Analysing the Relation Between Users and their Preferred Genres

April 12, 2020

Analysing the Relation Between Users and their Preferred Genres

Project for Predictive Modelling by-

E003 Aditya Chakraborti

E015 Aaditya Naik

E019 Aditya Pansare

E020 Utkarsh Pant

E024 Dibyanshu Patnaik

E028 Kartikeya Prakash

[0]:
```
# Installing additional packages needed for the Analysis
!pip install geopandas
```

```
Collecting geopandas
  Downloading https://files.pythonhosted.org/packages/83/c5/3cf9cdc39a6f25
52922f79915f36b45a95b71fd343cfc51170a5b6ddb6e8/geopandas-0.7.0-py2.py3-none-
any.whl (928kB)
     || 931kB 1.8MB/s
Requirement already satisfied: pandas>=0.23.0 in
/usr/local/lib/python3.6/dist-packages (from geopandas) (1.0.3)
Collecting fiona
  Downloading https://files.pythonhosted.org/packages/ec/20/4e63bc5c6e62df
889297b382c3ccd4a7a488b00946aaaf81a118158c6f09/Fiona-1.8.13.post1-cp36-cp36m-man
ylinux1_x86_64.whl (14.7MB)
     || 14.7MB 301kB/s
Collecting pyproj>=2.2.0
  Downloading https://files.pythonhosted.org/packages/ce/37/705ee471f71130
d4ceee41bbcb06f3b52175cb89273cbb5755ed5e6374e0/pyproj-2.6.0-cp36-cp36m-manylinux
2010_x86_64.whl (10.4MB)
     || 10.4MB 36.2MB/s
Requirement already satisfied: shapely in /usr/local/lib/python3.6/dist-
packages (from geopandas) (1.7.0)
Requirement already satisfied: pytz>=2017.2 in /usr/local/lib/python3.6/dist-
packages (from pandas>=0.23.0->geopandas) (2018.9)
```

```
Requirement already satisfied: numpy>=1.13.3 in /usr/local/lib/python3.6/dist-
packages (from pandas>=0.23.0->geopandas) (1.18.2)
Requirement already satisfied: python-dateutil>=2.6.1 in
/usr/local/lib/python3.6/dist-packages (from pandas>=0.23.0->geopandas) (2.8.1)
Collecting cligj>=0.5
  Downloading https://files.pythonhosted.org/packages/e4/be/30a58b4b0733850280d0
1f8bd132591b4668ed5c7046761098d665ac2174/cligj-0.5.0-py3-none-any.whl
Requirement already satisfied: click<8,>=4.0 in /usr/local/lib/python3.6/dist-
packages (from fiona->geopandas) (7.1.1)
Requirement already satisfied: attrs>=17 in /usr/local/lib/python3.6/dist-
packages (from fiona->geopandas) (19.3.0)
Collecting click-plugins>=1.0
  Downloading https://files.pythonhosted.org/packages/e9/da/824b92d9942f4e472702
488857914bdd50f73021efea15b4cad9aca8ecef/click_plugins-1.1.1-py2.py3-none-
any.whl
Requirement already satisfied: six>=1.7 in /usr/local/lib/python3.6/dist-
packages (from fiona->geopandas) (1.12.0)
Collecting munch
  Downloading https://files.pythonhosted.org/packages/cc/ab/85d8da5c9a45e072301b
eb37ad7f833cd344e04c817d97e0cc75681d248f/munch-2.5.0-py2.py3-none-any.whl
Installing collected packages: cligj, click-plugins, munch, fiona, pyproj,
geopandas
Successfully installed click-plugins-1.1.1 cligj-0.5.0 fiona-1.8.13.post1
geopandas-0.7.0 munch-2.5.0 pyproj-2.6.0
```

[0]:
```
# a) Load libraries
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import matplotlib.ticker as ticker

%matplotlib inline

import seaborn as sns
import geopandas as gpd

from sklearn.metrics import r2_score
from sklearn.model_selection import train_test_split
from sklearn.model_selection import KFold
from sklearn.model_selection import cross_val_score
from sklearn.metrics import classification_report
from sklearn.metrics import confusion_matrix
from sklearn.metrics import accuracy_score
from sklearn.linear_model import LogisticRegression
from sklearn.tree import DecisionTreeClassifier
from sklearn.neighbors import KNeighborsClassifier
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis
```

```
from sklearn.naive_bayes import GaussianNB
from sklearn.svm import SVC

import pathlib
import urllib.request
from datetime import datetime
```

```
/usr/local/lib/python3.6/dist-packages/statsmodels/tools/_testing.py:19:
FutureWarning: pandas.util.testing is deprecated. Use the functions in the
public API at pandas.testing instead.
  import pandas.util.testing as tm
```

[0]:
```
# Loading all the necessary datasets (IMPORTANT- Requires an active internet␣
↪connection)
movie_data = pd.read_csv("https://raw.githubusercontent.com/aadityanaik/
↪movielens-analysis/master/movies.dat",
                        sep="::", header=None, names=['MovieID','Title','Genres'],
                        dtype={'MovieID': np.int32, 'Title': np.str, 'Genres': np.
↪str}, encoding = 'latin1', engine='python')
users_data = pd.read_csv("https://raw.githubusercontent.com/aadityanaik/
↪movielens-analysis/master/users.dat",
                        sep="::", header=None,␣
↪names=['UserID','Gender','Age','Occupation','Zip-code'],
    dtype={'UserID': np.int32, 'Gender': np.str, 'Age': np.int32, 'Occupation' :␣
↪np.int32, 'Zip-code': np.str}, engine='python')
ratings_data = pd.read_csv("https://raw.githubusercontent.com/aadityanaik/
↪movielens-analysis/master/ratings.dat",
                        sep="::", header=None,␣
↪names=['UserID','MovieID','Rating','Timestamp'],
                dtype={'UserID': np.int32, 'MovieID': np.int32, 'Rating': np.
↪int32, 'Timestamp' : np.str}, engine='python')
zip_data = pd.read_csv("https://raw.githubusercontent.com/aadityanaik/
↪movielens-analysis/master/zip_code_database.csv")
```

We now take a look at the head of each dataset

[0]:
```
movie_data.head()
```

[0]:

	MovieID	Title	Genres
0	1	Toy Story (1995)	Animation|Children's|Comedy
1	2	Jumanji (1995)	Adventure|Children's|Fantasy
2	3	Grumpier Old Men (1995)	Comedy|Romance
3	4	Waiting to Exhale (1995)	Comedy|Drama
4	5	Father of the Bride Part II (1995)	Comedy

[0]:
```
users_data.head()
```

[0]:

```
   UserID Gender  Age  Occupation Zip-code
0       1      F    1          10    48067
1       2      M   56          16    70072
2       3      M   25          15    55117
3       4      M   45           7    02460
4       5      M   25          20    55455
```

[0]:

```python
ratings_data.head()
```

[0]:

```
   UserID  MovieID  Rating  Timestamp
0       1     1193       5  978300760
1       1      661       3  978302109
2       1      914       3  978301968
3       1     3408       4  978300275
4       1     2355       5  978824291
```

[0]:

```python
zip_data.head()
```

[0]:

```
   zip      type  ...  longitude irs_estimated_population_2015
0  501    UNIQUE  ...     -73.04                           562
1  544    UNIQUE  ...     -73.04                             0
2  601  STANDARD  ...     -66.72                             0
3  602  STANDARD  ...     -67.18                             0
4  603  STANDARD  ...     -67.15                             0

[5 rows x 15 columns]
```

We now perform preprocessing on these datasets to get the tables that will be useful for us

[0]:

```python
# Convert all zip codes to String
zip_data['zip'] = zip_data['zip'].astype(str)

# Make sure all zip codes are of length 5
zip_data['zip'] = zip_data['zip'].apply(lambda x: x.zfill(5))
```

[0]:

```python
# Showing the number of users per zip-code
users_data['Zip-code'].value_counts()
```

[0]:

```
48104    19
22903    18
94110    17
55104    17
55105    16
         ..
33186     1
67204     1
48154     1
```

```
37774      1
32726      1
Name: Zip-code, Length: 3439, dtype: int64
```

[0]:
```
# Merging the user data with the zip data to get the state in which each user␣
↪lives
mapping = pd.merge(users_data, zip_data, how="left", left_on=['Zip-code'],␣
↪right_on=['zip'], validate='m:m')
# Dropping unnecessary columns
mapping= mapping[['UserID', 'Gender', 'Age', 'Occupation',␣
↪'zip','state','county','timezone']]
mapping.head()
```

[0]:
```
   UserID Gender  Age  ... state            county          timezone
0       1      F    1  ...    MI    Oakland County   America/Detroit
1       2      M   56  ...    LA  Jefferson Parish   America/Chicago
2       3      M   25  ...    MN     Ramsey County   America/Chicago
3       4      M   45  ...    MA  Middlesex County  America/New_York
4       5      M   25  ...    MN   Hennepin County   America/Chicago

[5 rows x 8 columns]
```

[0]:
```
# Finding the number of reviews per user
ratings_data.groupby(['UserID']).size()
```

[0]:
```
UserID
1        53
2       129
3        51
4        21
5       198
       ...
6036    888
6037    202
6038     20
6039    123
6040    341
Length: 6040, dtype: int64
```

[0]:
```
# Average rating given by each user
ratings_data.groupby(['UserID']).agg({'Rating':'mean'})
```

[0]:
```
          Rating
UserID
1       4.188679
2       3.713178
3       3.901961
```

```
4       4.190476
5       3.146465
...          ...
6036    3.302928
6037    3.717822
6038    3.800000
6039    3.878049
6040    3.577713

[6040 rows x 1 columns]
```

[0]:
```python
# Number of ratings given to each movie
ratings_data.groupby(['MovieID']).size().sort_values()
```

[0]:
```
MovieID
402        1
2214       1
3382       1
2217       1
2218       1
        ...
480     2672
1210    2883
1196    2990
260     2991
2858    3428
Length: 3706, dtype: int64
```

[0]:
```python
# Average rating for each movie
avg_rating = ratings_data.groupby(['MovieID']).agg({'Rating':'mean'})
print(avg_rating)
```

```
           Rating
MovieID
1        4.146846
2        3.201141
3        3.016736
4        2.729412
5        3.006757
...           ...
3948     3.635731
3949     4.115132
3950     3.666667
3951     3.900000
3952     3.780928

[3706 rows x 1 columns]
```

[0]:
```python
# Merging Movie data with it's average rating
movie_data = pd.merge(movie_data, avg_rating, on=["MovieID"])
movie_data.tail()
```

[0]:

	MovieID	Title	Genres	Rating
3701	3948	Meet the Parents (2000)	Comedy	3.635731
3702	3949	Requiem for a Dream (2000)	Drama	4.115132
3703	3950	Tigerland (2000)	Drama	3.666667
3704	3951	Two Family House (2000)	Drama	3.900000
3705	3952	Contender, The (2000)	Drama|Thriller	3.780928

[0]:
```python
# Sorting wrt average ratings in descending order
top_movies = movie_data['Rating'].sort_values(ascending= False).head(10)
top_movies.head()
```

[0]:
```
3152    5.0
2955    5.0
3367    5.0
3414    5.0
3054    5.0
Name: Rating, dtype: float64
```

[0]:
```python
# Converting the Genres into a more usable format (binary vectors to represent␣
↪the genre)
movie_data_g = pd.concat([movie_data, movie_data.Genres.str.
↪get_dummies(sep='|')], axis=1)
movie_data_g.head()
```

[0]:

	MovieID	Title	...	War	Western
0	1	Toy Story (1995)	...	0	0
1	2	Jumanji (1995)	...	0	0
2	3	Grumpier Old Men (1995)	...	0	0
3	4	Waiting to Exhale (1995)	...	0	0
4	5	Father of the Bride Part II (1995)	...	0	0

[5 rows x 22 columns]

[0]:
```python
# Get a list of genres
all_genres_i = movie_data.Genres.str.get_dummies(sep='|').columns
print(all_genres_i)
```

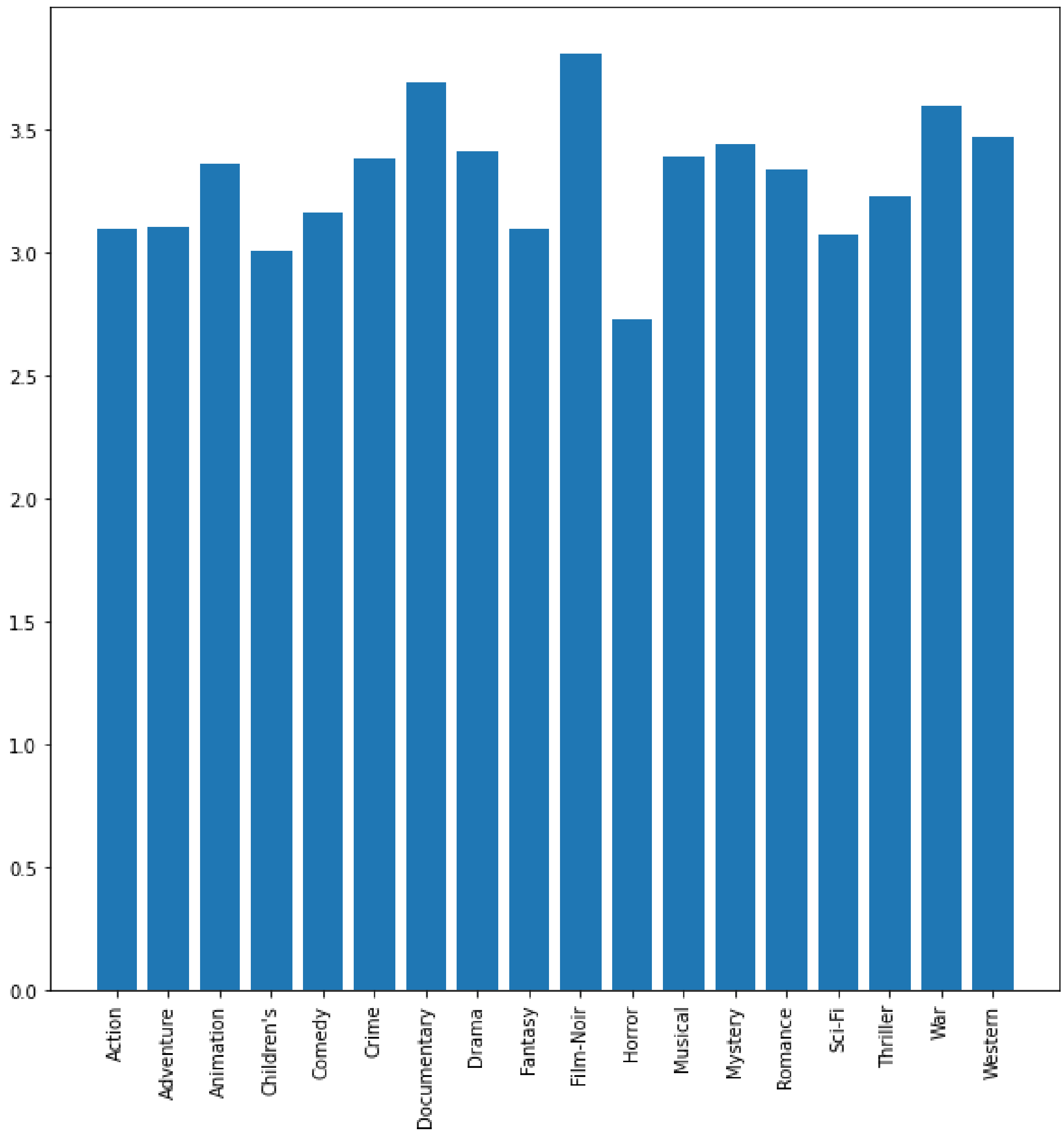
```
Index(['Action', 'Adventure', 'Animation', 'Children's', 'Comedy', 'Crime',
       'Documentary', 'Drama', 'Fantasy', 'Film-Noir', 'Horror', 'Musical',
       'Mystery', 'Romance', 'Sci-Fi', 'Thriller', 'War', 'Western'],
      dtype='object')
```

[0]:
```python
# Getting average rating for each genre

columns = ['genres']
all_genres= pd.DataFrame(all_genres_i, columns=columns)
all_genres.insert(1, "Rating", 0)
for iter in all_genres_i:
    #print(iter)
    x= movie_data_g.loc[movie_data_g[iter] == 1].agg({'Rating':'mean'})
    all_genres.loc[(all_genres.genres == iter),'Rating']= x[0]
all_genres
```

[0]:

	genres	Rating
0	Action	3.098295
1	Adventure	3.099664
2	Animation	3.358366
3	Children's	3.006325
4	Comedy	3.159924
5	Crime	3.385133
6	Documentary	3.687885
7	Drama	3.415331
8	Fantasy	3.098860
9	Film-Noir	3.806448
10	Horror	2.727053
11	Musical	3.388002
12	Mystery	3.440532
13	Romance	3.339019
14	Sci-Fi	3.073470
15	Thriller	3.225939
16	War	3.594314
17	Western	3.471926

[0]:
```python
# Plotting the average ratings in a bar graph
plt.figure(figsize=(10, 10))
plt.xticks(rotation=90)
plt.bar(all_genres['genres'], all_genres['Rating'])
plt.show()
```



[0]:
```
# Merging tables to get final workable table

u_r_map = pd.merge(mapping, ratings_data, how="left", on=["UserID"], validate='m:
 ↪m')
u_r_m_map = pd.merge(u_r_map, movie_data, how="left", on=["MovieID"])
final_map = pd.concat([u_r_m_map,u_r_m_map.Genres.str.get_dummies(sep='|')],␣
 ↪axis=1)
final_map
```

[0]:
```
          UserID Gender  Age  Occupation  ... Sci-Fi Thriller War Western
0              1      F    1          10  ...      0        0   0       0
```

```
1              1       F    1          10  ...       0         0   0         0
2              1       F    1          10  ...       0         0   0         0
3              1       F    1          10  ...       0         0   0         0
4              1       F    1          10  ...       0         0   0         0
...          ...     ...  ...         ...  ...     ...       ...  ..       ...
1000204     6040       M   25           6  ...       0         0   0         0
1000205     6040       M   25           6  ...       0         0   1         0
1000206     6040       M   25           6  ...       0         0   0         0
1000207     6040       M   25           6  ...       0         0   0         0
1000208     6040       M   25           6  ...       1         0   0         0

[1000209 rows x 32 columns]
```

[0]:
```
# To show the usefulness of the final map, we show the result of grouping by␣
 ↪User_ID
groups = final_map.groupby(['UserID'])
app = groups.apply(lambda x: x[x['Fantasy'] == 1])
app
```

[0]:
```
                UserID Gender  Age  Occupation  ... Sci-Fi Thriller War Western
UserID                                          ...
1      19            1      F    1          10  ...      0        0   0       0
       26            1      F    1          10  ...      1        0   0       0
       44            1      F    1          10  ...      1        0   0       0
2      60            2      M   56          16  ...      1        0   0       0
3      197           3      M   25          15  ...      0        0   0       0
...                ...    ...  ...         ...  ...    ...      ...  ..     ...
6040   999929     6040      M   25           6  ...      1        0   0       0
       999944     6040      M   25           6  ...      0        0   0       0
       999948     6040      M   25           6  ...      0        0   0       0
       1000198    6040      M   25           6  ...      1        0   0       0
       1000208    6040      M   25           6  ...      1        0   0       0

[36301 rows x 32 columns]
```

We now proceed to the data visualization.

We perform 5 main types of visualization, each of which will be described in the following cells.

First, we look at showing the statewise distribution of preferred genres by location.

Here, we consider the most popular genre in each state by taking a count of the number of reviews posted per genre in said states, and then plotted them over a map of the Mainland United States colour coded according to the genre.

[0]:
```
# Getting the shape files for the states
# NOTE- This requires an active internet connection

states_shape_filename = "cb_2018_us_state_20m.zip"
```

```
file_link = f"https://www2.census.gov/geo/tiger/GENZ2018/shp/
↪cb_2018_us_state_20m.zip"
states_shape_file = pathlib.Path(states_shape_filename)

with urllib.request.urlopen(file_link) as resp, open(states_shape_file, "wb") as␣
↪f:
  f.write(resp.read())
```

[0]:
```
# Reading the downloaded states shape files
states_gdf = gpd.read_file(f"zip://{states_shape_file}")
states_gdf.head()
```

[0]:
```
  STATEFP  ...                                           geometry
0      24  ...  MULTIPOLYGON (((-76.04621 38.02553, -76.00734 ...
1      19  ...  POLYGON ((-96.62187 42.77925, -96.57794 42.827...
2      10  ...  POLYGON ((-75.77379 39.72220, -75.75323 39.757...
3      39  ...  MULTIPOLYGON (((-82.86334 41.69369, -82.82572 ...
4      42  ...  POLYGON ((-80.51989 40.90666, -80.51964 40.987...

[5 rows x 10 columns]
```

[0]:
```
# Finding the preferred genre for each state

state_genre_count = {}
state_films_total = {}
i = 0
for _, row in final_map.iterrows():
  # if i >= 5:
  #   break
  state = row['state']
  genres = row['Genres'].split('|')
  # print(state, genres)

  if state not in state_genre_count:
    state_genre_count[state] = {}

  if state not in state_films_total:
    state_films_total[state] = 1
  else:
    state_films_total[state] += 1

  for genre in genres:
    if genre not in state_genre_count[state]:
      state_genre_count[state][genre] = 1
    else:
      state_genre_count[state][genre] += 1
```

```
  # i += 1

for state in state_genre_count:
  for genre in state_genre_count[state]:
    state_genre_count[state][genre] /= state_films_total[state]

statewise_pop_genre = []

for state in state_genre_count:
  max_count = 0
  max_genre = ""
  for genre in state_genre_count[state]:
    if state_genre_count[state][genre] > max_count:
      max_count = state_genre_count[state][genre]
      max_genre = genre
    elif state_genre_count[state][genre] == max_count:
      max_genre += "|" + genre
  statewise_pop_genre.append([state, max_genre])

statewise_pop_genre = pd.DataFrame(data = statewise_pop_genre, columns =␣
 ↪['STUSPS', 'genre'])

statewise_pop_genre.head()
```

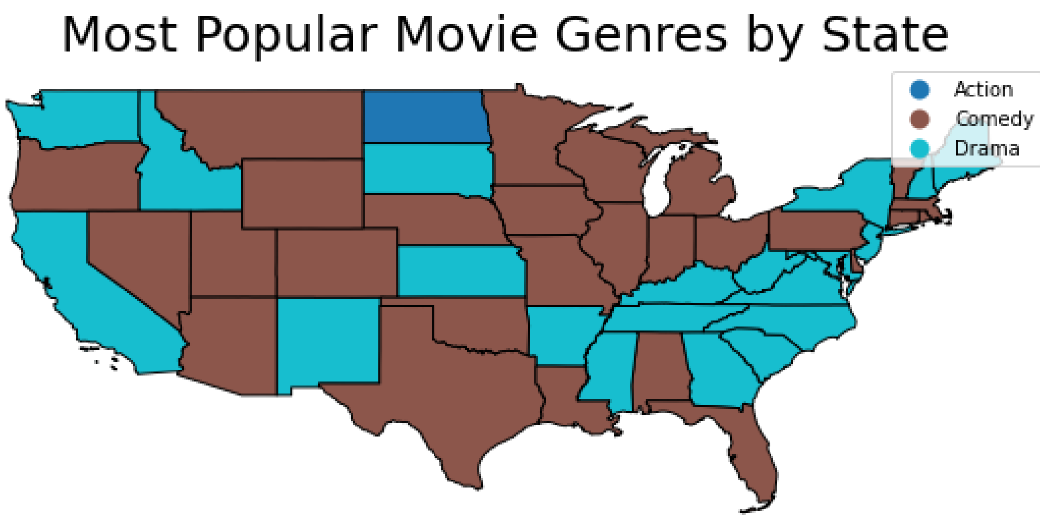
[0]:

	STUSPS	genre
0	MI	Comedy
1	LA	Comedy
2	MN	Comedy
3	MA	Comedy
4	CT	Comedy

[0]:

```
# Plotting the preferred genre for each state
loc_genre = states_gdf.merge(statewise_pop_genre, left_on='STUSPS',␣
 ↪right_on='STUSPS')
loc_genre = loc_genre.loc[~loc_genre['STUSPS'].isin(['PR', 'HI', 'AK'])]
fig, ax = plt.subplots(1, figsize=(10, 6))
ax.set_title('Most Popular Movie Genres by State', fontdict={'fontsize': '25',␣
 ↪'fontweight' : '3'})
ax.axis('off')
loc_genre.plot(column='genre', linewidth=1, ax=ax, edgecolor='0', legend='True')
```

[0]: <matplotlib.axes._subplots.AxesSubplot at 0x7fc28cc36710>



Plotting a heatmap showing the average ratings of age groups for each genre

The average ratings across age ranges can be accurately represented using heatmaps, such the the colour of each cell in the heatmap corresponds to the average rating users of a particular age group gave movies of each Genre as a whole. Below, we filter the main dataset by Genre and aggregate the ratings by average across age groups.*italicized text*

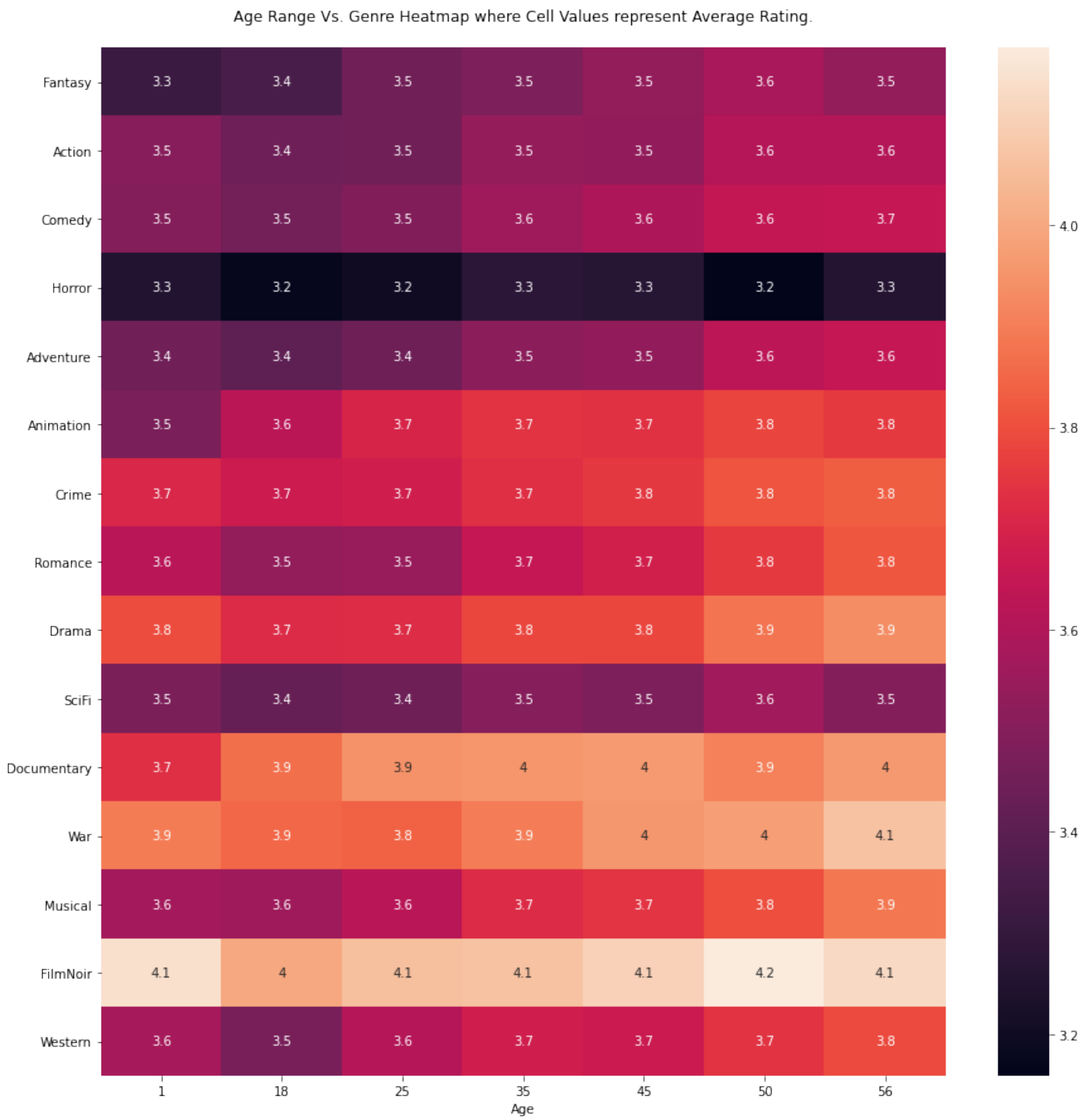
[0]:
```
# Renaming data frames
final_ratings_map = pd.DataFrame(final_map)

final_ratings_map = final_ratings_map.rename(columns = {"Rating_x": "Rating"})
final_ratings_map = final_ratings_map.rename(columns = {"Children's":␣
↪"Children"})
final_ratings_map = final_ratings_map.rename(columns = {"Sci-Fi": "SciFi"})
final_ratings_map = final_ratings_map.rename(columns = {"Film-Noir": "FilmNoir"})
```

[0]:
```
fantasy_df = final_ratings_map.query("Fantasy == 1")
fantasy_df = fantasy_df.groupby('Age').agg({'Rating': "mean"}).rename(columns =␣
↪{"Rating": "Fantasy"})

action_df = final_ratings_map.query("Action == 1")
action_df = action_df.groupby('Age').agg({'Rating': "mean"}).rename(columns =␣
↪{"Rating": "Action"})

comedy_df = final_ratings_map.query("Comedy == 1")
comedy_df = comedy_df.groupby('Age').agg({'Rating': "mean"}).rename(columns =␣
↪{"Rating": "Comedy"})

horror_df = final_ratings_map.query("Horror == 1")
```

```python
horror_df = horror_df.groupby('Age').agg({'Rating': "mean"}).rename(columns = {"Rating": "Horror"})

adventure_df = final_ratings_map.query("Adventure == 1")
adventure_df = adventure_df.groupby('Age').agg({'Rating': "mean"}).rename(columns = {"Rating": "Adventure"})

animation_df = final_ratings_map.query("Animation == 1")
animation_df = animation_df.groupby('Age').agg({'Rating': "mean"}).rename(columns = {"Rating": "Animation"})

crime_df = final_ratings_map.query("Crime == 1")
crime_df = crime_df.groupby('Age').agg({'Rating': "mean"}).rename(columns = {"Rating": "Crime"})

romance_df = final_ratings_map.query("Romance == 1")
romance_df = romance_df.groupby('Age').agg({'Rating': "mean"}).rename(columns = {"Rating": "Romance"})

childrens_df = final_ratings_map.query("Children == 1")
childrens_df = childrens_df.groupby('Age').agg({'Rating': "mean"}).rename(columns = {"Rating": "Children"})

drama_df = final_ratings_map.query("Drama == 1")
drama_df = drama_df.groupby('Age').agg({'Rating': "mean"}).rename(columns = {"Rating": "Drama"})

scifi_df = final_ratings_map.query(r"SciFi == 1")
scifi_df = scifi_df.groupby('Age').agg({'Rating': "mean"}).rename(columns = {"Rating": "SciFi"})

doc_df = final_ratings_map.query("Documentary == 1")
doc_df = doc_df.groupby('Age').agg({'Rating': "mean"}).rename(columns = {"Rating": "Documentary"})

war_df = final_ratings_map.query("War == 1")
war_df = war_df.groupby('Age').agg({'Rating': "mean"}).rename(columns = {"Rating": "War"})

musical_df = final_ratings_map.query("Musical == 1")
musical_df = musical_df.groupby('Age').agg({'Rating': "mean"}).rename(columns = {"Rating": "Musical"})

mystery_df = final_ratings_map.query("Mystery == 1")
mystery_df = mystery_df.groupby('Age').agg({'Rating': "mean"}).rename(columns = {"Rating": "Mystery"})
```

```
thriller_df = final_ratings_map.query("Thriller == 1")
thriller_df = thriller_df.groupby('Age').agg({'Rating': "mean"}).rename(columns␣
 ↪= {"Rating": "Thriller"})

filmnoir_df = final_ratings_map.query("FilmNoir == 1")
filmnoir_df = filmnoir_df.groupby('Age').agg({'Rating': "mean"}).rename(columns␣
 ↪= {"Rating": "FilmNoir"})

western_df = final_ratings_map.query("Western == 1")
western_df = western_df.groupby('Age').agg({'Rating': "mean"}).rename(columns =␣
 ↪{"Rating": "Western"})

heat_df = fantasy_df.join(action_df, how = 'left', on = 'Age')
heat_df = heat_df.join(comedy_df, how = 'left', on = 'Age')
heat_df = heat_df.join(horror_df, how = 'left', on = 'Age')
heat_df = heat_df.join(adventure_df, how = 'left', on = 'Age')
heat_df = heat_df.join(animation_df, how = 'left', on = 'Age')
heat_df = heat_df.join(crime_df, how = 'left', on = 'Age')
heat_df = heat_df.join(romance_df, how = 'left', on = 'Age')
heat_df = heat_df.join(drama_df, how = 'left', on = 'Age')
heat_df = heat_df.join(scifi_df, how = 'left', on = 'Age')
heat_df = heat_df.join(doc_df, how = 'left', on = 'Age')
heat_df = heat_df.join(war_df, how = 'left', on = 'Age')
heat_df = heat_df.join(musical_df, how = 'left', on = 'Age')
heat_df = heat_df.join(filmnoir_df, how = 'left', on = 'Age')
heat_df = heat_df.join(western_df, how = 'left', on = 'Age')

plt.subplots(figsize=(15, 15))
sns.heatmap(heat_df.T, annot = True).set(title="Age Range Vs. Genre Heatmap␣
 ↪where Cell Values represent Average Rating.\n")
```

```
[0]: [Text(0.5, 1.0, 'Age Range Vs. Genre Heatmap where Cell Values represent Average
     Rating.\n')]
```



	1	18	25	35	45	50	56
Fantasy	3.3	3.4	3.5	3.5	3.5	3.6	3.5
Action	3.5	3.4	3.5	3.5	3.5	3.6	3.6
Comedy	3.5	3.5	3.5	3.6	3.6	3.6	3.7
Horror	3.3	3.2	3.2	3.3	3.3	3.2	3.3
Adventure	3.4	3.4	3.4	3.5	3.5	3.6	3.6
Animation	3.5	3.6	3.7	3.7	3.7	3.8	3.8
Crime	3.7	3.7	3.7	3.7	3.8	3.8	3.8
Romance	3.6	3.5	3.5	3.7	3.7	3.8	3.8
Drama	3.8	3.7	3.7	3.8	3.8	3.9	3.9
SciFi	3.5	3.4	3.4	3.5	3.5	3.6	3.5
Documentary	3.7	3.9	3.9	4	4	3.9	4
War	3.9	3.9	3.8	3.9	4	4	4.1
Musical	3.6	3.6	3.6	3.7	3.7	3.8	3.9
FilmNoir	4.1	4	4.1	4.1	4.1	4.2	4.1
Western	3.6	3.5	3.6	3.7	3.7	3.7	3.8

Plotting a bar graph to show the average ratings of genres as given by each gender.

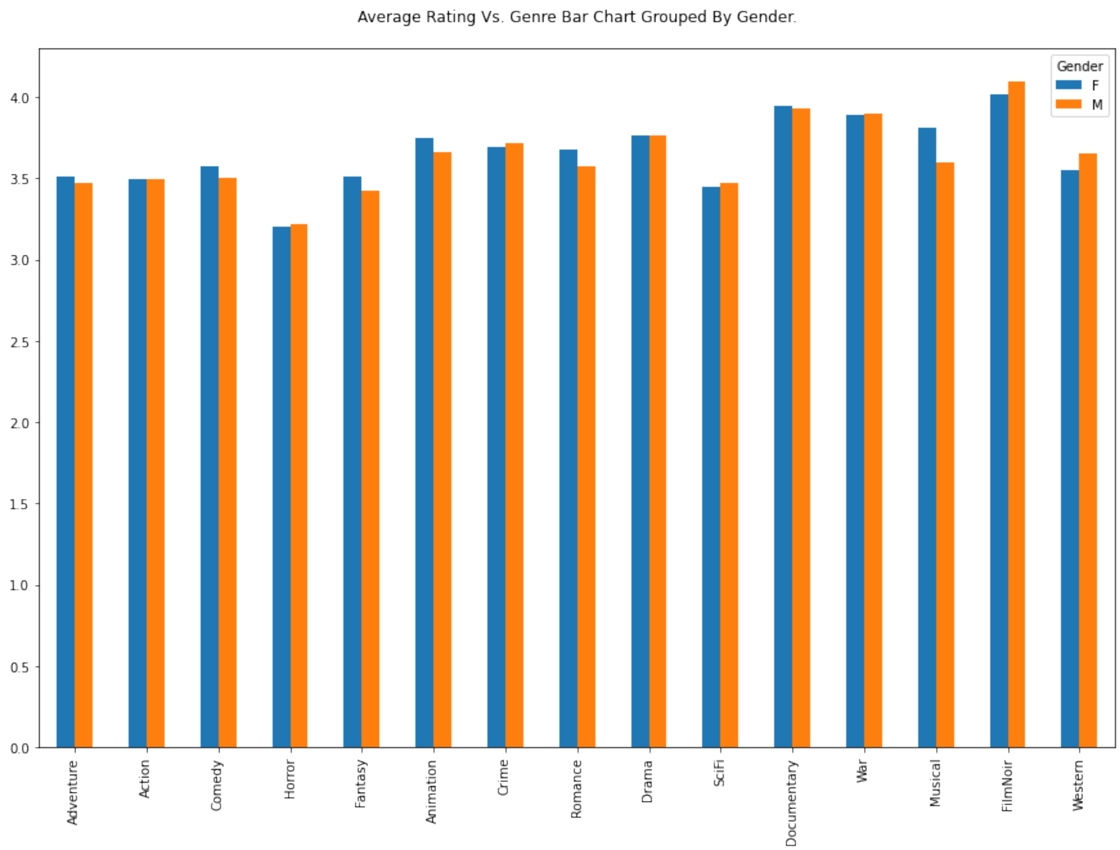
The average ratings across Genres can be accurately represented using bifurcated Bar Plots, such the the height of each bar in the plot corresponds to the average rating users of a particular age group gave movies of each Genre as a whole. Meanwhile, two bars in different colours uniquely represent Male and Female ratings for each Genre. Below, we filter the main dataset by Genre and aggregate the ratings by average across Gender.

[0]:
```
gen_ratings_by_ad = final_ratings_map.query('Adventure == 1')
gen_ratings_by_ad = gen_ratings_by_ad.groupby('Gender').agg({"Rating": "mean"}).
 ↪rename(columns = {"Rating": "Adventure"})

gen_ratings_by_action = final_ratings_map.query("Action == 1")
```

```
gen_ratings_by_action = gen_ratings_by_action.groupby('Gender').agg({"Rating": "mean"}).rename(columns = {"Rating": "Action"})

gen_ratings_by_f = final_ratings_map.query('Fantasy == 1')
gen_ratings_by_f = gen_ratings_by_f.groupby('Gender').agg({"Rating": "mean"}).rename(columns = {"Rating": "Fantasy"})

gen_ratings_by_comedy = final_ratings_map.query('Comedy == 1')
gen_ratings_by_comedy = gen_ratings_by_comedy.groupby('Gender').agg({"Rating": "mean"}).rename(columns = {"Rating": "Comedy"})

gen_ratings_by_horror = final_ratings_map.query('Horror == 1')
gen_ratings_by_horror = gen_ratings_by_horror.groupby('Gender').agg({"Rating": "mean"}).rename(columns = {"Rating": "Horror"})

gen_ratings_by_anim = final_ratings_map.query('Animation == 1')
gen_ratings_by_anim = gen_ratings_by_anim.groupby('Gender').agg({"Rating": "mean"}).rename(columns = {"Rating": "Animation"})

gen_ratings_by_crime = final_ratings_map.query('Crime == 1')
gen_ratings_by_crime = gen_ratings_by_crime.groupby('Gender').agg({"Rating": "mean"}).rename(columns = {"Rating": "Crime"})

gen_ratings_by_rom = final_ratings_map.query('Romance == 1')
gen_ratings_by_rom = gen_ratings_by_rom.groupby('Gender').agg({"Rating": "mean"}).rename(columns = {"Rating": "Romance"})

gen_ratings_by_drama = final_ratings_map.query('Drama == 1')
gen_ratings_by_drama = gen_ratings_by_drama.groupby('Gender').agg({"Rating": "mean"}).rename(columns = {"Rating": "Drama"})

gen_ratings_by_scifi = final_ratings_map.query('SciFi == 1')
gen_ratings_by_scifi = gen_ratings_by_scifi.groupby('Gender').agg({"Rating": "mean"}).rename(columns = {"Rating": "SciFi"})

gen_ratings_by_docu = final_ratings_map.query('Documentary == 1')
gen_ratings_by_docu = gen_ratings_by_docu.groupby('Gender').agg({"Rating": "mean"}).rename(columns = {"Rating": "Documentary"})

gen_ratings_by_war = final_ratings_map.query('War == 1')
gen_ratings_by_war = gen_ratings_by_war.groupby('Gender').agg({"Rating": "mean"}).rename(columns = {"Rating": "War"})

gen_ratings_by_music = final_ratings_map.query('Musical == 1')
gen_ratings_by_music = gen_ratings_by_music.groupby('Gender').agg({"Rating": "mean"}).rename(columns = {"Rating": "Musical"})
```

```
gen_ratings_by_fn = final_ratings_map.query('FilmNoir == 1')
gen_ratings_by_fn = gen_ratings_by_fn.groupby('Gender').agg({"Rating": "mean"}).
 ↪rename(columns = {"Rating": "FilmNoir"})

gen_ratings_by_west = final_ratings_map.query('Western == 1')
gen_ratings_by_west = gen_ratings_by_west.groupby('Gender').agg({"Rating":␣
 ↪"mean"}).rename(columns = {"Rating": "Western"})

bar_df = gen_ratings_by_ad.join(gen_ratings_by_action, how = "left", on =␣
 ↪"Gender")
bar_df = bar_df.join(gen_ratings_by_comedy, how = "left", on = "Gender")
bar_df = bar_df.join(gen_ratings_by_horror, how = "left", on = "Gender")
bar_df = bar_df.join(gen_ratings_by_f, how = "left", on = "Gender")
bar_df = bar_df.join(gen_ratings_by_anim, how = "left", on = "Gender")
bar_df = bar_df.join(gen_ratings_by_crime, how = "left", on = "Gender")
bar_df = bar_df.join(gen_ratings_by_rom, how = "left", on = "Gender")
bar_df = bar_df.join(gen_ratings_by_drama, how = "left", on = "Gender")
bar_df = bar_df.join(gen_ratings_by_scifi, how = "left", on = "Gender")
bar_df = bar_df.join(gen_ratings_by_docu, how = "left", on = "Gender")
bar_df = bar_df.join(gen_ratings_by_war, how = "left", on = "Gender")
bar_df = bar_df.join(gen_ratings_by_music, how = "left", on = "Gender")
bar_df = bar_df.join(gen_ratings_by_fn, how = "left", on = "Gender")
bar_df = bar_df.join(gen_ratings_by_west, how = "left", on = "Gender")

bar_df.T.plot.bar(figsize=(15, 10), by = "Gender").set(title="Average Rating Vs.␣
 ↪Genre Bar Chart Grouped By Gender.\n")
```

```
[0]: [Text(0.5, 1.0, 'Average Rating Vs. Genre Bar Chart Grouped By Gender.\n')]
```



Plotting a heatmap for the average ratings of each genre given by people from each occupation.

The average ratings across occupation can be accurately represented using heatmaps, such the the colour of each cell in the heatmap corresponds to the average rating users of a particular occupation gave movies of each Genre as a whole. Below, we filter the main dataset by Genre and aggregate the ratings by average across occupations.

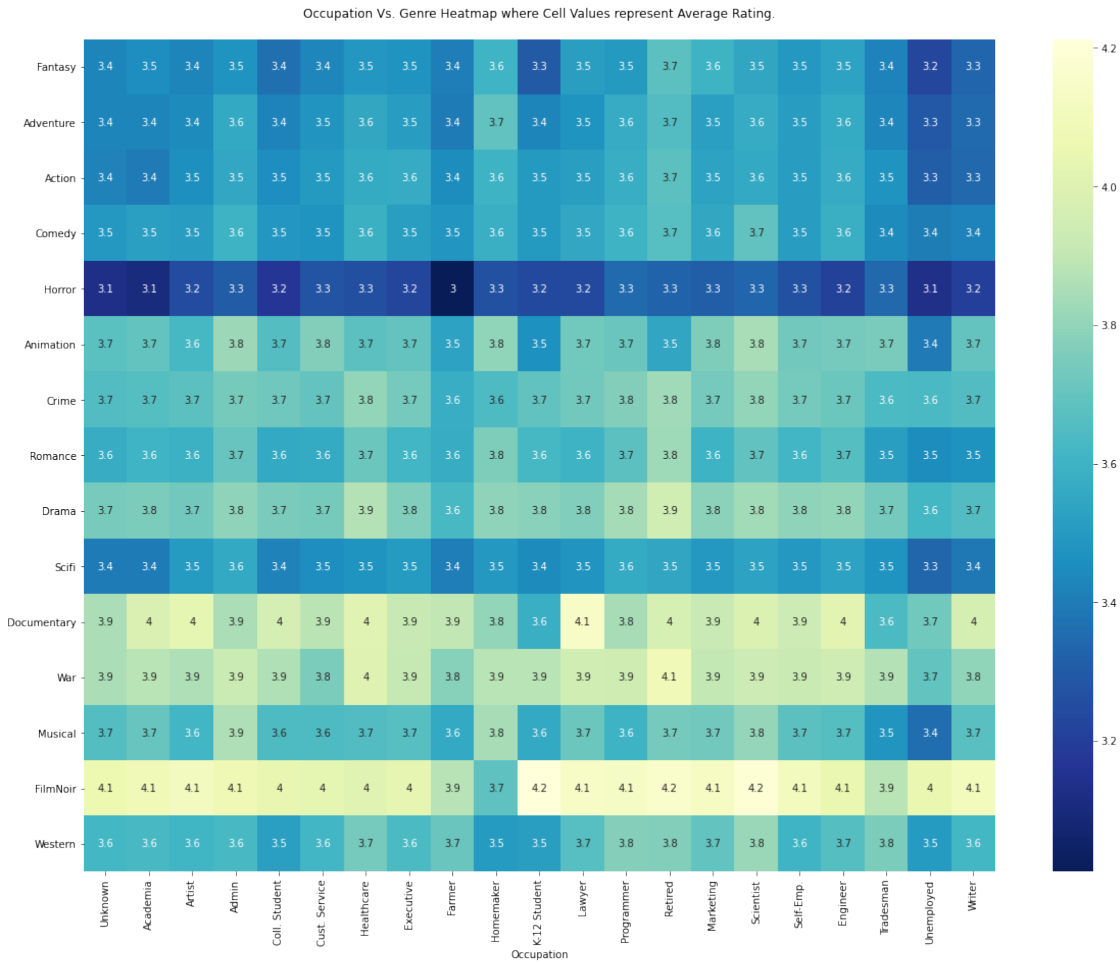
Occupation is chosen from the following choices: 0: "other" or not specified 1: "academic/educator" 2: "artist" 3: "clerical/admin" 4: "college/grad student" 5: "customer service" 6: "doctor/health care" 7: "executive/managerial" 8: "farmer" 9: "homemaker" 10: "K-12 student" 11: "lawyer" 12: "programmer" 13: "retired" 14: "sales/marketing" 15: "scientist" 16: "self-employed" 17: "technician/engineer" 18: "tradesman/craftsman" 19: "unemployed" * 20: "writer"

[0]:
```
fantasy_df_occ = final_ratings_map.query("Fantasy == 1")
fantasy_df_occ = fantasy_df_occ.groupby("Occupation").agg({"Rating": "mean"}).T.
 ↪rename(index={"Rating": "Fantasy"})



ad_df_occ = final_ratings_map.query("Adventure == 1")
ad_df_occ = ad_df_occ.groupby("Occupation").agg({"Rating": "mean"}).T.
 ↪rename(index={"Rating": "Adventure"})
```

```
ac_df_occ = final_ratings_map.query("Action == 1")
ac_df_occ = ac_df_occ.groupby("Occupation").agg({"Rating": "mean"}).T.
 ↪rename(index={"Rating": "Action"})

comedy_df_occ = final_ratings_map.query("Comedy == 1")
comedy_df_occ = comedy_df_occ.groupby("Occupation").agg({"Rating": "mean"}).T.
 ↪rename(index={"Rating": "Comedy"})

horror_df_occ = final_ratings_map.query("Horror == 1")
horror_df_occ = horror_df_occ.groupby("Occupation").agg({"Rating": "mean"}).T.
 ↪rename(index={"Rating": "Horror"})

anim_df_occ = final_ratings_map.query("Animation == 1")
anim_df_occ = anim_df_occ.groupby("Occupation").agg({"Rating": "mean"}).T.
 ↪rename(index={"Rating": "Animation"})

crime_df_occ = final_ratings_map.query("Crime == 1")
crime_df_occ = crime_df_occ.groupby("Occupation").agg({"Rating": "mean"}).T.
 ↪rename(index={"Rating": "Crime"})

rom_df_occ = final_ratings_map.query("Romance == 1")
rom_df_occ = rom_df_occ.groupby("Occupation").agg({"Rating": "mean"}).T.
 ↪rename(index={"Rating": "Romance"})

drama_df_occ = final_ratings_map.query("Drama == 1")
drama_df_occ = drama_df_occ.groupby("Occupation").agg({"Rating": "mean"}).T.
 ↪rename(index={"Rating": "Drama"})

scifi_df_occ = final_ratings_map.query("SciFi == 1")
scifi_df_occ = scifi_df_occ.groupby("Occupation").agg({"Rating": "mean"}).T.
 ↪rename(index={"Rating": "Scifi"})

docu_df_occ = final_ratings_map.query("Documentary == 1")
docu_df_occ = docu_df_occ.groupby("Occupation").agg({"Rating": "mean"}).T.
 ↪rename(index={"Rating": "Documentary"})

war_df_occ = final_ratings_map.query("War == 1")
war_df_occ = war_df_occ.groupby("Occupation").agg({"Rating": "mean"}).T.
 ↪rename(index={"Rating": "War"})

musical_df_occ = final_ratings_map.query("Musical == 1")
musical_df_occ = musical_df_occ.groupby("Occupation").agg({"Rating": "mean"}).T.
 ↪rename(index={"Rating": "Musical"})

fn_df_occ = final_ratings_map.query("FilmNoir == 1")
```

```
fn_df_occ = fn_df_occ.groupby("Occupation").agg({"Rating": "mean"}).T.
 ↪rename(index={"Rating": "FilmNoir"})

west_df_occ = final_ratings_map.query("Western == 1")
west_df_occ = west_df_occ.groupby("Occupation").agg({"Rating": "mean"}).T.
 ↪rename(index={"Rating": "Western"})


occ_df = pd.concat([fantasy_df_occ, ad_df_occ, ac_df_occ, comedy_df_occ,␣
 ↪horror_df_occ, anim_df_occ, crime_df_occ, rom_df_occ, drama_df_occ,␣
 ↪scifi_df_occ, docu_df_occ, war_df_occ, musical_df_occ, fn_df_occ, west_df_occ])

plt.subplots(figsize=(20, 15))
occupations = ["Unknown", "Academia", "Artist", "Admin", "Coll. Student", "Cust.␣
 ↪Service", "Healthcare", "Executive", "Farmer", "Homemaker", "K-12 Student",␣
 ↪"Lawyer", "Programmer", "Retired", "Marketing", "Scientist", "Self-Emp.",␣
 ↪"Engineer", "Tradesman", "Unemployed", "Writer"]
sns.heatmap(data = occ_df, annot = True, xticklabels=occupations,␣
 ↪cmap="YlGnBu_r").set(title="Occupation Vs. Genre Heatmap where Cell Values␣
 ↪represent Average Rating.\n")
```

[0]:
```
[Text(0.5, 1.0, 'Occupation Vs. Genre Heatmap where Cell Values represent
Average Rating.\n')]
```



Now we plot the timeseries graph of the average genre ratings over various intervals of time.

We have two forms of timeseries- monthwise and yearwise. In yearwise, we look at average ratings for each genre per year and plot them as a timeseries graph, whereas in monthwise, we consider the average ratings of each genre per month. The months range from 2000-04 (April 2000) to 2003-02 (February 2003).

[0]:
```
# Finding the average rating of each genre by year
year_genre_popularity = {}
year_genre_ratings = {}
years = []

year_map = pd.DataFrame(final_map)

year_map['Year'] = year_map.apply(lambda row : int(datetime.
 ↪utcfromtimestamp(int(row['Timestamp'])).strftime('%Y')), axis=1)

yearwise_genre_rating = {}
```

```python
for genre in all_genres['genres']:
  year_grouped = year_map.groupby(['Year', genre])
  app = year_grouped.agg({'Rating_x' : ['mean']})
  app = app.reset_index()
  app = pd.DataFrame(app[app[genre] == 1])

  for _, row in app.iterrows():
    y, r = int(row.values[0]), float(row.values[2])
    if genre not in yearwise_genre_rating:
      yearwise_genre_rating[genre] = {}
    yearwise_genre_rating[genre][y] = r

yearwise_genre_rating_dataframe = pd.DataFrame(yearwise_genre_rating)

yearwise_genre_rating_dataframe
```

[0]:

```
        Action  Adventure  Animation  ...  Thriller       War   Western
2000  3.501721   3.487544   3.699725  ...  3.581210  3.902247  3.641338
2001  3.378504   3.368108   3.549833  ...  3.473826  3.793912  3.603102
2002  3.303536   3.280788   3.570975  ...  3.420596  3.726646  3.587234
2003  3.378539   3.397143   3.422680  ...  3.431862  3.934783  3.554054

[4 rows x 18 columns]
```

[0]:

```python
# Finding the average rating of each genre by month
month_map = pd.DataFrame(final_map)

month_map['Month'] = month_map.apply(lambda row : datetime.
 ↪utcfromtimestamp(int(row['Timestamp'])).strftime('%Y-%m'), axis=1)

monthwise_genre_rating = {}

for genre in all_genres['genres']:
  month_grouped = month_map.groupby(['Month', genre])
  app = month_grouped.agg({'Rating_x' : ['mean']})
  app = app.reset_index()
  app = pd.DataFrame(app[app[genre] == 1])
  for _, row in app.iterrows():
    m, r = row.values[0], float(row.values[2])

    if genre not in monthwise_genre_rating:
      monthwise_genre_rating[genre] = {}

    monthwise_genre_rating[genre][m] = r

monthwise_genre_rating_dataframe = pd.DataFrame(monthwise_genre_rating)
```

```
monthwise_genre_rating_dataframe
```

[0]:

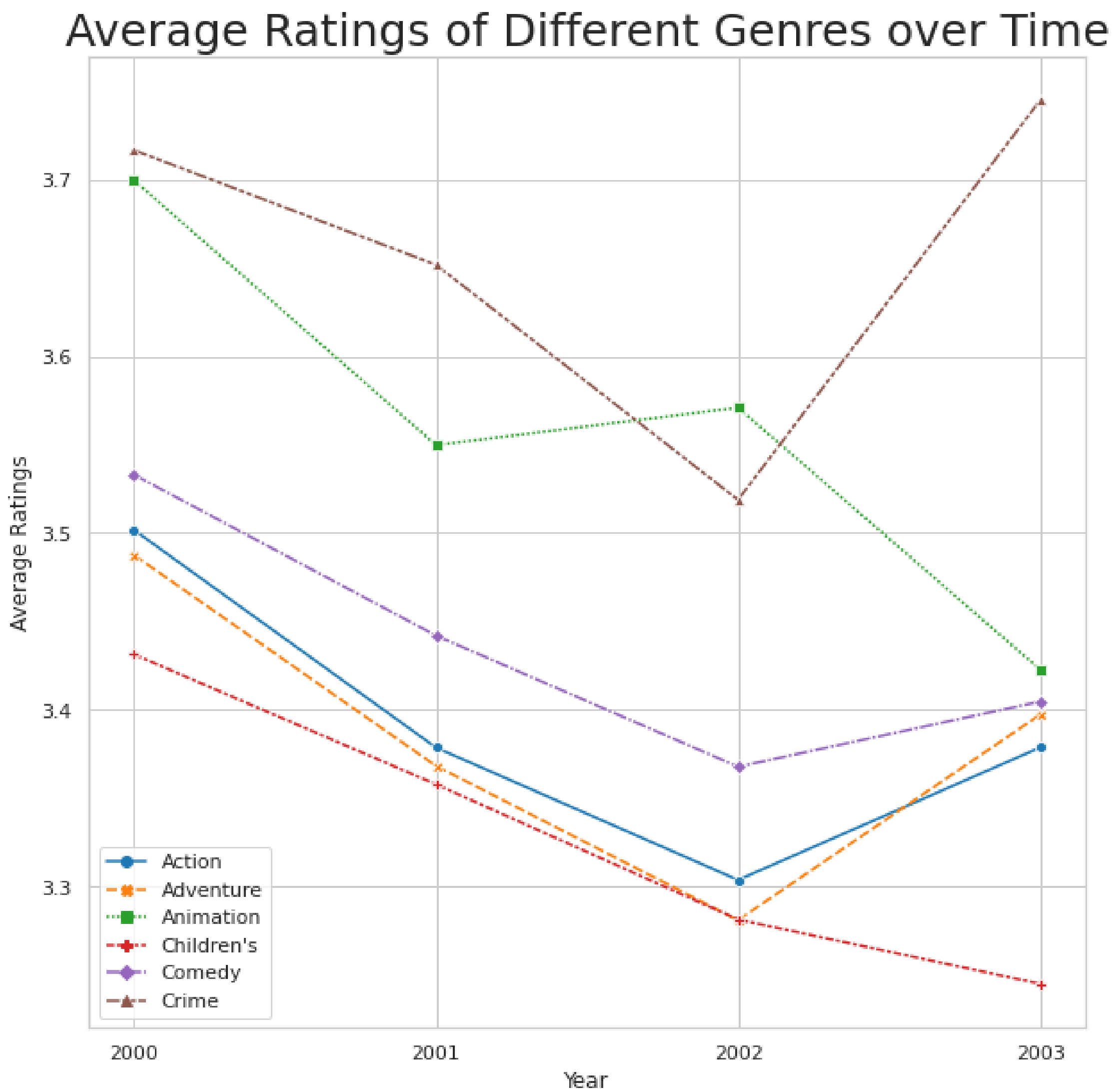
```
           Action  Adventure  Animation  ...  Thriller       War   Western
2000-04  3.525162   3.468177   3.680525  ...  3.607013  3.904092  3.489083
2000-05  3.520630   3.488710   3.715075  ...  3.634570  3.916632  3.614783
2000-06  3.525611   3.491458   3.640695  ...  3.644718  3.913909  3.654475
2000-07  3.516047   3.518291   3.712869  ...  3.620848  3.906446  3.655527
2000-08  3.518174   3.494874   3.727545  ...  3.585479  3.897484  3.707112
2000-09  3.501183   3.500575   3.737913  ...  3.594875  3.919865  3.686117
2000-10  3.526087   3.507561   3.722925  ...  3.595410  3.921401  3.684270
2000-11  3.473170   3.467142   3.665646  ...  3.544955  3.893238  3.594134
2000-12  3.501810   3.490637   3.733115  ...  3.560809  3.900706  3.613017
2001-01  3.456774   3.435806   3.627252  ...  3.514411  3.787671  3.626140
2001-02  3.415750   3.425261   3.650672  ...  3.539616  3.808370  3.477612
2001-03  3.319483   3.412478   3.516340  ...  3.484135  3.894737  3.585859
2001-04  3.317682   3.360153   3.600683  ...  3.384381  3.728571  3.545455
2001-05  3.258416   3.271084   2.995146  ...  3.302462  3.633621  3.910714
2001-06  3.418012   3.329480   3.369792  ...  3.524515  3.689362  3.473684
2001-07  3.310729   3.337778   3.505000  ...  3.402951  3.867220  3.552239
2001-08  3.324727   3.192982   3.655340  ...  3.407449  3.820513  3.218750
2001-09  3.415323   3.395425   3.675676  ...  3.487603  3.981818  3.706897
2001-10  3.191450   3.280142   3.677966  ...  3.434010  3.686567  3.487179
2001-11  3.439929   3.418251   3.543210  ...  3.634409  3.940171  3.914286
2001-12  3.389590   3.276873   3.484076  ...  3.488812  3.650224  3.826087
2002-01  3.470588   3.353659   3.524823  ...  3.598187  3.994624  3.472222
2002-02  3.366525   3.402174   3.446602  ...  3.443850  3.843750  3.620690
2002-03  3.052516   3.065421   3.221311  ...  3.188144  3.520000  3.509091
2002-04  3.384466   3.316716   4.000000  ...  3.398892  3.664430  3.645161
2002-05  3.410828   3.426036   3.489796  ...  3.693410  3.759615  3.500000
2002-06  3.307692   3.294118   3.365079  ...  3.377104  3.344828  3.761905
2002-07  3.197970   3.080645   3.688312  ...  3.299728  3.602041  3.363636
2002-08  3.081301   3.137324   3.416667  ...  3.251969  3.546218  3.617021
2002-09  3.299145   3.267717   3.428571  ...  3.285156  3.808219  3.950000
2002-10  3.544910   3.406977   3.466667  ...  3.481203  4.132075  3.928571
2002-11  3.374593   3.267176   3.661017  ...  3.508834  3.739496  3.125000
2002-12  3.285714   3.443548   4.051282  ...  3.418994  3.800000  3.592593
2003-01  3.491429   3.606557   3.530612  ...  3.466135  3.915789  3.631579
2003-02  3.255452   3.167665   3.312500  ...  3.400000  3.955056  3.472222

[35 rows x 18 columns]
```

Each plot for the genre timeseries is distributed into 3 parts with 6 genres plotted each.

[0]:

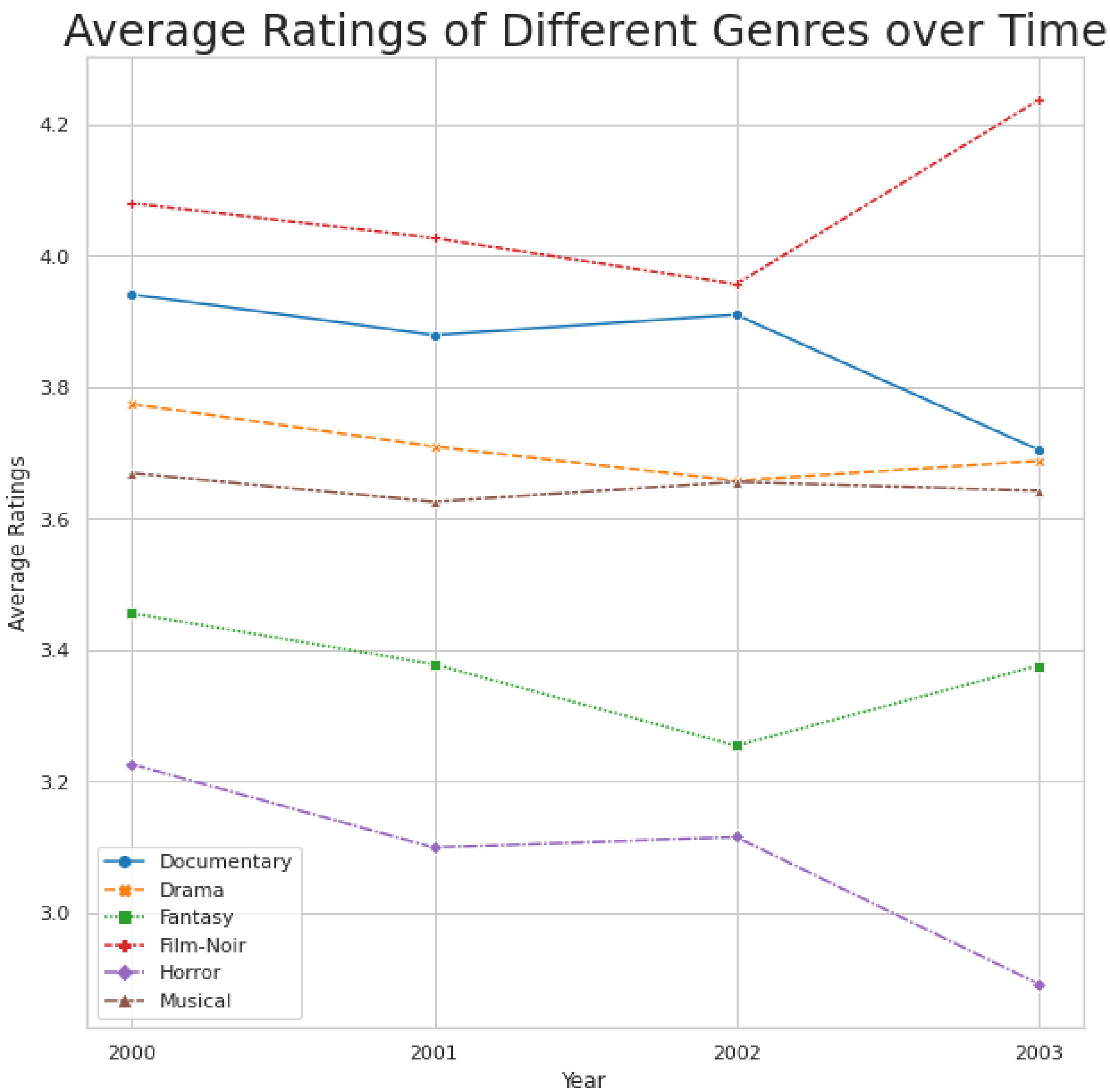
```
plt.figure(figsize=(10, 10))
sns.set(style="whitegrid")
ax = sns.lineplot(data=yearwise_genre_rating_dataframe.iloc[:,:6],␣
 ↪palette="tab10", markers=True)
```

```
ax.set_title('Average Ratings of Different Genres over Time',␣
 ↪fontdict={'fontsize': '25', 'fontweight' : '3'})
ax.set_xlabel('Year')
ax.set_ylabel('Average Ratings')
ax.xaxis.set_major_locator(ticker.MultipleLocator(1))
plt.show()
```



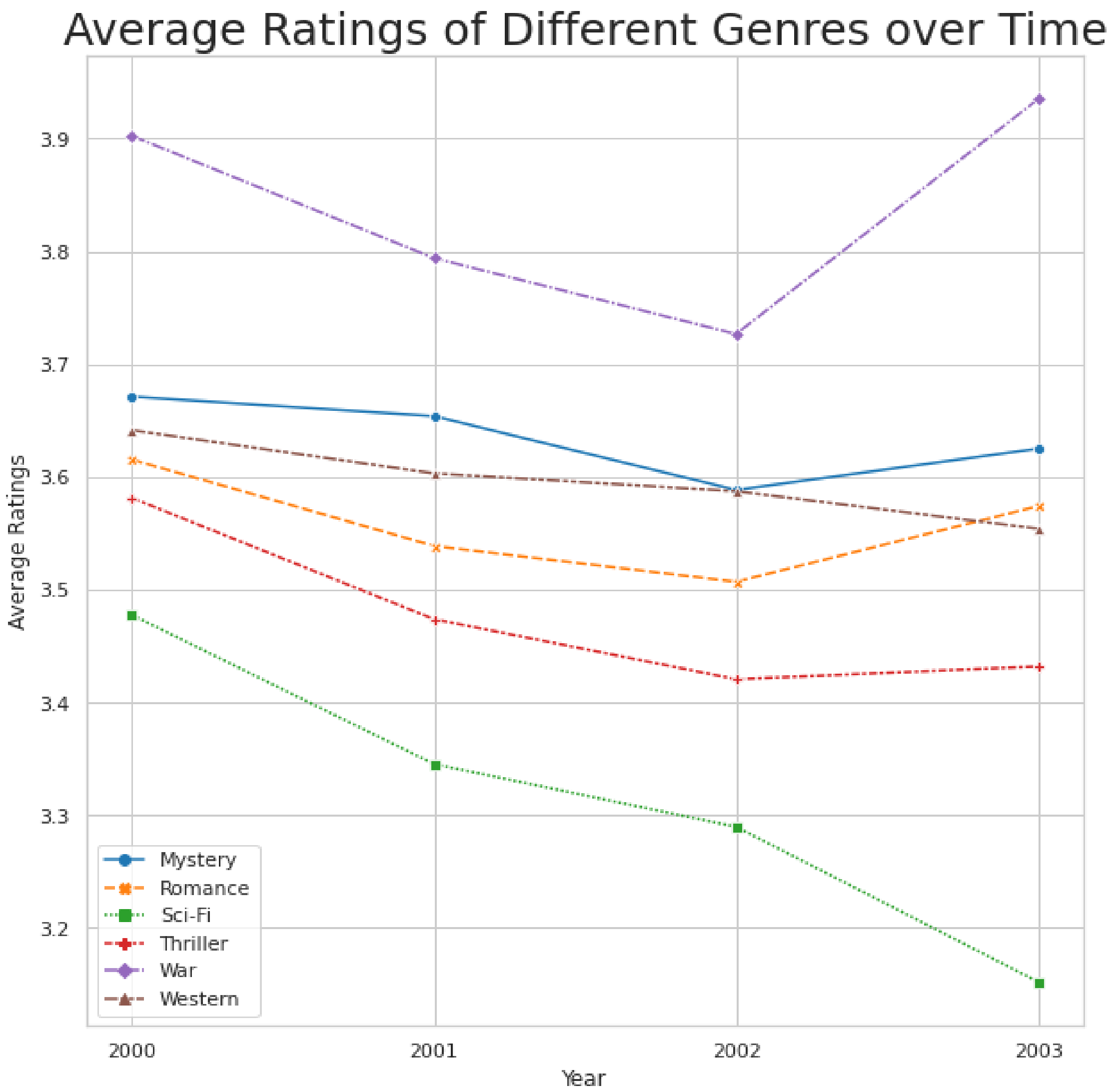
[0]:
```
plt.figure(figsize=(10, 10))
sns.set(style="whitegrid")
ax = sns.lineplot(data=yearwise_genre_rating_dataframe.iloc[:,6:12],␣
 ↪palette="tab10", markers=True)
ax.set_title('Average Ratings of Different Genres over Time',␣
 ↪fontdict={'fontsize': '25', 'fontweight' : '3'})
```

```
ax.set_xlabel('Year')
ax.set_ylabel('Average Ratings')
ax.xaxis.set_major_locator(ticker.MultipleLocator(1))
plt.show()
```

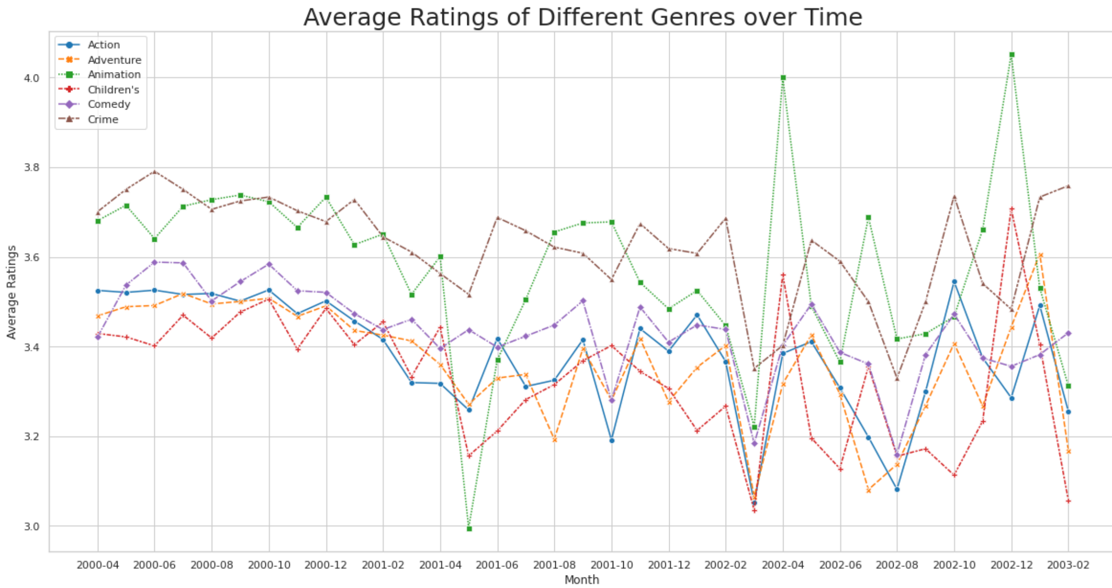


[0]:
```
plt.figure(figsize=(10, 10))
sns.set(style="whitegrid")
ax = sns.lineplot(data=yearwise_genre_rating_dataframe.iloc[:,12:],␣
 ↪palette="tab10", markers=True)
ax.set_title('Average Ratings of Different Genres over Time',␣
 ↪fontdict={'fontsize': '25', 'fontweight' : '3'})
ax.set_xlabel('Year')
ax.set_ylabel('Average Ratings')
```

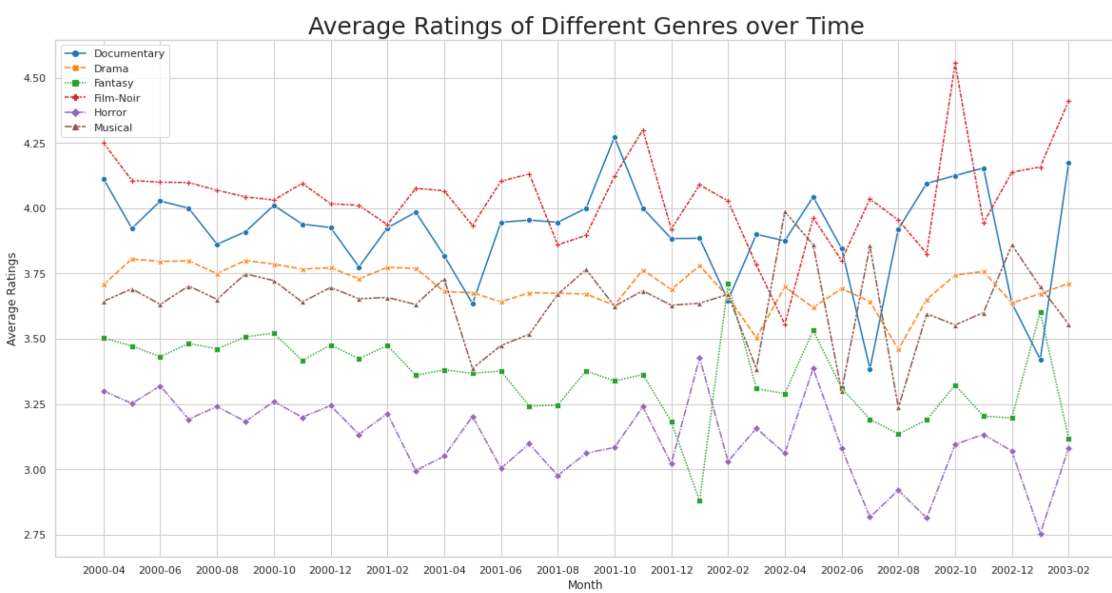
```
ax.xaxis.set_major_locator(ticker.MultipleLocator(1))
plt.show()
```



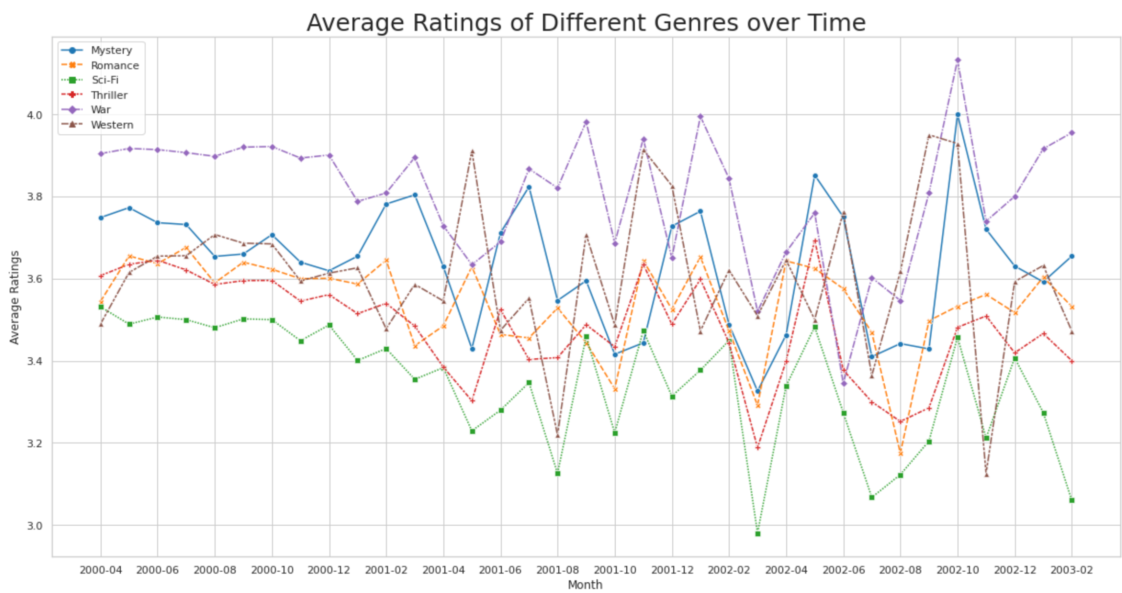
[0]:
```
plt.figure(figsize=(20, 10))
sns.set(style="whitegrid")
ax = sns.lineplot(data=monthwise_genre_rating_dataframe.iloc[:,:6],␣
↪palette="tab10", markers=True)
ax.set_title('Average Ratings of Different Genres over Time',␣
↪fontdict={'fontsize': '25', 'fontweight' : '3'})
ax.set_xlabel('Month')
ax.set_ylabel('Average Ratings')
ax.xaxis.set_major_locator(ticker.MultipleLocator(2))
plt.show()
```



[0]:
```
plt.figure(figsize=(20, 10))
sns.set(style="whitegrid")
ax = sns.lineplot(data=monthwise_genre_rating_dataframe.iloc[:,6:12],␣
 ↪palette="tab10", markers=True)
ax.set_title('Average Ratings of Different Genres over Time',␣
 ↪fontdict={'fontsize': '25', 'fontweight' : '3'})
ax.set_xlabel('Month')
ax.set_ylabel('Average Ratings')
ax.xaxis.set_major_locator(ticker.MultipleLocator(2))
plt.show()
```



[0]:
```
plt.figure(figsize=(20, 10))
sns.set(style="whitegrid")
ax = sns.lineplot(data=monthwise_genre_rating_dataframe.iloc[:,12:],
 ↪palette="tab10", markers=True)
ax.set_title('Average Ratings of Different Genres over Time',
 ↪fontdict={'fontsize': '25', 'fontweight' : '3'})
ax.set_xlabel('Month')
ax.set_ylabel('Average Ratings')
ax.xaxis.set_major_locator(ticker.MultipleLocator(2))
plt.show()
```



According to the above visualizations, we see that it is straightforward to be able to predict the user's favourite genre given certain information about the user. For now, we consider Age Group, Gender, Occupation and State.

[0]:
```
# Find the user's favourite genre and setting up the final dataset

user_list = list(users_data['UserID'])
user_list.insert(0,0)
df3=pd.DataFrame(user_list, columns=['UserID'])

for iter in all_genres_i:
    c=0
    groups = final_map.groupby('UserID')[iter].sum()
    groups= pd.DataFrame(data=groups)
```

```
    c=c+1
    df3.insert(c,iter,groups)

df3 = df3.iloc[1:]
df3['MaxGenre'] = df3[df3.columns.difference(['UserID'])].idxmax(axis=1)

df4 = df3.copy()
df4 = df4[['UserID','MaxGenre']]
df4 = pd.merge(df4,mapping, how="left", on='UserID' )
df4=df4[['UserID','MaxGenre','Gender','Age','Occupation','state']]
df4['state']=df4['state'].astype('category')
df4['state']=df4['state'].cat.codes
df4['Gender']=df4['Gender'].astype('category')
df4['Gender']=df4['Gender'].cat.codes
df4['MaxGenre']=df4['MaxGenre'].astype('category')
df4['MaxGenre']=df4['MaxGenre'].cat.codes
df4
```

[0]:

	UserID	MaxGenre	Gender	Age	Occupation	state
0	1	7	0	1	10	25
1	2	7	1	56	16	21
2	3	4	1	25	15	26
3	4	0	1	45	7	22
4	5	7	1	25	20	26
...	...	...	...	...	...	...
6035	6036	7	0	25	15	11
6036	6037	7	0	45	1	47
6037	6038	4	0	56	1	37
6038	6039	4	0	45	0	22
6039	6040	7	1	25	6	37

[6040 rows x 6 columns]

[0]:
```
# setting up the k-fold mechanism
cv = KFold(n_splits=10)
accuracies = list()
max_attributes = len(list(df4))
depth_range = range(1, max_attributes+1)
```

We now set up the Decision Tree on the data shown in the previous cells. We show the final accuracy of the Decision Tree.

[0]:
```
for depth in depth_range:
    fold_accuracy = []
    tree_model = DecisionTreeClassifier(max_depth = depth)

    print("Current max depth: ", depth, "\n")
```

```
    for train_fold, valid_fold in cv.split(df4):
        f_train = df4.loc[train_fold] # Extract train data with cv indices
        f_valid = df4.loc[valid_fold] # Extract valid data with cv indices

        # We fit the model with the fold train data
        model = tree_model.fit(X = f_train, y = f_train['MaxGenre'])

        # We calculate accuracy with the fold validation data
        valid_acc = model.score(X = f_valid, y = f_valid['MaxGenre'])
        fold_accuracy.append(valid_acc)

    avg = sum(fold_accuracy)/len(fold_accuracy)
    accuracies.append(avg)
    print("Accuracy per fold: ", fold_accuracy, "\n")
    print("Average accuracy: ", avg)
    print("=="*20)
    print("\n")
```

```
Current max depth:  1

Accuracy per fold:  [0.640728476821192, 0.6556291390728477, 0.7102649006622517,
0.6837748344370861, 0.6721854304635762, 0.6903973509933775, 0.6390728476821192,
0.7086092715231788, 0.7152317880794702, 0.6837748344370861]

Average accuracy:  0.6799668874172186
========================================


Current max depth:  2

Accuracy per fold:  [0.8774834437086093, 0.8576158940397351, 0.8857615894039735,
0.8890728476821192, 0.8874172185430463, 0.9188741721854304, 0.8956953642384106,
0.8923841059602649, 0.9105960264900662, 0.8990066225165563]

Average accuracy:  0.8913907284768212
========================================


Current max depth:  3

Accuracy per fold:  [0.9254966887417219, 0.9105960264900662, 0.9139072847682119,
0.9139072847682119, 0.9321192052980133, 0.9519867549668874, 0.9205298013245033,
0.9387417218543046, 0.9403973509933775, 0.9288079470198676]

Average accuracy:  0.9276490066225167
========================================
```

```
Current max depth:  4

Accuracy per fold:  [0.9701986754966887, 0.9784768211920529, 0.9751655629139073,
0.9801324503311258, 0.9834437086092715, 0.9933774834437086, 0.9801324503311258,
0.9884105960264901, 0.9735099337748344, 0.9867549668874173]

Average accuracy:  0.9809602649006622
========================================


Current max depth:  5

Accuracy per fold:  [0.9884105960264901, 0.9900662251655629, 0.9884105960264901,
0.9933774834437086, 0.9884105960264901, 0.9950331125827815, 0.9917218543046358,
0.9983443708609272, 0.9867549668874173, 0.9917218543046358]

Average accuracy:  0.991225165562914
========================================


Current max depth:  6

Accuracy per fold:  [1.0, 0.9950331125827815, 1.0, 0.9983443708609272,
0.9966887417218543, 1.0, 0.9966887417218543, 0.9983443708609272,
0.9966887417218543, 0.9983443708609272]

Average accuracy:  0.9980132450331126
========================================
```

We now show the relation between the max-depth of the decision tree and the accuracy for the same.

[0]:
```python
df = pd.DataFrame({"Max Depth": depth_range, "Average Accuracy": accuracies})
df = df[["Max Depth", "Average Accuracy"]]
print(df.to_string(index=False))
```

```
 Max Depth  Average Accuracy
         1          0.679967
         2          0.891391
         3          0.927649
         4          0.980960
         5          0.991225
         6          0.998013
```

We now show the implementation of Logistic Regression for the same problem.

[0]:
```
df5 = df4.copy()
df5 = df5[["MaxGenre"]]

# Preparing the logistic Regression Model
model = LogisticRegression(fit_intercept=True, max_iter=10000)

train_data = df4.values[:5040]
labels = df5[:5040]
eval_data = df4.values[1000:]
eval_labels = df5[1000:]

model.fit(train_data, labels)
eval_predictions = model.predict(eval_data)
```

```
/usr/local/lib/python3.6/dist-packages/sklearn/utils/validation.py:760:
DataConversionWarning: A column-vector y was passed when a 1d array was
expected. Please change the shape of y to (n_samples, ), for example using
ravel().
  y = column_or_1d(y, warn=True)
/usr/local/lib/python3.6/dist-packages/sklearn/linear_model/_logistic.py:940:
ConvergenceWarning: lbfgs failed to converge (status=1):
STOP: TOTAL NO. of ITERATIONS REACHED LIMIT.

Increase the number of iterations (max_iter) or scale the data as shown in:
    https://scikit-learn.org/stable/modules/preprocessing.html
Please also refer to the documentation for alternative solver options:
    https://scikit-learn.org/stable/modules/linear_model.html#logistic-
regression
  extra_warning_msg=_LOGISTIC_SOLVER_CONVERGENCE_MSG)
```

[0]:
```
print('Accuracy of the model on train data: {0}'.format(model.score(train_data,␣
 ↪labels)))
print('Accuracy of the model on eval data: {0}'.format(model.score(eval_data,␣
 ↪eval_labels)))
```

```
Accuracy of the model on train data: 0.8654761904761905
Accuracy of the model on eval data: 0.8724206349206349
```

We also demonstrate the accuracy of the SVC (Support Vector Classifier) model as provided by sklearn

[54]:
```
from sklearn.svm import SVC
clf = SVC(gamma='auto')

clf.fit(train_data, labels)

print('Accuracy of the SVC model on train data: {0}'.format(clf.
 ↪score(train_data, labels)))
```

```
print('Accuracy of the SVC model on eval data: {0}'.format(clf.score(eval_data,
↪eval_labels)))
```

```
/usr/local/lib/python3.6/dist-packages/sklearn/utils/validation.py:760:
DataConversionWarning: A column-vector y was passed when a 1d array was
expected. Please change the shape of y to (n_samples, ), for example using
ravel().
  y = column_or_1d(y, warn=True)
```

```
Accuracy of the model on train data: 0.9992063492063492
Accuracy of the model on eval data: 0.8807539682539682
```