Team 8
Team Leader: Li Yu
Stat 154 Final project
Fall 2016
Due: Dec 2nd 2016, 2pm at 309 Evans

Team Member Name	Contribution:	Signature
Li Yu	1. Supervised Learning Implementation (RF & SVM) 2. Unsupervised Learning Implementation (K-means) 3. Python Scripting	
Mark Beers	1. Python scripting for feature creation 2. Power Feature creation 3. Feature Filtering	
Victor Jin	1. Report write up 2. Power Feature creation 3. Feature Matrix Filtering	
Alex Leu	1. Report write up 2. Power Feature Creation	

Stat 154 Final Project Report

Li Yu[1] , Mark Beers[2], Alex Leu[3], Yoolim Jin[4]
December 2, 2016

I. Introduction

In 2015, controversy arose around Secretary of State Hillary Rodham Clinton's use of her personal email accounts on private servers. As a result of of Freedom of Information (FOI) lawsuits the State department released thousands of redacted emails. The purpose of this paper is to evaluate and implement several statistical machine learning techniques to classify 389 of those emails to their senders with high accuracy using 3505 emails as our training data. The model will developed in Python and rely heavily on *sklearn* and *nltk* packages. Methods used include feature selection, word stemming, and supervised learning techniques taught in class such as support vector machines and random forests as well as unsupervised learning methods such as K-Means Clustering.

II. Describe The Data

The given data set contained 3894 total emails from 5 different senders with the training set comprising 3505 of these. Each email in the training set was labeled with an integer 1-5 representing the sender of the email whereas we were asked to provide labels for the remaining 389 emails in our test set based off our statistical learning methods. The training set contained 40,864 raw features (i.e unique words). The proportional breakdown of emails sent by each sender is given by the following table:

---

[1] 23426019
[2] 23542994
[3] 25502883
[4] 22612692

Training Set Class Distribution

Sender	Total Emails	Percentage
1	685	19.54%
2	1023	29.19%
3	1241	35.43%
4	275	7.85%
5	281	8.02%

Table 1 Training Set Class Distribution

III. Feature Creation

The data was originally parsed into a word-feature matrix where every email was put into a row where each corresponding column represented an instance of a word in the email.This meant given the size of our training set we first created a 3505 x 40,864 word feature matrix. The convenience of a large data set is that we are able to split it into training sets and validation sets. The reasoning for doing so would be to cross-validate and adjust our tuning parameters on the test set then to verify accuracy on the validation set. We used 5-fold cross validation thus 80% of the training data was used as a training set with the remaining 20% being used as our validation set.

IV. Feature Filtering

The downside of having such a large data set is that it would take very long to run through this word feature matrix especially since many of the columns may contain zero; meaning a certain word was never used in the email. Our two major problems to solve were to reduce dimensionality for better interpretability and easier computation, and to find the features

that explains the variability of the model. Presented with this challenge, we first decided it would be best to decrease the dimension of this matrix. In order to do so we first removed stop words which are common words in the English language such as "a", "or", and "thus". These words appear so often that they do not help us in classifying the sender. This was done through Python's *nltk* library and removed 138 columns.

We also performed stemming on the data set using *nltk*'s SnowballStemmer which "[removes] morphological affixes from words"[5]. Essentially this stripped away suffixes and treated words with similar roots as the same. This greatly reduced the size of the matrix by taking away 10140 features.

Next we went beyond the given prompt to further reduce the amount of features. This included taking out punctuation and removing what we called "bad strings" which were bits of text that were included in most emails. Examples of this are how every email began with "unclassified u.s. department of state" or "case no.". There were only 9 "bad strings" but removing them helped greatly in improving prediction accuracy as almost every email contained some combination of these.

We chose to remove words that were repeated less than 6 times because we wanted to find a balance between reducing dimensions for an easier analysis and conserving certain words that are significant in the analysis. This had the greatest effect in reducing the dimension space by removing 23541 feature columns, with the end result of 7,051 predictors.

To find the predictors that explain the variability within the model while further reducing the dimensionality of the feature matrix, we looked into the variance of each word frequency according to its senders. We first created a matrix counting the raw frequency of the words used total per sender, and then standardized each frequency by dividing the total frequency of a word per sender by the total emails that each sender sent. That is, if the word "aaron" was used 5 times by *sender 1* and 6 times by *sender 2*, and each sent 5 and 6 emails respectively, the resulting feature matrix would have 1 and 1 as its value under its column. Then with these numbers we measured the variance of each columns to observe the significance of each word in identifying the top 6,000 important predictors, and eliminate others that were not as significant. The number

[5] http://www.nltk.org/api/nltk.stem.html

6,000 was chosen arbitrarily, and this process would eliminate the top 1,051 variables that did not show much variance and thus would not have much effect to the algorithm.

After adding 6 power features, we ultimately came up with a matrix that had 6,006 variables, and the process of elimination is shown in *Table 2*. One possible way to further reduce dimensionality of our feature matrix is to use Principle Component Analysis (PCA). This would, in practice, might seem a better choice because it would 1) reduce dimensions but not exclude all variables by embedding them to a combination of lower dimension data, and 2) we would be able to choose how many predictors we want in our analysis. But because we thought this would reduce the interpretability of each predictors, we excluded this method in our model.

Step	**Total # of Features**
Raw	40864
Remove Stop Words	40726
Stemming	30586
Bad Strings	30577
Low Frequency Words (<6)	7051
Low Variance Words	6000
Adding Power Features	6006

Table 2 Process of Eliminating Features

V. Power Features

Along with word features we tried some power features which are different in the fact that they do not rely on the individual words. Here is what we did:

- Length of the email
- Average length of individual words in the email
- Standard deviation of the word length in the email
- Presence of a phone number provided
- Whether it contains a certain phrase

For the last power feature listed we found that to be particularly interesting. We found out that in some emails senders often sent more URL links more than others. For example, sender 5, albeit 8% with 281 emails, sent 285 out of 655 total URL links that we identified in the training set. We thought this would be an important identification to add to the feature matrix, and further explored which links were used most frequently among each sender. Thus we added binary classification variables of whether an email contained the words including huffingtonpost (sender 5), guardian (sender 1 and 5), and .state.gov (sender 1, 2 and 3).

In addition to the listed power features we tried a few which included whether the sender linked a phone number and average length of email. The former was not a significant factor in accuracy while the latter did not occur frequently enough to be useful. One which proved to be incredibly useful was the revising of the "bad strings". As noted earlier we removed some of these "bad strings" but later realized that when "benghazi comm" was included in this it was alway sent from either sender 1, 2 or 3, so including this to our feature matrix would further increase the accuracy of our model. Also when we removed punctuation, websites would become a single string. Since it is highly unlikely that the example same website link would be referenced over 6 times so that means every website was discarded in our word feature matrix. Going back and revising this in order to allow websites as a feature we see that in our training set some websites were only referenced by a single sender which aided greatly in helping us classify who sent the email.

VI. Supervised Learning

A. Random Forests

We first considered a random forest model in where we fit n decision trees each on our training data. We trained a random forest classifier by averaging the predictions given by n decision trees on our cross validated set. The average value this model outputs will correspond to a classification on which sender the email came from. We considered only a subset of our 6000+ features at each decision node of our tree. This was an effort to decorrelate the trees which in turn reduces our model's variance. With aid of the *RandomForestClassifer* package from *scikit*

we were able to create a random forest classifier with two parameters; n-estimators and maximum amount of p features used. Because there are so many combinations of both estimators and features to test the way we went about doing this was randomly trying integers of each at certain intervals and then deciding whether to increase or decrease towards the next interval based off our cross-validation error.

To get the best random forest parameters, we first chose an arbitrary set of n and p (ie. [50, 100, 150, 200, 250, 300]), and ran a double for-loop to get all CV results from the combinations of those two parameters. Then, we chose the parameters that yielded the highest CV value, and tested a new list of numbers adjacent to the chosen number to cover a more accurate value for the parameters. For example, if $n = 100$, $p = 200$ were chosen, we would test another list n=[80, 90, 100, 110, 120], p=[180, 190, 200, 210, 220], and expand on those numbers and so on. Some arbitrarily chosen examples are printed out on *Table 3* and *Table 4*:

RF Accuracy Initial Test

Estimators	Features	CV - Value
1	1	37.2%
1	10	38.5%
10	1	48.9%
10	10	54.1%
100	100	67.9%
200	500	70.1%
300	300	69.5%
500	500	69.9%

Table 3 RF CV Accuracy

Narrowed Down RF Parameters

Estimators	Features	CV - Value
90	490	69.99%
95	500	69.99%
100	490	70.3%
100	500	69.8%
105	495	69.9%
105	510	70.41%
110	495	70.44%
110	510	70.0%

Table 4 Narrowed RF CV Accuracy

This returned a random forest classifier with 110 estimators and a maximum number of 495 features in these estimators as our best model. Our top 10 best predictors included the words " 'sid', 'cdm', 'len_email', 'u', 'pm', 'fw', 'fyi', 'subject' " as well as the power features 'avg_word_len' and 'Sd_word_len'. Somehow while performing this process we achieved a training set accuracy of 100% even after cross validating. We believed this to be a result of overfitting, and decided against using this to predict the test data.

The reason chose to use cross validation rather than an out-of-bag (OOB) error was because OOB uses bootstrapping that involves approximately 66% as its training set, and 33% as its test set, whereas a 5-fold cross validation uses 80% as its training set, and 20% as its test set, and thus is somewhat of a more "pessimistic" estimation of the forest's error.

Step	Total # of features	Total Accuracy	Accuracy per sender
RF	**6006**	**100%**	**100% for all**

Table 5 Total RF Accuracy on the Training Set

B. Support Vector Machine

Next we decided to fit our data with a linear kernel support vector machine (SVM). For this method we used the *scikit* package in python, and for our model we used the linear kernel. The multi-class classification methods included in the package are *LinearSVC*, which assumes a linear kernel and implements the "one-vs-the-rest" multi-class strategy[6], and thus training n-class models. The *SVC* and *NuSVC* (which allows a parameter to control the number of support vectors) allows for customization of kernels to be included within the function parameters, but implements the "one-against-one" strategy, and thus $5 * (5 - 1)/2 = 10$ classifiers are constructed, with each one trained from two classes. Thus we chose the linear kernel since it is quicker to train on large data sets with large values of p and is less prone to overfitting the training set, which is a crucial problem since $p > n$ in our dataset. In addition, other models of SVM's don't allow as much interpretability in terms of the weight of each feature.

Within our functions we used a for-loop to tune our error penalty c from 1 to 100 using 5-fold cross validation on our training set. Provided are a few examples of the results of tuning in *Table 6*:

Tuning Parameter (c)	CV - Value
0.002	63.3%
0.01	66.73%
0.15	66.8%
1	61.4%
5	58.5%
10	58.5%
50	57.23%

Table 6 SVM CV Accuracy

[6] http://scikit-learn.org/stable/modules/svm.html

Our best model has the tuning parameter around 0.015 and had a total accuracy of roughly 94.3%. Our top 10 features included the power feature " 'avg_word_len'," with our best words features being 'u', 'sid', 'sent', 'cdm', 'fw', 'b5', 'talk', 'sunday', 'fyi'. The values of each class accuracy is reported in the table below.

Step	Total # of features used	Total Accuracy	Accuracy per sender [1, 2, 3, 4, 5]
SVM	**6006**	**94.29%**	**[89.9%, 93.7%, 97.8%, 89.1%, 96.4%]**

Table 7 SVM Total Accuracy on the Training Set

VII. Unsupervised Learning

The unsupervised method we used was K-means clustering using the *KMeans* package by *scikit*. This first identifies what the five optimal clusters are by using the K-means algorithm. First the data are randomly assigned and labeled into five classifications, and then by identifying the centroid of each class, assigns the same data with a new classes that are closest to the centroid. By using the kmeans function within the package that runs this algorithm, we identified the five clusters.

Since we identified the clusters but not which cluster belonged to which class, the next challenge was to identify and label the clusters into their best fit classes. To do this we created a function that input the training data and tested the accuracy of a certain labeling of the clusters. Then we created an array of $5! = 120$ lists that has permutations of 1 to 5, and tested each of them using the for loop, and labeled the clusters according to the numbers that had the highest accuracy in the training set. The results are in *Table 8*.

The problem with this method is that we are working in a high dimension space (ie. 3505 x 6006) where there are more features than observations. This allows the data to be too "well separated" in their own space which does not work well for our clustering. This resulted in an accurarcy of only around 0.4017. Perhaps a way to improve this test would be to use Principle Component Analysis to further reduce the dimensionality of the data to choose a p such that $p < n$.

Step	Total # of features	Total Accuracy
K-Means	**6006**	**40.17%**

Table 8 K-Means Algorithm Total Accuracy on the Training Set

VIII. Conclusion

Overall we decided that our best model was our SVM model. While it achieved slightly lower accuracy than the random forest model, we believed the RF model to be "too accurate" resulting in a large possibility of overfitting the test set later on. Although the algorithm and functions were conveniently provided online in forms of packages, we not only had to read the documentation to see what parameters are asked and/or can be changed for our models, but also understood what each algorithm does and how they are valid in our modeling. Understanding the packages and the parameters is crucial, as in our initial model (that had a 65% accuracy on the test data) were based on SVM, but instead of choosing the value that had the highest CV value, we mistook it for CV error and chose the value that had the lowest CV value. Trials and errors like these taught us the importance of understanding and accurately implementing functions, and how feedback is important among teammates.