School of Electronic & Computer Engineering
Dublin City University

EE514 Machine Learning & Data Analysis

Coursework Assignment

Table of Contents

1 Introduction *3*

2 Pre-processing *4*

2.1 Data Splits – Training, Validation, and Testing **4**

2.1.1 Retrieve ham and spam emails form Enron Dataset 6

2.1.2 Perform Data Splits 6

2.2 Feature Extraction **8**

3 Exploratory Data Analysis *9*

4 Supervised Classification *12*

4.1 Feature Selection **12**

5 Model Selection *14*

6 Model Evaluation *17*

8 Appendix *19*

8.1 Bibliography **19**

8..2 Table of Figures **20**

1 Introduction

This assignment involves using machine learning and data analysis techniques studied in this module, to allow for the exploration and retrieval of important information content in a large dataset. The dataset used in this assignment in the published Enron Corporation email dataset [1].

The objective of this project is to perform some pre-processing techniques , feature extraction and exploratory data analysis on the dataset. It is hoped that these steps will allow for important information to be retrieved from the dataset, which will allow for the training of a classification model which will classify the emails into spam and non-spam (i.e. ham). The classifier will be training on a training set, validation on a hold-out validation set, and tested on a test set (i.e. test for the out-of-sample-error).

During this assignment, a number of feature extraction techniques will be explored with varying parameters (i.e. Word Count Frequency, TF-IDF, stop word removal). A number of classification models will also be tested (i.e. Naïve Bayes Classifier, Support Vector Classifier, AdaBoost Classifier, Logistic Regression Classifier, etc.). These classifiers will be fitted to the training set, and tested for accuracy, precision, and recall on the validation set. A general metric for deciding on a classifier can be seen below [2].

After various classification models are investigation, the ‘best’ classifier will be carried through to model evaluation where the out-of-sample error on the test set.

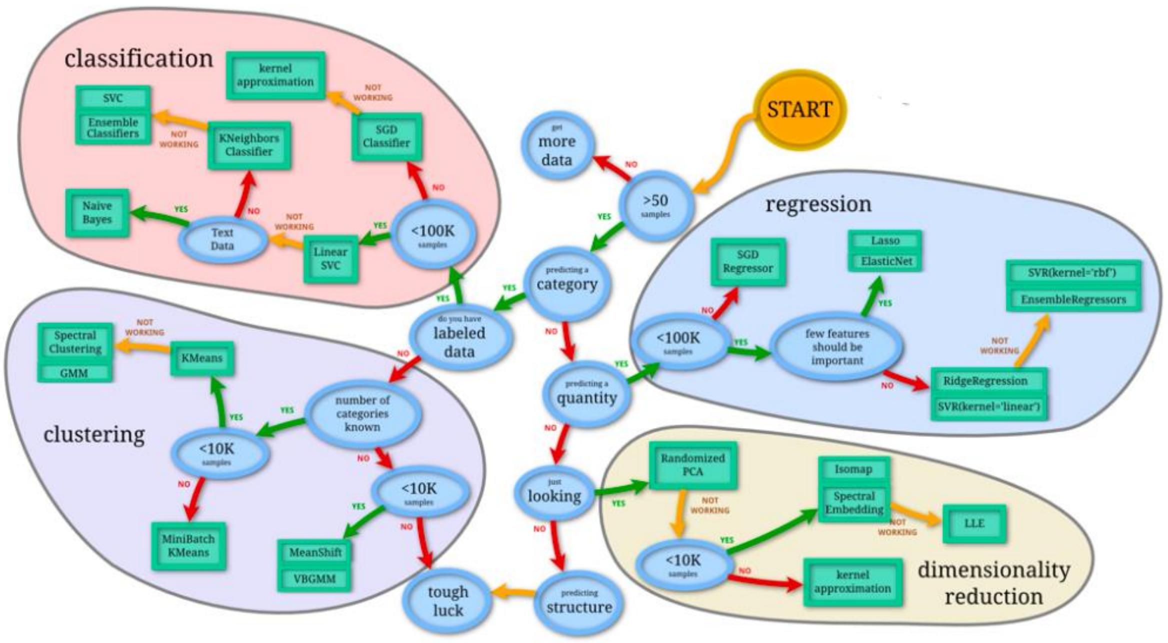


Figure 1: Choosing the right estimator[2]

2 Pre-processing

A number of pre-processing techniques are required before we begin to train and test various classifiers on the data. The emails will be split up into training, validation, and testing subsets, along with being set up in a specific format to allow the data sets to be correctly loaded to Spyder, a Python IDE found in the Anaconda Navigator [3]. The section will also outline various feature extraction techniques investigated in this assignment.

2.1 Data Splits – Training, Validation, and Testing

To properly explain why the data must be divided into training, validation, and testing subsets, we must first discussed how a supervised machine learning algorithm works. To design an algorithm to classify the Enron email dataset into spam and non-spam using supervised classification, the approach outline in Figure 2 below is followed. The dataset in first shuffled (to ensure a randomly distributed data), the data is then split into training and test – each of which takes two separate roots in the algorithm. The training dataset is sent to the learning algorithm, where the computer fits the input x (i.e. an email) to the output y (i.e. spam/ham). This learning uses this input/output data to design a model, which 'should' be able to predict where an email is ham or spam. The term 'should' is in inverted-commas as this may not always be the case. The performance of a model on a particular task is depends on a number of parameters (i.e. noise in the data, bias, etc.).

After fitting a model to the data, the algorithm can use the test set to makes its predictions. At this point the model is used for the first time. Using the predictions obtained from the test set, the algorithm can obtained an 'out-of-sample error estimate', as the test set is pre-processed into spam and ham emails. The out-of-sample error estimate is a percentage score on the accuracy and precision the classifier.

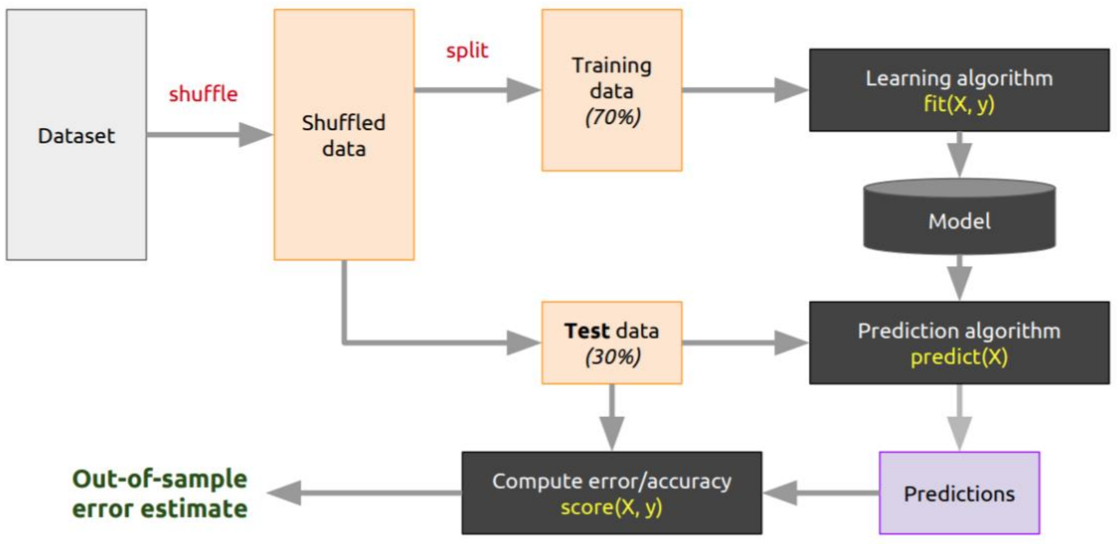

Dataset
shuffle
Shuffled data
split
Training data (70%)
Learning algorithm fit(X, y)
Model
Test data (30%)
Prediction algorithm predict(X)
Predictions
Compute error/accuracy score(X, y)
Out-of-sample error estimate

Figure 2: Supervised Classifier Design Algorithm [4]

Although this is the general approach used in supervised machine learning, commonly use classification models (Logistic Regression, SVM, Naïve Bayes, etc.) have a number on hyperparameters, which alter their functionality and the way they fit data in a number of different ways. One option would be to run the algorithm shown in Figure 2 above with varying hyperparameters, which would result in different out-of-sample errors, although this approach is a bad idea as it would introduce bias to the system. Selecting hyperparameters to minimize errors on the test set means you are overfitting the test set. There are two alternatives that can be considered in this situations: cross-validation, and hold out validation.

Cross validations includes splitting the training data into a number of folds (i.e. chunks), where one chunk is used for validations, i.e. testing, and the other chunks are used for training. This method is repeated so that every chunk is the 'validation' chunk once, and the average error is computed on the system. Cross validation works well on small data sets were you do not want the decrease the size of your training set.

Hold-out validation is useful for big-data (where cross-validation requires larger computation time), and the training data can be reduced with little effect on performance. In this case, the original dataset is split into training, validation, and testing subsets, as seen in Figure 3 below. The training set (50%) is used to fit a model to the input/output data, whereas the validation set is used to test this model for various hyperparameters with a hope to minimize error. The test set is locked away in a box and is not touched until all design, fitting, and testing of the classifier is complete. At this point the model predicts on the test set, and the out-of-sample error is obtained. The validation set provides the designer with an estimate of the accuracy and precision of the model before its application, while the test set provides the actually accuracy and precision of the model.

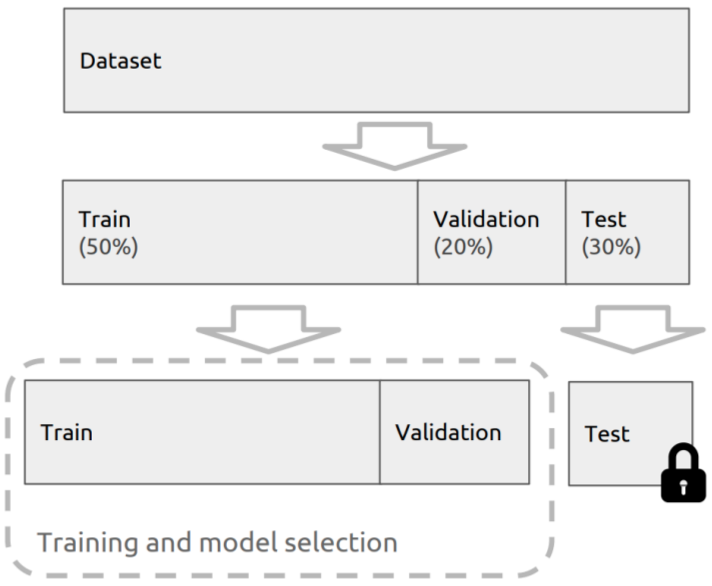
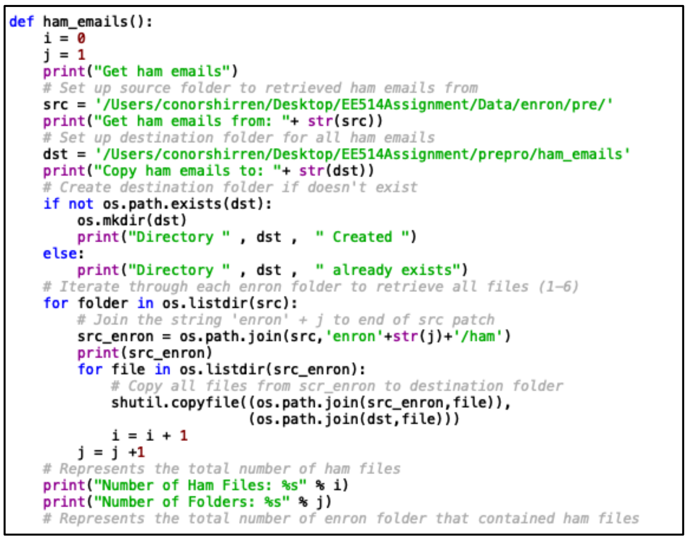


Figure 3: training, testing, and validation subsets [4]

The python script named ‘Preproccesing.py’ was developed to automatically perform all of the file management associated with the data splits for this assignment. This python scripts performed a number of functions:

2.1.1 Retrieve ham and spam emails form Enron Dataset

- Import the ham emails from the Enron dataset folder and place them in a new directory at the following path: '/Users/conorshirren/Desktop/EE514Assignment/prepro/ham_emails'. .
- Import the spam emails from the Enron dataset folder and place them in a new directory at the following path: '/Users/conorshirren/Desktop/EE514Assignment/prepro/spam_emails'
- If these destination paths for the spam and ham emails do not exist, they will be created.

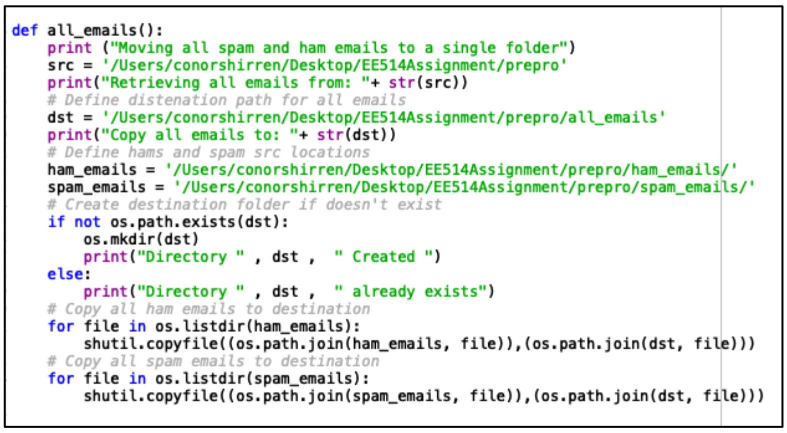


```python
def ham_emails():
    i = 0
    j = 1
    print("Get ham emails")
    # Set up source folder to retrieved ham emails from
    src = '/Users/conorshirren/Desktop/EE514Assignment/Data/enron/pre/'
    print("Get ham emails from: "+ str(src))
    # Set up destination folder for all ham emails
    dst = '/Users/conorshirren/Desktop/EE514Assignment/prepro/ham_emails'
    print("Copy ham emails to: "+ str(dst))
    # Create destination folder if doesn't exist
    if not os.path.exists(dst):
        os.mkdir(dst)
        print("Directory " , dst ,  " Created ")
    else:
        print("Directory " , dst ,  " already exists")
    # Iterate through each enron folder to retrieve all files (1-6)
    for folder in os.listdir(src):
        # Join the string 'enron' + j to end of src patch
        src_enron = os.path.join(src,'enron'+str(j)+'/ham')
        print(src_enron)
        for file in os.listdir(src_enron):
            # Copy all files from scr_enron to destination folder
            shutil.copyfile((os.path.join(src_enron,file)),
                            (os.path.join(dst,file)))
            i = i + 1
        j = j +1
    # Represents the total number of ham files
    print("Number of Ham Files: %s" % i)
    print("Number of Folders: %s" % j)
    # Represents the total number of enron folder that contained ham files
```

Figure 4: Code Used to import all ham emails from enron dataset

2.1.2 Perform Data Splits

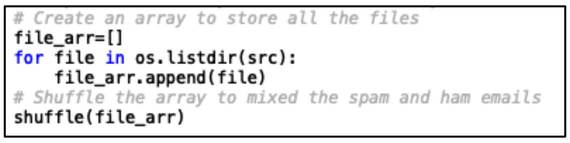
- Take all spam and ham emails and copy them to a new directory at the following path: '/Users/conorshirren/Desktop/EE514Assignment/prepro/all_emails'. If this directory does not exist, it will be created.



```python
def all_emails():
    print ("Moving all spam and ham emails to a single folder")
    src = '/Users/conorshirren/Desktop/EE514Assignment/prepro'
    print("Retrieving all emails from: "+ str(src))
    # Define distenation path for all emails
    dst = '/Users/conorshirren/Desktop/EE514Assignment/prepro/all_emails'
    print("Copy all emails to: "+ str(dst))
    # Define hams and spam src locations
    ham_emails = '/Users/conorshirren/Desktop/EE514Assignment/prepro/ham_emails/'
    spam_emails = '/Users/conorshirren/Desktop/EE514Assignment/prepro/spam_emails/'
    # Create destination folder if doesn't exist
    if not os.path.exists(dst):
        os.mkdir(dst)
        print("Directory " , dst ,  " Created ")
    else:
        print("Directory " , dst ,  " already exists")
    # Copy all ham emails to destination
    for file in os.listdir(ham_emails):
        shutil.copyfile((os.path.join(ham_emails, file)),(os.path.join(dst, file)))
    # Copy all spam emails to destination
    for file in os.listdir(spam_emails):
        shutil.copyfile((os.path.join(spam_emails, file)),(os.path.join(dst, file)))
```

Figure 5: Copy all emails to a single folder

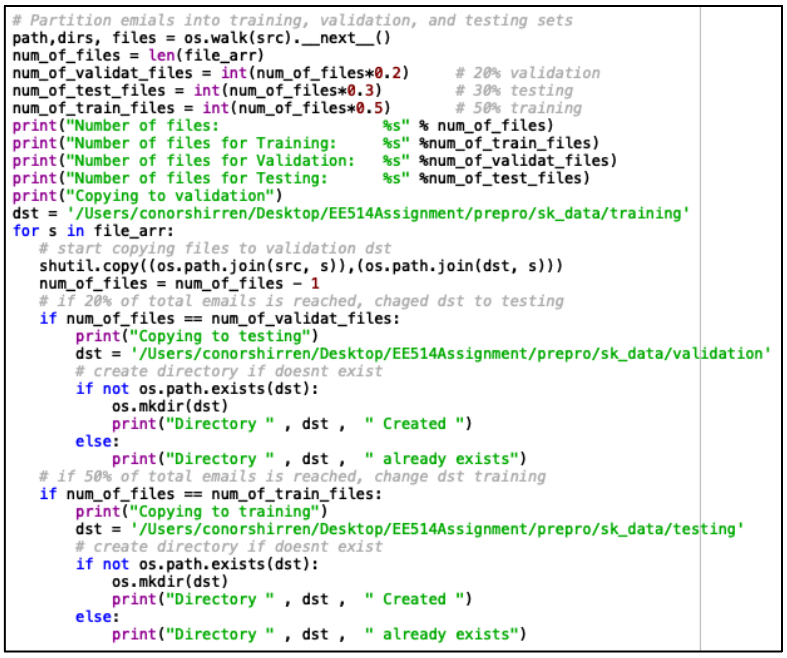
- Create three separate directories for each of the data subsets at the following paths:
  - '/Users/conorshirren/Desktop/EE514Assignment/prepro/sk_data/training'
  - '/Users/conorshirren/Desktop/EE514Assignment/prepro/sk_data/validation'
  - '/Users/conorshirren/Desktop/EE514Assignment/prepro/sk_data/testing'
- Create an array to store each of the files in which is the shuffled to mix up the the hams and spam files



```
# Create an array to store all the files
file_arr=[]
for file in os.listdir(src):
    file_arr.append(file)
# Shuffle the array to mixed the spam and ham emails
shuffle(file_arr)
```

Figure 6: Shuffling the ham and spam emails

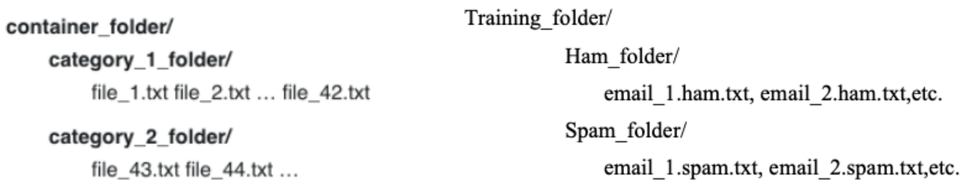
- Develop an algorithm that will automatically divide the shuffled files in file_arr[] into training, validation, and testing – with a respective proportion of 50%-20%-30% (as seen in Figure 3).



```
# Partition emials into training, validation, and testing sets
path,dirs, files = os.walk(src).__next__()
num_of_files = len(file_arr)
num_of_validat_files = int(num_of_files*0.2)     # 20% validation
num_of_test_files = int(num_of_files*0.3)        # 30% testing
num_of_train_files = int(num_of_files*0.5)       # 50% training
print("Number of files:                 %s" % num_of_files)
print("Number of files for Training:    %s" %num_of_train_files)
print("Number of files for Validation:  %s" %num_of_validat_files)
print("Number of files for Testing:     %s" %num_of_test_files)
print("Copying to validation")
dst = '/Users/conorshirren/Desktop/EE514Assignment/prepro/sk_data/training'
for s in file_arr:
   # start copying files to validation dst
   shutil.copy((os.path.join(src, s)),(os.path.join(dst, s)))
   num_of_files = num_of_files - 1
   # if 20% of total emails is reached, chaged dst to testing
   if num_of_files == num_of_validat_files:
       print("Copying to testing")
       dst = '/Users/conorshirren/Desktop/EE514Assignment/prepro/sk_data/validation'
       # create directory if doesnt exist
       if not os.path.exists(dst):
           os.mkdir(dst)
           print("Directory " , dst ,  " Created ")
       else:
           print("Directory " , dst ,  " already exists")
   # if 50% of total emails is reached, change dst training
   if num_of_files == num_of_train_files:
       print("Copying to training")
       dst = '/Users/conorshirren/Desktop/EE514Assignment/prepro/sk_data/testing'
       # create directory if doesnt exist
       if not os.path.exists(dst):
           os.mkdir(dst)
           print("Directory " , dst ,  " Created ")
       else:
           print("Directory " , dst ,  " already exists")
```

Figure 7: Splitting the dataset into training, testing and validation

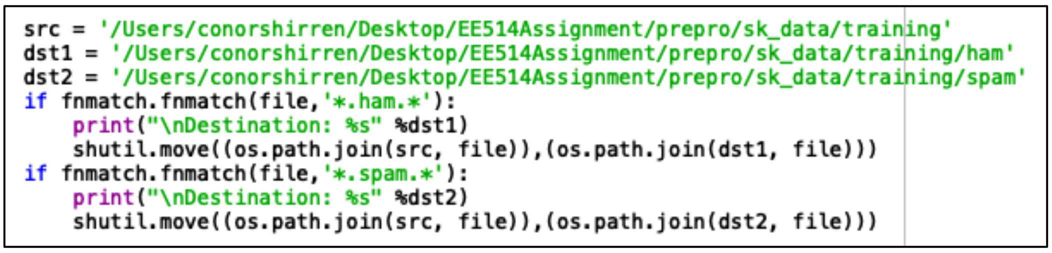
- Once the files are split into their respective subsets, they need to be re-split into ham and spam folder. This format is required as per the function 'sklearn.datasets.load_files' [5], as can be seen below



```
container_folder/
    category_1_folder/
        file_1.txt file_2.txt … file_42.txt
    category_2_folder/
        file_43.txt file_44.txt …
```

```
Training_folder/
    Ham_folder/
        email_1.ham.txt, email_2.ham.txt,etc.
    Spam_folder/
        email_1.spam.txt, email_2.spam.txt,etc.
```

Figure 8: Format required to load files in sklearn[5]



```
src = '/Users/conorshirren/Desktop/EE514Assignment/prepro/sk_data/training'
dst1 = '/Users/conorshirren/Desktop/EE514Assignment/prepro/sk_data/training/ham'
dst2 = '/Users/conorshirren/Desktop/EE514Assignment/prepro/sk_data/training/spam'
if fnmatch.fnmatch(file,'*.ham.*'):
    print("\nDestination: %s" %dst1)
    shutil.move((os.path.join(src, file)),(os.path.join(dst1, file)))
if fnmatch.fnmatch(file,'*.spam.*'):
    print("\nDestination: %s" %dst2)
    shutil.move((os.path.join(src, file)),(os.path.join(dst2, file)))
```

Figure 9: Code used to partition training data into ham and spam

- This step is repeated for each of the data subsets. File management is now complete.

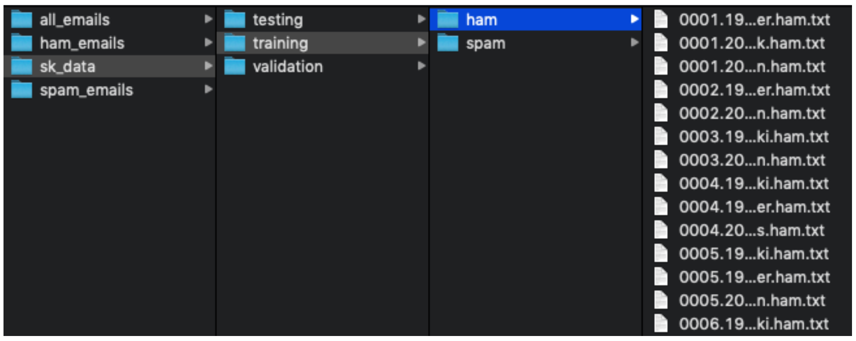
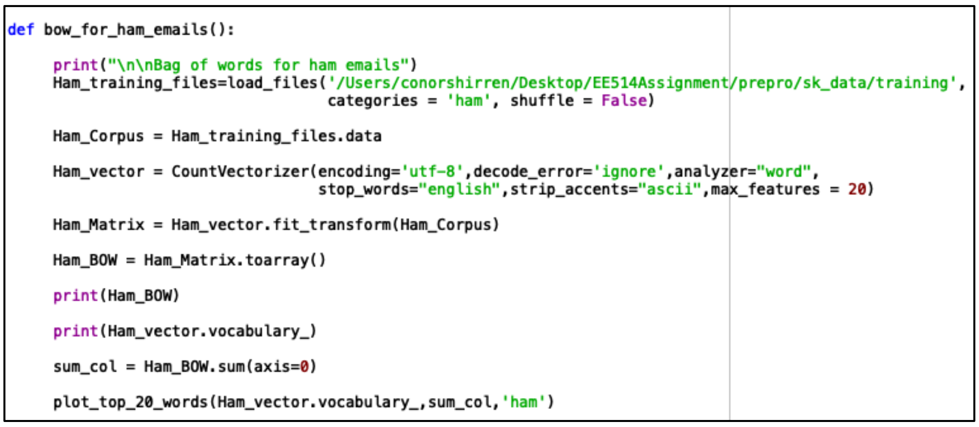


Figure 10: Current Directories

2.2 Feature Extraction

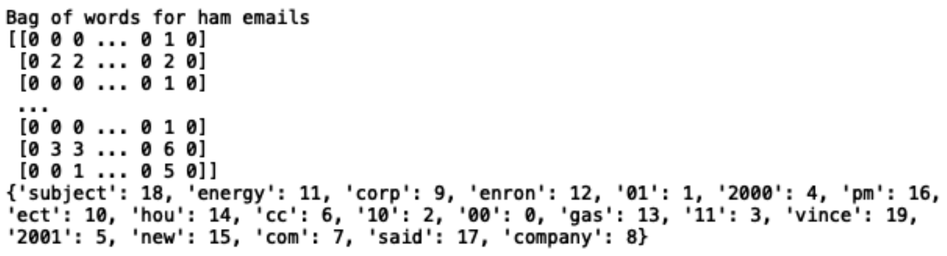
For these text files to be classified, they must be transform into the a format that can be used by the classifiers provided in the sklearn libraries [6]. To do this, the training data is transform into a bag of words, a representation of the vocabulary in the text, which stored useful information content. A bag of words ignores grammar, language and even sentence structure, but store the frequency of a word in a document corpus (i.e. a collection of text files), along with the complete vocabulary of the corpus. The code snippet shown in Figure 11 below shows how to transform a corpus into a bag of words.



```
def bow_for_ham_emails():

    print("\n\nBag of words for ham emails")
    Ham_training_files=load_files('/Users/conorshirren/Desktop/EE514Assignment/prepro/sk_data/training',
                                  categories = 'ham', shuffle = False)

    Ham_Corpus = Ham_training_files.data

    Ham_vector = CountVectorizer(encoding='utf-8',decode_error='ignore',analyzer="word",
                                 stop_words="english",strip_accents="ascii",max_features = 20)

    Ham_Matrix = Ham_vector.fit_transform(Ham_Corpus)

    Ham_BOW = Ham_Matrix.toarray()

    print(Ham_BOW)

    print(Ham_vector.vocabulary_)

    sum_col = Ham_BOW.sum(axis=0)

    plot_top_20_words(Ham_vector.vocabulary_,sum_col,'ham')
```

Figure 11: Bag of Words for ham emails

The output from the code snippet in Figure 11 display the bag of words along with the vocabulary of the of the corpus. For the purpose of testing, the CountVectorizer hyperparameter ‘max_features’ was set to 20 to display a sample vocabulary with the top 20 words in the corpus. This can be seen below.



```
Bag of words for ham emails
[[0 0 0 ... 0 1 0]
 [0 2 2 ... 0 2 0]
 [0 0 0 ... 0 1 0]
 ...
 [0 0 0 ... 0 1 0]
 [0 3 3 ... 0 6 0]
 [0 0 1 ... 0 5 0]]
{'subject': 18, 'energy': 11, 'corp': 9, 'enron': 12, '01': 1, '2000': 4, 'pm': 16,
'ect': 10, 'hou': 14, 'cc': 6, '10': 2, '00': 0, 'gas': 13, '11': 3, 'vince': 19,
'2001': 5, 'new': 15, 'com': 7, 'said': 17, 'company': 8}
```

Figure 12: Bag of Words and Vocabulary

3 Exploratory Data Analysis

To investigate this vocabulary further, the count frequency of the top 20 words were plotted using the defined function ‘plot_top_20_words()’ in the ‘BagOfWords.py’ python scripts. This plot can be seen below.

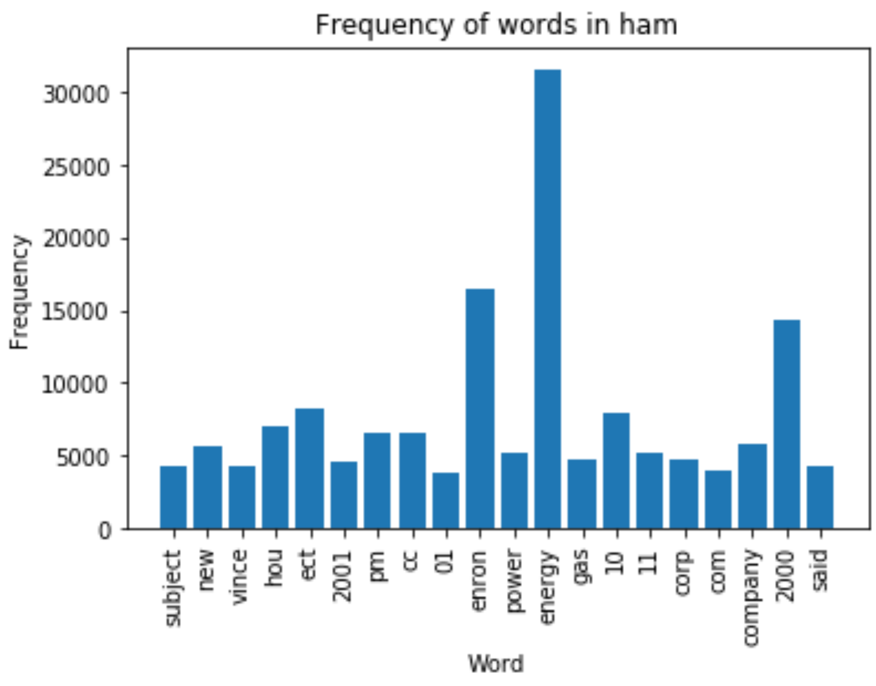


Figure 13: Word Frequency Plot of ham training set

It can be seen from the plot that common words that appear across the corpus include ‘energy’, ‘enron’, ’gas’, ’power’, which can be assumed to be all company related terms as Enron was an American energy provider. The same function was used to plot the top 20 words that appear in the spam corpus, which can be seen in Figure 14 below.

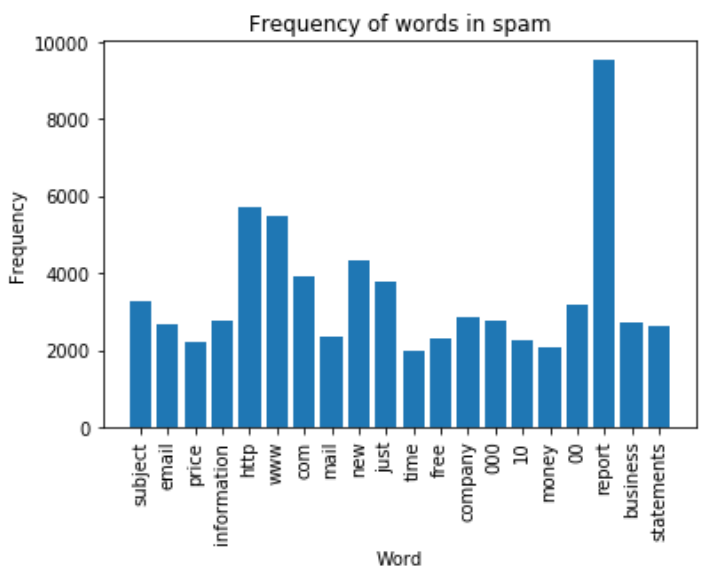


Figure 14: Word Frequency Plot in spam training set

It can be seen from the bar plot that a number of the words that commonly appear in spam email are representative of what you might consider spam. Words like 'free', 'price', and 'money' are all common terms found in spam. Also, it would be normal for a spam email to promoting/suggest going to a website via a URL, some common URL terms can be seen in the plot (i.e. http, www, com, etc.). The three words '10', '00', and '000' may also represent the inclusion of money in an email (i.e. $10,000) as the CountVectorizer would ignore punctuation and non-alphanumeric characters, leaving 10 000.

Along with explore word frequency across the corpus, difference in email length between ham and spam can also be obtained. This plot can be seen in Figure 15 below . It can be seen from Figure 15 that, on average, ham emails are long than spam emails – which is representative of the nature of spam emails (i.e. short advertisements, ads, etc.).

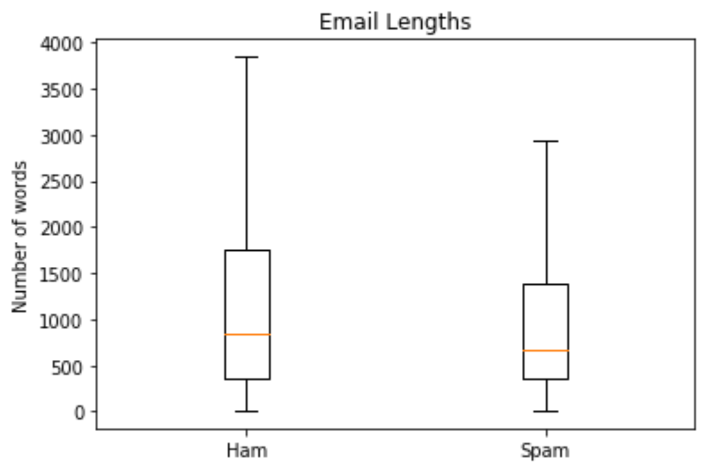
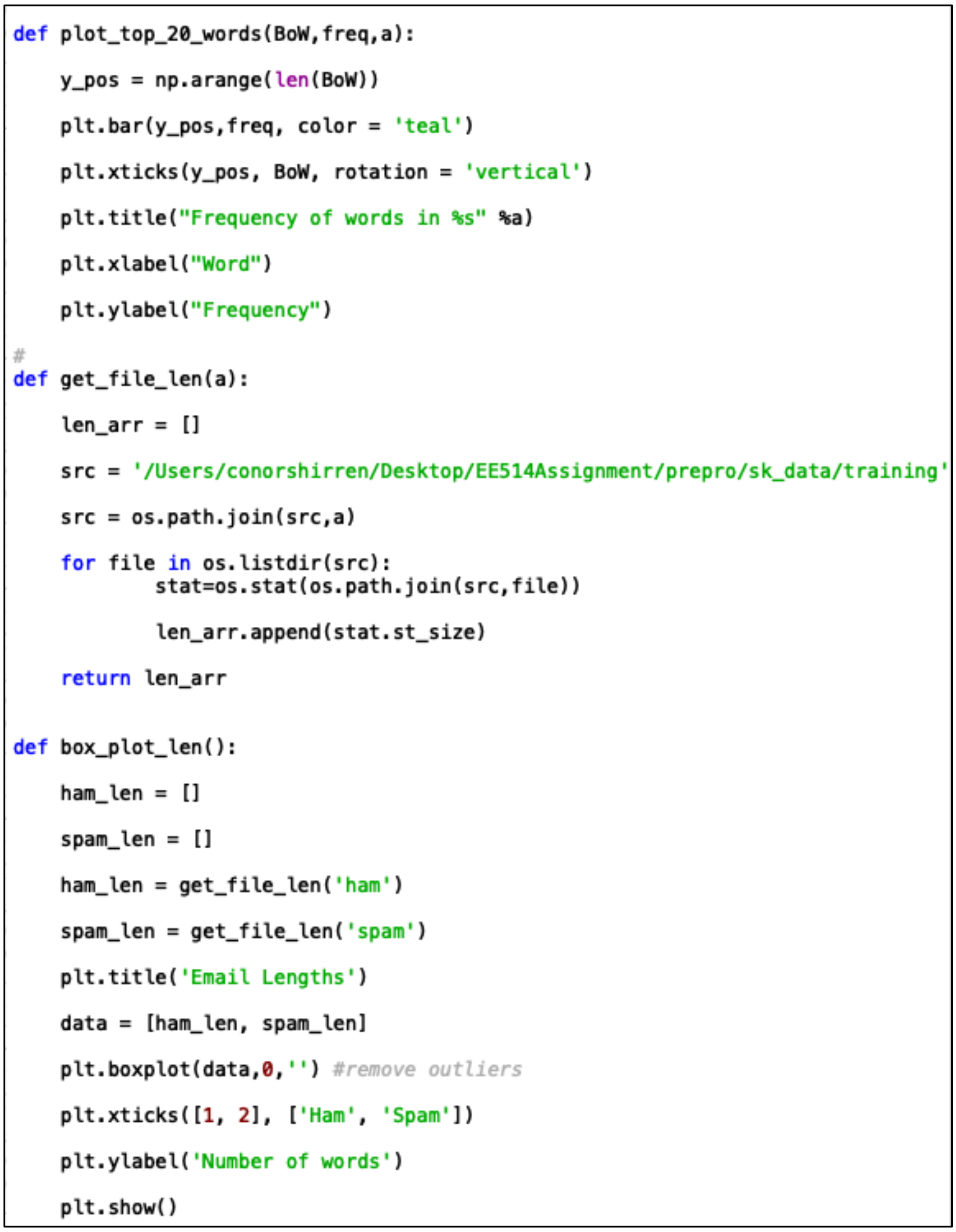


Figure 15: Email Length in Ham and Spam

The code for this exploratory data analysis can be seen below.



```
def plot_top_20_words(BoW,freq,a):

    y_pos = np.arange(len(BoW))

    plt.bar(y_pos,freq, color = 'teal')

    plt.xticks(y_pos, BoW, rotation = 'vertical')

    plt.title("Frequency of words in %s" %a)

    plt.xlabel("Word")

    plt.ylabel("Frequency")

#
def get_file_len(a):

    len_arr = []

    src = '/Users/conorshirren/Desktop/EE514Assignment/prepro/sk_data/training'

    src = os.path.join(src,a)

    for file in os.listdir(src):
            stat=os.stat(os.path.join(src,file))

            len_arr.append(stat.st_size)

    return len_arr


def box_plot_len():

    ham_len = []

    spam_len = []

    ham_len = get_file_len('ham')

    spam_len = get_file_len('spam')

    plt.title('Email Lengths')

    data = [ham_len, spam_len]

    plt.boxplot(data,0,'') #remove outliers

    plt.xticks([1, 2], ['Ham', 'Spam'])

    plt.ylabel('Number of words')

    plt.show()
```

Figure 16: Code for Box Plots and Bar Charts

4 Supervised Classification

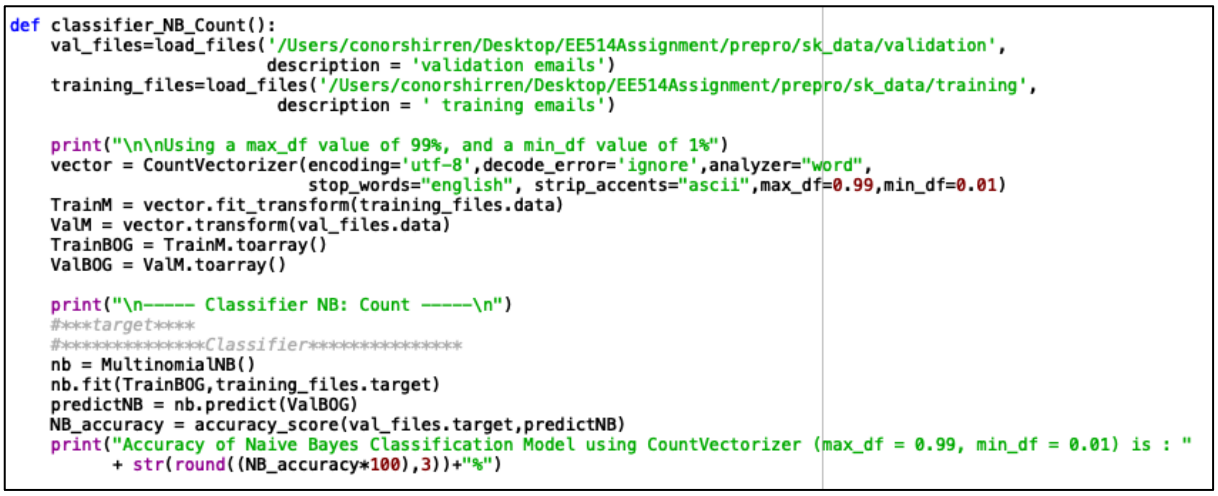
At this point, we are ready to starting fitting the input/output data to a model. To do this a number of steps are required, some of which have already been performed in the ‘Exploratory Data Analysis’ section of this report. This section perform fitting of the data on a single classifier, while varying certain features to try and improve the performance of the classifier. It is hoped that this section will provide the optimal features to be used when carrying out model selection where we will investigate a number classifiers.

4.1 Feature Selection

The first step in feature selection is to select a classifier to perform out tests on. The classifier that will be used in this section is the Naïve Bayes Classification Model [7], which apply Bayes Theorem with naïve independence assumptions. For example, a common spam associated word like ‘free’ contributes to the probability that an email containing it is spam, however, a ham email could also contain the word ‘free’, balance out with words more associated with the ham category (i.e. ‘Enron’, ‘energy’, ‘gas’, etc.) [8]. The Naïve Bayes Classifier is import form the ‘sklearn.naive_bayes’ as is called as ‘MultinominalNB()’.

The feature selection process will require a number of steps:

- Loading both training and validation files
- Create a Bag of Words as seen in Figure 12 using ‘CountVectorizer’ (in this case, a max number of features was not specified)
- Fit the training data to classifier
- Predict on the validation training data. The accuracy of the model can then be obtained.



```
def classifier_NB_Count():
    val_files=load_files('/Users/conorshirren/Desktop/EE514Assignment/prepro/sk_data/validation',
                         description = 'validation emails')
    training_files=load_files('/Users/conorshirren/Desktop/EE514Assignment/prepro/sk_data/training',
                          description = ' training emails')

    print("\n\nUsing a max_df value of 99%, and a min_df value of 1%")
    vector = CountVectorizer(encoding='utf-8',decode_error='ignore',analyzer="word",
                             stop_words="english", strip_accents="ascii",max_df=0.99,min_df=0.01)
    TrainM = vector.fit_transform(training_files.data)
    ValM = vector.transform(val_files.data)
    TrainBOG = TrainM.toarray()
    ValBOG = ValM.toarray()

    print("\n----- Classifier NB: Count -----\n")
    #***target****
    #**************Classifier***************
    nb = MultinomialNB()
    nb.fit(TrainBOG,training_files.target)
    predictNB = nb.predict(ValBOG)
    NB_accuracy = accuracy_score(val_files.target,predictNB)
    print("Accuracy of Naive Bayes Classification Model using CountVectorizer (max_df = 0.99, min_df = 0.01) is : "
          + str(round((NB_accuracy*100),3))+"%")
```

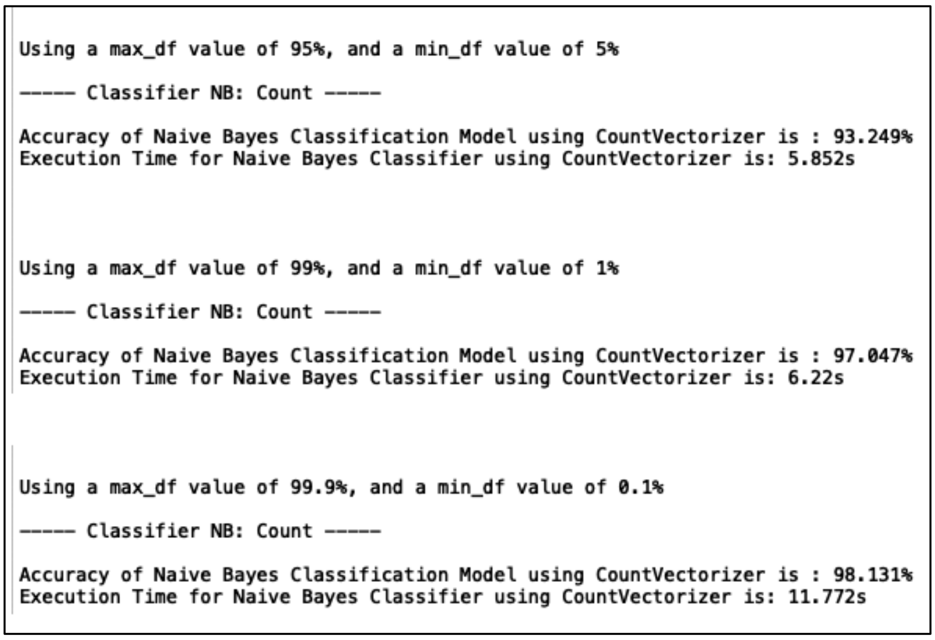
Figure 17: Code used for training and testing the Naive Bayes Classifier

This process was completed multiple times in the "FeatureSelection.py' Python script with a hope to reduce the error in the model. The classifier was tested using both CountVectorizer and TfidfVectorizer. The TF-IDF Vectorizer performs all of the functionality of the CountVectorizer, while also taking into account the inverse document frequency. TF-IDF can be useful as it can Isolate words that are used frequently in a single document (signifying importance to a particular email) while giving less importance to documents to words that appear frequency across the entire corpus (meaning they do not offer important information content to a particular email, and are just words commonly used in the English language).

The Naïve Bayes classifier was training used three differen feature values:

- max_df = 95% , min_df = 5%
- max_df = 99% , min_df = 1%
- max_df = 99.9% , min_df = 0.1%

The following python console was obtained, showing an increase in accuracy in classifier performance, although subsequently an increase in execution time for the model.

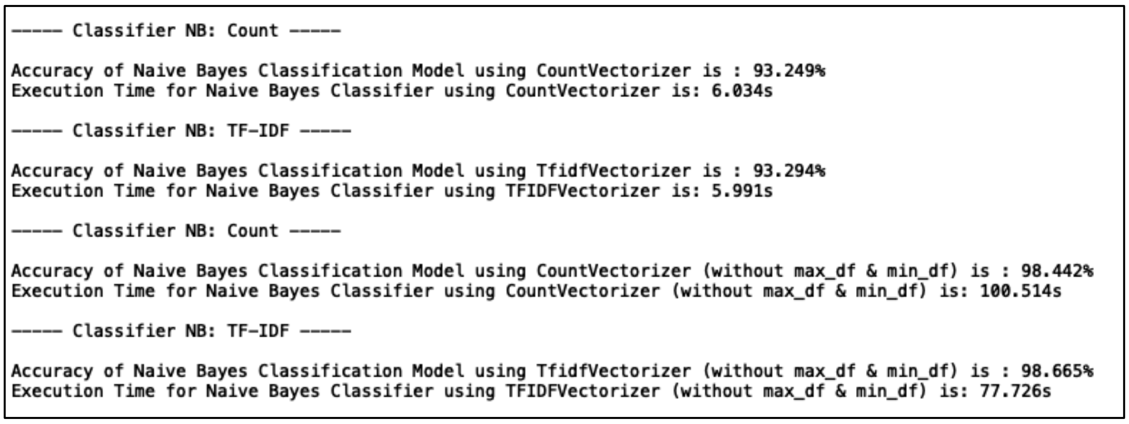


```
Using a max_df value of 95%, and a min_df value of 5%

----- Classifier NB: Count -----

Accuracy of Naive Bayes Classification Model using CountVectorizer is : 93.249%
Execution Time for Naive Bayes Classifier using CountVectorizer is: 5.852s



Using a max_df value of 99%, and a min_df value of 1%

----- Classifier NB: Count -----

Accuracy of Naive Bayes Classification Model using CountVectorizer is : 97.047%
Execution Time for Naive Bayes Classifier using CountVectorizer is: 6.22s



Using a max_df value of 99.9%, and a min_df value of 0.1%

----- Classifier NB: Count -----

Accuracy of Naive Bayes Classification Model using CountVectorizer is : 98.131%
Execution Time for Naive Bayes Classifier using CountVectorizer is: 11.772s
```

Figure 18: CountVectorizer with varying max_df and min_df

This may be problematic for very large training datasets. There must be a trade-off between performance and training time when designing a classification model.

These tests were also carried out on the TfidfVectorizer to compare the performance of Count v Tfidf. The results of this test can be seen in Figure 19. The model is classified using the default *max_df/min_df* values of 0.95/0.05, along with using the complete data set by not specifying a *max_df* or *min_df*. Setting a max_df value of 95% tells the vectorizer to not include words that appear in 95% of the corpus, and likewise to ignore words that appear in less thn 5% of the corpus. It can be seen in Figure 19 below that there is little difference between Count and Tfidf Vectorizers when using 95/5 setting. Utilizing the full dataset by not specifying max/min df values returns in a more accurate model, although the training time for the classifier greatly increased. For this reason, it has been decided to move onto the model selection phase, when the data will be transformed to a bag of words using CountVectorizer, using a max/min df pairing of 99%-1%.



```
----- Classifier NB: Count -----

Accuracy of Naive Bayes Classification Model using CountVectorizer is : 93.249%
Execution Time for Naive Bayes Classifier using CountVectorizer is: 6.034s

----- Classifier NB: TF-IDF -----

Accuracy of Naive Bayes Classification Model using TfidfVectorizer is : 93.294%
Execution Time for Naive Bayes Classifier using TFIDFVectorizer is: 5.991s

----- Classifier NB: Count -----

Accuracy of Naive Bayes Classification Model using CountVectorizer (without max_df & min_df) is : 98.442%
Execution Time for Naive Bayes Classifier using CountVectorizer (without max_df & min_df) is: 100.514s

----- Classifier NB: TF-IDF -----

Accuracy of Naive Bayes Classification Model using TfidfVectorizer (without max_df & min_df) is : 98.665%
Execution Time for Naive Bayes Classifier using TFIDFVectorizer (without max_df & min_df) is: 77.726s
```

Figure 19: Count v Tfidf Vectorizer with and without max_df/min_df

5 Model Selection

In this section, various classifiers will be investigated using the features determined in section 4 of this report. It is hoped that varying the parameters different classification models, the system which yields the lowest error will be obtained. The following classification model will be used in this section: Support Vector Classifier, Naïve Bayes Classifier, AdaBoost Classifier , Logistic Regression Classifier.

Each of these classifiers are trained using the same approach as in section 4, although in this section the hyperparameters of the classifier will be varied. Figure 23 below displays the performance in terms of accuracy, precision, recall and f-score. Accuracy and precision can be compared in the following figure.

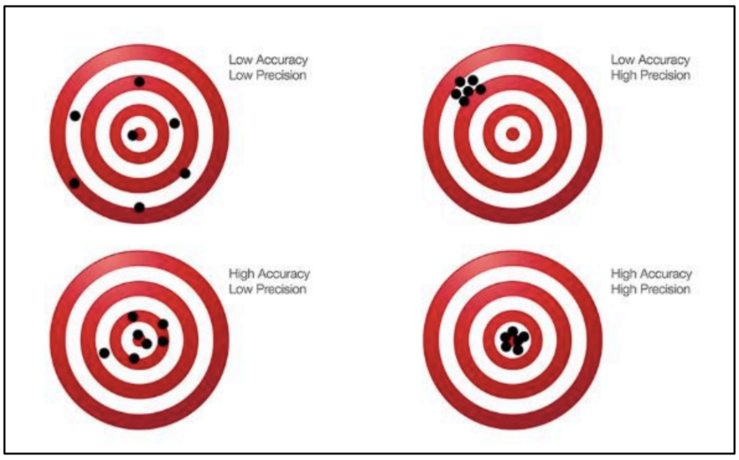


Figure 20: Accuracy v Precision[9]

Recall is similar to precision, as it calculates how many actual positives our model captures through labelling positive, whereas precision calculates how precise our model is out of those predicted positive.

$$\text{Precision} = \frac{True\ Positive}{True\ Positive + False\ Positive}$$

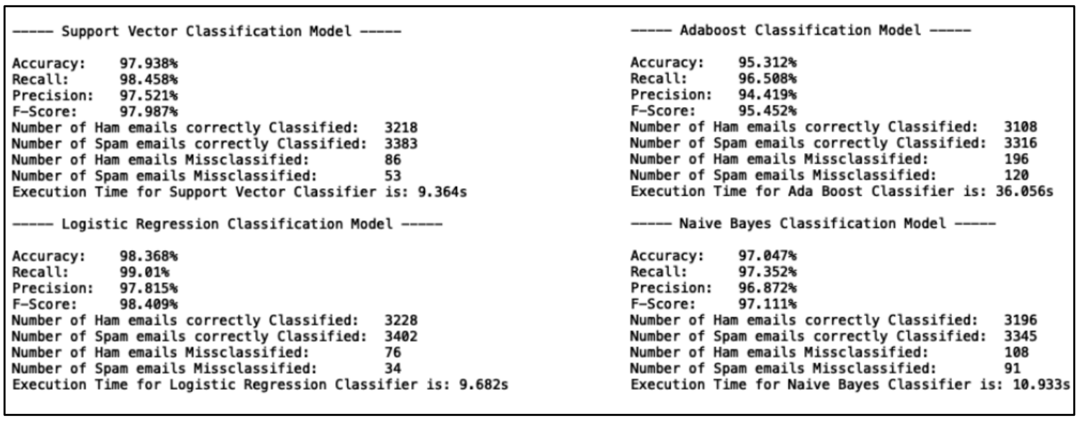
$$\text{Recall} = \frac{True\ Positive}{True\ Positive + False\ Negative}$$

Figure 21: Precision and Recall[10]

The F1-score is a function of both precision and recall, which aims to find a balance between both functions. The F1-score of a classifier provides a better estimate of its performance than its Accuracy score, since accuracy can be largely contributed by a large number of true negatives.

$$\text{F1} = 2 \times \frac{Precision * Recall}{Precision + Recall}$$

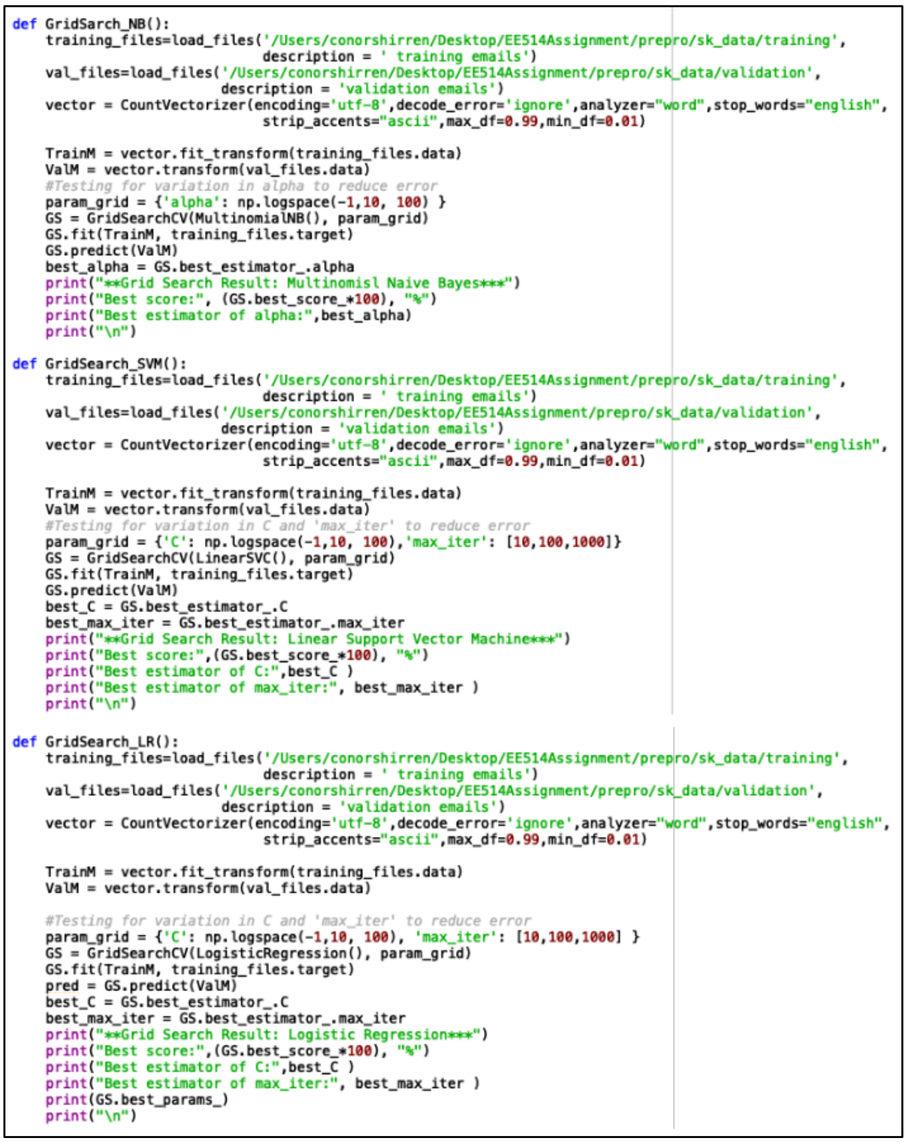
Figure 22: F1-Score as a peformance estimator



```
----- Support Vector Classification Model -----                 ----- Adaboost Classification Model -----

Accuracy:     97.938%                                          Accuracy:     95.312%
Recall:       98.458%                                          Recall:       96.508%
Precision:    97.521%                                          Precision:    94.419%
F-Score:      97.987%                                          F-Score:      95.452%
Number of Ham emails correctly Classified:   3218              Number of Ham emails correctly Classified:   3108
Number of Spam emails correctly Classified:  3383              Number of Spam emails correctly Classified:  3316
Number of Ham emails Missclassified:         86                Number of Ham emails Missclassified:         196
Number of Spam emails Missclassified:        53                Number of Spam emails Missclassified:        120
Execution Time for Support Vector Classifier is: 9.364s        Execution Time for Ada Boost Classifier is: 36.056s

----- Logistic Regression Classification Model -----           ----- Naive Bayes Classification Model -----

Accuracy:     98.368%                                          Accuracy:     97.047%
Recall:       99.01%                                           Recall:       97.352%
Precision:    97.815%                                          Precision:    96.872%
F-Score:      98.409%                                          F-Score:      97.111%
Number of Ham emails correctly Classified:   3228              Number of Ham emails correctly Classified:   3196
Number of Spam emails correctly Classified:  3402              Number of Spam emails correctly Classified:  3345
Number of Ham emails Missclassified:         76                Number of Ham emails Missclassified:         108
Number of Spam emails Missclassified:        34                Number of Spam emails Missclassified:        91
Execution Time for Logistic Regression Classifier is: 9.682s   Execution Time for Naive Bayes Classifier is: 10.933s
```

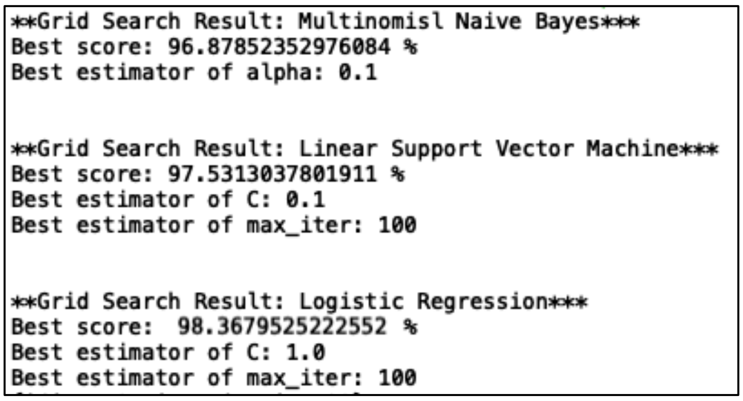
Figure 23: Performance of default classifiers

It can be seen from Figure 23 above that SVM, NB, and LR yield the best performance metrics on the default hyperparameters. These three classifiers will be test varying hyperparameters with a hope to further reduce error in the model. To do this, the sklearn.grid_search function GridSearchCV [11] will be used to automatically vary hyperparameters to reduce error. The code used run this GridSearchCV function can be found in the 'GridSearching.py' python script and can be seen in the figure below.



```python
def GridSarch_NB():
    training_files=load_files('/Users/conorshirren/Desktop/EE514Assignment/prepro/sk_data/training',
                              description = ' training emails')
    val_files=load_files('/Users/conorshirren/Desktop/EE514Assignment/prepro/sk_data/validation',
                         description = 'validation emails')
    vector = CountVectorizer(encoding='utf-8',decode_error='ignore',analyzer="word",stop_words="english",
                              strip_accents="ascii",max_df=0.99,min_df=0.01)

    TrainM = vector.fit_transform(training_files.data)
    ValM = vector.transform(val_files.data)
    #Testing for variation in alpha to reduce error
    param_grid = {'alpha': np.logspace(-1,10, 100) }
    GS = GridSearchCV(MultinomialNB(), param_grid)
    GS.fit(TrainM, training_files.target)
    GS.predict(ValM)
    best_alpha = GS.best_estimator_.alpha
    print("**Grid Search Result: Multinomisl Naive Bayes***")
    print("Best score:", (GS.best_score_*100), "%")
    print("Best estimator of alpha:",best_alpha)
    print("\n")

def GridSearch_SVM():
    training_files=load_files('/Users/conorshirren/Desktop/EE514Assignment/prepro/sk_data/training',
                              description = ' training emails')
    val_files=load_files('/Users/conorshirren/Desktop/EE514Assignment/prepro/sk_data/validation',
                         description = 'validation emails')
    vector = CountVectorizer(encoding='utf-8',decode_error='ignore',analyzer="word",stop_words="english",
                              strip_accents="ascii",max_df=0.99,min_df=0.01)

    TrainM = vector.fit_transform(training_files.data)
    ValM = vector.transform(val_files.data)
    #Testing for variation in C and 'max_iter' to reduce error
    param_grid = {'C': np.logspace(-1,10, 100),'max_iter': [10,100,1000]}
    GS = GridSearchCV(LinearSVC(), param_grid)
    GS.fit(TrainM, training_files.target)
    GS.predict(ValM)
    best_C = GS.best_estimator_.C
    best_max_iter = GS.best_estimator_.max_iter
    print("**Grid Search Result: Linear Support Vector Machine***")
    print("Best score:",(GS.best_score_*100), "%")
    print("Best estimator of C:",best_C )
    print("Best estimator of max_iter:", best_max_iter )
    print("\n")

def GridSearch_LR():
    training_files=load_files('/Users/conorshirren/Desktop/EE514Assignment/prepro/sk_data/training',
                              description = ' training emails')
    val_files=load_files('/Users/conorshirren/Desktop/EE514Assignment/prepro/sk_data/validation',
                         description = 'validation emails')
    vector = CountVectorizer(encoding='utf-8',decode_error='ignore',analyzer="word",stop_words="english",
                              strip_accents="ascii",max_df=0.99,min_df=0.01)

    TrainM = vector.fit_transform(training_files.data)
    ValM = vector.transform(val_files.data)

    #Testing for variation in C and 'max_iter' to reduce error
    param_grid = {'C': np.logspace(-1,10, 100), 'max_iter': [10,100,1000] }
    GS = GridSearchCV(LogisticRegression(), param_grid)
    GS.fit(TrainM, training_files.target)
    pred = GS.predict(ValM)
    best_C = GS.best_estimator_.C
    best_max_iter = GS.best_estimator_.max_iter
    print("**Grid Search Result: Logistic Regression***")
    print("Best score:",(GS.best_score_*100), "%")
    print("Best estimator of C:",best_C )
    print("Best estimator of max_iter:", best_max_iter )
    print(GS.best_params_)
    print("\n")
```

Figure 24: GridSearchCV()

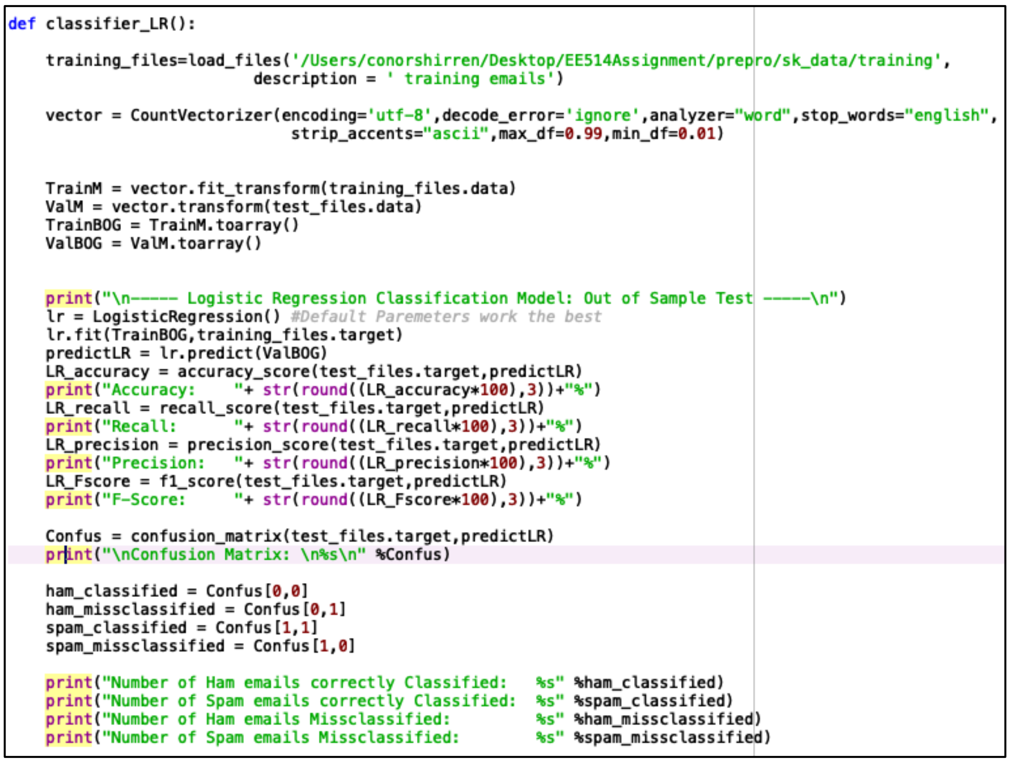


```
**Grid Search Result: Multinomisl Naive Bayes***
Best score: 96.87852352976084 %
Best estimator of alpha: 0.1


**Grid Search Result: Linear Support Vector Machine***
Best score: 97.5313037801911 %
Best estimator of C: 0.1
Best estimator of max_iter: 100


**Grid Search Result: Logistic Regression***
Best score:  98.3679525222552 %
Best estimator of C: 1.0
Best estimator of max_iter: 100
```

Figure 25: GridSearch for best Hyperparamters

6 Model Evaluation

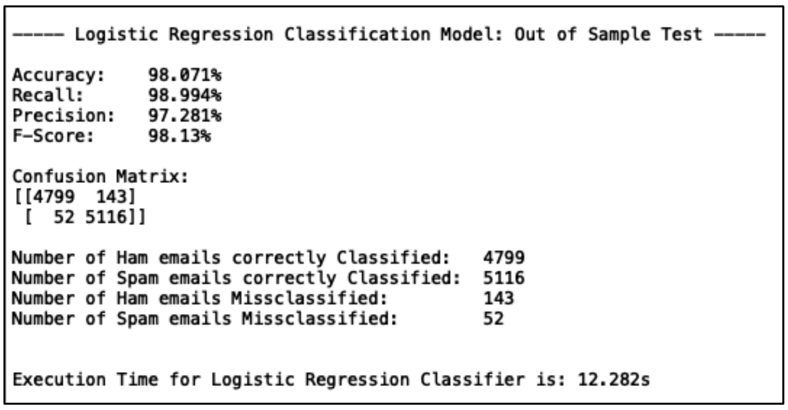
Section 5 of this report investigated the various models that can be used for the classification of the Enron email dataset as spam or ham. It can be seen in Figure 23 and Figure 25 that using the various features and hyperparameters tested in sections 4 & 5 that the logistic regression classification model yields the best performance. For this reason, the logistic regression model is chosen as our model to that will be used to obtain our out of sample error. The code for this section can be found in the 'ModelEvaluation.oy' python script. This Script can be seen in Figure 26 below.



```python
def classifier_LR():

    training_files=load_files('/Users/conorshirren/Desktop/EE514Assignment/prepro/sk_data/training',
                           description = ' training emails')

    vector = CountVectorizer(encoding='utf-8',decode_error='ignore',analyzer="word",stop_words="english",
                             strip_accents="ascii",max_df=0.99,min_df=0.01)


    TrainM = vector.fit_transform(training_files.data)
    ValM = vector.transform(test_files.data)
    TrainBOG = TrainM.toarray()
    ValBOG = ValM.toarray()


    print("\n----- Logistic Regression Classification Model: Out of Sample Test -----\n")
    lr = LogisticRegression() #Default Paremeters work the best
    lr.fit(TrainBOG,training_files.target)
    predictLR = lr.predict(ValBOG)
    LR_accuracy = accuracy_score(test_files.target,predictLR)
    print("Accuracy:    "+ str(round((LR_accuracy*100),3))+"%")
    LR_recall = recall_score(test_files.target,predictLR)
    print("Recall:      "+ str(round((LR_recall*100),3))+"%")
    LR_precision = precision_score(test_files.target,predictLR)
    print("Precision:   "+ str(round((LR_precision*100),3))+"%")
    LR_Fscore = f1_score(test_files.target,predictLR)
    print("F-Score:     "+ str(round((LR_Fscore*100),3))+"%")

    Confus = confusion_matrix(test_files.target,predictLR)
    print("\nConfusion Matrix: \n%s\n" %Confus)

    ham_classified = Confus[0,0]
    ham_missclassified = Confus[0,1]
    spam_classified = Confus[1,1]
    spam_missclassified = Confus[1,0]

    print("Number of Ham emails correctly Classified:   %s" %ham_classified)
    print("Number of Spam emails correctly Classified:  %s" %spam_classified)
    print("Number of Ham emails Missclassified:         %s" %ham_missclassified)
    print("Number of Spam emails Missclassified:        %s" %spam_missclassified)
```

Figure 26: Testing Model

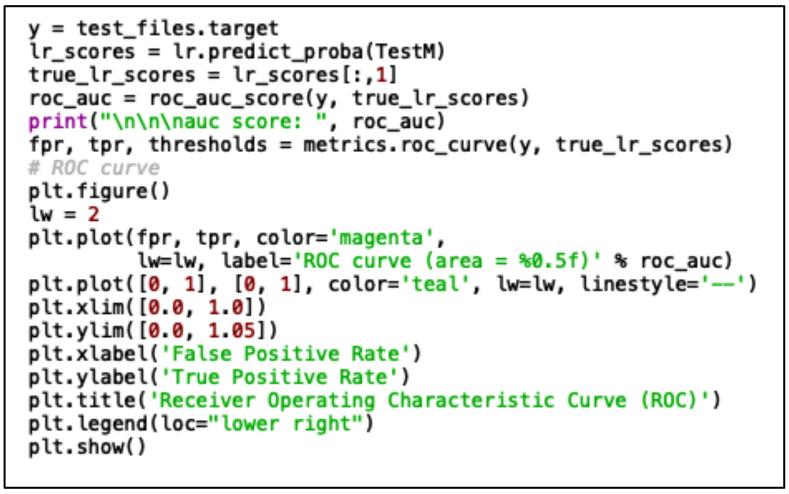
Predicting on the hold-out test set for the first time yields the following performance scores:



```
----- Logistic Regression Classification Model: Out of Sample Test -----

Accuracy:    98.071%
Recall:      98.994%
Precision:   97.281%
F-Score:     98.13%

Confusion Matrix:
[[4799  143]
 [  52 5116]]

Number of Ham emails correctly Classified:   4799
Number of Spam emails correctly Classified:  5116
Number of Ham emails Missclassified:         143
Number of Spam emails Missclassified:        52


Execution Time for Logistic Regression Classifier is: 12.282s
```

Figure 27: Out-of-Sample Performance

The following code can be used to obtain a ROC plot for the models prediction on the test set.



```
y = test_files.target
lr_scores = lr.predict_proba(TestM)
true_lr_scores = lr_scores[:,1]
roc_auc = roc_auc_score(y, true_lr_scores)
print("\n\n\nauc score: ", roc_auc)
fpr, tpr, thresholds = metrics.roc_curve(y, true_lr_scores)
# ROC curve
plt.figure()
lw = 2
plt.plot(fpr, tpr, color='magenta',
         lw=lw, label='ROC curve (area = %0.5f)' % roc_auc)
plt.plot([0, 1], [0, 1], color='teal', lw=lw, linestyle='--')
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.05])
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('Receiver Operating Characteristic Curve (ROC)')
plt.legend(loc="lower right")
plt.show()
```

Figure 28: ROC plot Code

The resultant ROC plot including an AUC score can be seen in Figure 29 below.

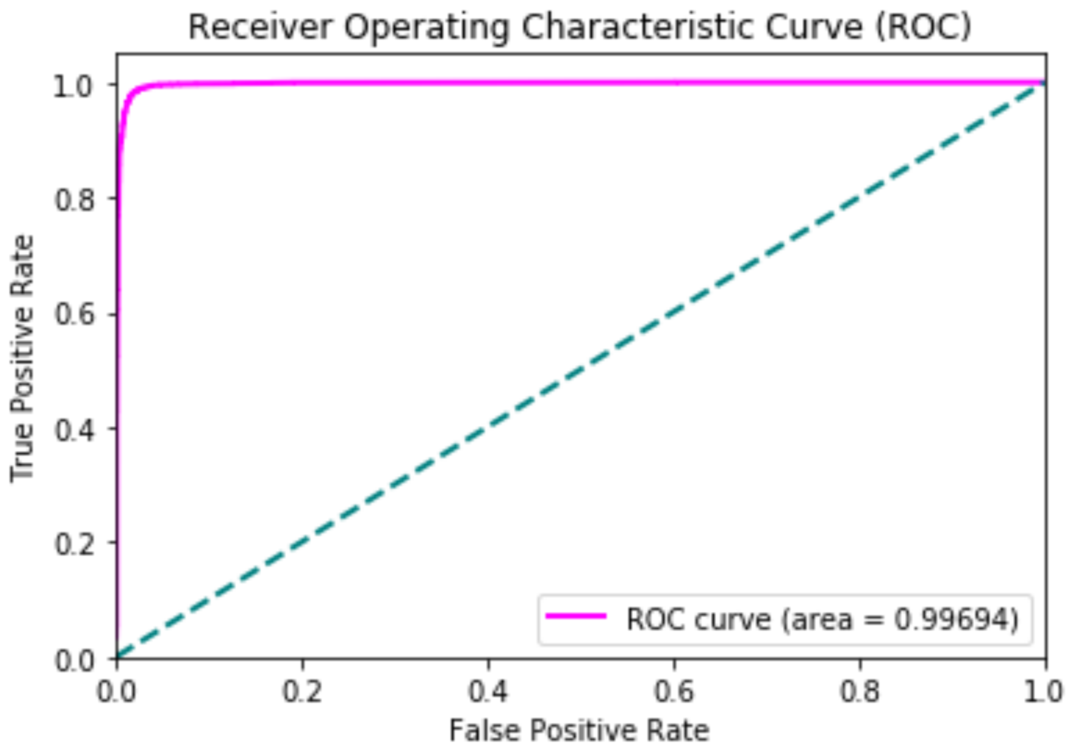


Figure 29: ROC Plot and AUC score

The auc score displays the area under the curve, displaying the percentage of the data that has been correctly classified. It can be seen from this plot that a fully automated and robust classifier has been design to classify spam and non-spam (i.e. ham) emails in the Enron Corporation Email Dataset [1]

8 Appendix

8.1 Bibliography

[1] "The Enron-Spam datasets." [Online]. Available: http://www2.aueb.gr/users/ion/data/enron-spam/. [Accessed: 14-Dec-2018].

[2] "Choosing the right estimator — scikit-learn 0.20.1 documentation." [Online]. Available: https://scikit-learn.org/stable/tutorial/machine_learning_map/index.html. [Accessed: 14-Dec-2018].

[3] "Spyder :: Anaconda Cloud." [Online]. Available: https://anaconda.org/anaconda/spyder. [Accessed: 14-Dec-2018].

[4] K. McGuinness, "Introduction to Supervised Learning," p. 91.

[5] "sklearn.datasets.load_files — scikit-learn 0.20.1 documentation." [Online]. Available: https://scikit-learn.org/stable/modules/generated/sklearn.datasets.load_files.html. [Accessed: 14-Dec-2018].

[6] "1. Supervised learning — scikit-learn 0.20.1 documentation." [Online]. Available: https://scikit-learn.org/stable/supervised_learning.html. [Accessed: 14-Dec-2018].

[7] "1.9. Naive Bayes — scikit-learn 0.20.1 documentation." [Online]. Available: https://scikit-learn.org/stable/modules/naive_bayes.html#multinomial-naive-bayes. [Accessed: 14-Dec-2018].

[8] "Document Classification with scikit-learn | zacstewart.com." [Online]. Available: http://zacstewart.com/2015/04/28/document-classification-with-scikit-learn.html. [Accessed: 14-Dec-2018].

[9] "Accuracy vs. Precision . . . Know the Difference? – DWK Life Sciences." .

[10] K. P. Shung, "Accuracy, Precision, Recall or F1?," *Towards Data Science*, 15-Mar-2018. [Online]. Available: https://towardsdatascience.com/accuracy-precision-recall-or-f1-331fb37c5cb9. [Accessed: 15-Dec-2018].

[11] "sklearn.grid_search.GridSearchCV — scikit-learn 0.16.1 documentation." [Online]. Available: https://scikit-learn.org/0.16/modules/generated/sklearn.grid_search.GridSearchCV.html. [Accessed: 15-Dec-2018].

8..2 Table of Figures

Figure 1: Choosing the right estimator[2] 3
Figure 2: Supervised Classifier Design Algorithm [4] 4
Figure 3: training, testing, and validation subsets [4] 5
Figure 4: Code Used to import all ham emails from enron dataset 6
Figure 5: Copy all emails to a single folder 6
Figure 6: Shuffling the ham and spam emails 7
Figure 7: Splitting the dataset into training, testing and validation 7
Figure 8: Format required to load files in sklearn[5] 7
Figure 9: Code used to partition training data into ham and spam 8
Figure 10: Current Directories 8
Figure 11: Bag of Words for ham emails 8
Figure 12: Bag of Words and Vocabulary 9
Figure 13: Word Frequency Plot of ham training set 9
Figure 14: Word Frequency Plot in spam training set 10
Figure 15: Email Length in Ham and Spam 10
Figure 16: Code for Box Plots and Bar Charts 11
Figure 17: Code used for training and testing the Naive Bayes Classifier 12
Figure 18: CountVectorizer with varying max_df and min_df 13
Figure 19: Count v Tfidf Vectorizer with and without max_df/min_df 14
Figure 20: Accuracy v Precision[9] 14
Figure 21: Precision and Recall[10] 15
Figure 22: F1-Score as a peformance estimator 15
Figure 23: Performance of default classifiers 15
Figure 24: GridSearchCV() 16
Figure 25: GridSearch for best Hyperparamters 16
Figure 26: Testing Model 17
Figure 27: Out-of-Sample Performance 17
Figure 28: ROC plot Code 18
Figure 29: ROC Plot and AUC score 18