Twitter search engine with personalized search

Belotti Federico 808708
f.belotti8@campus.unimib.it

Ventura Samuele 793060
s.ventura6@campus.unimib.it

Contents

1 Introduction

The project implements a Twitter search engine with a personalized search based on user profile. The goal of this search engine is to retrieve tweets that may be of interest to the user based on his personal profile.
For the development of the search engine **ElasticSearch** has been chosen as main tool. The project has been written in **Python 3.7** using the `elasticsearch` library, that wraps ElasticSearch's APIs to Python.
Flask, a simple web application framework, and a basic, but meaningful, web interface have been chosen to reach and retrieve Twitter's documents from the Elasticsearch index.
The dataset has been automatically created with the usage of `tweepy`, a python wrapper to the Twitter APIs: we have collected tweets to be used as documents and retrieved by Elasticsearch given a user query (**query tweets** from now on), and to create users profiles (**users tweets** from now on) for the personalized search.
To represent a generic user profile we have implemented two distinct approaches: one that simply represent a user by taking its most common words based on the *term frequency* weight, and one that leverages a particular type of distributional *word embeddings* [1, 2, 3].

Table 1: Dataset, composed by query and users tweets

Table 2: Tweets per topic

Topic	**Tweets**
Music	130768
Sport	168584
Politics	38262
Technology	104227
Economy	122821
Cinema	85047
Total	649727

Table 3: Tweets per user

User	**Tweets**
Aaron Donald	2919
Barack Obama	3231
Boris Johnson	3220
Elon Musk	3214
Eminem	935
Emma Watson	1757
Green Day	2226
IBM	3235
Joseph Stiglitz	763
Marcus Rushford	906
Ryan Reynolds	1500
Yunus Centre	3232
Total	27138

2 Dataset

To crawl tweets we have used `tweepy`, a simple wrapper to Twitter's APIs for python and we have chosen six different topic in which we are interested in: **music**, **sport**, **politics**, **technology**, **economy** and **cinema**. Statistics about the dataset are shown in table 1.

2.1 Crawling of query tweets

Tweets used to create the index has been collected in a temporal period from 21-01-2020 to 04-02-2020. From the json retrieved by the Twitter's APIs only a sub-group of properties have been saved and used, and they are listed in table 4. For a global comprehension of the tweet structure that one gets in response of a tweepy query, is shown an example in the listing below:

```
{
    "id": 306159,
    "tweet_id": "1223308223404478465",
    "created_at": "2020-01-31T18:13:03",
    "text": "#nowplaying #LadyGaga | Always Remember Us This Way ...",
    "name_user": "BB RADIO Playlist",
    "followers_count": 2591,
    "like": 0,
    "retweet": 0,
    "profile_image_url": "https://pbs.twimg.com/profile_images/...",
    "tweet_url": "https://twitter.com/_BB_RADIO_MUSIC/status/...",
    "country": "DE",
    "location": {"type": "Point", "coordinates": [13.1199934,52.381905]},
    "topic": "music"
}
```

Listing 1: Structure example of tweet

In the listing is shown a location field that has not been used: initially we had decided to use location linked to the tweet (latitude and longitude) as another relevance dimension but for the minor presence of this property in all tweets (3000 on 500000) we have decided to not use it.

2.2 Crawling users tweets

For every of our six topics we have chosen two distinct users to crawl tweets for and the only informations we have kept is the tweet text and its ID, as shown in the example below:

```
{
   "1224380698783555584": "@CathieDWood @sbarnettARK @jwangARK @ARKInvest First,
        we need to make it super safe &amp; easy to use, then determine greatest
        utility vs risk. From initially working to volume production &amp;
        implantation is a long road. As with vehicle safety, it will be much
        harder to pass our internal standards than minimum regulatory standards.",
  "1224218624514314240": "@flcnhvy @Tesla @thirdrowtesla Ok sure \ud83d\ude00
      Btw, we recruit great engineers from almost anywhere in the world, so this
       shouldn\u2019t be thought of as USA only. Also, work location can be Bay
      Area (preferred), but Austin (many of our chip designers are there) or
      potentially any Tesla Gigafactory.",
  "1224204420273098752": "Nothing medically dangerous, but 5g is getting a bit
      too greedy with their spectrum land grab https://t.co/1NlQCbeyE5",
  "1224203828209311745": "@TeslaGong @indiealexh @mojosusan @BLKMDL3 @ElonsBrain
       @CathieDWood @ARKInvest Intuitively, that should be straightforward to
      address, but there could be complexities I\u2019m unaware of"
}
```

Listing 2: Structure example of tweet

The complete list of users is shown in table 5.

3 User profile

In order to retrieve documents relevant to a particular user based on its interests, we have decided to model the user profile in two distinct ways: a **static** one, that expands a user query with a priori set of terms, that are the most common terms given the term-frequency matrix, and a **dynamic** one, that leverages word embeddings [1, 2, 3], expanding a user query with the set of terms that are most similar with the query terms, given their vector representations.

In order to create meaningful vectors representations, not all the words present in the corpus have to be taken into account, so every tweet is first preprocessed with a procedure explained in section 3.1.

Field	**Description**
ID	unique identifier of the tweet
created_at	the date (day and hour) when the tweet has been created
text	the corpus of the tweet
name_user	name of the user that published the tweet
followers_count	user followers number
favorite_count	number of likes the tweet has received
retweet_count	number of time the tweet has been retwitted
profile_image_URL_https	link to the user profile image
tweet_url	original tweet url
country_code	place associated with the tweet

Table 4: Tweet fields used in this work

Since we want to be able to recognize phrases, group of words better to be considered as a single one, e.g. the newspaper ”New York Times” can be considered as a single word ”New_York_Times”, to enrich the vectors representation we have also decided to learn models that are able to automatically identify those phrases, in particular we have learnt models to recognize bi-gram and tri-gram phrases, i.e. phrases of two and three words respectively, as specified in [2]. The score implemented to choose if two words are a bigram is the **normalized Pointwise Mutual Information**, defined as:

$$\mathrm{PMI} = \ln \frac{p(x,y)}{p(x)p(y)} \tag{1}$$

$$\mathrm{PMI}_n = -\frac{\mathrm{PMI}}{\ln p(x,y)} \tag{2}$$

where the **Pointwise mutual information** (**PMI**) is a measure of how much the actual probability of a particular co-occurrence of events $p(x,y)$ differs from what we would expect it to be on the basis of the probabilities of the individual events and the assumption of independence $p(x)p(y)$ [4].

Field	**User**	**Description**
Sport	Aaron Donald Marcus Rashford	American football player of the NFL English soccer player of the PL
Music	Green Day Eminem	American rock band The greatest rapper alive
Cinema	Emma Watson Ryan Reynolds	English actress American actor
Technology	IBM Elon Musk	American technology company Engineer and technology entrepreneur
Politics	Barack Obama Boris Johnson	Ex USA president Actual UK president
Economy	Joseph Stiglitz Yunus Centre	American economist Policy institute for social business

Table 5: Users profiles used in this work

3.1 Preprocessing

The preprocessing stage is fundamental to every NLP and IR pipeline, and can be carried out in multiple ways: our full algorithm is presented in Algorithm 1.

From the intuition that user hashtags can be very informative of its interests, and so useful not only for retrieving meaningful documents, but also for learning a better vector representation, we have developed our preprocess stage in a **hashtags-driven** way: in order to retain the information contained in the hashtags text, we remove the hashtag and we split the hashtags text by capital letters, so for example ”#freeIraqiPeople” becomes ”free Iraqi People”.
The removal part has been carried out with the usage of handcrafted regular

Algorithm 1 Preprocess

```
procedure PREPROCESS(tweet)
    tokens, yeah_tokens $\leftarrow \{\}$
    twitter_words $\leftarrow$ {”rt”, ”via”, ”fav”}
    tweet $\leftarrow$ remove_urls(tweet)
    tweet $\leftarrow$ remove_mentions(tweet)
    tweet $\leftarrow$ remove_emojis(tweet)
    tweet $\leftarrow$ remove_numbers(tweet)
    tweet $\leftarrow$ remove_hashtags(tweet)
    tweet $\leftarrow$ fix_contractions(tweet)
    tweet $\leftarrow$ remove_punctuation(tweet)
    tokens $\leftarrow$ tokenize(tweet)
    for $\forall$token $\in$ tokens do
        token $\leftarrow$ lemmatize(lower(token))
        if not (
                token is not alphanumeric
                or token $\in$ stopwords
                or token $\in$ twitter_words
                or (token is alphabetic and len(token) $< 2$)
          ) then yeah_tokens $\leftarrow$ yeah_tokens $\cup$ token
        end if
    end for
    return yeah_tokens
end procedure
```

expressions[1], and in particular:

- **remove_urls**: removes all the urls found in the tweet[2]
- **remove_mentions**: removes the Twitter mentions, so all the strings starting with the "@"
- **remove_emojis**: removes all the emojis[3]
- **remove_numbers**: removes sequences of only numbers that are not possible dates (years from 1800 and 2100), e.g. "H2P" and "2020" stay as they are, while "22" gets removed
- **fix_contractions**: fixes contractions, e.g. "don't" becomes "do not"[4]

We have used NLTK to tokenize the tweet and the WordNet [5] lemmatizer available from the NLTK package.

3.2 TF

Term frequency is a statistic, that has been widely used in the IR field due to its effectiveness and simplicity, intended to reflect how important a word is to a document in a collection or a corpus: the weight of a term that occurs in a document is simply proportional to the term frequency (Hans Peter Luhn).
In its simplest form the term frequency is computed as the raw count of the term in the document:

$$\text{tf(t,d)} = f_{t,d} \tag{3}$$

such that $f_{t,d}$ represent how many times term t appears in document d.
In order to obtain the probability that the term t appears in the document d, one can divide the raw count by the document length:

$$\text{tf(t,d)} = \frac{f_{t,d}}{|d|} \tag{4}$$

In order to prevent the bias introduced by the document length, the term frequency can be calculated as:

$$\text{tf(t,d)} = \frac{f_{t,d}}{\max_{t_i \in d} \text{tf}(t_i, d)} \tag{5}$$

[1] https://github.com/Sbarbagnem/sbarbasearch/blob/master/preprocess/tweet_preprocess.py

[2] https://gist.github.com/dperini/729294

[3] https://github.com/carpedm20/emoji

[4] https://github.com/kootenpv/contractions

So we have decided to represent a static user profile as a bag of words (BoW) with the most occurring n terms given the term frequency calculated as in equation 3. So, for example, the most 15 occurring terms for the Boris Johnson user are: "brexit", "get", "uk", "done", "today", "country", "people", "great", "conservative", "new", "back", "vote", "support", "let", "pm".
It's simplicity has drawbacks, in fact with the sole term frequency one cannot take into account two fundamental aspects: the fact that common terms with a higher term frequency does not provide much information as rare words sparse in the collection and the meaning of the words appearing in the corpus. The first aspect can be treated with the usage of td-idf weight, that filters out common terms, while the second with approaches that leverage the **distributional hypothesis**: words that occur in the same contexts tend to have similar meanings (John Rupert Firth), as the word embeddings does.

3.3 Word embeddings

Since [1] and [2], vector representations of natural language words has been widely used in many NLP tasks, due to their compact, meaningful and efficient representation.
Problems arise when one has vectors from different vector spaces and wants to compare them, for example through cosine similarity: since the basis of the spaces are different, vectors from different vector spaces cannot be compared, unless one finds that linear transformation that allows their comparison as if they lie in the same vector space [6]. The authors in [6], given representations of words from two different vector spaces, try to learn a linear transformation that is orthogonal (thus invertible) that can map any vectors from the two different spaces into a single shared one, in which vectors can be compared. If the spaces are more than two one can linearly align a new space with the already aligned ones.
For our purpose, words vector representations are used to learn a model for every single user and one also for the query, such that a user is represented as the set of words embeddings associated to the terms that appear in its tweets, same for the query.
So when a user search for a query, our system finds the most similar words to the query terms in the user space by cosine similarity, given their vector representations, and expands the query with those words, with the intuition that they could better represent the user given the query.
In order for different vector spaces to be comparable, they have to be aligned: in our work we rely on the alignment method of [3], in which, based on [1, 2],

they try to align vector representations from different period of time, but can be extended in multiple ways, for example to learn bias of articles from different newspapers.
The learning method goes as follows:

1. Learn a CBOW [2] model on the concatenation of query and users tweets: this serves as a global compass for the alignment of the others slices

2. Initialize the target matrix of every slice with the one from the compass: this matrix won't be updated during the training

3. For every slice, learn a CBOW model: in fact only the context matrix will be learned, as the target matrix remains fixed. Vectors in the context matrix will then be used as vectors representations for each word in the vocabulary

After the training phase of the slices, the vector spaces are aligned and can be compared. We have trained word embeddings of 100 dimensions and they are available to download, with all the other data, at https://drive.google.com/open?id=1fi7tyV-fhYGzysV93pR83DPcQcb0Wyhk.
The algorithm is depicted in figure 1 and, thanks to the open source, freely available at https://github.com/valedica/twec.

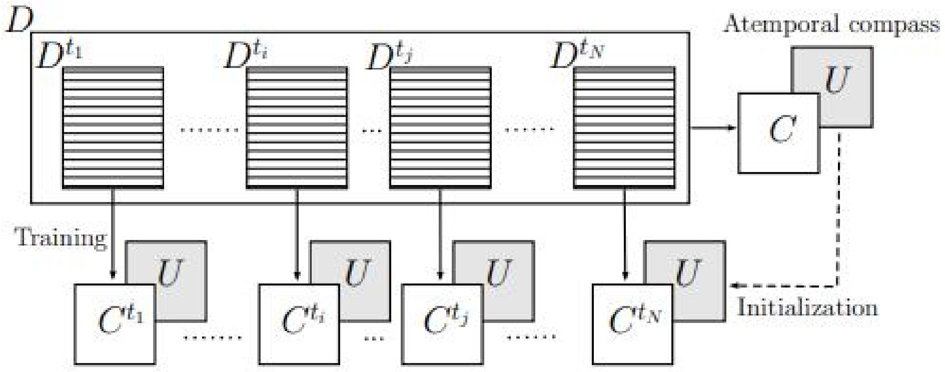


Figure 1: TWEC learning algorithm

4 Search engine

4.1 Index setting

Elasticsearch, to properly index documents, needs a mapping in which one has to define how tweet's fields will be processed, saved and retrieved.
First of all it has been set a **custom analyzer**, that converts a generic string into a stream of tokens or terms; this analysis process needs to happen not just at index time, but also at query time: the query string needs to be passed through the same (or a similar) analyzer so that the terms that it tries to find are in the same format as those that exist in the index. Our custom analyzer has the following properties:

- **tokenizer**: Elasticsearch built-in tokenizer that breaks text into tokens whenever it encounters a white space character
- **char_filter**: Elasticsearch built-in character filter that strips HTML elements from the text and replaces HTML entities with their decoded value
- **filter**: a set of functions that operates on the tokens obtained by the tokenizer
  - **classic**: a built-in classic filter that removes the English possessive from the end of words and removes dots from acronyms
  - **lowercase**: transforms token text to lowercase
  - **remove_digit_token**: removes tokens made of only numbers
  - **remove_link_token**: removes tokens made of only links
  - **remove_punctuation**: removes punctuation except "@#" because relevant for the search of hashtags and users in tweets
  - **remove_length_less_two**: removes tokens with a length shorter than two characters
  - **my_stemmer**: stems words with **porter2** stemmer

As similarity ranking measure between query terms and documents it has been defined a language model with the Jelineck-Mercer smoothing technique:

$$\begin{aligned} p(q|d,C) &= (1-\alpha)\cdot p(q|M_d) + \alpha \cdot p(q|M_C) \\ &= \prod_{t\in q}\left((1-\alpha)\frac{\mathrm{tf}(t,d)}{|d|} + \alpha\frac{\mathrm{tf}(t)}{|C|}\right) \end{aligned} \tag{6}$$

where:

- q is the query
- d is the document
- C is the collection of documents
- $\text{tf}(t, d)$ is the same as in equation 3
- $\prod_{t \in q} \frac{\text{tf}(t,d)}{|d|} = p(q|M_d)$ is the probability that the query has been generated from the language model of the document
- $\prod_{t \in q} \frac{\text{tf}(t)}{|C|} = p(q|M_C)$ is the probability that the query has been generated from the language model of the entire collection
- α is a parameter to define the smoothing trade trade-off

It has been further set a **mapping for the tweet's fields** that defines how every field is stored in the index. It's showed in table 6.

Only the fields "text", "topic" and "country" are indexed because they are the only needed for the retrieval process.

Fields "user", "profile_image" and "tweet_url" are only stored and not indexed because they are only used to show information about the retrieved tweets.

Field	**Type**	**Index**	**Analyzer**
Created_at	Date	True	//
Text	Text	True	custom_analyzer
User	Text	False	//
Like	Integer	False	//
Retweet	Integer	False	//
Profile_image	Keyword	False	//
Tweet_url	Keyword	False	//
Topic	Keyword	True	//
Country	Keyword	True	//

Table 6: Users profiles used in this work

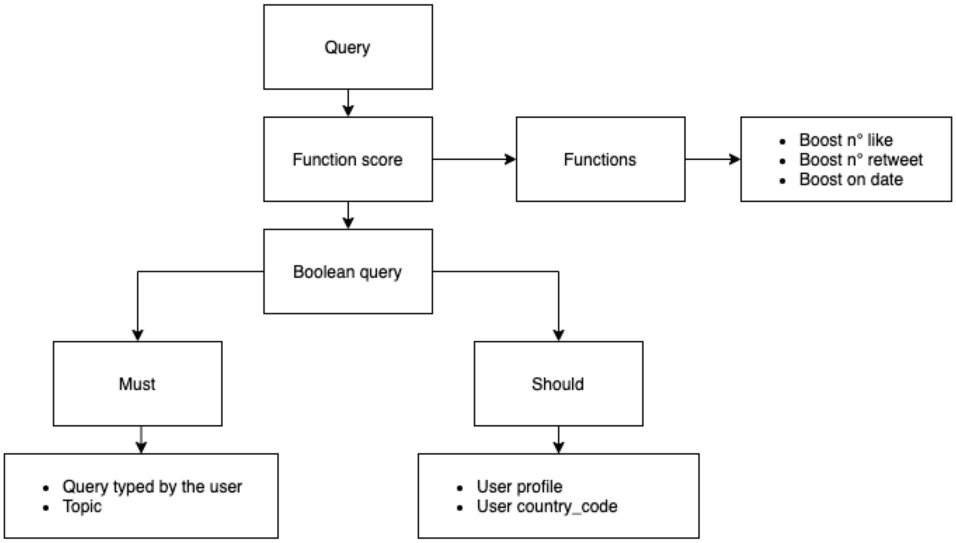


Figure 2: Query tree

4.2 Query explanation

With reference to the general query tree shown in figure 3, the specific structure of the dynamic query is the following:

- The root node **Function score** allows to modify the score of documents that are retrieved by a query. We have decided to improve the Retrieval Status Value (RSV) based on the number of likes, the number of retweets and the date of creation, in such way that the RSV receives a multiplicative boost of $\text{score}(r) \cdot \text{score}(l) \cdot \text{score}(d)$, where:
    - $\text{score}(r) = \sqrt{1+r}$, where r is the number of time the tweets has been retweetted
    - $\text{score}(l) = \sqrt{1+l}$, where l is the number of likes the tweet has received
    - $\text{score}(d) = \exp(\lambda \cdot \max(0, |d - \text{"now"}| - 5))$, where λ is a decay factor and d represent the tweet's field "created_at". This function decays the RSV of tweets that are created after five days from the query search by a factor of λ
- The node **Boolean query** allows matching boolean combinations of other queries based on one or more boolean clauses

- Two nodes that represent clauses of the boolean query: s **must** node for clauses that must appear in matching documents and will contribute to the score, s **should** node for the clauses that should appear in the matching document and can increase the retrieval score.
  In the must node there are:
  - a **query_string** node that takes the query typed by the user and that will be processed by Elasticsearch in order to match it with the "text" field of tweets stored in its index. The tweets retrieved are mostly dependent on this match's value
  - an optional **term** node. Its presence depends on the user's choice in the web interface, if she chooses to **filter by topic**, only tweets with those particular topic are retrieved

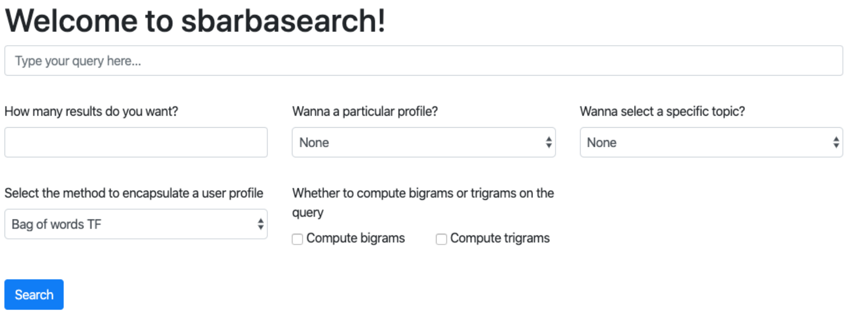
  While in the should node there are:
  - an optional **match** node that represent the **query expansion with terms from the user profile**, if some user profile is chosen in the web interface
  - an optional **term** node that match exactly the "country_code" of the user, if some user profile is chosen in the web interface

To summarize, in addition to the sole RSV based on tweet's text, other **relevance dimensions** have been added to improve the relevance of the retrieved documents: the date in which a tweet has been published, the number of likes has received, the number of times has been retweeted and the country code linked to the tweet.

5 Web Interface

From the web interface of our "sbarbasearch" engine, a user can perform the following choices:

- choose how many result to retrieve
- select a particular user profile for the personalized search
- select a possible topic to filter out the retrieved tweets
- select method to encapsulate a user profile (TF, embeddings, embeddings_mean), and hence possibly expand the query



Welcome to sbarbasearch!

Type your query here...

How many results do you want?

Wanna a particular profile? None

Wanna select a specific topic? None

Select the method to encapsulate a user profile: Bag of words TF

Whether to compute bigrams or trigrams on the query

□ Compute bigrams □ Compute trigrams

Figure 3: Interface

- choose if compute bigrams or trigrams on the query string

The system can be easily run and modified as specified at `https://github.com/Sbarbagnem/sbarbasearch`. All data used in this work are available at `https://drive.google.com/open?id=1fi7tyV-fhYGzysV93pR83DPcQcb0Wyhk`

6 Conclusion and future developments

In this work we have developed a personalized search engine based on user profile for tweets in predefined topics. To model the user profile we have adopted two different methodologies: one static, with a simple but effective bag of words and one dynamic that leverages a particular kind of aligned word embeddings. The search engine has been developed with Elasticsearch, that works in background creating an inverted file structure to store and retrieve documents given a user query.
We believe in the open source community, so our code is hosted freely at `https://github.com/Sbarbagnem/sbarbasearch`. As possible future developments we have:

- try to set a weight for every different relevance dimensions
- try to use different size of embeddings and other techniques to combine vectors
- measure the performance of our system with the typical IR performance measures

References

[1] T. Mikolov, K. Chen, G. Corrado, and J. Dean, “Efficient estimation of word representations in vector space,” *arXiv preprint arXiv:1301.3781*, 2013.

[2] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean, “Distributed representations of words and phrases and their compositionality,” in *Advances in neural information processing systems*, 2013, pp. 3111–3119.

[3] V. Di Carlo, F. Bianchi, and M. Palmonari, “Training temporal word embeddings with a compass,” in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 33, 2019, pp. 6326–6334.

[4] G. Bouma, “Normalized (pointwise) mutual information in collocation extraction,” *Proceedings of GSCL*, pp. 31–40, 2009.

[5] G. A. Miller, “Wordnet: a lexical database for english,” *Communications of the ACM*, vol. 38, no. 11, pp. 39–41, 1995.

[6] S. L. Smith, D. H. Turban, S. Hamblin, and N. Y. Hammerla, “Offline bilingual word vectors, orthogonal transformations and the inverted softmax,” *arXiv preprint arXiv:1702.03859*, 2017.