Module 14: Decision Trees

Quick Reference Guide

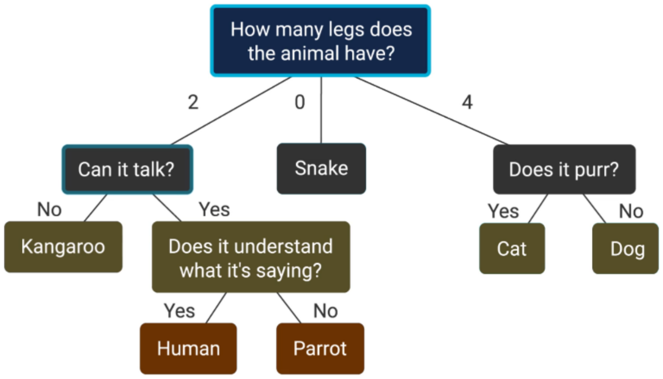
Learning Outcomes:

1. Identify the correct statements regarding decision trees and their outputs
2. Build a decision tree manually.
3. Articulate the correct inputs for building decision trees using scikit-learn and visualizing them using Graphiz.
4. Train a decision tree model with desired hyperparameters using scikit-learn.
5. Visualize a decision tree.
6. Evaluate decision tree splits.
7. Evaluate overfitting of decision trees.
8. Measure the impurity of a decision tree using entropy.
9. Compare the performance of different grid search algorithms.
10. Implement the decision tree algorithm.

Introduction

A **decision tree** is a **tree of questions** that must be **answered in a sequence** to produce a **predicted classification**.

For instance, this is what a decision tree classifier for animals may look like:

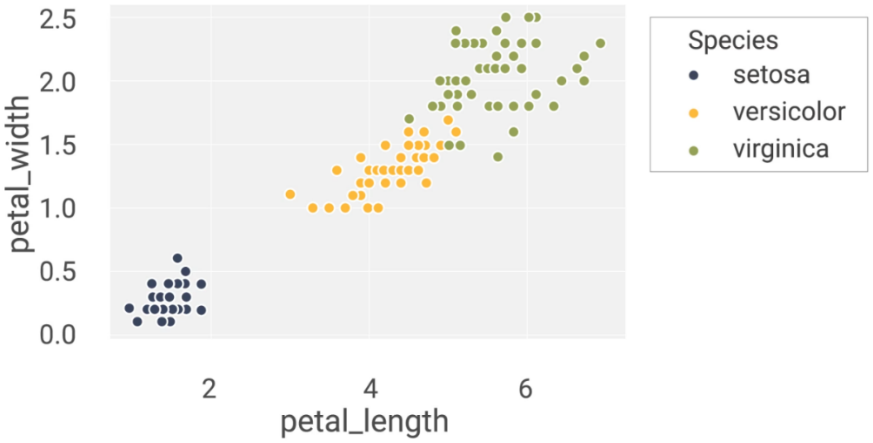


This module focuses on decision trees based on a flower dataset, consisting of 150 flower measurements of three different species of Iris — versicolor, setosa, and virginica.

sepal_length	sepal_width	petal_length	petal_width	species
5.5	2.5	4.0	1.3	versicolor
6.4	2.9	4.3	1.3	versicolor
4.8	3.4	1.6	0.2	setosa
5.3	3.7	1.5	0.2	setosa
6.7	2.5	5.8	1.8	virginica

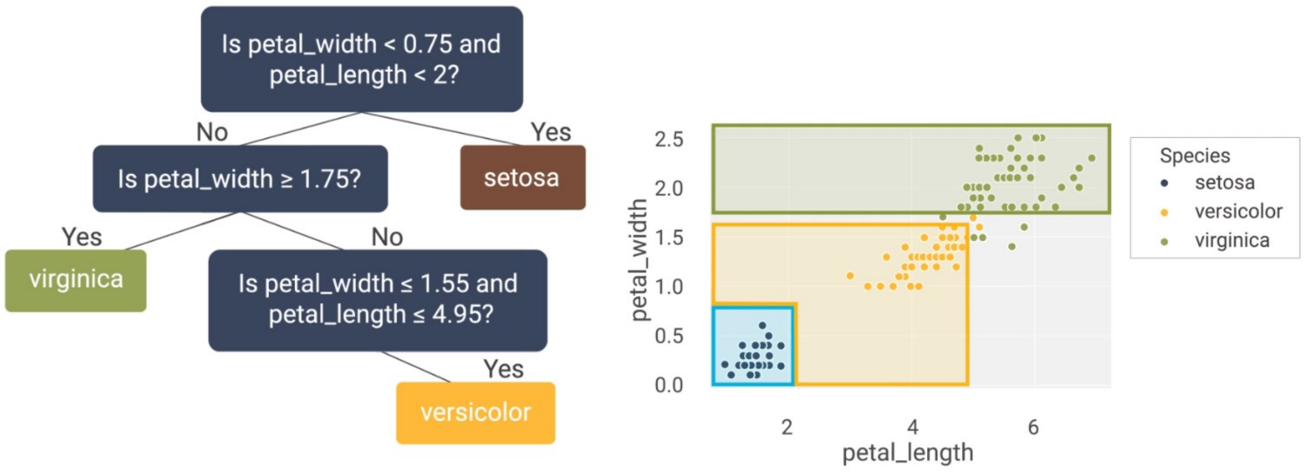
Building Decision Trees Visually

You can **build a decision tree manually** by referencing a **visual representation** of data like a scatterplot.



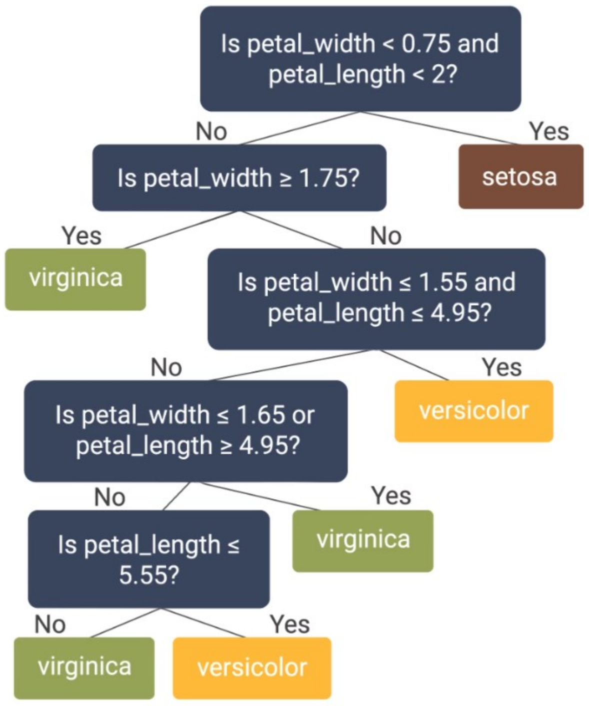
For instance, if you have a scatterplot representing a dataset of the petal width and petal length of different types of Iris flower species, you can **utilize the presented data to construct a decision tree** that predicts classifications based on 'yes' or 'no' answers.

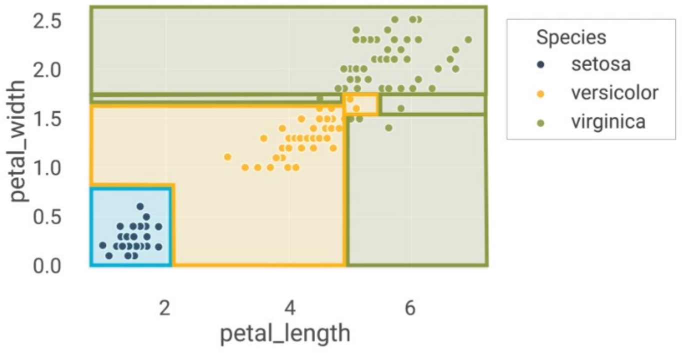
These answers will be based on a set of rules you define, with more rules being added at each step in a sequence. Specifying different rules with regard to petal width and length will yield different answers in terms of the species classification and their boundary within the scatterplot.



In this example, if the rules set limit the petal width to less than 0.75 and the petal length to less than 2, a 'yes' answer will indicate the setosa species. If the answer is 'no,' you can add other **arbitrary rules at each level** of the decision tree to determine the classification.

At each level, the rules may need to be more **restrictive in scope**, in order to **differentiate** between species that appear in close proximity on the scatterplot.


Is petal_width < 0.75 and petal_length < 2?
No
Yes
setosa
Is petal_width ≥ 1.75?
Yes
virginica
No
Is petal_width ≤ 1.55 and petal_length ≤ 4.95?
No
Yes
versicolor
Is petal_width ≤ 1.65 or petal_length ≥ 4.95?
No
Yes
virginica
Is petal_length ≤ 5.55?
No
virginica
Yes
versicolor



However, there seems to be a **training error** on the dataset for this model:

- The fact that the decision tree gets every single data point correct, indicates that it is **prone to overfitting**.
- The **decision boundary** is **arbitrary** and complicated, and if other data observations were added that the model was not trained on, the model would make significant errors.

Building Decision Trees in Scikit-Learn

Fitting a decision tree model in scikit-learn is similar to fitting any other scikit-learn estimator.

First, you create a **DecisionTreeClassifier** object and call the **fit** function on the data.

from sklearn import tree

decision_tree_model = tree.DecisionTreeClassifier(criterion='entropy')

decision_tree_model = decision_tree_model.fit(iris_data[[“petal_length”, “petal_width”]], iris_data[“species”])

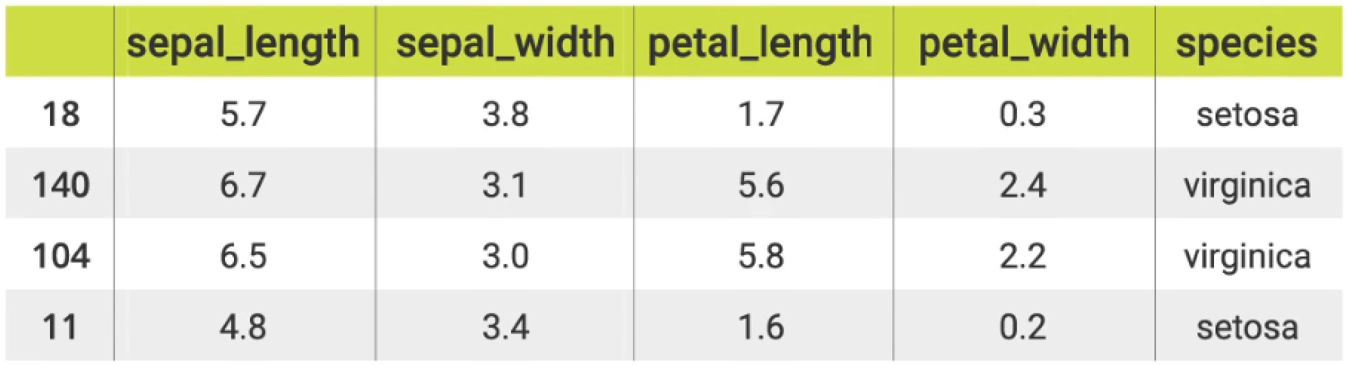
Next, you call the **predict()** function on an observation or a data frame of observations.

decision_tree_model.predict(four_random_rows[["petal_length", "petal_width"]])

For example, if your data sample consists of a table of four rows of observations about species of Iris flowers:

four_random_rows = iris_data.sample(4)

four_random_rows



	sepal_length	sepal_width	petal_length	petal_width	species
18	5.7	3.8	1.7	0.3	setosa
140	6.7	3.1	5.6	2.4	virginica
104	6.5	3.0	5.8	2.2	virginica
11	4.8	3.4	1.6	0.2	setosa

This will return the predicted class for each observation:

```
array([‘setosa’, ‘virginica’, ‘virginica’, setosa’], dtype=object)
```

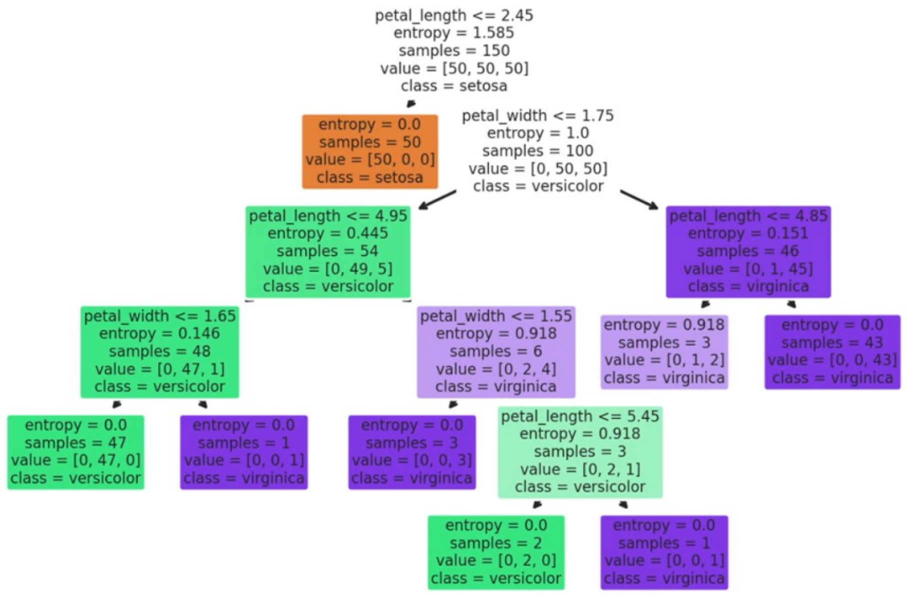
Scikit-learn enables you to automatically **generate a decision tree** diagram consisting of rules, using the **plot_tree()** function.

Along with the data model, feature names, and class names, the function allows you to specify aesthetic arguments like whether to display classifications as rounded or filled.

For example:

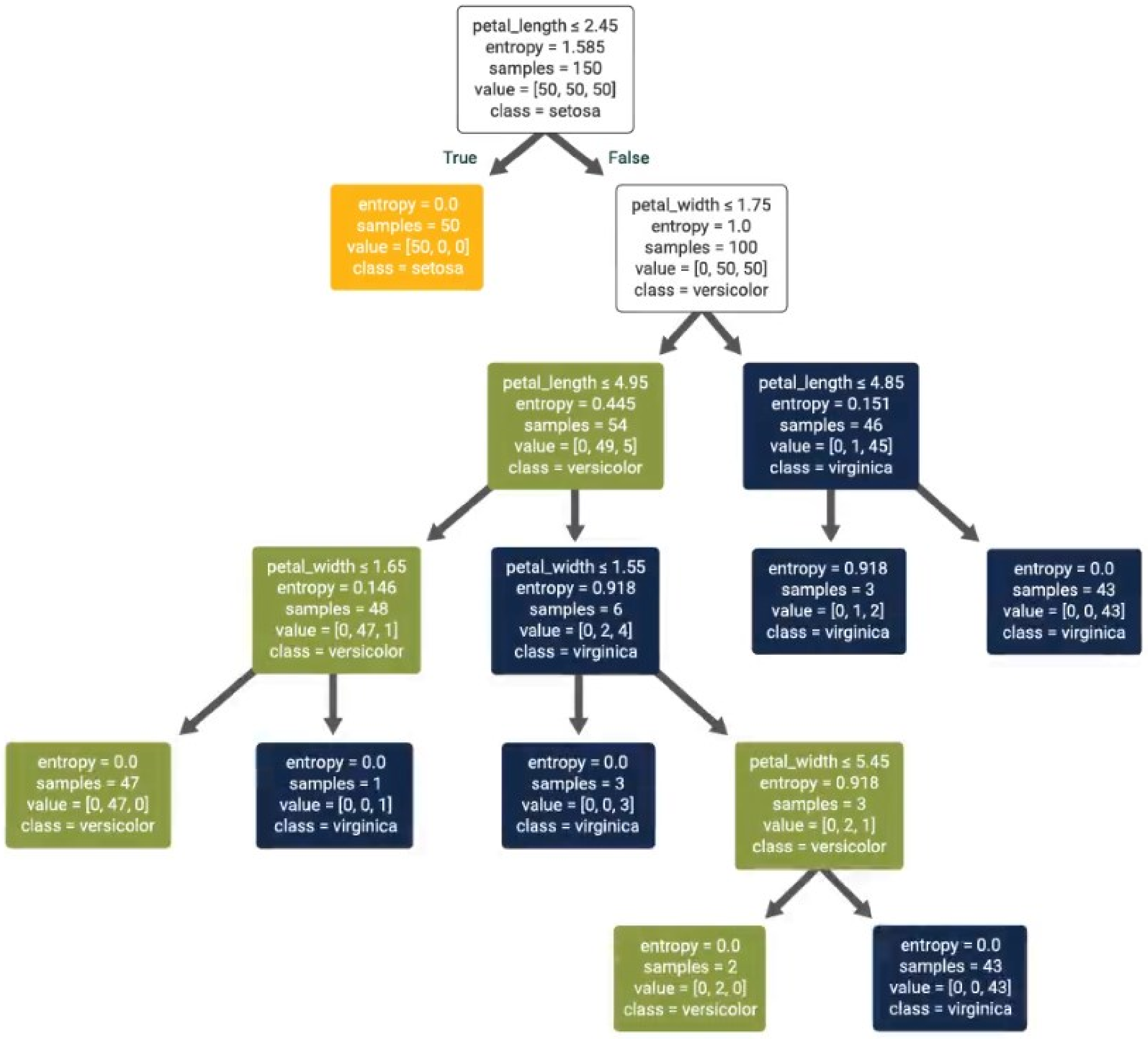
```
tree.plot_tree(decision_tree_model, feature_names = [“petal_length”,
“petal_width”],
                  class_names = [“setosa”, “versicolor”, “virginica”],
                   rounded = True, filled = True)
```

The default decision tree visualizer will return something like this



A better library for generating a decision tree is **export_graphviz**, which uses the **AT&T Graphviz library** and provides a cleaner tree layout. The code is:

```
import graphviz

dot_data = tree.export_graphviz(decision_tree_model, out_file=None,

                                feature_names=[ [“petal_length”, “petal_width”],

                                class_names=[“setosa”, “versicolor”, “virginica”],

                                filled=True, rounded=True)

graph = graphviz.Source(dot_data)

graph.render(format="png", filename="iris_tree")

graph
```



Scikit-learn decision trees only consider one feature at a time — that is why either petal_length or petal_width is displayed inside each node of the decision tree, not both simultaneously.

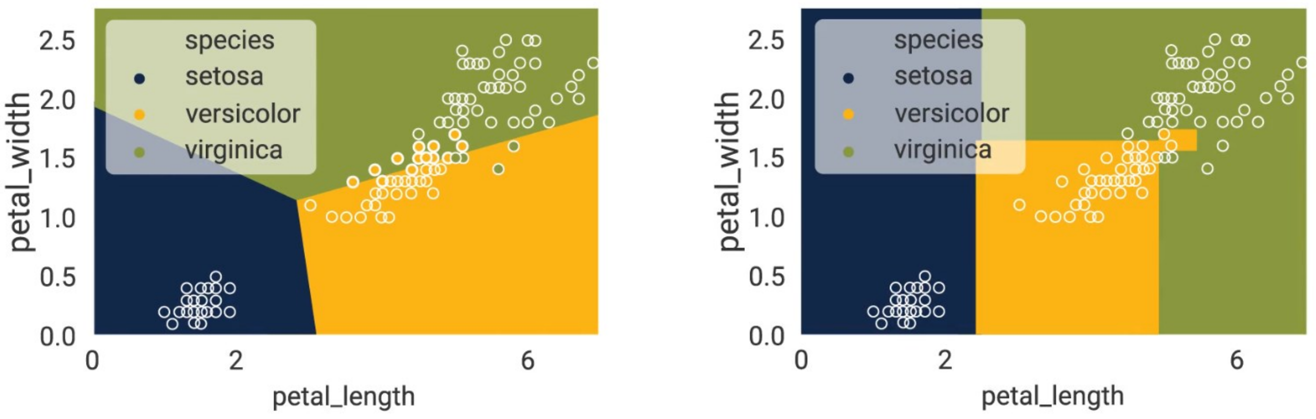
Each node contains certain information, for instance:

- **Sample**: the number of data samples

- **Value**: the number of samples in each class that remain after applying all of the rules above the current node
- **Class**: the most likely class, based on the information in the previous nodes

The **color of the node** reflects the class, as well as the likelihood of that class — the darker the color, the more confident the model is of the classification.

Consider this **boundary comparison** of a scikit-learn logistic regression model versus a decision tree model:



Unlike the logistic regression model, the conclusion is that for decision trees:

- Boundaries are non-linear
- They can achieve 100% accuracy

Measuring Our Decision Tree’s Training Error

The **accuracy_score()** function can be used to assess the accuracy of a model.

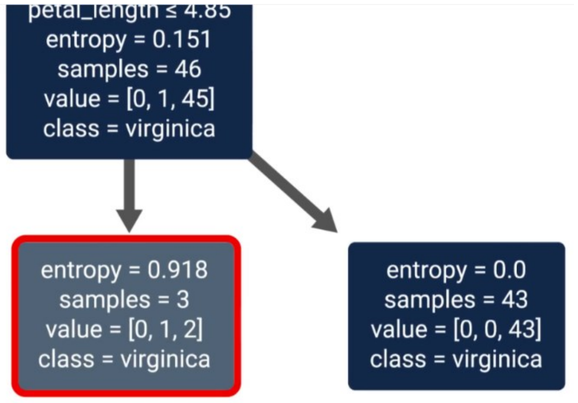
For instance, the decision tree model's predictions can be compared to the actual species in the Iris dataset using this code:

from sklearn.metrics import accuracy_score

predictions = decision_tree_model.predict(iris_data[["petal_length", "petal_width"]])

accuracy_score(predictions, iris_data["species"])

In this example, the score is returned as 99.3% accuracy. The reason why the decision tree does not have 100% accuracy, is because it contains an **impure node** — a node that contains three samples from different classes — since scikit-learn did not create a rule to differentiate the classes.



To understand why this happened, you use the **query()** function and include all the decision tree rules.

iris_data.query("petal_length > 2.45 and petal_width > 1.75 and petal_length < 4.85")

This returns a dataframe with three datapoints in it. And, in this case, it seems that the species could not be differentiated, since the length and width parameters of the flower was exactly the same for all three species.

	sepal_length	sepal_width	petal_length	petal_width	species
70	5.9	3.2	4.8	1.8	versicolor
126	6.2	2.8	4.8	1.8	virginica
138	6.0	3.0	4.8	1.8	virginica

Scikit-learn decision trees always have perfect accuracy on the training data, except when there are samples from different classes with the exact same features.

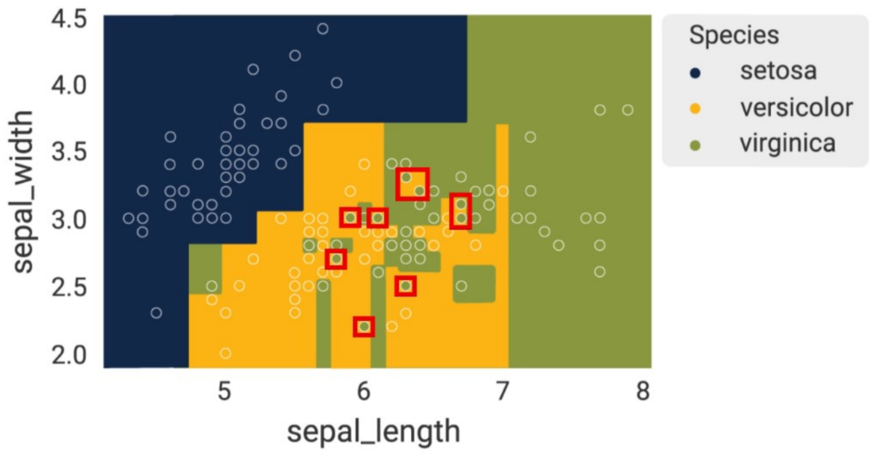
Decision Tree Overfitting

You can test for overfitting by:

- Observing the decision boundaries
- Looking at the decision tree diagram

Decision boundary observation

The scatterplot below displays the correlations between species' sepal width and length, instead of the petal width and length from the Iris dataset, as before.



Here it is obvious that overfitting has occurred. The decision boundaries indicated in the scatterplot appear **erratic** and the data points for the classes **overlap** at certain points, leading to some misclassifications. This means that 100% accuracy cannot be achieved.

Decision tree diagram observation

Including a **large number of features** in a model can lead to overfitting, but that is not always the case. It is best to generate and compare different decision trees to see which data leads to the most accurate predictions.

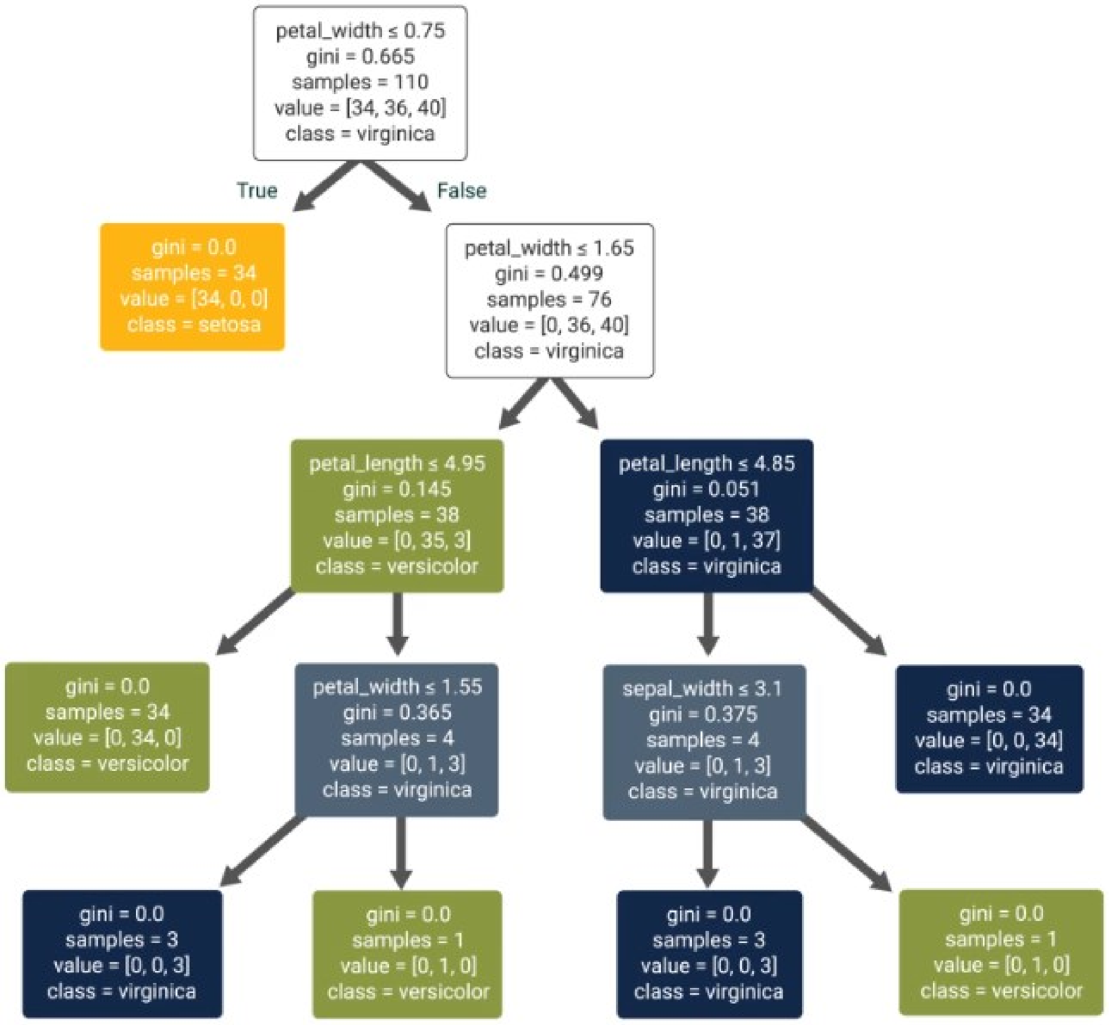
For example, if you incorporate all the measurements in the Iris species dataset (the petal and sepal width as well as length), it will not necessarily lead to overfitting. Instead, including all of the data can sometimes lead to a more accurate outcome by resolving small differences between classes that were not sufficiently addressed in other models.

The code for fitting a four-dimensional model in this example is:

decision_tree_model_4d = tree.DecisionTreeClassifier()

decision_tree_model_4d = decision_tree_model_4d.fit(train_iris_data[[“petal_length”, “petal_width”, “sepal_length”, “sepal_width”]], train_iris_data[“species”])

This generates the following the decision tree, which is more accurate than the tree using only petal or sepal measurements.



An Intuitive Look at Decision Tree Boundaries

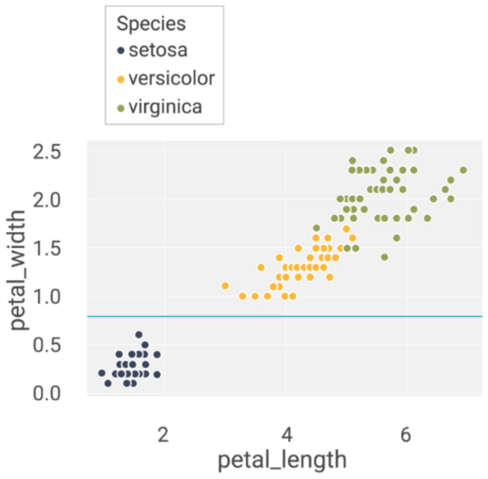
In the traditional decision tree generation algorithm, all the data starts at the root node.

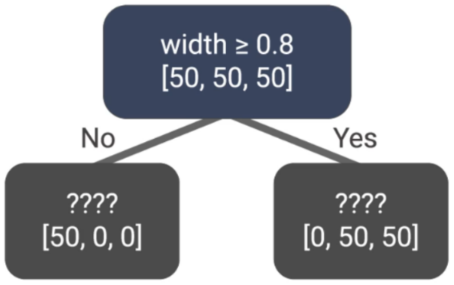
The following process is repeated until every node is either pure or unsplittable:

- Pick the best feature, x, and the best split value, β
- Split data into two nodes — one where $x < \beta$, and one where $x \geq \beta$

A **pure node** is defined as only having samples from one class. By comparison, an **unsplittable node** has overlapping data points from different classes.

Defining what is meant by a '**best split**' depends on the data you are working with. It is often done intuitively, and involves finding splits where the least amount of overlap occurs.





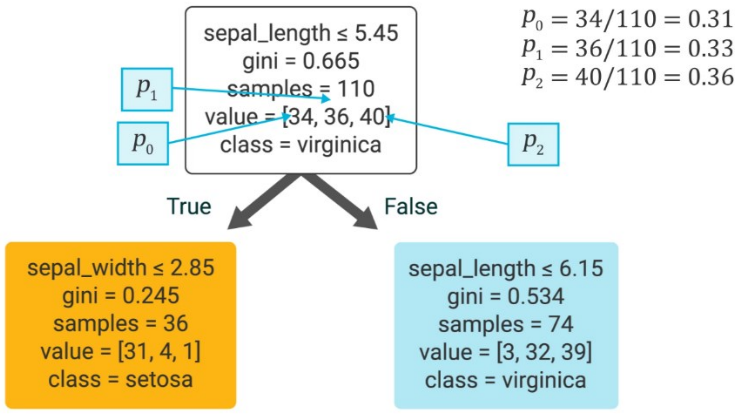
An algorithm that can automatically conduct this process of considering different splits is needed.

Entropy

Node entropy is a concept that can be used to determine best splits.

In this instance, the proportion of data points in a node with label C can be defined as p_c.

For example, in this decision tree node:



Next, you can define the entropy, S, of a node with this equation:

$$S = -\sum_{c} p_c \log_2 p_c$$

For example:

$$S = -0.31 \log_2 0.31 - 0.33 \log_2 0.33 - 0.36 \log_2 0.36$$

$$S = 0.52 + 0.53 + 0.53$$

$$S = 1.58$$

Therefore, in this example, the entropy is 1.58.

Entropy basically refers to how unpredictable a node is:

- **Low entropy**—more predictable
- **High entropy**—less predictable

For example, since the root node will give you very little certainty of what class the sample belongs to, it will have relatively high entropy. As you work your way down the nodes in a decision tree, the entropy will become less and less, as you gain more information about the likelihood of a sample belonging to a certain class. In nodes where all the samples belong to the same class, the entropy will become 0.0.

There are **four entropy properties**:

- A node where all data that is part of the same class has **zero entropy**

$$-1\ \log_2 1 = 0$$

- A node where data evenly split between two classes has **entropy 1**

$$-0.5\ \log_2 0.5\ - 0.5 \log_2 0.5 = 1$$

- A node where data evenly split between 3 classes has **entropy 1.58**

$$3 \times (-0.33 \log_2 0.33) = 1.58$$

- A node where data evenly split into C classes has **entropy log$_2$C**

$$\mathrm{C} \times (-1/\mathrm{C} \log_2 1/\mathrm{C}) = -\log_2 1/\mathrm{C} = \log_2 \mathrm{C}$$

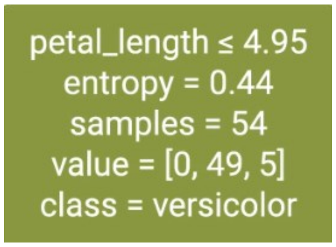
Using Entropy to Select Decision Trees

Scikit-learn uses a concept known as **weighted entropy** in its decision-making process. The weighted entropy of a node is its entropy scaled by the fraction of samples in that node. Weighted entropy is sometimes represented by the symbol WS, since S represents entropy.

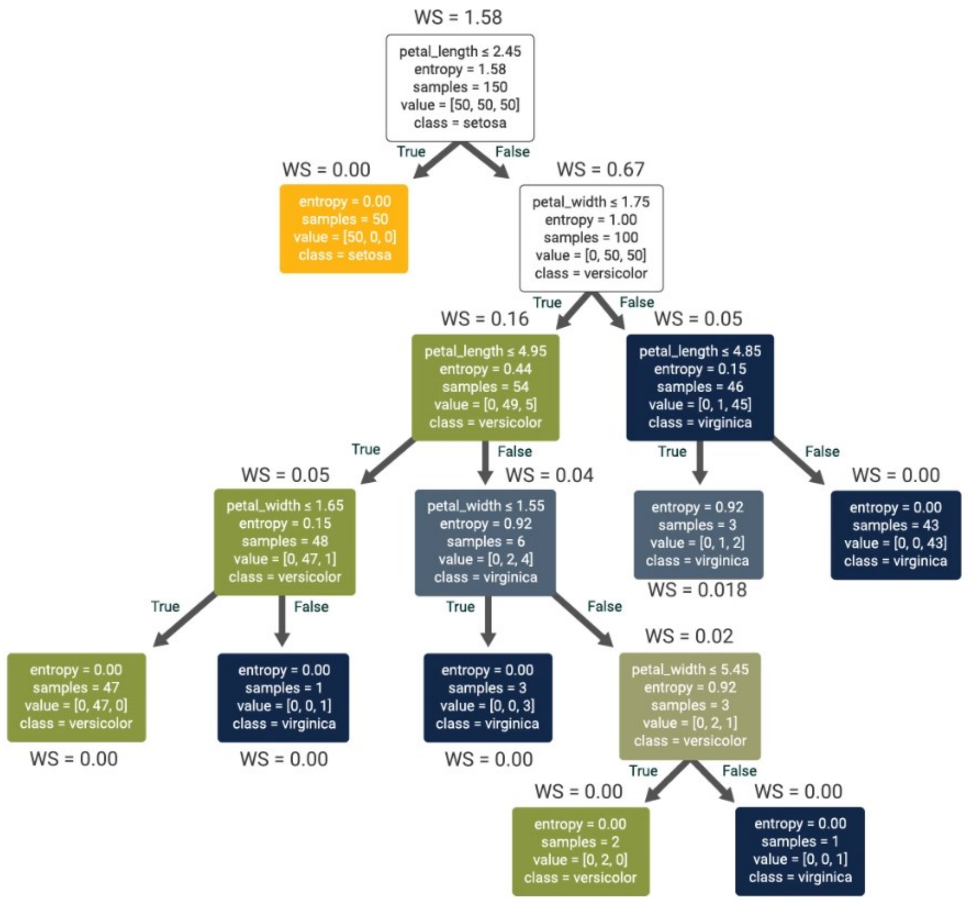
If you take this node with 54 samples, for example, the equation to establish the weighted entropy, or WS, will be:

$$54 \div 150 \times 0.445 = 0.16$$

$$WS = 0.16$$



The WS value decreases as more certainty about the classification is gained at each level further down the decision tree.

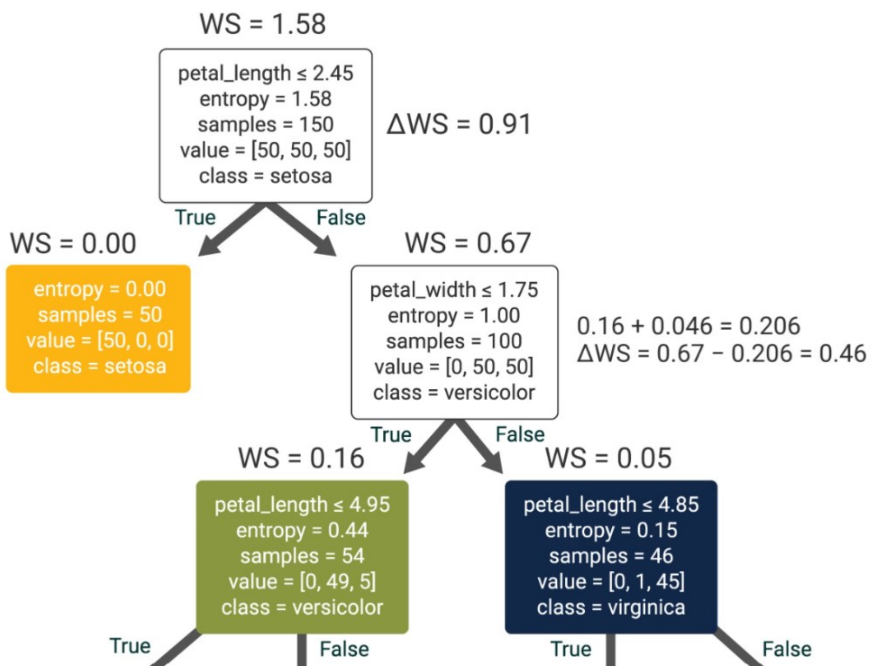


Each **decision rule** will **reduce the weighted entropy** by a certain amount.

For example, the root node has a WS of 1.58. The left node below it has WS of zero, and the right node has a WS of 0.67. The reduction in WS can be presented as follows:

$$\Delta\, WS = 1.58 - 0.67 = 0.91$$

As you move down, there are now two nodes that have weighted entropy values attached to the nodes to consider, so the equation is as follows:



The goal is always to **build decision tree rules that will yield the largest reduction in entropy** — the **largest ΔWS** values. Whereas traditional decision tree algorithms follow an iterative process to decide which split is the best by picking the split with the largest reduction in entropy, the scikit-learn algorithm does this using equations when determining splits.

Restricting Decision Tree Complexity

In order to avoid running the risk of overfitting, the growth of a decision tree sometimes needs to be restricted in some way.

Approach 1: Preventing growth

There are some rules you can set to prevent full growth:

- Do not split nodes containing < 1% of the samples — the scikit-learn hyperparameter for this rule is **min_sample_split**
- Do not allow nodes in the decision tree to be more than seven levels deep — the scikit learn hyperparameter for this rule is **max_depth**
- Do not create any splits if the ΔWS is less than 0.01 — the scikit learn hyperparameter for this rule is **min_impurity_decrease**

Decision tree models in scikit-learn have a **large number of hyperparameters**, since there are so many ways to naturally control growth. Therefore, doing a full grid search of all possible combination can be **computationally expensive**.

Approach 2: Pruning

The 'pruning' approach means that you allow the decision tree to grow fully and **remove some of the less useful branches afterward**. You can single out branches of the tree for possible pruning by looking at the number of samples there are to differentiate.

One way of pruning is by using a **validation set** to see if keeping the branch is useful or not. You run the model on the validation set twice — you keep the branch the first time, and replace it with the most common prediction the second time. If there is no significant impact on the validation error, the split can be deleted.

When a node contains multiple samples from different classes, the **predict()** method will return the most common class. The **predict_proba() method** returns the fraction of samples that belong to that class.

Evaluating Predictive Performance Compared to Humans

The ability to use data in order to make better predictions, lies at the core of the current artificial intelligence or AI revolution. A key aspect of many models — like supervised learning — is the **focus on prediction**.

A key issue for designers of new prediction tools, is how to evaluate whether or not the model is an improvement on the status quo — for instance whether a robo-advisor can outperform current human advisors.

An interesting example of whether a certain problem is amenable to machine learning, is a study by Jon Kleinberg and co-authors in *The Quarterly Journal of Economics* that focuses on bail applications. The question here, is whether machine learning, ML, can adequately do the job of a judge in predicting whether a bail applicant is likely to reoffend.

The authors of the study gathered data over a five-year period starting in 2008 to build a prediction model in New York City. The study considered criteria for defendants such as: Prior offenses, current charges, the location of the offenses, and the age of the defendants. The gathered data was limited to the data that was available at the time the decision was made.

Using this data, they built a boosted tree prediction model. However, while the prediction model fit the observed crime rate, it did not fit the patterns observed in the judge’s bail application decisions. They found that the judges made good decisions amongst lower-risk defendants, but not with higher-risk defendants, especially those with prior felonies.

The problem is that there is no way of gathering data about what could have happened — for instance, whether those who were denied bail would have reoffended. So it is hard to know whether an algorithm would fare better than the judges.

What the authors did to adjust for this fact, is to focus on the most lenient judges and then compare their crime and incarceration rate to that of the less lenient judges. Using this technique, the authors were able to build quite an efficient model. However, there are many ethical and policy implications and considerations when it comes to allowing algorithms to deny a person’s bail application.