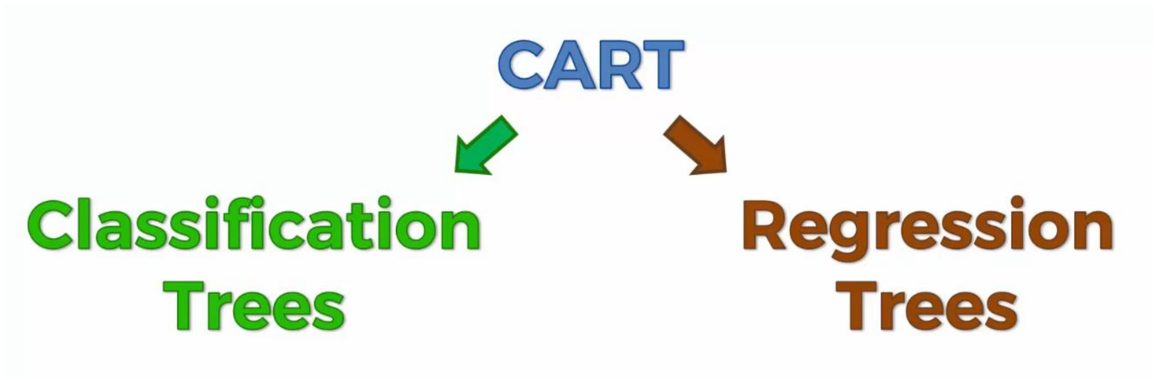
Supervised Machine Learning with Decision Tree Algorithm

In this we will study the Decision Tree algorithm, a powerful tool for classification and regression tasks.

You may have heard the term ‘CART’, which stands for Classification and Regression Tree. This umbrella term encompasses two types of trees: classification trees, which are used for categorizing data into distinct classes, and regression trees, which are used for predicting continuous values.



In this case study, we will focus on a dataset called 'Social Network Ads.' This dataset contains information about 400 customers of a car dealership that has just released a brand new luxury SUV. The strategy team gathered this data to understand which customers are most likely to buy the SUV, allowing them to target these customers with ads posted on social networks.

Each row in our dataset corresponds to a different customer, and for each customer, we have two key features: age and estimated salary. We also have a dependent variable, 'Purchased,' which indicates whether or not a customer has previously bought an SUV from this car company. The 'Purchased' column is binary: a value of 0 means the customer did not buy any previously launched SUV, and a value of 1 means the customer did.

Our goal is to train a classification model on this dataset to predict which customers will likely buy the new SUV based on their age and estimated salary. Once the model is trained, we can use it to predict for new customers whether they will buy the SUV (indicated by a predicted value of 1). This predictive capability is crucial for the strategy team, as it enables them to target potential buyers with tailored ads on social networks.

And so, without further ado, let's dive into the world of decision trees and see how we can implement this algorithm in our cloud-based development environment, Google Colaboratory!

First, we need to import the necessary libraries that will help us manipulate and visualise our data, as well as perform the calculations required for the decision tree algorithm.

```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
```

After importing these libraries, click on the play button, and we will see a small green tick on the right side of the code chunk. This suggests that the code ran successfully. Now that we have imported essential libraries, it's time to load our dataset.

```
dataset = pd.read_csv('Social_Network_Ads.csv')
```

Now that we have the essential libraries imported and our dataset loaded, let's proceed to extract the features and target variable. Before we proceed with extracting the data, we need to define our features (X) and target variable (y) from the dataset.

In our 'Social Network Ads' dataset, the features X represent the 'Age' and 'EstimatedSalary' columns. These are the independent variables we will use to predict the purchasing behavior of the customers. The target variable y corresponds to the 'Purchased' column, which indicates whether a customer bought the SUV which is 1 or did not buy the SUV which is 0.

```
X = dataset.iloc[:, :-1].values
y = dataset.iloc[:, -1].values
```

By dividing the dataset this way, we ensure that our model learns to predict the likelihood of purchase based on the customer's age and estimated salary. To build a robust model, we split our data into two parts: a training set and a testing set. The training set is used to teach the model, allowing it to learn the patterns and relationships between the features (age and estimated salary) and the target variable (purchased). The test set, on the other hand, is used to evaluate the model's performance on unseen data, helping us understand how well the model generalizes to new customers. Now, let’s split our dataset into training and testing sets to proceed with building and evaluating our model.

```
from sklearn.model_selection import train_test_split X_train, X_test,
y_train, y_test = train_test_split(X, y, test_size = 0.25, random_state = 0)
```

So we first import the train_test_split function from the sklearn.model_selection module, which helps us divide our dataset into two parts: training and testing sets. We then use this function to split our feature data X (which includes 'Age' and 'Estimated Salary') and our target variable y which includes ('Purchased'). Specifically, train_test_split(X, y, test_size = 0.25, random_state = 0) separates 75% of the data into the training set (X_train and y_train) and reserves 25% for the testing set (X_test and y_test). The test_size = 0.25 parameter ensures that one-quarter of our data is used for testing, while random_state = 0 guarantees that the split is reproducible each time we run the code. This split is essential as it allows us to train our model on one subset and validate its performance on another, ensuring it can generalize well to new, unseen data.

Let's print X_train, y_train, X_test, and y_test to see how our data is split between training and testing sets.

```
print(X_train)

print(y_train)

print(X_test)

print(y_test)
```

After splitting, we observe that X_train contains 75% of the customer data like age and estimated salary for training, paired with their purchase decisions in y_train.
Similarly, X_test holds the 25% feature data reserved for testing, with y_test containing the corresponding purchase decisions. For instance, one of the entry in X_test is [30, 87000], and its y_test counterpart is 0, indicating that a 30-year-old with an estimated salary of 87,000 did not purchase the SUV. After splitting our dataset into training and testing sets to build and evaluate our predictive model, the next step involves preprocessing the data to ensure effective model training. Here, we employ StandardScaler from sklearn.preprocessing to perform feature scaling. Feature scaling standardizes the range of independent variables or features of the data, which is crucial for many machine learning algorithms.
This process is crucial because it prevents features with larger numeric ranges from dominating those with smaller ranges, ensuring that our model learns equally from all features. Let's apply this transformation to X_train and X_test to prepare our data for model training and evaluation.

```
from sklearn.preprocessing import StandardScaler

sc = StandardScaler()

X_train = sc.fit_transform(X_train)

X_test = sc.transform(X_test)
```

In this code snippet, we instantiate a StandardScaler object sc to standardize our data. First, we use the sc.fit_transform(X_train) method on the training data (X_train). This method performs two key actions: it calculates the mean and standard deviation of each feature (column) in X_train and stores these values within the StandardScaler object—this is the "fit" step. Then, it transforms X_train by subtracting the mean and dividing by the standard deviation for each feature, centering the data around 0 with a standard deviation of 1. This scaling is essential for training the model effectively on data with consistent feature scales. For the testing data (X_test), we apply sc.transform(X_test) and here we only need the transform method because the scaler is already 'fit' to the training data's statistics, which uses the same scaling parameters (mean and standard deviation) computed from the X_train fit step. The transform method ensures that X_test is scaled in the same way as X_train, maintaining consistency and integrity in the data processing and creating a consistent foundation for our decision tree model to learn and make accurate predictions.

Now print X_train and X_test to observe the effect of this scaling and to confirm that both training and testing data are appropriately transformed and standardized.

```
print(X_train)
print(X_test)
```

By printing X_train and X_test, we can see how the data looks after scaling and ensure that our features are now on a standardized scale, which is crucial for the effective functioning of many machine learning algorithms.

Now that we have our data preprocessed, let’s move on into the training phase of our Decision Tree Classification model using the preprocessed data.

from sklearn.tree import DecisionTreeClassifier

Here, we're importing the DecisionTreeClassifier class from the sklearn.tree module of the scikit-learn library. This class allows us to create and train a decision tree model specifically designed for classification tasks. In our case, the goal is to classify customers based on their age and estimated salary (features) into two categories: likely to buy the SUV (predicted value of 1) or not likely to buy (predicted value of 0).

Continuing with the model building process, in the next we will create an instance of the DecisionTreeClassifier class we just imported. This step involves setting up the criteria for building our decision tree and configuring it with some initial settings.

classifier = DecisionTreeClassifier(criterion = 'entropy', random_state = 0)

Here we assign the classifier to a variable named classifier. This object will represent our decision tree model, think of it like creating a new building from a pre-designed template (the class), and "classifier" refers to a newly created decision tree classification model.

Here we also provide two parameters within the DecisionTreeClassifier function like the criterion for splitting (e.g., 'entropy' or 'gini') and the random_state for reproducibility. So in the first parameter criterion='entropy', this parameter specifies the method the decision tree will use to select the best splitting feature at each node. Here, we're using 'entropy,' which is a measure that determines the quality of a split, as studied previously. It calculates the randomness or impurity in the data and aims to make splits that reduce entropy, resulting in more homogeneous nodes. The goal is to create branches where the data is more organized and less mixed in terms of the target variable ('Purchased' in our case).

Next we have random_state=0, this parameter helps ensure reproducibility in the model creation process. By setting a fixed random state (here, 0), we guarantee that whenever we run this code, we'll get the same initial random seed for splitting the data. This allows for consistent results and easier debugging if needed.

Now that we have a well-defined decision tree model stored in the classifier variable, it's time to train it. We proceed to train the model using the fit method.

classifier.fit(X_train, y_train)

This is where the learning happens. Imagine the classifier as a student eager to learn from an expert teacher. In this case, the teacher is the training data, and the student is our decision tree model. Here we provide the training features (X_train) and the corresponding target variable (y_train) to the fit method. This data acts as the training examples for the decision tree model. The decision tree classifier analyzes the training data. It starts at the root node and considers all the features (age, salary) to determine the best splitting rule. Using the chosen criterion (entropy in our case), it creates branches that separate the data based on the feature values that minimize impurity. This process continues recursively until the data at each leaf node is sufficiently homogeneous regarding the target variable (purchased or not purchased). Here essentially, the decision tree is learning the relationships between the features and the purchase behavior based on the training examples. This knowledge is captured in the structure of the tree, with each split representing a decision rule and each leaf node representing a prediction outcome (likely to buy or not). And so by the end of the fit process, our classifier object is no longer a naive model but a trained decision tree that can make informed predictions based on the patterns it learned from the training data and ready to be used for making predictions on new, unseen data. Once the model is trained, we can use it to predict outcomes for new data. For example, let's predict whether a 30-year-old customer with an estimated salary of 87,000 will purchase the SUV: ect, which we create by calling the KMeans class. We then add parentheses.

print(classifier.predict(sc.transform([[30,87000]])))

Here we used the predict method of our trained classifier to predict the purchase decision for a new customer. Before making the prediction, we need to transform the input data using the same scaling applied to our training data (sc.transform([[30, 87000]])). This ensures that the new data point is scaled consistently with the training data.

The predict method outputs [0], indicating that, based on the model's learned patterns, a 30-year-old with an estimated salary of 87,000 is predicted not to purchase the SUV.

By checking the Social_Network_Ads.csv dataset, we can confirm that the specific entry for a 30-year-old with an estimated salary of 87,000 indeed has a Purchased value of 0, indicating that they did not buy the SUV. This corroborates our model's prediction, showing that it correctly identified this customer's purchasing behavior based on the patterns it learned during training.

Having successfully predicted the outcome for a single customer, let's now evaluate our model's performance on the entire test set to see how well it generalizes to unseen data.

y_pred = classifier.predict(X_test)

Here we apply the predict method to the X_test data to get the predicted purchase decisions (y_pred) for the customers in the test set. Remember, the testing set wasn't used for training, so these predictions represent the model's performance on unseen data. The output (y_pred) will be a NumPy array containing predicted purchase decisions (0 or 1) for each customer in the testing set. These predictions represent what our model has learned about which customers are likely to purchase the SUV, based on their age and estimated salary.

Next we will compare the predicted values (y_pred) with the actual outcomes (y_test).

```
print(np.concatenate((y_pred.reshape(len(y_pred),1),
y_test.reshape(len(y_test),1)),1)) ;
```

The predicted values (y_pred) and the actual purchase decisions (y_test) might have different shapes in NumPy arrays. To prepare them for a side-by-side comparison, we reshape them into column vectors using reshape(len(y_pred),1), where the reshape function in NumPy changes the shape of an array without changing its data. It allows us to specify a new shape for the array. len(y_pred) gives the number of elements in y_pred. It counts how many predictions we have, and 1 specifies that we want just one column. By putting 1 here, we are telling reshape to put all the elements into a single column. This ensures they have the same dimensions for the next step.

Next we use np.concatenate to combine the reshaped y_pred (predictions) and y_test (actual values) into a single array. This creates a table-like structure where each row shows a customer's predicted purchase decision next to their actual purchase decision from the testing set. By printing this concatenated array, we can visually inspect how well our model's predictions align with the real purchase behaviors. This can provide valuable insights into the model's accuracy and identify areas for potential improvement.

When we run this code, we obtain the following output:

In this output, the first column represents the predicted purchase decisions (y_pred) by our model. The second column shows the actual purchase decisions (y_test) from the test data. A pair like [0 0] means that the model correctly predicted that the customer would not purchase the SUV.

A pair like [1 1] indicates that the model correctly predicted that the customer would purchase the SUV. And discrepancies, such as [1 0] or [0 1], show instances where the model's prediction did not match the actual outcome.

By visually examining this table, we can gain valuable insights into the model's performance. Ideally, we would see a high number of rows where the predicted value in the first column matches the actual value in the second column. This indicates that the model is accurately predicting purchase behaviors for unseen data. In some rows, you will notice, the model predicted a purchase (1) but the actual outcome was no purchase (0). Such discrepancies highlight areas where the model might need improvement or where the data patterns are more challenging to predict accurately.

Moving on

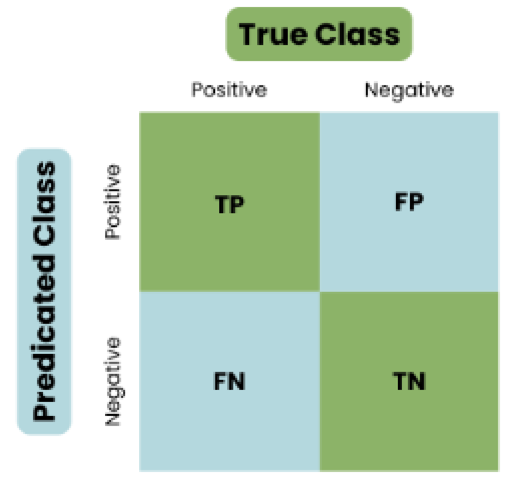
While visually inspecting the predicted and actual values gives us a preliminary understanding of our model's performance, it doesn't provide a complete picture. To quantitatively measure this performance, we can use specific metrics from the sklearn.metrics library, such as the confusion matrix and the accuracy score.

```
from sklearn.metrics import confusion_matrix, accuracy_score

cm = confusion_matrix(y_test, y_pred)

print(cm)

accuracy_score(y_test, y_pred)
```

The first metric which is the confusion matrix, is a summary of prediction results on a classification problem. It shows the number of correct and incorrect predictions made by the model compared to the actual outcomes. Therefore the the confusion_matrix function helps us visualize the agreement between the predicted and actual values.

The matrix helps us understand how well the model is performing in terms of the four key metrics: True Positives (TP), True Negatives (TN), False Positives (FP), and False Negatives (FN).

Here's how the confusion matrix is structured:



Where TN is True Negative, FP is False Positive, FN is False Negative, and lastly TP is True Positive. Now let’s understand these categories in detail.

- True Negative is the number of instances where the model correctly predicted no purchase (0) and the actual outcome was also no purchase (0). This represents a successful prediction of a negative class (no purchase).
- False Positive is the number of instances where the model predicted a purchase (1) but the actual outcome was no purchase (0). These are also known as Type I errors. In our case, a Type I error refers to mistakenly predicting a customer would buy the SUV (positive class) when they actually didn't (negative class). This can lead to wasted marketing efforts or incorrect resource allocation. Next is
- False Negative is the number of instances where the model predicted no purchase (0) but the actual outcome was a purchase (1). These are also known as Type II errors. Here, a Type II error

signifies failing to predict a customer would buy the SUV (positive class) when they actually did (negative class). And lastly we have

- True Positive which is the number of instances where the model correctly predicted a purchase (1) and the actual outcome was also a purchase (1). This represents a successful prediction of a positive class (purchase).

  Next we have is accuracy score which is calculated by the accuracy_score function which calculates the overall accuracy of the model. It's simply the ratio of correctly predicted instances (TP + TN) to the total number of instances. It gives us a single number that reflects the proportion of correct predictions made by the model out of all predictions. It is calculated as:

- Accuracy = (TP + TN) / (TP + TN + FP + FN)

Now that we've unpacked the meaning behind the confusion matrix and accuracy score, let's put this knowledge into action! We can use the confusion_matrix and accuracy_score functions from the sklearn.metrics library to evaluate our model's performance on the unseen testing data. Now, let's run the code to see these evaluations in action. By running this code, we will get a clear, numerical representation of our model's performance.

The output [[62 6] [3 29]] is the confusion matrix which is a 2x2 table for our predictions on the testing data, it summarizes the agreement between our model's predictions (y_pred) and the actual purchase decisions (y_test) in the testing data. Given we had 100 testing data points, this matrix summarizes how our model performed:

So on the top left of this 2x2 table is 62, which is True Positive, which means that the model correctly predicted that 62 customers would purchase the SUV (positive class), and they indeed did (positive class according to the actual data).

Next on the top right of this table we have 6 which is False Positive, which means that the model incorrectly predicted that 6 customers would purchase the SUV (positive class) when they actually didn't (negative class according to the actual data).

Now on the bottom left we have 3 which is false negative, which means that the model missed 3 customers who actually intended to buy the SUV (positive class according to the actual data) but the model predicted they wouldn't (negative class), or we can say that the model incorrectly predicted 'no purchase' for 3 instances where the actual outcome was 'purchase'.

And lastly on the bottom right we have 29, these are True Negatives (TN), which means that the model correctly predicted that 29 customers would not purchase the SUV (negative class), and they indeed didn't (negative class according to the actual data). Now next is .91, which is the accuracy score, and it means that our model correctly predicted 91% of the instances in the testing set. This is computed as the ratio of the sum of True Positives and True Negatives to the total number of test instances:

Accuracy=(TP+TN)/TP+TN+FP+FN=62+29/100=0.91

This high accuracy indicates that the model performs well in predicting whether a customer will purchase the SUV based on their age and estimated salary.

In summary, the confusion matrix and accuracy score together provide a comprehensive view of our model's performance. The confusion matrix offers a granular breakdown of each prediction category, while the accuracy score gives an overall performance measure. This helps us understand not just how often our model is correct, but also where it tends to make mistakes (such as predicting purchases that don't happen or missing actual purchases). This insight is crucial for evaluating and improving our predictive model.

To gain a deeper understanding of the model's decision boundaries and how well it separates the different classes (purchase vs. no purchase) in the training data, we can employ data visualization techniques.

We are about to plot a 2D graph. This plot has two axes: the X-axis represents the customers' age, and the Y-axis represents their estimated salary. Each point on this plot corresponds to an individual customer, either from our training set or our test set. What makes this plot particularly insightful is its depiction of prediction regions: areas where our decision tree model predicts class 0 (indicating the customer didn't purchase the SUV) and class 1 (indicating the customer did purchase the SUV). This visual representation will provide a clear understanding of how well our model distinguishes between these two outcomes. Let's first visualize the training set results to explore how our decision tree model categorizes customers based on age and estimated salary.

```
from matplotlib.colors import ListedColormap
X_set, y_set = sc.inverse_transform(X_train), y_train
X1, X2 = np.meshgrid(np.arange(start = X_set[:, 0].min() - 10, stop =
X_set[:, 0].max() + 10, step = 0.25), np.arange(start = X_set[:, 1].min() -
1000, stop = X_set[:, 1].max() + 1000, step = 0.25))
plt.contourf(X1, X2,
classifier.predict(sc.transform(np.array([X1.ravel(),
X2.ravel()]).T)).reshape(X1.shape), alpha = 0.75, cmap =
ListedColormap(('red', 'green')))
plt.xlim(X1.min(), X1.max())
plt.ylim(X2.min(), X2.max())
for i, j in enumerate(np.unique(y_set)):
plt.scatter(X_set[y_set == j, 0], X_set[y_set == j, 1], c =
ListedColormap(('red', 'green'))(i), label = j)
```

plt.title('Decision Tree Classification (Training set)')

plt.xlabel('Age')

plt.ylabel('Estimated Salary')

plt.legend()

Now let’s run this code cell and plot the elbow method graph.

plt.show()

Here we import ListedColormap from matplotlib.colors to create a colormap for different classes in the plot. Next we use the inverse_transform method of our scaler sc to revert the scaled features in X_train back to their original scales. This allows us to plot the data points in their original age and salary scales, making the plot more interpretable. We create a grid of values that cover the full range of our features, which are age and salary. This grid helps us draw the decision boundary, which is the line that separates the different classes (purchase or no purchase) that our model predicts. We use the np.meshgrid function to generate a grid of coordinates that extends a bit beyond the minimum and maximum values of age and salary in our dataset. This extra space around the data points makes the graph easier to see and understand. The decision boundary shows where the model changes its prediction from one class to another. In our graph, this boundary separates the areas where the model predicts a customer will not buy the SUV (red) from where it predicts they will buy the SUV (green).

Next we transform the grid points using the same scaler sc applied during model training, predict the class for each point on the grid using our classifier, and reshape the predictions to match the grid shape. The plt.contourf function fills the grid with colors corresponding to the predicted classes, creating a visual representation of the decision boundary. The colors 'red' and 'green' denote the two classes:

customers who are predicted not to purchase and those who are predicted to purchase, respectively.

We then set the limits of the plot to match the range of our grid and plot the original data points (X_set) using plt.scatter. Each point is colored according to its true class, with 'red' and 'green' matching the decision boundary colors.

And Finally, we add a title, axis labels, and a legend to make the plot informative and easy to interpret.

As mentioned previously

X-axis (Age): Represents the age of the individuals.

Y-axis (Estimated Salary): Represents the estimated salary of the individuals.

The background is divided into red and green regions.

Red Region: Represents the areas where the model predicts a 'no purchase' (class 0).

Green Region: Represents the areas where the model predicts a 'purchase' (class 1).

These regions show the decision boundaries created by the Decision Tree classifier based on the features (age and estimated salary).

Red Dots: Represent actual instances where the individual did not purchase the product (class 0).

Green Dots: Represent actual instances where the individual did purchase the product (class 1).

Next is the Decision Boundaries:

The boundaries between the red and green regions are determined by the decision rules of the Decision Tree classifier.

The intricate patterns and boxes show how the model has segmented the feature space to classify the data points.

And legend indicates the class labels for the data points:

Red (0): No purchase.

Green (1): Purchase.

A mesh grid is used to predict the class of each point in the feature space, which helps to create the decision boundary visualization.

Imagine placing a grid over the whole area of the graph, with many tiny squares. For each square in the grid, we ask the model to predict whether it falls in the 'purchase' or 'no purchase' area.

This process helps us visualize the decision boundaries. The grid allows us to see clearly where the model predicts one class over the other, forming the red and green regions we see in the background.

In a similar manner to visualizing the training set, we can visualize the test set results to see how well the model generalizes to new, unseen data. The code for plotting the test set results is very similar to the training set code, with just a slight change in the data used:

```
from matplotlib.colors import ListedColormap
X_set, y_set = sc.inverse_transform(X_test), y_test
X1, X2 = np.meshgrid(np.arange(start = X_set[:, 0].min() - 10, stop = X_set[:, 0].max() + 10, step = 0.25), np.arange(start = X_set[:, 1].min() - 1000, stop = X_set[:, 1].max() + 1000, step = 0.25))
plt.contourf(X1, X2, classifier.predict(sc.transform(np.array([X1.ravel(), X2.ravel()]).T)).reshape(X1.shape), alpha = 0.75, cmap = ListedColormap(('red', 'green')))
plt.xlim(X1.min(), X1.max())
plt.ylim(X2.min(), X2.max())
for i, j in enumerate(np.unique(y_set)):
```

```
plt.scatter(X_set[y_set == j, 0], X_set[y_set == j, 1], c = ListedColormap(('red', 'green'))(i), label = j)
plt.title('Decision Tree Classification (Test set)')
plt.xlabel('Age')
plt.ylabel('Estimated Salary')
plt.legend()
plt.show()
```

This test set visualization code follows the same steps as the training set visualization: creating a color map, setting up a mesh grid, predicting classes for each point in the grid, and plotting the data points. The key difference is that we use X_test and y_test instead of X_train and y_train. This means the plot shows how the model performs on new, unseen data rather than the data it was trained on.

In this plot, you can observe how well the model's predictions align with the actual test data points. Ideally, we want the red dots to be in the red regions and the green dots in the green regions. However, some misclassifications can occur, which are shown by dots that do not match the color of their region.

The alignment of red dots in red regions and green dots in green regions suggests that the model is correctly identifying a significant number of cases.
And the presence of red dots in green regions and green dots in red regions highlights instances where the model's predictions do not match the actual outcomes. These misclassifications provide insights into areas where the model may need improvement. And with this we come to an end of this segment.

Through this segment, we gained valuable hands-on experience building and evaluating a decision tree classification model. Remember that decision trees are just one type of machine learning algorithm, and there are many others available with their own strengths and weaknesses. As you continue your journey in machine learning, explore different algorithms and techniques to find the best solution for your specific data and problem.