

Lecture 008

Ensembles

Edward Rubin

Winter 2022

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

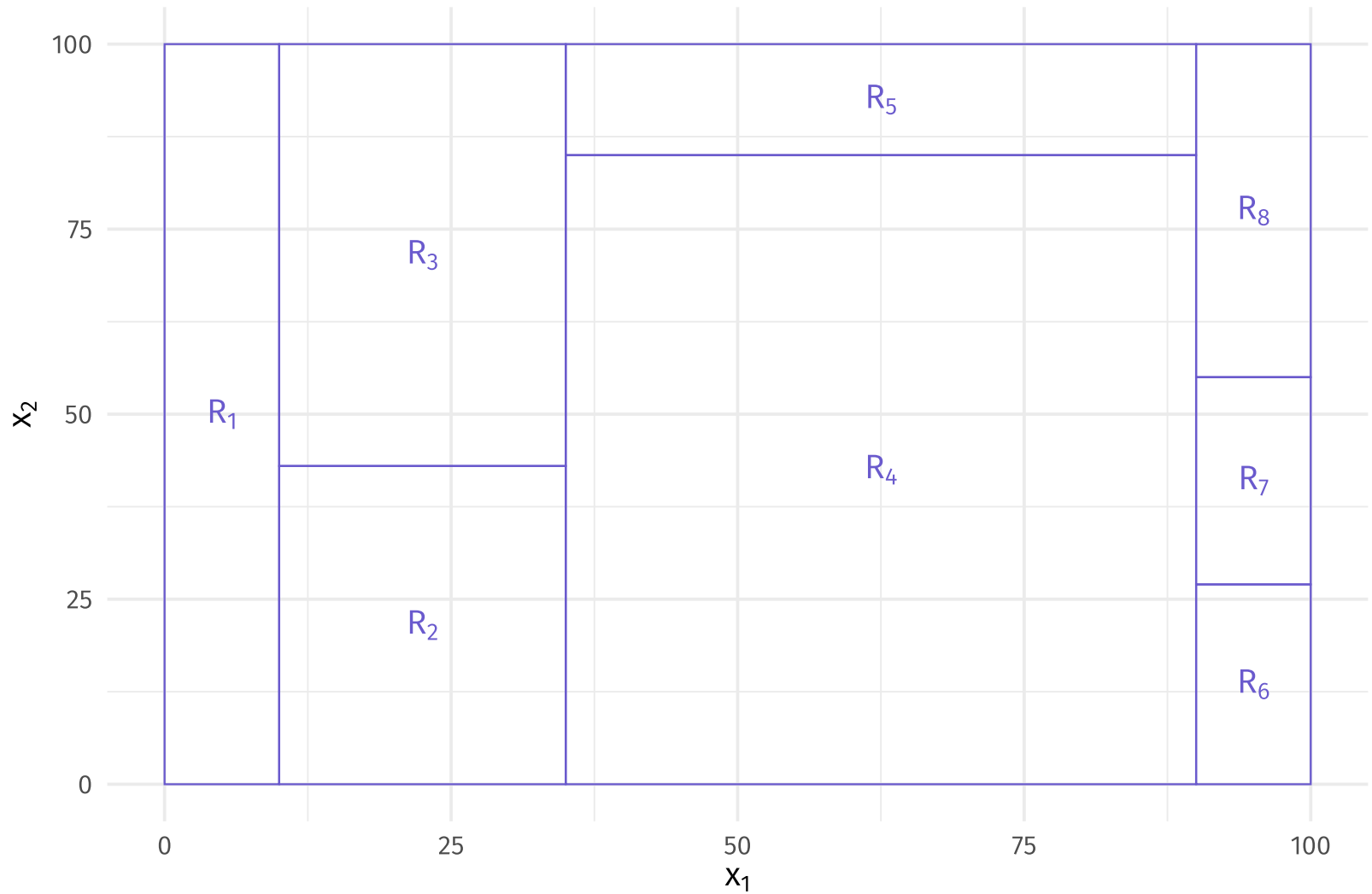
Classification trees

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

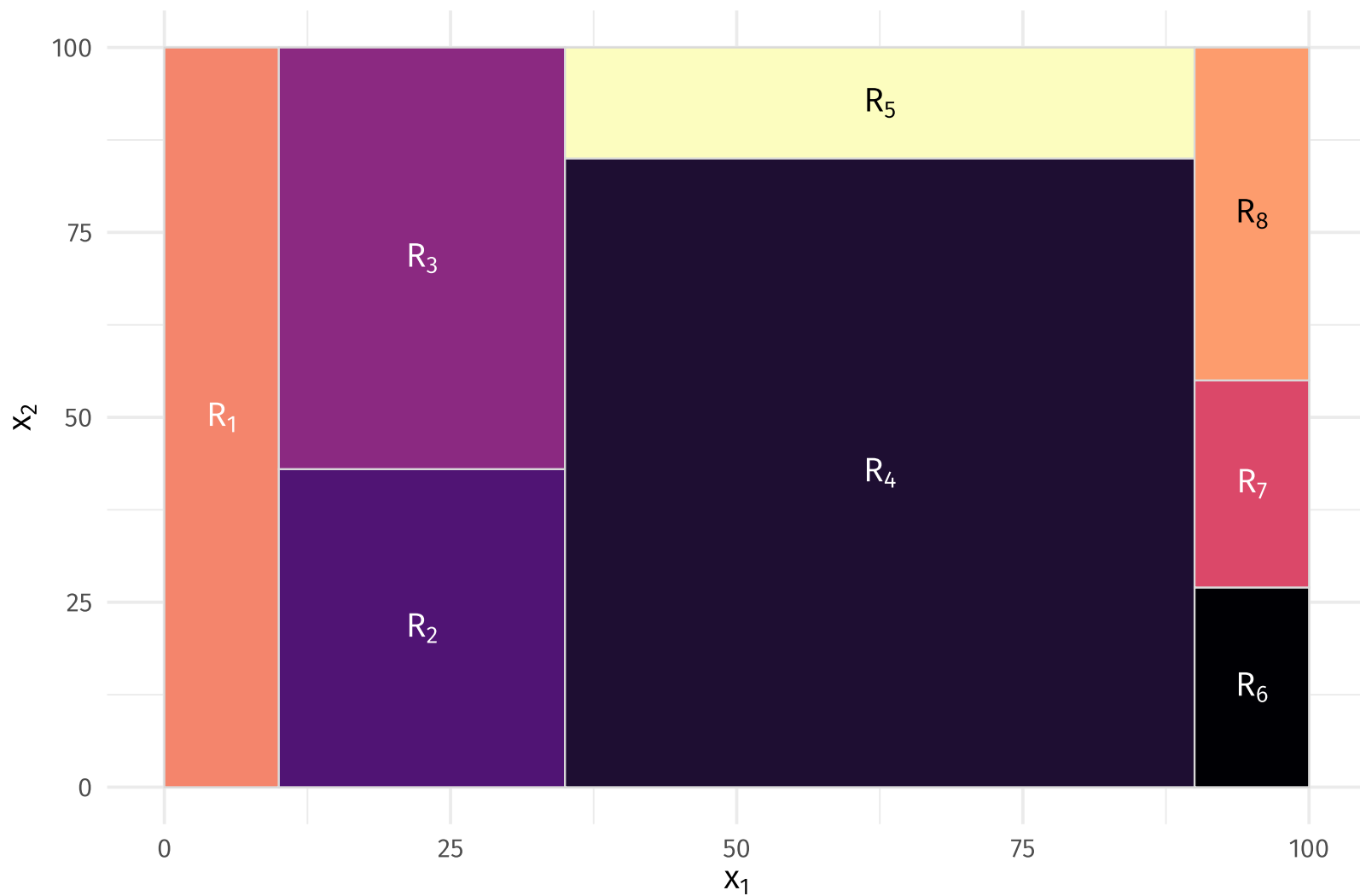
An additional nuance for **classification trees**: we typically care about the **proportions of classes in the leaves**—not just the final prediction.

🌴 ... or Gini index or entropy

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



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




Decision trees

Strengths and weaknesses

As with any method, decision trees have tradeoffs.

Strengths

- + Easily explained/interpreted
- + 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.

Ensemble methods

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.

Ensemble methods

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.

Ensemble methods

Out-of-bag error estimation

Bagging also offers a convenient method for evaluating performance.

For any bootstrapped sample, we omit $\sim n/3$ 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., $y_i - \hat{f}_{i,\text{OOB},i}(x_i)$

Ensemble methods

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/>

Ensemble methods

Bagging in R

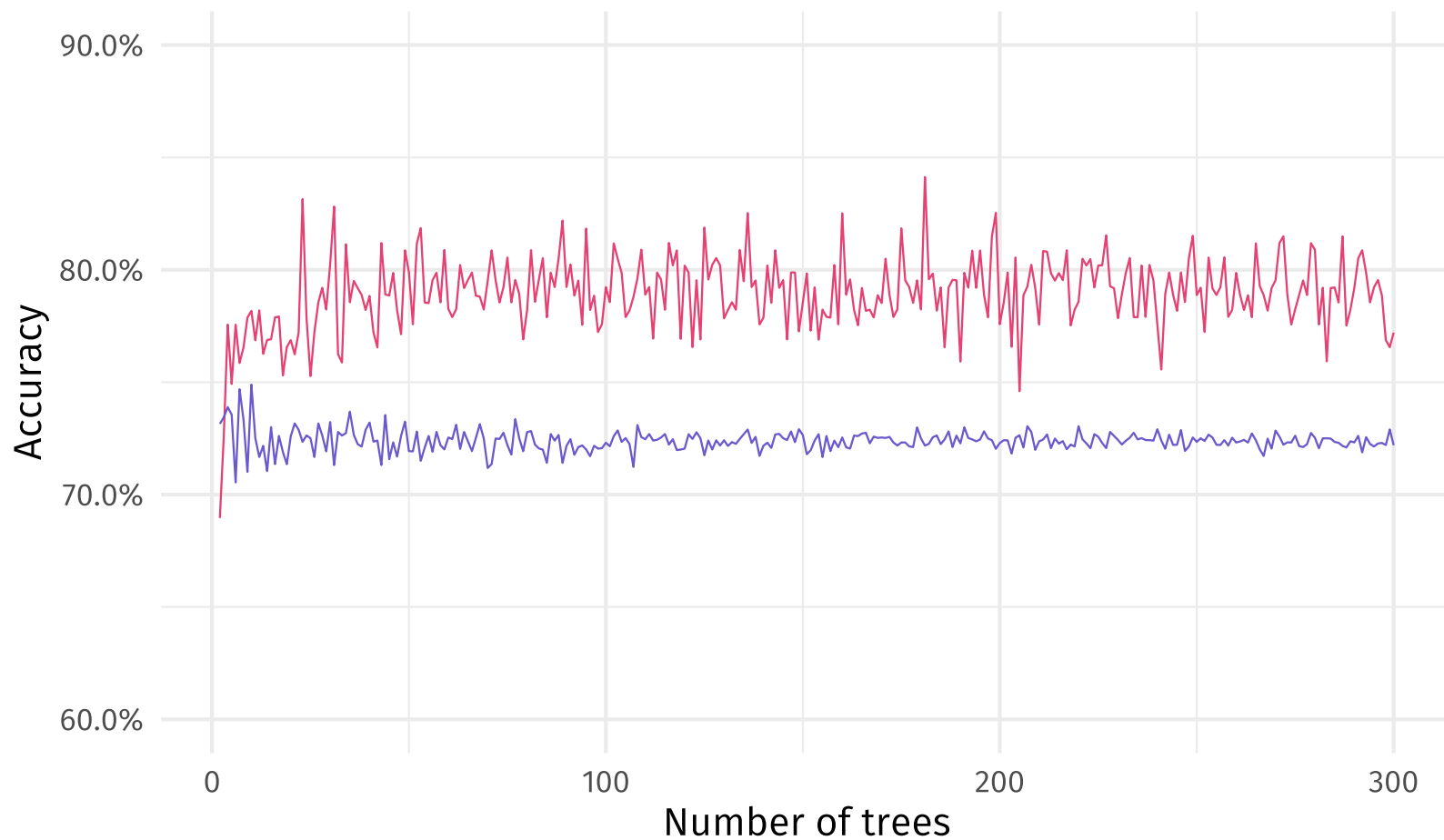
We can use `tidymodels` plus the `baguette` package to bag trees.

Function: `bag_tree()`

- "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 default 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



[Method, Estimate] — Bagged, CV — Bagged, OOB

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...

... and before *that*, let's briefly 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.). 🌳

Note By default, many variable-importance functions will scale importance.

🌳 This idea isn't exclusive to bagging/ensembles; it also works for a single tree.

In the case of "rpart" bagged trees...

```
# Recipe to clean data (impute NAs)
heart_recipe = recipe(heart_disease ~ ., data = heart_df) %>%
  step_medianimpute(all_predictors() & all_numeric()) %>%
  step_modeimpute(all_predictors() & all_nominal())
# Define the bagged tree model
heart_bag = bag_tree(
  mode = "classification",
  cost_complexity = 0,
  tree_depth = NULL,
  min_n = 2,
  class_cost = NULL
) %>% set_engine(
  engine = "rpart",
  times = 100
)
# Define workflow
heart_bag_wf = workflow() %>%
  add_model(heart_bag) %>%
  add_recipe(heart_recipe)
# Fit/assess with CV
heart_bag_fit = heart_bag_wf %>% fit(heart_df)
```

... the fitted object automatically includes variable importance.

```
#> Bagged CART (classification with 100 members)
```

```
#>
```

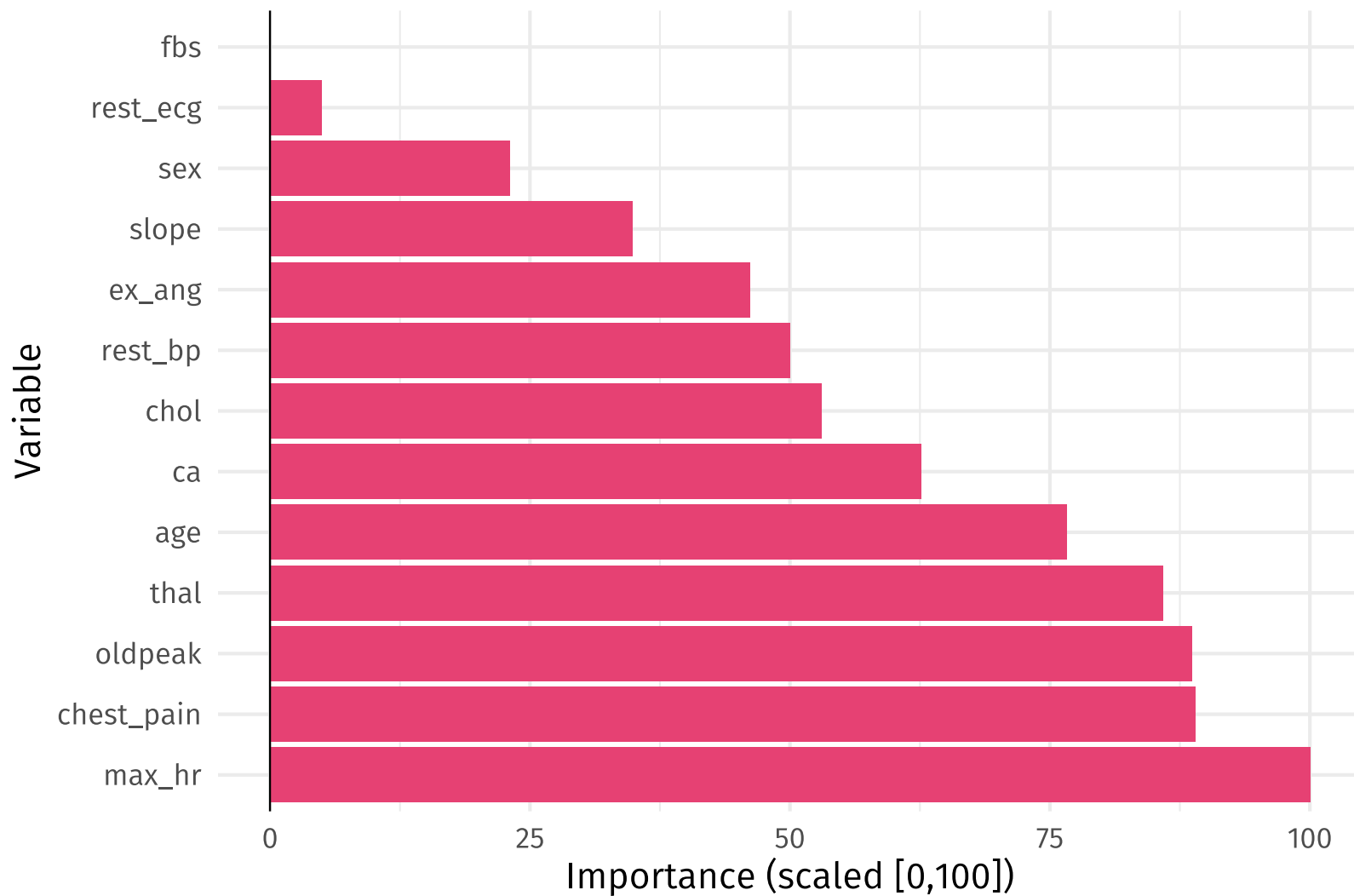
```
#> Variable importance scores include:
```

```
#>
```

```
#> # A tibble: 13 × 4
```

```
#>   term      value std.error  used
#>   <chr>    <dbl>    <dbl> <int>
#> 1 max_hr    41.1      0.839   100
#> 2 chest_pain 36.9      1.18    100
#> 3 oldpeak   36.8      0.806   100
#> 4 thal      35.7      1.35    100
#> 5 age       32.2      0.702   100
#> 6 ca        26.8      1.18    100
#> 7 chol      23.2      0.685   100
#> 8 rest_bp   22.1      0.589   100
#> 9 ex_ang    20.6      0.818    99
#> 10 slope    16.3      0.772   100
#> 11 sex      11.8      0.678   100
#> 12 rest_ecg  4.92      0.311    95
#> 13 fbs      3.02      0.210    93
```

Variable importance from our bagged tree model.



Ensemble methods

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.

Ensemble methods

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 .

Ensemble methods

Random forests

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

1. the **bootstrapped sample**
2. the m **randomly selected predictors** (for the split)

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 several **options** for training random forests with `tidymodels`. 🌲
E.g., `ranger`, `randomForest`, `spark`.

`rand_forest()` accesses each of these packages via their *engines*.

- The default engine is `"ranger"` (`ranger` package).
- The argument `mtry` gives m , the # of predictors at each split.

You've already seen the other hyperparameters for `ranger`:

- `trees` the number of trees in your (random) forest
- `min_n` min. # of observations

🌲 And even more if you look outside of `tidymodels`.

Ensemble methods

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

... and `ranger`

- Goal: Classification
- Three variables per split
- 100 trees in the forest
- At least 2 obs. to split
- Choose the `ranger` engine
- Set a `splitting rule`

```
# Define the random forest
heart_rf = rand_forest(
  mode = "classification",
  mtry = 3,
  trees = 100,
  min_n = 2
) %>% set_engine(
  engine = "ranger",
  splitrule = "gini"
)
```

Step 1: Define our parameter grid

```
# Define the parameter grid  
rf_grid = expand_grid(  
  mtry = 1:13,  
  min_n = 1:15  
)
```

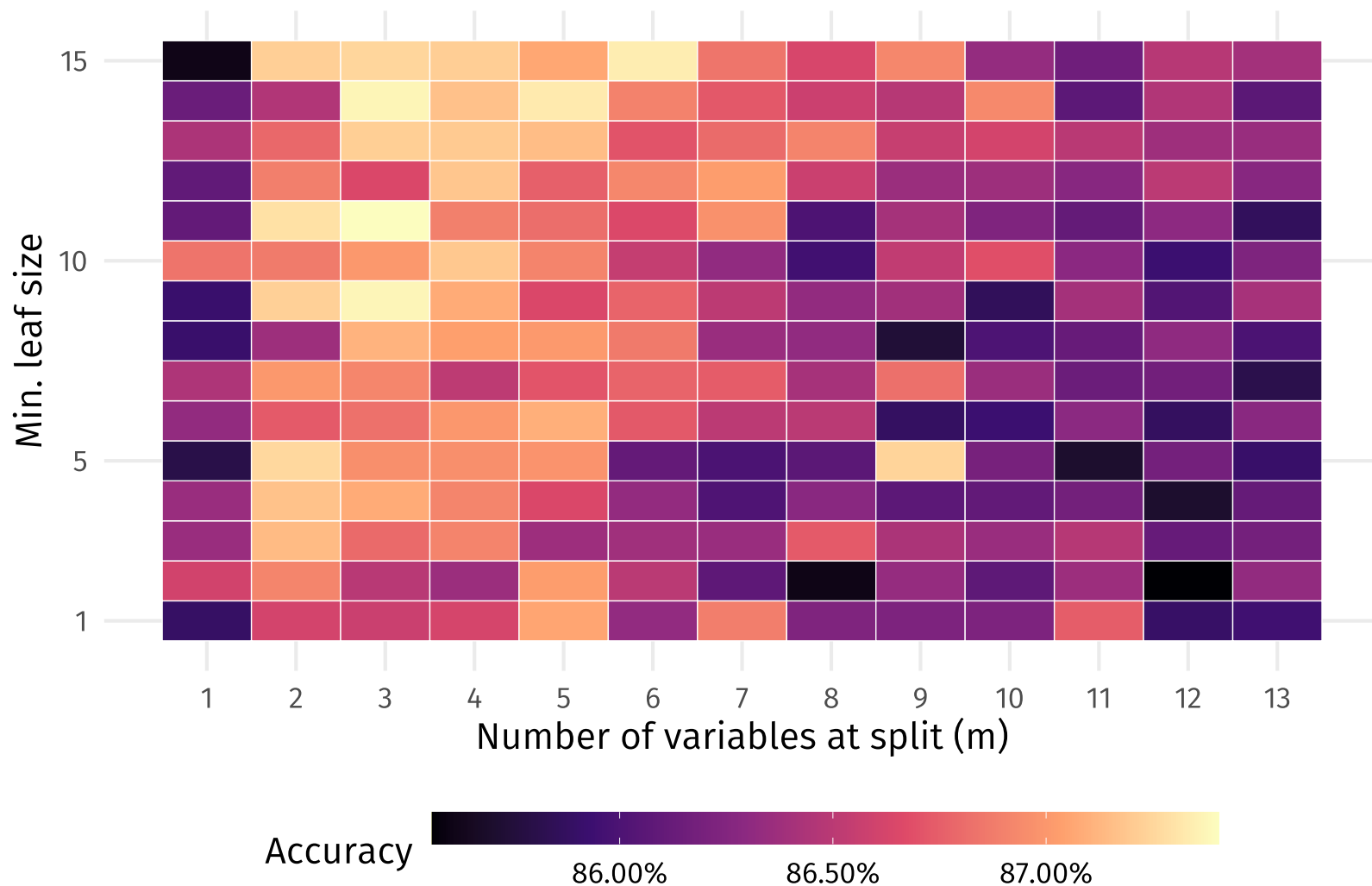
Step 2: Write a function that fits a RF using **given hyperparameters**.

```
# Function: One set of hyperparam
rf_i = function(i) {
  # Define the random forest
  heart_rf_i = rand_forest(
    mode = "classification",
    mtry = rf_grid$mtry[i],
    trees = 100,
    min_n = rf_grid$min_n[i]
  ) %>% set_engine(engine = "ranger", splitrule = "gini")
  # Define workflow
  heart_rf_wf_i =
    workflow() %>% add_model(heart_rf_i) %>% add_recipe(heart_recipe)
  # Fit
  heart_rf_fit_i = heart_rf_wf_i %>% fit(heart_df)
  # Return DF w/ OOB error and the hyperparameters
  tibble(
    mtry = rf_grid$mtry[i],
    min_n = rf_grid$min_n[i],
    # Note: OOB error is buried
    error_oob = heart_rf_fit_i$fit$fit$fit$prediction.error
  )
}
```

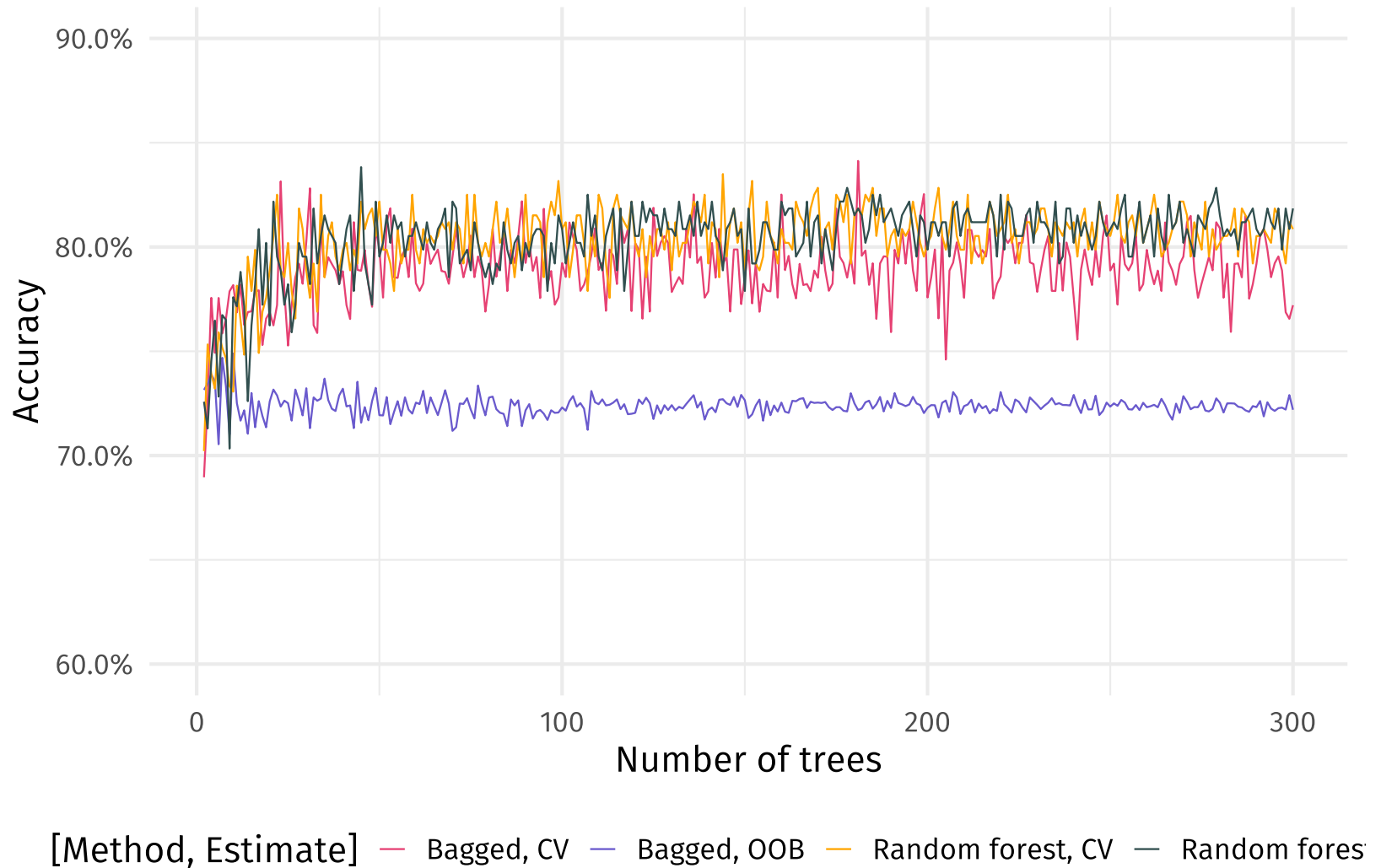
Step 3: Fit all of the forests (in `parallel`)!

```
# Fit the RFs on the grid
rf_tuning = mclapply(
  X = 1:nrow(rf_grid),
  FUN = rf_i,
  mc.cores = 12
) %>% rbindlist()
```


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 each other.

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).

Ensemble methods

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).
 - Individual trees are typically short—often $d = 1$ ("stumps").
 - *Remember* Trees learn from predecessors' mistakes, so no single tree needs to offer a perfect model.

Ensemble methods

How to boost

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

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

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

B. Update the model \hat{f} with "shrunk version" of new tree \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

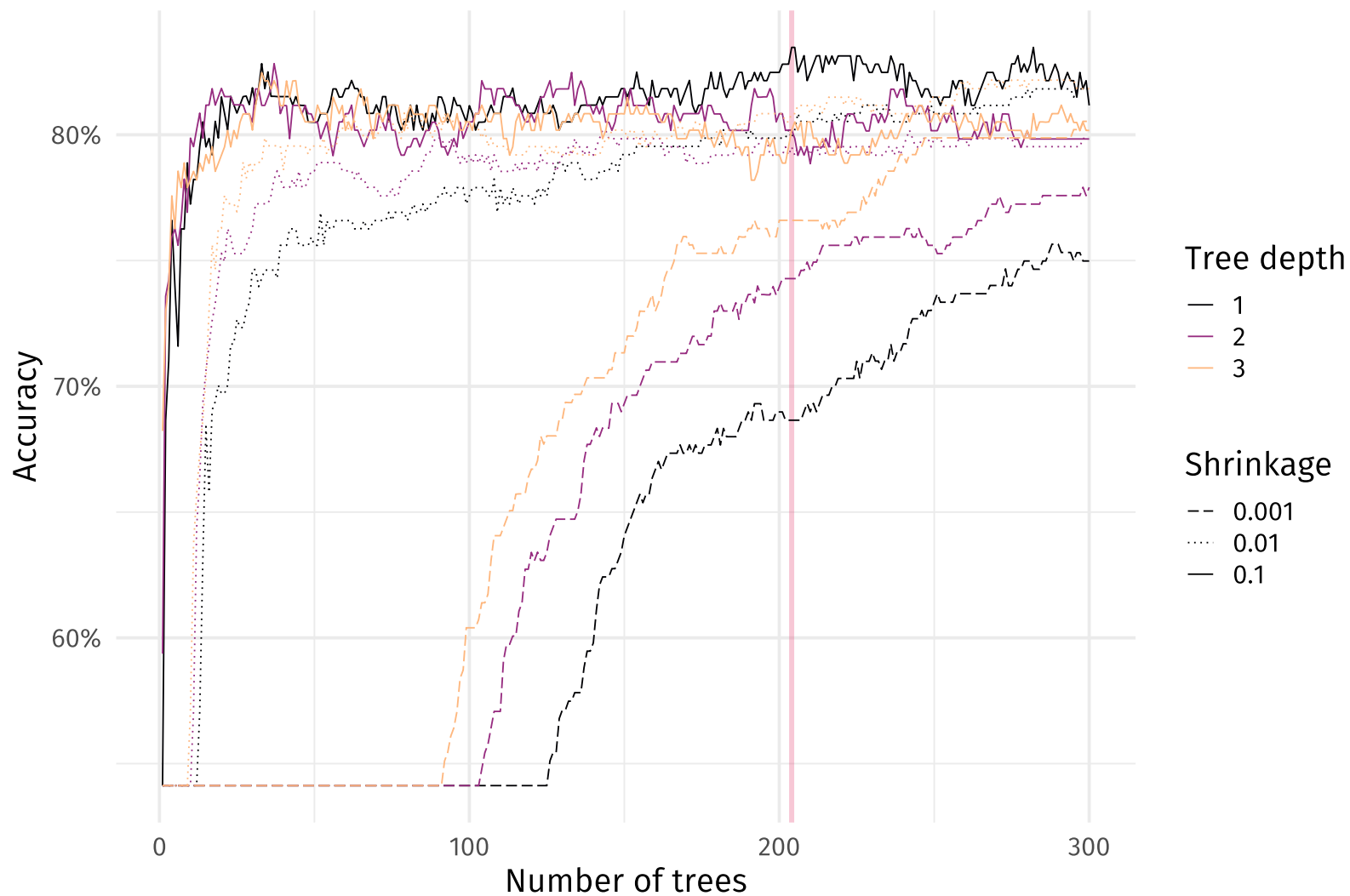
We will use `parsnips`'s `boost_tree()` to train boosted trees. 🌴

`boost_tree()` takes several parameters you've seen—plus one more:

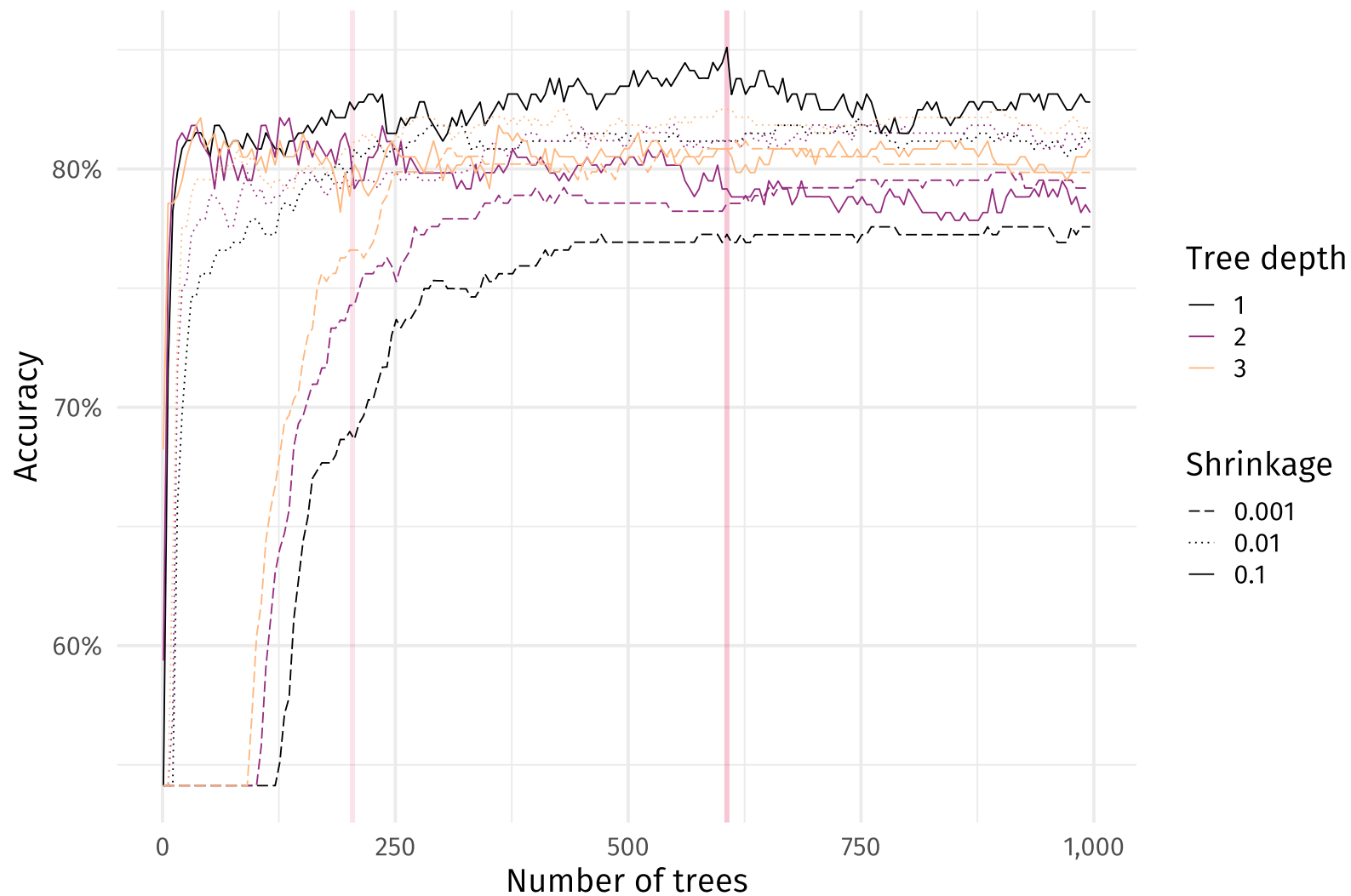
1. `mtry` number of predictors to try at each split
2. `trees`, the number of trees (B)
3. `min_n`, minimum observations to split
4. `tree_depth`, max. tree depth (max. splits from top)
5. `learn_rate`, the learning rate (λ)

🌴 This method uses the `xgboost` package.

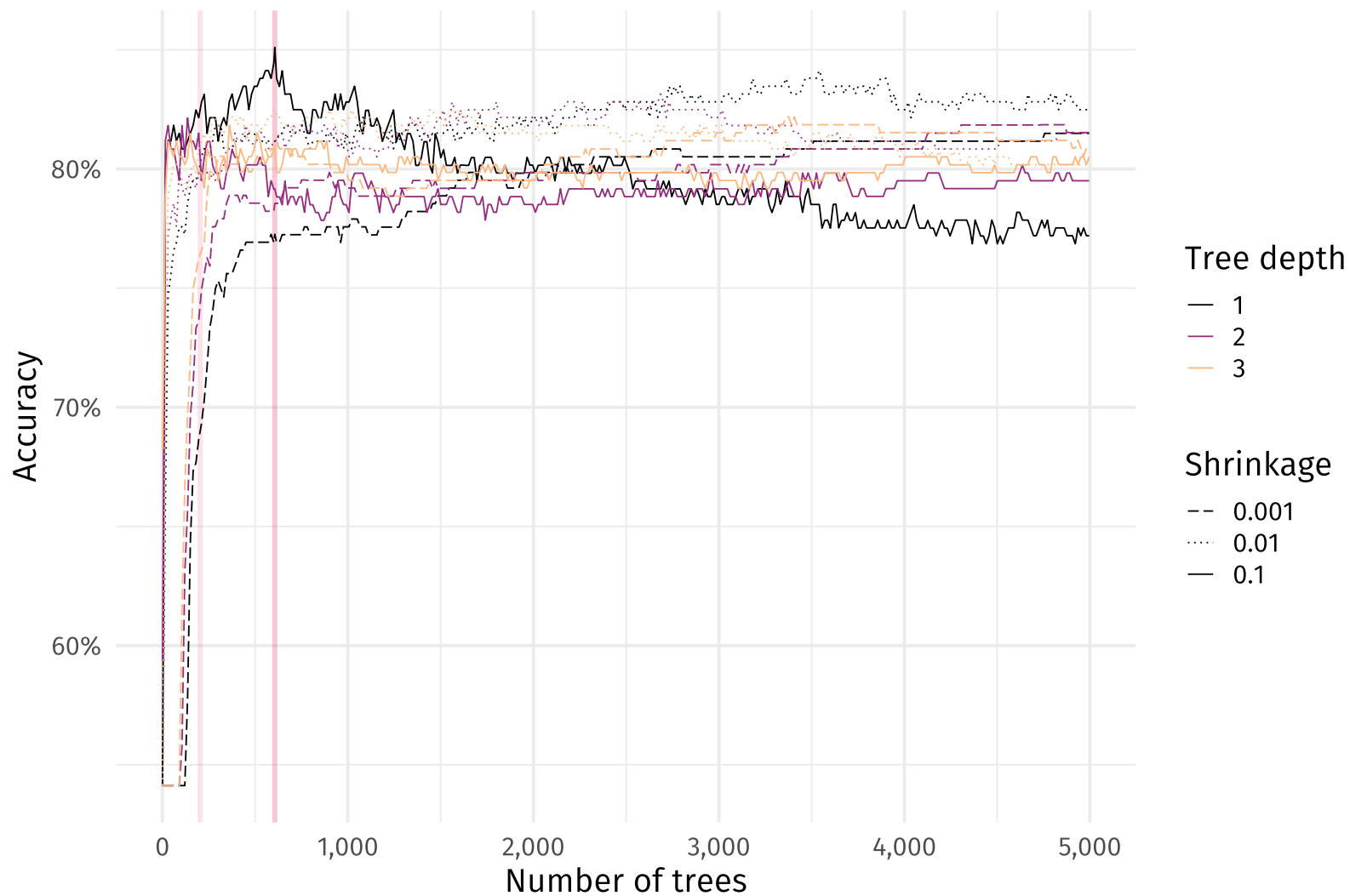
Comparing boosting parameters—notice the rates of learning



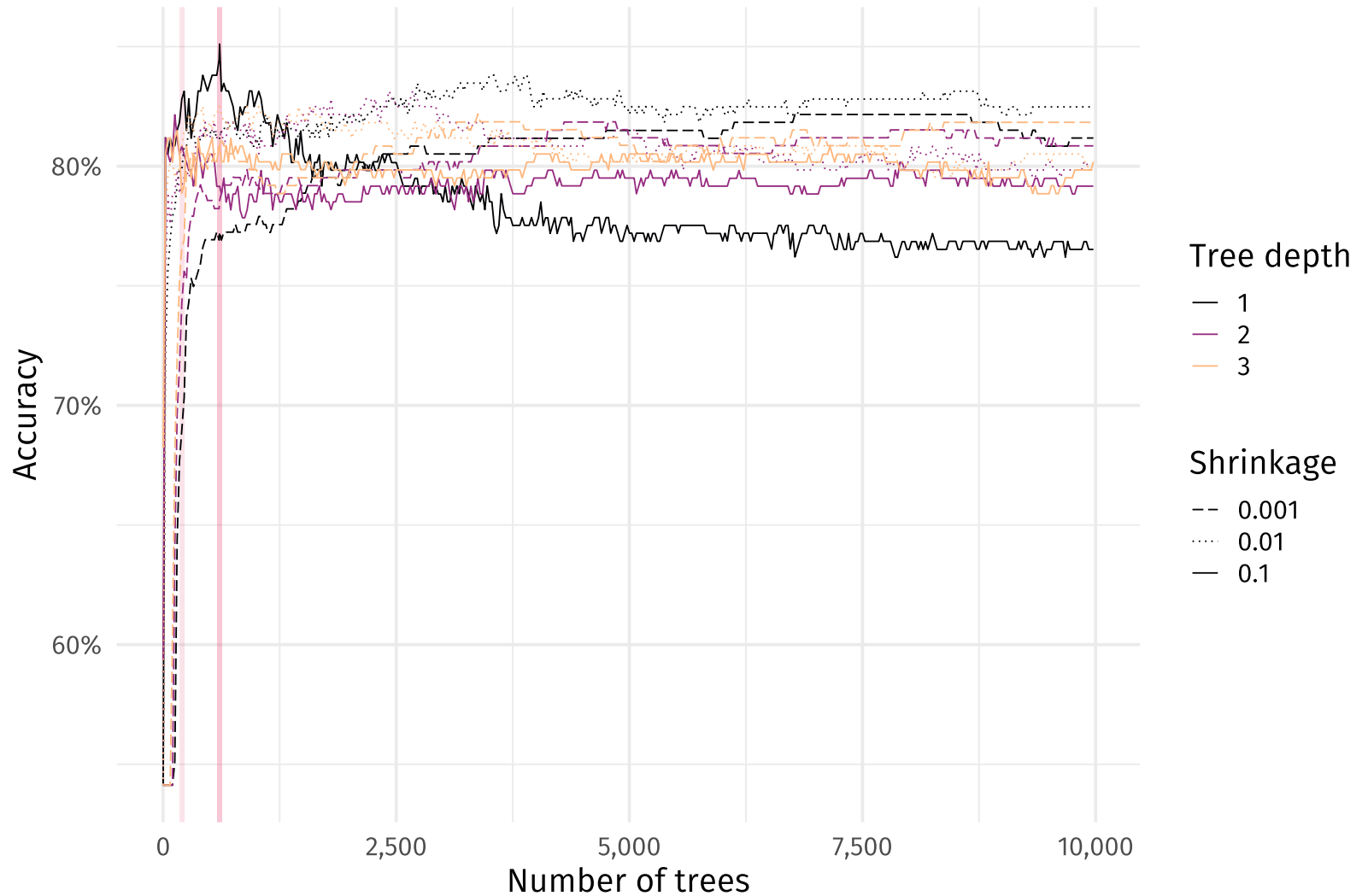
Comparing boosting parameters—more trees!



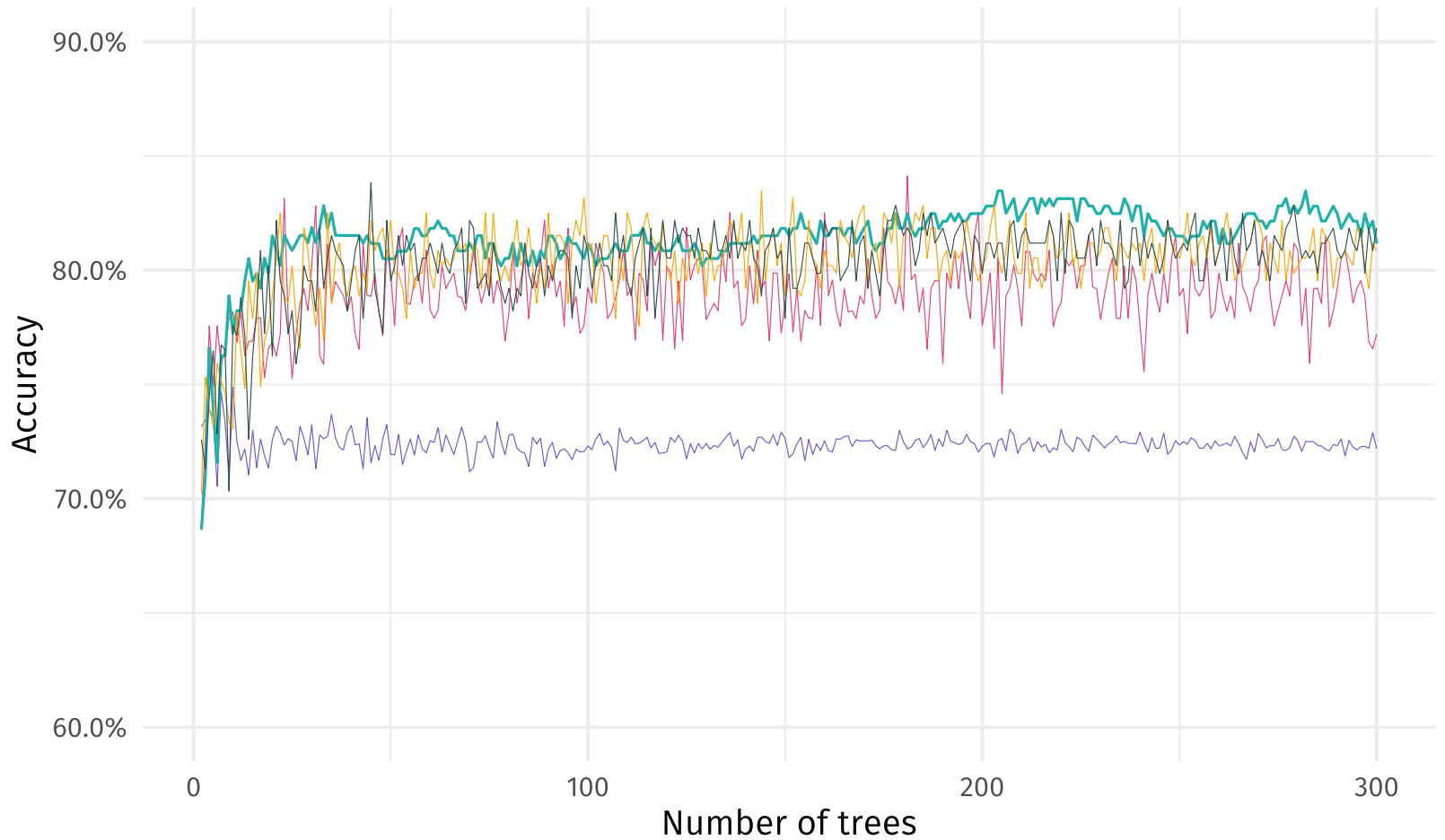
Comparing boosting parameters—even more trees!



Comparing boosting parameters—and even more trees!



Tree ensembles and the number of trees



[Method, Estimate] — Bagged, CV — Bagged, OOB — Boosted, CV — RF, CV — RF, OOB

Of course, there are a lot of other tree-based learning options:

- CatBoost (R)
- LightGBM (R)
- TabNet (R)

Sources

These notes draw upon

- [An Introduction to Statistical Learning \(ISL\)](#)
James, Witten, Hastie, and Tibshirani

Table of contents

Admin

- Today and upcoming

Decision trees

1. Fundamentals
2. Strengths and weaknesses

Other

- Sources/references

Ensemble methods

1. Introduction
2. Bagging
 - Introduction
 - Algorithm
 - Out-of-bag
 - In R
 - Variable importance
3. Random forests
 - Introduction
 - In R
4. Boosting
 - Introduction
 - Parameters
 - Algorithm
 - In R