Project 4 Analysis

**The Cleaning Process**

Hello everyone! we’re group number #5 and the members of this group are : Marisol Trejo, Alyssa Jones, Amanda Lor and me, Yesmelin Perdomo.

The topic we are predicting is: “Will this movie, prior to its release, be successful?” We defined success based on factors such as revenue, popularity, IMDb rating, Rotten Tomatoes rating, and Metacritic rating.

For this project, we aimed to gather as much information per movie as possible, ensuring the data was both accurate and recent. To achieve this, we retrieved data from various APIs.

While the IMDb API, available through Amazon AWS, offers a wealth of information, it is not free and is quite expensive. Therefore, we opted for alternatives like the TMDB API (The Movie Database API), which is a free non-commercial source, and the OMDB API (Open Movie Database), which we used in class and is also free. We used both APIs since they provided complementary data.

We began by calling the TMDB API, where we discovered that the data is paginated, with each page containing a set number of movies. We wrote code to calculate the total number of pages required to retrieve all desired movies. We then created a function to fetch the data in JSON format, storing all the movies in one place. Afterward, we split the data, as we needed the ratings from the OMDB API, which includes Rotten Tomatoes, Metacritic, and IMDb ratings. However, the OMDB API had limitations—it was only able to retrieve information for a specific title or IMDb ID, and the free version restricted us to 1,000 API calls per key.

As a result, we split the data into four parts, using the titles from the TMDB API to make calls to the OMDB API. We also used the TMDB API for financial information such as budget, box office, and revenue.

Next, we created a dictionary with the API keys from all four team members. Using these, we iterated through the list of movies, fetching financial details from the TMDB API and ratings data from the OMDB API. We combined this data using Python’s update() function to create a unified JSON object.

Finally, we added an extra key called “financial\_success,” which we defined ourselves. After some additional cleaning, such as extracting ratings from the dictionary, we filtered the data to ensure only movies with ratings were included. From there, the process was straightforward: we created DataFrames and wrote functions to minimize repetition. For example, we defined a function to select specific columns like "imdb\_id," "title," "release\_date," "runtime," "genre," "overview," "director," "actors," "imdb\_rating," and financial information such as "budget" and "revenue."

#splitting some columns

def splitting\_columns(dataframe):

dataframe[['star\_1','star\_2','star\_3']] = dataframe['Actors'].str.split(',', n=2, expand=True)

dataframe[['genre\_1','genre\_2','genre\_3']] = dataframe['Genre'].str.split(',', n=2, expand=True)

# Split the Director column

split\_directors = dataframe['Director'].str.split(',', n=1, expand=True)

# Create new columns with default values

dataframe['director\_1'] = split\_directors[0] # This will always exist

dataframe['director\_2'] = '' # Initialize with empty strings or NaN

# If there is a second column, assign it to director\_2

if split\_directors.shape[1] > 1:

dataframe['director\_2'] = split\_directors[1]

return(dataframe)

We continued by renaming columns to follow our snake\_case as opposed to the PascalCase format the json we retrieved had. We created conditions where a movie was considered successful if its revenue exceeded its budget, and the IMDb, Rotten Tomatoes, and Metacritic ratings were above 60%.

We then proceeded to convert certain columns to their correct data types, drop unnecessary columns, check the data types again, and finally apply our definition of success through specific conditions. If a movie's revenue exceeds its budget and its ratings on IMDb, Rotten Tomatoes, and Metacritic are above 60%, we assign a positive Boolean outcome. Otherwise, if it doesn’t meet these criteria, the result is negative. Here’s an example of the conditions we implemented:

# Create a mask for each condition

mask\_financial\_success = all\_movies\_df\_0['financial\_success'] == True

mask\_imdb\_rating = all\_movies\_df\_0['imdb\_rating'] > 6.0

mask\_rotten\_tomatoes\_rating = all\_movies\_df\_0['rotten\_tomatoes\_rating'] > 60

mask\_metacritic\_rating = all\_movies\_df\_0['metacritic\_rating'] > 60

all\_movies\_df\_0['outcome'] = mask\_financial\_success & mask\_imdb\_rating &

mask\_rotten\_tomatoes\_rating & mask\_metacritic\_rating

Once completed, our DataFrames were saved into four separate CSV’s.

**Data Processing**

The next step in our project was to process the data. Starting, we used pip to install Spark, which enabled us to run the application in Google Colab. If the user is using Google Colab, this is your first step, to ensure that the code will run. For those running the code in VS Code, you would need to install all necessary tools on your local machine and start by importing the required dependencies in your file, such as pathlib, pandas, and sklearn for your machine learning model. Next, we imported the Spark libraries to help us retrieve and process the data efficiently. Once Spark was set up, we retrieved the data by first initializing a Spark Session, setting the path for our previously created CSVs, and then reading them into Spark. We used specific attributes to properly handle the data, such as separating by commas (since they are CSVs), using inferSchema to determine data types, and—importantly—applying quote, escape, and multiline options because our 'overview' column contained these elements. This approach helped resolve a major issue we encountered.

Next, we joined all four CSVs to create a consolidated data frame. From there, we created a temporary view to explore the data further. For instance, we queried which actors appeared the most in movies, and checked the movies with the highest and lowest IMDb ratings. Additionally, we created a query to count the number of movies with specific IMDb ratings to find the ones that have the most in common.

Another key query was focused on financial success, a column in our dataset. This query filtered for movies where revenue exceeded the budget, sorting the results in descending order. As expected, movies like Avatar, Endgame, and Titanic topped the list, which aligned with online information.

After completing the data exploration, we exported the Spark DataFrame into a Pandas DataFrame to prepare for the machine learning model. We cleaned up the data by dropping unnecessary columns, correcting data types, and stripping non-numerical characters from the runtime column. Since we had several categorical columns, we applied a one-hot encoder. After merging the numerical and categorical data, we were ready to start feeding it into our machine learning model.  
   
**Logistic Regression Model**

We began by exploring a logistic regression model due to its reliability and efficiency to be able to rapidly train models with large datasets.

Our approach involved the following steps:

* **STEP 1)** **Data Preparation:**

We separated the features (X) from the target (Y) by splitting our preprocessed data into feature and target arrays.

* **STEP 2) Data Scaling:**

We trained and scaled our data using StandardScaler.

* **STEP 3) Model Creation:**

We created a LogisticRegression classifier, utilizing the newton-cg solver. We set the maximum number of iterations to 200 and chose a random state of 1.

* **STEP 4) Model Evaluation:**

After adjusting the number of iterations and changing the solver from SGD to NEWTON-CG, the best accuracy we achieved was 72%. Unfortunately, this did not meet the requirement of 75% so the next model we chose was using a Neural network.

**Deep Neural Net Model using TensorFlow**

We decided to explore a deep neural net model using TensorFlow, known for its ability to handle large amounts of data and often outperform traditional machine learning models in terms of accuracy.

Our approach involved the following steps:

* **STEP 1) Data Preparation**:

We separated the features from the target and trained and scaled our data, following the same process as before.

* **STEP 2)** **Model Parameters:**

We defined the parameters for our deep neural net, including:

- number\_input\_features: Determines the number of input features in the dataset.   
- hidden\_nodes\_layer\_1, hidden\_nodes\_layer\_2, and hidden\_nodes\_layer\_3: which define the number of hidden nodes in each of the three hidden layers (set to 32, 64, and 128, respectively).

* **STEP 3) Model Creation:**

We defined the model using the tf.keras.models.Sequential, allowing us to create a neural network layer by layer.   
- To start we have the hidden layers:   
 • The first hidden layer has 32 nodes and uses the ReLU activation function.   
 • The second hidden layer has 64 nodes and uses the ReLU activation function.   
 • The third hidden layer has 128 nodes and uses the Tanh activation function.   
 - Then we added the output layer with a single node and the Sigmoid activation function.

We chose three activation functions for our model: ReLU, Tanh, and Sigmoid. Here's why:

* ReLU is easy to implement, non-linear, and keeps many good properties of linear models.
* Tanh maps inputs to a range between -1 and 1, which is helpful when modeling outputs with similar ranges.
* Sigmoid maps inputs to a range from 0 to 1, making it perfect for binary classification problems (e.g., 0 or 1, yes or no).
* **STEP 4) Compile the model:**
* The loss function as binary cross-entropy which is suitable for binary classification problems.
* The optimizer as adam, which adapts the learning rate for each parameter based on the magnitude of the gradient, making it suitable for a wide range of problems.
* And accuracy as the metric.
* **STEP 4) Train the model**
* **STEP 5) Model Optimization:**

We achieved an accuracy rate of 75% after optimizing the model.

Initially, we selected a neural network architecture with 10 and 20 nodes, two hidden layers, one output layer, and trained it for 100 epochs, resulting in an accuracy of 68%. We experimented with adjusting the number of epochs, hidden layers, and activation functions, ultimately settling on 70 epochs, three hidden layers, and the Tanh activation function.

One main issue with this machine learning approach is the stabilization of the random state. Despite achieving a maximum accuracy of 75%, the accuracy changed with each model run, making it unreliable. Therefore, we decided not to pursue the project with this   
model, leading us to explore alternative approaches, such as Random Forest.

**Random Forest**

The final model that we decided to implement was the random forest model, due to its high accuracy and efficiency in being able to handle both regression and classification tasks. Random forest models use a large number of “trees” which help to reduce the variance and prediction error.

This model tends to generalize data better by splitting in between multiple trees, ensuring that no overfitting occurs. This ensures that “tight” data does not happen, which can overcrowd the trees, making the model perform poorly. Random forest models are able to manage missing data and values better than most models, leaving them less sensitive to outliers, ensuring proper data formation. Due to the high accuracy, we decided that the random forest model was the best model to showcase our data.

When starting the model, we had issues with deciding how many trees, or n\_estimators, should be used in our model. When starting low, we had an accuracy of around 71%, but this did not match our criteria, so we increased the n\_estimators from 50 to 100, and saw a decrease in our accuracy. After trial and error, we landed on an n\_estimators of 20 to showcase our model, however we were still unable to reach a 75% accuracy.

We then decided to add a random\_state, which controls the randomness involved in the data shuffling process, we first had it as 42, in which the data that resulted did not increase the accuracy, therefore we set it to one, having a slight increase to 72%. This code “random\_state=1” was then implemented throughout the rest of our code.

Although our accuracy was increasing, we were unable to reach the 75% accuracy mark, so we decided to add max\_depth and max\_features into our model. For max\_depth, we equaled it to “None”, as this ensured that the trees in the model will grow until all leaves of the model are “pure”. This means that each node in a tree gets filled, until it reaches a point, where overcrowding can be at risk. This also ensures that each node in the tree have the same minimum amount of data required as well. For max\_features, we set it equal to “sqrt”, which is the default for classification tasks for the random forest model. This guarantees that the performance and randomness of the model positively bounce off of each other, ensuring proper data splitting between the trees and nodes.

With this addition, we were able to increase the accuracy from 72% to 73%. Since we were still unable to reach 75% accuracy, we decided to add criterion. Criterion is used to measure the quality of a split in each node. We initially set criterion equal to “gini”, which is the default setting that measures the impurity of a node based on the distribution of the class labels. In short, it evaluates the best split at each node. We did not see a change in accuracy, therefore we set criterion equal to “entropy”. This performs the same actions as “gini”, but splits data homogenously, meaning on the most instances of a single class, which results in a higher information gain.

We then added the parameter class\_weight in order to assign weights to different classes of the dataset, which helped to balance our data. Our model depicted “class\_weight = {0: 1, 1: 5}”, which meant that class 0 is healthy and class 5 is unhealthy, or higher-risk data.

With this change, we were able to reach 74%, however the 75% threshold was not met. To accomplish this, we added more trees (n\_estimators) to our model, changing from 100 to 200, giving us an accuracy of 75.1%.

In the end, we were able to achieve 75% accuracy with our random forest model, with the following classifications:

n\_estimators= 200

criterion='entropy'

max\_depth =None

max\_features='sqrt'

class\_weight = {0: 1, 1: 5}

random\_state=1

**Challenges/Conclusion**

With all of the models used, we ran into multiple issues, including trying to achieve 75% accuracy and proper data implementation. One major issue that we dealt with regarded our API, as it was only able to pull 1000 rows of data, once per day, making us create multiple API keys and a for-loop to obtain new data, so that it did not overlap. Another issue we ran into was trying to achieve 75% accuracy, in which we went through multiple models and the addition and subtraction of certain columns and data. We were able to overcome this by adding in certain columns and using trial-and-error to achieve our accuracy.

In the future, we believe that fine-tuning the model, through trial-and-error is needed and changing of weight and columns can help to increase our accuracy and efficiency of the model. We also believe that simplifying the data into a single file, would be beneficial and it ensures formality and ease of use for the user.

In conclusion, we determined that the random forest model was the best model to use, as it provided the best accuracy with little room for error. Although we were able to achieve 75% accuracy with tensor flow, the randomness of the model, made it unreliable in the long run. Through trial and error, we were able to achieve the 75% accuracy threshold and accurately build a machine-learning model that answered the question of a future movie’s success based on previous information of a movie’s, genre, actor, revenue, and director.