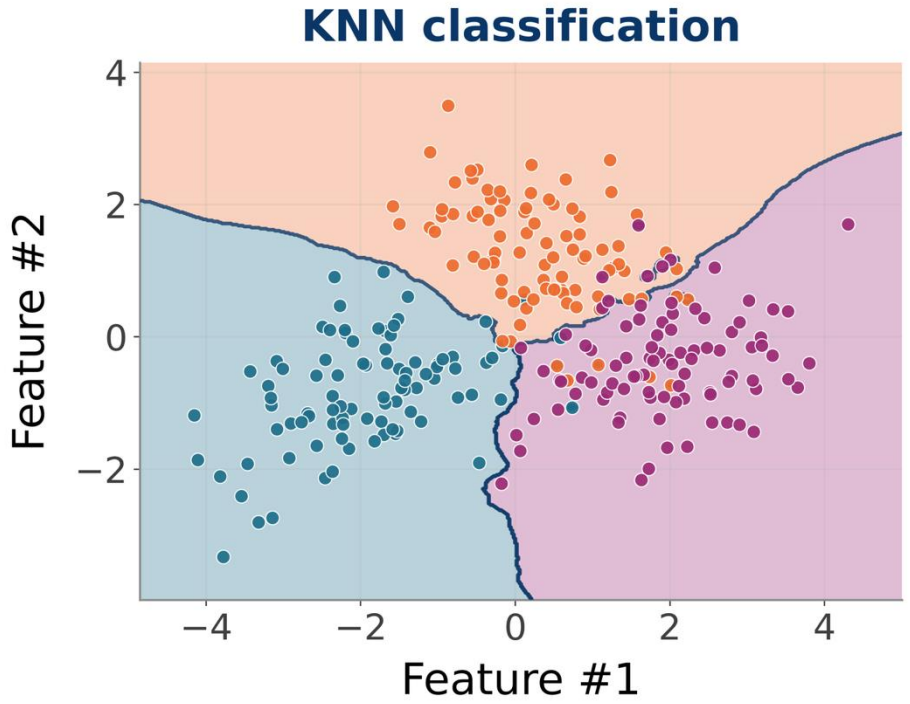
COM SCI M148, Lecture 12:
Decision Trees & Random Forests

Prof. Nathan Szymanski
nszymanski@seas.ucla.edu
Department of Materials Science

UCLA

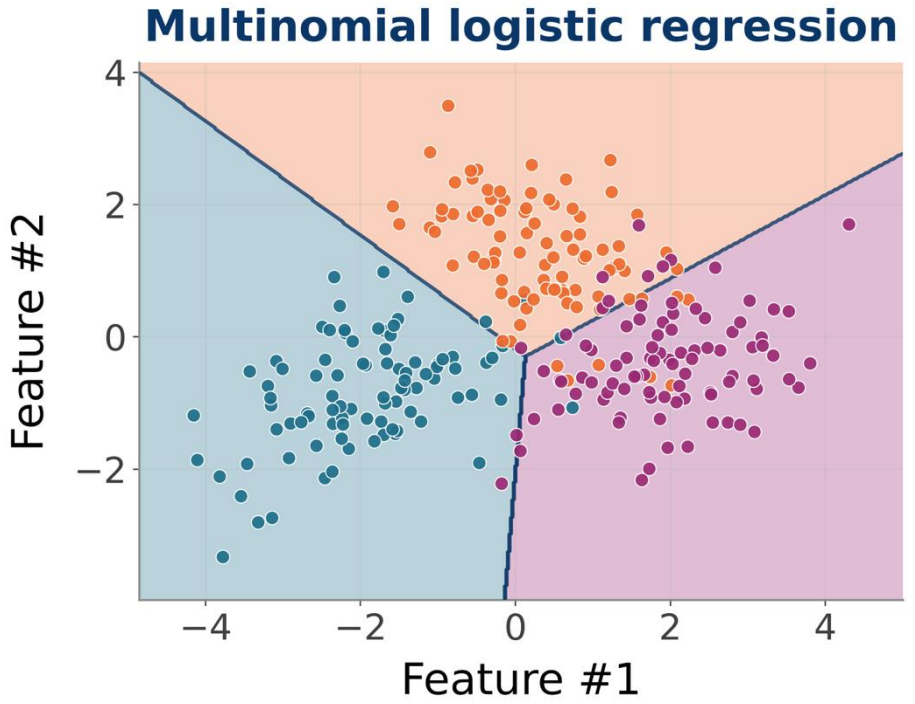
Samueli
School of Engineering

Classification models discussed so far…



This is a **non-parametric model**. We don't have to make any assumptions about $P(x)$ which maps features → classes

But we have to **save all training data**, since we use it every time we classify a new point

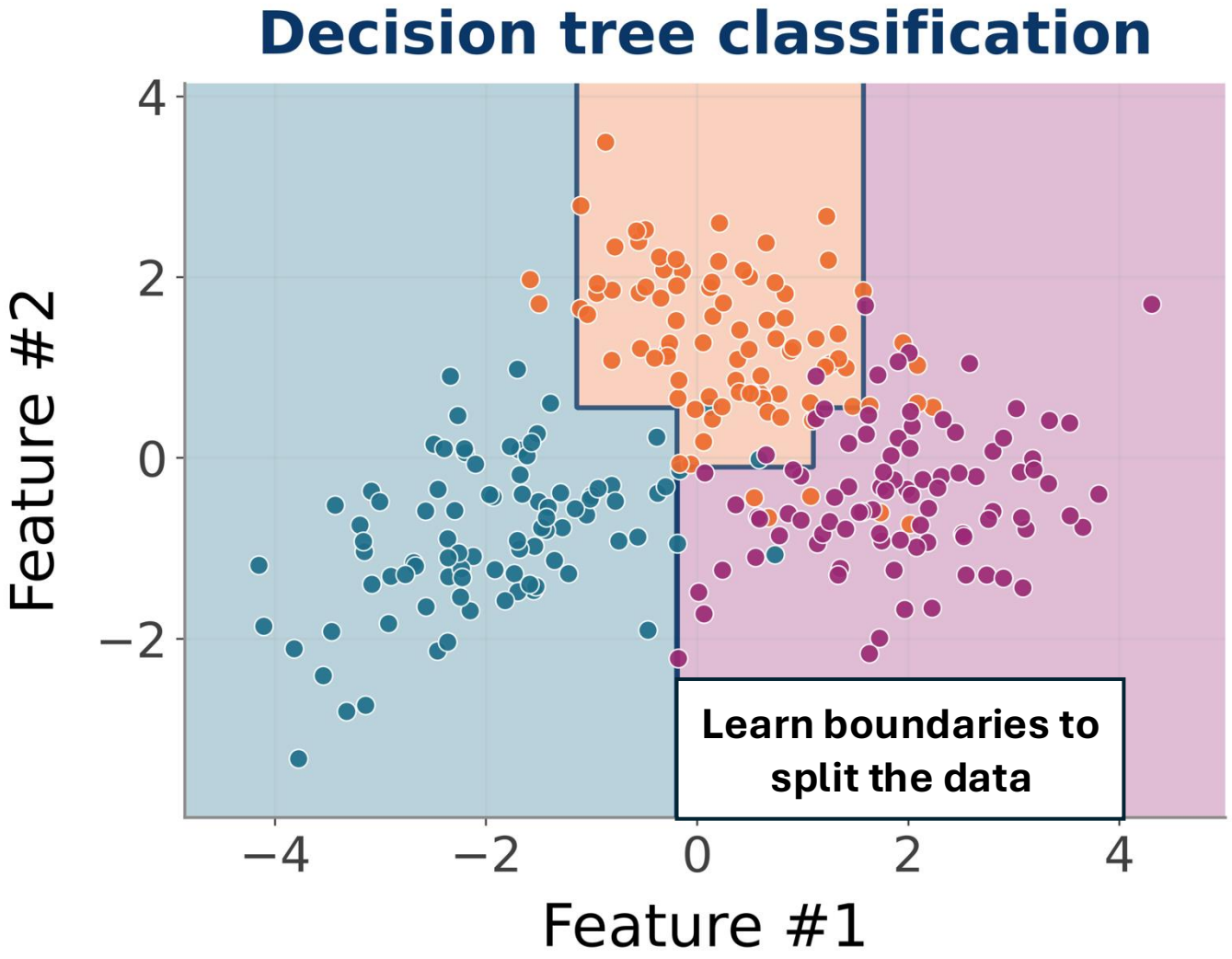


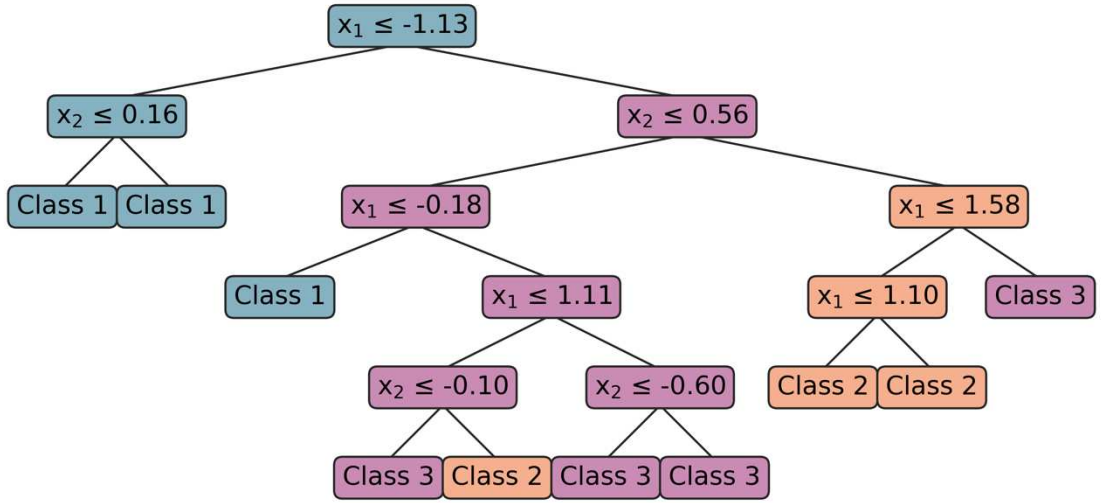
This is a **parametric model**. We assume a functional form:

$$\text{logit}(P) = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \cdots$$

Once trained, we can **throw away the training data** and just use the model

Today: Classification with decision trees





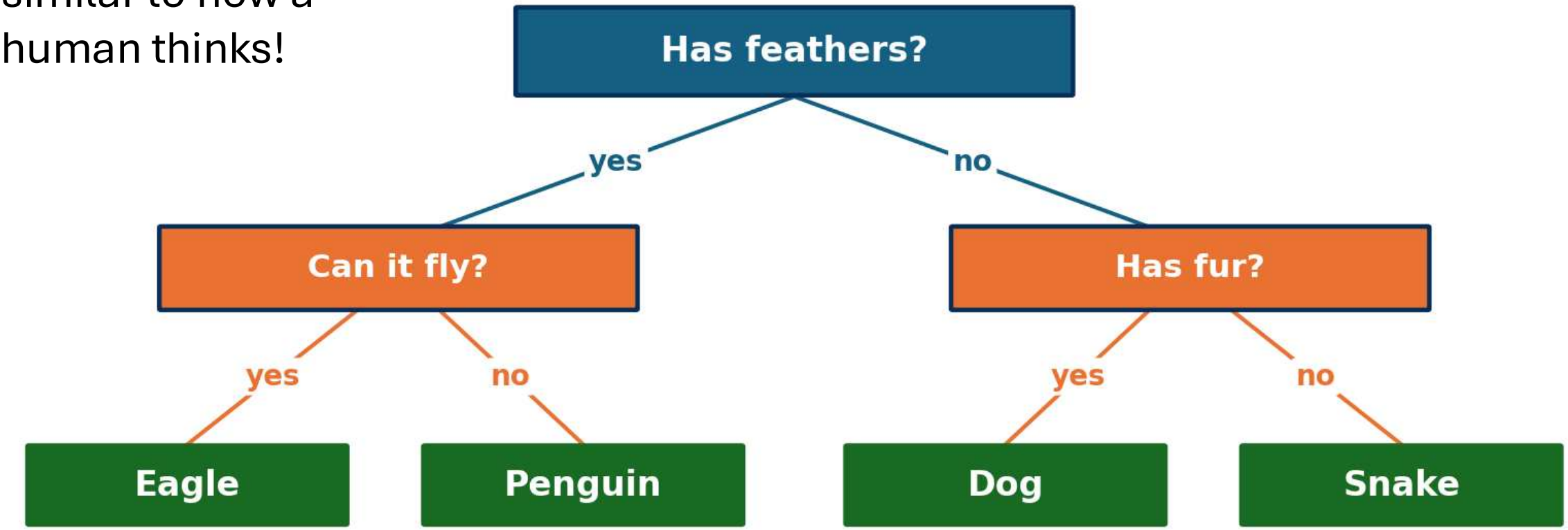
We learn **transferable rules** to decide which class a data point belongs to. This is **non-parametric** but we also **don't have to save all the data** once training is done!

A simple example

Think of decision trees as a **series of yes/no questions.** This is similar to how a human thinks!

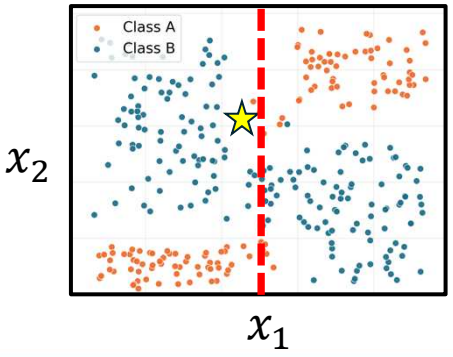


What is this animal?



The same idea applies to numerical data

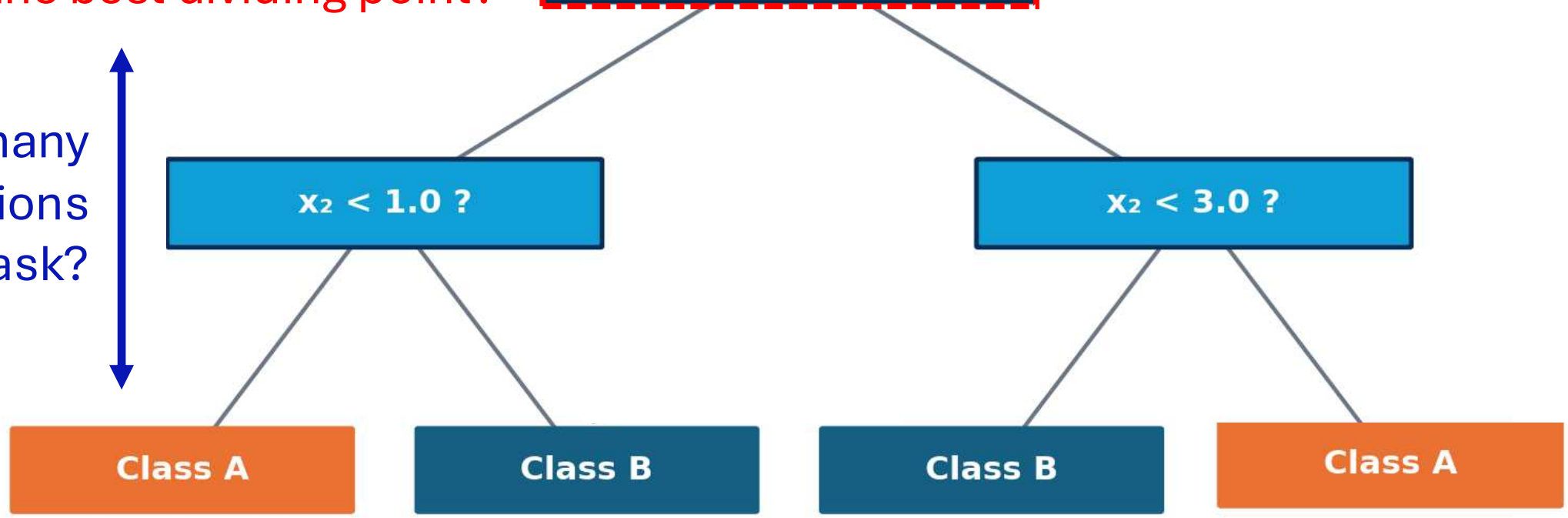
Which class does a new data point belong to?



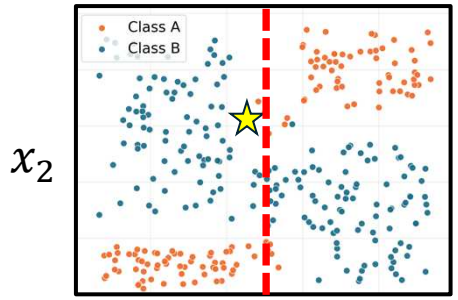
Model training is then just a matter of **coming up with the "right" questions…**

How do we know $x_1 < 2.5$ is the best dividing point?

How many questions to ask?



Some terminology



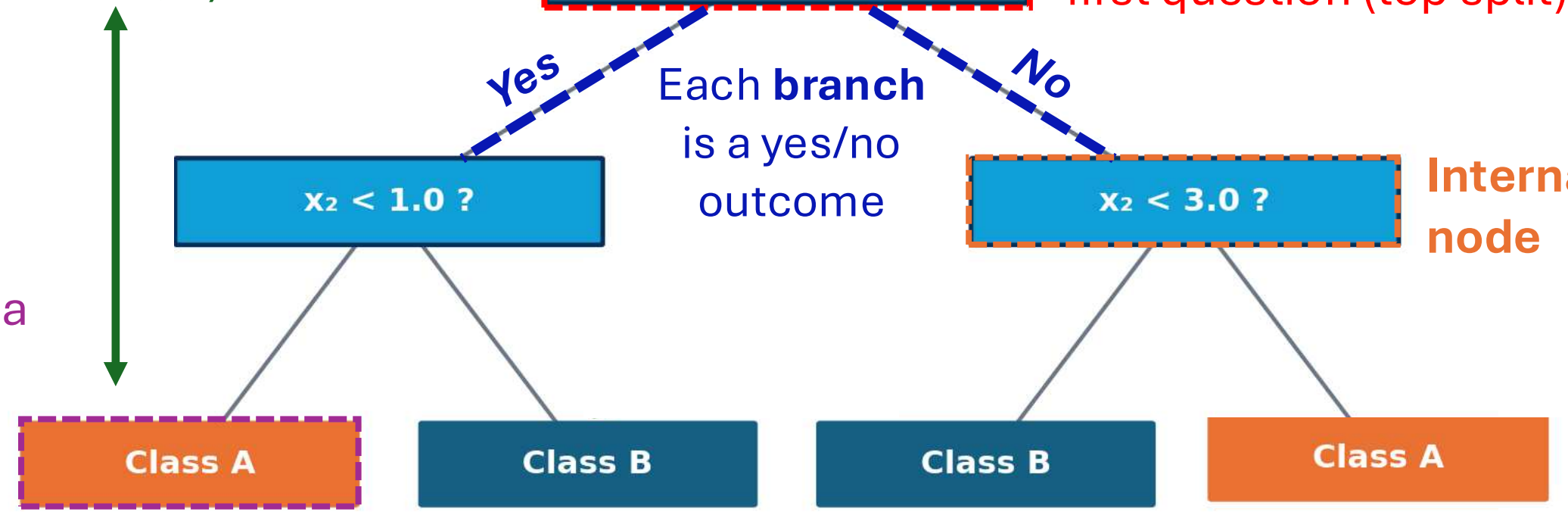
The **depth** is the number of branches from root to leaf (in this case, 2)

The **root node** is the very first question (top split)

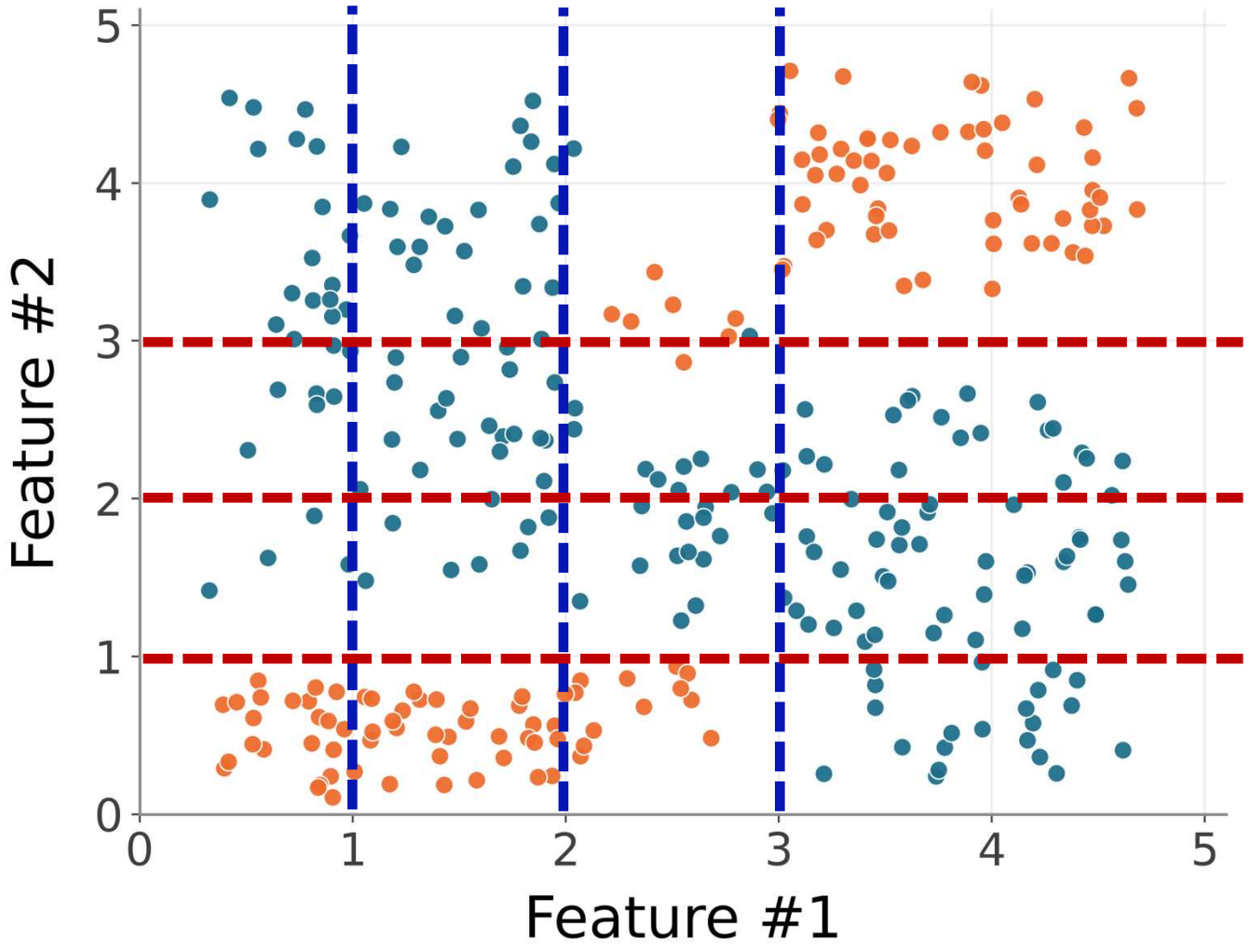
Each **branch** is a yes/no outcome

Internal node

A **leaf** is a terminal node → **a class**



Training a decision tree



Top-down approach:

- Start with all data in the root node

$$x_i < ?$$

- Consider possible questions
  - Which feature?
  - What threshold?

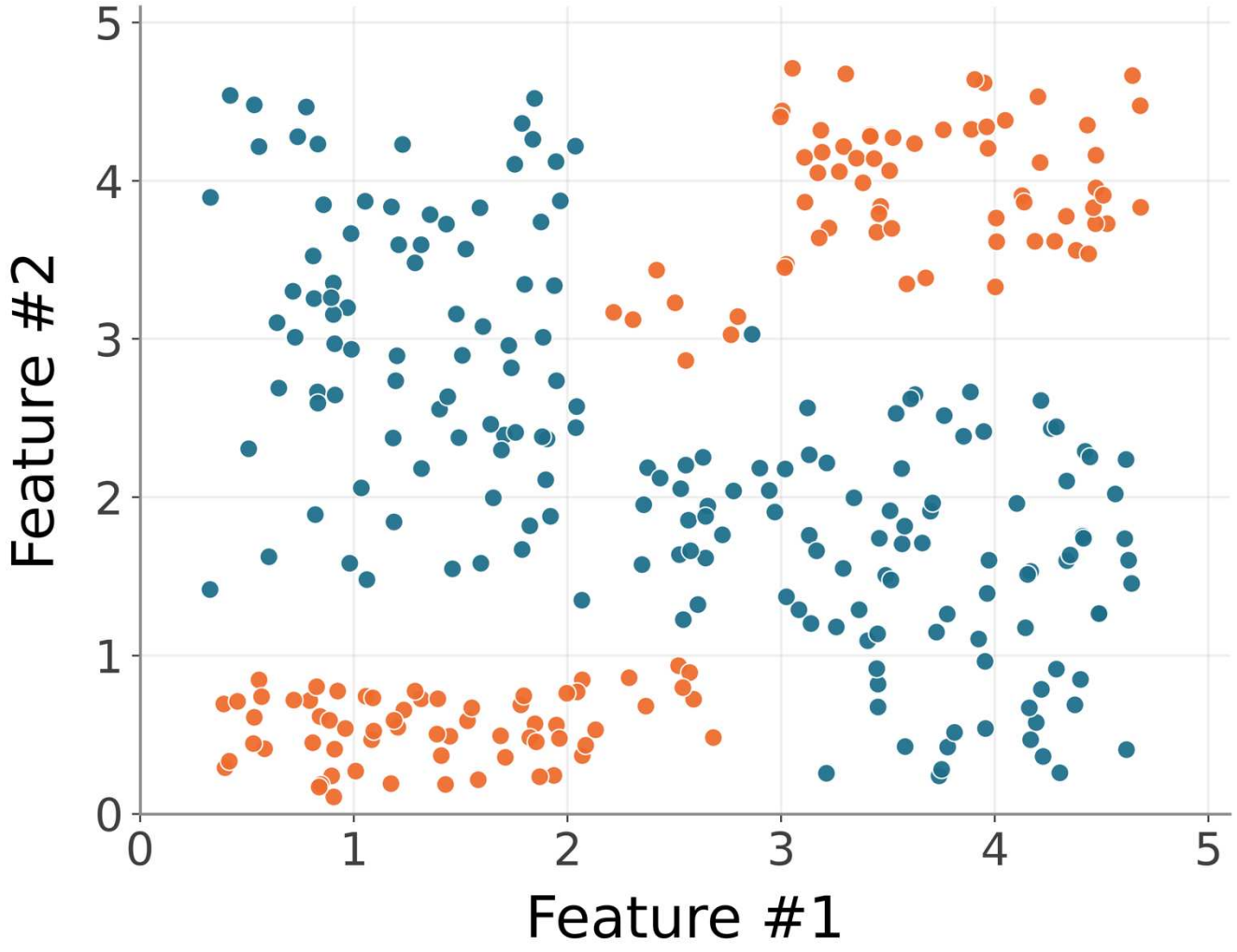
$x_1 < 1$ $x_2 < 1$

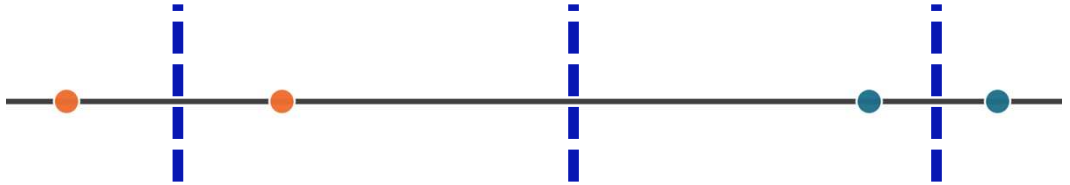
$x_1 < 2$ $x_2 < 2$

$x_1 < 3$ $x_2 < 3$

And so on for each feature

Training a decision tree





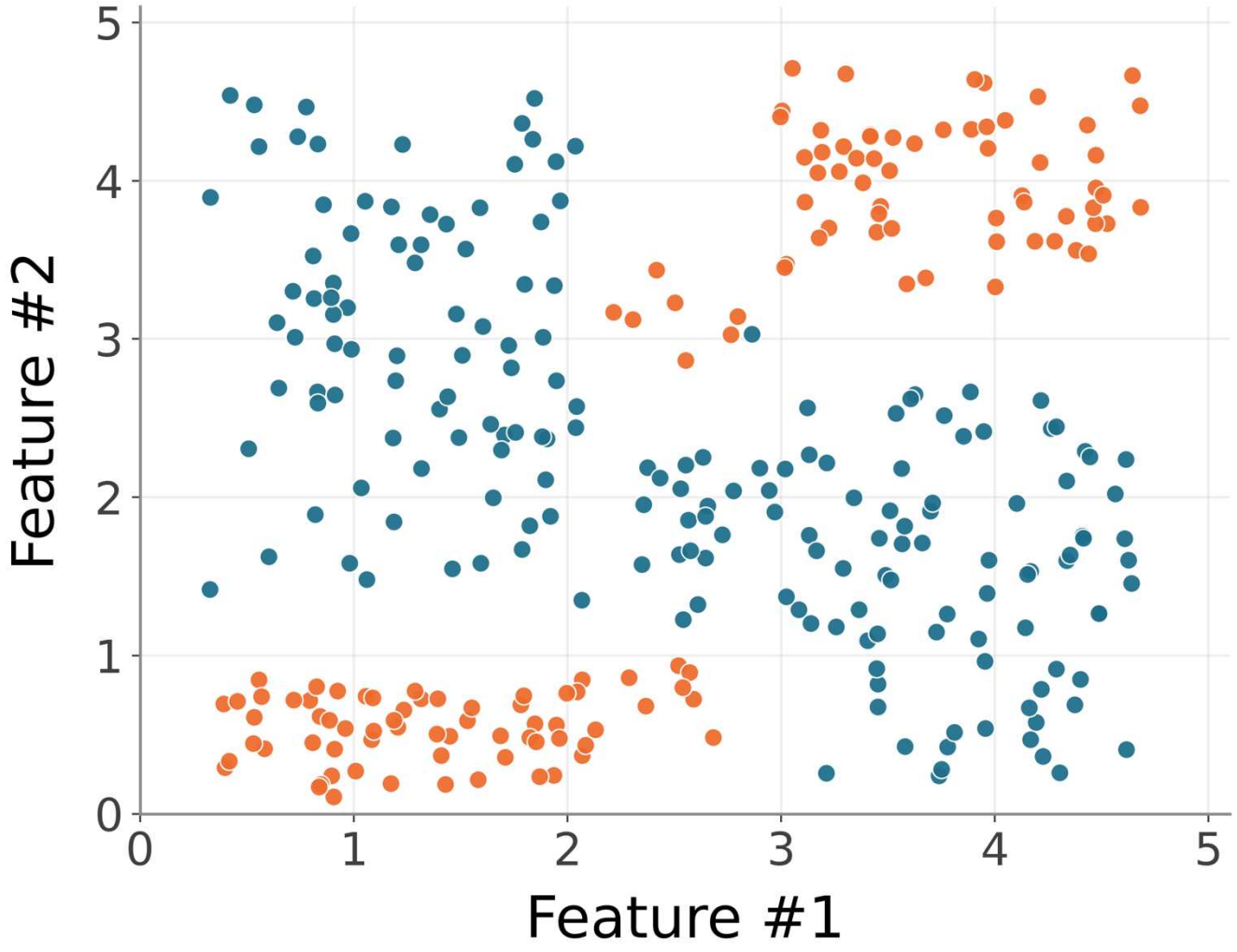
In practice, we test **midpoints** between neighboring samples

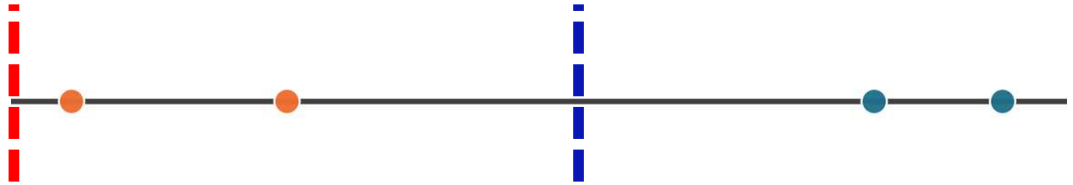
- What threshold?

$x_1 < 1$	$x_2 < 1$
$x_1 < 2$	$x_2 < 2$
$x_1 < 3$	$x_2 < 3$

And so on for each feature

Training a decision tree





In practice, we test **midpoints** between neighboring samples

The key is to pick the split that leads to the "**purest**" child nodes

A completely **pure** node: all samples belong to one class

A maximally **impure** node has a 50/50 mix of classes

Measuring node impurity

The Gini impurity:

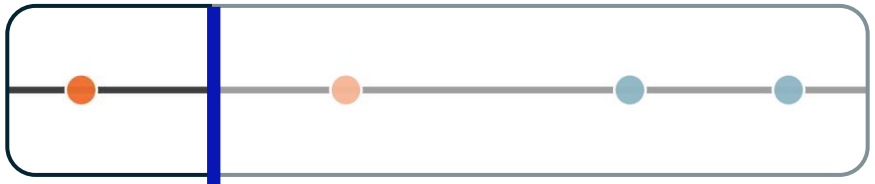
$$G = 1 - \sum_k p_k^2$$

$$p_k = \frac{\textbf{Number of samples in class } k}{\textbf{Total number of samples in the node}}$$

(sum this over all classes)

Measures how "mixed" samples are within a node

(**each node gets its own Gini impurity**)



$$G_{\text{left}} = 1 - \left(\left(\frac{1}{1}\right)^2 + \left(\frac{0}{1}\right)^2\right) = 0$$

1 sample in orange class

None in blue class

Measuring node impurity

The Gini impurity:

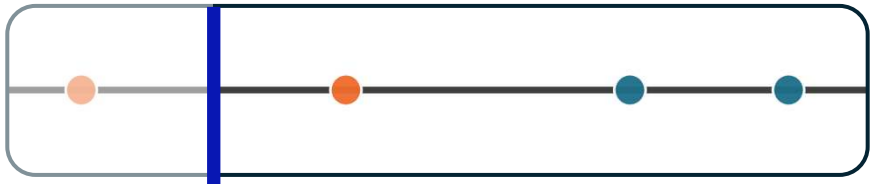
$$G = 1 - \sum_k p_k^2$$

$$p_k = \frac{\textbf{Number of samples in class } k}{\textbf{Total number of samples in the node}}$$

(sum this over all classes)

Measures how "mixed" samples are within a node

(**each node gets its own Gini impurity**)



Now: 3 samples total

$$G_{\text{left}} = 1 - \left(\left(\frac{1}{1}\right)^2 + \left(\frac{0}{1}\right)^2\right) = 0$$

$$G_{\text{right}} = 1 - \left(\left(\frac{1}{3}\right)^2 + \left(\frac{2}{3}\right)^2\right) = \frac{5}{9}$$

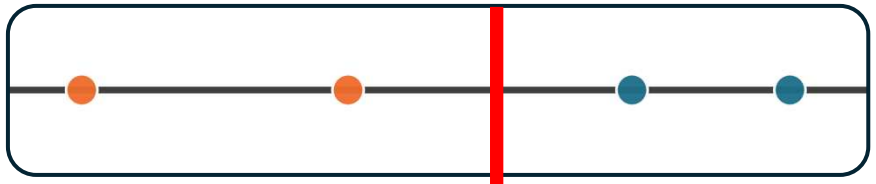
Measuring node impurity

The Gini impurity:

$$G = 1 - \sum_k p_k^2$$

$$p_k = \frac{\text{Number of samples in class } k}{\text{Total number of samples in the node}}$$

What if we split at a different point?

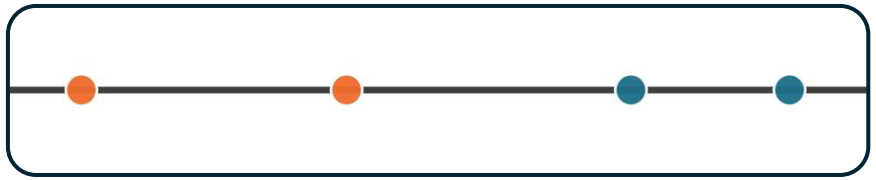


$$G_{\text{left}} = 1 - \left(\left(\frac{2}{2}\right)^2 + \left(\frac{0}{2}\right)^2 \right) = 0$$

$$G_{\text{right}} = 1 - \left(\left(\frac{0}{2}\right)^2 + \left(\frac{2}{2}\right)^2 \right) = 0$$

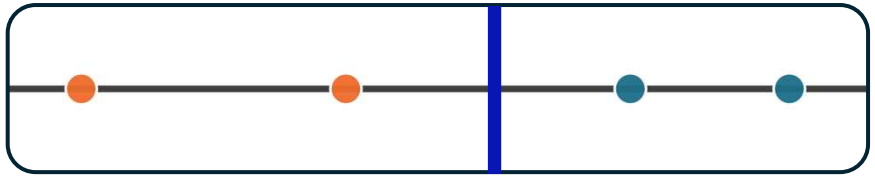
The information gain (IG): *Change* in Gini impurity

Before split (parent node)



$$G_{\text{parent}} = 1 - \left(\left(\frac{2}{4}\right)^2 + \left(\frac{2}{4}\right)^2 \right) = \frac{1}{2}$$

After split (children nodes)



$$G_{\text{left}} = 0 \qquad G_{\text{right}} = 0$$

$$\text{IG} = G_{\text{parent}} - \left(\frac{n_{\text{left}}}{n_{\text{tot}}} G_{\text{left}} + \frac{n_{\text{right}}}{n_{\text{tot}}} G_{\text{right}} \right)$$

The information gain (IG)

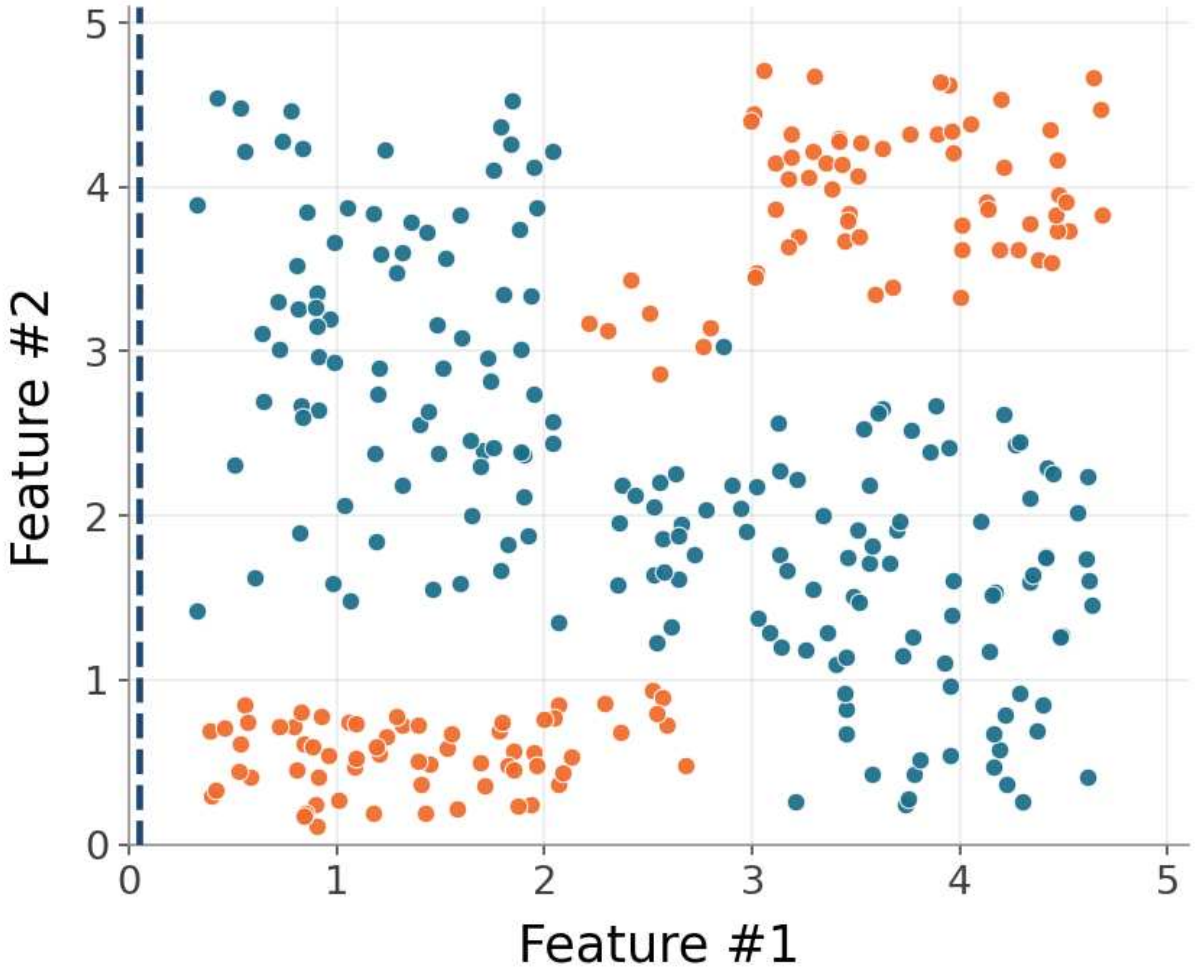
We'd like to **choose a split** that leads to largest reduction in impurity. In other words, **maximal information gain.**

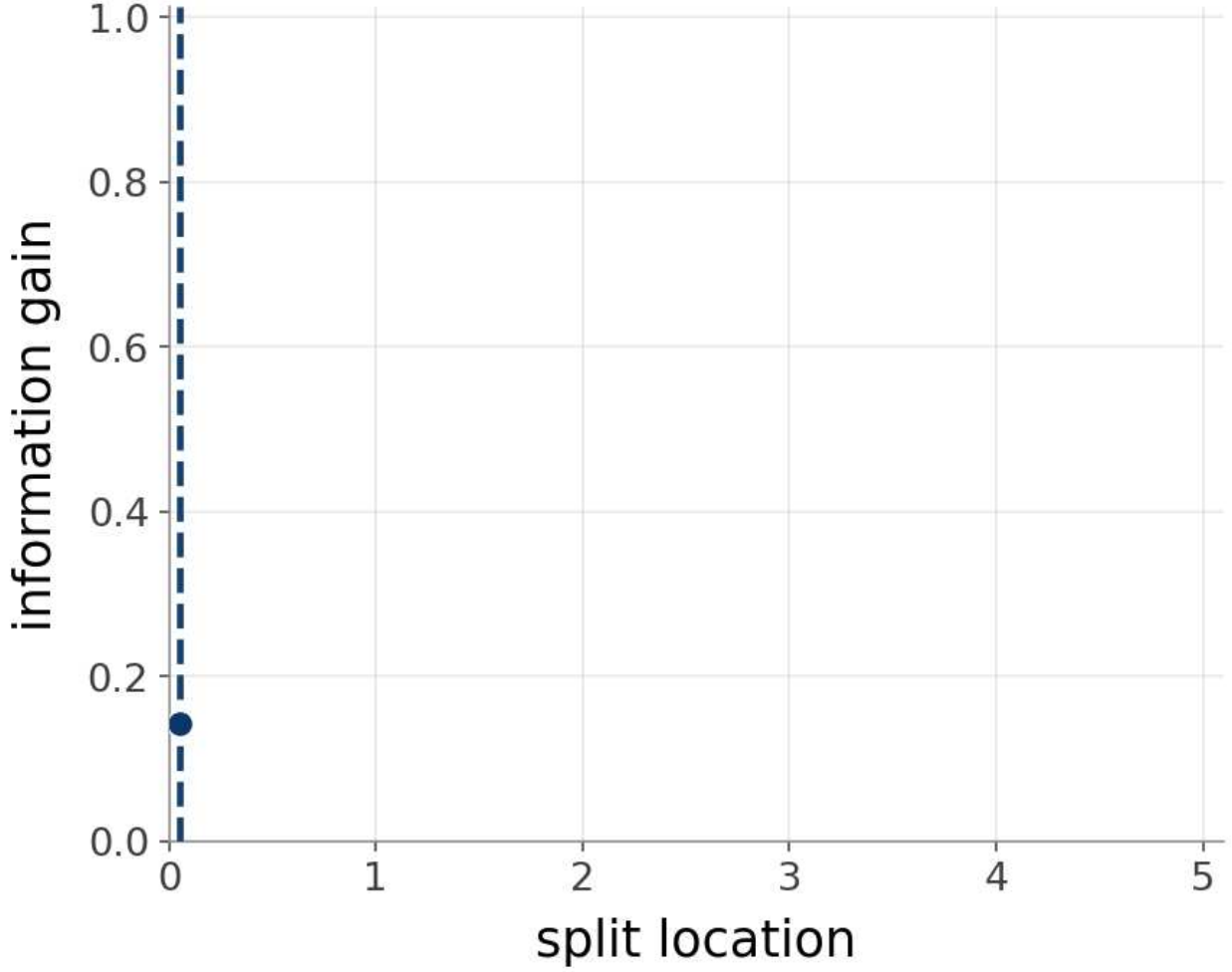
In this example:

$$IG = \frac{1}{2} - \left(\frac{2}{4}(0) + \frac{2}{4}(0) \right) = \frac{1}{2}$$

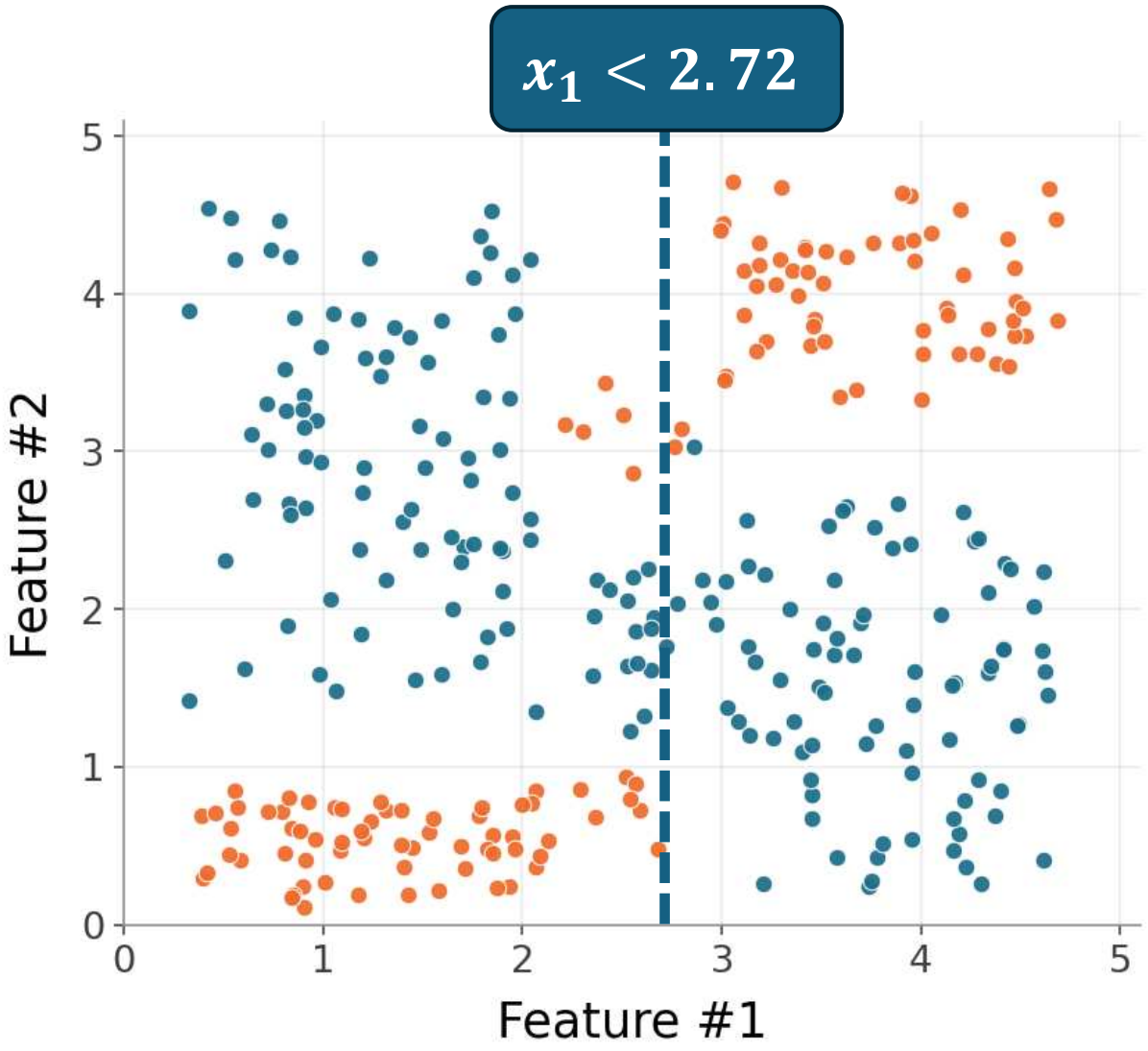
Maximizing the information gain

We can do a sweep over both features and monitor the information gain

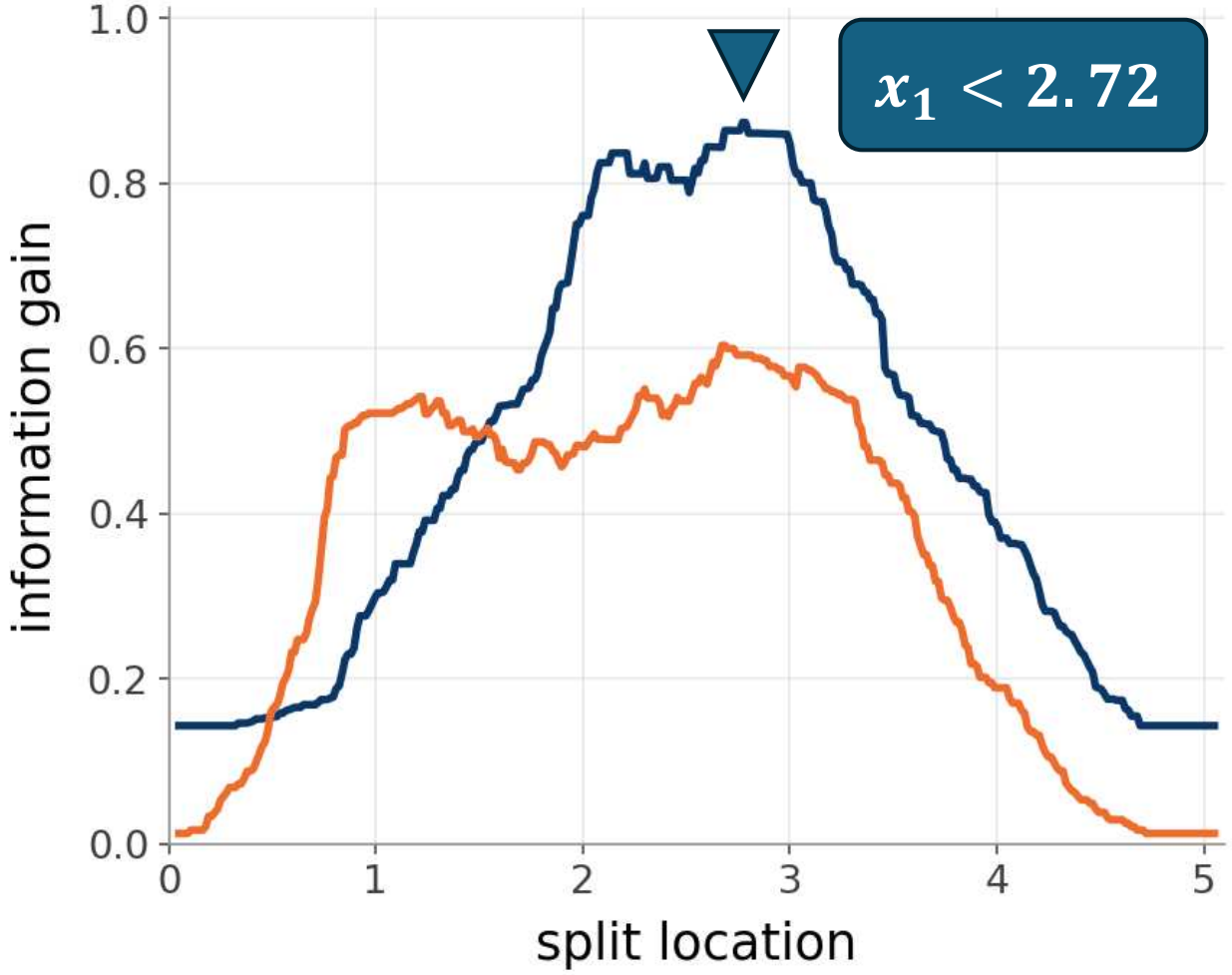




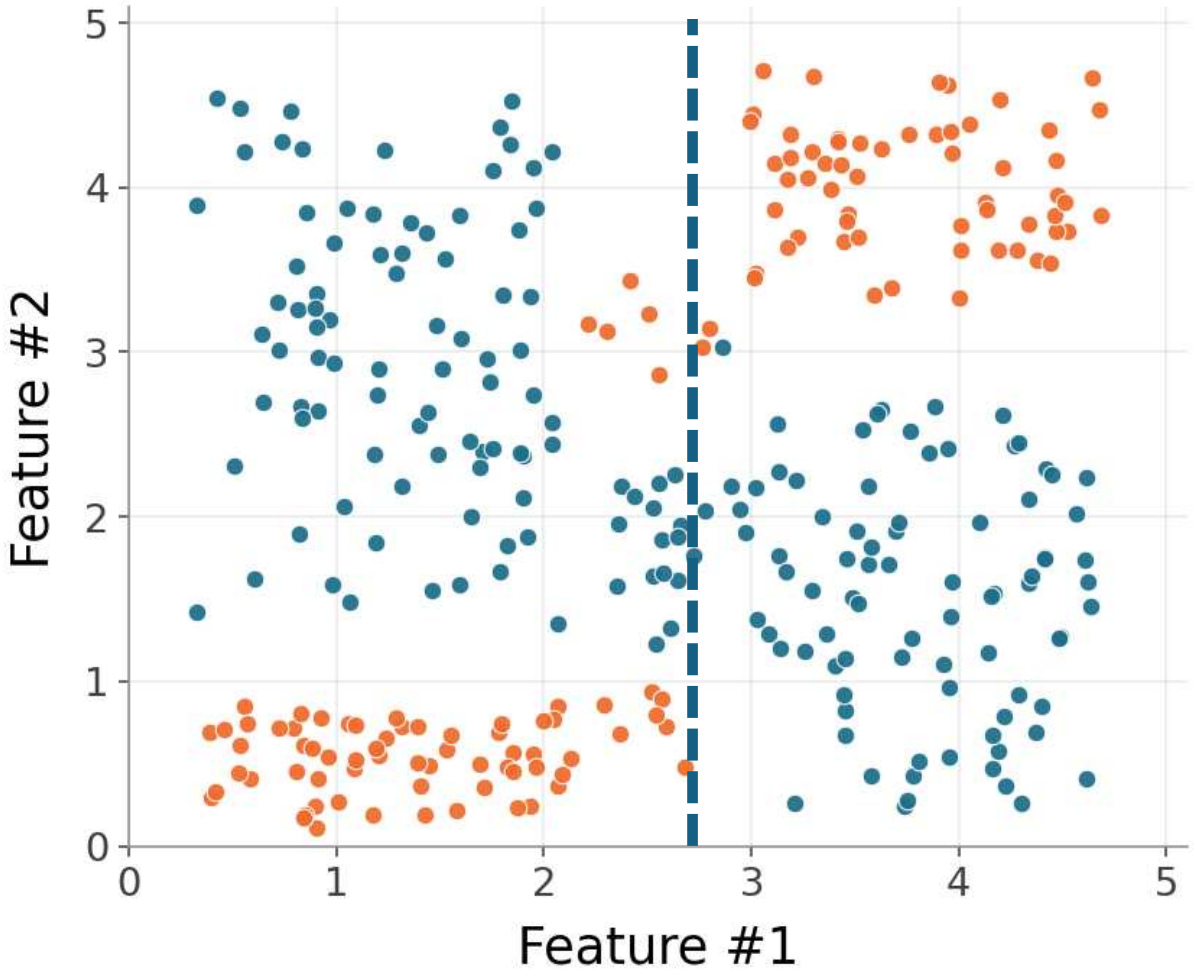
Maximizing the information gain



This is the best choice!

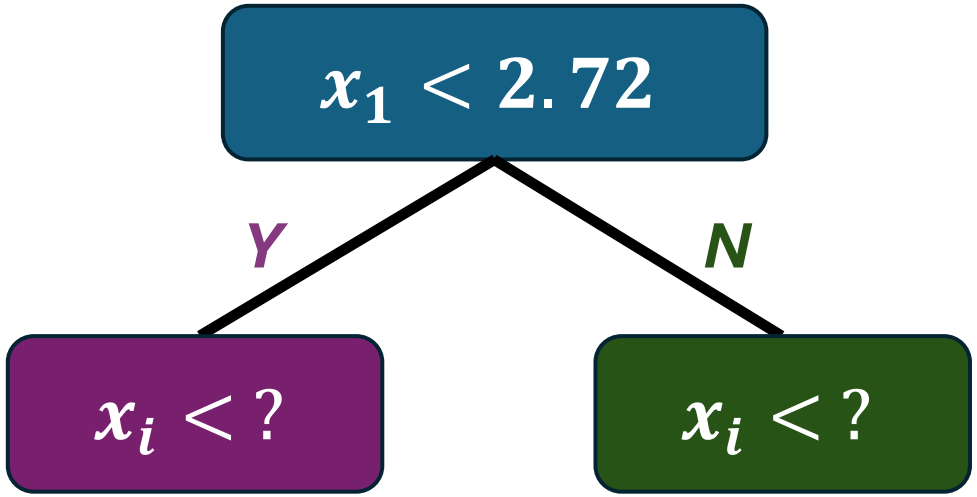


We have our first decision!



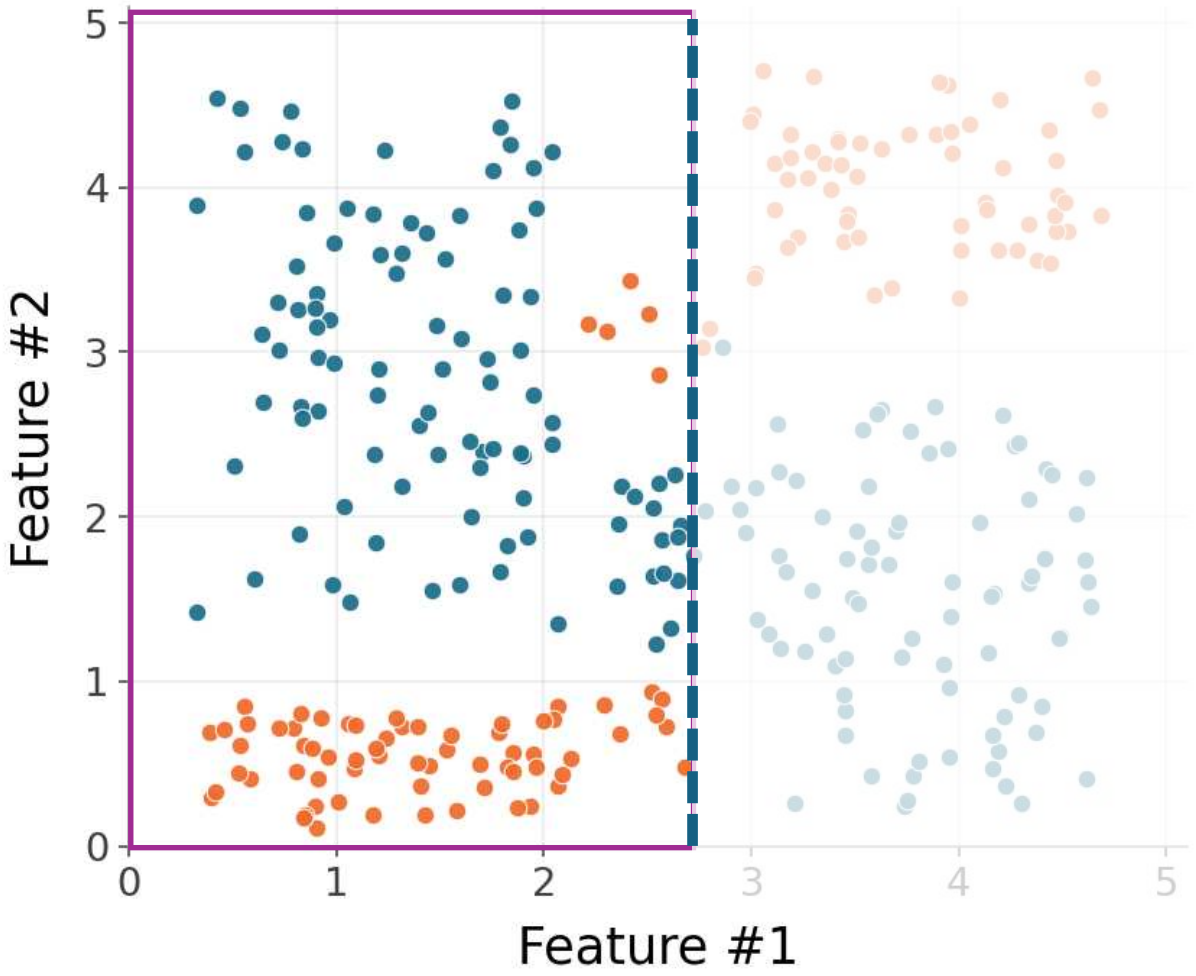
Top-down approach:

- Start with all data in the root node



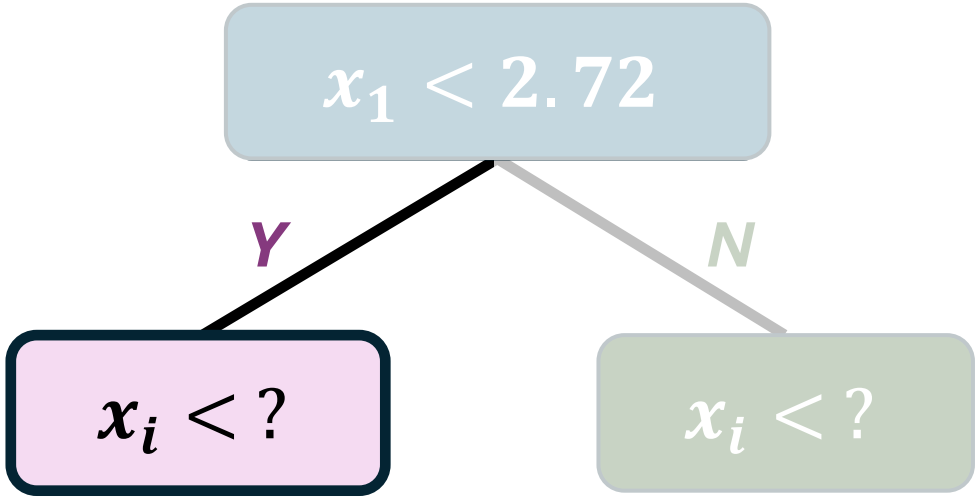
Then, simply **repeat the same process for each child node!**

Onto the next child node…



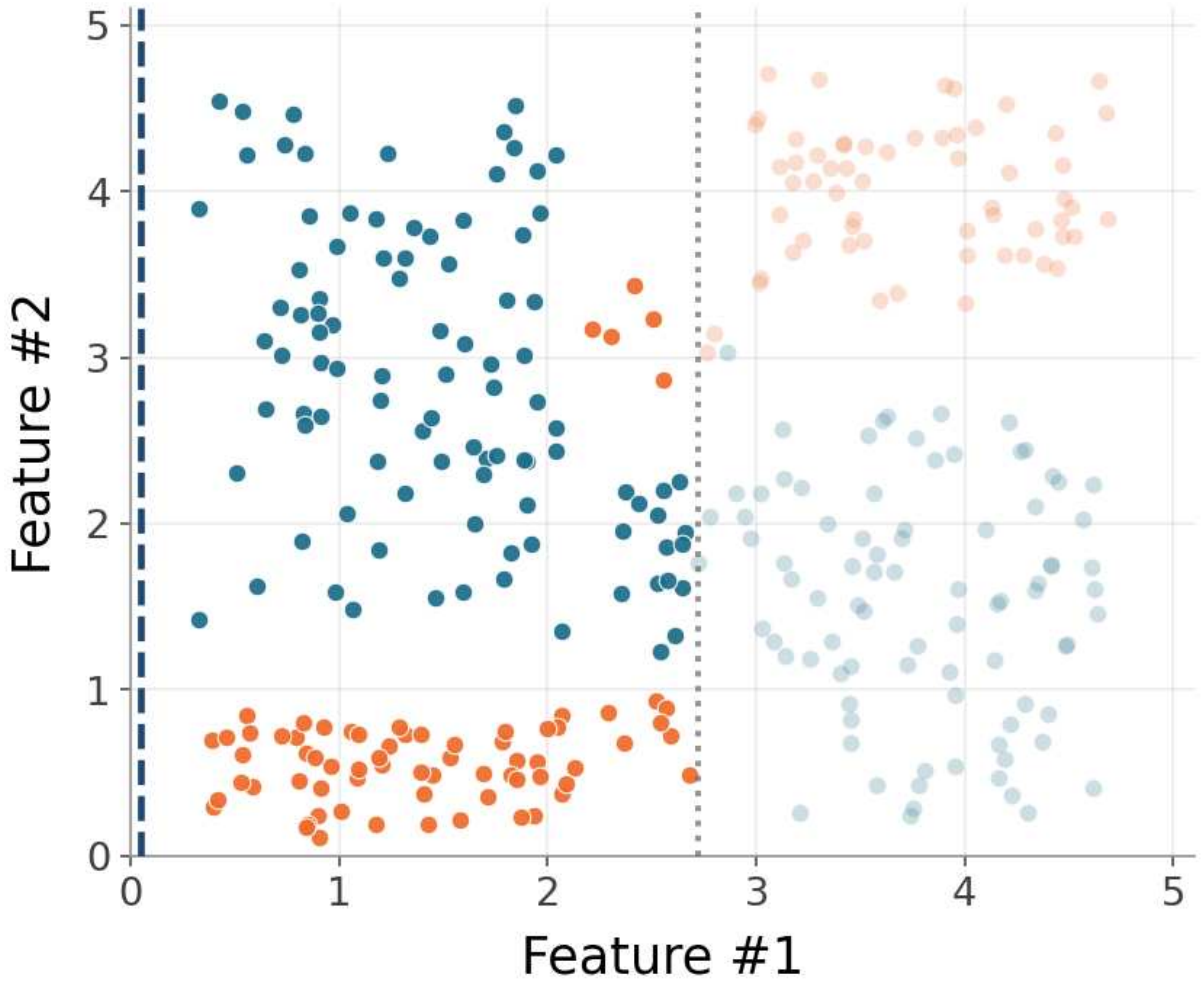
Top-down approach:

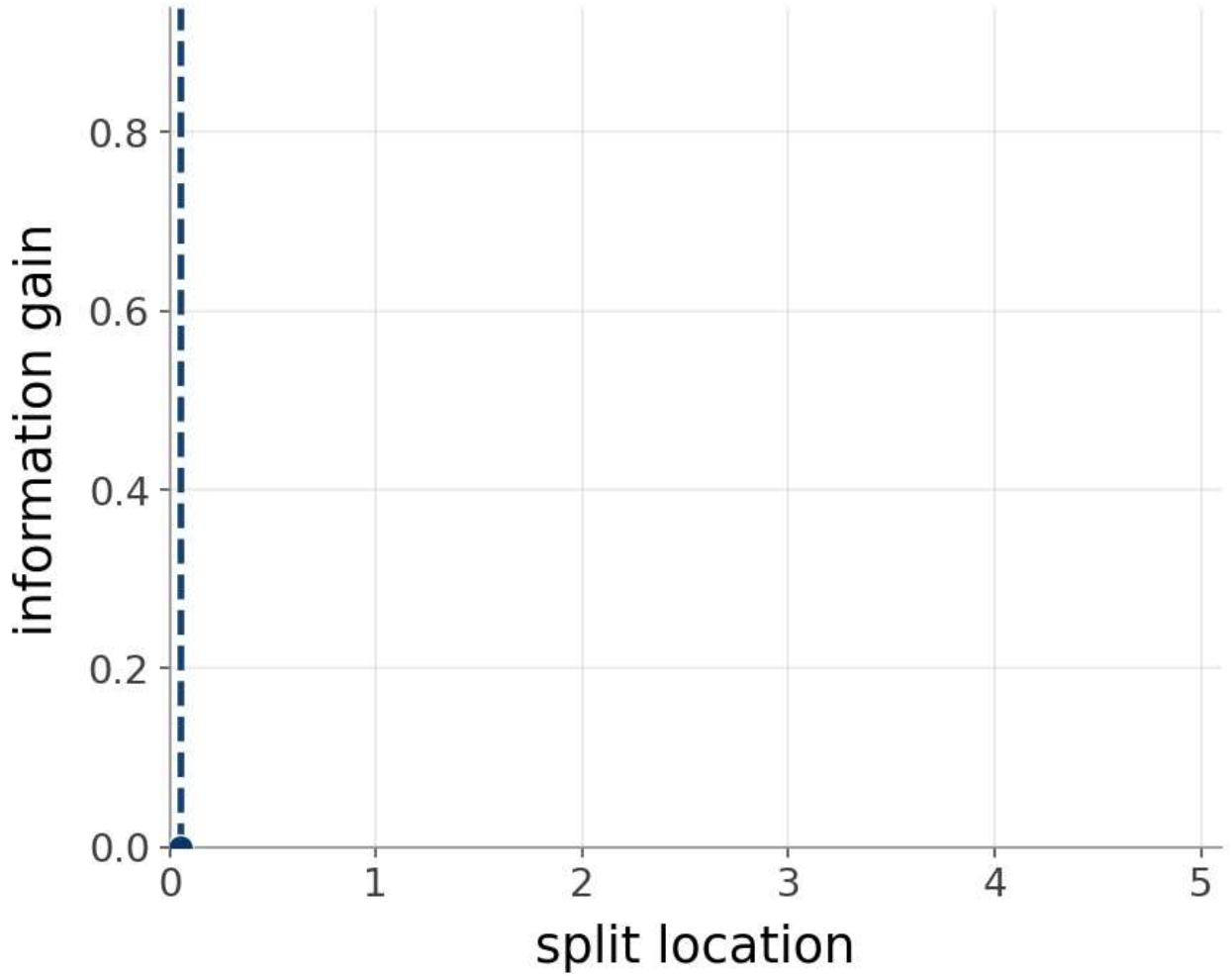
- Start with all data in the root node



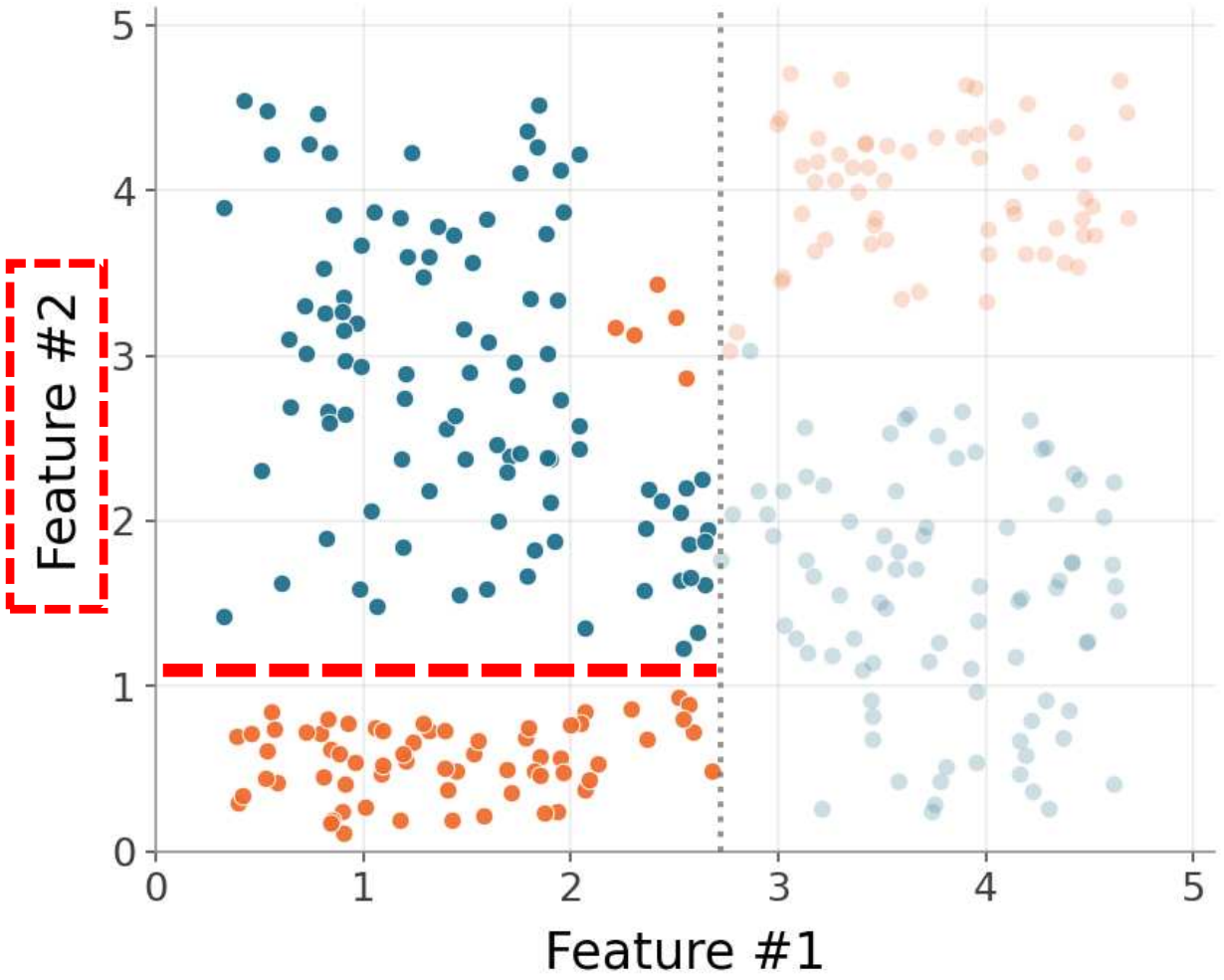
But now, we look at a subset of the data. **Decision trees partition the space.**

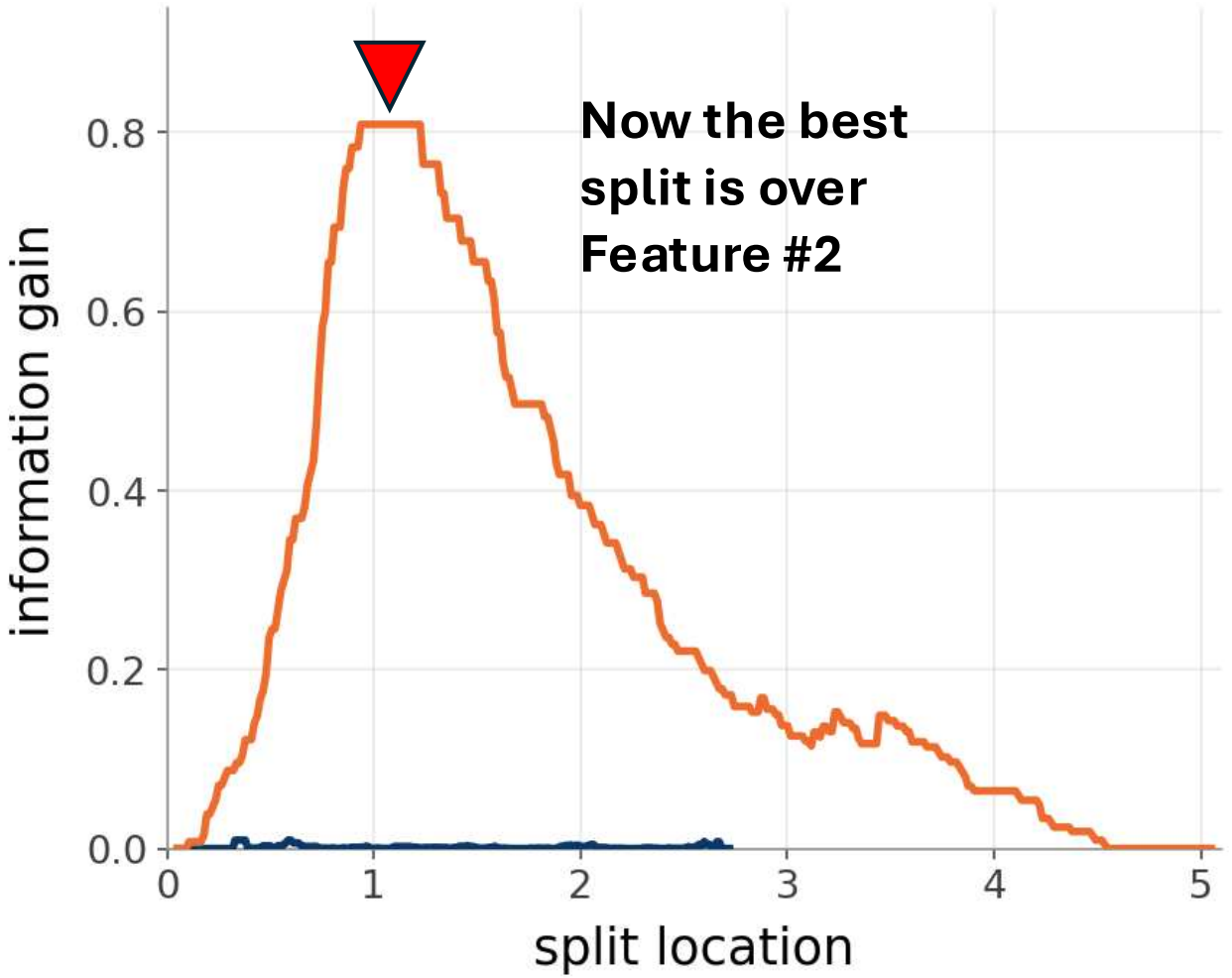
Maximize information gain *over the subspace*



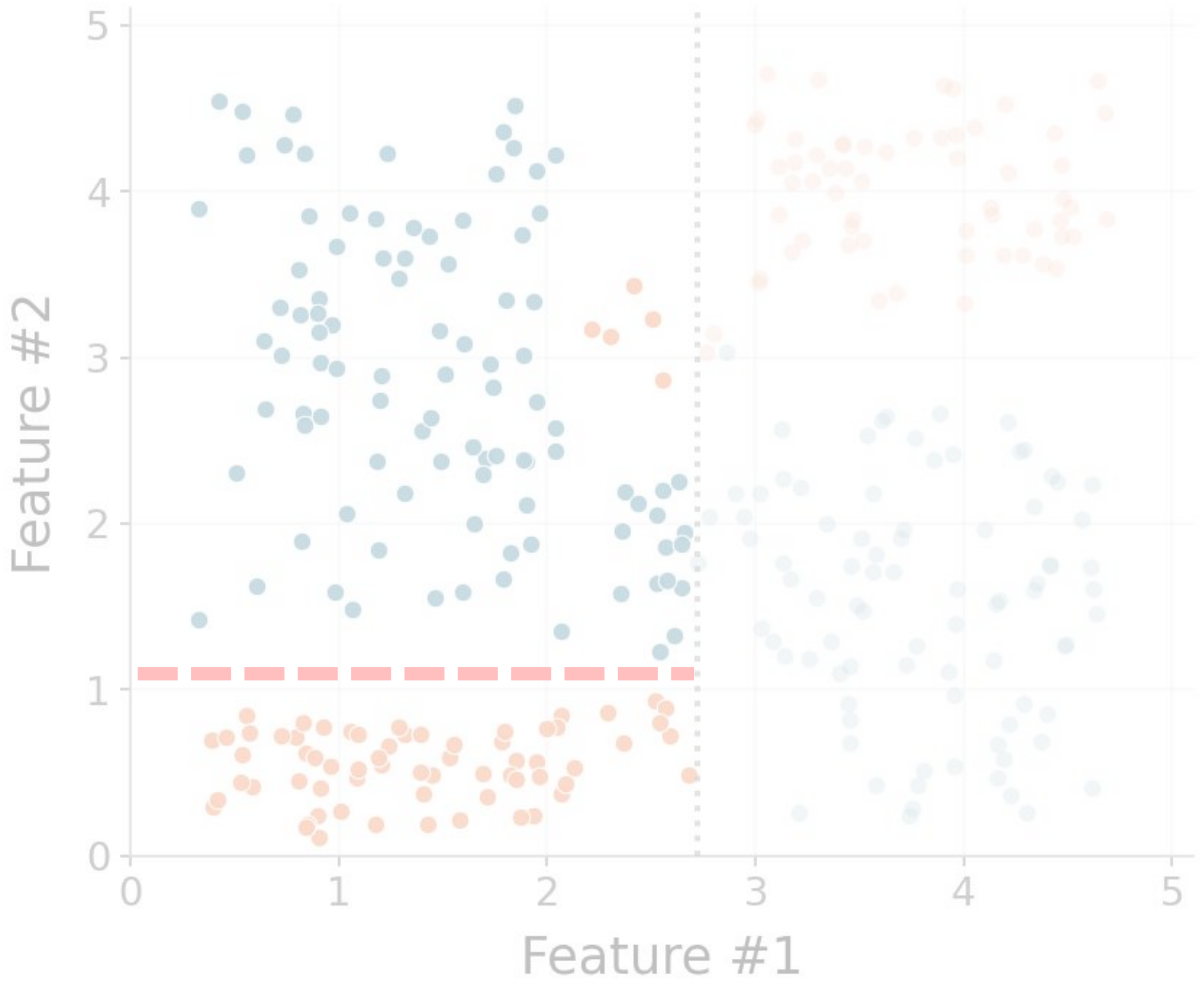


Maximize information gain *over the subspace*

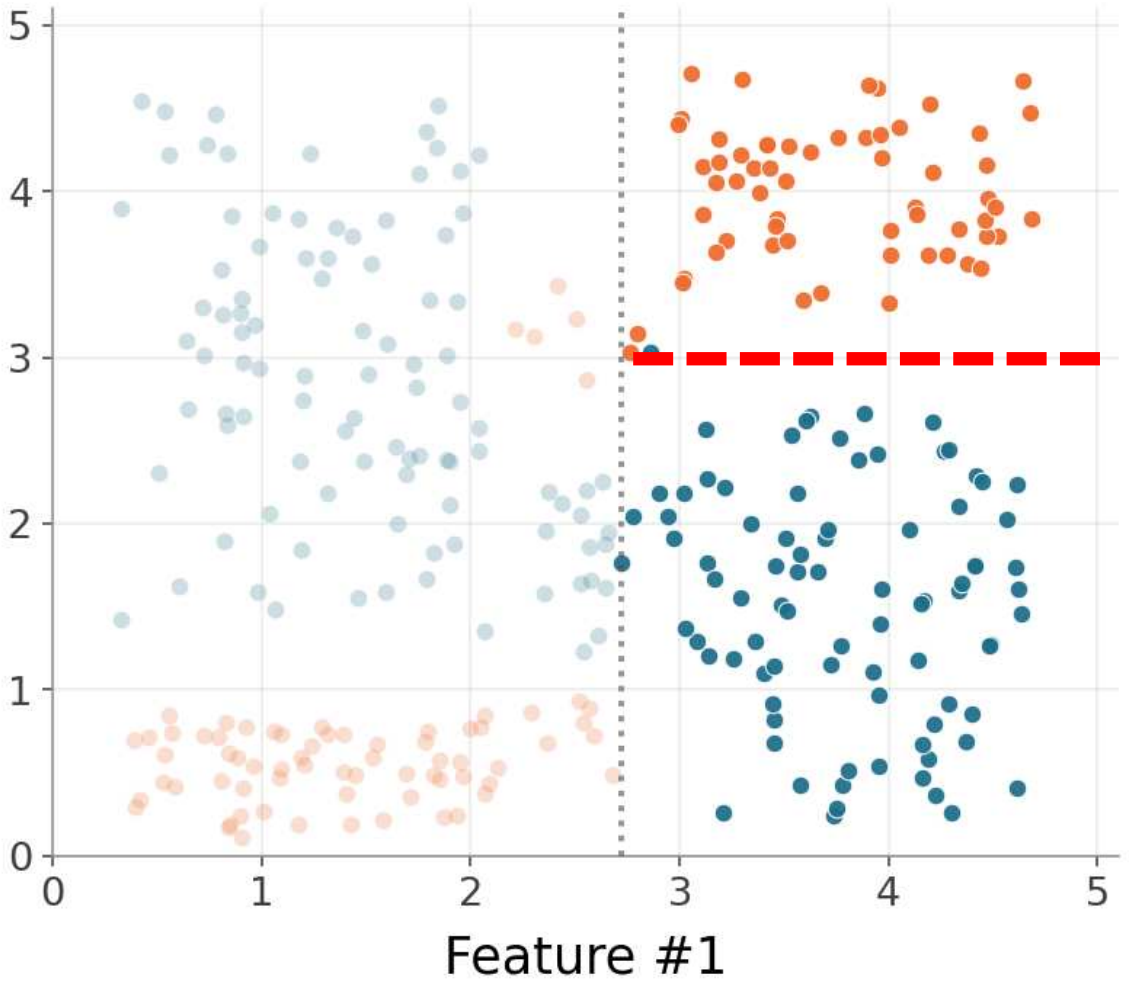




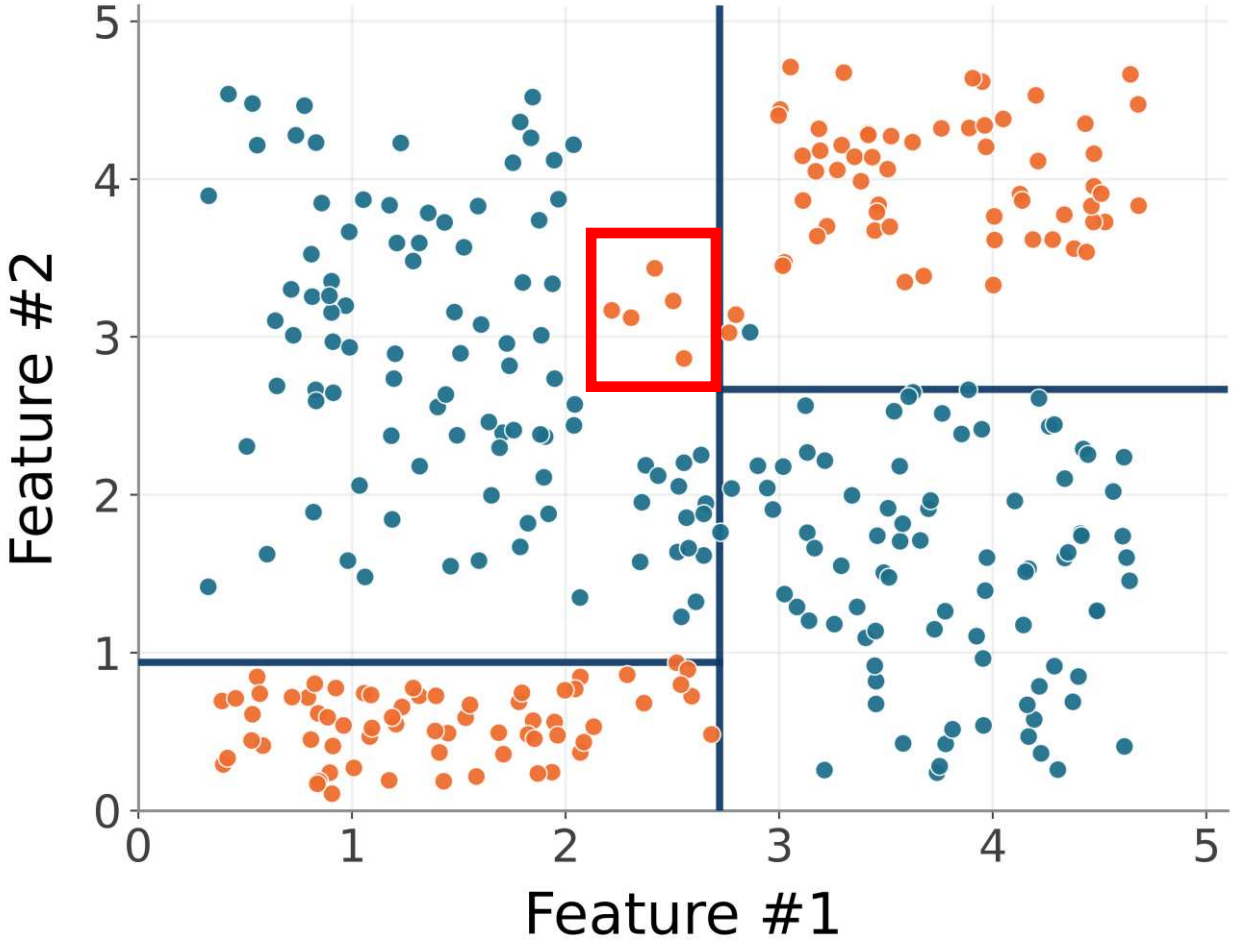
Do this for each node // subspace

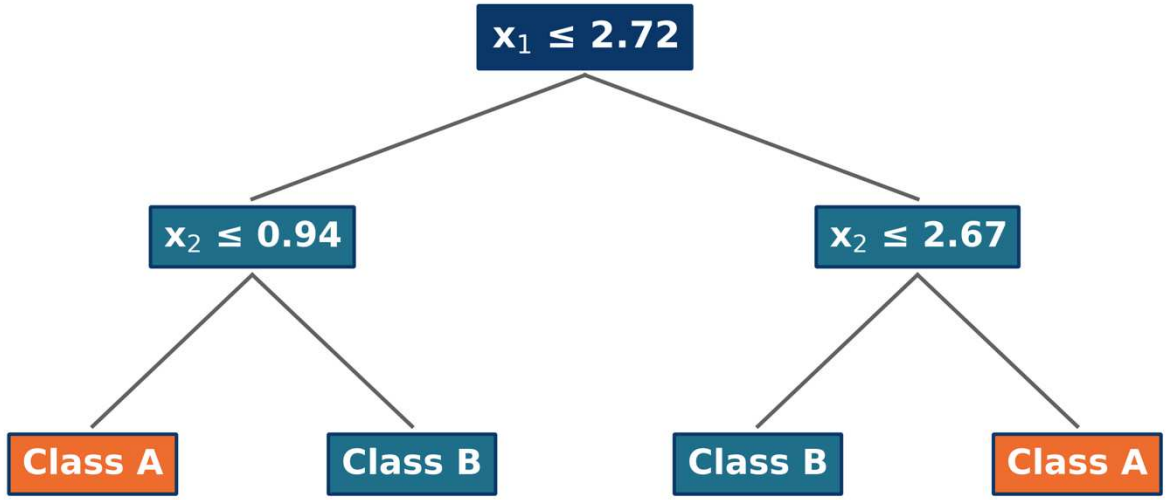


Repeat this for the other side



Knowing when to stop





Is this sufficient?

Or should we add more depth?

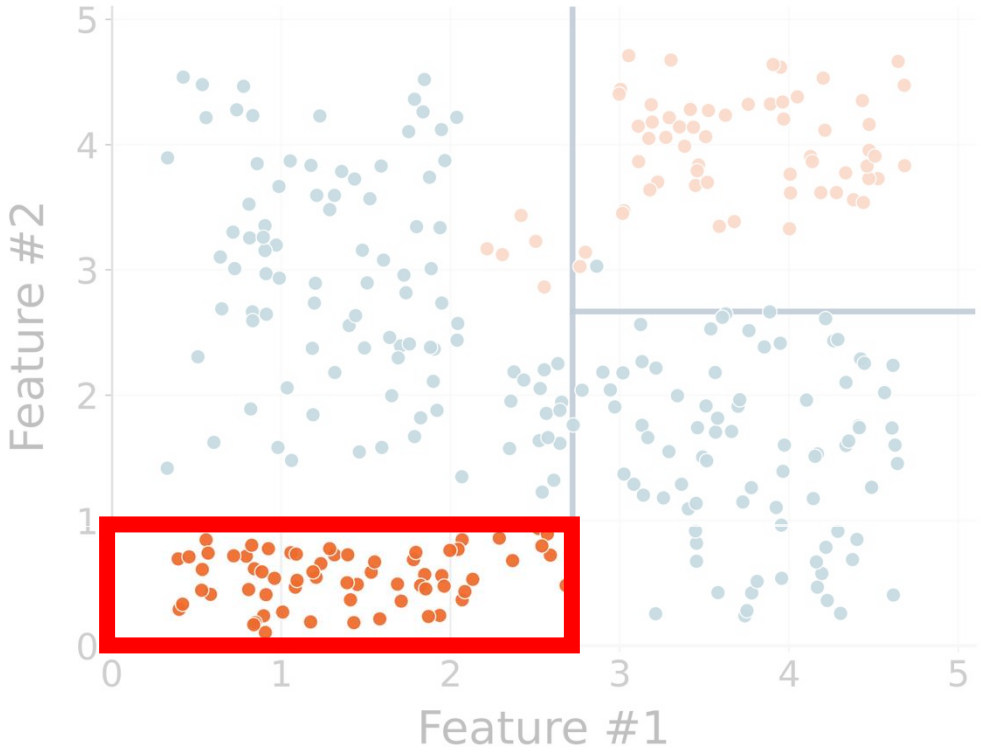
With more depth, we can capture more subtle groups…

But there's danger of overfitting

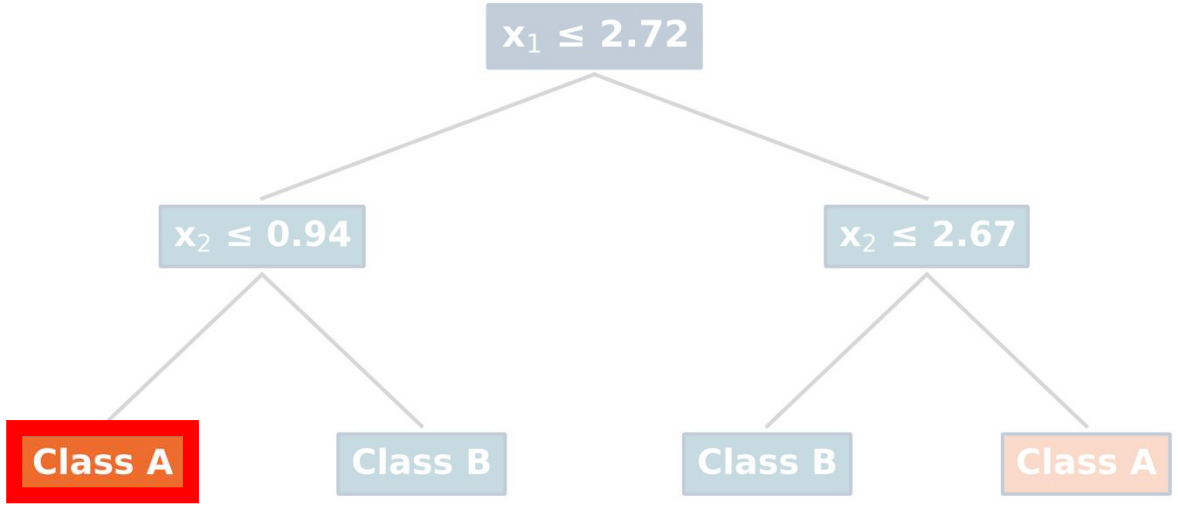
Common stopping criteria

Given a node containing N_{tot} samples, we might stop further splitting if:

1. The node is completely *pure* – all samples are of the same class



Gini impurity = 0



Common stopping criteria

Given a node containing N_{tot} samples, we might stop further splitting if:

1. The node is completely *pure* – all samples are of the same class

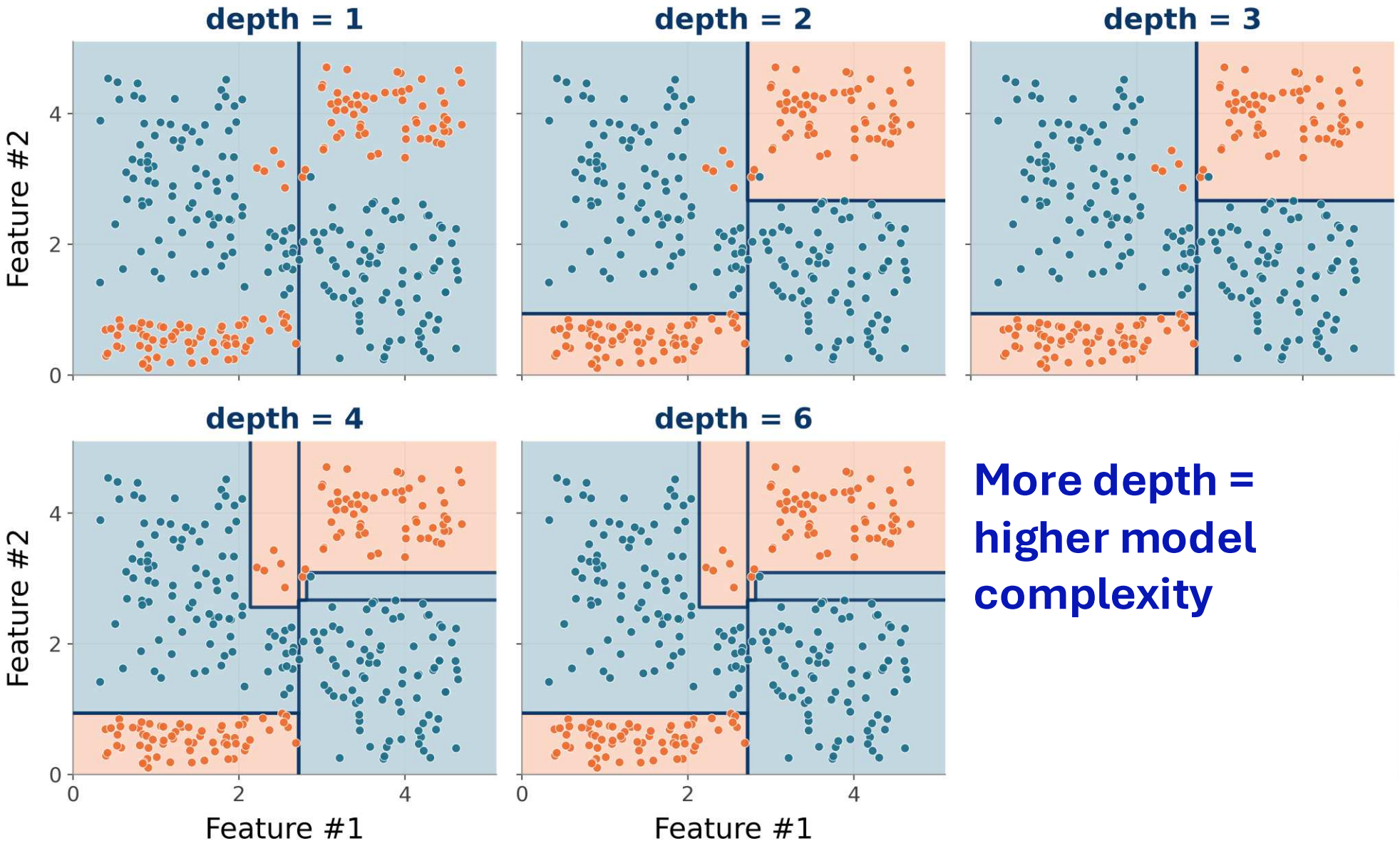
2. **The node has reached the allowed *maximum depth***

3. **The best available split gives *too little impurity reduction***

4. **A proposed child would contain *too few samples***

These are all **hyperparams** set by the user

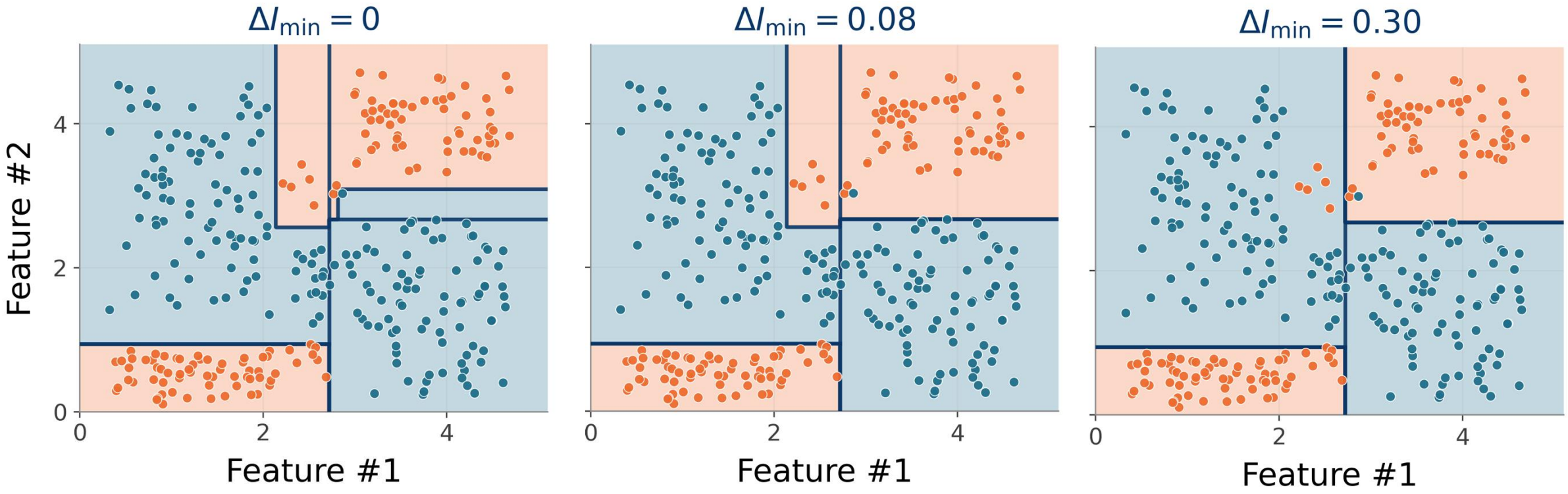
The influence of *maximum allowed depth*



More depth = higher model complexity

Greater allowed depth leads to more boundaries which can fit the few noisy samples

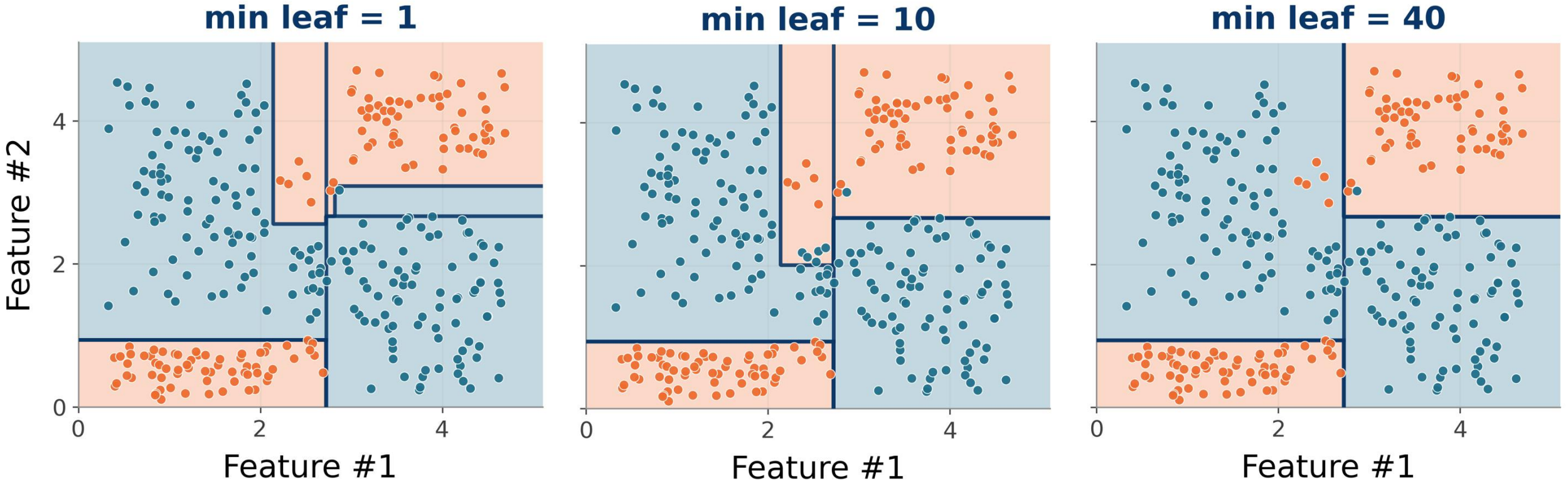
The influence of *minimum impurity reduction*



Similar effects are observed by allowing many splits,
even if those splits don't reduce impurity by much

Smaller allowed impurity reduction = more splits = higher model complexity

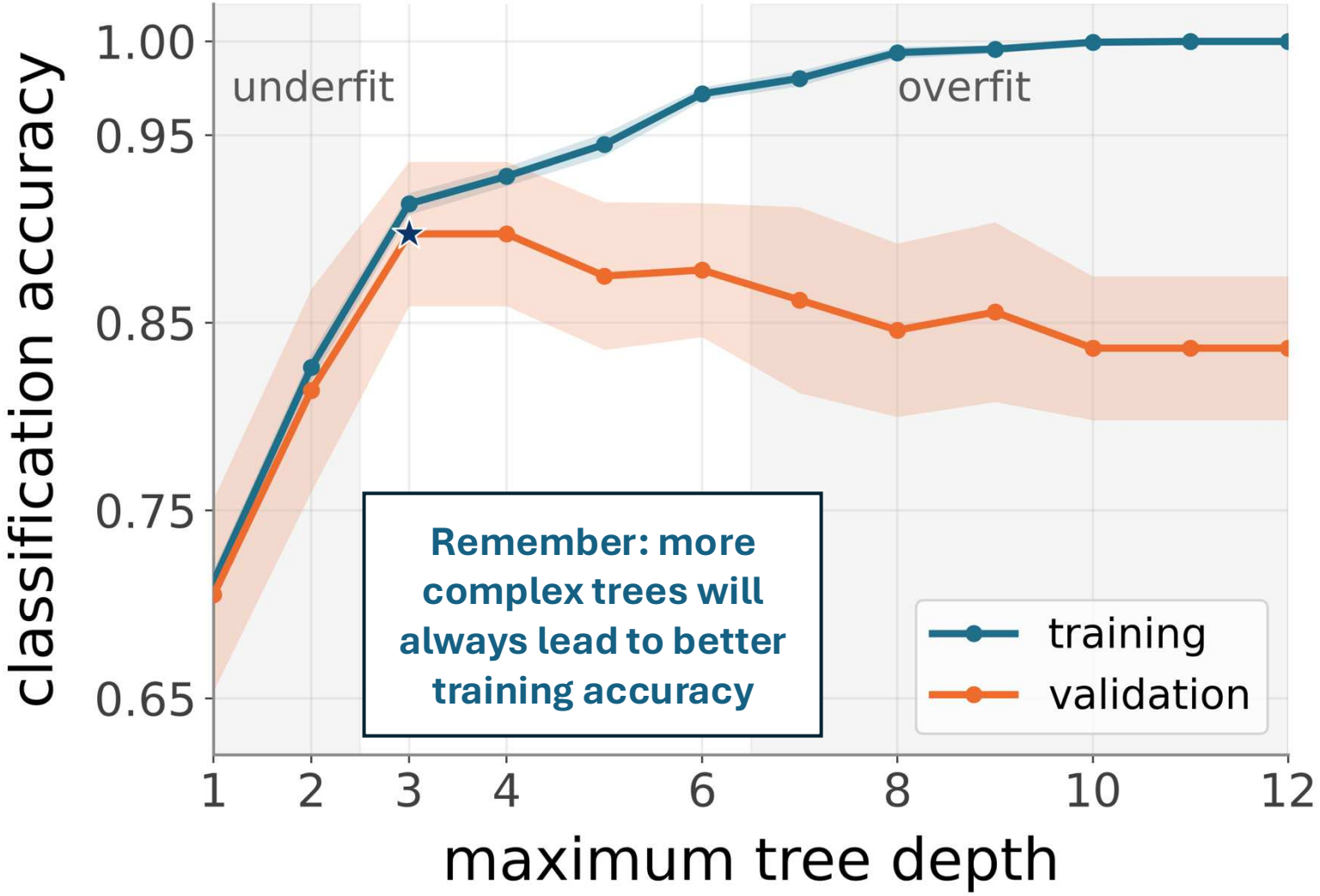
The influence of *minimum samples per leaf*



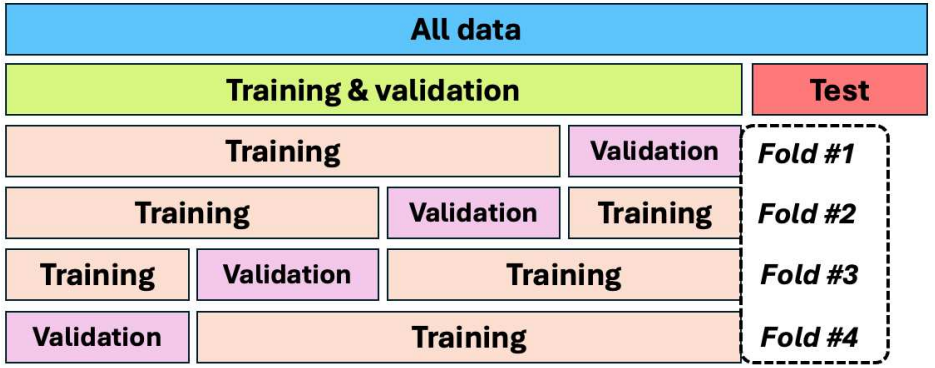
Similar effects are observed by allowing **fewer samples per leaf**

Fewer Allowed samples per leaf = more leaves = higher model complexity

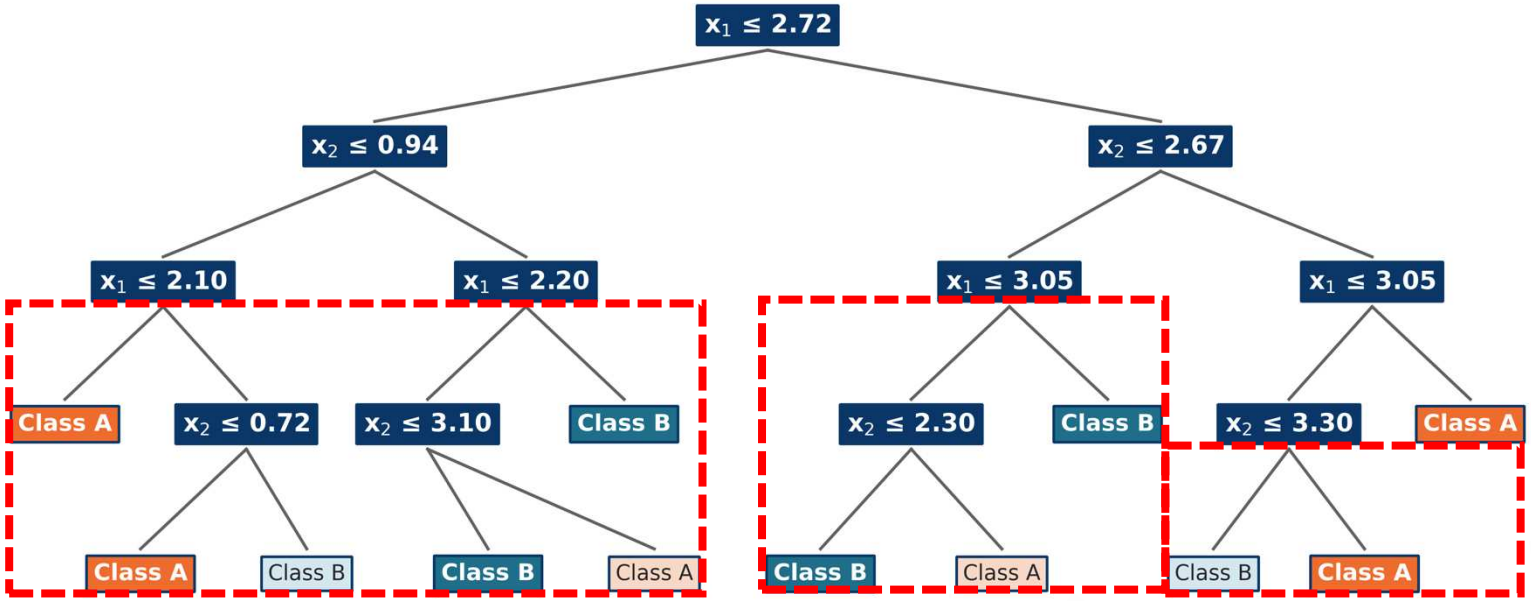
Under- *vs.* Over-fitting with decision trees



As with any hyperparameter, we use **cross-validation** to choose the right tree depth, minimum impurity reduction, and minimum number of samples per leaf



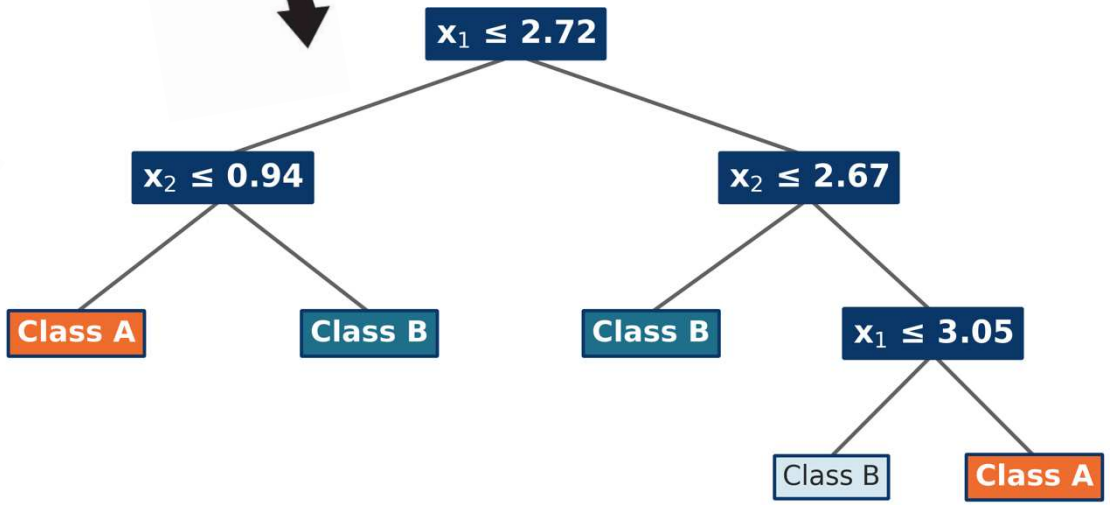
Pruning: Reduce complexity after the fact



What's nice about decision trees is they are very **interpretable**. We can decide ourselves which rules are reasonable.

Remove unnecessary branches. If done after the fact, as opposed to early stopping, this is called **post-pruning**





Post-pruning as a form of regularization

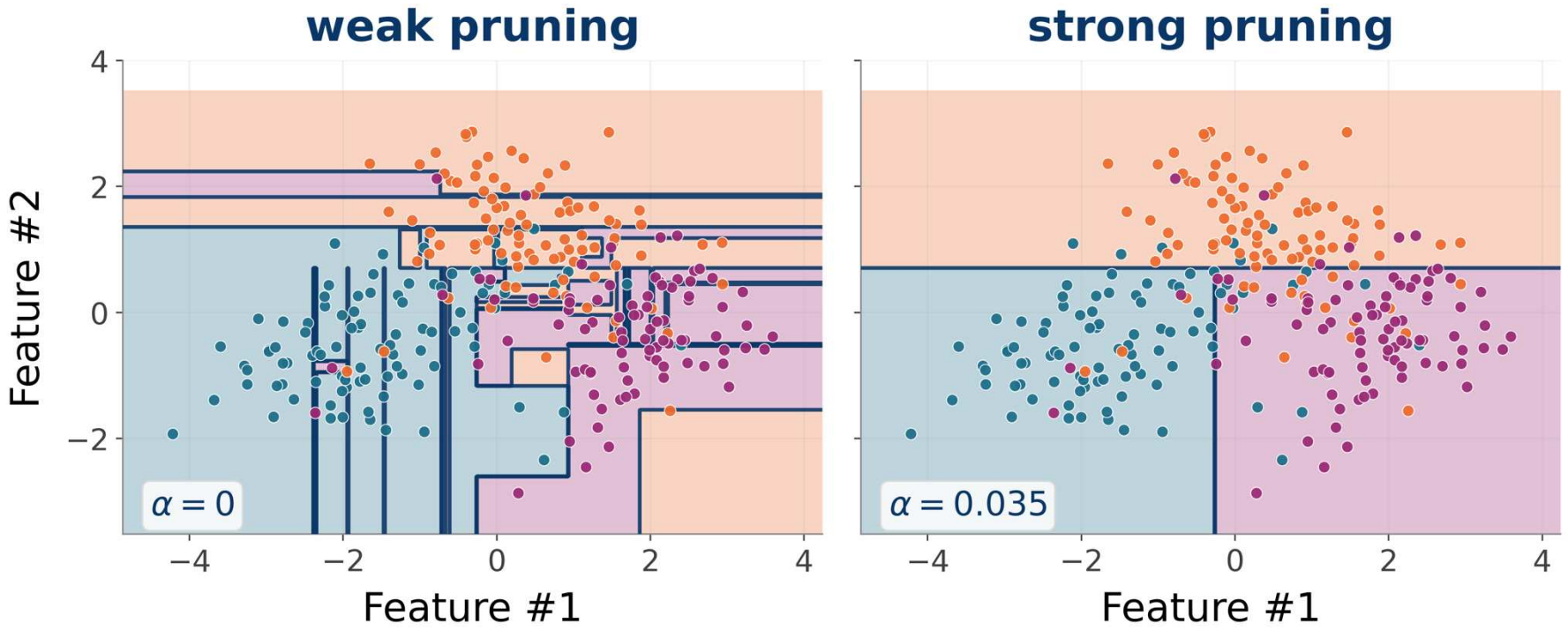
After building a large tree, prune it by minimizing:

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

$R(T)$: Average leaf impurity

$|T|$: The number of leaves

α controls the bias-variance tradeoff



Think of this as the decision-tree version of regularization!

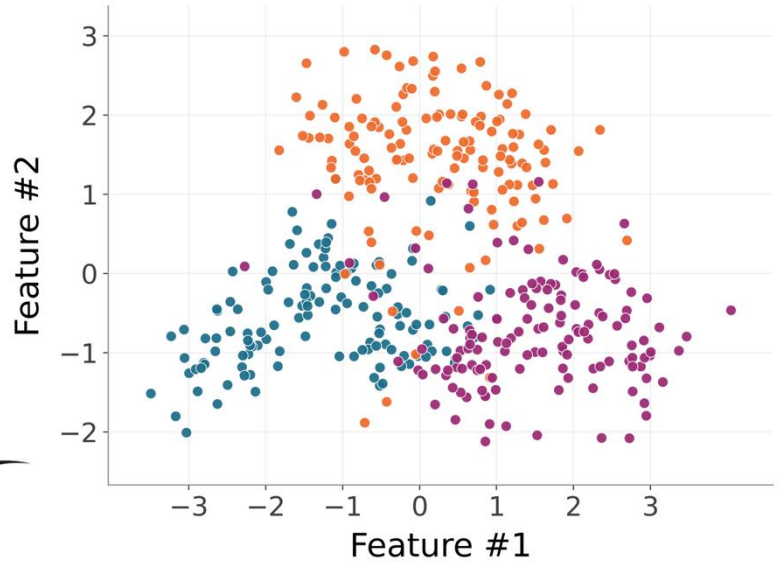
Larger α leads to simpler models

Questions?

Is one decision tree enough?

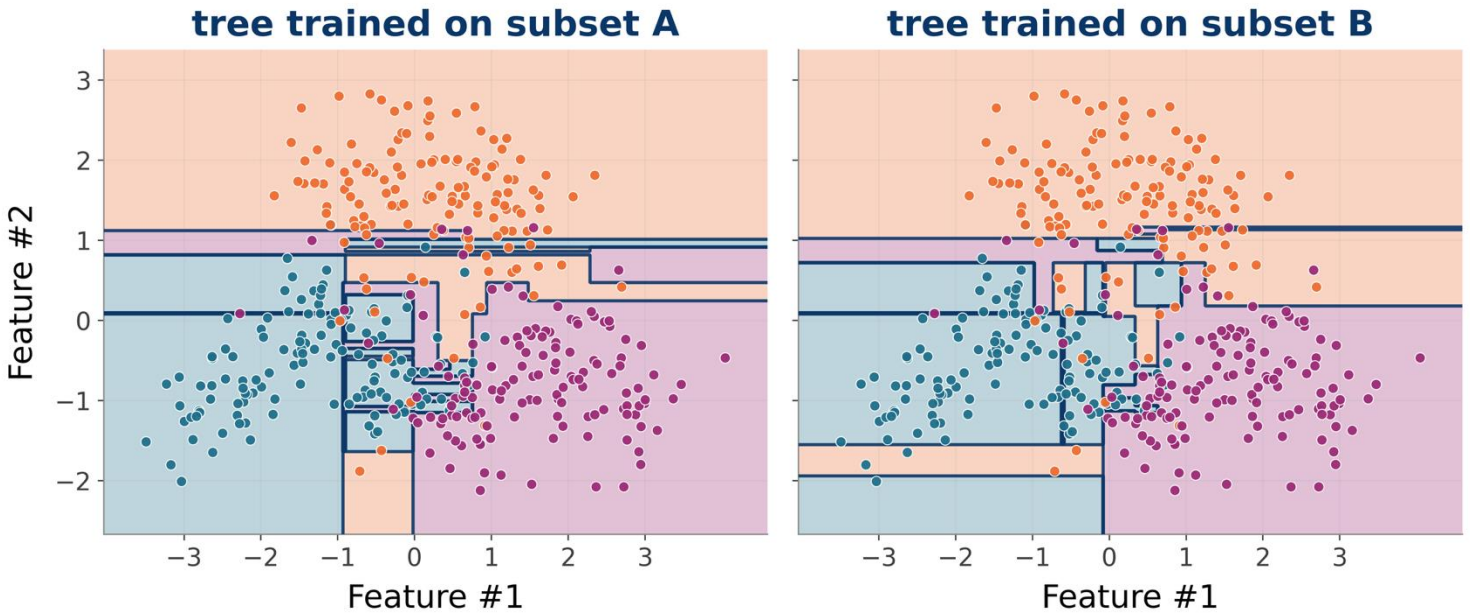
Dealing with high variance

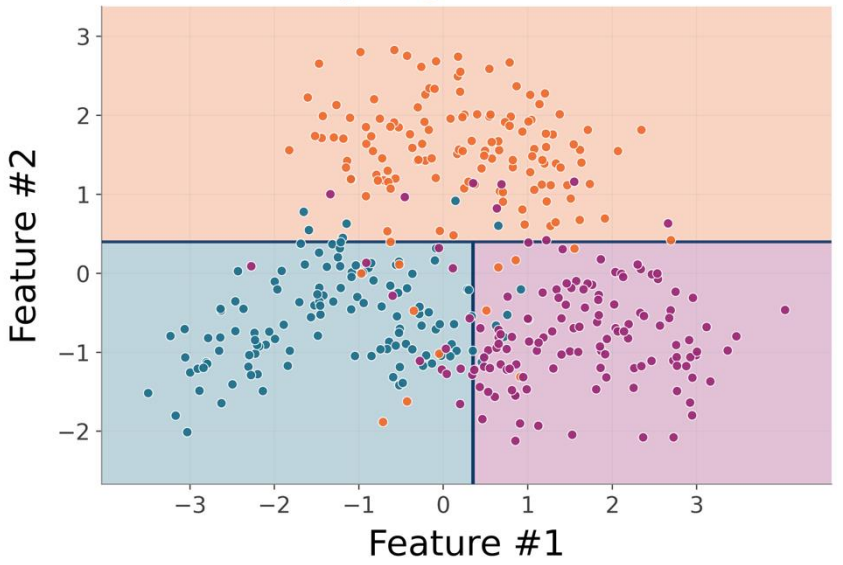
One tree is very sensitive to the training data



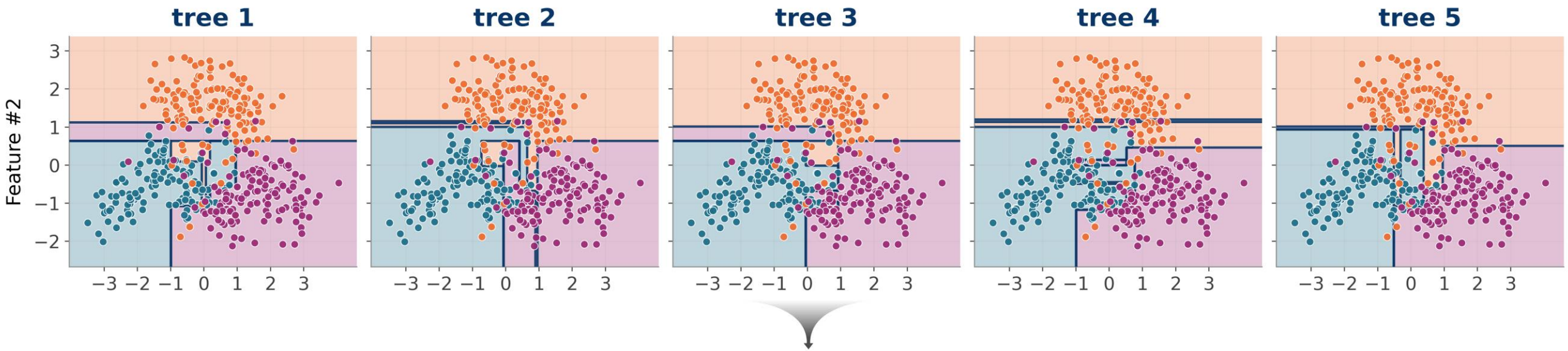
Decision trees can produce very different classification maps depending on which data we sample

Regularization helps to some extent, but then we can get overly simple models

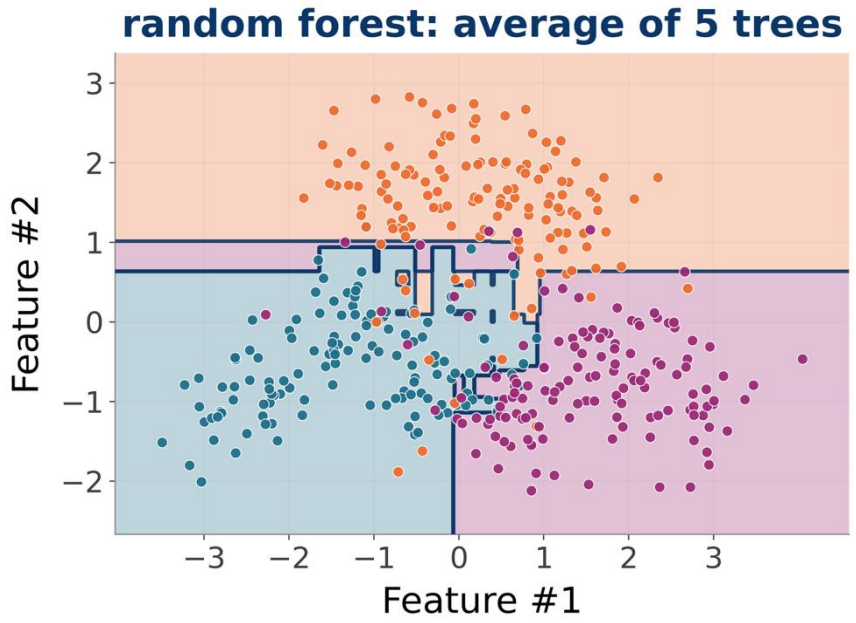




Random forests: Average over many trees!



Each tree is trained on different parts of the dataset, and it provides one vote. Whichever category gets the most votes is the final prediction



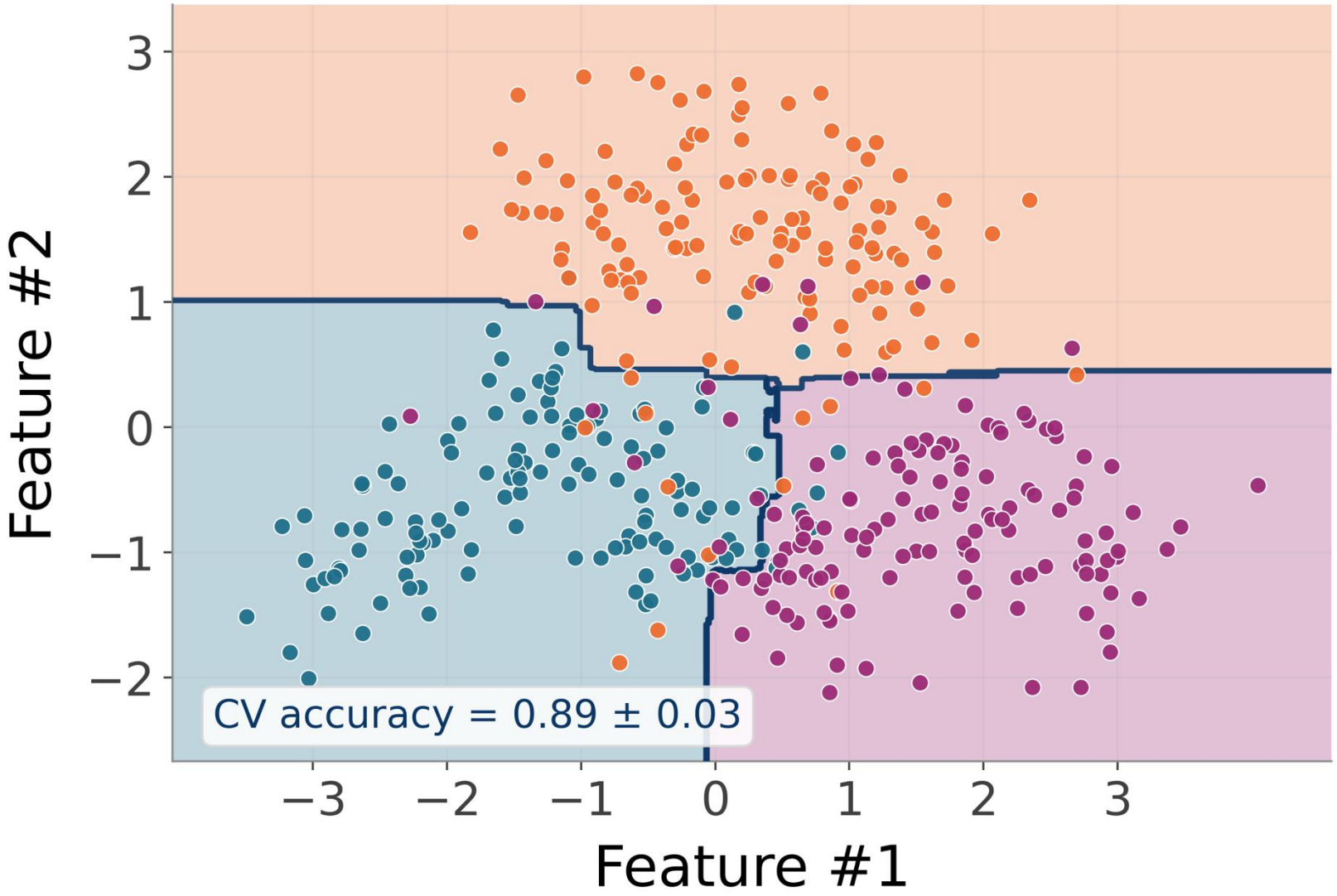
With just 5 trees, his still looks quite noisy…

What if we average over more trees?

Random forests: Average over many trees!



Random forests: Average over many trees!

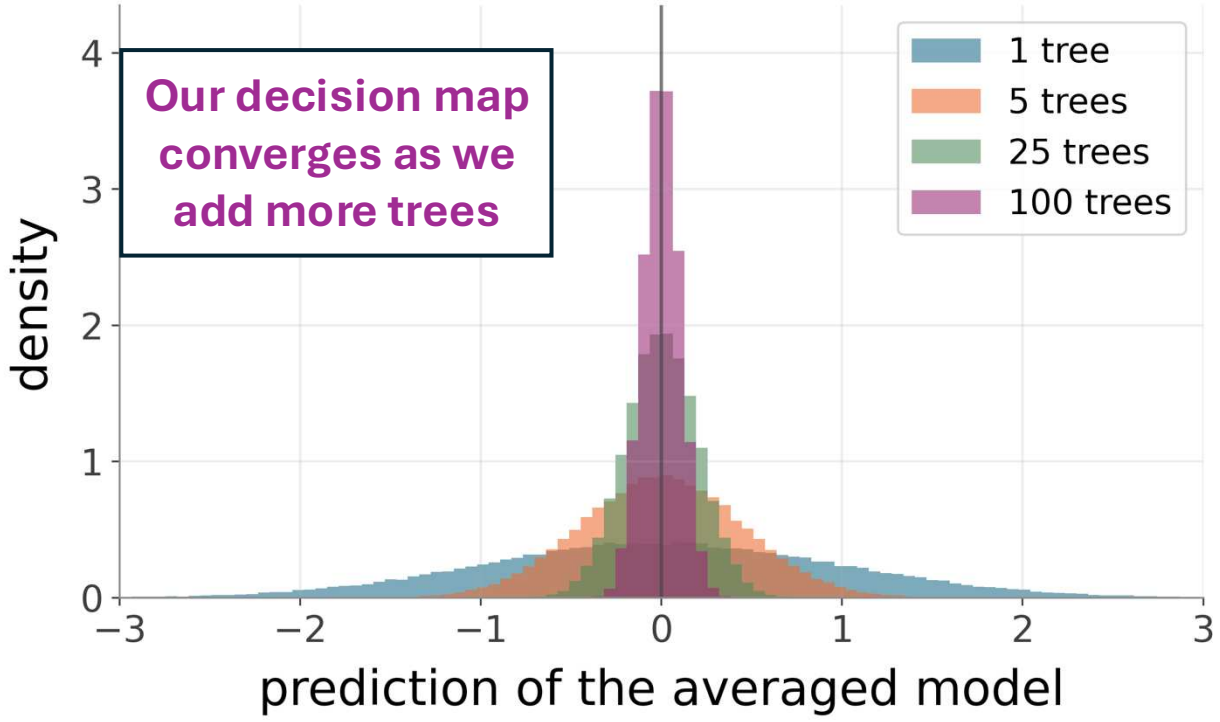


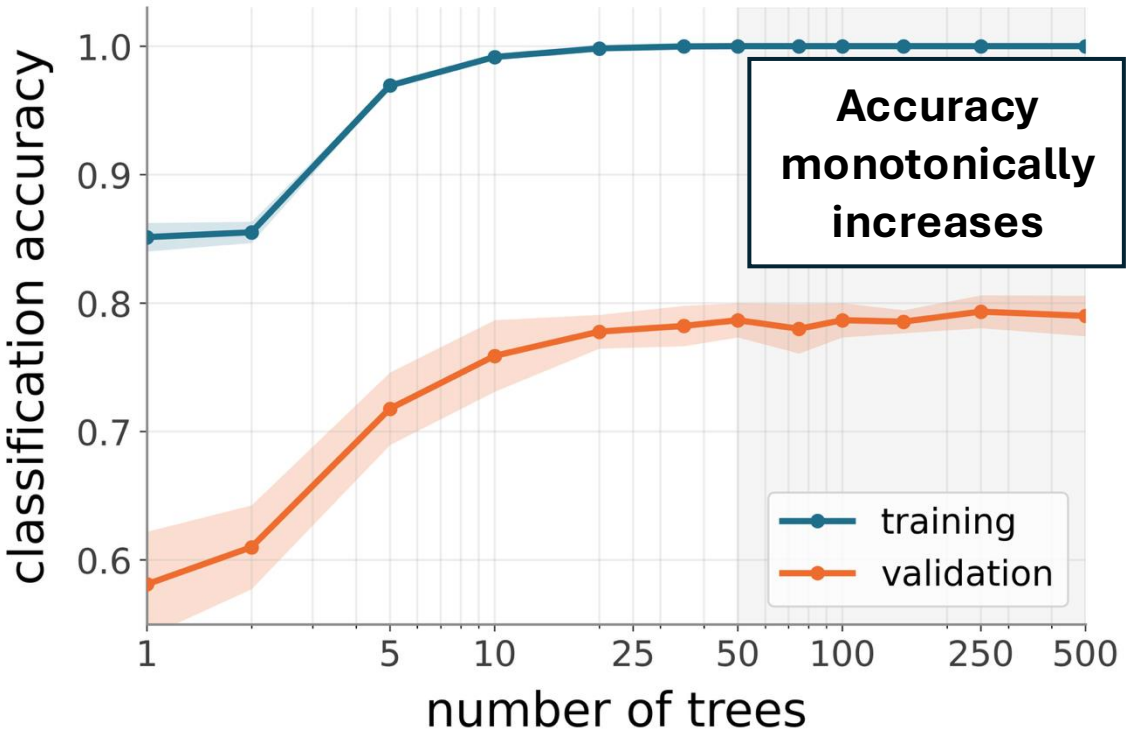
We get much more reasonable, semi-smooth decision boundaries by averaging over **1000 trees**

And now regularization becomes less of an issue. **Even if each individual model is overfit, the average is not!**

→ ***Ensemble*** **benefits**

Adding more trees → reduced variance



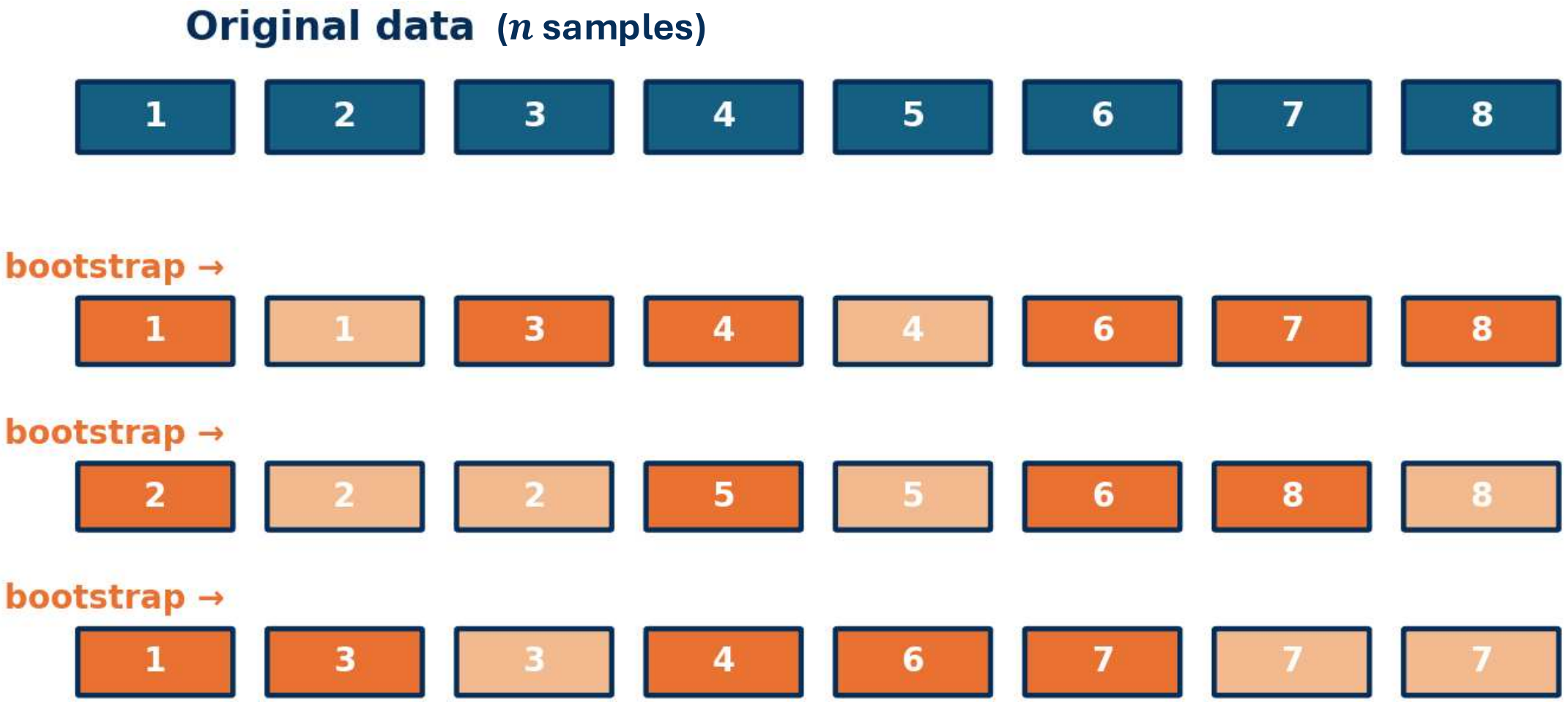


$$\text{Variance}(\text{Avg Prediction}) \propto \frac{1}{\text{Number of Models}}$$

Adding more trees generally improves performance but eventually plateaus (here we might stop at ~50 trees)

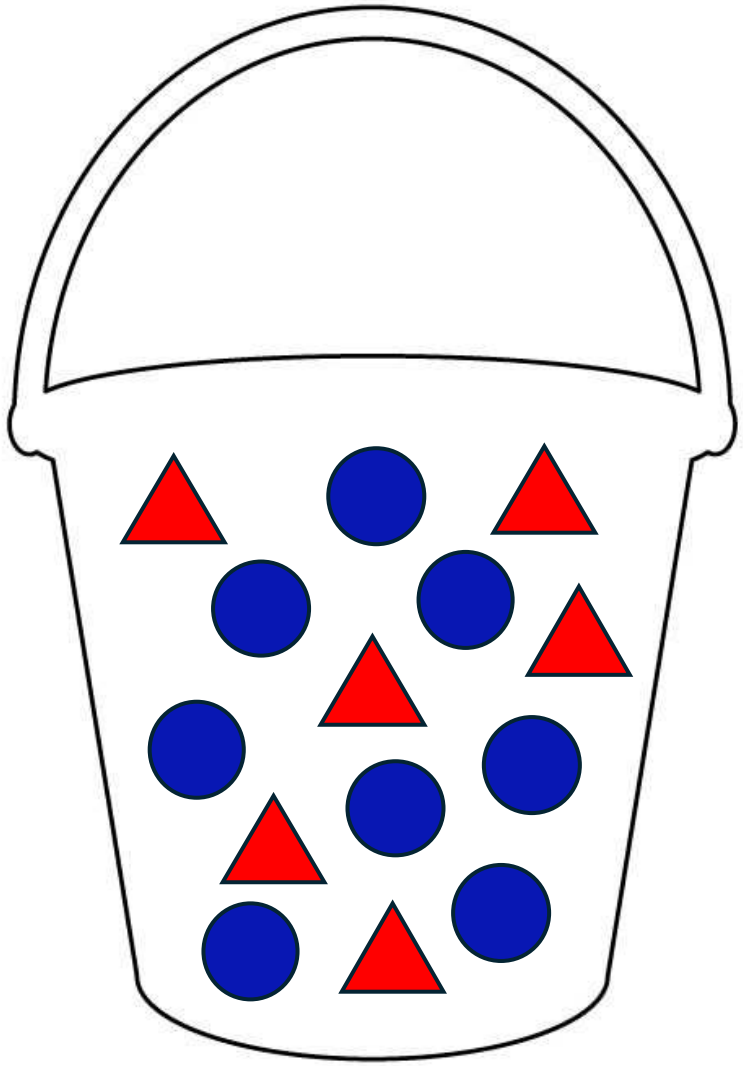
This assumes each model is ***independent***, differing in some way from the others (have many instances of the same model would not help)

How to make *different* models: Bootstrapping



Sample n points **with replacement** → some repeat, some are left out

How to make *different* models: Bootstrapping



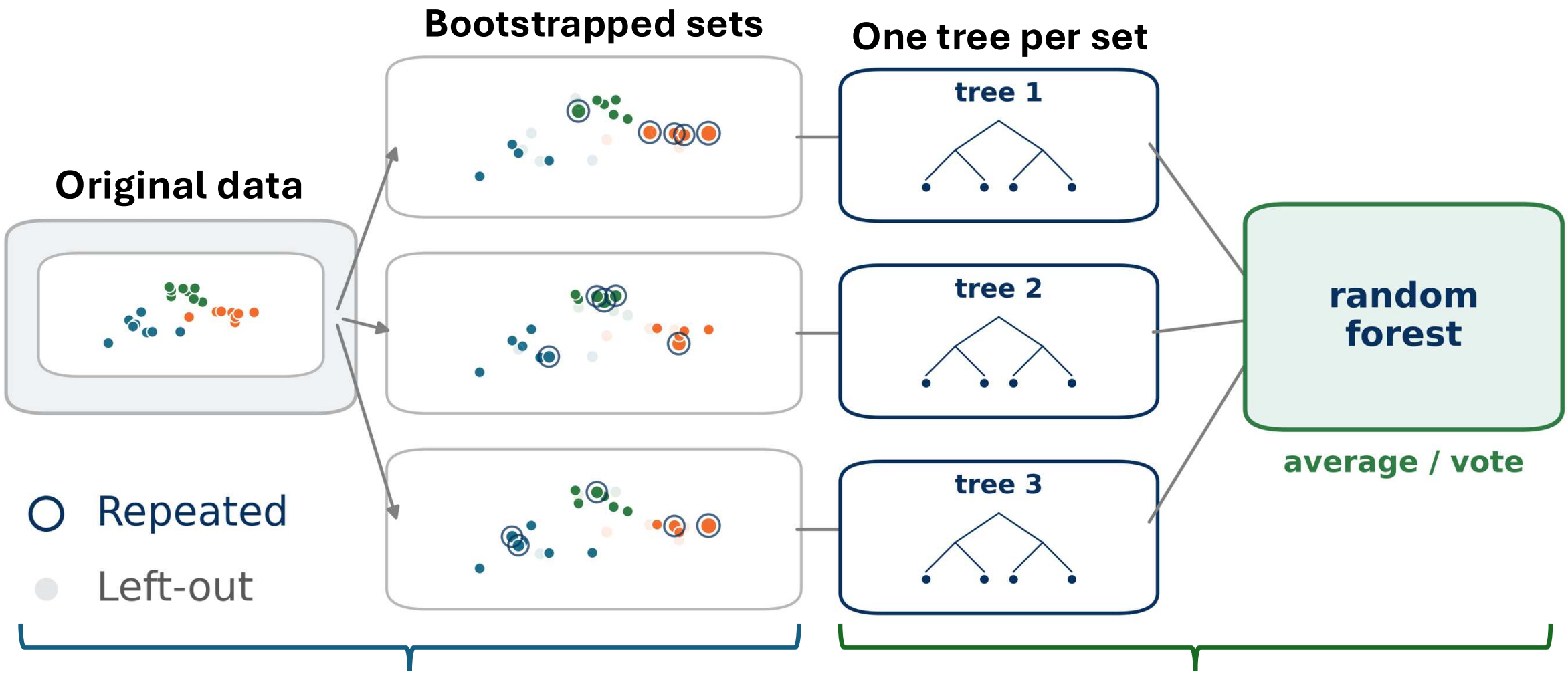
Recall:

I randomly choose a sample from this bucket

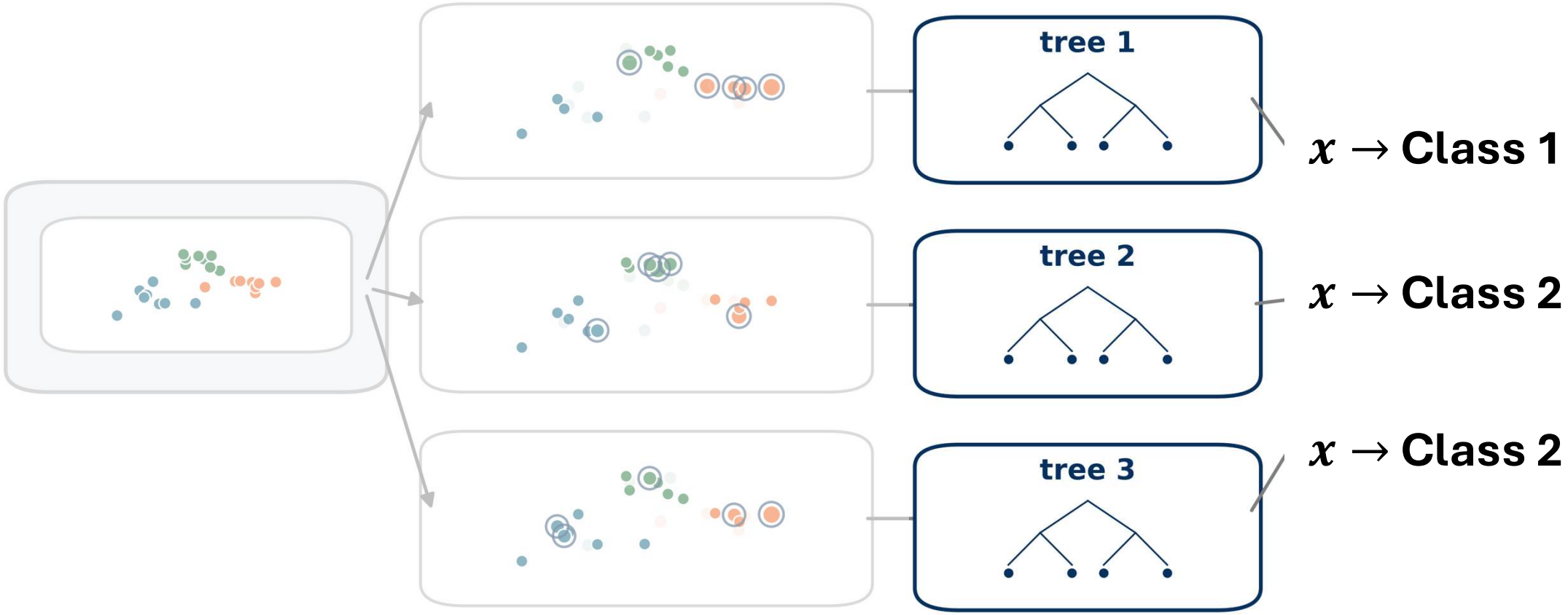
Then I put it back

And randomly choose again…

Bootstrapping and Bagging



Voting and class probability



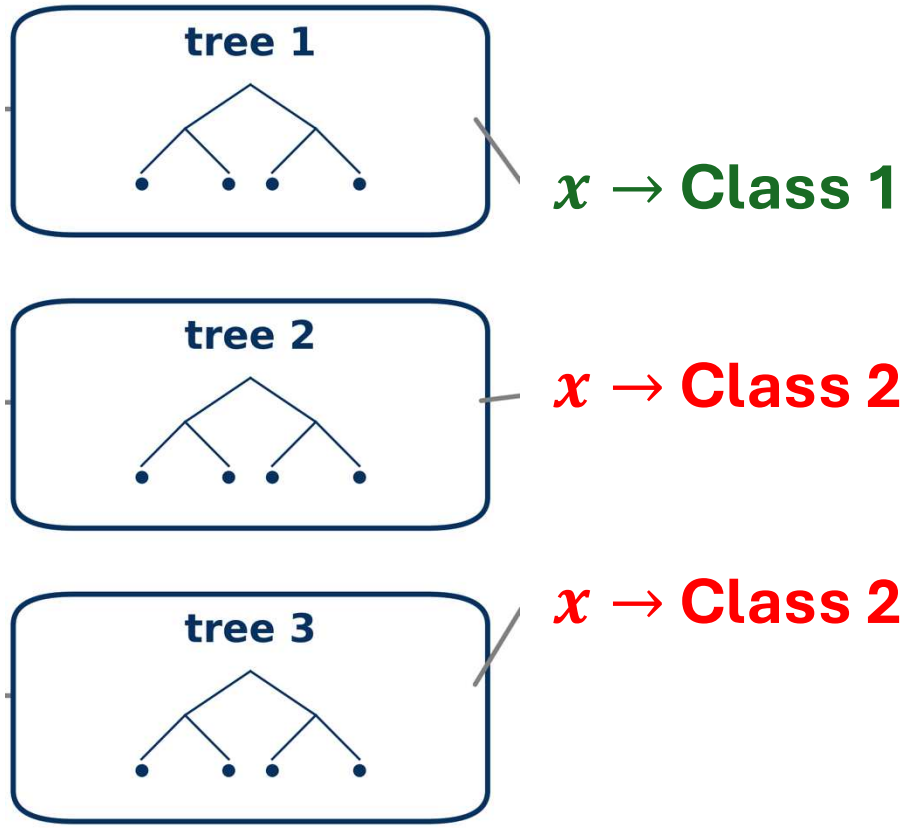
Given some new data (x), each tree might give a different result

Voting and class probability

We can formulate a **probability** based on the classification of the trees:

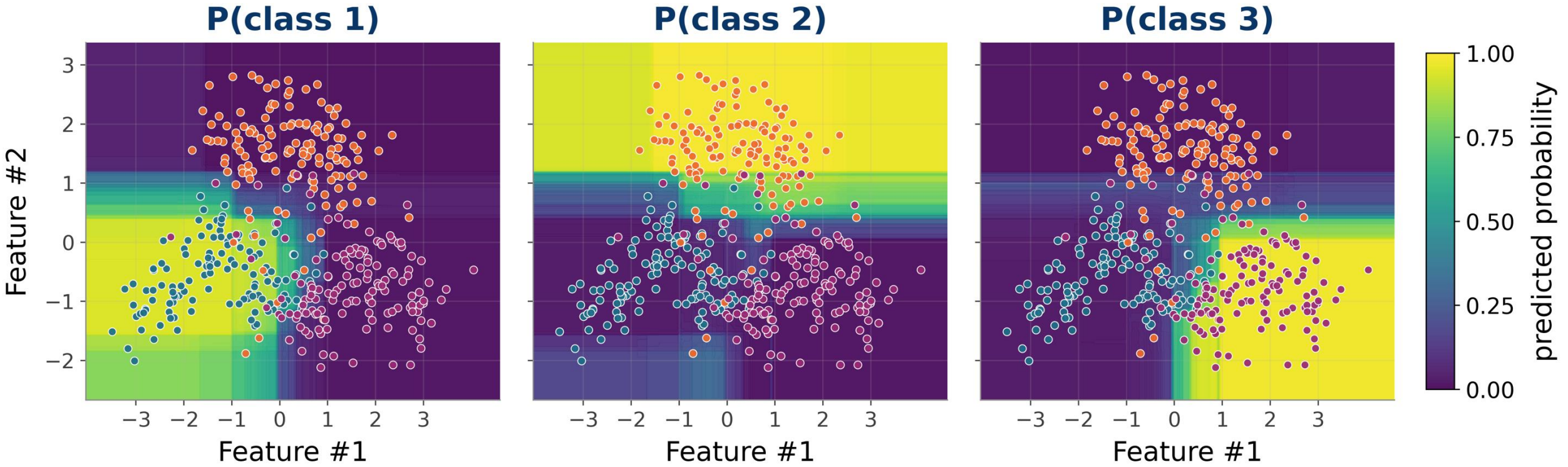
$$P(y = k|x) = \frac{\text{Number of votes for class } k}{\text{Total number of trees}}$$

So in this case, we would classify the sample as class 2 with a probability of about 66%



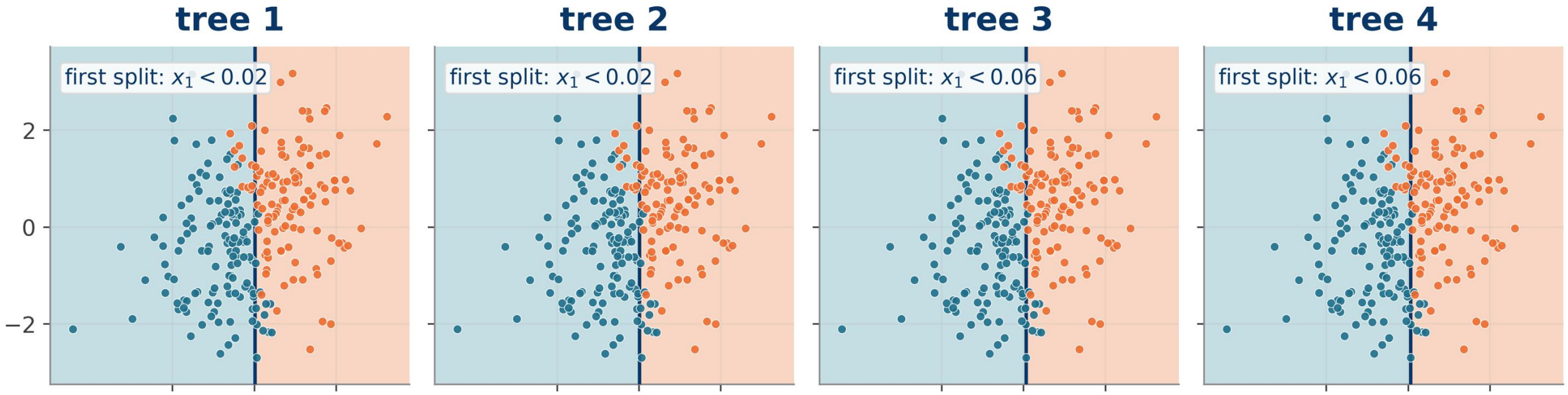
Given some new data (x), each tree might give a different result

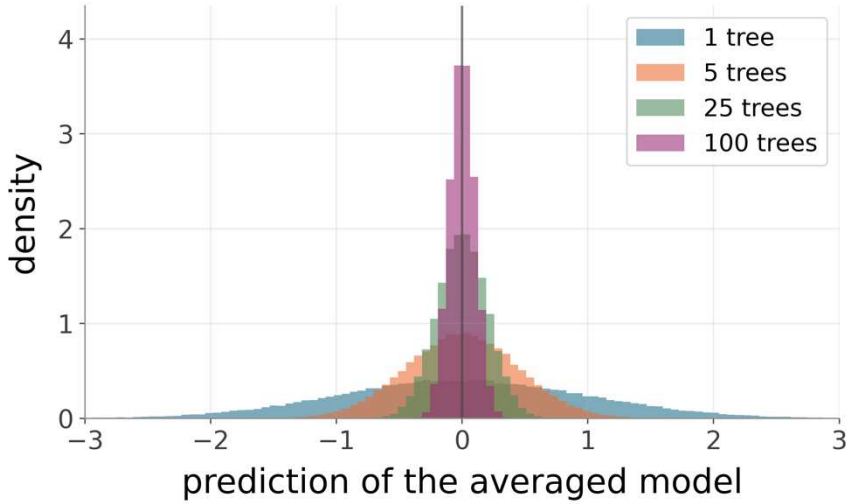
Probability maps from random forests



Similar to KNN – both use “votes” – but random-forest decision boundaries are often more aligned with each axis (since that’s how decisions are made)

One problem with bootstrapping and bagging





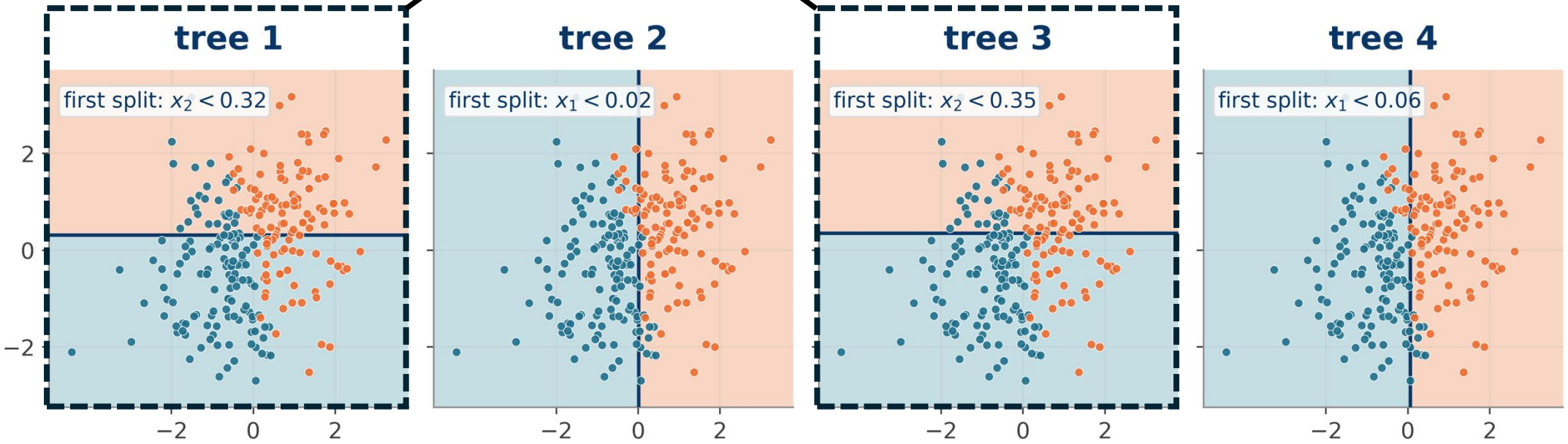
On this dataset, no matter which samples we include in our bootstrap, **the first decision will basically always be the same (a split along feature #1)**

Remember, we want our models to be as different as possible from one another!

One problem with bootstrapping and bagging

In these cases, Feature #1 was excluded, forcing the models to use Feature #2 for their split

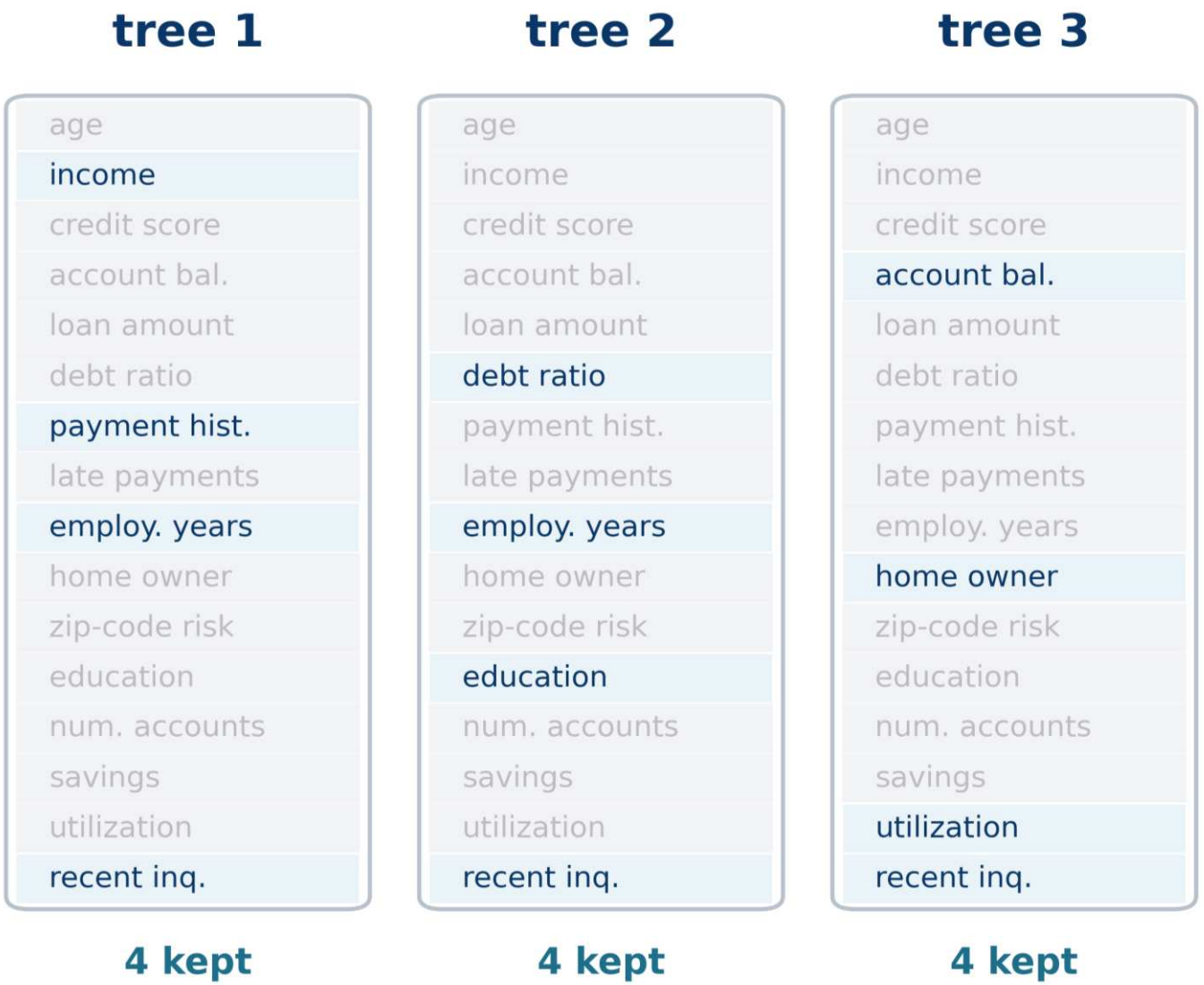
What we can do is **hold out a random subset of features** when performing each split



This is more useful with many features

tree 1	tree 2	tree 3	tree 4	tree 5
age	age	age	age	age
income	income	income	income	income
credit score	credit score	credit score	**credit score**	credit score
account bal.	account bal.	**account bal.**	account bal.	account bal.
loan amount	loan amount	loan amount	loan amount	**loan amount**
debt ratio	**debt ratio**	debt ratio	debt ratio	**debt ratio**
payment hist.	payment hist.	payment hist.	payment hist.	payment hist.
late payments	late payments	late payments	**late payments**	late payments
employ. years	**employ. years**	employ. years	employ. years	employ. years
home owner	home owner	**home owner**	**home owner**	home owner
zip-code risk	zip-code risk	zip-code risk	zip-code risk	zip-code risk
education	**education**	education	education	**education**
num. accounts	num. accounts	num. accounts	num. accounts	num. accounts
savings	savings	savings	savings	**savings**
utilization	utilization	**utilization**	**utilization**	utilization
recent inq.	**recent inq.**	**recent inq.**	recent inq.	recent inq.
4 kept	**4 kept**	**4 kept**	**4 kept**	**4 kept**

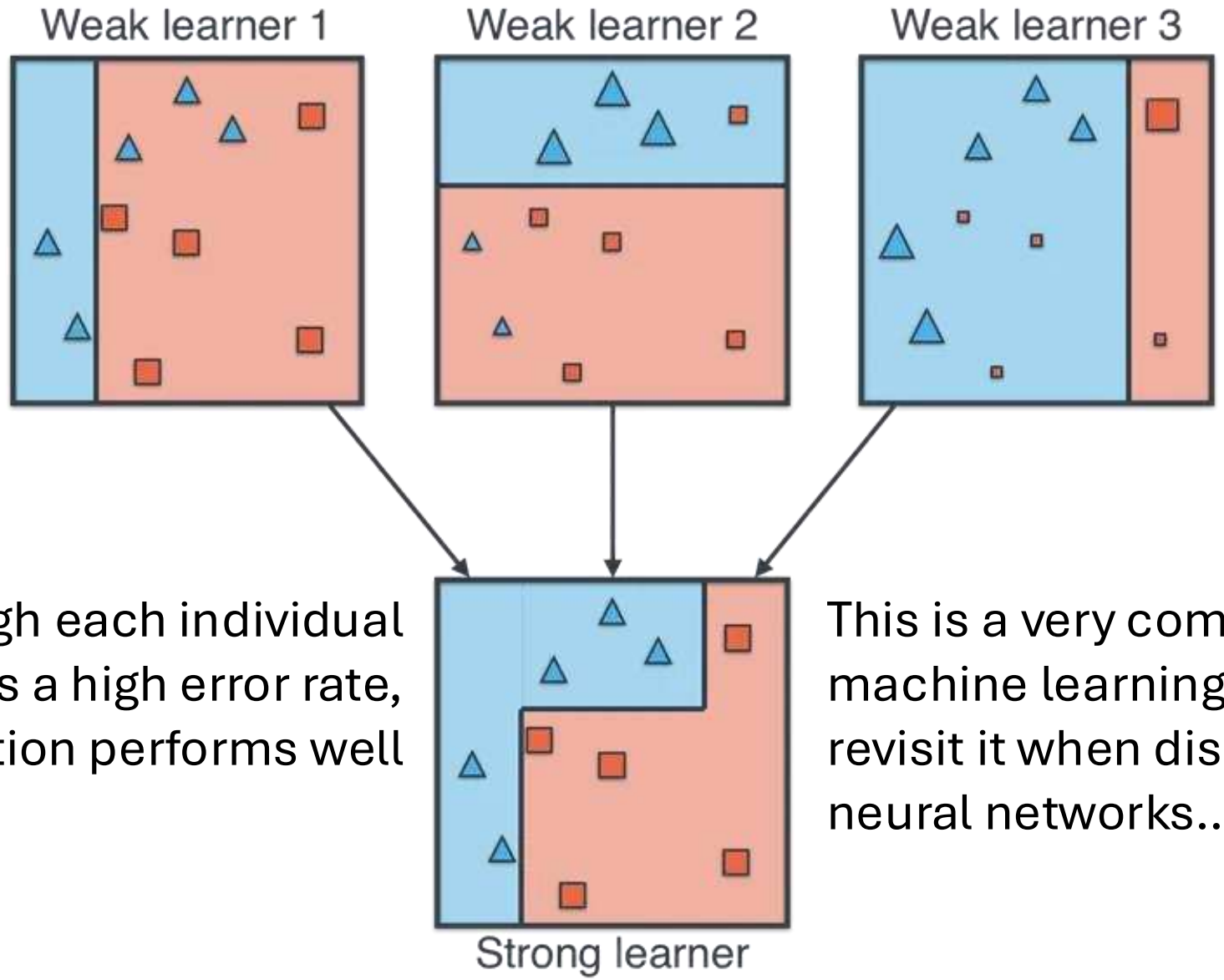
This is more useful with many features



It is typical to keep about $\sqrt{p}$ **features** when building each tree, where p is the total number of features in the data set

Hmmm….Aren't we losing a lot of information?

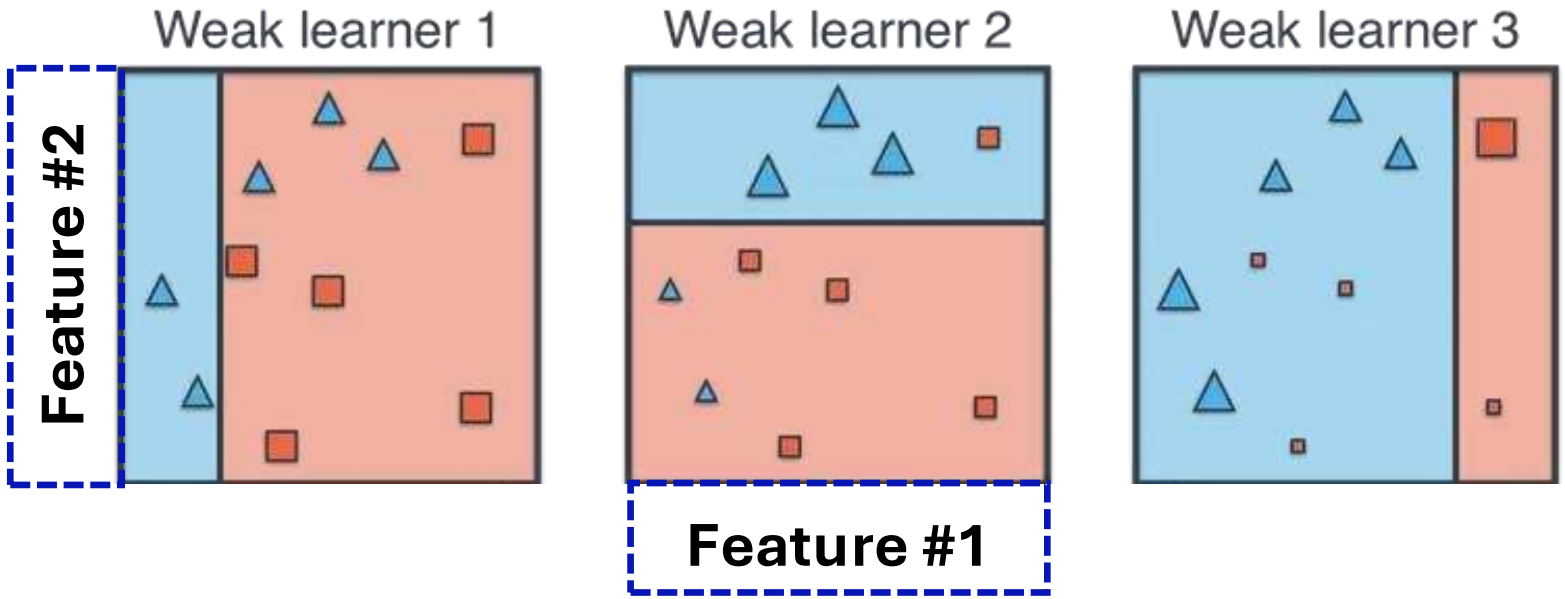
Many weak learners → one strong learner



Even though each individual learner has a high error rate, their aggregation performs well

This is a very common idea in machine learning! We will revisit it when discussing neural networks…

Which features matter the most?



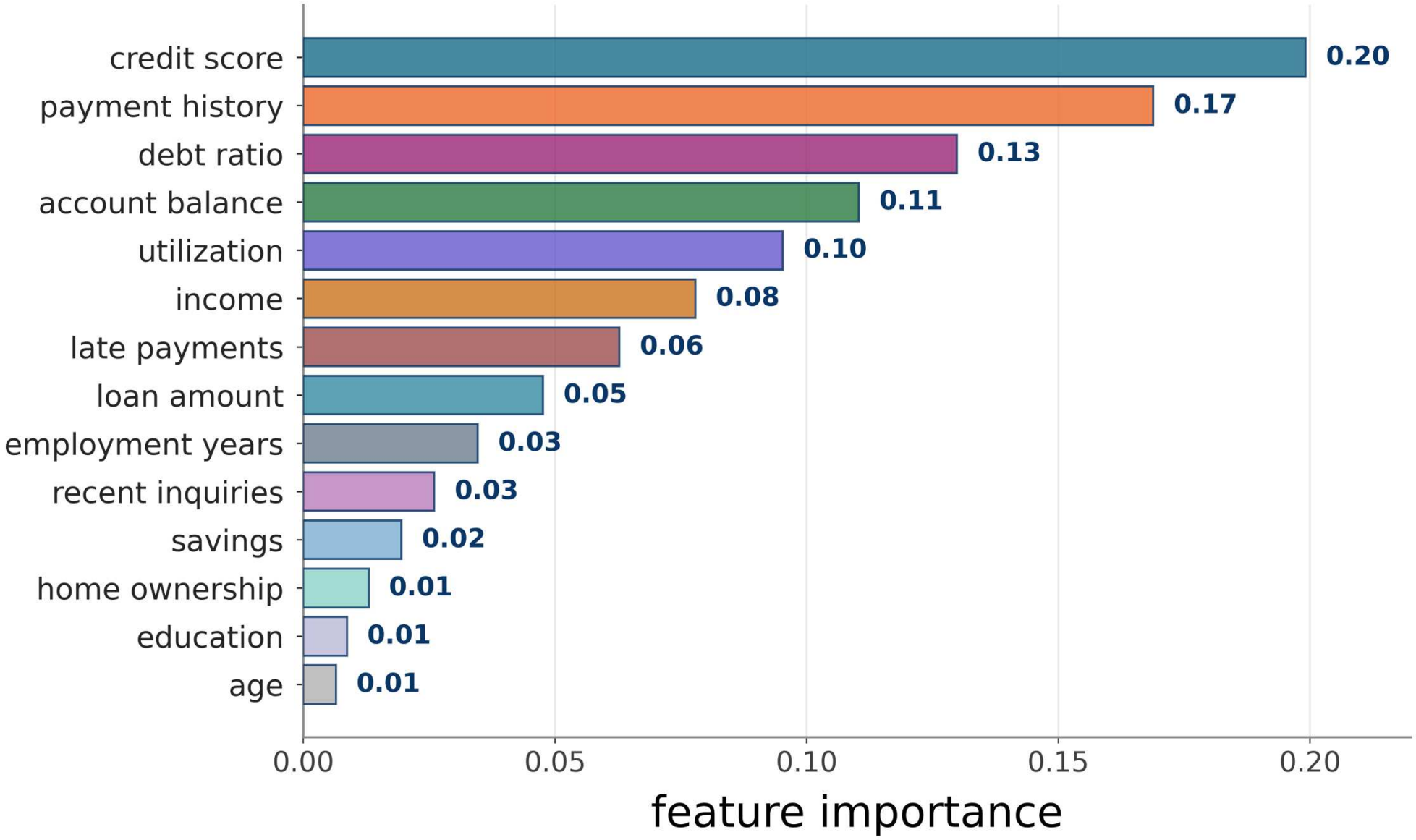
$$\text{Importance}(x_i) = \sum_{\text{trees}} \sum_{\text{nodes}} \frac{n_{\text{node}}}{N_{\text{tree}}} \text{IG}_{\text{node}}$$

Sum over **all nodes** in **all trees** which use that feature to make a decision

Remember: Information gain (IG) tells us how much a given split reduces impurity

→ **Important features lead to clear separation of classes**

Plotting feature importance



High importance → the feature matters more

Typically, we **normalize these values**, so they sum to one across all features

Questions?

Boosting

Can we build trees **non-randomly?** To fix each other's errors?

Bagging *vs.* Boosting

Bagging / Random Forest — parallel & independent



Equal votes

Boosting — sequential, each fixes the last one's mistakes

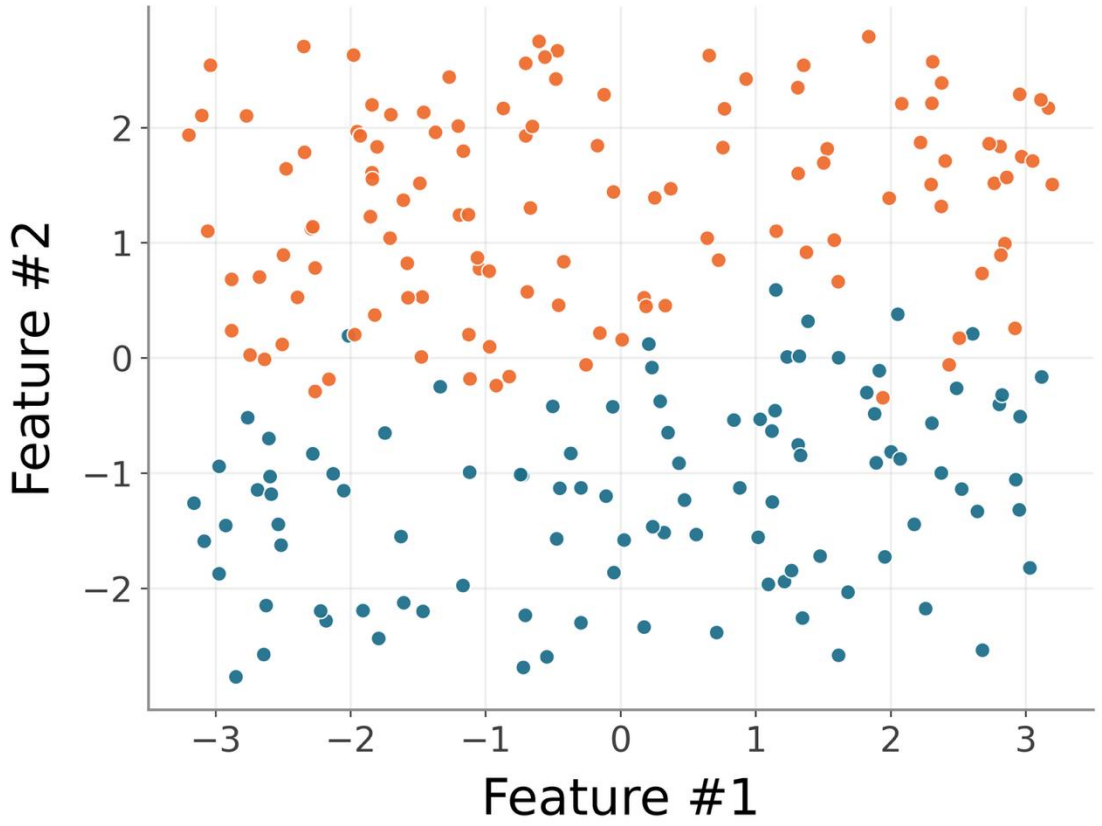


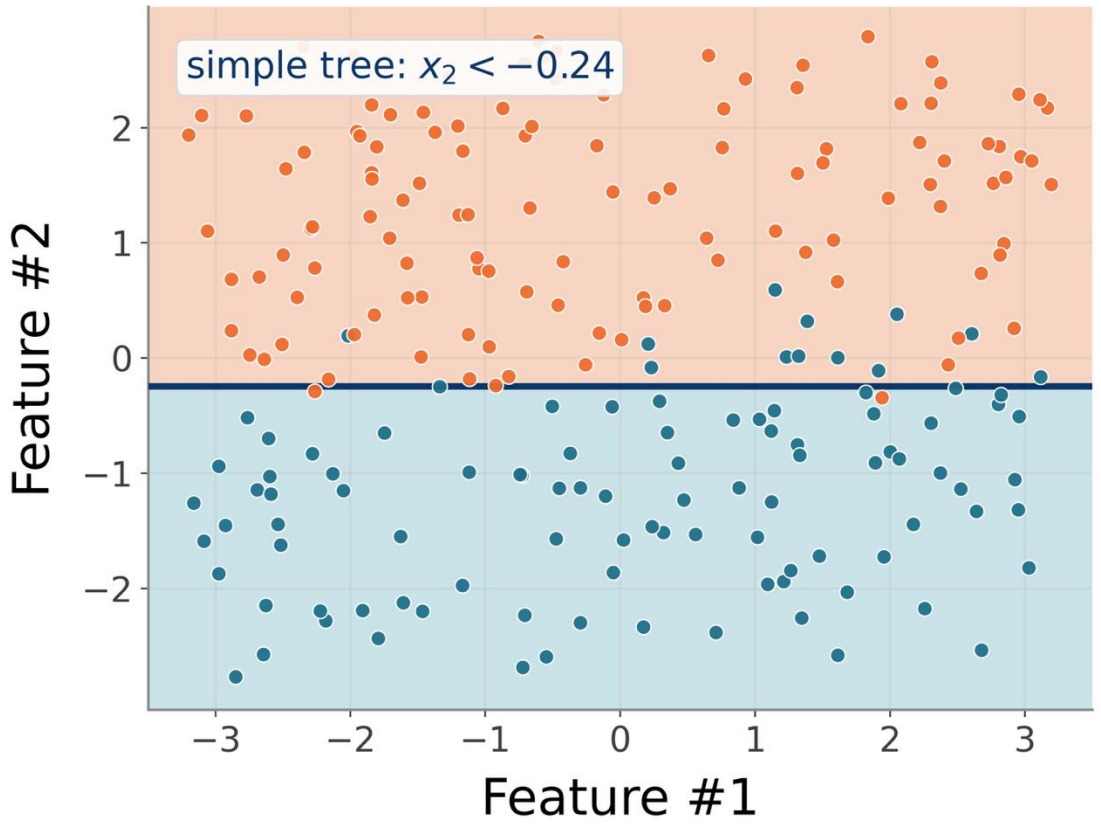
Weighted votes

Unlike bagging, boosting will build models in a **deliberate, non-random** fashion

Step 1: Train one simple model (just like before)

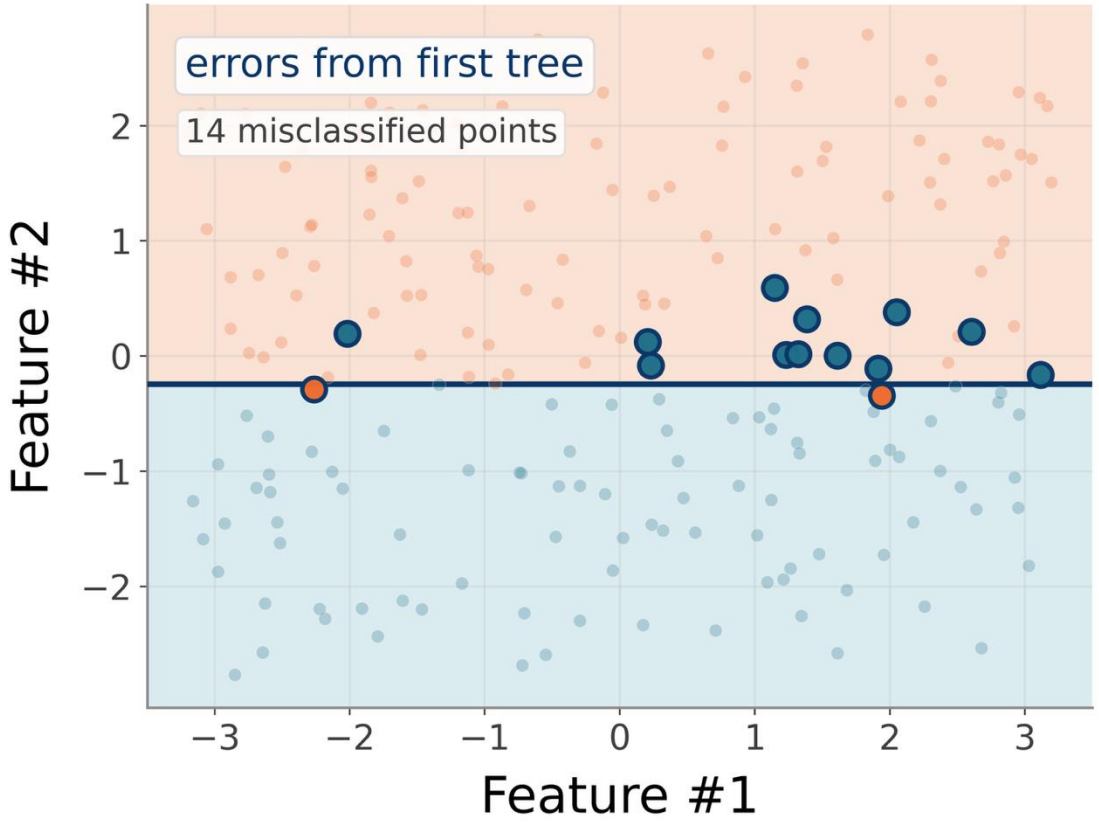
1. Train a simple decision tree on all the data – just one model

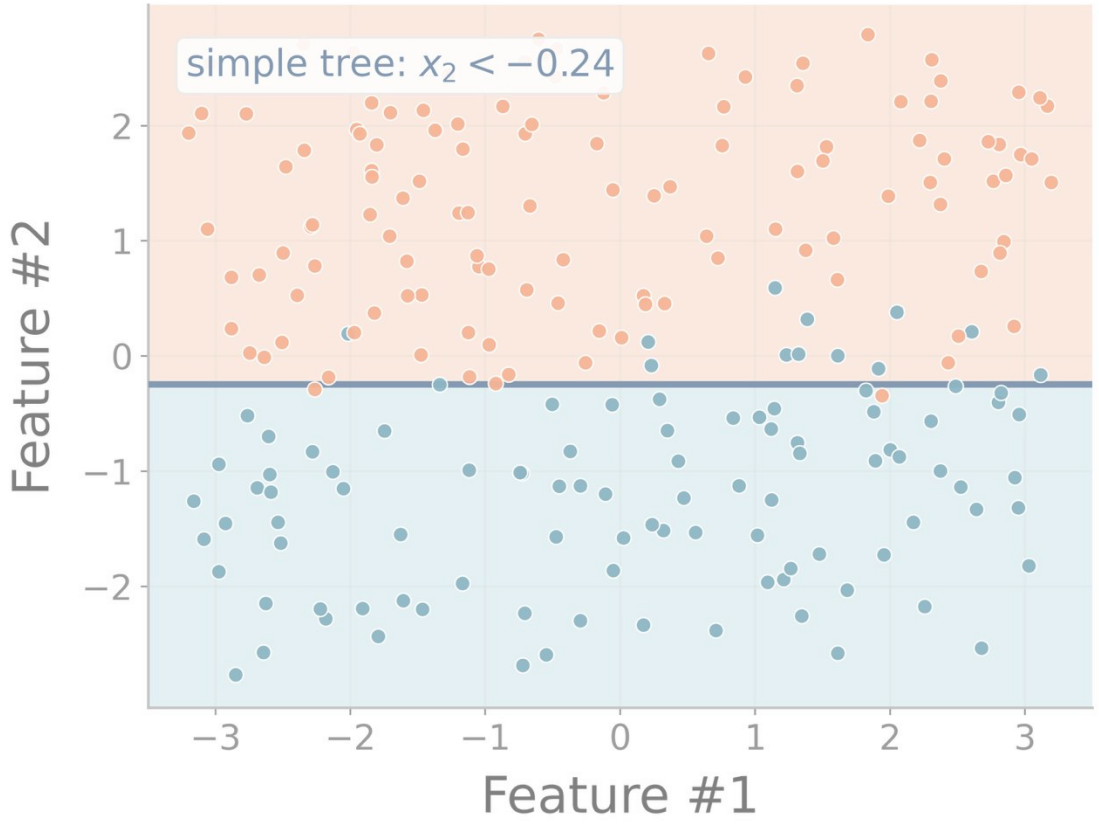




Step 2: Identify the “difficult” samples

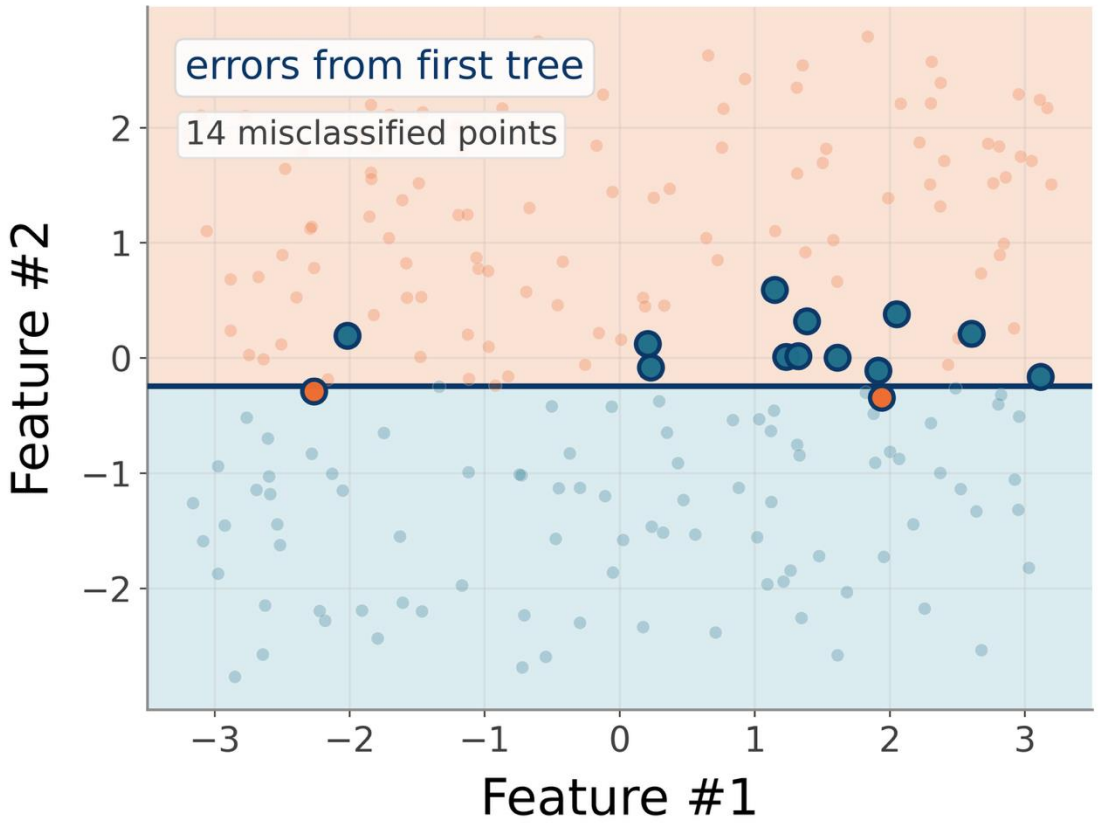
2. Identify the errors (misclassifications) made by that model





Step 3: Train a more specialized model

3. Train a new tree *to correct those errors* and form an ensemble



The Gini impurity:

$$G = 1 - \sum_k p_k^2$$

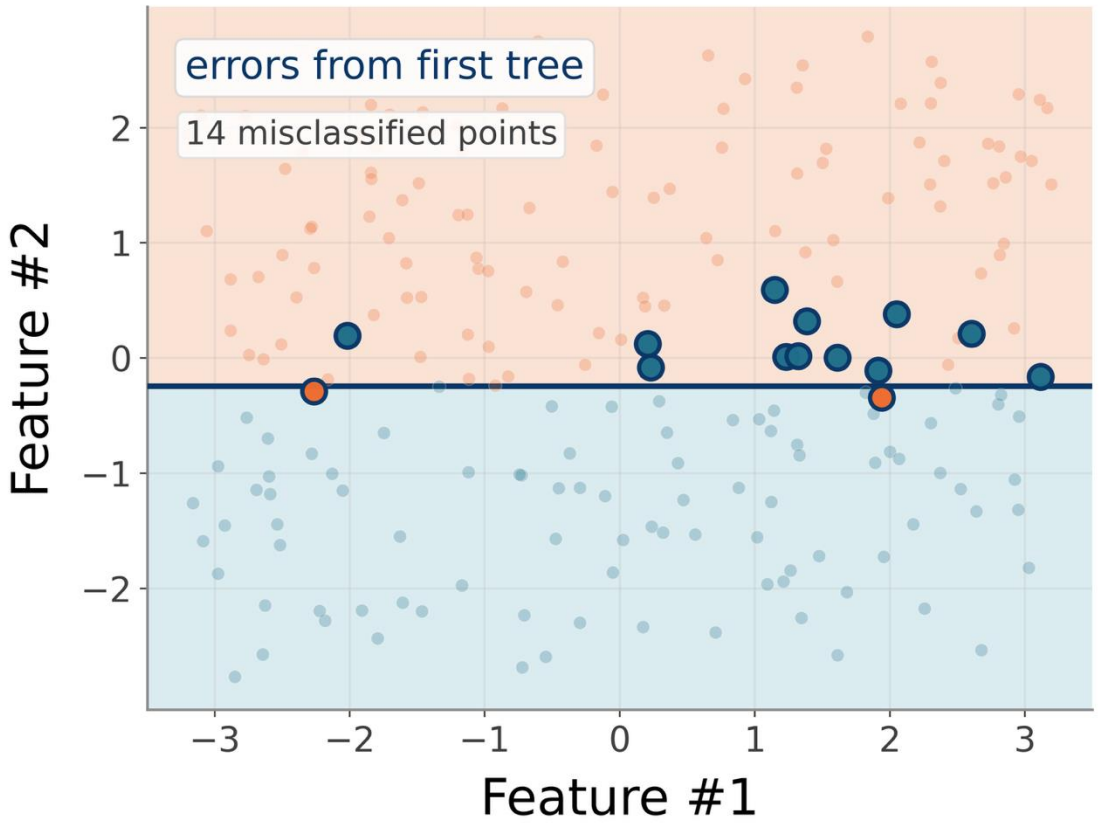
$$p_k = \frac{\text{Number of samples in class } k}{\text{Total number of samples in the node}}$$

$$p_k = \frac{\sum_{y_i=k} 1}{\sum_{i \in \text{node}} 1}$$

Typically, each sample contributes equally to the Gini impurity

Step 3: Train a more specialized model

3. Train a new tree *to correct those errors* and form an ensemble



The Gini impurity:

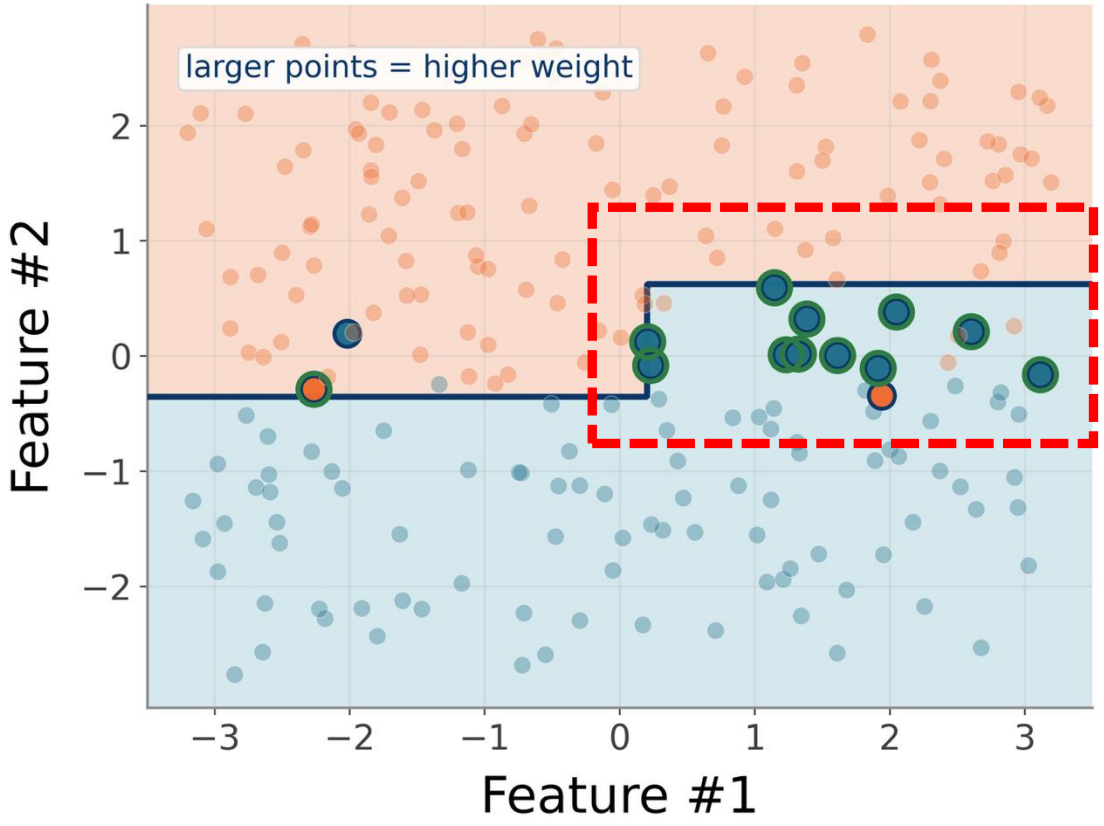
$$G = 1 - \sum_k p_k^2$$

$$p_k = \frac{\sum_{y_i=k} w_i}{\sum_{i \in \text{node}} w_i}$$

When boosting, we give more weight to the samples that were misclassified by the previous tree

Step 3: Train a more specialized model

3. Train a new tree *to correct those errors* and form an ensemble



Our new model specifically learned to classify the samples that were given higher weight!

When boosting, we give more weight to the samples that were misclassified by the previous tree

How are weights chosen: AdaBoost

1. Start by weighting all samples equally

$$w_i = \frac{1}{n}$$ ← **Total # samples**

2. After each tree, **increase the weights of misclassified samples** and decrease the weights of correctly classified samples

$$w_i^{t+1} \propto w_i^t \begin{cases} e^{+\alpha_t} \text{ if misclassified} \\ e^{-\alpha_t} \text{ if correct.} \end{cases}$$

$$\alpha_t = \frac{1}{2} \ln\left(\frac{1-\epsilon_t}{\epsilon_t}\right)$$

Where ϵ_t is the classification error of the original tree (t)

If the model is bad (high/random error, $\epsilon_t \sim$ **0.5**) we should update it do better **on *all* samples.**
→ Weight them equally: α_t **is small** and the **weights are not updated much**

If the model performs well overall (low error, ϵ_t **close to 0**) yet still misclassifies a few samples, we should give more weight to those samples and less to the ones already classified well
→ α_t **is large** and the **weights are changed a lot**

How are weights chosen: AdaBoost

1. Start by weighting all samples equally $w_i = \frac{1}{n}$

2. After each tree, **increase the weights of misclassified samples** and decrease the weights of correctly classified samples

$$w_i^{t+1} \propto w_i^t \begin{cases} e^{+\alpha_t} \text{ if misclassified} \\ e^{-\alpha_t} \text{ if correct.} \end{cases}$$

$$\alpha_t = \frac{1}{2} \ln\left(\frac{1-\epsilon_t}{\epsilon_t}\right)$$

Where ϵ_t is the classification error of the original tree (t)

This also controls how much of a vote each tree gets in the final ensemble

Small ϵ_t
Large α_t
Greater vote

$$P(y = k|x) \sum_{t=1}^{T} \alpha_t C_k(t)$$

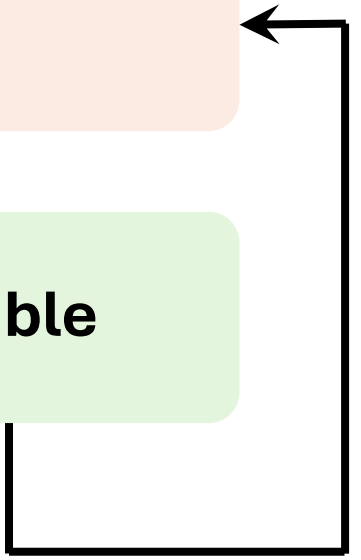
Where α_t gives higher weight to the classification $C_k(t)$ of a tree with low error

When to stop?

1. Train a simple decision tree on all the data: An ensemble of 1 model

2. Identify the errors (misclassifications) made by the ensemble

3. Train a new tree *to correct those errors* and add it to the ensemble

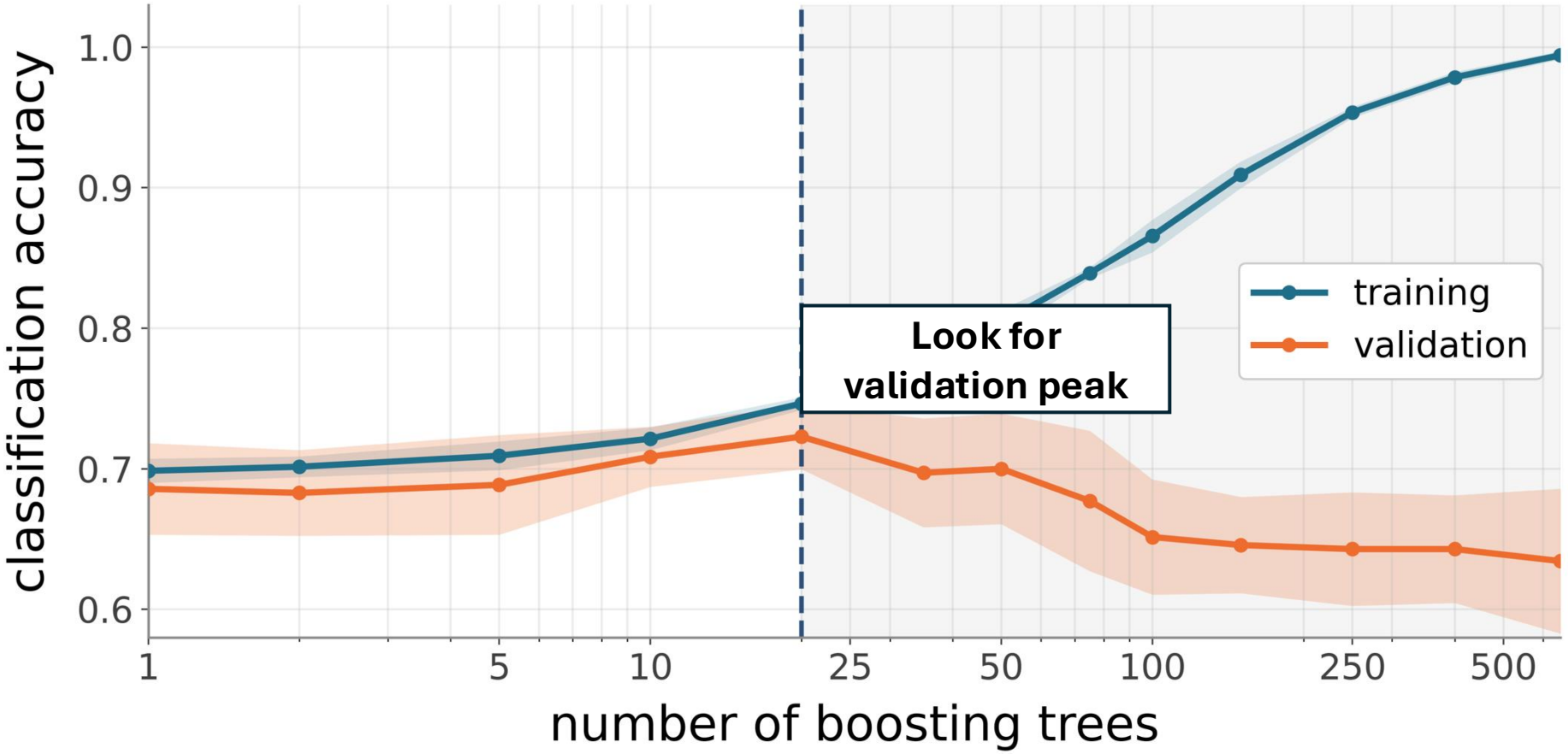


Keep training new decision trees until…

a) The maximum number of trees (which you set) is reached

b) The validation accuracy stops improving

With boosting, there is a risk of overfitting!



Unlike random forests, **more boosted trees are *not* always better** because later trees may begin fitting noise (**overfitting**). These trees are not independent!

Boosting usually gives better performance – but use with caution

Questions?

Comparing our classification models so far…

	Logistic reg.	k-NN	Trees / forests
boundary	linear *	flexible/jagged	axis-aligned boxes
scaling needed?	yes	yes (critical)	no
interpretable	coefficients	no	rules + importance
nonlinear?	* needs features	yes	yes, natively
predict speed	fast	slow	fast

- While individual trees are interpretable, random forests are less so…
- But random forests are great at handling non-linear data out of the box!
  No feature engineering nor standardization, and mixed types allowed
- With boosting, very strong performance is possible on complex tasks

Comparing our classification models so far…



XGBoost became famous through state-of-the-art performance on structured-data competitions and applications such as particle event classification, search ranking, fraud detection, and credit risk… Still used today!

We won’t cover XGBoost in detail, but know that it differs from AdaBoost by using gradient-based optimization of a loss function rather than reweighting misclassified observations.

What to know for HW + Exams

- How are decisions made (**Gini impurity, Information gain**)
  - You should be able to do these calculations by hand
- Influence of hyperparameters (tree depth, number of leaves, *etc.*)
  - You should know how these affect overfitting *vs.* underfitting
- Bootstrapping and **bagging *vs.* boosting**
- Understand how we measure feature importance