Mireia Pou Oliveras 251725
Iria Quintero García 254373
Javier González Otero 243078

Part 4: User Interface and Web Analytics

Github link:
TAG:

Introduction

The objective of this project is to develop a search engine.

After dealing with the ranking process, we now need to create the user interface and then apply some web analytics on it. For that, we will divide this last part of the project into the following subsections:

- Part 1: User interface

In Part 1, we create a user interface that, given an input query, displays the list of documents found for the query in the calculated ranking.

- Part 2: Web Analytics

In Part 2, we create a mechanism to track and analyze how people use the website and the search engine.

1. User interface

1.1 Search Page, Search Action, Search Function and Search Algorithms

The first part of the user interface consists in creating the search engine. In order to do it, we need to follow a pipeline.

The first thing to do is to create the **search page** itself. This requires creating a main web page with a central search box where users can enter a query and a button to execute the search. Once this is done, the next step is to implement the **search action**. That is, when the query is entered and the box clicked, the text needs to be sent to the search engine's function. This **search function** receives as parameter the query and calls the necessary **search algorithms** to retrieve the results that are better, faster, cleaner, and that better suit the user's information needs.

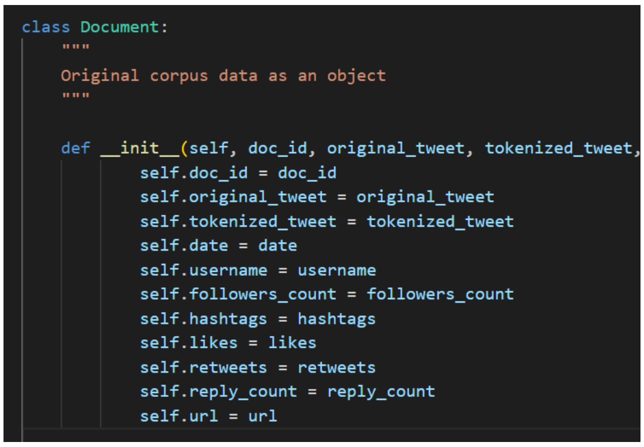
To achieve this configuration, we:

1. Used the skeleton uploaded in the Moodle, which already included the search page and the search action done.
2. Modified the search function, so that it calls our algorithm:

```python
ranked_result_items = search_tf_idf(search_query, self.index, self.corpus,
self.idf, self.tf)
```

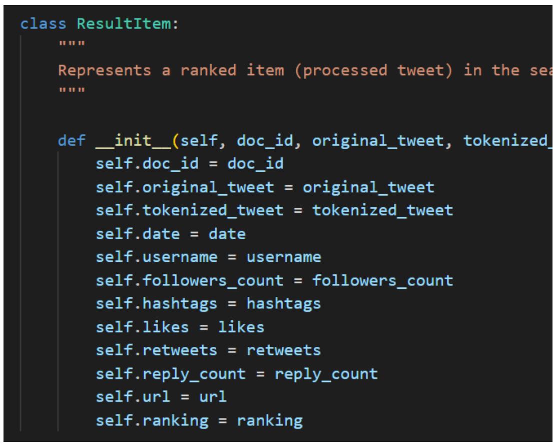
3. Added the necessary algorithms to the *algorithms.py* page. The functions we added were: build_terms, create_index_tfidf, rank_documents and search_tf_idf.

Once the functions were uploaded, we needed to modify them because the search engine works with **classes**, which can be found in the *objects.py* page. We first changed the classes so that they included the necessary parameters of the tweet:



```python
class Document:
    """
    Original corpus data as an object
    """

    def __init__(self, doc_id, original_tweet, tokenized_tweet,
            self.doc_id = doc_id
            self.original_tweet = original_tweet
            self.tokenized_tweet = tokenized_tweet
            self.date = date
            self.username = username
            self.followers_count = followers_count
            self.hashtags = hashtags
            self.likes = likes
            self.retweets = retweets
            self.reply_count = reply_count
            self.url = url
```



```python
class ResultItem:
    """
    Represents a ranked item (processed tweet) in the sea
    """

    def __init__(self, doc_id, original_tweet, tokenized_
        self.doc_id = doc_id
        self.original_tweet = original_tweet
        self.tokenized_tweet = tokenized_tweet
        self.date = date
        self.username = username
        self.followers_count = followers_count
        self.hashtags = hashtags
        self.likes = likes
        self.retweets = retweets
        self.reply_count = reply_count
        self.url = url
        self.ranking = ranking
```

Consequently, we adapted the functions so that they could work with these classes instead of working with jsons. The main changes were:

1. Create an index from a corpus (document class) instead of jsons.
2. Return ResultItems when ranking the documents, instead of the ranks by themselves.

1.2 Results Page and Document Details Page

The second part of the user interface consists in creating the interface that the users will search on. After making the search engine, the idea is to create a web page that displays the list of documents found for the query and in the calculated order/ranking.

In order to do it, we first modified the *web_app.py* in the following way:

```python
full_path = os.path.realpath(__file__) #get current path
path, filename = os.path.split(full_path)
file_path = path + "/processed_tweets.json"
```

Mireia Pou Oliveras 251725
Iria Quintero García 254373
Javier González Otero 243078

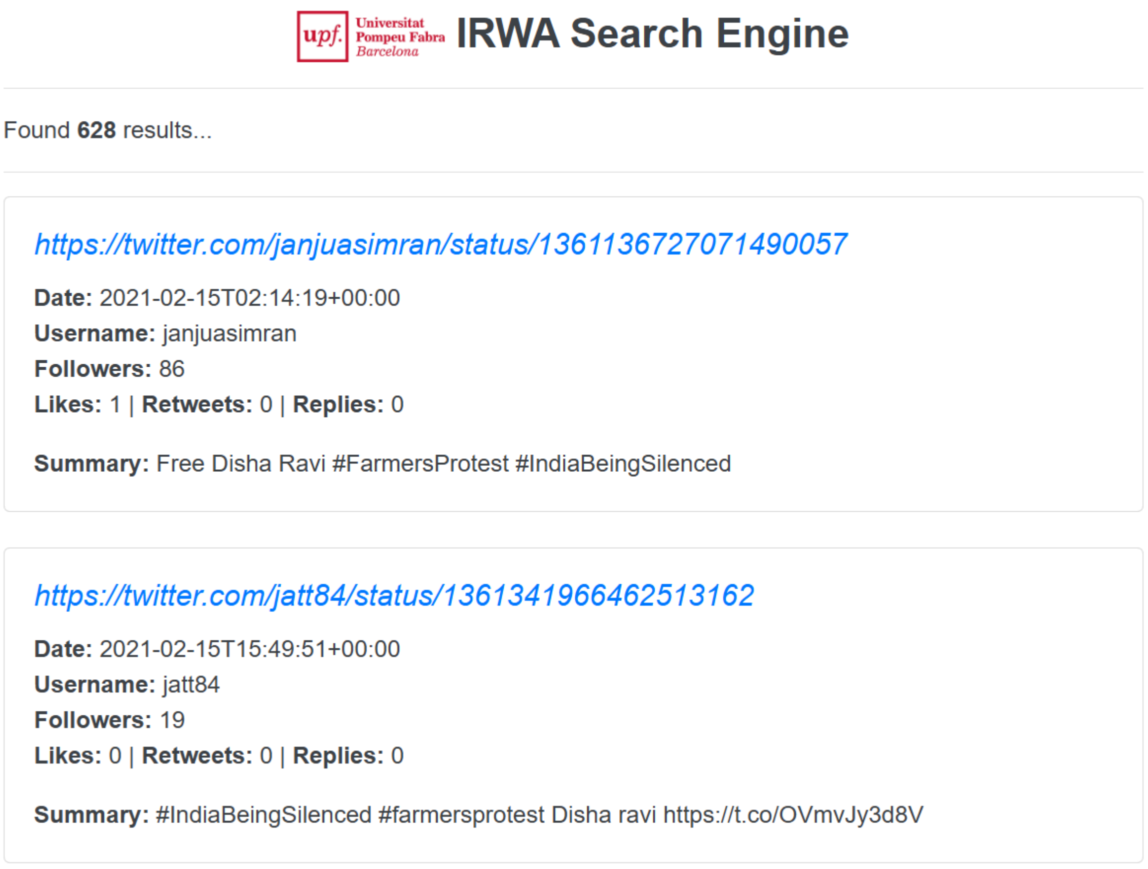
```
corpus = load_corpus(file_path)
# instantiate our search engine
search_engine = SearchEngine(corpus=corpus)
```

That is, we adapted the code so that it gets our jsons and then instantiated the previously created Search Engine.

Once this was working, we modified the *results.html* file so that apart from displaying the url of the tweet, it displayed it document in a square and it shows the next information:

- Date that the tweet was created
- Username of the person that uploaded the tweet
- Followers of the user
- Likes of the tweet
- Number of retweets the tweet has
- Number of replies the tweet has
- Summary: Display the original content of the tweet.

The final result has this format:


IRWA Search Engine

Found **628** results...

https://twitter.com/janjuasimran/status/1361136727071490057

Date: 2021-02-15T02:14:19+00:00
Username: janjuasimran
Followers: 86
Likes: 1 | **Retweets:** 0 | **Replies:** 0

Summary: Free Disha Ravi #FarmersProtest #IndiaBeingSilenced

https://twitter.com/jatt84/status/1361341966462513162

Date: 2021-02-15T15:49:51+00:00
Username: jatt84
Followers: 19
Likes: 0 | **Retweets:** 0 | **Replies:** 0

Summary: #IndiaBeingSilenced #farmersprotest Disha ravi https://t.co/OVmvJy3d8V

2. Web Analytics

2.1. Data Collection

For being able to store the analytics data we expanded the AnalyticsClass so it could capture diverse information about the user interactions (search queries, document clicks, session details, HTTP request metadata). The main functionalities are the following:

- Initialization: Initialize several attributes (queries, fact_clicks (track clicks on documents), session (containing data such as user-agent and IP addresses of each individual user session) and requests (a dictionary for tracking HTTP requests storing information like URL paths and methods).
- Recording data: AnalyticsClass provides methods for recording different types of data, such as *save_query(query_term)* for saving a search query to the queries list, *record_click(doc_id, search_query, rank)* for logging a click on a document, *record_session_activity(activity_type, query)* for recording a session activity and *log_http_request(request_type, url_path, request_method)* for logging HTTP request details.
- Exporting method *to_json()* to collect analytics data as JSON object for easier storing and visualizing.

2.2 Data Storage

In web_app.py AnalyticsData class is used for tracking and managing interactions that happen between the user and the web application. It is integrated in the following way:

1. Session tracking: When a user visits the homepage of the web app (on the / route), the app captures details about the user's session. It retrieves the user's IP address and user-agent (information about the browser and device) and initializes a session using the AnalyticsData class. The *start_session()* method of AnalyticsData is used here to record this session data, associating it with the current user.
2. Search Query Handling: When a user submits a search query (via the /search route), the search term is saved in the queries list using the *save_query(query_term)* method of the AnalyticsData class. The app processes the search request and returns the relevant search results while each query is tracked to understand what users are searching for.
3. Document Clicking Tracking: After displaying the search results, when a user clicks on a document (via the /doc_details route, which leads you to the tweet url), the r*ecord_click(doc_id, search_query, rank)* method is called, so it logs which document was clicked, along with the search query and the ranking of the document in the results. This helps in analyzing user behavior, such as which documents are most frequently clicked or how users interact with the search results.
4. Session Activity Recording: The app can also log detailed session activity using the *record_session_activity(activity_type, query)* method. For example, it might log when a user submits a search query or clicks on a document. These activities are tracked and saved in fact_clicks, which stores click counts and the relevant metadata for each document.

5. HTTP Request Tracking: The *log_http_request(request_type, url_path, request_method)* method is used to capture details about the HTTP requests made by the user. This includes the URL path the user is visiting and the method (GET, POST, etc.), which is valuable for understanding how users navigate through the app.
6. Exporting Data: After data collection, the analytics can be exported using the to_json() method, which generates a JSON object of the collected data. This can be used for further analysis, debugging or, in this case, visualization purposes.
7. Displaying Data: The app also offers a /dashboard route, which serves as an analytics dashboard. This view can display user interaction data, such as the most popular search queries or the most clicked documents, which are sourced from the AnalyticsData instance and visualized through the *dashboard.html* file.

2.3 Analytics Dashboard

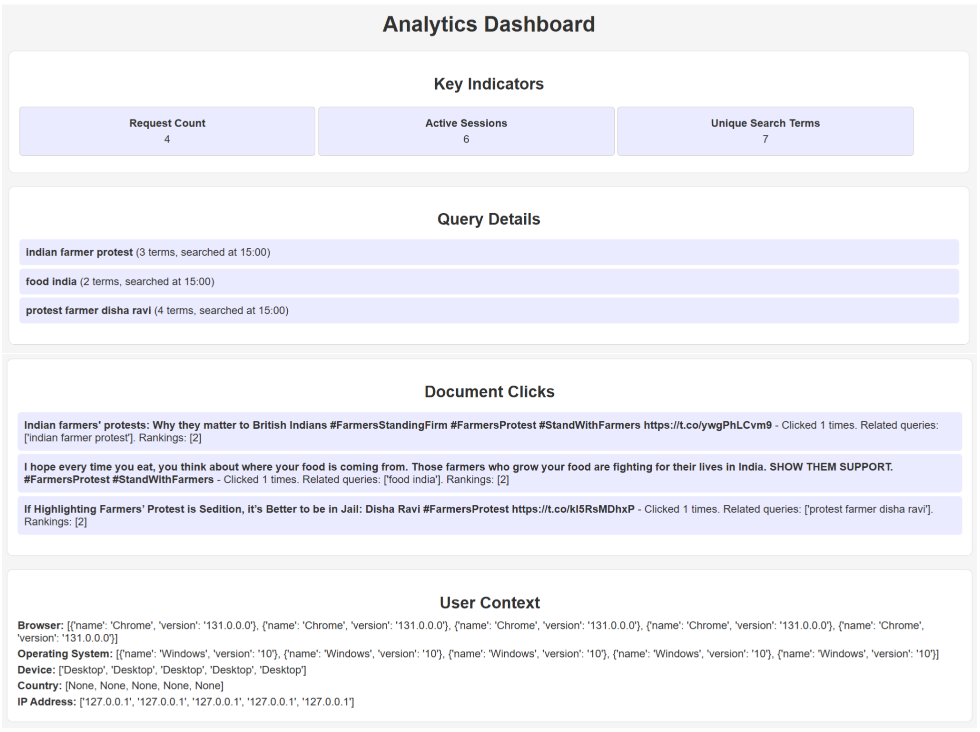
To show the analytics we have collected in the previous two points, we have modified the *dashboard.html* file.

First, we added some style so that the dashboard was visually more appealing. Overall, in this style section, we changed:

- The global styling (body)
- The header styling (to center the headers h1,h2)
- The section styling (to style the different sections of the dashboard)
- The boxes (we styled boxed that display the metrics)
- The lists and paragraphs.

Second, we added the metrics we found more important, such as request count, document clicks, query details, etc.

After changing this, the result is the next:



Analytics Dashboard

Key Indicators

Request Count	Active Sessions	Unique Search Terms
4	6	7

Query Details

indian farmer protest (3 terms, searched at 15:00)

food india (2 terms, searched at 15:00)

protest farmer disha ravi (4 terms, searched at 15:00)

Document Clicks

Indian farmers' protests: Why they matter to British Indians #FarmersStandingFirm #FarmersProtest #StandWithFarmers https://t.co/ywgPhLCvm9 - Clicked 1 times. Related queries: ['indian farmer protest']. Rankings: [2]

I hope every time you eat, you think about where your food is coming from. Those farmers who grow your food are fighting for their lives in India. SHOW THEM SUPPORT. #FarmersProtest #StandWithFarmers - Clicked 1 times. Related queries: ['food india']. Rankings: [2]

If Highlighting Farmers' Protest is Sedition, it's Better to be in Jail: Disha Ravi #FarmersProtest https://t.co/kl5RsMDhxP - Clicked 1 times. Related queries: ['protest farmer disha ravi']. Rankings: [2]

User Context

Browser: [{'name': 'Chrome', 'version': '131.0.0.0'}, {'name': 'Chrome', 'version': '131.0.0.0'}, {'name': 'Chrome', 'version': '131.0.0.0'}, {'name': 'Chrome', 'version': '131.0.0.0'}, {'name': 'Chrome', 'version': '131.0.0.0'}]
Operating System: [{'name': 'Windows', 'version': '10'}, {'name': 'Windows', 'version': '10'}, {'name': 'Windows', 'version': '10'}, {'name': 'Windows', 'version': '10'}, {'name': 'Windows', 'version': '10'}]
Device: ['Desktop', 'Desktop', 'Desktop', 'Desktop', 'Desktop']
Country: [None, None, None, None, None]
IP Address: ['127.0.0.1', '127.0.0.1', '127.0.0.1', '127.0.0.1', '127.0.0.1']