Movie Rating Prediction

Donghyeok "Danny" Kim
SUID: 06476511

1. Motivation

With the onset of the big data age, we are flooded with a large ocean of information that we cannot sort through. Recommendation systems are the basis for navigating much of the data we have today from Amazon's "Frequently Bought Together" recommendations to the recommendations as seen in Netflix. In my course, CS 102, we were expected to use the MovieLens ratings data, collected by GroupLens Research, to predict how users will rate movies they haven't watched based on their past movie ratings.

1.1. What makes a "good" movie?

What makes a person rate a movie a 5 versus a 1 star? There could be many different reasons: a compelling plot, stunning visuals, how good looking actors are, etc. But it's not just about the movie, there are individual preferences and biases that can make the exact same movie a five star to someone and a one star to someone else. What we can generalize about the data is that it can be majorly be divided between these two categories: user information and movie content.

2. Diving into the Data

The Movielens data provided information about users and the movie content on the movie, its title, genre, and some metadata about the users and their ratings. One unique thing to note is that the movies and alldata files were tab separated rather than comma separated.

```
import pandas as pd
import numpy as np
import pandas_profiling
users = pd.read_csv('users.csv')
movies = pd.read_csv('movies.tsv', sep='\t')
ratings = pd.read_csv('ratings.csv')
alldata = pd.read_csv('allData.tsv', sep='\t')
predict = pd.read_csv('predict.csv')
```

By using the pandas-profiling package, I was able to do some quick exploratory data analysis. You can see the graphs that show in the EDA.html file.

```
netflix_data = alldata.profile_report (sort='None',
    html={'style':{'full_width': True}}, progress_bar=True)
netflix_data.to_file('EDA.html')
```

The data provided had 31,620 observations with 10 different variables. Only the 2nd and 3rd genre columns have missing cells. This means that there is essentially no data cleaning that is necessary to be done, except for preprocessing the genre columns to be joined together if necessary for a solution.

The mean rating is 3.57, meaning that the ratings are a right-skewed distribution. Much of the raters are male, from the ages of 20 to 30, and there are 2353 unique users. You can also see that there are 1453 distinct movie titles of which the most popular genres are comedy or drama. The year the movie was released skews towards the present, and some of the most popular movie titles with the most ratings are the Star Wars franchise with just over 200 ratings.

2.1. Data Preprocessing

The data was to be used in different models and therefore needed to be preprocessed in multiple ways. The three ways were:

The data was separated into a train and test split of 80/20.

```
from sklearn.model_selection import train_test_split
trainset, testset = train_test_split(alldata, test_size= 0.2)
```

The ratings were placed in a sparse matrix by using a pivot table. 0's were imputed into NaN values for the cosine similarity solution. The SVD solution left the NaN's as is.

```
ratings_pivot = ratings.pivot(index='userID', columns='movieID',
                              values='rating')
ratings_pivot.fillna(0, inplace=True)
```

Loaded into the Surprise reader (we'll get to that further below)

```
from surprise import Reader, Dataset
reader = Reader(rating_scale=(1,5))
data = Dataset.load_from_df(ratings[['userID', 'movieID', 'rating']],
                            reader)
```

2.2. Initial Baseline: Average Movies

I wanted a baseline to compare my more complicated solutions that I wanted to try. An easy solution to implement rating movies is to simply find the average of all the ratings for a movie and call it a day. This can be done with one line of code:

```
average = pd.DataFrame(ratings.groupby(['movieID']).mean()['rating'])
```

The flaw of this approach is that the fewer ratings a movie has, the less useful it is since if there is only 1 rating, then the "true" rating could easily be something vastly different. (i.e. one 5 star rating, but after 200 ratings it would be 2 star rating)

2.3. Weighted Average

One way to tackle the problem mentioned above is to use weighted averages so that movies with less ratings are counteracted by weighing the average for the entire dataset greater as opposed to those with many ratings would already have a distribution that isn't as biased.

```
#Rating = Weight * Individual Rating + (1- Weight) * Global Rating

weight = (ratings.groupby(['movieID']).count()['rating']/
          sum(ratings.groupby(['movieID']).count()['rating']))
ind_rating = ratings.groupby(['movieID']).mean()['rating']
global_rating = ratings.mean()['rating']
weighted_avg_rating = pd.DataFrame(weight*ind_rating +
                                  (1-weight) * global_rating)
```

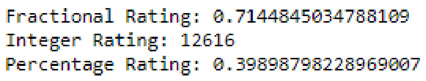
Unfortunately with the dataset that we were given, I could tell my method of weighing the ratings is not going to turn out well. As movies with the greatest rating were around 200 out of 30,000, thus biasing towards the global average too much. This essentially just predicts the overall average for every movie. I could adjust the weighing values to favor the individual ratings more than the global ratings, but I did not know what would be an appropriate adjustment.

2.4. Quick and Dirty Evaluation

To evaluate these two methods, I created an evaluation function according to the metrics described by Task A and Task B. Since these were average readings, I did not use a test set and instead evaluated from the entire dataset - which may cause a deviance from the other readings slightly. But the goal here is to get a rough estimate of where the Mean Absolute Error should be.

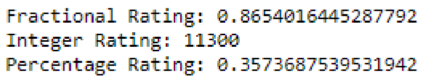
```
def Eval(test, actual):
    merged = test.merge(actual, how='inner', on ='movieID')
    distance = abs(merged['rating_x'] - merged['rating_y'])
    average_dist = sum(distance)/len(distance)
    number_correct = len(merged[merged.rating_x == merged.rating_y])
    percentage_correct = number_correct/len(merged)
    return (average_dist, number_correct, percentage_correct)
    simplified_ratings = ratings[['movieID', 'rating']]

#Average
average = pd.DataFrame(ratings.groupby(['movieID']).mean()['rating'])
average_dist, number_correct, percentage_correct = Eval(average.round(),
                                                  simplified_ratings)
print('Fractional Rating:', average_dist)
print('Integer Rating:', number_correct)
print('Percentage Rating:', percentage_correct)
```

This outputs:



```
Fractional Rating: 0.7144845034788109
Integer Rating: 12616
Percentage Rating: 0.39898798228969007
```

When we tried with the weighted average, as expected, it was clearly not as good:

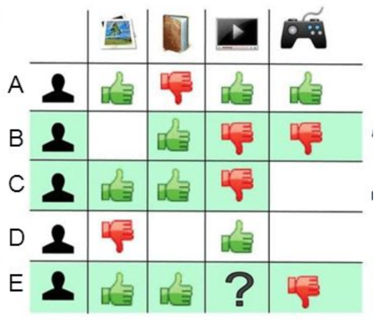


```
Fractional Rating: 0.8654016445287792
Integer Rating: 11300
Percentage Rating: 0.3573687539531942
```

But this tells us clearly that we're looking for a solution with a mean absolute error that is somewhere around 0.6 to 0.9!

3. Collaborative Filtering

Collaborative Filtering is a recommender system technique that uses information from a crowd of users to filter a recommendation for a specific user. We assume that if two people have similar opinions on one item, then it is more likely that they'll have the same opinion on a different item as well. (e.g.. if two people really like Star Wars and one of them likes Star Trek. The other person probably also likes Star Trek.) I use three collaborative filtering methods.



		🖼	📖	▶	🎮
A	👤	👍	👎	👍	👍
B	👤		👍	👎	👎
C	👤	👍	👍	👎	
D	👤	👎		👍	
E	👤	👍	👍	?	👎

I implemented the following solutions through NumPy and Pandas which will be posted on my Github. However, a simpler method is to use the Surprise library that is specifically made for recommender systems made the entire process of modeling and evaluation simplified into a few lines of code.

3.1. Memory User-Based (Cosine Similarity):

We can use cosine similarity or mean squared difference, or the pearson correlation coefficient as a metric of similarity. We are able to quantitatively measure how similar two movies are by placing the ratings of each movie as a vector and measuring the cosine of the angle between any two movies.

$$\text{cosine similarity}(u,v) = cos(\theta) = \frac{\langle u, v\rangle}{||u|| \cdot ||v||}$$

Once we have a similarity measure, we are able to rank order the similarity of the different movies, then we can take the average rating of the top k number of movies that are similar to our chosen movie.

$$\hat{r}_{ui} = \frac{\sum\limits_{v \in N_i^k(u)} \text{sim}(u,v) \cdot r_{vi}}{\sum\limits_{v \in N_i^k(u)} \text{sim}(u,v)}$$

As I was working on this solution, I realized this is exactly the idea behind the KNN Algorithm. The top number of k movies to average is the same as the k number of neighbors:

```
knn_algo = KNNBasic()
knn_algo.fit(trainset)
knn_test_pred = knn_algo.test(testset)
```

We can further increase our accuracy by taking baseline (our averages) estimates into account.

```
knnbase_algo = KNNBaseline()
knnbase_algo.fit(trainset)
knnbase_test_pred = knnbase_algo.test(testset)
```

3.2. Memory Item Based:

The same idea can be used for an item-based collaborative filtering. By gathering similar movies with the same content as another, we could determine the rating of the movie from its similar averages. However, given the distinct lack of information given about each movie (only given title and genre), there isn't enough to warrant implementing an algorithm. However, the idea would be to vectorize the text based on the count of the words or find the TF-IDF (Term Frequency - Inverse Document Frequency) and find the similarity to be rank-ordered and averaged.

It would be something like:

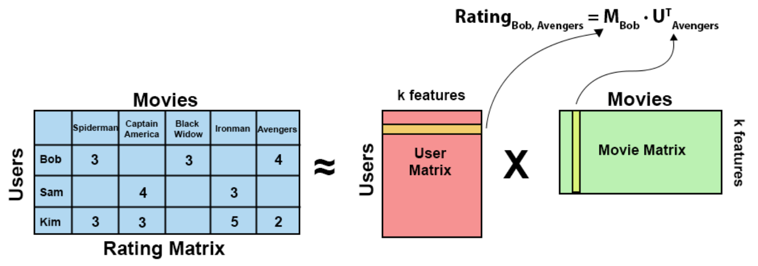
```
from sklearn.feature_extraction.text import CountVectorizer
From sklearn.metrics.pairwise import cosine_similarity
count_vector = CountVectorizer()
count = count_vector.fit_transform(genre_df)
similarity_scores = cosine_similarity(count)
```

Note: This is pseudo-code

3.3. Model Based ("Funk SVD"):

This method was popularized by Simon Funk) during the Netflix Prize competition for the best collaborative filtering algorithm. Instead of assuming that users are similar based on certain rules, we will instead assume that there are hidden rules that determine certain movies to be rated similarly.

When the original rating matrix is factored out, it gives two matrices. A User Matrix with each row as a user and a k number of columns for each feature a user has, and a Movie Matrix with each column as a movie and a k number of rows for each feature a movie has Once we compute M and U^T matrices that result from matrix factorization, we now have a low-rank approximation problem.



$\text{Rating}_{\text{Bob, Avengers}} = M_{\text{Bob}} \cdot U^T_{\text{Avengers}}$
Movies
Users
Rating Matrix
≈
k features
Users
User Matrix
X
Movies
Movie Matrix
k features

	Spiderman	Captain America	Black Widow	Ironman	Avengers
Bob	3		3		4
Sam		4		3	
Kim	3	3		5	2

$$R = M\Sigma U^T$$

Although it's commonly referred to as Funk SVD and the concept is similar, what we're doing is not exactly SVD. SVD requires a complete matrix, and what we have is a sparse matrix.

Since what we have is a sparse matrix, we cannot compute SVD on it. This means there are no associated RR^T and R^TR matrices, no eigenvalues, and no diagonal matrix Σ. Which means you cannot compute formal SVD. But doesn't this mean that we'll be missing some ratings in the end result if there is not enough information? *Yes.* According to Simon Funk, what you should do when some ratings are missing from the matrix is to simply ignore it and find all of the vectors p_u (rows of User matrix) and q_i (columns of Movie matrix). We can find those vectors by solving an optimization problem:

$$\min_{p_u, q_i} \sum_{r_{ui} \in R} (r_{ui} - p_u \cdot q_i)^2.$$

To solve this optimization problem we can use machine learning to approximate what the missing values are. In other words, we can use SGD (stochastic gradient descent) or ALS (Alternating Least Squares) to minimally iterate through while using RMSE (Root Mean Squared Error) or MAE (Mean Absolute Error) as the cost function.

```
#SVD - Singular Value Decomposition
svd = SVD()
svd_algo.fit(trainset)
svd_pred = svd_algo.test(testset)
```

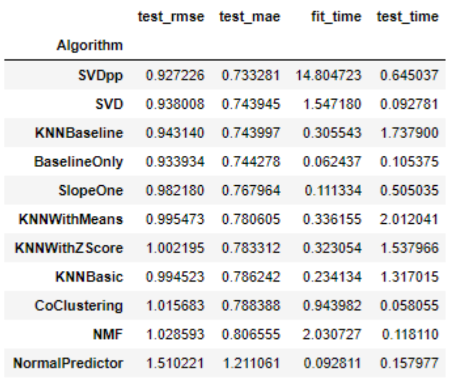
4. Evaluation

The conventional way for evaluating models is to use RMSE for continuous target variables, but given the problem by Task A, I chose to optimize for MAE. In general, RMSE is used when emphasizing errors with greater magnitude is more important than emphasizing smaller errors at a greater frequency (one value off by 10 > two values off by 5). MAE is more appropriate at catching errors "vanilla" - it does not place greater importance on magnitude (one value off by 10 = two values off by 5). Considering this, RMSE is most likely a better evaluation metric to use for real-world applications. You probably don't want a user to be recommended something wildly different than they would want.

Although I have an idea of what I believe would be good solutions to the problem, I wanted to see if there is an algorithm I did not consider that performs very well. I ran through all possible algorithms available in the Surprise package:

```
benchmark = []
for algorithm in [SVD(), SVDpp(), SlopeOne(), NMF(),
                  NormalPredictor(), KNNBaseline(), KNNBasic(),
                  KNNWithMeans(), KNNWithZScore(), BaselineOnly(),
                  CoClustering()]:
  #Perform cross validation
  results = cross_validate(algorithm, data, measures=['RMSE', 'MAE'],
                           cv=3, verbose=False)

  #Get results & append algorithm name
  tmp = pd.DataFrame.from_dict(results).mean(axis=0)
  tmp = tmp.append(pd.Series([str(algorithm).split(' ')[0].split('.')[-1]],
                                                     index=['Algorithm']))
  benchmark.append(tmp)
pd.DataFrame(benchmark).set_index('Algorithm').sort_values('test_mae')
```

Output:



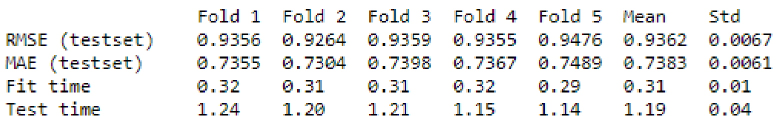
Algorithm	test_rmse	test_mae	fit_time	test_time
SVDpp	0.927226	0.733281	14.804723	0.645037
SVD	0.938008	0.743945	1.547180	0.092781
KNNBaseline	0.943140	0.743997	0.305543	1.737900
BaselineOnly	0.933934	0.744278	0.062437	0.105375
SlopeOne	0.982180	0.767964	0.111334	0.505035
KNNWithMeans	0.995473	0.780605	0.336155	2.012041
KNNWithZScore	1.002195	0.783312	0.323054	1.537966
KNNBasic	0.994523	0.786242	0.234134	1.317015
CoClustering	1.015683	0.788388	0.943982	0.058055
NMF	1.028593	0.806555	2.030727	0.118110
NormalPredictor	1.510221	1.211061	0.092811	0.157977

Here we see that SVDpp places first, SVD places second, KNN third, Baseline ratings (Averages) and then SlopeOne. We can disregard SVDpp because it takes into account implicit rating factors (e.g. how many times did a user watch a movie, did they watch the entire movie) which are not present in our dataset. I will assume that the algorithm is overfitting and is not a true representation of accuracy. SlopeOne is a linear regression solution for a sparse matrix, which I did not know about. I'd like to explore SlopeOne in the future as a continuation of this project. Other than these two, my approaches seem to perform well.

With parameter tuning I used grid search to iterate through different parameters along with a k-fold cross-validation (CV). A 3-fold CV is used for the grid search to lower the run time. Once a parameter setting is found a 5-fold cross validation is used to give a more accurate estimate:

```
#Memory-based Collaborative Filtering (User) Evaluation
param_grid = {'k': [10, 20, 30, 40, 50, 60, 70],
                        'sim_options': {'name': ['msd', 'cosine'],
                        'min_support': [1, 3, 5],
                        'user_based': [True]},
                        'bsl_options': {'reg_i': [5, 10, 15, 20],
                        'reg_u': [10, 15, 20, 25],
                        'n_epochs': [10, 15]}
                        }
gs = GridSearchCV(KNNBaseline, param_grid, measures=['rmse','mae'],
                  cv=3, refit=True)
gs.fit(data)
knn_algo = gs.best_estimator['mae']
cross_validate(knn_algo, data, measures=['rmse', 'mae'],
               cv=5, verbose=True)
```

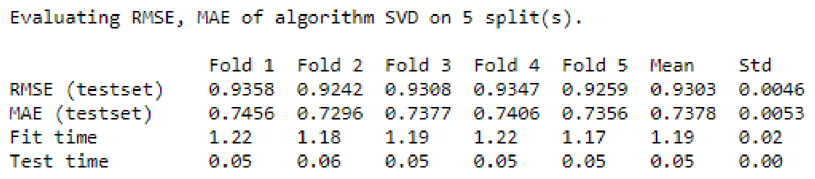
Output:



```
                  Fold 1  Fold 2  Fold 3  Fold 4  Fold 5  Mean    Std
RMSE (testset)    0.9356  0.9264  0.9359  0.9355  0.9476  0.9362  0.0067
MAE (testset)     0.7355  0.7304  0.7398  0.7367  0.7489  0.7383  0.0061
Fit time          0.32    0.31    0.31    0.32    0.29    0.31    0.01
Test time         1.24    1.20    1.21    1.15    1.14    1.19    0.04
```

```
#SVD Evaluation
SVD_params = {'n_factors': [80, 90, 100, 110, 120, 140],
              'n_epochs': [10, 15, 20, 25, 30],
                      'lr_all': [0.001, 0.003, 0.005, 0.008],
              'reg_all': [0.01, 0.02, 0.04, 0.8]}
gs_svd = GridSearchCV(SVD, SVD_params, measures=['rmse','mae'],
                      cv=3, refit=True)
gs_svd.fit(data)
svd_algo = gs_svd.best_estimator['mae']
cross_validate(algo, data, measures=['rmse', 'mae'], cv = 5,
               verbose =True)
```

Output:



```
Evaluating RMSE, MAE of algorithm SVD on 5 split(s).

                  Fold 1  Fold 2  Fold 3  Fold 4  Fold 5  Mean    Std
RMSE (testset)    0.9358  0.9242  0.9308  0.9347  0.9259  0.9303  0.0046
MAE (testset)     0.7456  0.7296  0.7377  0.7406  0.7356  0.7378  0.0053
Fit time          1.22    1.18    1.19    1.22    1.17    1.19    0.02
Test time         0.05    0.06    0.05    0.05    0.05    0.05    0.00
```

Finally, we use our models to predict what the values would be:

```
KNN_preds = []
for i in range(len(predict)):
_, _, _, rating_pred, _ = gs.predict(predict.userID[i], predict.movieID[i])
KNN_preds.append(rating_pred)
predict['KNN_preds'] = KNN_preds
KNN_df = predict.drop(['rating'], axis=1)
KNN_df.rename(columns={'KNN_preds':'rating'}, inplace=True)
KNN_df.to_csv('KNN_rating_v1.csv')
KNN_df.round().to_csv('KNN_rating_v2.csv')
```

5. Results

The predictions when submitted to the leaderboard resulted as such:

Algorithm	**Fractional Rating (Avg. Distance)**	**Integer Rating (Num. Correct of 200)**
SVD	0.6926	91
KNNBaseline	0.7017	87
(Prof Widom's Solution)	0.7036	86
KNNBasic	0.7709	77
Average	0.8127	71
Weighted Average2	0.8357	67
Weighted Average	0.9725	83

The SVD solution resulted in the best prediction, however KNNBaseline also beat the professor's solution as well.

6. Description of Files Used

Movie_rating.ipynb - Jupyter Notebook file that contains all python scripts for cleaning, preprocessing, modeling, and evaluation

EDA.html - Exploratory data analysis

V1predict - SVD model predictions for Task A

V2predict - SVD model predictions rounded for Task B

V1predict2 - KNN + Baseline model predictions for Task A

V2predict2 - KNN + Baseline model predictions rounded for Task B

V1predict3 - Baseline/Simple Rounding predictions for Task A

V2predict3 - Baseline/Simple Rounding predictions for Task B