Project Report Part 2

The steps required to execute the Python Jupyter Notebook are detailed in the README.md file. To improve the user experience and reduce the execution time, in the part of the project, we put the preprocessed data in a CVS file, which allows the user to skip the preprocess stage and execute this part of the project directly.

Indexing

To build the inverted index, as the hint suggested, we have used the practical labs we have been doing, and implemented a function called ‘create_index’ to build an inverted index. We have taken the function given in Lab 1, and adapted it in order to work with this dataset, as we have already preprocessed the tweets. The function maps each term of each document to a list that includes the document ID and the positions of the term in the document. To do that, we take each term in the document and we build a temporary dictionary, ‘current_page_index’, that stores what we have mentioned before, the document and position data. Then, we merge this index with the main index ‘index’. In this way, we make sure that each term does not have any duplicated data.

The ‘create_index’ function it is the back bone to implement the search functionality in the project, as it allows to look up for terms and retrieve them efficiently, or in another words, it makes easier to access directly to documents that contain all the terms in a query.

We have also implemented the same exact function of Lab 1 to preprocess each query that we use for testing our index, to ensure consistency. It lowercases terms, splits the query into terms, removes stopwords and applies stemming to simplify terms to their root forms. In this way, we improve retrieval performance, as we are grouping similar terms under a common root.

Once we have built the inverted index, we have taken the function ‘search’ and mofy it again in order to work with the dataset we have. The function ‘search’ retrieves all documents that match all terms in a query, which is called a conjuctive search’, a type of search that requires all the terms in the query to be present in the document for it to be considered a match. It first preprocesses the query using the ‘build_terms’ function, then it retrieves the documents IDs for the term, and finally, it intersects the set of document IDs for each term in the query, so we ensure that we only return the documents that contain all the documents that contain all the query terms.

To simulate the query searches, we got the top 10 terms in the dataset with the function ‘get_top_terms’, which identifies the top n terms by frequency that is the most commonly discussed topics. It gets the content of the tweets and puts it altogether to then split it into terms and count the frequency of all of them. It returns the 10 most common terms. After retrieving these terms, we have chosen as the testing queries, the following:

- “India protest”: This query aligns with the high frequency of terms like ‘india’ (7724 occurrences) and ‘protest’ (4787 occurrences). It targets the significant discussion around protests in India, capturing a broad yet significant topic within your dataset. This is important for understanding general sentiments or events related to protests in the region.
- “support farmers”: Given the prevalence of terms such as ‘farmersprotest’ (50272 occurrences) and ‘support’ (6004 occurrences), this query is highly relevant. It specifically

addresses the widespread discourse on supporting farmers, likely related to agricultural policies or farmer welfare, which looks to be a prominent issue in the data.

- "Modi shame": This query is crafted around the high occurrence of 'modi' (3113 occurrences). Including a sentiment or descriptive term like 'shame' might target specific discussions or criticisms related to policies or actions associated with Modi, the Prime Minister of India, offering insights into public sentiment regarding his administration.
- "BJP party": With 'bjp' being mentioned 2649 times, focusing on the political party directly allows for an analysis of content specifically related to the Bharatiya Janata Party. This could reveal discussions on political activities, policies, or public opinions directly connected to the party's actions and governance.
- "human rights violated": Although 'right' appears 3594 times, coupling it with 'violated' expands the context to human rights issues. This query is intended to explore discussions or reports on human rights violations, a topic of critical importance that may encompass various aspects of social and political discourse.

Then, using the function 'simulate_serach', we performed a test search with the queries we predefined to evaluate the search engine we have implemented before. So, the function, given the list of queries and the index we have created, uses the 'search' function to search each query term and get the documents the query appears in. For a more clear visualization, we only display the top 10 results.

Then we added another layer to search engine by implementing the TF-IDF algorithm to rank our search results, which allows documents with higher relevance (based on term importance) to rank higher in results. The function 'create_index_tfidf' builds an index weighted by TF-IDF that is it calculates the term frequency (TF) and the document frequency (DF) for each term of the dataset, and then it computes the inverse document frequency (IDF) to scale the importance of terms across the corpus.

As well as with the 'create_index' function, we build a temporary index that stores the position of each term in the document and the document ID associated with each term. Then, we compute the norm of the document to normalize term frequencies. Once the norm is calculated, we compute the term frequency and the document frequency. Finally, we merge the temporary index with the main index and compute the inverse document frequency for each term of the index.

In order to use the TF-IDF algorithm function, we created a function called 'rank_documents' that scores and ranks documents based on their TF-IDF similarity to the query. The function computes the TF-IDF vector for the query and each document and then it ranks the documents using cosine similarity to the query. Using cosine similarity helps measure the angle between the query and the document vectors, which effectively captures the relevance by how well terms align across documents and the query.

As we did with the inverted index, we implemented another search function 'search_tf_idf' that integrates the search and ranking components. It retrieves all the documents that contain the query terms (conjuctive match) and ranks them based on TF-IDF similarity. By using this function, we make sure that both relevance and precision are fulfilled, as we return the most relevant results first. To test it, we have used a single word and to check that it was working correctly, we have also printed the scores of each document, so we could see that the orderings were correct.

Evaluation

During the evaluation phase, we have used several metrics to measure the effectiveness of our search engine and assess both the relevance and the ranking quality of the results. We have applied metrics such as Precision@K, Recall@K, Average Precision, F1-Score, Mean Average Precision (MAP), Mean Reciprocal Rank (MRR), and Normalized Discounted Cumulative Gain (NDCG). Moreover, we have implemented Word2Vec embeddings and T-SNE visualization to observe document similarity and clustering.

As a first step, we have prepared the data we used in the evaluation process. First, we merged the evaluation data file with the tweets dataset using the document ID as a key, so we make sure that evaluation labels are together with the document content. Then, we categorized the documents by relevancy, either they are relevant, marked with a 1, or non-relevant, marked with a 0, for each predefined query. With the queries we have been given, we perform a TF-IDF weighted search for 'people's rights' and 'Indian government', which provided a list of ranked documents, using the function we implemented in the indexing part 'search_tf_idf'. Finally, we organized our five custom queries we defined earlier with relevance labels for a more personalized evaluation. Each query text and document set is added to dictionaries, so it is easier to retrieve relevant documents and results during each evaluation.

After defining the function for each evaluation method, we have tested them by defining for each query the following K values (5, 10, 15, 20, 50, 100, 150, total number of results). Then, we retrieved the relevant documents and the system's results, it iterates through each K value, calculating each selected metric.

Precision@K (P@K)

'precision_at_k' computes the precision for the top K documents, in the search results, which is the proportion of the retrieved documents in the top K that are relevant. This metric measures how well the search engine ranks relevant documents within the top positions.

We start by extracting the top K documents from the results list. Then , we iterate over each document we extracted before and we check if the document ID is in the ground truth. The precision is calculated as the ratio of relevant documents in he top K results to the total number of documents K being considered, if K is greater than 0, otherwise it is 0.

Recall@K (R@K)

'recall_at_k' computes the recall for the top K results, which is defined as the fraction of relevant documents within the top K retrieved documents out of the total relevant documents. In this way we can asses how well does the search engine find relevant documents overall, which is important in sistuations where missing relevant results can be a very big mistake.

In the function, we first check if K is larger than the number of documents in results, then, if so, it is set to the length of results, so we only consider available results. We iterate through the top K results and count how many of the documents are in the ground_truth. The recall is then calculated as the proportion of relevant documents retrieved within the top K results to the total number of relevant documents.

Average Precision@K (P@K)

‘average_precision_at_k’ computes the averages precision at K, which is basically taking the precision of different positions up to K and averages them, so it gives them a higher weight to rank correctly documents. With this method we analyze how does the search engine keep relevance across multiple rankings.

We iterate through the top K results, and for each result, it checks if the document ID is in the set of the ground truth. If is it, then it means that the document is relevant and we increment the number of relevant retrieved documents by 1, and we compute the precision at position i. Finally, we compute the average precision as the division between the cumulative precision and the total number of relevant documents.

F1-Score@K

‘f1_score_at_k’ is a function that calculates the F1-score that is a harmonic mean of precision and recall at K, which offers a balanced measure that captures false positives and false negatives.

Again, we iterate through each document extracted from the top K results, and if the document ID is in the ground truth set, we increment the number of relevant retrieved documents by 1. Then, we compute the precision by dividing the number of relevant retrieved documents by the number of documents extracted from the top K results. We calculate the recall by dividing the number of relevant retrieved documents by the number of relevant documents for the query. Fianlly, the F1 score is computed as twice the product of precision and recall, divided by the sum of precision and recall.

Mean Average Precision (MAP)

‘mean_average_precision_at_k’ averages the average precision for multiple queries, which provides another precision measure among all queries. With this metric we evaluate the effectiveness across multiple queries.

We start by iterating over pairs of the ground truth and results of a specific query, and for each query, we calculate the average precision at K using the previous function we have defined ‘average_precision_at_k’, which returns a score that represents how well the search engine has ranked relevant documents within the top K results for the current query. Then, we compute the mean average precision by dividing the scores we got earlier by the number of queries.

Mean Reciprocal Rank (MRR)

‘mean_reciprocal_rank’ computes the average reciprocal rank for the first relevant documents across multiple queries, which measures how early relevant documents appear in the results. Using MRR to evaluate the search engine shows how well it anks relevant documents near the top of results.

In the function , we iterate again over pairs of the ground truth and results of a specific query and we create another loop inside, where we iterate throught the results for the current query, so if the document ID of the current result is in the ground truth set, then the document is relevant, so we set ‘reciprical_rank’ to 1/’rank’ where rank is the position of the first relevant document in the results. Then, we compute the mean reciprocal rank as the average of all values of the reciprocal ranks we computed before.

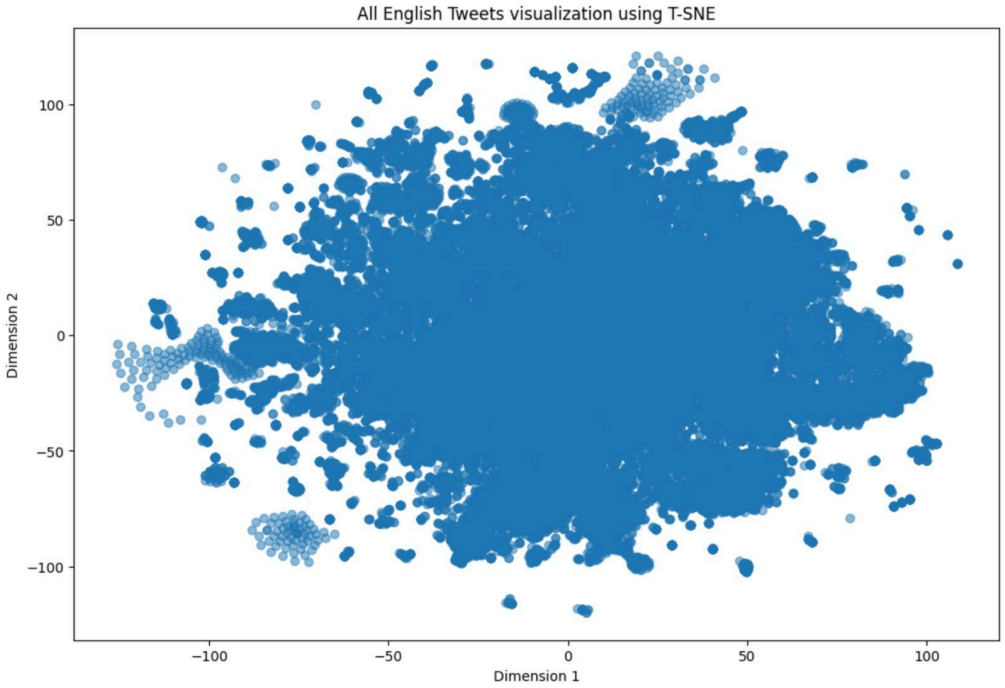
Normalized Discounted Cumulative Gain (NDCG)

'ndcg_at_k' evaluates ranking quality by assigning higher scores to relevant documents appearing near the top results. Using NDCG is crucial for ranking-based evaluation, rewarding systems that rank relevant documents higher.

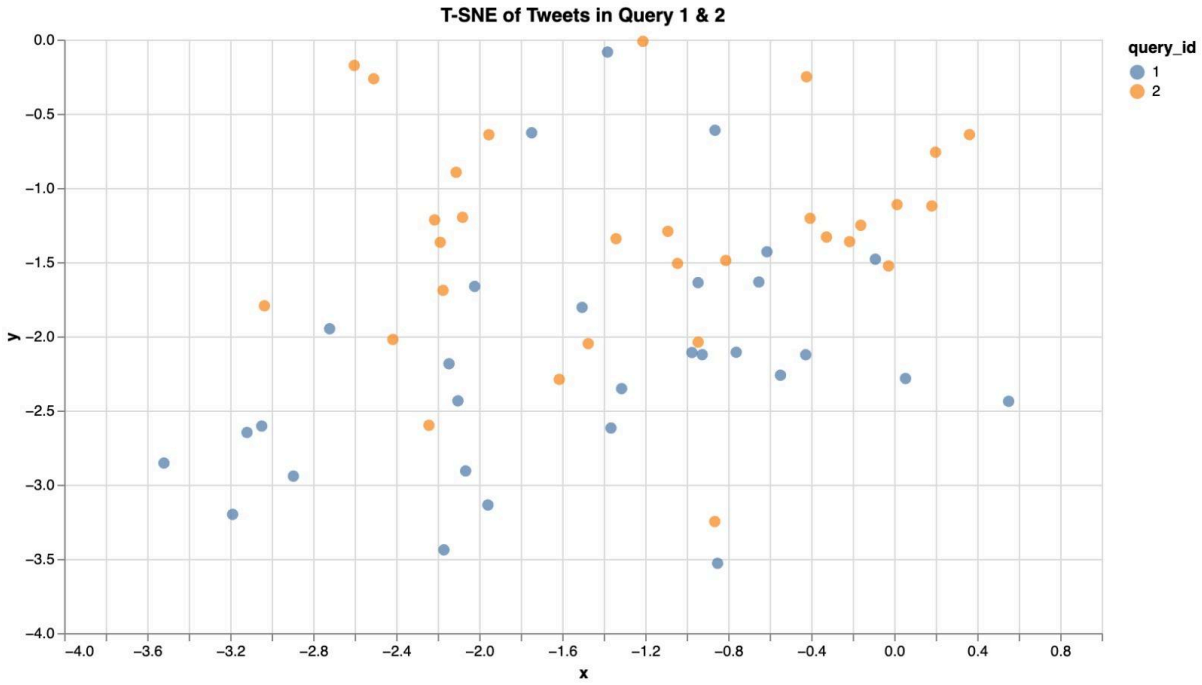
For this evaluation metric, we have defined an auxiliar function 'dcg_at_k' to compute the discounted cumulative gain. What the function is doing is computing the DCG either using the standard method or an alternative method, depending if the variable 'method' is 1 or 2, which by default is 1. Then, in 'ndcg_at_k', we are computing the normalized discounted cumulative gain by, first, calculating the binary relevance for the actual results, where each document in the top K results is assigned a relevance score of 1 if it is in the ground truth set or 0, otherwise. We call the previous function 'dcg_at_k' to compute the DCG for the actual results based on their relevance scores and, we also compute the ideal DCG by sorting the relevance for the actual results in descending order to represent the best-case scenario for ranking and then we call again the 'dcg_at_k' function using the sorted relevance scores. Finally, we return the normalized DCG, which is the ratio of the actual DCG to the ideal DCG.

Vector Representation using T-distributed Stochastic Neighbor Embedding (T-SNE)

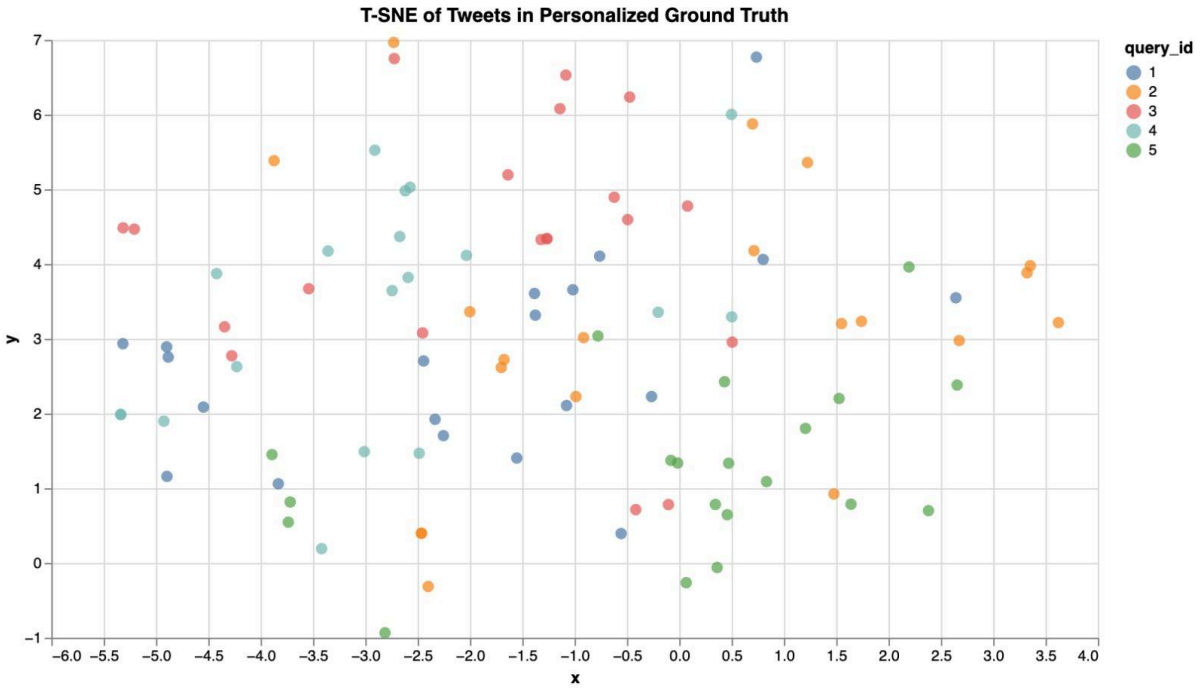
To represent the tweets in a two-dimensional scatter plot through the T-NSE algorithm we hav echosen word2vec to capture semantic relation ships between words. Using word2vec has allowed us to perform a deeper semantic analysis by creating vector representations of words that capture the semantic relationships based on the context, so words that appear in similar context will have similar embeddings.



The image above is the plot of all English tweets using T-SNE, and as we can observe there is a dense cluster of points and some outliers. The fact that there is a pretty dense cluster could indicate that most of the English tweets share common topics or language patterns. The outliers could represent some unique tweets that are significantly different from the majority of English tweets, asn they could be tweets that have specifics topics or personal opinions.



The above plot shows the T-SNE of tweets in queries 1 and 2 and we can see that there are 2 distinct clusters that overlap, probably indicating that the tweets in each query share some similarities but they are different. We can see a clear separation between the clusters suggesting that the tweets in both queries discuss different topics, but they overlap in some topics too as there is an overlap between the clusters. There are a few outliers, further away from the main clusters, which could be tweets that are less related to the main topic of the queries.



The last plot is the T-SNE of the tweets in the personalized ground truth and we can observe that there are several distinct clusters with slime overlapping between them. The clear separation between them

may indicate that the tweets in those clusters are discussing different topics, but them overlapping suggests that there are some common topics that the tweets are discussing. As well as before, there are some points that are further away from the main cluster, which could mean that there are some tweets that are less related to the main topics of the clusters.