Jiacong Wu

COM3110 Sentiment Analysis Assignment Report

In this assignment, we are tasked with implementing a Naive Bayes model from scratch, with the aim of predicting movie review sentiments. The movie review datasets contain data samples, each a series of tab separated values: a sentence ID, a movie review sentence, which has already been tokenized, and a sentiment class label from 0 (negative) to 4 (positive). This report details the implementation details, as well as the analysis of results collected from the model, under different conditions.

Load and preprocess data

\`Utilities.load_and_preprocess_data\` function returns a tuple of 3 lists, one for the sentence IDs of data samples, one for the movie review sentences, which are preprocessed in \`Utilities.preprocess_sentence\` by lowercasing every token. If we want to choose our own features instead of using all the tokens in the sentence as features, then it will undergo further processing, where an English stopwords list from the NLTK package is used to remove all words in the sentence that occurs in that list. We also define our own stop list of tokens which we want to get rid of, which is not in the NLTK stopwords list, from manually examining the dataset. \`Utilities.load_and_preprocess_data\` returns a list in the tuple for the sentiment class labels of the movie reviews, needed for supervised training.

Naive Bayes class

The \`NaiveBayes\` class defined in \`naive_bayes.py\` defines our Naive Bayes model for sentiment analysis. The model relies on equation 1 to make classification decisions

$$s^* = \underset{s_i}{\operatorname{argmax}}\ p(s_i|T) = \underset{s_i}{\operatorname{argmax}}\ p(T|s_i)p(s_i) \qquad (1)$$

s_i takes one each of the possible sentiment classes, s^* represents the predicted sentiment of some given text T, which is the most probable class out of all the classes, i.e. the class with the highest posterior $p(s_i|T)$. The \`NaiveBayes\` class' \`fit\` function is supplied the training data to obtain the following parameters for the model, needed for the equation above to make classification decisions later

- priors which stores prior probabilities $p(s_i)$ for all the sentiment classes, which is computed simply by relative frequency in equation 2, where J is the number of different classes, so the denominator is basically the total number of training labels

$$p(s_i) = \frac{count(s_i)}{\sum_{j=0}^{J} count(s_j)} \qquad (2)$$

- likelihoods which is a dictionary mapping each of the sentiment class labels, to words/tokens, to their likelihood values given the sentiment class, with Laplace smoothing applied as in equation 3, where $\sum_f count(t_f, s_i)$ computes the sum of all word/feature occurrences for a class and $|V|$ is the number of distinct features

$$p(t_j, s_i) = \frac{count(t_j, s_i) + 1}{(\sum_f count(t_f, s_i)) + |V|} \qquad (3)$$

  - we first store the number of occurrences for each word in each of the classes instead of the likelihood to calculate the likelihood later
  - likelihood of the entire sentence for a class, is then the product of the likelihoods of all the individual words/chosen features in the sentence for the class, given in equation 4, where N is the number of words/tokens chosen from the sentence to use as features

$$p(T|s_i) = \prod_{j=1}^{N} p(t_j|s_i) \qquad (4)$$

- number of distinct features which is the vocabulary size of the training data, needed to apply laplace smoothing, so words that don't appear in the training data will not significantly interfere with calculations by multiplying by a 0 likelihood value

The `NaiveBayes.predict` can be called to take in a list of unseen data samples, and outputs a list of prediction labels for them. It does so using equation 1 and the internal model parameters obtained previously using `fit` from the training data.

Evaluation

For evaluation, in the `Utilities.evaluate_performance` function, we take the average of the F1 score calculated for each class, each calculated with equation 5 where TP is the number of correctly classified labels of the current class being evaluated, FP is the count of instances where the other classes are predicted as being the current class and FN is the count of the current class being misclassified as belonging to other classes.

$$F1 = \frac{2 * TP}{2 * TP + FP + FN} \qquad (5)$$

Results

Many methods of feature selection methods have been tried, and compared against using all words as features, the results were as follows in the table below. `Utilties.apply_stopwords` uses a stopwords list downloaded from the NLTK library, which contains a list of non-content words to remove from the features. 'not' and 'nor' are kept as they will be helpful for the task of sentiment analysis. `Utilities.binarization` takes a sentence and removes multiple occurrences of each word in a sentence, essentially taking the set of a sentence as features to use. `Utilities.apply_negation` simply negates words that come after words like 'not' and 'nor' up to some punctuation, 'NOT_' followed by the word is used as a feature instead.

Macro F1 scores		-classes	
		5	3
-features	all_words	0.288	0.473
	NLTK stoplist	0.319	0.501
	Negation	0.313	0.487
	Binarization	0.296	0.478
	Neg+Bin	0.316	0.490
	NLTK+Neg	0.326	0.506
	NLTK+Bin	0.325	0.501
	NLTK+Bin+Neg	0.323	0.507

All of the techniques have slightly improved the macro F1 score compared to using all words as features, with using a NLTK stoplist being the most effective, individually, on average, with binarization the least effective, which makes sense as there is nothing to tune with this technique unlike the other two where further selection rules can be introduced. Combining different techniques, I have managed to further increase the score, with the NLTK+Negation combination being the most effective on average. Merging somewhat positive/negative sentiments into the positive/negative classes, reduced the number of labels which helped reduce classification uncertainty between similar sentiments, thus leading to significant improvements in the F1 scores.