Web Science H coursework Report

Ze Wang - 2550208W

31 March 2021

Section 1: Introduction

1.1 Overview

The objective of this course work is to develop a Twitter crawler for data collection in English and to conduct social media analytics. File structure as follows:

`requirements.txt` - program dependencies

`Credentails.py` - stores the credentials from twitter Developer account

`StreamingCrawler.py` - sets up and runs streaming API for 5 minutes

`DataGrouping.py` - groups the tweets collected in `StreamingCrawler.py`

`HybridCrawler.py` - runs hybrid architecture crawler for 30 minutes

`Imagex.jpg` - 5 images downloaded in `StreamingCrawler.py`

- The Github repository with the source code : https://github.com/Heath-Web/COMPSCI4077-WebScienceH
- Instructions on how to run the program are specified in the **README.md**

1.2 Implementation

All programs including, Streaming crawler, data grouping and hybrid architecture crawler has been implemented by using Python programming language version `anaconda python 3.7`. The data storage is supplied by MongoDB version `MongoDB 4.4.4 Community`.

Furthermore, several additional python Library are used:

`StandfordCoreNLP` - used to generate text vector

`emoji` - used to remove emoji from tweet text

1.3 Data Collected

Data was stored for both Streaming API and REST API in a database called "TwitterDB" of MongoDB and two collections called "Streaming_1" and "Hybrid_3". "Streaming_1" collection was used to store data collected by `StreamingCrawler.py` through Streaming API only and "Hybrid_3" collection is for data collected by `HybirdCrawler.py` which is a hybrid architecture twitter crawler.

For the streaming API of part 1, I ran it on March 29th from 08:51 to 08:56 UK Time and collected 7839 tweets. These data was stored in a collection "Streaming_1" in "TwitterDB" database. The grouped tweets of the streaming API in part 1 was saved in "GroupedTwitter_2". Moving onto hybrid architecture twitter crawler of part 3, 29173 tweets was collected from 06:59 to 7:29 UK Time on March 31th. The data collected by `HybirdCrawler.py` was stored in "Hybrid_3" collection. Of those 9820 tweets, 19353 was from Streaming API and 2000 was from REST API. The grouped tweets of the streaming crawler in part 3 was saved in "GroupedTwitter_3".

Section 2: Data crawl

2.1 Streaming API

Twitter Streaming API was used here for collection 1% data lasting for 5 minutes.

Filtering

All tweets were filtered with English, the coordinates of UK and Ireland, and the track words which are COVID relevant. The following code shows the tracking words, location and the filtering function.

```
# UK and Ireland
Loc_UK = [-10.392627, 49.681847, 1.055039, 61.122019]
Words_UK = ["COVID-19", "COVID", "Corona", "Virus", "Disease", "Case",
"Quarantine", "Isolation", "Infection", "NHS", "Positive", "Pandemic",
"Restrictions", "Lockdown", "Hospital", "Vaccine", "Infection Rate", "Variants"]

streamer.filter(locations= Loc_UK, track = Words_UK, languages = ['en'],
is_async=True) #locations = Loc_UK, track = Words_UK
```

Text cleaning and fields extraction

The collection of tweet was done in the `on_data(self, data)` function of an `StreamListener` class instance provided by tweepy Library . Before storing the tweets into Mongo database, I extracted relevant twitter fields including tweet_id , created date, text, user, coordinates, place object hashtags, mentions and multimedia etc., and remove the emoji form tweet text by using the emoji Library. The process of counting, cleaning text and extracting fields was completed in the function `processTweets(tweet)`.

```
# remove emoji form tweet text
new_text = re.sub(emoji.get_emoji_regexp(), r"", text)
```

Download multimedia

When processing tweet , the program will record the first five images and videos URL appeared in tweets and download them at the end. By using urllib Library, I acquired the response of the URL and write it in the current folder. The file name is constituted by"Image" or "Video" , number (0 to 4) and the format of multimedia which obtained through the regular expression.

```
# Dowload images or videos through url
def download(url,Num,type): # url: the url of images or videos; Num : mark
number of image or vedio; type : 'image' or 'video'
    try:
        request = urllib.request.Request(url) # format request of the url
        response = urllib.request.urlopen(request) # acquire the response
        result = response.read()
        if type == 'image':
            with open('.\\Image' + str(Num) + str(re.search(r'\.(\w*)$',url,re.I
| re.M).group()),'wb') as fp: # creat file and match the format of image
                fp.write(result) # store the image in  local
            print('Download image:','Figure', str(Num), "   url:" , url)
        if type == 'video':
            with open('.\\Video' + str(Num) + str(re.search(r'\.(\w*)$',url,re.I
| re.M).group()) ,'wb') as fp:# creat file and match the format of video
                fp.write(result) # store the video in  local
            print('Download image:','Figure', str(Num), "   url:" , url)
    except Exception as e:
        print("Some error occurred when download images and videos :", e)
```

Streaming API Crawler `StreamCrawler.py`

This program will automatically disconnect Streaming API 5 minutes after it started. The main thread will sleep for 5 minutes whereas the filtering, text cleaning and twitter fields extracting process is in the sub-thread. It also counted the collected tweet and outputted those number in the terminal which are shown in the following tabular form.

Total	Streaming API	No of re-tweets	No of quotes	No of Images	No of Videos	How many verified?	No of geo-tagged data	How many with locations/place Object
7839	7839	5105	1454	360	0	148	38	385

2.2 Data Grouping

In this stage, 7229 tweets were grouped in 4480 clusters by Single Pass Clustering method according to Cosine Similarity, and the other 610 tweets was considered as noisy tweets based on Quality Score of the tweet user. 7839 tweets in total are all came form the previous part, collected by Streaming API crawler.

Quality Score

Quality score is the average of five different weights which are User Verified Weight, User Profile image Weight, User Followers Weight, Account Age Weight and User Description Weight. If the user account is verified, did not use default profile image, has more followers, was active for longer periods of time and has some News & Journalism relevant terms in user description, then the higher quality score will be. When matching terms in user description , I implemented root forms by using the regular expression. The following codes shows how I calculate description weight. The process of computing quality score was completed in the function `GetQualityScore(tweet)`.

```
    # List of useful terms
    listTerms = ['news', 'report', 'journal', 'write', 'editor','media',
                 'official', 'NHS', 'health', 'care', 'COVID', 'hospital']
    # List of Spam terms
    listSpam = ['ebay', 'review', 'shopping', 'deal','sale', 'sales','link',
                'click', 'marketing','promote','discount', 'products', 'store',
                'diet', 'weight', 'porn', 'followback', 'follow back','lucky',
                'winners', 'prize', 'hiring']
    if tweet['user']['description'] == None:
        descriptionweight = 0.1 # Null description
    else:
        for term in listTerms:  # match word in ListTerms
            match_res = re.search(r'\s?' + term + r'(\w*)\s?', tweet['user']
['description'],re.I | re.M)  # use root forms
            if match_res != None:  # if match
                descriptionweight += 1
                match_counter += 1
        for term in listSpam:  # match word in ListSpam
            match_res = re.search(r'(\s?)' + term + r'(\w*)(\.*)\s?',
tweet['user']['description'],re.I | re.M)  # use root forms
            if match_res != None:  # if match
                descriptionweight += 0.1
                match_counter += 1
        if match_counter == 0: # do not match any terms but user still has non-
null description
```

```
            descriptionweight = 0.4
        else:
            descriptionweight = descriptionweight / match_counter
```

If the quality score of the tweet below 0.5, which means the user is more like a spammer or marketer etc., and this tweet will be consider as noisy tweet with low-quality text and will not be grouped in later stages.

Text Vector

Text Vector generation is realized by using Stanford CoreNLP Library. Firstly, I derive the Part-of-speech Tagging for tweet text and only remain URL, Adjective, Adverb, Verb and Noun words in vector so as to remove noise terms and stop words in tweet text. Otherwise, it will be an error posted by stanfordcoreNLP if text contains symbol %. And Mongodb does not allow any keys contain a '.' and keys start with '$'. So I replace them with other symbol like a space or a '~' in this step. The process of Text vector generation was completed in the function `Generate_vector(tweeet)`

Similarity

The measure used to computing similarity between tweet and cluster is Cosine Similarity Measure. Following the formula below, A represent text vector and B is the cluster vector.

$$\text{similarity} = \cos(\theta) = \frac{\mathbf{A} \cdot \mathbf{B}}{\|\mathbf{A}\| \|\mathbf{B}\|} = \frac{\sum\limits_{i=1}^{n} A_i B_i}{\sqrt{\sum\limits_{i=1}^{n} A_i^2} \sqrt{\sum\limits_{i=1}^{n} B_i^2}},$$

Here is the main part of function `Calculate_SIM(representation, text_vector)` and demonstrate the process of Cosine Similarity Calculation.

```
        sum1 = 0 # numerator, product of cluster vector and text vector
        sum2 = 0 # square of cluster vector length
        sum3 = 0 # square of text vector length
        if len(text_vector) == 0:# deal with zero
            sim = 0
        else:
            for key in representation.keys():
                sum2 += (representation[key] * representation[key])
                if key in text_vector:
                    # assume all terms in text appear once
                    sum1 += (representation[key] * (1 / len(text_vector)))   #
Compute numerator
            sum3 = 1 / len(text_vector)
            sim = sum1 / ((sum2 ** 0.5) * (sum3 ** 0.5))  # Cosine similarity
```

Single pass clustering

In terms of clustering algorithm, i use Single pass clustering. Briefly, it perform a single and sequential pass over tweets. And for each tweet, the algorithm decides whether it should be added in an already defined cluster or a new cluster. The main flow process of Single Pass clustering is shown below where the Quality score threshold and Cosine Similarity Threshold are 0.5. The tweet text vector is empty means all terms in the text is stop words, which will also be consider as a noise tweet. The process of Single pass clustering was completed in the function `Single_Pass_Clustering(tweet)`

```
for each tweet t in the sequence loop
    if GetQualityScore(tweet) < 0.5 then moving onto next tweet
    else if Generate_vector(tweet) is empety then moving onto next tweet
    else find a cluster c maximises Calculate_SIM(c,t vector)
         if Calculate_SIM(c, t vector) > 0.5, add tweet into c
         else create a new cluster whose only doucument is t
```

When clustering, I found that if Cosine Similarity Threshold was set under 0.5, then Two unrelated tweets would also be grouped in one cluster. However, The higher threshold value means there will be more cluster created. When threshold equals to 0.5, the average group size will be 1.6 roughly. So One main weakness of Single Pass clustering algorithm is that the time complexity is approach to `O(n^2)`. With the amount of cluster growing, the time of finding maximal similarity for each tweet will also increase. From several data grouping test previously, when the amount of cluster exceeds 3000, the speed of group a tweet will be significantly slower. One way to improve this problem is that once the similarity over 0.9,which means tweets are the same on a great probability, there is no need to compute similarity with other cluster anymore.

The structure of group representation is shown below. For quick access, this will be kept in the memory as a python dictionary first and be stored into "group_representation" collection in "GroupedTwitter_2" database at the end when grouping completed.

```
{'_id': 1,  # cluster id
 'cluster_id' : 1 , # cluster id
 'count' : 10 , # amount of Tweets
 'representation' : # groupe representation / cluster vector
    {'term' : weight,...}  # terms and their weight
}
```

DataGrouping `DataGrouping.py`

This program counted the clusters and outputted those number in the terminal which are shown in the following tabular form.

Total	Groups formed	Min size	Max size	Avg size	Noisy tweet
7839	4481	1	154	1.61	610

2.3 Hybrid architecture of Twitter Streaming & REST APIs

In this stage, 29173 effective tweets were collected, 9820 from Streaming API and 19353 from REST API.

Prioritize the groups

The main idea of prioritizing the groups is to look how fast the cluster is growing instead of the cluster size. Once a tweet is collected in the Stream Listener, the program will perform a single pass clustering to this tweet. The cluster number which the tweet is add in will be store in a 1000 length queue. This queue represents which cluster the last 1000 tweets were added in. In other words, by counting the cluster number in the queue and sort them, we can know which cluster raises the fastest over a period of time. And the counting and sort process is realized by `Counter` Library of Python.

```
ClusterNum_Queue = [] # a queue with latest 1000 Cluster number where tweet
(form Streaming API) was added
start_index = 0 # The start index (pointer) of the cluster number queue
MAXLENGTH = 1000 # MAXLENGTH  1000 of the queue

aim_coll = DataGrouping.Single_Pass_Clustering(tweet)  # grouping
if aim_coll != -1: # if aim_coll is -1 means this tweet is a noisy tweet
   # add cluster id into the queue
   if len(ClusterNum_Queue) < MAXLENGTH:
        ClusterNum_Queue.append(aim_coll)
   elif len(ClusterNum_Queue) == MAXLENGTH:
        ClusterNum_Queue[start_index] = aim_coll
        # move pointer
        if start_index < (MAXLENGTH - 1):
             start_index += 1
        else:
             start_index = 0

TopGrowingCluster = Counter(ClusterNum_Queue).most_common(50) # Top 50 fastest
growing cluster
```

Query Generation

After prioritizing the groups, i move onto the query generation for each cluster. The priority of things is hashtags, user mentions and terms in cluster vector. Find all hashtags in one cluster, if zero then find all mentions, and same to the terms. The query of REST API Search would be one of them not all or both of them. If one tweet in the cluster is geo-tagged, then the searching of tweets will base on 10km radius of the tweet coordinate. The process of Query generation was completed in the function `GetQueries(ClusterNum)`

```
 for tweet in  collection.find(): # find all tweet in the collection
        if tweet['coordinates'] != None: # if tweet is geo-tagged, generate geo
term
            geoTerm = str(tweet['coordinates'][0]) + str(tweet['coordinates']
[1]) + '10km'
        for hashtag in tweet['hashtags']: # find all hashtags of each tweet
            hashtag = '#' + hashtag
            if hashtag not in hashtags: # avoid repetition
                hashtags.append(hashtag)
        for mention in tweet['mentions']:# find all user mentions of each tweet
            mention = '@' + mention
            if mention not in mentions: # avoid repetition
                mentions.append(mention)

    # priority hashtags > user mentions > text terms
    # query ca onnly be one of them
    if hashtags != []:
        query = ' OR '.join(hashtags)
    elif mentions != []:
        query = ' OR '.join(mentions)
    else:
        query = ' '.join(DataGrouping.group_rep_list[ClusterNum]
['representation'])
    return query, geoTerm
```

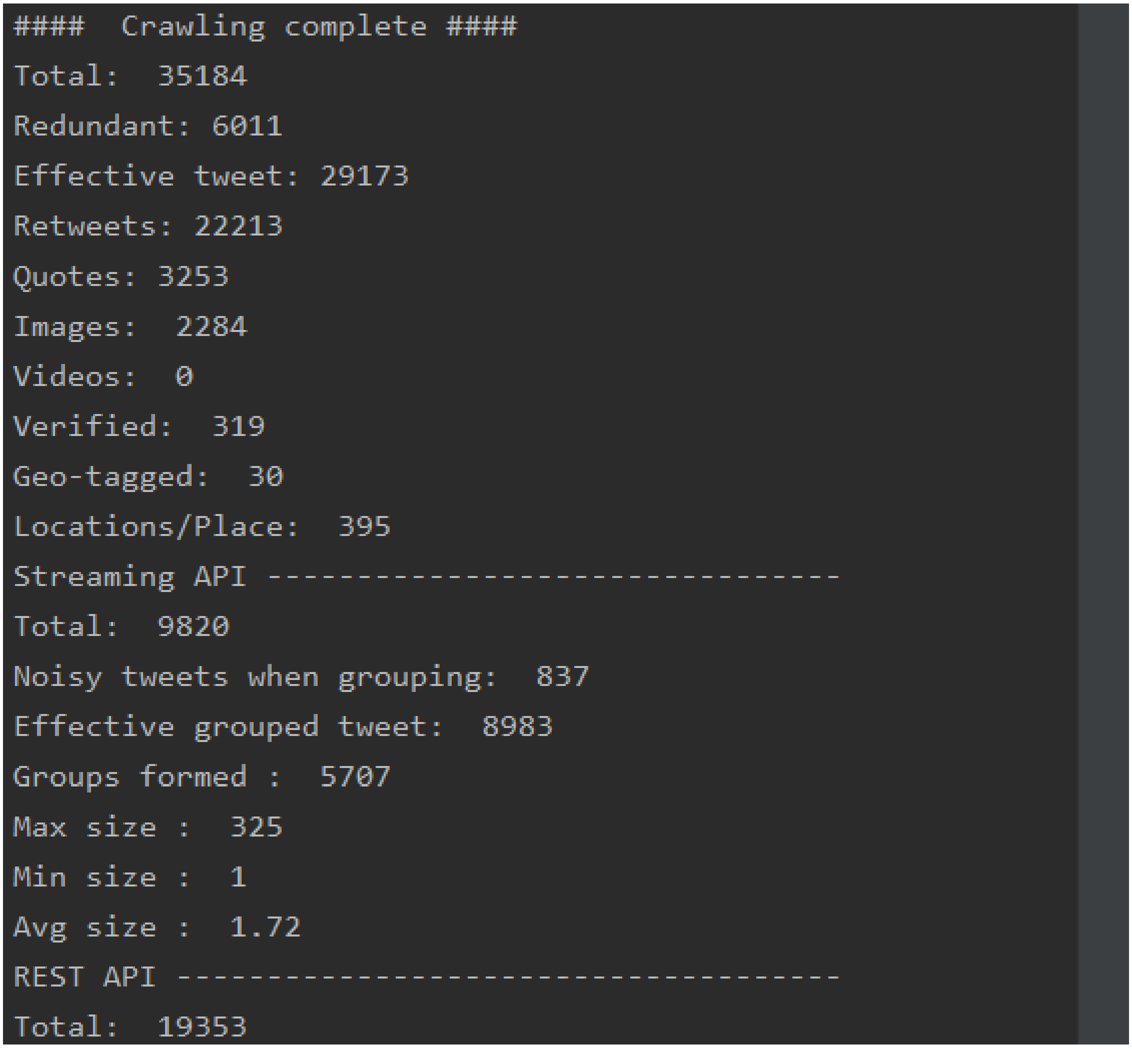
Hybrid Architecture Crawler `HybridCrawler.py`

This program will automatically disconnect Streaming API and terminate REST crawler 30 minutes after it started by set a timer in sub thread (`Class Timer(threading.Thread)`).

It also counted the collected tweet for both Streaming and REST APIs ,and outputted those number in the terminal which are shown in the following tabular form.

Total	Streaming API	REST API data	Redundant	No of Quotes	No of Re-tweets	No of geo-tagged data	No of Images	No of videos
35184	9820	19353	6011	3253	22213	30	2284	0

Output:



```
####  Crawling complete ####
Total:  35184
Redundant: 6011
Effective tweet: 29173
Retweets: 22213
Quotes: 3253
Images:  2284
Videos:  0
Verified:  319
Geo-tagged:  30
Locations/Place:  395
Streaming API --------------------------------
Total:  9820
Noisy tweets when grouping:  837
Effective grouped tweet:  8983
Groups formed :  5707
Max size :  325
Min size :  1
Avg size :  1.72
REST API -------------------------------------
Total:  19353
```

Effectiveness and Scheduler/ranker

At first the strategy of scheduler is that executing 180 queries and then sleep 15 minutes. However, when crawling, the top of fastest growing cluster did not change too much which meas the REST crawler search the same things from the same clusters. And the tweets returned were also almost the same (about 30000 tweets in total and 20000 redundant tweets).

One of the solutions is that allocate the 15 minutes rest time into each query which means the crawler will sleep (15*60)/180 = 5 seconds after each query. This will leave enough time for Streaming API crawler collecting data and the top of fastest growing cluster will be more changeable.

Meanwhile, I extended the period of REST crawler. In previous, getting the top 5 fastest growing cluster and querying 5 times for each cluster were considered as one period. And now in one period 50 queries to the top 50 fastest growing cluster are performed. The aim of this strategy is to cover more clusters.

```
if (counter % 50) == 0: # Every fifty queries
    TopGrowingCluster = Counter(ClusterNum_Queue).most_common(50) # Top 50 fastest growing cluster
ClusterNum = list(list(zip(*TopGrowingCluster))[0])[int(counter % 50)]
query, geoterm = GetQueries(ClusterNum) # get query form cluster (hashtags, mentions or terms)

# query
if geoterm != '':
    results = api.search(q=query, geocode=geoterm, count=80, lang="en", tweet_mode='extended')
else:
    results = api.search(q=query, count=100, lang="en", tweet_mode='extended')
# process results and insert into database
```