

Tune model parameters

Last modified on November 19, 2025 10:21:11 Eastern Standard Time

Crediting the materials

The majority of the material in this document is taken from <https://www.tidymodels.org/start/models/>

INTRODUCTION

Some model parameters cannot be learned directly from a data set during model training; these kinds of parameters are called **hyperparameters**. Some examples of hyperparameters include the number of predictors that are sampled at splits in a tree-based model (we call this `mtry` in `tidymodels`) or the learning rate in a boosted tree model (we call this `learn_rate`). Instead of learning these kinds of hyperparameters during model training, we can estimate the best values for these values by training many models on resampled data sets and exploring how well all these models perform. This process is called **tuning**.

To use code in this article, you will need to install the following packages: `rpart`, `rpart.plot`, `tidymodels`, and `vip`.

Note

All of the required packages are already **installed** on the mathr server. You will need to **load** the packages using the `library()` function to access the data and functions in each package.

```
library(tidymodels) # for the tune package, along with the rest of tidymodels

# Helper packages
library(rpart.plot) # for visualizing a decision tree
library(vip)        # for variable importance plots
```

THE CELL IMAGE DATA, REVISITED

In our previous *Evaluate your model with resampling* article, we introduced a data set of images of cells that were labeled by experts as well-segmented (WS) or poorly segmented (PS). We trained a random forest model to predict which images are segmented well vs. poorly, so that a biologist could filter out poorly segmented cell images in their analysis. We used resampling to estimate the performance of our model on this data.

```
data(cells, package = "modeldata")
cells

# A tibble: 2,019 x 58
  case  class angle_ch_1 area_ch_1 avg_inten_ch_1 avg_inten_ch_2 avg_inten_ch_3
<fct> <fct>    <dbl>    <int>      <dbl>      <dbl>      <dbl>
1 Test  PS      143.      185      15.7       4.95      9.55
2 Train PS      134.      819      31.9      207.      69.9
3 Train WS      107.      431      28.0      116.      63.9
4 Train PS       69.2      298      19.5      102.      28.2
5 Test  PS       2.89      285      24.3      112.      20.5
6 Test  WS      40.7      172     326.      654.     129.
7 Test  WS     174.      177     260.      596.     124.
8 Test  PS     180.      251      18.3       5.73      17.2
9 Test  WS     18.9      495      16.1      89.5      13.7
10 Test WS     153.      384      17.7      89.9      20.4
# i 2,009 more rows
# i 51 more variables: avg_inten_ch_4 <dbl>, convex_hull_area_ratio_ch_1 <dbl>,
#   convex_hull_perim_ratio_ch_1 <dbl>, diff_inten_density_ch_1 <dbl>,
#   diff_inten_density_ch_3 <dbl>, diff_inten_density_ch_4 <dbl>,
#   entropy_inten_ch_1 <dbl>, entropy_inten_ch_3 <dbl>,
#   entropy_inten_ch_4 <dbl>, eq_circ_diam_ch_1 <dbl>,
#   eq_ellipse_lwr_ch_1 <dbl>, eq_ellipse_oblate_vol_ch_1 <dbl>, ...
```

PREDICTING IMAGE SEGMENTATION, BUT BETTER

Random forest models are a tree-based ensemble method, and typically perform well with [default hyperparameters](#). However, the accuracy of some other tree-based models, such as [boosted tree models](#) or [decision tree models](#), can be sensitive to the values of hyperparameters. In this article, we will train a **decision tree** model. There are several hyperparameters for decision tree models that can be tuned for better performance. Let's explore:

- the complexity parameter (which we call `cost_complexity` in `tidymodels`) for the tree, and

*the maximum `tree_depth`.

Tuning these hyperparameters can improve model performance because decision tree models are prone to [overfitting](#). This happens because single tree models tend to fit the training data *too well* — so well, in fact, that they over-learn patterns present in the training data that end up being detrimental when predicting new data.

We will tune the model hyperparameters to avoid overfitting. Tuning the value of `cost_complexity` helps by [pruning](#) back our tree. It adds a cost, or penalty, to error rates of more complex trees; a cost closer to zero decreases the number tree nodes pruned and is more likely to result in an overfit tree. However, a high cost increases the number of tree nodes pruned and can result in the opposite problem—an underfit tree. Tuning `tree_depth`, on the other hand, helps by [stopping](#) our tree from growing after it reaches a certain depth. We want to tune these hyperparameters to find what those two values should be for our model to do the best job predicting image segmentation.

Before we start the tuning process, we split our data into **training** and **testing** sets, just like when we trained the model with one default set of hyperparameters. As before, we can use `strata = class` if we want our training and testing sets to be created using stratified sampling so that both have the same proportion of both kinds of segmentation.

```
set.seed(123)
cell_split <- initial_split(cells |> select(-case),
                             strata = class)

cell_train <- training(cell_split)
cell_test  <- testing(cell_split)
```

We use the **training** data for tuning the model.

TUNING HYPERPARAMETERS

Let's start with the `parsnip` package, using a `decision_tree()` model with the [rpart](#) engine. To tune the decision tree hyperparameters `cost_complexity` and `tree_depth`, we create a model specification that identifies which hyperparameters we plan to tune.

```
tune_spec <-
  decision_tree(
    cost_complexity = tune(),
    tree_depth = tune()
  ) |>
  set_engine("rpart") |>
  set_mode("classification")
```

```
tune_spec
```

Decision Tree Model Specification (classification)

Main Arguments:

```
cost_complexity = tune()  
tree_depth = tune()
```

Computational engine: `rpart`

Think of `tune()` here as a placeholder. After the tuning process, we will select a single numeric value for each of these hyperparameters. For now, we specify our `parsnip` model object and identify the hyperparameters we will `tune()`.

We can't train this specification on a single data set (such as the entire training set) and learn what the hyperparameter values should be, but we can train many models using resampled data and see which models turn out best. We can create a regular grid of values to try using some convenience functions for each hyperparameter:

```
tree_grid <- grid_regular(cost_complexity(),  
                          tree_depth(),  
                          levels = 5)
```

The function `grid_regular()` is from the `dials` package. It chooses sensible values to try for each hyperparameter; here, we asked for 5 of each. Since we have two to tune, `grid_regular()` returns $5 \times 5 = 25$ different possible tuning combinations to try in a tidy tibble format.

```
tree_grid |>  
  kable()
```

cost_complexity	tree_depth
0.0000000	1
0.0000000	1
0.0000032	1
0.0005623	1
0.1000000	1
0.0000000	4
0.0000000	4
0.0000032	4
0.0005623	4
0.1000000	4

cost_complexity	tree_depth
0.0000000	8
0.0000000	8
0.0000032	8
0.0005623	8
0.1000000	8
0.0000000	11
0.0000000	11
0.0000032	11
0.0005623	11
0.1000000	11
0.0000000	15
0.0000000	15
0.0000032	15
0.0005623	15
0.1000000	15

Here, you can see all 5 values of `cost_complexity` ranging up to 0.1. These values get repeated for each of the 5 values of `tree_depth`:

```
tree_grid %>%
  count(tree_depth)
```

```
# A tibble: 5 x 2
  tree_depth     n
    <int> <int>
1         1     5
2         4     5
3         8     5
4        11     5
5        15     5
```

Armed with our grid filled with 25 candidate decision tree models, let's create cross-validation folds for tuning:

```
set.seed(234)
cell_folds <- vfold_cv(cell_train)
```

Tuning in `tidymodels` requires a resampled object created with the `rsample` package.

MODEL TUNING WITH A GRID

We are ready to tune! Let's use `tune_grid()` to fit models at all the different values we chose for each tuned hyperparameter. There are several options for building the object for tuning:

- Tune a model specification along with a recipe or model, or
- Tune a `workflow()` that bundles together a model specification and a recipe or model preprocessor.

Here we use a `workflow()` with a straightforward formula; if this model required more involved data preprocessing, we could use `add_recipe()` instead of `add_formula()`.

```
set.seed(345)

tree_wf <- workflow() |>
  add_model(tune_spec) |>
  add_formula(class ~ .)

tree_res <-
  tree_wf |>
  tune_grid(
    resamples = cell_folds,
    grid = tree_grid
  )

tree_res
```

Tuning results

10-fold cross-validation

A tibble: 10 x 4

	splits	id	.metrics	.notes
	<list>	<chr>	<list>	<list>
1	<split [1362/152]>	Fold01	<tibble [75 x 6]>	<tibble [0 x 4]>
2	<split [1362/152]>	Fold02	<tibble [75 x 6]>	<tibble [0 x 4]>
3	<split [1362/152]>	Fold03	<tibble [75 x 6]>	<tibble [0 x 4]>
4	<split [1362/152]>	Fold04	<tibble [75 x 6]>	<tibble [0 x 4]>
5	<split [1363/151]>	Fold05	<tibble [75 x 6]>	<tibble [0 x 4]>
6	<split [1363/151]>	Fold06	<tibble [75 x 6]>	<tibble [0 x 4]>
7	<split [1363/151]>	Fold07	<tibble [75 x 6]>	<tibble [0 x 4]>
8	<split [1363/151]>	Fold08	<tibble [75 x 6]>	<tibble [0 x 4]>
9	<split [1363/151]>	Fold09	<tibble [75 x 6]>	<tibble [0 x 4]>
10	<split [1363/151]>	Fold10	<tibble [75 x 6]>	<tibble [0 x 4]>

Once we have our tuning results, we can both explore them through visualization and then select the best result. The function `collect_metrics()` gives us a tidy tibble with all the results. We had 25 candidate models and two metrics, `accuracy` and `roc_auc`, and we get a row for each `.metric` and model.

```
tree_res |>
  collect_metrics() |>
  kable()
```

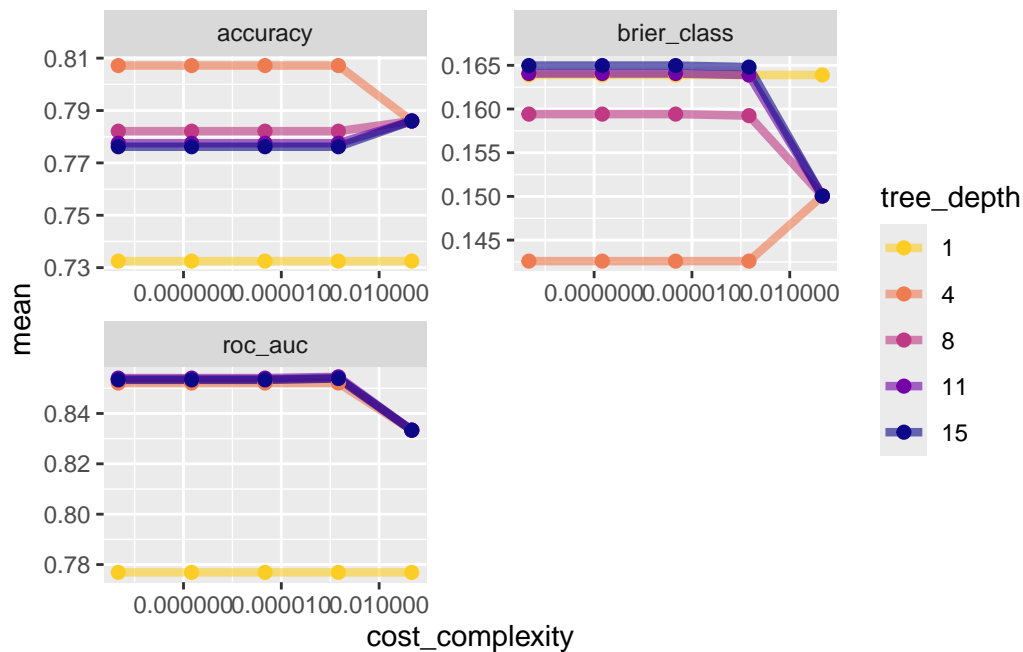
cost_complexity	tree_depth	.metric	.estimator	mean	n	std_err	.config
0.0000000	1	accuracy	binary	0.7324895	10	0.0147874	pre0_mod01_post0
0.0000000	1	brier_class	binary	0.1638940	10	0.0045544	pre0_mod01_post0
0.0000000	1	roc_auc	binary	0.7768876	10	0.0107360	pre0_mod01_post0
0.0000000	4	accuracy	binary	0.8071758	10	0.0119032	pre0_mod02_post0
0.0000000	4	brier_class	binary	0.1425954	10	0.0068123	pre0_mod02_post0
0.0000000	4	roc_auc	binary	0.8520807	10	0.0110442	pre0_mod02_post0
0.0000000	8	accuracy	binary	0.7821105	10	0.0121479	pre0_mod03_post0
0.0000000	8	brier_class	binary	0.1594185	10	0.0088417	pre0_mod03_post0
0.0000000	8	roc_auc	binary	0.8534851	10	0.0144354	pre0_mod03_post0
0.0000000	11	accuracy	binary	0.7774965	10	0.0132029	pre0_mod04_post0
0.0000000	11	brier_class	binary	0.1640622	10	0.0093940	pre0_mod04_post0
0.0000000	11	roc_auc	binary	0.8540427	10	0.0146328	pre0_mod04_post0
0.0000000	15	accuracy	binary	0.7761720	10	0.0132251	pre0_mod05_post0
0.0000000	15	brier_class	binary	0.1649736	10	0.0093745	pre0_mod05_post0
0.0000000	15	roc_auc	binary	0.8533505	10	0.0145442	pre0_mod05_post0
0.0000000	1	accuracy	binary	0.7324895	10	0.0147874	pre0_mod06_post0
0.0000000	1	brier_class	binary	0.1638940	10	0.0045544	pre0_mod06_post0
0.0000000	1	roc_auc	binary	0.7768876	10	0.0107360	pre0_mod06_post0
0.0000000	4	accuracy	binary	0.8071758	10	0.0119032	pre0_mod07_post0
0.0000000	4	brier_class	binary	0.1425954	10	0.0068123	pre0_mod07_post0
0.0000000	4	roc_auc	binary	0.8520807	10	0.0110442	pre0_mod07_post0
0.0000000	8	accuracy	binary	0.7821105	10	0.0121479	pre0_mod08_post0
0.0000000	8	brier_class	binary	0.1594185	10	0.0088417	pre0_mod08_post0
0.0000000	8	roc_auc	binary	0.8534851	10	0.0144354	pre0_mod08_post0
0.0000000	11	accuracy	binary	0.7774965	10	0.0132029	pre0_mod09_post0
0.0000000	11	brier_class	binary	0.1640622	10	0.0093940	pre0_mod09_post0
0.0000000	11	roc_auc	binary	0.8540427	10	0.0146328	pre0_mod09_post0
0.0000000	15	accuracy	binary	0.7761720	10	0.0132251	pre0_mod10_post0
0.0000000	15	brier_class	binary	0.1649736	10	0.0093745	pre0_mod10_post0
0.0000000	15	roc_auc	binary	0.8533505	10	0.0145442	pre0_mod10_post0
0.0000032	1	accuracy	binary	0.7324895	10	0.0147874	pre0_mod11_post0
0.0000032	1	brier_class	binary	0.1638940	10	0.0045544	pre0_mod11_post0

cost_complexity	tree_depth	.metric	.estimator	mean	n	std_err	.config
0.0000032	1	roc_auc	binary	0.7768876	10	0.0107360	pre0_mod11_post0
0.0000032	4	accuracy	binary	0.8071758	10	0.0119032	pre0_mod12_post0
0.0000032	4	brier_class	binary	0.1425954	10	0.0068123	pre0_mod12_post0
0.0000032	4	roc_auc	binary	0.8520807	10	0.0110442	pre0_mod12_post0
0.0000032	8	accuracy	binary	0.7821105	10	0.0121479	pre0_mod13_post0
0.0000032	8	brier_class	binary	0.1594185	10	0.0088417	pre0_mod13_post0
0.0000032	8	roc_auc	binary	0.8534851	10	0.0144354	pre0_mod13_post0
0.0000032	11	accuracy	binary	0.7774965	10	0.0132029	pre0_mod14_post0
0.0000032	11	brier_class	binary	0.1640622	10	0.0093940	pre0_mod14_post0
0.0000032	11	roc_auc	binary	0.8540427	10	0.0146328	pre0_mod14_post0
0.0000032	15	accuracy	binary	0.7761720	10	0.0132251	pre0_mod15_post0
0.0000032	15	brier_class	binary	0.1649736	10	0.0093745	pre0_mod15_post0
0.0000032	15	roc_auc	binary	0.8533505	10	0.0145442	pre0_mod15_post0
0.0005623	1	accuracy	binary	0.7324895	10	0.0147874	pre0_mod16_post0
0.0005623	1	brier_class	binary	0.1638940	10	0.0045544	pre0_mod16_post0
0.0005623	1	roc_auc	binary	0.7768876	10	0.0107360	pre0_mod16_post0
0.0005623	4	accuracy	binary	0.8071758	10	0.0119032	pre0_mod17_post0
0.0005623	4	brier_class	binary	0.1425954	10	0.0068123	pre0_mod17_post0
0.0005623	4	roc_auc	binary	0.8520807	10	0.0110442	pre0_mod17_post0
0.0005623	8	accuracy	binary	0.7821105	10	0.0121479	pre0_mod18_post0
0.0005623	8	brier_class	binary	0.1592420	10	0.0088655	pre0_mod18_post0
0.0005623	8	roc_auc	binary	0.8539733	10	0.0145191	pre0_mod18_post0
0.0005623	11	accuracy	binary	0.7774965	10	0.0132029	pre0_mod19_post0
0.0005623	11	brier_class	binary	0.1638858	10	0.0094199	pre0_mod19_post0
0.0005623	11	roc_auc	binary	0.8545493	10	0.0147226	pre0_mod19_post0
0.0005623	15	accuracy	binary	0.7761720	10	0.0132251	pre0_mod20_post0
0.0005623	15	brier_class	binary	0.1647972	10	0.0093904	pre0_mod20_post0
0.0005623	15	roc_auc	binary	0.8538940	10	0.0146192	pre0_mod20_post0
0.1000000	1	accuracy	binary	0.7324895	10	0.0147874	pre0_mod21_post0
0.1000000	1	brier_class	binary	0.1638940	10	0.0045544	pre0_mod21_post0
0.1000000	1	roc_auc	binary	0.7768876	10	0.0107360	pre0_mod21_post0
0.1000000	4	accuracy	binary	0.7860099	10	0.0123500	pre0_mod22_post0
0.1000000	4	brier_class	binary	0.1500544	10	0.0062346	pre0_mod22_post0
0.1000000	4	roc_auc	binary	0.8333631	10	0.0131428	pre0_mod22_post0
0.1000000	8	accuracy	binary	0.7860099	10	0.0123500	pre0_mod23_post0
0.1000000	8	brier_class	binary	0.1500544	10	0.0062346	pre0_mod23_post0
0.1000000	8	roc_auc	binary	0.8333631	10	0.0131428	pre0_mod23_post0
0.1000000	11	accuracy	binary	0.7860099	10	0.0123500	pre0_mod24_post0
0.1000000	11	brier_class	binary	0.1500544	10	0.0062346	pre0_mod24_post0
0.1000000	11	roc_auc	binary	0.8333631	10	0.0131428	pre0_mod24_post0
0.1000000	15	accuracy	binary	0.7860099	10	0.0123500	pre0_mod25_post0

cost_complexity	tree_depth	.metric	.estimator	mean	n	std_err	.config
0.1000000	15	brier_class	binary	0.1500544	10	0.0062346	pre0_mod25_post0
0.1000000	15	roc_auc	binary	0.8333631	10	0.0131428	pre0_mod25_post0

We might get more out of plotting these results:

```
tree_res |>
  collect_metrics() |>
  mutate(tree_depth = factor(tree_depth)) |>
  ggplot(aes(cost_complexity, mean, color = tree_depth)) +
  geom_line(linewidth = 1.5, alpha = 0.6) +
  geom_point(size = 2) +
  facet_wrap(~ .metric, scales = "free", nrow = 2) +
  scale_x_log10(labels = scales::label_number()) +
  scale_color_viridis_d(option = "plasma", begin = .9, end = 0)
```



We can see that our “stubbiest” tree, with a depth of 1, is the worst model according to both metrics and across all candidate values of `cost_complexity`. Our deepest tree, with a depth of 15, did better. However, the best tree seems to be between these values with a tree depth of 4. The `show_best()` function shows us the top 5 candidate models by default:

```
tree_res |>
  show_best(metric = "accuracy") |>
  kable()
```

cost_complexity	tree_depth	.metric	.estimator	mean	n	std_err	.config
0.0000000	4	accuracy	binary	0.8071758	10	0.0119032	pre0_mod02_post0
0.0000000	4	accuracy	binary	0.8071758	10	0.0119032	pre0_mod07_post0
0.0000032	4	accuracy	binary	0.8071758	10	0.0119032	pre0_mod12_post0
0.0005623	4	accuracy	binary	0.8071758	10	0.0119032	pre0_mod17_post0
0.1000000	4	accuracy	binary	0.7860099	10	0.0123500	pre0_mod22_post0

We can also use the `select_best()` function to pull out the single set of hyperparameter values for our best decision tree model:

```
best_tree <- tree_res |>
  select_best(metric = "accuracy")
best_tree |>
  kable()
```

cost_complexity	tree_depth	.config
0	4	pre0_mod02_post0

These are the values for `tree_depth` and `cost_complexity` that maximize accuracy in this data set of cell images.

FINALIZING OUR MODEL

We can update (or “finalize”) our workflow object `tree_wf` with the values from `select_best()`.

```
final_wf <-
  tree_wf |>
  finalize_workflow(best_tree)

final_wf
```

```
== Workflow =====
```

```
Preprocessor: Formula
Model: decision_tree()
```

```
-- Preprocessor -----
class ~ .
```

```
-- Model -----
Decision Tree Model Specification (classification)
```

```
Main Arguments:
  cost_complexity = 1e-10
  tree_depth = 4
```

```
Computational engine: rpart
```

Our tuning is done!

The last fit

Finally, let's fit this final model to the training data and use our test data to estimate the model performance we expect to see with new data. We can use the function `last_fit()` with our finalized model; this function *fits* the finalized model on the full training data set and *evaluates* the finalized model on the testing data.

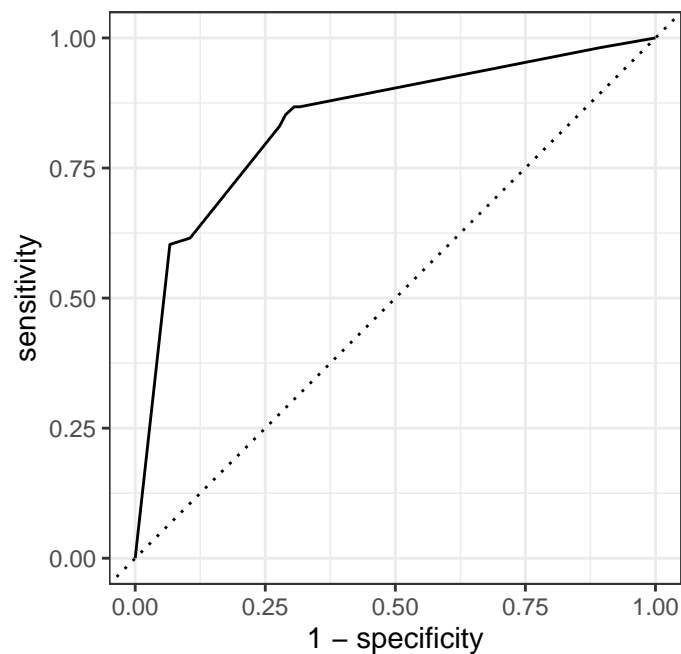
```
final_fit <-
  final_wf |>
  last_fit(cell_split)
```

```
final_fit |>
  collect_metrics()
```

```
# A tibble: 3 x 4
  .metric      .estimator .estimate .config
  <chr>        <chr>         <dbl> <chr>
1 accuracy    binary         0.802 pre0_mod0_post0
2 roc_auc     binary         0.840 pre0_mod0_post0
3 brier_class binary         0.148 pre0_mod0_post0
```

```
final_fit |>
  collect_predictions() |>
```

```
roc_curve(class, .pred_PS) |>
  autoplot()
```



The performance metrics from the test set indicate that we did not overfit during our tuning procedure.

The `final_fit` object contains a finalized, fitted workflow that you can use for predicting on new data or further understanding the results. You may want to extract this object, using [one of the `extract_` helper functions](#).

```
final_tree <- extract_workflow(final_fit)
final_tree
```

```
== Workflow [trained] =====
Preprocessor: Formula
Model: decision_tree()

-- Preprocessor -----
class ~ .

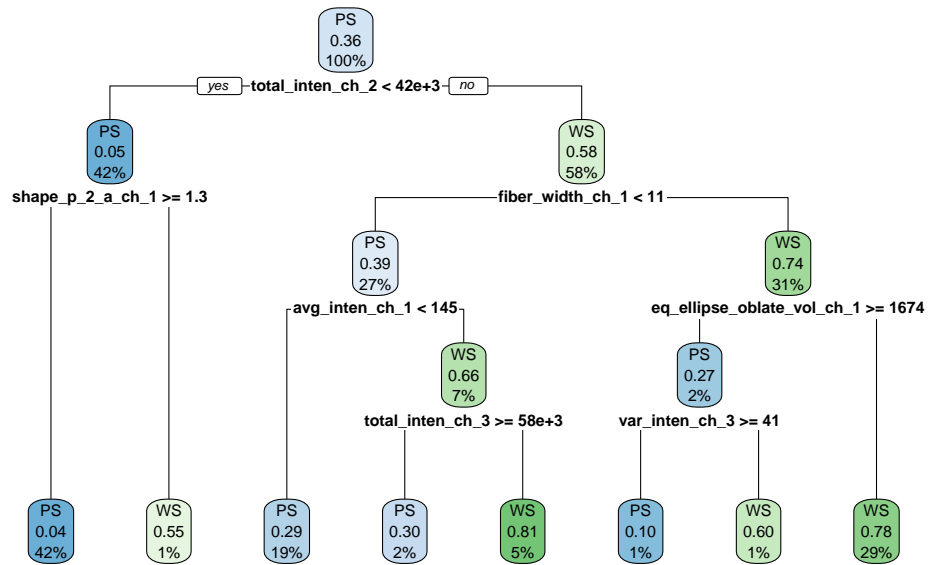
-- Model -----
n= 1514
```

```
node), split, n, loss, yval, (yprob)
  * denotes terminal node
```

```
1) root 1514 539 PS (0.64398943 0.35601057)
  2) total_inten_ch_2< 41732.5 642 33 PS (0.94859813 0.05140187)
    4) shape_p_2_a_ch_1>=1.251801 631 27 PS (0.95721078 0.04278922) *
    5) shape_p_2_a_ch_1< 1.251801 11 5 WS (0.45454545 0.54545455) *
  3) total_inten_ch_2>=41732.5 872 366 WS (0.41972477 0.58027523)
    6) fiber_width_ch_1< 11.37318 406 160 PS (0.60591133 0.39408867)
      12) avg_inten_ch_1< 145.4883 293 85 PS (0.70989761 0.29010239) *
      13) avg_inten_ch_1>=145.4883 113 38 WS (0.33628319 0.66371681)
        26) total_inten_ch_3>=57919.5 33 10 PS (0.69696970 0.30303030) *
        27) total_inten_ch_3< 57919.5 80 15 WS (0.18750000 0.81250000) *
    7) fiber_width_ch_1>=11.37318 466 120 WS (0.25751073 0.74248927)
      14) eq_ellipse_oblate_vol_ch_1>=1673.942 30 8 PS (0.73333333 0.26666667)
        28) var_inten_ch_3>=41.10858 20 2 PS (0.90000000 0.10000000) *
        29) var_inten_ch_3< 41.10858 10 4 WS (0.40000000 0.60000000) *
      15) eq_ellipse_oblate_vol_ch_1< 1673.942 436 98 WS (0.22477064 0.77522936) *
```

We can create a visualization of the decision tree using another helper function to extract the underlying engine-specific fit.

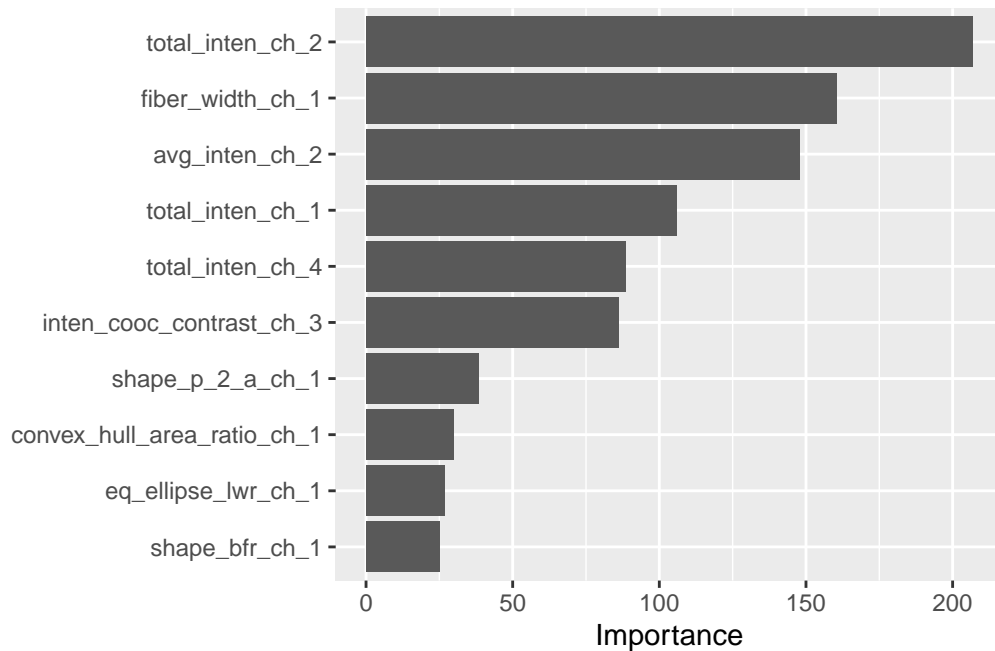
```
final_tree |>
  extract_fit_engine() |>
  rpart.plot(roundint = FALSE)
```



Perhaps we would also like to understand what variables are important in this final model. We can use the [vip](#) package to estimate variable importance [based on the model's structure](#).

```
library(vip)

final_tree |>
  extract_fit_parsnip() |>
  vip()
```



These are the automated image analysis measurements that are the most important in driving segmentation quality predictions.

We leave it to the reader to explore whether you can tune a different decision tree hyperparameter. You can explore the [reference docs](#), or use the `args()` function to see which `parsnip` object arguments are available:

```
args(decision_tree)
```

```
function (mode = "unknown", engine = "rpart", cost_complexity = NULL,
  tree_depth = NULL, min_n = NULL)
NULL
```

You could tune the other hyperparameter we didn't use here, `min_n`, which sets the minimum `n` to split at any node. This is another early stopping method for decision trees that can help prevent overfitting. Use this [searchable table](#) to find the original argument for `min_n` in the `rpart` package ([hint](#)). See whether you can tune a different combination of hyperparameters and/or values to improve a tree's ability to predict cell segmentation quality.

SESSION INFORMATION

R version 4.5.1 (2025-06-13)

Platform: x86_64-redhat-linux-gnu

Running under: Red Hat Enterprise Linux 9.6 (Plow)

Matrix products: default

BLAS/LAPACK: FlexiBLAS OPENBLAS-OPENMP; LAPACK version 3.9.0

locale:

```
[1] LC_CTYPE=en_US.UTF-8      LC_NUMERIC=C
[3] LC_TIME=en_US.UTF-8      LC_COLLATE=en_US.UTF-8
[5] LC_MONETARY=en_US.UTF-8  LC_MESSAGES=en_US.UTF-8
[7] LC_PAPER=en_US.UTF-8     LC_NAME=C
[9] LC_ADDRESS=C             LC_TELEPHONE=C
[11] LC_MEASUREMENT=en_US.UTF-8 LC_IDENTIFICATION=C
```

time zone: America/New_York

tzcode source: system (glibc)

attached base packages:

```
[1] stats      graphics  grDevices  utils      datasets  methods    base
```

other attached packages:

```
[1] vip_0.4.1          rpart.plot_3.1.3   rpart_4.1.24      yardstick_1.3.2
[5] workflowsets_1.1.1 workflows_1.3.0    tune_2.0.1        tailor_0.1.0
[9] rsample_1.3.1      recipes_1.3.1     parsnip_1.3.3     modeldata_1.5.1
[13] infer_1.0.9        dials_1.4.2       broom_1.0.10      tidymodels_1.4.1
[17] scales_1.4.0       lubridate_1.9.4    forcats_1.0.1     stringr_1.5.2
[21] dplyr_1.1.4        purrr_1.1.0       readr_2.1.5       tidyr_1.3.1
[25] tibble_3.3.0       ggplot2_4.0.0     tidyverse_2.0.0   knitr_1.50
```

loaded via a namespace (and not attached):

```
[1] tidyselect_1.2.1    viridisLite_0.4.2  timeDate_4051.111
[4] farver_2.1.2        S7_0.2.0           fastmap_1.2.0
[7] digest_0.6.37       timechange_0.3.0   lifecycle_1.0.4
[10] survival_3.8-3      magrittr_2.0.4     compiler_4.5.1
[13] rlang_1.1.6         tools_4.5.1        utf8_1.2.6
[16] yaml_2.3.10         data.table_1.17.8  labeling_0.4.3
[19] DiceDesign_1.10     RColorBrewer_1.1-3 withr_3.0.2
[22] nnet_7.3-20         grid_4.5.1         sparsevctrs_0.3.4
[25] future_1.67.0       iterators_1.0.14    globals_0.18.0
[28] MASS_7.3-65         tinytex_0.57       dichromat_2.0-0.1
[31] cli_3.6.5           rmarkdown_2.30     generics_0.1.4
[34] rstudioapi_0.17.1   future.apply_1.20.0 tzdb_0.5.0
[37] modelenv_0.2.0      splines_4.5.1      parallel_4.5.1
```


[40]	vctrs_0.6.5	hardhat_1.4.2	Matrix_1.7-4
[43]	jsonlite_2.0.0	hms_1.1.4	listenv_0.9.1
[46]	foreach_1.5.2	gower_1.0.2	glue_1.8.0
[49]	parallelly_1.45.1	codetools_0.2-20	stringi_1.8.7
[52]	gtable_0.3.6	GPfit_1.0-9	pillar_1.11.1
[55]	furrr_0.3.1	htmltools_0.5.8.1	ipred_0.9-15
[58]	lava_1.8.1	R6_2.6.1	lhs_1.2.0
[61]	evaluate_1.0.5	lattice_0.22-7	backports_1.5.0
[64]	class_7.3-23	Rcpp_1.1.0	prodlim_2025.04.28
[67]	xfun_0.53	pkgconfig_2.0.3	