**MULTINOMIAL**

I decided to use what the lecture slides called the "somewhat more subtle version" of smoothing when I added smoothing to prevent over-fitting; this “more subtle version” is where I used an alpha instead of one. My first chosen alpha was 0.5. I also used log probabilities instead of just probabilities to prevent underflow.

My first test run was an accuracy test. I wrote a function to pick the best category for a message feature based on my Multinomial Classifier, then ran it on twenty message features per category. In a test set of 20 categories, this meant I scored 400 features. 378 features were classified correctly, for a total of 93.25% correct classifications.

I then attempted to test different alphas to see how I could improve my accuracy, with these results:

|  |  |
| --- | --- |
| Alpha | Accuracy |
| 0.7 | 92.5% |
| 0.6 | 92.75% |
| 0.5 | 93.25% |
| 0.4 | 93.75% |
| 0.3 | 94.5% |

Unsurprisingly, accuracy improves as I used smaller alphas (and thus closer to over-fitting). I will leave my alpha at 0.5 for now and adjust the alpha when I reach the k-fold section, where changing the alpha is not as explicitly overfitting.

\*Note on the multinomial code: I took “(tab-separated, one per line)” to mean one set-of-twenty-messages-from-the-same-newsgroup per line. Thus, I have twenty lines of twenty tab-separated newsgroup guesses, rather than one newsgroup guess per each of four hundred lines.

\*Additional note on the multinomial code: I used an old CS124 homework assignment as a guide.

**K-FOLD**

The method we used to fold was not as random as it could have been had we used a randomizer; instead, its randomness is dependent on the order of the messages in the MessageIterator. We simply set the code to, for a fold n in 0…k, to put each n documents in the test set and the remaining in train. We could have had a more representative spread by using a stratified method of sorting, but using a random selection felt more realistic. Pseudo-random, really, but still relatively random.

I started the k-fold on the multinomial classifier, without feature selection, and with alpha = 0.5. I got an average accuracy of 84%. I then decided to test new alphas to try to improve this accuracy, yielding this graph:

This trend was the same as the trend before using k-fold; the lower the alpha, the higher the accuracy. However, 0 is too small and leads to over-fitted data and 1 is, according to lecture slides, simplistic; so I will continue to use my original of 0.5 as an alpha, since it is exactly between 0 and 1.

Average accuracy for multinomial without feature selection and alpha 0.5: 82%

Average accuracy for multivariate without feature selection and alpha 0.5: 83%

I am calculating error as % error = 100% - % accurate.

Error for multinomial classifier without feature selection and alpha 0.5: 18%

Error for binomial classifier without feature selection: 17%

**TWCNB**

I started out with simply CNB. I continued to use my original decision of 0.5 as an alpha, as explained above. As explained in the handout, I use the regular Naïve Bayes Multinomial code but changed two variables:  
numOccurrencesWordInDocsOfSameClass became numOccurrencesWordInDocsOfOtherClass

numOfWordsInDocsOfSameClass became numOfWordsInDocsOfOtherClass   
Then returned -1 times the natural log of the probability, to mirror the way the handout was now taking the negative sum of probabilities rather than the positive sum in order to give classes that “this word appears very infrequently in all other classes” a high score instead of a low one. This increased our accuracy to 95.5% instead of 93.25%.

Then I implemented WCNB. As Piazzza listed many difficulties using prior probabilities, I left them out. I simply implemented a trainWCNB() method that for every class, went through the vocabulary and summed the probabilities (standard NB, not CNB) for each vocabulary word. I then decided that, to integrate CNB into the weight normalization, I would treat wi,c = CNBi,c / the sum over the corpus for that class. This method gave me an accuracy of 95%, almost the same as CNB. I decided to experiment.

|  |  |  |
| --- | --- | --- |
| Accuracy by method | Classify used NB | Classify used CNB |
| trainWCNB used NB | <did not test: need CMB > | 95% |
| trainWCNB used CNB | 93% | 95.6% |

Since using CNB to for both of the following thetas:

Wi, c = log (Θ i, c)

Σk|Θk,c|Led to the highest accuracy, I’ll use that.

For the transform, I started by setting the frequency f’i = log(1 + fi). This is the frequency used when calculating theta, that is, for this equation, anything starting “number of”:

(number of occurrences of word in docs of other classes) + alpha

(number of words in docs of other classes) + alpha \* size of vocabulary of corpus

This increased my accuracy from 95.6% to 96.25%. This is all done with an alpha of 0.5.

**EXPERIMENTING**

The first test I decided to try was up-weighting subjects. Subjects seemed a reasonable zone to weight, as the subject of an email is intended as a summary of its contents, and are therefore more representative of an email’s class. I did the upweighting by adjusting my code (k=10 fold, TWCNB, 0.5 alpha, no feature selection) to count every word in the subject twice (in every location: numWords, numWords in class, etc). Without upweighting I scored 85% and with upweighting I scored 86%: a small but measurable increase.

I next commented out the line in parse() in MessageFeatures that stems words, to see how stemming affects accuracy. This also generated an accuracy of 85% but was significantly slower than the unstemmed version because of the sharp increase in vocabulary size. Though stemming may hypothetically cause loss of accuracy (if, for example, very important words that denote different classes stem to the same root), here it maintained its original accuracy (and could also, if the previously mentioned important words denoted the same class, improve accuracy- it is data dependent). Regardless, its resultant speed increase is likely significant enough to warrant implementation except in exceptional cases.

I then commented out the line in the same method as stemming which converted each line to lower case. This had the same basic result as stemming; it had no visible effect on the accuracy which remained 85%, but it significantly increased the time needed to run the program since the vocabulary size increased.