Sergi Solís Valdivieso u172769 - Hong-ming Xiang Vico u172801

IRWA Project Part IV: User Interface and Web Analytics

Statement

User Interface

It is now the time to give your search engine a user interface (UI) and apply some web analytics on it. You are asked to provide a Web application for entering the search query, displaying the search results and usage statistics.

Use the simple Python web framework **Flask** that runs its own development web server (a project boilerplate will be created in class as a hands-on part in the Web Analytics seminar).

Project content:

1. **Search page**: Create a main web page with a central search box for users to enter a query and a button to execute the search.
2. **Search action**: When the button is clicked in the UI, the search text must be passed to this engine's search function.
3. **Search function in the engine**: Provide in your search engine a general "search" function that receives a string as parameter. Add any other parameter you consider helpful for the implementation.
4. **The search algorithms**: The search function in turn must call the previously defined algorithms:
    a. Now that you have a real use case, optimize your algorithms for the goal of retrieving the best results, faster, cleaner, and that better suit the user's information needs.
    b. Structure your code in a way that can be used in the web application.
    c. The documents corpus will be the already provided Twitter entries.
5. **The results page**: create a web page that displays the list of documents found for the query and in the calculated order/ranking.
    a. Each result record must represent a document from the corpus so it must have at least the following properties:
        i. Title (in tweets we can use a substring with some initial characters of the full tweet text)
        ii. Summary description (probably the whole tweet text)
        iii. Creation date/time
        iv. URL (used to link to document details page)
        v. Other items you consider relevant for the results page.
6. **The document details page**: a page to display the whole document's information.
    a. Display other relevant properties present in the corpus for each document.

Web Analytics

You are required to provide a robust mechanism for effective tracking and analyzing how people use your website and the search engine's use. As this is for educational purposes you can store data in memory (the use of a database for long time persistence is optional and if used it must be very well documented in order for
teachers to reproduce in their environments for evaluation). Find all the documents that contain all the words in the query and sort them by their relevance with regard to the query.

1. **Data collection**: You can for example collect and store the following actions:
    a. For HTTP
        i. Requests data
        ii. Clicks
        iii. HTTP Sessions data
    b. For queries: number of terms, order, etc.
    c. For results (documents)
        i. clicks on documents.
        ii. to what query where related.
        iii. ranking of clicked documents.
        iv. dwell time: the time between clicks on a result document and coming back to the results page.
    d. For user context (visitor):
        i. browser, OS/computer/mobile, time of the day, date
        ii. IP address, country, city (optional bonus point)
    e. For sessions (optional bonus point)
        i. time based sessions: queries of a user in the same sit down (physical session).
        ii. missions: sequence of queries with the same goal (logical session), there can be multiple missions within a physical session.
        iii. research missions: span across multiple sit downs, days, weeks, etc.
    f. All other information that you find interesting (justify in your report)
2. **Data storage**: Design a data model to store the information that you collect about the website usage that you collect from above point. Consider a star schema for example with the following tables:
    a. Session
    b. Click
    c. Request
3. **The analytics dashboard**: Create a web page that displays the usage statistics.
    a. Define key indicators, metrics, and reports.
    b. Show how people use your website and search engine.
    c. Provide graphs with explanations of what is displayed.

GitHub Repository

All the code and resources for the project will be submitted to the following repository:
https://github.com/homexiang3/IRWA-2022-u172769-u172801
The repository TAG is:

IRWA-2022-u172769-u172801-part-4

Code development

Previous work considerations

This is the fourth part of the IRWA 2022 project, although we don't follow the same Google Collab schema on this part, previous work is considered to perform several actions. For further details visit:

Part 1 - Text Processing:
https://github.com/homexiang3/IRWA-2022-u172769-u172801/blob/main/Project/P1/IRWA-2022-u172769-u172801-part-1.pdf
Part 2 - Indexing and Evaluation:
https://github.com/homexiang3/IRWA-2022-u172769-u172801/blob/main/Project/P2/IRWA-2022-u172769-u172801-part-2.pdf
Part 3 - Ranking:
https://github.com/homexiang3/IRWA-2022-u172769-u172801/blob/main/Project/P3/IRWA-2022-u172769-u172801-part-3.pdf

User Interface

In this section, we will explain how we structure our code, the different functions to retrieve the relevant tweets in our search engine, and some frontend considerations related to the results and details page.

Code structure

We decided to work with the already existing code structure, the files modified for this first part are *"./myapp/search/algorithms.py"* where we will store the search logic such as the index creation and the ranking functions, *"./myapp/search/search_engine.py"* where we will define our search engine class and *"./web_app.py"* that manages the whole application.

In addition, we added some png images inside the *"./static"* folder for frontend purposes and a folder "./myapp/dump_files" that we will explain in detail below.

Algorithms

We will not enter in detail in this section since functions are already seen in previous project parts, the different functions are:

- Preprocessing functions that given text builds a preprocessed list of terms
- Create TF-IDF index
- Rank documents
- Search within query

Search Engine

First of all, since **TF-IDF** will be our selected ranking method for this project, we will need to create and define an index given our tweets corpus. We defined the needed dictionaries as our SearchEngine class parameters.

```
class SearchEngine:
    """educational search engine"""
    index = []
    tf = []
    df = []
    idf = []
    tweet_index = []
```

Next one, we define our create_index function that creates our different dictionaries. Here we face a problem while we were coding, each time that we update the project we need to wait a huge amount of time for the index creation, so we decided to use the pickle library to load and save the different dictionaries inside pickle files, placed in the directory *"./myapp/dump_files"*. So as the time that you test the code server will run smoothly. However, we decided to create a boolean flag that if you set True, the function will re-create the index again, calling to *algorithms.py* function *"create_tfidf_index"*, which is the same that the one provided in the project part 2 but adapted to take the information from this corpus. Create index function will be called from *web_app.py* on our initialization section

```
def create_index(self,corpus):

        # Check if file dumps exists to avoid index creation each time we run the server
        DUMP_DIR = Path('./myapp/dump_files')
        DUMP_DIR.mkdir(parents=True, exist_ok=True)

        index_filepath =  DUMP_DIR / 'index.pk'
        tf_filepath =  DUMP_DIR / 'tf.pk'
        df_filepath =  DUMP_DIR / 'df.pk'
        idf_filepath =  DUMP_DIR / 'idf.pk'
        twindex_filepath = DUMP_DIR / 'tweet_index.pk'

        always_create = False # Set it False if you want to load from dump if the file
exists or true if you want to create index always

        # Check if files exist to load data from cache or create a new tf-idf index and
the new files
        if always_create == False and index_filepath.exists() and tf_filepath.exists()
and df_filepath.exists() and idf_filepath.exists() and twindex_filepath.exists():
            print("Load data from cache!")
            self.index = pickle.load(index_filepath.open('rb'))
            self.tf = pickle.load(tf_filepath.open('rb'))
            self.df = pickle.load(df_filepath.open('rb'))
            self.idf = pickle.load(idf_filepath.open('rb'))
            self.tweet_index = pickle.load(twindex_filepath.open('rb'))
            print("Data loaded successfully!")
        else:
            print("Creating tf-idf index...")
            self.index, self.tf, self.df, self.idf, self.tweet_index =
create_tfidf_index(corpus)
            print("Finishing index creation...")

            print("Creating file dumps...")
            with open(index_filepath, "wb") as f:
              pickle.dump(self.index, f)
```

```
            with open(tf_filepath, "wb") as f:
              pickle.dump(self.tf, f)

            with open(df_filepath, "wb") as f:
              pickle.dump(self.df, f)

            with open(idf_filepath, "wb") as f:
              pickle.dump(self.idf, f)

            with open(twindex_filepath, "wb") as f:
              pickle.dump(self.tweet_index, f)

            print("Finishing file dumps...")
```

.

Finally, we have the search function and an auxiliary function named find_tweet. Find tweet returns a Document class object corresponding to the tweet with the given id. Here we use our tweets_index, which matches the tweet_id with the position inside the corpus, and then we retrieve using a list of corpus values.

```
def find_tweet(self, ll, id):
        corpus_index = self.tweet_index[id]
        item: Document = ll[corpus_index]
        return item
```

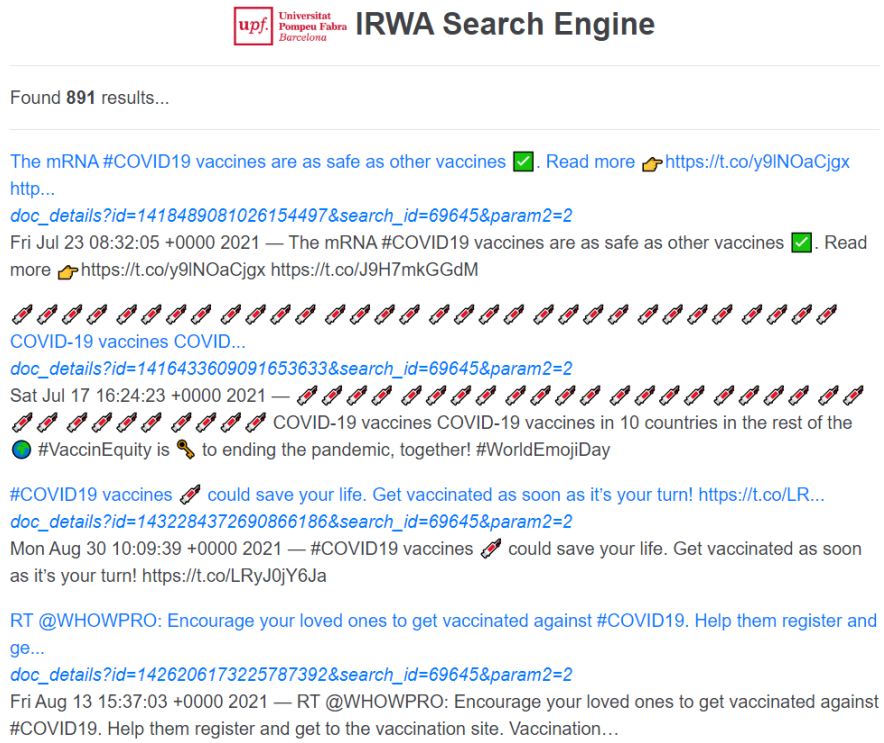
Search function simply calls to the search algorithm placed in *algorithms.py* that given the search query, index, tf, and idf returns the ranked tweets and scores. Then, we process each tweet into a ResultItem object as shown in the build demo example, also as ranked scores are ordered in the same order as the tweets, we add manually each ranking score using a counter. This function will be called in *web_app.py* in the *search_form_post* function.

```
def search(self, search_query, search_id, corpus):
        print("Search query:", search_query)

        results = []
        ##### your code here #####
        #results = build_demo_results(corpus, search_id)  # replace with call to search
algorithm

        ranked_docs , ranked_ranks = search_in_corpus(search_query, self.index,
self.idf, self.tf)
        ##### Given the list of tweets id process to take entire tweets #####
        ll = list(corpus.values())
        rank_counter = 0
        for id in ranked_docs:
            item = self.find_tweet(ll,id)
            results.append(ResultItem(item.id, item.title, item.description,
item.doc_date,
                                "doc_details?id={}&search_id={}&param2=2".format(item.id,
search_id), ranked_ranks[rank_counter]))
            rank_counter = rank_counter + 1
```

```
        return results
```

Results page

Once we have all the algorithmic and search engine logic, we could perform a search in our user interface. With the Flask application running we can navigate into the URL http://127.0.0.1:8088/ or http://localhost:8088/ to see the home page with the search bar. In this example, we will use the query *"covid vaccine"*. Then our server will perform the TF-IDF ranking and retrieve the results to the results page. As explained above, we decided to maintain the same format used in the demo for the results visualization (title, link, date, and description).



Details page

Maybe we want to see more information about the tweet such as the number of likes, retweets, and hashtags, so we need to create a details page to visualize all this data. In the original template, we have a simple HTML with 6 links, so we redefined the HTML file *"./templates/doc_details.html"* similarly to the results page, extending the base.html file and in this case, adding all the Document class object information.

```
{% extends "base.html" %}
{% block page_title %}{{ page_title }}{% endblock %}
{% block content %}
    {% if tweet is defined %}
        <div class="pb-3">
                    <h5>Tweet {{tweet.id}} Details</h5>
```

```
            <div class="doc-title">
                <a href="{{ tweet.url }}">{{ tweet.title }}...
                </a>
            </div>
            <div class="cited">
                <img src="{{ url_for('static', filename='link.png') }}"  width="16px"
alt="link icon"/><cite> <a href="{{ tweet.url }}">{{ tweet.url }}</a></cite>
            </div>
                      <div>
                              <img src="{{ url_for('static', filename='calendar.png') }}"
width="26px" alt="calendar icon"/> {{ tweet.doc_date }}
                      </div>
            <div class="doc-desc">
               {{ tweet.description }}
            </div class="d-inline pl-3">
                              <img src="{{ url_for('static', filename='heart.png') }}"
width="16px" alt="heart icon"/>  {{ tweet.likes}}
                      <div class="d-inline pl-3">
                              <img src="{{ url_for('static', filename='retweet.png') }}"
width="20px" alt="retweet icon"/>  {{ tweet.retweets}}
                      </div>
                      <div class="d-inline pl-3">
                              <img src="{{ url_for('static', filename='hashtag.png') }}"
width="16px" alt="hashtag icon"/>  {{ tweet.hashtags}}
                      </div>

        </div>
              <div class="d-inline"><a href="#" onclick="history.go(-1)">Go
Back</a></div>
              <div class="d-inline pl-3"><a href="#" onclick="history.go(-2)">Go Back 2
pages</a></div>
              <div class="d-inline pl-3"><a href="#" onclick="history.go(-3)">Go Back 3
pages</a></div>
              <div class="d-inline pl-3"><a href="#" onclick="history.go(-4)">Go Back 4
pages</a></div>
              <div><a href="/stats" >Stats</a></div>
              <div><a href="/dashboard" >Dashboard</a></div>
       {% endif %}
{% endblock %}
```

In the *web_app.py doc_details* function, we just modify a few lines to create the Document class object containing all the information from the tweet, using the *find_tweet* function from the searchEngine class and taking advantage of the URL format (we extract the tweet id from the first parameter and create the variable clicked_doc_id with the value), then we just create again the URL and we render the template with the tweet.

```
ll = list(corpus.values())
tweet = search_engine.find_tweet(ll,int(clicked_doc_id))
tweet.url = "doc_details?id={}&search_id={}&param2=2".format(tweet.id, p1)
return render_template('doc_details.html',tweet=tweet, page_title="doc_details")
```

For the information part, we display the id of the tweet, the title, link, date, description, likes, retweets, and hashtags, then we decided to add some png images as icons to enhance the result and we maintain the original template links to navigate through other pages. **Note:** *"Go Back"* link will need an additional refresh to resend the form

Web Analytics

For this part we decided to create a total of 6 fact tables to store the metrics. The different fact tables are created in *"./myapp/analytics/analytics_data.py"*.

fact_clicks is the default fact table that stores the most visited documents, **fact_queries** does the same but stores the number of times that we search for a query, **fact_click_queries** store the number of clicks that the user does within a specific query, **fact_browser,** and **fact_os** takes advantage of the already provided *httpagentparse*r library to store the user agent credentials, as we want to be specific we store the platform + version as the dictionary key eg. Windows 10, Chrome 108.0.0.0. Finally, we have the **fact_timeactivity** that keeps track of our web traffic, storing the number of events per time (in hours - minutes format)

```
# statistics table 1
    # fact_clicks is a dictionary with the click counters: key = doc id | value = click counter
    fact_clicks = dict([])

    # statistics table 2
    # fact_queries is a dictionary with the queries counters: key = query | value = query counter
    fact_queries = dict([])

    # statistics table 3
     # fact_click_queries is a dictionary with the clicks counters within a query: key = query | value = query counter
    fact_click_queries = dict([])

    # statistics table 4
```

```
    # fact_browser is a dictionary with the browser counters: key = browser| value =
browser counter
    fact_browser = dict([])

    # statistics table 5
    # fact_os is a dictionary with the os counters: key = os | value = os counter
    fact_os = dict([])

    # statistics table 6
    # fact_timeactivity is a dictionary with the number of events counters: key = time |
value = event counter
    fact_timeactivity = dict([])
```

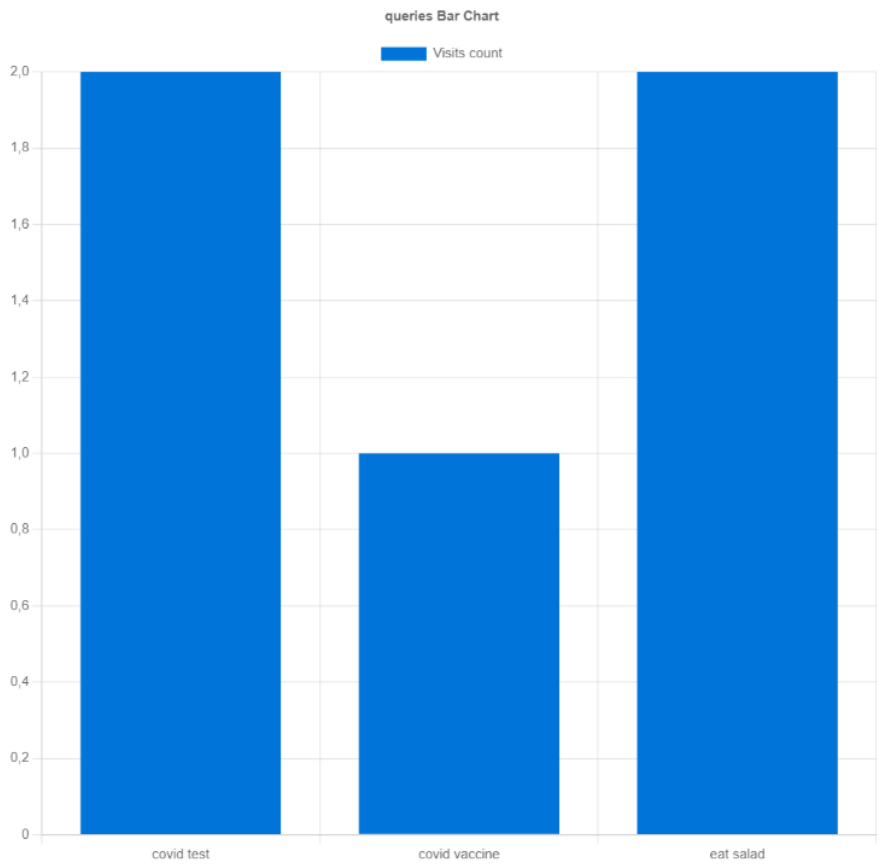
We collect the different data inside our web_app.py file, for example, fact_os and fact_browser is updated when someone enters in our home page, fact_queries when someone performs the search, fact_clicks and fact_click_queries when someone enters into the details page and fact_timeactivity is executed in each endpoint as a new event.

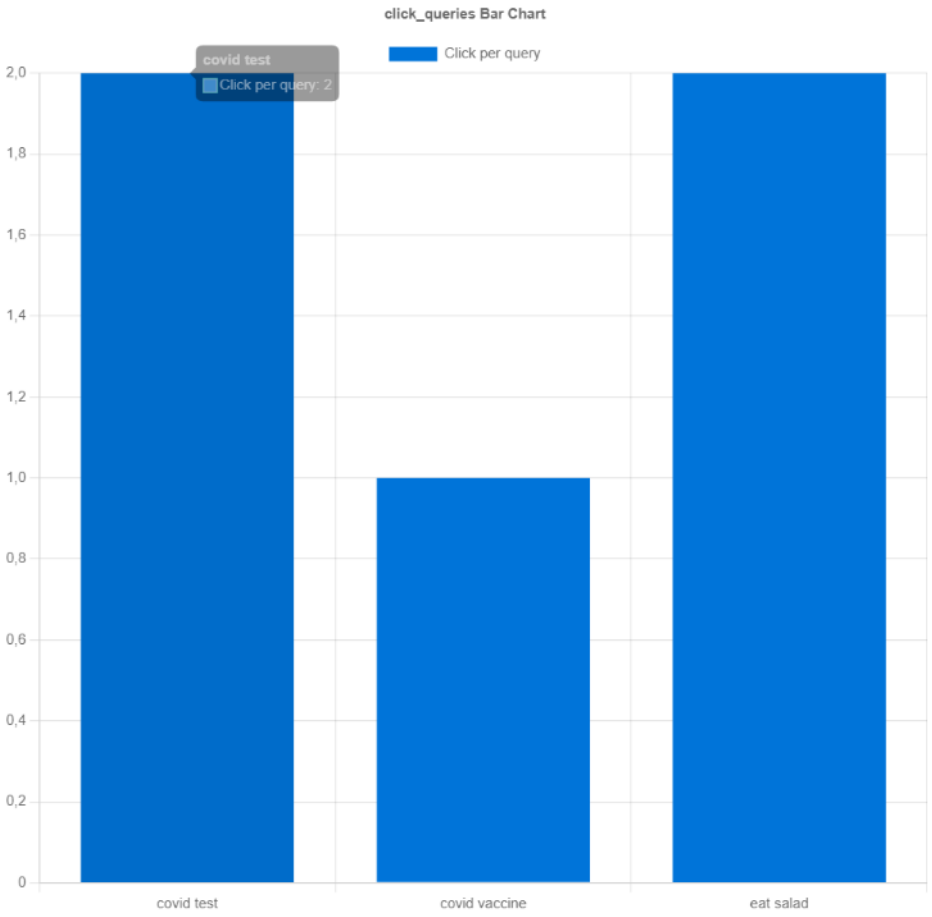
We also add two auxiliary functions inside our AnalyticsData class to increment the dictionary counter given the fact table and the key, and another to do the same but precalculating the current time.

```
def increment_fact(self,var,fact):
        if var in fact.keys():
            fact[var] += 1
        else:
            fact[var] = 1
def add_timeactivity(self):
        now = datetime.now()
        current_time = now.strftime("%H:%M") #we decided to use only hours - minutes for
simplicity
        self.increment_fact(current_time,self.fact_timeactivity)
```

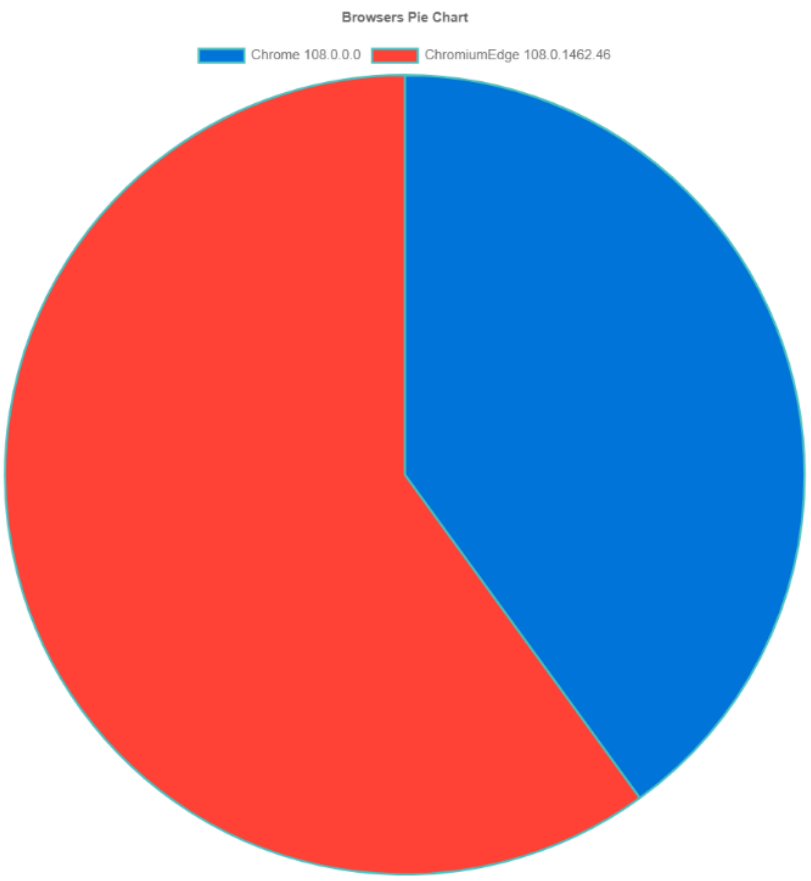
Then, in our dashboard function call inside *web_app.py* file, we sort and send each of the fact tables to be used in the Dashboard page.

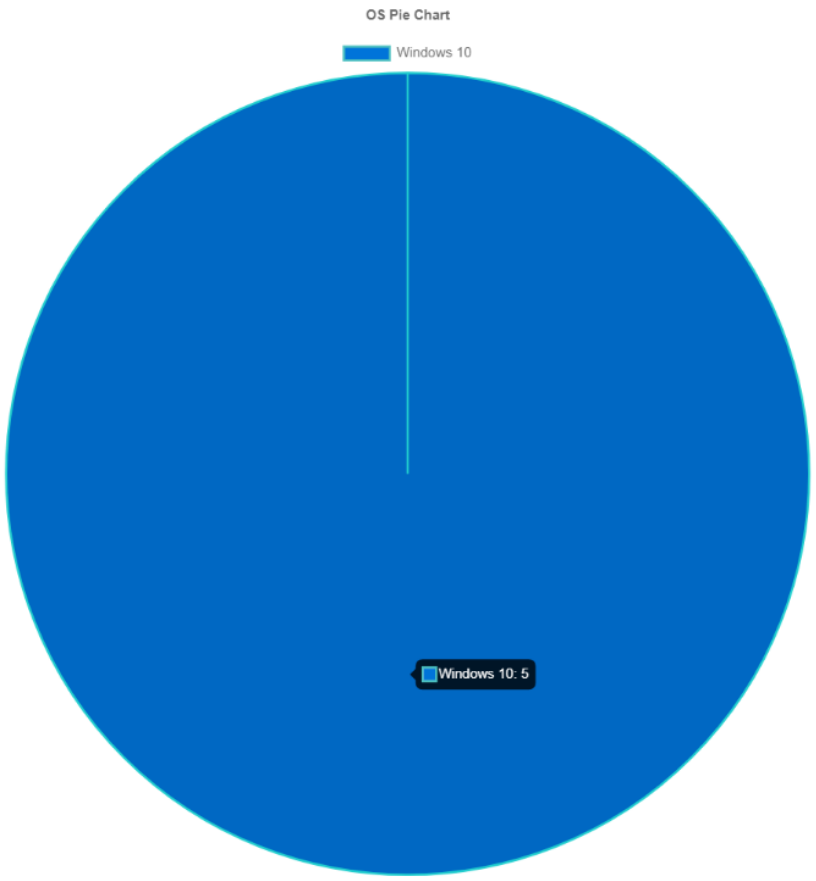
In the Dashboard webpage, we decided to keep the template's plot with the visited docs and then we add fact_queries and fact_click_queries as bar plots, fact_browsers and fact_os as pie charts, and fact_timeactivity as a line chart. We provide the different added charts images below.

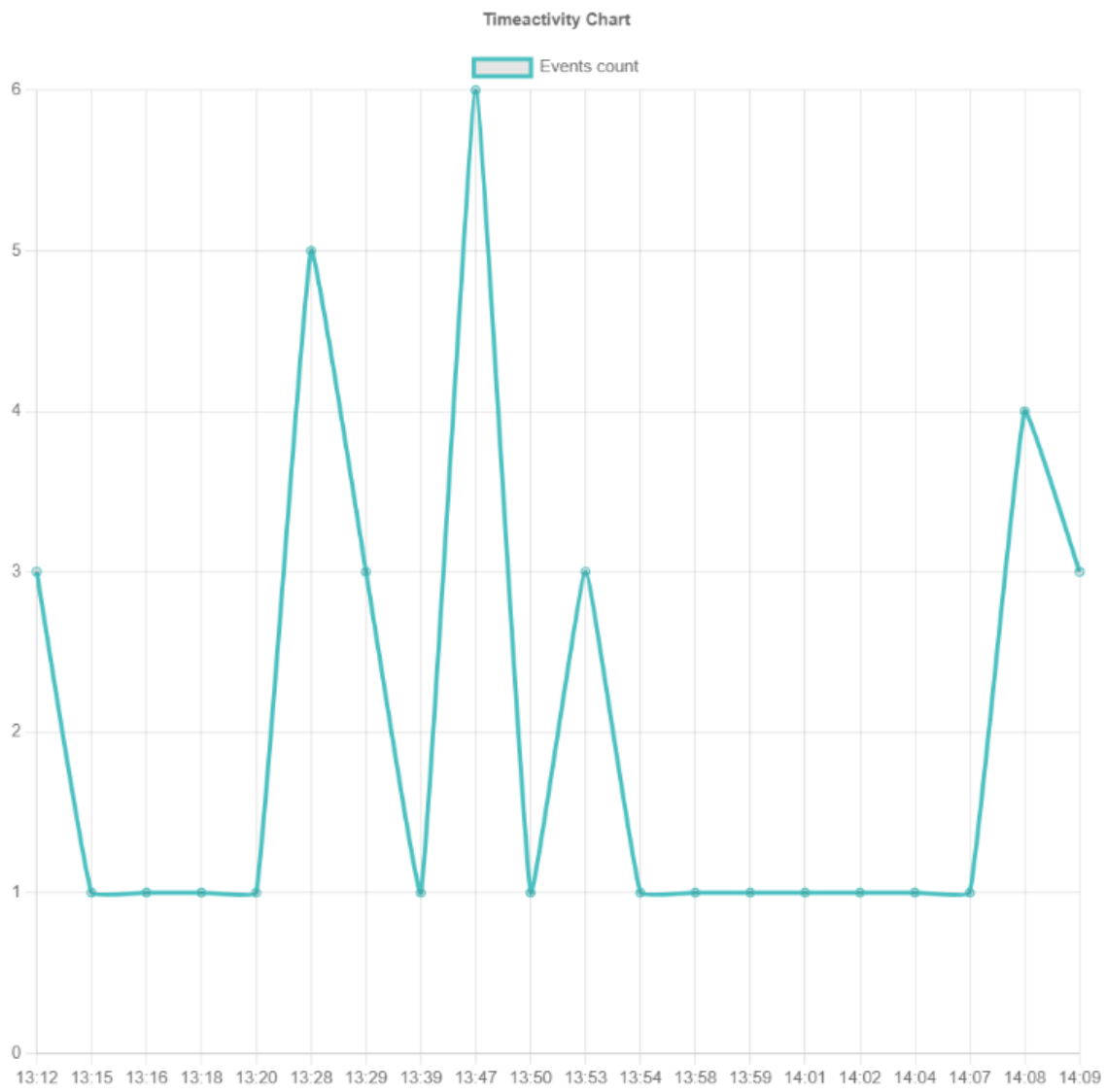

queries Bar Chart
Visits count
2,0
1,8
1,6
1,4
1,2
1,0
0,8
0,6
0,4
0,2
0
covid test
covid vaccine
eat salad


click_queries Bar Chart
Click per query
covid test
Click per query: 2
2,0
1,8
1,6
1,4
1,2
1,0
0,8
0,6
0,4
0,2
0
covid test
covid vaccine
eat salad

Sergi Solís Valdivieso u172769 - Hong-ming Xiang Vico u172801






Timeactivity Chart
Events count
6
5
4
3
2
1
0
13:12
13:15
13:16
13:18
13:20
13:28
13:29
13:39
13:47
13:50
13:53
13:54
13:58
13:59
14:01
14:02
14:04
14:07
14:08
14:09