Notes on Ch 10: Resampling for Evaluating Performance

The Caveman Coder

2025-08-20

The resubstitution approach

Resubstitution simply means using the same data used for training the model, to measure how well the model performs.

Example: Using a random forest model on the Ames dataset:

```
library(tidymodels)
```

```
## -- Attaching packages --------------------------------------- tidymodels 1.3.0 --
```

```
## v broom        1.0.8     v recipes      1.3.1
## v dials        1.4.1     v rsample      1.3.1
## v dplyr        1.1.4     v tibble       3.3.0
## v ggplot2      3.5.2     v tidyr        1.3.1
## v infer        1.0.9     v tune         1.3.0
## v modeldata    1.5.0     v workflows    1.2.0
## v parsnip      1.3.2     v workflowsets 1.1.1
## v purrr        1.1.0     v yardstick    1.3.2
```

```
## -- Conflicts ------------------------------------------ tidymodels_conflicts() --
## x purrr::discard() masks scales::discard()
## x dplyr::filter()  masks stats::filter()
## x dplyr::lag()     masks stats::lag()
## x recipes::step()  masks stats::step()
```

```
tidymodels_prefer()

ames <- mutate(ames, Sale_Price = log10(Sale_Price))

set.seed(502)
ames_split <- initial_split(ames, prop = 0.80, strata = Sale_Price)
ames_train <- training(ames_split)
ames_test <- testing(ames_split)

rf_model <-
  rand_forest(trees = 1000) |>
  set_engine("ranger") |>
  set_mode("regression")

rf_wflow <-
  workflow() |>
  add_formula(
    Sale_Price ~ Neighborhood + Gr_Liv_Area + Year_Built + Bldg_Type + Latitude + Longitude
  ) |>
```

```
  add_model(rf_model)

rf_fit <- rf_wflow |> fit(data = ames_train)

# the linear model created in the previous chapter
ames_rec <-
  recipe(
    Sale_Price ~ Neighborhood + Gr_Liv_Area + Year_Built + Bldg_Type + Latitude + Longitude,
    data = ames_train
  ) |>
  step_log(Gr_Liv_Area, base = 10) |>
  step_other(Neighborhood, threshold = 0.01) |>
  step_interact(~ Gr_Liv_Area:starts_with("Bldg_Type_")) |>
  step_ns(Latitude, Longitude, deg_free = 20)

lm_model <- linear_reg() |> set_engine("lm")

lm_wflow <-
  workflow() |>
  add_model(lm_model) |>
  add_recipe(ames_rec)

lm_fit <- fit(lm_wflow, ames_train)
```

Question: How to compare the linear and random forest models? Ans: Make predictions using the training set. This is called an *apparent metric* or *resubstitution metric*.

Creating a function the generates predictions and formats the results:

```
estimate_perf <- function(model, dat) {
  # Capture the names of the `model` and `dat` objects
  c1 <- match.call()
  obj_name <- as.character(c1$model)
  data_name <- as.character(c1$dat)
  data_name <- gsub("ames_", "", data_name)

  # Estimate these metrics:
  reg_metrics <- metric_set(rmse, rsq)

  model |>
    predict(dat) |>
    bind_cols(dat |> select(Sale_Price)) |>
    reg_metrics(Sale_Price, .pred) |>
    select(-.estimator) |>
    mutate(object = obj_name, data = data_name)
}
```

Calculating RMSE and R^2 and generate the resubstitution statistics:

```
estimate_perf(rf_fit, ames_train)
```

```
## # A tibble: 2 x 4
##   .metric .estimate object data
##   <chr>       <dbl> <chr>  <chr>
## 1 rmse       0.0364 rf_fit train
## 2 rsq        0.960  rf_fit train
```

```
estimate_perf(lm_fit, ames_train)
```

```
## # A tibble: 2 x 4
##   .metric .estimate object data
##   <chr>       <dbl> <chr>  <chr>
## 1 rmse       0.0760 lm_fit train
## 2 rsq        0.813  lm_fit train
```

Random forest performed better because on the log scale, its RMSE is about half as large as that of the linear model.

The next verification step is to make predictions using the test data:

```
estimate_perf(rf_fit, ames_test)
```

```
## # A tibble: 2 x 4
##   .metric .estimate object data
##   <chr>       <dbl> <chr>  <chr>
## 1 rmse       0.0704 rf_fit test
## 2 rsq        0.852  rf_fit test
```

```
estimate_perf(lm_fit, ames_test)
```

```
## # A tibble: 2 x 4
##   .metric .estimate object data
##   <chr>       <dbl> <chr>  <chr>
## 1 rmse       0.0741 lm_fit test
## 2 rsq        0.834  lm_fit test
```

Now, the test set RMSE for the random forest model is much worse. The test set RMSE for the linear model, is more consistent.

Testing the models on the training data will always result in an artificially optimistic estimate of performance (especially compared to test results on a dataset that the model has never "seen" before).

Resampling methods

These are methods of empirical simulation that emulate the process of using some data for modeling and different data for simulation.

Cross-validation

The most common form is v-fold cross-validation where the data are randomly partitioned into sets roughly equal size (aka the folds), and then a scheme is employed to assign some of the folds for "training" the model, and the other/s for evaluation.

Typical code for splitting the data for v-fold cross-validation (using `vfold_cv()`):

```
set.seed(1001)
ames_folds <- vfold_cv(ames_train, v = 10)
ames_folds
```

```
## #  10-fold cross-validation
## # A tibble: 10 x 2
##    splits             id
##    <list>             <chr>
##  1 <split [2107/235]> Fold01
##  2 <split [2107/235]> Fold02
##  3 <split [2108/234]> Fold03
```

```
##  4 <split [2108/234]> Fold04
##  5 <split [2108/234]> Fold05
##  6 <split [2108/234]> Fold06
##  7 <split [2108/234]> Fold07
##  8 <split [2108/234]> Fold08
##  9 <split [2108/234]> Fold09
## 10 <split [2108/234]> Fold10
```

To manually retrieve the paritioned data, the `analysis()` and `assessment()` functions can be used:

```
ames_folds$splits[[1]] |> analysis() |> dim()
```

```
## [1] 2107   74
```

Repeated cross-validation

This is a type of v-fold cross-validation where the assignment of training sets and validation sets are performed repeatedly. This effectively reduces the RSME to $\sigma/\sqrt{10R}$ where R is the number of replicates, and σ is the standard error.

Specifying the number of repeats in creating v-fold cross-validation, the argument `repeats` is used:

```
vfold_cv(ames_train, v = 10, repeats = 5)
```

```
## #  10-fold cross-validation repeated 5 times
## # A tibble: 50 x 3
##    splits             id      id2
##    <list>             <chr>   <chr>
##  1 <split [2107/235]> Repeat1 Fold01
##  2 <split [2107/235]> Repeat1 Fold02
##  3 <split [2108/234]> Repeat1 Fold03
##  4 <split [2108/234]> Repeat1 Fold04
##  5 <split [2108/234]> Repeat1 Fold05
##  6 <split [2108/234]> Repeat1 Fold06
##  7 <split [2108/234]> Repeat1 Fold07
##  8 <split [2108/234]> Repeat1 Fold08
##  9 <split [2108/234]> Repeat1 Fold09
## 10 <split [2108/234]> Repeat1 Fold10
## # i 40 more rows
```

Leave-one-out cross-validation

In this scheme, for each of the folds, one row is taken out, and these removed row is used for validating the model performance. This is computationally expensive and yields inferior results than other v-fold cross-validation methods.

Monte Carlo cross-validation

This is similar to the regular v-fold cross-validation except that here, the proportion of data allocate to validating the model performance is a fixed value, but randomly selected from the folds.

Creating Monte Carlo cross-validation:

```
mc_cv(ames_train, prop = 9/10, times = 20)
```

```
## # Monte Carlo cross-validation (0.9/0.1) with 20 resamples
## # A tibble: 20 x 2
##    splits             id
```

```
##    <list>              <chr>
##  1 <split [2107/235]> Resample01
##  2 <split [2107/235]> Resample02
##  3 <split [2107/235]> Resample03
##  4 <split [2107/235]> Resample04
##  5 <split [2107/235]> Resample05
##  6 <split [2107/235]> Resample06
##  7 <split [2107/235]> Resample07
##  8 <split [2107/235]> Resample08
##  9 <split [2107/235]> Resample09
## 10 <split [2107/235]> Resample10
## 11 <split [2107/235]> Resample11
## 12 <split [2107/235]> Resample12
## 13 <split [2107/235]> Resample13
## 14 <split [2107/235]> Resample14
## 15 <split [2107/235]> Resample15
## 16 <split [2107/235]> Resample16
## 17 <split [2107/235]> Resample17
## 18 <split [2107/235]> Resample18
## 19 <split [2107/235]> Resample19
## 20 <split [2107/235]> Resample20
```

Bootstrapping

A bootstrap sample of a dataset is a sample drawn with replacement from the dataset, having the same size as the source dataset.

In this setup, each data point has a 63.2% chance of getting included in the sample set at least once.

This means that if a bootstrap sample is made from the training set, on average, 36.8% of the training set will not be included in the bootstrap sample, and this remaining set, aka the out-of-bag sample, will be used as the assessment set.

From the **rssample**, bootstrap samples can be created using `bootstraps()`:

```
bootstraps(ames_train, times = 5)
```

```
## # Bootstrap sampling
## # A tibble: 5 x 2
##   splits             id
##   <list>             <chr>
## 1 <split [2342/847]> Bootstrap1
## 2 <split [2342/902]> Bootstrap2
## 3 <split [2342/871]> Bootstrap3
## 4 <split [2342/856]> Bootstrap4
## 5 <split [2342/870]> Bootstrap5
```

Rolling forecasting origin resampling

When the data have a strong time component, a resampling method should support modeling to estimate seasonal and other temporal trends within the data – any technique that randomly samples values from the training set will disrupt the model's ability to "learn" from these patterns.

- In the technique of rolling forecast origin resampling, the size of the initial analysis and assessment sets are first specified. The first iteration of resampling uses these sizes, starting from the beginning of the series, then on the second iteration, the same data is used, but they are shifted over by a set number of samples.

Example:
First iteration: analysis set -> row 1 to 8; validation set -> row 9 to 11
Second iteration: analysis set -> row 2 to 9; validation set -> row 10 to 12
And so on...

The resamples need not increment by one. For example, for large datasets, the incremental block could be a week or month instead of a day.

The resamples can also grow cumulatively instead of remaining similar in size.

Example: Using the **resample** to create rolling forecast origin resamples composed of six sets of 30-day blocks for the analysis set, and 30 days with a 29-day skip for the assessment set:

```
time_slices <-
  tibble(x = 1:365) |>
  rolling_origin(initial = 6 * 30, assess = 30, skip = 29, cumulative = FALSE)

time_slices
```

```
## # Rolling origin forecast resampling
## # A tibble: 6 x 2
##   splits           id
##   <list>           <chr>
## 1 <split [180/30]> Slice1
## 2 <split [180/30]> Slice2
## 3 <split [180/30]> Slice3
## 4 <split [180/30]> Slice4
## 5 <split [180/30]> Slice5
## 6 <split [180/30]> Slice6
```

```
data_range <- function(x) {
  summarize(x, first = min(x), last = max(x))
}

map_dfr(time_slices$splits, ~ analysis(.x) |> data_range())
```

```
## # A tibble: 6 x 2
##   first  last
##   <int> <int>
## 1     1   180
## 2    31   210
## 3    61   240
## 4    91   270
## 5   121   300
## 6   151   330
```

```
map_dfr(time_slices$splits, ~ assessment(.x) |> data_range())
```

```
## # A tibble: 6 x 2
##   first  last
##   <int> <int>
## 1   181   210
## 2   211   240
## 3   241   270
## 4   271   300
## 5   301   330
## 6   331   360
```

Estimating Performance

Resampling process:

1. The analysis set is used to preprocess the data, apply the preprocessing to itself, and use these preprocessed data to fit the model.

2. The preprocessing statistics produced by the analysis set are applied to the assessment set. The predictions from the assessment set estimate performance on new data.

Syntax:

```
model_spec |> fit_resamples(formula, resamples, ...)
model_spec |> fit_resamples(recipe,  resamples, ...)
workflow |> fit_resamples(            resampes, ...)
```

Optional arguments: - `metrics`: A metric set of performance statistics to compute. Default metrics for regression models are RMSE and R^2. Default metrics for classification models compute the area under the ROC and overall accuracy.
- `control`: A list created by `control_resamples()` with different arguments.

Commonly-used control arguments:
- `verbose`: prints process log when set to TRUE.
- `extract`: Used to retain objects from each model iteration. (To be discussed later)
- `save_pred`: Saves the assessment test predictions when set to TRUE.

Example: saving predictions in order to visualize the model fit and residuals:

```
keep_pred <- control_resamples(save_pred = TRUE, save_workflow = TRUE)

set.seed(1003)
rf_res <-
  rf_wflow |>
  fit_resamples(resamples = ames_folds, control = keep_pred)

rf_res
```

```
## # Resampling results
## # 10-fold cross-validation
## # A tibble: 10 x 5
##    splits             id     .metrics         .notes           .predictions
##    <list>             <chr>  <list>           <list>           <list>
##  1 <split [2107/235]> Fold01 <tibble [2 x 4]> <tibble [0 x 3]> <tibble>
##  2 <split [2107/235]> Fold02 <tibble [2 x 4]> <tibble [0 x 3]> <tibble>
##  3 <split [2108/234]> Fold03 <tibble [2 x 4]> <tibble [0 x 3]> <tibble>
##  4 <split [2108/234]> Fold04 <tibble [2 x 4]> <tibble [0 x 3]> <tibble>
##  5 <split [2108/234]> Fold05 <tibble [2 x 4]> <tibble [0 x 3]> <tibble>
##  6 <split [2108/234]> Fold06 <tibble [2 x 4]> <tibble [0 x 3]> <tibble>
##  7 <split [2108/234]> Fold07 <tibble [2 x 4]> <tibble [0 x 3]> <tibble>
##  8 <split [2108/234]> Fold08 <tibble [2 x 4]> <tibble [0 x 3]> <tibble>
##  9 <split [2108/234]> Fold09 <tibble [2 x 4]> <tibble [0 x 3]> <tibble>
## 10 <split [2108/234]> Fold10 <tibble [2 x 4]> <tibble [0 x 3]> <tibble>
```

Reconfiguring the resulting tibble to make it easy to see the metrics:

```
collect_metrics(rf_res)
```

```
## # A tibble: 2 x 6
##   .metric .estimator   mean     n std_err .config
```

```
##   <chr>   <chr>       <dbl> <int>   <dbl> <chr>
## 1 rmse    standard   0.0721    10 0.00305 Preprocessor1_Model1
## 2 rsq     standard   0.831     10 0.0108  Preprocessor1_Model1
```

The metrics displayed are the resampling estimates averaged over the indivudual replicates.

Getting the metrics for each resample using the option `summarize = FALSE` to `collect(metrics)`:

```
collect_metrics(rf_res, summarize = FALSE)
```

```
## # A tibble: 20 x 5
##    id     .metric .estimator .estimate .config
##    <chr>  <chr>   <chr>          <dbl> <chr>
##  1 Fold01 rmse    standard      0.0605 Preprocessor1_Model1
##  2 Fold01 rsq     standard      0.861  Preprocessor1_Model1
##  3 Fold02 rmse    standard      0.0653 Preprocessor1_Model1
##  4 Fold02 rsq     standard      0.861  Preprocessor1_Model1
##  5 Fold03 rmse    standard      0.0609 Preprocessor1_Model1
##  6 Fold03 rsq     standard      0.880  Preprocessor1_Model1
##  7 Fold04 rmse    standard      0.0699 Preprocessor1_Model1
##  8 Fold04 rsq     standard      0.818  Preprocessor1_Model1
##  9 Fold05 rmse    standard      0.0738 Preprocessor1_Model1
## 10 Fold05 rsq     standard      0.851  Preprocessor1_Model1
## 11 Fold06 rmse    standard      0.0682 Preprocessor1_Model1
## 12 Fold06 rsq     standard      0.822  Preprocessor1_Model1
## 13 Fold07 rmse    standard      0.0712 Preprocessor1_Model1
## 14 Fold07 rsq     standard      0.850  Preprocessor1_Model1
## 15 Fold08 rmse    standard      0.0870 Preprocessor1_Model1
## 16 Fold08 rsq     standard      0.784  Preprocessor1_Model1
## 17 Fold09 rmse    standard      0.0887 Preprocessor1_Model1
## 18 Fold09 rsq     standard      0.789  Preprocessor1_Model1
## 19 Fold10 rmse    standard      0.0752 Preprocessor1_Model1
## 20 Fold10 rsq     standard      0.795  Preprocessor1_Model1
```

Getting the assessment set predictions:

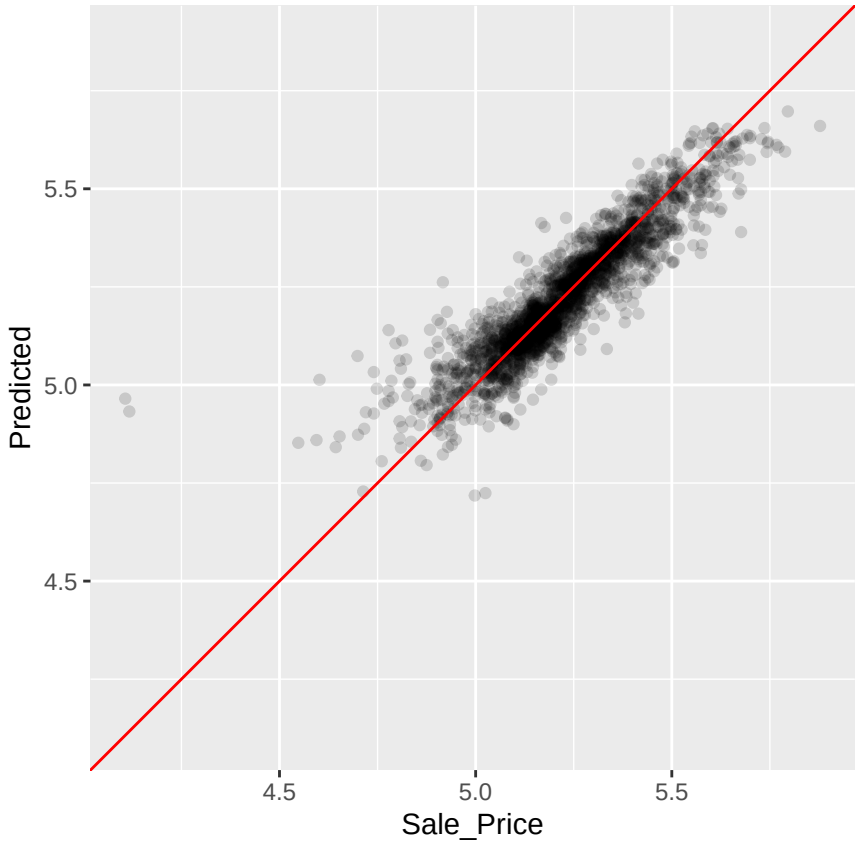
```
assess_res <- collect_predictions(rf_res)
assess_res
```

```
## # A tibble: 2,342 x 5
##    .pred id      .row Sale_Price .config
##    <dbl> <chr>  <int>      <dbl> <chr>
##  1  5.10 Fold01    10       5.09 Preprocessor1_Model1
##  2  4.92 Fold01    27       4.90 Preprocessor1_Model1
##  3  5.21 Fold01    47       5.08 Preprocessor1_Model1
##  4  5.13 Fold01    52       5.10 Preprocessor1_Model1
##  5  5.13 Fold01    59       5.10 Preprocessor1_Model1
##  6  5.13 Fold01    63       5.11 Preprocessor1_Model1
##  7  4.87 Fold01    65       4.91 Preprocessor1_Model1
##  8  4.98 Fold01    66       5.04 Preprocessor1_Model1
##  9  4.91 Fold01    67       4.84 Preprocessor1_Model1
## 10  4.91 Fold01    68       5.01 Preprocessor1_Model1
## # i 2,332 more rows
```

Plotting the predictions is helpful in understanding where the model failed.

Comparing the observed and held-out predicted values:

```
assess_res |>
  ggplot(aes(x = Sale_Price, y = .pred)) +
  geom_point(alpha = 0.15) +
  geom_abline(color = "red") +
  coord_obs_pred() +
  ylab("Predicted")
```



Finding out which rows were overpredicted by the model:

```
over_predicted <- assess_res |>
  mutate(residual = Sale_Price - .pred) |>
  arrange(desc(abs(residual))) |>
  slice(1:2)

over_predicted
```

```
## # A tibble: 2 x 6
##   .pred id      .row Sale_Price .config              residual
##   <dbl> <chr>  <int>      <dbl> <chr>                   <dbl>
## 1  4.97 Fold09    32       4.11 Preprocessor1_Model1   -0.858
## 2  4.93 Fold08   317       4.12 Preprocessor1_Model1   -0.815
```

Finding out which hourses were overpredicted by the model based on the row number above:

```
ames_train |>
  slice(over_predicted$.row) |>
  select(Gr_Liv_Area, Neighborhood, Year_Built, Bedroom_AbvGr, Full_Bath)
```

```
## # A tibble: 2 x 5
```

```
##   Gr_Liv_Area Neighborhood           Year_Built Bedroom_AbvGr Full_Bath
##         <int> <fct>                       <int>         <int>     <int>
## 1         832 Old_Town                     1923             2         1
## 2         733 Iowa_DOT_and_Rail_Road       1952             2         1
```

Identifying examples like these with especially poor performance can help us follow up and investigate why these specific predictors are so poor.

Using the validation set instead of cross-validation:

```
set.seed(52)

# Put 60% into training, 20% in validation, 20% in testing
ames_val_split <- initial_validation_split(ames, prop = c(0.6, 0.2))
ames_val_split
```

```
## <Training/Validation/Testing/Total>
## <1758/586/586/2930>
```

```
# Object used for resampling
val_set <- validation_set(ames_val_split)
val_set
```

```
## # A tibble: 1 x 2
##   splits             id
##   <list>             <chr>
## 1 <split [1758/586]> validation
```

```
val_res <- rf_wflow |> fit_resamples(resamples = val_set)

val_res
```

```
## # Resampling results
## # Validation Set (0.75/0.25)
## # A tibble: 1 x 4
##   splits             id         .metrics         .notes
##   <list>             <chr>      <list>           <list>
## 1 <split [1758/586]> validation <tibble [2 x 4]> <tibble [0 x 3]>
```

```
collect_metrics(val_res)
```

```
## # A tibble: 2 x 6
##   .metric .estimator   mean     n std_err .config
##   <chr>   <chr>       <dbl> <int>   <dbl> <chr>
## 1 rmse    standard   0.0728     1      NA Preprocessor1_Model1
## 2 rsq     standard   0.822      1      NA Preprocessor1_Model1
```

The results are much closer to the test set results than to the resubstitution estimates of performance.

Parallel processing

The **tune** package and the **foreach** package are used to facilitate parallel computations. This means that computations could be split across processors on the same computer or across different computers, depending on the chosen technology.

Viewing the number of possible worker processes on a single computer using the **parallel** package:

```
# The number of physical cores in the hardware:
parallel::detectCores(logical = FALSE)
```

```
## [1] 8
```

```
# The number of possible independent processes that can be
# simultaneously used:
parallel::detectCores(logical = TRUE)
```

```
## [1] 8
```

These can be different if the processors are using hyperthreading or another equivalent technology which creates virtual cores for each physical core.

Splitting computations by threads in a Unix-like system like Linux and macOS:

```
# for Unix-like systems
library(doMC)
registerDoMC(cores = 2)

# Now run fit_resamples()...
```

Resetting the computations to sequential processing:

```
registerDoSEQ()
```

Parallelizing computations using network sockets:

```
# for all operating systems
library(doParallel)

# create a cluster object and then register:
cl <- makePSOCKcluster(2)
registerDoParallel(cl)

# Now run fit_resamples()...

stopCluster(cl)
```

Saving the resampled objects

Extracting the linear regression model using the recipe from chapter 8:

```
extract_recipe(lm_fit, estimated = TRUE)
```

```
##
## -- Recipe ----------------------------------------------------------------------
##
## -- Inputs
## Number of variables by role
## outcome:   1
## predictor: 6
##
## -- Training information
## Training data contained 2342 data points and no incomplete rows.
##
```

```
## -- Operations

## * Log transformation on: Gr_Liv_Area | Trained

## * Collapsing factor levels for: Neighborhood | Trained

## * Interactions with: <none> | Trained

## * Natural splines on: Latitude Longitude | Trained
```

Saving hte linear model coefficients for a fitted model object from a workflow:

```
get_model <- function(x) {
  extract_fit_parsnip(x) |> tidy()
}

# testing the function
get_model(lm_fit)
```

```
## # A tibble: 69 x 5
##    term                          estimate std.error statistic    p.value
##    <chr>                            <dbl>     <dbl>     <dbl>      <dbl>
##  1 (Intercept)                    -0.167      0.289     -0.577 0.564
##  2 NeighborhoodCollege_Creek      -0.00962    0.0339    -0.284 0.777
##  3 NeighborhoodOld_Town           -0.0603     0.0126    -4.80  0.00000171
##  4 NeighborhoodEdwards            -0.0845     0.0284    -2.98  0.00292
##  5 NeighborhoodSomerset            0.0847     0.0194     4.38  0.0000126
##  6 NeighborhoodNorthridge_Heights  0.150      0.0351     4.29  0.0000188
##  7 NeighborhoodGilbert             0.0198     0.0319     0.620 0.535
##  8 NeighborhoodSawyer             -0.0996     0.0266    -3.74  0.000185
##  9 NeighborhoodNorthwest_Ames      0.0208     0.0168     1.24  0.214
## 10 NeighborhoodSawyer_West        -0.109      0.0322    -3.37  0.000771
## # i 59 more rows
```

Applying the function to the ten resampled fits:

```
ctrl <- control_resamples(extract = get_model)

ctrl
```

```
## grid/resamples control object
```

```
lm_res <- lm_wflow |> fit_resamples(resamples = ames_folds, control = ctrl)
lm_res
```

```
## # Resampling results
## # 10-fold cross-validation
## # A tibble: 10 x 5
##    splits             id     .metrics         .notes           .extracts
##    <list>             <chr>  <list>           <list>           <list>
##  1 <split [2107/235]> Fold01 <tibble [2 x 4]> <tibble [0 x 3]> <tibble [1 x 2]>
##  2 <split [2107/235]> Fold02 <tibble [2 x 4]> <tibble [0 x 3]> <tibble [1 x 2]>
##  3 <split [2108/234]> Fold03 <tibble [2 x 4]> <tibble [0 x 3]> <tibble [1 x 2]>
##  4 <split [2108/234]> Fold04 <tibble [2 x 4]> <tibble [0 x 3]> <tibble [1 x 2]>
##  5 <split [2108/234]> Fold05 <tibble [2 x 4]> <tibble [0 x 3]> <tibble [1 x 2]>
##  6 <split [2108/234]> Fold06 <tibble [2 x 4]> <tibble [0 x 3]> <tibble [1 x 2]>
##  7 <split [2108/234]> Fold07 <tibble [2 x 4]> <tibble [0 x 3]> <tibble [1 x 2]>
##  8 <split [2108/234]> Fold08 <tibble [2 x 4]> <tibble [0 x 3]> <tibble [1 x 2]>
##  9 <split [2108/234]> Fold09 <tibble [2 x 4]> <tibble [0 x 3]> <tibble [1 x 2]>
```

```
## 10 <split [2108/234]> Fold10 <tibble [2 x 4]> <tibble [0 x 3]> <tibble [1 x 2]>
```

Looking into the `.extracts` column with nested tibbles:

```
lm_res$.extracts[[1]]
```

```
## # A tibble: 1 x 2
##   .extracts         .config
##   <list>            <chr>
## 1 <tibble [69 x 5]> Preprocessor1_Model1
```

```
# Getting the results
lm_res$.extracts[[1]][[1]]
```

```
## [[1]]
## # A tibble: 69 x 5
##    term                          estimate std.error statistic    p.value
##    <chr>                            <dbl>     <dbl>     <dbl>      <dbl>
##  1 (Intercept)                     -0.239     0.308     -0.775 0.439
##  2 NeighborhoodCollege_Creek       -0.0144    0.0361    -0.401 0.689
##  3 NeighborhoodOld_Town            -0.0656    0.0134    -4.91  0.00000101
##  4 NeighborhoodEdwards             -0.0967    0.0299    -3.23  0.00124
##  5 NeighborhoodSomerset             0.0733    0.0210     3.49  0.000501
##  6 NeighborhoodNorthridge_Heights   0.154     0.0386     3.99  0.0000671
##  7 NeighborhoodGilbert              0.0217    0.0354     0.615 0.539
##  8 NeighborhoodSawyer              -0.116     0.0281    -4.11  0.0000405
##  9 NeighborhoodNorthwest_Ames       0.0130    0.0181     0.716 0.474
## 10 NeighborhoodSawyer_West         -0.114     0.0340    -3.35  0.000836
## # i 59 more rows
```

This might be a convoluted method for saving the model results – however, there is a method to the madness, so to speak. The `extract` function is flexible and does not assume that the user will only save a single tibble per resample.

Collecting and flattening out all of the results:

```
all_coef <- map_dfr(lm_res$.extracts, ~ .x[[1]][[1]])
all_coef
```

```
## # A tibble: 684 x 5
##    term                          estimate std.error statistic    p.value
##    <chr>                            <dbl>     <dbl>     <dbl>      <dbl>
##  1 (Intercept)                     -0.239     0.308     -0.775 0.439
##  2 NeighborhoodCollege_Creek       -0.0144    0.0361    -0.401 0.689
##  3 NeighborhoodOld_Town            -0.0656    0.0134    -4.91  0.00000101
##  4 NeighborhoodEdwards             -0.0967    0.0299    -3.23  0.00124
##  5 NeighborhoodSomerset             0.0733    0.0210     3.49  0.000501
##  6 NeighborhoodNorthridge_Heights   0.154     0.0386     3.99  0.0000671
##  7 NeighborhoodGilbert              0.0217    0.0354     0.615 0.539
##  8 NeighborhoodSawyer              -0.116     0.0281    -4.11  0.0000405
##  9 NeighborhoodNorthwest_Ames       0.0130    0.0181     0.716 0.474
## 10 NeighborhoodSawyer_West         -0.114     0.0340    -3.35  0.000836
## # i 674 more rows
```

```
# Show the replicates for a single predictor:
filter(all_coef, term == "Year_Built")
```

```
## # A tibble: 10 x 5
##    term       estimate std.error statistic  p.value
```

```
##    <chr>         <dbl>     <dbl>     <dbl>    <dbl>
##  1 Year_Built  0.00185  0.000143      13.0 4.17e-37
##  2 Year_Built  0.00183  0.000145      12.6 2.59e-35
##  3 Year_Built  0.00190  0.000144      13.2 3.45e-38
##  4 Year_Built  0.00186  0.000141      13.2 4.24e-38
##  5 Year_Built  0.00189  0.000145      13.0 2.07e-37
##  6 Year_Built  0.00187  0.000144      13.0 6.27e-37
##  7 Year_Built  0.00181  0.000144      12.6 3.64e-35
##  8 Year_Built  0.00189  0.000140      13.5 5.55e-40
##  9 Year_Built  0.00162  0.000140      11.5 6.31e-30
## 10 Year_Built  0.00182  0.000144      12.6 4.21e-35
```