7 A model workflow

In the previous chapter, we discussed the **parsnip** package, which can be used to define and fit the model. This chapter introduces a new object called a *model workflow*. The purpose of this object is to encapsulate the major pieces of the modeling *process* (previously discussed in Section 1.3). The workflow is important in two ways. First, using a workflow object encourages good methodology since it is a single point of entry to the estimation components of a data analysis. Second, it enables the user to better organize their projects. These two points are discussed in the following sections.

7.1 WHERE DOES THE MODEL BEGIN AND END?

So far, when we have used the term "the model", we have meant a structural equation that relates some predictors to one or more outcomes. Let's consider again linear regression as an example. The outcome data are denoted as y_i, where there are $i = 1 \ldots n$ samples in the training set. Suppose that there are p predictors $x_{i1}, \ldots, x_{ip}$ that are used in the model. Linear regression produces a model equation of

$$\hat{y}_i = \hat{\beta}_0 + \hat{\beta}_1 x_{i1} + \ldots + \hat{\beta}_p x_{ip}$$

While this is a *linear* model, it is only linear in the parameters. The predictors could be nonlinear terms (such as the $log(x_i)$).

> The conventional way of thinking about the modeling process is that it only includes the model fit.

For some data sets that are straightforward in nature, fitting the model itself may be the entire process. However, there are a variety of choices and additional steps that often occur *before* the model is fit:

- While our example model has p predictors, it is common to start with more than p candidate predictors. Through exploratory data analysis or using domain knowledge, some of the predictors may be excluded from the analysis. In other cases, a feature selection algorithm may be used to make a data-driven choice for the minimum predictor set for the model.
- There are times when the value of an important predictor is missing. Rather than eliminating this sample from the data set, the missing value could be *imputed* using other values in the data. For example, if x_1 were missing but was correlated with predictors x_2 and x_3, an imputation method could estimate the missing x_1 observation from the values of x_2 and x_3.
- It may be beneficial to transform the scale of a predictor. If there is not *a priori* information on what the new scale should be, we can estimate the proper scale using a statistical transformation technique, the existing data, and some optimization criterion. Other transformations, such as PCA, take groups of predictors and transform them into new features that are used as the predictors.

While the examples above are related to steps that occur before the model fit, there may also be operations that occur *after* the model is created. When a classification model is created where the outcome is binary (e.g., `event` and `non-event`), it is customary to use a 50% probability cutoff to create a discrete class prediction, also known as a "hard prediction". For example, a classification model might estimate that the probability of an event was 62%. Using the typical default, the hard prediction would be `event`. However, the model may need to be more focused on reducing false positive results (i.e., where true non-events are classified as events). One way to do this is to raise the cutoff from 50% to some greater value. This increases the level of evidence required to call a new sample an event. While this reduces the true positive rate (which is bad), it may have a more dramatic effect on reducing false positives. The choice of the cutoff value should be optimized using data. This is an example of a *post-processing* step that has a significant effect on how well the model works, even though it is not contained in the model fitting step.

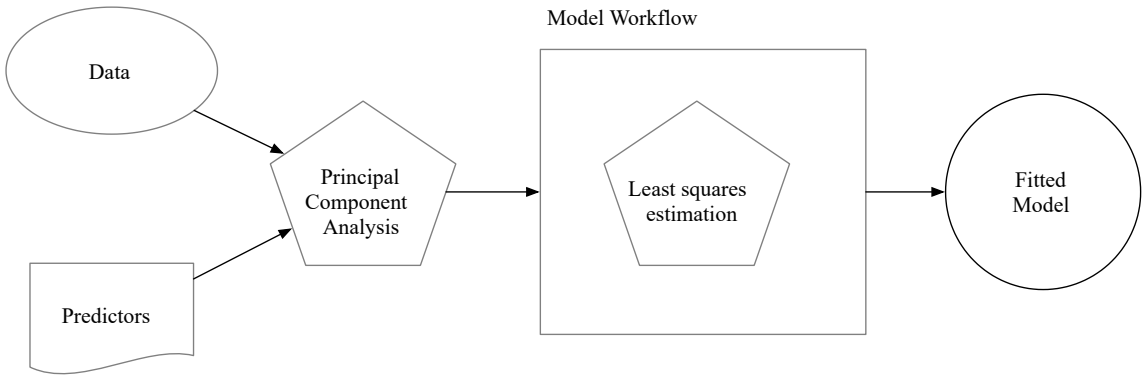
It is important to focus on the broader *modeling process*, instead of only fitting the specific model used to estimate parameters. This broader process includes any preprocessing steps, the model fit itself, as well as potential post-processing activities. In this book, we will refer to this broader process as the **model workflow** and include in it any data-driven activities that are used to produce a final model equation.

> In other software, such as Python or Spark, similar collections of steps are called *pipelines*. In tidymodels, the term "pipeline" already connotes a sequence of operations chained together with a pipe operator (such as `%>%`). Rather than using ambiguous terminology in this context, we call the sequence of computational operations related to modeling **workflows**.

Binding together the analytical components of a data analysis is important for another reason. Future chapters will demonstrate how to accurately measure performance, as well as how to optimize structural parameters (i.e. model tuning). To correctly quantify model performance on the training set, Chapter 10 advocates using *resampling* methods. To do this properly, no data-driven parts of the analysis should be excluded from validation. To this end, the workflow must include all significant estimation steps.

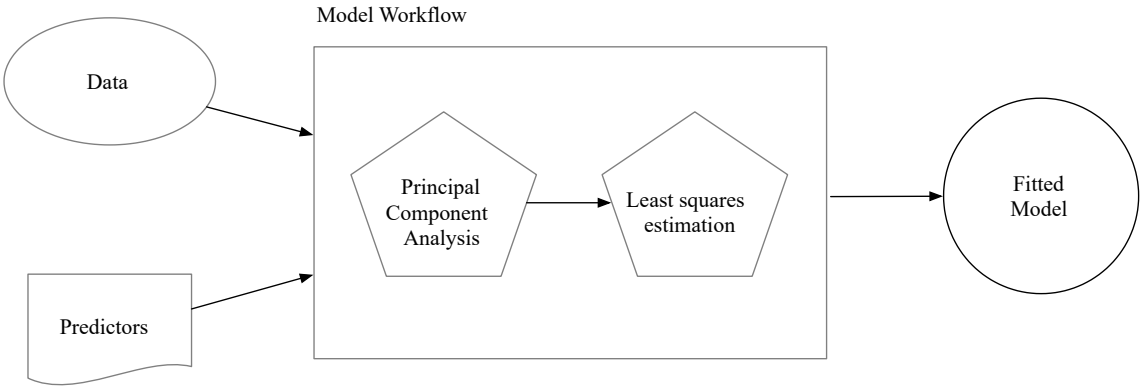
To illustrate, consider principal component analysis (PCA) signal extraction. We'll talk about this more in Section 8.4 as well as Chapter 17; PCA is a way to replace correlated predictors with new artificial features that are uncorrelated and capture most of the information in the original set. The new features could be used as the predictors and least squares regression could be used to estimate the model parameters.

There are two ways of thinking about the model workflow. The *incorrect* method would be to think of the PCA preprocessing step as *not being part of the modeling process*:



The fallacy here is that, although PCA does significant computations to produce the components, its operations are assumed to have no uncertainty associated with them. The PCA components are treated as *known* and, if not included in the model workflow, the effect of PCA could not be adequately measured.

An *appropriate* approach would be:



In this way, the PCA preprocessing is considered part of the modeling process.

7.2 WORKFLOW BASICS

The **workflows** package allows the user to bind modeling and preprocessing objects together. Let's start again with the Ames data and a simple linear model:

```
library(tidymodels)  # Includes the workflows package
tidymodels_prefer()

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

A workflow always requires a **parsnip** model object:

```
lm_wflow <-
  workflow() %>%
  add_model(lm_model)

lm_wflow
#> == Workflow ============================================================================
#> Preprocessor: None
#> Model: linear_reg()
#>
#> -- Model ----------------------------------------------------------------------------
#> Linear Regression Model Specification (regression)
#>
#> Computational engine: lm
```

Notice that we have not yet specified how this workflow should preprocess the data: `Preprocessor: None`.

If our model were very simple, a standard R formula can be used as a preprocessor:

```
lm_wflow <-
  lm_wflow %>%
  add_formula(Sale_Price ~ Longitude + Latitude)

lm_wflow
#> == Workflow ============================================================================
#> Preprocessor: Formula
#> Model: linear_reg()
#>
#> -- Preprocessor ----------------------------------------------------------------------
#> Sale_Price ~ Longitude + Latitude
#>
#> -- Model ----------------------------------------------------------------------------
#> Linear Regression Model Specification (regression)
#>
#> Computational engine: lm
```

Workflows have a `fit()` method that can be used to create the model. Using the objects created in Section 6.6:

```
lm_fit <- fit(lm_wflow, ames_train)
lm_fit
#> == Workflow [trained] ==========================================================
#> Preprocessor: Formula
#> Model: linear_reg()
#>
#> -- Preprocessor ----------------------------------------------------------------
#> Sale_Price ~ Longitude + Latitude
#>
#> -- Model ----------------------------------------------------------------------
#>
#> Call:
#> stats::lm(formula = ..y ~ ., data = data)
#>
#> Coefficients:
#> (Intercept)    Longitude     Latitude
#>     -300.25        -2.01         2.78
```

We can also `predict()` on the fitted workflow:

```
predict(lm_fit, ames_test %>% slice(1:3))
#> # A tibble: 3 × 1
#>   .pred
#>   <dbl>
#> 1  5.22
#> 2  5.22
#> 3  5.28
```

The `predict()` method follows all of the same rules and naming conventions that we described for the **parsnip** package in Section 6.3.

Both the model and preprocessor can be removed or updated:

```
lm_fit %>% update_formula(Sale_Price ~ Longitude)
#> == Workflow ============================================================================
#> Preprocessor: Formula
#> Model: linear_reg()
#>
#> -- Preprocessor ----------------------------------------------------------------------
#> Sale_Price ~ Longitude
#>
#> -- Model ----------------------------------------------------------------------------
#> Linear Regression Model Specification (regression)
#>
#> Computational engine: lm
```

Note that, in this new object, the output shows that the previous *fitted* model was removed since the new formula is inconsistent with the previous model fit.

7.3 ADDING RAW VARIABLES TO THE WORKFLOW

There is another interface for passing data to the model, the `add_variables()` function which uses a **dplyr**-like syntax for choosing variables. The function has two primary arguments: `outcomes` and `predictors`. These use a selection approach similar to the **tidyselect** back-end of **tidyverse** packages to capture multiple selectors using `c()`.

```
lm_wflow <-
  lm_wflow %>%
  remove_formula() %>%
  add_variables(outcome = Sale_Price, predictors = c(Longitude, Latitude))
lm_wflow
#> == Workflow ============================================================================
#> Preprocessor: Variables
#> Model: linear_reg()
#>
#> -- Preprocessor ----------------------------------------------------------------------
#> Outcomes: Sale_Price
#> Predictors: c(Longitude, Latitude)
#>
#> -- Model ----------------------------------------------------------------------------
#> Linear Regression Model Specification (regression)
#>
#> Computational engine: lm
```

The predictors could also have been specified using a more general selector, such as

```
predictors = c(ends_with("tude"))
```

One nicety is that any outcome columns accidentally specified in the predictors argument will be quietly removed. This facilitates the use of:

```
predictors = everything()
```

When the model is fit, the specification assembles these data, unaltered, into a data frame and passes it to the underlying function:

```
fit(lm_wflow, ames_train)
#> == Workflow [trained] ==========================================================
#> Preprocessor: Variables
#> Model: linear_reg()
#>
#> -- Preprocessor ----------------------------------------------------------------
#> Outcomes: Sale_Price
#> Predictors: c(Longitude, Latitude)
#>
#> -- Model -----------------------------------------------------------------------
#>
#> Call:
#> stats::lm(formula = ..y ~ ., data = data)
#>
#> Coefficients:
#> (Intercept)    Longitude     Latitude
#>     -300.25        -2.01         2.78
```

If you would like the underlying modeling method to do what it would normally do with the data, `add_variables()` can be a helpful interface. As we will discuss in Section 7.4.1, it also facilitates more complex modeling specifications. However, as we mention in the next section, models such as `glmnet` and `xgboost` expect the user to make indicator variables from factor predictors. In these cases, a recipe or formula interface will typically be a better choice.

As seen in the next chapter, a more powerful preprocessor (called a *recipe*) can also be added to a workflow.

7.4 HOW DOES A WORKFLOW USE THE FORMULA?

Recall from Section 3.2 that the formula method in R has multiple purposes (we will discuss this further in Chapter 8). One of these is to properly encode the original data into an analysis ready format. This can involve executing in-line transformations (e.g., `log(x)`), creating dummy variable

columns, creating interactions or other column expansions, and so on. However, there are many statistical methods that require different types of encodings:

- Most packages for tree-based models use the formula interface but **do not** encode the categorical predictors as dummy variables.
- Packages can use special in-line functions that tell the model function how to treat the predictor in the analysis. For example, in survival analysis models, a formula term such as `strata(site)` would indicate that the column `site` is a stratification variable. This means that it should not be treated as a regular predictor and does not have a corresponding location parameter estimate in the model.
- A few R packages have extended the formula in ways that base R functions cannot parse or execute. In multilevel models (e.g. mixed models or hierarchical Bayesian models), a model term such as `(week | subject)` indicates that the column `week` is a random effect that has different slope parameter estimates for each value of the `subject` column.

A workflow is a general purpose interface. When `add_formula()` is used, how should the workflow pre-process the data? Since the preprocessing is model dependent, workflows attempts to emulate what the underlying model would do *whenever possible*. If it is not possible, the formula processing should not do anything to the columns used in the formula. Let's look at this in more detail.

TREE-BASED MODELS

When we fit a tree to the data, the **parsnip** package understands what the modeling function would do. For example, if a random forest model is fit using the **ranger** or **randomForest** packages, the workflow knows predictors columns that are factors should be left as-is.

As a counter example, a boosted tree created with the **xgboost** package requires the user to create dummy variables from factor predictors (since `xgboost::xgb.train()` will not). This requirement is embedded into the model specification object and a workflow using **xgboost** *will* create the indicator columns for this engine. Also note that a different engine for boosted trees, C5.0, does not require dummy variables so none are made by the workflow.

This determination is made for *each model and engine combination*.

7.4.1 SPECIAL FORMULAS AND IN-LINE FUNCTIONS

A number of multilevel models have standardized on a formula specification devised in the **lme4** package. For example, to fit a regression model that has random effects for subjects, we would use the following formula:

```
library(lme4)
lmer(distance ~ Sex + (age | Subject), data = Orthodont)
```

The effect of this is that each subject will have an estimated intercept and slope parameter for `age`.

The problem is that standard R methods can't properly process this formula:

```
model.matrix(distance ~ Sex + (age | Subject), data = Orthodont)
#> Warning in Ops.ordered(age, Subject): '|' is not meaningful for ordered factors
#>      (Intercept) SexFemale age | SubjectTRUE
#> attr(,"assign")
#> [1] 0 1 2
#> attr(,"contrasts")
#> attr(,"contrasts")$Sex
#> [1] "contr.treatment"
#>
#> attr(,"contrasts")$`age | Subject`
#> [1] "contr.treatment"
```

The result is a zero row data frame.

> The issue is that the special formula has to be processed by the underlying package code, not the standard `model.matrix()` approach.

Even if this formula could be used with `model.matrix()`, this would still present a problem since the formula also specifies the statistical attributes of the model.

The solution in `workflows` is an optional supplementary model formula that can be passed to `add_model()`. For example, using the previously mentioned `strata()` function in the **survival** package, the `add_variables()` specification provides the bare column names and then the actual formula given to the model is set within `add_model()`:

```
library(survival)

parametric_model <-
  surv_reg() %>%
  set_engine("survival")
#> Warning: `surv_reg()` was deprecated in parsnip 0.1.6.
#> Please use `survival_reg()` instead.

parametric_workflow <-
  workflow() %>%
  # Pass the data along as-is:
  add_variables(outcome = c(fustat, futime), predictors = c(age, rx)) %>%
  add_model(parametric_model,
            # This formula is given to the model
            formula = Surv(futime, fustat) ~ age + strata(rx))

parametric_fit <- fit(parametric_workflow, data = ovarian)
parametric_fit
#> == Workflow [trained] ==========================================================
#> Preprocessor: Variables
#> Model: surv_reg()
#>
#> -- Preprocessor ----------------------------------------------------------------
#> Outcomes: c(fustat, futime)
#> Predictors: c(age, rx)
#>
#> -- Model ------------------------------------------------------------------------
#> Call:
#> survival::survreg(formula = Surv(futime, fustat) ~ age + strata(rx),
#>     data = data, model = TRUE)
#>
#> Coefficients:
#> (Intercept)         age
#>     12.8734     -0.1034
#>
```

```
#> Scale:
#>   rx=1   rx=2
#> 0.7696 0.4704
#>
#> Loglik(model)= -89.4   Loglik(intercept only)= -97.1
#>  Chisq= 15.36 on 1 degrees of freedom, p= 9e-05
#> n= 26
```

Notice how, in the call printed above, the model-specific formula was used.

7.5 CREATING MULTIPLE WORKFLOWS AT ONCE

There are some situations where the data require numerous attempts to find an appropriate model. For example:

- For predictive models, it is advisable to evaluate a variety of different model types. This requires the user to create multiple model specifications.
- Sequential testing of models typically starts with an expanded set of predictors. This "full model" is compared to a sequence of the same model that removes each predictor in turn. Using basic hypothesis testing methods or empirical validation, the effect of each predictor can be isolated and assessed.

In these situations, as well as others, it can become tedious or onerous to create a lot of workflows from different sets of preprocessors and/or model specifications. To address this problem, the **workflowset** package creates combinations of workflow components. A list of preprocessors (e.g., formulas, **dplyr** selectors, or feature engineering recipe objects discussed in the next chapter) can be combined with a list of model specifications, resulting in a set of workflows.

As an example, let's say that we want to focus on the different ways that house location is represented in the Ames data. We can create a set of formulas that capture these predictors:

```
location <- list(
  longitude = Sale_Price ~ Longitude,
  latitude = Sale_Price ~ Latitude,
  coords = Sale_Price ~ Longitude + Latitude,
  neighborhood = Sale_Price ~ Neighborhood
)
```

These representations can be crossed with one or more models using the `workflow_set()` function. We'll just use the previous linear model specification to demonstrate:

```
library(workflowsets)
location_models <- workflow_set(preproc = location, models = list(lm = lm_model))
location_models
#> # A workflow set/tibble: 4 × 4
#>   wflow_id        info             option    result
#>   <chr>           <list>           <list>    <list>
#> 1 longitude_lm    <tibble [1 × 4]> <opts[0]> <list [0]>
#> 2 latitude_lm     <tibble [1 × 4]> <opts[0]> <list [0]>
#> 3 coords_lm       <tibble [1 × 4]> <opts[0]> <list [0]>
#> 4 neighborhood_lm <tibble [1 × 4]> <opts[0]> <list [0]>
location_models$info[[1]]
#> # A tibble: 1 × 4
#>   workflow   preproc model      comment
#>   <list>     <chr>   <chr>      <chr>
#> 1 <workflow> formula linear_reg ""
extract_workflow(location_models, id = "coords_lm")
#> == Workflow ====================================================================
#> Preprocessor: Formula
#> Model: linear_reg()
#>
#> -- Preprocessor ----------------------------------------------------------------
#> Sale_Price ~ Longitude + Latitude
#>
#> -- Model -----------------------------------------------------------------------
#> Linear Regression Model Specification (regression)
#>
#> Computational engine: lm
```

Workflow sets are mostly designed to work with resampling, which is discussed in Chapter 10. In the object above, the columns `option` and `result` must be populated with specific types of objects that result from resampling. We will demonstrate this in more detail in Chapters 11 and 15.

In the meantime, let's create model fits for each formula and save them in a new column called `fit`. We'll use basic **dplyr** and **purrr** operations:

```
location_models <-
   location_models %>%
   mutate(fit = map(info, ~ fit(.x$workflow[[1]], ames_train)))
location_models
#> # A workflow set/tibble: 4 × 5
#>   wflow_id        info             option    result     fit
#>   <chr>           <list>           <list>    <list>     <list>
#> 1 longitude_lm    <tibble [1 × 4]> <opts[0]> <list [0]> <workflow>
#> 2 latitude_lm     <tibble [1 × 4]> <opts[0]> <list [0]> <workflow>
#> 3 coords_lm       <tibble [1 × 4]> <opts[0]> <list [0]> <workflow>
#> 4 neighborhood_lm <tibble [1 × 4]> <opts[0]> <list [0]> <workflow>
location_models$fit[[1]]
#> == Workflow [trained] ==========================================================
#> Preprocessor: Formula
#> Model: linear_reg()
#>
#> -- Preprocessor ----------------------------------------------------------------
#> Sale_Price ~ Longitude
#>
#> -- Model -----------------------------------------------------------------------
#>
#> Call:
#> stats::lm(formula = ..y ~ ., data = data)
#>
#> Coefficients:
#> (Intercept)    Longitude
#>     -176.46        -1.94
```

There's a lot more to workflow sets. Their nuances and advantages won't be illustrated until Chapter 15.

7.6 FUTURE PLANS

The two types of components in a workflow are preprocessors and models. There are also operations that might occur *after* the model is fit. An example of such a *post-processor* would be cutoff selection for two-class problems. Previously in this chapter, we discussed the idea of modifying the cutoff for a two-class problem. In the future, workflows will be able to attach a custom cutoff that is applied to probabilities after the model fit. Other approaches, such as probability calibration, could also be added as post-processors.

7.7 CHAPTER SUMMARY

In this chapter, you learned that the modeling process encompasses more than just estimating the parameters of an algorithm that connects predictors to an outcome. This process also includes preprocessing steps and operations taken after a model is fit. We introduced a concept called a **model workflow** that can capture the important components of the modeling process. Multiple workflows can also be created inside of a **workflow set**.

For the Ames data, the code used in later chapters 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")

lm_wflow <-
  workflow() %>%
  add_model(lm_model) %>%
  add_formula(Sale_Price ~ Longitude + Latitude)

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