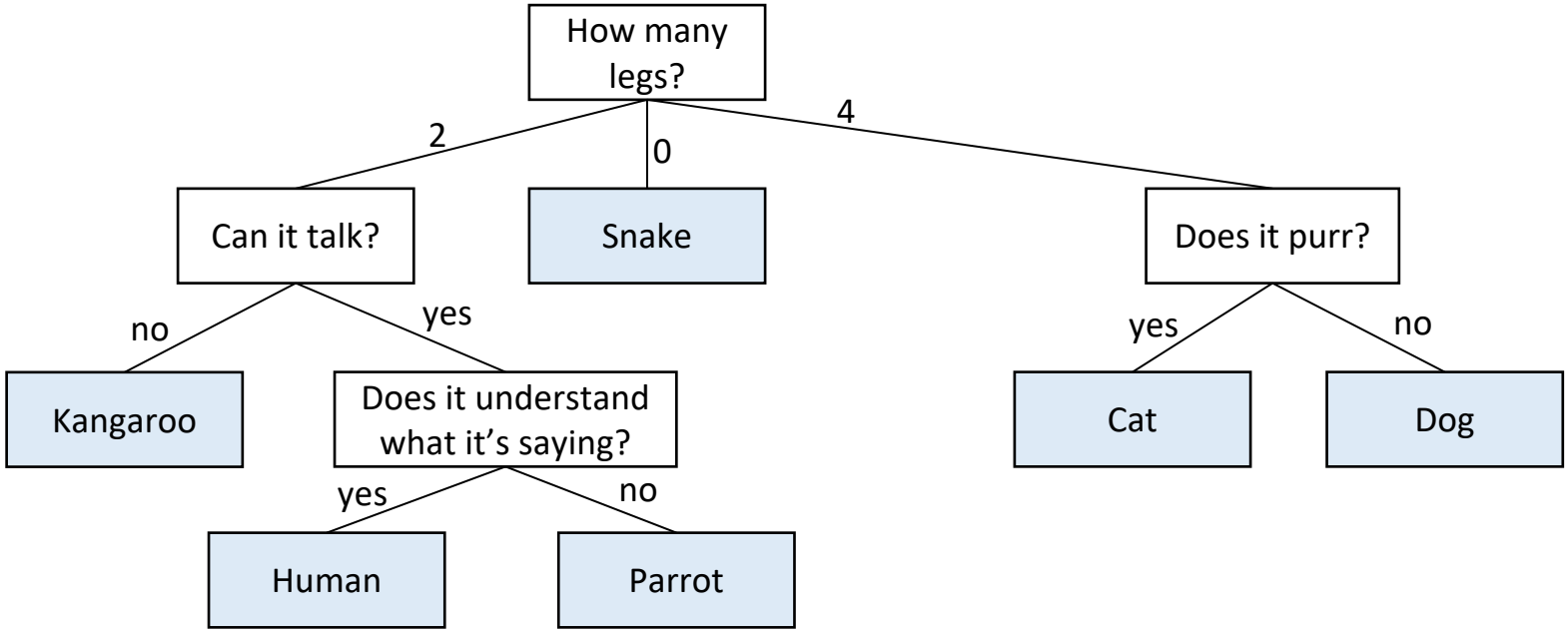
LECTURE 20

Decision Trees

Tree-based methods for classification and regression.

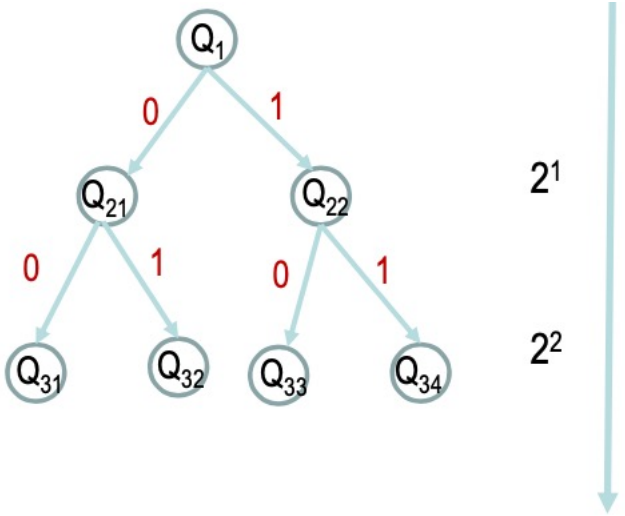
Decision Trees

A Decision Tree is a very simple way to classify data. It is simply a tree of questions that must be answered in sequence to yield a predicted classification.



The game of 20 questions

- The intuitive idea is similar to the game of 20-questions. The objective is to design an effective and balanced binary tree, each non-terminal node is a question, a test, or a rule, with 2 possible outcomes.



- After 20 questions, the tree will have about $2^{20} \approx 1\ million$ leaf nodes. Hopefully, any objects/concepts that a person can conceive belong to one of the leaf Node, and thus is localized.
- If a leaf note contains more than 1 object/concept, then they cannot be told apart.

Decision Tree Basics

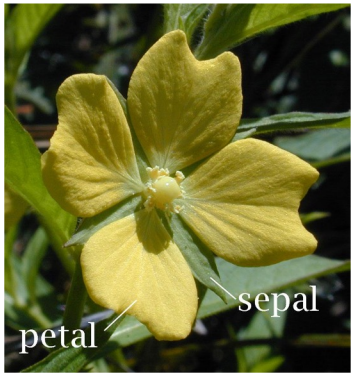
- **Decision Tree Basics**
- Decision Trees in scikit-learn
- Basic Decision Tree Generation
- Overfitting
- Restricting Decision Tree Complexity
- Random Forest

Example: Flower Classification

The Iris flower data set is a commonly used example:

- Created by statistician/biologist Ronald Fisher for his paper “The use of multiple measurements in taxonomic problems”.
- Data set consists of 150 flower measurements from 3 different species.
- For each, we have “petal length”, “petal width”, “sepal length”, “sepal width”.

Goal is to predict species from other data.



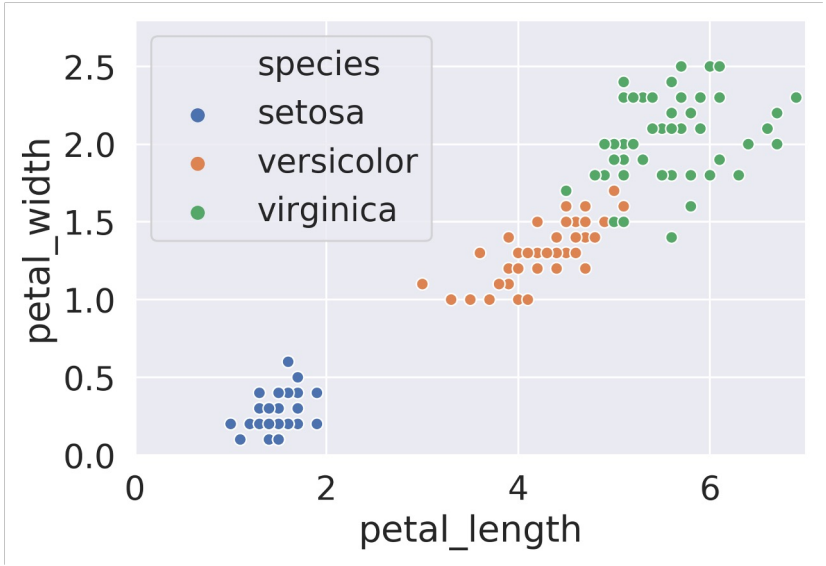
https://en.wikipedia.org/wiki/Sepal

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

Classifying Irises Using Linear Models

Example: Logistic Regression Using Petal Data Only

The plot below shows the width and length of the petals of each flower, with the species annotated in the form of color.



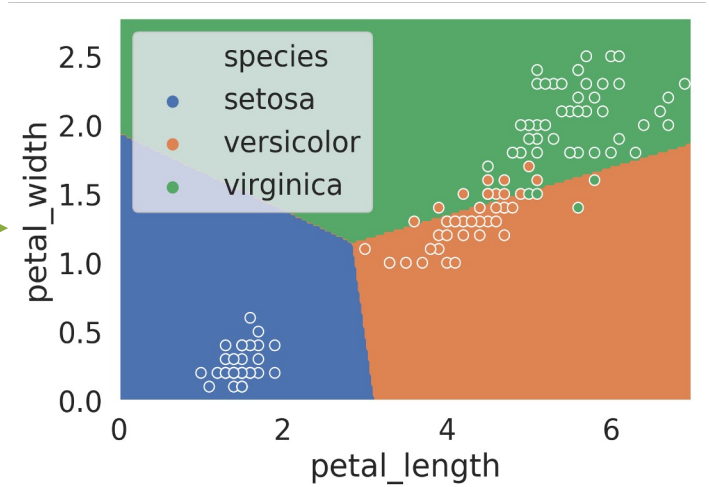
Example: Logistic Regression Using Petal Data Only

The plot below shows the width and length of the petals of each flower, with the species annotated in the form of color.

- If we train a 3 class logistic regression model in sklearn on this data, we end up with the model below.

```
from sklearn.linear_model import LogisticRegression
logistic_regression_model = LogisticRegression(multi_class = 'ovr')
logistic_regression_model = logistic_regression_model.fit(iris_data[["petal_length", "petal_width"]], iris_data["species"])
```

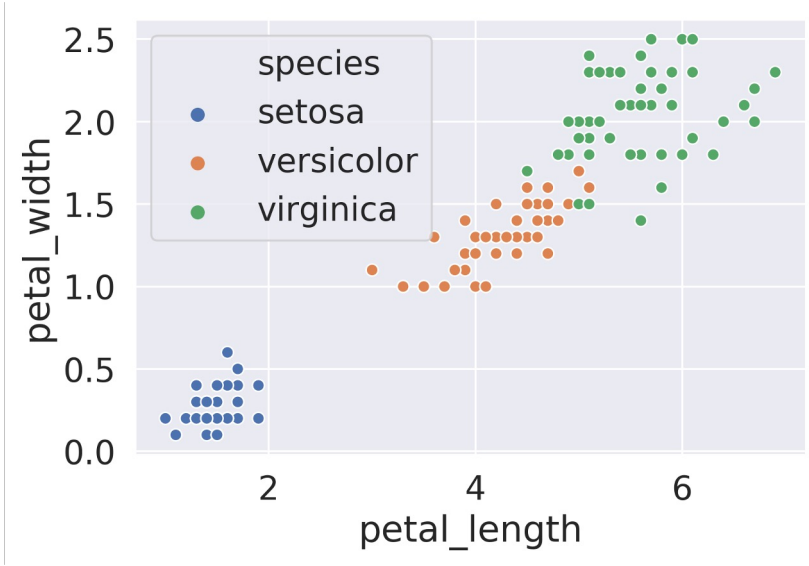
Because logistic regression models are linear, the decision boundaries are linear.



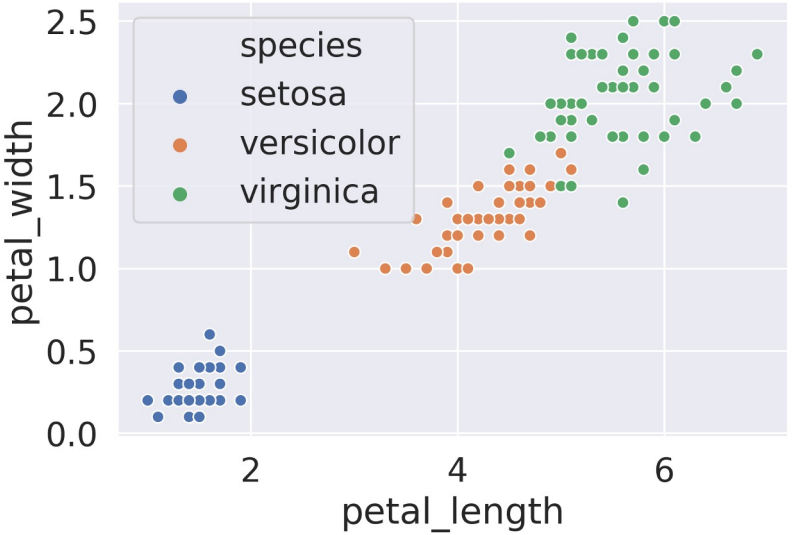
Example: Using Petal Data Only

The plot below shows the width and length of the petals of each flower, with the species annotated in the form of color.

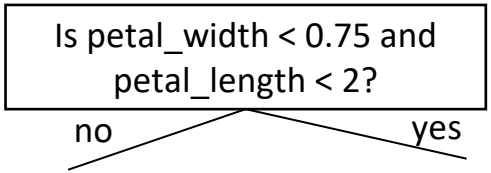
We can build a decision tree manually just by looking at this picture.

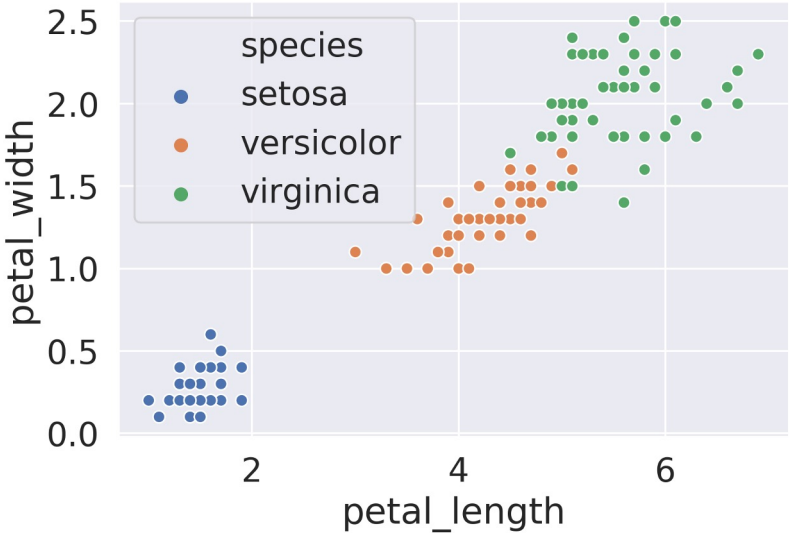


Example: Using Petal Data Only

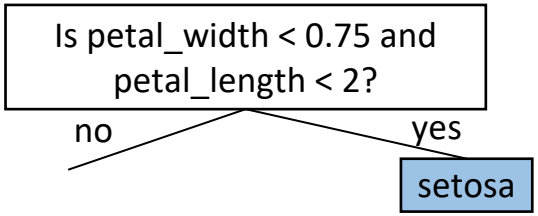


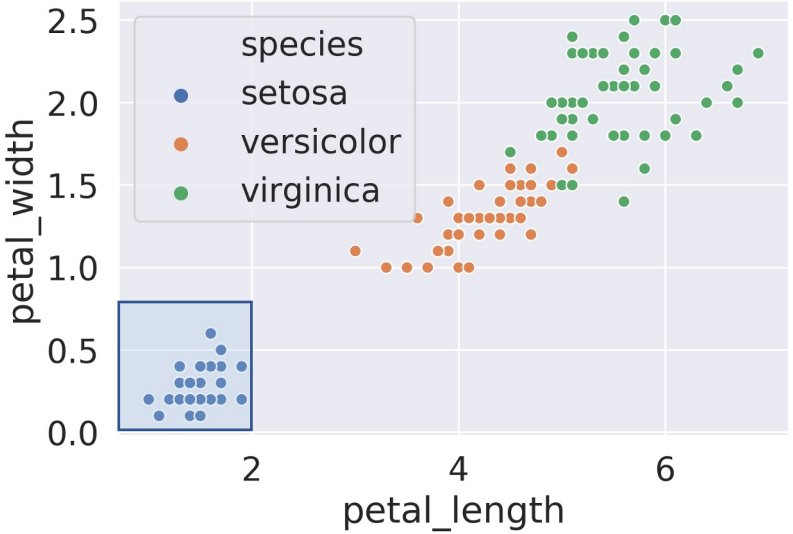
Example: Using Petal Data Only



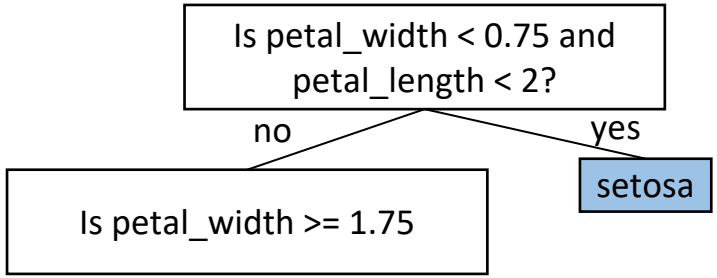


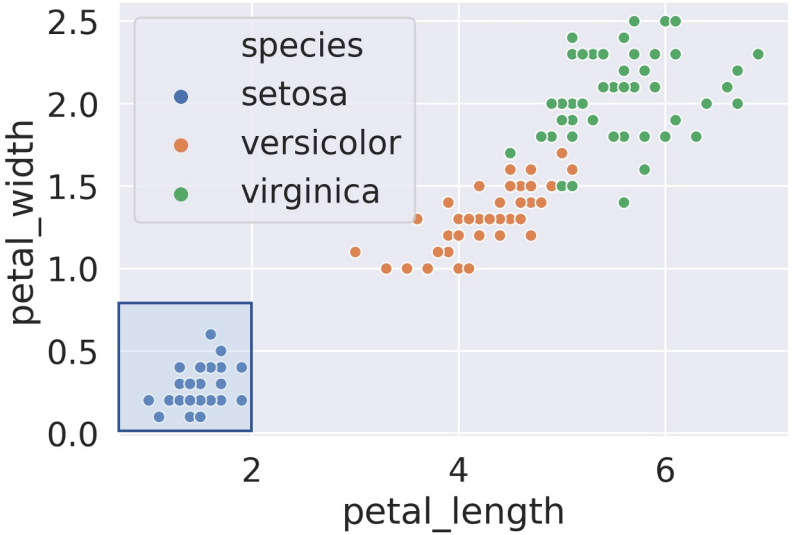
Example: Using Petal Data Only



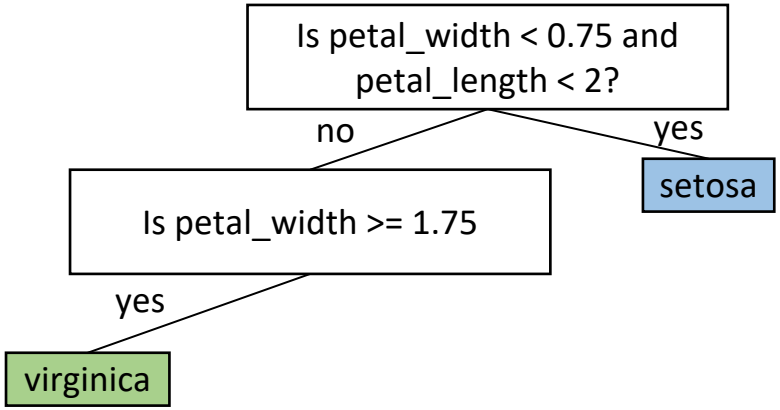


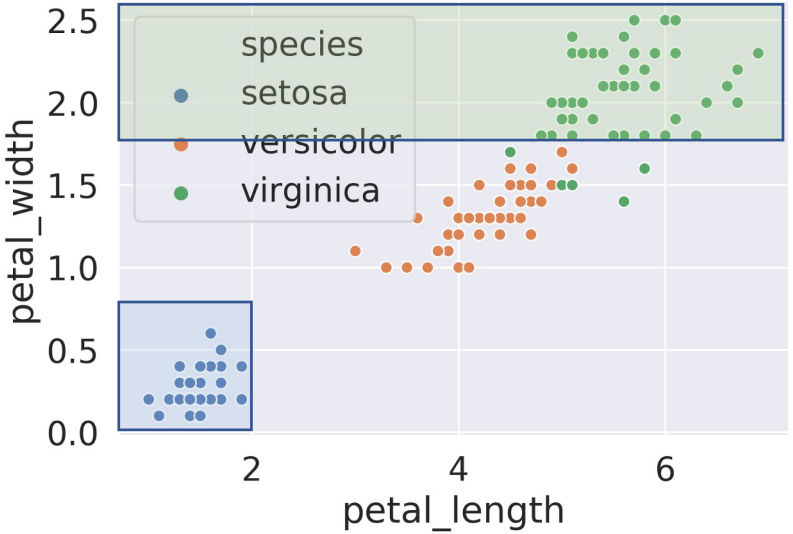
Example: Using Petal Data Only



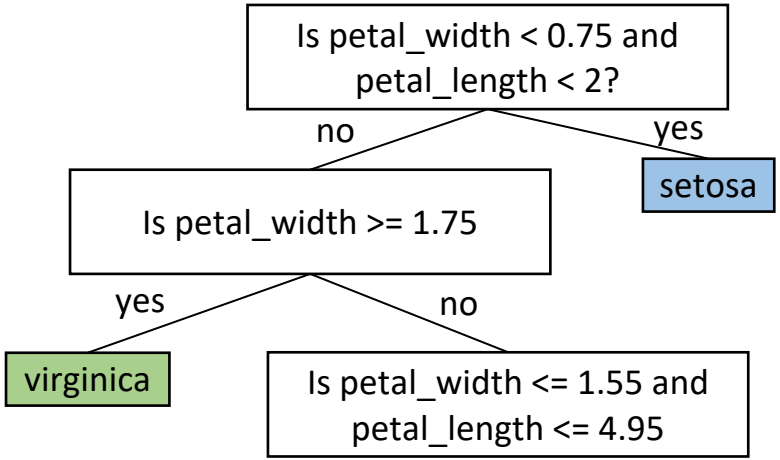


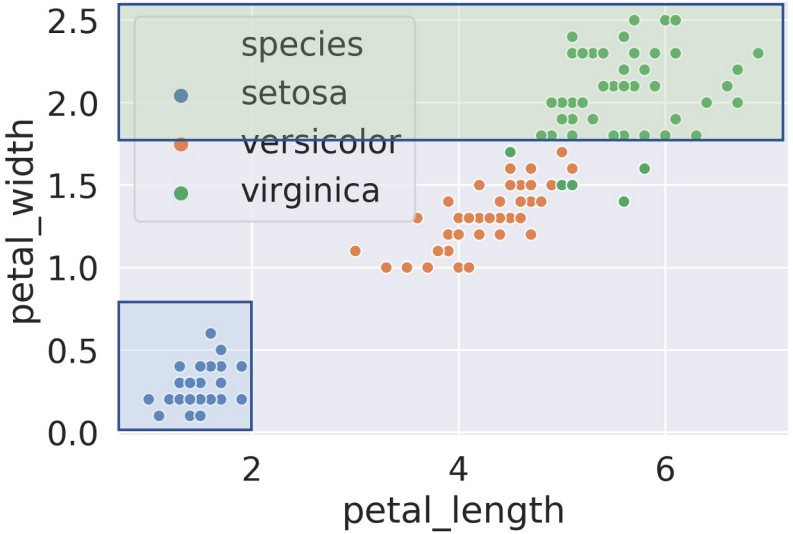
Example: Using Petal Data Only



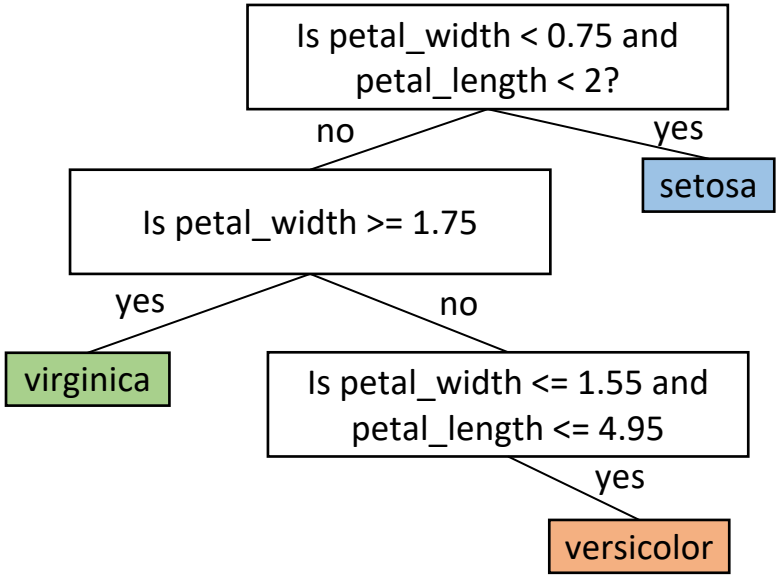


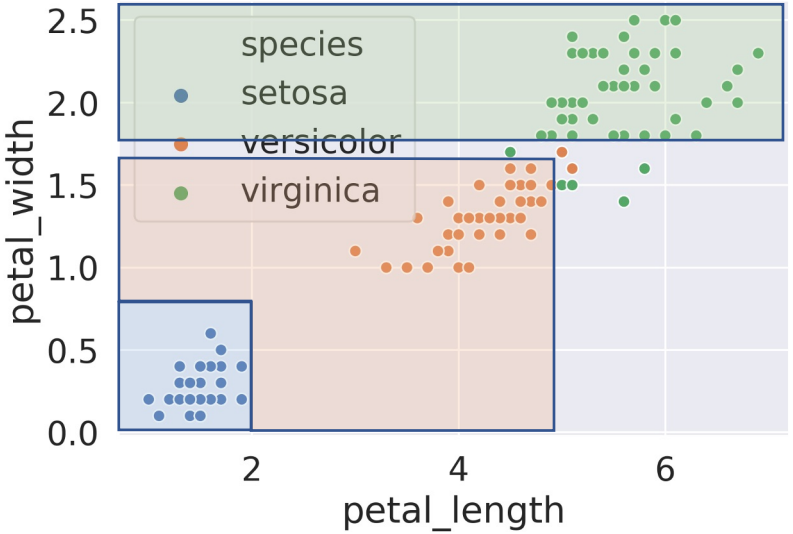
Example: Using Petal Data Only



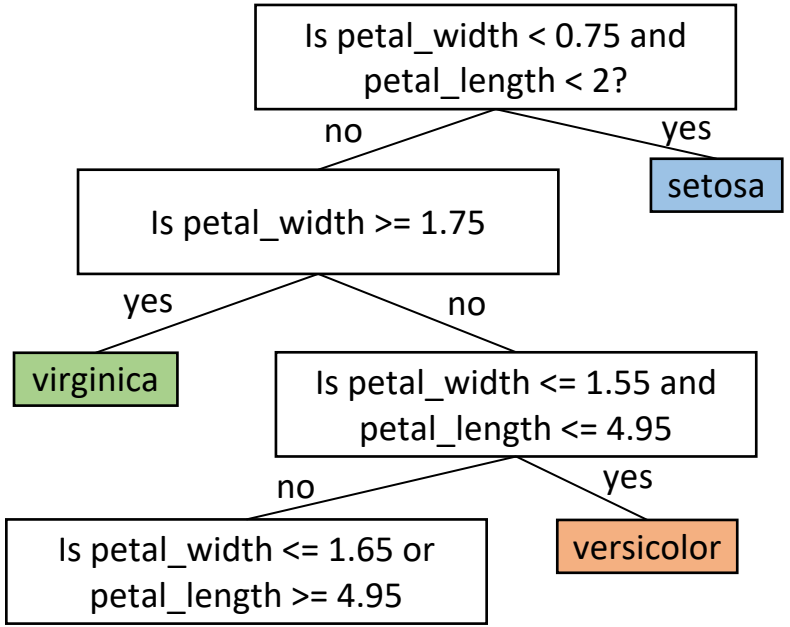


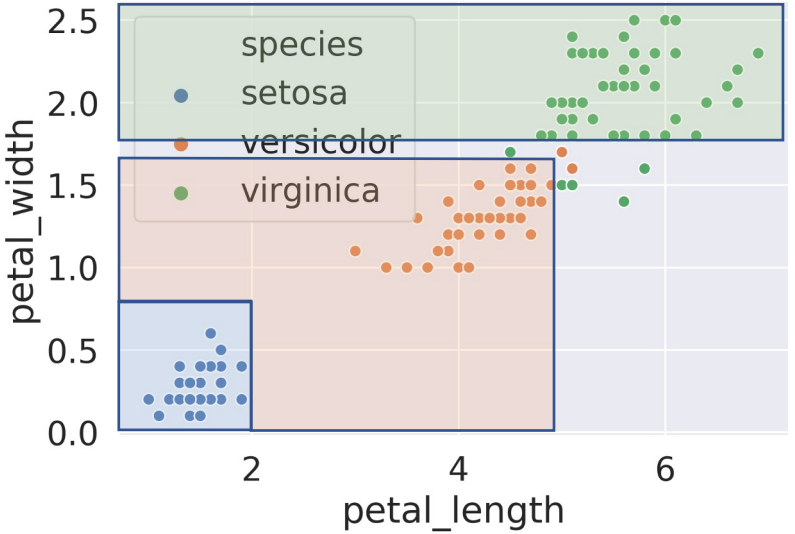
Example: Using Petal Data Only



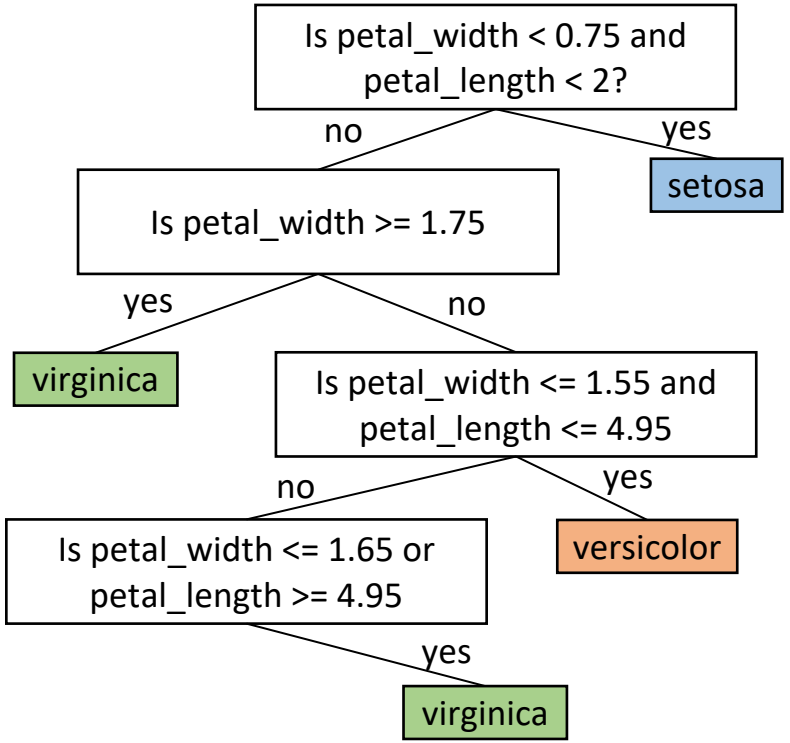


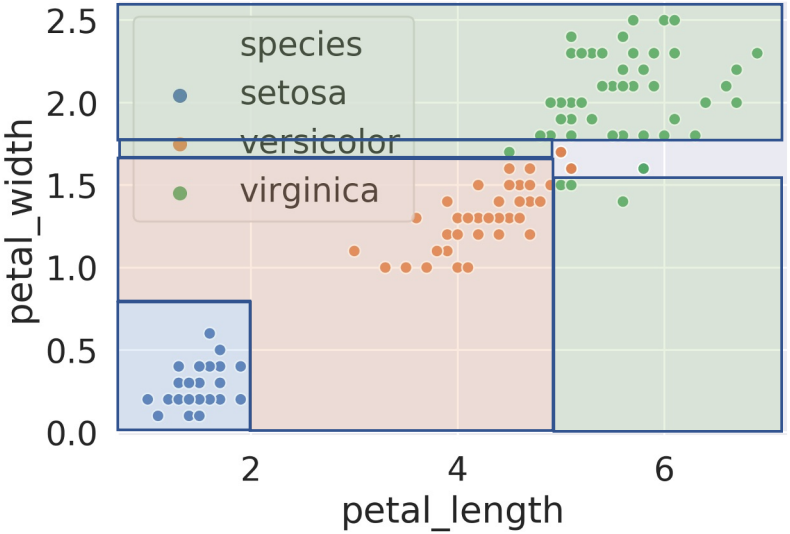
Example: Using Petal Data Only



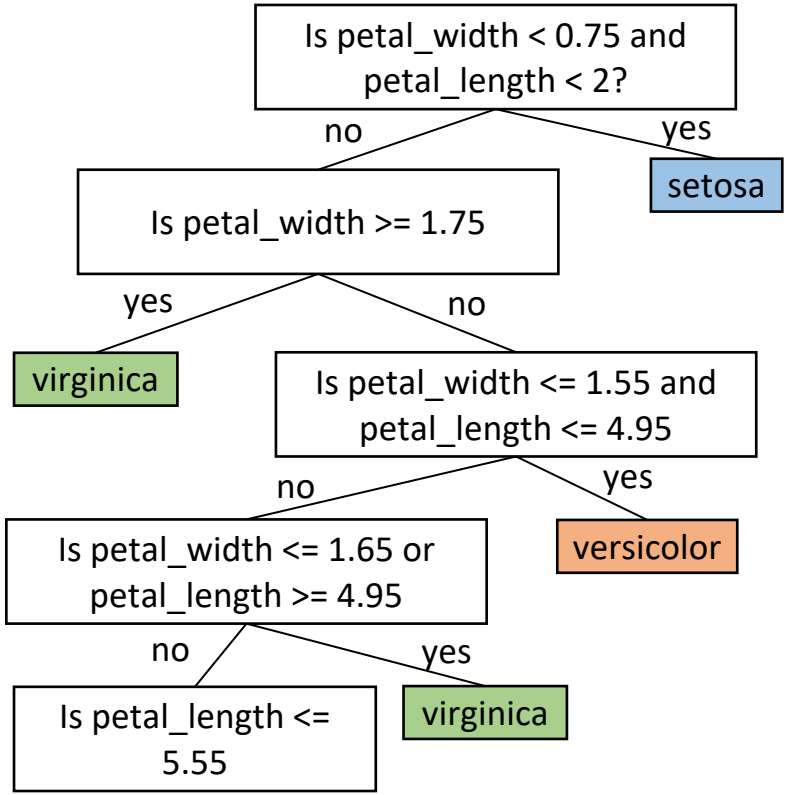


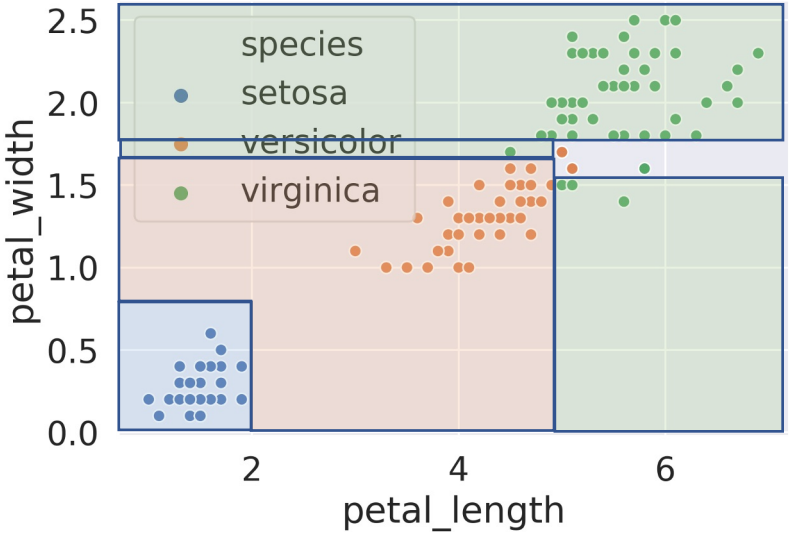
Example: Using Petal Data Only



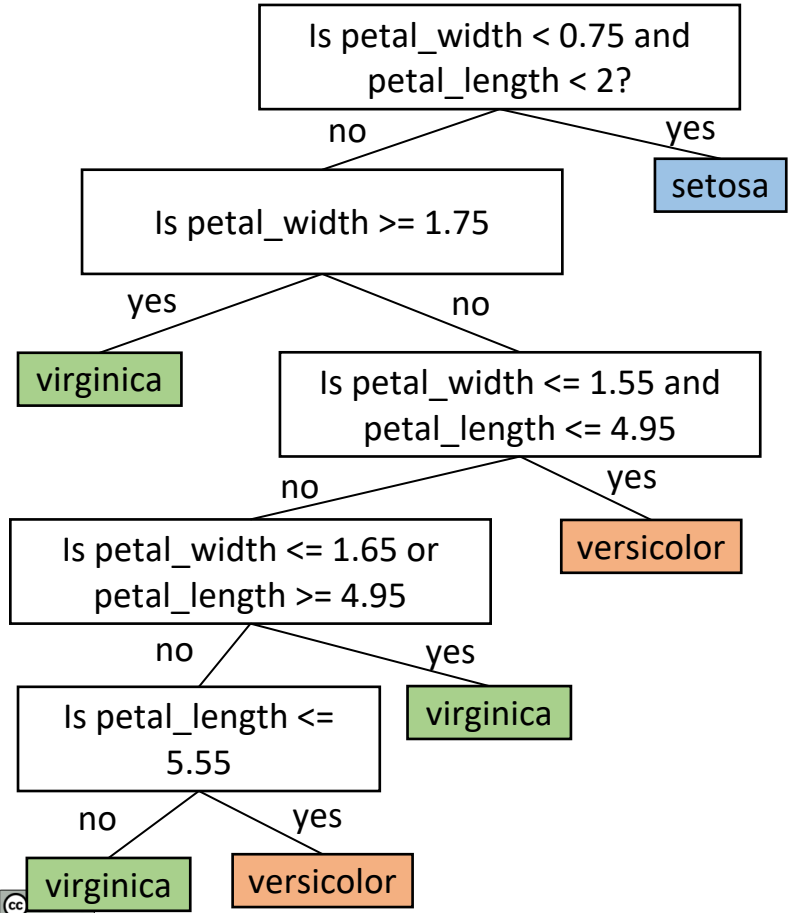


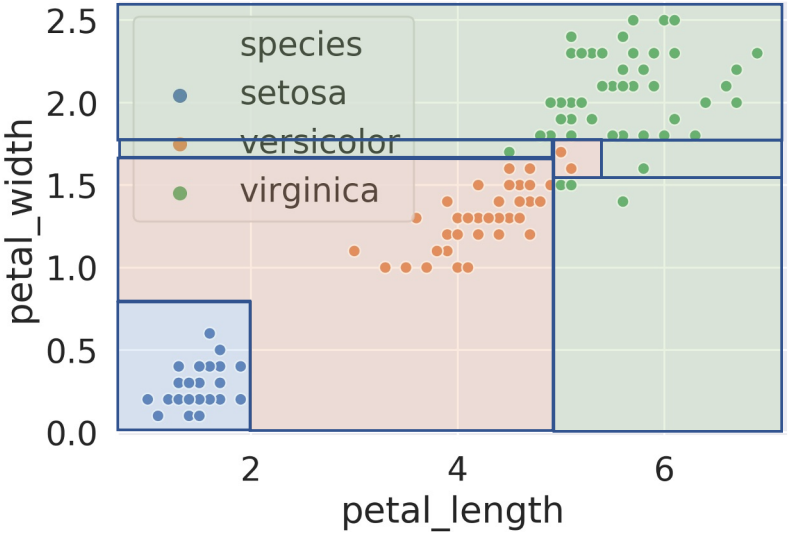
Example: Using Petal Data Only





Example: Using Petal Data Only





Decision Tree in scikit-learn

- Decision Tree Basics
- **Decision Trees in scikit-learn**
- Overfitting
- Basic Decision Tree Generation
- Restricting Decision Tree Complexity
- Random Forest

Decision Tree Models With scikit-learn

The code to build a decision tree model in `scikit-learn` is very similar to what we saw for building linear and logistic regression models:

```
from sklearn import tree
decision_tree_model = tree.DecisionTreeClassifier()
decision_tree_model = decision_tree_model.fit(iris_data[["petal_length", "petal_width"]], iris_data["species"])
```

Decision Tree Models With `scikit-learn`

The code to build a decision tree model in `scikit-learn` is very similar to what we saw for building linear and logistic regression models:

```
from sklearn import tree
decision_tree_model = tree.DecisionTreeClassifier()
decision_tree_model = decision_tree_model.fit(iris_data[["petal_length", "petal_width"]], iris_data["species"])
```

```
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

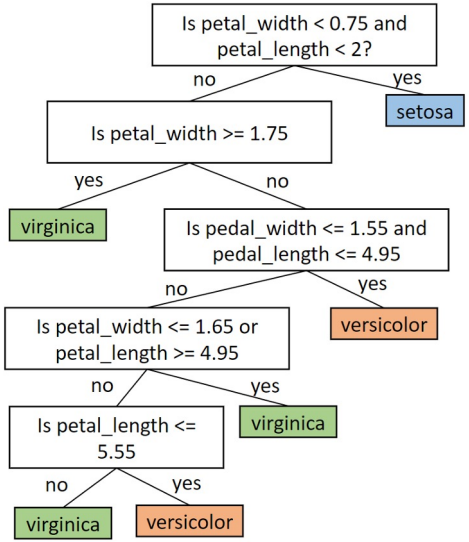
```
decision_tree_model.predict(four_random_rows[["petal_length", "petal_width"]])
```

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

Visualizing Decision Tree Models

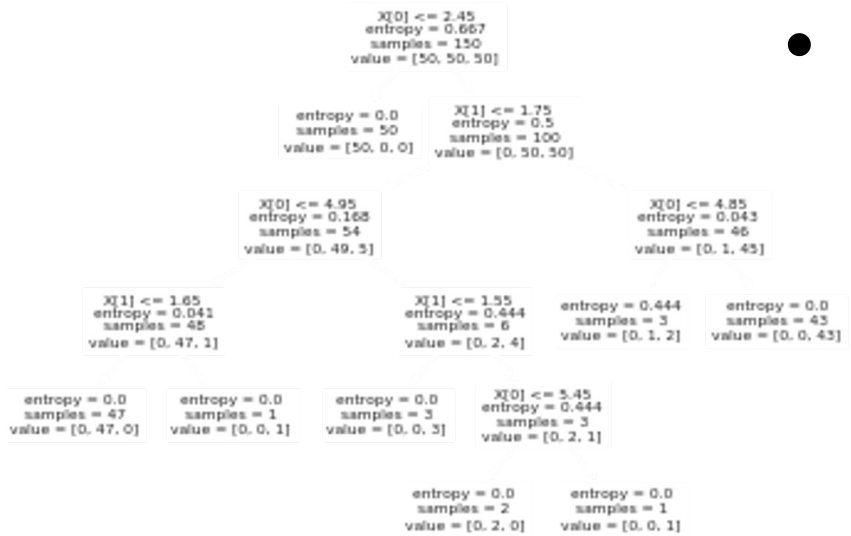
```
from sklearn import tree
decision_tree_model = tree.DecisionTreeClassifier()
decision_tree_model = decision_tree_model.fit(iris_data[["petal_length", "petal_width"]], iris_data["species"])
```

Suppose we want to visualize the decision tree, similar to what we saw earlier:



Visualizing Decision Tree Models

```
from sklearn import tree
decision_tree_model = tree.DecisionTreeClassifier()
decision_tree_model = decision_tree_model.fit(iris_data[["petal_length", "petal_width"]], iris_data["species"])
```



There is a built in `DecisionTree` visualizer

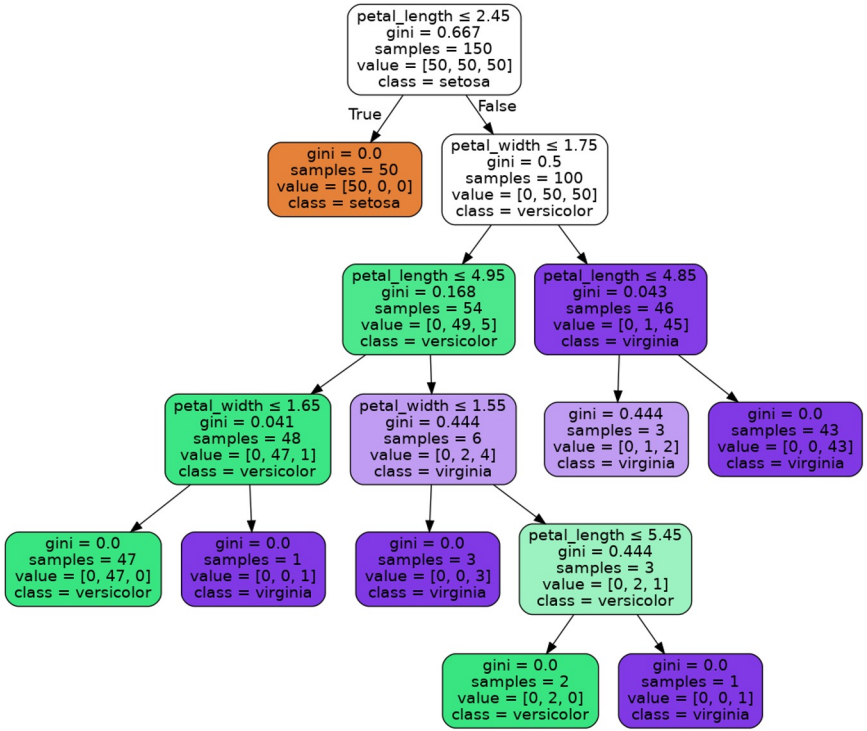
- Unfortunately, it isn't very good

```
tree.plot_tree(decision_tree_model)
```

Visualizing Decision Tree Models

Can use GraphViz to get a much nicer picture.

```
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,
                                special_characters=True)
graph = graphviz.Source(dot_data)
graph
```

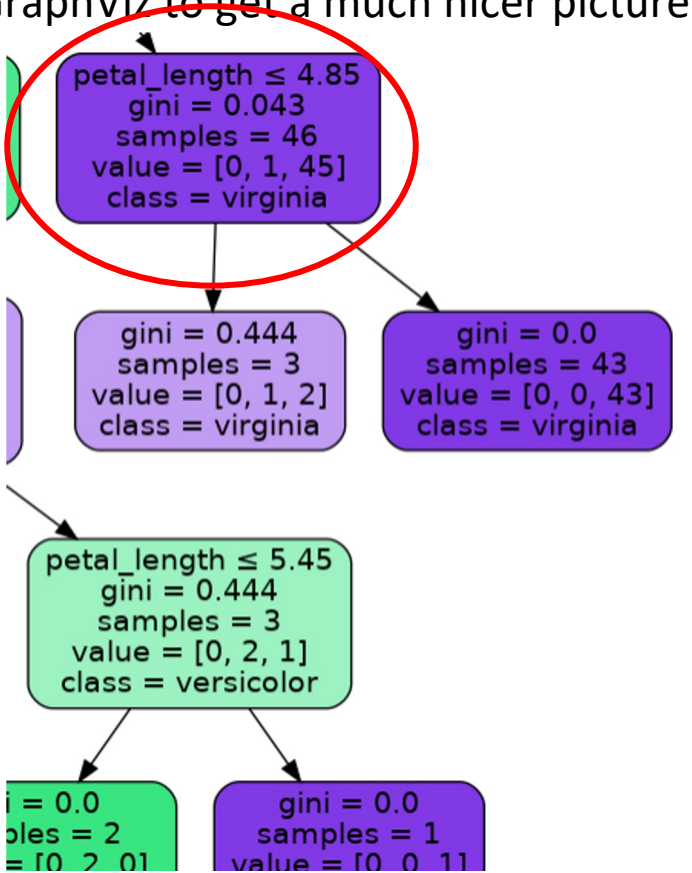


In each box, we see:

- The rule
- The gini impurity (chance that a sample would be misclassified if randomly assigned at this point)
- The number of samples still unclassified
- The number of samples in each class still unclassified
- The most likely class

Visualizing Decision Tree Models

Can use GraphViz to get a much nicer picture.




```
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,
                                special_characters=True)
graph = graphviz.Source(dot_data)
graph
```

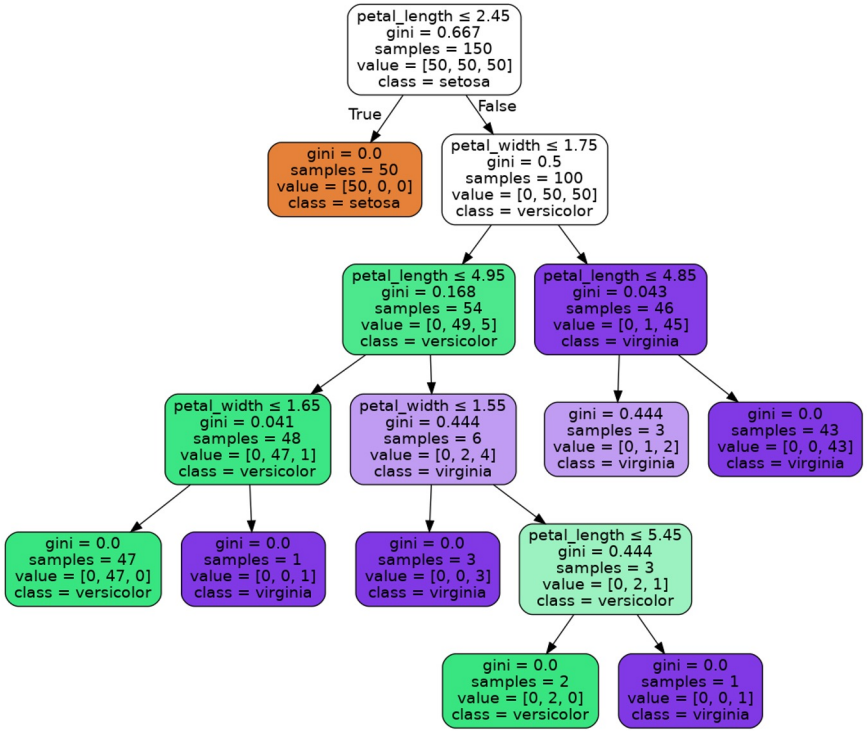
In each box, we see:

- The rule
- The gini impurity (chance that a sample would be misclassified if randomly assigned at this point)
- The number of samples still unclassified
- The number of samples in each class still unclassified
- The most likely class

Visualizing Decision Tree Models

Can use GraphViz to get a much nicer picture.

```
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,
                                special_characters=True)
graph = graphviz.Source(dot_data)
graph
```



In each box, we see:

- The rule
- The gini impurity (chance that a sample would be misclassified if randomly assigned at this point)
- The number of samples still unclassified
- The number of samples in each class still unclassified
- The most likely class

Basic Decision Tree Generation

- Decision Tree Basics
- Decision Trees in scikit-learn
- **Basic Decision Tree Generation**
- Overfitting
- Restricting Decision Tree Complexity
- Random Forest

Decision Tree Generation

In order to understand how to avoid overfitting, let's first discuss how decision trees are created from data

Traditional decision tree generation algorithm:

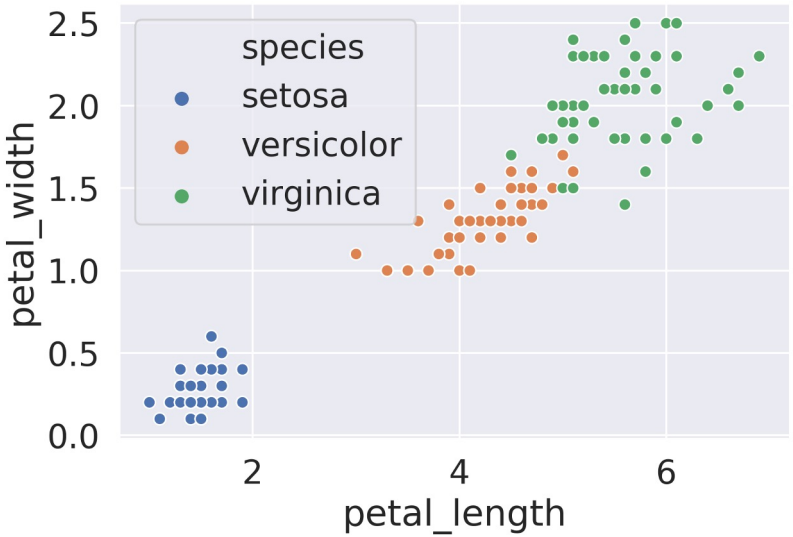
- All of the data starts in the root node
- **Repeat** until every node is either **pure** or **unsplittable**:
  - **Pick the best feature** x and **best split value** β, e.g. x = petal_length, β = 2
  - **Split data into two nodes**, one where $x < \beta$, and one where $x \geq \beta$

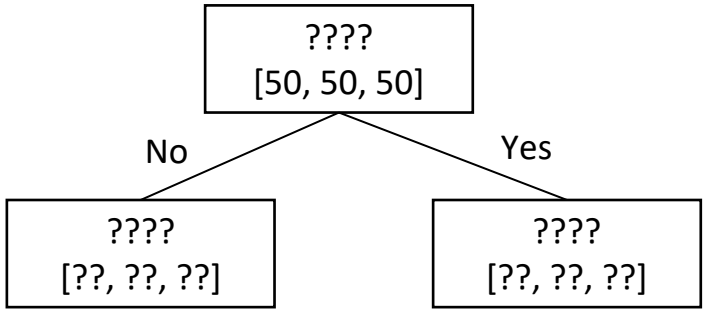
Notes: A node that has only one type is called a "**pure**" node. A node that has duplicate data that cannot be split is called "**unsplittable**".

Defining a Best Feature

Question: Which feature and split value is best?

- Equivalently: Which horizontal or vertical line do we want to draw?

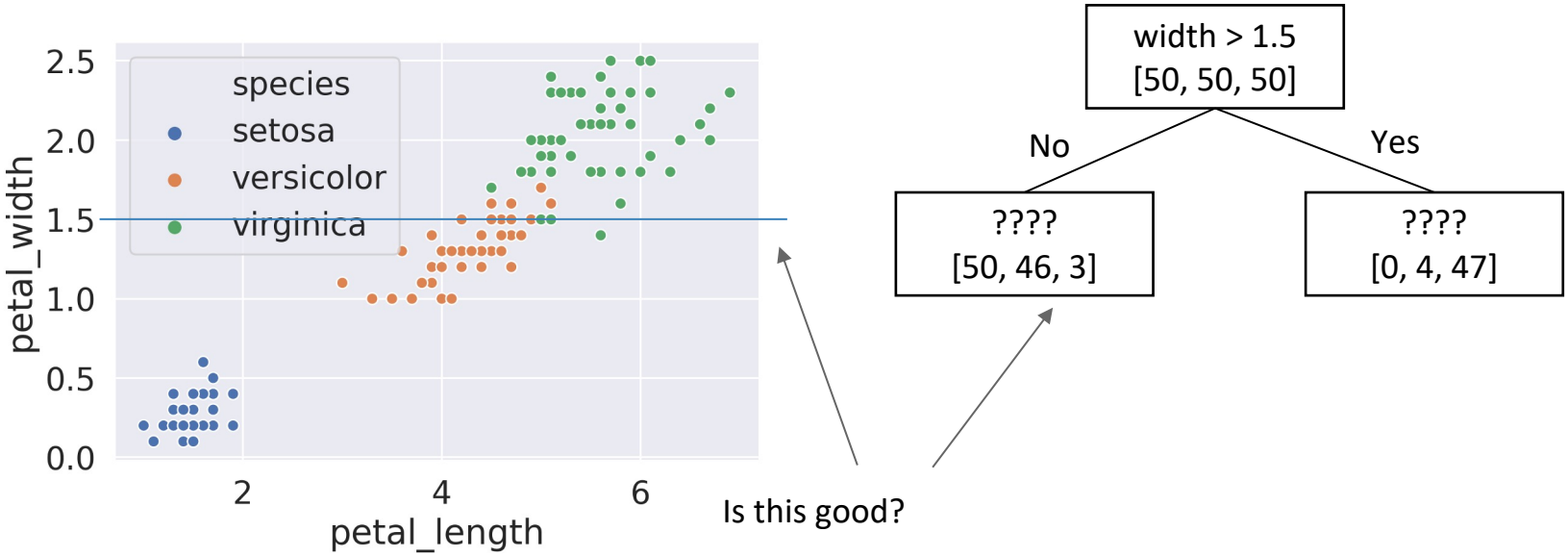




Defining a Best Feature

Question: Which feature and split value is best?

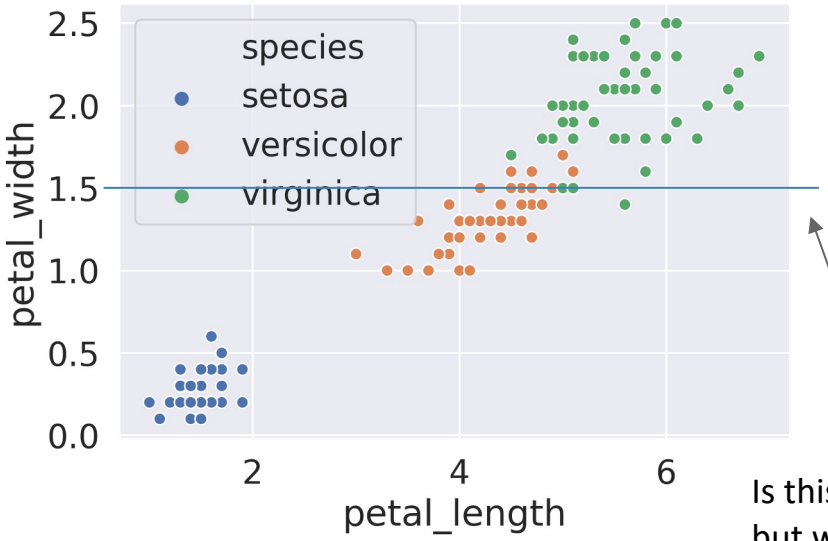
- Equivalently: Which horizontal or vertical line do we want to draw?

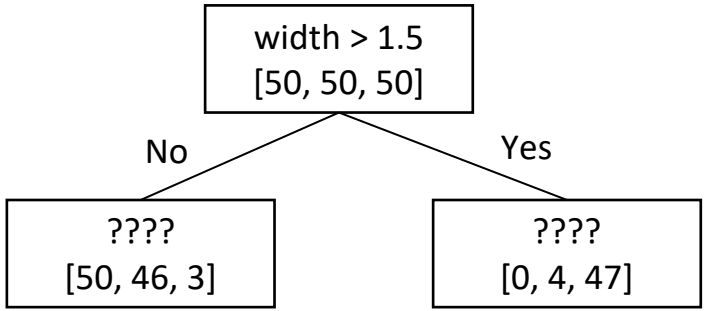


Defining a Best Feature

Question: Which feature and split value is best?

- Equivalently: Which horizontal or vertical line do we want to draw?



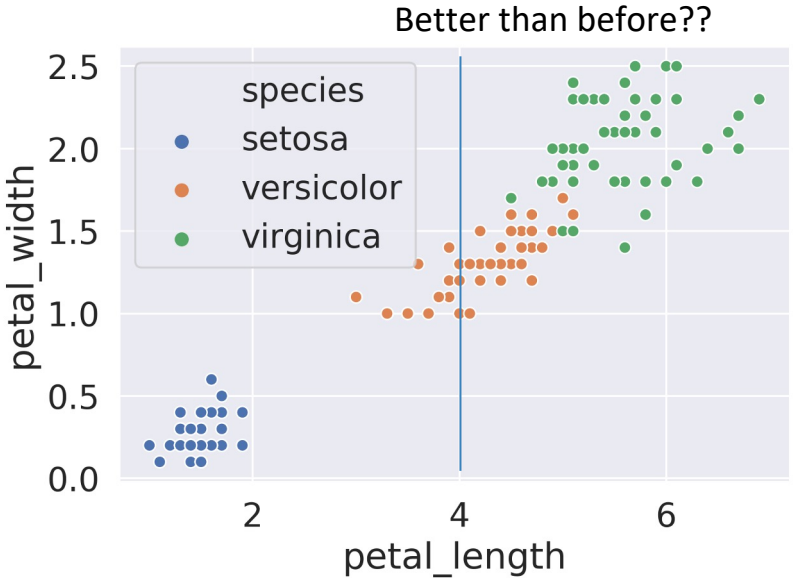


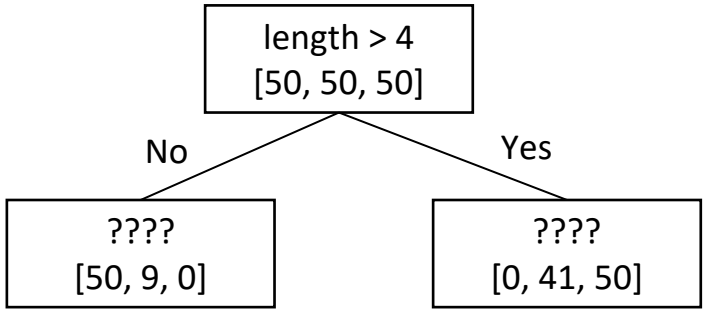
Is this good? It does help, but we could do better!

Defining a Best Feature

Question: Which feature and split value is best?

- Equivalently: Which horizontal or vertical line do we want to draw?

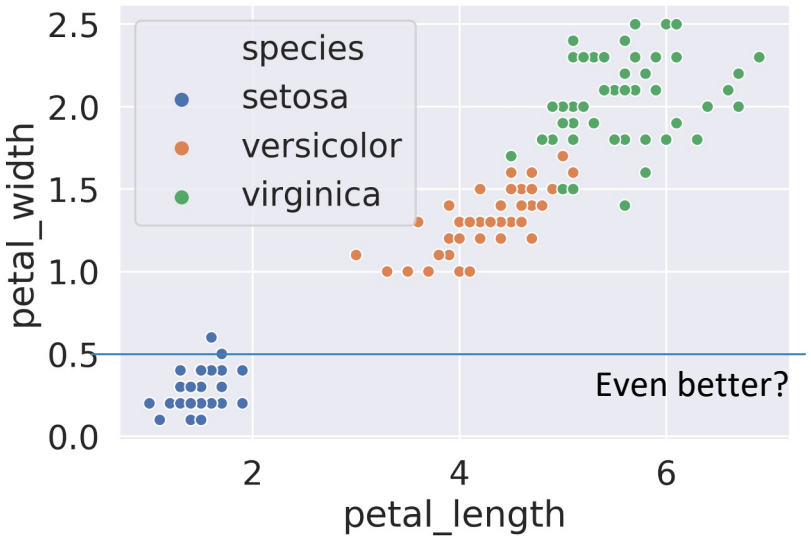


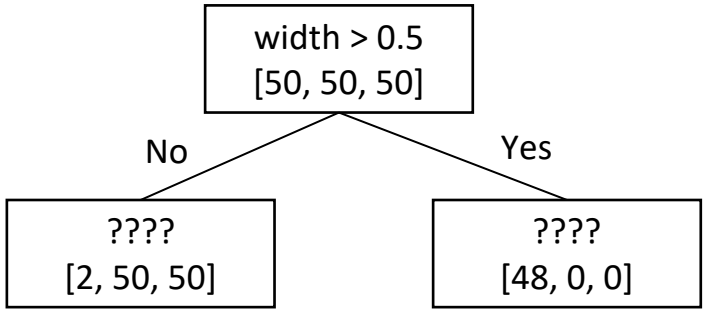


Defining a Best Feature

Question: Which feature and split value is best?

- Equivalently: Which horizontal or vertical line do we want to draw?



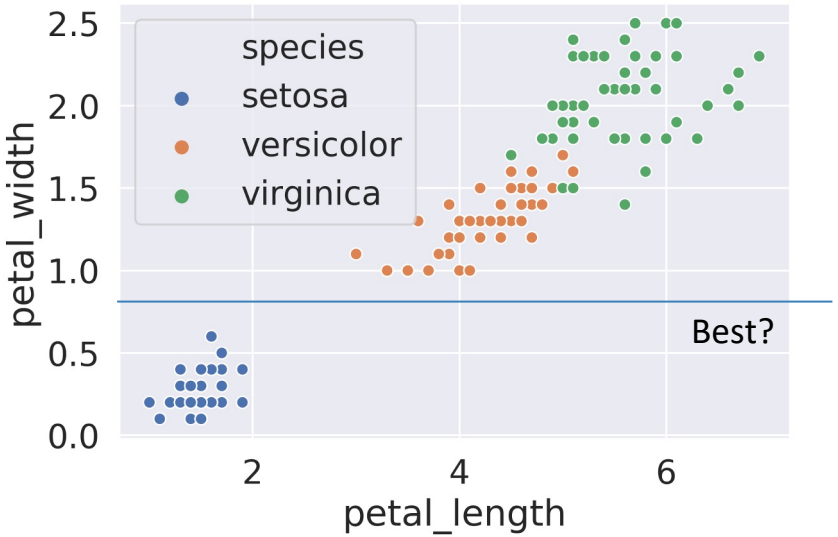


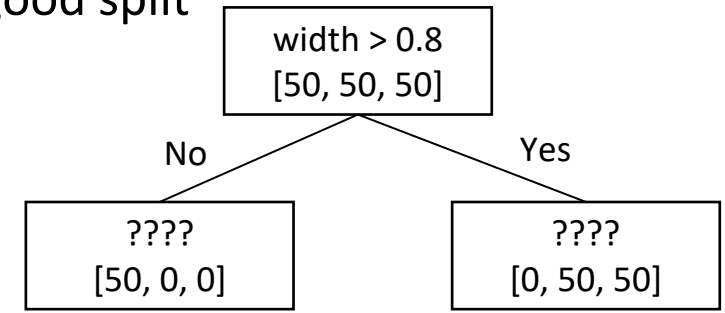
Defining a Best Feature

Question: Which feature and split value is best?

- Equivalently: Which horizontal or vertical line do we want to draw?

We need some sort of rigorous definition for a good split

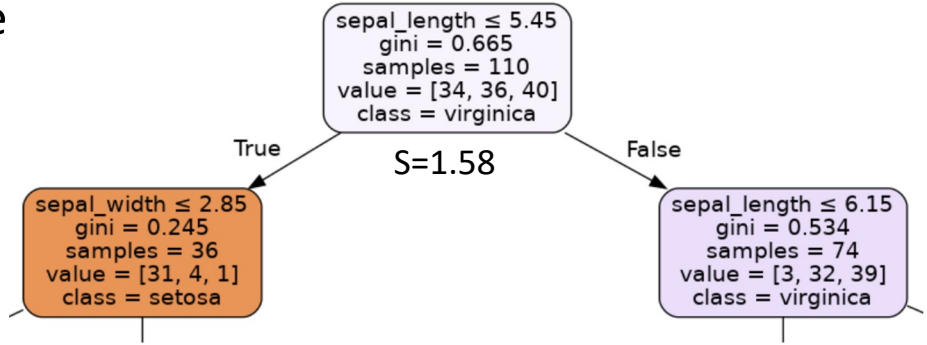




Node Entropy

Let p_C be the proportion of data points in a node with label C.

For example, for the node at the top of the decision tree, p_0 = 34/110 = 0.31, p_1 = 36/110 = 0.33, and p_2 = 40/110 = 0.36



Define the entropy S of a node as:

$$S = -\sum_C p_C \log_2 p_C$$

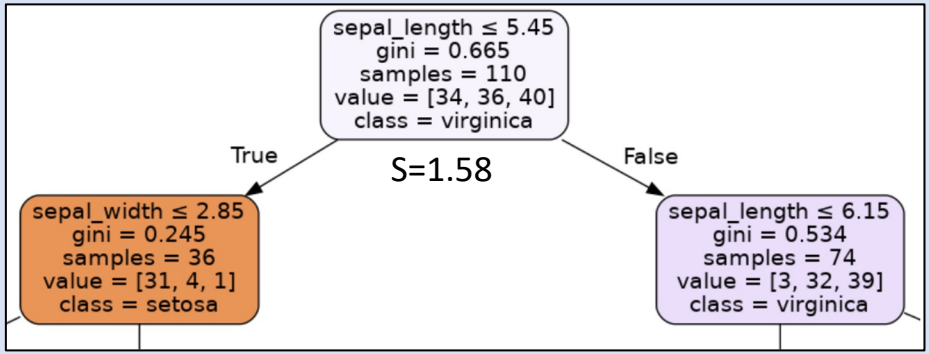
For example, S for the top node is:

$-0.31 \log_2 0.31 - 0.33 \log_2 0.33 - 0.36 \log_2 0.36$

$= 0.52 + 0.53 + 0.53 = 1.58$

Test Your Understanding

What is p_0?



What is the entropy of the node on the left with [31, 4, 1] in each class?

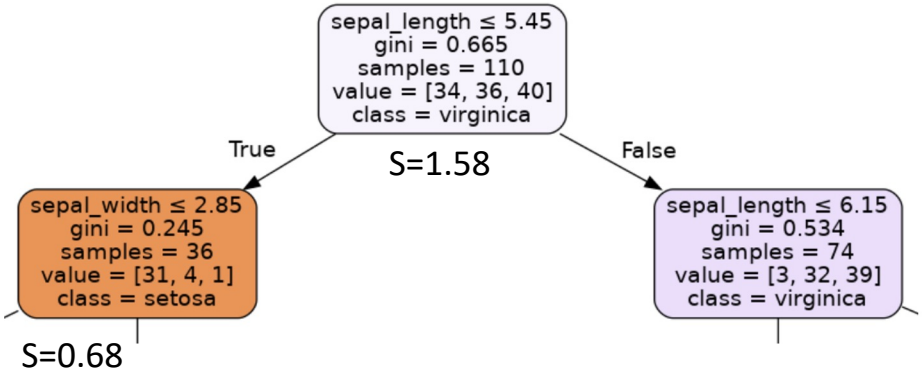
- Try writing out an expression, or try to compute an exact value

$$S = -\sum_{C} p_C \log_2 p_C$$

Test Your Understanding

What is the entropy of the node on the left with [31, 4, 1] in each class?

- p_0 = 31/36 = 0.86, p_1 = 4/36 = 0.11, and p_2 = 1/36 = 0.028
- S = $-0.86 \log_2 0.86 - 0.11 \log_2 0.11 - 0.028 \log_2 0.028 = 0.68$



Define the entropy S of a node as:

$$S = -\sum_C p_C \log_2 p_C$$

Can think of entropy as how unpredictable a node is:

- Low entropy means more predictable
- High entropy means more unpredictable

Exploring Entropy

Observations about entropy:

- A node where all data are part of the same class has zero entropy

  $-1 \log_2 \quad 1 = 0$

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

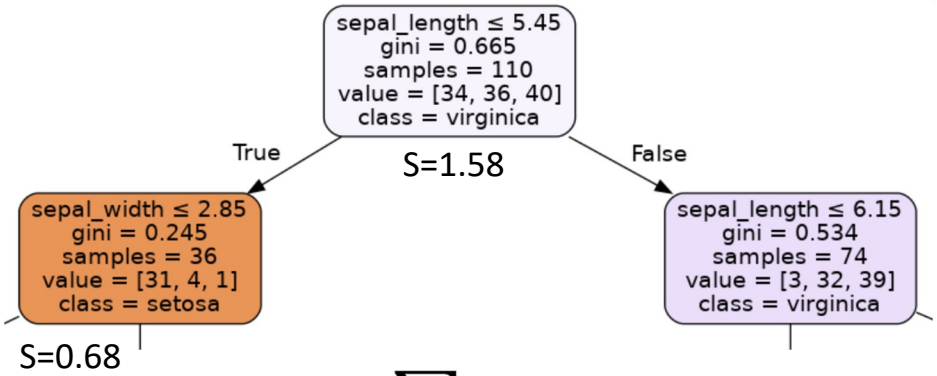
  $-0.5 \log_2 \quad 0.5 - 0.5 \log_2 \quad 0.5 = 1$

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

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

- A node where data are evenly split into C classes has entropy $\log_2 C$

  $C \times (-1/C \log_2 \quad 1/C) = -\log_2 \quad 1/C = \log_2 C$



$$S = -\sum_C p_C \log_2 p_C$$

Weighted Entropy as a Loss Function

We can use Weighted Entropy as a loss function in helping us decide which split to take

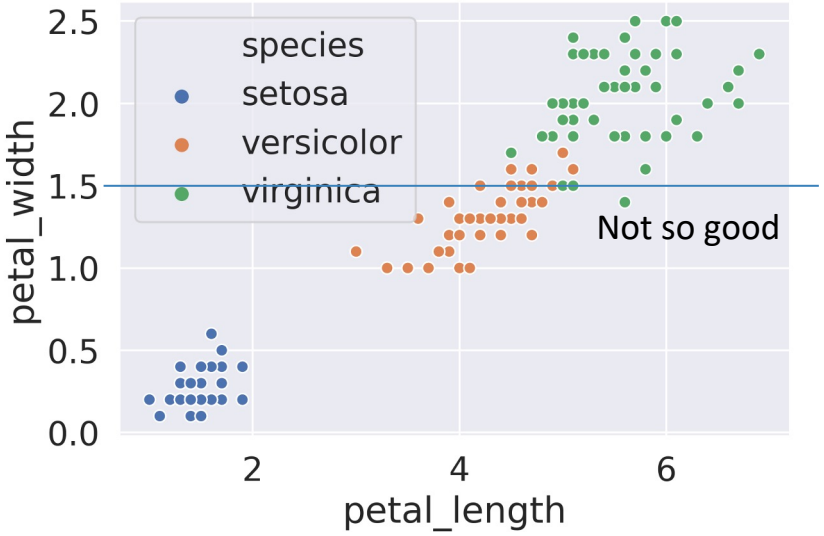
Suppose a given split results in two nodes X and Y with N_1 and N_2 total samples each. The loss of that split is given by:

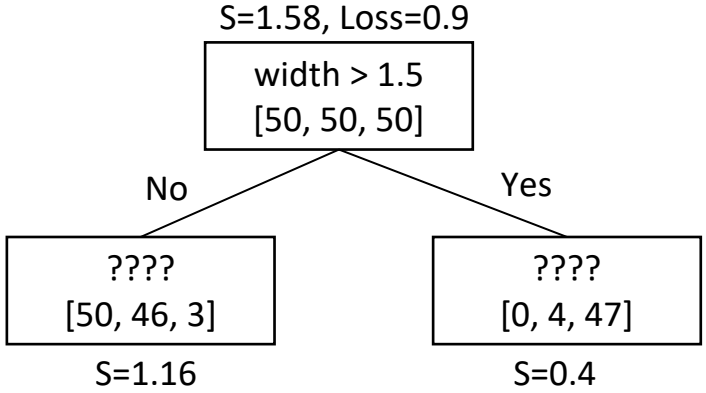
$$L = \frac{N_1 S(X) + N_2 S(Y)}{N_1 + N_2}$$

Defining a Best Feature

Split choice #1: width > 1.5. Compute entropy of child nodes:

- entropy([50, 46, 3]) = 1.16
- entropy([4, 47]) = 0.4
- Weighted average: 99/150 × 1.16 + 51/150 × 0.4 = **0.9**

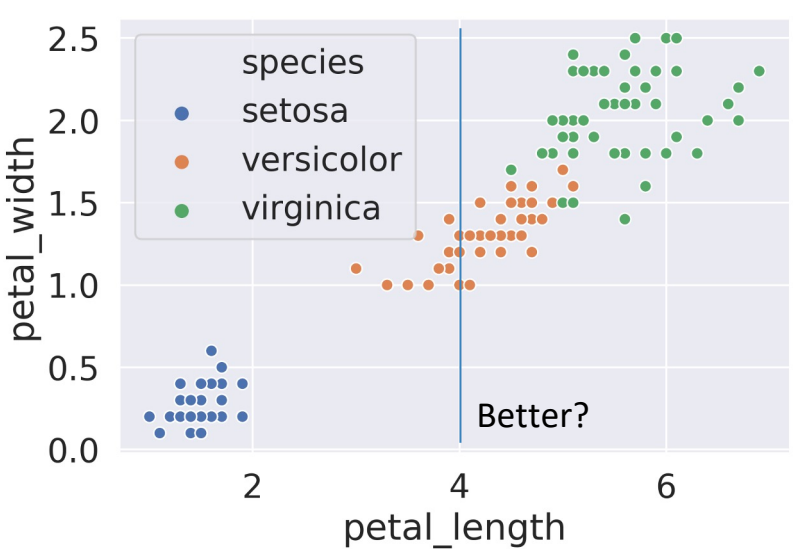


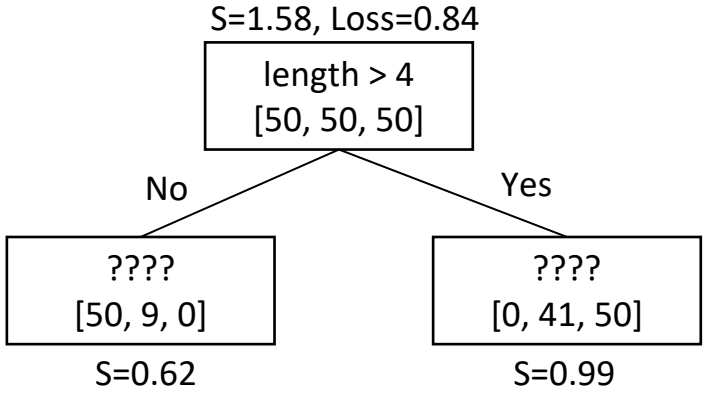


Defining a Best Feature

Split choice #2: length > 4. Compute entropy of child nodes:

- entropy([50, 9]) = 0.62
- entropy([41, 50]) = 0.99
- Weighted Average: **0.84**: Better than split choice #1!

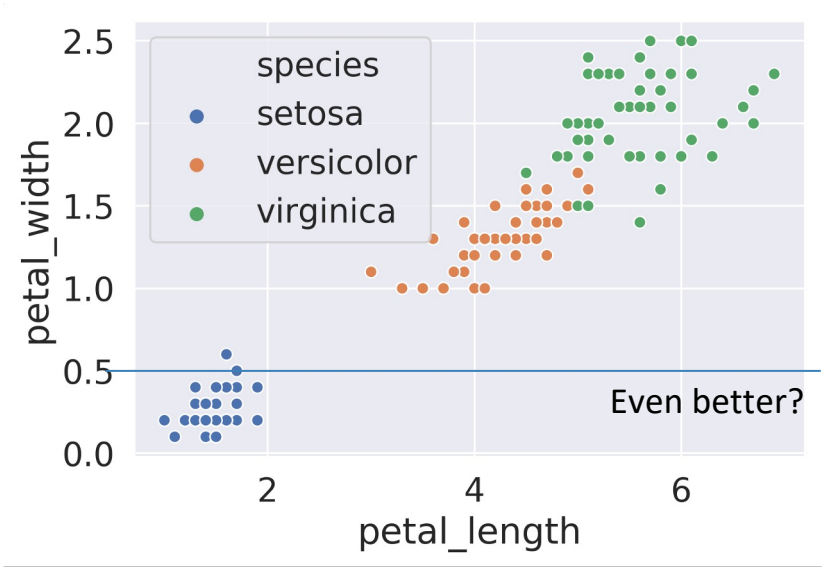


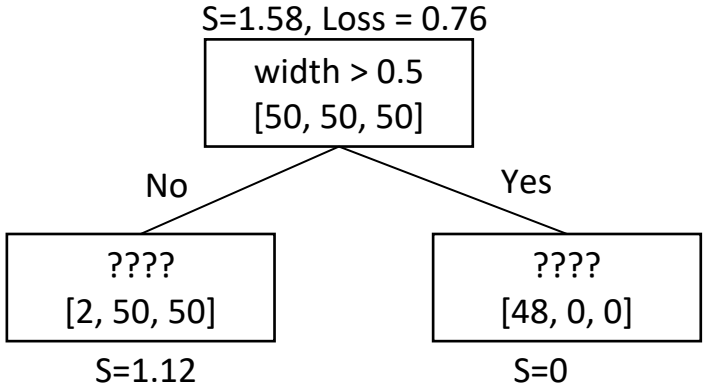


Defining a Best Feature

Split choice #3: width > 0.5. Compute entropy of child nodes:

- entropy([2, 50, 50]) = 1.12
- entropy([48]) = 0
- Weighted average: **0.76**: Lower than split choice #2!

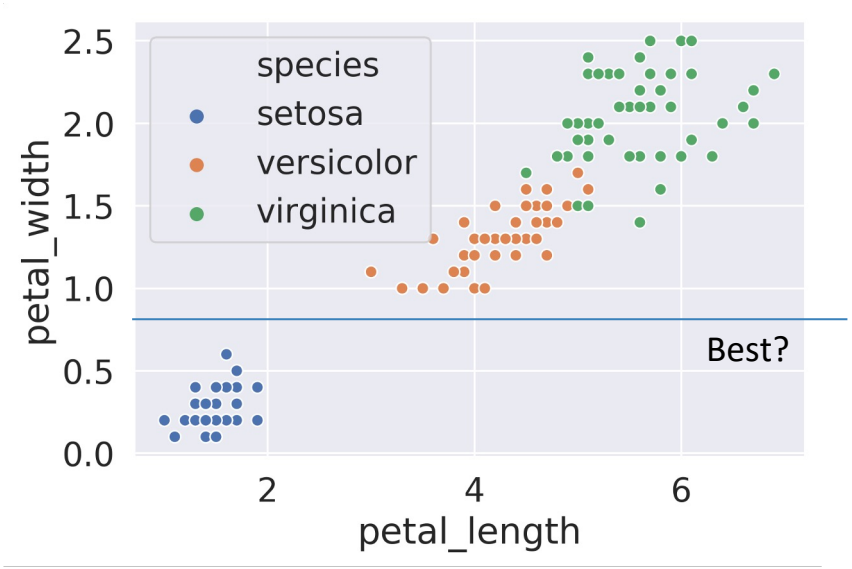


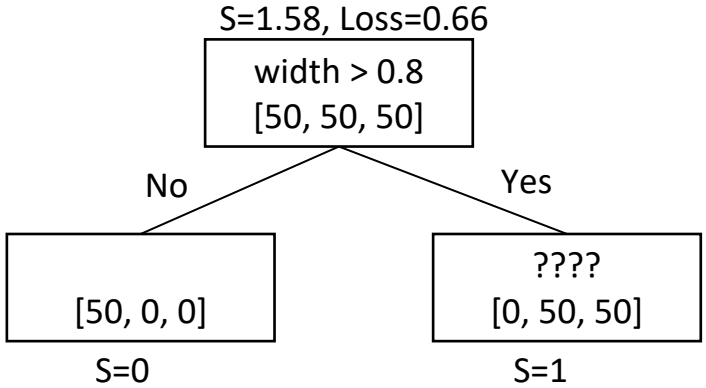


Defining a Best Feature

Split choice #4: width > 0.9. Compute entropy of child nodes:

- entropy([50, 50]) = 1
- entropy([50]) = 0
- Weighted average: **0.66**: Lower than split choice #3!





Decision Tree Generation

Traditional decision tree generation algorithm:

- All of the data starts in the root node
- Repeat until every node is either **pure** or **unsplittable**:
  - **Pick the best feature** x and **split value** β such that the **loss of the resulting split** is **minimized**, e.g. x = petal_width, β = 0.8 has loss 0.66
  - **Split data into two nodes**, one where $x < \beta$, and one where $x \geq \beta$

Notes: A node that has only one type is called a "**pure**" node. A node that has duplicate data that cannot be split is called "**unsplittable**"

Let's now turn our attention to avoiding overfitting

Overfitting

- Decision Tree Basics
- Decision Trees in scikit-learn
- Basic Decision Tree Generation
- **Overfitting**
- Restricting Decision Tree Complexity
- Random Forest

Overfitting and Decision Trees

With the exception of samples that have the exact same data, we can always get 100% accuracy. This should give us concern about possible overfitting

Let's see how bad things are by separating our data into train and test sets

- We'll use 110 training points and 40 testing points
- Performance on test set gives us an estimate of how well we generalize

```
train_iris_data, test_iris_data = np.split(iris_data.sample(frac=1), [110])
```

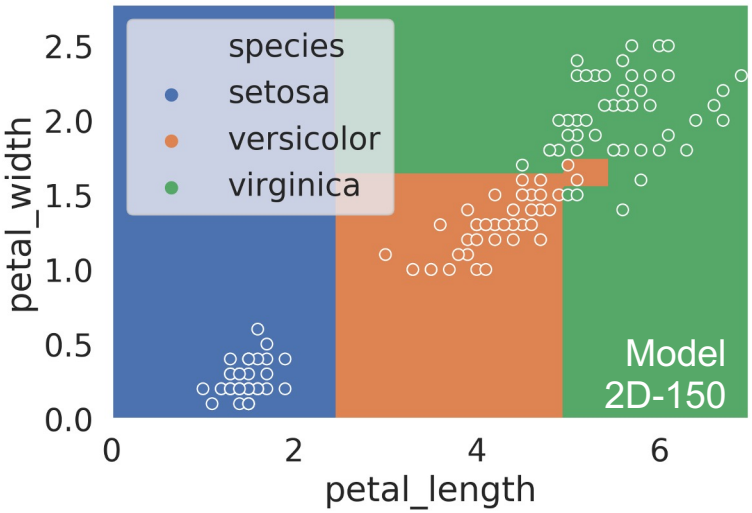
```
from sklearn import tree
decision_tree_model = tree.DecisionTreeClassifier()
decision_tree_model = decision_tree_model.fit(train_iris_data[["petal_length", "petal_width"]], train_iris_data["species"])
```

Visualizing Our New Model

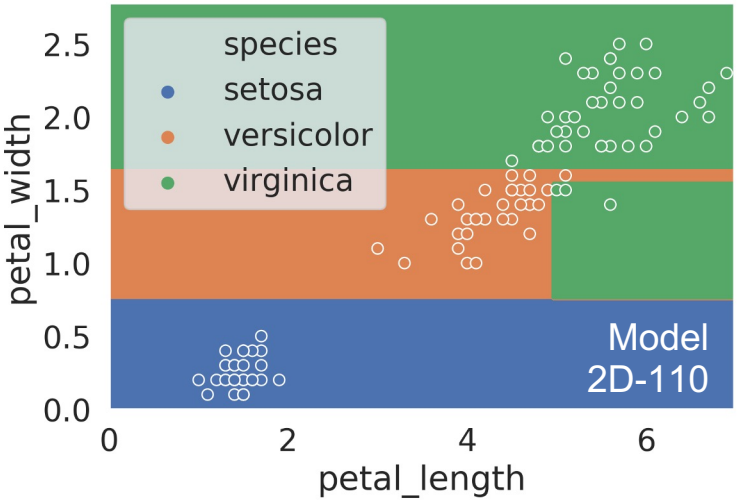
Comparing our model trained on 150 vs. 110 data points, we see slight differences in the generated model

- Naturally, both models get very high accuracy on the data that they use for training (100% for non-overlapping points)

Trained on all 150 points.



Trained on 110 points.

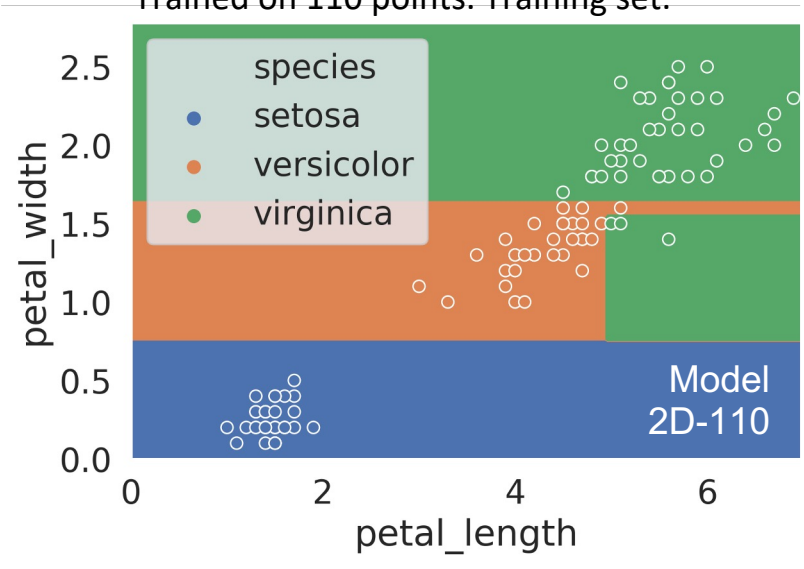


Performance of Our New Model on the Test Set

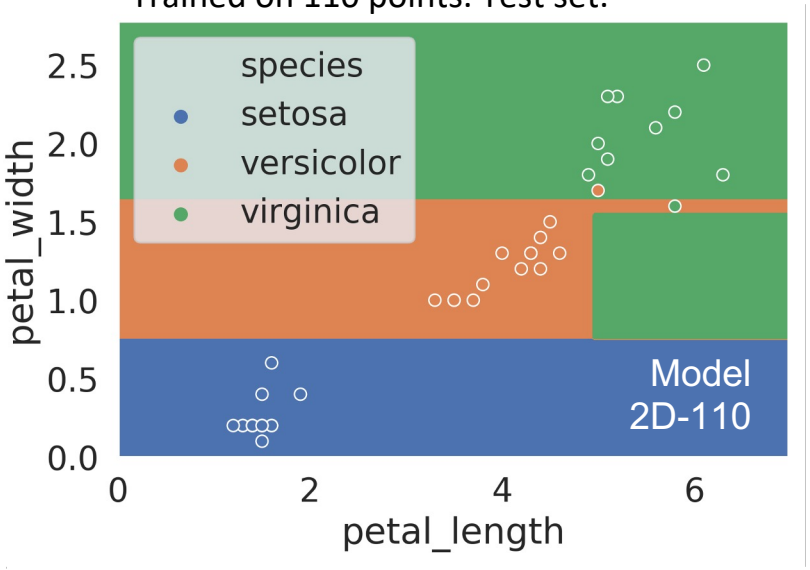
When we look at the performance of Model 2D-110 on the test set, we see that we make some errors that aren't just from overlapping data

- 99% accuracy on training set (2 overlapping points). 95% accuracy on test set

Trained on 110 points. Training set.



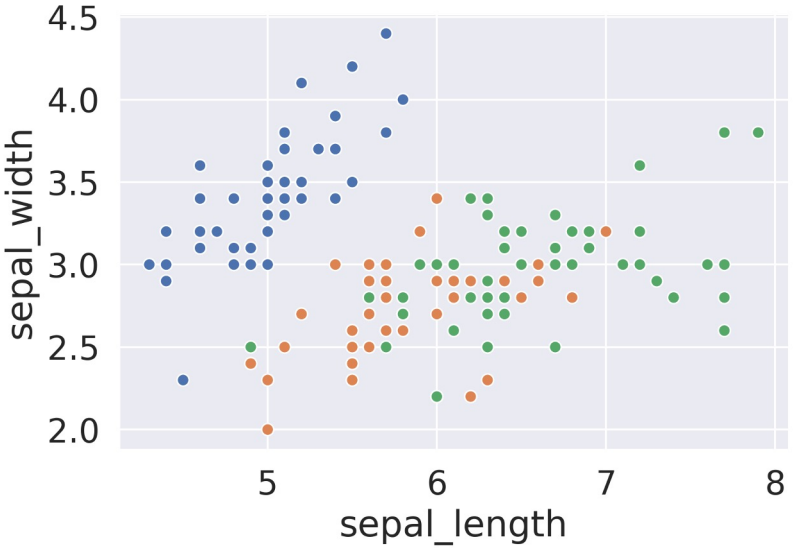
Trained on 110 points. Test set.



Overfitting and Sepal Data

If we use the sepal data only (no petal data), we will run into serious overfitting issues

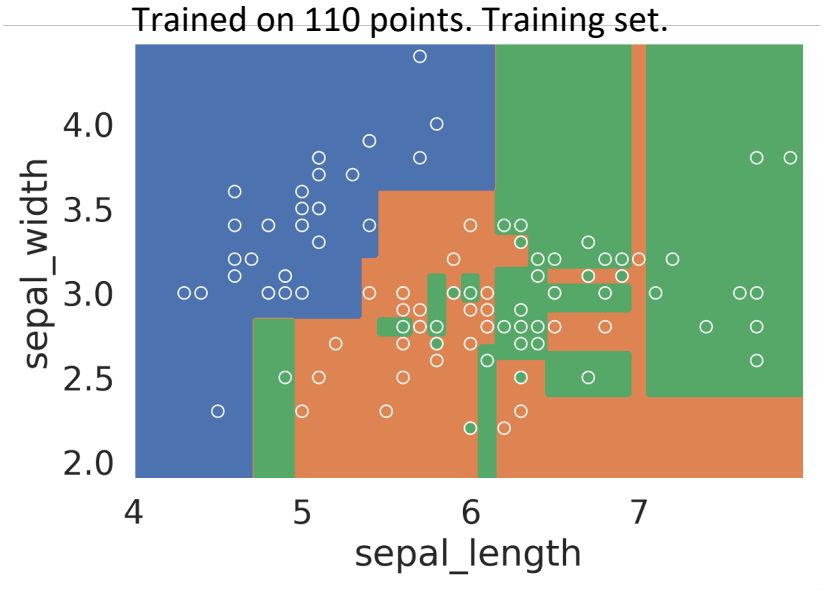
Below, we see our three flowers plotted in the sepal space



Sepal Decision Boundaries

After training on 110/150 points, we get the decision boundaries below

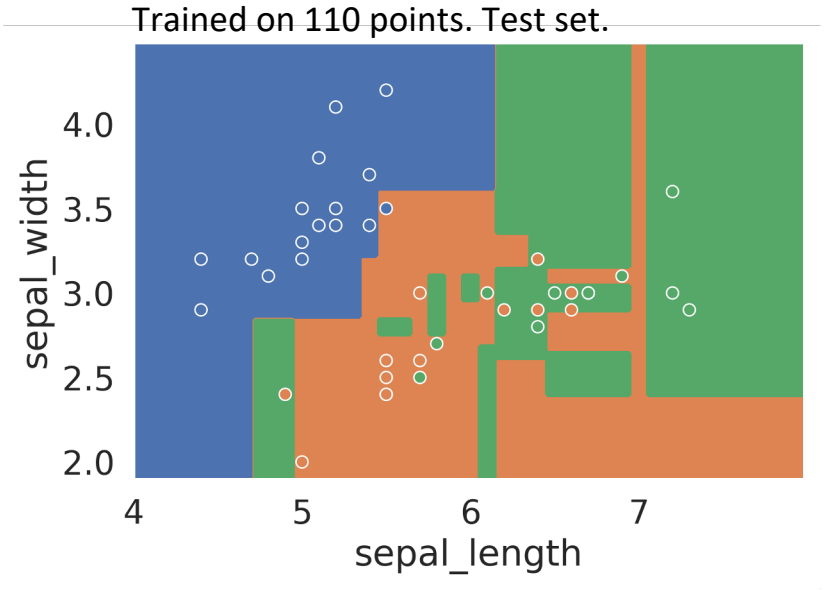
- Decision boundaries are erratic!
- Many overlapping points leads to only 93% accuracy on training set



Sepal Decision Boundaries and Test Set

Performance on test set is quite poor

- Only 70% accuracy: 28/40 predictions are correct

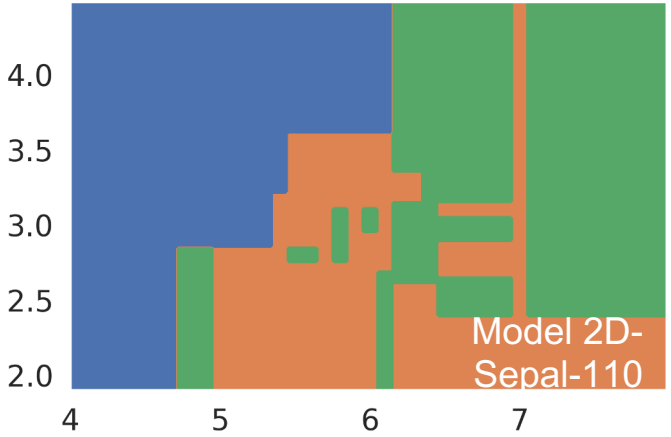


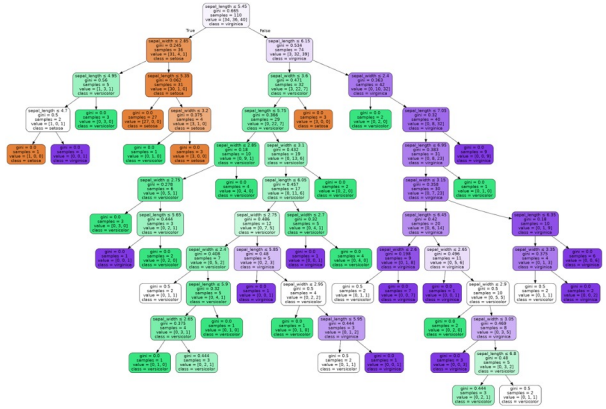
Overfitting Visualized

The decision boundaries for our sepal model were quite complex

- Or drawn out as a tree, we also see a highly complex structure

Next, we'll discuss strategies for preventing overfitting





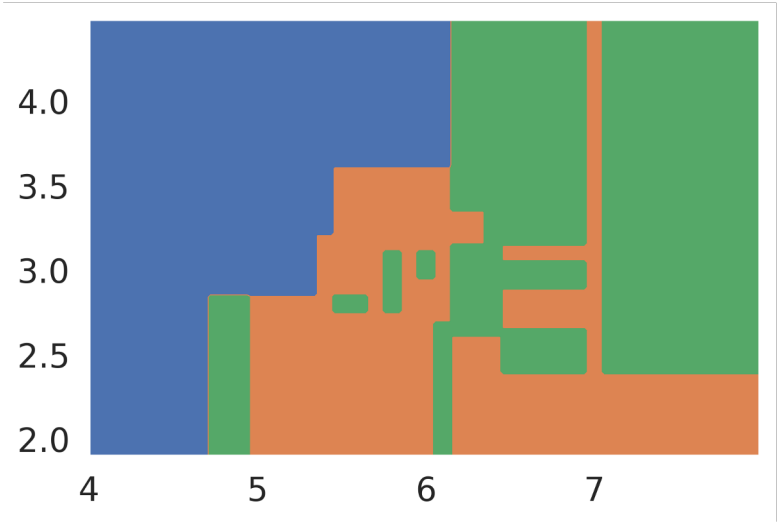
Restricting Decision Tree Complexity

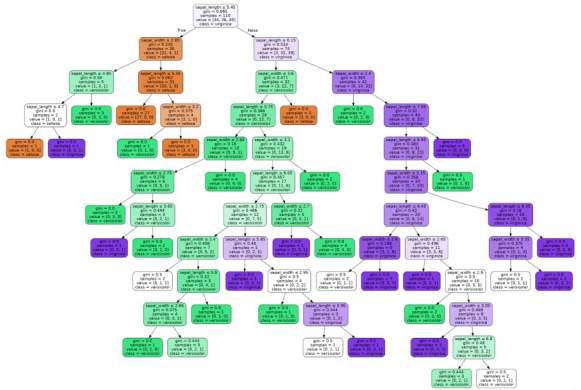
- Decision Tree Basics
- Decision Trees in scikit-learn
- Basic Decision Tree Generation
- Overfitting
- **Restricting Decision Tree Complexity**
- Random Forest

Overfitting and Our Algorithm

A “fully grown” decision tree built with our algorithm runs the risk of overfitting

One idea to avoid overfitting: Don’t allow fully grown trees

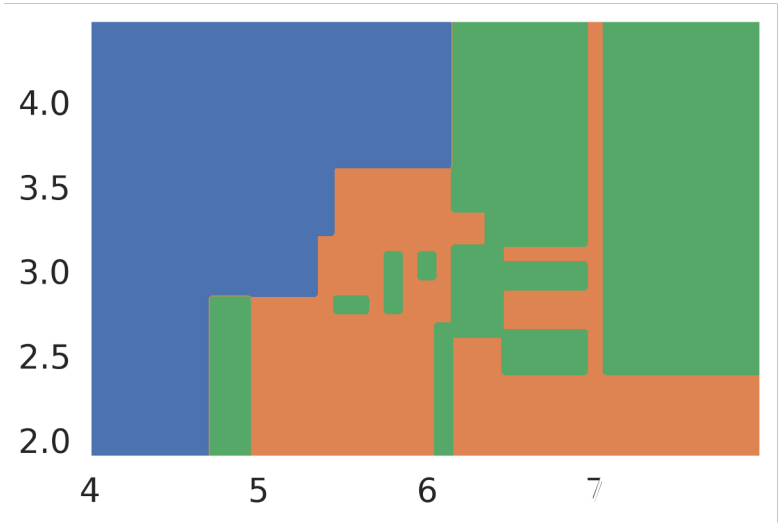


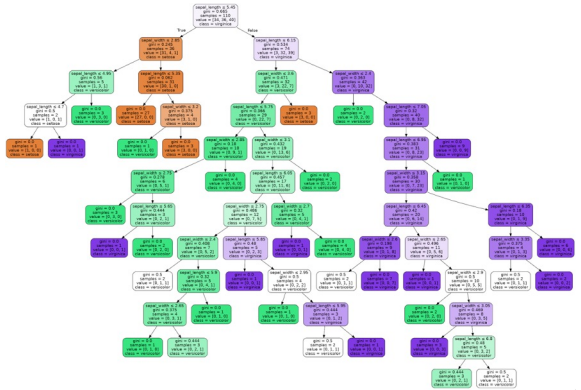


Regularization Terms and Decision Trees?

We can't just use the regularization term idea from linear models

- There is no global function being minimized
- Instead, the decision tree is built up node by node in a "greedy" fashion





Approach 1: Preventing Growth

Approach 1: Set one or more special rules to prevent growth

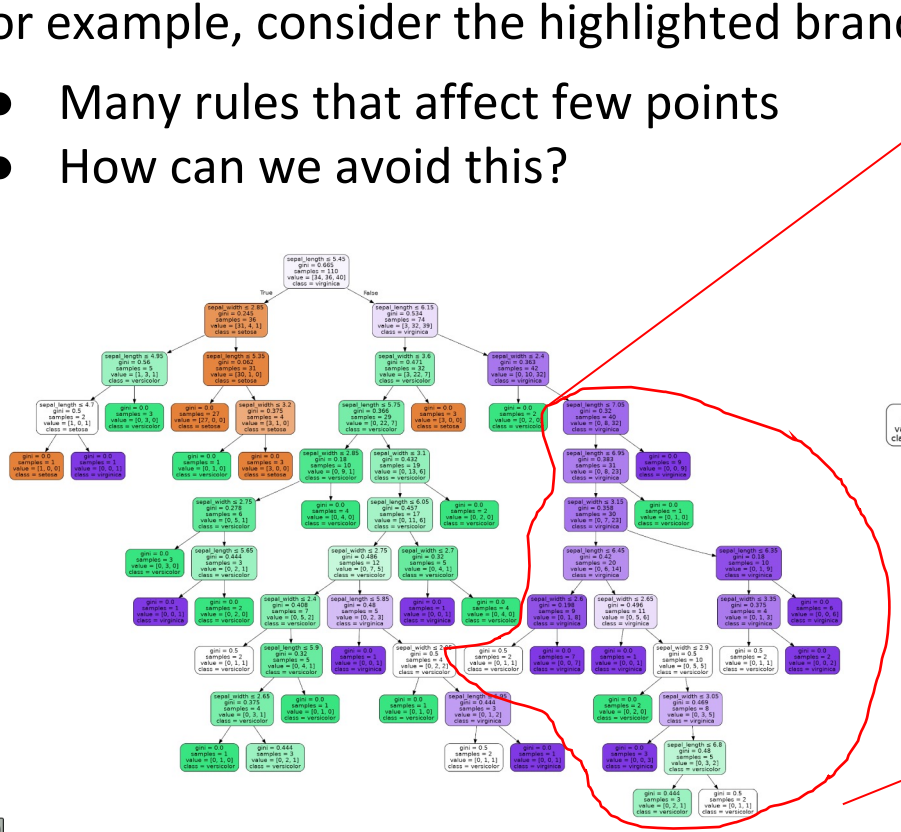
Examples:

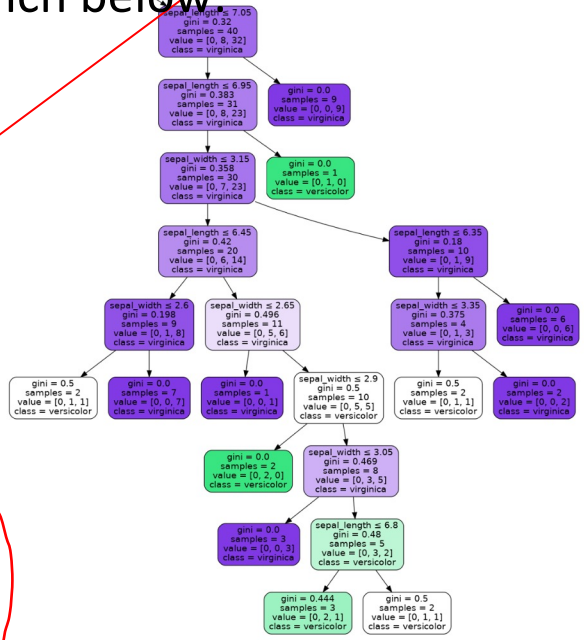
- Don’t split nodes with < 1% of the samples
- Don’t allow nodes to be more than 7 levels deep in the tree

Approach 2: Pruning

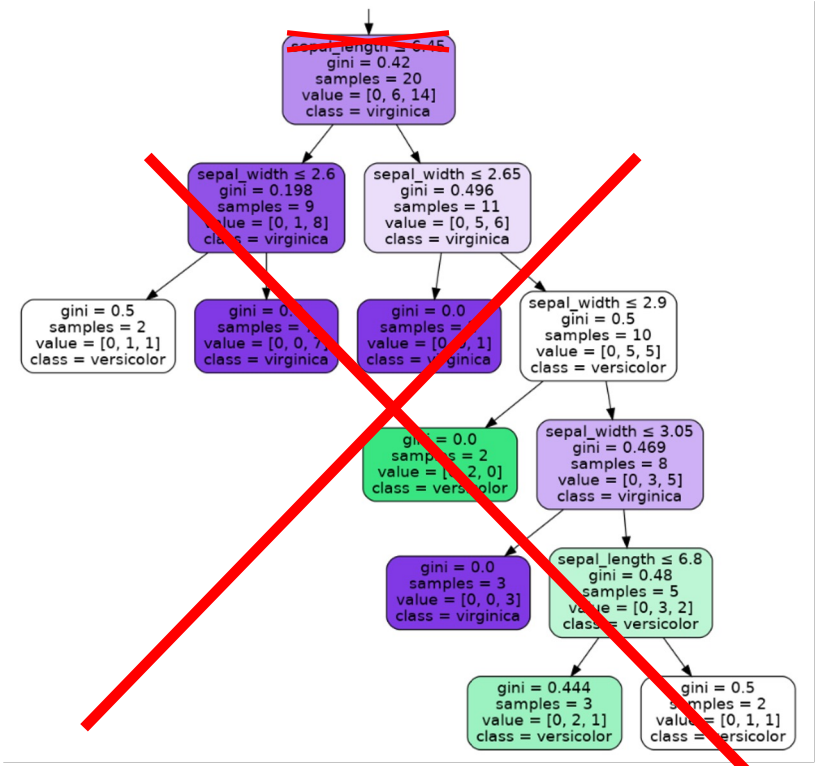
Approach 2: Let tree fully grow, then cut off less useful branches of the tree.
For example, consider the highlighted branch below:

- Many rules that affect few points
- How can we avoid this?





Specific Pruning Example



One way to prune:

- Before creating the tree, set aside a validation set
- If replacing a node by its most common prediction has no impact on the validation error, then don't split that node

Overfitting and Our Algorithm

A “fully grown” decision tree built with our algorithm runs the risk of overfitting

One idea to avoid overfitting: Don’t allow fully grown trees

- Approach 1: Set rules to prevent full growth
- Approach 2: Allow full growth then prune branches afterwards

Won’t discuss these in any great detail

- There’s a completely different idea called a “random forest” that is more popular and more beautiful

Random Forest

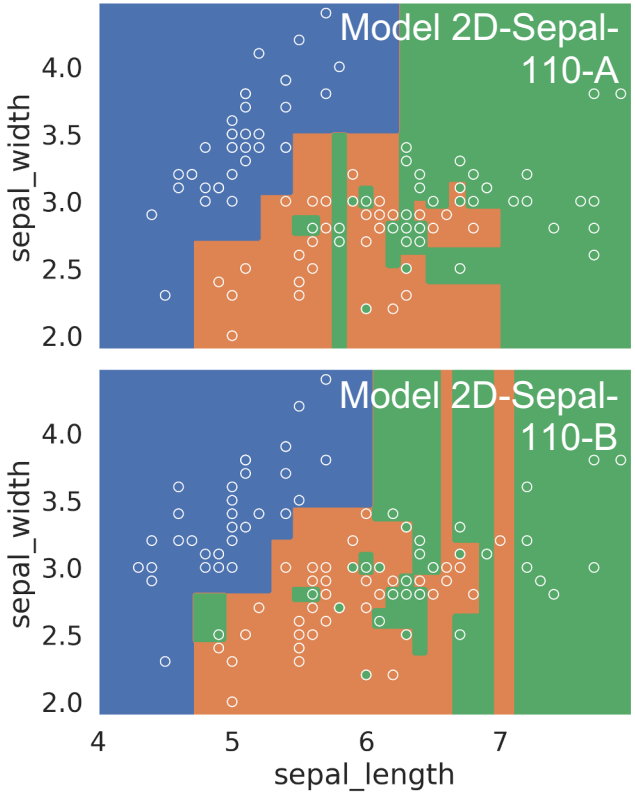
- Decision Tree Basics
- Decision Trees in scikit-learn
- Overfitting
- Basic Decision Tree Generation
- Restricting Decision Tree Complexity
- **Random Forest**

Random Forests: Harnessing Variance

As we've seen, fully-grown decision trees will almost always overfit data

- Low model bias, high model variance
- In other words, small changes in dataset will result in very different decision tree
- Example: Two models on the right trained on different subsets of the same data

Random Forest Idea: Build many decision trees and have them vote



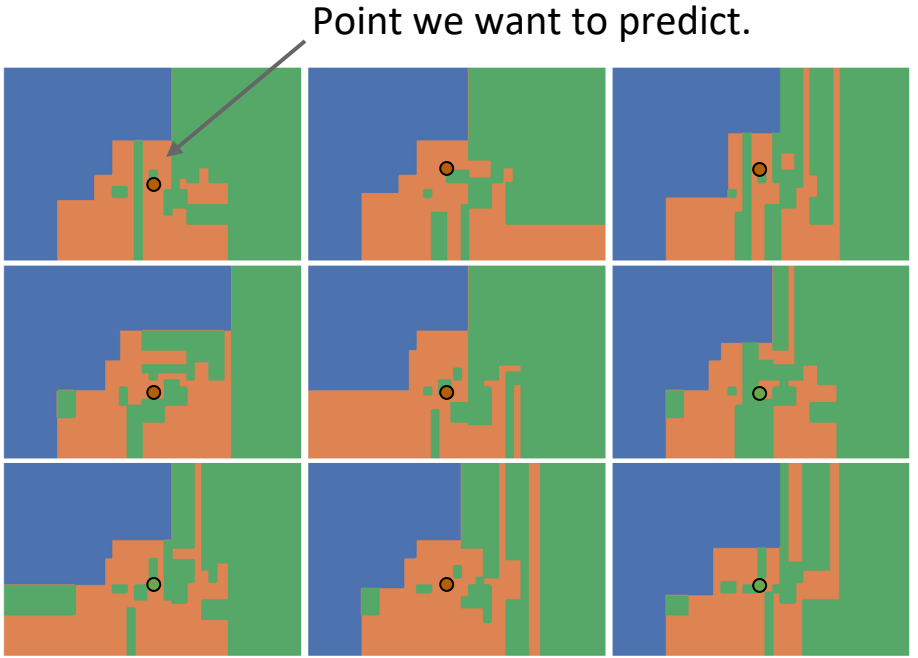
Random Forests: Harnessing Variance

What do we mean by vote?

- For a given x/y, use whichever prediction is most popular

Consider example at right with 9 models

- 6 votes orange, 3 votes green
- Random forest prediction is orange



Building Many Trees

Big fundamental problem: We only have one training set

How can we build many trees using one training set?

Building Many Trees

Big fundamental problem: We only have one training set.

How can we build many trees using one training set?

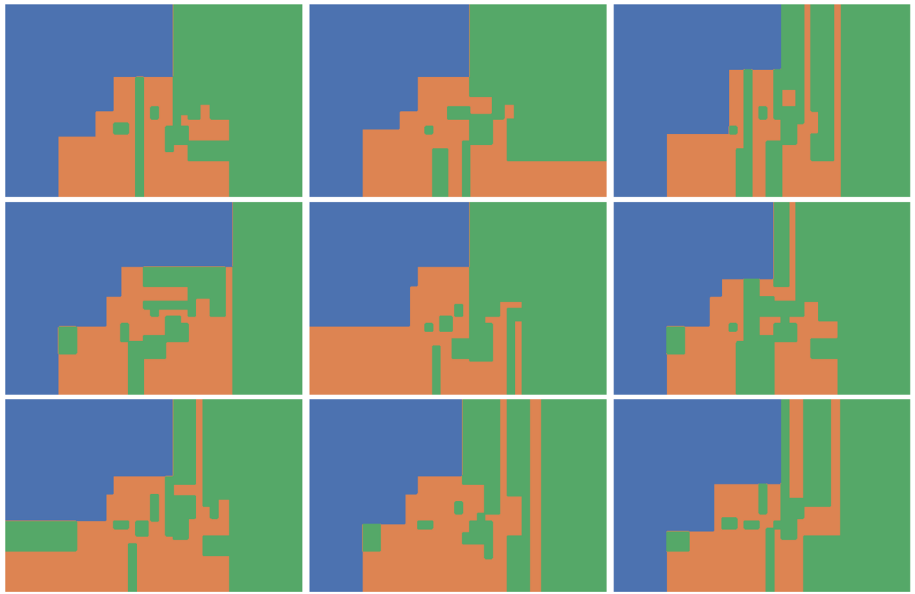
Bagging

Bagging: Short for Bootstrap AGGregatING

- Generate bootstrap resamples of training data
- Fit one model for each resample
- Final model = average predictions of each small model
- Invented by Leo Breiman in 1994

Random Forests

- Bagging often isn't enough to reduce model variance!
  - Decision trees often look very similar to each other
  - E.g., one strong feature always used for first split



Random Forests

- Bagging often isn't enough to reduce model variance!
  - Decision trees often look very similar to each other and thus make similar predictions
  - Ensemble will still have low bias and high model variance
  - E.g. one strong feature always used for first split
- Idea: Only use a sample of m features at each split
  - Usually m = sqrt(p) for decision trees used for classification
  - Here p is the number of features
- Algorithm creates individual trees, each overfit in a different way
  - The hope is that the the overall forest has low variance

Random Forests Algorithm

- *Bootstrap training* data **T** times. For each resample, fit a decision tree by doing the following:
  - Start with data in one node. Until all nodes pure:
  - Pick an impure node
  - *Pick a random subset of **m** features.* Pick the best feature x and split value β such that the loss of the resulting split is minimized, e.g. x = petal_width, β = 0.8 has loss 0.66
  - Split data into two nodes, one where $x < \beta$, and one where $x \geq \beta$
- To predict, ask the T decision trees for their predictions and take majority vote

This approach has two hyperparameters T and m

Avoiding Overfitting with Heuristics

We've seen many approaches to avoid overfitting decision trees

- Preventing growth
- Pruning
- Random forests

These ideas are generally "heuristic"

- Not provably best or mathematically optimal
- Instead, they are just ideas that somebody thought sounded good, implemented, then found to work in practice acceptably well

Why Random Forests?

- Versatile: does both regression and classification
- Invariant to feature scaling and translation
- Automatic feature selection
- Nonlinear decision boundaries without complicated feature engineering
- Doesn’t overfit as often as other nonlinear models (e.g. polynomial features)
- Example of **ensemble method**: combine the knowledge of many simple models to make a sophisticated model
- Example of using **bootstrap** to reduce model variance

Beyond Decision Trees for Classification

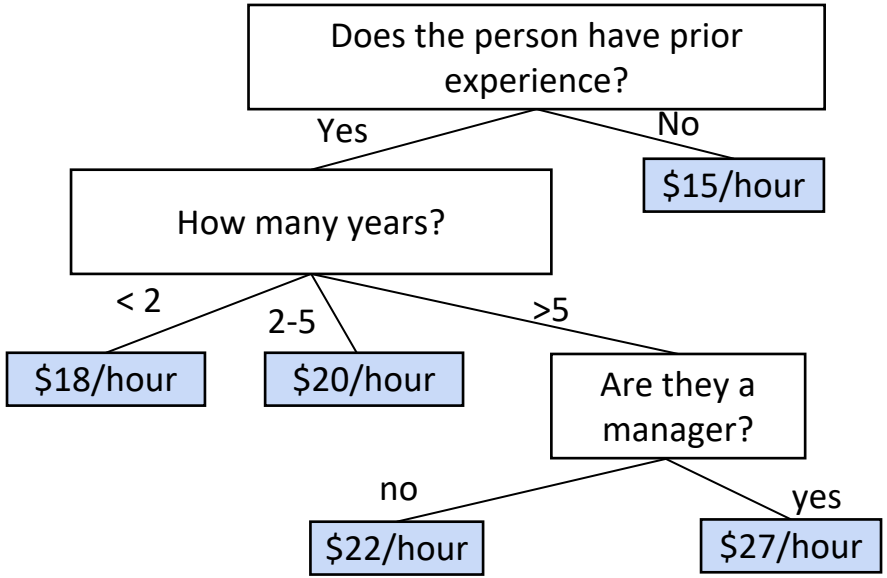
Trees for Regression

In earlier lectures we saw how we could use a logistic regression model for classification

- This lecture: We saw decision trees as an alternative technique for classification

We could do the same exercise for regression

- Rather than using a linear model, we could build a regression tree



Summary

Decision trees provide an alternate non-linear framework for classification and regression

- The underlying principle is fundamentally different
- Decision boundaries can be more complex
- Danger of overfitting is high

Keeping complexity under control is not nearly as mathematically elegant and relies on heuristic rules

- Hard constraints
- Pruning rules
- Random forests
  - Very interesting application of bootstrapping

Summary

In practice, you will see that there are actually even more conceptual frameworks for regression and classification:

- Linear models
- Decision trees
- Boosting (XGBoost, LightBoost)
- Nearest neighbors
- Support vector machines.
- Bayes Nets
- Perceptrons
  - Neural networks / deep learning
- And many many many many more