Ingrid Sancho Aventin - 218033
Vladimir Trukhaev - 219328
Information Retrieval & Web Analysis

Universitat Pompeu Fabra Barcelona

Final Project Part 2: Indexing and Evaluation

Introduction

This part of the final project consisted in indexing and evaluating our corpora from the World Health Organization Twitter account (*dataset_tweets_WHO.txt*). This had to be done in order to implement the TF-IDF (Term Frequency — Inverse Document Frequency) ranking and later on evaluating how it did using different evaluation methods. Said methods included Precision @ K, Average Precision @ K, Mean Average Precision, Mean Reciprocal Rank and Normalized Discounted Cumulative Gain.
As a submission we have this PDF file (*IRWA-2021-final-project-U162442-U161809-part-2*) and the following GitHub repository where you can find the code we have used to accomplish all the tasks required for the submission: https://github.com/vladimirtrukhaev/IRWA_Project. In said GitHub repository you can find a branch called *"IRWA-2021-final-project-part-2"* (https://github.com/vladimirtrukhaev/IRWA_Project/tree/IRWA-2021-final-project-part-2) where the code for this part of the project, this PDF file as well as the provided text file are located.

Indexing

1.1 Index Implementation

In this first implementation of the index we will create a simple inverted index where the terms will be stored as keys. For values we will have the ids of tweets and the position of such terms in the doc. All this information (for all terms) will be stored in a dictionary called "index". The implementation of such an index will be done by calling the function create_index, defined as shown in *Figure 1*.

```
def create_index(my_dict):
    """
    Implement the inverted index

    Argument:
    my_dict -- dictionary with word lists of tweets as values and tweet id as key

    Returns:
    index - the inverted index (implemented through a Python dictionary) containing terms as keys and the corresponding
    list of documents where these keys appears in (and the positions) as values.
    """
    index = defaultdict(list)
    title_index = {}
    for doc in my_dict:
        current_tweet_index = {} # create the index for the current tweet and store it in current_tweet_index
        for position, term in enumerate(my_dict[doc]): # Loop over all terms
            try:
                  # if the term is already in the index for the current tweet (current_tweet_index)
                  # append the position to the corresponding list
                   current_tweet_index[term][1].append(position)
            except:
                  # add the new term as dict key and initialize the array of positions and add the position
                    current_tweet_index[term] = [doc, array('I',[position])] #'I' indicates unsigned int (int in Python

          # merge the current tweet index with the main index
        for tweet_term, posting_tweet in current_tweet_index.items():
            index[tweet_term].append(posting_tweet)

    return index
```

Fig. 1. Function to Create a Simple Inverted Index.

We will successfully create the index by running the code below.

```
In [7]: start_time = time.time()
        index = create_index(my_dict)
        print("Total time to create the index: {} seconds".format(np.round(time.time() - start_time, 2 )))

        Total time to create the index: 0.12 seconds
```

Fig. 2. Creating a Simple Inverted Index.

1.2 Querying the Index

Next step is to implement the remaining functions in order to be able to search for the query terms in the previously created "index". The three functions that will help us achieve this will be explained below.

A. Build terms function

This function will be called in order to "text process" the input query. In this way the query will match the index terms (since we already preprocessed all tweets in the previous part of the practice). This function will be as following:

```
def build_terms(query):
    """
    Preprocess the input query removing stop words, stemming,
    transforming in lowercase and return the tokens of the text.

    Argument:
    query -- string (text) to be preprocessed

    Returns:
    query - a list of tokens corresponding to the input text after the preprocessing
    """

    stemmer = PorterStemmer()
    stop_words = set(stopwords.words("english"))

    query = query.lower() ## Transform in lowercase
    query = query.split() ## Tokenize the text to get a list of terms
    query = [word for word in query if not word in stop_words]  ## Eliminate the stopwords
    query = [stemmer.stem(word) for word in query] ## Perform stemming

    return query
```

Fig. 3. Text Processing with build_terms function.

B. Search function

In order to find which documents (in our case tweets) contain a given query we implemented a search function. This function will return a list of tweets where all the terms of the query appear. This means it will give a list of potential answers to our query search.

```python
def search(query, index):
    """
    The output is the list of documents that contain any of the query terms.
    So, we will get the list of documents for each query term, and take the union of them.

    Argument:
    query -- string (text) to split and search by
    index -- inverted index (in a form of a dictionary)

    Returns:
    docs - a list of ids
    """
    query = build_terms(query)
    docs = set()
    for term in query:
        try:
            # store in term_docs the ids of the docs that contain "term"
            term_docs=[posting[0] for posting in index[term]]
            # docs = docs Union term_docs
            docs = docs.union(term_docs)
        except:
            #term is not in index
            pass
    docs = list(docs)
    return docs
```

Fig. 4. Search function.

C. Creating dictionary with all information to be retrieved for each result

Whenever a doc (tweet) is retrieved we are asked to give certain information: Tweet, Username, Date, Hashtags, Likes, Retweets and Url. In order to do this we went back to the original dataset and created a new dictionary that contained such information as values and the tweets id as keys so that we could trace back the information of any doc. Such a dictionary will be called docs_info, and will be created as shown in *Figure 5*.

```python
#defining docs_info from original data
doc = "dataset_tweets_WHO.txt"
with open(doc, 'r') as file:
    data = json.load(file)

#initializing dictionary "docs_info" where value are all the information text and key is the tweet's id
keylist = []
for key in data:
    keylist.append(key)

docs_info = {}

for i in keylist:
    docs_info[i] = None

for key in data:
    tweet = data[key]["full_text"]
    username = data[key]["user"]["name"]
    date = data[key]["created_at"]
    hashtags = data[key]["entities"]["hashtags"]
    likes = data[key]["favorite_count"]
    retweets = data[key]["retweet_count"]
    try:
        url = data[key]["entities"]["media"][0]["expanded_url"]
    except: #sometimes we weren't able to find the url in the data, then:
        url = "https://twitter.com/WHO/status/%s" % (data[key]["id_str"])

    info = {"tweet": tweet, "username": username,
            "date": date, "hashtags": hashtags,
            "likes": likes,"retweets": retweets, "url": url}
    docs_info[key] = info
```

Fig. 5. Creating the dictionary.

1.3 Add ranking with TF-IDF

After querying our index we have to add a TF-IDF ranking. This mechanism is used to obtain the results sorted by relevance (if you notice in the previous steps we did not rank the results).
TF-IDF basically works by assigning each term in the document (in this case it is a list of Tweets) a weight based on its term frequency and the inverse document frequency. Meaning the higher the score, the more relevant a Tweet is to our search.
To add said ranking we will have to go through three different steps once again. These steps will be explained more in detail below.

A. Create index TF-IDF function

In this first step of implementing our ranking using TF-IDF we will have to create an inverted index and compute the TF (Term Frequency), DF (Document Frequency) and the IDF (Inverse Document Frequency).
We will take as input the Tweets we have as well as the total number of Tweets that appear in the provided text file. Using the formulas we have seen in Theory we will then compute the inverted index, TF, DF and IDF that will be returned as dictionaries at the end of our function.
This part of the implementation can be seen below:

```
def create_index_tfidf(my_dict, num_documents):
    """
    Implement the inverted index and compute tf, df and idf

    Argument:
    my_dict -- collection of Wikipedia articles
    num_documents -- total number of documents

    Returns:
    index - the inverted index (implemented through a Pyhon dictionary) containing terms as keys and the corresponding
    list of document these keys appears in (and the positions) as values.
    tf - normalized term frequency for each term in each document
    df - number of documents each term appear in
    idf - inverse document frequency of each term
    """

    index = defaultdict(list)
    tf = defaultdict(list)  #term frequencies of terms in documents (documents in the same order as in the main index)
    df = defaultdict(int)  #document frequencies of terms in the corpus
    idf = defaultdict(float)
    for doc in my_dict:
        current_tweet_index = {}
        for position, term in enumerate(my_dict[doc]): # terms contains page_title + page_text. Loop over all terms
            try:
                  # if the term is already in the index for the current page (current_tweet_index)
                  # append the position to the corresponding list
                   current_tweet_index[term][1].append(position)
            except:
                  # Add the new term as dict key and initialize the array of positions and add the position
                    current_tweet_index[term]=[doc, array('I',[position])] #'I' indicates unsigned int (int in Python)

        #normalize term frequencies
        # Compute the denominator to normalize term frequencies (formula 2 above)
        # norm is the same for all terms of a document.
        norm = 0
        for term, posting in current_tweet_index.items():
            # posting will contain the list of positions for current term in current document.
            # posting ==> [current_doc, [list of positions]]
            # you can use it to infer the frequency of current term.
            norm += len(posting) ** 2
        norm = math.sqrt(norm)

        #calculate the tf(dividing the term frequency by the above computed norm) and df weights
        for term, posting in current_tweet_index.items():
            # append the tf for current term (tf = term frequency in current doc/norm)
            tf[term].append(np.round(len(posting)/norm,4)) ## SEE formula (1) above
            #increment the document frequency of current term (number of documents containing the current term)
            df[term] = tf[term] # increment DF for current term

        #merge the current page index with the main index
        for term_page, posting_page in current_tweet_index.items():
            index[term_page].append(posting_page)

        # Compute IDF following the formula (3) above. HINT: use np.log
        for term in df:
            idf[term] = np.round(np.log(float(num_documents/len(df[term]))), 4)

    return index, tf, df, idf
```

Fig. 6. Defining the Index TF-IDF function.

By running the following code, we will successfully implement the indexing + TF-IDF values of our dictionary of tweets.

```
In [12]: start_time = time.time()
         num_documents = len(my_dict)
         index, tf, df, idf = create_index_tfidf(my_dict, num_documents)
         print("Total time to create the index: {} seconds" .format(np.round(time.time() - start_time, 2)))

         Total time to create the index: 91.4 seconds
```

Fig. 7. Generating the index with the TF-IDF values.

B. Rank documents function

After having our Index TF-IDF function implemented we will go on to do the same for the rank document function. This function will perform the ranking of the results of any search we will make using the TF-IDF weights we implemented in *Part A*.

It will take as input the term we want to search for, the Tweets provided in the text file, the inverted index, the title index and finally both the TF and IDF.

```
def rank_documents(terms, docs, index, idf, tf):
    """
    Perform the ranking of the results of a search based on the tf-idf weights

    Argument:
    terms -- list of query terms
    docs -- list of documents, to rank, matching the query
    index -- inverted index data structure
    idf -- inverted document frequencies
    tf -- term frequencies

    Returns:
    Print the list of ranked documents
    """

    # I'm interested only on the element of the docVector corresponding to the query terms
    # The remaining elements would became 0 when multiplied to the query_vector
    doc_vectors = defaultdict(lambda: [0] * len(terms)) # I call doc_vectors[k] for a nonexistent key k, the
    query_vector = [0] * len(terms)

    # compute the norm for the query tf
    query_terms_count = collections.Counter(terms)  # get the frequency of each term in the query.
    # Example: collections.Counter(["hello","hello","world"]) --> Counter({'hello': 2, 'world': 1})
    #HINT: use when computing tf for query_vector

    query_norm = la.norm(list(query_terms_count.values()))

    for termIndex, term in enumerate(terms):  #termIndex is the index of the term in the query
        if term not in index:
            continue

        ## Compute tf*idf(normalize TF as done with documents)
        query_vector[termIndex] =  query_terms_count[term]/len(terms)*idf[term]

        # Generate doc_vectors for matching docs
        for doc_index, (doc, postings) in enumerate(index[term]):
            # Example of [doc_index, (doc, postings)]
            # 0 (26, array('I', [1, 4, 12, 15, 22, 28, 32, 43, 51, 68, 333, 337]))
            # 1 (33, array('I', [26, 33, 57, 71, 87, 104, 109]))
            # term is in doc 26 in positions 1,4, .....
            # term is in doc 33 in positions 26,33, .....

            #tf[term][0] will contain the tf of the term "term" in the doc 26
            if doc in docs:
                doc_vectors[doc][termIndex] = tf[term][doc_index] * idf[term]  # TODO: check if multiply fo

    # Calculate the score of each doc
    # compute the cosine similarity between queyVector and each docVector:
    # HINT: you can use the dot product because in case of normalized vectors it corresponds to the cosine
    # see np.dot

    doc_scores=[[np.dot(curDocVec, query_vector), doc] for doc, curDocVec in doc_vectors.items() ]
    doc_scores.sort(reverse=True)
    result_docs = [x[1] for x in doc_scores]
    #print document titles instead if document id's
    #result_docs=[ title_index[x] for x in result_docs ]
    if len(result_docs) == 0:
        print("No results found, try again")
        query = input()
        docs = search_tf_idf(query, index)
    #print ('\n'.join(result_docs), '\n')
    return result_docs
```

Fig. 8. Function to rank documents according to the TF-IDF.

C. Search function (search_tf_idf)

Finally we will have to implement the search function as we did before but in this case it will incorporate the TF-IDF ranking. It will take the query of the user as the first input as well as the index we previously created as the second one. It will then return the user the Top 10 most relevant results (following the TF-IDF ranking) out of all the Tweets in our collection.

```python
def search_tf_idf(query, index):
    """
    output is the list of documents that contain any of the query terms.
    So, we will get the list of documents for each query term, and take the union of them.
    """
    query = build_terms(query)
    docs = set()
    for term in query:
        try:
            # store in term_docs the ids of the docs that contain "term"
            term_docs=[posting[0] for posting in index[term]]

            # docs = docs Union term_docs
            docs = docs.union(term_docs)
        except:
            #term is not in index
            pass
    docs = list(docs)
    ranked_docs = rank_documents(query, docs, index, idf, tf)
    return ranked_docs
```

Fig. 9. Search function.

Selected Queries

For this part of the final project we have selected the following five queries:

1. "Covid en España"
2. "how many covid cases"
3. "mortalidad covid"
4. "Covid prevention"
5. "Pandemia mundial"

After running our TF-IDF ranking function we got the results that can be seen below.

Fig. 10, 11, 12, 13 and 14. Results of running the aforementioned queries.

2. EVALUATION

In this part of the deliverable we will have to evaluate our ranking function using different methods that we have seen in class (P@K, AP@K, MAP, MRR and NDCG). We will explain how each one of the methods works as well as the differences between them.

2.1 Creating dataframe with ground truth

We started by creating a dataframe of 5 times 2399 entries and 4 different columns: “q_id”, “doc_id”, “predicted_relevance”, “bin_y_truth”. The first step was entering the correct “q_id” and “doc_id” so that every pair of (query, doc) had their own values. We then based our “predicted_relevance” on the previously computed doc_scores (in tf-idf function). This implementation was challenging and since we considered other ways to implement it (for example, we considered trying to compute also negative predicted relevance) it took more time than expected. In order to set the ground truth (“bin_y_truth”) we discarded the idea of going tweet by tweet with one of the given queries and analyzing whether a tweet was relevant or not. We did not know how to implement this variable since in the practice this was given in the input data. The only thing we came up with was to base it also on the tf-idf algorithm, only that this time for the top half of the retrieved tweets we would have 1 (relevant) and 0 otherwise. We are well aware this is absolutely not the optimal implementation but we had no other way to approach the “expert judges” roll, we based it on the algorithm we are in fact trying to evaluate, therefore it will be intrinsically biased and will not give accurate feedback on the search engine.

2.2 Evaluation Techniques

We struggled to implement the evaluation algorithms since it was hard for us to evaluate over a “ground truth” that we had implemented taking into account our own algorithm. The following evaluation techniques are successfully understood but we were not able to successfully implement them for our specific case. Having to work with Series arrised many errors during the runnings and it became very challenging to rebuild and correct the code at that point. Having this in consideration we still wanted to comment on all of the evaluation techniques:

2.2.1 Precision@K (P@K)

The first method we have used is called P@K (Precision @ K). P@K is a method that measures the number of relevant results among the top K documents. It is used to see if the users are getting relevant results at the top of their ranking.
This is why one of its drawbacks is that it does not take into account the positions of the relevant documents among the top K. Below you can see the

implementation in code of this method as well as the results retrieved when using it.
Unfortunately when writing the code for this part we got a P@K (in this case K was 10) of 0. This should not be possible as all the retrieved Tweets were assigned with a 1 in the "bin_y_true" column. Meaning that the P@K should always be 1 rather than 0 how we got in the implementation of our code.

2.2.2 Average Precision@K (AP@K)

Compared with P@K, AP@K is a better method for evaluating our ranking function. This is due to the fact that AP@K gives a better intuition of the model ability of sorting the results for a specific query. AP@K tells us how much the relevant documents are concentrated in the highest ranked predictions. Making the drawback that was present when using P@K non-existing.
In this case we encountered the same problem as when coding P@K, as the AP@K (K being 10 once again) was 0. This should not be possible as like we have said before, all the retrieved documents (Tweets in this case) were tagged with a 1. Making all of them relevant.

2.2.3 Mean Average Precision (mAP)

Computing the mAP is basically computing the mean of all the AP queries. The difference between mAP and the previous two methods/metrics is that mAP takes into account all the queries and not just one (as P@K and AP@K do).

2.2.4 Mean Reciprocal Rank (MRR)

In order to compute the MRR we first have to compute the Reciprocal Rank or RR score of one query. This is done by dividing 1 over K, where K is the rank position of the first relevant document. The MRR is then determined by the mean of the RR across different queries.
In this implementation we got different errors (that did not make sense to us) which limited our ability to get to an accurate solution, we could not get around those errors.

2.2.5 Normalized Discounted Cumulative Gain (NDCG)

To understand NDCG we have to look at DCG first. DCG penalizes highly relevant documents (in this case Tweets) that appear lower in the search results. This works because DCG uses a graded relevance value that is reduced logarithmically proportional to the position of said result.
When calculating the DCG for different queries a problem is encountered: some queries are harder than others meaning that the DCG score will be lower for

said queries. This is where normalization comes into play. Normalization solves this problem by scaling the results based on the best result seen. This is achieved by sorting all relevant documents in the corpus (the World Health Organization Twitter account in this case) by their relative relevance, producing the maximum possible DCG through position N. By doing so we will have created the Ideal DCG values which we will then use to obtain the NDCG. The NDCG is then calculated by dividing the Actual DCG values by the Ideal ones.

In our particular case, we do not have an ideal DCG since our ground truth, as explained before, is set by us using a binary implementation. We were confused about how to implement such “ground truth” since the dataset is massive for us to be analysing one by one and we have no way to know for sure the actual answer. For this reason, this evaluation technique cannot be run for our engine. Nevertheless, as it was stated in the *Part 2* of the exercise, we tried to implement the function for a general case.

2.3 Vector representation

We chose word2vec in order to represent the tweets in a two-dimensional scatter plot through the t-sne algorithm. To do so, we had to represent both the word and the tweet as vectors. We found this task challenging and we still doubt we implemented it correctly. We needed to first investigate and do research about the libraries that were used in the implementation of vector representation. We first approached this by looking for the most similar terms given a query term. For example we searched for “covid” and we got that “19”, “paraguay”, and “vacunado” were the top 3 most similar words, which means that they often appear together. We were able to plot the top 30 terms on a 2D plot. After this we proceed with the 3D scatter plot. We implemented it but unfortunately we only succeeded in the making of a simple one, similar to the previously mentioned 2D scatterplot. This task was very time consuming and it took us several hours just to understand what the code was doing. We tried our best to mold the code into our necessities, in order to plot both tweets and queries, but we could not figure it out.

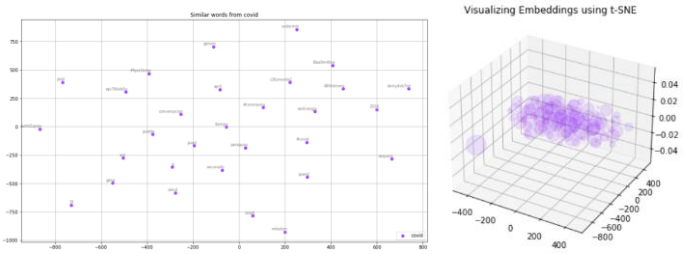

Fig. 15 and 16. Vector representation.