1. What is the estimated depth of a Decision Tree trained (unrestricted) on a one million instance training set?

A1. The estimated depth of a Decision Tree trained on a one million instance training set would depend on various factors, such as the number of features, the complexity of the problem, and the criterion used for splitting nodes. In general, an unrestricted decision tree may grow very deep and complex, leading to overfitting and poor generalization performance. To prevent overfitting, various regularization techniques such as pruning, early stopping, or ensemble methods can be used. In practice, the depth of a decision tree is often determined empirically using cross-validation or grid search.

2. Is the Gini impurity of a node usually lower or higher than that of its parent? Is it always lower/greater, or is it usually lower/greater?

A2. The Gini impurity of a node is usually lower than that of its parent after a split. The purpose of splitting a node is to increase the homogeneity of its child nodes, resulting in a lower impurity measure. Therefore, the Gini impurity of a node will usually be lower after a split than that of its parent. However, there may be cases where the Gini impurity remains the same after a split, for example, when all instances in a node have the same class label.

3. Explain if its a good idea to reduce max depth if a Decision Tree is overfitting the training set?

A3. Yes, reducing the maximum depth of a decision tree is a common technique used to address overfitting. Overfitting occurs when the decision tree becomes too complex and fits the noise in the training data, resulting in poor performance on unseen data. By reducing the maximum depth, the tree becomes less complex, and the model generalizes better to new data.

However, it is essential to find a balance between underfitting and overfitting. If the tree is too shallow, it may underfit the training data and lead to poor performance on both the training and test sets. Therefore, it is often necessary to experiment with different values of the maximum depth and evaluate the model's performance on a validation set before deciding on the optimal value.

4. Explain if its a good idea to try scaling the input features if a Decision Tree underfits the training set?

A4.   
Scaling input features is not necessary for Decision Trees as they don't rely on the scale of the features. In a Decision Tree, each split compares a feature to a threshold, and the scale of the feature does not matter as long as the threshold is adjusted accordingly. However, scaling can be helpful if the tree depth is limited and a feature has a much larger range of values than other features. In this case, the feature with the larger range of values may be prioritized over the other features, and scaling can help to prevent this bias. However, in general, adjusting the hyperparameters of the Decision Tree, such as the maximum depth, is the better approach to address underfitting.

5. How much time will it take to train another Decision Tree on a training set of 10 million instances if it takes an hour to train a Decision Tree on a training set with 1 million instances?

A5. Assuming that the training time scales linearly with the number of instances, training a Decision Tree on a training set with 10 million instances would take approximately 10 hours (1 hour for 1 million instances multiplied by 10 for 10 million instances).

6. Will setting presort=True speed up training if your training set has 100,000 instances?

A6.   
No, setting **presort=True** will actually slow down the training process for large datasets, such as the one with 100,000 instances. **presort=True** is useful only when the training set is small enough that it can be entirely sorted in memory. Otherwise, it will take a considerable amount of time and computational resources to sort the data, which will slow down the training process significantly. In general, it is best to leave **presort** as False, especially for large datasets.

7. Follow these steps to train and fine-tune a Decision Tree for the moons dataset:

a. To build a moons dataset, use make moons(n samples=10000, noise=0.4).

from sklearn.datasets import make\_moons

X, y = make\_moons(n\_samples=10000, noise=0.4, random\_state=42)

b. Divide the dataset into a training and a test collection with train test split().

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)

c. To find good hyperparameters values for a DecisionTreeClassifier, use grid search with cross-validation (with the GridSearchCV class). Try different values for max leaf nodes.

from sklearn.tree import DecisionTreeClassifier

from sklearn.model\_selection import GridSearchCV

param\_grid = {"max\_leaf\_nodes": list(range(2, 100)), "min\_samples\_split": [2, 3, 4]}

tree\_clf = DecisionTreeClassifier(random\_state=42)

grid\_search = GridSearchCV(tree\_clf, param\_grid, cv=5, verbose=1, n\_jobs=-1)

grid\_search.fit(X\_train, y\_train)

d. Use these hyperparameters to train the model on the entire training set, and then assess its output on the test set. You can achieve an accuracy of 85 to 87 percent.

from sklearn.metrics import accuracy\_score

best\_clf = grid\_search.best\_estimator\_

y\_pred = best\_clf.predict(X\_test)

accuracy = accuracy\_score(y\_test, y\_pred)

print("Accuracy:", accuracy)

8. Follow these steps to grow a forest:

a. Using the same method as before, create 1,000 subsets of the training set, each containing 100 instances chosen at random. You can do this with Scikit-ShuffleSplit Learn's class.

b. Using the best hyperparameter values found in the previous exercise, train one Decision Tree on each subset. On the test collection, evaluate these 1,000 Decision Trees. These Decision Trees would likely perform worse than the first Decision Tree, achieving only around 80% accuracy, since they were trained on smaller sets.

c. Now the magic begins. Create 1,000 Decision Tree predictions for each test set case, and keep only the most common prediction (you can do this with SciPy's mode() function). Over the test collection, this method gives you majority-vote predictions.

d. On the test range, evaluate these predictions: you should achieve a slightly higher accuracy than the first model (approx 0.5 to 1.5 percent higher). You've successfully learned a Random Forest classifier!

A8.

from sklearn.datasets import make\_moons

from sklearn.model\_selection import train\_test\_split, ShuffleSplit, GridSearchCV

from sklearn.tree import DecisionTreeClassifier

from scipy.stats import mode

# Step 1: Create a moons dataset

X, y = make\_moons(n\_samples=10000, noise=0.4)

# Step 2: Split the dataset into train and test sets

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=42)

# Step 3: Tune hyperparameters for a single Decision Tree using GridSearchCV

param\_grid = {'max\_leaf\_nodes': list(range(2, 100))}

tree\_clf = DecisionTreeClassifier(random\_state=42)

grid\_search = GridSearchCV(tree\_clf, param\_grid, cv=3)

grid\_search.fit(X\_train, y\_train)

best\_params = grid\_search.best\_params\_

# Step 4: Create a random forest by training multiple Decision Trees on random subsets of the training data

n\_trees = 1000

n\_instances = 100

forest = []

rs = ShuffleSplit(n\_splits=n\_trees, test\_size=len(X\_train) - n\_instances, random\_state=42)

for train\_index, \_ in rs.split(X\_train):

tree\_clf = DecisionTreeClassifier(max\_leaf\_nodes=best\_params['max\_leaf\_nodes'], random\_state=42)

tree\_clf.fit(X\_train[train\_index], y\_train[train\_index])

forest.append(tree\_clf)

# Step 5: Evaluate the performance of the forest on the test set

y\_pred = []

for tree\_clf in forest:

y\_pred.append(tree\_clf.predict(X\_test))

y\_pred\_majority\_votes, \_ = mode(y\_pred, axis=0)

accuracy = np.mean(y\_pred\_majority\_votes.ravel() == y\_test)

print('Random Forest Classifier Accuracy:', accuracy)