**MULTIVARIATE (BERNOULLI)**

I implemented the multivariate naïve bayes classifier with plus-1 smoothing. It performs the operation in 3 distinct steps. The first is the preparation for the classifier – this counts the number of docs in each class a word is part of. For the words of the message feature, I used the subject and the body (I weighted them the same, but optimizations could be done to weight them differently). After we had the corpus counts, we trained the binomial classifier. This took a count of the number of terms in each class and the number of documents in each class. Then for the entire vocabulary it got the multivatiate probability.

Fundamentally the multivariate probability is (# of documents in a class a term shows up in/# docs in the class). You hit a slight issue here when the word in the vocabulary doesn’t exist in the class – the probability is zero – which give the lack of having a word infinite weight when classifying. So I set up smoothing to prevent this. The smoothing I added 1 to the numerator and the number of terms in the vocabulary to the denominator. The initially perceived issue with doing the smoothing this way is if there was a word was not in the vocabulary, then you aren’t smoothing that value. But if it is not at all in the vocabulary, then it won’t have a value for any of the classes, and thus might as well be skipped when presented in the classification step – rendering words not taken into account in the vocabulary a non-issue.

For the final step is classification. For MessageFeature, we wanted to guess a class that it should be, so I got an initial probability of the class (the number of docs in the category/total number of docs in corpus). For all the probabilities, I took the log of them, so that instead of multiplying small probabilities (where we quickly would hit underflow issues) we can add larger numbers - which would take much longer to overflow. I kept a class probability tally that corresponded to the trained probability previously – and for each word in the MessageFeature, I added the log probability of that word being in that class to the total log probability. Because the values went from 0 to 1 (which translates to –infinity to 0 respectively), instead of looking for the largest probability, I looked for the largest summed value (which would be negative). The class that corresponded to this probability is deemed the one that is the most likely for that MessageFeature. When all was said and done, I was achieving 91.5% accuracy on the corpus we were provided. Note there was a bit of overfitting since for this version, I did no kfold – so the features being tested are actually part of the corpus.

**FEATURE SELECTION**

For feature selection, I was building off of the existing binomial code. There are a number of ways to implement feature selection, but for this exercise, I did a chi squared feature selection (where I limited the number of values to 300 features per class). The main change to the binomial methods that I needed to make was actually creating the feature list. So after doing the initial binomial prep, I added a method which created a feature set. This essentially went through all the instances of the number of documents in each class of each term and performed a chi-squared on it which looked like this:

*N11 = number of documents of the class the term shows up in*

*N01 = number of documents of the class the term doesn't show up in*

*N10 = the number of documents the term shows up in not of the class*

*N00 = number of documents the term does not show up in that are not in the class*

*N=N11+N01+N10+N))*

*Which translate programmatically to:*

*A=N11 = classFeatureList[word]*

*C=N01 = classFeatureList.size()-N11*

*B=N10 = totalFeatureList[class]-classFeatureList[word]*

*D=N00 = totalFeatureList.size()-classFeatureList.size()-(totalFeatureList[class]+classFeatureList[word])-=totalFeatureList.size()-(N10+N11)-N10*

*N=totalFeatureList.size()*

Then the chisquared operation was the one outlined in the lecture notes:

*N\*(A\*D-C\*B)\*(A\*D-C\*B)/((A+C)\*(B+D)\*(A+B)\*(C+D))*

I did a chi squared value for each term in each class – then I went through each class, and added the top K (in this case it was 300) features to a specific class feature set. Once I had this feature set, the binomial code only required very minimal changes. For the training – instead of training on the entire vocabulary, I only trained on the features (including smoothing). The for classification, I only added values of log probabilities for the union of the MessageFeature vocabulary and the feature-set (as opposed to all the word in the MessageFeature). When trained on this set (again without kfolding), we achieved an 88.75% accuracy.

This is less than the binomial – so I looked at the documents that it classified. The main thing that stood out was the terrible accuracy on documents 18 and 19. For example, for document 19, the first 20 messages get classified as 19,19,15,15,19,16,15,15,19,15,19,15,19,0,15,19,16,15,19,19. That’s over 50% wrong (9/20 are correct). I looked at the document sizes, and noticed something interesting.

class0:799

class1:973

class2:985

class3:982

class4:961

class5:980

class6:972

class7:990

class8:994

class9:994

class10:999

class11:991

class12:981

class13:990

class14:987

class15:997

class16:910

class17:940

class18:775

class19:628

Classes 0, 18, and 19 are significantly smaller than the other classes – which plays a large factor in class 18 and 19’s error (though surprisingly, document 0 seems mostly right). The problem with this is that the binomial itself has a significant amount of trouble on document 19 as well. So for now I’ll chalk this up to slight overfitting on the binomial portion which makes the extra features good discerners since we are testing on trained documents

**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%

I, also, want to consider is feature selection with multinomial and binomial. I implemented chi2 feature selection previously, but I had to implement it for the multinomial as well for the sake of comparison.

Error for multinomial classifier with feature selection and alpha 0.5: 27%

Error for binomial classifier with feature selection: 19%

So I still have the issue of discrepancy between feature selection and non-feature selected for the binomial – though it is a slight discrepancy. The interesting thing to note here is he the multinomial discrepancy is fairly large. That is more likely do to programmer error (that couldn’t be found in time), since it should not deviate that significantly.

The other thing I wanted to look at is how modifying the number of folds affected the accuracy. A Smaller number of folds means that the training corpus is smaller and the testing corpus is larger. So I tested multiple K values, and returned the average error for binomial and multinomial with and without feature selection and I got a table that looked like this

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| K | Binomial Without Features | Binomial With Features | Multinomial Without Features | Multinomial With Features |
| 2 | 23 | 19 | 19 | 29 |
| 3 | 20 | 20 | 17 | 28 |
| 4 | 21 | 20 | 18 | 27 |
| 5 | 20 | 19 | 16 | 27 |
| 10 | 20 | 19 | 16 | 27 |
| 20 | 18 | 18 | 15 | 27 |
| 30 | 19 | 18 | 15 | 27 |

For the small values of K, the error went way higher, which makes sense – since there much less data to train on. It seems to be the case on this set, but somewhere around the square root of the average size of the a class seems about right for the size of K (so in this case around 10). That way you reduce the diminishing returns a high number of folds (thus time it takes to process) and the obviously debilitating outcome for the error when the size of the training set is significantly reduced

**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.