MSCI 446 Project

Rosa Seohwa Kang (20717849)
Derek Xu (20661857)

Table of Contents

Abstract

It is important for businesses to apply data driven techniques in order to cater their content in ways that the public can enjoy. Many platforms encounter the issue of knowing what content to provide to users. Movie and TV show subscription platforms such as Netflix and Disney Plus are continuously adding and removing content from their platform each year. These subscription services must have processes set in place that determine content that is suitable and appropriate for users, posing as a challenge for platforms. This process is costly because acquiring distribution rights requires time, money, and labour to pick and choose which TV show or movie content users would most likely be willing to watch and enjoy on the platforms. The current media content that exist in a catalog (i.e. Netflix's movies and shows) must continuously be optimized based on several factors such as viewers' watch histories, similar users' watch histories, ratings, and more. By using a logistic regression model, a type of supervised learning, streaming services such as Netflix gain the capability of evaluating the performance and popularity of TV shows that may never have aired, a process that is important when evaluating if a new Netflix original should be produced for example. With unsupervised learning, we were able to find trends in the dataset by finding which frequent itemsets would cater to the general public in terms of what they would want to see in a movie catalog. In addition, we were able to find clusters of similar movies using a content-based approach by taking document term vectors of each movie and generating cosine similarity scores. By visualizing the clusters, we were able to detect common topics between clusters, as well as indicators of where businesses should put forth effort in adding movies based on what users enjoyed or did not enjoyed previously.

Introduction

Business Problem

The business problem poses as generating a personalized model for movie and TV show subscription services to find shows and movies to add, maintain, or remove from their catalog. This ensures that the platform's money is spent on maintaining distribution rights and licenses on movies that will drive viewer satisfaction, subscriptions, and not on content that viewers do not opt to watch. Mining data algorithms to improve this process of adding and removing content to a catalog allows the business to become more effective to users, as well as data driven and automated. Generically, this will reduce operational costs of gaining licensing for movies that are not currently valuable for the business platform's revenue and improves user engagement with a subscription platform.

The problem is relevant to movie subscription platforms and businesses that need to serve millions of users with improved selections of movie and tv show content, such as Netflix, Disney Plus, Youtube premium, Amazon Prime Video and many other stakeholders involved. In addition, the results of this problem would ensure that the users subscribed to these channels are provided with the TV show and movie content they would want, reducing program churn and dissatisfaction with the services. In essence, the solution to the business problem curates a carefully mined set of movies and shows that will work to satisfy both customer and business experiences from two different points of view. The tools and learning provided from both

Supervised and Unsupervised Learning components of this analysis will provide businesses with valuable insight for topics, frequently watched content that can pertain to a large group of users, and as well as most frequent movies and shows watched across the globe.

Supervised Learning

Streaming platforms such as Netflix and Disney plus have been consistently investing resources into producing as well as acquiring distribution rights for TV shows, as the concept of "binge watching" TV shows has risen in popularity since the rise of streaming platforms. Thus, there now exists a business need to ensure that the rotating catalog of TV shows available for consumption of streaming platforms will engage users. Additionally, prior to deciding to pick up new TV shows that have never aired anywhere else before (ex. Netflix Originals), knowledge of how popular a TV show will be with users is highly valuable. Therefore, the supervised learning task will be focused on predicting the average rating of a TV show using multiple feature sets and learning algorithms. These are detailed in table 1 of the appendix in Tables.

The target variable is a categorical interpretation of vote_average, rating_category. By segmenting vote_average into 3 distinct buckets, we convert the vote_average float into a categorical variable, and gain access to classification algorithms. This process and other data transformation processes will be explained in further detail in the "Data" section of the report. The main supervised learning algorithms used to predict rating_category are Logistic Regression, and the Decision Tree classifier. The selection of these algorithms over others, such as Linear Regression, will be discussed further in the "Results" section of the report.

Unsupervised Learning

From a business perspective it is valuable to search for patterns that can occur within a dataset of movies that also contains historical data of user-movie ratings. By taking a user-movie matrix that consists of the rating value that users have previously given each movie (from 1-5), we can sort the matrix to show the highest rated movies and highest rating or active users (in descending order) to obtain the most dense parts of the matrix. We can take a list of movies that each user has enjoyed watching i.e., rated a 5, and create a transaction for each user that contains movie items that they have rated highly. Doing so will allow us to mimic a transactional dataset of movies that each user has watched and enjoyed (due to sorting by highest rating) and generating a set of movie item transactions for each user. Association rule mining is applied to the itemsets of movies that are most relevant to users and derives frequent itemsets with a certain level of support and confidence relevant to the business. For example, an output of this Apriori algorithm could be:

$$\{The\ Dark\ Knight,\ \ Pulp\ Fiction\} \ -> \ \{The\ Godfather\}$$
$$Support: \ 0.5\ Confidence: \ 1$$

The explanatory variable is chosen as movie titles because it would help the business make decisions by looking at the relationships of movies that are involved in frequent itemsets. This can help derive patterns of which set of movies should be shown together or recommended for the business to add in their catalog since users likely enjoyed and highly rated all items in a transaction pertaining to an association rule created derived with a high support and confidence level. Association rules that generate predictions of occurrences of various movies based on

occurrences of other movies can also be interpreted in a conditional probability context. In the above example, the business can create inferences that if a user enjoyed The Dark Knight and Pulp Fiction, they are more likely to enjoy The Godfather. Therefore businesses can add these movies to their catalogs if it does not exist and they should either recommend this with a "You may also like... " prompt or simply have it show up in a "For you page".

On the other hand, by taking keywords or 'tag' words as explanatory variables that have been used to describe each movie, we can take text analytics as document term vectors and generate similarity calculations between the document term vectors extracted for each movie. Doing so creates the groundwork for taking a subset of related-content movies similar to a single movie in focus which can help decipher which movies to add to the catalog if the single movie has historically been highly rated, or which movies to remove from the catalog if the single movie compared has historically been low rated. In addition, we can cluster the similarity scores between each movie in order to find groups of similar movies described through tag data. This can help the business determine how to structure the movies they have in their catalog as well as analyze patterns across these groups in order to determine where they should allocate more funds for distribution rights, depending on the size and content of this data. It makes sense to cluster upon similarity scores between document term vectors of each movie in a content-based filtering approach, since it allows the model to show common content patterns within each cluster group as well. If the business would like to take the approach of spreading out their catalogs to involve an even spread of content, they could look for clusters that do not have as many movies compared to other clusters, and have a heavier hand in picking movies that adhere to that content within the cluster.

Related Work

Common applications using the TMDB API [9] and/or movielens dataset include focusing on movie filtering approaches such as collaborative filtering and content filtering in order to recommend users movies or shows to watch based on their tastes and previous choices. Some of them take on approaches of having the user input a list of movies with their personalized ratings, and have the model output a list of movies that they would like -- either by features that include movie genre, keywords, director, cast, ratings, and more. Though there are many methods of analyzing patterns with this data, our business problem is unique in the fact that it is trying to provide businesses rather than individual users recommendations of movies. As such, previous work with the datasets focus directly on recommending movies to individual users, but the business problem focuses on recommending large lists of movies and shows to include or take out in the catalogs of movie streaming platforms, thus solving a problem that addresses the corporations themselves, and inadvertently assists the experiences of users involved in the subscriptions of these platforms.

Supervised learning

A few prior works were discovered while researching existing work on predicting the success of TV shows. These included research papers from students at Stanford University [1], as well as sentiment analysis performed on social media to predict the popularity of TV shows [2]. While researching, many more works relating to predicting the success of movies were found as compared to work on TV shows [4][5][6]. A possible reason for this is that

comprehensive IMDb movie datasets are offered publicly. Although IMDb also offers some data on TV shows, the data present is not nearly as comprehensive as the movie data available.

The analysis performed here through supervised learning differs from the previous work in that the TMDB API is used as the primary data source, and offers different features and different vote_average data as compared to IMDB data. Additionally, analysis conducted for this project was focused around predicting future performance of TV shows that may not have ever aired, as well as ensuring that feature sets were as independent from the target variable as possible, and that feature sets did not include any proprietary data. To achieve this, vote_counts for all TV shows were dropped as an explanatory variable, as opposed to the previous work by stanford students that used vote_count as an explanatory variable.

This makes our resulting models and research more directly applicable to streaming platforms such as Netflix, who have their own proprietary viewership data. Training our supervised learning models on vote_count data which is proprietary to TMDB negatively influences the predictive power of the models when used to predict the success of TV shows which may never have aired, and whose viewership may drastically differ on different streaming platforms as compared to TMDB. Thus, vote_count was dropped as an explanatory variable.

Additionally, research into existing strategies used for determining the success of movies was done and transferred to this analysis on TV show success as well (citation here). This includes the use of TF-IDF vectors as possible pieces of the feature sets that were analyzed. This was to determine the correlation between major plot points and the success of a TV show, a strategy used in the literature [5][6] when trying to predict the success of a movie prior to its release, a similar business problem to ours.

Findings were consistent with previous works, however through the course of our analysis, we found that the usage of TF-IDF vectors in expressing major plot points as explanatory variables is quite promising and opens the door for further investigation. Surprisingly, the vote_count feature, despite being proprietary and intuitively related to our target variable vote_average and thus rating_category, did not have a large correlation with the target variable and thus predictive accuracy was maintained even after dropping this column.

Unsupervised learning

Very common use cases for the movielens dataset are approaches for creating movie recommender systems through demographic filtering, content based filtering, and collaborative filtering. The article [13] applies the movielens dataset to apply collaborative filtering using single value decomposition, where they apply the root mean square error and other metrics in order to improve the performance of predicting how a user would rate another movie based on their history of ratings for other movies. Our analysis is different because we used the rating data for collecting transactions of movies based on the level or ratings that users rate movies in order to execute association rule mining to the problem.

In another similar study to [14] the author applies collaborative filtering to sift through previous history of rating movies and how much they liked it through the value of the rating. It also trains the dataset using Python's surprise library in order to numerically predict the categorical variable which is the rating that a user would give a movie, which is different to our analysis that used this dataset to find associations and relations, rather than predictive analysis on categorical variables.

Another common use case for this dataset is to extract the tag data and genres for each movie in order to execute content data analysis. This analysis uses keyword occurrences and applies data visualization to view most highest rated genres [15]. What is similar with our analysis is that we also use tag data, but it is used for text mining with text vectorizers to generate document term vectors and relationships between movies instead.

A study that applies solely clustering to the movielens data set in order to infer similar groups of users through rating data, is the Unsupervised Classification Project [16]. The application first takes movies under specific genres, such as sci fi and romance, and takes the clustered rating values for the corresponding movies to visualize the cluster of movies through heatmaps. This study also takes the elbow method and provides a general explanation for choosing the right K-value in K Means clustering which was found useful when applying the elbow method to clustering similarity scores of vectorized text in the unsupervised problem. This analysis continues to take a user-movie matrix with the rating as the value in each cell and sorts it to show the highest rating users by highest rated movies in descending order. Sorting is a similar approach to the matrix we had when applying data transformation in the association rule mining. This is an approach taken in order to take the initial subset of the matrix that will be the most dense and takes previous behaviour of the highest active users (that rate the most) into account for the association model. This is also due to the fact that not all users in the dataset have rated every movie. However, the difference is that this application casts the matrix into a sparse dataframe to feed into KMeans and output 20 clusters of movies that have been clustered through its rating value. Our clustering analysis does not take rating values but rather the cosine similarity scores through text mining since it was more meaningful for us to decipher which movies were similar in terms or content, rather than rating behaviour. Lastly, this study takes a classification approach to predict what a user would watch a movie by identifying that each cluster column contains similar groups of users and movie columns will be clustered together. This is interesting because the logic follows that 10 users that rate 1 movie a 3, will be clustered together with movies and will take the average ratings of all similar users to predict the rating for a particular movie, which did not seem very clear or intuitive at first.

An interesting use case of this data application is taking the most popular genres year by year and visualizing which genres had growth throughout the years in terms of the level of movies. Tag data was also used to extract which words best summarizes a movie's plot. The study continues to apply this data to answer interesting questions, such as which movies were the most popular of every decade by using a weighted rating formula and analyzes which years were the best for a genre using the ratings table, which can provide businesses valuable insight [17]. This application does not take the common predictive analysis approach taught throughout the course, nor supervised/unsupervised learning techniques, however it works to answer interesting questions that are common for movie trends.

Data

TV show Data

The TV show dataset was manually created from data present on the TMDB API [8]. This was done by creating a script responsible for retrieving the first 1000 pages of TV Shows from the /tv endpoint, then retrieving all of these individual shows tv_show details. The two dataframes were concatenated together to form a dataframe containing all of the critical TV show data. After this was completed, significant data exploration was conducted to determine suitable supervised learning algorithm algorithms. This included plotting scatter plots to determine correlation among numeric variables, as well as histograms and statistics calculations to gain insight into usable columns.

Investigation into numeric fields was completed first, and can be seen in Figure 1 and Figure 2 in Figures.

Based on the data exploration performed on numeric variables, it was determined that there was little correlation amongst any of the possible target variables and the rest of the explanatory numeric variables. Linear regression was ruled out as a possible supervised learning algorithm, as it requires numeric variables. A similar conclusion was reached in the literature (stanford students), as they performed many trials using a multitude of linear regression algorithms, all of which underperformed compared to classification algorithms.

After it was determined that classification would be more ideal than numeric prediction, due to the low correlation amongst numeric variables alone, significant data transformation and cleaning was done to create a usable dataset upon which models could be trained. This was due to the limitations of sklearn in processing categorical data. Specifically, the sklearn Decision Tree classifier is unable to directly handle categorical data, and requires such data to be one-hot encoded into a vector of boolean value. This also allowed for the possibility of using Logistic Regression.

The most important feature to one-hot encode was the genre's feature. This column contained arrays of objects containing the name of the genre, and the genre id (Figure 3). This highly nested data type was unsuitable for pandas.getdummies, and data transformation needed to be applied here. This column was first mapped into a column of arrays of strings, which still would not be one-hot encodable directly by pandas. The sklearn MultiLableBinarizer was used here, as it was designed to be used to one-hot encode nested data. An example of the result can be seen in Figure 4.

Another key data transformation was to change first_air_date into an ordinal numeric datatype. This would allow for the classifiers we would use later to properly interpret the meaning of first_air_date, and also infer the order of dates, since sklearn’s logistic regression and decision tree classifier could not directly handle datetime objects, and the order of string dates would not be properly inferred by either classifier. Thus, all records in this column were transformed into an ordinal date integer, expressed as the number of days since the first day of the first year.

Finally, the runtime of TV shows is intuitively a very important feature when considering the success of a TV show. However, this data was also presented as a list of integers, representing the possible runtimes of a given TV show, as each episode could sometimes differ. Most of this data was a list of size 1 only, however to account for shows that contained multiple runtimes, this data was transformed from being a column of (lists of (integers)) to a column of mean(list(int)). Therefore, the model will use a mean_runtime feature to represent the runtime of a particular TV show.

Following these data transformations, the one-hot encoded columns of genres, ordinal first air dates, and mean runtimes were merged with the remaining features to form our start_df dataframe.

In order to predict a categorical variable as is required by classification models, vote_average needed to be converted into categorical data. This is a common data science task, and pandas.cut was used to apply this conversion. In order to determine an optimal number of bins as well as the bin cutoff points, prior work was consulted (stanford paper here). Additionally, we conducted our own data exploration into the frequency of vote_average ratings by plotting a histogram, and used those findings to determine the optimal number of bins and cutoff points. This data exploration was used to create 3 bins, and the vote_average column was transformed into categorical data, named rating_category. The rating_category column labels shows with ratings less than 7 as 'below average', between 7-8 as 'average', and above 8 as 'above average', corresponding to the shape of the histogram. This rating_category column was attached to start_df to finalize the initial dataframe, complete with all the explanatory variables that would be used at this phase of model training and the target variable. The histogram and statistics of vote_average are described in Figure 5. The statistics of rating_category are described in Figure 6.

Data cleaning completed included dropping all rows that contained NaN or null values in any of the columns of start_df. This included any rows that contained an empty list of genres or runtimes. Additionally, rows with vote_counts between 0-10 were also dropped. This aligned with the Stanford student's data cleaning strategy. Additionally, a histogram was plotted to show the frequency of vote_counts. It was determined through analyzing this data that dropping rows with vote_count between 0-10 was the most beneficial, since vote_average for these rows were very likely to be of low quality and representative power due to the extremely low number of votes. This can be seen in Figure 7.
In order to further explore the possible feature sets, TF-IDF vectorizer from sklearn was used to transform all words of all overviews of all TV shows present in the dataset into a matrix of TF-IDF values.

A significant amount of additional explanation is included directly in the supervised learning notebooks, and explains the thought process behind specific data transformations as they occur. This jupyter notebook has been submitted as part of this report, and a PDF copy of it is included in the appendix.

Movie Data

The movie data selected is downloaded from the Grouplens official site [12]. GroupLens is a research lab in the Department of Computer Science and Engineering at the University of Minnesota that works to curate relevant datasets that specialize in recommender systems, libraries, and geographic information systems. The set of movie data being used for the unsupervised learning tasks can be found under "MovieLens Latest Datasets" and is ml-latest-small dataset that consists of 100836 ratings and 3983 tag applications across 9742 movies and 610 users. Each user in the dataset has rated at least 20 movies. The tables being used are movies, ratings, and tags. See tables 2, 3 and 4 for Movies, Ratings and Tags table.

Table 2 consists of movieId, title with the year of release as well as the genres that are selected from the following:

Action	*Drama*	*Sci-Fi*
Adventure	*Fantasy*	*Thriller*
Animation	*Film-Noir*	*War*
Children's	*Horror*	*Western*
Comedy	*Musical*	*(no genres listed)*
Crime	*Mystery*	
Documentary	*Romance*	

The ratings table has each row representing one rating of one movie by one user and the user. Ratings are made on a 5-star scale, including half stars. The UTC timestamp represents the seconds in relation to when the rating was created. The tags table shows the tags or user-generated expressions regarding movies that the users have expressed on. Each tag is a word or a short phrase and the table also includes the UTC timestamp of which the tag was created.

Association Data Transformation

In order to transform and clean the data, we started off by merging the movie table and rating table by movieID, and only keeping the userId, movieId, rating, and title columns. The initial merged table is shown in Figure 8.

This dataframe was then pivoted in order to have the all userIds on the y-axis, or at axis=1, and each movie title as a column on the x-axis, or at axis=1. The values in each i-th row and j-th column is the rating that the user corresponding to the userId had rated the movie. There were 610 rows and 9719 movies, and this dataframe had shown many null values since not every user rates every movie. A sample of the data frame is shown in Figure 9.

In order to deal with the sparsity of this matrix and missing data, the solution was to sort the matrix first by the number of ratings that each user has made (in descending order) and then by the amount of ratings that each movie had gotten (also in descending order). Doing so allowed us to have the top left side of the dataframe to be the most dense. This also gave a clearer view of which movies were most popular amongst all the users that had rated movies. We proceeded to arbitrarily take the first 100 rows and 100 columns of data and called it the movieUserSortedMatrix. A sample of this data frame is shown in Figure 10.

This 100 x 100 matrix (movieUserSortedMatrix) has the most dense parts of the previous user-movie rating matrix. The movieUserSortedMatrix was then converted into a boolean matrix where if the value of the rating in each cell is less than 4, the value replaced the rating with a boolean value of 0. Rating values that were 4 or greater were replaced with a boolean value of 1. The sizing of this matrix as well as the boolean processing for the value of ratings were changed in the testing and results portion of this analysis. See Figure 11.

We would then proceed to create a list of movies corresponding to each userId row if the corresponding number under each movie is a 1 and consider it as a transaction with an itemset of movies. For example, the first transaction (corresponding to userId 413) would be:

```
 ['Forrest Gump (1994)', 'Shawshank Redemption, The (1994)', 'Pulp Fiction (1994)', 'Silence of the Lambs, The (1991)', 'Matrix, The (1999)', 'Star Wars: Episode IV - A New Hope (1977)', 'Jurassic Park (1993)', 'Braveheart (1995)', 'Terminator 2: Judgment Day (1991)', "Schindler's List (1993)", ... ]
```

In this case, since we picked the first 100 rows of the first 100 movies, there would be 100 transactions with each having a maximum of 100 items in each itemset. Note that we proceed to keep transactions only if there are at least 2 movies in each transaction in order to later generate association rules. The apriori algorithm was then used to create frequent itemsets with a minimum support of 0.8 and confidence of 1. The support value was chosen to be relatively high (80%) because in practice, having high confidence and low support does not validate evidence for a rule since the generated association rules would support only a low number of people. The results show frequent itemsets that we can use to generate association rules with 100% confidence. Again, these values can be changed in the results section of the analysis. See Figure 12 for sample data of association rules.

Text Mining and Clustering Data Transformation

The approach taken for text mining is by first merging the tag table and movie table by movieId in order to have a dataframe that consists of all the tags associated with a movie. The data was cleaned up by removing duplicate movieId - tag pairs and by grouping all rows by movieId to consolidate all tag words in its own row value under the tag column. A common aspect of all the movie tag words is that they were all words that could describe a movie's plot, cast, topic(s), genres and more. Therefore, tag data is fitting for potential analyses through text mining because topic modelling and sentiment analysis could be used to describe these movies. This could be fitting for businesses that would like to find movies that viewers have liked in the past by filtering for similar movies with common metadata. See Figure 13 for a sample of tag data.

In order to clean the tag text, we removed words that matched a regex that consisted of symbols or numbers as well as words that were included in the NLTK libraries' stopwords. Each word was also converted to its base form by using the library's lemmatizing function. The following Figure shows tag 2 as the corresponding list of "cleaned" tag words associated with the movieId. See Figure 14 for ample data of cleaning process for tag data

After achieving a clean version of the tag words that users have expressed towards each movie, the data was fed into the sklearn's TfidfVectorizer in order to achieve a tf-idf matrix that

numerically described how relevant a word is to this collection of tags. We took a step further to generate cosine similarity data amongst the tags for each movie by taking the tf-idf vectors, which allowed us to work with more numerical data. This opened opportunities for clustering amongst the distances of term vectors generated using cosine similarity scores, in order to further analyze patterns that could exist in a catalog of movies.

Results

Supervised Learning Results

Logistic Regression

Following the completion of data transformation and cleaning tasks, analysis started by using Logistic Regression for classification. This was done to simulate the prior work completed by the researchers and Stanford [1]. By simulating their work as closely as possible, differences in performance due to dataset differences could be identified, and further research and improvements could be made. The feature set used for this analysis was first_air_datetime_ordinal (numeric), number_of_seasons (numeric), number_of_episodes (numeric), vote_count (numeric), mean_runtime (numeric), genres (vector of booleans). The target variable was rating_category (categorical). The data was split into train and test sets, and the logistic regression model was then trained and tested (Figure 15).

Based on this initial result, we were able to conclude that the feature set suggested by the researchers at stanford university was at least somewhat correlated with predicting the target. We proceeded by performing additional data cleaning at this point.

After dropping rows with vote_counts between 0-10, a process described in the Jupyter notebook and in the “Data” section, the model was retrained and tested (Figure 16).

We were able to gain around 5% in prediction accuracy. However, at this point we realized that vote_count was data that is proprietary to TMDB, and streaming platforms such as Netflix would have much different viewership data. To make these models more usable for such corporations, we improve upon the previous works [1] by removing vote_count as an explanatory variable. The vote_count column was dropped and the model was retrained and tested (Figure 17).

A surprising finding was discovered here, as there was no loss of accuracy at all following the drop of the vote_count column. From the numeric data exploration above, it makes sense that this is the case, as the scatter plot of vote_average against vote_count showed very little correlation. Now, our working dataset is at its most independent and high performing state.

Decision Tree

A common approach in previous works relating to predicting the performance of movies was to use a Decision Tree classifier [4]. This approach was not taken by researchers at Stanford

[1]. Since our data was already in a format that was compatible with the sklearn Decision Tree classifier, we performed the same test/train split and trained and tested the decision tree model (Figure 18).

Unfortunately, using the decision tree directly reduced the accuracy of predictions. Some additional research was conducted, and prior works determining the optimal hyperparameters for decision trees was conducted [7]. The empirical analysis completed in this paper indicates that for the CART algorithm, which the sklearn decision tree classifier uses [21], the minsplit and minbucket hyperparameters are the most responsible for the performance of the final trees [7]. Specifically, the paper shows that low values of both minsplit and minbucket were correlated with better model performance. Therefore, we performed experimentation to see how altering min_samples_split and min_samples_leaf, which correspond to 'minsplit' and 'minbucket' respectively, would alter the performance of our decision tree (Figure 19).

From this analysis, it is clear that without a programmatic approach to selecting the min_samples_split and min_samples_leaf hyperparameters, there would be no improvements in model accuracy. Since the default values are not very large values, it is likely that they are close to the optimal values in terms of decision tree performance, according [7]. We abandoned further hyperparameter tuning at this point.

Adding to the feature set

Another common approach taken in prior works on predicting the performance of movies is to include data about the director, writers, and star actors as explanatory variables. This approach was not taken by Stanford researchers. Since our data is derived from TMDB, we did not have a 1:1 mapping of such data. Similar fields that are available to us from TMDB are: created_by, an array of objects, and production_companies, an array of objects. Star actors and directors were not available directly on the API, and the data processing that would be required to derive a usable dataset from the “crew members” data available made it unfeasible to use as part of this analysis. Exploration into these features revealed the following (Figure 20).

From this exploration, it is clear that neither of these candidate features would work well as explanatory variables. Both contain a very large amount of unique values, and the most frequently repeating value for both is an empty array. One-hot encoding such data would result in 1000+ columns, as well as the need to drop a significant number of rows due to missing data. Therefore, we abandon this approach.

TF-IDF

Some prior works on the prediction of movie performance, especially predicting movie performance based on data available prior to release, utilized the plot synopsis as a possible explanatory variable [5][6]. This approach was not seen in any prior works relating to predicting the performance of TV shows. An ‘overview’ column was available as part of the original TMDB dataset we created, and thus exploration into the possibility of using it in our prediction models was explored (Figure 21).

As expected, there were 1470 unique values since all TV shows have different synopses. However, a TF-IDF transformation would be applied here to make plot data usable by the classification algorithms anyways. We see that there are only 37 rows with missing data, and decide that the overview column is a good candidate to perform additional analysis on and add to our feature set.

The overview column was merged with our latest dataframe, third_df, in order to preserve row order prior to being transformed into a raw array during the TF-IDF vectorization process. Following this, the overview column was transformed using the sklearn TF-IDF Vectorizer, converted back into a dataframe, and merged with the original third_df to form our fourth_df. Some additional null values were created in this process, possibly due to words that contain non-unicode characters as part of the plot overviews. These rows were dropped (Figure 22). The data was split into test/train sets, and a new decision tree was trained and tested (Figure 23).

Unfortunately, this led to a large decrease in model performance. We trained the logistic regression model on this same data to compare results (Figure 24).

We don't see a significant improvement here.

Finally, we investigate the possibility of using the TF-IDF plot synopsis vector along with genre as the only explanatory variables (Figure 25).

Interestingly, we observe that even training the logistic regression model on only the genres vector and tf-idf plot synopsis vector alone resulted in performance that was almost equal to the performance of our base logistic regression model.

Additional model validation was conducted on the highest performing models, i.e. the logistic regression models, using k-fold cross validation, with 10 folds (Figure 26).

We see that model performance is similar across all three configurations of feature sets.

Unsupervised Learning Results

Association

The confidence measurement of an association rule allows us to measure how frequent the items in the rule occur out of all the transactions with the corresponding antecedents. The support of an itemset however is indicative of the number of transactions that contain all the items in that item set. In order to maintain a level of control for all trials of experiments, the size of all data used in the apriori algorithm will contain 100 itemsets of movies with each itemset containing at most 100 movies. As described in the Association Data Transformation section, the first 100 itemsets correspond to the movies that the first 100 users (sorted in descending rate activity i.e., users who rate the most) have rated. Itemsets will contain movies that have been rated a particular amount by these users such as higher than 4, or simply rated a 5.

In order to find common and meaningful trends in the dataset, it is important to be aggressive with the support value being set for finding association rules, since we want to see common combinations movies that are watched by most users. In addition, a higher min. confidence level set will help validate the existence of the items in the consequent given that the items in the antecedents exist out of all transactions. In practice and for businesses to find association rules that have a high level of support and confidence in order to find frequent itemsets of movies that most users would enjoy.

Trial 1
Itemsets Size: 100 items with each having maximum 100 movies
Itemsets: Transactions include movies that are rated at least 4 by users
Minimum Support: 0.6
Min Confidence: 0.8
Note: See Figure 27 for results.

From the results we see that it is common that movies from a series are in the same association rule with other movies from the same series. An explanation for this is that most avid movie watchers usually watch all movies of popular movie series such as Star Wars shown. We also see common combinations of Pulp Fiction, Fight Club and Matrix, which could show that users truly enjoyed all three movies and perhaps these movies should always be within a catalog since they're classic movies that most people have highly rated altogether.

Trial 2
Itemsets Size: 100 items with each having maximum 100 movies
Itemsets: Transactions include movies that are rated at least 4 by users
Minimum Support: 0.5
Min Confidence: 0.95
Note: See Figure 28 for results.

With a tighter confidence level, and slightly more relaxed minimum support, we still see common movies from the first prior, which includes Star Wars and it is noted that the pattern of movies from the same series in an association rule still exist.

Trial 3
Itemsets Size: 100 items with each having maximum 100 movies
Itemsets: Transactions include movies that are rated between 3 to 4 by users
Minimum Support: 0.5
Min Confidence: 0.9
Note: See Figure 29 for results.

By changing picking transactions of movies that have only been rated between 3 to 4, we see other patterns of commonly watched Star Wars movies, Pulp Fiction, Matrix and Fight club merging in the same association rules.

Trial 4
Itemsets Size: 100 items with each having maximum 100 movies

Itemsets: Transactions include movies that are rated between 2.5 to 4 by users
Minimum Support: 0.4
Min Confidence: 0.7
Note: See Figure 30 for results.

By altering the selection of movies through ratings more, we see that the results shown have more variety of movies, and potentially are more fitting to smaller subsets of groups as we lower the support and confidence levels.

Trial 5:
Itemsets Size: 30 items with each having maximum 30 movies
Itemsets: Transactions include movies that are rated at least 4 by users
Minimum Support: 0.85
Min Confidence: 0.1
Note: See Figure 31 for results.

By constraining the itemset sizes, we see that one of the most common association rules is that most users who watched and highly rated Fight Club will also watch and highly rate Pulp Fiction.

By analyzing the performance of this association model, it is apparent that the model searches for the most commonly watched combinations and frequent itemsets of movies. In addition, a flaw with this model is that it works to generate frequent itemsets that are fitting to the entire dataset if there is a high support level input. Therefore, the frequent itemsets and association rules generated with a high support level and confidence level are not reflective of a subset of users' frequent itemsets because it takes all movies that are common to all transactions instead. A common pattern from this technique involves similar movies, perhaps from the same movie series. We also see that movies following similar genres of comedy, thriller, action are shown to be involved in the association rules which can allow businesses to infer that these genres are commonly liked by most users.

Text Mining and Clustering

The features that are analyzed in this section of unsupervised learning are the tags and movies. Since most of the features in this movie dataset were not numerical, we wanted to generate relevant values that can be used for clustering. By applying the text vectorizer to all tag words for each movie, we generated document term vectors and were able to apply topic modelling which is shown in Figure 32.

With the topic modelling results, we can see sub categories of genres that include action, comedy, history, politics, children and family, and more. For example, topic 10 entails a Stephen King - like cluster of words that include comedy, mystery, and some comedy. For topic 8, we see political movies that address heavier topics in the world such as terrorism, racism, and we see that a word to describe some of these movies are thrilling -- possible due to the more serious topics in this cluster. Upon calculating cosine similarity scores of the document term vectors, we took another step to create a function that returns the first subset of similar movies that cluster

with a movie ID inputted. For example, since the movieId of Toy's Story is 1, we input 1 and get the first 20 results shown in Figure 33.

We see that the first half of the dataset are similar animated and comedy movies catered towards family and children and we then see the results gear off to still comedy, but less of family and children content. Since the first subset of movies do have content rather similar to Toy Story, we can validate that the system works for gaining similar movies based on the tag content. This method could be rather useful if movie platforms are looking for similar movies to a movie that is extremely well liked, in order to grab a handful for movies to add to the catalogs that viewers would also like. It could also work in the opposite way, where if a company would like to take a subset of movies out of the catalog, they can search for similar movies that have not been well liked historically with similar logic.

Upon using KMeans clustering to cluster the similarity scores between document term vectors, we researched to find that Multidimensional Scaling (MDS) is a common and effective method of taking document terms and mining data in such a way that they can be visualized to show similarities. Figure 34 in the appendix shows the clustered movies where K was set as 8 (from Elbow method of cosine similarity data) and where every 30 plots have their movie title printed. This was due to the large set of movies that were being printed and overlapping each other, making the diagram unreadable. The figure shows that there's one very large cluster compared to all the clusters and the titles show that perhaps the smaller clusters had distinguished topics compared to the other clusters. In addition, it is hypothesized that the large cluster includes a topic that there were more movies for, such as a combination of romance, comedy, and action. It is also hypothesized that the smaller clusters involved movies with topics that are less commonly watched in movies, such as thriller (see Poltergeist plot) and possibly more adult appropriate movies. By visualizing groups of movies with similar content, it allows businesses further insight into possibly which areas or categories of movies they should invest more in, as well as which subcategories of genres are more common than others.

Conclusion

The main findings of the supervised learning task for predicting rating_category indicate that a logistic regression model is superior to a decision tree in terms of algorithm prediction accuracy. This is potentially due to the fact that sklearn decision trees do not currently support categorical features directly, and one-hot encoded categorical features are more well suited towards Logistic Regression as opposed to Decision Trees. It was determined that the optimal feature set included the first_air_datetime_ordinal (numeric), number_of_seasons (numeric), number_of_episodes (numeric), mean_runtime (numeric), genres (vector of booleans). Analysis indicated that the overview TF-IDF with genres alone was capable of matching prediction accuracy compared to a logistic regression model trained on all other explanatory variables. We conclude that our optimal feature set allows for streaming platforms to estimate the performance and popularity of TV shows they wish to add to their catalog, and that further analysis and optimization of the overview TF-IDF vector is likely to further enhance the predictive power of the final logistic regression model.

The main findings of the unsupervised learning algorithm is that it curated frequent itemsets of movies that should be placed together in a catalog as well as recommendations of itemsets for the business to add to the catalog in order to capture the enjoyment of all users on the platform. In addition, the Text Mining Clustering analysis proved to that it can generate a list of similar movies by content through text vectorizing, however the clustered data from the similarity scores of document term vectors seemed to show which topics were the most common across all movies, and which topics were less popular. Both analysis provided insight and trends as to how to make decisions regarding adding and removing from the catalogs based on content, or associations on movies.

References

[1] Y. Chai, Y. Xu, and Z. Liu, “Behind the TV Shows: Top-Rated Series Characterization and Audience Rating Prediction” *cs.229.stanford.edu*. [Online]. Available: http://cs229.stanford.edu/proj2015/256_report.pdf. [Accessed: 09-Dec-2020].

[2] T. Kadam, G. Saraf, V. Dewadkar, and P. J. Chate, “TV Show Popularity Prediction using Sentiment Analysis in Social Network” *International Research Journal of Engineering and Technology (IRJET)* [Online]. Available: https://www.irjet.net/archives/V4/i11/IRJET-V4I11193.pdf. [Accessed: 09-Dec-2020].

[3] G. Holland, “Pandas Cut – Continuous to Categorical,” *AbsentData*, 26-Sep-2020. [Online]. Available: https://www.absentdata.com/pandas/pandas-cut-continuous-to-categorical/. [Accessed: 09-Dec-2020].

[4] P. Shahrivar, and C. Jernbacker, “Predicting movie success using machine learning techniques” *Examensarbete Inom Teknik* [Online]. Available: https://www.diva-portal.org/smash/get/diva2:1106715/FULLTEXT01.pdf. [Accessed: 09-Dec-2020].

[5] Michael T. Lash & Kang Zhao (2016) Early Predictions of Movie Success: The Who, What, and When of Profitability, *Journal of Management Information Systems*, 33:3, 874-903, DOI: 10.1080/07421222.2016.1243969

[6] A. Bhave, H. Kulkarni, V. Biramane and P. Kosamkar, "Role of different factors in predicting movie success," *2015 International Conference on Pervasive Computing (ICPC)*, Pune, 2015, pp. 1-4, doi: 10.1109/PERVASIVE.2015.7087152.

[7] R. G. Mantovani, T. Horváth, R. Cerri, S. B. Junior, J. Vanschoren, and A. C. P. de L. F. de Carvalho, “An empirical study on hyperparameter tuning of decision trees,” *arXiv.org*, 12-Feb-2019. [Online]. Available: https://arxiv.org/abs/1812.02207. [Accessed: 09-Dec-2020].

[8] The Movie Database. Retrieved from: "https://www.themoviedb.org/"

[9] M. Grootendorst, “Cluster Analysis: Create, Visualize and Interpret Customer Segments,” *Medium*, 10-Sep-2020. [Online]. Available: https://towardsdatascience.com/cluster-analysis-create-visualize-and-interpret-customer-segments-474e55d00ebb. [Accessed: 09-Dec-2020].

[10] M. Mithrakumar, “How to tune a Decision Tree?,” *Medium*, 12-Nov-2019. [Online]. Available: https://towardsdatascience.com/how-to-tune-a-decision-tree-f03721801680. [Accessed: 09-Dec-2020].

[11] G. Malato, “Visualize multidimensional datasets with MDS,” *Medium*, 07-Oct-2020. [Online]. Available: https://towardsdatascience.com/visualize-multidimensional-datasets-with-mds-64d7b4c16eaa. [Accessed: 09-Dec-2020].

[12] “MovieLens,” *GroupLens*, 21-May-2020. [Online]. Available: https://grouplens.org/datasets/movielens/. [Accessed: 09-Dec-2020].

[13] Ahmed, “https://www.kaggle.com/ibtesama/getting-started-with-a-movie-recommendation system,” *getting-started-with-a-movie-recommendation-system*, May-2020. [Online]. Available: https://www.kaggle.com/ibtesama/getting-started-with-a-movie-recommendation-system.

[14] M. Meng, *Movie Recommendation System based on MovieLens*, 13-Aug-2020. [Online]. Available: https://towardsdatascience.com/movie-recommendation-system-based-on-movielens-ef0df580cd0e. [Accessed: 09-Dec-2020].

[15] Jagannath Neupane, *Analysis of MovieLens dataset (Beginner'sAnalysis)*, 06-Nov-2017. [Online]. Available: https://www.kaggle.com/jneupane12/analysis-of-movielens-dataset-beginner-sanalysis. [Accessed: 09-Dec-2020].

[16] V. Roman, “Unsupervised Classification Project: Building a Movie Recommender with Clustering Analysis and...,” *Medium*, 21-Mar-2019. [Online]. Available: https://towardsdatascience.com/unsupervised-classification-project-building-a-movie-recommender-with-clustering-analysis-and-4bab0738efe6. [Accessed: 09-Dec-2020].

[17] P. Bandurski, *MovieLens Dataset Analysis*, 03-Feb-2017. [Online]. Available: https://www.kaggle.com/redroy44/movielens-dataset-analysis. [Accessed: 09-Dec-2020].

[18] R. Jain, “A beginner's tutorial on the apriori algorithm in data mining with R implementation,” *A beginner’s tutorial on the apriori algorithm in data mining with R implementation*, 05-Aug-2019. [Online]. Available: https://www.hackerearth.com/blog/developers/beginners-tutorial-apriori-algorithm-data-mining-r-implementation/. [Accessed: 09-Dec-2020].

[19] D. Barman and N. Chowdhury, “Movie Business Trend Prediction using Market Basket Analysis,” 09-Jul-2013. [Online]. Available: https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.403.361&rep=rep1&type=pdf.

[20] I. Dabbura, *K-means Clustering: Algorithm, Applications, Evaluation Methods, and Drawbacks*, 10-Aug-2020. [Online]. Available: https://towardsdatascience.com/k-means-clustering-algorithm-applications-evaluation-methods-and-drawbacks-aa03e644b48a. [Accessed: 09-Dec-2020].

[21] 1.10.Decision Trees, *scikit-learn*. [Online]. Available: https://scikit-learn.org/stable/modules/tree.html. [Accessed: 09-Dec-2020].

Appendix

Tables

Explanatory Variable (Feature)	**Data Type**
genres	Vector of Booleans (one-hot encoded categorical data)
first_air_datetime_ordinal	Integer representing number of days since the first day of the first year
number_of_seasons	integer
number_of_episodes	integer
mean_runtime	integer
overview_tfidf	Vector of TF-IDF of words contained in the overview

Table 1: Supervised learning explanatory variables

movieId	title	genres
1	Toy Story (1995)	Adventure|Animation|Children|Comedy|Fantasy
2	Jumanji (1995)	Adventure|Children|Fantasy
3	Grumpier Old Men (1995)	Comedy|Romance
4	Waiting to Exhale (1995)	Comedy|Drama|Romance
5	Father of the Bride Part II (1995)	Comedy

Table 2: Movies Table

userId	movieId	rating	timestamp
1	1	4.0	964982703
1	3	4.0	964981247
1	6	4.0	964982224
1	47	5.0	964983815

1	50	5.0	964982931

Table 3: Ratings Table

userId	movieId	tag	timestamp
2	60756	funny	964982703
2	60756	Highly Quotable	964981247
2	60756	will ferrell	964982224
2	89774	Boxing story	964983815
2	89774	MMA	964982931

Table 4: Tags Table

Figures

```
]: numeric.describe()
```

]:

	vote_average	popularity	number_of_seasons	number_of_episodes	id	vote_count
count	1557.000000	1557.000000	1557.000000	1557.000000	1557.000000	1557.000000
mean	7.096275	30.688473	4.712909	166.496468	48528.310212	197.073218
std	1.556956	25.204725	6.316557	522.294566	34150.439568	360.319468
min	0.000000	9.965000	1.000000	1.000000	29.000000	0.000000
25%	6.800000	17.939000	1.000000	18.000000	10317.000000	26.000000
50%	7.400000	22.861000	3.000000	51.000000	60654.000000	87.000000
75%	7.900000	32.855000	5.000000	128.000000	76231.000000	226.000000
max	10.000000	253.691000	62.000000	7771.000000	112878.000000	6222.000000

Figure 1: Numeric variable statistics

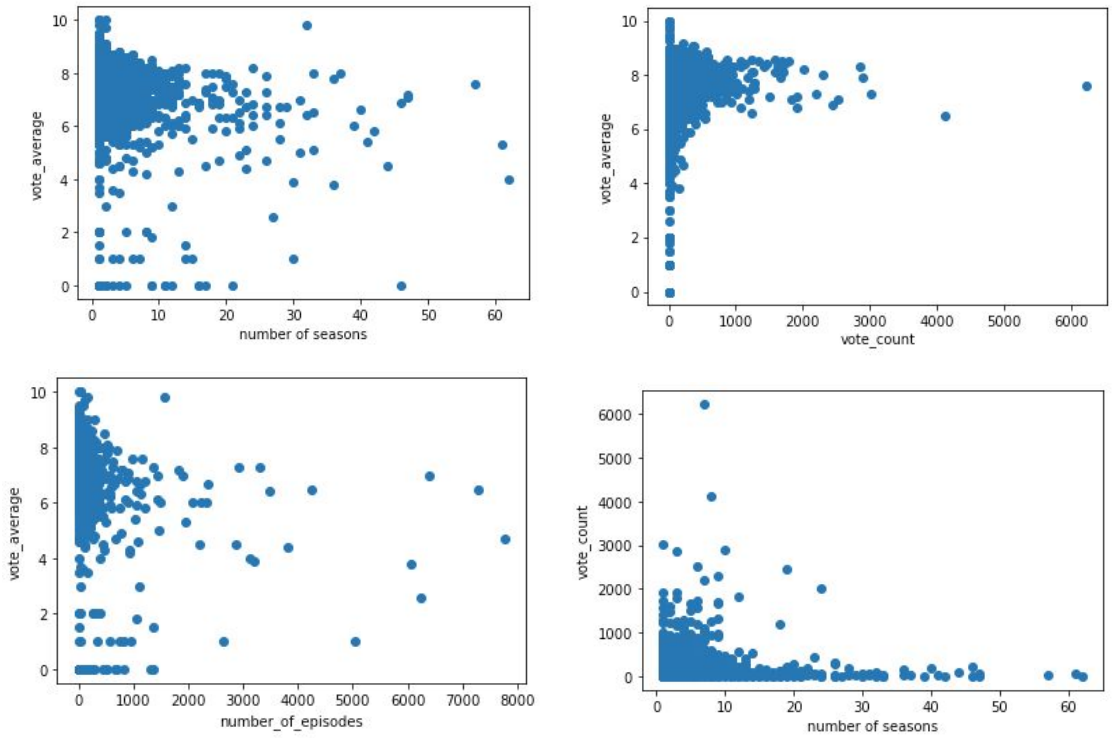


Figure 2: Numeric explanatory variables vs. vote average and vote count

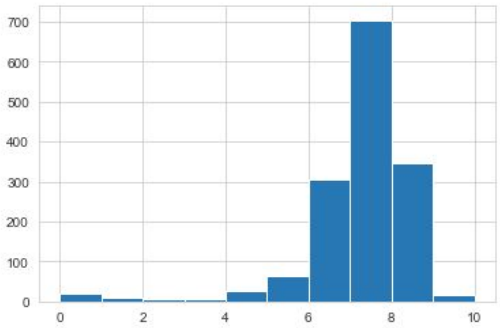
```
genres_col.head()
```

```
0    [{'id': 10759, 'name': 'Action & Adventure'}, ...
1    [{'id': 10759, 'name': 'Action & Adventure'}, ...
2                          [{'id': 18, 'name': 'Drama'}]
3    [{'id': 18, 'name': 'Drama'}, {'id': 10765, 'n...
4                          [{'id': 18, 'name': 'Drama'}]
Name: genres, dtype: object
```

Figure 3: Genres prior to transformation

Action	Action & Adventure	Adventure	Animation	...	News	Reality	Romance	Sci-Fi & Fantasy	Science Fiction	Soap	Talk	War	War & Politics	Western
0	1	0	0	...	0	0	0	0	0	0	0	0	0	0
0	1	0	0	...	0	0	0	0	0	0	0	0	0	0
0	0	0	0	...	0	0	0	0	0	0	0	0	0	0
0	0	0	0	...	0	0	0	1	0	0	0	0	0	0
0	0	0	0	...	0	0	0	0	0	0	0	0	0	0

Figure 4: Genres after one-hot encoding

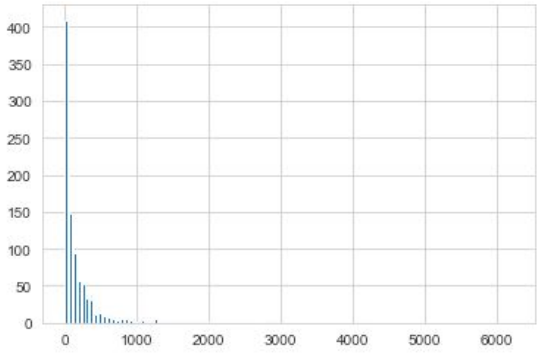



```
: start_df['vote_average'].describe()

: count    1507.000000
  mean        7.162973
  std         1.391311
  min         0.000000
  25%         6.800000
  50%         7.400000
  75%         7.900000
  max        10.000000
  Name: vote_average, dtype: float64
```

Figure 5: vote_average frequency and statistics

```
: start_df['rating_category'].describe()

: count         1507
  unique           3
  top        average
  freq          1044
  Name: rating_category, dtype: object
```

Figure 6: rating_category statistics




```
: df.loc[(df['vote_count'] >= 0) & (df['vote_count'] <= 10)].shape
: (197, 33)

: df.loc[(df['vote_count'] >= 10) & (df['vote_count'] <= 20)].shape
: (123, 33)

: df.loc[(df['vote_count'] >= 20) & (df['vote_count'] <= 30)].shape
: (100, 33)
```

Figure 7: vote_count frequencies and low vote_count occurrences

	userId	movieId	rating	timestamp	title	genres
0	1	1	4.0	964982703	Toy Story (1995)	Adventure|Animation|Children|Comedy|Fantasy
1	5	1	4.0	847434962	Toy Story (1995)	Adventure|Animation|Children|Comedy|Fantasy
2	7	1	4.5	1106635946	Toy Story (1995)	Adventure|Animation|Children|Comedy|Fantasy
3	15	1	2.5	1510577970	Toy Story (1995)	Adventure|Animation|Children|Comedy|Fantasy
4	17	1	4.5	1305696483	Toy Story (1995)	Adventure|Animation|Children|Comedy|Fantasy

Figure 8: Merged movie and rating table

title	'71 (2014)	'Hellboy': The Seeds of Creation (2004)	'Round Midnight (1986)	'Salem's Lot (2004)	'Til There Was You (1997)	'Tis the Season for Love (2015)	'burbs, The (1989)	'night Mother (1986)	(500) Days of Summer (2009)	*batteries not included (1987)	...	10th Victim, The (La decima vittima) (1965)	11'09"01 - September 11 (2002)	11:14 (2003)	11th Hour, The (2007)	12 Angry Men (1957)	12 Angry Men (1997)
userId																	
1	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	...	NaN	NaN	NaN	NaN	NaN	NaN
2	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	...	NaN	NaN	NaN	NaN	NaN	NaN
3	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	...	NaN	NaN	NaN	NaN	NaN	NaN
4	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	...	NaN	NaN	NaN	NaN	5.0	NaN
5	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	...	NaN	NaN	NaN	NaN	NaN	NaN

Figure 9: Sample data of user-movie rating matrix

title	Forrest Gump (1994)	Shawshank Redemption, The (1994)	Pulp Fiction (1994)	Silence of the Lambs, The (1991)	Matrix, The (1999)	Star Wars: Episode IV - A New Hope (1977)	Jurassic Park (1993)	Braveheart (1995)	Terminator 2: Judgment Day (1991)	Schindler's List (1993)	...	Ace Ventura: Pet Detective (1994)	Memento (2000)	Mask, The (1994)	Pirates of the Caribbean: The Curse of the Black Pearl (2003)
413	5.0	5.0	5.0	4.0	5.0	5.0	4.0	5.0	5.0	4.0	...	2.0	5.0	4.0	4.5
589	5.0	4.5	4.5	3.5	4.0	5.0	4.0	4.0	4.5	5.0	...	3.0	3.5	3.0	4.0
479	5.0	5.0	4.0	4.5	5.0	4.5	5.0	5.0	4.5	5.0	...	3.5	4.0	4.0	3.0
273	4.5	4.5	5.0	4.0	4.0	3.0	3.5	4.5	4.5	4.0	...	4.5	4.5	4.0	4.0
67	3.5	3.0	2.0	3.5	4.5	5.0	3.5	2.5	3.5	4.0	...	2.5	4.0	3.0	5.0

Figure 10: Sample data of sorted user-movie rating matrix

title	Forrest Gump (1994)	Shawshank Redemption, The (1994)	Pulp Fiction (1994)	Silence of the Lambs, The (1991)	Matrix, The (1999)	Star Wars: Episode IV - A New Hope (1977)	Jurassic Park (1993)	Braveheart (1995)	Terminator 2: Judgment Day (1991)	Schindler's List (1993)	...
413	1	1	1	1	1	1	1	1	1	1	...
589	1	1	1	0	1	1	1	1	1	1	...
473	0	1	1	1	1	1	1	0	1	1	...
479	1	1	1	1	1	1	1	1	1	1	...
67	0	0	0	0	1	1	0	0	0	1	...

Figure 11: Sample data of boolean user-movie rating matrix

	antecedents	consequents	support	confidence
0	(Fight Club (1999))	(Pulp Fiction (1994))	0.866667	1.0
1	(Usual Suspects, The (1995))	(Pulp Fiction (1994))	0.800000	1.0
2	(Star Wars: Episode IV - A New Hope (1977))	(Star Wars: Episode V - The Empire Strikes Back (1980))	0.833333	1.0
3	(Star Wars: Episode VI - Return of the Jedi (1983))	(Star Wars: Episode V - The Empire Strikes Back (1980))	0.833333	1.0
4	(Star Wars: Episode IV - A New Hope (1977), Pulp Fiction (1994))	(Star Wars: Episode V - The Empire Strikes Back (1980))	0.800000	1.0
5	(Star Wars: Episode VI - Return of the Jedi (1983), Pulp Fiction (1994))	(Star Wars: Episode V - The Empire Strikes Back (1980))	0.800000	1.0

Figure 12: Sample data of generated association rules

	movieId	tag	title
0	1	pixar fun	Toy Story (1995)
1	2	fantasy magic board game Robin Williams game	Jumanji (1995)
2	3	moldy old	Grumpier Old Men (1995)
3	5	pregnancy remake	Father of the Bride Part II (1995)
4	7	remake	Sabrina (1995)

Figure 13: Sample of Tag Data

	movieId	tag	tag1	tag2
0	1	pixar fun	pixar fun	['pixar', 'fun']
1	2	fantasy magic board game Robin Williams game	fantasy magic board game robin williams game	['fantasy', 'magic', 'board', 'game', 'robin',...
2	3	moldy old	moldy old	['moldy', 'old']
3	5	pregnancy remake	pregnancy remake	['pregnancy', 'remake']
4	7	remake	remake	['remake']

Figure 14: Sample data of Cleaning Process for Tag data

```
# Predict the classes of the testing data set
class_predict = start_model.predict(X_test)

# Compare the predicted classes to the actual test classes
metrics.accuracy_score(Y_test,class_predict)
```

```
0.6816976127320955
```

Figure 15: logistic regression 1 accuracy

```
# Fit the model
second_model.fit(X_train,Y_train)

# Predict the classes of the testing data set
class_predict = second_model.predict(X_test)

# Compare the predicted classes to the actual test classes
metrics.accuracy_score(Y_test,class_predict)
```

```
]: 0.7347560975609756
```

Figure 16: logistic regression 2 accuracy

```
# Fit the model
third_model.fit(X_train,Y_train)

# Predict the classes of the testing data set
class_predict = third_model.predict(X_test)

# Compare the predicted classes to the actual test classes
metrics.accuracy_score(Y_test,class_predict)
```

```
: 0.7347560975609756
```

Figure 17: logistic regression 3 accuracy

```
# Fit the model
classifier_model.fit(X_train,Y_train)

# Predict the classes of the testing data set
class_predict = classifier_model.predict(X_test)

# Compare the predicted classes to the actual test classes
metrics.accuracy_score(Y_test,class_predict)
```

```
0.6951219512195121
```

Figure 18: Decision tree 1 accuracy

```
# multiply both default values by 5
classifier_model_2 = DecisionTreeClassifier(min_samples_split=10, min_samples_leaf=5)

# Fit the model
classifier_model.fit(X_train,Y_train)

# Predict the classes of the testing data set
class_predict = classifier_model.predict(X_test)

# Compare the predicted classes to the actual test classes
metrics.accuracy_score(Y_test,class_predict)
```

```
0.6707317073170732
```

```
# multiply both default values by 5
classifier_model_2 = DecisionTreeClassifier(min_samples_split=10, min_samples_leaf=5)

# Fit the model
classifier_model.fit(X_train,Y_train)

# Predict the classes of the testing data set
class_predict = classifier_model.predict(X_test)

# Compare the predicted classes to the actual test classes
metrics.accuracy_score(Y_test,class_predict)
```

```
0.6707317073170732
```

```
# randomized trial
classifier_model_3 = DecisionTreeClassifier(min_samples_split=9, min_samples_leaf=3)

# Fit the model
classifier_model.fit(X_train,Y_train)

# Predict the classes of the testing data set
class_predict = classifier_model.predict(X_test)

# Compare the predicted classes to the actual test classes
metrics.accuracy_score(Y_test,class_predict)
```

```
0.6890243902439024
```

Figure 19: Decision tree hyperparameter tuning results

```
df['created_by'].describe()
```

```
count      1507
unique     1203
top          []
freq        305
Name: created_by, dtype: object
```

```
df['production_companies'].describe()
```

```
count      1507
unique      983
top           []
freq        323
Name: production_companies, dtype: object
```

Figure 20: additional feature exploration

```
df['overview'].describe()
```

```
count                                                  1470
unique                                                 1470
top       The story revolves around Shu Ouma, a high sch...
freq                                                      1
Name: overview, dtype: object
```

```
df['overview'].isnull().sum()
```

```
37
```

Figure 21: overview feature exploration

```
fourth_df.head()
```

	Action	Action & Adventure	Adventure	Animation	Comedy	Crime	...	zuo	zuri	zwick	álex	álvarez	ángel	écija	ömer	über	どうぞ
)	0	1	0	0	0	0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
)	0	1	0	0	0	1	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
)	0	0	0	0	0	0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
)	0	0	0	0	0	0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
)	0	0	0	0	0	0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0

Figure 22: TF-IDF joined with original feature set

```
# Fit the model
classifier_model.fit(X_train,Y_train)

# Predict the classes of the testing data set
class_predict = classifier_model.predict(X_test)

# Compare the predicted classes to the actual test classes
metrics.accuracy_score(Y_test,class_predict)
```

```
0.6173285198555957
```

Figure 23: decision tree with tf-idf performance

```
# Make our logistic regression model
fourth_log = LogisticRegression()

# Fit the model
fourth_log.fit(X_train,Y_train)

# Predict the classes of the testing data set
class_predict = fourth_log.predict(X_test)

# Compare the predicted classes to the actual test classes
metrics.accuracy_score(Y_test,class_predict)
```

```
: 0.7509025270758123
```

Figure 24: logistic regression with tf-idf performance

```
# Fit the model
fifth_log.fit(X_train,Y_train)

# Predict the classes of the testing data set
class_predict = fifth_log.predict(X_test)

# Compare the predicted classes to the actual test classes
metrics.accuracy_score(Y_test,class_predict)
```

```
0.7364620938628159
```

Figure 25: logistic regression on tf-idf only performance

Original logistic regression with vote_count dropped

```
# create dataset
X = third_df.drop(['rating_category', 'overview'], axis = 1)
Y = third_df['rating_category']

# create model
third_model = LogisticRegression()

# evaluate model
scores = cross_val_score(third_model, X, Y, scoring='accuracy', cv=10)

# report performance
print('Accuracy: %.3f (%.3f)' % (mean(scores), std(scores)))
```

```
Accuracy: 0.734 (0.011)
```

Logistic regression including TF-IDF

```
# create dataset
X = fourth_df.drop('rating_category', axis=1)
Y = fourth_df['rating_category']

# create model
fourth_log = LogisticRegression()

# evaluate model
scores = cross_val_score(fourth_log, X, Y, scoring='accuracy', cv=10)

# report performance
print('Accuracy: %.3f (%.3f)' % (mean(scores), std(scores)))
```

```
Accuracy: 0.725 (0.011)
```

Logistic regression of TF-IDF only

```
# create dataset
X = fifth_df.drop('rating_category', axis=1)
Y = fifth_df['rating_category']

# create model
fifth_log = LogisticRegression()

# evaluate model
scores = cross_val_score(fifth_log, X, Y, scoring='accuracy', cv=10)

# report performance
print('Accuracy: %.3f (%.3f)' % (mean(scores), std(scores)))
```

```
Accuracy: 0.737 (0.023)
```

Figure 26: Logistic regression k-fold cross validation

	antecedents	consequents	support	confidence
0	(Pulp Fiction (1994))	(Fight Club (1999))	0.64	0.810127
1	(Fight Club (1999))	(Pulp Fiction (1994))	0.64	0.914286
2	(Pulp Fiction (1994))	(Matrix, The (1999))	0.64	0.810127
3	(Matrix, The (1999))	(Pulp Fiction (1994))	0.64	0.842105
4	(Matrix, The (1999))	(Star Wars: Episode IV - A New Hope (1977))	0.62	0.815789
5	(Star Wars: Episode IV - A New Hope (1977))	(Matrix, The (1999))	0.62	0.805195
6	(Star Wars: Episode V - The Empire Strikes Back (1980))	(Matrix, The (1999))	0.62	0.826667
7	(Matrix, The (1999))	(Star Wars: Episode V - The Empire Strikes Back (1980))	0.62	0.815789
8	(Silence of the Lambs, The (1991))	(Pulp Fiction (1994))	0.60	0.909091
9	(Star Wars: Episode IV - A New Hope (1977))	(Pulp Fiction (1994))	0.63	0.818182
10	(Star Wars: Episode V - The Empire Strikes Back (1980))	(Pulp Fiction (1994))	0.61	0.813333
11	(Star Wars: Episode V - The Empire Strikes Back (1980))	(Star Wars: Episode IV - A New Hope (1977))	0.68	0.906667
12	(Star Wars: Episode IV - A New Hope (1977))	(Star Wars: Episode V - The Empire Strikes Back (1980))	0.68	0.883117
13	(Star Wars: Episode VI - Return of the Jedi (1983))	(Star Wars: Episode IV - A New Hope (1977))	0.63	0.926471
14	(Star Wars: Episode IV - A New Hope (1977))	(Star Wars: Episode VI - Return of the Jedi (1983))	0.63	0.818182
15	(Star Wars: Episode VI - Return of the Jedi (1983))	(Star Wars: Episode V - The Empire Strikes Back (1980))	0.64	0.941176
16	(Star Wars: Episode V - The Empire Strikes Back (1980))	(Star Wars: Episode VI - Return of the Jedi (1983))	0.64	0.853333
17	(Star Wars: Episode VI - Return of the Jedi (1983), Star Wars: Episode V - The Empire Strikes Back (1980))	(Star Wars: Episode IV - A New Hope (1977))	0.60	0.937500
18	(Star Wars: Episode VI - Return of the Jedi (1983), Star Wars: Episode IV - A New Hope (1977))	(Star Wars: Episode V - The Empire Strikes Back (1980))	0.60	0.952381

Figure 27: Trial 1 Results

	antecedents	consequents	support	confidence
0	(Godfather: Part II, The (1974))	(Godfather, The (1972))	0.53	1.000000
1	(Reservoir Dogs (1992))	(Pulp Fiction (1994))	0.55	0.982143
2	(Godfather, The (1972), Star Wars: Episode V - The Empire Strikes Back (1980))	(Star Wars: Episode IV - A New Hope (1977))	0.51	0.962264
3	(Matrix, The (1999), Star Wars: Episode VI - Return of the Jedi (1983))	(Star Wars: Episode V - The Empire Strikes Back (1980))	0.52	0.962963
4	(Star Wars: Episode IV - A New Hope (1977), Star Wars: Episode VI - Return of the Jedi (1983))	(Star Wars: Episode V - The Empire Strikes Back (1980))	0.60	0.952381

Figure 28: Trial 2 Results

10	(Terminator 2: Judgment Day (1991))	(Star Wars: Episode V - The Empire Strikes Back (1980))	0.51	0.944444
11	(Matrix, The (1999), Fight Club (1999))	(Pulp Fiction (1994))	0.53	0.929825
12	(Fight Club (1999), Star Wars: Episode IV - A New Hope (1977))	(Pulp Fiction (1994))	0.50	0.943396
13	(Star Wars: Episode V - The Empire Strikes Back (1980), Fight Club (1999))	(Pulp Fiction (1994))	0.51	0.944444
14	(Star Wars: Episode V - The Empire Strikes Back (1980), Godfather, The (1972))	(Star Wars: Episode IV - A New Hope (1977))	0.51	0.962264
15	(Star Wars: Episode VI - Return of the Jedi (1983), Matrix, The (1999))	(Star Wars: Episode IV - A New Hope (1977))	0.50	0.925926
16	(Star Wars: Episode VI - Return of the Jedi (1983), Matrix, The (1999))	(Star Wars: Episode V - The Empire Strikes Back (1980))	0.52	0.962963
17	(Pulp Fiction (1994), Star Wars: Episode V - The Empire Strikes Back (1980))	(Star Wars: Episode IV - A New Hope (1977))	0.56	0.918033
20	(Star Wars: Episode V - The Empire Strikes Back (1980), Raiders of the Lost Ark (Indiana Jones and the Raiders of the Lost Ark) (1981))	(Star Wars: Episode IV - A New Hope (1977))	0.51	0.927273
21	(Raiders of the Lost Ark (Indiana Jones and the Raiders of the Lost Ark) (1981), Star Wars: Episode IV - A New Hope (1977))	(Star Wars: Episode V - The Empire Strikes Back (1980))	0.51	0.927273
22	(Star Wars: Episode VI - Return of the Jedi (1983), Star Wars: Episode V - The Empire Strikes Back (1980))	(Star Wars: Episode IV - A New Hope (1977))	0.60	0.937500
23	(Star Wars: Episode VI - Return of the Jedi (1983), Star Wars: Episode IV - A New Hope (1977))	(Star Wars: Episode V - The Empire Strikes Back (1980))	0.60	0.952381

Figure 29: Trial 3 Results

	antecedents	consequents	support	confidence
0	(Jurassic Park (1993))	(Independence Day (a.k.a. ID4) (1996))	0.41	0.706897
1	(Independence Day (a.k.a. ID4) (1996))	(Jurassic Park (1993))	0.41	0.706897
2	(Independence Day (a.k.a. ID4) (1996))	(Men in Black (a.k.a. MIB) (1997))	0.43	0.741379
3	(Men in Black (a.k.a. MIB) (1997))	(Independence Day (a.k.a. ID4) (1996))	0.43	0.693548
4	(Jurassic Park (1993))	(Men in Black (a.k.a. MIB) (1997))	0.41	0.706897
5	(Men in Black (a.k.a. MIB) (1997))	(Jurassic Park (1993))	0.41	0.661290

Figure 30: Trial 4 Results

	antecedents	consequents	support	confidence
0	(Fight Club (1999))	(Pulp Fiction (1994))	0.866667	1.0

Figure 31: Trial 5 Results

```
Top 10 topics:
Topic 10: king, stephen, superhero, comedy, arthur, alien, england, dark, atmospheric, ending
Topic 1: queue, netflix, available, boxing, vietnam, holocaust, police, judaism, biopic, aid
Topic 8: politics, president, terrorism, africa, racism, cia, thrilling, business, south, america
Topic 3: movie, business, animal, music, hollywood, eerie, animation, lion, bear, camel
Topic 7: war, world, ii, civil, cold, holocaust, journalism, gulf, anime, europe
Topic 9: christmas, york, new, halloween, homeless, bad, city, frontal, nudity, controversial
Topic 4: religion, bible, pregnancy, classic, court, priest, saint, preacher, convent, business
Topic 2: disney, race, nanny, fish, animation, arthur, heartwarming, original, music, samuel
Topic 5: school, high, funny, teacher, highschool, prom, subtext, lesbian, clever, fantasy
Topic 6: shakespeare, sort, space, cinematography, classic, updated, amazing, leonardo, dicaprio, nasa
```

Figure 32: Top 10 topics

```
544                    Bug's Life, A (1998)
1524       Guardians of the Galaxy 2 (2017)
666                      Toy Story 2 (1999)
1510                      Big Hero 6 (2014)
1497                  The Lego Movie (2014)
1457                   Avengers, The (2012)
1427                             Up (2009)
77                      Pulp Fiction (1994)
1                            Jumanji (1995)
2                   Grumpier Old Men (1995)
3        Father of the Bride Part II (1995)
4                            Sabrina (1995)
5            American President, The (1995)
6                              Nixon (1995)
7                             Casino (1995)
8               Sense and Sensibility (1995)
9                          Get Shorty (1995)
10                           Copycat (1995)
11                Leaving Las Vegas (1995)
12                           Othello (1995)
```

Figure 33: Similar Movies

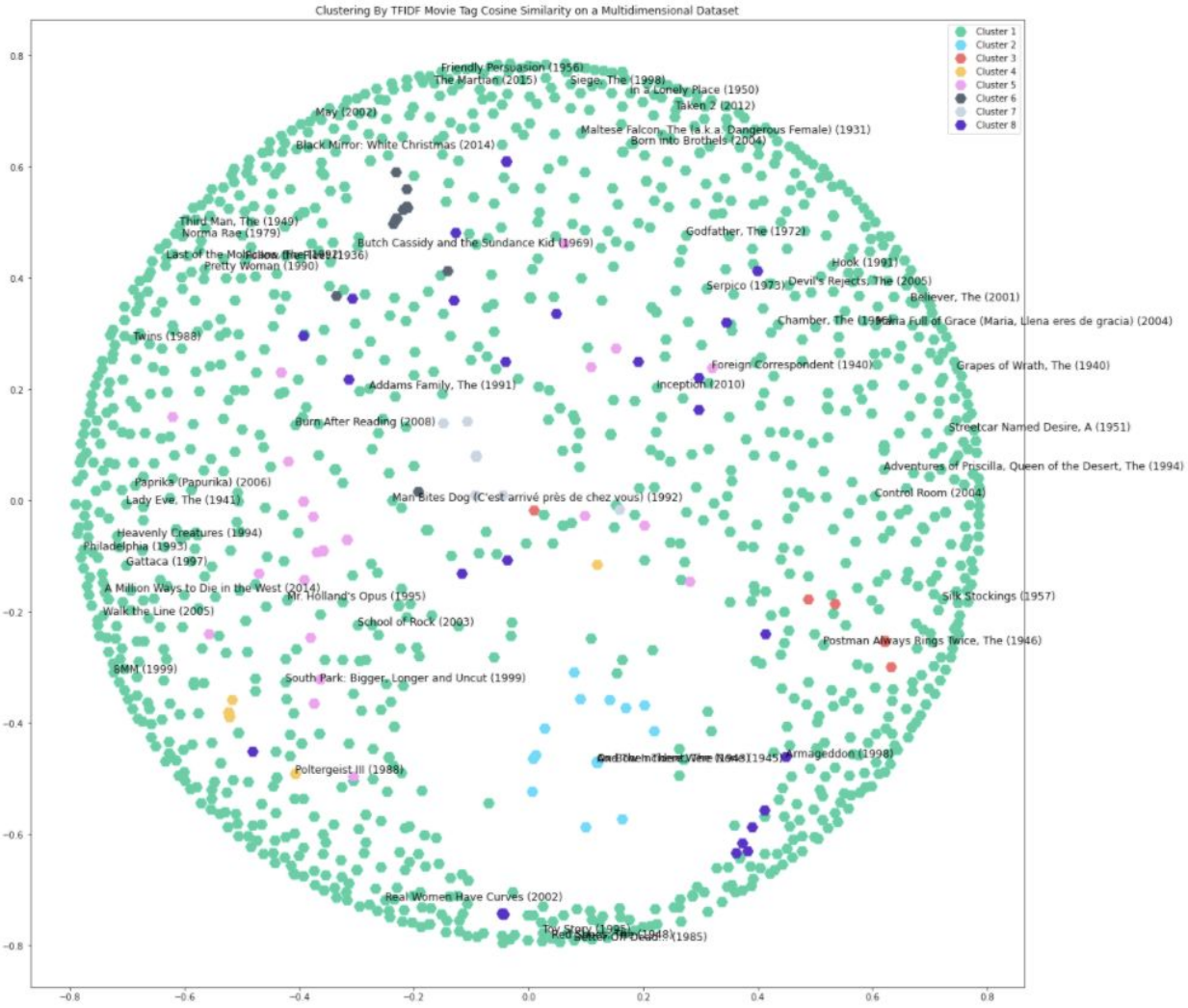


Figure 34: Clusters of Movies

Code

Unsupervised Learning

Association

Apriori Algorithm

Suppose we have transactions based on users' ratings for movies. A transaction of movies is considered as the set of movies that a user has highly rated. An association rule is a statement of the form A -> B, where A and B are itemsets.

- Support of A -> B = |AB|
- confidence of A -> B = |AB|/|A|

```
import pandas as pd
import numpy as np
import csv
from mlxtend.preprocessing import TransactionEncoder
from mlxtend.frequent_patterns import apriori, association_rules
```

```
movies_df = pd.read_csv('ml-latest-small/movies.csv')
movies_df.head()
```

	movieId	title	genres
0	1	Toy Story (1995)	Adventure|Animation|Children|Comedy|Fantasy
1	2	Jumanji (1995)	Adventure|Children|Fantasy
2	3	Grumpier Old Men (1995)	Comedy|Romance
3	4	Waiting to Exhale (1995)	Comedy|Drama|Romance
4	5	Father of the Bride Part II (1995)	Comedy

```
ratings_df = pd.read_csv('ml-latest-small/ratings.csv')
ratings_df.head()
```

	userId	movieId	rating	timestamp
0	1	1	4.0	964982703
1	1	3	4.0	964981247
2	1	6	4.0	964982224
3	1	47	5.0	964983815
4	1	50	5.0	964982931

```
print('There are', ratings_df['userId'].nunique(), 'users that have given', len(ratings_df), 'ratings consisting of', l
```

```
There are 610 users that have given 100836 ratings consisting of 9742 movies
```

```
# merge dataframe to show what users have rated movies
movie_ratings_df = pd.merge(ratings_df, movies_df, on='movieId')
movie_ratings_df.head()
```

	userId	movieId	rating	timestamp	title	genres
0	1	1	4.0	964982703	Toy Story (1995)	Adventure|Animation|Children|Comedy|Fantasy
1	5	1	4.0	847434962	Toy Story (1995)	Adventure|Animation|Children|Comedy|Fantasy
2	7	1	4.5	1106635946	Toy Story (1995)	Adventure|Animation|Children|Comedy|Fantasy
3	15	1	2.5	1510577970	Toy Story (1995)	Adventure|Animation|Children|Comedy|Fantasy
4	17	1	4.5	1305696483	Toy Story (1995)	Adventure|Animation|Children|Comedy|Fantasy

```
# Pivot the dataframe to show userId's on the left hand side of the matrix (key of each row) and movies to be
# the key of columns
movie_ratings_df = pd.merge(ratings_df, movies_df, on='movieId' )
movie_ratings_df.head()
movie_user_ratings_df = pd.pivot_table(movie_ratings_df, index='userId', columns='title', values='rating')
# once pivoted, axis: {0=userId row, 1=movieId column}

print('dataset dimensions: ', movie_user_ratings_df.shape)
print('Subset example:')
movie_user_ratings_df.iloc[:10, :40]
```

```
dataset dimensions:  (610, 9719)
Subset example:
```

title	'71 (2014)	'Hellboy': The Seeds of Creation (2004)	'Round Midnight (1986)	'Salem's Lot (2004)	'Til There Was You (1997)	'Tis the Season for Love (2015)	'burbs, The (1989)	'night Mother (1986)	(500) Days of Summer (2009)	*batteries not included (1987)	...	10th Victim, The (La decima vittima) (1965)	11'09"01 - September 11 (2002)	11:14 (2003)	11th Hour, The (2007)	12 Angry Men (1957)	12 Angry Men (1997)	12 Chairs (1971)	12 Chairs (1976)
userId																			
1	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	...	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
2	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	...	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
3	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	...	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
4	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	...	NaN	NaN	NaN	NaN	5.0	NaN	NaN	NaN

```
# Method to sort the matrix by users by applying a column with counts of non-Nan ratings that users made,
# sorting the matrix in descending order by this column, then removing this helper column
def sortByUsers(matrix_df, num_col):
    matrix_df = movie_user_ratings_df.append(matrix_df.count(axis=0), ignore_index=True)
    matrix_sorted_col_df = matrix_df.sort_values(len(matrix_df)-1, axis=1, ascending=False)
    matrix_sorted_col_df = matrix_sorted_col_df.drop(matrix_sorted_col_df.tail(1).index)
    return matrix_sorted_col_df.iloc[:, :num_col]
# Method to sort the matrix by movies by applying a row of total ratings non-Nan that each movie had gotten,
#  sorting the matrix in descending order by this row, then removing this helper row.
def sortByMovies(matrix_df, num_rows):
    matrix_df['total_ratings'] = pd.Series(matrix_df.count(axis=1))
    matrix_sorted_row_df = matrix_df.sort_values('total_ratings', ascending=False)
    matrix_sorted_by_movies = matrix_sorted_row_df.drop(['total_ratings'], axis=1)
    return matrix_sorted_by_movies.iloc[:num_rows, :]

# both methods only output the amount of rows or columns specified
```

```
# sort by users then by movies
userSortedMatrix = sortByUsers(movie_user_ratings_df, 100)
movieUserSortedMatrix = sortByMovies(userSortedMatrix, 100)
```

```
# Subset of of the first rows and columns that will have the least Nan values
movieUserSortedMatrix.head()
```

title	Forrest Gump (1994)	Shawshank Redemption, The (1994)	Pulp Fiction (1994)	Silence of the Lambs, The (1991)	Matrix, The (1999)	Star Wars: Episode IV - A New Hope (1977)	Jurassic Park (1993)	Braveheart (1995)	Terminator 2: Judgment Day (1991)	Schindler's List (1993)	...	Ocean's Eleven (2001)	Willy Wonka & the Chocolate Factory (1971)	Fifth Element, The (1997)	Batman Begins (2005)	Home Alone (1990)
413	5.0	5.0	5.0	4.0	5.0	5.0	4.0	5.0	5.0	4.0	...	4.0	4.0	5.0	4.5	3.0
67	3.5	3.0	2.0	3.5	4.5	5.0	3.5	2.5	3.5	4.0	...	3.5	0.5	3.0	2.5	1.0
479	5.0	5.0	4.0	4.5	5.0	4.5	5.0	5.0	4.5	5.0	...	3.5	3.5	3.5	4.0	4.0
273	4.5	4.5	5.0	4.0	4.0	3.0	3.5	4.5	4.5	4.0	...	4.0	3.5	4.0	3.5	3.5
598	3.5	4.0	5.0	3.0	5.0	5.0	4.0	3.5	4.5	NaN	...	4.0	3.0	4.0	3.0	3.0

5 rows × 100 columns

```
# All movie titles in the sorted matrix
movieIds = list(movieUserSortedMatrix.columns)
print(movieIds)
```

```
['Forrest Gump (1994)', 'Shawshank Redemption, The (1994)', 'Pulp Fiction (1994)', 'Silence of the Lambs, The (199
1)', 'Matrix, The (1999)', 'Star Wars: Episode IV - A New Hope (1977)', 'Jurassic Park (1993)', 'Braveheart (1995)',
'Terminator 2: Judgment Day (1991)', "Schindler's List (1993)", 'Fight Club (1999)', 'Toy Story (1995)', 'Star Wars:
Episode V - The Empire Strikes Back (1980)', 'Usual Suspects, The (1995)', 'American Beauty (1999)', 'Seven (a.k.a. S
e7en) (1995)', 'Independence Day (a.k.a. ID4) (1996)', 'Apollo 13 (1995)', 'Raiders of the Lost Ark (Indiana Jones an
d the Raiders of the Lost Ark) (1981)', 'Lord of the Rings: The Fellowship of the Ring, The (2001)', 'Star Wars: Epis
ode VI - Return of the Jedi (1983)', 'Godfather, The (1972)', 'Fugitive, The (1993)', 'Batman (1989)', 'Saving Privat
e Ryan (1998)', 'Lord of the Rings: The Two Towers, The (2002)', 'Lord of the Rings: The Return of the King, The (200
3)', 'Aladdin (1992)', 'Fargo (1996)', 'Sixth Sense, The (1999)', 'True Lies (1994)', 'Twelve Monkeys (a.k.a. 12 Monk
eys) (1995)', 'Lion King, The (1994)', 'Back to the Future (1985)', 'Speed (1994)', 'Shrek (2001)', 'Gladiator (200
0)', 'Men in Black (a.k.a. MIB) (1997)', 'Dances with Wolves (1990)', 'Mission: Impossible (1996)', 'Ace Ventura: Pet
Detective (1994)', 'Memento (2000)', 'Mask, The (1994)', 'Pirates of the Caribbean: The Curse of the Black Pearl (200
3)', 'Dark Knight, The (2008)', 'Alien (1979)', 'Beauty and the Beast (1991)', 'Die Hard (1988)', 'Mrs. Doubtfire (19
93)', 'Die Hard: With a Vengeance (1995)', 'Groundhog Day (1993)', 'Inception (2010)', 'Princess Bride, The (1987)',
'Finding Nemo (2003)', 'Good Will Hunting (1997)', 'Titanic (1997)', 'Stargate (1994)', 'Indiana Jones and the Last C
rusade (1989)', 'Star Wars: Episode I - The Phantom Menace (1999)', 'Batman Forever (1995)', 'Monty Python and the Ho
ly Grail (1975)', 'Pretty Woman (1990)', 'Léon: The Professional (a.k.a. The Professional) (Léon) (1994)', 'Dumb & Du
mber (Dumb and Dumber) (1994)', "One Flew Over the Cuckoo's Nest (1975)", 'X-Men (2000)', 'GoldenEye (1995)', 'Monste
rs, Inc. (2001)', 'Reservoir Dogs (1992)', 'Terminator, The (1984)', 'Kill Bill: Vol. 1 (2003)', 'Eternal Sunshine of
the Spotless Mind (2004)', 'American History X (1998)', 'Godfather: Part II, The (1974)', 'Babe (1995)', 'Goodfellas
(1990)', 'Aliens (1986)', 'Incredibles, The (2004)', 'Truman Show, The (1998)', 'Blade Runner (1982)', 'Twister (199
6)', 'Beautiful Mind, A (2001)', 'Spider-Man (2002)', 'E.T. the Extra-Terrestrial (1982)', 'Rock, The (1996)', 'Austi
n Powers: The Spy Who Shagged Me (1999)', 'Clockwork Orange, A (1971)', 'Ghostbusters (a.k.a. Ghost Busters) (1984)',
"Amelie (Fabuleux destin d'Amélie Poulain, Le) (2001)", 'Minority Report (2002)', "Ocean's Eleven (2001)", 'Willy Won
ka & the Chocolate Factory (1971)', 'Fifth Element, The (1997)', 'Batman Begins (2005)', 'Home Alone (1990)', 'Ghost
(1990)', 'Waterworld (1995)', 'Catch Me If You Can (2002)', 'Breakfast Club, The (1985)', 'Bourne Identity, The (200
2)']
```

```
# Specifying which movies we'll include in the transactions (this could change upon testing)
for movieId in movieIds:
    movieUserSortedMatrix.loc[movieUserSortedMatrix[movieId] == 0.5, movieId] = '0'
    movieUserSortedMatrix.loc[movieUserSortedMatrix[movieId] == 1, movieId] = '0'
    movieUserSortedMatrix.loc[movieUserSortedMatrix[movieId] == 1.5, movieId] = '0'
    movieUserSortedMatrix.loc[movieUserSortedMatrix[movieId] == 2, movieId] = '0'
    movieUserSortedMatrix.loc[movieUserSortedMatrix[movieId] == 2.5, movieId] = '0'
    movieUserSortedMatrix.loc[movieUserSortedMatrix[movieId] == 3, movieId] = '0'
    movieUserSortedMatrix.loc[movieUserSortedMatrix[movieId] == 3.5, movieId] = '0'
    movieUserSortedMatrix.loc[movieUserSortedMatrix[movieId] == 4, movieId] = '1'
    movieUserSortedMatrix.loc[movieUserSortedMatrix[movieId] == 4.5, movieId] = '1'
    movieUserSortedMatrix.loc[movieUserSortedMatrix[movieId] == 5, movieId] = '1'
```

```
movieUserSortedMatrix.head()
```

title	Forrest Gump (1994)	Shawshank Redemption, The (1994)	Pulp Fiction (1994)	Silence of the Lambs, The (1991)	Matrix, The (1999)	Star Wars: Episode IV - A New Hope (1977)	Jurassic Park (1993)	Braveheart (1995)	Terminator 2: Judgment Day (1991)	Schindler's List (1993)	...	Ocean's Eleven (2001)	Willy Wonka & the Chocolate Factory (1971)	Fifth Element, The (1997)	Batman Begins (2005)	Home Alone (1990)
413	0	0	0	1	0	0	1	0	0	1	...	1	1	0	0	1
67	1	1	0	1	0	0	1	1	1	1	...	1	0	1	1	0
479	0	0	1	0	0	0	0	0	0	0	...	1	1	1	1	1
273	0	0	0	1	1	1	1	0	0	1	...	1	1	1	1	1
598	1	1	0	1	0	0	1	1	0	NaN	...	1	1	1	1	1

5 rows × 100 columns

```
# Turn dataframe into a 2D array with 0's and 1's and existing Nan values
unclean_apriori_data = movieUserSortedMatrix.to_numpy()
print(unclean_apriori_data)
```

```
[['0' '0' '0' ... '1' '0' '1']
 ['1' '1' '0' ... '1' '1' '0']
 ['0' '0' '1' ... '0' '1' '0']
 ...
 ['1' '0' nan ... '0' '0' nan]
 ['1' '1' '1' ... nan nan '1']
 ['1' '0' '0' ... nan '1' nan]]
```

```
data = []
# Iterate through 2D array and create a list of movie titles for each row that had movies with a 1 value.
# Each list will correspond as a transaction in the dataset
for row in unclean_apriori_data:
    movies_in_row = []
    for i in range(len(row)):
        if row[i] == '1':
            movies_in_row.append(movieIds[i])
    if len(movies_in_row) >= 2:
        data.append(movies_in_row)
```

```
# All transactions
print(data[:1])
```

```
[['Silence of the Lambs, The (1991)', 'Jurassic Park (1993)', "Schindler's List (1993)", 'Toy Story (1995)', 'Seven
(a.k.a. Se7en) (1995)', 'Independence Day (a.k.a. ID4) (1996)', 'Apollo 13 (1995)', 'Batman (1989)', 'Lord of the Rin
gs: The Return of the King, The (2003)', 'Aladdin (1992)', 'Sixth Sense, The (1999)', 'True Lies (1994)', 'Lion King,
The (1994)', 'Speed (1994)', 'Shrek (2001)', 'Men in Black (a.k.a. MIB) (1997)', 'Dances with Wolves (1990)', 'Missio
n: Impossible (1996)', 'Mask, The (1994)', 'Dark Knight, The (2008)', 'Beauty and the Beast (1991)', 'Mrs. Doubtfire
(1993)', 'Die Hard: With a Vengeance (1995)', 'Groundhog Day (1993)', 'Titanic (1997)', 'Stargate (1994)', 'Indiana J
ones and the Last Crusade (1989)', 'Star Wars: Episode I - The Phantom Menace (1999)', 'Pretty Woman (1990)', 'Dumb &
Dumber (Dumb and Dumber) (1994)', "One Flew Over the Cuckoo's Nest (1975)", 'X-Men (2000)', 'GoldenEye (1995)', 'Mons
ters, Inc. (2001)', 'Kill Bill: Vol. 1 (2003)', 'Incredibles, The (2004)', 'Truman Show, The (1998)', 'Twister (199
6)', 'Beautiful Mind, A (2001)', 'E.T. the Extra-Terrestrial (1982)', 'Rock, The (1996)', 'Austin Powers: The Spy Who
Shagged Me (1999)', "Ocean's Eleven (2001)", 'Willy Wonka & the Chocolate Factory (1971)', 'Home Alone (1990)', 'Ghos
t (1990)', 'Catch Me If You Can (2002)', 'Bourne Identity, The (2002)']]
```

Transform your data into apriori algorithm

```
# Use the transaction encoder to turn values into True or False
oht = TransactionEncoder()
oht_array = oht.fit(data).transform(data)
favourite_movies_df = pd.DataFrame(oht_array, columns = oht.columns_)
favourite_movies_df.head()
```

	Ace Ventura: Pet Detective (1994)	Aladdin (1992)	Alien (1979)	Aliens (1986)	Amelie (Fabuleux destin d'Amélie Poulain, Le) (2001)	American Beauty (1999)	American History X (1998)	Apollo 13 (1995)	Austin Powers: The Spy Who Shagged Me (1999)	Babe (1995)	...	Titanic (1997)	Toy Story (1995)	True Lies (1994)	Truman Show, The (1998)	Twelve Monkeys (a.k.a. 12 Monkeys) (1995)	Twister (1996)	Usu Suspect Th (199
0	False	True	False	False	False	False	False	True	True	False	...	True	True	True	True	False	True	Fals
1	True	True	False	True	False	False	True	True	True	False	...	True	True	True	False	False	True	Tru
2	True	True	False	True	False	True	False	True	False	True	...	True	True	True	False	True	True	Tru
3	False	True	True	True	False	False	False	False	True	True	...	True	True	True	False	True	True	Tru
4	True	True	False	False	False	False	True	True	True	False	...	True	True	True	True	True	True	Tru

5 rows × 100 columns

```
# Apply the apriori algoithm from mlxtend (shown in tutorial) to generate frequent itemsets
from mlxtend.frequent_patterns import apriori

# The min_support can change with testing
frequent_itemsets = apriori(favourite_movies_df, min_support=0.4, use_colnames=True)
pd.set_option('max_colwidth', 1500)
pd.set_option("max_rows", None)
frequent_itemsets.head(10)
```

	support	itemsets
0	0.42	(Ace Ventura: Pet Detective (1994))
1	0.49	(Aladdin (1992))
2	0.49	(Alien (1979))
3	0.43	(Apollo 13 (1995))
4	0.51	(Austin Powers: The Spy Who Shagged Me (1999))
5	0.47	(Back to the Future (1985))
6	0.56	(Batman (1989))
7	0.40	(Batman Begins (2005))
8	0.44	(Beautiful Mind, A (2001))
9	0.40	(Beauty and the Beast (1991))

```
# Extract association rules by specifying min_threshold (that can change with testing)
rules = association_rules(frequent_itemsets, metric='confidence', min_threshold=0.7)
rule_results = rules[['antecedents', 'consequents', 'support', 'confidence']]
rule_results.head(100)
```

	antecedents	consequents	support	confidence
0	(Jurassic Park (1993))	(Independence Day (a.k.a. ID4) (1996))	0.47	0.770492
1	(Independence Day (a.k.a. ID4) (1996))	(Jurassic Park (1993))	0.47	0.723077
2	(Independence Day (a.k.a. ID4) (1996))	(Men in Black (a.k.a. MIB) (1997))	0.50	0.769231
3	(Men in Black (a.k.a. MIB) (1997))	(Independence Day (a.k.a. ID4) (1996))	0.50	0.769231
4	(Mission: Impossible (1996))	(Independence Day (a.k.a. ID4) (1996))	0.45	0.762712
5	(Speed (1994))	(Independence Day (a.k.a. ID4) (1996))	0.40	0.769231
6	(Star Wars: Episode I - The Phantom Menace (1999))	(Independence Day (a.k.a. ID4) (1996))	0.41	0.732143
7	(True Lies (1994))	(Independence Day (a.k.a. ID4) (1996))	0.40	0.833333
8	(Jurassic Park (1993))	(Men in Black (a.k.a. MIB) (1997))	0.45	0.737705

TFIDF Text Mining And Clustering

By taking the tag keywords for each movie, we will find the TFIDF matrices and find the cosine similarity between movies by using the mined text.

We will generate the top numbers of similar movies based on the movie input

We will to cluster the movies by cosine similarities and Kmeans

We will decide on K using the elbow method

```
import numpy as np
import pandas as pd
from sklearn import decomposition
from scipy import linalg
from sklearn.feature_extraction import stop_words
from sklearn.feature_extraction.text import TfidfVectorizer, CountVectorizer
from sklearn.metrics.pairwise import cosine_similarity
import nltk
```

```
movies_df = pd.read_csv('ml-latest-small/movies.csv')
movies_df = movies_df.drop('genres', 1)
movies_df.head(5)
```

	movieId	title
0	1	Toy Story (1995)
1	2	Jumanji (1995)
2	3	Grumpier Old Men (1995)
3	4	Waiting to Exhale (1995)
4	5	Father of the Bride Part II (1995)

```
tags_df = pd.read_csv('ml-latest-small/tags.csv')
tags_df = tags_df.drop('timestamp', 1)
tags_df = tags_df.drop('userId', 1)
tags_df.head(5)
```

	movieId	tag
0	60756	funny
1	60756	Highly quotable
2	60756	will ferrell
3	89774	Boxing story
4	89774	MMA

```
# Take only relevant data with movieId, title and tags
movie_tags_df = pd.merge(movies_df, tags_df, on='movieId')
movie_tags_df = movie_tags_df.drop_duplicates()
movie_tags_df.head(5)
```

	movieId	title	tag
0	1	Toy Story (1995)	pixar
2	1	Toy Story (1995)	fun
3	2	Jumanji (1995)	fantasy
4	2	Jumanji (1995)	magic board game
5	2	Jumanji (1995)	Robin Williams

```
# using NLTK from tutorial for cleaning text
# TAKEN FROM TUTORIAL 7
nltk.download('wordnet')
nltk.download('stopwords')
stopwords = nltk.corpus.stopwords.words('english')

import re, string

def clean_text(text):
#     remove text that match reject with numbers and symbols
  text = ''.join([word for word in text if not re.match(r'^-?\d+(?:\.\d+)?$', word)])
  text = ' '.join([word for word in text.split() if word not in stopwords])
  text = ''.join([word for word in text if word not in string.punctuation])
  text = text.lower()
  return text
```

```
[nltk_data] Downloading package wordnet to /Users/rosakng/nltk_data...
[nltk_data]   Package wordnet is already up-to-date!
[nltk_data] Downloading package stopwords to
[nltk_data]     /Users/rosakng/nltk_data...
[nltk_data]   Package stopwords is already up-to-date!
```

```
# This should not return words with numbers
clean_text('1900s stress Hungry')
```

```
'stress hungry'
```

```
# This may take a minute to load
# Group by movieId to consolidate all tags for a movie in a row
movie_tags_df = movie_tags_df.groupby(['movieId']).tag.apply(list).reset_index()
movie_tags_df['tag'] = movie_tags_df['tag'].str.join(' ')
movie_tags_df['title'] = movie_tags_df.apply(lambda row: get_movie_title(row.movieId), axis=1)
movie_tags_df.head()
```

	movieId	tag	title
0	1	pixar fun	Toy Story (1995)
1	2	fantasy magic board game Robin Williams game	Jumanji (1995)
2	3	moldy old	Grumpier Old Men (1995)
3	5	pregnancy remake	Father of the Bride Part II (1995)
4	7	remake	Sabrina (1995)

```
# Use tutorial's lemmatizing function to clean text
# TAKEN FROM TUTORIAL 7
wn = nltk.WordNetLemmatizer()
def lemmatizing(text):
  text = [wn.lemmatize(word) for word in text.split()]
  return str(text)
tag_text_df = movie_tags_df.copy()
tag_text_df = tag_text_df.drop('title', 1)
tag_text_df['tag1'] = tag_text_df['tag'].apply(lambda x: clean_text(str(x.strip())))
tag_text_df['tag2'] = tag_text_df['tag1'].apply(lambda x: lemmatizing(x))
tag_text_df.head()
```

	movieId	tag	tag1	tag2
0	1	pixar fun	pixar fun	['pixar', 'fun']
1	2	fantasy magic board game Robin Williams game	fantasy magic board game robin williams game	['fantasy', 'magic', 'board', 'game', 'robin',...
2	3	moldy old	moldy old	['moldy', 'old']
3	5	pregnancy remake	pregnancy remake	['pregnancy', 'remake']
4	7	remake	remake	['remake']

```
# Apply vectorizer to tag text
vectorizer_tfidf = TfidfVectorizer(stop_words='english')
vectors_tfidf = vectorizer_tfidf.fit_transform(tag_text_df['tag2'])
```

```
# TAKEN FROM TUTORIAL 7
# Apply topic modelling to generate term frequencies for 10 groups
import collections
clf = decomposition.NMF(n_components=10,random_state=1)
W1 = clf.fit_transform(vectors_tfidf)
H1 = clf.components_
vocab = np.array(vectorizer_tfidf.get_feature_names())
topics_frequency = collections.Counter(np.array(W1.argsort()[:,-1]))
topics_frequency = collections.OrderedDict(sorted(topics_frequency.items(),key= lambda kv: kv[1],reverse=True))
total_comments_count = sum(topics_frequency.values())

print('Top', 10, 'topics:')
for k,v in topics_frequency.items():
  print('Topic {}: {}'.format(k+1, ", ".join([str(x) for x in vocab[(-H1[k]).argsort(axis=-1)[:10]]])))
```

```
Top 10 topics:
Topic 10: king, stephen, superhero, comedy, arthur, alien, england, dark, atmospheric, ending
Topic 1: queue, netflix, available, boxing, vietnam, holocaust, police, judaism, biopic, aid
Topic 8: politics, president, terrorism, africa, racism, cia, thrilling, business, south, america
Topic 3: movie, business, animal, music, hollywood, eerie, animation, lion, bear, camel
Topic 7: war, world, ii, civil, cold, holocaust, journalism, gulf, anime, europe
Topic 9: christmas, york, new, halloween, homeless, bad, city, frontal, nudity, controversial
Topic 4: religion, bible, pregnancy, classic, court, priest, saint, preacher, convent, business
Topic 2: disney, race, nanny, fish, animation, arthur, heartwarming, original, music, samuel
Topic 5: school, high, funny, teacher, highschool, prom, subtext, lesbian, clever, fantasy
Topic 6: shakespeare, sort, space, cinematography, classic, updated, amazing, leonardo, dicaprio, nasa
```

Calculating Similarity

```
# Use cosine similarity function with the document term vectors to generate similarity scores
cos_similarity = cosine_similarity(vectors_tfidf)
```

```
# Creates a frame with movieIds and new indexes starting from 0 (Used for finding groups of similarity with a movieId)
movie_index_df = pd.Series(movie_tags_df.index, index=movie_tags_df['movieId']).drop_duplicates()
movie_index_df.head(20)
```

```
movieId
1       0
2       1
3       2
5       3
7       4
11      5
14      6
16      7
17      8
21      9
22     10
25     11
26     12
28     13
29     14
31     15
32     16
34     17
36     18
38     19
dtype: int64
```

```python
def get_similar_movies(movieId, num_results, cos_similarity=cos_similarity):
    # Movie index of movieId from all_movie_indices
    movie_index = movie_index_df[movieId]
    # Similarity scores of movies corresponding to movieId
    similarity_scores = list(enumerate(cos_similarity[movie_index]))
    # Sort movies based on the similarity scores in descending order
    sorted_similarity_scores = sorted(similarity_scores, key=lambda x: x[1], reverse=True)[1:num_results+1]
    # Get the top movie data corresponding to subset of scores
    similar_indexes = [i[0] for i in sorted_similarity_scores]
    # Return most similar movies
    return movie_tags_df['title'].iloc[similar_indexes]
```

```python
get_similar_movies(1, 20)
```

```
544                    Bug's Life, A (1998)
1524      Guardians of the Galaxy 2 (2017)
666                      Toy Story 2 (1999)
1510                       Big Hero 6 (2014)
1497                  The Lego Movie (2014)
1457                   Avengers, The (2012)
1427                                Up (2009)
77                       Pulp Fiction (1994)
1                             Jumanji (1995)
2                   Grumpier Old Men (1995)
3       Father of the Bride Part II (1995)
4                             Sabrina (1995)
5           American President, The (1995)
6                               Nixon (1995)
7                              Casino (1995)
8             Sense and Sensibility (1995)
9                          Get Shorty (1995)
10                            Copycat (1995)
11                Leaving Las Vegas (1995)
12                            Othello (1995)
Name: title, dtype: object
```

```python
print(cos_similarity)
```

```
[[1. 0. 0. ... 0. 0. 0.]
 [0. 1. 0. ... 0. 0. 0.]
 [0. 0. 1. ... 0. 0. 0.]
 ...
 [0. 0. 0. ... 1. 0. 0.]
 [0. 0. 0. ... 0. 1. 0.]
 [0. 0. 0. ... 0. 0. 1.]]
```

Clustering

```
import matplotlib.pyplot as plt

import sklearn
from sklearn.cluster import KMeans
import sklearn.metrics as sm
from sklearn import datasets
from sklearn.metrics import confusion_matrix, classification_report
```

```
%matplotlib inline
```

```
# Apply KMeans with the tfidf vectors to create K=8 clusters
clustering  = KMeans(n_clusters=8)
clustering.fit(vectors_tfidf)
clusters = clustering.labels_.tolist()
```

```
titles = movie_tags_df.title.to_numpy()
# Create a new dataframe of cluster values and corresponding titles
cluster_frame = pd.DataFrame({'title': titles, 'cluster': clusters}, index=[clusters], columns = ['title', 'cluster'])
cluster_frame.head()
```

	title	cluster
0	Toy Story (1995)	0
0	Jumanji (1995)	0
0	Grumpier Old Men (1995)	0
0	Father of the Bride Part II (1995)	0
0	Sabrina (1995)	0

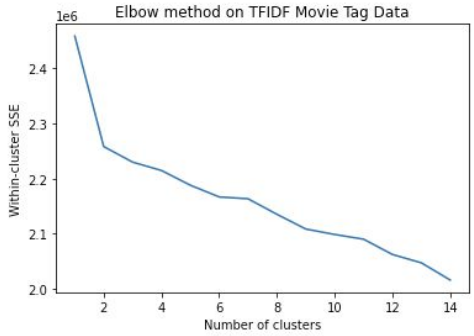
```
# Number of movies per cluster
print(cluster_frame['cluster'].value_counts())
```

```
0    1344
1     120
7      27
3      21
4      19
6      15
5      14
2      12
Name: cluster, dtype: int64
```

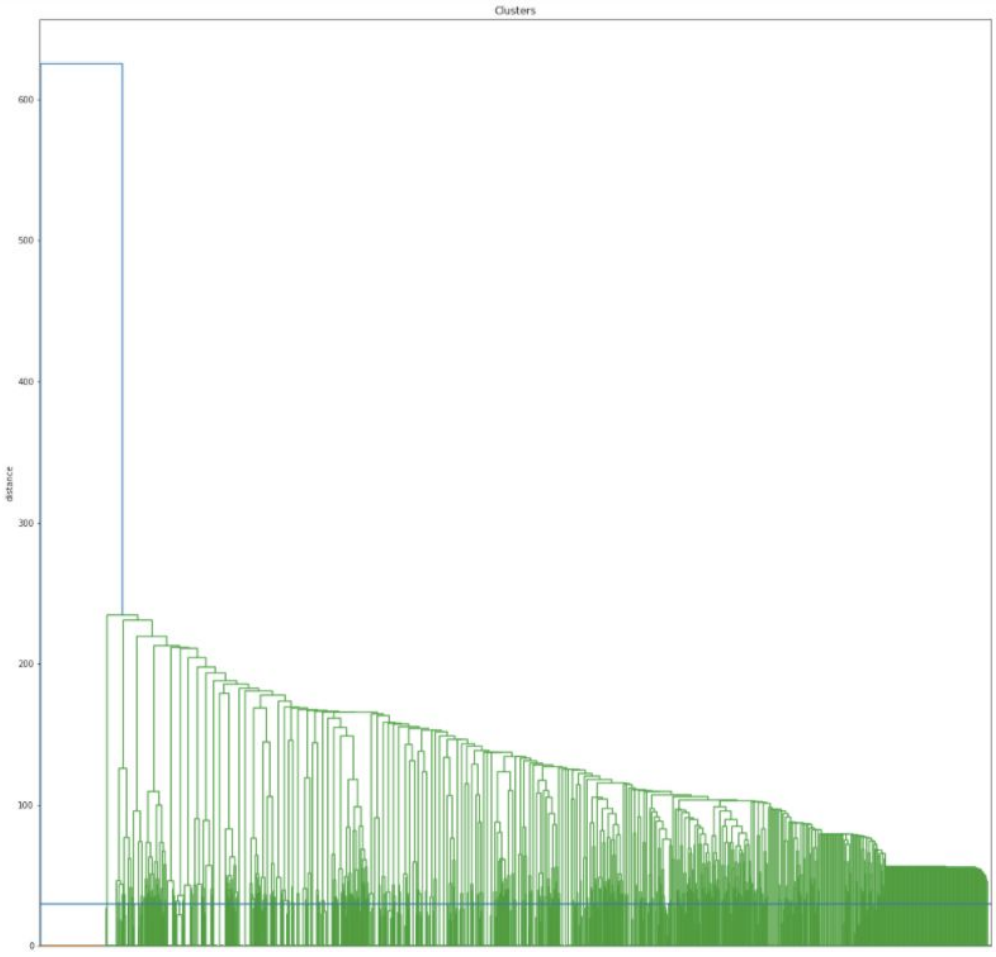
```
# This cell may take a minute to load
import matplotlib as mpl

from sklearn.manifold import MDS

# "MDS is a technique used for analyzing similarity or dissimilarity data. It attempts to model similarity or dissimila
# - MDS is a common way to visualize common groups of text using TFIDF vectorization
# https://scikit-learn.org/stable/modules/manifold.html

# Set dissimilarity as "precomputed" because we are inputting a pre-computed distance matrix
# Set components = 2 for 2D space
mds = MDS(n_components=2, dissimilarity="precomputed")

# The cosine distances of each plot is 1-cosine similarity between the plots
# Since we want to see the distances representative of each plot rather than similarity relative to each other, we plot
cosine_distances = 1-cos_similarity
X_2D_values = mds.fit_transform(cosine_distances)
# Setting all target values for X as 0 and y as 1 (See cosine similarity data which contain 0's and 1's)
x, y = X_2D_values[:, 0], X_2D_values[:, 1]
```

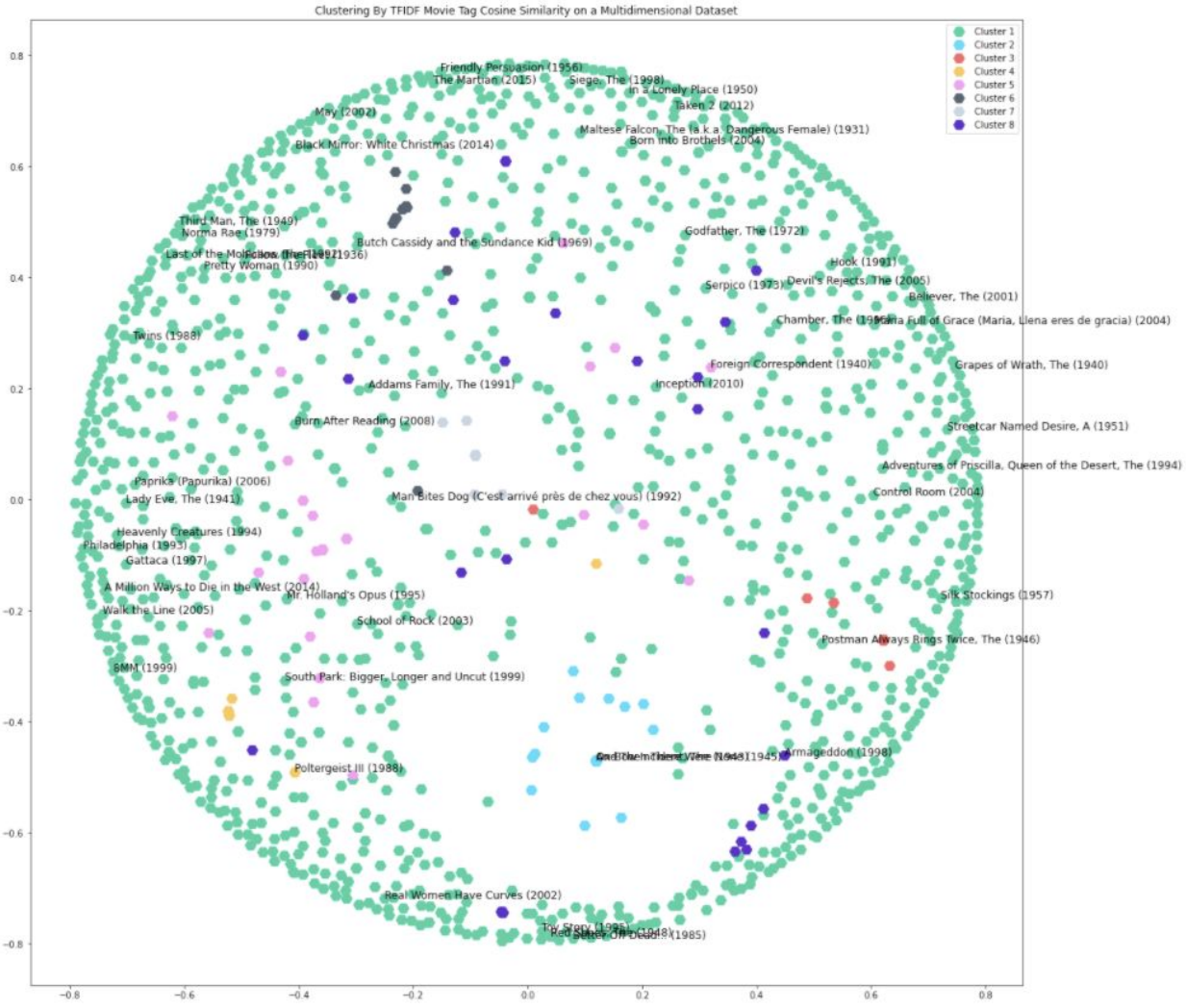
```
cluster_colors = {0: '#1ddla1', 1: '#48dbfb', 2: '#ff6b6b', 3: '#feca57', 4: '#ff9ff3', 5: '#576574', 6: '#c8d6e5', 7:
cluster_names = {0: 'Cluster 1', 1: 'Cluster 2', 2: 'Cluster 3', 3: 'Cluster 4', 4: 'Cluster 5', 5: 'Cluster 6', 6: 'Cl

graph_df = pd.DataFrame(dict(x_axis=x, y_axis=y, label=clusters, movie_title=titles))

#Create groups for each cluster
groups = graph_df.groupby('label')

# Create a figure an plot for both axes and set sizing
figure, axes = plt.subplots(figsize=(20, 20))

for name, group in groups:
    axes.plot(
        group.x_axis, group.y_axis, marker='H', linestyle='', ms=12, label=cluster_names[name], color=cluster_colors[na
    )

axes.set_title('Clustering By TFIDF Movie Tag Cosine Similarity on a Multidimensional Dataset')

# Legend
axes.legend()

#Dispersed Film title labels for every 50 movies
i = 0
while i <= 1572:
    axes.text(graph_df.iloc[i]['x_axis'], graph_df.iloc[i]['y_axis'], graph_df.iloc[i]['movie_title'], size=12)
    i += 30

plt.show()
```

```
# TAKEN FROM TUTORIAL
# Use elbow method to find the fitting number of clusters
from sklearn.preprocessing import scale

X = scale(cos_similarity)

#Elbow method
distortations = {}
for k in range(1,15):
  kmeans = KMeans(n_clusters=k)
  kmeans.fit(X)
  distortations[k] = kmeans.inertia_

plt.plot(list(distortations.keys()),list(distortations.values()))
plt.title('Elbow method on TFIDF Movie Tag Data')
plt.xlabel('Number of clusters')
plt.ylabel('Within-cluster SSE')
plt.show()
```




```
# TAKEN FROM TUTORIAL: show dendrogram of results
from scipy.cluster.hierarchy import dendrogram, linkage

# This dendrogram is shown for the purpose of seeing the dispersion of clusters
Z = linkage(scale(cosine_distances),'ward')

plt.figure(figsize=(20,20))
plt.title('Clusters')
plt.ylabel('distance')
# Removing labels along the x-axis due to all movies overlapping
plt.tick_params(axis='x', bottom='off', top='off', labelbottom='off')

dendrogram(Z)

plt.axhline(y=30)
plt.show()
```


Clusters
distance
600
500
400
300
200
100
0



Supervised Learning

jupyter TMDB TV Show Supervised Learning Last Checkpoint: a few seconds ago (autosaved)

Logout

File Edit View Insert Cell Kernel Widgets Help

Trusted | Python 3

Run Markdown

Data prep to make multiple categorical variables useable

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


# Math
import math

# Plot imports
import matplotlib.pyplot as plt
import seaborn as sns
sns.set_style('whitegrid')
%matplotlib inline

# Machine Learning Imports
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import train_test_split

# For evaluating our ML results
from sklearn import metrics
```

In [2]:
```
df = pd.read_csv('tmdb_tv_dataset_small.csv', converters={'genres': eval, 'episode_run_time': eval})
df.head()
```

Out[2]:

	id	popularity	vote_count	name	first_air_date	backdrop_path	overview	genre_ids	original_name	original_language	...	numb
0	76773	238.249	550	Station 19	2018-03-22	/PB1agKtni7zo4fea8yk6y4iE5L.jpg	A group of heroic firefighters at Seattle Fire...	[10759, 18]	Station 19	en	...	
1	71790	230.154	303	S.W.A.T.	2017-11-02	/gJSqr2prvTegFbL0cEWK9Qti3Vs.jpg	A locally born and bred S.W.A.T. lieutenant is...	[10759, 80, 18]	S.W.A.T.	en	...	
2	65494	253.691	647	The Crown	2016-11-04	/4InrdamBEM31unNiuEHGYTPX1e2.jpg	The gripping, decades-spanning inside story of...	[18]	The Crown	en	...	
							After a					

In [3]:
```
df.isnull().sum()
```

Out[3]:
```
id                        0
popularity                0
vote_count                0
name                      0
first_air_date            0
backdrop_path            94
overview                 54
genre_ids                 0
original_name             0
original_language         0
vote_average              0
poster_path              24
origin_country_x          0
created_by                0
episode_run_time          0
genres                    0
homepage                365
in_production             0
languages                 0
last_air_date             3
last_episode_to_air       3
next_episode_to_air    1451
networks                  0
number_of_episodes        0
number_of_seasons         0
origin_country_y          0
production_companies      0
production_countries      0
seasons                   0
spoken_languages          0
status                    0
tagline                1239
type                      0
dtype: int64
```

In [4]:
```
# drop rows with empty episode run times
df = df[df['episode_run_time'].map(lambda d: len(d)) > 0]
```

In [5]:
```
df.shape
```

Out[5]:
```
(1507, 33)
```

"The TV series data recorded on IMDb websites were extracted and provided by Andrej Krevl from the Stanford SNAP Research Group. We filtered and cleaned the data by excluding the TV series with very small numbers of ratings, say less than 10 viewer ratings, and those with missing feature information for the accuracy of prediction. Features of the data set include the genre (vector of Booleans), release year (numeric), number of seasons (numeric), number of episodes for each season (numeric), number of ratings (numeric), number of critics (numeric), number of reviewers (numeric), runtime (numeric), aspect ratio

In [6]:
```
# create a data frame with the above features

# FEATURES
# genres (is an array of objects, [{id, name}]. first convert to array of strings, then one hot encode)
genres_col = df['genres'].copy()

# release year (is a string)
first_air_date_col = df['first_air_date'].copy()

# number of seasons
number_of_seasons_col = df['number_of_seasons'].copy()

# number of episodes
number_of_episodes_col = df['number_of_episodes'].copy()

# number of ratings
vote_count_col = df['vote_count'].copy()

# number of critics: not available

# number of reviewers: not available

# runtime (is an array[int])
episode_run_time_col = df['episode_run_time'].copy()

# aspect ratio: not available

# color: not available

# TARGET
# average rating
vote_average_col = df['vote_average'].copy()
```

Now, we have to further transform the genres and episode_run_time columns to make them useable in logistic regression

In [7]:
```
type(genres_col)
```

Out[7]:
```
pandas.core.series.Series
```

In [8]:
```
type(episode_run_time_col)
```

Out[8]:
```
pandas.core.series.Series
```

In [9]:
```
genres_col.head()
```

Out[9]:
```
0    [{'id': 10759, 'name': 'Action & Adventure'}, ...
1    [{'id': 10759, 'name': 'Action & Adventure'}, ...
2                       [{'id': 18, 'name': 'Drama'}]
```

In [10]:
```
episode_run_time_col.head()
```

Out[10]:
```
0    [43]
1    [42]
2    [52]
3    [44]
4    [60]
Name: episode_run_time, dtype: object
```

In [11]:
```
# transform all entries in genres_df from [{id, name}] to [name]
genre_strings_col = genres_col.apply(lambda x: [genre['name'] for genre in x])
```

In [12]:
```
genre_strings_col[0]
```

Out[12]:
```
['Action & Adventure', 'Drama']
```

In [13]:
```
import statistics
```

In [14]:
```
episode_run_time_col.isnull().sum()
```

Out[14]:
```
0
```

In [15]:
```
# get the mean episode runtimes to represent runtime
mean_runtime_col = episode_run_time_col.apply(lambda x: statistics.mean(x))
```

In [16]:
```
mean_runtime_col.head()
```

Out[16]:
```
0    43.0
1    42.0
2    52.0
3    44.0
4    60.0
Name: episode_run_time, dtype: float64
```

In [17]:
```
first_air_date_col.head()
```

Out[17]:
```
0    2018-03-22
1    2017-11-02
2    2016-11-04
3    2014-10-07
4    2012-10-10
Name: first_air_date, dtype: object
```

In [18]:
```
# convert string start dates to be datetime values,
# and then further transform to ordinal values so that classifier can properly interpret the meaning of dates
first_air_datetime_ordinal_col = first_air_date_col.apply(lambda x: pd.to_datetime(x).toordinal())
```

In [19]:
```
first_air_datetime_ordinal_col.head()
```

Out[19]:
```
0    736775
1    736635
2    736272
3    735513
4    734786
Name: first_air_date, dtype: int64
```

In [20]:
```
# one-hot encode the genres and finish initial data preprocessing
from sklearn.preprocessing import MultiLabelBinarizer
mlb = MultiLabelBinarizer()
```

In order to properly one-hot encode our genres column, we need to use the MultiLabelBinarizer from sklearn. This handles nested categorical data, which is what our genres column is (a series of list of string).

https://stackoverflow.com/questions/45312377/how-to-one-hot-encode-from-a-pandas-column-containing-a-list

We start by forming the final starting dataframe

In [21]:
```
start_df = pd.DataFrame(
        {
            'first_air_datetime_ordinal': first_air_datetime_ordinal_col,
            'number_of_seasons': number_of_seasons_col,
            'number_of_episodes': number_of_episodes_col,
            'vote_count': vote_count_col,
            'mean_runtime': mean_runtime_col,
            'vote_average' : vote_average_col
        })
```

Now, we one-hot encode the genres and add it to our dataframe

In [22]:
```
start_df = start_df.join(
    pd.DataFrame(
        mlb.fit_transform(genre_strings_col),
        columns=mlb.classes_,
        index=df.index))
```

In [23]:
```
start_df.columns
```

Out[23]:
```
Index(['first_air_datetime_ordinal', 'number_of_seasons', 'number_of_episodes',
       'vote_count', 'mean_runtime', 'vote_average', 'Action',
```

In [23]:
```
start_df.columns
```

Out[23]:
```
Index(['first_air_datetime_ordinal', 'number_of_seasons', 'number_of_episodes',
       'vote_count', 'mean_runtime', 'vote_average', 'Action',
       'Action & Adventure', 'Adventure', 'Animation', 'Comedy', 'Crime',
       'Documentary', 'Drama', 'Family', 'Fantasy', 'History', 'Horror',
       'Kids', 'Music', 'Mystery', 'News', 'Reality', 'Romance',
       'Sci-Fi & Fantasy', 'Science Fiction', 'Soap', 'Talk', 'War',
       'War & Politics', 'Western'],
      dtype='object')
```

In [24]:
```
start_df.head()
```

Out[24]:

	first_air_datetime_ordinal	number_of_seasons	number_of_episodes	vote_count	mean_runtime	vote_average	Action	Action & Adventure	Adventure	Animation	...	N
0	736775	4	46	550	43.0	8.2	0	1	0	0	...	
1	736635	4	71	303	42.0	7.6	0	1	0	0	...	
2	736272	4	40	647	52.0	8.2	0	0	0	0	...	
3	735513	7	134	6222	44.0	7.6	0	0	0	0	...	
4	734786	9	182	965	60.0	8.3	0	0	0	0	...	

5 rows × 31 columns

In [25]:
```
start_df.isnull().sum()
```

Out[25]:
```
first_air_datetime_ordinal    0
number_of_seasons             0
number_of_episodes            0
vote_count                    0
mean_runtime                  0
vote_average                  0
Action                        0
Action & Adventure            0
Adventure                     0
Animation                     0
Comedy                        0
Crime                         0
Documentary                   0
Drama                         0
Family                        0
Fantasy                       0
History                       0
Horror                        0
Kids                          0
Music                         0
```

Create a categorical column to represent overall popularity

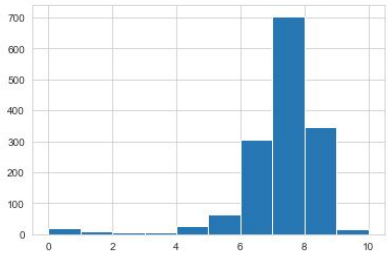
The final step to take is to convert our current target, vote_average, into a categorical class variable. The strategy used by the stanford students was to segment the vote_average data into multiple buckets (fair for below 8, good for 8-9, and Very popular for above 9).

We will explore our data further to determine possible segmentation strategies

In [26]:
```
import matplotlib.pyplot as plt
```

In [27]:
```
# Histogram of vote_average (current target we are looking at)
plt.hist(start_df['vote_average'])
```

Out[27]:
```
(array([ 21.,  11.,   7.,   7.,  25.,  63., 306., 704., 347.,  16.]),
 array([ 0.,  1.,  2.,  3.,  4.,  5.,  6.,  7.,  8.,  9., 10.]),
 <a list of 10 Patch objects>)
```



In [28]:
```
start_df['vote_average'].describe()
```

Out[28]:
```
count    1507.000000
mean        7.162973
std         1.391311
min         0.000000
25%         6.800000
50%         7.400000
75%         7.900000
max        10.000000
Name: vote_average, dtype: float64
```

From the histogram above, we can see that 7-8 is a distinct category, and 7.16 is the mean of the column.

We can also see that two other distinct categories exist below 7 and above 8. It seems possible to further segment the data, however the largest groupings of

In [29]:
```
rating_category = pd.cut(start_df['vote_average'], [0, 6, 8, 10], labels=['below average', 'average', 'above average'],
```

In [30]:
```
start_df['rating_category'] = rating_category
```

In [31]:
```
start_df.head()
```

Out[31]:

	first_air_datetime_ordinal	number_of_seasons	number_of_episodes	vote_count	mean_runtime	vote_average	Action	Action & Adventure	Adventure	Animation	...	F
0	736775	4	46	550	43.0	8.2	0	1	0	0	...	
1	736635	4	71	303	42.0	7.6	0	1	0	0	...	
2	736272	4	40	647	52.0	8.2	0	0	0	0	...	
3	735513	7	134	6222	44.0	7.6	0	0	0	0	...	
4	734786	9	182	965	60.0	8.3	0	0	0	0	...	

5 rows × 32 columns

In [32]:
```
start_df['rating_category'].describe()
```

Out[32]:
```
count         1507
unique           3
top       average
freq          1044
Name: rating_category, dtype: object
```

In [33]:
```
# check to see if this operation created any NaN/ null values
start_df.isnull().sum()
```

Out[33]:
```
first_air_datetime_ordinal    0
number_of_seasons             0
number_of_episodes            0
vote_count                    0
mean_runtime                  0
vote_average                  0
Action                        0
Action & Adventure            0
Adventure                     0
Animation                     0
Comedy                        0
Crime                         0
Documentary                   0
Drama                         0
Family                        0
Fantasy                       0
```

Before proceeding, we drop our original target column of vote_average, as we will now only be predicting rating_category

In [34]:
```
start_df = start_df.drop('vote_average', axis = 1)
```

Now, we are finally ready to run logistic regression!

Run logistic regression using these variables

In [35]:
```
# note: start_df.drop will not alter start_df in place, so our original start_df is unharmed!
# note 2: we omit vote_average from the X feature set, as it is directly related to rating_category
# from the segmentation we did earlier
X = start_df.drop('rating_category', axis = 1)
Y = start_df['rating_category']
```

In [36]:
```
X
```

Out[36]:

	first_air_datetime_ordinal	number_of_seasons	number_of_episodes	vote_count	mean_runtime	Action	Action & Adventure	Adventure	Animation	Comedy	...	Ne
0	736775	4	46	550	43.0	0	1	0	0	0	...	
1	736635	4	71	303	42.0	0	1	0	0	0	...	
2	736272	4	40	647	52.0	0	0	0	0	0	...	
3	735513	7	134	6222	44.0	0	0	0	0	0	...	
4	734786	9	182	965	60.0	0	0	0	0	0	...	
...	...	...	...	...	...	...	...	...	...	...	...	
1552	735169	5	398	2	48.0	0	0	0	0	1	...	
1553	735697	1	24	172	24.0	0	0	0	1	1	...	
1554	732725	9	41	49	90.0	0	0	0	0	0	...	
1555	737118	1	8	31	45.0	0	1	0	0	0	...	
1556	725377	7	28	64	100.0	0	0	0	0	0	...	

1507 rows × 30 columns

In [37]:
```
Y
```

Out[37]:
```
0       above average
1             average
2       above average
3             average
```

```
1553    above average
1554          average
1555          average
1556          average
Name: rating_category, Length: 1507, dtype: category
Categories (3, object): [below average < average < above average]
```

In [38]:
```
# Split the data
X_train, X_test, Y_train, Y_test = train_test_split(X,Y)

# Make our logistic regression model
start_model = LogisticRegression()

# Fit the model
start_model.fit(X_train,Y_train)
```

```
/Users/derekxu/opt/anaconda3/lib/python3.8/site-packages/sklearn/linear_model/_logistic.py:762: ConvergenceWarning: l
bfgs failed to converge (status=1):
STOP: TOTAL NO. of ITERATIONS REACHED LIMIT.

Increase the number of iterations (max_iter) or scale the data as shown in:
    https://scikit-learn.org/stable/modules/preprocessing.html
Please also refer to the documentation for alternative solver options:
    https://scikit-learn.org/stable/modules/linear_model.html#logistic-regression
  n_iter_i = _check_optimize_result(
```

Out[38]:
```
LogisticRegression()
```

In [39]:
```
# Predict the classes of the testing data set
class_predict = start_model.predict(X_test)

# Compare the predicted classes to the actual test classes
metrics.accuracy_score(Y_test,class_predict)
```

Out[39]:
```
0.6843501326259946
```

Based on this initial result, we can be sure that our naive feature set was at least somewhat correlated with predicting the target. This shows us that we can be confident in moving forward with this feature set and optimizing it as we go.

In order to improve our model's predictive accuracy, we can try various strategies, such as cleaning our data further, improving prediction accuracy by adding more features to our feature set, or trying classification using different supervised learning algorithms.

We will start by cleaning our data more.

Further data cleaning

The Stanford students cleaned their data by removing records where there were a very small number of ratings.

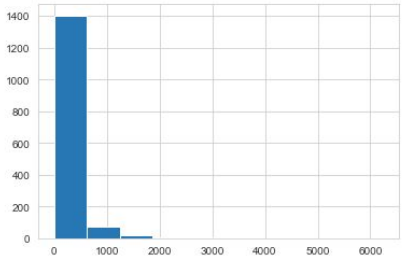
Further data cleaning

The Stanford students cleaned their data by removing records where there were a very small number of ratings.

In order to determine what the threshold of "very small number of ratings" is, we will again use a histogram to plot out the frequency of certain ranges of vote counts from our original df

In [40]:
```
plt.hist(df['vote_count'])
```

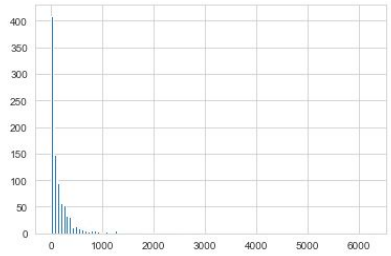
Out[40]:
```
(array([1.403e+03, 7.300e+01, 1.900e+01, 6.000e+00, 4.000e+00, 0.000e+00,
        1.000e+00, 0.000e+00, 0.000e+00, 1.000e+00]),
 array([   0. ,  622.2, 1244.4, 1866.6, 2488.8, 3111. , 3733.2, 4355.4,
        4977.6, 5599.8, 6222. ]),
 <a list of 10 Patch objects>)
```



We can see from this that the majority of vote counts are between 0-622.2. 10 bins were returned in the last section, so lets try 200 bins

In [41]:
```
plt.hist(df['vote_count'], bins=200)
```

Out[41]:
```
(array([409., 207., 147., 122.,  95.,  72.,  56.,  43.,  52.,  34.,  27.,
         32.,  23.,  12.,  19.,  13.,  13.,  10.,  10.,   7.,  12.,   6.,
          7.,   4.,   2.,   5.,   9.,   5.,   2.,   4.,   2.,   2.,   2.,
          0.,   3.,   1.,   1.,   1.,   2.,   3.,   5.,   0.,   1.,   0.,
          0.,   1.,   0.,   1.,   1.,   0.,   2.,   1.,   0.,   2.,   1.,
          2.,   0.,   1.,   1.,   0.,   0.,   2.,   0.,   0.,   1.,   0.,
          0.,   0.,   0.,   0.,   1.,   0.,   0.,   0.,   1.,   0.,   0.,
          0.,   1.,   0.,   0.,   1.,   0.,   0.,   0.,   0.,   0.,   0.,
          0.,   0.,   0.,   1.,   0.,   1.,   0.,   0.,   0.,   1.,   0.,
          0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,
          0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,
          0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,
          1.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,
```



From this, we can still see that the overwhelming majority of votes is between 0-31.11, the first bucket out of 200.

In [42]:
```
df.loc[(df['vote_count'] >= 0) & (df['vote_count'] <= 10)].shape
```

Out[42]:
```
(197, 33)
```

In [43]:
```
df.loc[(df['vote_count'] >= 10) & (df['vote_count'] <= 20)].shape
```

Out[43]:
```
(123, 33)
```

In [44]:
```
df.loc[(df['vote_count'] >= 20) & (df['vote_count'] <= 30)].shape
```

Out[44]:
```
(100, 33)
```

In [45]:
```
type(df.loc[(df['vote_count'] >= 0) & (df['vote_count'] <= 10)])
```

Out[45]:
```
pandas.core.frame.DataFrame
```

From this further investigation, we can see that there are 197 rows in our original dataframe whose vote_count is between 0-10. Therefore, we will drop all rows with vote_count between 0-10 and retrain our model

In [46]:
```
rows_to_drop = df.loc[(df['vote_count'] >= 0) & (df['vote_count'] <= 10)].index.tolist()
```

In [47]:
```
rows_to_drop
```

Out[47]:
```
[18,
 47,
 78,
 91,
 92,
 93,
```

In [48]:
```
second_df = start_df.drop(rows_to_drop)
```

In [49]:
```
second_df.shape
```

Out[49]:
```
(1310, 31)
```

In [50]:
```
start_df.shape
```

Out[50]:
```
(1507, 31)
```

In [51]:
```
print(1507-1310)
```

```
197
```

As expected, 197 rows were dropped corresponding to the rows of our original dataframe that had vote_counts between 0-10. Lets now try training our model again

In [52]:
```
# note: start_df.drop will not alter start_df in place, so our original start_df is unharmed!
X = second_df.drop('rating_category', axis = 1)
Y = second_df['rating_category']

# Split the data
X_train, X_test, Y_train, Y_test = train_test_split(X,Y)

# Make our logistic regression model
second_model = LogisticRegression()

# Fit the model
second_model.fit(X_train,Y_train)

# Predict the classes of the testing data set
class_predict = second_model.predict(X_test)

# Compare the predicted classes to the actual test classes
metrics.accuracy_score(Y_test,class_predict)
```

```
/Users/derekxu/opt/anaconda3/lib/python3.8/site-packages/sklearn/linear_model/_logistic.py:762: ConvergenceWarning: l
bfgs failed to converge (status=1):
STOP: TOTAL NO. of ITERATIONS REACHED LIMIT.

Increase the number of iterations (max_iter) or scale the data as shown in:
    https://scikit-learn.org/stable/modules/preprocessing.html
Please also refer to the documentation for alternative solver options:
    https://scikit-learn.org/stable/modules/linear_model.html#logistic-regression
  n_iter_i = _check_optimize_result(
```

Out[52]:
```
0.6798780487804879
```

In [53]:
```
confusion_matrix = metrics.confusion_matrix(y_true=Y_test, y_pred=class_predict, labels=['below average', 'average', 'a
```

In [54]:
```
confusion_matrix
```

Out[54]:
```
array([[  2,  18,   0],
       [  0, 219,   4],
       [  0,  83,   2]])
```

This bit of data cleaning granted us a 3% improvement in accuracy!

We will now proceed with adding in more features to make predictions on.

Adding to the feature set

At this point, the Stanford researchers concluded their work. However, possibly due to a smaller dataset, or data that is less accurate due to TMDB being a smaller and less popular web resource as compared to IMDB which they used, we will have to incorporate additional features to make predictions using.

https://www.diva-portal.org/smash/get/diva2:1106715/FULLTEXT01.pdf

By investigating previous work done in the field of predicting the success of Movies based on features available prior to box office release, we can draw interesting conclusions.

Firstly, we see that Actors, Directors, Writers, and Budgets are features that are accounted for in the above study.(https://www.diva-portal.org/smash/get/diva2:1106715/FULLTEXT01.pdf)

Similar fields that are available to us from TMDB are:

- created_by, an array of objects
- networks, an array of objects
- production_companies, an array of objects

Also available is data under the /credits api endpoint on TMDb. However, processing this data will not be feasible because it is not structured based on popularity of the actors, and attempting to one-hot encode a list of hundreds of actors, directors, extras, etc. will not be feasible.

The authors of (https://www.diva-portal.org/smash/get/diva2:1106715/FULLTEXT01.pdf) elected to use a decision tree classifier for the purposes of movie performance prediction, as opposed to the logistic regression model used in the stanford paper. We will attempt to adopt this model now.

Additionally, I will drop the vote_count feature at this point in time. Many prior works including (https://www.diva-portal.org/smash/get/diva2:1106715/FULLTEXT01.pdf) , (http://cs229.stanford.edu/proj2015/256_report.pdf) that use IMDb data incorporate the number of ratings/votes as a feature. However, for the purposes of this report, I believe that this column should be dropped entirely, since Netflix and other streaming platforms that are planning to rotate their catalogue of TV shows based on their own proprietary viewership will not be able to provide their vote_counts in a way that can correspond meaningfully with TMDb's own proprietary vote_count data. Therefore, we will proceed without using this column, and make

Drop vote_count

In [55]:
```
# drop vote_count from second_df, and try logistic regression again
third_df = second_df.drop('vote_count', axis=1)

# note: start_df.drop will not alter start_df in place, so our original start_df is unharmed!
X = third_df.drop('rating_category', axis = 1)
Y = third_df['rating_category']

# Split the data
X_train, X_test, Y_train, Y_test = train_test_split(X,Y)

# Make our logistic regression model
third_model = LogisticRegression()

# Fit the model
third_model.fit(X_train,Y_train)

# Predict the classes of the testing data set
class_predict = third_model.predict(X_test)

# Compare the predicted classes to the actual test classes
metrics.accuracy_score(Y_test,class_predict)
```

Out[55]:
```
0.7713414634146342
```

We note here that there is a minimal decrease in the accuracy, despite dropping the vote_count feature!

Now, we attempt to use a decision tree classifier directly on our data

Use Decision Tree classifier

In [56]:
```
from sklearn import tree
from sklearn.tree import DecisionTreeClassifier, export_graphviz

# we opt to use the default min_samples_split here
classifier_model = DecisionTreeClassifier()
```

In [57]:
```
# note: start_df.drop will not alter start_df in place, so our original start_df is unharmed!
X = third_df.drop('rating_category', axis = 1)
Y = third_df['rating_category']

# Split the data
X_train, X_test, Y_train, Y_test = train_test_split(X,Y)

# Fit the model
classifier_model.fit(X_train,Y_train)
```

Out[57]: 0.6737804878048781

We note a decrease in the overall prediction accuracy of the decision tree as compared to logistic regression.

To fix this, we can try to tune our decision tree parameters: https://towardsdatascience.com/how-to-tune-a-decision-tree-f03721801680

This (https://arxiv.org/abs/1812.02207) research paper indicates through empirical analysis that for the CART algorithm, which is used by the sklearn decision tree classifier (https://scikit-learn.org/stable/modules/tree.html#tree-algorithms-id3-c4-5-c5-0-and-cart), "hyperparameters such as 'minsplit' and 'minbucket' are the most responsible for the performance of the final trees"

Therefore, we will investigate how altering min_samples_split and min_samples_leaf, which correspond to 'minsplit' and 'minbucket' respectively, will alter the performance of our decision tree.

In [58]:
```
# multiply both default values by 5
classifier_model_2 = DecisionTreeClassifier(min_samples_split=10, min_samples_leaf=5)

# Fit the model
classifier_model.fit(X_train,Y_train)

# Predict the classes of the testing data set
class_predict = classifier_model.predict(X_test)

# Compare the predicted classes to the actual test classes
metrics.accuracy_score(Y_test,class_predict)
```

Out[58]: 0.6676829268292683

In [59]:
```
# randomized trial
classifier_model_3 = DecisionTreeClassifier(min_samples_split=9, min_samples_leaf=3)

# Fit the model
classifier_model.fit(X_train,Y_train)

# Predict the classes of the testing data set
class_predict = classifier_model.predict(X_test)

# Compare the predicted classes to the actual test classes
metrics.accuracy_score(Y_test,class_predict)
```

Out[59]: 0.6707317073170732

After a few attempts, it seems that performance enhancements by altering these hyperparameters are largely random and redundant. We avoid extremely large values, as they were reported to have very poor performance in the paper (https://arxiv.org/abs/1812.02207) on page 16.

Adding to our feature set

Adding to our feature set

The intution for adding this column is that we need some way of measuring the success of a TV show based on the actual people involved with the show. Since the TMDB dataset lacks a "directors" or "star actors" feature, I believe that the "created_by" feature best approximates the features used in the previous works such as (https://www.diva-portal.org/smash/get/diva2:1106715/FULLTEXT01.pdf).

Production companies is another possibility, however intuitively, there are very few large production companies, but all TV shows must be produced by them, or independently. There exist large production companies that have produced hit TV shows, and also much less popular tv shows. Creators on the other hand have a much closer artistic relationship with the TV shows that they create, possibly implying a stronger relationship that could be capitalized on by our supervised learning algorithms.

In [60]:
```
df['created_by'].describe()
```

Out[60]:
```
count     1507
unique    1203
top         []
freq       305
Name: created_by, dtype: object
```

In [61]:
```
df['production_companies'].describe()
```

Out[61]:
```
count     1507
unique     983
top          []
freq       323
Name: production_companies, dtype: object
```

After this investigation, neither the created_by, nor production_companies features are good candidates for adding into our feature set. Both contain a very large amount of unique values, and the most frequently repeating value for both is an empty array. One-hot encoding such data would result in 1000+ columns. Therefore, we abandon this approach

Add overview to feature set, TF-IDF analysis

Previous research papers (https://www.tandfonline.com/doi/full/10.1080/07421222.2016.1243969?casa_token=fhRY1Aba6ywAAAAA%3A5R579Zqpd1hLiXQizWaMB2dw6Kz0l3aw3-yayDENBijySIMIBo1g19Yl1HphY8QO8NneyQT8U231Ig) have discussed the efficacy of studying plot synopses of movies in predicting their performance prior to release. We will attempt to replicate this using the 'overview' feature of our original df

In [62]:
```
df['overview'].describe()
```

Out[62]:
```
count                                                1470
unique                                               1470
```

Add overview to feature set, TF-IDF analysis

Previous research papers (https://www.tandfonline.com/doi/full/10.1080/07421222.2016.1243969?casa_token=fhRY1Aba6ywAAAAA%3A5R579Zqpd1hLiXQizWaMB2dw6Kz0l3aw3-yayDENBijySlMIBo1g19YI1HphY8QO8NneyQT8U231Ig) have discussed the efficacy of studying plot synopses of movies in predicting their performance prior to release. We will attempt to replicate this using the 'overview' feature of our original df

In [62]:
```
df['overview'].describe()
```

Out[62]:
```
count                                                1470
unique                                               1470
top       Camelot is a historical-fantasy-drama televisi...
freq                                                    1
Name: overview, dtype: object
```

In [63]:
```
df['overview'].isnull().sum()
```

Out[63]:
```
37
```

From this exploration, we see that there are only 37 rows that contain null values in the overview section. This tells me that this column is a good candidate to add to our feature set!

In order to make these overviews useful in the context of classification, we will convert the entire overview column into TF-IDF format.

https://scikit-learn.org/stable/modules/feature_extraction.html#text-feature-extraction

prep overview data to be merged into third_df

In [64]:
```
# get overview column and drop null values
overview_col = df['overview'].dropna()
```

In [65]:
```
# check to make sure 1470 rows still present after dropping null values
overview_col.shape
```

Out[65]:
```
(1470,)
```

In [66]:
```
# merge first with third_df to preserve indexes
third_df = third_df.join(overview_col, how='inner')
```

In [67]:
```
third_df.shape
```

Out[67]:
```
(1301, 31)
```

Following this, we can turn our overview column into tfidf format

In [68]:
```
from sklearn.feature_extraction.text import TfidfVectorizer
vectorizer = TfidfVectorizer()
overview_col = third_df['overview'].copy()
overview_tfidf = vectorizer.fit_transform(overview_col.values.astype('U'))
```

In [69]:
```
overview_tfidf.shape[0]
```

Out[69]:
```
1301
```

In [70]:
```
# reset index to avoid concatenation errors after generating a new df using the tfidf vectorizer
third_df.reset_index(drop=True)
```

Out[70]:

	first_air_datetime_ordinal	number_of_seasons	number_of_episodes	mean_runtime	Action	Action & Adventure	Adventure	Animation	Comedy	Crime	...	Romanc
0	736775	4	46	43.0	0	1	0	0	0	0	...	
1	736635	4	71	42.0	0	1	0	0	0	1	...	
2	736272	4	40	52.0	0	0	0	0	0	0	...	
3	735513	7	134	44.0	0	0	0	0	0	0	...	
4	734786	9	182	60.0	0	0	0	0	0	0	...	
...	...	...	...	...	...	...	...	...	...	...	...	
1296	732373	4	109	26.0	0	0	0	1	0	0	...	

In [71]:
```
# create fourth_df out of new overview_tfidf columns
overview_tfidf_df = pd.DataFrame(overview_tfidf.toarray(), columns=vectorizer.get_feature_names())
fourth_df = third_df.join(overview_tfidf_df)
```

In [72]:
```
fourth_df.columns
```

Out[72]:
```
Index(['first_air_datetime_ordinal', 'number_of_seasons', 'number_of_episodes',
       'mean_runtime', 'Action', 'Action & Adventure', 'Adventure',
       'Animation', 'Comedy', 'Crime',
       ...
       'zuo', 'zuri', 'zwick', 'álex', 'álvarez', 'ángel', 'écija', 'ömer',
       'über', 'とらぶる'],
      dtype='object', length=11660)
```

In [73]:
```
fourth_df.shape
```

Out[73]:
```
(1301, 11660)
```

In [74]:
```
# drop the overview strings column
fourth_df = fourth_df.drop('overview', axis=1)
```

In [75]:
```
fourth_df.shape
```

Out[75]:
```
(1301, 11659)
```

In [76]:
```
fourth_df.head()
```

Out[76]:

	first_air_datetime_ordinal	number_of_seasons	number_of_episodes	mean_runtime	Action	Action & Adventure	Adventure	Animation	Comedy	Crime	...	zuo	zuri
0	736775	4	46	43.0	0	1	0	0	0	0	...	0.0	0.0
1	736635	4	71	42.0	0	1	0	0	0	1	...	0.0	0.0
2	736272	4	40	52.0	0	0	0	0	0	0	...	0.0	0.0
3	735513	7	134	44.0	0	0	0	0	0	0	...	0.0	0.0
4	734786	9	182	60.0	0	0	0	0	0	0	...	0.0	0.0

5 rows × 11659 columns

In [77]:
```
fourth_df.isnull().sum()
```

In [78]:
```
fourth_df = fourth_df.dropna()
```

In [79]:
```
fourth_df.shape
```

Out[79]:
```
(1107, 11659)
```

In [80]:
```
fourth_df.isnull().values.any()
```

Out[80]:
```
False
```

Now, lets attempt to use the new tfidf vector to get a more accurate answer

In [81]:
```
# we opt to use the default min_samples_split here, since our experimentation did not yield improved error rates
classifier_model = DecisionTreeClassifier()

# note: start_df.drop will not alter start_df in place, so our original start_df is unharmed!
X = fourth_df.drop('rating_category', axis = 1)
Y = fourth_df['rating_category']

# Split the data
X_train, X_test, Y_train, Y_test = train_test_split(X,Y)

# Fit the model
classifier_model.fit(X_train,Y_train)

# Predict the classes of the testing data set
class_predict = classifier_model.predict(X_test)

# Compare the predicted classes to the actual test classes
metrics.accuracy_score(Y_test,class_predict)
```

Out[81]:
```
0.7220216606498195
```

As we can see, there were no improvements yielded by using this tfidf vector in decision trees. Lets try again using the logistic regression

In [82]:
```
# note: start_df.drop will not alter start_df in place, so our original start_df is unharmed!
X = fourth_df.drop('rating_category', axis=1)
Y = fourth_df['rating_category']

# Split the data
X_train, X_test, Y_train, Y_test = train_test_split(X,Y)

# Make our logistic regression model
fourth_log = LogisticRegression()

# Fit the model
```

Out[82]: 0.6895306859205776

We see no improvement here. Lets try predicting using only the genres and tfidf.

In [83]:
```
fifth_df = fourth_df.drop(['first_air_datetime_ordinal', 'number_of_seasons', 'number_of_episodes', 'mean_runtime'], ax
```

In [84]:
```
# note: start_df.drop will not alter start_df in place, so our original start_df is unharmed!
X = fifth_df.drop('rating_category', axis=1)
Y = fifth_df['rating_category']

# Split the data
X_train, X_test, Y_train, Y_test = train_test_split(X,Y)

# Make our logistic regression model
fifth_log = LogisticRegression()

# Fit the model
fifth_log.fit(X_train,Y_train)

# Predict the classes of the testing data set
class_predict = fifth_log.predict(X_test)

# Compare the predicted classes to the actual test classes
metrics.accuracy_score(Y_test,class_predict)
```

Out[84]: 0.7184115523465704

k-fold cross validation

Finally, lets compare the performance between our 3 highest performing models https://scikit-learn.org/stable/modules/cross_validation.html#cross-validation

In [85]:
```
from sklearn.model_selection import KFold
from sklearn.model_selection import cross_val_score
from numpy import mean
from numpy import std
```

Original logistic regression with vote_count dropped

In [86]:
```
# create dataset
X = third_df.drop(['rating_category', 'overview'], axis = 1)
Y = third_df['rating_category']

# create model
third_model = LogisticRegression()

# evaluate model
scores = cross_val_score(third_model, X, Y, scoring='accuracy', cv=10)

# report performance
print('Accuracy: %.3f (%.3f)' % (mean(scores), std(scores)))
```

```
Accuracy: 0.734 (0.011)
```

Logistic regression including TF-IDF

In [87]:
```
# create dataset
X = fourth_df.drop('rating_category', axis=1)
Y = fourth_df['rating_category']

# create model
fourth_log = LogisticRegression()

# evaluate model
scores = cross_val_score(fourth_log, X, Y, scoring='accuracy', cv=10)

# report performance
print('Accuracy: %.3f (%.3f)' % (mean(scores), std(scores)))
```

```
Accuracy: 0.725 (0.011)
```

Logistic regression of TF-IDF only

In [88]:
```
# create dataset
X = fifth_df.drop('rating_category', axis=1)
Y = fifth_df['rating_category']

# create model
fifth_log = LogisticRegression()

# evaluate model
scores = cross_val_score(fifth_log, X, Y, scoring='accuracy', cv=10)

# report performance
print('Accuracy: %.3f (%.3f)' % (mean(scores), std(scores)))
```

```
Accuracy: 0.737 (0.023)
```