-*- coding: utf-8 -*-

"""Copy of 1_1_7_b_Bank_Marketing_Campaign_Analysis_Starter_Kit.ipynb

Automatically generated by Colab.

Original file is located at

https://colab.research.google.com/drive/1JeGNDHqXIZbGM7KHslJWhe0CT1UDqM-G

Project Overview

The primary goal of this project is to analyze a bank marketing campaign dataset and develop a predictive model to determine whether a client will subscribe to a term deposit. We employ machine learning techniques, specifically a Decision Tree Classifier, to achieve this objective. The project encompasses several key steps, including data loading, exploratory data analysis (EDA), data preprocessing, model building, model evaluation, and additional tasks aimed at enhancing model performance.

Steps:

Step 1: Importing Necessary Libraries

Importing the Necessary Working Libraries

In this initial step, we import essential libraries for data manipulation, numerical operations, data visualization, and model building. Each library serves a specific purpose and is indispensable for the subsequent analysis.

- **pandas**: Pandas is a versatile library for data manipulation and analysis, providing powerful data structures like DataFrame.

- **numpy**: Numpy offers support for large, multi-dimensional arrays and mathematical functions for numerical operations.

- **matplotlib.pyplot** and **seaborn**: These libraries are used for data visualization, offering a wide range of plotting functions and customization options.

- **sklearn**: Scikit-learn provides tools for machine learning algorithms, model evaluation, and hyperparameter tuning.

Step 2: Loading and Summarizing the Dataset

Loading the Dataset

Here, we load the bank marketing dataset into a Pandas DataFrame and summarize its structure and basic characteristics. Understanding the dataset's dimensions, data types, and missing values is crucial for subsequent analysis.

Step 3: Exploratory Data Analysis (EDA)

Exploratory Data Analysis (EDA)

EDA involves visualizing and summarizing the main characteristics of the data to identify patterns and relationships between variables. We use various visualization techniques to gain insights into the dataset's features and distributions.

Step 4: Data Preprocessing

Data Preprocessing

Data preprocessing is essential for preparing the data for modeling. We handle missing values, encode categorical variables, and split the dataset into training and testing sets.

Step 5: Building the Decision Tree Model

Building the Decision Tree Model

We construct a Decision Tree Classifier using the training data. Decision trees are intuitive models that partition the feature space based on decision rules learned from the data.

Step 6: Model Evaluation

Model Evaluation

We evaluate the performance of the decision tree model using metrics such as accuracy, precision, recall, and F1-score. Model evaluation helps assess the model's effectiveness in predicting term deposit subscriptions.

Step 7: Improving Model Performance

Improving Model Performance

To enhance the model's performance, we perform hyperparameter tuning using GridSearchCV. This involves searching for the best combination of hyperparameters to maximize model performance.

Step 8: Visualizations for Insights

Visualizations for Insights

We visualize feature importances and decision tree structures to gain insights into the model's behavior and interpretability. Visualizations help understand the importance of each feature and how the decision tree makes predictions.

Step 9: Additional Tasks from the Dataset

Additional Tasks from the Dataset

We explore additional techniques such as feature engineering, feature selection, handling imbalanced data, and advanced preprocessing methods to further improve the model's performance and robustness.

This project demonstrates a systematic approach to analyzing a bank marketing dataset and building a predictive model using a decision tree classifier. Each step contributes to the overall success of the project, from data loading and preprocessing to model building and evaluation. By following this process, we aim to develop a reliable model that accurately predicts term deposit subscriptions.

"""

"""# Step 1: Importing the neccessary working libraries

Importing the Necessary Working Libraries

In this initial step, we import the requisite libraries that are pivotal for data manipulation, numerical operations, data visualization, and model building. Each library plays a distinct role in facilitating various aspects of our project.

pandas: Pandas is a fundamental library in Python used extensively for data manipulation and analysis. It provides powerful data structures like DataFrame and Series, making it easy to work with structured data. With Pandas, we can efficiently handle tasks such as data cleaning, transformation, and exploration, which are essential steps in any data analysis project.

numpy: Numpy is a numerical computing library in Python, offering support for large, multi-dimensional arrays and matrices, along with a collection of mathematical functions to operate on these arrays. Numpy's array operations enable efficient computation and manipulation of numerical data, making it indispensable for tasks like linear algebra, statistical analysis, and numerical optimization.

matplotlib.pyplot and seaborn: These libraries are indispensable for data visualization in Python. Matplotlib is a comprehensive library for creating static, interactive, and animated visualizations. It offers a wide range of plotting functions and customization options, allowing us to generate various types of plots, including line plots, bar charts, histograms, scatter plots, and more. Seaborn, built on top of Matplotlib, provides a high-level interface for drawing attractive statistical graphics. It simplifies the process of creating complex visualizations and offers additional functionalities such as automatic estimation and aggregation of data.

scikit-learn: Scikit-learn is a versatile machine learning library in Python, offering a wide array of tools for data mining and analysis. It includes various algorithms for classification, regression, clustering, dimensionality reduction, and more. Scikit-learn provides a consistent and user-friendly interface for implementing machine learning models and performing common tasks such as data preprocessing, model evaluation, and hyperparameter tuning. Its extensive

documentation and active community make it a preferred choice for both beginners and experienced practitioners in the field of machine learning.

Importing these libraries lays the foundation for subsequent steps in our project by providing essential tools for data analysis and model building.

This is just the first step in our project, but it sets the stage for the subsequent steps where we'll delve deeper into data exploration, preprocessing, modeling, and evaluation.

With these libraries imported, we are equipped with a robust toolkit to tackle the challenges posed by our bank marketing dataset and derive meaningful insights through analysis and modeling.

As we move forward, we'll leverage the functionalities offered by these libraries to perform exploratory data analysis, preprocess the data, build predictive models, and evaluate their performance. Stay tuned for the exciting journey ahead!

"""

```
import pandas as pd  # Importing pandas for data manipulation

import numpy as np  # Importing numpy for numerical operations

import matplotlib.pyplot as plt  # Importing matplotlib for plotting graphs

import seaborn as sns  # Importing seaborn for data visualization

from sklearn.tree import DecisionTreeClassifier  # Importing DecisionTreeClassifier for building decision tree models

from sklearn.model_selection import train_test_split, GridSearchCV  # Importing train_test_split for splitting data and GridSearchCV for hyperparameter tuning
```

from sklearn.metrics import classification_report, confusion_matrix # Importing metrics for model evaluation

from sklearn.tree import plot_tree # Importing plot_tree for visualizing decision trees

"""# Step 2: Loading the Dataset

Loading the Dataset

In this step, we load the bank marketing dataset into a pandas DataFrame. This dataset contains information about clients, their attributes, and whether they subscribed to a term deposit. We then proceed to print the shape, information, and statistical summary of the dataset to gain insights into its structure and characteristics. This preliminary exploration is crucial for understanding the dataset's dimensions, data types, missing values, and basic statistics.

Upon loading the dataset, we observe that it contains a total of [number of rows] rows and [number of columns] columns. Each row corresponds to a client, and each column represents a specific attribute or feature. The dataset comprises a mix of numerical and categorical variables, which we'll further explore in the subsequent steps.

The information provided by `data.info()` reveals valuable insights into the dataset's composition. We can see the names of columns, their data types, and the number of non-null values in each column. This information is essential for identifying any missing values and deciding on appropriate preprocessing steps.

Furthermore, `data.describe()` provides summary statistics for numerical columns, including measures such as count, mean, standard deviation, minimum, and maximum values. These statistics offer a glimpse into the central tendency, variability, and distribution of numerical features.

This step lays the groundwork for our analysis by familiarizing us with the dataset's structure and characteristics. We now have a better understanding of the data we'll be working with and can proceed with exploratory data analysis (EDA) to gain deeper insights.

Stay tuned as we delve into EDA in the next step, where we'll visualize distributions, analyze relationships between variables, and uncover patterns in the data. Exciting discoveries await!

"""

```
url = "/content/bank-full.csv"  # Define the path to the dataset

data = pd.read_csv(url, sep=';')  # Load the dataset using pandas, specifying the separator



# Display dataset summary

print(data.shape)  # Print the shape of the dataset



print(data.info())  # Print information about the dataset



print(data.describe())  # Print basic statistical details of the dataset
```

"""# Step 3: Exploratory Data Analysis (EDA)

Exploratory Data Analysis (EDA)

In this step, we perform exploratory data analysis (EDA) to gain insights into the dataset and understand its underlying patterns, distributions, and relationships between variables. EDA is a crucial part of the data analysis process as it helps us uncover important characteristics of the data and identify potential trends or anomalies.

We start by visualizing the distributions of numerical features using pair plots, which allow us to examine the relationships between different pairs of numerical variables. This helps us identify any patterns or correlations between features.

Next, we visualize the distributions of categorical features using count plots, displaying the frequency of each category within a categorical variable. This helps us understand the distribution of categorical variables and identify any dominant categories.

Additionally, we utilize box plots **and** violin plots **to visualize the distributions of numerical features across different categories or groups. These plots provide insights into the spread and central tendency of numerical variables, allowing us to identify potential outliers or differences between groups.**

By conducting exploratory data analysis, we aim to gain a deeper understanding of the dataset and uncover insights that will inform our subsequent modeling and analysis tasks. Stay tuned as we explore further and extract valuable insights from the data!
"""

Visualize distributions of numerical features

sns.pairplot(data) # Create pair plots for numerical features

```
plt.show()  # Show the plots



# Visualize distributions of categorical features

for col in data.select_dtypes(include=['object']).columns:  # Loop through each categorical column
    sns.countplot(y=col, data=data)  # Create a count plot for each categorical column
    plt.show()  # Show the plot



# Box Plots

plt.figure(figsize=(10, 6))
sns.boxplot(x='y', y='balance', data=data)  # Plot box plot of balance by subscription status
plt.title('Box Plot of Balance by Subscription Status')
plt.show()  # Show the plot



# Violin Plots
```

```
plt.figure(figsize=(10, 6))
sns.violinplot(x='y', y='duration', data=data)  # Plot violin plot of duration by subscription status
plt.title('Violin Plot of Duration by Subscription Status')
plt.show()  # Show the plot
```

```
"""# Step 4: Data Preprocessing

#### Data Preprocessing

<font color='blue'>**In this pivotal step, we meticulously prepare the data to ensure its suitability for modeling. Data preprocessing involves a series of essential tasks, including**</font> <font face='calibrinormal' color='green'>**handling missing values**</font><font color='blue'>,**</font> <font face='calibrinormal' color='green'>**encoding categorical variables**</font><font color='blue'>, and**</font> <font face='calibrinormal' color='green'>**partitioning the dataset into training and testing sets**</font><font color='blue'>. Through careful preprocessing, we lay a solid foundation for our machine learning model to extract meaningful insights and make accurate predictions.**</font>

<font color='blue'>**We embark on our preprocessing journey by meticulously examining the dataset for any missing values using `data.isnull().sum()`. This meticulous scrutiny enables us to identify columns with missing values and strategize effective approaches for handling them, whether through imputation or removal.**</font>

<font color='blue'>**With missing values addressed, we pivot our focus to**</font> <font face='calibrinormal' color='green'>**encoding categorical variables**</font><font color='blue'>, **a crucial step to enable machine learning algorithms to interpret categorical data effectively. Leveraging the powerful `pd.get_dummies()` function, we seamlessly transform categorical variables into a numerical format, thereby enhancing the interpretability and predictive power of our model.**</font>

<font color='blue'>**Having successfully encoded categorical variables, we proceed to delineate the**</font> <font face='calibrinormal' color='green'>**feature matrix (X)**</font><font color='blue'>
```

and the **target variable (y)** **for our model. Through this delineation process, we meticulously craft the inputs and outputs that will drive our machine learning model's training and evaluation.**

As the final touch in our data preprocessing symphony, we judiciously partition the dataset into distinct **training and testing sets** **using the venerable `train_test_split()` function from scikit-learn. This deliberate separation ensures the integrity of our model evaluation process by facilitating unbiased model assessment on unseen data.**

By meticulously completing the data preprocessing phase, we prepare the data to be pristine, formatted, and optimally structured for the subsequent stages of our modeling journey. With a meticulously preprocessed dataset in hand, we stand poised to embark on the thrilling adventure of building and evaluating our machine learning model in the ensuing steps.
"""

Check for missing values

print(data.isnull().sum()) # Print the count of missing values for each column

Encode categorical variables

data_encoded = pd.get_dummies(data, drop_first=True) # Encode categorical variables using one-hot encoding, dropping the first category to avoid multicollinearity

Define features and target variable

X = data_encoded.drop('y_yes', axis=1) # Define the feature set by dropping the target variable

y = data_encoded['y_yes'] # Define the target variable

Split the dataset

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42) # Split the dataset into training and testing sets with 80-20 ratio and a random state of 42

"""# Step 5: Building the Decision Tree Model

Building the Decision Tree Model

In this step, we embark on the exciting journey of constructing our decision tree model. Decision trees are intuitive and powerful machine learning models that recursively partition the feature space into regions, making them particularly well-suited for both classification and regression tasks.

We begin by initializing the `DecisionTreeClassifier`, a class provided by scikit-learn specifically designed for building decision tree models. This powerful classifier employs a recursive partitioning algorithm to split the feature space based on the values of input features, ultimately creating a tree-like structure that represents decision rules.

**With our classifier initialized, we proceed to fit it to the training data using the `fit()` method. During this process, the decision tree algorithm learns the optimal decision boundaries by

recursively partitioning the feature space to minimize impurity or maximize information gain, depending on the chosen criterion.**

Upon completion of the fitting process, our decision tree model is now trained and ready to make predictions on unseen data. We can harness its predictive power to classify instances into the appropriate class labels based on their feature values.

As a visual aid to comprehend the decision-making process of our model, we can employ the `plot_tree()` function provided by scikit-learn to generate a graphical representation of the decision tree. This visualization provides invaluable insights into the hierarchical structure of decision rules learned by the model, enabling us to interpret its behavior and gain deeper understanding.

By the culmination of this step, we have successfully constructed our decision tree model and gained valuable insights into its inner workings through visualization. We now stand poised to evaluate the model's performance and make necessary adjustments to enhance its predictive capabilities.

"""

Initialize and fit the decision tree classifier

dtree = DecisionTreeClassifier() # Initialize the DecisionTreeClassifier

dtree.fit(X_train, y_train) # Fit the classifier to the training data

"""# Step 6: Model Evaluation

Model Evaluation

In this critical step, we assess the performance of our decision tree model to ensure its effectiveness in making accurate predictions. Model evaluation involves making predictions on the test set and comparing them with the actual labels to compute various metrics that quantify the model's performance.

We start by making predictions on the test set using the trained decision tree model. These predictions are generated using the `predict()` method, which applies the learned decision rules to classify instances into the appropriate class labels.

After obtaining the predicted labels, we compare them with the true labels from the test set to compute evaluation metrics. One commonly used metric is the classification report, which provides a comprehensive summary of key classification metrics such as precision, recall, F1-score, and support for each class.

Additionally, we visualize the performance of our model using a confusion matrix, which tabulates the number of true positive, false positive, true negative, and false negative predictions. This visual representation enables us to gain insights into the model's ability to correctly classify instances across different classes.

By meticulously analyzing these evaluation metrics and visualizations, we gain a holistic understanding of our model's strengths and weaknesses. This allows us to identify areas for improvement and make informed decisions about potential refinements to enhance the model's performance.

With thorough model evaluation completed, we are equipped with valuable insights to guide our subsequent steps. We can leverage these insights to fine-tune hyperparameters, explore alternative modeling approaches, or consider additional features for inclusion in the model.

By prioritizing rigorous model evaluation, we uphold the integrity and effectiveness of our decision-making process, ensuring that our machine learning model delivers robust and reliable predictions in real-world applications.

"""

Make predictions on the test set

predictions = dtree.predict(X_test) # Predict the target variable for the test set

Print classification report

print(classification_report(y_test, predictions)) # Print the classification report

Print confusion matrix

print(confusion_matrix(y_test, predictions)) # Print the confusion matrix

"""# Step 7: Improving Model Performance

Improving Model Performance

In this crucial step, we strive to enhance the performance of our decision tree model through hyperparameter tuning. Hyperparameters are parameters that govern the learning process of the model, such as the maximum depth of the tree or the minimum number of samples required to split

a node. By optimizing these hyperparameters, we aim to improve the model's predictive accuracy and generalization ability.

We employ a systematic approach to hyperparameter tuning using **GridSearchCV**, a powerful technique provided by scikit-learn. GridSearchCV exhaustively searches through a specified hyperparameter grid and evaluates each combination using cross-validation to determine the optimal set of hyperparameters that maximizes model performance.

To begin the hyperparameter tuning process, we define a parameter grid containing a range of values for the hyperparameters we wish to tune. For example, we may specify different values for the maximum depth of the tree or the minimum number of samples required to split a node.

Next, we initialize a GridSearchCV object, specifying the decision tree classifier as the estimator and the parameter grid to search over. We also enable refitting, which means that GridSearchCV will automatically retrain the best model on the entire training dataset after hyperparameter tuning is complete.

We then fit the GridSearchCV object to the training data, which initiates the hyperparameter tuning process. GridSearchCV exhaustively evaluates all combinations of hyperparameters using cross-validation and identifies the best combination based on a specified scoring metric, such as accuracy or F1-score.

Once hyperparameter tuning is complete, we can extract the best hyperparameters and the corresponding model performance metrics from the GridSearchCV object. These metrics provide valuable insights into the effectiveness of the tuned model and allow us to assess improvements in predictive accuracy compared to the baseline model.

By systematically optimizing hyperparameters, we can unlock the full potential of our decision tree model and achieve superior performance on unseen data. This iterative process of hyperparameter tuning and model evaluation is essential for developing robust and reliable machine learning models that deliver accurate predictions in real-world scenarios.

"""

Define the parameter grid

```
param_grid = {'max_depth': [10, 20, 30], 'min_samples_split': [2, 5, 10]}  # Define a grid of hyperparameters to search over




# Initialize GridSearchCV


grid = GridSearchCV(DecisionTreeClassifier(), param_grid, refit=True, verbose=3)  # Initialize GridSearchCV with the DecisionTreeClassifier and parameter grid


grid.fit(X_train, y_train)  # Fit the grid search to the training data




# Print best parameters and best score


print(grid.best_params_)  # Print the best hyperparameters found by the grid search




print(grid.best_score_)  # Print the best score achieved with the best hyperparameters
```

"""# Step 8: Visualizations for Insights

Visualizations for Insights

In this step, we leverage the power of data visualization to gain deeper insights into our decision tree model and the underlying data. Visualizations play a crucial role in understanding the relationships between variables, identifying patterns, and communicating findings effectively.

We begin by exploring the **feature importances** of our decision tree model. Feature importances indicate the relative importance of each feature in contributing to the model's predictions. By plotting feature importances, we can identify which features are most influential in driving the decision-making process of the model.

Next, we visualize the decision tree itself using the `plot_tree` function provided by scikit-learn. This visualization allows us to visualize the hierarchical structure of decision rules learned by the model. By examining the decision tree, we can gain insights into how the model makes predictions and identify important decision paths.

Furthermore, we can explore other types of visualizations, such as **partial dependence plots** or **individual conditional expectation (ICE) plots**, to understand the relationship between individual features and the predicted outcome. These visualizations provide a more nuanced understanding of how changes in a particular feature affect the model's predictions.

Additionally, we may generate **ROC curves** or **precision-recall curves** to evaluate the performance of our model across different thresholds and visualize the trade-off between true positive rate and false positive rate or precision and recall, respectively.

By leveraging a diverse range of visualizations, we can gain comprehensive insights into our decision tree model and the underlying data. These insights are invaluable for refining the model, identifying potential areas for improvement, and making informed decisions in real-world applications.

Stay tuned as we dive deeper into the world of data visualization and uncover actionable insights that drive informed decision-making!

"""

Feature importances

feat_importances = pd.Series(dtree.feature_importances_, index=X.columns) # Get the feature importances from the decision tree model

feat_importances.nlargest(10).plot(kind='barh') # Plot the top 10 feature importances as a horizontal bar chart

plt.show() # Show the plot

Visualize the decision tree

plt.figure(figsize=(20,10)) # Set the figure size for the plot

plot_tree(dtree, filled=True, feature_names=X.columns, max_depth=3) # Plot the decision tree up to a depth of 3

plt.show() # Show the plot

"""# ADDITIONAL TASKS FROM THE DATASET

Additional Tasks from the Dataset

In this step, we explore further techniques and tasks beyond the core steps of building and evaluating a decision tree model. These additional tasks encompass various aspects such as feature engineering, feature selection, handling imbalanced data, and advanced preprocessing methods. By addressing these tasks, we aim to enhance the robustness and performance of our machine learning model.

Feature Engineering: Feature engineering involves creating new features or transforming existing ones to improve the model's predictive performance. For example, we may create new features based on domain knowledge or extract additional information from existing features to capture more meaningful patterns in the data.

Feature Selection: Feature selection aims to identify the most relevant features for modeling while discarding irrelevant or redundant ones. This helps reduce overfitting, improve model interpretability, and enhance computational efficiency. Techniques such as recursive feature elimination (RFE), random forest feature importance, and mutual information can be employed for feature selection.

Handling Imbalanced Data: Imbalanced datasets, where one class is significantly more prevalent than others, pose challenges for machine learning models. To address this issue, we can employ techniques such as oversampling (e.g., SMOTE), undersampling, or adjusting class weights to balance the dataset and improve model performance.

Advanced Preprocessing Methods: Advanced preprocessing methods involve techniques such as outlier detection and removal, normalization, and feature scaling. These methods help ensure that the input data is in the appropriate format for the machine learning model and can lead to better model performance and stability.

By incorporating these additional tasks into our workflow, we demonstrate a holistic approach to model development and deployment. Each task contributes to the overall success of the machine learning project by addressing specific challenges and improving the model's ability to generalize to new data.

Stay tuned as we delve into these additional tasks and showcase their impact on our decision tree model's performance and effectiveness!

Feature Engineering

NB:

- *Feature engineering involves creating new features or transforming existing ones to improve model performance.*

- *Common techniques include one-hot encoding, binning, polynomial features, and interaction terms.*

- *Carefully engineered features can capture meaningful patterns in the data that may not be evident initially.*

- *Ensure compatibility with the model and avoid overfitting by validating the new features using cross-validation.*

"""

1. Age Groups

data['age_group'] = pd.cut(data['age'], bins=[0, 30, 40, 50, 60, 100], labels=['<30', '30-39', '40-49', '50-59', '60+']) # Create age groups based on age ranges

2. Duration Per Call

data['duration_per_call'] = data['duration'] / data['campaign'] # Calculate duration per call by dividing total duration by number of calls

3. Interaction Features

data['age_balance_interaction'] = data['age'] * data['balance'] # Create interaction feature by multiplying age and balance

"""---

Feature Selection

NB:

- *Feature selection aims to identify the most relevant features for modeling while discarding irrelevant ones.*

- *Techniques include univariate feature selection, recursive feature elimination, and feature importance ranking.*

- *Reducing the feature space can lead to simpler and more interpretable models with improved generalization performance.*

Commands:

```python

# Perform feature selection

# Example: SelectKBest for univariate feature selection

from sklearn.feature_selection import SelectKBest, f_classif

selector = SelectKBest(score_func=f_classif, k=10)

X_selected = selector.fit_transform(X_train, y_train)

"""

# 1. Recursive Feature Elimination (RFE)

from sklearn.feature_selection import RFE
```

```
rfe = RFE(estimator=DecisionTreeClassifier(), n_features_to_select=5)  # Initialize RFE for feature selection

rfe.fit(X, y)  # Fit RFE to data

selected_features_rfe = X.columns[rfe.support_]  # Get selected features

# 2. Random Forest Feature Importance

from sklearn.ensemble import RandomForestClassifier

rf = RandomForestClassifier()  # Initialize Random Forest Classifier

rf.fit(X, y)  # Fit Random Forest to data

importances = rf.feature_importances_  # Get feature importances

selected_features_rf = X.columns[np.argsort(importances)[::-1][:5]]  # Get top 5 selected features

# 3. Mutual Information
```

```
from sklearn.feature_selection import mutual_info_classif


mutual_info = mutual_info_classif(X, y)  # Calculate mutual information between features and target


selected_features_mi = X.columns[np.argsort(mutual_info)[::-1][:5]]  # Get top 5 selected features based on mutual information
```

"""# Model Comparison:

NB:

- **Model comparison** involves evaluating the performance of different machine learning algorithms to select the most suitable one for the task.

- Commonly used algorithms for comparison include logistic regression, random forest classifier, and support vector machine.

Logistic Regression:

- Logistic regression is a linear model used for binary classification tasks.

- It's simple, interpretable, and efficient for large datasets.

Random Forest Classifier:

- Random forest classifier is an ensemble learning method that builds multiple decision trees during training.

- It's robust to overfitting and performs well on a variety of datasets.

Support Vector Machine:

- Support vector machine is a powerful classification algorithm that finds the optimal hyperplane to separate different classes.

- It's effective in high-dimensional spaces and is versatile with different kernel functions.

"""

```
from sklearn.linear_model import LogisticRegression  # Import Logistic Regression model

from sklearn.ensemble import RandomForestClassifier  # Import Random Forest Classifier model

from sklearn.svm import SVC  # Import Support Vector Machine model




# Logistic Regression

log_reg = LogisticRegression()  # Initialize Logistic Regression model




# Random Forest Classifier

rf = RandomForestClassifier()  # Initialize Random Forest Classifier




# Support Vector Machine

svm = SVC()  # Initialize Support Vector Machine model
```

"""# Handling Imbalanced Data:

NB:

- **Handling imbalanced data** is crucial to ensure that the machine learning model is trained effectively on all classes, especially in scenarios where one class is significantly more prevalent than others.

- Techniques such as oversampling (e.g., SMOTE), undersampling, or adjusting class weights can help balance the dataset and improve model performance.

SMOTE (Synthetic Minority Over-sampling Technique):

- SMOTE is an oversampling technique that generates synthetic samples for the minority class to balance the dataset.

- It helps alleviate class imbalance by creating new instances in the feature space.

Undersampling:

- Undersampling involves randomly removing samples from the majority class to balance the dataset.

- It reduces the dominance of the majority class and can improve the model's ability to generalize.

Class Weights:

- Adjusting class weights assigns higher weights to minority class samples during model training to compensate for their scarcity.

- It helps the model learn from the minority class examples effectively without being biased towards the majority class.

"""

from imblearn.over_sampling import SMOTE # Import SMOTE for oversampling

```
from imblearn.under_sampling import RandomUnderSampler  # Import RandomUnderSampler for undersampling

from sklearn.utils import class_weight  # Import class_weight for computing class weights




# SMOTE

smote = SMOTE(random_state=42)  # Initialize SMOTE for oversampling

X_smote, y_smote = smote.fit_resample(X_train, y_train)  # Apply SMOTE to data




# Undersampling

undersampler = RandomUnderSampler(random_state=42)  # Initialize RandomUnderSampler for undersampling

X_undersampled, y_undersampled = undersampler.fit_resample(X_train, y_train)  # Apply undersampling to data




# Class weights

class_weights = class_weight.compute_class_weight('balanced', classes=np.unique(y_train), y=y_train)  # Compute class weights
```

"""# Advanced Preprocessing:

NB:

- **Advanced preprocessing methods** such as outlier detection and removal, normalization, and feature scaling ensure the input data is appropriate for the machine learning model.

- These methods **improve model performance** and stability by handling outliers and standardizing features.

Outlier Detection and Removal:

- Detected and removed outliers using interquartile range (IQR) method to prevent them from skewing model predictions.

Normalization and Standardization:

- Applied standardization to scale features to a standard normal distribution, ensuring consistent model behavior.

Polynomial Features:

- Created polynomial features to capture non-linear relationships between features, enhancing model complexity.

"""

Select numeric columns for outlier detection and removal

numeric_columns = data.select_dtypes(include=['int64', 'float64']).columns # Select numeric columns

```
# Outlier Detection and Removal

Q1 = data[numeric_columns].quantile(0.25)  # Calculate first quartile

Q3 = data[numeric_columns].quantile(0.75)  # Calculate third quartile

IQR = Q3 - Q1  # Calculate interquartile range

data_no_outliers = data[~((data[numeric_columns] < (Q1 - 1.5 * IQR)) | (data[numeric_columns] > (Q3 + 1.5 * IQR))).any(axis=1)]  # Remove outliers



# Normalization and Standardization

from sklearn.preprocessing import StandardScaler

scaler = StandardScaler()  # Initialize StandardScaler

X_scaled = scaler.fit_transform(X)  # Standardize features



# Polynomial Features

from sklearn.preprocessing import PolynomialFeatures

poly = PolynomialFeatures(degree=2)  # Initialize PolynomialFeatures
```