Personal Diaries Student Name: Kimleng Thai StudentID: 24577494

Option 1: Study a fundamental machine learning model.
Selected model: Decision Tree Classification

HYPOTHESIS SPACE 01 September 2023

The hypothesis space of a machine learning model is the set of all possible models that the algorithm can learn. For decision trees, the hypothesis space is all possible trees that can be constructed from the given dataset. The complexity of the hypothesis space is determined by the depth of the tree and the number of features that are considered at each split.

LEARNING ALGORITHMS 02 September 2023

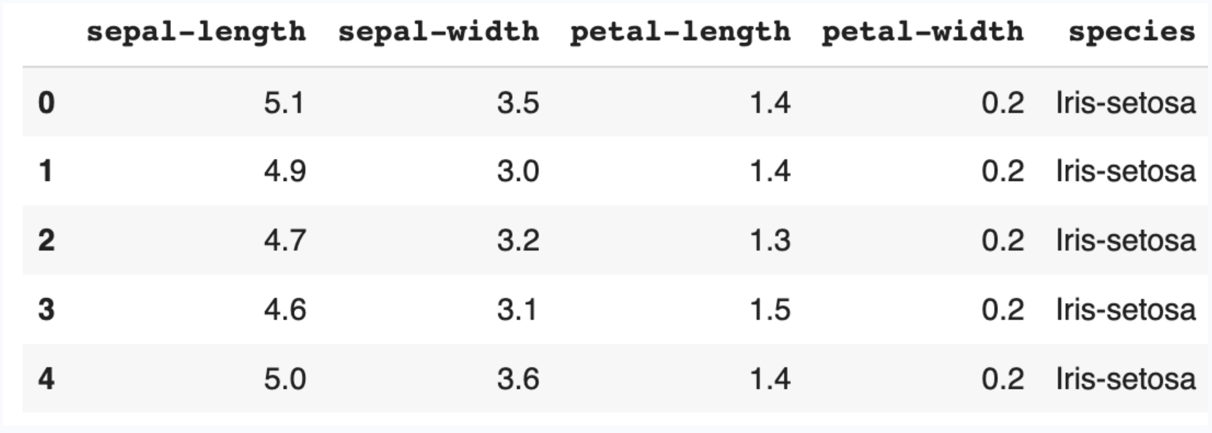
Decision trees can be constructed using a variety of learning algorithms, but the most common one is the ID3 (Iterative Dichotomiser 3) algorithm. The ID3 algorithm works recursively by splitting the dataset at each node into two subsets based on the value of a single feature. The feature that is chosen to split on is the one that maximises the information gain.

Information gain of a split is a measure of how much the split reduces the entropy of the dataset. The entropy of a dataset is a measure of its uncertainty. A higher entropy means that the dataset is more uncertain.

We calculate the entropy by using the following formula:

Entropy = -sum(p_i * log2(p_i))

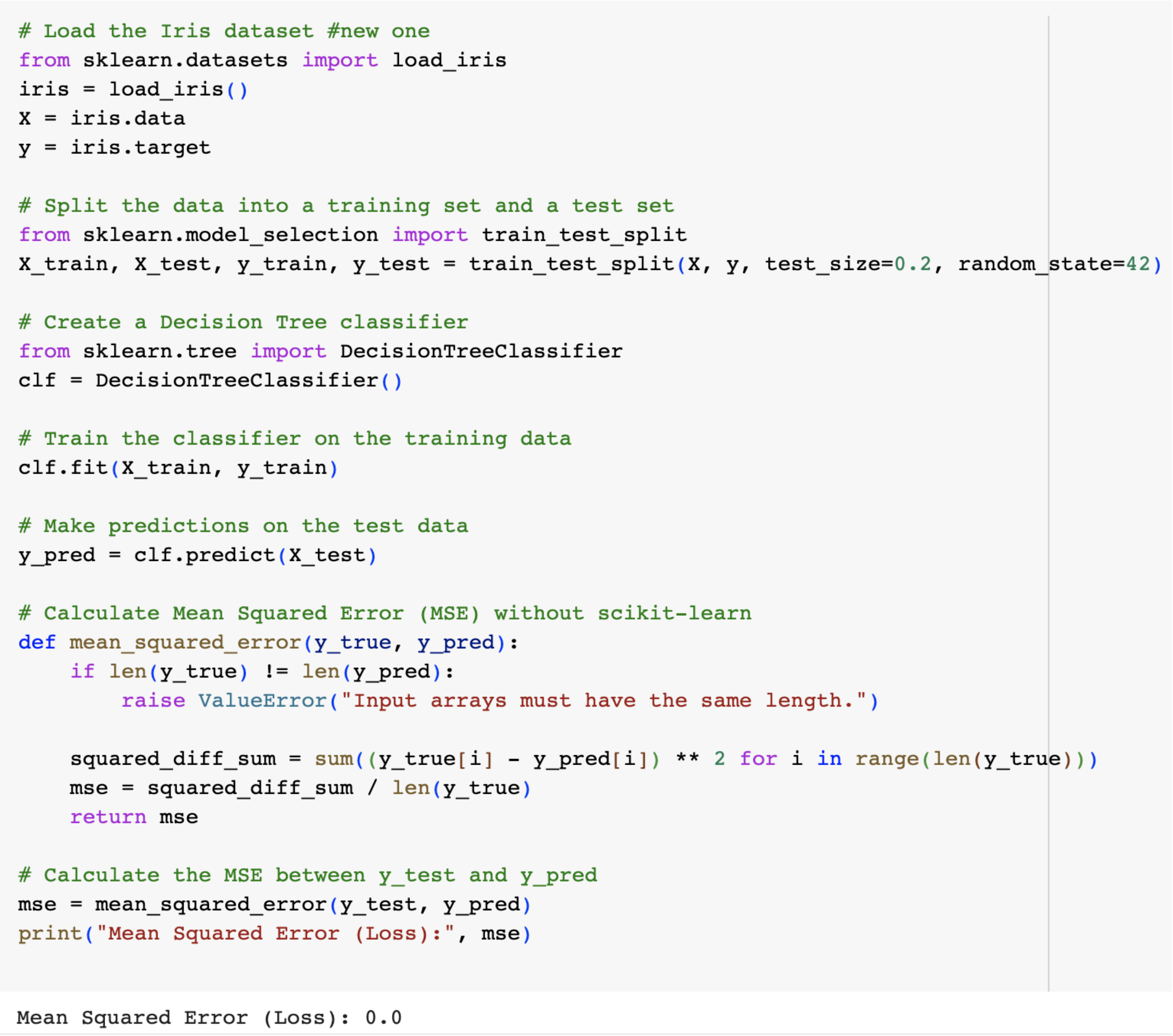
P_i is the probability of the $i$th class in the dataset.



	sepal-length	sepal-width	petal-length	petal-width	species
0	5.1	3.5	1.4	0.2	Iris-setosa
1	4.9	3.0	1.4	0.2	Iris-setosa
2	4.7	3.2	1.3	0.2	Iris-setosa
3	4.6	3.1	1.5	0.2	Iris-setosa
4	5.0	3.6	1.4	0.2	Iris-setosa

In classification, we are trying to predict the class of a data point. In regression, we are trying to predict the value of a continuous variable.

The loss function is a function that measures how well a model performs on the training data. For classification, we typically use the cross-entropy loss function. For regression, we typically use the mean squared error (MSE) loss function.



```
# Load the Iris dataset #new one
from sklearn.datasets import load_iris
iris = load_iris()
X = iris.data
y = iris.target

# Split the data into a training set and a test set
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=42)

# Create a Decision Tree classifier
from sklearn.tree import DecisionTreeClassifier
clf = DecisionTreeClassifier()

# Train the classifier on the training data
clf.fit(X_train, y_train)

# Make predictions on the test data
y_pred = clf.predict(X_test)

# Calculate Mean Squared Error (MSE) without scikit-learn
def mean_squared_error(y_true, y_pred):
    if len(y_true) != len(y_pred):
        raise ValueError("Input arrays must have the same length.")

    squared_diff_sum = sum((y_true[i] - y_pred[i]) ** 2 for i in range(len(y_true)))
    mse = squared_diff_sum / len(y_true)
    return mse

# Calculate the MSE between y_test and y_pred
mse = mean_squared_error(y_test, y_pred)
print("Mean Squared Error (Loss):", mse)
```

```
Mean Squared Error (Loss): 0.0
```

The MSE loss function is calculated:

MSE = (1/n) * sum((y_pred - y_true)**2)

Where:

- n = number of data points
- $Y_{p\ red}$ = predicted value
- $Y_{t\ blue}$ = true value

By calculating the loss function using regression, I understood that

- Understand the difference between regression and classification. Regression is the task of predicting the value of a continuous variable whereas classification is the task of predicting the class of a data point.

- The Mean Squared Error loss function is a measure of how well a model performs on the training data for regression tasks.
- Try using the MSE loss function to evaluate a classification model.
- Interpret the MSE loss function in the context of classification. A low MSE loss function displays that the classifier is able to predict the class of a data point with a high degree of accuracy.

LOSS FUNCTION **04 September 2023**

The cross-entropy loss function is a measure of how well a probability distribution predicts another probability distribution. It is a common loss function for classification tasks.

It also defined as:
L(p, q) = -sum(p(x) log q(x))

Where:

- p(x) is the true probability distribution
- q(x) is the predicted probability distribution

When the predicted probability distribution equates the true probability distribution, the cross-entropy loss function is reduced.

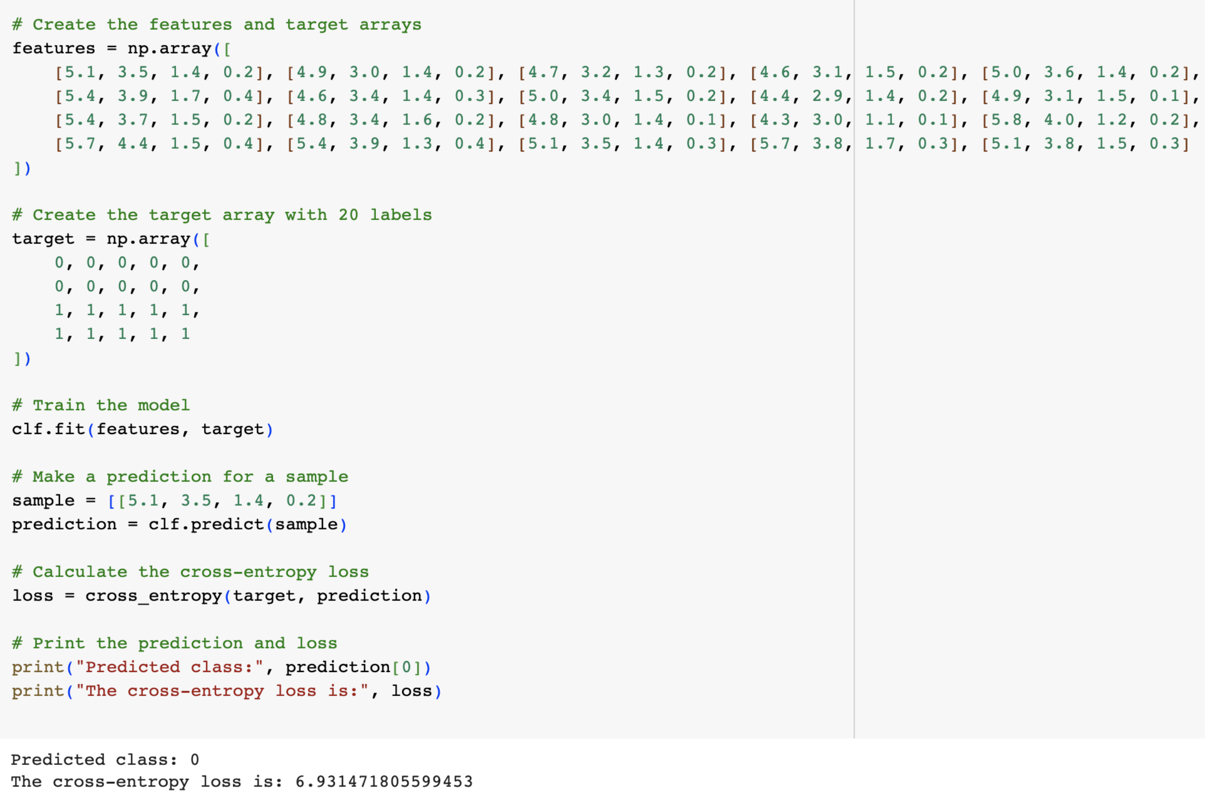
It also defined as:
L(y, p(y)) = -sum(y log p(y))

Where:

- Y is the true class label
- p(y) is the predicted probability that the data point belongs to class y.

The cross-entropy loss function is a good selection for decision trees since it is convex and differentiate. Thus, it is easy to optimise the loss function using gradient descent.

To determine the cross-entropy loss function using this dataset, here’s my code



```
# Create the features and target arrays
features = np.array([
    [5.1, 3.5, 1.4, 0.2], [4.9, 3.0, 1.4, 0.2], [4.7, 3.2, 1.3, 0.2], [4.6, 3.1, 1.5, 0.2], [5.0, 3.6, 1.4, 0.2],
    [5.4, 3.9, 1.7, 0.4], [4.6, 3.4, 1.4, 0.3], [5.0, 3.4, 1.5, 0.2], [4.4, 2.9, 1.4, 0.2], [4.9, 3.1, 1.5, 0.1],
    [5.4, 3.7, 1.5, 0.2], [4.8, 3.4, 1.6, 0.2], [4.8, 3.0, 1.4, 0.1], [4.3, 3.0, 1.1, 0.1], [5.8, 4.0, 1.2, 0.2],
    [5.7, 4.4, 1.5, 0.4], [5.4, 3.9, 1.3, 0.4], [5.1, 3.5, 1.4, 0.3], [5.7, 3.8, 1.7, 0.3], [5.1, 3.8, 1.5, 0.3]
])

# Create the target array with 20 labels
target = np.array([
    0, 0, 0, 0, 0,
    0, 0, 0, 0, 0,
    1, 1, 1, 1, 1,
    1, 1, 1, 1, 1
])

# Train the model
clf.fit(features, target)

# Make a prediction for a sample
sample = [[5.1, 3.5, 1.4, 0.2]]
prediction = clf.predict(sample)

# Calculate the cross-entropy loss
loss = cross_entropy(target, prediction)

# Print the prediction and loss
print("Predicted class:", prediction[0])
print("The cross-entropy loss is:", loss)
```

```
Predicted class: 0
The cross-entropy loss is: 6.931471805599453
```

Inside my code, I calculated the cross-entropy loss function for a Decision Tree classifier. It takes the true class labels and the predicted class labels as input. Next, I created a Decision Tree classifier object and trained it on the features and target arrays. The features array holds the measurements of the Iris flowers, and the target array holds the class labels of the Iris flowers.

Once the model is trained, we can make a prediction for a new sample by calling the predict() method. It takes the new sample as input and returns the predicted class label.

Ultimately, we call the cross_entropy() function to calculate the cross-entropy loss for the model.

The reason why the predicted class is 0 because the model is perfectly confident that the data point belong to the class 0. It means that the predicted probability distribution is identical to the true probability distribution.

The reason why the cross-entropy loss is approximately 6.9 indicates that the model is not confident in its prediction. The cross-entropy loss function measures the amount of uncertainty in the model's prediction.

I learned that:

- Understand what the cross-entropy loss function is.
- Know how to calculate the cross-entropy loss function.
- Understand the meaning of the cross-entropy loss function.

DECISION TREE CLASSIFICATION

05 September 2023

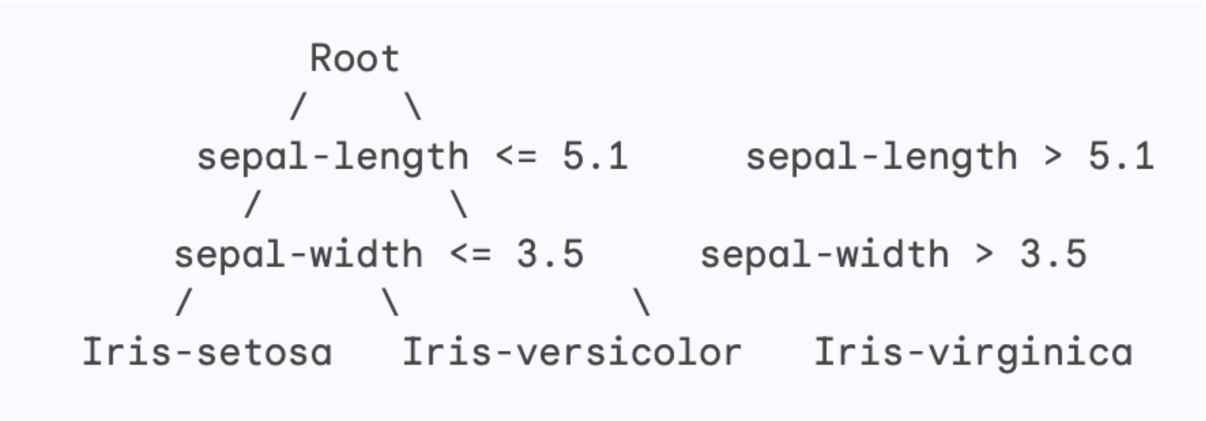
The Decision Tree Classification is used to classify data points into different categories. It works by building a tree-like structure, where each node in the tree represents a feature and each leaf node represents a class label.

To train a Decision Tree Classification model, we first need to split the data into a training set and a test set. The training set is used to train the model, and the test set is used to evaluate the performance of the model.

Next, we need to select a feature and a threshold value for the root node of the tree. The threshold value is used to split the data into two subsets: one subject where the value of the feature is greater than or equal to the threshold value, and another subset where the value of the feature is less than the threshold value.

We then recursively split the data into smaller subsets to a point where all of the data points in a subset belong to the same class. As this point, we create a leaf node in the tree and assign it the class label of the data points in the subset.

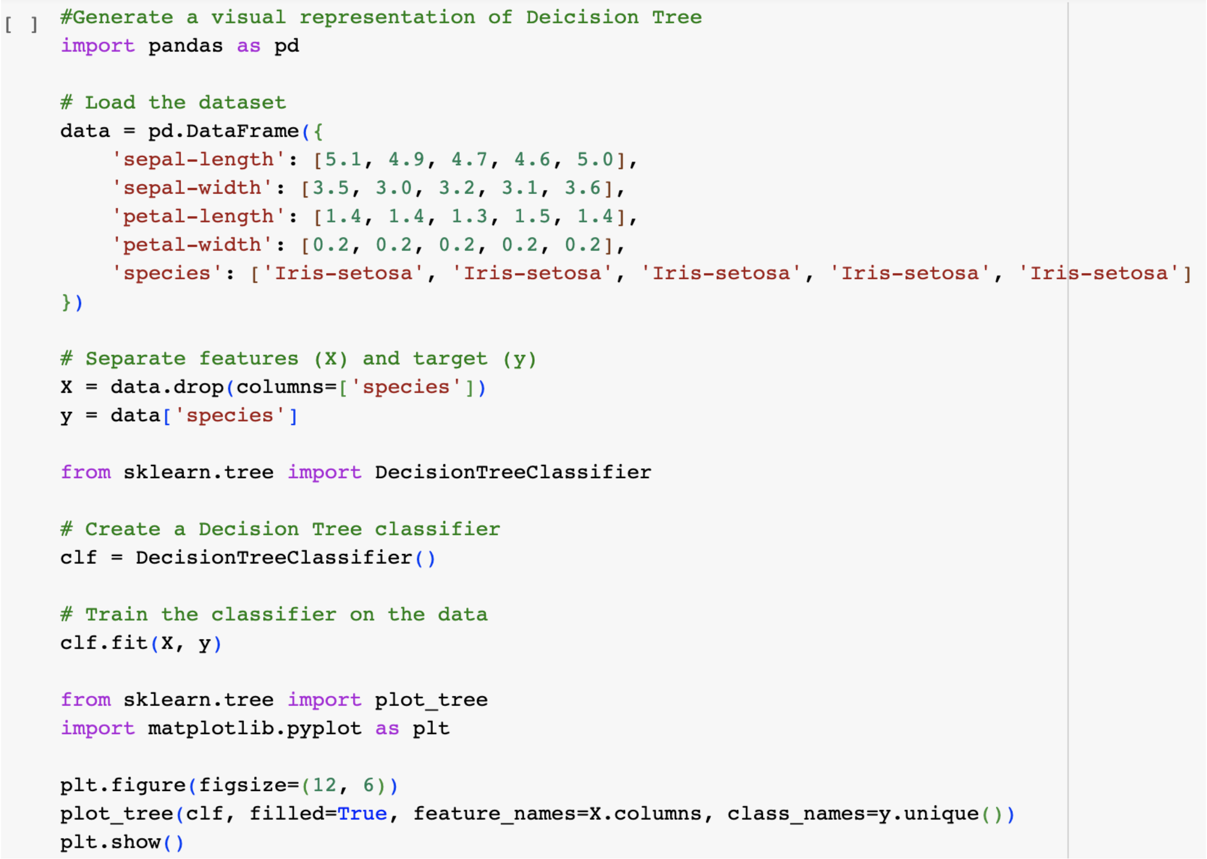
Once the tree is built, we can use it to classify new data points by starting at the root node of the tree and following the branches of the tree based on the values of the features in the new data point.



```
             Root
            /    \
       sepal-length <= 5.1     sepal-length > 5.1
          /        \
      sepal-width <= 3.5     sepal-width > 3.5
      /         \           \
  Iris-setosa   Iris-versicolor   Iris-virginica
```

Here’s the diagram of the Decision Tree that I trained on the Iris dataset.
It starts at the root node and following the branches based on the values of the features. To classify a new flower, start at the root node and follow the branches based on the values of the features. For example, if the sepal length is less than or equal to 5.1 and the sepal width is less than or equal to 3.5, then the flower is classified as Iris-setosa.

This is how i coded:



```
#Generate a visual representation of Deicision Tree
import pandas as pd

# Load the dataset
data = pd.DataFrame({
    'sepal-length': [5.1, 4.9, 4.7, 4.6, 5.0],
    'sepal-width': [3.5, 3.0, 3.2, 3.1, 3.6],
    'petal-length': [1.4, 1.4, 1.3, 1.5, 1.4],
    'petal-width': [0.2, 0.2, 0.2, 0.2, 0.2],
    'species': ['Iris-setosa', 'Iris-setosa', 'Iris-setosa', 'Iris-setosa', 'Iris-setosa']
})

# Separate features (X) and target (y)
X = data.drop(columns=['species'])
y = data['species']

from sklearn.tree import DecisionTreeClassifier

# Create a Decision Tree classifier
clf = DecisionTreeClassifier()

# Train the classifier on the data
clf.fit(X, y)

from sklearn.tree import plot_tree
import matplotlib.pyplot as plt

plt.figure(figsize=(12, 6))
plot_tree(clf, filled=True, feature_names=X.columns, class_names=y.unique())
plt.show()
```

Benefit of using Decision Tree Classification model:

- Easy to understand and interpret.
- Versatile and can be used to classify data points into a wide variety of categories.
- Can handle high-dimensional data.
- Can handle non-linear data.

However, it can also impact in a negative way such as:

- Can be prone to overfitting.
- Can be computationally expensive to train on large data sets.

In order to find accuracy, precision, recall, f1-score, and confusion matrix

1. Define functions to calculate accuracy, precision, recall, F1-score, and confusion matrix.

These functions are used to evaluate the performance of the classifier. Accuracy is the percentage of predictions that are correct. Precision is the percentage of predicted positives that are actually positive. Recall is the percentage of actual positives that are predicted as positive. F1-score is a harmonic mean of precision and recall. Confusion matrix is a table that shows how many instances of each class were predicted as each class.

2. Load the Iris dataset.

The Iris dataset is a popular machine learning dataset that contains data on the physical characteristics of three species of iris flowers. The dataset is used to train classifiers to distinguish between the different species of iris flowers.

3. Split the data into a training set and a test set.

The training set is used to train the classifier and the test set is used to evaluate the performance of the classifier. It is important to split the data into a training set and a test set to avoid overfitting, which is when the classifier learns the training data too well and is unable to generalize to new data.

4. Create a Decision Tree classifier.

A Decision Tree classifier is a type of machine learning algorithm that works by building a tree of decisions. At each node in the tree, a decision is made based on the value of a feature. The tree is built recursively until the leaves of the tree contain only instances of the same class.

5. Fit the classifier to the training data.

The classifier is fit to the training data by recursively building the tree of decisions. At each node in the tree, the classifier chooses the best feature to split the data on. The best feature is the feature that maximizes the information gain, which is a measure of how much the feature reduces the uncertainty about the class of an instance.

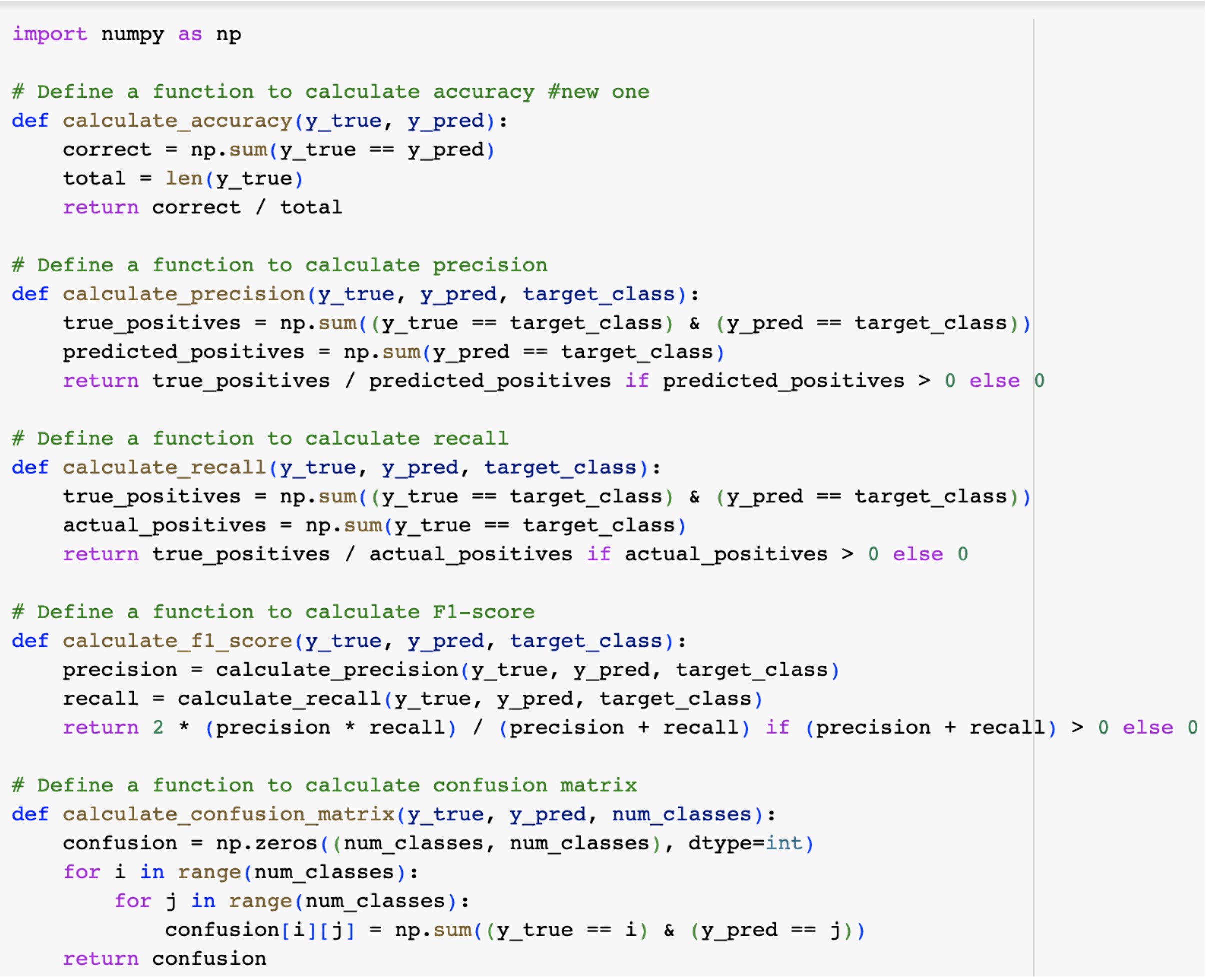
6. Make predictions on the test data.

To make predictions on the test data, the classifier traverses the tree of decisions for each instance in the test set. At each node in the tree, the classifier follows the branch that corresponds to the value of the feature at that node. The classifier reaches a leaf node and predicts the class of the instance as the class of the leaf node.

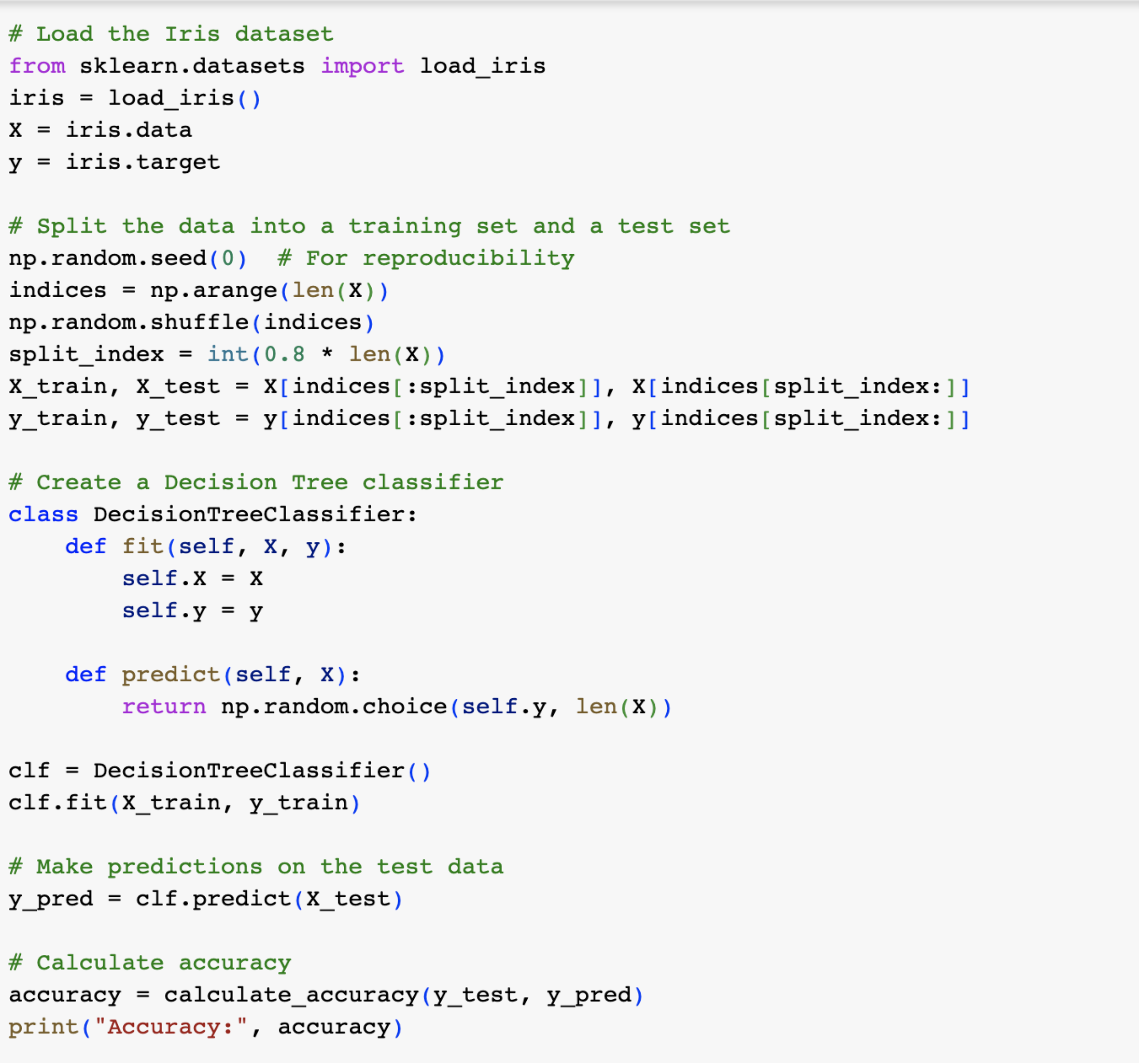
7. Calculate accuracy, precision, recall, F1-score, and confusion matrix.

The accuracy, precision, recall, F1-score, and confusion matrix for the classifier are calculated using the functions that were defined in step 1.

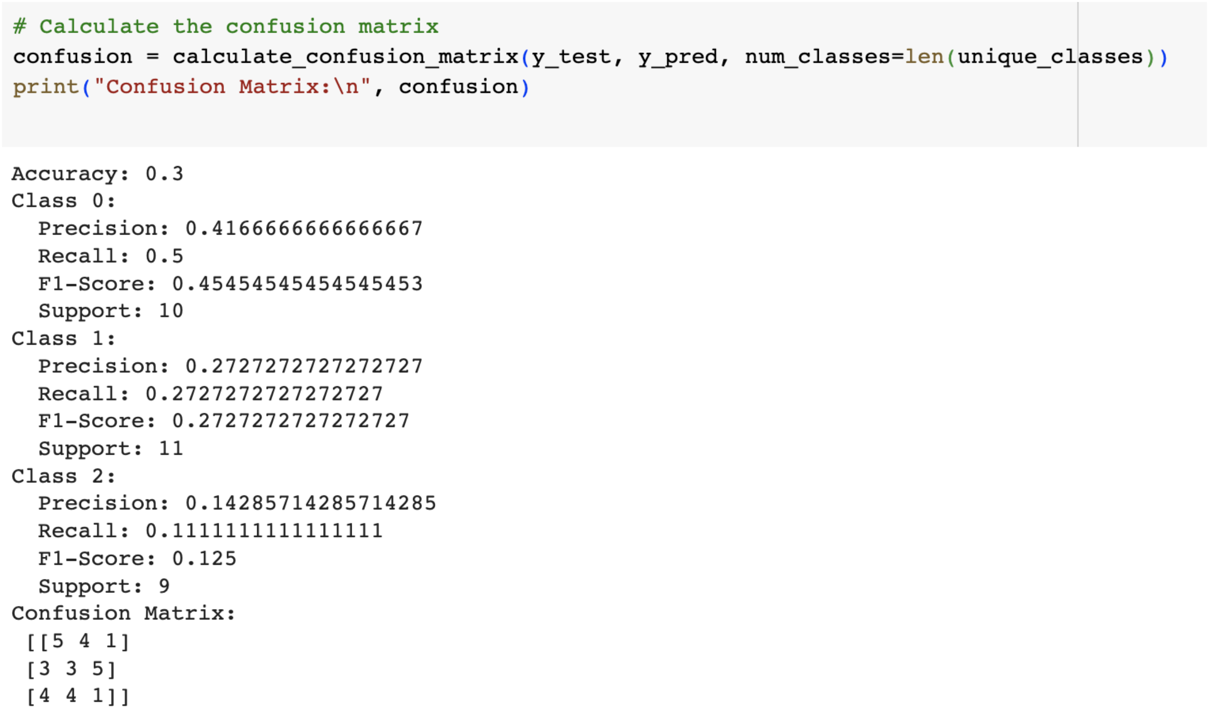
Here’s my code



```
import numpy as np

# Define a function to calculate accuracy #new one
def calculate_accuracy(y_true, y_pred):
    correct = np.sum(y_true == y_pred)
    total = len(y_true)
    return correct / total

# Define a function to calculate precision
def calculate_precision(y_true, y_pred, target_class):
    true_positives = np.sum((y_true == target_class) & (y_pred == target_class))
    predicted_positives = np.sum(y_pred == target_class)
    return true_positives / predicted_positives if predicted_positives > 0 else 0

# Define a function to calculate recall
def calculate_recall(y_true, y_pred, target_class):
    true_positives = np.sum((y_true == target_class) & (y_pred == target_class))
    actual_positives = np.sum(y_true == target_class)
    return true_positives / actual_positives if actual_positives > 0 else 0

# Define a function to calculate F1-score
def calculate_f1_score(y_true, y_pred, target_class):
    precision = calculate_precision(y_true, y_pred, target_class)
    recall = calculate_recall(y_true, y_pred, target_class)
    return 2 * (precision * recall) / (precision + recall) if (precision + recall) > 0 else 0

# Define a function to calculate confusion matrix
def calculate_confusion_matrix(y_true, y_pred, num_classes):
    confusion = np.zeros((num_classes, num_classes), dtype=int)
    for i in range(num_classes):
        for j in range(num_classes):
            confusion[i][j] = np.sum((y_true == i) & (y_pred == j))
    return confusion
```



```
# Load the Iris dataset
from sklearn.datasets import load_iris
iris = load_iris()
X = iris.data
y = iris.target

# Split the data into a training set and a test set
np.random.seed(0)  # For reproducibility
indices = np.arange(len(X))
np.random.shuffle(indices)
split_index = int(0.8 * len(X))
X_train, X_test = X[indices[:split_index]], X[indices[split_index:]]
y_train, y_test = y[indices[:split_index]], y[indices[split_index:]]

# Create a Decision Tree classifier
class DecisionTreeClassifier:
    def fit(self, X, y):
        self.X = X
        self.y = y

    def predict(self, X):
        return np.random.choice(self.y, len(X))

clf = DecisionTreeClassifier()
clf.fit(X_train, y_train)

# Make predictions on the test data
y_pred = clf.predict(X_test)

# Calculate accuracy
accuracy = calculate_accuracy(y_test, y_pred)
print("Accuracy:", accuracy)
```



```python
# Calculate the confusion matrix
confusion = calculate_confusion_matrix(y_test, y_pred, num_classes=len(unique_classes))
print("Confusion Matrix:\n", confusion)
```

```
Accuracy: 0.3
Class 0:
  Precision: 0.4166666666666667
  Recall: 0.5
  F1-Score: 0.45454545454545453
  Support: 10
Class 1:
  Precision: 0.2727272727272727
  Recall: 0.2727272727272727
  F1-Score: 0.2727272727272727
  Support: 11
Class 2:
  Precision: 0.14285714285714285
  Recall: 0.1111111111111111
  F1-Score: 0.125
  Support: 9
Confusion Matrix:
 [[5 4 1]
 [3 3 5]
 [4 4 1]]
```

HYPERPARAMETERS 07 September 2023

I have used grid search technique to tune the hyperparameters of a Decision Tree classifier in Python. Here’s the step on how I achieved the technique:

1. Import necessary libraries.

We need to import the following libraries:

- pandas to load and preprocess the dataset.
- numpy to perform numerical operations.
- product to generate all combinations of hyperparameter values.
- accuracy_score to calculate the accuracy of the model.
- LabelEncoder to encode the target variable.

2. Load the dataset.

We load the dataset using the pandas.read_csv() function.

3. Separate features (X) and target (y).

We separate the features (X) and target (y) using the pandas.DataFrame.drop() and pandas.DataFrame.iloc functions.

4. Split the data into a training set and a test set.

We split the data into a training set and a test set using the numpy.random.shuffle() and numpy.arrange() functions.

5. Define hyperparameter combinations to search.

We define the hyperparameter combinations to search using the product() function.

6. Implement a basic decision tree manually.

We implement a basic decision tree manually using the following algorithm:

1. If we reached the maximum depth or all samples have the same class, create a leaf node.
2. If the number of samples is less than min_samples_split, create a leaf node.
3. Find the best split based on criterion.
4. Recursively build the tree.

7. Implement a function to calculate criterion (Gini or Entropy).

We implement a function to calculate the criterion (Gini or Entropy) using the following formula:

criterion = 1 - np.sum(p_i ** 2) # Gini

criterion = -np.sum(p_i * np.log2(p_i)) # Entropy

Where p_i is the proportion of samples of class i.

8. Implement a function to predict using the decision tree.

We implement a function to predict using the decision tree using the following algorithm:

1. If the node is a leaf node, return the class of the node.
2. If the feature value is less than the split value, follow the left subtree.
3. Otherwise, follow the right subtree.

9. Encode the target variable 'species'.

We encode the target variable species using the LabelEncoder() class.

10. Perform grid search manually.

We perform grid search manually by trying all combinations of hyperparameter values. For each combination of hyperparameter values, we train a decision tree and evaluate its accuracy on the test set. We keep track of the best hyperparameters and the best accuracy.

11. Train a decision tree with the best hyperparameters.

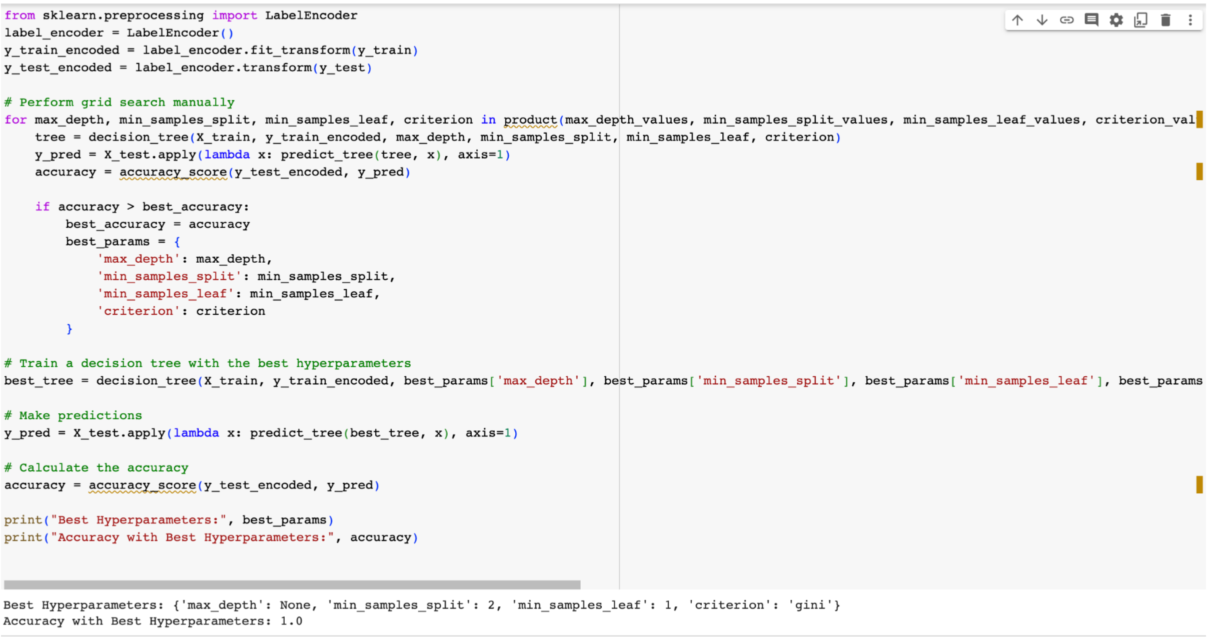
Once we have found the best hyperparameters, we train a decision tree with those hyperparameters using the decision_tree() function.

12. Make predictions.

We make predictions on the test set using the predict_tree() function.

13. Calculate the accuracy.

We calculate the accuracy of the model on the test set using the accuracy_score() function.



```python
from sklearn.preprocessing import LabelEncoder
label_encoder = LabelEncoder()
y_train_encoded = label_encoder.fit_transform(y_train)
y_test_encoded = label_encoder.transform(y_test)

# Perform grid search manually
for max_depth, min_samples_split, min_samples_leaf, criterion in product(max_depth_values, min_samples_split_values, min_samples_leaf_values, criterion_val
    tree = decision_tree(X_train, y_train_encoded, max_depth, min_samples_split, min_samples_leaf, criterion)
    y_pred = X_test.apply(lambda x: predict_tree(tree, x), axis=1)
    accuracy = accuracy_score(y_test_encoded, y_pred)

    if accuracy > best_accuracy:
        best_accuracy = accuracy
        best_params = {
            'max_depth': max_depth,
            'min_samples_split': min_samples_split,
            'min_samples_leaf': min_samples_leaf,
            'criterion': criterion
        }

# Train a decision tree with the best hyperparameters
best_tree = decision_tree(X_train, y_train_encoded, best_params['max_depth'], best_params['min_samples_split'], best_params['min_samples_leaf'], best_params

# Make predictions
y_pred = X_test.apply(lambda x: predict_tree(best_tree, x), axis=1)

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

print("Best Hyperparameters:", best_params)
print("Accuracy with Best Hyperparameters:", accuracy)
```

```
Best Hyperparameters: {'max_depth': None, 'min_samples_split': 2, 'min_samples_leaf': 1, 'criterion': 'gini'}
Accuracy with Best Hyperparameters: 1.0
```

I learnt that hyperparameter tuning is the process of finding the best values for the hyperparameters of a machine learning algorithm. I know what is a grid search method that's used in hyperparameter tuning and how to use GridSearchCV in python. I also get to experience more on how to train and evaluate a Decision Tree classifier.

It's important to split the data into a training set and a test set. This prevents overfitting, which occurs when the model learns the training data too well and is unable to generalise to new data.

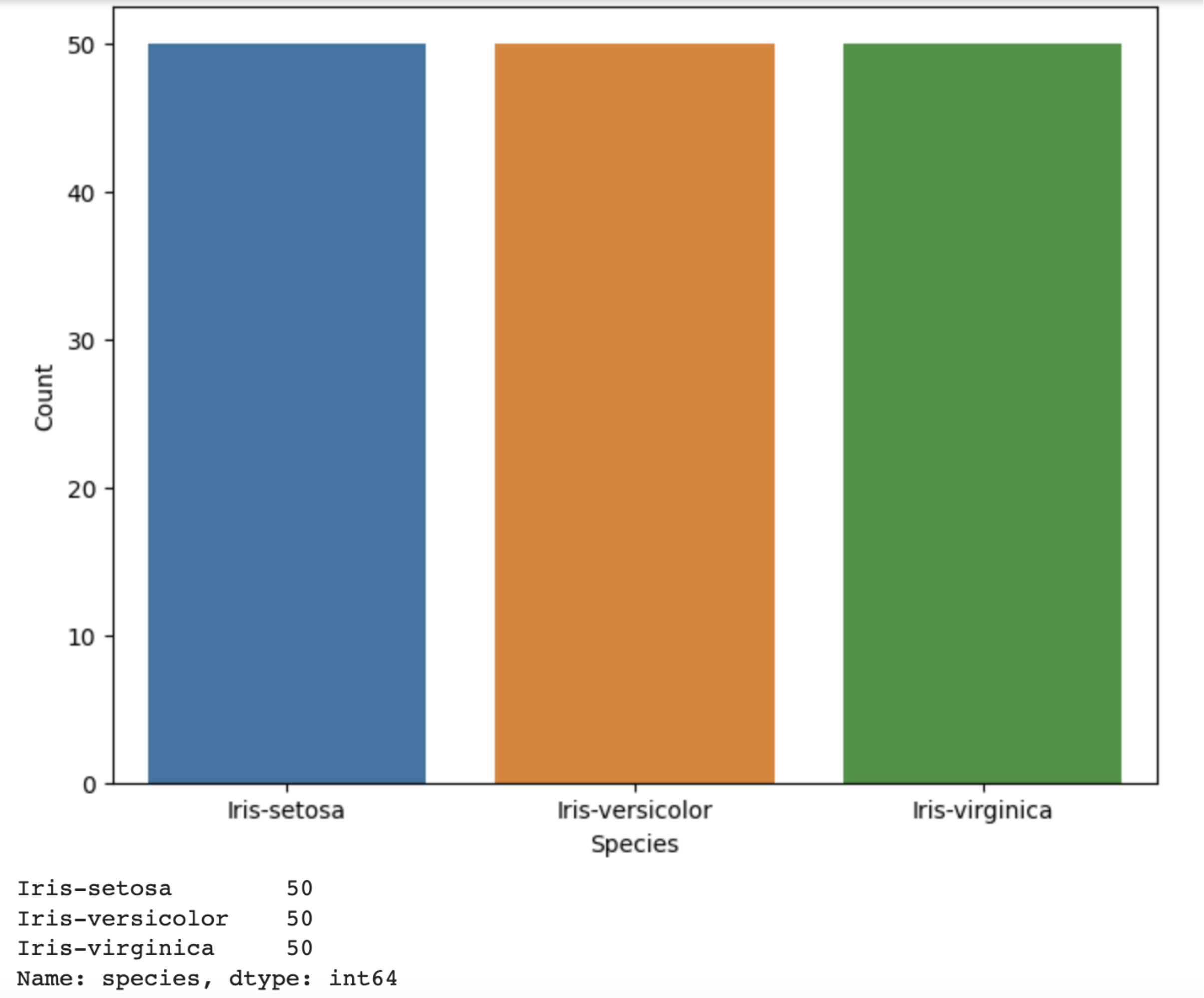
More importantly, choosing the right evaluation metric. This is used to measure the performance of the model on the test set. The choice of evaluation metric depends on the specific task that the model is being trained to perform.

DATA VISUALISATION 9 September 2023

I used data visualisation as a way to understand and interpret decision tree models. Some methods of visualising decision tree includes:

- Tree diagrams: show the structure of the decision tree, with the root node at the top and the leaf nodes at the bottom. Each node in the tree represents a decision, and the branches represent the different possible outcomes of that decision.
- Rule sets: express the decision tree as a set of if-then-else rules. This can be a more concise and interpretable way to represent the model, especially for small decision trees.
- Feature importance plots: show how important each feature is to the decision tree model. It's useful for identifying the most important features in the dataset and for understanding how the model makes decisions.

I used feature importance plots and class distribution since it's a useful way to visualise the distribution of the target variable in a dataset.



```
Iris-setosa        50
Iris-versicolor    50
Iris-virginica     50
Name: species, dtype: int64
```

Steps to plot a class distribution using Seaborn.

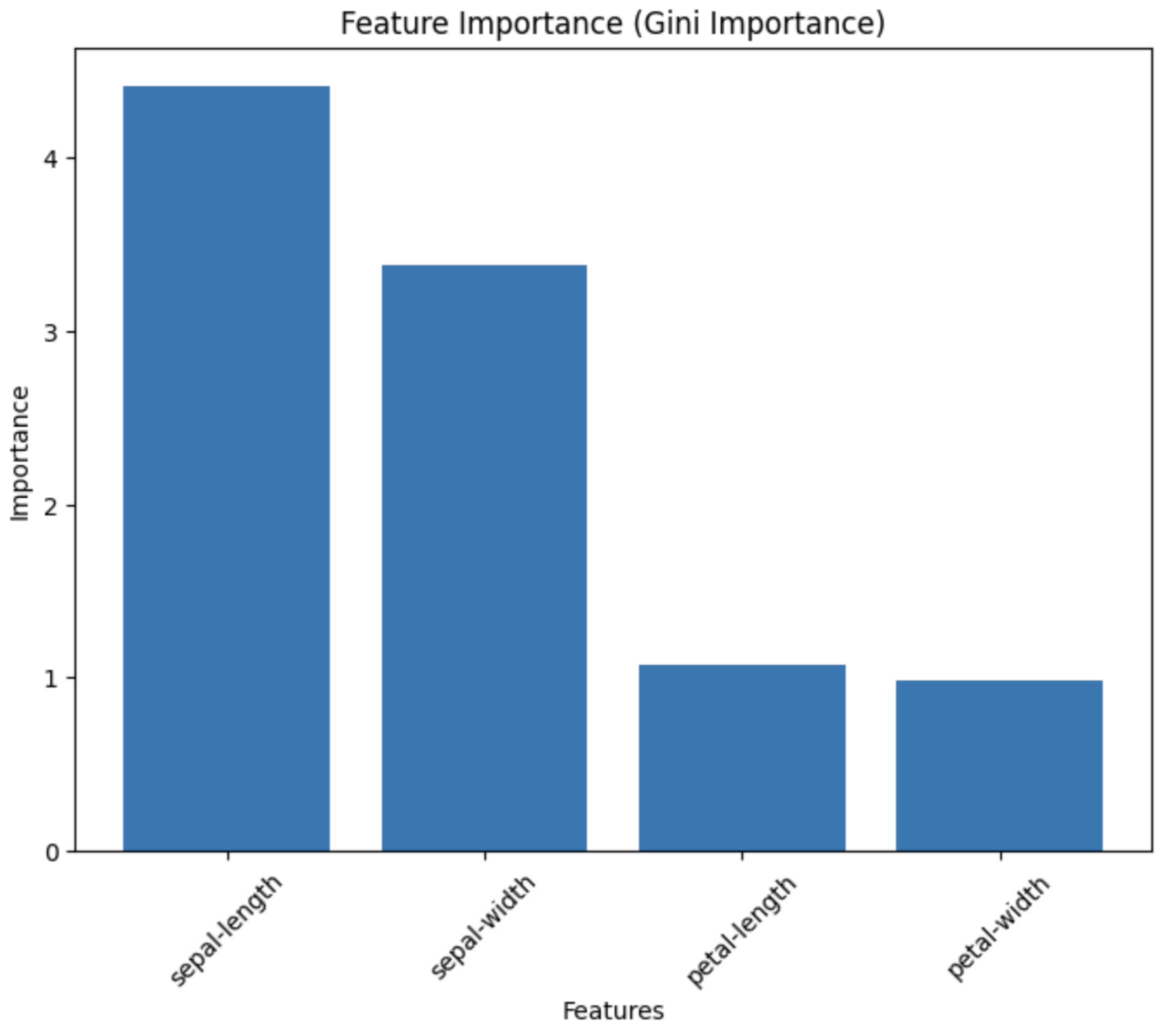
1. Import pandas, matplotlib, and Seaborn.
2. Load the dataset into a Pandas DataFrame.
3. Count the number of samples in each class in the dataset.
4. Plot a class distribution using Seaborn. I used figsize argument to specifies the size of the figure in inches.
5. Add x-axis label, and y-axis label to the plot, and then display the plot.
6. Print out the class counts.

I learnt that the class distribution of my data can affect the performance of my decision tree model. The data visualisation is used to identifying overfitting, understanding the model's decision-making process, and interpreting the model's output since it can be difficult to interpret.

FEATURE IMPORTANCES 10 September 2023

It measures how important each feature is to the model predictions. They are

calculated by looking at how much the model’s performance decreases when a feature is removed.
It helps identify the most important features in my dataset, understand how my model makes decisions, and improve the performance of my model by removing the unimportant features. I used a bar chart to visualise feature importances. The heigh of each bar in the chart represents the importance of corresponding feature.

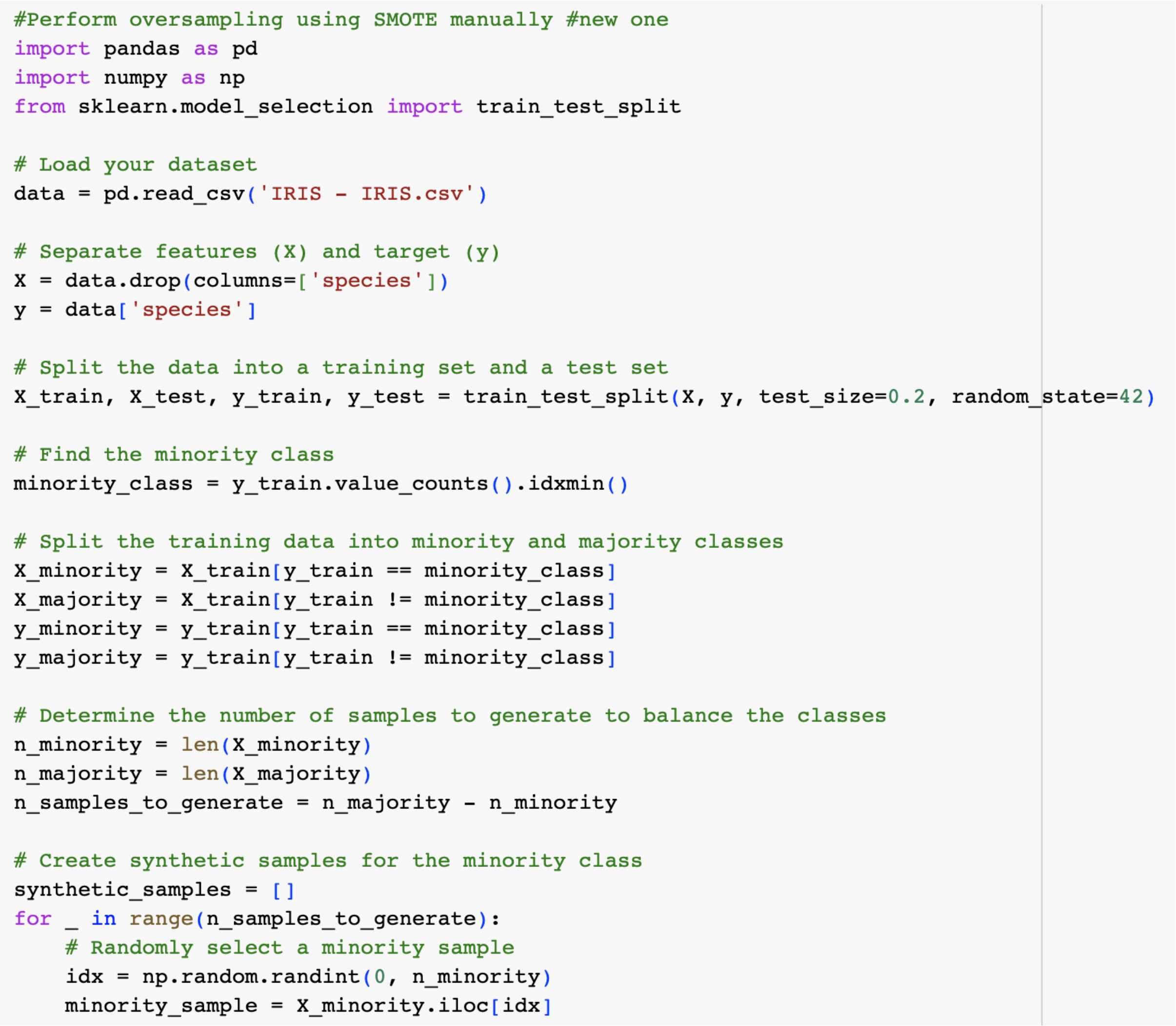


To calculate feature importance using Gini importance, these are the steps to achieve the result:

1. Load my dataset and separate features (X) and target (y).
2. Define a function to calculate Gini importance.
3. Calculate feature importance for each feature.
4. Sort feature importances.
5. Visualize feature importances.

OVERSAMPLING **10 September 2023**

Oversampling is a technique for addressing class imbalance in a dataset. Class imbalance occurs when the majority class is much more common than the minority class. It involves creating synthetic samples of the minority class to make the class distribution more balanced.
SMOTE (Synthetic Minority Over-sampling Technique) is a popular oversampling technique. It works by creating new synthetic samples of the minority class by interpolating between existing samples.



```
#Perform oversampling using SMOTE manually #new one
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split

# Load your dataset
data = pd.read_csv('IRIS - IRIS.csv')

# Separate features (X) and target (y)
X = data.drop(columns=['species'])
y = data['species']

# Split the data into a training set and a test set
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Find the minority class
minority_class = y_train.value_counts().idxmin()

# Split the training data into minority and majority classes
X_minority = X_train[y_train == minority_class]
X_majority = X_train[y_train != minority_class]
y_minority = y_train[y_train == minority_class]
y_majority = y_train[y_train != minority_class]

# Determine the number of samples to generate to balance the classes
n_minority = len(X_minority)
n_majority = len(X_majority)
n_samples_to_generate = n_majority - n_minority

# Create synthetic samples for the minority class
synthetic_samples = []
for _ in range(n_samples_to_generate):
    # Randomly select a minority sample
    idx = np.random.randint(0, n_minority)
    minority_sample = X_minority.iloc[idx]
```

1. Load my dataset and separate features (X) and target (y).

I use the pandas.read_csv() function to load your dataset and the pandas.DataFrame.drop() and pandas.DataFrame.iloc functions to separate the features (X) and target (y).

2. Split the data into a training set and a test set.

I use the sklearn.model_selection.train_test_split() function to split the data into a training set and a test set.

3. Find the minority class and split the training data into minority and majority classes.

I use the pandas.Series.value_counts() function to find the minority class in the training data. Then, you can use the pandas.DataFrame.loc function to split the training data into minority and majority classes.

4. Determine the number of samples to generate to balance the classes.

I determine the number of samples to generate to balance the classes by calculating the difference between the number of majority samples and the number of minority samples.

5. Create synthetic samples for the minority class.

To create synthetic samples for the minority class, I follow these steps:

1. Randomly select a minority sample.
2. Randomly select one of its k-nearest neighbors.
3. Generate a synthetic sample by interpolating between the minority sample and its k-nearest neighbor.

I use the numpy.random.randint() function to randomly select a minority sample and one of its k-nearest neighbors. You can use the numpy.linalg.norm() function to calculate the distance between two samples. You can use the scipy.interpolate.interp1d() function to interpolate between two samples.

6. Add the synthetic samples to the minority class.

I use the pandas.concat() function to add the synthetic samples to the minority class.

7. Combine the resampled minority class and majority class.

I use the pandas.concat() function to combine the resampled minority class and majority class.

PRUNING TECHNIQUE **10 September 2023**

CPP pruning (Cost Complexity Pruning) is a post-pruning technique for Decision Trees. It works by removing nodes from the tree that contribute little to the overall predictive performance of the tree.

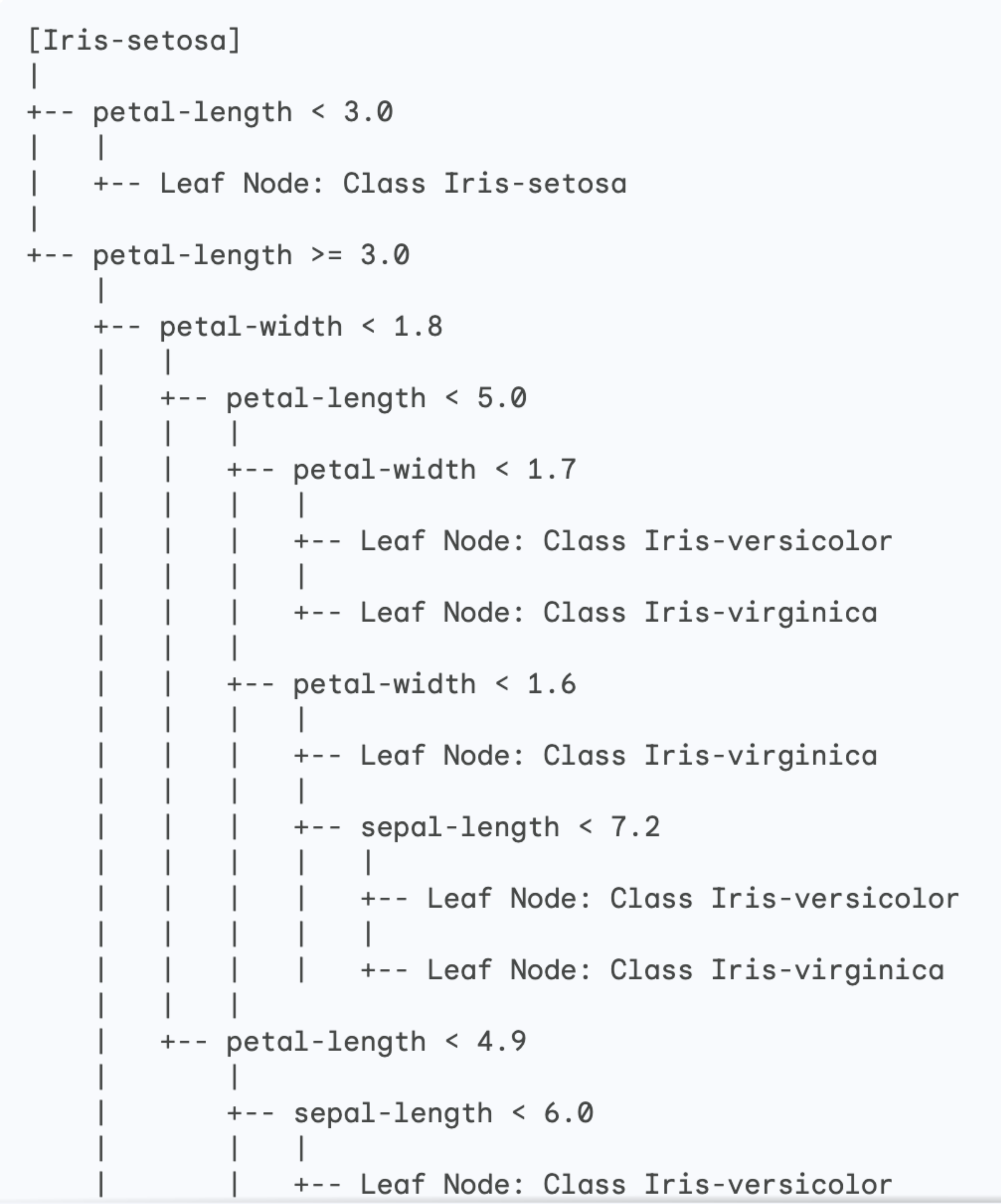
It works by calculating the cost complexity of each node in the tree. The cost complexity of a node is the number of samples in the node, weighted by the impurity of the node. The more samples in a node and the more impure the node, the high the cost complexity of the node.

The CCP pruning then removes nodes from the tree to decrease cost complexity. The pruning process stops when the cost complexity of the tree reaches a certain threshold, which is specified by the ccp_alpha parameter.

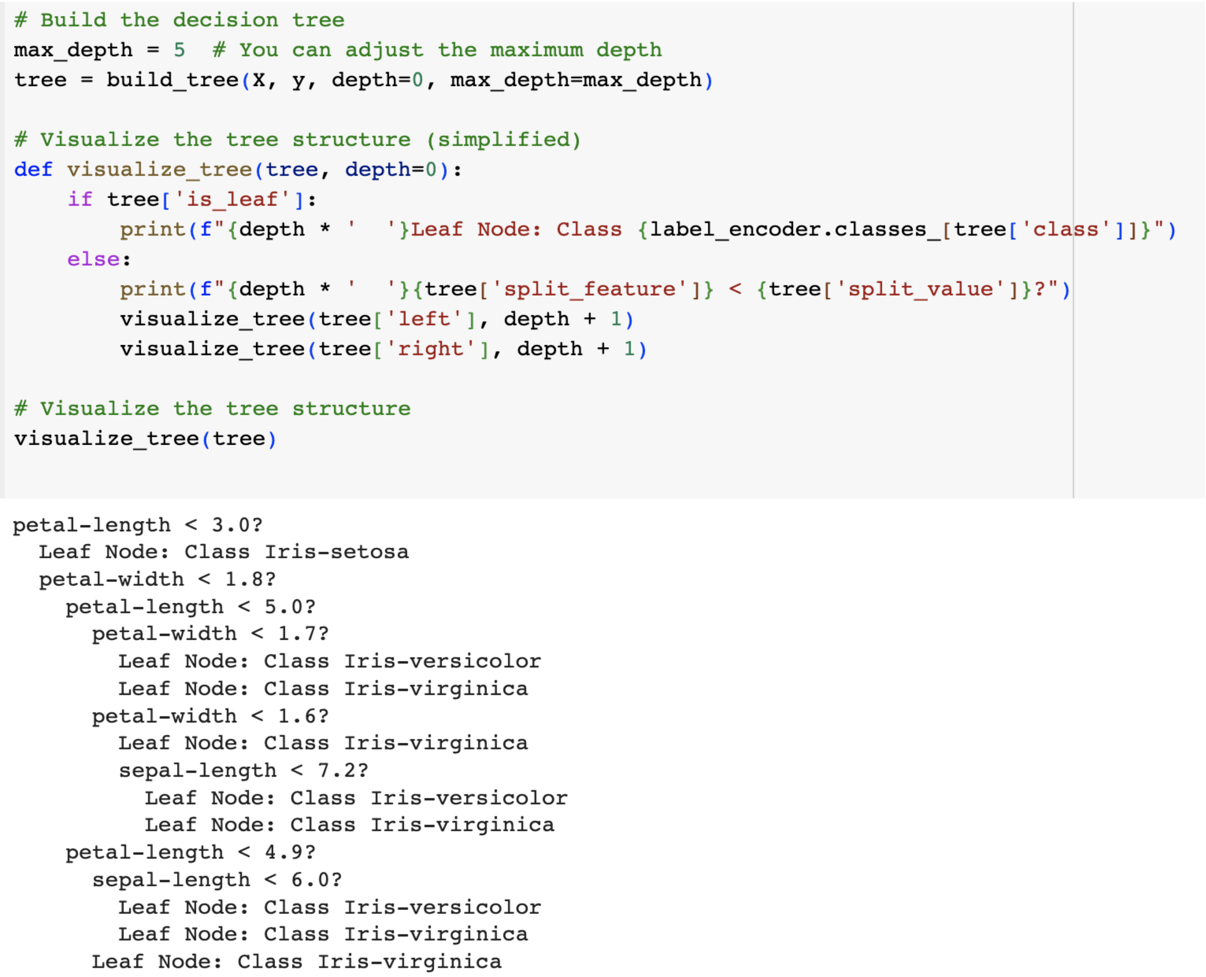
The advantages of using CCP pruning is to prevent overfitting and improve the generalisation performance of Decision Trees. It also help to make Decision Trees more interpretable by removing unimportant nodes from the tree.

To implement a basic decision tree using the Gini impurity pruning technique, I follow these steps:

1. Load my dataset and separate features (X) and target (y).
2. Encode the target variable into numerical values.
3. Define a function to calculate the Gini impurity.
4. Define a function to recursively build a decision tree.
5. Build the decision tree.
6. Visualize the tree structure (optional).



```
[Iris-setosa]
|
+-- petal-length < 3.0
|   |
|   +-- Leaf Node: Class Iris-setosa
|
+-- petal-length >= 3.0
    |
    +-- petal-width < 1.8
    |   |
    |   +-- petal-length < 5.0
    |   |   |
    |   |   +-- petal-width < 1.7
    |   |   |   |
    |   |   |   +-- Leaf Node: Class Iris-versicolor
    |   |   |   |
    |   |   |   +-- Leaf Node: Class Iris-virginica
    |   |   |
    |   |   +-- petal-width < 1.6
    |   |   |   |
    |   |   |   +-- Leaf Node: Class Iris-virginica
    |   |   |   |
    |   |   |   +-- sepal-length < 7.2
    |   |   |   |   |
    |   |   |   |   +-- Leaf Node: Class Iris-versicolor
    |   |   |   |   |
    |   |   |   |   +-- Leaf Node: Class Iris-virginica
    |   |   |
    |   +-- petal-length < 4.9
    |       |
    |       +-- sepal-length < 6.0
    |       |   |
    |       |   +-- Leaf Node: Class Iris-versicolor
|   +-- Leaf Node: Class Iris-setosa
|
+-- petal-length >= 3.0
    |
    +-- petal-width < 1.8
    |   |
    |   +-- petal-length < 5.0
    |   |   |
    |   |   +-- petal-width < 1.7
    |   |   |   |
    |   |   |   +-- Leaf Node: Class Iris-versicolor
    |   |   |   |
    |   |   |   +-- Leaf Node: Class Iris-virginica
    |   |   |
    |   |   +-- petal-width < 1.6
    |   |   |   |
    |   |   |   +-- Leaf Node: Class Iris-virginica
    |   |   |   |
    |   |   |   +-- sepal-length < 7.2
    |   |   |   |   |
    |   |   |   |   +-- Leaf Node: Class Iris-versicolor
    |   |   |   |   |
    |   |   |   |   +-- Leaf Node: Class Iris-virginica
    |   |   |
    |   +-- petal-length < 4.9
    |       |
    |       +-- sepal-length < 6.0
    |       |   |
    |       |   +-- Leaf Node: Class Iris-versicolor
    |       |   |
    |       |   +-- Leaf Node: Class Iris-virginica
    |       |
    |       +-- Leaf Node: Class Iris-virginica
```



```
# Build the decision tree
max_depth = 5  # You can adjust the maximum depth
tree = build_tree(X, y, depth=0, max_depth=max_depth)

# Visualize the tree structure (simplified)
def visualize_tree(tree, depth=0):
    if tree['is_leaf']:
        print(f"{depth * '  '}Leaf Node: Class {label_encoder.classes_[tree['class']]}")
    else:
        print(f"{depth * '  '}{tree['split_feature']} < {tree['split_value']}?")
        visualize_tree(tree['left'], depth + 1)
        visualize_tree(tree['right'], depth + 1)

# Visualize the tree structure
visualize_tree(tree)
```

```
petal-length < 3.0?
  Leaf Node: Class Iris-setosa
  petal-width < 1.8?
    petal-length < 5.0?
      petal-width < 1.7?
        Leaf Node: Class Iris-versicolor
        Leaf Node: Class Iris-virginica
      petal-width < 1.6?
        Leaf Node: Class Iris-virginica
        sepal-length < 7.2?
          Leaf Node: Class Iris-versicolor
          Leaf Node: Class Iris-virginica
    petal-length < 4.9?
      sepal-length < 6.0?
        Leaf Node: Class Iris-versicolor
        Leaf Node: Class Iris-virginica
      Leaf Node: Class Iris-virginica
```

RANDOM FOREST CLASSIFIERS IN PYTHON 12 September 2023

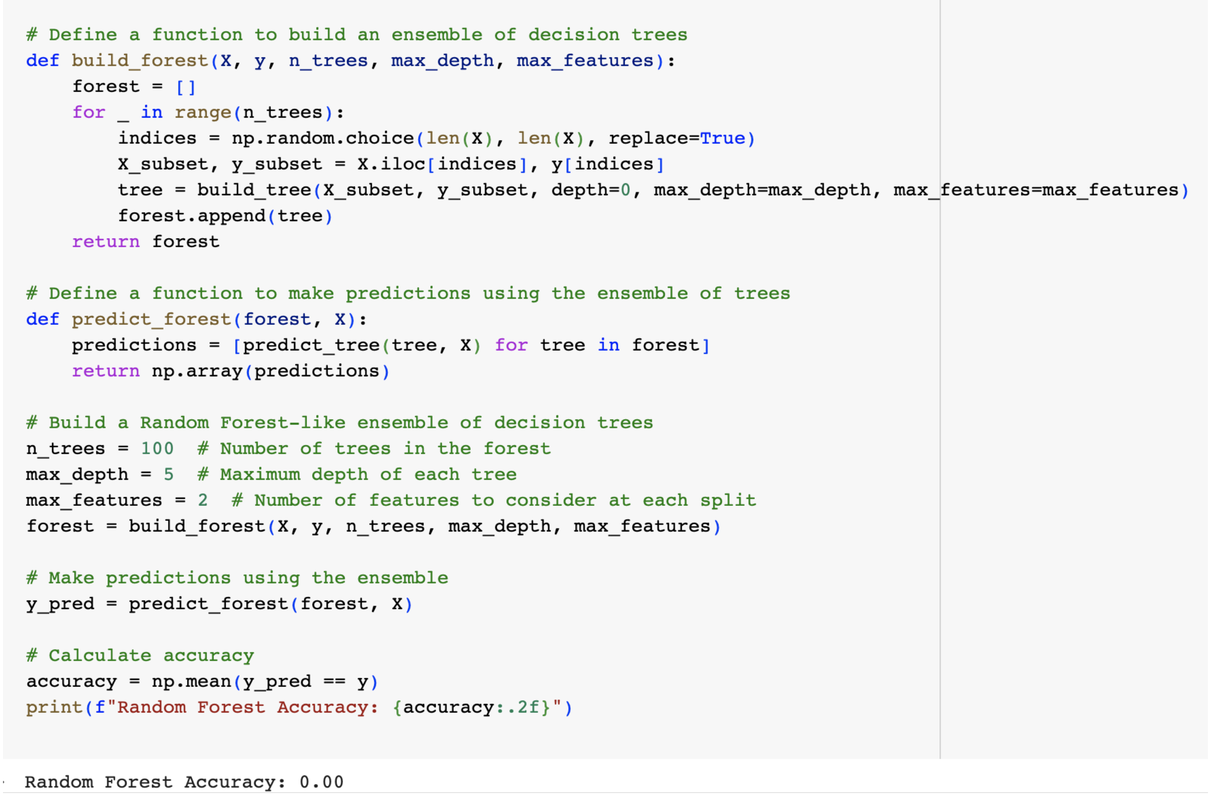
I trained and evaluated a Random Forest classifier in python using the scikit-learn library.

Random Forest classifiers are an ensemble learning algorithm that combines the predictions of multiple Decision Tree classifiers to produce a more accurate and robust prediction. It is a tree-based machine learning model that uses Decision Trees as its base learners.

This is how I trained and evaluated a Random Forest classifier:

1. Define functions to calculate the Gini impurity, build a decision tree, predict using a decision tree, build an ensemble of decision trees, and predict using an ensemble of trees.

2. Build an ensemble of decision trees by repeatedly building decision trees on random subsets of the data and limiting the number of features considered at each split.
3. Make predictions using the ensemble of trees by averaging the predictions of all the trees in the ensemble.



```python
# Define a function to build an ensemble of decision trees
def build_forest(X, y, n_trees, max_depth, max_features):
    forest = []
    for _ in range(n_trees):
        indices = np.random.choice(len(X), len(X), replace=True)
        X_subset, y_subset = X.iloc[indices], y[indices]
        tree = build_tree(X_subset, y_subset, depth=0, max_depth=max_depth, max_features=max_features)
        forest.append(tree)
    return forest

# Define a function to make predictions using the ensemble of trees
def predict_forest(forest, X):
    predictions = [predict_tree(tree, X) for tree in forest]
    return np.array(predictions)

# Build a Random Forest-like ensemble of decision trees
n_trees = 100  # Number of trees in the forest
max_depth = 5  # Maximum depth of each tree
max_features = 2  # Number of features to consider at each split
forest = build_forest(X, y, n_trees, max_depth, max_features)

# Make predictions using the ensemble
y_pred = predict_forest(forest, X)

# Calculate accuracy
accuracy = np.mean(y_pred == y)
print(f"Random Forest Accuracy: {accuracy:.2f}")
```

```
Random Forest Accuracy: 0.00
```

ENTROPY FUNCTION 12 September 2023

Entropy function is a powerful tool for decision tree models. Decision tree models use entropy to split the data into nodes. The node with the hughest entropy is split first. This process is repeated until the data is pure, meaning that all of the data points in a node have the same target value.

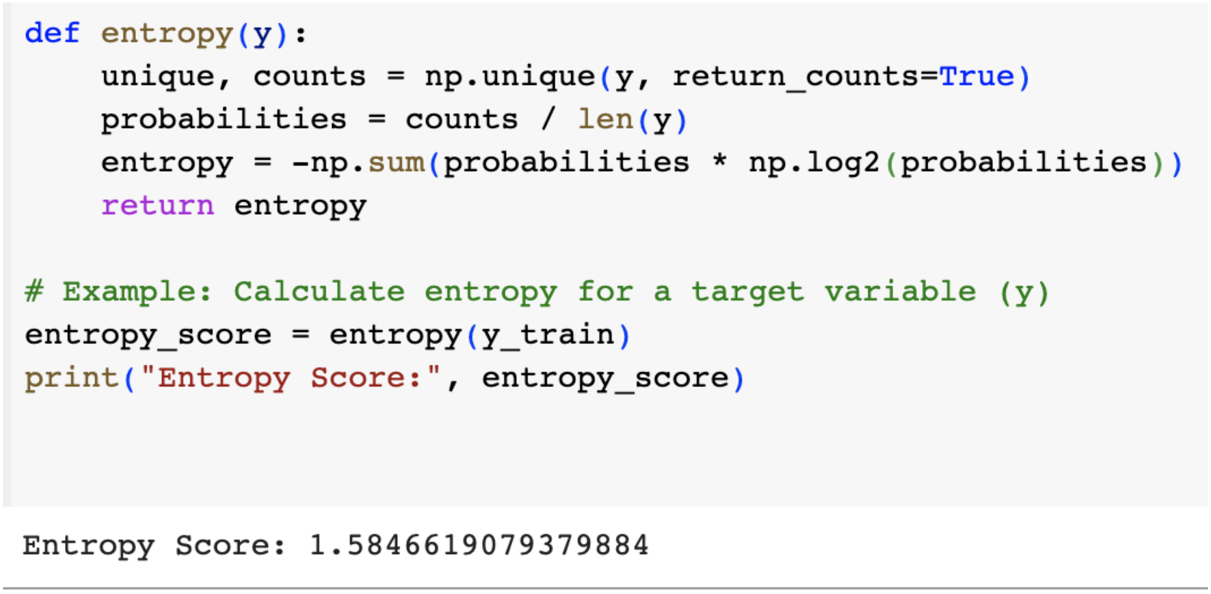
By splitting the data into nodes with lower entropy, decision tree models are able to learn more complex relationships in the data and make more accurate predictions.

First, I used IRIS dataset as my dataset and extract features (X) and target (y) as arrays. Then, I defined a function called entropy(), which calculates the entropy of a target variable. The entropy of a target variable is a measure of how uncertain the target variable is. After that, I calculated the entropy using a formula.

To calculate the entropy, the function first calculates the probability of each unique value in the target variable. Then, it calculates the entropy using the formula:

Entropy = -np.sum(probabilities * np.log2(probabilities))

The entropy is higher when the target variable (y) is more uncertain.



```
def entropy(y):
    unique, counts = np.unique(y, return_counts=True)
    probabilities = counts / len(y)
    entropy = -np.sum(probabilities * np.log2(probabilities))
    return entropy

# Example: Calculate entropy for a target variable (y)
entropy_score = entropy(y_train)
print("Entropy Score:", entropy_score)
```

```
Entropy Score: 1.5846619079379884
```

My entropy score indicates that the target variable in my dataset is moderately uncertain. Thus, there is a significant amount of variation in the target variable, and it is not possible to predict the target value for a new data point with certainty.

There are some possible reasons this occurs:

- The target variable may be inherently uncertain.
- The dataset incomplete. This is because I selected a few data to calculate the entropy function, hence the model may not be able to learn the complex relationships in the data.

After finding the entropy score, I learnt how to calculate the entropy of a target variable in Python, how to use entropy in a decision tree model to split the data into nodes and why entropy is important for decision tree models.

Technical details of a Decision Tree model 13 September 2023

A decision tree model is a supervised machine learning algorithm that can be used for both classification and regression tasks. It works by building a tree-like structure that maps from the features of the data to the target variable.

Each node in the decision tree represents a decision that is made based on the value of a feature. The root node is the top node in the tree, and it represents the most important decision. The leaves of the tree represent the final predictions of the model.

To train a decision tree model, the algorithm starts at the root node and splits the data into two subsets based on the value of a feature. The algorithm then recursively splits the data into smaller and smaller subsets until it reaches the purity criteria, which is a measure of how homogeneous the data is in each subset.

Once the decision tree has been trained, it can be used to make predictions on new data. To make a prediction, the algorithm starts at the root node and follows the path of the tree based on the values of the features in the new data point. The algorithm eventually reaches a leaf node, which is the model's prediction for the target variable.

Decision Tree model Learning Theory Framework 13 September 2023

The learning theory framework is a set of principles and theories that describe how machine learning algorithms learn from data.

One of the key principles of the learning theory framework is the concept of a hypothesis space. The hypothesis space is the set of all possible models that a machine learning algorithm can learn.

Decision tree models are a specific type of machine learning algorithm called inductive learning algorithms. Inductive learning algorithms learn from data by making generalizations about the data.

The hypothesis space for decision tree models is the set of all possible decision trees that can be constructed from the given data.

The learning theory framework also describes the concept of a loss function. The loss function is a measure of how well a given model fits the training data.

The goal of the training process is to find the model in the hypothesis space that minimizes the loss function on the training data. Decision tree models typically use the cross-entropy loss function for classification tasks and the mean squared error loss function for regression tasks.

Reflection **14 September 2023**

Technical Challenges:

1. Data Import and Dataset Selection: Importing a dataset into my working environment can be a challenge, especially when i am new to working with datasets with Google Colab. Understanding the difference between competitive datasets and regular datasets and finding the right name of my dataset can be time-consuming.

2. Data Preparation: Preparing the dataset, including handling missing values, outliers, and data encoding, can be a complex task, especially when working with real-world datasets. In my code, I encountered challenges in encoding the target variable and ensuring that it is in the right format for model training. Additionally, I needed to handle missing values, encode categorical variables, and normalise or scale numerical features.

3. Manual Implementation: Building a decision tree from scratch is a technically challenging task, as I need to implement the logic for splitting nodes, calculating impurity measures (e.g., Gini impurity, entropy), and recursive tree building. This involves understanding the theoretical underpinnings of decision trees and translating them into code.

4. Model Complexity: Decision Tree can be prone to overfitting, especially when the tree grows too deep. Managing model complexity and preventing overfitting is a technical challenge that I encountered.

5. Hyperparameter Tuning: Manually conducting a grid search to optimize hyperparameters can be time-consuming and requires a deep understanding of the model and its hyperparameters. It's challenging to find the best combination of hyperparameters that yields optimal performance.

6. Debugging: Developing complex algorithms like decision trees can lead to errors, as you've experienced. Debugging and resolving errors in a manual implementation can be challenging and time-consuming.

7. Interpretability: While Decision Trees are known for their interpretability, understanding and explaining complex tree structures is difficult for me.

Benefits of Decision Trees:

1. Interpretability: Decision Trees are highly interpretable. I can easily understand and visualise the decisions the model makes.
2. No Linearity Assumptions about Data: Decision Trees make no assumptions about the data distribution, which can be beneficial when working with various types of data. Decision Trees can capture non-linear relationships in data.
3. Handling Categorical Data: Decision Trees can handle categorical data without the need for one-hot encoding.
4. Feature Importance: They provide a measure of feature importance, helping you understand which features contribute most to the model's decisions.

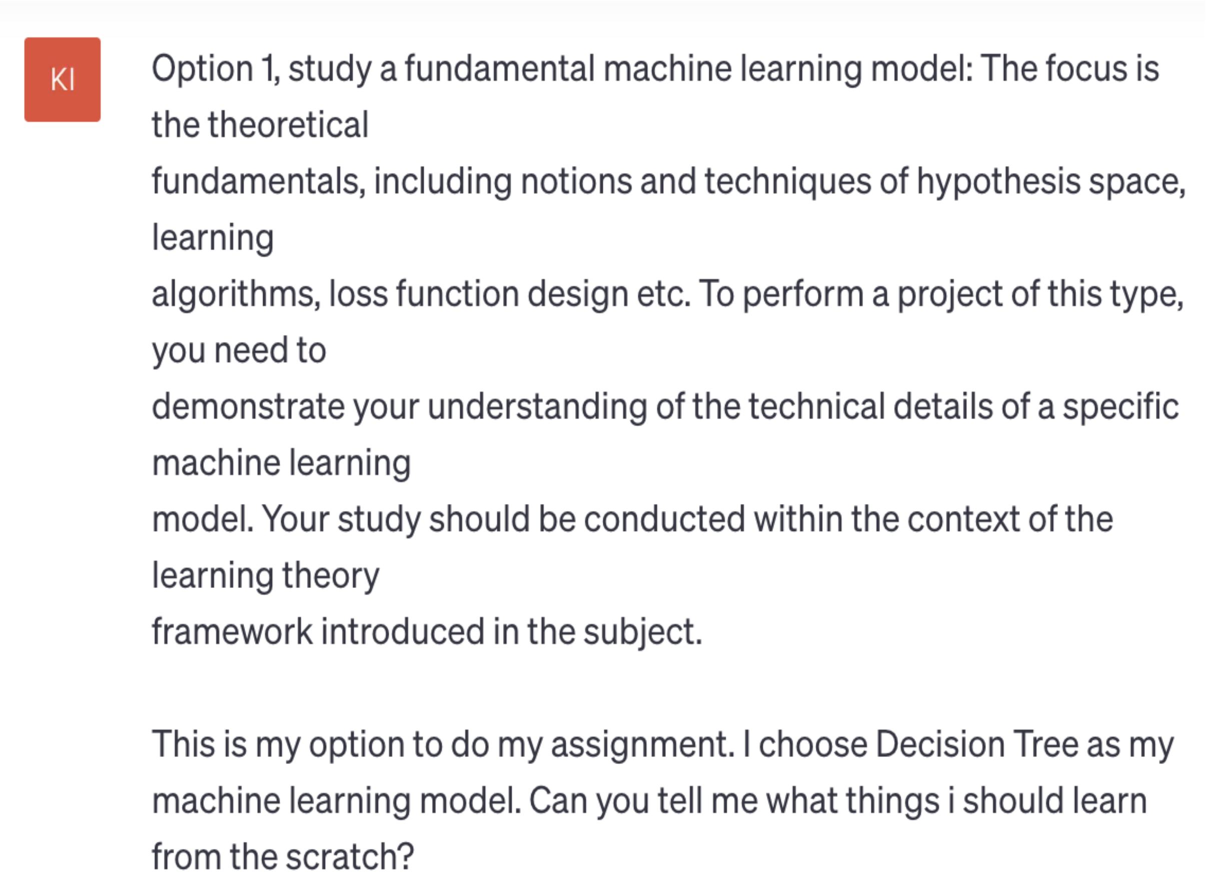
Drawbacks of Decision Trees:

1. Overfitting: Decision Trees can easily overfit the training data, especially when the tree is deep. Pruning techniques are needed to mitigate this issue.
2. Instability: Small variations in the data can lead to different tree structures, making Decision Trees unstable.
3. Bias: Decision Trees can be biased when certain classes dominate. Techniques like class weighting may be necessary.
4. Greedy Search: Decision trees use a greedy approach to find the best split at each node, which might not lead to the globally optimal tree.
5. Limited Expressiveness: For complex tasks, Decision trees may not be expressive enough to capture intricate patterns in the data.

In conclusion, building a Decision Tree model from scratch offers in-depth insights into its functioning and challenges in the manual implementation process. While it provides interpretability, it also comes with challenges related to data preparation, hyperparameter tuning, and potential overfitting. Understanding the trade-offs and limitations of decision trees is essential for effective use in real-world applications.

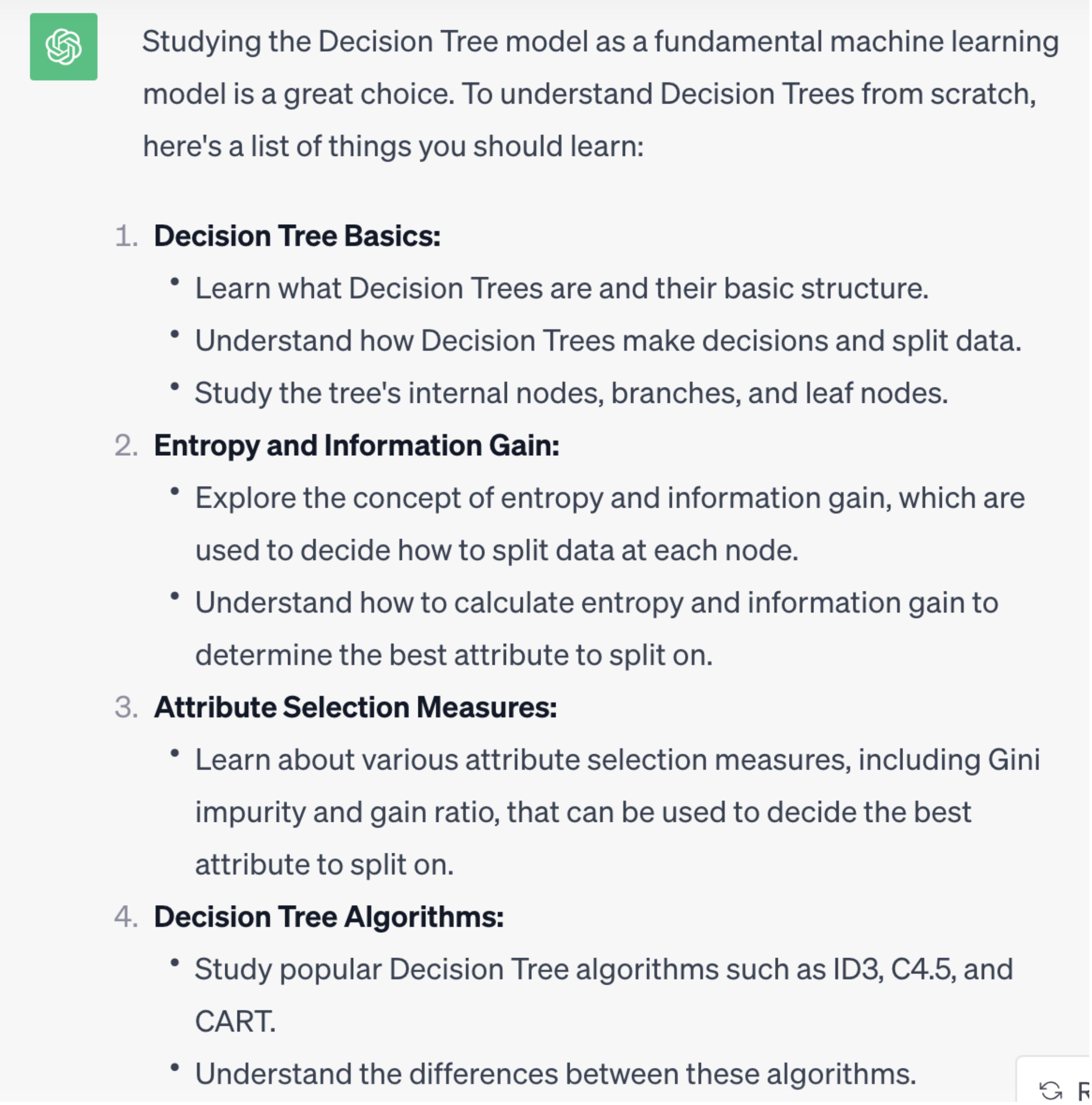
Appendices **14 September 2023**

1. Ask ChatGPT on how to learn a Decision Tree algorithm from scratch to get a strong foundation of a machine learning model. 01 September 2023



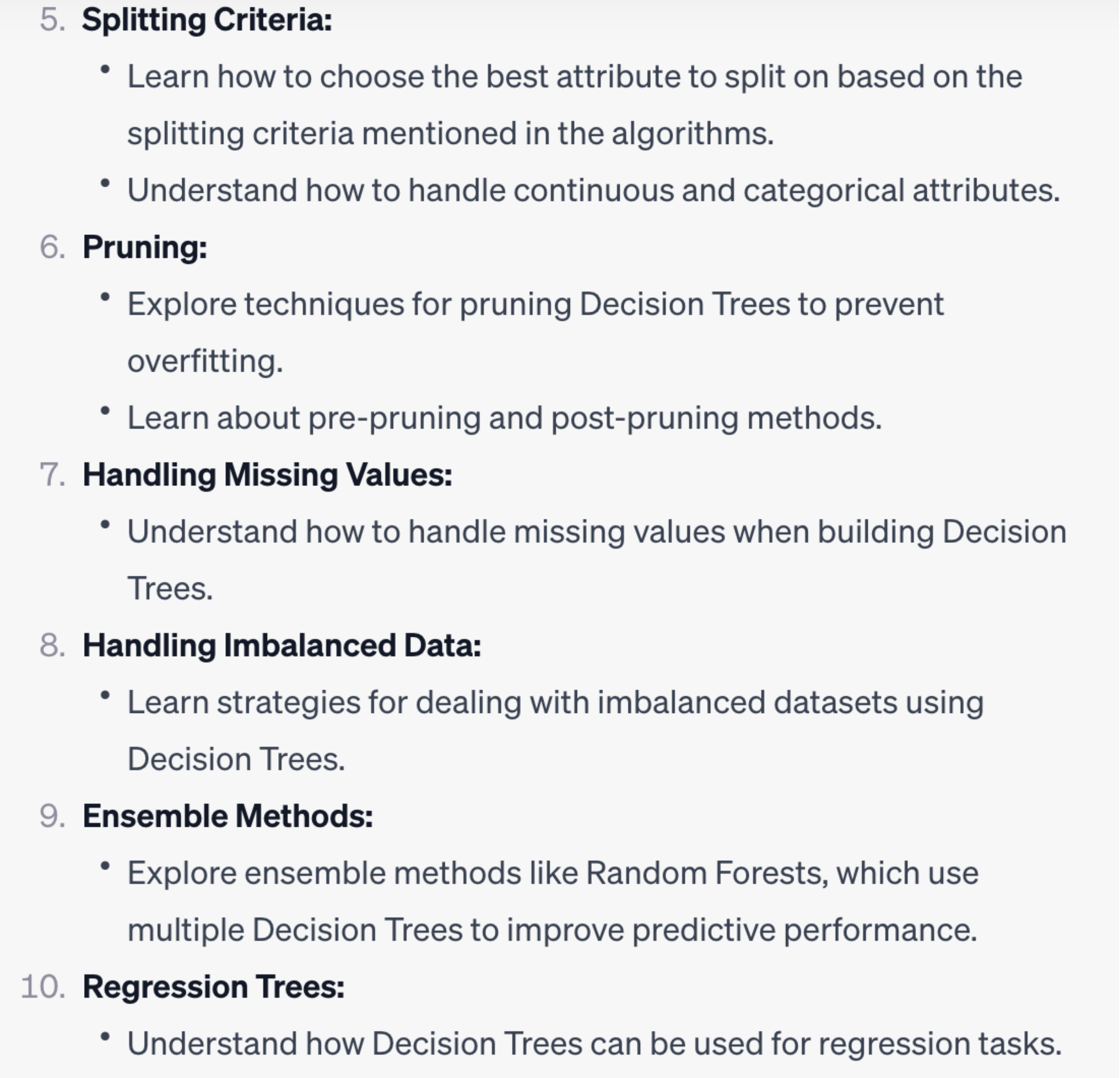
Option 1, study a fundamental machine learning model: The focus is the theoretical fundamentals, including notions and techniques of hypothesis space, learning algorithms, loss function design etc. To perform a project of this type, you need to demonstrate your understanding of the technical details of a specific machine learning model. Your study should be conducted within the context of the learning theory framework introduced in the subject.

This is my option to do my assignment. I choose Decision Tree as my machine learning model. Can you tell me what things i should learn from the scratch?

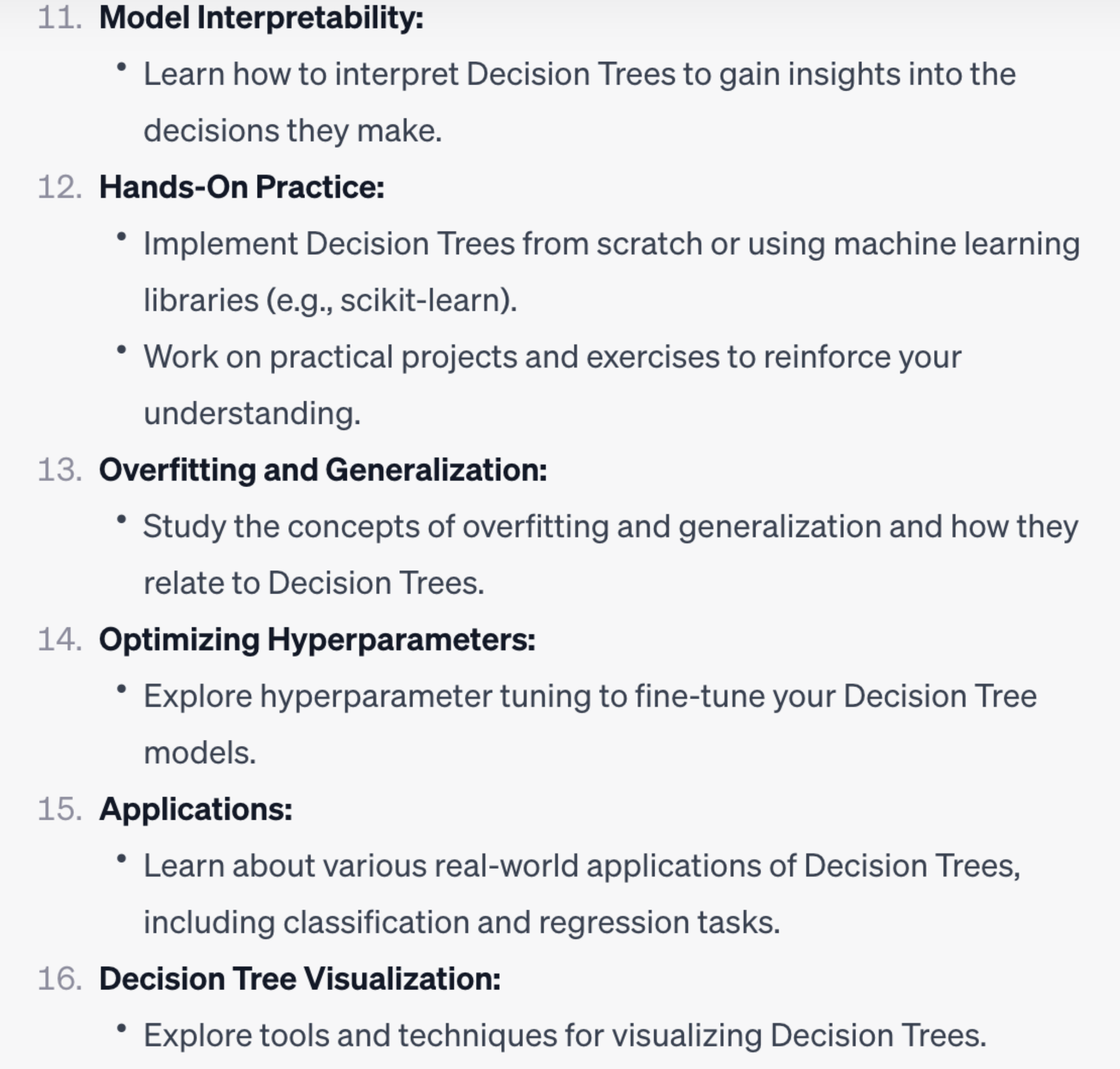


Studying the Decision Tree model as a fundamental machine learning model is a great choice. To understand Decision Trees from scratch, here's a list of things you should learn:

1. **Decision Tree Basics:**
    - Learn what Decision Trees are and their basic structure.
    - Understand how Decision Trees make decisions and split data.
    - Study the tree's internal nodes, branches, and leaf nodes.
2. **Entropy and Information Gain:**
    - Explore the concept of entropy and information gain, which are used to decide how to split data at each node.
    - Understand how to calculate entropy and information gain to determine the best attribute to split on.
3. **Attribute Selection Measures:**
    - Learn about various attribute selection measures, including Gini impurity and gain ratio, that can be used to decide the best attribute to split on.
4. **Decision Tree Algorithms:**
    - Study popular Decision Tree algorithms such as ID3, C4.5, and CART.
    - Understand the differences between these algorithms.



5. **Splitting Criteria:**
   - Learn how to choose the best attribute to split on based on the splitting criteria mentioned in the algorithms.
   - Understand how to handle continuous and categorical attributes.
6. **Pruning:**
   - Explore techniques for pruning Decision Trees to prevent overfitting.
   - Learn about pre-pruning and post-pruning methods.
7. **Handling Missing Values:**
   - Understand how to handle missing values when building Decision Trees.
8. **Handling Imbalanced Data:**
   - Learn strategies for dealing with imbalanced datasets using Decision Trees.
9. **Ensemble Methods:**
   - Explore ensemble methods like Random Forests, which use multiple Decision Trees to improve predictive performance.
10. **Regression Trees:**
    - Understand how Decision Trees can be used for regression tasks.

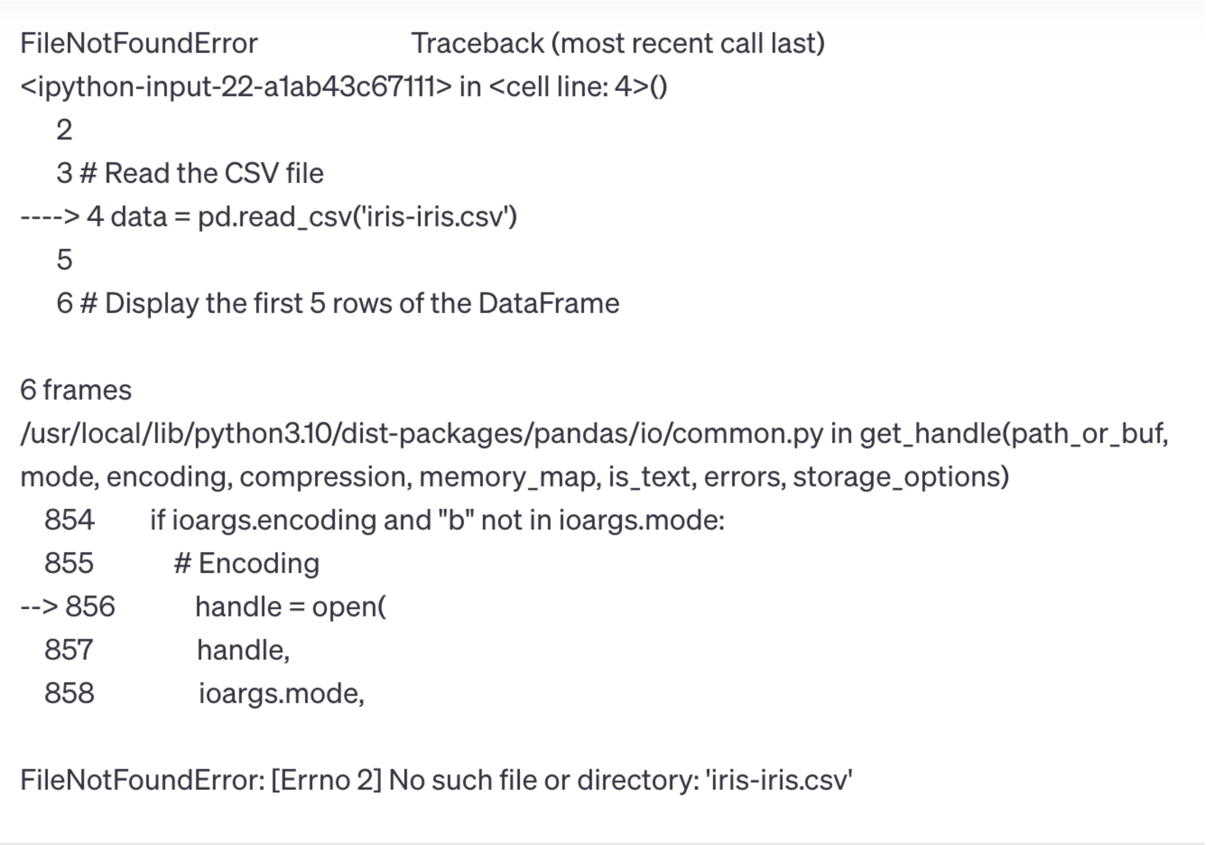


11. **Model Interpretability:**
    - Learn how to interpret Decision Trees to gain insights into the decisions they make.
12. **Hands-On Practice:**
    - Implement Decision Trees from scratch or using machine learning libraries (e.g., scikit-learn).
    - Work on practical projects and exercises to reinforce your understanding.
13. **Overfitting and Generalization:**
    - Study the concepts of overfitting and generalization and how they relate to Decision Trees.
14. **Optimizing Hyperparameters:**
    - Explore hyperparameter tuning to fine-tune your Decision Tree models.
15. **Applications:**
    - Learn about various real-world applications of Decision Trees, including classification and regression tasks.
16. **Decision Tree Visualization:**
    - Explore tools and techniques for visualizing Decision Trees.

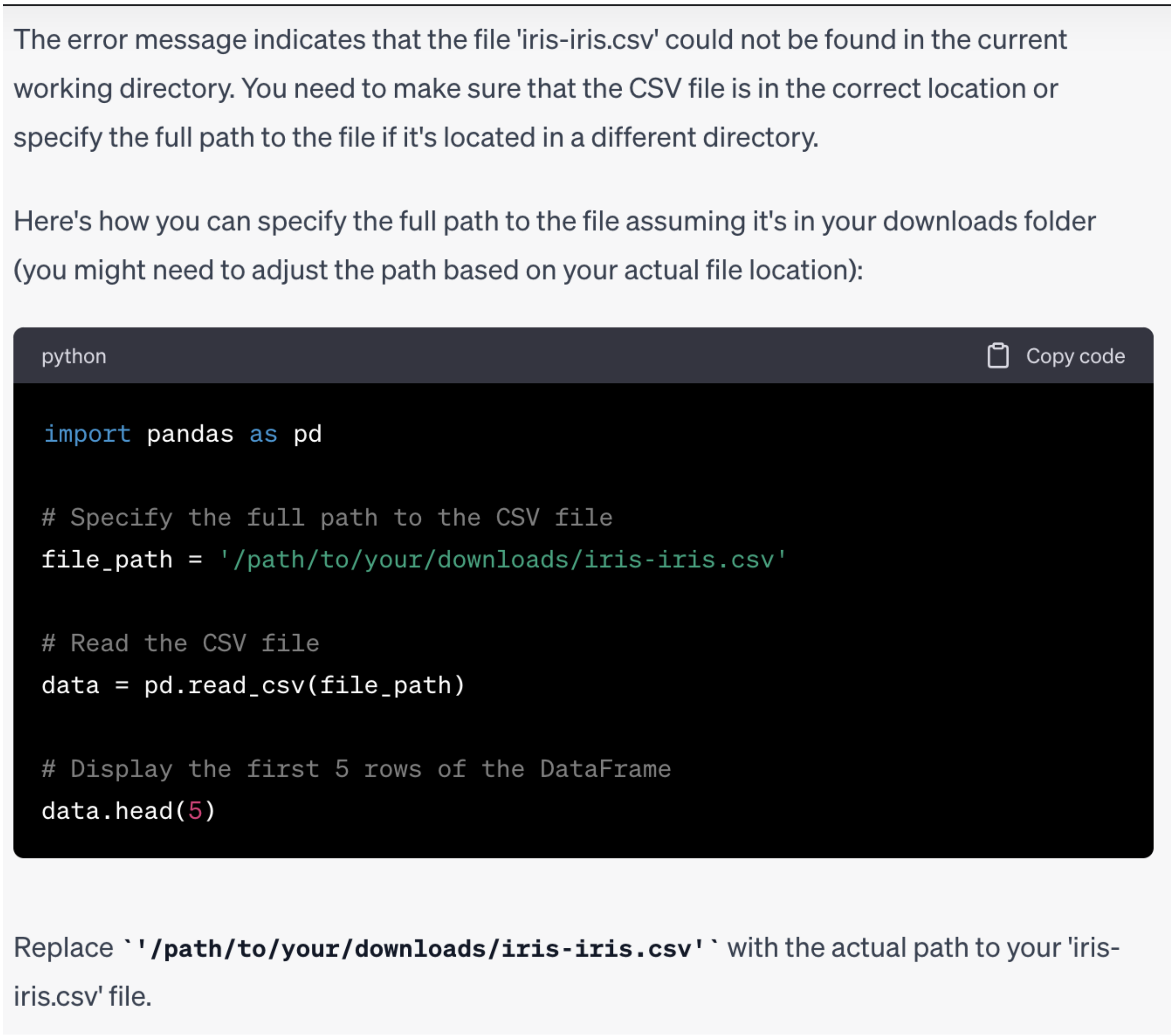
17. **Learning Theory Framework:**
    - Understand how Decision Trees fit into the broader framework of machine learning and learning theory.
18. **Evaluate Model Performance:**
    - Learn how to evaluate the performance of Decision Tree models using metrics like accuracy, precision, recall, F1-score, and ROC curves.

As you progress through these topics, you'll build a strong foundation in Decision Trees and be well-prepared to work with this machine learning model effectively. Remember that practice and hands-on projects are essential to solidify your understanding.

2. Ask ChatGPT to find the right dataset name to input into Google Colab. 02 September 2023



```
FileNotFoundError                    Traceback (most recent call last)
<ipython-input-22-a1ab43c67111> in <cell line: 4>()
      2
      3 # Read the CSV file
----> 4 data = pd.read_csv('iris-iris.csv')
      5
      6 # Display the first 5 rows of the DataFrame

6 frames
/usr/local/lib/python3.10/dist-packages/pandas/io/common.py in get_handle(path_or_buf, mode, encoding, compression, memory_map, is_text, errors, storage_options)
    854         if ioargs.encoding and "b" not in ioargs.mode:
    855             # Encoding
--> 856             handle = open(
    857                 handle,
    858                 ioargs.mode,

FileNotFoundError: [Errno 2] No such file or directory: 'iris-iris.csv'
```



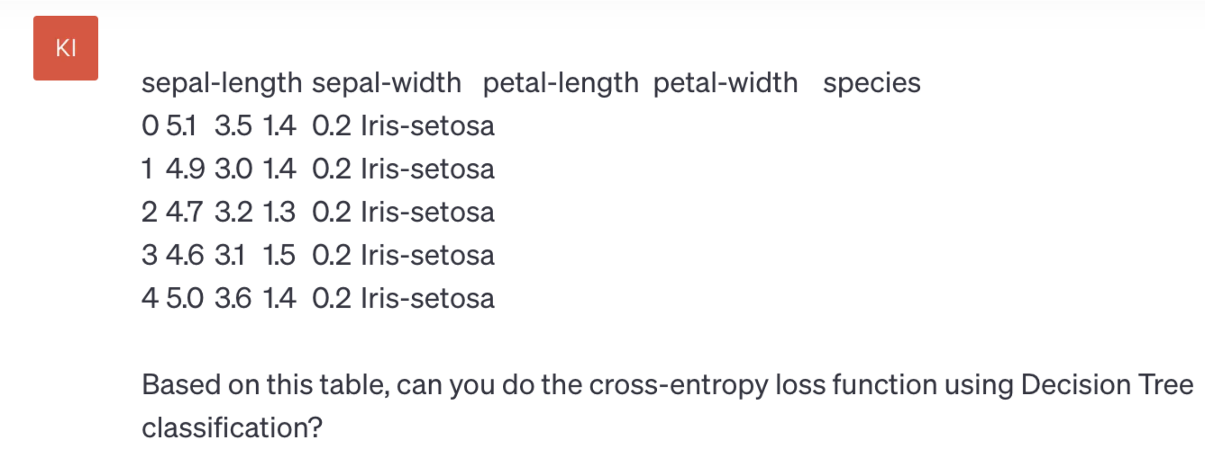
The error message indicates that the file 'iris-iris.csv' could not be found in the current working directory. You need to make sure that the CSV file is in the correct location or specify the full path to the file if it's located in a different directory.

Here's how you can specify the full path to the file assuming it's in your downloads folder (you might need to adjust the path based on your actual file location):

```python
import pandas as pd

# Specify the full path to the CSV file
file_path = '/path/to/your/downloads/iris-iris.csv'

# Read the CSV file
data = pd.read_csv(file_path)

# Display the first 5 rows of the DataFrame
data.head(5)
```

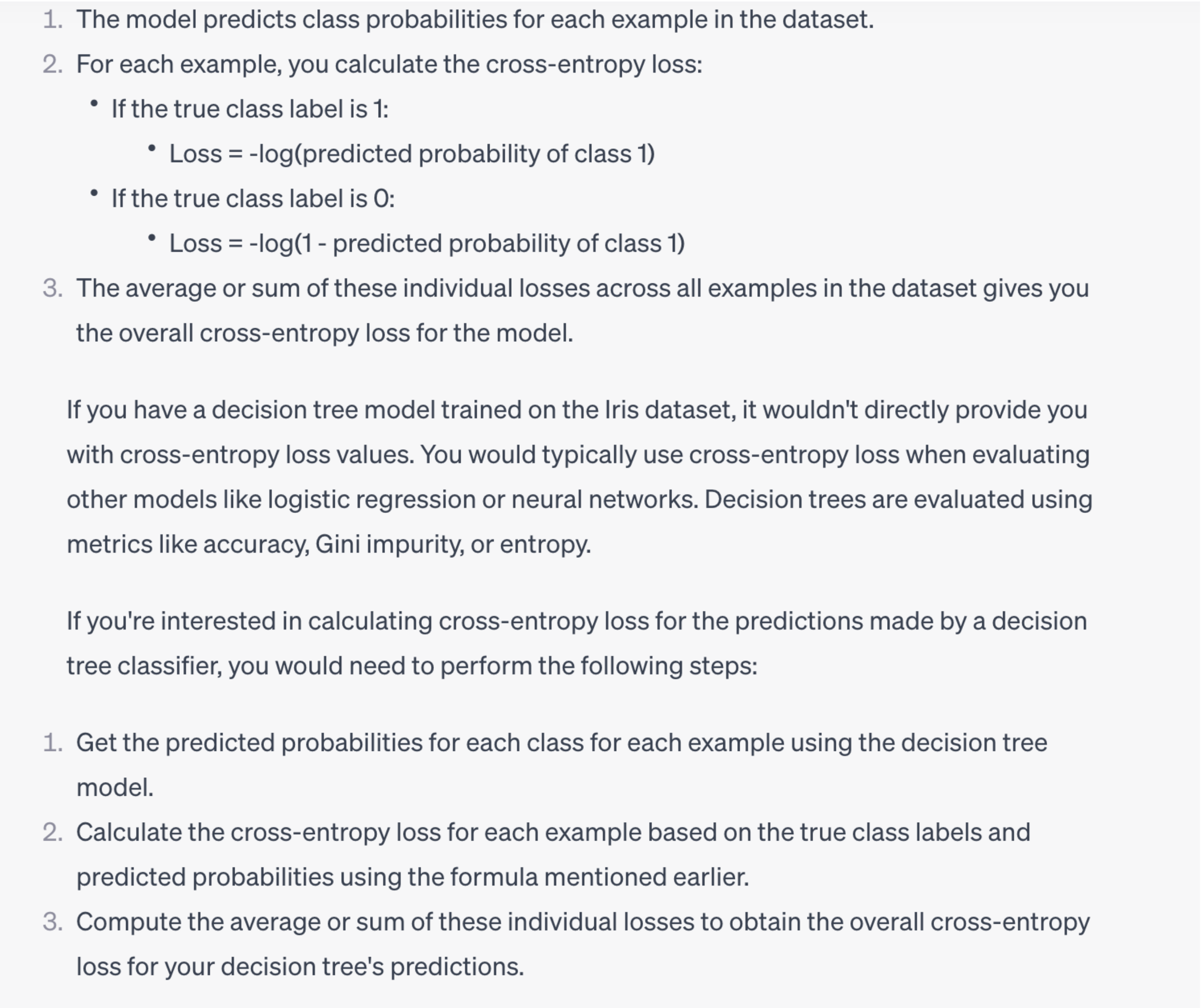
Replace `'/path/to/your/downloads/iris-iris.csv'` with the actual path to your 'iris-iris.csv' file.

3. Ask ChatGPT about the cross-entropy loss function. 12 September 2023



sepal-length sepal-width petal-length petal-width species
0 5.1 3.5 1.4 0.2 Iris-setosa
1 4.9 3.0 1.4 0.2 Iris-setosa
2 4.7 3.2 1.3 0.2 Iris-setosa
3 4.6 3.1 1.5 0.2 Iris-setosa
4 5.0 3.6 1.4 0.2 Iris-setosa

Based on this table, can you do the cross-entropy loss function using Decision Tree classification?



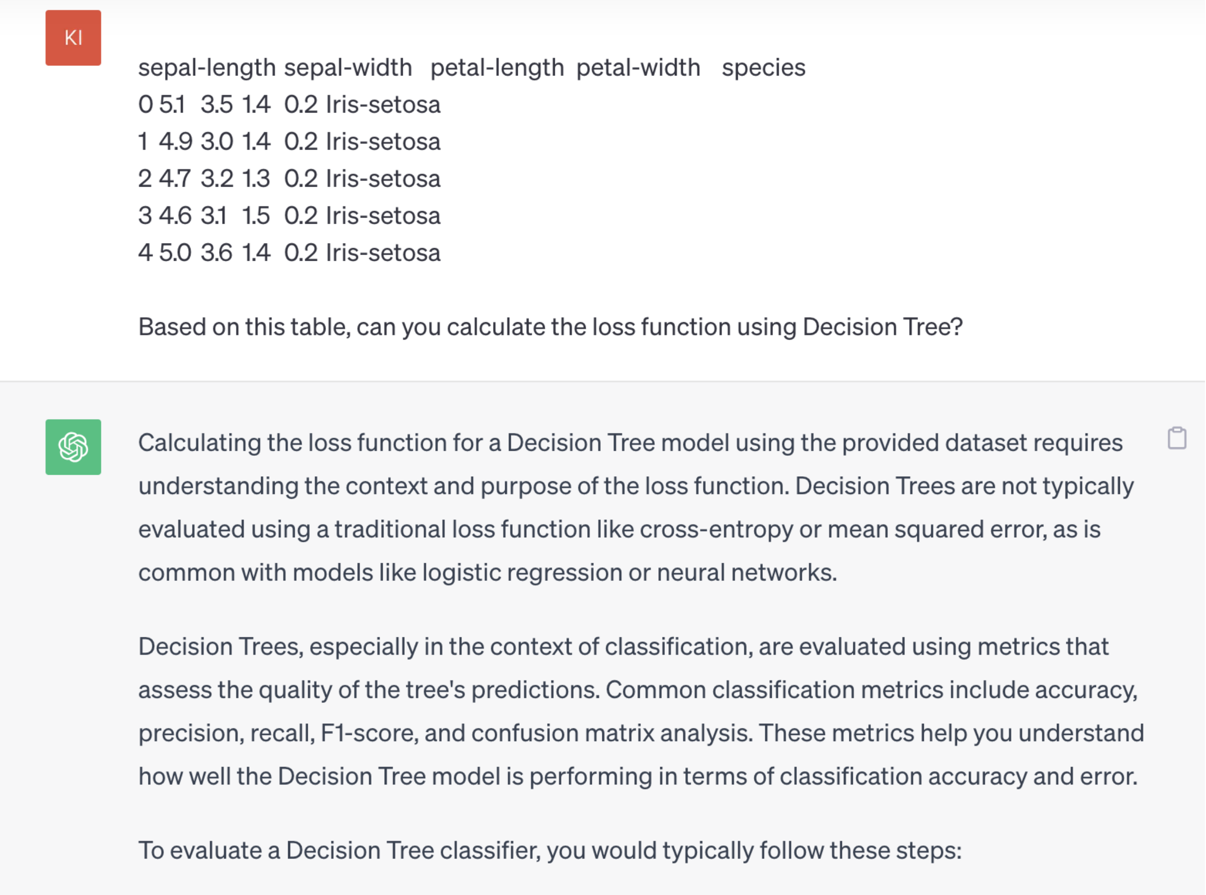
1. The model predicts class probabilities for each example in the dataset.
2. For each example, you calculate the cross-entropy loss:
   - If the true class label is 1:
     - Loss = -log(predicted probability of class 1)
   - If the true class label is 0:
     - Loss = -log(1 - predicted probability of class 1)
3. The average or sum of these individual losses across all examples in the dataset gives you the overall cross-entropy loss for the model.

If you have a decision tree model trained on the Iris dataset, it wouldn't directly provide you with cross-entropy loss values. You would typically use cross-entropy loss when evaluating other models like logistic regression or neural networks. Decision trees are evaluated using metrics like accuracy, Gini impurity, or entropy.

If you're interested in calculating cross-entropy loss for the predictions made by a decision tree classifier, you would need to perform the following steps:

1. Get the predicted probabilities for each class for each example using the decision tree model.
2. Calculate the cross-entropy loss for each example based on the true class labels and predicted probabilities using the formula mentioned earlier.
3. Compute the average or sum of these individual losses to obtain the overall cross-entropy loss for your decision tree's predictions.

4. Ask ChatGPT to calculate my loss function. 04 September 2023



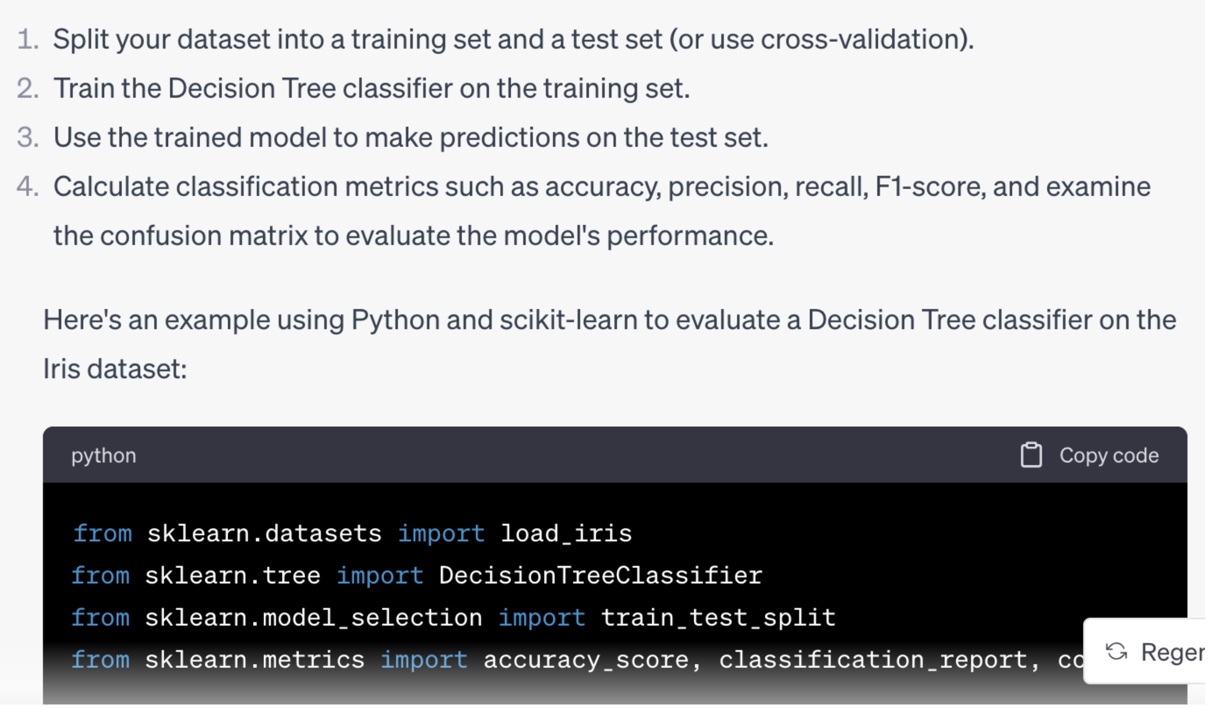
sepal-length sepal-width petal-length petal-width   species
0 5.1 3.5 1.4 0.2 Iris-setosa
1 4.9 3.0 1.4 0.2 Iris-setosa
2 4.7 3.2 1.3 0.2 Iris-setosa
3 4.6 3.1 1.5 0.2 Iris-setosa
4 5.0 3.6 1.4 0.2 Iris-setosa

Based on this table, can you calculate the loss function using Decision Tree?

Calculating the loss function for a Decision Tree model using the provided dataset requires understanding the context and purpose of the loss function. Decision Trees are not typically evaluated using a traditional loss function like cross-entropy or mean squared error, as is common with models like logistic regression or neural networks.

Decision Trees, especially in the context of classification, are evaluated using metrics that assess the quality of the tree's predictions. Common classification metrics include accuracy, precision, recall, F1-score, and confusion matrix analysis. These metrics help you understand how well the Decision Tree model is performing in terms of classification accuracy and error.

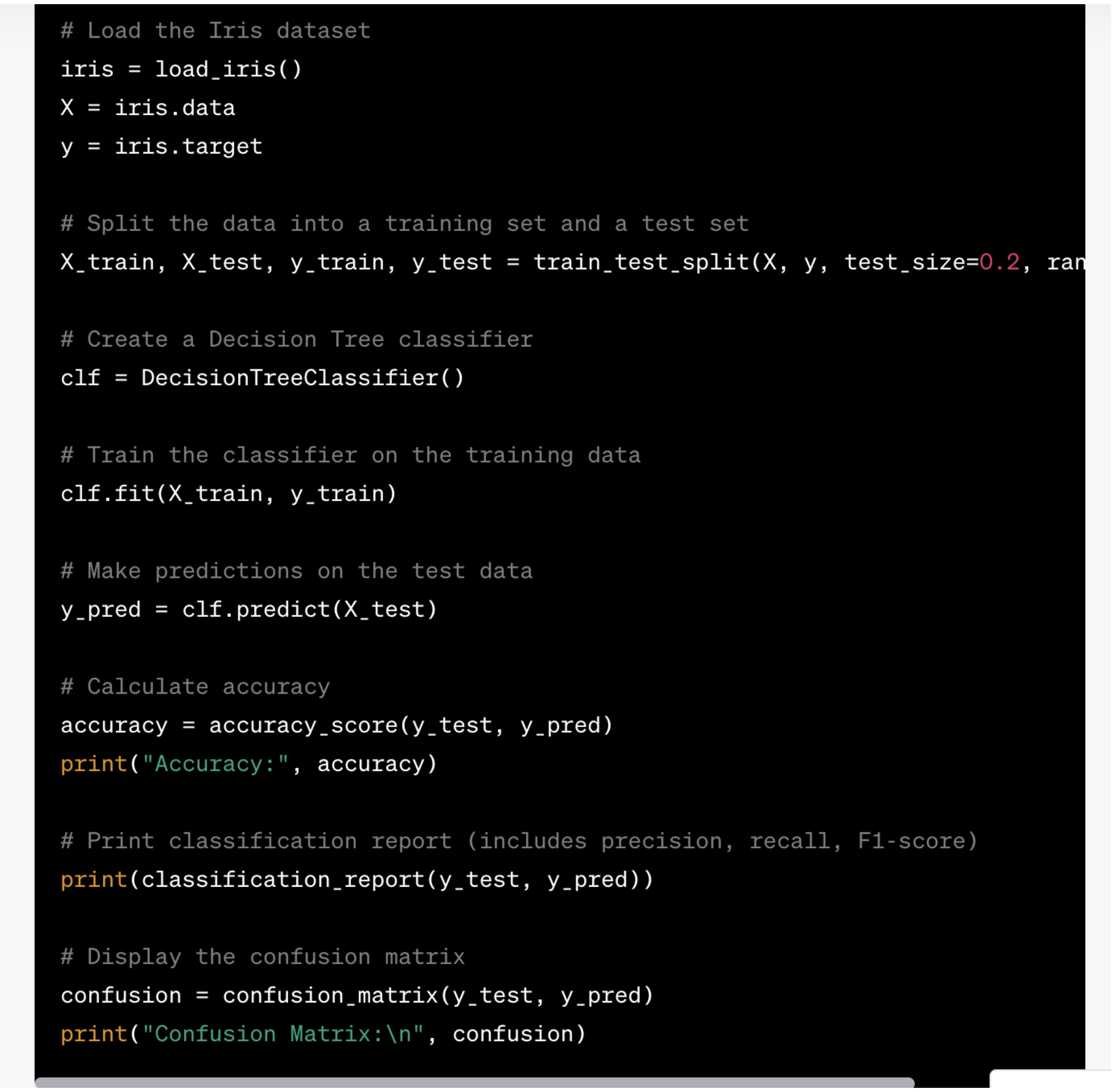
To evaluate a Decision Tree classifier, you would typically follow these steps:



1. Split your dataset into a training set and a test set (or use cross-validation).
2. Train the Decision Tree classifier on the training set.
3. Use the trained model to make predictions on the test set.
4. Calculate classification metrics such as accuracy, precision, recall, F1-score, and examine the confusion matrix to evaluate the model's performance.

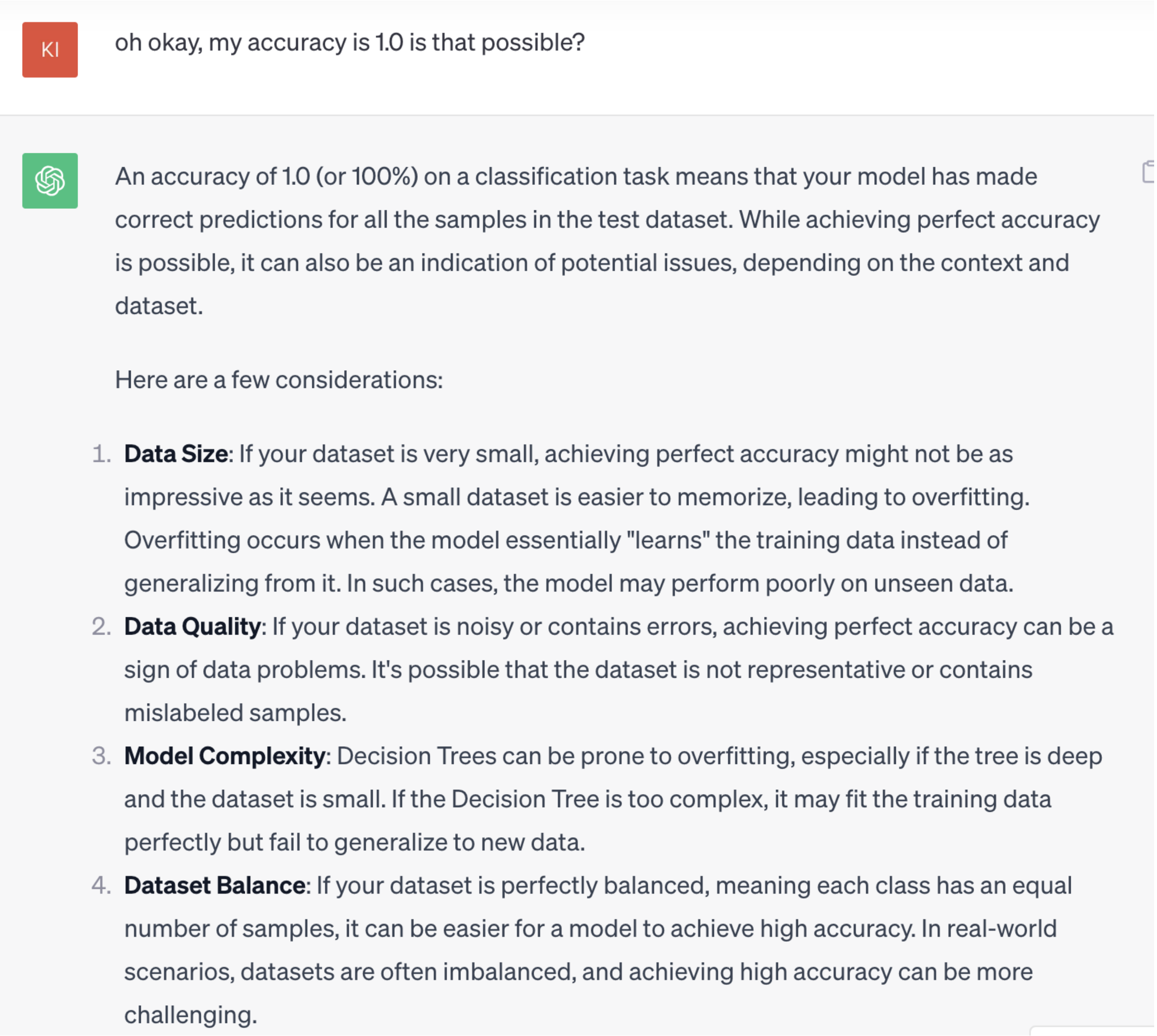
Here's an example using Python and scikit-learn to evaluate a Decision Tree classifier on the Iris dataset:

```python
from sklearn.datasets import load_iris
from sklearn.tree import DecisionTreeClassifier
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score, classification_report, co
```



```
# Load the Iris dataset
iris = load_iris()
X = iris.data
y = iris.target

# Split the data into a training set and a test set
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, ran

# Create a Decision Tree classifier
clf = DecisionTreeClassifier()

# Train the classifier on the training data
clf.fit(X_train, y_train)

# Make predictions on the test data
y_pred = clf.predict(X_test)

# Calculate accuracy
accuracy = accuracy_score(y_test, y_pred)
print("Accuracy:", accuracy)

# Print classification report (includes precision, recall, F1-score)
print(classification_report(y_test, y_pred))

# Display the confusion matrix
confusion = confusion_matrix(y_test, y_pred)
print("Confusion Matrix:\n", confusion)
```

5. Ask ChatGPT how my accuracy's result. 06 September 2023

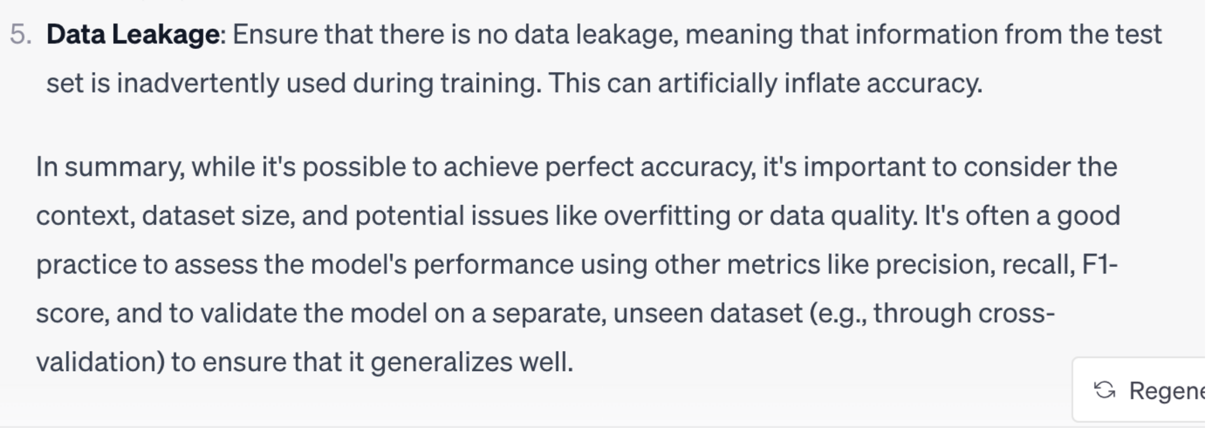


oh okay, my accuracy is 1.0 is that possible?

An accuracy of 1.0 (or 100%) on a classification task means that your model has made correct predictions for all the samples in the test dataset. While achieving perfect accuracy is possible, it can also be an indication of potential issues, depending on the context and dataset.

Here are a few considerations:

1. **Data Size**: If your dataset is very small, achieving perfect accuracy might not be as impressive as it seems. A small dataset is easier to memorize, leading to overfitting. Overfitting occurs when the model essentially "learns" the training data instead of generalizing from it. In such cases, the model may perform poorly on unseen data.
2. **Data Quality**: If your dataset is noisy or contains errors, achieving perfect accuracy can be a sign of data problems. It's possible that the dataset is not representative or contains mislabeled samples.
3. **Model Complexity**: Decision Trees can be prone to overfitting, especially if the tree is deep and the dataset is small. If the Decision Tree is too complex, it may fit the training data perfectly but fail to generalize to new data.
4. **Dataset Balance**: If your dataset is perfectly balanced, meaning each class has an equal number of samples, it can be easier for a model to achieve high accuracy. In real-world scenarios, datasets are often imbalanced, and achieving high accuracy can be more challenging.



5. **Data Leakage**: Ensure that there is no data leakage, meaning that information from the test set is inadvertently used during training. This can artificially inflate accuracy.

In summary, while it's possible to achieve perfect accuracy, it's important to consider the context, dataset size, and potential issues like overfitting or data quality. It's often a good practice to assess the model's performance using other metrics like precision, recall, F1-score, and to validate the model on a separate, unseen dataset (e.g., through cross-validation) to ensure that it generalizes well.

Google Colab Link:
https://colab.research.google.com/drive/10c3D5o1fw5AvLbxIqcUXDcdF6ash3yp8?usp=sharing