Classifying Spam Emails Using Clustering and Naïve Bayes

1. Introduction

Studies have shown that nearly half of the emails sent worldwide are spam. This can include a wide range of cybersecurity threats and unsolicited messages sent in bulk. The senders usually try to promote and sell unregulated goods or try to trick the recipient into handing over sensitive information, like a password or credit card numbers. The level of sophistication applied by spammers has increased with the use of malware and ransomware. Along with their sophistication, spam emails have increased the risks and costs faced by corporations into the billions of dollars. Due to the level of risks, it is critical to be able to filter unsolicited emails to prevent sensitive information from being gathered by these spammers.

2. Dataset

We were given a dataset of 9348 unique emails that were already classified as either spam or non-spam. A portion of those emails contained multiple parts that needed to be considered. Once we were able to parse the multi-part emails we had a data set that included 10805 records.

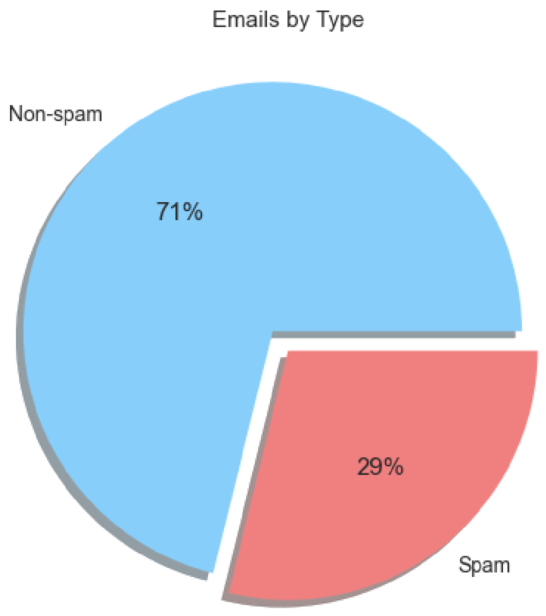


Figure 1. This distribution of emails will translate to predictions with 2 similar partitions despite train test splits.

2.1 Text Normalization

To help transform the data into a set of words that are their common base form we normalized the data. This helps reduce the information the engine must deal with when we're building our model. We began by utilizing the tokenization process which split the text objects into smaller units known as tokens. This left us with the basic meaningful units of each email. Our tokenization removed lowercase and special characters, filtered out stop words, and removed numbers and short tokens.

Once normalized we needed to convert our data to a mathematical representation to be able run our algorithms. Using tf-idf, or the Term Frequency times Inverse Document Frequency, we converted our email texts into a matrix of TF-IDF features, then added the number of clusters that each email belongs to as a categorical column into our data frame.

3. Clustering

To help with engineering additional classification indicators before we began building our classifier model, we incorporated a clustering analysis. An initial guess of K = 13 was used to cluster the features of our tf-idf matrix, and the resulting sum of squared distance can be illustrated below.

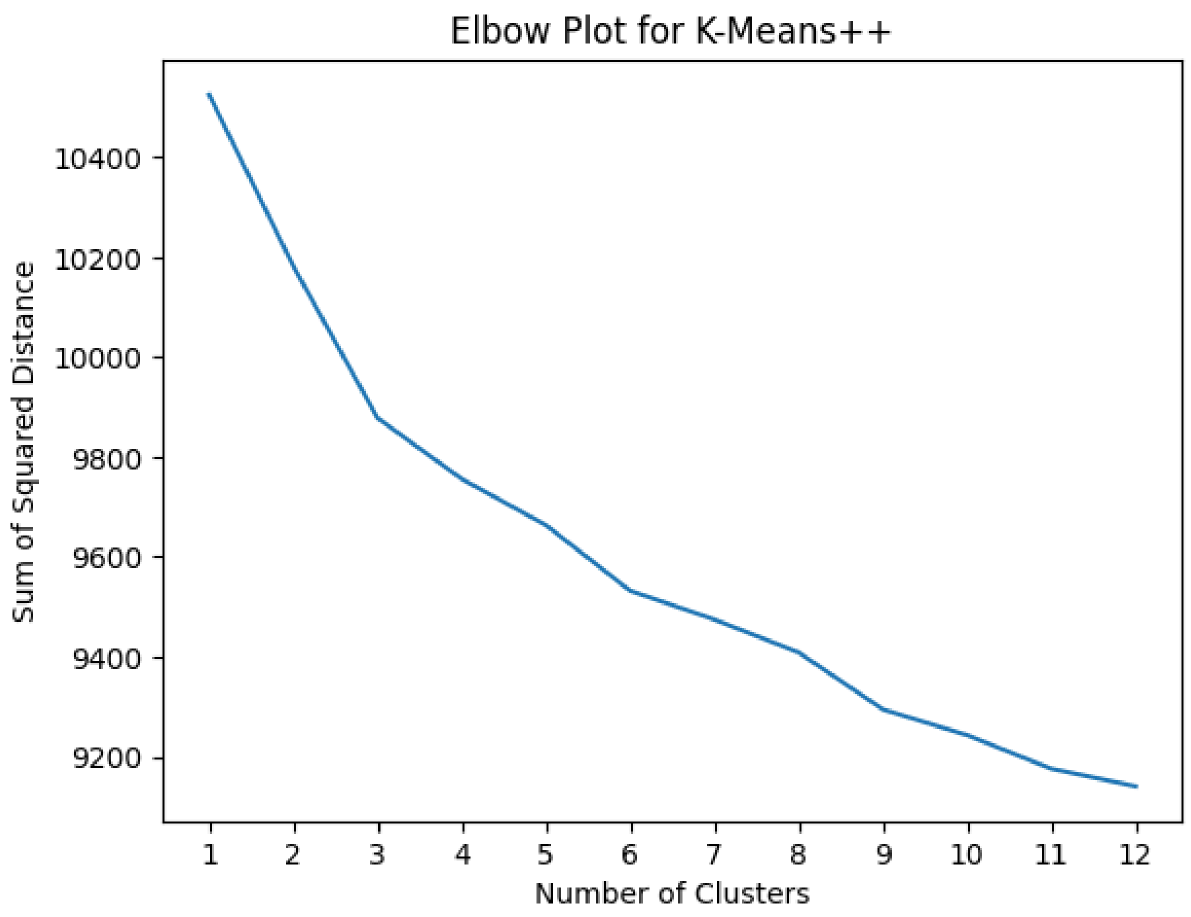


Figure 2. There is a major change in SSE before and after 3 clusters.

We have strong evidence that the best number of clusters is 3, however we will visualize below using the *SilhouetteVisualizer* from the *yellowbrick* package. The red dotted line represents the average silhouette score of the silhouette plot seen below. One cluster that had parts of its scores as negative can be seen in the plot, however the other 2 clusters had scores above the average silhouette score.

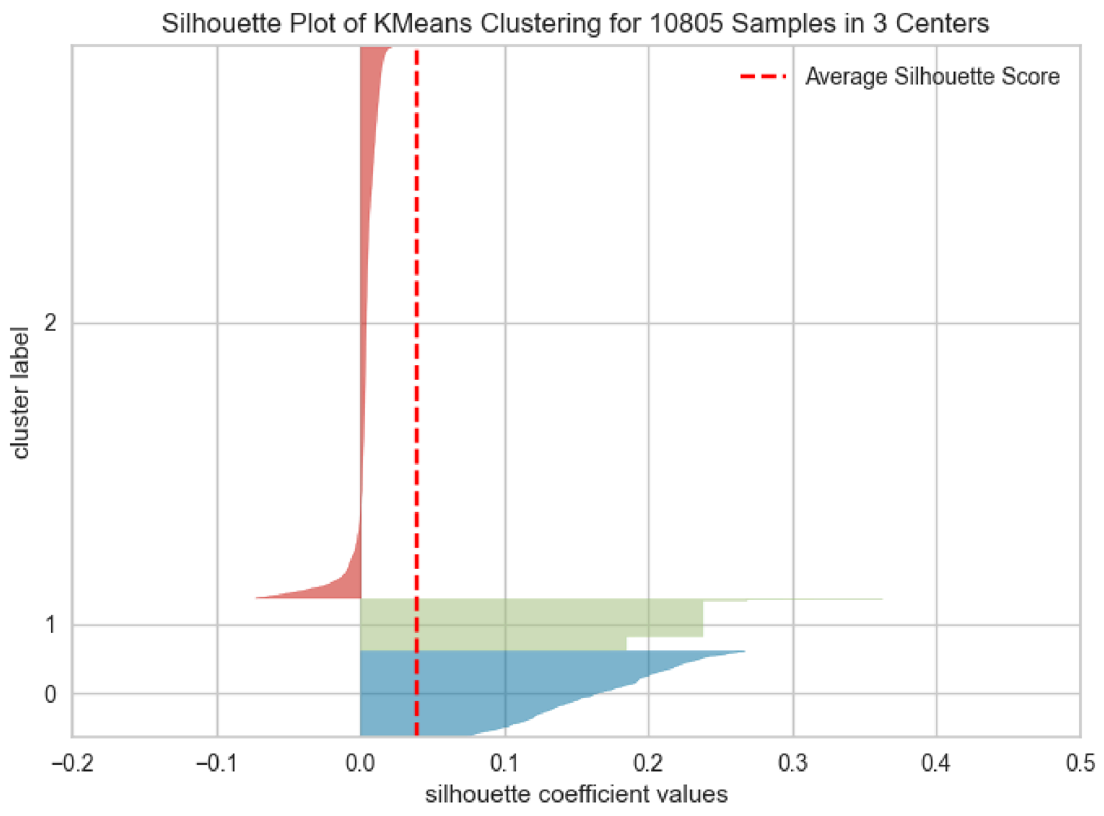


Figure 3. The finalized 3 clusters chosen for Naïve Bayes. The negative score seen here can be interpreted as them belonging to another cluster because some of its features are much farther from the cluster center.

4. Naïve Bayes Classifier

4.1. Model Building

All 5 possible supervised methods of Naive Bayes from sklearn, BernoulliNB, CategoricalNB, ComplementNB, GaussianNB, and MultinomialNB, were tested to determine the best method. The GaussianNB method resulted in the highest accuracy score from cross_val_score().

The next step is to get the classification reports. A column for the assigned cluster for each record was appended to our tf_matrix. The same 5 methods were performed again to determine their precision, recall and accuracy. The GuassianNB model still performed the best out of all of the methods. Table 1 in results shows the comparison of the 5 algorithms on our tf matrix with the clusters. Performing the 5 algorithms again without clusters for classification_report seemed unnecessary, so it was omitted from the table.

4.2. Evaluation Metrics

Precision evaluates how many of the model's positive results are **true positives**, or how many of the emails classified as spam are actually spam. This is the percentage of emails predicted as spam that were correctly classified.

- **Precision** = TP / (TP + FP)
- **False positive** represents the emails classified as spam by our model but were actually not spam while a false negative represents emails classified as not spam by our model but were actually spam. Our goal is to have as few non-spam emails classified as spam (false positive) as possible, to a point where predicting large amounts of actual spam emails are perfectly fine (high false negatives, or low recall).

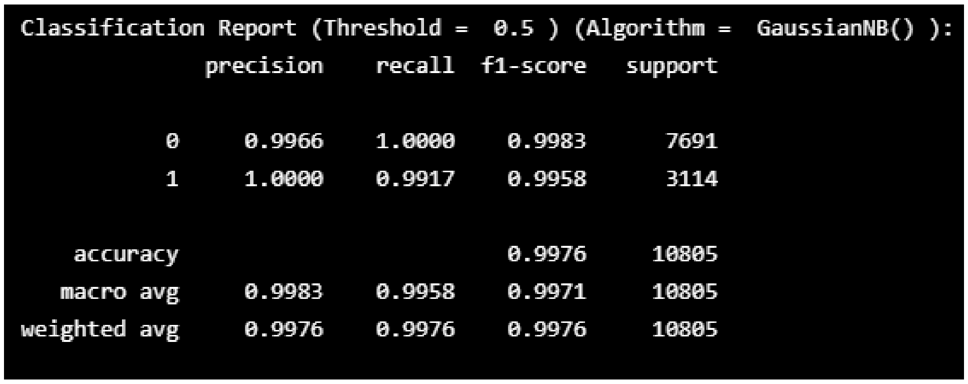
Recall calculates how many spam emails were actually classified as spam.

- **Recall** = TP / (TP + FN) - This is the percentage of actual spam emails that were correctly classified.
- **False negative**s represent the spam emails that were classified as non spam by our model.

Precision and recall are two extremely important model evaluation metrics. While precision refers to the percentage of your positive results which are true positives, recall refers to the percentage of total relevant results correctly classified by your algorithm. Unfortunately, it is not possible to maximize both these metrics at the same time, as one comes at the cost of another. Increasing precision means decreasing FP, and increasing recall means decreasing FN. Decreasing FP is much more valuable than decreasing FN, because we don't have to care for high amounts of non-spam emails that we misclassified. We don't want to misclassify any spam emails at all if possible, so in this case we want a higher precision and can sacrifice recall.

4.3. Results

Below are plots that represent of our precision vs recall vs threshold:



```
Classification Report (Threshold =  0.5 ) (Algorithm =  GaussianNB() ):
              precision    recall  f1-score   support

           0     0.9966    1.0000    0.9983      7691
           1     1.0000    0.9917    0.9958      3114

    accuracy                         0.9976     10805
   macro avg     0.9983    0.9958    0.9971     10805
weighted avg     0.9976    0.9976    0.9976     10805
```

Figure 4. This table showcases the high performing GaussianNB on our tf_matrix with clusters. "0" denotes non-spam emails, and "1" denotes spam emails.

Naïve Bayes Algorithms	*Precision*	*Recall*	*Accuracy*
Guassian	1.0000	.09907	0.9973
Guassian with clusters	**1.0000**	**0.9917**	**0.9976**
Bernoullli with clusters	0.9799	0.6426	0.8932
Categorical with clusters	0.6761	0.4679	0.7820
Complement with clusters	0.9441	0.5584	0.8632
Multimonial with clusters	0.9570	0.4865	0.8457

Table 5. Highlighted in bold is the best performing Naïve Bayes algorithm.

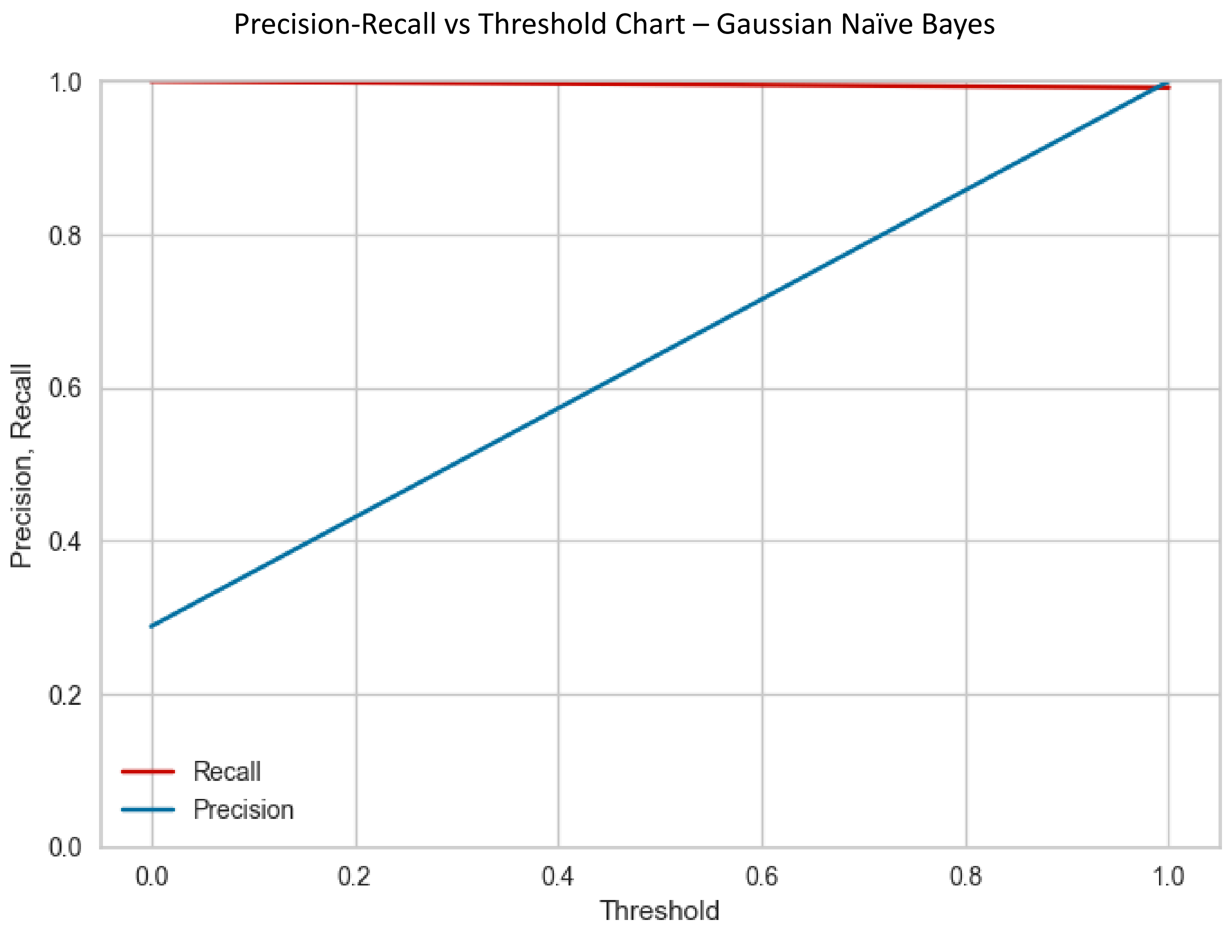


Figure 6. Peak precision is observed to be nearer 1.0 than 0.95. A trade-off is observed at around threshold 0.55 between precision and recall. Precision equals zero when threshold is both 0.0 and 1.0.

4. Conclusion

To conclude, we built a TF-IDF data frame containing all SpamAssasin emails and appended the cluster designation for each email record. Evaluating our competing models using GaussianNB was the best method to classify our emails, where both scores were extremely high being very close to 100% precision, recall and accuracy.

5. Appendix

```
import warnings

import sys

from os import listdir

from os.path import isfile, join

import pandas as pd

import email

import nltk

import re

import numpy as np

from sklearn.feature_extraction.text import TfidfVectorizer

from sklearn.cluster import KMeans

from sklearn.naive_bayes import GaussianNB

from sklearn.metrics import classification_report



warnings.filterwarnings("ignore")




def do_my_study():

    df, msgs = import_data()



    # print(df[df["filename"].str.contains('00947')])
    # print(len(msgs))

    tf_matrix = create_tfidf_matrix(df)

    # print(tf_matrix.shape[0])

    clusters = 3
    df = assign_clusters(df, clusters, tf_matrix, 'kmeans_cluster')

    # print(df['kmeans_cluster'].value_counts())

    tf_matrix_array_cluster = add_column_to_tf_matrix(
        tf_matrix, df, 'kmeans_cluster')

    # print(tf_matrix_array_cluster.shape)

    perform_NB_algorithm_and_evaluate(
        GaussianNB(),
        X=tf_matrix_array_cluster, y=df['is_spam'],
        threshold=0.5)
```

```
def import_data():
    user_input_path = sys.argv[1]


    directories = [
                'easy_ham',
                'easy_ham_2',
                'hard_ham',
                'spam',
                'spam_2'
                ]


    msgs = []
    file_name = []
    label = []
    for d in directories:
        mypath = user_input_path + '/SpamAssassinMessages/' + d + '/'
        # mypath = getcwd() + '/SpamAssassinMessages/' + d + '/'
        myfiles = [f for f in listdir(mypath) if isfile(join(mypath, f))]


        for file in myfiles:
            with open(mypath + file, encoding='latin1') as f:
                msg = email.message_from_file(f)
                for part in msg.walk():
                    if "spam" in d:
                        label.append(1)
                    else:
                        label.append(0)
                    file_name.append(file)
                    payload = part.get_payload()
                    msgs.append(payload)

    df = pd.DataFrame({"filename": file_name, "is_spam": label})
    df['messages'] = msgs

    return df, msgs


def normalize_text_func(text):
    stop_words = nltk.corpus.stopwords.words('english')

    # make all special characters lowercase and remove them
    text = re.sub(r'[^a-zA-Z0-9\s]', ' ', text, re.I | re.A)
    text = text.lower()
    text = text.strip()

    # tokenize document
    tokens = nltk.word_tokenize(text)

    # filter out stop words
    filtered_tokens = [token for token in tokens
```

```python
                       if token not in stop_words]

    # Remove numbers
    filtered_tokens = [token for token in filtered_tokens
                       if not token.isdigit()]

    # Remove short tokens
    filtered_tokens = [token for token in filtered_tokens
                       if len(token) > 2]

    # re-create a normalized document
    text = ' '.join(filtered_tokens)
    return text


def create_tfidf_matrix(df):
    normalize_text = np.vectorize(normalize_text_func)
    normalized_text = normalize_text(df['messages'].astype('str'))

    vectorizer = TfidfVectorizer()
    tf_matrix = vectorizer.fit_transform(str(i) for i in normalized_text)

    return tf_matrix
```

```
def assign_clusters(df, clusters, tf_matrix, column_name):
    km = KMeans(n_clusters=clusters,
                max_iter=10000,
                n_init=10,
                random_state=42).fit(tf_matrix)
    df[column_name] = km.labels_

    return df


def add_column_to_tf_matrix(tf_matrix, df, column_name):
    tf_matrix_array = tf_matrix.toarray()
    tf_matrix_array_column = np.append(
        tf_matrix_array, df[column_name].values.reshape(-1, 1), axis=1)

    return tf_matrix_array_column


def perform_NB_algorithm_and_evaluate(algorithm, X, y, threshold):
    algorithm.fit(X, y)

    y_preds = (algorithm.predict_proba(X)[:, 1] > threshold).astype('float')
    print("Classification Report ( Threshold = ", threshold,
          ") ( Algorithm = ", algorithm, "):")
    print(classification_report(y, y_preds, digits=4))



if __name__ == "__main__":


    do_my_study()
```