

6 Fitting models with parsnip

The **parsnip** package provides a fluent and standardized interface for a variety of different models. In this chapter, we both give some motivation for why a common interface is beneficial and show how to use the package.

In this chapter, we focus on how to `fit()` and `predict()` directly with a **parsnip** object. This may be a good fit for some straightforward modeling problems, but the next chapter illustrates a better approach to these tasks by combining models and preprocessors together into something called a `workflow` object.

6.1 CREATE A MODEL

Once the data have been encoded in a format ready for a modeling algorithm, such as a numeric matrix, they can be used in the model building process.

Suppose that a linear regression model was our initial choice for the model. This is equivalent to specifying that the outcome data is numeric and that the predictors are related to the outcome in terms of simple slopes and intercepts:

$$y_i = \beta_0 + \beta_1 x_{1i} + \dots + \beta_p x_{pi}$$

There are a variety of methods that can be used to estimate the model parameters:

- *Ordinary linear regression* uses the traditional method of least squares to solve for the model parameters.
- *Regularized linear regression* adds a penalty to the least squares method to encourage simplicity by removing predictors and/or shrinking their coefficients towards zero. This can be executed using Bayesian or non-Bayesian techniques.

In R, the **stats** package can be used for the first case. The syntax for `lm()` is

```
model <- lm(formula, data, ...)
```

where `...` symbolizes other options to pass to `lm()`. The function does *not* have an `x / y` interface, where we might pass in our outcome as `y` and our predictors as `x`.

To estimate with regularization, a Bayesian model can be fit using the **rstanarm** package:

```
model <- stan_glm(formula, data, family = "gaussian", ...)
```

In this case, the other options passed via `...` would include arguments for the *prior distributions* of the parameters as well as specifics about the numerical aspects of the model. As with `lm()`, only the formula interface is available.

A popular non-Bayesian approach to regularized regression is the `glmnet` model (Friedman, Hastie, and Tibshirani 2010). Its syntax is:

```
model <- glmnet(x = matrix, y = vector, family = "gaussian", ...)
```

In this case, the predictor data must already be formatted into a numeric matrix; there is only an `x / y` method and no formula method.

Note that these interfaces are heterogeneous in either how the data are passed to the model function or in terms of their arguments. The first issue is that, to fit models across different packages, the data must be formatted in different ways. `lm()` and `stan_glm()` only have formula interfaces while `glmnet()` does not. For other types of models, the interfaces may be even more disparate. For a person trying to do data analysis, these differences require the memorization of each package's syntax and can be very frustrating.

For tidymodels, the approach to specifying a model is intended to be more unified:

1. **Specify the type of model based on its mathematical structure** (e.g., linear regression, random forest, K -nearest neighbors, etc).

2. **Specify the *engine* for fitting the model.** Most often this reflects the software package that should be used.
3. **When required, declare the *mode* of the model.** The mode reflects the type of prediction outcome. For numeric outcomes, the mode is *regression*; for qualitative outcomes, it is *classification*¹⁰. If a model can only create one type of model, such as linear regression, the mode is already set.

These specifications are built *without referencing the data*. For example, for the three cases above:

```
library(tidymodels)
tidymodels_prefer()

linear_reg() %>% set_engine("lm")
#> Linear Regression Model Specification (regression)
#>
#> Computational engine: lm
```

```
linear_reg() %>% set_engine("glmnet")
#> Linear Regression Model Specification (regression)
#>
#> Computational engine: glmnet
```

```
linear_reg() %>% set_engine("stan")
#> Linear Regression Model Specification (regression)
#>
#> Computational engine: stan
```

Once the details of the model have been specified, the model estimation can be done with either the `fit()` function (to use a formula) or the `fit_xy()` function (when your data are already pre-processed). The **parsnip** package allows the user to be indifferent to the interface of the underlying model; you can always use a formula even if the modeling package's function only has the `x / y` interface.

The `translate()` function can provide details on how **parsnip** converts the user's code to the package's syntax:

```
linear_reg() %>% set_engine("lm") %>% translate()
#> Linear Regression Model Specification (regression)
#>
#> Computational engine: lm
#>
#> Model fit template:
#> stats::lm(formula = missing_arg(), data = missing_arg(), weights = missing_arg())
```

```
linear_reg(penalty = 1) %>% set_engine("glmnet") %>% translate()
#> Linear Regression Model Specification (regression)
#>
#> Main Arguments:
#>   penalty = 1
#>
#> Computational engine: glmnet
#>
#> Model fit template:
#> glmnet::glmnet(x = missing_arg(), y = missing_arg(), weights = missing_arg(),
#>   family = "gaussian")
```

```
linear_reg() %>% set_engine("stan") %>% translate()
#> Linear Regression Model Specification (regression)
#>
#> Computational engine: stan
#>
#> Model fit template:
#> rstanarm::stan_glm(formula = missing_arg(), data = missing_arg(),
#>   weights = missing_arg(), family = stats::gaussian, refresh = 0)
```

Note that `missing_arg()` is just a placeholder for the data that has yet to be provided.

Note that we supplied a required `penalty` argument for the `glmnet` engine. Also, for the `Stan` and `glmnet` engines, the `family` argument was automatically added as a default. As will be shown below, this option can be changed.

Let's walk through how to predict the sale price of houses in the Ames data as a function of only longitude and latitude:

```

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

lm_form_fit <-
  lm_model %>%
  # Recall that Sale_Price has been pre-logged
  fit(Sale_Price ~ Longitude + Latitude, data = ames_train)

lm_xy_fit <-
  lm_model %>%
  fit_xy(
    x = ames_train %>% select(Longitude, Latitude),
    y = ames_train %>% pull(Sale_Price)
  )

lm_form_fit
#> parsnip model object
#>
#> Fit time: 4ms
#>
#> Call:
#> stats::lm(formula = Sale_Price ~ Longitude + Latitude, data = data)
#>
#> Coefficients:
#> (Intercept) Longitude Latitude
#> -300.25 -2.01 2.78

lm_xy_fit
#> parsnip model object
#>
#> Fit time: 1ms
#>
#> Call:
#> stats::lm(formula = ..y ~ ., data = data)
#>

```

```
#> Coefficients:
#> (Intercept)    Longitude    Latitude
#>      -300.25        -2.01         2.78
```

The differences between `fit()` and `fit_xy()` may not be obvious.

When `fit()` is used with a model specification, this *almost always* means that dummy variables will be created from qualitative predictors. If the underlying function requires a matrix (like `glmnet`), it will make them. However, if the underlying function uses a formula, `fit()` just passes the formula to that function. We estimate that 99% of modeling functions using formulas make dummy variables. The other 1% include tree-based methods that do not require purely numeric predictors.¹¹

The `fit_xy()` function always passes the data as-is to the underlying model function. It will not create dummy variables before doing so.

Not only does **parsnip** enable a consistent model interface for different packages, it also provides consistency in the *model arguments*. It is common for different functions which fit the same model to have different argument names. Random forest model functions are a good example. Three commonly used arguments are the number of trees in the ensemble, the number of predictors to randomly sample with each split within a tree, and the number of data points required to make a split. For three different R packages implementing this algorithm, those arguments are:

Argument Type	ranger	randomForest	sparklyr
# sampled predictors	mtry	mtry	feature_subset_strategy
# trees	num.trees	ntree	num_trees
# data points to split	min.node.size	nodesize	min_instances_per_node

In an effort to make argument specification less painful, **parsnip** uses common argument names within and between packages. For random forests, **parsnip** models use:

Argument Type	parsnip
# sampled predictors	mtry
# trees	trees
# data points to split	min_n

Admittedly, this is one more set of arguments to memorize. However, when other types of models have the same argument types, these names still apply. For example, boosted tree ensembles also create a large number of tree-based models, so `trees` is also used there, as is `min_n`, and so on.

Some of the original argument names can be fairly jargon-y. For example, to specify the amount of regularization to use in a `glmnet` model, the Greek letter `lambda` is used. While this mathematical notation is commonly used in the statistics literature, it is not obvious to many people what `lambda` represents (especially those who consume the model results). Since this is the penalty used in regularization, **parsnip** standardizes on the argument name `penalty`. Similarly, the number of neighbors in a *K*-nearest neighbors model is called `neighbors` instead of `k`. Our rule of thumb when standardizing argument names is:

If a practitioner were to include these names in a plot or table, would the people viewing those results understand the name?

To understand how the **parsnip** argument names map to the original names, use the help file for the model (available via `?rand_forest`) as well as the `translate()` function:


```

rand_forest(trees = 1000, min_n = 5) %>%
  set_engine("ranger") %>%
  set_mode("regression") %>%
  translate()
#> Random Forest Model Specification (regression)
#>
#> Main Arguments:
#>   trees = 1000
#>   min_n = 5
#>
#> Computational engine: ranger
#>
#> Model fit template:
#> ranger::ranger(x = missing_arg(), y = missing_arg(), case.weights = missing_arg(),
#>   num.trees = 1000, min.node.size = min_rows(~5, x), num.threads = 1,
#>   verbose = FALSE, seed = sample.int(10^5, 1))

```

Modeling functions in **parsnip** separate model arguments into two categories:

- *Main arguments* are more commonly used and tend to be available across engines.
- *Engine arguments* are either specific to a particular engine or used more rarely.

For example, in the translation of the random forest code above, the arguments `num.threads`, `verbose`, and `seed` were added by default. These arguments are specific to the `ranger` implementation of random forest models and wouldn't make sense as main arguments. Engine-specific arguments can be specified in `set_engine()`. For example, to have the `ranger::ranger()` function print out more information about the fit:

```

rand_forest(trees = 1000, min_n = 5) %>%
  set_engine("ranger", verbose = TRUE) %>%
  set_mode("regression")
#> Random Forest Model Specification (regression)
#>
#> Main Arguments:
#>   trees = 1000
#>   min_n = 5
#>
#> Engine-Specific Arguments:
#>   verbose = TRUE
#>
#> Computational engine: ranger

```

6.2 USE THE MODEL RESULTS

Once the model is created and fit, we can use the results in a variety of ways; we might want to plot, print, or otherwise examine the model output. Several quantities are stored in a **parsnip** model object, including the fitted model. This can be found in an element called `fit`, which can be returned using the `extract_fit_engine()` function:

```

lm_form_fit %>% extract_fit_engine()
#>
#> Call:
#> stats::lm(formula = Sale_Price ~ Longitude + Latitude, data = data)
#>
#> Coefficients:
#> (Intercept)   Longitude   Latitude
#>   -300.25      -2.01       2.78

```

Normal methods can be applied to this object, such as printing, plotting, and so on:

```
lm_form_fit %>% extract_fit_engine() %>% vcov()
#>           (Intercept) Longitude Latitude
#> (Intercept)    212.621   1.6113032 -1.4686377
#> Longitude       1.611   0.0168166 -0.0008695
#> Latitude        -1.469 -0.0008695   0.0330019
```

Never pass the `fit` element of a **parsnip** model to a model prediction function, i.e., use `predict(lm_form_fit)` but **do not** use `predict(lm_form_fit$fit)`. If the data were preprocessed in any way, incorrect predictions will be generated (sometimes, without errors). The underlying model's prediction function has no idea if any transformations have been made to the data prior to running the model. See Section 6.3 for more on making predictions.

One issue with some existing methods in base R is that the results are stored in a manner that may not be the most useful. For example, the `summary()` method for `lm` objects can be used to print the results of the model fit, including a table with parameter values, their uncertainty estimates, and p-values. These particular results can also be saved:

```
model_res <-
  lm_form_fit %>%
  extract_fit_engine() %>%
  summary()

# The model coefficient table is accessible via the `coef` method.
param_est <- coef(model_res)
class(param_est)
#> [1] "matrix" "array"
param_est
#>           Estimate Std. Error t value Pr(>|t|)
#> (Intercept) -300.251    14.5815  -20.59 1.023e-86
#> Longitude    -2.013     0.1297  -15.53 8.126e-52
#> Latitude      2.782     0.1817   15.31 1.639e-50
```

There are a few things to notice about this result. First, the object is a numeric matrix. This data structure was mostly likely chosen since all of the calculated results are numeric and a matrix object is stored more efficiently than a data frame. This choice was probably made in the late 1970's when computational efficiency was extremely critical. Second, the non-numeric data (the labels for the coefficients) are contained in the row names. Keeping the parameter labels as row names is very consistent with the conventions in the original S language.

A reasonable next step might be to create a visualization of the parameters values. To do this, it would be sensible to convert the parameter matrix to a data frame. We could add the row names as a column so that they can be used in a plot. However, notice that several of the existing matrix column names would not be valid R column names for ordinary data frames (e.g. `"Pr(>|t|)"`). Another complication is the consistency of the column names. For `lm` objects, the column for the test statistic is `"Pr(>|t|)"`, but for other models, a different test might be used and, as a result, the column name would be different (e.g., `"Pr(>|z|)"`) and the type of test would be *encoded in the column name*.

While these additional data formatting steps are not impossible to overcome, they are a hindrance, especially since they might be different for different types of models. The matrix is not a highly reusable data structure mostly because it constrains the data to be of a single type (e.g. numeric). Additionally, keeping some data in the dimension names is also problematic since those data must be extracted to be of general use.

As a solution, the **broom** package has methods to convert many types of model objects to a tidy structure. For example, using the `tidy()` method on the linear model produces:

```
tidy(lm_form_fit)
#> # A tibble: 3 × 5
#>   term          estimate std.error statistic  p.value
#>   <chr>          <dbl>    <dbl>    <dbl>    <dbl>
#> 1 (Intercept) -300.      14.6     -20.6 1.02e-86
#> 2 Longitude    -2.01      0.130    -15.5 8.13e-52
#> 3 Latitude      2.78      0.182     15.3 1.64e-50
```

The column names are standardized across models and do not contain any additional data (such as the type of statistical test). The data previously contained in the row names are now in a column called `term` and so on. One important principle in the tidymodels ecosystem is that a function should return values that are *predictable, consistent, and unsurprising*.

6.3 MAKE PREDICTIONS

Another area where **parsnip** diverges from conventional R modeling functions is the format of values returned from `predict()`. For predictions, **parsnip** always conforms to the following rules:

1. The results are always a tibble.
2. The column names of the tibble are always predictable.
3. There are always as many rows in the tibble as there are in the input data set.

For example, when numeric data are predicted:

```
ames_test_small <- ames_test %>% slice(1:5)
predict(lm_form_fit, new_data = ames_test_small)
#> # A tibble: 5 × 1
#>   .pred
#>   <dbl>
#> 1  5.22
#> 2  5.22
#> 3  5.28
#> 4  5.24
#> 5  5.31
```

The row order of the predictions are always the same as the original data.

Why are there leading dot in some of the column names? Some tidyverse and tidymodels arguments and return values contain periods. This is to protect against merging data with duplicate names. There are some data sets that contain predictors named `pred` !

These three rules make it easier to merge predictions with the original data:

```
ames_test_small %>%
  select(Sale_Price) %>%
  bind_cols(predict(lm_form_fit, ames_test_small)) %>%
  # Add 95% prediction intervals to the results:
  bind_cols(predict(lm_form_fit, ames_test_small, type = "pred_int"))

#> # A tibble: 5 × 4
#>   Sale_Price .pred .pred_lower .pred_upper
#>   <dbl> <dbl>      <dbl>      <dbl>
#> 1    5.02  5.22      4.91      5.54
#> 2    5.24  5.22      4.91      5.54
#> 3    5.28  5.28      4.97      5.60
#> 4    5.06  5.24      4.92      5.56
#> 5    5.60  5.31      5.00      5.63
```

The motivation for the first rule comes from some R packages producing dissimilar data types from prediction functions. For example, the **ranger** package is an excellent tool for computing random forest models. However, instead of returning a data frame or vector as output, a specialized object is returned that has multiple values embedded within it (including the predicted values). This is just one more step for the data analyst to work around in their scripts. As another example, the **glmnet** package can return at least four different output types for predictions, depending on the model and characteristics of the data:

Type of Prediction	Returns a:
numeric	numeric matrix
class	<i>character</i> matrix
probability (2 classes)	numeric matrix (2nd level only)
probability (3+ classes)	3D numeric array (all levels)

Additionally, the column names of the results contain coded values that map to a vector called `lambda` within the `glmnet` model object. This excellent statistical method can be discouraging to use in practice because of all of the special cases an analyst might encounter that require additional code to be useful.

For the second tidymodels prediction rule, the predictable column names for different types of predictions are:

type value	column name(s)
numeric	.pred
class	.pred_class
prob	.pred_{class levels}
conf_int	.pred_lower , .pred_upper
pred_int	.pred_lower , .pred_upper

The third rule regarding the number of rows in the output is critical. For example, if any rows of the new data contain missing values, the output will be padded with missing results for those rows. A main advantage of standardizing the model interface and prediction types in **parsnip** is that, when different models are used, the syntax is identical. Suppose that we used a decision tree to model the Ames data. Outside of the model specification, there are no significant differences in the code pipeline:

```
tree_model <-  
  decision_tree(min_n = 2) %>%  
  set_engine("rpart") %>%  
  set_mode("regression")  
  
tree_fit <-  
  tree_model %>%  
  fit(Sale_Price ~ Longitude + Latitude, data = ames_train)  
  
ames_test_small %>%  
  select(Sale_Price) %>%  
  bind_cols(predict(tree_fit, ames_test_small))  
  
#> # A tibble: 5 × 2  
#>   Sale_Price .pred  
#>   <dbl> <dbl>  
#> 1     5.02  5.15  
#> 2     5.24  5.15  
#> 3     5.28  5.31  
#> 4     5.06  5.15  
#> 5     5.60  5.52
```

This demonstrates the benefit of homogenizing the data analysis process and syntax across different models. It enables the user to spend their time on the results and interpretation rather than having to focus on the syntactical differences between R packages.

6.4 PARSNIP-ADJACENT PACKAGES

The **parsnip** package itself contains interfaces to a number of models. However, for ease of package installation and maintenance, there are other tidymodels packages that have **parsnip** model definitions for other sets of models. The **discrim** package has model definitions for the set of classification techniques called *discriminant analysis* methods (such as linear or quadratic discriminant analysis). In this way, the package dependencies required for installing **parsnip** are reduced. A list of *all* of the models that can be used with **parsnip** (across different packages that are on CRAN) can be found at [tidymodels.org/find](https://www.tidymodels.org/find).

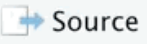

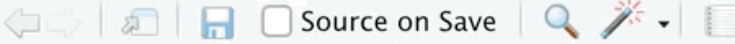
6.5 CREATING MODEL SPECIFICATIONS

It may become tedious to write many model specifications, or to remember how to write the code to generate them. The **parsnip** package includes an **RStudio addin** that can help. Either choosing this addin from the *Addins* toolbar menu or running the code:

```
parsnip_addin()
```

will open a window in the Viewer panel of the RStudio IDE with a list of possible models for each model mode. These can be written to the source code panel:

Untitled1* x



1

1:1 (Top Level) R Script

Files Plots Packages Help Viewer

Cancel Write out model specifications Done

Type of Model

☒ Classification
☐ Regression
☐ Tag parameters for tuning (if any)?

☐ bag_mars (earth)
☐ bag_tree (C5.0)
☐ bag_tree (rpart)
☐ boost_tree (C5.0)
☐ boost_tree (xgboost)
☐ C5_rules (C5.0)
☐ decision_tree (C5.0)
☐ decision_tree (rpart)
☐ discrim_flexible (earth)
☐ discrim_linear (MASS)
... ..

Match on (regex)

Write specification code

The model list includes models from **parsnip** and **parsnip**-adjacent packages that are on CRAN.

6.6 CHAPTER SUMMARY

This chapter introduced the **parsnip** package, which provides a common interface for models across R packages using a standard syntax. The interface and resulting objects have a predictable structure.

The code for modeling the Ames data that we will use moving forward 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)

lm_model <- linear_reg() %>% set_engine("lm")
```

REFERENCES

Friedman, J, T Hastie, and R Tibshirani. 2010. "Regularization Paths for Generalized Linear Models via Coordinate Descent." *Journal of Statistical Software* 33 (1): 1.

10. Note that **parsnip** constrains the outcome column of a classification model to be encoded as a *factor*; using binary numeric values will result in an error.↩
11. See Section 7.4 for more about using formulas in tidymodels.↩

