Twitter User Recommender for Topics using Graphical Database

GROUP 7

Anton Lindqvist antoli@kth.se
Anna Lindelöf anna@kth.se
Thiago Lobo thiagol@kth.se
Helder Martins helder@kth.se
Casper Renman casperr@kth.se

Abstract

1 Introduction

2 Background

3 Related work

4 Method

4.1 Crawling Twitter

In order to generate a significant dataset for our queries, we had to design a Python script which acts as a *crawler*, going through Twitter data and storing it. That is done through the Twitter API [7]. To do that, one has to sign up as a developer in twitter and obtain client credentials so that access to the API is granted to the app. The first step of the crawler script is to use these credentials so as to obtain an access token.

With access granted, our application is able to execute HTTP requests by means of the Requests library for python [5]. There are many possible requests offered by the Twitter API but we only used one such request, which queries the newest 100 tweets containing at least one of the hashtags given as input. It has the form: *https://api.twitter.com/1.1/search/tweets.json?q=hashtags&count=100&result_type="recent"&lang="en"*, where "hashtags" is a string containing all the hashtags we decided were relevant for our USA elections context.

So the basic working of the script consists of an infinite loop where this request is made and the returned *json* is parsed and interpreted by a set of functions which add *Tweets*, *Users*, *Hashtags* and *Words* to our database in a recursive way (i.e. if one tweet retweets a tweet that mentions a user, all this data is going to be properly processed and stored). Also, proper term extraction and stemming is done, before storing *Words*.

4.2 The Neo4j database

In order to store all the information fetched from Twitter, a suitable database had to be chosen beforehand. We wanted a solution where the relationship

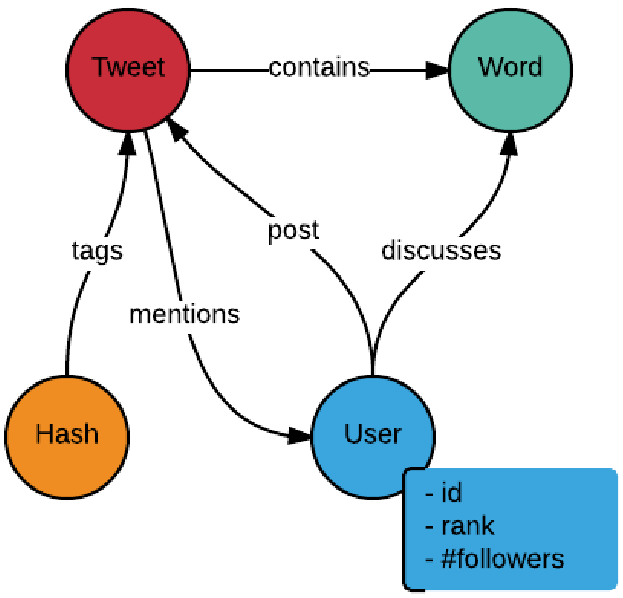

Figure 1: Graph schema used for the Twitter data.

between the different entities could be easily seen and queried, and that could effortlessly store the large amount of data that we planned to fetch. Our approach utilizes a *graph database* as a solution to our storing problem, more specifically a software called *Neo4j* [3]. This software allow us to faithfully represent the full structure of the Twitter database as a graph, where each different entity is a *node* and the relationship between the nodes an *edge*. Figure 1 details the nodes and the edges of our recommender engine.

The base node of the graph is the *Tweet*, which is a free text short message sent through Twitter. Every tweet is posted by an *User*, which is represented by the *post* relationship. A tweet may *mention* another user with the @ special character, and it may also *tag* a topic with #, represented by the node *Hash*. To this basic schema derived directly from Twitter, we added the node *Word* which are all the parsed words in the free text of the Tweet, properly tokenized and stemmed. This node is linked to the rest of the graph through the *contains* and *discusses* relationships, with a property specifying the amount of times this relationship happens for every tweet and user. This new entity allows us to easily represent the user as a *bag-of-words* document of every word that he discusses about in all of his tweets, thus allowing us to map the recommendation problem to the standard techniques used on information retrieval.

4.3 Parsing tweets and extracting topics

The goal of the project is to recommend users given topics. In order to recommend a user, the user needs to be associated with the topics the user talks about. Therefore the users tweets are parsed and the topics of the tweets are extracted. The topics are extracted by parsing the freetext of the tweets and extracting the nouns and adjectives. The choice of extracting nouns and adjectives was an empiric decision made by the group.

Extracting topics from tweets is done using the Natural Language Toolkit (NLTK) [2] which provides interfaces in Python for things like classification, tokenization and stemming.

4.3.1 Cleaning tweets

A tweet can contain hyperlinks, hashtags, mentions and other symbols. These are removed in order to properly parse the text of the tweet. Specifically, words starting with *#, @, & or http* are ignored. A few other words that commonly occur in a tweet were also ignored as they would not contribute to the cause. These are *don't, i'll, retweet and rt.*

4.3.2 Extracting nouns

The nouns (topics) are extracted by performing the following actions, provided by NLTK:

1. Lowercase all letters and tokenize the text into separate tokens
2. Remove words that are shorter than three characters (This was also a decision made by the group)
3. For each word, remove ignored symbols and words starting with a ignored symbol
4. Part of Speech-tag [6] the words
5. Pick the words that are tagged as `NN` (noun) or `JJ` (adjective)
6. Stem the words and return the result which is a list of words

4.4 PageRank

One of the most well-known ranking and scoring measures is called PageRank [4]. Made famous by Google in late 90's, its main idea is to use the auxiliary information, mainly the *link structure*, present in the World Wide Web as an *authority measure* of the web pages contained within. Representing the web as a graph were each node is a web page and the edges the links between a page and another, it is intuitive to see that nodes with higher number of *inlinks* (that is, the number of links arriving into a node) are of higher importance than the ones with no inlinks at all, just like a scientific article which is cited by several different sources, for example.

One of the core mechanics of the algorithm is the propagation of ranking through links. That is, for every outlink of a web page in the graph, its ranking is distributed evenly among all of them. That covers both the cases when a web page has several different low-ranked inlinks or when it has few high-ranked ones, they may have similar ranks since it is not the count that matters.

The PageRank algorithm outputs a probability, that is, the chance that an imaginary user will arrive at that web page, starting at any random node. The user interactions with web pages can be seen as a set of *random walks* where each user follows a link until they are done surfing or they are *bored* of following links and jump to another random web page instead. This bored state, which is also a probability and normally taken as 15%, is important as to also give a score to web pages which have no inlinks at all, for example web pages just recently created.

4.4.1 PageRank in the Twitter graph

Although the original PageRank algorithm was modeled with focus in the World Wide Web, its method could be applied to any problem which can be modelled as a graph. Specifically for Twitter, one could see each *user* of the platform as a node and every *mention* in the tweets of a user to another as a link. In the same way that web pages with high number of inlinks have a higher rank, users that are mentioned frequently will be considered more relevant for our recommendation engine, this process can be more clearly seen in Figure 2. Note that we actually do not analyse the content of the tweet, so tweets with positive or negative sentiment will have the same importance for ranking, one could think of it being a "any publicity is good publicity" kind of model.

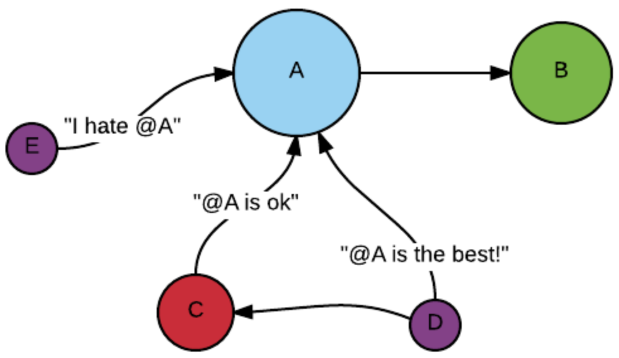

Figure 2: PageRank applied to the Twitter database. Every user is a node, while every mention in tweets is an edge. The size of the node is its relative rank among others.

The original PageRank algorithm considered following an outlink with equal probability among all the possible links. That is reasonable with the unstructured meta information available in the Web today, but is intuitive to reason that, with more information about these users, different probabilities could be applied to each one of them, depending on the task that we have at hand. For a user recommender engine, our approach used the *number of followers* as a good measure of importance. That is, users with high number of followers will be jumped to with higher probability in the random walk,so their score will be naturally higher. Our engine implemented both methods for evaluation, and the results are reported in the Experiments section.

4.4.2 PageRank Monte Carlo

The standard implementation of the PageRank computation is done via a method called power iteration, which involves finding the largest eigenvector of a transition matrix $\mathcal{P}$ composed of the transition probabilities between every web page of the World Wide Web, a process which is done over several iterations until convergence. This method, although popular and still used today by Google, has its drawbacks mainly regarding the speed of convergence, several passes may be needed until the desired precision is obtained. In our approach we explored a relatively new method, which utilize *Monte Carlo algorithms* to estimate the score of the nodes of the graph. As proposed by Avrachenkov et al. [1], the idea is that, if we sample the web page after a sufficient large amount of random walks, a probability distribution could be calculated with an acceptable degree of precision and with a faster convergence rate. While the power iteration method may require more than 50 iterations for an acceptable ranking to be reached, Avrachenkov et al. proposed method has ranks for the import pages after one iteration only.

Of the several different algorithms proposed, our engine implements the *Monte Carlo complete path*, which is detailed in the Algorithm 1. For every user in the Twitter database, we start a random walk beginning in that user and ending when the user is bored of following mentions. We keep the track of the total steps of all random walks and how many times each user was visited. A new user is selected to be followed in the walk from all the users that he mentions, which can be done applying equal probabilities to each one of them or with increased chance for higher number of followers. If a user does not mention anyone, we consider it a *sink* and jump to any other user in the database with the same method. After every user was at the beginning of the random walk for a set number of walks, we calculate the user rank simply by dividing the number of times each user was visited over all random walks by the total steps taken.

Algorithm 1 PageRank Monte Carlo, complete path

```
 1: procedure PageRank
 2:     for all walks do
 3:         for all user in users do
 4:             username ← user['username']
 5:             bored ← False
 6:             while ¬bored do
 7:                 totalSteps ←totalSteps + 1
 8:                 userSteps['username'] ← userSteps['username'] + 1
 9:                 mentions ← getUserMentions(username)
10:                 if mentions ∈ ∅ then
11:                     username ← getRandomUser(users)
12:                 else
13:                     username ← getRandomUser(mentions)
14:                 bored ← isUserBored()
15:     for all user in users do
16:         username ← user['username']
17:         ranks['username'] ← userSteps['username'] ÷ totalSteps
18:     return ranks
```

4.5 TF-IDF

Ranked user retrieval can be implemented by only using the aforementioned *PageRank* algorithm but by doing so, any query would return the same top listed users. While that might be interesting in some applications, that is not the case in our context. The words in the query should also be used to filter and rank the retrieved users.

TF-IDF is a well known solution to the problem of matching (in a ranked way) documents modelled as *bags-of-words*. Each document (including the input query) is represented by a vector of scores, each of which related to one of the possible terms in our dataset. The scores are calculated as follows: $tf_{w,d} * log_{10}(\frac{N}{df_w})$ where $tf_{w,d}$ is the number of times term w appears in document d, N is the total number of documents and df_w is the number of documents term w appears in. Then, *cosine-similarity* is used to compute how close the query is to each of the documents.

In our implementation, *User* nodes are documents containing each of the *Word* nodes they are linked to. This link contains the number of times this *Word* has been discussed by this *User*, that is, a $tf_{w,d}$ score. The final procedure can be seen in Algorithm 2.

4.6 Final Score

After retrieving the sets of *PageRank* and *TF-IDF* scores for all users that discuss any query term, we decided to first normalize them to zero-average and unitary variance so that they can be mixed together by means of a parameter α. Then, the final score, for each user u, is: $s_u = \alpha * \bar{s_{p_u}} + (1 - \alpha) * \bar{s_{t_u}}$.

4.7 Graphical user interface

5 Experimental results

5.1 Ranking algorithms

5.1.1 Only PageRank

resulting list/table

Algorithm 2 TF-IDF in a Graph Database

```
 1: procedure TF-IDF
 2:     $scores \leftarrow \emptyset$
 3:     $sizes \leftarrow \emptyset$
 4:     for all token in query do
 5:         $users \leftarrow$ graph.query(users that discuss 'token')
 6:         $df \leftarrow$ length(users)
 7:         $count \leftarrow$ # of occurences of 'token' in 'query'
 8:         $wtq \leftarrow count * log_{10}(\frac{length(documents)}{df})^2$
 9:         for all user in users do
10:             $score \leftarrow wtq *$ graph.query(# of times 'user' discusses 'token')
11:             if $user \notin scores$ then
12:                 scores['user'] $\leftarrow$ score
13:                 sizes['user'] $\leftarrow$ graph.query(# of words discussed 'userm')
14:             else
15:                 scores['user'] $\leftarrow$ scores['user'] + score
16:     for all user in scores do
17:         scores['user'] $\leftarrow \frac{scores['user']}{sizes['user']}$
18:     return $sort(scores)$
```

5.1.2 Only tf-idf

resulting list/table

Small reflection (relevance feedback). What do we think? What should alpha be?

5.1.3 Combination

Try 3 different alphas and show list/table

5.2 Evaluation

Summary of what alpha should be and why.

6 Evaluation of the result

7 Summary and Conclusions

References

[1] K. Avrachenkov, N. Litvak, D. Nemirovsky, and N. Osipova. Monte carlo methods in pagerank computation: When one iteration is sufficient. *SIAM Journal on Numerical Analysis*, 45:890–904, 2007.

[2] Steven Bird. Nltk: the natural language toolkit. pages 69–72, 2006.

[3] INC Neo Technology. Neo4j graph database, 2007.

[4] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. Monte carlo methods in pagerank computation: When one iteration is sufficient. *World Wide Web Internet And Web Information Systems*, 54:1–17, 1998.

[5] Kenneth Reitz. Requests: Http for humans, 2016.

[6] Helmut Schmid. Probabilistic part-of-speech tagging using decision trees. 12:44–49, 1994.

[7] INC Twitter. Twitter api, 2016.