1. INTRODUCTION

I was asked to implement a twitter search engine, which supported both the normal query terms search and emojis search. In addition, another feature that the system must present is the query expansion: the user must be able to define a series of synonyms, which will then be used to expand the query.

2. DESIGN

SCRAPING

The first need was to get a large collection of tweets. I thought 10'000 was a good size for a large collection of tweets, which was also confirmed to me by the assistant Ivan Sekulic. As advised in class, I used Scrapy[1] to scrape the 10'000 tweets. Scrapy is a Python framework that is used to build web scraping spiders.

To build and run a spider that works correctly and is not blocked by the target site, it is necessary that the spider respects the instructions contained in the *robots.txt* file of the site in question. So, I read the *robots.txt* file of Twitter. Unfortunately, the company only allow a few pages to be scraped:

1. Allow: /*?lang=
2. Allow: /hashtag/*?src=
3. Allow: /search?q=%23
4. Allow: /i/api/

My initial idea was to build a spider capable of scraping 10'000 random tweets, but the limitations of the *robots.txt* file didn't allow me to. I could just build a spider that would search for all the tweets containing a certain hashtag (2), containing a certain term (3), or make calls to their API. Furthermore, I had the possibility to search for any of the previous contents, specifying the language (1).

So, I decided to use the second type of search: look for tweets containing a particular hashtag. This way, I would have obtained my file containing 10'000 tweets. So, if for example I want to search all the tweets containing the hashtag *#hello*, the final part of the URL will be:

- /hashtag/hello

I said the *final* part, because I also had to decide the first part of the URL. When I started building the spider, I immediately noticed problems sending requests and downloading information from the website *www.twitter.com*. For this, I did some research on the internet and found numerous users who recommended sending requests to the mobile version of twitter *www.mobile.twitter.com*. So I did, and I haven't had any more problems of any kind. Hence, the complete URL from which my spider started is:

- https://www.mobile.twitter.com/hashtag/hello

However, it didn't seem like a great idea, since looking for tweets containing the same hashtag, I would have obtained 10'000 tweets containing also the same hashtag, so almost about the same topics. In addition, building the scraper that I then used, I noticed that Twitter does not allow to scraper more than 20 tweets per page, so per hashtag, with the same request.

[1] https://scrapy.org/

Here, I got the idea on which my project is based. To work around these limitations, I decided to search by hashtag by providing numerous terms instead of just one. This way I could send multiple scrape requests, each for a different hashtag, and each retrieving 20 tweets, ending up with a list of 10'000 tweets related to different topics. Therefore, I built a single spider which in turn sent requests for different hashtags.

For each hashtag I would have retrieved 20 tweets, but if some kind of error would have occurred, it may be that for each scrape request I would have obtained less than 20 tweets. What I did was calculate how many hashtag-search-terms I would have needed to get 10'000 tweets, considering the possibility of not getting exactly 20 tweets per search. This calculation is very simple:

$$\frac{1\ term}{20\ tweets} = \frac{n\ term}{10'000\ tweets} => n = 500$$

To be safe, I then decided to add 50 terms, ending up with 550 hashtag-search-terms. But how to get 550 random unique terms in an easy-to-read format? This is where the *www.randomlists.com*[2] comes into play. In fact, this website allows you to generate a maximum number of 2'467 random terms and to easily copy the result, in order to obtain a file with a term for each line. This is exactly what I did: I told the website to generate 550 random terms and I copied them into the *words.txt* file, that you can find attached.

TWEET STRUCTURE

FIGURE 1 – EXAMPLE OF TWEET

A tweet is displayed in different ways, depending on its content: it can contain a picture, a video, a text, hashtags, emojis, links, or just some of these.

Here on the left you can find an example of a tweet.[3]

There is the name, username and profile picture of whoever wrote it, the date it was posted and the statistics of the tweet: number of comments, shares and likes.
Furthermore, the tweet does not contain a simple text, but a text, an emoji, many hashtags, and a link to an article.
Therefore, I decided to extract only some information from each tweet, what seemed to me to be the most important and useful to represent a tweet. Some of them may be absent in some other tweets.

For instance, a tweet like the one shown in Figure 1 contains many hashtags. A different tweet may not even contain hashtags, but contain a picture.

The following list shows how I schematized and analyzed each tweet:

url The URL of the tweet

date The creation date of the tweet

full_name The name of the user who wrote the tweet

[2] https://www.randomlists.com/random-words
[3] You can find this tweet at the link: https://twitter.com/chidambara09/status/1317749123102109697

username The username of the user who wrote the tweet

profile_pic The profile picture of the user who wrote the tweet

content The text contained in the tweet

metadata The URL of the photo contained in the tweet

hashtag The list of hashtags present in the tweet done with #

tag The list of tags present in the tweet done with @

likes The number of likes the tweet has

retweets The number of retweets the tweet has

repost The number of reposts the tweet has

emoji The list of emojis present in the tweet

I decided to save my entire collection of tweets in a json file, as I have worked with such files before and know how they behave. So, from the folder where the spider was located, I executed the following command in the command line to run the spider:

```
> scrapy crawl twitterscraper -O tweets.json
```

As the documentation[4] says, this command tells scrapy to start crawling with the spider named *twitterscraper*, and save the result in a file named *tweets.json*. The *-O* flag indicates that if a file with the same name already exists, its contents will be overwritten.

Once I got the json file containing the collection of 10'000 tweets, I started the Solr server and indexed the tweets.[5] I used Solr to index the data as recommended by the professor.

TECHNOLOGIES

I had many ideas for the project, but how to connect all the pieces together? Last year, for the Software Atelier 3 course, we built *pushapp*, a single-page web app dedicated to the world of fitness. I found the single page factor interesting, and I therefore decided to propose it again in this project. But I had to decide which system to use to create the user interface. Since in the internship offered this year in the Field Project Atelier course, the company Zucchetti SA Switzerland asked me to build a web app using Vue.js, I decided to use the same system here, so as to increase my familiarity with it. Vue.js is a javascript framework for creating user interfaces and single-page applications, exactly what I need![6]

MAKE REQUESTS

Usually, to retrieve data you need to make calls to a backend part, which takes care of managing the interactions with the server and returning the data to the frontend part. Since I used Solr to index tweets, we can make calls directly to the virtual server on which Solr is running. The address of this server depends on our initial configuration of the Solr collection. In my case it is localhost: 8983.

[4] https://docs.scrapy.org/en/latest/intro/tutorial.html
[5] https://lucene.apache.org/solr/guide/8_6/solr-tutorial.html#solr-tutorial
[6] https://vuejs.org/

By accessing the Solr server from a browser and going to the part dedicated to queries, we can search our indexed tweets. For example, if we want to search by username, we need to specify the search field, *username* in this case, and the *value* we are looking for, for example *lucybishop2012*. By clicking on *Execute query*, our search results will be shown, complete with search statistics, such as speed and number of results for this query.

FIGURE 2 – TYPICAL SOLR RESPONSE TO A QUERY

In the upper part a URL is displayed, which if clicked takes us to a page containing the same result obtained before. By making a GET call from a frontend part to the same URL, we can achieve this in our web app. The URL is the following:

- http://localhost:8983/solr/tweets/select?q=username%3Alucybishop2012

By analyzing it, we can identify a structure: the part in orange contains the localhost on which the Solr server is running plus the name of our collection. The part in blue is the dynamic one, which by modifying it allows us to search for all kinds of tweets.

Here, I had some problems, as the CORS policies did not allow me to make requests from an external domain as the localhost on which *tweetsearch* was running. So, after a lot of research and attempts, I added the piece of code between lines 25 and 45 in the Solr configurations *web.xml* file. This file is located at path *solr-8.6.2/server/solr-webapp/webapp/WEB-INF*. I leave my *web.xml* file as an attachment.

In my application, I perform two different types of queries: filtered and not. For the non-filtered I modified the Solr schema file by adding a copy filed, so that I could perform a query search over the content of the tweet, the username and the full name of the author at the same time. The way I built the URL query is the following:

- http://localhost:8983/solr/tweets/select?q={query_inserted}&rows=50

With filtered queries, we can filter tweets by specifying values for hashtags, usernames and full names. The way the query is built is the following:

- http://localhost:8983/solr/tweets/select?q={filter_selected}%3A{query_inserted}&rows=50

In both cases, I retrieved a maximum number of 50 tweets by adding the *rows=50* parameter to the query URL. I decided 50 because according to my experience as a web user: nobody ever browses too many results. Rather, he decides to modify the entered query to find more relevant documents.

3. IMPLEMENTATION

COMPONENTS

When I started thinking about how to structure the project, I immediately thought that I might need several pages. For this I decided to insert a **menu**, through which you could navigate between them. This way I could expand the project as I pleased, without disturbing the pages already created.

Since I chose to have a simple, intuitive and easy to use design, the search page should not contain too many elements, capable of disturbing the user's attention. Therefore, I positioned the menu so that it was only shown when the *Menu* button was clicked (a kind of pop-up).

Also, I immediately recognized the idea of designing a custom **logo** as nice, you can see it in Figure 3.

FIGURE 3 – LOGO OF TWEETSEARCH

The *tweetsearch* interface is divided into 3 pages: Search, Thesaurus, and Credits. Each of these pages is presented in detail in the following sections.

i) SEARCH

This is also the main project page, the one I started designing first. I started thinking about what elements I would need to build a search engine dedicated to tweets, with also the ability to search by emoji and the query expansion feature.

Sure, I would have needed a **search bar** and a **search button**, and since the system had to support emoji search and the computer doesn't have an **emoji keyboard**, I decided to add a virtual one. I wanted a **filter**, through which it would be possible for a user to filter the field he was looking for during the search. Furthermore, to support the query expansion feature with synonyms, I wanted to add a section to load your own **synonym thesaurus** (for more details on this feature take a look at Thesaurus section). Finally, I had to choose how and where to represent the **tweets**.

These components had to be central, so I decided to place them in the center of the page, so as to be of greater impact. **Error! Reference source not found.** shows the sketch of the project interface made before starting so to have a reference, and the interface of the project once completed.

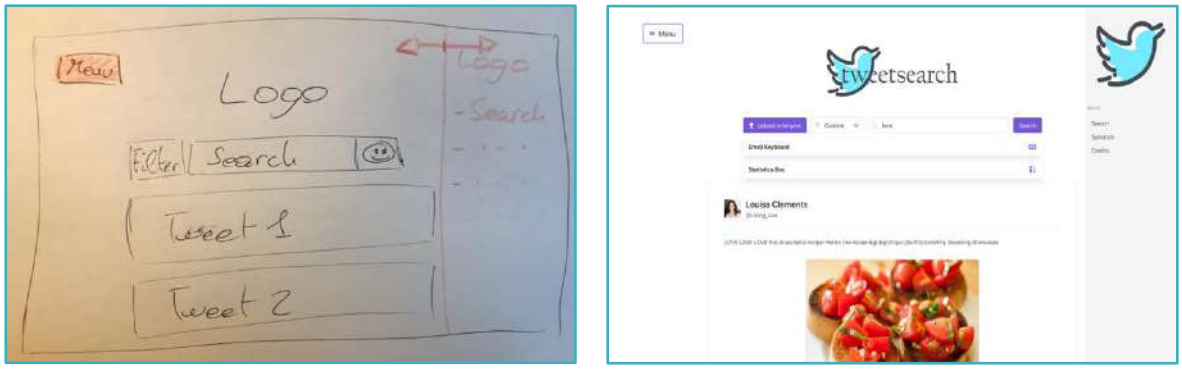

FIGURE 4 – INTERFACE: FROM SKETCH TO FINAL RESULT

As you can see, I changed a bit the layout: I found better to display the emoji keyboard below the search bar. Most important, I decided to add a more technical field: the **Statistics Box**. It appears only after entering a query, and contains information on the tweets obtained: the query entered, how many relevant tweets for this query there are in the collection, the score of the most relevant tweet and the time required to satisfy the request:

FIGURE 5 – STATISTICS BOX

ii) THESAURUS

The purpose of this page is to allow a user to create their own synonym thesaurus to expand the query while searching. Figure 6 shows the process of creating a dictionary. The user enters a minimum of two terms (less than two would not make sense in the case of synonyms) and creates a dictionary entry. After creating as many entries as he likes, the dictionary is ready to be downloaded by clicking on the green button.

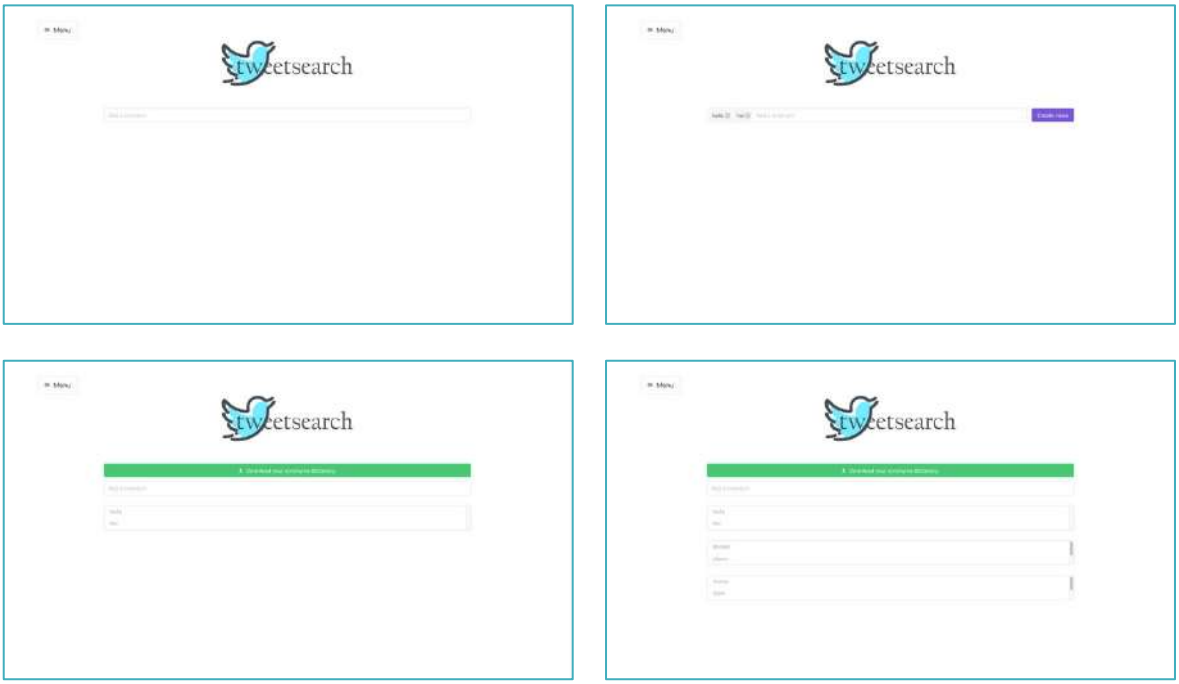

FIGURE 6 – SYNONYMS THESAURUS CREATION PROCESS

The file will be downloaded to the download folder in a file called *synonyms.json*. I chose to save the dictionaries in json format because I am more familiar with this type of file. Also, I needed to define a standard file format for the dictionaries so that I could process them all in the same way when searching.

Once the user has created its own synonym thesaurus, he can import it into the Search page by clicking on the *Upload Thesaurus* button. The scripts will automatically process the inserted file, expanding the query.

iii) CREDITS

The page contains information about the birth of this project and I briefly introduce myself.

4. EVALUATION

In order to test how users perceive *tweetsearch* I performed a user evaluation, which «refers to a collection of methods, skills and tools utilized to uncover how a person perceives a system before, during and after interacting with it»[7].

[7] https://en.wikipedia.org/wiki/User_experience_evaluation

I asked 5 real people to be participant to this user evaluation. They are 3 of my companions from USI and a friend of mine who studies computer science at ETHZ. So, everyone has good IT knowledge, uses search engines frequently and has an idea of how they work. I then built 5 artificial tasks for each user to perform, that I presented in the following order of difficulty:

1. Make a search about what you want and report how many relevant documents are found.
2. Search for tweets containing hashtag «hello» and scroll all the tweets to find the one last retrieved.
3. Search for all the tweets containing the emoji «□».
4. Search for the first tweet containing information about «obama» and «trump» and report the date of the tweet, number of likes and retweets.
5. Create a synonym thesaurus in which «good» is a synonym of «bad», download, use and remove it.

I decided to collect data using the Google Forms[8] service offered by Google, which automatically presents the results in understandable graphs. The data are semi-quantifiable thanks to the Likert charts that I used to collect them. I have divided the form-data collection into three parts: the first consists of asking the name and the consent to the user. The second is carried out during the performance of each task, and I ask questions related to the current task. The same questions are proposed for all tasks. The third is carried out at the end of the experiment and contains questions related to the general use of the system. In Figure 7 I have reported the questions asked.

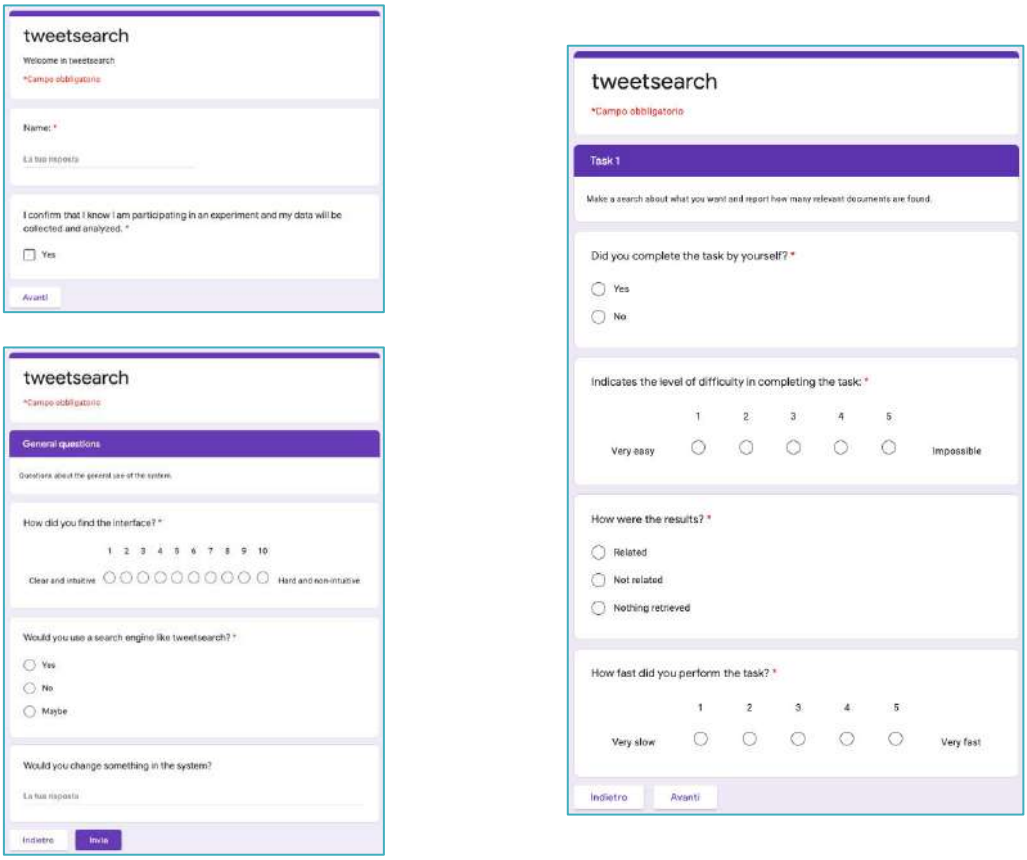

FIGURE 7 - THE QUESTIONS I ASKED USERS DURING AND AFTER THE SYSTEM EVALUATION

[8] https://www.google.com/forms/about/

METHODOLOGY

The conduct of the user evaluation is simple: I briefly introduce my search engine, explaining its nature and its characteristics at a high level, without influencing the user. After that, I give the user 5 minutes to explore and become more familiar with *tweetsearch*. After this training phase, I provide the first task to be performed, the easiest one, in an unlimited time. The user can perform as many queries as he wants.

When the user completes the task, he receives the first part of the form, in which he has to answer some questions. If the user fails to complete the task by himself, he is helped: this is taken into account in the form-data collection.

Once all the tasks have been performed and the questions related to them have been answered, the last part of the form is presented to the user. Here, we collect information about the general use of the system.

5. RESULTS

I present and comment the graphs I have built on the basis of the data collected.

GENERAL

- Figure 8: 1 = 'Clear and intuitive' | 5 = 'Hard and non-intuitive'

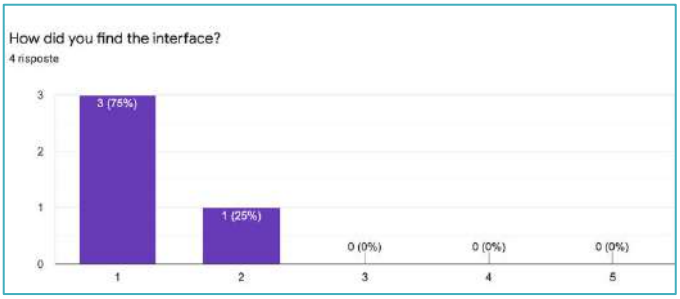

FIGURE 8 - USABILITY OF THE INTERFACE

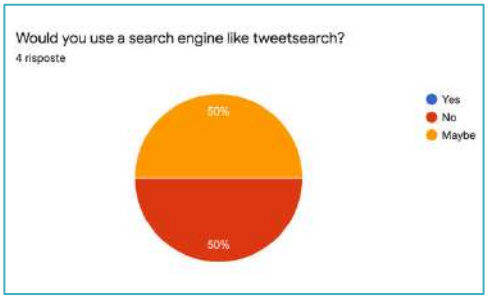

FIGURE 9 - WOULD YOU USE TWEETSEARCH?

Most of the participants liked the interface, finding it simple and easy to use. Also, for this reason they were able to complete the tasks without encountering particular difficulties. But none of my users would use *tweetsearch*. Half only *Maybe*, while the other half would not use it. I investigated the reason by asking the user directly about the reason for this choice. All stated that they would not use it simply because they do not use Twitter, and therefore they would not have the need.

TASK-RELATED

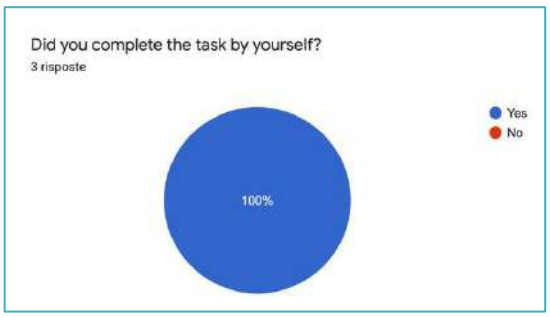

FIGURE 10 - ABILITY TO COMPLETE TASKS BY YOURSELF

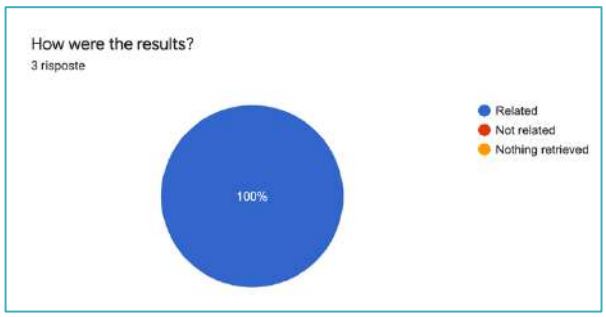

FIGURE 11 - RELEVANTNESS OF RESULTS

The two questions shown in Figure 10 and Figure 11 were asked for each task. In all cases the answer was the same: the task was within the reach of the user, who was able to complete it without needing help, and the results were *Related*.

- Figure 12:
  - Level of difficulty: 1 = 'Very easy' | 5 = 'Impossible'
  - Speed in executing task: 1 = 'Very fast' | 5 = 'Very slow'

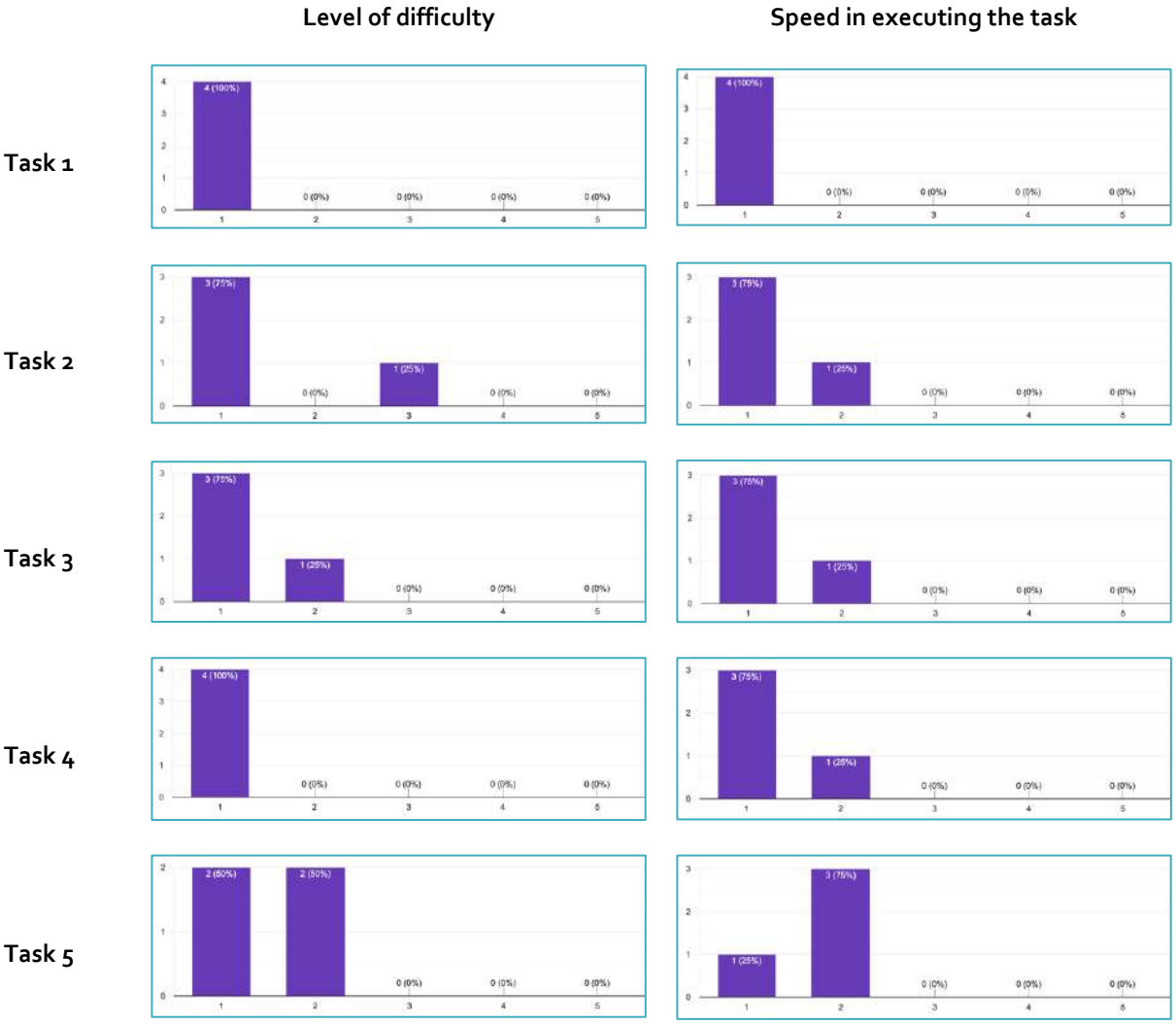

FIGURE 12 - DIFFICULTY AND SPEED OF EXECUTION OF EACH TASK

As you can see, all the tasks were found to be quite simple, and performed in a reasonable time. The first task, as expected, was the easiest and quickest to complete, as I wanted the user to start with a simple one. The one that generally took the longest time is the 5, concerning the creation and use of a thesaurus. Users did not take too long to perform this task but based on the timing to complete the previous four tasks, even simpler, it took a few more moments. Moreover, in a Likert chart from 1 to 5, task 5 was also given a difficulty level of 2 by half of the users. I asked users for clarification, as this had never happened in other tasks. They told me that they managed to solve it without problems, but that they cannot put the difficulty of this task on the same level as the previous four.

Only one user found task 3 more difficult because. This happened because the user forgot to remove the hashtag filter used previously in task 1, and he could not understand why he did not find the expected results. Then, he solved it and moved on to the next one.

SUGGESTIONS

Talking freely with the user, during and after the performance of the tasks, I wrote down some of their suggestions and general reflections on the system. I also collected some through the last question 'Would you change something in the system?'.

In general, all users have noticed and appreciated the simplicity and clarity of the system – exactly what I set out to build – which makes the system intuitive and easy to use. This already came out of Figure 8.

As I mentioned in General, many would not use the system simply because they do not use Twitter, but they later stated that if they did, they would appreciate a system like *tweetsearch*.

A user suggested to add the ability to sort tweets while searching. The idea is very interesting, in fact I had even thought about it, but unfortunately it cannot be applied with notable results. This is because I crawled the tweets using the hashtags they contain (take a look at Scraping section), taking the first results on Twitter. These results were globally the last written and most popular tweets containing that particular hashtag. So almost all of the tweets in my collection were produced on the same day. But with a larger collection, perhaps originated in another way, this feature could surely be added. Also because implementing it would be easy, since the date of each tweet is saved by the spider.

It might also be interesting to put a more prominent warning that you are using a synonym thesaurus or that a filter is selected. This warning already exists but, as revealed in task 2 in Figure 12, some users may struggle.

Another suggestion, coming from a single user, is to add dark mode to the system. I've thought about it and it might be a nice idea, as the interface is very bright, and it might be annoying to the eyes. This feature may also be added in the future.

6. CONCLUSIONS

I am satisfied with *tweetsearch*. I set myself some goals and managed to build a search engine that respected them all: intuitive and easy to use. This also emerged from the user evaluation.

The system appears to be reliable, in the sense that given a query it returns the most relevant tweets and satisfies the emoji search and query expansion. Both of these features are simple to use, and users have managed to complete them without too many problems.

To conclude, given the feedbacks and my impressions on the project, the base of the project is solid and well designed, ready to be expanded in the future with more features such as temporal sorting of tweets.

7. SUBMISSION FOLDER

- Information about how to run the project: README.txt
- Collection of tweets: tweets.json
- Report: tweetsearch_report.pdf
- Presentation: tweetsearch_presentation.pptx
- Project folder: tweetsearch
- Scraper folder: twitter
- Hashtag-search terms: words.txt
- User evaluation results: tweetsearch_data.csv
- Solr folder: solr-8.6.2