Daniel Delijani

CS 506

Professor Galletti

Midterm Report

Model Writeup – Amazon Reviews

Part 1: Preliminary Analysis

To begin working on the assignment, I began exploring the data. We were ultimately granted 7 columns:

ProductID, UserID, HelpfulnessNumerator, HelpfullnessDenominator, Score, Time, Summary, Text, and ID. Immediately, there were two categories I felt to be unuseful: Time and ID. I then thought of using Helpfulness as a means of modeling the data, but upon further review (see Figure 1) there does not appear a strong enough trend in the data worth developing a model over. The main categories I found intriguing were thus ProductID, UserID, Summary, and Text. For ProductID and UserID, I considered the possibility of predicting based on average score for a specific ProductID and a specific UserID. However, I found using summary and text to build predictions through Natural Language Processing to be logical, likely highly effective, and more engaging, allowing me to implement state-of-the-art techniques I learned in class. So, I did some analysis on Summary and Text and found some interesting results which would later inform decisions I make in part 3. First, I find the 20 most common words in each review category:

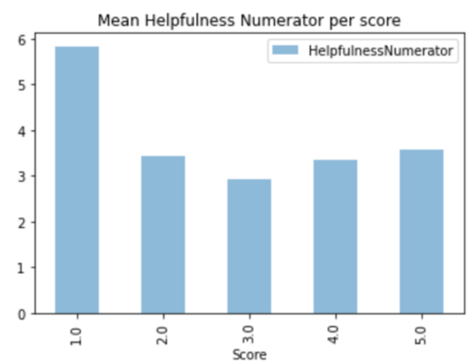

Figure 1

* Note: All code used to generate referenced preliminary analysis data is included in Preliminary Analysis.ipynb

1 Star Ratings: like, just, bad, time, good, don, really, people, story, make, did, watch, way, know, better, quote, acting, think, plot, great

2 Star Ratings: like, just, good, really, story, time, don, bad, better, make, way, did, people, great, plot, quot, know, think, little, does

3 Star Ratings: good, like, just, great, story, time, really, way, little, quot, love, don, people, best, does, watch, life, series, films, make

4 Star Ratings: great, like, good, just, time, love, story, really, best, series, watch, quot, life, people, way, season, don, seen, films, new

5 Star Ratings: great, like, good, just, time, love, story, really, best, series, watch, quot, life, people, way, season, don, seen, films, new

This was not the results I initially hoped for. The top words are pretty similar to each other. Thus, I would have to eventually learn to adjust for this in my model. I would perhaps filter out certain words, or weigh words that appear frequently less. I later found ways of doing so, which will be explained in Part 3.

Part 2: Feature Extraction

As previously mentioned, we have two topics that could be used for analysis. The first is Summary, which is the general summary the user gives his/her review. The second is Text, which is the actual review. Initially, I debated which one I should use, but then I figured why not use both? So, I wrote functions that clean the data (remove punctuation, make lowercase) called cleanstringsummary and cleanstringtext. I creatively developed two functions for this due to an unexpected problem I was facing, as will be described in part 5. Ultimately, my "processedinput" variable consisted of the data from the Text and Summary features, cleaned and concatenated.

Part 3: Flow, Decisions, Techniques Used

Before I delve into my ultimate decision to utilize SVM to develop my model, we must first develop the first two parts of the model. I decided to use sklearn's "Pipeline" to chain the various steps of the model, as I felt it to be the most clean and clear implementation.

Step 1: CountVectorizer. This method takes in my processedinput, tokenizes the data, and finds and stores the frequency of each word. This is a crucial step, as if strongly positive words are mentioned more often in a review that should surely be taken into account. In this step, I additionally pass in parameter max_df = .8. This caused all words that appear in 80% of reviews to be ignored in this step. I did this due to the understanding I gained of the data in my preliminary analysis. Certain words are very common among the entire dataset. For example, the word "just" is among the top 5 most used words in every rating category. So, words like "just" are omitted by passing in this parameter, allowing our model to better fit our data.

Step 2: TfidfTransformer. This is a step I felt must be implemented in my model as a result of the commonality of certain words in my preliminary analysis. TfidfTransformer allows us to consider the relative frequency of words among all reviews, allowing us to mitigate the value of words that appear frequently throughout all reviews regardless of rating. Therefore, more common words such as "know" will be much less valued in the model's evaluation of the processed input. Additionally, I found that setting sublinear_tf = True helped my model better fit the data, as it addresses the issue that 10 occurrences of a certain word is not necessarily 10x more significant than one occurrence of the word by applying 1 + log(tf) rather than tf.

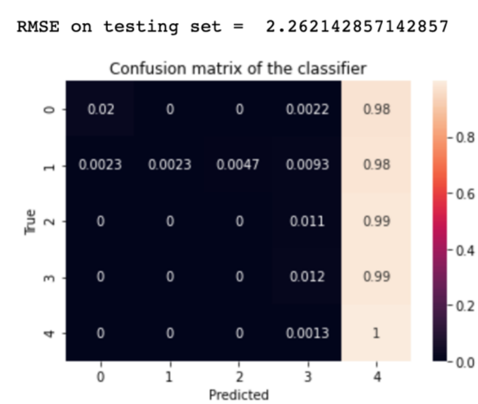

Figure 2

Step 3: SVM. Initially, I did not utilize SVM, as I found out of the large runtime the algorithm tends to have. My first inclination was to utilize a Naïve Bayes algorithm, as I learned it is highly scalable and is not sensitive to irrelevant features, which I felt are two points that were very important in properly predicting based off the particular data. However, when I implemented it I obtained a troubling result, as shown in figure 2. My model would consistently predict 5, regardless of the input. So, I wondered why that may be and explored the math behind the algorithm: $P(A|B) = \frac{P(B|A)\,P(A)}{P(B)}$

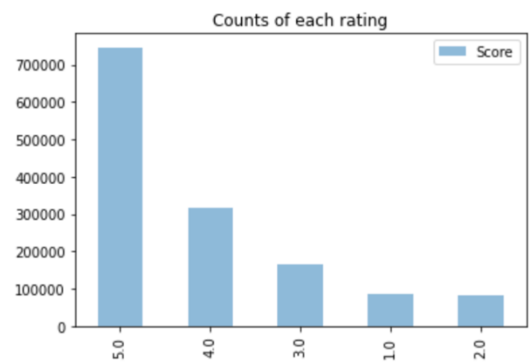

Figure 3

The probability of a certain class is clearly strongly affected by the frequency at which the class is represented in the training data. So, when I noticed this I ran some analysis on the data to find the frequency of each class in the data, and obtained Figure 3. Therefore, I found that I should switch the last element of my model. I knew that SVM has a large runtime so I was unsure about it. But after learning of the fact that it can be utilized well with biased data with the “class_weight” parameter I decided to try it with training a small subset of the data (initially 7000 datapoints).

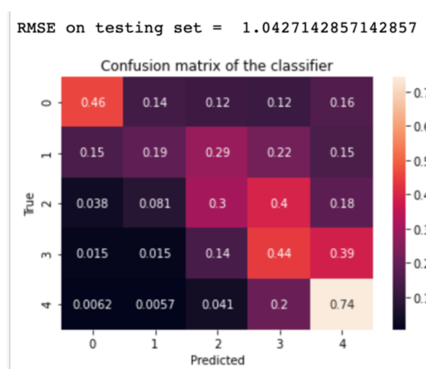

Figure 4

Also, I learned that setting the kernel to be ‘linear’ strongly helps with runtime. So, I set class_weight to ‘balanced’, kernel to be ‘linear’, and ran the model. It worked well, with a RMSE of 1.3 when being tested and trained on 7000 points each. However, I noticed the model predicted the true rating plus or minus 1 with great accuracy. It wasn’t getting the exact value enough. So, after doing some research I decided to increase the value of C, which penalizes each wrong data point, slowly adjusting it until I eventually got the most optimal value, which I found to be C = 2.0. This dropped my RMSE significantly down to 1.04 as seen in figure 4. This concludes the final step in the model Pipeline.

Part 4: Validation

Throughout the development process of my model, I continually used train_test_split to split my model into test and train sets, using train_size and test_size to control their sizes. I then extracted the same features from the test set and utilized RMSE to get a general understanding of the model's performance (as Kaggle used the same), and a confusion matrix to understand exactly what the model was getting wrong. For example, this proved invaluable in my decision to switch from the Naïve Bayes model in step 3 to the SVM. By the end of my testing, Figure 5 is my ultimate pair of performance metrics when being used to train on 60000 datapoints and tested on 7000 datapoints. Additionally, Figure 4.1 represents my final model when trained on 7000 datapoints and tested on 7000 datapoints, which only took my MacBook Air approximately 5 minutes to run.

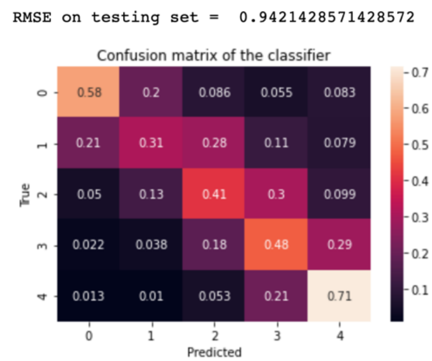

Figure 5

Part 5: Creativity/Challenges/Effort

While I faced a variety of challenges throughout the development of my model that had to be tweaked and accounted for, there are two challenges in particular that I had developed the most creative solutions to. First is my implementation of cleanstringsummary and cleanstringtext in my feature extraction stage. When looking to merge the two columns of the data, I struggled to figure out how to properly merge the subject and text fields while not having the last word in subject and the first in text being merged into the same word. I experimented with developing a new dataframe with exclusively spaces as their elements, for example, but this proved to have a very large time complexity. So, my choice was to split my ‘cleanstring’ function, which inputs a string and removes punctuation and changes all characters to lowercase into two functions, one for the summary column and one for the text column. For the summary, after I finished processing the string I return the processed string + ‘ ’ to account for the merge with text, and this ultimately proved successful. I apply the cleanstringsummary to the summary column, and cleanstringtext to the text column, then add the two together to concatenate corresponding elements. This out-of-the-box solution allowed me to properly obtain my input data for the train strings. However, I again encountered an issue when I went to test the model. Some values for text and summary were NaN, which raised errors in my code. Another creative workaround I utilized was to, in this case, return “good”. While some would suggest simply returning the empty string, I actually found that returning “good” was more effective, as it leads to a neutral rating in the case that both text and summary are NaN, frequently giving it a somewhat accurate response. I additionally tested using a variety of words such as “great”, “amazing”, “wonderful”, but I found simply returning “good” led to the most accurate estimate in these cases.

This concludes the process of developing the model, from data exploration to implementation and testing. Overall, though it was much work, it was rewarding and very engaging! Great way to utilize many of the skills we have been learning in class. Thank you for your time!