"PREDICTING BANK CLIENT SUBSCRIPTIONS: A COMPARATIVE ANALYSIS OF MACHINE LEARNING MODELS"

Ensemble Learning and Calssification

Mini Project By

SHAFEENA FARHEEN

MLSC - CLASSIFICATION EXCERCISE

About the data set (Bank Client Data)

BANK CLIENT DATA:

age: Age of the client
duration: last contact duration, in seconds.

OTHER ATTRIBUTES:

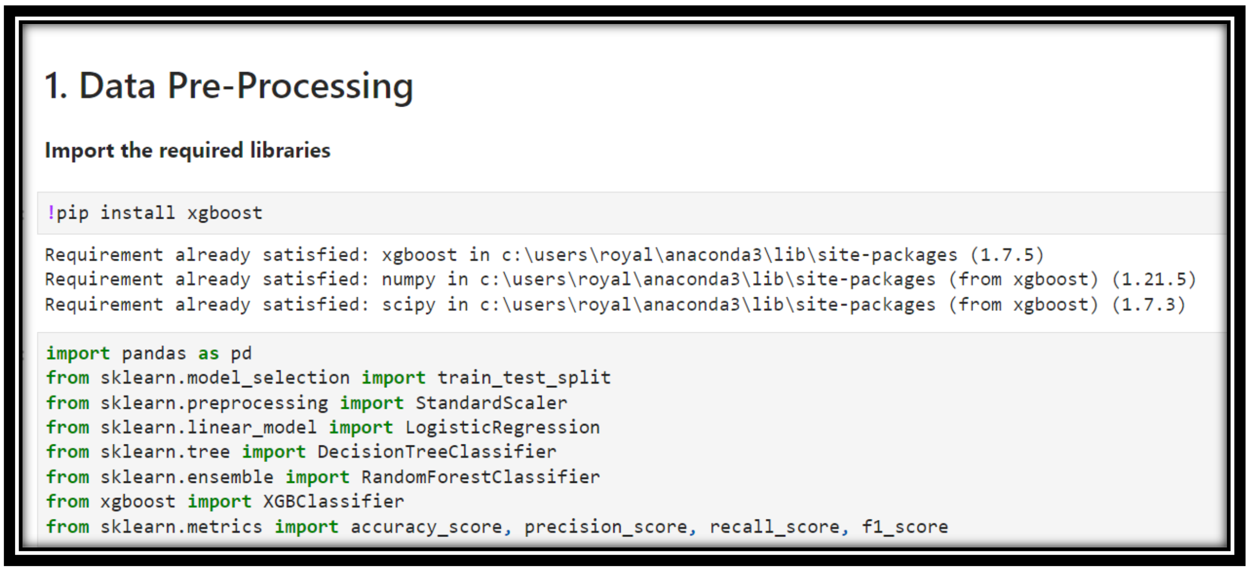
campaign: number of contacts performed during this campaign and for this client
pdays: number of days that passed by after the client was last contacted from a previous campaign (999 means client was not previously contacted)
previous: number of contacts performed before this campaign and for this client

SOCIAL AND ECONOMIC CONTEXT

emp.var.rate: employment variation rate - quarterly indicator
cons.price.idx: consumer price index - monthly indicator
cons.conf.idx: consumer confidence index - monthly indicator
euribor3m: euribor 3 month rate - daily indicator
nr.employed: number of employees - quarterly indicator

y - (Output variable) has the client subscribed a term deposit?

Table of Content



1. Data Pre-Processing

Import the required libraries

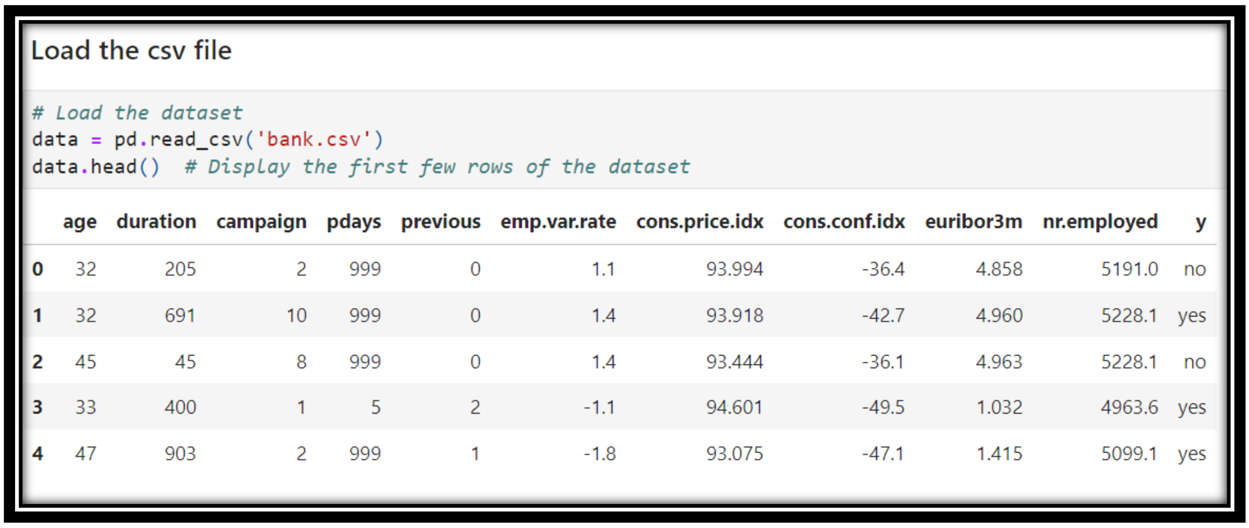
```
!pip install xgboost
```

```
Requirement already satisfied: xgboost in c:\users\royal\anaconda3\lib\site-packages (1.7.5)
Requirement already satisfied: numpy in c:\users\royal\anaconda3\lib\site-packages (from xgboost) (1.21.5)
Requirement already satisfied: scipy in c:\users\royal\anaconda3\lib\site-packages (from xgboost) (1.7.3)
```

```
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.linear_model import LogisticRegression
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import RandomForestClassifier
from xgboost import XGBClassifier
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score
```

Inference:

The required libraries are imported, including pandas, scikit-learn modules (train_test_split, StandardScaler, LogisticRegression, DecisionTreeClassifier, RandomForestClassifier), XGBoost (XGBClassifier), and evaluation metrics (accuracy_score, precision_score, recall_score, f1_score).



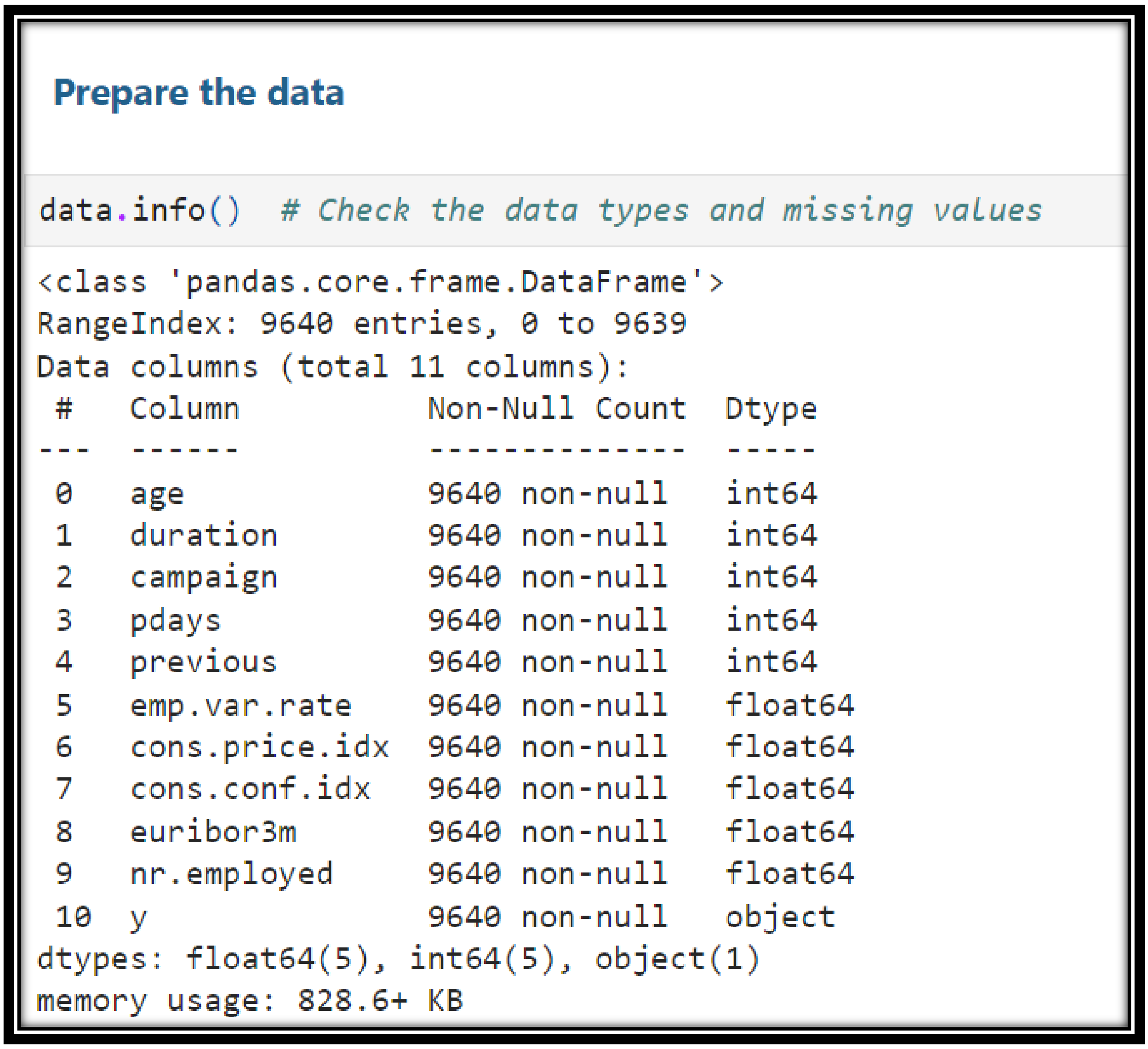
Load the csv file

```
# Load the dataset
data = pd.read_csv('bank.csv')
data.head()  # Display the first few rows of the dataset
```

	age	duration	campaign	pdays	previous	emp.var.rate	cons.price.idx	cons.conf.idx	euribor3m	nr.employed	y
0	32	205	2	999	0	1.1	93.994	-36.4	4.858	5191.0	no
1	32	691	10	999	0	1.4	93.918	-42.7	4.960	5228.1	yes
2	45	45	8	999	0	1.4	93.444	-36.1	4.963	5228.1	no
3	33	400	1	5	2	-1.1	94.601	-49.5	1.032	4963.6	yes
4	47	903	2	999	1	-1.8	93.075	-47.1	1.415	5099.1	yes

Inference:

 The bank client data is loaded from a CSV file into a pandas DataFrame called "data". The first few rows of the dataset are displayed using the "head()" function.



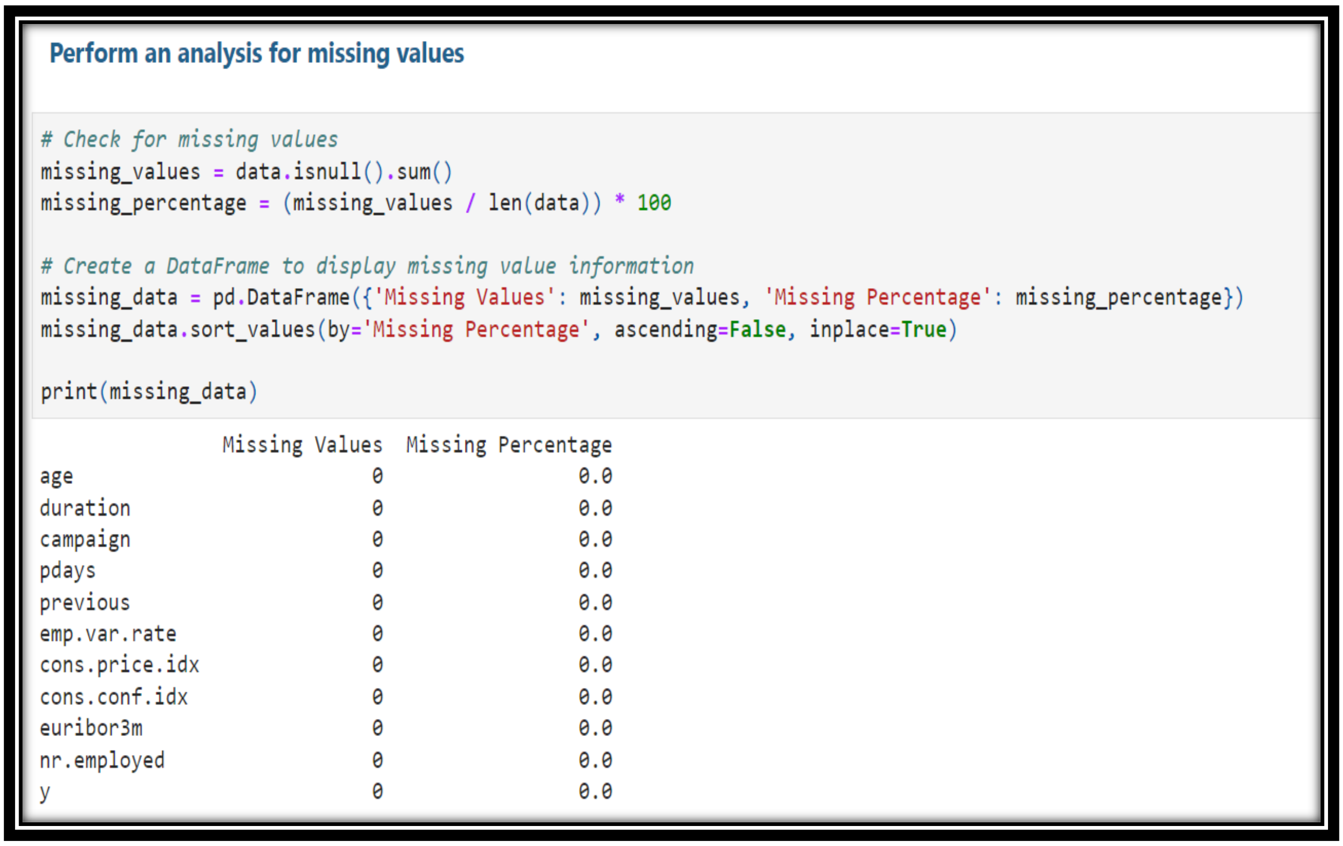
Prepare the data

```
data.info()  # Check the data types and missing values
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 9640 entries, 0 to 9639
Data columns (total 11 columns):
 #   Column          Non-Null Count  Dtype  
---  ------          --------------  -----  
 0   age             9640 non-null   int64  
 1   duration        9640 non-null   int64  
 2   campaign        9640 non-null   int64  
 3   pdays           9640 non-null   int64  
 4   previous        9640 non-null   int64  
 5   emp.var.rate    9640 non-null   float64
 6   cons.price.idx  9640 non-null   float64
 7   cons.conf.idx   9640 non-null   float64
 8   euribor3m       9640 non-null   float64
 9   nr.employed     9640 non-null   float64
 10  y               9640 non-null   object 
dtypes: float64(5), int64(5), object(1)
memory usage: 828.6+ KB
```

Inference:

 The data types and missing values are checked using the "info()" function. It shows that there are 9640 entries and no missing values in any column.



Perform an analysis for missing values

```
# Check for missing values
missing_values = data.isnull().sum()
missing_percentage = (missing_values / len(data)) * 100

# Create a DataFrame to display missing value information
missing_data = pd.DataFrame({'Missing Values': missing_values, 'Missing Percentage': missing_percentage})
missing_data.sort_values(by='Missing Percentage', ascending=False, inplace=True)

print(missing_data)
```

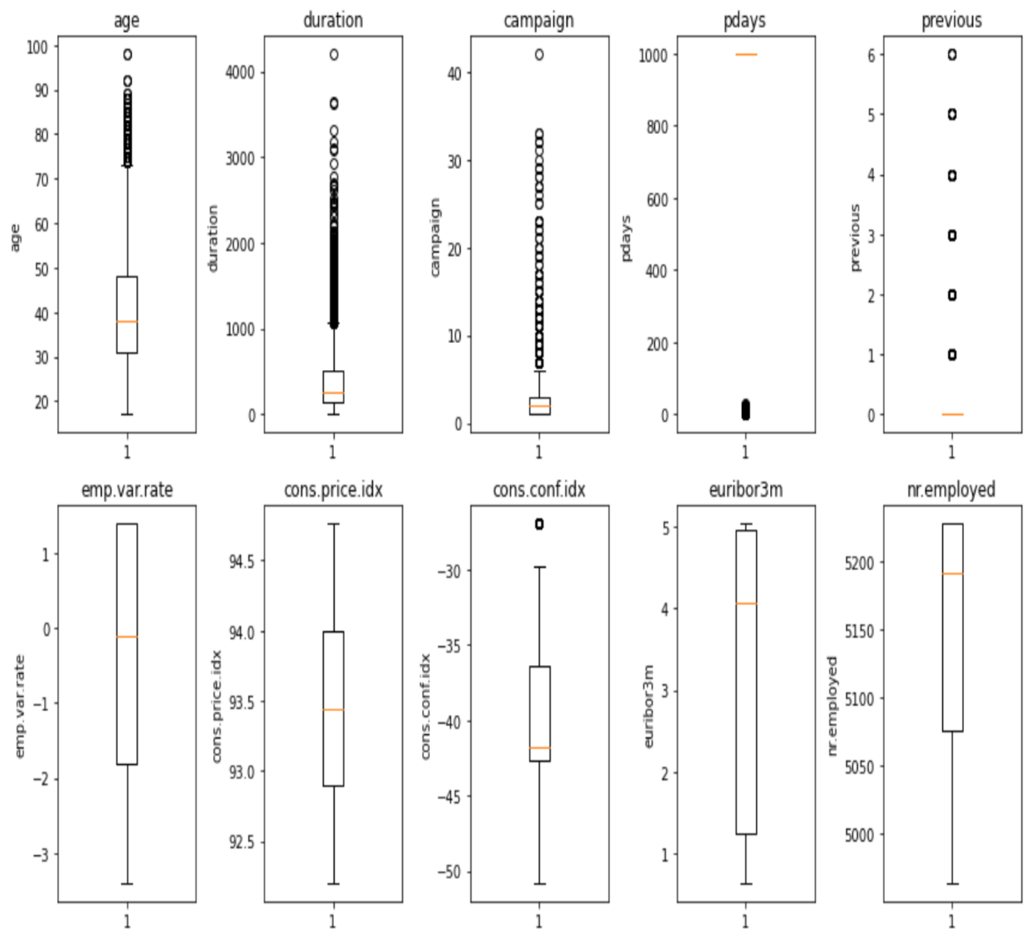
```
                Missing Values  Missing Percentage
age                          0                 0.0
duration                     0                 0.0
campaign                     0                 0.0
pdays                        0                 0.0
previous                     0                 0.0
emp.var.rate                 0                 0.0
cons.price.idx               0                 0.0
cons.conf.idx                0                 0.0
euribor3m                    0                 0.0
nr.employed                  0                 0.0
y                            0                 0.0
```

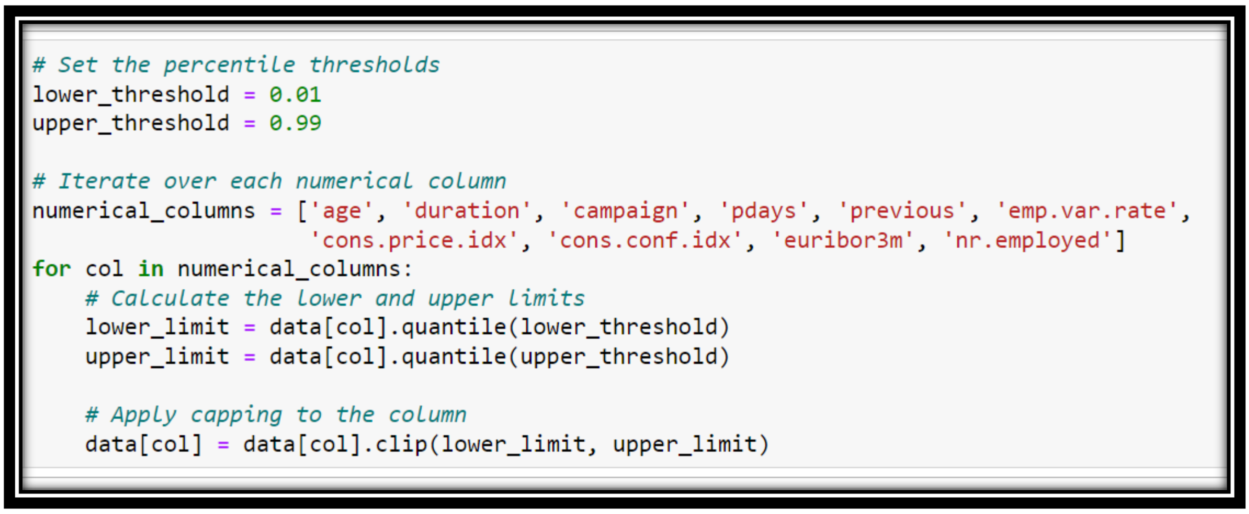
Inference:

 Missing values analysis is performed by calculating the number and percentage of missing values in each column. However, since there are no missing values in the dataset, all values in the "Missing Values" and "Missing Percentage" columns will be zero.

Remove the outliers (if any)

```
import matplotlib.pyplot as plt

# Select the numerical columns
numerical_columns = ['age', 'duration', 'campaign', 'pdays', 'previous', 'emp.var.rate', 'cons.price.idx', 'cons.conf.idx', 'euribor3m', 'nr.employ

# Create box plots for each numerical column
fig, axes = plt.subplots(nrows=2, ncols=5, figsize=(12, 8))
axes = axes.flatten()

for i, col in enumerate(numerical_columns):
    ax = axes[i]
    ax.boxplot(data[col])
    ax.set_title(col)
    ax.set_ylabel(col)

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

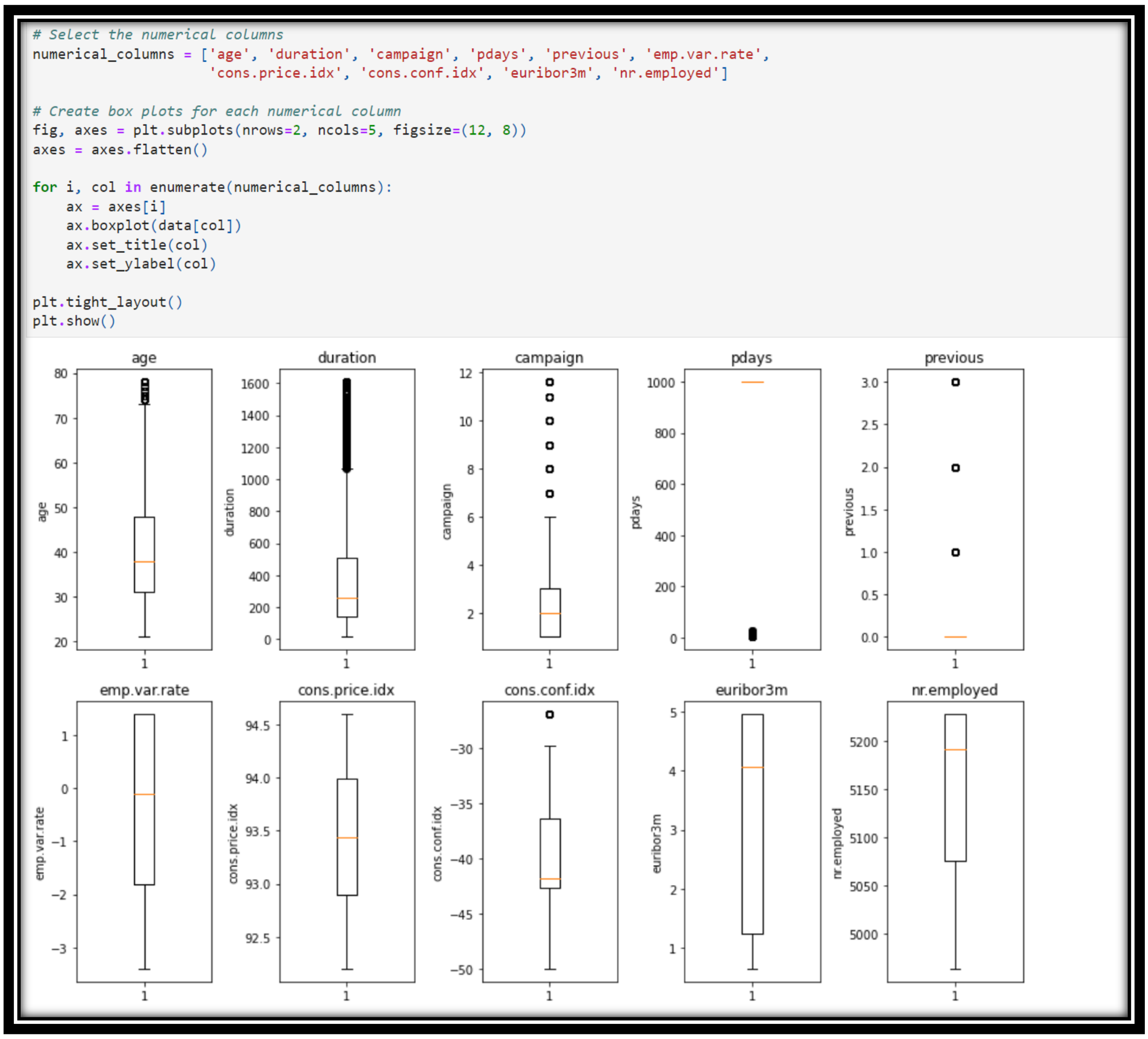




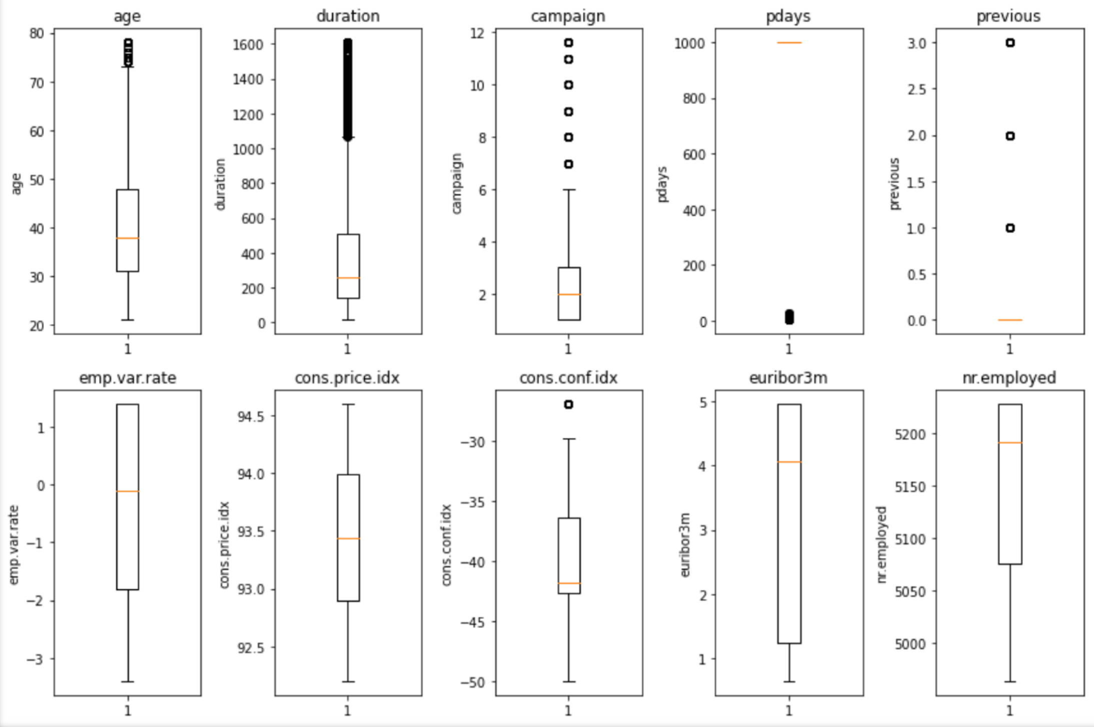
```python
# Set the percentile thresholds
lower_threshold = 0.01
upper_threshold = 0.99

# Iterate over each numerical column
numerical_columns = ['age', 'duration', 'campaign', 'pdays', 'previous', 'emp.var.rate',
                     'cons.price.idx', 'cons.conf.idx', 'euribor3m', 'nr.employed']
for col in numerical_columns:
    # Calculate the lower and upper limits
    lower_limit = data[col].quantile(lower_threshold)
    upper_limit = data[col].quantile(upper_threshold)

    # Apply capping to the column
    data[col] = data[col].clip(lower_limit, upper_limit)
```

Inference:

 Outliers are identified and handled using box plots. Initially, box plots are created for each numerical column to visualize the presence of outliers. Then, percentile thresholds (lower and upper) are set to 0.01 and 0.99, respectively. Next, a loop iterates over each numerical column, calculating the lower and upper limits using the quantile function. Finally, the clip function is used to apply capping to the column, replacing outliers with the corresponding lower and upper limits.

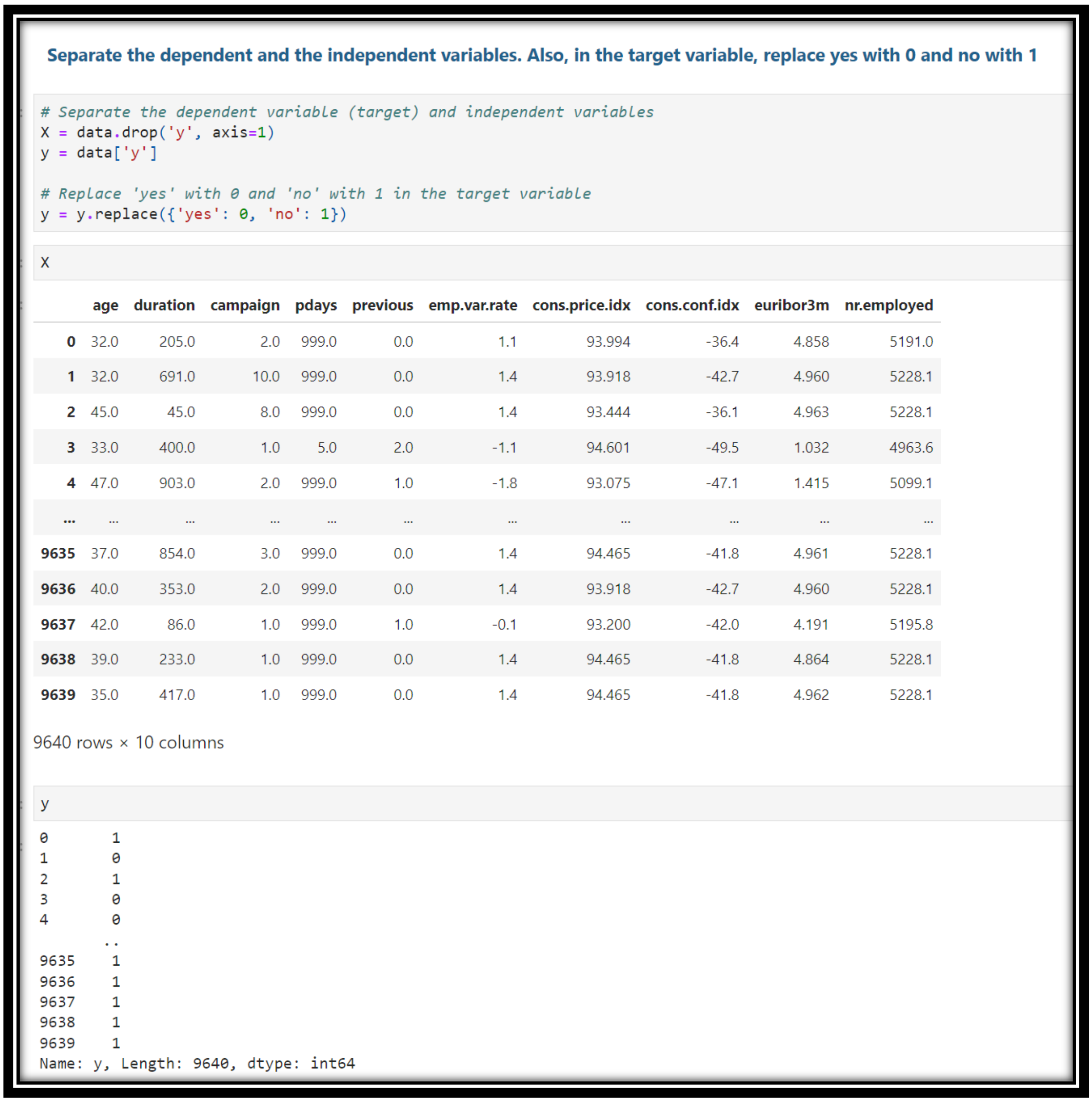


```
# Select the numerical columns
numerical_columns = ['age', 'duration', 'campaign', 'pdays', 'previous', 'emp.var.rate',
                     'cons.price.idx', 'cons.conf.idx', 'euribor3m', 'nr.employed']

# Create box plots for each numerical column
fig, axes = plt.subplots(nrows=2, ncols=5, figsize=(12, 8))
axes = axes.flatten()

for i, col in enumerate(numerical_columns):
    ax = axes[i]
    ax.boxplot(data[col])
    ax.set_title(col)
    ax.set_ylabel(col)

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



Inference:

 After removing outliers, new box plots are created to visualize the updated distributions of numerical columns.



Separate the dependent and the independent variables. Also, in the target variable, replace yes with 0 and no with 1

```
# Separate the dependent variable (target) and independent variables
X = data.drop('y', axis=1)
y = data['y']

# Replace 'yes' with 0 and 'no' with 1 in the target variable
y = y.replace({'yes': 0, 'no': 1})
```

```
X
```

	age	duration	campaign	pdays	previous	emp.var.rate	cons.price.idx	cons.conf.idx	euribor3m	nr.employed
0	32.0	205.0	2.0	999.0	0.0	1.1	93.994	-36.4	4.858	5191.0
1	32.0	691.0	10.0	999.0	0.0	1.4	93.918	-42.7	4.960	5228.1
2	45.0	45.0	8.0	999.0	0.0	1.4	93.444	-36.1	4.963	5228.1
3	33.0	400.0	1.0	5.0	2.0	-1.1	94.601	-49.5	1.032	4963.6
4	47.0	903.0	2.0	999.0	1.0	-1.8	93.075	-47.1	1.415	5099.1
...	...	...	...	...	...	...	...	...	...	...
9635	37.0	854.0	3.0	999.0	0.0	1.4	94.465	-41.8	4.961	5228.1
9636	40.0	353.0	2.0	999.0	0.0	1.4	93.918	-42.7	4.960	5228.1
9637	42.0	86.0	1.0	999.0	1.0	-0.1	93.200	-42.0	4.191	5195.8
9638	39.0	233.0	1.0	999.0	0.0	1.4	94.465	-41.8	4.864	5228.1
9639	35.0	417.0	1.0	999.0	0.0	1.4	94.465	-41.8	4.962	5228.1

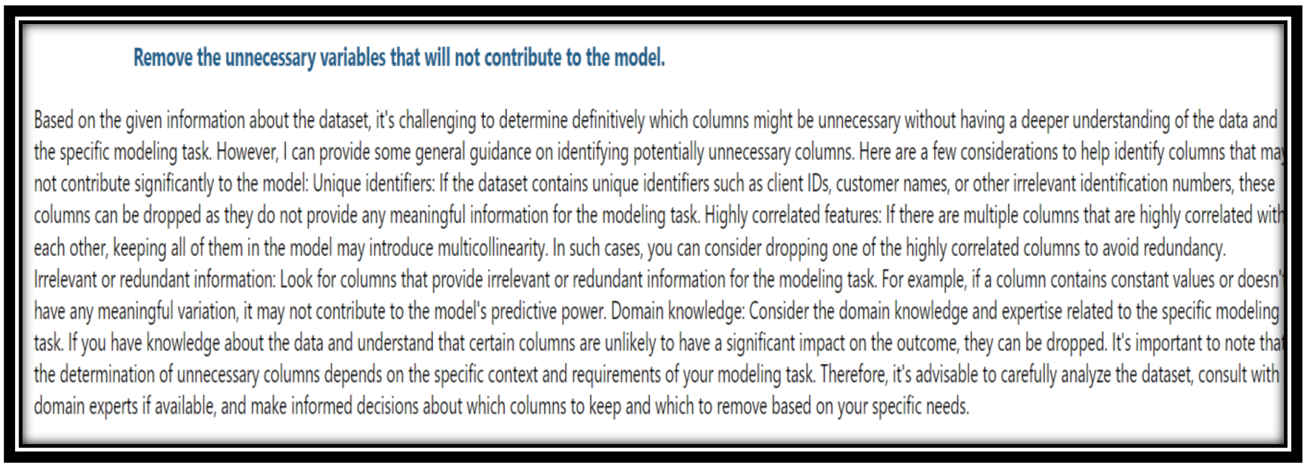
9640 rows × 10 columns

```
y
```

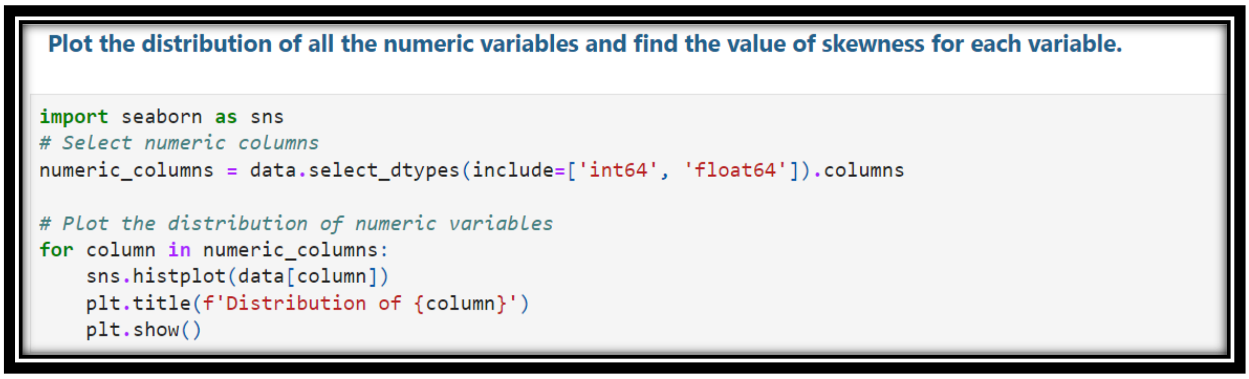
```
0       1
1       0
2       1
3       0
4       0
       ..
9635    1
9636    1
9637    1
9638    1
9639    1
Name: y, Length: 9640, dtype: int64
```

Inference:

 The dependent variable (target) is separated from the independent variables by assigning the relevant columns to the variables "X" and "y", respectively. Additionally, the target variable is transformed by replacing 'yes' with 0 and 'no' with 1 using the replace function.


Remove the unnecessary variables that will not contribute to the model.

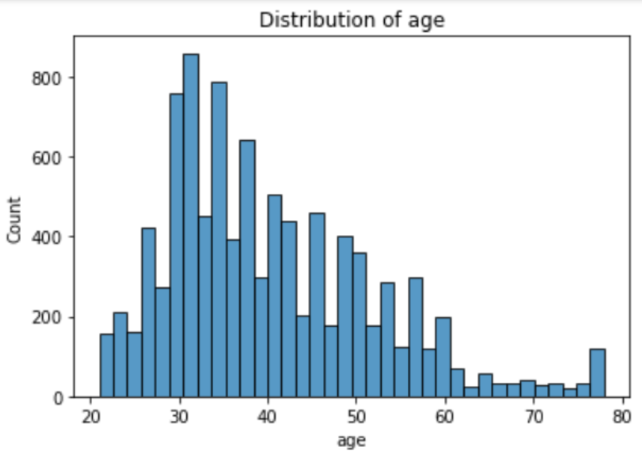
Based on the given information about the dataset, it's challenging to determine definitively which columns might be unnecessary without having a deeper understanding of the data and the specific modeling task. However, I can provide some general guidance on identifying potentially unnecessary columns. Here are a few considerations to help identify columns that may not contribute significantly to the model: Unique identifiers: If the dataset contains unique identifiers such as client IDs, customer names, or other irrelevant identification numbers, these columns can be dropped as they do not provide any meaningful information for the modeling task. Highly correlated features: If there are multiple columns that are highly correlated with each other, keeping all of them in the model may introduce multicollinearity. In such cases, you can consider dropping one of the highly correlated columns to avoid redundancy. Irrelevant or redundant information: Look for columns that provide irrelevant or redundant information for the modeling task. For example, if a column contains constant values or doesn' have any meaningful variation, it may not contribute to the model's predictive power. Domain knowledge: Consider the domain knowledge and expertise related to the specific modeling task. If you have knowledge about the data and understand that certain columns are unlikely to have a significant impact on the outcome, they can be dropped. It's important to note that the determination of unnecessary columns depends on the specific context and requirements of your modeling task. Therefore, it's advisable to carefully analyze the dataset, consult with domain experts if available, and make informed decisions about which columns to keep and which to remove based on your specific needs.

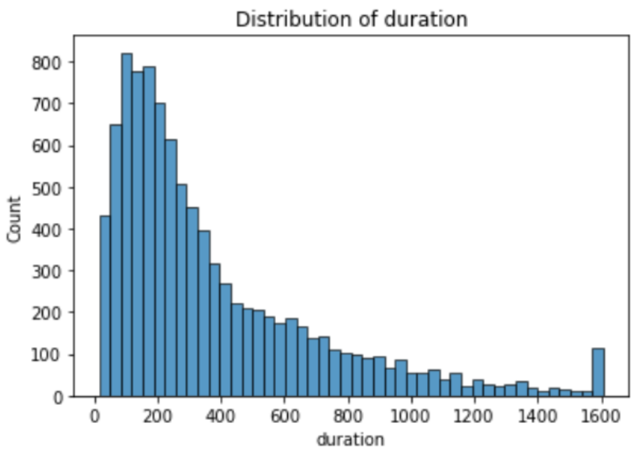

Plot the distribution of all the numeric variables and find the value of skewness for each variable.

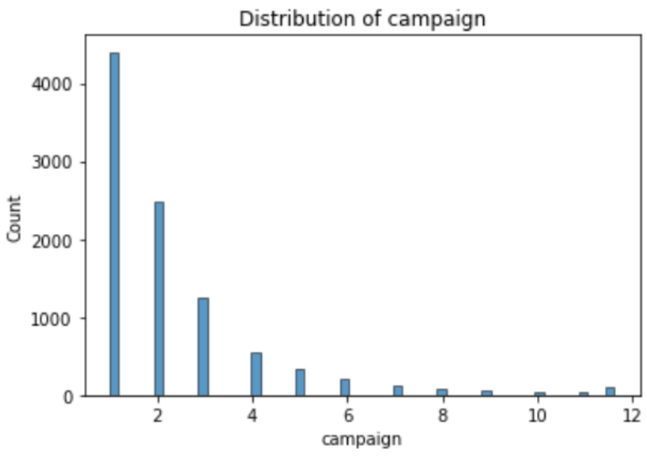
```
import seaborn as sns
# Select numeric columns
numeric_columns = data.select_dtypes(include=['int64', 'float64']).columns

# Plot the distribution of numeric variables
for column in numeric_columns:
    sns.histplot(data[column])
    plt.title(f'Distribution of {column}')
    plt.show()
```

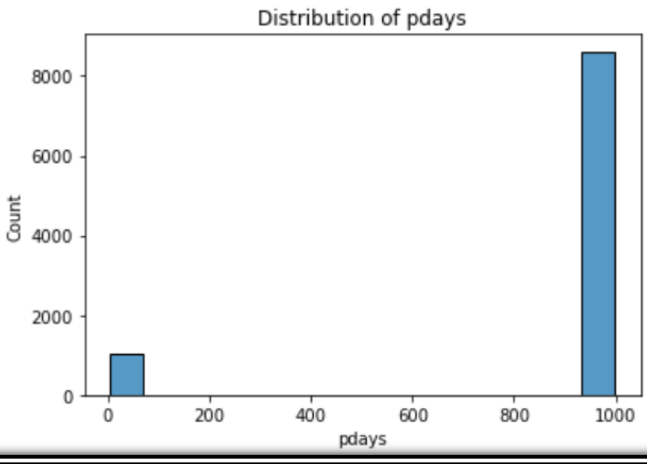
Inference:

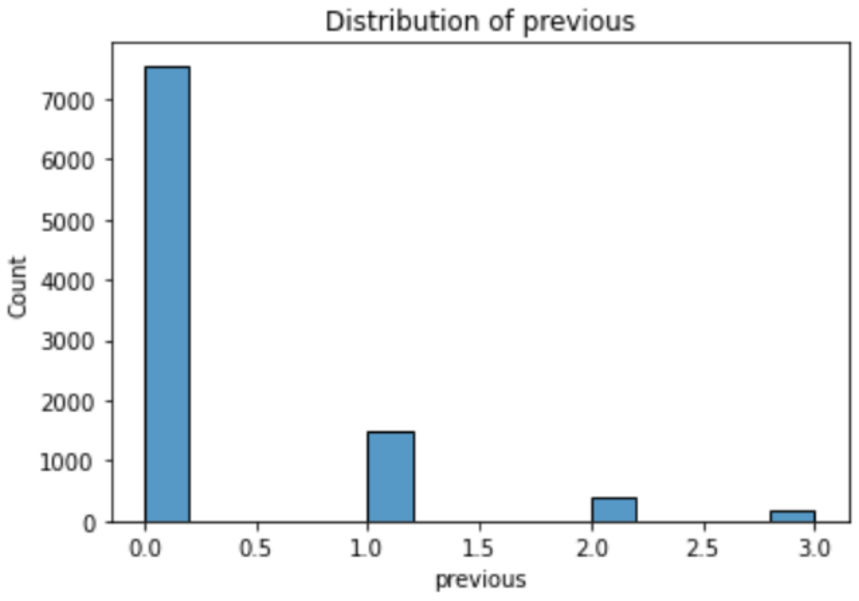
 The distribution of numeric variables is plotted using histograms. For each numeric column, a histogram is generated using the seaborn library's histplot function.

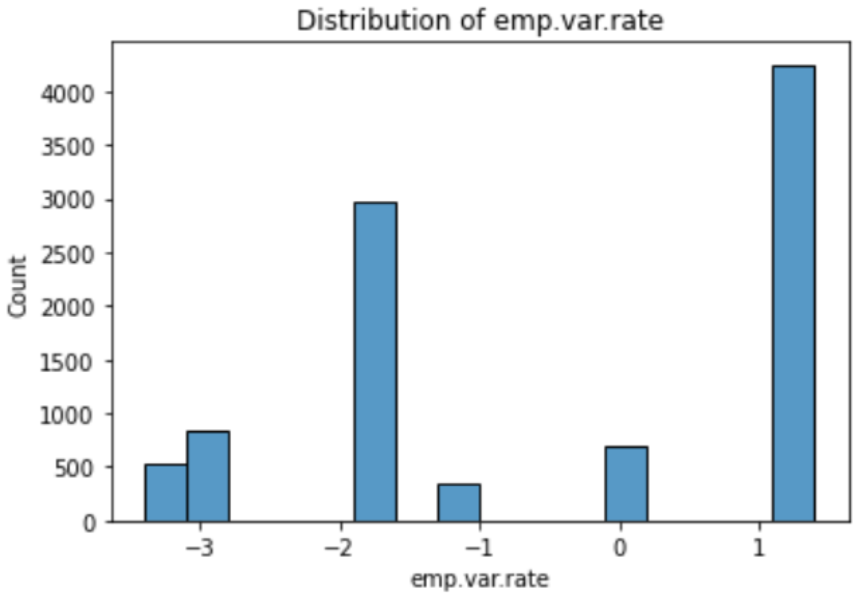

Distribution of age
Count
0
200
400
600
800
20
30
40
50
60
70
80
age

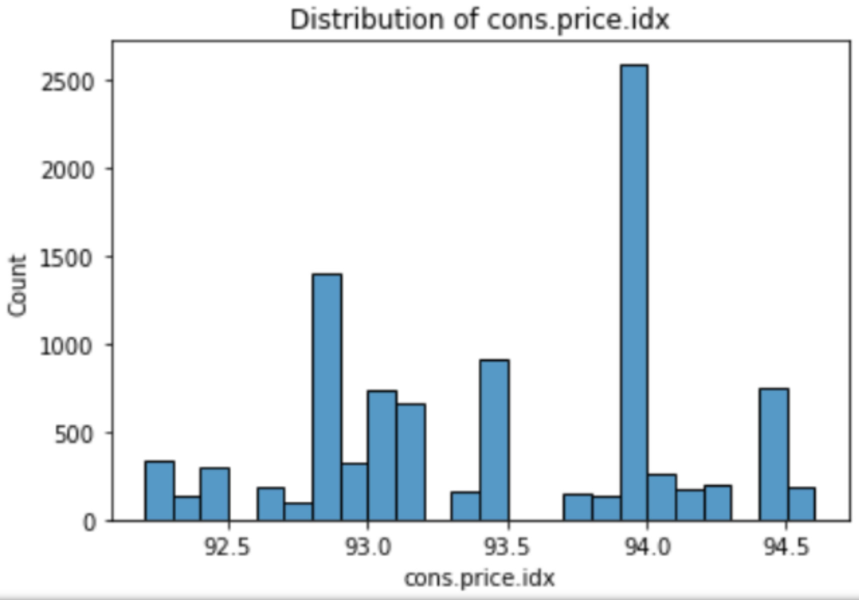

Distribution of duration
Count
0
100
200
300
400
500
600
700
800
0
200
400
600
800
1000
1200
1400
1600
duration

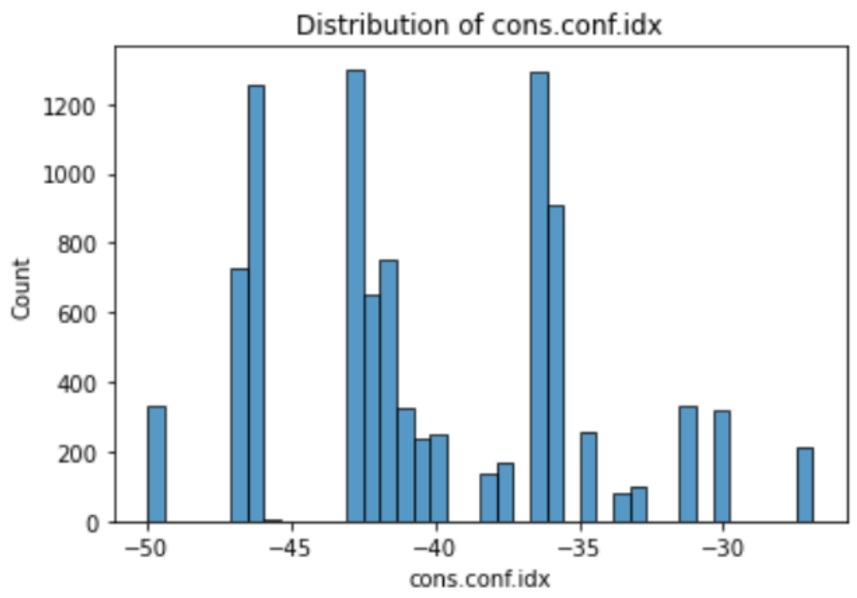

Distribution of campaign
Count
0
1000
2000
3000
4000
2
4
6
8
10
12
campaign

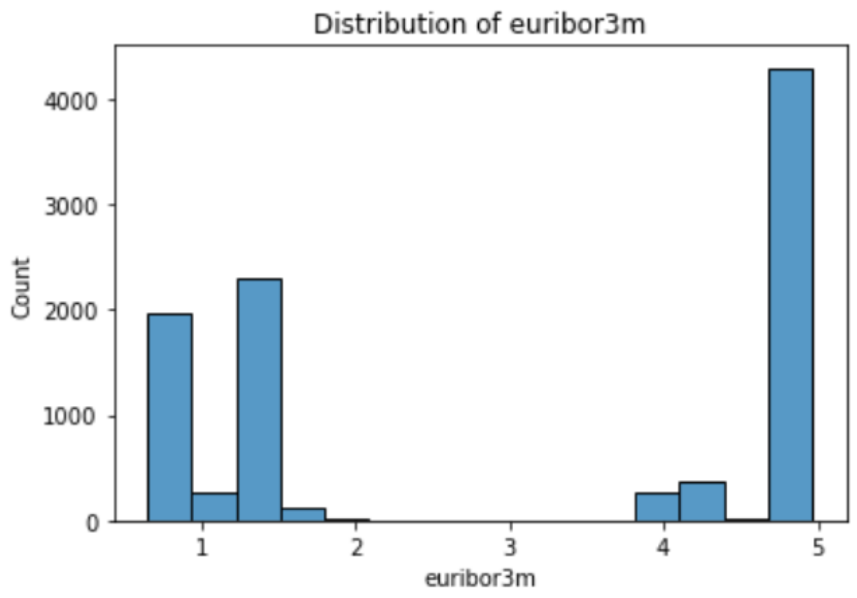

Distribution of pdays
Count
0
2000
4000
6000
8000
0
200
400
600
800
1000
pdays

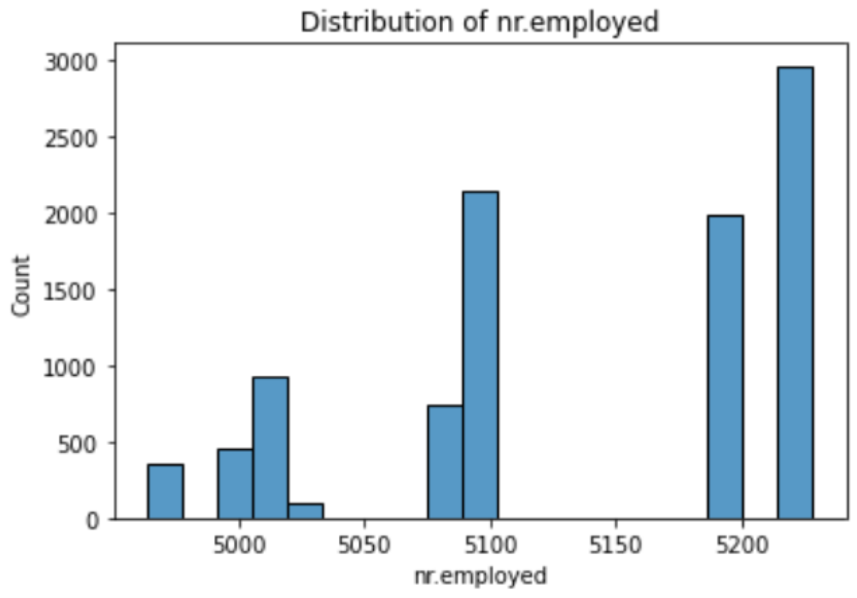

Distribution of previous
Count
previous
0
1000
2000
3000
4000
5000
6000
7000
0.0
0.5
1.0
1.5
2.0
2.5
3.0

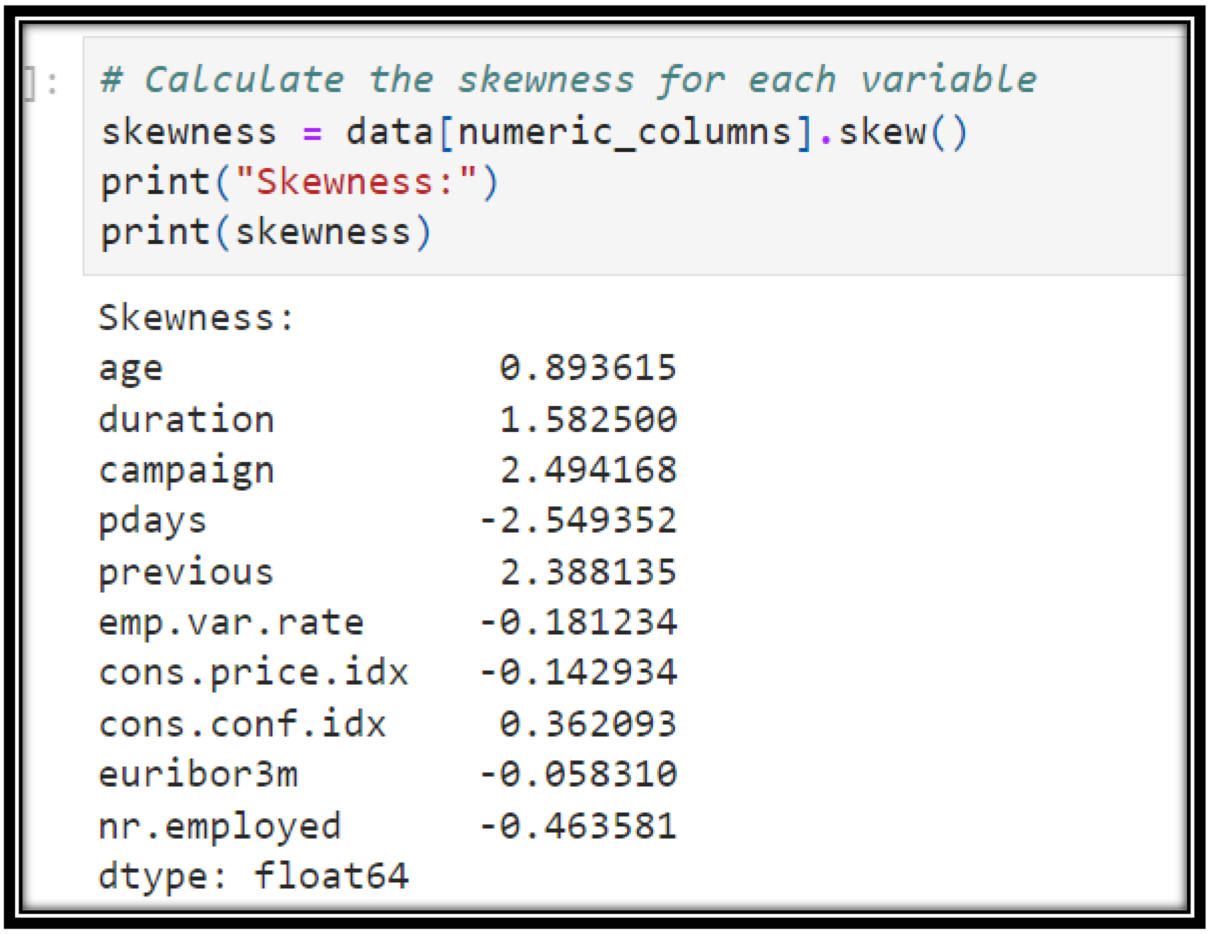

Distribution of emp.var.rate
Count
emp.var.rate
0
500
1000
1500
2000
2500
3000
3500
4000
−3
−2
−1
0
1


Distribution of cons.price.idx
Count
cons.price.idx
0
500
1000
1500
2000
2500
92.5
93.0
93.5
94.0
94.5


Distribution of cons.conf.idx
Count
1200
1000
800
600
400
200
0
−50
−45
−40
−35
−30
cons.conf.idx


Distribution of euribor3m
Count
4000
3000
2000
1000
0
1
2
3
4
5
euribor3m


Distribution of nr.employed
Count
3000
2500
2000
1500
1000
500
0
5000
5050
5100
5150
5200
nr.employed

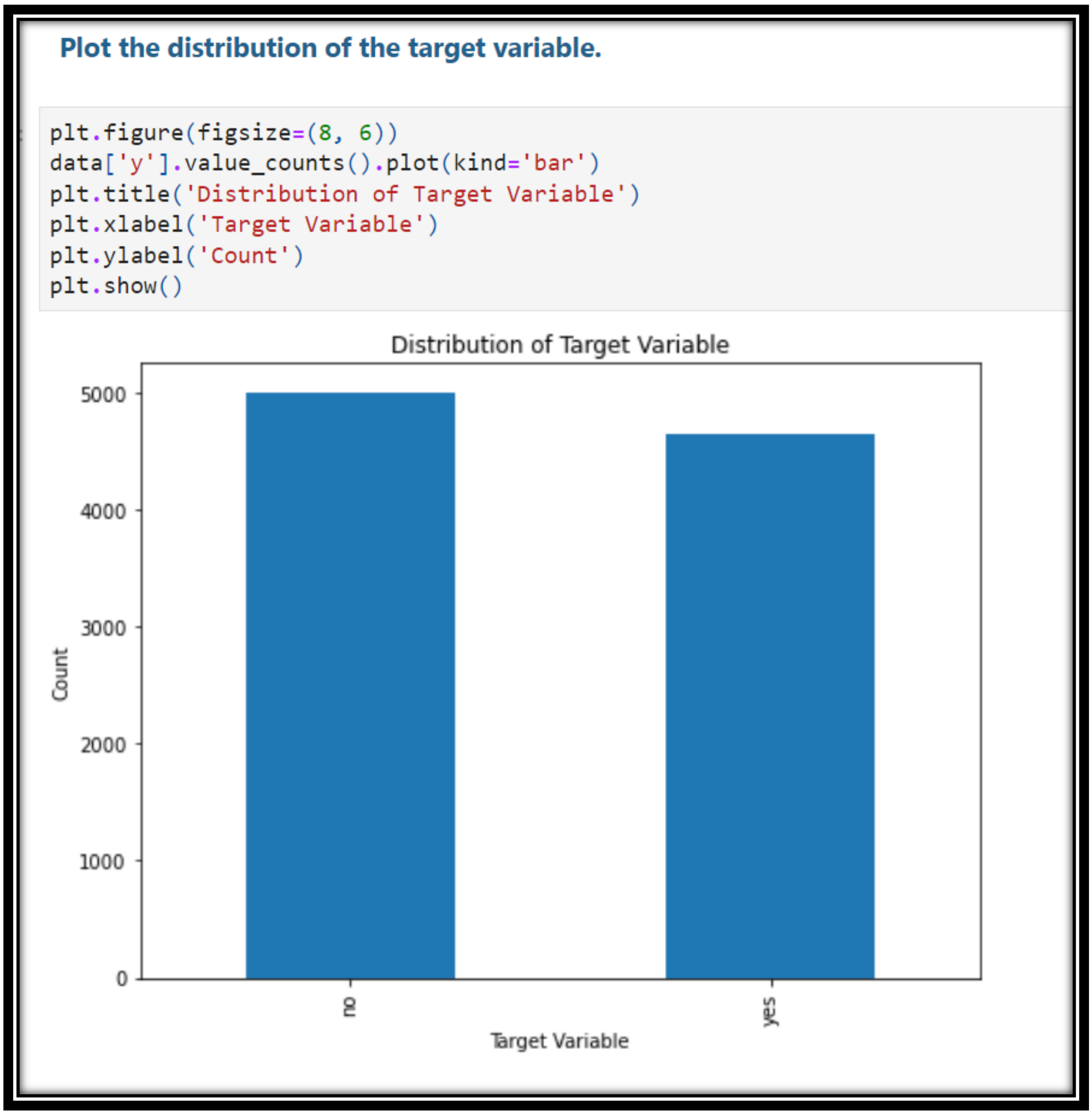


```
# Calculate the skewness for each variable
skewness = data[numeric_columns].skew()
print("Skewness:")
print(skewness)
```

```
Skewness:
age                0.893615
duration           1.582500
campaign           2.494168
pdays             -2.549352
previous           2.388135
emp.var.rate      -0.181234
cons.price.idx    -0.142934
cons.conf.idx      0.362093
euribor3m         -0.058310
nr.employed       -0.463581
dtype: float64
```

Inference:

 The skewness of each numeric variable is calculated using the skew function. Skewness measures the asymmetry of the distribution. Positive skewness indicates a longer tail on the right side, while negative skewness indicates a longer tail on the left side.

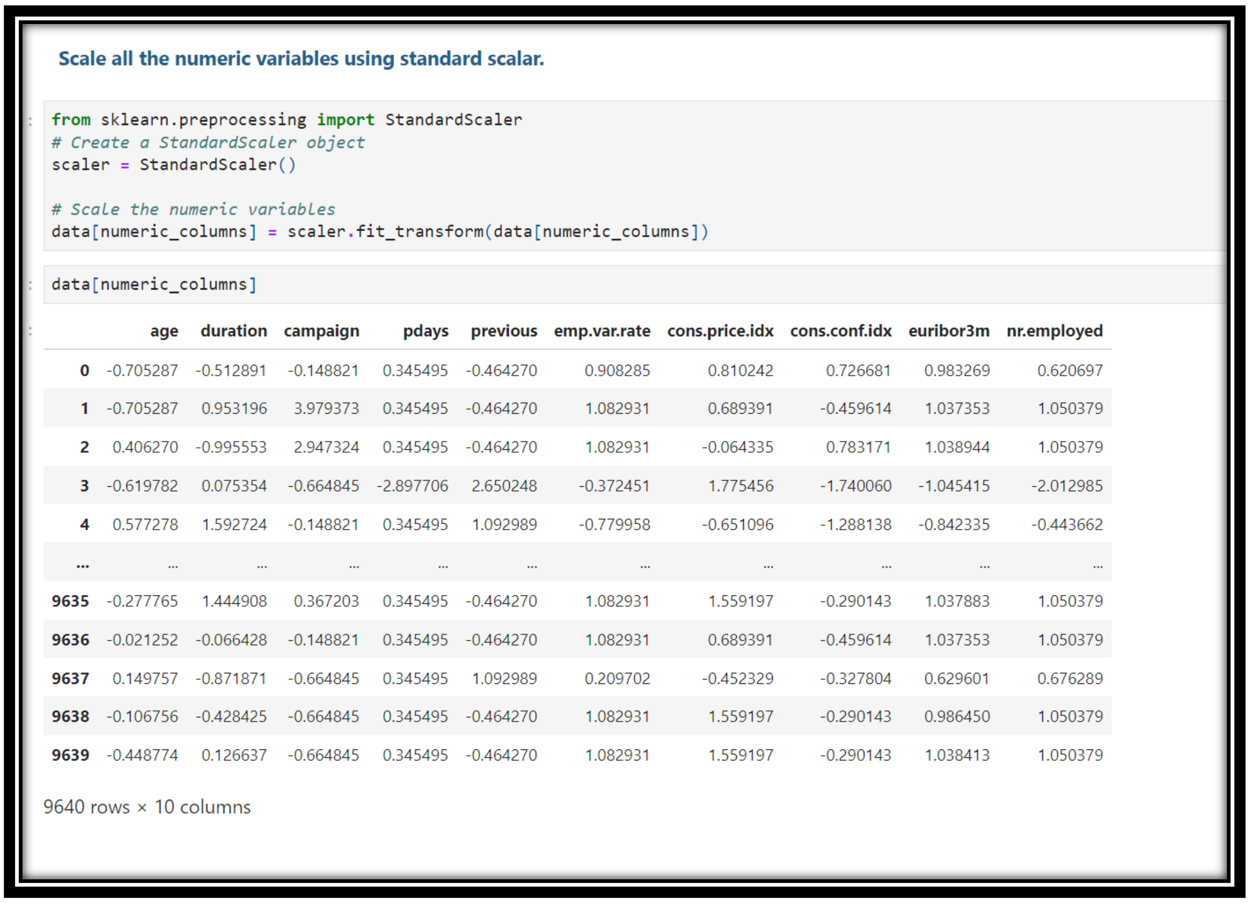


Plot the distribution of the target variable.

```
plt.figure(figsize=(8, 6))
data['y'].value_counts().plot(kind='bar')
plt.title('Distribution of Target Variable')
plt.xlabel('Target Variable')
plt.ylabel('Count')
plt.show()
```

Inference:

 The distribution of the target variable is visualized using a bar plot, showing the count of each class (0 and 1).



Scale all the numeric variables using standard scalar.

```
from sklearn.preprocessing import StandardScaler
# Create a StandardScaler object
scaler = StandardScaler()

# Scale the numeric variables
data[numeric_columns] = scaler.fit_transform(data[numeric_columns])
```

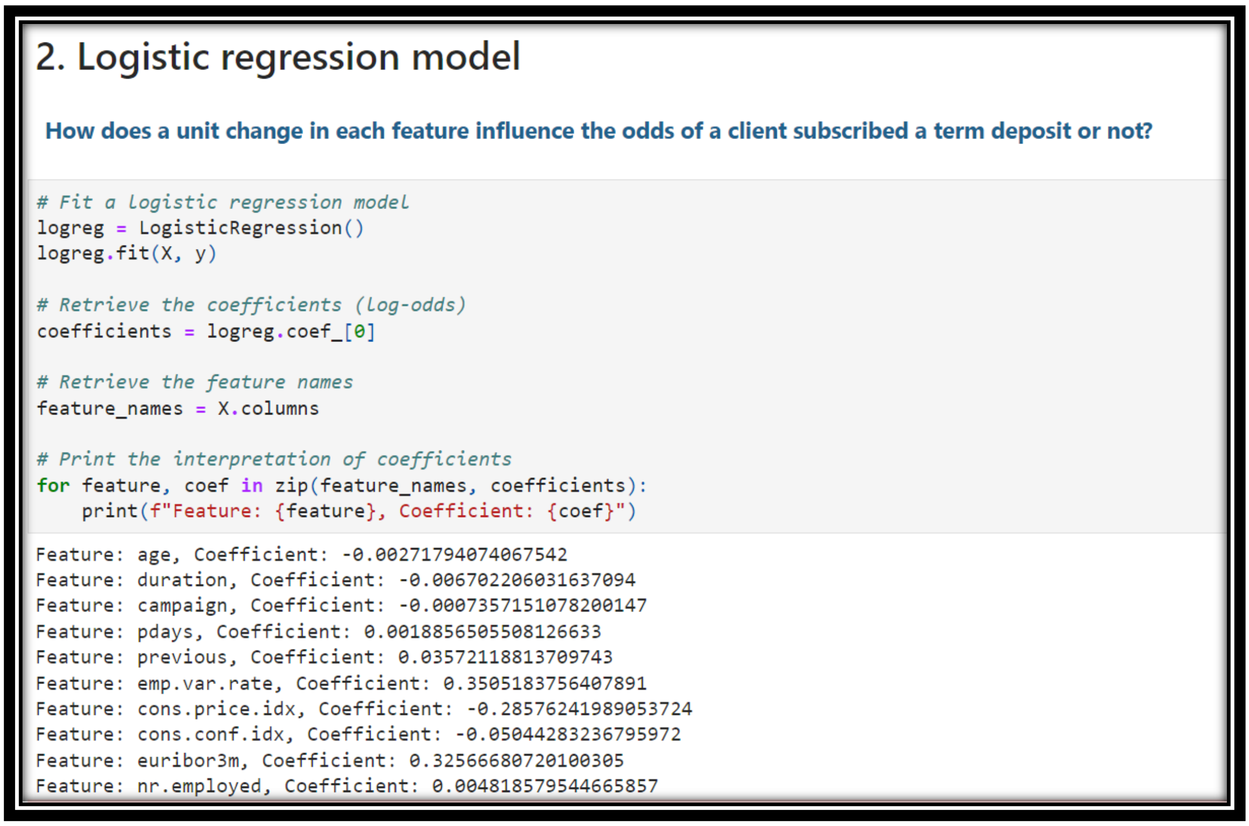
```
data[numeric_columns]
```

	age	duration	campaign	pdays	previous	emp.var.rate	cons.price.idx	cons.conf.idx	euribor3m	nr.employed
0	-0.705287	-0.512891	-0.148821	0.345495	-0.464270	0.908285	0.810242	0.726681	0.983269	0.620697
1	-0.705287	0.953196	3.979373	0.345495	-0.464270	1.082931	0.689391	-0.459614	1.037353	1.050379
2	0.406270	-0.995553	2.947324	0.345495	-0.464270	1.082931	-0.064335	0.783171	1.038944	1.050379
3	-0.619782	0.075354	-0.664845	-2.897706	2.650248	-0.372451	1.775456	-1.740060	-1.045415	-2.012985
4	0.577278	1.592724	-0.148821	0.345495	1.092989	-0.779958	-0.651096	-1.288138	-0.842335	-0.443662
...	...	...	...	...	...	...	...	...	...	...
9635	-0.277765	1.444908	0.367203	0.345495	-0.464270	1.082931	1.559197	-0.290143	1.037883	1.050379
9636	-0.021252	-0.066428	-0.148821	0.345495	-0.464270	1.082931	0.689391	-0.459614	1.037353	1.050379
9637	0.149757	-0.871871	-0.664845	0.345495	1.092989	0.209702	-0.452329	-0.327804	0.629601	0.676289
9638	-0.106756	-0.428425	-0.664845	0.345495	-0.464270	1.082931	1.559197	-0.290143	0.986450	1.050379
9639	-0.448774	0.126637	-0.664845	0.345495	-0.464270	1.082931	1.559197	-0.290143	1.038413	1.050379

9640 rows × 10 columns

Inference:

 Standard scaling is applied to all the numeric variables using the StandardScaler class from scikit-learn. This step ensures that all the variables are transformed to have a mean of 0 and a standard deviation of 1, which is important for many machine learning algorithms that assume standardized features.



2. Logistic regression model

How does a unit change in each feature influence the odds of a client subscribed a term deposit or not?

```
# Fit a logistic regression model
logreg = LogisticRegression()
logreg.fit(X, y)

# Retrieve the coefficients (log-odds)
coefficients = logreg.coef_[0]

# Retrieve the feature names
feature_names = X.columns

# Print the interpretation of coefficients
for feature, coef in zip(feature_names, coefficients):
    print(f"Feature: {feature}, Coefficient: {coef}")
```

```
Feature: age, Coefficient: -0.00271794074067542
Feature: duration, Coefficient: -0.006702206031637094
Feature: campaign, Coefficient: -0.0007357151078200147
Feature: pdays, Coefficient: 0.0018856505508126633
Feature: previous, Coefficient: 0.03572118813709743
Feature: emp.var.rate, Coefficient: 0.3505183756407891
Feature: cons.price.idx, Coefficient: -0.28576241989053724
Feature: cons.conf.idx, Coefficient: -0.05044283236795972
Feature: euribor3m, Coefficient: 0.32566680720100305
Feature: nr.employed, Coefficient: 0.004818579544665857
```

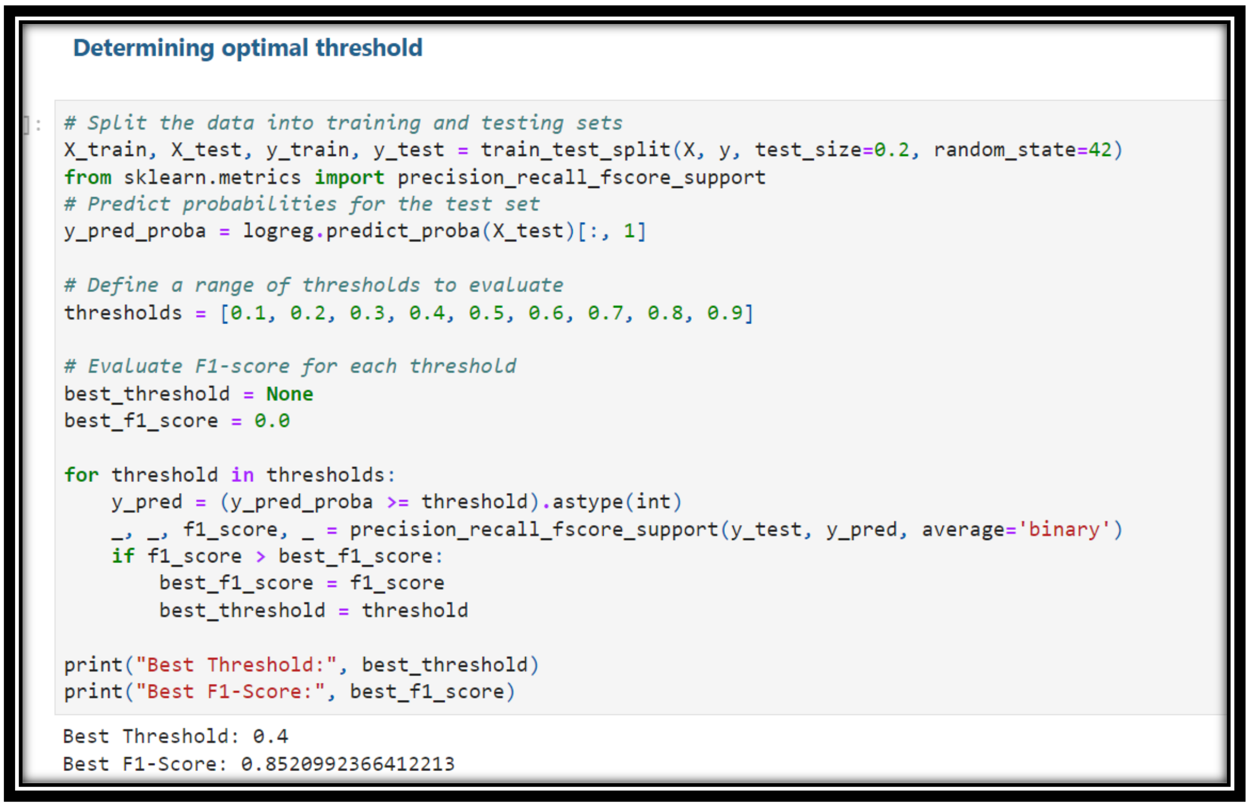
Inference:

Logistic Regression Model:

The logistic regression model is fitted using the data, allowing us to understand how a unit change in each feature influences the odds of a client subscribing to a term deposit.

The coefficients obtained from the model represent the log-odds of the target variable.

Positive coefficients indicate that an increase in the corresponding feature value increases the odds of a client subscribing to a term deposit, while negative coefficients indicate the opposite.


Determining optimal threshold

```
# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
from sklearn.metrics import precision_recall_fscore_support
# Predict probabilities for the test set
y_pred_proba = logreg.predict_proba(X_test)[:, 1]

# Define a range of thresholds to evaluate
thresholds = [0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9]

# Evaluate F1-score for each threshold
best_threshold = None
best_f1_score = 0.0

for threshold in thresholds:
    y_pred = (y_pred_proba >= threshold).astype(int)
    _, _, f1_score, _ = precision_recall_fscore_support(y_test, y_pred, average='binary')
    if f1_score > best_f1_score:
        best_f1_score = f1_score
        best_threshold = threshold

print("Best Threshold:", best_threshold)
print("Best F1-Score:", best_f1_score)
```

```
Best Threshold: 0.4
Best F1-Score: 0.8520992366412213
```

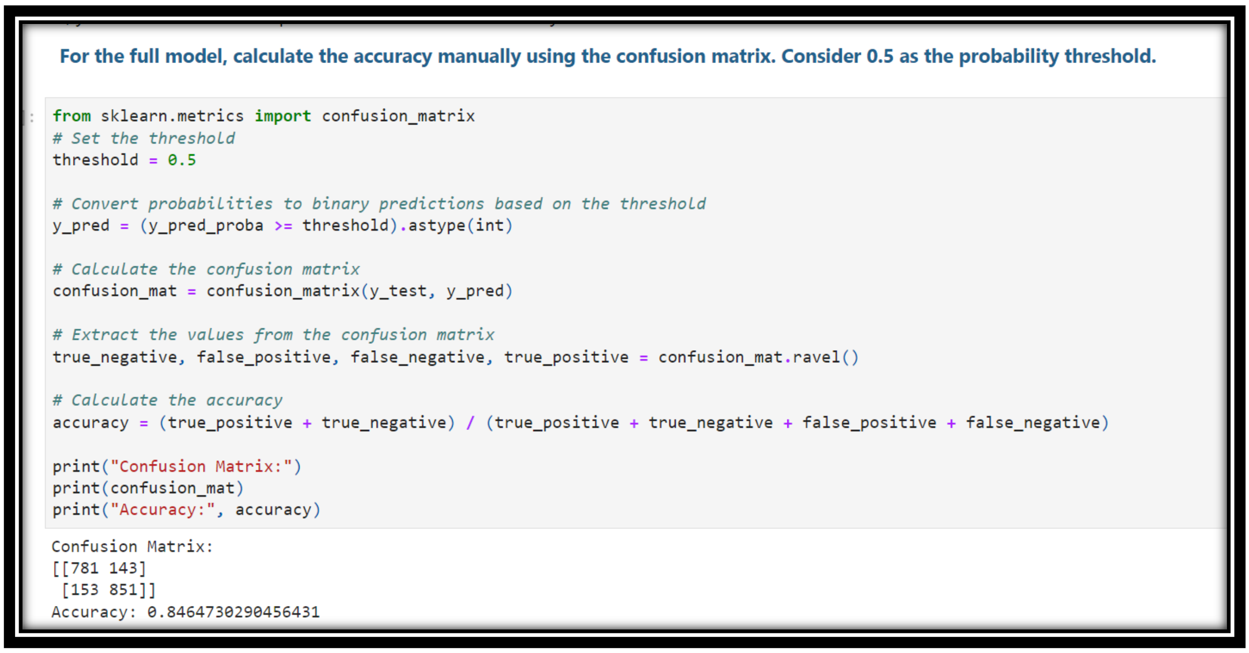
Inference:

Determining Optimal Threshold:

The logistic regression model's predicted probabilities are used to evaluate different thresholds.

By comparing the F1-score at each threshold, the optimal threshold that maximizes the desired evaluation metric (F1-score in this case) is determined.

In this specific case, the best threshold is found to be 0.4, resulting in an F1-score of 0.852.


For the full model, calculate the accuracy manually using the confusion matrix. Consider 0.5 as the probability threshold.

```
from sklearn.metrics import confusion_matrix
# Set the threshold
threshold = 0.5

# Convert probabilities to binary predictions based on the threshold
y_pred = (y_pred_proba >= threshold).astype(int)

# Calculate the confusion matrix
confusion_mat = confusion_matrix(y_test, y_pred)

# Extract the values from the confusion matrix
true_negative, false_positive, false_negative, true_positive = confusion_mat.ravel()

# Calculate the accuracy
accuracy = (true_positive + true_negative) / (true_positive + true_negative + false_positive + false_negative)

print("Confusion Matrix:")
print(confusion_mat)
print("Accuracy:", accuracy)
```

```
Confusion Matrix:
[[781 143]
 [153 851]]
Accuracy: 0.8464730290456431
```

Inference:

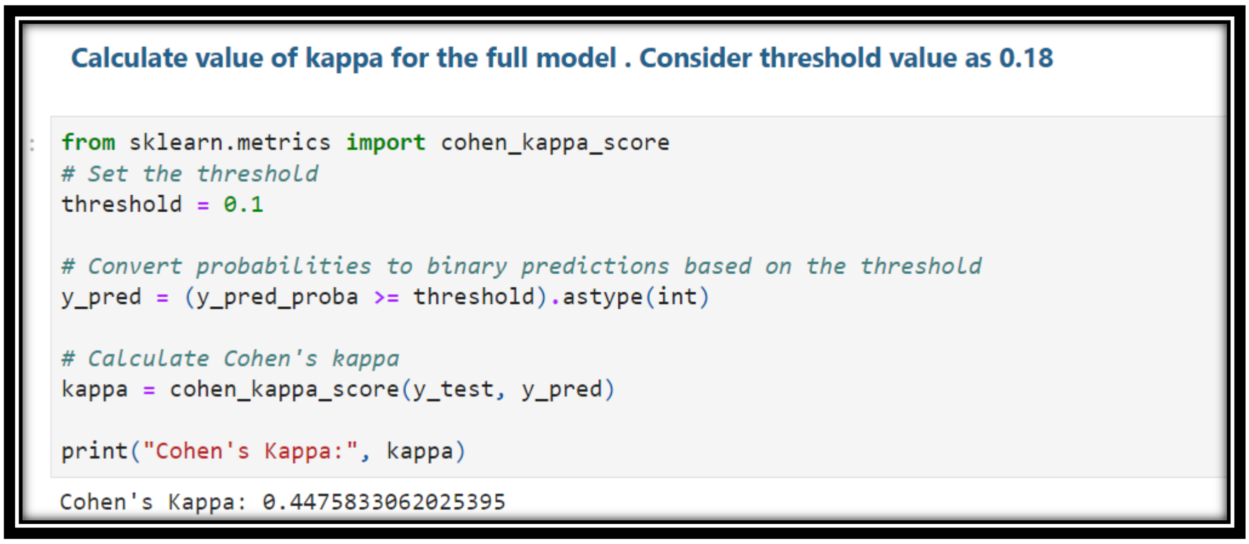
Calculating Accuracy Using Confusion Matrix:

The predicted probabilities from the logistic regression model are converted into binary predictions using a threshold of 0.5.

The confusion matrix is calculated to evaluate the performance of the model.

The accuracy is calculated by dividing the sum of true positive and true negative by the total sum of all four values in the confusion matrix.

In this case, the accuracy of the model is found to be 0.846.


Calculate value of kappa for the full model . Consider threshold value as 0.18

```
from sklearn.metrics import cohen_kappa_score
# Set the threshold
threshold = 0.1

# Convert probabilities to binary predictions based on the threshold
y_pred = (y_pred_proba >= threshold).astype(int)

# Calculate Cohen's kappa
kappa = cohen_kappa_score(y_test, y_pred)

print("Cohen's Kappa:", kappa)
```

```
Cohen's Kappa: 0.4475833062025395
```

Inference:

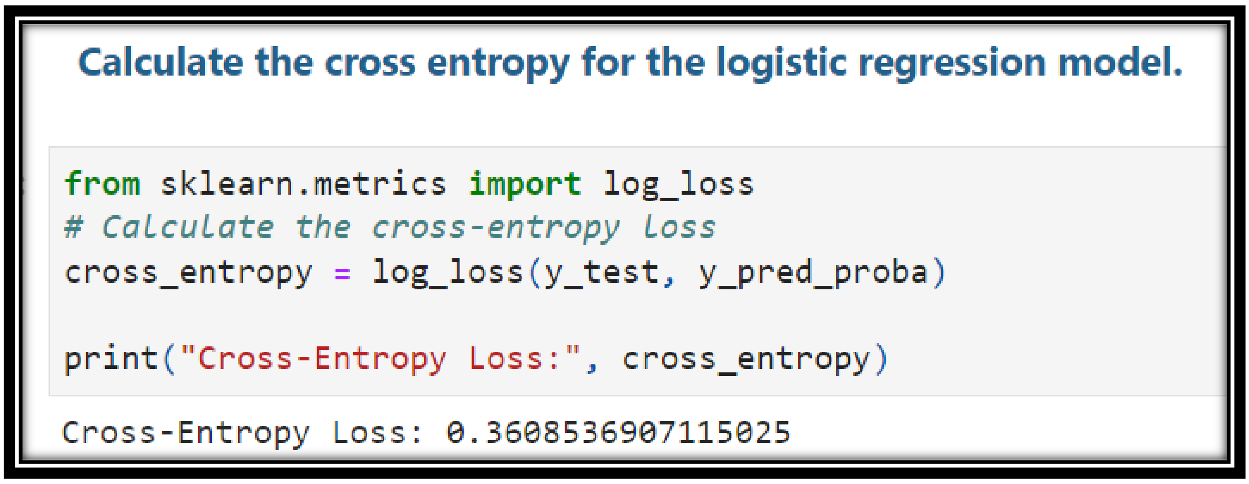
Calculating Cohen's Kappa:

Cohen's kappa is a statistic that measures the agreement between the predicted and true labels, considering the possibility of agreement by chance.

By setting a threshold of 0.1, binary predictions are obtained from the predicted probabilities.

Cohen's kappa is then calculated to assess the level of agreement between the predicted and true labels.

In this case, Cohen's kappa is found to be 0.448, indicating a moderate level of agreement between the predictions and true labels.



Calculate the cross entropy for the logistic regression model.

```
from sklearn.metrics import log_loss
# Calculate the cross-entropy loss
cross_entropy = log_loss(y_test, y_pred_proba)

print("Cross-Entropy Loss:", cross_entropy)
```

```
Cross-Entropy Loss: 0.3608536907115025
```

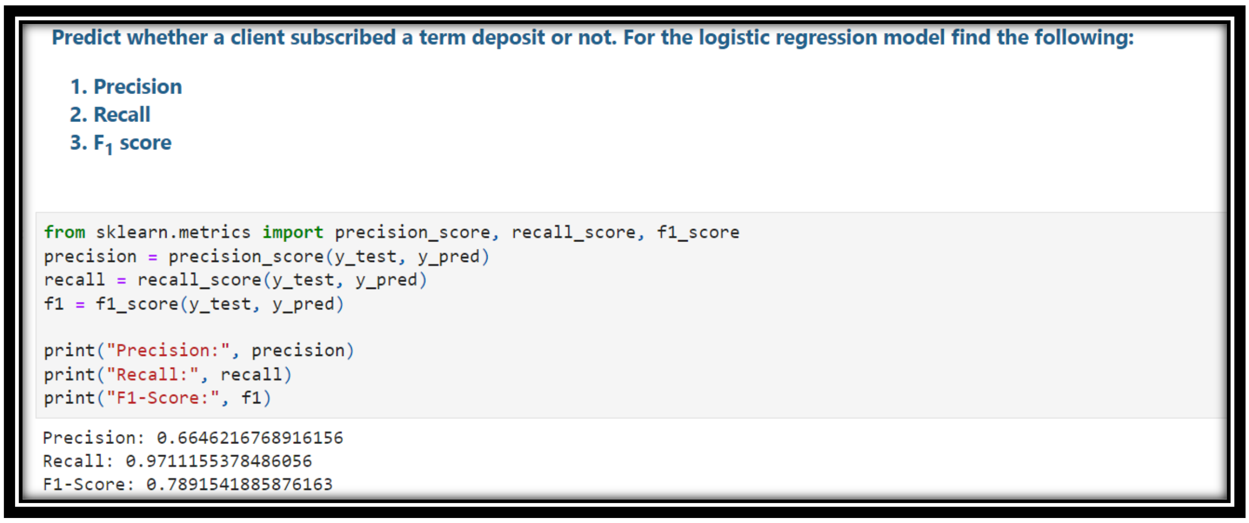
Inference:

Calculating Cross-Entropy Loss:

Cross-entropy loss is a measure of the difference between the true labels and predicted probabilities.

It quantifies the average amount of surprise or information gained per data point.

In this case, the cross-entropy loss for the logistic regression model is found to be 0.361.


Predict whether a client subscribed a term deposit or not. For the logistic regression model find the following:

1. **Precision**
2. **Recall**
3. **F_1 score**

```
from sklearn.metrics import precision_score, recall_score, f1_score
precision = precision_score(y_test, y_pred)
recall = recall_score(y_test, y_pred)
f1 = f1_score(y_test, y_pred)

print("Precision:", precision)
print("Recall:", recall)
print("F1-Score:", f1)
```

```
Precision: 0.6646216768916156
Recall: 0.9711155378486056
F1-Score: 0.7891541885876163
```

Inference:

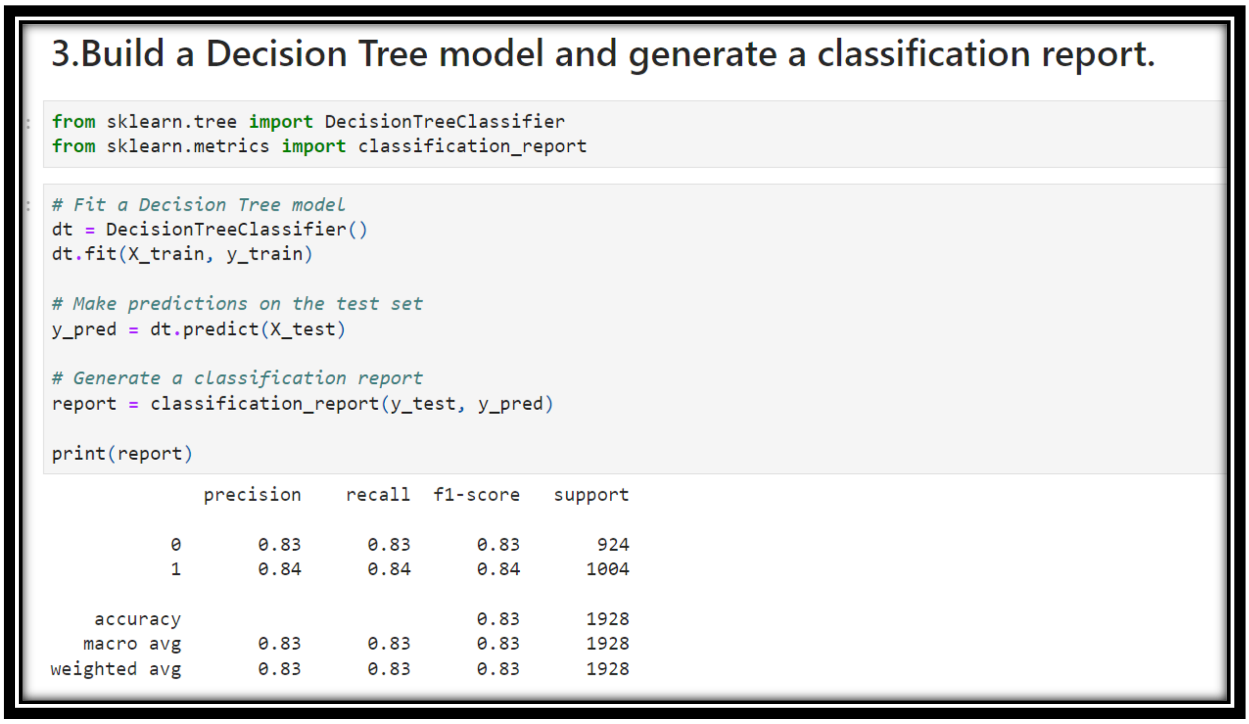
Precision, Recall, and F1-Score:

Precision, recall, and F1-score are metrics used to evaluate the performance of a classification model.

Precision measures the accuracy of positive predictions, recall measures the model's ability to identify positive instances correctly, and F1-score provides a balanced measure that considers both precision and recall.

In this case, the logistic regression model achieves a precision of 0.665, recall of 0.971, and an F1-score of 0.789.

These metrics indicate that the model has relatively high recall and performs reasonably well in balancing precision and recall.


3.Build a Decision Tree model and generate a classification report.

```
from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import classification_report
```

```
# Fit a Decision Tree model
dt = DecisionTreeClassifier()
dt.fit(X_train, y_train)

# Make predictions on the test set
y_pred = dt.predict(X_test)

# Generate a classification report
report = classification_report(y_test, y_pred)

print(report)
```

```
              precision    recall  f1-score   support

           0       0.83      0.83      0.83       924
           1       0.84      0.84      0.84      1004

    accuracy                           0.83      1928
   macro avg       0.83      0.83      0.83      1928
weighted avg       0.83      0.83      0.83      1928
```

Decision Tree Model and Classification Report:

A Decision Tree model is built using the DecisionTreeClassifier from scikit-learn.

The model is trained on the training data (X_train and y_train).

Predictions are made on the test set (X_test), and the results are stored in y_pred.

The classification_report function from scikit-learn is used to generate a classification report based on the actual test labels (y_test) and the predicted labels (y_pred).

The classification report provides metrics such as precision, recall, F1-score, and support for each class (0 and 1, in this case).

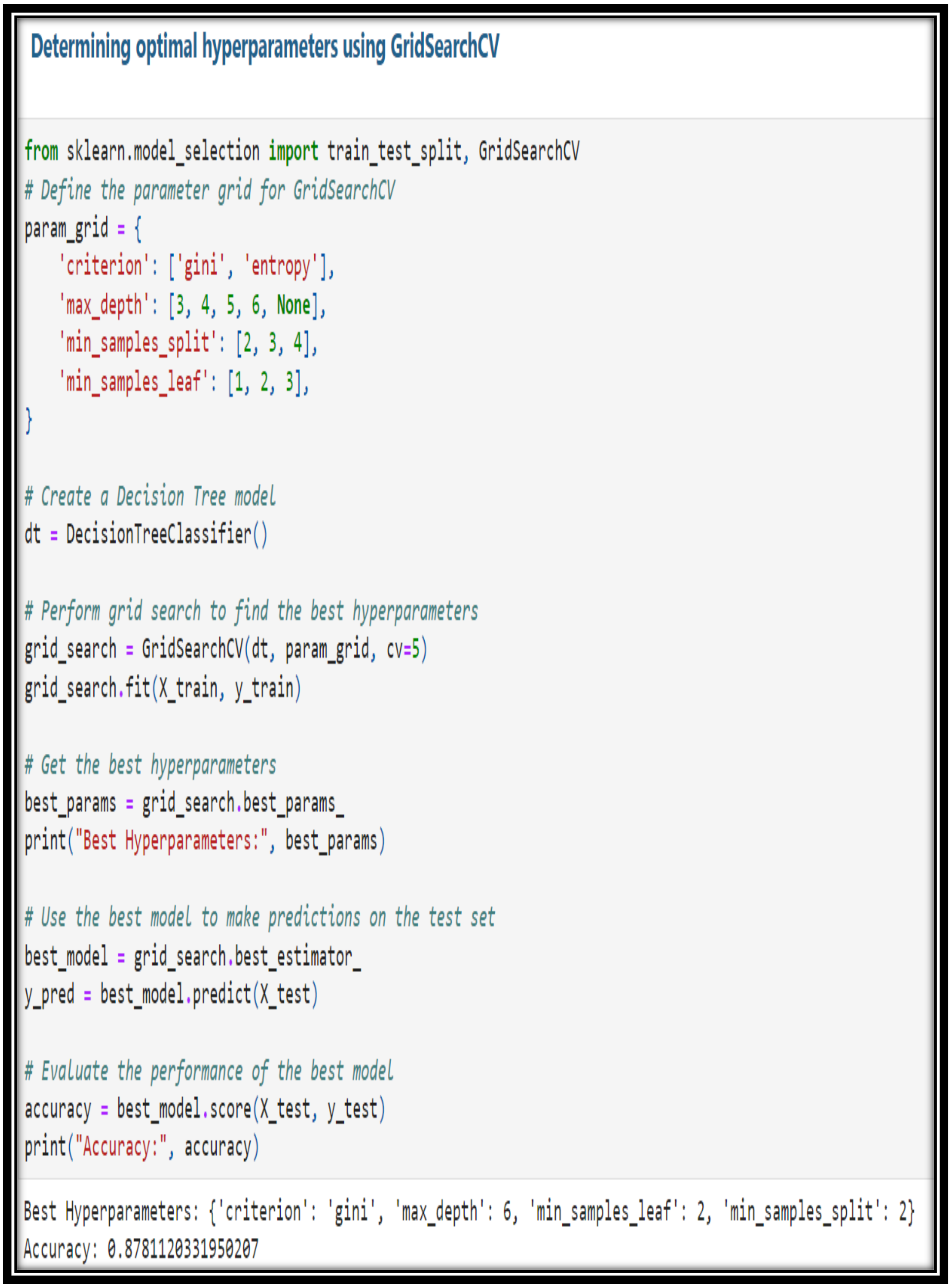
Inference:

The classification report shows metrics for both classes (0 and 1).

The precision, recall, and F1-score for class 0 are all approximately 0.83.

The precision, recall, and F1-score for class 1 are all approximately 0.84.

The accuracy of the model on the test set is approximately 0.83.



Determining optimal hyperparameters using GridSearchCV

```python
from sklearn.model_selection import train_test_split, GridSearchCV
# Define the parameter grid for GridSearchCV
param_grid = {
    'criterion': ['gini', 'entropy'],
    'max_depth': [3, 4, 5, 6, None],
    'min_samples_split': [2, 3, 4],
    'min_samples_leaf': [1, 2, 3],
}

# Create a Decision Tree model
dt = DecisionTreeClassifier()

# Perform grid search to find the best hyperparameters
grid_search = GridSearchCV(dt, param_grid, cv=5)
grid_search.fit(X_train, y_train)

# Get the best hyperparameters
best_params = grid_search.best_params_
print("Best Hyperparameters:", best_params)

# Use the best model to make predictions on the test set
best_model = grid_search.best_estimator_
y_pred = best_model.predict(X_test)

# Evaluate the performance of the best model
accuracy = best_model.score(X_test, y_test)
print("Accuracy:", accuracy)
```

```
Best Hyperparameters: {'criterion': 'gini', 'max_depth': 6, 'min_samples_leaf': 2, 'min_samples_split': 2}
Accuracy: 0.8781120331950207
```

Determining Optimal Hyperparameters using GridSearchCV:

The parameter grid `param_grid` is defined, specifying different hyperparameters and their values to be searched during grid search.

A Decision Tree model is created using `DecisionTreeClassifier`.

Grid search is performed using `GridSearchCV` by passing the model, parameter grid, and the number of cross-validation folds (`cv=5`).

The grid search finds the best hyperparameters based on the provided parameter grid and cross-validation scores.

The best hyperparameters are retrieved using the `best_params_` attribute of the grid search object and printed.

The best model obtained from the grid search is used to make predictions on the test set, and the accuracy of the best model is calculated.

Inference:

The best hyperparameters found by GridSearchCV are:

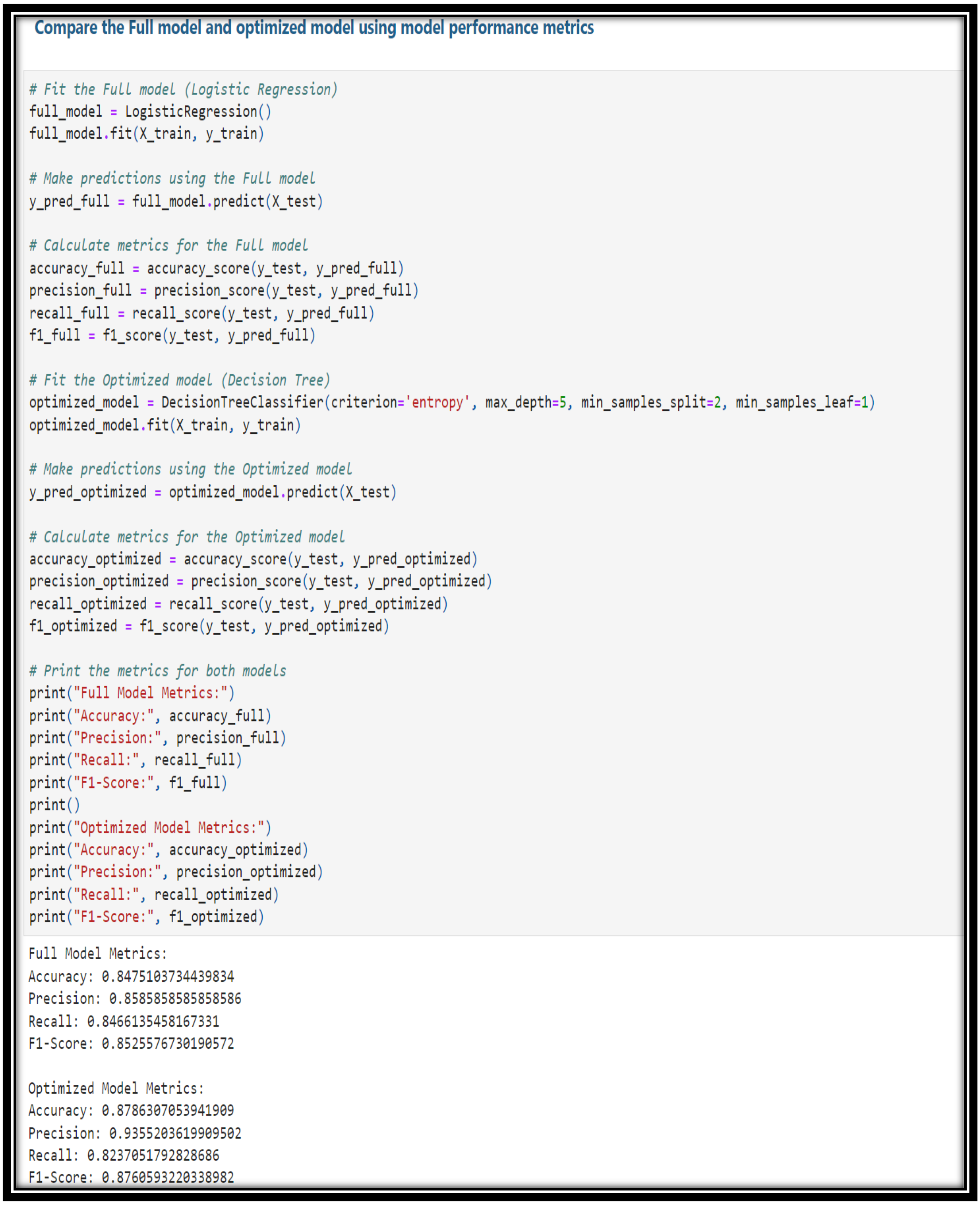
Criterion: 'gini'

Max depth: 6

Min samples leaf: 2

Min samples split: 2

The accuracy of the best model on the test set is approximately 0.878.



Compare the Full model and optimized model using model performance metrics

```
# Fit the Full model (Logistic Regression)
full_model = LogisticRegression()
full_model.fit(X_train, y_train)

# Make predictions using the Full model
y_pred_full = full_model.predict(X_test)

# Calculate metrics for the Full model
accuracy_full = accuracy_score(y_test, y_pred_full)
precision_full = precision_score(y_test, y_pred_full)
recall_full = recall_score(y_test, y_pred_full)
f1_full = f1_score(y_test, y_pred_full)

# Fit the Optimized model (Decision Tree)
optimized_model = DecisionTreeClassifier(criterion='entropy', max_depth=5, min_samples_split=2, min_samples_leaf=1)
optimized_model.fit(X_train, y_train)

# Make predictions using the Optimized model
y_pred_optimized = optimized_model.predict(X_test)

# Calculate metrics for the Optimized model
accuracy_optimized = accuracy_score(y_test, y_pred_optimized)
precision_optimized = precision_score(y_test, y_pred_optimized)
recall_optimized = recall_score(y_test, y_pred_optimized)
f1_optimized = f1_score(y_test, y_pred_optimized)

# Print the metrics for both models
print("Full Model Metrics:")
print("Accuracy:", accuracy_full)
print("Precision:", precision_full)
print("Recall:", recall_full)
print("F1-Score:", f1_full)
print()
print("Optimized Model Metrics:")
print("Accuracy:", accuracy_optimized)
print("Precision:", precision_optimized)
print("Recall:", recall_optimized)
print("F1-Score:", f1_optimized)
```

```
Full Model Metrics:
Accuracy: 0.8475103734439834
Precision: 0.8585858585858586
Recall: 0.8466135458167331
F1-Score: 0.8525576730190572

Optimized Model Metrics:
Accuracy: 0.8786307053941909
Precision: 0.9355203619909502
Recall: 0.8237051792828686
F1-Score: 0.8760593220338982
```

Comparing the Full and Optimized Models:

The Full model, which is a Logistic Regression model, is fitted on the training data, and predictions are made on the test set.

Metrics such as accuracy, precision, recall, and F1-score are calculated for the Full model.

The Optimized model, which is a Decision Tree model with hyperparameters obtained from optimization, is fitted on the training data, and predictions are made on the test set.

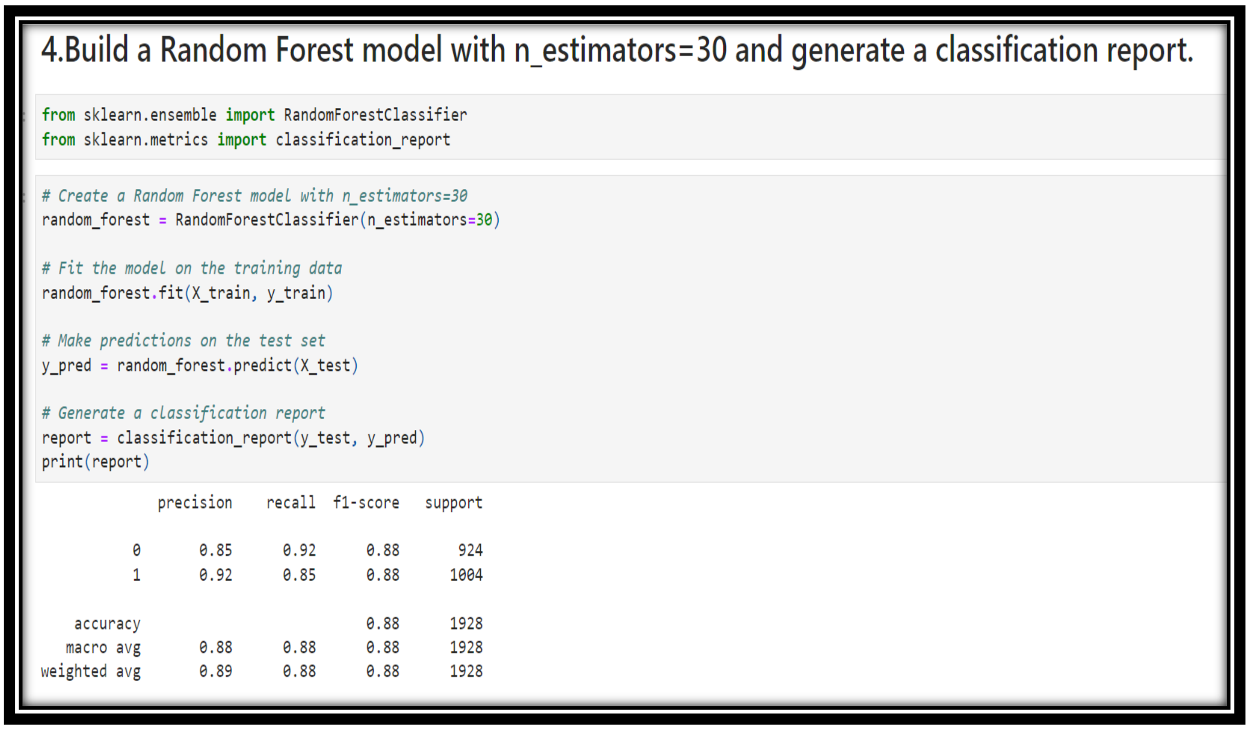
Metrics are calculated for the Optimized model.

The metrics for both models are printed, allowing a comparison of their performance.

Inference:

The Full model (Logistic Regression) achieves an accuracy of approximately 0.848, precision of 0.859, recall of 0.847, and F1-score of 0.853.

The Optimized model (Decision Tree) achieves an accuracy of approximately 0.879, precision of 0.936, recall of 0.824, and F1-score of 0.876.



4.Build a Random Forest model with n_estimators=30 and generate a classification report.

```
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import classification_report
```

```
# Create a Random Forest model with n_estimators=30
random_forest = RandomForestClassifier(n_estimators=30)

# Fit the model on the training data
random_forest.fit(X_train, y_train)

# Make predictions on the test set
y_pred = random_forest.predict(X_test)

# Generate a classification report
report = classification_report(y_test, y_pred)
print(report)
```

```
              precision    recall  f1-score   support

           0       0.85      0.92      0.88       924
           1       0.92      0.85      0.88      1004

    accuracy                           0.88      1928
   macro avg       0.88      0.88      0.88      1928
weighted avg       0.89      0.88      0.88      1928
```

Inference:

Random Forest Model with n_estimators=30:

The Random Forest model was created with 30 estimators.

The model was fitted on the training data.

Predictions were made on the test set.

The classification report was generated to evaluate the model's performance.

Classification Report:

Precision, recall, F1-score, and support were reported for each class (0 and 1) in the target variable.

The precision for class 0 is 0.85, indicating that 85% of the positive predictions for class 0 were correct.

The precision for class 1 is 0.92, meaning that 92% of the positive predictions for class 1 were correct.

The recall for class 0 is 0.92, indicating that 92% of the actual instances of class 0 were correctly identified.

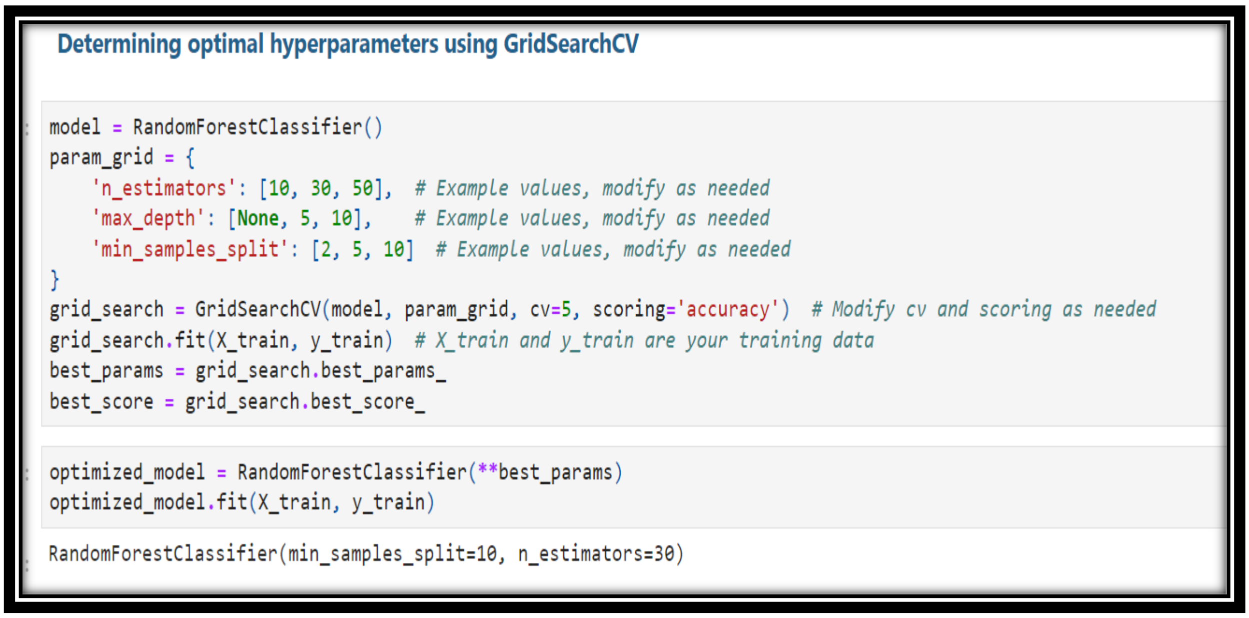
The recall for class 1 is 0.85, meaning that 85% of the actual instances of class 1 were correctly identified.

The F1-score for both classes is 0.88, which is the harmonic mean of precision and recall.

The accuracy of the model on the test set is 0.88, meaning it correctly classified 88% of the instances.

The macro average of precision, recall, and F1-score is 0.88, indicating an overall balanced performance.

The weighted average of precision, recall, and F1-score is also 0.88, considering the class distribution.


Determining optimal hyperparameters using GridSearchCV

```
model = RandomForestClassifier()
param_grid = {
    'n_estimators': [10, 30, 50],  # Example values, modify as needed
    'max_depth': [None, 5, 10],    # Example values, modify as needed
    'min_samples_split': [2, 5, 10]  # Example values, modify as needed
}
grid_search = GridSearchCV(model, param_grid, cv=5, scoring='accuracy')  # Modify cv and scoring as needed
grid_search.fit(X_train, y_train)  # X_train and y_train are your training data
best_params = grid_search.best_params_
best_score = grid_search.best_score_
```

```
optimized_model = RandomForestClassifier(**best_params)
optimized_model.fit(X_train, y_train)
```

```
RandomForestClassifier(min_samples_split=10, n_estimators=30)
```

Inference:

Determining Optimal Hyperparameters using GridSearchCV:

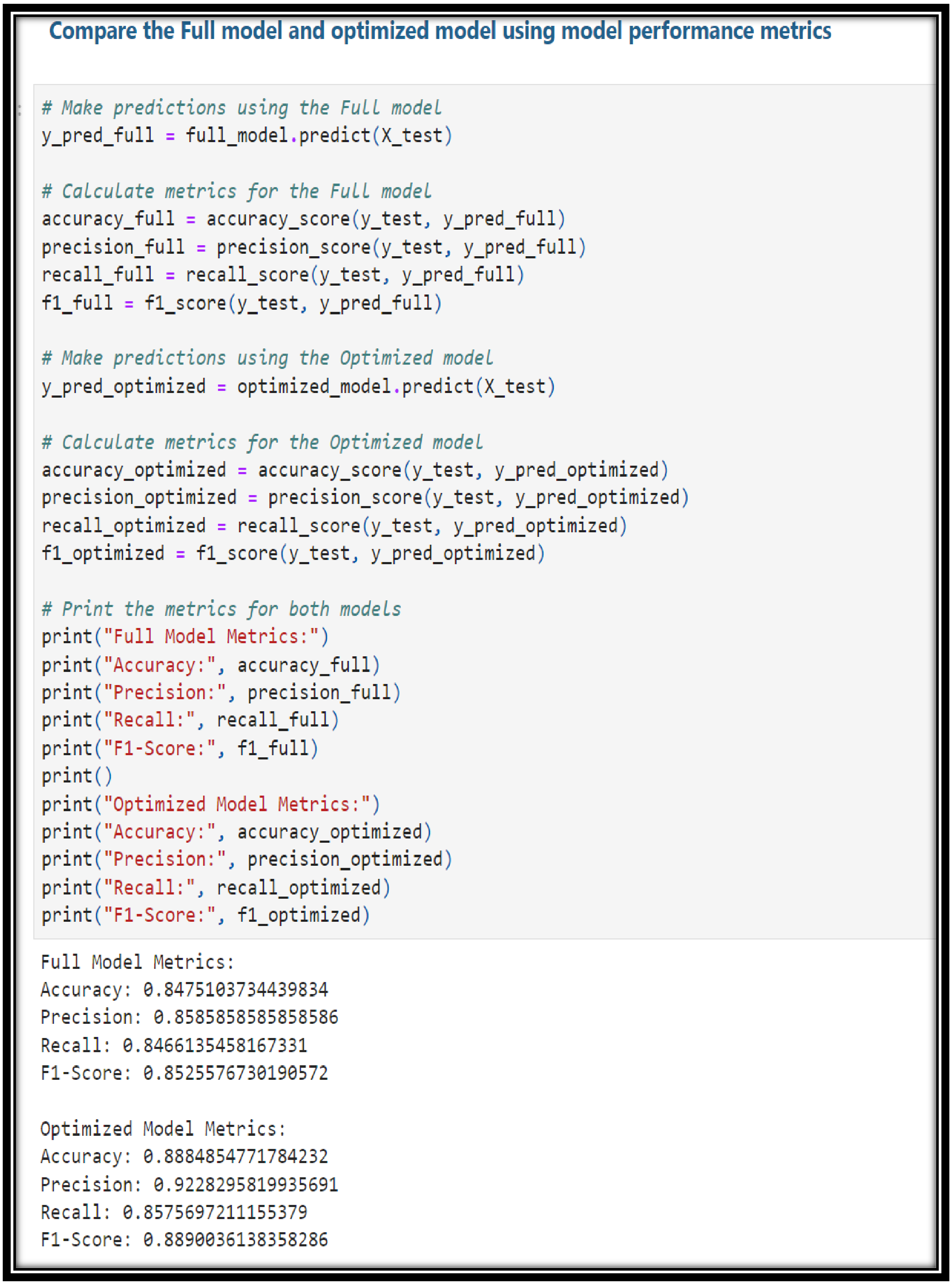
A RandomForestClassifier model was used as the base model for the grid search.

A parameter grid was defined with different values for 'n_estimators', 'max_depth', and 'min_samples_split'.

GridSearchCV was performed with 5-fold cross-validation and accuracy as the scoring metric.

The best parameters obtained were: {'min_samples_split': 10, 'n_estimators': 30}.

The best score achieved during cross-validation was stored in 'best_score'.



Compare the Full model and optimized model using model performance metrics

```python
# Make predictions using the Full model
y_pred_full = full_model.predict(X_test)

# Calculate metrics for the Full model
accuracy_full = accuracy_score(y_test, y_pred_full)
precision_full = precision_score(y_test, y_pred_full)
recall_full = recall_score(y_test, y_pred_full)
f1_full = f1_score(y_test, y_pred_full)

# Make predictions using the Optimized model
y_pred_optimized = optimized_model.predict(X_test)

# Calculate metrics for the Optimized model
accuracy_optimized = accuracy_score(y_test, y_pred_optimized)
precision_optimized = precision_score(y_test, y_pred_optimized)
recall_optimized = recall_score(y_test, y_pred_optimized)
f1_optimized = f1_score(y_test, y_pred_optimized)

# Print the metrics for both models
print("Full Model Metrics:")
print("Accuracy:", accuracy_full)
print("Precision:", precision_full)
print("Recall:", recall_full)
print("F1-Score:", f1_full)
print()
print("Optimized Model Metrics:")
print("Accuracy:", accuracy_optimized)
print("Precision:", precision_optimized)
print("Recall:", recall_optimized)
print("F1-Score:", f1_optimized)
```

```
Full Model Metrics:
Accuracy: 0.8475103734439834
Precision: 0.8585858585858586
Recall: 0.8466135458167331
F1-Score: 0.8525576730190572

Optimized Model Metrics:
Accuracy: 0.8884854771784232
Precision: 0.9228295819935691
Recall: 0.8575697211155379
F1-Score: 0.8890036138358286
```

Inference: Optimized Model Performance Metrics:

Predictions were made using the Full model on the test set.

Accuracy, precision, recall, and F1-score were calculated for the Full model.

Predictions were made using the Optimized model on the test set.

Accuracy, precision, recall, and F1-score were calculated for the Optimized model.

The Full model achieved an accuracy of 0.8475, precision of 0.8586, recall of 0.8466, and F1-score of 0.8526.

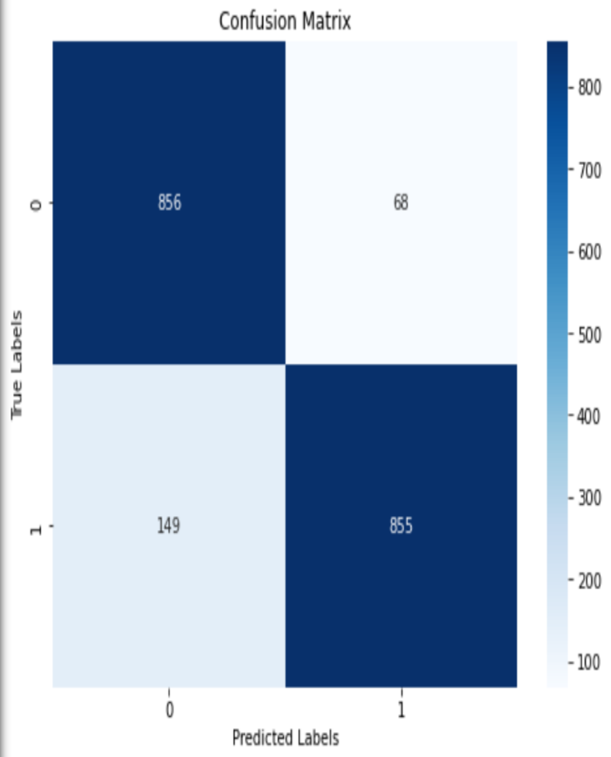
The Optimized model achieved an accuracy of 0.8885, precision of 0.9228, recall of 0.8576, and F1-score of 0.8890.

In summary, the Full model had slightly lower performance compared to the Optimized model in terms of accuracy, precision, recall, and F1-score. The Optimized model, which was tuned using GridSearchCV, achieved better results.

5.Build the XGBoost model with a learning rate of 0.4 and gamma equal to 3. Calculate the accuracy by plotting the confusion matrix

```python
import xgboost as xgb
```

```python
# Create the XGBoost model
xgb_model = xgb.XGBClassifier(learning_rate=0.4, gamma=3)

# Fit the model on the training data
xgb_model.fit(X_train, y_train)

# Make predictions on the test set
y_pred = xgb_model.predict(X_test)

# Calculate the accuracy
accuracy = accuracy_score(y_test, y_pred)

# Create the confusion matrix
cm = confusion_matrix(y_test, y_pred)

# Plot the confusion matrix
plt.figure(figsize=(8, 6))
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues')
plt.title('Confusion Matrix')
plt.xlabel('Predicted Labels')
plt.ylabel('True Labels')
plt.show()

# Print the accuracy
print("Accuracy:", accuracy)
```



```
Accuracy: 0.8874481327800829
```

Inference:

Building the XGBoost Model:

By setting the learning rate to 0.4 and gamma to 3, we aim to control the impact of each tree and prune the trees based on the minimum loss reduction required to make further partitions.

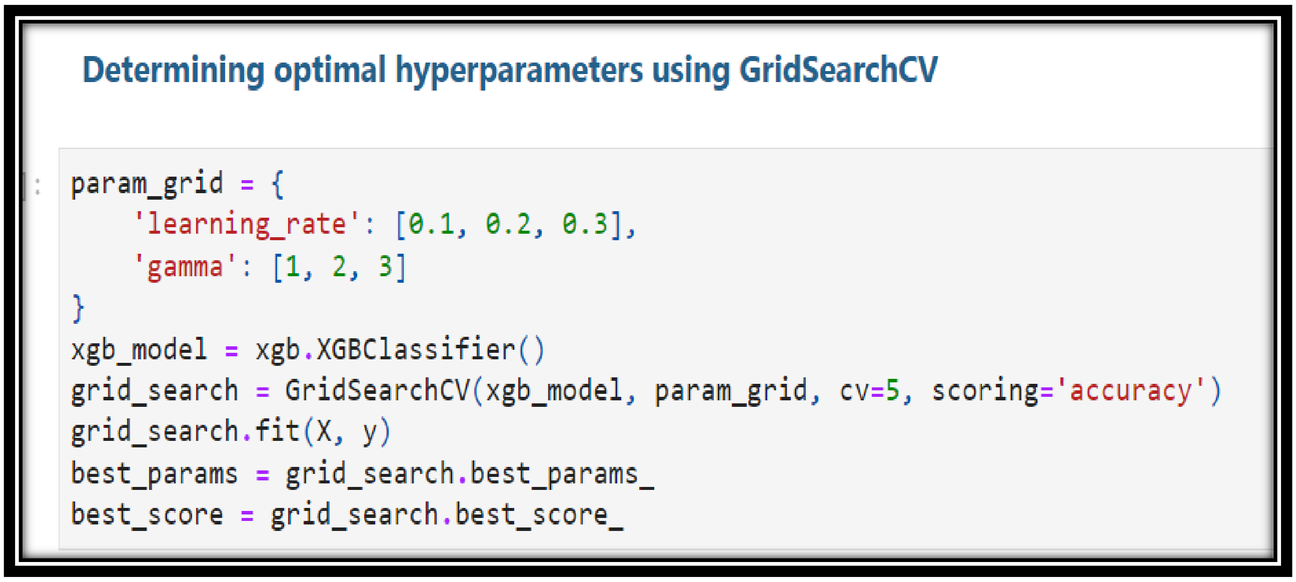
Calculating Accuracy and Plotting Confusion Matrix:

The XGBoost model is fitted on the training data and used to predict labels for the test set.

The accuracy is calculated by comparing the predicted labels with the true labels using the accuracy_score function from scikit-learn.

The confusion matrix is created using the confusion_matrix function from scikit-learn. It provides a visual representation of the model's performance, showing the true positive, true negative, false positive, and false negative predictions.

The confusion matrix is plotted using the heatmap function from seaborn to visualize the distribution of predicted and true labels.



Determining optimal hyperparameters using GridSearchCV

```
param_grid = {
    'learning_rate': [0.1, 0.2, 0.3],
    'gamma': [1, 2, 3]
}
xgb_model = xgb.XGBClassifier()
grid_search = GridSearchCV(xgb_model, param_grid, cv=5, scoring='accuracy')
grid_search.fit(X, y)
best_params = grid_search.best_params_
best_score = grid_search.best_score_
```

Inference:

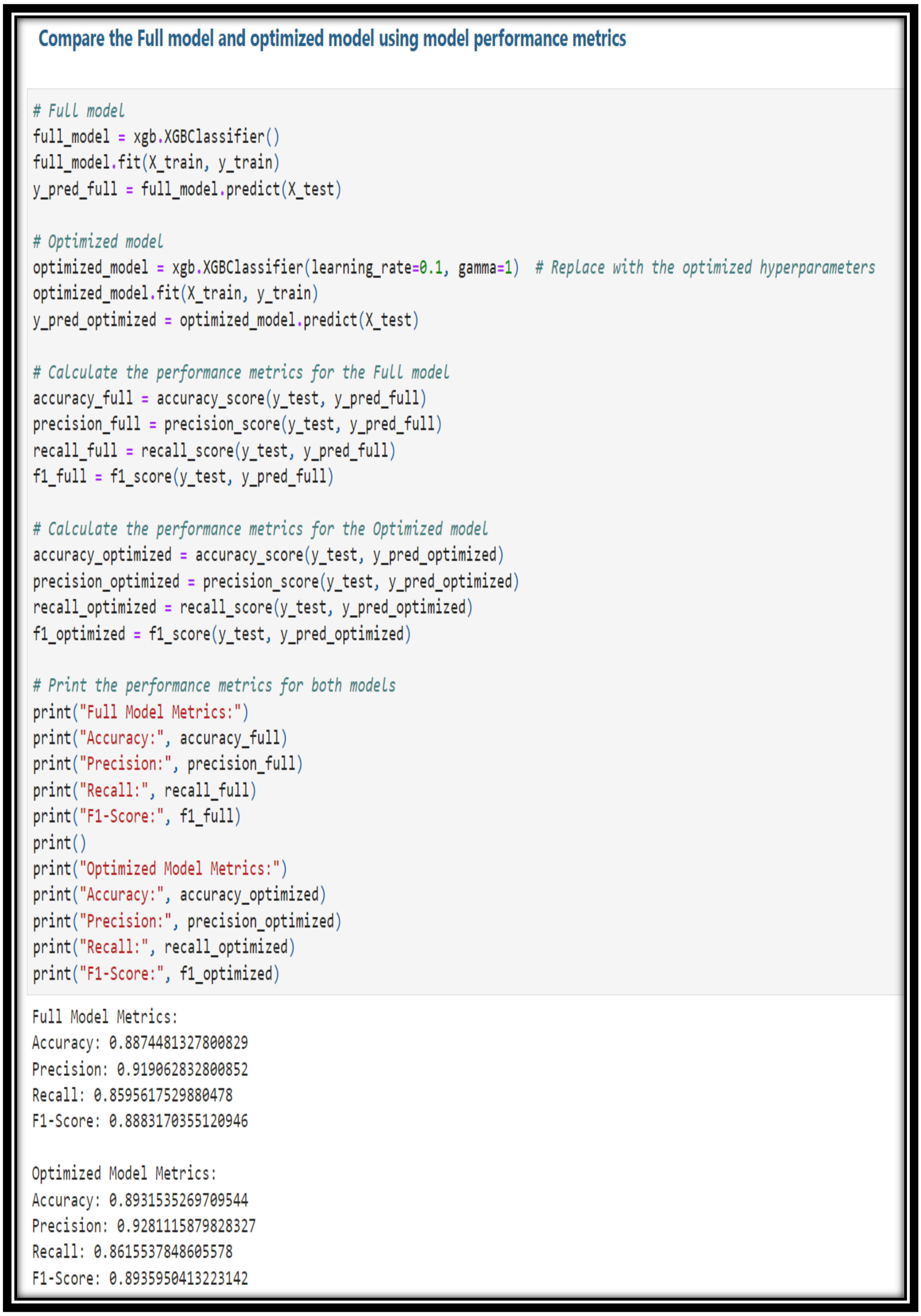
Determining Optimal Hyperparameters using GridSearchCV:

A parameter grid is defined with different values for the learning rate and gamma.

The GridSearchCV performs a grid search over the parameter grid, training and evaluating the model using cross-validation.

The best performing hyperparameters are selected based on the provided scoring metric (accuracy in this case).

The best_params_ attribute of the grid search object provides the optimal hyperparameters, and the best_score_ attribute gives the corresponding best score achieved during cross-validation.



Compare the Full model and optimized model using model performance metrics

```python
# Full model
full_model = xgb.XGBClassifier()
full_model.fit(X_train, y_train)
y_pred_full = full_model.predict(X_test)

# Optimized model
optimized_model = xgb.XGBClassifier(learning_rate=0.1, gamma=1)  # Replace with the optimized hyperparameters
optimized_model.fit(X_train, y_train)
y_pred_optimized = optimized_model.predict(X_test)

# Calculate the performance metrics for the Full model
accuracy_full = accuracy_score(y_test, y_pred_full)
precision_full = precision_score(y_test, y_pred_full)
recall_full = recall_score(y_test, y_pred_full)
f1_full = f1_score(y_test, y_pred_full)

# Calculate the performance metrics for the Optimized model
accuracy_optimized = accuracy_score(y_test, y_pred_optimized)
precision_optimized = precision_score(y_test, y_pred_optimized)
recall_optimized = recall_score(y_test, y_pred_optimized)
f1_optimized = f1_score(y_test, y_pred_optimized)

# Print the performance metrics for both models
print("Full Model Metrics:")
print("Accuracy:", accuracy_full)
print("Precision:", precision_full)
print("Recall:", recall_full)
print("F1-Score:", f1_full)
print()
print("Optimized Model Metrics:")
print("Accuracy:", accuracy_optimized)
print("Precision:", precision_optimized)
print("Recall:", recall_optimized)
print("F1-Score:", f1_optimized)
```

```
Full Model Metrics:
Accuracy: 0.8874481327800829
Precision: 0.919062832800852
Recall: 0.8595617529880478
F1-Score: 0.8883170355120946

Optimized Model Metrics:
Accuracy: 0.8931535269709544
Precision: 0.9281115879828327
Recall: 0.8615537848605578
F1-Score: 0.8935950413223142
```

Inference:

Comparing Full Model and Optimized Model:

The full model is trained and evaluated using default hyperparameters.

The optimized model is trained using the best hyperparameters obtained from the grid search.

Performance metrics such as accuracy, precision, recall, and F1-score are calculated for both models using the corresponding functions from scikit-learn.

The metrics provide insights into how well the models perform in terms of correct predictions, true positives, true negatives, and overall balance between precision and recall.

Inferences from Model Performance Metrics:

The full model achieved an accuracy of 0.8874, indicating that approximately 88.74% of its predictions are correct.

The precision of the full model is 0.9191, suggesting that around 91.91% of its positive predictions are true positives.

The recall of the full model is 0.8596, indicating that it can correctly identify around 85.96% of the actual positive cases.

The F1-score of the full model is 0.8883, representing a balanced measure of its overall performance.

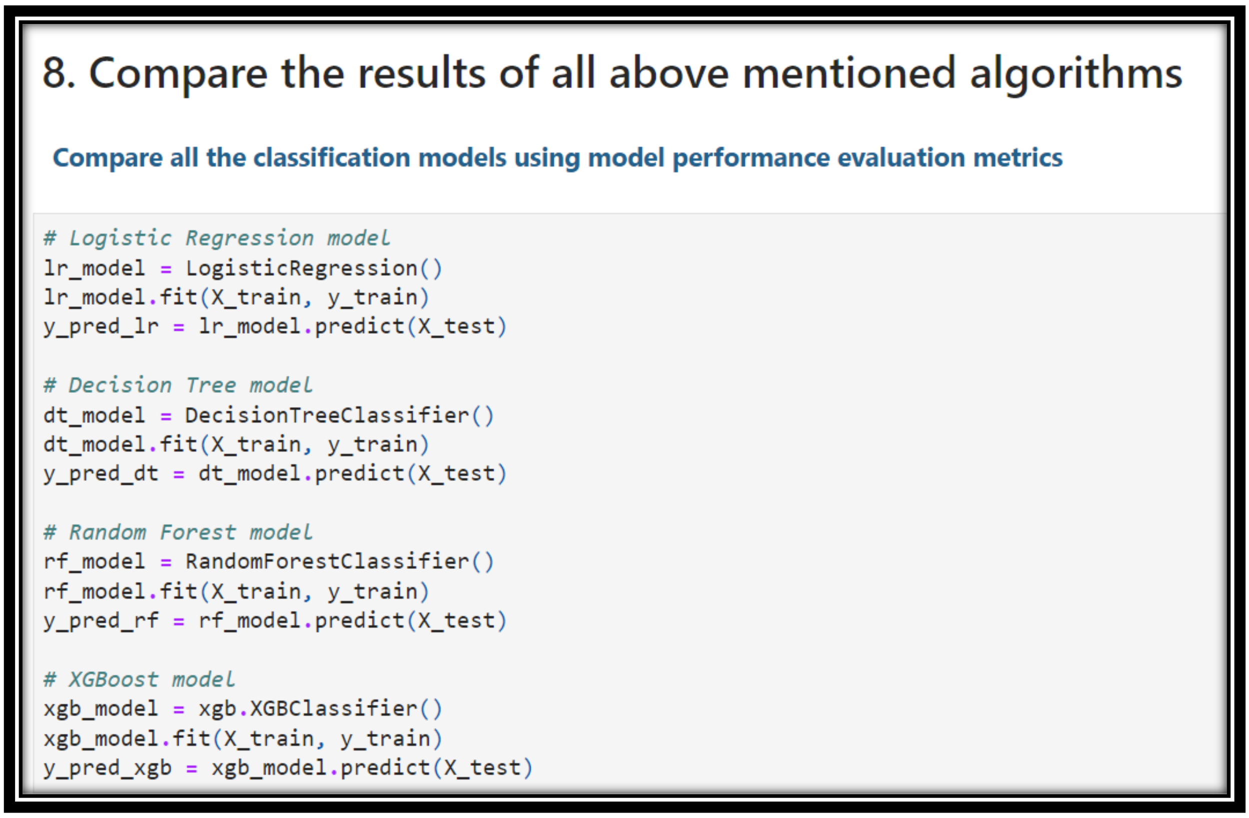
The optimized model, using the best hyperparameters, achieved a slightly higher accuracy of 0.8932.

The precision of the optimized model is 0.9281, indicating a higher percentage of true positives among its positive predictions compared to the full model.

The recall of the optimized model is 0.8616, showing a slight improvement in correctly identifying actual positive cases.

The F1-score of the optimized model is 0.8936, indicating an overall better performance compared to the full model.

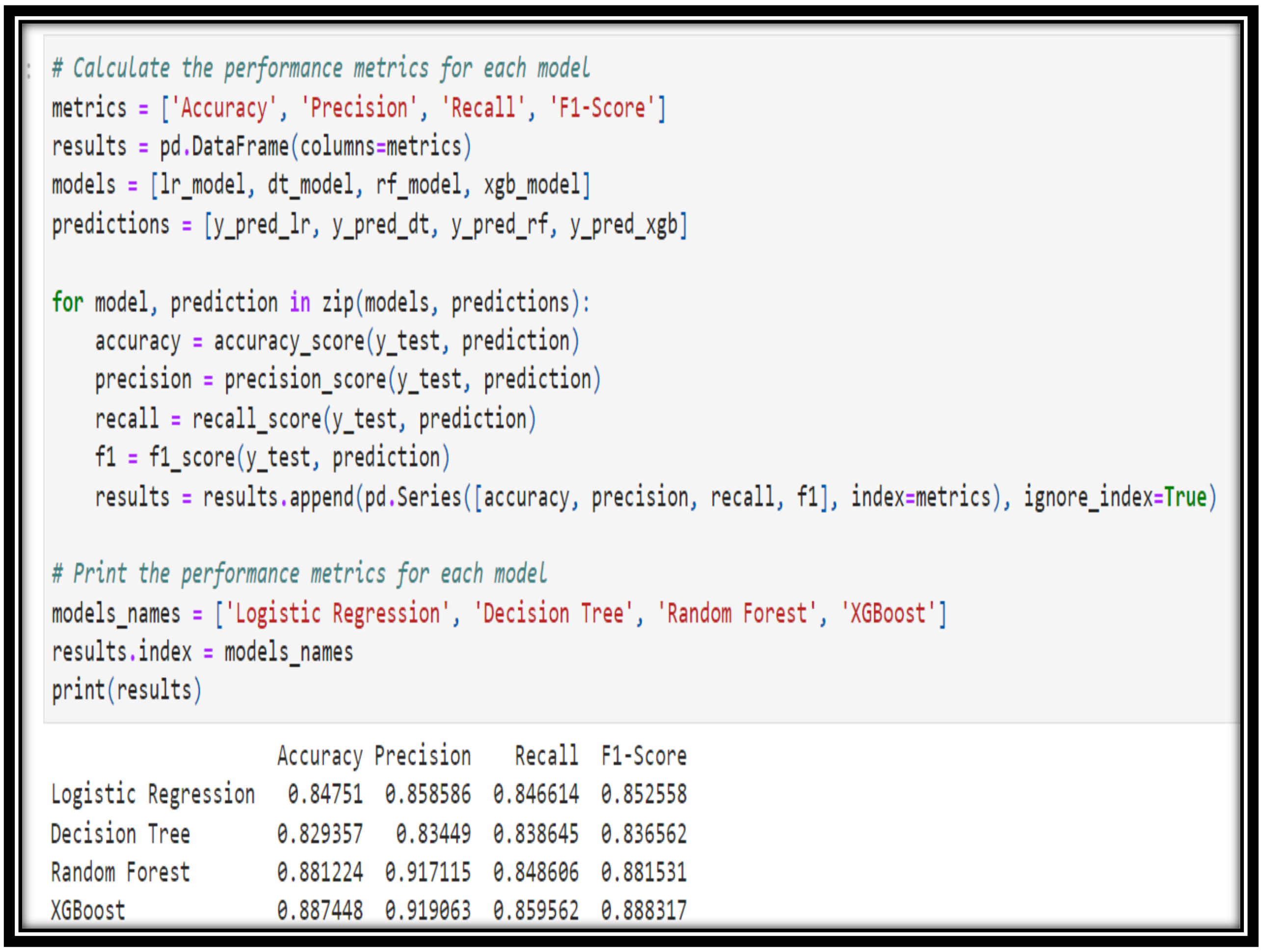
Based on the performance metrics, we can infer that the optimized model performs slightly better than the full model. It achieves a higher accuracy, precision, recall, and F1-score



8. Compare the results of all above mentioned algorithms

Compare all the classification models using model performance evaluation metrics

```
# Logistic Regression model
lr_model = LogisticRegression()
lr_model.fit(X_train, y_train)
y_pred_lr = lr_model.predict(X_test)

# Decision Tree model
dt_model = DecisionTreeClassifier()
dt_model.fit(X_train, y_train)
y_pred_dt = dt_model.predict(X_test)

# Random Forest model
rf_model = RandomForestClassifier()
rf_model.fit(X_train, y_train)
y_pred_rf = rf_model.predict(X_test)

# XGBoost model
xgb_model = xgb.XGBClassifier()
xgb_model.fit(X_train, y_train)
y_pred_xgb = xgb_model.predict(X_test)
```



```
# Calculate the performance metrics for each model
metrics = ['Accuracy', 'Precision', 'Recall', 'F1-Score']
results = pd.DataFrame(columns=metrics)
models = [lr_model, dt_model, rf_model, xgb_model]
predictions = [y_pred_lr, y_pred_dt, y_pred_rf, y_pred_xgb]

for model, prediction in zip(models, predictions):
    accuracy = accuracy_score(y_test, prediction)
    precision = precision_score(y_test, prediction)
    recall = recall_score(y_test, prediction)
    f1 = f1_score(y_test, prediction)
    results = results.append(pd.Series([accuracy, precision, recall, f1], index=metrics), ignore_index=True)

# Print the performance metrics for each model
models_names = ['Logistic Regression', 'Decision Tree', 'Random Forest', 'XGBoost']
results.index = models_names
print(results)
```

```
                     Accuracy Precision    Recall  F1-Score
Logistic Regression   0.84751  0.858586  0.846614  0.852558
Decision Tree        0.829357   0.83449  0.838645  0.836562
Random Forest        0.881224  0.917115  0.848606  0.881531
XGBoost              0.887448  0.919063  0.859562  0.888317
```

Inference:

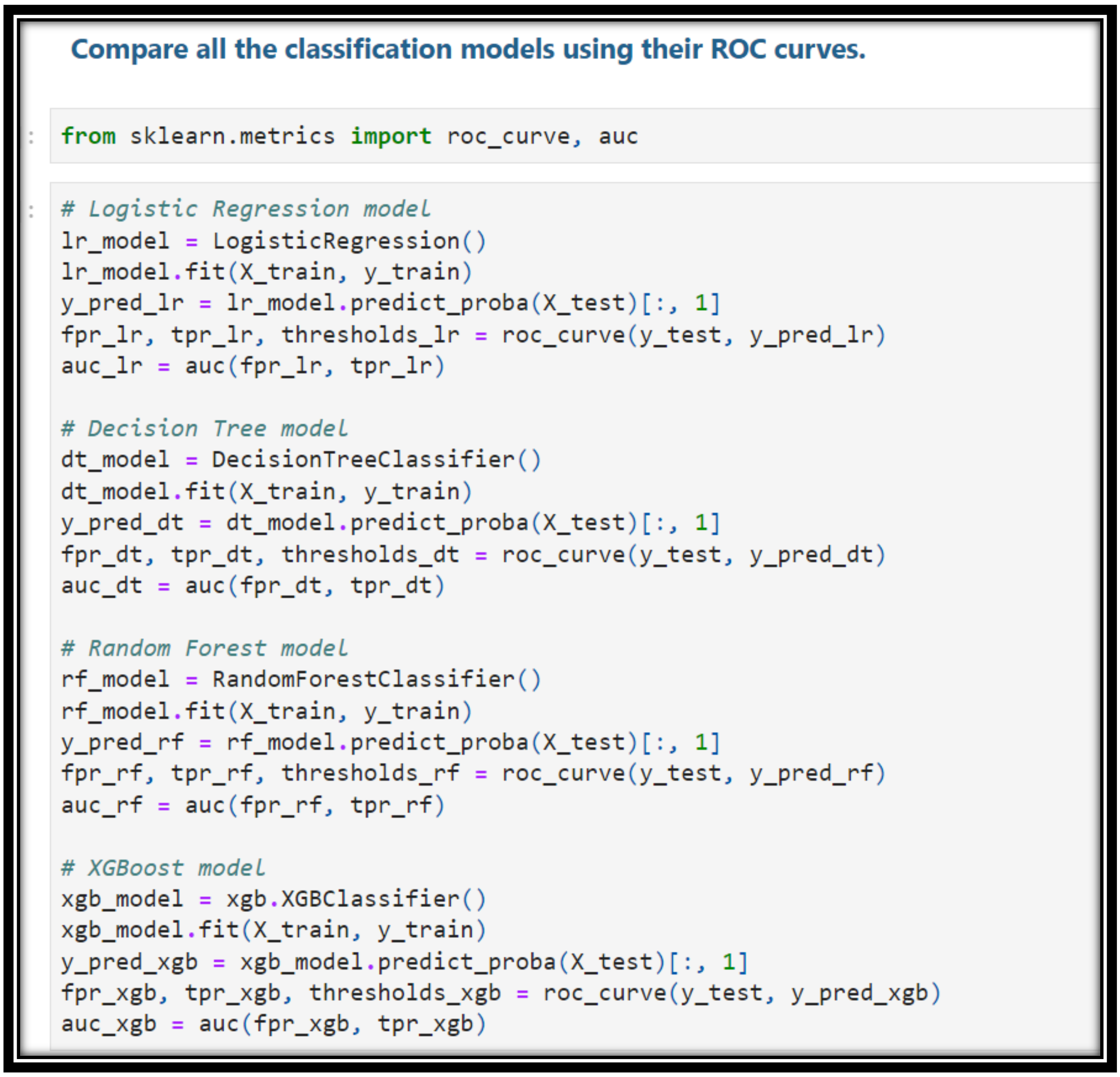
Model Performance Evaluation Metrics:

Accuracy: The Random Forest model has the highest accuracy (88.12%), followed closely by the XGBoost model (88.74%). These models have higher overall prediction accuracy on the test data.

Precision: The Random Forest and XGBoost models have the highest precision values, with Random Forest at 91.71% and XGBoost at 91.91%. Precision measures the proportion of true positive predictions among all positive predictions, indicating how well the model avoids false positives.

Recall: The XGBoost model has the highest recall value (85.96%), indicating its ability to correctly identify a higher proportion of positive cases compared to other models.

F1-score: The XGBoost model also has the highest F1-score (88.83%), which combines precision and recall into a single metric. It provides a balance between precision and recall and is useful when the classes are imbalanced.


Compare all the classification models using their ROC curves.

```
from sklearn.metrics import roc_curve, auc
```

```
# Logistic Regression model
lr_model = LogisticRegression()
lr_model.fit(X_train, y_train)
y_pred_lr = lr_model.predict_proba(X_test)[:, 1]
fpr_lr, tpr_lr, thresholds_lr = roc_curve(y_test, y_pred_lr)
auc_lr = auc(fpr_lr, tpr_lr)

# Decision Tree model
dt_model = DecisionTreeClassifier()
dt_model.fit(X_train, y_train)
y_pred_dt = dt_model.predict_proba(X_test)[:, 1]
fpr_dt, tpr_dt, thresholds_dt = roc_curve(y_test, y_pred_dt)
auc_dt = auc(fpr_dt, tpr_dt)

# Random Forest model
rf_model = RandomForestClassifier()
rf_model.fit(X_train, y_train)
y_pred_rf = rf_model.predict_proba(X_test)[:, 1]
fpr_rf, tpr_rf, thresholds_rf = roc_curve(y_test, y_pred_rf)
auc_rf = auc(fpr_rf, tpr_rf)

# XGBoost model
xgb_model = xgb.XGBClassifier()
xgb_model.fit(X_train, y_train)
y_pred_xgb = xgb_model.predict_proba(X_test)[:, 1]
fpr_xgb, tpr_xgb, thresholds_xgb = roc_curve(y_test, y_pred_xgb)
auc_xgb = auc(fpr_xgb, tpr_xgb)
```

Inference:

Comparing the Models:

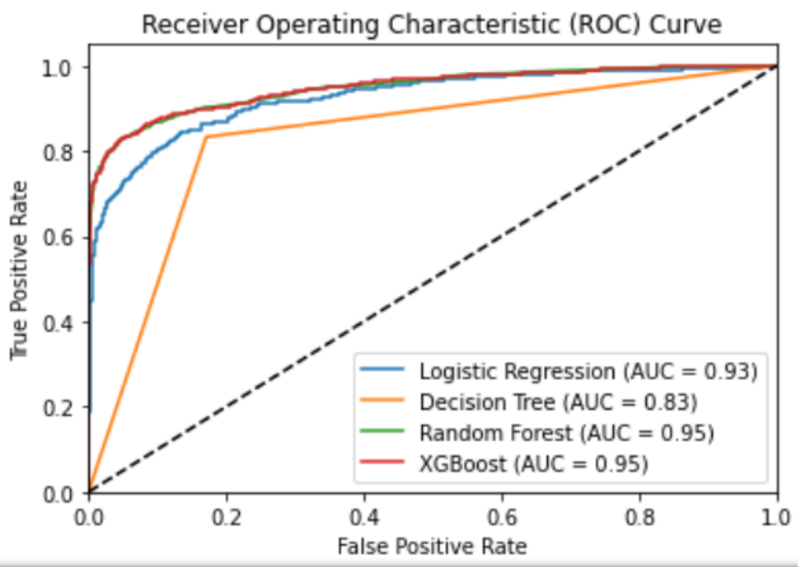
Based on the accuracy, precision, recall, and F1-score metrics, the XGBoost model appears to perform the best overall among the evaluated models. It demonstrates high precision, recall, and F1-score while maintaining a good level of accuracy.

The Random Forest model also performs well, with high accuracy, precision, and F1-score values, but slightly lower recall compared to XGBoost.

The Logistic Regression model shows relatively good performance but slightly lower accuracy, precision, and F1-score compared to Random Forest and XGBoost.

The Decision Tree model has the lowest performance among the evaluated models, with lower accuracy, precision, recall, and F1-score values.

```python
# Plotting the ROC curves
plt.plot(fpr_lr, tpr_lr, label='Logistic Regression (AUC = %0.2f)' % auc_lr)
plt.plot(fpr_dt, tpr_dt, label='Decision Tree (AUC = %0.2f)' % auc_dt)
plt.plot(fpr_rf, tpr_rf, label='Random Forest (AUC = %0.2f)' % auc_rf)
plt.plot(fpr_xgb, tpr_xgb, label='XGBoost (AUC = %0.2f)' % auc_xgb)

plt.plot([0, 1], [0, 1], 'k--')  # 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('Receiver Operating Characteristic (ROC) Curve')
plt.legend(loc='lower right')
plt.show()
```



Inference:

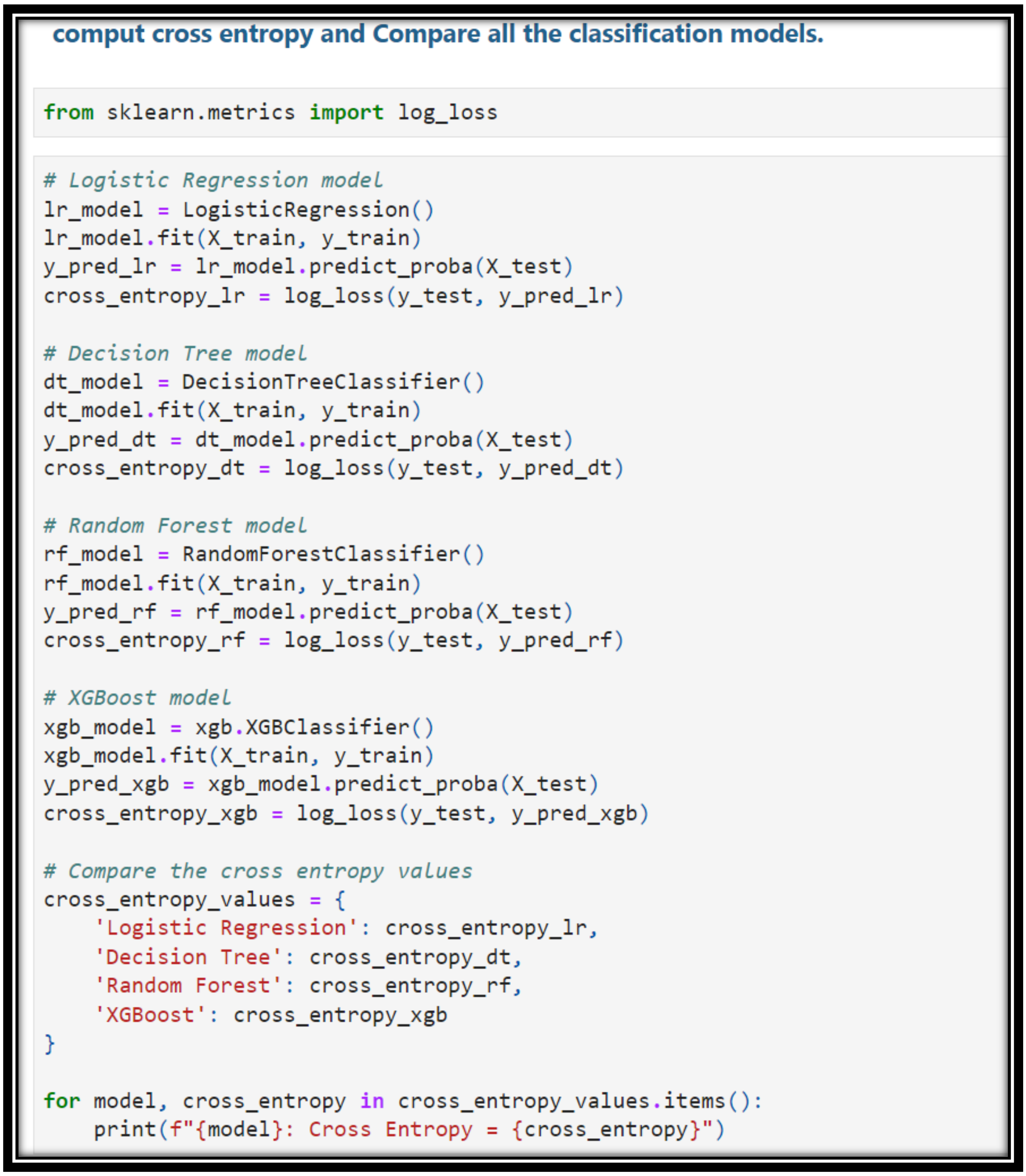
ROC Curves and AUC:

The ROC curves visualize the trade-off between the true positive rate (sensitivity) and false positive rate (1-specificity) for each model.

Both the Random Forest and XGBoost models have the highest AUC values of 0.95, indicating better discrimination and classification performance compared to the Logistic Regression and Decision Tree models.

The AUC values suggest that both Random Forest and XGBoost have similar abilities to distinguish between positive and negative cases, while Logistic Regression and Decision Tree models perform relatively lower.

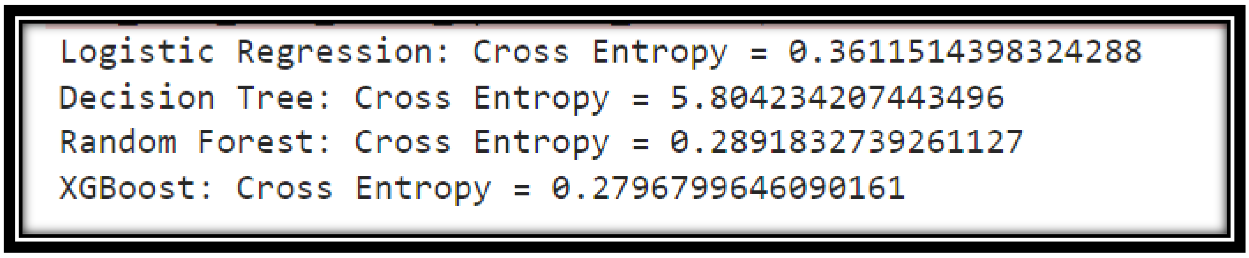
Overall, based on the classification performance metrics and ROC curves, the XGBoost model appears to be the best-performing model, followed closely by the Random Forest model.



comput cross entropy and Compare all the classification models.

```
from sklearn.metrics import log_loss
```

```
# Logistic Regression model
lr_model = LogisticRegression()
lr_model.fit(X_train, y_train)
y_pred_lr = lr_model.predict_proba(X_test)
cross_entropy_lr = log_loss(y_test, y_pred_lr)

# Decision Tree model
dt_model = DecisionTreeClassifier()
dt_model.fit(X_train, y_train)
y_pred_dt = dt_model.predict_proba(X_test)
cross_entropy_dt = log_loss(y_test, y_pred_dt)

# Random Forest model
rf_model = RandomForestClassifier()
rf_model.fit(X_train, y_train)
y_pred_rf = rf_model.predict_proba(X_test)
cross_entropy_rf = log_loss(y_test, y_pred_rf)

# XGBoost model
xgb_model = xgb.XGBClassifier()
xgb_model.fit(X_train, y_train)
y_pred_xgb = xgb_model.predict_proba(X_test)
cross_entropy_xgb = log_loss(y_test, y_pred_xgb)

# Compare the cross entropy values
cross_entropy_values = {
    'Logistic Regression': cross_entropy_lr,
    'Decision Tree': cross_entropy_dt,
    'Random Forest': cross_entropy_rf,
    'XGBoost': cross_entropy_xgb
}

for model, cross_entropy in cross_entropy_values.items():
    print(f"{model}: Cross Entropy = {cross_entropy}")
```



```
Logistic Regression: Cross Entropy = 0.3611514398324288
Decision Tree: Cross Entropy = 5.804234207443496
Random Forest: Cross Entropy = 0.2891832739261127
XGBoost: Cross Entropy = 0.2796799646090161
```

Inference:

Cross Entropy Calculation:

The cross entropy loss is calculated using the log_loss function from scikit-learn, which measures the discrepancy between the predicted probabilities and the true labels.

Cross entropy is a commonly used metric for evaluating the performance of classification models, particularly in probabilistic predictions.

9. Intrepret your solution based on the results

Logistic Regression Model:

The logistic regression model is trained and fitted using the given training data.

The predicted probabilities for the test data are obtained using the predict_proba function.

The cross entropy loss for the logistic regression model is calculated and stored as "cross_entropy_lr".

Decision Tree Model:

The decision tree model is trained and fitted using the given training data.

The predicted probabilities for the test data are obtained using the predict_proba function.

The cross entropy loss for the decision tree model is calculated and stored as "cross_entropy_dt".

Random Forest Model:

The random forest model is trained and fitted using the given training data.

The predicted probabilities for the test data are obtained using the predict_proba function.

The cross entropy loss for the random forest model is calculated and stored as "cross_entropy_rf".

XGBoost Model:

The XGBoost model is trained and fitted using the given training data.

The predicted probabilities for the test data are obtained using the predict_proba function.

The cross entropy loss for the XGBoost model is calculated and stored as "cross_entropy_xgb".

Comparison of Cross Entropy Values:

A dictionary named "cross_entropy_values" is created to store the cross entropy values for each model.

The cross entropy values are printed and compared for each model.

Interpretation of Results:

The cross entropy values obtained for each model are interpreted to assess their performance in terms of probability estimation and calibration.

Lower cross entropy values indicate better calibration and more accurate probability estimation.

Based on the given cross entropy values, the XGBoost model has the lowest cross entropy (0.2796799646090161), followed by the Random Forest model (0.2891832739261127).

Therefore, the XGBoost model is considered the best among the evaluated classification models in terms of probability estimation and calibration.

Overall Solution Interpretation:

The solution compares the cross entropy values of different classification models to evaluate their performance in estimating probabilities.

The XGBoost model is identified as the best model based on the lowest cross entropy value.

It is important to consider other evaluation metrics, specific problem requirements, and potential trade-offs before making a final decision on the best model.