WEB SCIENCE COURSEWORK REPORT: 2418094

TABLE OF CONTENTS

INTRODUCTION: TWITTER

SOFTWARE DESCRIPTION

The software developed for Twitter web crawler uses JavaScript as the programming language with Node.js (v8.12.0) (NPM version: 6.4.1) as the run-time environment for execution of the program. The data collected is stored in the MongoDB which is preferred due to its flexibility and ease of use to store unstructured data in a JSON format. The various node package manager (NPM) modules have been used, all of MIT license, which are helpful is collecting Twitter data and performing the analysis on the collecting tweets. Various methodologies and filters have been used for collection of the Twitter data.

This software uses two APIs for collecting data from Twitter, namely REST API and STREAMING API.

The REST API is used to collect keyword filtered data with a maximum limit of 100 tweets per call. This restriction is due to Twitter restriction of the number of tweets per call, present in the NPM module used. The REST API calls have been made every 10 seconds for a period of 5 minutes (for collecting sample data) or 1 hour (for collecting more data). Some errors were encountered in making these many calls, which are Limit Message errors (from the error logs). After these warnings, the connection is automatically restored and tweets keep coming for the specified time duration.

The second type of API used is STREAMING API. This API opens an endless stream of tweets where real-time tweets keep coming for the specified duration of time (5 minutes or 1 hour). Few filters have been used in these streaming API, for instance, keyword filtering (same set of keywords used for REST API), location filtering (in bounding box coordinates) and one stream without any filtering of data.

The various NPM modules/libraries have been used for the software which are Body Parser, Cors, Express, Minhash, Mongoose, Plotly, Twit, Winston and Winston Daily Rotate File, Stopword (NPM, 2014).

After the data collection is done, the software performs analytics on the collected data which will be discussed further in the report. The analysis includes counting of total tweets, redundant tweets, geo-tagged tweets, etc. Then, the software collects all texts from the tweets and forms clusters according to the Minhas LSH algorithm and gives out information about geo-tagged and user profile based statistics from the created clusters. Various graphs have been generated as a part of the analysis whose URL can be found out from the final output logs of the software.

The output of the running of the software can be found in the directory Twitter-Crawler-Logs and the final/major outputs of the analytics can be found in the directory Twitter-Crawler-Final-Output-Log, both located in the crawl-server directory.

TIME DESCRIPTION

The data collected from various methods has been done so for different time durations. The REST API and STREAMING API have been used three times overall, running for 1 hour each for 2 times, once at approximately 04:08:08 on 10th November 2018 till 05:08:08 and the other time on 22:18:21 till 23:18:55 on 11th November 2018.

It was also run for multiple times for 10 minutes duration, 6 times to compare the amount of data collected for each slot over an hour.

The REST API calls have been made at 5 minutes interval for 1 hour as well as at 10 seconds interval for 1 hour, 10 minutes and 5 minutes duration. The STREAMING API calls have been used for 1 hour, 10 minutes and 5 minutes durations as well.

Finally, the software also collected data for a 5-minute window which is submitted as a sample data along with the code (sampleData.tar – in the directory crawl-server/model).

Code Sample For Rest Call Timer, first and second run of the software, and timer for the streaming data:

```
    //REST call using the random keyword from array
    getTweetsUsingREST.getTweetsREST(randomKeyword);
    //repeat the rest call after 5 minutes - See comment @Line 113
    // }, 300000);
    //Commenting above as REST call is tested with 10 seconds interval till 1 Hour with 12 rate limit exceeded errors
}, 10000);

//set the time after which the REST API calls should stop (1 hour)
setTimeout(function () {
    clearInterval(intervalForRestCall);
    // }, 3600000);
    //Done for small testing purpose - stopping it after 5 minutes
}, 300000);
```

```
Current data stats for twitterCrawlerDB:
Date: 10th November 2018
Start Point at: 2018-11-10 04:08:08 info: No keyword filter stream started
End Point at: 2018-11-10 05:08:08 info: Location stream ended.
Total runtime: 1 Hour
```

```
Current data stats for secondTwitterDB:
Date: 11th November 2018
Start Point at: 2018-11-11 22:18:21 info: No keyword filter stream started
End Point at: 2018-11-11 23:18:55 info: Total overlapping data found between Geo-tagged and No Filter Streamed Data: 6
Total runtime: 1 Hour      You, 7 days ago • Collected new larger data set with modifications and performed basic analyt
```

```
  //stop the streaming of data after 1 hour
  // setTimeout(stopLocationStream,3600000);
  //Done for small testing purpose to collect sample data of 5 Minutes
  setTimeout(stopLocationStream, 300000);
```

ACTUAL CODE ACCESS

The actual code base for this software can be found out at https://github.com/Kinshuk1993/twitterWebScience.

DATA CRAWL: TWITTER

STREAMING API

The streaming API used in this software have been of the NPM library TWIT. I have used streaming endpoints provided by Twitter, namely statuses/sample.

```
exports.getTweetsSTREAMNoFilter = function () {
    /**
     * logger to confirm that calling is happening as expected, but tweets not coming
     * in many executions of the code
     * at all from twitter due to restrictions
     */
    logger.info('No keyword filter stream started');
    //action on getting tweets from glasgow
    streamTwitterDataWithoutKeyword.on('tweet', function (tweet) {
        //log the content to the log file with user name
        logger.info('Tweet without any keyword filter received by user: ' + JSON.stringify(tweet.user.screen_name));
        //save the incoming tweet to the database
        TweetsDB.tweetsSTREAMNoFilter(tweet).save(function (err, savedTweet) {
            //handle error case
            if (err) {
                //If error, log the error
                logger.error('Error occured in saving tweet by user ' + JSON.stringify(tweet.user.screen_name) + ' with no filtering to database: ' + JSON.stringify(err))
            } else {
                //Log the success message of saving to database
                logger.info('Tweet by user ' + JSON.stringify(tweet.user.screen_name) + ' with no filtering saved to database');
            }
        });
    });

    //action on getting limit messages without any keyword filter stream
    streamTwitterDataWithoutKeyword.on('limit', function (limitMessage) {
        //log the content to the log file
        logger.info('Limit message for no keyword filter stream received: ' + JSON.stringify(limitMessage));
    });

    //action on getting disconnect messages without any keyword filter stream
    streamTwitterDataWithoutKeyword.on('disconnect', function (disconnectMessage) {
        //log the content to the log file
        logger.warn('Disconnect message for no keyword filter stream received: ' + JSON.stringify(disconnectMessage));
    });

    //action on getting error messages without any keyword filter stream
    streamTwitterDataWithoutKeyword.on('error', function (errMsg) {
        //log the content to the log file
        logger.error('Error message for no keyword filter stream received: ' + JSON.stringify(errMsg));
    });
```

The sample end-point is used for streaming of data without any filter of keyword or location. The filter end-point is used for streaming of data with keyword filtering and the location filtering data. The keyword filtering uses an array of keywords which have been termed as trending topics on Google and Twitter by the google search (Hudgens, 2016) (Mondovo, 2018).

This streaming API without any filter starts as soon as the program starts and runs for as long as it is needed. This is because one can modify the time period it should run for in the code by commenting out. The time period is mentioned in milliseconds.

The data collected using this is stored in the database in a collection named as tweetsstreamnofilters. The total data collected using no filter stream for one hour in 2 different runs were 118151 and 136885 on 10th and 11th November 2018 respectively.

ENHANCING CRAWLING USING STREAMING AND REST API

The data collected using no filter stream has been further enhanced by using rest and enhanced streaming API. The streaming endpoint used here is statuses/filter and search/tweets are used for REST API.

An array of most common and trending keywords have been used for making rest and streaming calls.

The REST API uses random keywords from the array for each call whereas the streaming API uses all of those keywords for filtering out tweets containing those keywords.

REST calls are made in two ways, one every 5 minutes for 1 hour, where only 1287 tweets were collected in an hour and the other at every 10 seconds for 1 hour where 33792 tweets were collected.

```
// call the REST API every 5 minutes for a total duration of 1 hour
var intervalForRestCall = setInterval(function () {
    //work on a sample array as JS will modify original array otherwise
    var sampleArrayKeyword = []
    //fill data into sample array
    for (var i = 0; i < keywordArray.length; i++) {
        sampleArrayKeyword.push(keywordArray[i])
    }
    //variable to store the random keyword to search for using REST call
    //defaulted to keyword "WEATHER"
    var randomKeyword = "oneplus";
    //each time pull from that array and remove the one already used
    //simple check to check if contents of array
    if (sampleArrayKeyword.length > 0) {
        //get a random index of sample array
        var randomIndex = Math.floor(Math.random() * sampleArrayKeyword.length)
        //get word on the generated index
        randomKeyword = sampleArrayKeyword[randomIndex];
        //log the action
        logger.info('Keyword being searched for via the twitter REST call is: ' + randomKeyword);
        //remove the word from the array to avoid duplicate keyword search using REST call
        //Commenting this line as the frequency for REST call is increased and hence cannot clear the keyword array with every call
        // sampleArrayKeyword.splice(randomIndex, 1);
    }
    //REST call using the random keyword from array
    getTweetsUsingREST.getTweetsREST(randomKeyword);
    //repeat the rest call after 5 minutes - See comment @Line 113
    // }, 300000);
    //Commenting above as REST call is tested with 10 seconds interval till 1 Hour with 12 rate limit exceeded errors
}, 10000);

//set the time after which the REST API calls should stop (1 hour)
setTimeout(function () {
    clearInterval(intervalForRestCall);
    // }, 3600000);
    //Done for small testing purpose - stopping it after 5 minutes
}, 300000);
```

The data collected using streaming API are also done in two ways, one with only the keyword “Morning” which yielded 26437 tweets and for the second time, with the keyword array filter, which resulted in 172802 tweets. Both times, streaming was open for a 1-hour duration.

```
exports.getTweetsSTREAMKeywordFilter = function () {
    /**
     * logger to confirm that calling is happening as expected, but tweets not coming
     * in many executions of the code
     * at all from twitter due to restrictions
     */
    logger.info('Keyword filter stream started');
    //action on getting tweets using keywords filter stream
    streamTwitterDataKeyword.on('tweet', function (tweet) {
        //log the content to the log file with user name
        logger.info('Tweet with filter keyword received by user: ' + JSON.stringify(tweet.user.screen_name));
        //save the incoming tweet to the database
        TweetsDB.tweetsSTREAMKeywordFilter(tweet).save(function (err, savedTweet) {
            //handle error case
            if (err) {
                //If error, log the error
                logger.error('Error occured in saving tweet by user ' + JSON.stringify(tweet.user.screen_name) + ' with keyword filtering to database: ' + JSON.stringify(
            } else {
                //Log the success message of saving to database
                logger.info('Tweet by user ' + JSON.stringify(tweet.user.screen_name) + ' with keyword filtering saved to database');
            }
        });
    });

    //action on getting limit messages on keyword filter stream
    streamTwitterDataKeyword.on('limit', function (limitMessage) {
        //log the content to the log file
        logger.info('Limit message for keyword stream received: ' + JSON.stringify(limitMessage));
    });

    //action on getting disconnect messages on keyword filter stream
    streamTwitterDataKeyword.on('disconnect', function (disconnectMessage) {
        //log the content to the log file
        logger.warn('Disconnect message for keyword stream received: ' + JSON.stringify(disconnectMessage));
    });

    //action on getting error messages on keyword filter stream
    streamTwitterDataKeyword.on('error', function (errMsg) {
        //log the content to the log file
        logger.error('Error message for keyword stream received: ' + JSON.stringify(errMsg));
    });
```

For further analysis purpose, the REST and STREAMING calls were made for 10 minutes duration as well as for 5 minutes duration (for sample data submission).

The enhancement made is clear from the amount of data collected which can be seen from the below screenshot (from app.js – captured during the development).

```
* ##########################################################################################################################################################################
* Current data stats for twitterCrawlerDB:
* Date: 10th November 2018
* Start Point at: 2018-11-10 04:08:08 info: No keyword filter stream started
* End Point at: 2018-11-10 05:08:08 info: Location stream ended.
* Total runtime: 1 Hour
*
* REST Tweets Collected: 1287 ( REST Calls @ 5 Minutes Interval) (1687 after some more time)
* No Filter Stream Tweets Collected: 118151 (118255 after some more time)
* Tweets collected with Keyword Filtering using the word "MORNING": 26437 (29816 after some more time)
* Glasgow Geo-tagged Tweets Collected: 44 (48 after some more time)
* ##########################################################################################################################################################################
* Current data stats for secondTwitterDB:
* Date: 11th November 2018
* Start Point at: 2018-11-11 22:18:21 info: No keyword filter stream started
* End Point at: 2018-11-11 23:18:55 info: Total overlapping data found between Geo-tagged and No Filter Streamed Data: 6
* Total runtime: 1 Hour
*
* REST Tweets Collected: 33792 ( REST Calls @ 10 Seconds Interval with 12 error (extract the 11-11-2018-1.tar file and search "error:") received due to Rate Limit Exceeded)
* No Filter Stream Tweets Collected: 136885
* Tweets collected with Keyword Filtering using the words in keyword array: 172802
* Glasgow Geo-tagged Tweets Collected: 512
* ##########################################################################################################################################################################
```

GEOTAGGED DATA COLLECTION FOR GLASGOW FOR SAME TIME PERIOD

The streaming API is used for one more purpose which is to filter the incoming data according to the location coordinates for Glasgow. These coordinates have been used to create a bounding box for Glasgow, hence enabling incoming tweets to be from Glasgow location only.

The coordinates have been collected from the google search from http://boundingbox.klokantech.com/ (Klokantech, 2018).

These coordinates have then been used in the code with endpoint as statuses/filter with the parameter as the coordinates obtained. This stream is also run similar to other streams (keyword and no filter) and also for same time duration.

```
//Glasgow coordinates - Obtained using http://boundingbox.klokantech.com/
var glasgow = ['-4.393201', '55.781277', '-4.071717', '55.929638'];

/**
 * get a stream of glasgow tweets
 * with end point as statuses/filter
 */
var streamTwitterDataLocation = twitAuth.stream('statuses/filter', {
    locations: glasgow
});

exports.getTweetsSTREAMLocationFilter = function () {
    /**
     * logger to confirm that calling is happening as expected, but tweets not coming
     * in many executions of the code
     * at all from twitter due to restrictions
     */
    logger.info('Location filter stream started');
    //action on getting tweets from glasgow
    streamTwitterDataLocation.on('tweet', function (tweet) {
        //log the content to the log file with user name
        logger.info('Tweet from Glasgow received by user: ' + JSON.stringify(tweet.user.screen_name));
        //save the incoming tweet to the database
        TweetsDB.tweetsSTREAMLocationFilter(tweet).save(function (err, savedTweet) {
            //handle error case
            if (err) {
                //If error, log the error
                logger.error('Error occured in saving tweet by user ' + JSON.stringify(tweet.user.screen_name) + ' to database: ' + JSON.stringify(err));
            } else {
                //Log the success message of saving to database
                logger.info('Tweet by user ' + JSON.stringify(tweet.user.screen_name) + ' from Glasgow saved to database');
            }
        });
    });

    //action on getting limit messages on location stream
    streamTwitterDataLocation.on('limit', function (limitMessage) {
        //log the content to the log file
        logger.info('Limit message for location stream received: ' + JSON.stringify(limitMessage));
    });

    //action on getting disconnect messages on location stream
    streamTwitterDataLocation.on('disconnect', function (disconnectMessage) {
```

The data collected during two periods of 1 hour on 2 days are 44 and 512 respectively. The sample data submitted along with the code has 64 tweets during the time it was collected.

DATA ACCESS STRATEGY AND RESTRICTIONS FROM TWITTER AND TWIT MODULE

The data access from Twitter is done via the Twit Module which is installed via npm (ttezel, 2018).

Twitter has a restriction of a maximum number of REST calls that can be done in a 15-minute window, which is handled by the module as well by the software developed.

The way in which the Twit module handles the restriction is that, whenever the warning of limit exceeded message comes, it re-establishes the connection to Twitter and tries to fetch tweets again till it successfully gets a response from Twitter. In this software, the

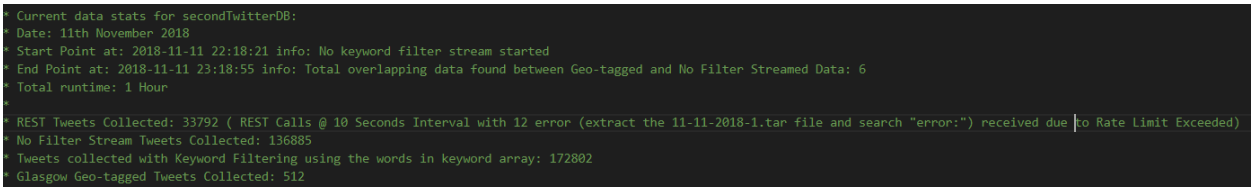
code is done in such a way that it makes the rest calls every 10 seconds (which initially was 5 minutes interval). This shift of 5 minutes to 10 seconds is done so that more data can be collected from the REST API calls.

The streaming API do not have such restrictions, as, during the testing of the API, it went on for 3 hours without stopping or throwing any error in the logs.

One restriction in using all 4 ways of collecting data (1 REST, 3 Streams) is that Twitter sometimes does not send data on either of the APIs and one has to terminate the program manually and start again – sometimes keep doing it for 4-5 times to get all tweets from all 4 API calls. But once that starts happening, it works flawlessly without stopping till manually stopped.

The software uses an array of top trending keywords (obtained after google search – for both Twitter and Google). This increases the possibility of gathering a very large amount of data, especially considering the different REST calls going through every time with a different keyword search. The same keyword array is also used for streaming tweets with keyword filtering, where tweets matching any keyword present in the array is received.

This approach has actually enhanced the tweet count and the amount of data collected as is evident from the below screenshot where the amount of data collected using keyword filtering stream is more than no filter stream data count.



BASIC DATA ANALYTICS: TWITTER

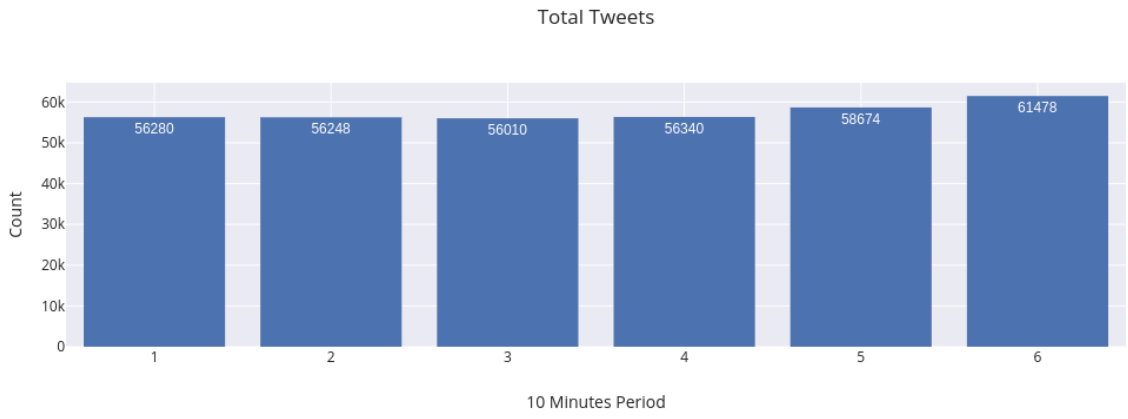
The amount of data collected varies on the time and day of the week where the data is collected. It also varies according to the type of filtering used, a number of keywords used for filtering and also on the number of API calls done in a fixed period of time.

TOTAL DATA COLLECTED

The total amount of data collected using all the REST calls and all 3 STREAM types is different for different runs. Total data collected during the first run was 145919. This was the data when keyword used in REST call was “Morning” and calls were made at 5 minutes interval for 1 hour.

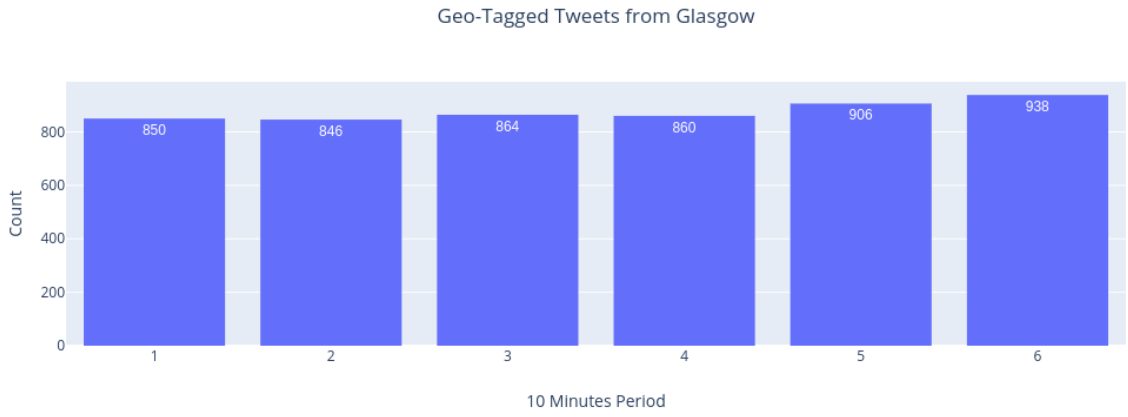
During the second run, the total data collected was 343991 when the rest calls were made at 10 seconds interval for 1 hour and the keyword array was used for filtering keyword search.

During the 10 minutes interval, all data were collected for 6 times and the following graph was obtained:



TWEETS FROM GLASGOW

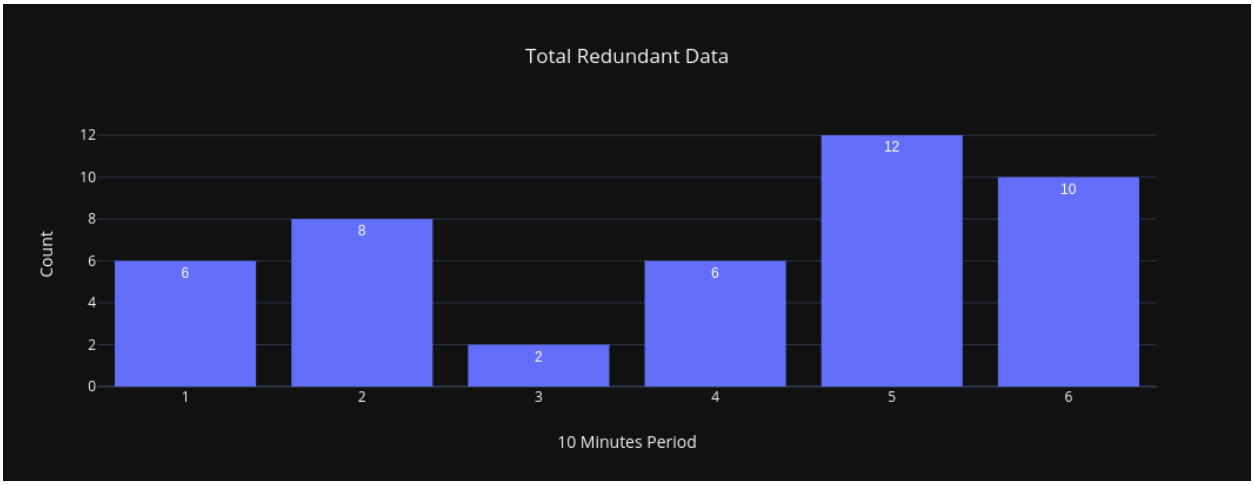
The total data collected from Glasgow over the different 10-minute interval in an hour using all 4 APIs are shown below.



It may be concluded from the above graph that not many users are present who have enabled geo-information in their tweets. This conclusion is done on comparing the data of total tweets received.

TOTAL REDUNDANT DATA

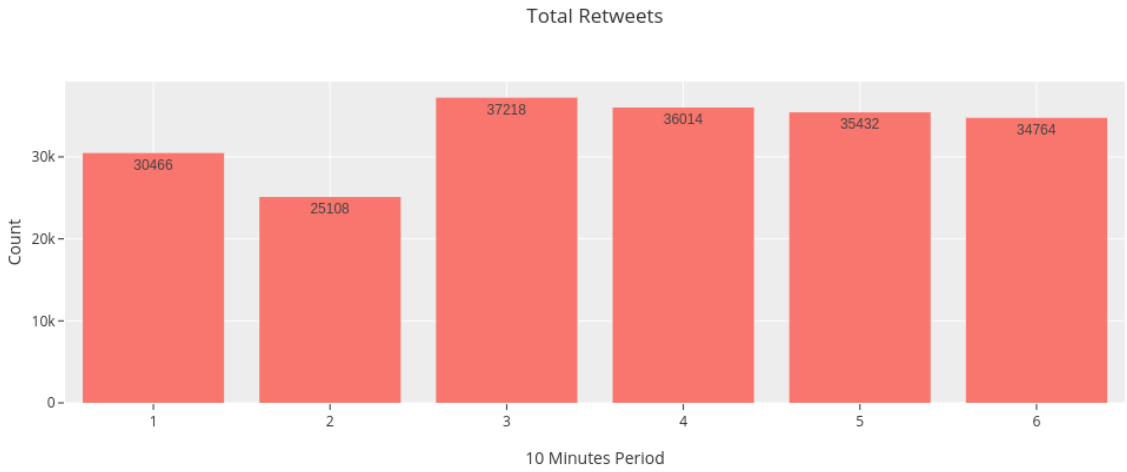
Since the data collection mechanism involves 4 different ways, namely, REST calls, and 3 streams (unfiltered, keyword filtered and location filtered), it is very much possible that the same tweet may have been captured by two or more APIs. The redundant data collected is depicted from the below graph:



From the above graph, it may be concluded that the mechanism to collect data works well, as the redundant data collected is almost negligible, especially considering the total tweets collected for every 10 minute period. Hence, most of the data collected from this software are unique in nature.

RE-TWEETS AND QUOTES

The number of retweets collected for 10 minute period over one hour is evident from the below bar graph:



And, the number of tweets collected which have quotes in them are analyzed as below:

From the above 2 graphs, it is conclusive that, almost half of the total tweets or more (in some cases) are retweets and not the new tweets. Also, a fair amount of tweets have quotes contained in them when the total tweets for the same period are taken into consideration.

ENHANCING THE GEOTAGGED DATA: TWITTER

GROUPING OF TWEETS: CLUSTERING

The main idea of collecting all tweets is to group them in a way that the similar tweets according to the text are placed together in a single group. This clustering is done by this software by an algorithm called Minhash Locality Sensitive Hashing. This algorithm is generally used for finding the similarity between sets, which are the texts in our use case. It is mainly used for finding a similar set of text and clustering them together. Jaccard similarity coefficient and minimum hash values are used to find the 2 similar text samples (Wikipedia, n.d.).

We use an implementation of this algorithm provided by the “Minhash” npm module. This software first takes all texts and removes all emojis and then creates an array of each text split by the space. The npm module named as “Stopword” is also used to remove the noisy words from the word array (Fergie, 2018). This is done for different languages to ensure the clusters formed are only done on the basis of important words.

```
async.forEachSeries(totalTweets, function (each, callback) {
        //create array of words for each tweet text
        var textSplit = each.text.split(' ');
        //remove the noise of different languages          Kinshuk Bhardwaj [57 minutes
        //remove arabic
        var updatedText = removeNoise.removeStopwords(textSplit, removeNoise.ar);
        //remove bengali
        var updatedText = removeNoise.removeStopwords(updatedText, removeNoise.bn);
        //remove Brazilian Portuguese
        var updatedText = removeNoise.removeStopwords(updatedText, removeNoise.br);
        //remove Danish noise
        var updatedText = removeNoise.removeStopwords(updatedText, removeNoise.da);
        //remove German noise
        var updatedText = removeNoise.removeStopwords(updatedText, removeNoise.de);
        //remove English noise
        var updatedText = removeNoise.removeStopwords(updatedText, removeNoise.en);
        //remove Spanish noise
        var updatedText = removeNoise.removeStopwords(updatedText, removeNoise.es);
        //remove Farsi noise
        var updatedText = removeNoise.removeStopwords(updatedText, removeNoise.fa);
        //remove French noise
        var updatedText = removeNoise.removeStopwords(updatedText, removeNoise.fr);
        //remove Hindi noise
        var updatedText = removeNoise.removeStopwords(updatedText, removeNoise.hi);
        //remove Italian noise
        var updatedText = removeNoise.removeStopwords(updatedText, removeNoise.it);
        //remove Japanese noise
        var updatedText = removeNoise.removeStopwords(updatedText, removeNoise.ja);
        //remove Dutch noise
        var updatedText = removeNoise.removeStopwords(updatedText, removeNoise.nl);
        //remove Norwegian noise
        var updatedText = removeNoise.removeStopwords(updatedText, removeNoise.no);
        //remove Polish noise
        var updatedText = removeNoise.removeStopwords(updatedText, removeNoise.pl);
        //remove Portuguese noise
        var updatedText = removeNoise.removeStopwords(updatedText, removeNoise.pt);
        //remove Punjab gurmukhi noise
        var updatedText = removeNoise.removeStopwords(updatedText, removeNoise.pa_in);
        //remove Russian noise
        var updatedText = removeNoise.removeStopwords(updatedText, removeNoise.ru);
        //remove Swedish noise
        var updatedText = removeNoise.removeStopwords(updatedText, removeNoise.sv);
        //remove Chinese simplified noise
        var updatedText = removeNoise.removeStopwords(updatedText, removeNoise.zh);
```

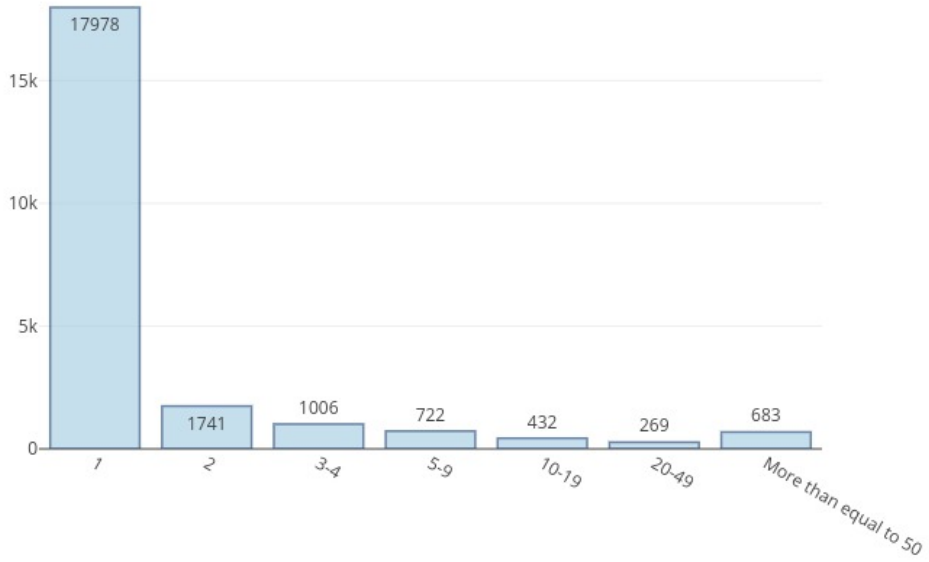
It then creates a hash for each set and updates it based on the existing hashes. Then all hashes are saved in an array. Then each of the text hash created is queried in the LSH index to find and group similar text sets into a single cluster. If a text has no similarity, then it is placed in a cluster with just one element.

```
//5. Function to work with minhash lsh and create hashes and indexes
function (totalTweets, callback) {
    var mArr = [];
    var count = 0;
    async.forEachSeries(totalTweets, function (each, callback) {
            //create array of string
            var textSplit = each.text.split();
            //new variable
            var m = 'm' + count;
            //increment count for next variable
            count++;
            //new hashing
            m = new Minhash();
            //update each sentence
            textSplit.map(function (w) {
                m.update(w);
            });
            //add hashes to the index after updation
            index.insert(each.id + "-" + each.place + "-" + each.geoEnabled, m);
            //add to m to late to compute closeness for each sentense
            mArr.push(m);
            //go to next iteration
            callback();
        },
        //final function for async
        function (errFinal) {
            if (errFinal) {
                //log error
                logger.error('Error in minhash calculation asynch final: ' + errFinal);
                //go to next function
                callback(null, totalTweets, mArr);
            } else {
                logger.info('kMeansClustering: Going to function 6 to form clusters using minhash lsh');
                //go to next function
                callback(null, totalTweets, mArr);
            }
        });
},
```

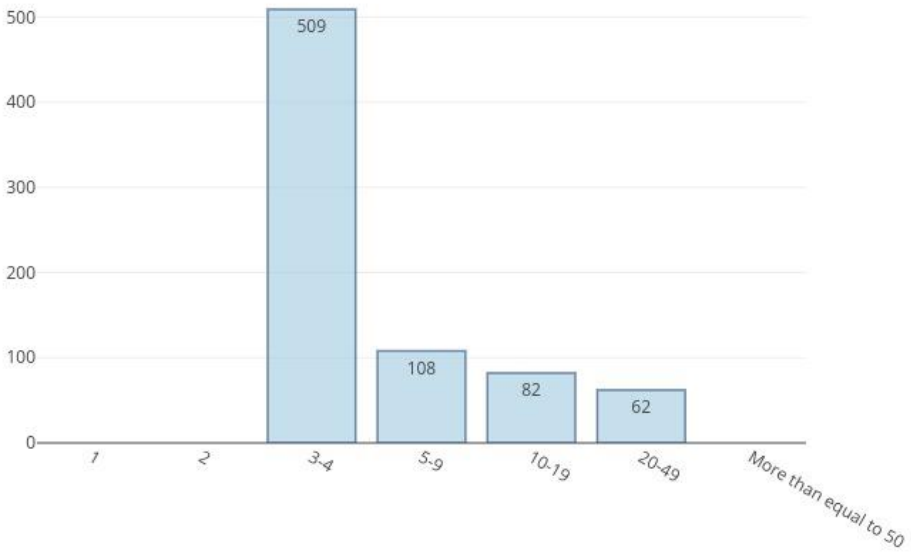
After the clustering is done, duplicate ones containing the same indexes are removed and unique ones are analyzed. The following graph shows the distribution of the clusters made with the number of elements in it.

```
//6. Function to perform query of lsh of each sentense
function (totalTweets, mArr, callback) {
    //store all clusters for each sentence
    var allMatches = [];
    //variable to store unique clusters
    var allMatchesUnique = [];
    //loop through all indexes formed
    async.forEachSeries(mArr, function (eachM, callback) {
            //find clusters
            var match = index.query(eachM);
            //form a cluster of arrays
            allMatches.push(match);
            //go to next index
            callback();
        },
        //final function for async
        function (errFinal) {
            if (errFinal) {
                //log error
                logger.error('Error matches in forming clusters based on minhash LSH: ' + errFinal);
                //go to next function
                callback(null, totalTweets, allMatchesUnique);
            } else {
                //convert the arrays to JSON strings and use a Set to get unique values
                var set = new Set(allMatches.map(JSON.stringify));
                //convert back to array of arrays again
                allMatchesUnique = Array.from(set).map(JSON.parse);
                logger.info('kMeansClustering: Going to function 7 to form the cluster graph');
                //go to next function
                callback(null, totalTweets, allMatchesUnique);
            }
        });
},
```

The below graph depicts that most of the groups have no similar tweets. Others have the size of the cluster as two or three and few have more than 10 as the size of the cluster.

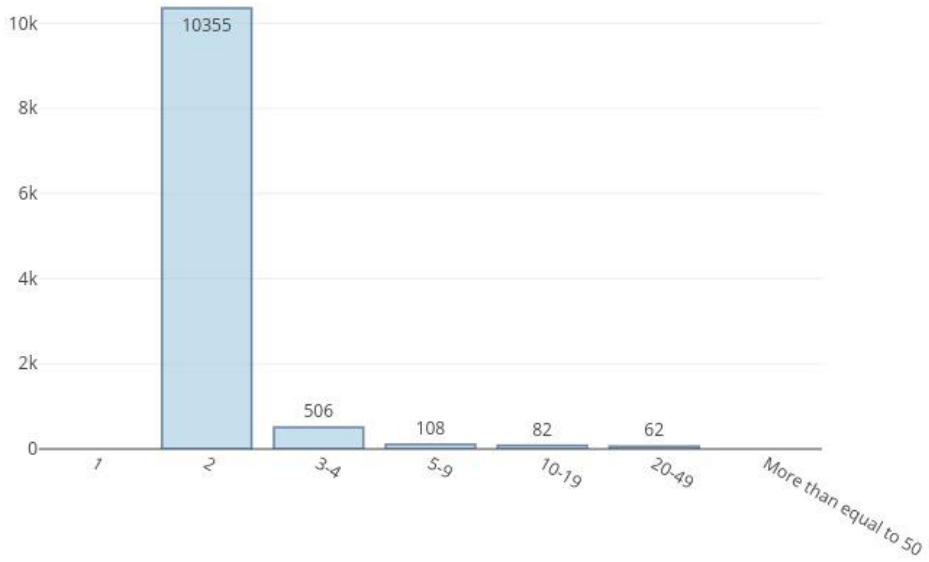


The next graph is showing the number of geo-tagged clusters among the clusters formed and their distribution among various sizes of the cluster.



It may be concluded that very few users are there who tweet with their geolocation enabled. This is in line with the conclusion made earlier with the total geo-location enabled data collected.

The following graph shows the distribution of the user profile geo information (the users who have geo_enabled as true in their profile, meaning their profile location) among the clusters formed.



From the above 2 graphs, it may be concluded that a large number of users have their profile location on, but their geo-location for tweets is not enabled. This again coincides with the conclusion that very few geo-tagged tweets exist from the total tweets collected.

Thus, it is very much possible that profile information may be enabled for a user, which for example may be shown as Glasgow, but the geo-tagged location shows the user tweeting from another location in the world, for instance, London.

GEO-LOCATION ASSIGNMENT TO CLUSTERS

The assignment of a geo-location to different clusters depends on the fact that if any tweet from the cluster has a location present within it or not. If no information is present, that cluster is not assigned any location.

The method used by this software to assign a cluster a location is right now using the given information in a way that it gives out the information about if a cluster belongs to the location Glasgow or not. The logic used in assigning this information is that, if a tweet is from Glasgow, then its place information would be Glasgow, which in turn may or may not have any coordinates assigned to it. A cluster is said to be belonging to the location Glasgow only on the following conditions:

1. If all tweets in that cluster are from Glasgow.
2. If more than 50% of the tweets in a cluster belong to location Glasgow.

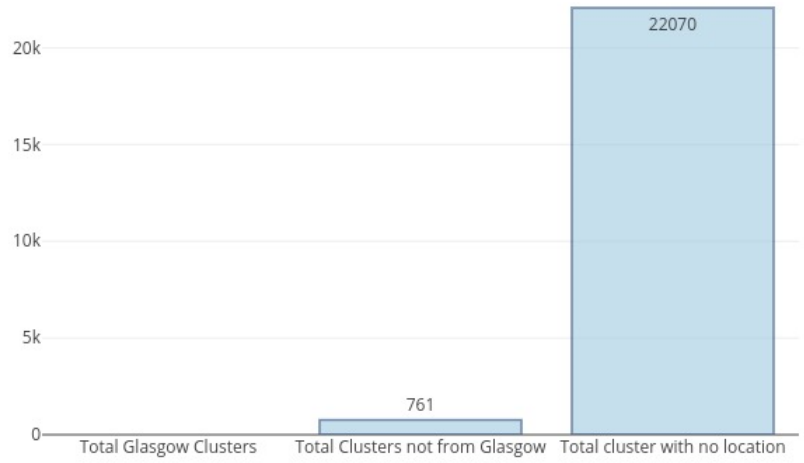
3. If one or more tweets in a cluster are from Glasgow and all other tweets have no location information at all.

The code implementation can be seen in the below screenshot:

```
// 12. Function to assign location to the clusters formed using Minhash LSH
function (totalTweets, allMatchesUnique, newArrayLocation, callback) {
    //variable to store glasgow cluster count, null cluster count
    var glasgowClusterCount = 0, noLocationClusterCount = 0;
    //for each location cluster array, check if occurence of glasgow is more than 50% of the total elements of the array
    async.forEachSeries(newArrayLocation, function(eachLocationCluster, callback) {
        //loop through each location cluster elements to check if it has glasgow
        async.forEachSeries(eachLocationCluster, function (eachLocation, callback) {
            //check if the location in current cluster is Glasgow or not
            if (eachLocation == 'Glasgow') {
                //if glasgow location increment count by one
                glasgowClusterCount += 1;
                //go to next location in the cluster
                callback();
            } else if (eachLocation == 'null') { //if no location is present in the cluster
                noLocationClusterCount += 1;
                //go to next location in cluster
                callback();
            } else {
                //if location is not glasgow, go to next location
                callback();
            }
        },
        //final function for internal loop
        function (errSingle) {
            if (errSingle) {···
            } else {
                //if more than half of the tweets have glasgow as location, then
                //assign the cluster location as glasgow
                if (glasgowClusterCount >= eachLocationCluster.length/2) {
                    //if so, then increment the total glasgow clusters
                    totalGlasgowClusters = totalGlasgowClusters + 1;
                    //make the count to zero for next cluster iteration
                    glasgowClusterCount = 0;
                    noLocationClusterCount = 0;
                    //go to next iteration
                    callback();
                } else if (noLocationClusterCount == eachLocationCluster.length) {
                    //if this condition matches, then cluster has no location information at all
                    //hence increment appropriate count
                    totalNullLocationClusters = totalNullLocationClusters + 1;
                    //make the count to zero for next cluster iteration
                    noLocationClusterCount = 0;
                    glasgowClusterCount = 0;
                    //go to next iteration
                    callback();
                } else if ((glasgowClusterCount != 0) && (glasgowClusterCount + noLocationClusterCount == eachLocationCluster.length)) {
                    //if this condition matches then cluster has location as glasgow but no other location tweet is present
                    totalGlasgowClusters = totalGlasgowClusters + 1
                    //make the count to zero for next cluster iteration
                    glasgowClusterCount = 0;
                    noLocationClusterCount = 0;
                    //go to next iteration
                    callback();
                } else {
                    //if this condition matches then cluster has location information      You, 31 minutes ago • Code enhanced for noise re
                    //but it is not a glasgow cluster
                    totalNotGlasgowClusters = totalNotGlasgowClusters + 1
                    //make the count to zero for next cluster iteration
                    glasgowClusterCount = 0;
                    noLocationClusterCount = 0;
                    //go to next iteration
                    callback();
                }
```

Using this methodology, this software analyses the clusters formed and finds information about the number of Glasgow clusters, a number of clusters which are not from Glasgow, but some other location and all those clusters which have no location information at all.

The below graph is obtained on plotting the result of the method used by this software to assign each cluster a location:



Based on the graph generated, it can be assumed that no clusters belong to the location, Glasgow. In fact, most of the clusters have no location assigned to them at all, which is also seconds the previous observations that the maximum tweets have no geolocation data present in them.

Also, from the above graph, a number of clusters are present which are having a geo-located tweet in them, but they can be from any other part of the world, as tweets obtained using the different APIs may result in the collection of data from all over the world.

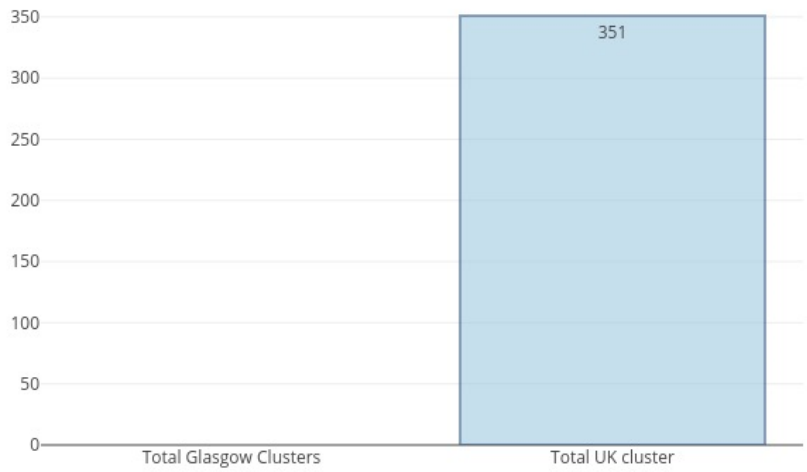
Finally, it can also be concluded that people may be talking about the same topics, but they may not be from the same location at that point of time as incoming tweets are distributed from all different parts of the world.

EVALUATION OF METHOD

The above method is evaluated on a basis that, since the number of clusters belonging to Glasgow location came out to be zero, so, clusters are tried to be assigned a country – the United Kingdom. A similar approach is used for assignment of a country to each cluster as discussed in the section GEO-LOCATION ASSIGNMENT TO CLUSTERS.

The code snippet is similar to the one present in the previous section and is implemented on the exact same conditions, but this time, for the country as the United Kingdom.

The following graph is obtained for clusters assigned a location United Kingdom:



This method has an advantage that even if the exact coordinate of the tweet is not present, a rough idea about the user location may be estimated to a certain extent and hence assigned a location.

The limitation of this method is that it might not work for locations mentioned in other languages like Spanish, Japanese, etc. It is also quite possible that even if the tweet belongs to the United Kingdom, the actual tweet might be about a different location (like Spain) and yet the tweet's actual location (which the user has chosen not to be shown) is totally in a third location (for example the U.S).

DATA CRAWLING: TUMBLR

INTRODUCTION: DATA COLLECTION MECHANISM

Along with the submission of the Twitter crawler, another software is submitted, which crawls the social media site named Tumblr (https://www.tumblr.com/).

The software uses a similar setup of JavaScript and Node.js (v8.12.0) (NPM version: 6.4.1) as the runtime environment. Tumblr has exposed official module names as "tumble.js" NPM module to access the different APIs of Tumblr (Tumblr, 2018).

The various NPM modules/libraries have been used for the software which are Body Parser, Cors, Express, Mongoose, Minhash, Tumblr.js, Winston and Winston Daily Rotate File (NPM, 2014).

ACTUAL CODE ACCESS

The actual code base for this software can be found out at https://github.com/Kinshuk1993/tumblrCrawler.

DATA RESTRICTIONS

The official Tumblr APIs and modules have a restriction that for every request made, only a maximum of 20 posts of any blog may be exposed to the developer, irrespective of the total number of the posts that particular blogs contain.

The restriction is such that the blogs returned for multiple calls for a blog return the exact same posts (which has been verified by the post id, which is a unique identifier). Multiple things have been tried, for example, putting a timer to the REST calls and trying to pass different parameters to collect different posts, but they return the exact same data for every iterative run of the software.

This restriction of maximum 20 can also be verified by looking at the response coming from the REST API call.

Also, few of the APIs like blogFollowers(), blogQueue(), blogDrafts() always throw an error when data is tried to be obtained using these. The assumption may be made that these APIs have either been deprecated or the names have been changed which Tumblr has not updated in their official JavaScript module.

DATA COLLECTION APPROACH

This software collects the currently authenticated user and then goes through each of the blogs that he/she follows explores the blog's data to crawl information in a restricted way. This restricted crawling is due to restriction from Tumblr end.

No timer approach has been used like the one used in Twitter as Tumblr returns the exact same data for one API call.

For data submission purpose, the authenticating user is only following 13 blogs, of which, one is named as staff, which is present by default for every Tumblr account.

CODE SAMPLE

The code in the below screenshot is used to save the details of each blog data that the current authenticating user is following in the database and then do analytics on the data collected.

```
    You, a day ago • Restructred the code and ready for work
async.waterfall([
    //1. Function to get names of all blogs the user follows
    function (callback) {
        client.userFollowing(function (err, follows) {
            async.forEachSeries(follows.blogs, function (eachBlog, callback) {
                    allBlogNameArray.push(eachBlog.name);
                    callback();
                },
                function (errInAsync) {
                    if (errInAsync) {
                        logger.error('Error in async series of saveEachBlogUserFollows');
                        callback(null, allBlogNameArray);
                    } else {
                        logger.info('Going to the next function to get posts of each blog the user follows');
                        callback(null, allBlogNameArray);
                    }
                });
        });
    },
    //2. Function to get each blog post and save in the database
    function (allBlogNameArray, callback) {
        async.forEachSeries(allBlogNameArray, function(eachBlog, callback) {
            client.blogPosts(eachBlog, function (err, resp) {
                TumblrDB.tumblrSchema(resp).save(function (err, savedBlog) {
                    //handle error case
                    if (err) {
                        //If error in saving to database, log it
                        logger.error('Error occured in saving blog to database: ' + JSON.stringify(err));
                        //continue to the next iteration
                        callback();
                    } else {
                        logger.info('Saved the ' + resp.blog.name + ' blog data');
                        //continue to the next iteration to save the next blog to the database
                        callback();
                    }
                });
            });
        },
        function(errFinal){
            if (errFinal) {
                logger.error('Error in async of finding blog data for each blog');
                callback(null);
            } else {
                logger.info('Going to function 3');
                callback(null);
            }
        });
    },
    //3. Function to perform analytics on database
    function (callback) {
        //function to perform analytics on the saved data
        tumblrController.analytics();
        callback(null);
    }
], function (err, result) {
    logger.info('Main waterfall completed, now working on event callback for other analytics');
})
```

The below code is used to find the total posts (photo, links, and texts) of each blog the current user is following:

```
//2. Function to count total post for each blog
function(totalBlogData, callback) {
    //loop over the blog data
    async.forEachSeries(totalBlogData, function(eachBlog, callback) {
        logger.info('Total posts of blog ' + JSON.parse(JSON.stringify(eachBlog)).blog.name + ' are: ' + JSON.parse(JSON.stringify(eachBlog)).total_posts);
        output.info('Total posts of blog ' + JSON.parse(JSON.stringify(eachBlog)).blog.name + ' are: ' + JSON.parse(JSON.stringify(eachBlog)).total_posts);
        callback();
    },
    function(err){
        //handle error and log it
        if (err) {
            logger.error('Error in counting total posts for each blog: ' + JSON.stringify(err));
            //go to next function
            callback(null, totalBlogData);
        } else {
            //go to next function
            callback(null, totalBlogData);
        }
    });
},
```

The following is the code sample to count the total number of link type posts for each blog. A similar approach has been followed and implemented to count the number of photo and text type posts for each blog the current user follows.

```
//4. Function to count link posts for each blog
function(totalBlogData, allBlogNameArray, callback) {
    async.forEachSeries(allBlogNameArray, function(eachBlog, callback) {
        client.blogPosts(eachBlog, {type: 'link'}, function (err, resp) {
            if (err) {
                logger.error('Error in finding blogs of type link: ' + JSON.stringify(err));
                callback();
            } else {
                logger.info('The number of link type posts for ' + eachBlog + ' is: ' + resp.posts.length);
                output.info('The number of link type posts for ' + eachBlog + ' is: ' + resp.posts.length);
                callback();
            }
        });
    },      You, 17 hours ago • Analytics and data saving done
    function(errFinal){
        if (errFinal) {
            logger.error('Error in async of finding posts of link type for each blog');
            callback(null, totalBlogData, allBlogNameArray);
        } else {
            logger.info('Going to function 5 to find number of text type posts fo each blog');
            callback(null, totalBlogData, allBlogNameArray);
        }
    });
},
```

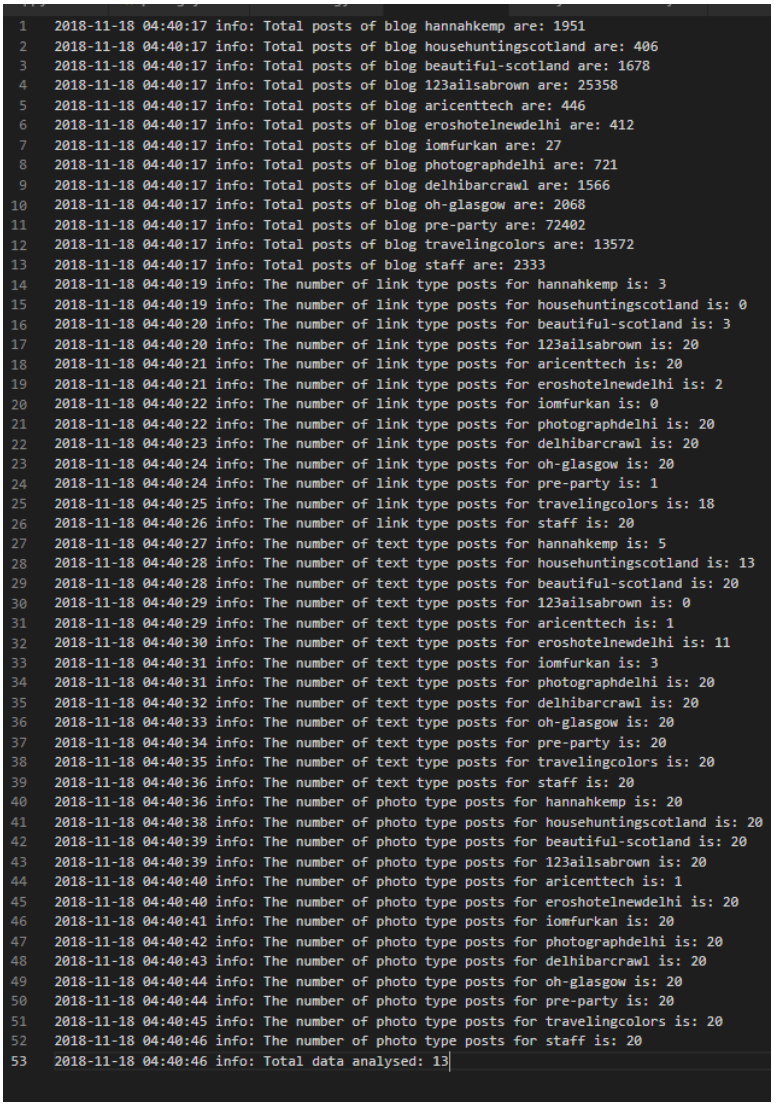
DATA ANALYSIS

The data collected is for the authenticating user. The following analysis is done for the data collected:

1. The total number of blogs the authenticating user follows.
2. For each of the blog that the user follows, count the total number of posts each blog contains.
3. For each of the blog the user follows, count the total number of “Link” type posts each blog contains (maximum being 20).

4. For each of the blog the user follows, count the total number of “Text” type posts each blog contains (maximum being 20).
5. For each of the blog the user follows, count the total number of “Photo” type posts each blog contains (maximum being 20 posts).

The following screenshot of output logs shows the above analysis done on the data collected:



```
2018-11-18 04:40:17 info: Total posts of blog hannahkemp are: 1951
2018-11-18 04:40:17 info: Total posts of blog househuntingscotland are: 406
2018-11-18 04:40:17 info: Total posts of blog beautiful-scotland are: 1678
2018-11-18 04:40:17 info: Total posts of blog 123ailsabrown are: 25358
2018-11-18 04:40:17 info: Total posts of blog aricenttech are: 446
2018-11-18 04:40:17 info: Total posts of blog eroshotelnewdelhi are: 412
2018-11-18 04:40:17 info: Total posts of blog iomfurkan are: 27
2018-11-18 04:40:17 info: Total posts of blog photographdelhi are: 721
2018-11-18 04:40:17 info: Total posts of blog delhibarcrawl are: 1566
2018-11-18 04:40:17 info: Total posts of blog oh-glasgow are: 2068
2018-11-18 04:40:17 info: Total posts of blog pre-party are: 72402
2018-11-18 04:40:17 info: Total posts of blog travelingcolors are: 13572
2018-11-18 04:40:17 info: Total posts of blog staff are: 2333
2018-11-18 04:40:19 info: The number of link type posts for hannahkemp is: 3
2018-11-18 04:40:19 info: The number of link type posts for househuntingscotland is: 0
2018-11-18 04:40:20 info: The number of link type posts for beautiful-scotland is: 3
2018-11-18 04:40:20 info: The number of link type posts for 123ailsabrown is: 20
2018-11-18 04:40:21 info: The number of link type posts for aricenttech is: 20
2018-11-18 04:40:21 info: The number of link type posts for eroshotelnewdelhi is: 2
2018-11-18 04:40:22 info: The number of link type posts for iomfurkan is: 0
2018-11-18 04:40:22 info: The number of link type posts for photographdelhi is: 20
2018-11-18 04:40:23 info: The number of link type posts for delhibarcrawl is: 20
2018-11-18 04:40:24 info: The number of link type posts for oh-glasgow is: 20
2018-11-18 04:40:24 info: The number of link type posts for pre-party is: 1
2018-11-18 04:40:25 info: The number of link type posts for travelingcolors is: 18
2018-11-18 04:40:26 info: The number of link type posts for staff is: 20
2018-11-18 04:40:27 info: The number of text type posts for hannahkemp is: 5
2018-11-18 04:40:28 info: The number of text type posts for househuntingscotland is: 13
2018-11-18 04:40:28 info: The number of text type posts for beautiful-scotland is: 20
2018-11-18 04:40:29 info: The number of text type posts for 123ailsabrown is: 0
2018-11-18 04:40:29 info: The number of text type posts for aricenttech is: 1
2018-11-18 04:40:30 info: The number of text type posts for eroshotelnewdelhi is: 11
2018-11-18 04:40:31 info: The number of text type posts for iomfurkan is: 3
2018-11-18 04:40:31 info: The number of text type posts for photographdelhi is: 20
2018-11-18 04:40:32 info: The number of text type posts for delhibarcrawl is: 20
2018-11-18 04:40:33 info: The number of text type posts for oh-glasgow is: 20
2018-11-18 04:40:34 info: The number of text type posts for pre-party is: 20
2018-11-18 04:40:35 info: The number of text type posts for travelingcolors is: 20
2018-11-18 04:40:36 info: The number of text type posts for staff is: 20
2018-11-18 04:40:36 info: The number of photo type posts for hannahkemp is: 20
2018-11-18 04:40:38 info: The number of photo type posts for househuntingscotland is: 20
2018-11-18 04:40:39 info: The number of photo type posts for beautiful-scotland is: 20
2018-11-18 04:40:39 info: The number of photo type posts for 123ailsabrown is: 20
2018-11-18 04:40:40 info: The number of photo type posts for aricenttech is: 1
2018-11-18 04:40:40 info: The number of photo type posts for eroshotelnewdelhi is: 20
2018-11-18 04:40:41 info: The number of photo type posts for iomfurkan is: 20
2018-11-18 04:40:42 info: The number of photo type posts for photographdelhi is: 20
2018-11-18 04:40:43 info: The number of photo type posts for delhibarcrawl is: 20
2018-11-18 04:40:44 info: The number of photo type posts for oh-glasgow is: 20
2018-11-18 04:40:44 info: The number of photo type posts for pre-party is: 20
2018-11-18 04:40:45 info: The number of photo type posts for travelingcolors is: 20
2018-11-18 04:40:46 info: The number of photo type posts for staff is: 20
2018-11-18 04:40:46 info: Total data analysed: 13
```

From the above data counting and analysis, it may be concluded that the authenticating user has a liking towards the blog posts of type “Photo” and is not much inclined towards “Link” or “Text” based blogs.

BIBLIOGRAPHY

1. Fergie, E. a. (2018). *https://www.npmjs.com/package/stopword*. Retrieved from https://www.npmjs.com/package/stopword.
2. Hudgens, R. (2016). *https://www.siegemedia.com/seo/most-popular-keywords*. Retrieved from Siege Media.
3. Klokantech. (2018). *http://boundingbox.klokantech.com/*. Retrieved from Bounding box Klokantech.
4. Mondovo. (2018). *https://www.mondovo.com/keywords/twitter-keywords*. Retrieved from Mondovo.
5. NPM, I. (2014). *https://www.npmjs.com/*. Retrieved from NPM.
6. ttezel. (2018). *https://www.npmjs.com/package/twit*. Retrieved from https://www.npmjs.com/package/twit.
7. Tumblr. (2018). *https://github.com/tumblr/tumblr.js*. Retrieved from https://github.com/tumblr/tumblr.js.
8. Wikipedia. (n.d.). *https://en.wikipedia.org/wiki/MinHash*. Retrieved from https://en.wikipedia.org/wiki/MinHash.