Lecture 008

Ensembles 🌲🌲🌲🎄🌲

Edward Rubin
18 February 2021

Admin

Today

Topic Ensembles (applied to decision trees)

Upcoming

Readings

- *Today* *ISL* Ch. 8.2
- *Next* *ISL* Ch. 9

Project Project topic and group due next week (Feb. 25th).

Decision trees

Review

Decision trees

Fundamentals

Decision trees

- split the *predictor space* (our $\mathbf{X}$) into regions
- then predict the most-common value within a region

Regression trees

- **Predict:** Region's mean
- **Split:** Minimize RSS
- **Prune:** Penalized RSS

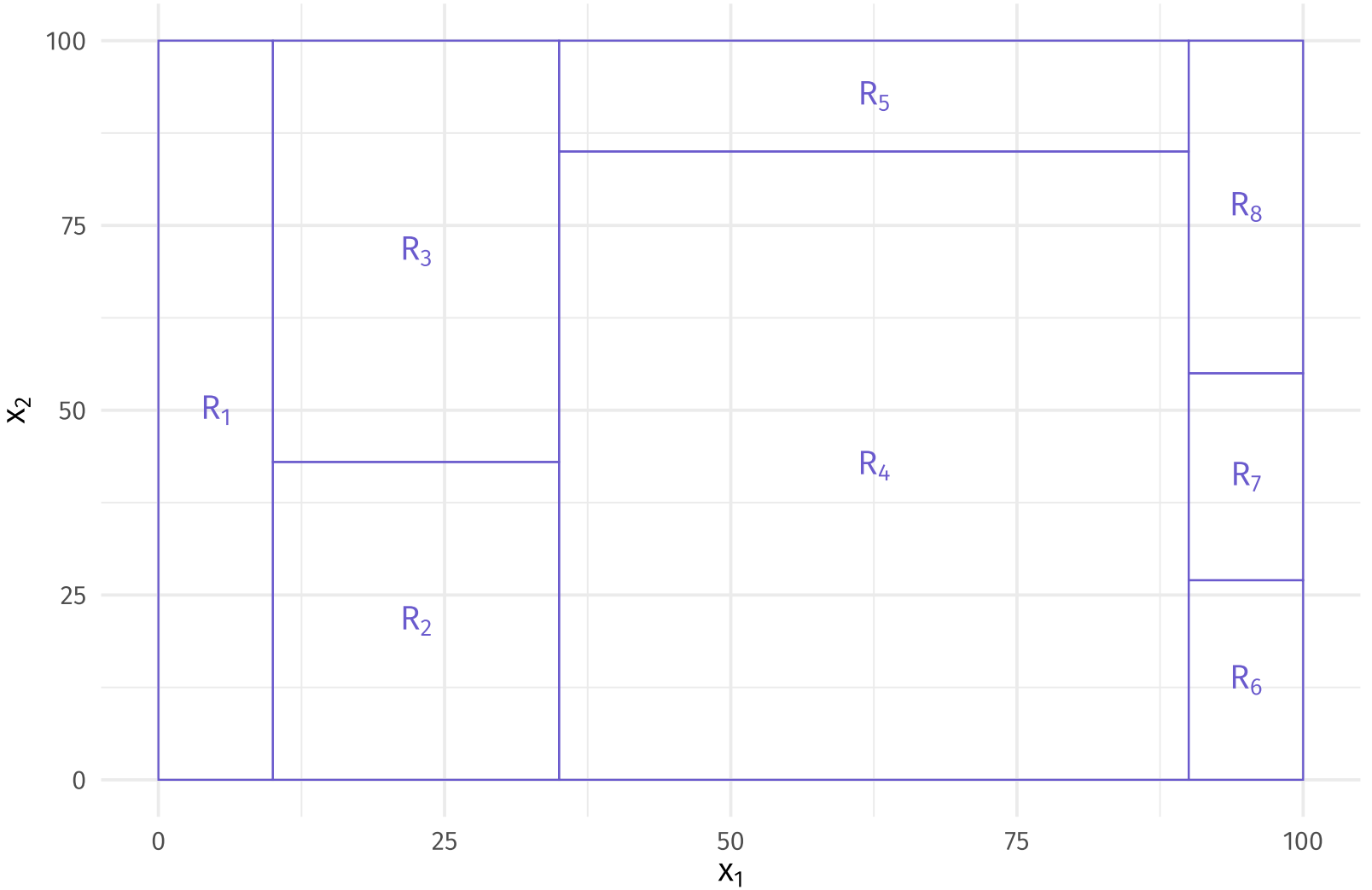
Classification trees

- **Predict:** Region's mode
- **Split:** Min. Gini or entropy.super
- **Prune:** Penalized error rate🌴

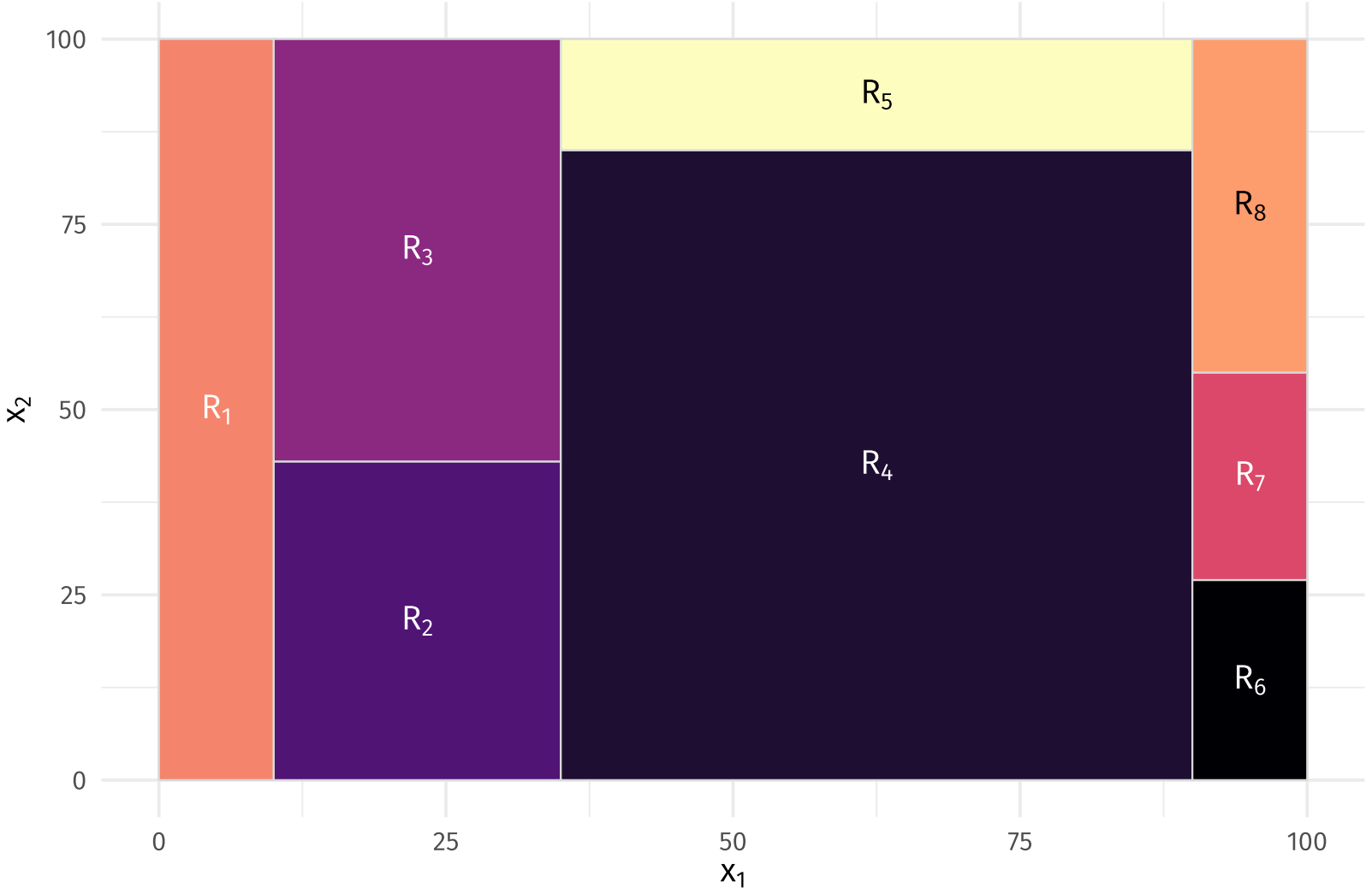
An additional nuance for classification trees: we typically care about the **proportions of classes in the leaves**—not just the final prediction.

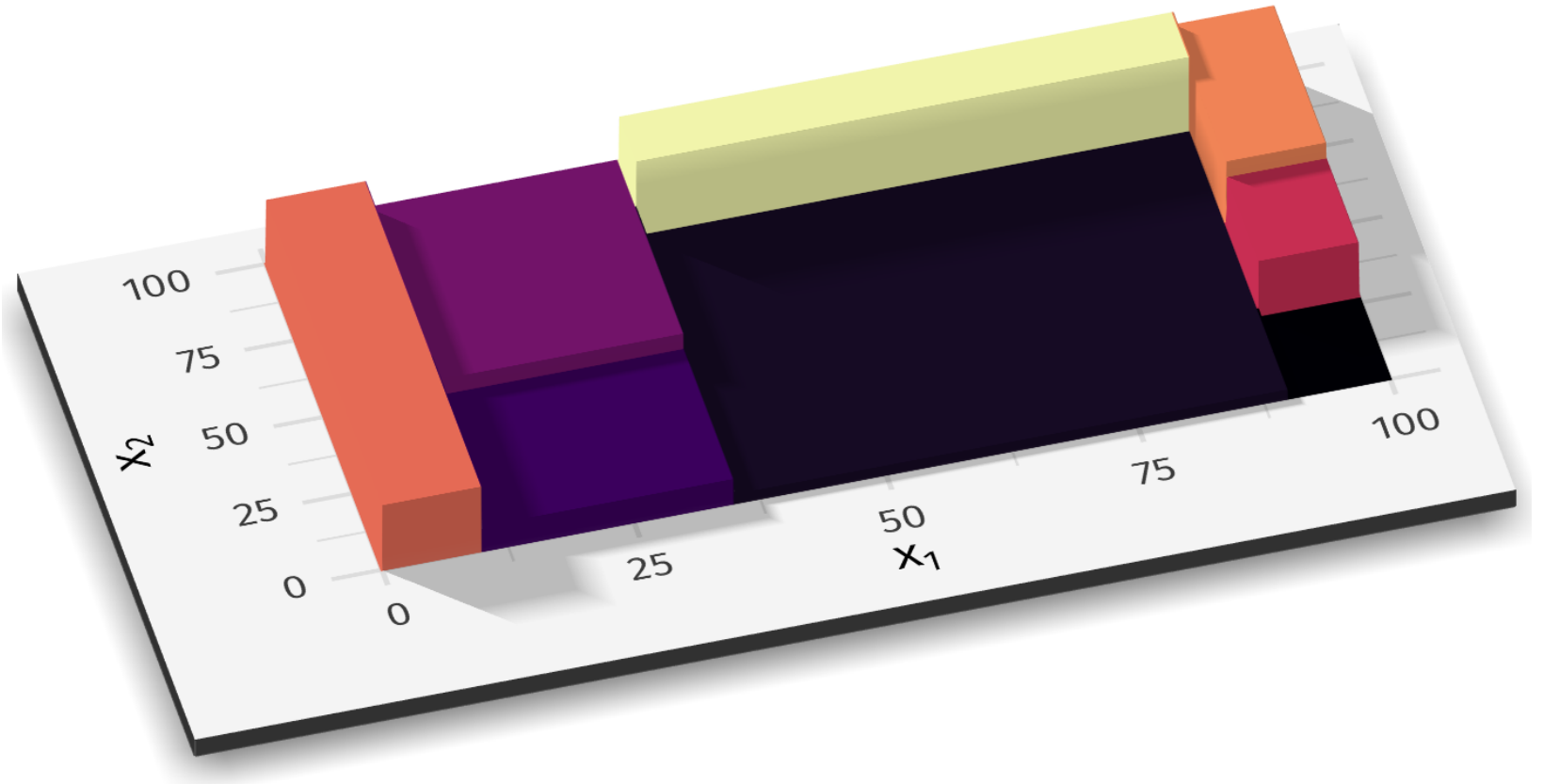
🌴 ... or Gini index or entropy

Example Each split in our tree creates **regions**.



Example Each region has its own **predicted value**.




100
75
50
25
0
x_2
0
25
50
75
100
x_1

Decision trees

Strengths and weaknesses

As with any method, decision trees have tradeoffs.

Strengths

+ Easily explained/interpretted
+ Include several graphical options
+ Mirror human decision making?
+ Handle num. or cat. on LHS/RHS🌳

Weaknesses

- Outperformed by other methods
- Struggle with linearity
- Can be very "non-robust"

Non-robust: Small data changes can cause huge changes in our tree.

Next: Create ensembles of trees🌲 to strengthen these weaknesses.🌴

🌳 Without needing to create lots of dummy variables!

🌲 Forests! 🌴 Which will also weaken some of the strengths.

Ensemble methods

Ensemble methods

Intro

Rather than focusing on training a **single**, highly accurate model, ensemble methods combine **many** low-accuracy models into a *meta-model*.

Today: Three common methods for **combining individual trees**

1. Bagging
2. Random forests
3. Boosting

Why? While individual trees may be highly variable and inaccurate, a combination of trees is often quite stable and accurate.🌲

🌲 We will lose interpretability.

Ensemble methods

Bagging

Bagging creates additional samples via **bootstrapping**.

Q How does bootstrapping help?

A *Recall:* Individual decision trees suffer from variability (*non-robust*).

This *non-robustness* means trees can change *a lot* based upon which observations are included/excluded.

We're essentially using many "draws" instead of a single one.🌴

🌴 Recall that an estimator's variance typically decreases as the sample size increases.

Bagging

Bootstrap aggregation (bagging) reduces this type of variability.

1. Create B bootstrapped samples
2. Train an estimator (tree) $\hat{f}^b(x)$ on each of the B samples
3. Aggregate across your B bootstrapped models:

$$\hat{f}_{\text{bag}}(x) = \frac{1}{B}\sum_{b=1}^{B}\hat{f}^b(x)$$

This aggregated model $\hat{f}_{\text{bag}}(x)$ is your final model.

Bagging trees

When we apply bagging to decision trees,

- we typically **grow the trees deep and do not prune**
- for **regression**, we **average** across the B trees' regions
- for **classification**, we have more options—but often take **plurality**

Individual (unpruned) trees will be very **flexible** and **noisy**,
but their **aggregate** will be quite **stable**.

The number of trees B is generally not critical with bagging.
$B = 100$ often works fine.

Out-of-bag error estimation

Bagging also offers a convenient method for evaluating performance.

For any bootstrapped sample, we omit ~n/3 observations.

Out-of-bag (OOB) error estimation estimates the test error rate using observations **randomly omitted** from each bootstrapped sample.

For each observation i:

1. Find all samples S_i in which i was omitted from training.
2. Aggregate the $|S_i|$ predictions $\hat{f}^b(x_i)$, *e.g.*, using their mean or mode
3. Calculate the error, *e.g.*, $y_i - \hat{f}_{i,\text{OOB},i}(x_i)$

Out-of-bag error estimation

When B is big enough, the OOB error rate will be very close to LOOCV.

Q Why use OOB error rate?

A When B and n are large, cross validation—with any number of folds—can become pretty computationally intensive.

Quick aside: Here is a tool to search `parsnip` models:

> https://www.tidymodels.org/find/parsnip/

Bagging in R

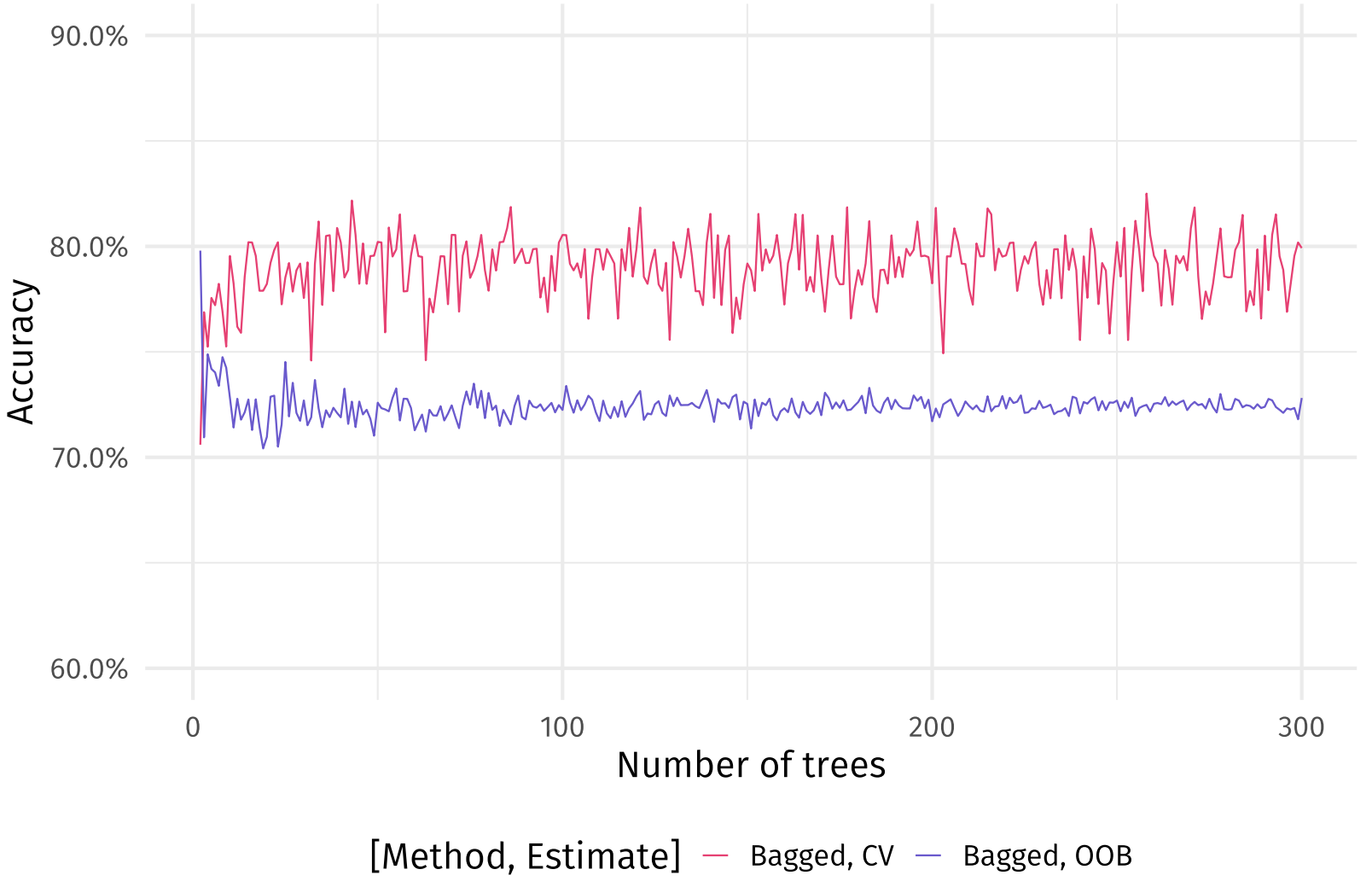
We can use `tidymodels` plus the `baguette` package to bag trees.

Function: `bag_treebag()`

- "Specifies" model for `parsnip`.
- `mode`: class., reg., or unknown
- `cost_complexity`: the penalty for model complexity (`Cp`)
- `tree_depth`: max. tree depth
- `min_n`: min. # obs. to split
- `class_cost`: magnify *cost*
- `rpart` is the defaul engine
- `times`: the number of trees

```
# Train a bagged tree model
bag_tree(
  mode = "classification",
  cost_complexity = 0,
  tree_depth = NULL,
  min_n = 2,
  class_cost = NULL
) %>% set_engine(
  engine = "rpart",
  times = 100
)
```

Bagging and the number of trees



Unfortunately, this combination of `rpart` / `baguette` / `parsnip` / `yardstick` doesn't (currently) offer OOB-based metrics. 😞

We *can* "trick" random forests (`ranger`) into doing OOB for bagged trees.

But first, we need to learn about random forests.

But before *that*, let's talk about variable importance.

Ensemble methods

Variable importance

While ensemble methods tend to **improve predictive performance**, they also tend **reduce interpretability**.

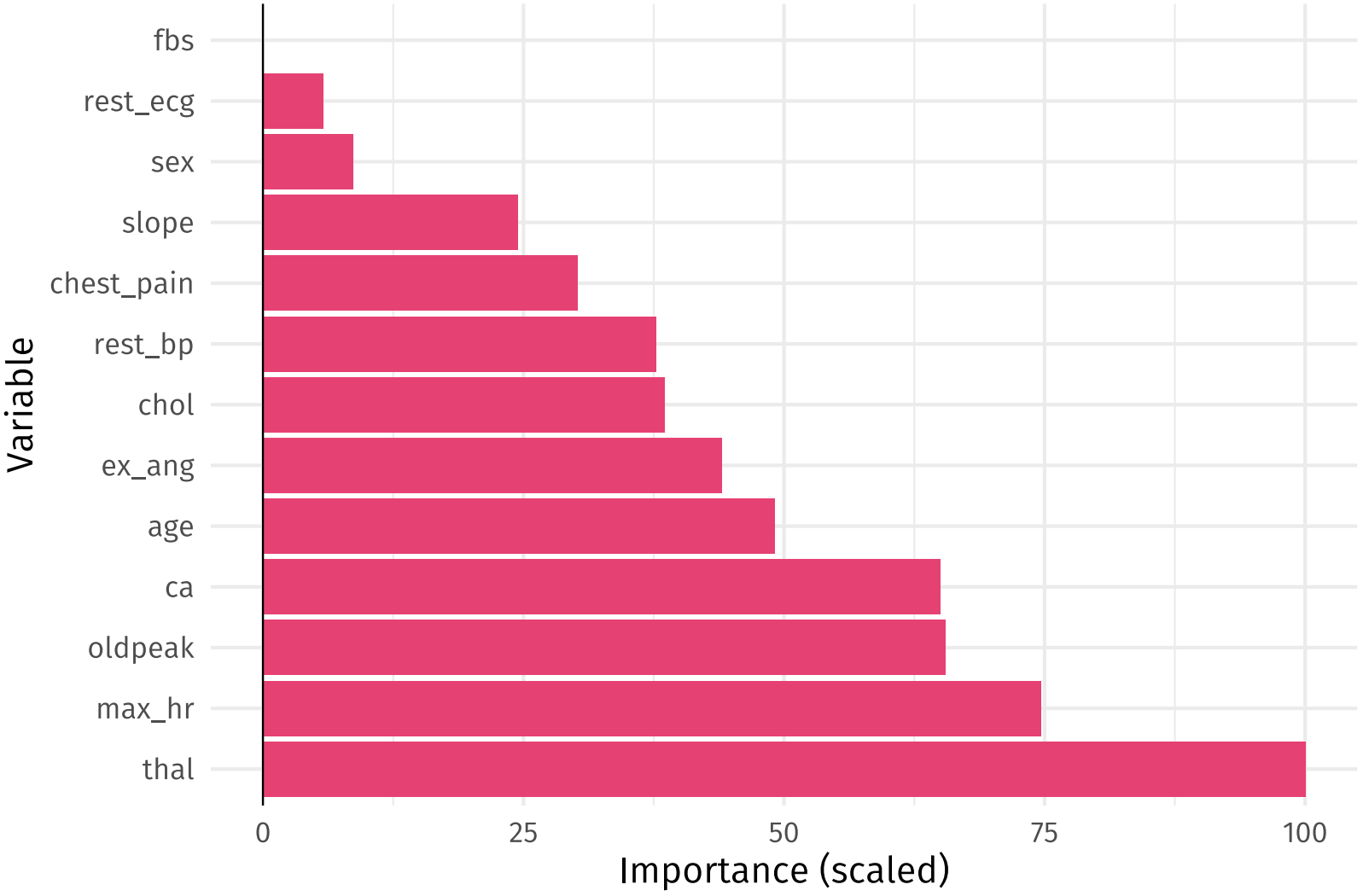
We can illustrate **variables' importance** by considering their splits' reductions in the model's performance metric (RSS, Gini, entropy, *etc.*).🌳

In R, we can use `caret`'s `varImp()` function to calculate variable important.

Note By default, `varImp()` will scale improtance between 0 and 100.

🌳 This idea isn't exclusive to bagging/ensembles—we can (and do) apply it to a single tree.

Variable importance from our bagged tree model.



Bagging

Bagging has one additional shortcoming...

If one variable dominates other variables, the **trees will be very correlated**.

If the trees are very correlated, then bagging loses its advantage.

Solution We should make the trees less correlated.

Random forests

Random forests improve upon bagged trees by *decorrelating* the trees.

In order to decorrelate its trees, a **random forest** only considers a random subset of m $(\approx \sqrt{p})$ predictors when making each split (for each tree).

Restricting the variables our tree sees at a given split

- nudges trees away from always using the same variables,
- increasing the variation across trees in our forest,
- which potentially reduces the variance of our estimates.

If our predictors are very correlated, we may want to shrink m.

Random forests

Random forests thus introduce **two dimensions of random variation**

1. the **bootstrapped sample**
2. the m **randomly selected predictors**

Everything else about random forests works just as it did with bagging.🎄

🎄 And just as it did with plain, old decision trees.

Ensemble methods

Random forests in R

You have *many* options for training random forests in R.
E.g., `party`, `Rborist`, `ranger`, `randomForest`.

`caret` offers access to each of these packages via `train`.

- *E.g.*, `method = "rf"` or `method = "ranger"`
- The argument `mtry` gives the number of predictors at each split.🌲
- Some methods have additional parameters, *e.g.*, `ranger` needs
  - minimal node size `min.node.size`
  - a splitting rule `splitrule`.

🌲 `predFixed` for `Rborist`.

Ensemble methods

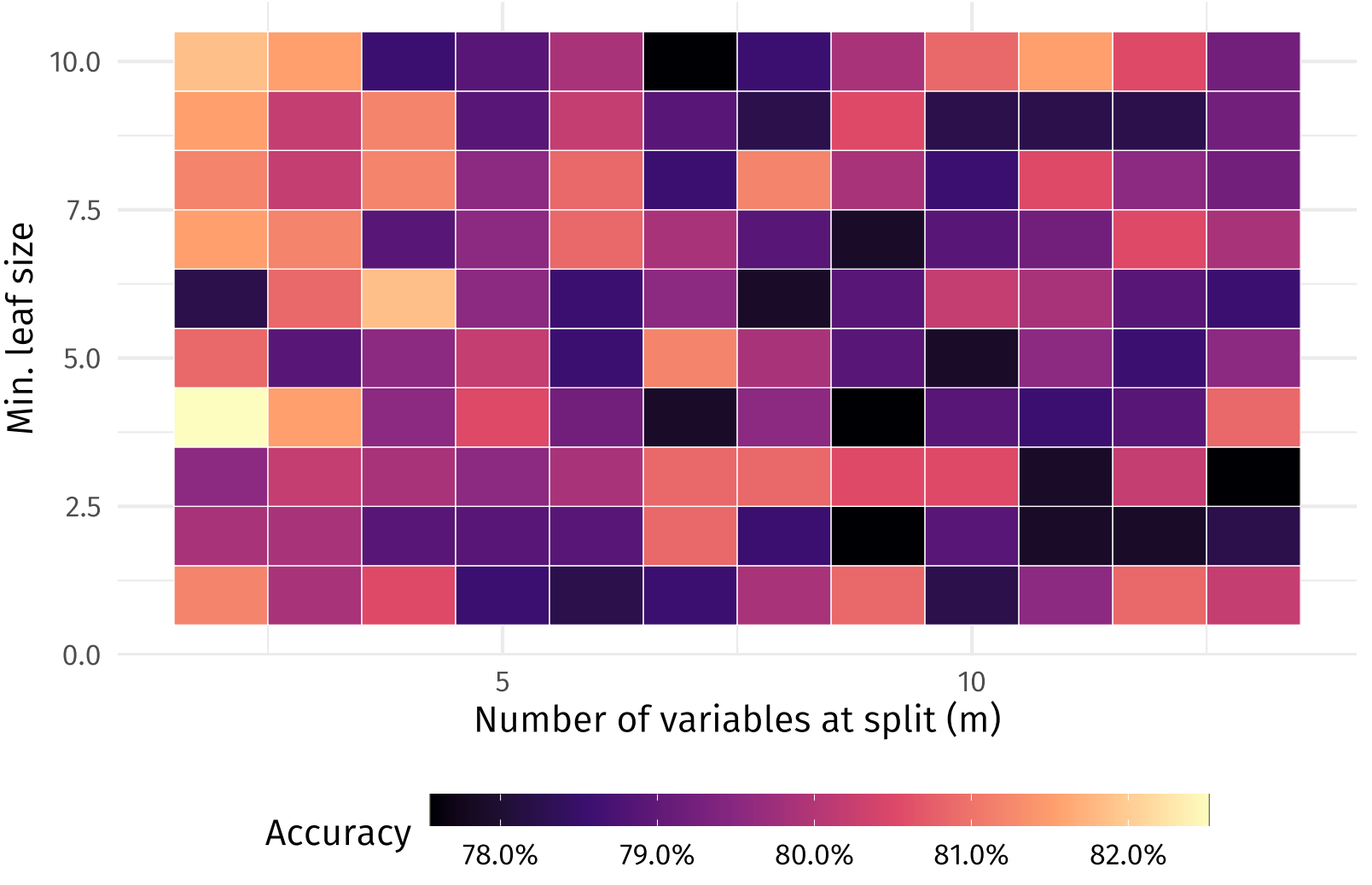
Training a random forest in R using `caret`...

... and `ranger`

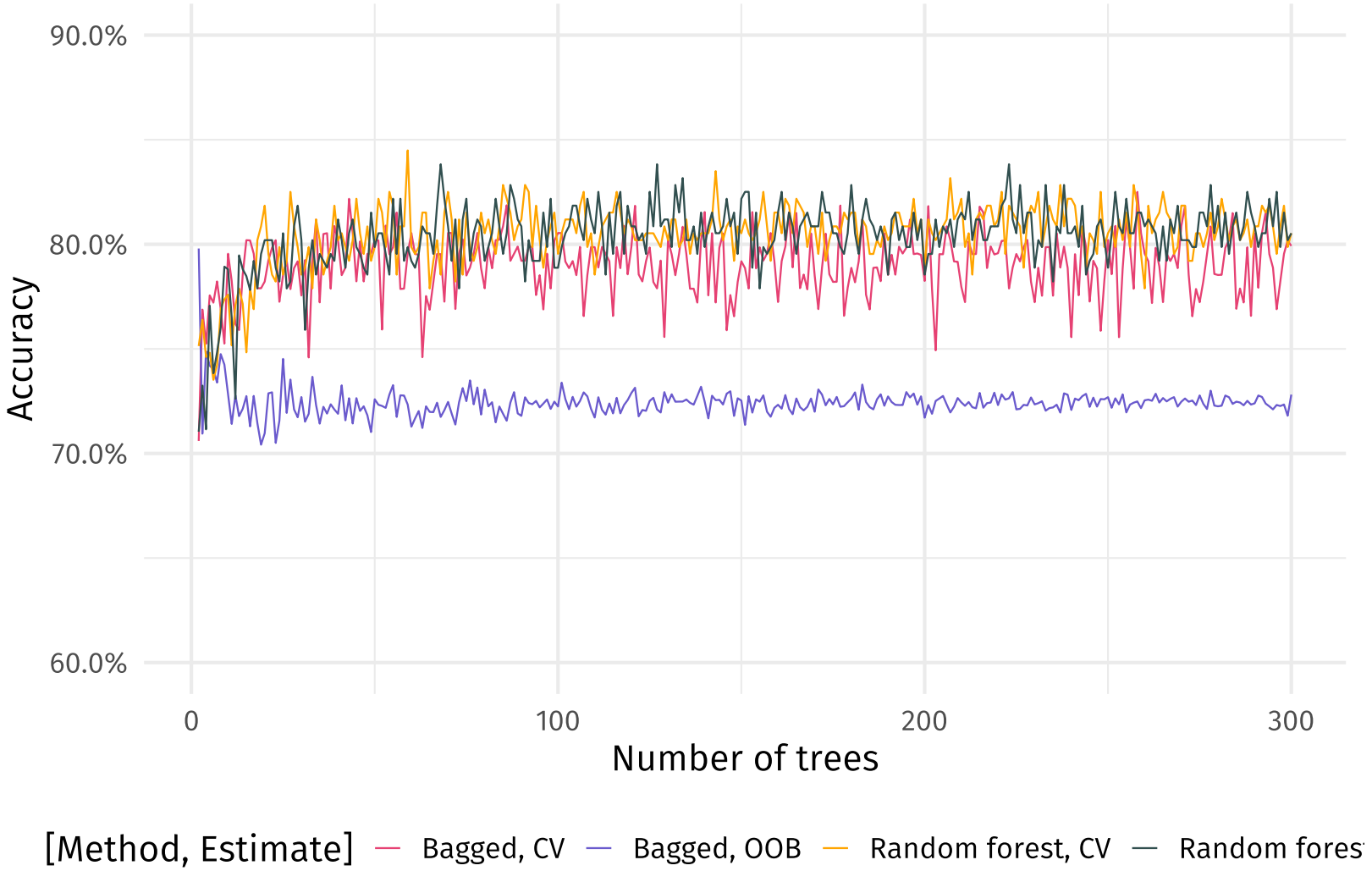
- Specify `"ranger"` for method
- Number of trees: `num.trees`
- We can still use OOB for error
- Parameters to choose/train
  1. m, # of predictors at a split
  2. the rule for splitting
  3. minimum size for a leaf

```
# Set the seed
set.seed(12345)
# Train the random forest
heart_forest = train(
  heart_disease ~ .,
  data = heart_df,
  method = "ranger",
  num.trees = 100,
  trControl = trainControl(
    method = "oob"
  ),
  tuneGrid = expand.grid(
    "mtry" = 2:13,
    "splitrule" = "gini",
    "min.node.size" = 1:10
  )
)
```

Accuracy (OOB) across the grid of our parameters.



Tree ensembles and the number of trees



Ensemble methods

Boosting

So far, the elements of our ensembles have been acting independently: any single tree knows nothing about the rest of the forest.

Boosting allows trees to pass on information to eachother.

Specifically, **boosting** trains its trees🌲 *sequentially*—each new tree trains on the residuals (mistakes) from its predecessors.

- We add each new tree to our model $\hat{f}$ (and update our residuals).
- Trees are typically small—slowly improving $\hat{f}$ *where it struggles*.

🌲 As with bagging, boosting can be applied to many methods (in addition to trees).

Boosting

Boosting has three **tuning parameters**.

1. The **number of trees** B can be important to prevent overfitting.
2. The **shrinkage parameter** λ, which controls boosting's *learning rate* (often 0.01 or 0.001).
3. The **number of splits** d in each tree (trees' complexity).
    - Individaul trees are typically short—often $d = 1$ ("stumps").
    - *Remember* Trees learn from predecessors' mistakes, so no single tree needs to offer a perfect model.

How to boost

Step 1: Set $\hat{f}(x) = 0$, which yields residuals $r_i = y_i$ for all i.

Step 2: For $b = 1, 2 \ldots, B$ do:

A. Fit a tree $\hat{f}^b$ with d splits.

B. Update the model $\hat{f}$ with "shrunken version" of new treee $\hat{f}^b$

$$\hat{f}(x) \leftarrow \hat{f}(x) + \lambda \hat{f}^b(x)$$

C. Update the residuals: $r_i \leftarrow r_i - \lambda \hat{f}^b(x)$.

Step 3: Output the boosted model: $\hat{f}(x) = \sum_b \lambda \hat{f}^b(x)$.

Ensemble methods

Boosting in R

We will use `caret`'s `method = "gbm"` to train boosted trees.🌴

`gbm` needs the three standard parameters of boosted trees—plus one more:

1. `n.trees`, the number of trees (B)
2. `interaction.depth`, trees' depth (max. splits from top)
3. `shrinkage`, the learning rate (λ)
4. `n.minobsinnode`, minimum observations in a terminal node

🌴 This method uses the `gbm` package.

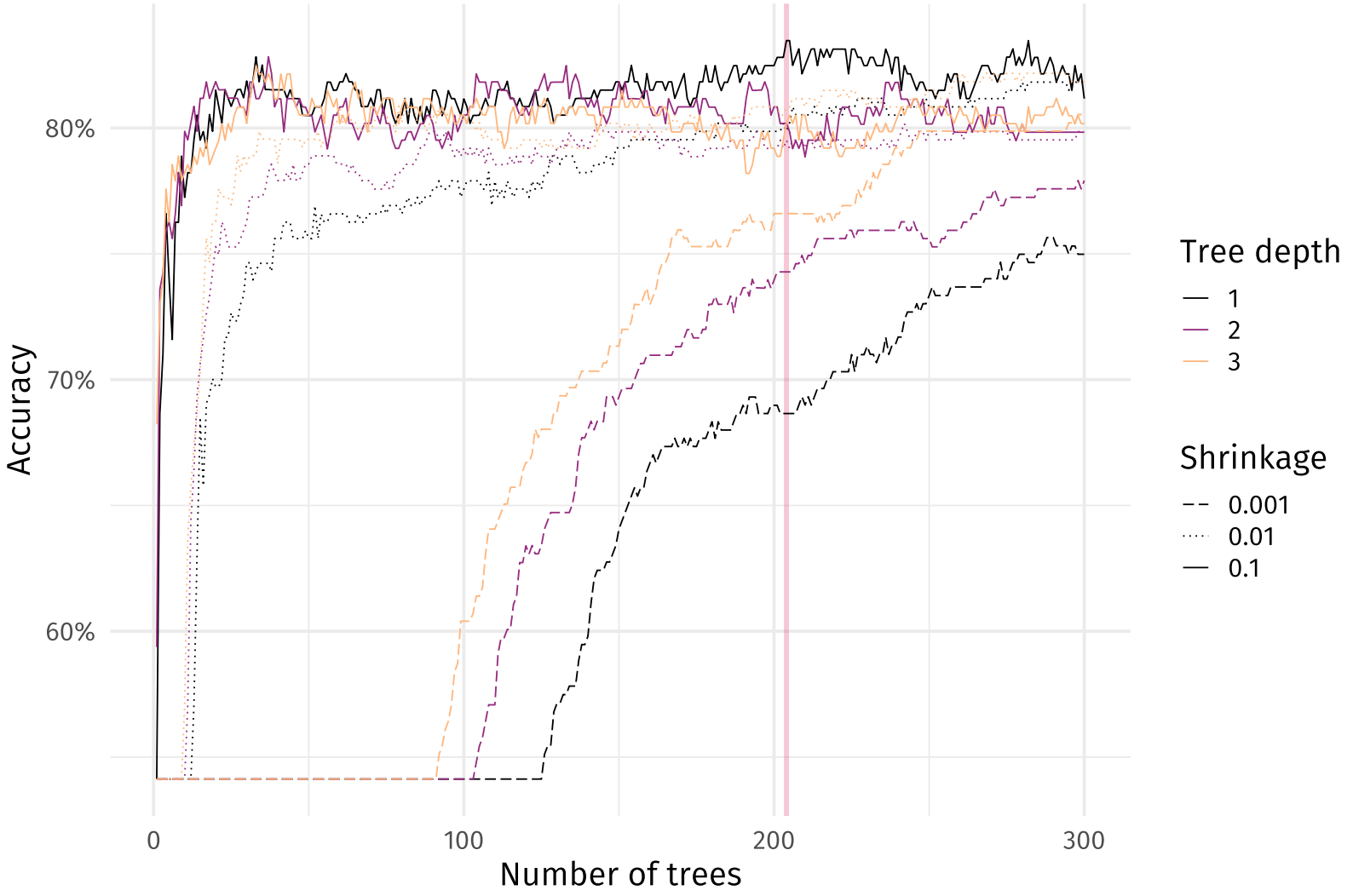
Ensemble methods

Boosting in R

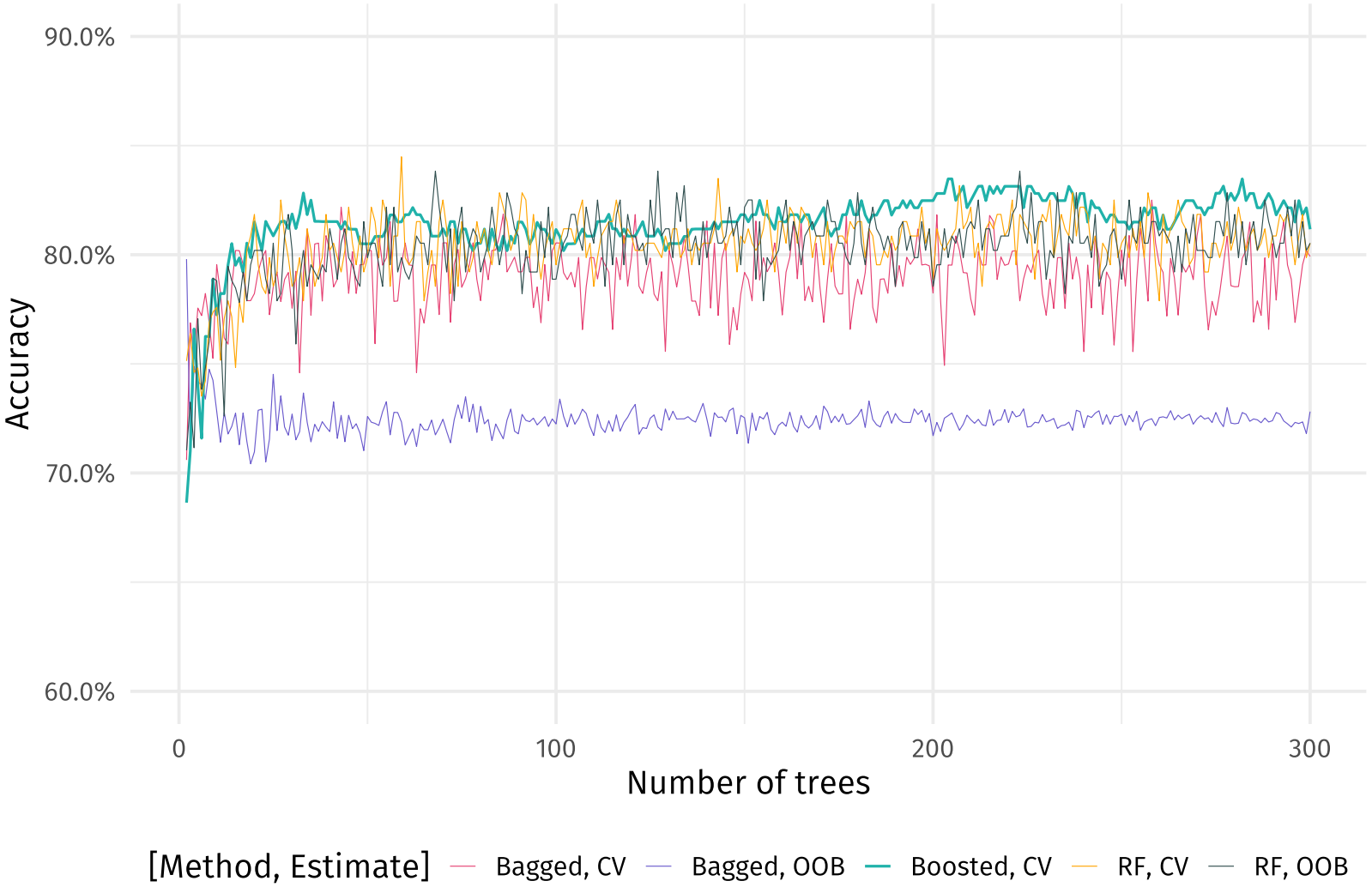
```
# Set the seed
set.seed(12345)
# Train the random forest
heart_boost = train(
  heart_disease ~ .,
  data = heart_df,
  method = "gbm",
  trControl = trainControl(
    method = "cv",
    number = 5
  ),
  tuneGrid = expand.grid(
    "n.trees" = seq(25, 200, by = 25),
    "interaction.depth" = 1:3,
    "shrinkage" = c(0.1, 0.01, 0.001),
    "n.minobsinnode" = 5
  )
)
```

- boosted trees via `gbm` package
- cross validation now (no OOB)
- CV-search of parameter grid
  - number of trees
  - tree depth (complexity)
  - shrinkage (learing rate)
  - minimum leaf size
    (not searching here)

Comparing boosting parameters—notice the rates of learning



Tree ensembles and the number of trees



Sources

These notes draw upon

- An Introduction to Statistical Learning (*ISL*)
  James, Witten, Hastie, and Tibshirani

Table of contents