AbsoluteMode Engine ™ API

AbsoluteMode.AGIengine.online

Basic Function:

If you have software making numerous routine sub-judgments leading to a conclusive final judgment in a single coherent case, call the API to *check the validity of alternative such judgments*, and/or to *generate alternative such judgments* if it is not possible to presently do so within your software.

A "judgment" is your software's typical *decided* output. A "judgment" has a legal ring to it but all domains are supported. Judgments need to be submitted to the API as grammatically valid English sentences, but supporting data can be random keywords. The response of the API, which runs the AbsoluteMode Engine, will be *consistent with your prior cases based on training*. An entire case, including all sub-judgments and a final judgment, can be submitted at once for analysis.

The hallmark of increased software generality is its ability to handle a wider variety of cases. The main purpose of the API is to provide a "generality channel" for your software to use in the event that it is unable to test possible alternative judgments, or it simply cannot determine the output at all. This might happen if some combination has not yet been coded or trained within your software, or perhaps the software has become brittle and is simply not being further modified, or perhaps the end user inputs an unexpected ad hoc prospective judgment, etc etc.

Input to API:

1) Routine judgments made by your software taken as true for training
2) Alternative, although unproven, passed judgments made by your software to be verified
3) Completely unknown judgments labeled "?" for which a realistic output judgment is desired

Output from API:

1) A decision of "realistic" or "unrealistic" of the entire case (all sub- and final judgments inclusive)
2) Derived judgments, if the input judgment had been labeled "?" (#3 above)

Basic Steps to Utilize the API:

Step 1
Determine the subject nature of the judgments. For the API to be useful, your software should be structured so that one or more sub-judgment(s) results in a conclusive single final judgment, all within a single discrete case. The sub-judgments should be grouped by distinct subjects. In a simple example, the prominent subjects on a cooking app could be ingredients and cooking instructions. Suppose the app makes sub-judgments about ingredients and cooking instructions. After these are input on the app in some case at hand, the app makes a final judgment about the potential flavor of the recipe, based on two sub-judgments about ingredients and cooking instructions. In this case there are two subjects that lead to a final judgment about the two subjects.

Step 2
Submit routine judgments as English propositions using the simple API described on the following pages. Your software should already be making such decisions – which could be very simple. In our recipe app, perhaps a sub-judgment made by your software is “The ingredients are low in carbohydrates.” You might have to do some mild reformatting to convert to a valid English sentence, however the benefit is that you can display a full sentence for your users and the server keeps a clear log of judgments made in sentences. There are also some other keywords you should include in the call which are described in the API below.

Step 3
Call the server in real time if your software needs to check the validity of any alternative sub-judgments or final judgments within a discrete case, and/or to generate alternative such judgments if it is not possible to presently do within your software. Perhaps our recipe app is limited and could not determine if the ingredients and cooking instructions yield a spicy dish. In that case, the server could be called for a validity check of the final judgment “This dish is spicy.” If so, the server responds with “realistic,” and if not, “unrealistic.” The decision is based on training data.

Benefits:

Generality increase with new capability to decide alternative judgments
A judgment by definition can often *realistically* be other than what it is. Perhaps you need to know if specific alternative, unforeseen, hypothetical judgments are *realistic given past behavior*.

Generality increase with new capability to determine unknown judgments
Judgments might be presently completely undecidable in your software for a variety of reasons – maybe nobody coded or trained ***this*** *particular combination* – but you need *realistic* judgments anyway based upon history.

Generality increase with new capability of recombining "ideas" in novel ways
Since the input judgments are grouped by subject (except the final judgment which is based on the subjects), the server builds virtual "ideas" about subjects which can be recombined in novel combinations by the calling software resulting in novel final judgments. Like all machine learning, the output depends a lot on the training data. The overall objective of this API is to allow a user to recombine usually routine judgments about its usual domain subjects in unforseen ways, real time.

Partially explainable output
For sub-judgments a symbolic term chain is returned. The final judgment is reached without a term chain but by contextual properties.

Full sentences saved in log
Entire cases are stored with judgments in grammatically correct English for later analysis and training.

Minimal Concepts Needed to Use the API:

There are only a few key concepts the developer needs to remember when using the server's API. Basically, the server is available in real time if the developer's software is stuck – that is the objective of all this. The developer utilizing the API needs to create judgments made by the developer's software at run time using English sentences, in the sequence such judgments are made, and pass them to the server. These judgments, along with each judgment's relevant keywords, form the fundamental data requirements of the server. If, after training offline, the developer's software is *unable to reach a judgment*, the server responds with an appropriate judgment based on precedent established with training calls.

1) System 1 vs System 2

The overall structure of the API together with calling user software is roughly analogous to the distinction between so-called "System 1" and "System 2" thinking that Daniel Kahneman made famous in his book "*Thinking, Fast & Slow."* Simply put, system 1 judgments are taken by the server as true. System 2 judgments are taken by the server to be uncertain. System 1 thinking is characterized as being fast, typical, and not subject to deliberation, whereas system 2 thinking is slower and requires deliberation, reflection, consideration of alternatives, and reasoning. We are assuming the calling software is unable to do the equivalent of system 2 reasoning needed for novel cases if such cases are undecidable in the present calling software. We can think of the calling software being limited to system 1. The server is doing the system 2 processing.

2) Judgment

A "judgment" is your software's typical *decided* output (eg "the recipe is spicy"). The server keeps track of the judgments that the calling software makes in the order made. The server accepts judgments singly, one at a time. The judgments need to be declared as grammatically correct English sentences (eg., "the ingredients are low in carbohydrates"). The sentences can be very simple. There is one judgment required for each system 1 or system 2 call. A system 1 call is always taken by the server as true, valid, and realistic. System 2 judgments trigger an evaluation by the server. The "?" symbol can be used in a system 2 call if the judgment is completely unknown and needs to be determined by the server. The calling software can switch in real time from system 1 to system 2.

3) Judgment Levels

There are three levels of judgments: 0, 1, and 2 in ascending precedence. Above, the level 1 judgment was described as a "sub-judgment" for clarifying its relation to a final judgment, a level 2 judgment. However, it is easier to use three level indicators. The lowest level 0 judgment can only be a system 1 judgment (see above), and is always just taken as-is by the server, ie., taken as true. Immediately following one or more level 0 judgments there must be one and only one level 1 judgment (the "sub-judgment" above) which must be a judgment based on the preceding one or more level 0 judgments: the level 1 judgment is the conclusion reached given the level 0 judgments made by the user's calling software. In our example above, we had two level 0 judgments – one about the ingredients and one

about the cooking instructions. (The judgments can be caveman simple, eg: “The cooking instructions are short” or whatever works, but it must be a valid English sentence.) The level 1 judgment could be, eg: “The recipe is a spicy main dish.” There is a hierarchy with this level scheme, but there are only three levels to remember. Whereas this scheme of only three judgment levels would seem inadequate to capture the totality of a software’s judgments, the goal is to “smash” – essentially – all submitted judgments in a single case into a level 2 final judgment. The server is neurosymbolically based, and is only partly using discrete judgments.

4) Essence

All judgments can have an *essence*. This is separate from the judgment but in the same judgment call transaction to the server (ie., is a separate field in JSON, see below). The developer should include in essence whatever values the developer’s software determined relevant to reaching the judgment at hand. In our recipe example, perhaps the essence in the ingredients judgment is: “low in fat, has curry leaves.” Note: values passed as “essence” need not be complete sentences. Random keywords are OK. However, it is a good idea to try to keep the values in the same general sequence if possible.

5) Appearance

Appearance should be the usual, relevant input to the developer’s software. Thus, this is what *appears* to the developer’s software as its input. In our example, perhaps appearance is “liquid, green.” Importantly, appearance is only applicable to level 0 judgments.

6) Case

The developer should have an application amenable to division into discrete cases. Each case will typically consist of sets of multiple level 0 judgments concluding in a level 1 judgment. A single level 2 judgment may be reached last. The level 2 judgment should be considered the final judgment for the case submitted to the server. A level 2 judgment is not required. In our recipe example, a single case was the recipe, however there was no level 2 judgment.

7) Subject

A set of level 0 judgments followed by a single level 1 judgment must have the same subject: all judgments must have the same subject in a valid English sentence. However, the final judgment, level 2, can be a different subject which is relevant to the totality of the case.

8) Realistic

A judgment is deemed realistic if it is consistent with past decisions known to be correct.

API:

1) /createcase

When the developer's software starts a new case, this call is to be made, and it returns a case number for the ensuing calls. The case number must be included with all subsequent calls.

2) /sys1_proposition

This transaction is to be used for judgments made by the developer's software that are to be taken as true. All levels (0 → 2) are acceptable. It is not necessary to specify the level as the server will do this automatically based upon a strict order scheme. If a sys1_proposition contains *appearance* (see above) it is classed as level 0; if omitted it is classed as level 1 or level 2. These propositions can include an *essence* (see above). Multiple level 0 judgments can precede one and only one level 1 proposition so long as they share the same subject. As many subgroups of level 0 with level 1 judgments as needed for the case are valid, each set having potentially different subjects, but there can only be a single level 2 judgment at the end of the case with any subject serving as the case's final judgment.

3) /sys2_proposition

This transaction is to be used for judgments made by the developer's software that are to be taken as uncertain and is only applicable to level 1 and level 2 judgments. Level 0 judgments are always taken as true. If the "?" character is submitted *as* the judgment, the server will return a relevant judgment that is applicable when a sys2_realistic transaction is submitted with the same case number. However, a potential judgment can also be passed instead of the "?" character in the sys2_proposition. In this case the server determines if the passed judgment is realistic or not (as described herein). Importantly, with this transaction "desired" terms can be passed. The desired terms must be in the processed judgment for the case to be "realistic" (see below).

4) /sys2_realistic

Basically, the server responds with either "REALISTIC" OR "UNREALISTIC" given all of the system 1 and system 2 transactions that made up the submitted case. If the "?" character is submitted *as any* judgment (level 1 or 2), the server will return derived judgments that are applicable. In addition, if any sys2_proposition contained "desired" terms, if and only if these terms are found in each applicable sys2_proposition will "realistic" be returned. If the case does not have all the "desired" terms, it is "unrealistic." The "desired" terms apply on to a judgment (not to essence or appearance).

5) /retract_that

Prior to the case being marked for training, a retract_that will remove the most recently added judgment to the case given (system 1 or 2). So effectively the case can be rebuilt as needed and resubmitted.

6) /traincase

This transaction marks the case for offline training. Until a case is marked for training, it can be modified and resubmitted using sys2_realistic.

How the Components Work

There are two main components necessary: an online REST API described above and a batch training job to be run at regular intervals. Presently all server software is written in Python, utilizing Flask/mod_wsge on Apache httpd.

REST API PROCESSING

First, the REST API is described. For all the transactions, the transaction is checked to make sure the REST submitter has the valid credentials, and if applicable the case number must be included.

1) If the user submits a /createcase call then a case is initialized in a MySQL database, and the case number is returned.

2) If a /sys1_proposition is submitted, it is added to the case in the MySQL database. Checks are made to ensure proper ordering. If a transaction contains “appearance” fields, it is deemed to be level 0 in all cases. A single level 1 judgment can only be inserted after level 0 cases with the same subject. Only one level 2 judgment is acceptable and it must follow a level 1 judgment. The server replies as needed (eg., “judgment added”, condition 200, “error out of order,” etc).

3) If a /sys2_proposition is submitted, it is added to the case in the MySQL database. Checks are made to ensure proper ordering. It cannot contain “appearance” fields (see above). A single level 1, system 2 judgment can only be inserted after level 0, system 1 cases with the same subject. Only one level 2 judgment is acceptable and it must follow a level 1 judgment. The server does not evalute a case until a sys2_realistic call is made below.

4) /sys2_realistic will trigger a call to the AI engine (see below). After the engine processes the whole case, the Apache server responds with either “realistic” or “unrealistic,” meaning that the submitted case is consistent with past cases or it isn’t. If any new judgments are derived these are returned as well. Note that a case can only be determined as “realistic” if all “desired” terms appear in respective judgments. Desired terms can be included in any sys2_proposition call. They are not required.

5) If a /retract_that is submitted, the most recent judgment in the MySQL database for the case number passed is deleted.

6) If a /traincase call is made, the MySQL database is updated with the training flag set to true (initialized false).

AI engine

If one were to summarize the design strategy in short form it would be “old AI symbolic and modern AI deep learning chaining in parallel.” Thus, this is a neuro-symbolic strategy.

When a sys2_realistic REST transaction is submitted, this means that the developer’s software is unable to proceed on its own. The engine inputs the entire case including all system 1 and system 2

judgments that have been posted using respective REST transactions as above (with any keyword absolutely required in a specific system 2 judgment specified as “desired” and any associated essence and appearance data as described above included if needed). The basic function of the engine is to evaluate the totality of judgments submitted, system 1 and 2, as a single, coherent, and unified case and try to determine whether or not the case as a whole is essentially “realistic.”

The basic structure of AI engine is similar to – yet has evolved from – that described in the book “Approximation Zero,” published 2013 by Michael Archbold, ISBN 0615882811.

The book (2013) describes a three part ontology: the “What,” “How,” and “Toolbox.” The “what” in the book basically consists of propositions that can act as a focal point in, and include, an overall idea. An idea is roughly an integration of multiple propositions in a case into a single understood totality, the whole works being the what. The “how,” is basically the means needed to create the “what.” Fundamentally the “how” is composed of a “megafunction” and an “oracle.” The megafunction (so called because it should be able to act – ultimately – on whatever judgment the oracle commands) inputs judgments from the oracle and outputs an understanding for the oracle, and the oracle inputs understandings from the megafunction and outputs a single selected judgment then input to the megafunction for execution. (An “understanding” was never adequately defined in the book, however it roughly meant whatever is needed by the oracle to choose one judgment from a set of possible judgments. This was unsatisfying to the author but left open.) The megafunction and oracle work in tandem to produce the what. The what is (again) roughly an understood idea with a propositional focal point. There is always a propositional focal point within an overall idea.

As described in the book, the process starts with a “main idea” being issued to the AI – ie., the user submits a case for evaluation. Upon receiving a main idea the oracle and megafunction interact until the “desired ends” of the main idea are “objectively realized.” These last two quoted expressions are drawn from the Hegelian ontology. The “toolbox” mentioned above consists of discrete, abstract rules which are utilized by the megafunction in generating the understanding for the oracle.

The present code (2022) is structured in an object oriented fashion using the book’s general scheme. The engine is composed of an oracle and a megafunction (both Python functions). The “how” as sketched above became Python executable code and the “what” sketched above became Python data structures. The “toolbox” consists of distillations of normalized, compressed, and trained models based upon all prior cases.

When a sys2_realistic transaction is processed, the oracle (again, part of the engine) initially chooses the last judgment that was submitted for the case (typically with /sys2_proposition) which is assumed to be the main idea. The entire case is then evaluated in turn, starting with the main idea, until the oracle decides that the main idea is realistic or unrealistic based upon the totality of the case.

Every judgment submitted in a case must be shown to be realistic to the oracle. A system 1 judgment, as mentioned above, is simply taken to be true (thus also “realistic”) by the engine. However any system 2 judgment must be evaluated.

The core of the system 2 judgment evaluation starts with the classic Aristotelian, categorical syllogism. The classic syllogistic example is:

Socrates is a man
all men are mortal
Socrates is mortal

The second premise, here a universal, is typically induced from an adequate number of particular cases.

A 21st century version of the classic syllogism, however, serves as the core for the engine's evaluation.

Given some input judgment, the engine tries to resolve a judgment to what is fundamentally, at bottom, a syllogism, and if and when the syllogism chains terms (Socrates → man → mortal), the judgment is deemed realistic. If the "Socrates is mortal" case were submitted to the engine for training, "Socrates is a man" would be the first judgment, followed by the "Socrates is mortal" judgment, but the server would learn "all men are mortal." There is no need to submit the "all men are mortal" major premise. Basically the "all men are mortal" premise learning is the object of deep learning, the benefit being that a great number of variations are possible (although obviously the context of mortality is rigid).

In addition, more than one minor premise is allowed. So in the above example, suppose other statements are made about Socrates: Socrates is a philosopher, Socrates is a stonemason, and so forth. The "all men are mortal" premise, the major, is really an ideal location to use deep learning instead of an old AI term chain. Omit the "all men are mortal" premise entirely. Mentally insert a CNN (convolutional neural network) there instead to do the same thing: the major premise is induced from particular prior cases, and serves as the bridge to the conclusion. The conclusion could become "Socrates is mortal, is skilled with his hands, and is wise" if multiple minor premise judgments are submitted as above.

So, we sort of crush a mass of syllogisms into one: call it a "mega-syllogism." This structure gives a great deal of latitude to any syllogism. Now we have one syllogistic structure that can handle a wide variety of propositions about Socrates.

If we take all possible statements about Socrates we can ever funnel through the CNN we can call the resulting functionality an "idea" – if we define an idea as an integration of possible or actual propositions; thus, in a manner of speaking an "idea" about Socrates is learned.

In addition to the CNN functionality, there is also an old fashioned term chain. In the above example, in order for "Socrates is skilled with bricks" to be realistic, the server will have to see a proposition "Socrates is a stonemason" directly followed by "Socrates is skilled with bricks" and learn the connection of "being a stonemason means skilled with bricks" (a de facto major premise) using offline training.

In this way the inference is held together **both** with a CNN and with old fashioned term chaining. Thus for a system 2 judgment to be deemed realistic it has to pass this two-part test.

A significant problem with all of AI is known as the combinatorial explosion. The problem is that there exists a seemingly infinite number of outcomes to most real world situations (save those tightly constrained) making it practically impossible to pre-code every eventuality. It is the *context* of the situation which narrows down the space of possible outcomes, making for a hopefully tractable search.

Just processing a plain syllogism, as above, is not enough to create a tractable context if the intent is generality. So, to create added context, the above mentioned fields of “appearance” and “essence” in the API are utilized by the engine to narrow down and refine the context. These fields are also based on the Hegelian ontology. These two fields are typically present in level 0 judgments which is always taken as true (system 1). The engine considers level 0 judgments to be de facto minor premises in a syllogism.

The engine gains context from appearance (what the input looks like) and essence (significant considerations used to make a judgment).

This added information narrows the context; however, this is taken a step further. In the Hegelian ontology something is not understood by what it is per se, but what it is like and unlike. A “reflection” of an apple might contain a like set of apples of similar color, and an unlike set of apples dissimilar color. Holding both sets in the mind in a single reflection concurrently gives an understanding of some apple in particular by comparison to what *this apple* is like and unlike. In psychology, this is basically known as a concept built from exemplars.

The intent of the present design is reduce the combinatorial explosion by creating such reflections alongside each judgment, the reflections adding crucial context to the syllogism, and feed these into the CNN for pattern matching along with the usual premises as illustrated above.

Thus when a judgment is processed by the engine, if there exists passed term keywords (which need not be complete sentences) describing appearance and/or essence, these keywords are used to build reflections that contain likenesses (and unlikenesses) to some other, known aspect of appearance and/or essence.

So, to take our Socrates example: if keywords describing Socrates’ appearance and/or essence relevant to the case at hand are passed concurrently with the judgment “Socrates is a man,” these are used to build reflections not unlike exemplar concepts based on contextually relevant essence and appearance keywords that have accompanied the judgment. This enhances context.

The reflections constructed are typical NLP embeddings. The likeness of terms needed for reflection to add context, as described above, are based on the well-known Word2Vec scheme. Using Word2Vec training, essence and appearance terms that regularly appear together are assigned vectors close to one another in vector space.

If an input judgment contains keyword terms in essence and/or appearance, some of these are selected by the engine for reflection. Not all passed terms are chosen for reflection – only those terms (in appearance and essence) that seem to be the most relevant given the context are chosen for reflection. The engine tries to favor terms that are most closely associated and relevant with the judgments that have been passed. The judgment is the source of contextual information. In our Socrates example, the engine would start with “Socrates, man, mortal” terms, then it would review past cases to ascertain which essence and appearance terms seem to occur the most. If certain appearance terms seem to occur a lot in past cases, given “Socrates, man, mortal,” and the same are in the present case, these are chosen for reflection.

However, if a more particular judgment is passed, such as “Socrates is a man lacking sandals” then the engine selects terms from essence and appearance that seem to be relevant to “lacking sandals” as well, the decision made by examining prior similar cases.

Once the engine selects the most contextually applicable terms, a reflection is built. The reflection is composed of about half a dozen (a variable parameter) embeddings of terms. The reflected embeddings are for those terms most like the selected applicable term(s) given Word2Vec training based on all prior cases. In this fashion we create a substructure analogous to an exemplar concept, or an Hegelian reflection.

There is a deliberate attempt to build something akin to mental concepts with reflections, although the similarity is merely analogous. A reflection of “apple” could give “orange, lemon, grapefruit,” or even “lettuce,” etc. Most people are familiar with the way Word2Vec associates terms. It depends upon the embedding created by Word2Vec training of prior trained system 1 and system 2 cases. It is hoped that the reflection described herein is akin to a mental concept, and thus closer to the psychological realm. It is the *structure* of mind that we are always striving toward. These reflections are intended, while admittedly somewhat crude, to narrow the context, to be tractable, and fed into a CNN to ameliorate the combinatorial explosion.

When we chain a syllogism, in summary, we are showing that the submitted judgment is realistic given prior cases. We are using multiple means – an old fashioned term chain along with a CNN that relies on reflected embeddings of the most contextually relevant terms. Simply put, if the term chain and the CNN succeed in showing that the premises support the judgment (the conclusion), and if “desired” terms are in the conclusion (if included), the judgment is realistic, and that is returned by the engine to the API caller.

There is only a single level 2 judgment allowed per case. The structure of the syllogism described above, symbolic and CNN processing, is also imposed on the level 2 judgment. The difference in processing for the engine is that the minor premises are all the level 1 judgments. The result is that the level 2 judgment binds together all of the level 1 judgments, thus achieving something akin to a single integration of the case. Thus level 1 judgments use level 0 judgments as minor premises in a syllogistic scheme, and the level 2 judgments use level 1 judgments accordingly. There is a hierarchy, but it isn’t much.

It is the job of the level 2 “final judgment” to reconcile all subordinate judgments in the overall context.

If the “?” character is submitted as a level 1 or level 2 judgment, the engine will return “realistic” along with the derived judgment or “unrealistic” depending on the outcome.

The design of the engine is such that the oracle selects a judgment and then the engine shifts control to the megafunction. The megafunction executes concepts that act on the data structures and objects. A concept, for example, could create a reflection for a particular term as discussed above. The oracle will try to show that each submitted system 2 judgment in the case is realistic. To do this it may need to branch. For example, to resolve a level 2 judgment, it needs to first resolve each related level 1 judgment.

Control shifts back and forth between the oracle and megafunction until a decision is reached – if the case is realistic or not realistic.

BATCH (OFFLINE) TRAINING

All submitted judgments are saved for training and as a log.

There are steps to build the Word2Vec embeddings, create the CNN's model, and create the old AI style term chain (a Python dictionary).

The “Mode of the Absolute” of Hegel

The engine is called the AbsoluteMode Engine, so named for Hegel. It’s impossible to really say with certainty what Hegel really meant by the “mode of the Absolute” in his massive “Science of Logic” since his writing was notoriously obfuscating.

A “mode” seems to be an instance of a mind – to start simply (like Spinoza described as part of a whole reality of one integration into God). At this stage of Hegel’s ontology, we’ve reached the union of his inner “essence” held by a mind and the outer reality of objects presented by “appearance.”

If we take some thought in a mind, it has an essentiality. If we then consider the world external to mind, delivered by its “appearance,” we have a linkage to the “actual” through a substrate. The actual can only exist as a union of inner and outer (essence and appearance respectively). It takes both. But importantly, Hegel probably meant that a single essence inflitrates all inner and outer and gives us actual, the entirety a single essence.

The absolute mode (kind of an extended mind) seems to mean the substrate necessary for linking inner essence and outer appearance. The absolute is that which is able to combine the inner, outer, and result in a union, the actual. Nothing is actual unless there is both an inner essence and outer existence. There is a substrate necessary to bridge the divide: the mode of the absolute.

Think of the absolute as akin to an absolute number which can take on more than one actual value. But *the* absolute is like an absolute concept, not an absolute number, which can itself take on more than one concept, in fact any concept possible, and joins *essence and appearance in a mode*. The *mode of the absolute is then the basis for anything actual.*