

Though promising, these results can definitely be improved. Performance prediction for DM workflows is still in its infancy, and we have done no more than provide a proof of concept. We certainly need more base-level experiments and more workflows in order to improve the accuracy of learned meta-models. We also need to investigate more thoroughly the different dataset characteristics that have been used in previous meta-learning efforts. Above all, we need more refined strategies for exploring the the joint space of dataset characteristics and workflow characteristics. A simple approach could be to build a model in two stages: first zoom in on the datasets and explore clusters or neighborhoods of datasets with similar characteristics; then within each neighborhood, identify the workflow characteristics that entail good predictive performance. Essentially, what we are trying to solve is a matching problem: the goal is to find the appropriate association of workflow and dataset characteristics, where appropriateness is defined in terms of predictive performance. One way to address this problem is to use collaborative filtering approaches that are also able to account for the properties of the matched objects.

Acknowledgements. This work is partially supported by the European Commission through FP7-ICT-2007-4.4 (Grant No. 231519) project e-LICO: An e-Laboratory for Interdisciplinary Collaborative research in Data Mining and Data-Intensive Sciences.

We thank all our partners and colleagues who have contributed to the development of the DMOP ontology: Simon Fischer, Dragan Gamberger, Simon Jupp, Agnieszka Lawrynowicz, Babak Mougouie, Raul Palma, Robert Stevens, Jun Wang.

References

1. Aha, D.W.: Lazy learning (editorial). Artificial Intelligence Review 11, 7–10 (1997)
2. Ali, S., Smith-Miles, K.: A meta-learning approach to automatic kernel selection for support vector machines. Neurocomputing 70(1-3), 173–186 (2006)
3. Anderson, M.L., Oates, T.: A review of recent research in metareasoning and metalearning. AI Magazine 28(1), 7–16 (2007)
4. Arimura, H.: Efficient algorithms for mining frequent and closed patterns from semi-structured data. In: Washio, T., Suzuki, E., Ting, K.M., Inokuchi, A. (eds.) PAKDD 2008. LNCS (LNAI), vol. 5012, pp. 2–13. Springer, Heidelberg (2008)
5. Bartlett, P.: For valid generalization, the size of the weights is more important than the size of the network. In: Advances in Nueral Information Processing Systems, NIPS-1997 (1997)
6. Basu, M., Ho, T.K. (eds.): Data Complexity in Pattern Recognition. Springer, Heidelberg (2006)
7. Bensusan, H., Giraud-Carrier, C.: Discovering task neighbourhoods through landmark learning performances. In: Proceedings of the Fourth European Conference on Principles and Practice of Knowledge Discovery in Databases, pp. 325–330 (2000)
8. Bensusan, H., Giraud-Carrier, C., Kennedy, C.: A higher-order approach to metalearning. In: Proceedings of the ECML 2000 workshop on Meta-Learning: Building Automatic Advice Strategies for Model Selection and Method Combination, June 2000, pp. 109–117 (2000)
9. Bernstein, A., Provost, F., Hill, S.: Toward intelligent assistance for a data mining process: An ontology-based approach for cost-sensitive classification. IEEE Transactions on Knowledge and Data Engineering 17(4), 503–518 (2005)
10. Bishop, C.: Pattern Recognition and Machine Learning. Springer, Heidelberg (2006)
11. Blockeel, H., Vanschoren, J.: Experiment Databases: Towards an Improved Experimental Methodology in Machine Learning. In: Kok, J.N., Koronacki, J., Lopez de Mantaras, R., Matwin, S., Mladenič, D., Skowron, A. (eds.) PKDD 2007. LNCS (LNAI), vol. 4702, pp. 6–17. Springer, Heidelberg (2007)
12. Brazdil, P., Gama, J., Henery, B.: Characterizing the applicability of classification algorithms using meta-level learning. In: Bergadano, F., De Raedt, L. (eds.) ECML 1994. LNCS, vol. 784, pp. 83–102. Springer, Heidelberg (1994)
13. Brazdil, P., Giraud-Carrier, C., Soares, C., Vilalta, R. (eds.): Metalearning: Applications to Data Mining. Springer, Heidelberg (2009)
14. Brezany, P., Janciak, I., Min Tjoa, A.: Ontology-based construction of grid data mining workflows. In: Nigro, H.O., Gonzalez Cisaro, S.E., Xodo, D.H. (eds.) Data Mining with Ontologies: Implementations, Findings and Frameworks, IGI Global (2008)
15. Bringmann, B.: Matching in frequent tree discovery. In: Proc.4th IEEE International Conference on Data Mining (ICDM 2004), pp. 335–338 (2004)

16. Cacoveanu, S., Vidrighin, C., Potolea, R.: Evolutional meta-learning framework for automatic classifier selection. In: Proceedings of the IEEE 5th International Conference on Intelligent Computer Communication and Processing (ICCP 2009), pp. 27–30 (2009)
17. Cannataro, M., Comito, C.: A data mining ontology for grid programming. In: Proc. 1st Int. Workshop on Semantics in Peer-to-Peer and Grid Computing, in conjunction with WWW 2003, pp. 113–134 (2003)
18. Chapman, P., Clinton, J., Khabaza, T., Reinartz, T., Wirth, R.: The CRISP-DM process model. Technical report, CRISP-DM consortium (1999), http://www.crisp-dm.org
19. Cherkassky, V.: Model complexity control and statistical learning theory. Natural Computing 1, 109–133 (2002)
20. Diamantini, C., Potena, D., Storti, E.: Supporting users in KDD process design: A semantic similarity matching approach. In: Proc. 3rd Planning to Learn Workshop (held in conjunction with ECAI 2010), Lisbon, pp. 27–34 (2010)
21. DomingosA, P.: unified bias-variance decomposition for zero-one and squared loss. In: Proceedings of the Seventeenth National Conference on Artificial Intelligence, pp. 564–569 (2000)
22. Duch, W., Grudzinski, K.: Meta-learning: Searching in the model space. In: Proc. of the Int. Conf. on Neural Information Processing (ICONIP), Shanghai 2001, pp. 235–240 (2001)
23. Duch, W., Grudziński, K.: Meta-learning via search combined with parameter optimization. In: Advances in Soft Computing, pp. 13–22. Springer, Heidelberg (2002)
24. Džeroski, S.: Towards a general framework for data mining. In: Džeroski, S., Struyf, J. (eds.) KDID 2006. LNCS, vol. 4747, pp. 259–300. Springer, Heidelberg (2007)
25. Fayyad, U., Piatetsky-Shapiro, G., Smyth, P.: From data mining to knowledge discovery: An overview. In: Advances in Knowledge Discovery and Data Mining, pp. 1–34. MIT Press, Cambridge (1996)
26. Frank, A., Asuncion, A.: UCI machine learning repository (2010)
27. Fürnkranz, J., Petrak, J.: An evaluation of landmarking variants. In: Proceedings of the ECML Workshop on Integrating Aspects of Data Mining, Decision Support and Meta-learning, pp. 57–68 (2001)
28. Geman, S., Bienenstock, E., Doursat, R.: Neural networks and the bias/variance dilemma. Neural Computation 4, 1–58 (1992)
29. Giraud-Carrier, C., Vilalta, R., Brazdil, P.: Introduction to the special issue on meta-learning. Machine Learning 54, 187–193 (2004)
30. Gordon, D., DesJardins, M.: Evaluation and selection of biases in machine learning. Machine Learning 20, 5–22 (1995)
31. Grąbczewski, K., Jankowski, N.: Versatile and efficient meta-learning architecture: knowledge representation and management in computational intelligence. In: IEEE Symposium on Computational Intelligence and Data Mining, pp. 51–58 (2007)
32. Data Mining Group. Predictive Model Markup Language (PMML), http://www.dmg.org/
33. Guyon, I., Gunn, S., Nikravesh, M., Zadeh, L.A. (eds.): Feature Extraction: Foundations and Applications. Springer, Heidelberg (2006)
34. Guyon, I., Weston, J., Barnhill, S., Vapnik, V.: Gene selection for cancer classification using support vector machines. Machine Learning 46, 389–422 (2002)
35. Hall, M.: Correlation-based Feature Selection in Machine Learning. PhD thesis, University of Waikato (1999)

36. Hilario, M., Kalousis, A.: Fusion of meta-knowledge and meta-data for case-based model selection. In: Siebes, A., De Raedt, L. (eds.) PKDD 2001. LNCS (LNAI), vol. 2168, pp. 180–191. Springer, Heidelberg (2001)
37. Hilario, M., Kalousis, A., Nguyen, P., Woznica, A.: A data mining ontology for algorithm selection and meta-mining. In: Workshop on Third-Generation Data Mining: Towards Service-Oriented Knowledge Discovery, SoKD 2009 (2009)
38. Ho, T.K., Basu, M.: Measures of geometrical complexity in classification problems. In: Data Complexity in Pattern Recognition, ch. 1, pp. 3–23. Springer, Heidelberg (2006)
39. Hotho, A., Maedche, A., Staab, S., Studer, R.: Seal-II - the soft spot between richly structured and unstructured knowledge. Journal of Universal Computer Science 7(7), 566–590 (2001)
40. Jankowski, N., Grąbczewski, K.: Building meta-learning algorithms basing on search controlled by machine complexity. In: IEEE World Congress on Computational Intelligence, pp. 3600–3607 (2008)
41. Kalousis, A.: Algorithm Selection via Meta-Learning. PhD thesis, University of Geneva (2002)
42. Kalousis, A., Gama, J., Hilario, M.: On data and algorithms: understanding inductive performance. Machine Learning 54, 275–312 (2004)
43. Kalousis, A., Hilario, M.: Representational issues in meta-learning. In: Proc. of the 20th International Conference on Machine Learning, Washington, DC, Morgan Kaufmann, San Francisco (2003)
44. Kietz, J.-U., Serban, F., Bernstein, A., Fischer, S.: Data mining workflow templates for intelligent discovery assistance and auto-experimentation. In: Proc. 3rd Workshop on Third-Generation Data Mining: Towards Service-Oriented Knowledge Discovery (SoKD 2010), pp. 1–12 (2010)
45. Kira, K., Rendell, L.: The feature selection problem: traditional methods and a new algorithm. In: Proc. Nat. Conf. on Artificial Intelligence (AAAI 1992), pp. 129–134 (1992)
46. Köpf, C., Keller, J.: Meta-analysis: from data characterization for meta-learning to meta-regression. In: PKDD 2000 Workshop on Data Mining, Decision Support, Meta-Learning and ILP (2000)
47. Leite, R., Brazdil, P.: Predicting a relative performance of classifiers from samples. In: Proc. International Conference on Machine Learning (2005)
48. Ler, D., Koprinska, I., Chawla, S.: Utilising regression-based landmarkers within a meta-learning framework for algorithm selection. In: Proc. ICML 2005 Workshop on Meta-Learning, pp. 44–51 (2005)
49. Liu, H., Setiono, R.: A probabilistic approach to feature selection—a filter solution. In: Proc. 13th International Conference on Machine Learning (ICML 1996), Bari, Italy, pp. 319–327 (1996)
50. Michie, D., Spiegelhalter, D.J., Taylor, C.C. (eds.): Machine learning, neural and statistical classification. Ellis-Horwood (1994)
51. Mitchell, T.M.: The need for biases in learning generalizations. Technical report, Rutgers University, New Brunswick, NJ (1980)
52. Morik, K., Scholz, M.: The MiningMart Approach to Knowledge Discovery in Databases. In: Intelligent Technologies for Information Analysis, Springer, Heidelberg (2004)
53. Panov, P., Dzeroski, S., Soldatova, L.: Ontodm: An ontology of data mining. In: Proceedings of the 2008 IEEE International Conference on Data Mining Workshops, pp. 752–760 (2008)

54. Pearl, J.: Heuristics: intelligent search strategies for computer problem solving. Addison-Wesley, Reading (1984)
55. Peng, Y., Flach, P., Brazdil, P., Soares, C.: Decision tree-based data characterization for meta-learning. In: 2nd International Workshop on Integration and Collaboration Aspects of Data Mining, Decision Support and Meta-Learning (2002)
56. Peng, Y., Flach, P., Soares, C., Brazdil, P.: Improved dataset characterisation for meta-learning. In: Discovery Science, pp. 141–152 (2002)
57. Pfahringer, B., Bensusan, H., Giraud-Carrier, C.: Meta-learning by landmarking various learning algorithms. In: Proc. Seventeenth International Conference on Machine Learning, ICML 2000, pp. 743–750. Morgan Kaufmann, San Francisco (2000)
58. Piatetskey-Shapiro, G.: Data mining and knowledge discovery: The third generation. In: Raś, Z.W., Skowron, A. (eds.) ISMIS 1997. LNCS, vol. 1325, Springer, Heidelberg (1997)
59. Quinlan, J.R.: Improved use of continuous attributes in c4.5. Journal of Artificial Intelligence Research 4, 77–90 (1996)
60. Rector, A.: Modularisation of domain ontologies implemented in description logics and related formalisms including OWL. In: Proc. International Conference on Knowledge Capture, K-CAP 2003 (2003)
61. Rendell, L., Seshu, R., Tcheng, D.: Layered concept-learning and dynamically variable bias management. In: Proc. of the 10th International Joint Conference on Artificial Intelligence, pp. 308–314 (1987)
62. Rice, J.: The algorithm selection problem. Advances in Computing 15, 65–118 (1976)
63. Schaffer, C.: A conservation law for generalization performance. In: Proc. of the 11th International Conference on Machine Learning, pp. 259–265 (1994)
64. Sikonja, M.R., Kononenko, I.: Theoretical and empirical analysis of ReliefF and RReliefF. Machine Learning 53, 23–69 (2003)
65. Skiena, S.: Implementing discrete mathematics: combinatorics and graph theory with Mathematica. Addison-Wesly Longman Publishing Co., Inc., Boston (1991)
66. Smith-Miles, K.A.: Cross-disciplinary perspectives on meta-learning for algorithm selection. ACM Computing Surveys 41(1) (2008)
67. Soares, C., Brazdil, P.B.: Zoomed ranking: Selection of classification algorithms based on relevant performance information. In: Zighed, D.A., Komorowski, J., Żytkow, J.M. (eds.) PKDD 2000. LNCS (LNAI), vol. 1910, pp. 126–135. Springer, Heidelberg (2000)
68. Soares, C., Brazdil, P., Kuba, P.: A meta-learning method to select the kernel width in support vector regression. Machine Learning 54(3), 195–209 (2004)
69. Souto, M., Prudêncio, R., Soares, R., Araújo, D., Costa, I., Ludermir, T., Schliep, A.: Ranking and selecting clustering algorithms using a meta-learning approach. In: International Joint Conference on Neural Networks (2008)
70. Srikant, R., Agrawal, R.: Mining sequential patterns: generalizations and performance improvements. In: Proc.5th International Conference on Extending Database Technology, pp. 3–17. Springer, Heidelberg (1996)
71. Suyama, A., Yamaguchi, T.: Specifying and learning inductive learning systems using ontologies. In: Working Notes from the 1998 AAAI Workshop on the Methodology of Applying Machine Learning: Problem Definition, Task Decomposition and Technique Selection (1998)
72. Todorovski, L., Sžeroski, S.: Experiments in meta-level learning with ILP. In: Żytkow, J.M., Rauch, J. (eds.) PKDD 1999. LNCS (LNAI), vol. 1704, pp. 98–106. Springer, Heidelberg (1999)

73. Tsymbal, A., Puuronen, S., Terziyan, V.Y.: Arbiter meta-learning with dynamic selection of classifiers and its experimental investigation. In: Advances in Databases and Information Systems, pp. 205–217 (1999)
74. Utgoff, P.E.: Machine learning of inductive bias. Kluwer Academic Publishers, Dordrecht (1986)
75. Utgoff, P.E.: Shift of bias for inductive learning. In: Michalski, R.S., Carbonell, J.G., Mitchell, T.M. (eds.) Machine Learning. An Artificial Intelligence Approach, ch. 5, vol. 2, pp. 107–148. Morgan Kaufmann, San Francisco (1986)
76. Vanschoren, J., Soldatova, L.: Exposé: An ontology for data mining experiments. In: International Workshop on Third Generation Data Mining: Towards Service-oriented Knowledge Discovery (SoKD 2010) (September 2010)
77. Vilalta, R., Drissi, Y.: A perspective view and survey of meta-learning. Artificial Intelligence Review 18, 77–95 (2002)
78. Vilalta, R., Giraud-Carrier, C., Brazdil, P., Soares, C.: Using meta-learning to support data mining. International Journal of Computer Science and Applications 1(1), 31–45 (2004)
79. Wolpert, D.: The lack of a priori distinctions between learning algorithms. Neural Computation 8(7), 1381–1390 (1996)
80. Yang, Q., Wu, X.: Ten challenging problems in data mining research. International Journal of Inform 5, 594–604 (2006)
81. Zaki, M.: Efficiently mining frequent trees in a forest: algorithms and applications. IEEE Transactions on Knowledge and Data Engineering 17 (2005)
82. Zakova, M., Kremen, P., Zelezny, F., Lavrac, N.: Automating knowledge discovery workflow composition through ontology-based planning. IEEE Transactions on Automation Science and Engineering (2010)

Optimal Support Features for Meta-Learning

Włodzisław Duch[1,2], Tomasz Maszczyk[1], and Marek Grochowski[1]

[1] Department of Informatics, Nicolaus Copernicus University, Grudziądzka 5, Toruń, Poland
[2] School of Computer Engineering, Nanyang Technological University, Singapore

Abstract. Meta-learning has many aspects, but its final goal is to discover in an automatic way many interesting models for a given data. Our early attempts in this area involved heterogeneous learning systems combined with a complexity-guided search for optimal models, performed within the framework of (dis)similarity based methods to discover "knowledge granules". This approach, inspired by neurocognitive mechanisms of information processing in the brain, is generalized here to learning based on parallel chains of transformations that extract useful information granules and use it as additional features. Various types of transformations that generate hidden features are analyzed and methods to generate them are discussed. They include restricted random projections, optimization of these features using projection pursuit methods, similarity-based and general kernel-based features, conditionally defined features, features derived from partial successes of various learning algorithms, and using the whole learning models as new features. In the enhanced feature space the goal of learning is to create image of the input data that can be directly handled by relatively simple decision processes. The focus is on hierarchical methods for generation of information, starting from new support features that are discovered by different types of data models created on similar tasks and successively building more complex features on the enhanced feature spaces. Resulting algorithms facilitate deep learning, and also enable understanding of structures present in the data by visualization of the results of data transformations and by creating logical, fuzzy and prototype-based rules based on new features. Relations to various machine-learning approaches, comparison of results, and neurocognitive inspirations for meta-learning are discussed.

Keywords: Machine learning, meta-learning, feature extraction, data understanding.

1 Introduction: Neurocognitive Inspirations for Meta-Learning

Brains are still far better in solving many complex problems requiring signal analysis than computational models. Already in 1855 H. Spencer in the book "Principles of Psychology" discussed associative basis of intelligence, similarity and dissimilarity, relations between physical events, "psychical changes", and activity of brain parts (early history of connectionism is described in [1]). Why are brains so good in complex signal processing tasks, while machine learning is so poor, despite development of sophisticated statistical, neural network and other biologically-inspired computational intelligence (CI) algorithms?

N. Jankowski et al. (Eds.): Meta-Learning in Computational Intelligence, SCI 358, pp. 317–358.
springerlink.com

Artificial neural networks (ANNs) drew inspiration from neural information processing at single neuron level, initially treating neurons as threshold logic devices, later adding graded response (sigmoidal) neurons [2] and finally creating detailed spiking neural models that are of interest mainly to people in computational neuroscience [3]. Attempts to understand microcircuits and draw inspirations from functions of whole neocortical columns have so far not been too successful. The Blue Brain Project [4] created biologically accurate simulation of neocortical columns, but the project did not provide any general principles how these columns operate. Computational neuroscience is very important to understand details of neural functions, but may not be the shortest way to computational intelligence. Situation in computational quantum physics and chemistry is analogous: despite detailed simulations of molecular properties little knowledge useful for conceptual thinking has been generated.

Neurocognitive inspirations for CI algorithms based on general understanding of brain functions may be quite useful. Intelligent systems should have goals, select appropriate data, extract information from data, create percepts and reason using information derived from them. Goal setting may be a hierarchical process, with many subgoals forming a plan of action or solution to a problem. Humans are very flexible in finding alternative solutions, but current CI methods are focused on searching for a single best solutions. Brains search for alternative solutions recruiting many specialized modules, some of which are used only in very unusual situations. A single neocortical column provides many types of microcircuits that respond in a qualitatively different way to the incoming signals [5]. Other cortical columns may combine these responses in a hierarchical way creating complex hidden features based on information granules extracted from all tasks that may benefit from such information. General principles, such as complementarity of information processed by parallel interacting streams with hierarchical organization are quite useful [6]. Neuropsychological models of decision making assume that noisy stimulus information from multiple parallel streams is accumulated until sufficient information is obtained to make reliable response [7]. Decisions may be made if sufficient number of features extracted by information filters provide reliable information.

Neurocognitive principles provide an interesting perspective on recent activity in machine learning and computational intelligence. In essence, learning may be viewed as a composition of transformations, with parallel streams that discover basic features in the data, and recursively combine them in new parallel streams of higher-order features, including high-level features derived from similarity to memorized prototypes or categories at some abstract level. In the space of such features knowledge is transferred between different tasks and used in solving problems that require sequential reasoning. Neurocognitive inspirations provide a new perspective on: Liquid State Machines [5], "reservoir computing" [8], deep learning architectures [9], deep belief networks [10], kernel methods [11], boosting methods that use week classifiers [12], ensemble learning [13, 14], various meta-learning approaches [15], regularization procedures in feedforward neural networks, and many other machine learning areas.

The key to understanding general intelligence may lie in specific information filters that make learning possible. Such filters have been developed slowly by the evolutionary processes. Integrative chunking processes [16] combine this information into

higher-level mental representations. Filters based on microcircuits discover phonemes, syllables, words in the auditory stream (with even more complex hierarchy in the visual stream), lines and edges, while chunking links sequences of lower level patterns into single higher-level patterns, discovering associations, motifs and elementary objects. Meta-learning tries to reach this level of general intelligence providing additional level of control to search for composition of various transformations, including whole specialized learning modules, that "break and conquer" difficult tasks into manageable subproblems. The great advantage of Lisp programming is that the program may modify itself. There are no examples of CI programs that could adjust themselves in a deeper way, beyond parameter optimization, to the problem analyzed. Constructive algorithms that add new transformations as nodes in a graphic model are a step in this direction.

Computational intelligence tries to create universal learning systems, but biological organisms frequently show patterns of innate behavior that are useful only in rare, quite specific situations. Models that are not working well on all data, but work fine in some specific cases should still be useful. There is "no free lunch" [17], no single system may reach the best results for all possible distributions of data. Therefore instead of a direct attempt to solve all problems with one algorithm, a good strategy is to transform them into one of many formulations that can be handled by selected decision models. This is possible only if relevant information that depends on the set goal is extracted from the input data stream and is made available for decision processes. If the goal is to understand data (making comprehensible model of data), algorithms that extract interesting features from raw data and combine them into rules, find interesting prototypes in the data or provide interesting visualizations of data should be preferred. A lot of knowledge about reliability of data samples, possible outliers, suspected cases, relative costs of features or their redundancies is usually ignored as there is no simple way to use it in the CI programs. Such information is needed to set the meta-learning goals.

Many meta-learning techniques have recently been developed to deal with the problem of model selection [15, 18]. Most of them search for optimal model characterizing a given problem by some meta-features (e.g. statistical properties, landmarking, model-based characterization), and by referring to some meta-knowledge gained earlier. For a given data one can use the classifier that gave the best result on a similar dataset in the StatLog Project [19]. However, choosing good meta-features is not a trivial issue as most features do not characterize the complexity of data distributions. In addition the space of possible solutions generated by this approach is bounded to already known types of algorithms. The challenge is to create flexible systems that can extract relevant information and reconfigure themselves finding many interesting solutions for a given task. Instead of a single learning algorithm designed to solve specialized problem, priorities are set to define what makes an interesting solution, and a search for configurations of computational modules that automatically create algorithms on demand should be performed. This search in the space of all possible models should be constrained by user priorities and should be guided by experience with solving problems of similar nature, experience that defines "patterns of algorithm behavior" in problem solving. Understanding visual or auditory scenes is based on experience and does not seem to require much creativity, even simple animals are better at it than artificial systems. With

no prior knowledge about a given problem finding an optimal sequence of transformations may not be possible.

Meta-learning based on these ideas requires several components:

- specific filters extracting relevant information from raw data, creating useful support features;
- various compositions of transformations that create higher-order features analyzing patterns in enhanced feature spaces;
- models of decision processes based on these high-order features;
- intelligent organization of search that discovers new models of decision processes, learning from previous attempts.

At the meta-level it may not be important that a specific combination of features proved to be successful in some task, but it is important that a specific transformation of a subset of features was once useful, or that distribution of patterns in the feature space had some characteristics that may be described by some specific data model and is easy to adapt to new data. Such information allows for generalization of knowledge at the level of search patterns for a new composition of transformations, facilitating transfer of knowledge between different tasks. Not much is known about the use of heuristic knowledge to guide the search for interesting models and our initial attempts to meta-learning, based on the similarity framework [20, 21] used only simple greedy search techniques. The Metal project [22] tried to collect information about general data characteristics and correlate it with the methods that performed well on a similar data. A system recommending classification methods has been built using this principle, but it works well only in a rather simple cases.

This paper is focused on generation of new features that provide good foundation for meta-learning, creating information on which search processes composing appropriate transformations may operate. The raw features given in the dataset description are used to create a large set of enhanced or hidden features. The topic of feature generation has received recently more attention in analysis of sequences and images, where graphical models known as Conditional Random Fields became popular [23], generating for natural text analysis sometimes millions of low-level features [24]. Attempts at meta-learning on the ensemble level lead to very rough granularity of the existing models and knowledge [25], thus exploring only a small subspace of all possible models, as it is done in the multistrategy learning [26]. Focusing on generation of new features leads to models that have fine granularity of the basic building blocks and thus are more flexible. We have partially addressed this problem in the work on heterogeneous systems [27–34]. Here various types of potentially useful features are analyzed, including higher-order features. Visualization of the image of input data in the enhanced feature space helps to set the priority for application of models that worked well in the past, learning how to transfer meta-knowledge about the types of transformations that have been useful, and transferring this knowledge to new cases.

In the next section various transformations that extract information forming new features are analyzed. Section three shows how transformation based learning may benefit from enhanced feature spaces, how to define goals of learning and how to transfer knowledge between learning tasks. Section four shows a few lessons from applying this line of thinking to real data. The final section contains discussion and conclusions.

2 Extracting Features for Meta-Learning

Brains do not attempt to recognize all objects in the same feature space. Even within the same sensory modality a small subset of complex features is selected, allowing to distinguish one class of objects from another. While the initial receptive fields react to relatively simple information higher order invariant features are extracted from signals as a result of hierarchical processing of multiple streams of information. Object recognition or category assignment by the brain is probably based on evaluation of similarity to memorized prototypes of objects using a few characteristic features [35], but for different classes of objects these features may be of quite different type, i.e. they are class specific. Using different complex features in different regions of the input space may drastically simplify categorization problems. This is possible in hierarchical learning models, graphical models, or using conditionally defined features (see section 2.8).

Almost all adaptive learning systems are homogeneous, based on elements extracting information of the same type. Multilayer Perceptron (MLP) neural networks use nodes that partition the input space by hyperplanes. Radial Basis Function networks based on localized functions frequently use nodes that provide spherical or ellipsoidal decision borders [36]. Similarity-based methods use the same distance function for each reference vector, decision trees use simple tests based on thresholds or subsets of values creating hyperplanes. Support Vector Machines use kernels globally optimized for a given dataset [37]. This cannot be the best inductive bias for all data, frequently requiring large number of processing elements even in cases when simple solutions exist. The problem has been addressed by development of various heterogenous algorithms [31] for neural networks [27–29],neurofuzzy systems [30], decision trees [32] and similarity-based systems [33, 34, 38, 39] and multiple kernel learning methods [40]. Class-specific high order features emerge naturally in hierarchical systems, such as decision trees or rule-based systems [41, 42], where different rules or branches of the tree use different features (see [43, 44]).

The focus of neural network community has traditionally been on learning algorithms and network architectures, but it is clear that selection of neural transfer functions determines the speed of convergence in approximation and classification problems [27, 45, 46]. The famous n-bit parity problem is trivially solved using a periodic function $\cos(\omega \sum_i b_i)$ with a single parameter ω and projection of the bit strings on weight vector with identical values $\mathbf{W} = [1, 1, ..1]$, while the multilayer perceptron (MLP) networks need $O(n^2)$ parameters and have great difficulty to learn such functions [47]. Neural networks are non-parametric universal approximators but the ability to learn requires flexible "brain modules", or transfer functions that are appropriately biased for the problem being solved. Universal learning methods should be non-parametric but they may be heterogeneous.

Initial feature space for a set of objects O is defined by direct observations, measurements, or estimations of similarity to other objects, creating the vector of raw input data ${}^0\mathbf{X}(O) = \mathbf{X}(O)$. These vectors may have different length and in general some structure descriptors, grouping features of the same type. Learning from such data is done by a series of transformations that generate new, higher order features. Several types of transformations of input vectors should be considered: component, selector, linear combinations and non-linear functions. Component transformations, frequently

used in fuzzy modeling [48], work on each input feature separately, scaling, shifting, thresholding, or windowing raw features. Each raw feature may give rise to several new features suitable for calculations of distances, scalar products, membership functions and non-linear combinations at the next stage. Selector transformations define subsets of vectors or subsets of features using various criteria for information selection, distribution of feature values and class labels, or similarity to the known cases (nearest neighbors). Non-linear functions may serve as kernels or as neural transfer functions [27]. These elementary transformations are conveniently presented in a network form.

Initial transformations $\mathcal{T}_1$ of raw data should enhance information related to the learning goals carried by new features. At this stage combining small subsets of features using Hebbian learning based on correlations is frequently most useful. A new dataset ${}^1\mathbf{X} = \mathcal{T}_1({}^0\mathbf{X})$ forms an image of the original data in the space spanned by a new set of features. Depending on the data and goals of learning, this space may have dimensionality that is smaller or larger than the original data. The second transformation ${}^2\mathbf{X} = \mathcal{T}_2({}^1\mathbf{X})$ usually extracts multidimensional information from pre-processed features ${}^1\mathbf{X}$. This requires an estimation which of the possible transformations at the $\mathcal{T}_1$ level may extract information that will be useful for specific $\mathcal{T}_2$ transformations. Many aspects can be taken into account defining such transformations, as some types of features are not appropriate for some learning models and optimization procedures. For example, binary features may not work well with gradient based optimization techniques, and standardization may not help if rule-based solutions are desired. Intelligent search procedures in meta-learning schemes should take such facts into account. Subsequent transformations may use $\mathcal{T}_2$ as well as $\mathcal{T}_1$ and the raw features. The process is repeated until the final transformation is made, aimed either at separation of the data, or at mapping to a specific structures that can be easily recognized by available decision algorithms. Higher-order features created after a series of k transformations ${}^k\mathbf{X}_i$ should also be treated in the same way as raw features. All features influence the geometry of decision regions; this perspective helps to understand their advantages and limitations. All these transformations can be presented in a graphical form. Meta-learning needs also to consider computational costs of different transformations.

2.1 Extracting Information from Single Features

Preprocessing may critically influence convergence of learning algorithms and construction of the final data models. This is especially true in meta-learning, as the performance of various methods if facilitated by different transformations, and it may be worthwhile to apply many transformations to extract relevant information from each feature. Raw input features may contain useful information, but not all algorithms include preprocessing filters to access it easily. How are features ${}^1\mathbf{X}_{ij} = \mathcal{T}_{1j}({}^0\mathbf{X}_i)$, created from raw features ${}^0\mathbf{X}_i$ applying transformation $\mathcal{T}_{1j}$, used by the next level of transformations? They are either used in an additive way in linear combinations for weighted products, or in distance/similarity calculation, or in multiplicative way in probability estimation, or as a logical condition in rules or decision trees with suitable threshold for its value. Methods that compute distances or scalar products benefit from normalization or standardization of feature values. Using logarithmic, sigmoidal, exponential, polynomial and other simple functions to make density of points in one dimension more

uniform may sometimes help to circumvent problems that require multiresolution algorithms. Standardization is relevant to additive use of features in distance calculation (nearest neighbor methods, most kernel methods, RBF networks), it also helps to initialize weights in linear combinations (linear discrimination, MLP), but is not needed for logical rules/decision trees.

Fuzzy and neurofuzzy systems usually include a "fuzzification step", defining for each feature several localized membership functions $\mu_k(X_i)$ that act as receptive fields, filtering out the response outside the range of significant values of the membership functions. These functions are frequently set in an arbitrary way, covering the whole range of feature values with several membership functions that have triangular, Gaussian or similar shapes. This is not the best way to extract information from single features [41]. Filters that work as receptive fields separate subsets or ranges of values that should be correlated with class distribution [49], "perceiving" subsets or intervals where one of the classes dominate. If the correlation of feature values in some interval $[X_{ia}, X_{ib}]$, or a subset of values with some target output is strong membership function $\mu_{ab}(X_i)$ covering these values is useful. This implies that it is not necessary to replace all input features by their fuzzified versions. Class-conditional probabilities $\mathcal{P}(C|X_i)$, as computed by Naive Bayes algorithms, may be used to identify ranges of X_i feature values where a single class dominates, providing optimal membership functions $\mu_k(X_i) = \mathcal{P}(C|X_i)/\mathcal{P}(X_i)$. Negative information, i.e. information about the absence of some classes in certain range of feature values, should also be segmented: if $\mathcal{P}(C_k|X_i) < \epsilon$ in some interval $[X_{ia}, X_{ib}]$ then a derived feature $H_{ikab}(X_i)$, where $H(\cdot)$ is a window-type function, carries valuable information that higher order transformations are able to use. Eliminators may sometimes be more useful than classifiers [50]. Projecting each feature value X_i on these receptive fields μ_k increases the dimensionality of the original data, increasing a chance of finding simple models of the data in the enhanced space.

2.2 Binary Features

Binary features B_i are the simplest, indicating presence or absence of some observations. They may also be created dividing nominal features into two subsets, or creating subintervals of real features $\{X_i\}$. Using filter methods [49], or such algorithms as 1R [51] or Quality of Projected Clusters [52], intervals of real feature values that are correlated with the targets may be selected and presented as binary features. From geometrical perspective binary feature is a label distinguishing two subspaces, projecting all vectors in each subspace on a point 0 or 1 on the coordinate line. The vector of n such features corresponds to all 2^n vertices of the hypercube.

Feature values are usually defined globally, for all available data. Some features are useful only locally in specific context. From geometrical perspective they are projections of vectors that belong to subspaces where specific conditions are met, and should remain undefined for all other vectors. Such conditionally defined features frequently result from questionnaires: if the answer to the last question was yes, then give additional information. In this case for each value $B_i = 0$ and $B_i = 1$ subspaces have different dimensionality. The presence of such features is incorporated in a natural way in graphical models [53], such as Conditional Random Fields [23], but the inference

process is then more difficult than using the flat data where standard classification techniques are used. Enhancing the feature space by adding conditionally defined features may not be so elegant as using the full power of graphical techniques but can go a long way towards improving the results.

Conditionally defined binary features may be obtained by imposing various restrictions on vector subspaces used for projections. Instead of using directly the raw feature B_i conditions $B_i = T \wedge LT_i(\mathbf{X})$, and $B_i = F \wedge LF_i(\mathbf{X})$ are added, where $LT(\mathbf{X}), LF(\mathbf{X})$ are logical functions defining the restrictions using feature vector $\mathbf{X}$. For example, other binary features may create complexes $LT = B_2 \wedge B_3 ... \wedge B_k$ that help to distinguish interesting regions more precisely. Such conditional binary features are created by branch segments in a typical decision tree, for example if one of the path at the two top levels is $X_1 < t_1 \wedge X_2 \geq t_2$, then this defines a subspace containing all vectors for which this condition is true, and in which the third and higher level features are defined.

Such features have not been considered in most learning models, but for problems with inherent logical structure decision trees and logical rules have appropriate bias [41, 42] and thus are a good source for generation of conditionally defined binary features. Similar considerations may be done for nominal features that may sometimes be grouped into larger subsets, and for each value restrictions on their projections applied.

2.3 Real-Valued Features

From geometrical perspective the real-valued input features acquired from various tests and measurements on a set of objects are a projection of some property on a single line. Enhancement of local contrast is very important in natural perception. Some properties directly relevant to the learning task may increase their usefulness after transformation by a non-linear sigmoidal function $\sigma(\beta X_i - t_i)$. Slopes β and thresholds t_i may be individually optimized using mutual information or other relevance measures independently for each feature.

Single features may show interesting patterns of $p(C|X)$ distributions, for example a periodic distribution, or k pure clusters. Projections on a line that show k-separable data distributions are very useful for learning complex Boolean functions. For n-bit parity problem $n + 1$ separate clusters may be distinguished in projections on the long diagonal, forming useful new features. A single large cluster of pure values is worth turning into a new feature. Such features are generated by applying bicentral functions (localized window-type functions) to original features [52], for example $Z_i = \sigma(X_i - a_i) - \sigma(X_i - b_i)$, where $a_i > b_i$. Changing σ into a step function will lead to a binary features, filtering vectors for which logical condition $X_i \in [a_i, b_i]$ is true. Soft σ creates window-like membership functions, but may also be used to create higher-dimensional features, for example $Z_{12} = \sigma(t_1 - X_1)\sigma(X_2 - t_2)$.

Providing diverse receptive fields for sampling the data separately in each dimension is of course not always sufficient, as two or higher-dimensional receptive fields are necessary in some applications, for example in image or signal processing filters, such as wavelets. For real-valued features simplest constraints are made by products of intervals $\prod_i [r_i^-, r_i^+]$, or product of bicentral functions defining hyperboxes in which projected vectors should lie. Other ways to restrict subspaces used for projection may

be considered, for example taking only vectors that are in a cylindrical area surrounding the X_1 coordinate $Z_{1d} = \sigma(X_1 - t_1)\sigma(d - ||\mathbf{X}||_{-1})$, where $||\mathbf{X}||_{-1}$ norm excludes X_1 feature. The point here is that transformed features should label different regions of feature space simplifying the analysis of data in these regions.

2.4 Linear Projections

Groups of several correlated features may be replaced by a single combination performing principal component analysis (PCA) restricted to small subspaces. To decide which groups should be combined standardized Pearson's linear correlation is calculated:

$$r_{ij} = 1 - \frac{|C_{ij}|}{\sigma_i \sigma_j} \in [-1, +1] \tag{1}$$

where the covariance matrix is:

$$C_{ij} = \frac{1}{n-1} \sum_{k=1}^{n} \left(X_i^{(k)} - \bar{X}_i \right) \left(X_j^{(k)} - \bar{X}_j \right); \quad i, j = 1 \cdots d \tag{2}$$

Correlation coefficients may be clustered using dendrogram or other techniques. Linear combinations of strongly correlated features allow not only for dimensionality reduction, but also for creation of features at different scales, from a combination of a few features, to a global PCA combinations of all features. This approach may help to discover hierarchical sets of features that are useful in problems requiring multiscale analysis. Another way to obtain features for multiscale problems is to do clusterization in the input data space and make local PCA within the clusters to find features that are most useful locally in various areas of space.

Exploratory Projection Pursuit Networks (EPPNs) [54, 55] is a general technique that may be used to define transformations creating new features. Quadratic cost functions used for optimization of linear transformations may lead to formulation of the problem in terms of linear equations, but most cost functions or optimization criteria are nonlinear even for linear transformations. A few unsupervised transformations are listed below:

- Principal Component Analysis (PCA) in its many variants provides features that correspond to feature space directions with the highest variance [17, 56, 57].
- Independent Component Analysis provides features that are statistically independent [58, 59].
- Classical scaling, or linear transformation embedding input vectors in a space where distances are preserved [60].
- Factor analysis, computing common and unique factors.

Many supervised transformations may be used to determine coefficients for combination of input features, as listed below.

- Any measure of dependency between class and feature value distributions, such as the Pearson's correlation coefficient, χ^2, separability criterion [61],

- Information-based measures [49], such as the mutual information between classes and new features [62], Symmetric Uncertainty Coefficient, or Kullback-Leibler divergence.
- Linear Discriminatory Analysis (LDA), with each feature based on orthogonal LDA direction obtained by one of the numerous LDA algorithms [17, 56, 57], including linear SVM algorithms.
- Fisher Discriminatory Analysis (FDA), with each node computing canonical component using one of many FDA algorithms [56, 63].
- Linear factor analysis, computing common and unique factors from data [64].
- Canonical correlation analysis [65].
- Localized projections of pure clusters using various projection pursuit indices, such as the Quality of Projected Clusters [52].
- General projection pursuit transformations [54, 55] provide a framework for various criteria used in searching for interesting transformations.

Many other transformations of this sort are known and may be used at this stage in transformation-based systems. **The Quality of Projected Clusters** (QPC) is a projection pursuit method that is based on a leave-one-out estimator measuring quality of clusters projected on $\mathbf{W}$ direction. The supervised version of this index is defined as [52]:

$$QPC(\mathbf{W}) = \sum_{\mathbf{X}} \left(A^{+} \sum_{\mathbf{X}_k \in \mathcal{C}_{\mathbf{X}}} G\left(\mathbf{W}^T(\mathbf{X}-\mathbf{X}_k)\right) - A^{-} \sum_{\mathbf{X}_k \notin \mathcal{C}_{\mathbf{X}}} G\left(\mathbf{W}^T(\mathbf{X}-\mathbf{X}_k)\right) \right) \tag{3}$$

where $G(x)$ is a function with localized support and maximum for $x = 0$ (e.g. a Gaussian function), and $\mathcal{C}_{\mathbf{X}}$ denotes the set of all vectors that have the same label as $\mathbf{X}$. Parameters A^{+}, A^{-} control influence of each term in Eq. (3). For large value of A^{-} strong separation between classes is enforced, while increasing A^{+} impacts mostly compactness and purity of clusters. Unsupervised version of this index may simply try to discover projection directions that lead to separated clusters. This index achieves maximum value for projections on the direction $\mathbf{W}$ that group vectors belonging to the same class into a compact and well separated clusters. Therefore it is suitable for multi-modal data [47]).

The shape and width of the $G(x)$ function used in E.q. (3) has influence on convergence. For continuous functions $G(x)$ gradient-based methods may be used to maximize the QPC index. One good choice is an inverse quartic function: $G(x) = 1/(1 + (bx)^4)$, but any bell-shaped function is suitable here. Direct calculation of the QPC index (3), as in the case of all nearest neighbor methods, requires $O(n^2)$ operations, but fast version, using centers of clusters instead of pairs of vectors, has only $O(n)$ complexity (Grochowski and Duch, in print). The QPC may be used also (in the same way as the SVM approach described above) as a base for creation of feature ranking and feature selection methods. Projection coefficients W_i indicate then significance of the i-th feature. For noisy and non-informative variables values of associated weights should decrease to zero during QPC optimization. Local extrema of the QPC index may

provide useful insight into data structures and may be used in a committee-based approach that combines different views on the same data. More projections are obtained repeating procedure in the orthogonalized space to create sequence of unique interesting projections [52].

Srivastava and Liu [66] analyzed optimal transformations for different applications presenting elegant geometrical formulation using Stiefel and Grassmann manifolds. This leads to a family of algorithms that generate orthogonal linear transformations of features, optimal for specific tasks and specific datasets. PCA seems to be optimal transformation for image reconstruction under mean-squared error, Fisher discriminant for classification using linear discrimination, ICA for signal extraction from a mixture using independence, optimal linear transformation of distances for the nearest neighbor rule in appearance-based recognition of objects, transformations for optimal generalization (maximization of margin), sparse representations of natural images and retrieval of images from a large database. In all these applications optimal transformations are different and may be found by optimizing appropriate cost functions. Some of the cost functions advocated in [66] may be difficult to optimize and it is not yet clear that sophisticated techniques based on differential geometry offer significant practical advantages. Simpler learning algorithms based on numerical gradient techniques and systematic search algorithms give surprisingly good results and can be applied to optimization of difficult functions [67].

2.5 Kernel Features

The most popular type of SVM algorithm with localized (usually Gaussian) kernels [11] suffers from the curse of dimensionality [68]. This is due to the fact that such algorithms rely on assumption of uniform resolution and local similarity between data samples. To obtain accurate solution often a large number of training examples used as support vectors is required. This leads to high cost of computations and complex models that do not generalize well. Much effort has been devoted to improvements of the scaling [69, 70], reducing the number of support vectors, [71], and learning multiple kernels [40]. All these developments are impressive, but there is still room for simpler, more direct and comprehensible approaches.

In general the higher the dimensionality of the transformed space the greater the chance that the data may be separated by a hyperplane [36]. One popular way of creating highly-dimensional representations without increasing computational costs is by using the kernel trick [11]. Although this problem is usually presented in the dual space the solution in the primal space is conceptually simpler [70, 72]. Regularized linear discriminant (LDA) solution is found in the new feature space ${}^2X = \mathbf{K}(\mathbf{X}) = K({}^1\mathbf{X}, \mathbf{X})$, mapping $\mathbf{X}$ using kernel functions for each training vector. Kernel methods work because they implicitly provide new, useful features $Z_i(\mathbf{X}) = K(\mathbf{X}, \mathbf{X}_i)$ constructed by taking the support vectors $\mathbf{X}_i$ as reference. Linear SVM solutions in the $\mathbf{Z}$ kernel feature space are equivalent to the SVM solutions, as it has been empirically verified [73].

Feature selection techniques may be used to leave only components corresponding to "support vectors" that provide essential support for classification, for example only those that are close to the decision borders or those close to the centers of cluster, depending on the type of the problem. Once a new feature is proposed it may be evaluated

on vectors that are classified at a given stage with low confidence, thus ensuring that features that are added indeed help to improve the system. Any CI method may be used in the kernel-based feature space $K(\mathbf{X})$. This is the idea behind Support Feature Machines [73]. If the dimensionality is large data overfitting is a big danger, therefore only the simplest and most robust models should be used. SVM solution to use LDA with margin maximization is certainly a good strategy.

Explicit generation of features based on different similarity measures [39] removes one of the SVM bottleneck allowing for optimization of resolution in different areas of the feature space, providing strong non-linearities where they are needed (small dispersions in Gaussian functions), and using smooth functions when this is sufficient. This technique may be called **adaptive regularization**, in contrast to a simple regularization based on minimization of the norm of the weight vector $||\mathbf{W}||$ used in SVM or neural networks. Although simple regularization enforces smooth decision borders decreasing model complexity it is not able to find the simplest solutions and may easily miss the fact that a single binary feature contains all information. Generation of kernel features should therefore proceed from most general, placed far from decision border (such vectors may be easily identified by looking at the $z = \mathbf{W} \cdot \mathbf{X}$ distribution for $\mathbf{W} = (\boldsymbol{m}_1 - \boldsymbol{m}_2)/||\boldsymbol{m}_1 - \boldsymbol{m}_2||$, where $\boldsymbol{m}_1$ and $\boldsymbol{m}_2$ denote center points of two opposite classes), to more specific, with non-zero contribution only close to decision border. If dispersions are small many vectors far from decision borders have to be used to create kernel space, otherwise all such vectors, independently of the class, would be mapped to zero point (origin of the coordinate system). Adding features based on linear projections will remove the need for support vectors that are far from decision borders.

Kernel features based on radial functions are projections on one radial dimension and in this sense are similar to the linear projections. However, linear projections are global and position independent, while radial projections use reference vector $K(\mathbf{X}, \mathbf{R}) = ||\mathbf{X} - \mathbf{R}||$ that allows for focusing on the region close to $\mathbf{R}$. Additional scaling factors are needed to take account of importance of different features $K(\mathbf{X}, \mathbf{R}; \mathbf{W}) = ||\mathbf{W} \cdot (\mathbf{X} - \mathbf{R})||$. If Gaussian kernels are used this leads to features of the $G(\mathbf{W}(\mathbf{X} - \mathbf{R}))$ type. More sophisticated features are based on Mahalanobis distance calculated for clusters of vectors located near decision borders (an inexpensive method for rotation of density functions with d parameters has been introduced in [27]), or flat local fronts using cosine distance.

There is a whole range of features based on projections on more than one dimension. Mixed "cylindrical" kernel features that are partially radial and partially linear may also be considered. Assuming that $||W|| = 1$ linear projection $y = \mathbf{W} \cdot \mathbf{X}$ defines one direction in the n-dimensional feature space, and at each point y projections are made from the remaining $n - 1$ dimensional subspaces orthogonal to $\mathbf{W}$, such that $||\mathbf{X} - y\mathbf{W}|| < \theta$, forming a cylinder in the feature space. In general projections may be confined to k-dimensional hyperplane and radial dimensions to the $(n-k)$-dimensional subspace. Such features have never been systematically analyzed and there are no algorithms aimed at their extraction. They are conditionally defined in a subspace of the whole feature space, so for some vectors they are not relevant.

2.6 Other Non-linear Mappings

Linear combinations derived from interesting projection directions may provide low number of interesting features, but in some applications non-linear processing is essential. The number of possible transformations in such case is very large. Tensor products of features are particularly useful, as Pao has already noted introducing functional link networks [74, 75]. Rational function neural networks [36] in signal processing [76] and other applications use ratios of polynomial combinations of features; a linear dependence on a ratio $y = x_1/x_2$ is not easy to approximate if the two features x_1, x_2 are used directly. The challenge is to provide a single framework for systematic selection and creation of interesting transformations in a meta-learning scheme.

Linear transformations in the kernel space are equivalent to non-linear transformations in the original feature space. A few non-linear transformations are listed below:

- Kernel versions of linear transformations, including radial and other basis set expansion methods [11].
- Weighted distance-based transformations, a special case of general kernel transformations, that use (optimized) reference vectors [39].
- Perceptron nodes based on sigmoidal functions with scalar product or distance-based activations [77, 78], as in layers of MLP networks, but with targets specified by some criterion (any criterion used for linear transformations is sufficient).
- Heterogeneous transformations using several types of kernels to capture details at different resolution [27].
- Heterogeneous nodes based or several type of non-linear functions to achieve multiresolution transformations [27].
- Nodes implementing fuzzy separable functions, or other fuzzy functions [79].
- Multidimensional scaling (MDS) to reduce dimensionality while preserving distances [80].

MDS requires costly minimization to map new vectors into reduced space; linear approximations to multidimensional scaling may be used to provide interesting features [60]. If highly nonlinear low-dimensional decision borders are needed large number of neurons should be used in the hidden layer, providing linear projection into high-dimensional space followed by squashing by the neural transfer functions to normalize the output from this transformation.

2.7 Adaptive Models as Features

Meta-learning usually leads to several interesting models, as different types of features and optimization procedures used by the search procedure may create roughly equivalent description of individual models. The output of each model may be treated as a high-order feature. This reasoning is motivated both from the neurocognitive perspective, and from the machine learning perspective. Attention mechanisms are used to save energy and inhibit parts of the neocortex that are not competent in analysis of a given type of signal. All sensory inputs (except olfactory) travel through the thalamus where their importance and rough category is estimated. Thalamic nuclei activate only those

brain areas that may contribute useful information to the analysis of a given type of signals [81].

Usually new learning methods are developed with the hope that they will be universally useful. However, evolution has implanted in brains of animals many specialized behaviors, called instincts. From the machine learning perspective a committee of models should use diverse individual models specializing in analysis of different regions of the input space, especially for learning difficult tasks. Individual models are frequently unstable [82], i.e. quite different models are created as a result of repeated training (if learning algorithms contains stochastic elements) or if the training set is slightly perturbed [83]. The mixture of models allows for approximation of complicated probability distributions improving stability of individual models. Specialized models that handle cases for which other models fail should be maintained. In contrast to boosting [12] and similar procedures [84] explicit information about competence of each model in different regions of the feature space should be used. Functions describing these regions of competence (or incompetence) may be used for regional boosting [85] or for integration of decisions of individual models [14, 86]. The same may be done with some features that are useful only in localized regions of space but should not be used in other regions.

In all areas where some feature or the whole model M_l works well the competence factor should reach $F(\mathbf{X}; M_l) \approx 1$ and it should decrease to zero in regions where many errors are made. A Gaussian-like function may be used, $F(||\mathbf{X}-\mathbf{R}_i||; M_l) = 1 - G(||\mathbf{X}-\mathbf{R}_i||^a; \sigma_i)$, where $a \geq 1$ coefficient is used to flatten the function, or a simpler $1/\left(1 + ||\mathbf{X}-\mathbf{R}_i||^{-a}\right)$ inverse function, or a logistic function $1-\sigma(a(||\mathbf{X}-\mathbf{R}_i|| - b))$, where a defines its steepness and b the radius where the value drops to 1/2. Multiplying many factors in the incompetence function of the model may decrease the competence values, therefore each factor should quickly reach 1 outside the incompetence area. This is achieved by using steep functions or defining a threshold values above which exactly 1 is taken.

The final decision based on results of all $l = 1 \ldots m$ models providing estimation of probabilities $\mathcal{P}(C_i|\mathbf{X}; M_l)$ for $i = 1 \ldots K$ classes may be done using majority voting, averaging results of all models, selecting a single model that shows highest confidence (i.e. gives the largest probability), selecting a subset of models with confidence above some threshold, or using simple linear combination [13]. In the last case for class C_i coefficients of linear combination are determined from the least-mean square solution of:

$$\begin{aligned} \mathcal{P}(C_i|\mathbf{X}; M) &= \sum_{l=1}^{m} \sum_{m} W_{i,l}(\mathbf{X}) \mathcal{P}(C_i|\mathbf{X}; M_l) \\ &= \sum_{l=1}^{m} \sum_{m} W_{i,l} F(\mathbf{X}; M_l) \mathcal{P}(C_i|\mathbf{X}; M_l) \end{aligned} \tag{4}$$

The incompetence factors simply modify probabilities $F(\mathbf{X}; M_l)\mathcal{P}(C_i|\mathbf{X}; M_l)$ that are used to set linear equations for all training vectors $\mathbf{X}$, therefore the solution is done in the same way as before. The final probability of classification is estimated by renormalization $\mathcal{P}(C_i|\mathbf{X}; M)/\sum_j \mathcal{P}(C_j|\mathbf{X}; M)$. In this case results of each model are used

as high order feature for local linear combination of results. This approach may also be justified using neurocognitive inspirations: thalamo-cortical loops control which brain areas should be strongly activated depending on their predicted competence.

In different regions of the input space (around reference vector $\mathbf{R}$) kernel features $K(\mathbf{X}, \mathbf{R})$ that use weighted distance functions should have zero weights for those input features that are locally irrelevant. Many variants of committee or boosting algorithms with competence are possible [13], focusing on generation of diversified models, Bayesian framework for dynamic selection of most competent classifier [87], regional boosting [85], confidence-rated boosting predictions [12], task clustering and gating approach [88], or stacked generalization [89, 90].

2.8 Summary of the Feature Types

Features are weighted and combined by distance functions, kernels, hidden layers, and in many other ways, but geometrical perspective shows what kind of information can be extracted from them. What types of subspaces and hypersurfaces that contained them are generated? An attempt to categorize different types of features from this perspective, including conditionally defined features, is shown below. X represents here arbitrary type of scalar feature, B is binary, N nominal, R continuous real valued, K is general kernel feature, M are motifs in sequences, and S are signals.

- B1) Binary, equivalent to unrestricted projections on two points.
- B2) Binary, constrained by other binary features, complexes $B_1 \wedge B_2 ... \wedge B_k$, subsets of vertices of a cube.
- B3) Binary, projection of subspaces constrained by a distance $B = 0 \wedge R_1 \in [r_1^-, r_1^+] ... \wedge R_k \in [r_k^-, r_k^+]$.
- N1-N3) Nominal features are similar to binary with subsets instead of intervals.
- R1) Real, equivalent to unrestricted orthogonal projections on a line, with thresholds and rescaling.
- R2) Real, orthogonal projections on a line restricted by intervals or soft membership functions, selecting subspaces orthogonal to the line.
- R3) Real, orthogonal projections with cylindrical constrains restricting distance from the line.
- R4) Real, any optimized projection pursuit on a line (PCA, ICA, LDA, QPC).
- R5) Real, any projection on a line with periodic or semi-periodic intervals or general 1D patterns, or posterior probabilities for each class calculated along this line $p(C|X)$.
- K1) Kernel features $K(\mathbf{X}, \mathbf{R}_i)$ with reference vectors $\mathbf{R}_i$, projections on a radial coordinate creating hyperspheres.
- K2) Kernel features with intervals, membership functions and general patterns on a radial coordinate.
- K3) General kernel features for similarity estimation of structured objects.
- M1) Motifs, based on correlations between elements and on sequences of discrete symbols.

- S1) Signal decompositions and projections on basis functions.
- T1) Other non-linear transformations restricting subspaces in a more complex way, rational functions, universal transfer functions.

Combinations of different types of features, for example cylindrical constraints with intervals or semi-periodic functions are also possible. The classification given above is not very precise and far from complete, but should give an idea what type of decision borders may be generated by different types of features. Higher-order features may be build by learning machines using features that have been constructed by earlier transformations. Relevance indices applied to these features, or feature selection methods, should help to estimate their importance, although some features may be needed for local representation of information only, so their global relevance may be low [49].

3 Transformation-Based Meta-Learning

A necessary step for meta-learning is to create taxonomy, categorizing and describing similarities and relations among transformations and facilitate systematic search in the space of all possible compositions of these transformations. An obvious division is between transformations optimized locally with well-defined targets, and adaptive transformations that are based on a distal criteria, where the targets are defined globally, for composition of transformations (as in backpropagation). In the second case interpretation of features implemented by hidden nodes is rather difficult. In the first case activity of the network nodes implementing fixed transformations has clear interpretation, and increased complexity of adding new node should be justified by discovery of new aspects of the data. Local $\mathcal{T}_2$ transformations have coefficients calculated directly from the input data or data after $\mathcal{T}_1$ transformation. They may be very useful for initialization of global adaptive transformations, or may be useful to find better solutions of more complex fixed transformations. For example, multidimensional scaling requires very difficult minimization and most of the time converges to a better solution if PCA transformations is performed first.

After initial transformations all data is converted to internal representation ${}^k\mathbf{X}$, forming a new image of the data, distributed in a simpler way than the original image. The final transformation should be able to extract desired information form this image. If the final transformation is linear $\mathbf{Y} = {}^{k+1}\mathbf{X} = \mathcal{T}_{k+1}({}^k\mathbf{X}; {}^k\mathbf{W})$ parameters ${}^k\mathbf{W}$ are either determined in an iterative procedure simultaneously with all other parameters $\mathbf{W}$ from previous transformations (as in the backpropagation algorithms [36]), or sequentially determined by calculating the pseudoinverse transformation, as is frequently practiced in the two-phase RBF learning [91]. Simultaneous adaptation of all parameters (RBF centers, scaling parameters, output layer weights) in experiments on more demanding data gives better results.

Three basic strategies to create composition of transformations are:

- Use constructive method adding features based on simple transformations; proceed as long as increased quality justifies added complexity [29, 92].
- Start from complex transformations and optimize parameters, for example using flexible neural transfer functions [28, 93], optimizing each transformation before adding the next one.

– Use pruning and regularization techniques for large network with nodes based on simple transformations and global optimization [36].

The last solution is the most popular in neural network community, but there are many other possibilities. After adding each new feature the image of the data in the extended feature space is changed and new transformations are created in this space, not in the original one. For example, adding more transformations with distance-based conditions may add new kernel features and start to build the final transformation assigning significant weights only to the kernel-based support features. This may either be equivalent to the kernel SVM (for linear output transformations) created by evaluation of similarity in the original input space, or to the higher-order nearest neighbor methods, so far little explored in machine learning. From geometrical perspective kernel transformations are capable of smoothing or flatting decision borders: using support vectors $\mathbf{R}$ that lie close to complex decision border in the input space $\mathcal{X}$ a combination of kernel features $\mathbf{W} \cdot K(\mathbf{X}, \mathbf{R}) = \text{const}$ lies close to a hyperplane in the kernel space $\mathcal{K}$. A single hyperplane after such transformation is frequently sufficient to achieve good separation of data. This creates similar decision borders to the edited k-NN approach with support vectors as references, although the final linear model avoids overfitting in a better way. However, if the data has complex logical structure, with many disjoint clusters from the same class, this is not an optimal approach.

Geometry of heteroassociative vector transformations, from the input feature space to the output space, is quite important and leads to transformations that will be very useful in meta-learning systems, facilitating learning of arbitrary problems. At each point of the input space relative importance of features may change. One way to implement this idea [38] is to create local non-symmetric similarity function $D(\mathbf{X} - \mathbf{Y}; \mathbf{X})$, smoothly changing between different regions of the input space. For example, this may be a Minkovsky function $D(\mathbf{X} - \mathbf{Y}; \mathbf{X}) = \sum_i s_i(\mathbf{X})|X_i - Y_i|$ with the scaling factor that depend on the point $\mathbf{X}$ of the input space. Many factors are very small or zero. They may be calculated for each training vector using local PCA, and interpolated between the vectors. Local Linear Embedding (LLE) is a popular method of this sort [94] and many other manifold learning methods have been developed. Alternatively a smooth mapping may be generated by MLP training or other neural networks to approximate desired scaling factors.

Prototype rules for data understanding and transformation may be created using geometrical learning techniques that construct a convex hull encompassing the data, for example an enclosing polytope, cylinder, a set of ellipsoids or some other surface enclosing the data points. Although geometrical algorithms may be different than neural or SVM algorithms, the decision surfaces they provide are similar to those offered by feedforward networks. A covering may be generated by a set of balls or ellipsoids following principal curve, for example using the piecewise linear skeletonization approximation to principal curves [95]. One algorithm of this type creates a "hypersausage" decision regions [96]. One-class SVM also provides covering in the kernel space [11].

Kernel methods expand dimensionality of the feature space if the number of samples is larger than the number of input features (see neurobiological justification of such projections in [5]). Enlarging the data dimensionality increases the chance to make the data separable, and this is frequently the goal of this transformation, ${}^2\mathbf{X} = \mathcal{T}_2({}^1\mathbf{X}; {}^1\mathbf{W})$.

Random linear projections of input vectors into a high-dimensional space ${}^2\mathbf{X} = \mathbf{L}({}^1\mathbf{X})$ are the simplest way to increase dimensionality, with the random matrix $\mathbf{L}$ that has more rows than columns. The final transformation is chosen to be linear $\mathbf{Y} = \mathcal{T}_3({}^2\mathbf{X}; {}^2\mathbf{W}) = {}^2\mathbf{W} \cdot {}^2\mathbf{X}$, although it may not be the best solution and other classifiers may be used on the enhanced feature space. This is basically equivalent to random initialization of feed-forward neural networks with linear transfer functions only. Such methods are used to start a two-phase RBF learning [91]. For simple data random projections work rather well [97], but one should always check results of linear discrimination in the original feature space, as it may not be significantly worse. Many non-random ways to create interesting features may certainly give better results. It may also be worthwhile to add pre-processed ${}^1\mathbf{X} = \mathcal{T}_1(\mathbf{X})$ features to the new features generated by the second transformation ${}^2\mathbf{X} = ({}^1\mathbf{X}, \mathcal{T}_2({}^1\mathbf{X}; {}^1\mathbf{W}))$, because they are easier to interpret and frequently contain useful information.

3.1 Redefining the Goal of Learning

Multi-objective optimization problems do not have a single best solution [98]. Usually data mining systems return just a single best model but if several criteria are optimized finding a set of Pareto optimal models is a better goal. For example, accuracy should be maximized, but variance should be minimized, or sensitivity should be maximized while the false alarm rate should be kept below some threshold. The search process for optimal models in meta-learning should explore many compositions of transformations retaining those that are close to the Pareto front. A forest of heterogeneous decision trees [32] is an example of a multi-objective meta-search in a model space restricted to decision trees. Heterogeneous trees use different types of rule premises, splitting the branches not only using individual features, but also using tests based on kernel features, defined by the weighted distances from the training data vectors. Adding distance-based conditions with optimal support vectors far from decision borders provides flat spherical borders that approximate hyperplanes in the border region. The beam search maintains at each stage k decision trees (search states), ordering them by their accuracy estimated using cross-validation on the training data [32]. This algorithm has found some of the simplest and most accurate decision rules that gave different tradeoffs between sensitivity and specificity.

Each data model depends on some specific assumptions about the data distribution in the input space, and is successfully applicable only to some types of problems. For example SVM and many other statistical learning methods [11] rely on the assumption of uniform resolution, local similarity between data samples, and may completely fail in case of high-dimensional functions that are not sufficiently smooth [68]. In such case accurate solution may require an extremely large number of training samples that will be used as reference vectors, leading to high cost of computations and creating complex models that do not generalize well. To avoid any bias useful "knowledge granules" in the data should be discovered. Support features created through parallel hierarchical streams of transformations that discover interesting aspects of data are focused on local improvements rather than some global goal, such as data separability. The image of the original data in the enhanced space may have certain characteristic patterns that the decision processes should learn about. The final transformations should have several

different biases and the meta-learning search should try to match the best one to the image of the data. The goal of learning should then focus on creation of one of the standard types of such images rather than linear separability.

One way to discover what type of structures emerge after data transformations is to use visualization of the data images in the original feature space and in the enhanced space [99, 100]. PCA, ICA and QPC projections may show interesting structures in the data. Multidimensional Scaling (MDS) [80] is a non-linear mapping that tries to faithfully display distances between vectors. Also projections based on directions obtained from linear SVM are useful. The first projection on $\mathbf{W}_1$ line for linearly separable data should give $y(\mathbf{X};\mathbf{W}_1) = \mathbf{W}_1 \cdot \mathbf{X} + \theta < 0$ for vectors from the first class, and $y(\mathbf{X};\mathbf{W}_1) > 0$ for the second class. The second best direction may then be obtained by repeating SVM calculations in the space orthogonalized to the $\mathbf{W}_1$ direction. This process may be repeated to obtain more dimensions. Fisher Discriminant Analysis (FDA) is another linear discriminant that may be used for visualization [56].

Visualization of transformations in case of difficult logical problems reveals the nature of difficulties and helps to set simpler goals for learning. Consider a parity-like problem: each vector labeled as even is surrounded by vectors labeled as odd and vice versa [47]. Localized transformations are not be able to generalize such information but linear projections may provide interesting views on such data. For n-bit parity linear projection $y = \mathbf{W} \cdot \mathbf{X}$, where $\mathbf{W} = [1, 1...1]$, counts the number of 1 bits, producing alternating clusters with vectors that belong to the odd and even classes. A periodic function (such as cosine) solves the parity problem using a single parameter, but will not handle other logical problems. In case of many Boolean functions finding transformations that lead to the k-separable solutions, with single-vectors from a single class in intervals $[y_i, y_{i+1}]$ along the projection line defines much easier goal than achieving separability. The whole feature space is divided into parallel slices, orthogonal to the $\mathbf{W}$ line. Such solutions are equivalent to a single prototype $\mathbf{P}_i$ in the middle of each $[y_i, y_{i+1}]$ interval, with the nearest neighbor decision rules using Euclidean distance function. They may also be generated using projections on a radial direction satisfying $K(\mathbf{X}, \mathbf{R}) = 1$ for $a \leq ||\mathbf{X}-\mathbf{R}|| \leq b$. This kernel feature is zero outside of the spherical shell between the distance a and b from $\mathbf{R}$. For binary hypercube such features discover large pure clusters of data.

The number of parameters that fully describes such solution in n-dimensional feature space is $n + k - 1$. If these prototypes are not on a single line the nearest neighbor rule will create Voronoi tessellation of the feature space and if each Voronoi region contains vectors from a single class the solution may be called q-separable, where q is the lowest number of Voronoi regions that is sufficient to separate the data into pure clusters. This requires qn parameters but depending on the distributions of these regions simpler solutions may exist. Consider for example a 3 by 3 regular board defined in two dimensions by 4 lines (two parallel lines in each direction). These lines divide the space into 9 regions, but instead of 9 prototypes (18 parameters) only 4 lines (12 parameters) are sufficient. On the other hand describing k hyperspheres in n-dimensional space is easy if prototypes with radial threshold functions are used, requiring $k(n + 1)$ parameters, while the same data distribution will be very hard to classify using transformations based on linear projections. Characterization of the complexity of the learning problem

should thus be done with reference to the types of transformations and the number of parameters that are needed to describe the solution.

Useful features may be generated capturing frequent correlations of inputs (Hebbian learning, PCA, ICA, discovering motifs), or searching for clusters of relatively pure data using linear and radial projections. Visualizing resulting images of data should reveal what types of methods are most appropriate for further analysis.

3.2 Transfer of Knowledge

According to the "no free lunch" theorem [17] no single adaptive system may reach the best results for all possible distributions of data. It is therefore worthwhile to look at what different algorithms may do well and when they fail. Data with simple logical structure require sharp decision borders provided by decision trees and rule-based systems [41, 42], but are quite difficult to analyze with statistical or neural algorithms. SVM will miss simple solution where the best answer is given by a single binary feature. Frequently data has Gaussian distribution and linear discrimination (linear SVM, simple MLP networks) provides the best solution. k-NN and SVM in kernelized form work well when decision borders have complex topology, but fail when sharp decision borders are needed or when data structure has complex Boolean logic [101]. Neural networks suffer from similar problems as SVM and will not converge for highly non-separable problems (in the k-separability sense). New methods are frequently invented and tested on data that are almost Gaussian-like, and thus are very easy to analyze, so it is important to assign complexity estimate for different classification problems. Basis Set Function networks (Radial or Separable) may provide local description but have problems with simple decision borders creating complex models.

Different adaptive systems have biases that makes them suitable for particular classes of problems. Discovering this bias and finding an appropriate model is usually done by tedious experimentations with combinations of pre-processing, filtering and selection, clusterization, classification or regression and post-processing techniques, combined with meta-learning procedures based on stacking, boosting, committees and other techniques. The number of possible combinations of different modules in large data mining packages exceeds now 10 billions, and new modules are still added. With proper control of search and complexity of generated models [102, 103] automatic composition of transformations guided by geometrical perspective for creation of features offers an interesting approach that may overcome the limits of the "no free-lunch" theorem. Universal learning is an elusive dream that will not be realized without diverse transformations, specific for each application. Success of meta-search relies on the availability of specific transformations for image analysis, multimedia streams, signal decomposition, text analysis, biosequences and many other problems. Finding proper representation of the problem is more than half of the solution. While these specific problems are not addressed here it is worthwhile to analyze methods that may be applied to derive useful features from typical measurements, as found in benchmark databases.

One strategy frequently used by people is to learn directly from others. Although each individual agent rarely discovers something interesting, in a population of agents that try different approaches accidental observations are exchanged and, if found useful, become common know-how. Transfer learning is concerned with learning a number of

related tasks together. In image, text analysis or robotics many methods have been devised for knowledge transfer. Related machine learning subjects include: learning from hints [104], lifelong learning [105], multi-task learning [106], cross-domain learning [107, 108], cross-category learning [109] and self-taught learning [110]. EigenTransfer algorithm [111] tries to unify various transfer learning ideas representing the target task by a graph. The task graph has nodes with vectors and labels, connecting the target and auxiliary data in the same feature space. Eigenvectors of this task graph are used as new features to transfer knowledge from auxiliary data to help classify target data. Significant improvements have been demonstrated in various transfer learning task domains.

Current approaches to transfer learning focus on using additional data to create a better learning model for a given training data. The same feature space is used and the same learning algorithm. This type of transfer learning is not suitable for meta-learning. In the Universal Learning Machine (ULM) algorithm [112] transfer of knowledge between different algorithms is made by sharing new higher-order features that have been successful in discovering knowledge granules in one of these algorithms. Decision trees and rule-based algorithms discovered binary features (B1-B3 type). Real R1-R4 types of features are discovered by projection pursuit, linear SVM and simple projections on the line connecting centers of local clusters. Naive Bayes provides $p(C|X)$ posterior probabilities along these lines. Edited k-NN and kernel methods find good kernel features based on similarity. The best features are easily identified using ranking methods. In the experiments performed using this idea [112] significant improvements almost in every algorithm has been found by adding a few features from other algorithms. For example, on the hypothyroid problem (3 classes, 3772 training cases and 3428 test cases, 15 binary and 6 continuous features) adding two binary features discovered by decision tree improved test results of SVM with Gaussian kernel from 94.1 to 99.5$\pm$0.4%, reducing the number of support vectors and order of magnitude. Naive Bayes algorithm fails on the original data, reaching only 41.3% accuracy, but in the enhanced space gives 98.1$\pm$0.8%. This data has inherent logical structure that cannot be extracted by Gaussian kernels or Naive Bayes but is captured by decision rules generated by the tree. Transfer of knowledge for meta-learning is possible on an abstract level between different models.

Universal Learning Machines are not restricted to any particular algorithm, trying to extract and transfer new features to new algorithm, enhancing the pool of all features. Support Features Machines (SFM) form an alternative to the SVM approach, using linear discriminant functions defined in such enhanced spaces [73]. For each vector $\mathbf{X}$ there are n input features plus m kernel features $Z_i(\mathbf{X}) = K(\mathbf{X}, \mathbf{X}_i), i = 1..m$. Linear models in the kernel space are as accurate as the kernel SVM, but creating this space explicitly allows for more flexibility. Simple solutions are not overlooked if original features are not discarded. Information granules from other models may be transferred, and mixing kernels of miscellaneous types and with various parameters allows for multiresolution in different parts of the input space.

4 Lessons from Illustrative Calculations

For illustration of the ideas presented in previous sections a few datasets with different characteristics are analyzed below: one artificial binary dataset (Parity), one artificial

set with nominal features (Monks 1), one microarray gene expression data [113], two medical datasets (Cleveland Heart Disease and Wisconsin Breast Cancer data), Spam database derived from texts, Ionosphere data with radar signal patterns. These data can be downloaded from the UCI Machine Learning Repository [114]. A summary of these datasets is presented in Tab. 1. Methods described above have been used for visualization of transformed images of different types of data to determine what kind of structures they create.

Table 1. Summary of used datasets

Title	#Features	#Samples	#Samples per class		Source
Parity_8	8	256	128 C_0	128 C_1	artificial
Monks_1	6	124	62 C_0	62 C_1	[114]
Leukemia	100	72	47 "ALL"	25 "AML"	[113]
Heart	13	270	150 "absence"	120 "presence"	[114]
Wisconsin	10	683	444 "benign"	239 "malignant"	[115]
Spam	57	4601	1813 "spam"	2788 "valid"	[114]
Ionopshere	34	351	224 "Type 1"	126 "Type 2"	[114]

MDS mappings and PCA, ICA, QPC, SVM projections in the original and in the enhanced feature spaces are shown using one-dimensional probability distributions and two-dimensional scatterograms. Analyzing distribution in Figs. 1 – 8 one can determine which classifier has the best bias and will create the simplest model of a given dataset. To check if an optimal choice has been made comparison with classification accuracies for each dataset using various classifiers has been done, in the original as well as in the reduced one and two-dimensional spaces. The following classifiers have been used:

1. Naive Bayesian Classifier (NBC)
2. k-Nearest Neighbors (kNN)
3. Separability Split Value Tree (SSV) [61]
4. Support Vector Machines with Linear Kernel (SVML)
5. Support Vector Machines with Gaussian Kernel (SVMG)

4.1 Parity

High-dimensional parity problem is very difficult for most classification methods. Many papers have been published about special neural network models that solve parity problem. The difficulty is quite clear: linear separation cannot be achieved by simple transformations because this is a k-separable problem (Fig. 1). For n-bit strings it can easily be separated into $n+1$ intervals [47, 101], but learning proper MLP weights to achieve it is very difficult. MDS does not show any interesting structure here, as all vectors from one class have their nearest neighbors from the opposite class. Therefore Gaussian RBF networks or kernel methods based on similarity are not able to extract useful information. PCA and SVM find a very useful projection direction $[1, 1..1]$, but the second direction does not help at all. FDA shows significant overlaps for projection on the first direction.

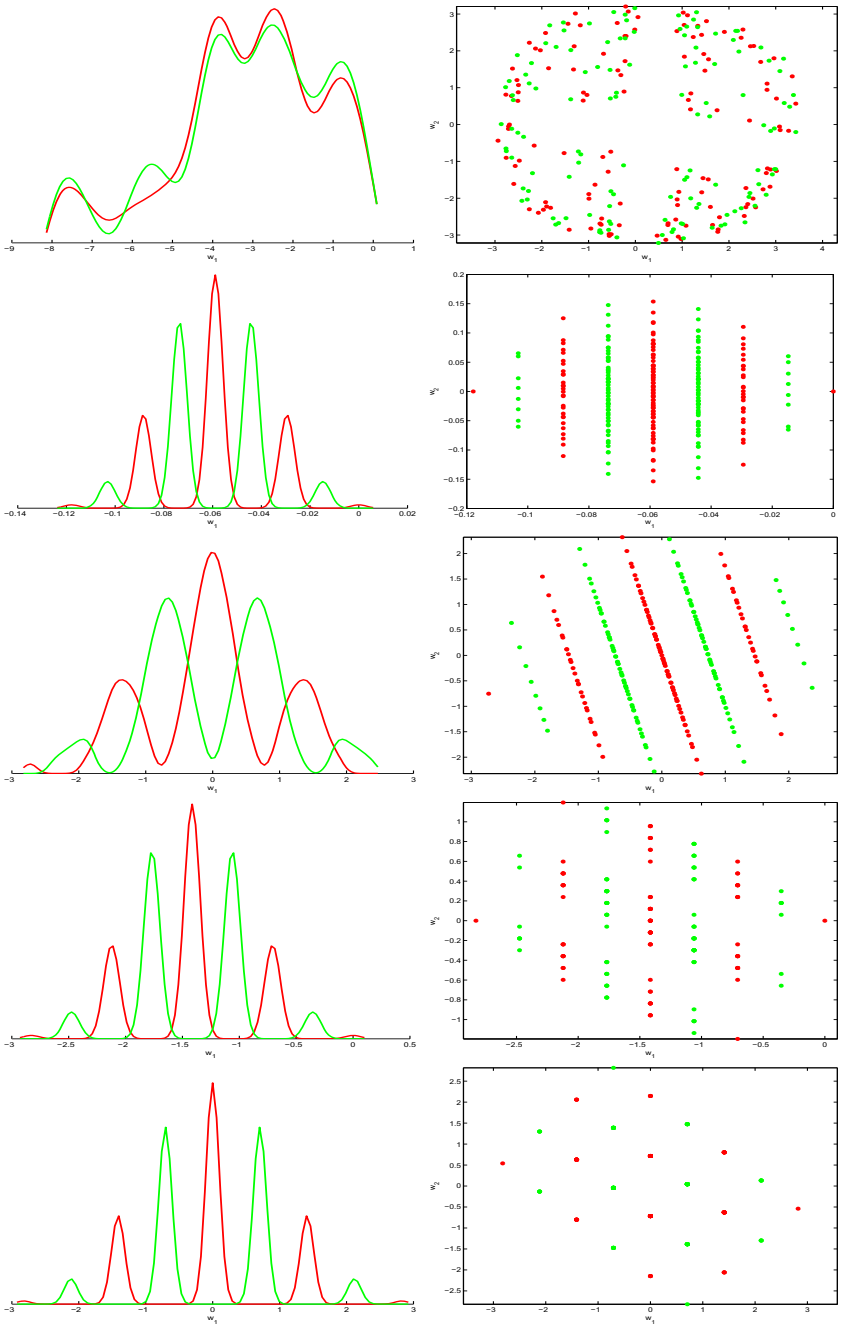


Fig. 1. 8-bit parity dataset, from top to bottom: MDS, PCA, FDA, SVM and QPC.

The QPC index has found two directions that are equally useful. Points that are in small clusters projected on the first direction belong to a large cluster projected on the second direction, giving much better chance for correct classification. In fact any two projections on the longest diagonals are equally useful. This example shows how visualization may point the way towards perfect solution of a difficult problem even in situations when most classifiers fail. Complexity of models created on the original data is high: for example, SVM takes all 256 vectors as support vectors, achieving results around the base rate (50%). Looking at Fig. 1 one can understand the type of non-linearity after projections. Meta-learning should discover that the best classifier to handle such data distribution is:

- any decision tree, after transformation to one dimension by PCA, SVM or two-dimensions by QPC (offering the most stable solution);
- NBC, in one or two-dimensions, combining the two QPC directions for the most robust solution, provided that it will use density estimation based on Gaussian mixtures or other localized kernels rather than a single Gaussian function;
- kNN on the 1D data reduced by PCA, SVM or QPC, with k=1, although it will make a small error for the two extreme points.
- SVM with Gaussian kernel works well on one or two-dimensional data reduced by SVM or QPC projections.

This choice agrees with the results of calculations [100] where the highest accuracy (99.6 ± 1.2) has been obtained by the SSV classifier on the 2D data transformed by SVM or QPC method. Results of NBC and kNN are not worse from the statistical point of view (within one standard deviation). kNN results on the original data with k≤ 15 are always wrong, as all 8 closest neighbors belong to the opposite class. After dimensionality reduction kNN with k=1 is sufficient. Another suggestion is to use radial projections, instead of linear projections. Due to the symmetry of the problem projection on any radial coordinates centered in one of the vertices will show $n + 1$ clusters like projection on the long diagonal.

Visualization in Fig. 1 also suggest that using 2D QPC projected data the nearest neighbor rule may be easily modified: instead of a fixed number of neighbors for vector $\mathbf{X}$, take its projections y_1, y_2 on the two dimensions, and count the number of neighbors $k_i(\epsilon_i)$ in the largest interval $y_i \pm \epsilon_i$ around y_i that contain vectors from a single class only, summing results from both dimensions $k_1(\epsilon_1) + k_2(\epsilon_2)$. This new type of the nearest neighbor rule has not been explored so far.

For problems with inherent complex logic, such as the parity or other Boolean functions [101], a good goal is to create k-separable solutions adding transformations based on linear or radial projections, and then solution of the problem becomes easy.

4.2 Monks_1

Monks_1 is an artificial dataset containing 124 cases, where 62 samples belong to the first class, and the remaining 62 to the second. Each sample is described by 6 attributes. Logical function has been used to create class labels. This is another example of dataset with inherent logical structure, but this time linear k-separability may not be a good

Table 2. Average classification accuracy given by 10-fold crossvalidation test for 8-bit parity problem with reduced features.

	# Features	NBC	kNN	SSV	SVML	SVMG
PCA	1	99.21±1.65	99.20±1.68 (1)	99.21±1.65 (13/7)	39.15±13.47 (256)	99.20±1.68 (256)
PCA	2	99.23±1.62	99.21±1.65 (1)	99.23±1.62 (13/7)	43.36±7.02 (256)	98.83±1.88 (256)
MDS	1	38.35±7.00	43.73±7.44 (4)	47.66±4.69 (1/1)	42.98±5.84 (256)	44.10±8.50 (256)
MDS	2	30.49±13.79	48.46±7.77 (1)	49.20±1.03 (1/1)	43.83±8.72 (256)	43.04±8.91 (256)
FDA	1	75.84±10.63	76.60±7.37 (10)	73.83±6.97 (17/9)	45.73±6.83 (256)	77.76±7.89 (256)
FDA	2	74.56±10.69	99.23±1.62 (1)	96.87±3.54 (35/18)	44.16±5.67 (256)	98.84±1.85 (256)
SVM	1	99.23±1.62	99.61±1.21 (1)	99.23±1.62 (13/7)	54.61±6.36 (256)	99.61±1.21 (9)
SVM	2	99.21±1.65	99.61±1.21 (1)	99.61±1.21 (13/7)	50.29±9.28 (256)	99.61±1.21 (43)
QPC	1	99.20±2.52	99.21±1.65 (1)	99.20±2.52 (13/7)	41.46±9.57 (256)	99.21±1.65 (256)
QPC	2	98.41±2.04	98.44±2.70 (1)	99.61±1.21 (13/7)	43.01±8.21 (256)	98.44±2.70 (24)
	ALL	23.38±6.74	1.16±1.88 (10)	49.2±1.03 (1/1)	31.61±8.31 (256)	16.80±22.76 (256)

goal. In Fig. 2 MDS does not show any structure, and PCA, FDA and SVM projections are also not useful. Only QPC projection shows clear structure of a logical rule. In this case a good goal for learning is to transform the data creating an image in the extended feature space that can be easily understood covering it with logical rules.

Table 3 shows that correct solution is achieved only in the two-dimensional QPC feature space, where only linear SVM fails, all other classifiers can easily handle such data. Decision tree offers the simplest model in this case although in crossvalidation small error has been made.

Table 3. Average classification accuracy given by 10-fold crossvalidation test for Monks_1 problem with reduced features.

	# Features	NBC	kNN	SSV	SVML	SVMG
PCA	1	56.98±14.12	53.97±15.61 (8)	57.94±11.00 (3/2)	63.71±10.68 (98)	58.84±12.08 (102)
PCA	2	54.67±13.93	61.28±17.07 (9)	61.34±11.82 (11/6)	63.71±10.05 (95)	67.17±17.05 (99)
MDS	1	67.94±11.24	69.48±10.83 (8)	68.58±10.44 (3/2)	69.61±11.77 (88)	64.67±10.88 (92)
MDS	2	63.52±16.02	67.75±16.51 (9)	66.98±12.21 (35/18)	64.74±16.52 (103)	62.17±15.47 (104)
FDA	1	72.05±12.03	69.35±8.72 (7)	67.82±9.10 (3/2)	69.93±11.32 (80)	72.37±9.29 (85)
FDA	2	64.48±17.54	69.29±13.70 (9)	68.65±14.74 (3/2)	69.23±10.57 (80)	70.96±10.63 (85)
SVM	1	70.38±10.73	70.12±8.55 (9)	70.32±16.06 (3/2)	71.98±13.14 (78)	72.82±10.20 (77)
SVM	2	71.79±8.78	69.29±10.93 (9)	69.35±9.80 (3/2)	72.75±10.80 (80)	68.65±13.99 (93)
QPC	1	72.56±9.70	81.34±12.49 (3)	82.43±12.22 (47/24)	67.50±13.54 (82)	67.43±17.05 (84)
QPC	2	100±0	100±0 (1)	98.46±3.24 (7/4)	66.92±16.68 (83)	99.16±2.63 (45)
	ALL	69.35±16.54	71.15±12.68 (10)	83.26±14.13 (35/18)	65.38±10.75 (83)	78.20±8.65 (87)

4.3 Leukemia

Leukemia contains microarray gene expressions data for two types of leukemia (ALL and AML), with a total of 47 ALL and 25 AML samples measured with 7129 probes [113]. Visualization and evaluations of this data is based here on the 100 features with the highest FDA ranking index.

This data showed a remarkable separation using both one and two-dimensional QPC, SVM and FDA projections (Fig. 3), showing more interesting data distributions than MDS or PCA. Choosing one of the three linear transformations (for example the QPC),

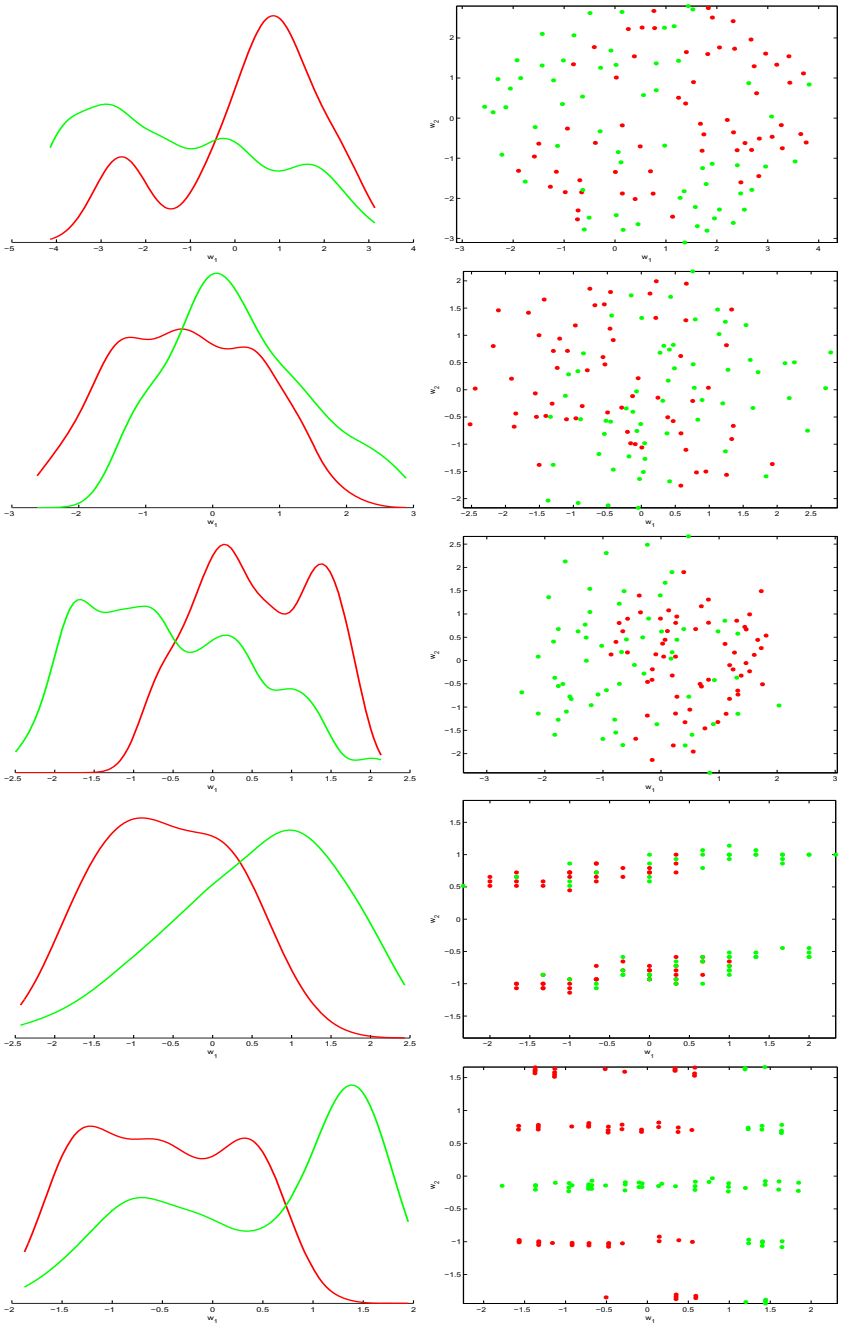


Fig. 2. Monks_1 data set, from top to bottom: MDS, PCA, FDA, SVM and QPC.

and projecting original data to the one-dimensional space, SSV decision tree, kNN, NBC and SVM classifiers, give 100% accuracy in the 10CV tests (Table 4). All these models are very simple, with k=1 for kNN, or decision trees with 3 nodes, or only 2 support vectors for linear SVM. Results on the whole data are worse than on these projected features. Results are slightly worse (1-2 errors) if features are selected and dimensionality reduced separately withing each crossvalidation fold. This shows that although the data is separable it may not be easy to find the best solution on the subset of such data.

In this case maximization of margin is a good guiding principle and dimensionality reduction is a very important factor, combining the activity of many genes into a single profile. As the projection coefficients are linear the importance of each gene in this profile may be easily evaluated. The data is very small and thus one should not expect that all variability of the complex phenomenon has been captured in the training set, therefore it is hard to claim that simple linear solutions should work well also on large samples in this type of data, and they should be preferred in the meta-learning process.

Table 4. Average classification accuracy given by 10-fold crossvalidation test for Leukemia dataset with reduced features.

	# Features	NBC	kNN	SSV	SVML	SVMG
PCA	1	98.57±4.51	98.57±4.51 (2)	95.71±6.90 (7/4)	98.57±4.51 (4)	98.57±4.51 (20)
PCA	2	98.57±4.51	98.57±4.51 (3)	95.81±5.16 (7/4)	97.14±6.02 (4)	97.14±6.02 (22)
MDS	1	92.85±7.52	91.78±7.10 (4)	91.78±14.87 (3/2)	91.78±9.78 (28)	91.78±7.10 (36)
MDS	2	98.57±4.51	97.32±5.66 (8)	95.71±6.90 (7/4)	97.32±5.66 (5)	98.75±3.95 (27)
FDA	1	100±0.00	100±0.00 (1)	100±0.00 (3/2)	100±0.00 (2)	100±0.00 (12)
FDA	2	100±0.00	100±0.00 (1)	100±0.00 (3/2)	100±0.00 (3)	100±0.00 (15)
SVM	1	100±0.00	100±0.00 (1)	100±0.00 (3/2)	100±0.00 (2)	100±0.00 (14)
SVM	2	100±0.00	100±0.00 (1)	100±0.00 (3/2)	100±0.00 (5)	100±0.00 (21)
QPC	1	100±0.00	100±0.00 (1)	100±0.00 (3/2)	100±0.00 (2)	100±0.00 (10)
QPC	2	100±0.00	100±0.00 (1)	100±0.00 (3/2)	100±0.00 (2)	100±0.00 (12)
	ALL	78.28±13.55	98.57±4.51 (2)	90.00±9.64 (5/3)	98.57±4.51 (16)	98.57±4.51 (72)

4.4 Heart and Wisconsin

Heart disease dataset consisting of 270 samples, each described by 13 attributes, 150 cases labeled as “absence”, and 120 as “presence” of heart disease. Wisconsin breast cancer dataset [115] contains samples describing results of biopsies on 699 patients, with 458 biopsies labeled as “benign”, and 241 as “malignant”. Feature 6 has 16 missing values, removing corresponding vectors leaves 683 examples. Both datasets are rather typical examples of medical diagnostic data.

The information contained in the Cleveland Heart training data is not really sufficient to make a perfect diagnosis data (Fig. 4). Best classification results are in this case around 85%, and distributions seem to be similar to overlapping Gaussians. Almost all projections show comparable separation of a significant portion of the data, although looking at probability distributions in one dimension SVM and FDA seem to have a bit of an advantage. In such case strong regularization is advised to improve generalization. For kNN this means that a rather large number of neighbors should be used (in most cases 10, the maximum allowed here, was optimal), for decision trees strong pruning

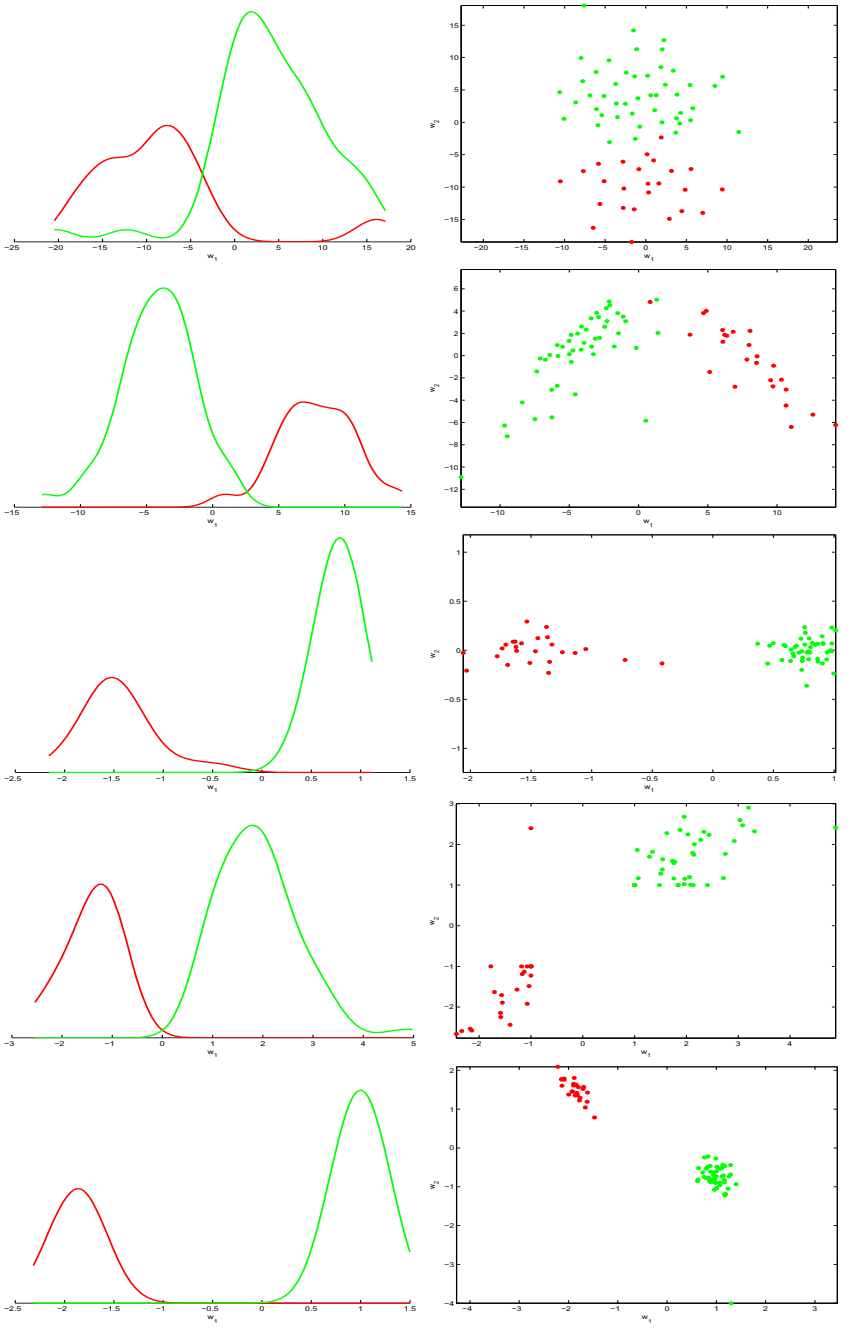


Fig. 3. Leukemia data set, from top to bottom: MDS, PCA, FDA, SVM and QPC.

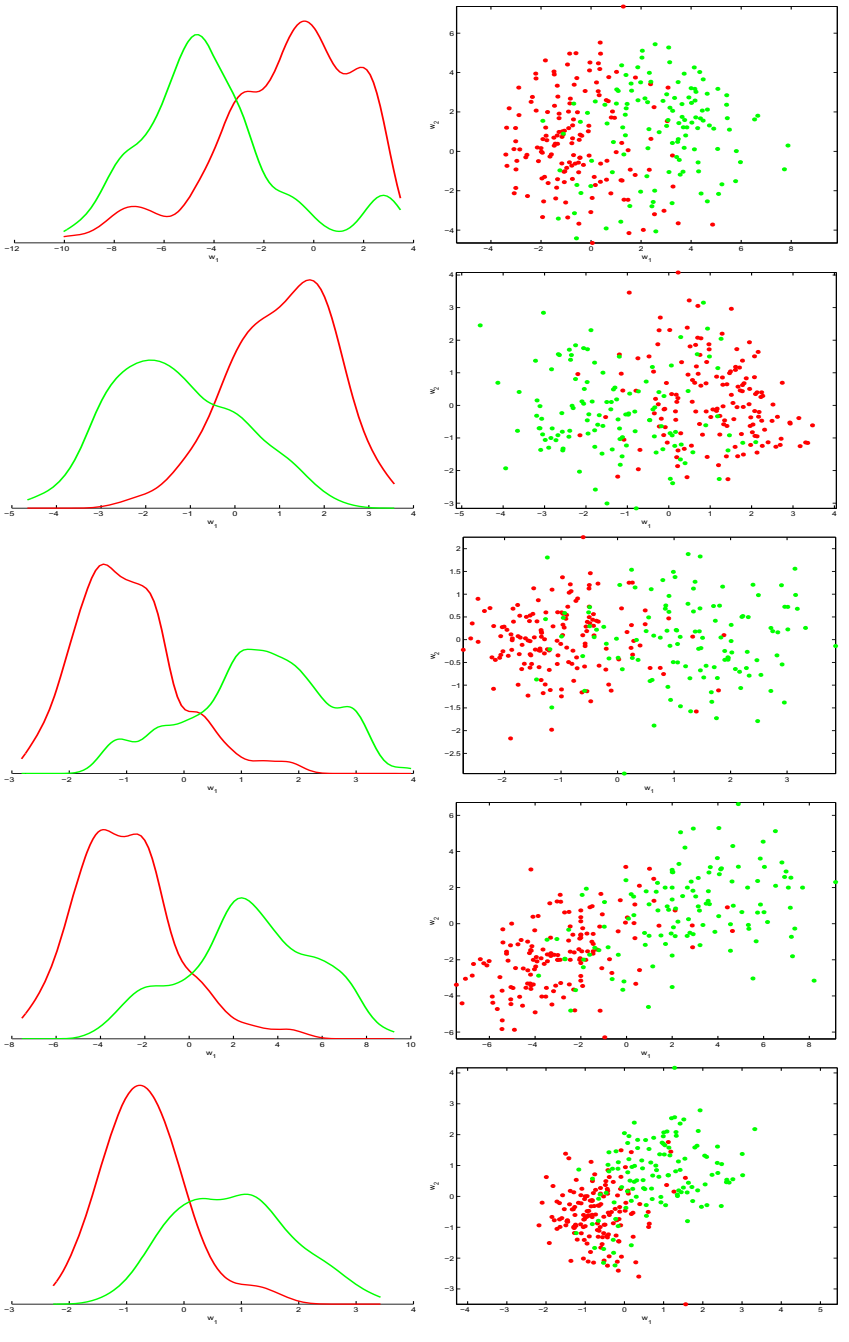


Fig. 4. Heart data set, from top to bottom: MDS, PCA, FDA, SVM and QPC.

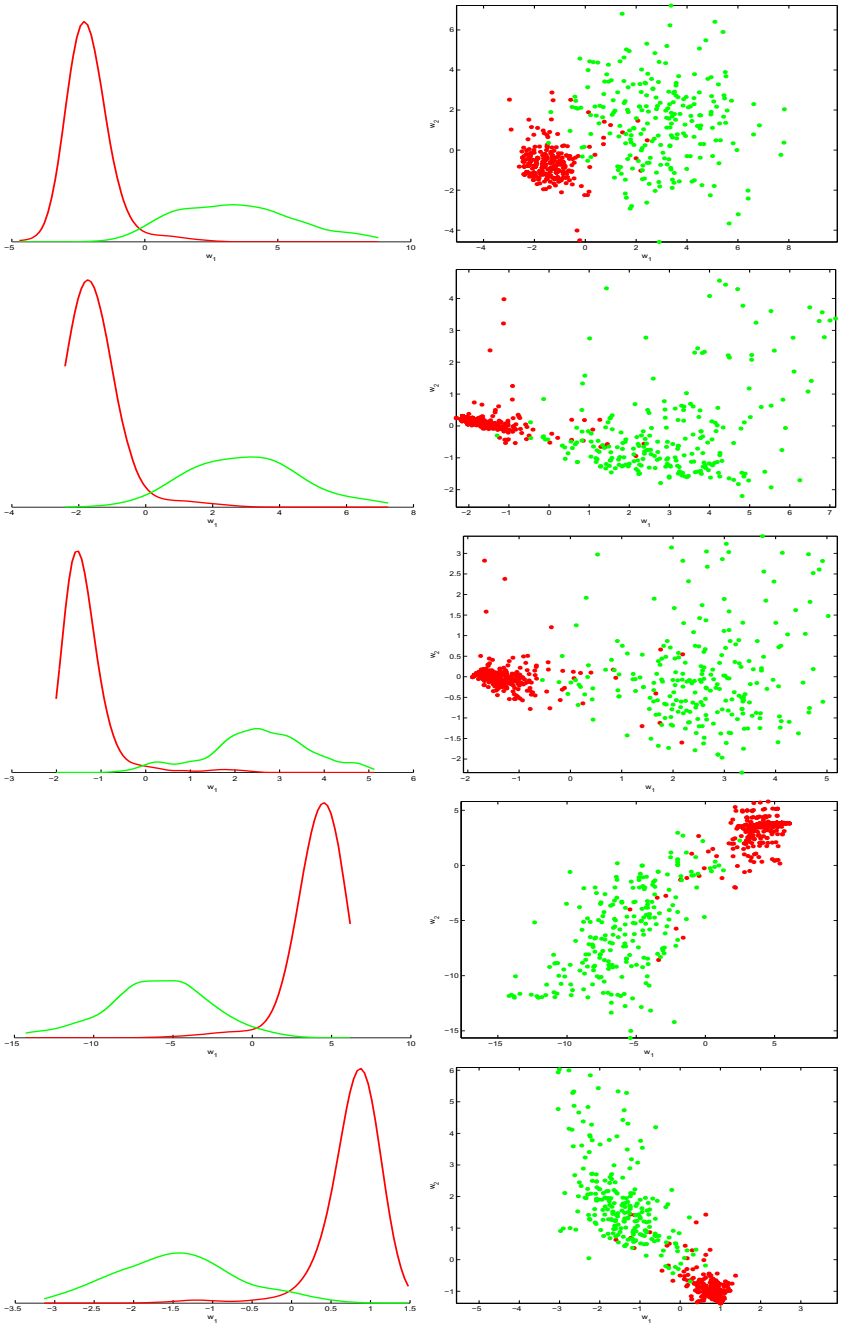


Fig. 5. Wisconsin data set, from top to bottom: MDS, PCA, FDA, SVM and QPC.

(SSV after FDA has only a root node and two leaves), while for SVM rather large value of C parameter and (for Gaussian kernels) large dispersions. The best recommendation for this dataset is to apply the simplest classifier – SSV or linear SVM on FDA projected data. Comparing this recommendation with calculations presented in table 5 confirms that this is the best choice.

The character of the Wisconsin breast cancer dataset is similar to the Cleveland Heart data, although separation of the two classes is much stronger (Fig. 5). Benign cases show high similarity in the MDS mapping and in all considered here linear projections, while malignant cases are much more diverse, perhaps indicating that several types of breast cancer are mixed together. It is quite likely that this data contains several outliers and should really be separable, suggesting that wider margins of classification should be used at the cost of a few errors. All methods give here comparable results, although reduction of dimensionality to two dimensions helps quite a bit to decrease the complexity of the data models. SVM is an exception, achieving essentially the same accuracy and requiring similar number of support vectors for the original and for the reduced data.

Again, the simplest classifier is quite sufficient here, SSV on FDA or QPC projections with a single threshold (a tree with just two leaves), or more complex (about 50 support vectors) SVM model with linear kernel on 2D data reduced by linear projection. One should not expect that much more information can be extracted from this type of data.

Table 5. Average classification accuracy given by 10-fold crossvalidation test for Heart dataset with reduced features.

	# Features	NBC	kNN	SSV	SVML	SVMG
PCA	1	80.74±6.24	75.92±9.44 (10)	79.25±10.64 (3/2)	81.11±8.08 (118)	80.00±9.43 (128)
PCA	2	78.88±10.91	80.74±8.51 (9)	79.62±7.03 (15/8)	82.96±7.02 (113)	80.00±9.99 (125)
MDS	1	75.55±6.80	72.96±7.62 (8)	77.40±6.16 (3/2)	77.03±7.15 (170)	73.70±8.27 (171)
MDS	2	80.74±9.36	80.37±8.19 (6)	81.11±4.76 (3/2)	82.96±6.09 (112)	82.59±7.20 (121)
FDA	1	85.18±9.07	84.81±5.64 (8)	84.07±6.77 (3/2)	85.18±4.61 (92)	85.18±4.61 (106)
FDA	2	84.07±8.01	82.96±6.34 (10)	83.70±6.34 (3/2)	84.81±5.36 (92)	84.81±6.16 (110)
SVM	1	85.92±6.93	82.59±7.81 (9)	83.33±7.25 (3/2)	85.55±5.36 (92)	85.18±4.61 (107)
SVM	2	83.70±5.57	82.96±7.44 (10)	84.81±6.63 (3/2)	85.55±7.69 (92)	84.07±7.20 (131)
QPC	1	81.48±6.53	81.85±8.80 (10)	82.22±5.46 (9/5)	82.59±8.73 (118)	82.59±10.33 (130)
QPC	2	84.44±7.96	85.55±4.76 (10)	83.33±7.25 (13/7)	85.92±5.46 (103)	85.18±4.93 (132)
	ALL	72.22±4.70	79.62±11.61 (9)	81.48±4.61 (7/4)	84.44±5.17 (99)	82.22±5.17 (162)

Table 6. Average classification accuracy given by 10-fold crossvalidation test for Wisconsin dataset with reduced features.

	# Features	NBC	kNN	SSV	SVML	SVMG
PCA	1	97.36±2.27	96.92±1.61 (7)	97.07±1.68 (3/2)	96.78±2.46 (52)	97.36±2.15 (76)
PCA	2	96.18±2.95	96.34±2.69 (7)	97.36±1.92 (3/2)	96.92±2.33 (53)	97.22±2.22 (79)
MDS	1	96.63±1.95	95.60±1.84 (7)	97.07±1.83 (3/2)	95.60±2.59 (54)	95.74±2.45 (86)
MDS	2	95.16±1.70	96.48±2.60 (3)	96.19±2.51 (9/5)	96.92±2.43 (52)	96.63±2.58 (78)
FDA	1	97.07±0.97	97.35±1.93 (5)	96.92±2.34 (3/2)	97.21±1.88 (52)	97.65±1.86 (70)
FDA	2	95.46±1.89	96.77±1.51 (9)	96.93±1.86 (11/6)	96.77±2.65 (51)	97.07±2.06 (74)
SVM	1	95.90±1.64	97.22±1.98 (9)	97.22±1.99 (3/2)	97.22±1.26 (46)	96.93±1.73 (69)
SVM	2	97.21±1.89	97.36±3.51 (10)	97.22±1.73 (3/2)	96.92±2.88 (47)	96.92±3.28 (86)
QPC	1	96.33±3.12	97.22±1.74 (7)	96.91±2.01 (3/2)	96.34±2.78 (62)	97.07±1.82 (84)
QPC	2	97.21±2.44	96.62±1.84 (7)	96.33±2.32 (3/2)	96.62±1.40 (54)	96.33±1.87 (107)
	ALL	95.46±2.77	96.34±2.52 (7)	95.60±3.30 (7/4)	96.63±2.68 (50)	96.63±2.59 (93)

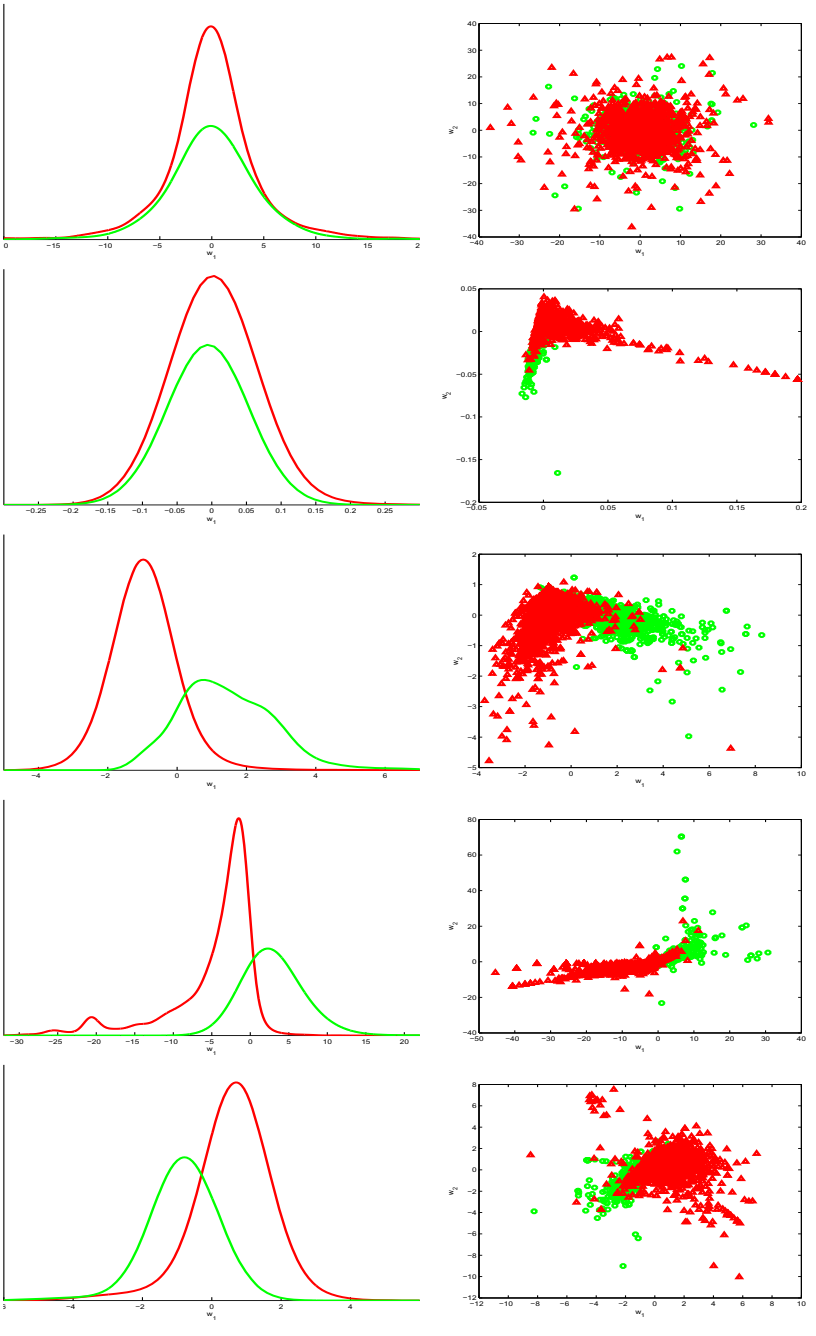


Fig. 6. Spambase data set, from top to bottom: MDS, PCA, FDA, SVM and QPC.

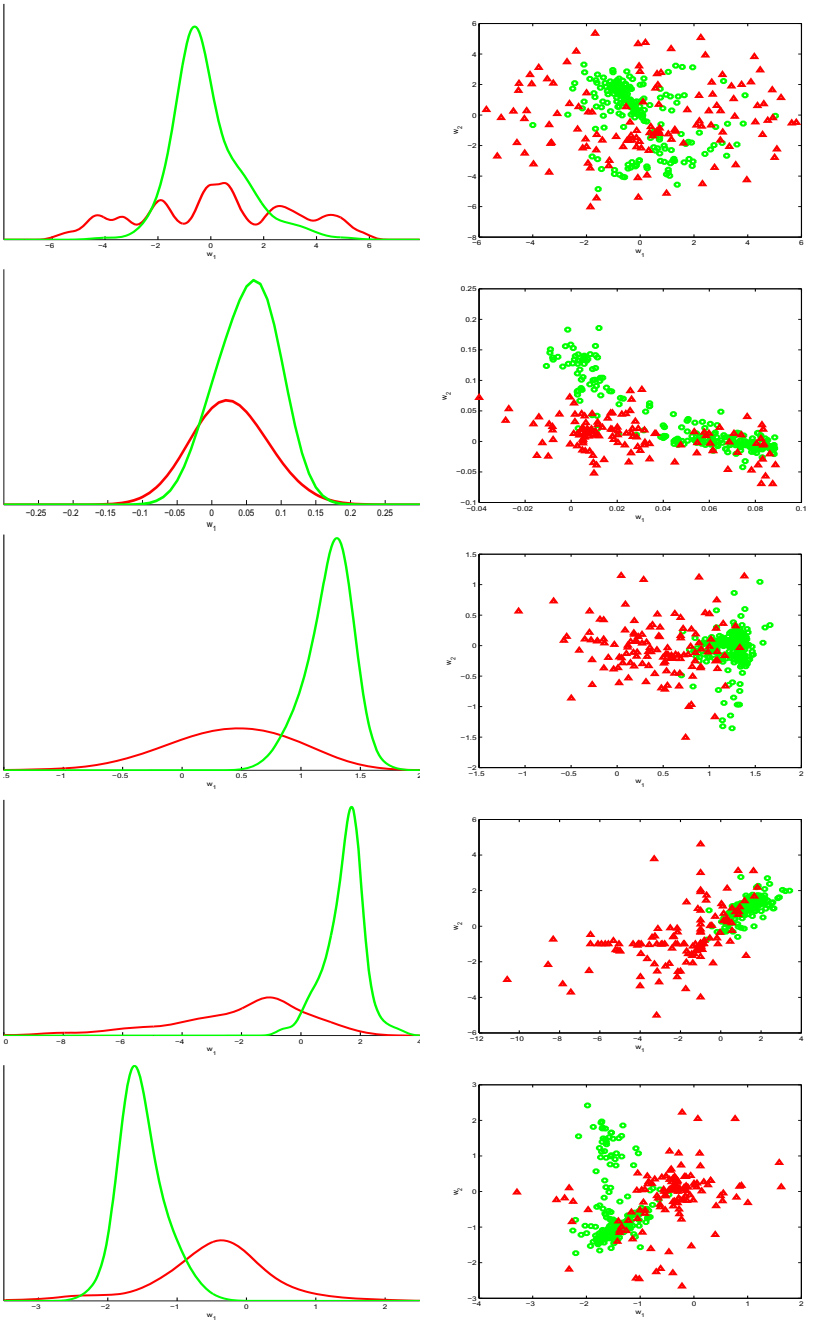


Fig. 7. Ionosphere data set, from top to bottom: MDS, PCA, FDA, SVM and QPC.

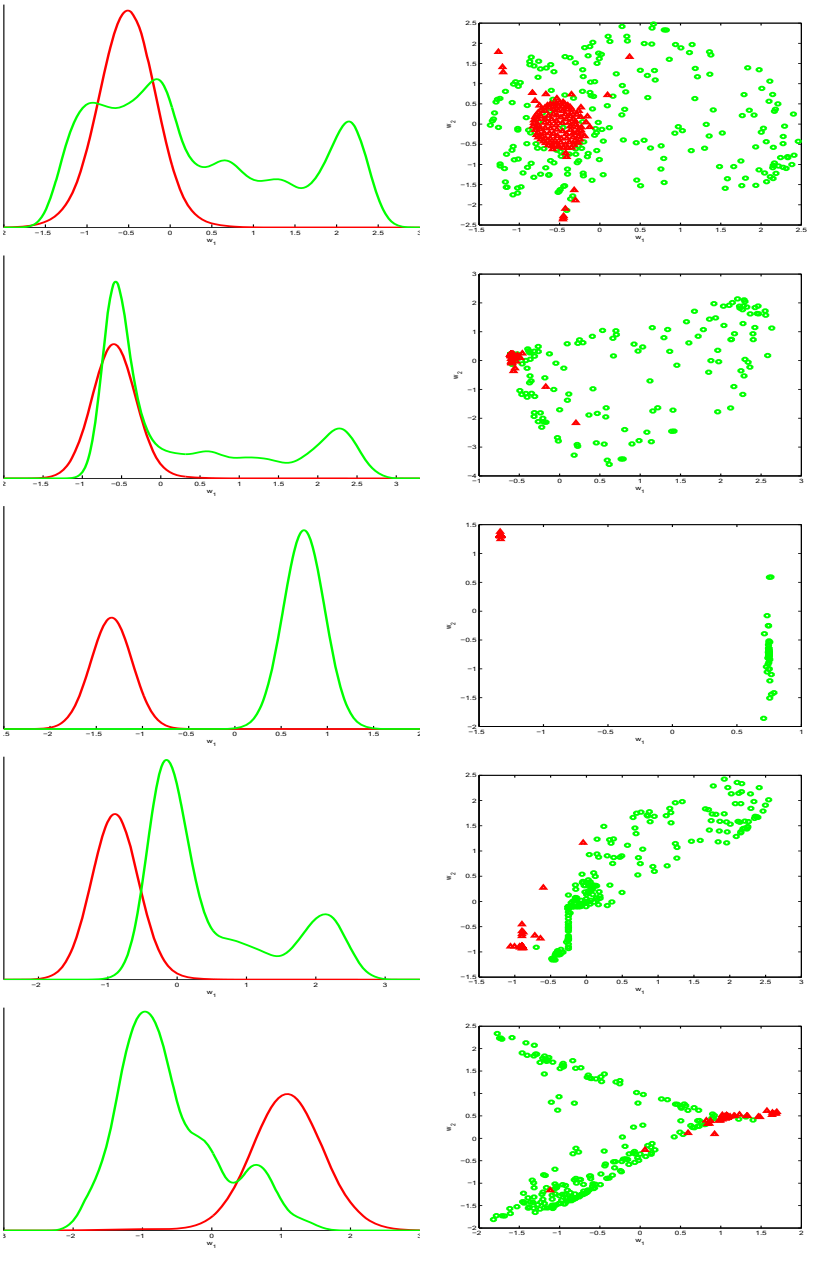


Fig. 8. Ionosphere data set in the kernel space, from top to bottom: MDS, PCA, FDA, SVM and QPC.

4.5 Spambase

Spam dataset is derived from a collection of 4601 emails described by 57 attributes. 1813 of these emails are real spam and 2788 are work related and personal emails. From Fig. 6 it is clear that MDS and PCA are not of much use in this problem, at least in a low number of dimensions. In case of PCA the second dimension helps a bit to separate data that belongs to different classes, but MDS is completely lost. FDA, QPC and linear SVM in 1-dimensional space look very similar, however adding second dimension shows some advantage of SVM. It is clear that in this case low-dimensional visualization is not able to capture much information about data distribution. Best results may be expected from large margin classifiers, linear SVM gives $93.1 \pm 0.7\%$ (C=1), and similar results from the Gaussian kernel SVM.

4.6 Ionosphere

Ionosphere dataset has 351 records, with different patterns of radar signals reflected from the ionosphere, 224 patterns in Class 1 and 126 in Class 2. First feature is binary, second is always zero, and the remaining 32 are continuous.

In this case (Fig. 7) MDS and all projections do not show much structure. To illustrate the effect of kernel transformation original features are replaced by Gaussian kernels with $\sigma = 1$, thus increasing the dimensionality of the space to 351. Now (Fig. 8) MDS shows focused cluster of signals from one class on the background of the second class, and projection methods show quite clear separation, with FDA showing surprisingly large separation. This shows that the data after the kernel transformation became linearly separable. The FDA solution has been found on the whole dataset, and it may not be possible to find such perfect solution in crossvalidation even if transductive learning is used.

5 Discussion and Conclusions

The holy grail of machine learning, computational intelligence, pattern recognition and related fields is to create intelligent algorithms that will automatically configure themselves and lead to discovery of all interesting models for arbitrary data. All learning algorithms may be presented as sequences of transformations. Current data mining systems contain many transformations that may be composed in billions of ways, therefore it is impossible to test all promising combinations of preprocessing, feature selection, learning algorithms, optimization procedures, and post-processing. Meta-level knowledge is needed to automatize this process, help to understand how efficient learning may proceed by search in the space of all transformation.

The main focus of this paper has been on generation of transformations, categorization of types of features using geometrical perspective, creation of new features, learning from other data models by feature transfer, understanding what kind of data distributions are created in the extended features space, and finding decision algorithms with proper bias for such data. Systematic explorations of features of growing complexity enables discovery of simple models that more sophisticated learning systems

will miss. Feature constructors described here go beyond linear combinations provided by PCA or ICA algorithms. In particular, kernel-based features offer an attractive alternative to current kernel-based SVM approaches, offering multiresolution and adaptive regularization possibilities. Several new types of features have been introduced, and their role analyzed from geometrical perspective. Mixing different kernels and using different types of features gives much more flexibility to create decision borders or approximate probability densities. Adding specific support features facilitates knowledge discovery. Good generalization is achieved by searching for large pure clusters of vectors that may be uncovered by specific information filters. Homogeneous algorithms create small clusters that are not reliable, but with many different filters the same vectors may be mapped in many ways to large clusters. This approach significantly extends our previous similarity-based framework [21] putting even higher demands on organization of intelligent search mechanism in the space of all possible transformations (see [103] and this volume).

Constructing diverse information filters leads to interesting views on the data, showing non-linear structures in the data that – if noticed – may be easy to handle with specific transformations. Systems that actively sample data, trying to "see it" through their filters, are more flexible than classifiers working in fixed input spaces. Once sufficient information is generated reliable categorization of data structures may be achieved. Although the final goal of learning is to discover interesting models of data, more attention should be paid to the intermediate representations, the image of data after transformation. Instead of hiding information in kernels and sophisticated optimization techniques features based on kernels and projection techniques make this explicit. Finding useful views on the data by constructing proper information filters is the best way to practical applications that automatically create all interesting data models for a given data. Objects may have diverse and complex structures, and different categories may be identified in different feature spaces derived by such filters and transformations. Once the structure of data image that emerges in the enhanced space is recognized, it may then be handled by a decision module specializing in handling specific type of nonlinearity. Instead of linear separability much easier intermediate goal is set, to create clear non-linear data image of a specific type.

Some benchmark problems have been found rather trivial, and have been solved with a single binary feature, one constrained nominal feature, or one new feature constructed as a projection on a line connecting means of two classes. Analysis of images, multimedia streams or biosequences will require even more sophisticated ways of constructing higher-order features. Thus meta-learning package should have general mechanisms controlling search, based on understanding of the type of transformations that may be useful for specific data, principles of knowledge transfer and goals of learning that go beyond separability of data, and modules with transformations specific for each field.

Neurocognitive informatics draws inspirations from neurobiological processes responsible for learning and forms a good basis for meta-learning ideas. So far only a few general inspirations have been used in computational intelligence, like for example threshold neurons organized in networks that perform parallel distributed processing. Even with our limited understanding of the brain many more inspirations may be drawn and used in practical learning and object recognition algorithms. Parallel

interacting streams of complementary information with hierarchical organization [6] may be linked to multiple information filters that generate new higher-order features. Accumulating noisy stimulus information from multiple parallel streams until reliable response is made [7] may be linked to confidence level of classifiers based on information from multiple features of different type. Kernel methods may be relevant for category learning in biological systems [116], although in standard formulations of SVMs it is not at all obvious. Explicit use of kernel features understood as similarity estimation to objects categorized using high-order features may correspond to various functions of microcircuits that are present in cortical minicolumns, extending the simple liquid state machine picture [5]. With great diversity of microcircuits a lot of information is generated, and relevant chunks are used as features by simple Hebbian learning of weights in the output layer. In such model plasticity of the basic feature detectors receiving the incoming signals may be quite low, yet fast correlation-based learning is still possible.

References

[1] Walker, S.F.: A brief history of connectionism and its psychological implications. In: Clark, A., Lutz, R. (eds.) Connectionism in Context, pp. 123–144. Springer, Berlin (1992)

[2] Anderson, J.A., Rosenfeld, E.: Neurocomputing - foundations of research. MIT Press, Cambridge (1988)

[3] Gerstner, W., Kistler, W.M.: Spiking Neuron Models. Single Neurons, Populations, Plasticity. Cambridge University Press, Cambridge (2002)

[4] Maass, W., Markram, H.: Theory of the computational function of microcircuit dynamics. In: Grillner, S., Graybiel, A.M. (eds.) Microcircuits. The Interface between Neurons and Global Brain Function, pp. 371–392. MIT Press, Cambridge (2006)

[5] Maass, W., Natschläger, T., Markram, H.: Real-time computing without stable states: A new framework for neural computation based on perturbations. Neural computation 14, 2531–2560 (2002)

[6] Grossberg, S.: The complementary brain: Unifying brain dynamics and modularity. Trends in Cognitive Sciences 4, 233–246 (2000)

[7] Smith, P.L., Ratcliff, R.: Psychology and neurobiology of simple decisions. Trends in Neurosciences 27, 161–168 (2004)

[8] Jaeger, H., Maass, W., Principe, J.: Introduction to the special issue on echo state networks and liquid state machines. Neural Networks 20, 287–289 (2007)

[9] Bengio, Y.: Learning deep architectures for AI. Foundations and Trends in Machine Learning 2, 1–127 (2009)

[10] Hinton, G., Osindero, S., Teh, Y.: A fast learning algorithm for deep belief nets. Neural Computation 18, 381–414 (2006)

[11] Schölkopf, B., Smola, A.J.: Learning with Kernels. Support Vector Machines, Regularization, Optimization, and Beyond. MIT Press, Cambridge (2001)

[12] Schapire, R., Singer, Y.: Improved boosting algorithms using confidence-rated predictions. Machine Learning 37, 297–336 (1999)

[13] Kuncheva, L.I.: Combining Pattern Classifiers. Methods and Algorithms. J. Wiley & Sons, New York (2004)

[14] Duch, W., Itert, L.: Competent undemocratic committees. In: Rutkowski, L., Kacprzyk, J. (eds.) Neural Networks and Soft Computing, pp. 412–417. Springer, Heidelberg (2002)

[15] Brazdil, P., Giraud-Carrier, C., Soares, C., Vilalta, R.: Metalearning: Applications to Data Mining. Cognitive Technologies. Springer, Heidelberg (2009)

[16] Newell, A.: Unified theories of cognition. Harvard Univ. Press, Cambridge (1990)
[17] Duda, R.O., Hart, P.E., Stork, D.G.: Patter Classification. J. Wiley & Sons, New York (2001)
[18] Vilalta, R., Giraud-Carrier, C.G., Brazdil, P., Soares, C.: Using meta-learning to support data mining. International Journal of Computer Science and Applications 1(1), 31–45 (2004)
[19] Michie, D., Spiegelhalter, D.J., Taylor, C.C.: Machine learning, neural and statistical classification. Elis Horwood, London (1994)
[20] Duch, W., Grudziński, K.: Meta-learning: searching in the model space. In: Proceedings of the International Conference on Neural Information Processing, Shanghai, pp. 235–240 (2001)
[21] Duch, W., Grudziński, K.: Meta-learning via search combined with parameter optimization. In: Rutkowski, L., Kacprzyk, J. (eds.) Advances in Soft Computing, pp. 13–22. Springer, New York (2002)
[22] Giraud-Carrier, C., Vilalta, R., Brazdil, P.: Introduction to the special issue on meta-learning. Machine Learning 54, 194–197 (2004)
[23] Sutton, C., McCullum, A.: An introduction to conditional random fields (2010)
[24] Duch, W., Matykiewicz, P., Pestian, J.: Neurolinguistic approach to natural language processing with applications to medical text analysis. Neural Networks 21(10), 1500–1510 (2008)
[25] Pedrycz, W.: Knowledge-Based Clustering: From Data to Information Granules. Wiley Interscience, Hoboken (2005)
[26] Michalski, R.S. (ed.): Multistrategy Learning. Kluwer Academic Publishers, Dordrecht (1993)
[27] Duch, W., Jankowski, N.: Survey of neural transfer functions. Neural Computing Surveys 2, 163–213 (1999)
[28] Duch, W., Jankowski, N.: Transfer functions: hidden possibilities for better neural networks. In: 9th European Symposium on Artificial Neural Networks, pp. 81–94. De-facto publications, Brusells (2001)
[29] Jankowski, N., Duch, W.: Optimal transfer function neural networks. In: 9th European Symposium on Artificial Neural Networks, pp. 101–106. De-facto publications, Bruges (2001)
[30] Duch, W., Adamczak, R., Diercksen, G.: Constructive density estimation network based on several different separable transfer functions. In: 9th European Symposium on Artificial Neural Networks, Bruges, Belgium (April 2001)
[31] Duch, W., Grąbczewski, K.: Heterogeneous adaptive systems. In: IEEE World Congress on Computational Intelligence, pp. 524–529. IEEE Press, Honolulu (2002)
[32] Grąbczewski, K., Duch, W.: Heterogeneous forests of decision trees. In: Dorronsoro, J.R. (ed.) ICANN 2002. LNCS, vol. 2415, pp. 504–509. Springer, Heidelberg (2002)
[33] Wieczorek, T., Blachnik, M., Duch, W.: Influence of probability estimation parameters on stability of accuracy in prototype rules using heterogeneous distance functions. Artificial Intelligence Studies 2, 71–78 (2005)
[34] Wieczorek, T., Blachnik, M., Duch, W.: Heterogeneous distance functions for prototype rules: influence of parameters on probability estimation. International Journal of Artificial Intelligence Studies 1 (2006)
[35] Ullman, S.: High-level vision: Object recognition and visual cognition. MIT Press, Cambridge (1996)
[36] Haykin, S.: Neural Networks - A Comprehensive Foundation. Maxwell MacMillian Int., New York (1994)
[37] Cristianini, N., Shawe-Taylor, J.: An Introduction to Support Vector Machines and other Kernel-Based Learning Methods. Cambridge University Press, Cambridge (2000)

[38] Duch, W.: Similarity based methods: a general framework for classification, approximation and association. Control and Cybernetics 29, 937–968 (2000)
[39] Duch, W., Adamczak, R., Diercksen, G.H.F.: Classification, association and pattern completion using neural similarity based methods. Applied Mathematics and Computer Science 10, 101–120 (2000)
[40] Sonnenburg, S., Raetsch, G., Schaefer, C., Schoelkopf, B.: Large scale multiple kernel learning. Journal of Machine Learning Research 7, 1531–1565 (2006)
[41] Duch, W., Adamczak, R., Grąbczewski, K.: A new methodology of extraction, optimization and application of crisp and fuzzy logical rules. IEEE Transactions on Neural Networks 12, 277–306 (2001)
[42] Duch, W., Setiono, R., Zurada, J.: Computational intelligence methods for understanding of data. Proceedings of the IEEE 92(5), 771–805 (2004)
[43] Duch, W.: Towards comprehensive foundations of computational intelligence. In: Duch, W., Mandziuk, J. (eds.) Challenges for Computational Intelligence, vol. 63, pp. 261–316. Springer, Heidelberg (2007)
[44] Baggenstoss, P.M.: The pdf projection theorem and the class-specific method. IEEE Transactions on Signal Processing 51, 668–672 (2003)
[45] Bengio, Y., Delalleau, O., Roux, L.N.: The curse of highly variable functions for local kernel machines. Advances in Neural Information Processing Systems 18, 107–114 (2006)
[46] Bengio, Y., Monperrus, M., Larochelle, H.: Non-local estimation of manifold structure. Neural Computation 18, 2509–2528 (2006)
[47] Duch, W.: K-separability. In: Kollias, S.D., Stafylopatis, A., Duch, W., Oja, E. (eds.) ICANN 2006. LNCS, vol. 4131, pp. 188–197. Springer, Heidelberg (2006)
[48] Kosko, B.: Neural Networks and Fuzzy Systems. Prentice-Hall International, Englewood Cliffs (1992)
[49] Duch, W.: Filter methods. In: Guyon, I., Gunn, S., Nikravesh, M., Zadeh, L. (eds.) Feature extraction, foundations and applications, pp. 89–118. Physica Verlag/Springer, Heidelberg (2006)
[50] Duch, W., Adamczak, R., Hayashi, Y.: Eliminators and classifiers. In: Lee, S.Y. (ed.) 7th International Conference on Neural Information Processing (ICONIP), Dae-jong, Korea, pp. 1029–1034 (2000)
[51] Holte, R.C.: Very simple classification rules perform well on most commonly used datasets. Machine Learning 11, 63–91 (1993)
[52] Grochowski, M., Duch, W.: Projection Pursuit Constructive Neural Networks Based on Quality of Projected Clusters. In: Kůrková, V., Neruda, R., Koutník, J. (eds.) ICANN 2008, Part II. LNCS, vol. 5164, pp. 754–762. Springer, Heidelberg (2008)
[53] Jordan, M.I., Sejnowski, T.J.: Graphical Models. Foundations of Neural Computation. MIT Press, Cambridge (2001)
[54] Jones, C., Sibson, R.: What is projection pursuit. Journal of the Royal Statistical Society A 150, 1–36 (1987)
[55] Friedman, J.: Exploratory projection pursuit. Journal of the American Statistical Association 82, 249–266 (1987)
[56] Webb, A.R.: Statistical Pattern Recognition. J. Wiley & Sons, Chichester (2002)
[57] Hastie, T., Tibshirani, J., Friedman, J.: The Elements of Statistical Learning. Springer, Heidelberg (2001)
[58] Hyvärinen, A., Karhunen, J., Oja, E.: Independent Component Analysis. Wiley & Sons, New York (2001)
[59] Cichocki, A., Amari, S.: Adaptive Blind Signal and Image Processing. Learning Algorithms and Applications. J. Wiley & Sons, New York (2002)
[60] Pękalska, E., Duin, R.: The dissimilarity representation for pattern recognition: foundations and applications. World Scientific, Singapore (2005)

[61] Grąbczewski, K., Duch, W.: The separability of split value criterion. In: Proceedings of the 5th Conf. on Neural Networks and Soft Computing, pp. 201–208. Polish Neural Network Society, Zakopane (2000)
[62] Torkkola, K.: Feature extraction by non-parametric mutual information maximization. Journal of Machine Learning Research 3, 1415–1438 (2003)
[63] Tebbens, J.D., Schlesinger, P.: Improving implementation of linear discriminant analysis for the small sample size problem. Computational Statistics & Data Analysis 52, 423–437 (2007)
[64] Gorsuch, R.L.: Factor Analysis. Erlbaum, Hillsdale (1983)
[65] Gifi, A.: Nonlinear Multivariate Analysis. Wiley, Boston (1990)
[66] Srivastava, A., Liu, X.: Tools for application-driven linear dimension reduction. Neurocomputing 67, 136–160 (2005)
[67] Kordos, M., Duch, W.: Variable Step Search MLP Training Method. International Journal of Information Technology and Intelligent Computing 1, 45–56 (2006)
[68] Bengio, Y., Delalleau, O., Roux, N.L.: The curse of dimensionality for local kernel machines. Technical Report Technical Report 1258, Dṕartement d'informatique et recherche opérationnelle, Universite de Montreal (2005)
[69] Tsang, I.W., Kwok, J.T., Cheung, P.M.: Core vector machines: Fast svm training on very large data sets. Journal of Machine Learning Research 6, 363–392 (2005)
[70] Chapelle, O.: Training a support vector machine in the primal. Neural Computation 19, 1155–1178 (2007)
[71] Tipping, M.E.: Sparse Bayesian Learning and the Relevance Vector Machine. Journal of Machine Learning Research 1, 211–244 (2001)
[72] Lee, Y., Mangasarian, O.L.: Ssvm: A smooth support vector machine for classification. Computational Optimization and Applications 20, 5–22 (2001)
[73] Maszczyk, T., Duch, W.: Support feature machines: Support vectors are not enough. In: World Congress on Computational Intelligence, pp. 3852–3859. IEEE Press, Los Alamitos (2010)
[74] Pao, Y.H.: Adaptive Pattern Recognition and Neural Networks. Addison-Wesley, Reading, MA (1989)
[75] Macias, J.A., Sierra, A., Corbacho, F.: Evolution of functional link networks. IEEE Transactions on Evolutionary Computation 5, 54–65 (2001)
[76] Leung, H., Haykin, S.: Detection and estimation using an adaptive rational function filters. IEEE Transactions on Signal Processing 12, 3365–3376 (1994)
[77] Duch, W., Adamczak, R., Diercksen, G.H.F.: Neural networks in non-euclidean spaces. Neural Processing Letters 10, 201–210 (1999)
[78] Duch, W., Adamczak, R., Diercksen, G.H.F.: Distance-based multilayer perceptrons. In: Mohammadian, M. (ed.) International Conference on Computational Intelligence for Modelling Control and Automation, pp. 75–80. IOS Press, Amsterdam (1999)
[79] Duch, W., Diercksen, G.H.F.: Feature space mapping as a universal adaptive system. Computer Physics Communications 87, 341–371 (1995)
[80] Cox, T., Cox, M.: Multidimensional Scaling, 2nd edn. Chapman and Hall, Boca Raton (2001)
[81] Thompson, R.: The Brain. The Neuroscience Primer. W.H. Freeman and Co, New York (1993)
[82] Breiman, L.: Bias-variance, regularization, instability and stabilization. In: Bishop, C.M. (ed.) Neural Networks and Machine Learning, pp. 27–56. Springer, Heidelberg (1998)
[83] Avnimelech, R., Intrator, N.: Boosted mixture of experts: An ensemble learning scheme. Neural Computation 11, 483–497 (1999)
[84] Bauer, E., Kohavi, R.: An empirical comparison of voting classification algorithms: bagging, boosting and variants. Machine learning 36, 105–142 (1999)

[85] Maclin, R.: Boosting classifiers regionally. In: Proc. 15th National Conference on Artificial Intelligence, Madison, WI, pp. 700–705 (1998),
[86] Duch, W., Itert, L.: Committees of undemocratic competent models. In: Rutkowski, L., Kacprzyk, J. (eds.) Proc. of Int. Conf. on Artificial Neural Networks (ICANN), Istanbul, pp. 33–36 (2003)
[87] Giacinto, G., Roli, F.: Dynamic classifier selection based on multiple classifier behaviour. Pattern Recognition 34, 179–181 (2001)
[88] Bakker, B., Heskes, T.: Task clustering and gating for bayesian multitask learning. Journal of Machine Learning Research 4, 83–99 (2003)
[89] Smyth, P., Wolpert, D.: Linearly combining density estimators via stacking. Machine Learning 36, 59–83 (1999)
[90] Wolpert, D.: Stacked generalization. Neural Networks 5, 241–259 (1992)
[91] Schwenker, F., Kestler, H., Palm, G.: Three learning phases for radial-basis-function networks. Neural Networks 14, 439–458 (2001)
[92] Duch, W., Maszczyk, T.: Almost random projection machine. In: Alippi, C., Polycarpou, M., Panayiotou, C., Ellinas, G. (eds.) ICANN 2009. LNCS, vol. 5768, pp. 789–798. Springer, Heidelberg (2009)
[93] Rutkowski, L.: Flexible Neuro-Fuzzy Systems. Kluwer Academic Publishers, Dordrecht (2004)
[94] Roweis, S., Saul, L.: Nonlinear dimensionality reduction by locally linear embedding. Science 290(5500), 2323–2326 (2000)
[95] Kégl, B., Krzyzak, A.: Piecewise linear skeletonization using principal curves. IEEE Transactions on Pattern Analysis and Machine Intelligence 24, 59–74 (2002)
[96] Shoujue, W., Jiangliang, L.: Geometrical learning, descriptive geometry, and biomimetic pattern recognition. Neurocomputing 67, 9–28 (2005)
[97] Huang, G., Chen, L., Siew, C.: Universal approximation using incremental constructive feedforward networks with random hidden nodes. IEEE Transactions on Neural Networks 17, 879–892 (2006)
[98] Miettinen, K.: Nonlinear Multiobjective Optimization. Kluwer Academic Publishers, Dordrecht (1999)
[99] Maszczyk, T., Duch, W.: Support vector machines for visualization and dimensionality reduction. In: Kůrková, V., Neruda, R., Koutník, J. (eds.) ICANN 2008, Part I. LNCS, vol. 5163, pp. 346–356. Springer, Heidelberg (2008)
[100] Maszczyk, T., Grochowski, M., Duch, W.: Discovering Data Structures using Meta-learning, Visualization and Constructive Neural Networks. In: Koronacki, J., Ras, Z.W., Wierzchon, S.T., Kacprzyk, J. (eds.). Advances in Machine Learning II. SCI, vol. 262, pp. 467–484. Springer, Heidelberg (2010)
[101] Grochowski, M., Duch, W.: Learning Highly Non-separable Boolean Functions Using Constructive Feedforward Neural Network. In: de Sá, J.M., Alexandre, L.A., Duch, W., Mandic, D.P. (eds.) ICANN 2007. LNCS, vol. 4668, pp. 180–189. Springer, Heidelberg (2007)
[102] Grabczewski, K., Jankowski, N.: Versatile and efficient meta-learning architecture: Knowledge representation and management in computational intelligence. In: IEEE Symposium on Computational Intelligence in Data Mining, pp. 51–58. IEEE Press, New York (2007)
[103] Grąbczewski, K., Jankowski, N.: Meta-learning with machine generators and complexity controlled exploration. In: Rutkowski, L., Tadeusiewicz, R., Zadeh, L.A., Zurada, J.M. (eds.) ICAISC 2008. LNCS (LNAI), vol. 5097, pp. 545–555. Springer, Heidelberg (2008)
[104] Abu-Mostafa, Y.S.: Learning from hints in neural networks. Journal of Complexity 6, 192–198 (1989)

[105] Thrun, S.: Is learning the n-th thing any easier than learning the first? In: Touretzky, D.S., Mozer, M.C., Hasselmo, M.E. (eds.) Advances in Neural Information Processing Systems, vol. 8, pp. 640–646. MIT Press, Cambridge (1996)
[106] Caruana, R., Pratt, L., Thrun, S.: Multitask learning. Machine Learning 28, 41 (1997)
[107] Wu, P., Dietterich, T.G.: Improving svm accuracy by training on auxiliary data sources. In: ICML (2004)
[108] Daumé III, H., Marcu, D.: Domain adaptation for statistical classifiers. Journal of Artificial Intelligence Research 26, 101–126 (2006)
[109] Raina, R., Ng, A.Y., Koller, D.: Constructing informative priors using transfer learning. In: Proceedings of the 23rd International Conference on Machine Learning, pp. 713–720 (2006)
[110] Raina, R., Battle, A., Lee, H., Packer, B., Ng, A.Y.: Self-taught learning: Transfer learning from unlabeled data. In: ICML 2007: Proceedings of the 24th International Conference on Machine learning (2007)
[111] Dai, W., Jin, O., Xue, G.R., Yang, Q., Yu, Y.: Eigentransfer: a unified framework for transfer learning. In: ICML, p. 25 (2009)
[112] Duch, W., Maszczyk, T.: Universal learning machines. In: Leung, C.S., Lee, M., Chan, J.H. (eds.) ICONIP 2009. LNCS, vol. 5864, pp. 206–215. Springer, Heidelberg (2009)
[113] Golub, T.R.: Molecular classification of cancer: Class discovery and class prediction by gene expression monitoring. Science 286, 531–537 (1999)
[114] Asuncion, A., Newman, D.: UCI machine learning repository (2007), `http://www.ics.uci.edu/~mlearn/MLRepository.html`
[115] Wolberg, W.H., Mangasarian, O.: Multisurface method of pattern separation for medical diagnosis applied to breast cytology. Proceedings of the National Academy of Sciences, U.S.A. 87, 9193–9196 (1990)
[116] Jäkel, F., Schölkopf, B., Wichmann, F.A.: Does cognitive science need kernels? Trends in Cognitive Sciences 13(9), 381–388 (2009)

Author Index