



DECEMBER 15, 2015

ANALYTICAL DATA MINING

FINDING LIKE –MINDED USERS BASED ON TWEETS

Under Professor:

Reza Zafarani

ROHITH ENGU

SUID: 678923180




Table of Contents

Introduction.....	2
Motivation.....	2
Related work	2
Algorithm.....	3
Dataset.....	3
Pre-Processing of data	3
Min Hash Signature Matrix Generation	5
Locality Sensitive Hashing (LSH)	5
Implementation.....	6
Results.....	6
Evaluation.....	7
Future Enhancement.....	8
Conclusion	8

Introduction

With the advent of social networking websites like twitter, people have a means to share their thoughts.

These thoughts are shared by tweeting. Twitter provides an API through which you can fetch currently tweeted tweets.

The structure of the fetched tweet consists of a Username, date, tweet-text and so on. In this project I would make use of the tweet text in order to group set of users who are like-minded.

Motivation

It is necessary for lot of people to know like-minded individuals. This type of data can be used for target advertising, estimating majority of votes during an election and so on. Twitter is an open platform which lot of users around the world use to share their thoughts in general.

Definition of Like-Minded people:

Here I would like to define what like-minded people mean in this project. If a user is tweeting a lot about some particular topic it definitely says he is interested in that topic.

For Example: if a user “u1” is tweeting a lot about “dogs” it is assumed that he definitely has some correlation with dogs, we can say that other users (u2, u3) who tweet about “dogs” are also having an interest in “dogs”. As the tweet is always about what user thinks or is opinionated about we can definitely say the users u1, u2, u3 are like minded. Main motivation for this project was to have the ability of finding these group of users who have similar thoughts.

Related work

There is a lot of existing work in grouping users in twitter, some of them are based on sentiment analysis of their tweets. The below paper focusses on fetching sentiment of user’s tweet for example: A particular company can find sentiment of their brand among users. ^[1]

[1] Twitter Sentiment Classification using Distant Supervision – Alec Go, Richa Bayani, Lei Huang

There is also a paper which finds similar communities based on the (celebrity) people a twitter user follows and also finding the topics in which they are interested, which could be useful for twitter to suggest friends with similar interests. ^[2]

[2] The Identification of Like-minded Communities on Online Social Networks - Kwan Hui Lim.

Lot of the previous work is based on text classification using various methods like SVM, Naïve Bayes Classifier especially when it comes to reviews of movies (IMDB) or restaurants (yelp).

My implementation of the like-minded people is very different from the previous work carried out in this area. My implementation is based on a simple similarity of words based on Locality Sensitive Hashing for Min-Hashed Signatures. I felt it to be very simple and useful to find similarity between two user's tweets and by knowing the similarity we can have a basis of saying that two users are like minded.

Algorithm

The implementation of this project is quiet intensive and lengthy. I would explain each of the steps followed to achieve the results.

Dataset

The dataset used for this project is taken from <http://help.sentiment140.com/for-students>

The format of the data is as shown below:

Data file format has 6 fields:

0 - the polarity of the tweet (0 = negative, 2 = neutral, 4 = positive)

1 - the id of the tweet (2087)

2 - the date of the tweet (Sat May 16 23:58:44 UTC 2009)

3 - the query (lyx). If there is no query, then this value is NO_QUERY.

4 - the user that tweeted (robotickilldozr)

5 - the text of the tweet (Lyx is cool).

As this data was already crawled and for the ease of use I made use of this data. I myself also crawled twitter using Twitter API's provided by Tweepy (python Library) but for now I used the existing data (I will share the python script while submitting the project).

From the above data format the only things we are interested in this project are the Username and his tweet text.

Following steps were done once I had this data.

Pre-Processing of data

This step is quiet important for further processing, it includes cleaning the data and organizing it such that users can be compared based on some words which are of meaning.

Step 1:

The Data set which I had was in CSV format with all the above mentioned data fields, I parsed each and every tweet and put them in a HashTable (HashMap) which stores username as the Key and tweet text without the stop words (I used Lucene Analysis library for removing stop words) as the value, I kept on appending the text if there were multiple tweets for the same user. I also eliminated non alphanumeric characters here.

Once I had this Hash Map I loop over the whole map to store each and every user's tweet text in a separate file, Ex: If there were 1000 users I made 1000 ".txt" files, I am implementing a document similarity kind of approach for comparing user's tweet with others so I felt it was convenient to do so.

As you can see the user is mapped to a number while saving his tweets' text, I saved this in a properties file for future reference.

Below diagram clarifies the first step:

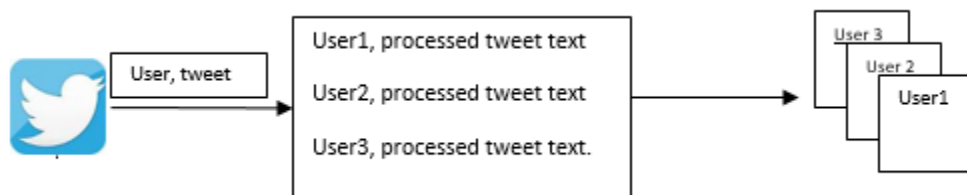


Figure 1: Preprocessing step.

Step 2:

The next thing is to match each document, in order to do that we need shingles to match each and every document, in order to fetch shingles, I ran a Map Reduce job which gave me top words from all of the content. The Map reduce job is a simple word count job. I have implemented it in two stages, in stage 1 word count for all the words are fetched. I needed a second Mapper and Reducer so that I can sort all the counts in descending order. The output of the Map reduce Job is a file which contains top occurring words from the data set in descending order.

From this file I pick some number of words for whom shingles are generated. Currently I am using a 3 character shingle for comparing.

All the shingles are stored in a Set so that there are no duplicates.

This step is shown in diagram as below:

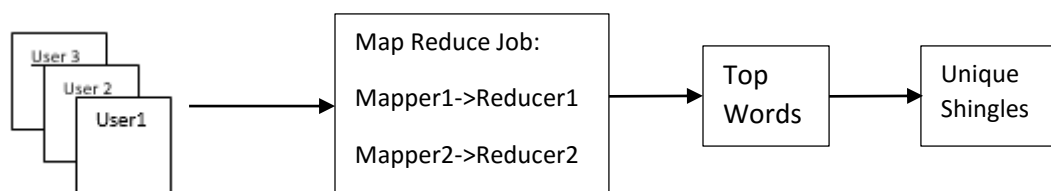


Figure 2: Shingle Generation

Min Hash Signature Matrix Generation

Step 1:

Initial Matrix Generation:

Once we generated the shingles, we can build the initial Matrix which maps shingles with user.

This Matrix consists of shingles as the row index and User as the column, If a particular shingle is present in the User Tweet Text Document, it is marked as 1 else 0. This is a very sparse Matrix with lot of zeros and few ones.

Due to memory constraints on my machine I could only use top 500 words with

Once I generated this Matrix I stored it in a CSV file (saving memory) for further usage.

Step 2:

Signature Matrix Generation:

Once we have the initial matrix with lot of zeros we now apply hash functions to generate signature Matrix. We perform LSH on this signature matrix to get the results, In this project I made use of 1000 hash functions to generate the matrix. Each value in this matrix represents the index of first occurrence of 1 in the initial matrix.

In brief:

Initial Matrix is representation of Shingles to Users,

Signature Matrix is representation of Hash Functions to Users.

Locality Sensitive Hashing (LSH)

The algorithm followed in generating the candidate pairs is from the book “Mining of Massive DataSets” –Lescovec, Rajaraman, Ullman.

Once we generate the Signature Matrix we divide it in bands, with number of rows in each band as number of hash functions / number of bands. I made use of 200 bands for 2000 hash function which makes it 10 rows in each band.

As per the algorithm the probability of two documents (users) agree on all the rows of at-least one band to become a candidate pair is $1-(1-s^r)^b$

Where s is the jaccard similarity between two sets (hash values), “r” is the number of rows per band and “b” is the number of bands used.

In order for a document pair to be a candidate pair to avoid lot of false positives and false negatives as per algorithm we set the similarity threshold to $(1/b)^{(1/r)}$

Which is about 0.588, I calculate this similarity by taking in Hash values for each band and comparing with every other user.

If the similarity between the Hash value set of a user in a band is more than the threshold value then I add the pair to a Hash Map.

This hash map is the final set of candidate pairs who have matched in at-least one of the bands. I saved this hash Map into a text file.

Implementation

While learning the LSH Algorithm it looks very easy to do by hand, but the actual implementation considering the number of users, hash functions, bands to keep in mind is a bit complicated when implemented through code.

All of my implementation is in java and self-implemented.

I have a Python script which crawls twitter (Not Used for this project dataset though).

A java project which does the Map Reduce Job.

A Java Project which implements the LSH algorithm.

All of the source code will be shared with execution steps in the ReadMe file attached along.

Results

I Preprocessed around 230 MB of data (tweets for different users), which had around 600000 unique users, each user on a manual go through had about 3-4 tweets in the dataset.

From the number of users as it was highly impossible for me to make use of all the users, I generated documents for 80000 users.

Due to the computing resource constraints (Max 1GB for Java) I could only generate results for about 500 users with 2000 hash functions.

I made use of the Similarity threshold which the book discussed (0.588) to filter the candidate pairs.

From all the bands number of unique candidate pairs came out to be approximately 25000 users.

I tried and tested with various parameters of Bands, hash functions, number of users, shingle length and number of top words to consider for shingling.

The number of unique count of candidate pairs for different parameters is as shown below:

Test No.	Number of Bands	Number of Hash Fns	Similarity Threshold	Number of Users	Shingle length	Number of top words to shingle	Unique Candidate Pair Count
1	200	2000	0.588	500	4	1000	~25000
2	50	1000	0.822	500	3	1000	~66000
3	100	1000	0.630	500	4	1000	~25000
4	100	1000	0.630	500	5	1000	~8000
5	200	1000	0.346	500	5	1000	~8000
6	200	1600	0.515	500	5	1000	~8000
7	200	1600	0.515	800	5	1000	~14000
8	250	2000	0.501	900	5	1200	~19000

From the Results shown in above table there are few things to be considered.

- Shingle Length: it plays major role in filtering out number of candidate pairs, from the test numbers 3, 4 I increased the shingle length from 4 to 5 character. I compare a shingle with the document text using String Contains method in java. Which compares if sequence of characters are present in the string for each shingle. There is a massive difference between tests 3, 4 when shingle size is changed. The book says length of shingle should be ideally 5.
- Number of Bands: from the Algorithm in book the candidate pairs can definitely be filtered by increasing or decreasing the number of bands.
- Number of Hash functions: In my findings increasing the number of hash functions has not much effect in the overall candidate pair count as shown in tests 5, 6.

Evaluation

Remember that the candidate pairs which are fetched does not mean that the pair is totally similar, it means that two Users tweet text document are similar in some part of their document but not all of it.

To evaluate the results, I manually looked up through some tweet documents and found very small similarity.

For Example : The candidate pair for one of the test case had [0,9]

0 as in from UserMapping.Properties file is for user "aryn505" and 9 as in "the1aries" whose tweet text documents are as shown below:

- 0- cerak ouch sucks soco never **good** idea
- 9- makes me wanna go back bed **good** morning tweeters busy monday me after extremely busy weekend

From the Naked eye we can definitely see a match for "good" which is why they were picked as candidate pair.

Future Enhancement

We can fetch candidate pairs from each of the band and save the count for each occurrence, from the count we can predict the similarity of the documents.

Conclusion

Lot of previous work had been done in grouping users based on their friends and interests, my approach was to find the people who think and write similarly. Even though there are lot of libraries for LSH implementation every implementation had different concept, I went with the algorithm which book talked about and implemented it myself.

I learnt a lot from this course, especially from the assignments.