R1 LLM Service

This project provides a FastAPI-based API that leverages a Mistral-based Large Language Model (Nous-Hermes-2-Mistral-7B-DPO) for text generation. It supports multi-turn conversations by persisting session history in a PostgreSQL database managed with Docker Compose.

Features

- **FastAPI Backend:** Provides endpoints for health checking and text generation.
- **LLM Integration:** Uses Hugging Face's Transformers to load and run the Mistral model.
- **Session Management:** Stores conversation history in PostgreSQL using SQLAlchemy.
- **Dockerized Database:** Easily set up and run PostgreSQL with Docker Compose.

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
}
```

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

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:**</p>

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.

Broker Agent – Matchmaking Service

Author

Beddouihech Maram

General Description

The Broker Agent – Matchmaking Service is used for routing requests to appropriate argumentation agents based on their advertised capabilities. This microservice employs text analysis techniques to match user inquiries with the most suitable services offered by argumentation agents. It interacts with the rAIson platform to ensure efficient and accurate service matching.

Installation

Clone the Repository

```bash git clone https://github.com/yourusername/aose-2025.git cd R9
```