Lab 7: Tree-Based Methods

PSTAT 131/231

Contents

Introduction

This lab will take a look at different tree-based models. In doing so, we will explore how changing hyperparameters can help improve performance. This chapter will use parsnip for model fitting and recipes and workflows to perform the transformations, and tune and dials to tune the hyperparameters of the model. `rpart.plot` is used to visualize the decision trees, using the `rpart` package as an engine, and `vip` is used to visualize variable importance.

Loading Packages

```
library(tidyverse)
library(tidymodels)
library(ISLR)
library(rpart.plot)
library(vip)
library(janitor)
library(randomForest)
library(xgboost)
```

Data

We will build a regression model using the `ames` data to predict the sale price, `sale_price`, as the response with the remaining variables as predictors.

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

ames <- as_tibble(ames) %>%
  clean_names()
```

We will also use the `Carseats` data set from the `ISLR` package to demonstrate a classification model. We create a new variable `High` to denote if `Sales <= 8`, then the `Sales` predictor is removed as it is a perfect predictor of `High`.

```
Carseats <- as_tibble(Carseats) %>%
  mutate(High = factor(if_else(Sales <= 8, "No", "Yes"))) %>%
  select(-Sales)
```

Activities

- Access the help page for `ames`. Familiarize yourself with the subject of the data set and the predictor variables.

The Initial Split

We can start, as normal, by splitting the data sets into training and testing sets, using stratified sampling.

```
set.seed(3435)
ames_split <- initial_split(ames, strata = "sale_price")

ames_train <- training(ames_split)
ames_test <- testing(ames_split)

Carseats_split <- initial_split(Carseats, strata = "High")

Carseats_train <- training(Carseats_split)
Carseats_test <- testing(Carseats_split)
```

Fitting Classification Trees

We will both be fitting a classification and regression tree in this section, so we can save a little bit of typing by creating a general decision tree specification using `rpart` as the engine.

```
tree_spec <- decision_tree() %>%
  set_engine("rpart")
```

Then this decision tree specification can be used to create a classification decision tree engine. This is a good example of how the flexible composition system created by `parsnip` can be used to create multiple model specifications.

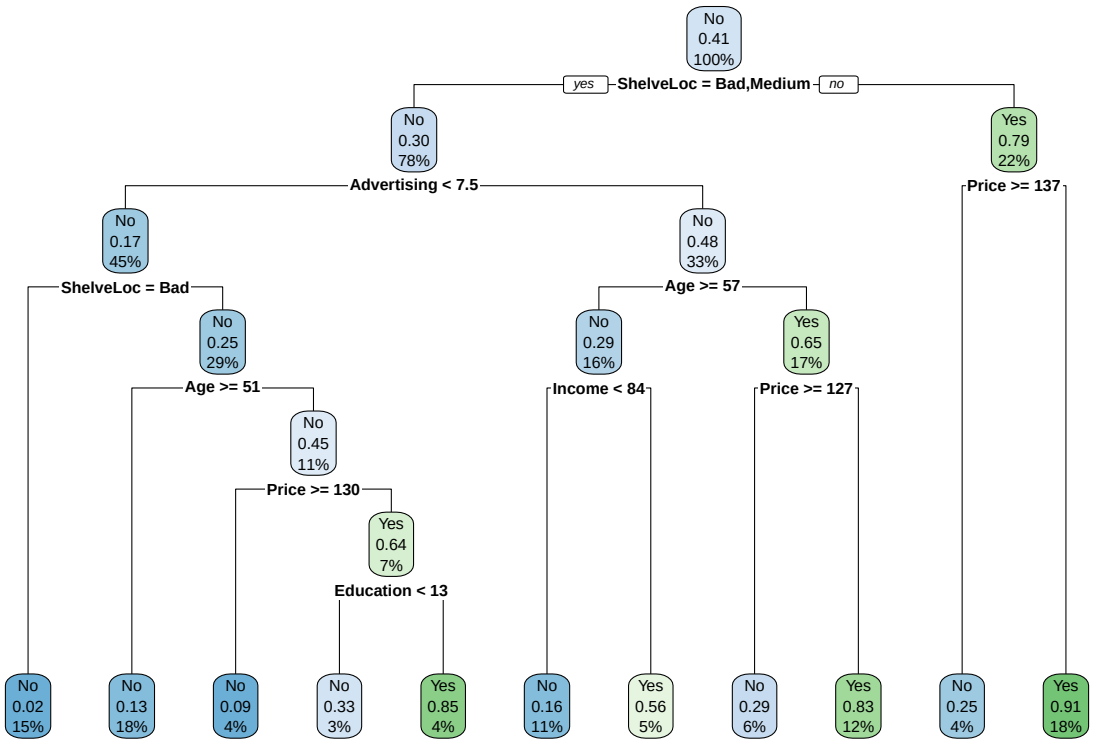
```
class_tree_spec <- tree_spec %>%
  set_mode("classification")
```

With both a model specification and our data, we are ready to fit the model.

```
class_tree_fit <- class_tree_spec %>%
  fit(High ~ ., data = Carseats_train)
```

The `rpart.plot` package provides functions to let us easily visualize the decision tree. As the name implies, it only works with `rpart` trees.

```
class_tree_fit %>%
  extract_fit_engine() %>%
  rpart.plot()
```



Activities

- From the visualization, what is the most important predictor of shelving location? How do you know?

We can check the training set accuracy of this model:

```
augment(class_tree_fit, new_data = Carseats_train) %>%
  accuracy(truth = High, estimate = .pred_class)
```

```
## # A tibble: 1 x 3
##   .metric  .estimator .estimate
##   <chr>    <chr>          <dbl>
## 1 accuracy binary         0.85
```

And look at the confusion matrix:

```
augment(class_tree_fit, new_data = Carseats_train) %>%
  conf_mat(truth = High, estimate = .pred_class)
```

```
##           Truth
## Prediction  No Yes
##        No  157  25
##        Yes  20  98
```

Let's check the testing set:

```
augment(class_tree_fit, new_data = Carseats_test) %>%
  conf_mat(truth = High, estimate = .pred_class)
```

```
##           Truth
## Prediction No Yes
##        No  44  16
##        Yes 15  25
```

```
augment(class_tree_fit, new_data = Carseats_test) %>%
  accuracy(truth = High, estimate = .pred_class)
```

```
## # A tibble: 1 x 3
##   .metric  .estimator .estimate
##   <chr>    <chr>          <dbl>
## 1 accuracy binary          0.69
```

Activities

- Compare the model's performance on the training set to its performance on the testing set.

Let us try to tune the `cost_complexity` of the decision tree, or the pruning penalty, to find a more optimal complexity. We use the `class_tree_spec` object and use the `set_args()` function to specify that we want to tune `cost_complexity`. This is then passed directly into the `workflow` object to avoid creating an intermediate object.

```
class_tree_wf <- workflow() %>%
  add_model(class_tree_spec %>% set_args(cost_complexity = tune())) %>%
  add_formula(High ~ .)
```

To be able to tune the hyperparameter, we need 2 more objects – a `resamples` object (we will use a k-fold cross-validation data set), and a grid of values to try. Since we are only tuning one hyperparameter, a regular grid is probably simplest.

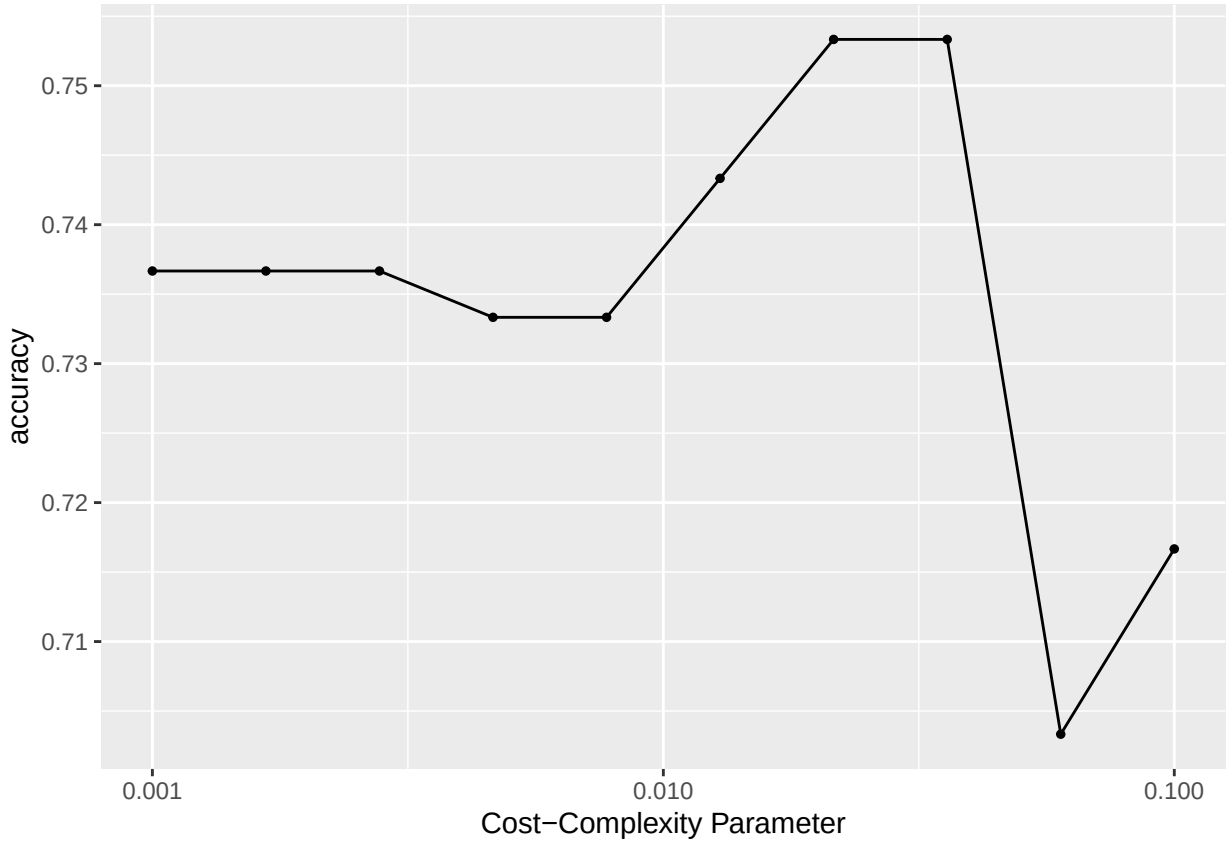
```
set.seed(3435)
Carseats_fold <- vfold_cv(Carseats_train)

param_grid <- grid_regular(cost_complexity(range = c(-3, -1)), levels = 10)
```

```
tune_res <- tune_grid(
  class_tree_wf,
  resamples = Carseats_fold,
  grid = param_grid,
  metrics = metric_set(accuracy)
)
```

Using `autoplot()` shows which values of cost_complexity appear to produce the highest accuracy:

```
autoplot(tune_res)
```



Activities

- Interpret the plot. What do you notice?

We can now select the best performing value with `select_best()`, finalize the workflow by updating the value of `cost_complexity`, and fit the model on the full training data set.

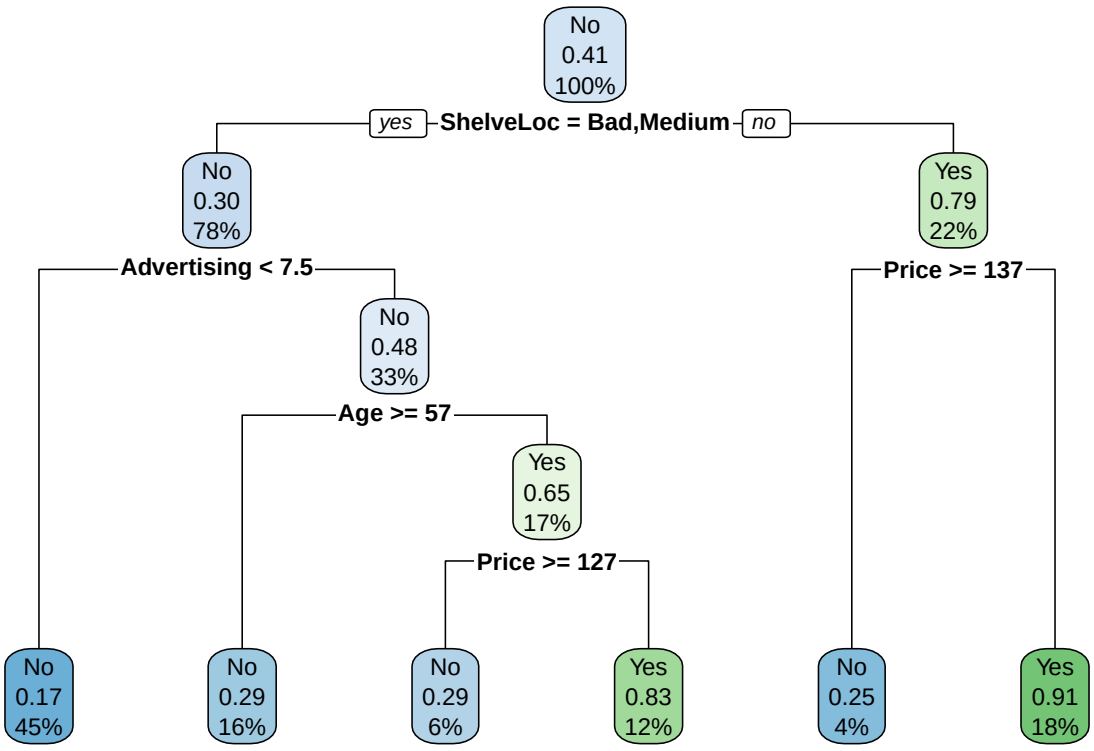
```
best_complexity <- select_best(tune_res)

class_tree_final <- finalize_workflow(class_tree_wf, best_complexity)

class_tree_final_fit <- fit(class_tree_final, data = Carseats_train)
```

At last we can visualize the model, and we see that the better-performing model is much less complex than the original model we fit.

```
class_tree_final_fit %>%
  extract_fit_engine() %>%
  rpart.plot()
```



Fitting Regression Trees

We will now show how we fit a regression tree. This is very similar to what we saw in the last section. The main difference here is that the response we are looking at will be continuous instead of categorical. We can reuse `tree_spec` as a base for the regression decision tree specification.

```
reg_tree_spec <- tree_spec %>%
  set_mode("regression")
```

We can fit the model to the training data set:

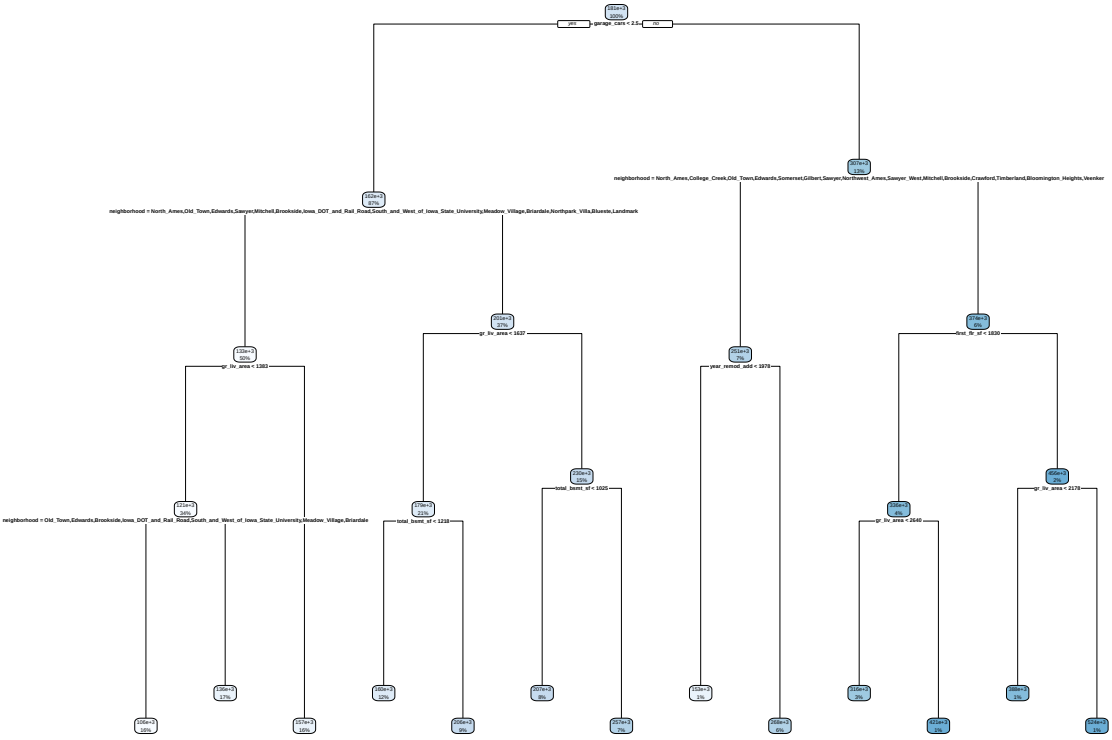
```
reg_tree_fit <- fit(reg_tree_spec,
                    sale_price ~ . -ms_sub_class -ms_zoning, ames_train)
```

```
augment(reg_tree_fit, new_data = ames_test) %>%
  rmse(truth = sale_price, estimate = .pred)
```

```
## # A tibble: 1 x 3
##   .metric .estimator .estimate
##   <chr>   <chr>          <dbl>
## 1 rmse    standard      38596.
```

and the `rpart.plot()` function works for the regression decision tree as well:

```
reg_tree_fit %>%
  extract_fit_engine() %>%
  rpart.plot()
```



Now let us again try to tune `cost_complexity` to find the best performing decision tree.

```
reg_tree_wf <- workflow() %>%
  add_model(reg_tree_spec %>% set_args(cost_complexity = tune())) %>%
  add_formula(sale_price ~ .)

set.seed(3435)
ames_fold <- vfold_cv(ames_train)

param_grid <- grid_regular(cost_complexity(range = c(-4, -1)), levels = 10)

tune_res <- tune_grid(
  reg_tree_wf,
  resamples = ames_fold,
```

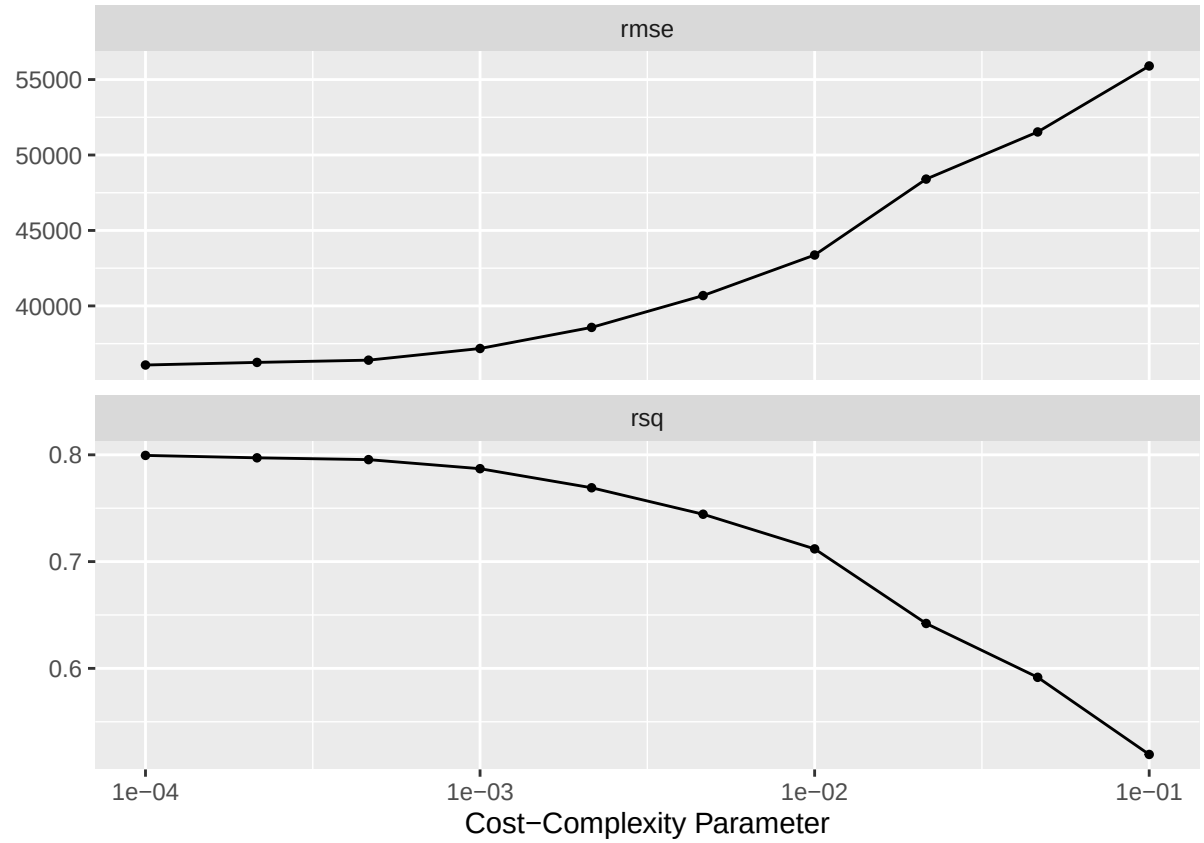
```
  grid = param_grid
)
```

Activities

- What are the defaults for `vfold_cv()`? In other words, how many folds were created here, etc.?

We can visualize the results:

```
autoplot(tune_res)
```



Activities

- Interpret the plots. What relationship(s) do you see between penalty (cost complexity) and performance?

We select the best-performing model according to `rmse` and fit the final model on the whole training data set.

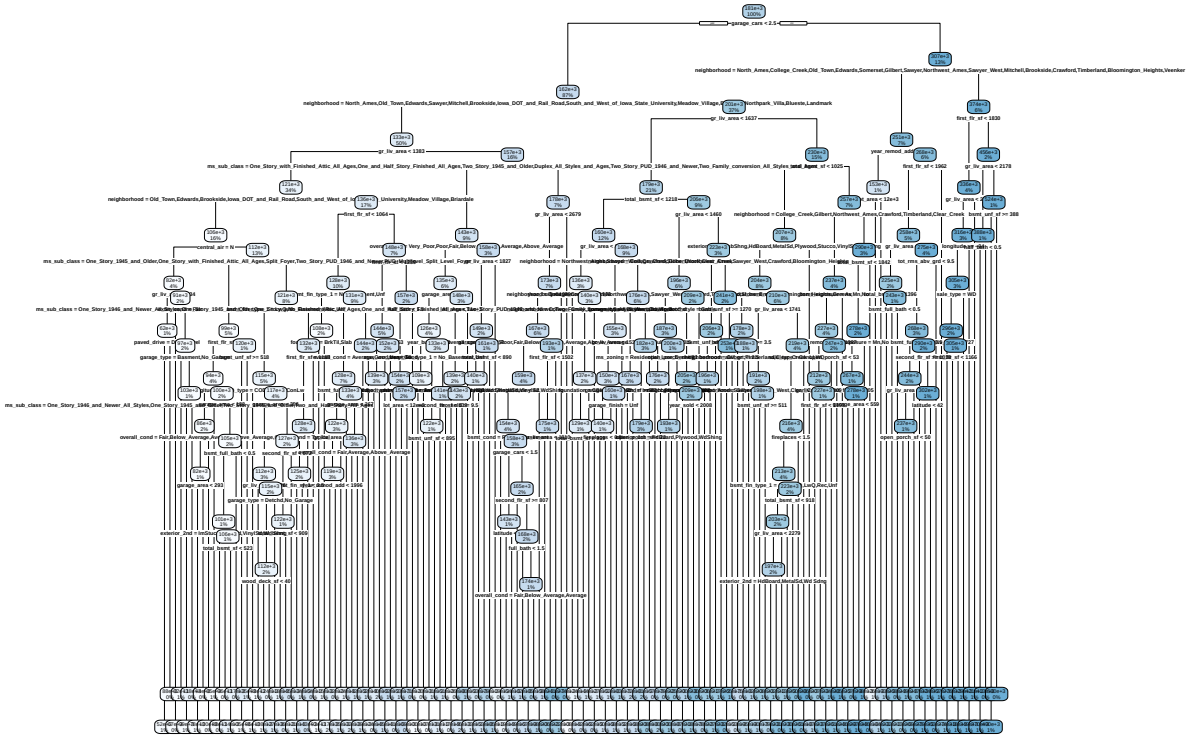
```
best_complexity <- select_best(tune_res, metric = "rmse")

reg_tree_final <- finalize_workflow(reg_tree_wf, best_complexity)

reg_tree_final_fit <- fit(reg_tree_final, data = ames_train)
```

Visualizing the model reveals a much more complex tree than what we saw in the last section.

```
reg_tree_final_fit %>%
  extract_fit_engine() %>%
  rpart.plot()
```



Let's check performance on the testing data set:

```
augment(reg_tree_final_fit, new_data = ames_test) %>%
  rmse(truth = sale_price, estimate = .pred)
```

```
## # A tibble: 1 x 3
##   .metric .estimator .estimate
##   <chr>   <chr>          <dbl>
## 1 rmse    standard      34875.
```

Bagging and Random Forests

Here we apply bagging and random forests to the Ames data set. We will be using the `randomForest` package as the engine. A bagging model is the same as a random forest where mtry is equal to the number of predictors. We can specify `mtry` to be `.cols()`, which means that the number of columns in the predictor matrix is used. This is useful if you want to make the specification more general and useable to many different data sets. `.cols()` is one of many descriptors in the `parsnip` package. We also set `importance = TRUE` in `set_engine()` to tell the engine to save the information regarding variable importance. This is needed if we want to use the `vip` package later.

```
bagging_spec <- rand_forest(mtry = .cols()) %>%
  set_engine("randomForest", importance = TRUE) %>%
  set_mode("regression")
```

We fit the model like normal:

```
bagging_fit <- fit(bagging_spec, sale_price ~ .,
                   data = ames_train)
```

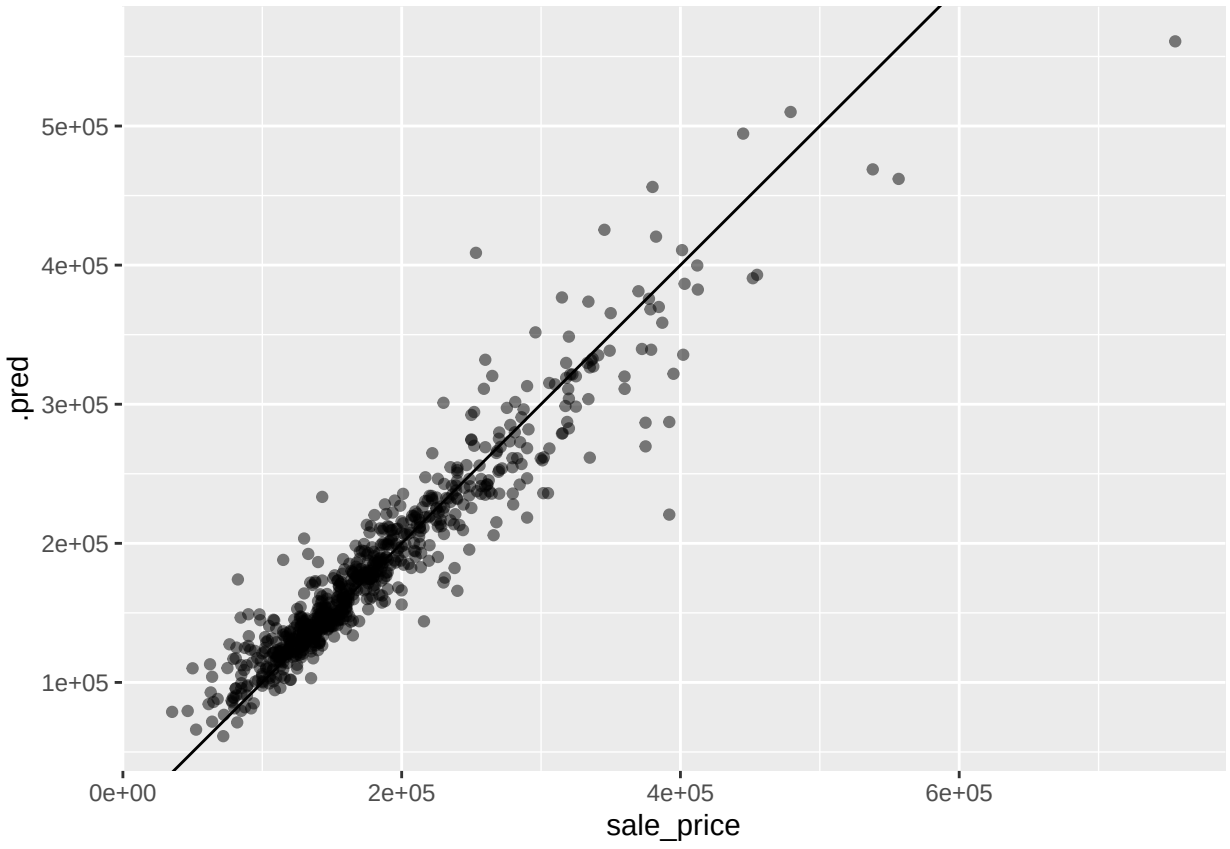
and we take a look at the testing performance:

```
augment(bagging_fit, new_data = ames_test) %>%
  rmse(truth = sale_price, estimate = .pred)
```

```
## # A tibble: 1 x 3
##   .metric .estimator .estimate
##   <chr>   <chr>          <dbl>
## 1 rmse    standard      25420.
```

We can also create a quick scatterplot between the true and predicted values:

```
augment(bagging_fit, new_data = ames_test) %>%
  ggplot(aes(sale_price, .pred)) +
  geom_abline() +
  geom_point(alpha = 0.5)
```

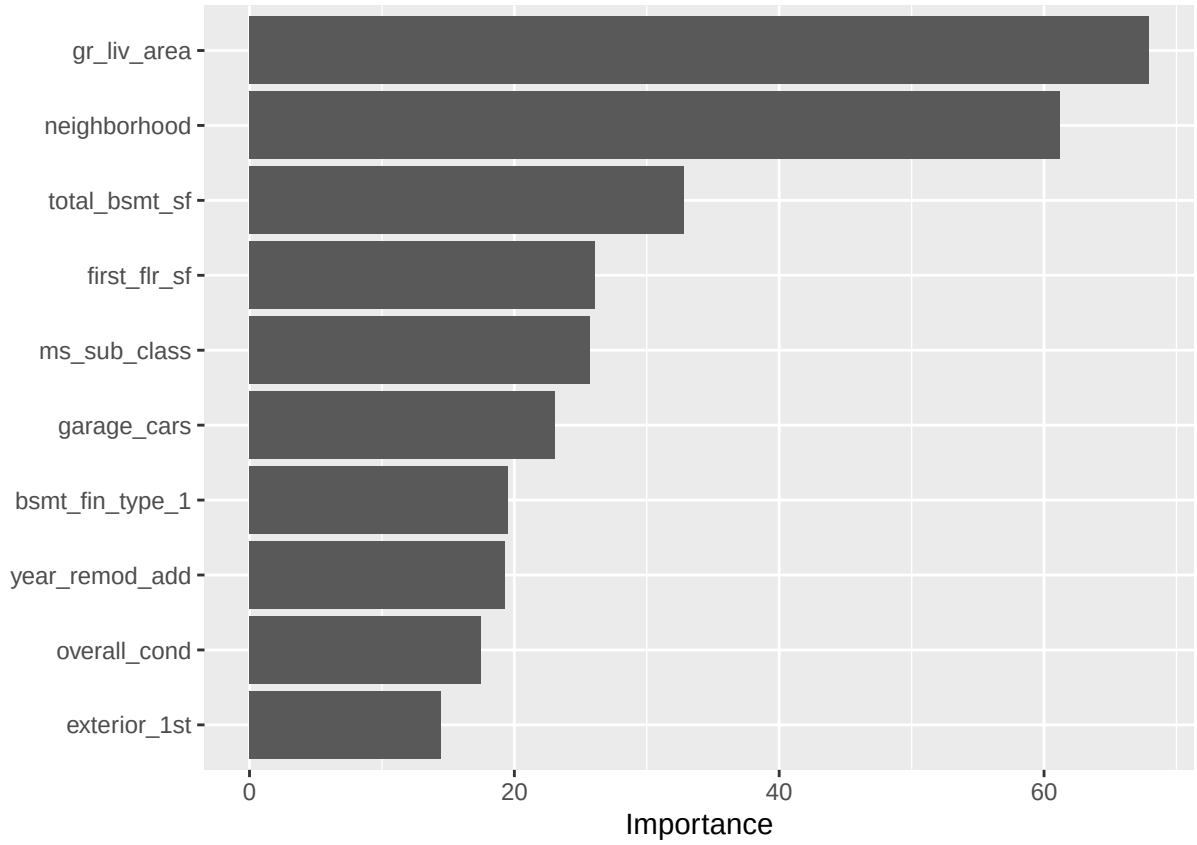


Activities

- Interpret the results. Do you notice anything interesting or unusual? Did this model perform better than the single decision tree?

Next, let us take a look at the variable importance.

```
vip(bagging_fit)
```



Next, let us take a look at a random forest. By default, `randomForest()` uses `p / 3` variables when building a random forest of regression trees, and `sqrt(p)` variables when building a random forest of classification trees. Here, we use `mtry = 6`.

```
rf_spec <- rand_forest(mtry = 6) %>%
  set_engine("randomForest", importance = TRUE) %>%
  set_mode("regression")
```

and fitting the model like normal:

```
rf_fit <- fit(rf_spec, sale_price ~ ., data = ames_train)
```

```
augment(rf_fit, new_data = ames_train) %>%
  rmse(truth = sale_price, estimate = .pred)
```

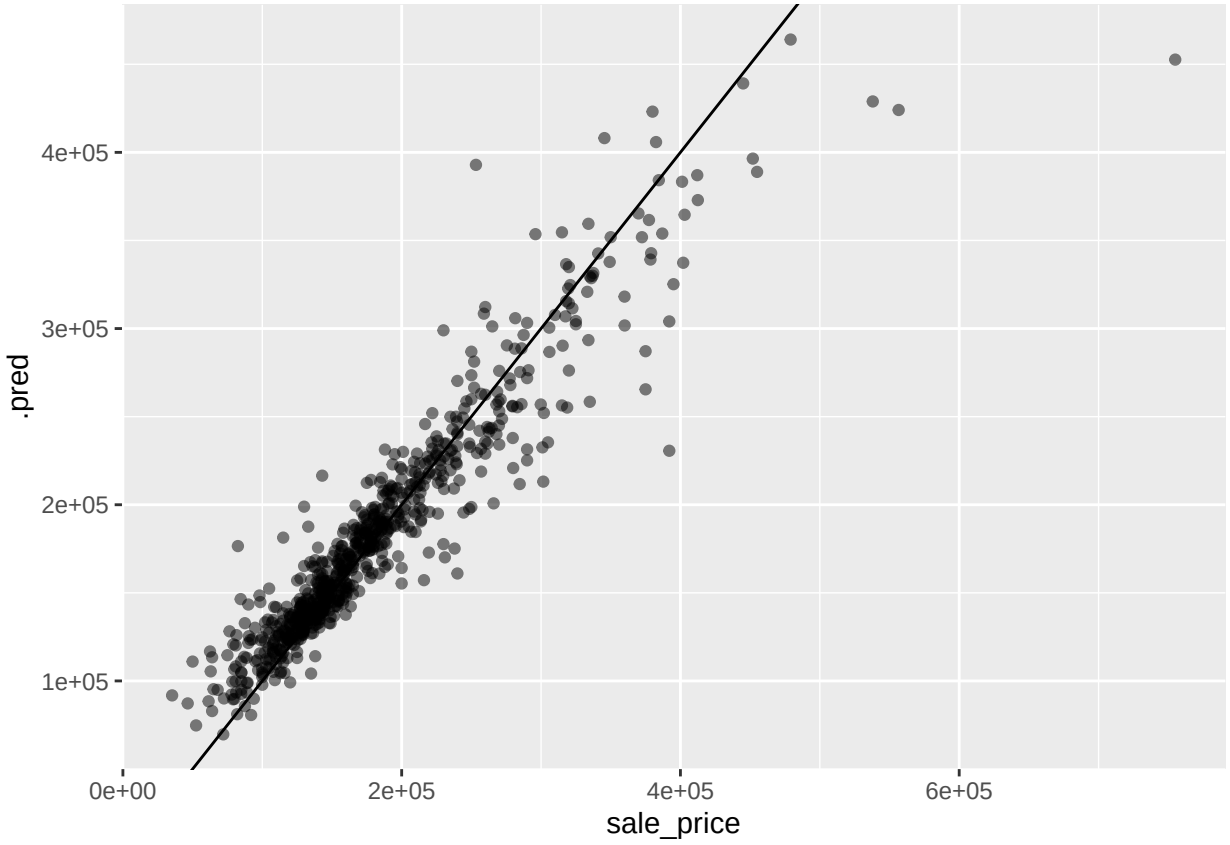
```
## # A tibble: 1 x 3
##   .metric .estimator .estimate
##   <chr>   <chr>          <dbl>
## 1 rmse    standard      12408.
```

Activities

- Out of the three models so far – decision tree, boosted tree, and random forest – which performed best on the testing set?

We can likewise plot the true values against the predicted values:

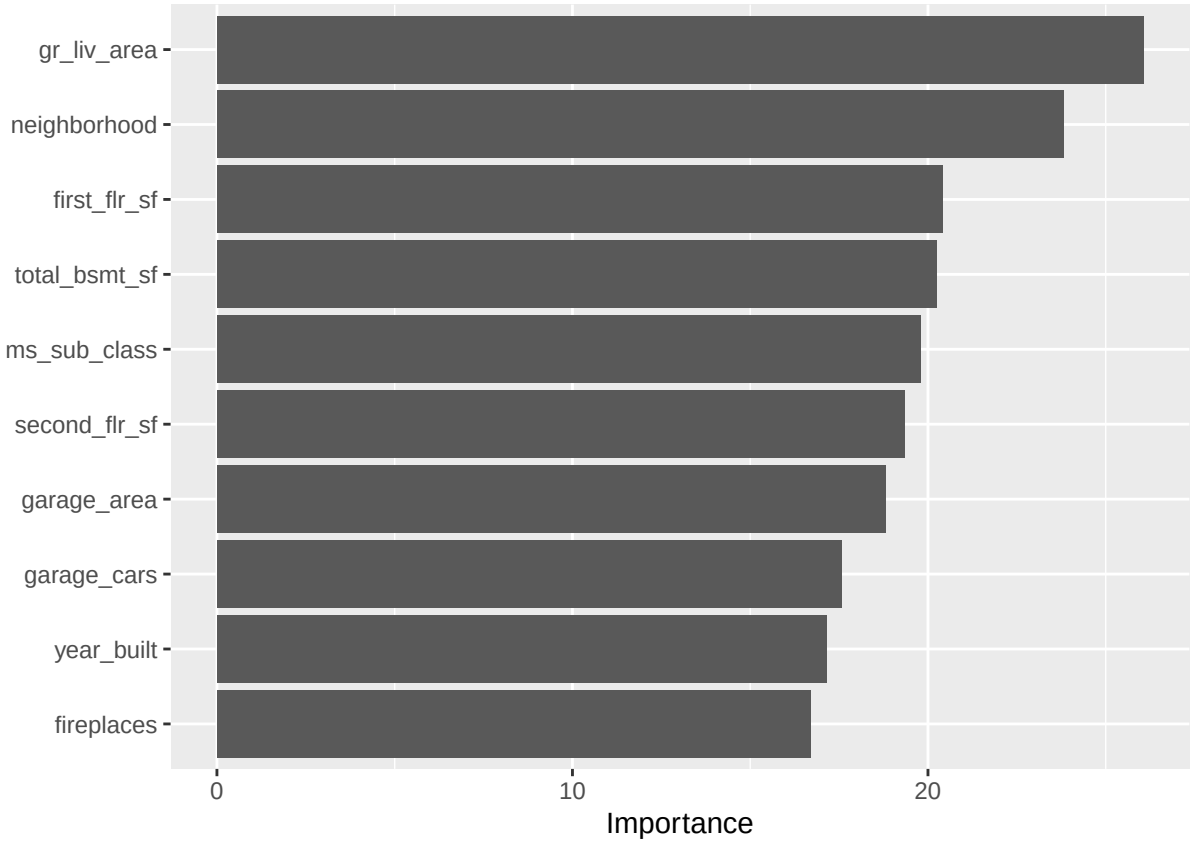
```
augment(rf_fit, new_data = ames_test) %>%
  ggplot(aes(sale_price, .pred)) +
  geom_abline() +
  geom_point(alpha = 0.5)
```



No discernible difference between this chart and the one we created for the bagging model.

And we can look at the variable importance plot:

```
vip(rf_fit)
```



We would normally want to perform hyperparameter tuning for the random forest model, so:

Activities

- Use cross-validation to tune `mtry` for the random forest model on the Ames data set.

Boosting

We will now fit a boosted tree model. The `xgboost` package gives a good implementation of boosted trees. It has many parameters to tune and we know that setting `trees` too high can lead to overfitting. Nevertheless, let us try fitting a boosted tree. We set `tree = 5000` to grow 5000 trees with a maximal depth of 4 by setting `tree_depth = 4`.

```
boost_spec <- boost_tree(trees = 5000, tree_depth = 4) %>%
  set_engine("xgboost") %>%
  set_mode("regression")
```

```
boost_fit <- fit(boost_spec, sale_price ~ ., data = ames_train)
```

```
augment(boost_fit, new_data = ames_test) %>%
  rmse(truth = sale_price, estimate = .pred)
```

```
## # A tibble: 1 x 3
```

```
##   .metric .estimator .estimate
##   <chr>   <chr>          <dbl>
## 1 rmse    standard      24778.
```

Activities

- Use cross-validation to tune `tree` and `tree_depth` for the boosted tree model on the Ames data set.

Resources

The free book Tidy Modeling with R is strongly recommended.

Source

Several parts of this lab come directly from the "ISLR Tidymodels Labs". Credit to Emil Hvitfeldt for writing and maintaining the open-source book.