6111 Advanced Database Project 1

Query expansion using the Rocchio algorithm

Authors

- Evan Ting-I Lu (tl3098)
- Pitchapa Chantanapongvanij (pc2806)

Files

1. `project1.py` (main program)
2. `README`
3. `transcripts.txt`

How to run our program

I. If you have access to our Google VM (tl3098@cs6111-instance)

1. Get to the home directory of our VM account, i.e., you should be seeing in the terminal `tl3098@cs6111-instance:~$`
2. We use Anaconda to manage our virtual environments. The default environment is (base). Please use `conda info --env` to list all the environments, and use `conda activate 6111_ADB_proj_1` to switch to the environment we prepared for this project.
3. `cd` to the folder of the main program with `cd ~/SP22-6111-ADB-Projects/project\ 1/`
4. Run the program with `python3 project1.py`

II. If you are setting up a new environment

1. Install the necessary packages:
   - python (we use 3.9.7 since there are some version issue for scikit-learn with python 3.8 when using with Anaconda)
   - google-api-python-client (2.37.0)
   - numpy (1.21.2)
   - scipy (1.7.3)
   - scikit-learn (>= 1.0.2, otherwise we found an issue with `get_feature_names_out()` method)

2. Copy our main program, i.e., `project1.py` to your environment.

3. Run it with Python 3

Description

General Structure

External Libraries

We used the following external libraries/packages:

- googleapiclient (retrieving search results from google)
- scikit-learn (Tfidfvectorizor method for calculating tf-idf scores)
- numpy (performing vector manipulation functions)
- pprint

High-level components

Our python program (project1.py) can be broken down into a main function, SmartQuery class, and its 5 class methods:

- *_init_*
- setter_query
- initiate_query
- get_user_feedback
- update_query

Methods

`Main()` :
Calls SmartQuery class and extracts initial query and target precision from users. The main method controls every iteration of the query-modification process by comparing the current precision to the target precision. If the target precision is met, it will output the feedback summary (original query, final precision, and that the desired precision is reached) to the user. More specifically, inside the looping process, the main function will be iteratively updating the query by calling `update_query()`. Following this, it will call `initiate_query()` to get the top 10 search results, call `get_user_feedback()` to get feedback on relevance, and repeat the process until the target precision is met.

`SmartQuery()`:
Object constructor that contains 5 methods including `__init__`, `setter_query()`, `initiate_query()`, `get_user_feedback()`, `update_query()`. These methods are for initializing all variables, assigning new query to current query, initating queries, outputting search results & getting relevance feedback from users, and updating query respectively.

`Initializer`:
Contains set variable and attribute values. This includes API key, search engine ID, Rocchio algorithm parameters, TF-IDF vectorizer parameters, and more. We made them fixed value in our program. However,

for future salability we can easily record these values in a `.yaml` file and import them at the time of `SmartQuery` instance initialization.

`Setter_query()`:
Takes in a new query string and updates the current query to the new value.

`Initiate_query()`:
Starts a query using Google JSON API with the current query value and returns the top 10 query result in its raw format which is a Python dictionary.

`Get_user_feedback()`:
Output top 10 query results to the user and record user feedback on relevant into a list for internal bookkeeping. Finally, it will return the current precision.

`Update_query()`:
Uses the Rocchio algorithm to calculate relevance and update query accordingly (top 2 words with the highest score will be added to the query).

Query Modification (Expansion) Process

We implemented the Rocchio's Algorithm for query modification/expansion. The necessary calculations are implemented in the `update_query()` method. After getting feedback from users on the top 10 search results, this function is reponsible for updating the query. We used scikit-learn method `TfidfVectorizer()` to convert the whole corpus into tf-idf score vector space. Much like other ML methods in *scikit-learn*, we use the method to contruct an object, then use the `fit_transform()` method to fit it on our data, i.e., corpus. The corpus is simply the top 10 search results' **titles** and **summaries**. The converted tf-idf score matrix is stored in the `doc_tfidf`. Similarly, we convert the query into vector space using the same `vectorizor`; each entry is the tf-idf score for a query term.

Next, we use two `list`s to keep track of the corresponding indices of relevant/irrelevant queries in the `doc_tfidf` score matrix. We then use the **Rocchio algorithm**'s formula to update the query:

$$\overrightarrow{Q_m} = \left(a \cdot \overrightarrow{Q_o}\right) + \left(b \cdot \frac{1}{|D_r|} \cdot \sum_{\overrightarrow{D_j} \in D_r} \overrightarrow{D_j}\right) - \left(c \cdot \frac{1}{|D_{nr}|} \cdot \sum_{\overrightarrow{D_k} \in D_{nr}} \overrightarrow{D_k}\right)$$

credit: wikipedia

After updating the query vector, which result we store in `new_query_vec`, we use numpy `argsort()` to get the indices of the top scoring words in decending order and store the result in the `idx` array.

Finally, we use the result in `idx` to retrieve the top 2 scoring words to expand the query (note that `idx` contains the indices of the words in descending order, and that we get the list of all words in the 10 search results by calling `TfIdfVectorizer`'s `get_feature_names_out` method, which we store it in a list variable `words`.

We make sure that the 2 words with the highest score are not already in the current query. If it is, we skip and retrieve the next word with highest score. We perform the uniqueness checking and then call `SmartQuery`'s `setter_query` method to update the query. Also note that we append the highest and second-highest words to the query consecutively. Therefore, in each iteration, the query word order will naturally be in descending score order from highest to lowest score.

Credentials

- Google Custom Search Engine JSON API Key

> AIzaSyBBi7WOxVf_3waDwBkM_roXIMaceDCsVog

- Engine ID:

> 91b445df8028cd711

Additional Information

For Rocchio algorithm's parameters, we referenced online resources and found there has been some commonly used settings. Particularly, we referenced this Kaggle Project: https://www.kaggle.com/jerrykuo7727/rocchio