Personalized Search Engine for Microblog Content

Information Retrieval Project Report

Matteo Costantini 795125
Dario Gerosa 793636
Michele Perrotta 795152

January 2019

1 Introduction

The goal of the project is to build a Search Engine for tweets crawled from Twitter. The solution must also offer personalised results with respect to a query using document-based personalization approach. The user have the option to select a topic and submit a query which will then customise the results based on the documents provided by the user to the system.

The proposed solution implements a Search Engine based on Lucene with re-ranking of the results to take into account the nature of the documents and the platform from which they come from (e.g. retweets and author's followers). The personalization is performed using a *bag-of-words* model with query expansion. All those functionalities are exposed via a Spring web service and a webapp written in Python with Flask allow to submit a query to the Search Engine and display the results.

The crawler used to gather tweets to be indexed by the Search Engine is written in Python using a wrapper of the officials *Twitter's APIs* and uses a *PostgreSQL* database to store the raw documents before pre-processing and indexing.

2 Crawling Process

The first step consisted in the implementation of a crawler in order to retrieve tweets from Twitter. For the assigned task we decided to opt for a focused crawling approach. A list of user has been selected for each topic of interest and for each user the crawler retrieved at most 3 200 tweets (which is the upper limit imposed by Twitter APIs). Each list has been chosen using the Twitter's *lists* feature to find relevant profiles which tweets

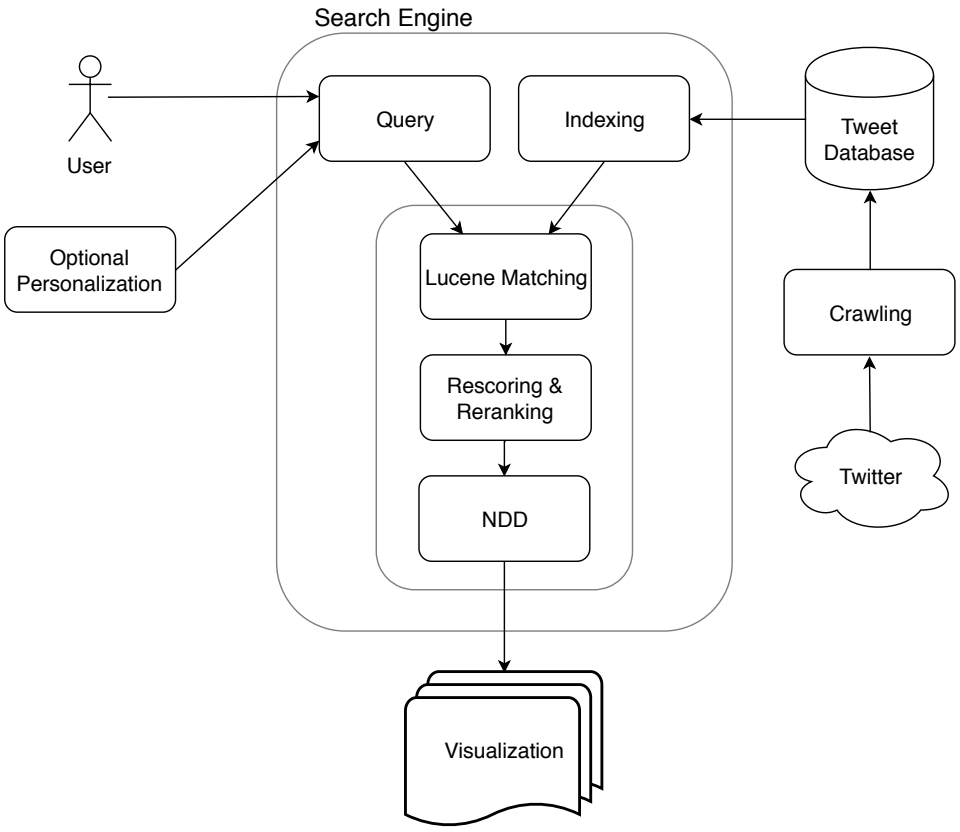

Figure 1: Overview of the system

about the given topic. The crawler has been implemented in Python using the `tweepy` package which wraps Twitter's official APIs using a PostgreSQL database to persist the data and the `multiprocessing` module to speed up the crawling process. The first phase consists in retrieving all the information regarding the selected users and storing them in the database. The second step is where the actual crawling of the tweets takes place. The crawler has been designed as a concurrent system: each process selects a new user to be processed from a shared queue containing all the users whose tweets need to be saved, then retrieves all the available tweets and after a basic pre-processing stores them in the database.

2.1 Dataset

In total 35 699 286 tweets have been crawled from 16 185 different twitter users after discarding all the tweets in a language different than english (based on the `lang` field in the JSON). The crawler stored, for each user, its followers, the topic to which the user belongs and the raw JSON returned by Twitter's APIs. In the same way, for each tweet

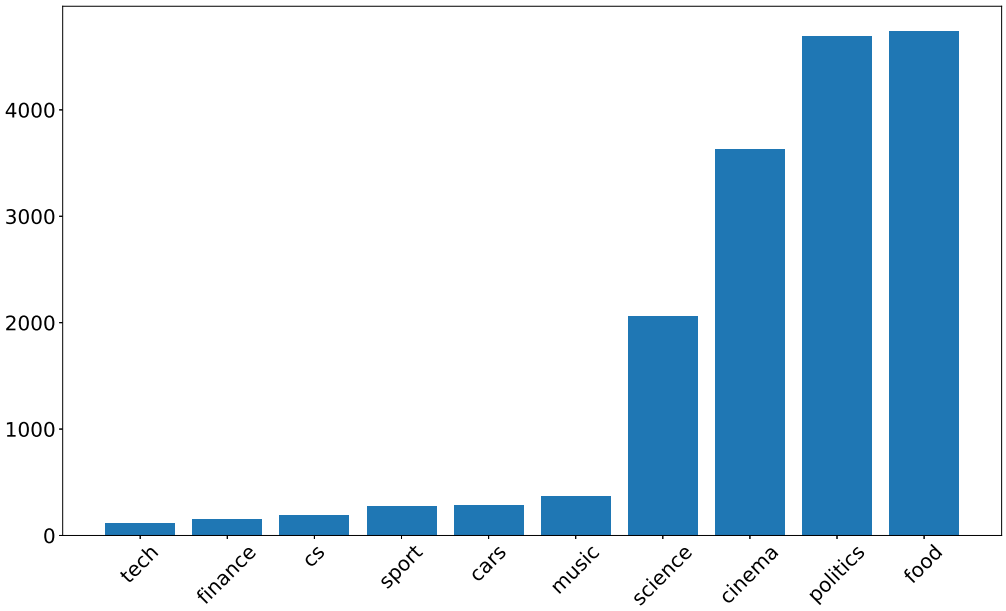

Figure 2: Number of users for each topic

its author and the corresponding raw JSON has been stored.

In figure 2 the distribution of users for each topic taken into account. It can be seen that the number of users is not uniformly distributed among topics and four of them have significantly more users. We decided not to use the same number of users in each topic since in the real world - not necessarily for the same categories used in this project - we expect that there are topics more discussed than others.

Percentile	**Follower**
5	226
10	391
20	746
30	1 210
40	1 879
50	2 954
60	4 625
70	7 851
80	14 563
90	39 785
95	105 922
97	231 844
99	1 262 254
99.5	3 120 360
100	104 683 236

Table 1

In Table 1 instead are reported the percentiles of the followers. It can be seen that almost

30% has less than 1 000 followers and about 5% of the most followed users have more than 100 000 followers with the majority of the others having between 10 000 and 40 000 followers.

3 Indexing Process

To index the documents a custom Lucene Analyzer has been adopted. The implemented custom Analyzer is based on the *Lucene Classic Analyzer* in addition of some *TokenFilter*.

1. **Addresses filter**: discards URLs and email addresses. Tweets contain usually shortened urls which doesn not carry any kind of information.
2. **Classic Tokenizer**: produces tokens using spaces and punctuation as splitting point and discards them. Unlike the Standard Tokenizer, the Classic Tokenizer preserves URLs, email addresses and strings with alphabetic characters and numbers separated with hyphens. Special characters such as `( ) [ ] &` and so on are considered splitting points (e.g. "AX-02", "21:35", "larry@google.com", "www.duckduckgo.com" are preserved and "document-based" is splitted discarding hyphen).
3. **Lower case transformation**.
4. **Stop word removal**.
5. **Porter stemming**.

The structure of the Lucene Document that represent a Tweet is composed of several Fields, each of which is used to either perform queries on it or to compute a custom score for the matching phase. Here is reported the list of the employed Fields with the respective use.

- `TWEET_ID`: the Id of the tweet which is then used to retrieve it for display
- `DATE`: creation date, used both to perform range queries and to chronologically order the results during the first matching phase described in 4.1.
- `TEXT`: the textual content of the tweet on which the query is made.
- `HASHTAG`: the list of hashtags used in the tweet used by the Search Engine jointly with `TEXT`

- `IS_QUOTE`, `IS_RETWEET`, `HAS_URLS`, `RETWEET_COUNT`, `FAVORITE_COUNT`, `USER_FOLLOWERS`, `USER_FOLLOWING`: fields used to re-rank results based on the nature of social platform (Twitter).

4 Search Engine

The Search Engine allows two different types of queries. The first one is based on the Boolean Model which will retrieve all relevant documents, followed by a chronological ordering of the tweets to select the n most recent documents which is finally followed by a re-ranking of the documents to account for other factors (such as the number of retweets). The motivation that led to the implementation of a chronological search is that tweets tends to become less important over time which is a consequence of the nature of Twitter, i.e. an on-line data streaming social media.

The second type of search, on the contrary, uses the standard Lucene matching system (Boolean Model to retrieve the relevant documents followed by the *Vector Space Model* using Lucene's *TFIDFSimilarity* - a revised cosine similarity - to compute the score of each document) to retrieve n relevant documents. This phase is then followed by the same re-ranking strategy applied described for the first type of query.

Lucene's Scoring Formula

VSM does not require weights to be *Tf-idf* values, but *Tf-idf* values are believed to produce search results of high quality, and so Lucene is using *Tf-idf*. VSM score of document d for query q is the *Cosine Similarity* of the weighted query vectors V(q) and V(d):

$$cs(q,d) = \frac{V(q) \cdot V(d)}{|V(q)||V(d)|} \tag{1}$$

Where $V(q) \cdot V(d)$ is the dot product of the weighted vectors, and $|V(q)|$ and |V(d)| are their Euclidean norms.
Lucene refines VSM score for both search quality and usability:

- A different document length normalization factor is used, which normalizes to a vector equal to or larger than the unit vector: doc-len-norm(d).
- At indexing, users can specify that certain documents are more important than others, by assigning a document boost. For this, the score of each document is also multiplied by its boost value doc-boost(d).

- Lucene is field based, hence each query term applies to a single field, document length normalization is by the length of the certain field, and in addition to document boost there are also document fields boosts.
- The same field can be added to a document during indexing several times.
- At search time users can specify boosts to each query, sub-query, and each query term, hence the contribution of a query term to the score of a document is multiplied by the boost of that query term query-boost(q).
- A document may match a multi term query without containing all the terms of that query (this is correct for some of the queries).

Under the simplifying assumption of a single field in the index, we get Lucene's Conceptual scoring formula:

$$s(q,d) = \text{query-boost}(q) \cdot \frac{V(q) \cdot V(d)}{|V(q)|} \cdot \text{doc-len-norm}(d) \cdot \text{doc-boost}(d) \tag{2}$$

In any case, the first step is the query construction which is done following these steps:

1. QueryParser creates a boolean clause, parsing the input text provided by the user.
2. The query is expanded and modified into a disjunction between the original clause and the queries on the Hashtag field (one for each term in the query).
3. If a date range or limit is specified the query is expanded with a RangeQuery on the the date field.
4. Finally, if a topic is given, the query will be furthermore expanded to account for user's interest.

4.1 Re-scoring Mechanism

This section explains how the actual re-ranking is performed after the Lucene's matching function has retrieved a set R of n documents. The final score of the documents is calculated as a linear combination of different scores, the first being the score computed by Lucene in the previous step, called *score* in the following formulas. Most of the

scores employed were introduced in the paper *Raking approaches for microblog search.*[1] The first term of the linear combination is given by the *base score* (3), bS that is, the normalized score computed by Lucene multiplied by a constant factor lw (in the current implementation $lw = 3$).

$$bS = lw \cdot \frac{s_d}{max_{i \in R}\ s_i} \tag{3}$$

The second term is the *follower score* (4) and takes into account the popularity of the author of a given tweet in the set of results. The idea is that a user is influential and shares useful information if he has a lot of followers. This could also happen if they are "socially active". In this case a user will have a high number of followers but will also follow a high number of users itself. Thus the number of followers is normalized by the sum of the number of followers ufi_d and the number of people the account is following ufo_d. Finally the term fw is a constant factor to weight the score (in the current implementation $fw = 1$).

$$fS = fw \cdot \frac{ufi_d}{ufi_d + ufo_d} \tag{4}$$

Another factor that it is taken into account, similar to the *follower score* is the *retweet score* (5), that is how much the tweet has been shared on the platform. The idea is that the higher the number of retweets r_d and favorites fav_d the higher is the quality of the information shared in the tweet. This score is divided by the *retweet score* of the document with most retweet and favorites in the set of documents R in order to normalize the score in the range $[0, 1]$ and the is multiplied by a factor rW (in the current implementation $rw = 1$).

$$rS = rw \cdot \frac{r_d + fav_d}{max_{i \in R}\ r_i + fav_i} \tag{5}$$

Since a tweet can be a quote ($q_d = 1$) of another tweet or a retweet (r_d), the final score takes also this information into account in the *quote score* (6) decreasing the score of the tweet if it's either a retweet or a quote. Both qw and rw are set to -0.5.

$$qrS = q_d \cdot qw + r_d \cdot rw \tag{6}$$

The *length score* (7) is based on the assumption that longer tweets contains more information (considering also that tweets are already very limited in length) and also to counterbalace

[1] Rinkesh Nagmoti, Ankur Teredesai, and Martine De Cock. "Ranking approaches for microblog search". In: *Proceedings of the 2010 IEEE/WIC/ACM International Conference on Web Intelligence and Intelligent Agent Technology-Volume 01.* IEEE Computer Society. 2010, pp. 153–157.

the tendency of penalizing the score of longer documents implemented in Lucene. The score again is normalized dividing it by the longest document in the set R and multiplying it for a constant factor $lw = 0.5$.

$$lS = lw \cdot \frac{l_d}{max_{i \in R}\, l_i} \tag{7}$$

The last factor that is taken into account is the presence of a URL in the tweet (8). Tweets are short documents and it is hard that they will contain a lot of information. The presence of an URL signals the intention of sharing additional information related to the tweet and that information can be retrieved at the specified URL. This value is constant for every tweet and is either 0 if the document does not contain a URL ($u_d = 0$) of uw if it does ($u_d = 1$).

$$uS = uw \cdot u_d \tag{8}$$

The final score (9) is thus calculated as the sum of all the previous terms and is used to re-rank the set of documents R retrieved in the previous step.

$$S = bS + fS + rS + qrS + lS + uS \tag{9}$$

4.2 Personalization

The personalization has been implemented using a *bag-of-words* model and query exapansion. For each topic the user select a number of documents that represent its interest and those documents are used to extract a bag of words. The personalization is topic dependent and the tweets selected by the user on a specific topic will only personalize queries against that specific topic.

To test the system 5 predefined user profiles have been created, each with at least 3 topics and at least 10 document for each topic. A custom user profile with the possibility of selecting documents interactively has also been implemented.

The bag of words for each topic is extracted from the documents provided by the user. Two strategies have been take into account: the first strategy used the same analyzer used during the indexing of the dataset to create a new index specific to the given topic and the given user and then the set of terms is used as bag of words; The second strategy is similar to the first one, bug only a subset of the terms is used to represent the user interests. For each term has been computed the tf on the newly created index and the idf on the original dataset. The terms were then sorted using the product of the two factors and only the most informative ones, for example the top 30 were selected for the bag of words.

The personalization is then achieved through a query expansion updating the original query to a disjunction between the original query and the expansion terms. For each term in the bag of word related to the topic chosen by the user, a new disjuction is added to the query.

4.3 Near Duplicate Detection

Finally, before returning the set of documents, the Query Engine performs Near Duplicate Detection on the set of documents retrieved in order to find similar tweets and to flag them. This is done using the *Overlap Coefficient* reported in 10, where the set of terms for each tweet consists of its bi-grams. If the $overlap(T_{d1}, T_{d2}) > 0.8$ for some documents $d1$ and $d2$ the system chooses the one with lower score and mark it as duplicate. The documents will still be returned and eventually visualised, but they will be flagged as duplicates.

$$overlap(X, Y) = \frac{|X \cap Y|}{min(|X|, |Y|)} \tag{10}$$

Another measure, similar to the Overlap Coefficient, is the Jaccard coefficient which is faster and usually used instead of the Overlap coefficient. The choice fell on the latter because tweets are extremely short documents. Two overlapping tweets differing only in one word would be marked as non duplicates because the low number of terms in the document. This problem arises less using the Overlap coefficient and for that reason it has been chosen instead of the Jaccard coefficient.

5 Experiments

The personalization has been tested searching for a generic word, such the word "car", both in the non-personalized mode and in the personalized mode (with topic "Cars" and user profile with an interest for brands of car such "BMW" and "Tesla"). The expected and empirically confirmed behaviour is a moderate modification of the results: new tweets about the user interests appear and the generic ones go down. In case of chronological search the Search Engine preserve the timeline and do not let many new tweets emerge. Using the same user as the previous example, which is interested in *BMW* and *Tesla*, and choosing again the topic *Cars* to perform a personalized search, if the query is based on words like *turtle* which is not related in any sense with the chosen topic, the personalization does not affects the results. An analogous experiment is searching for an ambiguous word such as "stock" and looking how a personalization with topic "finance" and user

profile with interest in some corporations and market trends provide enough contextual information to disambiguate results.

6 Conclusion

In this project we realised a custom Search Engine focused on microblog texts (Tweets, in this case) which is characterised by, as the name suggest, very short documents that cannot convey a lot of information. The techniques we adopted in the scoring system tried to capture the nature of the social network using the popularity of the tweet itself or of the author. Applying Near Duplicate Detection we tried to identify duplicates, retweets in particular, since they are widely used in Twitter and would have added too much noise to the search results.

Finally we adopted a simple *bag-of-words* model to personalize the query results. While it seems effective with queries on the entirety of the dataset (though far from perfect) the personalization fails at the customization of query results chronologically ordered.

Possible future developments are:

- Implement a custom Lucene Scorer to achieve a better personalization even on chronological search.
- Replace the *bag-of-words* model with a more effective one to improve the personalization.
- Test the Search Engine on an annotated tweets corpus.
- Statistically study on tweets fields such `RETWEET_COUNT`, `FAVORITE_COUNT`, `HASHTAG`, `FOLLOWERS`, `FOLLOWING`.
- Statistical study on how changing the scoring parameters affect ranking.
- Perform parameter optimisation as the weight currently used are based on empirical experiments