10 Resampling for evaluating performance

Chapter 9 described statistics for measuring model performance, but which data are best used to compute these statistics? Chapter 5 introduced the idea of data spending where the test set was recommended for obtaining an unbiased estimate of performance. However, we usually need to understand the effectiveness of the model *before using the test set*.

> In fact, typically we can't decide on *which* final model to take to the test set without making model assessments.

In this chapter, we describe an approach called resampling that can fill this gap. Resampling estimates of performance can generalize to new data. The next chapter complements this one by demonstrating statistical methods that compare resampling results.

To motivate this chapter, the next section demonstrates how naive performance estimates can often fail.

10.1 THE RESUBSTITUTION APPROACH

Let's again use the Ames data to demonstrate the concepts in this chapter. Section 8.8 summarizes the current state of our Ames analysis. It includes a recipe object named `ames_rec`, a linear model, and a workflow using that recipe and model called `lm_wflow`. This workflow was fit on the training set, resulting in `lm_fit`.

For a comparison to this linear model, we can also fit a different type of model. *Random forests* are a tree ensemble method that operate by creating a large number of decision trees from slightly different versions of the training set (Breiman 2001a). This collection of trees makes up the ensemble. When predicting a new sample, each ensemble member makes a separate prediction. These are averaged to create the final ensemble prediction for the new data point.

Random forest models are very powerful and they can emulate the underlying data patterns very closely. While this model can be computationally intensive, it is very low-maintenance; very little preprocessing is required (as documented in Appendix A).

Using the same predictor set as the linear model (without the extra preprocessing steps), we can fit a random forest model to the training set via the **ranger** package. This model requires no preprocessing so a simple formula can be used:

```
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)
```

How should the two models be compared? For demonstration, we will predict the training set to produce what is known as the “apparent error rate” or the “resubstitution error rate”. This function creates predictions and formats the results:

```
estimate_perf <- function(model, dat) {
  # Capture the names of the objects used
  cl <- match.call()
  obj_name <- as.character(cl$model)
  data_name <- as.character(cl$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)
}
```

Both RMSE and R^2 are computed. The resubstitution statistics are:

```
estimate_perf(rf_fit, ames_train)
#> # A tibble: 2 × 4
#>   .metric .estimate object data
#>   <chr>       <dbl> <chr>  <chr>
#> 1 rmse       0.0364 rf_fit train
#> 2 rsq        0.961  rf_fit train
estimate_perf(lm_fit, ames_train)
#> # A tibble: 2 × 4
#>   .metric .estimate object data
#>   <chr>       <dbl> <chr>  <chr>
#> 1 rmse       0.0751 lm_fit train
#> 2 rsq        0.819  lm_fit train
```

Based on these results, the random forest is much more capable of predicting the sale prices; the RMSE estimate is 2.06-fold better than linear regression. If these two models were under consideration for this prediction problem, the random forest would probably be chosen. The next step applies the random forest model to the test set for final verification:

```
estimate_perf(rf_fit, ames_test)
#> # A tibble: 2 × 4
#>   .metric .estimate object data
#>   <chr>       <dbl> <chr>  <chr>
#> 1 rmse       0.0695 rf_fit test
#> 2 rsq        0.853  rf_fit test
```

The test set RMSE estimate, 0.0695, is **much worse than the training set** value of 0.0364! Why did this happen?

Many predictive models are capable of learning complex trends from the data. In statistics, these are commonly referred to as *low bias models*.

> In this context, *bias* is the difference between the true data pattern and the types of patterns that the model can emulate. Many black-box machine learning models have low bias. Other models (such as linear/logistic regression, discriminant analysis, and others) are not as adaptable and are considered *high-bias* models. See Section 1.2.5 of Kuhn and Johnson (2020) for a discussion.

For a low-bias model, the high degree of predictive capacity can sometimes result in the model nearly memorizing the training set data. As an obvious example, consider a 1-nearest neighbor model. It will always provide perfect predictions for the training set no matter how well it truly works for other data sets. Random forest models are similar; re-predicting the training set will always result in an artificially optimistic estimate of performance.

For both models, this table summarizes the RMSE estimate for the training and test sets:

	RMSE Estimates	
object	train	test
lm_fit	0.0751	0.0754
rf_fit	0.0364	0.0695

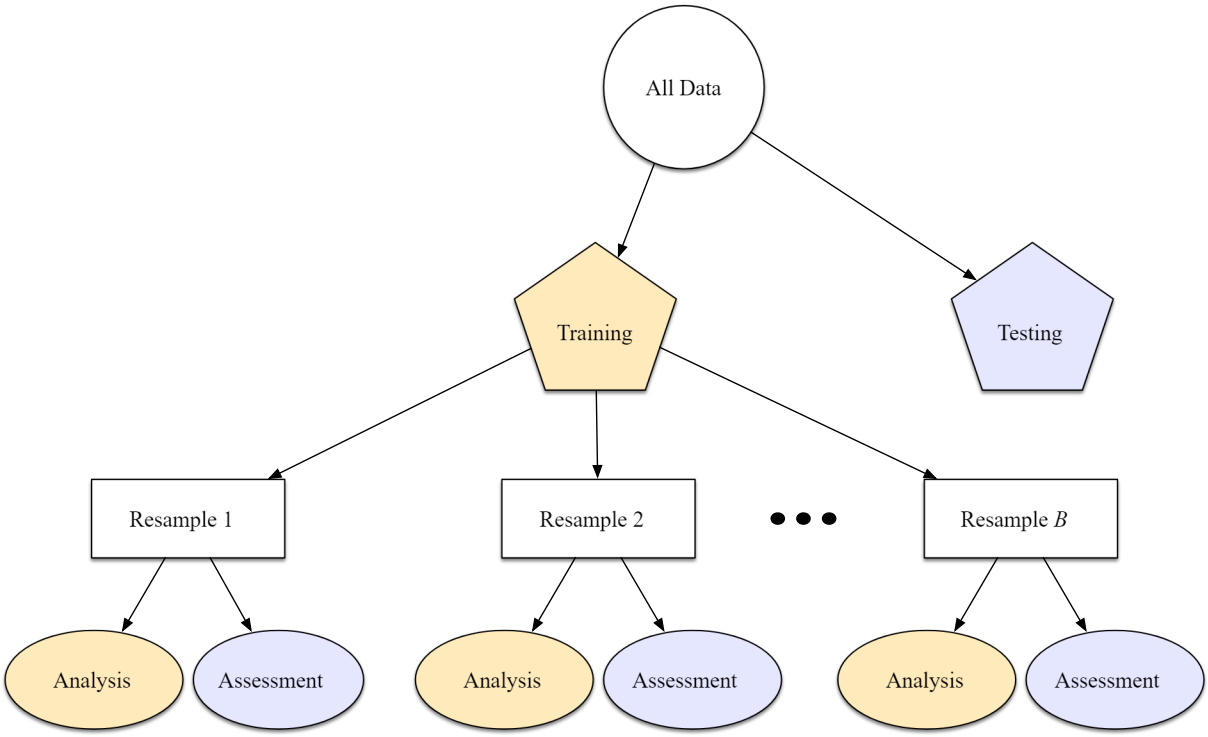
Notice that the linear regression model is consistent between training and testing, because of its limited complexity[13].

> The main take-away from this example is that re-predicting the training set is a **bad idea** for most models.

If the test set should not be used immediately, and re-predicting the training set is a bad idea, what should be done? *Resampling methods*, such as cross-validation or validation sets, are the solution.

10.2 RESAMPLING METHODS

Resampling methods are empirical simulation systems that emulate the process of using some data for modeling and different data for evaluation. Most resampling methods are iterative, meaning that this process is repeated multiple times. This diagram illustrates how resampling methods generally operate:



Resampling is only conducted on the training set. The test set is not involved. For each iteration of resampling, the data are partitioned into two subsamples:

- The model is fit with the **analysis set**.
- The model is evaluated with the **assessment set**.

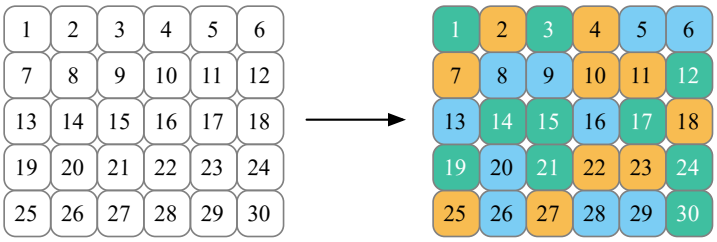
These are somewhat analogous to training and test sets. Our language of *analysis* and *assessment* avoids confusion with initial split of the data. These data sets are mutually exclusive. The partitioning scheme used to create the analysis and assessment sets is usually the defining characteristic of the method.

Suppose twenty iterations of resampling are conducted. This means that twenty separate models are fit on the analysis sets and the corresponding assessment sets produce twenty sets of performance statistics. The final estimate of performance for a model is the average of the twenty replicates of the statistics. This average has very good generalization properties and is far better than the resubstituion estimates.

The next section defines several commonly used methods and discusses their pros and cons.

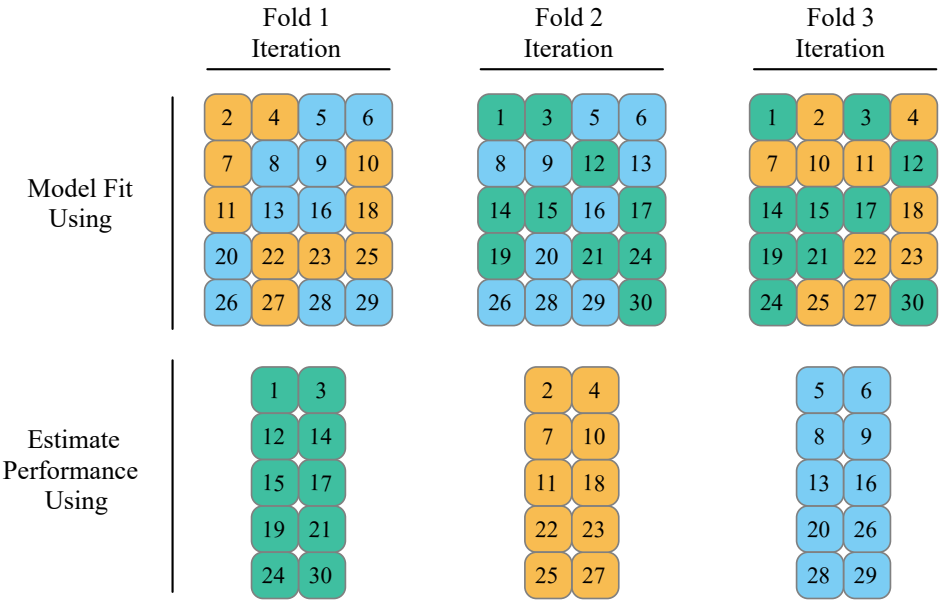
10.2.1 CROSS-VALIDATION

Cross-validation is a well established resampling method. While there are a number of variations, the most common cross-validation method is *V*-fold cross-validation. The data are randomly partitioned into *V* sets of roughly equal size (called the "folds"). For illustration, *V* = 3 is shown below for a data set of thirty training set points with random fold allocations. The number inside the symbols is the sample number:



The color of the symbols represent their randomly assigned folds. Stratified sampling is also an option for assigning folds (previously discussed in Section 5.1).

For 3-fold cross-validation, the three iterations of resampling are illustrated below. For each iteration, one fold is held out for assessment statistics and the remaining folds are substrate for the model. This process continues for each fold so that three models produce three sets of performance statistics.



When *V* = 3, the analysis sets are 2/3 of the training set and each assessment set is a distinct 1/3. The final resampling estimate of performance averages each of the *V* replicates.

Using *V* = 3 is a good choice to illustrate cross-validation but is a poor choice in practice. Values of *V* are most often 5 or 10; we generally prefer 10-fold cross-validation as a default.

> What are the effects of changing V? Larger values result in resampling estimates with small bias but substantial noise. Smaller values of V have large bias but low noise. We prefer 10-fold since noise is reduced by replication, as shown below, but bias is not. See Section **3.4** of Kuhn and Johnson (2020) for a longer description.

The primary input is the training set data frame as well as the number of folds (defaulting to 10):

```
set.seed(55)
ames_folds <- vfold_cv(ames_train, v = 10)
ames_folds
#> #  10-fold cross-validation
#> # A tibble: 10 × 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
#> # … with 4 more rows
```

The column named `splits` contains the information on how to split the data (similar to the object used to create the initial training/test partition). While each row of `splits` has an embedded copy of the entire training set, R is smart enough not to make copies of the data in memory[14]. The print method inside of the tibble shows the frequency of each: `[2K/220]` indicates that roughly two thousand samples are in the analysis set and 220 are in that particular assessment set.

To manually retrieve the partitioned data, the `analysis()` and `assessment()` functions return the corresponding data frames:

```
# For the first fold:
ames_folds$splits[[1]] %>% analysis() %>% dim()
#> [1] 2107   74
```

The **tidymodels** packages, such as **tune**, contain high-level user interfaces so that functions like `analysis()` are not generally needed for day-to-day work. Section 10.3 demonstrates a function to fit a model over these resamples.

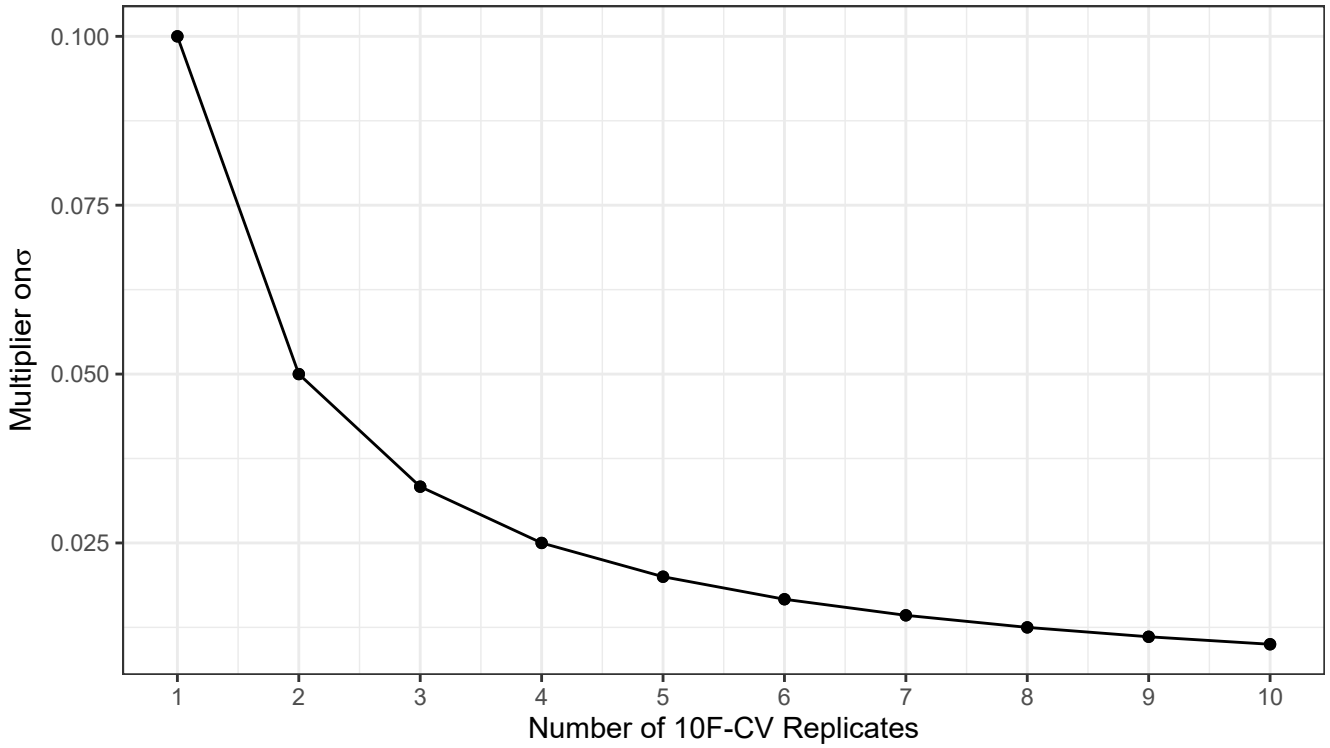
These **rsample** objects also always contain a character column called `id` that labels the partition. Some resampling methods require multiple `id` fields.

REPEATED CROSS-VALIDATION

There are a variety of variations on cross-validation. The most important is *repeated* *V*-fold cross-validation. Depending on the size or other characteristics of the data, the resampling estimate produced by *V*-fold cross-validation may be excessively noisy[15]. As with many statistical problems, one way to reduce noise is to gather more data. For cross-validation, this means averaging more than *V* statistics.

To create *R* repeats of *V*-fold cross-validation, the same fold generation process is done *R* times to generate *R* collections of *V* partitions. Now, instead of averaging *V* statistics, $V \times R$ statistics produce the final resampling estimate. Due to the Central Limit Theorem, the summary statistics from each model tend toward a normal distribution.

Consider the Ames data. On average, 10-fold cross-validation uses assessment sets that contain roughly 234 properties. If RMSE is the statistic of choice, we can denote that estimate's standard deviation as σ. With simple 10-fold cross-validation, the standard error of the mean RMSE is $\sigma/\sqrt{10}$. If this is too noisy, repeats reduce the standard error to $\sigma/\sqrt{10R}$. For 10-fold cross-validation with R replicates, the plot below shows how quickly the standard error[16] decreases with replicates:



Larger number of replicates tend to have less impact on the standard error. However, if the baseline value of σ is impractically large, the diminishing returns on replication may still be worth the extra computational costs.

To create repeats, invoke `vfold_cv()` with an additional argument `repeats`:

```
vfold_cv(ames_train, v = 10, repeats = 5)
#> #  10-fold cross-validation repeated 5 times
#> # A tibble: 50 × 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
#> # … with 44 more rows
```

LEAVE-ONE-OUT CROSS-VALIDATION

One early variation of cross-validation was leave-one-out (LOO) cross-validation where *V* is the number of data points in the training set. If there are n training set samples, n models are fit using $n - 1$ rows of the training set. Each model predicts the single excluded data point. At the end of resampling, the n predictions are pooled to produce a single performance statistic.

Leave-one-out methods are deficient compared to almost any other method. For anything but pathologically small samples, LOO is computationally excessive and it may not have good statistical properties. Although **rsample** contains a `loo_cv()` function, these objects are not generally integrated into the broader tidymodels frameworks.

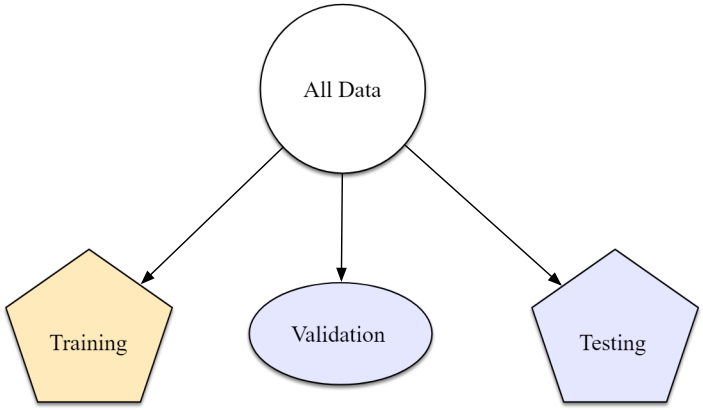
MONTE CARLO CROSS-VALIDATION

Finally, another variant of *V*-fold cross-validation is *Monte Carlo* cross-validation (MCCV, Xu and Liang (2001)). Like *V*-fold cross-validation, it allocates a fixed proportion of data to the assessment sets. The difference is that, for MCCV, this proportion of the data is randomly selected each time. This results in assessment sets that are not mutually exclusive. To create these resampling objects:

```
mc_cv(ames_train, prop = 9/10, times = 20)
#> # Monte Carlo cross-validation (0.9/0.1) with 20 resamples
#> # A tibble: 20 × 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
#> # … with 14 more rows
```

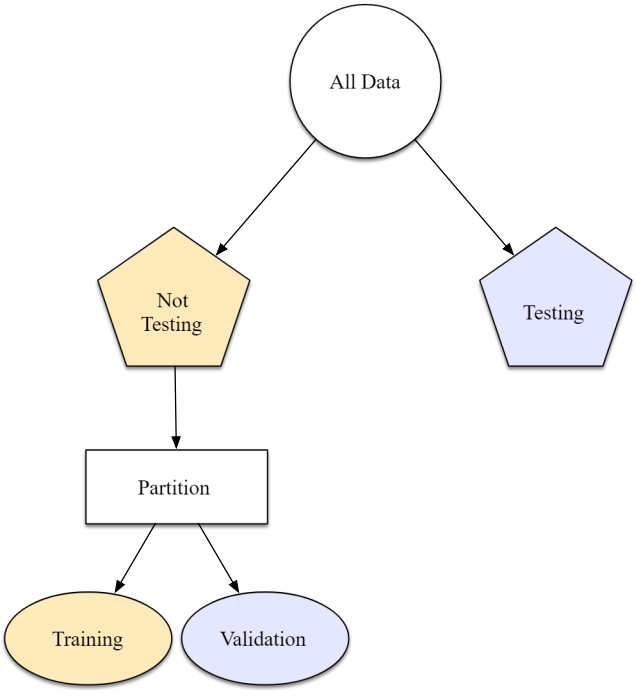
10.2.2 VALIDATION SETS

Previously mentioned in Section 5.3, a validation set is a single partition that is set aside to estimate performance, before using the test set:



Validation sets are often used when the original pool of data is very large. In this case, a single large partition may be adequate to characterize model performance without having to do multiple iterations of resampling.

With **rsample**, a validation set is like any other resampling object; this type is different only in that it has a single iteration[17]:



To create a validation set object that uses 3/4 of the data for model fitting:

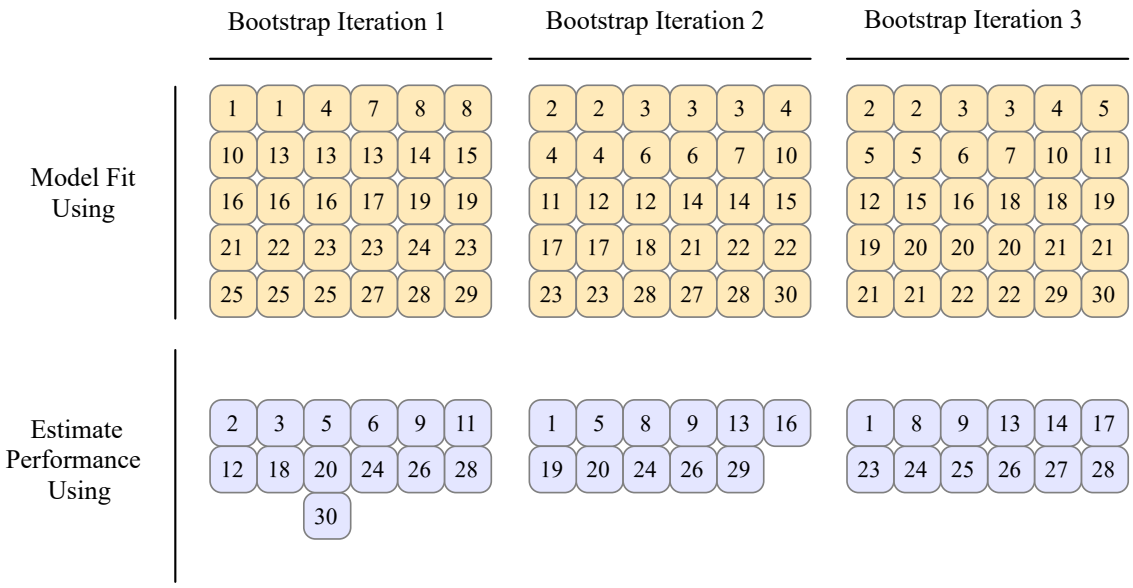
```
set.seed(12)
val_set <- validation_split(ames_train, prop = 3/4)
val_set
#> # Validation Set Split (0.75/0.25)
#> # A tibble: 1 × 2
#>   splits             id
#>   <list>             <chr>
#> 1 <split [1756/586]> validation
```

10.2.3 BOOTSTRAPPING

Bootstrap resampling was originally invented as a method for approximating the sampling distribution of statistics whose theoretical properties are intractable (Davison and Hinkley 1997). Using it to estimate model performance is a secondary application of the method.

A bootstrap sample of the training set is a sample that is the same size as the training set but is drawn *with replacement*. This means that some training set data points are selected multiple times for the analysis set. Each data point has a 63.2% chance of inclusion in the training set *at least once*. The assessment set contains all of the training set samples that were not selected for the analysis set (on average, with 36.8% of the training set). When bootstrapping, the assessment set is often called the "out-of-bag" sample.

For a training set of 30 samples, a schematic of three bootstrap samples is:



Note that the sizes of the assessment sets vary.

Using **rsample**:

```
bootstraps(ames_train, times = 5)
#> # Bootstrap sampling
#> # A tibble: 5 × 2
#>   splits             id
#>   <list>             <chr>
#> 1 <split [2342/830]> Bootstrap1
#> 2 <split [2342/867]> Bootstrap2
#> 3 <split [2342/856]> Bootstrap3
#> 4 <split [2342/849]> Bootstrap4
#> 5 <split [2342/888]> Bootstrap5
```

Bootstrap samples produce performance estimates that have very low variance (unlike cross-validation) but have significant pessimistic bias. This means that, if the true accuracy of a model is 90%, the bootstrap would tend to estimate the value to be less than 90%. The amount of bias cannot be empirically determined with sufficient accuracy. Additionally, the amount of bias changes over the scale of the performance metric. For example, the bias is likely to be different when the accuracy is 90% versus when it is 70%.

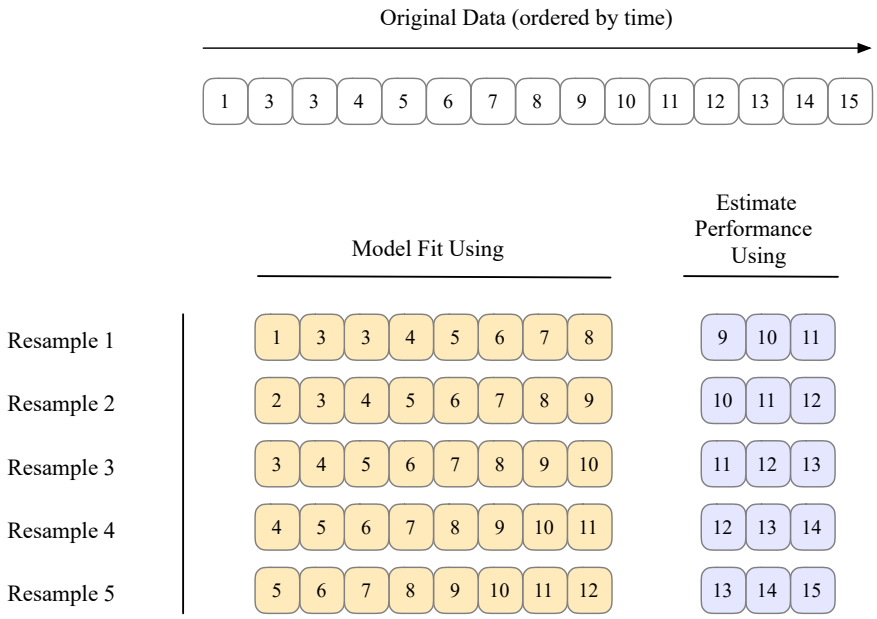
The bootstrap is also used inside of many models. For example, the random forest model mentioned earlier contained 1,000 individual decision trees. Each tree was the product of a different bootstrap sample of the training set.

10.2.4 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. A technique that randomly samples values from the training set can disrupt the model's ability to estimate these patterns.

Rolling forecast origin resampling (Hyndman and Athanasopoulos 2018) provides a method that emulates how time series data is often partitioned in practice, estimating the model with historical data and evaluating it with the most recent data. For this type of resampling, the size of the initial analysis and assessment sets are specified. The first iteration of resampling uses these sizes, starting from the beginning of the series. The second iteration uses the same data sizes but shifts over by a set number of samples.

To illustrate, a training set of fifteen samples was resampled with an analysis size of eight samples and an assessment set size of three. The second iteration discards the first training set sample and both data sets shift forward by one. This configuration results in five resamples:



There are a few different configurations of this method:

- The analysis set can cumulatively grow (as opposed to remaining the same size). After the first initial analysis set, new samples can accrue without discarding the earlier data.
- The resamples need not increment by one. For example, for large data sets, the incremental block could be a week or month instead of a day.

For a year's worth of data, suppose that six sets of 30-day blocks define the analysis set. For assessment sets of 30 days with a 29 day skip, the **rsample** code is:

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

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

map_dfr(time_slices$splits, ~   analysis(.x) %>% data_range())
#> # A tibble: 6 × 2
#>   first  last
#>   <int> <int>
#> 1     1   180
#> 2    31   210
#> 3    61   240
#> 4    91   270
#> 5   121   300
#> 6   151   330
map_dfr(time_slices$splits, ~ assessment(.x) %>% data_range())
#> # A tibble: 6 × 2
#>   first  last
#>   <int> <int>
#> 1   181   210
#> 2   211   240
#> 3   241   270
#> 4   271   300
#> 5   301   330
#> 6   331   360
```

10.3 ESTIMATING PERFORMANCE

Any of these resampling methods can be used to evaluate the modeling process (including preprocessing, model fitting, etc). These methods are effective because different groups of data are used to train the model and assess the model. To reiterate the process:

1. During resampling, the analysis set is used to preprocess the data, apply the preprocessing to itself, and use these processed 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.

This sequence repeats for every resample. If there are *B* resamples, there are *B* replicates of each of the performance metrics. The final resampling estimate is the average of these *B* statistics. If *B* = 1, as with a validation set, the individual statistics represent overall performance.

Let's reconsider the previous random forest model contained in the `rf_wflow` object. The `fit_resamples()` function is analogous to `fit()`, but instead of having a `data` argument, `fit_resamples()` has `resamples` which expects an `rset` object like the ones shown above. The possible interfaces to the function are:

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

There are a number of other optional arguments, such as:

- `metrics`: A metric set of performance statistics to compute. By default, regression models use RMSE and R^2 while classification models compute the area under the ROC curve and overall accuracy. Note that this choice also defines what predictions are produced during the evaluation of the model. For classification, if only accuracy is requested, class probability estimates are not generated for the assessment set (since they are not needed).
- `control`: A list created by `control_resamples()` with various options.

The control arguments include:

- `verbose`: A logical for printing logging.
- `extract`: A function for retaining objects from each model iteration (discussed below).
- `save_pred`: A logical for saving the assessment set predictions.

For our example, let's save the predictions in order to visualize the model fit and residuals:

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


set.seed(130)
rf_res <-
  rf_wflow %>%
  fit_resamples(resamples = ames_folds, control = keep_pred)
rf_res
#> # Resampling results
#> # 10-fold cross-validation
#> # A tibble: 10 × 5
#>   splits             id     .metrics         .notes           .predictions
#>   <list>             <chr>  <list>           <list>           <list>
#> 1 <split [2107/235]> Fold01 <tibble [2 × 4]> <tibble [0 × 1]> <tibble [235 × 4]>
#> 2 <split [2107/235]> Fold02 <tibble [2 × 4]> <tibble [0 × 1]> <tibble [235 × 4]>
#> 3 <split [2108/234]> Fold03 <tibble [2 × 4]> <tibble [0 × 1]> <tibble [234 × 4]>
#> 4 <split [2108/234]> Fold04 <tibble [2 × 4]> <tibble [0 × 1]> <tibble [234 × 4]>
#> 5 <split [2108/234]> Fold05 <tibble [2 × 4]> <tibble [0 × 1]> <tibble [234 × 4]>
#> 6 <split [2108/234]> Fold06 <tibble [2 × 4]> <tibble [0 × 1]> <tibble [234 × 4]>
#> # … with 4 more rows
```

The return value is a tibble similar to the input resamples, along with some extra columns:

- `.metrics` is a list column of tibbles containing the assessment set performance statistics.
- `.notes` is another list column of tibbles cataloging any warnings or errors generated during resampling. Note that errors will not stop subsequent execution of resampling.
- `.predictions` is present when `save_pred = TRUE`. This list column contains tibbles with the out-of-sample predictions.

While these list columns may look daunting, they can be easily reconfigured using **tidyr** or with convenience functions that tidymodels provides. For example, to return the performance metrics in a more usable format:

```
collect_metrics(rf_res)
#> # A tibble: 2 × 6
#>   .metric .estimator   mean     n std_err .config
#>   <chr>   <chr>       <dbl> <int>   <dbl> <chr>
#> 1 rmse    standard   0.0720    10 0.00244 Preprocessor1_Model1
#> 2 rsq     standard   0.835     10 0.00754 Preprocessor1_Model1
```

These are the resampling estimates averaged over the individual replicates. To get the metrics for each resample, use the option `summarize = FALSE`

Notice how much more realistic the performance estimates are than the resubstitution estimates from Section 10.1!

To obtain the assessment set predictions:

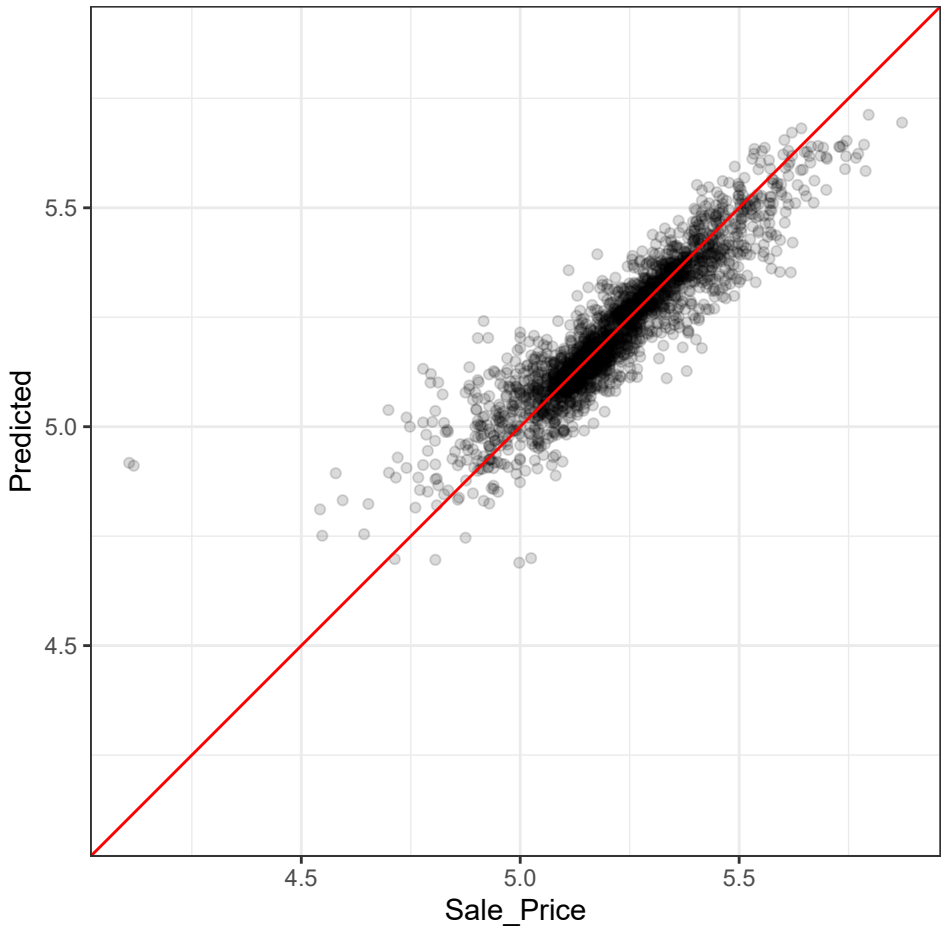
```
assess_res <- collect_predictions(rf_res)
assess_res
#> # A tibble: 2,342 × 5
#>   id     .pred  .row Sale_Price .config
#>   <chr>  <dbl> <int>      <dbl> <chr>
#> 1 Fold01  5.14    12       5.11 Preprocessor1_Model1
#> 2 Fold01  5.15    49       5.10 Preprocessor1_Model1
#> 3 Fold01  4.96    54       4.93 Preprocessor1_Model1
#> 4 Fold01  5.04    84       5.05 Preprocessor1_Model1
#> 5 Fold01  5.19    99       5.05 Preprocessor1_Model1
#> 6 Fold01  5.10   116       4.98 Preprocessor1_Model1
#> # … with 2,336 more rows
```

The prediction column names follow the conventions discussed for **parsnip** models. The observed outcome column always uses the original column name from the source data. The `.row` column is an integer that matches the row of the original training set so that these results can be properly arranged and joined with the original data.

> For some resampling methods, such as the bootstrap or repeated cross-validation, there will be multiple predictions per row of the original training set. To obtain summarized values (averages of the replicate predictions) use `collect_predictions(object, summarize = TRUE)`.

Since this analysis used 10-fold cross-validation, there is one unique prediction for each training set sample. These data can generate helpful plots of the model to understand where it potentially failed. For example, let's compare the observed and predicted values:

```
assess_res %>%
  ggplot(aes(x = Sale_Price, y = .pred)) +
  geom_point(alpha = .15) +
  geom_abline(col = "red") +
  coord_obs_pred() +
  ylab("Predicted")
```



There was one house in the training set with a low observed sale price that is significantly overpredicted by the model. Which house was that?

```
over_predicted <-
  assess_res %>%
  mutate(residual = Sale_Price - .pred) %>%
  arrange(desc(abs(residual))) %>%
  slice(1)
over_predicted
#> # A tibble: 1 × 6
#>   id     .pred  .row Sale_Price .config              residual
#>   <chr>  <dbl> <int>      <dbl> <chr>                   <dbl>
#> 1 Fold07  4.92    32       4.11 Preprocessor1_Model1   -0.811

ames_train %>%
  slice(over_predicted$.row) %>%
  select(Gr_Liv_Area, Neighborhood, Year_Built, Bedroom_AbvGr, Full_Bath)
#> # A tibble: 1 × 5
#>   Gr_Liv_Area Neighborhood Year_Built Bedroom_AbvGr Full_Bath
#>         <int> <fct>             <int>         <int>     <int>
#> 1         832 Old_Town           1923             2         1
```

These results can help us investigate why the prediction was poor for this house.

How can we use a validation set instead of cross-validation? From our previous **rsample** object:

```
val_res <- rf_wflow %>% fit_resamples(resamples = val_set)
val_res
#> # Resampling results
#> # Validation Set Split (0.75/0.25)
#> # A tibble: 1 × 4
#>   splits             id         .metrics         .notes
#>   <list>             <chr>      <list>           <list>
#> 1 <split [1756/586]> validation <tibble [2 × 4]> <tibble [0 × 1]>

collect_metrics(val_res)
#> # A tibble: 2 × 6
#>   .metric .estimator   mean     n std_err .config
#>   <chr>   <chr>       <dbl> <int>   <dbl> <chr>
#> 1 rmse    standard   0.0775     1      NA Preprocessor1_Model1
#> 2 rsq     standard   0.815      1      NA Preprocessor1_Model1
```

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

> In these analyses, the resampling results are very close to the test set results. The two types of estimates tend to be well correlated. However, this could be from random chance. A seed value of `55` fixed the random numbers before creating the resamples. Try changing this value and re-running the analyses to investigate whether the resampled estimates match the test set results as well.

10.4 PARALLEL PROCESSING

The models created during resampling are independent of one another. Computations of this kind are sometimes called “embarrassingly parallel”; each model *could* be fit simultaneously without issues. The **tune** package uses the **foreach** package to facilitate parallel computations. These computations could be split across processors on the same computer or across different computers, depending on the chosen technology.

For computations conducted on a single computer, the number of possible "worker processes" is determined by the **parallel** package:

```
# The number of physical cores in the hardware:
parallel::detectCores(logical = FALSE)
#> [1] 3

# The number of possible independent processes that can
# be simultaneously used:
parallel::detectCores(logical = TRUE)
#> [1] 3
```

The difference between these two values is related to the computer's processor. For example, most Intel processors use hyper-threading which creates two *virtual cores* for each physical core. While these extra resources can improve performance, most of the speed-ups produced by parallel processing occur when processing uses fewer than the number of physical cores.

For `fit_resamples()` and other functions in **tune**, parallel processing occurs when the user registers a *parallel backend package*. These R packages define how to execute parallel processing. On Unix and macOS operating systems, one method of splitting computations is by forking threads. To enable this, load the **doMC** package and register the number of parallel cores with **foreach**:

```
# Unix and macOS only
library(doMC)
registerDoMC(cores = 2)

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

This instructs `fit_resamples()` to run half of the computations on each of two cores. To reset the computations to sequential processing:

```
registerDoSEQ()
```

Alternatively, a different approach to parallelizing computations uses network sockets. The **doParallel** package enables this method (usable by all operating systems):

```
# All operating systems
library(doParallel)

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

# Now run fit_resamples()`...

stopCluster(cl)
```

Another R package that facilitates parallel processing is the **future** package. Like **foreach**, it provides a framework for parallelism. It is used in conjunction with **foreach** via the **doFuture** package.

> The R packages with parallel backends for **foreach** start with the prefix `"do"`.

Parallel processing with **tune** tends to provide linear speed-ups for the first few cores. This means that, with two cores, the computations are twice as fast. Depending on the data and type of model, the linear speedup deteriorates after 4-5 cores. Using more cores will still reduce the time it takes to complete the task; there are just diminishing returns for the additional cores.

Let's wrap up with one final note about parallelism. For each of these technologies, the memory requirements multiply for each additional core used. For example, if the current data set is 2 GB in memory and three cores are used, the total memory requirement is 8 GB (2 for each worker process plus the original). Using too many cores might cause the computations (and the computer) to slow considerably.

Schmidberger et al. (2009) gives a technical overview of these technologies.

10.5 SAVING THE RESAMPLED OBJECTS

The models created during resampling are not retained. These models are trained for the purpose of evaluating performance, and we typically do not need them after we have computed performance statistics. If a particular modeling approach does turn out to be the best option for our data set, then the best choice is to fit again to the whole training set so the model parameters can be estimated with more data.

While these models created during resampling are not preserved, there is a method for keeping them or some of their components. The `extract` option of `control_resamples()` specifies a function that takes a single argument; we'll use `x`. When executed, `x` results in a fitted workflow object, regardless of whether you provided `fit_resamples()` with a workflow. Recall that the **workflows** package has functions that can pull the different components of the objects (e.g. the model, recipe, etc.).

Let's fit a linear regression model using the recipe shown at the end of Chapter 8:

```
ames_rec <-
  recipe(Sale_Price ~ Neighborhood + Gr_Liv_Area + Year_Built + Bldg_Type +
           Latitude + Longitude, data = ames_train) %>%
  step_other(Neighborhood, threshold = 0.01) %>%
  step_dummy(all_nominal_predictors()) %>%
  step_interact( ~ Gr_Liv_Area:starts_with("Bldg_Type_") ) %>%
  step_ns(Latitude, Longitude, deg_free = 20)

lm_wflow <-
  workflow() %>%
  add_recipe(ames_rec) %>%
  add_model(linear_reg() %>% set_engine("lm"))

lm_fit <- lm_wflow %>% fit(data = ames_train)

# Select the recipe:
extract_recipe(lm_fit, estimated = TRUE)
#> Recipe
#>
#> Inputs:
#>
#>       role #variables
#>    outcome          1
#>  predictor          6
#>
#> Training data contained 2342 data points and no missing data.
#>
#> Operations:
#>
#> Collapsing factor levels for Neighborhood [trained]
#> Dummy variables from Neighborhood, Bldg_Type [trained]
#> Interactions with Gr_Liv_Area:(Bldg_Type_TwoFmCon + Bldg_Type_Duplex + Bldg_Type_Twnhs +
#> Natural Splines on Latitude, Longitude [trained]
```

We can save the linear model coefficients for a fitted model object from a workflow:

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

# Test it using:
# get_model(lm_fit)
```

Now let's apply this function to the ten resampled fits. The results of the extraction function is wrapped in a list object and returned in a tibble:

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

lm_res <- lm_wflow %>%  fit_resamples(resamples = ames_folds, control = ctrl)
lm_res
#> # Resampling results
#> # 10-fold cross-validation
#> # A tibble: 10 × 5
#>   splits             id     .metrics         .notes           .extracts
#>   <list>             <chr>  <list>           <list>           <list>
#> 1 <split [2107/235]> Fold01 <tibble [2 × 4]> <tibble [0 × 1]> <tibble [1 × 2]>
#> 2 <split [2107/235]> Fold02 <tibble [2 × 4]> <tibble [0 × 1]> <tibble [1 × 2]>
#> 3 <split [2108/234]> Fold03 <tibble [2 × 4]> <tibble [0 × 1]> <tibble [1 × 2]>
#> 4 <split [2108/234]> Fold04 <tibble [2 × 4]> <tibble [0 × 1]> <tibble [1 × 2]>
#> 5 <split [2108/234]> Fold05 <tibble [2 × 4]> <tibble [0 × 1]> <tibble [1 × 2]>
#> 6 <split [2108/234]> Fold06 <tibble [2 × 4]> <tibble [0 × 1]> <tibble [1 × 2]>
#> # … with 4 more rows
```

Now there is a `.extracts` column with nested tibbles. What do these contain?

```
lm_res$.extracts[[1]]
#> # A tibble: 1 × 2
#>   .extracts         .config
#>   <list>            <chr>
#> 1 <tibble [73 × 5]> Preprocessor1_Model1

# To get the results
lm_res$.extracts[[1]][[1]]
#> [[1]]
#> # A tibble: 73 × 5
#>   term                         estimate  std.error statistic   p.value
#>   <chr>                           <dbl>      <dbl>     <dbl>     <dbl>
#> 1 (Intercept)                  1.29     0.323         3.99   6.86e-  5
#> 2 Gr_Liv_Area                  0.000159 0.00000464   34.4    1.19e-204
#> 3 Year_Built                   0.00188  0.000148     12.7    1.25e- 35
#> 4 Neighborhood_College_Creek -0.00264  0.0375       -0.0706 9.44e-  1
#> 5 Neighborhood_Old_Town      -0.0834   0.0143       -5.85   5.86e-  9
#> 6 Neighborhood_Edwards       -0.108    0.0311       -3.48   5.04e-  4
#> # … with 67 more rows
```

This might appear to be a convoluted method for saving the model results. However, `extract` is flexible and does not assume that the user will only save a single tibble per resample. For example, the `tidy()` method might be run on the recipe as well as the model. In this case, a list of two tibbles will be returned.

For our more simple example, all of the results can be flattened and collected using:

```
all_coef <- map_dfr(lm_res$.extracts, ~ .x[[1]][[1]])
# Show the replicates for a single predictor:
filter(all_coef, term == "Year_Built")
#> # A tibble: 10 × 5
#>   term       estimate std.error statistic  p.value
#>   <chr>         <dbl>     <dbl>     <dbl>    <dbl>
#> 1 Year_Built  0.00188  0.000148      12.7 1.25e-35
#> 2 Year_Built  0.00184  0.000149      12.3 8.30e-34
#> 3 Year_Built  0.00185  0.000147      12.5 8.52e-35
#> 4 Year_Built  0.00192  0.000153      12.5 8.08e-35
#> 5 Year_Built  0.00183  0.000147      12.5 2.38e-34
#> 6 Year_Built  0.00194  0.000148      13.2 5.61e-38
#> # … with 4 more rows
```

Chapters 13 and 14 discuss a suite of functions for tuning models. Their interfaces are similar to `fit_resamples()` and many of the features described here apply to those functions.

10.6 CHAPTER SUMMARY

This chapter describes one of the fundamental tools of data analysis, the ability to measure the performance and variation in model results. Resampling enables us to determine how well the model works without using the test set.

An important function from the **tune** package, called `fit_resamples()`, was introduced. The interface for this function is also used in future chapters that describe model tuning tools.

The data analysis code, to date, for the Ames data is:

```
library(tidymodels)
data(ames)
ames <- mutate(ames, Sale_Price = log10(Sale_Price))

set.seed(123)
ames_split <- initial_split(ames, prop = 0.80, strata = Sale_Price)
ames_train <- training(ames_split)
ames_test  <-  testing(ames_split)

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_dummy(all_nominal_predictors()) %>%
  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)

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)

set.seed(55)
ames_folds <- vfold_cv(ames_train, v = 10)

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

set.seed(130)
rf_res <- rf_wflow %>% fit_resamples(resamples = ames_folds, control = keep_pred)
```

REFERENCES

Breiman, L. 2001a. “Random Forests.” *Machine Learning* 45 (1): 5–32.

Davison, A, and D Hinkley. 1997. *Bootstrap Methods and Their Application*. Vol. 1. Cambridge university press.

Hyndman, R, and G Athanasopoulos. 2018. *Forecasting: Principles and Practice*. OTexts.

Kuhn, M, and K Johnson. 2020. *Feature Engineering and Selection: A Practical Approach for Predictive Models*. CRC Press.

Schmidberger, M, M Morgan, D Eddelbuettel, H Yu, L Tierney, and U Mansmann. 2009. “State of the Art in Parallel Computing with R.” *Journal of Statistical Software* 31 (1): 1–27. https://www.jstatsoft.org/v031/i01.

Xu, Q, and Y Liang. 2001. “Monte Carlo Cross Validation.” *Chemometrics and Intelligent Laboratory Systems* 56 (1): 1–11.

13. It is possible for a linear model to nearly memorize the training set, like the random forest model did. In the `ames_rec` object, change the number of spline terms for `longitude` and `latitude` to a large number (say 1000). This would produce a model fit with a very small

resubstitution RMSE and a test set RMSE that is much larger.↩

14. To see this for yourself, try executing `lobstr::obj_size(ames_folds)` and `lobstr::obj_size(ames_train)`. The size of the resample object is much less than ten times the size of the original data.↩

15. For more details, see Section 3.4.6 of Kuhn and Johnson (2020).↩

16. These are *approximate* standard errors. As will be discussed in the next chapter, there is a within-replicate correlation that is typical of resampled results. By ignoring this extra component of variation, the simple calculations shown in this plot are overestimates of the reduction in noise in the standard errors.↩

17. In essence, a validation set can be considered a single iteration of Monte Carlo cross-validation.↩