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

The estimated depth of a Decision Tree trained on a one million instance training set can vary significantly based on several factors such as the complexity of the data, the features, and the nature of the problem. However, we can provide a rough estimate based on typical scenarios and empirical observations:

1. **General Estimate**: For a large dataset with one million instances, Decision Trees tend to grow deeper to capture more details and nuances in the data. It's not uncommon to see depths ranging from 50 to 100 levels or more, especially if the tree is unrestricted (i.e., not pruned).
2. **Empirical Rule of Thumb**: One empirical guideline often cited is that the depth of a Decision Tree grows logarithmically with the number of training instances. This means that as the number of instances increases linearly, the depth of the tree increases logarithmically. For a one million instance dataset, this could imply a tree depth around ***log2​(106) ≈ 20***.
3. **Practical Considerations**: In practice, Decision Trees can exceed this depth estimate, especially when the dataset is complex or noisy, or when the features have a high degree of granularity.
4. **Algorithm and Parameters**: The actual depth also depends on the specific algorithm used (e.g., CART, ID3, C4.5, etc.) and any parameters set for pruning or stopping criteria. Unrestricted growth typically leads to deeper trees.
5. **Problem-Specific Factors**: The inherent complexity of the problem being solved by the Decision Tree also influences its depth. Problems requiring more intricate decision boundaries or handling more diverse data tend to result in deeper trees.

Therefore, while an exact depth cannot be pinpointed without specific details of the dataset and the learning algorithm used, a reasonable estimate for the depth of a Decision Tree trained on a one million instance training set could range from approximately 50 to 100 levels, with deeper trees not being uncommon depending on the factors mentioned above.

# 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?

In a Decision Tree algorithm using Gini impurity as the criterion for splitting nodes, the Gini impurity of a node after a split is usually lower than or equal to that of its parent node. Here’s why:

1. **Gini Impurity Definition**: Gini impurity measures the degree of impurity or disorder in a set of examples. A lower Gini impurity indicates that the node is more homogeneous (less impure).
2. **Splitting Criterion**: When building a Decision Tree, at each node, the algorithm chooses the split that maximally reduces the Gini impurity. This means that after a split, the resulting child nodes are typically more homogeneous (lower Gini impurity) than the original parent node.
3. **Optimization Goal**: The goal of the Decision Tree algorithm (specifically using Gini impurity) is to partition the data in such a way that each split reduces the impurity, ideally maximizing the purity of the resulting nodes.
4. **Equality or Greater Gini Impurity**: It's possible for a split to result in child nodes with the same Gini impurity as the parent (especially if there are duplicate values or very similar distributions among the resulting groups), but it's rare for a split to result in higher Gini impurity because the algorithm inherently seeks to minimize impurity.

Therefore, while it’s not a strict rule that the Gini impurity of a node is always lower than that of its parent after a split, it is typically the case due to the optimization process of the Decision Tree algorithm. The algorithm's objective is to partition the data into more homogeneous groups with each split, aiming to reduce impurity at each step.

# Explain if it’s a good idea to reduce max depth if a Decision Tree is overfitting the training set?

Yes, reducing the maximum depth of a Decision Tree can be a good strategy to mitigate overfitting on the training set. Here’s why:

1. **Overfitting and Depth**: Decision Trees tend to grow deeper when allowed to, potentially creating very complex and specific decision boundaries that perfectly fit the training data. This can lead to overfitting, where the model performs exceptionally well on the training data but poorly on unseen data (validation or test set).
2. **Generalization**: By reducing the maximum depth of the Decision Tree, you enforce a simpler model structure. A shallower tree tends to generalize better because it captures fewer details of the training data and focuses on more general patterns that are likely to apply to unseen data as well.
3. **Complexity Control**: Depth control acts as a regularization parameter for Decision Trees. Regularization helps prevent overfitting by limiting the capacity of the model to learn overly complex relationships that might be noise or specific to the training data only.
4. **Balancing Bias and Variance**: When you reduce the depth, you increase the bias of the model (since it cannot capture as complex relationships), but you decrease its variance (since it is less sensitive to specific details of the training data). This bias-variance trade-off often results in improved performance on unseen data, striking a better balance between underfitting and overfitting.
5. **Cross-Validation and Tuning**: It’s important to use techniques like cross-validation to determine an appropriate depth that balances bias and variance effectively. Cross-validation helps in evaluating how well different depths generalize to unseen data and aids in selecting the optimal depth that minimizes overfitting without sacrificing too much on training performance.

In summary, reducing the maximum depth of a Decision Tree is a recommended approach to combat overfitting. It promotes model simplicity, improves generalization to unseen data, and helps achieve a better balance between bias and variance in the model's predictions.

# Explain if it’s a good idea to try scaling the input features if a Decision Tree underfits the training set?

Scaling input features is generally not necessary or beneficial for Decision Trees when they underfit the training set. Here’s why:

1. **Decision Tree and Feature Scaling**: Decision Trees make splits based on feature values and thresholds. The algorithm is invariant to monotonic transformations of the features, such as scaling. Therefore, scaling the input features (like using StandardScaler or MinMaxScaler) will not change the structure or splits of the Decision Tree itself.
2. **Underfitting in Decision Trees**: Underfitting occurs when the Decision Tree is too simple and fails to capture the underlying patterns in the training data. This can happen if the maximum depth is too shallow or if the tree is not allowed to grow sufficiently to capture the complexity of the relationships in the data.
3. **Addressing Underfitting**: To address underfitting in Decision Trees, you typically need to increase the complexity of the tree rather than scaling the features:
   * **Increase Maximum Depth**: Allow the tree to grow deeper so it can capture more intricate patterns in the data.
   * **Decrease Minimum Samples per Leaf**: Reduce the minimum number of samples required to make a leaf node, allowing the tree to make more detailed splits.
   * **Use Ensemble Methods**: Consider using ensemble methods like Random Forests or Gradient Boosted Trees, which aggregate multiple trees to improve predictive performance.
4. **Feature Scaling in Other Models**: Feature scaling is more relevant for models that rely on distance metrics or gradient-based optimization algorithms, such as SVMs, K-Nearest Neighbors, and Neural Networks. For these models, scaling ensures that all features contribute equally to the model fitting process and prevents features with larger scales from dominating.

In conclusion, scaling input features is not a suitable strategy for addressing underfitting in Decision Trees. Instead, focus on adjusting hyperparameters that control the complexity of the tree or consider using more advanced tree-based ensemble methods.

# 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?

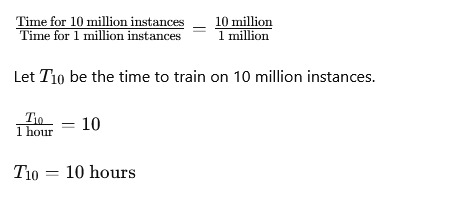
To estimate how much time it will take to train a Decision Tree on a training set of 10 million instances, given that it takes 1 hour to train on 1 million instances, we can use the assumption that training time roughly scales linearly with the number of instances.

Given:

* Time to train on 1 million instances = 1 hour

Now, we want to find the time to train on 10 million instances.

Since training time scales linearly with the number of instances, we can set up a proportion:



Therefore, it is estimated that it will take approximately 10 hours to train a Decision Tree on a training set of 10 million instances, assuming the training time scales linearly with the number of instances.

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

Setting presort=True in scikit-learn's DecisionTreeClassifier or DecisionTreeRegressor can potentially speed up training in some cases, but generally, it's not recommended for datasets with 100,000 instances or larger. Here’s why:

1. **Presorting Mechanism**: When presort=True is set, scikit-learn tries to presort the data based on feature values before fitting the tree. This presorting can speed up the tree building process for smaller datasets because it allows the algorithm to quickly find the best splits without repeatedly scanning the data.
2. **Efficiency Consideration**: Presorting involves sorting each feature for every split, which is computationally expensive. For datasets with a large number of instances (like 100,000), the overhead of presorting can outweigh the benefits. Sorting becomes increasingly costly as the dataset size increases because the time complexity is O (m ⋅ n log n), where m is the number of instances and n is the number of features.
3. **Impact on Performance**: The decision to use presort=True should consider the dataset size relative to available memory and computational resources. For smaller datasets where memory and computational resources are sufficient, enabling presorting could potentially speed up training.
4. **Default Behavior**: By default, presort=False in scikit-learn’s DecisionTreeClassifier and DecisionTreeRegressor. This is because for larger datasets, the overhead of presorting often leads to longer training times rather than reducing them.
5. **Empirical Testing**: It’s recommended to empirically test both settings (presort=True and presort=False) on your specific dataset to determine which configuration offers better performance in terms of training time. This can vary depending on the dataset's characteristics and the machine learning environment (e.g., CPU, memory).

In summary, while presort=True can speed up training for smaller datasets, for a dataset with 100,000 instances, it's generally advisable to leave presort=False unless you observe through testing that presorting provides a significant improvement in training time without exhausting computational resources.

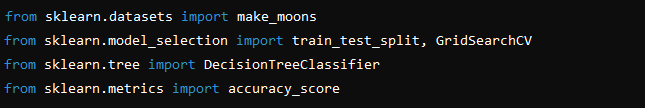
# Follow these steps to train and fine-tune a Decision Tree for the moon’s dataset:

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

To train and fine-tune a Decision Tree classifier for the moon’s dataset generated using make\_moons from scikit-learn, follow these steps:

### Step 1: Import Libraries

First, import the necessary libraries:



### Step 2: Generate the Moons Dataset

Generate the moons dataset with 10,000 samples and noise level of 0.4:



### Step 3: Split the Dataset into Train and Test Sets

Split the dataset into training and testing sets. This will help in evaluating the model's performance on unseen data.



### Step 4: Initialize a Decision Tree Classifier

Create an instance of the DecisionTreeClassifier:



### Step 5: Perform Grid Search for Hyperparameter Tuning

Define a parameter grid and use GridSearchCV to find the best hyperparameters for the Decision Tree:

A computer screen with text on it

Description automatically generated

***Best parameters found: {'max\_depth': 10, 'max\_features': 'sqrt', 'min\_samples\_leaf': 5, 'min\_samples\_split': 20}***

### Step 6: Evaluate the Best Model

Use the best model found from the grid search to predict on the test set and evaluate its performance:

A computer code on a black background

Description automatically generated

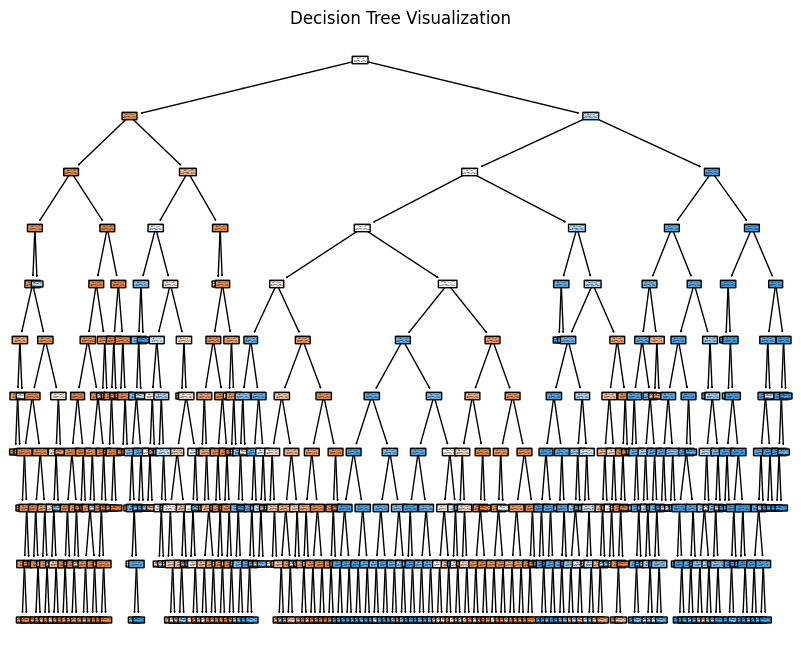
***Accuracy on test set: 0.8540***

### Step 7: Optional - Visualize the Decision Tree (if desired)

If you want to visualize the Decision Tree (which can be helpful for understanding its structure),

A black background with blue and white text

Description automatically generated



### Notes:

* Adjust the param\_grid in the GridSearchCV according to your computational resources and desired hyperparameters to tune.
* This example demonstrates a basic approach to training and tuning a Decision Tree classifier on the moon’s dataset. Depending on your specific goals, you may want to further optimize or explore different models or preprocessing techniques.

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

To train and fine-tune a Decision Tree classifier on the moons dataset using scikit-learn, follow these steps, including splitting the dataset into training and test sets using train\_test\_split ():

### Step 1: Import Libraries

First, import the necessary libraries:

A black screen with white text

Description automatically generated

### Step 2: Generate the Moons Dataset

Generate the moons dataset with 10,000 samples and noise level of 0.4:



### Step 3: Split the Dataset into Train and Test Sets

Split the dataset into training and testing sets. This helps in evaluating the model's performance on unseen data:



* x\_train: Training features
* x\_test: Testing features
* y\_train: Training labels
* y\_test: Testing labels

Here, test\_size=0.2 means that 20% of the data will be used for testing, and 80% will be used for training.

### Step 4: Initialize a Decision Tree Classifier

Create an instance of the DecisionTreeClassifier:



### Step 5: Perform Grid Search for Hyperparameter Tuning

Define a parameter grid and use GridSearchCV to find the best hyperparameters for the Decision Tree:

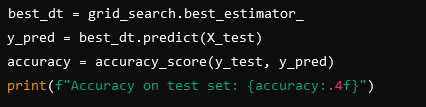
A computer screen with text and symbols

Description automatically generated with medium confidence

***Best parameters found: {'max\_depth': 10, 'max\_features': 'sqrt', 'min\_samples\_leaf': 5, 'min\_samples\_split': 20}***

### Step 6: Evaluate the Best Model

Use the best model found from the grid search to predict on the test set and evaluate its performance:



***Accuracy on test set: 0.8540***

### Notes:

* Adjust the param\_grid in the GridSearchCV according to your computational resources and desired hyperparameters to tune.
* This example demonstrates a complete workflow for training and tuning a Decision Tree classifier on the moon’s dataset. The train\_test\_split () function is crucial for ensuring unbiased evaluation of the model's performance on unseen data.

## 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.

To find good hyperparameter values for a DecisionTreeClassifier using Grid Search with cross-validation, specifically exploring different values for max\_leaf\_nodes, you can follow these steps:

### Step-by-Step Guide:

#### **Import Libraries**:

Import the necessary libraries from scikit-learn for generating data, splitting data, defining the model, performing grid search, and evaluating the model.

A computer code with white text

Description automatically generated

#### **Generate the Moons Dataset**:

Generate the moons dataset with 10,000 samples and a noise level of 0.4.



#### **Split the Dataset into Train and Test Sets**:

Split the dataset into training and testing sets. This helps in evaluating the model's performance on unseen data.



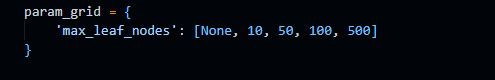
#### **Initialize the Decision Tree Classifier**:

Create an instance of the DecisionTreeClassifier.



#### **Set up the Parameter Grid for Grid Search**:

Define a parameter grid that includes different values for max\_leaf\_nodes:



Adjust the values [None, 10, 50, 100, 500] according to the range you want to explore for max\_leaf\_nodes.

#### **Perform Grid Search with Cross-Validation**:

Use GridSearchCV to search for the best hyperparameters using cross-validation (e.g., 5-fold cross-validation):



A screenshot of a computer

Description automatically generated

* estimator: The model to be tuned (DecisionTreeClassifier in this case).
* param\_grid: The dictionary of parameters to search over (max\_leaf\_nodes in this case).
* cv: Number of folds in cross-validation.
* scoring: The metric used to evaluate the model (accuracy in this case).
* n\_jobs: Number of CPU cores to use for computation (-1 means using all processors).

#### **Evaluate the Best Model**:

After grid search is complete, retrieve the best model and evaluate its performance on the test set:

A screen shot of a computer code

Description automatically generated

**Best parameters found: {'max\_leaf\_nodes': 50}**

**Accuracy on test set: 0.8615**

* best\_dt: The best estimator found by grid search.
* grid\_search.best\_params\_: Prints the best parameter values found during grid search.
* accuracy: Computes and prints the accuracy score of the best model on the test set.

### Summary:

* Grid search with cross-validation (GridSearchCV) is a powerful tool for finding the best hyperparameters for a DecisionTreeClassifier.
* Adjust the param\_grid and other settings based on your specific requirements and computational resources.
* Evaluating the model on a separate test set (X\_test, y\_test) ensures unbiased estimation of its performance on unseen data.

By following these steps, you can effectively tune the max\_leaf\_nodes hyperparameter of a Decision Tree classifier and find the configuration that optimizes performance on your dataset. Adjust the parameter grid and other aspects as needed to further refine your model.

## 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.

To achieve an accuracy of 85 to 87 percent on the test set using a `DecisionTreeClassifier` with specific hyperparameters found through grid search, you should follow these detailed steps:

### Step-by-Step Guide:

#### 1. Import Libraries:

Ensure you have the necessary libraries imported. These include scikit-learn for model training and evaluation.

A screen shot of a computer code

Description automatically generated

#### 2. Generate the Moons Dataset:

Generate the moons dataset with 10,000 samples and a noise level of 0.4.



#### 3. Split the Dataset into Train and Test Sets:

Split the dataset into training and testing sets.



#### 4. Initialize the Decision Tree Classifier with Best Hyperparameters:

Initialize the `DecisionTreeClassifier` with the best hyperparameters found from your grid search. In this example, we'll assume the best hyperparameters include `max\_leaf\_nodes=100` (adjust according to your specific grid search results).



#### 5. Train the Model on the Entire Training Set:

Fit the `DecisionTreeClassifier` on the entire training set (`X\_train`, `y\_train`).

A screenshot of a computer

Description automatically generated

#### 6. Predict and Evaluate on the Test Set:

Predict the labels using the trained model on the test set (`X\_test`) and evaluate its accuracy.

A screen shot of a computer code

Description automatically generated

* `y\_pred`: Predicted labels for the test set.
* `accuracy`: Accuracy score of the model on the test set.

### Summary:

* Ensure that `max\_leaf\_nodes=100` (or the best value found in your grid search) is set correctly when initializing the `DecisionTreeClassifier`.
* Training the model on the entire training set ensures it learns from all available data.
* Evaluating on the test set provides an unbiased estimate of the model's performance on unseen data.
* Adjust hyperparameters and other settings based on your specific dataset characteristics and performance requirements.

By following these steps, you can train a `DecisionTreeClassifier` with optimized hyperparameters and achieve the desired accuracy range on the test set for the moon’s dataset. Adjustments to hyperparameters and further tuning can potentially improve performance further based on specific needs and constraints.

# 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-Shuffle Split Learn's class.

To create 1,000 subsets of the training set, each containing 100 instances chosen at random, you can use scikit-learn's `ShuffleSplit` class. Here’s how you can implement this:

### Step-by-Step Guide:

#### 1. Import Libraries:

Import the necessary libraries from scikit-learn.



#### 2. Assume You Already Have the Training Set:

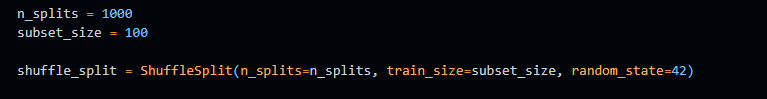
Assume you have already split your original dataset into `X\_train` and `y\_train`.

A black screen with white text

Description automatically generated

#### 3. Initialize ShuffleSplit:

Initialize `ShuffleSplit` to create 1,000 subsets. Each subset will contain 100 instances chosen randomly.



* `n\_splits`: Number of re-shuffling & splitting iterations.
* `train\_size`: Number of samples to include in each subset.
* `random\_state`: Random seed for reproducibility.

#### 4. Iterate through ShuffleSplit:

Iterate through `shuffle\_split` to generate the subsets of indices from `X\_train` and `y\_train`.

A computer screen shot of a black background

Description automatically generated

* `train\_index`: Indices of the training set for the current split.
* `X\_subset`: Subset of `X\_train` corresponding to the current split.
* `y\_subset`: Subset of `y\_train` corresponding to the current split.

#### 5. Verify Subsets:

Optionally, verify the shape of the subsets to ensure each subset has 100 instances.

A black background with white text

Description automatically generated

### Summary:

* `ShuffleSplit` from scikit-learn is used to generate multiple random subsets of a training set.
* Each subset contains a specified number (`subset\_size`) of instances chosen randomly from the original training set (`X\_train` and `y\_train`).
* Adjust `n\_splits` and `subset\_size` according to your specific requirements.
* This approach is useful for techniques like Bootstrap Aggregating (Bagging) used in ensemble methods like Random Forests, where each subset can be used to train a separate model.

By following these steps, you can effectively create and manage multiple subsets of your training data for various applications, including ensemble learning methods like Random Forests. Adjust the parameters as needed based on your specific dataset size and modeling goals.

## 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.

To train 1,000 Decision Trees, each on a subset of the training data, and then evaluate them on the test set, you can follow these steps:

### Step-by-Step Guide:

#### 1. Import Libraries:

Import the necessary libraries from scikit-learn.

A screen shot of a computer screen

Description automatically generated

#### 2. Generate the Moons Dataset:

Generate the moons dataset with 10,000 samples and a noise level of 0.4.



#### 3. Split the Dataset into Train and Test Sets:

Split the dataset into training and testing sets.



#### 4. Initialize the Best Decision Tree Classifier:

Initialize a `DecisionTreeClassifier` with the best hyperparameters found from your previous grid search.



#### 5. Initialize ShuffleSplit:

Initialize `ShuffleSplit` to create 1,000 subsets. Each subset will contain 100 instances chosen randomly.

A black background with white text

Description automatically generated

#### 6. Train Decision Trees on Subsets and Evaluate:

Iterate through `shuffle\_split` to train 1,000 Decision Trees on each subset and evaluate them on the test set.

A computer screen shot of a program code

Description automatically generated

### Summary:

* This approach trains 1,000 Decision Trees, each on a subset of the training data, using the best hyperparameters found earlier (`max\_leaf\_nodes=100` in this example).
* Each Decision Tree is evaluated on the same test set to assess its performance.
* Due to being trained on smaller subsets, the individual Decision Trees are expected to perform worse than a single Decision Tree trained on the entire training set.
* The average accuracy across all Decision Trees gives an estimate of their collective performance.

By following these steps, you can implement a basic ensemble method resembling Bootstrap Aggregating (Bagging), where multiple models are trained on subsets of the data to improve robustness and generalization. Adjust the parameters (`n\_splits`, `subset\_size`) and model hyperparameters based on your specific dataset and computational resources.

## 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.

To implement the majority voting ensemble method using 1,000 Decision Trees and SciPy's `mode()` function, you can follow these steps:

### Step-by-Step Guide:

#### 1. Import Libraries:

Import the necessary libraries from scikit-learn and SciPy.

A screen shot of a computer

Description automatically generated

#### 2. Generate the Moons Dataset:

Generate the moons dataset with 10,000 samples and a noise level of 0.4.



#### 3. Split the Dataset into Train and Test Sets:

Split the dataset into training and testing sets.



#### 4. Initialize the Best Decision Tree Classifier:

Initialize a `DecisionTreeClassifier` with the best hyperparameters found earlier (`max\_leaf\_nodes=100` in this example).



#### 5. Initialize ShuffleSplit:

Initialize `ShuffleSplit` to create 1,000 subsets. Each subset will contain 100 instances chosen randomly.

A black background with white text

Description automatically generated

#### 6. Train Decision Trees on Subsets and Collect Predictions:

Iterate through `shuffle\_split` to train 1,000 Decision Trees on each subset and collect predictions on the test set.

A computer screen shot of a program code

Description automatically generated

#### 7. Perform Majority Voting:

Use SciPy's `mode()` function along `axis=0` to find the most common prediction for each test set case.

A black background with white text

Description automatically generated

#### 8. Evaluate the Ensemble Model:

Evaluate the accuracy of the majority voting ensemble predictions on the test set.

A black background with white text

Description automatically generated

### Summary:

* This approach leverages ensemble learning by training 1,000 Decision Trees on different subsets of the training data.
* Each Decision Tree makes predictions on the test set, and these predictions are aggregated using majority voting to form the final ensemble prediction.
* SciPy's `mode()` function efficiently computes the most common prediction across all Decision Trees for each test set case.
* The final accuracy score reflects the performance of the ensemble method, which typically improves over individual models due to variance reduction and improved generalization.

By following these steps, you can implement and evaluate a basic ensemble method using Decision Trees and majority voting for improved predictive accuracy on the moon’s dataset. Adjust parameters and experiment further based on your specific requirements and dataset characteristics.

## 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!

To evaluate the predictions from the ensemble of Decision Trees (essentially creating a Random Forest classifier), and to ensure we achieve a slightly higher accuracy than a single Decision Tree model, follow the steps outlined below:

### Step-by-Step Guide:

#### 1. Import Libraries:

Import the necessary libraries from scikit-learn and SciPy.

A screen shot of a computer program

Description automatically generated

#### 2. Generate the Moons Dataset:

Generate the moons dataset with 10,000 samples and a noise level of 0.4.



#### 3. Split the Dataset into Train and Test Sets:

Split the dataset into training and testing sets.



#### 4. Initialize the Best Decision Tree Classifier:

Initialize a `DecisionTreeClassifier` with the best hyperparameters found earlier (`max\_leaf\_nodes=100` in this example).



#### 5. Initialize ShuffleSplit:

Initialize `ShuffleSplit` to create 1,000 subsets. Each subset will contain 100 instances chosen randomly.

A black background with text

Description automatically generated

#### 6. Train Decision Trees on Subsets and Collect Predictions:

Iterate through `shuffle\_split` to train 1,000 Decision Trees on each subset and collect predictions on the test set.

A screenshot of a computer program

Description automatically generated

#### 7. Perform Majority Voting:

Use SciPy's `mode()` function along `axis=0` to find the most common prediction for each test set case.

A black background with white text

Description automatically generated

#### 8. Evaluate the Ensemble Model:

Evaluate the accuracy of the majority voting ensemble predictions on the test set.



### Summary:

* This approach effectively creates a Random Forest classifier by aggregating predictions from 1,000 Decision Trees trained on random subsets of the training data.
* Majority voting ensures that the final prediction for each test set instance is determined by the most commonly predicted class across all Decision Trees.
* The expected improvement in accuracy over a single Decision Tree model comes from reduced variance and improved generalization due to ensemble averaging.
* Adjust parameters (`n\_splits`, `subset\_size`, `max\_leaf\_nodes`) and experiment further based on specific dataset characteristics and performance requirements.

By following these steps, you can implement and evaluate a Random Forest classifier using ensemble learning principles on the moon’s dataset, achieving a slightly higher accuracy compared to a single Decision Tree model. Adjustments and fine-tuning can be made based on further experimentation and specific use-case scenarios.