Marcus Martin - mmart149
Suhas Ejnavarzala - sejna001
Samuel Tapia - stapi010
Abdelrahim Hentabli - ahent001

CS242 Project 1

Collaboration Details

- Marcus: Crawler/Report
- Suhas: Crawler/Report
- Samuel: Lucene/Report
- Rahim: Lucene/Crawler/Report

Python Crawler

Crawling System Architecture

In this project, I was responsible to write the python code for twitter crawler based on recent presidential elections based on the keywords:

```
'Biden', 'President', 'Harris', 'Trump', 'Pence', 'Inauguration', 'Impeachment', 'Ted Cruz',
'Mitch McConnell', 'Congress', 'Senate', 'House of Representatives'
```

As a part of the python code, the first step is to import all the libraries csv, os, threading, requests and tweepy libraries. These libraries are needed to download tweets based on keywords.

Before initiating the twitter crawler, the following variables must be declared from the imported libraries. The environment variables include MAX_SIZE, CONSUMER_KEY, CONSUMER_SECRET,ACCESS_TOKEN, ACCESS_SECRET. The tweets authentication handler will use Consumer key and consumer secret attributes. The input value for these can be obtained by creating an account in twitter.

As a part of the code, a function download is initialized whose main purpose is to download a file at link to a file location and create subdirectories when necessary. The requests module is used to download large data files. There is a check for race condition where the system's substantive behavior is dependent on the sequence or timing of other uncontrollable events. The output file is written in a binary mode. Python makes no changes to data when the file is being written. The *r.iter*_content method allows us to specify the size of data to download by defining the chunk-size in bytes. In this case, it's set at 1024 bytes.

A second function *customer listener* is created which creates a file name based on the URL but replaces characters to be more readable. The download of tweets is started in a thread and the data retrieval speed increases as downloads as I/O bound. The code also has the capabilities to download in parallel (*multithreading*) when required. There is also a condition as a part of the class *download*, where the crawler stops after collecting 250 MB of data. For this project, We have collected 269 MB worth of tweets. The twitter crawler output is written to the file *output.csv*. The columns that are downloaded as a part of the twitter feed include

- Text
- Timestamp
- Geolocation
- User ID
- Links
- Hashtag(s)
- Retweet Count

The code also distills the hashtags from the text tweets of some tweets into a convenient data structure for further examination.

The code also checks whether the tweet was under or over 140 characters, which was the old twitter limit. The twitter API is managed in such a way that newer tweets that use the larger tweet limits are stored in a different way. The crawling code checks which of the standards the tweets are stored, to ensure that the full, non-truncated text of the tweets is stored. Furthermore, the code checks whether a tweet is a retweet. If it is, it retrieves the retweeted text in its entirety. This is done by using the tweepy API to retrieve the retweeted text's full text.

Finally, for the sake of formatting and user experience, the twitter crawler also prints out a current total (in KB/MB) of the size of tweets collected.

Crawler Set up

To execute the Twitter crawler code, we first needed to create a Twitter developer account. After creating the account, we created a developer project with an associated app. Once Twitter generated authorization credentials for the developer app, we decided to use OAuth 1a Authentication to authorize the app to make Twitter API requests from the developer account. Since authorization credentials are required for each request, OAuth 1a Authentication needs to be given authorization credentials called keys and tokens that will be used with each request. Two keys called consumer and consumer secret act as the username and password for the developer app, and two tokens called access and access secret identify the developer account that is making a request.

We first give the consumer and consumer secret keys to Tweepy’s OAuthHandler function and then open an authorization url that gives the user the option to authorize the app. Once the app is authorized, a pin is given to the user that is then used to get the access and access secret

tokens for the developer account. Now that we have all the authentication credentials, the app will be able to make Twitter API requests.

Deploy the Crawler Instructions

Install Python 3.9.1 from python.org and setup the environment variable for Python.

Please follow the instructions below to run the crawler script:

- Run command in project folder directory: python DeployCrawler.py
    - Authorize the app and enter the pin when prompted

Please follow the instructions below to run the crawler with Jupyter Notebook:

Environment Setup

- Run commands in project folder directory:
- Pip install pipenv
- pipenv install tweepy==3.9.0
- pipenv install jupyter
- pipenv install python-dotenv
- pipenv shell

Deploy Crawler

- Run the command in project folder directory: jupyter notebook
- Open DeployCrawler.ipynb and run the program
- Authorize the app and enter the pin when prompted

Lucene

Environment Setup

1. Download Lucene Core JAR libraries
    a. lucene-core-8.7.0
2. Unpack the download to a desired location.
3. Then we must create a PATH_TO_LUCENE with the following variables, which is handled in the bash script in the Build section.
    a. for example, path to lucene may be "/usr/lib/lucene-8.7.0" if the lucene directory is there
4. Compile script, takes the user's path and adds the following paths to the classpath in order to compile and run.
    a. …/core/lucene-core-8.7.0.jar
    b. …/queryparser/lucene-queryparser-8.7.0.jar
    c. …/analysis/common/lucene-analyzers-common-8.7.0.jar
    d. .../lib/opencsv-3.8.jar

Build Lucene Index

1. Installing
    a. To run the code, you first need to clone the repository or download the files.
2. Compiling and Running
    a. You need to export a PATH_TO_LUCENE variable for the compile and run scripts to run correctly

```
export PATH_TO_LUCENE=<your path to lucene>
```

        i. This needs to be done each time you reopen your terminal, or you can put it in your .bashrc/.zshrc
    b. To compile on a linux system, change to the main directory and run the compile.sh script. You may need to give it execute permissions with

```
chmod +x compile.sh
./compile.sh
```

    c. To create an index, you need to be in the main directory and run the index.sh script. Again, you may need to give execute permissions to the file with

```
chmod +x index.sh
./index.sh
```

Run Queries on Index

After the index has been compiled and ran, you will be able to query the index file.
To run queries on the index, you need to be in the main directory and run the query.sh script. It will need execute permissions.

```
chmod +x query.sh
./query.sh
```

This will allow us to submit queries and return the results from the index we created. No parameters are required since the bash scripts are handling this for us. The querying runs in an infinite loop, and requires "quit" to exit.

Fields in the Lucene index, with justification (e.g., indexing hash tags separately due to their special meaning in Twitter).

We used the following fields for indexing:

- **id** - All users have an ID associated with them, therefore, it would be ideal to query based on ID. Thus indexing the ID would allow us to rank tweets accordingly.

- **tweet content** - The content of the tweet is the string of text that a user submits in a tweet. Indexing the words in the tweet is the main source of data we are looking for when searching tweets.
- **hashtag** - Hashtags are a set of keys that are associated with tweets. We decided to index this as a separate field to allow for more flexibility when submitting queries. Our ideal system would be search hashtags and if no results are found we then can output tweets that have the word in the content.
- **content** - Content is a concatenated string of the id, tweet content, and hashtag. The reason for this inclusion is to allow us to search the entire tweet not just by id or hashtag, but as a whole string. Allowing for a better score result, in the resulting queries.

Text analyzer choices, with justification (e.g., removing stop words from web documents; using separate analyzers for hashtags and keywords).

We decided to use the StandardTextAnalyzer and in doing so we made modifications in how we indexed our data. Since the analyzer tokenizes the words not including the STOP_WORDS, we made a decision to add a new field that includes the entire concatenation of all the fields we have. In doing this, the analyzer will rank tweets not only on, id, content, and hashtag, but as a whole string. This allows for more natural results when we do our queries. In addition, we also double counted hashtags, by including it as a separate field. This allows for top results to include the hashtag and for the lower results to not include the tag but the word itself.

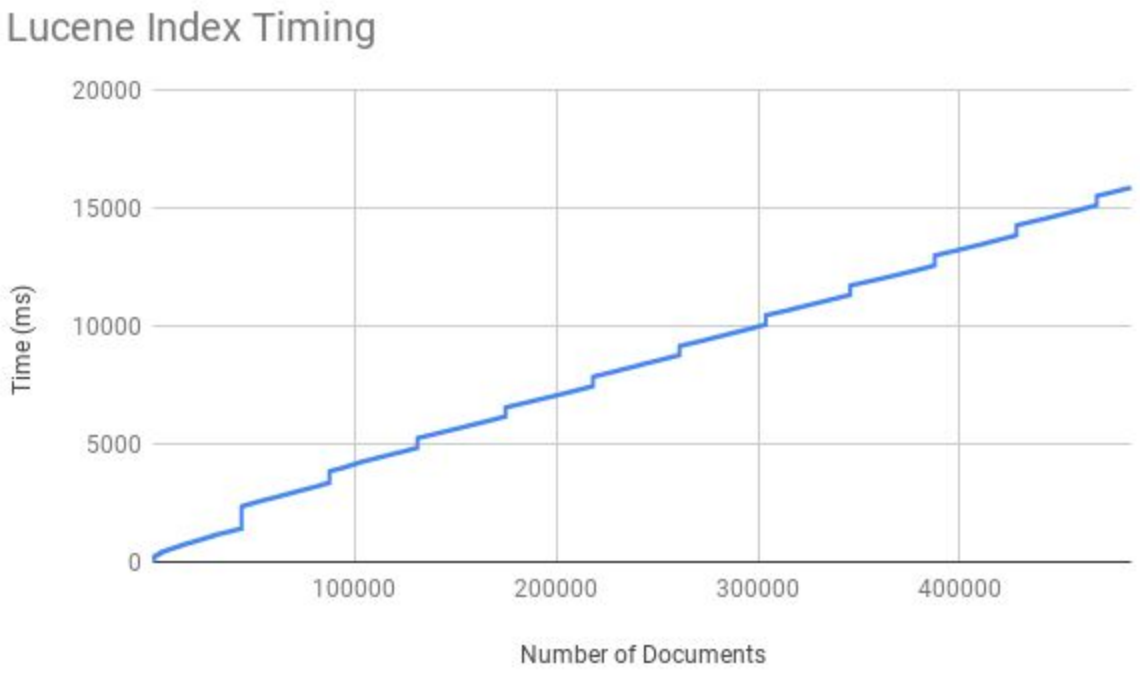
Our goal was to populate results that not only fit the query exactly, but to also provide the user with more information that could be related to the query. We also decided to stick with the standard set of stop words provided by the standard analyzer. We felt that the current set of words is ideal for searching tweets and did not want to omit more data from the query. Below is the set of stop words that are used.

STOP_WORDS = {"a", "an", "and", "are", "as", "at", "be", "but", "by", "for", "if", "in", "into", "is", "it", "no", "not", "of", "on", "or", "such", "that", "the", "their", "then", "there", "these", "they", "this", "to", "was", "will", "with"}.

Report the run time of the Lucene index creation process. E.g., a graph with run time on the y axis and number of documents on the x axis.

Overview Limitations.

Most of the issues we encountered with creating a crawler and using Lucene came in the environment set up. More Lucene than the crawler, but understanding how the environment works gave us a better understanding in how to better deploy the code in different workstations. Moreover, this allowed us to better prepare the repository to better handle different environments.



Obstacles and Solutions

Trying to obtain geolocation for tweets in a political theme, became difficult, the rate of populating data was extremely slow and deterred us from actually getting geotags. Thus, we left out geotags from our crawler. In the future we plan to use geotags, but would probably change the crawler tactics to further increase the performance of the crawler.

In the crawling of the tweets, a large amount of issues arose with the tweepy API truncating tweets in our output file. There was a large collection of '...' at the end of the tweet text. A large part of the changes to the revised twitter crawler code were just to get rid of those. We worked hard to remove these because it would greatly reduce the accuracy of our indexing.

Also, a large issue arose when we were updating the crawler code, because the code takes quite a while to finish retrieving tweets. Running the updated code at each iteration resulted in quite a few issues along the way. Our most recent tweets file was smaller due to the fact that we ran it up until the end of the project.

In order to better understand our results, we added scores to print with all the results, this allows for us to test our methods and theories when we created the index.