Spam Email Detection using Machine Learning Algorithms

1. Introduction

This report presents the results of implementing machine learning (ML) algorithms such as Decision Trees, Random Forest, and Support Vector Machine for email spam detection. The dataset selected for the email spam detection is UCI Spambase that can be accessed from the link (https://doi.org/10.24432/C53G6X). The dataset has 4601 emails with 57 features and the last column of data shows whether the e-mail is labelled spam or not. The features (1-48) represent the percentage occurrence of specific words in the email text such as make, address, free, money, business, credit, you, your, etc. Each value is calculated as:

$$Word\ Frequency = \frac{number\ of\ times\ word\ appear\ in\ the\ email}{Total\ number\ of\ words\ in\ the\ email} \times 100$$

The features (49-54) represent the percentage of special characters (; ([! $ #) in the email text and values are calculated in the same manner as for features (1-48). The features (55-57) measure patterns related to the use of uppercase letters, which are often used in spam emails for emphasis, such as:

- **Feature 55: Average length of capital letter runs.**
    - Average number of consecutive uppercase letters in a sequence.
- **Feature 56: Longest capital letter run.**
    - Maximum number of consecutive uppercase letters in any sequence.
- **Feature 57: Total capital letter occurrences.**
    - Total number of uppercase letters in the email.

Out of 4601 emails, 1813 (39.4%) messages are spam and 2788 (60.6%) are not spam.

2. Data Analysis using Python

To perform data analysis in Python we will start by importing the different libraries mentioned below:

```
import pandas as pd
import numpy as np
import seaborn as sns
from matplotlib import pyplot as plt
from scipy.stats import chi2_contingency
```

```
from sklearn.preprocessing import StandardScaler, MinMaxScaler
from sklearn.model_selection import train_test_split
from statsmodels.formula.api import logit
from sklearn import metrics
from sklearn.metrics import classification_report
from sklearn.model_selection import cross_val_score
from sklearn.ensemble import RandomForestClassifier
from sklearn.tree import DecisionTreeClassifier
```

We can set the names of the columns (features) and read the data by using the code below.

```
names_list_filepath = 'spambase/names.txt'
attribute_names = []

with open(names_list_filepath, 'r') as file:
    attribute_names = file.read().splitlines()

data = pd.read_csv('spambase/spambase.data', names=attribute_names)
data
```

The output of the above code is depicted below.

Out[ ]:

	word_freq_make	word_freq_address	word_freq_all	word_freq_3d	word_freq_our	word_freq_over	word_freq_remove	word_freq_internet	word_freq_order	word_freq_mail	...	char_freq_;	char_freq_(	char_freq_[	char_freq_!	char_freq_$	char_freq_#	capital_run_length_average	capital_run_length_longest	capital_run_length_total	Class
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

We can count the number of spam and non-spam messages by using the code below.

```
class_counts = data['Class'].value_counts()
print(class_counts)
```

The output of the above code is depicted below that shows 1813 messages out of the total 4601 emails are spam.

```
Class
0   2788
1   1813
Name: count, dtype: int64
```

We can use the command below to get summary statistics of all numerical columns in the data.

```
data.describe()
```

`describe()` generates descriptive statistics, including:

- Count → Number of non-null values in each column.

- Mean → Average value of each column.
- Std (Standard Deviation) → Spread of the data.
- Min → Minimum value.
- 25% (Q1) → First quartile (25th percentile).
- 50% (Median, Q2) → Middle value (50th percentile).
- 75% (Q3) → Third quartile (75th percentile).
- Max → Maximum value in the column.

Out[ ]:

	word_freq_make	word_freq_address	word_freq_all	word_freq_3d	word_freq_our	word_freq_over	word_freq_remove	word_freq_internet	word_freq_order	word_freq_mail	...	word_freq_conference	char_freq_;	char_freq_(	char_freq_[	char_freq_!	char_freq_$	char_freq_#	capital_run_length_average	capital_run_length_longest	capital_run_length_total
count	4601.000000	4601.000000	4601.000000	4601.000000	4601.000000	4601.000000	4601.000000	4601.000000	4601.000000	4601.000000	...	4601.000000	4601.000000	4601.000000	4601.000000	4601.000000	4601.000000	4601.000000	4601.000000	4601.000000	4601.000000
mean	0.104553	0.213015	0.280656	0.065425	0.312223	0.095901	0.114208	0.105295	0.090067	0.239413	...	0.031869	0.038575	0.139030	0.016976	0.269071	0.075811	0.044238	5.191515	52.172789	283.289285
std	0.305358	1.290575	0.504143	1.395151	0.672513	0.273824	0.391441	0.401071	0.278616	0.644755	...	0.285735	0.243471	0.270355	0.109394	0.815672	0.245882	0.429342	31.729449	194.891310	606.347851
min	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	...	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	1.000000	1.000000	1.000000
25%	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	...	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	1.588000	6.000000	35.000000
50%	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	...	0.000000	0.000000	0.065000	0.000000	0.000000	0.000000	0.000000	2.276000	15.000000	95.000000
75%	0.000000	0.000000	0.420000	0.000000	0.380000	0.000000	0.000000	0.000000	0.000000	0.160000	...	0.000000	0.000000	0.188000	0.000000	0.315000	0.052000	0.000000	3.706000	43.000000	266.000000
max	4.540000	14.280000	5.100000	42.810000	10.000000	5.880000	7.270000	11.110000	5.260000	18.180000	...	10.000000	4.385000	9.752000	4.081000	32.478000	6.003000	19.829000	1102.500000	9989.000000	15841.000000

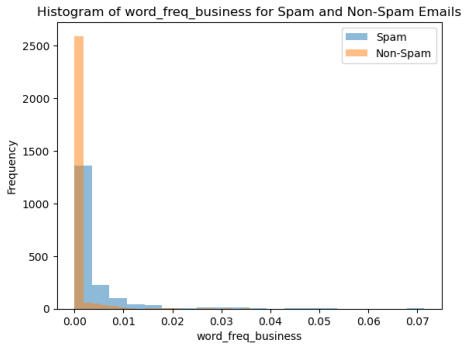
8 rows × 57 columns

We can filter spam and non-spam messages using the code below and then can run descriptive analysis on spam and non-spam emails.

```
spam = data[data['spam'] == True]
non_spam = data[data['spam'] == False]
spam.describe()
non_spam.describe()
```

We can create a function to plot the Histograms for spam and non-spam emails against any feature.

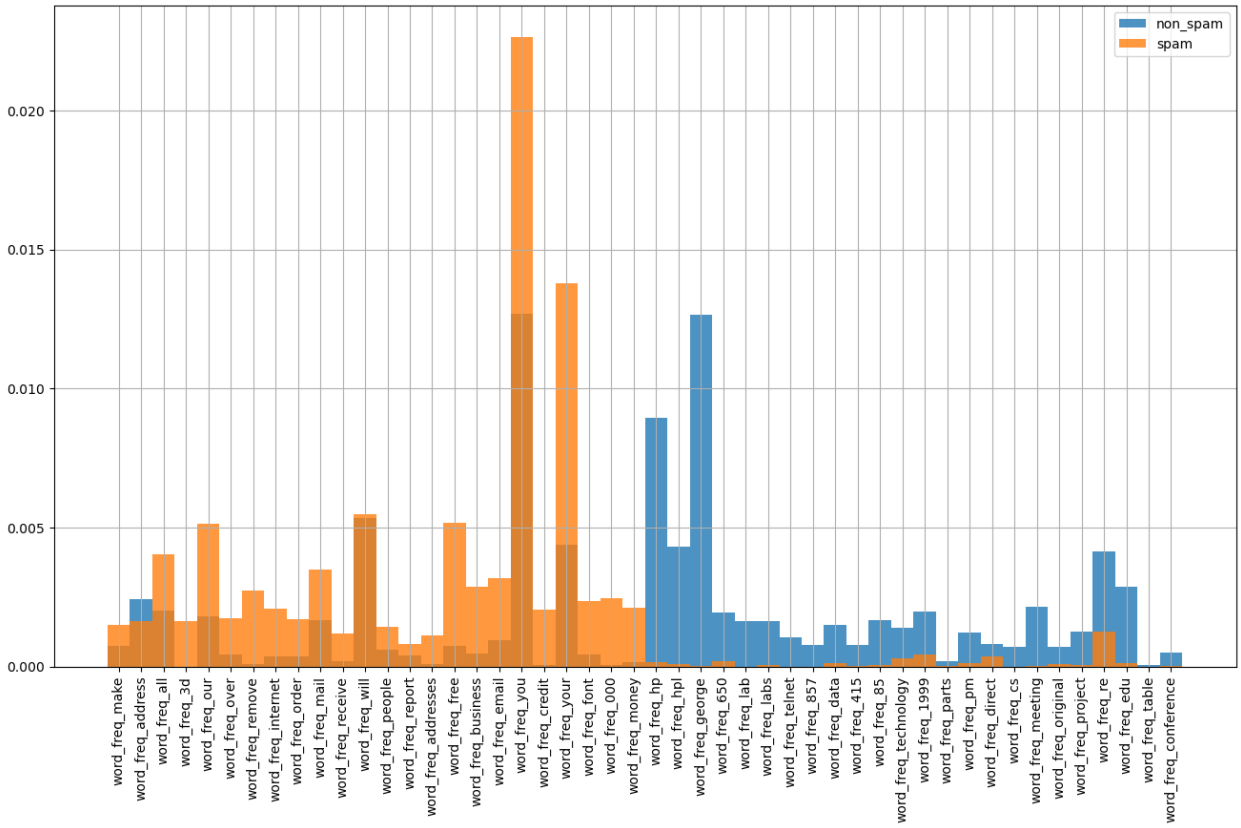
```
def plot_histogram(feature, spam, non_spam):
    plt.hist(spam[feature], bins=20, alpha=0.5, label='Spam')
    plt.hist(non_spam[feature], bins=20, alpha=0.5, label='Non-Spam')
    plt.xlabel(feature)
    plt.ylabel('Frequency')
    plt.title(f'Histogram of {feature} for Spam and Non-Spam Emails')
    plt.legend()
    plt.show()


plot_histogram('word_freq_business', spam, non_spam)
```



In the histogram plot of word “business” in spam and non-spam messages we can see most non-spam emails rarely contain the word "business". Some spam emails contain "business" more frequently than non-spam emails. However, the overall frequency remains very low across both categories. As both spam and non-spam distributions heavily overlap at low values, this feature alone may not be a strong indicator for classifying spam.

We can compute the average word frequency for spam and non-spam emails separately by using the code below.

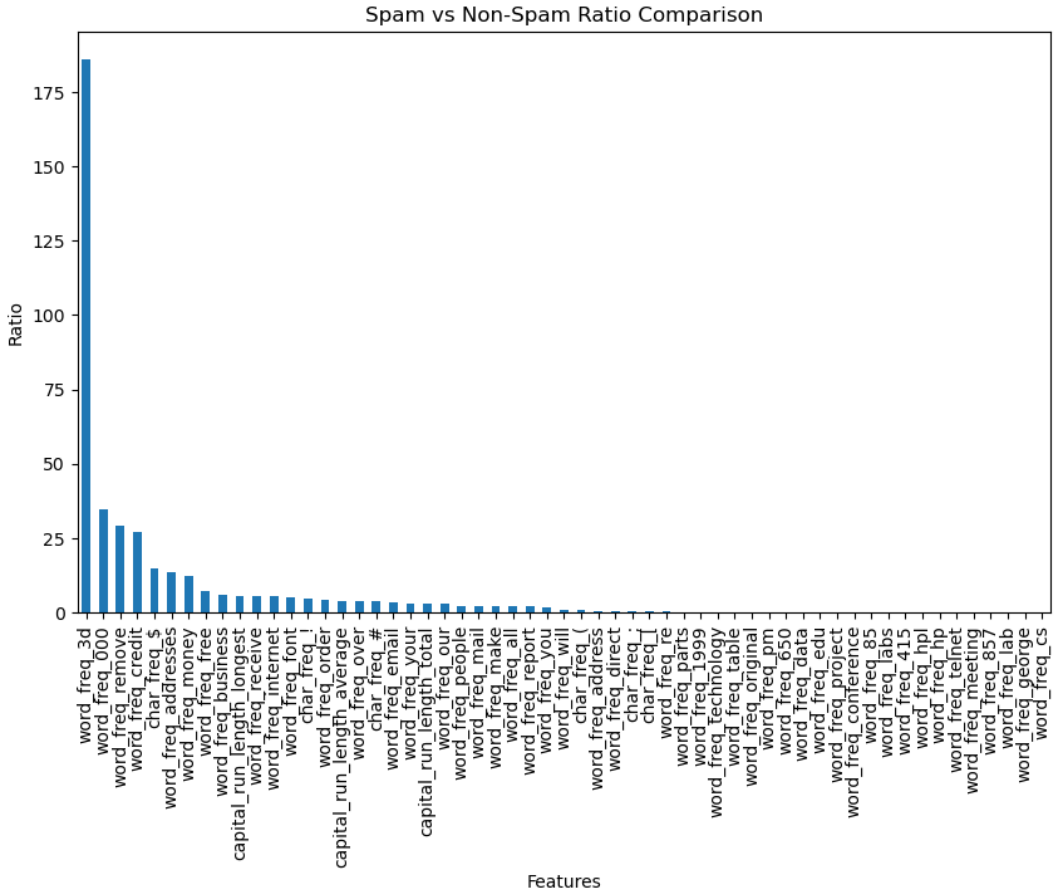
```
mean_wf = data.groupby('spam').mean()
mean_wr_fr = mean_wf.iloc[:, 0:-9]
nospam_wr_fr = mean_wr_fr.iloc[0]
spam_wr_fr = mean_wr_fr.iloc[1]
plt.figure(figsize=(16, 9))
plt.bar(nospam_wr_fr.index, nospam_wr_fr.values, width=1, alpha=0.8)
plt.bar(spam_wr_fr.index, spam_wr_fr.values, width=1, alpha=0.8)
plt.xticks(rotation='vertical')
plt.legend(['non_spam', 'spam'])
plt.grid()
plt.show()
```



We can compute the mean feature values for spam and non-spam emails, and then further compute the ratio of these means to determine which features are more prevalent in spam

emails. Then, we can order the data so that the features with the highest spam-to-non spam ratio appear at the top. These are the most indicative features of spam.

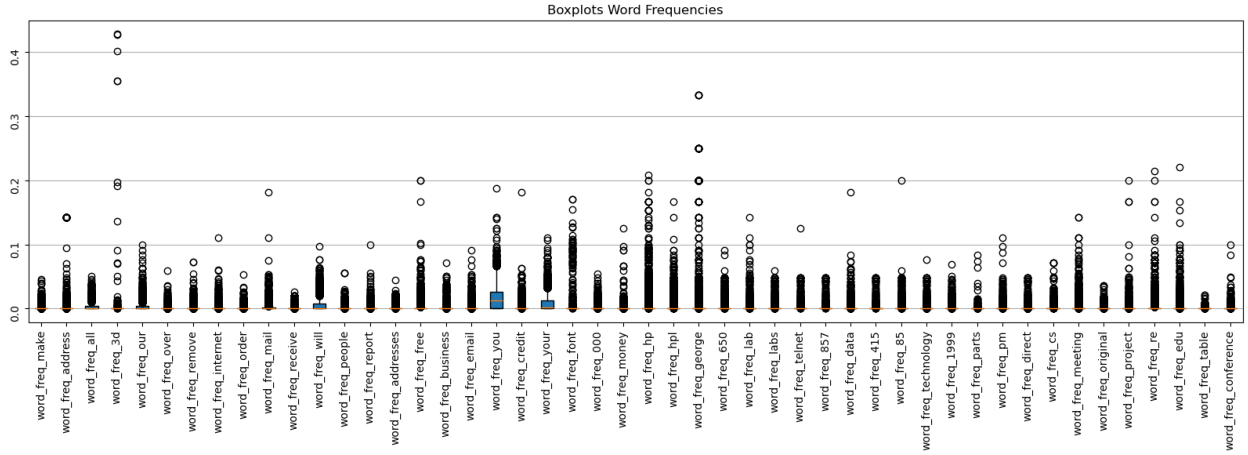
```
spam_mean = spam.mean()
non_spam_mean = non_spam.mean()
spam_diff = pd.concat(
    [spam_mean, non_spam_mean, spam_mean/non_spam_mean], axis=1)
spam_diff = spam_diff[:-1]
spam_diff.columns = ['Spam', 'Non-Spam', 'Ratio']

spam_diff.sort_values(by='Ratio', ascending=False, inplace=True)
spam_diff_mean = spam_diff['Ratio'].mean()
selected_spam_diff = spam_diff[spam_diff['Ratio'] > spam_diff_mean]
spam_diff['Ratio'].plot(kind='bar', figsize=(10, 6))
plt.xlabel('Features')
plt.ylabel('Ratio')
plt.title('Spam vs Non-Spam Ratio Comparison')
plt.show()
```



In the figure above we can see features occurring more in spam messages as compared to non-spam messages in the left side of the plot.

We can perform boxplot visualization of different features in the data using the code below. It helps in understanding data distribution, outliers, and variance among different features.

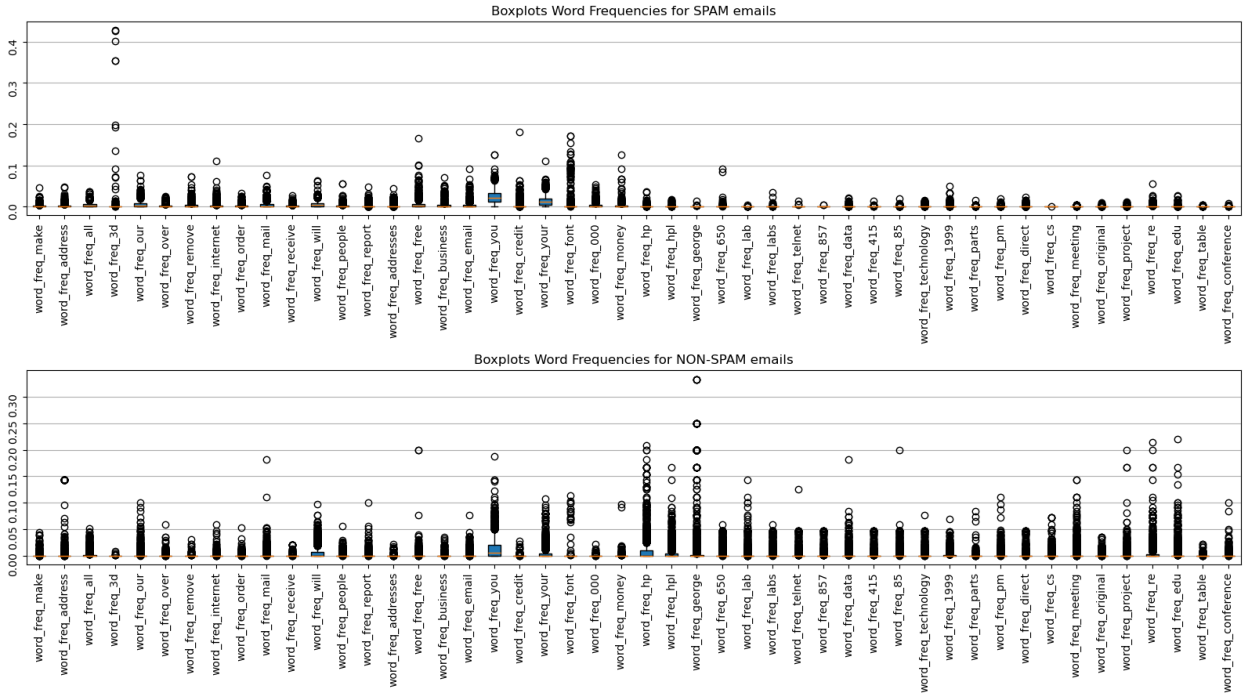
```
data_wr_fr = data.iloc[:, :-10] #Selects all features excluding the last 10 columns.
data_char_freq = data.iloc[:, -10:-4] # Extracts features from index -10 to -4.
data_capital_run = data.iloc[:, -4:-1] # Extracts features from index -4 to -1.
def draw_boxplot(ax, label, data):
    ax.boxplot(data,
               vert=True,
               patch_artist=True,
               labels=data.columns)
    ax.set_title(label)
    ax.yaxis.grid(True)
    ax.tick_params(labelrotation=90)

fig, ax = plt.subplots(figsize=(20, 5))
draw_boxplot(ax, 'Boxplots Word Frequencies', data_wr_fr)
plt.show()
```



We can compare word frequency distributions between spam and non-spam emails using boxplots by using the code below. It creates two separate boxplots, one for spam emails and another for non-spam emails, allowing us to observe differences in word frequency distributions.

```
fig, (ax1, ax2) = plt.subplots(nrows=2, ncols=1, figsize=(20, 9))
draw_boxplot(ax1, 'Boxplots Word Frequencies for SPAM emails',
data_wr_fr[data['spam']==True])
draw_boxplot(ax2, 'Boxplots Word Frequencies for NON-SPAM emails',
data_wr_fr[data['spam']==False])
fig.subplots_adjust(hspace=0.8)
plt.show()
```

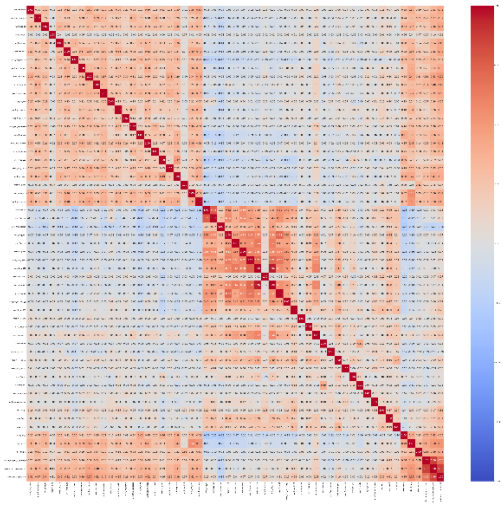
We can observe more outliers in the spam messages for the features 3d, money, credit etc. Outliers indicate words that have unusually high occurrences in some emails, making them useful for detecting spam.



2.1 Analyzing Correlations in the Spambase Dataset

We can observe the correlation between different features in the dataset using the code below.

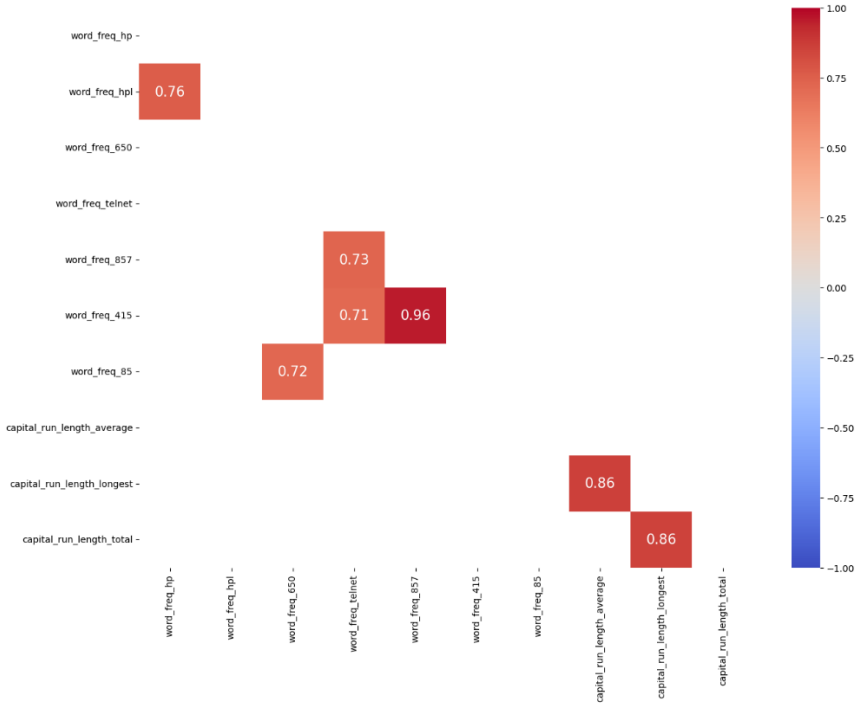
```
import seaborn as sns
new_df = data.iloc[:, :-1].copy()
plt.rcParams.update({'figure.figsize':(60,55), 'figure.dpi':100})
correlation_matrix = new_df.corr(method='spearman')
sns.heatmap(correlation_matrix, annot=True, fmt=".2f", vmin=-1, vmax=1, cbar=True,
cmap='coolwarm', annot_kws={'size': 15})
plt.show()
```



In the figure above we can see positive correlation between the features with red or orange shades, negative correlation with blue shades and white shade represents no correlation between the features.

We can identify highly correlated features (correlation > 0.7) and visualize them using a heatmap by using the below code.

```
threshold = 0.7
high_corr = correlation_matrix[abs(correlation_matrix) > threshold]
np.fill_diagonal(high_corr.values, np.nan)
mask = np.triu(np.ones_like(high_corr, dtype=bool))
inverse_mask = ~mask
high_corr_masked = high_corr * inverse_mask
high_corr_masked.dropna(how='all', axis=1, inplace=True)
high_corr_masked.dropna(how='all', axis=0, inplace=True)
mask = np.triu(np.ones_like(high_corr_masked, dtype=bool))
plt.rcParams.update({'figure.figsize':(15,11), 'figure.dpi':100})
sns.heatmap(high_corr_masked, mask=mask, annot=True, fmt=".2f", vmin=-1, vmax=1,
cbar=True, cmap='coolwarm', annot_kws={'size': 15})
plt.show()
```



We can observe the highest correlation between the features 415 and 857.

2.3 Machine Learning Models Implementation

2.3.1 Decision Trees

Decision trees are like a flowchart that helps a machine make decisions. They start with a big question (like, "Is this email spam?") and then branch out into smaller true-false questions, like "Does it have the word 'free' a lot?" or "Are there lots of exclamation marks?" Each answer leads to another question or a final decision, splitting the problem into simpler pieces until it figures out the answer.

We can write a function in python to visualize both the confusion matrix and the classification report side by side.

```
def plot_confusion_matrix_and_classification_report(y_test, y_test_predict,
title=""):
    # Confusion Matrix
    conf_matrix = metrics.confusion_matrix(y_test, y_test_predict)
    cm_display = metrics.ConfusionMatrixDisplay(confusion_matrix=conf_matrix,
display_labels=['Non-Spam', 'Spam'])

    # Classification Report
    class_report = classification_report(y_test, y_test_predict, output_dict=True)
    df_report = pd.DataFrame(class_report).transpose()

    # Plotting
    fig, ax = plt.subplots(1, 2, figsize=(16, 8))

    # Confusion Matrix
    cm_display.plot(ax=ax[0])
    ax[0].set_title('Confusion Matrix')


    # Classification Report Metrics
    df_report.iloc[:-3, :-1].plot(kind='bar', ax=ax[1])
    ax[1].set_title('Classification Report Metrics')
    ax[1].set_xticklabels(['Non-Spam', 'Spam'], rotation=0)

    fig.suptitle(title, fontsize=16)

    plt.tight_layout()
    plt.show()
```

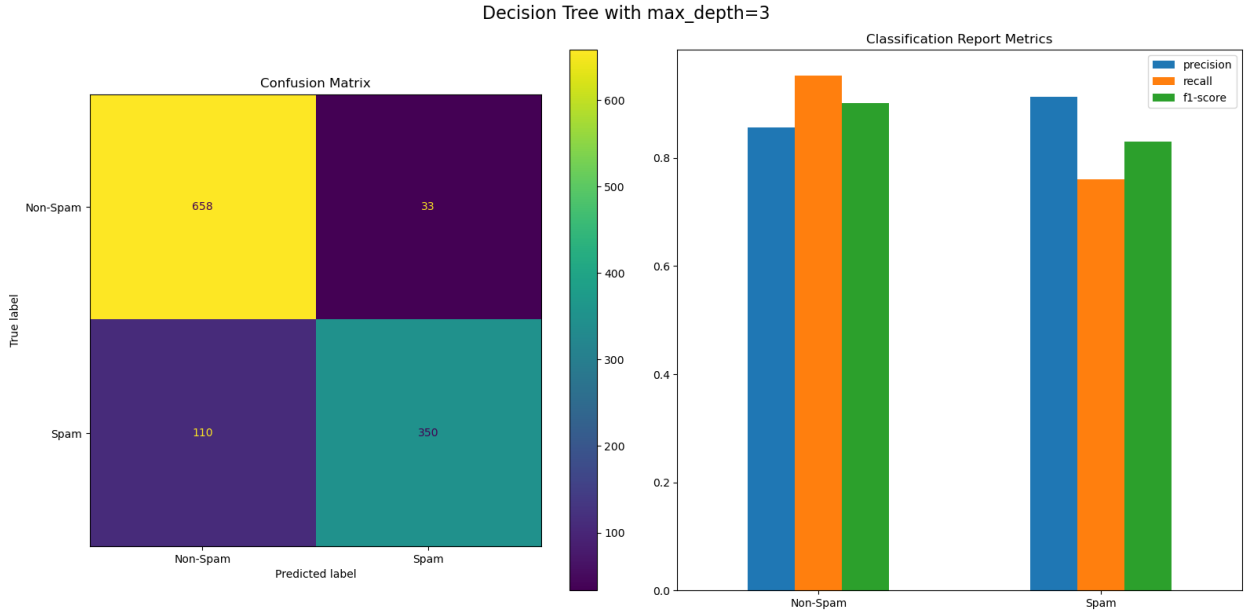
We can identify important features for spam classification using Logistic Regression by using the below code.

```
data_attributes = data.columns.tolist()[:-1]
high_corr_attributes = ['word_freq_hpl', 'word_freq_telnet', 'word_freq_857',
'word_freq_85', 'capital_run_length_longest']
data_attributes_no_corr = [attr for attr in data_attributes if attr not in
high_corr_attributes]
email_train, email_test = train_test_split(data, test_size=0.25, random_state=0)

formula = "spam ~ " + " + ".join(data_attributes_no_corr)
model = logit(formula, email_train).fit()
p_values = model.pvalues
```

```
if 'Intercept' in p_values.index:
    p_values.drop('Intercept', inplace=True)
ordered_p_values = p_values.sort_values(ascending=False).round(3)
useful_p_values = ordered_p_values[ordered_p_values < 0.05]
useful_attributes = useful_p_values.index.tolist()
```

Then we can write a code that trains a Decision Tree Classifier on selected useful attributes for spam detection.

```
data_useful_attributes = data[useful_attributes + ['spam']].copy()
X_data = data_useful_attributes.drop(columns=['spam'])  # Features
y_data = data_useful_attributes['spam']  # Target variable
X_train, X_test, y_train, y_test = train_test_split(X_data,y_data, test_size=0.25,
random_state=0)
dt = DecisionTreeClassifier(max_depth=3, random_state=0)
train_tree_preds = dt.fit(X_train,y_train)
y_test_predict = dt.predict(X_test)
plot_confusion_matrix_and_classification_report(y_test, y_test_predict,
title="Decision Tree with max_depth=3")
print("Classification Report")
print(classification_report(y_test, y_test_predict))
```




```
Classification Report
              precision    recall  f1-score   support

           0       0.86      0.95      0.90       691
           1       0.91      0.76      0.83       460

    accuracy                           0.88      1151
   macro avg       0.89      0.86      0.87      1151
weighted avg       0.88      0.88      0.87      1151
```

We can see precision of decision tree classifier is higher for spam emails than for non-spam messages. Whereas, precision for non-spam emails is higher than for spam emails.

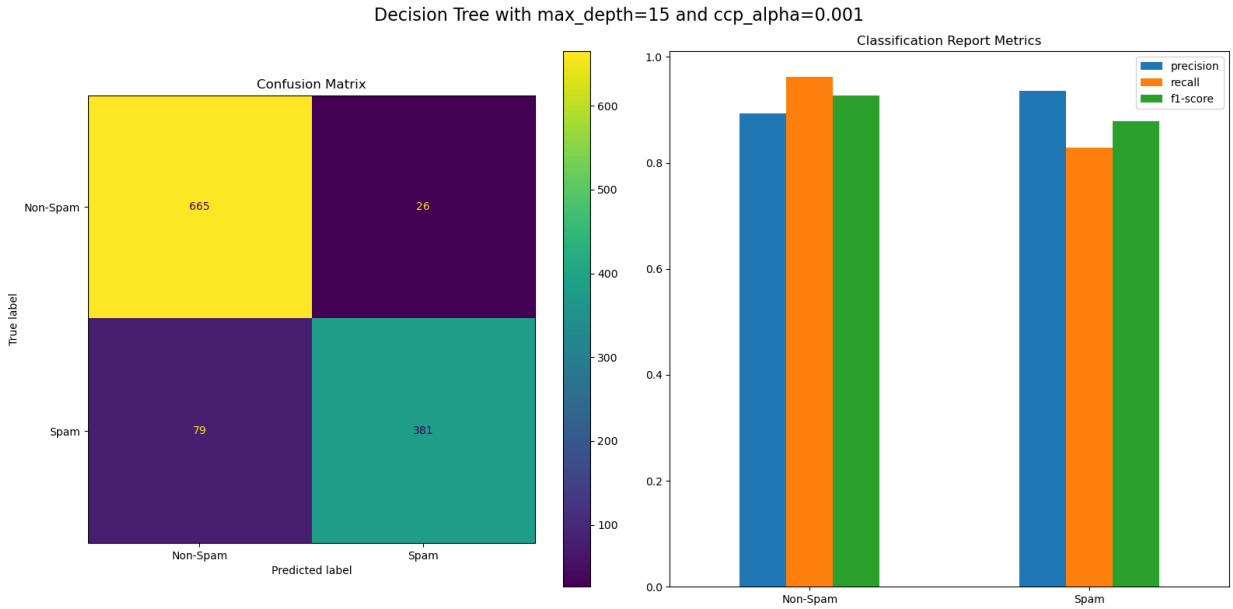
We can perform hyperparameter tuning for a Decision Tree Classifier using GridSearchCV, testing different tree depths (max_depth) and pruning strengths (ccp_alpha). The goal is to find the best combination that maximizes accuracy while avoiding overfitting.

```
param_grid = {
    'max_depth': [5, 10, 15, 20],  # Different max_depth values to test
    'ccp_alpha': [0.0, 0.001, 0.01, 0.1]  # Different pruning parameters to test
}
clf = DecisionTreeClassifier(random_state=0)
grid_search = GridSearchCV(clf, param_grid, cv=5, scoring='accuracy')
grid_search.fit(X_data, y_data)
print("Best Parameters:", grid_search.best_params_)
print("Best Accuracy Score:", grid_search.best_score_)
print("Depth of Best Tree:", grid_search.best_estimator_.get_depth())
```

```
Best Parameters: {'ccp_alpha': 0.001, 'max_depth': 15}
Best Accuracy Score: 0.8987135910871926
Depth of Best Tree: 11
```

We can then train the decision tree classifier on the best parameters and compare them with the previous model.

```
# Train the Decision Tree Classifier
dt = DecisionTreeClassifier(ccp_alpha=0.001, max_depth=15, random_state=0)
dt.fit(X_train, y_train)
y_test_predict = dt.predict(X_test)
plot_confusion_matrix_and_classification_report(y_test, y_test_predict,
title="Decision Tree with max_depth=15 and ccp_alpha=0.001")
```



```
Classification Report
              precision    recall  f1-score   support

           0       0.89      0.96      0.93       691
           1       0.94      0.83      0.88       460

    accuracy                           0.91      1151
   macro avg       0.91      0.90      0.90      1151
weighted avg       0.91      0.91      0.91      1151
```

We can observe values of all the parameters have improved than previously.

2.3.2 Random Forrest

The second algorithm we have selected to solve email spam problem is Random Forrest. Random Forest is a team of decision trees working together. It builds lots of these trees while learning from the data and picks the most popular choice among them.

For our Random Forest, we set it up to make 100 trees (that's the default number), and we limited each tree to only go three steps deep so it wouldn't get too complicated and overthink the patterns. This teamwork approach usually makes Random Forest better than a single decision tree because it smooths out mistakes and keeps things balanced.

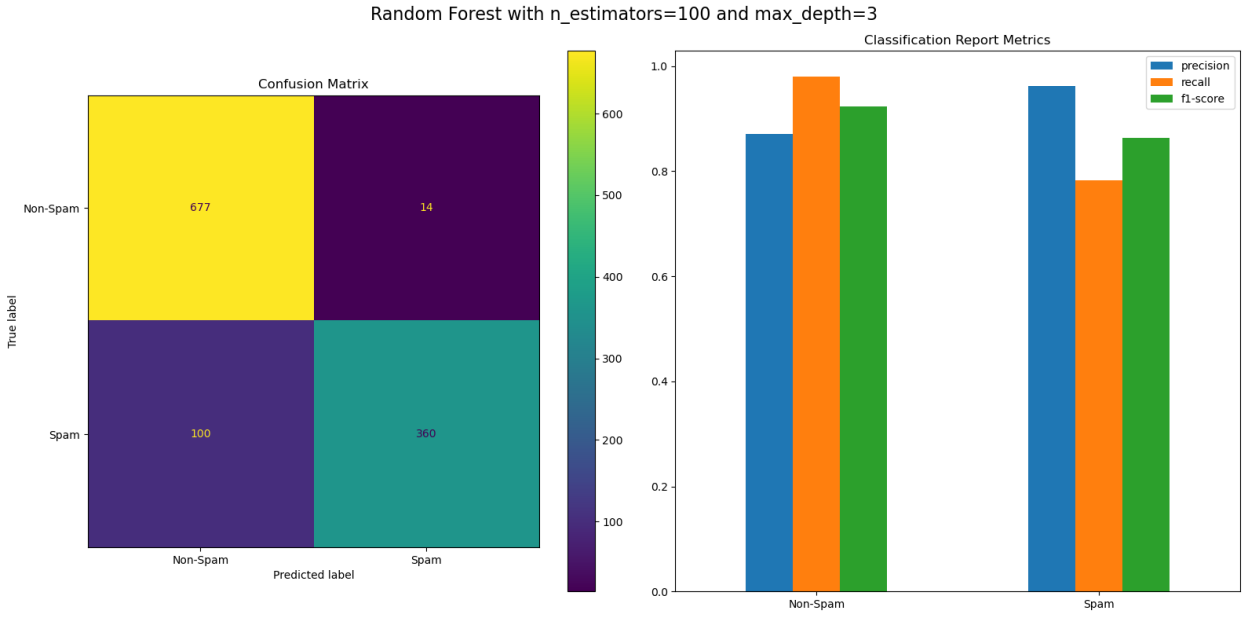
When we tested it, the Random Forest scored an average accuracy of 0.9039 (out of 1) across different checks. That's good, but it's only a tiny bit better than what we got with decision tree before. It shows that our single decision tree was already doing a good job, especially since we had tweaked it just right, like setting the perfect depth and trimming it so it wouldn't overdo things.

```
rf_model = RandomForestClassifier(n_estimators=100, max_depth=3, random_state=0)
rf_model.fit(X_train, y_train)
y_test_predict = rf_model.predict(X_test)
plot_confusion_matrix_and_classification_report(y_test, y_test_predict, title="Random
Forest with n_estimators=100 and max_depth=3")
```

The above code trains a Random Forest Classifier on a dataset by creating 100 decision trees to improve robustness. It limits each decision tree to a depth of 3 levels to reduce overfitting and ensures reproducibility. It evaluates its performance using a confusion matrix and classification report.

```
Classification Report
              precision    recall  f1-score   support

           0       0.87      0.98      0.92       691
           1       0.96      0.78      0.86       460

    accuracy                           0.90      1151
```

```
   macro avg       0.92      0.88      0.89      1151
weighted avg       0.91      0.90      0.90      1151
```



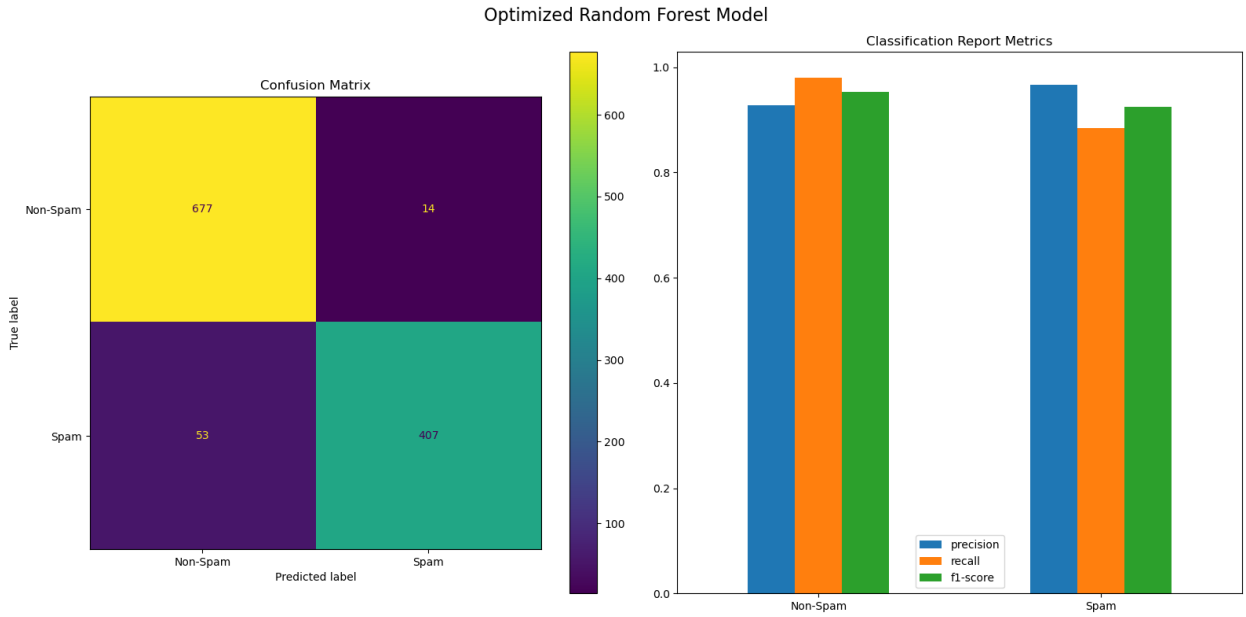
We can observe the above results to be comparable to the tweaked decision tree classifier.

We can evaluate the Random Forest classifier using cross-validation, ensuring robust and unbiased accuracy estimation.

```
param_grid = {
    'max_depth': [5, 10, 15],  # Limit tree depth
    'min_samples_split': [2, 5, 10],  # Minimum samples required to split
    'min_samples_leaf': [1, 3, 5],  # Minimum samples in each leaf
    'ccp_alpha': [0.0, 0.001, 0.01]  # Cost complexity pruning
}

# Initialize Random Forest
rf_model = RandomForestClassifier(n_estimators=100, random_state=0)

# Perform Grid Search with 5-fold cross-validation
grid_search = GridSearchCV(rf_model, param_grid, cv=5, scoring='accuracy', n_jobs=-1,
verbose=2)
grid_search.fit(X_train, y_train)

# Print Best Parameters
print("Best Parameters:", grid_search.best_params_)
print("Best Accuracy Score:", grid_search.best_score_)

Fitting 5 folds for each of 81 candidates, totalling 405 fits
Best Parameters: {'ccp_alpha': 0.0, 'max_depth': 15, 'min_samples_leaf': 1,
'min_samples_split': 2}
Best Accuracy Score: 0.9481159420289854
```

We can now train the random forrest model with the best parameters and plot the confusion matrix and different metrics.

```
best_params = grid_search.best_params_
best_rf = RandomForestClassifier(
    n_estimators=100,  # Keep the number of trees fixed
    max_depth=best_params['max_depth'],
    min_samples_split=best_params['min_samples_split'],
    min_samples_leaf=best_params['min_samples_leaf'],
    ccp_alpha=best_params['ccp_alpha'],  # Pruning parameter
    random_state=0
)
best_rf.fit(X_train, y_train)
y_test_predict = best_rf.predict(X_test)

plot_confusion_matrix_and_classification_report(y_test, y_test_predict,
title="Optimized Random Forest Model")
print("Classification Report")
print(classification_report(y_test, y_test_predict))
```




```
Classification Report
              precision    recall  f1-score   support

           0       0.93      0.98      0.95       691
           1       0.97      0.88      0.92       460

    accuracy                           0.94      1151
   macro avg       0.95      0.93      0.94      1151
weighted avg       0.94      0.94      0.94      1151
```

We can observe all the metrics have improved quite a lot compared to previous model.

2.3.3 Support Vector Machine

The third machine learning algorithm that we are going to implement is support vector machine (SVM). We are going to implement SVM using three different kernels namely linear, polynomial and radial basis function. We will select the best model and perform cross-validation with parameter tuning to see if there is any improvement in the results.

```
data_useful_attributes = data[useful_attributes + ['spam']].copy()
X_data = data_useful_attributes.drop(columns=['spam'])  # Features
y_data = data_useful_attributes['spam']  # Target variable
X_train, X_test, y_train, y_test = train_test_split(X_data,y_data, test_size=0.25,
random_state=0)
```

Now we are going to search for the best kernel type for the SVM model to achieve highest accuracy.

```
from sklearn.svm import SVC
from sklearn.model_selection import GridSearchCV
from sklearn.metrics import classification_report
# Define the parameter grid
param_grid = {'kernel': ['linear', 'poly', 'rbf']}
svm = SVC()
grid_search = GridSearchCV(svm, param_grid, cv=5, scoring='accuracy')
# Perform grid search
grid_search.fit(X_train, y_train)
print("Best Parameters:", grid_search.best_params_)
print("Best Score:", grid_search.best_score_)
best_svm = grid_search.best_estimator_
```

We can find the best kernel is linear with the 74.6% accuracy.

```
Best Parameters: {'kernel': 'linear'}
Best Score: 0.7460869565217391
```

We can optimize a SVM by tuning C and Gamma as shown in the below code.

```
from sklearn.svm import SVC
from sklearn.model_selection import GridSearchCV
from sklearn.metrics import classification_report
# Define the parameter grid
param_grid = {
    'C': [0.1, 1, 10],
    'kernel': ['linear', ‘poly’, 'rbf'],
    'gamma': [0.01, 0.1, 1, 10]
}
svm = SVC()
grid_search = GridSearchCV(svm, param_grid, cv=5, scoring='accuracy',n_jobs=-1)
# Perform grid search
grid_search.fit(X_train, y_train)
print("Best Parameters:", grid_search.best_params_)
print("Best Score:", grid_search.best_score_)
best_svm = grid_search.best_estimator_
```

The best kernel comes out to be linear with C=10 and accuracy of 82%.

```
Best Parameters: {'C': 10, 'kernel': 'linear'}
Best Score: 0.82

We can then evaluate the best model on the test set by using the code below.

# Evaluate the best model on the test set
y_pred = best_svm.predict(X_test)
print("\nClassification Report (Test Set):")
print(classification_report(y_test, y_pred))

Classification Report (Test Set):
              precision    recall  f1-score   support

           0       0.86      0.85      0.85       691
           1       0.77      0.78      0.78       460

    accuracy                           0.82      1151
   macro avg       0.81      0.81      0.81      1151
weighted avg       0.82      0.82      0.82      1151

We can further improve the accuracy by normalizing the data.
scaler = StandardScaler()
# Fit on training set only
scaler.fit(X_train)
# Apply transform to both the training set and the test set
X_train_scaled = scaler.transform(X_train)
X_test_scaled = scaler.transform(X_test)
# Train the SVM Classifier on the scaled data
svm_model = SVC(kernel='linear')  # You can change the kernel as needed
svm_model.fit(X_train_scaled, y_train)
# Predict on the scaled test data
y_pred = svm_model.predict(X_test_scaled)
print("\nClassification Report:")
print(classification_report(y_test, y_pred))
```

The accuracy is improved to 91% by normalizing the data.

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

           0       0.92      0.94      0.93       691
           1       0.91      0.87      0.89       460

    accuracy                           0.91      1151
   macro avg       0.91      0.91      0.91      1151
weighted avg       0.91      0.91      0.91      1151
```

3. Critical Reflection

3.1 Challenges Faced

3.1.1 Decision Trees

Decision trees try to track every small piece of information in the data, which can make them grow too big and detailed. For example, in an email it will look for words and characters instead of finding general patterns, which results in overfitting, and it makes them bad at labelling spam in new emails they haven't seen before.

Decision trees split data based on one question at a time (e.g., "Does it have 'free' in it?"), which can miss how things work together, like how "free" and "!!!" combined might result in better classifier than alone.

If the data's noisy (some emails are mislabeled or have typos), a decision tree can lead to wrong guesses.

3.1.2 SVM (Support Vector Machines)

SVMs draw a boundary to separate spam from non-spam. The decision of selecting boundary results influences the accuracy of the model.

SVMs work with patterns in the data, but if you've got lots of emails or lots of features (like the 57 in Spambase), SVMs get very slow and take forever to train.

If spam and non-spam emails look really similar (both use "money" a lot), SVMs can have a hard time drawing a clean line between them, especially if the data isn't perfectly clean.

3.1.3 Random Forests

Random Forests use lots of trees for balancing, if we don't limit how deep each tree goes or how many features they look at, they can overfit the training data, which will result in low accuracy on new data.

Building 100 trees (or more) takes more computing power and time than a single decision tree or even an SVM. For a small dataset like Spambase it's fine, but with bigger, real-world email flows, it can slow things down.

Unlike a single decision tree where you can follow the flowchart, Random Forests are like a crowd of opinions. That makes it difficult to figure out why it flagged an email as spam, which can be a problem if you need to tweak the parameters of the system.

3.1.4 Shared Challenges Across All Three

Spammers keep changing buzzwords and formatting of their messages. Therefore, the models need to be updated by getting trained on the new spam emails to catch the new spam emails.

With 57 features in Spambase, it's difficult to know which feature really does matter. Too many unimportant features can confuse all three methods, while missing key ones can weaken results.

If there's more non-spam messages than spam messages (imbalanced data) in the data, these models might just guess the majority class, missing the minority one.

3.2 Learnings and Improvements

- Developed better programming skills for data analysis and model implementation.
- Gained deeper insights into the practical challenges of AI-driven decision-making.
- Improved ability to critically evaluate models and optimize their performance.

4. Conclusion

This report presents the results of different machine learning algorithms to detect spam emails. The tasks undertaken have strengthened my technical expertise in data analysis, machine learning, and AI-based decision-making. This knowledge will be beneficial for future endeavors in the field of AI and data science.