Kyle Jackson and Robert Truong

CS 584

HW4

**HW4: Movie Recommender System**

Team name: econguy19

Rank & Public Score HW3-Text (as of 11/13/2016): 11, 0.84

*Approach and Methodology*

Our approach for this project was as follows: read the rating data into a CSR matrix, use a nearest neighbor item-based, user-based, and external content collaborative filtering (using the tag data) to determine ratings, and return the results.

The data were first read into a Pandas DataFrame. These data were in the following format for each observation:

*userIDi movieIDi ratingi*

where *i* represents the observation index. The data were then input into a CSR matrix from SciPy. At this step, the features represented unique movies and each observation represented a user. In addition, the movie tags were transformed to a CSR matrix with the tags as features for each movie.

We initially went with a user-based neighborhood model to predict user-movie rating combinations. We went with this method as opposed to an item-based approach due to a number of movies that had no ratings at all. We used Kyle’s HW1 implementation of the k-nearest neighbor algorithm and tweaked it slightly for the purposes of this project. Our first nearest neighbor implementation (defined as kNN in our src/main\_user-based file) went as follows:

1. For each *useri* and *moviei,f*, where *i* represents the row index of the user in the rating matrix and *f* represents the column index:
   1. Calculate the cosine similarity between *useri and user1…n*, where *n* represents the total number of users in the rating matrix
   2. Get the ratings of the *k* most similar users and average their rating for *moviei,f*, where *k* represents the number of neighbors specified by us
      1. If none of the *k* neighbors rated *moviei,f*, average all ratings of *useri*
      2. If *useri* has no previous ratings, average all ratings across *movief*
      3. If there are no ratings for *movieif*, assign a rating of 3.0
   3. Get the appropriate rating from the previous step and assign that as the rating prediction in the test file
2. Return the predictions as a Pandas DataFrame

This initial implementation took around 1.5 hours to run, and our RMSE plateaued around 0.88 on the leaderboards as we varied *k*. It was here that we then tried an approach that also incorporated item-based filtering along with the user-based filtering, as well as the tag data after that. The kNN implementation was modified to incorporate these changes, by having the for-loop in the code calculate ratings for all three methods. The item-based method basically worked exactly the same as the user-based, but utilized the transpose of the matrix instead of the original, and handling things based on movie ID in place of users. The tag-based method used a word-count matrix and would find other movies with similar tags, and return a rating consisting of the mean of each neighbor’s average rating. The loop would have each method return 0 if the output was invalid (like for instance, in the case where the item-based filtering resulted in getting closest neighbors that were also just empty rows), and would just take the average of the non-zero ratings.

The item-based filtering, user-based filtering, and the use of an external source each required separate matrices, so a high level look at our new algorithm is as follows:

1. Calculate the pairwise cosine similarities for the user, item, and movie tag matrices
2. For each *useri* and *moviei,f*, where *i* represents the row index of the user in the rating matrix and *f* represents the column index:
   1. Average the ratings across the *k* most similar users
   2. Average the ratings across the *k* most similar items (movies)
   3. Average the average ratings across the *k* most similar movie tags (external content)
   4. Take the mean of these three ratings and assign that as the predicted rating.
      1. If there are no ratings, assign rating of 3.5
3. Return all the predicted ratings as a Numpy array

This implementation performed better than the previous implementation, with the RMSE score plateauing at 0.84. It also ran much faster, at around 15 minutes, compared to the previous 1.5 hours. Some reasons include computing the cosine similarity for the matrices once as opposed to doing it for each iteration and using more efficient data structures, such as using dictionaries to enumerate the IDs instead of the IDs themselves. For instance, the user IDs attain a maximum at around 60,000, but using a dictionary meant we could index by the number of unique user IDs (i.e. it skips gaps), which reduced the dimensionality to around 2000. A similar tactic was done for movie IDs as well, because they suffered a similar problem.

We went with cosine similarity since it was already part of Kyle’s k-nearest neighbor implementation from HW1. If we were to continue this assignment, we might have tried a number of distance/similarity metrics, such as the Pearson correlation. For the cosine similarity, we used sciki-learn’s [cosine\_similarity](http://scikit-learn.org/stable/modules/generated/sklearn.metrics.pairwise.cosine_similarity.html) implementation, which computes the normalized dot product between two vectors X and Y:

K(X, Y) = <X, Y> / (||X||\*||Y||)

The second decision we had to make was the value of *k*, or the number of nearest neighbors to determine a rating across all three matrices (item, user, and movie tags). We tried a number of values and found that our RMSE plateaued when *k* 150. Our final submission used 150 for *k*. We also experimernted with incorporating SVD to reduce the dimensionality even further, but we found this was unnecessary because performing the cosine similarities outside the loop made the time it took for each iteration small enough already such that the speedup that SVD should have provided was not significantly better. SVD might have been justified for the purposes of de-noising, but it didn’t seem to help scores that much.

We attempted to do holdout validation on the training set, but the results we got from doing this were vastly different from running the algorithm on the test set. Our holdout results averaged around .67, but our test set results never gave us anything better than .84. We conjecture this might be because the training set has no cold start users, and the presence and handling of those significantly skewed the data.

*References*

In addition to the slides from class and documentation online for the various packages we used, Charu C. Aggarwal’s *Recommender Systems* proved to be an excellent reference.