Hyperparameters and Model Tuning

August 3, 2024

Table of contents


```
#install.packages("xgboost")
library(tidyverse)
library(tidymodels)
library(patchwork)
library(kableExtra)
library(modeldata)

tidymodels_prefer()

options(kable_styling_bootstrap_options = c("hover", "striped"))

#Set ggplot base theme
theme_set(theme_bw(base_size = 14))

ames <- read_csv("https://raw.githubusercontent.com/koalaverse/homlr/master/data/ames.csv")
```

Recap

In our most recent notebooks, we've gone beyond *Ordinary Least Squares* and explored additional classes of model. We began with *penalized* least squares models like Ridge Regression and the LASSO. We extended our knowledge of model classes to *nearest neighbor* and *tree-based*

models as well as *ensembles* of models in the previous notebook. We ended that notebook with a short discussion on parameter choices that must be made prior to model training – such parameters are known as *hyperparameters*. In this notebook, we learn how to use *cross-validation* to *tune* our model *hyperparameters*.

Objectives

In this notebook, we'll accomplish the following:

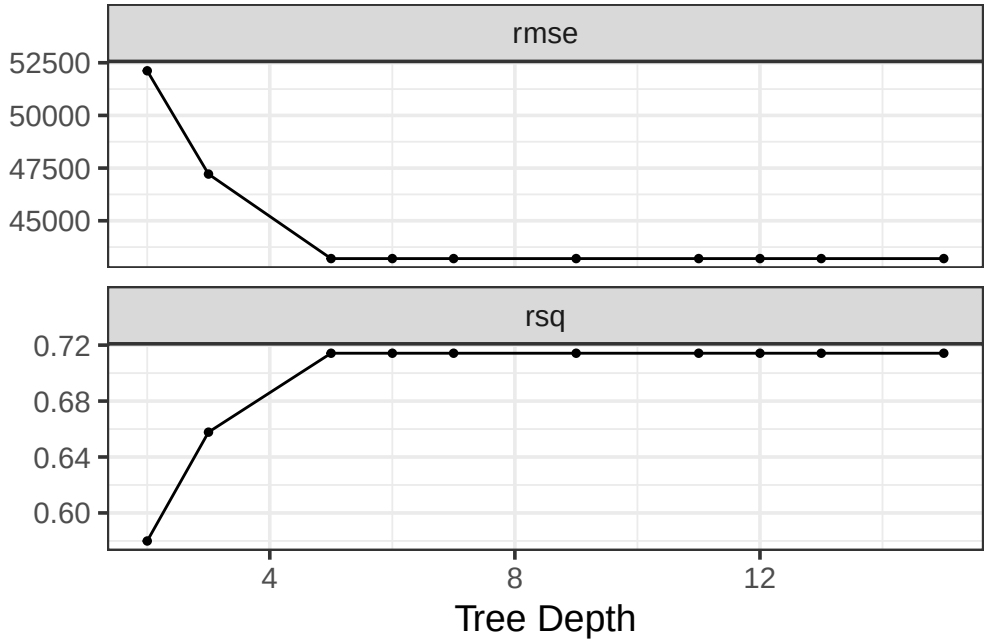
- Use `tune()` for model parameters as well as in feature engineering steps to identify hyperparameters that we want to tune through cross-validation.
- Use cross-validation and `tune_grid()` to tune the hyperparameters for a single model, identify the best hyperparameter choices, and fit the model using those best choices.
- Build a `workflow_set()`, choose hyperparameters that must be tuned for each model and recipe, use *cross-validation* to tune models and select "optimal" hyperparameter values, and compare the models in the workflow set.

Tuning Hyperparameters for a Single Model

Let's start with a decision tree model and we'll tune the tree depth parameter. We'll work with the `ames` data again for now.

```
ames_known_prices <- ames %>%
  filter(!is.na(Sale_Price))

ames_split <- initial_split(ames_known_prices, prop = 0.9)
ames_train <- training(ames_split)
ames_test <- testing(ames_split)

ames_folds <- vfold_cv(ames_train, v = 5)

tree_spec <- decision_tree(tree_depth = tune()) %>%
  set_engine("rpart") %>%
  set_mode("regression")

tree_rec <- recipe(Sale_Price ~ ., data = ames_train) %>%
  step_other(all_nominal_predictors()) %>%
  step_unknown(all_nominal_predictors()) %>%
  step_impute_median(all_numeric_predictors())

tree_wf <- workflow() %>%
```

```
  add_model(tree_spec) %>%
  add_recipe(tree_rec)

set.seed(123)
tree_results <- tree_wf %>%
  tune_grid(ames_folds, grid =12)

tree_results %>%
  autoplot()
```



We see from the plots above that deeper trees seemed to perform better than shallow trees. We don't observe much improvement in performance after a depth of 5. The risk of overfitting increases with deeper trees. We do seem to get some benefit by increasing the depth of the tree beyond 4. For this reason, I'll choose a tree depth of 5. The output of `show_best()` below shows our best-performing depths in terms of RMSE.

```
tree_results %>%
  show_best(n = 10) %>%
  kable() %>%
  kable_styling()
```

```
Warning in show_best(., n = 10): No value of `metric` was given; "rmse" will be
used.
```

tree_depth	.metric	.estimator	mean	n	std_err	.config
12	rmse	standard	43204.35	5	1936.036	Preprocessor1_Model01
7	rmse	standard	43204.35	5	1936.036	Preprocessor1_Model02
9	rmse	standard	43204.35	5	1936.036	Preprocessor1_Model03
11	rmse	standard	43204.35	5	1936.036	Preprocessor1_Model05
6	rmse	standard	43204.35	5	1936.036	Preprocessor1_Model06
15	rmse	standard	43204.35	5	1936.036	Preprocessor1_Model07
5	rmse	standard	43204.35	5	1936.036	Preprocessor1_Model08
13	rmse	standard	43204.35	5	1936.036	Preprocessor1_Model10
3	rmse	standard	47215.74	5	1988.689	Preprocessor1_Model04
2	rmse	standard	52117.48	5	2198.444	Preprocessor1_Model09

We can now build a final fit using this depth.

```
best_params <- tibble(tree_depth = 5)

tree_wf_final <- tree_wf %>%
  finalize_workflow(best_params)

tree_fit <- tree_wf_final %>%
  fit(ames_train)

tree_fit
```

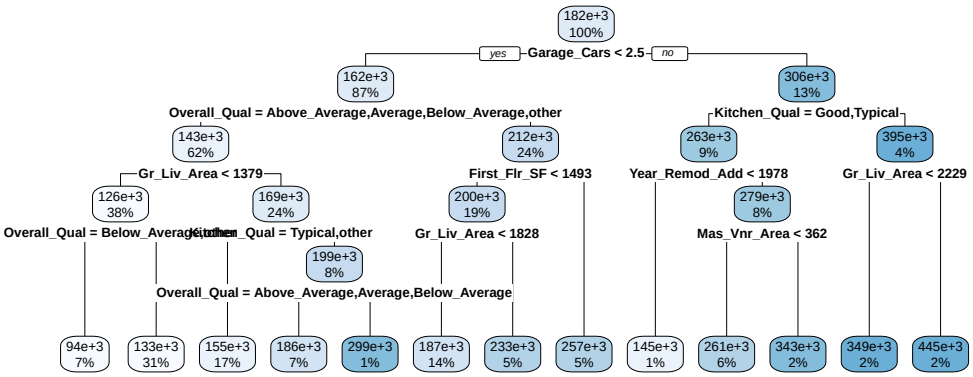
```
== Workflow [trained] ==========================================================
Preprocessor: Recipe
Model: decision_tree()

-- Preprocessor ----------------------------------------------------------------
3 Recipe Steps

* step_other()
* step_unknown()
* step_impute_median()

-- Model -----------------------------------------------------------------------
n= 2637

node), split, n, deviance, yval
      * denotes terminal node
```

```
 1) root 2637 1.699382e+13 181516.50
   2) Garage_Cars< 2.5 2283 6.808258e+12 162162.80
     4) Overall_Qual=Above_Average,Average,Below_Average,other 1643 3.041569e+12 142562.80
       8) Gr_Liv_Area< 1379 1000 8.960352e+11 125737.20
        16) Overall_Qual=Below_Average,other 192 1.678880e+11  93565.31 *
        17) Overall_Qual=Above_Average,Average 808 4.821998e+11 133382.00 *
       9) Gr_Liv_Area>=1379 643 1.422152e+12 168730.10
        18) Kitchen_Qual=Typical,other 441 5.277815e+11 154657.70 *
        19) Kitchen_Qual=Excellent,Good 202 6.163768e+11 199452.50
          38) Overall_Qual=Above_Average,Average,Below_Average 178 3.001805e+11 185965.90 *
          39) Overall_Qual=other 24 4.369547e+10 299478.50 *
     5) Overall_Qual=Good,Very_Good 640 1.515153e+12 212479.90
      10) First_Flr_SF< 1493 501 7.876153e+11 200153.10
        20) Gr_Liv_Area< 1827.5 361 3.555821e+11 187392.20 *
        21) Gr_Liv_Area>=1827.5 140 2.216658e+11 233058.00 *
      11) First_Flr_SF>=1493 139 3.770249e+11 256909.60 *
   3) Garage_Cars>=2.5 354 3.815558e+12 306331.40
     6) Kitchen_Qual=Good,Typical 238 1.339670e+12 263201.50
      12) Year_Remod_Add< 1977.5 28 4.028807e+10 145103.60 *
      13) Year_Remod_Add>=1977.5 210 8.567931e+11 278947.90
        26) Mas_Vnr_Area< 361.5 165 3.601444e+11 261488.10 *
        27) Mas_Vnr_Area>=361.5 45 2.619175e+11 342967.30 *
     7) Kitchen_Qual=Excellent 116 1.124816e+12 394822.00
      14) Gr_Liv_Area< 2229 61 1.702073e+11 349245.80 *
      15) Gr_Liv_Area>=2229 55 6.873694e+11 445370.10 *
```

We can see the tree as well, using `rpart.plot()`.

```
library(rpart.plot)

tree_fit_for_plot <- tree_fit %>%
  extract_fit_engine()

rpart.plot(tree_fit_for_plot, tweak = 1.5)
```

```
Warning: Cannot retrieve the data used to build the model (so cannot determine roundint and i
To silence this warning:
    Call rpart.plot with roundint=FALSE,
    or rebuild the rpart model with model=TRUE.
```



The model we built above can be interpreted and can also be utilized to make predictions on new data just like out previous models. Next, let's look at how we can tune multiple models with a variety of hyperparameters in a `workflow_set()`. We'll fit a *LASSO*, a *random forest*, and a *gradient boosted* model.

Tuning Hyperparameters Across a Workflow Set

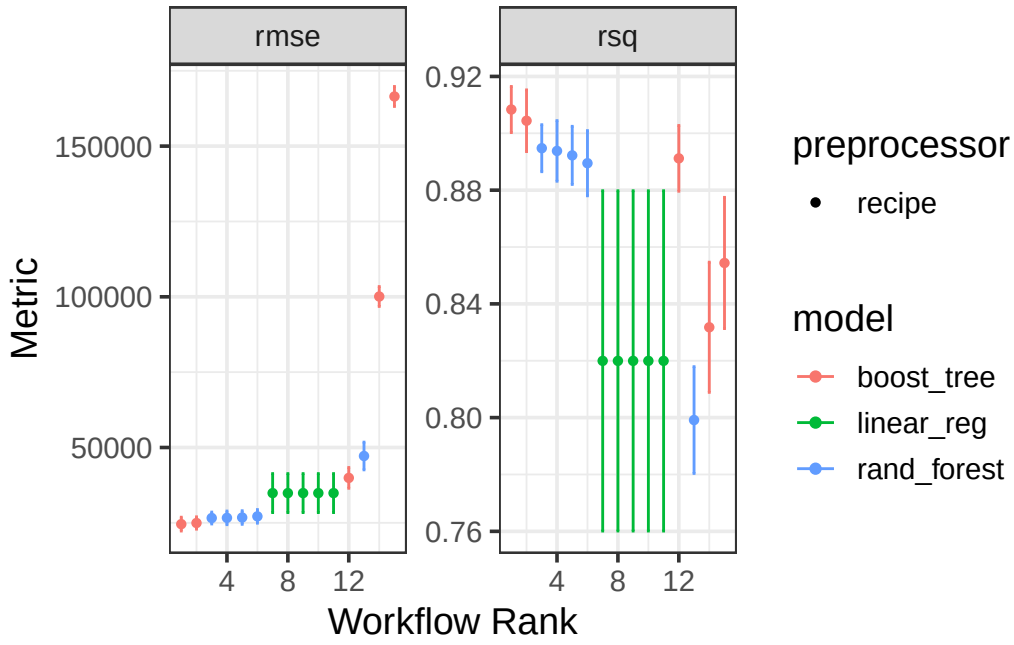
Let's create model specifications and recipes for each of the models mentioned in earlier notebooks.

```
doParallel::registerDoParallel()

lasso_spec <- linear_reg(penalty = tune(), mixture = 1) %>%
  set_engine("glmnet")

rf_spec <- rand_forest(mtry = tune(), trees = 100) %>%
  set_engine("ranger") %>%
  set_mode("regression")

gb_spec <- boost_tree(mtry = tune(), trees = 100, learn_rate = tune()) %>%
  set_engine("xgboost") %>%
  set_mode("regression")

rec <- recipe(Sale_Price ~ ., data = ames_train) %>%
  step_impute_knn(all_predictors()) %>%
  step_other(all_nominal_predictors(), threshold = 0.10) %>%
  step_dummy(all_nominal_predictors())

rec_list = list(rec = rec)
model_list = list(lasso = lasso_spec, rf = rf_spec, gb_tree = gb_spec)
```

```r
model_wfs <- workflow_set(rec_list, model_list, cross = TRUE)

grid_ctrl <- control_grid(
  save_pred = TRUE,
  parallel_over = "everything",
  save_workflow = TRUE
)

grid_results <- model_wfs %>%
  workflow_map(
    seed = 123,
    resamples = ames_folds,
    grid = 5,
    control = grid_ctrl)
```

```
i Creating pre-processing data to finalize unknown parameter: mtry
i Creating pre-processing data to finalize unknown parameter: mtry
```

```r
grid_results %>%
  autoplot()
```



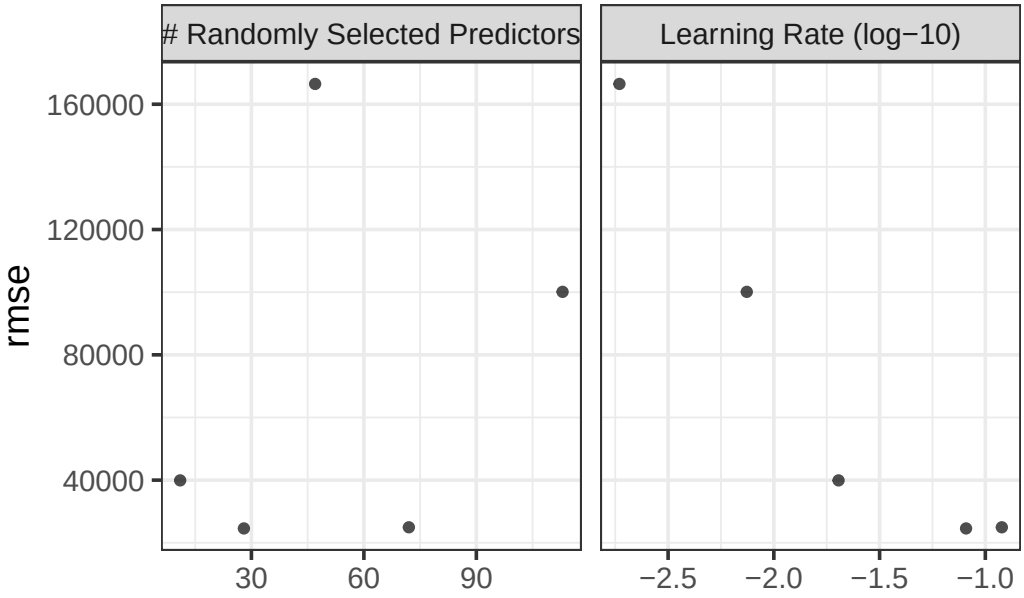
Now let's see what the best models were!

```
grid_results %>%
  rank_results() %>%
  kable() %>%
  kable_styling()
```

wflow_id	.config	.metric	mean	std_err	n	preprocessor	model
rec_gb_tree	Preprocessor1_Model3	rmse	2.457369e+04	1391.4521313	5	recipe	boost_tr
rec_gb_tree	Preprocessor1_Model3	rsq	9.083623e-01	0.0049237	5	recipe	boost_tr
rec_gb_tree	Preprocessor1_Model2	rmse	2.494505e+04	1242.5126311	5	recipe	boost_tr
rec_gb_tree	Preprocessor1_Model2	rsq	9.044156e-01	0.0065949	5	recipe	boost_tr
rec_rf	Preprocessor1_Model5	rmse	2.659544e+04	1195.5022689	5	recipe	rand_for
rec_rf	Preprocessor1_Model5	rsq	8.947585e-01	0.0050227	5	recipe	rand_for
rec_rf	Preprocessor1_Model3	rmse	2.665282e+04	1371.2899968	5	recipe	rand_for
rec_rf	Preprocessor1_Model3	rsq	8.937991e-01	0.0064809	5	recipe	rand_for
rec_rf	Preprocessor1_Model1	rmse	2.676280e+04	1373.1462777	5	recipe	rand_for
rec_rf	Preprocessor1_Model1	rsq	8.922302e-01	0.0062129	5	recipe	rand_for
rec_rf	Preprocessor1_Model2	rmse	2.714602e+04	1370.7725823	5	recipe	rand_for
rec_rf	Preprocessor1_Model2	rsq	8.894834e-01	0.0069672	5	recipe	rand_for
rec_lasso	Preprocessor1_Model5	rmse	3.487079e+04	3930.7864113	5	recipe	linear_re
rec_lasso	Preprocessor1_Model5	rsq	8.199507e-01	0.0363800	5	recipe	linear_re
rec_lasso	Preprocessor1_Model1	rmse	3.487079e+04	3930.7864113	5	recipe	linear_re
rec_lasso	Preprocessor1_Model1	rsq	8.199507e-01	0.0363800	5	recipe	linear_re
rec_lasso	Preprocessor1_Model2	rmse	3.487079e+04	3930.7864113	5	recipe	linear_re
rec_lasso	Preprocessor1_Model2	rsq	8.199507e-01	0.0363800	5	recipe	linear_re
rec_lasso	Preprocessor1_Model3	rmse	3.487079e+04	3930.7864113	5	recipe	linear_re
rec_lasso	Preprocessor1_Model3	rsq	8.199507e-01	0.0363800	5	recipe	linear_re
rec_lasso	Preprocessor1_Model4	rmse	3.487079e+04	3930.7864113	5	recipe	linear_re
rec_lasso	Preprocessor1_Model4	rsq	8.199507e-01	0.0363800	5	recipe	linear_re
rec_gb_tree	Preprocessor1_Model1	rmse	3.991434e+04	2093.1941989	5	recipe	boost_tr
rec_gb_tree	Preprocessor1_Model1	rsq	8.911876e-01	0.0070286	5	recipe	boost_tr
rec_rf	Preprocessor1_Model4	rmse	4.717707e+04	2793.5954271	5	recipe	rand_for
rec_rf	Preprocessor1_Model4	rsq	7.991412e-01	0.0114148	5	recipe	rand_for
rec_gb_tree	Preprocessor1_Model5	rmse	1.001042e+05	2003.4675900	5	recipe	boost_tr
rec_gb_tree	Preprocessor1_Model5	rsq	8.317716e-01	0.0139171	5	recipe	boost_tr
rec_gb_tree	Preprocessor1_Model4	rmse	1.664788e+05	2009.9023878	5	recipe	boost_tr
rec_gb_tree	Preprocessor1_Model4	rsq	8.543906e-01	0.0140263	5	recipe	boost_tr

The model performing the best was the *gradient boosted tree ensemble*. Let's see what hyper-parameter choices led to the best performance.

```
grid_results %>%
  autoplot(metric = "rmse", id = "rec_gb_tree")
```



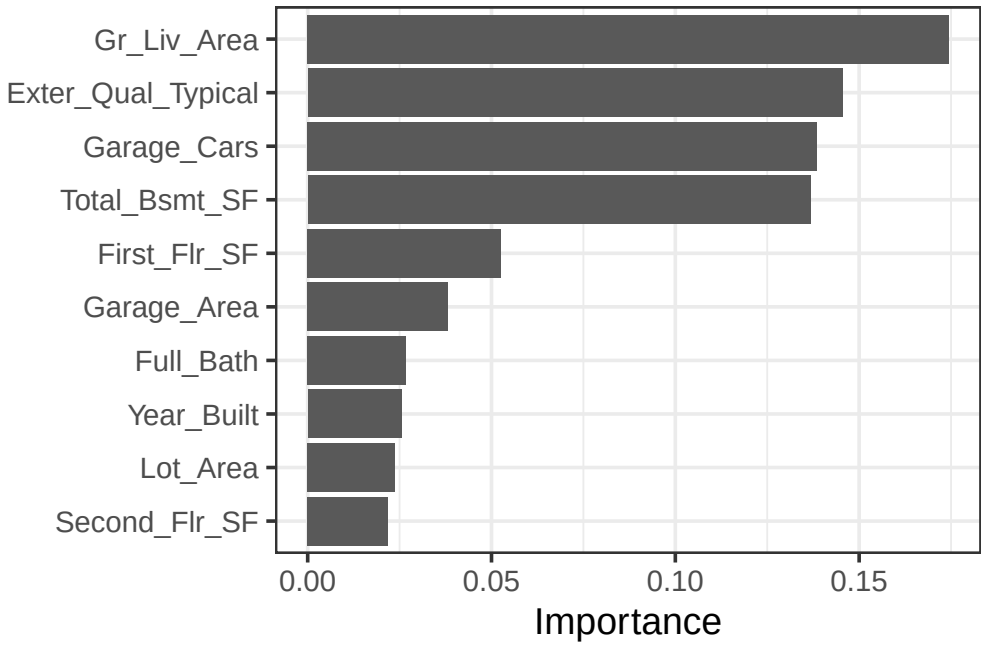
It seems that a number of randomly selected parameters of near 30 gave the best performance and learning rates near 0.1 did as well. We'll construct this model and fit it to our training data.

```
set.seed(123)
gb_tree_spec <- boost_tree(mtry = 30, trees = 100, learn_rate = 0.1) %>%
  set_engine("xgboost") %>%
  set_mode("regression")

gb_tree_wf <- workflow() %>%
  add_model(gb_tree_spec) %>%
  add_recipe(rec)

gb_tree_fit <- gb_tree_wf %>%
  fit(ames_train)
```

Such a model doesn't have much interpretive value but can make very good predictions. We can identify the predictors which were most important within the ensemble by using `var_imp()`.

```
library(vip)
gb_tree_fit %>%
```

```
  extract_fit_engine() %>%
  vip()
```



From the plot above, we can see the features that were most the important predictors of *selling price* within the ensemble. Note that the important predictors will shuffle around slightly each time you re-run the ensemble. Before we close this notebook, let's take a look at how well this model predicts the selling prices of homes in our *test* set.

```
gb_results <- gb_tree_fit %>%
  augment(ames_test) %>%
  select(Sale_Price, .pred) %>%
  rmse(Sale_Price, .pred)

gb_results %>%
  kable() %>%
  kable_styling(bootstrap_options = c("striped", "hover"))
```

.metric	.estimator	.estimate
rmse	standard	18669.21

This final ensemble of models predicted selling prices of homes with an root mean squared error of $ 18,669.21.

Summary

In this notebook, we saw how to build a workflow set consisting of several models with tunable hyperparameters. We explored a *space-filling grid* of hyperparameter combinations with a `workflow_map()`. After identifying a best model and optimal(*) hyperparameter choices, we fit the corresponding model to our training data and then assessed that model's performance on our test data.