TEAM 5: Riley, Taleen and Lior

Computer Assignment 02- Spam Email Detection

A. Setting up the environment:

In the first step, we import the necessary libraries that will help us with proceeding with developing the codes.

```
#importing the necessary libraries
import os
import numpy as np
from collections import Counter
from sklearn.naive_bayes import MultinomialNB
from sklearn.naive_bayes import GaussianNB
from sklearn.feature_extraction.text import CountVectorizer
from sklearn.metrics import accuracy_score
import email
import email.policy
```

B. Defined two functions:

make_Dictionary-

```
def make_Dictionary(root_dir):
  all_words = []
  emails = [os.path.join(root_dir,f) for f in os.listdir(root_dir)]
  for mail in emails:
    with open(mail) as m:
      for line in m:
        words = line.split()
        all_words += words
  dictionary = Counter(all_words)
  list_to_remove = list(dictionary)

  for item in list_to_remove:
    if item.isalpha() == False:
      del dictionary[item]
    elif len(item) == 1:
      del dictionary[item]
  dictionary = dictionary.most_common(3000)
  return dictionary
```

This function creates a dictionary of the 3000 most common words from a collection of email files- the ones in the train-mail folder. In this step, we extracted email addresses by defining the sets and concentrating on the most prevalent terms that might be suggestive of an email's content and, consequently, its classification, one can generate a manageable and significant feature set for training a machine learning model, for the spam classifier. The function, make_Dictionary, is constructed to process the set of emails and list a dictionary of the most used words found within those emails, excluding non-alphabetic words and single characters.

1. Initialize variables- create empty list to store all words that appears in the emails and it also generate a list of file paths for the email files in the given root directory
2. Process each email- using for loops it iterates through each email file, opens it and iterates through each line in the email content. Then it splits each line into individual words and adds them to the list of words created.
3. Count word occurrences- using a counter object to count the occurrences of each word.
4. Remove unwanted words- iterate through all the words in the list of words and remove non-alphabetic words, and single-character words from the dictionary of words (bag of words).
5. Finalize dictionary (bag of words)- select the 3000 most common words and return them as a final dictionary.

extract_email_features-

```
def extract_features(mail_dir):
  files = [os.path.join(mail_dir,fi) for fi in os.listdir(mail_dir)]
  features_matrix = np.zeros((len(files),3000))
  train_labels = np.zeros(len(files))
  count = 1;
  docID = 0;
  for fil in files:
    with open(fil) as fi:
      for i, line in enumerate(fi):
        if i ==2:
          words = line.split()
          for word in words:
            wordID = 0
            for i, d in enumerate(dictionary):
              if d[0] == word:
                wordID = i
                features_matrix[docID,wordID] = words.count(word)
      train_labels[docID] = 0;
      filepathTokens = fil.split('/')
      lastToken = filepathTokens[len(filepathTokens)-1]
      if lastToken.startswith("spmsg"):
        train_labels[docID] = 1;
        count = count + 1
      docID = docID + 1
   return features_matrix, train_labels
```

This function extracts features and labels from a collection of email files for a spam classification task. Feature_matrix function is extracting the arrays with dimensions including all rows and columns that have one out of the 3000 most common words in the dictionary. This code combines a basic naming principle to distinguish between spam and non-spam emails.

1. Initialize variables- similar to the other function it generate a list of file paths for the email files in the given root directory, initialize a matrix to store features for each email, initialize an array to store labels for each email (whether it is spam or non-spam), and initialize a counters.
2. Process each email- using a for loop it iterates through each email file, opens it and iterates through each line in the email content.
3. Extract features- it checks the third line of the email, splits the line into words and using a for loop iterates through each word and finds its index in the provided dictionary created earlier. Then it updates the features matrix with the count of each word.
4. Set labels- it extracts the last token from the file path and checks if it starts with “spmsg”. If it does then it identifies this email as spam and update its label to 1 (means spam, while 0 means non-spam)
5. Move to the next email and return results- incrementing the email id it moves to check the next email and returns the final feature matrix and label to the current email.

C. Clean and prepare the dataset (both train and test emails datasets):

```
train_dir = './train-mails'
test_dir = './test-mails'
dictionary = make_Dictionary(train_dir)
```

```
print ("reading and processing emails from TRAIN and TEST folders")
features_matrix, labels = extract_features(train_dir)
test_features_matrix, test_labels = extract_features(test_dir)
```

1. Connecting the path of the files to locate them in the code.
2. Call the function make_Dictionary on the train-mail folder of files- to create a dictionary of the 3000 most common words from this collection of email files. It evaluates all the emails found in this directory to show them in the dictionary of the most common words found in those emails.
3. Call the function extract_email_features on train- mail folder of files- to create the X_train (feature matrix) and y_train (label). Extracts features based on the created dictionary. Labels the emails to show if they're spam or not.
4. Call the function extract_email_features on train- mail folder of files- to create the X_test (feature matrix) and y_test (label). Extracts features based on the created dictionary. Labels the emails to show if they're spam or not.

D. Train- Naive Bayes algorithm module:

```
print("Training Model using Gaussian Naive Bayes algorithm .....")
model = GaussianNB()
model.fit(features_matrix, labels)
print("Training completed")
```

Train Naive Bayes classifier with the X_train and y_train- the email files information under train-email folder when X_train are the emails features and the y_train are the respected emails labels whether they are spam or non-spam. We found the multinomial Bayes and Gaussian Bayes returned similar results, but we decided to use the Gaussian Bayes algorithm.

E. Test:

```
print("Testing trained model to predict Test Data labels")
predicted_labels = model.predict(test_features_matrix)
print("Completed classification of the Test Data .... now printing Accuracy Score by comparing the Predicted Labels with the Test Labels:")
```

Test the trained model by using the model predicted from the test-email data, X_test which is the test features matrix. Using this info the model predicts whether these emails are to be considered spam or not. Using features retrieved from emails that have not been seen during training, the model makes predictions to label the emails. The prediction process anticipates the label with the highest likelihood by calculating the likelihood of each sample under the learned Gaussian distribution for each label.

F. Performance (accuracy):

```
accuracy = accuracy_score(test_labels, predicted_labels)
print(accuracy)
```

After the test there is a need to check how good the model is able to predict and detect spam emails. Therefore, we calculate accuracy score comparing the y_test (test_labels), actuall labels of the emails in test-email folder, with the predicted labels the model generated. This accuracy score helps understand the quality of the model. Additionally, we calculated the confusion matrix, another performance indicator, to show how many emails the model predicted falsely. We found that the accuracy of the model was affected by how many words were input in the initially defined dictionary. With 3000 words in the dictionary, the accuracy of the model was approximately 96%. When we changed the number of common words to 1,730 the model accuracy increased to just over 98%.