Recommendations Based on User Ratings on Raunk.com

Eric Conner
econner@stanford.edu

Zach Galant
zgalant@stanford.edu

Jeremy Keeshin
jkeeshin@stanford.edu

December 14, 2011

Background

Raunk.com is a website that lets you rate any item from 0-10. The items on the site have tags for organization. The goal of our project was to use machine learning to predict what items a user may like based on their ratings.

Naive Bayes To Predict Ratings

For each tag, we assign a probability that the user will rate an item with that tag in a certain rating bucket. When predicting an item, we look at all of the tags of the item and find the probability of rating each of its tags in a certain bucket, multiply them together, and take the bucket with the maximum probability.

Testing

We used LOOCV to test our predictions. We trained on all of a user's ratings except the one we were predicting, and we compared our predicted bucket to the actual bucket in which the rating fell. We repeated this for many items to compute our accuracy. Accuracy, however, was not the best metric for measuring the effectiveness of our algorithm. e.g. A prediction in bucket 8-9 incorrectly classifies a rating of 9.1.

To provide a better metric, we looked at the average error of the predictions. For every item, we took the average value in the predicted bucket and calculated the absolute value of the difference between the prediction and the actual rating.

Parameters

To tweak our algorithm, we changed the number of rating buckets. Having fewer buckets increased our accuracy because we had a wider range of guessing. Average error generally increased, since our predictions were less precise. We ran it using 4, 5, and 10 buckets and got average errors of 1.56, 1.42, and 1.48 respectively. Unfortunately, if you just guessed the average rating for the user each time, you get an average error of 1.25, so the algorithm doesn't actually help at all for general items.

Movies

We have a lot of data on movie ratings, so we focused on movies and actually made improvements with the predictions. We wrote a script to tag movies using data from IMDB. We were able to use 10 buckets and improve average error to 0.67 on movies from 0.71 before tagging with IMDB data. Both average errors yield better results than just predicting the average movie rating for that user each time, which gives an average error of 0.79.

Conclusion

The algorithm didn't improve our predictions for items overall, but it worked well with movies. It appears that the algorithm works with a lot of well tagged data, but with a smaller amount of sparsely tagged data, it doesn't help predictions.

Item Similarity Measure

One goal was to give recommendations to users who haven't rated a lot of items. To do this, we attempted to find pairs of items that are generally given close to the same rating for any arbitrary user. This can be visualized by plotting items X,Y in 2D space. Then plot users on this space as points (User's Rating of X, User's Rating of Y).

To find similar items, plot the 45 degree line from (0,0) to (10,10) and find pairs of items that deviate from this line the least. This gives us more information about how related to X an item is than just simply finding highly rated items from people who also rated X high.

If a pair of items X,Y have a small deviation from the trend line, we know that users who like X also generally like Y. We also know that users who dislike X also generally dislike Y, and users who were generally in the middle on X had similar thoughts about Y.

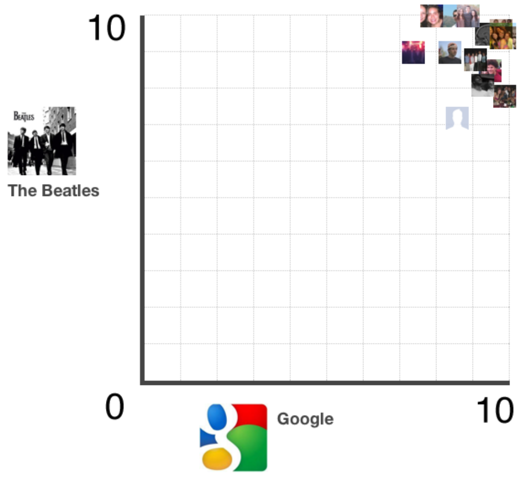

Figure 1: Everyone likes The Beatles and Google, but they are unrelated

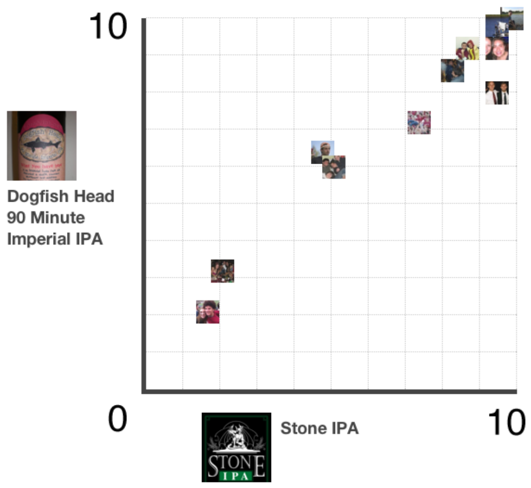

Figure 2: People generally like both beers at about the same level, so this is much more informative

If we only know one item a user likes, we can find pairs that involve that item that minimize deviation from the trend line. We focused our testing on musical artists and found some interesting recommendations. Here are a few examples:

Jimi Hendrix: Led Zeppelin, Marvin Gaye, The Beach Boys, Neil Young, Creedence Clearwater Revival, The Clash, Queen, The Rolling Stones, The Who, Bob Dylan.

Bon Jovi: The Doors, Guns N' Roses, Aerosmith, Creedence Clearwater Revival, AC/DC, Elvis Presley, Green Day, The White Stripes

Third Eye Blind: Dispatch, Coldplay, Dave Matthews Band, Blink-182, Lupe Fiasco, Katy Perry

Rick Ross: Big Boi, Ludacris, Kanye West, Jay-Z, 50 Cent, Dr. Dre, Tupac Shakur, Lil Wayne, Eminem, Lupe Fiasco

If we know, for example, that a user likes Third Eye Blind, we can recommend Dispatch, Coldplay, etc to that user.

Online Learning using the Perceptron

We also trained the perceptron using the online learning model to classify whether a user would rate a given item or not. A positive label corresponded to a user selecting a rating between 0 and 10, while a negative label was given for clicking skip. Our goal was to achieve good performance on random items so that we could ultimately build a recommender that would suggest to the user the next item he or she was most likely to rate given all of his or her previous ratings.

In the initial implementation, each example was a binary vector over the approximately 2700 tags in the system, where a 1 in position i indicated that the item was tagged with the ith tag. This ran too slowly, because for every item that the user rated the system would have to generate a vector over all tags. We needed these predictions to process quickly, as it should take no more than a second or two to load a page on the site.

The solution was instead to use maps to represent the tag vectors and parameter vector, since any tag not in the item's tags would contribute a 0 in the calculation of $\theta^T x$. We implemented this version of online learning and then tested it over a few manually inputted sets of about 100 ratings. We quickly found that the most time-consuming piece of testing the algorithm would be the manual data entry each time. To fix this we decided to implement an interface that could be used to gather anonymous ratings on Amazon's Mechanical Turk. We posted 100-rating HITs at 50 cents each.

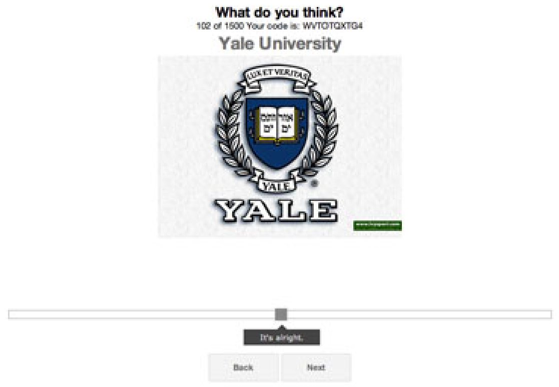

Figure 3: Interface Presented to Mechanical Turk Users

After the Turk user had rated 100 or more items, the system would generate a code that the user would then be required to input on the Mechanical Turk website in order to verify that the requested work had been completed. We found however that, while this method did generate lots and lots of data, almost all of it was junk. Users were incentivized to fake their ratings in order to get through as many ratings as possible. For the most part, users would rate ten to twenty random items and then skip the rest until they got to 100 or 200 ratings.

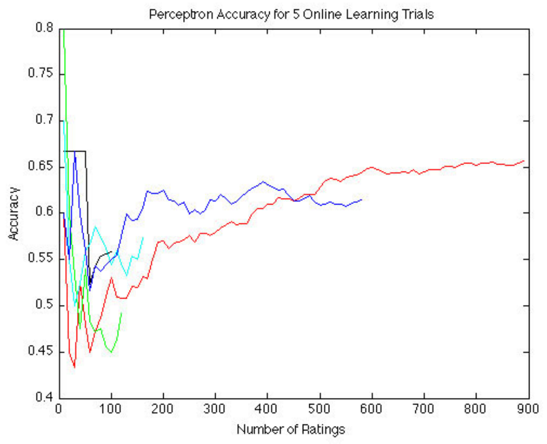

Figure 4: Five Trials Before Reducing the Feature Space

We decided to instead just ask volunteer users to test the system, but this would limit our capacity to try many different ideas since it was going to be difficult to get much data quickly in the online learning format. Figure 4 shows the accuracy of 5 trials run this way. We found that the classifier would level out at an accuracy of about 0.60 if given enough data. This performance was poor because we were finding that users, on average, were skipping about 60% of items so the classifier was doing no better than predicting with the probability of a skip.

The most obvious first step to improve classification was to reduce the feature space considered. The initial implementation consisted of a feature space of approximately 2700 dimensions (one for each tag in the system), far too many features for sets of examples of only a few hundred to a thousand ratings. We decided to reduce the feature space to 15 dimensions, one for each tag that was most likely to apply to a given item.

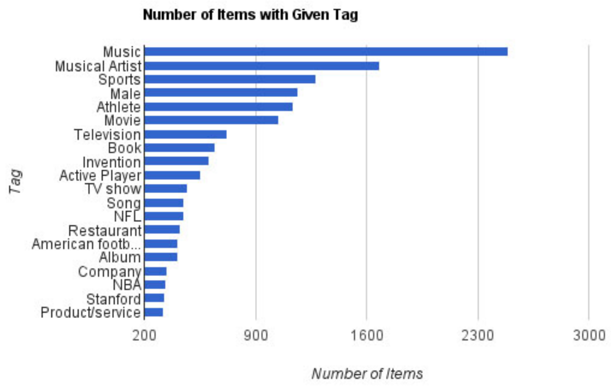

Figure 5: Number of items Tagged per Tag

Figure 5 shows the data considered to determine which tags to choose. We took the top 15 tags and used those as features.

Figure 6 shows the accuracy of a trial after reducing the feature space. The system performed almost exactly the same as it did before reducing the feature space. After investigating the data further, we found that our sets of items and tags were simply too sparse to make meaningful predictions based on tags alone. For example, movies, one of the most numerous items in the system, only came up about 5% of the time in a trial of 700 ratings and each of these movies did not have enough differentiating tags in order to give much information about each movie. The user, on average, rated only about half of these movies, which neutralized the presence of the isMovie feature. We suspected it was too general a feature to give any information about whether the user would actually rate the specified movie.

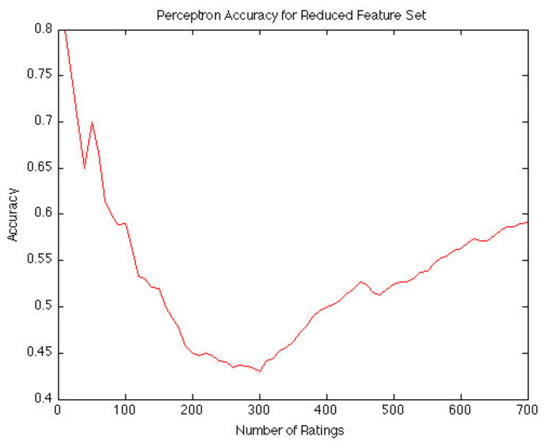

Figure 6: Classification Over a Reduced Set of 15 Features

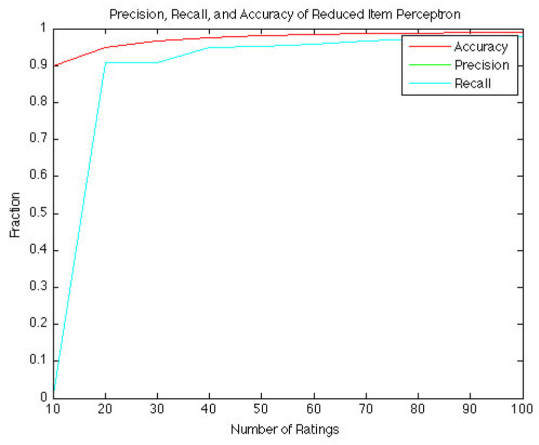

Figure 7: One vs. Many Classifier

To test this theory we decided to assume a fictitious user that skips every item that is not a movie and rates every item that is a movie. Thus, isMovie would be an incredibly informative tag about whether or not the user will rate the movie. We also began by biasing the prediction toward skip (i.e. if we had no knowledge of the item we assumed the user would skip it). In this case, as figure 7 shows, the perceptron was able to achieve above 90% accuracy using the general set of tags because it simply reinforced the movie tag and down weighted all the other tags. Thus, we surmised that the prediction could work if the set of items had enough overlapping tags to differentiate items beyond general categories such as isMovie, in which the user is only likely to rate about half the items anyway.

Recommendations with K-Means Clustering

We used k-means clustering to come up with recommendations for users within a given tag. To illustrate, we will explain the method on movies. Let the vector $V_u = [r_1, r_2, ...r_n]$ be a vector of a user's ratings on all n movies. Using k-means clustering, we group the users into k clusters. Then when trying to compute recommendations for a new user, first we compute their ratings vector. Then we assign them to the nearest cluster, and then based on all of the users in this cluster, recommend the items rated highest by that group.

The intuition is to find a group of people with similar tastes as you, and let them recommend you movies.

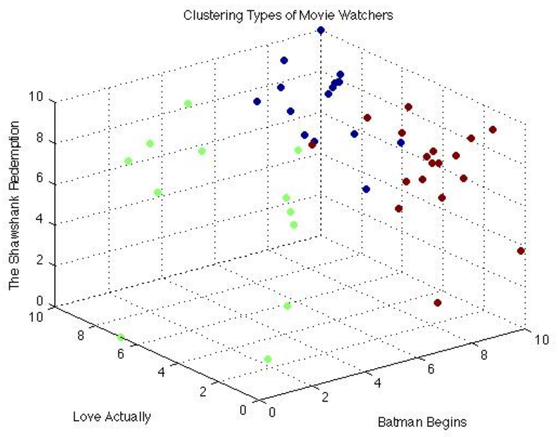

Figure 9: From this 3D plot, you can visualize the method. We are trying to place the users in n-dimensional "movie space" and cluster them based on similarity.

Using $k = 7$ here is one sample of movie clusters:

Cluster 1: The Godfather, Pulp Fiction, Schindler's List, The Silence of the Lambs, Taxi Driver, American History X

Cluster 2: The Lion King, 101 Dalmations, Love Actually, Forrest Gump, Aladdin, North by Northwest

Cluster 4: The Usual Suspects, Memento (film), Office Space, The Shawshank Redemption, Fight Club, Ferris Bueller's Day Off

Cluster 5: Toy Story, Toy Story 3, Toy Story 2 ,Batman: The Dark Knight ,Monsters, Inc, Ratatouille (film)

We had a wide variance in the number of users as-

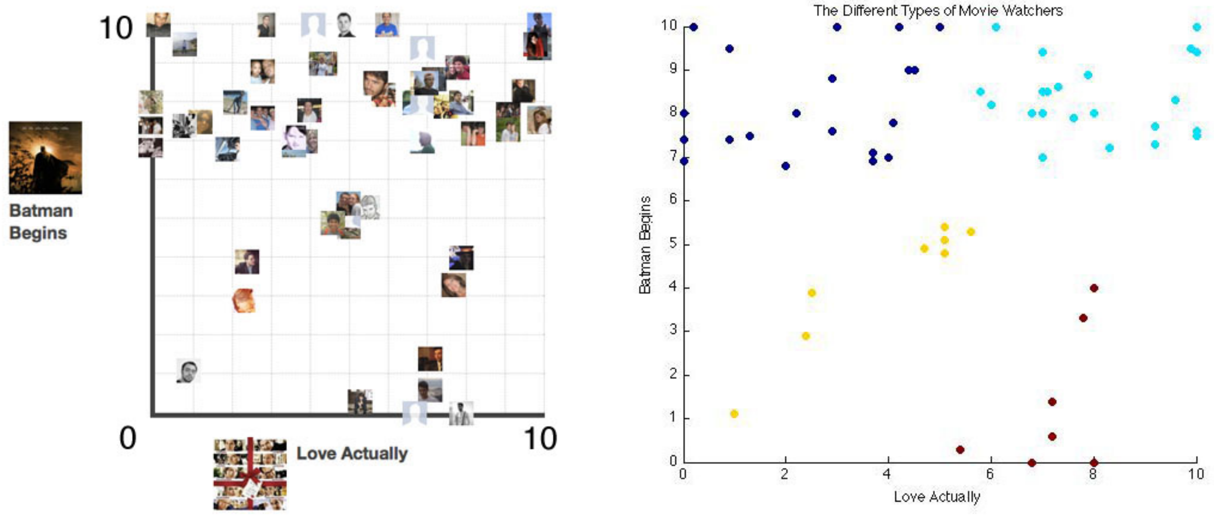

Figure 8: A simplified example shows users' ratings plotted in 2D movie space

signed to clusters. We only trained with users with > 10 ratings in the category, and in movies that ended up being 88 users. This would result in a wide range of users assigned to each bucket, and sometimes very skewed buckets. The main issue with our clustering was the sparsity of the data.

We came up with a performance metric to evaluate our algorithm chosen clusters using the Jaccard Index and testing with human users. We chose the tag beer, and gave participants a list of 71 beers. We then asked them to make five groups of beers that went together, with a maximum of 7 beers in each group. The grouping metric was vague, but in recommending items, individual taste groupings may be more complex than simply partitioning by type. We then measured the Jaccard Index, which is used the measure the similarity between datasets between the human created clusters and computer generated clusters.

$$J(A,B) = \frac{|A \cap B|}{|A \cup B|}$$

We ran our clustering algorithm on beers wtih $k = 5$ and 7 items per group, and for each user found the maximum Jaccard Index for any pair of groups.

User	1	2	3	4	5	6	Avg
Max(J)	0.57	0.4	0.38	0.5	0.5	0.3	**0.44**

We think this is quite good performance for an initial clustering recommendation system. If you compare this to the expected value for the Jaccard Index of choosing a random group of g items out of a set of T items, we find this is significantly better than random:

The number of successes, or the number in the intersection of two sets is distributed $x \sim HypGeom(T,g,g)$, and the Jaccard Index for two sets of length g with i in the intersection is $\frac{i}{2g-i}$.

$$E_x[J(A,B)] = \sum_i P_{x\sim h(T,g,g)}(x=i)J(A_i,B)$$

$$= \sum_{i=0}^{g} \frac{\binom{g}{i}\binom{T-g}{g-i}}{\binom{T}{g}} \left(\frac{i}{2g-i}\right)$$

For $T = 71$ and $g = 7$, $E_x[J(A,B)] = 0.0554$. This means you would expect a random cluster share no items with a human generated cluster, and so in achieving $\bar{J} = 0.44$, we were able to choose a reasonable cluster by human standards learned from only ratings.