CAP4770 – Intro to Data Science

Decision Trees

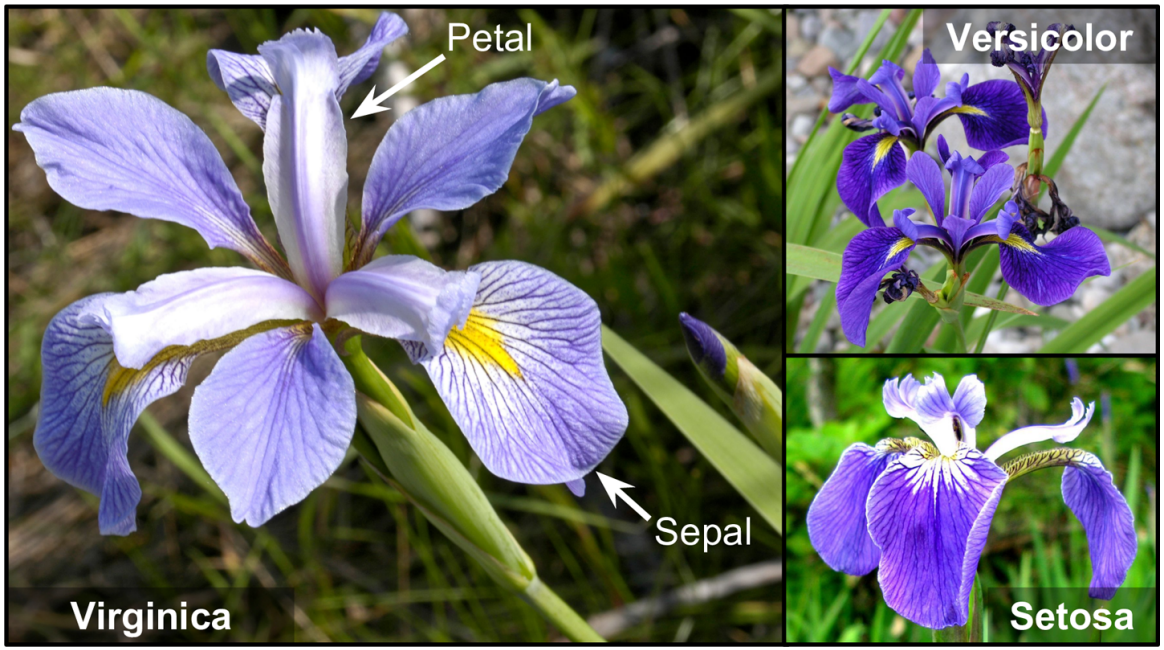
Prof. Ye Xia

Overview

- Decision trees are versatile machine learning algorithms that can perform both classification and regression tasks, and even multioutput tasks.
- They are 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.

Iris Data

- We will use a famous dataset that contains the sepal and petal length and width of 150 iris flowers of three different species: Iris Setosa, Iris Versicolor, and Iris Virginica.



- Sepal is the outer part of the flower that provides protection for the

flower during its bud stage. They are green in most flowers (not for iris).

- Petals are the inner part, usually brightly colored, that surrounds the reproductive units of flowers.
- Load iris data, which is dictionary like

```
from sklearn.datasets import load_iris
iris = load_iris()
list(iris.keys())
```

```
['data',
 'target',
 'frame',
 'target_names',
 'DESCR',
 'feature_names',
 'filename',
 'data_module']
```

```
iris.feature_names
```

```
['sepal length (cm)',
 'sepal width (cm)',
 'petal length (cm)',
 'petal width (cm)']
```

```
iris.target_names
```

```
array(['setosa', 'versicolor', 'virginica'], dtype='<U10')
```

- Input: 4 features per instance; in `iris.data`
- Output value: from $\{0, 1, 2\}$ representing three classes; in `iris.target`

```
print(iris.DESCR)
```

```
.. _iris_dataset:

Iris plants dataset
--------------------

**Data Set Characteristics:**

    :Number of Instances: 150 (50 in each of three classes)
    :Number of Attributes: 4 numeric, predictive attributes and the class
    :Attribute Information:
        - sepal length in cm
        - sepal width in cm
        - petal length in cm
        - petal width in cm
        - class:
                - Iris-Setosa
                - Iris-Versicolour
                - Iris-Virginica
```

Training and Visualizing a Decision Tree

- For now, we will use only **two** features: petal length and petal width.
- Apply a decision tree classifier to the iris dataset.

```
from sklearn.tree import DecisionTreeClassifier

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*.

```python
from sklearn.tree import export_graphviz

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

- Produce a visual view of the tree by entering at the command line:

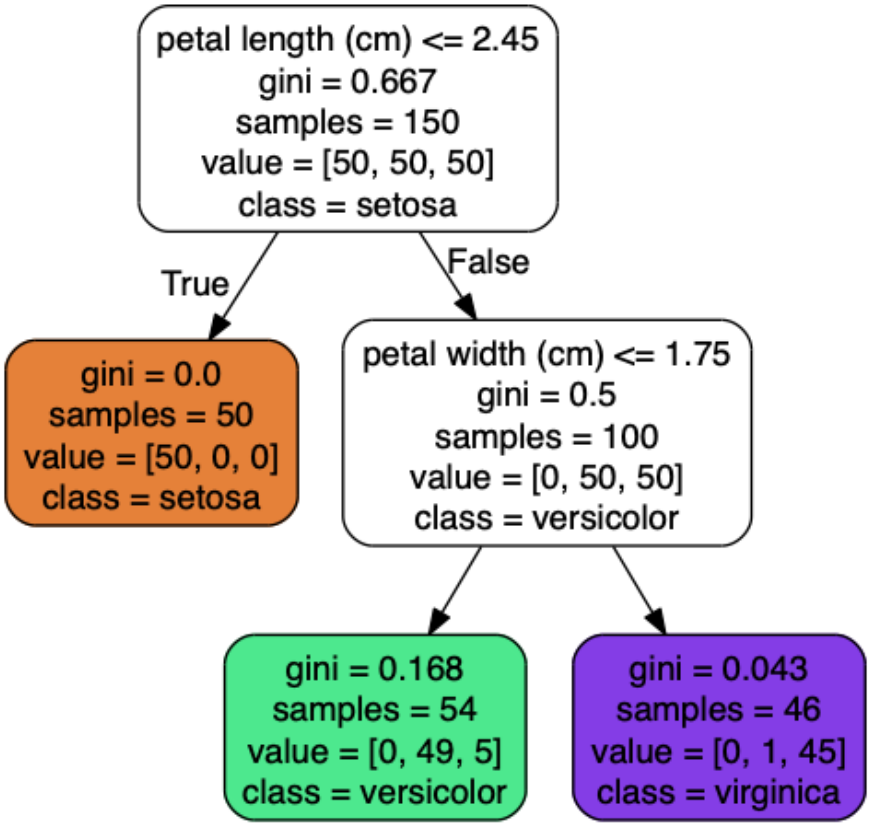
```
$ dot -Tpng iris-tree.dot -o iris-tree.png
```

`dot` is a graph layout engine that produces hierarchical or layered drawings of directed graphs.

- Alternatively, you can directly plot the tree from within Scikit-Learn:

```
from sklearn import tree
tree.plot_tree(tree_clf)
```

- The tree produced by `dot` looks like the following:



- `samples`: how many training instances the node applies to.
- `value`: 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.
- `gini`: the Gini measure of *impurity*; a node is "pure" (`gini=0`) if all training instances it applies to belong to the same class.

  For node i, let G_i denote its `gini` value. Let p_{ik} be the fraction of class k instances among the training instances in the ith node. G_i is computed/defined as

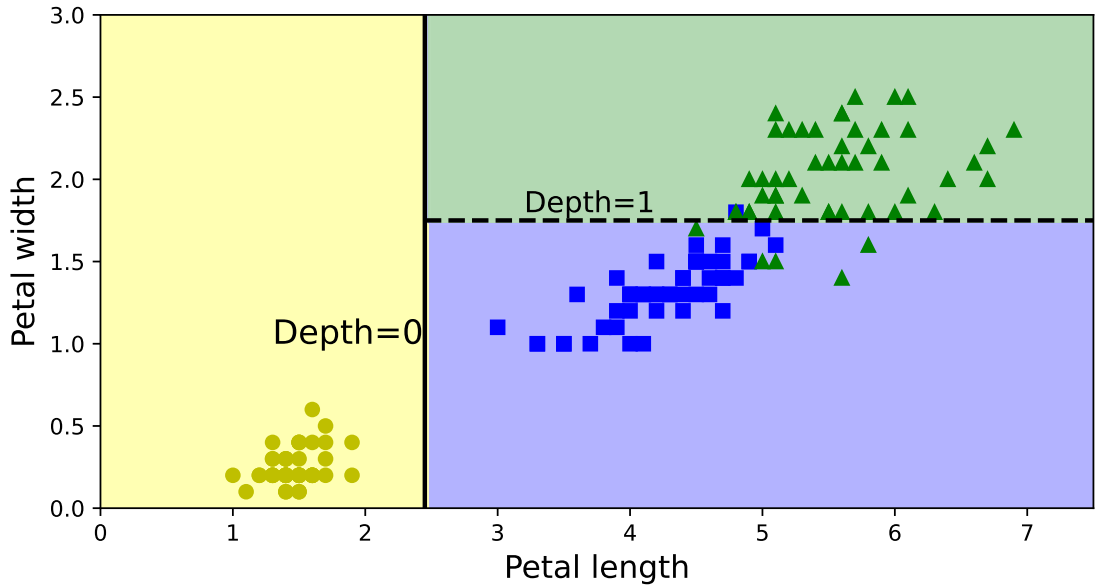
$$G_i = 1 - \sum_k p_{ik}^2.$$

  $G_i = 0$ if all the instances at node i belong to the same class. G_i is maximized when the instances are evenly distributed among the classes.

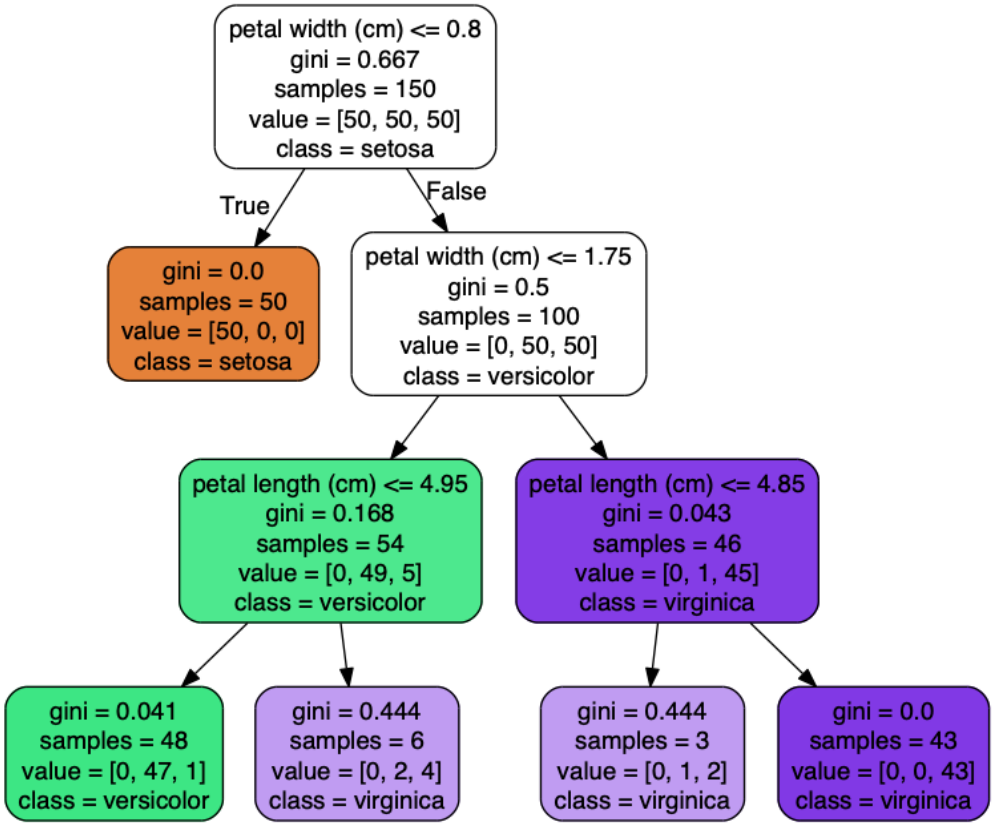
- Scikit-Learn implements the CART algorithm, which produces only

binary trees (i.e., questions only have yes/no answers). However, other algorithms such as ID3 can produce decision trees with nodes that have more than two children.

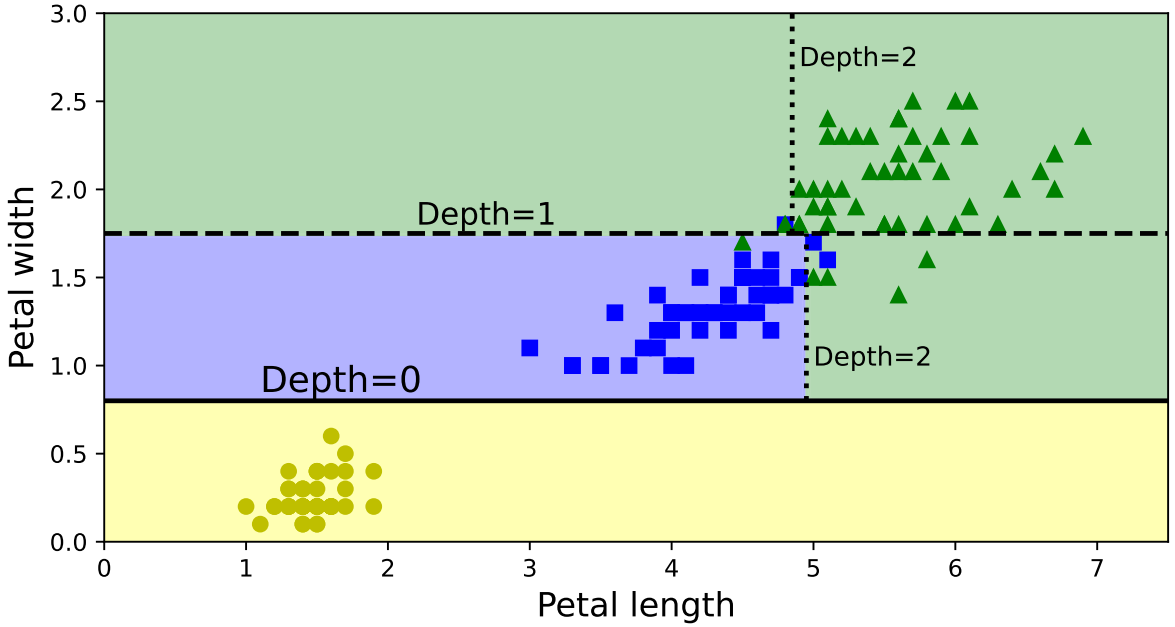
- One of the many qualities of decision trees is that they require very little data preparation. In fact, they don't require feature scaling or centering at all.
- The decision boundaries are shown below.



- For `max_depth = 3`, the decision tree looks like the following:



The decision regions and boundaries are:



Note that the two regions with `petal_width` > 1.75 are both Iris Virginica.

Prediction and Prediction Probability

- For prediction, let's try the case of `max_depth = 2`.

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

```
array([[0.        , 0.90740741, 0.09259259]])
```

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

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

  - The prediction probabilities are constants for each leaf node (or equivalently, for each decision region).
  - The computation is based on the number of instances in the `value` field. Given all the training instances falling in a node/region and their true classes, one can compute the fraction of each class, and that fraction is interpreted as the prediction probability for the class.

E.g., for the bottom-left node, the fractions are

$$[0/54, 49/54, 5/54] = [0, 0.90740741, 0.09259259],$$

which are the fractions shown in the earlier code output.

The CART Training Algorithm

- Scikit-Learn implements a version of the Classification and Regression Tree (CART) algorithm to train decision trees (also called "growing" trees).
  - The implementation doesn't handle categorical input features directly. One needs to convert a categorical input into numerical values, e.g., by one-hot encoding or integer-encoding for ordinal data.
  - There are other decision-tree training algorithms (not implemented in Scikit-Learn) that handle categorical input natively, such as ID3, C4.5 and C5 (these algorithms grew out of handling categorical input).
- At the root node, the CART algorithm splits the training set into two subsets using a single feature j and a threshold t_j.

To do that, it searches for the pair (j, t_j) that produces the purest subsets (weighted by their size). Specifically, it solves the following problem:

$$\min_{j,t_j} \frac{m_l}{m} G_l + \frac{m_r}{m} G_r,$$

where

- G_l or G_r: the impurity of the left or right subset
- m_l or m_r: the number of instances in the left or right subset
- m: the number of all the training instances

- 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 the recursion once certain stopping condition is met, for instance, it has reached 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`, `min_weight_fraction_leaf`, and `max_leaf_nodes`).
- CART is a greedy heuristic algorithm.

Complexity of CART Implementation (Training Phase)

The textbook has some issues about this. Here is updated reasoning.

- The number of steps taken to build the tree depends on the implementation. The implementation below takes $O(nm \log m)$ steps of computation.
  - m: the number of training instances
  - n: the number of features
- One-time effort: For each input feature, sort all the instances in increasing order of that feature value. We get n sorted arrays. This takes $O(nm \log m)$ steps.
- Next, consider the root level. For each feature j, check the m instances in increasing order of the values of feature j.
  - For each instance i, if we imagine using the feature value x_{ij} as

the threshold value t_j to split all the instances, we can directly compute the optimization objective $\frac{m_l}{m}G_l + \frac{m_r}{m}G_r$ for that split (need some thinking here).

- As we iterative over the instances i, we keep track of the instance that gives the smallest objective value.
- Since we can do all this in one scan of the instances, the number of steps is $O(m)$ for feature k.
- We need to do the above for all the features and in the end find the smallest objective value over all features and all thresholds. The number of steps is $O(nm)$.
- Once we find the best feature-threshold pair for splitting the instances into the two child nodes, we scan all the instances again to do the actual splitting.

  For each child node, we also need to do some maintenance to build the n sorted arrays for all the instances falling in that child node. This can be done by scanning the original sorted arrays of

all the instances.

This takes $O(nm)$ steps.

- For the root node, the number of steps is $O(nm)$.

- We repeat the above procedure for all the nodes in each level.
- Suppose the binary tree is more or less balanced. The depth of the tree is $O(\log_2 m)$ (assuming each leaf node covers a constant number of training instances).
- The total number of steps $T(n, m)$ satisfies the master equation:

$$T(n, m) = O(nm) + 2T(n, m/2).$$

The iteration is in m only. We learned from CLR (Introduction to Algorithms), $T(n, m) = O(nm \log m)$.

This is actually easy to see for a balanced tree. The second level has two subproblems of the size $n, m/2$. The number of steps is equal to $2O(nm/2) = O(nm)$. The number of steps at every level is $O(nm)$. There are $O(\log_2 m)$ levels.

- The one-time sorting also takes $O(nm \log m)$ steps. Therefore, the training phase takes $O(nm \log m)$ steps.

Meaning of Gini Impurity

- Consider a set of instances falling into K classes. Let p_k be the fraction of class-k instances in the set.
- Suppose we draw an instance uniformly at random from the set. The probability that it belongs to class k is equal to p_k.
- Suppose we classify the instance into one of the K classes randomly according to the probabilities $p_1, \ldots, p_K$.
- The probability of mis-classification is has the same expression as the Gini Impurity.

$$\sum_{k=1}^{K} p_k \sum_{j \neq k} p_j = \sum_{k=1}^{K} p_k(1 - p_k) = \sum_{k=1}^{K} p_k - \sum_{k=1}^{K} p_k^2 = 1 - \sum_{k=1}^{K} p_k^2.$$

- Gini impurity is maximized when $p_k = 1/K$ for all k. By Jensen's

inequality,

$$\frac{1}{K}\sum_{k=1}^{K} p_k^2 \geq \left(\frac{1}{K}\sum_{k=1}^{K} p_k\right)^2 = \frac{1}{K^2}. \tag{1}$$

Then,

$$\sum_{k=1}^{K} p_k^2 \geq \frac{1}{K}.$$

Equality is achieved when $p_k = 1/K$ for all k.

Jensen's Inequality: Suppose X is a random variable. When f is a convex function, we have $f(E[X]) \leq E[f(X)]$.

The equality holds if and only if f is a linear function.

In (1), the random variable X takes the value p_k for each k with probability $1/K$. The function f is $f(x) = x^2$.

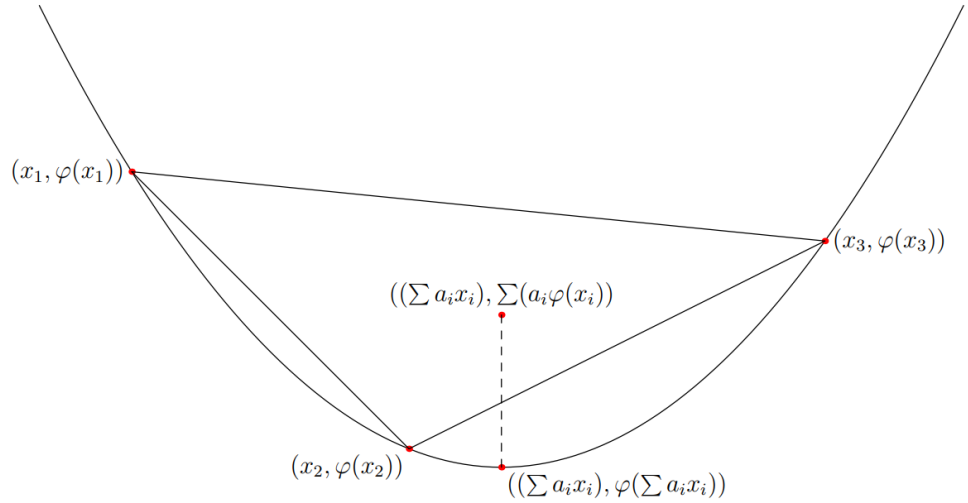


Figure 1: The positive weights satisfy $\sum_i a_i = 1$, and therefore, can be understood as probabilities. The weighted point lies in the convex hull (the triangle) of the original points, which lies above the function itself by the definition of convexity. (Wikipedia)

Alternative Training Objective: Information Gain

- Consider a set of instances falling into K classes. Let p_k be the fraction of class-k instances in the set.
- We can interpret the p_k's as probabilities. The entropy is defined as

$$H(p_1, \ldots, p_k) = \sum_{k=1}^{K} p_k \log_2 \frac{1}{p_k} = -\sum_{k=1}^{K} p_k \log_2 p_k.$$

- The entropy is often understood as a measure of disorder or lack of information.

  It is maximized when $p_k = 1/K$ for all k.

- Entropy is equal to the minimum of the average number of bits needed to represent a class. It is the limit of compression.

  Suppose you have a long text with L letters in length. Suppose there are K distinct letters (i.e., K is the alphabet size). Suppose each

letter is drawn in IID fashion from the K distinct letters according to the probabilities $p_1, \ldots, p_k$. Over the long text, you will expect that the fraction of the type-k letter is equal to p_k.

You will represent each letter by a bit pattern. Suppose you want to use the smallest number of bits to represent the entire text. The natural thing to do is to use short bit-patterns for very frequent letters, and long bit-patterns for infrequent letters.

There is a way to represent type-k letter using $\log_2 \frac{1}{p_k}$ bits (see Huffman code). Then, the expected number of bits needed to represent the text is equal to

$$\sum_k p_k L \log_2 \frac{1}{p_k} = -L \sum_{k=1}^{K} p_k \log_2 p_k = LH(p_1, \ldots, p_k).$$

This turns out to be the best you can do in terms of how much you can compress.

Therefore, the entropy is equal to the minimum of the average

number of bits needed to represent a letter.

- Suppose at a node S on a decision tree, the entropy is H_S, which is calculated using the fractions of the instances that fall into different classes.

  If we split the instances at S into left and right child nodes, we can calculate the entropy at the left and right nodes, denoted by H_l and H_r respectively.

  The information gain is:

$$H_S - \left(\frac{m_l}{m} H_l + \frac{m_r}{m} H_r \right).$$

- In the tree-generation algorithms of ID3, C4.5 and C5.0, the split is determined by which input-feature and value pair gives the largest information gain.

Why Is It an Information Gain: Intuitive Argument

- We want to argue: $H_S \geq \left(\frac{m_l}{m} H_l + \frac{m_r}{m} H_r\right)$.

  Intuitively, more information is provided during the split of the tree.

  The split is based on a feature j and threshold t_j. On the left child, every instance i satisfies $x_{ij} \leq t_j$; on the right child, every instance i satisfies $x_{ij} > t_j$.

  If the fact that an arbitrary instance, say i, belongs to a class k has anything to do with its value of feature j, x_{ij}, then knowing a range in which x_{ij} lies provides more information about how to classify instance i.

  When we know more information about the probability of different classes, we can compress better. $\frac{m_l}{m} H_l + \frac{m_r}{m} H_r$ measures the best compression after knowing this extra information. Therefore, $H_S \geq \left(\frac{m_l}{m} H_l + \frac{m_r}{m} H_r\right)$.

Why Is It an Information Gain: Formal Argument

- Formally, suppose, at the parent node S, the split is based on feature j and threshold t_j. Suppose the total number of instances at the parent node is m, which is a constant.

  Suppose we choose an instance uniformly at random.

  Let W the random variable representing the class of the chosen instance chosen. We have $P(W = k) = p_k$.

  Let T the random variable representing the value of feature j for the chosen instance.

- Let $H(W)$ be the **entropy** of W, which is

$$H(W) = -\sum_{k=1}^{K} p_k \log_2 p_k = H_S.$$

- There is an underlying joint probability $P(W = k, T \leq t)$ for each k and t, which is equal to the fraction of the instances for which the event

$\{W = k, T \leq t\}$ occurs.

- Therefore, there is the conditional probability $P(W = k|T \leq t)$. In particular, let $t = t_j$.

$$P(W = k|T \leq t_j) = \frac{P(W = k, T \leq t_j)}{P(T \leq t_j)}.$$

- Note that $m \times P(T \leq t_j)$ is the number of instances sent to the left child node; $m \times P(W = k, T \leq t_j)$ is the number of class-k instances at the left child node.
- Therefore, $P(W = k|T \leq t_j)$ is equal to the fraction of instances at the left child node that belong to class k, which we will denote as l_k.
- Similarly, $P(W = k|T > t_j)$ is equal to the fraction of instances at the right child node that belong to class k, which we will denote as r_k.

- Let $Z = \mathbf{1}_{\{T \leq t_j\}}$ be the indicator random variable, i.e., $Z = 1$ when $T \leq t_j$; $Z = 0$ otherwise.
- $P(W = k|T \leq t_j)$ can be written as $P(W = k|Z = 1)$; $P(W = k|T > t_j)$ can be written as $P(W = k|Z = 0)$.

- Let $H(W|Z=1)$ be the entropy of W given we know $Z=1$ (i.e., $T \leq t_j$) defined using the conditional probabilities:

$$H(W|Z=1) = -\sum_{k=1}^{K} P(W=k|Z=1)\log_2 P(W=k|Z=1)$$
$$= -\sum_{k=1}^{K} l_k \log_2 l_k = H_l.$$

  Similarly,

$$H(W|Z=0) = -\sum_{k=1}^{K} P(W=k|Z=0)\log_2 P(W=k|Z=0)$$
$$= -\sum_{k=1}^{K} r_k \log_2 r_k = H_r.$$

- Note $H(W|Z=1) = H_l$ and $H(W|Z=0) = H_r$.
- Let $H(W|Z)$ be the **conditional entropy** of W given Z, defined by

'averaging' the above over $Z = 0$ and $Z = 1$.

$$
\begin{aligned}
H(W|Z) &= -\sum_{z\in\{0,1\}} P(Z=z)H(W|Z=z) \\
&= -\sum_{z\in\{0,1\}} P(Z=z)\sum_{k=1}^{K} P(W=k|Z=z)\log_2 P(W=k|Z=z) \\
&= -\sum_{z\in\{0,1\}}\sum_{k=1}^{K} P(Z=z)P(W=k|Z=z)\log_2 P(W=k|Z=z) \\
&= -\sum_{z\in\{0,1\}}\sum_{k=1}^{K} P(W=k,Z=z)\log_2 P(W=k|Z=z) \\
&= -\sum_{z\in\{0,1\}}\sum_{k=1}^{K} P(W=k,Z=z)\log_2 \frac{P(W=k,Z=z)}{P(Z=z)}
\end{aligned}
$$

The last expression above is usually taken as the definition of conditional entropy.

- Note that $P(Z=1) = P(T \le t_j) = \sum_{k=1}^{K} P(W=k, T \le t_j)$. This is equal to the fraction of the instances whose feature j's value is no more than

t_j. Since any such instance is sent to the left child node, we see that

$$P(Z = 1) = \frac{m_l}{m}.$$

Similarly,

$$P(Z = 0) = \frac{m_r}{m}.$$

- Then, we have

$$\begin{aligned} H(W|Z) &= \frac{m_l}{m} H(W|Z = 1) + \frac{m_r}{m} H(W|Z = 0) \\ &= \frac{m_l}{m} H_l + \frac{m_r}{m} H_r. \end{aligned}$$

- The quantity $I(W; Z) \triangleq H(W) - H(W|Z)$ is called **mutual information** between W and Z. It quantifies the "amount of information" (in bits) obtained about one random variable by observing the other random variable. It is also called the **information gain**.

  $H(W)$ measures the amount of disorder of W; $H(W|Z)$ measures the amount of disorder of W given Z. The difference is the reduction in disorder or gain of information in W when Z is observed.

$$
\begin{aligned}
I(W;Z) &= -\sum_{k=1}^{K} P(W=k)\log_2 P(W=k) \\
&\quad + \sum_{z\in\{0,1\}}\sum_{k=1}^{K} P(W=k,Z=z)\log_2 \frac{P(W=k,Z=z)}{P(Z=z)} \\
&= -\sum_{z\in\{0,1\}}\sum_{k=1}^{K} P(W=k,Z=z)\log_2 P(W=k) \\
&\quad + \sum_{z\in\{0,1\}}\sum_{k=1}^{K} P(W=k,Z=z)\log_2 \frac{P(W=k,Z=z)}{P(Z=z)} \\
&= \sum_{z\in\{0,1\}}\sum_{k=1}^{K} P(W=k,Z=z)\log_2 \frac{P(W=k,Z=z)}{P(W=k)P(Z=z)}
\end{aligned}
$$

The last expression is usually taken as the definition of mutual information.

- The mutual information is the **Kullback-Leibler divergence** of the joint distribution $P_{W,Z}$ w.r.t. the product of the marginal distributions

$P_W \cdot P_Z$.

- Using Jensen's inequality, it is easy to show $I(W;Z) \geq 0$, which is the same as $H(W) - H(W|Z) \geq 0$. Then, we get

$$H_S - (\frac{m_l}{m} H_l + \frac{m_r}{m} H_r) \geq 0.$$

- Some details about the use of Jensen's inequality:

$$
\begin{aligned}
I(W;Z) &= \sum_{z\in\{0,1\}} \sum_{k=1}^{K} P(W=k, Z=z) \log_2 \frac{P(W=k, Z=z)}{P(W=k)P(Z=z)} \\
&= -\sum_{z\in\{0,1\}} \sum_{k=1}^{K} P(W=k, Z=z) \log_2 \frac{P(W=k)P(Z=z)}{P(W=k, Z=z)} \\
&\geq -\log_2 \left(\sum_{z\in\{0,1\}} \sum_{k=1}^{K} P(W=k, Z=z) \frac{P(W=k)P(Z=z)}{P(W=k, Z=z)} \right) \\
&= -\log_2 \left(\sum_{z\in\{0,1\}} \sum_{k=1}^{K} P(W=k)P(Z=z) \right) \\
&= -\log_2 1 = 0.
\end{aligned}
$$

Here, we use the fact that the $-\log_2(\cdot)$ function is convex. The random variable X is

$$X(k,z) = \frac{P(W=k)P(Z=z)}{P(W=k, Z=z)}.$$

(A random variable is really a function of the outcome of a random

experiment. Here, the experiment is to choose a random instance, and an outcome is a pair k and z.)

When applied to the $-\log_2$ function, Jensen's inequality says $-\log_2 E[X] \leq E[-\log_2 X]$.

- Since $-\log_2$ is strictly convex on its domain, the inequality above is strict.
- Hence, as long as the both child nodes have at least one instance,

$$H_S - (\frac{m_l}{m} H_l + \frac{m_r}{m} H_r) > 0.$$

- This implies that as long as the parent node has more than one instances, there is always a way to split the instances into the two child nodes, and hence, the splitting of the decision tree does not get stuck.

Overfitting – Pre-training Solutions

As the tree depth increases, a decision tree can often overfit. Solutions: set hyperparameters to restrict the tree. This is also known as **regularization**.

- `max_depth`: the maximum tree depth.
- `min_samples_split`: the minimum number of instances required to split an internal node.
- `min_impurity_decrease`: A node will be split if this split induces a decrease of the impurity greater than or equal to this value.
- `min_samples_leaf`: the minimum number of instances required to be at a leaf node. A split point at any depth will only be considered if it leaves at least `min_samples_leaf` training instances in each of the left and right branches.

Post-training Solutions – Tree Pruning

Minimal Cost-Complexity Pruning:

- Define a cost-complexity measure $R_\alpha(T)$ of a given tree T:

$$R_\alpha(T) = R(T) + \alpha|L(T)|$$

  - $\alpha \geq 0$: the complexity parameter
  - $L(T)$: the set of leaf nodes of tree T
  - $R(T)$: the total sample weighted impurity of the leaf nodes

- The cost-complexity measure of a single node t is defined as

$$R_\alpha(t) = R(t) + \alpha,$$

  where $R(t)$ is the impurity at node t.

- Let T_t be the tree rooted at node t.

- In general $R(T_t) < R(t)$. (Think about information gain.) However, one can find a large enough α such that

$$R_\alpha(T_t) = R(T_t) + \alpha|L(T_t)| = R(t) + \alpha = R_\alpha(t).$$

  The α that achieves the equality is called the effective α for node t, denoted by $\alpha_{\text{eff}}(t)$. That is

$$\alpha_{\text{eff}}(t) = \frac{R(t) - R(T_t)}{|L(T_t)| - 1}.$$

  An internal node with the smallest value of $\alpha_{\text{eff}}(t)$ is the weakest link and will be pruned first. (For such a node, there is not a lot of information gain by growing the tree from that node on.)

- Algorithm: After the initial decision tree is built,

  0. sort the internal nodes in increasing order of α_{eff};
  1. find the internal node t with the smallest value of $\alpha_{\text{eff}}(t)$;
  2. if $\alpha_{\text{eff}}(t) \geq$ `ccp_alpha`(a hyperparameter), break the loop;

3. otherwise, prune all its descendant nodes of t;
   re-calculate $R_\alpha(T_{t'})$ for each ancestor node t' of node t, update $\alpha_{\text{eff}}(t')$, and update the sorted list of α_{eff};
4. repeat from step 1.

- Effects of different values for `ccp_alpha`:

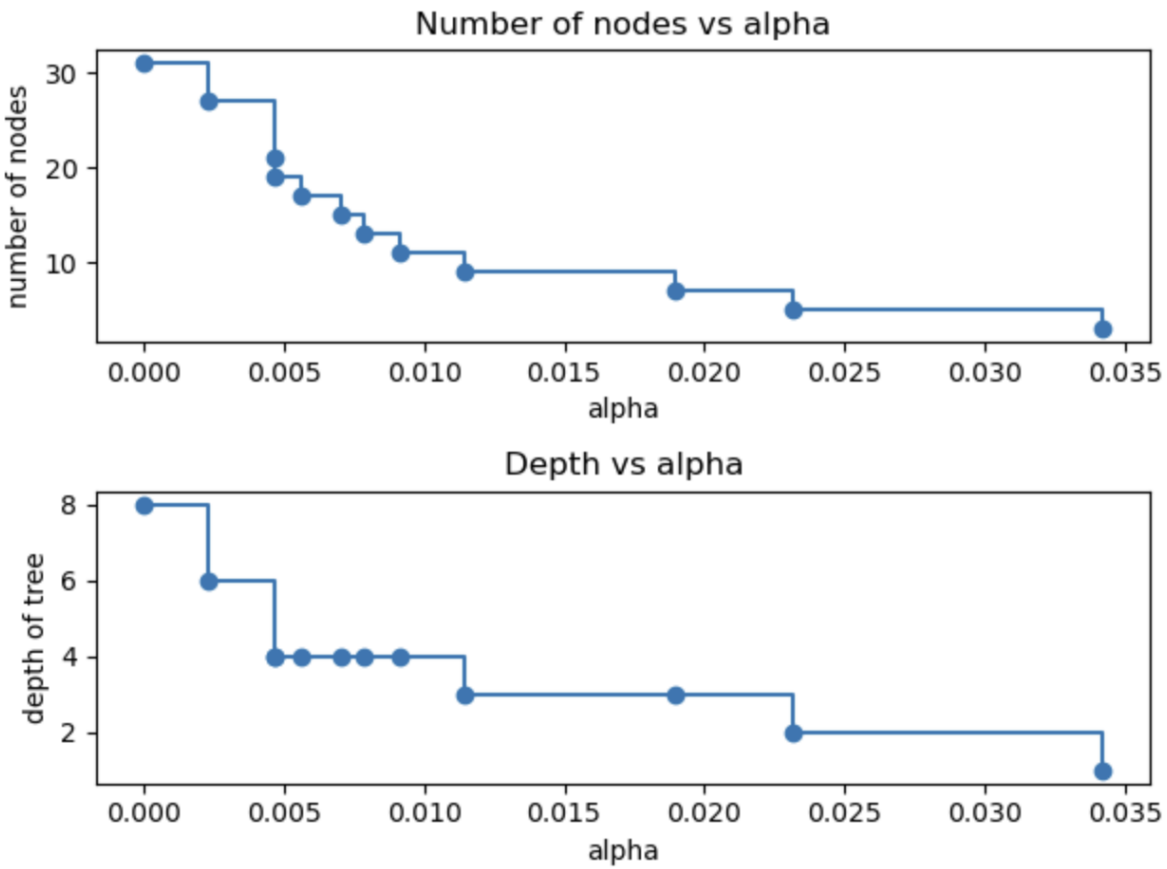


Figure 2: Tree characteristics after pruning up to the `ccp_alpha` value (labeled as 'alpha' on the horizontal axis). This example is based on a different dataset.

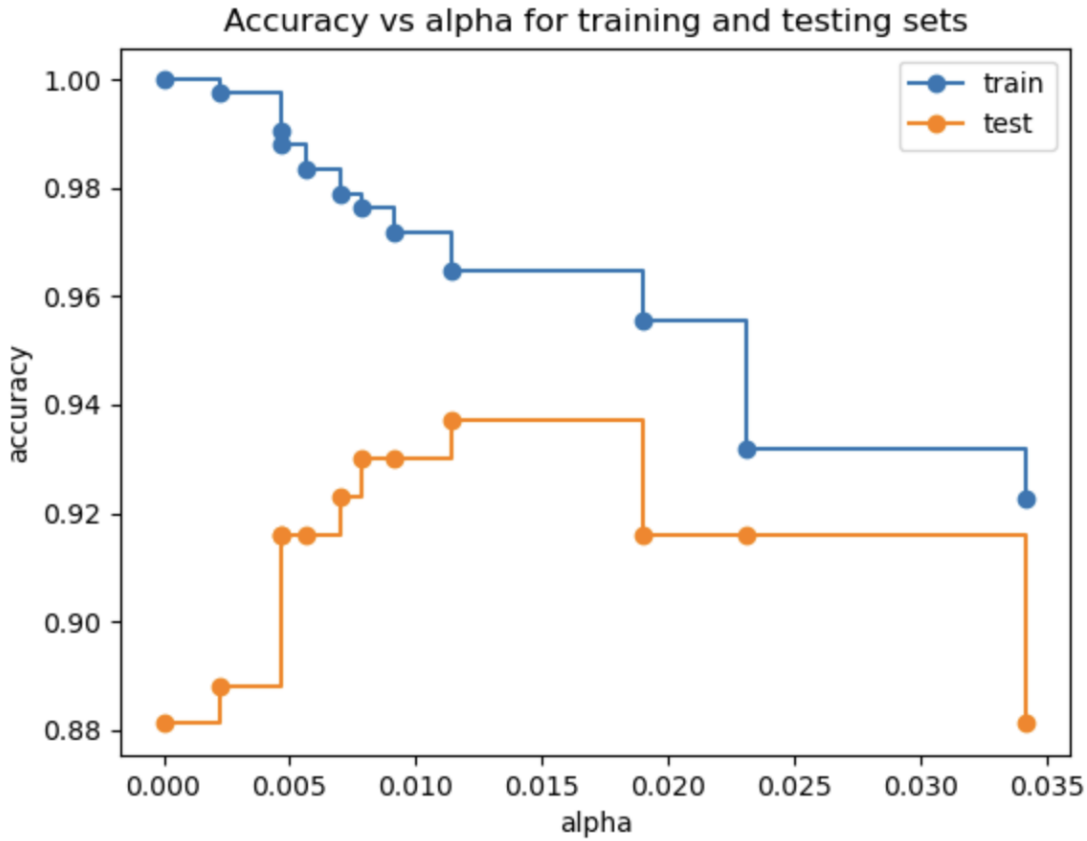


Figure 3: Training and testing accuracies after pruning up to the `ccp_alpha` value (labeled as 'alpha' on the horizontal axis).

- In fine-tuning your model, you can explore setting different values for `ccp_alpha`.

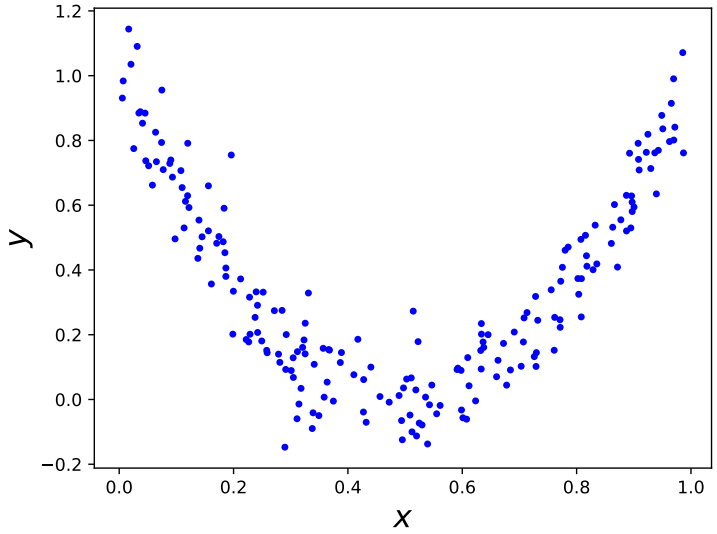
There are other tree-pruning strategies.

- Reduced-Error Pruning: Greedily remove nodes based on validation set performance.

  This generally improves performance but can be problematic for limited data set sizes.

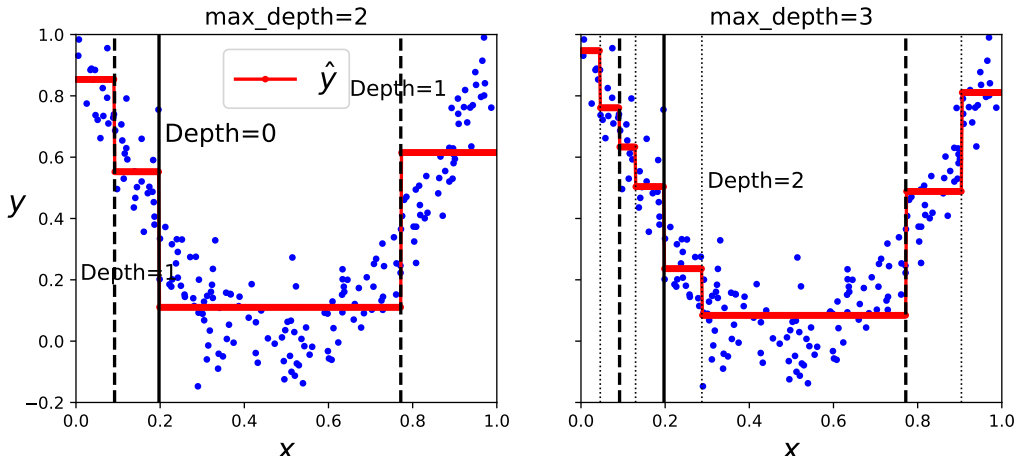
Decision Tree for Regression

We will consider the following dataset for regression.



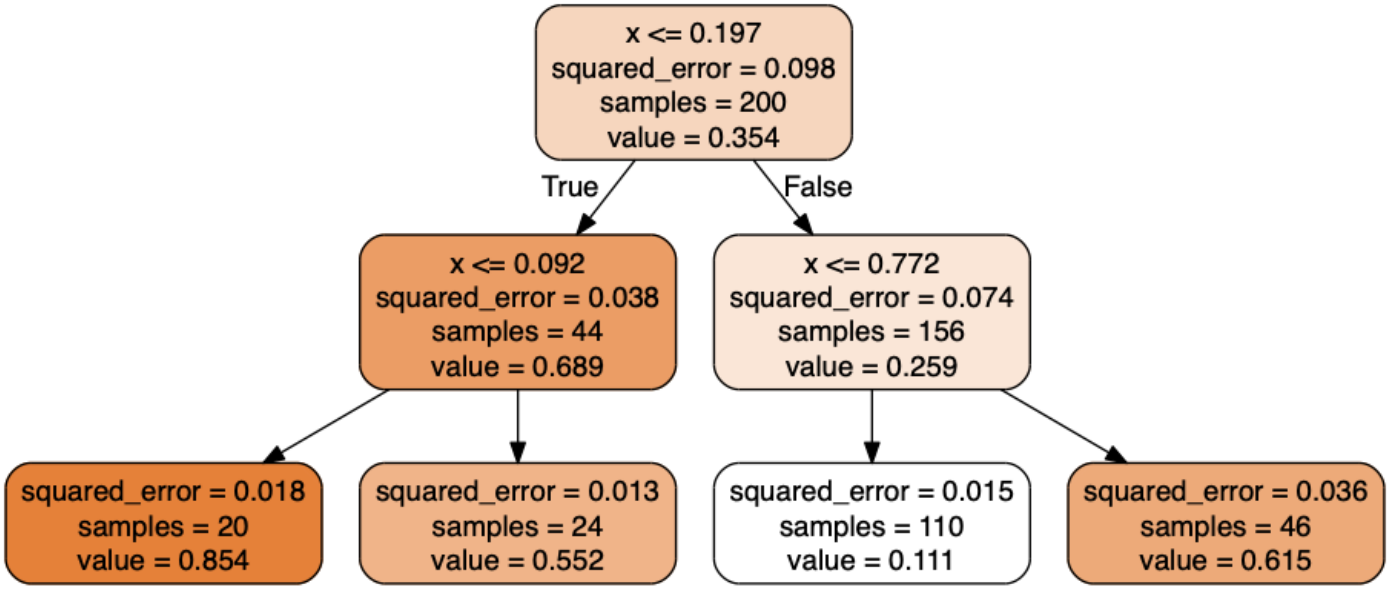

```
from sklearn.tree import DecisionTreeRegressor

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

Here are the prediction results for two decision tree models with different `max_depth`.



- We see that the domain for x is split into different regions.
- The predicted value in each region is a constant, which is the average of the output values of all the training instances falling in that region.
- Increasing the tree depth generally leads to more regions, and therefore, better precision (a smaller squared error) in approximating the output values.

Here is the decision tree with `max_depth` $= 2$.

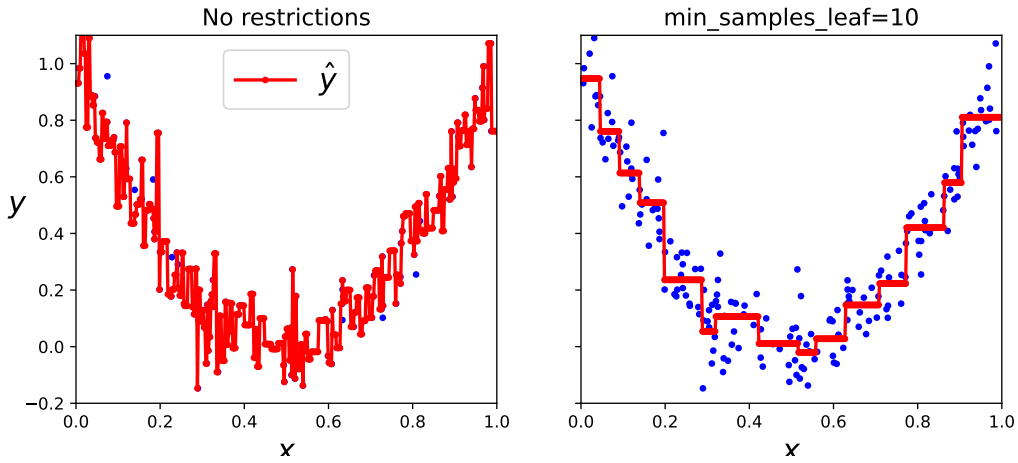


- Instead of predicting a class in each node, it predicts a value.
- Example: Suppose $x = 0.6$. Go down the tree and you get $\hat{y} = 0.111$.
- This prediction is the average target value of the 110 training instances associated with this leaf node, and it results in a mean squared error equal to 0.015 over these 110 instances.

CART algorithm for regression: At each node, it tries to split the falling into that node in a way that minimizes the sample-weighted MSE:

$$\frac{m_l}{m}\text{MSE}_l + \frac{m_r}{m}\text{MSE}_r.$$

Without regularization, decision trees are prone to overfitting when dealing with regression tasks.



Just setting `min_samples_leaf`$= 10$ results in a much more reasonable model.

Instability

- Decision trees tend to have decision boundaries perpendicular to axes (e.g., $x_1 \leq 1.75$ or not), which makes them sensitive to training set rotation.

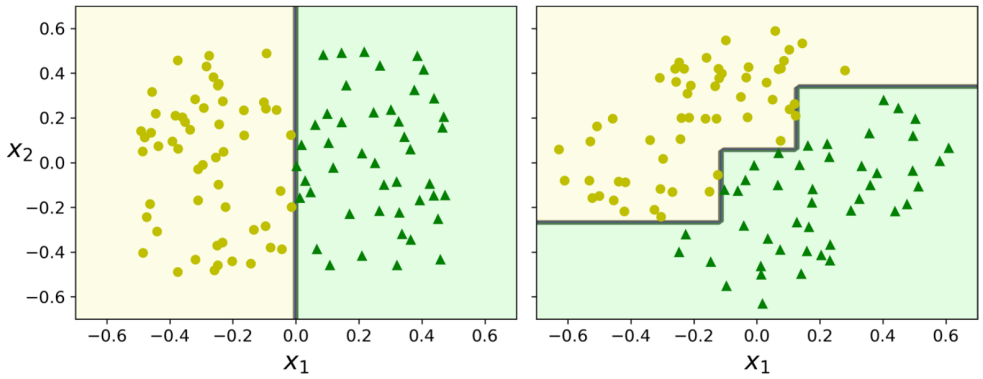


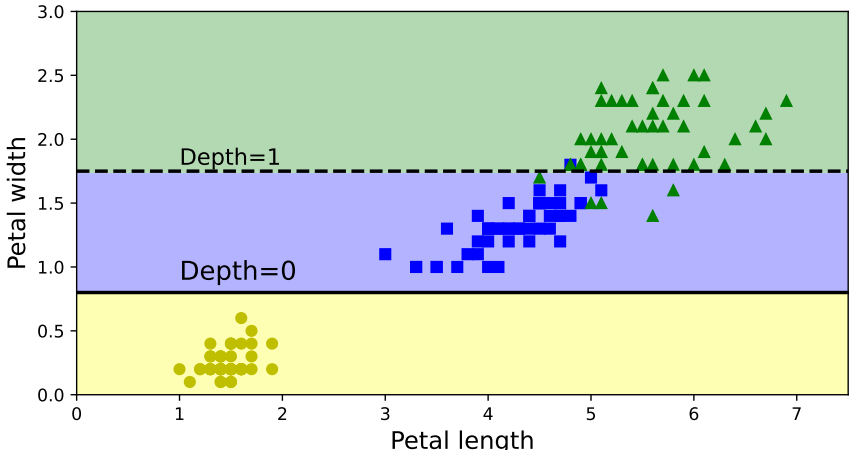
Figure 4: A binary classification problem with decision tree. Left: decision boundary under the original dataset. Right: after the dataset is rotated by $45°$, the decision boundary looks unnecessarily complicated, which may not generalize well.

- One way to limit this problem is to use Principal Component Analysis, which often results in a better orientation of the training data.
- More generally, the main issue with decision trees is that they are very sensitive to small variations in the training data or other random factors.
  - 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 a very different decision tree.
  - As another example, the training algorithm used by Scikit-Learn may involve randomness. If the improvement in the performance criterion is identical for several splits, then one split is selected randomly.

    Because of that, you may get very different models even on the same training data, unless you set the `random_state`

hyperparameter.

The following shows the decision boundaries for the iris classification problem when we set the decision tree hyperparameters to `max_depth=2,random_state=40`.



- Random Forests can limit instability by averaging predictions over many trees, as we will see later.

Practical Tips about using Decision Trees

See more at: https://scikit-learn.org/stable/modules/tree.html#tree

- Decision trees tend to overfit on data with a large number of features. Getting the right ratio of samples to number of features is important.
- Consider performing dimensionality reduction (PCA, ICA, or feature selection) beforehand to give your tree a better chance of finding features that are discriminative.
- Visualize your tree as you are training by using the export function. Use `max_depth=3` as an initial tree depth to get a feel for how the tree is fitting to your data, and then increase the depth.
- Remember that the number of samples required to populate the tree doubles for each additional tree level. Use `max_depth` to control the size of the tree to prevent overfitting.

- Use `min_samples_split` or `min_samples_leaf` to ensure that multiple instances inform every decision in the tree. Try `min_samples_leaf=5` as an initial value.
- ...