MP1 Report
CS165A

Norman Chung
5550926
normanchung@ucsb.edu

Architecture: Explanation of your code architecture (briefly describe the classes and basic functionality)

For my code architecture, I have two dictionaries in python. One is for the positive reviews, and one is for the negative reviews. I parse through the training data, and put the words in the relevant dictionary. I make sure to keep count of how many times the word shows up, and then use that in the calculation for its probability. Then, we move to the testing function. Here, we run the test on the training data and the testing data, without looking at the label for each review. The probability of whether a word is positive or negative is calculated, and at the end of the review, whichever is higher means the review is positive or negative. I keep track of how many are correct, and use this to determine the accuracy of my results.

Preprocessing: How you cleaned and how you represent a document (features)

For the preprocessing, I implemented a bag-of-words model. I represented the documents in the two dictionaries, either positive or negative. By using a dictionary, the look-up time would be O(1). Within the dictionary, I stored a string, which would be the unique word, along with another dictionary, which held a counter to count how many times the word was in the document, along with the probability variable. The counter was incremented as we parsed through each line, and the probability was calculated at the end. In order to deal with underflow, rather than multiplying the probabilities together, I added the log of the probabilities together, using numpy.log10(). For smoothing, my equation was (number of times a word shows up + alpha), divided by the (number of words in the positive or negative dictionary + (alpha * the total number of words in the text document)). Since many people did not find success dealing with stop-word removal from piazza posts, I did not implement this. As a result, my accuracy on the public testing data was 84.06%.

Model Building: How you train the Naive Bayes Classifier

With the training data, when I first parse through the text file, I first check the label, to see whether it is a positive review or negative review. I then insert each word into the correct dictionary, and keep count of how many times a word shows up. Once this is finished, I calculate the probability of a word showing up in a positive or negative review by adding the (number of times it shows up + alpha), then divide that by the (number of words in the positive or negative dictionary + (alpha * the total number of words in the text document)). In order to deal with underflow, rather than multiplying the probabilities together, I added the log of the probabilities together, using numpy.log10(). For smoothing, my equation was (number of times a word shows up + alpha), divided by the (number of words in the positive or negative

dictionary + (alpha * the total number of words in the text document)). For my alpha, I started from 1 and worked my way to 20. However, the accuracy got lower as I increased the number, so I started decreasing my alpha, until I got 0.01, which gave me my accuracy of 84.06%. This concludes the training part of my code.

Results: Your results on the provided datasets (accuracy, running time). Also give a sample of the 10 most important features for each class (positive or negative)

For the results, the accuracy of running the training data was 91.24%, while the accuracy for the testing data was 84.06%. The running time to train my algorithm was 4.69 seconds, and the running time to label/test the training data and testing data was 36.24 seconds, which was done locally, not on CSIL. The running time for CSIL was different, but I am currently having issues logging in, so I emailed the help desk for UCSB engineering, and so I cannot display the results from that! The top 10 most important features out of all the classes were the words with the highest probability, or the lowest probabilities, as this indicates that either the word is very indicative of its class, or the exact opposite. These words and their probabilities are: from the positive class, nauseating, 9.040479733012054e-07, beautiful, 0. 00021769565152115247, amazing, 0005320884541766464, best, 0014671709101434158, sucked, 2. 159370308367705e-05, badly, 1. 2598200861774511e-05, and from the negative class, bad, 0006863613172822749, , sucks, 0.0002326357518811581, bored, 0. 00027205891864015195, frustrating, 0. 00032995059595290863. These show the words with the highest probability and lowest, showing how that word either reflects their class, or reflects exactly the opposite.

Challenges: The challenges you faced and how you solved them

The challenges I faced were difficult to figure out for a while. My biggest challenge was CSIL crashing on me, and I wasn't allowed to execute or edit any of my files. I had to contact the UCSB help desk for them to restart my account, so I could continue to edit. In terms of challenges in the code, the smoothing function took some time to work through, in the situation of figuring out what goes were, with which variables, and what to set those variables to. There were also a lot of syntax challenges, when it came to importing and using libraries like numpy. I also started calculating accuracy wrong, and took me some time to realize the correct equation to use, as I over-complicated things. It was a slight challenge using a dictionary, so it was sometimes confusing to access the specific variable I wanted for a while, but I figured it out and it was easy after that.

Norman Chung
5550926
normanchung@ucsb.edu

Weaknesses: Weaknesses in your method (you cannot say the method is without flaws) and suggest ways to overcome them.

The weaknesses in my method are as follows. I am unsure whether I am utilizing the smoothing factor to its best, and there may be a better alpha variable that I didn't use. I can overcome this by studying the function more, and understanding how the alpha variable affects the code more in depth. Another weakness is that I should've made functions for the bigger parts of my code, as I learned that it is better for real-world applications, for others to understand my code more easily, even if it all makes sense to me. An obvious weakness is that the implementation does not meet the 85% accuracy standard, as I did not account the best for smoothing, or stop words, and things like that, so it isn't the most accurate when testing.