Hands-on Machine Learning

6. Decision Trees

1.
Training and Visualizing a Decision Tree

Build a Decision Tree

- Train a `DecisionTreeClassifier` on the iris dataset:

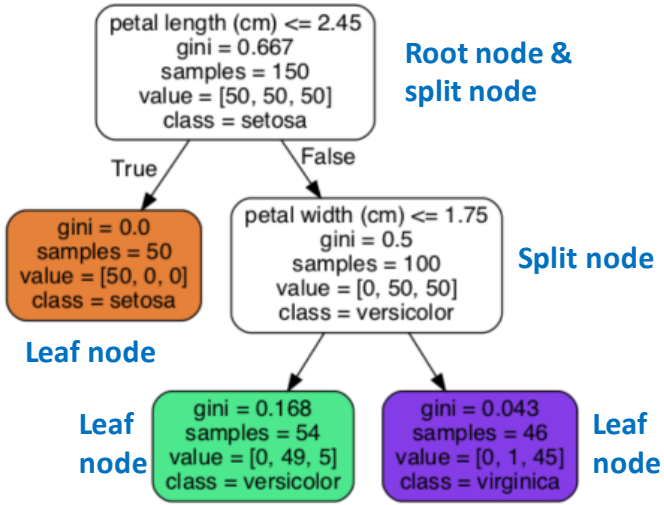
```
from sklearn.datasets import load_iris
from sklearn.tree import DecisionTreeClassifier

iris = load_iris(as_frame=True)
X_iris = iris.data[["petal length (cm)", "petal width (cm)"]].values
y_iris = iris.target

tree_clf = DecisionTreeClassifier(max_depth=2, random_state=42)
tree_clf.fit(X_iris, y_iris)
```

Visualize a Decision Tree

- Visualize the Decision Tree by 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=str(IMAGES_PATH / "iris_tree.dot"),
        feature_names=["petal length (cm)", "petal width (cm)"],
        class_names=iris.target_names,
        rounded=True,
        filled=True
    )
```

```
from graphviz import Source

Source.from_file(IMAGES_PATH / "iris_tree.dot")
```

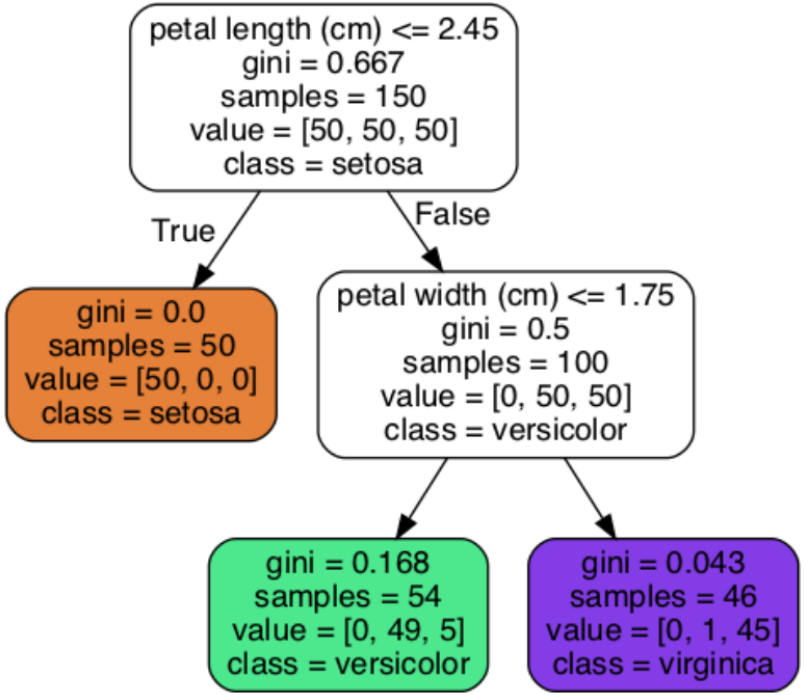


- Use the `dot` command-line tool to convert this *.dot* file to PNG:

```
$ dot -Tpng iris_tree.dot -o iris_tree.png
```

Node Attributes in a Decision Tree

- `samples` attribute counts how many training instances it applies to.
- `value` attribute tells you how many training instances of each class this node applies to.
- `gini` attribute measures its *Gini impurity*: a node is "pure" (`gini=0`) if all training instances it applies to belong to the same class.



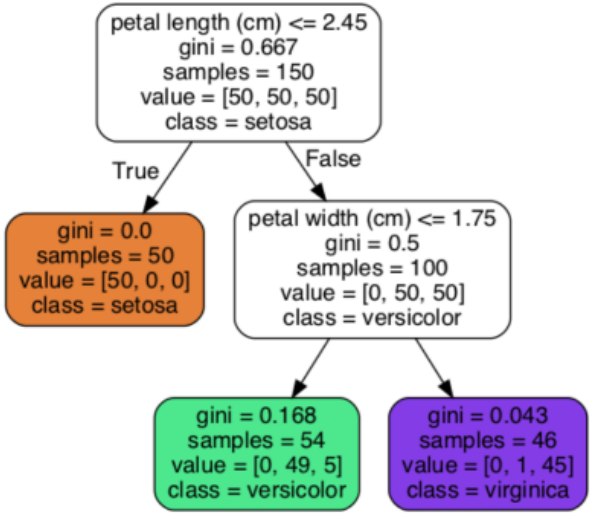
Gini Impurity

- Gini impurity G_i of the i-th node:

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

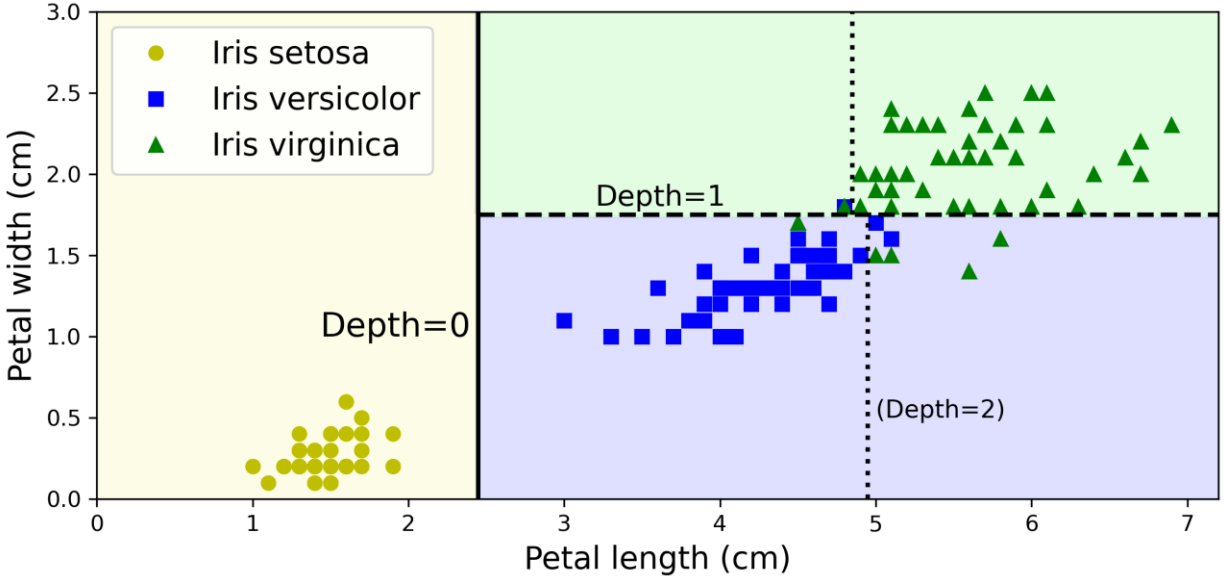
- $p_{i,k}$ is the ratio of class k instances among the training instances in the i-th node.
- Example (green node):

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



Decision Boundaries

- Decision tree's decision boundaries when `max_depth` is set to 3.



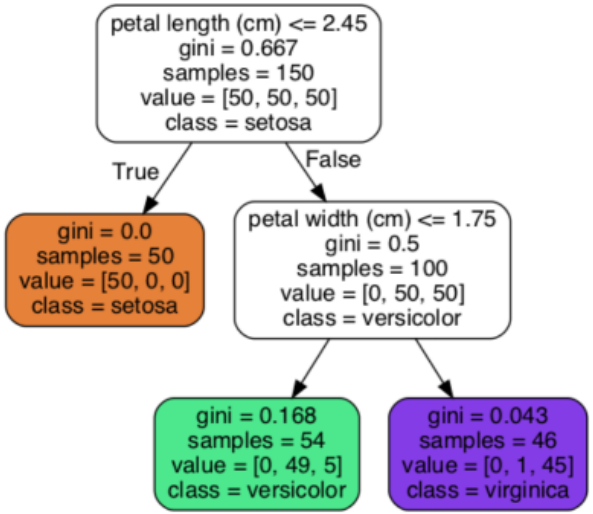
Model Interpretation

- Decision trees are intuitive, and their decisions are easy to interpret.
  - Such models are often called *white box models*.
- Neural networks are considered *black box models*: they make great predictions, but it is hard to explain in simple terms why the predictions were made.
- The field of *interpretable ML* aims at creating ML systems that can explain their decisions in a way humans can understand.

The COMPAS Race Bias

Estimating Class Probabilities

- A decision tree can estimate the probability that an instance belongs to a particular class k.
  - traverse the tree to find the leaf node for this instance, and then return the ratio of training instances of class k in this node.
- *Example*. A flower whose petals are 5 cm long and 1.5 cm wide: 0% *setosa* (0/54), 90.7% *versicolor* (49/54), and 9.3% *virginica* (5/54).




```
tree_clf.predict_proba([[5, 1.5]]).round(3)
```

```
array([[0.   , 0.907, 0.093]])
```

```
tree_clf.predict([[5, 1.5]])
```

```
array([1])
```

2.
The CART Training Algorithm

The CART Training Algorithm

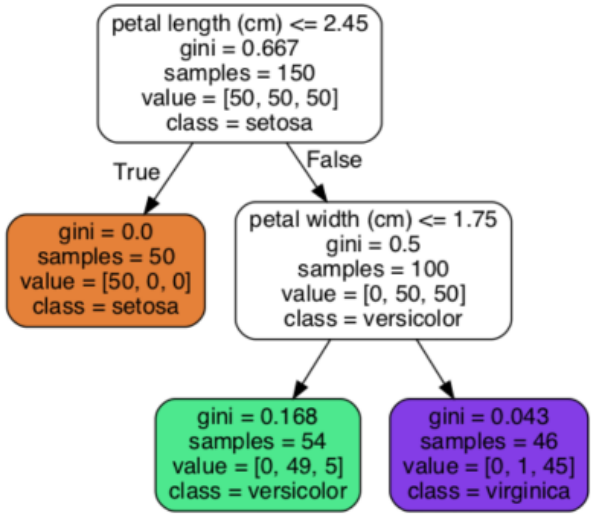
- Scikit-Learn uses the *Classification and Regression Tree* (CART) algorithm to train decision trees.
  - The algorithm first splits the training set into two subsets using a single feature k and a threshold t_k (e.g., petal length ≤ 2.45 cm).
- How does it choose k and t_k?
  - It searches for the pair (k, t_k) that produces the purest subsets, weighted by their size by minimizing the cost function:

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

  - $G_{\text{left/right}}$ measures the gini impurity of the left/right subset.
  - $m_{\text{left/right}}$ is the number of instances in the left/right subset.

The CART Training Algorithm

- Once the CART algorithm has split the training set in two, it splits the subsets using the same logic, 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.



- A few other hyperparameters control additional stopping conditions: `min_samples_split`, `min_samples_leaf`, and `max_leaf_nodes`.

The CART Training Algorithm

- The CART algorithm is a *greedy algorithm*: it greedily searches for an optimum split at the top level, then repeats the process at each subsequent level.
  - It does not check whether or not the split will lead to the lowest possible impurity several levels down.
- Finding the optimal decision tree is an *NP-hard* problem.
  - It requires $O(2^m)$ time, making the problem intractable even for small training sets.
- Prediction complexity = $O(\log_2 m)$ (traversing the decision tree)
- Training complexity = $O(n \times m \log_2 m)$

Gini Impurity or Entropy?

- By default, the `DecisionTreeClassifier` class uses the Gini impurity measure, but you can select the *entropy* measure instead by setting the `criterion` hyperparameter to `"entropy"`.
- In ML, entropy is frequently used as an impurity measure: a set's entropy is zero when it contains instances of only one class.

$$H_i = -\sum_{k=1}^{n} p_{i,k} \log_2 p_{i,k}$$

- Gini impurity is slightly faster to compute, while entropy tends to produce slightly more balanced trees.

3.
Regularization

Regularization Hyperparameters

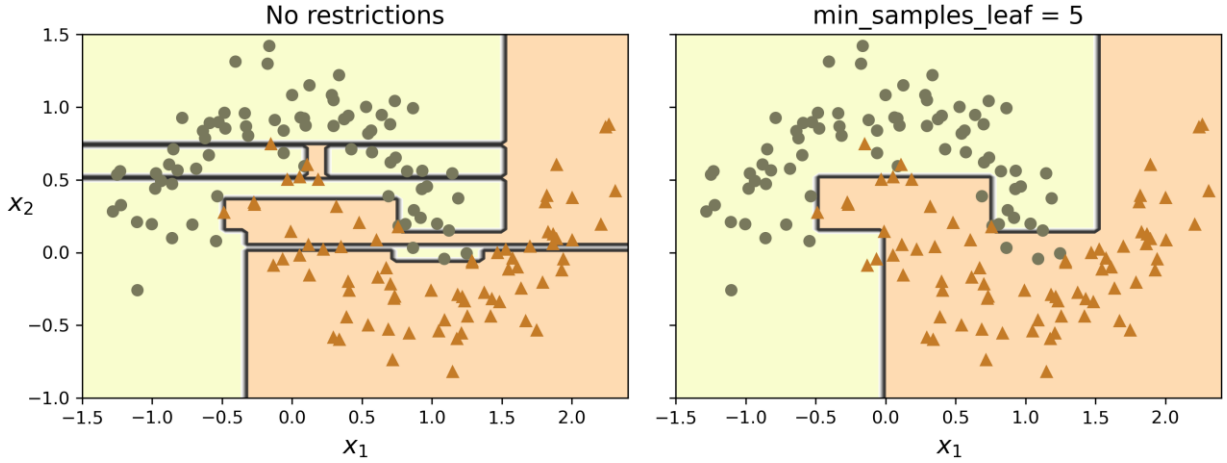
- Decision trees make few assumptions about the training data.
  - e.g. linear models assume that the data is linear.
- If left unconstrained, the tree structure will adapt itself to the training data, fitting it very closely—most likely overfitting it.
- Decision tree is a *nonparametric model*: the number of parameters is not determined prior to training.
  - A *parametric model*, e.g. a linear model, has a predetermined number of parameters, so its degree of freedom is limited, reducing the risk of overfitting.

Regularization Hyperparameters

- `max_depth`: maximum depth of the decision tree.
- `max_features`: maximum number of features that are evaluated for splitting at each node.
- `max_leaf_nodes`: maximum number of leaf nodes.
- `min_samples_split`: minimum number of samples a node must have before it can be split.
- `min_samples_leaf`: minimum number of samples a leaf node must have to be created.
- `min_weight_fraction_leaf`: same as `min_samples_leaf` but expressed as a fraction of the total number of weighted instances.

Regularized Decision Tree

```python
from sklearn.datasets import make_moons

X_moons, y_moons = make_moons(n_samples=150, noise=0.2, random_state=42)

tree_clf1 = DecisionTreeClassifier(random_state=42)
tree_clf2 = DecisionTreeClassifier(min_samples_leaf=5, random_state=42)
tree_clf1.fit(X_moons, y_moons)
tree_clf2.fit(X_moons, y_moons)
```

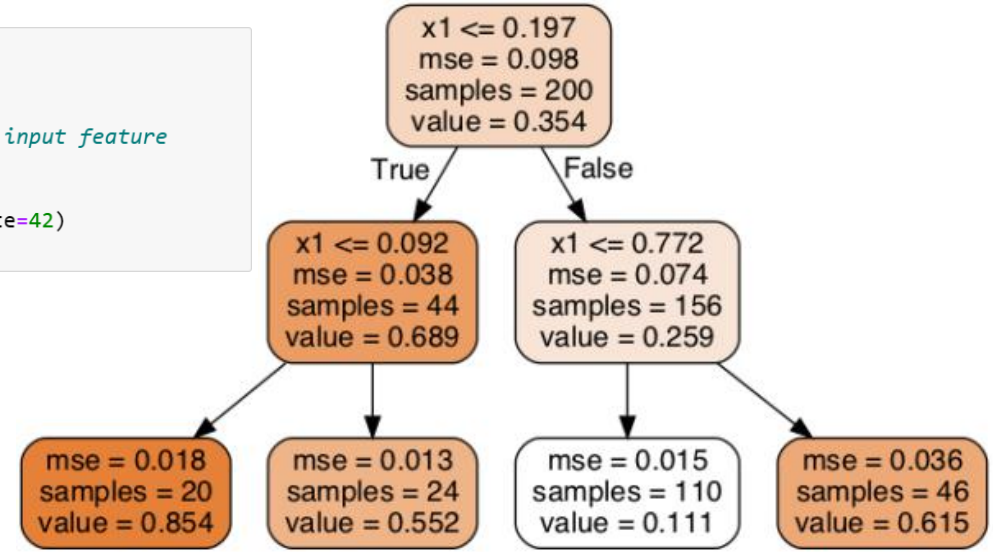


4.
Regression

Decision Trees for Regression

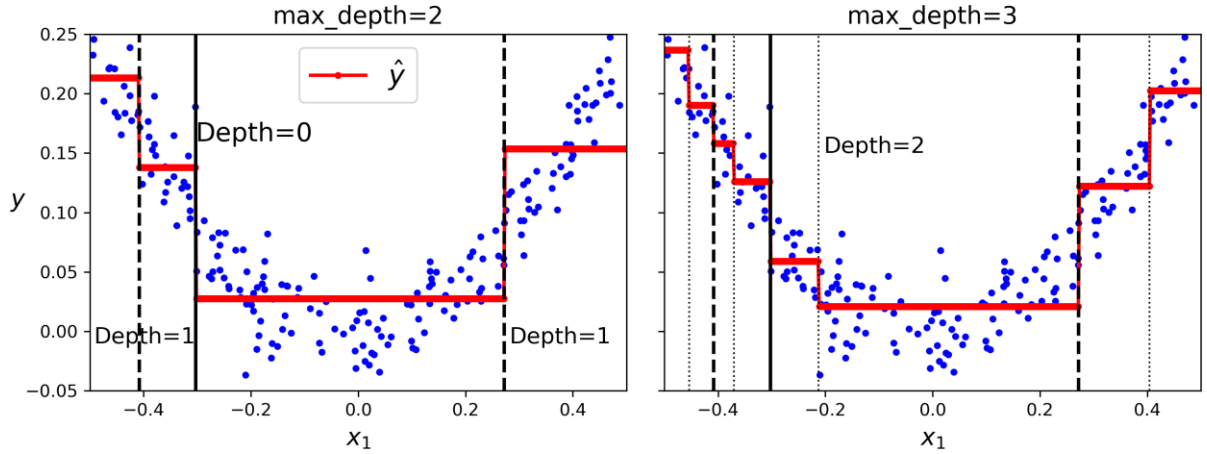
- Decision trees are also capable of performing regression tasks.

```
from sklearn.tree import DecisionTreeRegressor

np.random.seed(42)
X_quad = np.random.rand(200, 1) - 0.5  # a single random input feature
y_quad = X_quad ** 2 + 0.025 * np.random.randn(200, 1)

tree_reg = DecisionTreeRegressor(max_depth=2, random_state=42)
tree_reg.fit(X_quad, y_quad)
```



Decision Trees for Regression

- The predicted value for each region is 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.



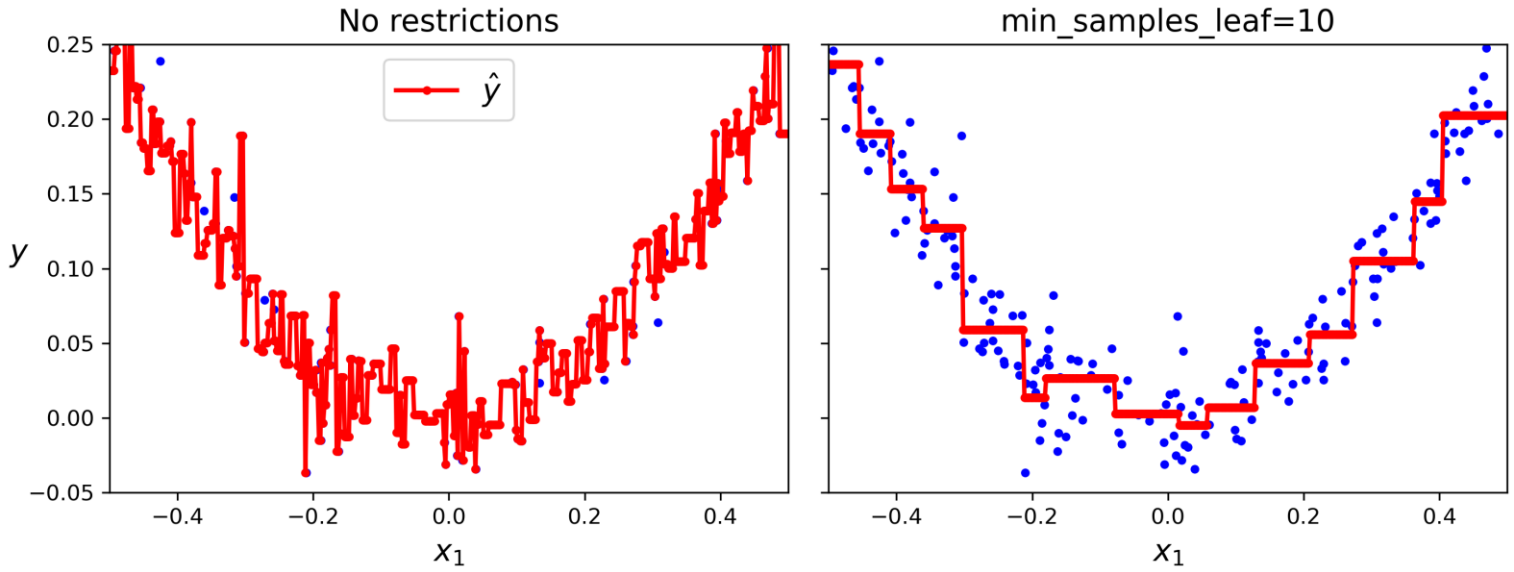
The CART Algorithm for Regression

- The CART algorithm splits the training set in a way that minimizes the MSE (instead of impurity).

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

- Just like for classification tasks, decision trees are prone to overfitting when dealing with regression tasks.

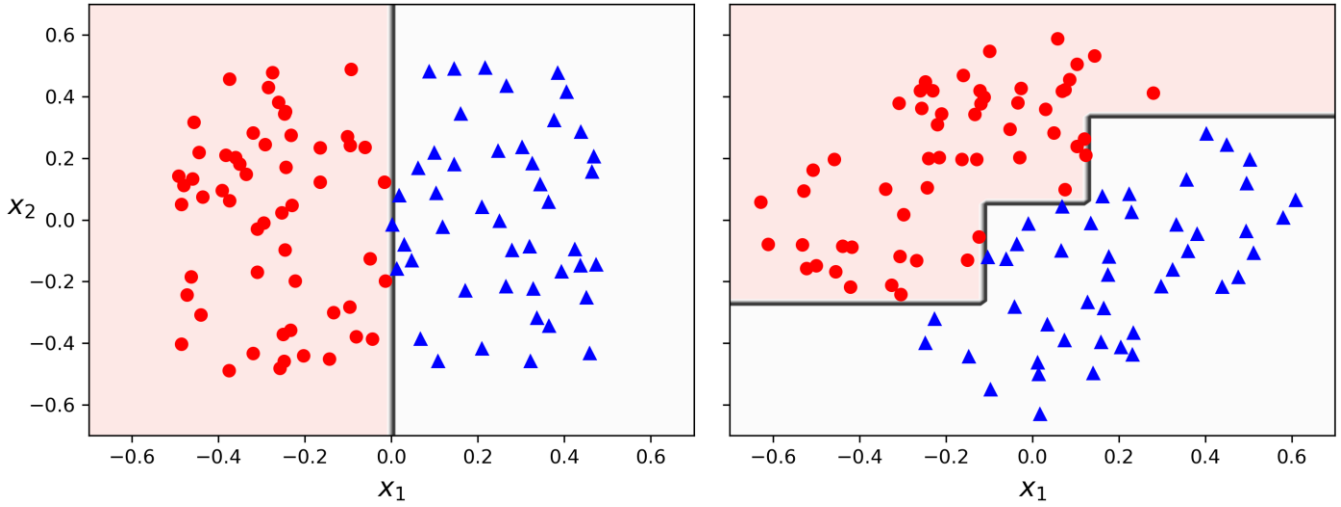
Regularized Decision Tree for Regression



5.
Challenges

Sensitivity to Axis Orientation

- Decision trees like orthogonal decision boundaries (all splits are perpendicular to an axis), which makes them sensitive to the data's orientation.



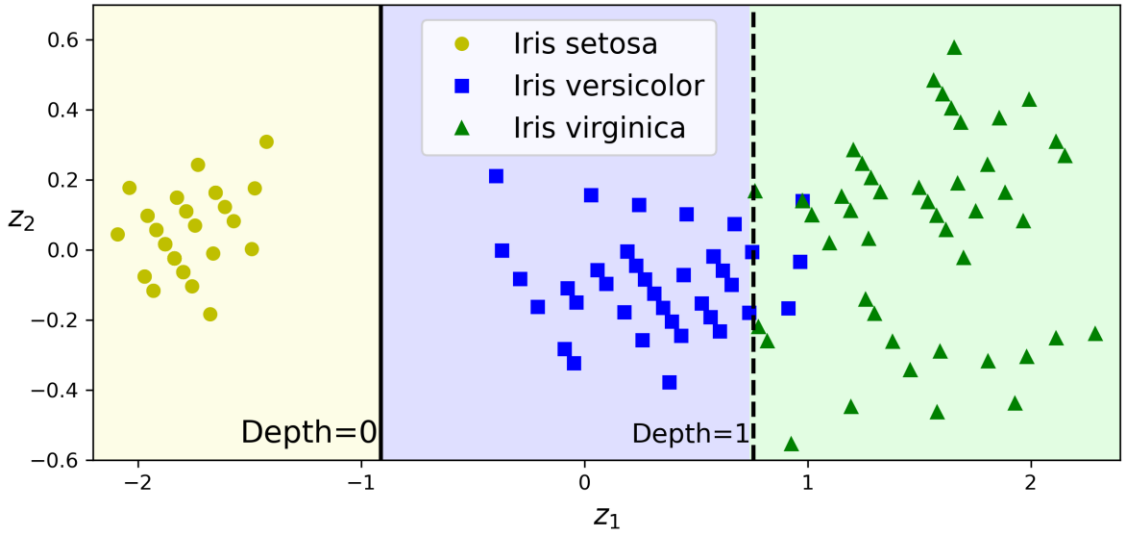
Sensitivity to Axis Orientation

- One way to limit this problem is to scale the data, then apply a *principal component analysis* (PCA) transformation.
  - PCA rotates the data in a way that reduces the correlation between the features.
- Create a pipeline that scales the data and rotates it using PCA, then train a `DecisionTreeClassifier` on that data:

```
from sklearn.decomposition import PCA
from sklearn.pipeline import make_pipeline
from sklearn.preprocessing import StandardScaler

pca_pipeline = make_pipeline(StandardScaler(), PCA())
X_iris_rotated = pca_pipeline.fit_transform(X_iris)
tree_clf_pca = DecisionTreeClassifier(max_depth=2, random_state=42)
tree_clf_pca.fit(X_iris_rotated, y_iris)
```

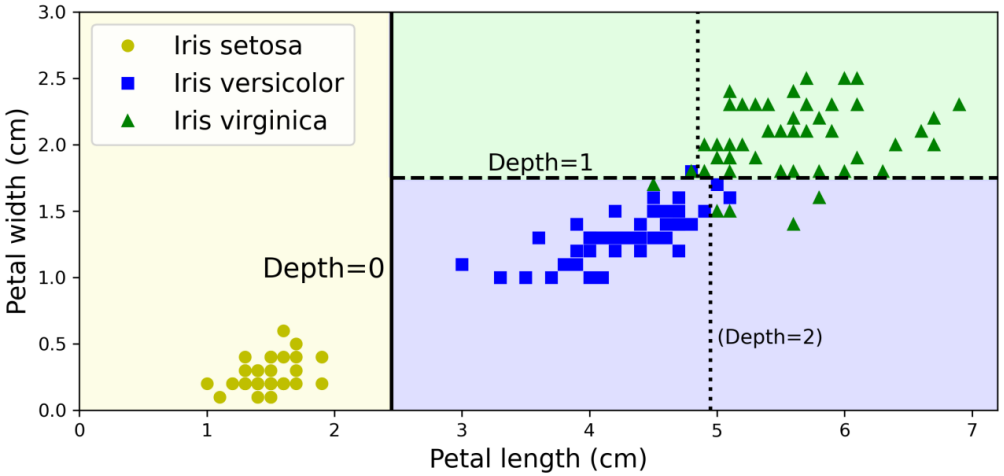
Sensitivity to Axis Orientation

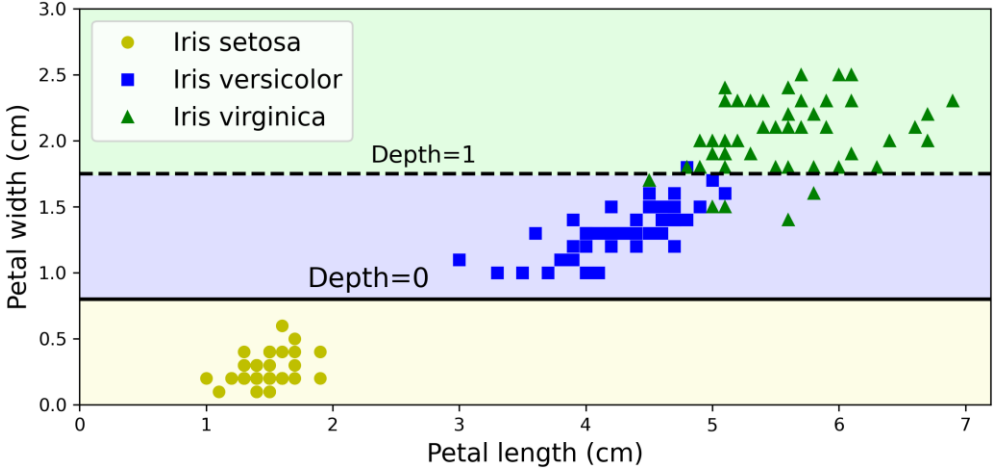
- The rotation makes it possible to fit the dataset pretty well using only one feature, z_1, which is a linear function of the original petal length and width.



Decision Trees have a High Variance

- The main issue with decision trees: high variance.
  - small changes to the hyperparameters or to the data may produce very different models.
- Since the training algorithm used by Scikit-Learn randomly selects the set of features to evaluate at each node, even retraining the same decision tree on the exact same data may produce a very different model.
- By averaging predictions over many trees, it's possible to reduce variance significantly.
  - Such an *ensemble* of trees is called a *random forest*.


Iris setosa
Iris versicolor
Iris virginica
Depth=1
Depth=0
(Depth=2)
Petal width (cm)
Petal length (cm)


Iris setosa
Iris versicolor
Iris virginica
Depth=1
Depth=0
Petal width (cm)
Petal length (cm)