Application of Email Spam Filtering Algorithms to SMS Data

Yishu Xue[*]

26 April 2018

Abstract

In this project, multiple popular algorithms for Email spam filtering are implemented on a Short Message Service (SMS) dataset. Different methods for representing the dataset using matrices were attempted. In addition to utilizing only tokens, other characteristics of the message, such as proportion of numbers or capital letters, were explored. The final classification results were presented. Unsupervised learning was also performed on the dataset.

Keywords: Document classification; Feature extraction; Model tuning; Imbalanced data

1 Introduction

The fast development in information technology made communication between people worldwide easier than ever. These advances are always accompanied by challenges. Huge amounts of spam emails and texts are sent everyday. A spam is defined to be *"unwanted communication intended to be delivered to an indiscriminate target, directly or indirectly, notwithstanding measures to prevent its delivery"*. (Cormack, 2008)

While spam filtering technologies have been widely used by major email service providers such as Gmail and Outlook, its application to mobile Short Message Service (SMS) is less pervasive. The iPhone, for example, has an "unprotected" inbox. Anybody who knows your mobile phone number or iCloud account can send you messages without being blocked.

In this project, we look forward to building different classification models on the SMS dataset. Different methods to convert text to matrices were attempted. Helpfulness of features in the dataset, other than the tokens, was studied.

This project is implemented using the both R and the **sklearn** package (Pedregosa et al., 2011) in Python. The rest of this project report is organized as follows: In Section 2, we briefly describe the SMS dataset, and do basic visualizations. Methods to convert text data to sparce matrices are described and implemented in Section @ref{sec:vectorizer}. Results of common classification algorithms are given in Section 4.

2 The SMS Dataset

The dataset is open data from Kaggle.com. The original dataset has two columns, the first column being the labels, and the second column being the message content. All messages are in English language.

[*]yishu.xue@uconn.edu; Ph.D. student at Department of Statistics, University of Connecticut.

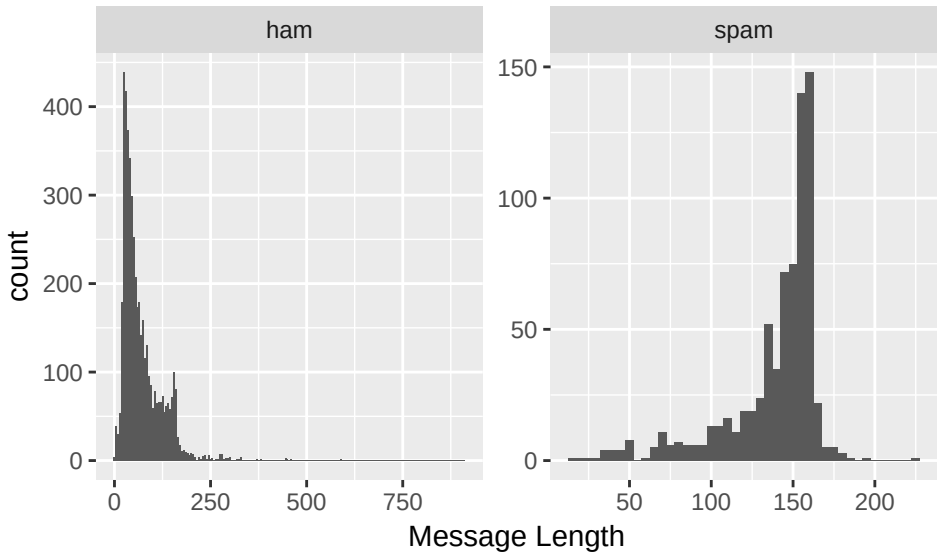

Figure 1: Histogram of message length by type.

There are 5,572 messages in total, with 747 (13.41%) of them being spams, and the other 4,825 (86.59%) being hams. The first five messages are shown below:

```
## [1] "Go until jurong point, crazy.. Available only in bugis n great world la e buffet... Cir
## [2] "Ok lar... Joking wif u oni..."
## [3] "Free entry in 2 a wkly comp to win FA Cup final tkts 21st May 2005. Text FA to 87121 to
## [4] "U dun say so early hor... U c already then say..."
## [5] "Nah I don't think he goes to usf, he lives around here though"
```

The third message, which is longer and contains more numbers than others do, is a spam message. It is of interest whether these characteristics can be generalized and utilized in classifying spams.

Figure 1 indicates that, compared to shorter messages with length less than 125 characters, longer messages with length between 125 and 200 characters have higher probability of being spam. The histogram for hams is positively skewed, while the histogram for spam is negatively skewed.

It can be seen from Figure 2 that, compared to spams, hams, with one exception of purely numbers, tend to have smaller proportions of numbers.

The difference in proportions of capital letters for ham and spam is not quite significant in Figure 3. It is, however, still possible to include it in the set of predictors, and see if it will make contributions to increasing classification accuracy.

3 Convert Text to Sparse Matrices with Vectorizers

The most common way to extract numerical features from text content involves tokenizing, i.e., assigning each token with an unique integer id, counting the occurence of tokens in piece of text content, and sometimes normalizing the rows of the resulting matrix before applying any algorithm. A token can be any piece of text - a word, a phrase, or a sentence. This conversion process is generally called vectorization. Two frequently used vectorizers are the `CountVectorizer` and the

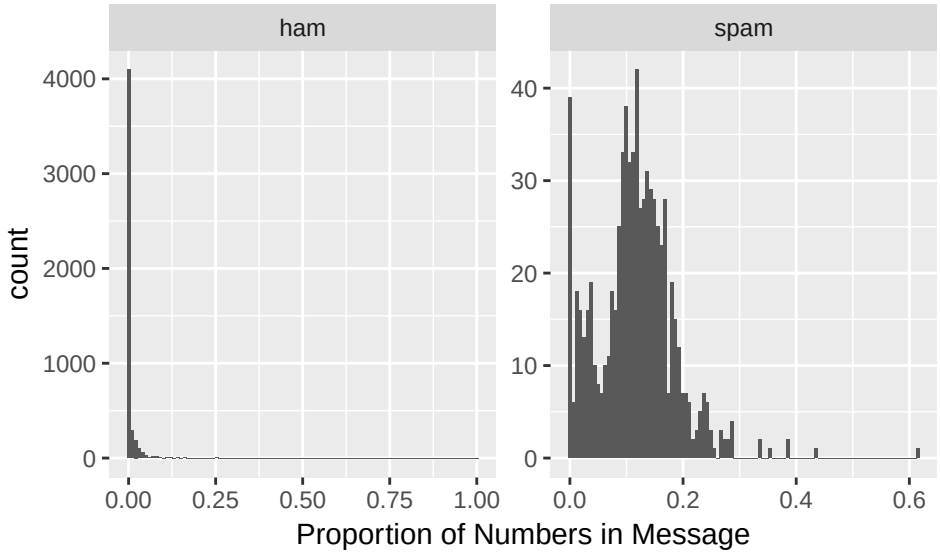

Figure 2: Histogram of proportion of numbers in message by type.

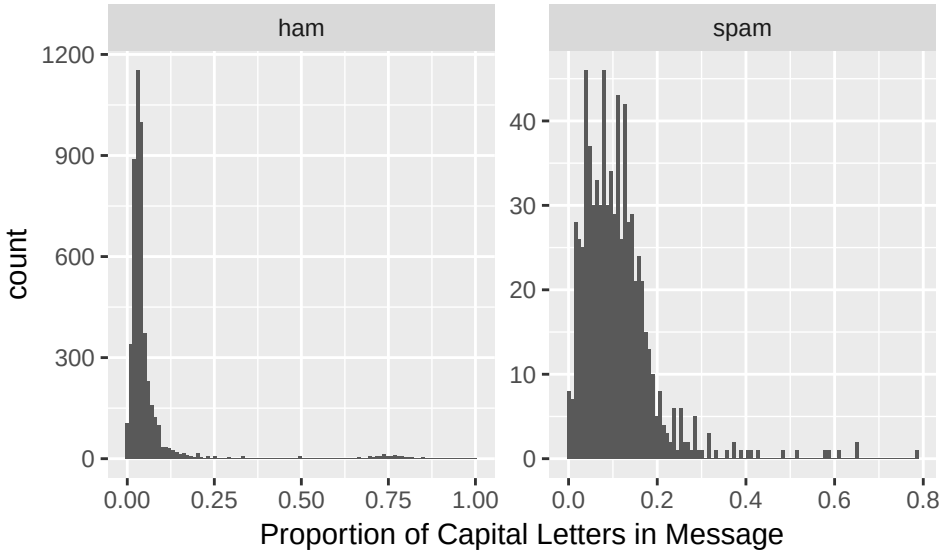

Figure 3: Histogram of proportion of capital letters in message by type.

`TfidfVectorizer`. Both vectorizers are implemented using Python. In this project, due to the relatively small sample size, we only consider single words. For larger documents, it is possibly more meaningful to consider phrases of two or three words.

3.1 `CountVectorizer`

The `CountVectorizer`, as illustrated by its name, counts the frequency of appearance of each token in the document. See the simple example below:

```
vectorizer = CountVectorizer()
exampleCorpus = [
    'This is the first sentence.',
    'This is the second second sentence',
    'And the third one',
    'Is this the first document?'
]
X = vectorizer.fit_transform(exampleCorpus)
print(X.toarray())
```

```
## [[0 0 1 1 0 0 1 1 0 1]
##  [0 0 0 1 0 2 1 1 0 1]
##  [1 0 0 0 1 0 0 1 1 0]
##  [0 1 1 1 0 0 0 1 0 1]]
```

In the output above, each column corresponds to a single word that appeared. The column names are, respectively, "and", "document", "first", "one", "second", "sentence", "the", "third", and "this". The (i, j)th entry in the output matrix is the occurence of the i th token, i.e., word, in the j th item in the corpus.

3.2 `TfidfVectorizer`

The `CountVectorizer` serves as an efficient method to represent a corpus of text using a matrix. It can, however, sometimes be misleading, when the text corpus is large. Some words, such as "the", "a" and "is" in English, will appear quite frequently, and therefore carrying very little information about the actual contents of the document. Feeding the count data directly to a classifier could cause these highly frequent words to shadow the frequencies of rarer, yet more interesting words. It is, therefore, reasonable to consider a weighting strategy, where the weight is decided both by the prevalence of a word in the whole corpus and in a single piece of document (Sparck Jones, 1972).

Suppose we index the document usiing d and the terms using t. `Tfidf` is the product of `tf`, which stands for term-frequency and denote as $tf(t, d)$, and `idf`, which stands for inverse document-frequency, denoted as $idf(t)$.

$tf(t, d)$ is defined to be the frequency of the tth token in the dth document. And using the default settings in Python, $idf(t)$ is defined to be

$$idf(t) = \log \frac{1 + n_d}{1 + df(d, t)} + 1,$$

where n_d is the total number of documents, $df(d,t)$ is the number of documents that contain term t, and the 1's in both the numerator and the denominator part of the logarithm serve as a smoothing parameter. Smoothing can be disabled by specifying `smooth_idf = False` in the options.

The last step of this conversion is normalization. By default, the resulting tf-idf vectors are normalized by the Euclidean norm. It can alternatively normalized by the ℓ_1 norm by setting `norm = 'l1'`.

Using the same example corpus as above, the tf-idf transformed matrix is obtained by `TfidfVectorizer`.

```
vectorizer1 = TfidfVectorizer(norm = "l1")
X1 = vectorizer1.fit_transform(exampleCorpus)
print(X1.toarray())
```

```
## [[0.    0.    0.234 0.189 0.    0.    0.234 0.155 0.    0.189]
##  [0.    0.    0.    0.139 0.    0.436 0.172 0.114 0.    0.139]
##  [0.284 0.    0.    0.    0.284 0.    0.    0.148 0.284 0.   ]
##  [0.    0.279 0.22  0.178 0.    0.    0.    0.145 0.    0.178]]
```

```
vectorizer2 = TfidfVectorizer(norm = "l2")
X2 = vectorizer2.fit_transform(exampleCorpus)
print(X2.toarray())
```

```
## [[0.    0.    0.516 0.418 0.    0.    0.516 0.342 0.    0.418]
##  [0.    0.    0.    0.267 0.    0.837 0.33  0.218 0.    0.267]
##  [0.553 0.    0.    0.    0.553 0.    0.    0.288 0.553 0.   ]
##  [0.    0.608 0.479 0.388 0.    0.    0.    0.317 0.    0.388]]
```

3.3 Application of Vectorizers to the SMS Data

We apply both vectorizers to the SMS data to create the matrix for classification. It is important to notice that, although tf-idf adjusts the matrix for high frequency words, due to the relatively small sample size, it is better if we could remove some words that appear "too frequently".

In the **nltk** (Bird et al., 2009) in Python, a collection of "stopwords" is provided. A stop word is a commonly used word, which is used so frequently that search engines have been programmed to ignore it. The first 10 stop words are printed below. For the full list of stopwords, run `print(stopwords)` instead of `print(stopwords[0:10])`.

```
from nltk.corpus import stopwords
stop_words = stopwords.words("english")
stopwords = [str(stop_words[x]) for x in range(len(stop_words))]
print(stopwords[0:10])
```

```
## ['i', 'me', 'my', 'myself', 'we', 'our', 'ours', 'ourselves', 'you', "you're"]
```

We remove these stopwords from each SMS messgae, and then convert the messages to sparse matrices using the two vectorizers. The row numbers of the resulting matrices will equal the total number of messages in the corpus, and the column numbers will be the number of tokens in the cleaned corpus. The tf-idf transformation used the default option, Euclidean norm, in the

normalization step. The resulting matrices are "sparse" in the sense that while there can be a huge number of tokens from a corpus, a single piece of text could only contain a small portion of them, and therefore the matrix will have many 0 entries.

```
## remove stopwords from text message
def clean_message(text):
   text = text.translate(str.maketrans("", "", string.punctuation))
   text = [word for word in text.split() if word not in stopwords]

   return " ".join(text)
```

```
to_process = sms["Text"].copy()
to_process = to_process.str.lower()
text_cleaned = to_process.apply(clean_message)
vectorizer1 = CountVectorizer("english")
features_count = vectorizer1.fit_transform(text_cleaned)
vectorizer2 = TfidfVectorizer("english", norm = "l2")
features_tfidf = vectorizer2.fit_transform(text_cleaned)
print(type(features_tfidf))
```

```
## <class 'scipy.sparse.csr.csr_matrix'>
```

4 Classification Algorithms, Model Tuning, and Results

The classification algorithms considered include Support Vector Machine, Logistic Regression with elasticnet penalty, Decision Tree, Multinomial Naive Bayes, K-Nearest Neighbors, and four ensemble methods: Random Forest, AdaBoost, Bagging, and ExtraTrees. All algorithms except for ExtraTrees are quite commonly implemented in machine learning applications. They are all provided by the Python **sklearn** package, and can be imported using the following code. The code for the model training and testing part are not quite important and for sake of space we omit it here.

```
from sklearn.linear_model import LogisticRegression
from sklearn.linear_model import SGDClassifier
from sklearn.svm import SVC
from sklearn.svm import LinearSVC
from sklearn.naive_bayes import MultinomialNB
from sklearn.tree import DecisionTreeClassifier
from sklearn.neighbors import KNeighborsClassifier
from sklearn.ensemble import RandomForestClassifier
from sklearn.ensemble import AdaBoostClassifier
from sklearn.ensemble import BaggingClassifier
from sklearn.ensemble import ExtraTreesClassifier
```

ExtraTrees (Geurts et al., 2006), short for "extremely randomized trees", is essentially a more randomized version of Random Forest. In Random Forest, a random subset of features is used, and the algorithm looks for the most discriminative thresholds. In ExtraTrees, however, random thresholds are drawn for each candidate feature, and the best performer is picked as the splitting rule. This procedure, at the cost of introducing a slightly increase in bias, effectively reduces the

Table 1: Performance of selected classifiers on the testing dataset obtained by `CountVectorizer`.

Classifier	Prediction.Accuracy
Support Vector Machine	0.9749
Logistic Regression	0.9839
Decision Tree	0.9695
Multinomial Naive Bayes	0.9767
K-Nearest Neighbor	0.8511
Random Forest	0.9740
AdaBoost	0.9614
Bagging	0.9713
ExtraTrees	0.9803

Table 2: Performance of selected classifiers on the testing dataset obtained by `TfidfVectorizer`.

Classifier	Prediction.Accuracy
Support Vector Machine	0.9300
Logistic Regression	0.9632
Decision Tree	0.9659
Multinomial Naive Bayes	0.9722
K-Nearest Neighbor	0.8924
Random Forest	0.9767
AdaBoost	0.9659
Bagging	0.9776
ExtraTrees	0.9740

variance of the model.

To avoid any additional source of variation in model performance on the feature matrices given by the two vectorizers, the same row indices were used to partition the matrices into training sets of 80% its size, and testing sets of the other 20% size. The training set contained 3,882 (87.1%) hams and 575 (12.9%) spams. The testing set contained the rest 943 (84.6%) hams and 172 (15.6%) spams. The model performance was measured using the proportion of messages in the testing set that were correctly assigned labels, calculated by `accuracy_score` function from **sklearn**.

```
from sklearn.metrics import accuracy_score
```

Each of these models have been tuned to give their best performance on the sparse matrices generated by `CountVectorizer` and `TfidfVectorizer`, respectively. The results are summarized below:

It can be seen from Tables 1 and 2 that Support Vector Machine, Logistic Regression with elasticnet penalty, Decision Tree, Multinomial Naive Bayes and ExtraTrees have better performance with `CountVectorizer`, while the other three ensemble methods, Random Forest, AdaBoost and Bagging perform better on `TfidfVectorizer`.

It is also worth noticing that K-Nearest Neighbor, in either case, has less than 90% accuracy.

Table 3: Performance of selected classifiers on the extended testing dataset.

Classifier	Prediction.Accuracy
Support Vector Machine	0.9812
Logistic Regression	0.9857
Decision Tree	0.9704
Multinomial Naive Bayes	0.9776

Considering the fact that 84.6% of the testing data is in a single ham class, it is not much better than a random guess.

The reason why `TfidfVectorizer` is not outperforming `CountVectorizer` might be due to the relatively small sample size. With 9,376 tokens and 5,572 messages, it is not quite necessary to consider weighting schemes such as tf-idf. Therefore, we consider using `CountVectorizer` in our subsequent analysis. It is also worth noticing that, in terms of accuracy, the ensemble methods are not significantly better than the first three relatively simple methods. Considering the computation efficiency, we use the first four methods in the next section.

5 Assessment of Usefulness of Additional Features

In Section 2, we saw three additional features of messages: length, the proportion of numbers, and the proportion of capital letters. It is of interest whether adding these features to the sparse matrices will help increase classification accuracy or not.

To begin with, all three features are appended to the sparse matrix obtained using `CountVectorizer` as three additiona columns. Same process was repeated, adding two of them at each time. The four chosen algorithms were then run on the extended matrices. It turned out that the addition of `Length` kept distorting all models. A possible explanation could be the fact that there are too many hams compared to spams in both the training and testing sets, and the scale, which is unnormalized, is too large compared to the entries in the original matrix. Normalizing `Length` using its maximum value turned out to be not helping, either, as the longest message has 910 characters, and the normalization suppresses over 97.4% observations into the range (0, 0.2).

The extended matrix was chosen to be the one obtained using `CountVectorizer`, together with two columns for `pnum` and `pcap`. The four classifiers were trained, and their performances on the testing set are presented in Table 3. It can be seen that, including these two additional features brings positive, although small, improvement to all four classifiers.

6 Dealing with Imbalanced Data

With over 85% of the messages being hams, the dataset is imbalanced in nature. Common ways of dealing with imbalanced data include upsampling the relatively smaller group, downsampling the relatively large group, and using ensemble methods with more weak learners. It has, however, been proved in the model tuning step that using a large number of weak learners did not make the ensemble methods better than the simpler ones.

Table 4: Performance of selected classifiers trained on the resampled, more balanced training set.

Classifier	Prediction.Accuracy
Support Vector Machine	0.9758
Logistic Regression	0.9703
Decision Tree	0.9439
Multinomial Naive Bayes	0.9683
K-Nearest Neighbor	0.9626
Random Forest	0.9910
AdaBoost	0.9551
Bagging	0.9616
ExtraTrees	0.9928

Also, in this case, it is more reasonable to downsample the ham class. If we upsample the spam class, we could possibly have multiple messages in the training and testing dataset that are exactly the same. Using information from a message to classify the exact piece of message correctly doesn't mean that the classifier is good. In reality, it is rare that two messages are exactly the same. Therefore, we use the downsampling approach here to cope with imbalance.

We randomly selected 20% (965) of hams and 80% (597) spams from the original dataset, and combined them into a new training set. The rest 3860 hams and 150 spams are combined into a new testing set. This essentially sets the testing benchmark for a "good" classifier to 96.26%, since we could achieve this accuracy simply by labeling every testing message as ham.

The chosen four classifiers were trained on this new training set plus `pnum` and `pcap` for each message, and their performances on the testing set are given below in Table 4.

It is couter-intuitive that with an relatively expanded set of spam messages, Decision Tree is worse than others, not even exceeding the 96.26% benchmark. This could be due to the fact that it is just one learner. We tried the ensemble methods as well, and both Random Forest and ExtraTrees have nearly 100% accuracy.

By comparing the performance of classifiers using training sets that have different proportions of hams and spams, and the type of testing messages that were incorrectly classified, we obtain an interesting observation that, when the proportion of spams is low, it is better to use simpler methods such as Logistic Regression with elastic net and Naive Bayes, while when the proportion of hams is higher, Random Forest or its more randomized cousin, ExtraTrees, become the best choice. This also involves the tradeoff between safety and accuracy when implementing this classifier in reality - do we want to be safer at the cost of having more mislabeled hams, or do we want to receive every useful messgae at the cost of receiving some unsolicited spams? We believe the appropriate proportions of ham and spam in the training set makes an interesting topic for further exploration.

7 Discussions

In this project, we implemented multiple classification algorithms on the SMS spam dataset. Depending on the way that the dataset is vectorized, the classifiers had varying performances. Also,

the accuracy of the classifiers also depend on the proportion of spam and ham messages in the data on which they were trained. In dealing with the imbalanced data, we encountered the tradeoff between being conservative and safe, or otherwise. This is echoed in reality, as sometimes Gmail could mark an useful email as spam and put it in the junk mail box, or allow spams to enter our inboxes.

One limitation to what we did in this project is the relatively small sample size, and the imbalance of ham and spam in the dataset. In reality, if a SMS spam filter is going to be implemented, we expect that more data could be collected to increase its accuracy. Phrases, which have been discussed in the interim report, could also help improving the classification accuray. Due to the small sample size, it was not implemented. Interested readers can refer to Wei Shi's topic presentation for more information regarding the usage of phrases in text mining.

Acknowledgment

We would like to thank Wenjie Wang for providing this nice and delicate template for scientific report writing. Our gratitude also goes to Wei Shi, for many useful discussions during the process of this project.

References

Bird, S., Klein, E., and Loper, E. (2009), *Natural Language Processing with Python: Analyzing Text with the Natural Language Toolkit*, " O'Reilly Media, Inc.".

Cormack, G. V. (2008), "Email Spam Filtering: A Systematic Review," *Foundations and Trends® in Information Retrieval*, 1, 335–455.

Geurts, P., Ernst, D., and Wehenkel, L. (2006), "Extremely Randomized Trees," *Machine Learning*, 63, 3–42.

Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., et al. (2011), "Scikit-learn: Machine Learning in Python," *Journal of Machine Learning Research*, 12, 2825–2830.

Sparck Jones, K. (1972), "A Statistical Interpretation of Term Specificity and Its Application in Retrieval," *Journal of Documentation*, 28, 11–21.