ARTIFICIAL INTELLIGENCE

LAB PROJECT

Prepared for

Prof. Aneesh Chivukula

Department of Computer Science and Information Systems

‘

Prepared by

Kota Venkata Bhargav	2020B2A82088G
Rohit Akash R	2020B1A32073G
Sahil Sudesh	2020B4A31868G

BUSINESS UNDERSTANDING

PROBLEM STATEMENT

The business objective is to develop a credit default prediction model to assess the risk of customers defaulting on their credit card payments. By accurately identifying high-risk customers, the goal is to minimize financial losses and improve decision-making in credit card lending.

DATA UNDERSTANDING

DATA DESCRIPTION

The objective of this is to predict the probability that a customer does not pay back their credit card balance amount in the future based on their monthly customer profile. The target binary variable is calculated by observing 18 months performance window after the latest credit card statement, and if the customer does not pay due amount in 120 days after their latest statement date it is considered a default event.

The dataset contains aggregated profile features for each customer at each statement date. Features are anonymized and normalized, and fall into the following general categories:

* D_* = Delinquency variables

* S_* = Spend variables

* P_* = Payment variables

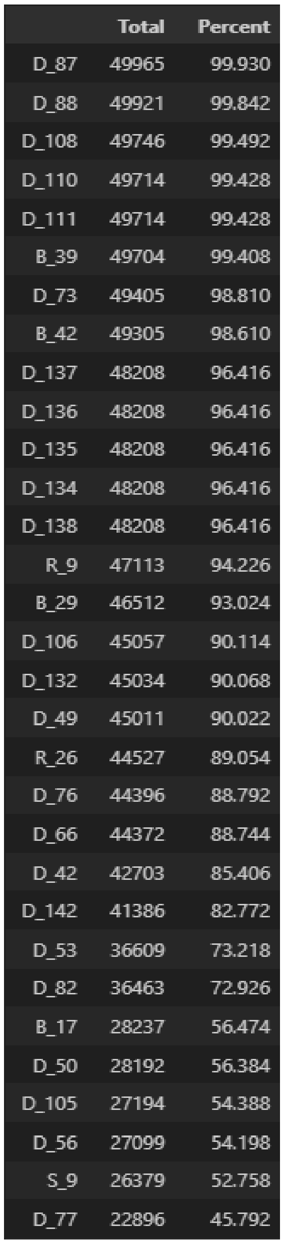
* B_* = Balance variables

* R_* = Risk variables

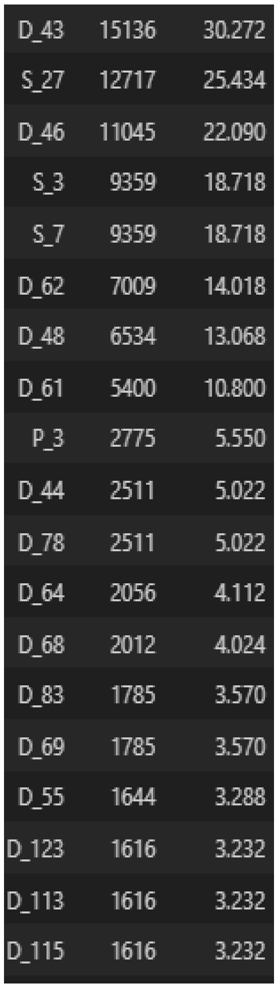
DEALING WITH MISSING VALUES

Since the dataset is too large, we will engage in dropping a few columns if the percentage of missing values there is higher than a specified percentage. We will then apply some methods like mean, median, mode imputation etc.

First we check the percentage of missing values in each column.

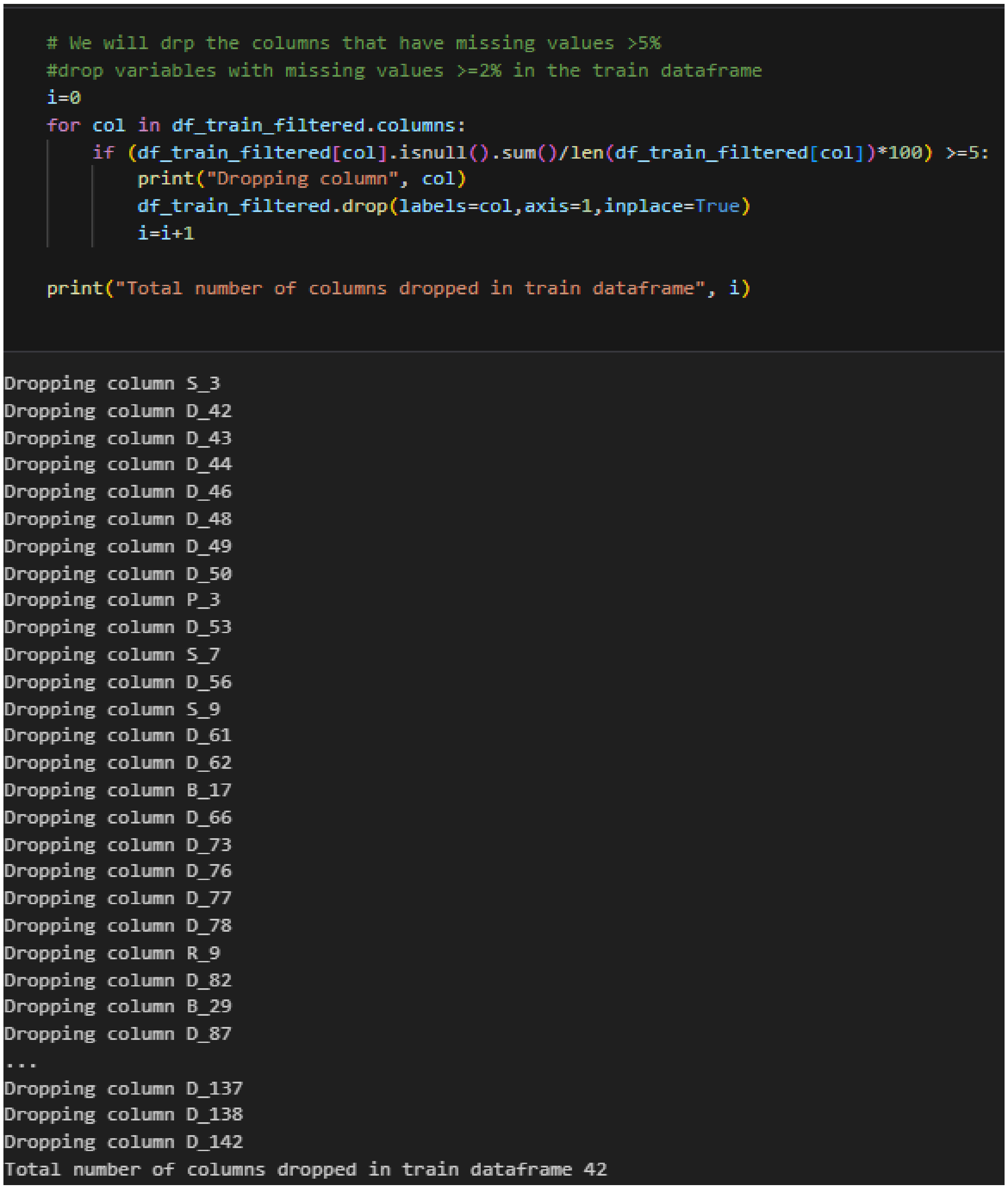


	Total	Percent
D_87	49965	99.930
D_88	49921	99.842
D_108	49746	99.492
D_110	49714	99.428
D_111	49714	99.428
B_39	49704	99.408
D_73	49405	98.810
B_42	49305	98.610
D_137	48208	96.416
D_136	48208	96.416
D_135	48208	96.416
D_134	48208	96.416
D_138	48208	96.416
R_9	47113	94.226
B_29	46512	93.024
D_106	45057	90.114
D_132	45034	90.068
D_49	45011	90.022
R_26	44527	89.054
D_76	44396	88.792
D_66	44372	88.744
D_42	42703	85.406
D_142	41386	82.772
D_53	36609	73.218
D_82	36463	72.926
B_17	28237	56.474
D_50	28192	56.384
D_105	27194	54.388
D_56	27099	54.198
S_9	26379	52.758
D_77	22896	45.792



D_43	15136	30.272
S_27	12717	25.434
D_46	11045	22.090
S_3	9359	18.718
S_7	9359	18.718
D_62	7009	14.018
D_48	6534	13.068
D_61	5400	10.800
P_3	2775	5.550
D_44	2511	5.022
D_78	2511	5.022
D_64	2056	4.112
D_68	2012	4.024
D_83	1785	3.570
D_69	1785	3.570
D_55	1644	3.288
D_123	1616	3.232
D_113	1616	3.232
D_115	1616	3.232

We will drop the column that has missing values >5%



```
# We will drp the columns that have missing values >5%
#drop variables with missing values >=2% in the train dataframe
i=0
for col in df_train_filtered.columns:
    if (df_train_filtered[col].isnull().sum()/len(df_train_filtered[col])*100) >=5:
        print("Dropping column", col)
        df_train_filtered.drop(labels=col,axis=1,inplace=True)
        i=i+1

print("Total number of columns dropped in train dataframe", i)
```

```
Dropping column S_3
Dropping column D_42
Dropping column D_43
Dropping column D_44
Dropping column D_46
Dropping column D_48
Dropping column D_49
Dropping column D_50
Dropping column P_3
Dropping column D_53
Dropping column S_7
Dropping column D_56
Dropping column S_9
Dropping column D_61
Dropping column D_62
Dropping column B_17
Dropping column D_66
Dropping column D_73
Dropping column D_76
Dropping column D_77
Dropping column D_78
Dropping column R_9
Dropping column D_82
Dropping column B_29
Dropping column D_87
...
Dropping column D_137
Dropping column D_138
Dropping column D_142
Total number of columns dropped in train dataframe 42
```

Similarly, for testing data the percentage of missing values is found and the corresponding columns are dropped.

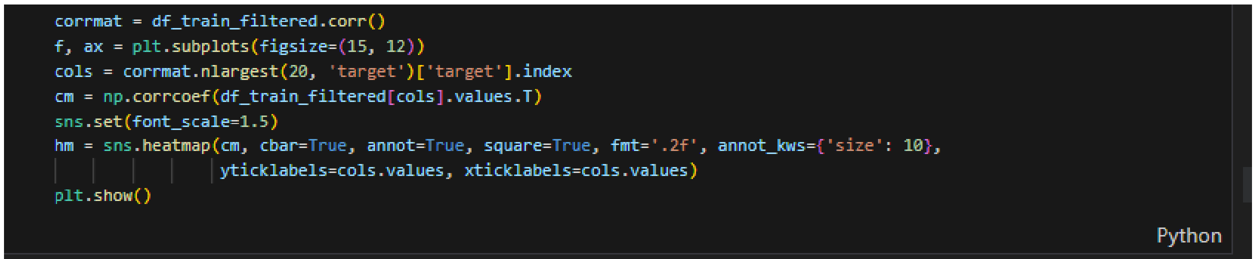
Iterative Imputer for filling missing values

Iterative Imputer, also known as Multivariate Imputation by Chained Equations (MICE), is a technique used to impute missing values in a dataset by iteratively estimating the missing values based on observed values and relationships within the data.

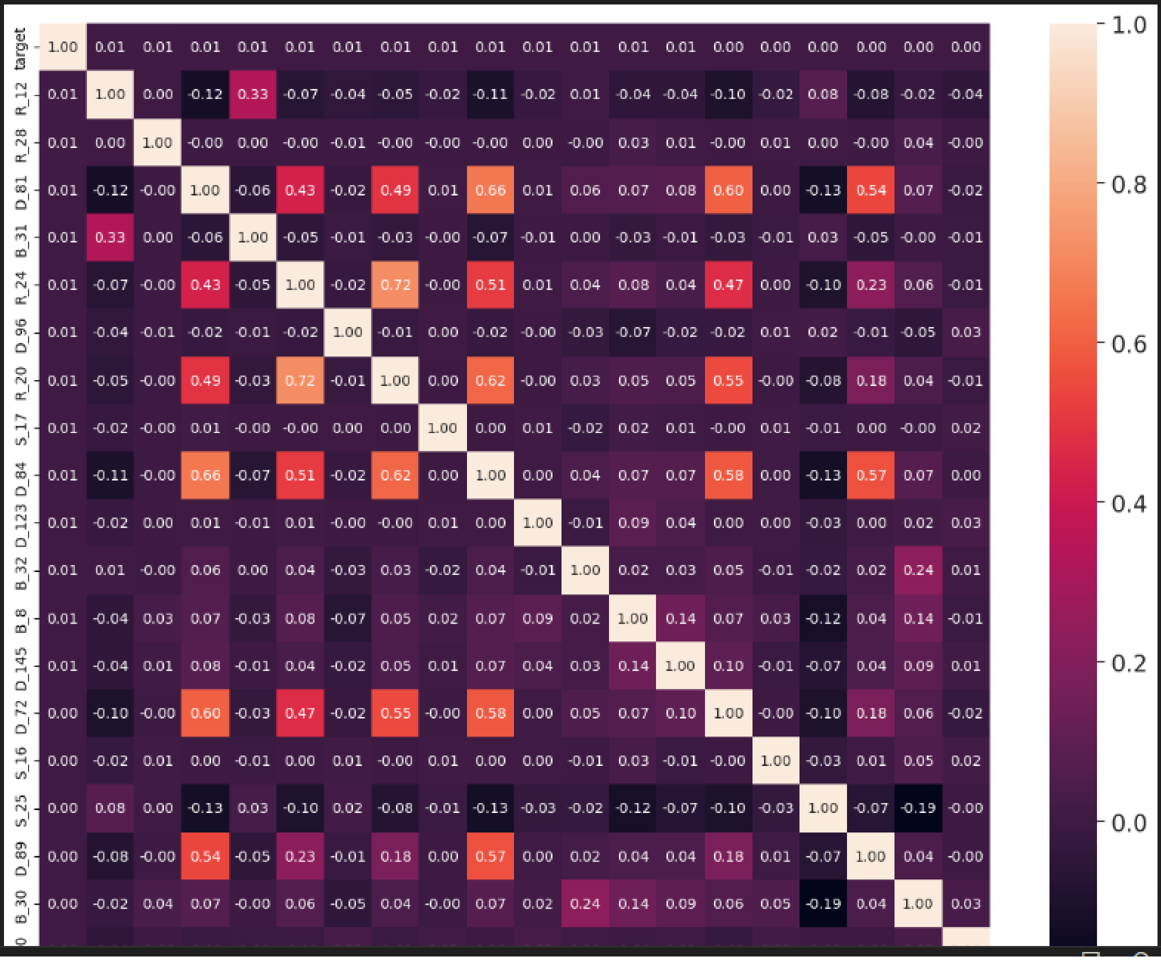
- Initialization: The iterative imputation process starts by initializing the missing values in the dataset with some initial estimates. The most common approach is to use simple imputation methods like mean, median, or mode to fill in the missing values.
- Model Fitting: Once the initial estimates are in place, a predictive model is chosen, such as a linear regression model, a decision tree, or a random forest, depending on the nature of the data. The model is then fitted using the observed values, treating the column with missing values as the target variable and the remaining columns as input features.
- Value Estimation: After fitting the model, the missing values in the target variable column are estimated using the learned model. The estimated values replace the missing values in that column.
- Chained Imputation: The process is repeated for each column with missing values, one column at a time. In each iteration, the column with missing values is considered as the target variable, and the remaining columns (including those that have already been imputed) are used as input features for the model. The model is fitted and the missing values in the target variable column are estimated.
- Convergence: The iterations continue until convergence is achieved, which typically means that the imputed values stabilize or the maximum number of iterations is reached. At convergence, the missing values in the dataset have been imputed using multiple iterations, taking into account the relationships between the variables.

DATA VISUALISATION

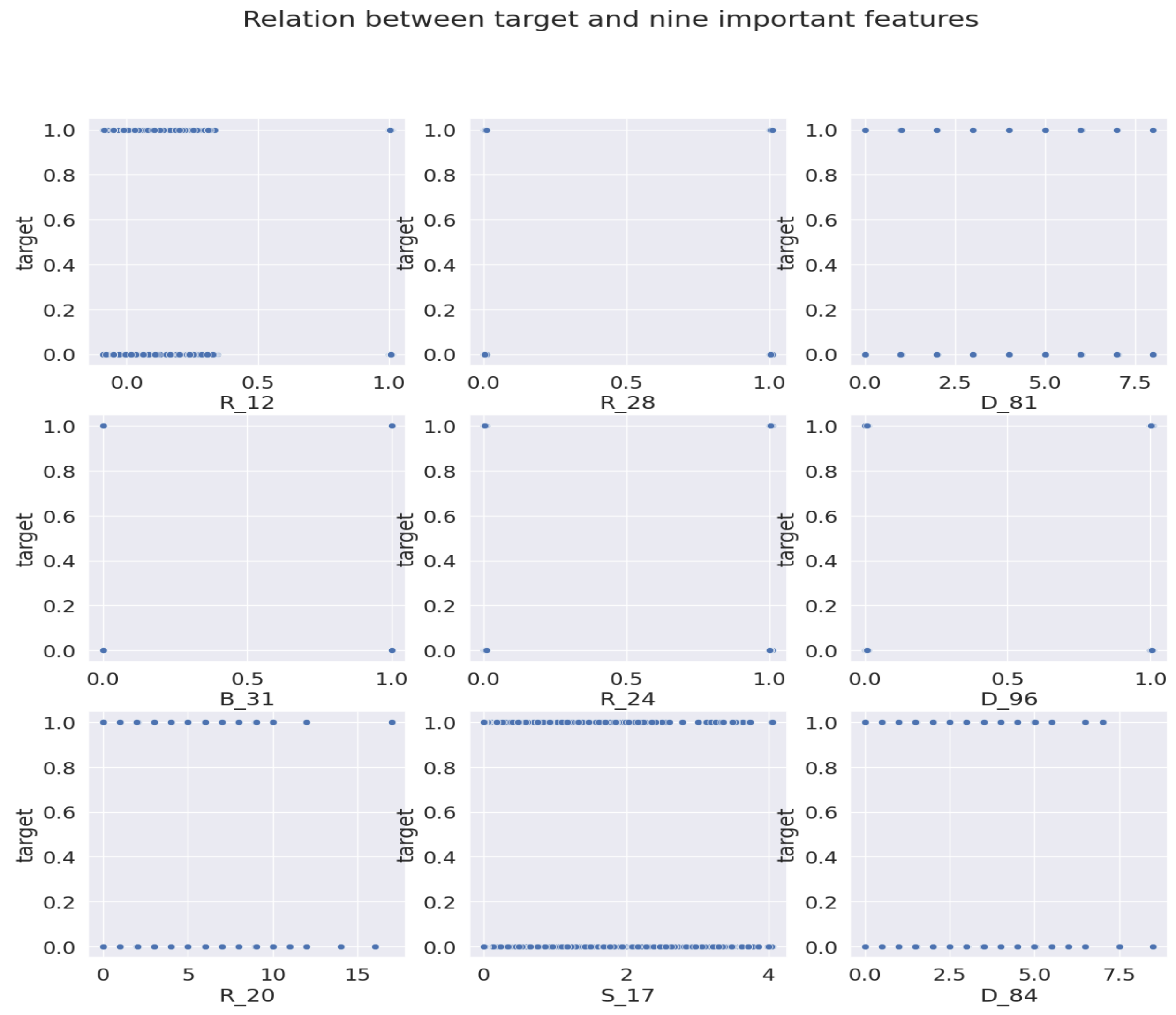
We first draw a heatmap to do smt



```python
corrmat = df_train_filtered.corr()
f, ax = plt.subplots(figsize=(15, 12))
cols = corrmat.nlargest(20, 'target')['target'].index
cm = np.corrcoef(df_train_filtered[cols].values.T)
sns.set(font_scale=1.5)
hm = sns.heatmap(cm, cbar=True, annot=True, square=True, fmt='.2f', annot_kws={'size': 10},
                 yticklabels=cols.values, xticklabels=cols.values)
plt.show()
```



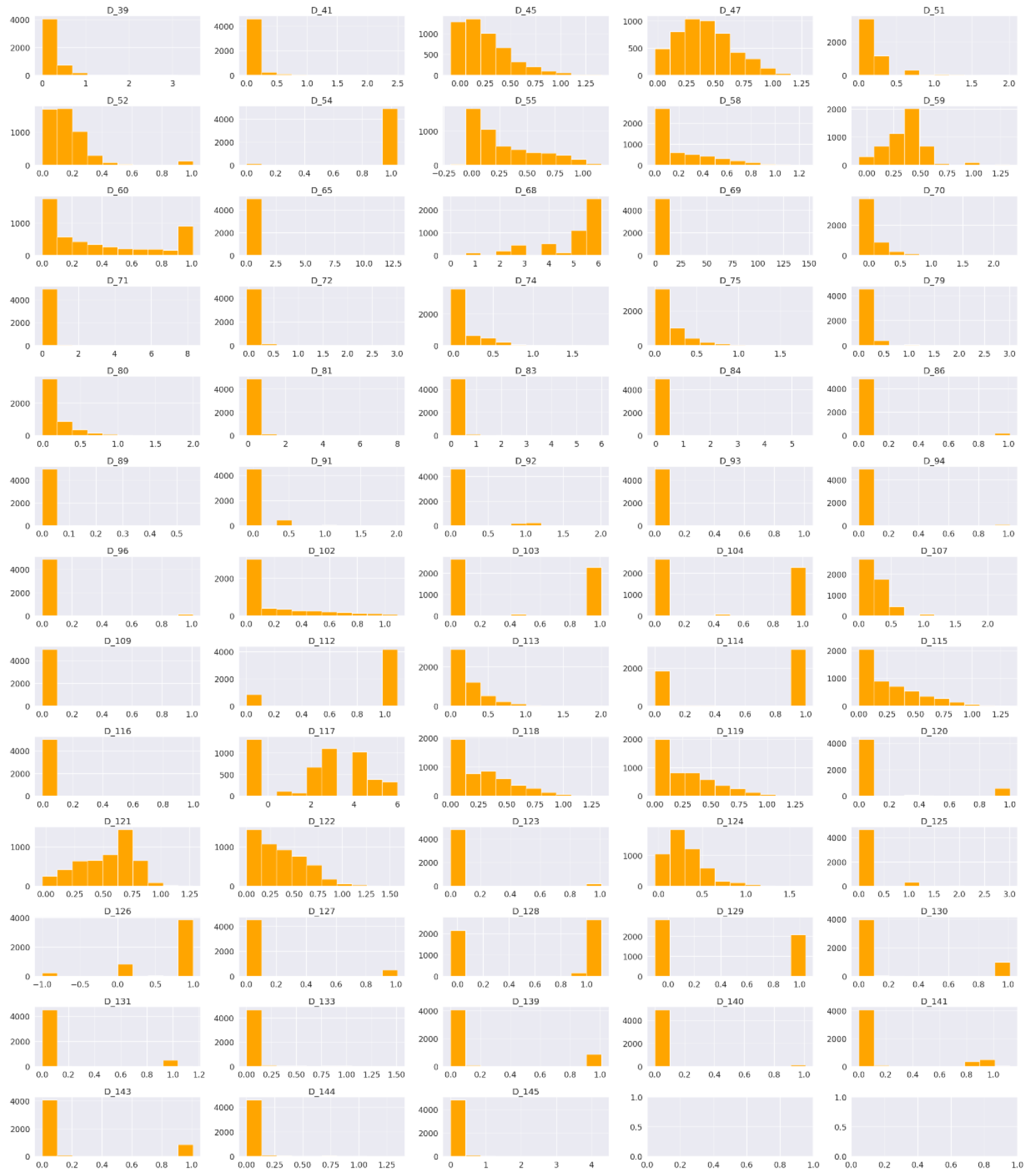
In the above plot, we observe an imbalance between the two classes.



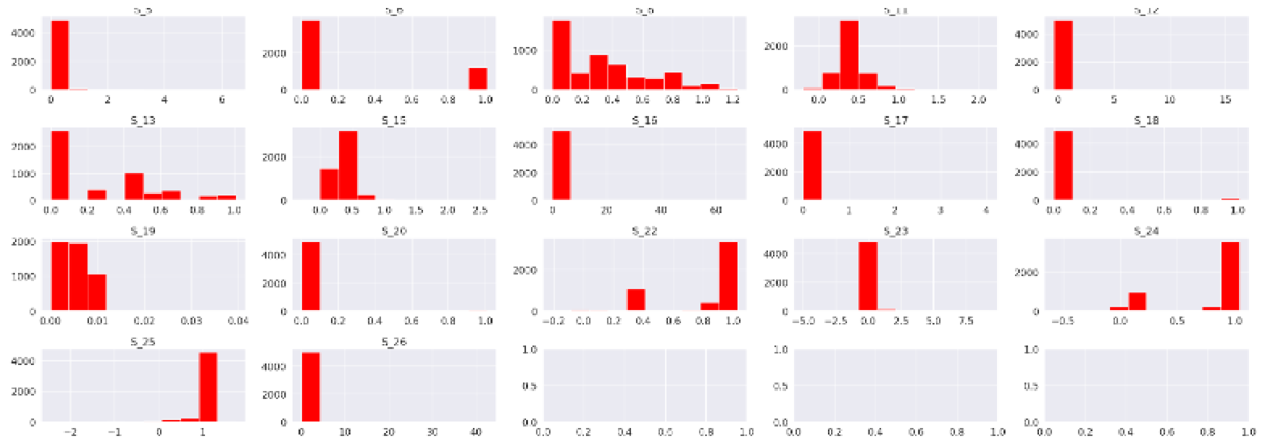
An outlier is an observation that significantly deviates from the other observations in a dataset. It is a data point that falls outside the expected range or pattern of the majority of the data. Outliers can occur due to various reasons, including measurement errors, data entry mistakes, natural variations in the data, or rare events.

Outliers can have a significant impact on statistical analyses, data modeling, and machine learning algorithms. They can distort summary statistics, affect the estimation of parameters, and influence the overall results and interpretations of analyses. Therefore, it is important to identify and properly handle outliers in data analysis.

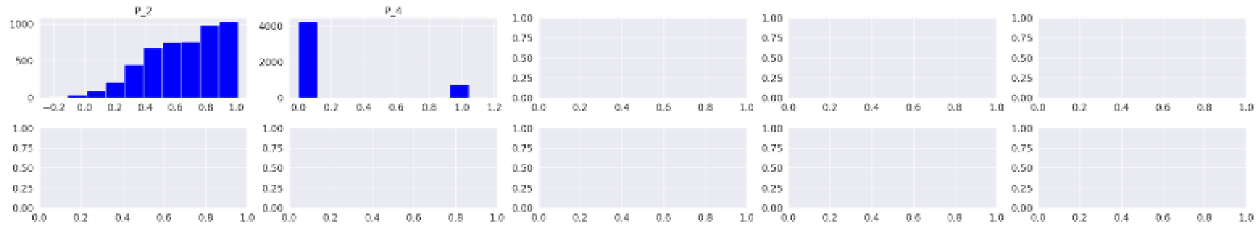
Distribution of D variables



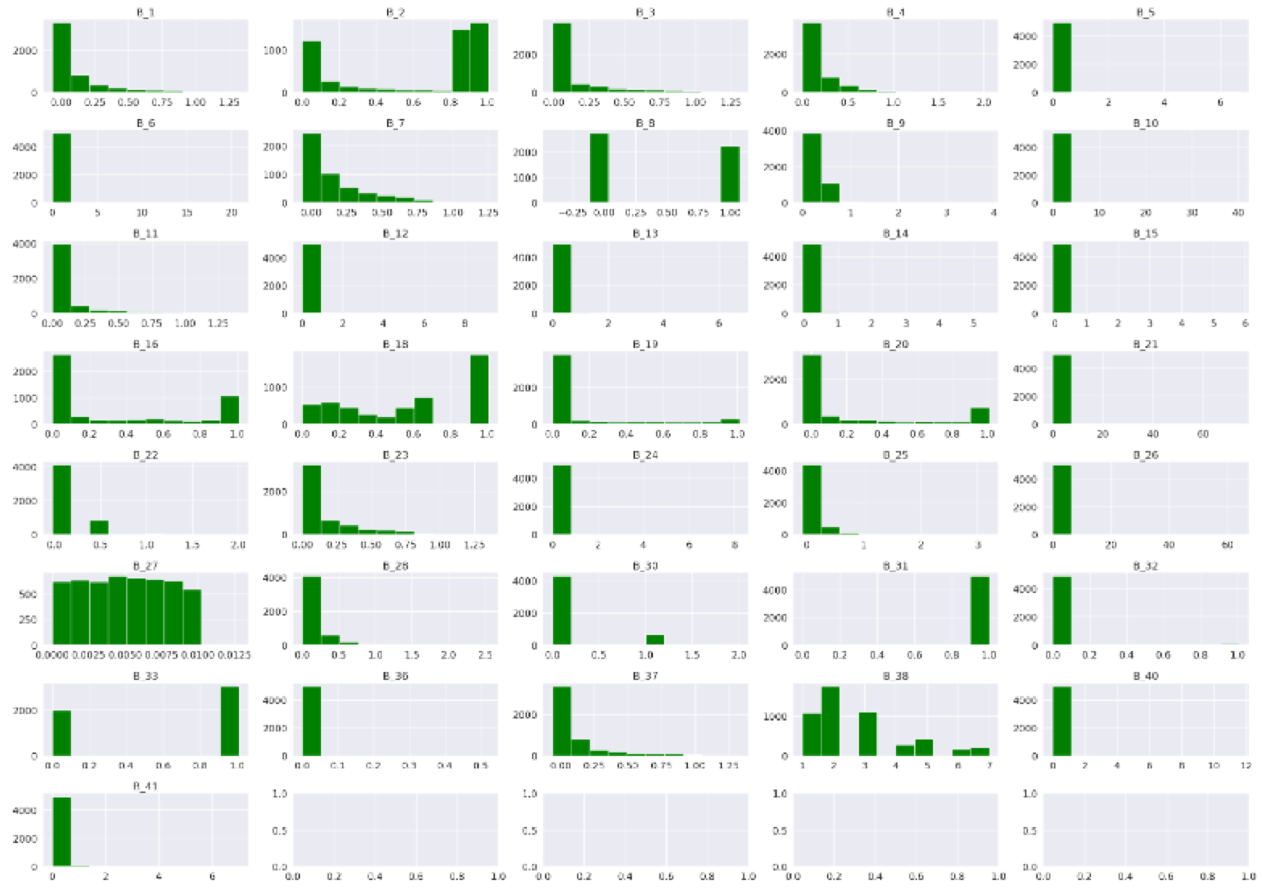
Distribution for S variables



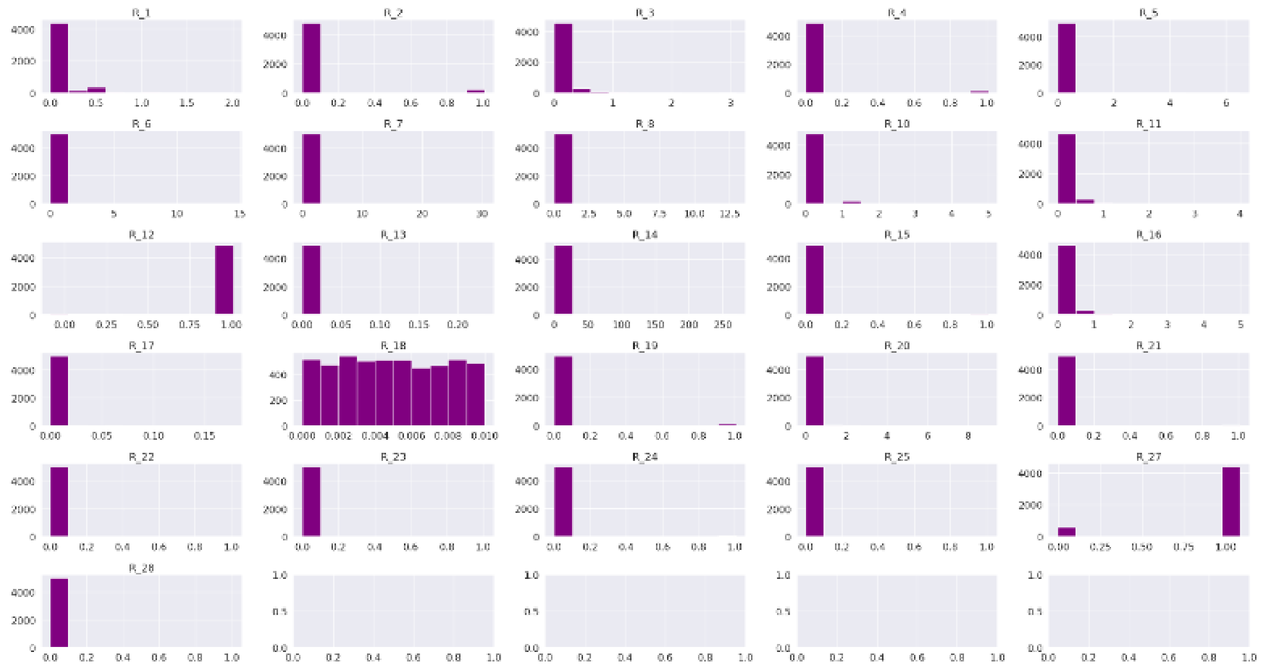
Distribution for S variables



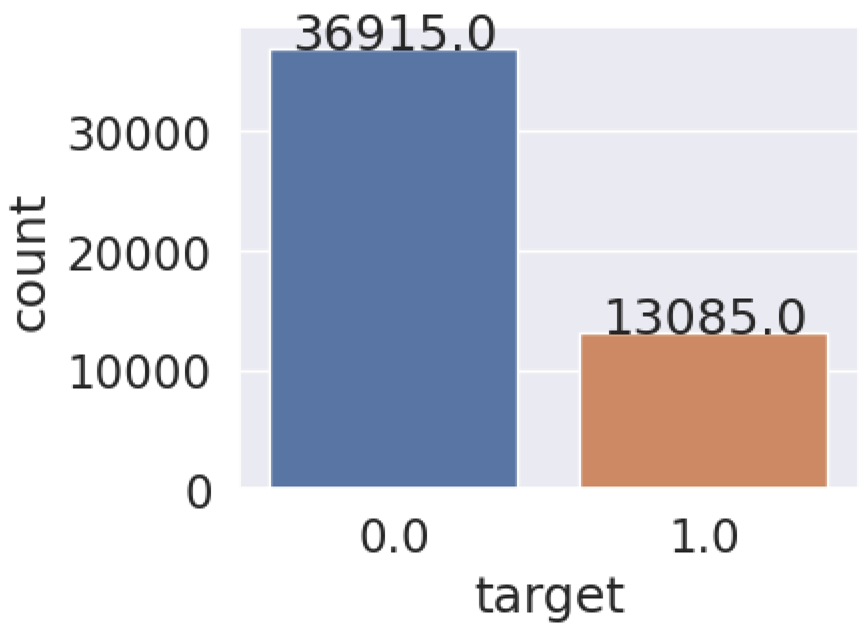
Distribution for B variables



Distribution for B variables

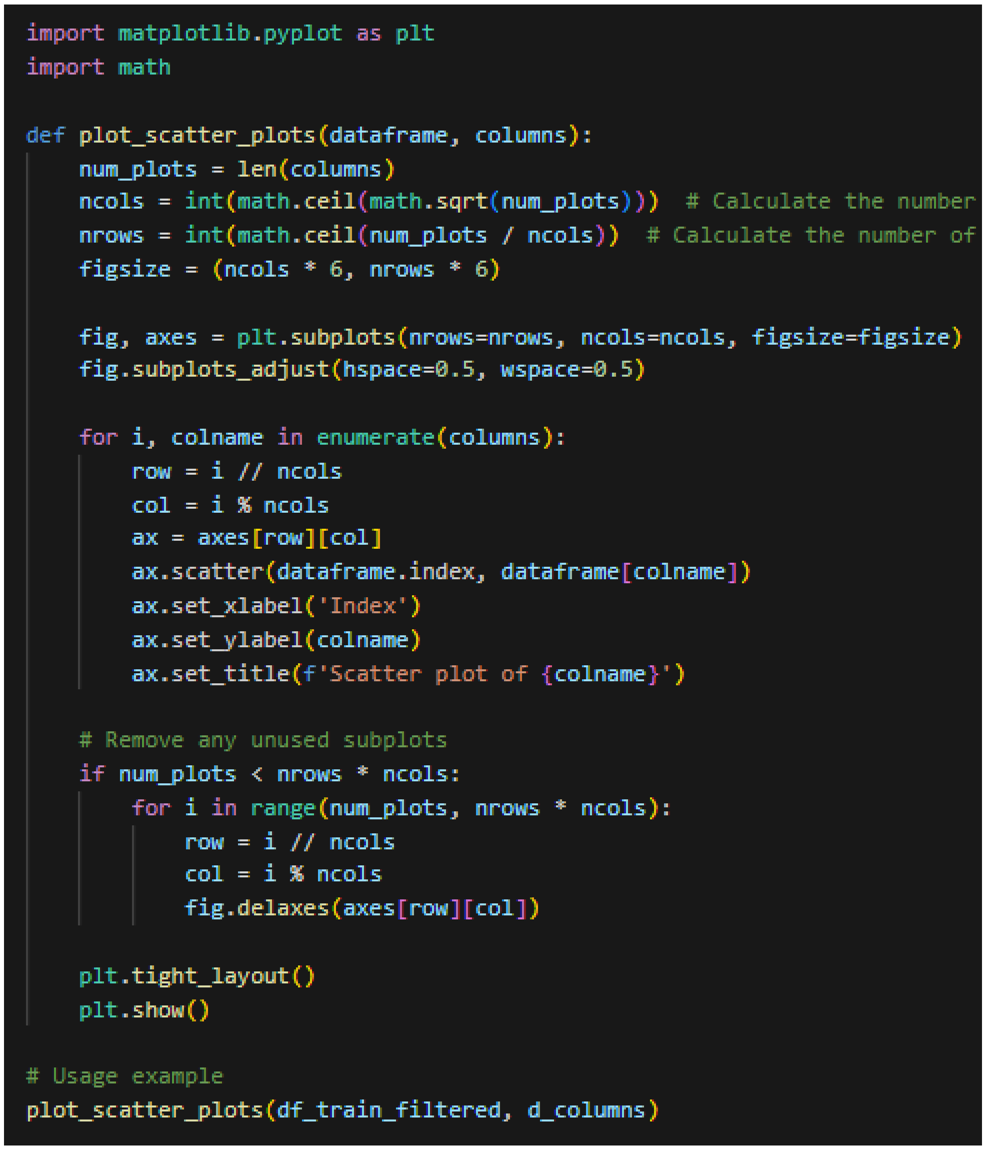


Target attribute count

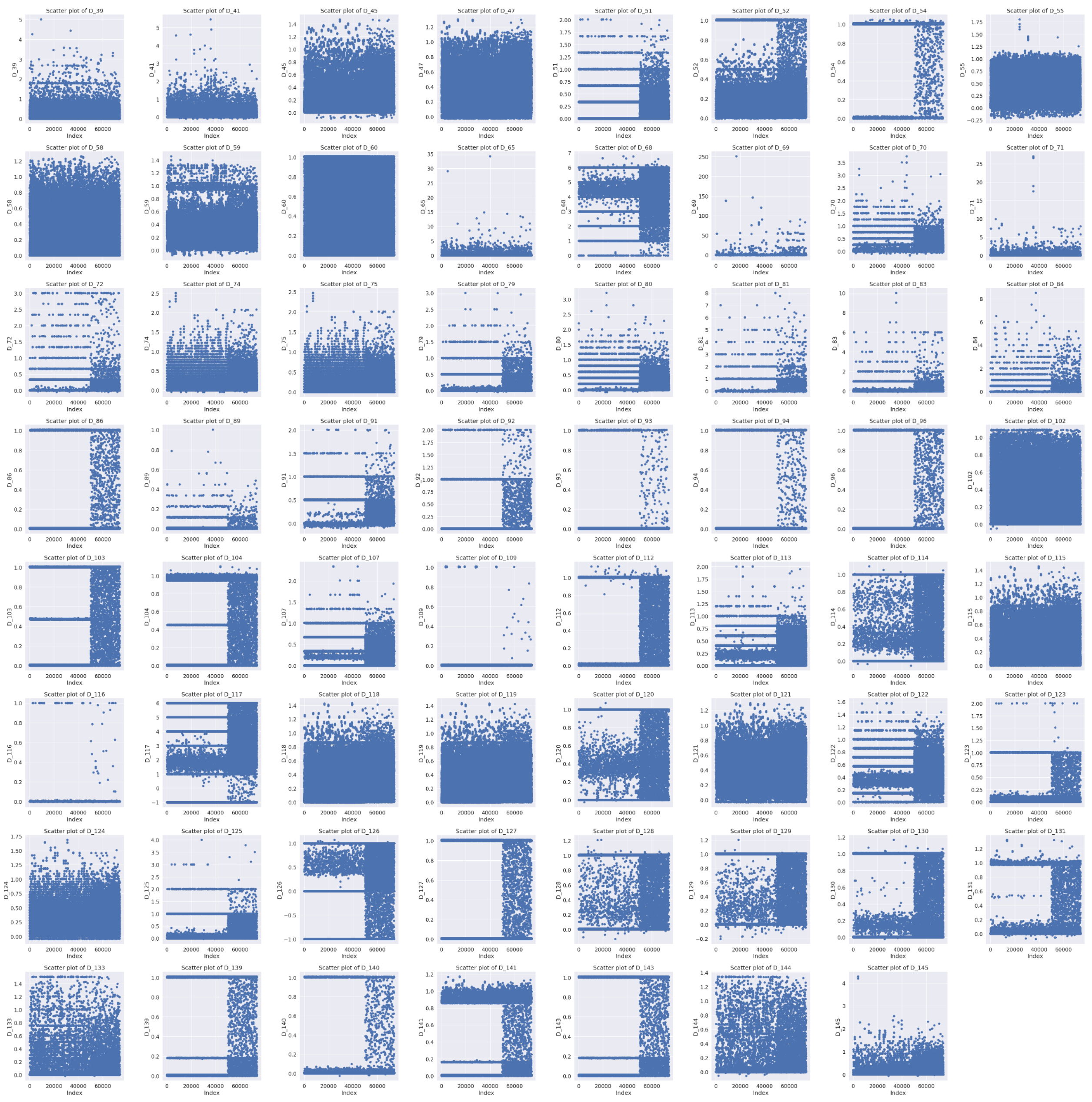


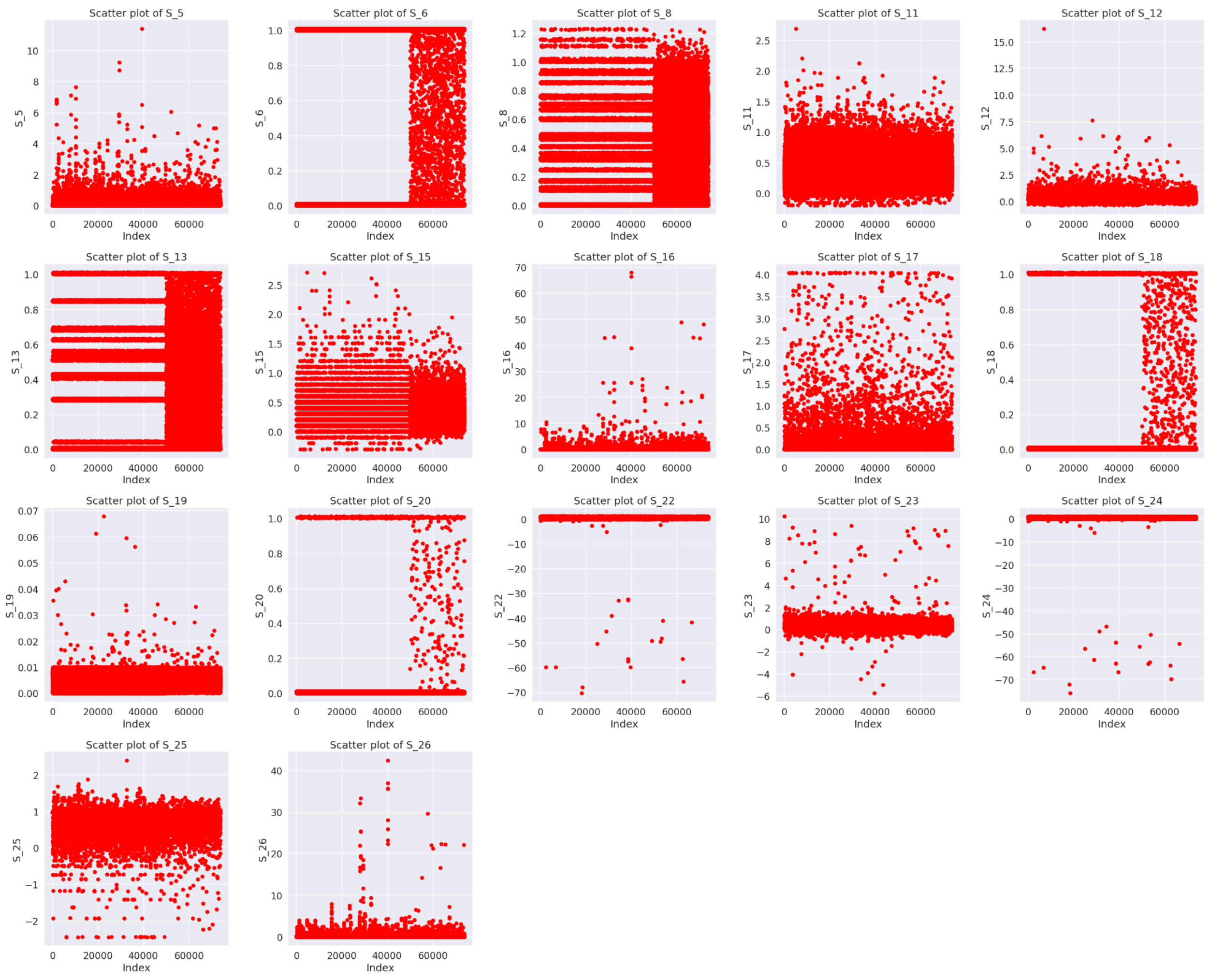
Clearly a Data Imbalance is observed between the 2 classes.

SCATTER PLOT FOR EACH COLUMN



```python
import matplotlib.pyplot as plt
import math

def plot_scatter_plots(dataframe, columns):
    num_plots = len(columns)
    ncols = int(math.ceil(math.sqrt(num_plots)))  # Calculate the number
    nrows = int(math.ceil(num_plots / ncols))  # Calculate the number of
    figsize = (ncols * 6, nrows * 6)

    fig, axes = plt.subplots(nrows=nrows, ncols=ncols, figsize=figsize)
    fig.subplots_adjust(hspace=0.5, wspace=0.5)

    for i, colname in enumerate(columns):
        row = i // ncols
        col = i % ncols
        ax = axes[row][col]
        ax.scatter(dataframe.index, dataframe[colname])
        ax.set_xlabel('Index')
        ax.set_ylabel(colname)
        ax.set_title(f'Scatter plot of {colname}')

    # Remove any unused subplots
    if num_plots < nrows * ncols:
        for i in range(num_plots, nrows * ncols):
            row = i // ncols
            col = i % ncols
            fig.delaxes(axes[row][col])

    plt.tight_layout()
    plt.show()

# Usage example
plot_scatter_plots(df_train_filtered, d_columns)
```


Scatter plot of D_39
Scatter plot of D_41
Scatter plot of D_45
Scatter plot of D_47
Scatter plot of D_51
Scatter plot of D_52
Scatter plot of D_54
Scatter plot of D_55
Scatter plot of D_58
Scatter plot of D_59
Scatter plot of D_60
Scatter plot of D_65
Scatter plot of D_68
Scatter plot of D_69
Scatter plot of D_70
Scatter plot of D_71
Scatter plot of D_72
Scatter plot of D_74
Scatter plot of D_75
Scatter plot of D_79
Scatter plot of D_80
Scatter plot of D_81
Scatter plot of D_83
Scatter plot of D_84
Scatter plot of D_86
Scatter plot of D_89
Scatter plot of D_91
Scatter plot of D_92
Scatter plot of D_93
Scatter plot of D_94
Scatter plot of D_96
Scatter plot of D_102
Scatter plot of D_103
Scatter plot of D_104
Scatter plot of D_107
Scatter plot of D_109
Scatter plot of D_112
Scatter plot of D_113
Scatter plot of D_114
Scatter plot of D_115
Scatter plot of D_116
Scatter plot of D_117
Scatter plot of D_118
Scatter plot of D_119
Scatter plot of D_120
Scatter plot of D_121
Scatter plot of D_122
Scatter plot of D_123
Scatter plot of D_124
Scatter plot of D_125
Scatter plot of D_126
Scatter plot of D_127
Scatter plot of D_128
Scatter plot of D_129
Scatter plot of D_130
Scatter plot of D_131
Scatter plot of D_133
Scatter plot of D_139
Scatter plot of D_140
Scatter plot of D_141
Scatter plot of D_143
Scatter plot of D_144
Scatter plot of D_145
Index


Scatter plot of S_5
Scatter plot of S_6
Scatter plot of S_8
Scatter plot of S_11
Scatter plot of S_12
Scatter plot of S_13
Scatter plot of S_15
Scatter plot of S_16
Scatter plot of S_17
Scatter plot of S_18
Scatter plot of S_19
Scatter plot of S_20
Scatter plot of S_22
Scatter plot of S_23
Scatter plot of S_24
Scatter plot of S_25
Scatter plot of S_26
Index

DATA PREPARATION

First we need to address the data imbalance

SMOTE (Synthetic Minority Oversampling Technique) is a popular algorithm for addressing class imbalance in machine learning datasets. It is an oversampling technique that aims to balance the class distribution by generating synthetic samples for the minority class. Here's some information you can include in your report about SMOTE:

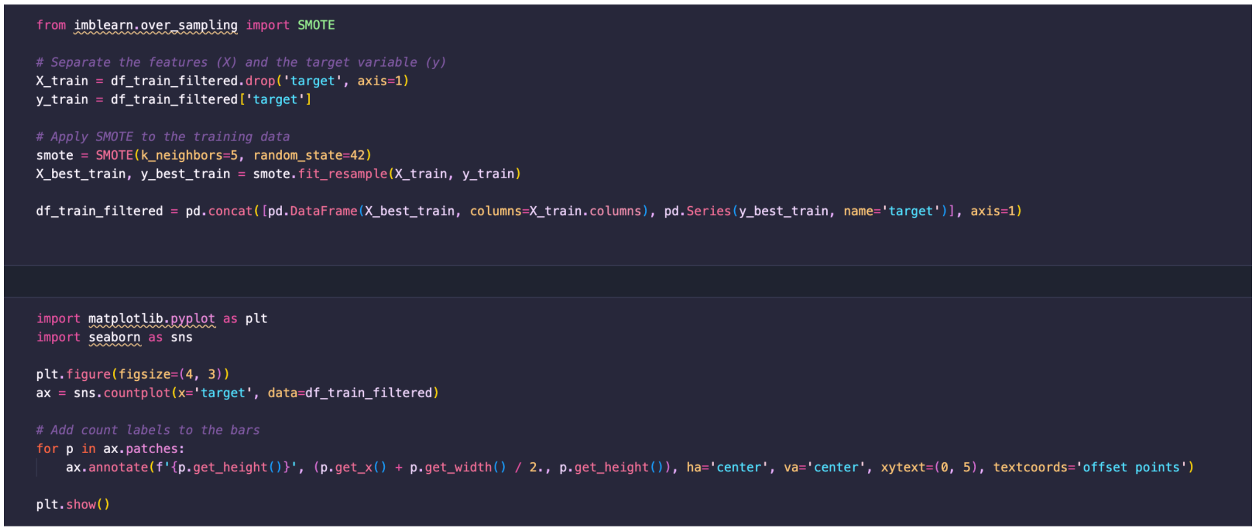
- Introduction to SMOTE:

SMOTE is designed to alleviate the problem of class imbalance, where one class (the minority class) has significantly fewer samples than the other class(es) (the majority class).It was introduced by Nitesh V. Chawla, et al., in their paper "SMOTE: Synthetic Minority Over-sampling Technique" in 2002.

- How SMOTE Works:

SMOTE works by creating synthetic samples in the feature space of the minority class.The basic idea is to identify the feature space neighborhood of a minority class sample and create synthetic examples by interpolating between existing minority class samples. For each minority class sample, SMOTE selects a few nearest neighbors (based on a distance metric like Euclidean distance) and generates synthetic samples along the line segments connecting the sample to its neighbors. The synthetic samples are generated by randomly selecting a specific number of nearest neighbors and computing a weighted average of the feature values to create new synthetic examples.

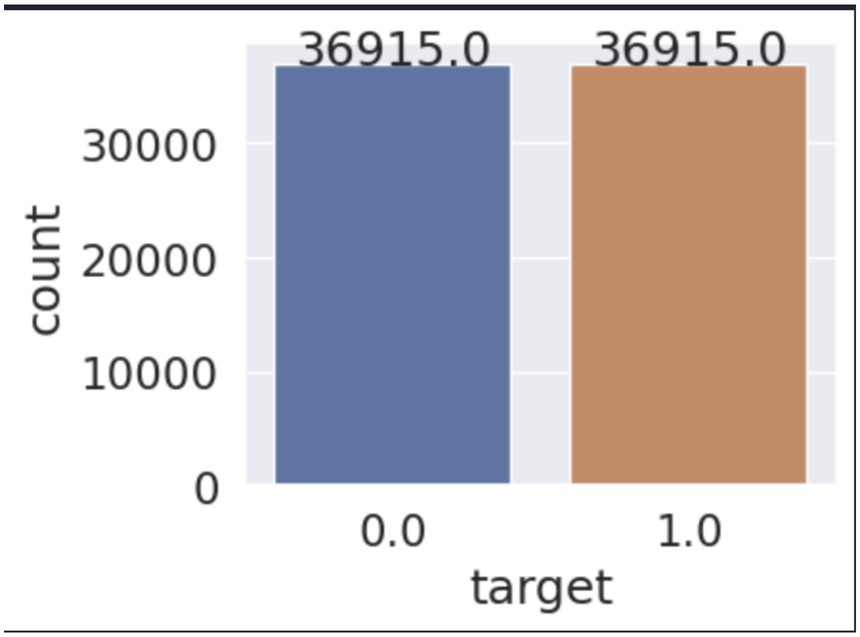
- Implementing SMOTE:
- SMOTE is widely available in machine learning libraries, such as the imbalanced-learn library in Python, where you can easily apply it to your dataset. You can use the SMOTE class from the imblearn.over_sampling module to perform SMOTE oversampling on your data.



```python
from imblearn.over_sampling import SMOTE

# Separate the features (X) and the target variable (y)
X_train = df_train_filtered.drop('target', axis=1)
y_train = df_train_filtered['target']

# Apply SMOTE to the training data
smote = SMOTE(k_neighbors=5, random_state=42)
X_best_train, y_best_train = smote.fit_resample(X_train, y_train)

df_train_filtered = pd.concat([pd.DataFrame(X_best_train, columns=X_train.columns), pd.Series(y_best_train, name='target')], axis=1)
```

```python
import matplotlib.pyplot as plt
import seaborn as sns

plt.figure(figsize=(4, 3))
ax = sns.countplot(x='target', data=df_train_filtered)

# Add count labels to the bars
for p in ax.patches:
    ax.annotate(f'{p.get_height()}', (p.get_x() + p.get_width() / 2., p.get_height()), ha='center', va='center', xytext=(0, 5), textcoords='offset points')

plt.show()
```

And thereby our Dats is no longer imbalanced.



MODELING

1. MODEL SELECTION

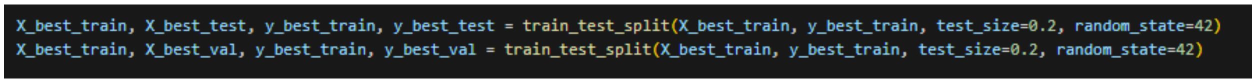
We use the DecisionTreeClassifier from sklearn library to build our model.

The model parameters description is as follows:

1. criterion= 'gini' the function to measure the quality of a split. Either 'gini' or 'entropy'.
2. splitter='best' : The strategy to choose the best split. Either 'best' or 'random'
3. max_depth=   None : The maximum depth of the tree. If None, the nodes are expanded until all leaves are pure or until they contain less than the min_samples_split
4. min_samples_split= 2 : The minimum number of samples required to split a node.
5. min_samples_leaf= 1 : The minimum number of samples require to be at a leaf node.
6. min_weight_fraction_leaf= 0.0 : The minimum weighted fraction of the sum of weights of all the input samples required to be at a node.
7. max_features=None : The number of features to consider when looking for the best split. Can be: int, float, auto' (the square root of number of features), 'sqrt' (same as auto), 'log2' (log of number of features), None (the number of features)
8. random_state=None : The control for the randomness of the estimator
9. max_leaf_nodes=None : Grow a tree with a maximum number of nodes. If None, then an unlimited number is possible.
10. min_impurity_decrease=0.0 : A node will be split if this split decreases the impurity greater than or equal to this value.
11. class_weight=None : Weights associated with different classes.
12. ccp_alpha=   0.0 : Complexity parameter used for Minimal Cost-Complexity Pruning.

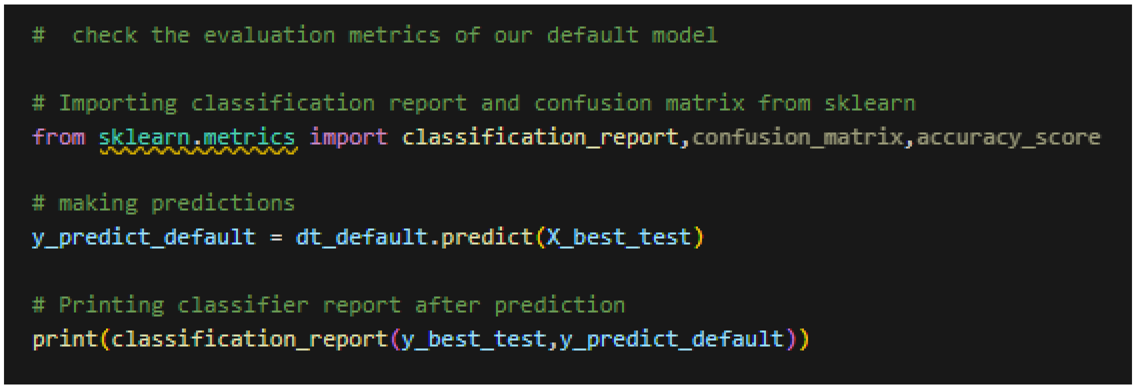
2. MODEL TRAINING

Then the data is split into train, test and split based on the code:



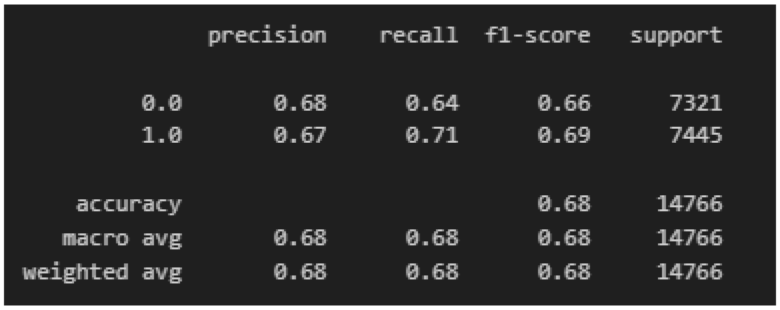
```
X_best_train, X_best_test, y_best_train, y_best_test = train_test_split(X_best_train, y_best_train, test_size=0.2, random_state=42)
X_best_train, X_best_val, y_best_train, y_best_val = train_test_split(X_best_train, y_best_train, test_size=0.2, random_state=42)
```

We now use the sklearn algorithm on this split data and also print the confusion matrix and classification report. We then print the classifier report after prediction.



```
#  check the evaluation metrics of our default model

# Importing classification report and confusion matrix from sklearn
from sklearn.metrics import classification_report,confusion_matrix,accuracy_score

# making predictions
y_predict_default = dt_default.predict(X_best_test)

# Printing classifier report after prediction
print(classification_report(y_best_test,y_predict_default))
```

We get the corresponding output:



```
              precision    recall  f1-score   support

         0.0       0.68      0.64      0.66      7321
         1.0       0.67      0.71      0.69      7445

    accuracy                           0.68     14766
   macro avg       0.68      0.68      0.68     14766
weighted avg       0.68      0.68      0.68     14766
```

MODEL PERFORMANCE ANALYSIS

Decision tree models are prone to overfitting, and it tries to consider the noisy data too. So there is a need for hyperparameter tuning, such as restricting the depth of the tree or by using more mathematically sound methods such as pruning

The main idea is to iteratively prune the decision tree by removing branches or nodes that do not significantly improve the model's predictive power. This pruning process is guided by a complexity parameter called alpha, often referred to as ccp_alpha in scikit-learn's DecisionTreeClassifier.

Cost complexity pruning, also known as minimal cost complexity pruning or weakest link pruning, is a technique used to prune decision trees based on a cost-complexity measure. The goal of pruning is to create a simpler and more generalized decision tree by removing unnecessary branches and nodes. Cost complexity pruning helps to prevent overfitting and improve the model's ability to generalize to unseen data.

During pruning, the algorithm calculates the impurity reduction that would be achieved by removing a node and its subtree. It compares this reduction with the corresponding alpha value to determine whether the removal is justified. Nodes with smaller impurity reductions relative to their alpha values are pruned, leading to a simpler and more regularized tree.

Cost complexity pruning helps to avoid overfitting by controlling the complexity of the decision tree and preventing it from memorizing the training data. It strikes a balance between model complexity and performance by finding an optimal trade-off point.

The process of cost complexity pruning typically involves the following steps:

1. Start with an initial decision tree: We begin with a fully grown decision tree that has not been pruned.
2. Calculate the cost complexity of each subtree:

   For each internal node in the decision tree, we calculate the cost complexity measure. The cost complexity measure is defined as the sum of two components: the total impurity of the subtree and the complexity cost. The complexity cost is given by α times the number of leaf nodes in the subtree.

3. Determine the effective alpha values:

   The effective alpha values represent the minimum alpha values required to remove subtrees at each internal node without increasing the total cost complexity of the tree. We can obtain a list of effective alpha values by iteratively considering each internal node and selecting the minimum alpha value that allows pruning while maintaining the overall cost complexity.

4. Prune the decision tree:

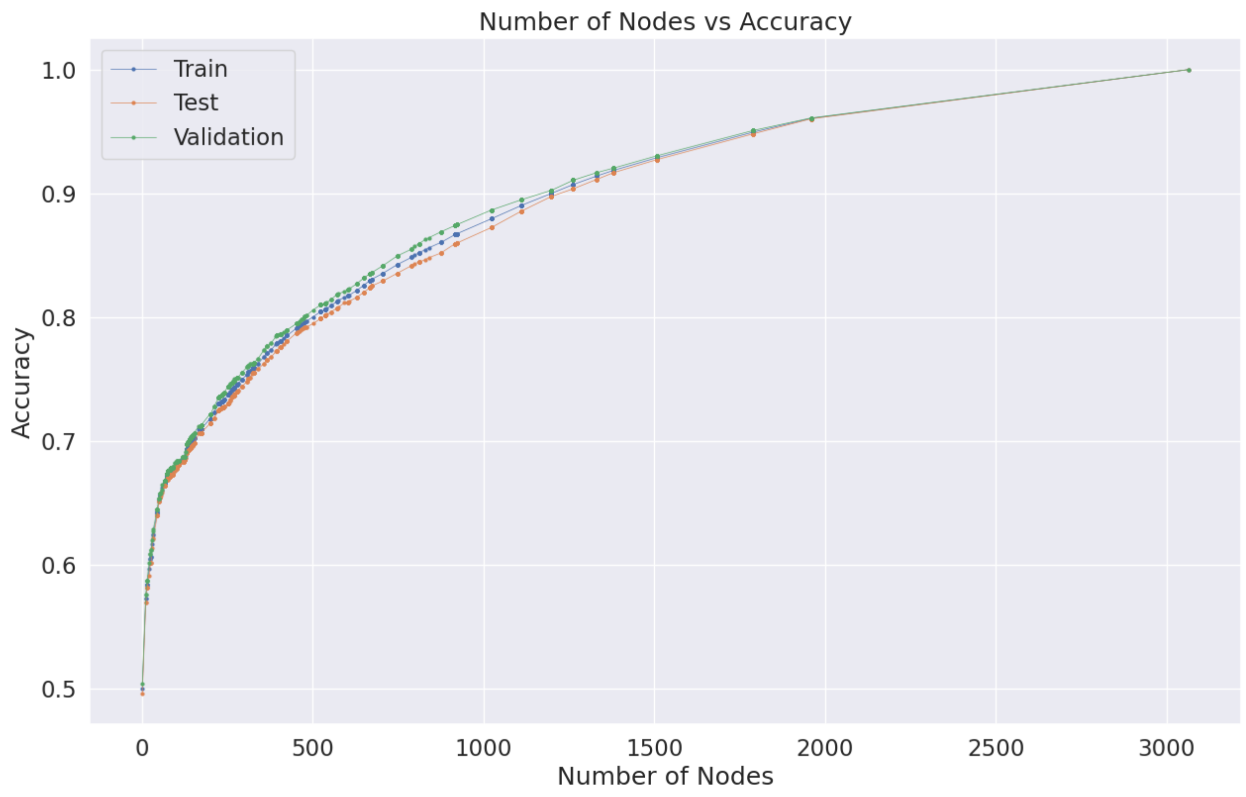
   Using the effective alpha values, we prune the decision tree by removing the subtrees associated with alpha values greater than the effective alpha. This process involves removing the unnecessary branches and nodes that contribute less to the overall predictive power of the tree.

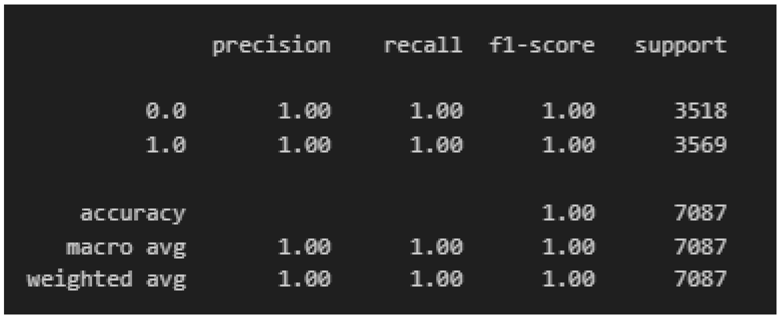
5. Select the optimal pruned tree:

   After pruning, we evaluate the pruned decision trees on a validation set or using cross-validation. We select the decision tree with the highest performance, such as accuracy or F1-score, on the validation set as the optimal pruned tree.

The code performs cost complexity pruning on a decision tree classifier to find an optimal decision tree model. It prunes the decision tree using different values of the alpha parameter, which controls the amount of regularization applied during pruning. The code then evaluates the pruned decision trees on training, testing, and validation datasets based on their accuracy scores. It plots the number of nodes versus accuracy to visualize the trade-off between model complexity and performance. Finally, it selects the optimal decision tree based on the highest validation accuracy and evaluates it on the testing dataset using the classification report.

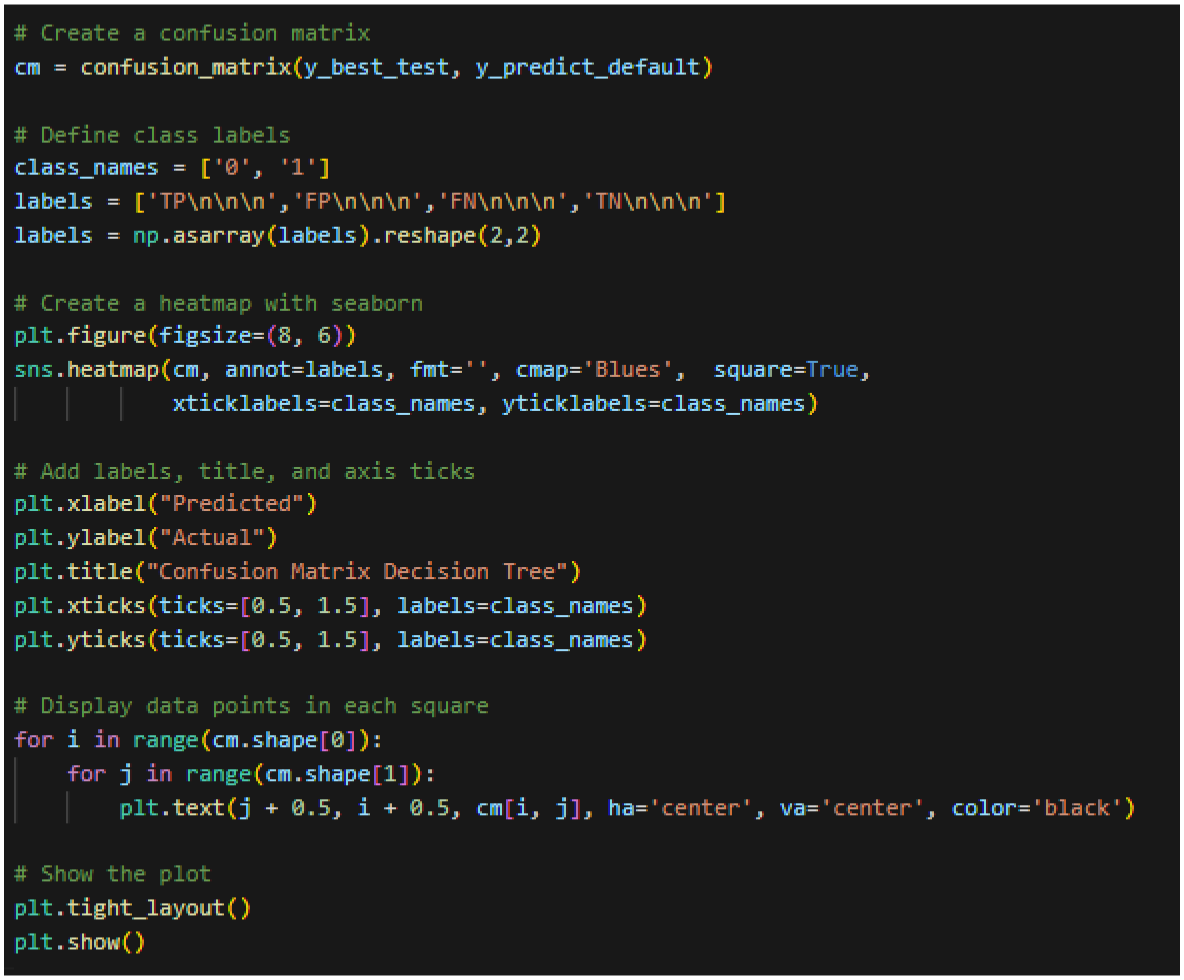
The output plot comes out as follows:



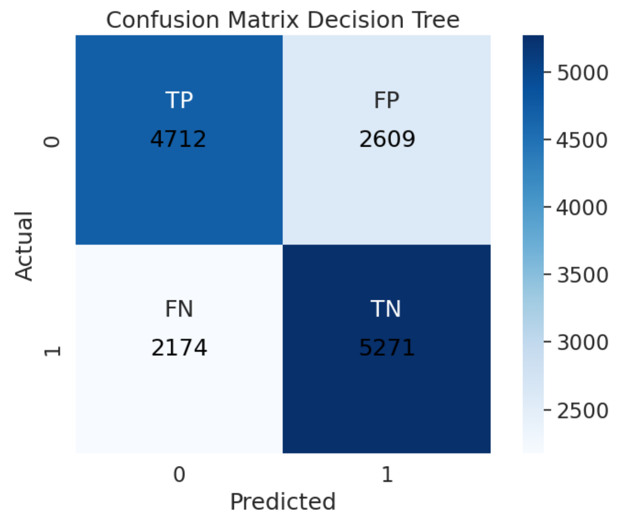


```
              precision    recall  f1-score   support

         0.0       1.00      1.00      1.00      3518
         1.0       1.00      1.00      1.00      3569

    accuracy                           1.00      7087
   macro avg       1.00      1.00      1.00      7087
weighted avg       1.00      1.00      1.00      7087
```

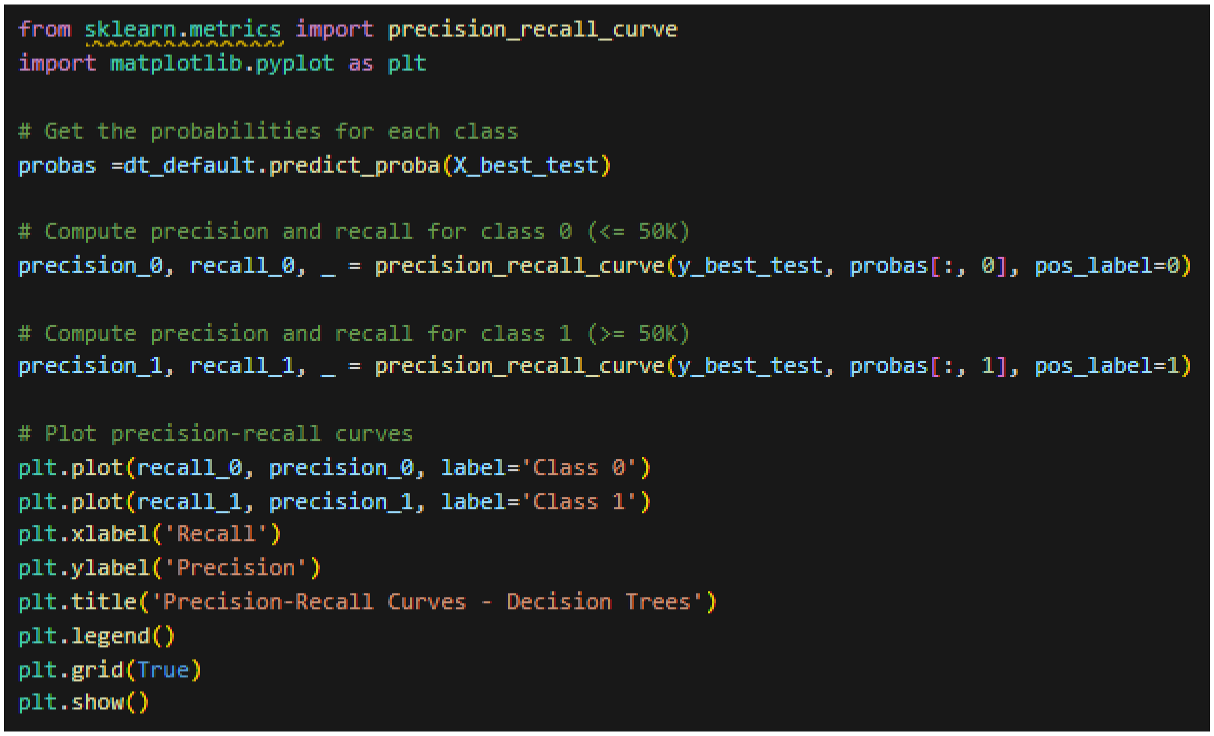
Now we create a confusion matrix:



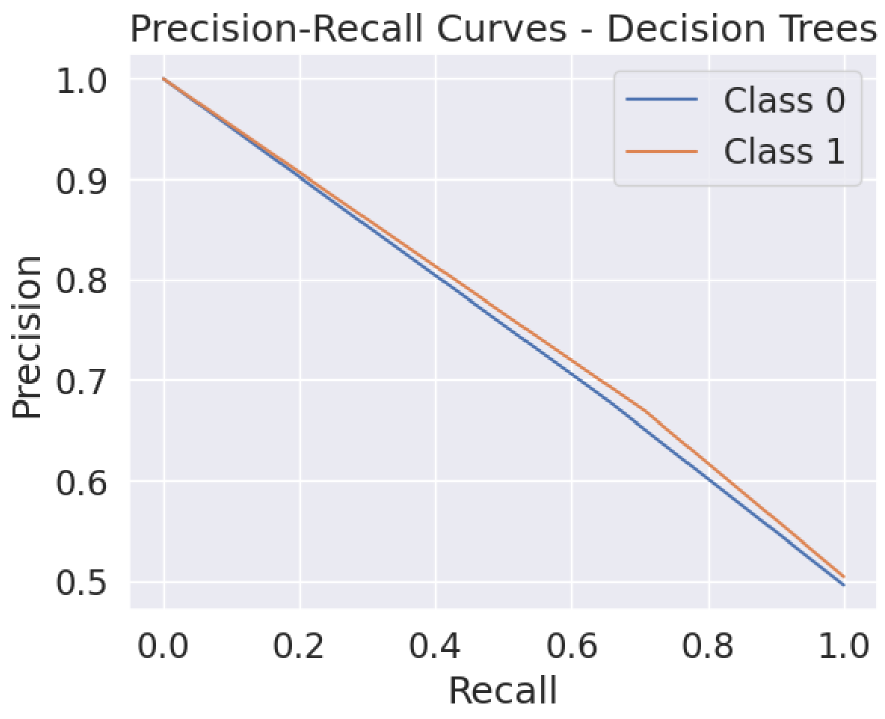
```
# Create a confusion matrix
cm = confusion_matrix(y_best_test, y_predict_default)

# Define class labels
class_names = ['0', '1']
labels = ['TP\n\n\n','FP\n\n\n','FN\n\n\n','TN\n\n\n']
labels = np.asarray(labels).reshape(2,2)

# Create a heatmap with seaborn
plt.figure(figsize=(8, 6))
sns.heatmap(cm, annot=labels, fmt='', cmap='Blues',  square=True,
            xticklabels=class_names, yticklabels=class_names)

# Add labels, title, and axis ticks
plt.xlabel("Predicted")
plt.ylabel("Actual")
plt.title("Confusion Matrix Decision Tree")
plt.xticks(ticks=[0.5, 1.5], labels=class_names)
plt.yticks(ticks=[0.5, 1.5], labels=class_names)

# Display data points in each square
for i in range(cm.shape[0]):
    for j in range(cm.shape[1]):
        plt.text(j + 0.5, i + 0.5, cm[i, j], ha='center', va='center', color='black')

# Show the plot
plt.tight_layout()
plt.show()
```



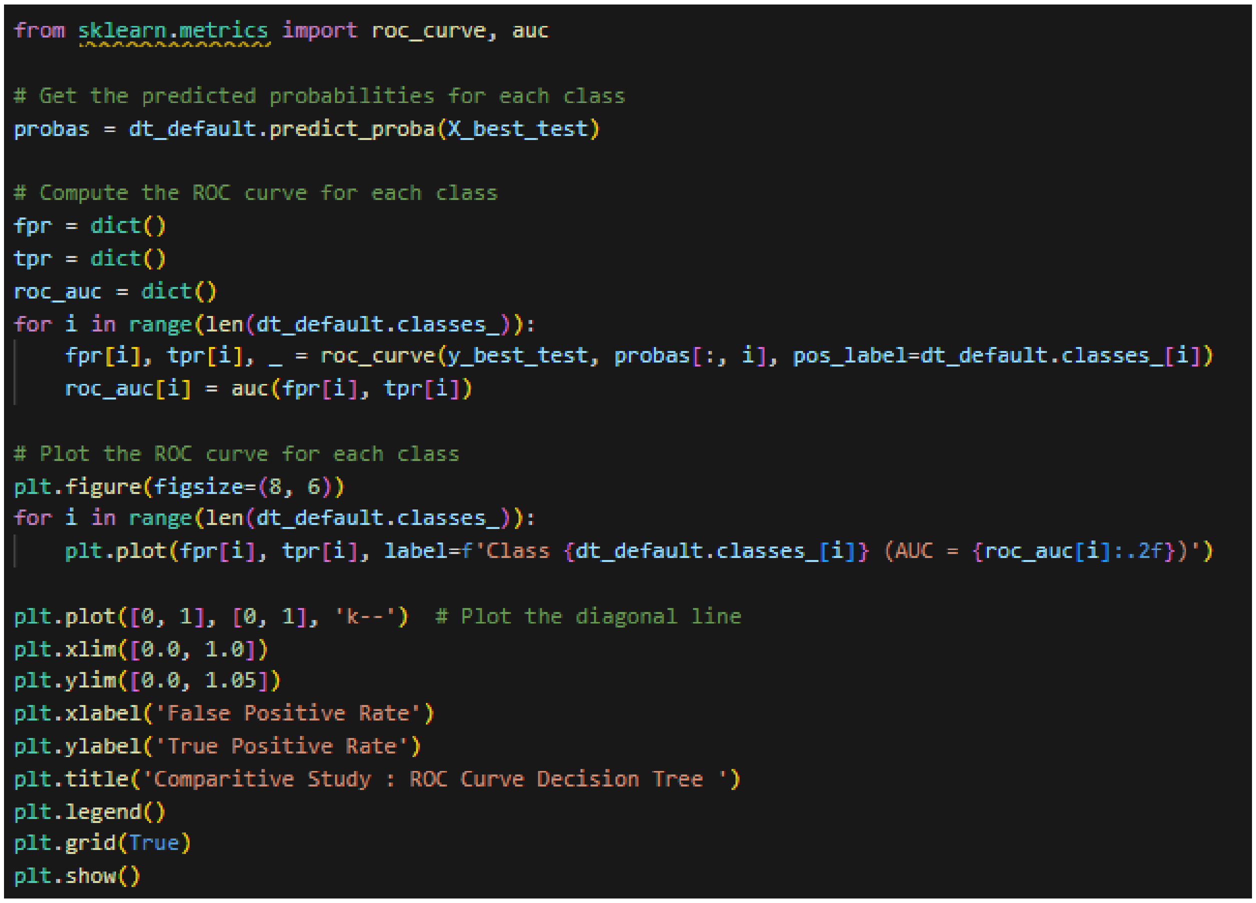
The precision-recall curve is as follows:



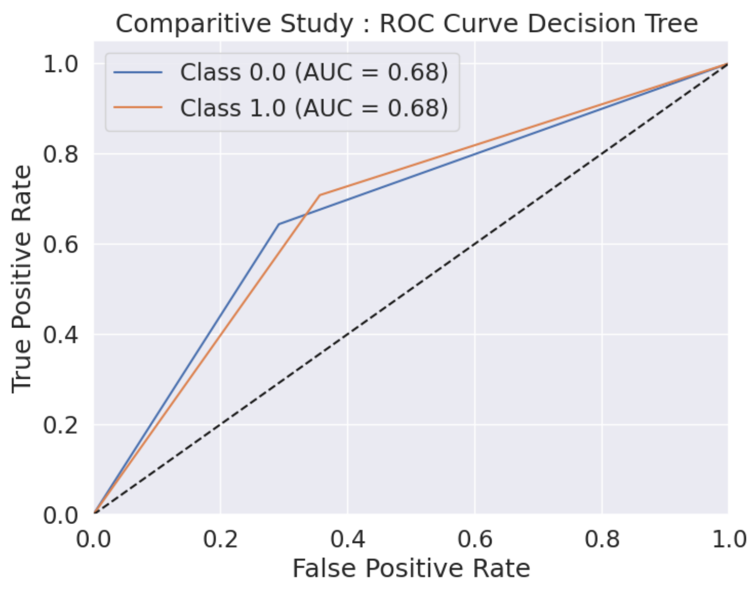
```
from sklearn.metrics import precision_recall_curve
import matplotlib.pyplot as plt

# Get the probabilities for each class
probas =dt_default.predict_proba(X_best_test)

# Compute precision and recall for class 0 (<= 50K)
precision_0, recall_0, _ = precision_recall_curve(y_best_test, probas[:, 0], pos_label=0)

# Compute precision and recall for class 1 (>= 50K)
precision_1, recall_1, _ = precision_recall_curve(y_best_test, probas[:, 1], pos_label=1)

# Plot precision-recall curves
plt.plot(recall_0, precision_0, label='Class 0')
plt.plot(recall_1, precision_1, label='Class 1')
plt.xlabel('Recall')
plt.ylabel('Precision')
plt.title('Precision-Recall Curves - Decision Trees')
plt.legend()
plt.grid(True)
plt.show()
```



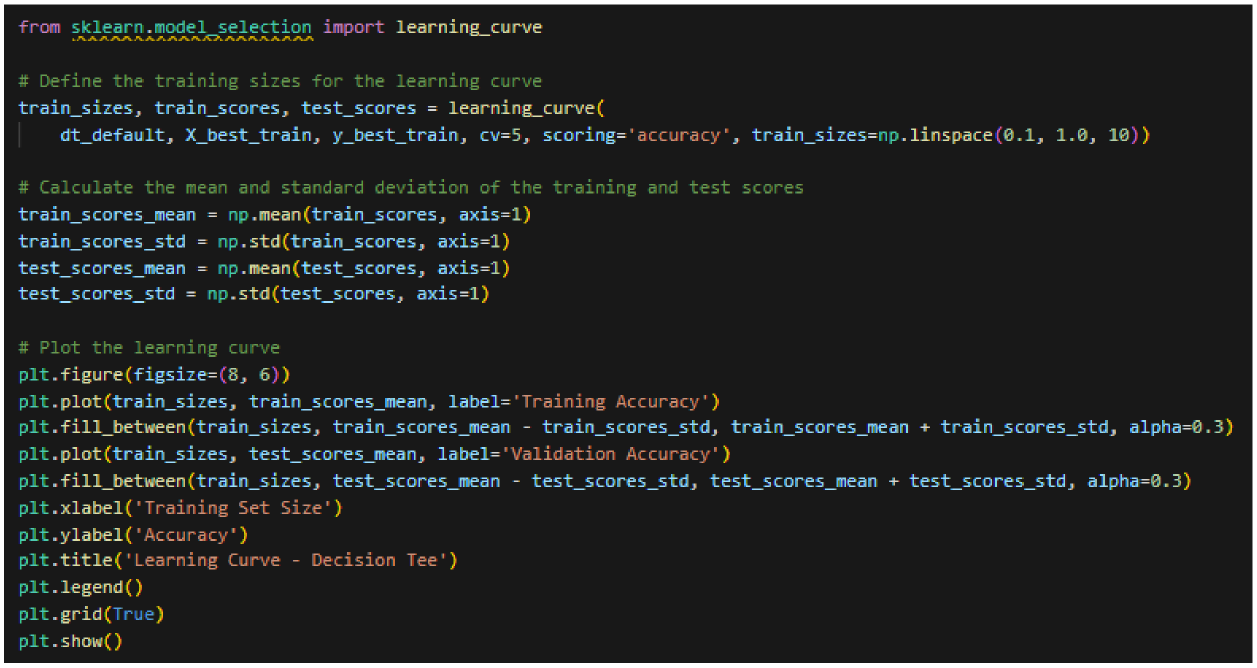
The ROC curve is as follows:



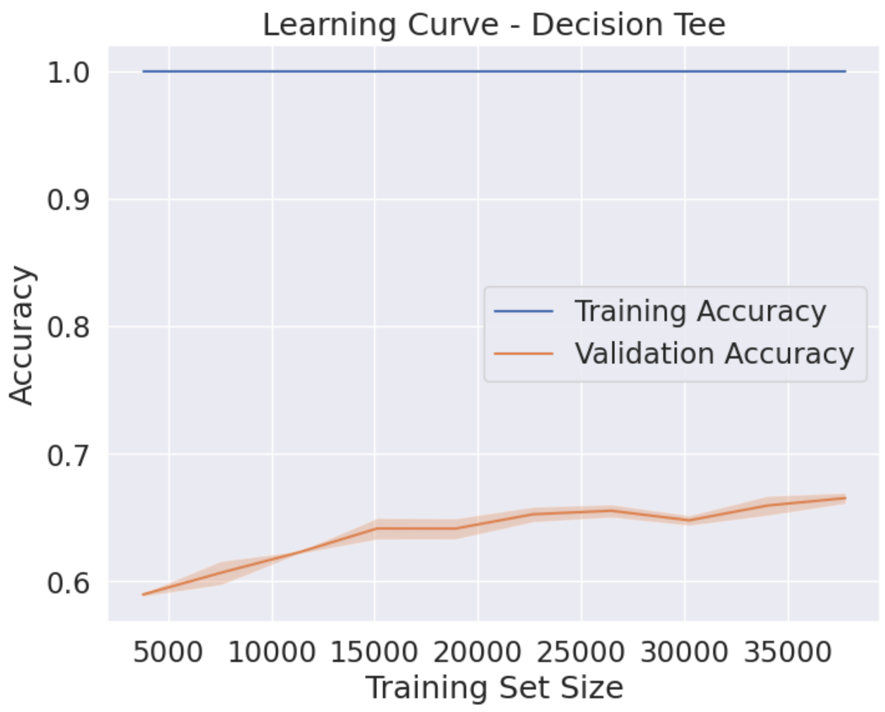
```
from sklearn.metrics import roc_curve, auc

# Get the predicted probabilities for each class
probas = dt_default.predict_proba(X_best_test)

# Compute the ROC curve for each class
fpr = dict()
tpr = dict()
roc_auc = dict()
for i in range(len(dt_default.classes_)):
    fpr[i], tpr[i], _ = roc_curve(y_best_test, probas[:, i], pos_label=dt_default.classes_[i])
    roc_auc[i] = auc(fpr[i], tpr[i])

# Plot the ROC curve for each class
plt.figure(figsize=(8, 6))
for i in range(len(dt_default.classes_)):
    plt.plot(fpr[i], tpr[i], label=f'Class {dt_default.classes_[i]} (AUC = {roc_auc[i]:.2f})')

plt.plot([0, 1], [0, 1], 'k--')  # Plot the diagonal line
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.05])
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('Comparitive Study : ROC Curve Decision Tree ')
plt.legend()
plt.grid(True)
plt.show()
```



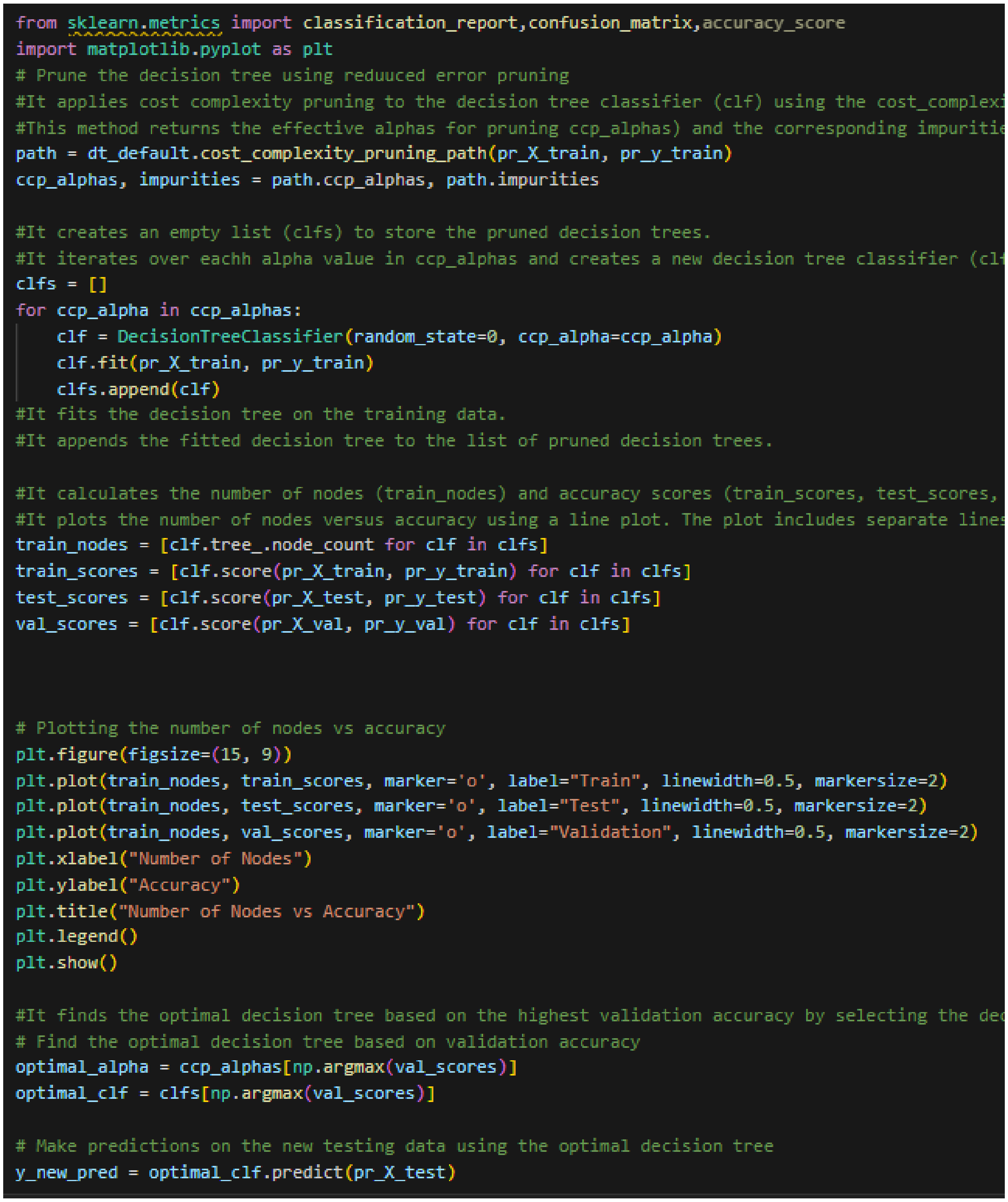
The learning curve is as follows:



```
from sklearn.model_selection import learning_curve

# Define the training sizes for the learning curve
train_sizes, train_scores, test_scores = learning_curve(
    dt_default, X_best_train, y_best_train, cv=5, scoring='accuracy', train_sizes=np.linspace(0.1, 1.0, 10))

# Calculate the mean and standard deviation of the training and test scores
train_scores_mean = np.mean(train_scores, axis=1)
train_scores_std = np.std(train_scores, axis=1)
test_scores_mean = np.mean(test_scores, axis=1)
test_scores_std = np.std(test_scores, axis=1)

# Plot the learning curve
plt.figure(figsize=(8, 6))
plt.plot(train_sizes, train_scores_mean, label='Training Accuracy')
plt.fill_between(train_sizes, train_scores_mean - train_scores_std, train_scores_mean + train_scores_std, alpha=0.3)
plt.plot(train_sizes, test_scores_mean, label='Validation Accuracy')
plt.fill_between(train_sizes, test_scores_mean - test_scores_std, test_scores_mean + test_scores_std, alpha=0.3)
plt.xlabel('Training Set Size')
plt.ylabel('Accuracy')
plt.title('Learning Curve - Decision Tee')
plt.legend()
plt.grid(True)
plt.show()
```


Learning Curve - Decision Tee
1.0
0.9
0.8
0.7
0.6
Accuracy
Training Accuracy
Validation Accuracy
5000
10000
15000
20000
25000
30000
35000
Training Set Size

CODE FOR PRUNING:



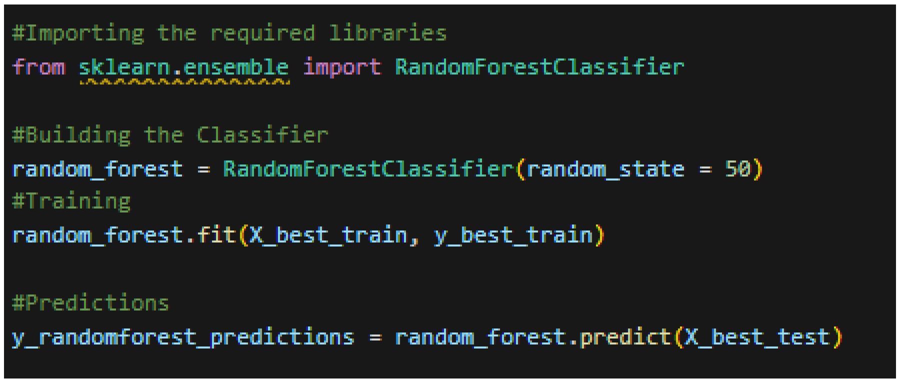
```
from sklearn.metrics import classification_report,confusion_matrix,accuracy_score
import matplotlib.pyplot as plt
# Prune the decision tree using reduuced error pruning
#It applies cost complexity pruning to the decision tree classifier (clf) using the cost_complexi
#This method returns the effective alphas for pruning ccp_alphas) and the corresponding impuritie
path = dt_default.cost_complexity_pruning_path(pr_X_train, pr_y_train)
ccp_alphas, impurities = path.ccp_alphas, path.impurities

#It creates an empty list (clfs) to store the pruned decision trees.
#It iterates over eachh alpha value in ccp_alphas and creates a new decision tree classifier (cl
clfs = []
for ccp_alpha in ccp_alphas:
    clf = DecisionTreeClassifier(random_state=0, ccp_alpha=ccp_alpha)
    clf.fit(pr_X_train, pr_y_train)
    clfs.append(clf)
#It fits the decision tree on the training data.
#It appends the fitted decision tree to the list of pruned decision trees.

#It calculates the number of nodes (train_nodes) and accuracy scores (train_scores, test_scores,
#It plots the number of nodes versus accuracy using a line plot. The plot includes separate lines
train_nodes = [clf.tree_.node_count for clf in clfs]
train_scores = [clf.score(pr_X_train, pr_y_train) for clf in clfs]
test_scores = [clf.score(pr_X_test, pr_y_test) for clf in clfs]
val_scores = [clf.score(pr_X_val, pr_y_val) for clf in clfs]



# Plotting the number of nodes vs accuracy
plt.figure(figsize=(15, 9))
plt.plot(train_nodes, train_scores, marker='o', label="Train", linewidth=0.5, markersize=2)
plt.plot(train_nodes, test_scores, marker='o', label="Test", linewidth=0.5, markersize=2)
plt.plot(train_nodes, val_scores, marker='o', label="Validation", linewidth=0.5, markersize=2)
plt.xlabel("Number of Nodes")
plt.ylabel("Accuracy")
plt.title("Number of Nodes vs Accuracy")
plt.legend()
plt.show()

#It finds the optimal decision tree based on the highest validation accuracy by selecting the dec
# Find the optimal decision tree based on validation accuracy
optimal_alpha = ccp_alphas[np.argmax(val_scores)]
optimal_clf = clfs[np.argmax(val_scores)]

# Make predictions on the new testing data using the optimal decision tree
y_new_pred = optimal_clf.predict(pr_X_test)
```

RANDOM FOREST CLASSIFIER:

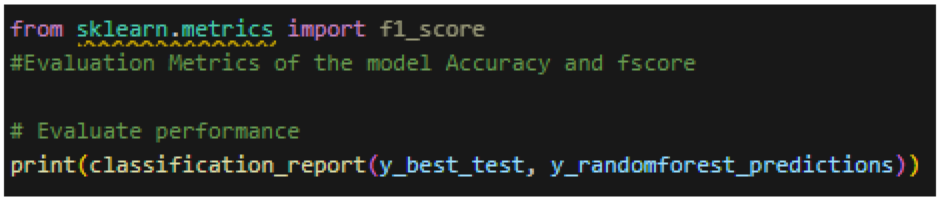
Random Forest is a popular ensemble learning algorithm used in machine learning for classification and regression tasks. It is an extension of the single decision tree algorithm.

In a Random Forest, multiple decision trees are trained on different subsets of the original dataset, with each tree making an independent prediction. The final prediction is obtained by aggregating the predictions of all the individual trees. The aggregation process can be done by taking the majority vote (for classification) or averaging (for regression).

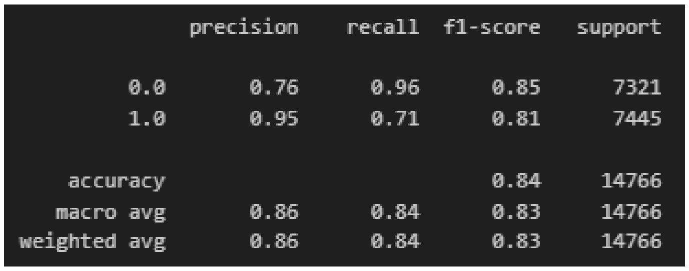
We run the same data now on the random forest classifier to do a comparative study with the decision tree we built above.



```python
#Importing the required libraries
from sklearn.ensemble import RandomForestClassifier

#Building the Classifier
random_forest = RandomForestClassifier(random_state = 50)
#Training
random_forest.fit(X_best_train, y_best_train)

#Predictions
y_randomforest_predictions = random_forest.predict(X_best_test)
```

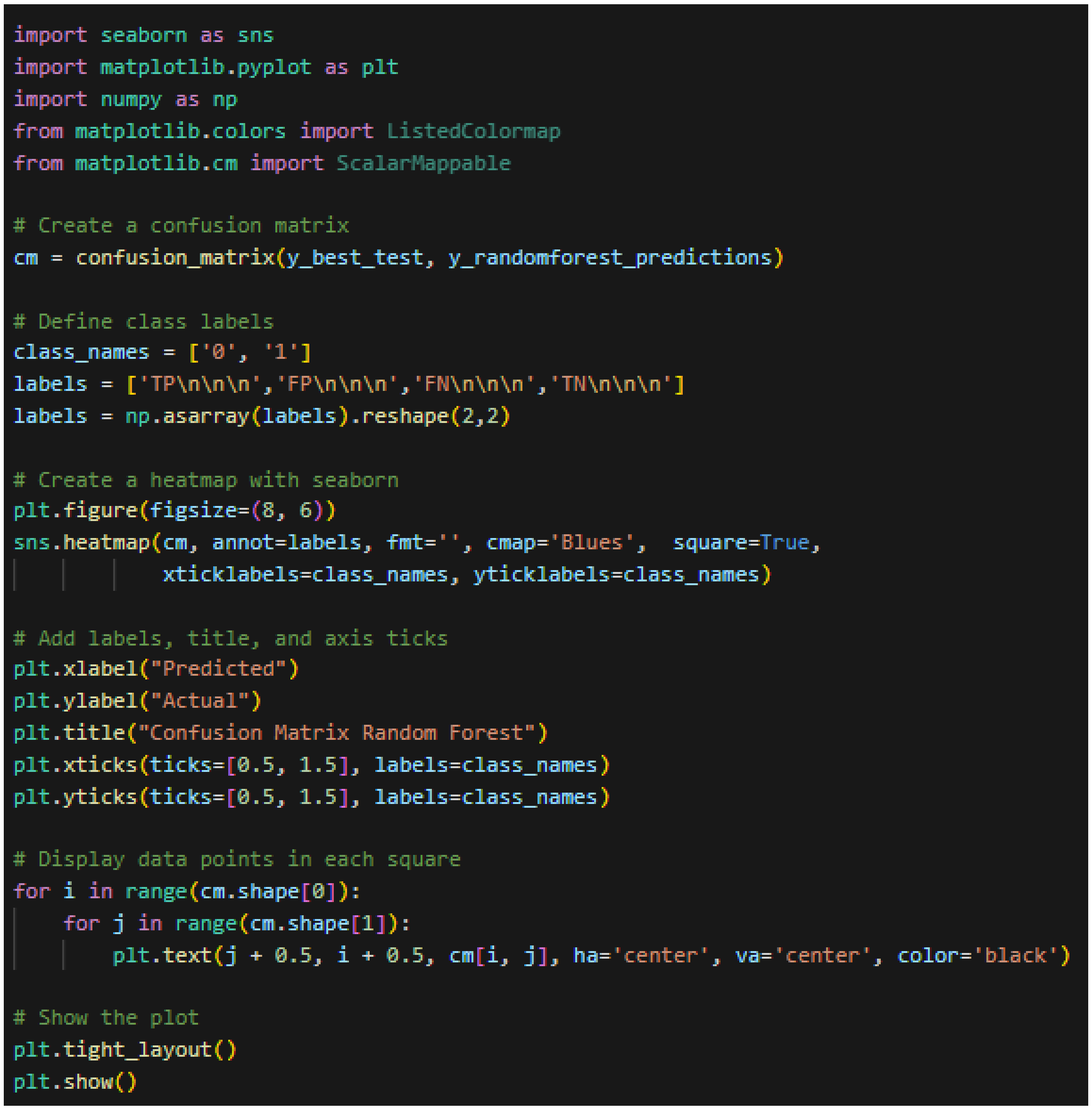


```python
from sklearn.metrics import f1_score
#Evaluation Metrics of the model Accuracy and fscore

# Evaluate performance
print(classification_report(y_best_test, y_randomforest_predictions))
```



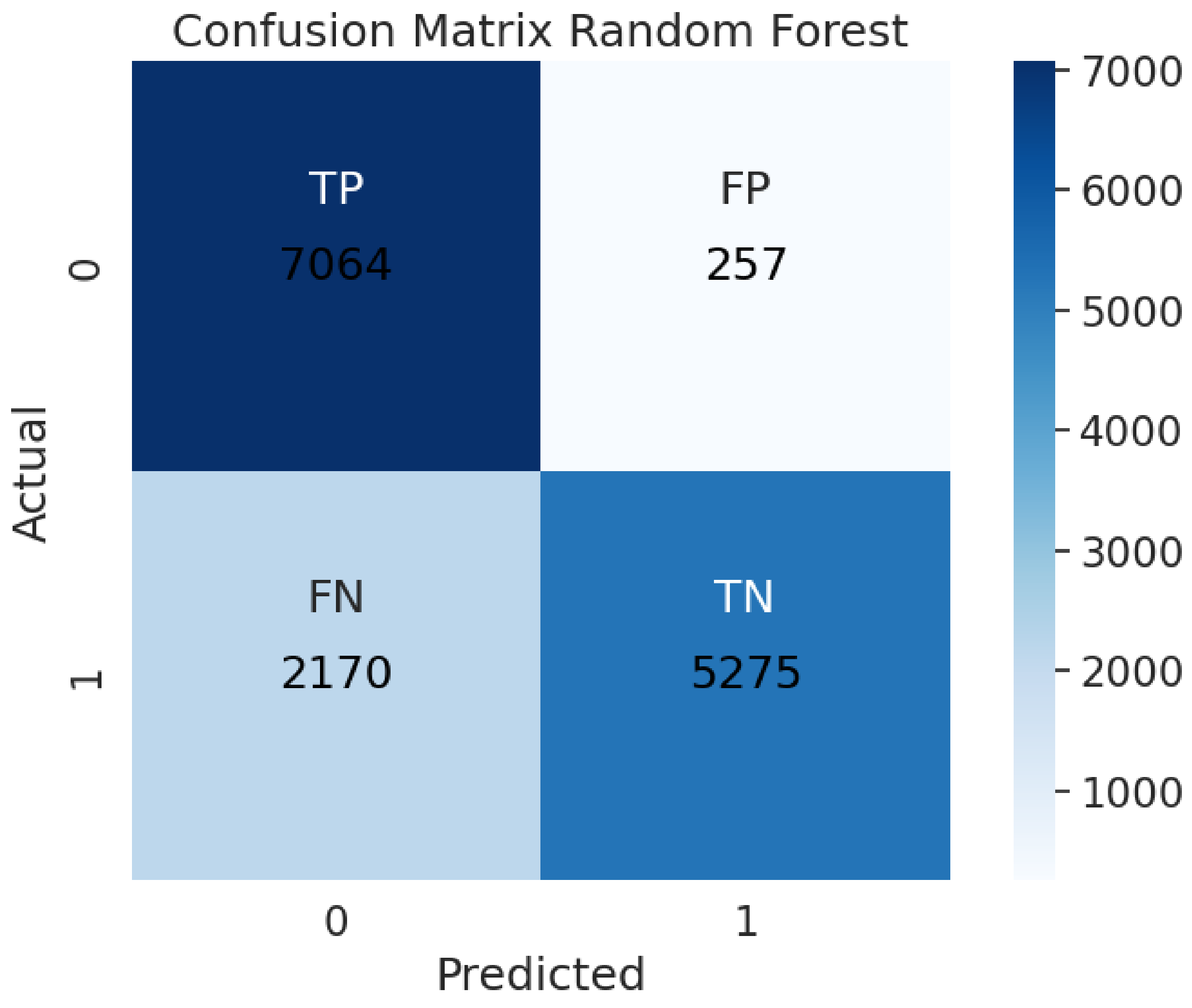
```
              precision    recall  f1-score   support

         0.0       0.76      0.96      0.85      7321
         1.0       0.95      0.71      0.81      7445

    accuracy                           0.84     14766
   macro avg       0.86      0.84      0.83     14766
weighted avg       0.86      0.84      0.83     14766
```

Performance metrics such as the confusion matrix is obtained.

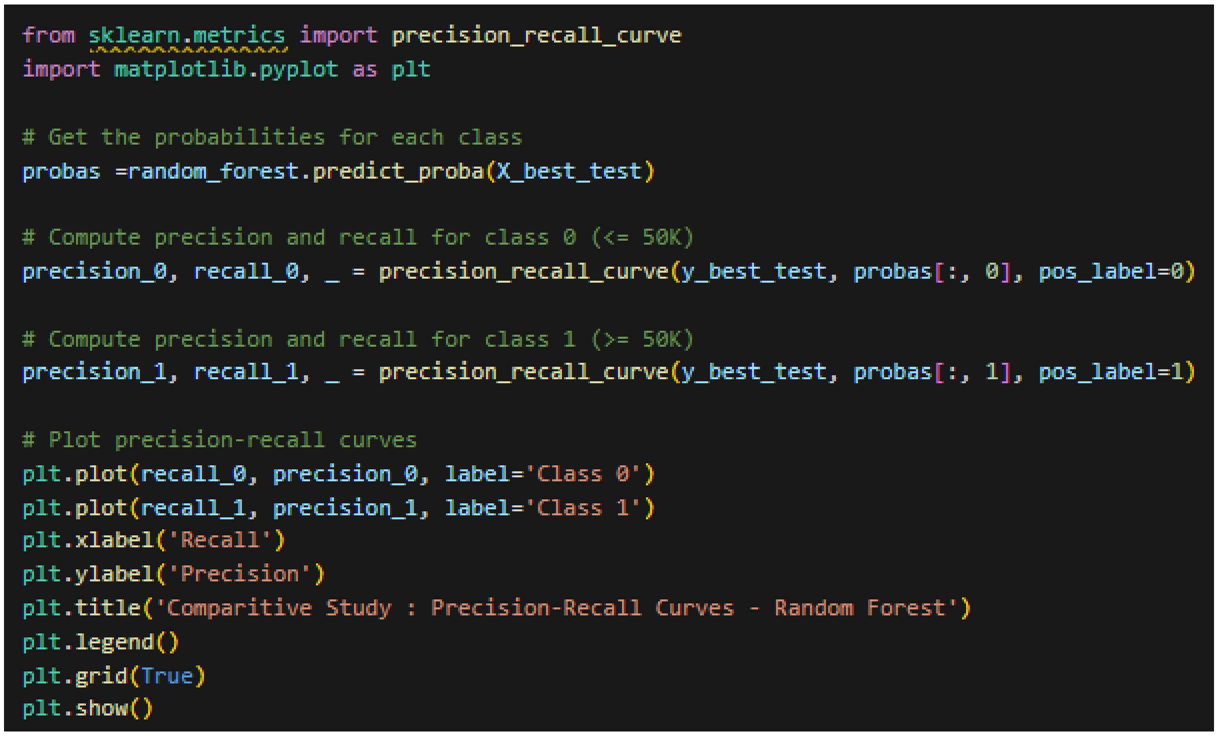


```python
import seaborn as sns
import matplotlib.pyplot as plt
import numpy as np
from matplotlib.colors import ListedColormap
from matplotlib.cm import ScalarMappable

# Create a confusion matrix
cm = confusion_matrix(y_best_test, y_randomforest_predictions)

# Define class labels
class_names = ['0', '1']
labels = ['TP\n\n\n','FP\n\n\n','FN\n\n\n','TN\n\n\n']
labels = np.asarray(labels).reshape(2,2)

# Create a heatmap with seaborn
plt.figure(figsize=(8, 6))
sns.heatmap(cm, annot=labels, fmt='', cmap='Blues',  square=True,
            xticklabels=class_names, yticklabels=class_names)

# Add labels, title, and axis ticks
plt.xlabel("Predicted")
plt.ylabel("Actual")
plt.title("Confusion Matrix Random Forest")
plt.xticks(ticks=[0.5, 1.5], labels=class_names)
plt.yticks(ticks=[0.5, 1.5], labels=class_names)

# Display data points in each square
for i in range(cm.shape[0]):
    for j in range(cm.shape[1]):
        plt.text(j + 0.5, i + 0.5, cm[i, j], ha='center', va='center', color='black')

# Show the plot
plt.tight_layout()
plt.show()
```

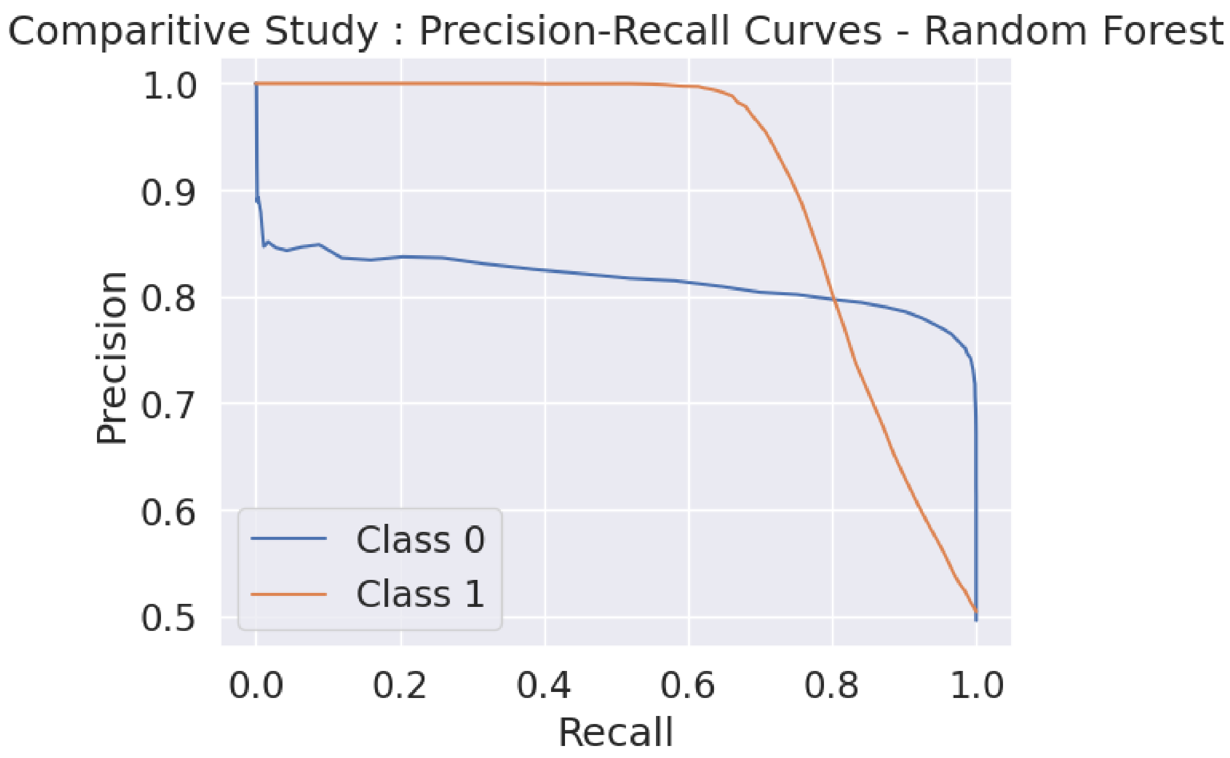
The confusion matrix is obtained as follows:



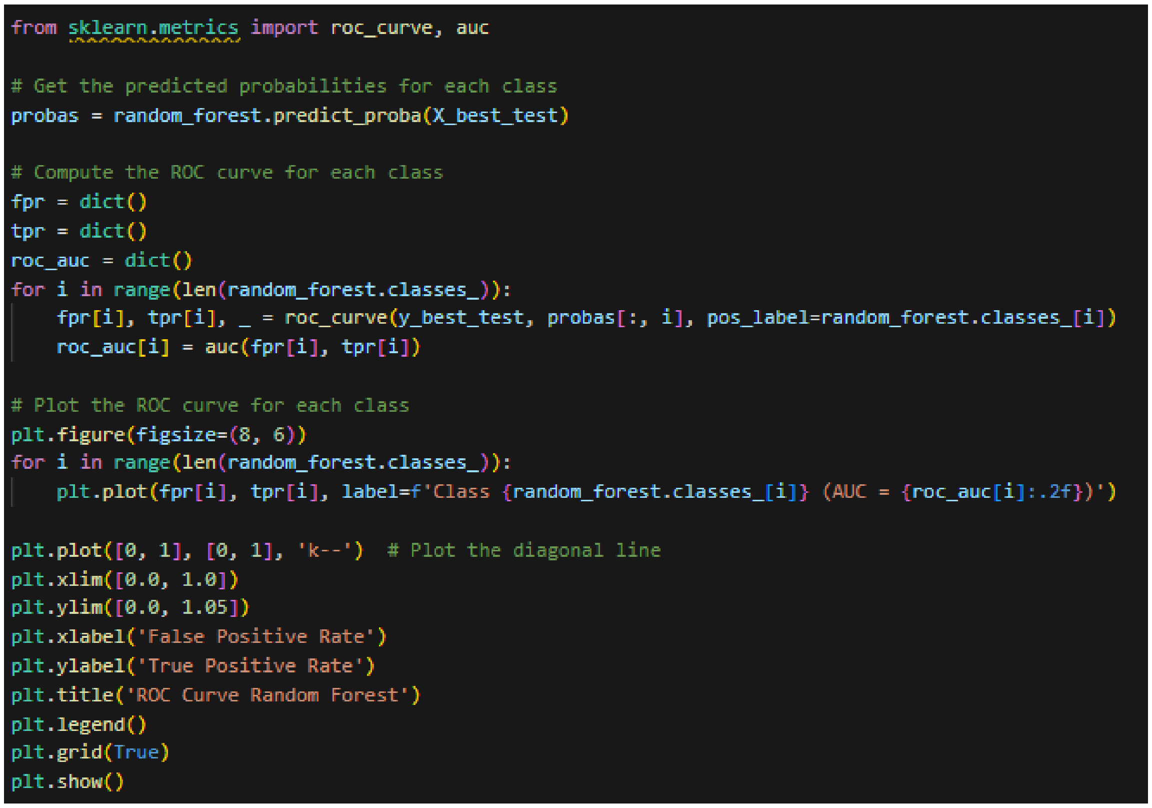
The precision recall curves of the random forest is now drawn:



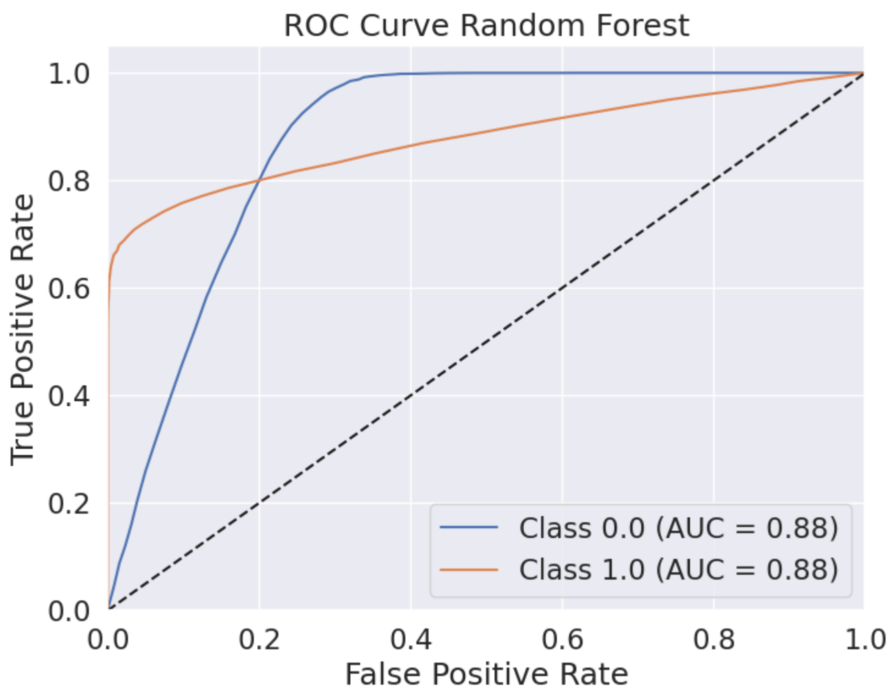
```python
from sklearn.metrics import precision_recall_curve
import matplotlib.pyplot as plt

# Get the probabilities for each class
probas =random_forest.predict_proba(X_best_test)

# Compute precision and recall for class 0 (<= 50K)
precision_0, recall_0, _ = precision_recall_curve(y_best_test, probas[:, 0], pos_label=0)

# Compute precision and recall for class 1 (>= 50K)
precision_1, recall_1, _ = precision_recall_curve(y_best_test, probas[:, 1], pos_label=1)

# Plot precision-recall curves
plt.plot(recall_0, precision_0, label='Class 0')
plt.plot(recall_1, precision_1, label='Class 1')
plt.xlabel('Recall')
plt.ylabel('Precision')
plt.title('Comparitive Study : Precision-Recall Curves - Random Forest')
plt.legend()
plt.grid(True)
plt.show()
```



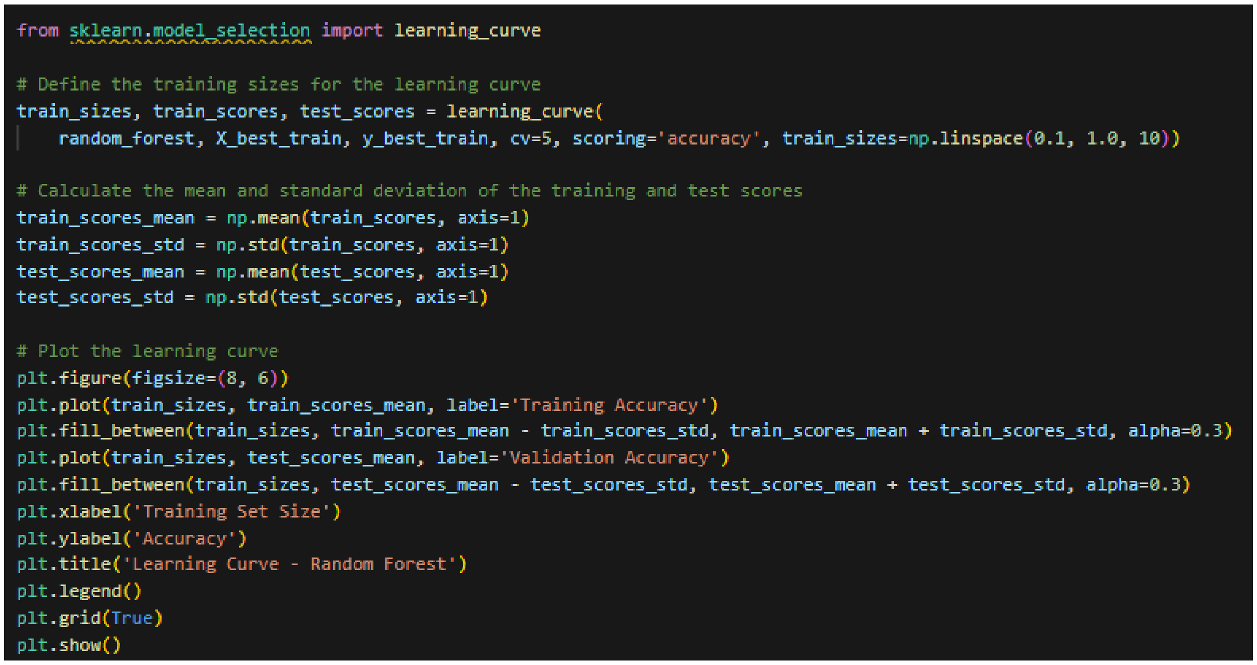
ROC curve of the random forest is obtained:



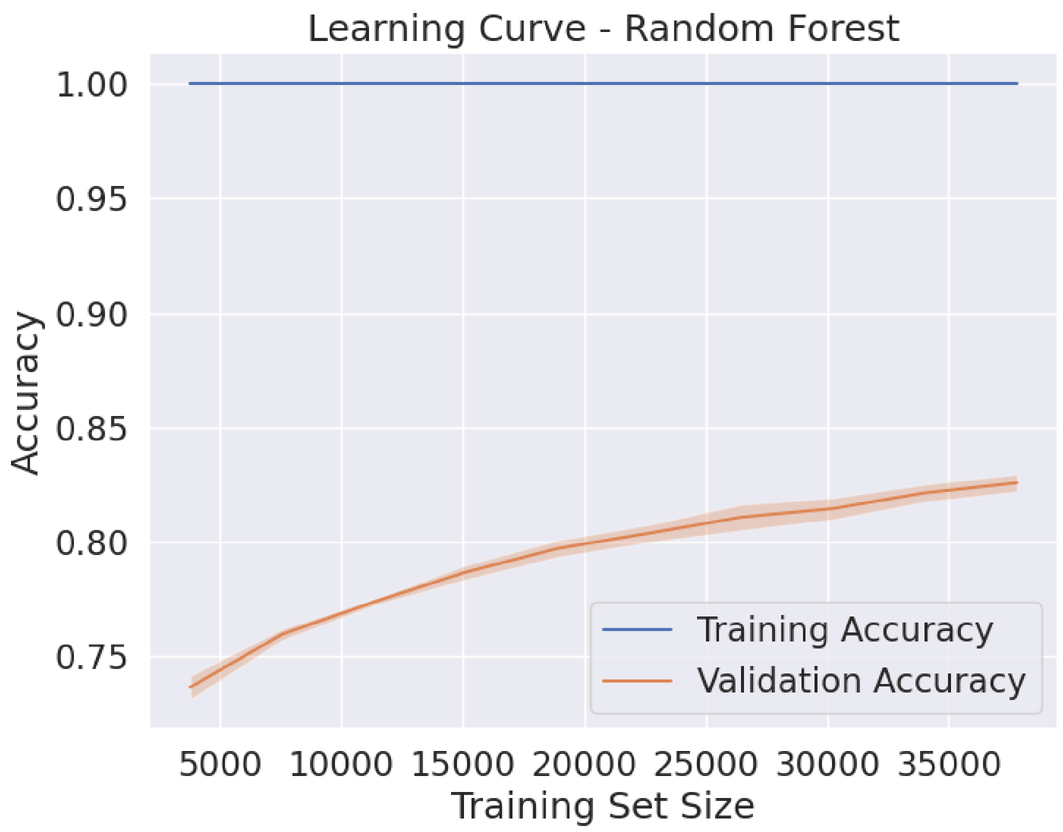
```
from sklearn.metrics import roc_curve, auc

# Get the predicted probabilities for each class
probas = random_forest.predict_proba(X_best_test)

# Compute the ROC curve for each class
fpr = dict()
tpr = dict()
roc_auc = dict()
for i in range(len(random_forest.classes_)):
    fpr[i], tpr[i], _ = roc_curve(y_best_test, probas[:, i], pos_label=random_forest.classes_[i])
    roc_auc[i] = auc(fpr[i], tpr[i])

# Plot the ROC curve for each class
plt.figure(figsize=(8, 6))
for i in range(len(random_forest.classes_)):
    plt.plot(fpr[i], tpr[i], label=f'Class {random_forest.classes_[i]} (AUC = {roc_auc[i]:.2f})')

plt.plot([0, 1], [0, 1], 'k--')  # Plot the diagonal line
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.05])
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('ROC Curve Random Forest')
plt.legend()
plt.grid(True)
plt.show()
```



Learning curve of the random forest is obtained as follows:



```
from sklearn.model_selection import learning_curve

# Define the training sizes for the learning curve
train_sizes, train_scores, test_scores = learning_curve(
    random_forest, X_best_train, y_best_train, cv=5, scoring='accuracy', train_sizes=np.linspace(0.1, 1.0, 10))

# Calculate the mean and standard deviation of the training and test scores
train_scores_mean = np.mean(train_scores, axis=1)
train_scores_std = np.std(train_scores, axis=1)
test_scores_mean = np.mean(test_scores, axis=1)
test_scores_std = np.std(test_scores, axis=1)

# Plot the learning curve
plt.figure(figsize=(8, 6))
plt.plot(train_sizes, train_scores_mean, label='Training Accuracy')
plt.fill_between(train_sizes, train_scores_mean - train_scores_std, train_scores_mean + train_scores_std, alpha=0.3)
plt.plot(train_sizes, test_scores_mean, label='Validation Accuracy')
plt.fill_between(train_sizes, test_scores_mean - test_scores_std, test_scores_mean + test_scores_std, alpha=0.3)
plt.xlabel('Training Set Size')
plt.ylabel('Accuracy')
plt.title('Learning Curve - Random Forest')
plt.legend()
plt.grid(True)
plt.show()
```


Learning Curve - Random Forest
Accuracy
Training Set Size
Training Accuracy
Validation Accuracy