EECS 4412 Project Report

Names and student numbers

Bryan Embree, 213 078 608
Vishal Malik, 214 537 146
Mohammad Omer Uddin, 214 427 611

Introduction

The objective of this project was to create a model capable of accurately predicting the sentiment (positive, neural, or negative) of Yelp reviews for businesses. The model was created by examining the training data (40000 pairs of text reviews and their sentiments) and looking for regularities in the words used for each different sentiment. Each review and its given sentiment (classification) form an example. In order to be able to build the model, the review portion of each example needed to be "cleaned" so that the model will be better able to learn from it. Once cleaned, there was still a need to select best words (the ones that are the best predictors of a review's classification). This is to both make the model faster to train, and to remove words that are not as useful when making a classification prediction. Using the cleaned input with all the non-selected words removed, the model could then be trained with examples, with the count of each word as features. After the model had been built using the training data, the model could then be used classify new reviews, such as the given testing data after it undergoes a similar cleaning process.

Preprocessing

We start off by first removing all punctuation, numbers, and other non-alphabetical symbols, then we convert all letters to lowercase. Any word of length two or less is removed after that. Then all words that contained sequences of the same consecutive letters, were reduced to a length of 1. This was to make words that had variations in their spelling more uniform (for example, "yummmmy" and "yummy" both become "yumy"). Next, all words contained in the "stop word" file were removed (the words too common or general purpose to be used for prediction). "told", "said", "ask" and "that" were manually added to the stop word list. These words were chosen because they are frequent but do not offer sentimental information. The words that remained were then "stemmed." The stemmer attempts to simplify words by removing their suffixes so that there will be less variation in the text and their sentiment still preserved. This is all done in `CleanText.py` with `train.csv` as input.
After that initial process, `RemoveInfrequentWords.py` is run to remove all the words that occurred only in one review. This was to remove words that were not statistically relevant for future classifications during the training and testing phases.
For the testing dataset, we first convert it into `test_formatted.csv` which follows the same format as the training set, by using Microsoft Excel (the class field was filled with "neutral"). Then same process is used for cleaning with `CleanText.py`, but `RemoveInfrequentWords.py` was not run because feature selection will remove the same words.

Feature selection

The output file of `RemoveInfrequentWords.py` is fed into `sparceARFFGen.py` to create a sparse arff file that Weka would be able to use it for creating the model. This file has the count of all the unique words as features, followed by the class attribute. This will also create file a file `features.list` that will contain the list of the features (words) contained in the generated ARFF file so that the same features can be selected in the testing data.

With the data cleaned, the most relevant words could then be selected by Weka's attribute selection feature. The full training set was used to find these variables using the Ranker method. In the end we settled on Chi-squared attributes to rank the attributes, randomly since it was very similar to info gain. We selected the top k = 1000 (k specified by the user in the `TopK.py` file) attributes and then the rest were removed using Weka's "Remove" filter. To create the ARFF for the test data, `features.list` was used to create an ARFF file with the same attributes as the training data using `sparceARFFGen_testing.py`. Weka automatically ignored all the extra features (since the model only uses 1000).

Building the model

Using the top 100 features of the cleaned training data, various models were experimented with using a split of 60% for training and the remainder for testing. Testing with 100 features gave a reasonable training time and a good indicator of which model would likely be worth developing further with more features. The Chi-squared metric produced better results based on the initial testing when compared to information gained, so Chi-squared was used for generating the possible models. Using bigrams are also tested, but they did not significantly improve the accuracy and used up feature spots which had lowered the number of top words that could be used. Twelve models were created using the above metrics, with the top three being: SMO, Logistic, and BayesNet. These models had a root mean squared error rate 0.3977, 0.3583, and 0.3741, respectively. The Logistic model was selected for further training with more features based on its lower error rate. It and also offers the benefit of being relatively fast when compared to others, taking 17 seconds to build which is almost half of SMO's time of 32 seconds. This means more attributes can be used in the final training set and provide better predictive powers. Using the Logistic model with 1000 features and the entire training set, the model created had an accuracy of 77.975% and a mean squared error rate of .3204. We decided to use the entire training set to build the final model because even though this risks overfitting the data, we did not find evidence of this.As the accuracy of both are almost same.

Classifying the test data

With the model built and the testing data cleaned and in the correct format, the model can classify each example in the data by loading the test data and running it on the imported trained model. We selected Weka to output the prediction as CSV, copied them to a text file and then ran `outputPredictionsFormatter.py` on it to get the prediction file.

Conclusions

Text classification requires a lot of preprocessing to make the text suitable for classification. It contains a lot of noise, useless symbols, redundant words, and uncommon ones which need to be filtered out before anything useful can be done with it. Classifying neutral reviews correctly proved to be the hardest class to classify correctly. Experimentation is required to see which method of feature selection and which classifier produce superior results. There was no simple way to predict what was best beforehand. Good results can be achieved by cleaning of the data, use of feature selection, and proper choice of a model. There are many

ways in which the results could be improved. More cleaning of the input and the use of more features for generating the model could improve the model's accuracy. Further experimentation with different criteria for feature selections could yield better results. There are also parameters for each model in Weka that can be tweaked, which may improve the results depending on the data. A lot of experimentation is needed to find the optimal cleaning method, attributes, and model. There is no "silver bullet" when it comes to data mining, with each process offering its own advantages and disadvantages.

Appendix

How to run the programs:

Files Needed in same Directory

1. CleanText.py
2. CleanTesting.py
3. outputPredictionsFormatter.py
4. RemoveInFrequentWords.py
5. sparceARFFGen_testing.py
6. sparceARFFGen.py
7. TopK.py
8. stop_words.lst
9. train.csv
10. test.csv (modified to be in the same format as train.csv using Excel)

Note: all files are assumed to be in the same directory and all python programs should be run with python3.

1. Run `CleanText.py` in the same directory as the training data (`train.csv`). This will output `CleanText_out.csv`. The testing data is formatted using Excel on `test.csv` and produces `test_formatted.csv`, which is then processed by running `CleanTesting.py` to produce `cleaned_test.csv`
2. Run `RemoveInfrequentWords.py`, this will output `RemoveInfrequentWords_out.csv`.
3. Run `sparceARFFGen.py` and redirect it's output to a file. For example, "`python sparceARFFGen.py > sparceARFFGen_out.arff`".
4. Open `sparceARFFGen_out.arff` in Weka and run the feature selection with information gain and Chi-squared as feature selections. Copy all the text output Weka's console and saved each file as "`Chi Selection Variables.txt`" and "`IG Selection Variables.txt`".
5. Run `TopK.py` and specify the k variable in the program's source code to adjust the number of best performing variables selected. `TopK.py` relies on the two manually created text files from above in step 4. This will output two files: `ChiTopK.txt` and `IGTopK.txt`. These two files contain the top k variables from each metric along with the class attribute number.
6. Using Weka's Remove filter, copy the list of k attributes from the metric you are using and select the option to invert the selection. Apply the filter. Now the data only contains the top k attributes. This can then be saved as its own ARFF file.
7. With the features selected, choose the model you wish to learn and once completed, the model will have been trained using those features. The model can then be saved.

8. Use `sparceARFFGen_testing.py` to produce `sparceARFFGen_testing_out.arff` by running the command "`python3 sparceARFFGen_testing.py > sparceARFFGen_testing_out.arff`" which produces the usable testing data for Weka.
9. Import `sparceARFFGen_testing_out.arff`, use the same previous remove filter. Then save the file and load it as a test set. On the learned model, rerun the testset. The results will be displayed in the Weka window and can be saved (make sure the Weka is configured to output predictions as CSV). Copy the output predictions (without any headers) into a file called `predictions.csv`. Then just run `outputPredictionsFormatter.py` in that same directory to get `Formated_Predictions.txt`.