CM2604 Machine Learning Coursework Report

Module Leader – Mr. Prasan Yapa

Name : Lisara Gajaweera

RGU ID : 2118813

IIT ID : 20211029

Table of Contents

Tables

Figures

Introduction

The data set is attained by the UCL Machine Learning repository[1] . The classification of email as spam and non-spam is carried out using the machine learning models based on K Nearest Neighbors and Decision Trees.

Corpus Preparation

The UCL Spambase data set is already prepared and publicly available to use in machine learning experiments. The data and metadata were available separately. So labelling data headers was done.There are 4601 instances in the data set. And 57 attributes are considered in the data set.

Remove null/missing values.

There are no null or missing values.

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
word_freq_hpl 0
word_freq_george 0
word_freq_650 0
word_freq_lab 0
word_freq_labs 0
word_freq_telnet 0
word_freq_meeting 0
word_freq_original 0
word_freq_project 0
word_freq_re 0
word_freq_edu 0
word_freq_table 0
word_freq_conference 0

word_freq_technology 0
word_freq_1999 0
word_freq_parts 0
word_freq_857 0
word_freq_data 0
word_freq_415 0
word_freq_85 0
word_freq_pm 0
word_freq_direct 0
word_freq_cs 0
char_freq_; 0
char_freq_(0
char_freq_[0
char_freq_! 0
char_freq_$ 0
char_freq_# 0
capital_run_length_average 0
capital_run_length_longest 0
capital_run_length_total 0
spam 0

Remove Duplicates

The duplicated data removal is done using drop_duplicates() function. When removing duplicates, 391 records were removed.

```
data.shape # before dropping duplicates

(4601, 58)

data =data.drop_duplicates() #drop duplicates

print(data.shape)

(4210, 58)
```

Figure 1 Removing Duplicate data

Check data types.

From inf() function the information about the data is acquired.

```
 0   word_freq_make              4210 non-null   float64
 1   word_freq_address           4210 non-null   float64
 2   word_freq_all               4210 non-null   float64
 3   word_freq_3d                4210 non-null   float64
 4   word_freq_our               4210 non-null   float64
 5   word_freq_over              4210 non-null   float64
 6   word_freq_remove            4210 non-null   float64
 7   word_freq_internet          4210 non-null   float64
 8   word_freq_order             4210 non-null   float64
 9   word_freq_mail              4210 non-null   float64
 10  word_freq_receive           4210 non-null   float64
 11  word_freq_will              4210 non-null   float64
 12  word_freq_people            4210 non-null   float64
 13  word_freq_report            4210 non-null   float64
 14  word_freq_addresses         4210 non-null   float64
 15  word_freq_free              4210 non-null   float64
 16  word_freq_business          4210 non-null   float64
 17  word_freq_email             4210 non-null   float64
 18  word_freq_you               4210 non-null   float64
 19  word_freq_credit            4210 non-null   float64
 20  word_freq_your              4210 non-null   float64
 21  word_freq_font              4210 non-null   float64
 22  word_freq_000               4210 non-null   float64
 23  word_freq_money             4210 non-null   float64
 24  word_freq_hp                4210 non-null   float64
 25  word_freq_hpl               4210 non-null   float64
 26  word_freq_george            4210 non-null   float64
 27  word_freq_650               4210 non-null   float64
 28  word_freq_lab               4210 non-null   float64
 29  word_freq_labs              4210 non-null   float64
 30  word_freq_telnet            4210 non-null   float64
 31  word_freq_857               4210 non-null   float64
 32  word_freq_data              4210 non-null   float64
```

Figure 3 Data information 1

```
 33  word_freq_415               4210 non-null   float64
 34  word_freq_85                4210 non-null   float64
 35  word_freq_technology        4210 non-null   float64
 36  word_freq_1999              4210 non-null   float64
 37  word_freq_parts             4210 non-null   float64
 38  word_freq_pm                4210 non-null   float64
 39  word_freq_direct            4210 non-null   float64
 40  word_freq_cs                4210 non-null   float64
 41  word_freq_meeting           4210 non-null   float64
 42  word_freq_original          4210 non-null   float64
 43  word_freq_project           4210 non-null   float64
 44  word_freq_re                4210 non-null   float64
 45  word_freq_edu               4210 non-null   float64
 46  word_freq_table             4210 non-null   float64
 47  word_freq_conference        4210 non-null   float64
 48  char_freq_;                 4210 non-null   float64
 49  char_freq_(                 4210 non-null   float64
 50  char_freq_[                 4210 non-null   float64
 51  char_freq_!                 4210 non-null   float64
 52  char_freq_$                 4210 non-null   float64
 53  char_freq_#                 4210 non-null   float64
 54  capital_run_length_average  4210 non-null   float64
 55  capital_run_length_longest  4210 non-null   int64
 56  capital_run_length_total    4210 non-null   int64
 57  spam                        4210 non-null   int64
dtypes: float64(55), int64(3)
```

Table 3 data infromation contd

The cleaned data are then saved in a new file to use.

```
#Save the cleaned processed data set into a new csv file
from pathlib import Path
filepath = Path("newSpamData.csv")
filepath.parent.mkdir(parents=True,exist_ok=True)
data.to_csv(filepath)
```

Solution methodology

Scaling – Standardization

Since there are 57 features with different scales, the differences can increase the difficulty in modeling. Since here we are going to use KNN algorithm which measures distance between data points, the model can be affected because of these differences in the scales. When we are using the Decision Trees algorithm it is not affected.

In standardization the data set values are rescaled as the mean of the data is 0, and standard deviation is 1. When standardizing data the assumption that the data fit a Gaussian distribution is made.

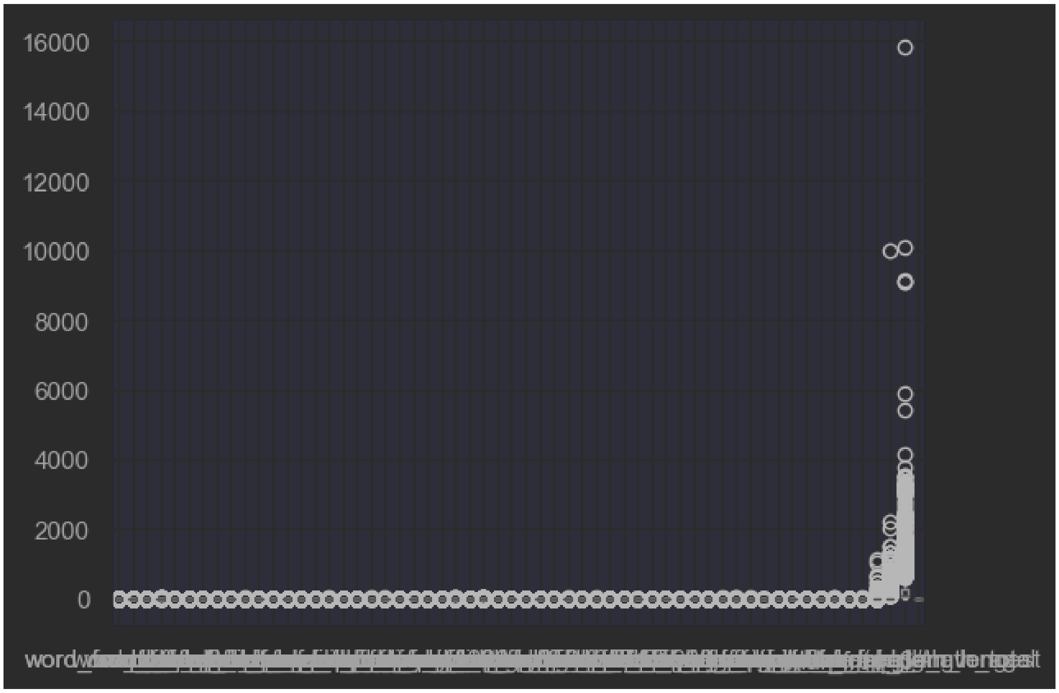

Figure 2 Boxplot of data before standardization

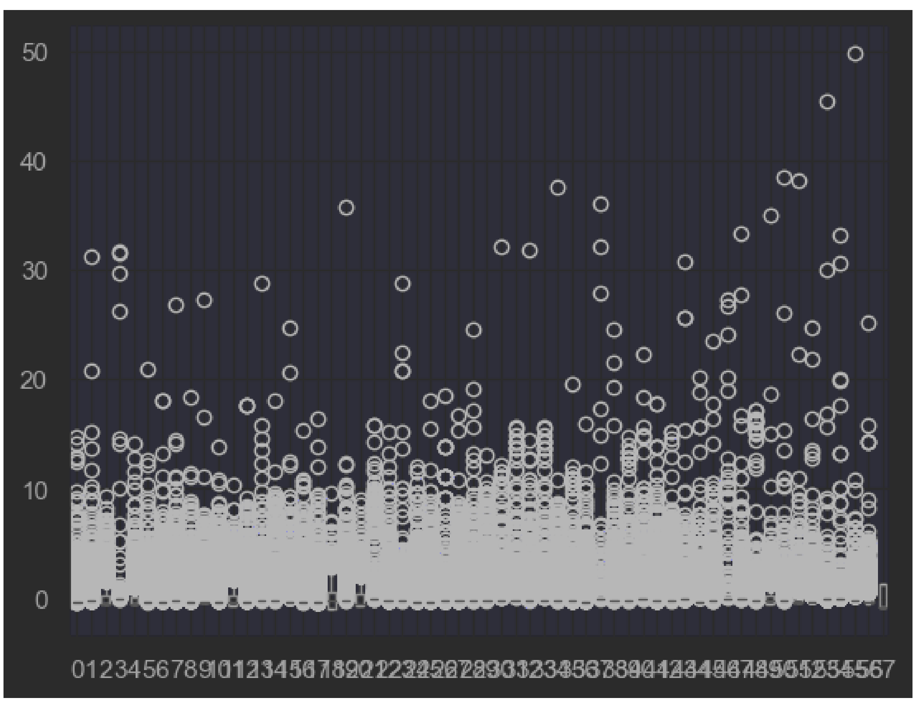

Figure 3 Box plot of data after standardization

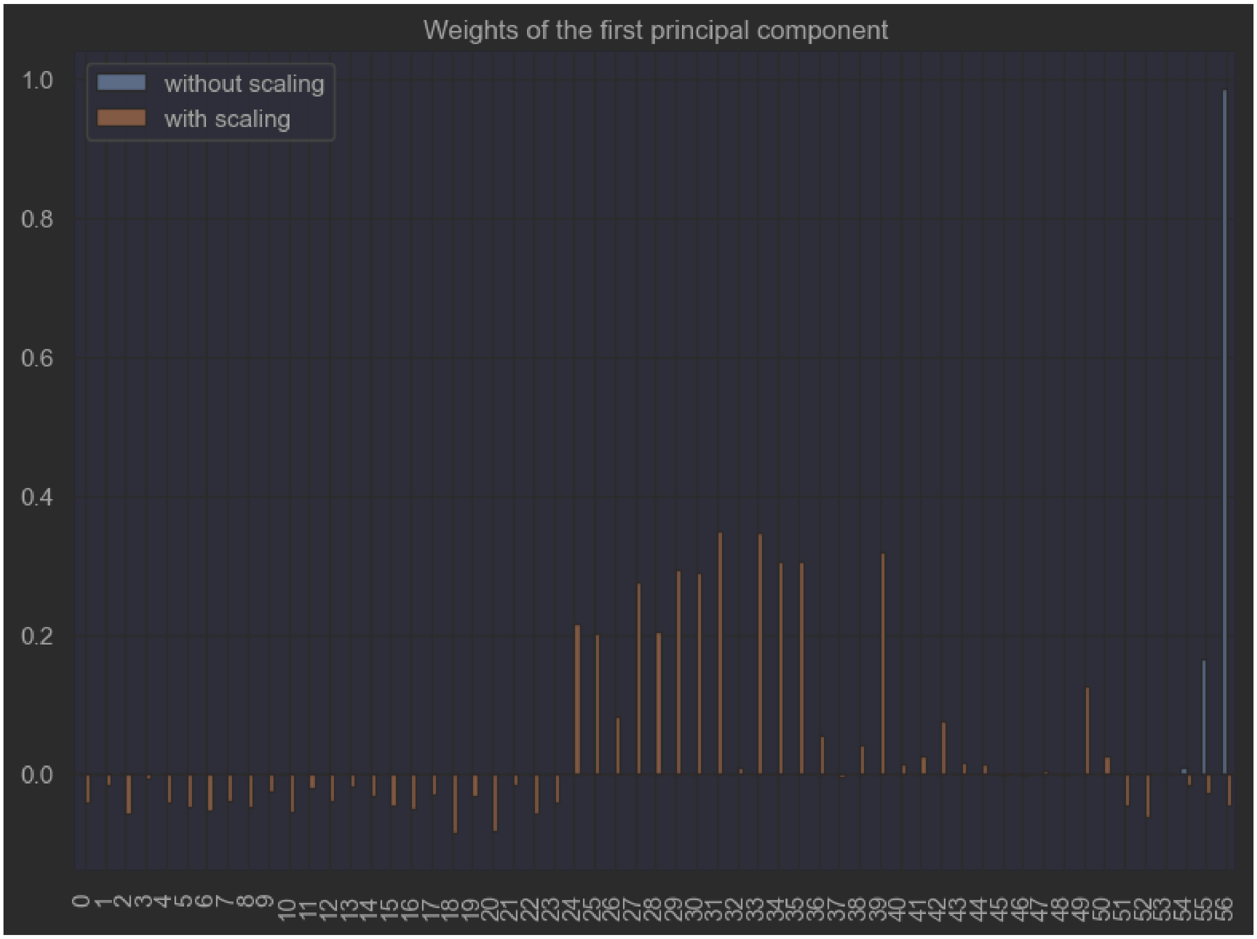

Figure 4 Weights of the 1st principal component with and without scaling

Dimensionality reduction

As the data set consists of 57 features, which is a huge amount, we have to do dimensionality reduction.

Here Principal component Analysis(PCA) is used to compress the get a dimensionally reduced smaller set of uncorrelated features by finding a set of variables called principal components. These components capture the most variance in the data. Before using the KNN and Decision tree machine learning techniques the PCA is used.

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

# Visualize data
plt.scatter(X_pca[y==0, 0], X_pca[y==0, 1], label='Non-spam') # creates datapoints that are labeled as non spam
plt.scatter(X_pca[y==1, 0], X_pca[y==1, 1], label='Spam')
plt.xlabel("first principal component")
plt.ylabel("second principal component")
plt.legend()
plt.show()
```

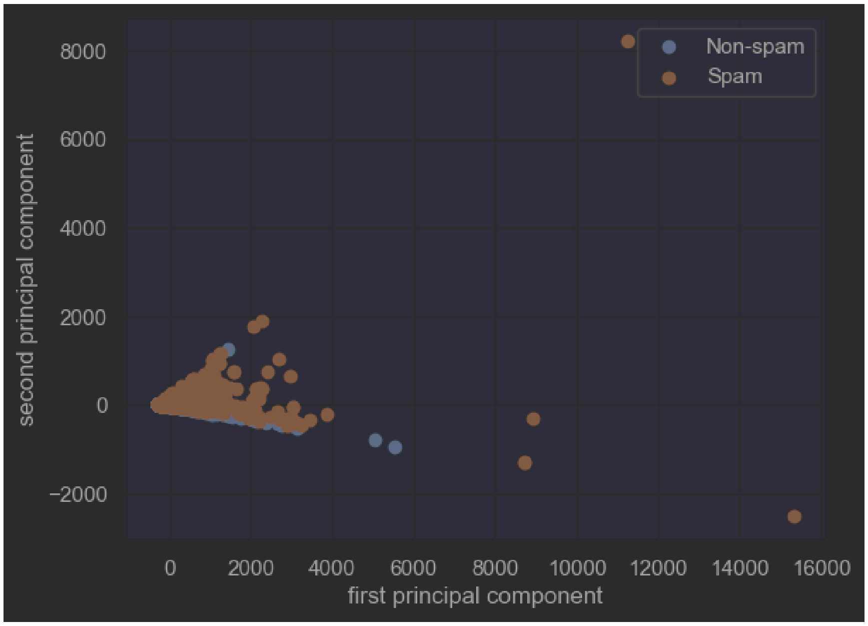

Figure 5 Scatterplot of 1st two principal components & separation between spam and non-spam emails

Here the spam and non-spam data points are called separately.

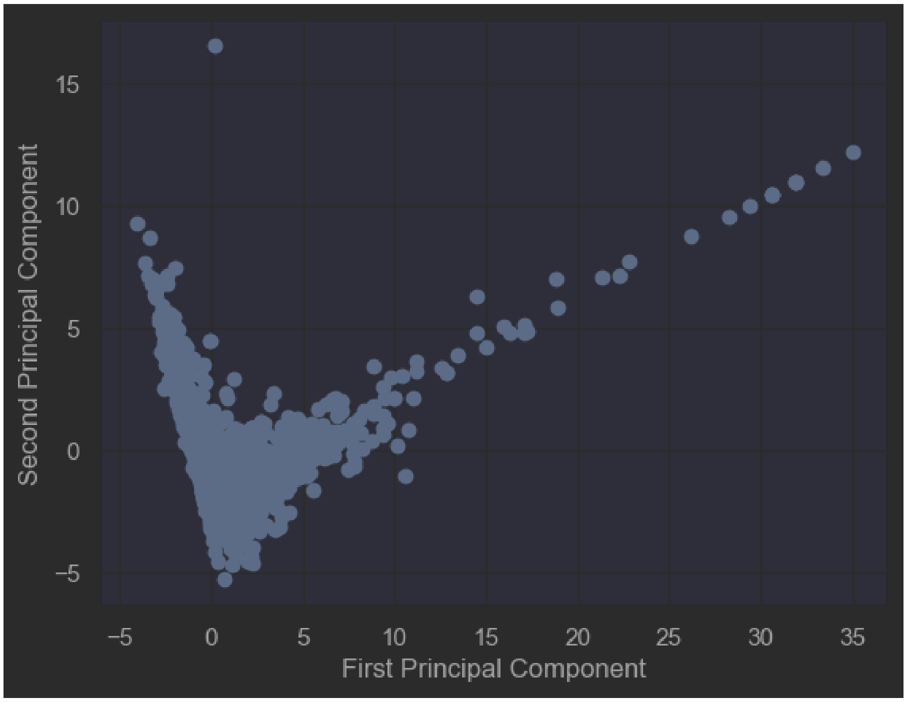

Figure 6 1st two principal components against each other

In figure 9 the color of the points indicates the density of all the data points.

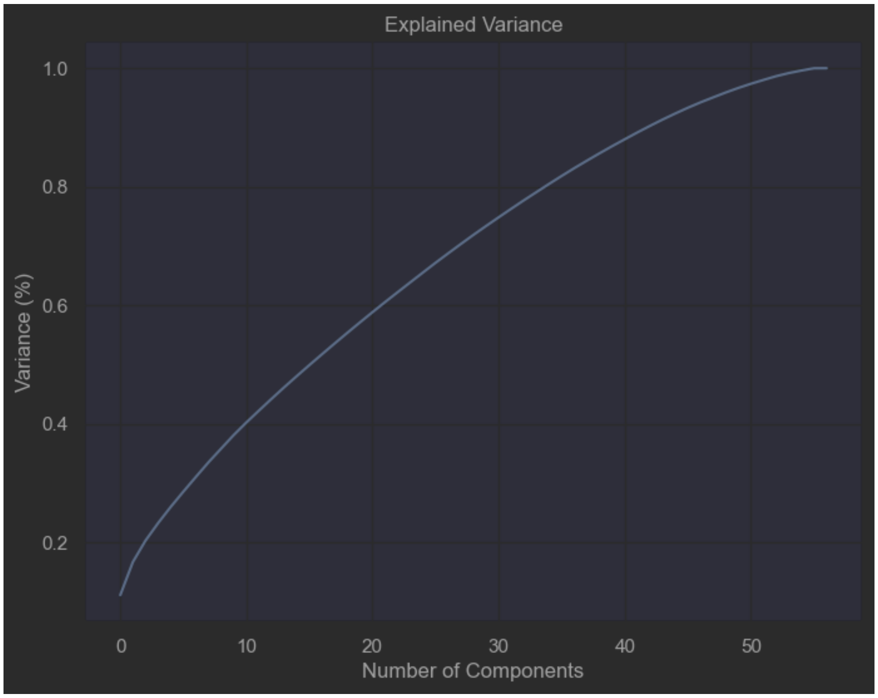

Figure 7 PCA variance % and number of components considered

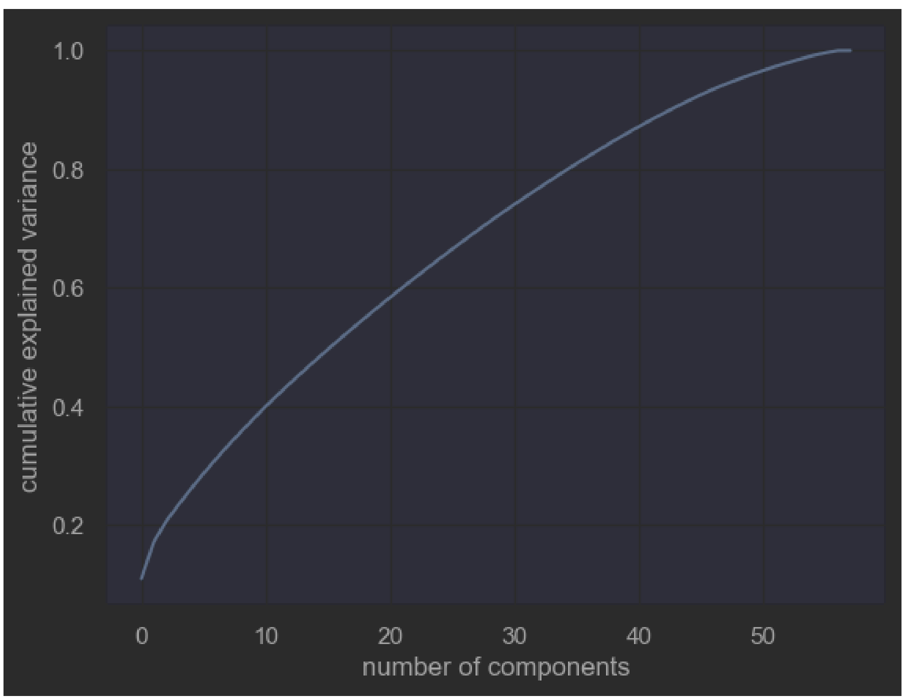

Figure 8 plot of number of components vs cumulative explained variance

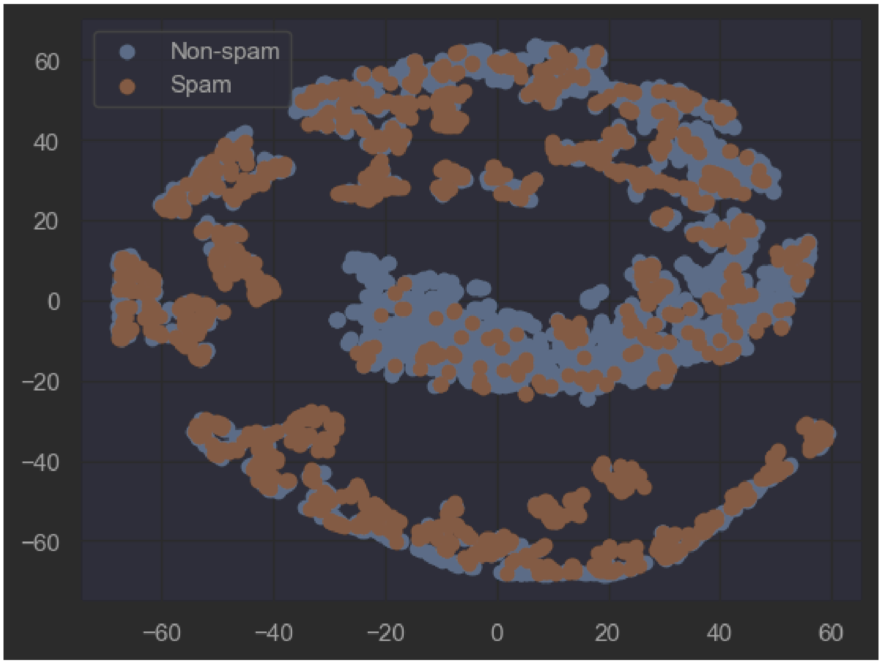

Figure 9 scatterplot of transformed data points by TSNE

KNN

Model evaluation

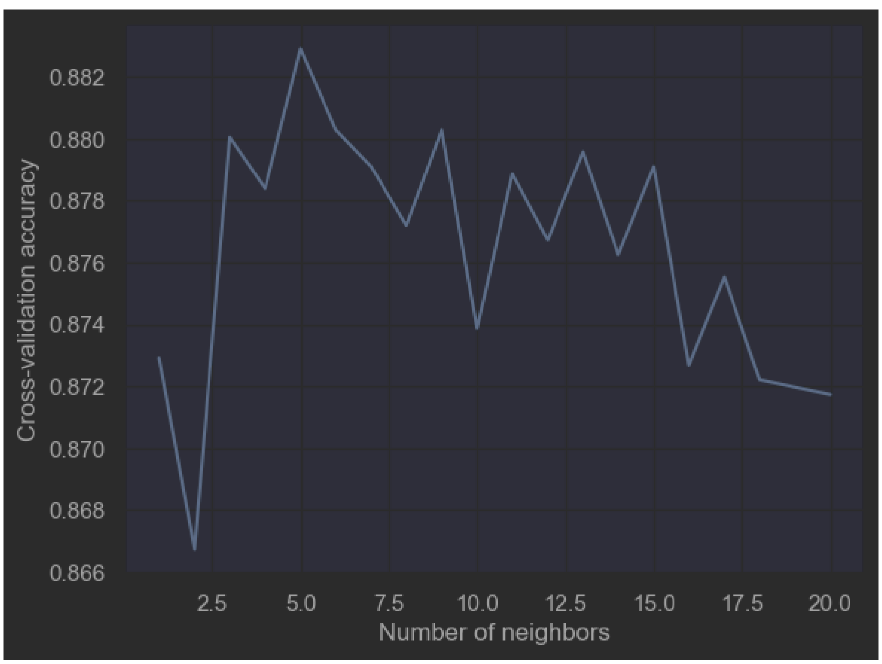

Figure 10 cross-validation accuracy vs n_neighbors

Here the cross validation (CV) accuracy is plotted as a function of number neighbors considered in KNN. As we can see the CV accuracy is highest when n_neighbors=5. From gridsearch and checking best performing n_neighbor value also given as 5 with a mean score of 0.9024433790045938.

Therefore the KNN is applied with the nearest neighbor value as 5.

Accuracy, precision, recall,F1 score

accuracy of KNN model: 0.886039886039886

Precision of KNN model: 0.8918918918918919

Sensitivity of KNN model: 0.826879271070615

Specificity of KNN model: 0.826879271070615

Figure 11 Evaluation of KNN

confusion matrix

0	0.94	0.89	0.91	633
1	0.84	0.91	0.87	420
accuracy			0.90	1053
macro avg	0.89	0.90	0.89	1053
weighted avg	0.90	0.90	0.90	1053

```
[[562  71]
 [ 39 381]]
```

Figure 12 Confusion matrix for KNN

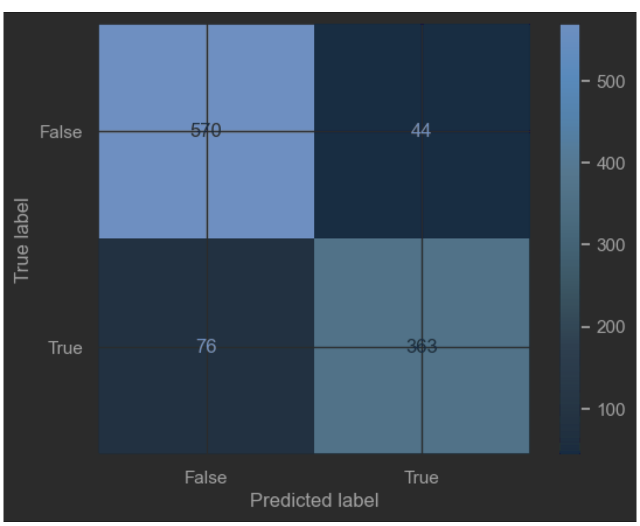

Figure 13 Confusion matrix-KNN

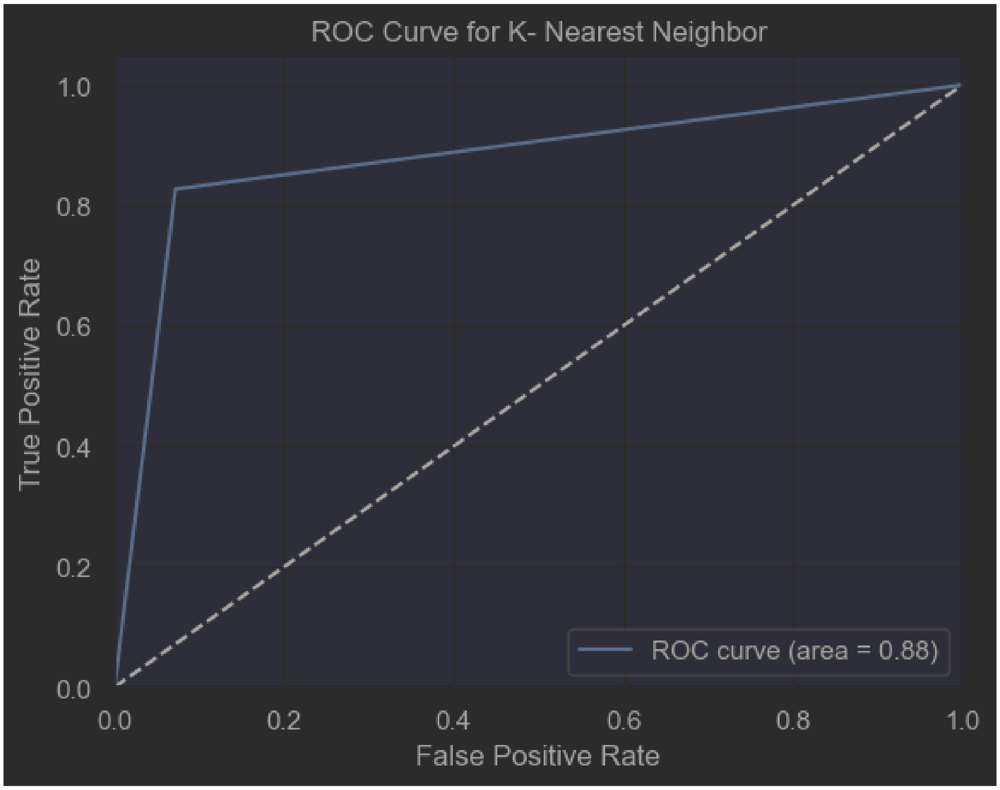

Figure 14 ROC Curve for KNN

Experimental results

```
Train score 0.9331643965790307
Test score 0.8860398860398860
Train Confusion matrix
```

Figure 15 Train and test score of KNN

Decision Tree

The cleaned dataset is used with the model.

Model evaluation

Fitting data without fine tuning

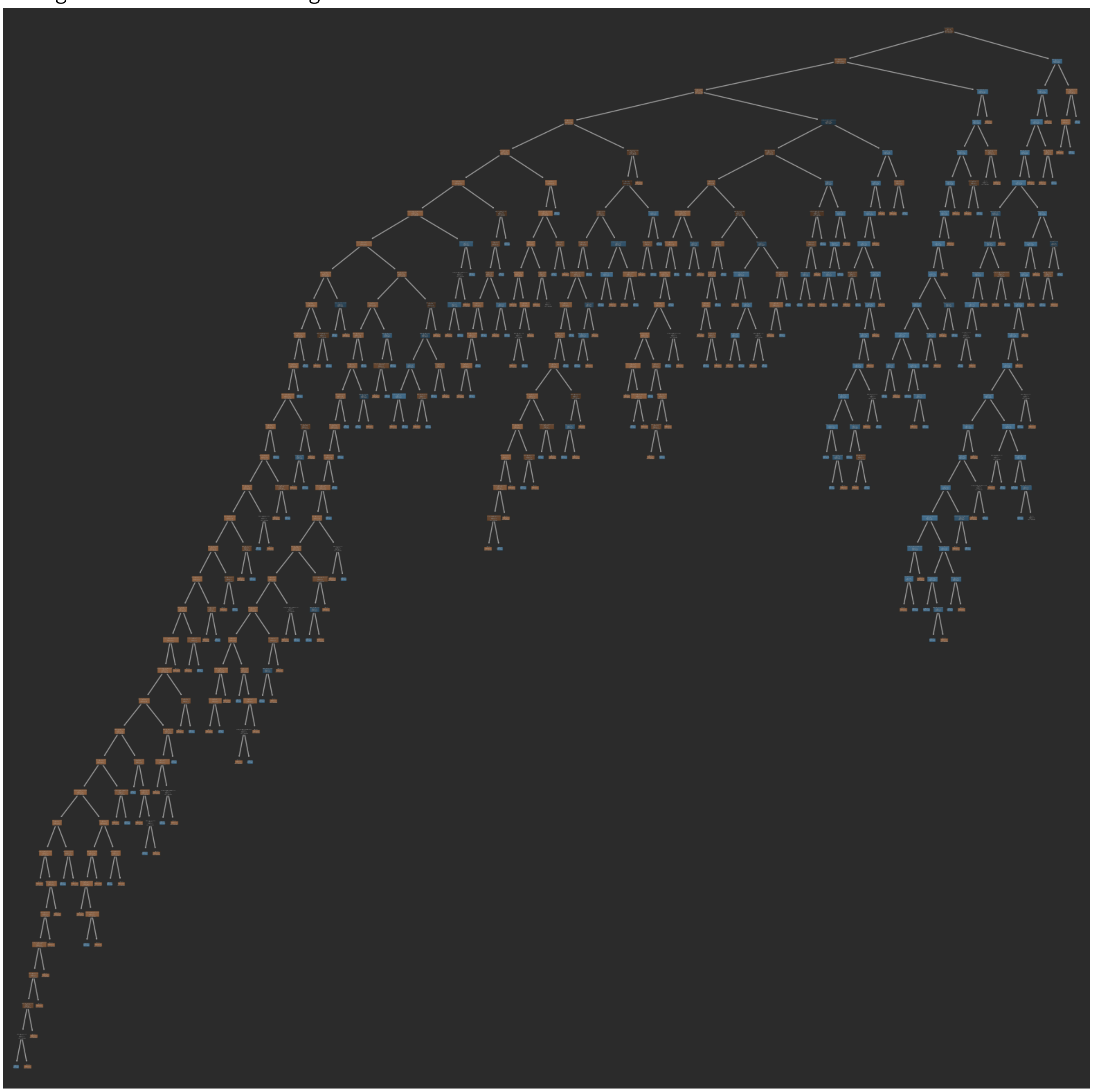

Table 4 Decision tree before fine tuning

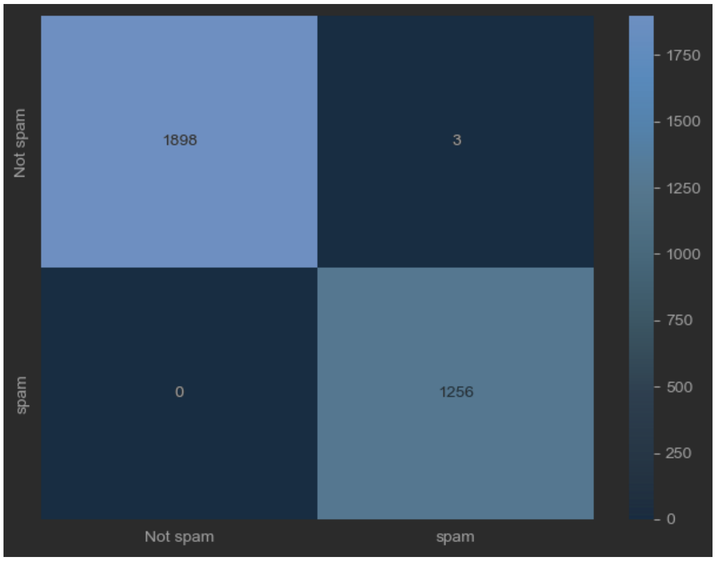

Figure 16 Train confusion matrix before pruning

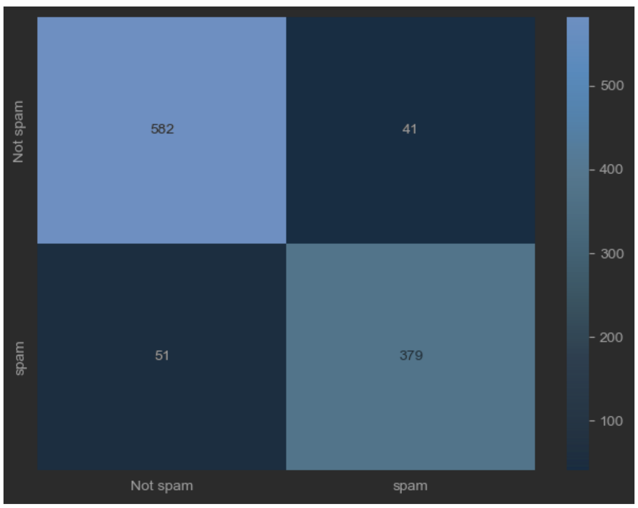

Figure 17 Test confusion matrix before pruning.

Pre pruning

Here we stop the growing of the tree at an early stage by setting constraints. The grid search through parameters is done and the optimum values are chosen.

Here following parameters are controlled.

- maximum depth of the tree
- minimum number of samples needed to split an interval node.
- minimum number of samples needed to be a leaf node.

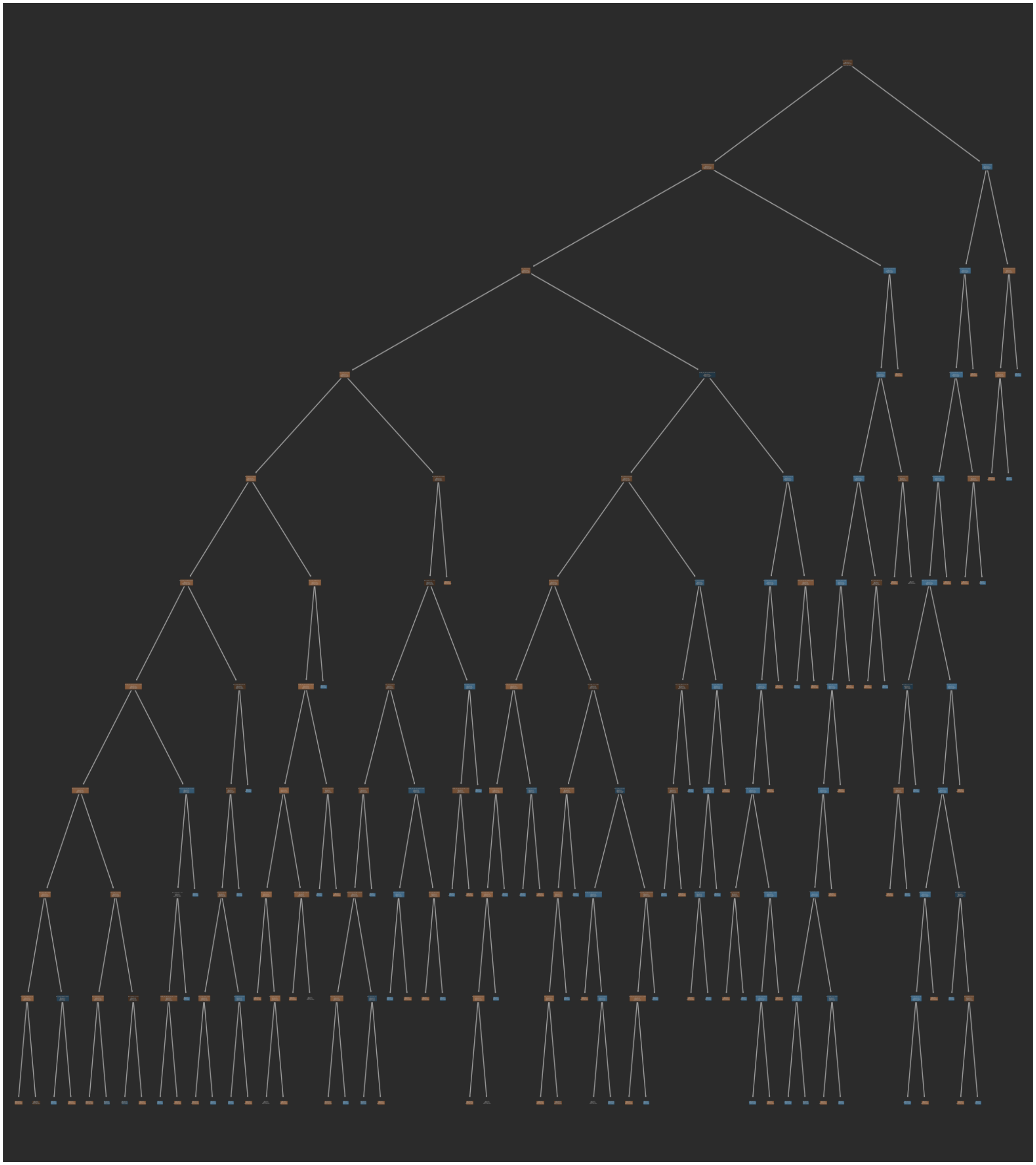

Figure 18 Decision tree after pre-pruning

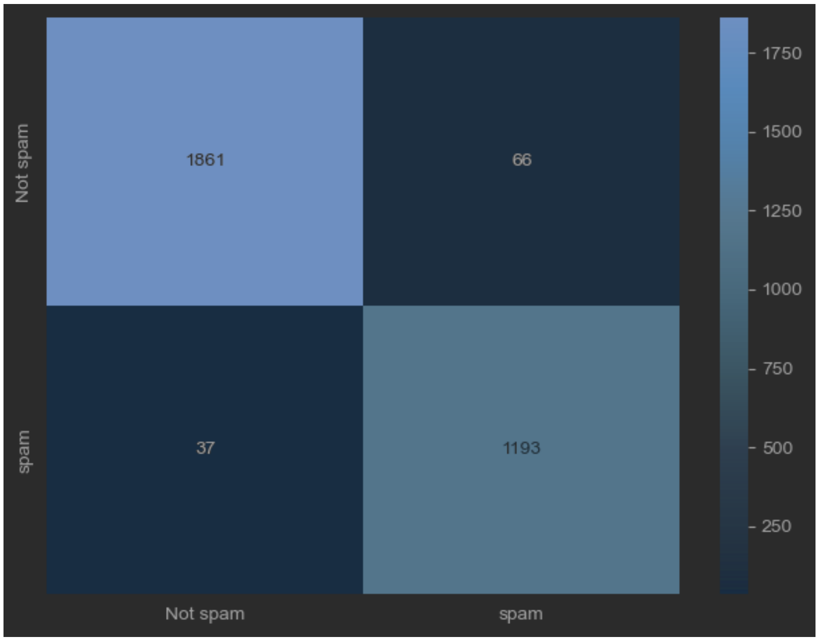

Figure 19 Confusion matrix for training set -after pre pruning

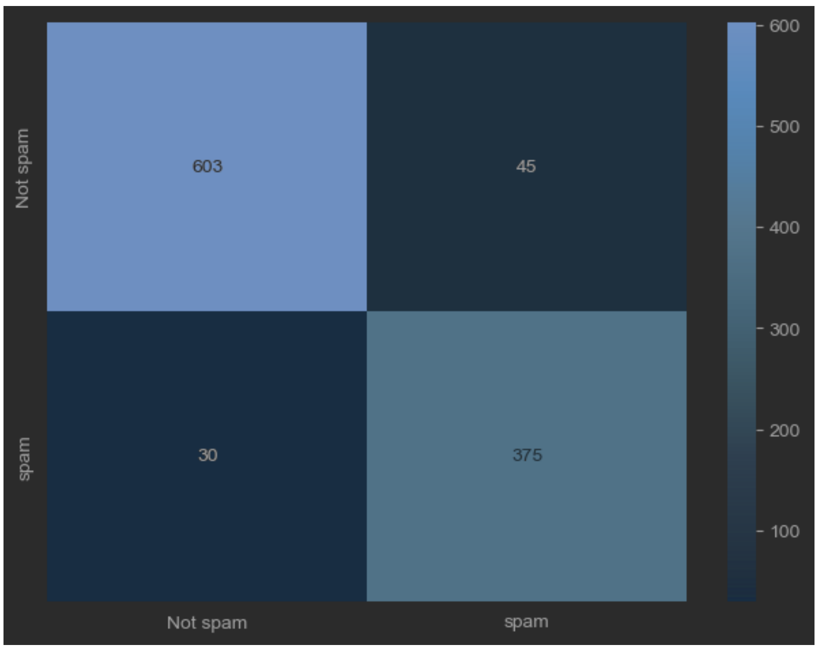

Figure 20 Test confusion matric after pre pruning

After pruning there is an improvement in test accuracy.

Post pruning

For further improvements let's do cost complexity pruning as a post pruning technique to avoid overfitting as decision trees are more likely to get overfitted.

Cost complexity pruning

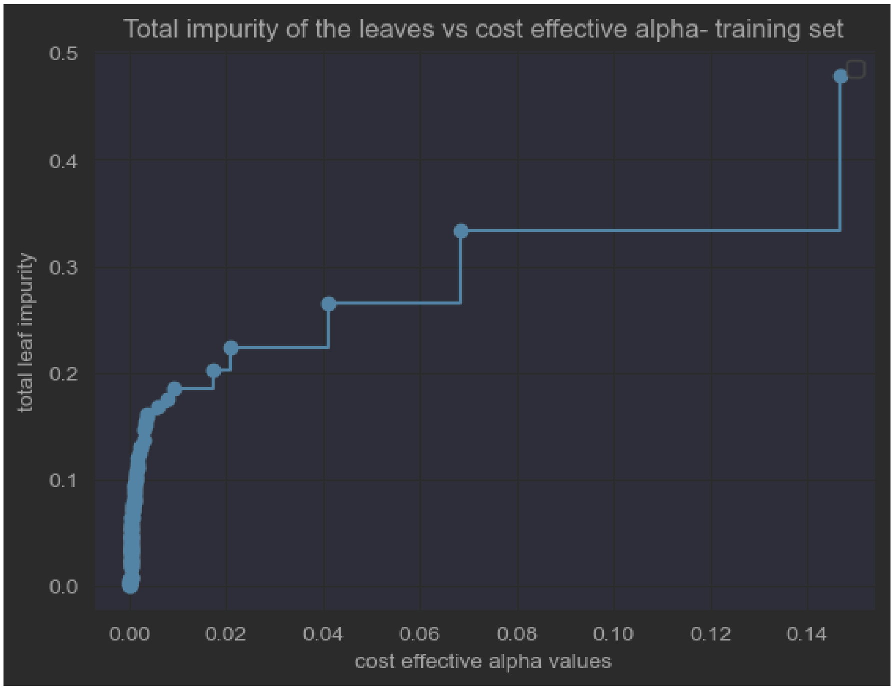

Figure 21 plot of leaf impurity vs alpha values

Here shows the total leaf impurities as a function of alpha values.

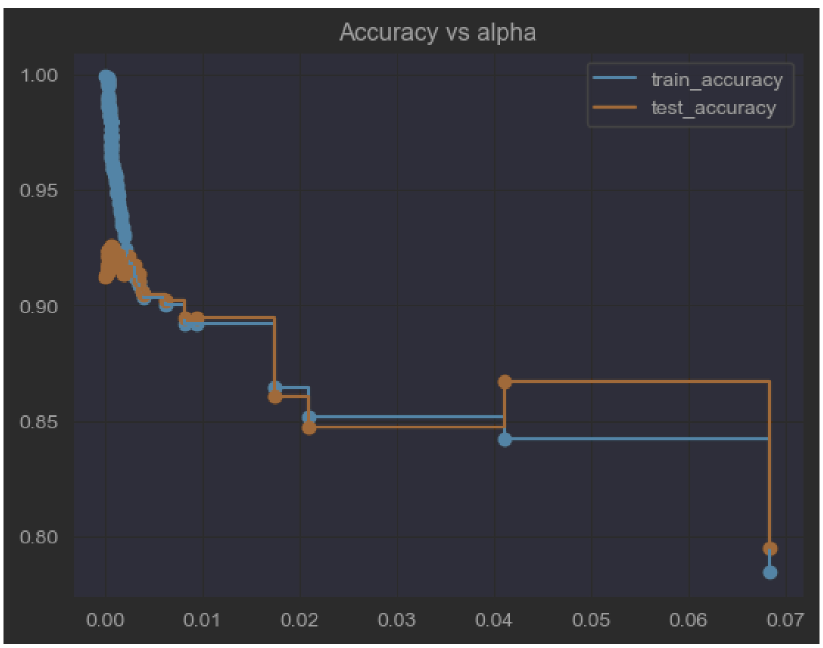

Figure 22 Test and Train accuracy vs alpha

Evaluation criteria

Accuracy, precision, sensitivity, specificity

accuracy of Decision Tree model: 0.8955365622032289

Precision of Decision Tree model: 0.8429203539823009

Sensitivity of Decision Tree model: 0.9071428571428571

Specificity of Decision Tree model: 0.9071428571428571

Step	Train data set		Test data set	
	Confusion matrix	score	Confusion matrix	score
Without fine tuning	[[1898 3] [0 1256]]	0.9990497307570478	[[582 41] [51 379]]	0. 912630579297246
After Pre-pruning	[[1861 66] [37 1193]]	0.9673740893253089	[[603 45] [30 375]]	0.9287749287749287
After Post-pruning	[[1802 246] [96 1013]]	0.8916693063034526	[[596 76] [35 344]]	0.8945868945868946

Table 5 Decision tree test and train score

Confusion Matrix

```
           0       0.93      0.91      0.92       633
           1       0.87      0.90      0.88       420

    accuracy                           0.91      1053
   macro avg       0.90      0.90      0.90      1053
weighted avg       0.91      0.91      0.91      1053

[[579  54]
 [ 44 376]]
```

Figure 23 Confusion matrix of Decision Tree

ROC Curve

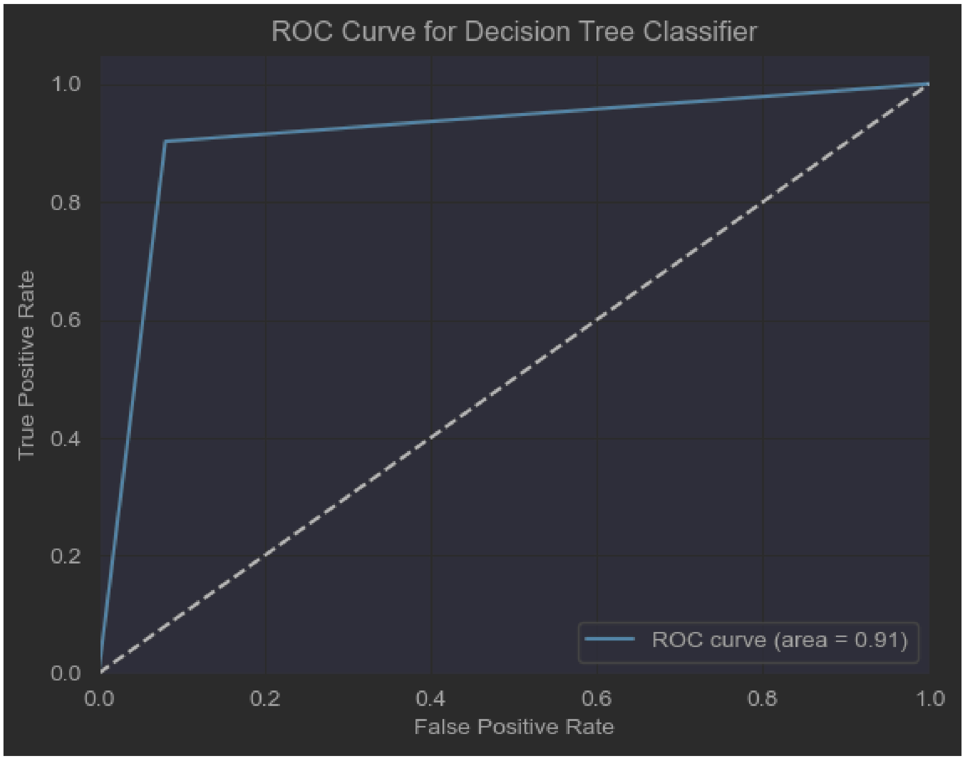

Figure 24 ROC curve of Decision Tree

Experimental results

The Decision Tree model gives a 0.8946 accuracy when trained and tested.

```
Train score 0.8916693063034526
Test score 0.8945868945868946
```

Figure 25 Test and Train score

Evaluation of two models

	KNN		Decision Tree
accuracy		>	
Precision		<	
Sensitivity		>	
Specificity		>	
F1 score		<	

Both the models shares almost the same accuracy. Here it shows there are some changes between two models in precision, sensitivity, specificity and f1 score.

Table 6 Model evaluation

Limitations and Possible further enhancements

Limitations

Limitations- KNN Model

- Performs poorly on high dimensional datasets.
- Computationally expensive when the data set is large as the algorithm calculates the distance between the data of the data set
- Choosing of k-nearest neighbors is crucial as it highly depends on the accuracy of the model

Limitations – Decision Tree

- Tend to overfit.
- When there is a large number of levels the algorithm can be biased towards those variables.
- Sensitive to small changes in the data.

Future Enhancements- KNN Model

- Perform on high dimensional datasets by reducing the dimension to an optimum level.
- Use distance weighting.

Future Enhancements – Decision Tree

- Minimize overfitting using pruning, setting up constraints in an optimal level.
- Getting a cleaner data set with reduced number of outliers, missing values.
- Using ensemble techniques to improve the performance.(Random Forest/Boosting)

Limitations of data set

- Dataset is not very large.
- Duplicate data were present.

References

https://archive.ics.uci.edu/ml/datasets/Spambase
https://datascience.stackexchange.com/questions/38395/standardscaler-before-or-after-splitting-data-which-is-better

https://stackoverflow.com/questions/63037248/is-it-correct-to-use-a-single-standardscaler-before-splitting-data

https://stackoverflow.com/questions/37221425/which-feature-scaling-method-to-use-before-pca

https://www.researchgate.net/post/Is-it-necessary-to-normalize-data-before-performing-principle-component-analysis

https://www.kaggle.com/code/arunmohan003/pruning-decision-trees-tutorial

https://stats.stackexchange.com/questions/202287/why-standardization-of-the-testing-set-has-to-be-performed-with-the-mean-and-sd

https://machinelearningmastery.com/tune-number-size-decision-trees-xgboost-python/

Appendix

KNN-python notebook


```
{
 "cells": [
  {
   "cell_type": "markdown",
   "source": [
    "# Importing the dataset"
   ],
   "metadata": {
    "collapsed": false
   }
  },
  {
   "cell_type": "code",
   "execution_count": null,
   "outputs": [],
   "source": [
    "import pandas as pd\n",
    "import matplotlib.pyplot as plt\n",
    "\n",
    "data = pd.read_csv(\"spambase.csv\")\n",
    "data.head()"
   ],
   "metadata": {
    "collapsed": false
   }
  },
  {
   "cell_type": "code",
   "execution_count": null,
   "outputs": [],
   "source": [
    "data.isnull().sum() #check for null values/missing values in each
column"
   ],
   "metadata": {
    "collapsed": false
   }
  },
  {
   "cell_type": "code",
   "execution_count": null,
   "outputs": [],
   "source": [
    "data.shape # before dropping duplicates"
   ],
   "metadata": {
    "collapsed": false
   }
```

```
  },
  {
   "cell_type": "code",
   "execution_count": null,
   "outputs": [],
   "source": [
    "data =data.drop_duplicates() #drop duplicates"
   ],
   "metadata": {
    "collapsed": false
   }
  },
  {
   "cell_type": "code",
   "execution_count": null,
   "outputs": [],
   "source": [
    "print(data.shape)"
   ],
   "metadata": {
    "collapsed": false
   }
  },
  {
   "cell_type": "code",
   "execution_count": null,
   "outputs": [],
   "source": [
    "data.info()"
   ],
   "metadata": {
    "collapsed": false
   }
  },
  {
   "cell_type": "code",
   "execution_count": null,
   "outputs": [],
   "source": [
    "data.describe()"
   ],
   "metadata": {
    "collapsed": false
   }
  },
  {
   "cell_type": "code",
   "execution_count": null,
   "outputs": [],
   "source": [
    "#Save the cleaned processed data set into a new csv file\n",
    "from pathlib import Path\n",
    "filepath = Path(\"newSpamData.csv\")\n",
    "filepath.parent.mkdir(parents=True,exist_ok=True)\n",
    "data.to_csv(filepath)"
   ],
   "metadata": {
```

```
    "collapsed": false
   }
  },
  {
   "cell_type": "code",
   "execution_count": null,
   "outputs": [],
   "source": [
    "import seaborn as sns\n",
    "#Using Pearson Correlation\n",
    "plt.figure(figsize=(10,10))\n",
    "cor = data.corr()\n",
    "sns.heatmap(cor, cmap=plt.cm.Reds)\n",
    "plt.show()"
   ],
   "metadata": {
    "collapsed": false
   }
  },
  {
   "cell_type": "code",
   "execution_count": null,
   "outputs": [],
   "source": [
    "#Correlation with output variable\n",
    "cor_target = abs(cor[\"spam\"])\n",
    "#Selecting highly correlated features\n",
    "relevant_features = cor_target[cor_target>0.15]\n",
    "relevant_features"
   ],
   "metadata": {
    "collapsed": false
   }
  },
  {
   "cell_type": "code",
   "execution_count": null,
   "outputs": [],
   "source": [
    "relevant_features.count()"
   ],
   "metadata": {
    "collapsed": false
   }
  },
  {
   "cell_type": "markdown",
   "source": [
    "# divide into X and y data sets"
   ],
   "metadata": {
    "collapsed": false
   }
  },
  {
   "cell_type": "code",
   "execution_count": null,
```

```
   "outputs": [],
   "source": [
    "X = data.iloc[:, 0:-1].values  #get all the rows of all the columns
except last one to X data set #Features\n",
    "y = data.iloc[:, -1].values  # get the y data set by taking the lst
column of all the rows # Target variables"
   ],
   "metadata": {
    "collapsed": false
   }
  },
  {
   "cell_type": "markdown",
   "source": [
    "# Splitting the data set  into training and test sets"
   ],
   "metadata": {
    "collapsed": false
   }
  },
  {
   "cell_type": "code",
   "execution_count": null,
   "outputs": [],
   "source": [
    "from sklearn.model_selection import train_test_split\n",
    "X_train,X_test,y_train,y_test = train_test_split(X,y,test_size=0.25,
random_state=42, shuffle=True)"
   ],
   "metadata": {
    "collapsed": false
   }
  },
  {
   "cell_type": "code",
   "execution_count": null,
   "outputs": [],
   "source": [
    "data.boxplot()"
   ],
   "metadata": {
    "collapsed": false
   }
  },
  {
   "cell_type": "markdown",
   "source": [
    "# Feature scaling\n",
    "a method used to normalize the range of independent variables or
features of data.\n",
    "It normalizes data within a certain range"
   ],
   "metadata": {
    "collapsed": false
   }
  },
  {
```

```
  "cell_type": "code",
  "execution_count": null,
  "outputs": [],
  "source": [
   "from sklearn.preprocessing import StandardScaler\n",
   "\n",
   "ssc = StandardScaler()\n",
   "X_train_ssc = ssc.fit_transform(X_train)\n",
   "X_test_ssc = ssc.transform(X_test)"
  ],
  "metadata": {
   "collapsed": false
  }
 },
 {
  "cell_type": "code",
  "execution_count": null,
  "outputs": [],
  "source": [
   "from pandas import DataFrame\n",
   "\n",
   "scaled = ssc.fit_transform(data)\n",
   "scaled = DataFrame(scaled)\n",
   "scaled.boxplot()"
  ],
  "metadata": {
   "collapsed": false
  }
 },
 {
  "cell_type": "markdown",
  "source": [],
  "metadata": {
   "collapsed": false
  }
 },
 {
  "cell_type": "markdown",
  "source": [
   "# PCA"
  ],
  "metadata": {
   "collapsed": false
  }
 },
 {
  "cell_type": "code",
  "execution_count": null,
  "outputs": [],
  "source": [
   "# # Normalize data\n",
   "from sklearn.decomposition import PCA\n",
   "X_norm = (X - X.mean()) / X.std()"
  ],
  "metadata": {
   "collapsed": false
  }
```

```
  },
  {
   "cell_type": "code",
   "execution_count": null,
   "outputs": [],
   "source": [
    "pca = PCA(n_components=2).fit(X_train)\n",
    "scaled_pca = PCA(n_components=2).fit(X_train_ssc)\n",
    "X_train_transformed = pca.transform(X_train)\n",
    "X_train_std_transformed = scaled_pca.transform(X_train_ssc)\n",
    "\n",
    "first_pca_component = pd.DataFrame(\n",
    "    pca.components_[0],  columns=[\"without scaling\"]\n",
    ")\n",
    "first_pca_component[\"with scaling\"] = scaled_pca.components_[0]\n",
    "first_pca_component.plot.bar(\n",
    "    title=\"Weights of the first principal component\", figsize=(8,
6)\n",
    ")\n",
    "\n",
    "_ = plt.tight_layout()"
   ],
   "metadata": {
    "collapsed": false
   }
  },
  {
   "cell_type": "code",
   "execution_count": null,
   "outputs": [],
   "source": [
    "# Reduce dimensionality using PCA\n",
    "pca = PCA()\n",
    "X_pca = pca.fit_transform(X)\n",
    "\n",
    "# Visualize data\n",
    "plt.scatter(X_pca[y==0, 0], X_pca[y==0, 1], label='Non-spam') # creates
datapoints that are labeled as non spam\n",
    "plt.scatter(X_pca[y==1, 0], X_pca[y==1, 1], label='Spam')\n",
    "plt.xlabel(\"first principal component\")\n",
    "plt.ylabel(\"second principal component\")\n",
    "plt.legend()\n",
    "plt.show()"
   ],
   "metadata": {
    "collapsed": false
   }
  },
  {
   "cell_type": "code",
   "execution_count": null,
   "outputs": [],
   "source": [
    "from sklearn.decomposition import PCA\n",
    "import numpy as np\n",
    "# Reduce dimensionality using PCA\n",
    "X_train_pca = pca.fit_transform(X_train_ssc)\n",
```

```
    "\n",
    "X_test_pca = pca.transform(X_test_ssc)\n",
    "\n",
    "plt.figure(figsize=(8,6))\n",
    "plt.plot(np.cumsum(pca.explained_variance_ratio_))\n",
    "plt.xlabel('Number of Components')\n",
    "plt.ylabel('Variance (%)') #for each component\n",
    "\n",
    "plt.title('Explained Variance')\n",
    "plt.show()"
   ],
   "metadata": {
    "collapsed": false
   }
  },
  {
   "cell_type": "code",
   "execution_count": null,
   "outputs": [],
   "source": [
    "# Visualize data\n",
    "scaled_pca = pca.fit_transform(scaled)\n",
    "plt.scatter(scaled_pca[:, 0], scaled_pca[:, 1], cmap ='plasma')\n",
    "\n",
    "# labeling x and y axes\n",
    "plt.xlabel('First Principal Component')\n",
    "plt.ylabel('Second Principal Component')"
   ],
   "metadata": {
    "collapsed": false
   }
  },
  {
   "cell_type": "code",
   "execution_count": null,
   "outputs": [],
   "source": [
    "import seaborn as sns\n",
    "sns.set()\n",
    "pca = PCA().fit(scaled)\n",
    "plt.plot(np.cumsum(pca.explained_variance_ratio_))\n",
    "plt.xlabel('number of components')\n",
    "plt.ylabel('cumulative explained variance');"
   ],
   "metadata": {
    "collapsed": false
   }
  },
  {
   "cell_type": "markdown",
   "source": [
    "# t-SNE"
   ],
   "metadata": {
    "collapsed": false
   }
  },
```

```
  {
   "cell_type": "code",
   "execution_count": null,
   "outputs": [],
   "source": [
    "# Reduce dimensionality using t-SNE\n",
    "from sklearn.manifold import TSNE\n",
    "tsne = TSNE(n_components=2, random_state=42)\n",
    "X_tsne = tsne.fit_transform(X)\n",
    "\n",
    "# Visualize data\n",
    "plt.scatter(X_tsne[y==0, 0], X_tsne[y==0, 1], label='Non-spam')\n",
    "plt.scatter(X_tsne[y==1, 0], X_tsne[y==1, 1], label='Spam')\n",
    "plt.legend()\n",
    "plt.show()"
   ],
   "metadata": {
    "collapsed": false
   }
  },
  {
   "cell_type": "markdown",
   "source": [
    "# KNN"
   ],
   "metadata": {
    "collapsed": false
   }
  },
  {
   "cell_type": "code",
   "execution_count": null,
   "outputs": [],
   "source": [
    "from sklearn.neighbors import KNeighborsClassifier\n",
    "\n",
    "knn = KNeighborsClassifier(n_neighbors=3,metric=\"minkowski\", p=2)\n",
    "# p - power parameter # metric-metrics used for distance computation #
minkowski- euclidean distance\n",
    "knn.fit(X_train_pca,y_train) #fitting classifier to the training set"
   ],
   "metadata": {
    "collapsed": false
   }
  },
  {
   "cell_type": "code",
   "execution_count": null,
   "outputs": [],
   "source": [
    "y_knn_predic = knn.predict(X_test_pca) # predicting the test set
results\n",
    "print (y_knn_predic[0:5])\n",
    "print (y_test[0:5])\n",
    "knn.score(X_test_pca,y_knn_predic)\n",
    "# to visually compare the prediction to the actual values"
   ],
```

```
  "metadata": {
   "collapsed": false
  }
 },
 {
  "cell_type": "markdown",
  "source": [
   "Cross-validation accuracy vs number of nearest neighbors"
  ],
  "metadata": {
   "collapsed": false
  }
 },
 {
  "cell_type": "code",
  "execution_count": null,
  "outputs": [],
  "source": [
   "from sklearn.model_selection import cross_val_score\n",
   "import numpy as np\n",
   "\n",
   "# Define range of values for n_neighbors\n",
   "n_neighbors_range = range(1, 21)\n",
   "# Perform cross-validation with different values of n_neighbors\n",
   "X_ssc = ssc.fit_transform(X)\n",
   "\n",
   "\n",
   "cv = 5\n",
   "cv_scores = []\n",
   "for n_neighbors in n_neighbors_range:\n",
   "    knn = KNeighborsClassifier(n_neighbors=n_neighbors)\n",
   "    scores = cross_val_score(knn, X_ssc,y , cv=cv)\n",
   "    cv_scores.append(np.mean(scores))\n",
   "\n",
   "# Plot cross-validation accuracy as a function of n_neighbors\n",
   "plt.plot(n_neighbors_range, cv_scores)\n",
   "plt.xlabel('Number of neighbors')\n",
   "plt.ylabel('Cross-validation accuracy')\n",
   "plt.show()"
  ],
  "metadata": {
   "collapsed": false
  }
 },
 {
  "cell_type": "markdown",
  "source": [],
  "metadata": {
   "collapsed": false
  }
 },
 {
  "cell_type": "code",
  "execution_count": null,
  "outputs": [],
  "source": [
   "# from sklearn.model_selection import cross_val_score\n",
```

```
    "# import numpy as np\n",
    "#\n",
    "# #create a new KNN model\n",
    "# knn_cv = KNeighborsClassifier(n_neighbors=5)\n",
    "#\n",
    "# #train model with cv of 5\n",
    "# cv_scores = cross_val_score(knn_cv, X, y, cv=5)\n",
    "#\n",
    "# #print each cv score (accuracy) and average them\n",
    "# print(cv_scores)\n",
    "# print('cv_scores mean:{}'.format(np.mean(cv_scores)))"
   ],
   "metadata": {
    "collapsed": false
   }
  },
  {
   "cell_type": "code",
   "execution_count": null,
   "outputs": [],
   "source": [
    "from sklearn.model_selection import GridSearchCV\n",
    "\n",
    "#create new a knn model\n",
    "knn2 = KNeighborsClassifier()\n",
    "\n",
    "#create a dictionary of all values we want to test for n_neighbors\n",
    "param_grid = {'n_neighbors': np.arange(1, 25)}\n",
    "\n",
    "# Task 3: Use gridsearch to test all values for n_neighbors\n",
    "knn_gscv = GridSearchCV(KNeighborsClassifier(),param_grid)\n",
    "\n",
    "#fit model to data\n",
    "knn_gscv.fit(X_train_pca,y_train)\n",
    "\n",
    "#check top performing n_neighbors value\n",
    "knn_gscv.best_params_"
   ],
   "metadata": {
    "collapsed": false
   }
  },
  {
   "cell_type": "code",
   "execution_count": null,
   "outputs": [],
   "source": [
    "#check mean score for the top performing value of n_neighbors\n",
    "knn_gscv.best_score_"
   ],
   "metadata": {
    "collapsed": false
   }
  },
  {
   "cell_type": "code",
   "execution_count": null,
```

```
   "outputs": [],
   "source": [
    "# get the model for 5 nearest neighbors\n",
    "knn_new = KNeighborsClassifier(n_neighbors=5)\n",
    "knn_new.fit(X_train_pca,y_train)\n",
    "knn_predicted_new = knn_new.predict(X_test_pca)\n",
    "knn_new.score(X_test_pca,knn_predicted_new)"
   ],
   "metadata": {
    "collapsed": false
   }
  },
  {
   "cell_type": "code",
   "execution_count": null,
   "outputs": [],
   "source": [
    "# Getting the Confusion Matrix\n",
    "from sklearn import metrics\n",
    "from sklearn.metrics import confusion_matrix\n",
    "cm = confusion_matrix(y_test, knn_predicted_new)\n",
    "cm_display =
metrics.ConfusionMatrixDisplay(confusion_matrix=cm,display_labels =
[False,True])\n",
    "cm_display.plot(cmap=plt.cm.Blues)\n",
    "plt.show()"
   ],
   "metadata": {
    "collapsed": false
   }
  },
  {
   "cell_type": "markdown",
   "source": [
    "accuracy = True +ve/(True +ve + False +ve)\n",
    "sensitivity = True +ve/(True +ve + False -ve)\n",
    "specificity = True -ve/(True +ve + False -ve)\n",
    "F-score = 2*(Precision*sensitivity)/(Precision+sensitivity)"
   ],
   "metadata": {
    "collapsed": false
   }
  },
  {
   "cell_type": "code",
   "execution_count": null,
   "outputs": [],
   "source": [
    "from sklearn.metrics import precision_score\n",
    "from sklearn import metrics\n",
    "\n",
    "accuracy_knn =metrics.accuracy_score(y_test,knn_predicted_new)\n",
    "prec_knn = precision_score(y_test, knn_predicted_new)\n",
    "Sensitivity_recall_knn = metrics.recall_score(y_test, knn_predicted_new)
# Sensitivity- how well the model predicts something is positive\n",
    "Specificity_knn = metrics.recall_score(y_test, knn_predicted_new) #
Specificity- how well the model predicts something is negative\n",
```

```
    "F_score_knn = metrics.f1_score(y_test,knn_predicted_new)\n",
    "\n",
    "print(\"accuracy of KNN  model: \",accuracy_knn)\n",
    "print(\"Precision of KNN  model: \",prec_knn)\n",
    "print(\"Sensitivity of KNN  model: \",Sensitivity_recall_knn)\n",
    "print(\"Specificity of KNN model: \",Specificity_knn)\n",
    "print(\"F1 score of KNN  model: \",F_score_knn)"
   ],
   "metadata": {
    "collapsed": false
   }
  },
  {
   "cell_type": "code",
   "execution_count": null,
   "outputs": [],
   "source": [
    "from sklearn.metrics import classification_report\n",
    "\n",
    "print(classification_report(y_test, knn_predicted_new))\n",
    "print(confusion_matrix(y_test, knn_predicted_new))"
   ],
   "metadata": {
    "collapsed": false
   }
  },
  {
   "cell_type": "code",
   "execution_count": null,
   "outputs": [],
   "source": [
    "from sklearn.metrics import confusion_matrix\n",
    "import seaborn as sns\n",
    "features = data.columns\n",
    "classes = ['Not spam','spam']\n",
    "\n",
    "\n",
    "# helper function- to get the confusion matrix\n",
    "def plot_confusionmatrix(y_train_pred,y_train,dom):\n",
    "    print(f'{dom} Confusion matrix')\n",
    "    cf = confusion_matrix(y_train_pred,y_train)\n",
    "    sns.heatmap(cf,annot=True,yticklabels=classes\n",
    "                ,xticklabels=classes,cmap='Blues', fmt='g')\n",
    "    plt.tight_layout()\n",
    "    plt.show()"
   ],
   "metadata": {
    "collapsed": false
   }
  },
  {
   "cell_type": "code",
   "execution_count": null,
   "outputs": [],
   "source": [
    "y_train_pred_knn = knn_new.predict(X_train_pca)\n",
    "y_test_pred_knn = knn_new.predict(X_test_pca)\n",
```

```
    "from sklearn import metrics\n",
    "print(f'Train score {accuracy_score(y_train_pred_knn,y_train)}')\n",
    "print(f'Test score {accuracy_score(y_test_pred_knn,y_test)}')\n",
    "plot_confusionmatrix(y_train_pred_knn,y_train,dom='Train')\n",
    "plot_confusionmatrix(y_test_pred_knn,y_test,dom='Test')"
   ],
   "metadata": {
    "collapsed": false
   }
  },
  {
   "cell_type": "markdown",
   "source": [
    "# Decision Tree"
   ],
   "metadata": {
    "collapsed": false
   }
  },
  {
   "cell_type": "code",
   "execution_count": null,
   "outputs": [],
   "source": [
    "from sklearn.tree import DecisionTreeClassifier\n",
    "\n",
    "dtclassifier = DecisionTreeClassifier(random_state=42) #create a
decision tree classifier instance\n",
    "dtclassifier.fit(X_train_ssc,y_train) #fitting the classifier to the
training data set\n",
    "y_dtc_predic = dtclassifier.predict(X_test_ssc)\n",
    "\n",
    "print (y_dtc_predic[0:5])\n",
    "print (y_test[0:5])"
   ],
   "metadata": {
    "collapsed": false
   }
  },
  {
   "cell_type": "code",
   "execution_count": null,
   "outputs": [],
   "source": [
    "from sklearn.metrics import accuracy_score\n",
    "accuracy_dt = accuracy_score(y_test,y_dtc_predic)\n",
    "accuracy_dt"
   ],
   "metadata": {
    "collapsed": false
   }
  },
  {
   "cell_type": "markdown",
   "source": [
    "# Evaluation of the models"
   ],
```

```
   "metadata": {
    "collapsed": false
   }
  },
  {
   "cell_type": "markdown",
   "source": [
    "Let's check the accuracy of the models"
   ],
   "metadata": {
    "collapsed": false
   }
  },
  {
   "cell_type": "code",
   "execution_count": null,
   "outputs": [],
   "source": [
    "from sklearn import metrics\n",
    "import matplotlib.pyplot as plt\n",
    "print(\"KNN Accuracy: \",
metrics.accuracy_score(y_test,knn_predicted_new))\n",
    "print(\"Decision Tree Accuracy: \",
metrics.accuracy_score(y_test,y_dtc_predic))"
   ],
   "metadata": {
    "collapsed": false
   }
  },
  {
   "cell_type": "code",
   "execution_count": null,
   "outputs": [],
   "source": [
    "# Validation accuracy of the model acreoss the range 1-21\n",
    "train_acc = []\n",
    "valid_acc = []\n",
    "\n",
    "for max_d in range(1,21):\n",
    "    model = DecisionTreeClassifier(max_depth=max_d, random_state=42)\n",
    "    model.fit(X_train_pca, y_train)\n",
    "    print('The Training Accuracy for max_depth {} is:'.format(max_d),
model.score(X_train_pca, y_train))\n",
    "    train_acc.append(model.score(X_train_pca, y_train))\n",
    "    print('The Validation Accuracy for max_depth {} is:'.format(max_d),
model.score(X_test_pca, y_test))\n",
    "    valid_acc.append(model.score(X_test_pca, y_test))\n",
    "    print('')"
   ],
   "metadata": {
    "collapsed": false
   }
  },
  {
   "cell_type": "code",
   "execution_count": null,
   "outputs": [],
```

```
   "source": [
    "max_depth=[range(20)]\n",
    "\n",
    "plt.scatter(max_depth,train_acc)\n",
    "plt.scatter(max_depth,valid_acc)\n",
    "\n",
    "plt.plot(max_depth,train_acc,label = \"Train
accuracy\",drawstyle=\"steps-post\")\n",
    "plt.plot(max_depth,valid_acc,label = \"Validation
accuracy\",drawstyle=\"steps-post\")\n",
    "plt.plot()\n",
    "\n",
    "# plt.xlabel(\"maximum depth\")\n",
    "\n",
    "plt.legend()\n",
    "plt.title(\"Training accuracy and validation accuracy change with
maximum depth\")\n",
    "plt.show()"
   ],
   "metadata": {
    "collapsed": false
   }
  },
  {
   "cell_type": "markdown",
   "source": [
    "roc curve and auc for knn model"
   ],
   "metadata": {
    "collapsed": false
   }
  },
  {
   "cell_type": "code",
   "execution_count": null,
   "outputs": [],
   "source": [
    "from sklearn.metrics import roc_curve,auc\n",
    "import matplotlib.pyplot as plt\n",
    "\n",
    "# get false +ve and true + rate for different threshold vales\n",
    "fpr,tpr,thresholds = roc_curve(y_test,knn_predicted_new)\n",
    "\n",
    "#area under the curve\n",
    "roc_auc = auc(fpr,tpr)\n",
    "\n",
    "# Plot the ROC curve\n",
    "plt.plot(fpr, tpr, label='ROC curve (area = %0.2f)' % roc_auc)\n",
    "plt.plot([0, 1], [0, 1], 'k--')\n",
    "plt.xlim([0.0, 1.0])\n",
    "plt.ylim([0.0, 1.05])\n",
    "plt.xlabel('False Positive Rate')\n",
    "plt.ylabel('True Positive Rate')\n",
    "plt.title('ROC Curve for K- Nearest Neighbor')\n",
    "plt.legend(loc=\"lower right\")\n",
    "plt.show()"
   ],
```

```
   "metadata": {
    "collapsed": false
   }
  },
  {
   "cell_type": "code",
   "execution_count": null,
   "outputs": [],
   "source": [
    "from six import StringIO\n",
    "import matplotlib.image as mpimg\n",
    "from sklearn import tree\n",
    "%matplotlib inline\n",
    "import pydotplus\n",
    "\n",
    "dot_data = StringIO()\n",
    "filename = \"spamtree.png\"\n",
    "\n",
    "featureNames = data.columns[0:57]\n",
    "# targetNames = data[\"Drug\"].unique().tolist()\n",
    "\n",
    "out=tree.export_graphviz(dtclassifier,feature_names=featureNames,\n",
    "                         out_file=dot_data,\n",
    "                         class_names= [\"ham\", \"spam\"],\n",
    "                         filled=True,\n",
    "                         special_characters=True,\n",
    "                         rotate=False)\n",
    "\n",
    "graph = pydotplus.graph_from_dot_data(dot_data.getvalue())\n",
    "graph.write_png(filename)\n",
    "img = mpimg.imread(filename)\n",
    "\n",
    "plt.figure(figsize=(100, 200))\n",
    "plt.imshow(img,interpolation='nearest')"
   ],
   "metadata": {
    "collapsed": false
   }
  }
 ],
 "metadata": {
  "kernelspec": {
   "display_name": "Python 3",
   "language": "python",
   "name": "python3"
  },
  "language_info": {
   "codemirror_mode": {
    "name": "ipython",
    "version": 2
   },
   "file_extension": ".py",
   "mimetype": "text/x-python",
   "name": "python",
   "nbconvert_exporter": "python",
   "pygments_lexer": "ipython2",
   "version": "2.7.6"
```

```
  }
 },
 "nbformat": 4,
 "nbformat_minor": 0
}
```

Decision tree- python notebook

```
   "cell_type": "code",
   "execution_count": null,
   "outputs": [],
   "source": [
    "from sklearn.model_selection import train_test_split\n",
    "\n",
    "x_train,x_test,y_train,y_test = train_test_split(X,y,stratify=y)\n",
    "print(x_train.shape)\n",
    "print(x_test.shape)"
   ],
   "metadata": {
    "collapsed": false
   }
  },
  {
   "cell_type": "markdown",
   "source": [
    "First fitting the normal decision tree without fine tuning and check the
results"
   ],
   "metadata": {
    "collapsed": false
   }
  },
  {
   "cell_type": "code",
   "execution_count": null,
   "outputs": [],
   "source": [
    "from sklearn.tree import DecisionTreeClassifier\n",
    "dtclf = DecisionTreeClassifier(random_state=0)\n",
    "dtclf.fit(x_train,y_train)\n",
    "y_train_pred = dtclf.predict(x_train)\n",
    "y_test_pred = dtclf.predict(x_test)"
   ],
   "metadata": {
    "collapsed": false
   }
  },
  {
   "cell_type": "markdown",
   "source": [
    "Visualizing the decision tree"
```

```
   ],
   "metadata": {
    "collapsed": false
   }
  },
  {
   "cell_type": "code",
   "execution_count": null,
   "outputs": [],
   "source": [
    "from sklearn import tree\n",
    "\n",
    "plt.figure(figsize=(20,20))\n",
    "features = data.columns\n",
    "classes = ['Not spam','spam']\n",

"tree.plot_tree(dtclf,feature_names=features,class_names=classes,filled=True)
\n",
    "plt.show()"
   ],
   "metadata": {
    "collapsed": false
   }
  },
  {
   "cell_type": "code",
   "execution_count": null,
   "outputs": [],
   "source": [
    "from sklearn.metrics import roc_curve,auc\n",
    "import matplotlib.pyplot as plt\n",
    "from sklearn.tree import DecisionTreeClassifier\n",
    "\n",
    "# get false +ve and true + rate for different threshold vales\n",
    "fpr,tpr,thresholds = roc_curve(y_test,y_test_pred)\n",
    "\n",
    "#area under the curve\n",
    "roc_auc = auc(fpr,tpr)\n",
    "\n",
    "# Plot the ROC curve\n",
    "plt.plot(fpr, tpr, label='ROC curve (area = %0.2f)' % roc_auc)\n",
    "plt.plot([0, 1], [0, 1], 'k--')\n",
    "plt.xlim([0.0, 1.0])\n",
    "plt.ylim([0.0, 1.05])\n",
    "plt.xlabel('False Positive Rate')\n",
    "plt.ylabel('True Positive Rate')\n",
    "plt.title('ROC Curve for Decision Tree Classifier')\n",
    "plt.legend(loc=\"lower right\")\n",
    "plt.show()"
   ],
   "metadata": {
    "collapsed": false
   }
  },
  {
   "cell_type": "code",
   "execution_count": null,
```

```
  "outputs": [],
  "source": [
   "from sklearn.metrics import confusion_matrix\n",
   "import seaborn as sns\n",
   "\n",
   "# helper function- to get the confusion matrix\n",
   "def plot_confusionmatrix(y_train_pred,y_train,dom):\n",
   "    print(f'{dom} Confusion matrix')\n",
   "    cf = confusion_matrix(y_train_pred,y_train)\n",
   "    sns.heatmap(cf,annot=True,yticklabels=classes\n",
   "                ,xticklabels=classes,cmap='Blues', fmt='g')\n",
   "    plt.tight_layout()\n",
   "    plt.show()"
  ],
  "metadata": {
   "collapsed": false
  }
 },
 {
  "cell_type": "code",
  "execution_count": null,
  "outputs": [],
  "source": [
   "from sklearn.metrics import accuracy_score\n",
   "\n",
   "print(f'Train score {accuracy_score(y_train_pred,y_train)}')\n",
   "print(f'Test score {accuracy_score(y_test_pred,y_test)}')\n",
   "plot_confusionmatrix(y_train_pred,y_train,dom='Train')\n",
   "plot_confusionmatrix(y_test_pred,y_test,dom='Test')"
  ],
  "metadata": {
   "collapsed": false
  }
 },
 {
  "cell_type": "code",
  "execution_count": null,
  "outputs": [],
  "source": [
   "print(confusion_matrix(y_test_pred,y_test))"
  ],
  "metadata": {
   "collapsed": false
  }
 },
 {
  "cell_type": "markdown",
  "source": [
   "# Pre pruning"
  ],
  "metadata": {
   "collapsed": false
  }
 },
 {
  "cell_type": "markdown",
  "source": [
```

```
    "Here we stop the growing of the tree at an early stage by setting
constraints\n",
    "The grid search through parameters is done and the optimum values are
chosen"
   ],
   "metadata": {
    "collapsed": false
   }
  },
  {
   "cell_type": "markdown",
   "source": [
    "Here following parameters are controled\n",
    "- maximum depth of the tree\n",
    "- minimum number of samples needed to split an interval node\n",
    "- minimum number of samples needed to be a leaf node"
   ],
   "metadata": {
    "collapsed": false
   }
  },
  {
   "cell_type": "code",
   "execution_count": null,
   "outputs": [],
   "source": [
    "from sklearn.model_selection import GridSearchCV\n",
    "\n",
    "params = {'max_depth': [2,4,6,8,10,12],\n",
    "           'min_samples_split': [2,3,4],\n",
    "           'min_samples_leaf': [1,2]}\n",
    "\n",
    "dtclf = DecisionTreeClassifier()\n",
    "gcv = GridSearchCV(estimator=dtclf,param_grid=params)\n",
    "gcv.fit(x_train,y_train)"
   ],
   "metadata": {
    "collapsed": false
   }
  },
  {
   "cell_type": "code",
   "execution_count": null,
   "outputs": [],
   "source": [
    "model_dtc = gcv.best_estimator_\n",
    "model_dtc.fit(x_train,y_train)\n",
    "\n",
    "y_train_pred = model_dtc.predict(x_train)\n",
    "y_test_pred = model_dtc.predict(x_test)\n",
    "\n",
    "print(f'Train score {accuracy_score(y_train_pred,y_train)}')\n",
    "print(f'Test score {accuracy_score(y_test_pred,y_test)}')\n",
    "plot_confusionmatrix(y_train_pred,y_train,dom='Train')\n",
    "plot_confusionmatrix(y_test_pred,y_test,dom='Test')"
   ],
   "metadata": {
```

```
    "collapsed": false
   }
  },
  {
   "cell_type": "code",
   "execution_count": null,
   "outputs": [],
   "source": [
    "print(confusion_matrix(y_test_pred,y_test))"
   ],
   "metadata": {
    "collapsed": false
   }
  },
  {
   "cell_type": "code",
   "execution_count": null,
   "outputs": [],
   "source": [
    "plt.figure(figsize=(20,20))\n",
    "features = data.columns\n",
    "classes = ['No spam','spam']\n",

"tree.plot_tree(model_dtc,feature_names=features,class_names=classes,filled=T
rue)\n",
    "plt.show()"
   ],
   "metadata": {
    "collapsed": false
   }
  },
  {
   "cell_type": "markdown",
   "source": [
    "After pruning there is an improvement in test accuracy."
   ],
   "metadata": {
    "collapsed": false
   }
  },
  {
   "cell_type": "markdown",
   "source": [
    "# Post pruning\n",
    "For further improvements let's do cost complexity pruning as a post
pruning technique to avoid overfitting as decison trees are more likely to
get overfitted.\n",
    "\n",
    "## Cost Complexity pruning"
   ],
   "metadata": {
    "collapsed": false
   }
  },
  {
   "cell_type": "code",
   "execution_count": null,
```

```
   "outputs": [],
   "source": [
    "path = dtclf.cost_complexity_pruning_path(x_train, y_train)\n",
    "ccp_alphas, impurities = path.ccp_alphas, path.impurities\n",
    "print(ccp_alphas)"
   ],
   "metadata": {
    "collapsed": false
   }
  },
  {
   "cell_type": "code",
   "execution_count": null,
   "outputs": [],
   "source": [
    "plt.scatter(ccp_alphas,impurities)\n",
    "    plt.plot()\n",
    "    plt.plot(ccp_alphas,impurities,drawstyle=\"steps-post\")\n",
    "    plt.xlabel(\"cost effective alpha values\")\n",
    "    plt.ylabel(\"total leaf impurity\")\n",
    "    plt.legend()\n",
    "    plt.title(\"Total impurity of the leaves vs cost effective alpha-
training set\")\n",
    "plt.show()"
   ],
   "metadata": {
    "collapsed": false
   }
  },
  {
   "cell_type": "code",
   "execution_count": null,
   "outputs": [],
   "source": [
    "# For each alpha the model is appended to a list\n",
    "model_list = []\n",
    "for ccp_alpha in ccp_alphas:\n",
    "    clf = tree.DecisionTreeClassifier(random_state=0,
ccp_alpha=ccp_alpha)\n",
    "    clf.fit(x_train, y_train)\n",
    "    model_list.append(clf)"
   ],
   "metadata": {
    "collapsed": false
   }
  },
  {
   "cell_type": "markdown",
   "source": [
    "The last element in models and alpha values are removed as it is a
trivial tree with a single node"
   ],
   "metadata": {
    "collapsed": false
   }
  },
  {
```

```
   "cell_type": "code",
   "execution_count": null,
   "outputs": [],
   "source": [
    "model_list = model_list[:-1]\n",
    "ccp_alphas = ccp_alphas[:-1]\n",
    "\n",
    "dtclf = DecisionTreeClassifier()\n",
    "dtclf.fit(x_train,y_train)\n",
    "\n",
    "tree = dtclf.tree_\n",
    "\n",
    "node_counts = [dtclf.tree_.node_count for model in model_list]\n",
    "depth = [dtclf.tree_.max_depth for model in model_list]\n",
    "\n",
    "plt.scatter(ccp_alphas,node_counts)\n",
    "plt.scatter(ccp_alphas,depth)\n",
    "plt.plot(ccp_alphas,node_counts,label='no of nodes')\n",
    "plt.plot(ccp_alphas,depth,label='depth')\n",
    "plt.legend()\n",
    "plt.show()"
   ],
   "metadata": {
    "collapsed": false
   }
  },
  {
   "cell_type": "markdown",
   "source": [
    "Here the observations are not clear enough"
   ],
   "metadata": {
    "collapsed": false
   }
  },
  {
   "cell_type": "code",
   "execution_count": null,
   "outputs": [],
   "source": [
    "train_acc = []\n",
    "test_acc = []\n",
    "for c in model_list:\n",
    "    y_train_pred = c.predict(x_train)\n",
    "    y_test_pred = c.predict(x_test)\n",
    "    train_acc.append(accuracy_score(y_train_pred,y_train))\n",
    "    test_acc.append(accuracy_score(y_test_pred,y_test))\n",
    "\n",
    "plt.scatter(ccp_alphas,train_acc)\n",
    "plt.scatter(ccp_alphas,test_acc)\n",
    "plt.plot(ccp_alphas,train_acc,label='train_accuracy',drawstyle=\"steps-
post\")\n",
    "plt.plot(ccp_alphas,test_acc,label='test_accuracy',drawstyle=\"steps-
post\")\n",
    "plt.legend()\n",
    "plt.title('Accuracy vs alpha')\n",
    "plt.show()"
```

```
   ],
   "metadata": {
    "collapsed": false
   }
  },
  {
   "cell_type": "markdown",
   "source": [
    "We can choose alpha as = 0.01"
   ],
   "metadata": {
    "collapsed": false
   }
  },
  {
   "cell_type": "code",
   "execution_count": null,
   "outputs": [],
   "source": [
    "\n",
    "clf_ = DecisionTreeClassifier(random_state=42,ccp_alpha=0.01)\n",
    "clf_.fit(x_train,y_train)\n",
    "y_train_pred = clf_.predict(x_train)\n",
    "y_test_pred = clf_.predict(x_test)\n",
    "\n",
    "print(f'Train score {accuracy_score(y_train_pred,y_train)}')\n",
    "print(f'Test score {accuracy_score(y_test_pred,y_test)}')\n",
    "plot_confusionmatrix(y_train_pred,y_train,dom='Train')\n",
    "plot_confusionmatrix(y_test_pred,y_test,dom='Test')"
   ],
   "metadata": {
    "collapsed": false
   }
  },
  {
   "cell_type": "markdown",
   "source": [
    "overfitting is not happening and the performance on the test data have
improved"
   ],
   "metadata": {
    "collapsed": false
   }
  },
  {
   "cell_type": "code",
   "execution_count": null,
   "outputs": [],
   "source": [
    "data.head()"
   ],
   "metadata": {
    "collapsed": false
   }
  },
  {
   "cell_type": "code",
```

```
   "execution_count": null,
   "outputs": [],
   "source": [
    "features = data.columns[:-1]\n",
    "classes = ['No spam','Spam']\n",
    "\n",
    "from sklearn.tree import export_graphviz\n",
    "import graphviz\n",
    "dtclf = DecisionTreeClassifier()\n",
    "dtclf.fit(x_train, y_train)\n",
    "\n",
    "dot_data = export_graphviz(dtclf, out_file=None, feature_names=features,
class_names=classes)\n",
    "graph = graphviz.Source(dot_data)\n",
    "graph.format = 'jpg' # set the output format to JPG\n",
    "graph.render(\"Spam- non spam decision tree\") # creates a PDF file with
the visualization"
   ],
   "metadata": {
    "collapsed": false
   }
  },
  {
   "cell_type": "code",
   "execution_count": null,
   "outputs": [],
   "source": [
    "from sklearn.metrics import classification_report, confusion_matrix\n",
    "\n",
    "predictions=dtclf.predict(x_test)\n",
    "\n",
    "print(classification_report(y_test, predictions))\n",
    "print(confusion_matrix(y_test, predictions))"
   ],
   "metadata": {
    "collapsed": false
   }
  },
  {
   "cell_type": "markdown",
   "source": [
    "Here the size of the tree has reduced"
   ],
   "metadata": {
    "collapsed": false
   }
  },
  {
   "cell_type": "code",
   "execution_count": null,
   "outputs": [],
   "source": [
    "from sklearn.metrics import precision_score\n",
    "from sklearn import metrics\n",
    "\n",
    "accuracy_dtc =metrics.accuracy_score(y_test,predictions)\n",
    "prec_dtc = precision_score(y_test, predictions)\n",
```

```
    "Sensitivity_recall_dtc = metrics.recall_score(y_test, predictions) #
Sensitivity- how well the model predicts something is positive\n",
    "Specificity_dtc = metrics.recall_score(y_test, predictions) #
Specificity- how well the model predicts something is negative\n",
    "F_score_dtc = metrics.f1_score(y_test,predictions)\n",
    "\n",
    "print(\"accuracy of Decision Tree  model: \",accuracy_dtc)\n",
    "print(\"Precision of Decision Tree  model: \",prec_dtc)\n",
    "print(\"Sensitivity of Decision Tree  model:
\",Sensitivity_recall_dtc)\n",
    "print(\"Specificity of Decision Tree model: \",Specificity_dtc)\n",
    "print(\"F1 score of Decision Tree  model: \",F_score_dtc)"
   ],
   "metadata": {
    "collapsed": false
   }
  },
  {
   "cell_type": "code",
   "execution_count": null,
   "outputs": [],
   "source": [],
   "metadata": {
    "collapsed": false
   }
  }
 ],
 "metadata": {
  "kernelspec": {
   "display_name": "Python 3",
   "language": "python",
   "name": "python3"
  },
  "language_info": {
   "codemirror_mode": {
    "name": "ipython",
    "version": 2
   },
   "file_extension": ".py",
   "mimetype": "text/x-python",
   "name": "python",
   "nbconvert_exporter": "python",
   "pygments_lexer": "ipython2",
   "version": "2.7.6"
  }
 },
 "nbformat": 4,
 "nbformat_minor": 0
}
```