Capstone 1: Data Wrangling

The data acquired for the first capstone project was obtained from The Movies Database (TMDb) website using a personal API key that they provided to me. The data was collected over the course of eight days by the use of a request and storage program that I wrote. It sequentially requested movie and actor data based on the unique TMDb ID for each item, starting from ID number 1. The movie and actor data were collected separately through iterations of the requests script and stored in their own JSON files to be munged by additional programs. The sequence of requests terminated when the iterator reached the ID of the item that was added to the TMDb database most recently. These final ID's can be found with a specific type of request from the website.

When I first started collecting data for this capstone, I pulled many datasets from different sources, such as Internet Movie Database (IMDb), Kaggle, Movie Lens (Group Lens website), and Opus Data (The Numbers movie website). For this capstone to succeed, I would need financial data for a lot of movies. This data is difficult to find, as most production companies are very secretive about their financial numbers. Metadata for acting roles would be useful for having features that distinguish different people, as well. I searched through these sources with these items in mind. The datasets from IMDb and Movie Lens didn't have useful data for this project. The Opus Data datasets didn't have the proper keys that could be used to join their data with other data sources.

While the IMDb website has a lot of financial information about movies, most of it is only viewable to subscribers of their premium service. I wasn't sure if being a subscriber would allow my API key to request this data for download. So, I put this option on the back burner as a last resort. There is a python module (imdbapi.py) that can be used to access the data through an API. I spent some time learning how to use this module in case I needed more financial data for my final dataset.

I found three datasets on Kaggle that I felt I should inspect to see if any of them would be useful for financial data. The movie_metadata.csv dataset contained over 3,800 values for both budget and gross revenue. It also contained the same number of Facebook like values for the top three billing roles for each movie. The tmdb_5000_movie.csv dataset had over 4,800 values for both budget and revenue. There was no useful metadata for actors.

The the_movies.csv dataset looked really good at first. It has over 7,000 values for budget and revenue. I thought that I could use this one as my starting dataset, but I soon ran into troubles. The data was collected by a Kaggle user through an API, but before the values were stored as a JSON file, they were converted into strings. Through this, the data was in a form that pandas JSON methods could not accept as arguments. Ultimately, I was able to get around this, but then I began to question the accuracy of the data. The first red flag was that the dataset contained rows whose IMDb IDs were not unique. There were 27 pairs of observations that had the same unique identifier, but some of those pairs did not contain duplicate values throughout

their remaining features., as would be the case if just the IMDb ID was mislabeled. Either one or both of the rows of these pairs were mislabeled data. I made a few API requests to compare the website data with these rows in hopes of simply dropping the ones that held incorrect data. It turned out that some of the duplicate pairs were such that neither observation in the pair had the correct data. At this point, I questioned the accuracy of the entire dataset. I could either test the data by sampling it and comparing those samples to requests that I would have to make from a website or simply build my own dataset. As the data was available from the TMDb website with an unlimited request rate, I chose the latter.

Creating the request and storage program was simpler than I had anticipated. I needed to account for failed requests, because the connection would terminate after three tries. I scripted a counter with a sleep failsafe that waited three seconds until sending the next request in the event of a third failed attempt. After collecting my data, I had many large JSON files, each with 100,000 rows. I created a program that merged and stored this data into two very large JSON files, one with 2,700,000 rows for the person requests and one with 800,000 rows for the movie requests. They contained many empty rows that were the result of failed requests, mostly due to IDs being removed from the TMDb database at some point. By building the datasets through iterating over a range of integers I hoped that their TMDb IDs would be unique, which would simplify any incorporation of data from other sources which used that key. When I checked, this was not the case for the actor dataset. I was not sure of the reason for this, but I knew that there will always be the option to go back and do some sample requests manually to see if this error was due to the website or my program. Regardless, these observations were relatively small in number. So, I removed the duplicate rows for now.

After concatenating the data, I wrote a program to get it into a form such that it was generalizable to a variety of analysis methods, including some beyond the scope of this project. For instance, there was a lot of text data in the form of biography, reviews, taglines, and overviews. These could be analyzed with NLP algorithms to look for correlations that may reveal movie themes or public opinion about an actor that point to which people would have the most success as profit generators. There was data on full cast members in the movies, as well, which could be included in a graph for network analysis.

With both the person and movie datasets, I performed the standard data wrangling methods of dropping both the empty rows and unnecessary features, checking for missing data in the form of zero values and empty lists and dictionaries, converting dates to datetime objects, and ensuring all values were of the data type that would be most appropriate for the project. For both datasets, I dropped all of the rows that corresponded to adult films, as well.

For the person dataset, after checking the value counts of the movie department feature, I noticed that there were counts for actors and acting. So, I combined those columns. The ID feature values were mostly floats, but had several strings. Upon observing these rows, I saw that the values for each of their columns were labeled as “Acting”. I removed those rows. Also, I dropped the observations for those that represented non-acting jobs, as I did not need the crew

data for this project. The gender column had four different values. After searching the TMDb chat room, I discovered that gender zero was for missing data. I never found out what the fourth gender represented, but as there were just a few of them, they were removed along with the gender zero observations. This left gender one for female and gender 2 for male, which I converted to value zero to express the variable with binary values. The actor birthday variable had some strange values that went back into the 1800s. Some of these came from documentary movies, where people from archival footage are listed in the credits. I left these values in the dataset, as they belonged to legitimate roles that could be a factor in bringing revenue to a documentary movie.

For the movie dataset, I dropped all rows that did not have English as the original language. Also, I only kept movies that had values labeled released for the movie status column, as other types of projects should not have revenue numbers. The genre values were contained in dictionaries within lists. Most of the lists consisted of multiple dictionaries. Each dictionary represented one of nineteen different TMDb base genre types, keyed by a number with a value that was a string description of the genre. I wrote a script to sort through the column and return a Series, keeping only the genre name strings in unnested lists. Then, I replaced the old column with the one I had created. I verified that the IMDb IDs were all unique, as well. This gave me two ways to check if two movies were the same, in case I needed to fill in missing values with data from another source. I dropped all rows for movies with runtimes under 75 minutes, as that is the cutoff length for feature length films.The credits column consisted of dictionaries, each containing two lists, one for cast and one for crew. I extracted the two lists with the Pandas JSON normalize function and put them into separate columns. Finally, the reviews column contained dictionaries, as well. I only wanted to keep the text of the reviews, which I extracted the same way and put them into their own column, as well. Once again, the two datasets were stored as JSON files to be further wrangled on a per use case.

As mentioned, the data was given a generalized form, as missing values had not been removed or imputed and most features were retained. Now, I had the ability to perform further munging to get the data into the best form to be used in this particular project. As far as dealing with missing values, I felt that this dataset was so rich that I could proceed by simply dropping their rows. If I found it necessary to have more observations, I could easily rebuild the datasets with simple modifications to my programs, then extract more values from other sources, while keying them in using the unique IDs found in each observation.

Because many of the dates in the datasets were pre-epoch, I needed to store them in ISO 8601 format, which gave them timezone stamps. After reading in the data, I converted these dates back to Pandas datetime objects, while removing the timezone information. This made the time data usable, again. I dropped the IMDb ID column, as I wasn't planning on adding data at this point. I removed any text data, as well as the movie crew column. These were gathered for projects that could involve more advanced methods of analysis.

I decided to represent the monetary values in today's dollar amounts. I obtained the Consumer Price Index (CPI) from the Federal Reserve Economic Data (FRED) website. From this, I created a CPI multiplier column and converted the values to current dollar amounts. This put the data on equal footing for comparison. I decided to keep the high end monetary outliers, because this project aims to analyze the return on investment numbers, and I wanted to observe the maximum range of what has been achieved for that figure. As far as the low end monetary outliers were concerned, I felt that there should be a minimum budget value to be considered for this project. After looking at a lot of the movies in that range, I decided it would be best to set the low end cutoff at $40,000. That didn't seem very high, but I would have lost some good data on well known actors had I lifted the line higher. All of the other outliers are real values that reflect the broad range of movies being made. They were retained for now.

As mentioned earlier, I wanted to have return on investment values for each movie to use as aggregated metrics for each actor associated with the movie. I chose a simple profit over budget ratio to use as these values. While not reflecting the true nature of movie economics, these ratios can be used to weigh actors against each other, without implying any literal monetary significance.

Similarly, I combined the vote count and vote average for each movie, but instead of using a simple ratio, I used a true Bayesian average. This added a weighting factor which adjusted the average score of movies with low vote counts to be more reflective of the average vote across all movies.

The genre lists that I created for each observation potentially contained multiple base genres. I extracted those base genres and featurized them into 19 columns. This meant that if a movie was described by multiple base genres, it was now represented as having a count of one for each of those genres.

The final feature of the movie dataset was the cast information. Each observation was a list of dicts. Each dict held the information that pertained to that movie for one actor. I extracted the TMDb ID for each person to use as the unique key when it came time to combine both datasets. The other item I kept was the actor's billing in the movie. This would be needed to weight each actor as to their importance in the movie. It was assumed lead actors would be more responsible for the success or failure of a movie than someone in a minor role. Through the use of Pandas explode and JSON normalize methods, I was able to featurize these two sets of values from the cast lists.

As mentioned, I wanted to apply a weighting factor to the actors which amplified those at the top of the billing. The transform function I chose, killed off an actor's influence around the fifth or sixth spot in the billing order. These weighted billing orders were then absorbed into the ROI data for each movie to give weighted ROI values. The billing order feature was then dropped.

After merging the two datasets, I created an age feature derived from the release date and the birthday features, whose values are the age of the actor at the time of the movie release. Then, the parent features were removed along with the movie ID and actor name features, as those values could be found in the parent datasets if needed.

The dataset was then rearranged through a Pandas groupby call by the actors, with the genre values being summed over each actor. The remaining values were averaged over each actor, then binned to reduce discontinuity in the response, due to any punctuated effects in the data. The observations, at over 26,000 in number, far outnumbered these 4 non-genre predictors. This allowed these predictors to be binned such that the range of response within each predictor band would be small, allowing the average response to be more precisely determined. The data type of these variables were then converted to integer to increase performance. Finally, I noticed that the range of the weighted actor ROI was vast, due to the exponential in my weighting function. I applied the natural log function to these values to confine their range. The data is now ready for EDA and modeling.