A Glimpse into the Art of Spam Filters

Vincent van der Meulen and David Yuan

Introduction

Ever since the first spam email was sent in 1978, Nigerian princes, phishers, cure-all pills, and fake lottery emails have scammed billions from individuals and corporations alike. To protect users, spam filters were made to differentiate between valid and potentially dangerous emails. Our goal was to implement spam filters using several different algorithms, and evaluate the effectiveness of our predictions. First we will go over the data set itself, and what features we transformed and created for modeling. Then we will discuss how we set up our machine learning pipeline, the models we chose, and our rationale for choosing them. Finally we talk about and interpret the results of our models.

Data and Data Prep

To have a standardized set of "spam or ham" messages that were prelabeled, we decided to use the SMS Spam dataset from the University of California, Irvine, on Kaggle. This data contained a column representing the message, and a label of "ham" or "spam." After initial data reformatting, we decided to add several more features that may be indicative of spam emails (as outlined by a paper by UC Berkeley). Among these features was a count of non-standard punctuations, which we interpreted as an instance where a space was inserted before a punctuation, but not after. An example would be: "Hi .There ." We used regular expressions to help find the number of occurrences of non-standard punctuations, with the punctuations being: commas, periods, apostrophes, exclamation points, hashtags, forward slashes, "at" symbols, percent signs, carats, semicolons, and quotation marks. The other feature we added in was the ratio of uppercase to lowercase letters. Our rationale was that spam emails would contain more grammatical errors and tend to use more uppercase letters.

Another feature we wanted to consider was a count of the number of known spam words. It would be pointless to create a list of "spam" words using our current dataset (in this instance, each value for spam would be 100% and 0% for ham). Instead, we downloaded a list of 27,000 spam words from GitHub, and counted the number of of blacklisted words that appeared in the message. We removed all punctuation from the test data, after which we created lists of lemmetized and stemmed messages, since we were curious which one would perform best in our models.

The Pipeline

To be able to quickly try out different models while mitigating the dangers of risks such as overfitting, we set up a pipeline consisting out of several common operations. As we saw in the previous section, every email consists out of words and we can look at them from various perspectives. However, our classifiers need more tangible features (numbers) to be able to assign a spam or ham label. Hence, the first step in our pipeline is to assign numeric values to the words used in emails. The most naive solution here is to use a bag-of-words approach. In other words, we can count the number of times a word occurs in an email and train our model on the resulting vectors. The problem with this approach is that emails

have different lengths. As a result, two spam emails that are essentially the same but vary drastically in length will appear to be completely different.

A better variation of the bag-of-words approach normalizes word counts by the number of words in the document. In other words, we take an email's word counts and divide them by the number of words used in the email. At this point, we have reached the algorithm that the "tf" (term frequency) in tf-idf refers to. We could have stopped here but instead decided to also pay attention to the underuse, or overuse, of certain words. If a word such as "money" frequently appears in our corpus, it will help us little in deciding whether an email is spam or not. However, if a word (or symbol), such as "$," is rare, the presence or absence of this word could be a sign that an email is or is not spam. Naturally, underuse and overuse only matters when usage varies across spam and ham emails. Nonetheless, looking at inverse document frequency ("idf") can make spam classification a lot easier since it gives us a better sense of the characteristics of spam and ham.

Besides tf-idf, grid search and k-fold cross validation are essential parts of the pipeline. We use grid search since, regardless of the model we use, we want to tune both the hyperparameters of tf-idf and our model itself. For instance, when it comes to tf-idf, we might want to explore whether the idf step indeed gets us better results. We combine grid search with k-fold cross validation to prevent overfitting. We choose to run cross-validation with 10 folds. We ran the pipeline on the original text data, the lemmatized text, and the stemmed text.

Model Selection

In total, we used our pipeline in combination with three models: Multinomial Naive Bayes, Stochastic Gradient Descent, and Perceptron. We opted for Multinomial Naive Bayes because Naive Bayes is quick and our data is sparse (very few words of the corpus are present in each email). We chose to use Multinomial Naive Bayes, as opposed to Bernoulli Naive Bayes and other options, because it takes into account word counts, which in this case were our tf-idf scores. The function we implemented for Naive Bayes also includes smoothing, so our model will still do relatively well even on words in the test data it has never seen in the training messages.

For our second model, we decided to use Stochastic Gradient Descent as Gradient Descent is both easy to understand and interpret. We decided to use Stochastic Gradient Descent over Batch Gradient Descent because needing to look over all the training examples makes Batch Gradient Descent a lot slower than Stochastic Gradient, which only needs to look at a single training example per iteration. We chose Hinge as our loss function, because hinge is more sensitive to outliers, which TF-IDF stresses.

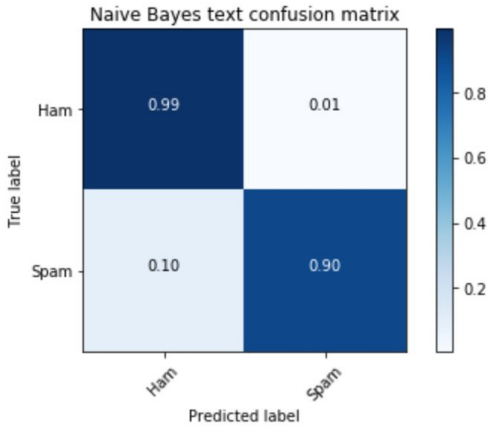
Our final model is a Perceptron, which we can think of as a single layered neural net. Perceptrons are binary classifiers that use a linear function with weights to determine the output. We opted against adding in a hidden layer since we thought it was not necessary for this data set. We hoped the Perceptron would be able to observe patterns in spam words that the other classifiers were not able to.

Results

Naive Bayes - Original Text Data

Score: 0.980967917346 **ROC**: 0.948915293906 **Time:** 6.55 seconds

	precision	**recall**	**f1-score**	**support**
ham	0.98	0.99	0.99	1587
spam	0.95	0.90	0.93	252
avg / total	0.98	0.98	0.98	1839

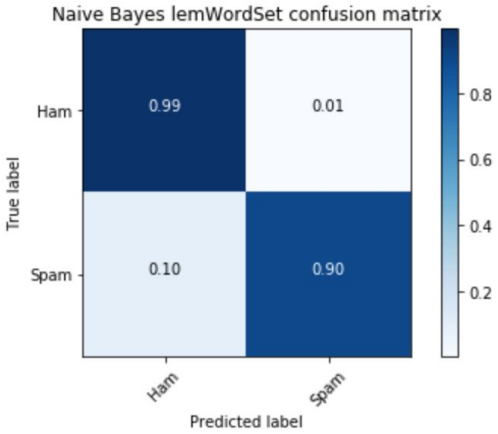


Confusion Matrix Naive Bayes - Original Text Data

Naive Bayes - Lemmatized Text

Score: 0.981511691136 **ROC**: 0.948915293906 **Time:** 6.87 seconds

	precision	**recall**	**f1-score**	**support**
ham	0.99	0.99	0.99	1587
spam	0.96	0.90	0.93	252
avg / total	0.98	0.98	0.98	1839

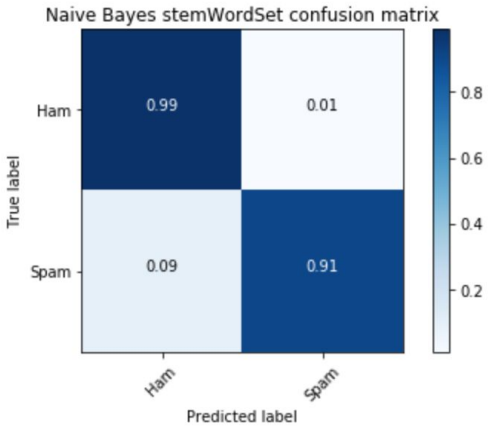


Confusion Matrix Naive Bayes - Lemmatized Text

Naive Bayes - Stemmed Text

Score: 0.979336595976 **ROC**: 0.949639181444 **Time**: 7.92 seconds

	precision	**recall**	**f1-score**	**support**
ham	0.99	0.99	0.99	1587
spam	0.94	0.91	0.92	252
avg / total	0.98	0.98	0.98	1839

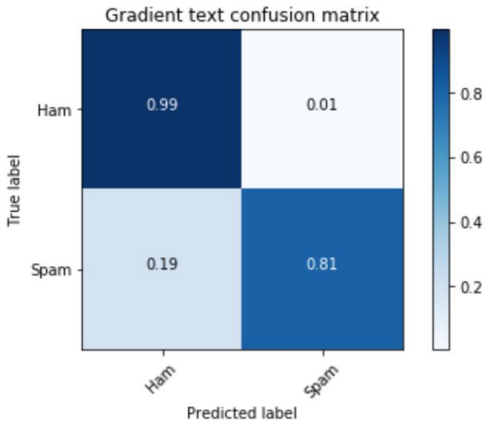


Confusion Matrix Naive Bayes - Stemmed Text

Stochastic Gradient Descent - Original Text Data

Score: 0.968461120174 **ROC**: 0.903280373271 **Time**: 6.88 Seconds

	precision	**recall**	**f1-score**	**support**
ham	0.97	0.99	0.98	1587
spam	0.95	0.81	0.88	252
avg / total	0.97	0.97	0.97	1839

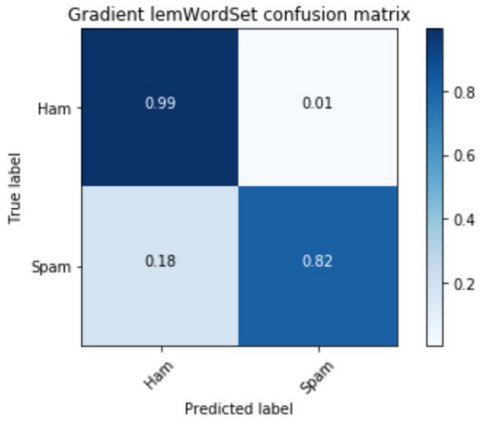


Confusion Matrix Stochastic Gradient Descent - Original Text Data

Stochastic Gradient Descent - Lemmatized Text

Score: 0.970092441544 **ROC**: 0.905894619978 **Time**: 7.14 seconds

	precision	**recall**	**f1-score**	**support**
ham	0.97	0.99	0.98	1587
spam	0.96	0.82	0.88	252
avg / total	0.97	0.97	0.97	1839

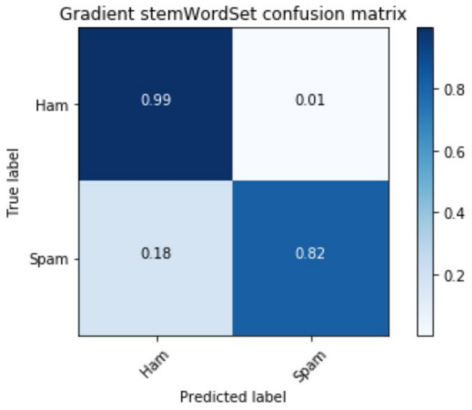


Confusion Matrix Stochastic Gradient Descent - Lemmatized Text

Stochastic Gradient Descent - Stemmed Text

Score: 0.971179989125 **ROC**: 0.908193806823 **Time**: 7.49 seconds

	precision	**recall**	**f1-score**	**support**
ham	0.97	0.99	0.98	1587
spam	0.96	0.82	0.89	252
avg / total	0.97	0.97	0.97	1839

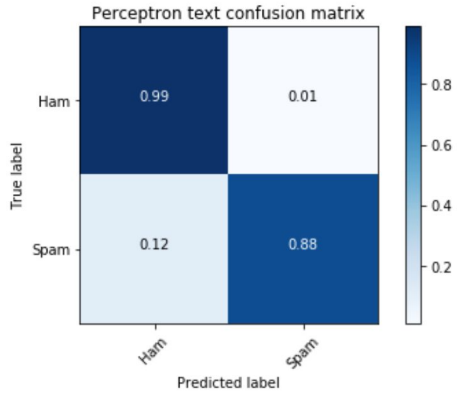


Confusion Matrix Stochastic Gradient Descent - Stemmed Text

Perceptron - Original Text Data

Score: 0.976073953235 **ROC**: 0.93773441954 **Time**: 6.89 seconds

	precision	**recall**	**f1-score**	**support**
ham	0.98	0.99	0.99	1587
spam	0.94	0.88	0.91	252
avg / total	0.98	0.98	0.98	1839

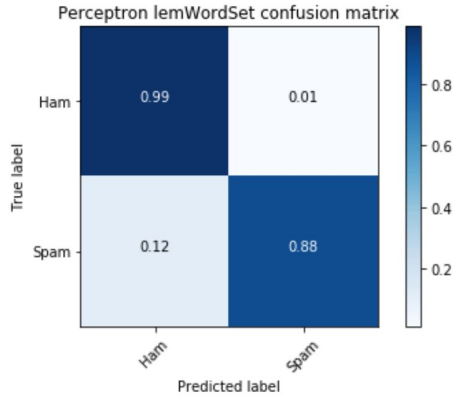


Confusion Matrix Perceptron - Original Text Data

Perceptron - Lemmatized Text

Score: 0.972811310495 **ROC**: 0.935844060371 **Time**: 8.03 seconds

	precision	**recall**	**f1-score**	**support**
ham	0.98	0.99	0.98	1587
spam	0.91	0.88	0.90	252
avg / total	0.97	0.97	0.97	1839

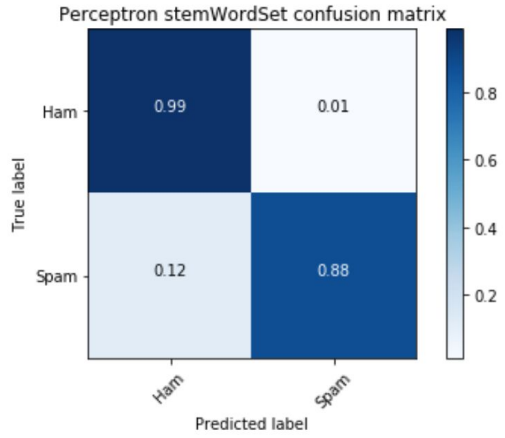


Confusion Matrix Perceptron - Lemmatized Text

Perceptron - Stemmed Text

Score: 0.973355084285 **ROC**: 0.93449005311 **Time**: 8.03 seconds

	precision	**recall**	**f1-score**	**support**
ham	0.98	0.99	0.98	1587
spam	0.92	0.88	0.90	252
avg / total	0.97	0.97	0.97	1839



Confusion Matrix Perceptron - Stemmed Text

Discussion

Looking at the results, we can conclude that we have an easy time distinguishing spam from ham. Our lowest score (0.968461120174) appears when we use Stochastic Gradient Descent using the original text data. However, a score of 0.97 is still incredibly high. A high score is expected since we rely on popular machine learning algorithms that have been improved over the years. We also did extensive data prep and took various measures (grid search and k-fold cross validation) to ensure both performance and scalability. Nonetheless, it is good to note that scores in the 0.95 - 1.00 are so high that it might be worth using these same techniques on other data sets. We believe that we could achieve even higher accuracy if we were given more attributes of an email, other than the message itself. If we were given elements such as subject, email addresses, and so forth, it may be much easier to identify spam.

While all scores are really high, Naive Bayes has the highest average score (0.981 vs. 0.970 for stochastic descent and 0.974 for perceptron). It performs just as well (0.98) for original text data and lemmatized text, but is faster for original text data. Again, we need to acknowledge that these are all excellent scores however. When it comes to runtime, Naive Bayes is also the fastest algorithm on average (7.11 seconds vs. 7.17 seconds for gradient descent vs. 7.65 seconds for perceptrons). It is interesting to note that the exception here is stemmed text; all the other algorithms are quicker to finish for stemmed text than Naive Bayes.

Naive Bayes is arguably the best model for this data set but what is the runner up model? Perceptron outperforms SDG with an average score difference of 0.004. When it comes to true positives versus false positives, the area under the curve, there is a bigger difference. There are downsides to using Perceptron however. It is clear that the perceptron is a lot slower than the SDG. Specifically, it takes 0.48, half a second, longer to run Perceptron. Both models have excellent performance and are good enough to be used in the real world. The deciding factor is the context of use. If for some reason we are unable to store the model, meaning that we need to train it every time the user opens their inbox, we might want to go with SDG instead of Perceptron. That is, if we can not use Naive Bayes.

While it is great that we have high levels of overall precision, it is worthwhile to dive into the nature of our incorrectly predicted data. Analyzing false positive/false negative ratios makes a lot of sense

since there was a much larger number of “ham” messages than “spam” messages. Perceptron had a false positive rate of around 0.12, gradient descent had a false positive rate of around 0.18, and naive bayes had a false positive rate of around 0.09. This is another argument to use Naive Bayes: it seems less biased to assigning the “ham” tag to any data set. This means that it has found a stronger indicator of “spam” than the other two datasets. False negative rates were around 0.01 across all models. For more details regarding this, please refer to the tables in the “results” section.

Looking across the datasets, it seems that the original text (without lemmetizing or stemming the data) yielded the best average results for our models (0.975 vs 0.9748 for lemmetizing and 0.9746 for stemming). This does not seem like a huge difference, but it makes a large difference when we consider the scale and number of spam emails. It is curious, however, that the raw data set yielded better results than the lemmetized and stemmed datasets, even though lemmatization and stemming is meant to remove inflections and thus heavily reduce the number of features that needed to be considered.

Conclusion

Spam filters have greatly improved since the first spam email was sent in 1978. The various developments in both machine learning and its application to spam filters, such as Paul Graham’s 2002 improvements, have made it easy to get astoundingly close to state of the arts results with basic setups. In this paper, we explored how well we could detect spam ourselves using a Kaggle data set. To get the best possible results, we first prepared our data and segmented it into different data sets. We then set up a pipeline focused on predictive power (grid search) while adjusting for overfitting (k-fold cross validation). For reasons we discussed in the “Model Selection” section, we ran this pipeline for three models: Multinomial Naive Bayes, Stochastic Gradient Descent, and Perceptron. Naturally, we also ran every model once for every data set (original, lemmatized, and stemmed). Ultimately, we got the best score (0.98) and runtime (6.55 seconds) by using Naive Bayes in combination with our original text data set. The other models and data sets gave us comparable results however, leading to the observation that if we did not rely on Naive Bayes, we would choose Stochastic Gradient Descent over Perceptron if we valued speed at the cost of accuracy. Our results are close to state of the art, which we believe to be 0.99 following a conversation with Dr. Jevin West. In the future, we would like to use our approach on other data sets to both verify it and explore if we can do even better with more features, such as email addresses, and actual email data, such as subject line.