Movie Recommendation System

You pick the movie, I'll choose the restaurant...

1. Project Overview

Choosing movies can be a stressful, high-stakes endeavor for anyone. And while users get stuck in indecision, the social media and streaming platfrom lose engagement and eventually profit. Fortunately, this program is here to help. It provides movie recommendations for any new user, based on their movie preferences. The program asks the user to rate five random movies from the MovieLens (https://grouplens.org/datasets/movielens/) database of almost 10,000 movies. Based on the ratings, the program utilizes a user-based collaborative filtering model from surprise (https://surpriselib.com/) to provide five recommendations.

2. Business Case

With the vast entertainment options available, low engagement and user churn in any social media or streaming service can hurt profit. Considering that 20% of adults are indecisive and 67% of relationship agreements never get resolved, picking a movie can be a daunting task, fueling decision paralysis and disengagement. Luckily, machine learning can relieve indecision by providing recommendations for any user, based on their preferences.

3. Data Understanding

3.1 Approach

This recommendation system gets a user to rate five random movies from an existing database, and then returns five recommendations. The program uses a collaborative filtering model - no content-based or hybrid filtering was performed. Specifically, it relies on the model's ability to predict how any user would rate any movie. The `surprise` module is well suited for explicit ratings system with collaborative filter.

Because no hybrid or content filtering is used, we're only interested in utilizing the data files containing our users, the movies, and the ratings. Other information won't be needed so we'll keep that in mind as we inspect the data.

3.2 Source Data

Dataset Background

So, we have our data spanning over 4 separate csv files. We also have a README file which may tell us how this data interacts. Let's open that file to gain some insight.

In [2]:
```
file_path = 'data/README.txt'

with open(file_path) as file:
    print(file.read())
```

```
Summary
=======

This dataset (ml-latest-small) describes 5-star rating and free-text taggin
g activity from [MovieLens](http://movielens.org), a movie recommendation s
ervice. It contains 100836 ratings and 3683 tag applications across 9742 mo
vies. These data were created by 610 users between March 29, 1996 and Septe
mber 24, 2018. This dataset was generated on September 26, 2018.

Users were selected at random for inclusion. All selected users had rated a
t least 20 movies. No demographic information is included. Each user is rep
resented by an id, and no other information is provided.

The data are contained in the files `links.csv`, `movies.csv`, `ratings.csv
` and `tags.csv`. More details about the contents and use of all these file
s follows.

This is a *development* dataset. As such, it may change over time and is no
t an appropriate dataset for shared research results. See available *benchm
```

Summary

This dataset (ml-latest-small) describes 5-star rating and free-text tagging activity from MovieLens (http://movielens.org), a movie recommendation service. It contains 100836 ratings and 3683 tag applications across 9742 movies. These data were created by 610 users between March 29, 1996 and September 24, 2018. This dataset was generated on September 26, 2018.

Users were selected at random for inclusion. All selected users had rated at least 20 movies. No demographic information is included. Each user is represented by an id, and no other information is provided.

The data are contained in the files `links.csv`, `movies.csv`, `ratings.csv` and `tags.csv`. More details about the contents and use of all these files follows.

3.3 Data Files

Ratings Data File Structure (ratings.csv)

All ratings are contained in the file `ratings.csv`. Each line of this file after the header row represents one rating of one movie by one user, and has the following format:

```
userId,movieId,rating,timestamp
```

Tags Data File Structure (tags.csv)

All tags are contained in the file `tags.csv`. Each line of this file after the header row represents one tag applied to one movie by one user, and has the following format:

```
userId,movieId,tag,timestamp
```

Movies Data File Structure (movies.csv)

Movie information is contained in the file `movies.csv`. Each line of this file after the header row represents one movie, and has the following format:

```
movieId,title,genres
```

Links Data File Structure (links.csv)

Identifiers that can be used to link to other sources of movie data are contained in the file `links.csv`. Each line of this file after the header row represents one movie, and has the following format:

```
movieId,imdbId,tmdbId
```

Summary of Files

After reviewing the files, it appears that we only care about the `ratings.csv` file. It contains, per our description, "one rating of one movie by one user." This is precisely the data we care about, so we can put aside the other files to pursue our collaborative filtering. So when we go into inspection and data preparation, we will keep this in mind

3.4 Data Inspection

Let's go ahead and see if we can verfiy some of this data in the `ratings.csv` file. I'm going to go ahead and import this file into PANDAS one-by-one to make sure the data matches the description.

In [3]:
```
import pandas as pd
import numpy as np
import random
```

Ratings File Summary

In [4]:
```
ratings_df = pd.read_csv('data/ratings.csv')
ratings_df.head(5)
```

Out[4]:

	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

In [5]:
```
ratings_df.describe()
```

Out[5]:

	userId	movieId	rating	timestamp
count	100836.000000	100836.000000	100836.000000	1.008360e+05
mean	326.127564	19435.295718	3.501557	1.205946e+09
std	182.618491	35530.987199	1.042529	2.162610e+08
min	1.000000	1.000000	0.500000	8.281246e+08
25%	177.000000	1199.000000	3.000000	1.019124e+09
50%	325.000000	2991.000000	3.500000	1.186087e+09
75%	477.000000	8122.000000	4.000000	1.435994e+09
max	610.000000	193609.000000	5.000000	1.537799e+09

It's interesting that the rating in 25-75 percentile range are from 3.0-4.0, meaning user generally rate movies favorably.

In [6]:
```
ratings_df.isna().sum()
```

Out[6]:
```
userId       0
movieId      0
rating       0
timestamp    0
dtype: int64
```

In [7]:
```
unique_movies = list(ratings_df['movieId'].unique())
print('Number of movies: ', len(unique_movies), '\n')

unique_users = list(ratings_df['userId'].unique())
print('Number of ratings: ', len(unique_users))

```

```
Number of movies:  9724

Number of ratings:  610
```

So, we have confirmed no null values, as well as 10,0836 movie ratings and a maximum userID of 610. All of our ratings our .5 - 5.0 and... we have 9,724 movies. This looks promising so far and matches our README.

As we mentioned before, we're only concerne with the ratings file so we'll move on to data prep.

4. Data Preparation

4.1 Data Approach

So we have a lot of data present in these files. For this project, we will only need to use the ratings file, and apply a matrix factorization to it.

For the ratings file, we will drop the timestamp, as we're not as interested in the time series data.

Dropping timestamp.

I will drop the timestamp from each of the `ratings_df` and `tags_df` .

In [8]:
```
ratings = ratings_df.drop('timestamp', axis = 1)
```

5. Modeling & Evaluation

As was noted in the previous section, we can create our model simply from the `ratings.csv` file. To create our baseline model, we're going to use the `surprise` module. We will compare SVD and a variety of KNN based methods within the `surprise` module to determine which is the most accurate for our dataset. For consistency sake, will use RSME (Root Square Mean Error).

We will also establish a baseline of Random prediction ratings to see how the RSME compares.

5.1 Random and Median for Baseline

To see how well our model does, we will compare over a "random" model that predict a user's ratings. THis model will just pick ratings at random between (.5 and 5.0). We will also compare what the RMSE would be if we compared it to the median.

In [9]:
```
#let's create a column to test our first random column, which is a random r
rand_rate = ratings
rand_rate['predicted'] = np.random.randint(1,10, rand_rate.shape[0])/2

rand_rate
```

Out[9]:

	userId	movieId	rating	predicted
0	1	1	4.0	4.0
1	1	3	4.0	2.0
2	1	6	4.0	4.5
3	1	47	5.0	2.5
4	1	50	5.0	3.0
...	...	...	...	...
100831	610	166534	4.0	4.0
100832	610	168248	5.0	0.5
100833	610	168250	5.0	2.0
100834	610	168252	5.0	4.5
100835	610	170875	3.0	3.0

100836 rows × 4 columns

We successfully created a new column called 'predicted' for all of the movies. Now, let's see if we can

In [10]:
```
rmse = np.sqrt(((rand_rate['predicted'] - rand_rate['rating']) ** 2).mean(
rmse
```

Out[10]:
```
1.9375348294776356
```

Our RMSE for this random baseline is 1.94. I hope we can beat that in our `surprise` modules

Now, let's do the same test utilizing the median rating. According to our inspection above, the 50% score was 3.5

In [11]:
```
med_rate = ratings
med_rate['predicted'] = 3.5

med_rate
```

Out[11]:

	userId	movieId	rating	predicted
0	1	1	4.0	3.5
1	1	3	4.0	3.5
2	1	6	4.0	3.5
3	1	47	5.0	3.5
4	1	50	5.0	3.5
...	...	...	...	...
100831	610	166534	4.0	3.5
100832	610	168248	5.0	3.5
100833	610	168250	5.0	3.5
100834	610	168252	5.0	3.5
100835	610	170875	3.0	3.5

100836 rows × 4 columns

In [12]:
```
rmse = np.sqrt(((med_rate['predicted'] - med_rate['rating']) ** 2).mean())
rmse
```

Out[12]:
```
1.0425252322754481
```

```
Aha! So by just predicting the median value, our RMSE is only 1.04.

#### Summary
Our random choice model was not very accurate, with an RMSE of 1.94. Our median value prediction model fared much better - and RMSE of 1.04. This is about a point off, which isn't bad for a model on a (0-5) scale.  We need a model to perform better than this!
```

5.2 `surpise` module models

Now that we have established RSME from random and median models, let's go ahead and try some of the beefier models in surprise. First, we're going to read in our dataset and establish test and trainsets. Then we will proceed to work through our model types.

Reading our Dataset

To begin, we will go through the process of reading in our dataset into the surprise dataset format. This will make the subsequent modeling a little more fluid.

In [17]:
```
#import the relevant item from surprise
from surprise.model_selection import cross_validate
from surprise.prediction_algorithms import SVD
from surprise.prediction_algorithms import KNNWithMeans, KNNBasic, KNNBase
from surprise.model_selection import GridSearchCV
from surprise.model_selection import train_test_split
from surprise import accuracy
```

As a way to validate the date, we're going to create a test and train set of data.

In [18]:
```
#read in dataset to surprise format
from surprise import Reader, Dataset
reader = Reader()
data = Dataset.load_from_df(ratings,reader)

# we will create a test set for validation, this will be used later when w
trainset, testset = train_test_split(data, test_size=0.2)
```

In [19]:
```
#check to make sure item's loaded properly and create a new trainset.
dataset = data.build_full_trainset()
print('Number of users: ', dataset.n_users, '\n')
print('Number of ratings: ', dataset.n_items)
```

```
Number of users:  610

Number of ratings:  9724
```

This matches our original check so... we've appeared to load the data successfully.

Model-Based Methods (Matrix Factorization) - SVD with suprise module

Below we will use the surprise method to create a SVD model, with tuned hyperparameters. We will utilize GridSearchCV for this.

In [20]:
```
## we will set up a SVD model with appropriate hyperparameters.

#established some initial hyperparameters
params = {'n_factors': [20, 50, 100],
          'reg_all': [0.02, 0.05, 0.1],
          'n_epochs': [5, 10, 15],
          'lr_all': [.002, .005, .010]}

#instantiate GridSearchCV model
g_s_svd = GridSearchCV(SVD,param_grid=params,n_jobs = -1,joblib_verbose=5)

#fit our ratings dataset "data" onto the model
g_s_svd.fit(data)
```

```
[Parallel(n_jobs=-1)]: Using backend LokyBackend with 4 concurrent workers.
[Parallel(n_jobs=-1)]: Done  10 tasks      | elapsed:   14.5s
[Parallel(n_jobs=-1)]: Done  64 tasks      | elapsed:   58.3s
[Parallel(n_jobs=-1)]: Done 154 tasks      | elapsed:  2.4min
[Parallel(n_jobs=-1)]: Done 280 tasks      | elapsed:  5.3min
[Parallel(n_jobs=-1)]: Done 405 out of 405 | elapsed:  9.5min finished
```

In [21]:
```
print(g_s_svd.best_score)
print(g_s_svd.best_params)
```

```
{'rmse': 0.8629065192573494, 'mae': 0.6627385253795459}
{'rmse': {'n_factors': 100, 'reg_all': 0.05, 'n_epochs': 15, 'lr_all': 0.01},
'mae': {'n_factors': 100, 'reg_all': 0.05, 'n_epochs': 15, 'lr_all': 0.01}}
```

Now let's run the model we have an print the results of our testset.

In [22]:
```
svd = SVD(n_factors=100, n_epochs=15, lr_all=0.010, reg_all=0.05)
svd.fit(trainset)
predictions = svd.test(testset)
print(accuracy.rmse(predictions))
```

```
RMSE: 0.8651
0.8650715807530015
```

Okay, we see a RMSE of .86. This... isn't bad on a scale of 0.5-5.0. Essentially it's under 1, which feels good and beats our baseline, but not under 0.5, which would feel better. Our model had an rmse of 0.87, let's establish that as OUR BASELINE MODEL.

Our optimal parameters are n_factors = 100 and reg_all = .05. This is convenient that these are in the middle of our range. We'll do a few quick spot checks to see if we can improve this.

In [23]:
```
svd = SVD(n_factors=100, n_epochs=20, lr_all=0.050, reg_all=0.05)
svd.fit(trainset)
predictions = svd.test(testset)
print(accuracy.rmse(predictions))
```

```
RMSE: 0.8640
0.8639629266342134
```

In [24]:
```
svd = SVD(n_factors=150, n_epochs=25, lr_all=0.010, reg_all=0.05)
svd.fit(trainset)
predictions = svd.test(testset)
print(accuracy.rmse(predictions))
```

```
RMSE: 0.8585
0.8585001642972838
```

In [25]:
```
svd = SVD(n_factors=200, n_epochs=30, lr_all=0.010, reg_all=0.05)
svd.fit(trainset)
predictions = svd.test(testset)
print(accuracy.rmse(predictions))
```

```
RMSE: 0.8597
0.8596641766038512
```

So... it did go down, but it barely moved. Suffice to say that perhaps we've created a largely optimized model. We can return to this later. Our hyperparameters are {n_factors=150, n_epochs=25, lr_all=0.010, reg_all=0.05}

Memory-Based Methods (Neighborhood-Based) KNN with surprise

To begin with, we can calculate the more simple neighborhood-based approaches. We can start with KNNBasic. With KNNBasic, we'll need a trainset and a testset in order to cross-validate results. We also run a few examples to determine the best hyperparameters

We'll import the relevant first.

In [26]:
```
#initiating KNN Basic with pearson similarity matric and user_based simili
knn_basic = KNNBasic(sim_options={'name':'pearson', 'user_based':True})
knn_basic.fit(trainset)
predictions = knn_basic.test(testset)
print(accuracy.rmse(predictions))
```

```
Computing the pearson similarity matrix...
Done computing similarity matrix.
RMSE: 0.9753
0.9753330994599322
```

With the KNN Basic, we have to set some of our hyper parameters. We'll try both "cosine" and "pearson". We'll also establish user based similarity, as there are fewer users than movies so this will save us considerable time. If we had thousands of users and only a handful of movies, we would consider an item based similarity.

Let's try cosine below.

In [27]:
```
#initiating KNN Basic with pearson correlation
knn_basic = KNNBasic(sim_options={'name':'cosine', 'user_based':True})
knn_basic.fit(trainset)
predictions = knn_basic.test(testset)
print(accuracy.rmse(predictions))
```

```
Computing the cosine similarity matrix...
Done computing similarity matrix.
RMSE: 0.9752
0.9751602971078402
```

In [28]:
```
#initiating KNN Basic with pearson correlation
knn_basic = KNNBasic(sim_options={'name':'pearson', 'user_based':False})
knn_basic.fit(trainset)
predictions = knn_basic.test(testset)
print(accuracy.rmse(predictions))
```

```
Computing the pearson similarity matrix...
Done computing similarity matrix.
RMSE: 0.9706
0.9706489484223293
```

In [29]:
```
#initiating KNN Basic with pearson correlation
knn_basic = KNNBasic(sim_options={'name':'cosine', 'user_based':False})
knn_basic.fit(trainset)
predictions = knn_basic.test(testset)
print(accuracy.rmse(predictions))
```

```
Computing the cosine similarity matrix...
Done computing similarity matrix.
RMSE: 0.9787
0.9787151489086232
```

Okay, so we tried to utilize both hyperparameters here, and we got a larger error. Nearly an entire point. We'll sidestep the cross-validation here and see if we can run a different neighborhood based model. This model utilizes ALS (Alternative Linear Squares) method. We'll try both options and see which is better.

In [30]:
```
# cross validating with KNNBaseline
knn_baseline = KNNBaseline(sim_options={'name':'pearson', 'user_based':Tru
knn_baseline.fit(trainset)
predictions = knn_baseline.test(testset)
print(accuracy.rmse(predictions))
```

```
Estimating biases using als...
Computing the pearson similarity matrix...
Done computing similarity matrix.
RMSE: 0.8801
0.8801035004209004
```

In [31]:
```
# cross validating with KNNBaseline
knn_baseline = KNNBaseline(sim_options={'name':'cosine', 'user_based':True
knn_baseline.fit(trainset)
predictions = knn_baseline.test(testset)
print(accuracy.rmse(predictions))
```

```
Estimating biases using als...
Computing the cosine similarity matrix...
Done computing similarity matrix.
RMSE: 0.8814
0.8813916557808571
```

So, the KNN Baseline module performs better thand the KNN Basic, but not better than the SVD.

Summary

The method with the lowest RMSE (0.859) was a user-based, SVD with tuned hyperparameters {n_factors=150, n_epochs=25, lr_all=0.010, reg_all=0.05}.

Let's go ahead and build our recommender using the SVD!!!

6. Implementation

6.1 Overview

Now that we have this model (Step 1), we will proceed to develop our recommender system, including how to work through the "cold start" problem. We will do that utilizing the following steps.

Step 1 (previously created): Prior to input from the new user, we created user-based collaborative filtering prediction model to predict how an existing user would rate a movie from the database.

Step 2: Prompt user to rate five movies.

Step 3: Add user's rating to the existing database

Step 4: Use the model from step 1 to predict how new users movie would rate (1-5) for all movies in the database and sort highest to lowest

Step 5: Output the top 5 recommendations

So... let's go to Step 2.

6.2 - Step 2: Prompt User

Below is the function used to prompt a user to rate movies, (1 - 5) for random movies in the database.

In [32]:

```
#create function to be called based on the number of movies created. This
def movie_rater(movie_df, num, last_user, genre=None):
    userID = last_user + random.randint(0,1000)
    rating_list = []

#loop through for each recommendation
    while num > 0:

        if genre:
            movie = movie_df[movie_df['genres'].str.contains(genre)].sampl
        else:
            movie = movie_df.sample(1)

        print(f"\n {movie.title} {movie.genres}\n")
        rating = input('How do you rate this movie on a scale of 1-5, pres

        if rating == 'n':
            continue
        elif (0 < float(rating) and float(rating) < 5.1):
            rating_one_movie = {'userId':userID,'movieId':movie['movieId']
            rating_list.append(rating_one_movie)
            num -= 1
        else:
            rating_again = input("Please choose either n, if you haven't s
            if rating_again == 'n':
                continue
            elif (0 < float(rating) and float(rating) < 5.1):
                rating_one_movie = {'userId':userID,'movieId':movie['movie
                rating_list.append(rating_one_movie)
                num -= 1
            else:
                print("You're struggling with directions. Let's try a diff
                continue
    return rating_list
```

Input

In [33]:
```
# Let's call our new function
last_user = ratings['userId'].max()
user_rating = movie_rater(movies_df, 5, last_user)
```

```
 6542    Sydney White (2007)
Name: title, dtype: object 6542    Comedy
Name: genres, dtype: object

How do you rate this movie on a scale of 1-5, press n if you have not seen:
n

 5153    Man Who Came to Dinner, The (1942)
Name: title, dtype: object 5153    Comedy
Name: genres, dtype: object

How do you rate this movie on a scale of 1-5, press n if you have not seen:
n

 8655    John Mulaney: New In Town (2012)
Name: title, dtype: object 8655    Comedy
Name: genres, dtype: object

How do you rate this movie on a scale of 1-5, press n if you have not seen:
5

 6895    Saw V (2008)
Name: title, dtype: object 6895    Crime|Horror|Thriller
Name: genres, dtype: object

How do you rate this movie on a scale of 1-5, press n if you have not seen:
4

 9491    CHiPS (2017)
Name: title, dtype: object 9491    Action|Comedy|Drama
Name: genres, dtype: object

How do you rate this movie on a scale of 1-5, press n if you have not seen:
1

 1984    Mummy, The (1959)
Name: title, dtype: object 1984    Horror
Name: genres, dtype: object

How do you rate this movie on a scale of 1-5, press n if you have not seen:
1

 1261    Starship Troopers (1997)
Name: title, dtype: object 1261    Action|Sci-Fi
Name: genres, dtype: object

How do you rate this movie on a scale of 1-5, press n if you have not seen:
2
```

Okay, so we prompted the user and got their viewing history. Let's move on to Step 3.

6.3 - Step 3: Add user ratings to database and rerun model

In [34]:
```
## add the new ratings to the original ratings DataFrame
user_ratings = pd.DataFrame(user_rating)
new_ratings_df = pd.concat([ratings, user_ratings], axis=0)
new_data = Dataset.load_from_df(new_ratings_df,reader)
```

Whelp... that was easy, we now have the user's information here in the database... Let's go to Step 4

6.4 - Step 4: Predict new user movie preferences

Now that we have a "new" database, similar to the old one, let's rerun our prediction model. First, we will rerun the model, and then we will create new predictions for all of the movie's in the database, sort from greatest to least.

In [36]:
```
# train a model using the new combined DataFrame, recall our parameters fr
svd = SVD(n_factors=150, n_epochs=25, lr_all=0.010, reg_all=0.05)
svd.fit(new_data.build_full_trainset())
```

Out[36]:
```
<surprise.prediction_algorithms.matrix_factorization.SVD at 0x2cfb3bc0520>
```

In [37]:
```
# make predictions for the user, to do this, predict ratings for every mov
list_of_movies = []

for m_id in ratings['movieId'].unique():
    list_of_movies.append((m_id,svd.predict(last_user,m_id)[3]))

ranked_movies = sorted(list_of_movies, key=lambda x:x[1], reverse=True)
```

6.5 - Step 5: Provide recommendations for 5 movies.

Now that we have a "new" database, similar to the old one, let's rerun our prediction model. First, we will rerun the model, and then we will create new predictions for all of the movie's in the database, sort from greatest to least.

In [38]:
```
# return the top n recommendations using the
def recommended_movies(user_ratings,movie_title_df,n):
        for idx, rec in enumerate(user_ratings):
            title = movie_title_df.loc[movie_title_df['movieId'] == int(re
            print('Recommendation # ', idx+1, ': ', title, '\n')
            n-= 1
            if n == 0:
                break

recommended_movies(ranked_movies,movies_df,5)
```

```
Recommendation #  1 :  659    Godfather, The (1972)
Name: title, dtype: object

Recommendation #  2 :  224    Star Wars: Episode IV - A New Hope (1977)
Name: title, dtype: object

Recommendation #  3 :  898    Star Wars: Episode V - The Empire Strikes Bac
k...
Name: title, dtype: object

Recommendation #  4 :  5621    Neon Genesis Evangelion: The End of Evangelio
n...
Name: title, dtype: object

Recommendation #  5 :  900    Raiders of the Lost Ark (Indiana Jones and th
e...
Name: title, dtype: object
```

And there we have it! I like all of these movies except for no. 4. I haven't seen it. Maybe I'll watch it later.

Summary

So... we were able to successfully implement a collaborative filtering model utilizing the surprise module and the Movielens database. Our model has an RMSE of .854, which is less than our baseline random model and less than 1 point. And... we were able to successfully implement to build a recommender system.

In [ ]:
```
1
```