DSGA-1004 Big Data:
Movie Recommendation System Project

Bhavna Thakar and Seon Hye Yang (Group 111)

May 17, 2022

1 Abstract

The purpose of this paper is to develop and analyze recommendation systems using the MovieLens data provided by GroupLens. To do this, two models were developed: a baseline popularity model and an alternating least squares model (ALS). The popularity model recommended the most popular movies to users while the ALS model used Apache Spark's ASL model to provide tailored recommendations using matrix factorization and latent factor analysis. The efficacy of the popularity model was 84.412 for the small dataset and 85.102 for the large dataset. The ALS model was evaluated using RMSE calculations using a grid search analysis with different ranks and regularization parameters. The lowest RMSE evaluated was 0.803 which was done with a rank of 20 and regularization parameter λ of 0.803. In addition to RMSE, evaluation was done using average error. To further explore the data, an extension was done to explore error analysis of the model where the error per genre was determined. The Github link for this project can be found here: `https://github.com/nyu-big-data/final-project-group_111`

2 Introduction

Recommendation systems are used in a myriad of ways, from song suggestions on Spotify to product reviews on Amazon. One way that recommendation systems can be particularly useful is with movie data. If one person loves *Under the Tuscan Sun*, how likely would it be that they also love *Die Hard*? And if the likelihood is low, what movie should they watch instead? This paper aims to develop a system by which users are given personalized recommendations based on movies that they enjoyed using ratings as a means of prediction.

In order to implement this system, two models had to be created: a baseline model and a more complicated model. Both a popularity-based recommendation system and an alternating least squares recommendation system were constructed to tackle this task. While the popularity-based model recommends the most popular movies, it is not personalized. Fans of *Under the Tuscan Sun* and *Die Hard* alike will receive the same list of movies recommended. Alternating least squares, on the other hand, creates recommendations based on the way people rate movies. It does this using matrix factorization–creating item-item, user-user, and user-item matrices.

3 Data

The data used for this project comes from the MovieLens data collected by GroupLens, a research lab in the Department of Computer Science and Engineering at the University of Minnesota. There are two sets of data: `ml-latest-small` which consists of 100836 ratings and 3683 tag applications across 9742 movies and `ml-latest` which consists of 27753444 ratings and 1108997 tag applications across 58098 movies. Each set has the following `links.csv, movies.csv, ratings.csv, tags.csv`.

In order to use the data effectively, it needs to be partitioned into train, validation, and test sets. For every user, at least a portion of the user's ratings needs to appear in the training set so that the user's embedding wouldn't be purely random when evaluated in the test set. To do this, the package `StratifiedKFold` from `sklearn.model_selection` was used. First, the dataset was split into train and test. Then, the resuling test dataset was split into test and validation. This is because the first one splits it in half–50% test and 50% train. The resulting 50% of the data is split in two in order to make tests and validation. Therefore, we had 50% training, 25% testing, and 25% validation.

The below code and output shows how the data was partitioned:

```
import numpy as np
import pandas as pd
from sklearn.model_selection import StratifiedKFold
small_p = pd.read_parquet('small.parquet')
large_p = pd.read_parquet('large.parquet')
skf = StratifiedKFold(n_splits=2)
ids = small_p["userId"]
for train, rest in skf.split(small_p, small_p["userId"]):
      trains = small_p.loc[train]
      rests = small_p.loc[rest]
for test, val in skf.split(rests, rests["userId"]):
      tests = rests.iloc[test]
      vals = rests.iloc[val]
len(small_p) == len(train)+len(vals)+len(tests)
      True
```

`trains`

	userId	movieId	rating	timestamp
0	1	1	4.0	964982703
1	1	3	4.0	964981247
2	1	6	4.0	964982224
3	1	47	5.0	964983815
4	1	50	5.0	964982931
...	...	...	...	...
100180	610	44191	3.5	1493844860
100181	610	44195	3.5	1493845111
100182	610	44199	4.0	1493847397
100183	610	44397	3.5	1493848181
100184	610	44555	3.0	1493845173

50418 rows × 4 columns

`tests`

	userId	movieId	rating	timestamp
116	1	1967	4.0	964981710
117	1	2000	4.0	964982211
118	1	2005	5.0	964981710
119	1	2012	4.0	964984176
120	1	2018	5.0	964980523
...	...	...	...	...
100505	610	86835	2.5	1493848027
100506	610	86880	3.0	1493846112
100507	610	86898	5.0	1493847938
100508	610	87222	3.5	1479542292
100509	610	87232	4.0	1493845469

25209 rows × 4 columns

`vals`

	userId	movieId	rating	timestamp
174	1	2644	4.0	964983393
175	1	2648	4.0	964983414
176	1	2654	5.0	964983393
177	1	2657	3.0	964983426
178	1	2692	5.0	964981855
...	...	...	...	...
100831	610	166534	4.0	1493848402
100832	610	168248	5.0	1493850091
100833	610	168250	5.0	1494273047
100834	610	168252	5.0	1493846352
100835	610	170875	3.0	1493846415

25209 rows × 4 columns

4 Methods

Two recommendation systems were built and implemented: a baseline popularity recommendation model and an ALS model using Spark's ALS model.

4.1 Popularity Model

For this model, we convert csv files to parquet files because parquet files ran faster than csv files, and parquet files save time when running on the cluster. For the small dataset, efficiently written code was enough to run on the cluster. For the large dataset, we first applied Dask Dataframe and did npartition=100, 1000, 2000. However, this was not working and took a very long time to run on the cluster. Eventually, we decided to partition the large data files themselves. We partitioned the large data into 100 files and ran them on the cluster. With the small dataset, we were able to push the training, testing, and validation sets for the small data to github. Although we were able to successfully implement training, testing, and validation sets for the large data, we were unable to push them to github because they were simply too large. Instead we transferred all our data files to the cluster.

The popularity model was designed by calculating the ratings rated by users and using the mean. As discussed in the lecture, we implemented a weighted bias or bias model where we made a weighted bias for the ratings. We use the formula $w_r = (v/(v+m)) * r + (m/(v+m)) * c$ where $r =$ `row["mean"]`, $v =$ `row["count"]`, $m =$ `totalcount/50000`, and $c =$ `np.mean(ratings["mean"])`. And we did on the training dataset.

In order to measure the performance or accuracy against the validation data, we calculated `accuracy = 1 - errors`. First we got the top 100 movies from the training set and the top 100 movies from the validation set, and we calculated the difference between training and validation set. We defined out errors as `err = np.absolute((ValidationRating - WeightedRating) / WeightedRating)`. We then append the errors and did `(1 - np.mean(errs)) * 100)`. We did this for small and large files.

Then, we performed this against the testing data. What we did was, we adjusted the weighted bias a little bit in order to get a better accuracy and lower RMSE and MSE results. For example, in our validation data, we applied the formula $w_r = (v/(v+m)) * r + (m/(v+m)) * c$ where $r =$ `row["mean"]`, $v =$ `row["count"]`, $m =$ `totalcount / 50000`, and $c =$ `np.mean(ratings["mean"])` on our training data. Then for the testing data, we applied the formula $w_r = (v/(v+m)) * r + (m/(v+m)) * c$ where $r =$ `row["mean"]`, $v =$ `row["count"]`, $m =$ `totalcount / 70000`, and $c =$ `np.mean(ratings["mean"])` on our training data. We did this for small and large files.

As we can see from the results, our model improved when tested with test dataset.

For our evaluation criteria, we performed accuracy, RMSE and MSE using the formula MSE =`np.square(np.subtract(y-a`
`y-predicted)).mean()` and RMSE = `math.sqrt(MSE)`.

	RMSE	MSE	Accuracy
Validation	0.6894398838310366	0.4753273534169533	83.24173763637899
Test	0.6101358927897185	0.37226580767030687	84.41217569397914

Table 1: Table showing evaluation results for the small dataset

	RMSE	MSE	Accuracy
Validation	0.7098110075600017	0.5038316664533448	79.38270832545653
Test	0.6540944307194949	0.4278395242982601	85.10165739960313

Table 2: Table showing evaluation results for the large dataset

4.2 ALS Model

The ALS model uses matrix factorization to make representation of the user-item matrix, with users being playlists in this project. The ALS algorithm uses latent factors to predict the expected preference of a given user and provide an adequate recommendation. The mathematical formula for latent factorization is given as follows:

$$min_{x^*,y^*} \sum_{u,i} (p_{ui} - x_u^T y_i)^2 + \lambda(||x_u||_2^2 + ||y_i||_2^2)$$

The two parameters that need to be optimized in this ALS model are rank, which is the dimension of the latent factors, and the regularization parameter.

When implementing the model in Python, parquet files of the movies and the ratings from the training data were passed through Spark. After joining these via SQL, the ALS model was fitted to this data. The method `coldStartStrategy="drop"` was used so that any users that contained NaN values were dropped and not used. This allowed for a more simple and streamlined implementation of the ALS algorithm. Each model was stored using a naming convention that easily identified rank and regularization parameter λ. The purpose of tuning for rank and regularization parameter is to account for bias and variation. The parameter λ reduces the variance while the rank reduces the bias. The best combination of the two results in a model that is accurate but does not overfit the data.

5 Evaluation

The primary evaluation metric used to determine the efficacy of the recommendation models was root mean square error (RMSE). The mathematical equation behind RMSE is as follows, where y is the actual rating and $\hat{y}$ is the rating predicted by the model:

$$\sqrt{\sum_{i=1}^{D} (y - \hat{y})^2}$$

In order to optimize this, a grid search was done by testing different ranks in the range of $[10, 15, 20]$ and different regularization parameters in the range of $[0.01, 0.1, 1]$. The RMSE was calculated for each of the nine ALS models and the model with the lowest RMSE was deemed to be the most accurate. The following table shows the RMSE for each of the models used in hyperparameter tuning.

		Lamba		
R		0.01	0.1	1
a	10	1.054	0.826	1.333
n	15	1.090	0.821	1.329
k	20	1.088	**0.803**	1.331

Table 3: Grid search of RMSE where different regularization parameters λ and ranks are tested to determine lowest RMSE

As evidenced by the above table, the λ value of 0.1 and the rank of 20 result in the ALS model with the lowest RMSE, and thus results in the optimal model. Generally, a higher regularization parameter is favorable when the rank is also high as a high rank can be indicative of overfitting.

In addition to RMSE, the second choice of evaluation criteria was average error which was computed using the absolute error metric. This was done by taking the absolute value of the difference between real ratings

and predicted ratings. Using this evaluation metric, the accuracy of the ALS model on the small dataset was determined to be 0.849 and the accuracy of the ALS model on the large dataset was determined to be 0.862.

6 Extension

The extension done was the qualitative error analysis of errors by genre. For this implementation the goal was to determine whether there were certain genres of movies that were prone to high errors. This error analysis was done using the dictionary of movies created in the evaluation step so that the average error could be computed per genre. The qualitative error model outputs an average error for each individual rating grouped by genre so that the overall accuracy of the predicted ratings can be evaluated. This is done by iterating through the genres and taking the average error per rating. Because there are some movies without listed genres, the below query was used to avoid them and create a dataframe wite genres to be parsed:

```
genreMovie = spark.sql('SELECT movieId, genres FROM movies WHERE genres != "(no genres listed)"').toP
```

The code used to actually calculate the error per genre was as follows:

```
for genre in genreErr.keys():
      t = sum(genreErr[genre])/len(genreErr[genre])
      genreErr[genre] = t
```

What was determined was that genres that are widely used to label movies–namely comedy and drama–had significantly higher errors that niche genres like Western. The accuracy for these high-error genres scored below 0.8.

7 Contribution

Seonhye Yang - Worked on popularity model and partitioning the datasets into training, testing, and validation. Also worked on the final report.

Bhavna Thakar - Worked on ALS model, evaluation, and extension. Also worked on the final report.