**Dataset:**

All tests were performed using the Yelp review polarity dataset from the Yelp Dataset Challenge 2015 (<http://www.yelp.com/dataset_challenge>). The original dataset contains 560,000 samples of both positive and negative reviews, which I have down sampled to create an imbalanced dataset to approximate numbers of ACM data provided to me by George (~97/3 split with ~300 ACM tables).

|  |  |  |
| --- | --- | --- |
| Class | Training Count | Testing Count |
| Negative | 9962 | 1995 |
| Positive | 299 | 59 |

**Text Cleaning:**

All cleaning of the test text dataset was performed using a python script with the SpaCy package unless otherwise stated. Cleaning was performed in the following process:

1. Manually remove special characters and excess newline delimiters that SpaCy would miss
2. Cast text to lowercase to avoid SpaCy’s quirk of not tagging stop words that contained a capital letter (Also ran tests on data cleaned without this step)
3. Tokenize the text at the word level
4. Removal of stop words (If step 2 skipped, any stop words containing a capital letter were not removed)
5. Removal of punctuation
6. Lemmatize each word token (Also ran tests on data cleaned without this step)

**Text Preparations for Modeling:**

I used the following text preparation/vectorization methods on the cleaned text before modeling. I will indicate which methods (by number) were used with each algorithm.

1. Bag of Words (including single tokens, bigrams, and trigrams)
2. Term Frequency – Inverse Document Frequency (tests ranging from single word tokens up to 6-grams)
3. Token sequence vectors (testing including only single word tokens and token chunking up to trigrams)
4. Token sequence with individual word embeddings (Capped at 2000 words per document. Any documents with fewer than 2000 words were zero padded on the right. Starting with pretrained GloVe embedding vectors in neural nets.)
5. Mean text embedding
6. Bag of Words + mean text embedding
7. TF-IDF + mean text embedding
8. Token sequence vectors + mean text embedding
9. Bag of Words + PoS word counts
10. TF-IDF + PoS word counts
11. Token sequence with individual word embeddings and PoS word counts and ratios

**Class Imbalance Treatments:**

Below is a list of treatments used to handle the large imbalance of classes in the dataset. I will indicate (by letter) which treatment was used for each text preparation technique.

1. No Treatment
2. Random Under Sampling
3. Random Over Sampling
4. SMOTE
5. Custom: I created 33 (total number of negatives/total number of positives) balanced datasets, each comprised of all positive samples and a unique set of negative samples. Each dataset was used to train its own classifier, and all 33 classifiers were used to create a single ensemble classifier. (This treatment was only used for Naïve Bayes, Linear Logistic Regression, and Support Vector Classifier tests for the obvious reason of time required to train 33 classifiers to use in an ensemble.)

**Modeling Algorithms tested:**

Below is a list of classification algorithms I have tested, along with a list of text preparations used on each one.

1. Naïve Bayes (1(A, B, C, D, E), 2(A, B, C, D, E), 3(A, B, C, D), 5(A, B, C, D), 9(A, B, C, D), 10(A, B, C, D)) Note: I had to shift text embedding values to all positive for use with NB
2. Linear Logistic Regression (1(A, B, C, D, E), 2(A, B, C, D, E), 3(A, B, C, D), 5(A, B, C, D), 6(A, B, C, D), 7(A, B, C, D), 8(A, B, C, D), 9(A, B, C, D), 10(A, B, C, D))
3. Support Vector Classifier (1(A, B, C, D, E), 2(A, B, C, D, E), 3(A, B, C, D), 5(A, B, C, D), 6(A, B, C, D), 7(A, B, C, D), 8(A, B, C, D), 9(A, B, C, D), 10(A, B, C, D))
4. Neural Networks
   1. Fully connected dense networks (4(A, B, C, D)), 11(A, B, C, D))
   2. CNN (4(A, B, C, D), 11(A, B, C, D))
   3. RNN-LSTM (4(A, B, C, D), 11(A, B, C, D))
   4. RNN-GRU (4(A, B, C, D) , 11(C))
   5. CNN + RNN-LSTM (11(C))
   6. CNN + RNN-GRU (11(C))

Note: I have not done a deep dive on tuning hyperparameters for each of these models. I have not done any fine tuning of parameters in each layer of the Neural Nets and have been sticking to common use parameters from papers and articles on the subject.

**Non-NN Results**

Just like the last set of tests, models fit using tf-idf performed better than models fit using word counts, mean embedding vectors, token sequences, etc.

As I have run these tests on a highly imbalanced dataset and I am more interested in the true positive rate than false negatives, accuracy is no longer an appropriate measure of a model’s success. Instead I am using the F1 score and recall in determining the most successful models. In general, Naive Bayes models performed best in these tests and produced very transparent results.

In terms of recall scores, models using under-sampling to correct the class imbalance have outperformed those using over-sampled and SMOTE corrected data. However, this does often come at the cost of an increased false positive rate as demonstrated by lower F1 scores. For example, a Naïve Bayes model using the full cleaning process and the three standard correction methods yielded the following results:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | Under-sampling | | Over-Sampling | | SMOTE | |
|  | True Pos. | True Neg. | True Pos. | True Neg. | True Pos. | True Neg. |
| Predicted Pos. | 48 | 11 | 35 | 24 | 39 | 20 |
| Predicted Neg. | 300 | 1695 | 62 | 1933 | 105 | 1890 |

Under-sampling:

* Recall = 0.814
* F1 score = 0.236

Over-sampling:

* Recall = 0.593
* F1 score = 0.449

SMOTE:

* Recall = 0.661
* F1 score = 0.384

An ensemble of the three models using the mean predicted probabilities basically averaged out these results, yielding the following:

|  |  |  |
| --- | --- | --- |
|  | True Pos. | True Neg. |
| Predicted Pos. | 38 | 21 |
| Predicted Neg. | 87 | 1908 |

Ensemble:

* Recall = 0.644
* F1 score = 0.413

I was able to obtain slightly better results using under-sampling and a trimmed down data cleaning process where I did not cast the text to lowercase and did not lemmatize the words.

|  |  |  |
| --- | --- | --- |
|  | True Pos. | True Neg. |
| Predicted Pos. | 48 | 11 |
| Predicted Neg. | 234 | 1761 |

However, this improvement may be entirely due to the nature of sentiment analysis where:

* Specific word use is important and root words do not always convey the same meaning, especially in sentiment analysis.
* Capitalization of text is u usually indicative of anger, so SpaCy’s quirk of keeping stop words containing a capital letter became a way for the model to be better able to predict the negative class here.

My custom ensemble method performed almost as well, with the following results:

|  |  |  |
| --- | --- | --- |
|  | True Pos. | True Neg. |
| Predicted Pos. | 46 | 13 |
| Predicted Neg. | 218 | 1777 |

The addition of PoS word counts and total length of text information also tended to increase the model’s ability to recognize the negative class, but at the cost of missing more of the positives. In some cases, the inclusion of PoS count ratios had the opposite effect and degraded the model’s ability to recognize the negative class, greatly increasing the false positive rate.

In comparison to the Naïve Bayes results, Linear Logistic Regression and SVC based models run on the same data also tended to have higher false negative rates.

**NN Results**

I will break down results based on each of the aspects tested below.

**Inputs/Auxiliary Inputs**

Given the results from the previous set of tests, I skipped running any neural net models using random or self-trained word embedding vectors and have only been running tests starting with the pre-trained GloVe vectors. I have also added an auxiliary input consisting of Parts of Speech word counts and ratios to each of the tested neural networks. The additional information provided by this auxiliary input does improve the models’ ability to classify the minority class across most tested architectures, and the best performing model did include this auxiliary input.

**Balancing Treatments**

* No Treatment: Was not very good at finding minority (positive) class documents and tended to classify everything as the majority (negative) class.
* Random Under Sampling: Best performing treatment tested. Was able to detect most of the minority class but did tend to mis-classify 10%-20% of testing samples. Had the highest true positive rate when training and validating with two under sampled datasets as opposed to training with an under sampled set and validating with an imbalanced set.
* Random Over Sampling: Produced similar results to training without balancing the classes. Tended to classify most of the dataset as majority (negative) class.
* SMOTE: Caught slightly more minority class documents than Random Over Sampling, but at the cost of an increased false positive rate.

**Architectures**

Simple neural networks including 2-4 layer fully connected networks, single layer RNN (GRU and LSTM), and single layer CNN performed very poorly, tending to predict nearly everything in a single class independent of treatments for balancing classes.

The best performing architecture was a single layer RNN (GRU) with average and max pooling plus auxiliary input fed into 2 fully connected dense layers, trained and validated on under sampled datasets. This model was trained using a batch size of 40 for 10 epochs, but was technically overfit after the 4th epoch. Results are shown below:

* Recall = 0.898
* F1 score = 0.835

|  |  |  |
| --- | --- | --- |
|  | True Pos. | True Neg. |
| Predicted Pos. | 53 | 6 |
| Predicted Neg. | 15 | 44 |

The next best architecture is the same as the previous one discussed, with the addition of a single CNN layer added after the RNN (GRU).

* Recall = 0.847
* F1 score = 0.813

|  |  |  |
| --- | --- | --- |
|  | True Pos. | True Neg. |
| Predicted Pos. | 50 | 9 |
| Predicted Neg. | 14 | 45 |

Some of the deeper and more complex architectures achieved lower false positive rates, but at the cost of losing some of the true positives. However some of these may prove to be better at locating pages with ACM tables, since that is quite different from the sentiment analysis I have been using as a proxy.

**Batch Size**

I am still running tests on the effects of increasing the batch size and number of epochs in training, but current results using the best performing architecture show that a smaller batch sizes (range 10-50) have performed better than larger batch sizes (range 100-500).

Some of my results so far are listed below:

* Smaller batch sizes do tend to overfit sooner because the weights are updated on losses calculated from fewer samples at a time.
* Larger batch sizes can cause very large fluctuations in the model accuracy.
* The best model discussed above was trained with a batch size of 40, and further tests on the same architecture with varying batch sizes all returned worse results.