Github: https://github.com/dani-upf/IRWA-2022-u171404-u172940-u173482

Ranking

The goal of this part is to return the top-20 documents related to a query. In the previous section we implemented the tf-idf classical scoring and evaluated its performance. Now we are going to implement 3 ranking algorithms (tf-idf+cosine similarity, our-score+cosine similarity and BM25) and check its ranking in the 5 queries defined in the previous lab.

q1: Last hurricane in Florida

q2: Terrible hurricane north carolina

q3: Medical assistance Florida

q4: Last news report Ian

q5: Dealing with east coast tragedy

1) Three ways of ranking

A) TF-IDF + cosine similarity:

For the classical TF-iDF implementation both queries and documents are represented in a V-vector space where V is the vocabulary and then we rank the cosine between query and documents to obtain the top-k documents for a query.

For representing queries/documents as vectors we use the basic tf-idf weights $w_{i,j} = (1 + log f_{i,j}) * log(N/df_i)$ where df is the document frequency, f the raw frequency and N the total number of documents in the collection (if a term had frequency 0 the result should be 0). Iterating through all the terms in the inverted index we get a final vector representation of the documents and queries.

Finally we compute the cosine similarity as $cos(q, d) = (q * d)/|q||d|$.

For all the rankings we computed the top-20 but here we will show the top-5:

Ranking for query 1 : {'doc_1655', 'doc_341', 'doc_1012', 'doc_2507', 'doc_3287'}

Ranking for query 2 : {'doc_1110', 'doc_241', 'doc_2151', 'doc_1889', 'doc_2430'}

Ranking for query 3 : {'doc_3255', 'doc_1192', 'doc_1732', 'doc_1331', 'doc_670'}

Ranking for query 4 : {'doc_2901', 'doc_1104', 'doc_2283', 'doc_3150', 'doc_2507'}

Ranking for query 5 : {'doc_253', 'doc_336', 'doc_3285', 'doc_3961', 'doc_1151'}

B) Your-Score + cosine similarity

We are going to create a new score and combine it with cosine similarity. As it is common when improving information retrieval models we are going to take the well-known TF-IDF cosine similarity model and combine it with our new defined popularity score:

$$OurScore = \lambda * Cosine\ Similarity + (1 - \lambda) * Popularity\ Score$$

λ is a tune parameter between 0 and 1, Cosine similarity is the classical tf-idf+cosine similarity score and Popularity Score is the new score Your-Score + cosine similarity, which is explained below. Overall, this new model is a combination of the previous model and the new model in which a high λ gives more importance to the classical similarity and low λ to the popularity of the tweets.

Description of Popularity Score: we were asked to implement this new score as a cosine similarity score, therefore we will also represent tweets as popularity vectors but instead of computing its similarity to queries, we are going to compute its popularity score as the length of the popularity vector. The popularity vector will be a vector of all the social features available: likes, retweets and comments (we do not consider things like followers since we want popular tweets not popular people) and normalized between 0 and 100 (dividing a value by its highest value and multiplying it by 100) because it might be, for example, more worthy having 1,000 comments than 1,000 likes.
We have decided to compute popularity score as the length of the popularity vector because for example imagine a tweet with 50 points for likes and 10 for comments: Its popularity score would be (50,10) and its score would be $\sqrt{50^2 + 10^2} = 50.44$. If likes increased from 50 to 55 its score would be 55.9 but if comments increased to 15 its new score would be 52.20. That means that for popularity we value having extremely high values for some features which may become viral (like the most liked tweet in history) rather than just well-balanced tweets with 20 points in everything.

Finally, cosine similarity has values between 0 and 1 so our popularity score will also be normalized to 1, so the range of values for our score will be between 0 and 1.

Ranking with lambda = 0 is just returning the most popular tweets while lambda=1 is the cosine similarity score. After some trial and error we think that the parameter lambda=0,75 is appropriate since from empirical evidence it gives a boost to popular tweets, but not too much to change all the ranking, only for the most popular cases.

Top 5:

$\lambda = 0$ (just most popular tweets -> always same tweets)

Ranking for query 1 : {'doc_1586', 'doc_838', 'doc_970', 'doc_3397', 'doc_3458'}

Ranking for query 2 : {'doc_1586', 'doc_838', 'doc_970', 'doc_3397', 'doc_3458'}

Ranking for query 3 : {'doc_1586', 'doc_838', 'doc_970', 'doc_3397', 'doc_3458'}

Ranking for query 4 : {'doc_1586', 'doc_838', 'doc_970', 'doc_3397', 'doc_3458'}

Ranking for query 5 : {'doc_1586', 'doc_838', 'doc_970', 'doc_3397', 'doc_3458'}

$\lambda = 0.75$

Ranking for query 1 : {'doc_838', 'doc_970', 'doc_1012', 'doc_2507', 'doc_3287'}

Ranking for query 2 : {'doc_1110', 'doc_838', 'doc_970', 'doc_1889', 'doc_2430'}

Ranking for query 3 : {'doc_3255', 'doc_1192', 'doc_1732', 'doc_838', 'doc_1331'}

Ranking for query 4 : {'doc_2901', 'doc_838', 'doc_2283', 'doc_970', 'doc_3397'}

Ranking for query 5 : {'doc_253', 'doc_3285', 'doc_838', 'doc_970', 'doc_1151'}

Note that here some of the documents coincide with the original tdf-id, but others have been replaced by more popular tweets.

C) BM 25

Finally we have implemented the BM25 ranking algorithm. We are using short queries, therefore we will use the basic BM25 version: recall that for each document we computed the retrieved statues value (RSV) of a query.

$$RSV_d = \sum_{t \in q} [log(\frac{N}{df_t})] * \frac{(k_1+1)tf_{td}}{k_1((1-b)+b*(\frac{L_d}{L_{ave}}))+tf_{td}}$$

$k1$ and b are tuning parameters for controlling the importance of the term frequency and document length. We will use k1=1.5 and b=0.75 which are known to work well.

Top 5:

Ranking for query1:['doc_3287', 'doc_1012', 'doc_3954', 'doc_3088', 'doc_3884']

Ranking for query2:['doc_1110', 'doc_3253', 'doc_3418', 'doc_2151', 'doc_3898']

Ranking for query3:['doc_1331', 'doc_670', 'doc_1192', 'doc_1732', 'doc_1466']

Ranking for query4:['doc_2283', 'doc_725', 'doc_3150', 'doc_2368', 'doc_3057']

Ranking for query5:['doc_253', 'doc_1151', 'doc_336', 'doc_3285', 'doc_2182']

BM25 vs Tf-idf

The main difference between the tf-idf cosine similarity and BM25 is that while the former is motivated by cosine similarity in a vector space, the latter is motivated by probability theory. Vector space does not directly translate the notion of “is the document good for the user” since the most similar documents can be very relevant or non-relevant. Probability approach is a cleaner formalization of what we want an IR system to do: return relevant models to the user

Vector space model:

- Advantages:
    - term-weighting improves quality of the answer set
    - partial matching allows retrieval of docs that approximate the query conditions
    - cosine ranking formula sorts documents according to a degree of similarity to the query
    - document length normalization is naturally built-in into the ranking
- Disadvantages:
    - It assumes independence of index terms

Probabilistic ranking approach (BM25):

- Advantages:
    - Efficient
    - Performs well in a wide variety of tasks
- Disadvantages:
    - Many assumptions and heuristics that make it difficult to improve the framework.

In summary, both models present different ways of answering the same question, with its positive and negative points. It depends on the application but in general BM25 has shown to be more robust.

2) Word2vec + cosine similarity

For the last part of Part 3 we implemented the Word2Vec embedding + cosine similarity ranking. First, what is the Word2vec embedding? Word embeddings are vector representations of a particular word, and as in the vector model they are useful to compute the cosine similarity between documents and queries. However, when you represent documents as vector you are assuming that for example a if a document is {“Good”}, d2 is {“Nice”} and d3 is {“Horrible”} the vocabulary would be {“Good”, “Nice”, “Horrible”} and the documents would be 3d vectors with its tf-idf weights. The similarity between d1 and d2 is the same than between d2 and d3, which obviously should not be the same: Good is much more similar to Nice than Horrible.

Here is where the idea of distributed representations enters: introduce the dependence between words. Using Neural networks Word2Vec embedding allows to implement this approach. There are two main architectures used: CBOW which tends to better represent frequent words and be a bit faster, and Skip Grams, suited better for small amounts of data and rare words.

With this intuitive idea we used the python library gensim, imported Word2Vec and common_texts, and computed the top-20 cosine similarity ranking using this Word2vec embedding representation.

We first gather the documents ids of the query if they are in the inverted_index dictionary. Then, if they are also in the tweets_dictionary, they are introduced as input to our model. Finally, we will compute the average of the words in the vector and the cosine similarity (the final rankings can be found at the end of the notebook).

3) A better representation than word2vec

Following the same idea of putting close words that are similar, we could expand it and try to put close similar sentences (sentence2vec). For example the sentence "The early bird catches the worm" (an english saying) is very close to "It is good to wake up early" because although each word is marginally different, when you group them together they have the same meaning. The same could be said of Doc2Vec, where the algorithm will learn feature representation of documents, which can help for comparing documents, among others.

It is a good idea to imagine it as taking the average of the meaning, so for whereas Word2Vec is good for capturing the meaning between words, Sent2Vec is would be much better in capturing the relation between sentences of the text and Doc2Vec for the general tags/features of a document. Therefore, the advantages and disadvantages of each embedding depend on the context of the problem.

.