OVERVIEW & EXAMPLE

In short, the API works as follows. During typical execution the user's calling software posts to the AbsoluteMode Engine, herein called server, using REST a limited version of its usual proceedings and decisions: data and judgment for its cases as they occur in sequence.

The server gradually learns patterns of the calling software's behavior. If the calling software then, subsequently and after server batch training has been completed, encounters a situation for which it has either a potential but uncertain solution or no solution at all (effectively an undecidable state), it calls the server for "deliberation," and the server analyzes the case at hand and responds with either "realistic" or "unrealistic" – along with a propositional judgment based upon the present case in light of prior cases.

The key to the solution is that the server bases its decision not by using the same literal code-wise interpretation the calling software performs, but rather on whether or not the calling software's behavior is determined to be *essentially realistic or unrealistic based upon its past behavior*.

This is then analogous to system 1 vs. system 2 thinking: the user's software makes the usual, fast system 1 decisions, but system 2 "deliberation" by the server is required for unusual cases. The server is not so much inferring new knowledge per se as it is just solving a different, not-coded variation based upon existing knowledge derived from routine cases. Subsequently, the server's decision can be used for training to increase generality for future calls. The server saves the case history in a natural language format for an audit trail.

EXAMPLE

As a very simple example, suppose you, the developer, have an app which is used for recipes. The app inputs ingredients and cooking instructions, and then describes the qualities of the resulting dish. Suppose then the app has an ingredients screen, cooking screen, and finally a result screen.

Clearly, an explosion of combinations of ingredients and cooking instructions is possible. Let us suppose that, unfortunately as the situation is, you, also a chef, only encoded the most familiar and typical combinations, and the app is somewhat brittle, being unable to answer mild twists in the input. You only coded the most common cases.

When the app is running, it sends to the server described herein system 1 transactions summarizing each of the app's screens. Thus there is a system 1 transaction specifying the ingredients, a system 1 transaction specifying the cooking instructions, and a system 1 transaction specifying the qualities of the resulting dish. Let us suppose, for this example, that one hundred such routine cases are sent to the server.

At this stage, the server has to run an offline learning job; it reloads its models, and the app continues usage.

Now suppose an end user of the app proposes an ingredient not typically used for a particular recipe type in question, and suppose your calling software coding contains no conditions which can handle the combination in question. In this case the recipe app sends, as usual, a system 1 transaction for ingredients (with the unusual ingredient included) and a system 1 transaction for the cooking instructions to the server. But now the recipe app, having no solution and thus requiring deliberation,

instead sends a *system 2* transaction to the server which simply consists of a question mark: "?", tipping off the server that the app is unable to proceed.

The server evaluates the case. First of all, the server has learned with offline learning that ingredients and cooking times are needed for a recipe to be the result (this is a symbolic chain, discussed below). It verifies these pieces are duly in place. Assuming these are present, secondly the server determines the resulting dish and its qualities based upon the *ingredients' and cooking instructions' contextual essentialities, not their specific value comparison* (this is a deep learning step).

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.

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, 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) Judgment

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"). 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. Additionally, entry of an ellipsis ('…') at the end of the judgment in a system 2 call will result in the server completing the judgment. The calling software can switch in real time from system 1 to system 2 judgments.

2) Judgment Levels

There are three levels of judgments: 0, 1, and 2 in ascending precedence. The lowest level 0 judgment can only be a system 1 judgment (see below), and is always just taken as-is by the server, ie., taken as true. Immediately following a level 0 judgment there must be one and only one level 1 judgment 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.

3) System 1 vs System 2

System 1 judgments are taken by the server as true. System 2 judgments are taken by the server to be uncertain.

4) Essence (OPTIONAL)

Level 0 judgments can have an *essence* defined as a list of terms. 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. However, a call is made to an OpenAI large language model to include essence values based upon the judgment's context provided – so you can just use that instead of providing your own terms. 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 in this subgroup need the same subject in the 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 GENERAL DESCRIPTION

Below each REST transaction is informally described. A list of inputs to the server and output to the server are described. JSON is used (details elsewhere).

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.**

Input:	*<none>*	
Output:	*case number*	*integer*

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, however it is recommended for clarity. 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) however an essence can be determined automatically. 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.

Input:	*case number*	*integer*
	proposition	*English sentence*
	appearance	*terms from your input which led to proposition* **ONLY IF level is 0**
	essence	*terms capturing essence of proposition* **ONLY IF level is 0** **can use the automatic OpenAI results**
	level	*integer 0, 1, 2*
Output:	*<none>*	

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). If part of a judgment is submitted along with an ellipsis ('…') the server attempts to complete the judgment. Note: the synthetic propositions (aka judgments) are not generated until a call is made to sys2_realistic below.

Input:	*case number*	*integer*
	proposition	*1) English sentence, or* *2) "?" to signal sentence generation, or* *3) sentence fragment followed by "…" to signal the complete English sentence generation*
	level	*integer 1, 2*
	desired	*terms desired in the proposition*
Output:	*<none>*	

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 only to a judgment (not to essence or appearance). That means that a direct check is made to ensure that all “desired” terms are in the resulting judgment. The server makes a call to an OpenAI large language model to finish creation of an English sentence if needed. Importantly, you can call sys2_realistic as many times as desired so long as the case is not yet marked for training (below). If there is not a level 2 final judgment, only the most recent level 1 judgment is processed. So you can evaluate step-by-step if desired.

Input:	*case number*	*integer*
Output:	*decision*	*REALISTIC, UNREALISTIC, or IMPOSSIBLE*
	judgments	*list of all resulting judgments in the case in reverse order of submission: level 2 followed by its preceding level 1 judgments.*
	info	*list of entire case in AME format*

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.

Input:	*case number*	*integer*
Output:	*<none>*	

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.

Input:	*case number*	*integer*
Output:	*<none>*	

Copyright 2023 Michael Archbold