INOFRMATICS INSTITUTE OF TECHNOLOGY

In collaboration with

ROBERT GORDEN UNIVERSITY ABERDEEN

BSc. Artificial Intelligence & Data Science

Level 05

CM 2604

Machine Learning

Coursework Report

NADUN SHAMIKA SENARATHNE

IIT ID: 20210488

RGU ID: 2117538

Table of Contents

ROBERT GORDON UNIVERSITY ABERDEEN TEF Gold

INFORMATICS INSTITUTE OF TECHNOLOGY

Corpus Preparation

The UCI website, which contains a huge selection of datasets that may be utilized for any machine learning task, is where the Spambase Data Set was collected from. The last column of the dataset, which has around 4600 data and 58 attributes/features, indicates if the provided email is spam or not. A spam email is indicated by the number 1, and a non-spam email by the number 0. By examining the qualities that show if a certain character or given word is commonly occurring in a letter, the emails were divided into spam and no spam groups.

In terms of the data preprocessing, actions like data cleansing were taken. The processes included examining and eliminating null values, eliminating duplicate values, and eliminating outliers.

K-nearest neighbors and Decision Tree, two machine learning techniques, were used to classify emails as spam or not spam. Following careful management of the data cleaning phase, the dataset was split into train and test halves, with 70% of the data being designated as train data and the remaining portion as test data. It is considerably better to allocate between 20 and 30% because the data can then have a larger percentage of training. Yet it varies according on the circumstances. There is no such thing as an ideal split percentage for data training.

Solution Methodology

KNN and Decision Tree algorithms were used to classify spam and non-spam emails. An overview of the two algorithms that were employed may be found below.

KNN, or the k-nearest neighbor algorithm, is a non-parametric model that locates the k data points that are closest to a test point given a scenario, and then classifies the test point based on the majority class of the k-neighbors.

Decision Tree,The decision tree method divides the dataset into groups that resemble tree topologies. According to the values of the inputs, the trees will have a stopping criterion.

Using the Spambase dataset, which initially divides the data into train and test sets, the KNN and Decision Tree models were employed. Then, features were employed as inputs, and labels were used to produce results. The PCA technique was used to complete the classification challenge since it enhances the models' effectiveness and performance. To select the ideal parameters and enhance the models' output, methods like grid search were used. Moreover, feature engineering was carried out since it finds the dataset's most instructive characteristics, develops new features, and captures the intricate interactions between labels.

Evaluation Criteria

The metrics employed for the spam, no spam email categorization scenario, including as accuracy, precision, recall, and F1-scores, are described below along with the justifications for their use.

- **Accuracy**: In the case described, it is essential to have a high accuracy rate because it is crucial to properly differentiate and identify spam and non-spam emails in order to prevent missing crucial communications.
- **Precision & Recall**: For the classification instance, precision and recall metrics were utilized because they both represent the model's capacity to properly identify the number of true positives and false positives separately.
- **F1-Score**: This score was utilized to assess the model's performance since it balances precision and recall and provides a thorough picture of the models' overall effectiveness.

Accuracy

$$\text{Accuracy} = \frac{(TP + TN)}{(TP + FP + TN + FN)}$$

Precision

$$\text{Precision} = \frac{TP}{(TP + FP)}$$

Recall

$$Recall = \frac{True\ Positive(TP)}{True\ Positive(TP) + False\ Negative(FN)}$$

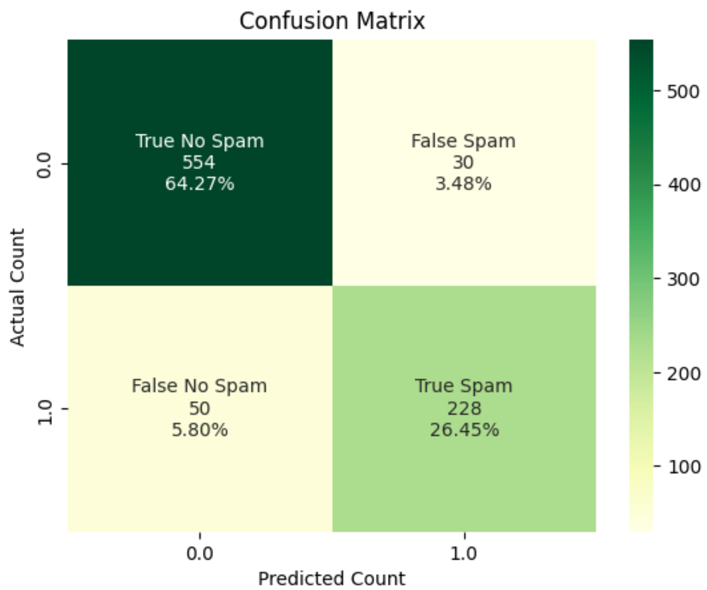
F1-Score

$$F_1 = 2 * \frac{precision * recall}{precision + recall}$$

Model Evaluation

	KNN with PCA	Decision Tree with PCA	KNN without PCA	Decision Tree without PCA
Train and Test split proportion	Train – 75%, Test – 25%	Train – 75%, Test – 25%	Train – 70%, Test – 30%	Train – 70%, Test – 30%
Test Accuracy	90.71%	86.31 %	90.13 %	91.58 %
Precision (weighted)	91%	86%	90%	92%
Recall (weighted)	91%	86%	90%	92%
F1-Score (weighted)	91%	86%	90%	92%

Confusion matrix for KNN (with PCA):

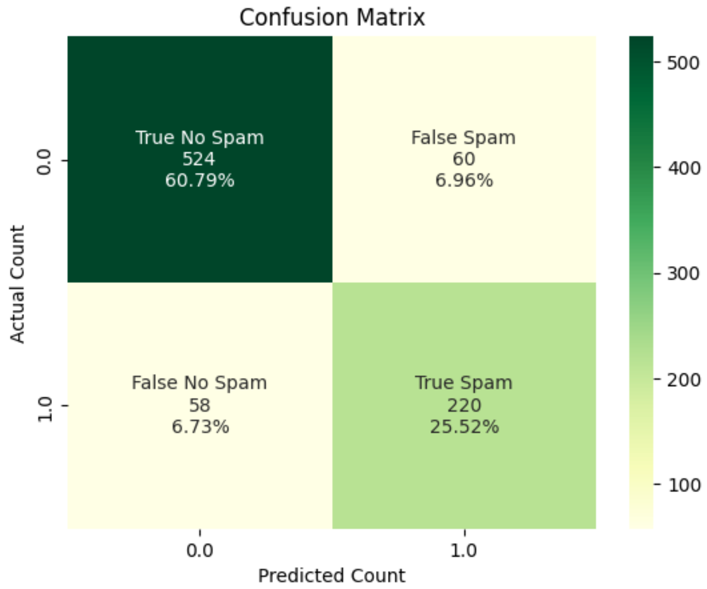


Classification Report for KNN (PCA):

Classification Report :

	precision	recall	f1-score	support
0.0	0.92	0.95	0.93	584
1.0	0.88	0.82	0.85	278
accuracy			0.91	862
macro avg	0.90	0.88	0.89	862
weighted avg	0.91	0.91	0.91	862

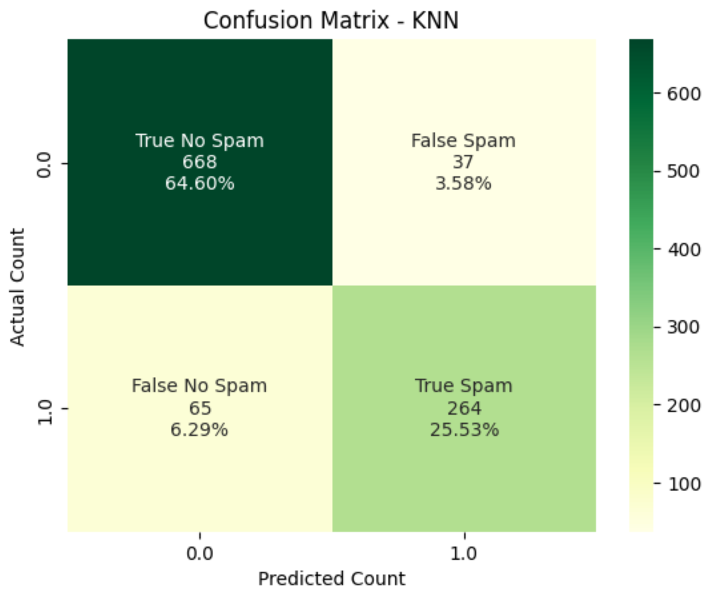
Confusion matrix for Decision Tree (with PCA):



Classification Report for Decision Tree (PCA):

```
Classification Report :
              precision    recall  f1-score   support

         0.0       0.90      0.90      0.90       584
         1.0       0.79      0.79      0.79       278

    accuracy                           0.86       862
   macro avg       0.84      0.84      0.84       862
weighted avg       0.86      0.86      0.86       862
```

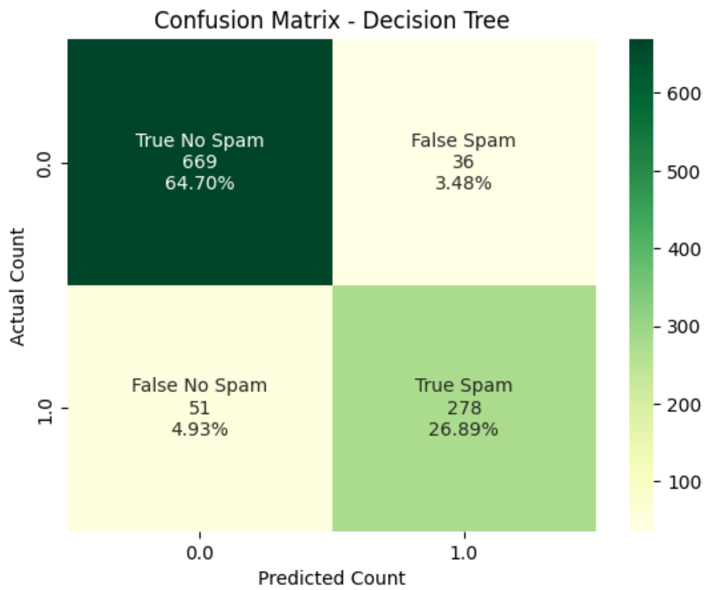
Confusion matrix for KNN (without PCA):



Classification Report for KNN (without PCA):

```
Classification Report :
              precision    recall  f1-score   support

         0.0       0.91      0.95      0.93       705
         1.0       0.88      0.80      0.84       329

    accuracy                           0.90      1034
   macro avg       0.89      0.87      0.88      1034
weighted avg       0.90      0.90      0.90      1034
```

Confusion matrix for Decision Tree (without PCA):



Classification Report for Decision Tree (without PCA):

```
Classification Report :
              precision    recall  f1-score   support

         0.0       0.93      0.95      0.94       705
         1.0       0.89      0.84      0.86       329

    accuracy                           0.92      1034
   macro avg       0.91      0.90      0.90      1034
weighted avg       0.92      0.92      0.92      1034
```

Based on the metrices that were utilized, both models gave good scores of accuracies. When PCA was applied, KNN, however, outperformed Decision Tree. By having a walkthrough classification report it can be argued that a high precision can be obtained if the amount of false positives are taken into consideration and high recall can be obtained if the false negatives were taken into consideration.

Experimental Results

Using the K-Nearest and Decision tree algorithms, an experiment was done to evaluate and categorize the results of spam and no spam emails. More than 4800 emails made up the Spambase dataset, which was split into 25% of the data for testing and 75% of the data for training.

In K-Nearest methods, the number of k components was set automatically to determine the Euclidian distance between each data point from the hyperparameter tuning by passing its value. To reduce the problems associated with overfitting, the decision tree's tree depth was also passed the parameters associated with it.

To improve the performance of the two models, various metrics were applied to the training data. According to the findings, KNN had an accuracy rate of 90.71% after PCA was applied, whereas decision tree had an accuracy rate of 86.31 %.

Overall, it was found that KNN was more effective than decision tree when PCA was used, but decision tree was more effective when PCA was not used.

Limitations & Further Enhancements

Limitations:

- As the dataset used to train the model does not contain any real-world data, data bias may have an impact on the model's performance.
- A decision tree has a potential of being overfitted, especially if it is too complicated, the data is noisy, or the training process was improper.

Ways to overcome the limitations:

- Utilizing a more representative dataset to provide results that are more exact and accurate.
- Techniques like pruning can be used for decision tree classification to address the overfitting issue.

Future enhancements:

- Testing out various methods, such as Naive Bayes and Support Vector Machines (SVM), and comparing their performance to that of Decision Tree and K-Nearest Neighbors algorithms.
- Using the same models to various tasks, like sentiment analysis and topic classification, and evaluating the outcomes.

GitHub project URL:

The whole code that was used to categorize spam and non-spam email using KNN and Decision Tree classifiers is available at the GitHub link below.

github.com/Nadun999

Appendix:

Import dependencies

```
import pandas as pd
import numpy as np
import re
from sklearn import preprocessing
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split, cross_val_score
from sklearn.feature_extraction.text import CountVectorizer
from sklearn.tree import DecisionTreeClassifier
from sklearn.neighbors import KNeighborsClassifier
import seaborn as sn
import matplotlib.pyplot as plt
from sklearn.metrics import accuracy_score, confusion_matrix, classification_report
from sklearn.model_selection import GridSearchCV
from sklearn.decomposition import PCA
```

Reading the dataset

```
# Reading in the spambase data from a CSV file and storing it in a pandas dataframe
# The header parameter is set to None since the data does not contain column headers
data = pd.read_csv('spambase.data', header=None)


data
```

	0	1	2	3	4	5	6	7	8	9	...	48	49	50	51	52	53	54	55	56	57
0	0.00	0.64	0.64	0.0	0.32	0.00	0.00	0.00	0.00	0.00	...	0.000	0.000	0.0	0.778	0.000	0.000	3.756	61	278	1
1	0.21	0.28	0.50	0.0	0.14	0.28	0.21	0.07	0.00	0.94	...	0.000	0.132	0.0	0.372	0.180	0.048	5.114	101	1028	1
2	0.06	0.00	0.71	0.0	1.23	0.19	0.19	0.12	0.64	0.25	...	0.010	0.143	0.0	0.276	0.184	0.010	9.821	485	2259	1
3	0.00	0.00	0.00	0.0	0.63	0.00	0.31	0.63	0.31	0.63	...	0.000	0.137	0.0	0.137	0.000	0.000	3.537	40	191	1
4	0.00	0.00	0.00	0.0	0.63	0.00	0.31	0.63	0.31	0.63	...	0.000	0.135	0.0	0.135	0.000	0.000	3.537	40	191	1
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
4596	0.31	0.00	0.62	0.0	0.00	0.31	0.00	0.00	0.00	0.00	...	0.000	0.232	0.0	0.000	0.000	0.000	1.142	3	88	0
4597	0.00	0.00	0.00	0.0	0.00	0.00	0.00	0.00	0.00	0.00	...	0.000	0.000	0.0	0.353	0.000	0.000	1.555	4	14	0
4598	0.30	0.00	0.30	0.0	0.00	0.00	0.00	0.00	0.00	0.00	...	0.102	0.718	0.0	0.000	0.000	0.000	1.404	6	118	0
4599	0.96	0.00	0.00	0.0	0.32	0.00	0.00	0.00	0.00	0.00	...	0.000	0.057	0.0	0.000	0.000	0.000	1.147	5	78	0
4600	0.00	0.00	0.65	0.0	0.00	0.00	0.00	0.00	0.00	0.00	...	0.000	0.000	0.0	0.125	0.000	0.000	1.250	5	40	0

4601 rows × 58 columns

Adding headers to the dataset

```
# Open and read the file containing the column names for the spam dataset
with open('./spambase.names') as spam:
    text = spam.read()

# Use regular expression to find the column names from the text
# The column names are enclosed in a newline character followed by one or more alphanumeric characters or underscores,
# then optionally followed by non-alphanumeric characters and a colon
labels = re.findall(r'\n(\w*_?\W?):', text)

# Read the spam dataset file into a pandas dataframe
# Specify the header as None because the column names are included in the data file
# Specify the names of the columns as the labels found earlier plus an additional 'spam' column
df = pd.read_csv('./spambase.data', header=None, names=labels + ['spam'])

df
```

	word_freq_make	word_freq_address	word_freq_all	word_freq_3d	word_freq_our	word_freq_over	word_freq_remove	word_freq_internet
0	0.00	0.64	0.64	0.0	0.32	0.00	0.00	0.00
1	0.21	0.28	0.50	0.0	0.14	0.28	0.21	0.07
2	0.06	0.00	0.71	0.0	1.23	0.19	0.19	0.12
3	0.00	0.00	0.00	0.0	0.63	0.00	0.31	0.63
4	0.00	0.00	0.00	0.0	0.63	0.00	0.31	0.63
...	...	...	...	...	...	...	...	...
4596	0.31	0.00	0.62	0.0	0.00	0.31	0.00	0.00
4597	0.00	0.00	0.00	0.0	0.00	0.00	0.00	0.00
4598	0.30	0.00	0.30	0.0	0.00	0.00	0.00	0.00
4599	0.96	0.00	0.00	0.0	0.32	0.00	0.00	0.00
4600	0.00	0.00	0.65	0.0	0.00	0.00	0.00	0.00

4601 rows × 58 columns

Checking for duplicate values

```
duplicate = df[df.duplicated()]
duplicate
```

	word_freq_make	word_freq_address	word_freq_all	word_freq_3d	word_freq_our	word_freq_over	word_freq_remove	word_freq_internet
26	0.0	0.0	0.000000	0.0	0.0	0.000000	0.0	0.0
103	0.0	0.0	0.125490	0.0	0.0	0.108844	0.0	0.0
104	0.0	0.0	0.000000	0.0	0.0	0.000000	0.0	0.0
105	0.0	0.0	0.125490	0.0	0.0	0.108844	0.0	0.0
106	0.0	0.0	0.000000	0.0	0.0	0.000000	0.0	0.0
...	...	...	...	...	...	...	...	...
4439	0.0	0.0	0.145098	0.0	0.0	0.000000	0.0	0.0
4441	0.0	0.0	0.145098	0.0	0.0	0.000000	0.0	0.0
4537	0.0	0.0	0.000000	0.0	0.0	0.000000	0.0	0.0
4541	0.0	0.0	0.000000	0.0	0.0	0.000000	0.0	0.0
4550	0.0	0.0	0.000000	0.0	0.0	0.000000	0.0	0.0

394 rows × 58 columns

Dropping duplicate data

```
df = df.drop_duplicates()
df
```

	word_freq_make	word_freq_address	word_freq_all	word_freq_3d	word_freq_our	word_freq_over	word_freq_remove	word_freq_internet
0	0.000000	0.044818	0.125490	0.0	0.032	0.000000	0.000000	0.000000
1	0.046256	0.019608	0.098039	0.0	0.014	0.047619	0.028886	0.006301
2	0.013216	0.000000	0.139216	0.0	0.123	0.032313	0.026135	0.010801
3	0.000000	0.000000	0.000000	0.0	0.063	0.000000	0.042641	0.056706
4	0.000000	0.000000	0.000000	0.0	0.063	0.000000	0.042641	0.056706
...	...	...	...	...	...	...	...	...
4596	0.068282	0.000000	0.121569	0.0	0.000	0.052721	0.000000	0.000000
4597	0.000000	0.000000	0.000000	0.0	0.000	0.000000	0.000000	0.000000
4598	0.066079	0.000000	0.058824	0.0	0.000	0.000000	0.000000	0.000000
4599	0.211454	0.000000	0.000000	0.0	0.032	0.000000	0.000000	0.000000
4600	0.000000	0.000000	0.127451	0.0	0.000	0.000000	0.000000	0.000000

4207 rows × 58 columns

Checking for any Null values

```
df.isnull().values.any()
```

```
False
```

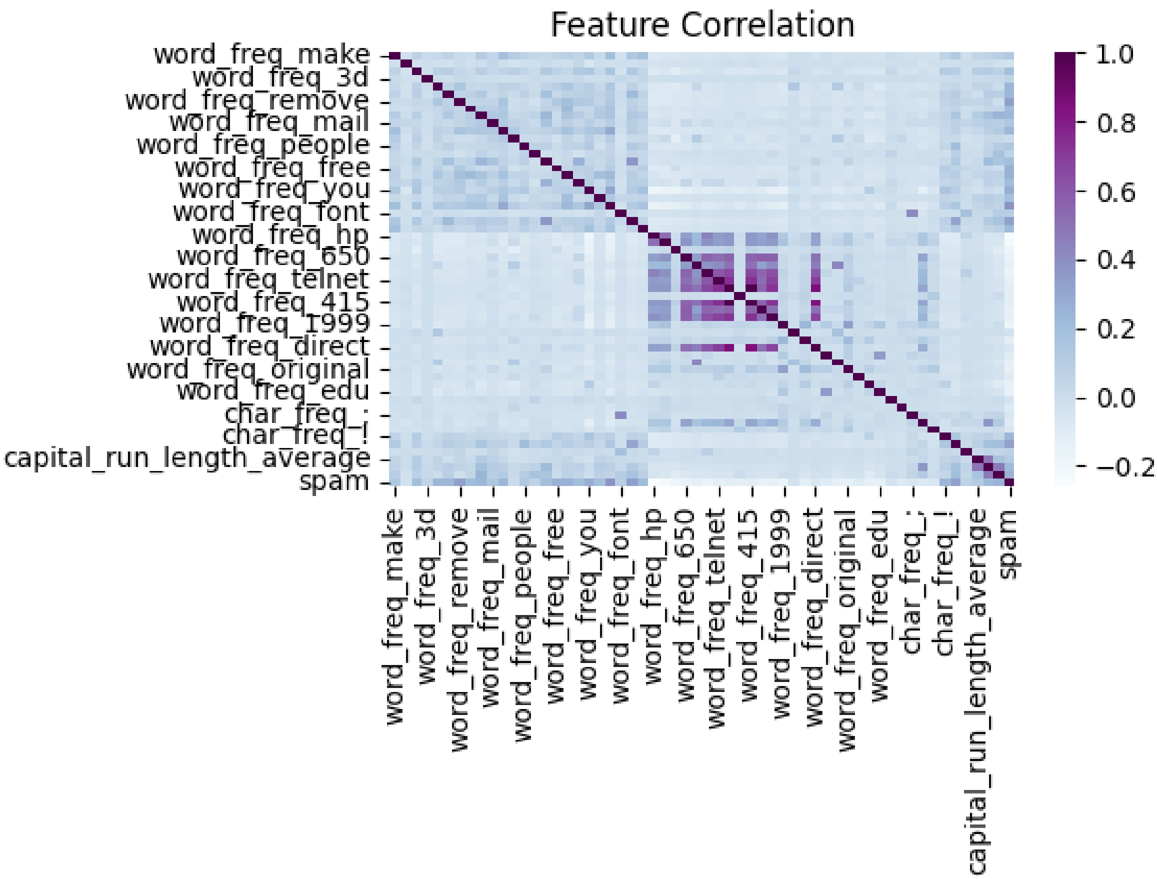
```
df.isnull().sum()
```

```
word_freq_make                0
word_freq_address             0
word_freq_all                 0
word_freq_3d                  0
word_freq_our                 0
word_freq_over                0
word_freq_remove              0
word_freq_internet            0
word_freq_order               0
word_freq_mail                0
word_freq_receive             0
word_freq_will                0
word_freq_people              0
word_freq_report              0
word_freq_addresses           0
word_freq_free                0
word_freq_business            0
word_freq_email               0
word_freq_you                 0
word_freq_credit              0
word_freq_your                0
word_freq_font                0
word_freq_000                 0
word_freq_money               0
word_freq_hp                  0
...
capital_run_length_average    0
capital_run_length_longest    0
capital_run_length_total      0
spam_nospam                   0
```

Getting a feature correlation plot

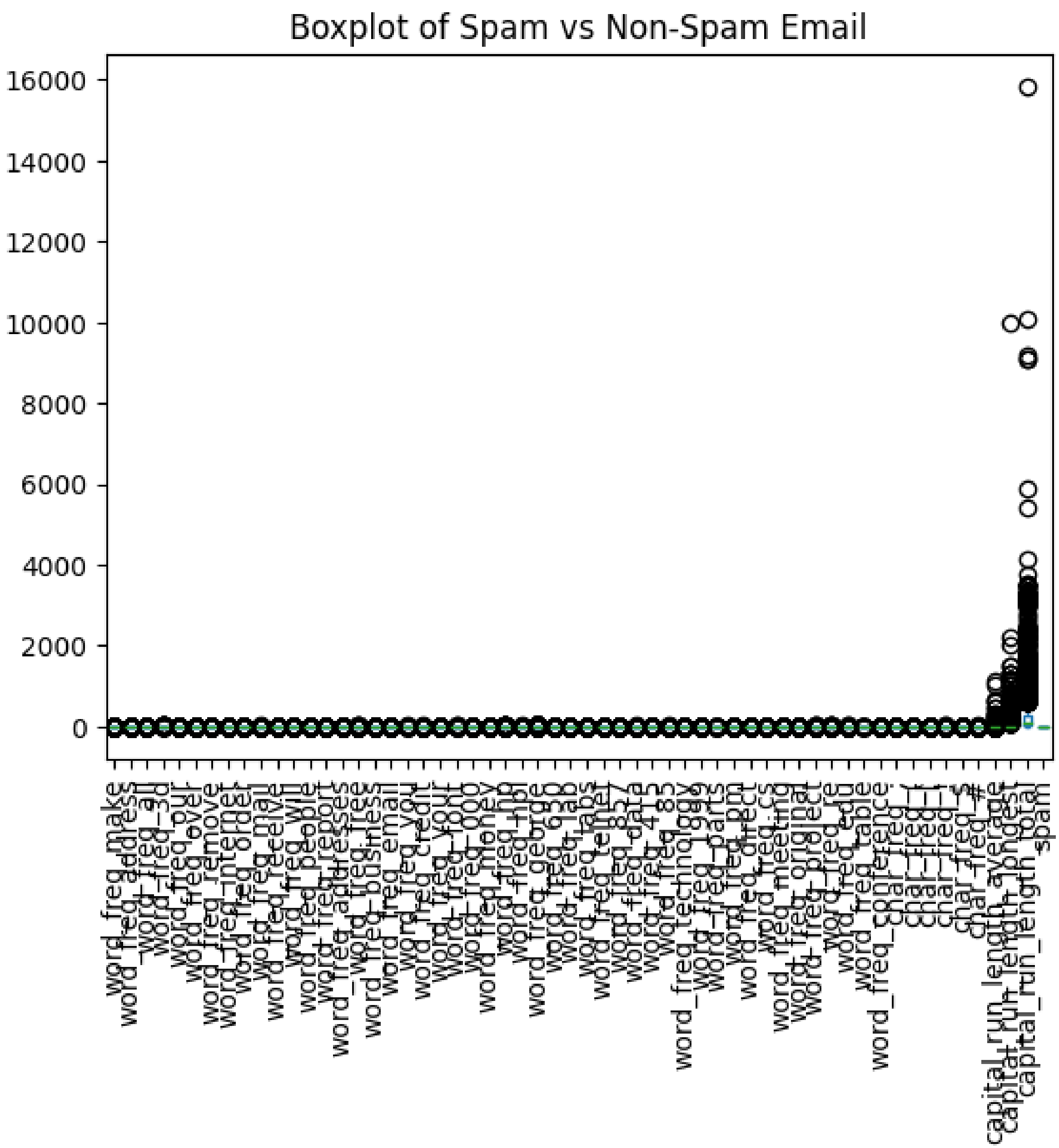
```
# Calculate the correlation matrix of the features in the dataframe
correlation = df.corr()

# The correlation matrix shows the pairwise correlations between all the features in the dataframe
# A correlation of 1 indicates a perfect positive correlation (when one feature increases, so does the other)
# A correlation of -1 indicates a perfect negative correlation (when one feature increases, the other decreases)
# A correlation of 0 indicates no correlation between the features
# Values between -1 and 1 indicate varying degrees of correlation

# Visualize the correlation matrix using a heatmap
# The 'cmap' argument specifies the color map to use for the heatmap
sn.heatmap(correlation, cmap="BuPu")

# Add a title to the heatmap
plt.title("Feature Correlation")

# Ensure tight layout of the heatmap in the figure
```

```
plt.tight_layout()


# Show the heatmap
plt.show()
```



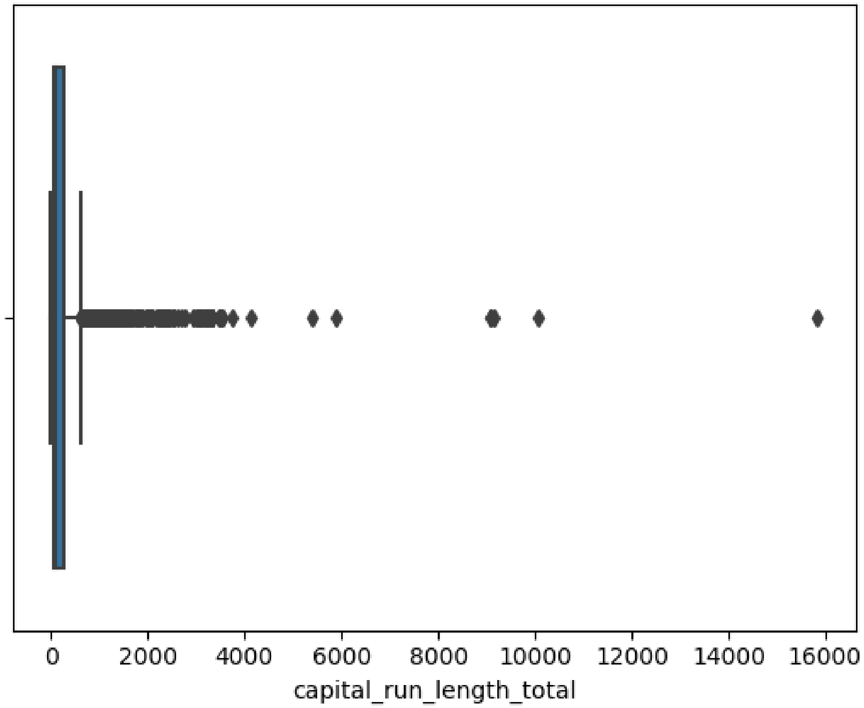
Checking for outliers

```
# Create a boxplot of the dataframe
# The 'title' argument specifies the title of the plot
# The 'rot' argument specifies the rotation angle of the x-axis labels
df.plot.box(title='Boxplot of Spam vs Non-Spam Email', rot=90)


# Show the boxplot
plt.show()
```



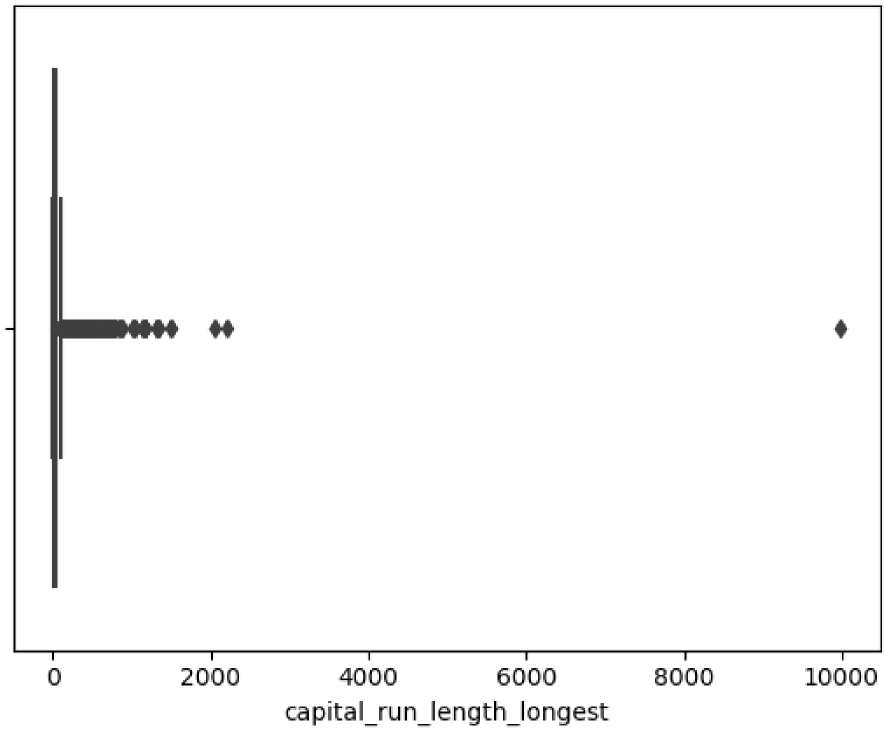
Boxplot of capital_run_length_total

```
sn.boxplot(x = df['capital_run_length_total'])
```



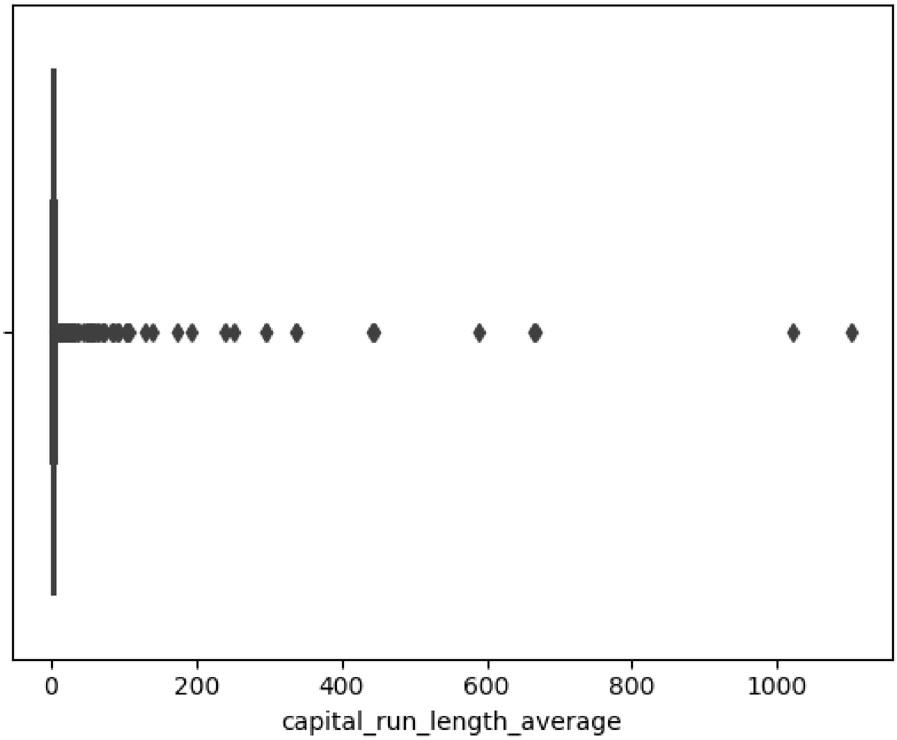
Boxplot of capital_run_length_longest

```
sn.boxplot(x = df['capital_run_length_longest'])
```



Boxplot of capital_run_length_average

```
sn.boxplot(x = df['capital_run_length_average'])
```

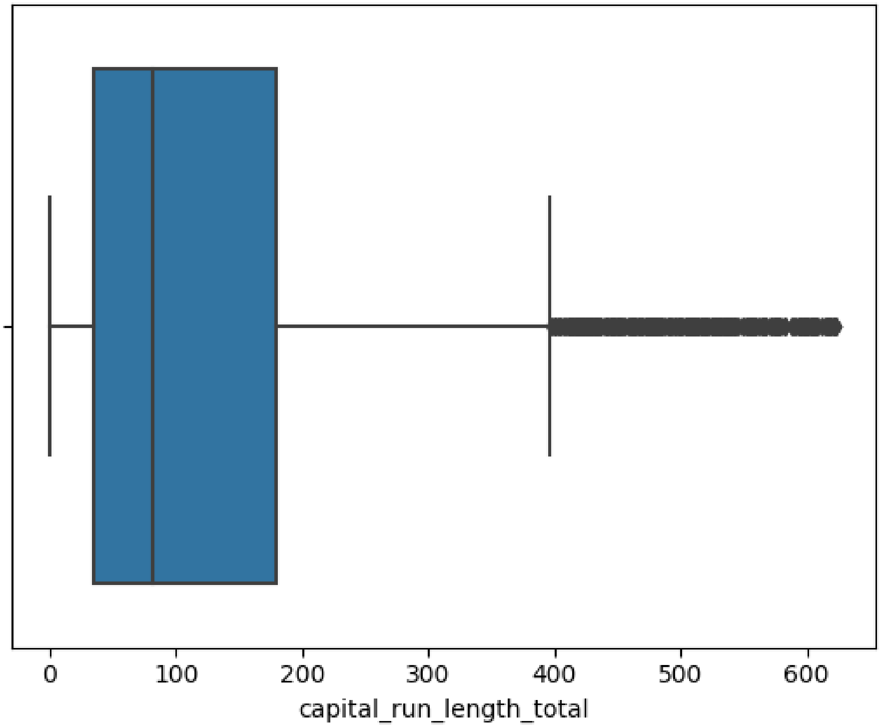


Making all the outliers as Null values from IQR technique

```
# For each of the three specified features, calculate the 75th and 25th percentiles
# using the numpy percentile function
for x in ['capital_run_length_total','capital_run_length_longest','capital_run_length_average']:
    q75, q25 = np.percentile(df.loc[:, x], [75, 25])

    # Calculate the interquartile range (IQR) of the feature
    intr_qr = q75 - q25

    # Calculate the maximum and minimum values allowed for the feature
    max_val = q75 + (1.5 * intr_qr)
    min_val = q25 - (1.5 * intr_qr)

    # Replace any values below the minimum or above the maximum with NaN
    df.loc[df[x] < min_val, x] = np.nan
    df.loc[df[x] > max_val, x] = np.nan
```

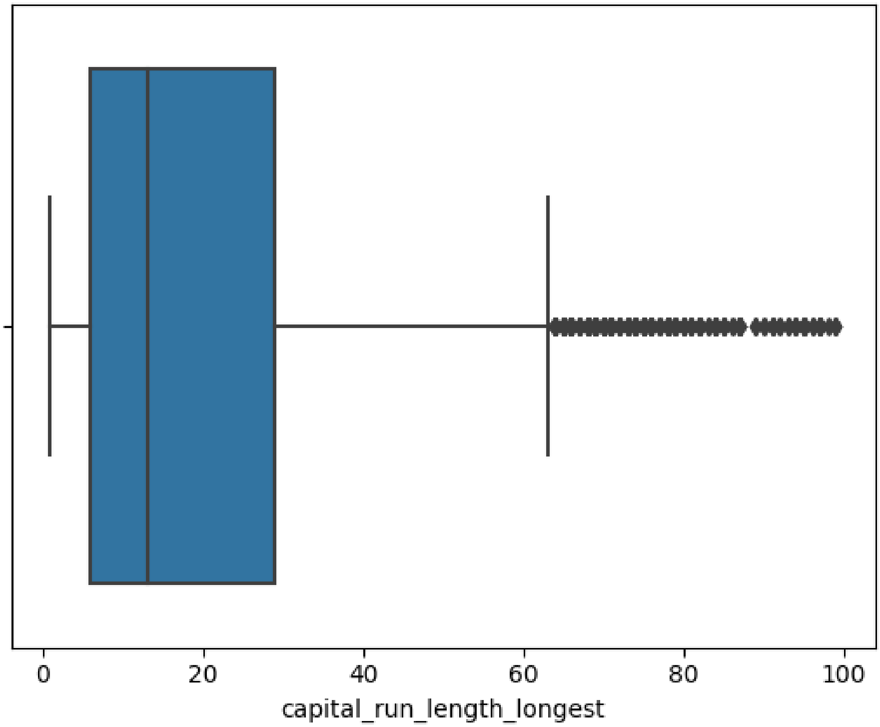
Boxplot of capital_run_length_total without outliers

```
sn.boxplot(x = df['capital_run_length_total'])
```



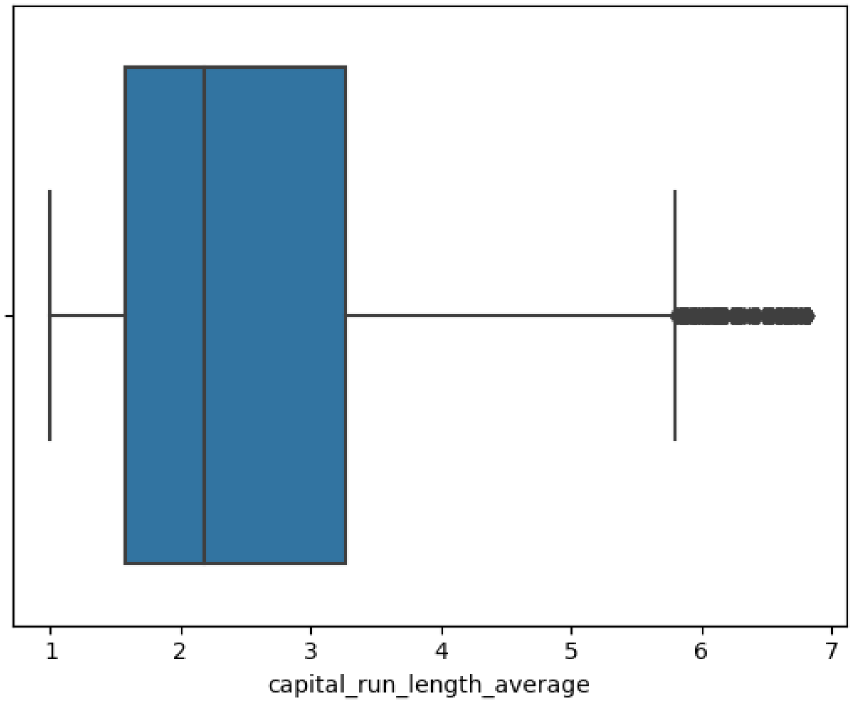
Boxplot of capital_run_length_longest without outliers

```
sn.boxplot(x = df['capital_run_length_longest'])
```



Boxplot of capital_run_length_average without outliers

```
sn.boxplot(x = df['capital_run_length_average'])
```



Checking for Null values after turning all the outliers as Null values

```
df.isnull().values.any()
```

```
True
```

```
df.isnull().sum()
```

```
word_freq_make                 0
word_freq_address              0
word_freq_all                  0
word_freq_3d                   0
word_freq_our                  0
word_freq_over                 0
word_freq_remove               0
word_freq_internet             0
word_freq_order                0
word_freq_mail                 0
word_freq_receive              0
word_freq_will                 0
word_freq_people               0
word_freq_report               0
word_freq_addresses            0
word_freq_free                 0
word_freq_business             0
word_freq_email                0
word_freq_you                  0
word_freq_credit               0
word_freq_your                 0
word_freq_font                 0
word_freq_000                  0
word_freq_money                0
word_freq_hp                   0
...
capital_run_length_average   329
capital_run_length_longest   418
capital_run_length_total     497
spam_nospam                    0
```

Removing all the Null values

```
# Drop all rows that contain NaN values from the dataframe
df2 = df.dropna()

# Reset the index of the dataframe after dropping NaN values
df2 = df.dropna().reset_index(drop=True)

df = df2
df
```

	word_freq_make	word_freq_address	word_freq_all	word_freq_3d	word_freq_our	word_freq_over	word_freq_remove	word_freq_internet
0	0.00	0.64	0.64	0.0	0.32	0.00	0.00	0.00
1	0.00	0.00	0.00	0.0	0.63	0.00	0.31	0.63
2	0.00	0.00	0.00	0.0	0.63	0.00	0.31	0.63
3	0.00	0.00	0.00	0.0	1.85	0.00	0.00	1.85
4	0.00	0.00	0.00	0.0	1.92	0.00	0.00	0.00
...	...	...	...	...	...	...	...	...
3441	0.31	0.00	0.62	0.0	0.00	0.31	0.00	0.00
3442	0.00	0.00	0.00	0.0	0.00	0.00	0.00	0.00
3443	0.30	0.00	0.30	0.0	0.00	0.00	0.00	0.00
3444	0.96	0.00	0.00	0.0	0.32	0.00	0.00	0.00
3445	0.00	0.00	0.65	0.0	0.00	0.00	0.00	0.00

3446 rows × 58 columns

Converting the dataframe into a numpy array

```
dataset = df.to_numpy()
print(dataset,dataset.shape)
```

```
[[  0.      0.64    0.64 ...  61.    278.      1.  ]
 [  0.      0.      0.   ...  40.    191.      1.  ]
 [  0.      0.      0.   ...  40.    191.      1.  ]
 ...
 [  0.3     0.      0.3  ...   6.    118.      0.  ]
 [  0.96    0.      0.   ...   5.     78.      0.  ]
 [  0.      0.      0.65 ...   5.     40.      0.  ]] (3446, 58)
```

Separating the features of the dataset as the X variable

```
[[  0.      0.64    0.64   ...   3.756  61.    278.   ]
 [  0.      0.      0.     ...   3.537  40.    191.   ]
 [  0.      0.      0.     ...   3.537  40.    191.   ]
 ...
 [  0.3     0.      0.3    ...   1.404   6.    118.   ]
 [  0.96    0.      0.     ...   1.147   5.     78.   ]
 [  0.      0.      0.65   ...   1.25    5.     40.   ]] (3446, 57)
```

Separating the labels of the dataset as the y variable

```
y = dataset[:,57]
print(y,y.shape)
```

```
[1. 1. 1. ... 0. 0. 0.] (3446,)
```

Normalizing Dataset

Before normalizing

```
df.describe()
```

	word_freq_make	word_freq_address	word_freq_all	word_freq_3d	word_freq_our	word_freq_over	word_freq_remove	word_freq_internet
count	3446.000000	3446.000000	3446.000000	3446.000000	3446.000000	3446.000000	3446.000000	3446.000000
mean	0.094779	0.092716	0.268056	0.005818	0.308175	0.085267	0.093688	0.096164
std	0.309801	0.474629	0.529981	0.134848	0.701947	0.281174	0.356036	0.420321
min	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000
25%	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000
50%	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000
75%	0.000000	0.000000	0.360000	0.000000	0.360000	0.000000	0.000000	0.000000
max	4.540000	14.280000	5.100000	7.070000	10.000000	5.880000	7.270000	11.110000

8 rows × 58 columns

Normalizing

```
# Instantiate a MinMaxScaler object with a feature range of (0, 1)
scaler = preprocessing.MinMaxScaler(feature_range=(0, 1))

# Use the fit_transform method to scale the original dataframe and store the result in a new variable
normalized_scale = scaler.fit_transform(df)

# Create a new dataframe using the scaled data, with the same indices and column names as the original dataframe
df_scale = pd.DataFrame(normalized_scale, index=df.index, columns=df.columns)

# Overwrite the original dataframe with the scaled data
df = df_scale
```

After normalizing

```
df.describe()
```

	word_freq_make	word_freq_address	word_freq_all	word_freq_3d	word_freq_our	word_freq_over	word_freq_remove	word_freq_internet
count	4601.000000	4601.000000	4601.000000	4601.000000	4601.000000	4601.000000	4601.000000	4601.000000
mean	0.023029	0.014917	0.055031	0.001528	0.031222	0.016310	0.015709	0.009477
std	0.067259	0.090376	0.098852	0.032589	0.067251	0.046569	0.053843	0.036100
min	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000
25%	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000
50%	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000
75%	0.000000	0.000000	0.082353	0.000000	0.038000	0.000000	0.000000	0.000000
max	1.000000	1.000000	1.000000	1.000000	1.000000	1.000000	1.000000	1.000000

8 rows × 58 columns

Removing the label colum from the dataframe

```
df.drop('spam_nospam', axis=1, inplace=True)
df
```

	word_freq_make	word_freq_address	word_freq_all	word_freq_3d	word_freq_our	word_freq_over	word_freq_remove	word_freq_internet
0	0.000000	0.044818	0.125490	0.0	0.032	0.000000	0.000000	0.000000
1	0.000000	0.000000	0.000000	0.0	0.063	0.000000	0.042641	0.056706
2	0.000000	0.000000	0.000000	0.0	0.063	0.000000	0.042641	0.056706
3	0.000000	0.000000	0.000000	0.0	0.185	0.000000	0.000000	0.166517
4	0.000000	0.000000	0.000000	0.0	0.192	0.000000	0.000000	0.000000
...	...	...	...	...	...	...	...	...
3441	0.068282	0.000000	0.121569	0.0	0.000	0.052721	0.000000	0.000000
3442	0.000000	0.000000	0.000000	0.0	0.000	0.000000	0.000000	0.000000
3443	0.066079	0.000000	0.058824	0.0	0.000	0.000000	0.000000	0.000000
3444	0.211454	0.000000	0.000000	0.0	0.032	0.000000	0.000000	0.000000
3445	0.000000	0.000000	0.127451	0.0	0.000	0.000000	0.000000	0.000000

3446 rows × 57 columns

Converting the above dataframe to X variable as features

```
X = df.to_numpy()
print(X,X.shape)
```

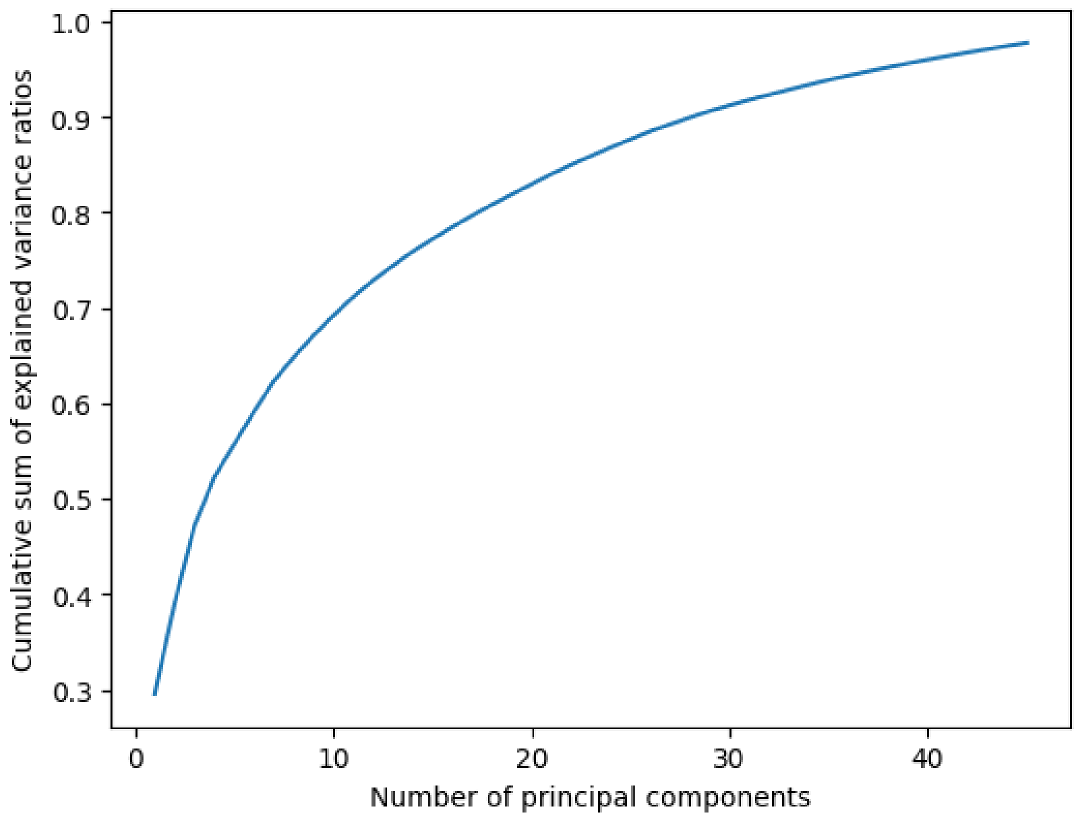
```
[[0.         0.04481793 0.1254902  ... 0.47337685 0.6185567  0.44533762]
 [0.         0.         0.         ... 0.43576091 0.40206186 0.30546624]
 [0.         0.         0.         ... 0.43576091 0.40206186 0.30546624]
 ...
 [0.0660793  0.         0.05882353 ... 0.06939196 0.05154639 0.18810289]
 [0.21145374 0.         0.         ... 0.02524906 0.04123711 0.12379421]
 [0.         0.         0.12745098 ... 0.04294057 0.04123711 0.06270096]] (3446, 57)
```

Create a PCA instance

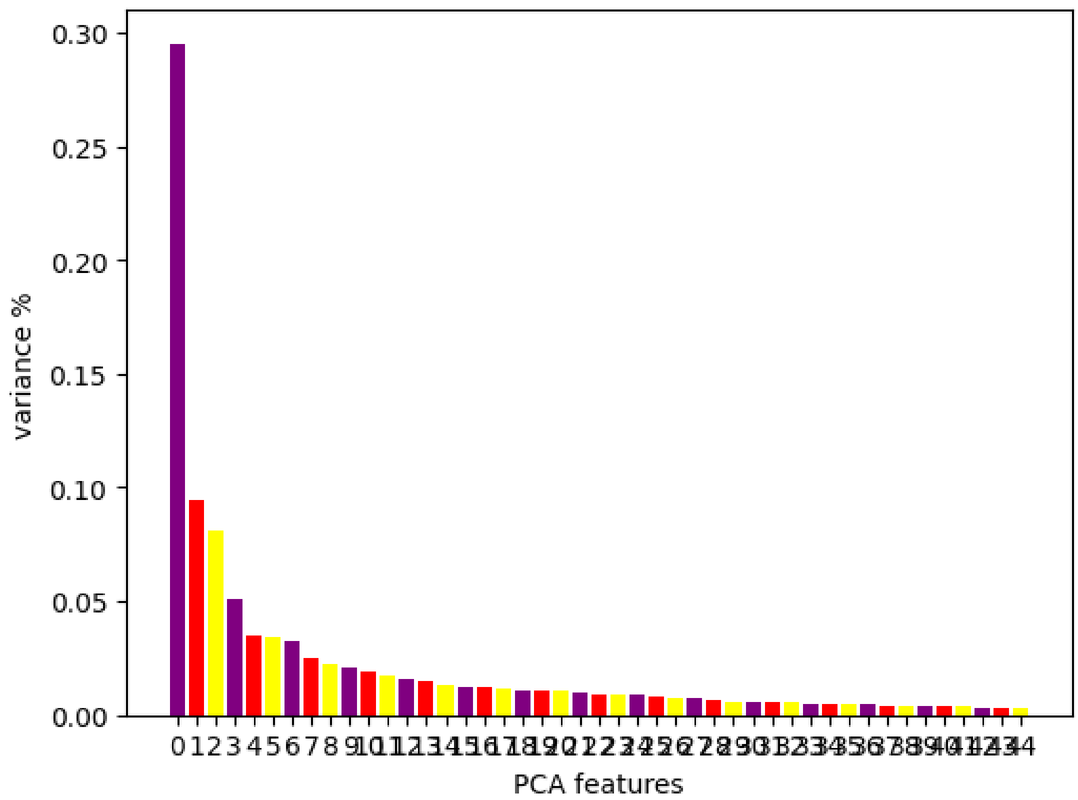
```
# Instantiate a PCA object with n_components = 45 as when taking 90% varience, 45
components
# make most of the effect on the final result
pca = PCA(n_components=45)

# Used to plot the explained_variance_ratio_ histogram
# Use the fit_transform method to fit the PCA model to the data and transform the data
pca_Components = pca.fit_transform(df)

# Print the shape of the transformed data
print(pca_Components.shape)
```

Plot the explained variances

```
# Calculate the cumulative sum of explained variance ratios using the cumsum
function from numpy
cumulative_variances = np.cumsum(pca.explained_variance_ratio_)

# Plot the cumulative sum of explained variance ratios using matplotlib
plt.plot(range(1, len(cumulative_variances) + 1), cumulative_variances)
plt.xlabel("Number of principal components")
plt.ylabel("Cumulative sum of explained variance ratios")
plt.show()
```




```
# Define a list of features to be used in the plot
features = range(pca.n_components_)

# Plot the explained variances for each principal component using the bar function from matplotlib
plt.bar(features, pca.explained_variance_ratio_, color=('purple','red','yellow'))

# Add labels to the x-axis and y-axis using the xlabel and ylabel functions
plt.xlabel('PCA features')
plt.ylabel('variance %')

# Set the tick labels on the x-axis to the feature numbers using the xticks function
plt.xticks(features)

# Print the list of features
print(features)
```



Save components to a DataFrame

```
PCA_components = pd.DataFrame(pca_Components)

PCA_components
```

	0	1	2	3	4	5	6	7	8	9	...
0	0.537848	-0.035220	-0.056560	0.032648	0.099548	-0.121080	-0.054140	0.003357	0.020350	-0.043943	...
1	0.307380	0.009136	-0.088984	-0.020581	-0.040519	-0.076328	0.002362	0.082825	-0.048081	-0.009284	...
2	0.307377	0.009102	-0.089009	-0.020580	-0.040522	-0.076340	0.002372	0.082821	-0.048075	-0.009267	...
3	-0.028438	0.084251	-0.102778	-0.128006	-0.028127	0.011627	-0.018201	-0.075278	-0.062237	-0.051100	...
4	-0.149706	-0.121434	-0.000343	0.084991	-0.062061	0.036545	0.134959	0.088275	-0.062977	-0.009157	...
...	...	...	...	...	...	...	...	...	...	...	...
3441	-0.243629	-0.117062	0.074935	-0.018885	0.133932	0.032479	0.068488	-0.002150	-0.032634	-0.024342	...
3442	-0.263743	-0.062001	-0.111827	0.128734	-0.144539	-0.014978	-0.019637	0.123884	0.028551	-0.016320	...
3443	-0.170922	-0.110308	0.075334	-0.006718	0.066067	0.022291	0.087727	0.029098	-0.024002	-0.028395	...
3444	-0.240866	-0.108070	0.043830	-0.009824	-0.042867	-0.032102	0.002093	0.013218	-0.046409	-0.001177	...
3445	-0.267650	-0.099056	-0.038454	0.091422	-0.002482	-0.008885	-0.099601	0.114974	-0.018088	0.017434	...

3446 rows × 45 columns

Performing PCA to the features dataset "X"

```
X = pca.fit_transform(X)
```

Splitting the dataset

```
# Split the data and target into training and testing sets using train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.30)


# Print the training data and its shape
```

```
print("\ntrain data :\n", X_train, X_train.shape)

# Print the testing data and its shape
print("\ntest data :\n", X_test, X_test.shape)

# Print the training target and its shape
print("\ntrain target :\n", y_train, y_train.shape)

# Print the testing target and its shape
print("\ntest target :", y_test, y_test.shape)
```

```
train data :
 [[-0.19937435  0.03921395  0.09182268 ... -0.00930954 -0.01748444
  -0.01886634]
 [-0.19141223 -0.02895166 -0.0597452  ...  0.00482977  0.0115138
  -0.00172018]
 [-0.16614551  0.01477345 -0.05272037 ...  0.06216313  0.00647848
  -0.04968411]
 ...
 [-0.28334652 -0.03837933  0.00749852 ... -0.01229992 -0.00189587
  -0.01501963]
 [-0.25920829 -0.06948218 -0.0717862  ...  0.00817339  0.00066739
   0.04024868]
 [-0.18869497  0.00630534  0.10588686 ... -0.01278499  0.0188192
   0.01966653]] (2412, 45)

test data :
 [[-2.60149074e-01 -4.46970439e-02  6.32355016e-02 ...  8.94733059e-03
  -4.83991490e-03 -7.22273336e-06]
 [-2.41019775e-01 -4.08865869e-02 -8.41379321e-02 ...  4.26029110e-02
   2.91971151e-02 -3.12993164e-02]
 [ 7.13090172e-01  2.29559836e-01 -3.04480049e-01 ... -1.35581275e-02
   9.46964161e-03 -3.34152553e-03]
 ...
 [ 2.08886173e-01  1.33606329e-01 -3.50217444e-01 ... -1.28358828e-02
...
train target :
 [0. 0. 0. ... 0. 1. 0.] (2412,)

test target : [0. 0. 1. ... 1. 1. 0.] (1034,)
```

Decision Tree Classifier Model

```
# Create an instance of the DecisionTreeClassifier class
model_dt = DecisionTreeClassifier()
```

Performing Hyperparameter Tuning

```
# Define the hyperparameter grid to search over
param_grid = {
    "max_depth": [2, 4, 6, 8, 10],
    "min_samples_split": [2, 4, 6, 8, 10],
    "min_samples_leaf": [1, 2, 3, 4, 5]
}

# Create an instance of the GridSearchCV class with the decision tree model and hyperparameter grid
grid_search = GridSearchCV(model_dt, param_grid, cv=5)

# Fit the GridSearchCV instance to the training data
grid_search.fit(X_train, y_train)
```

```
# Print the best hyperparameters and corresponding score found by GridSearchCV
print("Best hyperparameters: ", grid_search.best_params_)
print("Best score: ", grid_search.best_score_)
```

Passing the above mentioned parameters to model

```
model_dt = grid_search.best_estimator_
```

Training the model

```
model_dt.fit(X_train,y_train)
```

```
▾ DecisionTreeClassifier
DecisionTreeClassifier(max_depth=10, min_samples_leaf=4)
```

Predicting

```
y_predict_dt = model_dt.predict(X_test)
```

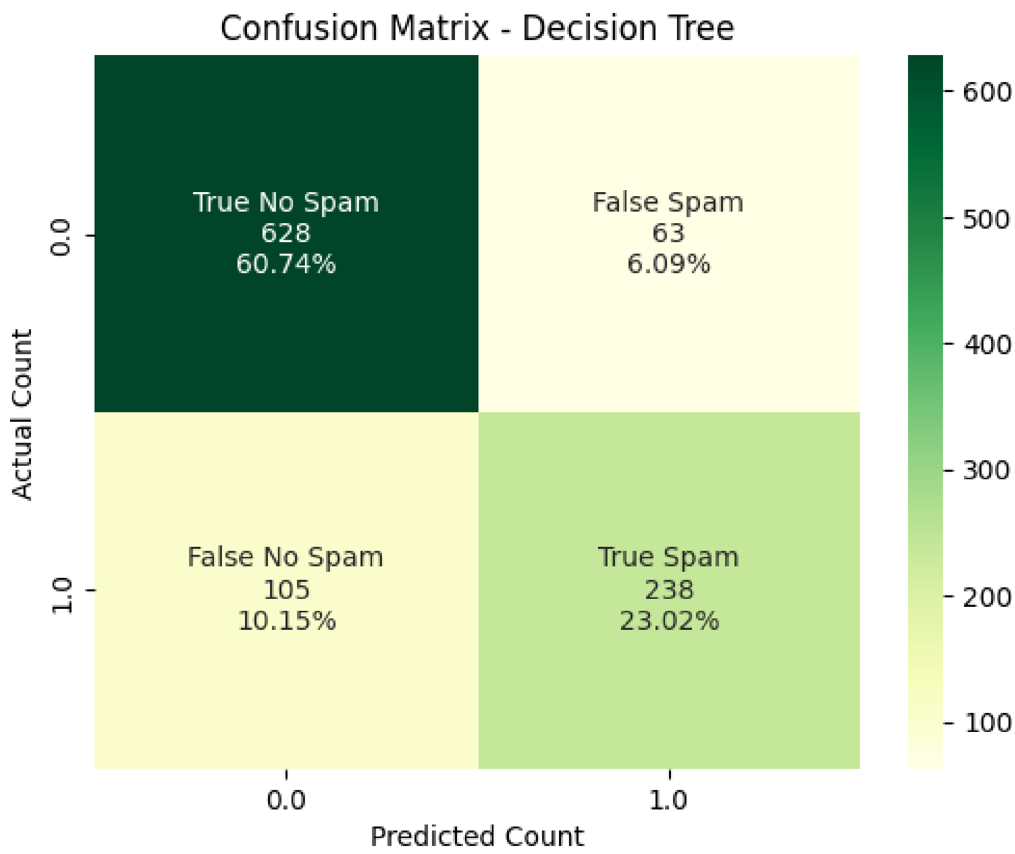
Model accuracy

```
# Model Validation Accuracy
accuracy = accuracy_score(y_test,y_predict_dt)
print("accuracy : ",accuracy)

# Model Confusion Matrix
conf_mat_dt = confusion_matrix(y_test, y_predict_dt)
print("\nconfusion matrix : \n",conf_mat_dt)

# Model Classification Report
clf_report = classification_report(y_test, y_predict_dt)
print("\nClassification Report : ")
print(clf_report)

# Model Cross Validation Score
score = cross_val_score(model_dt, X, y, cv=3)
print("\nCross Validation Score : ",score)
```

```
accuracy :  0.8375241779497099

confusion matrix :
 [[628  63]
 [105 238]]

Classification Report :
              precision    recall  f1-score   support

         0.0       0.86      0.91      0.88       691
         1.0       0.79      0.69      0.74       343

    accuracy                           0.84      1034
   macro avg       0.82      0.80      0.81      1034
weighted avg       0.83      0.84      0.83      1034


Cross Validation Score :  [0.83637946 0.82767624 0.74303136]
```

```
# Generate the confusion matrix for the model


# The confusion_matrix function from scikit-learn is used to calculate the confusion matrix.
# The labels parameter is set to model_dt.classes_ to ensure that the labels in the confusion matrix
# match the classes in the model.
conf_mat_dt = confusion_matrix(y_test, y_predict_dt,labels=model_dt.classes_)

# calculates the group names, group counts, and group percentages for each cell in the confusion matrix.
# These values are used to create the annotations for each cell in the matrix.
group_names = ['True No Spam','False Spam',
            'False No Spam','True Spam']

group_counts = ["{0:0.0f}".format(value) for value in
            conf_mat_dt.flatten()]
group_percentages = ["{0:.2%}".format(value) for value in
             conf_mat_dt.flatten()/np.sum(conf_mat_dt)]
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)

# the sns.heatmap function from the seaborn library is used to plot the confusion matrix with the annotations.
# The yticklabels and xticklabels parameters are set to model_dt.classes_ to ensure that the labels on the y and x
axes
# match the classes in the model. The title, xlabel, and ylabel parameters are used to set the title and labels for
the
# plot.
ax = sn.heatmap(conf_mat_dt, annot=labels,yticklabels=model_dt.classes_,xticklabels=model_dt.classes_, fmt='',
cmap='YlGn')
ax.set(
        title='Confusion Matrix - Decision Tree',
        xlabel='Predicted Count',
        ylabel='Actual Count')
ax.plot
```



K Nearest Neighbour Classifier Model

```
model_Knn = KNeighborsClassifier()
```

Performing hyperparameter tuning

```
# Define the parameter grid for tuning the hyperparameters
param_grid_K = {'n_neighbors': [3, 5, 7, 9, 11], 'weights': ['uniform', 'distance']}

# Create a GridSearchCV object to tune the hyperparameters using cross-validation
grid_search_K = GridSearchCV(model_Knn, param_grid_K, cv=5)

# Fit the GridSearchCV instance to the training data
grid_search_K.fit(X_train, y_train)

# Print the best parameters and the best score
print("Best parameters: ", grid_search_K.best_params_)
print("Best score: ", grid_search_K.best_score_)
```

Passing the above mentioned parameters to model

```
model_Knn = grid_search_K.best_estimator_
```

Training the Model

```
model_Knn.fit(X_train,y_train)
```

Predicting

```
y_predict_Knn = model_Knn.predict(X_test)
```

Model accuracy

```
# Model Validation Accuracy
accuracy = accuracy_score(y_test,y_predict_Knn)
print("accuracy : ",accuracy)

# Model Confusion Matrix
conf_mat_knn = confusion_matrix(y_test, y_predict_Knn)
print("\nconfusion matrix : \n",conf_mat_knn)

# Model Classification Report
clf_report = classification_report(y_test, y_predict_Knn)
print("\nClassification Report : ")
print(clf_report)
```

```
# Model Cross Validation Score
score = cross_val_score(model_Knn, X, y, cv=3)
print("\nCross Validation Score : ",score)
```

```
accuracy :  0.8839458413926499

confusion matrix :
 [[650  41]
 [ 79 264]]

Classification Report :
              precision    recall  f1-score   support

         0.0       0.89      0.94      0.92       691
         1.0       0.87      0.77      0.81       343

    accuracy                           0.88      1034
   macro avg       0.88      0.86      0.87      1034
weighted avg       0.88      0.88      0.88      1034


Cross Validation Score :  [0.86335944 0.86597041 0.80400697]
```

```
# Generate the confusion matrix for the model

# The confusion_matrix function from scikit-learn is used to calculate the confusion matrix.
# The labels parameter is set to model_dt.classes_ to ensure that the labels in the confusion matrix
# match the classes in the model.
conf_mat_knn = confusion_matrix(y_test, y_predict_Knn,labels=model_Knn.classes_)

# calculates the group names, group counts, and group percentages for each cell in the confusion matrix.
# These values are used to create the annotations for each cell in the matrix.
group_names = ['True No Spam','False Spam',
          'False No Spam','True Spam']

group_counts = ["{0:0.0f}".format(value) for value in
           conf_mat_knn.flatten()]
```

```
group_percentages = ["{0:.2%}".format(value) for value in
                    conf_mat_knn.flatten()/np.sum(conf_mat_knn)]
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)

# the sns.heatmap function from the seaborn library is used to plot the confusion matrix with the annotations.
# The yticklabels and xticklabels parameters are set to model_dt.classes_ to ensure that the labels on the y and x
axes
# match the classes in the model. The title, xlabel, and ylabel parameters are used to set the title and labels for
the
# plot.
ax = sn.heatmap(conf_mat_knn, annot=labels,yticklabels=model_Knn.classes_,xticklabels=model_Knn.classes_,
fmt='', cmap='YlGn')
ax.set(
        title='Confusion Matrix - KNN',
        xlabel='Predicted Count',
        ylabel='Actual Count')
ax.plot
```