Movie Recommendations

Author: Carlos Garza

Overview

This notebook details the creation and deployment of a recommendation system for movies. Utilizing the CRISP-DM framework, singular value decomposition, and various model tuning techniques, the backend of a recommendation application that takes input from a user regarding personal taste in genre and films previously watched and outputs a user defined quantity of movie recommendations was created.

Business Problem

A new streaming company called ML Movies wants to implement an active movie recommendation system for its users that takes user input to calculate a curated list of movie recommendations. Using a list of available films that have previously been rated by other users, develop a recommendation algorithm that generates curated movie recommendations.

Data

The data for this project is sourced from [MovieLens (https://grouplens.org/datasets/movielens/latest/)](https://grouplens.org/datasets/movielens/latest/). The data is summarized below.

In [1]:
```
# Import Libraries
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
```

In [2]:
```
# Import csv files as pd dataframes and examine first few lines

links = pd.read_csv('ml-latest-small/links.csv')
links.head(2)
```

Out[2]:

	movieId	imdbId	tmdbId
0	1	114709	862.0
1	2	113497	8844.0

In [3]:
```
movies = pd.read_csv('ml-latest-small/movies.csv')
movies.head(2)
```

Out[3]:

	movieId	title	genres
0	1	Toy Story (1995)	Adventure|Animation|Children|Comedy|Fantasy
1	2	Jumanji (1995)	Adventure|Children|Fantasy

In [4]:
```
ratings = pd.read_csv('ml-latest-small/ratings.csv')
ratings.head(2)
```

Out[4]:

	userId	movieId	rating	timestamp
0	1	1	4.0	964982703
1	1	3	4.0	964981247

In [5]:
```
tags = pd.read_csv('ml-latest-small/tags.csv')
tags.head(2)
```

Out[5]:

	userId	movieId	tag	timestamp
0	2	60756	funny	1445714994
1	2	60756	Highly quotable	1445714996

In [6]:
```
# check for missing data

links.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 9742 entries, 0 to 9741
Data columns (total 3 columns):
 #   Column   Non-Null Count  Dtype
---  ------   --------------  -----
 0   movieId  9742 non-null   int64
 1   imdbId   9742 non-null   int64
 2   tmdbId   9734 non-null   float64
dtypes: float64(1), int64(2)
memory usage: 228.5 KB
```

In [7]:
```
movies.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 9742 entries, 0 to 9741
Data columns (total 3 columns):
 #   Column   Non-Null Count  Dtype
---  ------   --------------  -----
 0   movieId  9742 non-null   int64
 1   title    9742 non-null   object
 2   genres   9742 non-null   object
dtypes: int64(1), object(2)
memory usage: 228.5+ KB
```

In [8]:
```
ratings.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 100836 entries, 0 to 100835
Data columns (total 4 columns):
 #   Column     Non-Null Count   Dtype  
---  ------     --------------   -----  
 0   userId     100836 non-null  int64  
 1   movieId    100836 non-null  int64  
 2   rating     100836 non-null  float64
 3   timestamp  100836 non-null  int64  
dtypes: float64(1), int64(3)
memory usage: 3.1 MB
```

In [9]:
```
tags.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 3683 entries, 0 to 3682
Data columns (total 4 columns):
 #   Column     Non-Null Count  Dtype 
---  ------     --------------  ----- 
 0   userId     3683 non-null   int64 
 1   movieId    3683 non-null   int64 
 2   tag        3683 non-null   object
 3   timestamp  3683 non-null   int64 
dtypes: int64(3), object(1)
memory usage: 115.2+ KB
```

In [10]:
```
# check ratings table for placeholder values

for column in ratings.columns:
    print(column)
    print(ratings[column].value_counts().head())
    print('\n')
```

```
userId
414    2698
599    2478
474    2108
448    1864
274    1346
Name: userId, dtype: int64


movieId
356     329
318     317
296     307
593     279
2571    278
Name: movieId, dtype: int64


rating
4.0    26818
3.0    20047
5.0    13211
3.5    13136
4.5     8551
Name: rating, dtype: int64


timestamp
1459787998    128
1459787997    124
1459787996     85
1459787995     37
828124616      37
Name: timestamp, dtype: int64
```

In [11]:
```
# check genres and genre combinations available
movies.genres.value_counts()
```

Out[11]:
```
Drama                                    1053
Comedy                                    946
Comedy|Drama                              435
Comedy|Romance                            363
Drama|Romance                             349
                                         ...
Animation|Sci-Fi|IMAX                       1
Mystery|Romance|Sci-Fi|Thriller             1
Action|Adventure|Crime|Horror|Thriller      1
Fantasy|Mystery|Western                     1
Crime|Drama|Thriller|War                    1
Name: genres, Length: 951, dtype: int64
```

The tables look relatively clean from the source. Some preprocessing is neccesary to drop unneccesary columns and format the movie genre column help filter preferences in the final algorithm.

Data Processing

The Ratings dataframe will be used to create the prediction model. The movies and links dataframes will be used to help with user defined filtering parameters and useful output information.

Unnecessary columns will be dropped and the genre column in the movies dataframe will be converted to a list before progressing to baseline models.

In [12]:
```
# Remove unneccesary columns from ratings and links dataframes
ratings.drop('timestamp', axis=1, inplace=True)
links.drop('tmdbId', axis=1, inplace=True)
```

In [13]:
```
# Convert genre column to genre list for easier genre searching in final al
movies['genreList'] = movies['genres'].map(lambda x: x.split('|'))
movies.drop('genres', axis=1, inplace=True)
```

In [14]:
```
movies.head()
```

Out[14]:

	movieId	title	genreList
0	1	Toy Story (1995)	[Adventure, Animation, Children, Comedy, Fantasy]
1	2	Jumanji (1995)	[Adventure, Children, Fantasy]
2	3	Grumpier Old Men (1995)	[Comedy, Romance]
3	4	Waiting to Exhale (1995)	[Comedy, Drama, Romance]
4	5	Father of the Bride Part II (1995)	[Comedy]

In [15]:
```
# gather list of genres for final algorithm
genres = []
for movie in movies.genreList:
    for genre in movie:
        if genre in genres:
            continue
        else:
            genres.append(genre)
```

Baseline Model

Models will be created using the Surprise library, as its implementation of singular value decompositon uses a modified algorithm created by Simon Funk that ignores items that have not been rated by users.

This will allow users to provide as much or as little input as they would like.

In [16]:
```
from surprise import Reader, Dataset
from surprise.prediction_algorithms import SVD, KNNWithMeans, KNNBasic
from surprise.model_selection import cross_validate
```

In [17]:
```
reader = Reader()
data = Dataset.load_from_df(ratings, reader)
dataset = data.build_full_trainset()
print('Number of users: ', dataset.n_users, '\n')
print('Number of items: ', dataset.n_items)
```

```
Number of users:  610

Number of items:  9724
```

To keep processing time down, algorithms will be user based.

Baselines will cross validated and evaluated by their RMSE.

In [18]:
```
# SVD
svd_baseline = SVD()
cv_svd_baseline = cross_validate(svd_baseline, data, n_jobs=-1)
```

In [19]:
```
# KNNWithMeans
knn_means_baseline = KNNWithMeans()
cv_knn_means_baseline = cross_validate(knn_means_baseline, data, n_jobs=-1)
```

In [20]:
```
# KNNBasic
knnBasic_baseline = KNNBasic()
cv_knnBasic_baseline = cross_validate(knnBasic_baseline, data, n_jobs=-1)
```

In [21]:
```
print('SVD average RMSE: ', np.mean(cv_svd_baseline['test_rmse']))
print('KNNWithMeans average RMSE: ', np.mean(cv_knn_means_baseline['test_rm
print('KNNBasic average RMSE: ', np.mean(cv_knnBasic_baseline['test_rmse']))
```

```
SVD average RMSE:  0.8731703288977742
KNNWithMeans average RMSE:  0.8976456118912
KNNBasic average RMSE:  0.9471913107084768
```

The baseline SVD model outperformed the K Nearest Neighbor models in initial tests, so that model will move forward with tuning.

Model Tuning

The model will be tuned using a cross validating grid search

In [22]:
```
from surprise.model_selection import GridSearchCV
```

In [23]:
```
params = {'n_factors':[50, 100],'n_epochs': [30, 35],
              'lr_all': [0.005, 0.007], 'reg_all': [0.07, 0.1]}
```

In [24]:
```
g_s_svd = GridSearchCV(SVD,param_grid=params,n_jobs=-1)
g_s_svd.fit(data)
print(g_s_svd.best_score)
print(g_s_svd.best_params)
```

```
{'rmse': 0.8564893160940918, 'mae': 0.6568883413528066}
{'rmse': {'n_factors': 100, 'n_epochs': 35, 'lr_all': 0.007, 'reg_all':
0.07}, 'mae': {'n_factors': 100, 'n_epochs': 35, 'lr_all': 0.007, 'reg_al
l': 0.07}}
```

After tuning hyperparameters, the model's RMSE slightly decreased.

Model Evaluation

With the model's hyperparameters tuned, the final model can be instantiated and cross validated to evaluate the model's performance before deployment.

In [25]:
```
final_svd = SVD(n_factors=100, n_epochs=35, lr_all=0.007, reg_all=0.07)
cv_final_svd = cross_validate(final_svd, data, n_jobs=-1)
```

In [26]:
```
print('Average RMSE: ', np.mean(cv_final_svd['test_rmse']))
print('Average MAE: ', np.mean(cv_final_svd['test_mae']))
print('Average fit time: ', np.mean(cv_final_svd['fit_time']))
print('Average test time: ', np.mean(cv_final_svd['test_time']))
```

```
Average RMSE:  0.8558114773852168
Average MAE:  0.6563600072242719
Average fit time:  6.536901664733887
Average test time:  0.10914049148559571
```

Type *Markdown* and LaTeX: α^2

Deployment

To use the model to provide recommendations, user input is first required to develop a profile for the user. The user's data is added to the ratings dataframe before our model is fit to the dataframe, generating predicted ratings for unwatched movies.

The pseudocode for the algorithm is as follows:

- Ask user if they are looking for a particular genre of movie. filter accordingly.
- Ask user how many movies they would like their predictions based upon.
- Gather ratings from users on their defined quantity of movies
- Add data to ratings dataframe
- Apply SVD model
- Ask user how many recommendations they would like
- Return appropriate amount of movie recommendations

The completed algorithm can be found in the deployment folder of this repository.

Conclusions

- Using singular value decomposition, the model created has a RMSE of 0.85.
- The .py file using the SVD model provides the flexibility to filter by genre and is intuitive enough for a non technical audience.
- The algorithm consistantly finds movies to recommend that it predicts the user will rate at least 4/5.

Future Work

In the future, this project can be imporved and expanded in a number of ways.

- Create GUI for a user to interact with the algorithm
- Code for possibility to select more than one genre
- Create more robust code that is more flexible with user input
- Create a way to save recommendations or save and update a user profile

Citation

F. Maxwell Harper and Joseph A. Konstan. 2015. The MovieLens Datasets: History and Context. ACM Transactions on Interactive Intelligent Systems (TiiS) 5, 4: 19:1–19:19. [https://doi.org/10.1145/2827872 (https://doi.org/10.1145/2827872)](https://doi.org/10.1145/2827872)