CSE 587 Lab 2 Report

DATA AGGREGATION, BIG DATA ANALYSIS AND VISUALIZATION

BY

TANNU PRIYA

RAKSHIT MURTHY VISWANATHAM

Introduction

In this Lab, we are trying to amplify our skills from Lab 1 where we will gather big data, analyze the data and then visualize it. We will be using various data sources and technologies with the logic and algorithm of MapReduce to fetch our data, clean it, run it through the algorithms and then visualize it. The 3 data sources that we are using will be Twitter, New York Times API, and Commoncrawl and get data relevant to our topics. Then we will use our MapReduce logic to first get a count of all the words in our data then we will find some important words and then send it through another MapReduce that will find the word next to it and give a count of most occurring pairs. The biggest aim with this lab is to understand the working of the Hadoop File System and how parallel programming works. Therefore, we are using AWS EMR and Docker with Cloudera HDFS to load our processed data and then run the algorithms on it and examine the output. We are gathering data for current U.S. Political Affairs which include the Russia Investigation, Robert Mueller's Report on that, Health Care situation, the Guns topic. We are also seeing what kind news is out there for the 2020 Democratic Candidates since the elections is already a talking topic. We will also be gathering data for Immigration related affairs and gather data on the climate change the green new deal that will combat it and was introduced by Alexandria Ocasio-Cortez and healthcare as well.

Technologies and APIs Used

- AWS EMR: We use this technology to run our HDFS system as it has Hadoop framework built-in to the Elastic MapReduce. We create clusters that can use the MR algorithm to work with the data that we have stored in the S3 bucket on AWS as well. It is cost-effective and easy to scale as required for the job.
- Python 3 was used to code all the scrapers, pre-processing code and the MR algorithm as well.
- AWS S3 is a cheap cloud storage and that is where we stored all the required data and the algorithm that we will use in the EMR. We also have secured it so that only the team partners can access the bucket and store the data on it.
- AWS EC2 instance was used to gather all the Commoncrawl data to keep the machine light.
- Tableau to visualize the results and create the required word clouds.
- We used various pyhton3 packages that helped in gathering and cleaning our data like:
    - Tweepy, which is used to access the Twitter API and gather all the required tweets.
    - Yanytapi through NYT API to gather all the required articles.
    - Boto3 AWS package to get Commoncrawl articles and store our data as well.
    - Nltk library was used to do the stemming on our text.
    - Many other smaller packages were used to remove regex, emojis and clean data.

Data Scraping

We scrapped data from three different sources. Twitter, NYT and Commoncrawl.

Twitter

For twitter, we first put in the credentials to get it ready and then use the required keyword to start searching for the results. We can also select how many tweets per search do we want. Once the results have been gathered, we go through each of them and first make sure that they have not been retweeted. Once the condition has been met, we start to get the full text from the Tweet object and retrieve the date of the tweet, if it was retweeted, id of tweet and the language of the tweet just to make sure they were all in English. Once we have gathered all this required information in the form of a JSON object, then we dump it into a file that can later be uploaded to our S3 bucket.

We used the Cursor package within Tweepy as it can help us keep the connection alive when we have hit the Twitter rate limit and once the limit has been passed then we can again continue the search. We also used the datetime package to keep the all the dates in the data in the same format if we needed.

Another piece to the Twitter search is the twitter_gather_clean.py. This python file is used to gather all the twitter json files that contain all the tweets that we have gathered by basic cleaning them before they go into pre-processing and mainly making sure that our tweets are all unique in every possible way.

We use the os package to gather all the tweet files across the directory and then start reading each of our file. We want to mainly make sure that the id of every tweet across every file has not been repeated and even though we have made sure that they were no retweets, there could be a chance that we have come across a tweet that was copied and posted but then we just add these tweets to our entire corpus to make sure that the next tweets can go through this check.

Once we have all the data required then we compile all these tweets and dump only the text of the tweet into one big file that can be used to pre-processed and then sent to EMR. We have more than 33,000 tweets that are unique, no retweeted and contain information for all our keywords within our main topic. The following is an image containing the json of each tweet object.


```
{
    "type": "twitter",
    "full_text": "@peterdaou No. I love Beto. Give more coverage
    to the women. And especially cut off Mayo Abercrombie and
    Fitch.",
    "id": 1114999563385962499,
    "created_at": "2019-04-07T21:13:06",
    "lang": "en",
    "retweeted": false
}
```


Figure 1 Sample gathered tweet

New York Times

Similar to Twitter we have used the yanytapi that uses the NYT Search API to search a list of the articles. We use our credentials, then we start searching for the articles that would contain our keyword in the headline, because we do not want any article containing our keyword even though no relevant to the keyword or the topic, we only want to deal with the articles that contain the keyword in the headline, therefore allowing us to only gather articles that can be relevant to us. In addition, we are also making sure that we are gathering articles in 2019 as that is the data relevant to us.

When the list of articles is gathered, we go through each one of them and first making sure that it is an article and not a video or tag or archive. Then we use the URL of the article and get the full article. Once we have the HTML body of the article, we put it through BeautifulSoup and parse it so that each HTML tag can be recognized. Once that is done, then we retrieve all the 'p' tags as they would contain the body of the article as that is all we need from the article itself. We also make sure that we are ignoring any parts of the articles that is sponsored ads, advertisements and the author's information as that is all required form the article.

We also make sure that no article is repeated within our same search to avoid any sort of repeating. Once all the conditions have been met, we store the similar details like we did for Twitter. We store the full body of the article and including the id, date, and the headline of the article. These details can then help us in the future to make sure none of the articles gathered across all the keywords are the same.

After all the articles have been gathered, we send through a cleaner like the twitter one, where all the articles are compared to make sure none are repeated, then some basic cleaning like removing URLs is being done so that pre-processing does not have to worry about the individual quirks of the NYT articles. At the end of it, only the body of each article is stored and dumped into one big article that can be used later to go through pre-processing and then def to our S3 bucket for AWS EMR to perform all the required processed to get our output.

Commoncrawl

This is the tricky part of the data gathering. There is no ready to use package and there is ton of unfiltered data that can potentially give us the wrong results. Therefore, after a lot of trial and error, we came up with a way to crawl the data we wanted and then store it in a file for further processing and storing it.

First, we got the segment of data we wanted to scour against from commoncrawl. We made sure that is just the first segment in the data for March 2019. Once we have this segment loaded into our code. We run it against the Boto3 package and access the public commoncrawl dataset and get the required keys of the files that we want to access to get the data that is needed.

We want to make sure to the WET extension as we want WET type of files. This type of file provides us with only the required plaintext of the article and the id and URL of the article. This way we do not need more packages and parsers to read the article's body and just access the data and extract if required. The key to each file in the bucket of the segment of the commoncrawl will tell us which key to access. Using this key, we then access the same commoncrawl bucket to get the gzip file with that key. We have used the same boto3 package to get the gzip file and store it locally. Once we have this gzip file contained our WET file. We unzip it using the gzip package and then send it to another function that can now parse it.

This function now has the WET file and it stores it to a temp file as the WARC-WET package needs the file to have a specific file extension. Therefore, we wrote the unzipped file to a temp WET file. Once written, we then open it using the package for WARC which can be used to now read the file. Each file has about 100,000 records of various different things like articles, videos, archives and just about everything that is on the internet. Hence, we setup an array that contains the words that we do not want the URL to have as these words means that the URL is a video or tag or search phrase on that website and therefore means nothing to us as it will contain no real information.

For each record in the file, we check the URL does not have any of those above words that can potentially mean useless data. The payload within the record is now read and decoded to make sure that we can do a language check on it. We want to make sure that the record's payload is in English, because we cannot do anything with records that are not English. If the payload is in English then we allow it to go ahead and therefore we can now check that the payload can contain any of our keywords.

We have a list of all the keywords that match our topic and therefore we want to make sure that that article contains any of our keywords. If it contains any of the keywords then want to extract that article and therefore store for further usage. Once we have made sure that the article contains our keywords, then we add it to a corpus that can later be written to a text file.

We know that commoncrawl makes sure that all the records are never repeated within any file therefore, we do not need to make sure of getting the same articles again and again. If the corpus of a file contains more than one article with our keywords, then we want to write it to a file and store it on our s3 bucket. We store them on an S3 bucket because these files are huge and we do not want to save them locally on our machine.

For the purpose of this project we only went through 20 files within a segment as even though we are doing big-data project. It is relevantly much larger than the amount of data that was gathered for Twitter and NYT. After all the files have gone through and extracted all the required articles from it, we end our process and delete any local files that were stored for the process.

When being stored in boto3, the file names for each extracted text file now also contains how many articles were gathered from the commoncrawl WET file. For example, CC-MAIN-20190326220507-20190327002507-00000_310.txt contains 310 articles that we wanted.

After all this process is done, we send it through another cleaner program like we did with Twitter and NYT. This cleaner again will make sure that there are no escape characters and URLs within our articles as they are of no need to us. We also consolidate all these articles into one main file that can then go into our central pre-processing code and be used for the MapReduce Programs.

All this process is taken place in the EC2 instance as we do not want to overload our local machine with all the writing and reading to files. This is pretty simple. We use AWS to setup an EC2 instance that is replica of our local machine and therefore can do the entire process similar to our local machine but much quicker and anything breaking will not cause any issue to our local machine as we can easily terminate an EC2 instance and restart the entire process. The following image shows the setup and the configuration that was used to setup my EC2 instance. I used a t2.large instance that is 2 vCPUs and 8 GB memory and therefore is enough for our program to run as that is the only thing that will run on it and is cheap enough to run for any type of budget user.

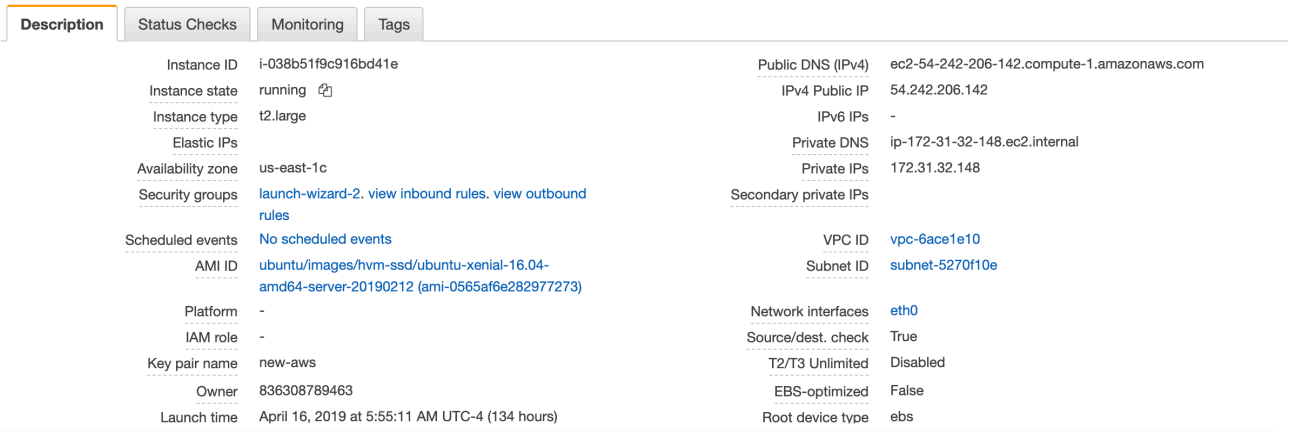

Figure 2 EC2 Instance configuration

Once we have the instance setup, we SSH into it with the user ubuntu and providing the security pair that was used to setup the instance. After logging in, we want to make sure that python is installed in it and install all the required packages with pip install command. Sometimes, we do not have pip installed as well, therefore we want to make sure that is installed as well. In addition, similar to local machine, we want to run aws cli and make sure that the AWS credentials are installed as well to connect to the commoncrawl and our local bucket.

After the entire architecture is setup, we move our python file and we use the command python <file_name> to run the file and therefore initiating our progress. The file will run similarly to how it would have run on our machine but much quicker and allowing us to do other work and over-loading our machine.

Pre-Processing

We are preprocessing every data after collection for below operations:

- **Decode and Lower Case:** We are decoding in UTF-8 and lowercasing using string lower() method of python for entire corpus before proceeding further with preprocessing
- **Remove stopwords**: We are using Natural Language Toolkit of python to remove stopwords from our data. First, we are importing English stopwords of **nltk** corpus and then we are removing every instance of stopwords from data and stored in clean_tokens.
- **Stemming**: We are then stemming the clean tokens using **WordNetLemmatizer**.
- **Regex for Punctuations**: We are processing our stemmed words with a Regex for remove everything but words from the corpus.
- **Writing File**: We are then writing the clean data to multiple files using file write method of python. These files contain our preprocessed data.
- The output is then stored in multiples files, which we are further feeding to out MR systems.
- We then also want to make sure that all the files are then stored in a directory that can then be uploaded AWS S3 bucket to be run through our EMR and give us the needed output.

Word Count

Mapper

The work of mapper in Hadoop Map Reduce is to assigns account value to every instance of word and emits it to the reducer. Mapper writes the intermediate results to local disk before being sent over the network. Since network and disk latencies are relatively expensive compared to other operations, reductions in the amount of intermediate data translate into increases in algorithmic efficiency. Also, in MapReduce, local aggregation of intermediate results is one of the keys to efficient algorithms.

Reducer

Reducer then takes the input form Mapper and count every instance of it from the input file. That is then emitted from the MR system as output. The output of MR is key-value pair for each unique term in the document.

We are then feeding the output of the Word count MR to Word Co- occurrence MR for finding the co-occurrences of top 10 words in the text file. Working of Word Co-occurrence MR is explained below.

Word Co-occurrence

Mapper

We have designed Hadoop Co-occurrence Map Reduce such that it is taking two inputs. First, is the list of top 10 most frequent words of the corpus and Second, is the corpus itself. It is then finding the instance to top words in the file and the co-occurring word with it. We have accomplished mapper tasks by two nested loops: the outer loop iterates over 10 most frequent word of the corpus, and the inner loop iterates over all neighbors of the of the 10 topwords (the right element in the pair). Then mapper emits that word and co-occurring word as input for the reducer with the count of 1.

Reducer

For co-occurrence Reducer, each pair that it gets as input from mapper corresponds to a cell in the word co-occurrence matrix. The MapReduce execution framework guarantees that all values associated with the same key are brought together in the reducer. Thus, in this case our reducer simply sums up all the values associated with the same co-occurring word pair to arrive at the absolute count of the joint event in the corpus, which is then emitted as the final key-value pair. Each pair corresponds to a cell in the word co-occurrence matrix.

Hadoop Infrastructure

To set up Hadoop infrastructure for storing and computing Big Data using Map reduce computations we have set up Amazon EMR provides a managed Hadoop framework that makes it easy, fast, and cost-effective to process vast amounts of data across dynamically scalable Amazon EC2 instances. We can also run other popular distributed frameworks such as Apache Spark, HBase, Presto, and Flink in EMR, and interact with data in other AWS data stores such as Amazon S3 and Amazon DynamoDB. EMR Notebooks, based on the popular Jupyter Notebook, provide a development and collaboration environment for ad hoc querying and exploratory analysis.

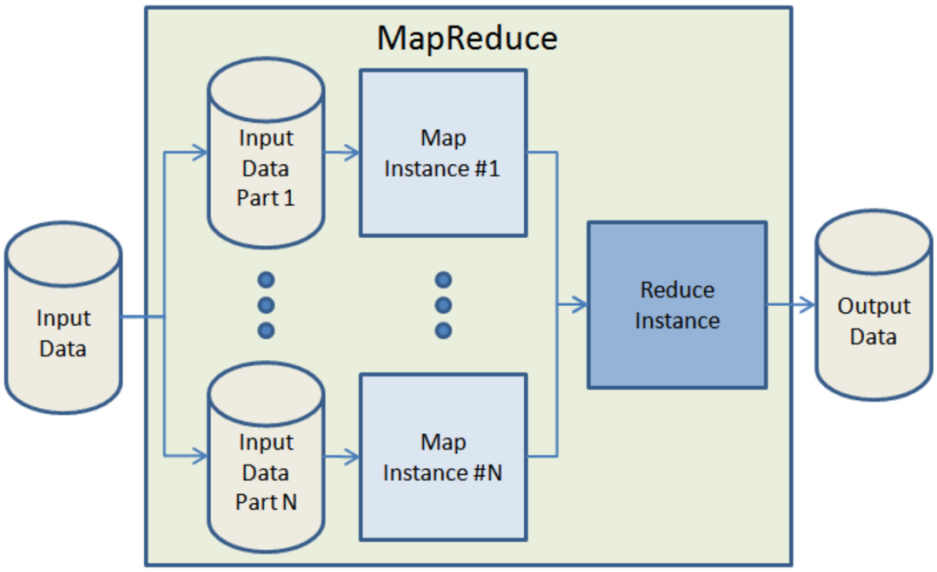

Figure 3 MapReduce Workflow

MapReduce takes care of splitting up the input data into chunks of more or less equal size, spinning up a number of processing instances for the map phase (which must, by definition, be something that can be broken down into independent, parallelizable work units) apportioning the data to each of the mappers, tracking the status of each mapper, routing the map results to the reduce phase, and finally shutting down the mappers and the reducers when the work has been done. It is easy to scale up MapReduce to handle bigger jobs or to produce results in a shorter time by simply running the job on a larger cluster.

Using Elastic MapReduce, we can create, run, monitor, and control Hadoop jobs with point-and-click ease. We don't have to go out and buys scads of hardware. We don't have to rack it, network it, or administer it.

We don't have to monitor it, tune it, or spend time upgrading the system or application software on it. We can also run world-scale jobs anytime we would like, while remaining focused on our results. We followed below steps below steps for executing Map Reduce on Amazon EMR:

Figure 4 Steps of using AWS EMR

S3 bucket

Amazon S3 or Amazon Simple Storage Service is a "simple storage service" offered by Amazon Web Services that provides object storage through a web service interface. Amazon S3 uses the same scalable storage infrastructure that Amazon.com uses to run its global e-commerce network.

We are using S3 bucket for storing all our MR code and input files. We are also generating output in the S3 bucket. Our S3 bucket looks something like Figure 1. Where maaper.py is our mapper code and reducer.py is our reducer code. The input directory has all the Twitter/NYT/Common Crawl data.

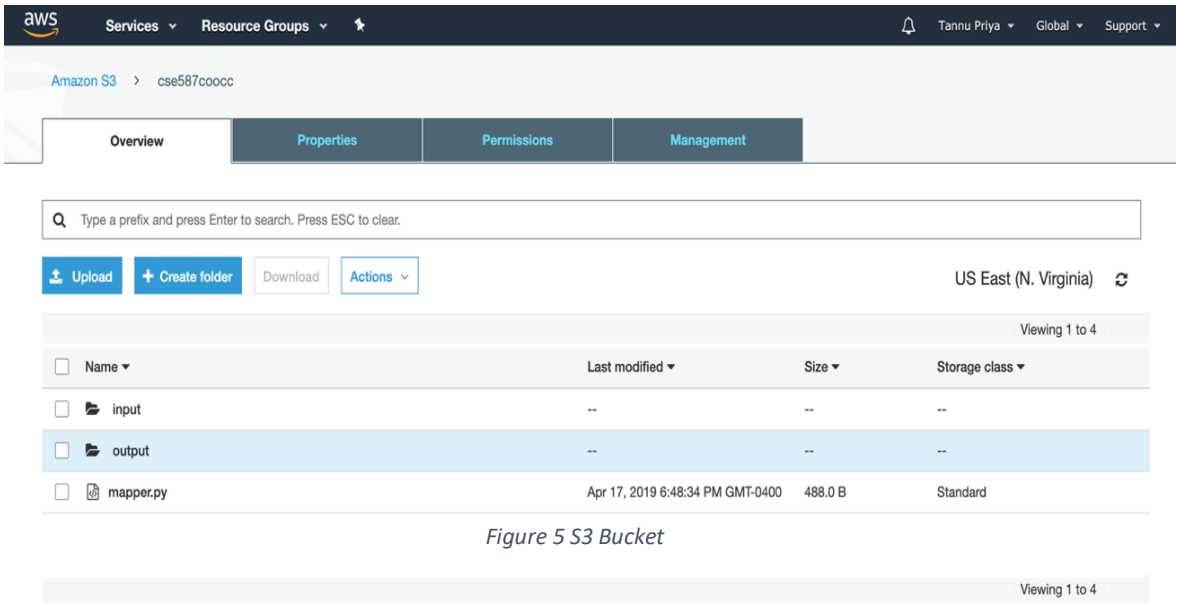

Figure 5 S3 Bucket

Amazon EMR

Amazon EMR is based on Apache Hadoop, a Java-based programming framework that supports the processing of large data sets in a distributed computing environment. MapReduce is a software framework that allows developers to write programs that process massive amounts of unstructured data in parallel across a distributed cluster of processors or stand-alone computers. We are creating clusters on EMR to do the MR. We used Hadoop 2.8.5 for MR and mapped the code and input files with the S3 bucket. We are using instance type to be m4.large with 4 vCore, 8 GiB memory, EBS only storage EBS Storage:32 GiB, we are also updating these configurations as and when needed. Post that we run two Nodes:

1) Master 2) Core

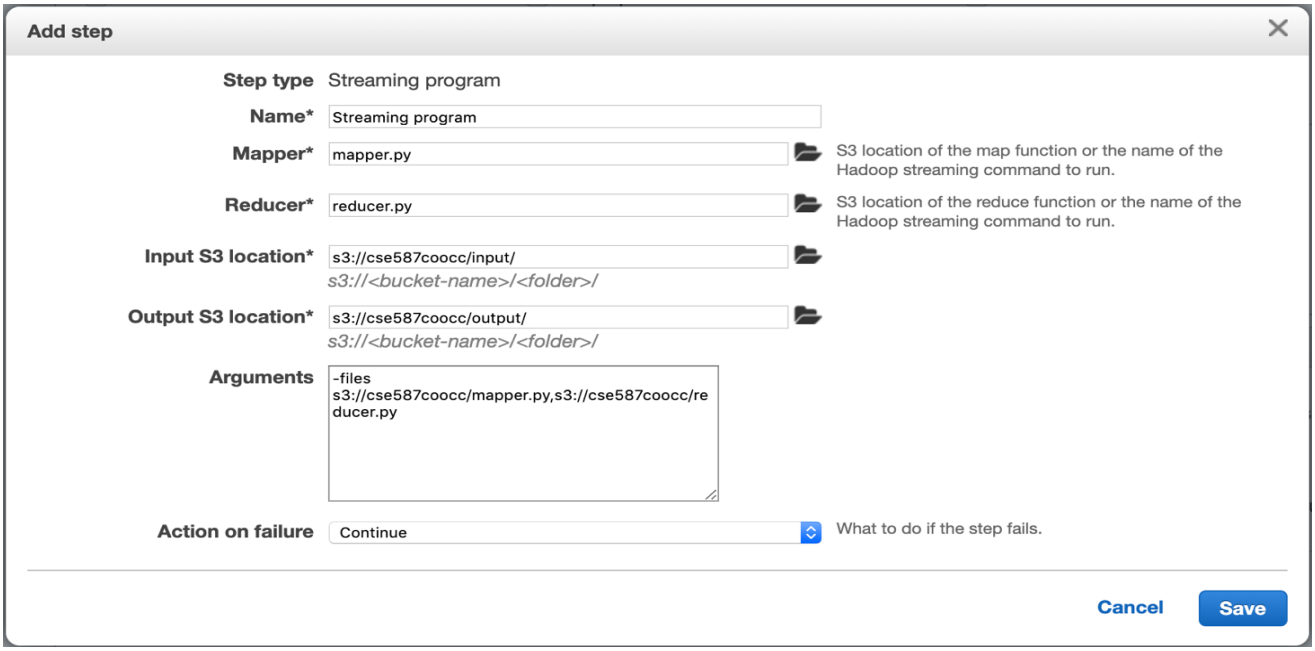

Figure 6 EMR Step

Once we are done with 4 steps of:

1. **Software and Steps**: In this step, we specify we are using emr – 2.23.0 and Core Hadoop: Hadoop 2.8.5 as Hadoop Application. We did not use AWS Glue Data Catalog for table metadata.

2. **Hardware**: For hardware we specify Instance type to be m4.large, we are updating this with the size of data we are feeding. We are keeping Number of instances as 2 for 1 master and 2 core nodes. We are also updating the number of instances based on out data.

3. **General Cluster settings**: In General Configuration, we are specifying the cluster name and turning on the Logging option mentioning the S3 bucket to store logs of our jobs.

4. **Security**: In Security and access we are going ahead with EC2 key pair and then keep the Permission to be Default. EMR role to be default EMR role with EC2 instance profile.

We can then create our cluster and the cluster would generate output/logs based on status of jobs executed. Once the cluster completes its looks something like Figure2:

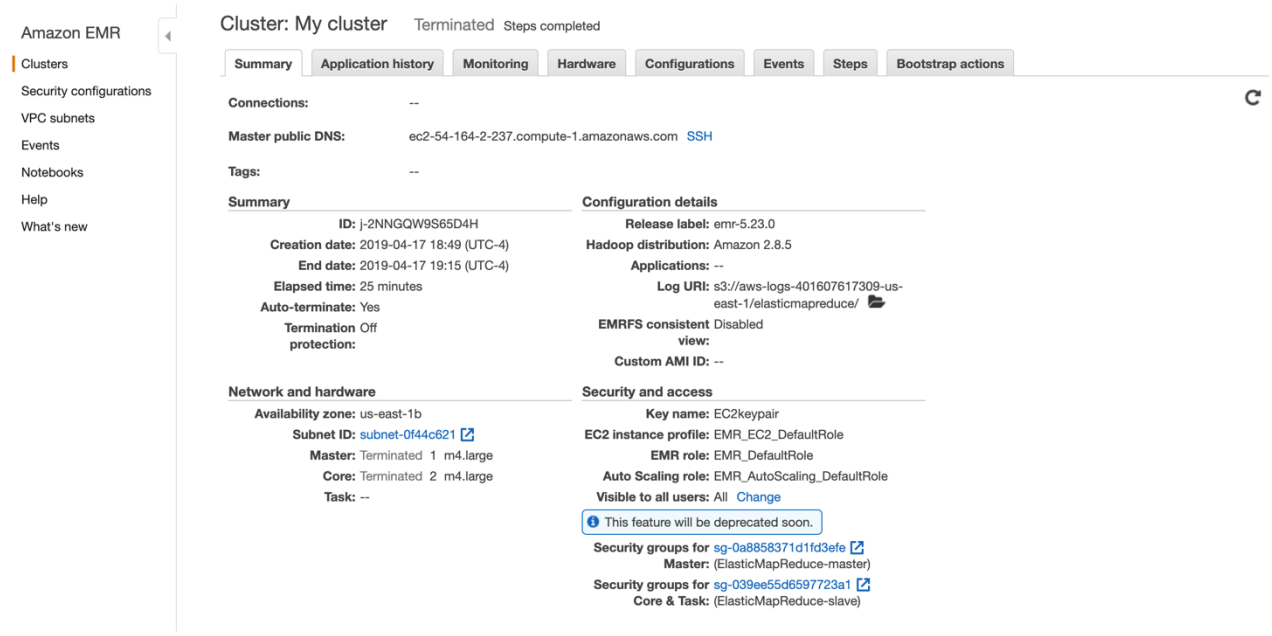

Figure 7 EMR Cluster Summary Page

Visualization

We are using Tableau to visualize the results that we got from the EMR output. We first download all the output files. Now no HDFS system actually adds the file extensions to the output files and therefore, we want to manually add the .txt extension to each of the file. After doing that, we start a new workbook in Tableau and first of all we want to get our data. We want to add a new data source and do this, which is shown as following.

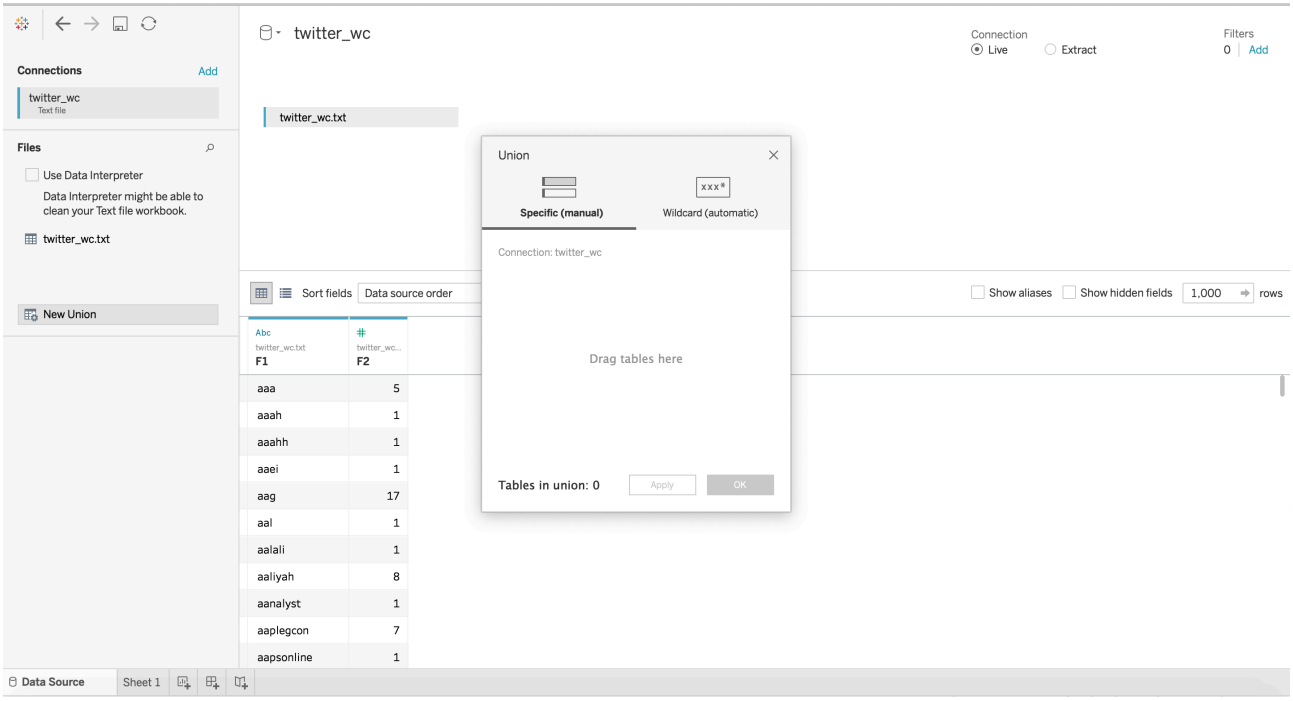

Figure 8 Add Data Source for Tableau

Once we have a data source by importing all the text files in a union, we want to create a sheet to graph the word cloud. We do this by using the Word and Count data columns. We will put the words in the rows part of the sheet as it is the core data and the Count, which is the measure, we add that to the Text Field in the Marks section. Once added, we can automatically see the word and its count showing, for the purpose of this image, it has been sorted by descending, which can be done by the button shown in the circle. The following image shows the Word added to row and Count to the Marks column.

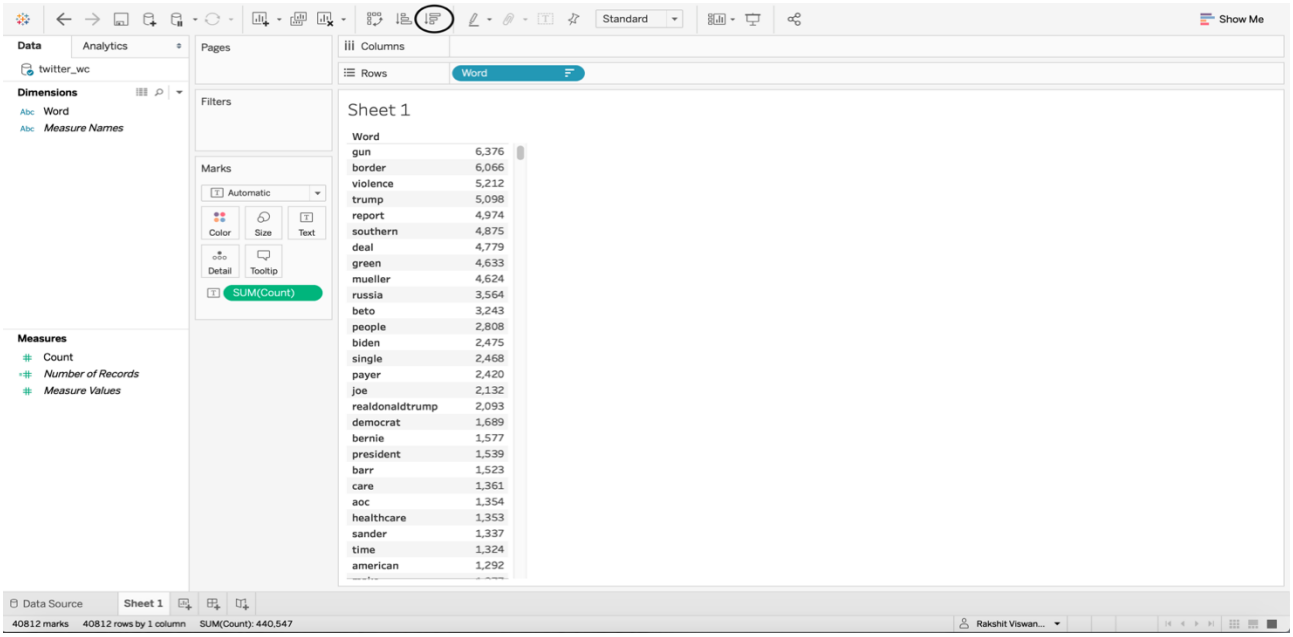

Figure 9 Sheets mode in Tableau

Once we have all the data added to the sheet, we can now create a word cloud, because the data will have a lot of words with just one count, we want to add a filter to see words that have a good count as that we expedite the word cloud and we do not need words with only one count. For the purpose of this image, we are using the minimum word count to be 500.

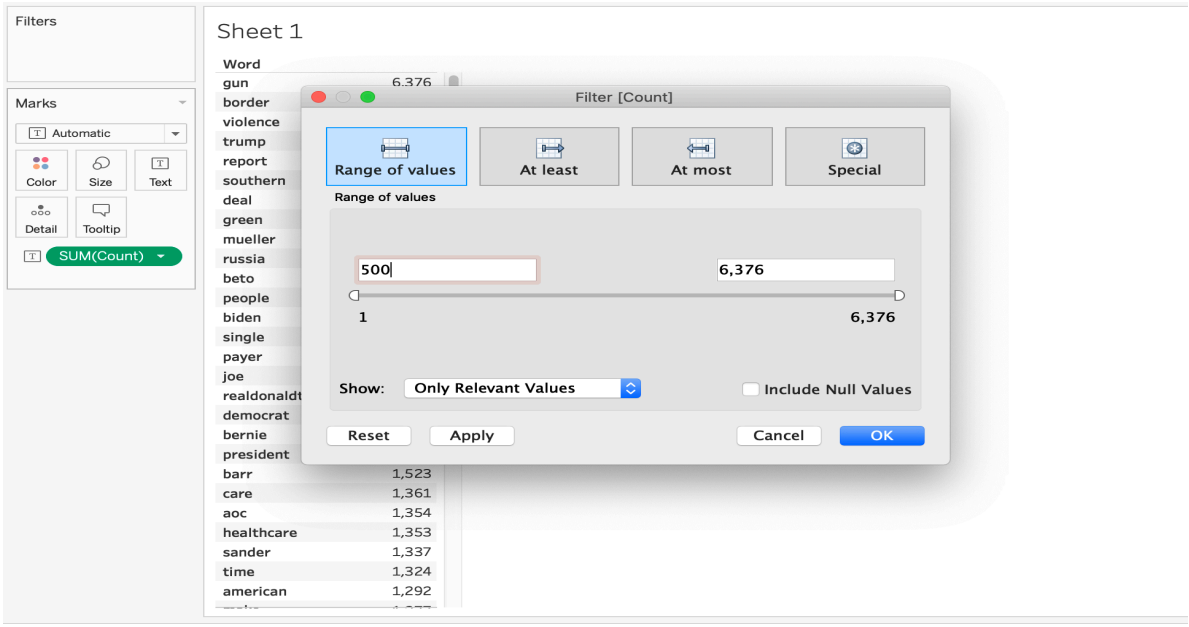

Figure 10 Filter to improve our word cloud

After adding this filter, we can then convert this to a word cloud. We do that by clicking on the Show Me, and picking the treemaps option. Once selected, it will give a cloud in boxes, we can convert that by changing the type from Automatic to Text in the dropdown. The following image illustrates all of that.

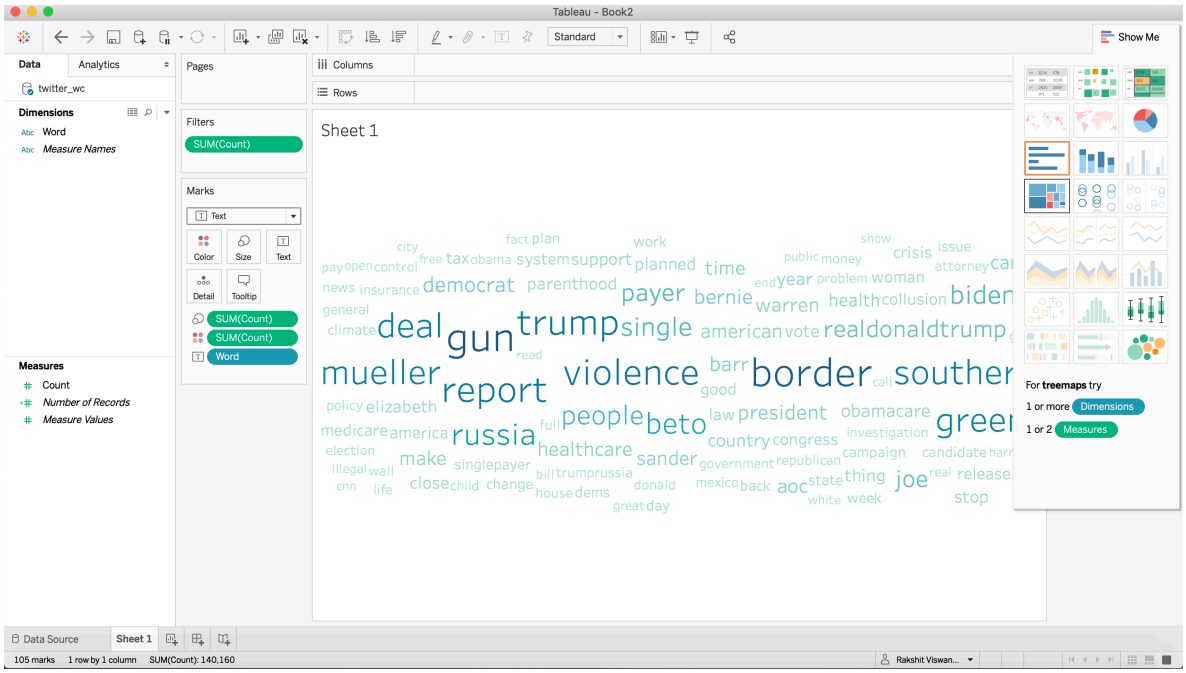

Figure 11 Shows how to convert the table to Word Cloud

We can right click on the Word button on the Rows and pick the filter option, then we can go to "Top" tab and select to find top by field and find the top 10 in the extracted data. The following image shows that.

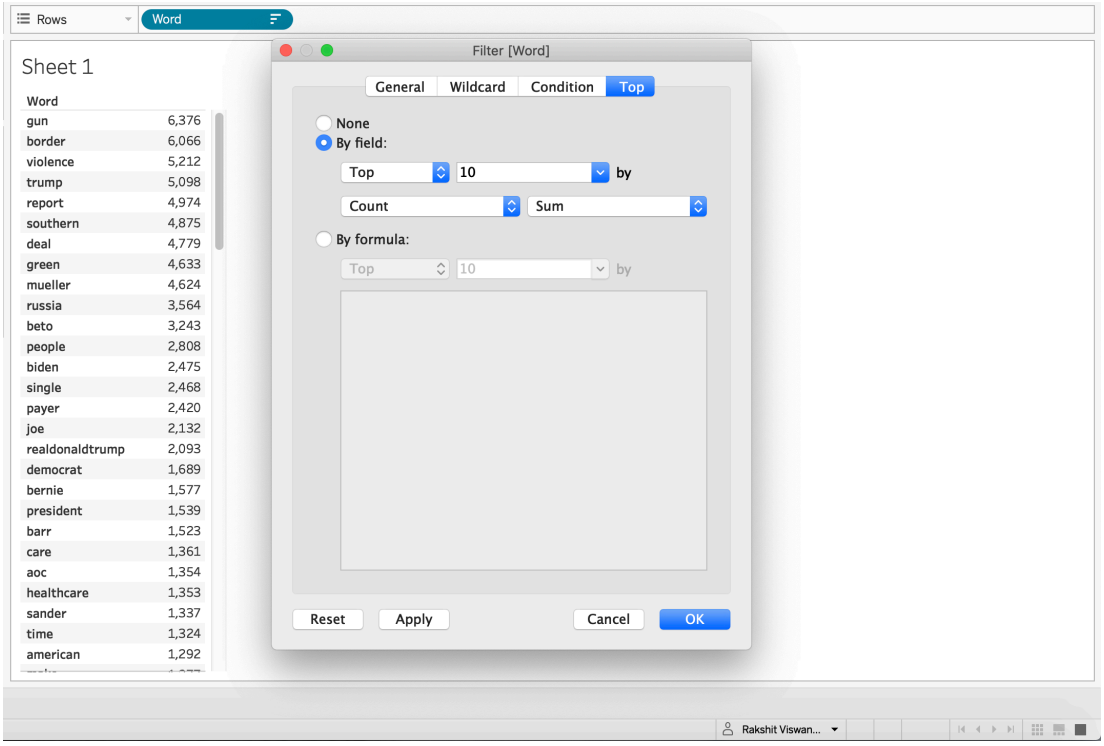

Figure 12 How to get Top 10

We can do similar for the co-occurrence, except when importing that data for that, we want to make sure the field separator is the tab and not the separation between 2 words in the pair. The following image shows that.

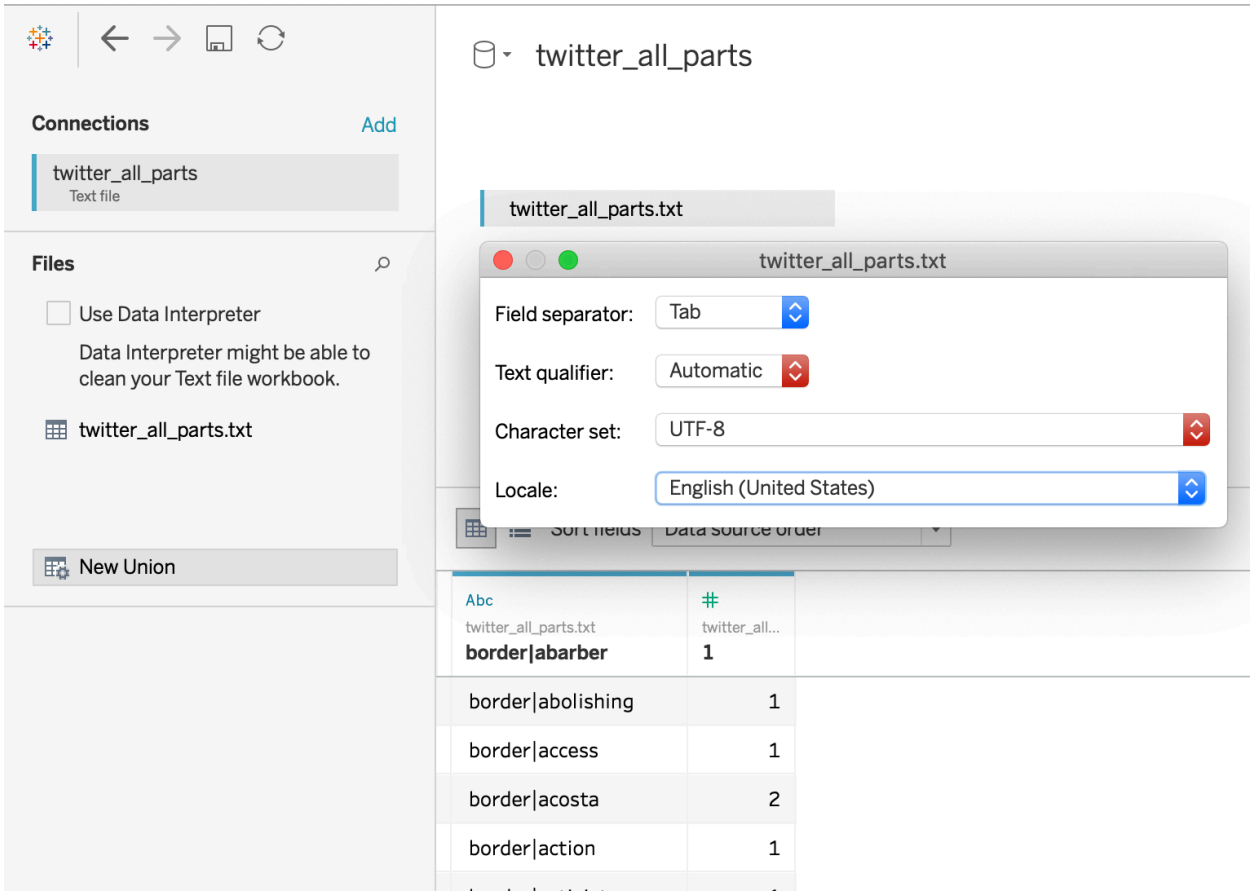

Figure 13 Changing field separators

After adding all the required word clouds, we can now create a story that will look like a PowerPoint presentation with live editing of the sheets. This way we can easily tab between the sheets and use the sheets and edit the data as well.

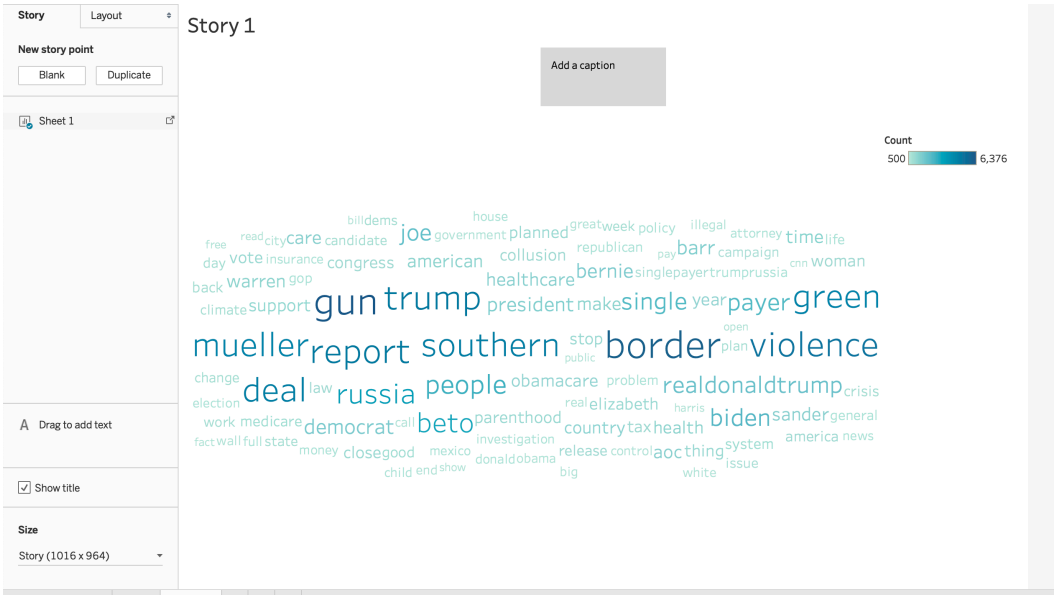

Figure 14 Story mode showing tabs for multiple sheets

Results

Twitter

For twitter we were able to find out all the required top 10 words from running the word cloud, which is shown below. The top 10 words do match all our keywords and reflects what is the main issue within our topic of discussion and what the people are actually talking about. We will then compare this to the NYT and Commoncrawl data and see how they behave. After getting the top 10 words, we then go on run the Co-occurrence MR on the same data so that we can find out the most co-occurring pairs based on the top 10 words. The top 10 words for Twitter are: "gun", "border", "violence", "trump", "report", "southern", "deal", "green", "mueller", "russia".

Twitter Word Count

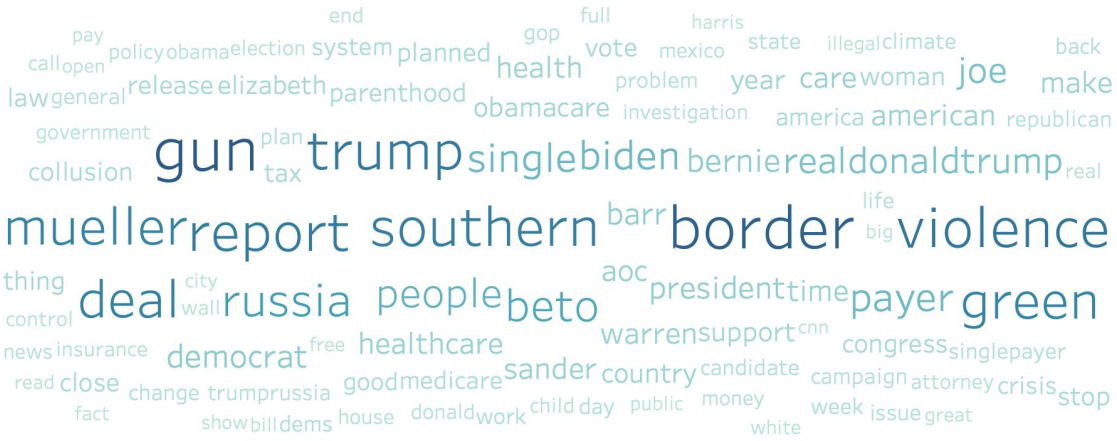

Figure 15 Twitter Word Count Word Cloud

Twitter Co-Occurence on Top 10 words

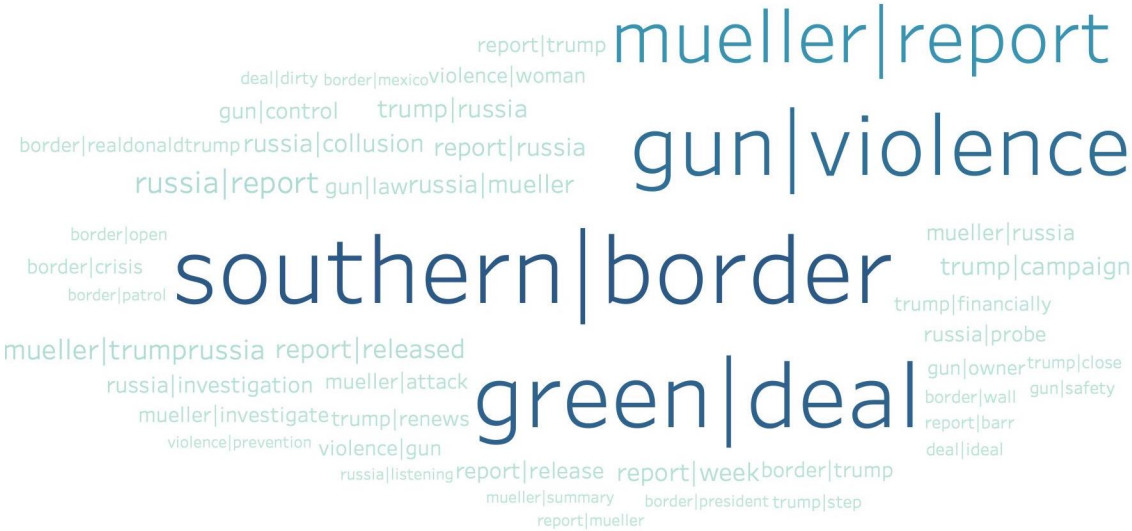

Figure 16 Co-occurrence Pairs for Twitter

New York Times

Similarly, we did the same for New York Times data and created the required word clouds. We then found out the top 10 words, which were mostly similar, but a little different as sometimes the news can report about a few things that the people do not want to talk about, or does not have the same public emotion. The top 10 words for New York Times are: "trump", "president", "mueller", "report", "democrat", "barr", "campaign", "state", "democratic", "micheal". Using these 10 words, we then run the co-occurrence MR to then find all the top pairs. The following images show that.

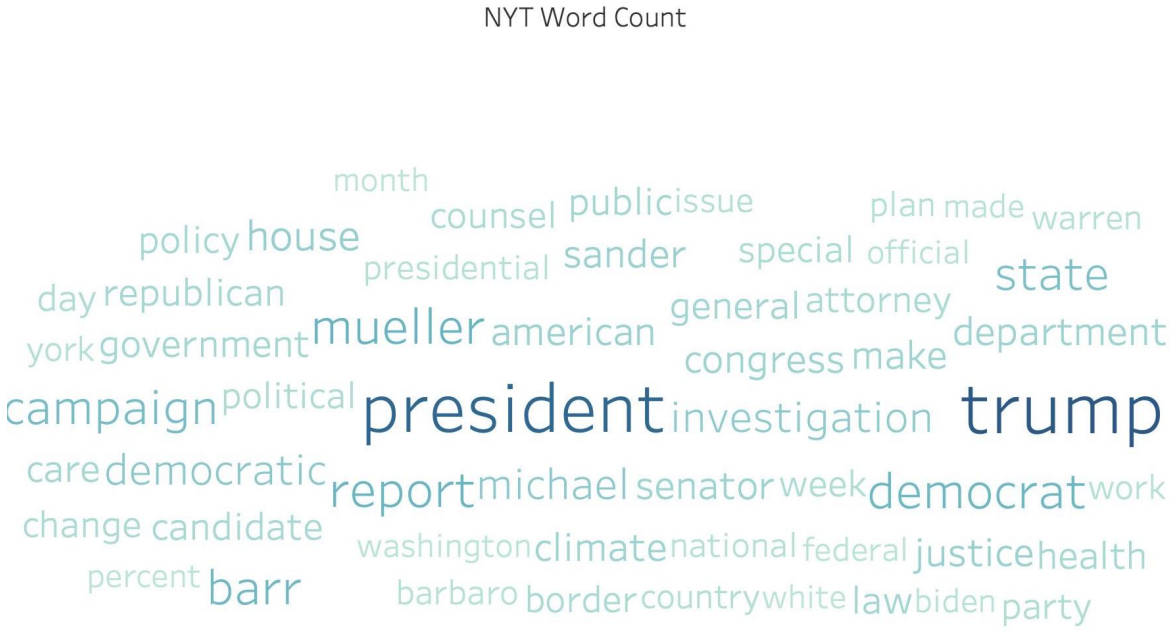

Figure 17 New York Times Word Count Word Cloud

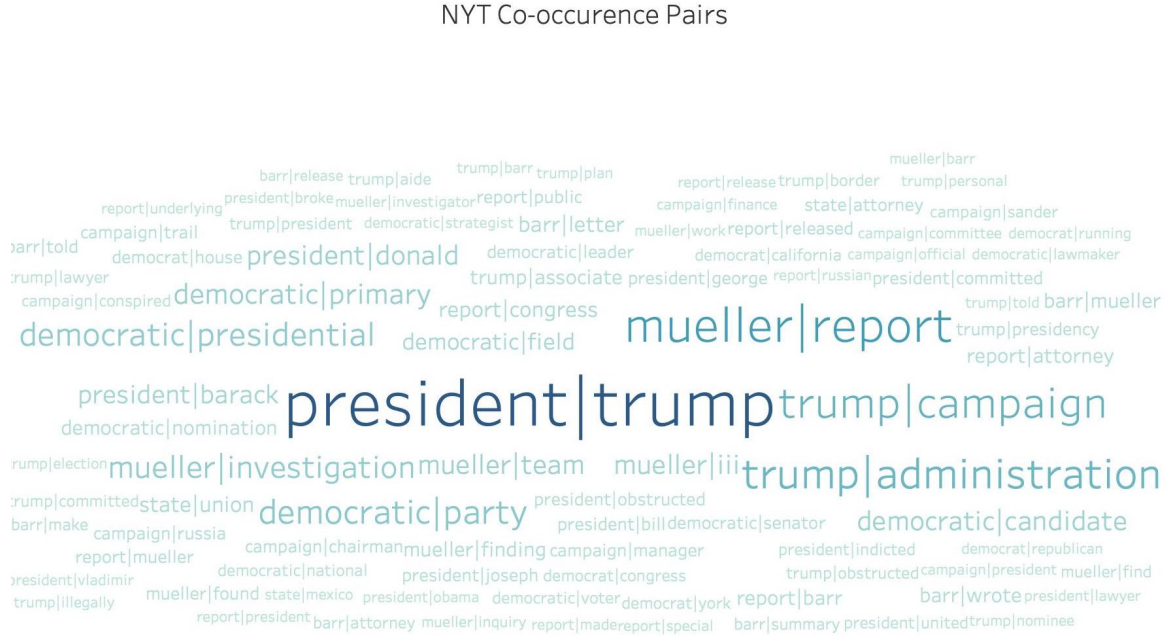

Figure 18 New York Times Co-occurring pairs

Commoncrawl

The commoncrawl data is noisier as we have exact filters as part of this lab, therefore, we had bare minimum resources and therefore can only gather so much helpful data. Based on our keywords, we can see that some topics definitely had more information than the others due to the constraints we had placed. The word cloud shows few keywords relevant to our topic keywords but the top 10 and most of the words are still relevant in an indirect way to our topic. With the top 10 words, we got the output of the co-occurrence MR and we can see the most of the pairs have something to do with our topic and therefore we are now sure that we have gathered the right data. Because of the lack of tightness of data, we can see a lot of words not matching our topic's keywords. The following is the Top 10 for commoncrawl: "trump", "health", "news", "service", "march", "day", "law", "state", "home", "business"

Figure 19 Commoncrawl Word Count

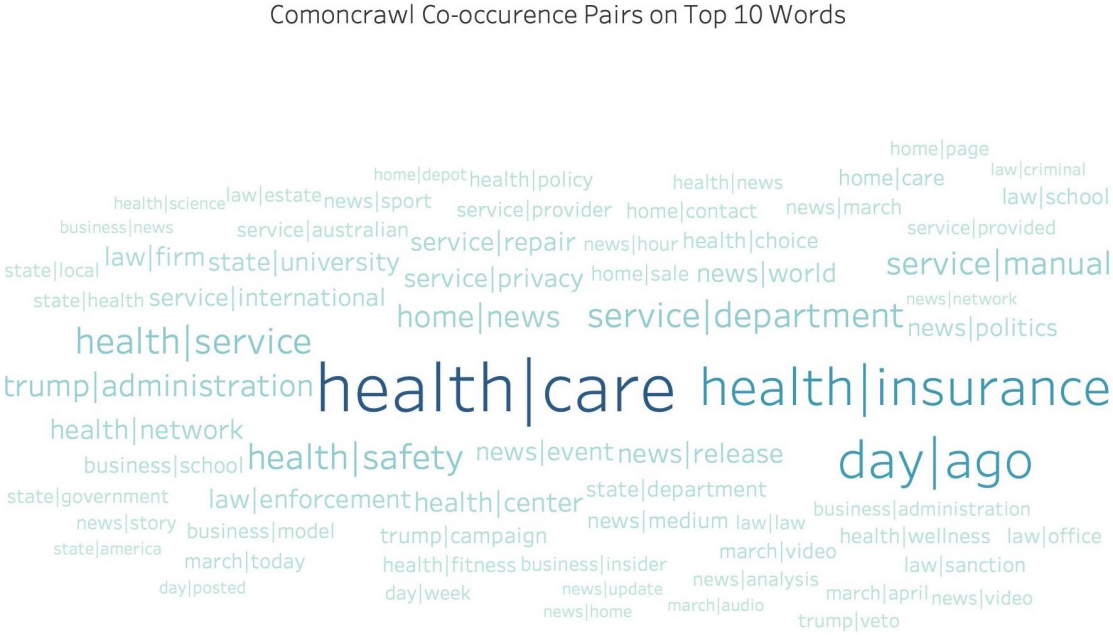

Figure 20 Commoncrawl Co-occurrence Pairs

Conclusion

We conclude with the fact that among all our data sources we have all the required keywords matching our topic and that the data get really loose for commoncrawl as it has the most noise. On the other hand, we can see across the board we can see that trump is a common topic among everything and that healthcare was another important topic for all the sources. In addition, this project has helped us understand how the HDFS system works and how it is different than a regular sequential system. We can also understand that with more resources, we can do more connectivity and see how the data can be immediately transmitted from one to the other software without any manual steps.