Predicting Movies' IMDb Ratings

George Washington University
Introduction to Data Mining - DATS 6103
Group 2
Sahara Ensley, Adam Kritz, Joshua Ting

Table of Contents

1. Introduction

Every year there are hundreds of feature films released in US theaters alone (1). In 2016 there were over 700 movies released in theaters and this number has been steadily rising since the 1980s (1). IMDb (Internet Movie Database) launched in the 1990's and has been cataloging media ever since. As of September 2021 there are almost 600,000 movies recorded on IMDb. Titles go on the database along with metadata ranging from the budget that the movie/TV show had to the individual actors and writing staff. In addition to the facts of the production, titles receive ratings by registered IMDb users that are then aggregated to a single rating for each released title in the database.

With the ever growing media industry, the question of what makes a popular movie is a lucrative one. We decided to use data from IMDb to try and model this aggregated rating score and determine if we could predict how well a movie would score based on the features of the film and its production.

In this report we will cover the following:

- Background on the dataset our feature choices, cleaning method, and models we chose to use
- Preprocessing methods we used
- Modeling procedures and the development of our best model
- The results of our model
- Conclusions we can draw from our results

2. Background

2.1 Data Description

We got our data on the IMDb catalog from Kaggle. (https://www.kaggle.com/stefanoleone992/imdb-extensive-dataset). It was provided by Stefano Leone and was updated within the last two years. The data began as four individual CSV files, Movies, Ratings, Names, and Title Principals, each containing specific information. "Movies" is the general dataset about movies, containing information like genre, duration, release date, etc. "Ratings" contains information about ratings for movies, and what age group and gender the ratings came from. In both the Movies and Ratings datasets, each movie listed contains a unique identifier that is consistent across the two datasets. "Names" contains the different names of actors and information about each of them, and each actor has their own unique identifier as well. "Title Principals" is a dataset that connects Names dataset to Movies and Ratings datasets using the unique identifiers. Title Principals also contains information about the actors' role in each movie.

After some consideration, we decided that we wanted our target variable, which we will talk about shortly, to be weighted average vote from the Ratings dataset. This meant that we no longer had use for the rest of the Ratings dataset, as the variables in it are used to

mathematically determine the weighted average vote, and could therefore not be used as predictors. Initially, we planned to include more information from the Names dataset, like average height of cast and percent male versus female of cast, however, after further inspection, the dataset appeared to be missing large amounts of data, so we felt it could not be used. This also prevented us from using any information from the Title Principals dataset, as it was only used to connect the datasets. Due to these factors, we ended up only initially including information from the Movies dataset, and weighted average vote from the Ratings dataset.

2.2 Target Variable

Initially, we had several potential target variables we were interested in. We thought about trying to predict the female average vote or male average vote for movies from the Ratings dataset, and then comparing the differences between the two. This would allow us to see what variables make males enjoy movies versus what variables make females enjoy movies. However, at this point, we felt this might be too in-depth of a study, largely because we had not found out what variables make all people enjoy movies. We narrowed down our target variable to average vote and weighted average vote.

Instead of showing the true average vote for a movie, IMDb shows the weighted average vote of a movie. This allows them to prevent a high volume of votes in a short period of time from drastically influencing a movie's score. This measure stops things like organized review-bombing from destroying a movie's IMDb rating. Because IMDb shows weighted average vote on their website as their central metric, we decided to make that our target variable. Weighted average vote can range from one to ten, and is based on user voting.

2.3 Variables Not Included

We had several decisions to make when deciding what variables to include from the Movies dataset. The easiest variables to remove were average vote and metascore, both variables that are based on the same metrics as weighted average vote, and therefore could not be used as predictors. We chose not to include votes, number of reviews from critics, and number of reviews from users as we felt these were not actually features of the movie itself. At the beginning of our project, we were also unsure what these three variables meant due to poor documentation. These variables may influence the weighted average vote, but this was not the effect we were looking to measure. Lastly, we chose not to include the language of the movie, as we felt it would be too highly correlated with the region of the movie.

2.4 Variables Included

Of the variables we included, the four numerical variables needed the least work. The four numerical variables are: duration, budget, worldwide gross income, and USA gross income. Duration was the only variable that we did not change, whatsoever. Budget, worldwide gross

income, and USA gross income we decided to adjust for inflation to prevent our results from being inaccurate.

Most of the eight categorical variables needed far more work than the numerical ones because most of the categorical variables had high cardinality and a simple One-Hot Encoding would result in an extremely sparse and high dimensional dataset. Of the categorical variables, release date was the only one that required no functional changes. Here is a list of changes for each of the other variables:

- Actors, writer, director, production company – Because we chose not to include information from the Names dataset, we could not include traditional information about actors, writers, or directors. Instead, we decided to use the frequency of actors, writers, directors, and production companies within the data as our feature. This can act as a, somewhat imprecise, proxy for 'popularity' of a member of cast or crew; the intuition being that more popular people receive more opportunities for roles in cast and production crew.
- Title and IMDb description – Since natural language processing was out of the scope of our work, we needed to find another way to use these two variables. We decided to create four categories for each of these variables: number of words, ratio of vowels to non-vowels, ratio of long words to short words, and ratio of capital letters to lowercase letters. While this effectively changes the entire context of these variables, we felt it would still be interesting to use in predictions.
- Country – We felt that including each individual country would not make sense in this context, as many countries vastly overshadow others in terms of movie production. Instead we decided to divide this into six regions, Africa, Americas, Europe, Asia, Oceania, and no country listed.
- Genre – IMDb lists three genres for each of its movies in no particular order. Due to this, we could not simply list a movie as one genre, and instead had to find a way to list all genres for each movie. We decided to simply keep each set of genres as a variable and ended up with over 700 genre combinations.

2.5 Modelling Choices

For this project, we were interested in looking into which modelling techniques could perform the best on our data. We primarily wanted to use the different models present in the SKlearn package. We planned to try a set of models from the package, and then improve on the ones that had the best initial showing using different hyperparameters for the model. From here, we could narrow it down to one specific model to use for our predictions for weighted average vote.

3. Experimental Setup - Data Preprocessing

3.1 General Preprocessing

The first step of our preprocessing pipeline was to merge the necessary datasets into a main dataframe. The initial Movies dataset contained 22 features and 85,855 observations while the initial Ratings dataset contained 49 features and 85,855 observations. Given that the unique identifiers in these datasets were the same, 'imdb_title_id' we were able to merge these two datasets seamlessly. We also obtained a dataset containing inflation information (https://fred.stlouisfed.org/series/CPIAUCNS). This dataset contained the Consumer Price Index (CPI) by year dating back to 1913. CPI is a measure of how the price of a fixed amount of goods and services changes over time. By indexing this dataset to the CPI of January 2021 we were able to get a multiplier that allowed us to convert a US dollar amount from any year into 2021 dollars. Given that we have the year a movie was produced, we were able to merge this inflation multiplier into the full movies dataset by the year column. The Names dataset was then merged with the Title_principals dataset by the unique actor identifier.

There were two minor edits that were made to the datasets before the feature cleaning began. First, the column "worlwide_gross_income" was renamed "worldwide_gross_income" to correct the typo. Second, there was a single movie that did not fit the format of the other movies, it was a TV movie released in 2019. We decided to drop this single row from the dataframe to keep the structure consistent.

The next general preprocessing step was to remove unnecessary features from our dataframe. The variables we chose to include were covered in the previous section, all others were dropped. It's important to note that this includes variables from the merged inflation dataset, once the inflation multiplier was applied to the columns pertaining to money this feature was also removed from the dataset. Finally, the data frame was split into train, test, and validation to ensure proper fitting and transformations.

The code we used to do the general preprocessing was accomplished by the following 3 wrapper functions.

Function	Output
`def clean_inflation(inflation):`	Inflation dataset with calculated multiplier
`def merge_and_clean_names(names, title_principals):`	Actors names merged with movie they appeared in and order of appearance
`def merge_and_clean_movies(movies, ratings, inflation):`	Cleaned movies data frame merged with ratings and adjusted monetary values
`def get_train_test_val(data, test_size = 0.3, val_size = 0.2):`	Data frame split into df_train, df_test, and df_val using the passed sizes.

Figure 1: General preprocessing helper functions.

3.2 Preprocessing Numerical Features

As previously mentioned, the numerical features required the least amount of cleaning.

Monetary Features

Budget, *World Income*, and *USA Income* were all cleaned in the same way. First we removed any value that was not represented in US dollars. This was determined by finding the values that did not start with "$" but instead another identifier. Once this was done the "$" was removed from the values and they were converted to integers. Finally the budget multiplier was applied to the value and the original money columns were removed from the dataset. The formula for converting the money columns is as follows:

$$AdjustedDollars = CPI_{2021}/CPI_{year} * originalDollars \quad (1)$$

Where CPI_{year} is defined as the CPI value for the year the movie was released and CPI_{2021} is defined as the CPI value for January 2021.

This was accomplished with the following helper function.

Function	Output
`def clean_money(money):`	Monetary value stripped of special characters and dropped if not in US dollars

Figure 2: Monetary values feature helper function.

3.3 Preprocessing Categorical Features

As stated earlier, most of the categorical variables had high cardinality and a simple One-Hot Encoding would result in an extremely sparse and high dimensional dataset. Therefore, we needed to come up with clever ways to encode our categorical features. The categorical features were transformed and encoded until each of them was represented by a numerical value.

Date

Date was transformed by expanding it into 3 separate columns, *date_year*, *date_month*, and *date_day*. For example "11/02/2021" would become three values of *date_year* - 2021, *date_month* - 11, and *date_day* - 02. There were no null date values so imputing was not a concern.

This was accomplished with the following helper function.

Function	Output
`def expand_date(df, col_to_expand, keep_original = False):`	Dataframe with specified column expanded into day month and year and original column dropped unless otherwise specified

Figure 3: Date features helper functions.

Country

Country was originally a string containing anywhere from 0 to 25 listed countries. Country referred to the location that the production company was located and thus where the financing originated from. The countries were listed in order of importance with the most significant country of origin being listed first. To reduce the number of features we decided to only take the first listed country and renamed it the *primary country*. From here we wanted to reduce the number of features further so we assigned each country to a *region*. The regions were determined by a list of ISO region and country codes that were taken from the UN statistics website and loaded to Github (https://github.com/lukes/ISO-3166-Countries-with-Regional-Codes). Each country was linked to a region by the country name, in the places where the name of the country in the IMDb dataset did not exactly match the name in the ISO dataset the IMDb country name was mapped to the ISO country name. This was usually a small error such as a missing “The” or incorrect capitalization. There were a few cases where the country no longer exists, such as The Soviet Union. In these cases the movie was given “No Region” as there is no consistent mapping of previous countries to current regions. Finally, each region was One Hot Encoded in the dataset, resulting in 7 columns of binary values.

This was accomplished with the following helper functions.

Function	Output
`def get_primary_country(x):`	Returns the first listed country and edits name to match ISO official country name if needed
`def to_region(df):`	Dataframe with new column mapping country to region using UN dataframe

Figure 4: Countries feature helper functions.

Genre

Genre was originally set up similar to country as a string containing 1 to 3 listed genres. These values were listed alphabetically and therefore could not be reduced to a primary genre. We transformed the string value to a list of 1 to 3 genres. These values were then transformed by taking each unique combination of genres and mapping it to a binarized string using log base 2. This binarized string was then mapped to 10 binary columns in our dataset labeled as *genre_1* through *genre_10*. There were 732 unique genre combinations, since $log_2(732) = 9.15$, all combinations were captured in these 10 columns.

This was accomplished using the following helper functions:

Function	Output
`def binary_encoder(dict_map):`	Dictionary with the original dictionary values mapped to a binary string with log base 2
`def binary_encoder_fit(df_train, col_name):`	Fits a binary encoder to the train dataframe and returns the data frame with the desired column binary encoded
`def binary_encoder_transform(df, bin_df, col_name):`	Returns dataframe with desired column transformed with the binary encoder fit to the train dataset

Figure 5: Genres feature helper functions.

Cast, Crew and Production Company

Cast, *crew*, and *production company* were all transformed in the same way. A proxy for popularity of cast and crew was approximated by taking a frequency of appearance in the training dataset per person. In cases where there were multiple actors/actresses, directors, or writers per movie, an average of the frequencies of appearance is used.

Additionally, we weighted each person's frequency by importance of role to the film. We had data on the order of importance for each person within each movie from order of 1 to 10, where 1 is highest and 10 is lowest. We assigned a weighting of $10/10$ for Order 1, $9/10$ for Order 2 and so on until $1/10$ for Order 10. We could write this weighting relationship as a systems of 2 linear equations of the form:

$$Order_p * m + b = Weight\ multiplier_p \quad (2)$$

where *p* equals person *p* in the current movie. Assigning constraint values yields:

$$1 * m + b = \frac{10}{10} \quad (3)$$

and

$$2 * m + b = \frac{9}{10} \quad (4)$$

where we can solve for m and b to change equation (2) to equation (4):

$$Order_p * (-\frac{1}{10}) + (\frac{11}{10}) = Weight\ multiplier_p \quad (4)$$

Equation (4) is what we use to calculate the weighting multiplier on each person's frequency of occurrence.

The final average weighted frequency by order of importance can then be described by the following equation

$$Frequency_{c,m} = \frac{1}{n}\sum_{p=1}^{n} Frequency_p * Weight\ multiplier_p \quad (5),$$

where *c* = actors/actress, directors, or writers,
m = the current movie,
n = total number of people in category *c* for the movie *m*, and
p the current person in the current movie *m* of category *c.*

Because there was only one production company there was no weighting necessary. Any missing frequencies were filled with the median value, this was important if the test dataset saw a novel name, that frequency was interpreted as the median.

This was accomplished using the following helper functions.

Function	Output
`def solve_linear_transformation(X, Y):`	Calculated weighted popularity by order of mention
`def get_frequencies(all_items_list, col_name):`	Frequency of mention in the names data frame
`def fit_weighted_popularity_casts(df_train, names):`	Fits the weighted frequency to the train dataset
`def transform_weighted_popularity_casts(df, popularity_fitted):`	Transforms the necessary columns using the fitted weighted popularity
`def fit_production_company_frequency(df_train):`	Fits the frequency of the production company using the train data set
`def transform_production_company_frequency(df, production_company_fitted):`	Transforms the data frame using the fitted production company frequency

Figure 6: Cast, Crew, & Production Company features helper functions.

Title and Description

Title and *description* were transformed the same way. Both features were expanded into 4 columns. The first feature extracted was the number of words, this became *title_n_words* and *description_n_words*. The ratio of long words was calculated by $\#\ long\ words\ /\ \#\ total\ words$ where $long\ words$ is defined as having more than 4 letters. This became *title_ratio_long_words* and *description_ratio_long_words*. The ratio of vowels was determined by $\#\ vowels\ /\ \#\ total\ letters$. This became *title_ratio_vowels* and *description_ratio_vowels*. The ratio capital letters was determined by $\#\ capital\ letters/\#\ total\ letters$. This became *title_ratio_captial_letters* and *description_ratio_capital_letters*. Once these 4 new features were extracted for both *title* and *description*, the original columns were dropped.

This was accomplished using the following helper functions

Function	Output
`def n_words(df, col_name):`	Data frame with calculated number of words in a string by a given column
`def ratio_long_words(df, col_name, n_letters):`	Data frame with calculated ratio of long words (defined by parameter) in a string in a given column
`def ratio_vowels(df, col_name):`	Data frame with calculated ratio of vowels in a string in a given column
`def ratio_capital_letters(df, col_name):`	Data frame with calculated ratio of capital leters in a string in a given column

Figure 7: Title and Description features helper functions.

Final Steps

Following these feature transformations any missing values were imputed by filling the missing values with the mean of the feature in the training dataset. Finally, all the features and the target were scaled using a standard scaler fitted to the train dataset and applied to the test dataset.

The preprocessing as a whole was accomplished using these two wrapper functions.

Function	Output
`def autobots_assemble(df_train, df_test, df_val, names, target):`	Transformed df_train, df_test, and df_val. Along with the scaler used for the target variable and the untransformed data frame for plotting purposes. Called by the main preprocessing function.
`def preprocess(ratings, movies, names, title_principals, inflation):`	Main wrapper for the preprocessing. Returns transformed data split into train test and validation

Figure 8: Preprocessing wrapper functions.

4. Exploratory Data Analysis

All code to recreate the following plots and statistics can be found in the file TestEDA.py located in the main Code folder of our repository.

Following transformations we did exploratory data analysis to see how our data was structured and if any initial patterns appeared before attempting to model. Beginning with our target variable, weighted average vote, we can get the distribution of values as follows.

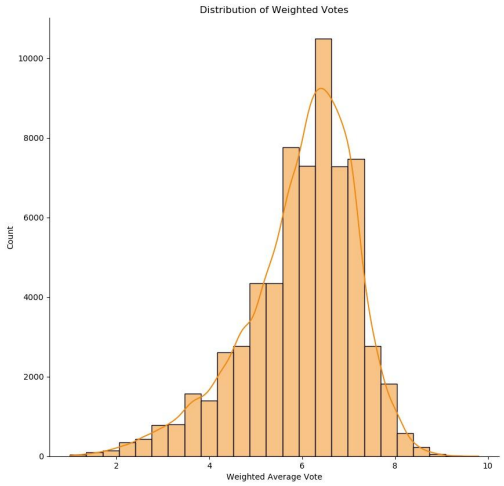

Figure 9: Distribution of *Weighted Average Votes*.

This highlights the normal structure of our dependent variable but we did note that it is slightly skewed towards votes above 5 ($\bar{x} = 5.96, \sigma = 1.19$).

Looking next towards features we can plot the distributions of our monetary columns. Due to the skew towards large values we plotted it on a log scale to highlight the normal distribution.

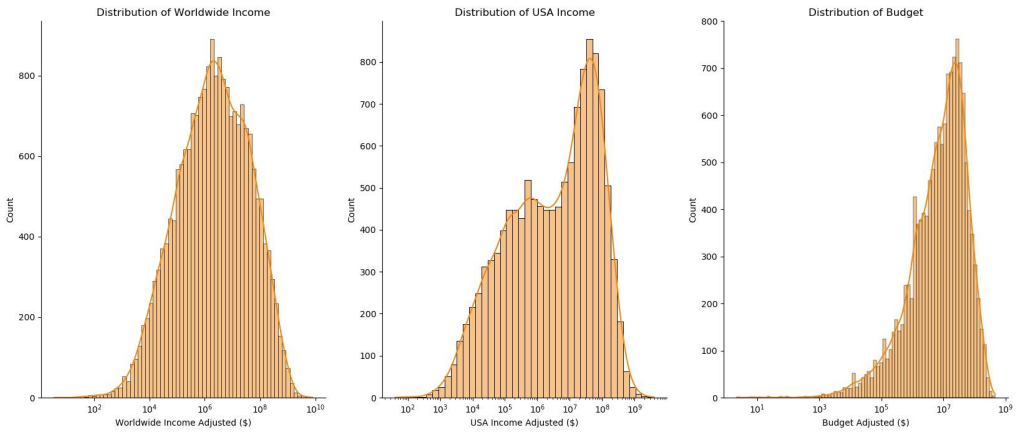

Figure 10: Distributions of *Worldwide Income, USA Income,* and *Budget*.

The relatively normal shape of these variables, worldwide income ($\bar{x} = 42461743$, $\sigma = 155374066$), USA income ($\bar{x} = 39540564$, $\sigma = 103658832$), and adjusted budget ($\bar{x} = 25411175$, $\sigma = 39822572$), was encouraging for their use in the model. Next we can plot the duration distribution.

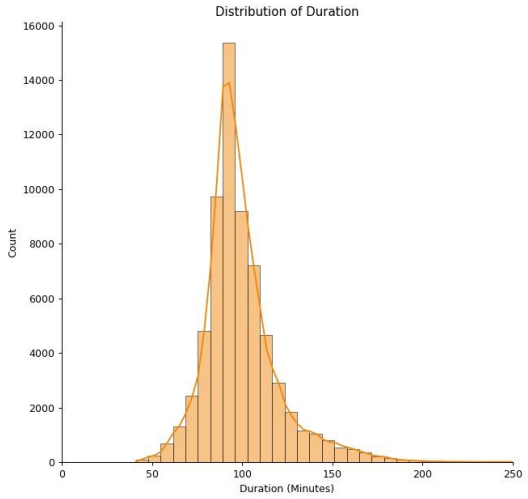

Figure 11: Distribution of *Duration*.

Similarly this variable was normally distributed ($\bar{x} = 99.66$, $\sigma = 22.71$). Next we looked at the frequency of movies by region.

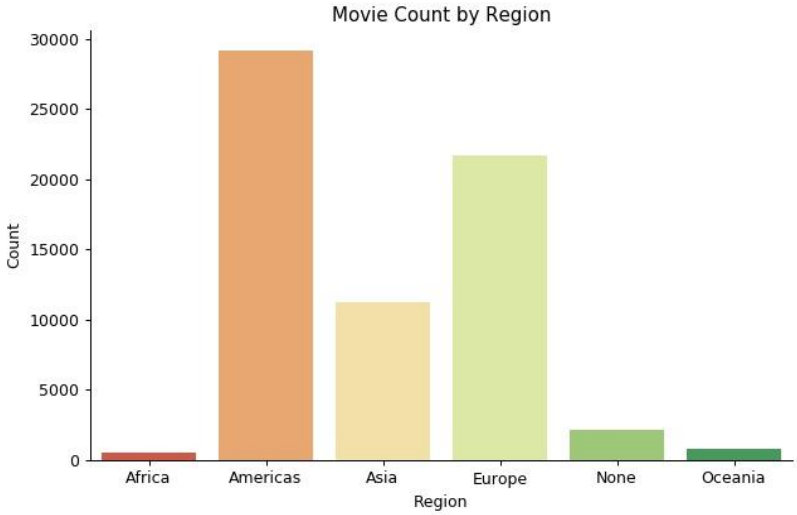

Figure 12: Movie count by region.

It's clear that there is a class imbalance with the Americas, Asia, and Europe accounting for the vast majority of movies on the IMDb database. This is likely due to these regions having the three largest gross domestic products (GDPs) of the world, thus they finance the most movies.

We next looked at low level interactions between our features and the weighted average vote.

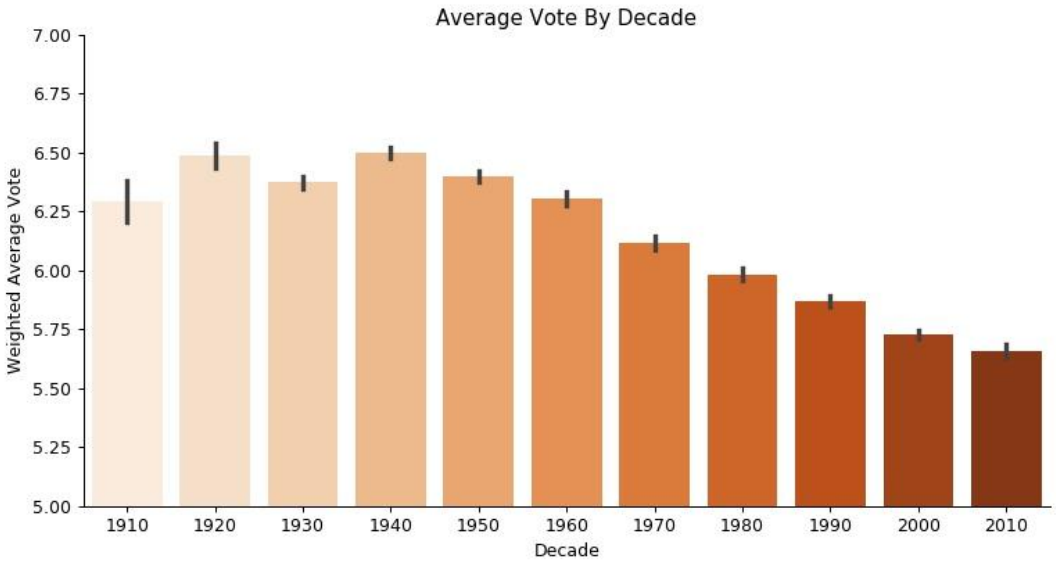

Figure 13: Distribution of votes by decade.

By plotting the vote by decade we can see that it appears the average vote is declining as the movies become more modern. We ran a One-Way ANOVA and confirmed that the decade did significantly affect the weighted average vote of a movie ($p = 0.0$)

Next we looked at the interaction between the monetary values and the weighted average vote.

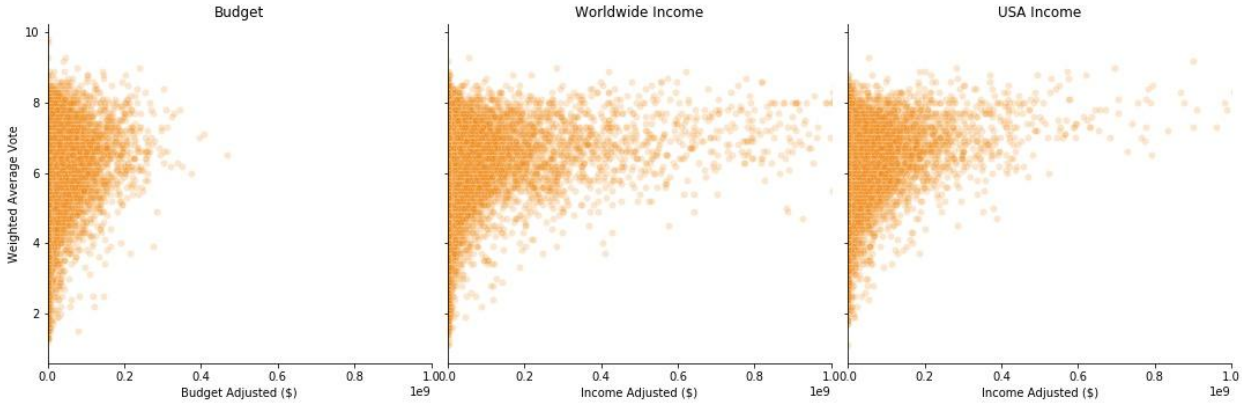

Figure 14: Interaction of monetary values and *Weighted Average Vote*.

In this graph we can see that there is a slight positive trend between the average vote and the budget ($pearson's\ r = 0.23$), worldwide income ($pearson's\ r = 0.16$), and usa income ($pearson's\ r = 0.18$). Again this boded well for these features used in the model.

The final EDA plot that we looked at was a full correlation matrix between all our numeric transformed variables.

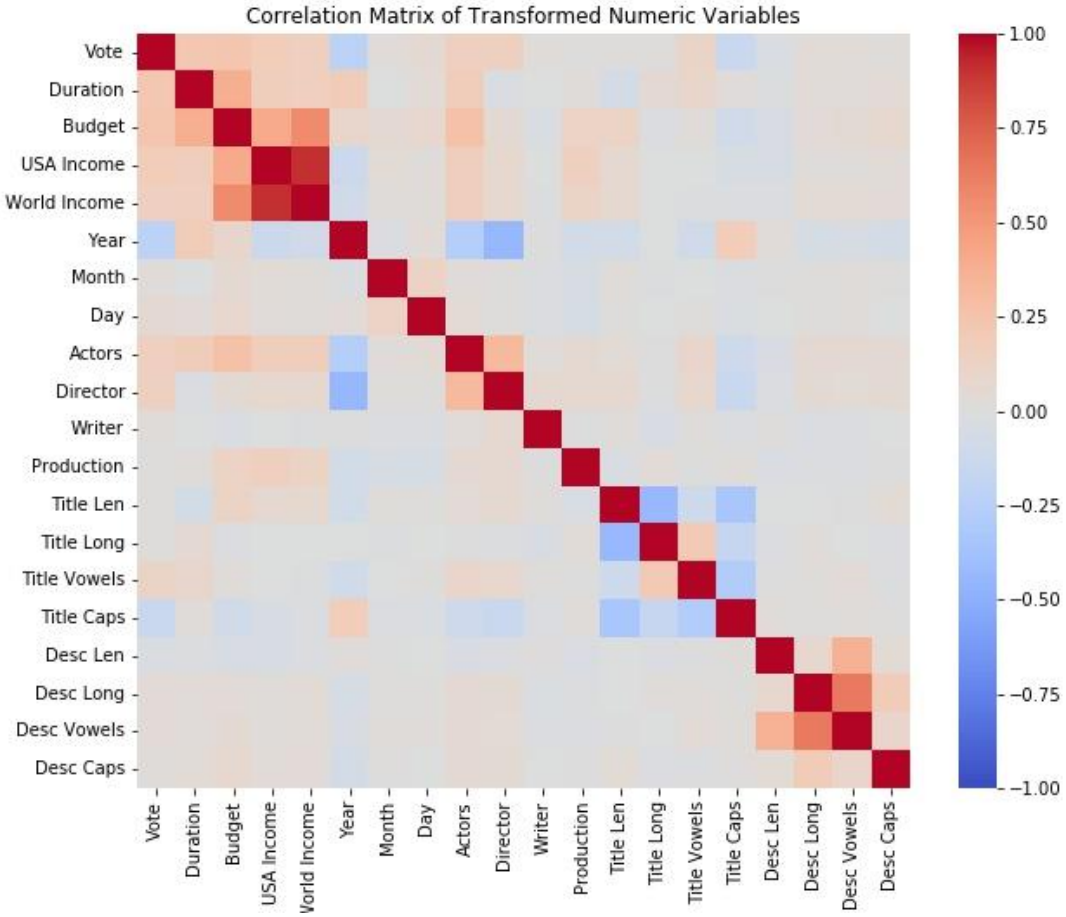

Figure 15:Correlation matrix of numerical features.

This confirmed our findings from the previous plots that there is a slight positive correlation between the average vote and the monetary features, and a slight negative correlation with year. After this EDA we felt comfortable moving to modeling the weighted average vote of the movies.

5. Experimental Setup - Modeling

5.1 Introduction

Our objective in modeling was to create a model that fits relatively well to our dataset and would allow us to predict a satisfiable *weighted average IMDb rating*. Our high level strategy for the modeling phase was the following:

1. **Base Models**: Test out a handful of regression models using default SKlearn's parameters.
2. **Hyperparameter Tuning I**: Perform gridsearch validation on top 3 performers from step (1) using a select handful of hyperparameters.
3. **Hyperparameter Tuning II**: Select top 1 performing model from gridsearch validation for even further refined hyperparameter tuning.
4. **Retrain and Test**: Choose the best combination of hyperparameters for the best performing model from step (3) as our final model and retrain on the entire training and validation dataset then predict on test dataset.

5.2 Code Setup and Architecture

Our code for modeling is saved in *../Final-Project-Group2/Code/* directory. There are three Python scripts, written in python3, where modeling is performed:

1. *models_helper.py*
2. *models.py*
3. *main.py*

1. *models_helper.py*

This Python file is the backbone to our modeling process. It holds the majority of objects and methods used for the modeling phase. It consists of the following 4 classes.

```
class Dataset:
class Model:
class ModelTuner(Model):
class Plotter:
```

The **Dataset** class represents our data and wraps our train, validation, and testing datasets into one object and has several methods to help with general data manipulation and returning specific objects related to our dataset. The table below shows a summary of these methods.

Method	Description
`__init__(self, train_df, test_df, val_df, random_seed, label)`	`Init method, set attributes, set numpy random seed`
`get_train_test_val_dfs(self)`	`Method to get a train, test, val as pd dataframes`
`split_features_target(self)`	`Method to split train, test, val data to features and target per dataset`
`data_as_arrays(self)`	`Method to convert data from pandas to np arrays/matrices for training`
`get_data_as_arrays(self)`	`Method to return data as np arrays`

`get_train_val_predefined_split(self)`	`Method to combine train and validation dataset into one and set predefined split for CV`

Figure 16: Dataset class methods.

An instance of this object can be created in the following ways.

```
data = Dataset(df_train, df_test, df_val, random_seed, label='weighted_average_vote')
```

The **Model** class represents our model and it wraps much of the functionalities to train, calculate error, etc into one object. The table below shows a summary of these methods.

Method	Description
`__init__(self, random_seed, train_x=None, train_y=None, val_x=None, val_y=None, test_x=None, test_y=None, name=None, target_scaler=None)`	`Init method for Model class`
`save_model(self, filename, model)`	`Method to save object model`
`load_model(self, filename)`	`Method to load object model`
`evaluate(self, test_x=[])`	`Method to evaluate model on test data`
`train(self)`	`Method to train model`
`get_val_score(self)`	`Method to get score for this estimator`
`get_params(self)`	`Method to get parameters for this estimator`
`get_model_nickname(self)`	`Method to return the model nickname assigned during object instantiation`
`get_model(self)`	`Method to return the model for object`
`onstruct_model(self, model)`	`Method to construct model (instantiate sklearn model)`
`set_score(self, score)`	`Method to set the score used for this model`
`get_error_in_context(self, val_x=[], val_y=[])`	`Method to get the error of model in context to target`
`get_error(self, val_x=[], val_y=[])`	`Method to get the error of model`
`set_params_to_tune(self, params_dict)`	`Method to set the paramaters to test for tuning the model`
`set_params(self, params_dict)`	`Method to set parameters to model`

Figure 17: Model class methods.

An instance of this object can be created in the following ways.

```
linear_model = Model(random_seed, train_X, train_Y, val_X, val_Y, test_X, test_Y,
'linear_sgd', target_scaler=ss_target)
```

The **ModelTuner(Model)** class is a child class of the **Model** class and represents an object to help tune the hyperparameters of the parent model object. The table below shows a summary of these methods.

Method	Description
`__init__(self, path, random_seed, train_x, train_y, test_x=None, test_y=None, name=None, target_scaler=None, ps=None, models_pipe=None, params=None)`	`Init method for ModelTuner, child of Model`
`make_directory(self)`	`Helper method to make directory path if not created`
`do_gridsearchcv(self, save_results=True, validation_curves=True, learning_curves=True)`	`Method to perform GridSearchCV to tune hyperparameters of select models`

Figure 18: ModelTuner(Model) class methods.

An instance of this object can be created in the following ways.

```
gridsearchcv = ModelTuner(get_repo_root() + '/results/', random_seed, X_train_val, Y_train_val, test_x=test_X, test_y=test_Y, name='gridsearchcv', target_scaler=ss_target, ps=ps, models_pipe=models, params=param_grids)
```

The **Plotter** class represents a plotting object with functionality to help make the different plots we needed for modeling. The table below shows a summary of these methods.

Method	Description
`__init__(self, path, name, savename)`	`Init method for Plotter`
`model_comparison(self, score_dict, score, saveplot=True, show=False, alt=0)`	`Method to plot a bar chart of models' avg scores`
`learning_curves(self, lc_results, saveplot=True, show=False, alt=0)`	`Method to plot learning curves from CV results`
`validation_curves(self, vc_results, saveplot=True, show=False, alt=0)`	`Method to plot validation curves from GridsearchCV results`
`most_important_features(self, train_df, model, saveplot=True, show=False, alt=0)`	`Method to perform analysis to get most important features from RandomForest and plot bar chart.`
`s_random_bar(self, scores_dict, saveplot=True, show=False, alt=0)`	`Method to plot a bar chart of models' avg scores`

Figure 19: Plotter class methods.

An instance of this object can be created in the following ways.

```
cv_plotter = Plotter(get_repo_root() + '/results/model_plots/', name='Tuning Model', savename='tuning_model')
```

2. *models.py*

models_helper.py is then imported into *models.py* where the actual object instantiation and modeling is assembled.

```
from models_helper import Dataset, Model, ModelTuner, Plotter
```

The entire modeling is structured under a wrapper function in order to easily port into *main.py* later on.

```
run_modeling_wrapper(df_train, df_test, df_val, ss_target, df_test_untouched,
random_seed = 33, run_base_estimators = False, run_model_tuning = False,
fast_gridsearch = True, save_model = False, demo = False)
```

3. *main.py*

models.py is then imported into our main script *main.py* where it is integrated with the preprocessing and GUI phases of the project.

```
import models as mdl
```

The modeling wrapper has a few boolean parameters that a user can set in order to skip some parts of modeling that may take a while to execute.

```
mdl.run_modeling_wrapper(df_train, df_test, df_val, ss_target, df_test_untouched,
run_base_estimators = False, #Run base models comparison or not
run_model_tuning = False, #Run hyperparameter tuning and gridsearchcv or not
fast_gridsearch = False, #Skip most of gridsearchcv to run faster
save_model = True, #Save best model results or not
demo = True) #Demo = True, will skip all of modeling since we already have results
```

The following table outlines the different options to set in the modeling execution wrapper or using command line Python execution of *main.py* and what they mean for runtime. The command line execution is shown in Mac/Linux commands, for Windows, use *py -3* instead of *python3*.

Option	Estimated Runtime*	Command Line Execution	Manual Settings in main.py	What it Does

Nap Mode	2+ hours	`python3 main.py nap`	`run_base_estimators = True`
			`run_model_tuning = True`
			`fast_gridsearch = False`
			`demo = False`
				Run all processes from scratch: Base models, 2 phases of GridSearchCV, model evaluation, model selection, predict test data, outputs results.

Lunch Break Mode	10-15 mins	`python3 main.py lunch`	`run_base_estimators = False`
			`run_model_tuning = True`
			`fast_gridsearch = True`
			`demo = False`
				Run with a smaller set of hyperparameters to tune with. It significantly compresses the time spent on hyperparameter tuning by only iterating over a smaller set of combinations.

Have a Coffee Mode	5 mins	`python3 main.py coffee`	`run_base_estimators = False`
			`run_model_tuning = False`
			`fast_gridsearch = False`
			`demo = False`
				Skip model hyperparameter tuning entirely. It skips the hyperparameter tuning entirely and instead loads the already found best model pickled in the results directory

Demo Mode	1 mins	`python3 main.py demo`
		or just
		`python3 main.py`
			`run_base_estimators = False`
			`run_model_tuning = False`
			`fast_gridsearch = False`
			`demo = True`
				Skips modeling entirely and is useful to focus on other portions of the project like the GUI.

Figure 20: Options for choosing which parts of modeling to execute. *actual runtime depends on machine.

5.3 Model Scoring

To judge an estimator's performance, we will use the Mean Squared Error (MSE) of the predicted rating against the actual rating on the validation dataset. MSE is a measure of the error, or difference, between the predicted and expected output and can be described as,

$$MSE = \frac{1}{n} \sum_{i=1}^{n} (y_{actual,i} - y_{predicted,i})^2 \qquad (6)$$

where n is the number of observations in the validation dataset and i is the ith observation in n. The model with the lowest MSE will have the highest score.

5.4 Base Models

Our initial lineup of models to try are the default SKlearn's implementations of:

Model Type	SKlearn's Object	Pros	Cons
Linear Regression	SGDRegressor()[6]	Simple to implement and understand	Doesn't work well for non-linearly separable dataset
Random Forest	RandomForestRegressor()[7]	Less prone to overfitting	Large number of trees can be slow to use. Can't extrapolate on data outside range of trained data.
Gradient Boosting	GradientBoostingRegressor()[8]	Fits each subsequent tree on the residuals which can learn very well	Generally slower to fit than Random Forest and more prone to overfitting
Adaptive Boosting	AdaBoostRegressor()[9]	Uses many decision stumps and each stump gets a weighted vote	Not as robust to outliers as it tries to fit to every datapoint
K-Nearest Neighbors	KNeighborsRegresor()[10]	Simple to understand and low number of hyperparameters	Not as efficient as dataset grows

Figure 21: Base SKlearn models to try.

We have detailed a few pros and cons for each model that we will have to consider in our model selection, understanding that these pros and cons are not exhaustive and can be relative depending on which models you compare to. Additionally, the next table outlines the details on each model's optimization method and loss function.

Model Type	SKlearn's Object	Optimization Method	Loss Function
Linear Regression	SGDRegressor()[6]	Stochastic Gradient Descent	Mean Squared Error
Random Forest	RandomForestRegressor()[7]	Build tree to split data	Mean Squared Error
Gradient Boosting	GradientBoostingRegressor()[8]	Build weak trees to split data and use gradient descent to fit residuals	Mean Squared Error
Adaptive Boosting	AdaBoostRegressor()[9]	Build weak trees to split data and samples with high errors are weighted higher for use in next tree	Mean Squared Error
K-Nearest Neighbors	KNeighborsRegresor()[10]	Calculate the minimum distance and take the mean for K number of neighbors	Mean Squared Error

Figure 22: Summary of optimization method and loss function for each model.

The results from the initial training round found that the Random Forest and Gradient Boosting models were the top performers with the KNN in third as shown in the figure below.

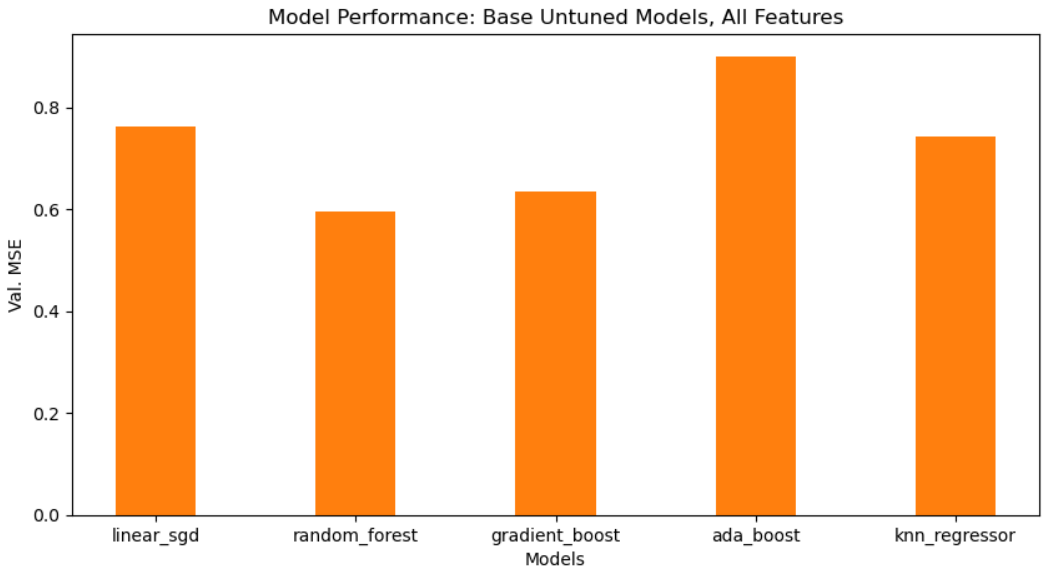

Figure 23: Random Forest, Gradient Boosting, and KNN are top 3 performers.

5.5 Hyperparameter Tuning I

Hyperparameters are model parameters that are set prior to training and are not, usually, learned through training while parameters in typically machine learning lingo represents model

parameters that are learned during training such as weights and biases. The following table shows the hyperparameters that we will tweak during our tuning phases.

Model Type	Hyperparameter	Hyperparameter Meaning	Dtype and Default
Random Forest	n_estimators	The number of trees in a forest.	int, default = 100
	min_sampes_split	Minimum samples required to split an internal node.	int or float, default = 2
	min_samples_leaf	Minimum samples to be at a leaf node.	int or float, default = 1
	max_features	Number of features to consider for each tree.	{'auto', 'sqrt', 'log2'} or int or float, default = 'auto
	max_depth	Max depth allowed for each tree	int, default = None
Gradient Boosting	learning_rate	Shrinks the contribution of each tree.	float, default = 0.1
	n_estimators	The number of boosting stages to use.	int, default = 100
	min_samples_split	Minimum samples required to split an internal node.	int or float, default = 2
K-Nearest Neighbors	n_neighbors	Number of neighbors to use.	int, default = 5
	p	p = 1 (Manhattan Distance), p = 2 (Euclidean Distance)	int, default = 2

Figure 24: Hyperparameter definitions. Source: SKlearn model documentations.

In our first hyperparameter tuning phase, we selected our top 3 performing estimators and adjusted a few hyperparameters to see if we can improve upon their base scores. For this phase, we used SKlearn's *GridSearchCV()*[11] to iterate through each combination of hyperparameters for each model. We will perform a K-Fold cross validation phase after training on each combination where $k = 1$ and the validation set used will be our predefined validation dataset.

Each model's hyperparameter grids for this phase and their best performing hyperparameters are shown in the table below.

Model Type	Hyperparameter Grids	Best Performing Hyperparameters
Random Forest	n_estimators: [100, 200, 400]	n_estimators = 400
	min_sampes_split: [2, 4, 8, 16]	min_sampes_split = 2
Gradient Boosting	learning_rate: [0.01, 0.1]	learning_rate = 0.1
	n_estimators: [100, 200, 400]	n_estimators = 400
	min_samples_split: [2, 4, 8, 16]	min_samples_split = 2
K-Nearest Neighbors	n_neighbors: [3, 5, 7, 9]	n_neighbors = 9
	p: [1, 2]	p = 1

Figure 25: Hyperparameter tuning I.

The following three figures show the validation curves for each of our models where we are plotting MSE on the y-axis and different hyperparameters that were tried on the x-axis. This is a useful visualization to see how a different hyperparameter value changes the model's MSE.

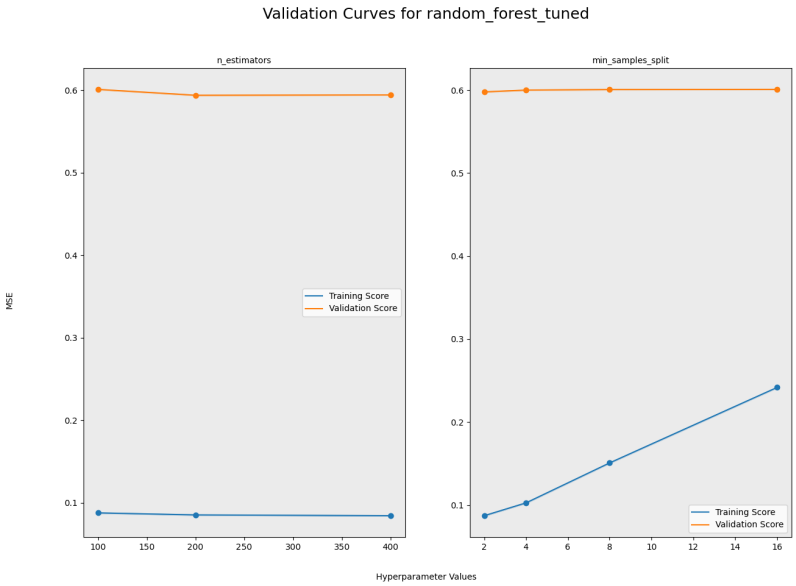

Figure 26: Validation curves[12] for Random Forest model.

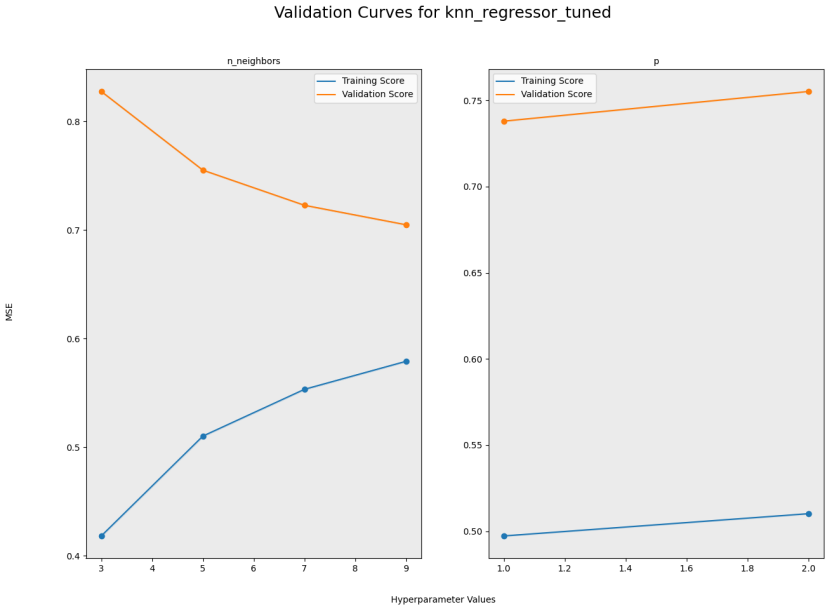

Figure 27: Validation curves[12] for KNN model.

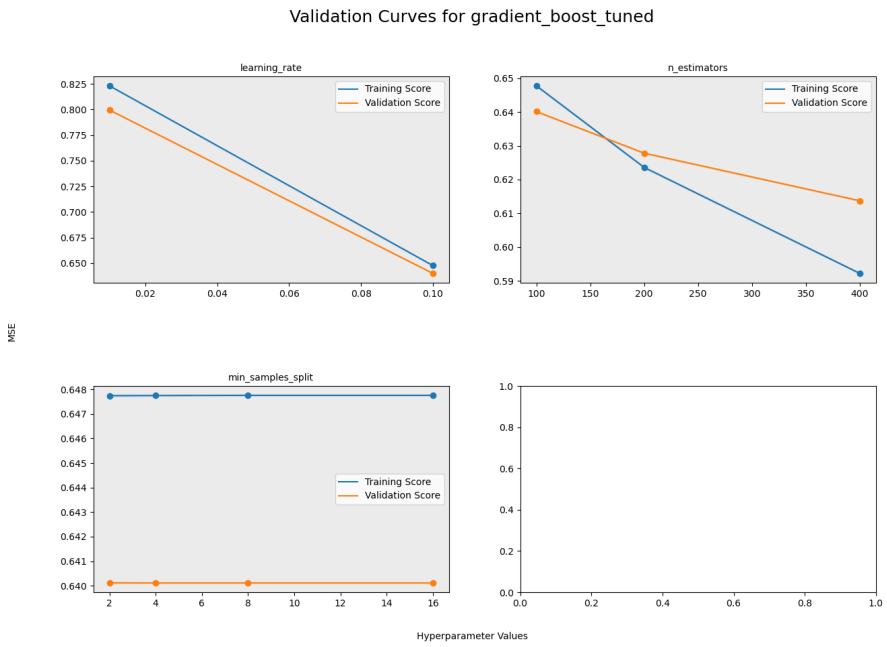

Figure 28: Validation curves[12] for Gradient Boosting model.

It is interesting to note that generally across all three models, the training score is lower, quite considerable so in some cases, than the validation scores. This can be a sign of overfitting where our learned models have high variance and are not generalizing as well. We will try to add some regularization parameters in our next hyperparameter tuning phase for our best model.

Lastly, the following figure shows the best performing model after our first phase of hyperparameter tuning. Our best model is still the Random Forest. In the next phase, we will attempt to further refine the tuning for our Random Forest model and will keep overfitting in mind.

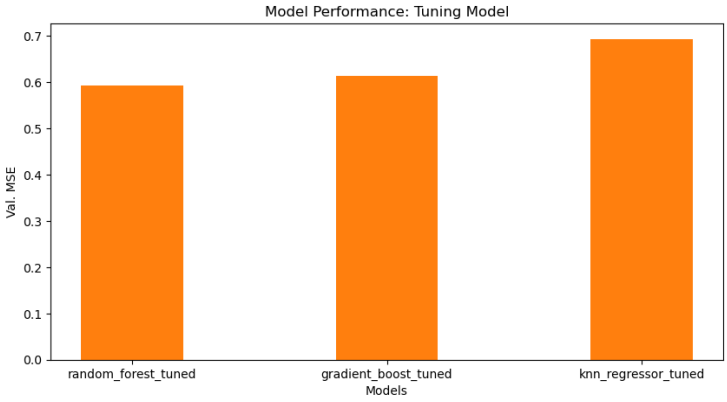

Figure 29: MSE comparison of best hyperparameters for each model after tuning phase I.

5.6 Hyperparameter Tuning II

In this phase, we will now focus solely on the Random Forest model in an attempt to fine tune it more and reduce overfitting. Our new set of hyperparameters to use are:

Model Type	Hyperparameter Grids	Best Performing Hyperparameters
Random Forest	n_estimators: [200, 300, 400, 500]	n_estimators = 500
	min_samples_leaf: [2, 4, 8, 12]	min_sampes_leaf = 2
	max_feature: [0.3, 0.5, 0.7]	max_feature = 0.7
	max_depth: [5, 10, 15, 25]	max_depth = 25

Figure 30: Hyperparameter tuning II.

We decided to try increasing the number of trees in our forest, increase the minimum samples allowed at a leaf, and reduce the maximum number of features used when building a tree in order to add regularization to our model and address overfitting. The next two figures show the validation curves and learning curves for our Random Forest model.

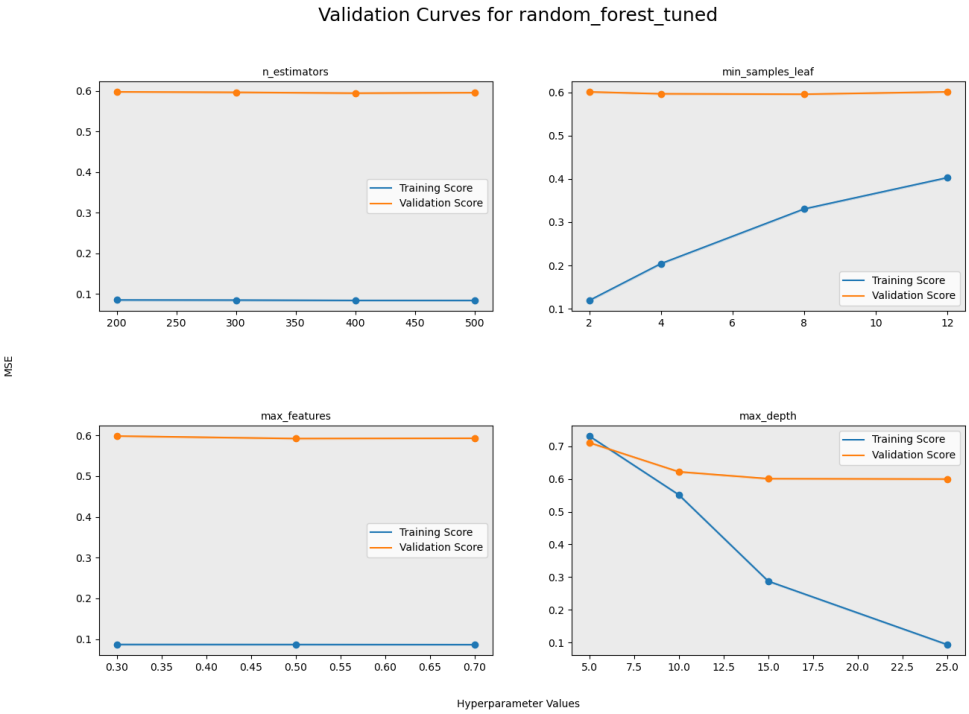

Figure 31: Validation curves[12] for Random Forest model.

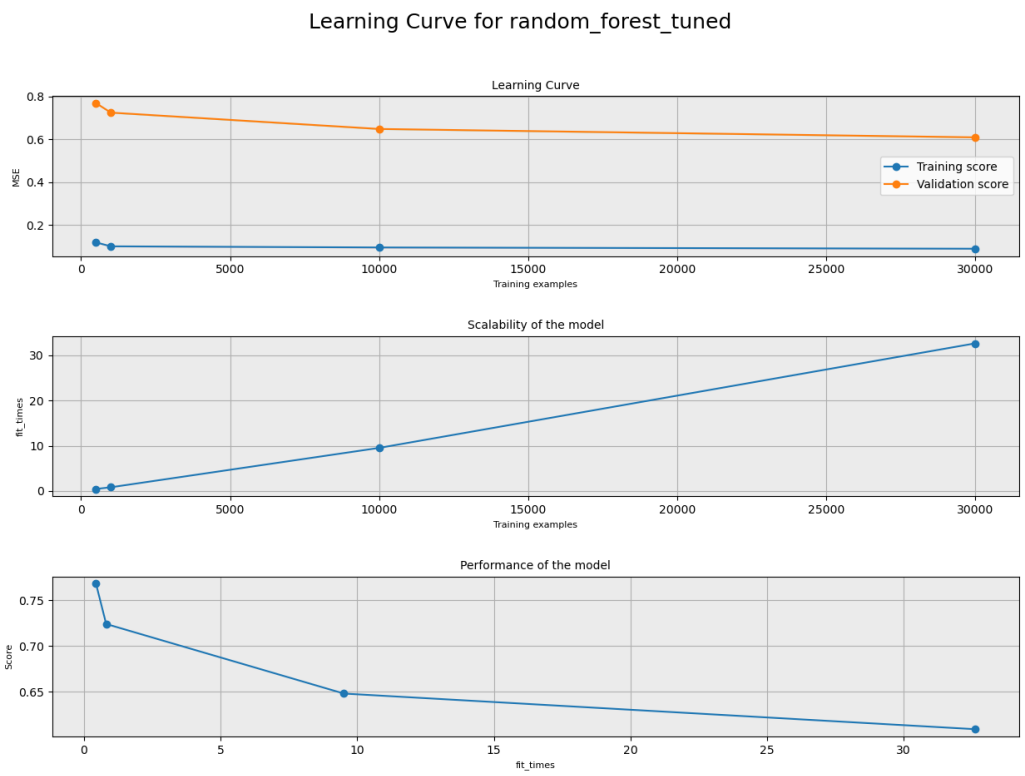

Figure 32: Learning curves[13] for Random Forest model.

It is interesting to note that only max depth at around 6 levels is where the training and validation scores are roughly equal while the other hyperparameters did not address overfitting that heavily. We also plotted the learning, scalability, and performance curves of our model. Note that our MSE seems to slowly plateau with additional training data. The time complexity of our model seems to be roughly in the order of $O(n)$ which is not bad. The performance curve shows that the model's MSE decreases with longer fit times which can generally mean a more complex model or fitting more training data.

5.7 Retrain and Test

Lastly, we retrained our model on the combined training and validation dataset and the relative feature importance of each feature can be visualized in the figure below.

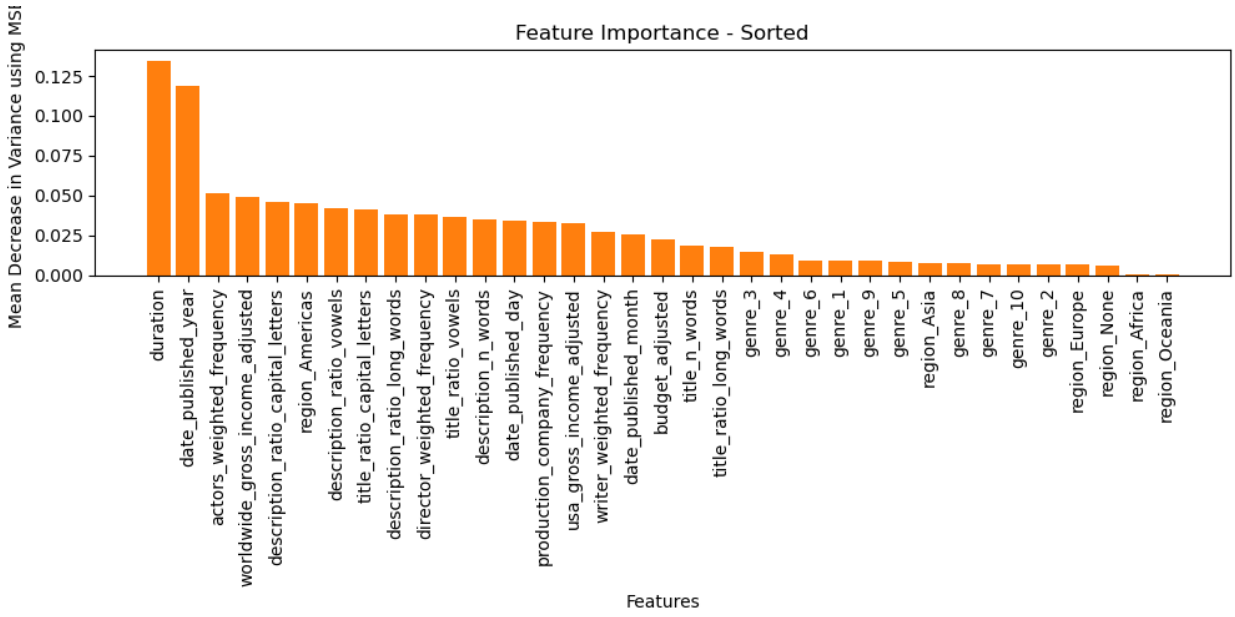

Figure 33: Feature importance of our Random Forest model.

Some interesting things to note is that *duration* and *date_published_year* have the highest importance. The third highest is *actors_weighted_frequency* which suggests movies' ratings are driven by the acting cast more than the director, writer, or production company. Additionally, a movie being made in the Americas is more important to predicting the ratings than any of the other regions. On the other hand, genre doesn't seem to have that big of an impact on the predictability of movies' ratings. This could also have been a result of the method that we used to encode genres; perhaps there is a better method.

Finally, we then proceeded to test on our testing dataset where the results are discussed in the next section.

6. Results

6.1 Testing and Model Evaluation

Our final MSE on the testing dataset is 0.602 on the scaled target label and is 0.869 once we reversed the scaling performed on the target label. The MSE by themselves are not the best to explain to humans how well our model is performing. We wanted to add context into this score that can explain truly how good our model is. We will do this in two ways:

1. Calculate the RMSE for the inverse scaled target label MSE which will bring us back into the same scale as the weighted average IMDb rating by reversing the squared operation in the MSE. We can think of the RMSE being used in a setting such as, $Prediction\ Rating\ \pm\ RMSE$, in a one-to-one comparison against the prediction rating. Although not exactly apples-to-apples as using pure average error, this can still give us some idea of how good our model is that is intuitive for humans to understand. Our RMSE for the inverse scaled target label is 0.931. **This means that on (root mean squared) average, our error between the actual and predicted is $\pm\ 0.931$. In the rating's range of 1-10, this is not a bad error although also not amazing.**

Test Scores
MSE = 0.602
MSE (Inverse Rescale) = 0.869
RMSE (Inverse Rescale) = **0.931**

Figure 34: *Our Model's test scores*.

2. We can judge the MSE of our model against a model that randomly generates values between 1 and 10, the range of the weighted average IMDb rating. We will call this model a *Random Model* and the Random Forest model *Our Model*. The *Random Model* can be generated like so:

```
test_Y_random = np.random.uniform(1, 10, size=test_Y.shape)
```

   The MSE of *Our Model* vs the *Random Model* is as follows:

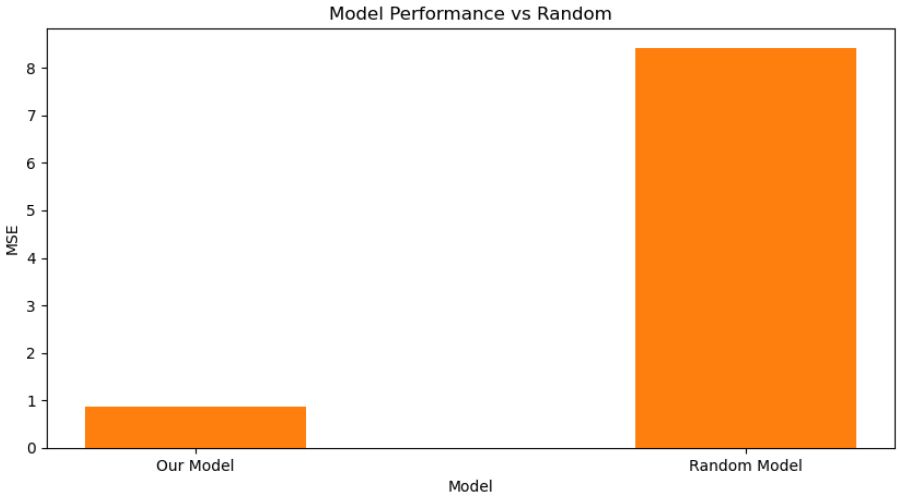

Figure 35: *Our Model* (Tuned Random Forest Model) vs *Random Model* (Random Generating Model).

The MSE of our tuned model is almost a magnitude lower than the MSE for the Random Generated Model. Although this seems significant at first glance, let's perform statistical testing to confirm that the two predicted rating distributions are truly statistically different.

We wanted to use a *2-Sample T-Test* to check if the means of the two predicted rating distributions are different. Before we can do that, we need to first confirm that our two distributions are approximately normal and have approximately equal variances in order to use the T-Test.

- We will first performed the *Shapiro-Wilk Test*[14] on each distribution to confirm normality where $H_0: Data\ drawn\ are\ from\ normal\ distribution$ with an $\alpha = 0.05$.
- Then we will performed a *Bartlett Test*[15] on both distributions to confirm equal variances where $H_0: All\ input\ samples\ have\ same\ variance$ with an $\alpha = 0.05$.

The table below shows our results from our 2 tests.

Test	p-value	Action
Shapiro-Wilk Test on Our Model Ratings	4.75e-40	Reject H_0
Shapiro-Wilk Test on Random Model Ratings	0	Reject H_0
Bartlett Test	0	Reject H_0

Figure 36: *Hypothesis testing results for normality and equal variance*.

Since all our p-values were below our α value, we rejected the null hypotheses for all three tests and our two distributions are not normal and do not have equal variances. Therefore, we cannot use a *2-Sample T-Test.* Instead, we will use:

- The *Mann Whitney U Test*[16] which is the non-parametric test that checks differences in medians or difference in locations between 2 distributions where $H_0: The\ distribution\ under\ sample\ x\ is\ the\ sample\ y$ with an $\alpha = 0.05$.

Test	p-value	Action
Mann-Whitney U Test	3.80e-29	Reject H_0

Figure 37: *Mann-Whitney U Test results for independence.*

With a low p-value we can reject our null hypothesis and confirm that the two medians of our predicted rating and the random generating rating distributions are different.

We were able to use the two steps outlined above to gain some understanding of how well our model performs in the context of the problem and against just randomly guessing. The table below shows a sample of 20 test ratings and the predicted ratings from *Our Model* and the *Random Model.* In this sample, *Our Model* has a smaller error 80% of the time (84% on the entire testing dataset).

Actual Rating	*Our Model* Predicted Rating	*Random Model* Predicted Rating	*Our Model* Error	*Random Model* Error	Smaller Error
4.7	5.3	5.4	0.6	0.1	RM
7.2	6.2	9.7	-1.0	3.5	OM
6.8	6.7	4.5	-0.1	-2.2	OM
6.4	6.2	1.7	-0.2	-4.5	OM
6.5	5.6	4.2	-0.9	-1.4	OM
6.3	6.0	2.5	-0.3	-3.5	OM
5.7	6.8	9.8	1.1	3.1	OM
7.2	6.0	8.9	-1.2	2.9	OM
4.4	6.3	5.4	1.9	-0.8	RM
6.8	6.4	4.6	-0.4	-1.8	OM
5.6	6.1	5.1	0.5	-1.0	OM
5.2	5.4	7.5	0.2	2.1	OM
8.2	6.9	3.2	-1.3	-3.7	OM
7.4	6.4	6.6	-1.0	0.2	RM
5.3	5.9	2.3	0.6	-3.6	OM
5.1	6.3	2.8	1.2	-3.5	OM

6.2	7.0	1.7	0.8	-5.3	OM
5.8	5.9	9.6	0.1	3.7	OM
7.8	6.0	1.5	-1.8	-4.5	OM
7.3	6.4	6.4	-0.9	0.0	RM

Figure 38: Sample of 20 ratings from test data showing the Actual, Predicted and a Randomly generated rating.

An interactive prediction game pitting your guess against *Our Model*'s prediction can be found in the GUI.

7. Conclusions

As can be seen from the results section, our model performs significantly better than the random model. This means that we were successfully able to create a model that can predict the weighted average vote of a movie with high accuracy. This also means that there is a strong relationship between movies' features and their IMDb score.

After finishing our project, there are a few ways we felt we could improve our research. Firstly, our research was partially limited due to missing or unusable data in the Names dataset. This prevented us from using valuable information about actors. Second, given more time, we could come up with a better way to encode the genres of each movie. While we encoded each set of three movies as its own separate genre, this does not make much sense practically. A movie that is an action, romance, comedy, is far closer to an adventure, romance, comedy, than it is to a science fiction, political, drama. Our encoding method did not reflect this, as we did not group genres based on their contents. We would like to find a better method of grouping genres, similar to how we grouped countries into regions.

Thirdly, we would be interested in reducing the dimensionality of our data further. While we used many different features in our work, we did not pinpoint which of these features had the largest impact on weighted average vote. In the future, we would spend more time on exploratory data analysis or analyzing our models to determine which features were the most important, while removing the rest. This would also allow us to draw conclusions about which features have the greatest effect on weighted average vote.

Lastly, while our modelling was successful, it left a lot to be desired in terms of efficiency. Running the full modeling module took upwards of two hours in most cases due to the extensive GridSearchCV and hyperparameter tuning. This is far too long for an average person to use when running the code, so instead we preloaded the best model from our work when people run our project. We would like to find a way to reduce the time it takes to run the full modeling module, so anyone is able to do it.

We also believe our project opens the door for further research in this field. As previously mentioned, IMDb also organizes ratings by males and females, as well as different age groups. In a future study, we could use what we have learned here and attempt to predict how weighted average vote differs based on sex and/or age. On the other hand, we could look from a more profit-oriented perspective and attempt to predict USA or worldwide gross income instead.

We were also interested in having a better understanding of how titles and descriptions play a role in affecting the weighted average vote of a movie. In a future project, we would like to use natural language processing on these two variables, and potentially other parts of the movie, like scripts, to understand how they affect weighted average vote.

Beyond the information we had in this dataset, there are many other variables we could investigate that affect movies. Screentime of actors, whether the movie is a sequel or not, where

and how the movie was released, as well as many other variables could all be used to predict weighted average vote.

We are also interested in using different modeling techniques in our work. In the future, we could use SKlearn's *VotingRegressor()* model, or another form of ensemble model. We could also move away from SKlearn, and instead focus on implementing modeling techniques from other packages. Lastly, we could further expand our options by employing more advanced modeling techniques using machine learning.

Our research has uncovered an interesting relationship between movies and their characteristics. Our model could guess if a movie would be critically successful based on its characteristics alone. Companies in the movie industry are constantly looking for which movies to invest in, and which ones to reject. This is traditionally done through gut instinct, where the heads of a company will read a script and listen to a writer or director's movie pitch. However, like many other industries looking to optimize, it may be beneficial for movie production companies to use predictive analytics when investing in movies.

While our model may not be perfectly applicable to movie production companies, as it does not focus on predicting profits, it is easy to imagine a more advanced model that could do this. A predictive model could easily look at a script, writer, director, and actors and make a prediction as to how much money a movie would make. It could even use natural language processing to predict if the script will be successful, or other techniques to calculate exactly what the budget for a movie should be. This would allow movie production companies to make investments without the errors of human judgement. In fact, these sorts of models may already be in use in the movie industry without the knowledge of the public.

Of course, this method is not without its drawbacks, and would likely make movie production companies prioritize making safer movies they knew would make money instead of taking risks with movies. This is a trend that is already present in Hollywood, with movie production companies prioritizing sequels to popular movies, or including incredibly popular actors with a significant fanbase in movies, no matter if they are a good fit for the role. These trends could become more problematic with the inclusion of predictive analytics models. While our model, and other models like it, could be beneficial for generating profits, it remains to be seen if they would have a negative effect on artistic creativity in the movie industry.

8. References

1. https://stephenfollows.com/how-many-films-are-released-each-year/
2. https://www.imdb.com/pressroom/stats/
3. https://help.imdb.com/article/imdb/track-movies-tv/ratings-faq/G67Y87TFYYP6TWAV#
4. https://www.kaggle.com/stefanoleone992/imdb-extensive-dataset
5. https://help.imdb.com/article/imdb/track-movies-tv/weighted-average-ratings/GWT2DSBYVT2F25SK?ref_=helpsect_pro_2_8#
6. https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.SGDRegressor.html
7. https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestRegressor.html
8. https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.GradientBoostingRegressor.html
9. https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.AdaBoostRegressor.html
10. https://scikit-learn.org/stable/modules/generated/sklearn.neighbors.KNeighborsRegressor.html
11. https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.GridSearchCV.html
12. https://scikit-learn.org/stable/auto_examples/model_selection/plot_validation_curve.html#sphx-glr-auto-examples-model-selection-plot-validation-curve-py
13. https://scikit-learn.org/stable/auto_examples/model_selection/plot_learning_curve.html#sphx-glr-auto-examples-model-selection-plot-learning-curve-py
14. https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.shapiro.html
15. https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.bartlett.html#scipy.stats.bartlett
16. https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.mannwhitneyu.html
17. https://www.visualcapitalist.com/global-gdp-by-region-distribution-map/
18. https://github.com/lukes/ISO-3166-Countries-with-Regional-Codes
19. https://fred.stlouisfed.org/series/CPIAUCNS

9. Appendix

9.1 Random Seed

Random seed used in the modeling phase was *33*.

9.2 Computer Listing

The Python packages that we used are in the *requirements.txt* file.

```
certifi==2021.10.8
charset-normalizer==2.0.8
cycler==0.11.0
fonttools==4.28.2
idna==3.3
imageio==2.13.1
joblib==1.1.0
kaggle==1.5.12
kiwisolver==1.3.2
matplotlib==3.1.2
networkx==2.6.3
numpy==1.21.3
packaging==21.3
pandas==1.2.4
Pillow==8.4.0
pyparsing==3.0.6
PyQt5==5.15.6
PyQt5-Qt5==5.15.2
PyQt5-sip==12.9.0
python-dateutil==2.8.2
python-slugify==5.0.2
pytz==2021.3
PyWavelets==1.2.0
QtPy==1.9.0
requests==2.26.0
scikit-image==0.18.1
scikit-learn==1.0.1
scipy==1.7.1
seaborn==0.11.2
setuptools-scm==6.3.2
six==1.16.0
sklearn==0.0
text-unidecode==1.3
```

```
threadpoolctl==3.0.0
tifffile==2021.11.2
tomli==1.2.2
tqdm==4.62.3
urllib3==1.26.7
```

9.3 IMDb License

IMDb, IMDb.COM, and the IMDb logo are trademarks of IMDb.com, Inc. or its affiliates.