# HW5

## APPLY LOGISTIC REGRESSION TO PREDICT IMDB_SCORE (LOW OR HIGH) ON IMDB


Dhanabordee Mekintharanggur
6238077121

# Step 1 - Loading the Data

The data is downloaded as a .csv file format and is placed in the same directory as the .ipynb file

The csv is read using pd.read_csv to read the data in the dataframe format for further exploration and preprocessing

| | director_name | num_critic_for_reviews | duration | director_facebook_likes | actor_3_facebook_likes | actor_2_name | actor_1_facebook_likes | |
|---|---|---|---|---|---|---|---|---|
| 0 | James Cameron | 723.0 | 178.0 | 0.0 | 855.0 | Joel David Moore | 1000.0 | 7 |
| 1 | Gore Verbinski | 302.0 | 169.0 | 563.0 | 1000.0 | Orlando Bloom | 40000.0 | 3 |
| 2 | Sam Mendes | 602.0 | 148.0 | 0.0 | 161.0 | Rory Kinnear | 11000.0 | 2 |
| 3 | Christopher Nolan | 813.0 | 164.0 | 22000.0 | 23000.0 | Christian Bale | 27000.0 | 4 |
| 4 | Doug Walker | NaN | NaN | 131.0 | NaN | Rob Walker | 131.0 | |
| ... | ... | ... | ... | ... | ... | ... | ... | |
| 5038 | Scott Smith | 1.0 | 87.0 | 2.0 | 318.0 | Daphne Zuniga | 637.0 | |
| 5039 | NaN | 43.0 | 43.0 | NaN | 319.0 | Valorie Curry | 841.0 | |
| 5040 | Benjamin Roberds | 13.0 | 76.0 | 0.0 | 0.0 | Maxwell Moody | 0.0 | |
| 5041 | Daniel Hsia | 14.0 | 100.0 | 0.0 | 489.0 | Daniel Henney | 946.0 | |
| 5042 | Jon Gunn | 43.0 | 90.0 | 16.0 | 16.0 | Brian Herzlinger | 86.0 | |

Notice that there are NaN values present in multiple columns. This has to be dealt with in the next step.

# Step 2 - Exploring and Cleaning the Data

There are originally 5043 rows of data

Multiple columns contain null values

Fortunately, the "imdb_score" column that I will be predicting has no empty rows.

```
RangeIndex: 5043 entries, 0 to 5042
Data columns (total 27 columns):
 #   Column                     Non-Null Count  Dtype
---  ------                     --------------  -----
 0   director_name              4939 non-null   object
 1   num_critic_for_reviews     4993 non-null   float64
 2   duration                   5028 non-null   float64
 3   director_facebook_likes    4939 non-null   float64
 4   actor_3_facebook_likes     5020 non-null   float64
 5   actor_2_name               5030 non-null   object
 6   actor_1_facebook_likes     5036 non-null   float64
 7   gross                      4159 non-null   float64
 8   genres                     5043 non-null   object
 9   actor_1_name               5036 non-null   object
 10  movie_title                5043 non-null   object
 11  num_voted_users            5043 non-null   int64
 12  cast_total_facebook_likes  5043 non-null   int64
 13  actor_3_name               5020 non-null   object
 14  facenumber_in_poster       5030 non-null   float64
 15  plot_keywords              4890 non-null   object
 16  movie_imdb_link            5043 non-null   object
 17  num_user_for_reviews       5022 non-null   float64
 18  language                   5031 non-null   object
 19  country                    5038 non-null   object
...
 25  aspect_ratio               4714 non-null   float64
 26  movie_facebook_likes       5043 non-null   int64
dtypes: float64(13), int64(3), object(11)
memory usage: 1.0+ MB
```

Upon **selecting only the columns with numerical values**, these are the remaining columns.

Some of the columns do contain empty values, but are at most 17% empty, which means that the response is not to immediately remove the entire column but rather to fill in the missing values or remove the rows specifically

At later point, I have decided to scale the data, normalize the data, and filling the missing value due to the normal distribution of the data as shown in the next page

```
gross                       17.529248
budget                       9.756098
aspect_ratio                 6.523895
title_year                   2.141582
director_facebook_likes      2.062265
num_critic_for_reviews       0.991473
actor_3_facebook_likes       0.456078
num_user_for_reviews         0.416419
duration                     0.297442
facenumber_in_poster         0.257783
actor_2_facebook_likes       0.257783
actor_1_facebook_likes       0.138806
cast_total_facebook_likes    0.000000
num_voted_users              0.000000
movie_facebook_likes         0.000000
imdb_score                   0.000000
dtype: float64
```

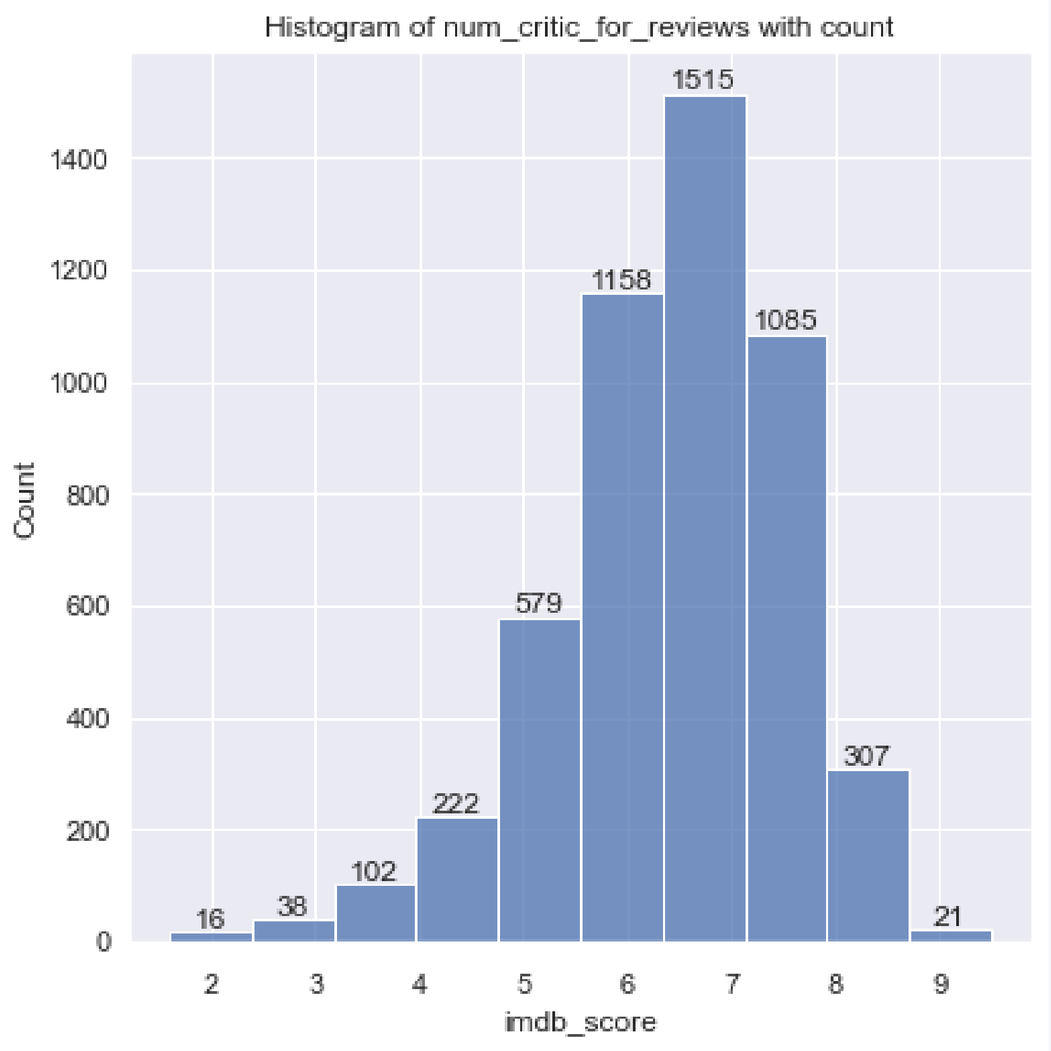
| | num_user_for_reviews | director_facebook_likes | actor_3_facebook_likes | duration |
|---|---|---|---|---|
| count | 5022.000 | 4939.000 | 5020.000 | 5028.000 |
| mean | 272.771 | 686.509 | 645.010 | 107.201 |
| std | 377.983 | 2813.329 | 1665.042 | 25.197 |
| min | 1.000 | 0.000 | 0.000 | 7.000 |
| 25% | 65.000 | 7.000 | 133.000 | 93.000 |
| 50% | 156.000 | 49.000 | 371.500 | 103.000 |
| 75% | 326.000 | 194.500 | 636.000 | 118.000 |
| max | 5060.000 | 23000.000 | 23000.000 | 511.000 |

# Step 2 - Exploring and Cleaning the Data

Visualizing the histogram of "imdb_score", it can be seen that the values appear to be somewhat **normally distributed**

The mean imdb_score is at 6.442 points

| | imdb_score |
|---|---|
| count | 5043.000 |
| mean | 6.442 |
| std | 1.125 |
| min | 1.600 |
| 25% | 5.800 |
| 50% | 6.600 |
| 75% | 7.200 |
| max | 9.500 |



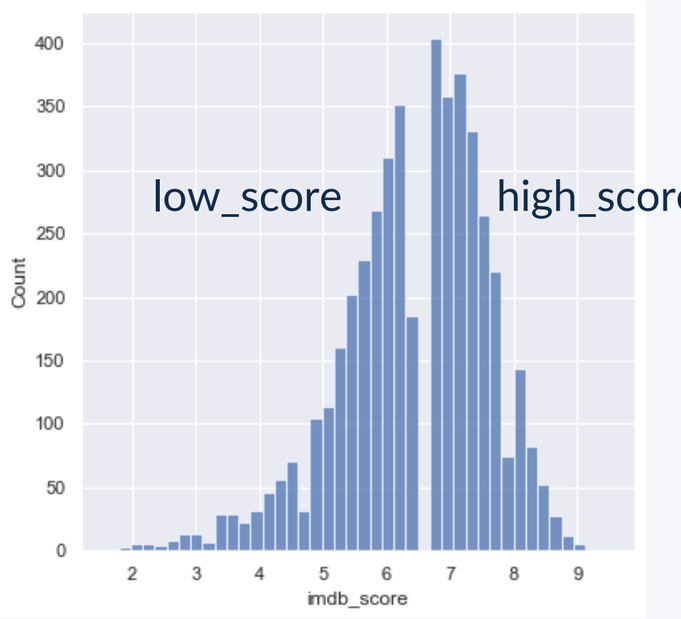
# Step 2 - Preprocess the Data

## Encode imdb_score as high or low imdb score (binary values)

Next, the imdb_score is categorized as a "high" or a "low".

Since there is practically no middle point to separate the high and low scores and in order to remove the ambiguity between films with close IMDb scores, the middle 10% percent of the data are removed.

The data between the 45-percentile and 55-percentile are dropped from the data frame, leaving 2 distinct sections to be categorized as a "high_score" represented by value of 1 in the high_imdb_score column, and a value of 0 in the same column for "low_score"




| | high_imdb_score |
|---|---|
| 0 | 1 |
| 1 | 1 |
| 2 | 1 |
| 3 | 1 |
| 4 | 1 |
| ... | ... |
| 5037 | 0 |
| 5038 | 1 |
| 5039 | 1 |
| 5040 | 0 |
| 5041 | 0 |

The new column is then added to the dataframe

the "movie_score_target" is a data frame as shown

# Step 2 - Preprocess the Data

## Split train-test

The data are then separated into x and y values
x: all the numerical columns except for 'imdb_score' and 'high_imdb_score' binary value column
y: the 'high_imdb_score' column created earlier

70% of the data will be used for training, and the remaining 30% will be used for testing

```
from sklearn.model_selection import train_test_split
x_train,x_test,y_train,y_test=train_test_split(
  x,
  movie_score_target,
  test_size=0.3,
  shuffle=True,
  stratify=movie_score_target,
  random_state=30
)
```

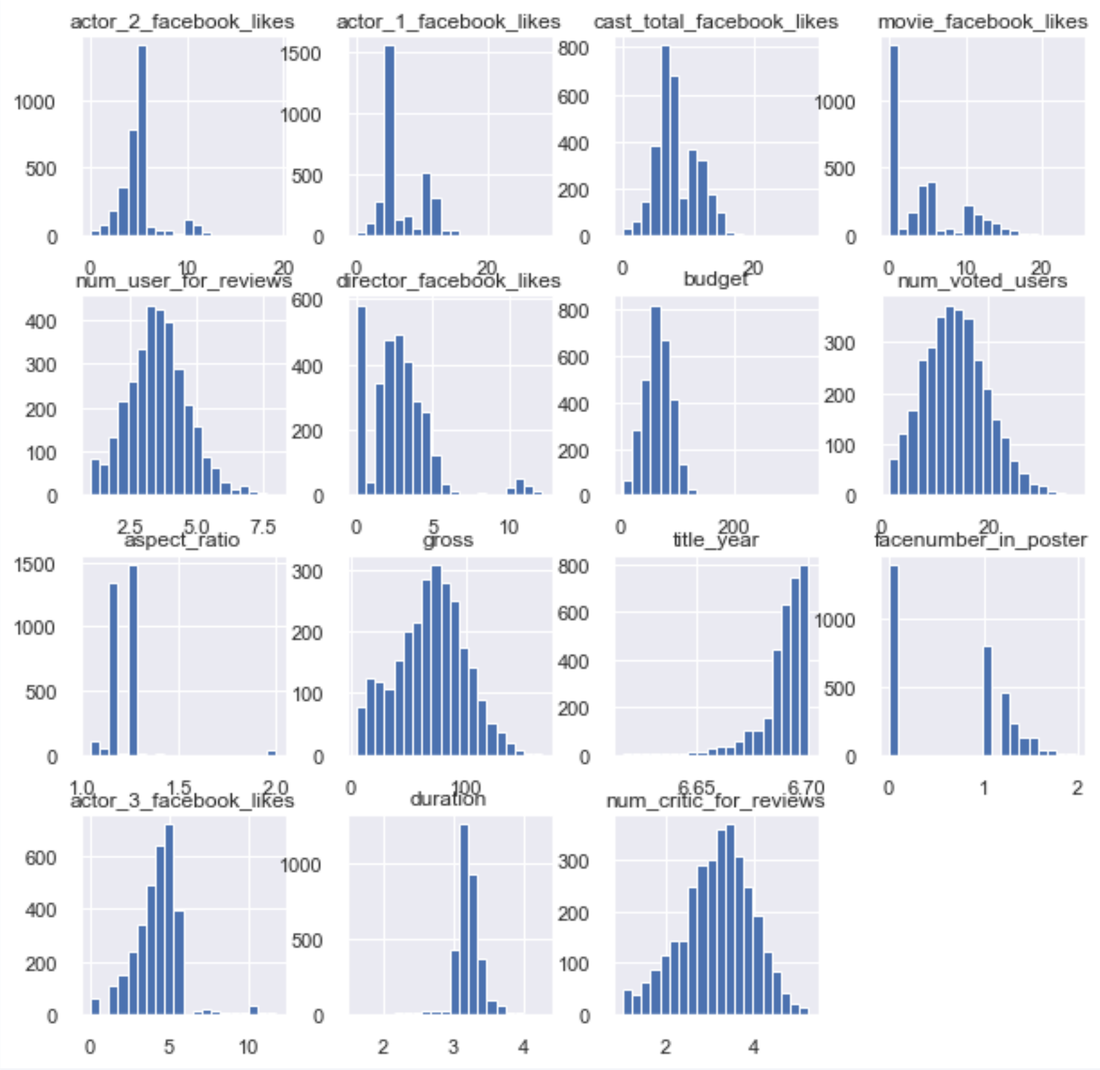
# Step 2 - Preprocess the Data

## Standardize features (comparable units) in train dataset

Each column is normalized to ensure that each feature has comparable units in the training dataset.

The normalization is done by taking the square roots of the numerical values of each column twice.

In doing so, it can be ensured that the data has a more normal distribution behavior to prepare the data to fill the missing values in the next step.



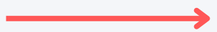
# Step 2 - Preprocess the Data

## Fill missing values

At this point, there are still some null values in the training dataset. Since the data have been normalized, it is possible to fill in the missing values using the mean of each specific column.

```
imputer=impute.SimpleImputer(missing_values=np.nan,strategy='median')
imputer.fit(x_train)
x_train=pd.DataFrame(imputer.transform(x_train),columns=x_train.columns)
x_test=pd.DataFrame(imputer.transform(x_test),columns=x_test.columns)
```

```
actor_2_facebook_likes          9
actor_1_facebook_likes          3
cast_total_facebook_likes       0
movie_facebook_likes            0
num_user_for_reviews           17
director_facebook_likes        76
budget                        332
num_voted_users                 0
aspect_ratio                  224
gross                         607
title_year                     78
facenumber_in_poster           11
actor_3_facebook_likes         14
duration                       11
num_critic_for_reviews         36
dtype: int64
```



```
actor_2_facebook_likes        0
actor_1_facebook_likes        0
cast_total_facebook_likes     0
movie_facebook_likes          0
num_user_for_reviews          0
director_facebook_likes       0
budget                        0
num_voted_users               0
aspect_ratio                  0
gross                         0
title_year                    0
facenumber_in_poster          0
actor_3_facebook_likes        0
duration                      0
num_critic_for_reviews        0
dtype: int64
```

# Step 2 - Preprocess the Data

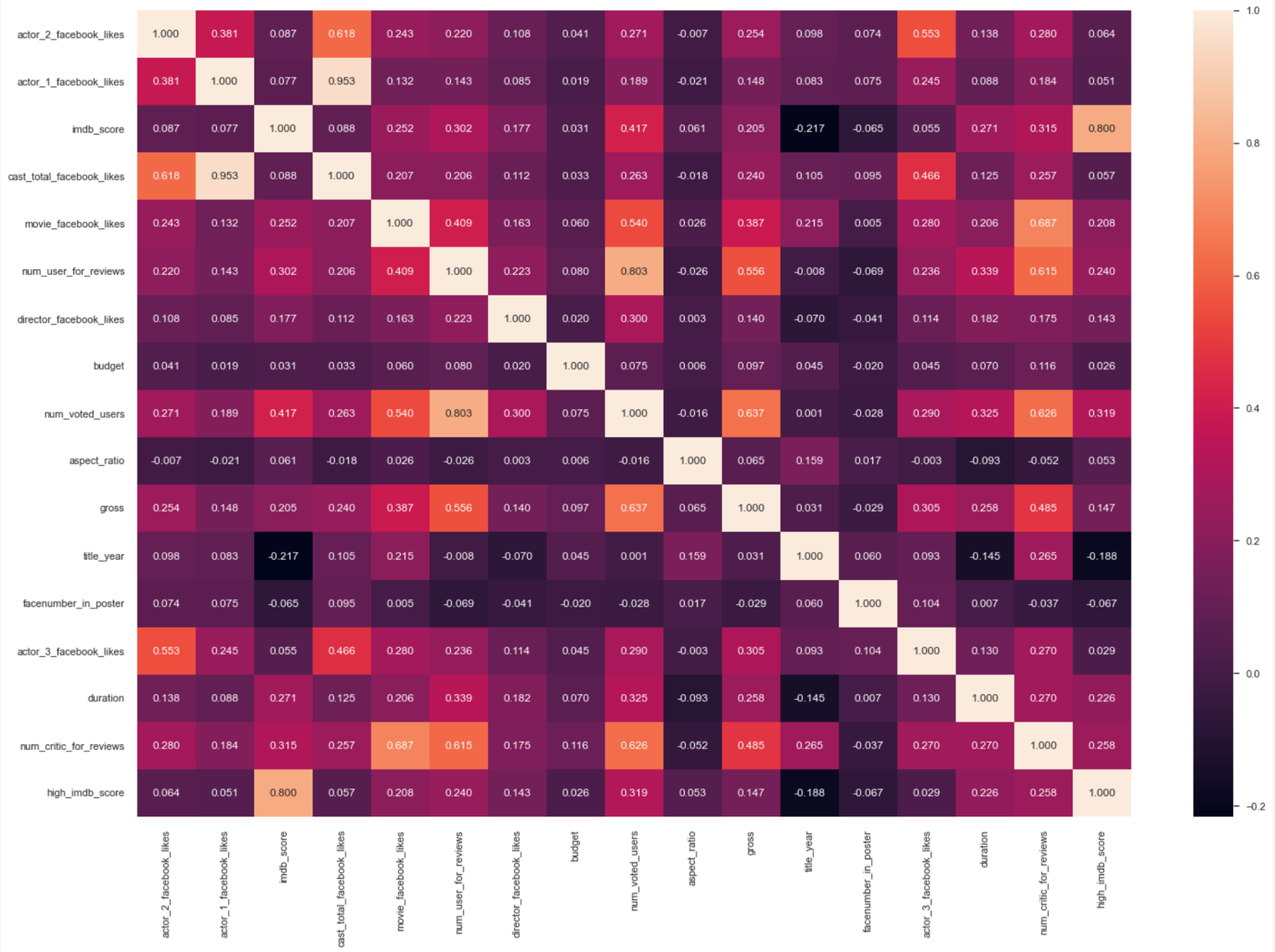
## Select features in train dataset

The dataset at the moment only contains numerical values.

The next step is to identify the useful numerical features that will be helpful for training the logistic regression model.

It is important the remove any highly-correlated features to avoid multicollinearity

The heatmap is plotted, and the features with high correlation are identified and chosen for removal



# Step 2 - Preprocess the Data

## Select features in train dataset

### Features with high correlation

| | | |
|---|---|---|
| num_user_for_reviews | num_voted_users | actor_1_facebook_likes |
| num_voted_users | num_user_for_reviews | actor_2_facebook_likes |
| gross | num_critic_for_reviews | actor_3_facebook_likes |
| num_critic_for_reviews | gross | cast_total_facebook_likes |

**Columns to drop:**

- gross
- num_user_for_reviews
- num_critic_for_reviews
- actor_1_facebook_likes
- actor_2_facebook_likes
- actor_3_facebook_likes

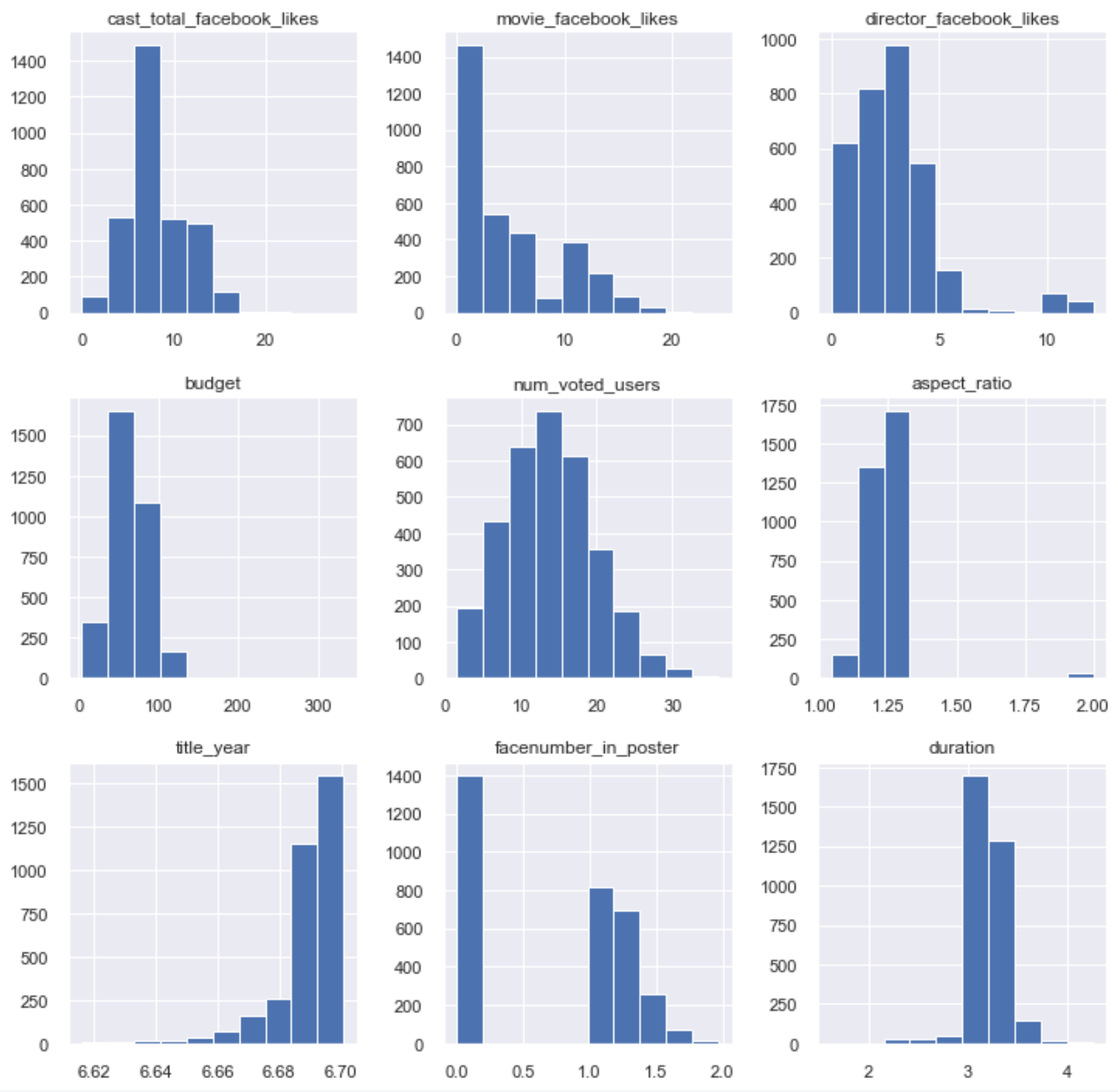
The columns with higher magnitude of correlation than 0.5 with each other suggests multicollinearity in the features, and should be removed as they may add noise when fitting the model.

The only features with the highest correlation with the "imdb_score" in each group are kept.

The rest of the features are dropped.

# Step 2 - Preprocess the Data

## Preprocess (based on train dataset) and predict test dataset (scaling)



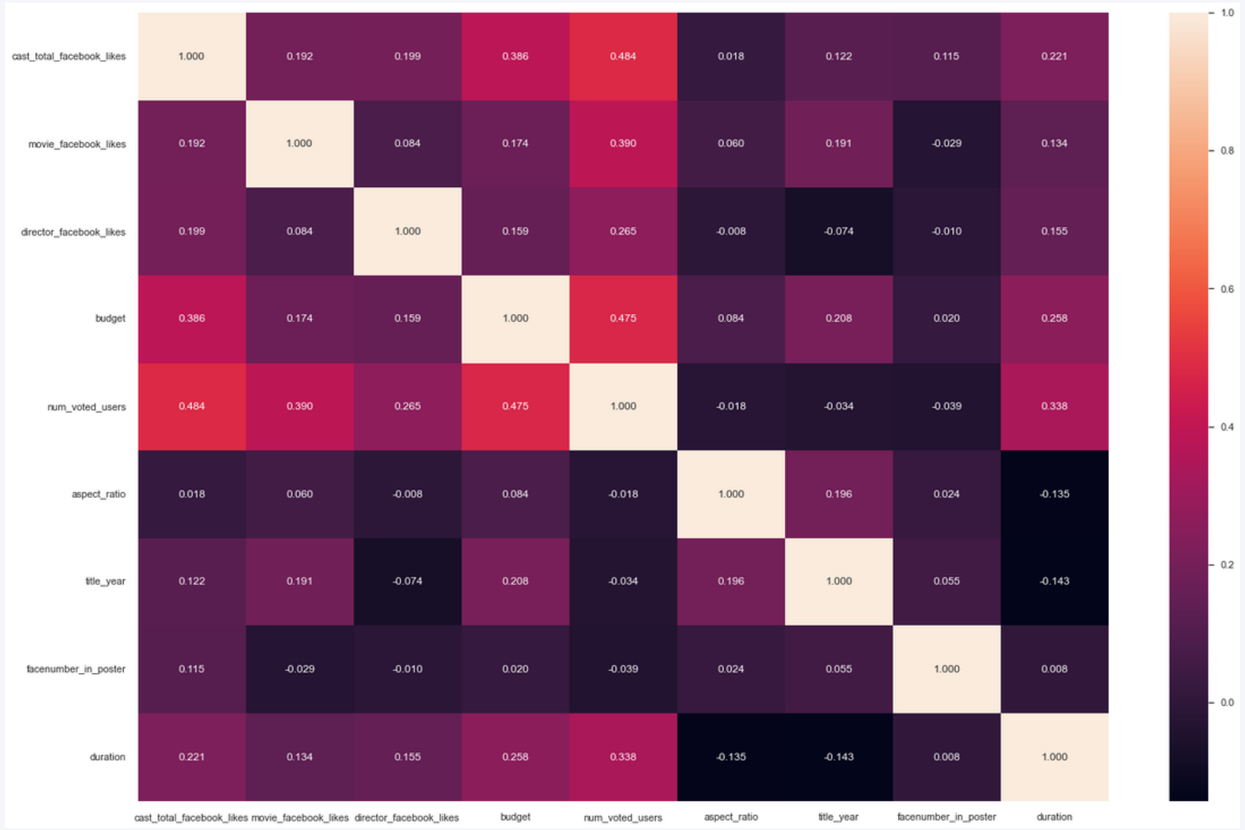

The train data that will be used to fit the logistic regression model is plotted as a histogram on the left. Some of the data are still different in terms of scale. Thus, the StandardScaler is used to fit the x_train data and transform them into using a more similar scale. Since the data have been previously normalized, the standard scaler can be used.

Also, after transforming the x_train data, the x_test data must also be scale-transformed.

# Step 3 - Model the Data

## Preprocess (based on train dataset) and predict test dataset (scaling)

From the pair plot and the heatmap, it can be seen that there is no correlations exceeding 0.5 between features anymore




# Step 3 - Model the Data

## Fitting the logistic regression model

```
#Create logistic regression
logr =LogisticRegression()

#Train the model using the training sets
logr.fit(x_train, y_train)

#Predict the response for test dataset
y_pred = logr.predict(x_test)
```

The processed x_train and y_train are then used to fit the Scikitlearn LogisticRegression model

Afterward, the model is used to make a prediction whether the IMDb score will be low or high in y_test using x_test data

```
high_imdb_score
1                1649
0                1610
dtype: int64
```

y_train value counts

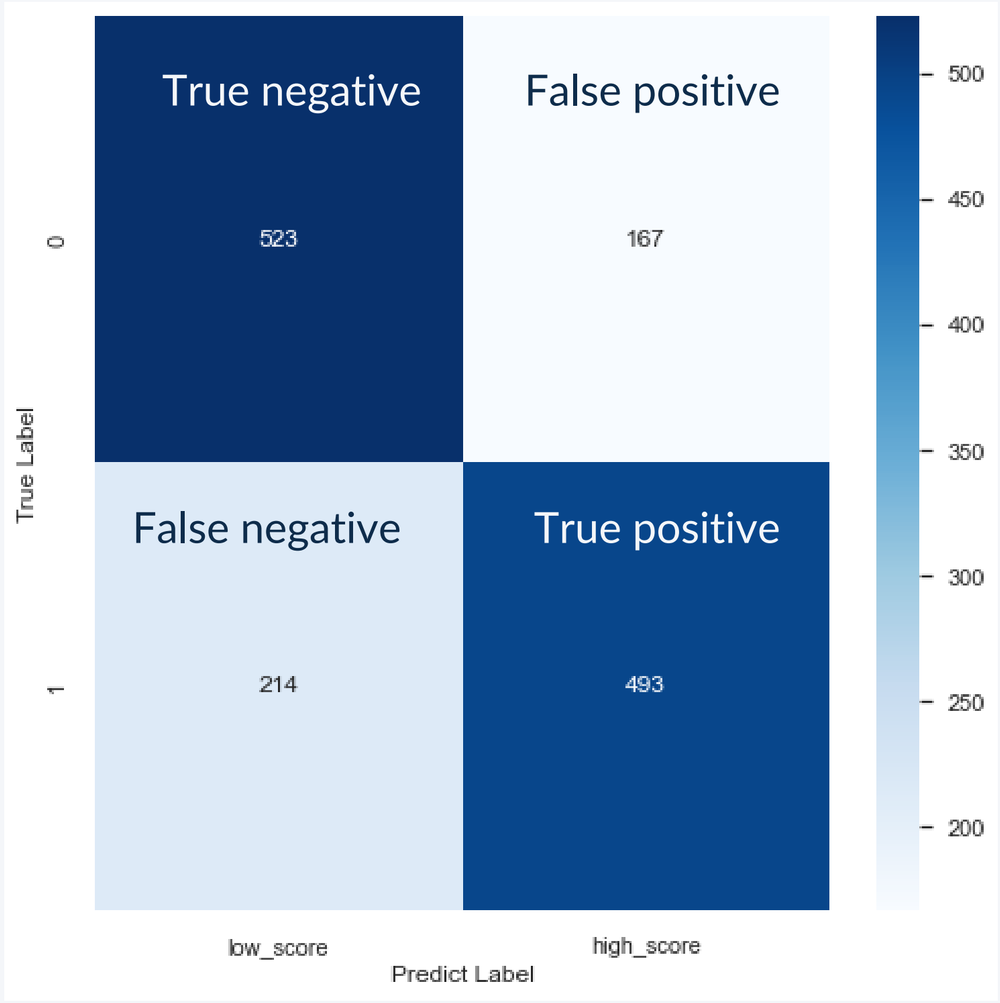
```
high_imdb_score
1                707
0                690
dtype: int64
```

y_test value counts

# Result Interpretation

## Accuracy

The confusion matrix visualizes the model's prediction results




## Accuracy

The training data has an **accuracy of 0.733**
The model is able to predict the test data correctly with an **accuracy of 0.727**

- The dataset is balanced as there are a nearly identical numbers of high and low imdb score movies
- Since this model is used to predict a social science project, an accuracy of 0.727 is considered as very well acceptable

# Result Interpretation

## Recall

523 low-scoring movies are correctly identified out of 690 movies

- recall = 0.76
- precision = 0.71

493 high-scoring movies are correctly identified out of 707 movies

- recall = 0.70
- precision = 0.75

I believe that both recall and precision are of similar importance, here. Instead it depends on what is a more severe case between misclassifying a high IMDB score movie as low as a low score one, or misclassifying a low IMDB score movie as a high score one.

In the first case, precision is more important, while in the latter case, recall will be more important

```
Classification Report:
              precision    recall  f1-score   support

   low_score       0.71      0.76      0.73       690
  high_score       0.75      0.70      0.72       707

    accuracy                           0.73      1397
   macro avg       0.73      0.73      0.73      1397
weighted avg       0.73      0.73      0.73      1397


Accuracy on train:   0.733

Accuracy on test:  0.727
```

# Result Interpretation

## F1 Score

The model is able to achieve a relatively high f1 score of 0.73 as both the precision and recall are high in the social science context.

Since it is unclear whether the precision or the recall should be given more importance in this project, the F1 score serves as a decent balanced metric for evaluating the model here.

```
Classification Report:
              precision    recall  f1-score   support

   low_score       0.71      0.76      0.73       690
  high_score       0.75      0.70      0.72       707

    accuracy                           0.73      1397
   macro avg       0.73      0.73      0.73      1397
weighted avg       0.73      0.73      0.73      1397


Accuracy on train:   0.733

Accuracy on test:  0.727
```

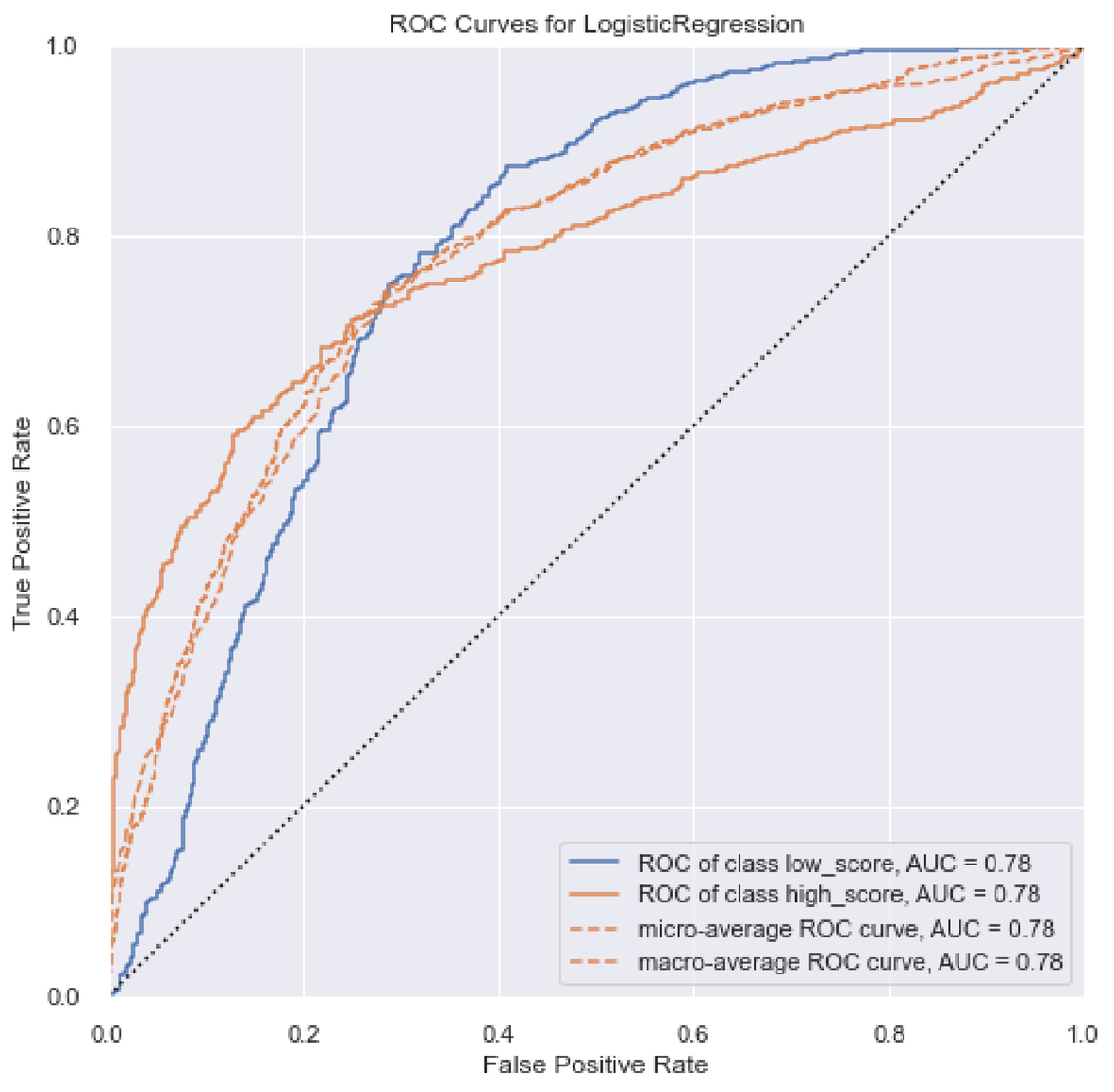
# Result Interpretation

## AUC scores

The AUC scores measure the ability of the binary classification model to separate the high_imdb and low_imdb cases.

Considering that the model achieves and AUC score of 0.78 for all cases (ROC of low_score, ROC of high_scoref, micro-avg ROC, and macro-average ROC).

An AUC of 0.78 is considered as acceptable but not excellent. While it is definitely better than a random coin flip (AUC of 0.5), it is still not always reliable.



# Result Interpretation

## Logistic Regression Equation, Log-odd, Odd

The intercept of the model is 0.07536737, with the coefficient for each feature as shown:

```
logr.intercept_
✓ 0.0s
array([0.07536737])
```

**Logistic Regression Equation:**

log (y1/y0) = 0.07536737 + 1.121112(x1) + 0.362813(x2) + ...

Meaning

- log(y1/y0) : log-odds
- x1, x2, x3 ... x9: each of the corresponding feature
- For each 1 unit change of the feature, the equation reveals how much the log-odd changes in response

| | coef |
|---|---|
| num_voted_users | 1.121112 |
| duration | 0.362813 |
| aspect_ratio | 0.253847 |
| movie_facebook_likes | 0.181063 |
| director_facebook_likes | 0.133364 |
| facenumber_in_poster | -0.097919 |
| cast_total_facebook_likes | -0.300441 |
| title_year | -0.309871 |
| budget | -0.681513 |

# Result Interpretation

## Logistic Regression Equation, Log-odd, Odd

Meaning
(y1/y0)before, (y1/y0)after : odds before and after the feature value changes
x1, x2, x3 ... x9: each of the corresponding feature

For each 1 unit change of the feature, it reveals how much the odd changes in response

```
logr.intercept_
✓ 0.0s
array([0.07536737])
```

| | coef |
|---|---|
| num_voted_users | 3.068264 |
| duration | 1.437367 |
| aspect_ratio | 1.288975 |
| movie_facebook_likes | 1.198490 |
| director_facebook_likes | 1.142666 |
| facenumber_in_poster | 0.906722 |
| cast_total_facebook_likes | 0.740491 |
| title_year | 0.733542 |
| budget | 0.505851 |