CSE 454
Final Report
TasteCliq

Samrach Nouv, Andrew Hau, Soheil Danesh, and John-Paul Simonis

Goals

Your goals for the project

Create an online service which allows people to discover new media based on their current likes, with a current focus on movie recommendations.

Design - System and Algorithmic

Your system design and algorithmic choices

Facebook Application

The service is implemented as a facebook application. This allows us to authenticate users using their facebook account, and programmatically retrieve their favorite movies (and other media) from their facebook account. The website is implemented with Ruby on Rails, and shows up in a frame within facebook.com.

Library Page

Purpose: provide portal for (1) adding items to library (via a search bar), (2) for viewing items currently in library, and (3) for viewing recommended items at a glance.

Algorithms used: for #3, using a SlopeOne recommendation algorithm which compares current user's ratings to others' ratings, and predicts how current user would rate other items in the database. We sort by the highest predicted rating and return the top n movie/media results.

Recommendations Page

Purpose: (1) display netflix-style media recommendations based on the user's current library, and (2) display a particular item and its associated similar items.

Algorithms used: for #1 and #2, we calculate the Pearson Correlation Coefficient of two items' ratings to estimate the degree to which two items are positively correlated in terms of rating. We then precompute the correlation coefficients for all items and cache the top 10 (or so) similar items for each item. This allows for fast retrieval of item similarities.

For #1, we first group the user's library items by their similarity. Then, for each group, we roll up the similar items for each item in the group, and display those items next to the group of items.

Recommendation Service

We used Apache Mahout Taste, a "flexible, fast collaborative filtering engine for Java" to generate our recommendations. Taste supports the most common collaborative filtering algorithms for user-based, item-based, and other classes of algorithms.

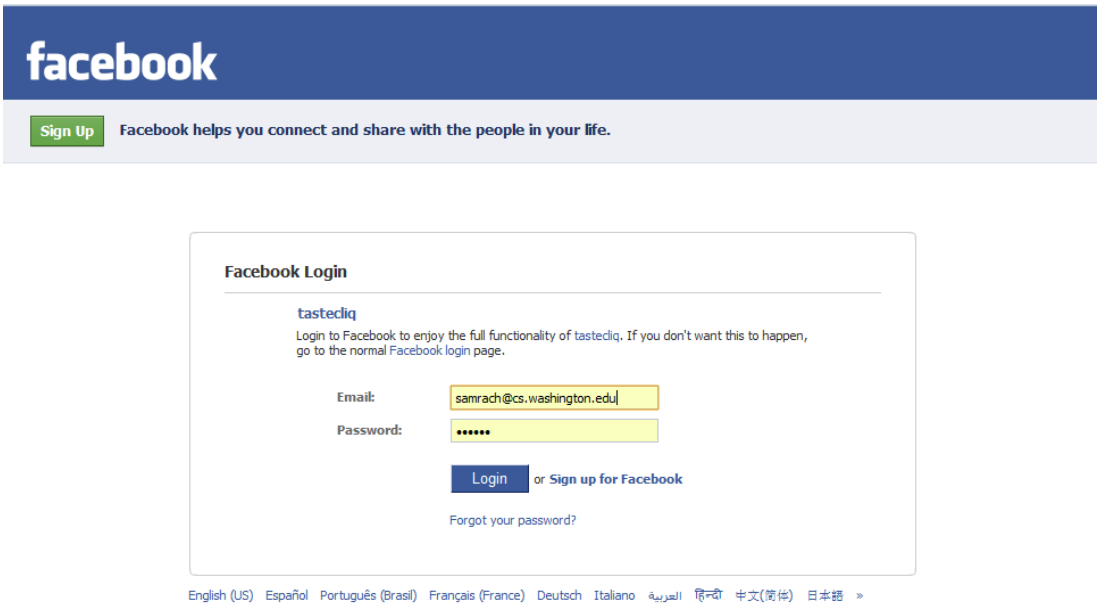
To facilitate communication between the Ruby on Rails application and the Java Taste library, we implemented a web service in Java which takes HTTP-based queries and returns recommendations computed by the Taste library in XML form.

Usage Scenarios

Sample screens of typical usage scenarios

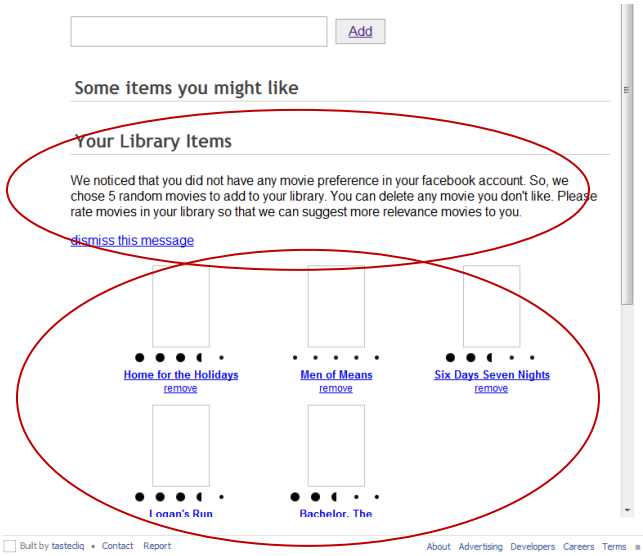
Login or start Tastecliq:

http://apps.facebook.com/tastecliq



User's favorite movies are imported automatically:

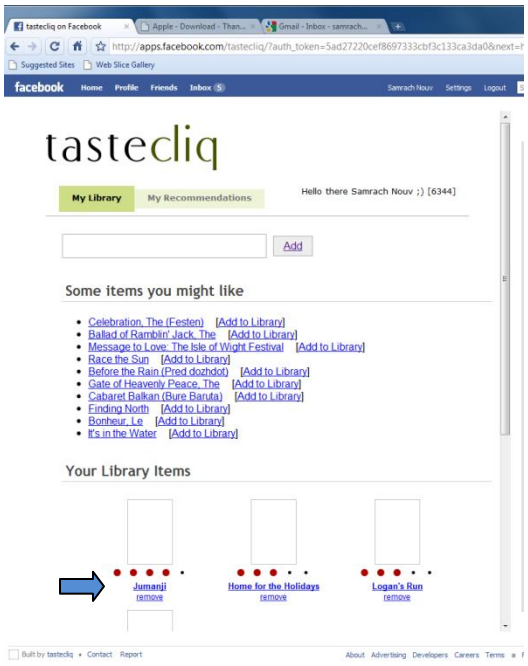
When a user has no favorite movies, Tastecliq notifies and adds five movies are randomly added to user's Tastecliq library:



Users can add or remove items from their Tastecliq library. Tastecliq also auto completes item names while users are typing:

As users rate more movies, more movie recommendations show up:

Users find similar movies to a particular movie by clicking on a movie's title:



Users can also find similar movies to those in their library by clicking on "My Recommendations" tab:

Experiments

Experiments and results that show how effective your system is, and where it could be improved.

To measure the accuracy of the slope one recommendations we devised the following experiment:

We took a rating from each user and treated it as the test data set. The remaining ratings were used as a training data set. We then generated recommendations for each user using the training data set and calculated the average difference between the user's actual rating and the recommended rating.

The average difference between actual and recommended movies was 1.7 / 5. We also tried decreasing the size of the training data set which resulted in less accurate recommendations. This was to ensure that the recommendation algorithm was actually working.

To test the item-item similarity algorithm, we chose pairs of movies from our data set that we thought were similar and then used the item-item similarity algorithm to generate 15 similar movies for one of the movies in the pair. The other movie in the pair was present in the recommended list 70% of the time.

Conclusions and Ideas

Anything you considered surprising or that you learned. What would you do differently if you could?

Not all recommendation algorithms are made alike, and each has a particular purpose. Knowing more about each algorithm earlier would have provided us more time to implement better/different recommendation algorithms, and optimize them. This article by authors at the University of Minnesota offered more information on exactly how each algorithm works.

Whatever algorithm you end up using, it is important to analyze whether or not it is recommending items that are relevant. This requires both a good definition of relevancy, depending on your business needs, and a good dataset with which to test the recommender. In our project we had a good dataset to work with (grouplens), but being unfamiliar with recommendation systems in general, didn't realize early enough the value of evaluating (especially qualitatively) the effectiveness of our recommender. We also decided not to use movie metadata in our project, which would have improved the relevancy of our results by preventing recommendations from being too far away from the item in question.

Performance also turned out to be an issue. Building models from which one can generate recommendations takes a lot of time (minutes to hours) when working with a huge dataset, which means that implementers need to scale up hardware, and scale down and constrain the volume of data that is being processed. This means prioritizing more popular items, trimming or excluding irrelevant items from the model dataset, and caching recommendations.

Ideas for Future Work

Hybridization of recommendations

Our recommendation service was limited by only using one type of recommender, which had no access to genre and author/director data. We think that hybridization of recommendations (including alternative recommendation algorithms which would function with other item metadata) would have increased the usefulness of the website. One problem we had was that recommendations were often not in the same genre and not by the same author/director. This is because of the inherent unreliability of similarities based solely on (user, item, rating) tuples.

The idea here is to consider all the different types of things users are searching for. They are searching for items by the same author, items in the same genre, and also items that were rated similarly (these items might be more of a stretch than same-author and same-genre). An empirical look at Amazon's recommendations shows that they mix same-author, same-genre, and also similar-rated items. Their similar-rated items seem to be based on those-who-bought-this-also-bought-this style recommendations. Also, this style of recommendations tends to pick up the same-genre and same-author items without specifically using those two recommendation algorithms.

User Neighborhood - Extending site for social networking

Algorithms for calculating user neighborhoods (and getting a particular user's nearest "taste" neighbors) were available through the Taste library. An obvious extension of our current project would be to connect users by showing them other users with similar tastes. Or, in a non-social application, creating "interest graphs" which summarize the breadth of categories of items that a neighborhood of users like.

UI Extensions - Addressing limitations of the Textbox

Recommendation systems are more likely to be used if they are built on top of an activity that users already do. (Examples include facebook favorites lists, streaming music sites like Pandora and lala, commerce sites like Amazon.) In other words, forcing the user to type an item, search for it, and add it to a library is unreasonable, barring some other incentive to use the product.

To get around this, while still providing a recommendation-centric product, companies like Glue have developed browser addons which include recommendations in the user's normal browsing experience. This product allows users to add their favorite items to Glue from any webpage they visit, instead of limiting users to a one-website experience. Also, it integrates seamlessly with Facebook and Twitter with their open authentication systems. This allows for an instant social experience, since there is no need to create an entirely new community of users.

Randomization of Recommendations:

A certain level of randomization is required for pages that have a high repeat-view rate, so that users don't tire of old content.

This is heavily dependent on the application in question, but in our application, a certain level of randomness was desired for our SlopeOne (general) recommendations, and also our item recommendations. To do this, we could have experimented with returning not only the top results, but also lower-similarity results. E.g. currently, our current implementation recommends about 10 items which all have a predicted rating of 4.9-5.0. We could include a variety of results that were predicted to have ratings of 4.5-5.0, in order to provide more dynamic results.

This also means quickly adjusting to new trends in the user's behavior. Even if the user wanted a lot of books on Hindu mythology last year, they might be interested in a completely different category now.

Appendices

Which people did what parts of the work? Were there problematic dynamics in your group?

Samrach: Facebook integration, rails web app. Wrote script which automatically imported grouplens data into our database.

Andrew: Web service master, implemented all aspects of recommendation engine.

Soheil: Tester, implemented tests to determine efficacy of recommendation algorithms.

JP: Rails web app implementer, user interface design. (I hope your eyes don't hurt.)

Everyone: Collaborated on UI prototyping, recommendation system, research.

What externally-written code (if any) was used in your project?

Facebookr - Ruby on Rails Facebook api wrapper.

Taste Mahout - Java library for generating recommendations.

Usage:

OS or browser requirements: requires a modern web browser. Note that Firefox currently has a bug where the scrollbar for the embedded facebook application is not present, and so it may be necessary to use Internet Explorer, Safari, or Google Chrome to get around this.

README file: Not necessary for browsing, but necessary for running the server. See below for developer-targeted Readme.

Website URL: http://apps.facebook.com/tastecliq

Developer README

Running the Rails Application

Official TasteCliq

The official tastecliq URL is http://apps.facebook.com/tastecliq. You will need a Facebook account to access the application.

Prerequisites for running Tastecliq on your own web server

1. Ruby and Ruby on Rails is installed.
2. Ruby has gems installed for the database engine of your choice (e.g. mysql). Some databases may have custom requirements.
3. You have changed **tastecliq/configs/database.yml** to match your database information.
4. You have checked out the source code for tastecliq from **svn+ssh://attu.cs.washington.edu/projects/instr/09au/cse454/c/tastecliq**
5. Depending on where your recommendations server is located, you will need to change the base URI in **tastecliq/app/models/xml_request.rb**

Running TasteCliq on your own web server

Once your database configuration information is ready (see #3 of previous section), you will need to run **rake db:migrate** from the **tastecliq/** directory.

To run the rails application, **cd** to the **tastecliq/** directory and run **ruby script/server**. You will then be able to access the application through http://localhost:3000 (the default ruby on rails localhost port), or if a domain name points to your web server, you can access it there.

Connecting Taste Mahout with Tastecliq's Recommender

To connect the Java Recommender, Taste Mahout must be installed first.

This can be obtained from their site: http://cwiki.apache.org/MAHOUT/buildingmahout.html

Alternatively, it can be SVN'd using:

svn co http://svn.apache.org/repos/asf/lucene/mahout/trunk

The instructions to build the recommender are also on the first link. Everything must be built, compiled, and installed before you can get the jar dependency files.

Also, Maven will be required for building/compiling/installing the project:

http://maven.apache.org/

(NOTE: These steps may take a long time).

After the initial steps have been completed, you can move the jar file **TasteCliqRecommender.jar** to **trunk/taste-web/lib**. This JAR file contains our recommender class and its dependencies.

Then follow the following steps:

1.) Copy and paste the custom servlet and its singleton files **RecommenderServlet.java** and **RecommenderSingleton.java** to the **/trunk/taste-web/src/main/java/org/apache/mahout/cf/taste/web** directory.
2.) Copy and paste **pom.xml** to **/trunk/taste-web/**
3.) If you are not there, go back to /trunk/taste-web and from there you can use the command: **mvn package** (sometimes mvn clean package helps).
4.) There are two ways to run the server.
    - You can go to the directory /trunk/taste-web/target and find the "mahout-taste-webapp-0.3-SNAPSHOT.war", renamed it, and deploy it was a WAR file. (NOTE: the number 0.3 changes with each iteration update).
    - In the directory /trunk/taste-web/ you can run the comman "mvn -Djetty.port=(your port number) jetty:run-war. (NOTE: If you do not include a port number it will default as 8080).