Fakharyar Khan
February 25th, 2023
Professor Sable
Natural Language Processing

Project #1: Text Categorization

Instructions on Running Program

I developed my system in version 12.3.1 of the macOS Monterey operating system. To run the program you need python 3.9.12 or a later version installed. You will also need to install the following libraries in order to run the program: pandas, numpy, sklearn, and nltk. You can install these libraries by entering the following command into your terminal: pip3 install <package name>. Finally, to run the program, place the program in the same directory that the training and testing documents are in. After that, go onto the terminal and cd into that directory and enter the following command: python3 Khan_F_ECE_467_Project1.py. The program will prompt you to enter in the path to the file that contains the list of training documents as well as the path to the file containing the list of testing documents. After you enter those paths in, the program will begin training on the training set that you have provided. Once it finishes training[1], it will prompt you for the name of the output file that you would like the classifications to be written in.

Implementation Details and Performance Evaluation

For my program, I used the Naive Bayes algorithm to categorize the documents. I used the NLTK library to tokenize the documents and also used it to normalize the tokens so that all of the words were lowercase and converted to their base forms. The main benefit that this has is that it increases the validity of our assumption that the probability of a word appearing in a document is independent of that of another word appearing in the same document since before if you found a word like rob then you were likely to also find words like robs, robbed, or Robberies in the same document. As a result, the Naive Bayes estimate better reflects the actual probability of a document being from a certain category so in theory it should perform better. Normalizing the tokens increased the accuracy of the model by around .5% and while that wasn't as much of an improvement as I was hoping for, the increase was still worth the extra computation spent in normalizing the text. I also tried stemming instead of lemmatization. I didn't think that this would make the model perform better than if I had used lemmatization since stemming can occasionally give the same stem for two words that have very different meanings. For example if you used the porter stemming that the NLTK library provides, the word politician and polite would both have the same stem: polit even though the two words aren't related to one another. With lemmatization, the model did roughly .8% better than with stemming so I decided to stick with lemmatization.

[1] This can take up to 10 minutes for both the first and third projects and around 1 minute for the second.

I also removed stop words and punctuation from the tokens since there didn't seem to be any benefit in including them and in fact, it's possible that they were hurting the model by adding noise to the probabilities calculated. This didn't really noticeably increase the overall accuracy of the model but it did make the model faster as removing all of the stop words and punctuation made the list of tokens for each document significantly faster.

In order to ensure that the model wouldn't assign a probability of 0 to a document with a word that it has never encountered in the training set, I decided to use Laplace smoothing which is where you add a small constant, the smoothing parameter, to the frequency count of each term in each category. There's also a general form of Laplace smoothing where you add a term to the denominator so that the probabilities add to 1 but I found that that made my model perform slightly worse so I decided to stick with just adding a small constant. Originally I had allocated around 5-10% of the training set to a validation set which I would use to find the optimal value for the smoothing parameter. However, I found that this would lead to overfitting as the validation set was too small to be representative of the training set. As a result, the parameter chosen ended up not being the best one for the testing set. Instead, I decided to use 4-fold cross validation so as to prevent overfitting and get a more accurate estimate on the accuracy that my model will have on the testing set. Unfortunately, this did increase the training time for my model from around 1 minute to roughly 3 minutes. But it was worth it since it increased the model's performance by around 1% giving it an 89% overall accuracy.

When I looked at the confusion matrix for my model, I found that it did really well in classifying all of the categories except the other category. While the other categories had an F1 score greater than 0.87, the other category had a 0.63 F1 score. It makes sense that the model has such a hard time classifying documents in the other category because that category doesn't really have a singular topic like the other categories do. As a result, I think the model failed to find any "signature" words that the other category uses. In fact, because these news categories are somewhat topically related to one another and are likely to use the same words (but probably not at the same frequency), a document in a grab bag category like the other category would probably be frequently incorrectly identified as being a part of another category. This explains why the precision score for the other category was much lower than the rest at around 62%. But even worse is that the recall score for that category is also not so good at around 64%. This means that the other category is also dragging down the precision scores of the rest of the categories too.

To measure the performance of the model on the other corpuses, I kept 80% of their training sets in training and used the remaining 20% as a testing set. For corpus 2, the model took around 15 seconds to completely train and classify the documents in the test set and achieved an overall accuracy of 87.7%. The main issue that the model had with this corpus was that it frequently identified areas that were actually inside as outside which caused the precision score for the inside category to be around 78%. Interestingly enough it achieved a very high precision of 92% on the outside category. I believe that this likely happened because in corpus 2 around 70% of the documents are from the outside category so it probably didn't have enough data to learn the inside category as well as it did with the outside category. And using only 80% of the training set likely only made this worse. As a result, the model should perform better on the actual testing set since it will have access to the entire training set.

Finally for corpus 3, it took around 5 minutes for the program to finish and the model was able to obtain an accuracy of 92.1%. Just as with corpus 2, this corpus suffered from an insufficient amount of data for one of its categories. In this case, there were 6 categories and the entertainment category only made up 4.6% of all documents in the corpus and that there were only 14 entertainment documents in the test set. Because of the small amount of data the model had on entertainment documents, the recall score was pretty low at 71%. Despite this, in general the model did very well on this corpus.

I found that with this corpus, counting the number of times a word appears in a document instead of just if the word occurs in the document significantly raised the accuracy of the model by around 4 percentage points. Intuitively this does make sense because when we only look at the number of documents in which a word occurs, we lose a lot of the discriminative power that comes from observing multiple occurrences of a word in one document. For example, both World News and US News might have the word US in their documents but we're much more likely to see that word in the US News and if we only look at the number of documents that the word US occurs, that insight might not be as apparent to the model. At the same time, counting the number of occurrences of a word in a document means that some of our probability values invalid as they will end up being greater than 1. This would take away some of the interpretability of our model since it's no longer calculating probabilities and instead some quantity that seems to correlate with the actual probability.

Below is a table summarizing the performance of the model on the three corpuses.

Corpus	Execution Time	Overall Accuracy
Corpus #1	~3 minutes	89.4%
Corpus #2	~15 seconds	87.7%
Corpus #3	~5 minutes	92.1%

Attempts at Improving Performance (3/13/23)

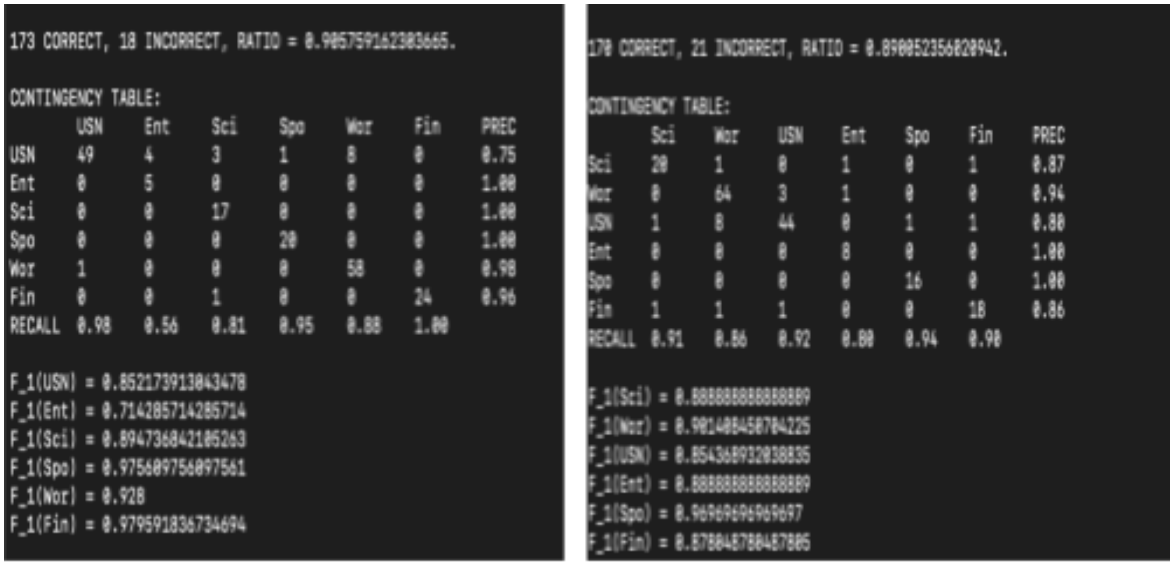
After my first submission I came up with a couple of ideas on how to improve the accuracy of my classifier but none of them seemed to really work out. The first thing I tried was adding the POS tags to the vocabulary of the classifier. However, knowing the frequency of the individual POS tags in a document didn't seem like it would be very useful to me. So instead I took trigrams of the list of tags and made those triplets a part of my vocabulary. This ended up hurting the performance of my classifier on all three corpi so I decided to not use it.

After that, I had an idea that instead of iterating over a predetermined list of possible values to tune the smoothing parameter, I could use a genetic algorithm to search for the best possible parameter value. For the first generation, I generated 5 values randomly chosen from 0 to 1. Then for each iteration of the algorithm I increased the population by a factor of 1.5 by choosing the most fit (the accuracy of the model when that parameter was used) individual out of 5 randomly selected members of the population to create a mutated copy of itself. In order to

make sure that the algorithm didn't become too computationally expensive, I would eliminate any individual that had lived for more than 2 generations and would stop the algorithm if the best score it received didn't change after three generations. Even with this, the algorithm wasn't very efficient and there were a lot of improvements I could have made and I would have done it if the results I had received made it worth making those improvements. However, it looks like what ended up happening was that it would do incredibly well when doing cross validation and then get around 85% accuracy on the test set. The craziest example of this was with the second corpus. On the training set the model found that the best parameter value was something like 0.00078… and with it it scored a 99.5% accuracy on average across all folds (which would be one misclassification for every 200 documents) which is insane. When I saw that I thought there had to be something wrong with how I was calculating the error but for that version of the program I hadn't touched the part of the code that computes the error. Additionally it was giving more reasonable error rates for the other corpi so I think the model just had an easier time overfitting on the second corpus because it only had two classes. I could have tried to do leave one out cross validation to really reduce the chance of overfitting but that would have meant an incredibly long training time. Using a genetic algorithm to tune the smoothing parameter also made my training time much larger and that would make it very difficult to make changes to the algorithm and see if it improved the classifier's accuracy on corpus 2 and 3 so I decided to not invest time to see if I can get it to work.

The final thing I tried out was addressing the class imbalance in the corpuses. In all of the corpuses, there's always one class where there aren't that many data points in the training set. As a result, the classifier tends to classify that class incorrectly and this significantly lowers the overall accuracy of the classifier. My first approach in solving this issue was providing greater weight to errors on documents that are less represented in the class. So a misclassification of a document that's in a category that makes up say 10% of the corpus would lead to a penalty of 1- 0.1. However, this method didn't really improve results and only slightly increased the recall score for the smallest group.

My other approach was assigning words that haven't been encountered in the training set randomly to some category with probability 1-p where p is the probability that the document is that category. The reason for this is that since we have fewer documents of that category, the word is more likely to come from the categories that we don't have many documents for. This however ended up being a large



173 CORRECT, 18 INCORRECT, RATIO = 0.9057591623036649.

CONTINGENCY TABLE:

	USN	Ent	Sci	Spo	Wor	Fin	PREC
USN	49	4	3	1	8	0	0.75
Ent	0	5	0	0	0	0	1.00
Sci	0	0	17	0	0	0	1.00
Spo	0	0	0	20	0	0	1.00
Wor	1	0	0	0	58	0	0.98
Fin	0	0	1	0	0	24	0.96
RECALL	0.98	0.56	0.81	0.95	0.88	1.00	

F_1(USN) = 0.852173913043478
F_1(Ent) = 0.714285714285714
F_1(Sci) = 0.894736842105263
F_1(Spo) = 0.975609756097561
F_1(Wor) = 0.928
F_1(Fin) = 0.979591836734694

170 CORRECT, 21 INCORRECT, RATIO = 0.8900523560209424.

CONTINGENCY TABLE:

	Sci	Wor	USN	Ent	Spo	Fin	PREC
Sci	20	1	0	1	0	1	0.87
Wor	0	64	3	1	0	0	0.94
USN	1	8	44	0	1	1	0.80
Ent	0	0	0	8	0	0	1.00
Spo	0	0	0	0	16	0	1.00
Fin	1	1	1	0	0	18	0.86
RECALL	0.91	0.86	0.92	0.80	0.94	0.90	

F_1(Sci) = 0.8888888888888889
F_1(Wor) = 0.9014084507042254
F_1(USN) = 0.8543689320388835
F_1(Ent) = 0.8888888888888889
F_1(Spo) = 0.9696969696969697
F_1(Fin) = 0.8780487804878049

Confusion Matrix for Classifier Before and After Correcting for Unbalanced Classes

overcorrection and while it did increase the recall of minority categories significantly, it was at the expense of the recall and precision scores of the other categories as can be seen above. To remedy this, I made the amount I added when there's a new word in the test set a parameter ranging from 0 to 1. Unfortunately even with this, the classifier wasn't able to increase its accuracy from before for any of the corpuses so I decided to abandon the approach.

One thing that I did find was that applying the full formula for laplace smoothing where you modify the denominator as well as the numerator of the conditional probability helped increase the performance of the classifier on the second corpus by a percent and having the smoothing parameter range from 0.001 to 0.01 also improved performance in the second corpus.

Below is a table summarizing the performance of my classifier on each of the corpuses. There were a couple of redundant calculations being made in my code and once I removed those, the execution time decreased somewhat but not significantly.

Corpus	Execution Time	Overall Accuracy
Corpus #1	~3 minutes	89.39%
Corpus #2	~15 seconds	86.2%
Corpus #3	~5 minutes	90.57%

Finally in order to more reliably predict the results I would get on the testing sets for corpuses 2 and 3, I thought of using cross validation to split the training set into multiple folds and then take the average performance across all folds. However, because I also tune my model by using cross validation, this would have ended up being really computationally expensive so I didn't do this. But if the results for this pre-submission end up being below what I'm expecting I might try doing this as well as increasing the number of folds used in tuning the model as a last ditch effort.

Final Update (3/20/23)

After playing around with different tokenizers and lemmatizers I found that surprisingly for corpus 1, if no lemmatizer is used, the accuracy on the test set increases to 90.29%. Additionally I found that using the porter stemmer instead of lemmatization drastically increased the performance of the classifier on corpus 3 to an average accuracy of roughly 91%. I assumed that lemmatization was always preferable to stemming but it looks like the way stemming was reducing these words made it so that more words in the same category got mapped to the same root. It's possible that this is more likely to occur in stemming than in lemmatization as lemmatization tries to be more careful not to lose the meaning of the word when reducing it. Also on corpus 3 I found that using the tweet tokenizer, a tokenizer used for handling tweets, gave fairly good performance on this corpus which is pretty interesting.

Finally, I cranked up the number of folds I used in my cross validation for tuning to 8 folds so that I could avoid overfitting on the training set and for corpus 3, I used the full form of

laplace smoothing but used a version that Professor Sable showed me where you add the same constant, the smoothing parameter, to both the numerator and denominator.

Ideally what I would have liked to do was treat the type of tokenizer used, using stemming or lemmatization or neither, and the inclusion of other normalization techniques as parameters in my model. Then I could use cross validation to choose the best set of parameters for each corpus so that I wouldn't have to do it manually. Unfortunately this would have been fairly computationally expensive as you would have to try all of the combinations of normalization techniques since these are discrete parameters. If I had thought of the idea a week earlier it might have been possible to do it.