PROJECT REPORT

IRWA - PART 2 - G102-2

Clàudia Morales - u199906
Roger Sola - u199780
Queralt Zamora - u199903
G102-2

1. Introduction

This document is the Part 2 Project Report, prepared by team G102-2. Here, it is described all the necessary information about the elaboration of the code for this second part, including the assumptions, methods, and algorithms employed. Each subsection presents the reasoning behind the decisions made and is classified into two main parts and a conclusion.

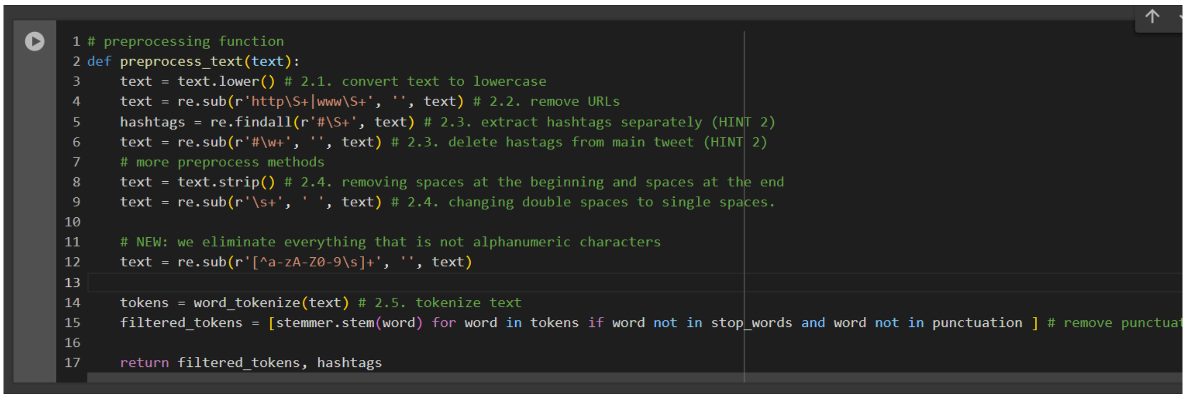
The team developed the code using a Google Colaboratory file, which required importing libraries and the provided dataset. The project aims to build and evaluate a search engine for tweets using TF-IDF ranking, query evaluation metrics and visualization techniques.

The repository for the deliveries has been shared with the labs teacher Francielle Do Nascimento, but since it is a public repository, the link to get in is the following:

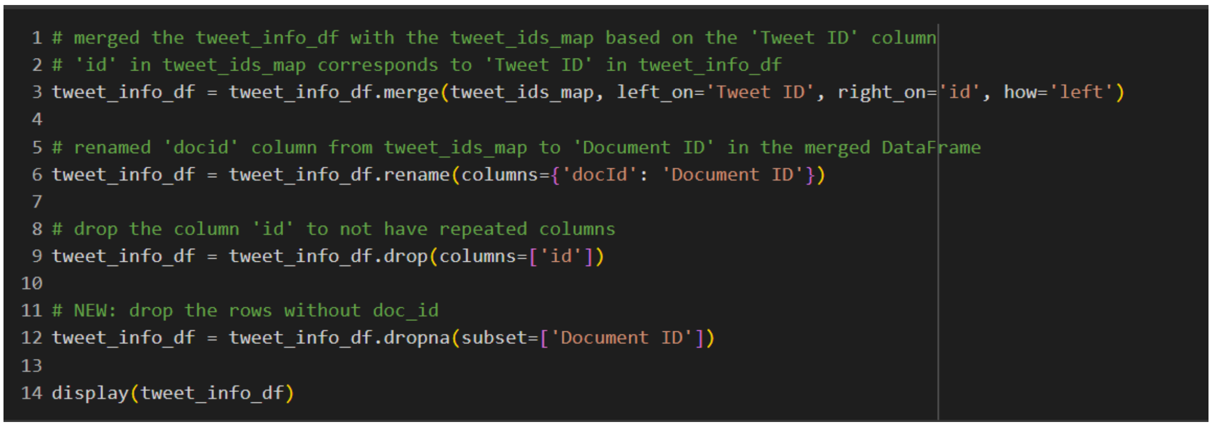
https://github.com/QuerZamora/IRWA_G102_2

IMPORTANT

To improve our project results, we made some modifications to the preprocessing stage. Specifically, we refined the tokenization process by removing additional stopwords and special characters, leaving only clean words without punctuation, emojis, or characters outside the Latin alphabet.



```python
# preprocessing function
def preprocess_text(text):
    text = text.lower() # 2.1. convert text to lowercase
    text = re.sub(r'http\S+|www\S+', '', text) # 2.2. remove URLs
    hashtags = re.findall(r'#\S+', text) # 2.3. extract hashtags separately (HINT 2)
    text = re.sub(r'#\w+', '', text) # 2.3. delete hastags from main tweet (HINT 2)
    # more preprocess methods
    text = text.strip() # 2.4. removing spaces at the beginning and spaces at the end
    text = re.sub(r'\s+', ' ', text) # 2.4. changing double spaces to single spaces.

    # NEW: we eliminate everything that is not alphanumeric characters
    text = re.sub(r'[^a-zA-Z0-9\s]+', '', text)

    tokens = word_tokenize(text) # 2.5. tokenize text
    filtered_tokens = [stemmer.stem(word) for word in tokens if word not in stop_words and word not in punctuation ] # remove punctuat

    return filtered_tokens, hashtags
```



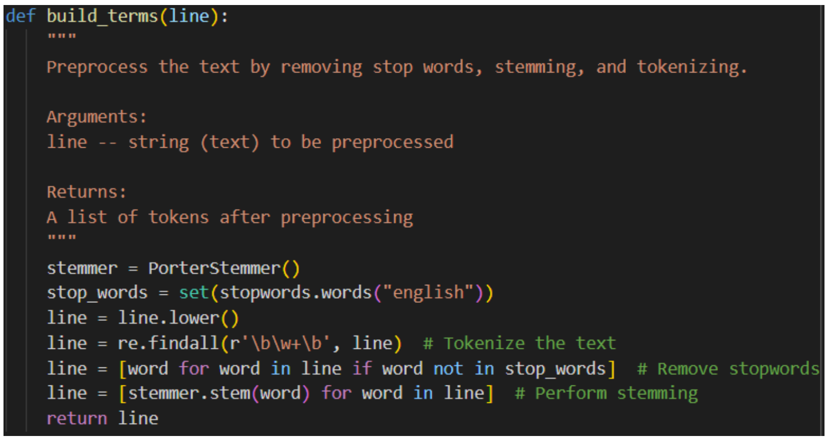
```python
# merged the tweet_info_df with the tweet_ids_map based on the 'Tweet ID' column
# 'id' in tweet_ids_map corresponds to 'Tweet ID' in tweet_info_df
tweet_info_df = tweet_info_df.merge(tweet_ids_map, left_on='Tweet ID', right_on='id', how='left')

# renamed 'docid' column from tweet_ids_map to 'Document ID' in the merged DataFrame
tweet_info_df = tweet_info_df.rename(columns={'docId': 'Document ID'})

# drop the column 'id' to not have repeated columns
tweet_info_df = tweet_info_df.drop(columns=['id'])

# NEW: drop the rows without doc_id
tweet_info_df = tweet_info_df.dropna(subset=['Document ID'])

display(tweet_info_df)
```

Clàudia Morales - u199906
Roger Sola - u199780
Queralt Zamora - u199903
G102-2

2. Indexing

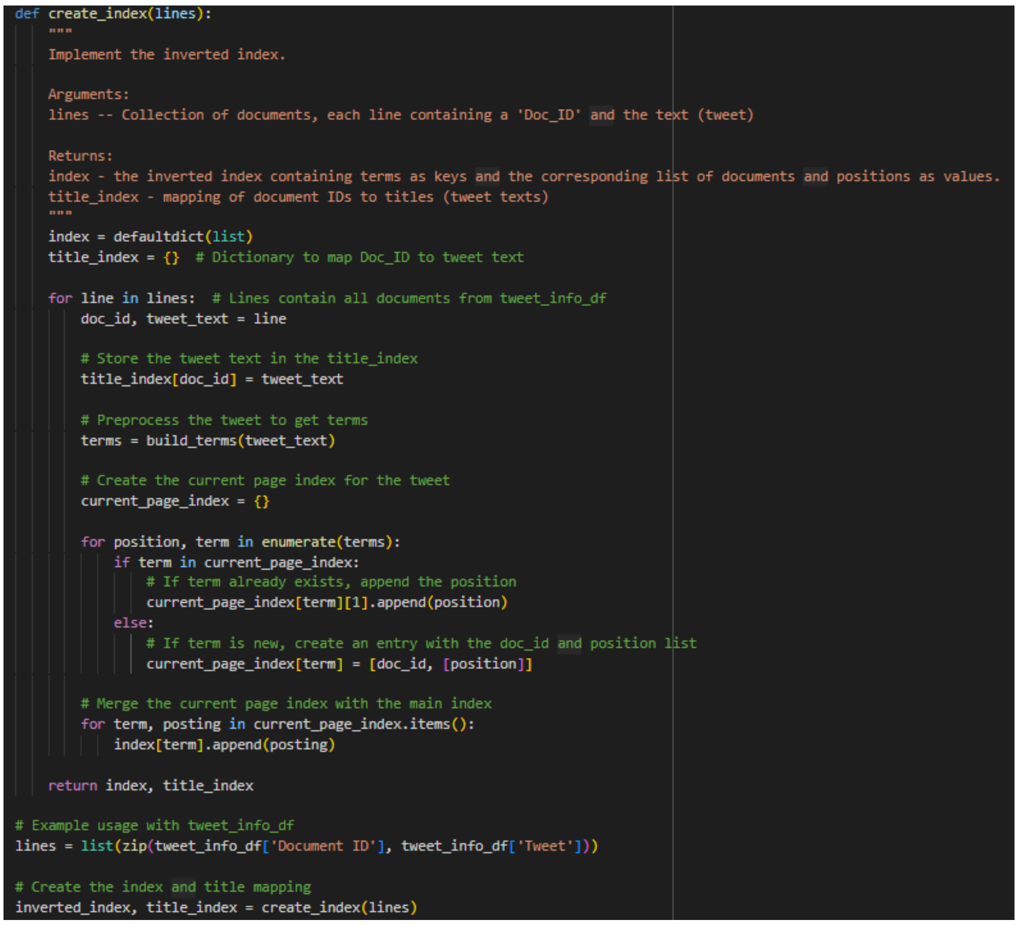
2.1. Build inverted index

Using the build_terms function, each tweet was tokenized, converted to lowercase, and filtered to remove stopwords. Stemming was also applied, which reduces words to their root form.



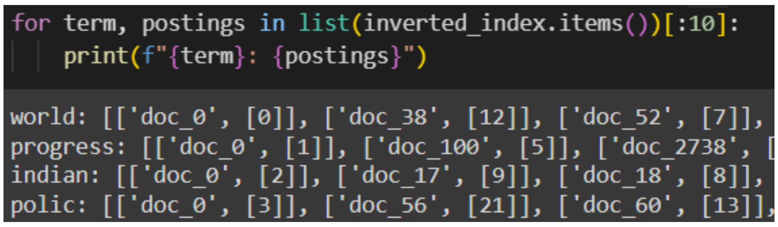
```python
def build_terms(line):
    """
    Preprocess the text by removing stop words, stemming, and tokenizing.

    Arguments:
    line -- string (text) to be preprocessed

    Returns:
    A list of tokens after preprocessing
    """
    stemmer = PorterStemmer()
    stop_words = set(stopwords.words("english"))
    line = line.lower()
    line = re.findall(r'\b\w+\b', line)  # Tokenize the text
    line = [word for word in line if word not in stop_words]  # Remove stopwords
    line = [stemmer.stem(word) for word in line]  # Perform stemming
    return line
```

The function create_index iterates through each tweet in the dataset, assigning each a unique Doc_ID. Each term in the processed tweet text is then added to the inverted index. For each term, the index maintains the document ID where the term appears and a list of positions where the term occurs within that document, allowing us to capture term location, which is going to be useful for ranking algorithms.



```python
def create_index(lines):
    """
    Implement the inverted index.

    Arguments:
    lines -- Collection of documents, each line containing a 'Doc_ID' and the text (tweet)

    Returns:
    index - the inverted index containing terms as keys and the corresponding list of documents and positions as values.
    title_index - mapping of document IDs to titles (tweet texts)
    """
    index = defaultdict(list)
    title_index = {}  # Dictionary to map Doc_ID to tweet text

    for line in lines:  # Lines contain all documents from tweet_info_df
        doc_id, tweet_text = line

        # Store the tweet text in the title_index
        title_index[doc_id] = tweet_text

        # Preprocess the tweet to get terms
        terms = build_terms(tweet_text)

        # Create the current page index for the tweet
        current_page_index = {}

        for position, term in enumerate(terms):
            if term in current_page_index:
                # If term already exists, append the position
                current_page_index[term][1].append(position)
            else:
                # If term is new, create an entry with the doc_id and position list
                current_page_index[term] = [doc_id, [position]]

        # Merge the current page index with the main index
        for term, posting in current_page_index.items():
            index[term].append(posting)

    return index, title_index

# Example usage with tweet_info_df
lines = list(zip(tweet_info_df['Document ID'], tweet_info_df['Tweet']))

# Create the index and title mapping
inverted_index, title_index = create_index(lines)
```

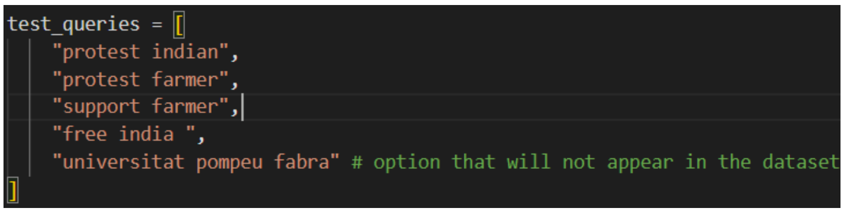
The inverted index uses a dictionary, where each term ID is a key, and the value is a list of postings. Each posting includes the document ID and the positions list, which corresponds where the term appears in each document.



```
for term, postings in list(inverted_index.items())[:10]:
    print(f"{term}: {postings}")

world: [['doc_0', [0]], ['doc_38', [12]], ['doc_52', [7]],
progress: [['doc_0', [1]], ['doc_100', [5]], ['doc_2738', [
indian: [['doc_0', [2]], ['doc_17', [9]], ['doc_18', [8]],
polic: [['doc_0', [3]], ['doc_56', [21]], ['doc_60', [13]],
```

2.2. Propose test queries

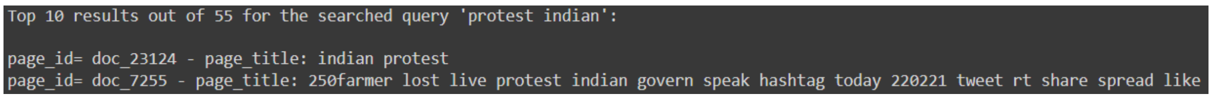
For evaluating our search engine, we defined five different test queries:



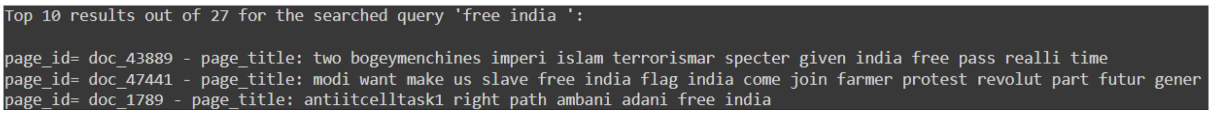
```
test_queries = [
    "protest indian",
    "protest farmer",
    "support farmer",
    "free india ",
    "universitat pompeu fabra" # option that will not appear in the dataset
]
```

The search function begins by preprocessing each query, ensuring that they match with the terms stored in the inverted index. Then we want to find the documents that contain all the query terms, implementing a conjunctive search where only documents with all the terms are retrieved.

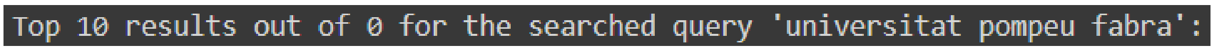
Moreover, the function evaluates if the terms appear consecutively in each matched document, doing this we search the exact results that user wants, for queries expecting specific phrases. Finally, the function returns the top 10 results for each query. Some results are:



```
Top 10 results out of 55 for the searched query 'protest indian':

page_id= doc_23124 - page_title: indian protest
page_id= doc_7255 - page_title: 250farmer lost live protest indian govern speak hashtag today 220221 tweet rt share spread like
```



```
Top 10 results out of 27 for the searched query 'free india ':

page_id= doc_43889 - page_title: two bogeymenchines imperi islam terrorismar specter given india free pass realli time
page_id= doc_47441 - page_title: modi want make us slave free india flag india come join farmer protest revolut part futur gener
page_id= doc_1789 - page_title: antiitcelltask1 right path ambani adani free india
```



```
Top 10 results out of 0 for the searched query 'universitat pompeu fabra':
```

Clàudia Morales - u199906
Roger Sola - u199780
Queralt Zamora - u199903
G102-2

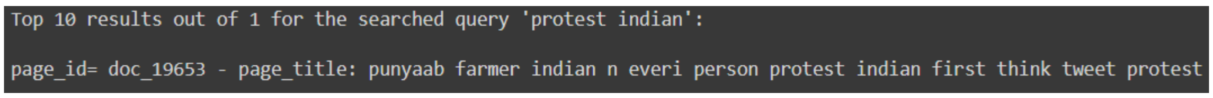
2.3. Rank your results

To rank documents we implemented a TF-IDF (Term Frequency-Inverse Document Frequency). This method involves two steps, create the index with term weight and rank the documents based on these weights.

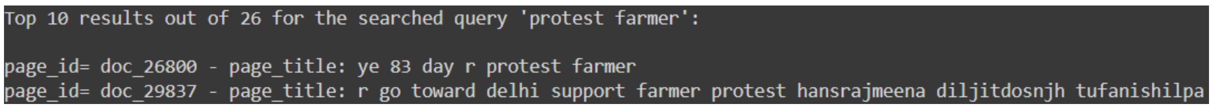
For the first step, we computed term frequencies for each document and document frequencies for each term with a double-pass indexing structure for each document. Then, we normalized the frequencies and we calculated the inverse document frequencies doing the logarithmic ratio of the total documents to the term's document frequency. With this process we get the weight of each term in each document.

For the second step, we applied a cosine similarity measure. Each query term's weight in a document is calculated by multiplying its normalized TF with the IDF value. Using the dot product between the query vector and document vectors, the algorithm computes similarity scores for each document.

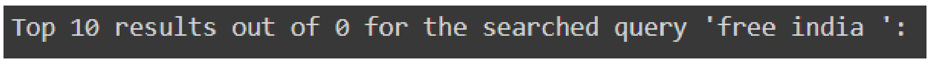
Finally, we stored the results by score, higher score means better match with query. Some results are:



```
Top 10 results out of 1 for the searched query 'protest indian':

page_id= doc_19653 - page_title: punyaab farmer indian n everi person protest indian first think tweet protest
```



```
Top 10 results out of 26 for the searched query 'protest farmer':

page_id= doc_26800 - page_title: ye 83 day r protest farmer
page_id= doc_29837 - page_title: r go toward delhi support farmer protest hansrajmeena diljitdosnjh tufanishilpa
```

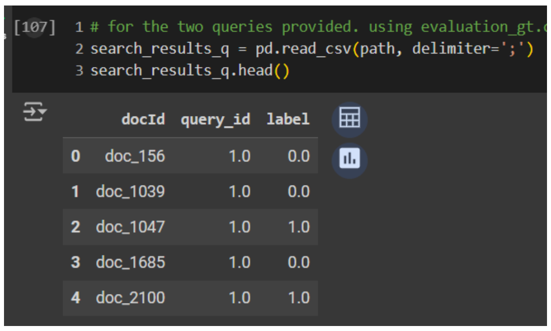


```
Top 10 results out of 0 for the searched query 'free india ':
```

3. Evaluation

3.1. Preparing the baseline

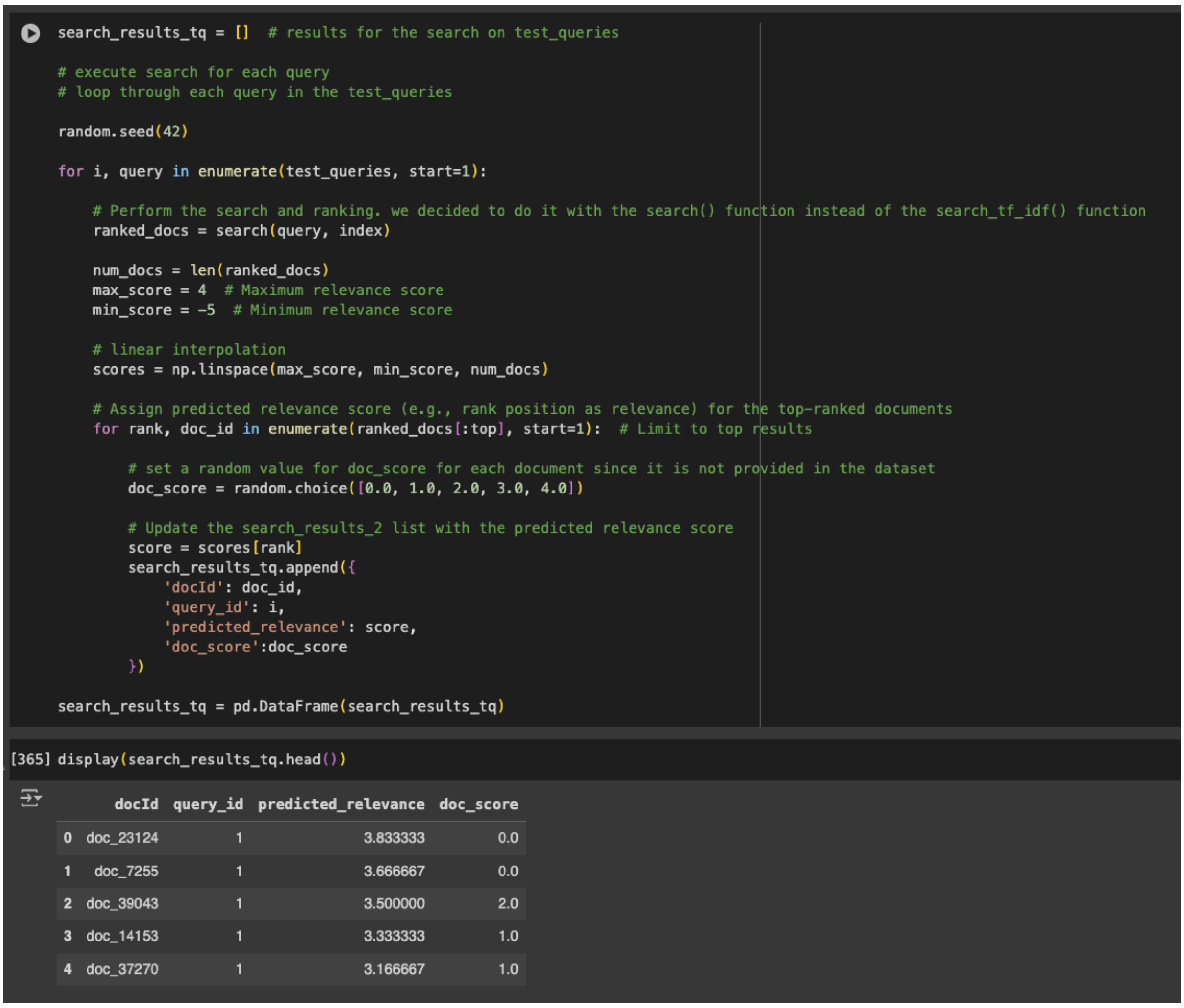
To evaluate the ranking accuracy of our search engine, we compute some performance metrics using the two provided queries, "people's rights?" and "Indian government?". To do this we are going to use a subset of the dataset, with information about relevance for each document provided in evaluation_gt.csv. The relevance follows a binary distribution, 1 for relevant, 0 for non-relevant. This is going to help us in our ground truth for performance evaluation.



```
[107]
# for the two queries provided. using evaluation_gt.
search_results_q = pd.read_csv(path, delimiter=';')
search_results_q.head()
```

	docId	query_id	label
0	doc_156	1.0	0.0
1	doc_1039	1.0	0.0
2	doc_1047	1.0	1.0
3	doc_1685	1.0	0.0
4	doc_2100	1.0	1.0

To do the evaluation we compute metrics such as Precision@K, Mean Average Precision (MAP), and Mean Reciprocal Rank (MRR), which measure how well the search engine identifies relevant documents. To assign predicted relevance scores, we perform a search for each query and rank the returned documents randomly, with a random seed. Finally, we apply a linear interpolation to distribute relevance scores from a maximum relevance of 4 to a minimum of -5 for the top documents, which then are mapped back to the evaluation dataset for computing the metrics. And then we use a similar procedure for our test_queries:



```
search_results_tq = []  # results for the search on test_queries

# execute search for each query
# loop through each query in the test_queries

random.seed(42)

for i, query in enumerate(test_queries, start=1):

    # Perform the search and ranking. we decided to do it with the search() function instead of the search_tf_idf() function
    ranked_docs = search(query, index)

    num_docs = len(ranked_docs)
    max_score = 4  # Maximum relevance score
    min_score = -5  # Minimum relevance score

    # linear interpolation
    scores = np.linspace(max_score, min_score, num_docs)

    # Assign predicted relevance score (e.g., rank position as relevance) for the top-ranked documents
    for rank, doc_id in enumerate(ranked_docs[:top], start=1):  # Limit to top results

        # set a random value for doc_score for each document since it is not provided in the dataset
        doc_score = random.choice([0.0, 1.0, 2.0, 3.0, 4.0])

        # Update the search_results_2 list with the predicted relevance score
        score = scores[rank]
        search_results_tq.append({
            'docId': doc_id,
            'query_id': i,
            'predicted_relevance': score,
            'doc_score':doc_score
        })

search_results_tq = pd.DataFrame(search_results_tq)
```

```
[365] display(search_results_tq.head())
```

```
       docId  query_id  predicted_relevance  doc_score
0  doc_23124         1             3.833333        0.0
1   doc_7255         1             3.666667        0.0
2  doc_39043         1             3.500000        2.0
3  doc_14153         1             3.333333        1.0
4  doc_37270         1             3.166667        1.0
```

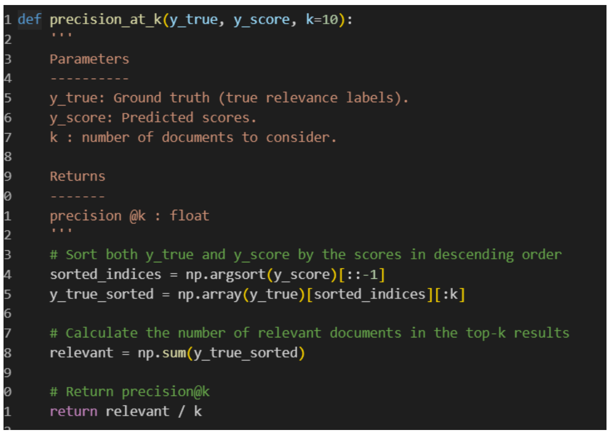
3.2. Evaluating

In this section, we computed the performance of the implemented algorithm through different evaluation techniques, focusing on metrics such as Precision@K (P@K), Recall@K (R@K), Average Precision@K, F1-Score@K, Mean Average Precision (MAP), Mean Reciprocal Rank (MRR), and Normalized Discounted Cumulative Gain (NDCG).

For the evaluation of the queries, we calculated the matrices with values of k = 3, k = 5, and k = 10. This allows us to observe the behavior of the metrics at different levels of depth. Below you can find an explanation of each metric.

3.2.1 Precision@K

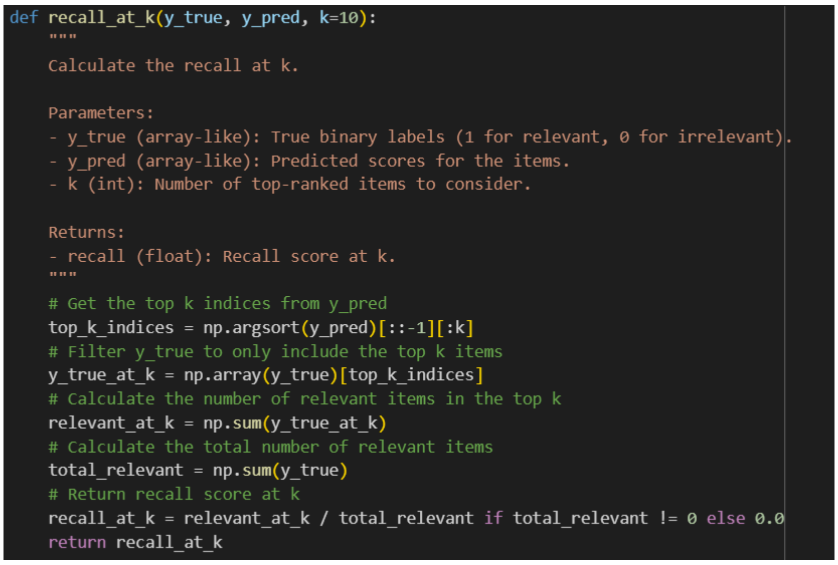
This metric measures the precision at the top K results for each query, evaluating how many relevant documents appear in the highest positions on the results list. High Precision@K results indicate that the system retrieves relevant documents in the top positions.



```python
def precision_at_k(y_true, y_score, k=10):
    '''
    Parameters
    ----------
    y_true: Ground truth (true relevance labels).
    y_score: Predicted scores.
    k : number of documents to consider.

    Returns
    -------
    precision @k : float
    '''
    # Sort both y_true and y_score by the scores in descending order
    sorted_indices = np.argsort(y_score)[::-1]
    y_true_sorted = np.array(y_true)[sorted_indices][:k]

    # Calculate the number of relevant documents in the top-k results
    relevant = np.sum(y_true_sorted)

    # Return precision@k
    return relevant / k
```

3.2.2 Recall@K

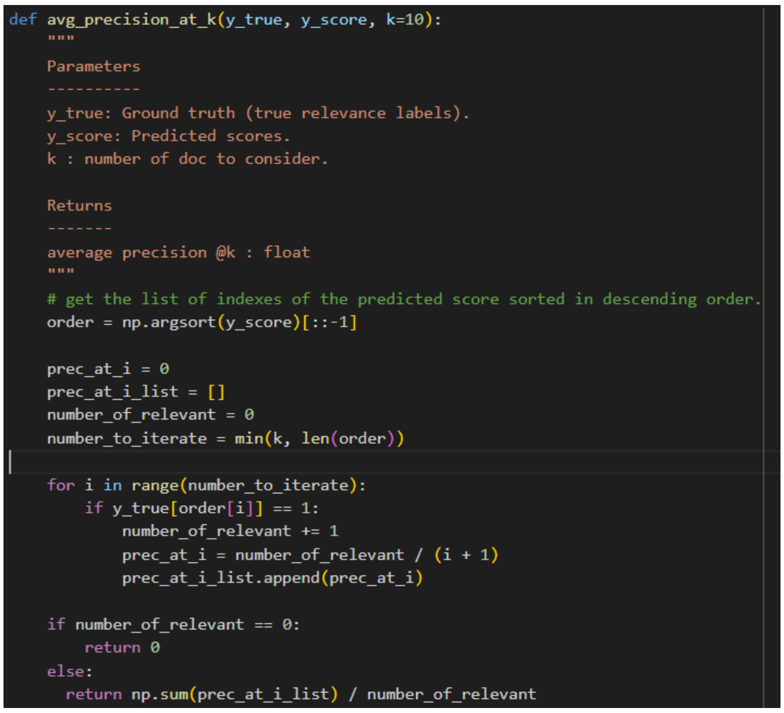
Recall@K represents the fraction of relevant documents among the top K results. A high Recall@K implies that most relevant documents are being captured in the upper part of the list, which is crucial for queries with a high number of relevant documents.



```python
def recall_at_k(y_true, y_pred, k=10):
    """
    Calculate the recall at k.

    Parameters:
    - y_true (array-like): True binary labels (1 for relevant, 0 for irrelevant).
    - y_pred (array-like): Predicted scores for the items.
    - k (int): Number of top-ranked items to consider.

    Returns:
    - recall (float): Recall score at k.
    """
    # Get the top k indices from y_pred
    top_k_indices = np.argsort(y_pred)[::-1][:k]
    # Filter y_true to only include the top k items
    y_true_at_k = np.array(y_true)[top_k_indices]
    # Calculate the number of relevant items in the top k
    relevant_at_k = np.sum(y_true_at_k)
    # Calculate the total number of relevant items
    total_relevant = np.sum(y_true)
    # Return recall score at k
    recall_at_k = relevant_at_k / total_relevant if total_relevant != 0 else 0.0
    return recall_at_k
```

3.2.3 Average Precision@K

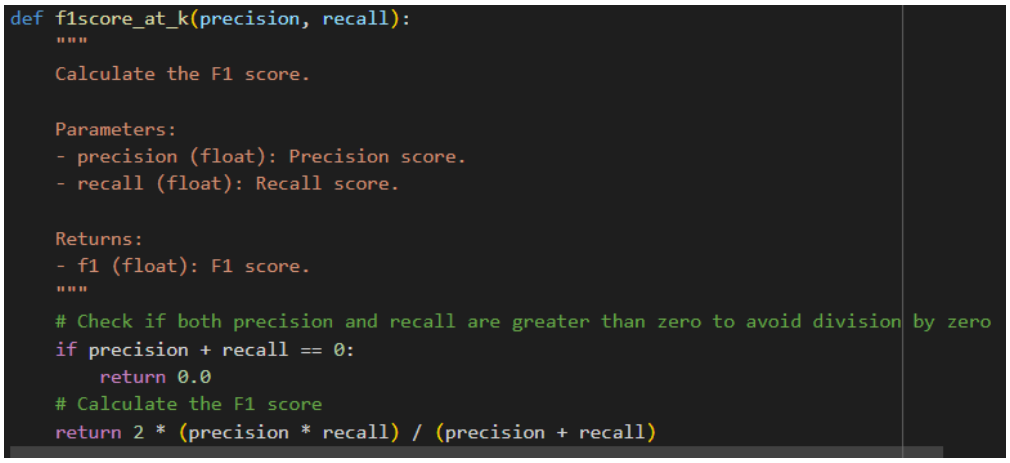
This metric provides the average precision calculated at each position in the list up to K. The mean of these precisions indicates the concentration of relevant documents throughout the ranked list, promoting a uniform distribution of relevance.



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

    Returns
    -------
    average precision @k : float
    """
    # get the list of indexes of the predicted score sorted in descending order.
    order = np.argsort(y_score)[::-1]

    prec_at_i = 0
    prec_at_i_list = []
    number_of_relevant = 0
    number_to_iterate = min(k, len(order))

    for i in range(number_to_iterate):
        if y_true[order[i]] == 1:
            number_of_relevant += 1
            prec_at_i = number_of_relevant / (i + 1)
            prec_at_i_list.append(prec_at_i)

    if number_of_relevant == 0:
        return 0
    else:
      return np.sum(prec_at_i_list) / number_of_relevant
```

3.2.4 F1-Score@K

The F1-Score combines precision and recall into a single metric, providing a balance between both. A high F1-Score at low K values indicates that the system effectively returns a high percentage of relevant documents in the top positions.

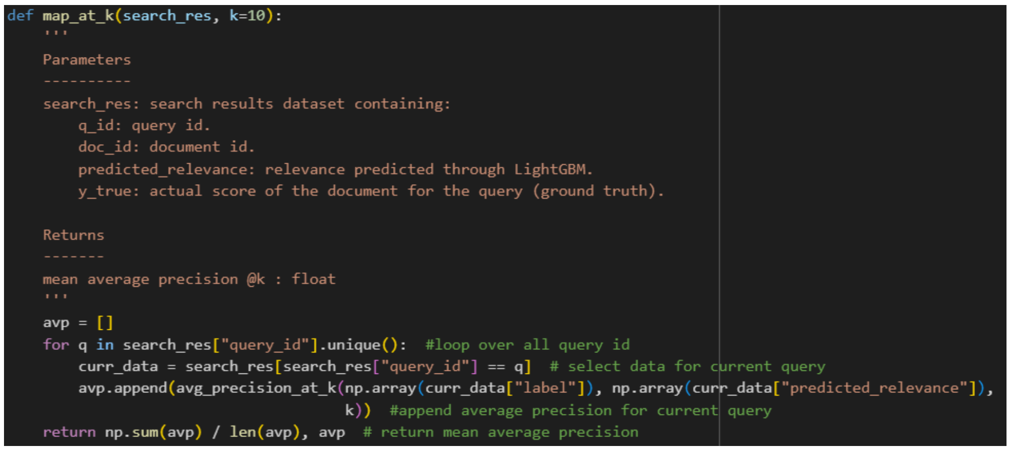


```python
def f1score_at_k(precision, recall):
    """
    Calculate the F1 score.

    Parameters:
    - precision (float): Precision score.
    - recall (float): Recall score.

    Returns:
    - f1 (float): F1 score.
    """
    # Check if both precision and recall are greater than zero to avoid division by zero
    if precision + recall == 0:
        return 0.0
    # Calculate the F1 score
    return 2 * (precision * recall) / (precision + recall)
```

Clàudia Morales - u199906
Roger Sola - u199780
Queralt Zamora - u199903
G102-2

3.2.5 Mean Average Precision

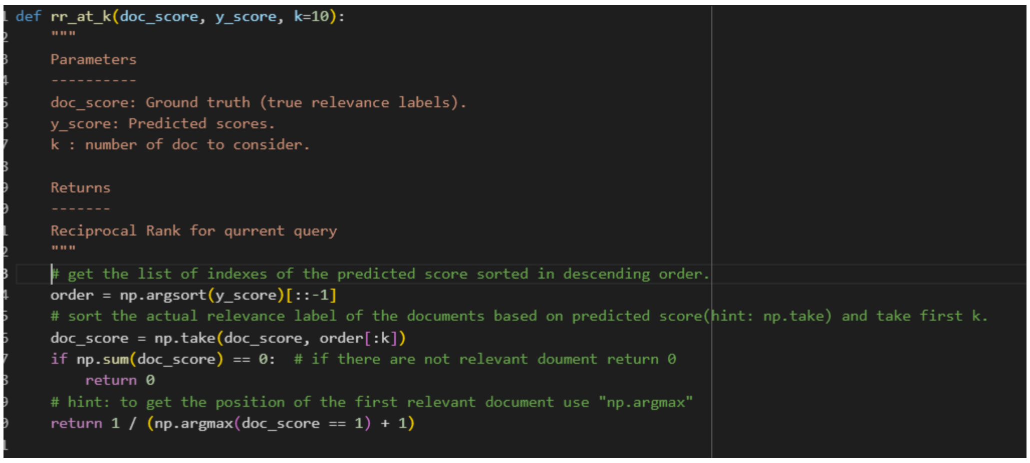
MAP is the mean precision calculated across several queries, giving an overall view of the system's effectiveness. A high MAP value means that, on average, the system consistently returns relevant documents across all evaluated queries.



```python
def map_at_k(search_res, k=10):
    '''
    Parameters
    ----------
    search_res: search results dataset containing:
        q_id: query id.
        doc_id: document id.
        predicted_relevance: relevance predicted through LightGBM.
        y_true: 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 id
        curr_data = search_res[search_res["query_id"] == q]  # select data for current query
        avp.append(avg_precision_at_k(np.array(curr_data["label"]), np.array(curr_data["predicted_relevance"]),
                                      k))  #append average precision for current query
    return np.sum(avp) / len(avp), avp  # return mean average precision
```

3.2.6 Mean Reciprocal Rank

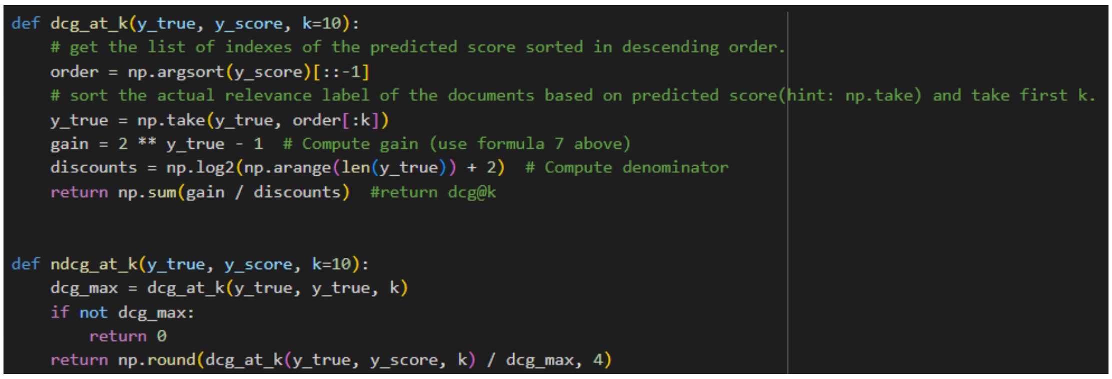
MRR measures the position of the first relevant document in the results list. A high MRR indicates that relevant documents often appear in the first positions. In our case, it has been calculated for k = 3, k = 5, and k = 10.



```python
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
    """
    # get the list of indexes of the predicted score sorted in descending order.
    order = np.argsort(y_score)[::-1]
    # sort the actual relevance label of the documents based on predicted score(hint: np.take) and take first k.
    doc_score = np.take(doc_score, order[:k])
    if np.sum(doc_score) == 0:  # if there are not relevant doument return 0
        return 0
    # hint: to get the position of the first relevant document use "np.argmax"
    return 1 / (np.argmax(doc_score == 1) + 1)
```

3.2.7 Normalized Discounted Cumulative Gain

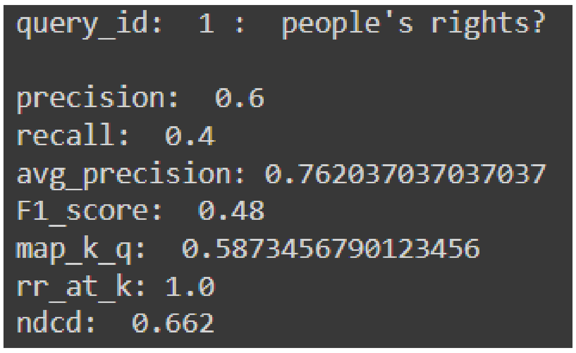
NDCG considers the position of each relevant document in the list, applying a logarithmic discount for documents in lower positions. It is especially useful for assessing whether relevant documents appear in higher positions, maximizing benefit for the user. We calculated NDCG@10 for each query and the mean across all.



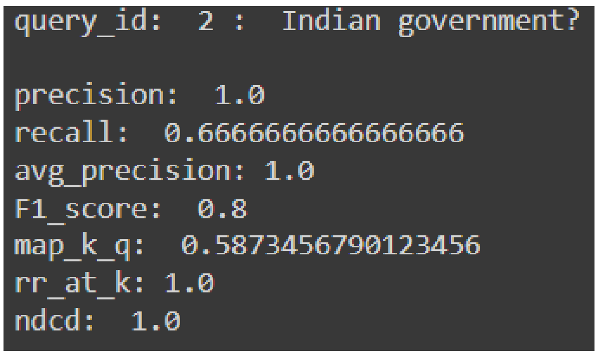
```python
def dcg_at_k(y_true, y_score, k=10):
    # get the list of indexes of the predicted score sorted in descending order.
    order = np.argsort(y_score)[::-1]
    # sort the actual relevance label of the documents based on predicted score(hint: np.take) and take first k.
    y_true = np.take(y_true, order[:k])
    gain = 2 ** y_true - 1  # Compute gain (use formula 7 above)
    discounts = np.log2(np.arange(len(y_true)) + 2)  # Compute denominator
    return np.sum(gain / discounts)  #return dcg@k


def ndcg_at_k(y_true, y_score, k=10):
    dcg_max = dcg_at_k(y_true, y_true, k)
    if not dcg_max:
        return 0
    return np.round(dcg_at_k(y_true, y_score, k) / dcg_max, 4)
```

All the query results from the different evaluation techniques:

for the provided queries:



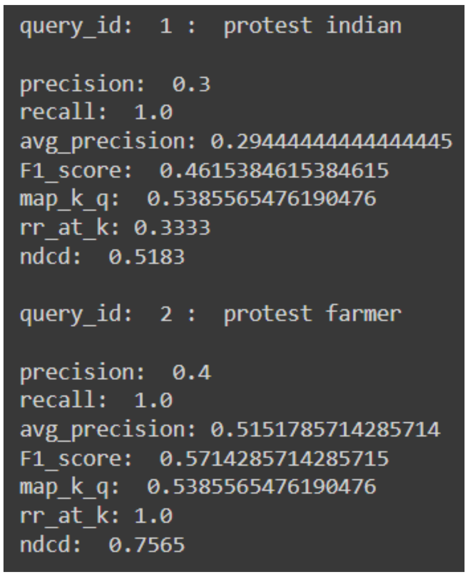
```
query_id:  1 :  people's rights?

precision:  0.6
recall:  0.4
avg_precision: 0.762037037037037
F1_score:  0.48
map_k_q:  0.5873456790123456
rr_at_k: 1.0
ndcd:  0.662
```



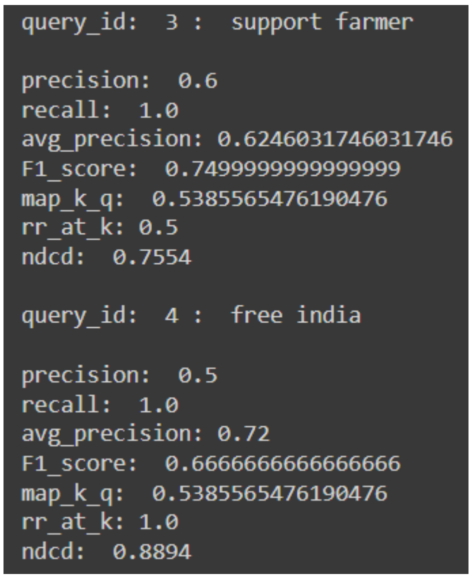
```
query_id:  2 :  Indian government?

precision:  1.0
recall:  0.6666666666666666
avg_precision: 1.0
F1_score:  0.8
map_k_q:  0.5873456790123456
rr_at_k: 1.0
ndcd:  1.0
```

Clàudia Morales - u199906
Roger Sola - u199780
Queralt Zamora - u199903
G102-2

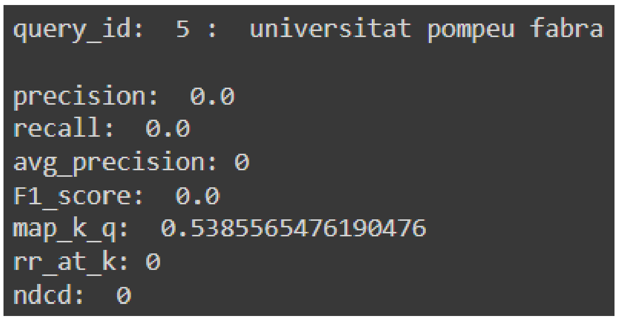
for our test_queries:



```
query_id:  1 :  protest indian

precision:  0.3
recall:  1.0
avg_precision: 0.29444444444444445
F1_score:  0.46153846153846156
map_k_q:  0.5385565476190476
rr_at_k: 0.3333
ndcd:  0.5183

query_id:  2 :  protest farmer

precision:  0.4
recall:  1.0
avg_precision: 0.5151785714285714
F1_score:  0.5714285714285715
map_k_q:  0.5385565476190476
rr_at_k: 1.0
ndcd:  0.7565
```

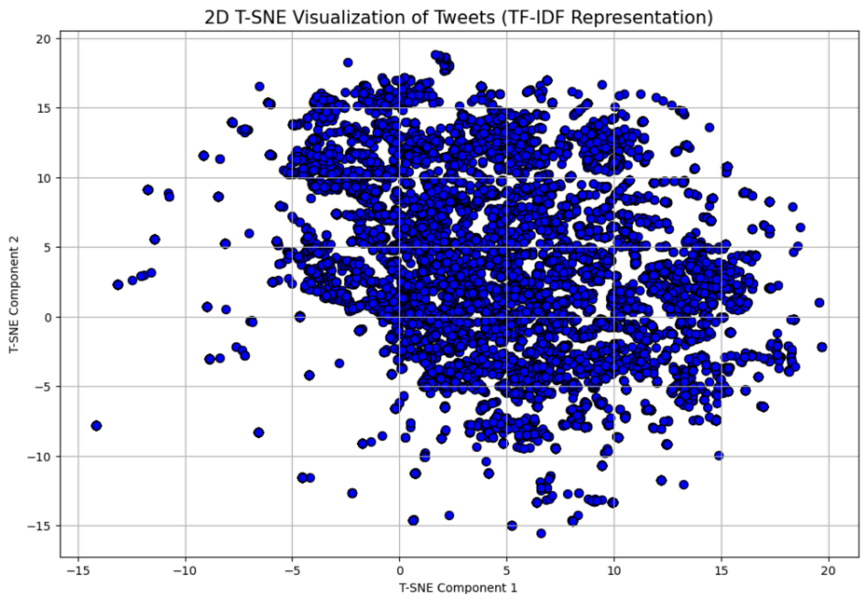


```
query_id:  3 :  support farmer

precision:  0.6
recall:  1.0
avg_precision: 0.6246031746031746
F1_score:  0.7499999999999999
map_k_q:  0.5385565476190476
rr_at_k: 0.5
ndcd:  0.7554

query_id:  4 :  free india

precision:  0.5
recall:  1.0
avg_precision: 0.72
F1_score:  0.6666666666666666
map_k_q:  0.5385565476190476
rr_at_k: 1.0
ndcd:  0.8894
```



```
query_id:  5 :  universitat pompeu fabra

precision:  0.0
recall:  0.0
avg_precision: 0
F1_score:  0.0
map_k_q:  0.5385565476190476
rr_at_k: 0
ndcd:  0
```

3.3. Final representation



In the previous plot we can see a graphic which represents tweets based on their TF-IDF vectors, each dot represents an individual tweet positioned according to its word frequency and relevance. Since we can only observe one cluster, this could mean that the tweets may not contain strongly differentiated content based on their TF-IDF scores, or it could be due to T-SNE's parameter settings. The majority of the points are concentrated near the center, indicating that many tweets share similar word distributions. This makes sense when you consider that they are all about the same topic: the farm protests in India.

4. Conclusions

This project involved building and evaluating an information retrieval system on a dataset of tweets. In the indexing phase, we constructed an inverted index and ranked results using TF-IDF, defining five test queries to assess retrieval effectiveness. In the evaluation phase, we applied various metrics like Precision@K, Recall@K, MAP, MRR, and NDCG, each revealing different strengths and limitations in capturing relevant tweets. Additionally, we visualized tweet clusters with T-SNE, showing dense groupings that reflected similarity. Overall, the project demonstrated the capabilities and challenges of using TF-IDF for short-text retrieval and highlighted potential areas for refinement in future implementations.