Project Members

Eric Kimbrel, Erik Turnquist, Zack Allred, Rob Hanlon

Goals for Project

When designing this project we wanted to create something that we would use and could recommend to our friends. Most of us had used Twitter previously so we began to brainstorm what additional features we would like added to to Twitter. Twitter is great at finding interesting facts about all Twitter users, but it does not allow you to easily discover interesting local trends from your followers. We developed a very ambitious list of features which included:

- Visualization of Twitter relationships using the HTML 5 canvas tag
- Determining similarity between different Twitter users
- Given a list of users and their tweets determine trends in their conversations
- Recommend followers to a given user (those with high similarity)
- Identify groups of users that tweet about similar topics

We thought that providing an interesting UI was paramount in all of these features, which is why a dynamic graph view was chosen as the main UI feature. It allowed us not only to show similarity in an intuitive way but it also allowed the user to easily select users for trend analysis. Although not all the features initially proposed were finished, the essential core features were. These include the dynamic graph visualization, determining similarity between users, and determining trends given a group of users and their tweets. The remaining features would be built on top of these fundamental features and could be implemented in the future.

System Design

The system were designed to be as decoupled as possible so experimentation could more easily take place. There were two main parts to the system:

- The frontend written in Ruby and JavaScript
- The backend server written in Java

The frontend runs on JRuby on Rails. We made this design choice for several reasons:

- Ruby on Rails is a well-regarded framework that makes it very easy to build a web app
- Java is more appropriate for doing fast calculations
- JRuby allows for a bridging of the two, and allows for instantiation of Java objects directly from JRuby.

In addition to using JRuby on Rails, we made use of several Ruby libraries to make our lives easier. First off, we used Resque (http://github.com/defunkt/resque), a Redis-backed Ruby queueing system for backgrounding database and Twitter API jobs. These jobs store JRuby-wrapped Java objects in a central Distributed Ruby object queue, and operate upon them as needed. In order to facilitate OAuth authentication with Twitter, we used the twitter-auth Rails plugin (http://github.com/mbleigh/twitter-auth), which made adding OAuth to our frontend a five minute process.

The UI is built atop the `canvas` DOM element, a new element introduced by HTML5. The graph is automatically laid out in a loosely force-directed manner, which prevents nodes from overlapping and allows graph layout to be determined purely by the similarity score between users. To allow for this, we combined Processing.js (http://processingjs.org), a framework that provides many drawing primitives in order to easily draw upon a `canvas`, as well as a port of the Traer Physics library (http://www.cs.princeton.edu/~traer/physics/) that we wrote, to the end of node positioning. Lastly, we used the popular jQuery JavaScript framework to add and remove text from arbitrary DOM elements.

The back end server is made up of three main parts. The first consists of updating the data loaded from Twitter, the second determines possible trends, and the third finds similarity between users.

To update a users data new friends and followers as well as their tweets must be loaded. This part gathers data from the twitter API using twitter4j, a java interface to the Twitter API. How it gathers data and processes it is outlined in the algorithm portion of the report. This process is run continually in the background so that a users data will always be up to date, and also when a user first logged in.

The trending engine had to major iterations. The first used word counts (computed after the tweets were updated) as a simple way to determine popular words among a group of users. Twitter distinguishes special words tags with a "hash tag" as topics so if found those were automatically included in the trend. The second iteration used as a basis the Yahoo Term Extractor API which when given text attempts to determine important words.

To determine similarity, a user's tweets are treated as a document, and then the documents of both users are compared. At first we simply did a euclidean distance on the documents, however through experimentation we found that cosine similarity of the tf-idf vectors because it took into account the document frequency of the terms.

Algorithms

Automatically Updating Twitter Data

The following is the basic methodology for obtaining new tweets and users from the Twitter API and inserting them into the database. The following values are used as constants in the algorithm and have been tuned for the optimal performance for the network and load delay times:

Friend/Followers to pull on each process task: ff_process_num
Tweets to pull on each process task: tweets_process_num
Max friends/followers to keep for each user: ff_max_num
Max tweets to keep for a user: tweets_max_num

1. Get the list of every user that has logged into the app
2. Retrieve a set of
   ff_process_num
   friends/followers that have not been added to the database, making sure not to exceed
   ff_max_num stored in the database.
3. For each new friend/follower, get a list of the newest
   tweets_process_num tweets
4. For each existing user get the newest tweets (making sure not to obtain tweets that have already been added), up to
   tweets_process_num.
5. If the total amount of tweets for a user exceeds
   tweets_max_num, then delete the oldest tweets so there are only

tweets_max_num left.

Similarity

While two methods were implemented for similarity only one is used in the current code.

Tf-idf Cosine Similarity:

```
Similarity(twitterer_1, twitterer_2)
    1. tf_1, tf_2 <- the word counts (sorted in ascending order) of twitterer_1 and twitterer_2 respectively (from the database).
    2. df <- the document frequency of every term (retreived by an sql query in DBConnection.java)
    3. D <- the number of twitterers in the database (retreived by an sql query in DBConnection.java)
    4. tf_idf_1, tf_idf_2 <- currently empty td-idf vectors.
    5. For each term, t, in df:
           idf <- log( D / df(t) )
           tf_idf_1(t) <- t in tf_1 ? tf_1(t) * idf : 0
           tf_idf_2(t) <- t in tf_2 ? tf_2(t) * idf : 0
    6. return (tf_idf_1 * tf_idf_2)/(|tf_idf_1| * |tf_idf_2|)
```

Trends

Two methods of trending were implemented. One based off Word Counts and another.

Solution 1: Word Count based approach to Local Trending:
With this method we pull trends directly from the words tweeted by users. In twitter words marked with a '#' are used to indicate a topic, and so all words with a '#' are considered for trends before considering any other words. The algorithm is as follows:

1. Convert a users tweets into rows in a Database: (user_id,word,count)
2. When n trends are requested for a set of users S:

```
SELECT word
FROM wordcounts
WHERE word like '#%' and id = 1 or id = 2 or .....
GROUP BY word
ORDER BY sum(count) desc
limit n;
```

If fewer than n trends are returned we then consider all words (not just those with a '#') to fill in the required number of trends. To ensure more meaningful results words fewer than 5 letters are not considered.

Solution 2: Topic Extraction based approach to Local Trending:
The Topic Extraction method converts a users tweets into a set of topics using Yahoos term extraction API (http://developer.yahoo.com/search/content/V1/termExtraction.html). The result is simply a list of topics for each user. The topics are not weighted via frequency, they are simply extracted from the text.

The topics returned are then stored into rows the a Database: (user_id, topic)

When trends are requested for a set of users the topics shared by the greatest number of users is returned. If topics are not shared between users then some topics will be returned, but they may or may not be meaningful to the group.

Benefits and Disadvantages: a set of subjective measurements to show how well the different methods work.

	Word Count	**Term Extraction**
Users have explicit control of trends	**Yes, users can mark things with a # to say its a trend**	**No. Users words are converted, # tags are ignored**
Frequency of topics matter	**Yes, topics that get talked about a lot are reflected heavily**	**No. Shared topics are reflected, but they are not necessarily the most talked about topics.**
Trends actually reflect what people are talking about	**Not Really.. Most users don't use the # enough, and one word does not really capture a topic.**	**Yes. Returns rich terms that really show what people are saying.**
Small group trends	**Works well since word frequency still reflects what is talked about the most.**	**Works poorly because small groups may not have overlap.**
Large Group trends	**Works well.**	**Works well.**

Usage Scenario

A social media professional with an active presence on Twitter would like to know how well he or she is connected with her or his followers. However, this person would like to have this experience in a more visual way, that lets him or her actually see how close the connections are. In this case, a pure listing of users organized by TF-IDF score would not be as fulfilling, in terms of being able to conceptualize a relationship. A network graph, of this person's followers and friends, would be the ideal visual representation, but a raw, uninformed network graph would be useless. By weighting according to similarity, this social media professional could have the ability to see with which of the users they should focus their efforts on making strong connections. In addition, by highlighting a group of users, especially closely related ones, this professional could hone in on what conversations are being had in a quick manner, and could easily go forth and do research on said topics to the end of being more closely involved in a conversation. Our tool could thus be an integral part of a business, allowing quick and informative glances at a business's social network.

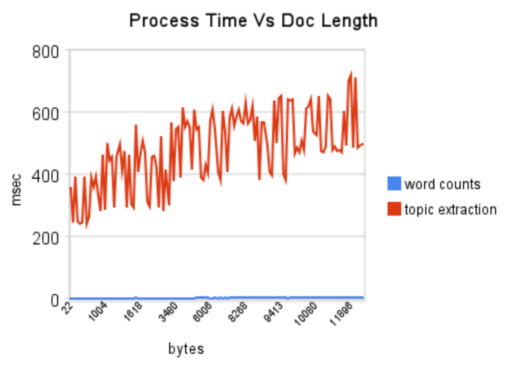
Experiments

Experimental Data on Trending -

Phase 1: Consider all of a users tweets as a single document and translate the document into word counts or a list of topics.

Experiment Description:

1. For each twitter user in the database, concatenate each of their tweets into one document.
2. Note the length of the document.
3. Measure the time required to count the use of each word in the document
4. Measure the time required for Yahoo to return a list of terms from the document.

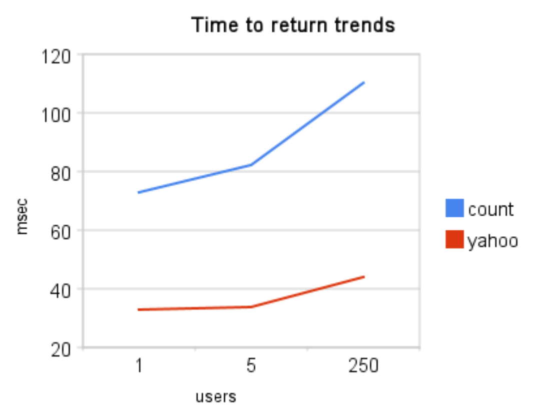
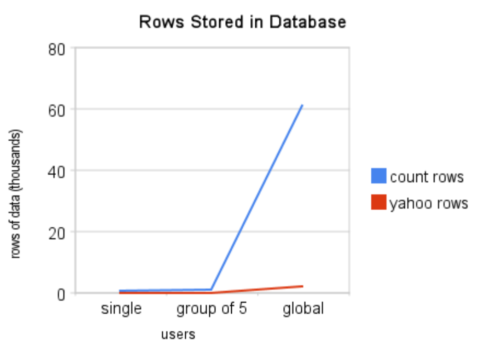
	Processing word counts is fast and efficient. Term extraction takes significantly longer to complete as document size increases. Some of this increase is likely due to network latency while using the Yahoo API.

Phase 2: Return trends for a group of users

Experiment Description:

1. Create 3 user groups for testing. One with a single user, one with 5 users, and another with 250 users. These group sizes were selected to show the overhead of using the mechanism, as well as the increased work load due to adding additional users to a query.
2. For each user group ask for the trends 100 times and take the average measurement.

3. Look at the rows in the database tables to determine the storage space required by the different algorithms in order to return trends of each group size.

	The Term Extraction version returns trends significantly faster for both small and large user groups.
	The difference in time is due to the amount of data that is stored in the database. The number of topics stored for a group of users stays small, while the world counts grow quickly.

Difficulties/What We Learned

Trending Difficulties

Finding trends from natural language is much more difficult than we first expected. Most intelligent solutions such as suffix tree clustering require an initial set of labeled data, or a previously defined list of categories. Since we wanted to discover trends from such a diverse set of documents (tweets) it wasn't clear how to use these well known techniques. Instead we implemented a more naive solution based off of word frequency.

Twitter4j Difficulties

During development of the background automatic update functionality Twitter changed the way they paginate results, which broke the twitter4j library. After much investigation into the Twitter API and the twitter4j source code, it was found that instead of specifying page number (which simply would increment) Twitter chose to reply with a random number that would be used as an index into the next page. After changing the correct source files and building the library, the changes fixed the pagination issues. As a result we were able to unexpectedly contribute to an open source project.

UI Difficulties

The canvas tag and browser JavaScript engines are not ready for widespread use in heavily animated graphs with a large amount of nodes. Initially, we were inspired by demos of the canvas tag found created by 9elements and Bomomo (see appendix for links), which showcase eye-pleasing animations done purely in HTML5. Upon further inspection of the aforementioned sites, however, one can see that the 9elements demo has either simply or hard-coded particle updates, and Bomomo actually has very few moving parts. Attempting to draw a large force-directed graph of nodes using a physics library using a single thread of execution within a browser window severely limited the size of graph that could automatically be built and laid out. In order to the make the graph actually usable and animated, one may have to bite the bullet and create the UI in Flash or Java. If canvas is a must, the server could maintain and calculate graph state, and the graph could remain in sync via AJAX calls. Another option would be heavy optimization of the physics library, as well as reducing the number of moving parts on the graph. This would require the most research and toying, most certainly.

Conclusions and Future Ideas

By providing a visualization tool on top of the twitter network, and allowing user selection of trending groups, we have made twitter data more discoverable and usable. Several Challenges arose and provided opportunities for future work.

Future Work

1. Scale - One Server and a relational database clearly is not capable of supporting a large scale Internet service. To increase scalability we would distribute processing and data across a cluster of machines using Hadoop. The UI faces scalability issue as well, such as drawing graphs to show larger portions of the twitter network. This is a computationally heavy process that requires significant work to come up with an elegant solution.

2. Trends / Similarity - Basing these metrics off of word choices provided a good way to start the project but significant improvements could be made. Machine learning techniques that used labeled training data to intelligently select topics from user tweets would lead to better trends discovery and an improved similarity rating.

3. Additional Features - We would like to add more features to the project that might be of use for twitter users. For example clustering users based on their similarity, searching for users who tweet about certain topics, recommending friends, etc. Extensions of the network graph could make it possible to highlight clusters of users, making it easier to find similarities in closely related groups, and highlight levels of connection by depth. This would be an interesting problem; that is, making tree highlight by depth an intuitive interface element could be difficult. Another future feature that would be useful would be the ability to restructure the graph after hiding users, in order to be more clearly see users that have not yet been discovered.

Appendices

Work Distribution

Erik - Designed a background service that automatically pulls new data from Twitter.
Eric - Worked and implemented various trending algorithms.
Zack - Worked and implemented a similarity engine.
Rob - Designed and implemented the UI.

Links

Bomomo: http://bomomo.com/
9elements Canvas Demo: http://9elements.com/io/projects/html5/canvas/

API/Libraries

twitter4j - A Java library for the Twitter API
Yahoo Term Extraction

Instructions for Code Usage

OS Requirements
- Linux/Unix based
- Mozilla Firefox
- Java
- Ruby 1.8.7 and JRuby 1.4.0

The README.txt is in the root directory of the project folder