Machine Learning
Final Project: Text Classification

Greg Guyles

April 28, 2014

1 Introduction

This paper discuses performing sentiment analysis on a set of movie reviews, my goal for this project was to classify reviews as positive or negative using several supervised learning algorithms. In section 2 I discuss the technologies used for this process, section 3 describes the general strategy used for evaluation and the results from the models tested. Section 4 describes an attempt to use the results of multiple classifiers to generate an improved prediction.

2 Platform Used

All calculations were performed on a virtual server hosted in the Google Compute Engine. The server resources were allocated as needed and ranged from systems using 2-8 cores and 7.5-56 GB of memory, this included the largest system offered by Google Compute Engine at the time of this writing. I did not attempt to distribute the work over a cluster of servers as I did not believe this would be achievable in the time frame of this project. Every attempt to parallelize the processing over multiple cores was made and this helped to achieve results in a workable time period. At times parallelization was inhibited by the 56 GB memory limit of Google Compute Engine, in this case jobs were manually initiated on separate virtual servers.

All the coding was completed in the Python programming language making use of the Numpy[1], Scipy[1], and Scikit-learn[2] package. The iPython[3] server was also installed to provide an easy interface for interacting with the remote server and was used primarily for script testing and debugging. I made significant use of the Scikit-learn documentation and official tutorials for this project, in particular the Working With Text Data[9] tutorial.

3 Model Evaluation

In order to train the classifiers, text data was vectorized such that each N-gram of the entire training set represented a column of a matrix. In this case a unigram represented a word and bigram represented a two word sequence (i.e. for the phase "The dog ran" the set containing all unigram and bigrams would be: ['the', 'dog', 'ran', 'the dog', 'dog ran']). Which N-grams were included in the set was one of the parameters adjusted when testing predictions.

Each training example was represented as a row of this matrix and the value of each index indicated the prevalence of that N-gram in that document. Each value was a real number between

0 and 1 and represented the percentage of the document that was comprised of that N-gram. Without any parameter selection each row of this matrix would sum to 1.

The general format for determining model parameters was to preform an exhaustive grid search on a broad set of parameters using single fold cross validation, then to complete a second narrow parameter search using 10-fold validation. This approach allowed parameter testing on the entire training set that returned results in a time frame appropriate with the duration of the course. With an indication of the well suited parameters a smaller search was then completed using 10-fold cross validation in order to best select the optimum parameters. The single fold validation set was not randomized meaning there was not variability on this tested data, this allowed for a true comparison of each parameter set at the risk of selecting a solution that over-fit the test set. The test set was a 25% portion of the training data, comprising of 6,250 examples, that was omitted from parameter selection and used to generate a general accuracy score, this data was included in the training set when generating the Kaggle submission file. When referring to a model's accuracy and when displaying confusion matrices I am referring to this 25% subset, when referring to the accuracy that the Kaggle submission page generated I explicitly note it as the Kaggle test accuracy.

3.1 Linear SVM

I initially began training using a Linear Support Vector Machine. I picked this model to obtain a first benchmark because I knew the computation time would be minimal, allowing for quick parameter experimentation. Using the Pipeline[8] object from Scikit-learn allowed me to easily preform a grid search varying parameters for both the text vectorizer and the Linear SVM classifier in one step.

The initial search examined values of the C parameter to the SVM as well as the N-gram range and Frequency Threshold of the vectorizer which converted the text documents into a matrix. Setting the N-gram range to (1, 1) would cause the vectorizer to only include unigrams where (1, 2) would include unigrams and bigrams, setting this to (2, 2) would include bigrams only. The frequency threshold limited the inclusion of an N-grams above a certain frequency in the training set, setting Max Frequency Threshold to 0.9 would exclude all values that occurred more frequently than 90% of the training set, setting the Min Frequency Threshold would exclude values occurring less often than the frequency.

Table 1: Parameters tested for Linear SVM
Optimal Parameters **underlined**

Component	Parameter	Values Tested
Classifier	C	0.1, **1**, 10, 100, 1000
Vectorizer	N-gram Range	(1, 1), **(1, 2)**, (2, 2), (1, 3)
Vectorizer	Max Frequency Threshold	**0.75**, 0.85, 0.9, 0.95, 0.99
Vectorizer	Min Frequency Threshold	**None**, 0.01, 0.05, 0.1

The most influential parameter that was tested was clearly the Min Frequency Threshold, models trained with the lowest value (0.01) or None most often produced the highest cross validation scores by an improvement of 5-10 points. For this reason I removed the Min Threshold parameter and did not test further with other models. Following this, the C parameter was quite influential, values of 0.1 and 1 were found to fit best.

Both the N-gram Range and the Max Frequency Threshold were not as clearly decisive; while the grid search process favored the inclusion of unigrams and bigrams as well as a Max Frequency of 0.75 these values only showed slight improvement over the field. A separate smaller search was conducted using 10-fold cross validation and determining the inclusion of unigrams and bigram was preferred with a score of 0.905 to bigrams alone with 0.888. Further searching also proved that smaller Max Frequency Threshold values provided a better fit, 0.75 was used for this model and lower values were experimented with in later models.

Table 2: Linear SVM Confusion Matrix
Accuracy: 0.907787

	Neg	Pos
Neg	2790	318
Pos	264	2878

3.2 SVM with (Gaussian) RBF Kernel

Fitting the RBF kernel to SVM followed a work flow similar to the linear SVM example. I was surprised to find quite similar results to the linear SVM indicating that the data may be linearly separable.

Table 3: Parameters tested for RBF SVM
Optimal Parameters **underlined**

Component	Parameter	Values Tested
Classifier	C	1, 100, **500**, 1000, 3000, 5000, 10000
Classifier	γ	1, **0.1**, 0.01, 0.001
Vectorizer	N-gram Range	(1, 1), (1, 2), (2, 2), **(1, 3)**, (1, 4)
Vectorizer	Max Frequency Threshold	**0.1**, 0.5, 0.7, 0.8, 0.9, 0.99

The Linear SVM cross validation scores for the inclusion of trigrams was not far from the scores that did not include trigrams so I was not surprised to see that they were found to be optimal here. What I was surprised to see was the improvement made by the lowered Max Frequency Threshold, this setting excluded the top 90% of frequently used N-grams. By including trigram the set would contain millions of N-grams so only working with this subset is still a significant number of data points.

The initial grid search of RBF SVM took 29 hours, 15 minutes with work distributed over 8 cores on a single server. This was the main reasoning for using single-fold cross validation first then preforming 10-fold cross validation on a smaller set. To completely preform 10-fold cross validation may have taken over a week or required distribution over multiple servers and I did not feel either option would fit into the time frame of this project.

Table 4: RBF SVM Confusion Matrix
Accuracy: 0.91008

	Neg	Pos
Neg	2803	306
Pos	256	2885

3.3 Random Forrest

Due to the success my classmates found with the Random Forrest classifier on an earlier project I was determined to implement this for the movie review project. This was a rather difficult task mainly due to the requirements of the Scikit-Learn's ExtraTreesClassifier()[4] which implements this classifier. The Scipy library has a very efficient way of storing sparse matrices which I made use of when processing multi-N-gram matrices with over 1 million features, this became a problem with the ExtraTreeClassifier() which would not accept this data type but instead required a dense matrix.

Table 5: Parameters tested for Random Forrest
Optimal Parameters **underlined**

Component	Parameter	Values Tested
Classifier	Min Samples Split	2, 4, 8, **16**, 32, 64
Classifier	Min Samples Leaf	2, 4, **8**, 16, 32, 64
Classifier	Max Depth	None, 256, 512, **1024**, 2048, 4096
Vectorizer	N-gram Range	**(1, 2)**
Vectorizer	Max Frequency Threshold	**0.1**, 0.5

The solution to this dilemma was found in the Scikit-Learn feature selection library; the SelectKBest()[6] object fit on the training and target data and select the most important features. This used the chi2[5] (chi-squared) statistic to identify features that were most relevant to classification. I first vectorized the training data using unigrams and bigrams which produced 1,048,576 features then using SelectKBest() I selected the top 80,000 features to convert to a dense matrix for use with the ExtraTreeClassifier(). Choosing 80k features was a bit arbitrary, I also did some testing using 120k and 100k, initially guessing that using about 10% of the original features would be a good staring point. Of these test using 80k produced the best results but the testing was not standardized and I can not say with confidence that any improvement was due to this level of feature selection. For this project I was primarily interested in running the test successfully without risk of a memory error; selecting 80k features required about 30 GB of memory to fit the model which worked well on my 56 GB server. Further experimentation regarding the optimal number of features would be an interesting subject for future work.

Table 6: Random Forest Confusion Matrix
Accuracy: 0.830080

	Neg	Pos
Neg	2675	463
Pos	544	2568

3.4 Naive Bayes

Testing Naive Bayes provided a classifier that trained quickly and produced impressive results. I implemented this using Scikit-learn's MultinomialNB()[7] classifier without a smoothing parameter or any class prior probabilities. I ran a grid search to select the best vectorizer parameters which produced results consistent with other models by using unigrams, bigrams and trigrams and selecting a low Max Frequency Threshold.

Table 7: Parameters tested for Naive Bayes
Optimal Parameters **underlined**

Component	Parameter	Values Tested
Vectorizer	N-gram Range	(1, 1), (1, 2), **(1, 3)**
Vectorizer	Max Frequency Threshold	**0.1**, 0.3, 0.5, 0.7)

Table 8: Naive Bayes Confusion Matrix
Accuracy: 0.89024

	Neg	Pos
Neg	2814	376
Pos	310	2750

4 Combination Result

I theorized that these different models would find different fittings of the data and that I might be able to use this to identify errors in one particular model. After training each model on the entire training set and predicting the Kaggel test results I took these results and used Microsoft Excel to preformed a simple "Best of Three" comparison for each test example using Linear SVM, Random Forest and Naive Bayes. The resulting set received a Kaggle test score accuracy of 0.89856 which fell short of an earlier submission of 0.90252 for linear SVM. Although it didn't show an improvement here, I think this is a strategy worth investigating to understand differences in how the models fit the data.

5 Conclusion

The best model choice was a bit of a tossup between Linear SVM and RBF SVM. RBF showed a better initial accuracy of 0.91008 vs 0.907787, yet when submitted to Kaggele both sets produced a Kaggle test accuracy of 0.90252. Though the predicted targets were not identical it does appears that the RBF kernel is finding a mostly linear fit to the data.

I felt the most interesting and perhaps influential subject in this project was vectorization and parameter selection. The inclusion of bigrams and trigrams in the feature set showed noticeable prediction improvements for all classifiers. I was surprised to see that the Max Frequency Threshold was usually optimal a value as low as 0.1. By including bigrams and trigrams the feature set is quite large so training on this subset is still quite sizable, ignoring the frequently use N-grams allowed the models to focus on the words and word sequences that are particularly important to sentiment analysis.

The significant amount of processing used for this project was made possible due to cloud resources, in this case Google Compute Engine. This allowed for efficient multi-core programming, for massive matrices to be held in memory and for me to work with multiple virtual servers at a time. At the busiest point I was working with 26 cores and 180 GB of memory spread across 5 servers; not all of these cores were in use as in the case of the Random Forest classifier where I needed a high memory machine which was only available with 8 cores. Fortunately I had credits on this service which had been supplied to many CU students, without these credits the cloud usage for this project would have cost about $125.00.

References

[1] Eric Jones, Travis Oliphant, Pearu Peterson, et al. SciPy: Open source scientific tools for Python. Available: http://www.scipy.org/, 2001–.

[2] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[3] Fernando Pérez and Brian E. Granger. IPython: a system for interactive scientific computing. *Computing in Science and Engineering*, 9(3):21–29, May 2007.

[4] scikit-learn developers. sklearn.ensemble.ExtraTreesClassifier. Available: http://scikit-learn.org/stable/modules/generated/sklearn.ensemble.ExtraTreesClassifier.html, April 2014.

[5] scikit-learn developers. sklearn.feature_selection.chi2. Available: http://scikit-learn.org/stable/modules/generated/sklearn.feature_selection.chi2.html, April 2014.

[6] scikit-learn developers. sklearn.feature_selection.SelectKBest. Available: http://scikit-learn.org/stable/modules/generated/sklearn.feature_selection.SelectKBest.html, April 2014.

[7] scikit-learn developers. sklearn.naive_bayes.MultinomialNB. Available: http://scikit-learn.org/stable/modules/generated/sklearn.naive_bayes.MultinomialNB.html, April 2014.

[8] scikit-learn developers. sklearn.pipeline.Pipeline. Available: http://scikit-learn.org/stable/modules/generated/sklearn.pipeline.Pipeline.html, April 2014.

[9] scikit-learn developers. Working with Text Data. Available: http://scikit-learn.org/dev/tutorial/text_analytics/working_with_text_data.html, April 2014.