Twitter Analysis

Brandon Cooper
William Farmer
Kevin Gomez

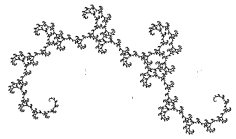
December 6, 2013

Contents

1 Introduction 1
2 Materials and Methods 1
2.1 Data Acquisition and Formatting 2
2.2 Data Cleaning 3
2.3 Data Entry 4
2.3.1 Create Matrix 6
2.3.2 Replace 7
2.3.3 Followers 7
2.3.4 Normalize 8
2.4 Analysis 8
3 Analysis 9
4 Implications 9
5 Discussion 9
1 Attachments 10

List of Figures

List of Equations



Abstract

The purpose of this project was to analyze twitter data and determine a ranking based on the influence of the users. The first step was to determine an initial algorithm for what a twitter userâĂŹs influence should be based on. In this case, our group used retweets, followers and favorites to rank the profiles influence. Once the initial algorithm was created, the necessary data from twitter was obtained and a stochastic matrix consisting of probability vectors was created using the algorithm. The power method was then applied to the probability matrix and the eigenvector of the matrix was obtained. In order to obtain the actual rankings for the different users, the Perron-Frobenius theorem was applied to make sure the matrix was irreducible. Once that was determined, our final ranking was obtained.

1 Introduction

We are seeking a method to rank a random sample of Twitter users in Boulder based on how influential they are on other users. In order to rank a user based on their influence of any other random user, we first defined a value of influence based on a user's tweet, or posting on Twitter. This value is a weighted sum of three important interactions in Twitter: followers, favorites, and retweets.

$$Influence = x \cdot (followers) + y \cdot (favorites) + z \cdot (retweets) \tag{1}$$

To understand these three interactions, take for example user A and user B. If user A follows user B on Twitter, anything that user A tweets will appear on user B's news feed. This is the most common form of interaction, and therefore holds the smallest magnitude in our definition of influence. If user A favorites a tweet from user B, user A expresses his/her interest in user B's tweet because of its humor, importance, relevance, etc. Users can see one another's favorited tweets, so if user B is commonly favorited there is a good chance others will see his/her tweets. Finally, the least common interaction, a retweet is when user B allows a tweet from user A to show on his/her page. In doing so, all of the followers of user B see the tweet that user A posted. This broadens the scope of their influence, as many more followers are able to see the tweet without having to search for it. This can quickly allow someone with a small amount of followers to reach a large amount of users.

After defining the influence of a Twitter user, we streamed a random sample of tweets in the Boulder area into our database. From there, we harvested only the retweets and discarded all other tweets. We chose to sample solely retweets to ensure that our population is at least moderately influential; otherwise, we could end up with several thousand users that are never retweeted and a mere handful of influential Twitter users. At this point we had all relevant factors of influence for the n posters.

We proceeded to create a sparse matrix of $n \times n$ dimensions with each value corresponding to the influence of the user in the each row to the user in each column. For example, if user A had no interaction with user B, the value at AB is zero. If however, user C retweeted user F, the value at FC=z. We then normalized the rows of this matrix to create probability vectors in each row. At this point, we needed to find the dominant eigenvalue and its corresponding eigenvector.

We chose the iterative system of the power method to determine the eigenvector and corresponding eigenvalue for our probability matrix A.

2 Materials and Methods

Because there was a large amount of data that needed to be analyzed, there were numerous tools that were necessary for an effective study. The tools used include Twitter's development API as well as coding methods for analysis of the data. The first step to figuring out the ranking was to obtain the data from twitter that needed to be analyzed. Twitter's development API allowed us to take the necessary data in order to rank the users. The data Twitter's API provided included the tweets of different users. For the ranking method used, the only tweets that were needed were the retweets. We only wanted the retweets because retweets were rare and they would allow for a more accurate ranking. The idea behind it was that it took out any data that could be considered an outlier.

Once the data from twitter was acquired through the API, the tweets needed to be processed in a way that the ranking method could then be applied. Doing the work by hand would have been nearly impossible due to the amount of data and the complexity of the ranking methods. In order to analyze the data effectively, coding was necessary. Python, R and SQLite were the three coding languages used for the analysis. Python was used mainly because it is an all-purpose language that has simple syntax and library support. It also works well with Twitter's API because it has a built in library for HTTP requests. R was used because it can handle large data sets and has built in tools for visualization. It can also handle larger matrices which would be necessary for the data being ranked. SQLite was used because it was easy to use and is great for working with database framework. With those three languages, the ranking method could be applied to the twitter data. Python was used solely to obtain the data from Twitter's API. It took the data and stored it for later use. Once the data was stored, the ranking algorithm we created was applied through the coding language R. The application of out ranking algorithm gave us a matrix, which was then normalized to give us our

stochastic matrix. The Perron-Frobenius theorem verified it was irreducible, which allowed us to proceed with finding the ranks. Then through use of R again, the power method was applied and the eigenvector of the matrix was created.

More specifically there were three main steps involved in analysis.

1. Data Acquisition and Formatting (Python)
2. Data Cleaning (Python)
3. Data Entry (R)
4. Analysis (R)

2.1 Data Acquisition and Formatting

The first step of this process was to import the data from Twitter's streaming API.

./code/twitterImport.py

```
21 # Limit to English around Boulder, CO
22 data       = {'language':'en', 'geocode':'40,105,50mi'}
23 # Stream URL
24 response   = requests.get('https://stream.twitter.com/1.1/statuses/sample.json',
25                           params=data, auth=auth, stream=True)
```

Twitter provides a handy url for streaming a random subset of all Twitter traffic. In order to avoid overfitting, or selection bias we used this random subset for all data analysis. We limited our search to English posts from Boulder, CO in order to hopefully acquire a densely connected network of related posts.

The next step is to either create the SQLITE database or open a previously existing one. The tables in the database are created with the following python code, and look like Table 1.

./code/twitterImport.py

```
29 cursor.execute("""CREATE TABLE users(
30                     id TEXT PRIMARY KEY NOT NULL,
31                     name TEXT NOT NULL,
32                     followers TEXT,
33                     favourites_count INT,
34                     followers_count INT,
35                     friends_count INT)""")
36 cursor.execute("""CREATE TABLE posts(
37                     id TEXT PRIMARY KEY NOT NULL,
38                     time TEXT,
39                     entities TEXT,
40                     user TEXT)""")
41 cursor.execute("""CREATE TABLE retweets(
42                     id TEXT,
43                     end TEXT,
44                     count INT)""")
```

users	**posts**	**retweets**
id (text)	id (text)	id (text)
name (text)	time (text)	end (text)
followers (text)	entities (text)	count (int)
favourites_count (int)	user (text)	
followers_count (int)		
friends_count (int)		

Table 1: Database Tables

Once our database has been established, we move on to the actual data import process. The code block below shows the steps in the order they are completed. For a more detailed reference, please see the attached source code.

./code/twitterImport.py

```
61 posts = take(limit, response)                    # Get posts from stream
62 add_post_users(posts, cursor, connection)        # Add posts and users
63 add_mentioned_users(posts, cursor, connection)   # Add all mentioned users
64 update_retweet_count(posts, cursor, connection) # Updates Retweets
65 update_user_info(connection, cursor)             # Update mentioned profiles if missing
```

After this has been completed we have a full database filled with many unique users and posts.

2.2 Data Cleaning

After our initial data has been imported, we need to clean the data. In order for our analysis method to work, we need to acquire an irreducible[1] adjacency matrix.[2]. The first step is to establish a new database entirely using the name of the previous database. The structure of this new database is identical to the old, save it is missing the posts table.

After the database is established, we scan our old database and do three things:

1. Find the *most important* user. This is determined by who has the most edges (retweets) connecting them to the other users.
2. Find all users that have an edge between them and the most important user.
3. Recursively apply step 2 to obtain the complete network.

Step 1 is accomplished by aggregating the retweet table into absolute scores counting how many times a user was retweeted. The user with the most retweets is defined as our "most important user" and only connected users are added to the clean database.

./code/pyRank.py

```
108 for retweet in cursor.execute("SELECT * FROM retweets"):  # Grab all retweets
109     dst            = retweet[1]                           # and create dict
110     score          = retweet[2]                           # of all user scores
111     most_important = dst
112     try:
113         user_scores[dst] += score
114     except KeyError:
115         user_scores[dst] = score
116 for user in user_scores:                                   # Remove any that
117     try:                                                   # have less than 1
118         if user_scores[user] > user_scores[most_important]:
119             most_important = user
120     except KeyError:
121         pass
```

All users connected to our important user are then added to the new database by looping through user addition until the network no longer changes size from one iteration to the next.

[1] Irreducible in this case meaning that all nodes are connected to each other, and the matrix cannot be split into two or more adjacency matrices

[2] Where an adjacency matrix is the matrix representation of our network of users connected by retweets

./code/pyRank.py

```
123 cleaned_scores[most_important] = user_scores[most_important]
124 retweets = [retweet for retweet in cursor.execute("SELECT * FROM retweets").fetchall()]
125 pastsize = 0
126 current_size = len(cleaned_scores)
127 while pastsize != current_size:
128     pastsize = len(cleaned_scores)
129     counter = 0
130     for retweet in retweets:  # Grab all add to dict all users that
131         src     = retweet[0]                                    # Add to dict
132         dst     = retweet[1]
133         score   = retweet[2]
134         try:
135             cleaned_scores[dst]
136             cleaned_scores[src] = score
137             retweets.pop(counter)
138             edges.append((src, dst))
139             counter -= 1
140         except KeyError:
141             pass
142         counter += 1
143     current_size = len(cleaned_scores)
```

This then gives us a list of all users connected to the most important user, as well as the most important user itself. We then iterate through the old database and copy all relevant information from the old database to the new one if and only if the user exists in our list of connected/important users. At this point the data has been cleaned and all further analysis will be performed on the clean database.

2.3 Data Entry

Now that we have a reliable data source, we can analyze the data. This is accomplished using the R language. R is very good at handling large quantities of data, and as a result is ideal for this situation. The R code for the analysis is contained in `./code/ranking.R`, and can be run separately from the python program, however this is not recommended.

The first step to importing the data from our previously established clean SQLITE database is to create the user dataframe.[3] As part of this operation, we need to establish a list of users by querying the user table from our clean database.

./code/ranking.R

```
15 setup_users <- function(con) {
16     print("Establishing Users")
17     query      <- dbSendQuery(con, "SELECT * FROM users")
18     userresult <- fetch(query, n=-1)
19     cleared    <- dbClearResult(query)
20     size       <- dim(userresult)[1]
21     users      <- c(userresult[,1])
22     users
23 }
```

Once we have our list of users, we can then construct the dataframe by finding the weight of the user from individual database queries.

[3] In R, a dataframe is a special datatype that is useful for many applications. For more information, I would recommend `http://stat.ethz.ch/R-manual/R-devel/library/base/html/data.frame.html`

./code/ranking.R

```
25 user_weights <- function(con) {
26     print("Establishing Edge Weights")
27     users   <- setup_users(con)
28     size    <- length(users)
29     weights <- c()
30
31     get_weight <- function(con, user, i) {
32         query   <- dbSendQuery(con, gsub("%1", user,
33                           "SELECT favourites_count,
34                                   followers_count FROM users
35                            WHERE id=%1"))
36         result  <- fetch(query, n=-1)
37         cleared <- dbClearResult(query)
38         weight  <- FAVORT * result[1,1] + FOLLOW * result[1,2]
39         weight
40     }
41     lapply(1:size, function(x) weights <<- append(weights,
42                                                   get_weight(con,
43                                                              users[x],
44                                                              x)))
45     weights_data <- data.frame(user=users, weight=weights)
46     weights_data
47 }
```

Now that we have our user weights dataframe, we can determine the links between users, and construct yet another dataframe consisting of all follower links between our users.

./code/ranking.R

```
49 get_links <- function(con, users) {
50     print("Establishing User Links")
51     query   <- dbSendQuery(con, "SELECT id, followers
52                                  FROM users
53                                  WHERE followers!='[]'")
54     result  <- fetch(query, n=-1)
55     cleared <- dbClearResult(query)
56     links   <- data.frame(from=c(), to=c())
57     lapply(1:dim(result)[1], function(x)
58                               links <<- rbind(links[1:dim(links)[1],],
59                                               result[x,]))
60 }
```

We finally have all the information we need in the correct formats in order to create our matrix. Four separate functions are referenced in order to optimize our code for speed. These functions are described below.

./code/ranking.R

```
70 fill_matrix <- function(con, users, links) {
71     size     <- nrow(users)
72     rankings <- create_matrix(size)
73     print("Adding Retweet Edges")
74     query    <- dbSendQuery(con, "SELECT * FROM retweets")
75     result   <- fetch(query, n=-1)
76     cleared  <- dbClearResult(query)
77     print("    Replacing Data")
78     mclapply(1:dim(result)[1], function(x) rankings <<- replace(rankings,
79                                                                 result,
80                                                                 users,
81                                                                 x))
82     print("Adding Follower Edges")
83     query    <- dbSendQuery(con, "SELECT id, followers
84                                   FROM users
85                                     WHERE followers!='[]'")
86     result   <- fetch(query, n=-1)
87     cleared  <- dbClearResult(query)
88     mclapply(1:dim(result)[1], function(x) rankings <<- followers(rankings,
89                                                                   result,
90                                                                   users,
91                                                                   x))
92     print("Normalizing")
93     mclapply(1:size, function(x) rankings <<- normalize(rankings,x))
94     #print("Writing")
95     #write.big.matrix(rankings, "output.txt")
96     rankings
97 }
```

2.3.1 Create Matrix

./code/ranking.R

```
62 create_matrix <- function(size){
63     # Create our nxn file backed matrix
64     rankings <- big.matrix(size, size, type="double",
65                                    backingfile="rankings",
66                                    descriptorfile="rankings.desc")
67     rankings
68 }
```

This generates an $n \times n$ filebacked matrix to be populated later.

2.3.2 Replace

./code/ranking.R

```
121 replace <- function(rankings, result, users, row) {
122     #  |T
123     # -|-
124     # F|
125     row           <- result[row,]
126     from          <- row[1,1]
127     to            <- row[1,2]
128     i             <- which(users$user == from)
129     j             <- which(users$user == to)
130     c             <- strtoi(row[1,3])
131     weight        <- RTWEET * c + users[i,2]
132     rankings[i,j] <- weight
133     rankings
134 }
```

Replace takes each row in our SQL query and adds a link from retweets in our adjacency matrix in the correct position.

2.3.3 Followers

./code/ranking.R

```
 99 followers <- function(rankings, result, users, row) {
100     row          <- result[row,]
101     user         <- row[1,1]
102     followers    <- strsplit(row[1,2], ", |[^0-9]")
103     follow_count <- length(followers[[1]])
104
105     get_link <- function(rankings, users, to, from) {
106         i             <- which(users$user == from)
107         j             <- which(users$user == to)
108         weight        <- FOLLOW + users[i,2]
109         rankings[i,j] <- rankings[i,j] + weight
110         rankings
111     }
112
113     mclapply(1:follow_count, function(x)
114                                  rankings <<- get_link(rankings,
115                                                        users,
116                                                        user,
117                                                        followers[[1]][x]))
118     rankings
119 }
```

Followers takes each row in our SQL query and adds a link from followers in our adjacency matrix in the correct position.

2.3.4 Normalize

./code/ranking.R

```
136 normalize <- function(rankings, x) {
137     row          <- rankings[x,]
138     total        <- sum(row)
139     if (total == 0) {
140         total <- 1
141     }
142     rankings[x,] <- row / total
143     rankings
144 }
```

Normalize replaces each row with the row divided by its sum. This gives us a probability matrix.

2.4 Analysis

In order to determine the rankings of the users, we first construct an arbitrary vector with length equal to the number of users with unit length 1. Our matrix is of the form

$$\begin{pmatrix} 1 \\ 0 \\ 0 \\ \vdots \\ 0 \end{pmatrix} \tag{2}$$

This matrix is also a filebacked matrix, and uses 200-bit precision to allow for accurate vectors.

./code/ranking.R

```
173 A        <- big.matrix(nrow(users), 1, type="double",
174                                        backingfile="A",
175                                        descriptorfile="A.desc")
176 A[1,] <- 1
177 A <- A[,] / mpfr(norm(A[,]), precBits=200)
```

We then pass the vector and matrix to our power method. Which runs however many iterations are specified. This has two steps per iteration:

1. Establish our new vector as our original matrix multiplied by our vector.
2. Normalize our new vector, and set the old vector as the new one (to be used in the next iteration).

./code/ranking.R

```
150 power_method <- function(rankings, A, iterations) {
151     for (i in 1:iterations) {
152         B <- rankings[,] %*% A
153         A <- B / norm(B[,])
154     }
155     A
156 }
```

3 Analysis

4 Implications

5 Discussion

This study shows how a ranking system could be applied to Twitter and give a ranking for users based on their influence. Through application of matrix methods, users on Twitter were ranked according to retweets, followers, and favorites. The ranking system was created through the application of the Perron-Frobenius theorem as well as the use of power method. For this particular study, a smaller data set was used due to the complexity of the ranking system as the matrices increase in size. A larger data set would also be hard to model because some users are hardly connected to twitter and donâĂŹt have much information. Their data might skew the system unless a significantly large data set is analyzed. The successful creation of the Twitter influence ranking shows how other ranking systems can be applied to almost anything.

1 Attachments


LaTeXSource Code