Sergi Solís Valdivieso u172769 - Hong-ming Xiang Vico u172801

IRWA Project Part II: Indexing and Evaluation

Statement

Indexing

1. **Build inverted index:** After having pre-processed the data, you can then create the inverted index.
HINT - you may use the vocabulary data structure, like the one seen during the Practical Labs:
{
Term_id_1: [document_1, document_2, document_4],
Term_id_2: [document_1, document_3, document_5, document_6],
etc...
}
Documents information: Since we are dealing with conjunctive queries (AND), each of the returned documents should contain all the words in the query.

2. **Propose test queries:** Define five queries that will be used to evaluate your search engine (e.g., "covid pandemic", "covid vaccine")

HINT: How to choose the queries? The selection of the queries is up to you but it's suggested to select terms based on the popularity (keywords ranked by term frequencies or by TF-IDF, etc...).

3. **Rank your results:** Implement the TF-IDF algorithm and provide ranking based results.

Evaluation

● There will be 2 main evaluation components:

1. A baseline with 3 queries and the ground truth files for each query will be given to you, using a subset of documents from the dataset.
    a. Query 1: Landfall in South Carolina
    b. Query 2:Help and recovery during the hurricane disaster
    c. Query 3:Floodings in South Carolina

2. You will be the expert judges, so you will be setting the ground truth for each document and query in a binary way for the test queries that you defined in step 2 at the indexing stage.

● For the prior evaluation components you must evaluate your algorithm by using different evaluation techniques and only for the second component (your queries) comment in each of them how they differ, and which information gives each of them:

- **Precision@K (P@K)**
- **Recall@K (R@K)**
- **Average Precision@K (P@K)**
- **F1-Score**
- **Mean Average Precision (MAP)**
- **Mean Reciprocal Rank (MRR)**
- **Normalized Discounted Cumulative Gain (NDCG)**

● Choose one vector representation, TF-IDF or word2vec, and represent the tweets in a two-dimensional scatter plot through the T-SNE (T-distributed Stochastic Neighbor Embedding) algorithm. To do so, you may need first to represent the word as a vector, and then the tweet, i.e., resulted as the average value over the words involved. Any other option rather than T-SNE may be used, but needs to be justified.

HINT: You don't have to know all the theoretical details used in T-SNE, just use the proper library and generate the output and play with it.

Also, you can choose to perform an alternative method to generate a 2-dimensional representation for the word embeddings (like PCA).

Here some T-SNE examples which may be good guidelines for the task:

1. https://towardsdatascience.com/google-news-and-leo-tolstoy-visualizing-word2vec-word-embeddings-with-t-sne-11558d8bd4d
2. https://towardsdatascience.com/visualizing-word-embedding-with-pca-and-t-sne-961a692509f5
3. https://stackoverflow.com/questions/40581010/how-to-run-tsne-on-word2vec-created-from-gensim

GitHub Repository

All the code and resources for the project will be submitted to the following repository:
https://github.com/homexiang3/IRWA-2022-u172769-u172801
And the repository TAG is :

IRWA-2022-u172769-u172801-part-2

Code development

Previous work considerations

This is the second part of the IRWA 2022 project so we skip some previous code documentation provided in the first report such as Google Drive connection, how to load JSON data and how we preprocess the data.

For further details visit:

Sergi Solís Valdivieso u172769 - Hong-ming Xiang Vico u172801

https://github.com/homexiang3/IRWA-2022-u172769-u172801/blob/main/Project/P1/IRWA-2022-u172769-u172801-part-1.pdf

Adjustments

For this lab we added some packets used for mathematical calculations. The list of all the packets used in this project is placed below:

```
import nltk
nltk.download('stopwords')
nltk.download('punkt')
from collections import defaultdict
from array import array
from nltk.stem import PorterStemmer
from nltk.corpus import stopwords
import collections
import json
import re
from tabulate import tabulate
stemmer = nltk.stem.SnowballStemmer('english')
stopwords = set(stopwords.words('english'))
# Packets needed for lab 2
import math
import numpy as np
import collections
import pandas as pd
from numpy import linalg as la
from sklearn.manifold import TSNE
import matplotlib.pyplot as plt
from gensim.models.word2vec import Word2Vec
```

We implemented the map provided in file “*tweet_document_ids_map.csv*”, using panda dataframe to load the data in the file.

```
id_map= pd.read_csv("/content/drive/Shareddrives/IRWA/PROJECT/data/tweet_document_ids_map.csv", sep='\t',
                    engine='python', names = ["doc_id","tweet_id"])
id_map.head()
```

	doc_id	tweet_id
0	doc_1	1575918182698979328
1	doc_2	1575918151862304768
2	doc_3	1575918140839673873
3	doc_4	1575918135009738752
4	doc_5	1575918119251419136

Then, in our Tweet class we added a new attribute name doc_id, that stores the doc_id value from our dataframe based on the map between the tweet_id from our JSON and the dataframe.

```
doc_id = id_map.loc[id_map['tweet_id'] == lines[i]['id'], 'doc_id'].iloc[0]
```

Also, we decided to modify the preprocess function to return the list of terms preprocessed instead of the whole sentence, since we just need the terms for this part of the project. Left image is the code used in part 1 and right image is the code used in part 2.

```
# Preprocess text
def preprocess(text):
    text = text.replace('\\n', '')
    text = remove_emojis(text)
    text = remove_punctuation(text)
    text = remove_numbers(text)
    text = remove_white_space(text)
    words = nltk.tokenize.word_tokenize(text)
    words = [stemmer.stem(word) for word in words]
    words = remove_stopwords(words)
    words = remove_https(words)
    text = " ".join(words)
    return text
```

```
# Preprocess text
def preprocess(text):
    text = text.replace('\\n', '')
    text = remove_emojis(text)
    text = remove_punctuation(text)
    text = remove_numbers(text)
    text = remove_white_space(text)
    words = nltk.tokenize.word_tokenize(text)
    words = [stemmer.stem(word) for word in words]
    words = remove_stopwords(words)
    words = remove_https(words)
    return words
```

Finally, we deleted some debug / checking parts of code since we already proved the correctness of our code in the first part and we want to maintain a clear code.

Indexing

Create Index

For the simple index creation we create a dictionary that stores each term and assigns for each one of them a list with all the documents where appears and the position/s of the term in each doc. We also implemented an additional dictionary named *"tweet_index"* that matches tweet_id with the position in the tweets list (to optimize the process match the data later).

```
# Create index function
def create_simple_index(tweets):
    index = defaultdict(list)
    tweet_index = {}  # dictionary to map tweet id with index in tweets list
    counter = 0 # keep track of index inside tweets
    for t in tweets:  # For each tweet

        tweet_id = t.id
        terms = preprocess(t.tweet) #preprocess tweet and return list of terms
        tweet_index[tweet_id] = counter # Save original tweets position with tweet id to recover all the information
        counter = counter + 1 # Move to next tweets position

        current_page_index = {}
        for position, term in enumerate(terms): #  Loop over all terms
            try:
                # if the term is already in the index for the current page (current_page_index)
                # append the position to the corresponding list
                current_page_index[term][1].append(position)
            except:
                # Add the new term as dict key and initialize the array of positions and add the position
                current_page_index[term] = [tweet_id, array('I', [position])]  #'I' indicates unsigned int (int in Python)

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

```
# Apply index function for all the tweets
index, tweet_index = create_simple_index(tweets)
# Print first 10 results of word 'hurrican' (stemmed word)
print("First 10 Index out of",len(index['hurrican']),"results for the term 'hurrican': {}\n".format(index['hurrican']))
# Print first 10 results of word 'Hurricane'(not stemmed word)
print("First 10 Index out of",len(index['Hurricane']),"results for the term 'Hurricane': {}\n".format(index['Hurricane']))
```

```
First 10 Index out of 796 results for the term 'hurrican': [[1575918105854984192, array('I', [7, 11])], [1575918088473788429, array('I', [7])],

First 10 Index out of 0 results for the term 'Hurricane': []
```

Search Query

First of all we preprocess terms of the query to match the terms in the index, then for each term in the query, we are trying to retrieve all docs that has this term, we use a set to avoid repetitions (eg. both words appear in same doc but we don't want to retrieve it both times!)

```
def search(query, index):
    query = preprocess(query) #create list of query terms (each term is preprocessed to match terms in index)
    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 |= set(term_docs)
        except:
            #term is not in index
            pass
    docs = list(docs)
    return docs
```

Display Query

We create a loop of 5 queries, for each of them we execute the function search to retrieve the documents matching the terms and we prepare our visualization table for our top 5 tweets.

```
# Define 5 querys to visualize - display top 5 tweets (without any rank or order)
# example used in our report: 1. covid pandemic (15) 2. hurricane ian (1087) 3. south carolina (354) 4. god bless (45) 5. help victims (417)
for i in range(5):
  print("Insert your query (i.e.: 'covid pandemic'):\n")
  query = input()
  docs = search(query, index)
  top = 5
  visualization_tweets = []
  #create table headers
  headers = ['DOC_ID','ID','TWEET','USERNAME','DATE','HASHTAGS','LIKES', 'RETWEETS', 'URL']
  print("\n======================\nSample of {} results out of {} for the searched query:\n".format(top, len(docs)))
  #create table of tweets for each match
  for d_id in docs[:top]:
      t = tweet_index[d_id]
      visualization_tweets.append(tweets[t])
  #print table
  print(tabulate(visualization_tweets, headers=headers, tablefmt='grid'))
```

Results

We use the set of 5 queries "damage florida", "hurricane ian", "south carolina", "god bless" and "help victims". The results in this case seems to be low relevant for the queries, given that we are not applying any weighting schema (first matches of any of the terms are retrieved first).

```
damage florida

======================
Sample of 3 results out of 1044 for the searched query:
```

DOC_ID	ID	TWEET
doc_6	1575918105854984192	Ace Handyman Services hopes everyone was safe during the Hurricane. #HurricaneIan #AHS #BringingHelpfulToYourHome https://t.co/BfpOq7tJ
doc_30	1575917717600681984	#HurricaneIan Ian makes 3rd landfall S of Georgetown, South Carolin #wxtwitter #scwx WPDE: https://t.co/EUV6TAWq8i https://t.co/hHDKK4uaUM
doc_46	1575917411001200640	All is safe and well here at Valor Robotics. Our team made it throu

```
hurricane ian

======================
Sample of 3 results out of 1087 for the searched query:
```

DOC_ID	ID	TWEET
doc_6	1575918105854984192	Ace Handyman Services hopes everyone was safe during the Hurricane. Any damages caused by the hurricane is our firs #HurricaneIan #AHS #BringingHelpfulToYourHome https://t.co/BfpOq7tJE0
doc_18	1575917983062380545	#BREAKING Hurricane Ian has made landfall in #SoutCarolina. The storm reformed into a hurricane over the Atlantic
doc_24	1575917833573179392	Just heard from my husband. He and his unit are in North Port assisting the Fire Department and rescuing people. Th

```
south carolina

======================
Sample of 3 results out of 354 for the searched query:
```

DOC_ID	ID	TWEET
doc_10	1575918057037303808	How pissed is GOD to send #HurricaneIan to Florida and South Carolina!? The #MAGA cult has angered GOD and are paying for their sins. @RonDeSantisFL @scgovernorpress #Florida #SouthCarolina #MyrtleBeach
doc_30	1575917717600681984	#HurricaneIan Ian makes 3rd landfall S of Georgetown, South Carolina as Cat. 1 hurricane; widespread damage, e #wxtwitter #scwx WPDE: https://t.co/EUV6TAWq8i https://t.co/hHDKK4uaUM
doc_36	1575917617281658880	@itsbethbooker Hi Beth. So happy & relieved to see that you finally got to hug your mom! Lots of love &amp

```
god bless

======================
Sample of 3 results out of 45 for the searched query:
```

DOC_ID	ID	TWEET
doc_10	1575918057037303808	How pissed is GOD to send #HurricaneIan to Florida and South Carolina!? The #MAGA cult has angered GOD and are paying for their sins. @RonDeSantisFL @scgovernorpress #Florida #SouthCarolina #MyrtleBeach
doc_246	1575915002573205505	God reacts to “Don’t Say Gay Bill” and the treatment of humans. #HurricaneIan
doc_567	1575911109998964736	Releasing merchandise early from @impressink to help #HurricaneIan victims. Half the profits

```
help victims

=====================
Sample of 3 results out of 417 for the searched query:

+----------+-------------------+--------------------------------------------------------------------------------------------------
| DOC_ID   |                ID | TWEET
+==========+===================+==================================================================================================
| doc_20   | 1575917943426535424 | @XfinitySupport is busy doing nothing to help those of us who have been affected by #HurricaneIan. Our
+----------+-------------------+--------------------------------------------------------------------------------------------------
| doc_27   | 1575917773376540672 | In the aftermath of a Disaster some just can’t resist taking advantage of the vulnerability.
|          |                   |
|          |                   | Disaster related scams can happen to anyone, awareness helps reduce the chances it’s you.
|          |                   |
|          |                   | #HurricaneIan #PuertoRico #Florida #Georgia #SouthCarolina  https://t.co/4ncfVvmu6F
+----------+-------------------+--------------------------------------------------------------------------------------------------
| doc_42   | 1575917468996239360 | If you need to be out on the road, #OnStar Advisors are here to help with routing assistance. Just pus
+----------+-------------------+--------------------------------------------------------------------------------------------------
```

TF-IDF Implementation

Our new create index function does the same that the simple one but now for each word-doc pair computes the term frequency, document frequency and inverse document frequency.

```python
# Create index function
def create_tfidf_index(tweets,num_tweets):
    index = defaultdict(list)
    tweet_index = {}  # dictionary to map tweet id with index in tweets list
    counter = 0 # keep track of index inside tweets
    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) # inverse document frequency for each term
    for t in tweets:  # for all tweets

        tweet_id = t.id
        terms = preprocess(t.tweet) #preprocess tweet and return list of terms
        tweet_index[tweet_id] = counter # Save original tweets position with tweet id to recover all the information
        counter = counter + 1 # Move to next tweets position
        current_page_index = {}

        for position, term in enumerate(terms):
            try:
                # if the term is already in the dict append the position to the corresponding list
                current_page_index[term][1].append(position)
            except:
                # Add the new term as dict key and initialize the array of positions and add the position
                current_page_index[term] = [tweet_id, 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_page_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[1]) ** 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_page_index.items():
            # append the tf for current term (tf = term frequency in current doc/norm)
            tf[term].append(np.round(len(posting[1]) / norm, 4)) ## SEE formula (1) above
            #increment the document frequency of current term (number of documents containing the current term)
            df[term] += 1 # increment DF for current term

        # Compute IDF
        for term in df:
            idf[term] = np.round(np.log(float(num_tweets / df[term])), 4)

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

    return index, tf, df, idf, tweet_index
```

We implemented a new auxiliary function, that given a query, the docs with at least one query ranks the docs with the tf-idf vectorization and cosine similarity between docs and queries.

```
def rank_documents(terms, docs, index, idf, tf, title_index):

    # 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 ke
    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.

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

        # Generate doc_vectors for matching docs
        for doc_index, (doc, postings) in enumerate(index[term]):
            if doc in docs:
                doc_vectors[doc][termIndex] = tf[term][doc_index] * idf[term]

    # Calculate the score of each doc
    # compute the cosine similarity between queyVector and each docVector:

    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]
    result_rank = [x[0] for x in doc_scores] #get rank value
    #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_tfidf(query, index)
    return result_docs, result_rank
```

Search function is basically the same as in the simple case, but now docs are sorted by rank using the function *“rank_documents”:*

```
def search_tfidf(query, index):
    query = preprocess(query)#create list of query terms (each term is preprocessed to match terms in index)
    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 |= set(term_docs)
        except:
            #term is not in index
            pass
    docs = list(docs)
    ranked_docs, ranked_score = rank_documents(query, docs, index, idf, tf, tweet_index)#rank docs
    return ranked_docs, ranked_score
```

Similarly as we did in the simple case, we prepare a loop of 5 queries but now we search for the top ranked docs. Notice that we also take the ranked score to print it on screen and check if the result is correct.

```
# Define 5 querys to visualize - top 3 ranked tweets displayed
# example used in our report: 1. covid pandemic (15) 2. hurricane ian (1087) 3. south carolina (354) 4. god bless (45) 5. help victims (417)
for i in range(5):
  print("Insert your query (i.e.: 'covid pandemic'):\n")
  query = input()
  ranked_docs, ranked_score = search_tfidf(query, index)
  top = 3
  visualization_tweets = []
  #create table headers
  headers = ['DOC_ID','ID','TWEET','USERNAME','DATE','HASHTAGS','LIKES', 'RETWEETS', 'URL']
  print("\n======================\nSample of {} results out of {} for the searched query:\n".format(top, len(ranked_docs)))
  #create table of tweets for each match
  for d_id in ranked_docs[:top]:
      t = tweet_index[d_id]
      visualization_tweets.append(tweets[t])
  #print ranked score
  print("Ranked Scores:", ranked_score[:top])
  #print table
  print(tabulate(visualization_tweets, headers=headers, tablefmt='grid'))
```

Results

We run again the set of 5 queries “damage florida”, “hurricane ian”, “south carolina”, “god bless” and “help victims”. The results in this case are much more accurate of the queries since we are taking into account TF-IDF weights, specially, we can notice how short tweets with query words appear in the top 3 most relevant documents. We can also confirm that ranked scores are in descending order.

```
damage florida

======================
Sample of 3 results out of 1044 for the searched query:

Ranked Scores: [2.8749627271127562, 2.8749627271127562, 2.8613919386486755]
+----------+--------------------+------------------------------------------------------------
| DOC_ID   |                 ID | TWEET
+==========+====================+============================================================
| doc_1555 | 1575895366167412736 | The damage from #HurricaneIan is “catastrophic” and historic.
|          |                    | https://t.co/OY323JCzyQ 02
+----------+--------------------+------------------------------------------------------------
| doc_2687 | 1575873273317163008 | Just over Blind Pass Bridge on #Sanibel #Captiva Rd. Significant
+----------+--------------------+------------------------------------------------------------
| doc_339  | 1575913950578757632 | as #Biden spends 3 of 5 minutes talking about Russia he hasn't a
+----------+--------------------+------------------------------------------------------------
```

```
hurricane ian

======================
Sample of 3 results out of 1087 for the searched query:

Ranked Scores: [2.1383822503267584, 1.8517338503002754, 1.8517338503002754]
+----------+---------------------+----------------------------------------------------------------------------------------------------------+-----------------+
| DOC_ID   |                  ID | TWEET                                                                                                    | USERNAME        |
+==========+=====================+==========================================================================================================+=================+
| doc_634  | 1575910361298968576 | Hurricane IAN #Ian #HurricaneIan #HurricanIan #Huracan #HuracanIan #Hurricane https://t.co/HbllO4Q3vB    | cesarharamillo  |
+----------+---------------------+----------------------------------------------------------------------------------------------------------+-----------------+
| doc_640  | 1575910304159977472 | Hurricane Ian before and after #HurricaneIan https://t.co/XZstkI2pN2                                     | kadenfields8    |
+----------+---------------------+----------------------------------------------------------------------------------------------------------+-----------------+
| doc_1217 | 1575902689040666626 | Hurricane Ian on tour😭                                                                                  | besmarterpeople |
|          |                     | #HurricaneIan                                                                                            |                 |
+----------+---------------------+----------------------------------------------------------------------------------------------------------+-----------------+
```

```
south carolina

======================
Sample of 3 results out of 354 for the searched query:

Ranked Scores: [5.750153366279658, 4.979349988118859, 4.8059638892671845]
+----------+---------------------+----------------------------------------------------------------------------------------------+----------------+
| DOC_ID   |                  ID | TWEET                                                                                        | USERNAME       |
+==========+=====================+==============================================================================================+================+
| doc_254  | 1575914929898782720 | South Carolina #HurricaneIan https://t.co/yTA4dFUC2V                                         | webgyrl2       |
+----------+---------------------+----------------------------------------------------------------------------------------------+----------------+
| doc_174  | 1575915969913839616 | South Carolina #HurricaneIan here we go                                                      | TheAstuteGaloot |
+----------+---------------------+----------------------------------------------------------------------------------------------+----------------+
| doc_493  | 1575912058163408896 | Just south of Myrtle Beach in South Carolina. #HurricaneIan #Ian #ScWx https://t.co/ErHr5X5cQO | Damian_Wx      |
+----------+---------------------+----------------------------------------------------------------------------------------------+----------------+
```

```
god bless

======================
Sample of 3 results out of 45 for the searched query:

Ranked Scores: [11.008039858203505, 11.008039858203505, 10.557137146207896]
+----------+--------------------+----------------------------------------------------------------------------------------------
| DOC_ID   |                 ID | TWEET
+==========+====================+==============================================================================================
| doc_3782 | 1575859119042416640 | Good morning Patriotsus If you have the ability to help, please join me and support the https://t.c
+----------+--------------------+----------------------------------------------------------------------------------------------
| doc_3800 | 1575858935281881088 | My thoughts and prayers go out special to those affected by the hurricane Ian.. God Bless. 🙏
|          |                    | #Florida
|          |                    | #HurricaneIan https://t.co/8dcUI9MRA7
+----------+--------------------+----------------------------------------------------------------------------------------------
| doc_2372 | 1575877434712334336 | @KellyClarksonTV I just wanted to let @kellyclarkson know that we are praying for those who are aff
+----------+--------------------+----------------------------------------------------------------------------------------------

help victims

======================
Sample of 3 results out of 417 for the searched query:

Ranked Scores: [8.030304493960108, 6.862258668688771, 6.349543088247527]
+----------+--------------------+----------------------------------------------------------------------------------------------
| DOC_ID   |                 ID | TWEET
+==========+====================+==============================================================================================
| doc_321  | 1575914189071138818 | A list of ways you can help the victims of #HurricaneIan
|          |                    | https://t.co/D77WKvhKhI
+----------+--------------------+----------------------------------------------------------------------------------------------
| doc_632  | 1575910393767133184 | @DonaldJTrumpJr #MAGA doesn't believe in #climatechange so how can you politicize these folks?  Trump's woukd just
+----------+--------------------+----------------------------------------------------------------------------------------------
| doc_3057 | 1575868689768857601 | #HurricaneIan —&gt; How to help victims of Hurricane Ian - CBS News https://t.co/JD7qUh9AZ9
+----------+--------------------+----------------------------------------------------------------------------------------------
```

Evaluation

Load Data

First of all we loaded the data provided in evaluation_gt.csv using the pandas dataframe library. We check how the results look, using the head() function. Then, we check that in effect we just have two labels for our relevances.

```
evaluation_data = pd.read_csv("/content/drive/Shareddrives/IRWA/PROJECT/data/evaluation_gt.csv")
evaluation_data.head()
```

	doc	query_id	label
0	doc_12	1	1
1	doc_9	1	1
2	doc_18	1	1
3	doc_45	1	1
4	doc_501	1	1

```
# Our ground truth consist in a binary classification 1 represents that the query presents results and 0 that not
print_result = evaluation_data["label"].unique()
print("The ground truth of our dataset is composed of {} Relevance Levels: {}".format(len(print_result), sorted(print_result)))
```

```
The ground truth of our dataset is composed of 2 Relevance Levels: [0, 1]
```

Provided data with our algorithm

We want to compare our algorithm results with the data provided to evaluate the algorithm.

First of all, we will need to create a new column in the dataframe named “y_predicted” that stores the ranked score for each of the docs retrieved with the provided query in our system.

```
evaluation_data.insert(2,"y_predicted",0)
evaluation_data.head()
```

Then we define the function that does this process for each of the queries, this function, given the query and the number of the query retrieves the data and stores the score for each doc in the provided dataset, then at the end returns a subset of the dataframe with only the specific query information.

```
#function that assigns y_predicted = ranked_score of our algorithm
def add_y_predicted(query, num_query):
  query_res = evaluation_data[evaluation_data["query_id"] == num_query]
  ranked_docs, ranked_score = search_tfidf(query, index)
  pos = 0
  for d_id in ranked_docs:
    t = tweets[tweet_index[d_id]]
    query_res.loc[query_res["doc"]==t.doc_id, "y_predicted"] = ranked_score[pos]
    pos += 1
  return query_res
```

Then we execute the function for each query and visualize one of the dataframes (notice that some non-relevant docs will get retrieved given the score order on the top 10)

```
q1 = "Landfall in South Carolina"
q2 = "Help and recovery during the hurricane disaster"
q3 = "Floodings in South Carolina"

query1_df = add_y_predicted(q1,1)
query2_df = add_y_predicted(q2,2)
query3_df = add_y_predicted(q3,3)

#see result example (we will sort later by y_predicted to take top k results)
query2_df.sort_values("y_predicted", ascending=False).head(10)
```

	doc	query_id	y_predicted	label
19	doc_504	2	2.861818	1
17	doc_402	2	2.781130	1
12	doc_268	2	2.220176	1
14	doc_321	2	1.090972	1
49	doc_1233	2	1.084372	0
11	doc_175	2	1.041029	1
10	doc_158	2	0.721055	1
16	doc_373	2	0.676735	1
15	doc_358	2	0.508649	1
13	doc_303	2	0.461077	1

Although we will measure this provided dataset with all the measures specified, we will only explain our custom queries results since the results as is requested in the statement. (Results for this dataset are very straightforward and predictable given results near 1 almost for all measures)

Custom queries dataframes preparation

Given our set of queries Q = [“damage florida”, “hurricane ian”, “south carolina”, “god bless”, “help victims”] for each of them we visualize and take the top 20 results (since provided dataset has 20 retrieves for each query we believe that is enough), and create the associated dataframe for each one.

First of all, we have the definition of our visualize and retrieve results function, which will print the top 20 results of the query and retrieve the information.

```
def visualize_retrieve_top20(query):
  top = 20
  rank = 1
  visualization_tweets = []
  headers = ['TWEET']
  ranked_docs, ranked_score = search_tfidf(query, index)
  for d_id in ranked_docs[:top]:
    t = tweets[tweet_index[d_id]]
    visualization_tweets.append([t.tweet])
  print(tabulate(visualization_tweets, headers=headers, tablefmt='grid'))
  return ranked_docs[:top],ranked_score[:top]
```

Then, for each query, we will execute the function and store the values, after visualizing the top 20 tweets we design this list of binary relevances following our own judgment.

```
custom_q1 = "damage florida"
rd_q1, rs_q1 = visualize_retrieve_top20(custom_q1)
q1_relevance = [0,0,1,1,0,1,1,0,1,1,1,1,0,0,0,1,0,1,1,1]
```

```
+-----------------------------------------------------------------------------------------------------------------
| TWEET
+=================================================================================================================
| The damage from #HurricaneIan is “catastrophic” and historic.
| https://t.co/OY323JCzyQ 02
+-----------------------------------------------------------------------------------------------------------------
| Just over Blind Pass Bridge on #Sanibel #Captiva Rd. Significant damage here. Structures missing and damages. #Ian #Hurri
+-----------------------------------------------------------------------------------------------------------------
| as #Biden spends 3 of 5 minutes talking about Russia he hasn't a clue of the damage done in Florida. Drone footage shows
+-----------------------------------------------------------------------------------------------------------------
| . #HurricaneIan is making its way into the Carolinas as Florida works to survey the damage.
|
| https://t.co/m3ABKeOxBG
```

After that, we define a function that given the ranked docs, scores, our relevance list, and the query id, return a specific data frame with all the data from the top 20 docs for each of the queries.

```
def create_df(rd,rs,rel,q_id):
  doc_ids = []
  q_ids = []
  for d_id in rd:
     t = tweets[tweet_index[d_id]]
     doc_ids.append(t.doc_id)
     q_ids.append(q_id)
  df = pd.DataFrame(list(zip(doc_ids, q_ids,rs,rel)),
                 columns =['doc', 'query_id','y_predicted','label'])
  return df
```

Finally, we execute the previous function for each of our custom queries and check that the results are correct.

```
#Create custom queries df
custom_q1_df = create_df(rd_q1,rs_q1,q1_relevance,1)
custom_q2_df = create_df(rd_q2,rs_q2,q2_relevance,2)
custom_q3_df = create_df(rd_q3,rs_q3,q3_relevance,3)
custom_q4_df = create_df(rd_q4,rs_q4,q4_relevance,4)
custom_q5_df = create_df(rd_q5,rs_q5,q5_relevance,5)
#Show 1 as an example
custom_q5_df.head()
```

	doc	query_id	y_predicted	label
0	doc_321	5	8.030304	1
1	doc_632	5	6.862259	0
2	doc_3057	5	6.349543	1
3	doc_1081	5	6.305326	0
4	doc_3865	5	6.305326	1

With all these steps done, we will evaluate the custom queries made by us to draw conclusions about how they differ and which one of them is the best in each situation.

Precision@K (P@K)

```
def precision_at_k(doc_score, y_score, k=10): #binary relevance, predicted relevance, k for a given query
    """
    Parameters
    ----------
    doc_score: Ground truth (true relevance labels).
    y_score: Predicted scores.
    k : number of doc to consider.

    Returns
    -------
    precision @k : float

    """
    order = np.argsort(y_score)[::-1] #we get the ranking of the documents accoirding to the predicted score/ use n
    doc_score = np.take(doc_score, order[:k]) # align the binary relevance to the corresponding document / use the
    relevant = sum(doc_score == 1) #get number of relevant documents
    return float(relevant) / k #calculate precision at k, which is the number of relevant documents retrieved at k
```

Custom Queries

```
[ ] k = 10

    pcq1 = precision_at_k(custom_q1_df["label"],custom_q1_df["y_predicted"],k)
    pcq2 = precision_at_k(custom_q2_df["label"],custom_q2_df["y_predicted"],k)
    pcq3 = precision_at_k(custom_q3_df["label"],custom_q3_df["y_predicted"],k)
    pcq4 = precision_at_k(custom_q4_df["label"],custom_q4_df["y_predicted"],k)
    pcq5 = precision_at_k(custom_q5_df["label"],custom_q5_df["y_predicted"],k)

    print(pcq1,pcq2,pcq3,pcq4,pcq5)

    0.6 1.0 0.9 0.6 0.7
```

Since higher precision means that an algorithm returns more relevant results than irrelevant ones, we can say that the custom queries q2 and q3 (“hurricane ian” and “south carolina”) will give us a better quality result than the others. We believe that this behavior is due to the high correlation between these two words, which retrieves more precise results

Recall@K (R@K)

```
def recall_at_k(doc_score, y_score, k=10): #binary relevance, predicted relevance, k for a given query
    """
    Parameters
    ----------
    doc_score: Ground truth (true relevance labels).
    y_score: Predicted scores.
    k : number of doc to consider.

    Returns
    -------
    recall @k : float

    """
    order = np.argsort(y_score)[::-1] #we get the ranking of the documents accoirding to the predicted score/ use np.argsort and [::1]
    doc_score = np.take(doc_score, order[:k]) # align the binary relevance to the corresponding document / use the indexes of point 1 t
    all_relevants = sum(np.take(doc_score) == 1) # take all relevant docs
    relevant = sum(doc_score == 1) #get number of relevant documents
    return float(relevant) / all_relevants #calculate recall at k, which is the number of relevant documents among all relevant docs
```

```
[ ] k = 10
    rcq1 = recall_at_k(custom_q1_df["label"],custom_q1_df["y_predicted"],k)
    rcq2 = recall_at_k(custom_q2_df["label"],custom_q2_df["y_predicted"],k)
    rcq3 = recall_at_k(custom_q3_df["label"],custom_q3_df["y_predicted"],k)
    rcq4 = recall_at_k(custom_q4_df["label"],custom_q4_df["y_predicted"],k)
    rcq5 = recall_at_k(custom_q5_df["label"],custom_q5_df["y_predicted"],k)

    print(rcq1,rcq2,rcq3,rcq4,rcq5)

    0.5 0.5555555555555556 0.6428571428571429 0.75 0.5384615384615384
```

On the other hand, high recall means that an algorithm returns most of the relevant results. So we can say that query 4 (“god bless”) will give a better result in the quantity scope. That means that in proportion, there are more relevant docs in the top 10 than in the other ones given all the docs labeled as relevant.

Average Precision@K (P@K)

```
def avg_precision_at_k(doc_score, y_score, k=10): #binary relevance, predicted relevance, k for a given query
    """
    Parameters
    ----------
    doc_score: Ground truth (true relevance labels).
    y_score: Predicted scores.
    k : number of doc to consider.

    Returns
    -------
    average precision @k : float
    """
    gtp = np.sum(doc_score == 1) #Total number of gt positives
    order = np.argsort(y_score)[::-1] #same as for precision
    doc_score = np.take(doc_score, order[:k]) #same as for precision
    ## if all documents are not relevant
    if gtp == 0:
        return 0
    n_relevant_at_i = 0
    prec_at_i = 0
    for i in range(len(doc_score)):
        if doc_score[i] == 1: #only add the P@k when the doc is relevant
            n_relevant_at_i += 1
            prec_at_i += n_relevant_at_i / (i + 1) #calculate P@K (#docs relevant at k/k)
    return prec_at_i / gtp #return ap
```

Custom Queries

```
[ ] k = 10
    avgp_cq1 = avg_precision_at_k(np.array(custom_q1_df["label"]),np.array(custom_q1_df["y_predicted"]),k)
    avgp_cq2 = avg_precision_at_k(np.array(custom_q2_df["label"]),np.array(custom_q2_df["y_predicted"]),k)
    avgp_cq3 = avg_precision_at_k(np.array(custom_q3_df["label"]),np.array(custom_q3_df["y_predicted"]),k)
    avgp_cq4 = avg_precision_at_k(np.array(custom_q4_df["label"]),np.array(custom_q4_df["y_predicted"]),k)
    avgp_cq5 = avg_precision_at_k(np.array(custom_q5_df["label"]),np.array(custom_q5_df["y_predicted"]),k)
    print(avgp_cq1,avgp_cq2,avgp_cq3,avgp_cq4,avgp_cq5)

    0.25502645502645505 0.5555555555555556 0.6277777777777779 0.6163690476190476 0.37152014652014653
```

Getting more correct predictions, leads to a better PR curve and, as a result, to a higher Average Precision. In this case, the best metric value will be given by query 3 (“south carolina” ← was predictable since it is the one with the highest precision) but we can see how other ones have lower average precision compared to the regular precision probably because they have retrieved non-relevant in highest ranks.

F1-Score

```
def f1_score(precision,recall):
  return 2*(precision*recall)/(precision+recall)
```

```
f1 = f1_score(0.5,1)
print(f1)
```

Custom Queries

```
[ ] f1_cq1 = f1_score(pcq1,rcq1)
    f1_cq2 = f1_score(pcq2,rcq2)
    f1_cq3 = f1_score(pcq3,rcq3)
    f1_cq4 = f1_score(pcq4,rcq4)
    f1_cq5 = f1_score(pcq5,rcq5)

    print(f1_cq1,f1_cq2,f1_cq3,f1_cq4,f1_cq5)

    0.5454545454545454 0.7142857142857143 0.75 0.6666666666666665 0.608695652173913
```

The F1-score is used as a statistical measure to rate performance, as it is the harmonic mean between precision and recall. In this case, the query that gives the best result is again query 3 ("south carolina") due to its high value in precision and recall.

Mean Average Precision (MAP)

```
def map_at_k(search_res, k=10): #receives all the search esults dataframe containing all the queries and the results and relevances
    """
    Parameters
    ----------
    search_res: search results dataset containing:
        query_id: query id.
        doc_id: document id.
        predicted_relevance: relevance predicted through LightGBM.
        doc_score: actual score of the document for the query (ground truth).

    Returns
    -------
    mean average precision @ k : float
    """
    avp = []
    for q in search_res["query_id"].unique():  # loop over all query ids
        curr_data = search_res[search_res["query_id"] == q]  # select data for current query (get a slice of the dataframe keeping o
        avp.append(avg_precision_at_k(np.array(curr_data["is_relevant"]),
                   np.array(curr_data["predicted_relevance"]), k))  #append average precision for current query
    return np.sum(avp) / len(avp), avp  # return mean average precision
```

Custom Queries

```
[ ] k = 10
    allcq_df = pd.concat([custom_q1_df, custom_q2_df, custom_q3_df, custom_q4_df, custom_q5_df], ignore_index=True)
    custom_map_k, custom_avp_k = map_at_k(allcq_df, k)

    custom_map_k

    0.48524979649979655
```

mAP is a way to summarize the precision-recall curve into a single value representing the average of all precisions. In this case, it gives a low value because the average precision in all the custom queries does not offer enough quality, we thought that this is because of the ambiguity of some queries selected.

Mean Reciprocal Rank (MRR)

```
def rr_at_k(doc_score, y_score, k=10):
    """
    Parameters
    ----------
    doc_score: Ground truth (true relevance labels).
    y_score: Predicted scores.
    k : number of doc to consider.

    Returns
    -------
    Reciprocal Rank for qurrent query
    """

    order = np.argsort(y_score)[::-1]  # get the list of indexes of the predicted sco
    doc_score = np.take(doc_score, order[
                            :k])  # sort the actual relevance label of the documents
    if np.sum(doc_score) == 0:  # if there are not relevant doument return 0
        return 0
    return 1 / (np.argmax(doc_score == 1) + 1)  # hint: to get the position of the fi
```

```
def mrr(search_results,k = 10):
  RRs = []
  for q in search_results['query_id'].unique():  # loop over all query ids, get rrs for each query at each k
      labels = np.array(search_results[search_results['query_id'] == q]["label"])  # get labels for current query
      scores = np.array(search_results[search_results['query_id'] == q]["y_predicted"])  # get predicted score for current query
      RRs.append(rr_at_k(labels, scores, k))  # append RR for current query
  return np.round(float(sum(RRs) / len(RRs)), 4)  # Mean RR at current k
```

```
k = 10
custom_mrr_value = mrr(allcq_df,k)

custom_mrr_value
```

```
0.8667
```

Mean Reciprocal Rank indicates the probability of correctness and in this case we can see that the model based on custom queries is quite good, meaning that overall the first relevant doc appears fast enough.

Normalized Discounted Cumulative Gain (NDCG)

```
def dcg_at_k(doc_score, y_score, k=10): #doc_scire are the labels (ground truth)
    order = np.argsort(y_score)[::-1]  # get the list of indexes of the predicted
    doc_score = np.take(doc_score, order[:k])  # sort the actual relevance label
    gain = 2 ** doc_score - 1  # First we calculate the upper part of the formula
    discounts = np.log2(np.arange(len(doc_score)) + 2)  # Compute denominator (np
    return np.sum(gain / discounts)  #return dcg@k


def ndcg_at_k(doc_score, y_score, k=10):
    dcg_max = dcg_at_k(doc_score, doc_score, k) #ideal dcg
    #print(dcg_max)
    if not dcg_max:
        return 0
    return np.round(dcg_at_k(doc_score, y_score, k) / dcg_max, 4)
```

Custom Queries

```
k = 10
ndcg_cq1 = np.round(ndcg_at_k(np.array(custom_q1_df["label"]),np.array(custom_q1_df["y_predicted"]), k), 4)
ndcg_cq2 = np.round(ndcg_at_k(np.array(custom_q2_df["label"]),np.array(custom_q2_df["y_predicted"]), k), 4)
ndcg_cq3 = np.round(ndcg_at_k(np.array(custom_q3_df["label"]),np.array(custom_q3_df["y_predicted"]), k), 4)
ndcg_cq4 = np.round(ndcg_at_k(np.array(custom_q4_df["label"]),np.array(custom_q4_df["y_predicted"]), k), 4)
ndcg_cq5 = np.round(ndcg_at_k(np.array(custom_q5_df["label"]),np.array(custom_q5_df["y_predicted"]), k), 4)

print(ndcg_cq1,ndcg_cq2,ndcg_cq3,ndcg_cq4,ndcg_cq5)

0.4865 1.0 0.9306 0.7798 0.688
```

NDCG is a quality ranking measure that tells us whether a list of documents has been ordered from most relevant to least relevant. In this case, we can see that the best ranking of documents related to custom queries is the second one (“hurricane ian”). That’s because the first 10 results of this query are relevant so the order is perfect in binary relevance. On the other hand (“damage florida”) is the lowest one, because the top 10 results are pretty far from the ideal one, meaning that there is a lot of non-relevance in the first 10 results.

Vector Representation

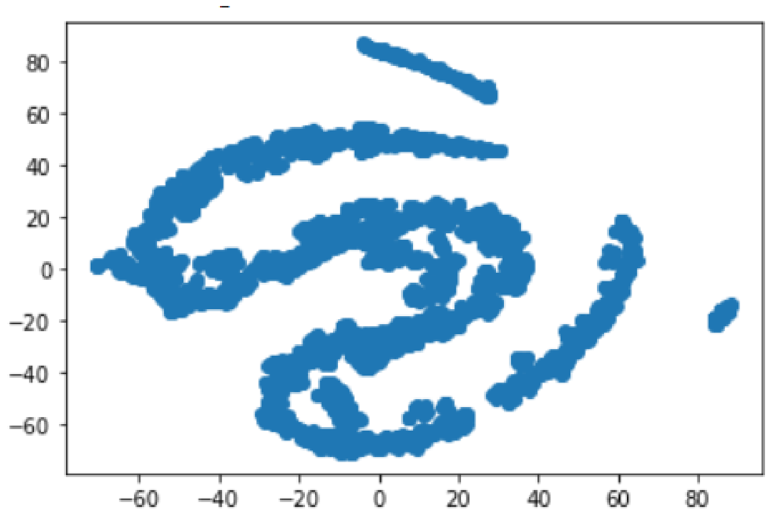
We decided to use the word2vec method to represent all the tweets as vectors. First of all, we define a function that takes for all the tweets, all the terms, and computes the average value of them given the word2vec model. Then returns a list with the word2vec value for each tweet.

```
def vectorize_tweets(tweets,model):
  word2vec_tweets = []
  for tw in tweets:
    terms = preprocess(tw.tweet)
    values = []
    for t in terms:
      values.append(model.wv[t])#append each of the terms word2vector in values
    word2vec_tweets.append(sum(values)/len(values))#create for each tweet the avg value of their word2_vec
  return word2vec_tweets
```

Then, we create a list named corpus that stores the tweets in terms sublists (given Word2Vec this format is needed), and create our model with this corpus.
Finally, we apply vectorization for all the tweets and use TSNE to transform tweets into 2D vectors and plot them on the screen.

```
# Prepare corpus of sentences separed in terms
corpus = []
for t in tweets:
  terms = preprocess(t.tweet)
  corpus.append(terms)

# Create model with word2vec and our corpus of tweets
model = Word2Vec(corpus, workers=4, size=100, min_count=1, window=10, sample=1e-3)
# Vectorize tweets
vectorized_tweets = vectorize_tweets(tweets,model)
X = vectorized_tweets
# Apply TSNE
tsne = TSNE(n_components=2)
X_tsne = tsne.fit_transform(X)
# Plot TSNE
plt.scatter(X_tsne[:, 0], X_tsne[:, 1])
plt.show()
```


80
60
40
20
0
−20
−40
−60
−60
−40
−20
0
20
40
60
80