CSCE-638, Programming Assignment #2

Sentiment Analysis

Li-Hen Chen 928003907

0. Package I used.

Skilearn:

Scikit-learn s a free software machine learning library for the Python programming language. It features various classification, regression and clustering algorithms including support vector machines, random forests, gradient boosting, k-means and DBSCAN, and is designed to interoperate with the Python numerical and scientific libraries NumPy and SciPy. I used text.CountVectorizer function in the package to turn the words into vectors.

Numpy:

Numpy is a library for the Python programming language, adding support for large, multi-dimensional arrays and matrices, along with a large collection of high-level mathematical functions to operate on these arrays.

1. How to compile and run your code.

We program the code in Python3 and execute the program under the same directory of the NaiveBayes.py file and Perceptron.py.

In the terminal

- For NaiveBayes.py -f (whether to enable "Filter stop words" option) -b (whether to enable "binarized version of the multinomial Naive Bayes classifier")

  python NaiveBayes.py ../data/imdb1
  OR
  python NaiveBayes.py -f ../data/imdb1
  OR
  python NaiveBayes.py -b ../data/imdb1

- For Perceptron.py iteration_num
  python Perceptron.py ../data/imdb1/ 1 (run for 1 iteration)

2. Results and Analysis

2.1 Naive Bayes

Naïve Bayes is a simple("naïve") classification method based on "Bayes rule" which relies on very simple representation of document – Bag of words. For a document d and a class c, the equation can be denoted by:

$$P(c \mid d) = \frac{P(d \mid c)P(c)}{P(d)}$$

In this assignment, I adopted "log space " in order to prevent underflow because multiplying lots of probabilities can result in floating-point underflow. Since log(xy) = log(x) + log(y), it's better to sum logs of probabilities instead of multiplying probabilities.

$$c_{NB} = \underset{c_j \in C}{\operatorname{argmax}} \log P(c_j) + \sum_{i \in positions} \log P(x_i \mid c_j)$$

Num of folds	Naive	Naïve with stop words	Binary Naive	Binary Naïve with stop words
1	76.5%	76.5%	80.5%	80.0%
2	85.0%	82.5%	84.0%	81.0%
3	83.5%	81.5%	83.5%	85.0%
4	82.5%	83.0%	82.5%	80.5%
5	81.5%	79.5%	83.5%	80.5%
6	82.0%	83.0%	82.5%	82.5%
7	83.5%	83.5%	84.5%	84.0%
8	82.5%	83.5%	83.5%	83.0%
9	75.5%	76.0%	79.0%	77.0%
10	84.0%	82.0%	85.5%	84.0%
Average accrucay	81.65%	81.10%	82.90%	81.75%

2.2 Perceptron

In this assignment, the "Perceptron" algorithm can be denoted as below pseudo code. I can tell from the results that the more iterations, the more accuracy the results is. However, inevitably the training time also increases when you apply more iterations. In this case, 100 iterations took me 10 minutes to train on the simple case.

```
Initalize weight vector w = 0
Loop for K iterations
    Loop For all training examples x_i
        if sign(w * x_i) != y_i
            w += (y_i - sign(w * x_i)) * x_i
```

Beside of training and accuracy of this part, what made me curious most is that we used a nice trick. Instead of using parameters by the end of training to classify test examples, use the average of the parameters after updating with each data instance in each iteration. The reason can be described below. It can be shown that, if possible, the perceptron algorithm will eventually converge to a setting of parameter values that correctly classifies the entire data set. Unfortunately, this often leads to poor generalization. Improved generalization ability is available by using weight averaging. Weight averaging is accomplished by modifying the standard perceptron algorithm so that the final weights returned are the average of all weight vectors encountered during the algorithm.

Num of folds	**1 Iteration**	**10 Iterations**	**50 Iterations**	**100 Iterations**
1	58.0%	78.0%	83.5%	81.0%
2	57.0%	70.5%	81.0%	83.0%
3	69.5%	84.0%	84.0%	84.5%
4	76.0%	75.0%	86.0%	86.5%
5	69.5%	86.0%	83.5%	82.5%
6	72.0%	86.5%	85.5%	86.0%
7	54.0%	81.0%	85.5%	86.0%
8	77.0%	81.0%	83.5%	84.5%
9	61.0%	82.5%	82.0%	81.5%
10	71.5%	78.0%	85.5%	87.0%
Average accrucay	66.5%	77.9%	84.0%	84.3%

2.3 Extra Credit

N-grams of texts are extensively used in text mining and natural language processing tasks. They are basically a set of co-occuring words within a given window and when computing the n-grams you typically move one word forward (although you can move X words forward in more advanced scenarios).

We only take one word in part 1 and part 2, which means we consider one word each time. But we know multiple words sometimes can have specific meaning. If we just consider on word each time may not be enough to catch all the meanings in the sentence. As a result, we can count the neighbor words at the same time in order to record all the meanings. Although this approach indeed consumes more memory and space, it enhances the performance of our model. The result was improved **77.9% to 80.1%** in "Perceptron" when we did 10 iterations. Also The result was improved **81.65% to 82.5%** in basic "Naïve Bayes".

Beside the N-gram approach, there are lots of features to improve "Sentimental Analysis" task like TD-IF and pre-trained word embedding models (GloVe, word2vec). I don't have time to try all of the approaches, but maybe I can test these approaches on the final project.

3. Any known bugs, problems, or limitations of your program

The accuracy of different folds varied a lot. I am not sure whether it is caused by the dataset or due to different algorithms. Maybe we can solve the problem by shuffling the whole dataset or collecting more data to prevent the bias in the dataset.