IMDB Movie Rating Prediction by Jing Kunzler

Code details can be found at https://github.com/jingkunzler211/IMDB_prediction.

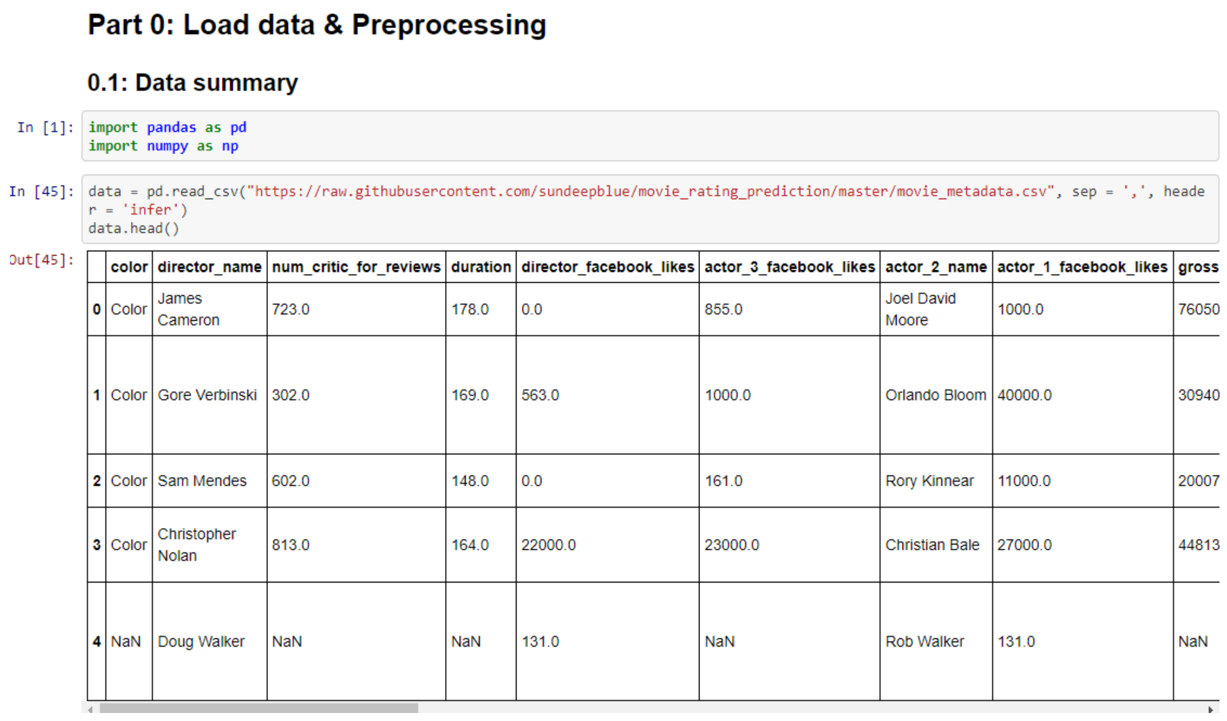
I. Overview of the dataset

This project used the scraped data from IMDB and www.the-numbers.com by https://github.com/sundeepblue/movie_rating_prediction in 2016, which contains 5043 movies and 28 columns.

```
<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
```

The goal of this project is to find the best algorithm and model performance for predicting imdb ratings with features selected from the data provided.

Upon first scanning the dataset, we can see that many columns contain NaN values, and there are some columns we don’t need for this project. In the next section, I’ll go into details about the transformations made in preparation for further analysis.



Part 0: Load data & Preprocessing

0.1: Data summary

```
In [1]:  import pandas as pd
         import numpy as np
```

```
In [45]: data = pd.read_csv("https://raw.githubusercontent.com/sundeepblue/movie_rating_prediction/master/movie_metadata.csv", sep = ',', heade
         r = 'infer')
         data.head()
```

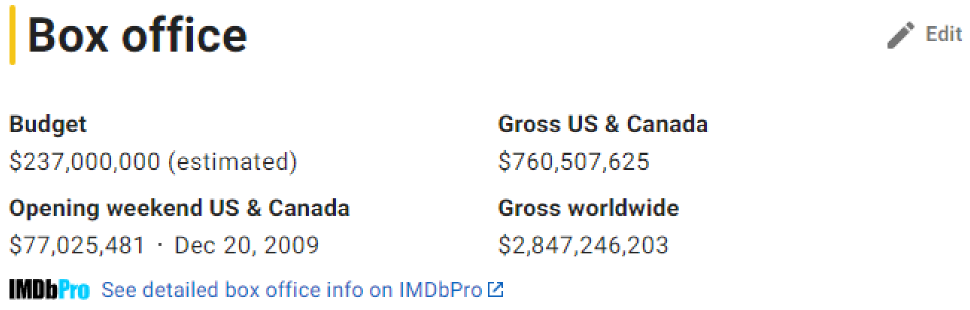
Out[45]:

	color	director_name	num_critic_for_reviews	duration	director_facebook_likes	actor_3_facebook_likes	actor_2_name	actor_1_facebook_likes	gross
0	Color	James Cameron	723.0	178.0	0.0	855.0	Joel David Moore	1000.0	76050
1	Color	Gore Verbinski	302.0	169.0	563.0	1000.0	Orlando Bloom	40000.0	30940
2	Color	Sam Mendes	602.0	148.0	0.0	161.0	Rory Kinnear	11000.0	20007
3	Color	Christopher Nolan	813.0	164.0	22000.0	23000.0	Christian Bale	27000.0	44813
4	NaN	Doug Walker	NaN	NaN	131.0	NaN	Rob Walker	131.0	NaN

II. Preprocessing

As mentioned above, there are columns not important to this project, and they are “color”, "facenumber_in_poster", and "aspect_ratio", therefore they are removed.

Secondly, when I went onto the imdb site to better understand the scale and unit of certain features, I realized that budget and gross are collected in dollar amount, but in the raw scale they will significantly outweigh the importance of other features with a different and much smaller range and scale, so I converted these columns into millions of dollars by applying the floordiv function.



Box office

Edit

Budget
$237,000,000 (estimated)

Gross US & Canada
$760,507,625

Opening weekend US & Canada
$77,025,481 · Dec 20, 2009

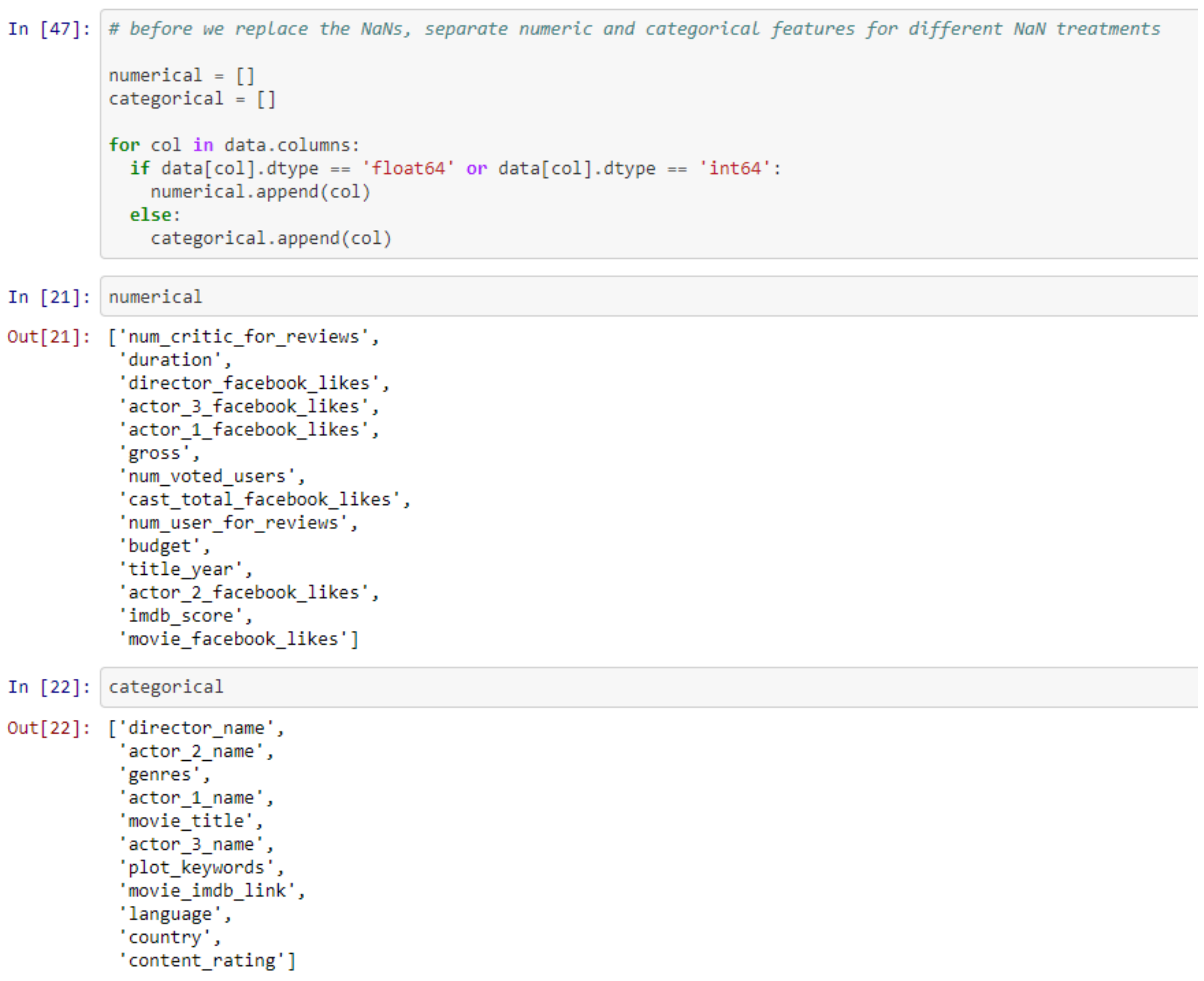
Gross worldwide
$2,847,246,203

IMDbPro See detailed box office info on IMDbPro

(Example: Avatar box office info from https://www.imdb.com/title/tt0499549/?ref_=nv_sr_srsg_0)

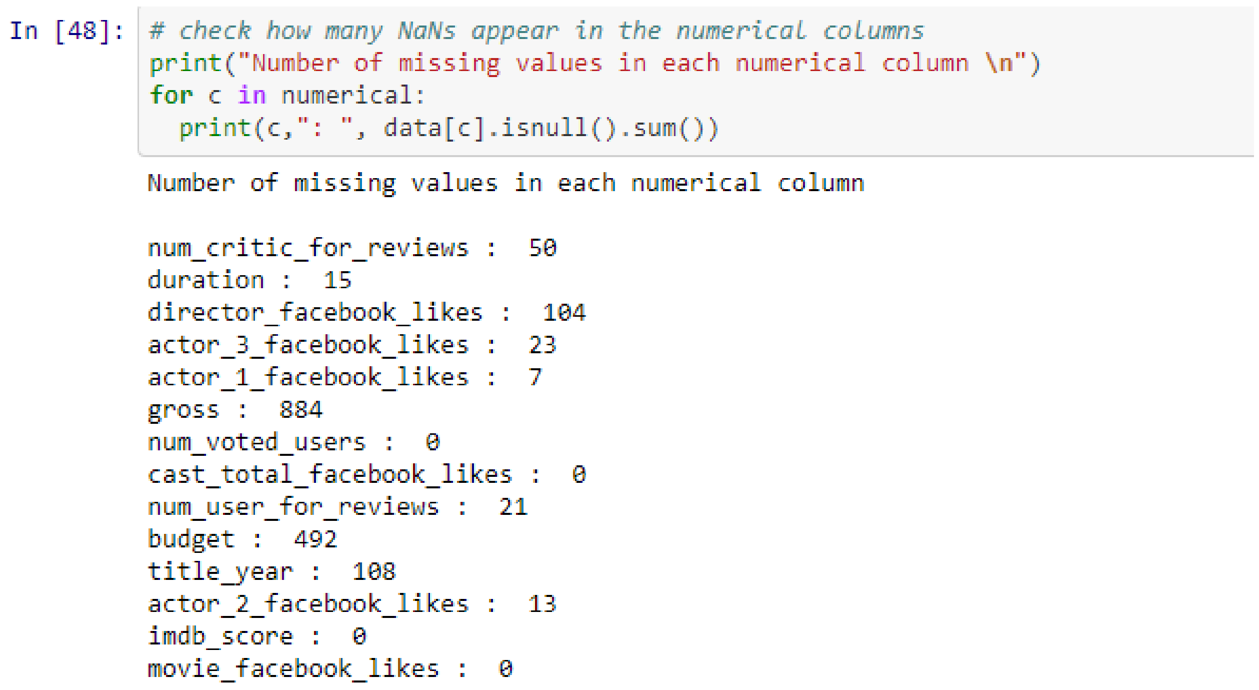
Next up is removing NaN values. There are many ways to treat NaN values, sometimes simply by removing the rows with NaN values, sometimes replacing them with other metrics for numerical features, such as median, mode, mean...etc., and for categorical features, replacing with empty string or most frequent value. We cannot be sure about which method to choose before studying the nature of the data for those features with NaN value carefully.

First, let’s divide the features into numerical and categorical. Knowing the data types of each feature (see picture 1), we can simply select those with a type of ‘float64’ or ‘int64’ as numerical, and ‘object’ ones as numerical.



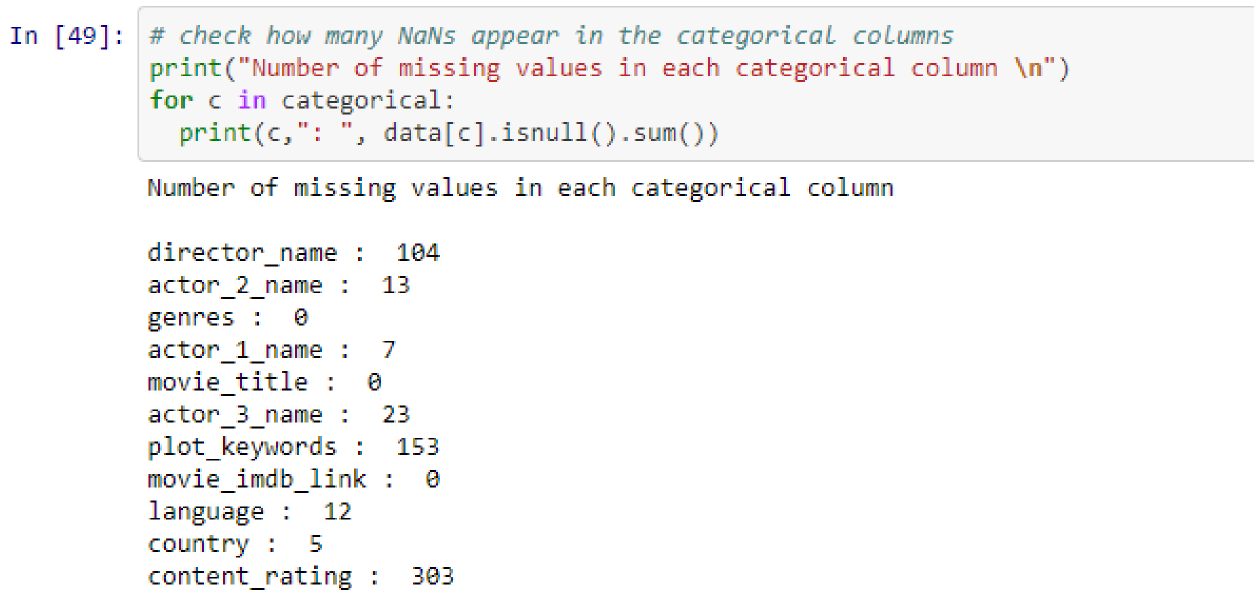
```
In [47]: # before we replace the NaNs, separate numeric and categorical features for different NaN treatments

         numerical = []
         categorical = []

         for col in data.columns:
           if data[col].dtype == 'float64' or data[col].dtype == 'int64':
             numerical.append(col)
           else:
             categorical.append(col)

In [21]: numerical

Out[21]: ['num_critic_for_reviews',
          'duration',
          'director_facebook_likes',
          'actor_3_facebook_likes',
          'actor_1_facebook_likes',
          'gross',
          'num_voted_users',
          'cast_total_facebook_likes',
          'num_user_for_reviews',
          'budget',
          'title_year',
          'actor_2_facebook_likes',
          'imdb_score',
          'movie_facebook_likes']

In [22]: categorical

Out[22]: ['director_name',
          'actor_2_name',
          'genres',
          'actor_1_name',
          'movie_title',
          'actor_3_name',
          'plot_keywords',
          'movie_imdb_link',
          'language',
          'country',
          'content_rating']
```

Then we check the count of NaNs for each feature to see whether the number is small enough and unessential to the entire population, so that removal of those rows would not be problematic, or other methods should be conducted.



```
In [48]: # check how many NaNs appear in the numerical columns
         print("Number of missing values in each numerical column \n")
         for c in numerical:
           print(c,": ", data[c].isnull().sum())
```

```
Number of missing values in each numerical column 

num_critic_for_reviews :  50
duration :  15
director_facebook_likes :  104
actor_3_facebook_likes :  23
actor_1_facebook_likes :  7
gross :  884
num_voted_users :  0
cast_total_facebook_likes :  0
num_user_for_reviews :  21
budget :  492
title_year :  108
actor_2_facebook_likes :  13
imdb_score :  0
movie_facebook_likes :  0
```



```
In [49]: # check how many NaNs appear in the categorical columns
         print("Number of missing values in each categorical column \n")
         for c in categorical:
           print(c,": ", data[c].isnull().sum())
```

```
Number of missing values in each categorical column 

director_name :  104
actor_2_name :  13
genres :  0
actor_1_name :  7
movie_title :  0
actor_3_name :  23
plot_keywords :  153
movie_imdb_link :  0
language :  12
country :  5
content_rating :  303
```

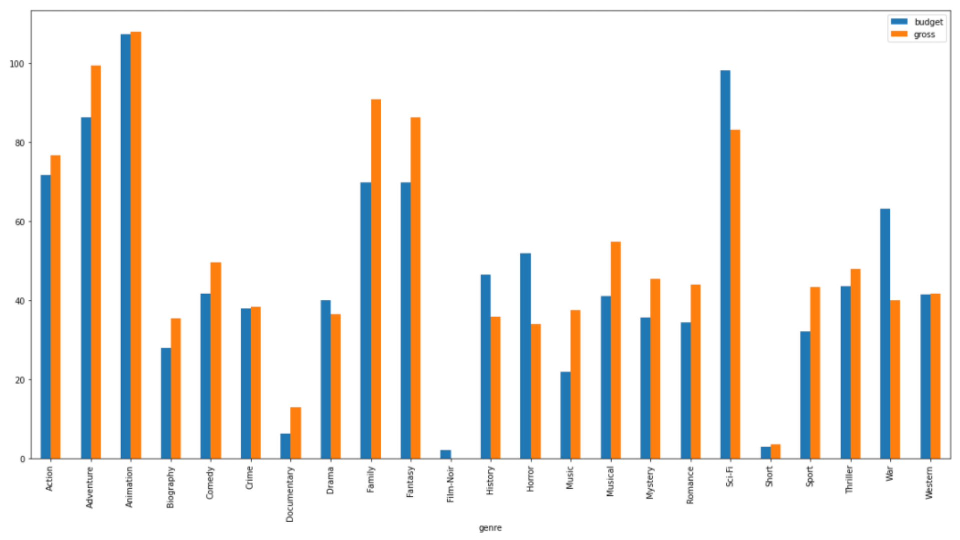
From the counts above, we can see some numerical features have large counts of missing values, such as gross and budget. The former has about 17.5% of missing data relative to the population, and the latter has close to 10%, both should definitely be handled with care and caution. On the other hand, the missing value counts for categorical features aren’t as problematic, therefore let’s talk about handling categorical missing values first, to start with the simple ones.

One idea is to cross reference other data sources unlimited to IMDB to retrieve some of these features, such as director name, actor names, plot keywords, content rating...etc.. However, I don’t think this information makes or breaks our analysis or prediction, so it’s not worth the extra effort doing so. We also can’t assume any of these values using any type of most frequent values, because the nature of their association with each movie is unique and cannot be derived, in a quantitative way.

For example, if someone is a big fan of Christopher Nolan, with strong domain knowledge, they might help filling in the missing director name for other Christopher Noolan movies, but quantitatively, it’s hard to look at 104 movies with missing director names and immediately diagnosing, “this looks like / it might be a Christopher Nolan movie”. Same goes with the other features in this list. So all of these NaN values will be replaced with “”, i.e. an empty string.

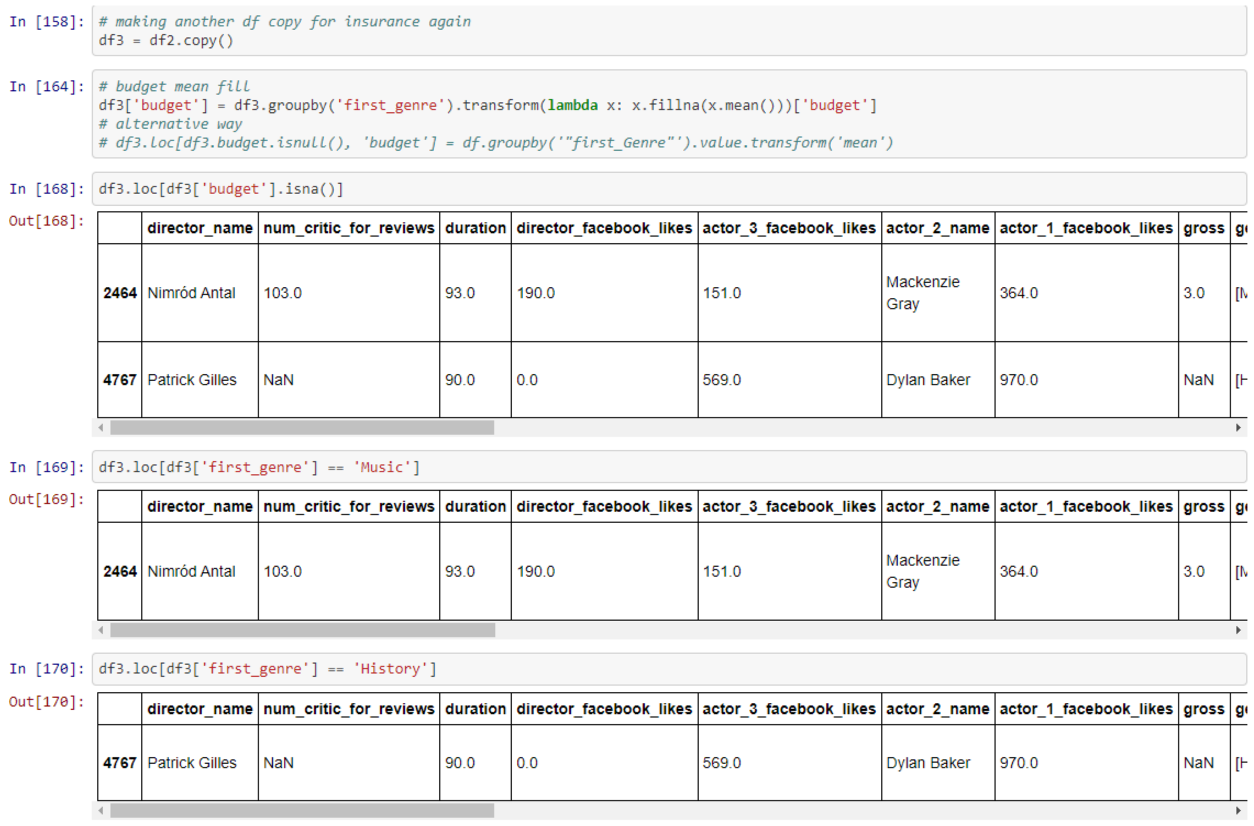
Now we can move onto handling the missing values for numerical features. Recall our observation on how budget and gross have a significant amount of NaNs, and should be treated with TLC? Here I made an assumption that the mean budget and gross is heavily influenced by genre, so we cannot simply replace missing budget and gross with the mean values of the entire population. It doesn’t make sense to compare the budget of an action film with that of a romantic comedy. Same goes with the box office of a blockbuster action film with that of a documentary.

With such an assumption in mind, it is important for us to find the mean budget and gross for each genre. Now we face some further complications. Each movie that has a “genres” value is tagged with at least one, but not limited to one genre. How do we calculate the mean values by genre when movies can repeatedly enter different genre counts? Hence I made a second assumption. The first genre listed for each movie must be the most important genre that captures the essence of this movie’s nature. One may argue that IMDB may enter genres by alphabetical order, and not by true importance sequence, however if we accept that theory, we cannot solve the current argument easily. By assuming that the first genre definitely describes a movie accurately enough, we can use this arbitrary rule to get a rough sense of the mean budget and gross across genres.



From the visualization above, we can see that the mean budget and gross grouped by genre varies greatly depending on the genre, which confirms our theory and assumption.

In the original dataframe, I created a new column called “first_genre” to capture the first genre tag in each row. Then I filled the rows with missing budget or gross values with the mean for each depending on their “first genre” value.



```
In [158]: # making another df copy for insurance again
          df3 = df2.copy()
```

```
In [164]: # budget mean fill
          df3['budget'] = df3.groupby('first_genre').transform(lambda x: x.fillna(x.mean()))['budget']
          # alternative way
          # df3.loc[df3.budget.isnull(), 'budget'] = df.groupby('"first_Genre"').value.transform('mean')
```

```
In [168]: df3.loc[df3['budget'].isna()]
```

Out[168]:

	director_name	num_critic_for_reviews	duration	director_facebook_likes	actor_3_facebook_likes	actor_2_name	actor_1_facebook_likes	gross	g
2464	Nimród Antal	103.0	93.0	190.0	151.0	Mackenzie Gray	364.0	3.0	[M
4767	Patrick Gilles	NaN	90.0	0.0	569.0	Dylan Baker	970.0	NaN	[H

```
In [169]: df3.loc[df3['first_genre'] == 'Music']
```

Out[169]:

	director_name	num_critic_for_reviews	duration	director_facebook_likes	actor_3_facebook_likes	actor_2_name	actor_1_facebook_likes	gross	g
2464	Nimród Antal	103.0	93.0	190.0	151.0	Mackenzie Gray	364.0	3.0	[M

```
In [170]: df3.loc[df3['first_genre'] == 'History']
```

Out[170]:

	director_name	num_critic_for_reviews	duration	director_facebook_likes	actor_3_facebook_likes	actor_2_name	actor_1_facebook_likes	gross	g
4767	Patrick Gilles	NaN	90.0	0.0	569.0	Dylan Baker	970.0	NaN	[H

After my initial attempt to transform all qualifying rows, I noticed that there were still two rows missing budget values, so I looked into why. Turns out that each row is in a unique genre, and that genre only has one data point, so they are not important to the population, and can be removed. Similarly, two rows remained missing with gross values, and they were removed as well.

Movies with missing year were removed too, because again year cannot be assumed.

For the rest of the numerical features, population mean was applied. The remaining dataset contains 4932 rows and 27 columns. Our preliminary data preprocessing was completed, so I saved the processed data to file in google drive, and it will be the new dataset used in the exploratory analysis in the next section.

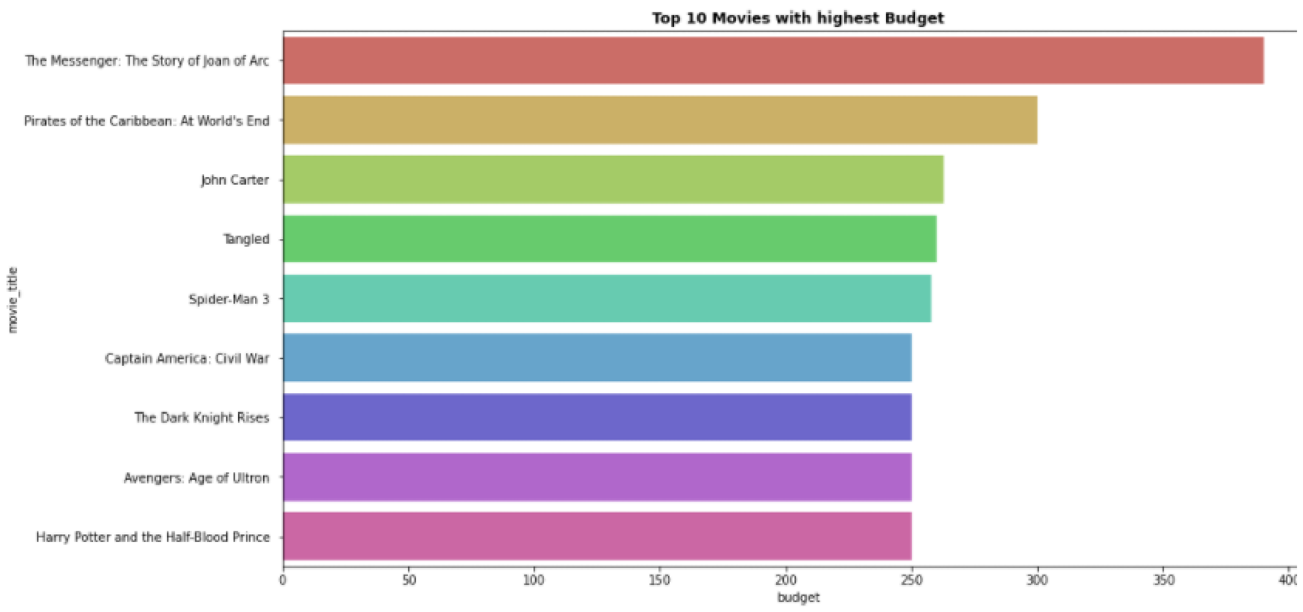
III. Exploratory Data Analysis

1. Top 10 most expensive movies

Initially when I looked up the top 10 most expensive movies made, the result was ridiculous. Not only I've never heard of any of those movies, most of them had titles that hinted that they are not English movies. Upon research on the IMDB site, a lot of non-English movies reported, or estimated budget and box office in a local currency, such as Japanese Yen.

There were about 300 movies that either were not released in English, or currencies reported were not in USD. Unfortunately I had to delete these movies, as there is no easy way to gather currency rate to USD for a lot of unique countries. After that, the result finally makes sense - seeing some of the Avengers movies on this list definitely feels right!

```python
top_budget = df1.sort_values(['budget'], ascending = False)
fig,axs=plt.subplots(figsize=(15, 8))
g=sns.barplot(y=top_budget['movie_title'][:10],x=top_budget['budget'][:10], palette = 'hls')
g.set_title("Top 10 Movies with highest Budget", weight = "bold")
plt.show()
```

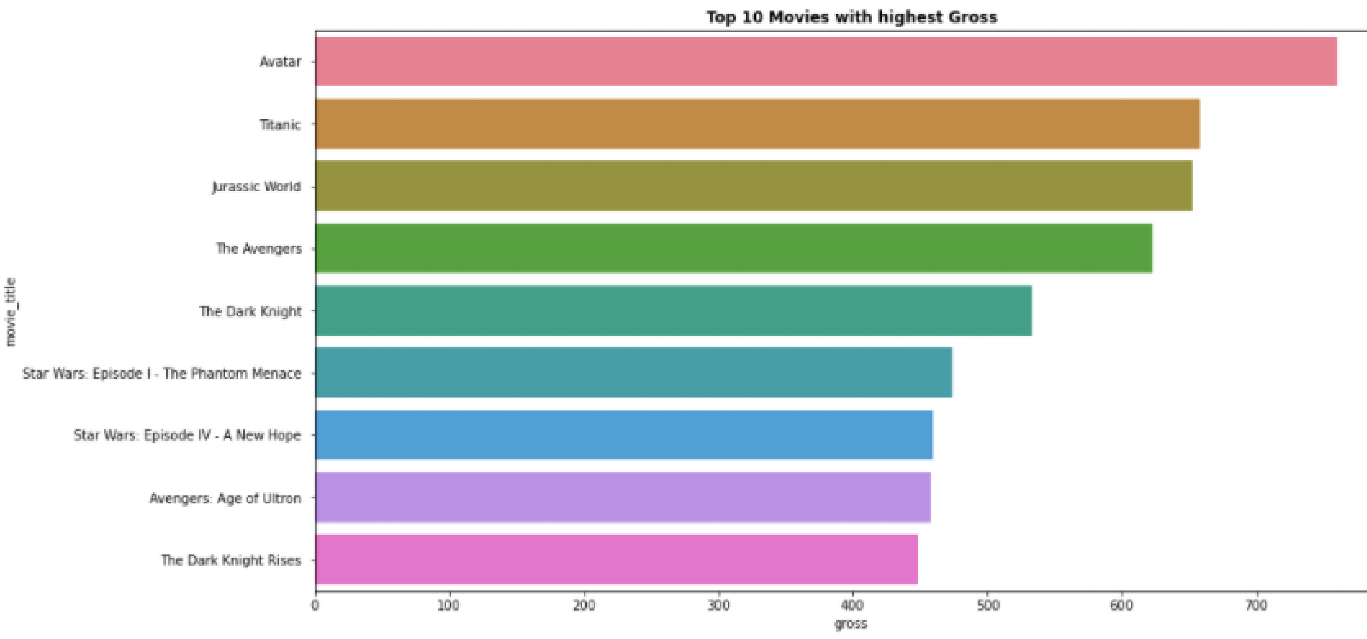


(budget by millions of $)

2. Top 10 movies by highest gross

Not much surprise in this one.

```
top_gross = df1.sort_values(['gross'], ascending = False)
fig,axs=plt.subplots(figsize=(15, 8))
g=sns.barplot(y=top_gross['movie_title'][:10],x=top_gross['gross'][:10], palette = 'husl')
g.set_title("Top 10 Movies with highest Gross", weight = "bold")
plt.show()
```

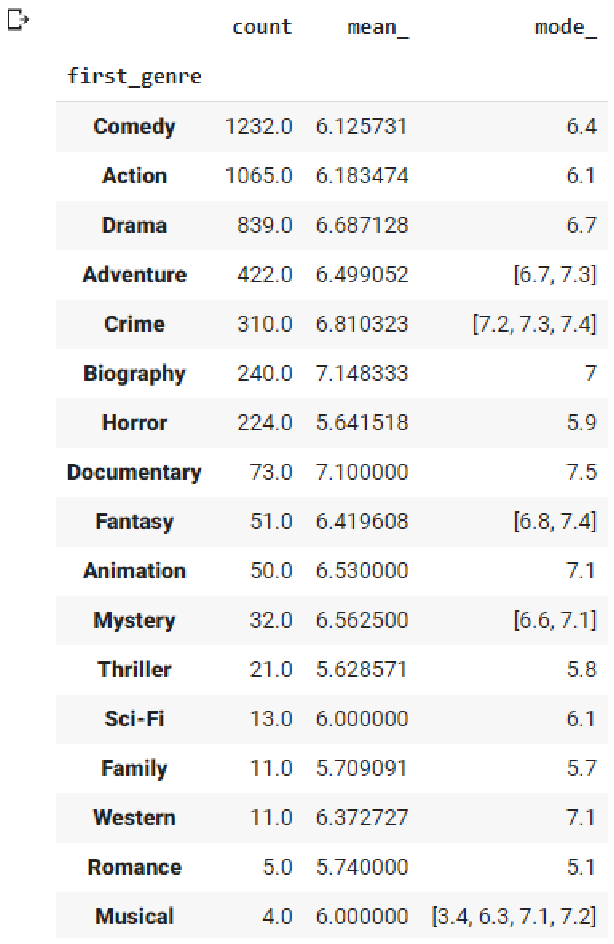


(gross by millions of $)

3. Count of movies, Mean imdb scores and mode scores by genre

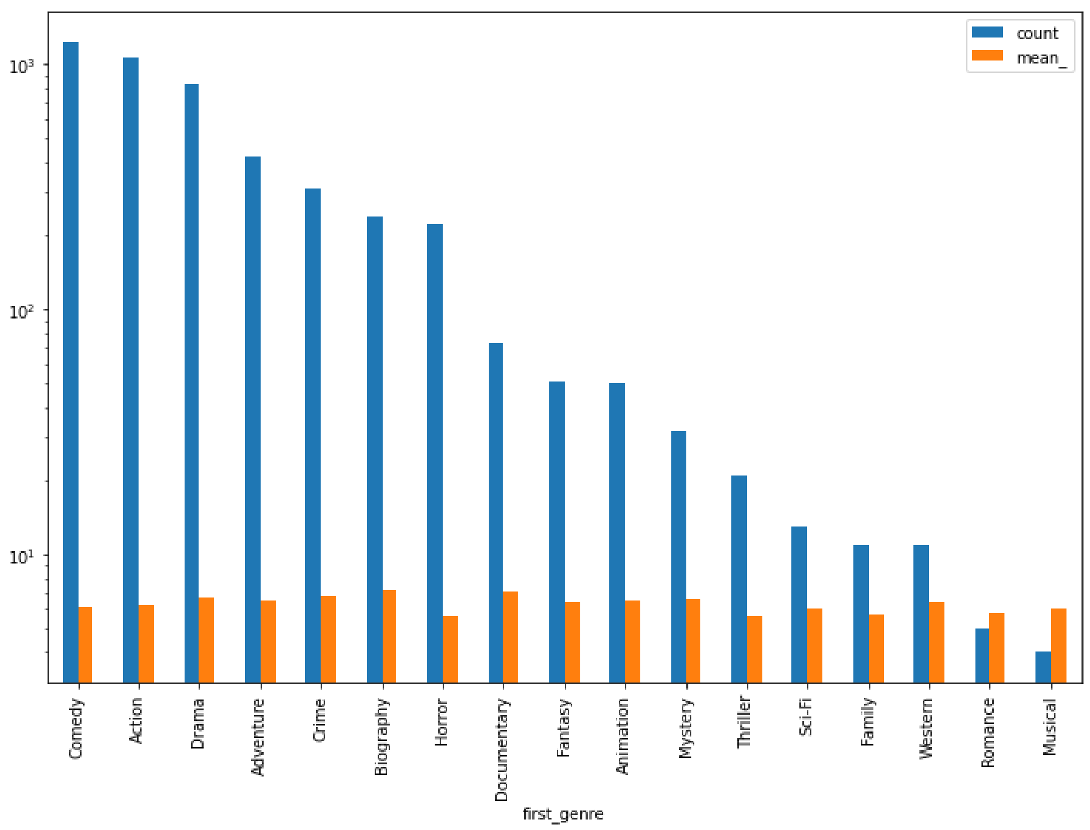
In the next few queries, we are taking a deeper look at the data segmented by genre.

Firstly, what does the distribution of the count of movies and the imdb scores look like for each genre? In this query I'm comparing the size, mean imdb scores and mode scores side by side.



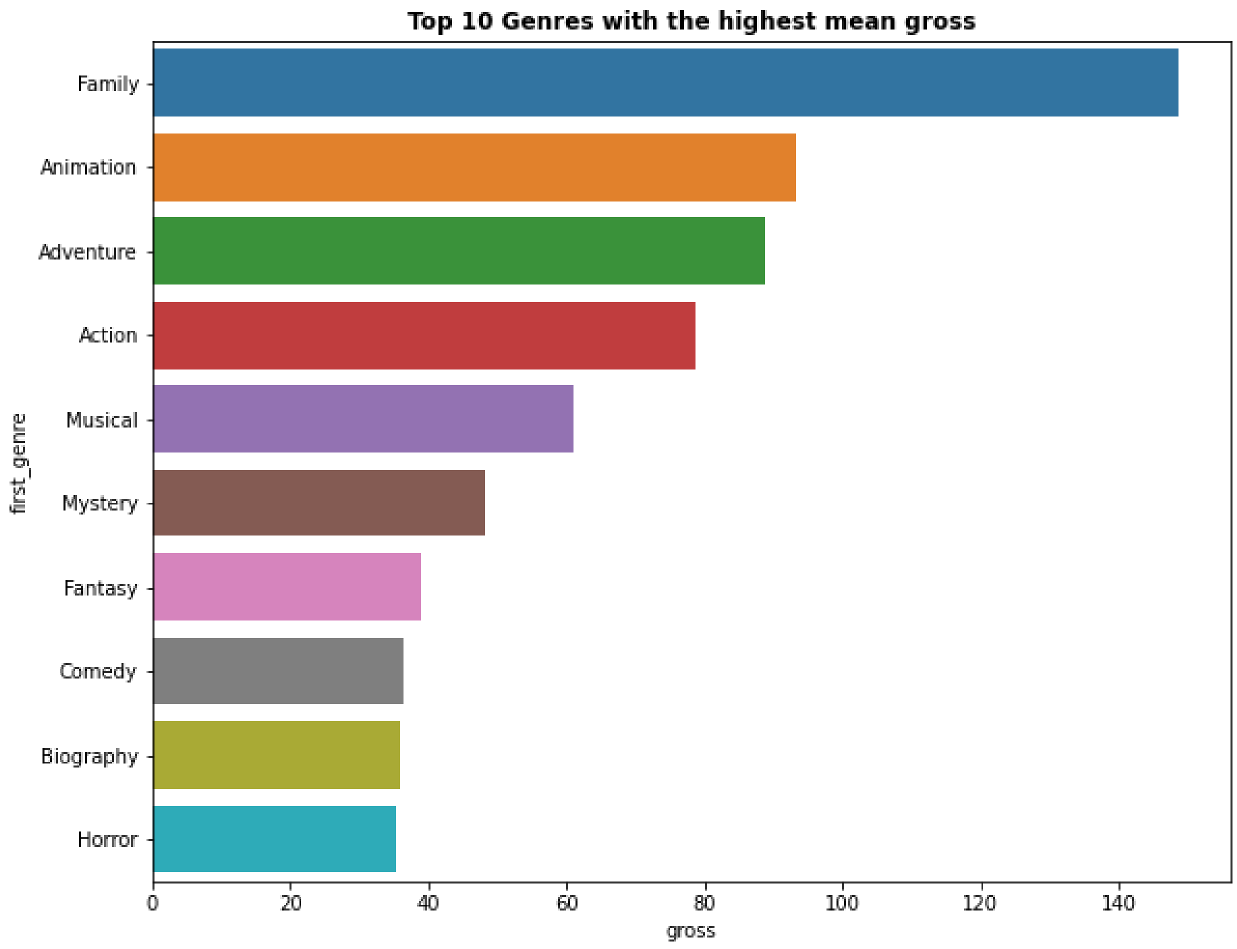
first_genre	count	mean_	mode_
Comedy	1232.0	6.125731	6.4
Action	1065.0	6.183474	6.1
Drama	839.0	6.687128	6.7
Adventure	422.0	6.499052	[6.7, 7.3]
Crime	310.0	6.810323	[7.2, 7.3, 7.4]
Biography	240.0	7.148333	7
Horror	224.0	5.641518	5.9
Documentary	73.0	7.100000	7.5
Fantasy	51.0	6.419608	[6.8, 7.4]
Animation	50.0	6.530000	7.1
Mystery	32.0	6.562500	[6.6, 7.1]
Thriller	21.0	5.628571	5.8
Sci-Fi	13.0	6.000000	6.1
Family	11.0	5.709091	5.7
Western	11.0	6.372727	7.1
Romance	5.0	5.740000	5.1
Musical	4.0	6.000000	[3.4, 6.3, 7.1, 7.2]

As we can see in the table above, the largest genres by number don't necessarily have the highest mean scores. It is understandable, as the larger the sample size, the more possible mean is "diluted" to be lower than a small sample size.



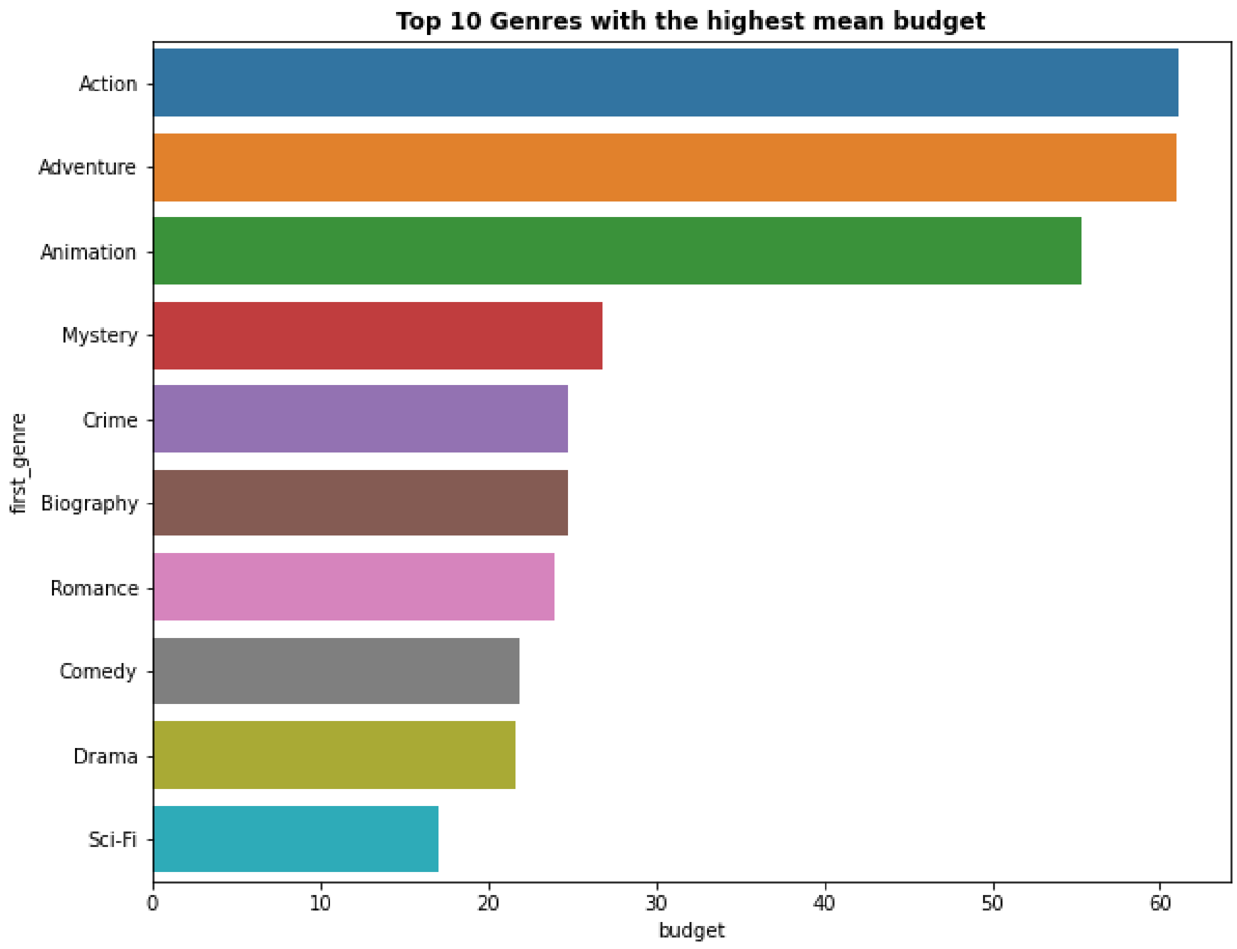
(size of genre relative to the mean imdb scores, at log scale)

4. Top 10 Genres with the highest mean gross



(gross by millions of $)

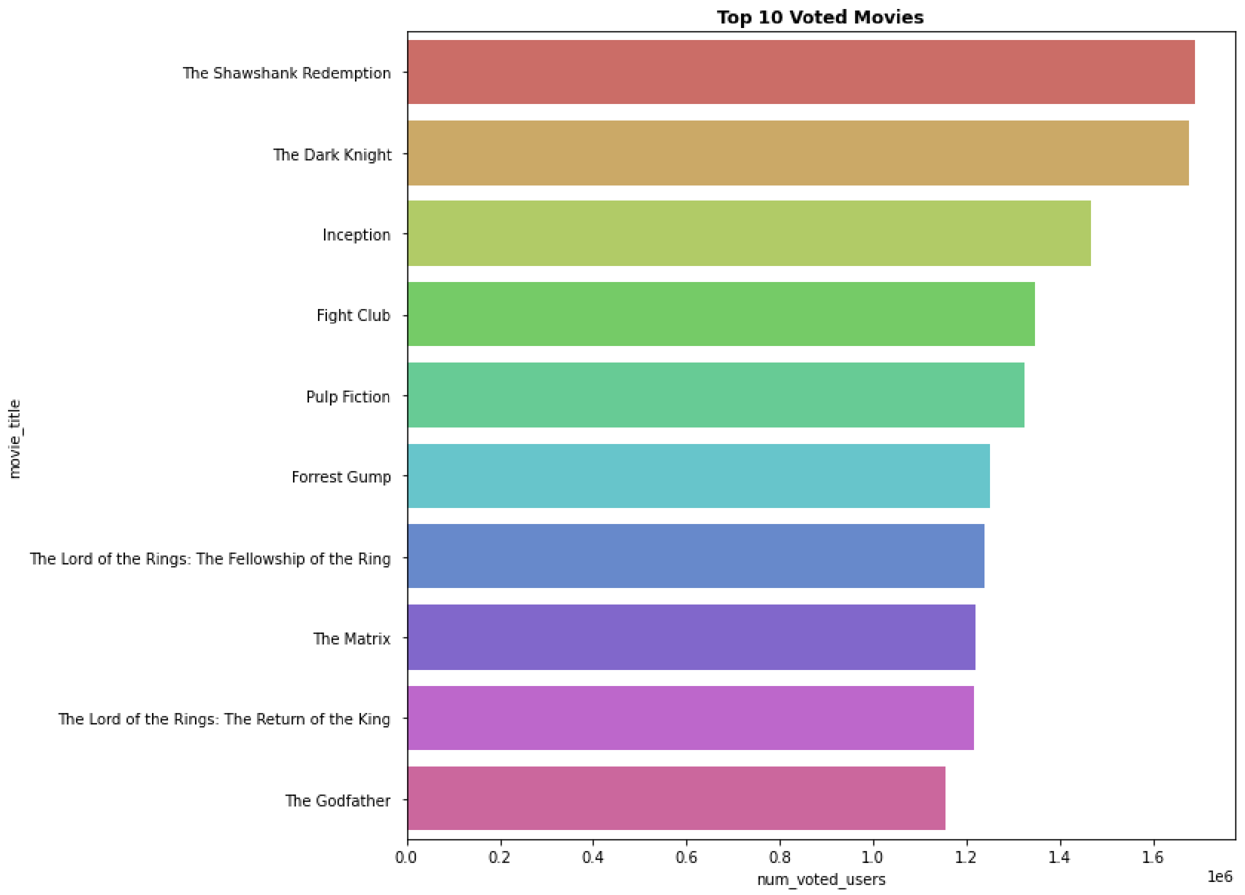
5. Top 10 Genres with the highest mean budget



(budget in millions of $)

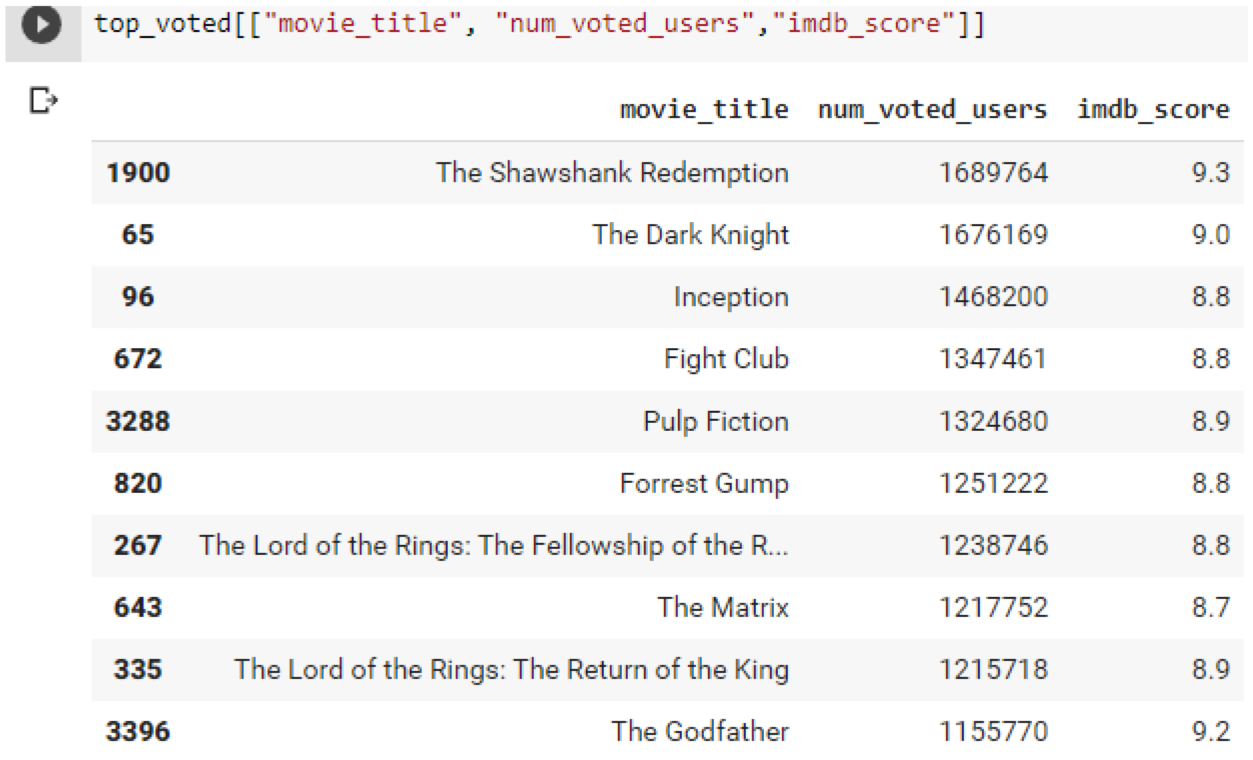
6. Top 10 Most voted movies

If we define popularity on imdb by the most voted, here's a list of the top 10 most popular movies. Any guesses?



(num of voted users by millions)

7. Top 10 most voted films vs their imdb scores



```
top_voted[["movie_title", "num_voted_users","imdb_score"]]
```

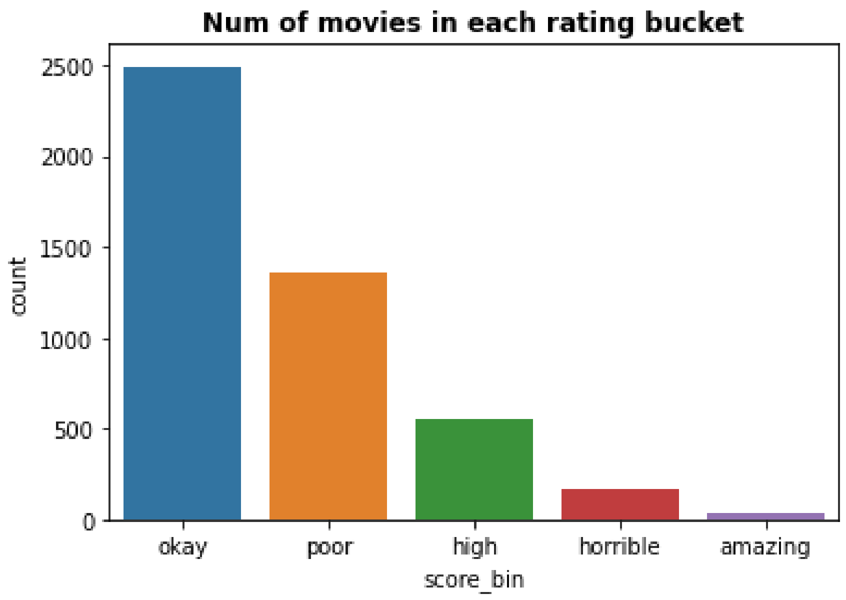
	movie_title	num_voted_users	imdb_score
1900	The Shawshank Redemption	1689764	9.3
65	The Dark Knight	1676169	9.0
96	Inception	1468200	8.8
672	Fight Club	1347461	8.8
3288	Pulp Fiction	1324680	8.9
820	Forrest Gump	1251222	8.8
267	The Lord of the Rings: The Fellowship of the R...	1238746	8.8
643	The Matrix	1217752	8.7
335	The Lord of the Rings: The Return of the King	1215718	8.9
3396	The Godfather	1155770	9.2

It's not surprising that these "classics" are also rated so highly.

8. How many movies are in each rating bucket?

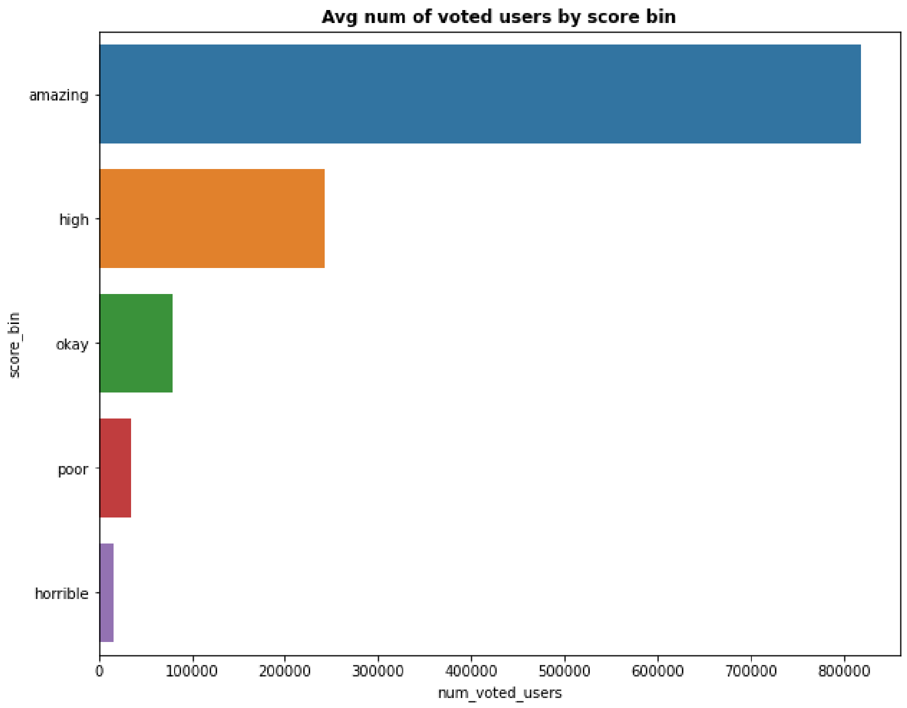
After seeing how the top 10 most voted movies all have such high ratings, it made me curious about how many movies in this dataset are regarded as terrible, decent, good, or amazing. I defined a few buckets based on my understanding on imdb rating like the following:

- horrible = 0 - 4
- poor = 4.1 - 6
- okay = 6.1 - 7.5
- high = 7.6 - 8.5
- amazing = 8.6 - 10



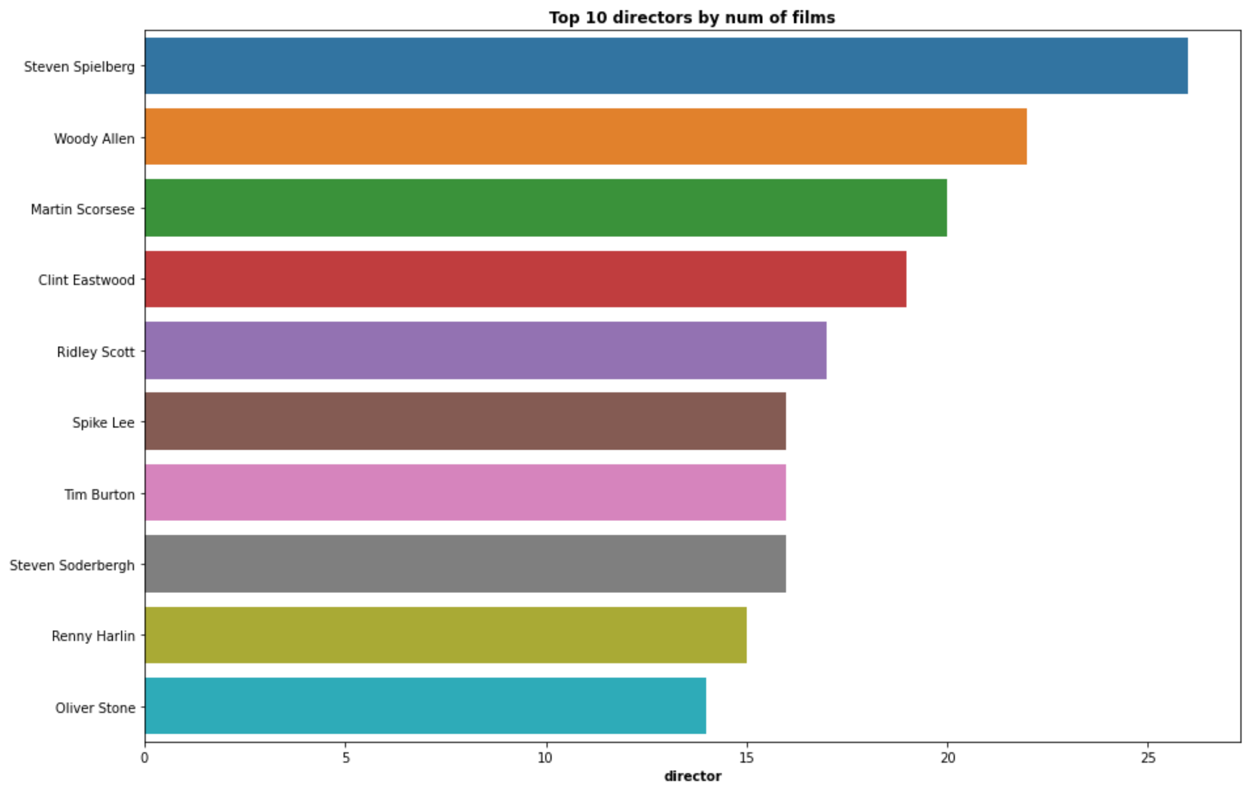
9. Avg number of users voted by score bin

From the last visualization, we can see that the number of movies in the amazing (8.6 & above) bucket is rather small, and most movies either fall in the okay or poor categories. But the top 10 most voted movies all have amazing ratings, so it made me wonder what the mean number of users voted look like for each bin.

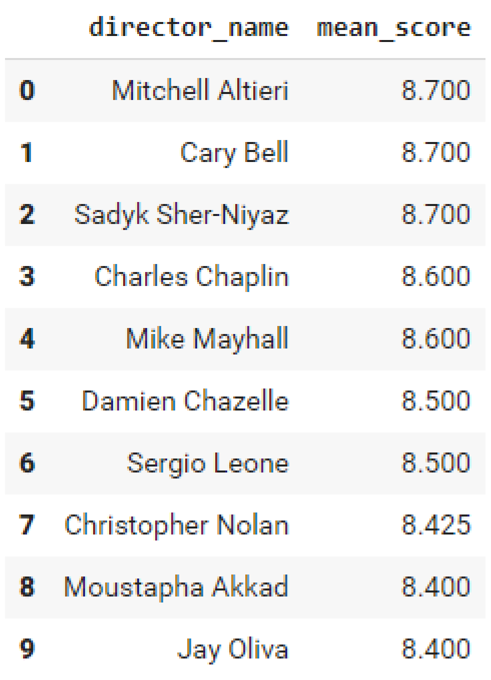


Interestingly, it’s the amazing movies that really get a large number of votes on average.

10. Top 10 directors by number of films



11. Top 10 directors by mean imdb score

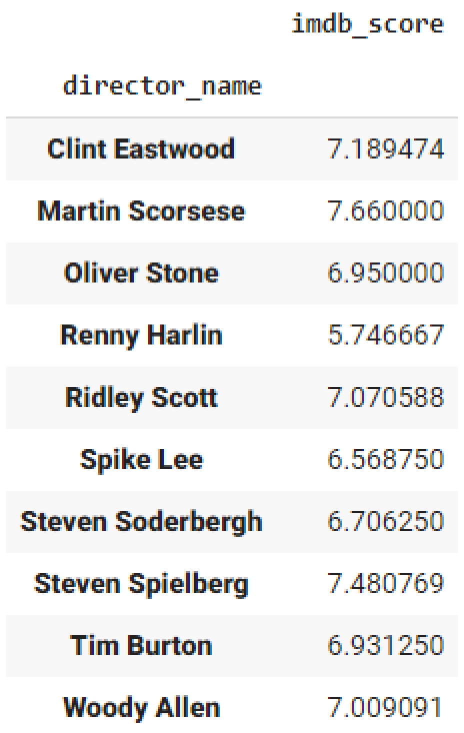


	director_name	mean_score
0	Mitchell Altieri	8.700
1	Cary Bell	8.700
2	Sadyk Sher-Niyaz	8.700
3	Charles Chaplin	8.600
4	Mike Mayhall	8.600
5	Damien Chazelle	8.500
6	Sergio Leone	8.500
7	Christopher Nolan	8.425
8	Moustapha Akkad	8.400
9	Jay Oliva	8.400

This list is different from the list of top 10 directors by count of films, which doesn't say much, because if some of these directors only have one or a few movies scoring very well, then their mean would be significantly higher than the other directors with a lot of varying movies under their belt. Let's find out the mean score for the top 10 directors by count to get more insights.

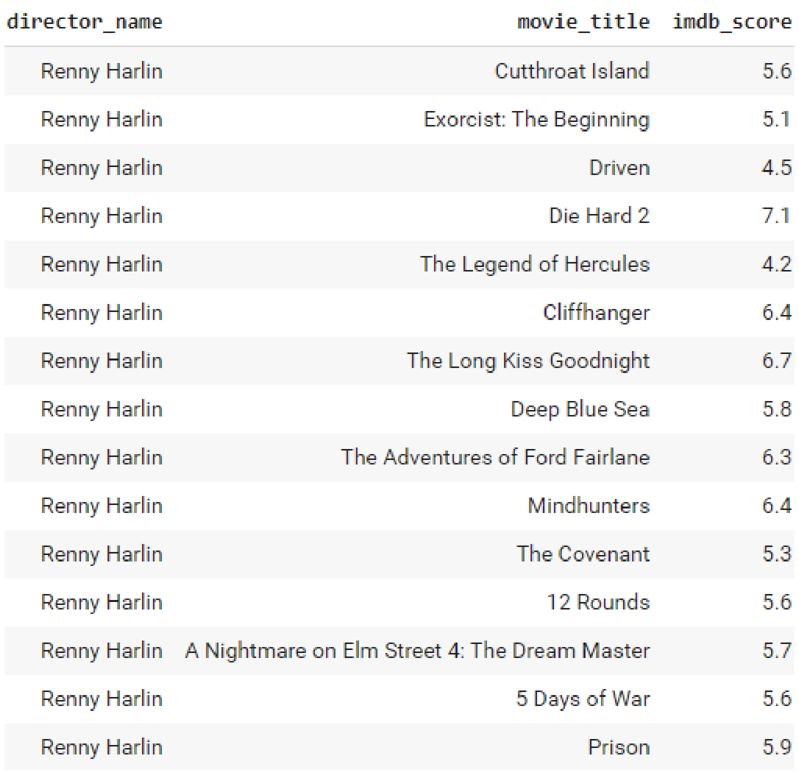
12. Avg score for top 10 directors by count

First I extracted the top 10 director names by count into a list, and filtered the dataframe by movies that were filmed by these directors only, and then selected only the director name and imdb score columns. Then calculated the mean score for each director.



	imdb_score
director_name	
Clint Eastwood	7.189474
Martin Scorsese	7.660000
Oliver Stone	6.950000
Renny Harlin	5.746667
Ridley Scott	7.070588
Spike Lee	6.568750
Steven Soderbergh	6.706250
Steven Spielberg	7.480769
Tim Burton	6.931250
Woody Allen	7.009091

Recalling our score bins, we can see that for the directors with the most films, the mean score generally falls in the okay range, with one exception - Renny Harlin lands in the poor range. I also looked at the complete list of movies directed by him and corresponding imdb scores as of 2016, shown by the table below.



director_name	movie_title	imdb_score
Renny Harlin	Cutthroat Island	5.6
Renny Harlin	Exorcist: The Beginning	5.1
Renny Harlin	Driven	4.5
Renny Harlin	Die Hard 2	7.1
Renny Harlin	The Legend of Hercules	4.2
Renny Harlin	Cliffhanger	6.4
Renny Harlin	The Long Kiss Goodnight	6.7
Renny Harlin	Deep Blue Sea	5.8
Renny Harlin	The Adventures of Ford Fairlane	6.3
Renny Harlin	Mindhunters	6.4
Renny Harlin	The Covenant	5.3
Renny Harlin	12 Rounds	5.6
Renny Harlin	A Nightmare on Elm Street 4: The Dream Master	5.7
Renny Harlin	5 Days of War	5.6
Renny Harlin	Prison	5.9

IV. Regression prediction

Finally, we are at the last portion of this project, predicting imdb scores with the best regression model. We'll discuss feature selection & engineering for different families of models, and the model selection process including base model, grid search cross validation, testing and performance evaluation.

1. Feature selection

Besides our target variable, imdb_score, the other columns I removed for a variety of reasons.

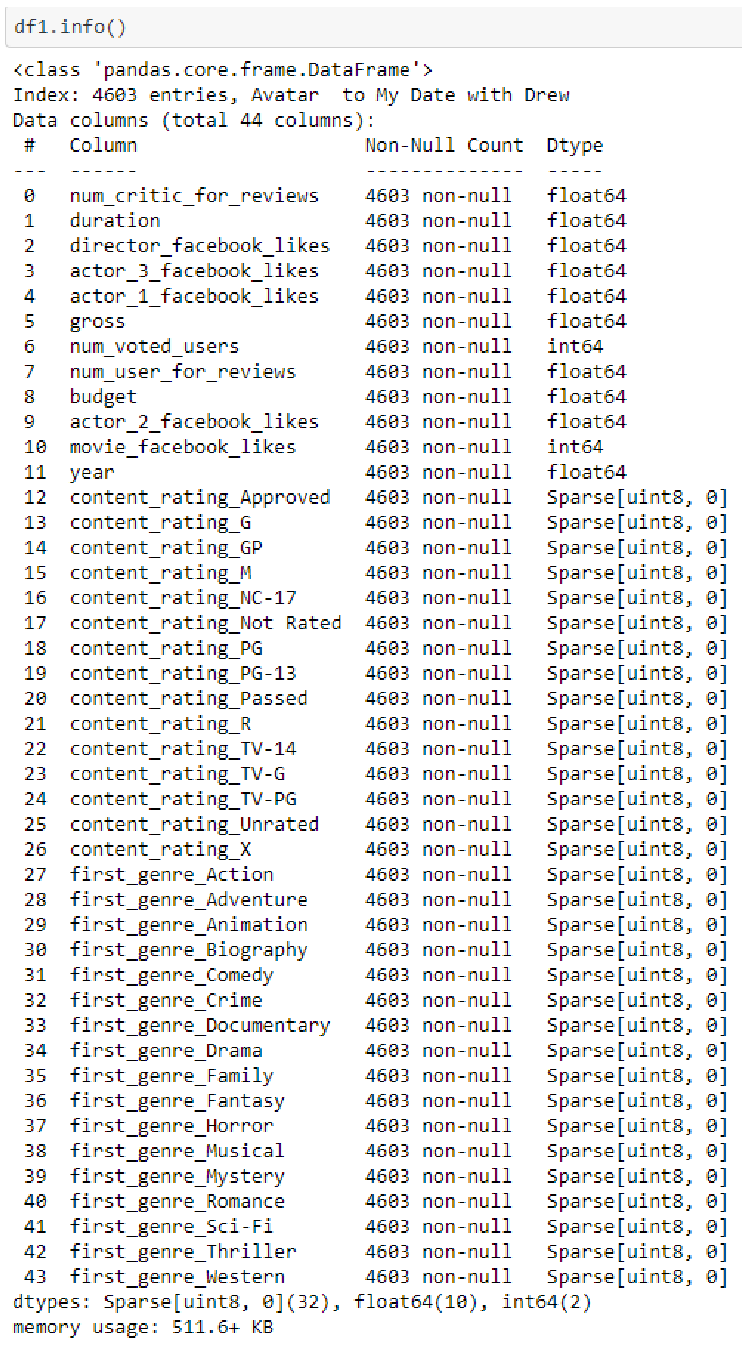
```
# features need to be removed
remove = ["director_name", "actor_1_name", "actor_2_name", "actor_3_name", "genres",
        "cast_total_facebook_likes", "language", "country", "imdb_score", "title_year",
"plot_keywords", "movie_imdb_link"]
```

Director names just won't do well in a regression model, since there are way too many unique names, so I assumed that we can get that insight from director facebook likes, which is unique in the association. Similarly, the actor names are removed while their facebook likes remain for the same logic.

In the beginning of our exploratory analysis, I mentioned that we discovered and eliminated non-English movies and those with non-USD currency, so all remaining movies are in English, which also means country isn't so important either.

Title_year is removed because it's a repeat of another column called year, which I created initially to format the release year better. Plot keywords could be saved for an unsupervised cluster analysis as a different topic, but is really difficult to deal with in the same encoding way like first_genre or content_rating.

As for content_rating and first_genre, I encoded them with Pandas get_dummies. For ease of reading, I reset the index to movie_title. At this point the dataset consists of 4603 entries, and 44 columns (see the picture below for a full list of feature names).



```
df1.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Index: 4603 entries, Avatar  to My Date with Drew
Data columns (total 44 columns):
 #   Column                     Non-Null Count  Dtype
---  ------                     --------------  -----
 0   num_critic_for_reviews     4603 non-null   float64
 1   duration                   4603 non-null   float64
 2   director_facebook_likes    4603 non-null   float64
 3   actor_3_facebook_likes     4603 non-null   float64
 4   actor_1_facebook_likes     4603 non-null   float64
 5   gross                      4603 non-null   float64
 6   num_voted_users            4603 non-null   int64
 7   num_user_for_reviews       4603 non-null   float64
 8   budget                     4603 non-null   float64
 9   actor_2_facebook_likes     4603 non-null   float64
 10  movie_facebook_likes       4603 non-null   int64
 11  year                       4603 non-null   float64
 12  content_rating_Approved    4603 non-null   Sparse[uint8, 0]
 13  content_rating_G           4603 non-null   Sparse[uint8, 0]
 14  content_rating_GP          4603 non-null   Sparse[uint8, 0]
 15  content_rating_M           4603 non-null   Sparse[uint8, 0]
 16  content_rating_NC-17       4603 non-null   Sparse[uint8, 0]
 17  content_rating_Not Rated   4603 non-null   Sparse[uint8, 0]
 18  content_rating_PG          4603 non-null   Sparse[uint8, 0]
 19  content_rating_PG-13       4603 non-null   Sparse[uint8, 0]
 20  content_rating_Passed      4603 non-null   Sparse[uint8, 0]
 21  content_rating_R           4603 non-null   Sparse[uint8, 0]
 22  content_rating_TV-14       4603 non-null   Sparse[uint8, 0]
 23  content_rating_TV-G        4603 non-null   Sparse[uint8, 0]
 24  content_rating_TV-PG       4603 non-null   Sparse[uint8, 0]
 25  content_rating_Unrated     4603 non-null   Sparse[uint8, 0]
 26  content_rating_X           4603 non-null   Sparse[uint8, 0]
 27  first_genre_Action         4603 non-null   Sparse[uint8, 0]
 28  first_genre_Adventure      4603 non-null   Sparse[uint8, 0]
 29  first_genre_Animation      4603 non-null   Sparse[uint8, 0]
 30  first_genre_Biography      4603 non-null   Sparse[uint8, 0]
 31  first_genre_Comedy         4603 non-null   Sparse[uint8, 0]
 32  first_genre_Crime          4603 non-null   Sparse[uint8, 0]
 33  first_genre_Documentary    4603 non-null   Sparse[uint8, 0]
 34  first_genre_Drama          4603 non-null   Sparse[uint8, 0]
 35  first_genre_Family         4603 non-null   Sparse[uint8, 0]
 36  first_genre_Fantasy        4603 non-null   Sparse[uint8, 0]
 37  first_genre_Horror         4603 non-null   Sparse[uint8, 0]
 38  first_genre_Musical        4603 non-null   Sparse[uint8, 0]
 39  first_genre_Mystery        4603 non-null   Sparse[uint8, 0]
 40  first_genre_Romance        4603 non-null   Sparse[uint8, 0]
 41  first_genre_Sci-Fi         4603 non-null   Sparse[uint8, 0]
 42  first_genre_Thriller       4603 non-null   Sparse[uint8, 0]
 43  first_genre_Western        4603 non-null   Sparse[uint8, 0]
dtypes: Sparse[uint8, 0](32), float64(10), int64(2)
memory usage: 511.6+ KB
```

Next up I splitted the train / validation / test datasets. In fact, I only splitted 80:20 for train and test, because I'm using the test set in the cross validation stage and will use the entire dataset as the final testing data with the best model.

```
X1 = X.copy()
Y1 = Y.copy()
```

```
from sklearn import model_selection
```

```
X_train, X_test, y_train, y_test = model_selection.train_test_split(X1, Y1, test_size=0.20,
shuffle=True)
print('training data has ' + str(X_train.shape[0]) + ' observation with ' + str(X_train.shape[1]) +
' features')
print('test data has ' + str(X_test.shape[0]) + ' observation with ' + str(X_test.shape[1]) + '
features')
```

```
training data has 3682 observation with 44 features
test data has 921 observation with 44 features
```

2. Feature Engineering

Why am I doing feature engineering after splitting the training and testing data? Because we should always hypothetically regard the test data as unseen data. If we do transformation and engineering before using the test set, there could be a data leakage problem, because the training and testing set is sampled randomly. If the model “knew” the perfectly engineered data as a whole, the training data has been contaminated.

In this project, I picked four models to evaluate, and they are KNN, Stochastic Gradient Descent (SGD), Random Forest, and Gradient Boosting tree model. For KNN and SGD, it’s important to normalize the data so feature weights will not be biased heavily by features at a much larger scale than the others. I chose the MinMaxScaler from sklearn.

```
from sklearn.preprocessing import MinMaxScaler

# fit scaler on training data
norm = MinMaxScaler().fit(X_train)

# transform training data
X_train_norm = norm.transform(X_train)

# transform testing dataabs
X_test_norm = norm.transform(X_test)
```

For Random Forest and GBT, normalization is not needed. Tree based algorithms in nature ignore the scaling problem because partitioning is done the same way regardless.

The evaluator used for all 4 models is mean squared error.

3. KNN algorithm

a. Base model

```
from sklearn.metrics import mean_squared_error
from sklearn.neighbors import KNeighborsRegressor
```

```
model = KNeighborsRegressor(n_neighbors=5, metric = 'euclidean')
print(model)

model.fit(X_train,y_train)

KNeighborsRegressor(algorithm='auto', leaf_size=30, metric='euclidean',
                    metric_params=None, n_jobs=None, n_neighbors=5, p=2,
                    weights='uniform')

pred_y = model.predict(X_test)

score=model.score(X_test,y_test)
print(score)

mse = mean_squared_error(y_test, pred_y)
print("Mean Squared Error:",mse)

0.2090375513095487
Mean Squared Error: 1.0746440825190011
```

b. Grid Search CV

The base model was not very bad, but also not great either. With grid search cross validation, we fine tune a set of hyperparameters to check the best model available for KNN.

Below is a function I wrote to print the best score, parameter set in a clean format.

```
from sklearn.model_selection import GridSearchCV

# helper function for printing out grid search results
def print_grid_search_metrics(gs):
  print ("Best score: " + str(gs.best_score_))
  print ("Best parameters set:")
  best_parameters = gs.best_params_
  for param_name in sorted(best_parameters.keys()):
    print(param_name + ':' + str(best_parameters[param_name]))

# hyper tuning K
parameters = {
  'n_neighbors':[1,3,5,7,9]
}
Grid_KNN = GridSearchCV(KNeighborsRegressor(metric = 'euclidean'),parameters,
cv=5, refit = True)
Grid_KNN.fit(X_train, y_train)
print_grid_search_metrics(Grid_KNN)
```

```
Best score: 0.19808765952194243
Best parameters set:
N_neighbors:9
```

```python
# save the best Knn model from gridsearch
best_KNN_model = Grid_KNN.best_estimator_
```

c. Validation on test set

With the best KNN model saved from the previous step, now we'll validate its performance on the testing set, to see whether we will overfit.

```python
best_KNN_model.fit(X_test, y_test)
```

```
KNeighborsRegressor(algorithm='auto', leaf_size=30, metric='euclidean',
                    metric_params=None, n_jobs=None, n_neighbors=9, p=2,
                    weights='uniform')
```

```python
pred_y_test = best_KNN_model.predict(X_test)

score=best_KNN_model.score(X_test,y_test)
print(score)

mse = mean_squared_error(y_test, pred_y_test)
print("Mean Squared Error:",mse)
```

```
0.37079140158368207
Mean Squared Error: 0.8548766102331068
```

Mean squared error improved from our base model, which means our best model didn't overfit on the testing set. Finally we'll apply it to the entire dataset to evaluate its final performance.

d. Model performance evaluation on the entire dataset

```python
best_KNN_model.fit(X1, Y1)
pred_y_all = best_KNN_model.predict(X1)

score=best_KNN_model.score(X1, Y1)
print(score)

mse = mean_squared_error(Y1, pred_y_all)
print("Mean Squared Error:",mse)
```

```
0.3786777408024073
Mean Squared Error: 0.767583111390049
```

The reason for choosing KNN as an option is because this is a small dataset, and KNN calculates really fast purely based on distance. Its disadvantages are when working with high dimensionality the distance function becomes costly, and it's sensitive to outliers and missing data. I think with 44 features, the dimensionality we have isn't significant, however distance calculation on encoded categorical features is never that ideal in my opinion. This is our first model, and we can compare its performance to the following choices.

4. Stochastic Gradient Descent Regressor

a. Base model

```
from sklearn.linear_model import SGDRegressor
clf = SGDRegressor(max_iter=1000, tol=1e-3, random_state= 42)
clf.fit(X_train, y_train)
y_pred_sgd = clf.predict(X_test)
```

```
Mean squared error:
 5.350491667481222e+38
```

From the base model I knew this is a horrible model, compared to KNN. But I wanted to see how much grid search CV can improve on MSE, especially since the default learning rate (0.0001) could be too low for the model to converge well.

b. Grid Search CV

```
param_grid = {
   'alpha': [0.0001 , 0.001 , 0.01 , 0.1 , 1 , 10 , 100],
   'learning_rate': ['constant', 'optimal', 'invscaling'],
   'max_iter': [1000, 2000, 3000]
}
GRID_SGD = GridSearchCV(clf, param_grid, refit = True)
GRID_SGD.fit(X_train, y_train)
```

```
Best score: -2.785298996915254e+30
Best parameters set:
alpha:100
learning_rate:optimal
max_iter:1000
```

Horrible, again. I did continue to validate and evaluate performance in the python notebook, but I will omit the details in this report, as we can already tell how this is possibly the worst one. This is probably because Stochastic Gradient Descent Rregressor is still a linear regression, and with the amount of features and the nature of our features we have, a linear relationship just isn't realistic.

5. Random Forest Regressor

a. Base model

As previously mentioned, tree based algorithms don't require normalization, so a copy of the original dataset was splitted again for random forest and GBT.

```
from sklearn.ensemble import RandomForestRegressor
RF_regressor = RandomForestRegressor(random_state = 42)
RF_regressor.fit(X1_train, y1_train)
```

```
RandomForestRegressor(bootstrap=True, ccp_alpha=0.0, criterion='mse',
                      max_depth=None, max_features='auto', max_leaf_nodes=None,
                      max_samples=None, min_impurity_decrease=0.0,
                      min_impurity_split=None, min_samples_leaf=1,
                      min_samples_split=2, min_weight_fraction_leaf=0.0,
                      n_estimators=100, n_jobs=None, oob_score=False,
                      random_state=42, verbose=0, warm_start=False)
```

```
base_RF_pred_y = RF_regressor.predict(X1_test)

score=RF_regressor.score(X1_test,y1_test)
print(score)

mse = mean_squared_error(y1_test, base_RF_pred_y)
print("Mean Squared Error:",mse)
```

```
0.5198268043465617
Mean Squared Error: 0.5731125939196527
```

Among the base models, random forest is already the best. Next we tuned for the best hyperparameters.

b. Grid Search CV

```
parameters = {
    'n_estimators' : [60, 80, 100, 120, 140],
    'max_features': ["auto", "sqrt", "log2"]
}
Grid_RF = GridSearchCV(RF_regressor, parameters, cv=5, refit = True)
Grid_RF.fit(X1_train, y1_train)
```

```
Best score: 0.5134058647370467
Best parameters set:
max_features:auto
n_estimators:140
```

The hyperparameters we tuned are the number of trees and the number of features to consider.

```
best_RF_model = Grid_RF.best_estimator_
```

c. Validation on test set

```
best_RF_model.fit(X1_test, y1_test)
RandomForestRegressor(bootstrap=True, ccp_alpha=0.0, criterion='mse',
                      max_depth=None, max_features='auto', max_leaf_nodes=None,
                      max_samples=None, min_impurity_decrease=0.0,
                      min_impurity_split=None, min_samples_leaf=1,
                      min_samples_split=2, min_weight_fraction_leaf=0.0,
                      n_estimators=140, n_jobs=None, oob_score=False,
                      random_state=42, verbose=0, warm_start=False)

pred_y_test_rf = best_RF_model.predict(X1_test)

score=best_RF_model.score(X1_test,y1_test)
print(score)

mse = mean_squared_error(y1_test, pred_y_test_rf)
print("Mean Squared Error:",mse)

0.9220988427331143
Mean Squared Error: 0.09297923064991469
```

The test set is really small so it makes sense that the model did extremely well on such a small sample size (1000+ entries). This is why we would evaluate the actual performance on the entire dataset.

d. Model performance evaluation on the entire dataset

```
best_RF_model.fit(X2,Y2)
pred_y_rf_all = best_RF_model.predict(X2)

score=best_RF_model.score(X2,Y2)
print(score)

mse = mean_squared_error(Y2, pred_y_rf_all)
print("Mean Squared Error:",mse)
0.9354247043691606
Mean Squared Error: 0.0797764857213796
```

Note in this code above, X2 and Y2 are deep copies of the original X and Y data. We can see that Random Forest is by far the best algorithm. Because a classic tree algorithm improved the MSE so much, I wanted to try another tree algorithm, especially an ensemble one, to see if MSE can be further improved, therefore I chose Gradient Boosting Tree regressor.

6. Gradient Boosting Tree Regressor

a. Base model

```
from sklearn.ensemble import GradientBoostingRegressor
base_GBR = GradientBoostingRegressor(random_state=42)

base_GBR.fit(X1_train, y1_train)
base_GBR_pred_y = base_GBR.predict(X1_test)

score=base_GBR.score(X1_test,y1_test)
print(score)

mse = mean_squared_error(y1_test, base_GBR_pred_y)
print("Mean Squared Error:",mse)
```

```
0.49666344765894305
Mean Squared Error: 0.6007593088035642
```

The MSE on the base GBT model isn't as good as Random Forest, but not worse by a lot.

b. Grid Search CV

```
parameters = {
    'learning_rate': [0.05, 0.1, 0.25, 0.5],
    'n_estimators' : [60, 80, 100, 120, 140],
    'max_features': ["auto", "sqrt", "log2"]
}
Grid_GBR = GridSearchCV(base_GBR, parameters, cv=5, refit = True)
Grid_GBR.fit(X1_train, y1_train)
```

```
Best score: 0.5251711024075416
Best parameters set:
learning_rate:0.25
max_features:auto
n_estimators:140
```

```
best_GBR_model = Grid_GBR.best_estimator_
```

c. Validation on test set

```
best_GBR_model.fit(X1_test, y1_test)
```

```
GradientBoostingRegressor(alpha=0.9, ccp_alpha=0.0, criterion='friedman_mse',
                          init=None, learning_rate=0.25, loss='ls', max_depth=3,
                          max_features='auto', max_leaf_nodes=None,
                          min_impurity_decrease=0.0, min_impurity_split=None,
                          min_samples_leaf=1, min_samples_split=2,
                          min_weight_fraction_leaf=0.0, n_estimators=140,
                          n_iter_no_change=None, presort='deprecated',
                          random_state=42, subsample=1.0, tol=0.0001,
                          validation_fraction=0.1, verbose=0, warm_start=False)
```

```
pred_y_test_gbr = best_GBR_model.predict(X1_test)

score=best_GBR_model.score(X1_test,y1_test)
print(score)

mse = mean_squared_error(y1_test, pred_y_test_gbr)
print("Mean Squared Error:",mse)
```

```
0.9247441691323127
Mean Squared Error: 0.0898218909896998
```

Again, the best model did really well on the testing set, though slightly worse than Random Forest. Finally, we evaluate the model on the entire dataset.

d. Model performance evaluation on the entire dataset

```
best_GBR_model.fit(X2,Y2)

pred_y__gbr_all = best_GBR_model.predict(X2)

score=best_GBR_model.score(X2,Y2)
print(score)

mse = mean_squared_error(Y2, pred_y__gbr_all)
print("Mean Squared Error:",mse)
```

```
0.7506000450775129
Mean Squared Error: 0.3081093434945819
```

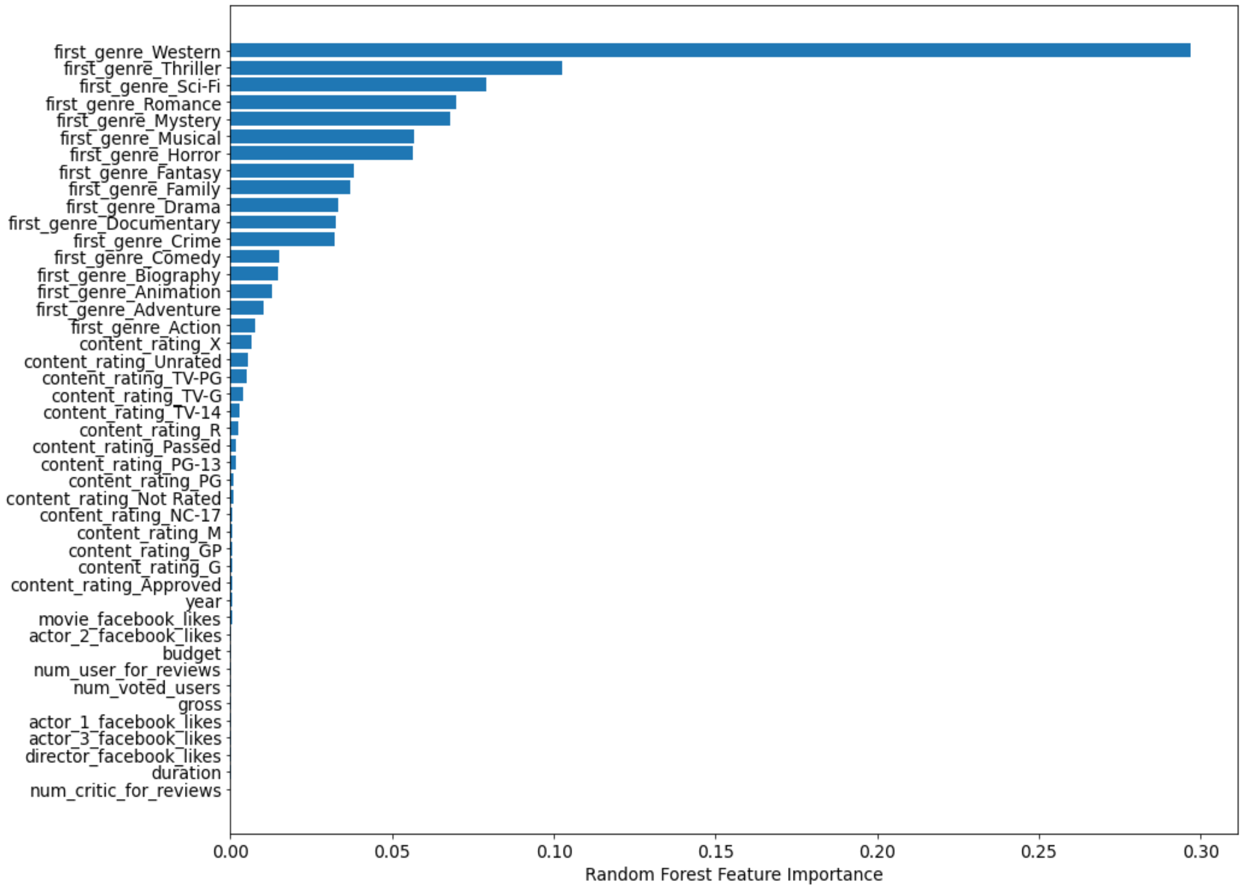
7. Best Model & Feature Importance

Our algorithm performance ranked by MSE turns out to be: Random Forest > Gradient Boosting Tree > KNN > SGD. Random Forest and Gradient Boosting Tree had similar performance, while KNN farther from the tree algorithms. SGD proved that linear regression is far from ideal for an inherently nonlinear problem. So the last step in our prediction is to compute feature importance from the best model, and rank the parameters.

There are two ways feature importance can be computed in the sci-kit learn implementation. The first is the default function that comes in the Random Forest module - mean decrease impurity, which for regression it means variance decrease.

1. RF Built-in Feature Importance

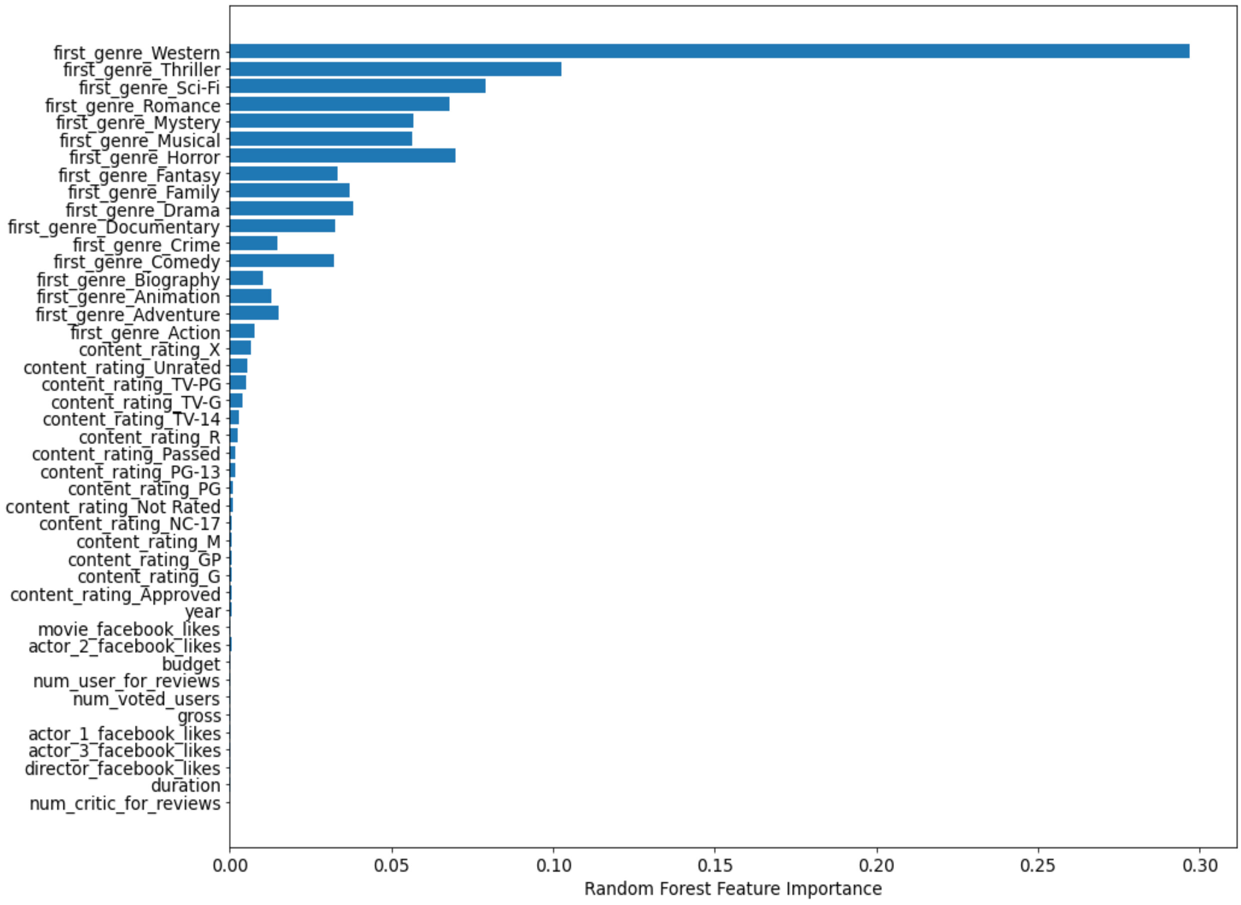
```
from sklearn.inspection import permutation_importance
from matplotlib import pyplot as plt
sorted_idx = best_RF_model.feature_importances_.argsort()
plt.barh(X2.columns, best_RF_model.feature_importances_[sorted_idx])
plt.xlabel("Random Forest Feature Importance")
```



2. Permutation based importance

"This method will randomly shuffle each feature and compute the change in the model's performance. The features which impact the performance the most are the most important ones." - a nice explanation from this article: https://mljar.com/blog/feature-importance-in-random-forest/

```
perm_importance = permutation_importance(best_RF_model, X2, Y2)
sorted_idx = perm_importance.importances_mean.argsort()
plt.barh(X2.columns, best_RF_model.feature_importances_[sorted_idx])
plt.xlabel("Random Forest Feature Importance")
```



The results from the two ways of computation are consistent. Since the most highly ranked features are dummie variables, in interpretation we can say that genre and content_rating are very important in predicting imdb scores. If we look at the rest of the features besides genre and content rating, year, movie_facebook_likes, actor_2_facebook_likes, budget, and num_users_for review are the top 5 most important features in our best model.