Estellers, Oriol - 242142

Fuentes, Raimon - 242176

Ribas, Pol - 241620

Github link and TAG: IRWA-2023-part-3

Part 3: Ranking

This Part 3 project is based on the implementation of Part 1&2, since we will work with the pre-processed tweets. Therefore, we will use the code that we had already done, which refers to both the pre-processing and the exploratory analysis, as well as indexing and evaluation. We have also included the improvements recommended in the project's second deliverable feedback, which include outputting the ranked-based results not only with its corresponding id, but also with its text and metadata (likes, retweets,...) as requested in the first part. Moreover, since we are working with conjunctive queries, the retrieved documents will have all the terms in the query. We would also like to clarify that we have selected the relevant documents manually, because we have checked one by one that the documents selected as relevant not only contain the terms in the search, but it actually is relevant to the information need.

For the first part, our task is to create our own score for the classification of the documents. Our main idea is to use some of the information appearing in the dataset that we are not taking into account for any of the tasks. After observing which of the information could be useful, our conclusion was that using aspects such as the number of followers from the user or the number of retweets and likes from the tweet would be some interesting parameters for computing our own score. After careful consideration, we concluded the following points:

- Number of likes and retweets: Among the relevant documents, the ones having more likes and retweets should not be considered more important, but more influential. Hence, we have decided to give more importance to the tweets having bigger numbers regarding the retweets and likes. Moreover, since we are all Twitter members as well, for us it is more common to give a like than a retweet. Hence, we have decided to give more importance to the retweets than the likes, since they are more uncommon to be used. For this reason, we have decided to give the retweets a 60% relevance and the likes a 40%.

- Number of followers: Once we have done this, if we are in a case where two tweets have the same number of likes and tweets, which one would we consider as more important? Here is where we use the number of followers, since a user with less followers is not as likely to get as many likes and retweets as someone who has a bigger number of followers. Therefore, we have also given more importance to users having less followers since they are less likely to receive interactions in their tweets.

Our starting points were the TF-IDF and the cosine similarity functions implemented previously. Since, apart from the popularity, we want to take into account the similarity of content between the tweets and the query as well, we did not remove anything from the previous code, but we added the different aspects that we wanted to consider to make our desired combination.

To implement our own score we added a new dimension which is basically the popularity result, and it ranges between 0 and 1 (1 being the maximum popularity score). Since we can calculate this score for all the documents, but not for the queries, because we do not have likes and tweets for the queries, we set this value to 1 for the queries, that way, the higher the popularity result is, the higher the similarity between the document and the query in the popularity dimension is as well.

Therefore, the dimensions of our vectors are the number of terms plus one, which is the popularity score.

To calculate this score we have created the following formula:

$$P\_SCORE = (n\_likes \cdot 0.6 + n\_retweets \cdot 0.4) / \max(followers \cdot 0.3,\ 1)$$

Here we consider the number of retweets to be more important than the number of likes.

Then we divide by the number of followers of the user of the tweet, that way we give more importance to the popularity of the tweet if it was written by a user with less followers. The denominator is maximum 1, to not give that much advantage to the users with very few followers. If we did not do that, a user with one follower would see his popularity scores become very high, since we would be dividing the number of likes and retweets by 0.3.

After that we just apply the cosine similarity as we did before but with one extra dimension, which is the popularity score, and we can rank the most relevant documents combining the content and the popularity.

We can see in the results that the ranking combines the content and the popularity of each tweet. Since we are combining these two aspects, we do not detect a clear pattern in the ranked results. However, if we try to not take into account the followers of each user, we can observe how the likes and retweets become more important in the final result.

Our scoring method is not perfect and has several pros and cons, which are the following:

- Pros: We are taking into account other features from the tweets and users that might affect the importance of the tweet. Likes and retweets could directly affect the relevance of each tweet, and it is important to take them into consideration as well. Also, by giving more importance to the users having less followers, we are kind of equalizing the possibilities that all the tweets can be relevant.
- Cons: Some of the cons that we have found is that there are other features that could directly affect if a tweet can go viral or not. For example, a user having his account verified is more likely that his tweets go viral rather than someone who doesn’t. Also, since we are using the likes and retweets to compute our score, someone that has his account set as private has less probabilities of receiving interactions compared to someone who has his account as public.

Some future improvements that could be added to our scoring method is also taking into consideration other metrics that don’t appear in our dataset, such as the number of comments or the number of visualizations. For example, a tweet is less likely to be commented than to be liked, and this could probably be added in order to compute the score of our ranking method. On the other hand, Moving onto the second part, in order to return a top-20 list of documents for each of the 5 queries using word2vec and cosine similarity, we have to do the following steps:

1. Train a Word2Vec model: using all tweets of the corpus, we train a word2vec model. That is, all vocabulary words (the set of different words among all tweets) are represented as a vector.
2. Represent each tweet as a vector: for each word of every tweet, we look if it is in the model. If it is, we append it into an array which we will, after going through all of the tweet's words, compute its mean. This average will be the tweet representation.
   In order to optimize this process for all tweets, we create a function that we call as many times as tweets we have.
3. PCA: with the purpose of reducing the complexity of the cosine similarity computation, we reduce the dimensionality of all tweets representation using PCA. After plotting the elbow graphic, we can conclude that the first three components will retain nearly 100% of the variance, so it's the number of components we will be using.

Once we have completed these steps, for each query ranking, we follow the next steps:

1. Search relevant documents: for each query, we find its relevant documents using the tf-idf search. Nonetheless, we will just store the tweet_id and its corresponding text, as we are not interested in the tf-idf. We just want to store all the relevant documents, regardless of their tf-idf score. Before storing all entries in a dictionary, the tweet text is transformed into a vector using the function "get_tweet_vector()" that we created before.
   As a sum up, the dictionary will have as keys the tweet_id and its entry will be the tweets' vector representation.
2. PCA: we apply PCA to all tweet representations and store them into a new dictionary. This second dictionary will have tweet_id as keys and the entries will be the tweets' vector representation after applying PCA.
3. Cosine similarity computation: we compute the similarity scores, which we will later sort to obtain the ranked results. Notice that we don't normalize the vectors used in the np.dot(), as they already are unit vectors since they come from PCA.

We would like to remark a couple of aspects that we think that are relevant for this section of the delivery:

- As in this part of the project we are working with conjunctive queries, some of the ones we used in the last part won't work now, as we obtained results in the second part because we didn't retrieve documents that had all query terms. As we do so in this part, original queries "Crimea bridge explosion", "Pentagon provides 18 HIMARS" and "Mykloaiv explosion bus station" have been adapted in order to obtain results.
- Since we have decided to work with very specific queries, we can't return the top-20 documents because in all five queries we retrieve less than 20 tweets. However, the code is designed to return only the top-20 if more general queries are used.
- We have decided to do dimensionality reduction because we were suggested to do so.

Finally, for the third part, we will discuss how transformer-based embeddings such as BERT or RoBERTa affect the retrieval process compared to traditional embedding methods such as word2vec, taking into account that cosine similarity between vectors is usually employed to retrieve relevant documents.

Let’s start from the beginning by the definition of a transformer. Transformers are big encoder-decoder models able to process a whole sequence with a sophisticated attention mechanism[1]. Since this type of neural network architecture works on the whole sequence, they can learn long-range dependencies. Also, some parts of the architecture can be processed in parallel, making the training much faster. They employ a new attention model which enables them to weigh the importance of different words depending on the context they are located in.

BERT it’s just a pre-trained Transformer encoder stack, and it was one of the biggest achievements in NLP as it allows, with little training effort, to fine-tune the model for our specific task[1]. This model captures bidirectional context by considering both left and right context in each training instance. Similarly, RoBERTa is an extension of BERT which optimizes the hyperparameters of BERT. Word2Vec on the other hand is a traditional embedding model that learns the representation of words based on the number of times they appear in a text. Differentially to BERT or RoBERTa, Word2Vec generates fixed-size vectors for each word independently from the context they are located in.

Therefore, when comparing both of the methods there is one main difference between transformer-based embeddings and traditional embedding: Word2Vec models generate embeddings that are context-independent, so that there is just one vector representation for each word. Different senses of the word (if any) are combined into one single vector. On the other hand, the BERT model generates embeddings that allow us to have multiple vector representations for the same word based on the context the word is used. Thus, BERT embeddings are context-dependent[2].

Since we are working tweets, there are several factors that should be considered when studying which of the two embedding methods should be used. The first one is that tweets are usually short and direct texts, written in a fast way and without entering much in detail on the structure of the text. Therefore, in tweets it is more difficult to study the context of a word in a sentence because it can happen that words do not have context at all. Hence,

[1] https://www.baeldung.com/cs/transformer-text-embeddings
[2] Difference between Word2Vec and BERT | The Startup (medium.com)

considering this first factor, it would be better to use Word2Vec. Another factor that should be considered is the computational overhead, where traditional embeddings can be less computational expensive since they are not taking into account the context of multiple senses for each word. Finally, BERT and RoBERTa can be fine-tuned for specific IR tasks, adapting the pre-trained model to the nuances of the dataset we are working with, which is the tweets dataset, improving the overall performance of the retrieving.

All in all, it directly depends on the dataset of the problem. When working with tweets, since they are usually short and without context, it would probably be better to use traditional embeddings like Word2Vec since the computational difference is not that high. Nevertheless, if the dataset is formed with longer tweets, it would be interesting to consider to switch to the transformer-based embeddings like BERT or RoBERTa since they are more capable of detecting the context for each word, but also in this case the computational difference would be higher and transformer-based methods would be key in this task.