FootTraffic

A Traffic-Driven Location-Based Web Search Engine

Checkpoint

Department of Computer Science & Engineering
Texas A&M University

Authors:	*Emails:*
William Chen	wchen16@cse.tamu.edu
Eric Wood	ebw0178@cse.tamu.edu

November 14, 2011

1 Milestone Walk

We set out a base set of technologies and resources we want to use, and in this milestone, we installed, configured, and learned them more in-depth so we can use them appropriately. We were off on a slow start, and that was expected. A majority of the code written so far has been devoted to creating our infrastructure; wrappers have been built to make accessing APIs simple and extensible, and logic for generating traffic patterns is already in place. Our development so far has taken place in a private Github repository, and we've already created a production environment with Heroku. We've decided to house our main databases on our personal servers.

2 Milestone Jog

During this milestone, we planned out our backend strategy and briefly brainstormed how the UI looks like and what features we should include/concentrate on. There are many uses for a traffic-pattern based searches, but right now we are limiting ourselves with the scenario where you would like to find similar [1] venues in the surrounding area.

In addition, we have kicked off a rake task that iterates through a 22 million lines of data, provided by the research paper, into a sqlite3 database. The conversion took about a week, and the database size is around 3 GB. Architectural changes during that week have prompted us to migrate to a MySQL database, which will allow us to operate on the data in a distributed fashion; MySQL allows for remote connections.

2.1 Backend Approach

There will be two databases for storing incoming tweets from Foursquare and for storing location/venue data. `Checkins` database will serve as a buffer between a script that is rate-limited (to conform with Twitter API) and a tweet processor. In other words, the script will dump all the relevant tweets it can scrape into the `Checkins` database, and on the other side, a processor will parse and add additional information to the tweet, which is explained in the next part.

If the `Checkins` database is not empty, the processor reads every tweet, parses it, fetches additional information, and adds it to the second database, `locations`. When processing a group of tweets, it hashes them based on their location ID and from the DateTime associated with each checkin generates a traffic pattern. Every location has an unique ID from Twitter, and we use that to query the `locations` database. If such location exists, append the traffic and recompute the pattern by averaging the two patterns together. If not, start off parallel

[1]Similarity is based on a combination of traffic-pattern and category. Relevancy is based on geospatial location radius

threads to third-party APIs and fetch all the information there is on that location. For example, we ask Google what the category, address, name, boundary box (for drawing on maps), and etc. If the location is a restaurant, we can ask Yelp for reviews, ratings, hours of operations, and anything else we can get our hands on. We're currently using a Rails plugin named Delayed Jobs for queueing, allowing each request for "metadata" for a location via an API to get run independently. With this approach we can process data much quicker since it avoids bottlenecks from HTTP requests which block. Once things are cemented we plan on running consumers on at least three other computers we own and having them process HTTP requests in the queue.

3 Other Thoughts

If you would like to view our private Github repo, we would be more than happy to grant you access.