CLASSIFYING SPAM VS NOT SPAM EMAILS

IBM Machine Learning: Classification

JUNE 13, 2022

RATEEB YEHYA
Coursera

Table of Contents

1. About the Data Set

The dataset is a *csv* file containing information of 5712 randomly picked email files and their respective labels for spam or not spam.

The file contains 5172 rows and 3002 columns. Each row represents an email observation. The first column indicates the observation number, and the last column indicates the labels for predictions: 1 for spam and 0 for not spam.

The dataset is extracted from *Kaggle*. *https://www.kaggle.com/datasets/balaka18/email-spam-classification-dataset-csv*

2. Objective

The objective of this report is to predict the target class label (Spam versus Not Spam) from the available features, which are the 3000 word counts available in the dataset. Our objective here is more prediction oriented than interpretation oriented because we do not really care about which word/words appear the most in spam emails. What we really care about is to correctly predict the email as a spam.

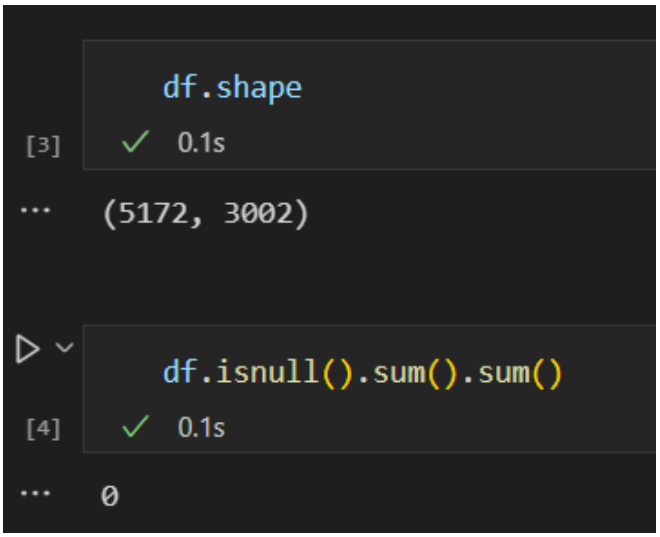
3. Exploratory Data Analysis and Feature Engineering

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score, classification_report, precision_recall_fscore_support, confusion_matrix
from sklearn.metrics import precision_score, recall_score, f1_score, roc_auc_score
```

[1] ✓ 2.4s

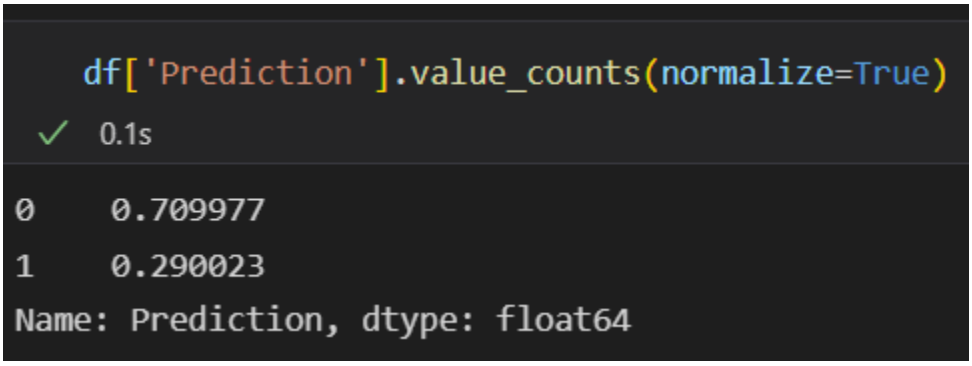
```
df = pd.read_csv('D:/Uni/Masters/Python/IBMMachineLearning/Course 3/FinalProj/emails.csv')
print(df.head())
```

[2] ✓ 1.6s



First we import all the necessary libraries in Python. As can be seen, the data contains 5172 rows and 3002 columns. Moreover, the dataset has no null values.

Before running any machine learning algorithm, it is important to check the distribution of the target class in order to identify whether the dataset is balanced or unbalanced. We can see from the figure below that the classes are not balanced, where about 71% of the target class is labeled 0 (not spam) and the rest is labeled spam.



As a result, it might be useful to use class weighting, over-sampling, under-sampling, or random sampling to improve the predictions of the model.

4. Train-Test-Split

For the training and test data, we will use a random state of 42 and a test size of 30%. Moreover, we will include the argument $stratify = y$ to ensure that the training and test sets have the same class distribution as the original dataset.

```
rs = 42
feature_cols = df.columns.drop('Prediction')
X = df[feature_cols]
y = df['Prediction']
X_train, X_test, y_train, y_test = train_test_split(X, y , random_state = rs, test_size = 0.3, stratify = y)
```

[10] ✓ 0.4s

+ Code + Markdown

```
y_train.value_counts(normalize=True)
```

[11] ✓ 0.9s

```
0    0.709945
1    0.290055
Name: Prediction, dtype: float64
```

```
y_test.value_counts(normalize = True)
```

[12] ✓ 0.1s

```
0    0.710052
1    0.289948
Name: Prediction, dtype: float64
```

We can see here that the resulting train and test sets have the same class distribution as the original dataset, with around 71% having a label 0.

5. Model 1: Logistic Linear Regression

In our first model, we will use a Logistic Linear Regression to model the data. We will evaluate the model using different score metrics such as recall, precision, f1, and roc_auc_score. Moreover, we will plot the confusion matrix of the model.

```
LR = LogisticRegression(random_state = rs, max_iter = 1000)
LR.fit(X_train, y_train)
y_pred = LR.predict(X_test)

print('For Normal Linear Regression: ')
print('--Accuracy is: ', accuracy_score(y_test, y_pred)) #0.971
print('--Recall is: ', recall_score(y_test, y_pred)) #0.951
print('--Precision is: ', precision_score(y_test, y_pred)) #0.949
print('--F1 score is: ', f1_score(y_test, y_pred)) #0.95
print('--AUC score is: ', roc_auc_score(y_test, y_pred)) #0.965
```

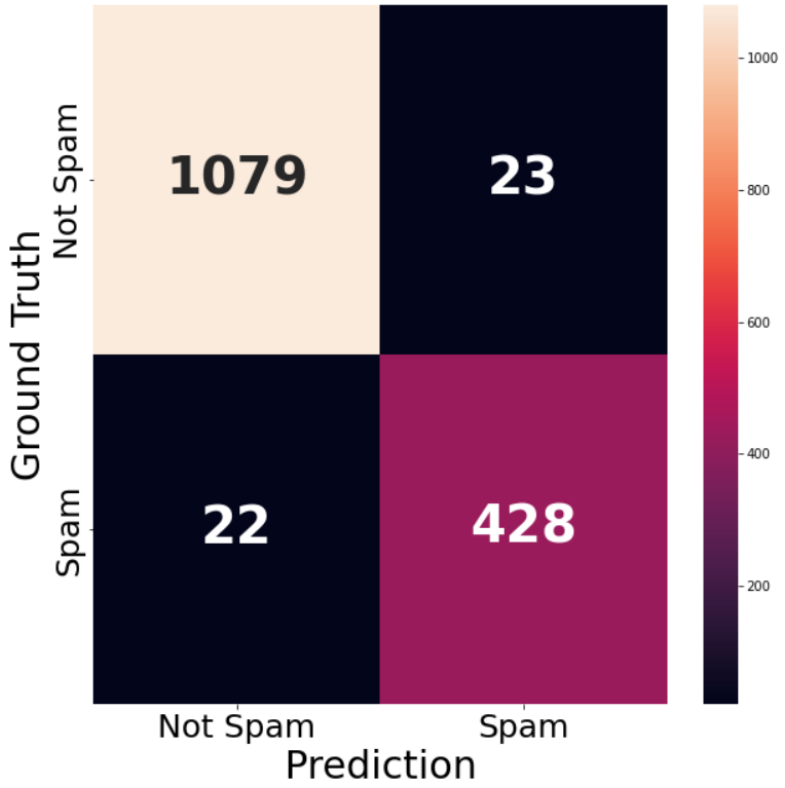
✓ 20.5s

```
For Normal Linear Regression:
--Accuracy is:  0.9710051546391752
--Recall is:  0.9511111111111111
--Precision is:  0.9490022172949002
--F1 score is:  0.9500554938956715
--AUC score is:  0.9651199838677151
```

We can see here that the Logistic Regression Model does an extremely good job in classifying the data. With a recall and precision of 0.95, the model is satisfactory even without having to use random sampling to account for the imbalanced state of the minority class.

```
import seaborn as sns
cm = confusion_matrix(y_test, y_pred)
fig, ax = plt.subplots(figsize=(10,10))
ax = sns.heatmap(cm, annot = True, fmt = 'd', annot_kws={'size':40, 'weight':'bold'})

labels = ['Not Spam', 'Spam']
ax.set_xticklabels(labels, fontsize=25)
ax.set_yticklabels(labels, fontsize = 25)
ax.set_ylabel('Ground Truth', fontsize = 30)
ax.set_xlabel('Prediction', fontsize = 30)
plt.show()
```



The confusion matrix above shows that we incorrectly predicting Spam 23 times out of 1102 Not Spams, as well as incorrectly predicting Not Spam 22 out of 450 Spams.

6. Model 2: K Nearest Neighbors

Our second model is K Nearest Neighbors. In this model, we will tune in different values of k and select the one that maximizes the $F1$ score. Before running the model, we shall scale all the features using either $MinMaxScaler$ or $StandardScaler$. The reason why scaling is important here is because K-Nearest Neighbors operates on distance between data points. Therefore, without scaling, we run into the problems of having unscaled distances that would result in model inefficiencies. For scaling, we will use the $MinMaxScaler$.

Note how we use the fit_transform method on the training dataset and then the transform method on the test set.

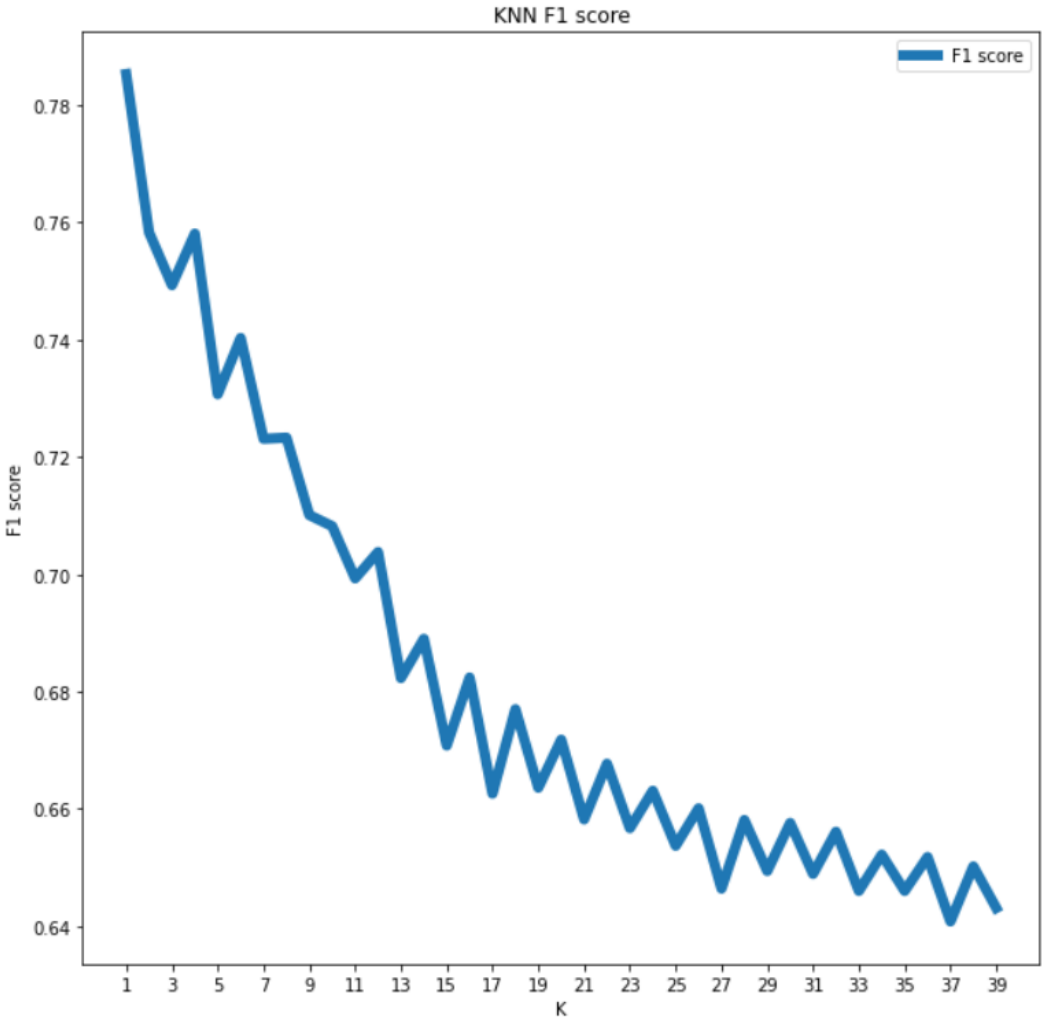
Now we iterate on the value of k, ranging from 1 to 40.

```
max_k = 40
f1_scores = list()
error_rates = list()
for k in range(1,max_k):
    knn = KNeighborsClassifier(n_neighbors=k)
    knn = knn.fit(X_train_minmax, y_train)
    y_pred = knn.predict(X_test_minmax)
    f1 = f1_score(y_test, y_pred)
    f1_scores.append(((k,round(f1,4))))
    accuracy = accuracy_score(y_test,y_pred)
    error_rates.append((k, 1-accuracy))
f1_results = pd.DataFrame(f1_scores, columns = ['K', 'F1 score'])
error_results = pd.DataFrame(error_rates, columns = ['K', 'Error Rates'])
```

We also plot the variation of $F1\_score$ and $Error\_rate$ versus K.

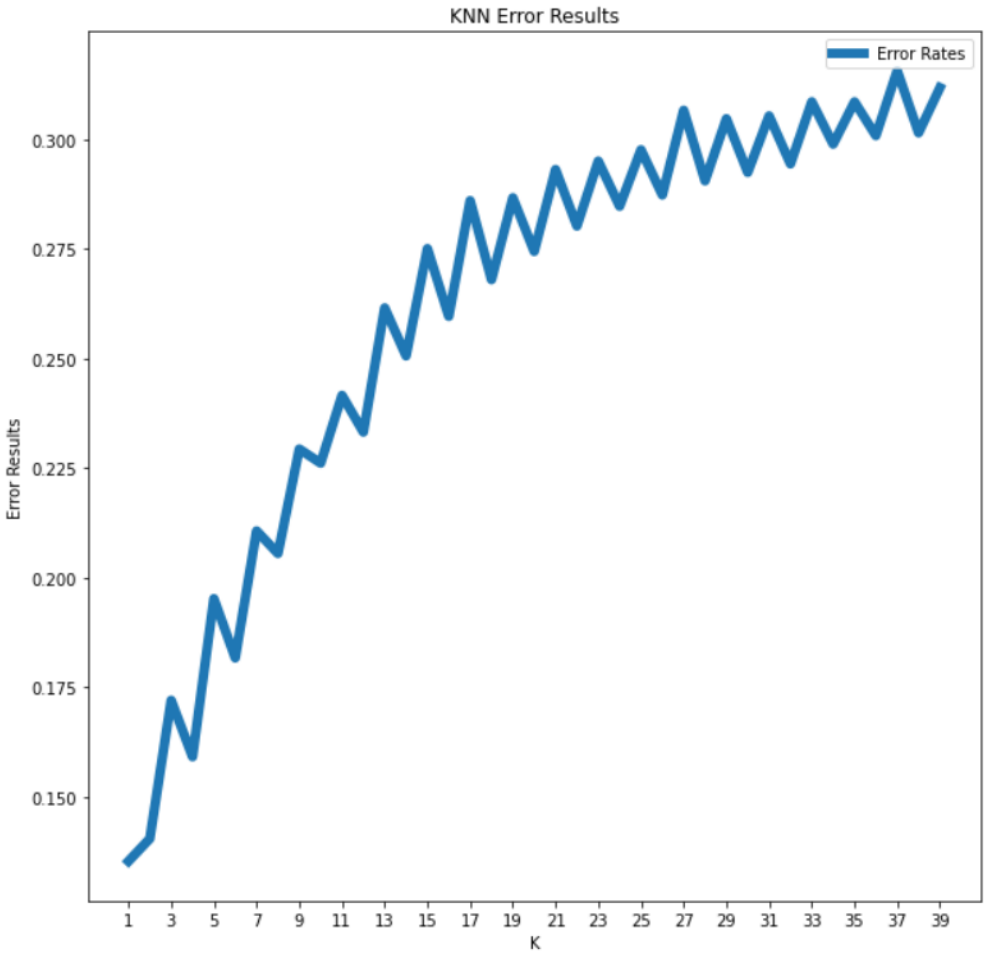
```
plt.figure(dpi = 300)
ax = f1_results.set_index('K').plot(figsize = (10,10), linewidth=6)
ax.set(xlabel = 'K', ylabel='F1 score')
ax.set_xticks(range(1,max_k,2))
plt.title('KNN F1 score')
plt.show()
```

✓ 0.3s



```
plt.figure(dpi=300)
ax = error_results.set_index('K').plot(figsize=(10,10), linewidth=6)
ax.set(xlabel = 'K', ylabel='Error Results')
ax.set_xticks(range(1,max_k,2))
plt.title('KNN Error Results')
plt.show()
```

✓ 0.4s



We can see from the graphs that the optimal value of K is strangely equal to 1. So we will use $K = 1$ to fit our model and calculate the corresponding error metrics and confusion matrix as shown below.

```
knn_best = KNeighborsClassifier(n_neighbors=1)
knn_best.fit(X_train_minmax, y_train)
y_pred = knn_best.predict(X_test_minmax)
print('For KNN Classification: ')
print('--Accuracy is: ', accuracy_score(y_test, y_pred))
print('--Recall is: ', recall_score(y_test, y_pred))
print('--Precision is: ', precision_score(y_test, y_pred))
print('--F1 score is: ', f1_score(y_test, y_pred))
print('--AUC score is: ', roc_auc_score(y_test, y_pred))
```

[93] ✓ 1.1s

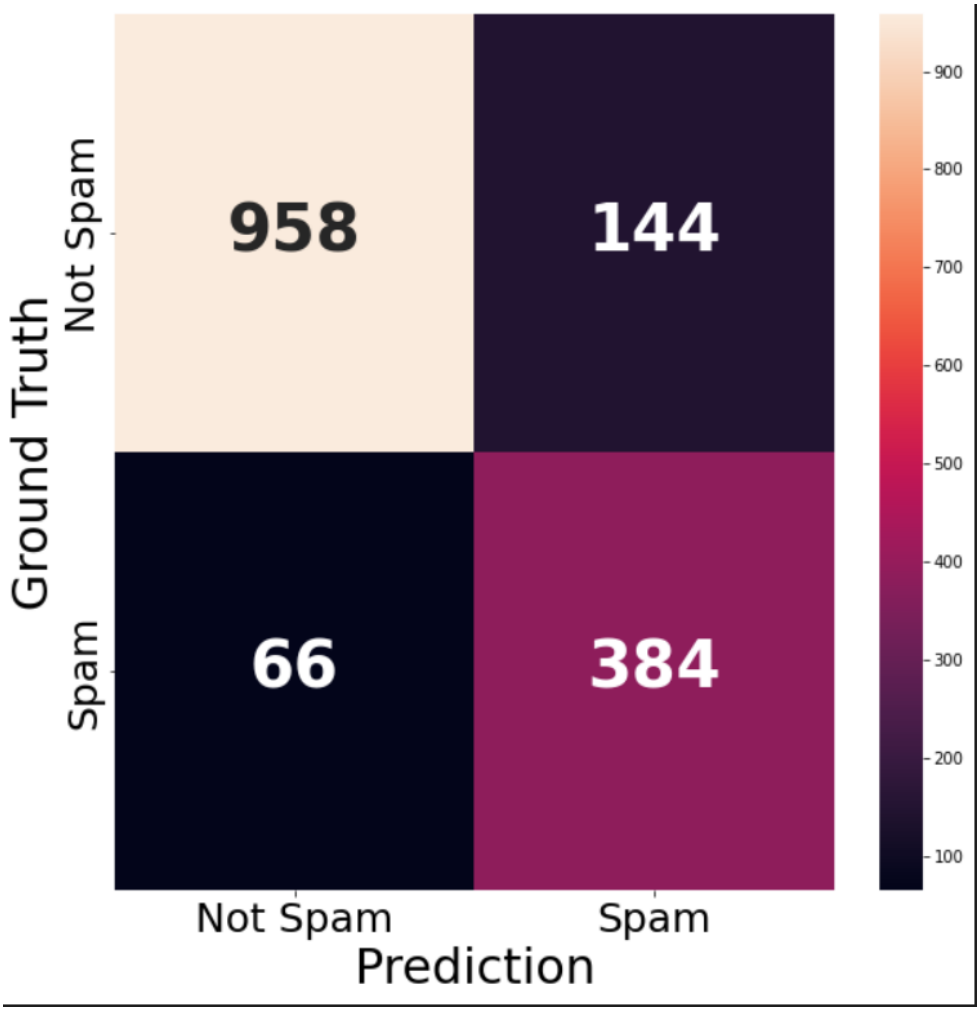
```
For KNN Classification:
--Accuracy is:  0.8646907216494846
--Recall is:  0.8533333333333334
--Precision is:  0.7272727272727273
--F1 score is:  0.785276073619632
--AUC score is:  0.8613309134906231
```

We can see here that although the *K Nearest Neighbor* model does a good job in regards to recall and precision, the Logistic Regression model does a much better job.

As for the confusion matrix of the *K Nearest Neighbor* model, it is represented below.

```
cm = confusion_matrix(y_test, y_pred)
fig, ax = plt.subplots(figsize=(10,10))
ax = sns.heatmap(cm, annot = True, fmt = 'd', annot_kws={'size':40, 'weight':'bold'})

labels = ['Not Spam', 'Spam']
ax.set_xticklabels(labels, fontsize=25)
ax.set_yticklabels(labels, fontsize = 25)
ax.set_ylabel('Ground Truth', fontsize = 30)
ax.set_xlabel('Prediction', fontsize = 30)
plt.show()
```



We can see that the confusion matrix of the Logistic Regression model produces more accurate results.

7. Model 3: Decision Tree Classifier

In our third model, we use a Decision Tree Classifier with a maximum depth equal 10 and maximum features equal to 60. Note that we have pre-set the maximum depth and maximum features for computational efficiency. For optimal values, we can use Grid Search to tune in these hyper parameters. However, it would take a long time to do so.

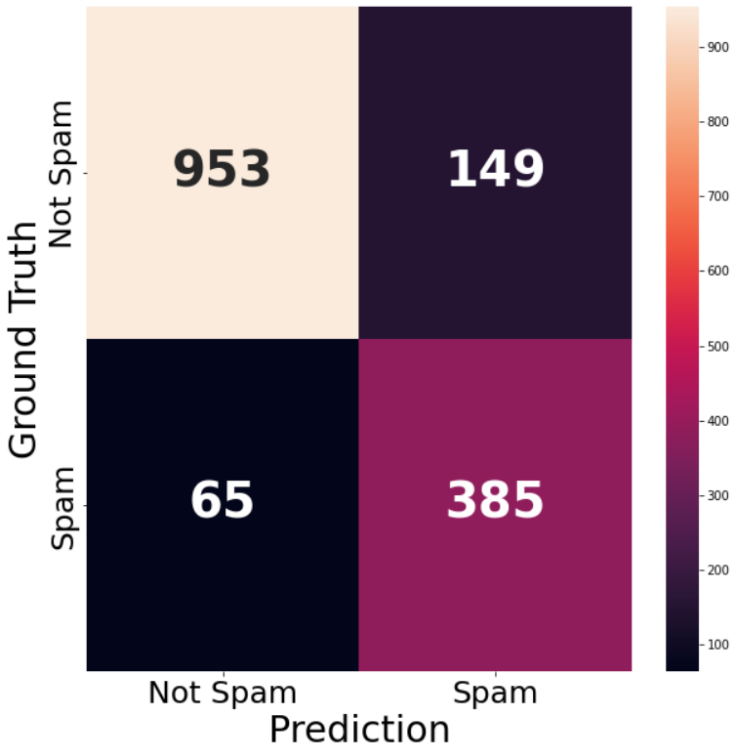
We calculate the error metrics and the confusion matrix as shown below.

```
    dt = DecisionTreeClassifier(random_state=rs, max_depth = 10, max_features=60)
    dt = dt.fit(X_train, y_train)
    y_pred = dt.predict(X_test)
    print('For Decision Tree Classification: ')
    print('--Accuracy is: ', accuracy_score(y_test, y_pred))
    print('--Recall is: ', recall_score(y_test, y_pred))
    print('--Precision is: ', precision_score(y_test, y_pred))
    print('--F1 score is: ', f1_score(y_test, y_pred))
    print('--AUC score is: ', roc_auc_score(y_test, y_pred))
133]  ✓ 0.2s

··· For Decision Tree Classification:
    --Accuracy is:  0.8621134020618557
    --Recall is:  0.8555555555555555
    --Precision is:  0.7209737827715356
    --F1 score is:  0.782520325203252
    --AUC score is:  0.8601734220608994
```

The model does a good job in predicting the classes. However, Logistic Regression still has the best scores.

The confusion matrix of the Decision Tree Model is shown below:



8. Summary and Key Insights

It is obvious that the Logistic Regression Model does best in predicting our classes, with an $F1_{score} = 0.95$. Thus, we will use it for our predictions. Moreover, although our objective is having the right predictions rather than trying to explain interpretability, Logistic Regression is a self-interpretable model which we can use to find the coefficients of those words that affect the most what our prediction is going to be. Therefore, we are able to identify the words that detect spams the most.

To expand on this further, we can use $LR.coef\_$ to print out the coefficients of the words that add the highest weight on predicting the outcome as spam or not spam.

```
pd.DataFrame({'features':LR.feature_names_in_.ravel(),
              'coeffs': LR.coef_.ravel()}).set_index('features').sort_values(by = 'coeffs')
```

90] ✓ 0.1s

features	coeffs
enron	-1.656393
hpl	-1.192369
hp	-1.019510
deal	-0.911428
attached	-0.797488
...	...
gra	0.703588
http	0.861817
one	0.906714
z	0.921194
mo	1.051936

3001 rows × 1 columns

We can see that the words that have the highest absolute correlation with a spam email are ‘enron’, ‘hpl’, ‘hp’, and ‘mo’.

9. Future Suggestions

To further optimize the Logistic Regression model, we can add a penalty as one of its arguments so that we can help regularizing it. In that way, we may obtain better score. Moreover, we can try and use a Random Forest Classifier along with SMOAT or class re-weighting. However, our Logistic Regression Model seems satisfactory in correctly labeling the emails as spam, with an overall error of around 3%