Agent-Oriented Software Engineering Project

Project Description

Authors

M2 DAI Students

Introduction

This document provides comprehensive details about the project, including the technologies employed, the project's architecture, and the division of tasks. Each role is associated with a dedicated repository (`R1` , `R2` , etc.).

Installation and Execution

To install the necessary dependencies for this project, please run the command: `python3 installer.py` .

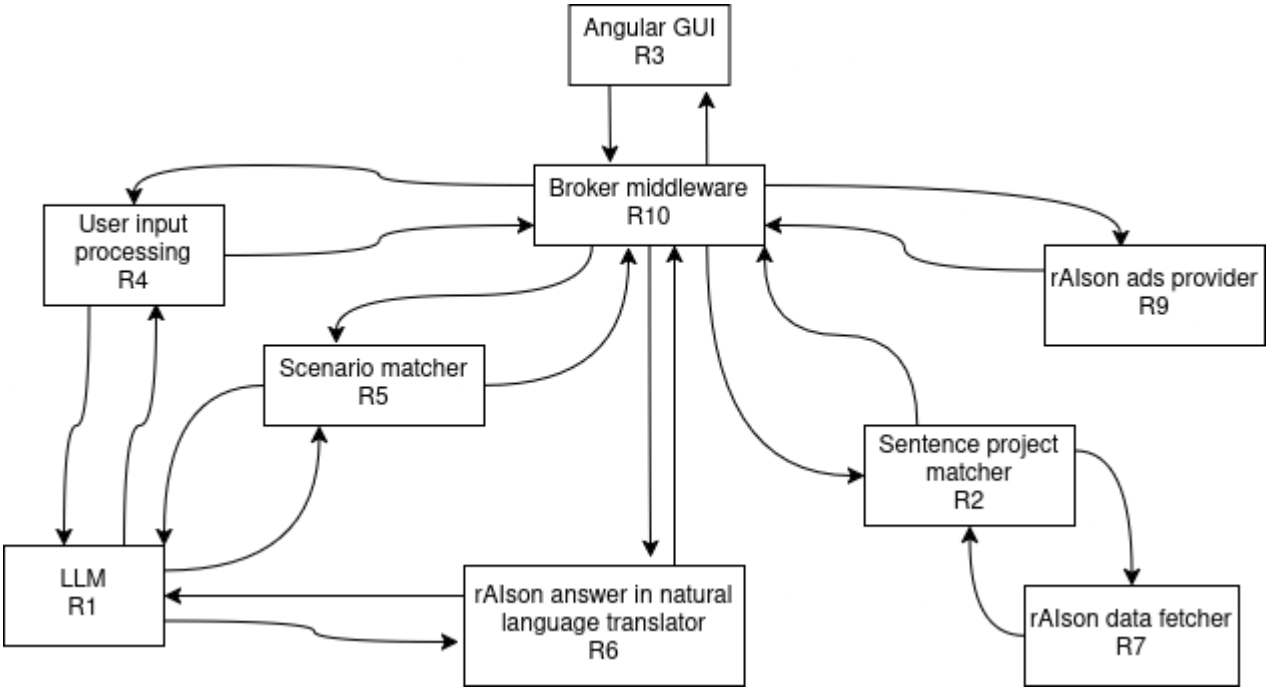
To execute the project, use the command: `python3 launcher.py` .

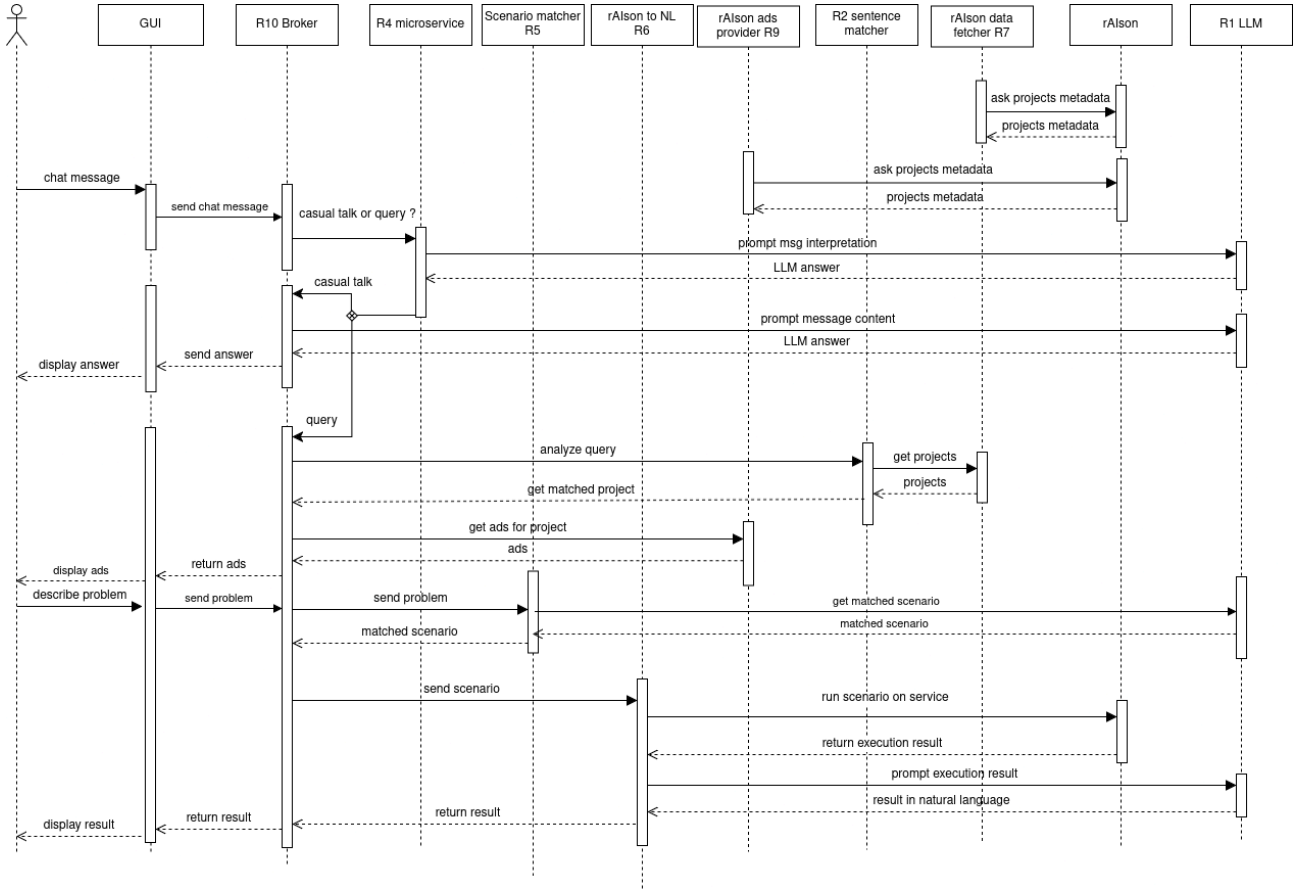
Technologies Used

This project is developed in Python. The API is built using Python FastAPI, a modern, fast, and reliable framework known for its performance and ease of use. The Large Language Model (LLM) utilized is `Nous-Hermes-2-Mistral-7B-DPO` , provided by HuggingFace. This LLM is deployed using a Docker container, ensuring consistent and isolated environments for development and deployment.

Architecture

We have adapted the project's architecture from an agent-based communication model to a microservices architecture using API endpoints. Each microservice plays the role of an agent, and they are coordinated by a broker middleware which sits between the frontend and the other agents. This approach enhances the ease of creation and maintenance. Below is a sequence diagram illustrating how our application operates:





Here are the different roles and their functionalities:

- **R1 LLM Service**: Utilizes Docker to launch the LLM and creates API endpoints to query the model.
- **R2 Sentence Matcher**: A sentence matcher based on SBERT that determines which rAIson project should be queried.
- **R3 Web GUI**: A web interface with a chat window that allows users to interact with the application.
- **R4 User Input Processing**: Queries R1 to ascertain whether a user input is a casual message or a query for a rAIson service.
- **R5 Scenario Matcher**: Processes user input to match it with rAIson service scenarios.
- **R6 rAIson Solution in Natural Language**: Executes the scenario on the appropriate rAIson service and queries the LLM to translate the result into natural language.
- **R7 rAIson data fetcher**: initialize an argumentation agent by fetching scenarios and options
- **R8 Communication Facilitator**: ensures communication between agents from R5 to R8
- **R9 Matchmaking Service**: does routing requests to appropriate argumentation agents based on their advertised capabilities.
- **R10 Broker**: Acts as an interface between the GUI and the backend microservices.
- **R11 and R12; Install scripts and running**: install and launch scripts for the microservices
- **R13 Project coordination**: handling coordination and communication

Task Division

The following students are assigned to each role:

- **R1**: Daniel Latorre
- **R2**: Tristan Duquesne
- **R3**: Abdou Aziz Thiam
- **R4**: Thanina Ait Ferhat
- **R5**: Nassim Lattab
- **R6**: Mohamed Azzaoui
- **R7**: Lynda Benkerrou
- **R8**: Yaya Latifou
- **R9**: Maram Beddouihech

- **R10**: Benlamara Kamilia
- **R11**: Cheikh Tidiane Diouf
- **R12**: Florian Posez
- **R13**: Théophile Romieu

R1 LLM Service

Author

Daniel Latorre

General Description

This project provides a FastAPI-based API that leverages a Mistral-based large language model, specifically the Nous-Hermes-2-Mistral-7B-DPO, for text generation. In addition to offering robust endpoints for generating text and managing sessions, this repository documents the state of the art in large language models and details the lineage, performance, and advantages of the selected model.

Table of Contents

Overview

The **R1 LLM Service** is built on FastAPI and provides endpoints for health checking, text generation, and session management. It integrates a state-of-the-art large language model that has been fine-tuned using Direct Preference Optimization (DPO) to ensure robust and aligned outputs. Conversation history is persisted in a PostgreSQL database, which is deployed via Docker Compose.

State of the Art

Historical Context and Evolution

The modern era of large language models began with the introduction of the Transformer architecture by Vaswani et al. (2017)[^1]. This breakthrough enabled models to learn complex relationships in text through self-attention mechanisms. Early work with encoder-only models, exemplified by BERT (Devlin et al., 2019) [^2], showcased the potential of bidirectional contextual

understanding for a variety of language tasks. Shortly thereafter, the field witnessed a shift toward decoder-only models like GPT-2 and GPT-3 (Brown et al., 2020)[^3], which, through unsupervised generative training, demonstrated the capacity to produce coherent and contextually relevant text. As model sizes increased, from billions to hundreds of billions of parameters, there was a marked improvement in capabilities, particularly in zero-shot and few-shot learning. Recent models such as GPT-4 (OpenAI, 2023)[^4], Google's Gemini, and Meta's LLaMA series have further advanced these capabilities, illustrating that even models with relatively modest parameter counts (e.g., 7B–65B) can achieve impressive performance when trained on high-quality data.

Diverse Architectures and Alignment Challenges

In addition to the GPT and BERT families, the landscape now includes models from EleutherAI (such as GPT-Neo and GPT-J) and the BigScience project's BLOOM, each offering unique advantages in terms of openness and performance. More recently, the Mistral architecture has emerged as an efficient alternative, emphasizing rapid inference and resource efficiency. Models based on Mistral not only perform strongly on reasoning benchmarks but are also optimized for deployment on less powerful hardware, a key advantage for real-world applications. Despite these architectural advances, a central challenge remains: aligning the model's outputs with human values and task-specific objectives. Traditional approaches like Reinforcement Learning from Human Feedback (RLHF) involve training a separate reward model followed by a reinforcement learning phase, a process that is often complex, computationally expensive, and sometimes unstable.

Advances in Alignment without Reinforcement Learning

Recent innovations have sought to simplify the alignment process by eliminating the need for a separate reward model. Direct Preference Optimization (DPO)[^5] is one such method that reframes the alignment challenge as a classification problem over human preference pairs. Instead of relying on the multi-stage RLHF pipeline, DPO directly adjusts the model's output probabilities by maximizing the log probability difference between preferred and dispreferred responses relative to a reference policy. This approach significantly reduces computational overhead and the need for extensive hyperparameter tuning, while achieving comparable or superior alignment performance. Empirical studies have shown that DPO-trained models perform exceptionally well in tasks such as sentiment modulation, summarization, and dialogue, offering a more stable and efficient alternative to traditional RLHF methods.

Criteria for Selecting a Large Language Model

Selecting an appropriate large language model involves a careful balance of several factors. Performance on standardized benchmarks—such as AGIEval, BigBench, GPT4All, and TruthfulQA—is critical, as these metrics assess the model's reasoning, generative coherence, and factual accuracy. Equally important is computational efficiency; models with a moderate parameter count that support quantization techniques can be deployed on hardware with limited resources. The alignment and safety of a model are paramount, and approaches like DPO help ensure that outputs adhere to human preferences and reduce the likelihood of generating harmful content. Moreover, the model's openness, reflected in its permissive license and integration with platforms like Hugging Face, facilitates reproducibility, community collaboration, and seamless deployment. Finally, the use of standardized prompt formats, such as ChatML, streamlines integration into conversational applications.

Selected Model: Nous-Hermes-2-Mistral-7B-DPO

Model Core Architecture

Nous-Hermes-2-Mistral-7B-DPO is a flagship 7B Hermes model. It is derived from Teknium's OpenHermes-2.5-Mistral-7B and has been further refined using Direct Preference Optimization. The model was fine-tuned on one million instructions and chat interactions of GPT-4 quality or better, primarily using synthetic data as well as other high-quality datasets from the Teknium/OpenHermes-2.5 repository.

Performance and Empirical Evaluation

Empirical evaluations confirm that Nous-Hermes-2-Mistral-7B-DPO has improved performance across all tested benchmarks. On the GPT4All benchmark, it achieves an average accuracy of approximately 73.72%, demonstrating robust generalization and reasoning capabilities. The AGIEval benchmark shows an average score of around 43.63%, indicating its competence in handling complex sentiment and logical reasoning tasks. BigBench evaluations, with an average score near 41.94%, further attest to its

proficiency in inference-based and multiple-choice tasks. Additionally, TruthfulQA results underscore its enhanced factual accuracy. Beyond these quantitative metrics, the model's performance in multi-turn dialogue and instruction-following tasks is exemplary, making it highly effective for practical applications.

Practical Advantages

The practical advantages of Nous-Hermes-2-Mistral-7B-DPO are numerous. Its moderate parameter count, combined with support for efficient quantization techniques (e.g., 4-bit mode, which can reduce VRAM usage to around 5GB), makes it accessible for deployment on a variety of hardware configurations. The use of DPO as the training paradigm ensures that the model closely adheres to human preferences, thereby producing reliable and safe outputs. Integration is further facilitated by its use of the ChatML prompt format, which enables structured multi-turn dialogue.

Rationale for Selection

The selection of Nous-Hermes-2-Mistral-7B-DPO is justified by its strong empirical performance, efficient resource utilization, and advanced alignment capabilities. The model's impressive performance across benchmarks such as GPT4All, AGIEval, BigBench, and TruthfulQA demonstrates its robust reasoning and generative abilities. Its efficient design, evidenced by its moderate parameter count and compatibility with quantization, ensures that it can be deployed in environments with limited hardware resources. Finally, the model's seamless integration with standardized prompt formats and its permissive open-source license make it an ideal choice for conversational AI to content generation.

Project Structure

```
R1/
├── app/
│   └── main.py          # FastAPI application code
├── docker-compose.yml   # Docker Compose configuration for PostgreSQL
├── README.md            # Project documentation (this file)
├── requirements.txt     # Python dependencies
└── .env                 # Environment variables (optional)
```

Setup Instructions

Prerequisites

- **Python 3.8+**
- **Docker & Docker Compose**
  Get Docker | Docker Compose Installation

1. Clone the Repository

```
git clone <repository_url>
cd R1
```

2. Create & Activate a Virtual Environment

On Unix/MacOS:

```
python3 -m venv .venv
source .venv/bin/activate
```

On Windows:

```
python -m venv venv
venv\Scripts\activate
```

3. Install Python Dependencies

```
pip install -r requirements.txt
```

4. Configure Environment Variables (Optional)

Create a `.env` file in the project root with the following content:

```
DATABASE_URL=postgresql://user:password@localhost:5432/db_name
```

Note: Adjust the values as necessary.

5. Start the PostgreSQL Database with Docker Compose

Ensure Docker is running, then execute:

```
docker-compose up -d
```

This command will:

- Download and run the PostgreSQL image.
- Create a database named **db_name** (as specified in the Docker Compose file).
- Expose port `5432` on your host.

6. Run the FastAPI Application

Start the API server using Uvicorn:

```
uvicorn app.main:app --host 0.0.0.0 --port 8000
```

Your API should now be accessible at http://0.0.0.0:8000.

API Endpoints

Health Check

- **URL:** `/health`
- **Method:** `GET`
- **Description:** Checks if the service is running.
- **Example:**

```
curl http://localhost:8000/health
```

Response:

```
{
  "status": "OK"
}
```

Generate Text

- **URL:** `/generate`
- **Method:** `POST`
- **Description:** Generates a response from the LLM based on user input.
- **Payload Example:**

```
{
  "session_id": "example-session",
  "user_message": "Hello, how are you?",
  "max_new_tokens": 200,
  "temperature": 0.7,
  "repetition_penalty": 1.1
}
```

- **Example using curl:**

```
curl -X POST "http://localhost:8000/generate" \
     -H "Content-Type: application/json" \
     -d '{
           "session_id": "example-session",
           "user_message": "Hello, how are you?",
           "max_new_tokens": 200,
           "temperature": 0.7,
           "repetition_penalty": 1.1
         }'
```

- **Response:**

```
{
  "response": "Generated response text from the model.",
  "session_id": "example-session"
}
```

Clear Session

- **URL:** `/clear_session`
- **Method:** `DELETE`
- **Description:** Clears the session history for the specified session ID by deleting all stored messages from the database. This endpoint is useful for resetting the conversation history.
- **Payload Example:**

```
{
"session_id": "example-session"
}
```

- **Example using curl:**

```
curl -X DELETE "http://localhost:8000/clear_session" \
   -H "Content-Type: application/json" \
   -d '{"session_id": "example-session"}'
```

- **Response:**

```
{
"status": "Session cleared",
"session_id": "example-session",
"deleted_messages": 5
}
```

References

[^1]: Vaswani, A., Shazeer, N., Parmar, N., et al. (2017). *Attention is All You Need*. Advances in Neural Information Processing Systems. Link.

[^2]: Devlin, J., Chang, M.-W., Lee, K., & Toutanova, K. (2019). *BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding*. Proceedings of NAACL-HLT. Link.

[^3]: Brown, T., Mann, B., Ryder, N., et al. (2020). *Language Models are Few-Shot Learners*. Advances in Neural Information Processing Systems. Link.

[^4]: OpenAI. (2023). *GPT-4 Technical Report*. arXiv preprint.

[^5]: Rafailov, R., Sharma, A., Mitchell, E., Manning, C. D., Ermon, S., & Finn, C. (2023). *Direct Preference Optimization: Your Language Model is Secretly a Reward Model*. NeurIPS 2023. Link.

Sentence matcher

Author

Role R2; Tristan Duquesne

General description

This section contains the work for the sentence matcher microservice. The sentence matcher microservice is responsible for matching user-provided sentences with a list of sentences. These sentences can either be provided via an HTTP POST message, or if ignored, are automatically set to correspond to descriptions, scenarios and options of the students' rAIson projects.

Initial Research Summary

Approaches

Sentence-matching in this context is a fuzzy process. The TLDR of our research is that there are basically three approaches we can take:

- **Embedding-based**: Use a model that can embed full sentences into a high-dimensional space and then compare the distance or angle between them. This is great for nuanced prompts and complex sentences, but can be overkill for simple word matching.
- **Lexicon-based**: Use a lexical database to match words and synonyms. This is great for simple word matching and synonyms, but struggles with complex sentences and idioms.

- **LLM-based**: Use a Large Language Model and prompt engineering to query the model's opinion on whether the input string (i.e., the user prompt) matches some base document (i.e., project data).

Tools

We identified some tools which, alone or combined, can help match sentences in an approximate but effective way:

SBERT

SBERT (Sentence Bidirectional Encoder Representations from Transformers) is a trained model that embeds sentence vectors into a high-dimensional space. SBERT can be obtained by doing `python3 -m pip install sentence-transformers`.

- Pros:
  - Captures contextual meaning, not just word overlap.
  - Great for nuanced prompts: "Looking to repair my computer" could match "Buying spare electronic parts".
  - Fast once embeddings are computed.
- Cons:
  - Needs a pretrained model (like SBERT) — might be overkill if you just want simple word matching.
  - Can struggle with very short prompts where context is limited.

Universal Sentence Encoder (USE)

Similar to SBERT, Google's USE is a powerful model also used for sentence embedding. It has a large and a lite version. Info about this model can be found here https://www.kaggle.com/models/google/universal-sentence-encoder/tensorFlow2 . We did not manage to get it to work though. The code for it is available, but commented, as loading these models is quite slow.

WordNet

WordNet is an English lexical database developed by Princeton university. It can match words to their synonyms and help define distance between lexical fields. It can be found as part of the NLTK library. You can install it by doing `python3 -m pip install nltk` and then runnning `nltk.download('wordnet')` and `nltk.download('omw-1.4')` within a Python script.

- Pros:
  - Excellent for handling synonyms and basic concept relationships.
  - Doesn't need massive neural models (purely rule-based), so very cheap.
- Cons:
  - Doesn't capture nuanced meanings or complex sentences.
  - Struggles with phrases and idioms ("My computer seems to have kick the bucket" won't link to "My computer is broken").

TF-IDF

TF-IDF (Term Frequency-Inverse Document Frequency) is a standard NLP algorithms. It scores words based on how often they appear in a sentence versus the whole document set. Rare words weigh more, so "electronic" counts more than "buying". While not specifically adapted to our problem on its own, it could help isolate the most relevant words and improve matching speed and/or reduce errors due to noise in casual text. TF-IDF can be found as a part of the scikit-learn library. You can install it by doing `python3 -m pip install scikit-learn`.

- Pros:
  - Simple and interpretable (shows how important a word is to a document).
  - Very fast, especially for larger corpora.
- Cons:
  - Ignores word order ("My computer needs a repair." and "My repair needs a computer." are the same to TF-IDF).
  - No sense of synonymy of lexical closeness ("treasure" and "gold" aren't linked).
  - Will need some sort of default text corpus to baseline against (e.g. the NLTK Brown corpus for English).

Distance Matching

By this, we mean raw string distance algorithms (Levenshtein distance, etc.). Again, while not directly useful to our problem, if we provide the user with the list of scenarios for the rAIson project, and the user has to select some scenarios, this can provide a pretty accurate and cheap way to ensure matching while also allowing for spelling mistakes, etc.

- Pros:
  - Good for matching noisy inputs or typos.
  - Lightweight, no model training needed.
- Cons:
  - Doesn't capture *any* meaning, only surface-level text.

Word2Vec

Word2Vec is a model that embeds words into a high-dimensional space. It can be used to compare words and find synonyms. It can be found as part of the gensim library. You can install it by doing `python3 -m pip install gensim`.

- Pros:
  - Great for finding synonyms and related words.
  - Can be used to find word analogies
- Cons:
  - Only works at the lexicon level.

GloVe

GloVe is another word embedding model, similar to Word2Vec. Unlike Word2Vec, it is trained on word co-occurrences so encodes a minimal amount of context. It can be found as part of the gensim library. You can install it by doing `python3 -m pip install gensim`.

- Pros:
  - Great for finding synonyms and related words.
  - Can be used to find word analogies.
- Cons:
  - Only works at the lexicon level.

LLMs

Large Language Models are models like GPT-4o, LLaMa 3, etc. They can be used for sentence matching by providing a base document and a prompt, and then querying the model for its opinion on whether the prompt matches the document. This is a very powerful tool, but it can be expensive to use and requires a lot of data to train.

- Pros:
  - Can be very powerful and flexible.
  - Can often handle complex sentences and nuanced meanings.
- Cons:
  - Expensive and slow to train and run.
  - Generally unexplainable, so hard to fix if it doesn't have desirable results.
  - Needs data to fine-tune.
  - Getting the right outputs needs prompt engineering, data engineering, and solid benchmarking.

Implementation

Because we wanted to aim for speed, cheapness, measurability and consistency, we thus concentrated our efforts on non-LLM approaches. We provided studied 2 lexicon-based approaches (synset distances and word embeddings) and 1 sentence-based approach (SBERT).

Commonalities: similarity matrices, extrema and averaging

The core of our approach was to build all comparisons between a set of input strings, and a set of baseline strings. This can be understood as a weighted bipartite graph, and represented as a similarity matrix, where rows correspond to baseline sentences, and columns correspond to input sentences. Each cell in the matrix is the similarity score between the corresponding baseline and input sentence. In one case (wordnet) this score is a distance, in the other cases (word embeddings, sentence embeddings) this score is a cosine similarity.

We then computed the extrema of these matrices over the columns. For distances, a minimum; for similarities, a maximum. The idea here is that if a string is similar to *any* of the strings in the baseline, it should be considered a match. This gives us a good way to notice matches and remove noise.

Finally, since the amount of words in the input might not be the same everywhere, combining the scores for the various strings in the input needed some clever resolution, one that would not be sensitive to changes in prompt length. We thus averaged the scores in various ways to get a final score for each input sentence. The three averages we provided are arithmetic, softmax and harmonic.

The arithmetic mean is the most basic, the softmax mean gives more weight to higher scores, and the harmonic mean is a way to create distance between matches which are consistently strong and penalize those that are only partly good matches.

Specific to lexicon-based approaches: TF-IDF

We tried to use TF-IDF to filter out noise and only keep rare words (those potentially relevant to thematic matchings only), limit the amount of tokens to cross for the 2 lexicon-based matchers, and improve the overall score. Since the lexicon-based matchers proved to be less performant than the sentence-based matcher, this part of the code is mostly unused. It is still present in the sentence matcher file. This portion of the code could still be interesting to use and study, especially if someone were to try to improve the word embeddings approach, which looked like it could be useful if more work was done on it.

Synset distances: WordNet

The idea of the approach here was to use WordNet, a knowledge graph of English words, to find synonyms and related words. We used the NLTK library to access WordNet and compute the distance between synsets. The distance was computed using the shortest path between two synsets in the WordNet graph. The idea was to compare the distance between the synsets of the words in the input and the baseline, and then average these distances in the way described above.

We fudged around with the various choices of distance metrics available through wordnet, but never got to a point where we found the order of distances to match the order of relevance that we thought was semantically true. We thus abandoned this approach, but left the code in place for future reference. It is also testable via the API.

Word embeddings: Word2Vec, GloVe

This approach fared a bit better. We used the `gensim` library to load Word2Vec and GloVe models to provide our word embeddings for our TF-IDF filtered words. We then computed the cosine similarity between the vectors of the words in the input and the baseline. We then averaged these similarities in the way described above.

The results were far more semantically consistent. However, there was little difference in score between results, whether they were really relevant, somewhat relevant, or not relevant. So this was too imperfect for a threshold-based approach.

Sentence embeddings: SBERT

(Note, we also tried to use Google USE, but failed to get it working.)

This approach was the most successful. We used the `sentence-transformers` library to load the SBERT model and compute the sentence embeddings for the input and baseline sentences. We then computed the cosine similarity between these embeddings, and averaged them in the way described above.

The results were semantically consistent, and the scores were more spread out. This was the most promising approach, and the one we chose to keep as the default for the final microservice. The code is still present for the other approaches, and can be tested via the API. However, there needs to be further testing to figure out thresholds appropriately.

Results and limitations

We found cosine similarity with SBERT to work the best, because it was the most semantically consistent and had the most spread out scores.

Admittedly, our testing is quite minimal. We need a lot more fine-grained testing to establish which choice of model, which similarity metric, and which averaging function are best. We are not confident as to which is the best approach, in general, and for this use case. However, our approach is good enough for a prototype, so we stopped there. The kind of benchmarking required to improve upon this would need a dataset of prompts and expected matches, and a way to measure the performance of the various approaches. Because this is a complex, time-consuming task, we considered it to be out of scope.

Also, we did not try the LLM approach, but once such a dataset of comparison baselines is available, it would be very interesting to compare it with the one given. Although LLMs are not explainable, they are very powerful: with some explainable alternate model to compare against, this could mitigate some of the trustworthiness issues of LLM outputs (hallucinations). Potentially, we could also fine-tune a SotA model to provide a more accurate and nuanced matching.

Running

When running this microservice for the first time, it will download the appropriate models from the Internet: this will take a while (a few minutes).

When running this microservice again, it will load the models from the local cache, which is much faster, but still takes a while (20-30 seconds or so).

The version of Python used for development is 3.10, but some other (recent-ish) versions of Python 3 should also work fine.

Installation

At the root of the R2 directory, you should add a file called `.env` containing a rAIson API key. The syntax for it is as follows (where `...` is to be replaced by the key):

```
RAISON_API_KEY = "..."
```

You can install the necessary dependencies by running:

```
python3 -m pip install -r requirements.txt
```

You can also set up a `venv` for this purpose if necessary.

If you don't want to use a `venv`, but might have version conflicts with other python packages locally, you can run:

```
python3 -m pip install -U -r requirements.txt
```

Testing

If you want to test the sentence matcher on its own, you can do so by running the following command:

```
python3 -m src.sentence_matcher
```

This will call the `if __name__ == "__main__":` block in `src/sentence_matcher.py` and run the sentence matcher on some predefined sentences. This can be edited for quick debugging.

Microservice / agent

Since there are some data and models to be loaded before the sentence matcher microservice is online, you cannot launch it with `uvicorn src.role2_service:app --port=8002`, you should instead do `python3.10 -m src.role2_service`, which will read the `if __name__ == "__main__":` block in `src/role2_service.py` (where the data loaders are called), then launch the microservice. Calling uvicorn immediately ignores the data loading and will cause issues with the microservice.

You can then test its behavior with something like this:

```
curl -X POST 0.0.0.0:8002/match_for_scenario -H "Content-Type: application/json" \
    -d "{\"user_input\": \"I'm wondering if I should ban someone from my discord server\", \"get_max\": true}"
```

API

The sentence matcher microservice exposes 3 routes:

- `/match`: the general route which calls the sentence matcher general request handler. This route takes a somewhat complex set of arguments (as a JSON), and can thus let the sentence matcher serve for various purposes:
  - `user_input : str` : required, the user input which we need to compare to a set of baseline documents.
  - `documents : DocumentDict`: optional, a dictionary of document IDs and document content (list of sentence). If not provided, this is automatically filled with a dictionary of the rAIson project's description, scenarios and options.
  - `model : ModelQueryKey`: optional, the model/method to use for the matching. If not provided, this is automatically set to `"sbert"`.
  - `mean_mode : MeanMode_Literal`: optional, one of `"arithmetic"`, `"softmax"`, or `"harmonic"`. This sets the average used to harmonize sets of word comparisons. If not provided, this is automatically set to `"harmonic"`.
  - `dist_mode : WNDistanceMode_Literal`: optional, only applies to the `"wordnet"` model. If not provided, this is automatically set to `"path"`.
  - `alpha : float`: optional, only applies to the `"softmax"` mean. If not provided, this is automatically set to `1.5`.
  - `epsilon : float`: optional, only applies to the `"harmonic"` mean. If not provided, this is automatically set to `1e-6`.
- `/match_for_ad`: a simplified call to help R7.
- `/match_for_scenario`: a simplified call to help R5.

Code structure

The code for R2 is structured as follows:

- `project_data.py`: Contains utils to manipulate metadata and data of the rAIson projects. Some of this metadata is not provided by the rAIson API (project title and a description), so it is hardcoded in this file. The rest is obtained by querying the rAIson API (elements and options).
- `sentence_matcher.py`: Contains the sentence matcher utility functions. This file is responsible for matching user-provided sentences with a list of predefined sentences (corresponding to scenarios and options of rAIson projects). It is structured in sections, as follows:
  - Typing and constants: some semantic type aliasing and default values.
  - Similarity scoring utils: various functions, types and constants to help define the degree of similarity between words and sentences. The functions in question help to study similarity (cosine), dissimilarity (distance), and to average scores over multiples inputs when crossing multiple input words or sentences with multiple baseline words or sentences.
  - TF-IDF utils: functions to compute the TF-IDF score of a sentence, and filter words based on this analysis. I kept it here because it was important for the study of the different methods, but the chosen SBERT matcher does not use it.
  - User input vs basline comparison functions: these are the three functions providing an output based on our approaches (one for WordNet, one for word-embedding cosine similarities, and the final, which acts as our default, the sentence-embedding cosine similarities).

    - Model loaders: utils to load the SBERT model and the WordNet database. As mentioned, USE, while present as comments in the code, does not work.
    - Request handler: the core function that handles the POST requests for the microservice, and acts as a sort of gateway for the various ways one can call the sentence matcher.
    - Main: a small block for direct testing of the code in this file.
- `role2_service.py`: Contains the FastAPI microservice for the sentence matcher. This file is responsible for exposing the sentence matcher as a microservice, and is structured as follows:
    - Typing and constants: some semantic typing and default values, using Pydantic, to help FastAPI handle the JSON data.
    - Request handlers: the FastAPI routes for the sentence matcher microservice. There are 3: `match`, the general route which calls the sentence matcher general request handler; `match_for_ad` which is a simplified call to help R7; `match_for_scenario` which is a simplified call to help R5.
    - Main: loading the requisite data models, and launching the FastAPI/uvicorn app.

TODOs

- Add an LLM similarity matcher: a comparison function which does some prompt engineering and then calls R1 (and corresponding API route)
- Fix Google USE

Frontend

Author

R3 Abdou Aziz Thiam

Description

This project was built using Angular CLI version 19.0.6.

Development server

To start a local development server, run:

```
ng serve
```

Once the server is running, open your browser and navigate to `http://localhost:4200/`. The application will automatically reload whenever you modify any of the source files.

Code scaffolding

Angular CLI includes powerful code scaffolding tools. To generate a new component, run:

```
ng generate component component-name
```

For a complete list of available schematics (such as `components`, `directives`, or `pipes`), run:

```
ng generate --help
```

Building

To build the project run:

```
ng build
```

This will compile your project and store the build artifacts in the `dist/` directory. By default, the production build optimizes your application for performance and speed.

Running unit tests

To execute unit tests with the Karma test runner, use the following command:

```
ng test
```

Running end-to-end tests

For end-to-end (e2e) testing, run:

```
ng e2e
```

Angular CLI does not come with an end-to-end testing framework by default. You can choose one that suits your needs.

Additional Resources

For more information on using the Angular CLI, including detailed command references, visit the Angular CLI Overview and Command Reference page.

Process User Input

Author

Role R4: Thanina AIT FERHAT

This is a FastAPI application designed to process user input and classify it as either a casual conversation or a service request. The application interacts with two APIs:

1. R1 API

A language model API that classifies the user's input and generates responses based on the type of interaction.

2. Broker API

An API that processes decision-making requests from users, helping to make informed decisions based on their input.

The application saves conversation histories in a PostgreSQL database, which can be queried to retrieve the conversation history for a given session.

Features

- Classifies user input as either a casual conversation or a decision-making request.
- Calls the **R1 API** for natural language generation.
- Interacts with the **Broker API** for decision-making requests.
- Stores conversation history in a PostgreSQL database.
- Exposes an endpoint to process user input and respond accordingly.

Requirements

- Python 3.7+
- PostgreSQL
- FastAPI
- SQLAlchemy
- Pydantic
- Requests

Setup

1. Clone the Repository

```
git clone https://github.com/your-username/r4-process-user-input.git
cd R4-process-user-input
```

2. Create a Virtual Environment

```
python -m venv venv
source venv/bin/activate  # On Windows: venv\Scripts\activate
```

3. Install Dependencies

```
pip install -r requirements.txt
```

4. Set Up the Database Ensure that you have a PostgreSQL instance running. Create a new database and update the DATABASE_URL in the .env file with the appropriate credentials.

Example DATABASE_URL in .env file

```
DATABASE_URL=postgresql://user:password@localhost:5432/db_name
```

Run the following command to create the necessary tables in the database:

```
python role4_service.py
```

5. Running the Application To start the FastAPI server, run:

```
uvicorn app.role4_service:app --reload
```

The application will be available at http://127.0.0.1:8004.

API Endpoints

To test the `/process_input` endpoint, you can send a POST request with the following JSON body:

```
{
    "session_id": "session123",
    "user_message": "Can you help me choose a restaurant?"
}
```

Example curl request:

```
curl -X 'POST' \
  'http://127.0.0.1:8004/process_input' \
  -H 'Content-Type: application/json' \
  -d '{
  "session_id": "session123",
  "user_message": "Can you help me choose a restaurant?"
}'
```

The response will be a JSON object with the following structure:

```
{
    "response": "Sure, I can help you with that!",
    "session_id": "session123"
}
```

The `/classify_input` endpoint is responsible for classifying the user's input as either a service request or a casual conversation. It uses the `classify_input` function to determine the type of input based on the user's message.

This endpoint receives a `UserInput` object containing the user's session ID and message. It then classifies the message and returns a boolean indicating whether the input is a service request (`true`) or a casual conversation (`false`).

To use this endpoint, send a POST request with the following JSON body:

```
{
  "session_id": "session123",
  "user_message": "Can you help me choose a restaurant?"
}
```

Example cURL request:

```
{
curl -X 'POST' \
  'http://127.0.0.1:8000/classify_input' \
  -H 'Content-Type: application/json' \
  -d '{
  "session_id": "session123",
```

```
  "user_message": "Can you help me choose a restaurant?"
}
```

Response The response will be a boolean indicating whether the user's input is a service request (true) or a casual conversation (false).

Example Response:

```
{
true
}
```

`

1. Classify User Input

The application starts by classifying the user's input using the `classify_input` function. This function determines whether the input corresponds to:

- **A casual conversation**.
- **A service request** (decision-making request).

If the input contains keywords related to decision-making, it will be classified as a service request. Otherwise, it will be considered a casual conversation.

2. Casual Conversation

If the input is classified as a casual conversation:

- The application sends the input to the **R1 API**, which generates an appropriate response.
- This interaction is then recorded in the database, including the response generated by the assistant.

3. Decision Request

If the input is classified as a service request:

- The application sends the input to the **Broker API** to process the request.
- The response generated by the **Broker API** is then sent to the **R1 API**, which transforms this response into natural language.
- Finally, the response is recorded in the database, including the response generated by the assistant.

Role 5 - Scenarios Matcher

Author

Role R5, **Nassim Lattab**

Purpose: This script implements Role 5 in our multi-role system. It is responsible for matching user input phrases with scenarios associated with a specific project. More precisely, **Role 5** processes user input to match it against predefined scenarios retrieved from the Ai-Raison API. The process involves receiving a project identifier and user phrases, constructing a detailed prompt for a locally hosted LLM, and finally parsing the response to extract the matched scenarios. All results are formatted in JSON and forwarded to for further processing.

The script is as follows:

1. **Retrieves** the list of scenarios for a given project from the Ai-Raison API.
2. **Builds** a prompt that combines the retrieved scenarios with user phrases.
3. **Sends** the prompt to a locally hosted LLM API (via FastAPI) for matching.
4. **Extracts** and returns the matching scenarios from the LLM's JSON response.
5. **Formats** the output as JSON to be forwarded to the Broker.

Project Overview

In our multi-role architecture, **Role 5** handles the scenario matching process:

- **Input:** In order to function properly, **Role 5** relies on two key inputs that allow it to fetch relevant scenarios and understand the user's needs:

  - **Project Identifier (`project_id`):** A unique string that is used to query the Ai-Raison API. This API returns project metadata including all available scenarios (elements) and options.
    Example: `"PRJ15875"`

  - **User Input Phrases (`user_input`):**
    A list of sentences that express the user's requirements or requests.
    Example: `["I'd like to have my computer repaired."]`

  These inputs are typically provided as JSON in a POST request to the `/match` endpoint. A **Pydantic** model ensures that `project_id` is a string and `user_input` is a list of strings, offering basic validation.

- **Process:**

  i. The script constructs an API URL using the project ID and fetches metadata from the Ai-Raison API.
  ii. It extracts scenario labels from the metadata.
  iii. It builds a detailed prompt combining the list of scenarios and the user phrases, instructing the LLM to return a JSON object with the matched scenarios.
  iv. It calls the LLM API (running locally at `http://localhost:8000/generate`) to get a response.
  v. The raw LLM output is preprocessed using regex to extract only the JSON block.
  vi. The extracted JSON is parsed and returned as a dictionary.

- **Output:** After processing the inputs, Role 5 generates a structured JSON object that encapsulates both the original request and the system's matching results. The final output is a JSON object that contains:

  - **project_id**: The project identifier.
  - **user_input**: The original user input text.
  - **matched_scenarios**: An array containing the scenarios that best match the user's input.
  - **info**: A string field for additional information (e.g., error messages or processing details). This field may be empty on success. Below is a global structure for the final output:

```
{
    "project_id": "PRJID05",
    "user_input": ["The user input text here."],
    "matched_scenarios": [
        "scenario1", "scenario2"
    ],
    "info": "..."
}
```

  Note : The list may include additional scenario names.

Algorithm

The matching algorithm in Role 5 orchestrates a sequence of operations designed to retrieve relevant scenarios, construct a suitable prompt, interact with the LLM, and deliver the final results. The process follows these detailed steps:

Unified Processing Flow and Algorithm

1. **Scenario Retrieval**

   - **API Call & Metadata Parsing**
     The function `get_project_scenarios(project_id)` constructs an endpoint URL (for instance, `https://api.ai-raison.com/executions/PRJ15875/latest`) and calls `get_data_api(url, api_key)` to fetch the project metadata. Next, `extract_elements_and_options(metadata)` parses this metadata to extract scenario labels, retaining only the keys (scenario names) as the list of possible scenarios.

2. **Prompt Construction**

   - **Instruction & Formatting**
     The function `build_prompt(scenarios, user_input)` creates a prompt instructing the LLM to match user requests with the provided scenarios and to return only a strict JSON output. Specifically, the prompt includes:
     - A clear directive for the LLM.
     - The list of retrieved scenarios (formatted as a comma-separated string).
     - The user input phrases.

3. **LLM Invocation and Response Processing**

   - **Call LLM API**
     The function `call_llm(session_id, prompt, host)` sends the prompt to the LLM API (e.g., `http://localhost:8000/generate`), specifying parameters such as `max_new_tokens`, `temperature`, and `repetition_penalty` to shape the generation process.
   - **Response Handling**
     The raw response from the LLM is processed using a regular expression to extract the JSON block. If extraction or parsing fails, the algorithm logs an error and returns an empty `matched_scenarios` list with a relevant message in the `info` field.

4. **Final Output and Forwarding**

   - After the JSON is successfully parsed, the system enriches the result by adding the original `project_id` and `user_input` fields, and sets the `info` field to empty upon success.
   - The resulting JSON object is then forwarded to the Broker via an HTTP POST.

Pseudocode Representation

```
def match_scenarios_with_llm(project_id, user_input):
    scenarios = get_project_scenarios(project_id)
    prompt = build_prompt(scenarios, user_input)
    llm_output = call_llm("matching_scenarios_session", prompt)

    extracted_json = extract_JSON(llm_output)
    if not extracted_json:
        result = error_result(project_id, user_input, "No JSON object found")
    else:
        parsed_result = parse_JSON(extracted_json)
        if not parsed_result:
            result = error_result(project_id, user_input, "Could not parse JSON")
        else:
            result = formatted_result(project_id, user_input, parsed_result)

    return result # Broker retrieves results
```

This step-by-step algorithm ensures that the user input is accurately matched to the available scenarios by leveraging the generative capabilities of the LLM while enforcing strict JSON output for reliable downstream processing.

Code Structure

- **Configuration:**
  Sensitive data such as the API key are stored in a separate configuration file `(config.py)`. Note: Ensure that this file is excluded from version control (e.g., via `.gitignore`).

- **Functions:**

  - `extract_elements_and_options(metadata)`
    Extracts scenario labels (elements) and options from the metadata.

  - `get_data_api(url, api_key)`
    Retrieves JSON data from the Ai-Raison API.

  - `get_project_scenarios(project_id)`
    Constructs the URL and extracts a list of scenario labels for the specified project.

  - `build_prompt(scenarios, user_phrases)`
    Creates a prompt that instructs the LLM to perform scenario matching using only JSON output.

  - `call_llm(session_id, prompt, host)`
    Sends the prompt to the local LLM API endpoint and retrieves the response.

  - `match_scenarios_with_llm(project_id, user_phrases)`
    Orchestrates the entire process from retrieving scenarios to parsing the LLM's JSON response.

- **Regex Preprocessing:**
  Before attempting to parse the LLM response, a regex is used to extract only the JSON block from the raw output to handle any extra text returned by the model.

Setup Instructions

0. Cloning the Repository

If you have access to the repository, you can clone it with:

```
git clone <repository_url>
```

Then navigate to the R5 folder before starting the service.

```
cd R5
```

1. Configuration:

Create a `config.py` file (or similar) that includes your sensitive information such as the api key for AI-Raison API:

```
# config.py
api_key = "YOUR_API_KEY_HERE"
```

Note : Make sure this file is not shared or pushed to public repositories.

2. Dependencies:

Install the required packages using:

```
pip install requests
```

3. LLM API:

> **Important:** Role 5 depends on the Role 1 LLM Service to be running.

Ensure Role 1 is set up according to its README and that your local LLM API (R1) is running. You can start it with:

```
uvicorn app.main:app --host 0.0.0.0 --port 8000
```

Verify that the health endpoint (`http://localhost:8000/health`) returns the expected JSON response (e.g., `{"status": "OK"}`). You can check it with the following command:

```
curl http://localhost:8000/health
```

4. Run the Service:

Start the Role 5 service with:

```
uvicorn role5_service:app --host 0.0.0.0 --port 8005
```

This starts the service on port **8005**, making the following endpoints available:

- Health Check: `http://localhost:8005/health`
- Matching Endpoint: `http://localhost:8005/match`

5. Testing the Communication:

You can test the service with a separate Python script (e.g., `test_role5.py`) that sends a POST request with hard-coded values, as the Broker do:

```
import json
import requests

url = "http://localhost:8005/match"

payload = {
    "project_id": "PRJ15875",
    "user_phrases": ["I'd like to have my computer repaired"]
}

try:
    response = requests.post(url, json=payload)
    response.raise_for_status()
    result = response.json()
    print("Matched Scenarios Result:")
    print(json.dumps(result, indent=2, ensure_ascii=False))
```

```
except Exception as e:
    print("Error during test:", e)
```

Run this script with:

```
python test_role5.py
```

Example Usage

For testing, if you send the following input:

```
# Test data
payload = {
    "project_id": "PRJ15875",
    "user_input": ["I'd like to have my computer repaired. The product is not repairable but is under warranty,
what can you do for me?"]
}
```

The receive output is:

```
PS D:\Documents\M2\Agent Oriented Software Engineering\aose-2025> python -u "d:\Documents\M2\Agent Oriented Software Engineering\aose-2025\R5\test_role5.py"
Matched Scenarios Result:
{
  "project_id": "PRJ15875",
  "user_input": [
    "I'd like to have my computer repaired. The product is not repairable but is under warranty, what can you do for me?"
  ],
  "matched_scenarios": [
    "product under warranty",
    "non repairable product"
  ],
  "info": ""
}
PS D:\Documents\M2\Agent Oriented Software Engineering\aose-2025>
```

This JSON output can then be returned to be forwarded once it is configured.

Notes

- **Prompt Engineering:** The prompt provided to the LLM is crucial. Ensure it clearly instructs the LLM to return only valid JSON and no additional commentary.
- **Error Handling:** The script includes basic error handling in case the LLM response cannot be parsed as JSON. You may further enhance this with more robust logging and recovery strategies.
- **Customization:** Adjust the API URLs, generation parameters (e.g., max_new_tokens, temperature), and other configurations according to your environment and requirements.

Log Files

Effective logging is critical for monitoring, debugging, and auditing the matching process. In our implementation, logging occurs at several key stages:

- **Request Logging:** Log the incoming project_id and user_input with a timestamp when a request is received at the /match endpoint.
- **Prompt and API Call Logging:** Log the complete prompt and details of the outgoing LLM API call (including parameters and session ID), as well as the raw LLM output.

- **Response Processing Logging:** Log the result of the regex extraction and JSON parsing. If errors occur, log the error details and problematic output.
- **Forwarding Logging:** Log the final JSON result before it is returned, along with any errors encountered during the HTTP POST.

This streamlined logging approach ensures essential traceability without excessive repetition, allowing issues to be quickly identified and resolved. Example Log Entry we could have :

```
2025-02-24 15:30:45,123 INFO [Role5] Received request for project_id: PRJ15875 with user_input: ["I'd like to ha
2025-02-24 15:30:45,456 DEBUG [Role5] Constructed prompt: "You are an AI assistant that matches user requests...
2025-02-24 15:30:46,789 DEBUG [Role5] LLM raw output: "{ "matched_scenarios": ["repair request"] }"
2025-02-24 15:30:46,900 INFO [Role5] Successfully parsed LLM response. Matched scenarios: ["repair request"]
```

This detailed logging strategy provides full traceability of the entire matching process, from the moment a request is received to when the result is forwarded to the next service. It ensures that any issues can be quickly identified and resolved, which is critical in a multi-service architecture where errors in one role can affect the overall system performance.

R5 Conclusion

In summary, **Role 5** serves as a critical component within our multi-role architecture, bridging user input and project-specific scenarios through an LLM-based matching process. By leveraging the Ai-Raison API to retrieve scenario labels and constructing a clear, instructive prompt, this service ensures that user requests are accurately classified. Robust logging, strict JSON output handling, and optional error recovery measures further enhance reliability. Once Role 5 completes its matching task, it seamlessly forwards the results to the Broker, integrating smoothly with the broader system. This design promotes modularity, scalability, and maintainability, enabling future enhancements or customizations without disrupting the core functionality.

Role 6 - Solving Problems

Author

Role R6, Mohamed Azzaoui

General Description

This service is a microservice that finds solutions based on user input and matched scenarios with the platform AI Raison. It interacts with an LLM to generate clear and relevant explanatory responses. The service communicates with an LLM provided by **Role R2** to create a clear response for the user and receives input data (the found scenarios and user's sentence) from **Role R5** via a dedicated endpoint.

Installation

Clone this repository:

```
git clone <https://github.com/truite-codeuse/aose-2025.git>
cd <R6>
```

Create and activate a virtual environment:

```
python -m venv venv
```

```
source venv/bin/activate  # On Windows: venv\Scripts\activate
```

Install required dependencies:

```
pip install -r requirements.txt
```

Create an API key:

If you don't have an API key, follow the instructions on **AI Raison's website** to generate one.

Create a file private_information.py at the project root containing your API key:

api_key = "YOUR_API_KEY"

Run the service:

```
uvicorn role6_service:app --host 0.0.0.0 --port 8006 --reload
```

API Endpoints

Health Check

- **Route:** `/health`
- **Method:** `GET`
- **Description:** Checks if the service is running.
- **Response:**

```
{"status": "OK"}
```

Find a Solution

- **Route:** `/find_solution`
- **Method:** `POST`
- **Description:** Finds a solution based on user input and provided scenarios. This endpoint receives information from **Role R5** and processes it by calling an **LLM from Role R2** to generate an appropriate response.

Request of R5 (JSON):

```
{
  "project_id": "project_id",
  "user_input": ["example user input"],
  "matched_scenarios": ["scenario1", "scenario2"],
  "info": "additional information"
}
```

Response of R6 (JSON):

```
{
  "text": "Generated AI response"
```

```
}
```

Process Explanation

1. The endpoint receives data from **Role R5**.
2. It calls the `find_solution_llm` function to retrieve a solution using an **LLM from Role R2**.
3. If an error occurs, it logs the request and returns a server error message.
4. Otherwise, it returns the generated solution to the user.

Endpoint Code:

```python
@app.post("/find_solution", response_model=MatchResponse)
def match_endpoint(request: MatchRequest):
    try:
        result = find_solution_llm(request.project_id, request.matched_scenarios, request.user_input)
    except Exception as e:
        print(f"REQUEST RECEIVED = {request}")
        print(f"Error occurred: {str(e)}")  # Log the error
        raise HTTPException(status_code=500, detail="An error occurred on the server.")

    return MatchResponse(text=result)
```

Key Features

AI Raison API Integration:

- Retrieve project metadata.

```python
def get_data_api(url, api_key):
    """
    Retrieves data from the API and extracts labels and IDs.

    Parameters:
        url (str): The API URL.
        api_key (str): API key for authentication.

    Returns:
        metadata: Returned metadata
    """
    headers = {"x-api-key": api_key}

    try:
        response = requests.get(url, headers=headers)
        if response.status_code == 200:
            return response.json()
        elif response.status_code == 400:
            print("Error 400: Invalid request.")
        else:
            print(f"Error {response.status_code}: {response.text}")
    except Exception as e:
        print(f"An error occurred: {e}")
    return None
```

- Extract associated elements and options.

```python
def extract_elements_and_options(metadata):
    """
    Extracts labels and IDs of elements, as well as the IDs of options.
```

```
    Parameters:
        metadata (dict): Data containing 'elements' and 'options' sections.

    Returns:
        tuple:
            - A dictionary {label: id} for elements.
            - A list of dictionaries [{"id": id}] for options.
    """
    elems = {}  # Stores labels and IDs of elements
    opts = []   # Stores IDs of options

    elem_items = metadata.get('elements', [])
    opts_items = metadata.get('options', [])

    for item in elem_items:
        label = item.get("label")
        id_value = item.get("id")
        elems[label] = id_value

    for item in opts_items:
        id_value = item.get("id")
        opts.append({"id": id_value})

    return elems, opts
```

- Send scenarios to the API to evaluate valid solutions.

```
def call_api(project_id, scenario):
    """
    Calls the API to evaluate scenarios and returns valid options.

    Parameters:
        project_id (str): The ID of the project.
        scenario (list): List of labels of elements to include in the scenario.

    Returns:
        list: Valid solutions or None if none.
    """
    base_url = "https://api.ai-raison.com/executions"
    url = f"{base_url}/{project_id}/latest"

    metadata = get_data_api(url, api_key)
    elements, options = extract_elements_and_options(metadata)

    headers = {
        "x-api-key": api_key,
        "Content-Type": "application/json"
    }

    ids = [{"id": elements[case]} for case in scenario]

    payload = {
        "elements": ids,
        "options": options,
        "limit": len(options)
    }

    try:
        response = requests.post(url, headers=headers, json=payload)
        if response.status_code == 200:
            metadata = response.json()
        elif response.status_code == 400:
            print("Error 400: Invalid request.")
        else:
            print(f"Error {response.status_code}: {response.text}")
    except Exception as e:
        print(f"An error occurred: {e}")
```

```
    return check_solutions(metadata)
```

Request of R5 (JSON):

the url for the request is "https://api.ai-raison.com/executions/{project_id}/latest"

```
{
  "elements": "example_scenario",
  "options": "example_possible_results",
  "limit": "max_number_of_solutions"
}
```

Response of Ai-Raison (JSON):

```
{
  "options": "dictionary_of_options_with_boolean_values"
}
```

LLM Interaction (Role R2):

- Build a structured prompt using user requests and available options.
- Send the prompt to an LLM model to generate an explanatory response.

Middleware & FastAPI API:

- REST interface allowing users to send requests and receive solutions.
- Logging and error handling for service robustness.
- Direct communication with **Role R5** for input data retrieval.

Project Structure

```
R6/
|-- role6_service.py        # Initializes the FastAPI service and uses my microservice when listening to a reque
|-- test.py                 # Testing my work independently
|-- api.py                  # Connects and extracts data from AI Raison API
|-- requirements.txt        # List of dependencies
|-- private_information.py  # File containing the API key (not versioned)
```

Requirements File (`requirements.txt`)

```
fastapi
uvicorn
requests
pydantic
```

Argumentation Agent Initialization API

Author

Role R7: Lynda BENKERROU

Description

This project provides a FastAPI allowing you to initialize an argumentation agent with scenarios and options extracted from an `Ai-Raison` service. This API receives a project identifier (`project_id`) and returns the associated scenarios and options for that project.

The API exposes the following endpoints:

- **/initialize**: Initializes the agent by retrieving the scenarios and options associated with a given project.
- **/health**: Allows you to check if the server is functioning properly.

Prerequisites

Before starting, make sure you have the following:

- **Python 3.7+**
- **Pip** (Python package manager)
- A `config.py` file containing a valid API key for the `Ai-Raison` API (the key should be in the form `api_key = 'your_api_key'`).

Installation

1. Clone this repository:

```
git clone https://your-repository-url
cd role7-argumentation-agent
```

2. Create a virtual environment (optional but recommended):

```
python -m venv venv
```

3. Activate the virtual environment:
   - On Windows:

```
venv\Scripts\activate
```

   - On MacOS/Linux:

```
source venv/bin/activate
```

4. Install the required dependencies:

```
pip install -r requirements.txt
```

5. Ensure you have a `config.py` file containing your API key, for example:

```
api_key = 'your_api_key'
```

Run the server

Start the FastAPI server with Uvicorn:

```
uvicorn role7_service:app --reload
```

This will start the API on `http://127.0.0.1:8007`.

Endpoints

1. `/initialize` (POST Method)

Initializes the agent by retrieving the scenarios and options associated with a given project.

Example Request:

POST http://127.0.0.1:8007/initialize

```
{
  "project_id": "your_project_id"
}
```

Response:

```
{
  "project_id": "your_project_id",
  "scenarios": ["Scenario 1", "Scenario 2", "Scenario 3"],
  "options": ["Option 1", "Option 2"]
}
```

2. `/health` (GET Method)

Checks if the server is healthy.

Response:

```
{
  "status": "OK"
}
```

Project Structure

The project is organized as follows:

```
/role7-argumentation-agent
│
├── main.py                # Main file with endpoint definitions
├── config.py              # Contains the API key to access the Ai-Raison API
├── requirements.txt       # List of required Python dependencies
└── README.md              # Project documentation
```

How it works

Agent Initialization:

- The API receives a `project_id` via the `/initialize` endpoint.
- This `project_id` is used to make a request to the Ai-Raison API to retrieve the associated scenarios, options, and description.

Retrieving Scenarios, Options, and Description:

- Once the data is fetched from the Ai-Raison API, it is extracted and formatted to be sent in the response.
- The response includes:
  - **Scenarios**: Labels of the scenarios.
  - **Options**: IDs of the options.
  - **Description**: Description of the service.
  - **Service Elements**: Metadata related to the service invocation (e.g., execution ID, timestamp, status).

Health Check:

- The `/health` endpoint allows you to quickly check if the server is functioning correctly.

Role 8 - Argumentation Agent

Author

Role R8, **YAYA Latifou**

📌 Purpose

This script implements **Role 8** in our multi-role system. It is responsible for integrating various argumentation modules (**R5-R8**) and ensuring seamless interaction with the **Broker Agent (R9)**. Specifically, **Role 8** is in charge of:

1. **Registering itself as a service provider** within the multi-agent system.
2. **Interacting with Role 5 (Scenario Matcher)** to retrieve relevant argumentation scenarios.
3. **Communicating with Role 6 (Problem Solver)** to derive reasoning-based decisions.
4. **Publishing its services to the Broker (R9)** to be discoverable by other agents.

The implementation ensures that user input is correctly processed and routed through the necessary argumentation components before returning a structured response.

🛠️ Project Overview

🔹 Input

For optimal functionality, **Role 8** relies on two key inputs:

- **Project Identifier (`project_id`)**:
  - A unique string used to query the **Scenario Matcher (R5)**.
  - Example: `"PRJ15875"`
- **User Message (`message`)**:
  - The natural language input provided by the user.
  - Example: `"What is the best strategy for my project?"`

These inputs are sent as JSON in a **POST request** to the `/process` endpoint.

🔹 Processing Pipeline

1. **Register with the Broker (R9)** to be available as a service provider.
2. **Retrieve matching scenarios** from **R5** based on the user request.
3. **Send the matched scenarios to R6** for reasoning-based decision-making.
4. **Return the final response** to the requesting agent (e.g., the LLM Agent).

🔹 Output

Role 8 generates a structured JSON response containing:

- `response` : The final decision or recommendation.
- `source` : Indicates that the response was generated by Role 8.

Example Output:

```
{
    "response": "Based on argumentation analysis, we recommend Strategy X for PRJ15875.",
    "source": "R8"
}
```

⚙️ Implementation Details

🔹 Registering with the Broker (R9)

Upon startup, **Role 8** registers itself as an available argumentation service.

```
def register_with_broker():
    """
    Registers R8 with the Broker Agent (R9).
    """
    payload = {
        "agent_name": "Argumentation_Agent",
        "service": "argumentation",
        "description": "Processes user input and provides argumentation-based solutions."
    }

    try:
        response = requests.post(BROKER_API_URL, json=payload)
```

```
        response.raise_for_status()
        print("[INFO] R8 registered successfully with the Broker.")
    except requests.exceptions.RequestException as e:
        print(f"[ERROR] Failed to register R8 with the Broker: {e}")
```

🔹 Fetching Argumentation Scenarios from R5

Role 8 sends a request to **R5** to retrieve relevant argumentation scenarios.

```
def call_role5(project_id: str, message: str) -> list:
    """
    Requests matching scenarios from R5.
    """
    payload = {"project_id": project_id, "user_input": [message]}

    try:
        response = requests.post(ROLE5_API_URL, json=payload)
        response.raise_for_status()
        return response.json()["matched_scenarios"]
    except requests.exceptions.RequestException as e:
        raise HTTPException(status_code=500, detail=f"Error in R5: {e}")
```

🔹 Requesting Argumentation-Based Decisions from R6

Once relevant scenarios are obtained from **R5**, they are sent to **R6** for reasoning.

```
def call_role6(project_id: str, matched_scenarios: list, user_input: list) -> str:
    """
    Requests a reasoning-based decision from R6.
    """
    payload = {
        "project_id": project_id,
        "user_input": user_input,
        "matched_scenarios": matched_scenarios,
        "info": "Generated by R8"
    }

    try:
        response = requests.post("http://localhost:8006/find_solution", json=payload)
        response.raise_for_status()
        return response.json()["text"]
    except requests.exceptions.RequestException as e:
        raise HTTPException(status_code=500, detail=f"Error in R6: {e}")
```

🔹 Processing Requests

This function serves as the main entry point for handling user requests.

```
@app.post("/process")
def process_argumentation_request(request: ArgumentRequest):
    """
    Handles an argumentation request by:
    1. Fetching relevant scenarios from R5.
    2. Sending those scenarios to R6 for a reasoning-based decision.
    3. Returning the final response.
    """
    matched_scenarios = call_role5(request.project_id, request.message)

    if not matched_scenarios:
        return {"response": "No matching scenarios found.", "source": "R8"}
```

```
    solution = call_role6(request.project_id, matched_scenarios, [request.message])

    return {"response": solution, "source": "R8"}
```

🚀 **Developed by YAYA Latifou** | Multi-Agent Argumentation System | 2025

Broker Agent – Matchmaking Service

Author

R9- Beddouihech Maram

Description

This project provides an API built with FastAPI that matches user prompts to available services based on cosine similarity. It utilizes `TF-IDF` (Term Frequency-Inverse Document Frequency) and `cosine similarity` to compare a user's request to the descriptions of available services.

The API exposes a matchmaking endpoint where users can submit a prompt, and the system will return a list of services that match the prompt based on their descriptions.

Prerequisites

Before starting, make sure you have the following:

- **Python 3.7+**
- **Pip** (Python package manager)

Installation

1. Clone this repository:

```
git clone https://your-repository-url
cd argumentation-agent-matchmaking
```

2. Create a virtual environment (optional but recommended):

```
python -m venv venv
```

3. Activate the virtual environment:

   - On Windows:

```
venv\Scripts\activate
```

- On MacOS/Linux:

```
source venv/bin/activate
```

4. Install the required dependencies:

```
pip install -r requirements.txt
```

Run the server

Start the FastAPI server with Uvicorn:

```
uvicorn role9_service:app --reload
```

This will start the API on `http://127.0.0.1:8009`.

Endpoints

1. `/matchmaking` (POST Method)

This endpoint receives a user prompt and returns a list of services that match the prompt based on cosine similarity.

Example Request:

```
POST http://127.0.0.1:8009/matchmaking
Content-Type: application/json

{
  "user_prompt": "Describe the available options for argumentation agents."
}
```

Response:

```
{
  "status": "success",
  "matched_services": {
    "project_id_1": {
      "description": "A detailed description of service 1.",
      "scenarios": ["Scenario 1", "Scenario 2"],
      "similarity_score": 0.75
    },
    "project_id_2": {
      "description": "A detailed description of service 2.",
      "scenarios": ["Scenario 3", "Scenario 4"],
      "similarity_score": 0.68
    }
  }
}
```

If no matching services are found:

```
{
  "status": "No matching services found."
}
```

2. `/health` (GET Method)

Checks if the server is healthy.

Response:

```
{
  "status": "OK"
}
```

How It Works

Agent Initialization:

- The API receives a `user_prompt` via the `/matchmaking` endpoint.
- This `user_prompt` is used to match against service descriptions from available services.

Retrieving Scenarios, Options, and Description:

- Once the data is fetched, it is compared using **TF-IDF vectorization** and **cosine similarity**.
- The response includes:
  - **Description**: The description of the matching service.
  - **Scenarios**: The scenarios related to the service.
  - **Similarity Score**: The cosine similarity score between the user prompt and service description.

Health Check:

- The `/health` endpoint allows you to quickly check if the server is functioning correctly.

Role 11 - State of the Art and Deployment Process

Author

Role 11 ; Cheikh Tidiane DIOUF

State of the Art of the Technologies

1. FastAPI (Web Framework)

FastAPI is a modern and fast web framework for building RESTful APIs in Python. It is built on top of **Starlette** and **Pydantic**, designed for high performance while being easy to use. Here are the key points regarding FastAPI:

- **Performance**: FastAPI is one of the fastest Python frameworks for creating APIs, surpassing even established frameworks like **Flask** and **Django** in some cases, largely due to its use of **asyncio** for asynchronous calls and routing via **Starlette**.
- **Type Hints and Automatic Validation**: With **Python type annotations**, FastAPI automatically generates interactive API documentation (via **Swagger UI**) and performs data validation with **Pydantic**. This ensures that incoming data is validated upon arrival.
- **Ease of Use**: The combination of automatic documentation, data validation, and minimal code to create API routes greatly simplifies service development.
- **Interoperability**: It integrates easily with other Python ecosystem technologies, making it a great choice for microservices and distributed architectures.

2. Hugging Face Transformers (Language Models)

Language models like those developed by **Hugging Face** are based on the **Transformer architecture**. These models are used for processing natural language, including text understanding, generation, and translation. The use of these models in the project is crucial for performing intelligent and contextual text processing. Key features of **Hugging Face Transformers**:

- **Transformer Architecture**: The Transformer architecture relies on attention mechanisms to efficiently handle dependencies between words in a sequence, regardless of their position in the sentence. This mechanism allows the model to process long text sequences without losing context, making it particularly powerful for translation, generation, and understanding tasks.
- **Pre-trained Models**: Hugging Face provides a vast library of pre-trained models like **GPT**, **BERT**, **T5**, etc., which are ready to be used and fine-tuned for specific tasks. These models are fine-tuned for a variety of Natural Language Processing (NLP) tasks, significantly reducing training time.
- **Easy Integration**: The **Transformers API** makes it easy to integrate language models into Python applications. Users can directly load models and tokenizers, enabling fine-tuning for specific use cases.
- **Performance and Scalability**: Transformer-based models are often computationally heavy, but they deliver top-tier results, making them ideal for cutting-edge AI applications. In this project context, these models are used to understand user requests and generate relevant responses.

3. Pydantic (Data Validation)

Pydantic is a Python library used for data validation. It is widely used in **FastAPI** to automatically perform consistent and reliable validation of incoming data. Key features of **Pydantic**:

- **Type and Structure Validation**: Pydantic allows developers to define data models using standard Python type annotations. It automatically validates incoming data based on these types, making error detection easy.
- **Performance**: Pydantic is designed for speed and efficiency, allowing the validation of thousands of records per second with precise results.
- **Error Handling**: When data doesn't match the expected format (e.g., incorrect type or missing values), Pydantic generates clear error messages, which helps developers quickly identify and fix issues.

4. Regular Expressions (Regex)

Regular expressions (Regex) are used to search, extract, and manipulate strings based on specific patterns. They are particularly useful in contexts where data is not strictly structured. In the project, regex plays a crucial role in extracting relevant information from results generated by language models.

- **Extracting Unstructured Data**: Regex allows for the identification and extraction of text subsets or patterns within complex strings, such as results generated by a language model or unformatted JSON responses.
- **Format Verification**: Regex is also used for validating or filtering certain formats, like project IDs or dates, improving the accuracy of the data processing.

5. Docker (Containerization)

Docker is a tool for creating isolated and portable environments for applications. It is particularly useful in microservice architectures like this project.

- **Portability**: Dockerized services can be deployed on any environment that supports Docker, making deployment and dependency management easier.
- **Service Isolation**: Docker helps separate services, ensuring they don't interfere with each other, which is crucial in complex environments where multiple services are running concurrently.

Justification of Technology Choices

Why FastAPI?

FastAPI was chosen for its speed and high performance, which is essential in a real-time service environment like this. Its ease of use, automatic validation with **Pydantic**, and automatic documentation generation via **Swagger UI** make it an ideal choice for fast and efficient development.

Why Hugging Face and Transformer Models?

Transformer-based models, like those from **Hugging Face**, are highly effective at processing long and complex text sequences, which makes them particularly suited for handling user queries and generating contextually relevant responses. Their pre-training on vast amounts of data allows them to capture rich contexts and respond accurately to user demands.

Why Pydantic for Data Validation?

The use of **Pydantic** ensures that all incoming data to the API is automatically validated before processing. This guarantees that the service works with reliable and consistent data, reducing the likelihood of errors due to invalid inputs.

Why Use Regular Expressions?

Regular expressions are a quick and efficient way to extract specific information from unstructured text. They are particularly useful for filtering or analyzing results generated by language models, which may be imprecise or contain additional information.

Platform installers and launchers

Author

Role R12; Florian Posez

General Description

Installer script

The installer Python script automates the setup of a virtual environment and the installation of dependencies for multiple project components. It performs the following tasks:

1. **Create and Activate a Virtual Environment**
    - If a `venv` directory does not exist, the script creates a virtual environment using `python3 -m venv venv`.
2. **Install Dependencies from `requirements.txt`**

- The script scans all subdirectories that follow the `R<n>` naming pattern (e.g., `R1`, `R2`, `R3`).
- If a `requirements.txt` file is found within a directory, the script installs the required dependencies using `pip`.

3. **Execute Bash Installation Scripts (`install.sh`)**

   - In each `R<n>` directory, if an `install.sh` script exists, the script executes it using the `bash` command.
   - This ensures that any additional setup required by individual components is completed.

4. **Completion Message**

   - After executing all necessary installations, the script notifies the user that the setup process is complete and prompts them to activate the virtual environment before running the project.

Launcher script

The launcher script automates the launching of all microservices by scanning all the `R<n>` directories and executing the `launch.sh` scripts if founded

---