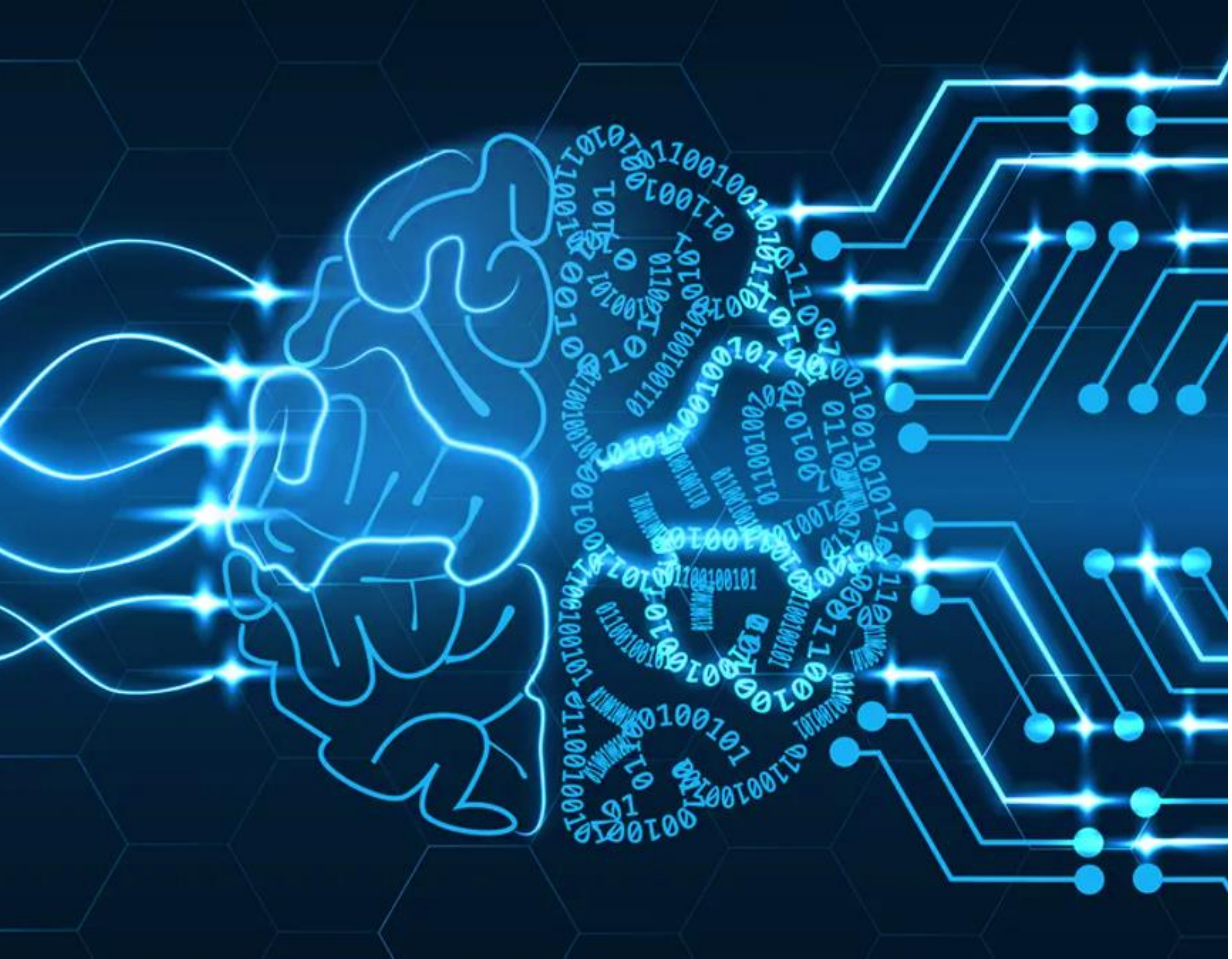


درخت تصمیم در یادگیری ماشین

فروردین ۱۴۰۲

دانشکده مهندسی صنایع

دانشگاه صنعتی شریف

دکتر مهدی شریف‌زاده

آرش ملک پور

سعیدرضا زواشکیانی

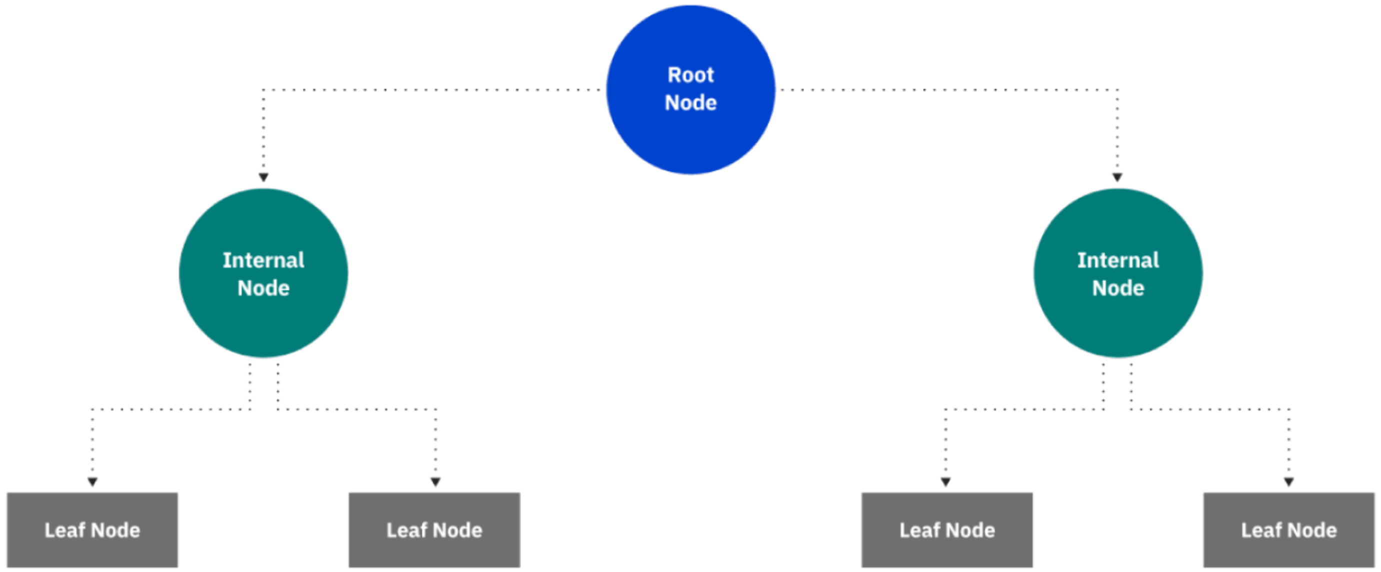
بسم الله الرحمن الرحیم

Decision Tree

Decision Trees can perform both classification and regression tasks, and even multioutput tasks. They are very powerful algorithms, capable of fitting complex datasets.

Decision Trees are also the fundamental components of Random Forests, which are among the most powerful Machine Learning algorithms available today.

For example, in Chapter 2 you trained a *DecisionTreeRegressor* model on the California housing dataset, fitting it perfectly (actually, overfitting it).



IRIS Dataset

The *Iris* flower data set or Fisher's *Iris* data set is a multivariate dataset collected the data to quantify the morphologic variation of *Iris* flowers of three related species. The data set consists of 50 samples from each of three species of *Iris* (*Iris setosa*, *Iris virginica* and *Iris versicolor, respectively from left to right*).
Four features were measured from each sample: the length and the width of the sepals and petals, in centimeters.

Training and Visualizing a Decision Tree

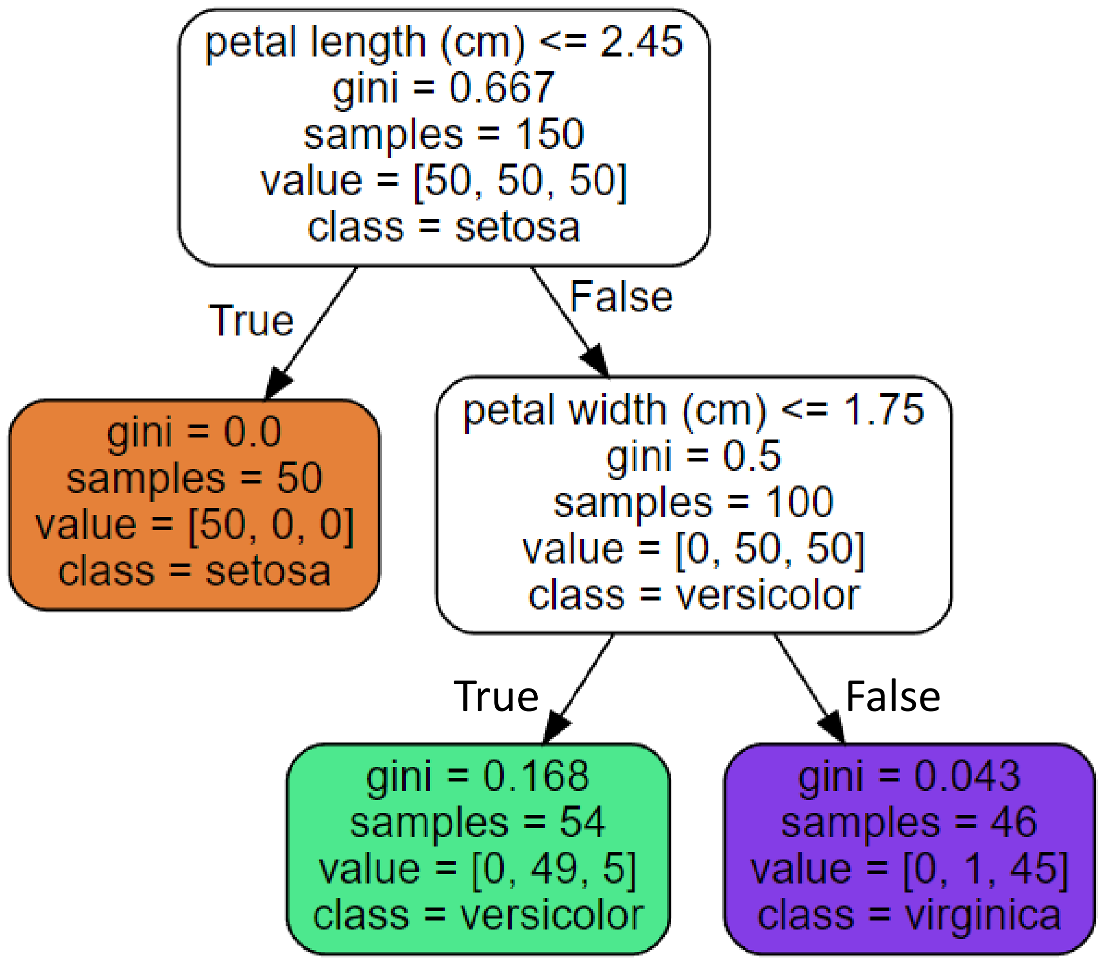
To understand Decision Trees, let’s just build one and take a look at how it makes predictions. The following code trains a DecisionTreeClassifier on the iris dataset:

```
from sklearn.datasets import load_iris
from sklearn.tree import DecisionTreeClassifier

iris = load_iris()
X = iris.data[:, 2:] # petal length and width
y = iris.target

tree_clf = DecisionTreeClassifier(max_depth=2)
tree_clf.fit(X, y)
```

You can visualize the trained Decision Tree by first using the *export_graphviz()* method to output a graph definition file called *iris_tree.dot*

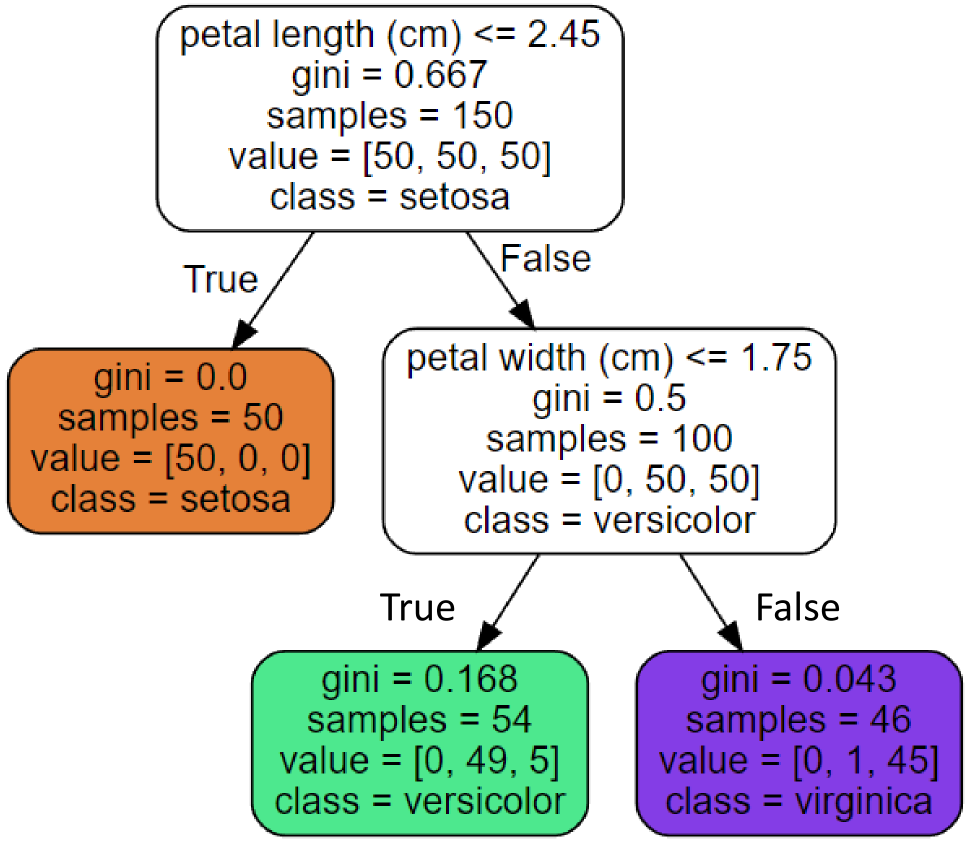
```
from sklearn.tree import export_graphviz

export_graphviz(
        tree_clf,
        out_file=image_path("iris_tree.dot"),
        feature_names=iris.feature_names[2:],
        class_names=iris.target_names,
        rounded=True,
        filled=True
    )
```

Training and Visualizing a Decision Tree



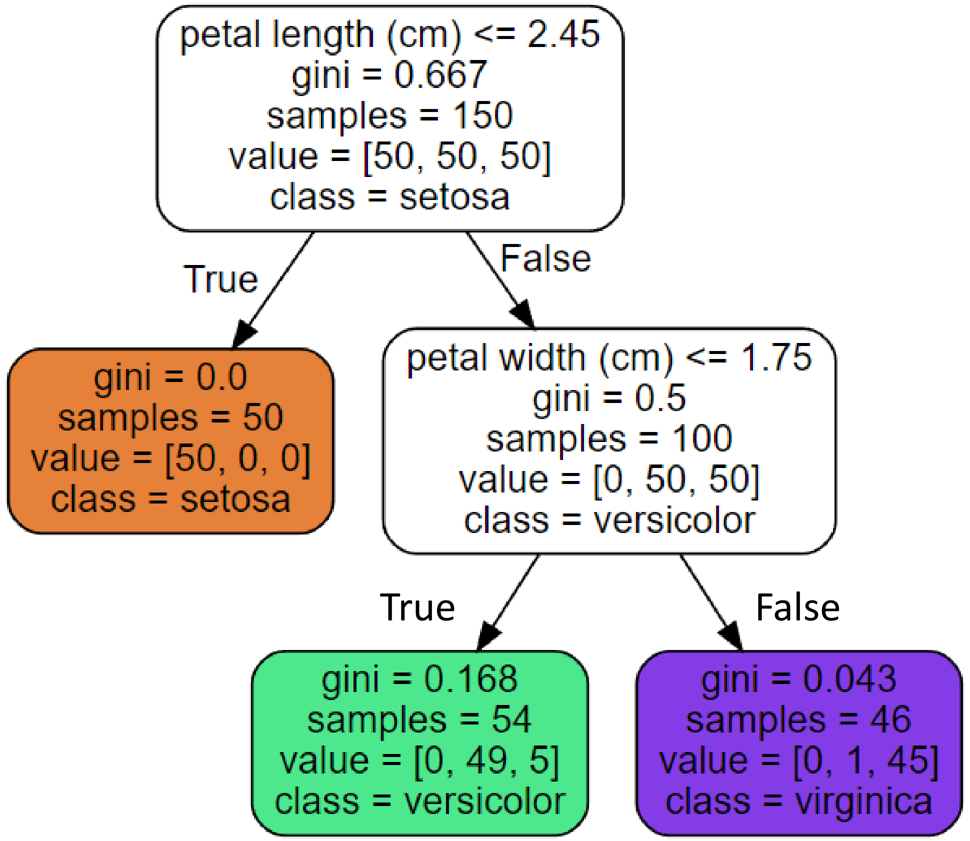
Making Predictions

Suppose you find an iris flower and you want to classify it. You start at the root node (depth 0, at the top): this node asks whether the flower's petal length is smaller than 2.45 cm. If it is, then you move down to the root's left child node (depth 1, left). In this case, it is a leaf node (i.e., it does not have any children nodes), so it does not ask any questions: you can simply look at the predicted class for that node and the Decision Tree predicts that your flower is an Iris-Setosa (class=setosa).



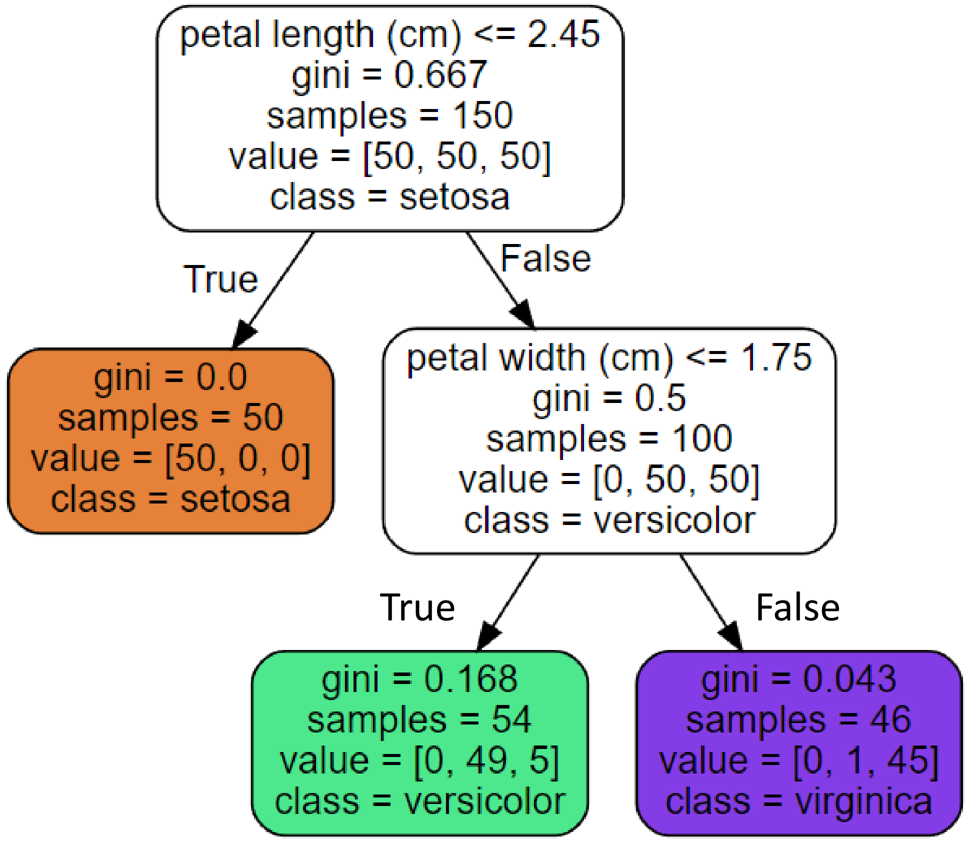
Making Predictions

Now suppose you find another flower, but this time the petal length is greater than 2.45 cm. You must move down to the root’s right child node (depth 1, right), which is not a leaf node, so it asks another question: is the petal width smaller than 1.75 cm? If it is, then your flower is most likely an Iris-Versicolor (depth 2, left). If not, it is likely an Iris-Virginica (depth 2, right). It’s really that simple.



Making Predictions

A node's ***samples*** attribute counts how many training instances it applies to. For example, 100 training instances have a petal length greater than 2.45 cm (depth 1, right), among which 54 have a petal width smaller than 1.75 cm (depth 2, left). A node's **value** attribute tells you how many training instances of each class this node applies to: for example, the bottom-right node applies to 0 Iris-Setosa, 1 Iris-Versicolor, and 45 Iris-Virginica.



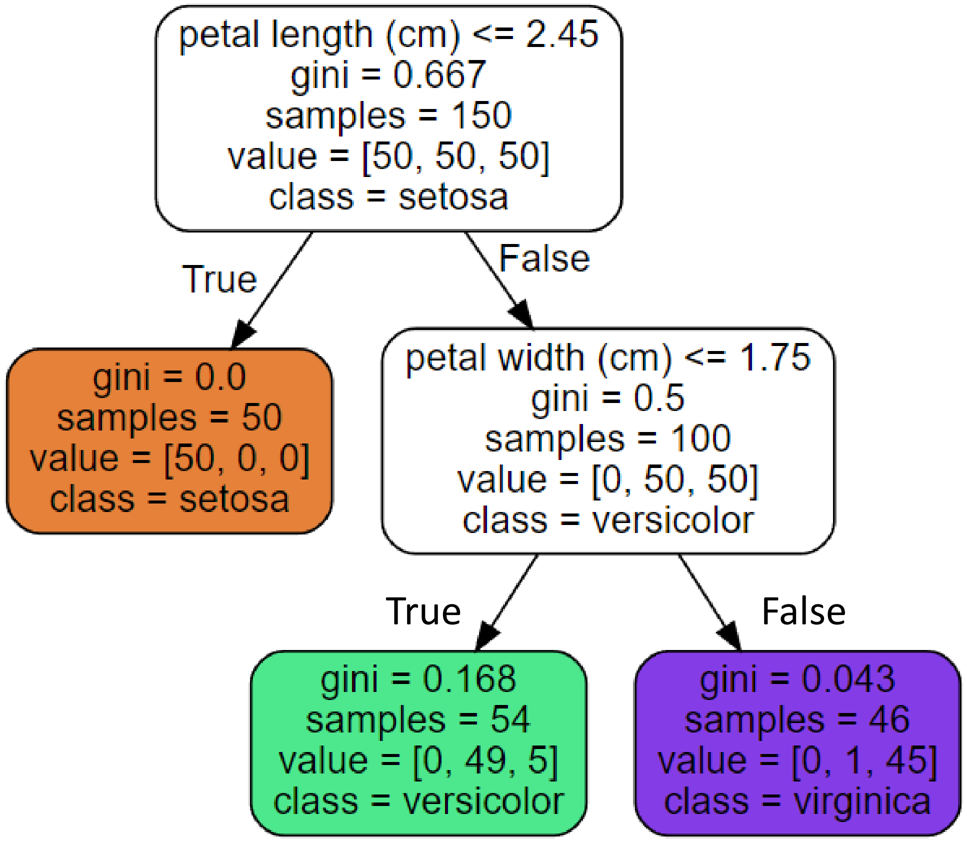
Making Predictions

Finally, a node's **gini** attribute measures its impurity: a node is "pure" (**gini**=0) if all training instances it applies to belong to the same class. For example, since the depth-1 left node applies only to Iris-Setosa training instances, it is pure and its **gini** score is 0. Gini Equation shows how the training algorithm computes the **gini** score Gi of the ith node. For example, the depth-2 left node has a **gini** score equal to

$$1 - \left(\frac{0}{54}\right)^2 - \left(\frac{49}{54}\right)^2 - \left(\frac{5}{54}\right)^2 \approx 0.168$$

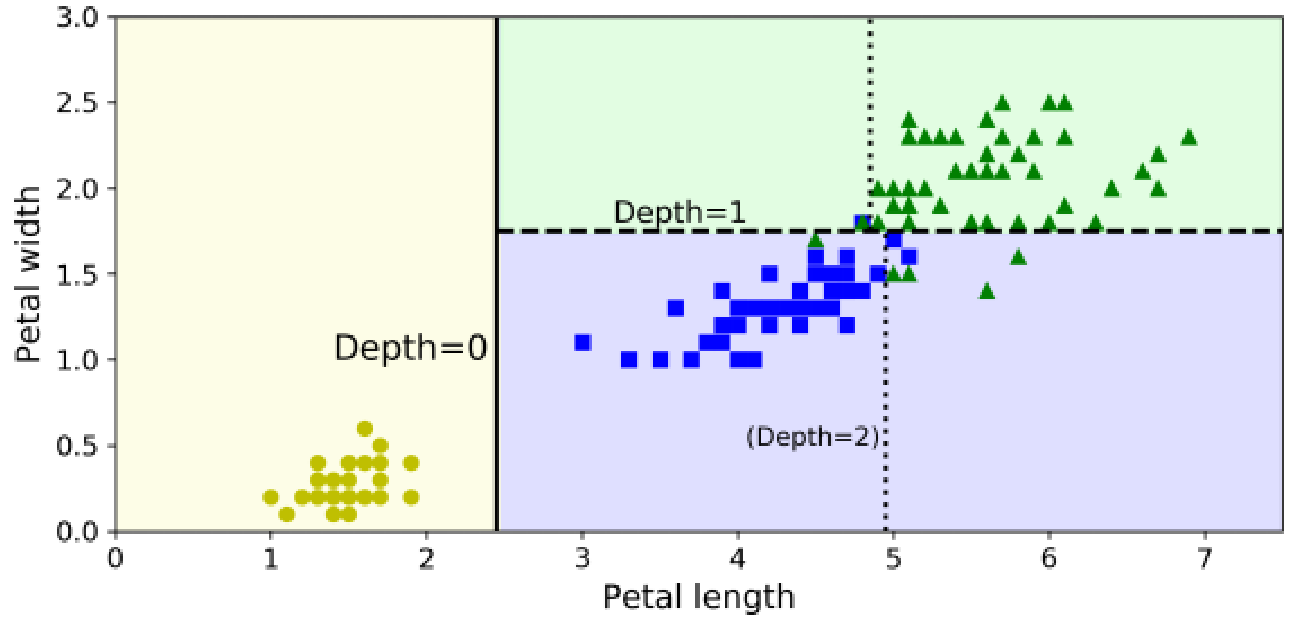
$$G_i = 1 - \sum_{k=1}^{n} {p_{i,k}}^2$$

- $p_{i,k}$ is the ratio of class k instances among the training instances in the i^{th} node.



Decision Tree’s decision boundaries

Figure below shows this Decision Tree’s decision boundaries. The thick vertical line represents the decision boundary of the root node (depth 0): petal length = 2.45 cm. Since the left area is pure (only Iris-Setosa), it cannot be split any further. However, the right area is impure, so the depth-1 right node splits it at petal width = 1.75 cm (represented by the dashed line). Since **max_depth** was set to 2, the Decision Tree stops right there. However, if you set **max_depth** to 3, then the two depth-2 nodes would each add another decision boundary (represented by the dotted lines).



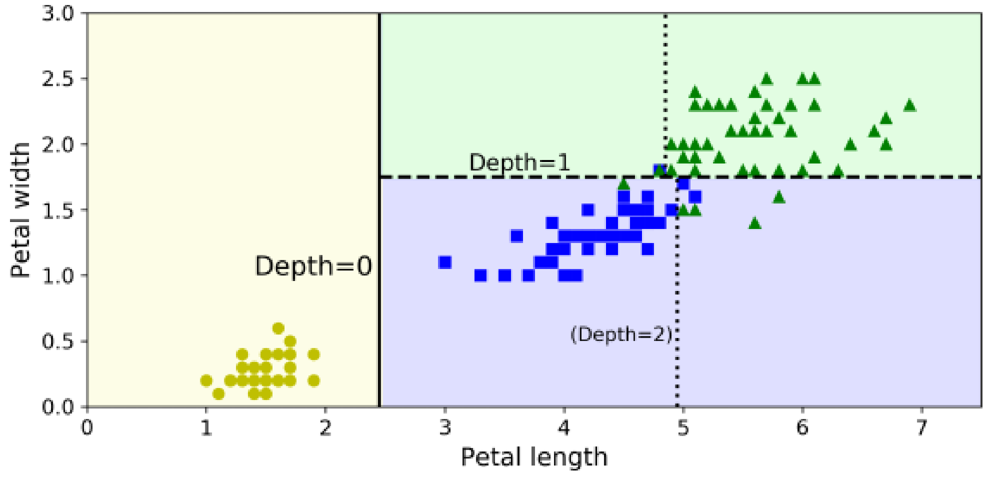
Estimating Class Probabilities

A Decision Tree can also estimate the probability that an instance belongs to a particular class k: first it traverses the tree to find the leaf node for this instance, and then it returns the ratio of training instances of class k in this node. For example, suppose you have found a flower whose petals are 5 cm long and 1.5 cm wide. The corresponding leaf node is the depth-2 left node, so the Decision Tree should output the following probabilities: 0% for Iris-Setosa (0/54), 90.7% for Iris-Versicolor (49/54), and 9.3% for Iris-Virginica (5/54). And of course, if you ask it to predict the class, it should output Iris-Versicolor (class 1) since it has the highest probability.

Estimating Class Probabilities

```
In [5]: tree_clf.predict_proba([[5, 1.5]])

Out[5]: array([[0.        , 0.90740741, 0.09259259]])

In [6]: tree_clf.predict([[5, 1.5]])

Out[6]: array([1])
```

Perfect! Notice that the estimated probabilities would be identical anywhere else in the bottom-right rectangle of Figure 6-2—for example, if the petals were 6 cm long and 1.5 cm wide (even though it seems obvious that it would most likely be an *Iris virginica* in this case)



The CART Training Algorithm

Scikit-Learn uses the Classification And Regression Tree (CART) algorithm to train Decision Trees (also called "growing" trees). The idea is really quite simple: the algorithm first splits the training set in two subsets using a single feature k and a threshold t_k (e.g., "petal length ≤ 2.45 cm"). How does it choose k and t_k? It searches for the pair (k, t_k) that produces the purest subsets (weighted by their size). The cost function that the algorithm tries to minimize is given in next slide.

Equation 6-2. CART cost function for classification

$$J(k, t_k) = \frac{m_{\text{left}}}{m} G_{\text{left}} + \frac{m_{\text{right}}}{m} G_{\text{right}}$$

$$\text{where} \begin{cases} G_{\text{left/right}} & \text{measures the impurity of the left/right subset,} \\ m_{\text{left/right}} & \text{is the number of instances in the left/right subset.} \end{cases}$$

The CART Training Algorithm

Equation 6-2. CART cost function for classification

$$J(k, t_k) = \frac{m_{\text{left}}}{m} G_{\text{left}} + \frac{m_{\text{right}}}{m} G_{\text{right}}$$

$$\text{where} \begin{cases} G_{\text{left/right}} \text{ measures the impurity of the left/right subset,} \\ m_{\text{left/right}} \text{ is the number of instances in the left/right subset.} \end{cases}$$

Once the CART algorithm has successfully split the training set in two, it splits the subsets using the same logic, then the sub-subsets, and so on, recursively. It stops recursing once it reaches the maximum depth (defined by the **max_depth** hyperparameter), or if it cannot find a split that will reduce impurity

CART, A Greedy Algorithm

As you can see, the CART algorithm is a **greedy algorithm**: it greedily searches for an optimum split at the top level, then repeats the process at each level. It does not check whether or not the split will lead to the lowest possible impurity several levels down. A greedy algorithm often produces a reasonably good solution, but it is not guaranteed to be the optimal solution. Unfortunately, finding the optimal tree is known to be an **NP-Complete problem**: it requires **O(exp(m))** time, making the problem intractable even for fairly small training sets. This is why we must settle for a “reasonably good” solution.

Note: A greedy algorithm is an approach for solving a problem by selecting the best option available at the moment. It doesn't worry whether the current best result will bring the overall optimal result. The algorithm never reverses the earlier decision even if the choice is wrong.

Gini Impurity or Entropy?

By default, the Gini impurity measure is used, but you can select the entropy impurity measure instead by setting the criterion hyperparameter to "entropy". The concept of entropy originated in thermodynamics as a measure of molecular disorder: entropy approaches zero when molecules are still and well ordered. It later spread to a wide variety of domains, including Shannon’s information theory, where it measures the average information content of a message: entropy is zero when all messages are identical. In Machine Learning, it is frequently used as an impurity measure: a set’s entropy is zero when it contains instances of only one class.

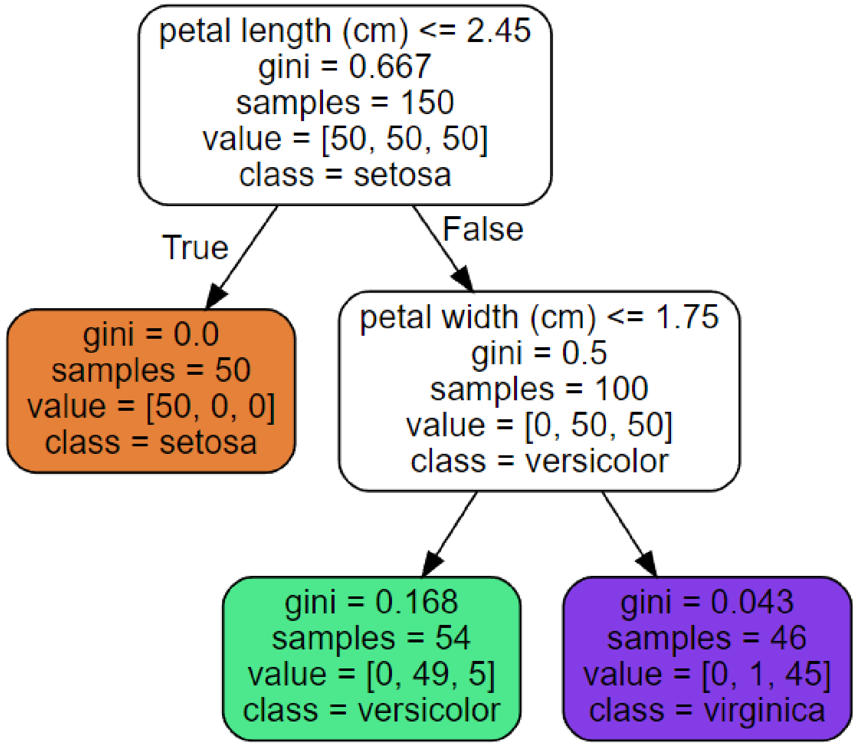
Gini Impurity or Entropy?

Below equation shows the definition of the entropy of the i^{th} node.

$$H_i = -\sum_{\substack{k=1 \\ p_{i,k} \neq 0}}^{n} p_{i,k} \log_2 \left(p_{i,k}\right)$$

For example, the depth-2 left node in Figure has an entropy equal to:

$$-\frac{49}{54} \log_2\left(\frac{49}{54}\right) - \frac{5}{54} \log_2\left(\frac{5}{54}\right) \approx 0.445.$$



Gini Impurity or Entropy?

So should you use Gini impurity or entropy? The truth is, most of the time it does not make a big difference: they lead to similar trees. Gini impurity is slightly faster to compute, so it is a good default. However, when they differ, Gini impurity tends to isolate the most frequent class in its own branch of the tree, while entropy tends to produce slightly more balanced trees.

Regularization Hyperparameters

Decision Trees make very few assumptions about the training data (as opposed to linear models, which obviously assume that the data is linear, for example). If left unconstrained, the tree structure will adapt itself to the training data, fitting it very closely, and most likely overfitting it. Such a model is often called a **nonparametric model**, not because it does not have any parameters (it often has a lot) but because the number of parameters is not determined prior to training, so the model structure is free to stick closely to the data. In contrast, a parametric model such as a linear model has a predetermined number of parameters, so its degree of freedom is limited, reducing the risk of overfitting (but increasing the risk of underfitting).

Regularization Hyperparameters

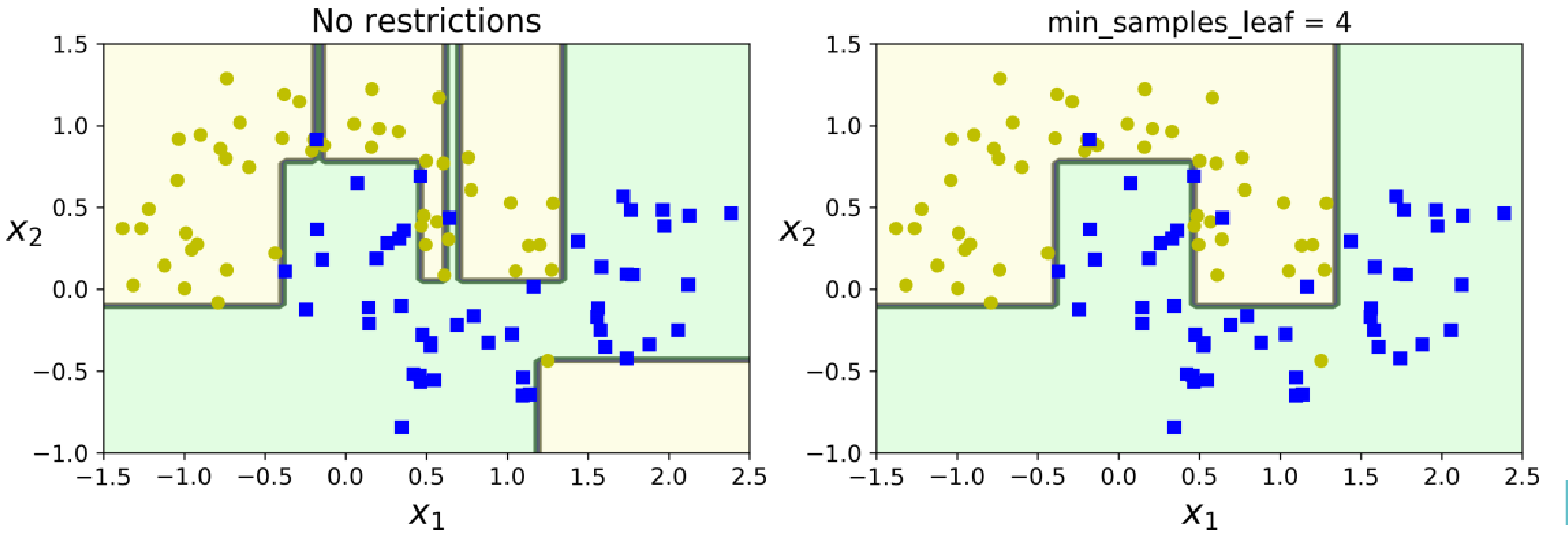
To avoid overfitting the training data, you need to restrict the Decision Tree's freedom during training. As you know by now, this is called regularization. The regularization hyperparameters depend on the algorithm used, but generally you can at least restrict the maximum depth of the Decision Tree. In Scikit-Learn, this is controlled by the **max_depth** hyperparameter (the default value is **None**, which means unlimited). Reducing **max_depth** will regularize the model and thus reduce the risk of overfitting.

Regularization Hyperparameters

The **DecisionTreeClassifier** class has a few other parameters that similarly restrict the shape of the Decision Tree: **min_samples_split** (the minimum number of samples a node must have before it can be split), **min_samples_leaf** (the minimum number of samples a leaf node must have), **min_weight_fraction_leaf** (same as **min_samples_leaf** but expressed as a fraction of the total number of weighted instances), **max_leaf_nodes** (maximum number of leaf nodes), and **max_features** (maximum number of features that are evaluated for splitting at each node). Increasing **min_*** hyperparameters or reducing **max_*** hyperparameters will regularize the model.

Regularization Hyperparameters

Figure below shows two Decision Trees trained on the moons dataset. On the left, the Decision Tree is trained with the default hyperparameters (i.e., no restrictions), and on the right the Decision Tree is trained with **min_samples_leaf=4**. It is quite obvious that the model on the left is overfitting, and the model on the right will probably generalize better.

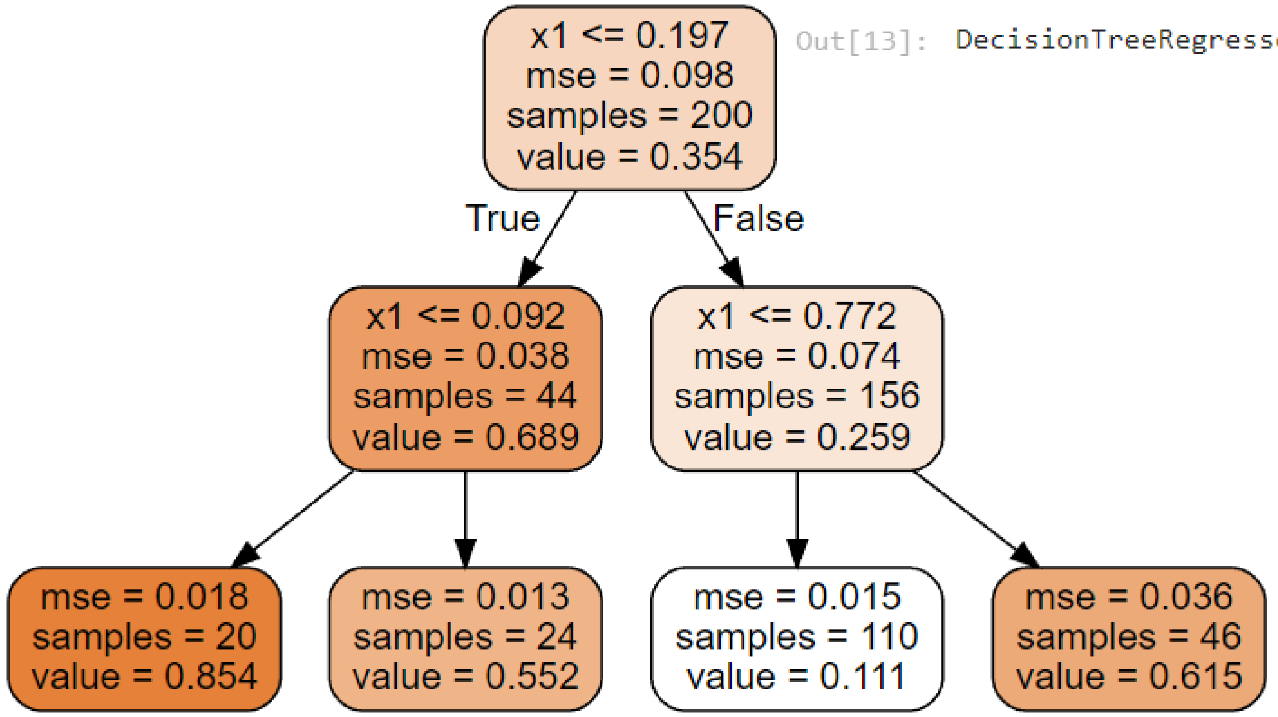


Regression

Decision Trees are also capable of performing regression tasks. Let’s build a regression tree using Scikit-Learn’s **DecisionTreeRegressor** class, training it on a noisy quadratic dataset with **max_depth=2**:

```
In [13]: from sklearn.tree import DecisionTreeRegressor

         tree_reg = DecisionTreeRegressor(max_depth=2, random_state=42)
         tree_reg.fit(X, y)

Out[13]: DecisionTreeRegressor(max_depth=2, random_state=42)
```

Out[16]:

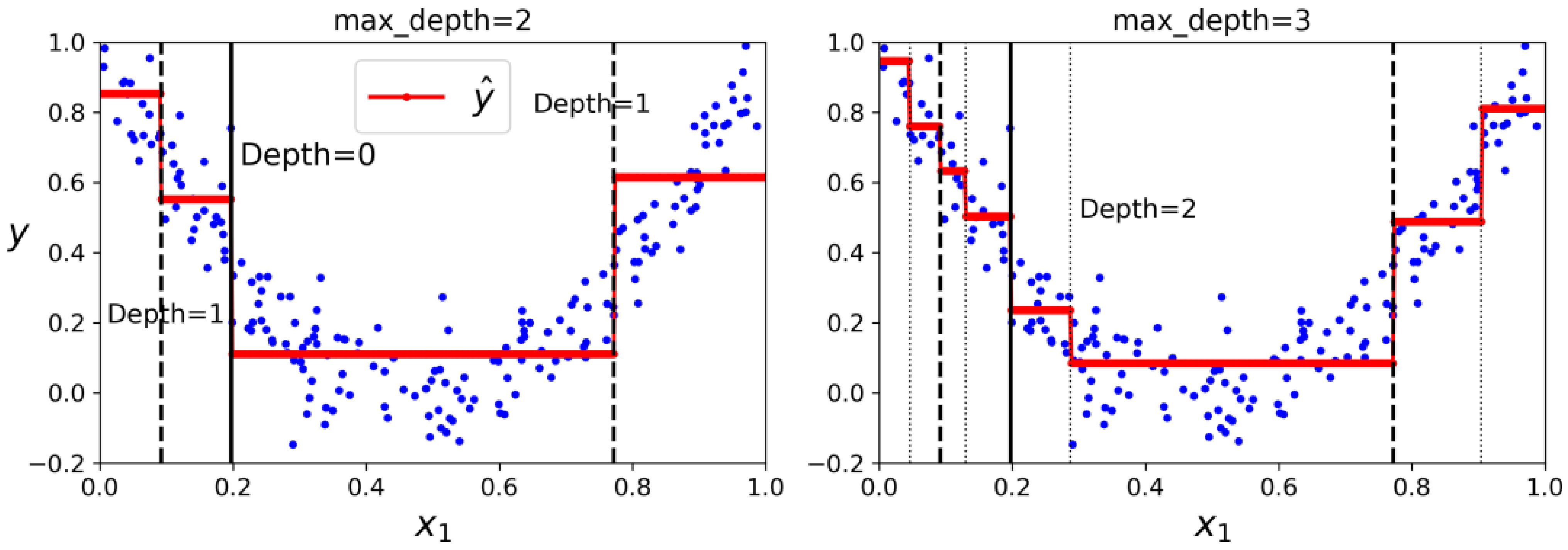


Regression

This tree looks very similar to the classification tree you built earlier. The main difference is that instead of predicting a class in each node, it predicts a value. For example, suppose you want to make a prediction for a new instance with $x_1 = 0.6$. You traverse the tree starting at the root, and you eventually reach the leaf node that predicts value=0.1106. This prediction is simply the average target value of the 110 training instances associated to this leaf node. This prediction results in a Mean Squared Error (MSE) equal to 0.0151 over these 110 instances.

Regression

This model's predictions are represented on the left of Figure Below. If you set **max_depth=3**, you get the predictions represented on the right. Notice how the predicted value for each region is always the average target value of the instances in that region. The algorithm splits each region in a way that makes most training instances as close as possible to that predicted value.



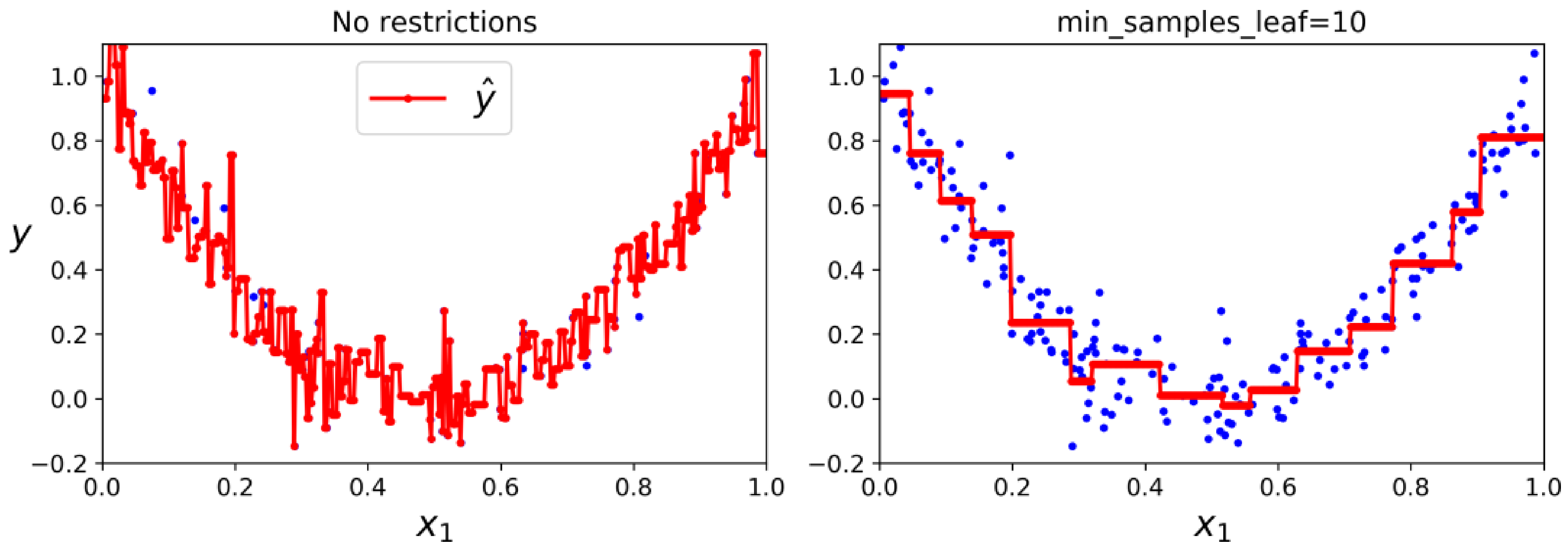
CART Cost Function for Regression

The CART algorithm works mostly the same way as earlier, except that instead of trying to split the training set in a way that minimizes impurity, it now tries to split the training set in a way that minimizes the MSE. Equation Below shows the cost function that the algorithm tries to minimize.

$$J(k, t_k) = \frac{m_{\text{left}}}{m}\text{MSE}_{\text{left}} + \frac{m_{\text{right}}}{m}\text{MSE}_{\text{right}} \quad \text{where} \begin{cases} \text{MSE}_{\text{node}} = \sum_{i \in \text{node}} \left(\hat{y}_{\text{node}} - y^{(i)}\right)^2 \\ \hat{y}_{\text{node}} = \frac{1}{m_{\text{node}}} \sum_{i \in \text{node}} y^{(i)} \end{cases}$$

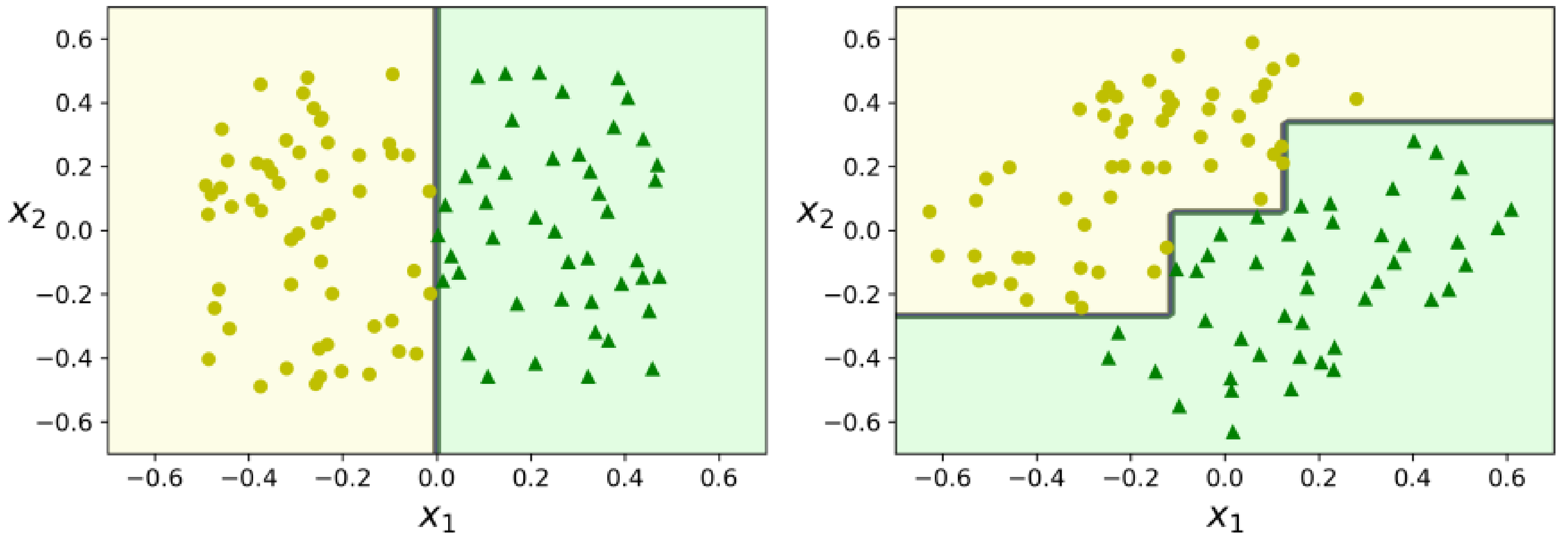
Regularizing a Decision Tree regressor

Just like for classification tasks, Decision Trees are prone to overfitting when dealing with regression tasks. Without any regularization (i.e., using the default hyperparameters), you get the predictions on the left of Figure Below. It is obviously overfitting the training set very badly. Just setting **min_samples_leaf=10** results in a much more reasonable model, represented on the right of Figure Below:



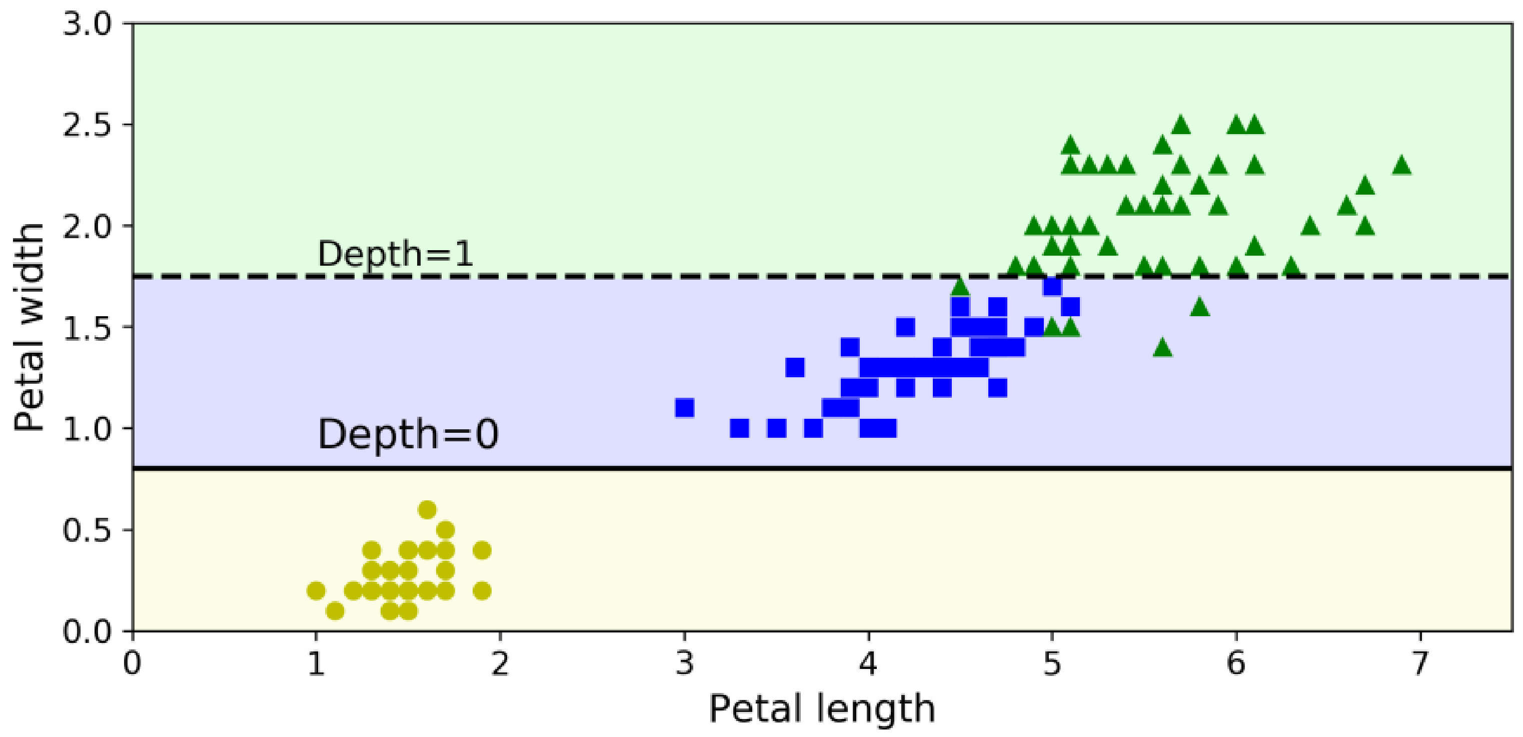
Instability

Decision Trees have a few limitations. First, as you may have noticed, Decision Trees love orthogonal decision boundaries (all splits are perpendicular to an axis), which makes them sensitive to training set rotation. For example, Figure Below shows a simple linearly separable dataset: on the left, a Decision Tree can split it easily, while on the right, after the dataset is rotated by 45°, the decision boundary looks unnecessarily convoluted. Although both Decision Trees fit the training set perfectly, it is very likely that the model on the right will not generalize well. One way to limit this problem is to use PCA, which often results in a better orientation of the training data.



Instability

More generally, the main issue with Decision Trees is that they are very sensitive to small variations in the training data. For example, if you just remove the widest *Iris Versicolor* from the iris training set (the one with petals 4.8 cm long and 1.8 cm wide) and train a new Decision Tree, you may get the model represented in Figure Below.



Random Forests can limit this instability by averaging predictions over many trees, as we will see in the next chapter.

References

- *Gelron, A. (2019). Hands-on machine learning with Scikit-Learn, Keras and TensorFlow: concepts, tools, and techniques to build intelligent systems (2nd ed.). O'Reilly.*