Project #1: Text Classification
CS 7650 Natural Language Processing

Arvind Krishnaa Jagannathan
GT ID: 902891874

January 24, 2013

General Details and Project Structure

The tarball submitted online has the following structure:

1. A directory called *src* which contains the code for the classifiers and for the plotting functionality.
2. A directory called *Response* which contains the response files of my "main" classifier and "special" classifier. My best "main" classifier was the *Naive Bayes*, with a smoothing parameter of $\alpha = 10^{-3}$. I tried to improve on the Naive Bayes Classifier according to the paper [1] mentioned in the project specification. My best classifier from that was the *Complement Naive Bayes* classifier, with the same smoothing parameter as above. (More details in following sections)
3. A directory called *generated_files* which contain files where I have dumped the data produced by the classifiers. Some of the files in that directory are used to plot the data, and some are used to just to verify the metrics.
4. Directories *train*, *dev* and *test* contain the training, development and test data respectively.
5. The files *train.key*, *dev.key* and *scorer.py* are used to obtain the accuracy metrics of the training and development data.
6. The file *sentiment-vocab.tff* is the sentiment vocabulary [2]
7. The directory *Report* contains the tex generated files for this Report. The file *report.pdf* is available at the root of the tarball.

Please consider this part of the report as a README for the project hierarchy.

1 Data Processing

I got 11,083 distinct tokens, some of them being nonsensical such as aat, ee and so on. These may be words misspelt or represent something specific to a domain. I have implemented it as a *defaultdict* in Python (from collections.defaultdict).

2 Word Lists

Deliverable 1

The Python file which does the rule based classification is *RuleBasedClassifier.py*. The accuracy of the Rule-Based classifier I got was 22.55%. The response file for this deliverable is *generated_files/rule_response*. Here is the resulting confusion matrix after running the *scorer* script on the response file.

```
3 classes in key: set(['NEG', 'OBJ', 'POS'])
2 classes in response: set(['NEG', 'POS'])
confusion matrix
key     NEG     POS
NEG     19      18
OBJ     102     79
POS     14      43
----------------
accuracy: 0.2255 = 62/275
```

Deliverable 2

I performed the trial-and-error on the training data, and found that if I set the tuning parameter (absolute value of the difference between the number of positive and negative words) as 6, I get the highest accuracy (I tried values from 1 to 10). I ran the "Tweaked Classifier" on the development data (Python file *TweakedRuleBasedClassifier_dev.py*) with this value of the tuning parameter and I got an accuracy rate of 65.45%. The response file for this deliverable is *generated_files/tweaked_response_dev6* and here is the confusion matrix after running the *scorer* script on the response file.

```
3 classes in key: set(['NEG', 'OBJ', 'POS'])
3 classes in response: set(['NEG', 'OBJ', 'POS'])
confusion matrix
key     NEG     OBJ     POS
NEG     1       33      3
OBJ     5       166     10
POS     0       44      13
----------------
accuracy: 0.6545 = 180/275
```

3 Naive Bayes

Deliverable 3

After training the Naive Bayes classifier, I applied it to the development data and got an accuracy of 72.00%. The pertinent files are

- **Python File for Naive Bayes**: *NaiveBayesClassifier.py*
- **Response File**: *generated_files/naive_response*

Here is the confusion matrix after running the *scorer* script on the response file.

```
3 classes in key: set(['NEG', 'OBJ', 'POS'])
3 classes in response: set(['NEG', 'OBJ', 'POS'])
confusion matrix
key     NEG     OBJ     POS
NEG     12      24      1
OBJ     12      169     0
POS     8       32      17
----------------
accuracy: 0.7200 = 198/275
```

3.1 Smoothing

Deliverable 4

I tried values of α from 10^0 to 10^{-7}. The plot of the values of α versus the accuracy rate is given in Figure 1.

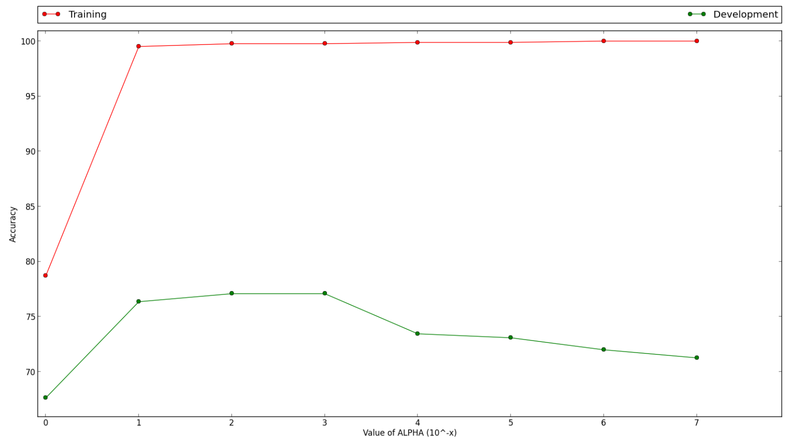

Figure 1: Plot of accuracy vs. α on both the training and development data

I got the best accuracy for the value of $\alpha = 10^{-3}$, which was 77.09%. The pertinent files are

- **Python File**: *SmoothingBayesClassifier.py*
- **Response File**: *generated_files/smoothing_response_6*

Here is the confusion matrix after running the *scorer* script on the response file.

```
3 classes in key: set(['NEG', 'OBJ', 'POS'])
3 classes in response: set(['NEG', 'OBJ', 'POS'])
confusion matrix
key     NEG     OBJ     POS
NEG     19      16      2
OBJ     16      161     4
POS     6       19      32
----------------
accuracy: 0.7709 = 212/275
```

Deliverable 5

The plot of a random subset of the $\log(\theta)$ values for the the maximum and minimum values of α (10^0 and 10^{-7} respectively) is represented in Figure 2.

The plotter script for this deliverable *deliverable5.py* also computes the correlation coefficient, by using the spatial distance correlation method of Python's *sciPy* [3]. I got the value of the correlation as **0.8364**.

Explanation

If the value of correlation had been 1, it would have meant that the $\log(\theta)$ values for the words were the same for both the maximum and minimum values of α are the same. A relatively high value like **0.8364** (quite close to the maximum value of 1) indicates that there is just a small variation in the weights assigned for each α. In other words, the variance in the $\log(\theta)$ for different value of α is non-zero but quite small. However, because of this small variance, the classification accuracy changes depending on the value of α.

Moreover, the correlation indicates that if the value of α is very small 10^{-7} (which is almost 0 practically), the smoothing effect is almost null and the classifer is effectively the same as the Naive Bayes. With a very large value of α ($10^0 = 1$, large is a relative term of course), we introduce a bias in the classifer. This is to our advantage with respect to the dataset of this particular project, because the distribution of classes in the training as well as the development data is the same (there is significantly more documents marked OBJ in training as well as testing). And clearly the best accuracy will result in a value of α somewhere halfway between the maximum and minimum value, which is true in my case as well.

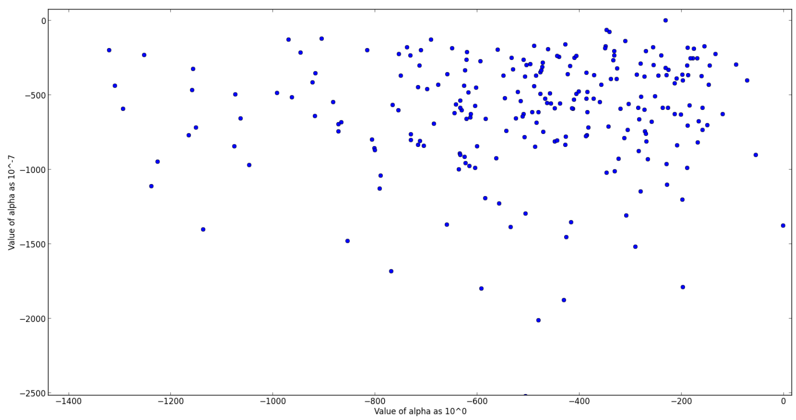

Figure 2: Plot of $\log \theta$ values for maximum and minimum values of the smoothing parameter.

Deliverable 6

The words which are most predictive of positive versus negative text from the training data are:

1. city
2. religious
3. republican
4. police
5. enjoy

Only the word *enjoy* appears in the sentiment vocabulary
The words which are most predictive of negative versus positive text from the training data are:

1. poker
2. games
3. rated
4. century
5. documentary

None of these words appear in the sentiment vocabulary. These words have been determined as well as checked for presence in the sentiment vocabulary by the Python scripts *MostPredictiveWords_Positive_Negative_Bayes.py* and *MostPredictiveWords_Negative_Positive_Bayes.py* and have been written into the files *generated_files/top5pos_neg* and *generated_files/top5neg_pos* respectively.

4 Perceptron

I implemented the perceptron by representing each document's feature function as a sparse array in *numpy*, whose size = number of distinct tokens in the training data. I was able to obtain a significant performance boost this way compared to representing it as a normal array (I was able to do 30 passes on training data in 40 seconds using *numpy* as opposed to 2 passes per minute using the default array structure). I am pretty confident of my implementation, but I get a low accuracy rate on development data, (best case 60% worst case 48%).

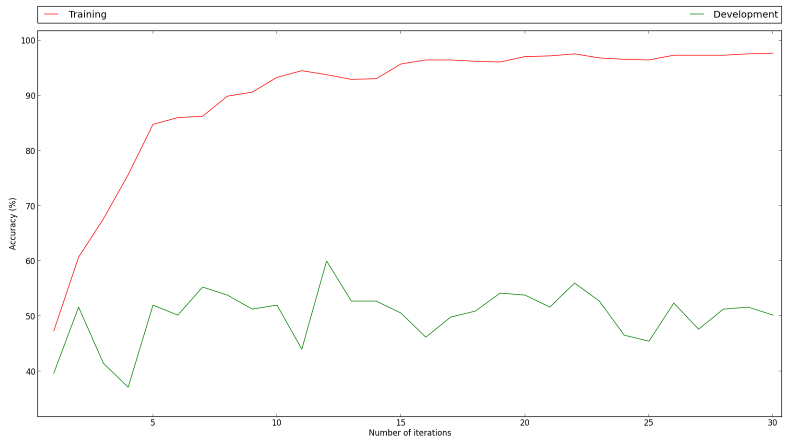

Figure 3: Plot of accuracy vs. number of iterations on training and development data

Deliverable 7

After making 30 passes on the training and development data, the plot for the accuracy vs. number of iterations is represented in Figure 3 The final confusion matrix for the training and development data are as follows:

```
Training Perceptron Response
-------------------------------------
3 classes in key: set(['NEG', 'OBJ', 'POS'])
3 classes in response: set(['NEG', 'OBJ', 'POS'])
confusion matrix
key     NEG     OBJ     POS
NEG     137     4       2
OBJ     5       509     2
POS     2       4       157
----------------
accuracy: 0.9769 = 803/822

Development Perceptron Response
---------------------------------------------
3 classes in key: set(['NEG', 'OBJ', 'POS'])
3 classes in response: set(['NEG', 'OBJ', 'POS'])
confusion matrix
key     NEG     OBJ     POS
NEG     3       24      10
OBJ     14      112     55
POS     3       31      23
----------------
accuracy: 0.5018 = 138/275
```

Pertinent files for this part of the project are:

- **Python file for the Perceptron**: *PerceptronClassifier_proper.py*
- **Python file for plotting**: *deliverable7.py*
- **File which has the accuracy measure after each pass**:
  1. For training data: *generated_files/training_perceptron_data_akj*

2. For development data: *generated_files/development_perceptron_data_akj*

- **Response File for the last pass**:
  1. For training data: *generated_files/training_perceptron_response_akj*
  2. For development data: *generated_files/perceptron_response_akj*

Deliverable 8

The words which are most predictive of positive versus negative text from the training data are:

1. about
2. enjoy
3. china
4. recommend
5. fitzgerald

Only the words *enjoy* and *recommend* appear in the sentiment vocabulary
The words which are most predictive of negative versus positive text from the training data are:

1. abandonment
2. destruction
3. death
4. the
5. at

The words *abandonment*, *destruction* and *death* occur in the sentiment vocabulary.

These words have been determined as well as checked for presence in the sentiment vocabulary by the Python script *MostPredictiveWords_Positive_Negative_Negative_Positive_Perceptron.py* (a single file with a ridiculous name I formed due to lack of imagination) and have been written into the files *generated_files/top5pos_neg_perceptron* and *generated_files/top5neg_pos_perceptron* respectively.

Two interesting observations I derive from the results of deliverables 8 and 6.

1. The perceptron being a error-driven approach, is able to identify the words which contribute to the class of a particular document, whereas the Naive Bayes method purely looks at the statistical occurrences of the words. Thus, I believe that had the documents in the development data have almost the same number of Obj, Pos and Neg documents, the perceptron classifier would have outperformed the Naive Bayes.
2. The top 5 list of the perceptron classifer has more words in the sentiment vocabulary than the Naive Bayes. This also substantiates the above point. As an aside, the word *enjoy* is the only common one between the two classifiers' top 5 lists.

5 Averaged Perceptron

Deliverable 9

I have performed the weight averaging after every iteration over the training data. After making 30 passes on the training and development data, the plot for the accuracy vs. number of iterations is represented in Figure 4 It is very noticeable that the accuracy measure (on the development data) does not fluctuate as much (a.k.a thrash) as it did without weight averaging and is visibly "smoother".

The final confusion matrix for the training and development data are as follows:

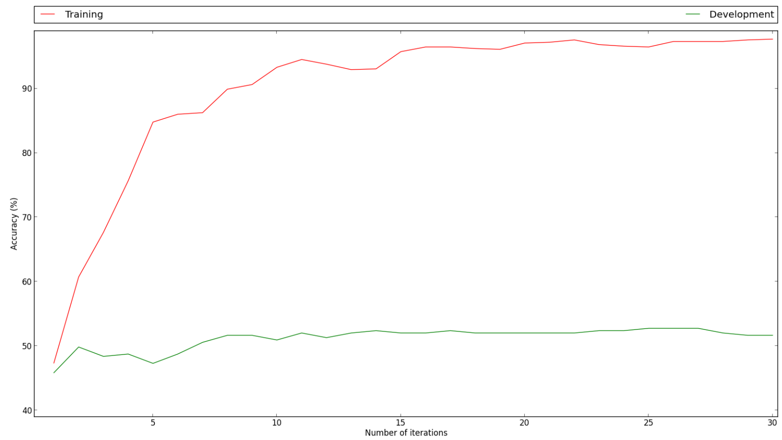

Figure 4: Plot of accuracy vs. number of iterations on training and development data - Averaged Perceptron

```
Averaged Training Perceptron Response
-------------------------------------
3 classes in key: set(['NEG', 'OBJ', 'POS'])
3 classes in response: set(['NEG', 'OBJ', 'POS'])
confusion matrix
key     NEG     OBJ     POS
NEG     137     4       2
OBJ     5       509     2
POS     2       4       157
----------------
accuracy: 0.9769 = 803/822
```

```
Averaged Development Perceptron Response
-------------------------------------
3 classes in key: set(['NEG', 'OBJ', 'POS'])
3 classes in response: set(['NEG', 'OBJ', 'POS'])
confusion matrix
key     NEG     OBJ     POS
NEG     4       24      9
OBJ     15      119     47
POS     3       35      19
----------------
accuracy: 0.5164 = 142/275
```

Pertinent files for this part of the project are:

- **Python file for the Perceptron**: *AveragedPerceptronClassifier_proper.py*
- **Python file for plotting**: *deliverable9.py*
- **File which has the accuracy measure after each pass**:
  1. For training data: *generated_files/training_perceptron_data_akj*
  2. For development data: *generated_files/averaged_development_perceptron_data_akj*
- **Response File for the last pass**:

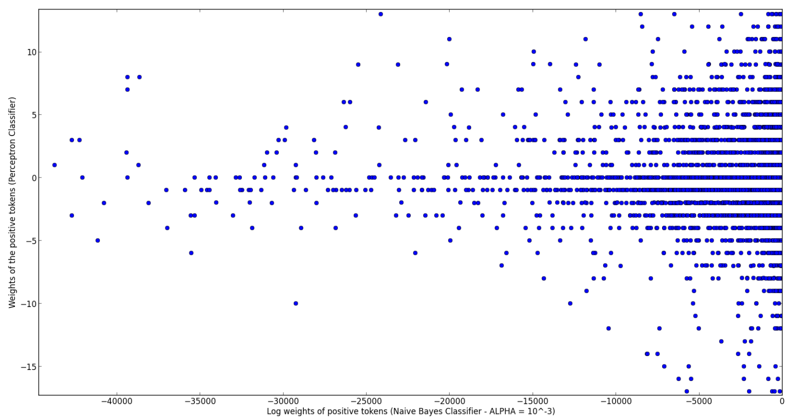

Figure 5: Averaged Perceptron weights vs. $\log(\theta)$ from Naive Bayes (for $\alpha = 10^{-3}$)

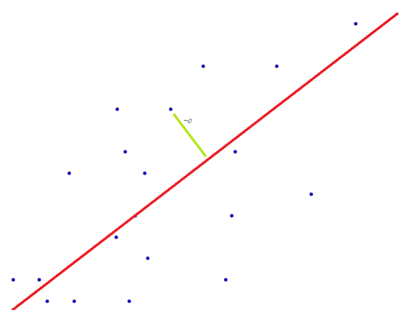

Figure 6: Figure 5 zoomed in

1. For training data: *generated_files/training_perceptron_response_akj*
2. For development data: *generated_files/averaged_perceptron_response_akj*

Deliverable 10

The plot of Perceptron weights vs. $\log(\theta)$ from Naive Bayes (for $\alpha = 10^{-3}$) is a very interesting one. The graph in Figure 5 is a random sampling of all the points. Notice that there are several data points for which either x or the y value is 0. Such points (by definition) are not considered for computing correlation [3].

The correlation measure, found similar to deliverable 5 came out to be **0.983837736138**. This shows that for all the "interesting points" (notice the clustering just before the line $x = 0$, around the imaginary line $x = y$), the weights assigned by both the Averaged Perceptron as well as the Smoothing Naive Bayes classifers are almost identical. Due to the poorly assigned scales for the axes by *matplotlib* (and my inability to correct it), the above mentioned clustering may not be obvious. Figure 6 provides a highly zoomed in version of the above plotting. Note that the line in red represents the line $x = y$ and since its zoomed in (I think about 1000% zoom) the distance of points from the line is actually miniscule (as indicated in the diagram almost 0).

I thought such a high correlation would translate in more words in their respective top 5 lists to be common, but I guess very small floating point differences must exist. So I think that both the Naive Bayes as well as the Perceptron classifiers are able to almost identically separate the tokens into the three classes, the differences arise primarily because of the nature of the data set.

6 Making it Better

Deliverable 11

I implemented all the methods to improve the Naive Bayes classifier, from the paper provided as reference [1] mainly because they were really easy to understand and fairly straight forward to implement. I believed that the accuracy rates of each of these techniques would be significantly higher than that of the Naive Bayes (NB). I expected that the Transformed Weight-Normalized Complementary Naive Bayes (TWCNB) method would give the highest accuracy. My rationale was as follows:

1. It takes into account the probability of a token appearing in classes other than the class under consideration. (CNB)
2. It performs weight normalization of the $\log(\theta)$ weights (although I have done the same in my implementation of the Naive Bayes classifier itself) (WCNB)
3. It takes into account length of each document as well as the "importance" measure of a token (an IDF measure), meaning frequently occurrring words will not be given higher weight. (informally, TWCNB = NB + CNB + WCNB + IDF measure)

However, when I ran the classifiers on the development data, my accuracy rates were in the following order,

Best Smoothened NB $\sim$ Complement NB >WCNB $\sim$ TWCNB >NB

I assume that the improvement techniques did not do much to improve the accuracy of the Smoothened Naive Bayes, simply because both the training data and development data are similarly biased towards the OBJ class. Even in the paper it was mentioned that Naive Bayes would drastically degrade if the development data had a different bias (or no bias) when compared to the training data. To test this, I trained the classifiers on the training data and then applied them on the development data. However, I decided to interchange every document marked as OBJ into POS and vice-versa in the *dev.key* file. Then I ran the scorer script on the response files of each classifier. I expected all of them to perform poorly (of course), but I thought the classifiers using weight normalization would perform better (even in such perplexing scenarios). And I was proven right, the new accuracy measures were ordered as,

Complement NB $\sim$ TWCNB >WCNB $\sim$ Best Smoothened NB >NB

This shows that in general, TWCNB would outperform Naive Bayes. Pertinent files can be found inside *generated_files/test* directory

7 Bake Off

Deliverable 12

The best system from the main part of the project was the smoothened Naive Bayes (!). I ran it on the test data (Best_Smoothing_Bayes_test_data.py) and the response is the file **jagannathan-arvind_krishnaa.main.response** inside the *Response* directory.

Deliverable 13

The best system from the making it better part of the project was the complement Naive Bayes. I ran it on the test data (Best_Better_Bayes1_test_data.py) and the response is the file **jagannathan-arvind_krishnaa.special.response** inside the *Response* directory.

References

[1] J.D. Rennie, L. Shih, J. Teevan, D. Karger, et al. Tackling the poor assumptions of naive bayes text classifiers. In *MACHINE LEARNING-INTERNATIONAL WORKSHOP THEN CONFERENCE-*, volume 20, page 616, 2003.

[2] T. Wilson, J. Wiebe, and P. Hoffmann. Recognizing contextual polarity: An exploration of features for phrase-level sentiment analysis. *Computational linguistics*, 35(3):399–433, 2009.

[3] Scientific Python. Spatial distance correlation
URL: http://docs.scipy.org/doc/scipy/reference/generated
/scipy.spatial.distance.correlation.html?highlight=correlation#scipy.spatial.distance.correlation.