Creating a Movie Recommendation Engine using Naïve Bayes for Text Classification

Dylan McCardle
COMP 5600
Auburn University
Auburn, AL 36832
dcm0033@auburn.edu

Corey Myers
COMP 5600
Auburn University
Auburn, AL 36832
cam0112@auburn.edu

Evan Sheehan
COMP 5600
Auburn University
Auburn, AL 36832
res0038@auburn.edu

Abstract

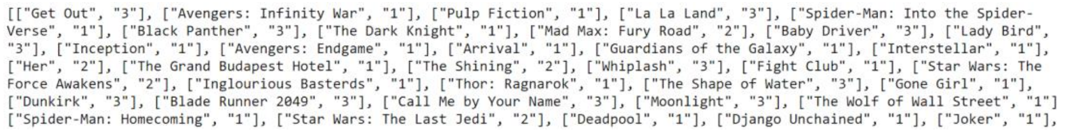
An increase in the amount of available data in the world demands computation in order to tailor and present that data in the most appropriate way. In this paper we propose a model for utilizing Naïve Bayes for text classification with the intent to make movie recommendations based on user input. We constructed our own data set consisting of movie titles and dictionaries with key-value word and word frequency pairs, representing the collection of words in the reviews for each movie. We manually implemented Naïve Bayes to binarily classify these dictionaries into 'like' or 'dislike' based on the initial user input. Unfortunately, our final iteration, using logarithmic Naïve Bayes, yielded no meaningful results. Further testing and optimization is needed to determine the validity of the model proposed here.

1 Problem formulation

The exponential increase of data in the past 30 years can be difficult to comprehend. The creation of the internet allowed masses of data to presented to the user, and information overload very quickly became a problem. We can contextualize this issue on a smaller scale when discussing media. What is the best way to discover new music and movies? It must be tailored and presented to the user in some way, or else discovering media you enjoy would be a laborious task. Among other organizational and presentational tools, recommendation engines emerged. These engines collect information about the user and utilize algorithms to make new recommendations based on observations about that data. In this paper we present a simple model for recommending movies using Naïve Bayes and text classification on sets of user movie reviews.

Initially, the user had to input three to five movies that they liked and disliked. Then, the reviews for the movies will be parsed, creating a vocabulary of important words found. The system will then locate any movies that have similar or contrary word vocabularies to the user-inputted liked and disliked movies. The system will then output three to five different movies that the user may like and dislike based on the similarities and dislikes of words between the input movies and output movies.

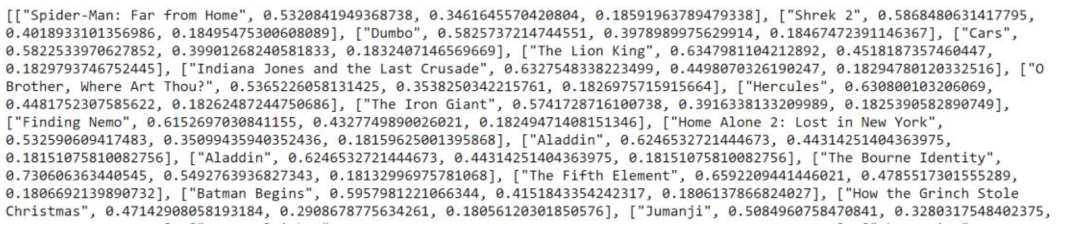
After consideration, the most optimal strategy for the final iteration of the system was to have the user input a list of movies that they have liked, disliked, or have not seen. Then, the system will parse through the top 100 reviews of each of the top 1000 most popular movies. The system will then output the list of movies with a percentage score depending on if the system thinks the user will like, dislike, or has not seen a certain movie, depending on the word vocabulary in each review.



```
[["Get Out", "3"], ["Avengers: Infinity War", "1"], ["Pulp Fiction", "1"], ["La La Land", "3"], ["Spider-Man: Into the Spider-
Verse", "1"], ["Black Panther", "3"], ["The Dark Knight", "1"], ["Mad Max: Fury Road", "2"], ["Baby Driver", "3"], ["Lady Bird",
"3"], ["Inception", "1"], ["Avengers: Endgame", "1"], ["Arrival", "1"], ["Guardians of the Galaxy", "1"], ["Interstellar", "1"],
["Her", "2"], ["The Grand Budapest Hotel", "1"], ["The Shining", "2"], ["Whiplash", "3"], ["Fight Club", "1"], ["Star Wars: The
Force Awakens", "2"], ["Inglourious Basterds", "1"], ["Thor: Ragnarok", "1"], ["The Shape of Water", "3"], ["Gone Girl", "1"],
["Dunkirk", "3"], ["Blade Runner 2049", "3"], ["Call Me by Your Name", "3"], ["Moonlight", "3"], ["The Wolf of Wall Street", "1"]
["Spider-Man: Homecoming", "1"], ["Star Wars: The Last Jedi", "2"], ["Deadpool", "1"], ["Django Unchained", "1"], ["Joker", "1"],
```

Figure 1: Rated movies

The above picture shows a sample of movies that the user may input into the system: putting a “1” if they liked the movie, a “2” if they disliked the movie, or “3” if they have not seen the movie. The system will then create dictionaries of words from the reviews of movies that the user input. The words will then be compared to the words in reviews of the top 1000 movies and create a percentage of how likely the user would like, dislike, or not have seen the movie, as seen in the picture below.



```
[["Spider-Man: Far from Home", 0.5320841949368738, 0.3461645570420804, 0.18591963789479338], ["Shrek 2", 0.5868480631417795,
0.4018933101356986, 0.18495475300608089], ["Dumbo", 0.5825737214744551, 0.3978989975629914, 0.18467472391146367], ["Cars",
0.5822533970627852, 0.39901268240581833, 0.1832407146569669], ["The Lion King", 0.6347981104212892, 0.4518187357460447,
0.1829793746752445], ["Indiana Jones and the Last Crusade", 0.6327548338223499, 0.4498070326190247, 0.18294780120332516], ["O
Brother, Where Art Thou?", 0.5365226058131425, 0.3538250342215761, 0.1826975715915664], ["Hercules", 0.630800103206069,
0.4481752307585622, 0.18262487244750686], ["The Iron Giant", 0.5741728716100738, 0.3916338133209989, 0.1825390582890749],
["Finding Nemo", 0.6152697030841155, 0.4327749890026021, 0.18249471408151346], ["Home Alone 2: Lost in New York",
0.532590609417483, 0.35099435940352436, 0.18159625001395868], ["Aladdin", 0.6246532721444673, 0.44314251404363975,
0.18151075810082756], ["Aladdin", 0.6246532721444673, 0.44314251404363975, 0.18151075810082756], ["The Bourne Identity",
0.730606363440545, 0.5492763936827343, 0.18132996975781068], ["The Fifth Element", 0.6592209441446021, 0.4785517301555289,
0.180669213989073 2], ["Batman Begins", 0.5957981221066344, 0.4151843354242317, 0.1806137866824027], ["How the Grinch Stole
Christmas", 0.47142908058193184, 0.2908678775634261, 0.18056120301850576], ["Jumanji", 0.5084960758470841, 0.3280317548402375,
```

Figure 2: Output of results

2 Infrastructure

In order to specify the parameters of the data set more minutely, we decided to not use any pre-constructed sets and instead created our own data set. The method in which we created this data set changed a few times but the aspects that stayed constant were the ways in which we gathered the raw data itself. The premise for collecting data was to mostly use Selenium WebDriver with BeautifulSoup4. Selenium is a tool often used for automated web testing. It allows us to write scripts that scrape websites without sending more http requests than would be expected from a normal user. BeautifulSoup is a popular Python library used for processing HTML data. Selenium was used for navigation and basic scraping while BeautifulSoup was utilized more for grabbing harder to reach information within the HTML. A side effect of Selenium emulating user navigation is that it can be very slow since it waits for a page to load completely before it completes the next action. Because of this, our initial approach to gathering data ended up being inefficient and impractical.

2.1 Initial approach

The initial plan was to dynamically construct the data set, which would utilize parts of some pre-constructed data sets to avoid information overload and make the results more relevant for the user case. For example, say the user inputs 3 movies they like into the program. Selenium opens a Google Chrome window and navigates to IMDB. The movie titles are searched and some or all of the reviews are extracted from the HTML. Selenium would then grab a small number of related movies according to IMDB for each movie. This would begin to build an initial data set of similar movies. For each of those similar movies, we could run the same Selenium process for collecting its movie reviews and gathering more similar movies. This would clearly need to have some upper limit as to not run forever.

There are two main issues with this approach. First, using IMDB’s own recommendation engine as a basis for our own immediately colors our results. We could not guarantee the extent to which our results are because of an effective implementation of Naïve Bayes instead of a result of using IMDB’s own engine as the basis for our data set. The second issue with this approach is the performance of Selenium. Selenium is designed to act like a user, and users cannot necessarily gather raw data quickly like a computer can. Selenium waits for a page to fully load to continue the program. Because of this, it could take 15-20 seconds to gather the top 100 reviews for a single movie. A user input of 3 liked and 3 disliked movies would take nearly 3 minutes to complete. That is only considering the time cost of the input information. Building a useful data set, say at least 100 movies, would take way too long for something that needs to be done at runtime. For these two reasons, we decided to redesign our data gathering approach.

2.2 Implemented approach

For our implemented approach to building a data set, we decided against the idea of dynamically creating one at runtime. Instead, we decided to continue to utilize Selenium with BeautifulSoup but prebuild the data set. We decided to gather 1000 movies and 100 user reviews for each of these movies. For the movie titles, we utilized another movie database website called Letterboxd.com. We used Letterboxd to sort the movies by most popular of all time, in descending order. Letterboxd can display 72 movies on a single page, each with their title information in the HTML. This is optimal because it means we only need to navigate through 14 pages to extract the 1000 movie names for our data set.

We used this list of 1000 movie titles as a data set to search for reviews with. We chose to use IMDB for user review data. Although Letterboxd has plenty of user reviews to pull from, we found anecdotally that the reviews were shorter and often devoid of traditionally meaningful data. For example, many of the top-rated reviews were a short a joke about something that occurred in the movie. Using the method previously described, we navigated IMDB using Selenium and grabbed the 100 most helpful reviews for each movie. The speed of Selenium is still an issue in this case, but we only need to build the data set once. We designed the navigator to be able to iteratively output files in the case of a crash or a user exit. This was necessary considering the amount of time it takes to gather information for 1000 movies using this method.

Once the raw data was gathered, we did a small amount of preprocessing including the removal of stopwords from reviews (mostly articles such as ‘a’ and ‘the’). We organized each set of reviews as a dictionary of key-value pairs. The key is a string, representing some word that occurred in the set of reviews. The value is an integer representing the amount of times that word occurred in the set of reviews. This data set allows us to easily grab the frequency any specific word occurs in the reviews of any specific movie title.

3 Naïve Bayes variations and usefulness

Naïve Bayes is an algorithm that relies on conditional independence to assign class definitions to documents. In the case of our problem, the class definitions that the Naïve Bayes will be predicting is whether or not the movie is liked or disliked by the user. The algorithm is described in figure 3 below. C_k is the probability that a document is in the class k, or in our case like or dislike. P(x) is the probability of each individual feature 3oolean3g, or the words contained in the movie reviews. P(x|C_k) is the probability that the feature x appears in class C_k , or the probability that a word x_i appears in class C_k . This can be calculated by multiplying each probability of a word occurring in a class by each other. This total that you get will give you the probability that the document you are classifying belongs in class C_k. This is the basic implementation of Naïve Bayes theorem.

$$p(C_k \mid \mathbf{x}) = \frac{p(C_k)\, p(\mathbf{x} \mid C_k)}{p(\mathbf{x})}$$

Figure 3: Naïve Bayes probability

3.1 Naïve Bayes Usefulness to our problem

Naïve Bayes when implemented in the way described above is actually not all that useful. This comes from the fact that when you do all of the multiplication presented in naïve Bayes, if the probability of any word being in class C_k is equal to 0, the entire probability evaluates to 0. To fix

this, we implement something known as laplace or +1 smoothing. Basically, when calculating all the probabilities, we add 1 to the number of features x_i in C_k so that every probability has at least a small percentage chance of 4oolean4g. This fixes the problem of having a zero probability.
It is not the only problem with this model of Naïve Bayes though.One more issue which very much impacts our data is decimal underflow error. When multiplying these decimals over and over and over again, each one is very small, and they get progressively smaller with each multiplication operation. This eventually leads to underflow where the number gets so small it just evaluates to 0, meaning all the calculations mean nothing. A solution to this problem is logarithmic smoothing. By performing the operations above in log space we can evaluate the function as a sum of logarithms. This is possible because $\log(ab) = \log(a) + \log(b)$. This solves our problem of decimal underflow as the problem is no longer multiplying very small quantities but summing them. At this point we finally have a usable formula.

3.2 Naïve Bayes variations

There are several naïve Bayes variations which can be used to alter the feature set, the algorithm, or both, to provide different results. The first of these is gaussian naïve Bayes. Gaussian Naïve Bayes is an algorithm that uses continuous data and a gaussian distribution of that data to make class distinctions. This is not useful in our case as our data is not distributed in a continuous distribution, but is distributed between features.

Another naïve Bayes algorithm is Bernoulli naïve Bayes, which calculates each probability, not by the number of features, but by the inclusion of these features in the document. It calculates each feature as a 4oolean "is in document" or "not in document". A hallmark of this variation on the algorithm is that it does not require laplace smoothing, as it accounts for the non-inclusion of a feature within a document. The reason we did not include this Naïve Bayes implementation is because the number of words within our data, means that if the word does not appear, which it will generally not, it will weigh it more so than the inclusion of that word, meaning our data will skew heavily towards the non-inclusion of features.

The last variation which we will discuss that we did not implement is tree based Naïve Bayes. It uses decision trees with each leaf node being a populated Naïve Bayes distribution. We did not implement this as it works best with a very small sets of data to compare to, whereas we have very large vocabularies within each class with which to compare to.

4 Error analysis

Error analysis initially seemed tricky in our case. It was difficult to see how we could guarantee our algorithm was working or not working on something as subjective as movie preference. Our solution was to cycle through the data set to create a hand-labeled list of movies with the like and dislike classes (and unseen) assigned. From this list we created smaller list that only included the liked and disliked movies, the unseen would be irrelevant for error analysis.

To test random data and confirm that our trends line up statistically, we also created a program that would rate every movie like or dislike, with a certain probability to choose one class over the other.

Using this list, we constructed a script that can take a percentage and split the list into training and test sets. For example, if we have a list of 100 movies that we rated with 60 liked and 40 disliked, using 10% of both the liked and disliked set will give us two training sets of size 6 and 4 respectively. The remaining movies, 54 liked and 36 disliked, are used as the test set.

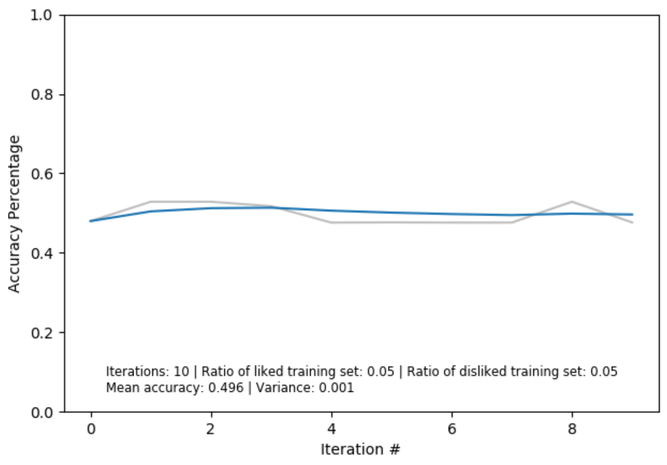


Figure 4: Recommendation for randomly rated movies

Using this method, we found so far that we have no meaningful results. The implementation is incomplete at best and this model is incompatible with this problem at worst. Results show that there is a strong skew in the way probabilities are calculated, leading to the test set being almost entirely labeled as 'like' or entirely labeled as 'dislike.' For randomly rated data, we can see random recommendations as expected, as show in Figure 4. However, for real data that does not distribute evenly between classes, we often see the model make recommendations for only the most common class, or at least have a heavy bias towards that class. Further testing and optimization is necessary to determine the validity of this model in this context.