Untitled1

June 22, 2021

1 Import

[ ]:
```
import os
from google.colab import drive
from argparse import Namespace
import numpy as np
import pandas as pd
import httpimport
import torch
import torch.optim as optim
from tqdm import tqdm_notebook, tqdm
import math
import statsmodels.formula.api as smf
import statsmodels.api as sm
from numpy import genfromtxt
import csv
import urllib.request
import seaborn as sns
import matplotlib.pyplot as plt
from matplotlib.pyplot import figure
from sklearn.preprocessing import StandardScaler
import matplotlib
from sklearn.linear_model import LinearRegression
from sklearn.linear_model import Ridge
from sklearn.linear_model import Lasso

import torch
import torch.nn as nn
import torch.optim as optim
import torch.nn.functional as F
from torch.utils.data import Dataset, DataLoader
from torch.autograd import Variable
import numpy as np
import pandas as pd
from sklearn.metrics import mean_squared_error
from sklearn.model_selection import train_test_split
from torch.utils.data import TensorDataset, DataLoader
```

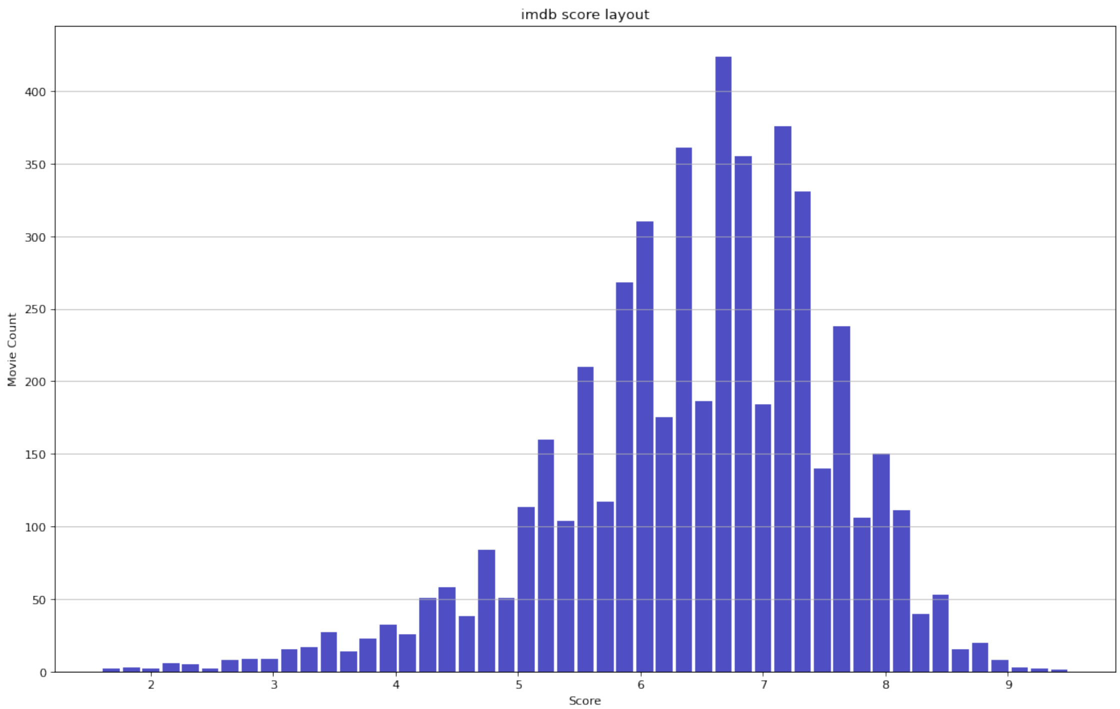
```
from tqdm import tqdm
from datetime import datetime
import torch.nn.functional as F
import random
import statsmodels.api as sm
import sklearn
from sklearn.metrics import mean_squared_error, r2_score
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import roc_auc_score, accuracy_score, precision_score,␣
 ↪recall_score
from sklearn.metrics import hamming_loss
from sklearn.linear_model import LogisticRegression
from sklearn.linear_model import SGDClassifier
from sklearn.naive_bayes import GaussianNB
```

#Data Preprocessing and EDA

[ ]:
```
url = "https://github.com/sundeepblue/movie_rating_prediction/raw/master/
 ↪movie_metadata.csv"
data = pd.read_csv(url)
data.head()
```

[ ]:
```
   color      director_name  …  aspect_ratio  movie_facebook_likes
0  Color      James Cameron  …          1.78                 33000
1  Color     Gore Verbinski  …          2.35                     0
2  Color         Sam Mendes  …          2.35                 85000
3  Color  Christopher Nolan  …          2.35                164000
4    NaN        Doug Walker  …           NaN                     0

[5 rows x 28 columns]
```

[ ]:
```
figure(figsize=(16, 10), dpi=80)
n, bins, patches = plt.hist(x=data['imdb_score'], bins='auto',␣
 ↪color='#0504aa',alpha=0.7, rwidth=0.85)
plt.grid(axis='y', alpha=0.75)
plt.xlabel('Score')
plt.ylabel('Movie Count')
plt.title('imdb score layout')
maxfreq = n.max()
```



We want to make a prediction for imdb scores. And as we know, the quality of movies is highly related to the person who direct them. Let's make a breif view of our producers.

[ ]:
```
director = data.groupby('director_name').size().sort_values(ascending=False).
  ↪head(20)
director
```

[ ]:
```
director_name
Steven Spielberg      26
Woody Allen           22
Martin Scorsese       20
Clint Eastwood        20
Ridley Scott          17
Tim Burton            16
Spike Lee             16
Steven Soderbergh     16
Renny Harlin          15
Oliver Stone          14
Sam Raimi             13
Michael Bay           13
Barry Levinson        13
Robert Zemeckis       13
John Carpenter        13
Joel Schumacher       13
```

```
Ron Howard           13
Robert Rodriguez     13
Richard Donner       12
Shawn Levy           12
dtype: int64
```

Steven Spielberg is a reputated director, so as Woody Allen. I really appreciate their masterpieces during my childhood. I believe lots of audiences facinate their work as well. How about the average scores of their works?

[ ]:
```
name_list = director.index.tolist()
average = {}
for i in name_list:
  sum = data['imdb_score'][data['director_name'] == i].sum()
  average[i] = sum/director[i]
sorted(average.items(), key=lambda x: -x[1])
```

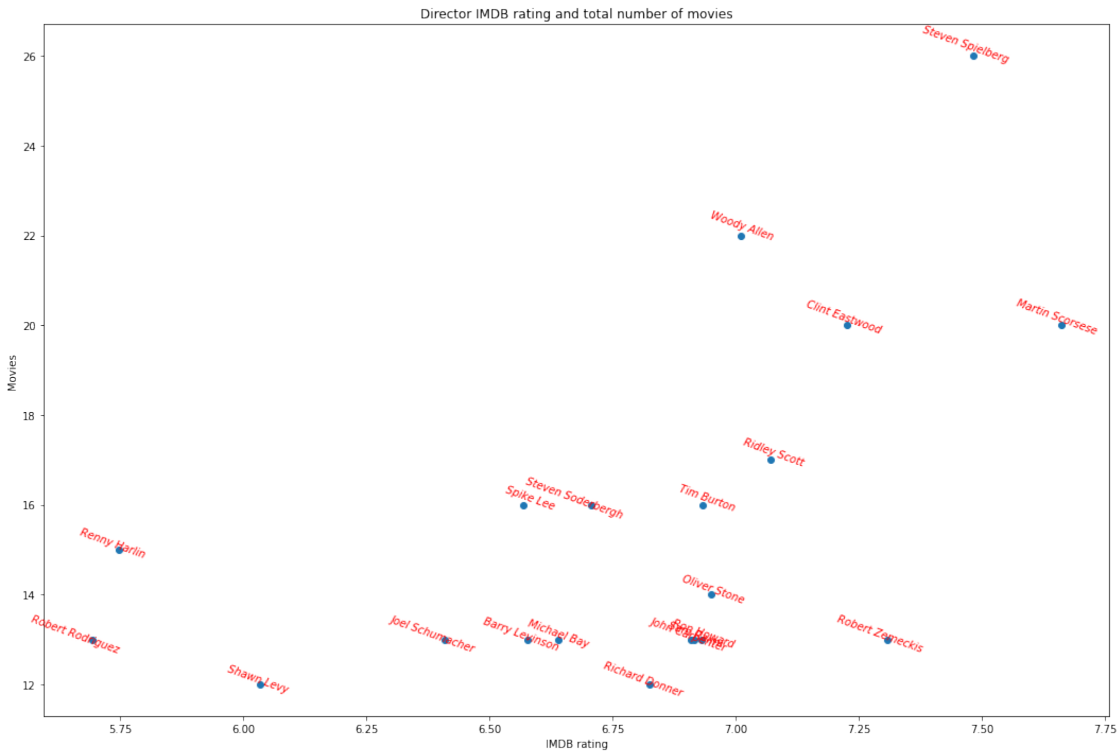
[ ]:
```
[('Martin Scorsese', 7.659999999999999),
 ('Steven Spielberg', 7.480769230769232),
 ('Robert Zemeckis', 7.307692307692308),
 ('Clint Eastwood', 7.225),
 ('Ridley Scott', 7.070588235294117),
 ('Woody Allen', 7.00909090909091),
 ('Oliver Stone', 6.95),
 ('Tim Burton', 6.93125),
 ('Ron Howard', 6.930769230769231),
 ('John Carpenter', 6.915384615384615),
 ('Sam Raimi', 6.907692307692307),
 ('Richard Donner', 6.824999999999999),
 ('Steven Soderbergh', 6.70625),
 ('Michael Bay', 6.638461538461538),
 ('Barry Levinson', 6.576923076923076),
 ('Spike Lee', 6.56875),
 ('Joel Schumacher', 6.407692307692307),
 ('Shawn Levy', 6.033333333333334),
 ('Renny Harlin', 5.746666666666666),
 ('Robert Rodriguez', 5.6923076923076925)]
```

Christopher Nolan, my favorite sci-film director has the best overall imdb score. His moive is not just so good, or brilliant. It's beyond that. I have seen Inception for multiple times. Maybe I should do one more time. Ok, so we should agree that talented directors and casts will make a moive more entertaining and quality, which always result in a better rating. And we should consider that in our model.

[ ]:
```
director_score = pd.DataFrame(average.items(),columns=['Director','Score'])
director_score['Movies'] = director.to_list()
director_score
```

[ ]:
```
             Director     Score  Movies
0    Steven Spielberg  7.480769      26
1         Woody Allen  7.009091      22
2     Martin Scorsese  7.660000      20
3      Clint Eastwood  7.225000      20
4        Ridley Scott  7.070588      17
5          Tim Burton  6.931250      16
6           Spike Lee  6.568750      16
7   Steven Soderbergh  6.706250      16
8        Renny Harlin  5.746667      15
9        Oliver Stone  6.950000      14
10          Sam Raimi  6.907692      13
11        Michael Bay  6.638462      13
12     Barry Levinson  6.576923      13
13    Robert Zemeckis  7.307692      13
14     John Carpenter  6.915385      13
15    Joel Schumacher  6.407692      13
16         Ron Howard  6.930769      13
17   Robert Rodriguez  5.692308      13
18     Richard Donner  6.825000      12
19         Shawn Levy  6.033333      12
```

[ ]:
```python
plt.rcParams['axes.unicode_minus']=False
director=director_score['Director']
score=director_score['Score']
movies=director_score['Movies']

fig=plt.figure(figsize=(18,12))
ax=plt.subplot(1,1,1)
ax.scatter(score,movies)
ax.set_title("Director IMDB rating and total number of movies")
ax.set_xlabel("IMDB rating")
ax.set_ylabel("Movies")
for i in range(len(movies)):
    ax.text(score[i]*1.01, movies[i]*1.01, director[i],
            fontsize=10, color = "r", style = "italic", weight = "light",
            verticalalignment='center',␣
 ↪horizontalalignment='right',rotation=-20)
plt.show()
```



As we can see, directors with good reputation also have produced lots of movies. Maybe audiences love those famous figures and lead to celebrity effect.

[ ]:
```
director_score = sorted(average.items(), key=lambda x: -x[1])
type(director_score)
```

[ ]:
```
list
```

[ ]:
```
data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 5043 entries, 0 to 5042
Data columns (total 28 columns):
 #   Column                     Non-Null Count  Dtype
---  ------                     --------------  -----
 0   color                      5024 non-null   object
 1   director_name              4939 non-null   object
 2   num_critic_for_reviews     4993 non-null   float64
 3   duration                   5028 non-null   float64
 4   director_facebook_likes    4939 non-null   float64
 5   actor_3_facebook_likes     5020 non-null   float64
 6   actor_2_name               5030 non-null   object
 7   actor_1_facebook_likes     5036 non-null   float64
 8   gross                      4159 non-null   float64
```

```
 9   genres                     5043 non-null   object
 10  actor_1_name               5036 non-null   object
 11  movie_title                5043 non-null   object
 12  num_voted_users            5043 non-null   int64
 13  cast_total_facebook_likes  5043 non-null   int64
 14  actor_3_name               5020 non-null   object
 15  facenumber_in_poster       5030 non-null   float64
 16  plot_keywords              4890 non-null   object
 17  movie_imdb_link            5043 non-null   object
 18  num_user_for_reviews       5022 non-null   float64
 19  language                   5031 non-null   object
 20  country                    5038 non-null   object
 21  content_rating             4740 non-null   object
 22  budget                     4551 non-null   float64
 23  title_year                 4935 non-null   float64
 24  actor_2_facebook_likes     5030 non-null   float64
 25  imdb_score                 5043 non-null   float64
 26  aspect_ratio               4714 non-null   float64
 27  movie_facebook_likes       5043 non-null   int64
dtypes: float64(13), int64(3), object(12)
memory usage: 1.1+ MB
```

This dataset contains lots of information, including 9 characteristic variables, 15 numeric variables and 2 categorical variables. Before we processing feature engineering, I want to check data integrity so that missing values and outliers won't affect our prediction result. The first step is to find out how many missing values we have here.

[ ]:
```
data.isna().sum()
```

[ ]:
```
color                         19
director_name                104
num_critic_for_reviews        50
duration                      15
director_facebook_likes      104
actor_3_facebook_likes        23
actor_2_name                  13
actor_1_facebook_likes         7
gross                        884
genres                         0
actor_1_name                   7
movie_title                    0
num_voted_users                0
cast_total_facebook_likes      0
actor_3_name                  23
facenumber_in_poster          13
plot_keywords                153
movie_imdb_link                0
num_user_for_reviews          21
```

```
language                       12
country                         5
content_rating                303
budget                        492
title_year                    108
actor_2_facebook_likes         13
imdb_score                      0
aspect_ratio                  329
movie_facebook_likes            0
dtype: int64
```

That's a lot missing here. It is unpropriate if we processing the raw data without cleaning them out. I would assume that those directors and actors who do not have "facebook likes" is because they do not have a official facebook account or is not avaliable currently. I would fill them as 0 to make it works.

[ ]:
```
data['director_facebook_likes'] = data['director_facebook_likes'].fillna(0)
data['actor_1_facebook_likes'] = data['actor_1_facebook_likes'].fillna(0)
data['actor_2_facebook_likes'] = data['actor_2_facebook_likes'].fillna(0)
data['actor_3_facebook_likes'] = data['actor_3_facebook_likes'].fillna(0)
```

Face number in movies' poster should not affect review scores, as well as their imdb links, the number of reviews and aspect ratio. We have to admit that movie title might have positive or negative influence here. For instance, rating of movies in franchise series are affected by their precessors. Audiences always expect more on great movies' successors. On another hand, moives which named after a famous charactor, like spider-man or Donald Duck, will give their audiences a special impression. Still, I will delete these columns because their influence in this dataset is minimal and this will save us some time.

[ ]:
```
data_wo_useless_attributes = data.
 ↪drop(columns=['color','movie_imdb_link','num_user_for_reviews','aspect_ratio','movie_title'
```

[ ]:
```
data_wo_useless_attributes = data_wo_useless_attributes.
 ↪drop(columns=['facenumber_in_poster'])
```

Drop the na rows for numeric varibles. It is not approriate if we just fill them with means or median here. As we have enough samples here and na rows of numeric varibles are no more than 400 rows, I would say it is safer to focus on rows without missing values.

[ ]:
```
data_numeric_cleaned = data_wo_useless_attributes.
 ↪dropna(subset=['num_critic_for_reviews','duration','director_name','title_year','gross','bu
```

[ ]:
```
data_categoric_cleaned = data_numeric_cleaned.
 ↪dropna(subset=['director_name','actor_1_name','actor_2_name','actor_3_name','content_rating
```

[ ]:
```
data_categoric_cleaned.isna().sum()
```

```
[ ]: director_name                0
     num_critic_for_reviews       0
     duration                     0
     director_facebook_likes      0
     actor_3_facebook_likes       0
     actor_2_name                 0
     actor_1_facebook_likes       0
     gross                        0
     genres                       0
     actor_1_name                 0
     num_voted_users              0
     cast_total_facebook_likes    0
     actor_3_name                 0
     content_rating               0
     budget                       0
     title_year                   0
     actor_2_facebook_likes       0
     imdb_score                   0
     movie_facebook_likes         0
     dtype: int64
```

Index needs to be resets here to make sure following step will work well.

```
[ ]: data_categoric_cleaned = data_categoric_cleaned.reset_index(drop=True)
```

```
[ ]: data_categoric_cleaned.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 3833 entries, 0 to 3832
Data columns (total 19 columns):
 #   Column                     Non-Null Count  Dtype
---  ------                     --------------  -----
 0   director_name              3833 non-null   object
 1   num_critic_for_reviews     3833 non-null   float64
 2   duration                   3833 non-null   float64
 3   director_facebook_likes    3833 non-null   float64
 4   actor_3_facebook_likes     3833 non-null   float64
 5   actor_2_name               3833 non-null   object
 6   actor_1_facebook_likes     3833 non-null   float64
 7   gross                      3833 non-null   float64
 8   genres                     3833 non-null   object
 9   actor_1_name               3833 non-null   object
 10  num_voted_users            3833 non-null   int64
 11  cast_total_facebook_likes  3833 non-null   int64
 12  actor_3_name               3833 non-null   object
 13  content_rating             3833 non-null   object
 14  budget                     3833 non-null   float64
 15  title_year                 3833 non-null   float64
```

```
 16  actor_2_facebook_likes     3833 non-null   float64
 17  imdb_score                 3833 non-null   float64
 18  movie_facebook_likes       3833 non-null   int64
dtypes: float64(10), int64(3), object(6)
memory usage: 569.1+ KB
```

2 Characteristic Variables and categorical variables

Genres is a little difficult to deal with. It has been put into one column and separate by "|". I will divide them into several columns using one hot coding. This is the only variable I want to use one hot coding since the labels is not so many.

[ ]:
```
data_categoric_cleaned['genres'].describe()
```

[ ]:
```
count                     3833
unique                     751
top       Comedy|Drama|Romance
freq                       150
Name: genres, dtype: object
```

I create a list to store all genres values.

[ ]:
```
genres = []
for i in range(len(data_categoric_cleaned)):
  temp = data_categoric_cleaned['genres'][i].split('|')
  for n in range(len(temp)):
    if temp[n] not in genres:
      genres.append(temp[n])
```

Create new empty columns for genres.

[ ]:
```
for i in genres:
  temp = [0]*len(data_categoric_cleaned)
  col_name = i
  data_categoric_cleaned[col_name] = temp
```

One-hot coding and we have genres attributes separated.

[ ]:
```
for i in range(len(data_categoric_cleaned)):
  temp = data_categoric_cleaned['genres'][i].split('|')
  for n in range(len(temp)):
    data_categoric_cleaned[temp[n]][i] = 1
```

```
/usr/local/lib/python3.7/dist-packages/ipykernel_launcher.py:4:
SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame

See the caveats in the documentation: https://pandas.pydata.org/pandas-
```

```
docs/stable/user_guide/indexing.html#returning-a-view-versus-a-copy
  after removing the cwd from sys.path.
```

[ ]:
```
data_categoric_cleaned[genres]
```

[ ]:
```
      Action  Adventure  Fantasy  …  Music  Documentary  Film-Noir
0          1          1        1  …      0            0          0
1          1          1        1  …      0            0          0
2          1          1        0  …      0            0          0
3          1          0        0  …      0            0          0
4          1          1        0  …      0            0          0
…        …        …      …   …     …        …
3828       0          0        0  …      0            0          0
3829       0          0        0  …      0            0          0
3830       1          0        0  …      0            0          0
3831       0          0        0  …      0            0          0
3832       0          0        0  …      0            1          0

[3833 rows x 22 columns]
```

Drop the original genres column

[ ]:
```
data_categoric_cleaned = data_categoric_cleaned.drop(columns=['genres'])
data_ready = data_categoric_cleaned
```

#One hot encoding for content rating

[ ]:
```
content_rating = pd.get_dummies(data_ready.content_rating,␣
 ↪prefix='content_rating')
content_rating
```

[ ]:
```
      content_rating_Approved  …  content_rating_X
0                           0  …                 0
1                           0  …                 0
2                           0  …                 0
3                           0  …                 0
4                           0  …                 0
…                        …   …             …
3828                        0  …                 0
3829                        0  …                 0
3830                        0  …                 0
3831                        0  …                 0
3832                        0  …                 0

[3833 rows x 12 columns]
```

[ ]:
```
data_ready= data_ready.drop('content_rating',axis = 1)
data_ready = data_ready.join(content_rating)
```

```
data_ready
```

[ ]:
```
            director_name  …  content_rating_X
0           James Cameron  …                 0
1          Gore Verbinski  …                 0
2              Sam Mendes  …                 0
3       Christopher Nolan  …                 0
4          Andrew Stanton  …                 0
…                     …  …               …
3828        Shane Carruth  …                 0
3829      Neill Dela Llana  …                 0
3830    Robert Rodriguez  …                 0
3831         Edward Burns  …                 0
3832             Jon Gunn  …                 0

[3833 rows x 51 columns]
```

#Directors and Actors

As I mentioned before, directors and actors have hugh influence on movies' quality. We surely don't want to drop these variables. But we have thounds of labels and one-hot coding won't going to make it. As a result, I will replace their name by average imdb score of their works. And this number would represent their influence on movies' quality. However, it happens when an actor only starred in one movie and it has a pretty high rating, while he is not acutally famous and make contribute to that fancy score. Thus, I will only consider the popular ones, that is, Top 50 for directors and Top 150 for actors. Other people would only get a overall average imdb score.

[ ]:
```
director = data_ready.groupby('director_name').size().
↪sort_values(ascending=False).head(50)
name_list = director.index.tolist()
average = {}
for i in name_list:
  sum = data_ready['imdb_score'][data_ready['director_name'] == i].sum()
  average[i] = sum/director[i]
imdb_average = data_ready['imdb_score'].mean()
score_column = [imdb_average]*len(data_ready)
data_ready['director_score'] = score_column
for i in range(len(data_ready)):
  if data_ready['director_name'][i] in name_list:
    data_ready['director_score'][i] = average[data_ready['director_name'][i]]
```

```
/usr/local/lib/python3.7/dist-packages/ipykernel_launcher.py:12:
SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame

See the caveats in the documentation: https://pandas.pydata.org/pandas-
docs/stable/user_guide/indexing.html#returning-a-view-versus-a-copy
  if sys.path[0] == '':
```

[ ]:
```python
df1 = data_ready[['actor_1_name','imdb_score']]
df2 = data_ready[['actor_2_name','imdb_score']]
df3 = data_ready[['actor_3_name','imdb_score']]
df1.columns = ['name','imdb_score']
df2.columns = ['name','imdb_score']
df3.columns = ['name','imdb_score']

actor_concat = pd.concat([df1,df2,df3],axis = 0)
actor = actor_concat.groupby('name').size().sort_values(ascending=False).
 ↪head(150)
actor
```

[ ]:
```
name
Robert De Niro          47
Morgan Freeman          44
Johnny Depp             39
Bruce Willis            39
Matt Damon              35
                        ..
David Oyelowo           11
Kristin Scott Thomas    11
Tom Hardy               11
Nathan Lane             11
Romany Malco            10
Length: 150, dtype: int64
```

[ ]:
```python
name_list = actor.index.tolist()
average = {}
for i in name_list:
  sum = actor_concat['imdb_score'][actor_concat['name'] == i].sum()
  average[i] = sum/actor[i]
imdb_average = data_ready['imdb_score'].mean()
score_column = [imdb_average]*len(data_ready)
data_ready['actor_1_score'] = score_column
data_ready['actor_2_score'] = score_column
data_ready['actor_3_score'] = score_column
for i in range(len(data_ready)):
  if data_ready['actor_1_name'][i] in name_list:
    data_ready['actor_1_score'][i] = average[data_ready['actor_1_name'][i]]
for i in range(len(data_ready)):
  if data_ready['actor_2_name'][i] in name_list:
    data_ready['actor_2_score'][i] = average[data_ready['actor_2_name'][i]]
for i in range(len(data_ready)):
  if data_ready['actor_3_name'][i] in name_list:
    data_ready['actor_3_score'][i] = average[data_ready['actor_3_name'][i]]
data_finished = data_ready.
 ↪drop(columns=['director_name','actor_1_name','actor_2_name','actor_3_name'])
```

```
/usr/local/lib/python3.7/dist-packages/ipykernel_launcher.py:13:
SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame

See the caveats in the documentation: https://pandas.pydata.org/pandas-
docs/stable/user_guide/indexing.html#returning-a-view-versus-a-copy
  del sys.path[0]
/usr/local/lib/python3.7/dist-packages/ipykernel_launcher.py:16:
SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame

See the caveats in the documentation: https://pandas.pydata.org/pandas-
docs/stable/user_guide/indexing.html#returning-a-view-versus-a-copy
  app.launch_new_instance()
/usr/local/lib/python3.7/dist-packages/ipykernel_launcher.py:19:
SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame

See the caveats in the documentation: https://pandas.pydata.org/pandas-
docs/stable/user_guide/indexing.html#returning-a-view-versus-a-copy
```

[ ]:
```python
data_finished
```

[ ]:
```
      num_critic_for_reviews  duration  …  actor_2_score  actor_3_score
0                      723.0     178.0  …       6.459144       6.459144
1                      302.0     169.0  …       6.459144       6.459144
2                      602.0     148.0  …       6.459144       6.459144
3                      813.0     164.0  …       7.266667       7.055556
4                      462.0     132.0  …       6.459144       6.459144
…                          …       …  …             …             …
3828                   143.0      77.0  …       6.459144       6.459144
3829                    35.0      80.0  …       6.459144       6.459144
3830                    56.0      81.0  …       6.459144       6.459144
3831                    14.0      95.0  …       6.459144       6.459144
3832                    43.0      90.0  …       6.459144       6.459144

[3833 rows x 51 columns]
```

3 Dataset Split

[ ]:
```python
df = data_finished.sample(n = len(data_finished), random_state = 1)
df_data = df.reset_index(drop = True)
df_test=df_data.sample(frac=0.30,random_state=42)
```

[ ]:
```python
df_train = df_data.drop(df_test.index)
```

[ ]:
```
col_to_use = [c for c in list(df_train.columns) if c != 'imdb_score']
print('Number of attributes:', len(col_to_use))
```

```
Number of attributes: 50
```

[ ]:
```
X_train = df_train[col_to_use].values
X_test = df_test[col_to_use].values
y_train = df_train['imdb_score'].values
y_test = df_test['imdb_score'].values
print('Training shapes:',X_train.shape, y_train.shape)
print('Testing shapes:',X_test.shape, y_test.shape)
```

```
Training shapes: (2683, 50) (2683,)
Testing shapes: (1150, 50) (1150,)
```

#Simple Multiple Linear Regression

In this section, I build a very simple linear regression model as baseline.

[ ]:
```
y_train_a = y_train[:, np.newaxis]
model = LinearRegression()
model.fit(X_train, y_train_a)
predicts = model.predict(X_train)
R2 = model.score(X_train, y_train_a)
print('R2 = %.3f' % R2)
coef = model.coef_
intercept = model.intercept_
```

```
R2 = 0.510
```

Print out the coefficients

[ ]:
```
print(model.coef_, model.intercept_)
```

```
[[ 2.70883066e-03  3.55507208e-03 -5.59449149e-06  2.06409374e-05
   3.38771744e-05 -7.55508897e-10  2.40908781e-06 -3.25872946e-05
   2.80442009e-11 -2.74853818e-02  2.70217136e-05 -5.56086774e-07
  -2.15239587e-01 -3.61999366e-02 -1.04619311e-01 -1.26337145e-01
  -1.47651091e-01 -4.67495234e-02  8.27405488e-01 -1.25923585e-01
  -1.28848276e-01  4.21869143e-02  2.11041090e-02 -1.15995904e-01
   4.46069174e-01  8.36354779e-02  1.29653834e-01  6.68189254e-02
  -4.09519883e-01  1.00665257e-01  1.51313349e-01  1.20464296e-02
   9.68588259e-01 -7.48238020e-01  9.31309005e-04 -1.47046577e-01
  -5.97934228e-02  3.50160474e-02 -1.88684141e-01  5.03294612e-01
  -3.46058594e-02 -1.44973975e-01 -7.33025727e-01  4.76272706e-02
   3.25879720e-01  3.95380744e-01  1.43622546e-01  1.79400114e-01
   2.87293649e-01  1.06431000e-01]] [55.82515277]
```

[ ]:
```
LR_result = model.predict(X_test)
```

#Ridge and LASSO Regression

In order to prevent over-fitting of the model, we often need to add regularization items when building a linear model, generally there are L1 regularization and L2 regularization. Ridge regression(L2) and Lasso(L1) can prevet overfitting. Ridge regression reduces the regression coefficients without abandoning any feature, making the model relatively stable, but compared with Lasso regression, this will leave a lot of model features and poor model interpretation. The regularization term of ridge regression has a constant coefficient alpha to adjust the weight of the mean square error term and the regularization term of the loss function. Larger alpha means more penalty. I set two initial alpha to find their difference.

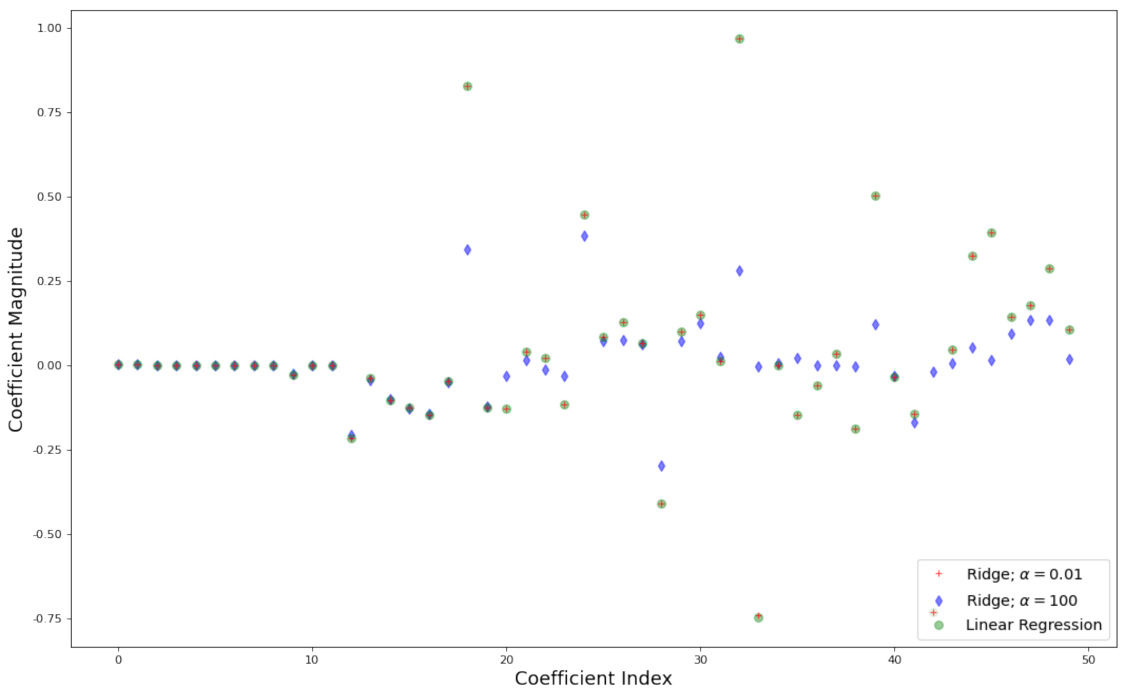
[ ]:
```
lr = LinearRegression()
lr.fit(X_train, y_train)

rr = Ridge(alpha=0.01)
rr.fit(X_train, y_train)

rr100 = Ridge(alpha=100)
rr100.fit(X_train, y_train)

train_score=lr.score(X_train, y_train)
test_score=lr.score(X_test, y_test)

Ridge_train_score = rr.score(X_train,y_train)
Ridge_test_score = rr.score(X_test,y_test)

Ridge_train_score100 = rr100.score(X_train,y_train)
Ridge_test_score100 = rr100.score(X_test,y_test)

print ("linear regression train score:", train_score)
print ("linear regression test score:", test_score)
print ("ridge regression train score low alpha:", Ridge_train_score)
print ("ridge regression test score low alpha:", Ridge_test_score)
print ("ridge regression train score high alpha:", Ridge_train_score100)
print ("ridge regression test score high alpha:", Ridge_test_score100)
```

```
linear regression train score: 0.5098707388489385
linear regression test score: 0.4619513375550573
ridge regression train score low alpha: 0.5098707081208159
ridge regression test score low alpha: 0.46195844086670446
ridge regression train score high alpha: 0.49196034919115267
ridge regression test score high alpha: 0.4544551529509967
```

```
/usr/local/lib/python3.7/dist-packages/sklearn/linear_model/_ridge.py:148:
LinAlgWarning: Ill-conditioned matrix (rcond=2.16297e-22): result may not be
accurate.
  overwrite_a=True).T
/usr/local/lib/python3.7/dist-packages/sklearn/linear_model/_ridge.py:148:
LinAlgWarning: Ill-conditioned matrix (rcond=1.66749e-18): result may not be
```

accurate.
  overwrite_a=True).T

```
[ ]: figure(figsize=(16, 10), dpi=80)
plt.plot(rr.coef_,alpha=0.
 ↪7,linestyle='none',marker='+',markersize=5,color='red',label=r'Ridge;␣
 ↪$\alpha = 0.01$',zorder=7)
plt.plot(rr100.coef_,alpha=0.
 ↪5,linestyle='none',marker='d',markersize=6,color='blue',label=r'Ridge;␣
 ↪$\alpha = 100$')
plt.plot(lr.coef_,alpha=0.
 ↪4,linestyle='none',marker='o',markersize=7,color='green',label='Linear␣
 ↪Regression')

plt.xlabel('Coefficient Index',fontsize=16)
plt.ylabel('Coefficient Magnitude',fontsize=16)
plt.legend(fontsize=13,loc=4)
plt.show()
```



As we can see, when alpha = 0.01, the coefficient is pretty close to linear regression. Lots of coefficient remain the same. We are still going to have much overfitting. To save some time, I evade creating a loop to find the best alpha, but rather using sklearn to calculate it at once.

```
[ ]: from sklearn.linear_model import RidgeCV
ridgecv = RidgeCV(alphas=[0.01, 0.1, 0.5, 1, 5, 7, 10, 30,100, 200])
```

```
ridgecv.fit(X_train, y_train)
print("Best alpha should be:" + str(ridgecv.alpha_))
```

```
Best alpha should be:5.0
```

Ok, alpha = 5. Fit the model again and we will make comparision later.

[ ]:
```
rr5 = Ridge(alpha=5)
rr5.fit(X_train, y_train)
rr_result = rr5.predict(X_test)
```

```
/usr/local/lib/python3.7/dist-packages/sklearn/linear_model/_ridge.py:148:
LinAlgWarning: Ill-conditioned matrix (rcond=1.02748e-19): result may not be
accurate.
  overwrite_a=True).T
```

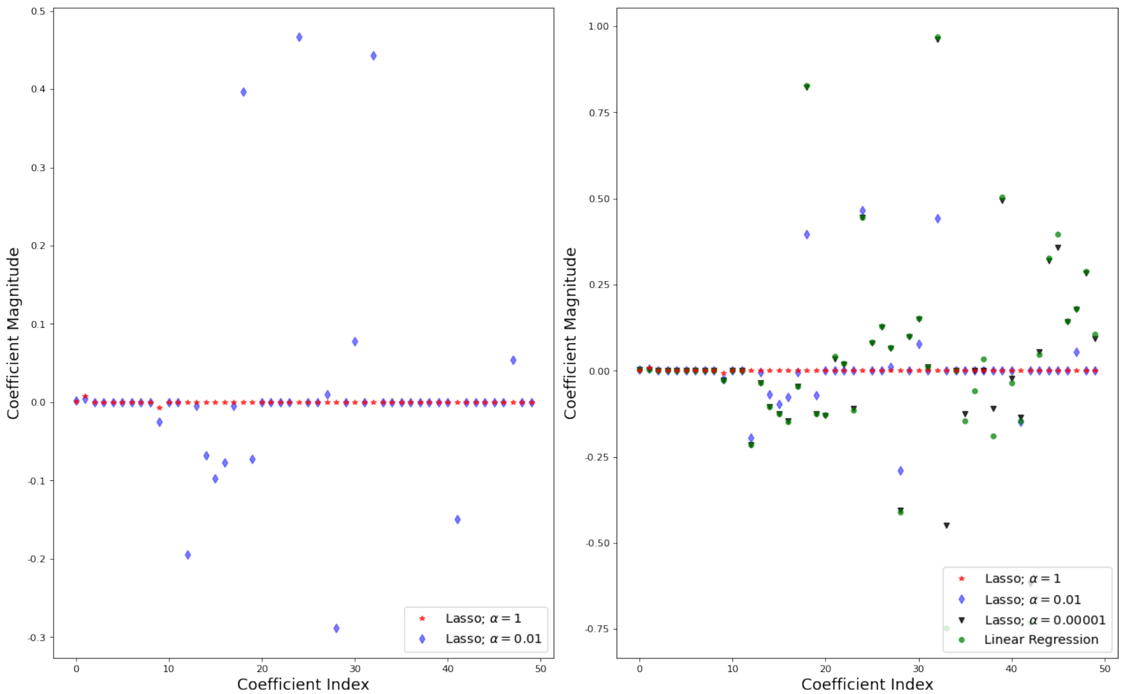
Without abandoning any feature, ridge regression reduces the regression coefficients, making the model relatively stable. But compared with Lasso regression, this will leave a lot of model features and poor model interpretation.

[ ]:
```
lasso = Lasso() #  alpha =1
lasso.fit(X_train,y_train)

train_score=lasso.score(X_train,y_train)
test_score=lasso.score(X_test,y_test)

coeff_used = np.sum(lasso.coef_!=0)
print("training score:", train_score )
print ("test score: ", test_score)
print ("number of features used: ", coeff_used)

lasso001 = Lasso(alpha=0.01, max_iter=10e5)
lasso001.fit(X_train,y_train)

train_score001=lasso001.score(X_train,y_train)
test_score001=lasso001.score(X_test,y_test)

coeff_used001 = np.sum(lasso001.coef_!=0)
print ("training score for alpha=0.01:", train_score001 )
print ("test score for alpha =0.01: ", test_score001)
print ("number of features used: for alpha =0.01:", coeff_used001)

lasso00001 = Lasso(alpha=0.0001, max_iter=10e5)
lasso00001.fit(X_train,y_train)

train_score00001=lasso00001.score(X_train,y_train)
test_score00001=lasso00001.score(X_test,y_test)

coeff_used00001 = np.sum(lasso00001.coef_!=0)
```

```
print ("training score for alpha=0.0001:", train_score00001 )
print( "test score for alpha =0.0001: ", test_score00001)
print ("number of features used: for alpha =0.0001:", coeff_used00001)

lr = LinearRegression()
lr.fit(X_train,y_train)
lr_train_score=lr.score(X_train,y_train)
lr_test_score=lr.score(X_test,y_test)
print ("LR training score:", lr_train_score )
print ("LR test score: ", lr_test_score)
figure(figsize=(16, 10), dpi=80)

plt.subplot(1,2,1)
plt.plot(lasso.coef_,alpha=0.
↪7,linestyle='none',marker='*',markersize=5,color='red',label=r'Lasso;␣
↪$\alpha = 1$',zorder=7) # alpha here is for transparency
plt.plot(lasso001.coef_,alpha=0.
↪5,linestyle='none',marker='d',markersize=6,color='blue',label=r'Lasso;␣
↪$\alpha = 0.01$') # alpha here is for transparency

plt.xlabel('Coefficient Index',fontsize=16)
plt.ylabel('Coefficient Magnitude',fontsize=16)
plt.legend(fontsize=13,loc=4)
plt.subplot(1,2,2)
plt.plot(lasso.coef_,alpha=0.
↪7,linestyle='none',marker='*',markersize=5,color='red',label=r'Lasso;␣
↪$\alpha = 1$',zorder=7) # alpha here is for transparency
plt.plot(lasso001.coef_,alpha=0.
↪5,linestyle='none',marker='d',markersize=6,color='blue',label=r'Lasso;␣
↪$\alpha = 0.01$') # alpha here is for transparency
plt.plot(lasso00001.coef_,alpha=0.
↪8,linestyle='none',marker='v',markersize=6,color='black',label=r'Lasso;␣
↪$\alpha = 0.00001$') # alpha here is for transparency
plt.plot(lr.coef_,alpha=0.
↪7,linestyle='none',marker='o',markersize=5,color='green',label='Linear␣
↪Regression',zorder=2)
plt.xlabel('Coefficient Index',fontsize=16)
plt.ylabel('Coefficient Magnitude',fontsize=16)
plt.legend(fontsize=13,loc=4)
plt.tight_layout()
plt.show()
```

```
/usr/local/lib/python3.7/dist-
packages/sklearn/linear_model/_coordinate_descent.py:476: ConvergenceWarning:
Objective did not converge. You might want to increase the number of iterations.
Duality gap: 1049.4450971643942, tolerance: 0.30315481997763694
  positive)
```

```
training score: 0.3354963706780224
test score:  0.28416090719225273
number of features used:  12
training score for alpha=0.01: 0.4852405649033092
test score for alpha =0.01:  0.4477227092856373
number of features used: for alpha =0.01: 27
training score for alpha=0.0001: 0.5098119359755454
test score for alpha =0.0001:  0.46212615450498623
number of features used: for alpha =0.0001: 47
LR training score: 0.5098707388489385
LR test score:  0.4619513375550573
```



When we look at the picture on the left, we can see that for alpha = 1,most of the coefficients are zero or close to zero. Lasso kills too many features. And with alpha = 0.01, this model has more features. When we set alpha = 0.0001, 47 non-zero feature attributes remain in the model and the training and test scores are the same as the basic linear regression. Based on this result, alpha = 0.01 could be a little better.

[ ]:
```
lasso_result = lasso001.predict(X_test)
```

#Model Comparation

[ ]:
```
result_table = pd.DataFrame(y_test)
result_table.columns = ['IMDB_score']
result_table['Linear'] = LR_result
result_table['Ridge'] = rr_result
```

```
result_table['Lasso'] = lasso_result
result_table.sort_values(by=['IMDB_score'])
new_table = result_table.reset_index(drop=True)
new_table
```

[ ]:

```
      IMDB_score    Linear     Ridge     Lasso
0            4.3  5.688365  5.695836  5.806589
1            4.6  6.126685  6.118443  6.167607
2            6.5  5.987510  5.995089  6.169073
3            6.6  6.307683  6.302308  6.168252
4            7.3  6.662066  6.663338  6.661530
…              …         …         …         …
1145         7.5  5.984478  6.002163  6.017775
1146         6.8  6.471635  6.487194  6.620229
1147         6.7  6.435611  6.386730  6.210141
1148         8.0  7.287553  7.291819  7.550669
1149         1.9  5.736781  5.746833  5.763123

[1150 rows x 4 columns]
```

[ ]:

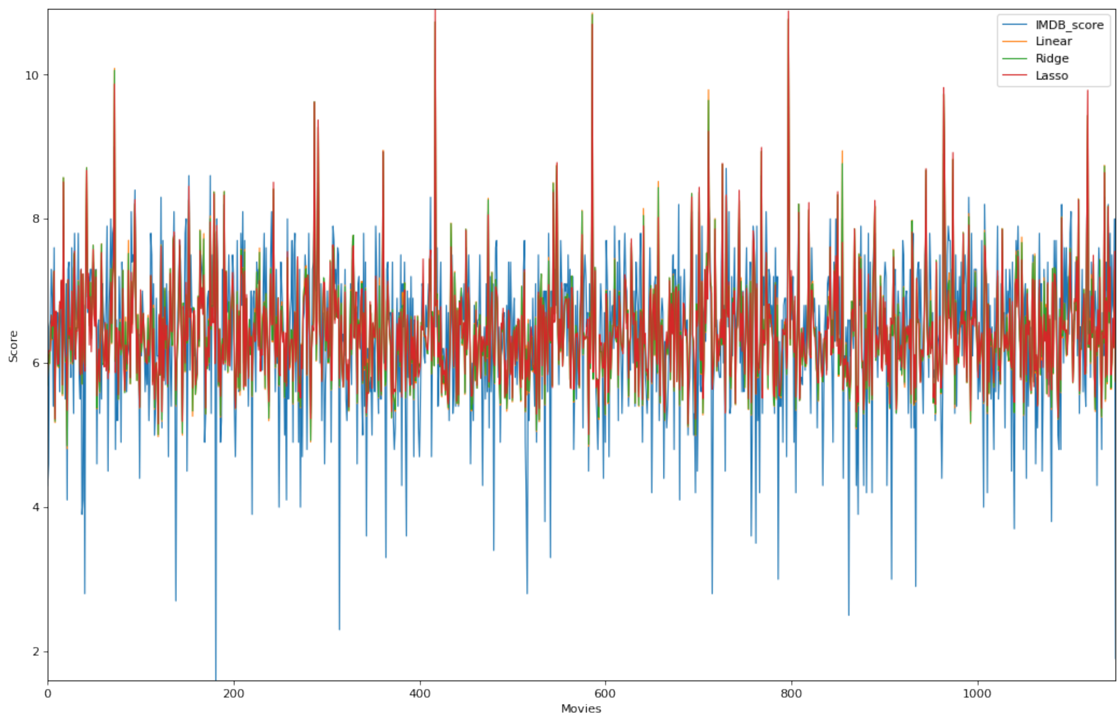
```
IMDB_score = new_table['IMDB_score']
linear = new_table['Linear']
ridge = new_table['Ridge']
lasso = new_table['Lasso']
index = range(len(new_table))
```

[ ]:

```
figure(figsize=(16, 10), dpi=80)
plt.plot(index,  IMDB_score,label='IMDB_score',linewidth=1)
plt.plot(index, linear, label='Linear',linewidth=1)
plt.plot(index, ridge, label='Ridge',linewidth=1)
plt.plot(index, lasso, label='Lasso',linewidth=1)
plt.legend()

plt.margins(0)
plt.subplots_adjust(bottom=0.10)
plt.xlabel('Movies')
plt.ylabel("Score")
```

[ ]:

```
Text(0, 0.5, 'Score')
```



We can hardly see any detail here, it is a mess. But we can tell that for some movies with pretty low score, three models cannot tell precisely. And their predictions are sometimes staying higher than the real score.

[ ]:
```
print(f"Linear Regression MSE {mean_squared_error(LR_result, y_test)}")
print(f"Linear Regression RMSE {np.sqrt(mean_squared_error(LR_result,␣
↪y_test))}")
print(f"Linear Regression R^2 {r2_score(y_test, LR_result)}")
```

```
Linear Regression MSE 0.584968462720589
Linear Regression RMSE 0.7648323101965483
Linear Regression R^2 0.46195133755505724
```

[ ]:
```
print(f"Ridge Regression MSE {mean_squared_error(rr_result, y_test)}")
print(f"Ridge Regression RMSE {np.sqrt(mean_squared_error(rr_result, y_test))}")
print(f"Ridge Regression R^2 {r2_score(y_test, rr_result)}")
```

```
Ridge Regression MSE 0.5835446818982589
Ridge Regression RMSE 0.7639009634096942
Ridge Regression R^2 0.4632609181835694
```

[ ]:
```
print(f"Lasso Regression MSE {mean_squared_error(lasso_result, y_test)}")
print(f"Lasso Regression RMSE {np.sqrt(mean_squared_error(lasso_result,␣
↪y_test))}")
print(f"Lasso Regression R^2 {r2_score(y_test, lasso_result)}")
```

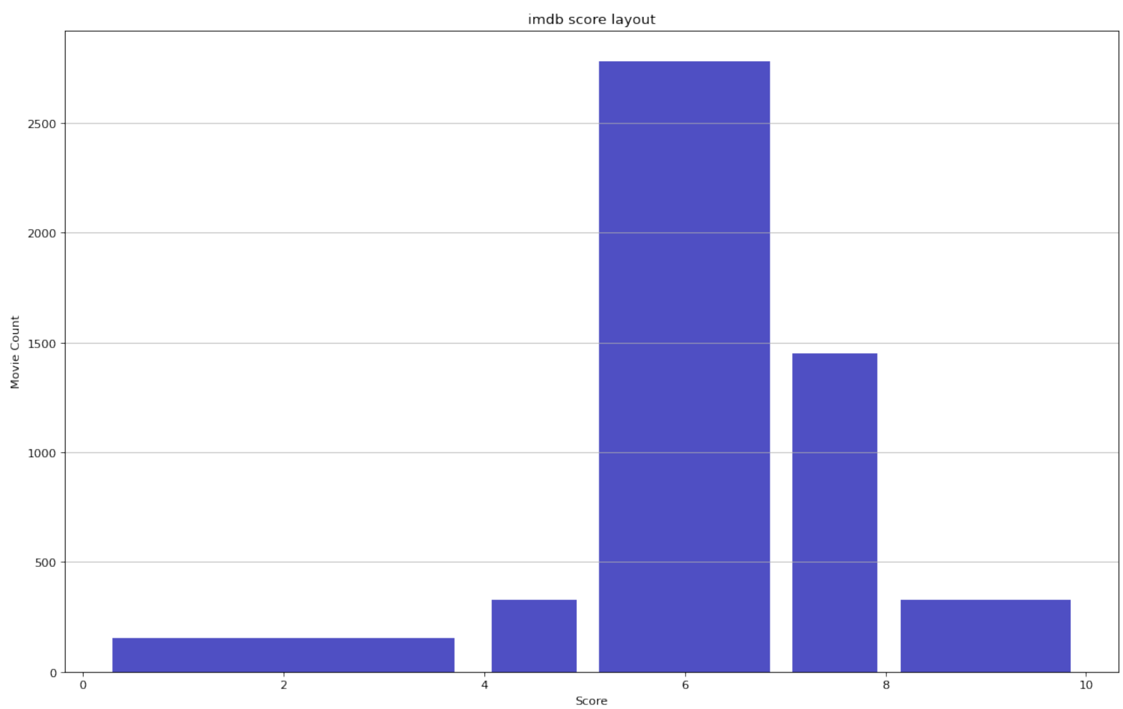
```
Lasso Regression MSE 0.6004378791253496
Lasso Regression RMSE 0.7748792674509686
Lasso Regression R^2 0.4477227092856373
```

Compared to the baseline model, lasso and ridge both did good job while ridge regression is a little better. I would say this model cannot make a perfect prediction of IMDB score yet, but it do tell some story about it and can be tuned further.

#Classification models

Sometimes we rate movies as “average”,“good”,“attractive”,“execellent” based on its performance. If we build some intervals for IMDB score, we can convert numerical data to categorical data, then its score transfers to words. Let’s map these to describe words for the numerical data. Here I use 5 bins:0~4(Bad) 4~5(Below average) 5~7(average) 7~8(Good) 8~10(Execellent). Noted here intervals are not equal cause the score distribution is not symmetrical. In order to map the score with common sense, I use this specific bin range. Before we deploy models, let’s take a overview.

[ ]:
```
figure(figsize=(16, 10), dpi=80)
n, bins, patches = plt.hist(x=data['imdb_score'], bins=[0,4,5,7,8,10],␣
 ↪color='#0504aa',alpha=0.7, rwidth=0.85)
plt.grid(axis='y', alpha=0.75)
plt.xlabel('Score')
plt.ylabel('Movie Count')
plt.title('imdb score layout')
maxfreq = n.max()
```



[ ]:
```
bins = [0,4,5,7,8,10]
names = ['Bad', 'Below Average', 'Average', 'Good', 'Execellent']

data_finished['imdb_score'] = pd.cut(data_finished['imdb_score'], bins,␣
↪labels=names)
```

[ ]:
```
data_finished['imdb_score']
```

[ ]:
```
0             Good
1             Good
2          Average
3       Execellent
4          Average
           …
3828       Average
3829       Average
3830       Average
3831       Average
3832       Average
Name: imdb_score, Length: 3833, dtype: category
Categories (5, object): ['Bad' < 'Below Average' < 'Average' < 'Good' <
'Execellent']
```

Like I did before, split the dataset one more time.

[ ]:
```
df = data_finished.sample(n = len(data_finished), random_state = 1)
df_data = df.reset_index(drop = True)
df_test=df_data.sample(frac=0.30,random_state=42)
df_train = df_data.drop(df_test.index)
col_to_use = [c for c in list(df_train.columns) if c != 'imdb_score']
print('Number of attributes:', len(col_to_use))
X_train = df_train[col_to_use].values
X_test = df_test[col_to_use].values
y_train = df_train['imdb_score'].values
y_test = df_test['imdb_score'].values
print('Training shapes:',X_train.shape, y_train.shape)
print('Testing shapes:',X_test.shape, y_test.shape)
```

```
Number of attributes: 50
Training shapes: (2683, 50) (2683,)
Testing shapes: (1150, 50) (1150,)
```

First I want to try KNN, k-nearest neighbors, which is very popular and basic in classification problems. In this case, I will look at the closest 3 neighbors. It is not appropriate to set the number of neighbors larget that 5 in this case, because we have 5 intervals and samples we have is limited. If we use a larger number, it's likely to predict most of our samples as "Average", which is a result of overfitting. In order to find the best parameter here, we can tune the parameter in further discussion.

[ ]:
```
knn=KNeighborsClassifier(n_neighbors = 3)
knn.fit(X_train, y_train)
```

[ ]:
```
KNeighborsClassifier(algorithm='auto', leaf_size=30, metric='minkowski',
                     metric_params=None, n_jobs=None, n_neighbors=3, p=2,
                     weights='uniform')
```

[ ]:
```
knn.predict(X_test[:100,:])
```

[ ]:
```
array(['Average', 'Good', 'Average', 'Good', 'Average', 'Average',
       'Average', 'Average', 'Average', 'Average', 'Average', 'Average',
       'Below Average', 'Average', 'Average', 'Good', 'Good', 'Average',
       'Average', 'Average', 'Below Average', 'Average', 'Average',
       'Average', 'Average', 'Average', 'Average', 'Good', 'Average',
       'Execellent', 'Average', 'Average', 'Average', 'Average', 'Good',
       'Average', 'Average', 'Average', 'Average', 'Average', 'Average',
       'Average', 'Good', 'Average', 'Good', 'Average', 'Average',
       'Average', 'Good', 'Average', 'Average', 'Good', 'Average',
       'Average', 'Good', 'Average', 'Average', 'Average', 'Average',
       'Average', 'Average', 'Average', 'Average', 'Average', 'Good',
       'Average', 'Average', 'Average', 'Average', 'Average', 'Average',
       'Good', 'Average', 'Average', 'Average', 'Average', 'Average',
       'Average', 'Average', 'Average', 'Average', 'Good', 'Average',
       'Average', 'Good', 'Average', 'Average', 'Good', 'Below Average',
       'Average', 'Average', 'Average', 'Average', 'Average', 'Good',
       'Average', 'Average', 'Average', 'Below Average', 'Average'],
      dtype=object)
```

[ ]:
```
y_train_pred= knn.predict(X_train[:,:])
y_test_pred= knn.predict(X_test[:,:])
```

[ ]:
```
y_train_one = pd.get_dummies(y_train)
y_test_one = pd.get_dummies(y_test)
y_train_pred_one = pd.get_dummies(y_train_pred)
y_test_pred_one = pd.get_dummies(y_test_pred)
if 'Bad' not in df:
  bad = [0]*len(y_test_pred_one)
  y_test_pred_one['Bad'] = bad

y_test_one = y_test_one[['Bad','Below Average', 'Average','Good','Execellent']].
 ↪to_numpy()
y_train_one = y_train_one[['Bad','Below Average',␣
 ↪'Average','Good','Execellent']].to_numpy()
y_test_pred_one = y_test_pred_one[['Bad','Below Average',␣
 ↪'Average','Good','Execellent']].to_numpy()
y_train_pred_one = y_train_pred_one[['Bad','Below Average',␣
 ↪'Average','Good','Execellent']].to_numpy()
```

```
[ ]: y_test_one = pd.get_dummies(y_test)
     y_test_one = y_test_one[['Bad','Below Average', 'Average','Good','Execellent']]
     y_test_one = y_test_one.to_numpy()
```

```
[ ]: knn_train_roc_score = roc_auc_score(y_train_one, y_train_pred_one,␣
      ↪multi_class='ovr')
     knn_train_hamming = hamming_loss(y_train, y_train_pred)
```

```
[ ]: knn_test_roc_score = roc_auc_score(y_test_one, y_test_pred_one,␣
      ↪multi_class='ovr')
     knn_test_hamming = hamming_loss(y_test, y_test_pred)
     print(knn_test_roc_score)
     print(knn_test_hamming)
```

```
0.4807062270816977
0.8382608695652174
```

This is the standard logistic regression model

```
[ ]: lr=LogisticRegression(random_state = 1)
     lr.fit(X_train, y_train)
```

```
/usr/local/lib/python3.7/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)
```

```
[ ]: LogisticRegression(C=1.0, class_weight=None, dual=False, fit_intercept=True,
                        intercept_scaling=1, l1_ratio=None, max_iter=100,
                        multi_class='auto', n_jobs=None, penalty='l2',
                        random_state=1, solver='lbfgs', tol=0.0001, verbose=0,
                        warm_start=False)
```

```
[ ]: y_train_pred= lr.predict(X_train)
     y_test_pred= lr.predict(X_test)
```

```
[ ]: y_train_pred_one = pd.get_dummies(y_train_pred)
     y_test_pred_one = pd.get_dummies(y_test_pred)
     if 'Bad' not in y_test_pred_one:
       bad = [0]*len(y_test_pred_one)
       y_test_pred_one['Bad'] = bad
     if 'Below Average' not in y_test_pred_one:
```

```
  bad = [0]*len(y_test_pred_one)
  y_test_pred_one['Below Average'] = bad
y_test_pred_one = y_test_pred_one[['Bad','Below Average',␣
↪'Average','Good','Execellent']].to_numpy()
y_train_pred_one = y_train_pred_one[['Bad','Below Average',␣
↪'Average','Good','Execellent']].to_numpy()

lr_train_roc_score = roc_auc_score(y_train_one, y_train_pred_one,␣
↪multi_class='ovr')
lr_train_hamming = hamming_loss(y_train, y_train_pred)

lr_test_roc_score = roc_auc_score(y_test_one, y_test_pred_one,␣
↪multi_class='ovr')
lr_test_hamming = hamming_loss(y_test, y_test_pred)
print(lr_roc_score)
print(lr_hamming)
```

```
0.512384648524283
0.4573913043478261
```

If we have lots of data, like all movies' IMDB scores, logistic regression may take a long time to compute. We have an alternative approach called stochastic gradient descent that works similarly to logistic regression but doesn't use all the data at each iteration.

[ ]:
```
sgdc=SGDClassifier(loss = 'log',alpha = 0.1,random_state = 1)
sgdc.fit(X_train, y_train)
```

[ ]:
```
SGDClassifier(alpha=0.1, average=False, class_weight=None, early_stopping=False,
              epsilon=0.1, eta0=0.0, fit_intercept=True, l1_ratio=0.15,
              learning_rate='optimal', loss='log', max_iter=1000,
              n_iter_no_change=5, n_jobs=None, penalty='l2', power_t=0.5,
              random_state=1, shuffle=True, tol=0.001, validation_fraction=0.1,
              verbose=0, warm_start=False)
```

[ ]:
```
y_train_pred = sgdc.predict(X_train)
y_test_pred = sgdc.predict(X_test)
```

[ ]:
```
y_train_pred_one = pd.get_dummies(y_train_pred)
y_test_pred_one = pd.get_dummies(y_test_pred)
if 'Bad' not in y_train_pred_one:
  bad = [0]*len(y_train_pred_one)
  y_train_pred_one['Bad'] = bad
if 'Below Average' not in y_train_pred_one:
  bad = [0]*len(y_train_pred_one)
  y_train_pred_one['Below Average'] = bad
if 'Bad' not in y_test_pred_one:
  bad = [0]*len(y_test_pred_one)
```

```
  y_test_pred_one['Bad'] = bad
if 'Below Average' not in y_test_pred_one:
  bad = [0]*len(y_test_pred_one)
  y_test_pred_one['Below Average'] = bad
y_test_pred_one = y_test_pred_one[['Bad','Below Average',␣
 ↪'Average','Good','Execellent']].to_numpy()
y_train_pred_one = y_train_pred_one[['Bad','Below Average',␣
 ↪'Average','Good','Execellent']].to_numpy()

sgdc_train_roc_score = roc_auc_score(y_train_one, y_train_pred_one,␣
 ↪multi_class='ovr')
sgdc_train_hamming = hamming_loss(y_train, y_train_pred)

sgdc_test_roc_score = roc_auc_score(y_test_one, y_test_pred_one,␣
 ↪multi_class='ovr')
sgdc_test_hamming = hamming_loss(y_test, y_test_pred)
print(sgdc_roc_score)
print(sgdc_hamming)
```

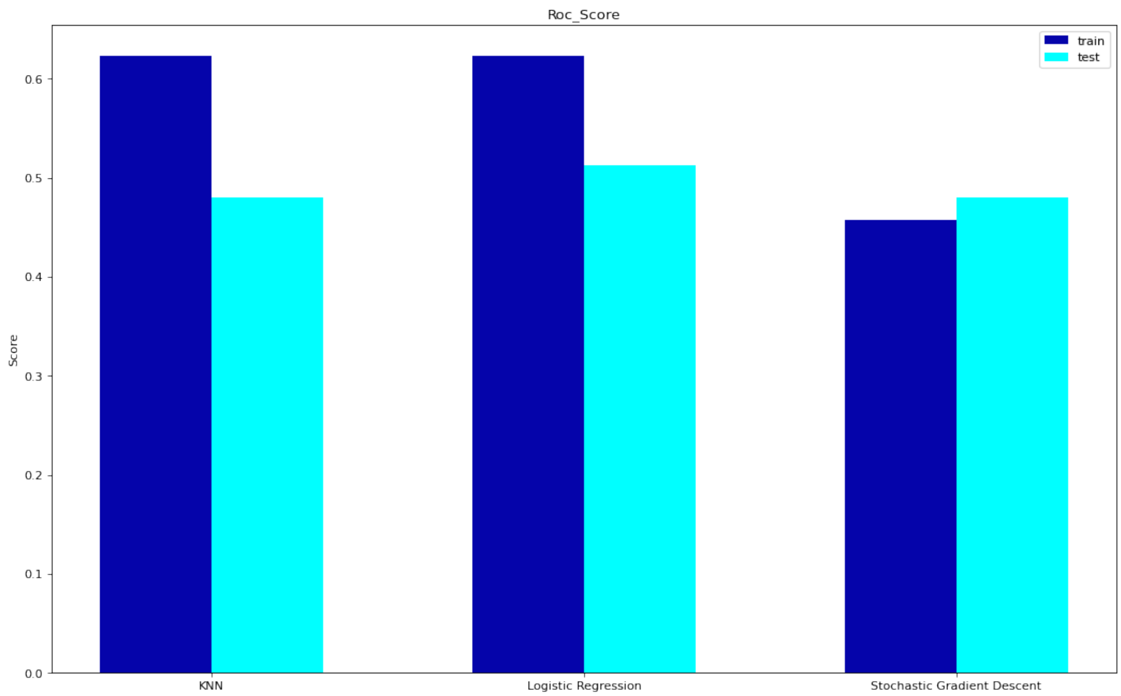
```
0.512384648524283
0.4573913043478261
```

#Classification Model Comparision

In this case, we have a multi-class classification problem here thus it is inappropriate to use accuracy or F1 score for model evaluation. Here I choose ROC AUC score and hamming loss as evaluation stats for our classification models. Roc_AUC_score is the area under the receiver operating characteristic curve (ROC AUC) from prediction scores. Higher the score, more accurate the model is. Since we have multiple classes here, this score is computed by the AUC of each class against the rest. It treats the multiclass case in the same way as the multilabel case.

[ ]:
```
Model_name = ('KNN','Logistic Regression', 'Stochastic Gradient Descent')
train_score = [knn_train_roc_score, lr_train_roc_score ,sgdc_train_roc_score]
test_score = [knn_test_roc_score, lr_test_roc_score, sgdc_test_roc_score]

bar_width = 0.3
index_train = np.arange(len(Model_name))
index_test = index_train + bar_width

figure(figsize=(16, 10), dpi=80)
plt.bar(index_train, height=train_score, width=bar_width, color='#0504aa',␣
 ↪label='train')
plt.bar(index_test, height=test_score, width=bar_width, color='#00FFFF',␣
 ↪label='test')

plt.legend()
plt.xticks(index_train + bar_width/2, Model_name)
plt.ylabel('Score')
```

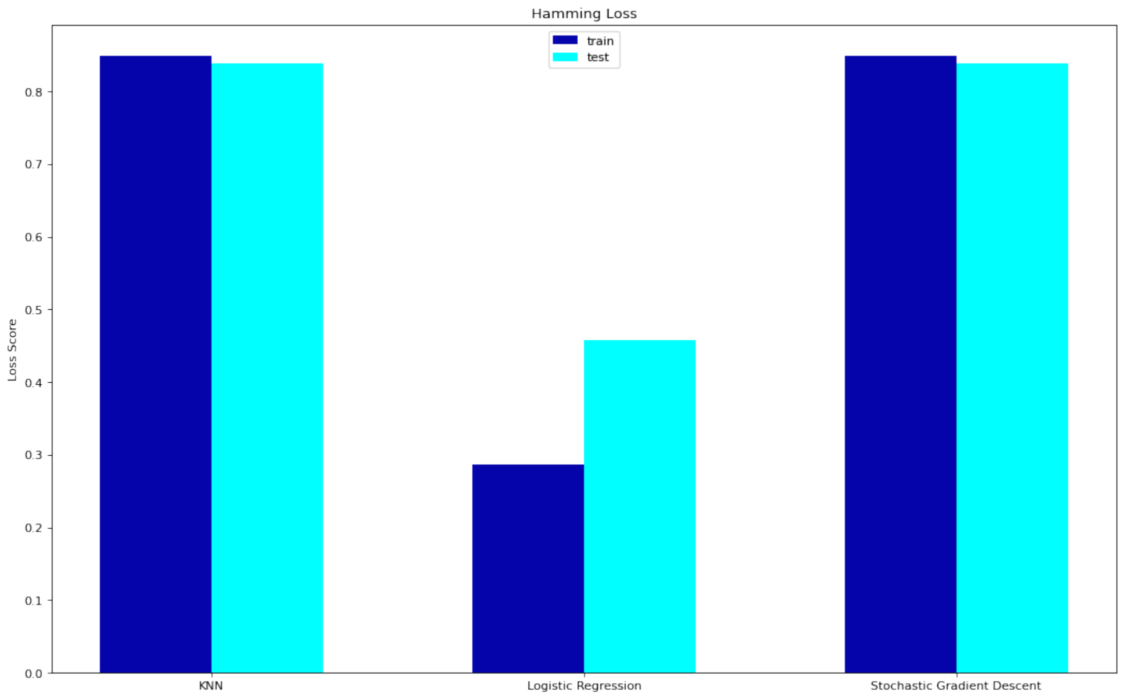
```
plt.title('Roc_Score')

plt.show()
```



It is pretty weird that in stochastic gradient descent model we even have a better test score that trainning score. But the most important thing is to compare the ROC score of the testset. Here Logistic regression has the best performance over the other two.

[ ]:
```
Model_name = ('KNN','Logistic Regression', 'Stochastic Gradient Descent')
train_score = [knn_train_hamming, lr_train_hamming,sgdc_train_hamming]
test_score = [knn_test_hamming, lr_test_hamming, sgdc_test_hamming]

bar_width = 0.3
index_train = np.arange(len(Model_name))
index_test = index_train + bar_width

figure(figsize=(16, 10), dpi=80)
plt.bar(index_train, height=train_score, width=bar_width, color='#0504aa',␣
 ↪label='train')
plt.bar(index_test, height=test_score, width=bar_width, color='#00FFFF',␣
 ↪label='test')

plt.legend()
plt.xticks(index_train + bar_width/2, Model_name)
plt.ylabel('Loss Score')
```

```
plt.title('Hamming Loss')

plt.show()
```



The Hamming loss is the fraction of labels that are incorrectly predicted. Lower the score, better the predition. Logistic Regression give us a pretty good result here. Both KNN and stochastic gradient descent have lots of wrong prediction based on hamming loss. Based on stats above, I would suggest Logistic Regression is the best approch to do IMDB rating system classfication in these models. However, with parameter tuning and further improvement, I believe we can still improve the result a lot and reduce calculating costs.