😊 Decision Tree Algorithm

A decision tree algorithm is a method used in machine learning to make decisions based on certain features or characteristics of data. You can think of it like a flowchart where each question helps narrow down the choices until you arrive at a conclusion or prediction.

How It Works

1. **Tree Structure**: Imagine a tree where:
    - Each **branch** represents a question about the data.
    - Each **leaf** (end point) represents a final decision or outcome.
2. **Asking Questions**: The algorithm starts at the top of the tree and asks questions based on the features of the data. Each question splits the data into subsets.
3. **Making Predictions**: By following the branches based on the answers to the questions, you eventually reach a leaf node that provides the predicted outcome.

Example

Let's say you want to predict whether someone will buy a product based on a few questions about them.

- **Root Node (Top Question)**:
    - "Is the customer older than 30?"
        - **Yes**: Go to the next question.
        - **No**: Leaf node: "Not likely to buy."
- **Next Question** (if yes):
    - "Does the customer have a high income?"
        - **Yes**: Leaf node: "Likely to buy."
        - **No**: Leaf node: "Not likely to buy."

Simple Example

Consider a decision tree for classifying fruit based on color and size:

- **Root Node**: "Is the fruit red?"
    - **Yes**:
        - "Is it small?"
            - **Yes**: Leaf node: "Cherry."
            - **No**: Leaf node: "Apple."
    - **No**:

        - "Is it yellow?"
            - **Yes**: Leaf node: "Banana."
            - **No**: Leaf node: "Unknown fruit."

Benefits

- **Easy to Understand**: Decision trees are straightforward and visual, making them easy to interpret.
- **No Need for Feature Scaling**: They don't require normalization of data.
- **Versatile**: Can be used for both classification (categorical outcomes) and regression (numerical outcomes).

Drawbacks

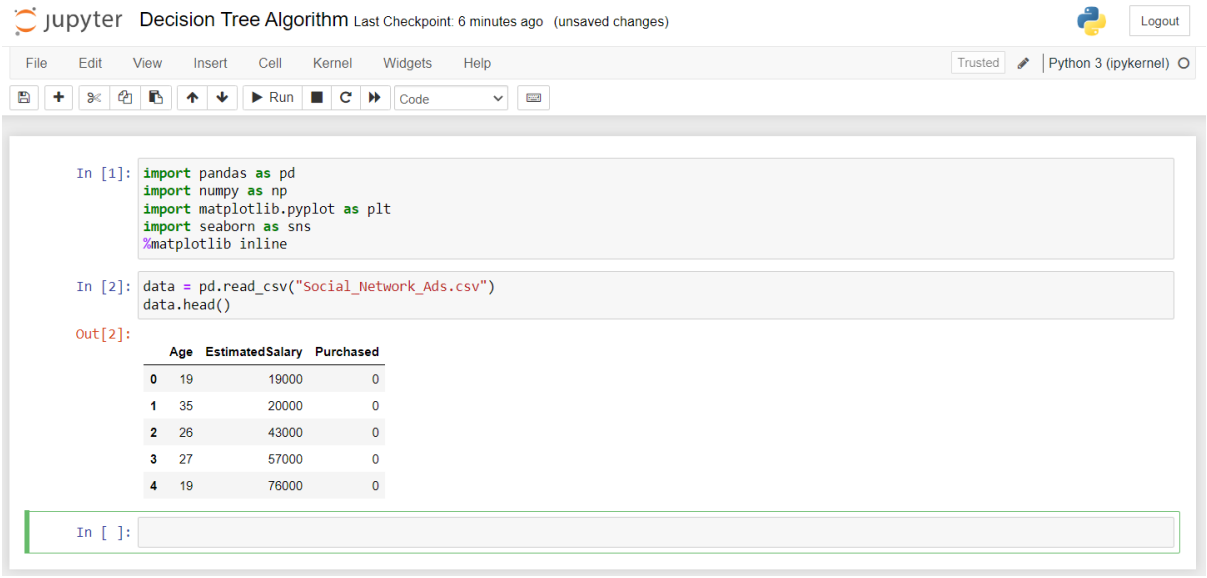
- **Overfitting**: Decision trees can become too complex and fit the noise in the data rather than the actual pattern.
- **Instability**: Small changes in the data can lead to different tree structures.

In summary, a decision tree is like a series of questions that help you make a prediction or classification based on the features of your data. It's intuitive and easy to visualize!

😄 To begin with the Lab:

1. In your Jupyter Notebook, upload Social Network Ads CSV file and create a new Python Kernel.

2. Then we import some imported libraries and load our dataset as a data frame and display some of its entries.



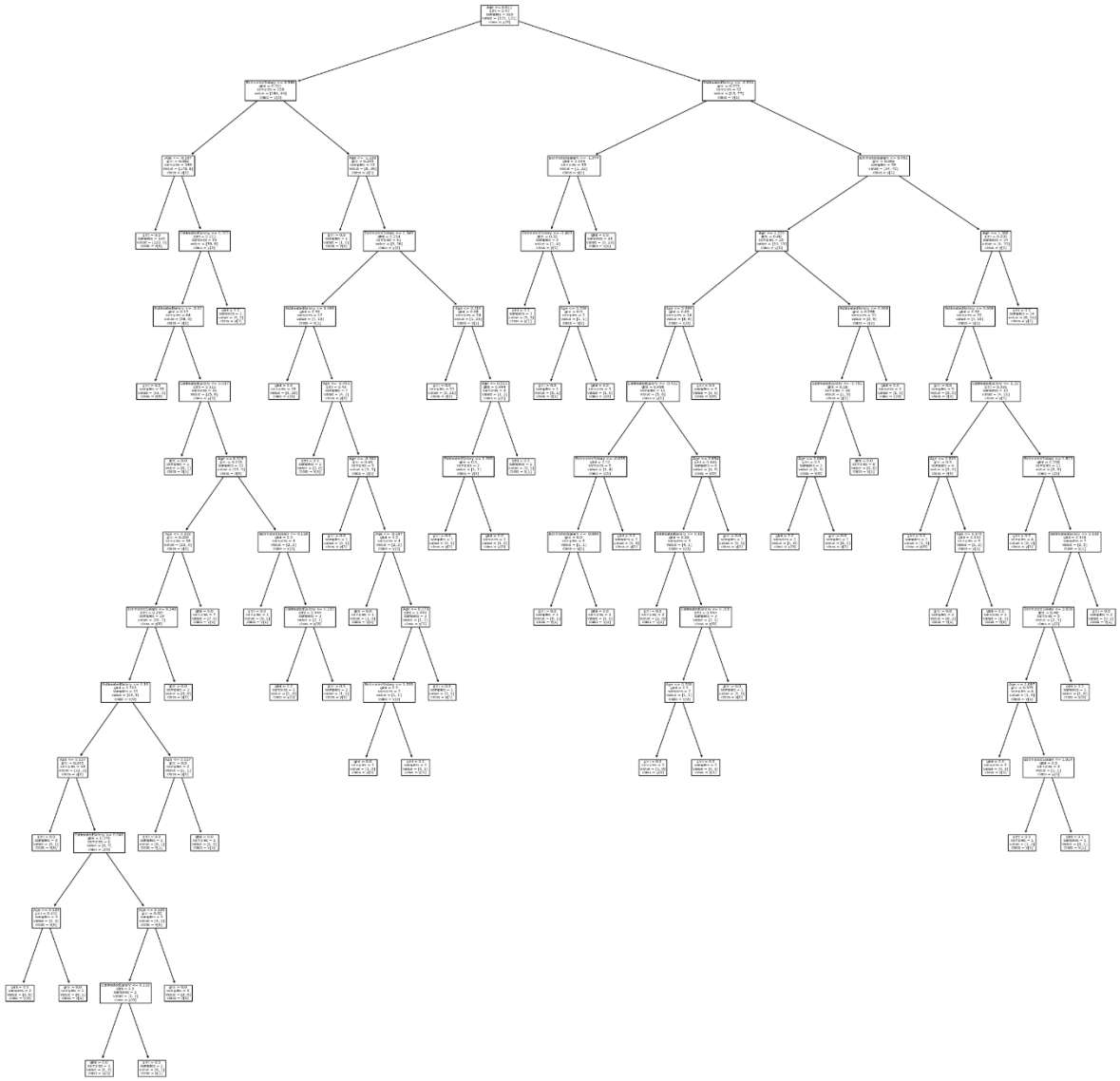
3. Now, what I'm going to do is first, I'm going to separate my features and the target.
4. The code is setting up a machine learning workflow to predict whether someone will purchase a product. It separates the data, trains a decision tree model on part of the data, makes predictions on another part, and then evaluates how well the model did.
5. The accuracy score for this dataset is **0.9**, which means the model has **90% accuracy**.
6. When we calculate the **F1 score** for this dataset, we find it to be **0.8260**.
7. So, this is the overall performance summary we get from the **decision tree classifier** that we have trained on this dataset.

```
In [3]: X = data.drop(columns='Purchased')
        y = data['Purchased']

        from sklearn.model_selection import train_test_split
        X_train,X_test,y_train,y_test = train_test_split(X,y,test_size=0.2, random_state=0)

        from sklearn.preprocessing import StandardScaler
        ss = StandardScaler()
        X_train_transform = ss.fit_transform(X_train)
        X_test_transform = ss.transform(X_test)

In [4]: from sklearn.tree import DecisionTreeClassifier
        tree = DecisionTreeClassifier(random_state=0)
        tree.fit(X_train_transform,y_train)

Out[4]: DecisionTreeClassifier
        DecisionTreeClassifier(random_state=0)

In [5]: y_pred_test = tree.predict(X_test_transform)

In [6]: from sklearn.metrics import accuracy_score,f1_score
        accuracy_score(y_test,y_pred_test)

Out[6]: 0.9

In [7]: f1_score(y_test,y_pred_test)

Out[7]: 0.8260869565217391
```

8. Now, let's visualize the **decision tree** that we just created.
9. To do this, we'll use a function called **plot_tree**.
10. Looking at the documentation for **plot_tree**, the first parameter it requires is the **decision tree model** itself.
11. Below you can see the decision tree.

In [8]:
```
from sklearn.tree import plot_tree
```

In [9]:
```
plt.figure(figsize=(30,30))
plot_tree(tree,feature_names=X.columns,class_names=True)
plt.show()
```



12. I will evaluate the model and calculate the **F1 score** using the training data. I'll use **y_pred_train** to predict the values for **Y_train**. Next, I'll validate this using **Y_train** and **y_train**.
13. Now, when I check the F1 score on the training data, I get an F1 score of **0.9958**.
14. This indicates that there is clearly an **overfitting** issue, meaning the model performs extremely well on the training data but may not generalize well to new, unseen data.

```
In [10]: y_pred_train = tree.predict(X_train_transform)
         f1_score(y_train,y_pred_train)

Out[10]: 0.995850622406639
```

15. **Overfitting** and **underfitting** are two important concepts in machine learning that can lead to poor performance of a model.
    Overfitting occurs when a model learns the training data too well, including its noise and outliers, resulting in poor performance on new data.
    Underfitting happens when a model is too simple and fails to capture the underlying patterns in the data.
16. Both issues can negatively impact how well a machine-learning model works.
17. If I want to implement a k-fold cross-validation scenario, I'll start by importing KFold from the sklearn.model_selection module. I'll also import the numpy library. Next, I'll create a random dataset using NumPy.

In [11]:
```
from sklearn.model_selection import KFold
import numpy as np
X_a = np.random.randint(0,10,(4,2))
y_a = np.array([1,2,3,4])
print(X_a)
kf = KFold(n_splits=2)

for train_index,test_index in kf.split(X_a):
    print(f"Train: {train_index}, Test: {test_index}")
    X_train_a,X_test_a = X_a[train_index],X_a[test_index]
    print(X_train_a)
    print(X_test_a)
    y_train_a,y_test_a = y_a[train_index],y_a[test_index]
    print(y_train_a)
    print(y_test_a)
```

```
[[5 7]
 [5 4]
 [7 8]
 [7 8]]
Train: [2 3], Test: [0 1]
[[7 8]
 [7 8]]
[[5 7]
 [5 4]]
[3 4]
[1 2]
Train: [0 1], Test: [2 3]
[[5 7]
 [5 4]]
[[7 8]
 [7 8]]
[1 2]
[3 4]
```

18. We use the k-fold cross-validation technique during hyperparameter tuning. When tuning hyperparameters, we apply k-fold validation along with the hyperparameter tuning algorithm. This approach helps us evaluate the model more effectively by ensuring that we test its performance on different subsets of the data.
19. **Hyperparameters** are settings that we configure before training a machine learning model.
20. In machine learning, it's important to differentiate between **parameters** and **hyperparameters**:
21. **Parameters** are the internal variables that the learning algorithm estimates during training, based on the given dataset. These include weights and biases that help the model make predictions.

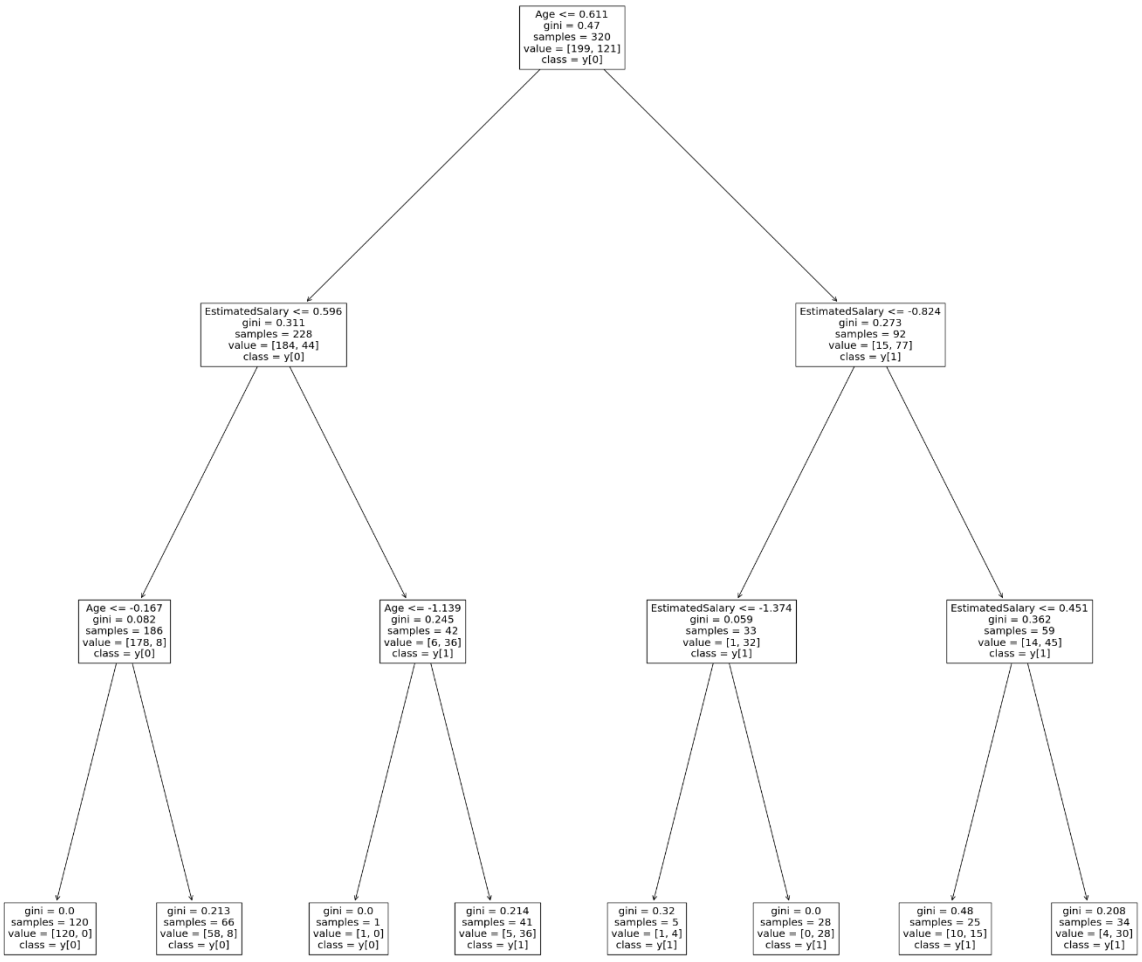
22. **Hyperparameters**, on the other hand, are set before training and control the learning process itself. They dictate things like the learning rate, the number of trees in a random forest, or the number of clusters in k-means.
23. So, while the algorithm learns the parameters from the data, hyperparameters are chosen beforehand and can significantly affect the model's performance.
24. If I want to see the default hyperparameters for the **decision tree algorithm**, here are some of the default values:
    alpha: 0
    max_depth: None
    max_features: None
25. These default values are used to help the model learn the patterns between **X** (input features) and **Y** (target variable).

```
In [12]: tree.get_params()

Out[12]: {'ccp_alpha': 0.0,
          'class_weight': None,
          'criterion': 'gini',
          'max_depth': None,
          'max_features': None,
          'max_leaf_nodes': None,
          'min_impurity_decrease': 0.0,
          'min_samples_leaf': 1,
          'min_samples_split': 2,
          'min_weight_fraction_leaf': 0.0,
          'random_state': 0,
          'splitter': 'best'}
```

26. Now, when I refer to a **manual technique**, I'm interested in finding the patterns between **X** (input features) and **Y** (target variable) to achieve better performance.
27. To illustrate this, when initializing the **decision tree algorithm**, I will set specific values for some hyperparameters. I'll call this model **tree2**.
28. During this initialization, I will train the model using **X_train** and **Y_train**, along with a **random_state** for reproducibility. One of the hyperparameters I'll specify is **max_depth**, which I will set to **3**.
29. With the maximum depth set to 3, I will then proceed to make predictions on both my training data and my test data.

In [13]:
```
from sklearn.tree import DecisionTreeClassifier
tree2 = DecisionTreeClassifier(random_state=0, max_depth=3)
tree2.fit(X_train_transform,y_train)
y_pred_test = tree2.predict(X_test_transform)
y_pred_train = tree2.predict(X_train_transform)

print(f"f1-score on training data is : {f1_score(y_train,y_pred_train)}")
print(f"f1-score on test data is : {f1_score(y_test,y_pred_test)}")
plt.figure(figsize=(30,30))
plot_tree(tree2,feature_names=X.columns,class_names=True)
plt.show()
```

```
f1-score on training data is : 0.889763779527559
f1-score on test data is : 0.9130434782608695
```

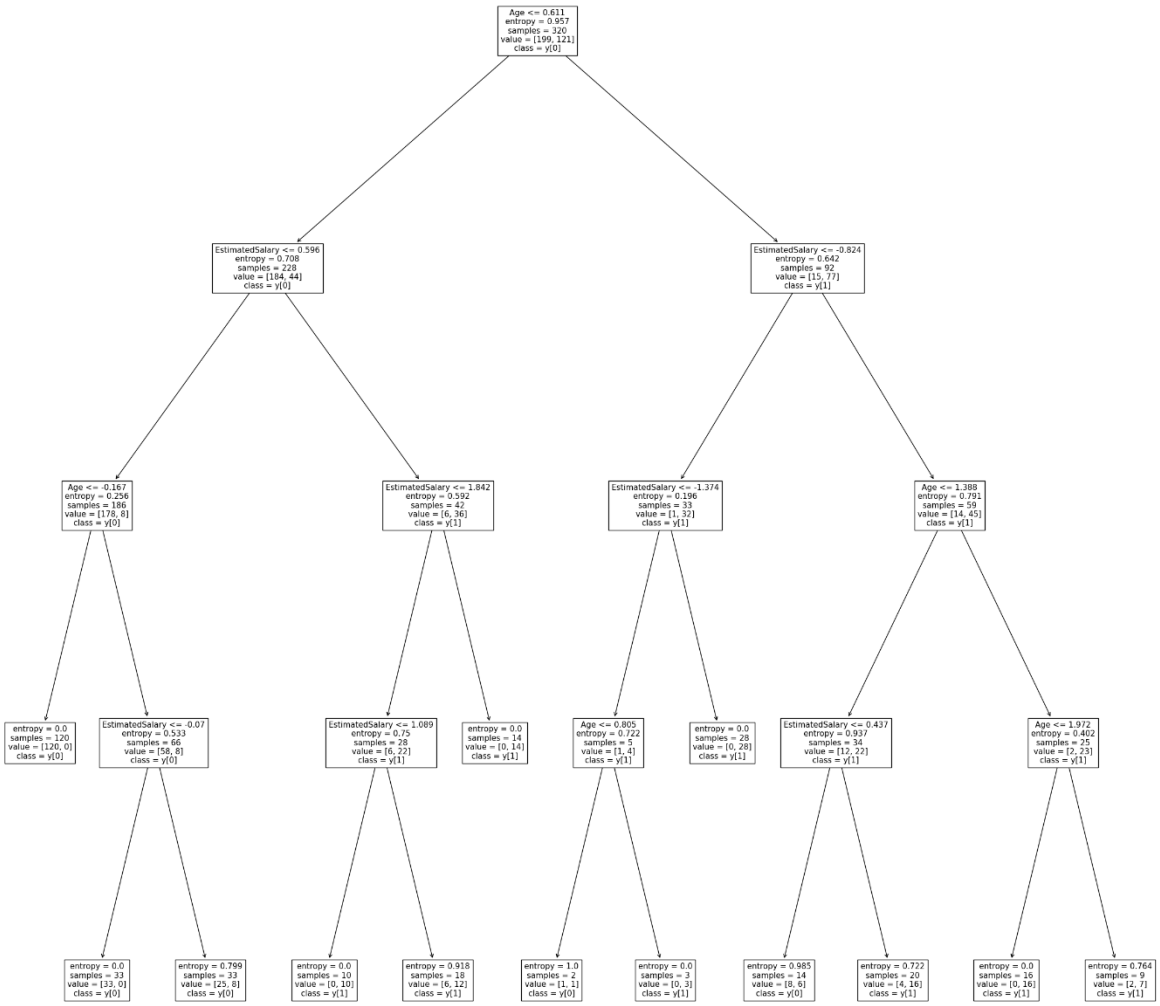


30. When I set the **maximum depth** to **3** and use the **Gini criterion** (the default criterion), the F1 score for my training data is **0.8897**, and for my test data, it is **0.91**. This shows how the decision tree performs with the maximum depth limited to 3.

31. Now, I'll change the criterion while keeping the same hyperparameters. This time, I'll set the **criterion** to **entropy**.

In [14]:
```
from sklearn.tree import DecisionTreeClassifier
tree2 = DecisionTreeClassifier(random_state=0,criterion='entropy', max_depth=4)
tree2.fit(X_train_transform,y_train)
y_pred_test = tree2.predict(X_test_transform)
y_pred_train = tree2.predict(X_train_transform)

print(f"f1-score on training data is : {f1_score(y_train,y_pred_train)}")
print(f"f1-score on test data is : {f1_score(y_test,y_pred_test)}")
plt.figure(figsize=(30,30))
plot_tree(tree2,feature_names=X.columns,class_names=True)
plt.show()
```

```
f1-score on training data is : 0.8870292887029289
f1-score on test data is : 0.9090909090909091
```



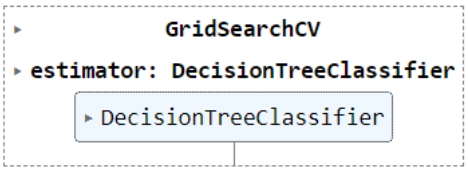
32. From this, we can see that by modifying the hyperparameters of the **decision tree algorithm**, I get different outcomes. This clearly indicates that tuning hyperparameters is essential for improving model performance.
33. However, manual tuning can be challenging because it's often difficult to find the best values for these hyperparameters to achieve an optimal model. This manual approach isn't practical in every situation.
34. For example, if I have to test **120 different combinations** of hyperparameters, I would need to create **120 models** manually and then compare their performances, which is very time-consuming.
35. Therefore, we need an **automated method** for hyperparameter tuning, and one such method is **grid search**.
36. You can think of grid search as a brute-force approach to hyperparameter tuning, where it systematically explores different combinations of hyperparameter values to find the best model performance.

In [15]:
```
from sklearn.model_selection import GridSearchCV
```

In [16]:
```
tree = DecisionTreeClassifier(random_state=42)
param_grid = {"criterion":["gini", "entropy", "log_loss"],
              "max_depth":[2,4,5,6],
              "min_samples_split":[4,2,6]}
# Total model combinations = 3*4*3 = 36
# kfold = 5
# Total iterations = 36*5 = 180
grid = GridSearchCV(tree, cv=5,param_grid=param_grid,scoring='accuracy',verbose=1)
grid.fit(X_train_transform,y_train)
```

```
Fitting 5 folds for each of 36 candidates, totalling 180 fits
```

Out[16]:



In [17]:
```
grid.best_estimator_
```

Out[17]:

```
DecisionTreeClassifier(max_depth=2, min_samples_split=4, random_state=42)
```

In [18]:
```
grid.best_params_
```

Out[18]:
```
{'criterion': 'gini', 'max_depth': 2, 'min_samples_split': 4}
```

In [19]:
```
grid.best_score_
```

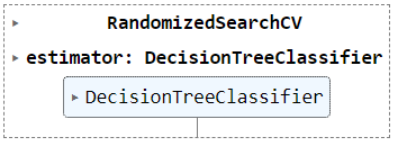
Out[19]:
```
0.9
```

37. In addition to the GridSearchCV technique, there's another method for hyperparameter tuning called the random search technique.
38. Unlike grid search, which tests all possible combinations of hyperparameters, random search samples a specified number of hyperparameter combinations from a predefined set of values. This can often lead to finding good parameter values more quickly, especially in cases where the search space is large.

In [20]:
```
# Randomsearch
from sklearn.model_selection import RandomizedSearchCV
tree = DecisionTreeClassifier(random_state=42)
param_grid = {"criterion":["gini", "entropy", "log_loss"],
              "max_depth":[2,4,5,6],
              "min_samples_split":[4,2,6]}
random = RandomizedSearchCV(tree, cv=5,param_distributions=param_grid,scoring='accuracy',verbose=1)
random.fit(X_train_transform,y_train)
```

```
Fitting 5 folds for each of 10 candidates, totalling 50 fits
```

Out[20]:



In [21]:
```
random.best_params_
```

Out[21]:
```
{'min_samples_split': 6, 'max_depth': 2, 'criterion': 'log_loss'}
```

In [22]:
```
random.best_score_
```

Out[22]:
```
0.9
```