CS 242: INFORMATION RETRIEVAL SYSTEMS

Twitter Trend Checker

Group-5

Team Members: -

- **Aditya Krrish Komarapalayam Singaravelu (862324414)**
- **Aparajita Satish Ramanathan (862324503)**
- **Harshitha Sarva (862323552)**
- **Malmurugan Sukumar (862246504)**
- **Sujith Kumar Sashikanth (862319856)**

Contributions:

All the work was brainstormed together, everyone involved learnt all the concepts involved and divided and conquered the work based in parts and module and formed sub teams to complete the project.

Aditya Krrish Komarapalayam Singaravelu (862324414)	**Part A:** Worked on Visual Studio to generate several data using various hash tags and on IntelliJ to write code for Lucene basic structures. **Part B:** Worked on Mongo-DB and Hadoop connections, Mongo-DB Flow, Data extraction for extra credit and report along with **Aparajita**.
Aparajita Satish Ramanathan (862324503)	**Part A:** Worked solely on Intellij and wrote code for complete indexing strategy for given Json file and wrote the project Report. **Part B:** Worked on MongoDB and Hadoop connections, Data extraction for exrta credit and report along with **Aditya**.
Harshitha Sarva (862323552)	**Part A:** Implemented Authentication and completely coded data extraction using Tweepy and worked on Tweepy strategy with **Sujith.** **Part B:** Developed code for front-end UI, fetching data from Mongo-DB and Sentiment Analysis for extra credit with **Sujith**.
Malmurugan Sukumar (862246504)	**Part A:** Worked on Intellij and wrote code for Lucene searching strategy from the index and chose the Lucene Analyzer for the project along with its architecture. **Part B:** Wrote code for Hadoop Inverted index generation by implementing Map-Reduce strategies and sentence pre-processing techniques using MRjobs python package.
Sujith Kumar Sashikanth (862319856)	**Part A:** Implemented and coded Tweepy architecture and worked on completely data conversion and extraction using Tweepy along with **Harshitha**. **Part B:** Developed code for front-end UI, fetching data from Mongo-DB and Sentiment Analysis for extra credit with **Harshitha**.

1. Overview:

"The Twitter trend checker" is a search engine targeted to retrieve tweets on trending topics and apply sentimental analysis on the tweets and find the positive or negative influence it is having at a particular location for the topic in discussion.

1.2 Overall Architecture:

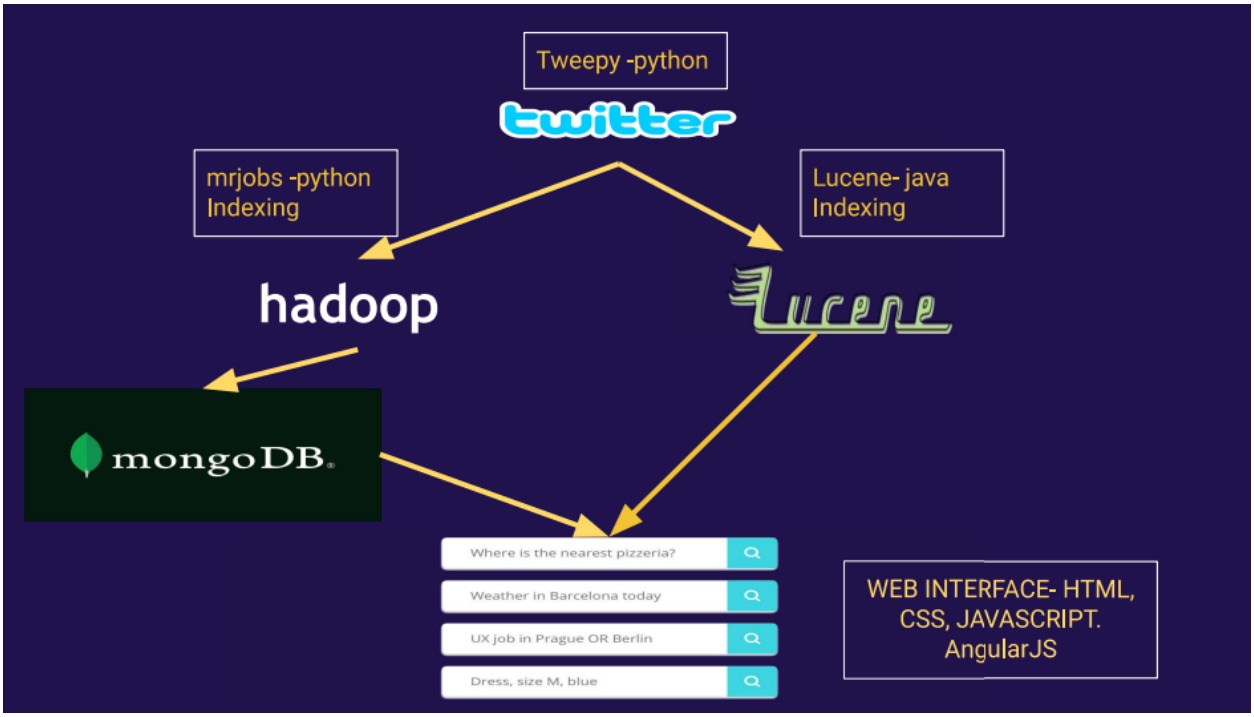

Fig. 1.1: Overall Architecture

The overall architecture of the Twitter Trend Checker project

1. Twitter Data Extraction.
2. Hadoop Inverted Index Creation.
3. Lucene Index Creation.
4. Mongo-DB setup and inverted index storage.
5. Front End UI and data fetch from Mongo-DB.

2 Twitter Data Extraction:

The twitter data was streamed and extracted using **Tweepy**. Tweepy is an open-source python-based twitter streaming API.

With Tweepy, we scraped more than a **gigabyte** of twitter data with all the essential fields (tweet text, hashtags, user-id, etc.) and stored it in a structured JSON format.

3. Hadoop:

Hadoop is an open-source tool that is used to process massive data, this project makes use of some features of Hadoop to create indexing for the same Twitter data mentioned in Part A alongside Lucene.

3.1 Hadoop Indexing:

To create the inverted index for every word in the twitter dataset, we use Hadoop Map-Reduce implemented on Mrjobs. Mrjobs is a python package that runs Hadoop Map-Reduce on python code.

There are two functions in Map-Reduce, the Mapper function processes the huge data and creates smaller chunks of processed data. The Reducer function shuffles and reduces data into desired output.

3.2 Mapper Function:

Input: Tweet data. Where each line from twitter dataset is passed as an input.

Output: (Key: word, Value: tweet)

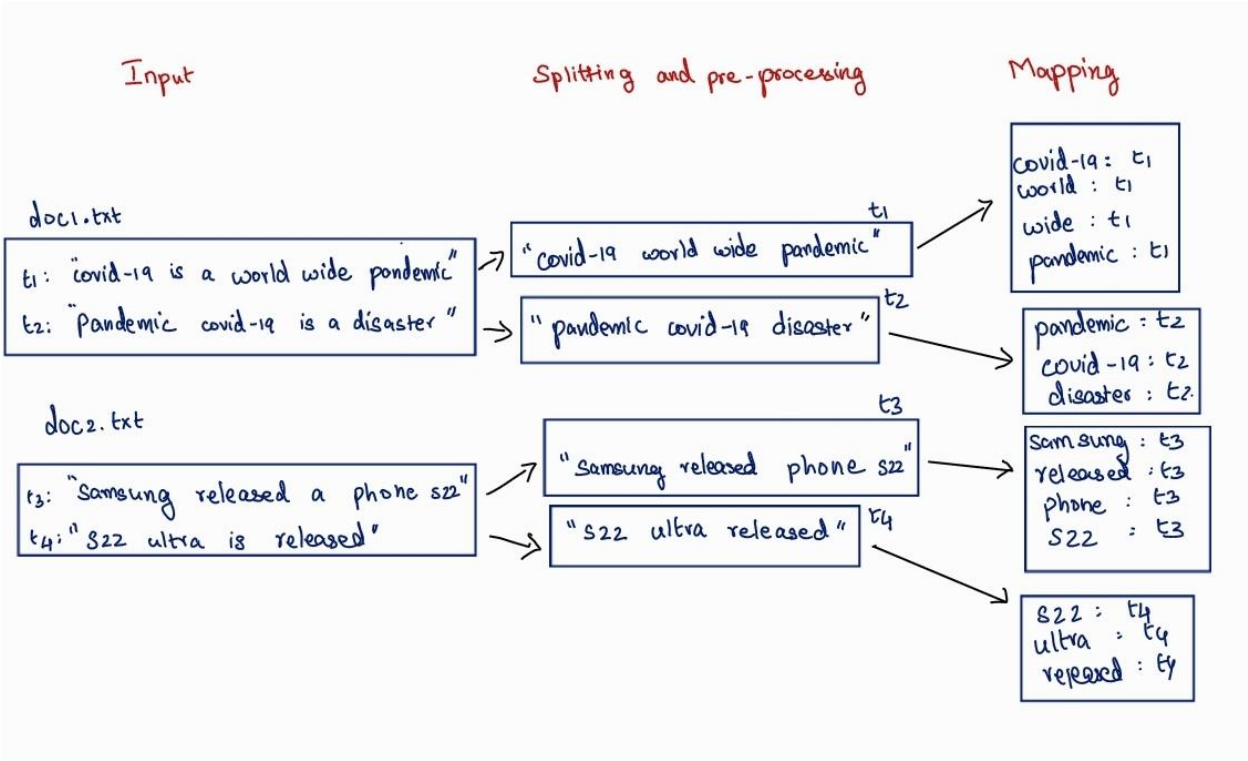

Fig. 3.1: Mapper Function Architecture

1. We read through every line of the twitter dataset and retrieve the tweet text.

2. We perform data pre-processing techniques like tokenization, punctuation removal, number removal, stemming and stop-word removal using the nltk library.

3. Finally, we iterate through every word the processed tweet and **yield** the word as the key and the tweet as the value.

```
def mapper(self, _, line):
    data = ast.literal_eval(line)
    string = data['Tweet']

    # Removing punctuations and numbers
    whitelist = set('abcdefghijklmnopqrstuvwxyz ABCDEFGHIJKLMNOPQRSTUVWXYZ')
    answer = ''.join(filter(whitelist.__contains__, string))
    stop_words = set(stopwords.words('english'))

    # Tokenizing
    word_tokens = word_tokenize(answer)
    filtered_sentence = [w for w in word_tokens if not w.lower() in stop_words]
    filtered_sentence = []
    ps = PorterStemmer()

    #Stemming and Stop-word Removal
    for w in word_tokens:
        w = ps.stem(w)
        if w not in stop_words:
            filtered_sentence.append(w)

    answer = (" ").join(filtered_sentence)

    for word in answer.split(" "):
        yield(word.lower(),string)
```

Fig. 3.2: Mapper code Segment

3.3 Reducer Function:

Input: (Key: word, Value: list of tweets)

Output : (Key: word, Value: sorted list of tweets with respect to word frequency)

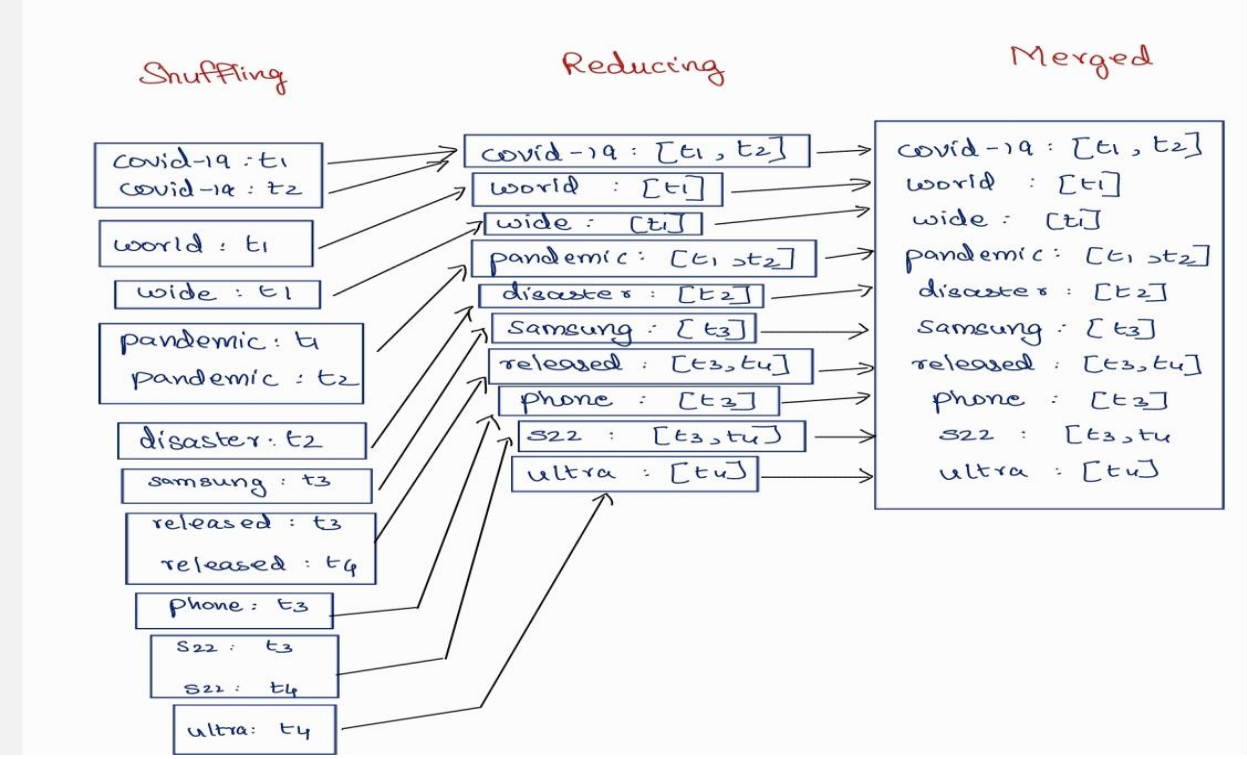

Fig. 3.3: Reducer Function Architecture

1. We iterate through every tweet and get the frequency.

2. Then we perform sort operation with respect to frequency and create a set of all the tweets in the list to remove duplicates.

3. Finally, we **yield** the word as the key and sorted list of tweets as value.

```
    def reducer(self, word, listlines):
        counts = collections.Counter(list(listlines))
        new_list = sorted(listlines, key=lambda x: -counts[x])
        skinny = list(dict.fromkeys(new_list))
        yield (word, skinny)

if __name__ == '__main__':
    Count.run()
```

Fig. 3.4 Reducer code segment

3.4 Key-Features of Hadoop Indexing:

1. We perform data pre-processing techniques like tokenization, punctuation and number removal, stemming and stop-word removal so that we don't index unwanted characters like numbers, punctuations and stop-words. Due to this we save a lot of undesired inverted indices and reduce the time taken to produce the indices.

2. We also consider the keywords in the **hashtags** after removing the punctuations. Keywords in hashtags receive a boost in weight as the same keyword will be present in both the tweet text and in the hashtag.

3. Ranking of the Hadoop-created index is with respect to the keyword frequency. Since the inverted index generated by map-reduce is already sorted, we upload the inverted index to Mongo-DB without any change in order.

3.5 Hadoop Inverted Index Output:

```
"acceleration"    ["#NIH has played a major role in developing #COVID tests. Through the Rapid Acceleration o
"accenture" ["5 Ways #Insurance companies rose up to the #COVID-19 challenge. (Accenture Insurance Blog) #Tec
"access"    ["The WHO has prequalified the first monoclonal antibody, tocilizumab, to treat COVID-19. The mov
"accessible"      ["Asking  the @AmerAcadPeds for mental health training and resources easily accessible for
"accompanied"     ["The tragedy of #HongKong two years into the #COVID pandemic - its deserted airport around
"according" ["8) What\u2019s more, positive #COVID screening results reached a record level in Denmark two da
"accounts"  ["#Covid updates in #Canada: \n3,177 Kcases(\u21910.3%):ON accounts for 34%,QC(28%),AB(16%)&amp;o
"accurate"  ["#NIH has played a major role in developing #COVID tests. Through the Rapid Acceleration of Diag
"accused"   ["Someone on here accused me of being like a soldier still fighting WWII after it was over becaus
"acknowledge"     ["Remembering Every Life Lost To Covid \n\nhttps:\/\/t.co\/djtjdDTPqc #via @IdeaSpies \n\nA
"acquired"  ["Domestic COVID-19 Status Update February 12:\nTotal: +54,941 (54,828 acquired locally)\n\n#coro
"across"    ["The #COVID outbreak is disrupting supply chains across \ud83c\uddf8\ud83c\udde7. \ud83c\udde6\u
"actasifphg"      ["#NakedFaceBreathing should not be a thing in a global airborne pandemic, yet here we are.
"acting"    ["16,000 residents in Ontario #LTC facilities have died during #covid @fordnation\n\nOn your watc
"action"    ["The queues for mandatory testing at Discovery Bay, #HongKong today. Meanwhile \u2066@SCMPNews\u
"actionrna" ["Nature, Feb 2022\n\"Pyrimidine inhibitors synergize with nucleoside analogues to block SARS-CoV
"activation"      ["During these challenging times, the members of the\u00a0@NVNationalGuard\u00a0have underg
"activewear"      ["Futuristic coating for hospital fabrics and activewear kills COVID and E. coli #coating #
"actual"    ["What the actual f**k??\n\n#aids #covid #vaccine https:\/\/t.co\/mp824VnRLE"]
"acute"     ["\u2018Acute Phase\u2019 Of COVID-19 Pandemic Could End By Mid-Year If 70% Gets Vaccinated: WHO
"adamcarolla"     ["Comedian @AdamCarolla reacts to Democrats' sudden reversal on #COVID masking policies: \u
"adding"    ["NEW #COVID VARIANT \u201cOMNICRON\u201d DISCOVERED IN SOUTH AFRICA \nJavid said, adding the var
"address"   ["#impactsofcovid\n#COVID #Coviddata #COVID19 \n\nhttps:\/\/t.co\/OBahpBZ6QH \n\nImpacts of COVID
```

Fig. 3.5 Hadoop Inverted Index

4. Lucene

4.1 Overview of the Lucene indexing strategy:

Lucene Core is an open-source Java library that can provide various strong indexing features and innovative search features.

For the Analyzer, we picked Standard Analyzer as it has features remove stop words, generate tokens and convert text to lower-case.

4.2 Index Fields:

A single document is made of one or more fields, a field is the smallest unit of the index. Lucene provides different types of fields including String, Text, and NumericDoc. This project indexed tweets and hashtags in String Field format. Every word in the tweet except for the stop words comes together to give keys that are indexed. These are the keys that are used to fetch relevant tweets based on the words used. Then, we use hashtags as it retrieves the topics with maximum coverage.

4.3 Lucene QueryParser:

QueryParser is a Lexer which interprets a string into a Lucene Query using JavaCC. This project uses LUCENE_36 release along with standard analyzer for stemming because the analyzer used to create the index will be used on the terms and phrases in the query string. So, it is important to choose an analyzer that will not interfere with the terms used in the query string.

```
String parserQuery = "";
QueryParser queryparser = new QueryParser(Version.LUCENE_36, f: "tweet", new StandardAnalyzer(Version.LUCENE_36));
```

Fig.4.1: QueryParser

4.4 Regular Expression Searches

Lucene supports regular expression searching pattern between forward slashes "/", using this features a tokenizer is created which then uses the regular expression to eliminate extra junk works.

```
//Tokenizer is used to remove junk character that might be indexed otherwise
StringTokenizer tokenJunkElimination = new StringTokenizer(queryString, delim: " ~`!#$%^&*()_-+={[}]|;'<>,./?\"\'\\/\n\t\b\f\r");
```

Fig. 4.2: String Tokenizer

4.5 Field Grouping:

Lucene supports using parentheses to group clauses to form sub queries, it also supports using parentheses to group multiple clauses to a single field.

4.6 Multiple Field:

Fortunately, Lucene has classes which supports multiple field search, this project makes use of **MultiFieldQueryParser**, which constructs queries to search multiple fields.

5. Runtime Comparison between Lucene and Hadoop Index Creation:

5.1 Runtime Graph Lucene:

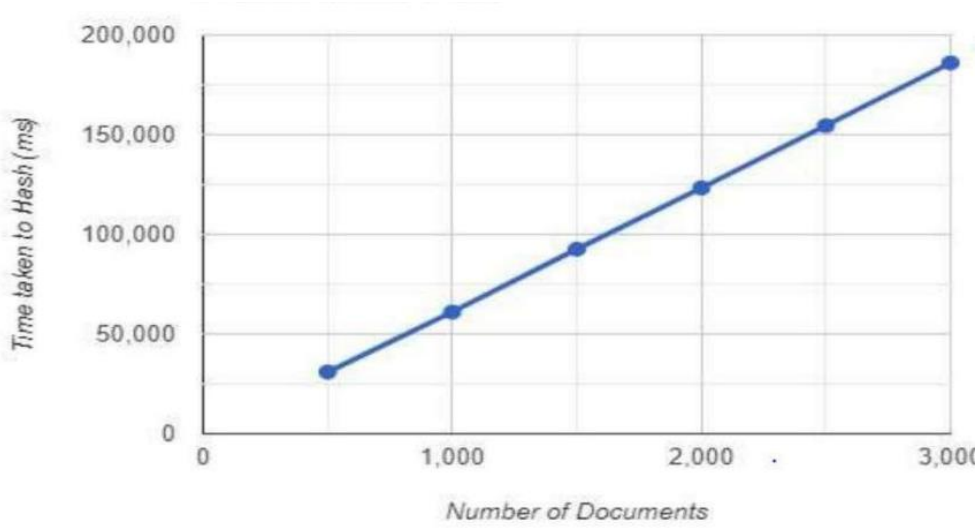

Fig.5.1: Lucene Runtime Graph

5.2 Runtime Graph Hadoop:

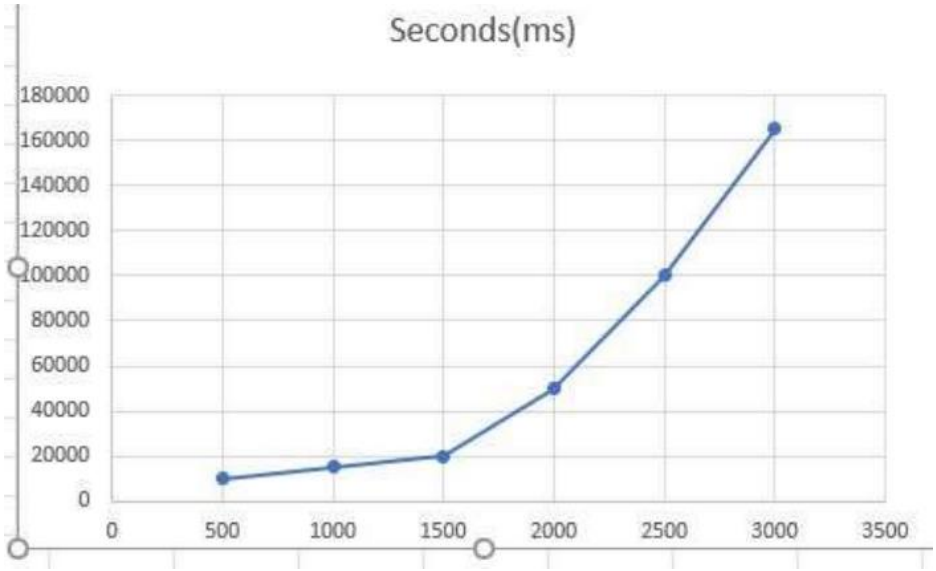

Fig.5.2: Hadoop Runtime Graph

Comparing the time-taken taken to index documents between Lucene and Hadoop, we can find that **Hadoop index is created faster than Lucene**. (To index 3000 documents Lucene takes around **200 seconds** whereas, Hadoop takes around **160 seconds**.) We can also notice that index creation time for Lucene is exactly proportional to the number of documents, unlike Hadoop.

6. MongoDB:

1. MongoDB is a No-SQL database that stores data in JSON-like format. The output retrieved in the form of inverted index from the Hadoop mapreduce code is stored in the MongoDB database.
2. Based on the query typed by the user in the search engine, the hadoop mapreduce program executes and fetches the data from the database and displays it in the UI.
3. The output is of the form word1:[Tweet1,Tweet2....] which shows the keyword and the tweets that particular keyword occurs in.
4. Each of these words are stored in a single collection in the database in MongoDB. The words and Tweets are stored in the form of a [key,value] pair within the collection.

7. Front End:

The Frontend of this project has two parts:

- Web Server - Python Flask
- Framework - Angular JS

Setting up the Python Flask server - We created a 2 APIS for the project.

1. /home - home page

2. /out - which displays the output of the search query

The flask server has the data retrieval code from the database. When the server receives a response from the UI with all the search filters and search words, it retrieves the data from MONGODB and displays the results accordingly.

As part of the search filters, we included lucene search and hadoop search which displays results based on the tool the user selects.

We also included **sentiment analysis** which displays the results with the output of the analyzer next to it.

After the search query is given, selections are made. The response is set to “GET” method in the flask server. The method looks up for the word in DB and returns the data to the HTML code in angular framework, eventually triggering all the relevant files(CSS and JS) results are displayed with or without sentiment analysis in a new page with an API - /out.

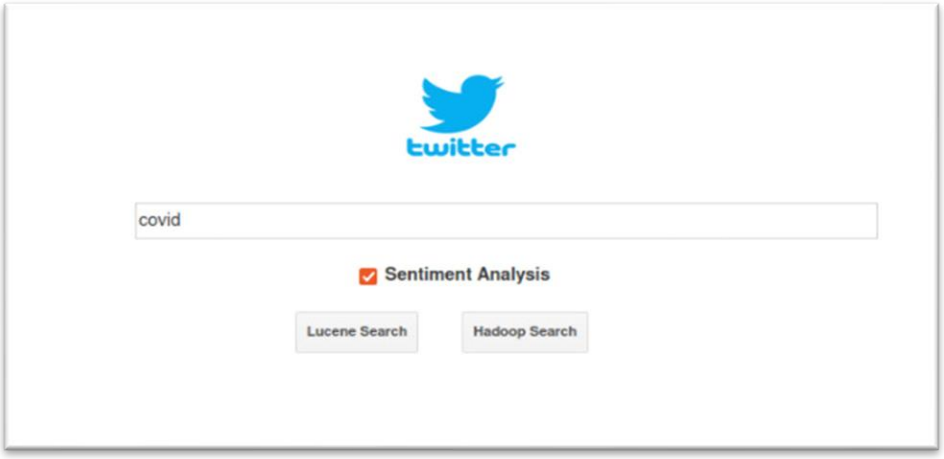

Fig.7.1: UI Homepage

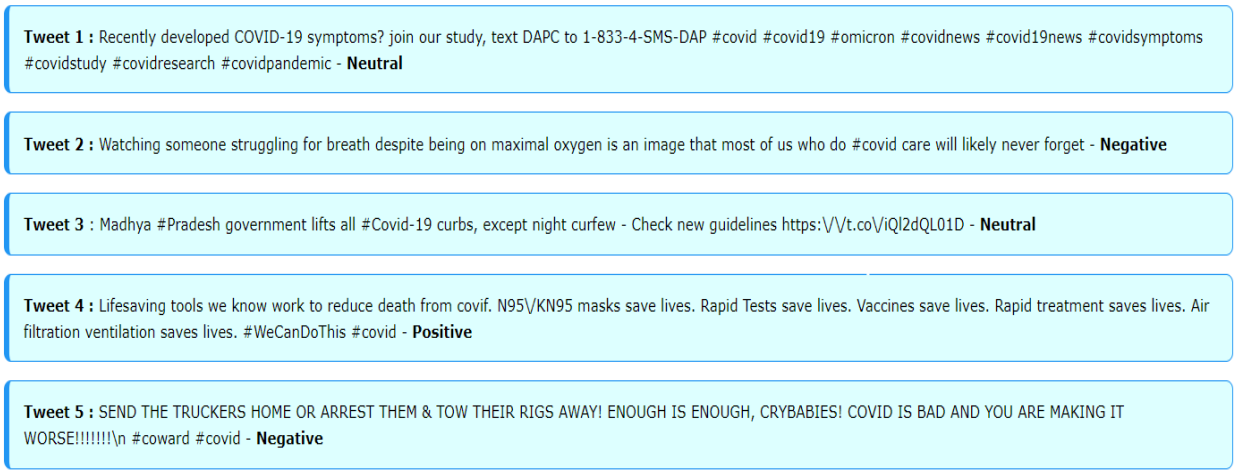

Fig.7.2: UI Output page

8. Extra Credit:

Sentiment Analysis, often known as Opinion Mining or Emotion AI, is a method of determining the public's feelings on a given topic. Major companies have realized that being sentiment-aware can help them gain insights into user behavior, track and manage their online presence and image, and use that information to boost brand loyalty and advocacy, marketing message, and product development.

We use the BERT model and the Pytorch Framework to develop a Sentiment Classifier to evaluate the tweets that we obtain as the input from the Mongo-DB retrieval phase. The SMILE Twitter dataset was utilized initially for the training purposes. This dataset was accumulated between May 2013 and June 2015. We'll use BERT and the Pytorch library to handle text preprocessing to build a Sentiment Classifier. The performance of the classifier was evaluated based on the Accuracy Score.

The BERT transformer model that is used over here internally consists of a self-attention mechanism. The BERT architecture is made up of a series of Encoders stacked on top of each other. It accepts a string of words and passes it to the Encoders. Each layer employs self-awareness passes its results through a feed-forward network, and then passes the information on to the next encoder. Self-attention examines each word, as well as the placements of all other words in the vicinity. When the model examines the word it, for example, it also looks at all the other words in the sentence at the same time, with a particular focus on the terms closely similar to the ones in the input. BERT employs Transformer, an attention mechanism that learns contextual relationships between words or sub-words in a text.

Using train test split, we need to split our dataset into a train and val set. We have utilized 75 percent of the dataset as training data, and the remaining 25% will be used to assess performance (holdout set). Because the classes were unbalanced, the forecasts that resulted were similarly unbalanced. Overall, we got an 82 percent accuracy rating. When working with imbalanced datasets, the difficulty is that most machine learning techniques will overlook the minority class, resulting in poor performance, even though performance on the minority class is often the most significant. Minority classes such as furious, disgust, sorrowful, and surprise are especially significant in sentiment analysis, where we want to know honest and constructive comments from the community to repair faults and enhance our product and service. Thus, through this BERT powered pipeline we were able to identify the sentiments of each tweet.

9. Obstacles and Solution:

Obstacle 1: One of the biggest obstacles faced during inverted index creation in Hadoop is the ranking the tweets with respect to the frequency.

Solution: Wrote code in the reducer function to order the tweets with respect to frequency of the keyword before yielding the results.

Obstacle 2: Lot of undesirable indices created during Hadoop index creation.

Solution: Performed sentence pre-processing like stop-word removal and stemming to overcome this obstacle

Obstacle 3: Duplicate tweets were retrieved while streaming it from Tweepy.

Solution: Once all the tweets are retrieved, we perform SET operation to remove duplicates.

10. Limitations:

1. The inverted indices generated does not have tweets that are contextually related.

Ex: The word **bat** will be mapped to both: baseball bat and creature bat.

2. The code to extract Tweets lacks parallelism but since Tweepy API is faster than regular RESTFUL APIs, this limitation can be overlooked.

3. Lucene indices are restricted to its own formatting. It's not a big limitation, yet it prevents the code from using other customized intuitive ranking and indexing algorithms.

11. Instructions on how to deploy the system:

To run the Hadoop Indexing Code:

a. Inside the hadoop-code file run the below line in the terminal:

   python hadoop_inveted_index.py data/*.txt > output.txt

b. To push the inverted indices to Mongo-DB run the below line in the terminal in IR_UI\modules:

   python mongodb_insertion

c. To view the UI, run this in Terminal:
   ng serve
   localhost:5000/