**MP4: Application of Naive Bayes**

Bum Jun Kim (*bkim102*) || Samuel Hung (*shung5*)

**Part I: Training and Development**

In this assignment, we are asked to train a model to label a set of words as either “ham” or “spam” and then use this model to correctly label sets of words that we have not seen before. For training the data set, we used the following formula to get the posterior probability for both the “spam” class and the “ham” class:

To help us calculate the probabilities of each word in a class, we used the “bag of words” approach, where we disregard the order of the words and only look at the number of occurrences for each distinct word. To do this in Python, we implemented the bags of words using a dictionary, where the key is *word* and the value is the number of occurrences for that word.

Going back to the posterior probability function mentioned above, we use this twice in our code during the development stage — once for “spam” and another for “ham”. It basically sums up the logarithms of the conditional probabilities of *word* given the class and by the end, we should have two values: a final probability of the word being “spam” and another for “ham”. We compare these two values and for the one that is higher, we label our set of words. This process iterates until there are no more sets of words to process.

Breaking down the section , we translated this to be the number of occurrences of *word* in the class divided by the total number of words in the class, repeats included. To prevent zero probabilities, we set the current occurrence value for an unknown word to be equal to the Laplace smoothing factor, which is passed into the *naiveBayes* function from the start. Every occurrence after the first occurrence we increment the count by one.

We found the optimal value for the Laplace smoothing factor to be 0.0001. We tried many different values and noticed a trend — the accuracy of the development labels would increase as we went down from 1 to 0.0001 and it would decrease after we keep decreasing by a factor of ten. Although it’s interesting to note that this “optimized” smoothing factor has adverse effects when running the our model on the training set. We found the optimal smoothing factor for the training set to be the default value, 1.0.

Some optimizations we added were changing every non-numeric character to lowercase and ignoring stop words. However, this seemed to have a negligible effect on the results.

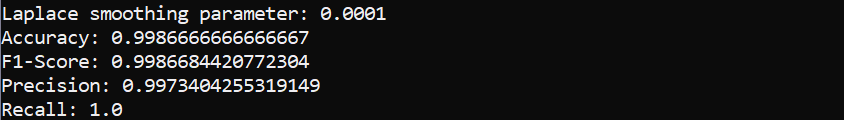
Results for *train\_set* with default smoothing parameter value (1.0):



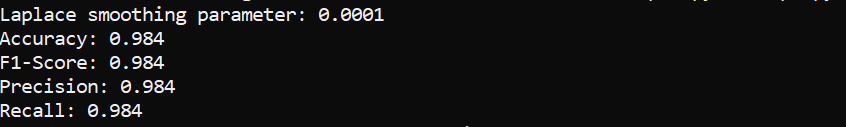
Results for *devel\_set* with default smoothing parameter value (1.0):



Results for *train\_set* with optimal smoothing factor (0.0001):



Results for *devel\_set* with optimal smoothing factor (0.0001):



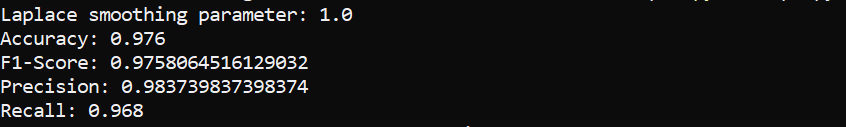
**Part II: Stemming**

Running the code with *--stemming* causes the labeling accuracy to decrease. We did not anticipate this result, but it seems that it occurs due to the trimming of data that could potential distinguish the word as “spam” or “ham”. Stemming a word essentially removes its suffix — this information is used to change the function or meaning of a word. Although stemming reduced our accuracy (by around 1-2%), it could potentially help in the case that the words given in the dataset that share the same prefixes have the same general meaning or function.

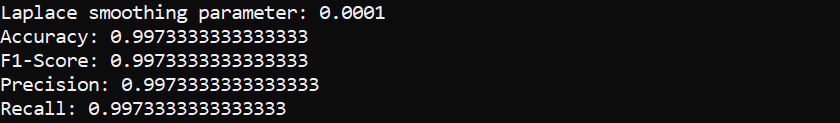
Results for *train\_set* with default smoothing parameter value (1.0) with stemming:



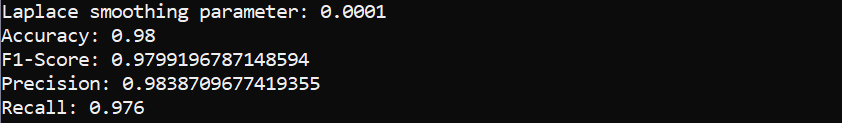
Results for *devel\_set* with default smoothing parameter value (1.0) with stemming:



Results for *train\_set* with optimal smoothing factor (0.0001) with stemming:



Results for *devel\_set* with optimal smoothing factor (0.0001) with stemming:



**Questions answered**

The naive bayes got its name from the fact that it makes logical jumps on how it derived its equations, such as:

p(E1|E2, … En, C) ≅ p( E1|C),

Assuming naively that E1 ~ En are conditionally independent, thus naively assuming that

p(C| E1, E2, … En) ∝ p(C)p(E1|C)p(E2|C) … p(En|C)

Without checking the conditional independence of the variables E1~En

The naive bayes would work best when working with sets with higher conditional independence between each variables, but would work poorly for ones that have variables that have high conditional dependence. Most well known types of problems naive bayes fails is the XOR problems where you are trying to say true when one of the two evidences are true, but not when both of them are true.

An Example where Naive Bayes fails would be is classifying good citizen on his voting activity for a single voting event. One input would be Voted Online, and the other would be Voted In Person. If one of the two(voted online, voted in person) is true, the person should be labeled good citizen. However, if both are true, the person should be labeled bad citizen for doubling his votes. The naive Bayes algorithm in this case would not be able to catch this exception at all since

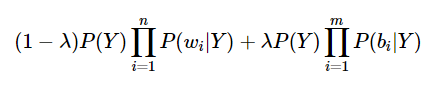
p(Good Citizen| VO, VIP) ∝ p(Good Citizen)p(VO|Good Citizen)p(VIP|Good Citizen)

p(Bad Citizen| VO, VIP) ∝ p(Bad Citizen)p(VO|Bad Citizen)p(VIP|Bad Citizen)

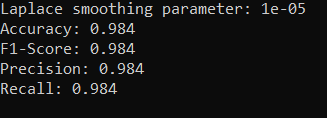
And from this formula, say the Bad Citizen population who makes two votes are smaller than Bad citizen population who does not vote at all, the algorithm would fail labeling the double voting people properly.

**Extra Credit: Bigrams, case insensitive check, stop words**

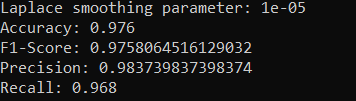
We implemented bigrams by concatenating the string at position index and the string at position *index+1*, and putting the concatenated string into a bag of bigram. Just like how unigrams were treated, using dictionary to store the bigram and number of occurrences on the training set. If the bigram occurs only once, we replace the “1” with the Laplace smoothing value. On the development set, when the bigram contained a stopword in it, we exclude it so it wouldn’t make any effect on the probability. When an unknown bigram showed up, we substituted it’s probability with the smoothing parameter just as we did on the unigram.



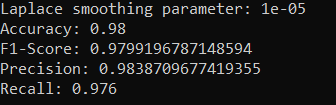
The bigrams scored all(accuracy, F-1 score, Precision, Recall) slightly lower than unigrams.



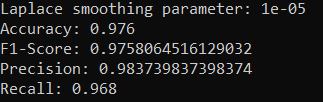
Output with lambda value of 0(unigram weight 100%)



Output with lambda value of 1(bigram weight 100%)



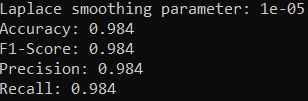
Output with lambda value of 0.1~0.2



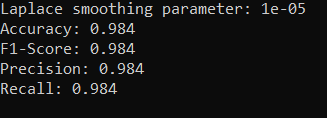
Output with lambda value of 0.3~1.0

As shown above, all lambda values between 0 and 1 with a granularity of 0.1 was tested, but unfortunately the lambda values in between 0 to 1 did not bear any more accurate model than pure unigram model. So purely according to the statistics, the most optimal lambda would be 0. However, since there was only one test set, on a different test set, other percentages might improve the results.

We also implemented a case insensitive bag of words dictionary, using the casefold() string method. In other words, two same words with different capitalization would still be regarded as the same word. However, as shown on the two images below, the new feature did not make any noticeable differences in accuracy.

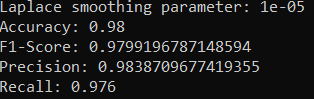


With case insensitive bag of words

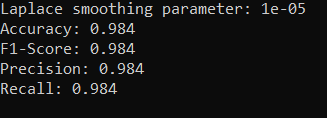


With case sensitive bag of words

Another feature we included was stop word checks. Stop words are words such as “the”, “for” and such words that appears quite often, but its frequency should not be counted in for determining between spam and ham. Stop words does not add any context to the mail, thus provides almost no information helpful for us to determine whether a mail is spam or not. We first tried importing the stop words from the nltk library, but our local python did not support nltk without downloading extra libraries, we just added a list of stop words as a local variable. We checked whether a word or a bigram held the stop words, and if it did, we excluded it from being added to the probability. Although miniscule, excluding the stop words from the probability improved our result as shown below



Without Stop words filter



With Stop words filter