Mining Twitter

Brian Schirmacher

June/July 2015

Contents

1 Introduction

This project entails collecting large amounts of data from the Twitter platform using the Twitter API. The API allows for the collection of 1% of current public Tweets; that is we will stream around 1% of all current Tweets from Twitter. At full volume, this amounts to approximately 5 million Tweets per day. Data will be collected and processed in order to determine patterns in data and perhaps detect recurrences of data from the Twitter Sample. Naturally we will wish to exclude certain feeds and types of Tweets, more of which will be discussed later. Finding texts which are exact matches is easy in computing, as one text either fully matches another or it does not. In the case that one text does not match another, we are only aware that it does not match and we have no indication of how similar the texts may be. The ultimate goal of this project is to be able to find text(s) which are similar quickly so that we can see Tweet texts which are very similar to others but do not match exactly. It is imperative that we are able to do this quickly and efficiently so that we do not require months of processing time in order to get results. The rest of this report details how this was achieved.

2 The Twitter API

We will be streaming Tweets using Twitter's API. There are 2 forms of the API, namely the REST API and the Streaming API. The REST API provides us with an interface to read and write Twitter data. We can, for example, author new Tweets, read author profiles and access follower data. The Streaming API, which we will be using, provides an interface to access Twitter's stream of data. We will be provided with a stream of public Tweets. This is a random sample of the global stream. There is also a way to access 100% of global Tweets called the Firehose, but this is expensive to gain access to. Gaining access to Twitter's API requires registering an application on Twitter's developers page. This will provide you with a consumer token and you can then generate an access token. Both of these must be provided to authenticate your app using the OAuth Protocol in order to access the API.

3 Implementation

3.1 Streaming from Twitter

In order to stream data from Twitter, we used a package called Tweepy[4]. This is an open source package designed to make interacting with Twitter's Streaming API seamless, although some personal modifications were added. This is implemented in Python. We use a client to make calls to the Tweepy package and process the stream provided by Twitter. Firstly we authenticate our application using the OAuth Protocol. We then make the call to use the GET statuses/sample.html through the Tweepy package. Twitter pushes the data to us in JSON (JavaScript Object Notation). The Tweet volume of the sample stream was usually seen to be between 100k and 320k Tweets per hour. This fluctuates naturally depending on the global usage of Twitter: sometimes usage is high and the Streaming API will provide us with a large volume of Tweets, and sometimes usage is lower in which case we will be provided with a smaller volume of Tweets. We write this data (in compressed form) to "filename".json.gz files for later processing. In streaming the data from Twitter, we must be sure that we do not fall behind on processing the data stream. If we fall too far behind, Twitter will disconnect us. In this implementation, this was usually seen as a "ProtocolError (IncompleteRead)". When receiving these errors, we must try to reconnect the stream in order to continue streaming data. In order to do so, we handle the thrown exception and implement a backoff strategy so that Twitter does not block us for making too many reconnect requests per unit of time. At first disconnect, we wait one minute, then 2 minutes at second disconnect, 4 minutes at third disconnect and so forth. If the reconnect waiting time exceeds 20 mins, we exit the script and we must manually restart the script. If this happens more than once, it is very likely that the connection is too slow. For safety, a minimum connection speed of 5Mbps should be used, although 10Mbps is recommended. It is also important that Unicode is handled properly, as much of the text in the "text" field (the actual text of the Tweet) in the JSON file is encoded in some form of Unicode, so UTF-8 encoding should be appropriate.

3.1.1 Local Implementation

Streaming was first implemented locally. Most issues were ironed out testing the script locally, and it was quickly noticed that the local connection was too slow to handle the stream from Twitter when it was being pushed to us at a high volume.

3.1.2 Cloud Implementation

It was decided to run the Python script to collect the data on a cloud server. A Digital Ocean Server was rented, and all tweets are stored by writing them to files on this server for the time being, although the memory on the server will need to be freed periodically to make room for new incoming data. We chose to use these servers as they are not at all short on bandwidth and should be able to handle large volumes of data. The files on the server will be downloaded on an hourly basis using a script and synced with the files on the local system. This bash script was written and added to the crontab (scheduling table) to be run every hour on the local machine.

3.2 Processing the Data

The processing of the data is done in Java. A useful Java UI was set up using the Swing library. A full list of available files is presented to the user, with the user

being able to choose files which contains a full days' worth of collected Tweets, or files which contains hourly collections with the time-stamp of the file (time-stamp is recorded as the time when the file was created, so the file contains data collected between the time-stamp marked on the file and the time-stamp plus 1 hour). The user can select one of these files at a time and load the file data (compression/decompression and handling of the JavaScript Object Notation of the file is handled for the user). File loading takes some time (usually between 10 and 25 seconds for hourly files and between 5 and 7 minutes for daily files) depending on the volume of data contained in the file. As the file is being loaded, the SimHash value of each Tweet's Text is calculated, which is why loading the file is slow. More information on SimHashing will be given later in this text. Once the file is successfully loaded, some attributes of the data in the file are presented to the user, namely the number of Tweet objects, the number of user deleted Tweets (recalled Tweets), the total number of Tweets (including Re-Tweets), the number of Re-Tweets, the number of different (unique, case sensitive) HashTags and the number of different users that authored Tweets in the file (deletes are ignored here). The user may then load further information regarding the contents of the file, such as the 30 most popular HashTags in the sample to get a rough idea of what is trending on Twitter.
Each JSON Object is read into memory, one at a time. If the Object is a Tweet (as opposed to a delete object), we extract the text of the Tweet, calculate it's SimHash value (depending on what the user has specified we calculate either a 32 bit or 64 bit SimHash, 128 bit support will be added later) and store it into a Hashtable. The algorithm for calculating a SimHash value will be discussed in the next section. Thus we only store unique Tweets into the Hashtable. It is also extremely improbable that 2 different Strings will hash to the same SimHash value, so we don't lose data this way. This Hashtable is held as an Instance variable of a class, so that we can access it outside of the class and make use of it later. Note that some of the Hashtable implementations used in this project are part of Sedgewick & Wayne's Algorithms Library for their Algorithms book[5].

3.3 The SimHash Algorithm

The SimHash Algorithm is defined as follows from the following steps[2]:
1) Choose a global fingerprint size (say 32, 64 or 128 bits). My Java implementation allows the user to specify whether a 32 bit or 64 bit hash size should be used. We will refer to this fingerprint as an array v[ ] and we will assume each entry is initialised to zero.
2) Shingle each String you wish to compute the SimHash of into bigrams (transform each String into a set of 2 character Strings).
3) For each bigram of a given String, compute a hash value of the bigram using an ordinary hash function which produces the same size (in bits) as the fingerprint size you chose above. My implementation used MurmurHash2 [1] in it's 32 and 64 bit forms.
4) For each bigram hash, for each bit i in this hash, if the bit is a 0, we increase

the value of v[i] by 1, or if the bit is 1 we subtract 1 from v[i]. We can then interpret this array v[ ] as a binary number, which we will call the SimHash of the document/String. The SimHash Algorithm is powerful because in order to compute the similarity of 2 Strings, all we have to do is use an XOR operation and count the number of 1 bits. This number, when divided by the size oif the hash and subtracted from 100, will give us a fair representation of how similar the Strings are.
This implementation made use of a library consisting of the SimHash functions by [6].

4 Speed of Matching

We now have a way to quickly determine similarity of 2 Strings. The rest of this report details the application to big data and quickly finding matches in big data sets. In an inefficient implementation, the problem of matching easily becomes $O(n^2)$ due to the fact that you will most likely run into n(n+1)/2. However, we can exploit the fact that when the SimHash values are sorted, values with a low bitwise hamming distance end up being close together. This allows us to find values in clusters by running through an array of the SimHash values. This part is $O(n)$. We can also exploit bit rotations, so when we rotate a few bits and sort again, we will hopefully find more matches. So the total runtime of finding matches becomes $O(n \log n)$ (from the sorting) + $O(n)$. This leaves us with a final running time of $O(n \log n)$, which will scale well onto large sets of data. All that remains is the details of the implementation on how to achieve this, although this should now be easy. We should also consider the fact that bitwise rotations may reveal other matching Strings to us. If we rotate bitwise to the left and sort again we may find other Strings whose hamming distance is low next to each other[3].

5 Results and Discussion

While the reading and parsing of the files is rather slow, it was quite surprising how fast the final implementation of the algorithm performed with slight modifications and tweaking of the data structures and the way in which data was processed. An important point to note is that results achieved with a 64 bit hash were much more accurate than those obtained with a 32 bit hash function. When the 64 bit hash function was used, false matches (noise) were minimal. The data was matched extremely closely using a 95% lower limit, with only a few characters differing between comparisons. It was also surprising to note how accurate this implementation of the SimHash Algorithm was. Results were very promising and further tweaking of the algorithm and data structures could prove to make the algorithm even more efficient and accurate than the current implementation.

6 Conclusion

Results indicate that the SimHash Algorithm is indeed a very powerful method for finding near duplicates of Strings in large data sets. Furthermore, this implementation is fast with a good time complexity and isn't overly intensive on Java's memory heap. The most memory intensive part of the implementation was parsing the file and storing the Strings and their SimHash values in memory. When using the 64 bit hash, file loading took approximately 447 seconds (5 million JSON objects in 7.5 minutes), and the matching of around 3.4 million Tweet texts was done in approximately 5-6 seconds on a 95% matching lower limit, with a further 5-10 seconds for writing all the matches to a file (170k similar match sets found).

References

[1] Austin Appleby. Murmurhash - [https://code.google.com/p/smhasher/wiki/murmurhash].

[2] Titouan Galopin. Galopin, t. - [http://www.titouangalopin.com/blog/2014-05-29-simhash].

[3] Mat Kelcey. Mat kelcey blog - [http://matpalm.com/resemblance/].

[4] A. Roesslein, J. Hill. Tweepy - twitter for python - [https://github.com/tweepy/tweepy].

[5] K. Sedgewick, R. Wayne. *Algorithms, 4th Edition.* Addsion-Wesley, 2011.

[6] Cheng Zhang. Zhang, c. - [https://github.com/sing1ee/simhash-java].