Twitter Sentiment Analysis and Search Engine

CS242 Final Project(Group 6)

Mehrnaz Ayazi
Soroosh Safari Loaliyan
Manuel Argueta Rodriguez
Javad Saberlatibari

Prof. Vagelis Hristidis
March 17, 2021

Member Contribution on Each part

Sorted by contribution of each member

Report:

- Mehrnaz Ayazi,
- Javad Saberlatibari,
- Soroosh Safari Loaliyan,
- Manuel Argueta Rodriguez

Hadoop Map-Reduce Job:

- Manuel Argueta Rodriguez,
- Javad Saberlatibari, Soroosh,
- Safari Loaliyan,
- Mehrnaz Ayazi

Hadoop Search:

- Javad Saberlatibari, Soroosh,
- Safari Loaliyan,
- Mehrnaz Ayazi,
- Manuel Argueta Rodriguez

Lucene Indexing and search:

- Soroosh Safari Loaliyan,
- Javad Saberlatibari,
- Mehrnaz Ayazi,
- Manuel Argueta Rodriguez

Twitter Data Crawler:

- Mehrnaz Ayazi,
- Soroosh Safari Loaliyan,
- Javad Saberlatibari,
- Manuel Argueta Rodriguez

Introduction

The goal of this project is to :

1. Crawl twitter user timeline
2. Determine each tweet's sentiment
3. Index the data using lucena and Hadoop
4. Create an interface that let's user search for a query and see the results with the tweet's sentiment towards the subject

What makes this project different?

1. Sentiment Analysis of the tweets and the data gathered by the program
2. multithreaded data collection
3. Tweet duplication check
4. User duplication check

Tweepy is a Python library for accessing the Twitter API. It is great for simple automation and creating Twitter bots. In this project, we use Tweepy to access tweets in different users' timelines. We then save the collected data in JSON format and hand it to our Lucena indexer. Lucene indexes the data by its text, writer, date, ... Indexing the data with these columns lets our user search the data by the text, author, date, etc. and gives the user more freedom in their search, as well as better search results since each of these attributes, have different weights and values.

Hadoop indexing which indexes data gathered by our crawler into an inverted index containing document IDs that contain each word and the term frequency/ inverted term frequency of that word in the document.

And finally, we have a user interface that lets the user choose their search method between Lucene and Hadoop and return the most relevant tweets to the query based on the indexing method user chose.

In the following Chapters, We will explain each section in detail.

Crawler

Twitter has a plethora of information that can be retrieved using Twitter streaming APIs. We decided to use collect our data in the form of user tweets. To do this, we crawled Twitter using Tweepy to collect tweets in English until we reached our limit of 250MB of data. This Twitter streamer is not limited to a particular subject, but rather is allowed to collect general data starting with a specific user. The application of this streamer is versatile and may be modified later to include only desired tweets from a particular subject or user.

We are using Textblob to analyze tweet text and determine whether it's a positive, negative or neutral tweet.

There are a few essential functions that need to be considered for crawling Twitter. Among them, the crawler needs to be able to authenticate a user in order to crawl, the listener needs to handle tweets from an incoming stream and establish a connection to the API, and the API needs to be in place to provide methods for various parameters. These provide a basic architecture for our crawling system, and we use the API to make adjustments for our project's purposes. In order to develop the tweets crawler, we exploit from python.

The main functions in our Crawler are:

- `get_friend_list(twitter_user)`

  Get's a twitter user as an input and return all the friends of that particular user. Friends are defined as people who follow the user or the user follows

- `recieveTweets(tweets)`

  Gets an array as an input and fills the array with tweets from twitter users timeline. This Function is the main function of the crawler. It starts crawling timelines with the user authenticated by us and adds other user to a queue if they weren't already crawled. After Crawling each user, the userID is marked as check so it won't be crawled again. Same happens for each tweet. tweet_to_data_frame and tweets_to_data_frame are also used in this function to convert collected tweets to data frame and then a json file for each tweet.

- `tweet_to_data_frame(tweet)`

Takes a tweets as an input and creates a data frame based on the tweet with columns: id, screen_name(Author's Username), name, location, description, followers_count, len(Length of the tweet), date, source, likes, retweets

- `tweets_to_data_frame(tweets)`

Works just like tweet_to_data_frame only this function creates a data frame of tweets for each user so it can be later used to plot user's data

- `analyze_sentiment(tweet)`

This Methods takes a tweet and analyzes its sentiment using TextBlob. We assign 1 to tweet if it's a positive tweet, 0 if it's neutral and -1 if it's a negative tweet.

Example of Json output file

[{"tweets":"\"What's been stopping Gronk is Tampa Bay's need to run a certain offense in order to protect Tom Brady\"\u2026 https:\/\/t.co\/Z3C6jfXyRY","id":1357776669101195280,"screen_name":"PatMcAfeeShow","name":"Pat McAfee","location":"Indianapolis","description":"Award winning wrestler. Kicked off a SuperBowl. Tossed bowls to Red Panda. LIVE M-Friday Noon-3est on YouTube & @siriusxm. @FDSportsbook\u2019s cool.","followers_count":2015738,"len":128,"date":1612554158000,"source":"Twitter Media Studio","likes":96,"retweets":8}

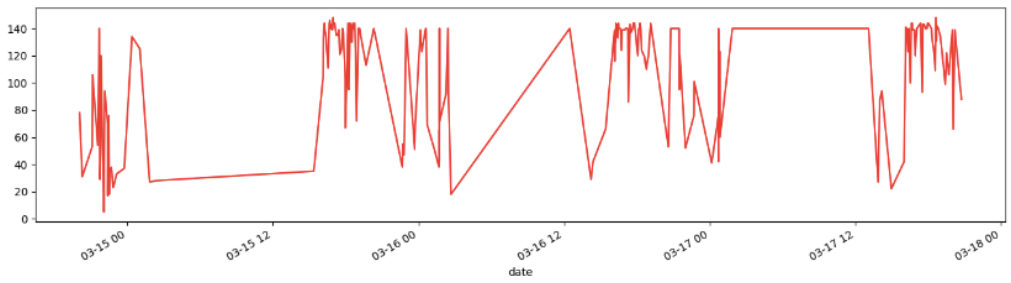

Fig.1:Crawled data shown in a graph(Length of the tweets/date)

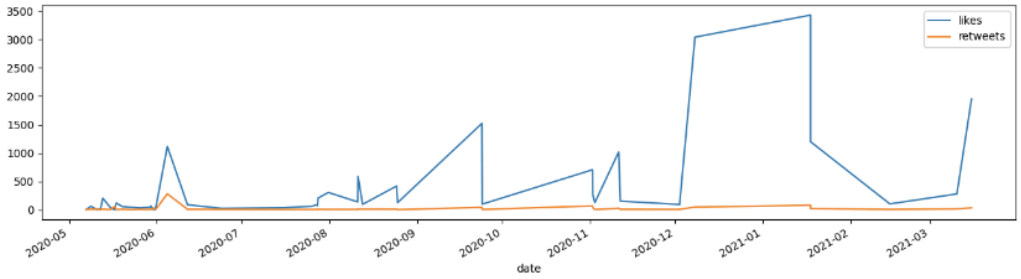

Fig.2: Likes vs retweets for a user over time

Lucene indexer

We develop our indexer by exploiting the Java-based Lucene. The input of our indexer is the *json* files which are the output of our crawler. In the following sections, we will briefly describe each part of our indexer.

The indexer indexes the data by:

1. Tweet id
2. Author username
3. Location
4. Author description
5. Author follower counts
6. Tweets Length
7. Date
8. Source of the tweet
9. Likes
10. Retweets

These datas are the most important datas to determine the rank of a tweet when a query is searched. Using these columns we can optimize our search methods in a way that the user receives the best results based on their search and history. (For example if the user is located in the US tweets from the US have higher priority)

- **TextFileFilter.java**

Since our indexer accepts the *json* files only as inputs, we put a filter on all the indexer inputs to determine the type of the input files in the *TextFileFilter.java*.

- **Searcher.java**

In order to easily test and evaluate the indexer, we developed *Searcher.java*. The reason is by proving “search in indexed tweets”, we are able to validate the correctness of our indexer functionality.

- **LuceneConstants.java**

The constant values which are required by our indexer to correctly execute are in *LuceneConstants.java* including file name, file path, and the maximum number of searches.

- **Indexer.Java**

The main implementation of our indexer is in *Indexer.java*. At first, it selects a field from the list of *json* files. Then in the main loop, the indexer inserts a document based on the field type, e.g., tweets, name, screen_name, etc.

Indexer uses StandardAnalyzer, Standard analyzer use StandardTokenizer, LoweCaseFilter and StopFilter to parse and index documents.

- **LuceneTester.java**

Our main function that calls the Indexer and Searcher is in *LuceneTester.java*.

Hadoop Map-reducer and Search

Our Hadoop map-reducer contains three parts. A mapper, a reducer and a combiner. We are saving tf-idf of each word in our documents(tweets) and the document ID in the output of our Map-Reduce. We use MRJob to write our map-reducer.

We're going to emit a per-tweet TF-IDF. To do this, we need three MapReduce tasks:

- The first will calculate the number of documents, for the numerator in IDF.(We should determine this in the crawler part of the project by specifying users that are crawled and number of tweets we get from each user's time line)
- The second will calculate the number of documents each term appears in, for the denominator of IDF, and emits the IDF (log(total # documents/# documents with wordX)).
- The third calculates a per-tweet IDF for each term after taking both the second MapReduce's term IDF and the tweet corpus as input.The third MapReduce multiplies per-tweet term frequencies by per-term IDFs. This means it needs to take as input the IDFs calculated in the last step **and** calculate the per-tweet TFs.

```
Counters from step 1:
  FileSystemCounters:
    FILE_BYTES_READ: 499365431
    FILE_BYTES_WRITTEN: 61336628
    S3_BYTES_READ: 1405888038
    S3_BYTES_WRITTEN: 8354556
  Job Counters :
    Launched map tasks: 189
    Launched reduce tasks: 85
    Rack-local map tasks: 189
  Map-Reduce Framework:
    Combine input records: 0
    Combine output records: 0
    Map input bytes: 1405888038
    Map input records: 516893
    Map output bytes: 585440070
    Map output records: 49931418
    Reduce input groups: 232743
    Reduce input records: 49931418
    Reduce output records: 232743
    Reduce shuffle bytes: 27939562
    Spilled Records: 134445547
```

Fig.3 Report of the analyzed data

Choosing TF-IDF as a measurement for ranking lets us have a good ranking method for our data without unnecessary computation. This method is easily generalizable to big data and long queries as well as being simple and easy to compute.

Our Map reducer contains the following methods. We used MRJobs for writing these map-reduce methods.

```
keyword_mapper(self, _, tweet)
```

This method receives a tweet as an input and emits (key, value) pair of (word, DocID)

```
count_combiner(self, word, id):
```

This methods add a 1 to (word, DocID) pairs so we can sum them up in later stages to calculate term frequency

```
word_counts(self, word, counts):
```

This method adds up the number of times a word is repeated in a document. In other words it calculates the term frequency of each word in all documents. This method is a reducer to keyword_mapper function.

```
total_word_mapper(self, word, doc_counts):
```

This method calculates total number of the words in all document so that it can be used to calculate document frequencies of the words using sum of their term frequencies.

```
tfidf_reducer(self, word, weights):
```

This method is used as a reducer to word_count method(or a combiner in the whole system). It calculates tf-idf of the terms using the (key, value) of the input and total_word_mapper output(total number of words in all documents)

Search:

After indexing our data using Hadoop, we can easily find the most relevant document based on user query and our inverted index. First we find all the documents containing important words in the query and then sort them using TF-IDF of the query in each document.

User interface