Universitat Pompeu Fabra Barcelona

IRWA FINAL PROJECT

PART 3

RANKING

Jordi Guillén: u198641
Anira Besora: u189647
Ana Cereto: u199767
Marina Castellano: u188311

INDEX

Github/ Tag: https://github.com/Marinagrup2/IRWA.git

1. Scores

1.1. TF-IDF + cosine similarity

For this ranking we used the previously defined functions in part 1 `search_tf_idf` and `rank_documents` to retrieve the ranking of the top 20 tweets from our queries.

The ranking of the documents for the first query is as follows:

Query	Ranking	
Farmers protest	1. doc_25479	2. doc_445
	3. doc_40148	4. doc_42449
	5. doc_20026	6. doc_23201
	7. doc_46574	8. doc_41343
	9. doc_22193	10. doc_16840
	11. doc_39710	12. doc_22377
	13. doc_27696	14. doc_32542
	15. doc_21124	16. doc_17150
	17. doc_16847	18. doc_19903
	19. doc_12181	20. doc_17097

TF-IDF is a technique where we score a document for a query by calculating the product of the term frequency (TF) of a term in a document and the inverse document frequency (IDF), the inverse of the frequency of a term across all documents in the collection, giving less weight to common terms, so taking into account the rarity of the term.

Some of the pros of this technique are:

- *Simple and Efficient:* TF-IDF is straightforward and relatively fast to compute.
- *Intuitive:* It's easy to interpret, with an easy score (higher→ more relevant) as it directly emphasizes high term frequency and downplays common words via IDF, showing the relevance of documents by rare terms that might occur.

The cons are:

- *Document Length:* Without proper normalization, TF-IDF can favor longer documents, as they often have more occurrences of query terms.
- *Context:* The frequency of terms in the documents or the whole collection don't take into consideration the whole context of the document, and the ranking could contain not relevant documents.

1.2. Our_score + cosine similarity

In this ranking, we compute the document score using the number of likes and retweets each document has, and combine it with the score obtained in the cosine similarity.
For this, we have created 3 functions:

- `cosine_similarity(v1, v2):` This function computes the cosine similarity between two vectors.
    - The formula is: $cosine\_similarity(v1, v2) = \frac{v1 * v2}{|v1| \, x \, |v2|}$
- `rank_documents_your_score(terms, docs, index, idf, tf, title_index):` This function computes the score obtained by the cosine similarity, the number of likes, and the number of retweets. Then combines them to compute a final score.
    - Cosine similarity score: we prepare the docs and query as vectors and compute the cosine similarity in the function mentioned before.
    - Like score: We compute the log of the number of likes of the document. We add one in case the 'likeCount' is 0.
    - Retweet score: We compute the log of the number of retweets of the document. We add one in case the 'retweetCount' is 0.
    - Final score: we combine the 3 scores of course giving more weight to the cosine similarity because the other two scores are a bit trivial.

- `search_your_score(query, index, title_index):` we iterate over each query and rank the documents based on the score computed in the function rank_documents_your_score. Then we print the top 20 with the highest score.

This ranking method prioritizes the similarity between the query and document content, placing a higher emphasis on this parameter, while also incorporating tweet popularity within each document. Popularity, in this context, is determined by the number of likes and retweets.

However, this approach may not always yield the most accurate results. Sometimes, documents with lower popularity may actually align more closely with the query, yet they rank lower due to their popularity scores. As a result, valuable but less popular tweets might be overlooked.

On the other hand, if your goal is to find the most popular content related to your query, this ranking strategy can be highly effective. It better captures trending discussions around a topic, even if they aren't the closest matches to your query.

The ranking of the documents for the first query is as follows:

Query	Ranking	
Farmers protest	1. doc_46206	11. doc_38379
	2. doc_23286	12. doc_41653
	3. doc_45142	13. doc_9995
	4. doc_38012	14. doc_18306
	5. doc_13630	15. doc_42516
	6. doc_44034	16. doc_31312
	7. doc_38262	17. doc_36828
	8. doc_20292	18. doc_27804
	9. doc_26876	19. doc_11883
	10. doc_26064	20. doc_37099

1.3. BM25

To calculate the BM25 ranking, we first created a function called `calculate_doc_lengths_and_avg`, which returns a dictionary containing the length of each document (Ld) and the average document length (Lave). In the `rank_documents_bm25` function, we iterate over each term in the query. For each term, we find the documents that contain it in the index, retrieve the term frequency within each document, and obtain the document length. Using this information, we apply the BM25 formula with parameters k1 = 1.5 and b = 0.75. We then multiply the result by the term's IDF and add it to the document's score. This process is repeated for all terms in the query, cumulatively summing the BM25 scores for each document.

The ranking of the documents for the first query is as follows:

Query	Ranking	
Farmers protest	1. doc_27696	2. doc_19226
	3. doc_32847	4. doc_39691
	5. doc_21124	6. doc_2426
	7. doc_30684	8. doc_36652
	9. doc_38587	10. doc_41627
	11. doc_44691	12. doc_44712
	13. doc_44982	14. doc_884
	15. doc_5548	16. doc_5552
	17. doc_9763	18. doc_11632
	19. doc_16380	20. doc_23124

BM25 is a technique that uses the IDF of the term but introduces two parameters, k1 to control the term frequency, so that high term frequency terms don’t increase the score too much, and b for document length regulation, so there is no bias toward longer docs.

Some of the pros of this technique are:

- *Frequency regulation:* With the **k1** parameter, BM25 prevents high term frequencies from overly boosting scores, making it more reliable for longer documents or documents with repetitive terms.
- *Length Normalization:* The **b** parameter lets BM25 factor in document length, which makes it more effective for datasets with varying document lengths (like short posts versus long articles).

The cons are:

- *Complexity:* BM25's parameter tuning and more complex formula make it harder to implement and understand than TF-IDF.
- *Parameter Sensitivity*: The choice of **k1** and **b** values significantly affects performance, and finding optimal values requires tuning for each collection.

1.4. Comparing TF-IDF to BM25

The main difference between the models lies in the parameters used to regulate the term frequency and the document length in BM25, whereas the TF-IDF does not take into consideration these factors.

In terms of the rankings, we can see that the main difference is that the content of the tweets in BM25 is shorter, whereas the tweets in TF-IDF were longer. This suggests that TF-IDF may give preference to tweets with greater length due to the lack of document length normalization.

However, relying solely on term frequency and IDF to rank tweets may not be the best approach, as multiple factors influence the relevance of tweets for users. For example, elements such as hashtags (which we emphasized by tripling their weight during preprocessing), as well as likes and retweets, contribute to relevance. In fact, if we were to create our own scoring method —which we will explore next as an additional step— these factors would be valuable to include.

2. Word2vec + cosine similarity

To calculate the Word2Vec + cosine similarity ranking, we first create embeddings for the tweets in our documents. For each tweet and query, we position them in the n-dimensional space and compute the cosine similarity to determine how close, and thus how similar, they are.

Next, we implement the following functions:

- `cosine_similarity(v1, v2):` This function computes the cosine similarity between two vectors.

  The formula is: $cosine\_similarity(v1, v2) = \frac{v1 * v2}{|v1| \, x \, |v2|}$

- `model_word2vec(docs):` This function Generates the model for word2vec.
  - We extract the words from the tweets of the documents and create an embedding space using the Word2Vec function
  - Here are some details about the Word2Vec function:
    - sentences = words: This specifies the data used to train the model. In our case, the words from the tweets.
    - workers = 4: This tells the model to use 4 parallel threads to train, which speeds up processing but can introduce non-deterministic results (different runs may give slightly different embeddings).
    - window = 3: This defines the maximum distance between the target word and surrounding words. A window of 3 means the model considers up to 3 words before and after each target word.
    - min_count = 200: This specifies the minimum frequency of words to include in the model. Any word that appears fewer than 200 times in the words list is ignored. This helps to filter out rare words that would not have a good or big impact.
  - `tweet2vec(tweet,model):` This function generates a vector representation of a tweet by averaging the vectors of the words in the tweet.
    - Tokenization: The tweet text is split into individual words.

- Vector Lookup: For each word, it tries to get the word vector from model.
- Handling missing words: If a word is not found in the model's vocabulary, it appends a zero vector of the model's vector size.
- Averaging: If there are vectors, it calculates the mean across these vectors to get a single representation vector for the tweet. If the tweet is empty, it returns a zero vector of the model's size.

- `search_word2vec(query, title_index, top_k=20):` This function ranks documents based on their similarity to a query.
  - Generate the model for embedding: model_word2vec is called to generate the embedding for the vectors with the documents.
  - Convert Query to Vector: tweet2vec is used to generate a vector representation of the query.
  - Document Vector Calculation: For each document in title_index, tweet2vec generates a vector based on the document's content.
  - Cosine Similarity Calculation: The cosine similarity score is computed between query_vector and each doc_vector.
  - Handling missing vectors: Handles cases where either vector is zero (returns a score of 0).
  - Sorting and Top-20 Results: Documents are sorted by similarity score, highest to lowest. It returns the top k document IDs based on their similarity scores.

The ranking of the documents for the first query in one of the executions is as follows:

Query	Ranking	
	1. doc_4681	11. doc_45714
	2. doc_7438	12. doc_14703
	3. doc_16557	13. doc_38887
	4. doc_20877	14. doc_41627

Farmers protest	5. doc_21436	15. doc_21599
	6. doc_21450	16. doc_21562
	7. doc_30415	17. doc_14054
	8. doc_42839	18. doc_19779
	9. doc_22147	19. doc_2418
	10. doc_5727	20. doc_13937

In this approach, we transform both the query and each document (tweet) into vector representations within an embedding space, where each dimension in this space captures semantic relationships between words based on a pre-trained Word2Vec model. This embedding space allows us to position words, phrases, and even whole sentences as points, where similar meanings tend to be closer when we compute the cosine angles between them.

3. Can you imagine a better representation than word2vec? Justify your answer.

For document or tweet similarity tasks, Doc2Vec and Sentence2Vec are generally better representations than Word2Vec, as they are designed to capture broader contextual information at the sentence or document level.

Word2Vec generates word-level embeddings by learning word associations based on their co-occurrence in contexts. While effective for capturing word relationships, it often falls short for document or sentence similarity tasks because it does not directly capture information at the sentence or document level.

Sentence2Vec generates embeddings specifically optimized for sentences or short texts by leveraging transformer models like BERT or Sentence-BERT. This technique optimizes embeddings to directly reflect semantic similarity, making it highly effective for applications like sentence similarity and clustering.
Sentence2Vec uses a transformer model that generates embeddings for sentences or short texts by directly optimizing for semantic similarity tasks.

Doc2Vec extends Word2Vec to create fixed-length embeddings for entire documents. In addition to learning word vectors, it learns unique document vectors that capture the overall context of each document, making it suitable for document-level tasks.

Sentence2Vec:

For **Sentence2Vec**, the pros are the following:

- **High Semantic Accuracy**: It captures both syntax and semantic relationships, making it highly effective for short texts like tweets or sentences that we can find in our collection.
- **Better Performance for Similarity Tasks**: Sentence Transformers are optimized for tasks like similarity scoring and clustering, yielding higher accuracy and relevance in retrieval tasks compared to Word2Vec or Doc2Vec.

But it also has some cons such as:

- **More Computationally Intensive**: It is generally slower and more resource-heavy than Word2Vec and Doc2Vec, which could be a limitation for very large datasets such as our collection of tweets.
- **Limited Interpretability**: Sentence Transformers are complex, and the embeddings they produce are harder to interpret compared to simpler models like Word2Vec.

  On the other hand, Doc2Vec is an extension of Word2Vec to learn fixed-length vector representations for entire documents. It incorporates a unique document vector for each document alongside the word vectors in the model training process so it aims to capture semantic relationships at the document level, providing a unique representation for each document rather than simply averaging word vectors.

Doc2Vec:

The pros for **Doc2vec** are the following:

- **Captures Document-Level Context**: Unlike Word2Vec, which captures word-level semantics, Doc2Vec learns representations that incorporate the context of the entire document, preserving some structure and content uniqueness, which is more useful in our context of tweets as they are not very extensive and the context could determine the relevance for the user.
- **Better for Short Texts**: Since Doc2Vec explicitly learns document-level embeddings, it is more effective than averaging word embeddings, especially for short texts like tweets as we said, where individual words may not fully represent the context.

But it has its drawbacks:

- **Computationally Intensive to Train**: Doc2Vec is more complex and resource-intensive to train than Word2Vec, as it requires a unique vector for each document in the collection.
- **Limited Transferability**: Doc2Vec is generally trained on a specific dataset (dependent on training data), so the model may not perform well on data very different from the training data, such as tweets versus longer articles.

In the following table we can see a comparison summary on different features of each technique:

Feature	Word2Vec	Sentence2Vec	Doc2Vec
Granularity	Word-level	Sentence-level	Document-level
Training	Requires large text corpus	Sentence embeddings, typically with deep learning models	Similar to Word2Vec, with document context
Usage	Word similarity, word analogies	Sentence similarity, classification	Document similarity, classification
Input	Single words	Sentences	Documents
Training Speed	Fast	Moderate	Moderate to Slow
Dimensionality	Dependent on training data	Depends on sentence length	Dependent on training data
Memory Usage	Lower	Moderate	Higher

Image 1: summarized comparison of Word2Vec, Sentence2Vec, and Doc2Vec across various key aspects[1]

For our collection of tweets, **Sentence2Vec** would likely be the best representation, as it's designed to generate high-quality embeddings for short texts, preserving both semantic meaning and contextual information. This would provide the most accurate results for tweet similarity without the limitations of averaging word embeddings, as in Word2Vec, and it is more suitable than Doc2Vec if we prioritize semantic accuracy over computational efficiency.

[1] GeeksforGeeks. (2024, February 16). Comparing Word2Vec, Sentence2Vec, and Doc2Vec: A Comprehensive Analysis. GeeksforGeeks. https://www.geeksforgeeks.org/comparing-word2vec-sentence2vec-and-doc2vec-a-comprehensive-analysis/

4. Webgrafia

- https://towardsdatascience.com/word2vec-explained-49c52b4ccb71
- https://towardsdatascience.com/multi-class-text-classification-with-doc2vec-logistic-regression-9da9947b43f4
- https://towardsdatascience.com/text-embeddings-comprehensive-guide-afd97fce8fb5