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.

Logistic Regression

For our logistic regression we started off by initializing a spark session to retrieve our data to read four CSV files to use for our machine learning. Afterwards, we merged them into a DataFrame using the union operation. Once completed, the next step was to clean the data before modeling.

To do this, we converted the DataFrame to a pandas Dataframe. Removing unnecessary columns and changing the data types accordingly. Additionally, identifying the categorical columns that would help our machine learning get an accurate score. Then using the OneHotEncoder we converted the categorical columns into a numerical format for our machine learning model by separating the features from the target, to train and scale the data by defining the X and Y with our data. Next, we split the data and scaled it to fit the model. Afterwards, we were ready to proceed creating the logistic regression model.

We initiated the model with the solver set to the default, with a parameter of 200. Furthermore, we put in the X and Y values we set from our data to get our final machine learning score. However, our accuracy score was not reaching the 75% threshold required. Although a few modifications were made, there was no drastic changes. Therefore, we decided to change models to see how it would compare with this logistic regression 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. As a natural fit for binary classification problems, logistic regression aims to predict one of two outcomes.

Our approach involved the following steps:

1. Data Preparation: We separated the features (X) from the target (Y) by splitting our preprocessed data into feature and target arrays.   
 2. Data Scaling: We trained and scaled our data using StandardScaler.   
 3. Model Creation: We created a LogisticRegression classifier, utilizing the newton-cg solver, which combines the Newton-Raphson method with the conjugate gradient method. We set the maximum number of iterations to 200 and chose a random state of 1.   
 4. Model Evaluation: After adjusting the number of iterations and changing the solver, the best accuracy we achieved was 72%.

Unfortunately, this did not meet the requirement of 75%, so the next model we choose was using a Neural network.

Deep Neural Net Model using TensorFlow

Next, 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:

1. Data Preparation: We separated the features from the target and trained and scaled our data, following the same process as before.   
 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: define the number of hidden nodes in each of   
the three hidden layers (set to 32, 64, and 128, respectively).

3. Model Creation: We defined the model using the tf.keras.models.Sequential API, allowing us to create a neural network layer by layer.   
 4. Hidden Layers: We added 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.   
 5. Output Layer: We added the output layer with a single node and the sigmoid activation function.   
 6. Activation Functions: We chose these activation functions due to their properties:   
 • ReLU is simple to implement, non-linear, and preserves many properties of linear models.   
 • Tanh maps the input to a range of (-1, 1), helping with modeling outputs with   
similar ranges.   
 • Sigmoid maps the input to a range of (0, 1), suitable for binary classification   
problems.   
 7. 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 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 ensures 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 has 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 ensures 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 helps 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 to our model, 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/Conculsion

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 only being able to pull 1000 rows of data, once per day, making us create multiple API keys and a for-loop to obtain new data that 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 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.