Documentation – Sarra Ben Brahim

API documentation for an endpoint that accepts a query parameter and uses the GET method:

GET /api/similar_queries

Returns a list of similar queries to the specified query.

Request

GET /api/similar_queries

Query params :

Parameter	Type	Required	Description
'query'	string	yes	The query to find similar queries for.

Response

Returns a JSON array of strings representing similar queries to the specified query with a user message.

Example :

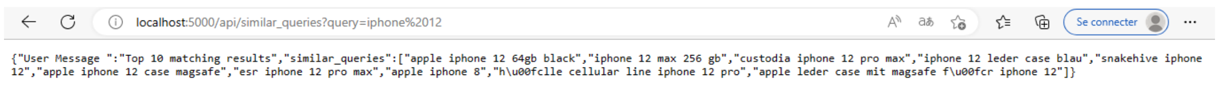
GET /api/similar_queries?query=iphone 12

Response:

HTTP/1.1 200 OK

Content-Type: application/json

{"User Message ":"Top 10 matching results","similar_queries":["apple iphone 12 64gb black","iphone 12 max 256 gb","custodia iphone 12 pro max","iphone 12 leder case blau","snakehive iphone 12","apple iphone 12 case magsafe","esr iphone 12 pro max","apple iphone 8","h\u00fclle cellular line iphone 12 pro","apple leder case mit magsafe f\u00fcr iphone 12"]}



```
{"User Message ":"Top 10 matching results","similar_queries":["apple iphone 12 64gb black","iphone 12 max 256 gb","custodia iphone 12 pro max","iphone 12 leder case blau","snakehive iphone
12","apple iphone 12 case magsafe","esr iphone 12 pro max","apple iphone 8","h\u00fclle cellular line iphone 12 pro","apple leder case mit magsafe f\u00fcr iphone 12"]}
```

Error Responses :

Status Code	Description
400	Bad Request.The « query » param is missing.

Usage :

To use this API endpoint, send a GET request to /api/similar_queries with the query parameter set to the query string you want to find similar queries for.

For example, to find similar queries for the word "iphone 12", you would send a GET request to **/api/similar_queries?query**=iphone 12. The response will be a JSON array of strings representing similar queries to the word "iphone 12".

The work is basically divided on 2 scripts :

1. **Script n°1 : Model.py**
2. **Script n°2 : App.py**

   I will detail in this part what both scripts do :

   Script n°1 : Model.py :

The code is a script for preprocessing and training a machine learning model to predict the relevance score of search queries.

The relevance score is value between [0-1] that represents the relevance of a query.

It is calculated based on the heuristic : multiplication of three given values :

- **success_rate [0-1]**
- **conversion_rate [0-1]**
- **searches per month normalized [0-1]**

  It is then normalized.

  This will help us in the application for ranking the queries.

  (See Script n°2 description)

Here's a detailed documentation of the code:

Importing libraries

The script starts by importing the required libraries: nltk, scipy, re, pandas, numpy, sklearn, joblib, and pickle.

Try-except block

Next, the code includes a try-except block to handle any exceptions that might occur during the import of libraries. It also downloads the necessary nltk resources.

Tokenizer

The code defines a tokenizer object to tokenize the text data.

Text Preprocessing

The process_text() function removes stop words from the text data in German and English languages. The clean_dataset() function uses the tokenizer object to tokenize the data and then applies the process_text() function to clean the text data. It also calculates a relevance score for each search query based on a heuristic and adds this as a new column to the DataFrame.

Data Splitting

The split_data() function splits the dataset into training (80%) and testing (20%) sets. It returns the queries and relevance scores of the training and testing sets.

Feature Engineering

The feature_engineering() function calculates the TF-IDF (term frequency-inverse document frequency) for the training and testing queries and stores them in sparse matrices.

Model Training

The train_model() function trains the model to predict the relevance score of search queries. It uses the Lasso model with hyperparameters alpha and iterations. The trained model is saved as a joblib file.

Model Evaluation

The evaluate_model() function evaluates the model performance using the test set based on mean square error and root mean square error metrics.

Model Testing

The test_new_query() function tests the model on a new query and calculates the predicted relevance score for the new query.

Overall

Finally, the code executes the functions in the following order: clean_dataset(), split_data(), feature_engineering(), and train_model(),evaluate_model(),test_new_query() .

The preprocessed dataset, trained model, and vectorizer object are saved as CSV, and pickle files, respectively.

Script n°2 : app.py :

This is a Python script that defines a Flask web service for finding the top 10 most similar queries to a given input query. The service loads preprocessed data, a trained model, and a vectorizer from saved files, and uses them to calculate the similarity scores between the input query and the queries in the dataset. The service returns the top 10 most similar queries in JSON format.

The **load_data()** function loads the preprocessed data, trained model, and vectorizer from saved files. It returns a pandas DataFrame df containing the original queries, a pandas Series **relevance_scores** containing the relevance scores of the preprocessed queries, a trained model relevance_model for predicting relevance scores, a vectorizer vectorizer for transforming queries into a vector space representation, and a sparse matrix tfidf_matrix representing the vector space representation of the original queries.

The **index()** function returns a simple message to confirm that the service is running.

The **get_similar_queries()** function is the main function that finds the top 10 most similar queries to a given input query. It first gets the input query from the query string in the URL. If no query is given, it returns a JSON object with an error message and an empty list of similar queries. It then transforms the input query into a vector space representation using **the vectorizer**, and predicts a relevance score for the query using **the relevance model**. It then finds the indices of the queries in the dataset that have a relevance score greater than or equal to the predicted relevance score.

It then extracts the relevant queries and their vector space representations from the preprocessed data and the sparse matrix, respectively. It then calculates the cosine similarity scores between the input query and the relevant queries, and returns the top 10 most similar queries in a JSON object.

The **if __name__ == "__main__":** block at the end of the script starts the Flask web service on port 5000.

The explaination of such approach :

Filtering the queries based on their predicted relevance score before computing the cosine similarity is a useful approach.

This allows to limit the number of queries that you need to compare with the input query, potentially reducing the computation time and improving the efficiency of the system.

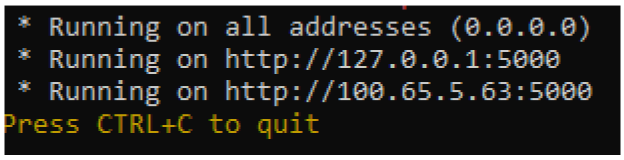
So , we first predict the relevance score of the input query using a machine learning model, and then filter out all the queries that have a lower relevance score than the predicted relevance score of the input query.

Second compute the cosine similarity between the remaining queries and the input query, and return the top similar queries based on the similarity score.

This approach help to reduce the noise in the results and improve the accuracy of the system, especially when dealing with large datasets or when the computation time is a constraint.

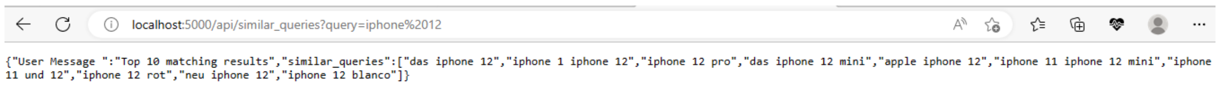
Steps to run the script :

1. Open command promt and run : **python app.py**
   ⇨ This will show up in the cmd :

   ```
    * Running on all addresses (0.0.0.0)
    * Running on http://127.0.0.1:5000
    * Running on http://100.65.5.63:5000
   Press CTRL+C to quit
   ```

   You can then run in the browzer : **http://127.0.0.1:5000**
2. You run in the browzer :
   http://127.0.0.1:5000/api/similar_queries?query=iphone 12
   ⇨ This will show up in the browzer :



localhost:5000/api/similar_queries?query=iphone%2012

```
{"User Message ":"Top 10 matching results","similar_queries":["das iphone 12","iphone 1 iphone 12","iphone 12 pro","das iphone 12 mini","apple iphone 12","iphone 11 iphone 12 mini","iphone 11 und 12","iphone 12 rot","neu iphone 12","iphone 12 blanco"]}
```

How to run the docker container ?

1. **Make sure that the dockerfile and requirements.txt and all the projects files are running in the same directory.**
2. **Build the Docker image by running the following command in your terminal:**
   docker build -t your_image_name .
3. **Once the image is built, you can start the container with the following command:**
   run docker run -p 5000:5000 your_image_name

Questions :

How did you calculate the similarity of the queries and why did you choose this method? What are its limitations?

➔I used cosine similarity to calculate the similarity between the input query and queries in the dataset.
Using cosine similarity is a common method for measuring the similarity between two vectors in a high-dimensional space. In the case of text data, the vectors are typically representations of the documents or queries in a vector space model, such as TF-IDF. Cosine similarity is a good choice for measuring text similarity for the following reasons:

⇨ It is a widely used and well-understood metric for measuring vector similarity.

⇨ It is computationally efficient and can be applied to large datasets. It takes into account the relative magnitude of the vectors, not just their orientation, which makes it suitable for comparing text documents of different lengths.

However, there are **some limitations** to using cosine similarity for text similarity:

- It does not take into account the meaning or context of the words in the text. This means that two documents with similar word frequencies may not necessarily be semantically similar.

- It is sensitive to the presence of common words that are not informative for similarity, such as stopwords. This can lead to misleading results if not properly handled.
- It assumes that each word in the text is independent and equally important, which may not be the case in all situations.

How would you quantify how well the similarity service is doing? How could it be improved ?

There are several ways to quantify how well the similarity service is doing:

- **Precision, Recall and F1-Score**: Precision measures the proportion of retrieved results that are relevant, while recall measures the proportion of relevant results that are retrieved. F1-score is the harmonic mean of precision and recall. A higher precision, recall or F1-score indicates better performance.

To improve the similarity service, here are some suggestions:

- Use a larger and more diverse dataset: A larger and more diverse dataset would improve the coverage of the service, and allow it to handle a wider range of queries.
- Experiment with different similarity metrics: The service currently uses cosine similarity, which is a widely used metric for text similarity. However, there are other similarity metrics that may perform better for certain types of queries. Experimenting with different similarity metrics could lead to better performance.
- Fine-tune the model: The service currently uses a Lasso model for predicting relevance scores. Fine-tuning the model, or trying different models altogether, could lead to better performance.
- Use deep learning models: Deep learning models such as neural networks have shown promising results for text similarity tasks. Using deep learning models could improve the performance of the service.
- Use query expansion: Query expansion involves adding more terms to a query to improve the chances of retrieving relevant results. Using query expansion techniques such as synonym replacement or stemming could improve the performance of the service.