Case Study 3:
Using Naive Bayes to Classify and Filter Spam Emails

Kebur Fantahun, Eli Kravez, Halle Purdom

February 14, 2022

Introduction

Electronic mail or E-mail is one of the main ways that modern businesses communicate and collaborate. Several benefits such as instant sending and record of a message, the ability to speak to colleagues from home, security and convenience are all what make email preferable to other means of communication. That said, since emails are widely published on the internet so that any business can be easily reached, each email server has to deal with malicious, non-essential messages from people outside their respective organization. Malicious messages that contain viruses, unwanted and advertising emails that make it through are called spam while will deem the "real" messages, ham. Spam disrupts the businesses ability to work by flooding the email. A business would benefit from a spam filter, so that only necessary emails can get through. Simple spam filters might sort spam emails out by searching for keywords while a complex filter might use naive Bayes' or another trial and error based method so that the filter can learn based on prior spam messages how to attack new future spam. In this report naive Bayes classification models are created using a count vectorizer and term frequency-inverse document frequency (TF-IDF) to classify the spam and non-spam emails.

Methods

Data Preparation

The data given were in the format of three folders containing non-spam emails, and two additional folders containing spam emails. Two functions were created to parse through the folders containing the emails. The first accesses the folder and loops through each individual email file, which then calls the second function that will parse through the individual emails. If the emails are not of the type text/plain or test/html and they are multipart, they go into a recursive loop where the parse email function is called again to parse through the individual parts of that email. This parse email function also records whether an email is spam or not spam, outputting two arrays: data and target. All the "/n" characters were removed from the emails, and no further NLP processing was conducted as we are not familiar with the processes. After implementing these functions on all five folders, the only email types that could not be parsed through are of the image, signature, or multipart/alternative type. The final count of numbers of emails parsed through after processing is 9,703 emails.

The distribution of the emails versus spam after the data was processed was 7,085 non-spam emails and 2,616 spam emails.

Model Development

To build the spam classifier, naive Bayes classification is used in two models, where one uses a count vectorizer and the other uses TF-IDF. Clustering is also used to group the emails and add labels that could potentially help with the performance of the models.

The data was split with shuffling into a train and test set at an 80/20 split. The models were developed and parameters were optimized on the train data while the performance was calculated on the test data. K-fold cross validation with seven folds was also used on the train data to make sure the distribution was even amongst all the data. The precision values, the chosen metric for this study, were all very similar showing that the data is distributed evenly.

After the data is split, the count vectorizer and TF-IDF are used to transform the train and test data separately. Grid search was then used to optimize the alpha values based on the highest precision.

Precision was chosen as the appropriate metric to optimize the models. When filtering for spam emails, a business or organization would not want non spam-emails being filtered to the spam box. It would be better for some spam emails to break through the filter while ensuring no actual emails are lost because of the filter. By maximizing precision, the false positives are prevented so that no non-spam emails are incorrectly classified as spam.

The models could be further optimized by tuning the probabilities on the naive Bayes classifiers. The default threshold is 50%, but this threshold can be changed to get the best performance metrics for the models. The models were already performing extremely well in this study, so tuning this would not give much increase in performance. In future work or with a different dataset that caused the models to not perform as well, this may be another aspect that could be optimized for the models.

Kmeans++ was used to cluster the data into groups. These clusters are used to add features into the model to try and improve the performance of the models. To find the optimal number of clusters, the elbow method was used on the within-cluster sum of squares (WCSS) versus number of clusters graphs.

Results

Clustering Features

The optimal number of clusters found for the models was six clusters for both models. The TF-IDF model was a little less obvious of an "elbow", but was chosen to be the same number of clusters as the count vectorizer model. After adding these features into the datasets, they introduced almost no differences in the performance of the model. Because of this they were excluded from the final models below.

Final Count Vectorizer and TF-IDF Models

From the grid search algorithm, the best alpha value found that maximized precision was 1.41. This alpha had an extremely negative impact on recall. Because the overall goal of this is to create a well functioning spam filter, the alpha of 0.1 was used. The models still perform very well in terms of precision with this alpha, and the accuracy and recall metrics also perform very well with this alpha. The performance results of the models can be seen in the below table:

Naive Bayes Model	**Precision**	Recall	Accuracy
Count Vectorizer	**0.98641**	0.96030	0.98557
TF-IDF	**0.99020**	0.95463	0.98506

The TF-IDF model performed slightly better than the count vectorizer model. Both models performed extremely well in classifying the spam and non-spam emails.

Conclusion

In conclusion, the final TF-IDF model yielded a greater precision than the count vectorizer model, although it had a slightly lower accuracy. In our final model, clustering was not used as a feature since it did not introduce any significant performance changes to the models. This final model also had very high performance of precision, recall, and accuracy metrics across the board, and therefore would perform very well as a spam filter.

Further work on the models would include tuning of the naive Bayes probability threshold and testing other clustering algorithms other than kmeans++ to see if they would introduce any significant improvement to the performance of the models. Further NLP processing of the emails could also be added to the parse emails function.

Case Study 3

Build a spam classifier using naive Bayes and clustering. You will have to create your own dataset from the input messages. Be sure to document how you created your dataset.

In [ ]:

```
# Import libraries
import os
import numpy as np
import email


import email
from html.parser import HTMLParser
from bs4 import BeautifulSoup
from sklearn.feature_extraction.text import CountVectorizer
from sklearn.model_selection import train_test_split
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.naive_bayes import MultinomialNB
from sklearn.metrics import accuracy_score
from sklearn.metrics import precision_score, recall_score, confusion_matrix, accuracy_s
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.cluster import KMeans
from sklearn.model_selection import GridSearchCV, RandomizedSearchCV
from sklearn.metrics import confusion_matrix
from sklearn.metrics import precision_score, recall_score, confusion_matrix, accuracy_s
from sklearn.metrics import classification_report
from sklearn.metrics import accuracy_score
from sklearn.model_selection import cross_val_score
import pandas as pd
```

In [ ]:

```
# Function to access each data folder and file within folder
def parse_folder(list, file, data_var, target_var, label):
    for i in list:
            with open(file+i, "r",encoding='utf-8', errors='ignore') as f:
                x = email.message_from_file(f)
                mailType = x.get_content_type()
            parse_email(x, mailType, data_var, target_var, label)

# Parse through specific email
def parse_email(x, mailType, data_var, targets_var, label):
    global count
    if mailType == "text/plain":
        output = x.get_payload()
        output = output.replace("\n"," ")
        data_var.append(output)
        targets_var.append(label)
        count = count+1
    elif mailType == "text/html":
        tmp = BeautifulSoup(x.get_payload(), 'html.parser')
        tmp = tmp.text.replace("\n"," ")
        data_var.append(tmp)
        targets_var.append(label)
        count = count+1
    # If file type cannot be parsed, implement recusion of this function for a multipar
```

```
    else:
        if "multipart" in str(mailType):
            for j in x.get_payload():
                try:
                    parse_email(j, j.get_content_type(), data_var, targets_var, label)
                except:
                    y=1 # remove to print error statement
                    #print(x.get_content_type())
        #else:
            #print(f"msg not parsed: ",x.get_content_type())
```

In [ ]:

```
# Confusion matrix visualization
def conf_mtrx(y_test, preds):

    cf_matrix = confusion_matrix(y_test, preds)
    print(cf_matrix)

    group_names = ['True Neg','False Pos','False Neg','True Pos']
    group_counts = ["{0:0.0f}".format(value) for value in
                    cf_matrix.flatten()]
    group_percentages = ["{0:.2%}".format(value) for value in
                        cf_matrix.flatten()/np.sum(cf_matrix)]
    labels = [f"{v1}\n{v2}\n{v3}" for v1, v2, v3 in
            zip(group_names,group_counts,group_percentages)]
    labels = np.asarray(labels).reshape(2,2)

    x_axis_labels = ['ham', 'spam']
    y_axis_labels = ['ham', 'spam']

    sns.heatmap(cf_matrix, annot=labels, fmt='', cmap='Blues', xticklabels=x_axis_label
                yticklabels=y_axis_labels)
    plt.show()
```

In [ ]:

```
# Kmeans visualization of WCSS graph for elbow method
def kmeans_find_k(data):
    wcss = []
    for i in range(1, 11):
        kmeans = KMeans(n_clusters = i, init = 'k-means++', random_state = 42)
        kmeans.fit(data)
        wcss.append(kmeans.inertia_)

    plt.plot(range(1, 11), wcss)
    plt.xlabel('Number of clusters')
    plt.ylabel('WCSS')
    plt.show()
```

In [ ]:

```
# Count how many emails get parsed
count = 0

# Parse through each eamil in the 5 folders
easy_ham = "./SpamAssassinMessages/easy_ham/"
spam = "./SpamAssassinMessages/spam/"
easy_ham_2 = "./SpamAssassinMessages/easy_ham_2/"
spam_2 = "./SpamAssassinMessages/spam_2/"
hard_ham = "./SpamAssassinMessages/hard_ham/"
```

```
not_spamList = os.listdir(easy_ham)
spamList = os.listdir(spam)
not_spamList2 = os.listdir(easy_ham_2)
spamList2 = os.listdir(spam_2)
hard_ham2 = os.listdir(hard_ham)

data = [] # Initialize array for data
targets = [] # Initialize array for target

parse_folder(not_spamList, easy_ham, data, targets, 0)
parse_folder(spamList, spam, data, targets, 1)
parse_folder(not_spamList2, easy_ham_2, data, targets, 0)
parse_folder(spamList2, spam_2,data, targets, 1)
parse_folder(hard_ham2, hard_ham, data, targets, 0)
```

In [ ]:

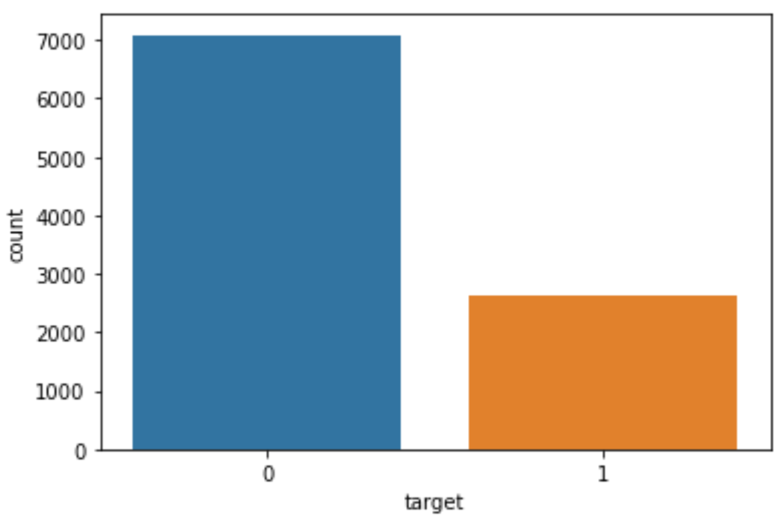
```
# Final count of parsed emails
len(data)
print(count)
```

```
9703
```

In [ ]:

```
# Visualize target distribution: 1 is spam and 0 is not spam
my_pd = pd.DataFrame(targets, columns =['target'])

ax = sns.countplot(x='target', data=my_pd)
plt.show()
```



In [ ]:

```
# Initialize count vectorizer and tfidf
cv = CountVectorizer()
tf = TfidfVectorizer()
```

In [ ]:

```
# Train test split 80/20 shuffle
X_train, X_test, y_train, y_test = train_test_split(data, targets, test_size=0.2, shuff
```

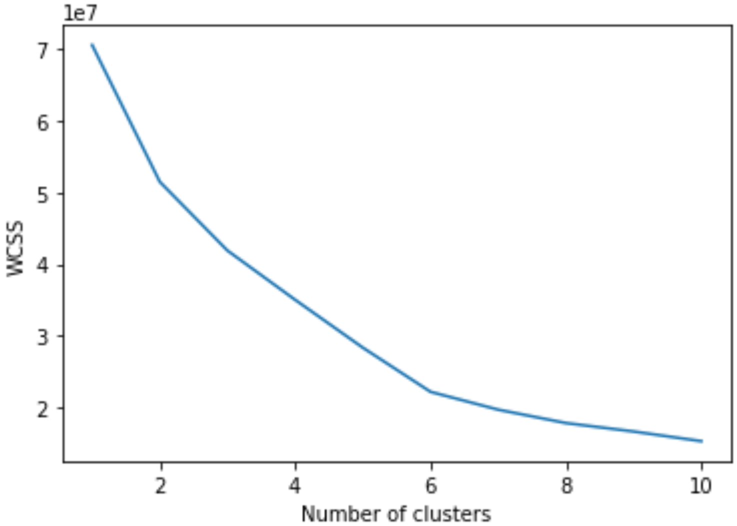
In [ ]:

```
# Transform the train and test data with cv and tfidf
cv_data = cv.fit_transform(X_train)
tf_data = tf.fit_transform(X_train)
```

```
cv_test_data = cv.transform(X_test)
tf_test_data = tf.transform(X_test)
```

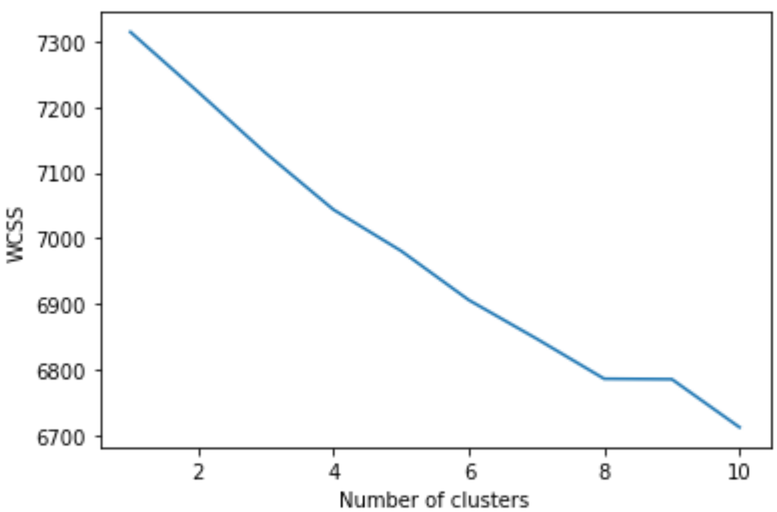
In [ ]:

```
# CV kmeans clustering WCSS graph for elbow method
kmeans_find_k(cv_data)
```



In [ ]:

```
# TFIDF kmeans clustering WCSS graph for elbow method
kmeans_find_k(tf_data)
```



In [ ]:

```
# Six clusters for both models- run kmeans++ clustering on train/test data

# cv train data
km = KMeans(6,random_state=42,init='k-means++', n_init=14, max_iter=100, tol=0.00001, c
km.fit(cv_data)
cv_clusters = km.labels_.tolist()

# cv test data
km.fit(cv_test_data)
cv_clusters_test = km.labels_.tolist()

# tf test data
km.fit(tf_data)
```

```
tf_clusters = km.labels_.tolist()

# cv test data
km.fit(tf_test_data)
tf_clusters_test = km.labels_.tolist()
```

In [ ]:

```
# Add kmeans++ clusters to the data as features when set to True
add_features = False      #To add these features to model, change to True (didn't intro

if add_features == True:
    from scipy.sparse import coo_matrix, hstack
    cv_data = hstack((cv_data,np.array(cv_clusters)[:,None])).A
    cv_test_data = hstack((cv_test_data,np.array(cv_clusters_test)[:,None])).A

    tf_data = hstack((tf_data,np.array(tf_clusters)[:,None])).A
    tf_test_data = hstack((tf_test_data,np.array(tf_clusters_test)[:,None])).A
```

CountVectorizer

In [ ]:

```
# Create NB model
model = MultinomialNB(alpha=0.1)
```

In [ ]:

```
# Fit to train data
model.fit(cv_data, y_train)
```

Out[ ]:

```
MultinomialNB(alpha=0.1)
```

In [ ]:

```
# TAKES LONG TIME TO RUN: uncomment to run grid search results

# grid_params = {
#   'alpha': np.linspace(0, 1.5, 100)
# }
# clf = GridSearchCV(model, grid_params,  scoring='precision')
# clf.fit(tf_data, y_train)
# print("Best Score: ", clf.best_score_)
# print("Best Params: ", clf.best_params_)
```

In [ ]:

```
# Get predictions from model
test_preds = model.predict(cv_test_data)
```

In [ ]:

```
# Get accuracy of model
accuracy_score(test_preds,y_test)
```

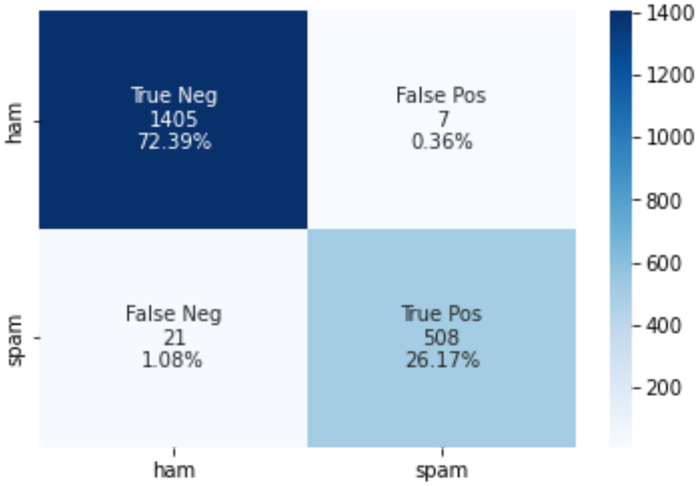
Out[ ]:

```
0.9855744461617723
```

In [ ]:

```
# Confusion matrix of model
conf_mtrx(y_test, test_preds)
```

```
[[1405    7]
 [  21  508]]
```



In [ ]:

```
# Performance metrics of model
precision = precision_score(y_true=y_test, y_pred=test_preds)
recall = recall_score(y_true=y_test, y_pred=test_preds)
accuracy = accuracy_score(y_true=y_test, y_pred=test_preds)

print(f"Accuracy: {accuracy:.5f}")
print(f"Precision: {precision:.5f}")
print(f"Recall: {recall:.5f}")
```

```
Accuracy: 0.98557
Precision: 0.98641
Recall: 0.96030
```

In [ ]:

```
# Classification report
print(classification_report(y_test,test_preds,target_names=['ham', 'spam']))
```

```
              precision    recall  f1-score   support

         ham       0.99      1.00      0.99      1412
        spam       0.99      0.96      0.97       529

    accuracy                           0.99      1941
   macro avg       0.99      0.98      0.98      1941
weighted avg       0.99      0.99      0.99      1941
```

In [ ]:

```
# Looking at distribution of train data to ensure no uneven distribution
cross_val_score(model, tf_data, y_train, cv=7, scoring='precision')
```

Out[ ]:

```
array([0.99295775, 0.99283154, 0.98639456, 0.97315436, 0.98976109,
       0.98269896, 0.9929078 ])
```

TFIDF

In [ ]:

```
# Fit model to tf data
model.fit(tf_data, y_train)
```

Out[ ]:

```
MultinomialNB(alpha=0.1)
```

In [ ]:

```
# Predictions
preds = model.predict(tf_test_data)
```

In [ ]:

```
# Accuracy of model
accuracy_score(preds, y_test)
```

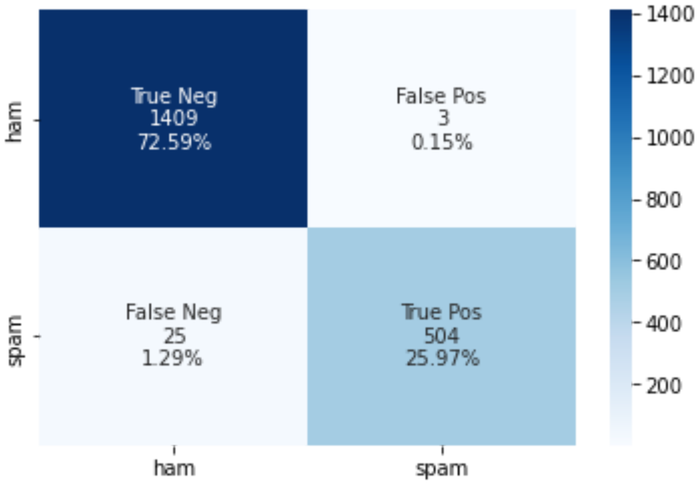
Out[ ]:

```
0.9855744461617723
```

In [ ]:

```
# Confusion Matrix
conf_mtrx(y_test, preds)
```

```
[[1409    3]
 [  25  504]]
```



In [ ]:

```
# Performance metrics
precision = precision_score(y_true=y_test, y_pred=preds)
recall = recall_score(y_true=y_test, y_pred=preds)
accuracy = accuracy_score(y_true=y_test, y_pred=preds)

print(f"Accuracy: {accuracy:.5f}")
print(f"Precision: {precision:.5f}")
print(f"Recall: {recall:.5f}")
```

```
Accuracy: 0.98557
Precision: 0.99408
Recall: 0.95274
```

In [ ]:

```
# Classification Report
print(classification_report(y_test,preds,target_names=['ham', 'spam']))
```

```
              precision    recall  f1-score   support

         ham       0.98      1.00      0.99      1412
        spam       0.99      0.95      0.97       529

    accuracy                           0.99      1941
   macro avg       0.99      0.98      0.98      1941
weighted avg       0.99      0.99      0.99      1941
```