

Final Project Part 4: User Interface and Web Analytics

Introduction

This fourth and last part of the final project consisted in building a User Interface and applying Web Analytics on the searches. Said web application had to be able to accept search queries, displaying the results of the aforementioned queries as well as the usage statistics. As a submission we have this PDF file (*IRWA-2021-final-project-U162442-U161809-part-4*) and the following GitHub repository where you can find the code we have used to accomplish all the tasks required for the submission: https://github.com/vladimirtrukhaev/IRWA_Project. In said GitHub repository you can find a branch called "IRWA-2021-final-project-part-4" (https://github.com/vladimirtrukhaev/IRWA_Project/tree/IRWA-2021-final-project-part-4) where the code for this part of the project, this PDF file as well as the provided

User Interface

In order to create the UI we were given some templates that really help us structure such a complex implementation. We will explain our reasoning behind the way we completed and structured the code.

We will first explain how the environment is initialized, that is to create the corpus (info_dict), the index (index) and scores (tf and idf) before the search function is accessed. We found that the initialization took around 1-2 minutes to complete, and that it would not be optimal or sensible to have this run every time we run the search function. Therefore, and because we are working with a limited and static dataset, we are able to create those variables and store them in files, so that for the following executions it is ready to go!

First of all, our web_ap.py starts by loading the corpus using the function `load_documents_corpus()` on the utils file.

`load_documents_corpus()`

This function accesses the original file **once** and gets useful information of each tweet, at the end it returns a dictionary of tweets ids as keys and, **class** `Document` as values (Document has tweet, username, hashtags... etc). If running it for the first time, it will also convert this dictionary to the file `docs_info.txt`.

We also use this function to create another file (*my_dict_raw.txt*) which is a dictionary whose keys are ids and its values are only the tweet text. This dictionary will eventually (after text processing it) be used when creating the index.

In this way, the original (and bigger) dataset only needs to be accessed **once**: the first time we run the app and never again. This whole function is implemented such that, if it finds *docs_info.txt*, it accesses it directly, otherwise it means we are restarting the app, and therefore will create the previously mentioned files.

With the corpus loaded we pass it as a variable to the SearchEngine class.

```
searchEngine = SearchEngine(corpus) (INITIALIZATION)
```

In the initialization we decided to create the following variables:
self.info_dict, *self.index*, *self.tf* and *self.idf*.

We needed to pass the corpus as a parameter so that we could have it as *self.info_dict* variable. The rest of the self variables (*self.index*, *self.tf*, *self.idf*) are created by calling the function of the *algorithms.py* file: *create_index_tfidf(my_dict, num_documents)*- and the parameter *my_dict* is implemented by the function *build_data()*:

```
build_data()
```

This function (implemented in the same *search_engine* file) returns the text-processed dictionary of tweets that will be used to create the index. This function either gets the dictionary directly from *my_dict.txt* (if it finds it), or (if it is the first run) it creates *my_dict* by modifying the previously mentioned *my_dict_raw.txt* - this dictionary has the whole text of each tweet, we need to process it. We lower, clean, tokenize, remove stopwords, and stem the tweets by calling to the functions implemented on *algorithms.py* (those are the same function that we use in Part 1 of this practice).

```
create_index_tfidf(my_dict, num_documents)
```

This function works similar to the above ones: it will search for the index.txt, idf.txt and tf.txt files, if it cannot find them (which probably means it is the first execution) we will create them and return them. We decided to implement the score as tf-idf since it was the one that worked better in the previous practices.

Up to this point, everything created will be the base of our search engine: we have the index, the score, and the corpus as info_dict (where we can find all the useful information).

Now, whenever a user enters a query and clicks “search” the search function of the SearchEngine class will be called as *results = searchEngine.search(search_query)* - query being the user’s input. Now we will explain line by line what this function does, recall that info_dict, index, tf and idf are the SearchEngine variables:

search(self, search_query) code lines:

```
query = build_terms(search_query)
```

The first step is to text process the input query so that it matches any of our terms correctly. For this we use the function build_terms(query) (created in algorithms.py) that works similarly to build data() but with a string as the input and output.

```
docs = search_alg(query, self.index)
```

Then we use a function called search_alg() (implemented in algorithm.py) to look for those documents whose content match the query terms.

If works as it did on previous practices, at the end it returns a list of docs id that do match the query.

```
results = rank_documents(query, docs, self.index, self.idf, self.tf)
```

Now having a list with the matching tweets we will call the rank_documents function that will take that list (docs), the query, the index, idf and tf (this is why we kept them self. variables, so that we could call them now).

Now it is time to build the output that will be the list of `class TweetInfo` class that the function "SearchEngine.search" returns.

`class TweetInfo:`

This Class is a simpler version of the `class Document`, and has the following variables: `tweet_title`, `date`, `likes`, `retweets`, `doc_page`. This class allows us to show the important information of each tweet in the results page.

The *output* is built as a list of `TweetInfo` items created from the results ids and the `self.info_dict (corpus)`. At the end we return the list *output*.


The output then is used to render the `results.html` with the appropriate information. We can check that it does it successfully, this is an example of the four first results for the query "how many covid cases":

As we can see, we decided to only display in the results page the first 100 characters of each tweet, the likes, retweets and the date. This is because displaying a "description" would mean to display the whole tweet and we saved that information for the `doc_details` part, which is displayed when someone clicks on the tweet. As shown below, the `doc_details` page of fourth result looks like that:

upf. Universitat
Pompeu Fabra
Barcelona

IRWA Search Engine

Found 187 results...

 WHO clinical case definition for post #COVID19 condition, also called 'long COVID'

<https://t.co/Wo...>

Likes: 723 Retweets: 373

07 Oct 2021 12:28:10

"We know that COVID-19 is just one challenge you face, as the recent cases of Marburg in Guinea and ...

Likes: 38 Retweets: 12

24 Aug 2021 09:27:28

@DrTedros "Although #VaccinEquity will help to end the pandemic, its effects will continue to be fel...

Likes: 146 Retweets: 37

07 Oct 2021 14:03:18

RT @DrTedros: Almost 4 million #COVID19 cases were reported to @WHO last week - many of these were d...

Likes: 0 Retweets: 322

30 Jul 2021 20:35:57

Tweet Details - IRWA Search Engine

Date & Time: 30 Jul 2021 20:35:57

User: World Health Organization (WHO)

Tweet:

RT @DrTedros: Almost 4 million #COVID19 cases were reported to @WHO last week - many of these were driven by the highly-transmissible Delta...

Hashtags : COVID19

Number of likes: 0

Number of retweets: 322

URL: <https://twitter.com/WHO/status/1421207960395685889>

[Go Back](#)
[Go Back 2 pages](#)
[Go Back 3 pages](#)
[Go Back 4 pages](#)
[Stats](#)

Here we will have the information that is shown in the `doc_details` page. We pass all the information with the `corpus` variable by accessing it with the click information. We had to "clean" some variables such as date and hashtags so that it will show nice information to the end user.

Web Analytics

For the web analytics part of this project we were asked to provide a mechanism that could be used to track people that use our website and the search engine. To do so we created different functions and classes in the various files we had. We had to modify the `analytics_data.py`, the `dashboard.html`, `web_app.py` amongst others.

```
@app.route('/doc_details', methods=['GET'])
def doc_details():
    # getting request parameters:
    # user = request.args.get('user')
    clicked_doc_id = request.args["id"]

    if clicked_doc_id in analytics_data.fact_clicks.keys():
        analytics_data.fact_clicks[clicked_doc_id].counter += 1
    else:
        analytics_data.fact_clicks[clicked_doc_id] = Click(clicked_doc_id, corpus[str(clicked_doc_id)].tweet, 1)

    print("click in id={} - fact_clicks len: {}".format(clicked_doc_id, len(analytics_data.fact_clicks)))
```

In this part of the code that is located inside the `web_app.py` file we basically count the amount of clicks a document gets in order to later on show it on our dashboard.

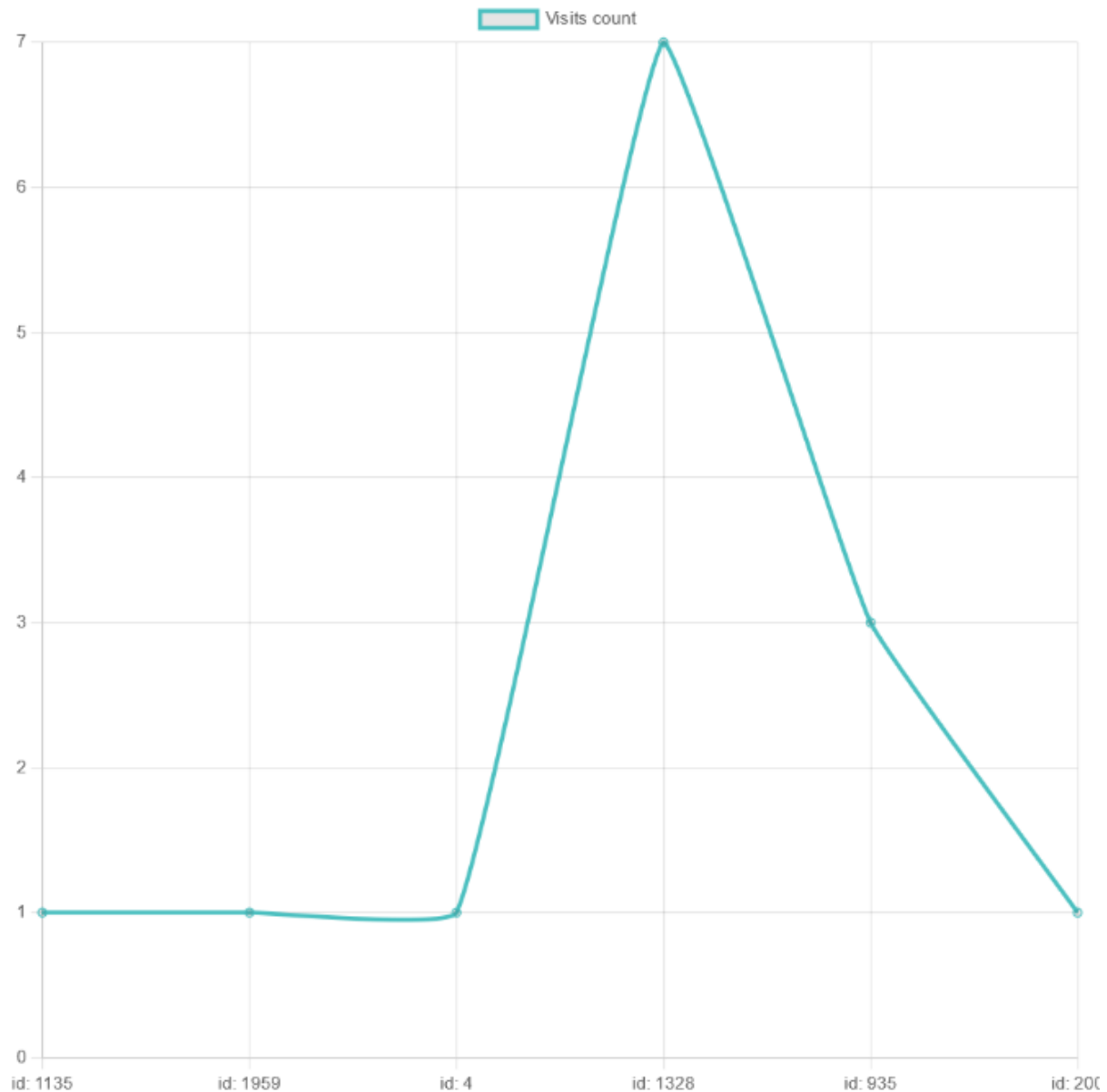
To pass this information to the dashboard we first have to go through:

```
@app.route('/dashboard', methods=['GET'])
def dashboard():
    visited_docs = list(analytics_data.fact_clicks.values())
    for doc in visited_docs: print(doc)
    # simulate sort by ranking
    #visited_docs.sort(key=lambda doc: doc.counter, reverse=True)

    return render_template('dashboard.html', visited_docs=visited_docs)
```

With that return, we save the `visited_docs` variable that contains the amount of visits a document (in this case Tweet) has as well as the ID of said Tweet. We then use the `visited_docs` value in our `dashboard.html` file and show the ranking of the documents the user has visited in the following way:

Ranking of Visited Documents



Unfortunately after having tried numerous times we could not get another dashboard to work.

However, we have also implemented different functions that are used in order to get the user's browser, IP address, the date and time and so on. To do so we used different libraries such as the datetime, and the request function from the flask library. Unfortunately it was not possible to get the user's location data as we were using the localhost IP address which is 127.0.0.1 and has no location data assigned.

```
browser_list = list()
browser = request.headers.get('User-Agent')
browser_list.append(browser)

now = datetime.now()
date_time = now.strftime("%d/%m/%Y %H:%M:%S")

ip_list = list()
ip_add = request.remote_addr
ip_list.append(ip_add)
```

Conclusion

Bringing up rear this whole report and project we would like to say that this part was definitely the most tedious one as we had to code in different languages. However it was one of the most informative project parts we have done so far.

At some point we thought that we would not be able to finish the project before the deadline however, we kept striving and coding until we had everything coded and the report finished. This was due to the fact that at first we did not know how to start but after some tries and the help of our teacher Gustavo we got onto the right path. Honestly, the project turned out better than we expected and everything works completely fine and as it should be.

In our opinion this part, even though it was a tedious one, helped us a lot to understand different concepts when talking about Search Engines and Web Analytics. It was also a good practice to improve our coding skills as we were required to work with different programming languages.