Introduction to tidymodels

SoCal RUG Hackathon 2022-04

Emil Hvitfeldt

2022-04-02

About Me

- Software Engineer at RStudio PBC

- R package developer, about 10 packages on CRAN (textrecipes, themis, paletteer, prismatic, textdata)

- Co-author of "Supervised Machine Learning for Text Analysis in R" with Julia Silge

- Located in sunny California

- Has 3 cats; Presto, Oreo, and Wiggles

Our goals for this tutorial

- Introduce tidymodels and its general philosophy on modeling.

- Help you become proficient with the core packages for modeling.

- Point you to places to learn more and get help.

Why tidymodels?

There are several other modeling frameworks in R that try to:

- create a uniform, cohesive, and unsurprising set of modeling APIs

Examples are caret, mlr3, and others.

- caret is more favorable for people who prefer base R/traditional interfaces.
- mlr3 is more pythonic and also has many features.
- tidymodels would probably be preferable to those who place importance on a tidy R interface, a large number of features, and the idea that the interfaces should enable the "pit of success".

The tidymodels package

There are a lot of tidymodels packages but about 90% of the work is done by 5 packages. (**rsample**, **recipes**, **parsnip**, **tune**, and **yardstick**)

The best way to get started with tidymodels is to use the **tidymodels** meta-package. It loads the core packages plus some tidyverse packages.

Some helpful links:

- List of all tidymodels functions
- List of all parsnip models
- List of all recipe steps

The tidymodels package

```
library(tidymodels)
```

```
── Attaching packages ──────────────────────────────────────── tidymodels 0.1.4 ──
✓ broom        0.7.9      ✓ recipes      0.1.17
✓ dials        0.0.10     ✓ rsample      0.1.0
✓ dplyr        1.0.7      ✓ tibble       3.1.5
✓ ggplot2      3.3.5      ✓ tidyr        1.1.4
✓ infer        1.0.0      ✓ tune         0.1.6
✓ modeldata    0.1.1      ✓ workflows    0.2.4
✓ parsnip      0.1.7      ✓ workflowsets 0.1.0
✓ purrr        0.3.4      ✓ yardstick    0.0.8
── 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()
• Use tidymodels_prefer() to resolve common conflicts.
```

Managing name conflicts

```
tidymodels_prefer(quiet = FALSE)
```

```
## [conflicted] Will prefer dplyr::filter over any other package
## [conflicted] Will prefer dplyr::select over any other package
## [conflicted] Will prefer dplyr::slice over any other package
## [conflicted] Will prefer dplyr::rename over any other package
## [conflicted] Will prefer dials::neighbors over any other package
## [conflicted] Will prefer plsmod::pls over any other package
## [conflicted] Will prefer purrr::map over any other package
## [conflicted] Will prefer recipes::step over any other package
## [conflicted] Will prefer themis::step_downsample over any other package
## [conflicted] Will prefer themis::step_upsample over any other package
## [conflicted] Will prefer tune::tune over any other package
## [conflicted] Will prefer yardstick::precision over any other package
## [conflicted] Will prefer yardstick::recall over any other package
```

Base R and tidyverse differences

Base R/caret

```
mtcars <- mtcars[order(mtcars$cyl),]
mtcars <- mtcars[, "mpg", drop = FALSE]


# ------------------------------------------------------

mtcars$mp         # matches incomplete arg
mtcars[, "mpg"]   # a vector


# ------------------------------------------------------

num_args <- function(x) length(formals(x))


num_args(caret::trainControl) +
  num_args(caret:::train.default)
```

```
38
```

tidyverse/tidymodels

```
mtcars %>%
  arrange(cyl) %>%
  select(mpg)

# ------------------------------------------------------

tb_cars <- as_tibble(mtcars)
tb_cars$mp         # fails
tb_cars[, "mpg"]   # A tibble

# ------------------------------------------------------



num_args(linear_reg) + num_args(set_engine) +
  num_args(tune_grid) + num_args(control_grid) +
  num_args(vfold_cv)
```

```
23
```

Example data set

These data are used in our Feature Engineering and Selection book.

Several years worth of pre-pandemic data were assembled to try to predict the daily number of people entering the Clark and Lake elevated ("L") train station in Chicago.

For predictors,

- the 14-day lagged ridership at this and other stations (units: thousands of rides/day)
- weather data
- home/away game schedules for Chicago teams
- the date

The data are in `modeldata`. See `?Chicago`.

Define the score of "The Model"

In tidymodels, there is the idea that a model-oriented data analysis consists of

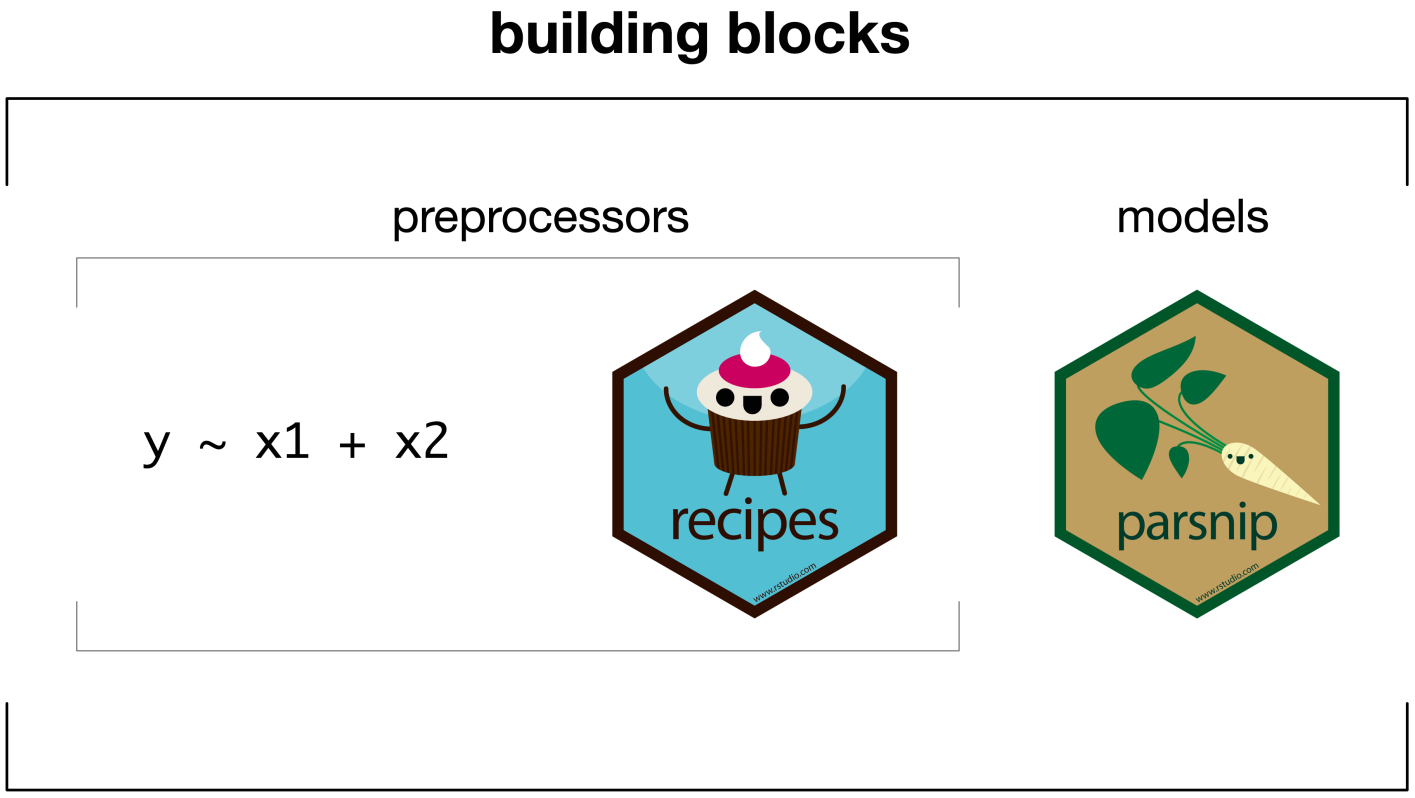
- a preprocessor, and
- a model

The preprocessor might be a simple formula or a sophisticated recipe.

It's important to consider both of these activities as part of the data analysis process.

- Post-model activities should also be included there (e.g. calibration, cut-off optimization, etc.)
- We don't have those implemented yet.

Basic tidymodels components



A relevant example

Let's say that we have some highly correlated predictors and we want to reduce the correlation by first applying principal component analysis to the data.

- AKA principal component regression

A relevant example

Let's say that we have some highly correlated predictors and we want to reduce the correlation by first applying principal component analysis to the data.

- AKA principal component regression feature extraction

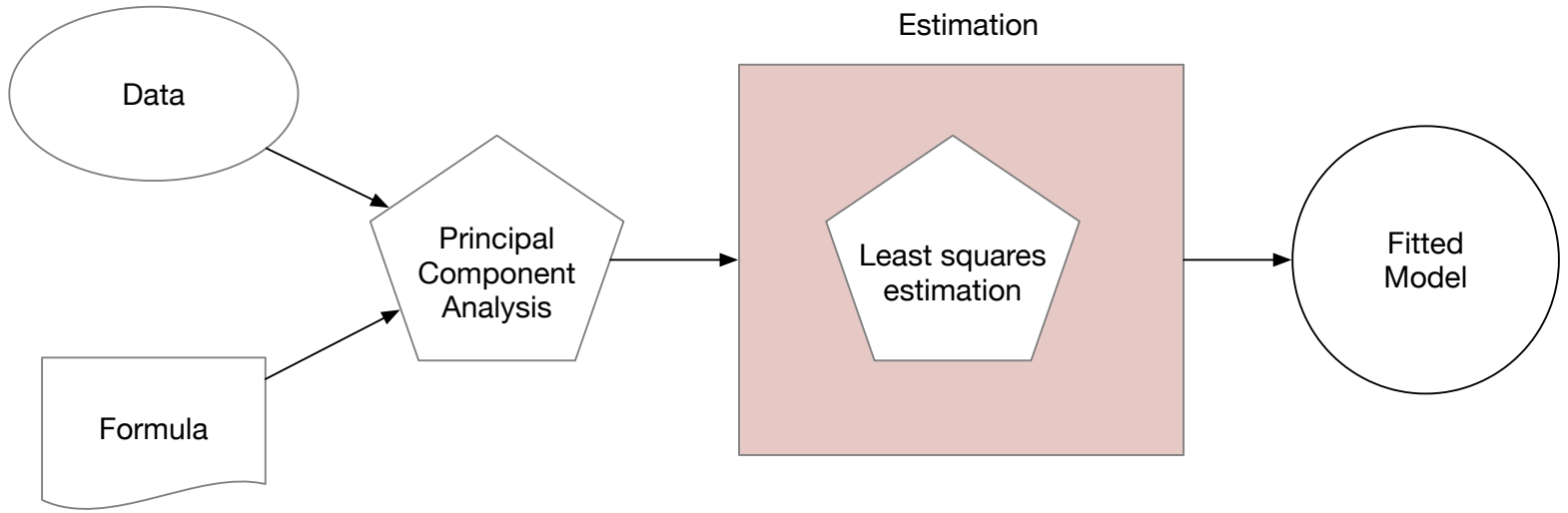
A relevant example

Let's say that we have some highly correlated predictors and we want to reduce the correlation by first applying principal component analysis to the data.

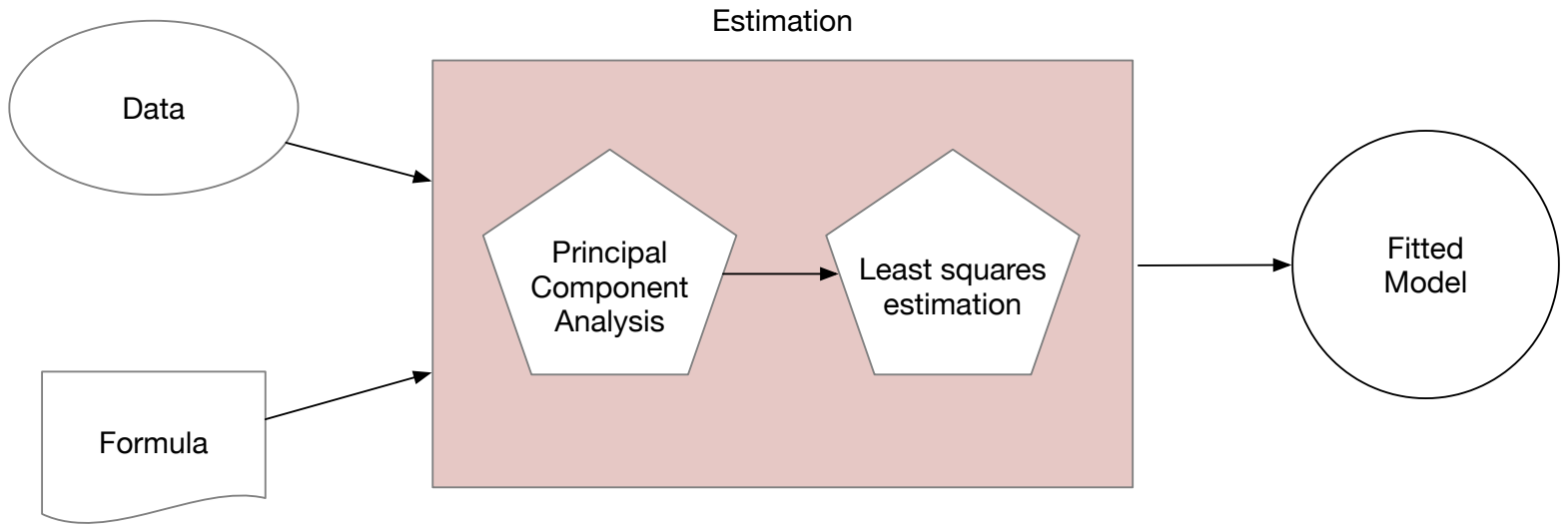
- AKA principal component regression feature extraction

What do we consider the estimation part of this process?

Is it this?


Data
Formula
Principal Component Analysis
Estimation
Least squares estimation
Fitted Model

Or is it this?


Estimation
Data
Formula
Principal Component Analysis
Least squares estimation
Fitted Model

What's the difference?

It is easy to think that the model fit is the only estimation steps.

There are cases where this could go really wrong:

- Poor estimation of performance (buy treating the PCA parts as known)

- Selection bias in feature selection

- Information/data leakage

These problems are exacerbated as the preprocessors increase in complexity and/or effectiveness.

We'll come back to this at the end of this section

Data splitting

Always have a separate piece of data that can contradict what you believe

Data splitting and spending

How do we "spend" the data to find an optimal model?

We **typically** split data into training and test data sets:

- **Training Set**: these data are used to estimate model parameters and to pick the values of the complexity parameter(s) for the model.

- **Test Set**: these data can be used to get an independent assessment of model efficacy. **They should not be used during model training** (like, at all).

Data splitting and spending

The more data we spend, the better estimates we'll get (provided the data is accurate).

Given a fixed amount of data:

- Too much spent in training won't allow us to get a good assessment of predictive performance. We may find a model that fits the training data very well, but is not generalizable (overfitting)

- Too much spent in testing won't allow us to get a good assessment of model parameters

Statistically, the best course of action would be to use all the data for model building and use statistical methods to get good estimates of error.

From a non-statistical perspective, many consumers of complex models emphasize the need for an untouched set of samples to evaluate performance.

Mechanics of data splitting

There are a few different ways to do the split: simple random sampling, **stratified sampling based on the outcome**, by date, or methods that focus on the distribution of the predictors.

For stratification:

- **classification**: this would mean sampling within the classes to preserve the distribution of the outcome in the training and test sets

- **regression**: determine the quartiles of the data set and sample within those artificial groups

For **time series**, we often use the most recent data as the test set.

Splitting with Chicago data

`initial_split()` can be used when we use randomness to make the split.

Let's put the last two weeks of data into the test set. `initial_time_split()` can be used for this purpose:

```
set.seed(1234)
chi_split <- initial_time_split(Chicago, prop = 1 - (14/nrow(Chicago)))
chi_split
```

```
## <Analysis/Assess/Total>
## <5684/14/5698>
```

```
chi_train <- training(chi_split)
chi_test  <- testing(chi_split)

c(training = nrow(chi_train), testing = nrow(chi_test))
```

```
## training  testing
##     5684       14
```

Creating models in R

Specifying models in R using formulas

To fit a model to the housing data, the model terms must be specified. Historically, there are two main interfaces for doing this.

The **formula** interface using R formula rules to specify a **symbolic** representation of the terms:

Variables + interactions

```
# day_of_week is not in the data set but day_of_week = lubridate::wday(date, label = TRUE)
model_fn(ridership ~ day_of_week + Western + day_of_week:Western, data = chi_train)
```

Shorthand for all predictors

```
model_fn(ridership ~ ., data = chi_train)
```

Inline functions / transformations

```
model_fn(log10(ridership) ~ ns(Western, df = 3) + ., data = chi_train)
```

Downsides to formulas

- You can't nest in-line functions such as `model_fn(y ~ pca(scale(x1), scale(x2), scale(x3)), data = dat)`.

- All the model matrix calculations happen at once and can't be recycled when used in a model function.

- For very **wide** data sets, the formula method can be extremely inefficient.

- There are limited **roles** that variables can take which has led to several re-implementations of formulas.

- Specifying multivariate outcomes is clunky and inelegant.

- Not all modeling functions have a formula method (consistency!).

Specifying models without formulas

Some modeling function have a non-formula (XY) interface. This usually has arguments for the predictors and the outcome(s):

```
# Usually, the variables must all be numeric
pre_vars <- c("Austin", "Clark_Lake", "California")
model_fn(x = chi_train[, pre_vars],
         y = chi_train$ridership)
```

This is inconvenient if you have transformations, factor variables, interactions, or any other operations to apply to the data prior to modeling.

Overall, it is difficult to predict if a package has one or both of these interfaces. For example, `lm` only has formulas.

There is a **third interface**, using **recipes** that will be discussed later that solves some of these issues.

A linear regression model

Let's start by fitting an ordinary linear regression model to the training set. You can choose the model terms for your model, but I will use a very simple model:

```
simple_lm <- lm(ridership ~ Clark_Lake + humidity, data = chi_train)
```

Before looking at coefficients, we should do some model checking to see if there is anything obviously wrong with the model.

To get the statistics on the individual data points, we will use the awesome `broom` package:

```
simple_lm_values <- augment(simple_lm)
names(simple_lm_values)
```

```
## [1] "ridership"  "Clark_Lake" "humidity"   ".fitted"    ".resid"
## [6] ".hat"       ".sigma"     ".cooksd"    ".std.resid"
```

Fitting via tidymodels

The parsnip package

- A tidy unified **interface** to models

- `lm()` isn't the only way to perform linear regression

    - **glmnet** for regularized regression

    - **stan** for Bayesian regression

    - **keras** for regression using tensorflow

- But...remember the consistency slide?

    - Each interface has its own minutiae to remember

    - **parsnip** standardizes all that!

parsnip in action

1) Create specification

2) Set the engine

3) Fit the model

```
spec_lin_reg <- linear_reg()
spec_lin_reg
```

```
## Linear Regression Model Specification (regression)
##
## Computational engine: lm
```

```
spec_lm <- spec_lin_reg %>% set_engine("lm")
spec_lm
```

```
## Linear Regression Model Specification (regression)
##
## Computational engine: lm
```

```
fit_lm <- fit(
  spec_lm,
  ridership ~ Clark_Lake + humidity,
  data = chi_train
)

fit_lm
```

```
## parsnip model object
##
##
## Call:
## stats::lm(formula = ridership ~ Clark_Lake + humidi
##
## Coefficients:
## (Intercept)   Clark_Lake     humidity
##    1.764496     0.883693    -0.002497
```

Note: Models have default engines. We don't really need to use `set_engine("lm")` for this example.

Alternative engines

With **parsnip**, it is easy to switch to a different engine, like Stan, to run the same model with alternative backends.

```
spec_stan <-
  spec_lin_reg %>%
  # Engine specific arguments are
  # passed through here
  set_engine("stan", chains = 4, iter = 1000)

# Otherwise, looks exactly the same!
fit_stan <- fit(
  spec_stan,
  ridership ~ Clark_Lake + humidity,
  data = chi_train
)
```

```
coef(fit_stan$fit)
```

```
##  (Intercept)   Clark_Lake     humidity
##  1.768420874  0.883585654 -0.002524197
```

```
coef(fit_lm$fit)
```

```
##  (Intercept)   Clark_Lake     humidity
##  1.764495522  0.883692888 -0.002497053
```

Duplicate computations

Note that, for both of these fits, some of the computations are repeated.

For example, the formula method does a fair amount of work to figure out how to turn the data frame into a matrix of predictors.

When there are special effects (e.g. splines), dummy variables, interactions, or other components, the formula/terms objects have to keep track of everything.

In cases where there are a lot of **predictors**, these computations can consume a lot of resources. If we can save them, that would be helpful.

The answer is a **workflow** object. These bundle together a preprocessor (such as a formula) along with a model.

A modeling workflow

We can optionally bundle the recipe and model together into a ~~pipeline~~ workflow:

```
reg_wflow <-
  workflow() %>%    # attached with the tidymodels package
  add_model(spec_lm) %>%
  add_formula(ridership ~ Clark_Lake + humidity) # or add_recipe() or add_variables()

reg_fit <- fit(reg_wflow, data = chi_train)
reg_fit
```

```
## == Workflow [trained] ==========================================================
## Preprocessor: Formula
## Model: linear_reg()
##
## -- Preprocessor ----------------------------------------------------------------
## ridership ~ Clark_Lake + humidity
##
## -- Model -----------------------------------------------------------------------
##
## Call:
## stats::lm(formula = ..y ~ ., data = data)
##
## Coefficients:
## (Intercept)   Clark_Lake     humidity
##    1.764496     0.883693    -0.002497
```

Swapping models

```
stan_wflow <-
  reg_wflow %>%
  update_model(spec_stan)

set.seed(21)
stan_fit <- fit(stan_wflow, data = chi_train)
stan_fit
```

```
## == Workflow [trained] ==========================================================
## Preprocessor: Formula
## Model: linear_reg()
##
## -- Preprocessor ----------------------------------------------------------------
## ridership ~ Clark_Lake + humidity
##
## -- Model -----------------------------------------------------------------------
## stan_glm
##  family:       gaussian [identity]
##  formula:      ..y ~ .
##  observations: 5684
##  predictors:   3
## ------
##             Median MAD_SD
## (Intercept) 1.8    0.2
## Clark_Lake  0.9    0.0
## humidity    0.0    0.0
```

Workflows

Once the first model is fit, the preprocessor (i.e. the formula) is processed and the model matrix is formed.

New models don't need to repeat those computations.

Some other nice features:

- Workflows are smarter with data than `model.matrix()` in terms of new factor levels.

- Other preprocessors can be used: recipes and `dplyr::select()` statements (that do no data processing).

- As will be seen later, they can help organize your work when a sequence of models are used.

- A workflow captures the entire modeling process (mentioned earlier) and a simple `fit()` and `predict()` sequence are used for all of the estimation parts.

Using workflows to predict

```
# generate some bogus data (instead of using the training or test sets)
set.seed(3)
shuffled_data <- map_dfc(Chicago, ~ sample(.x, size = 10))

predict(stan_fit, shuffled_data) %>% slice(1:3)
```

```
## # A tibble: 3 × 1
##   .pred
##   <dbl>
## 1  15.9
## 2  18.5
## 3  17.4
```

```
predict(stan_fit, shuffled_data, type = "pred_int") %>% slice(1:3)
```

```
## # A tibble: 3 × 2
##   .pred_lower .pred_upper
##         <dbl>       <dbl>
## 1        10.2        22.0
## 2        12.6        24.6
## 3        11.3        23.4
```

The tidymodels prediction guarantee!

- The predictions will always be inside a **tibble**.
- The column names and types are **unsurprising**.
- The number of rows in `new_data` and the output **are the same**.

This enables the use of `bind_cols()` to combine the original data and the predictions.

Evaluating models

tidymodels has a lot of performance metrics for different types of models (e.g. binary classification, etc).

Each takes a tibble as an input along with the observed and predicted column names:

```
pred_results <-
  predict(stan_fit, shuffled_data) %>%
  bind_cols(shuffled_data)

# Data was randomized; these results should be bad
pred_results %>% rmse(truth = ridership, estimate = .pred)
```

```
## # A tibble: 1 × 3
##   .metric .estimator .estimate
##   <chr>   <chr>          <dbl>
## 1 rmse    standard        6.17
```

Multiple metrics/KPIs

A **metric set** can bundle multiple statistics:

```
reg_metrics <- metric_set(rmse, rsq, mae, ccc)

# A tidy format of the results
pred_results %>% reg_metrics(truth = ridership, estimate = .pred)
```

```
## # A tibble: 4 × 3
##   .metric .estimator .estimate
##   <chr>   <chr>          <dbl>
## 1 rmse    standard       6.17
## 2 rsq     standard       0.304
## 3 mae     standard       4.36
## 4 ccc     standard       0.434
```

broom methods

parsnip and workflow fits have corresponding broom tidiers:

```
glance(reg_fit)
```

```
## # A tibble: 1 × 12
##   r.squared adj.r.squared sigma statistic p.value    df  logLik    AIC    BIC
##       <dbl>         <dbl> <dbl>     <dbl>   <dbl> <dbl>   <dbl>  <dbl>  <dbl>
## 1     0.780         0.780  3.07    10094.       0     2 -14448. 28905. 28931.
## # … with 3 more variables: deviance <dbl>, df.residual <int>, nobs <int>
```

```
tidy(reg_fit)
```

```
## # A tibble: 3 × 5
##   term        estimate std.error statistic  p.value
##   <chr>          <dbl>     <dbl>     <dbl>    <dbl>
## 1 (Intercept)  1.76      0.222       7.94  2.46e-15
## 2 Clark_Lake   0.884     0.00622   142.    0
## 3 humidity    -0.00250   0.00295    -0.847 3.97e- 1
```

broom methods

For `augment()` we require the data to predict and attach

```
augment(reg_fit, shuffled_data %>% select(Clark_Lake, humidity, ridership))
```

```
## # A tibble: 10 × 4
##    Clark_Lake humidity ridership .pred
##         <dbl>    <dbl>     <dbl> <dbl>
##  1      16.3      78        6.36 15.9
##  2      19.2      70       19.3  18.5
##  3      17.8      45       15.7  17.4
##  4      19.9      50        4.31 19.3
##  5      18.7      61       18.7  18.1
##  6       2.13     54.5      8.44  3.51
##  7       4.56     43        1.74  5.69
##  8      17.1      77.5     14.7  16.6
##  9      19.4      59       14.6  18.7
## 10      16.7      39       15.3  16.5
```

What is feature engineering?

First thing's first: what's a feature?

I tend to think of a feature as some representation of a predictor that will be used in a model.

Old-school features:

- Interactions
- Polynomial expansions/splines
- PCA feature extraction

"Feature engineering" sounds pretty cool, but let's take a minute to talk about **preprocessing** data.

Two types of preprocessing

Two types of preprocessing

Easy examples

For example, centering and scaling are definitely not feature engineering.

Consider the `date` field in the Chicago data. If given as a raw predictor, it is converted to an integer.

Spoiler alert: the date is the most important factor. It can be re-encoded as:

- Days since a reference date 😪
- Day of the week ❤️❤️❤️❤️
- Month 😪
- Year ❤️❤️
- Indicators for holidays ❤️❤️❤️
- Indicators for home games for NFL, NBA, etc. 😪

General definitions

- Data preprocessing are the steps that you take to make your model successful.

- Feature engineering are what you do to the original predictors to make the model do the least work to predict the outcome as well as possible.

We'll demonstrate the recipes package for all of your data needs.

Recipes prepare your data for modeling

The package is extensible framework for pipeable sequences of feature engineering steps provides preprocessing tools to be applied to data.

Statistical parameters for the steps can be estimated from an initial data set and then applied to other data sets.

The resulting processed output can then be used as inputs for statistical or machine learning models.

A first recipe

```
chi_rec <-
  recipe(ridership ~ ., data = chi_train)

# If ncol(data) is large, you can use
# recipe(data = chi_train)
```

Based on the formula, the function assigns columns to roles of "outcome" or "predictor"

```
summary(chi_rec)
```

```
## # A tibble: 50 × 4
##    variable         type    role      source
##    <chr>            <chr>   <chr>     <chr>
##  1 Austin           numeric predictor original
##  2 Quincy_Wells     numeric predictor original
##  3 Belmont          numeric predictor original
##  4 Archer_35th      numeric predictor original
##  5 Oak_Park         numeric predictor original
##  6 Western          numeric predictor original
##  7 Clark_Lake       numeric predictor original
##  8 Clinton          numeric predictor original
##  9 Merchandise_Mart numeric predictor original
## 10 Irving_Park      numeric predictor original
## # … with 40 more rows
```

A first recipe - work with dates

```
chi_rec <-
  recipe(ridership ~ ., data = chi_train) %>%
  step_date(date, features = c("dow", "month", "year"))
```

This creates three new columns in the data based on the date. Now that the day-of-the-week column is a factor.

A first recipe - work with dates

```
chi_rec <-
  recipe(ridership ~ ., data = chi_train) %>%
  step_date(date, features = c("dow", "month", "year")) %>%
  step_holiday(date)
```

Add indicators for major holidays. Specific holidays, especially those ex-US, can also be generated.

At this point, we don't need `date` anymore. Instead of deleting it (there is a step for that) we will change its **role** to be an identification variable.

A first recipe - work with dates

```
chi_rec <-
  recipe(ridership ~ ., data = chi_train) %>%
  step_date(date, features = c("dow", "month", "year")) %>%
  step_holiday(date) %>%
  update_role(date, new_role = "id")
```

`date` is still in the data set but tidymodels knows not to treat it as an analysis column.

A first recipe -create indicator variables

```
chi_rec <-
  recipe(ridership ~ ., data = chi_train) %>%
  step_date(date, features = c("dow", "month", "year")) %>%
  step_holiday(date) %>%
  update_role(date, new_role = "id") %>%
  step_dummy(all_nominal_predictors())
```

For any factor or character predictors, make binary indicators.

There are **many** recipe steps that can convert categorical predictors to numeric columns.

A first recipe - filter out constant columns

```
chi_rec <-
  recipe(ridership ~ ., data = chi_train) %>%
  step_date(date, features = c("dow", "month", "year")) %>%
  step_holiday(date) %>%
  update_role(date, new_role = "id") %>%
  step_dummy(all_nominal_predictors()) %>%
  step_zv(all_predictors())
```

In case there is a holiday that never was observed, we can delete any zero-variance predictors that have a single unique value.

Note that the selector chooses all columns with a role of "predictor"

A first recipe - normalization

```
chi_rec <-
  recipe(ridership ~ ., data = chi_train) %>%
  step_date(date, features = c("dow", "month", "year")) %>%
  step_holiday(date) %>%
  update_role(date, new_role = "id") %>%
  step_dummy(all_nominal_predictors()) %>%
  step_zv(all_predictors()) %>%
  step_normalize(all_numeric_predictors())
```

This centers and scales the numeric predictors.

Note that this will use the training set to estimate the means and standard deviations of the data.

All data put through the recipe will be normalized using those statistics (there is no re-estimation).

A first recipe - reduce correlation

```
chi_rec <-
  recipe(ridership ~ ., data = chi_train) %>%
  step_date(date, features = c("dow", "month", "year")) %>%
  step_holiday(date) %>%
  update_role(date, new_role = "id") %>%
  step_dummy(all_nominal_predictors()) %>%
  step_zv(all_predictors()) %>%
  step_normalize(all_numeric_predictors()) %>%
  step_corr(all_numeric_predictors(), threshold = 0.9)
```

To deal with highly correlated predicors, find the minimum predictor set to remove to make the pairwise correlations are less than 0.9.

There are other filter steps too,

Other possible steps

```
chi_rec <-
  recipe(ridership ~ ., data = chi_train) %>%
  step_date(date, features = c("dow", "month", "year")) %>%
  step_holiday(date) %>%
  update_role(date, new_role = "id") %>%
  step_dummy(all_nominal_predictors()) %>%
  step_zv(all_predictors()) %>%
  step_normalize(all_numeric_predictors()) %>%
  step_pca(all_numeric_predictors())
```

PCA feature extraction...

Other possible steps

```
chi_rec <-
  recipe(ridership ~ ., data = chi_train) %>%
  step_date(date, features = c("dow", "month", "year")) %>%
  step_holiday(date) %>%
  update_role(date, new_role = "id") %>%
  step_dummy(all_nominal_predictors()) %>%
  step_zv(all_predictors()) %>%
  step_normalize(all_numeric_predictors()) %>%
  step_umap(all_numeric_predictors(), outcome = ridership)
```

A fancy machine learning supervised dimension reduction technique

Other possible steps

```
chi_rec <-
  recipe(ridership ~ ., data = chi_train) %>%
  step_date(date, features = c("dow", "month", "year")) %>%
  step_holiday(date) %>%
  update_role(date, new_role = "id") %>%
  step_dummy(all_nominal_predictors()) %>%
  step_zv(all_predictors()) %>%
  step_normalize(all_numeric_predictors()) %>%
  step_ns(Clark_Lake, deg_free = 10)
```

Nonlinear transforms like **natural splines** and so on.

Recipes are estimated

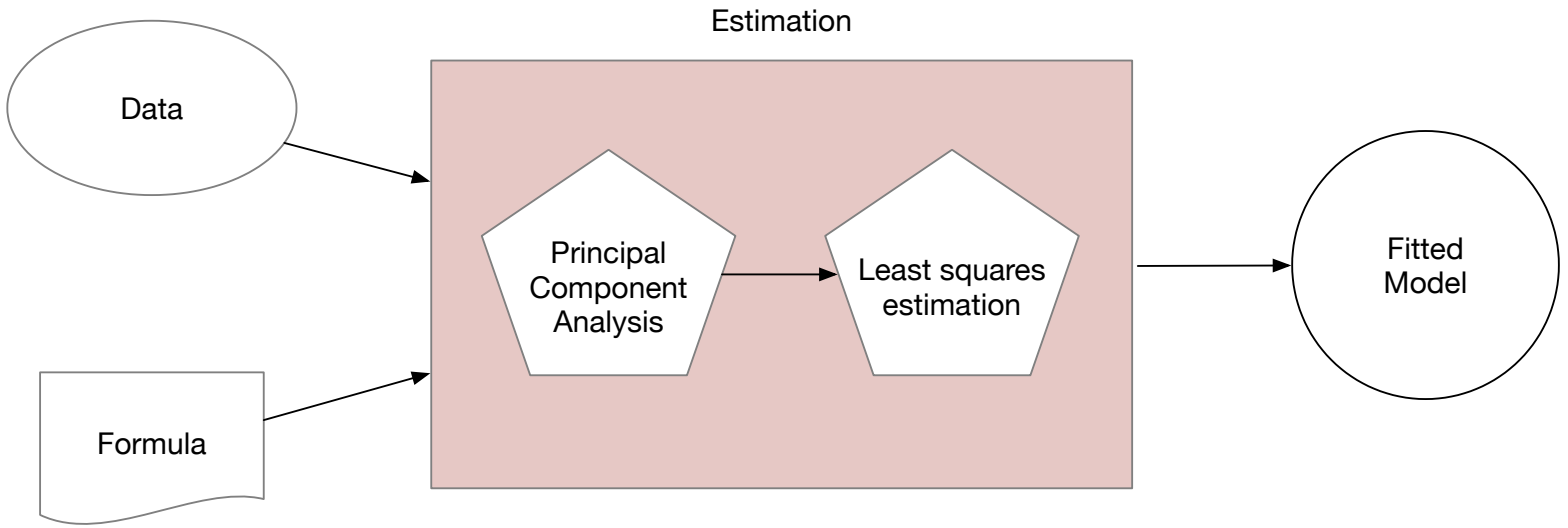
Every preprocessing step in a recipe that involved calculations uses the **training set**. For example:

- Levels of a factor
- Determination of zero-variance
- Normalization
- Feature extraction

and so on.

Once a a recipe is added to a workflow, this occurs when `fit()` is called.

Recipes follow this strategy



Adding recipes to workflows

Let's stick to a linear model for now and add a recipe (instead of a formula):

```
lm_spec <- linear_reg()

chi_wflow <-
  workflow() %>%
  add_model(lm_spec) %>%
  add_recipe(chi_rec)

chi_wflow
```

```
## ══ Workflow ════════════════════════════════════
## Preprocessor: Recipe
## Model: linear_reg()
##
## ── Preprocessor ────────────────────────────────
## 6 Recipe Steps
##
## • step_date()
## • step_holiday()
## • step_dummy()
## • step_zv()
## • step_normalize()
## • step_corr()
##
## ── Model ───────────────────────────────────────
## Linear Regression Model Specification (regression)
##
## Computational engine: lm
```

Estimate via `fit()`

Let's stick to a linear model for now and add a recipe (instead of a formula):

```
chi_fit <- chi_wflow %>% fit(chi_train)
chi_fit
```

```
## == Workflow [trained] ==========================================
## Preprocessor: Recipe
## Model: linear_reg()
##
## -- Preprocessor ----------------------------------------------
## 6 Recipe Steps
##
## • step_date()
## • step_holiday()
## • step_dummy()
## • step_zv()
## • step_normalize()
## • step_corr()
##
## -- Model -----------------------------------------------------
##
## Call:
## stats::lm(formula = ..y ~ ., data = data)
##
## Coefficients:
##       (Intercept)   Washington_Wells         temp_ch
##         13.611685          -0.102278           -0.01
```

Prediction

When `predict()` is called, the fitted recipe is applied to the new data before it is predicted by the model:

```
predict(chi_fit, chi_test)
```

```
## # A tibble: 14 × 1
##    .pred
##    <dbl>
##  1 20.6
##  2 21.4
##  3 21.7
##  4 21.5
##  5 20.8
##  6  8.41
##  7  7.39
##  8 20.2
##  9 21.6
## 10 21.5
## 11 21.2
## 12 20.7
## 13  8.86
## 14  7.60
```

Tidying a recipe

`tidy(recipe)` gives a summary of the steps:

```
tidy(chi_rec)
```

```
## # A tibble: 6 × 6
##   number operation type      trained skip  id
##    <int> <chr>     <chr>     <lgl>   <lgl> <chr>
## 1      1 step      date      FALSE   FALSE date_7QH
## 2      2 step      holiday   FALSE   FALSE holiday_
## 3      3 step      dummy     FALSE   FALSE dummy_2V
## 4      4 step      zv        FALSE   FALSE zv_V2iDF
## 5      5 step      normalize FALSE   FALSE normaliz
## 6      6 step      corr      FALSE   FALSE corr_932
```

After fitting the recipe, you might want access to the statistics from each step. We can pull the fitted recipe from the workflow and choose which step to tidy by number or `id`

```
chi_fit %>%
  extract_recipe() %>%
  tidy(number = 5) # For step normalize
```

```
## # A tibble: 138 × 4
##    terms            statistic value id
##    <chr>            <chr>     <dbl> <chr>
##  1 Austin           mean       1.52 normalize_Q41jN
##  2 Quincy_Wells     mean       5.58 normalize_Q41jN
##  3 Belmont          mean       4.09 normalize_Q41jN
##  4 Archer_35th      mean       2.21 normalize_Q41jN
##  5 Oak_Park         mean       1.32 normalize_Q41jN
##  6 Western          mean       2.87 normalize_Q41jN
##  7 Clark_Lake       mean      13.6  normalize_Q41jN
##  8 Clinton          mean       2.44 normalize_Q41jN
##  9 Merchandise_Mart mean       4.67 normalize_Q41jN
## 10 Irving_Park      mean       3.41 normalize_Q41jN
## # … with 128 more rows
```

Debugging a recipe

90% of the time, you will want to use a workflow to estimate and apply a recipe.

If you have an error, the original recipe object (e.g. `chi_rec`) can be estimated manually with a function called `bake()` (analogous to `fit()`).

This returns the fitted recipe. This can help debug any issues.

Another function (`bake()`) is analogous to `predict()` and gives you the processed data back.

Fun facts about recipes

- Once `fit()` is called on a workflow, changing the model does not re-fit the recipe.

- A list of all known steps is here.

- Some steps can be skipped when using `predict()`.

- The order of the steps matters.

- There are **recipes**-adjacent packages with more steps: **embed**, **timetk**, **textrecipes**, **themis**, and others.

    - Julia and I have written an amazing text processing book: Supervised Machine Learning for Text Analysis in R

- There are a lot of ways to handle categorical predictors even those with novel levels.

- Several **dplyr** steps exist, such as `step_mutate()`.

What resample?

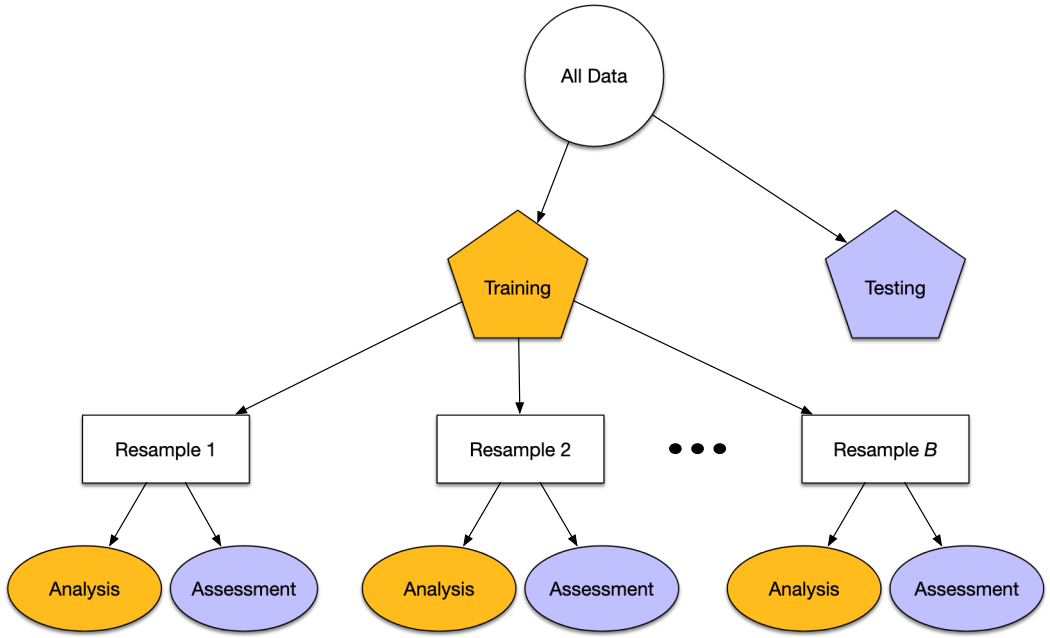
Resampling methods

These are additional data splitting schemes that are applied to the **training** set and are used for **estimating model performance**.

They attempt to simulate slightly different versions of the training set. These versions of the original are split into two model subsets:

- The **analysis set** is used to fit the model (analogous to the training set).

- Performance is determined using the **assessment set**.

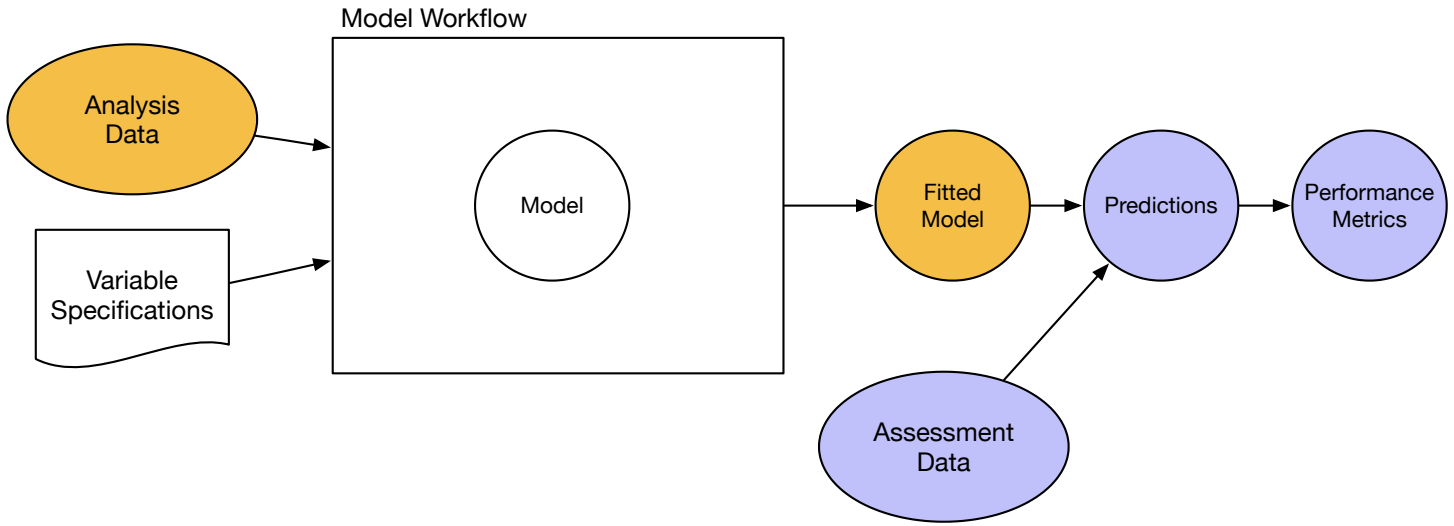


This process is repeated many times.

There are different flavors of resampling but we will focus on one method in these notes.

The model workflow and resampling

All resampling methods repeat this process multiple times:



The final resampling estimate is the average of all of the estimated metrics (e.g. RMSE, etc).

V-Fold cross-validation

Here, we randomly split the training data into V distinct blocks of roughly equal size (AKA the "folds").

- We leave out the first block of analysis data and fit a model.

- This model is used to predict the held-out block of assessment data.

- We continue this process until we've predicted all V assessment blocks

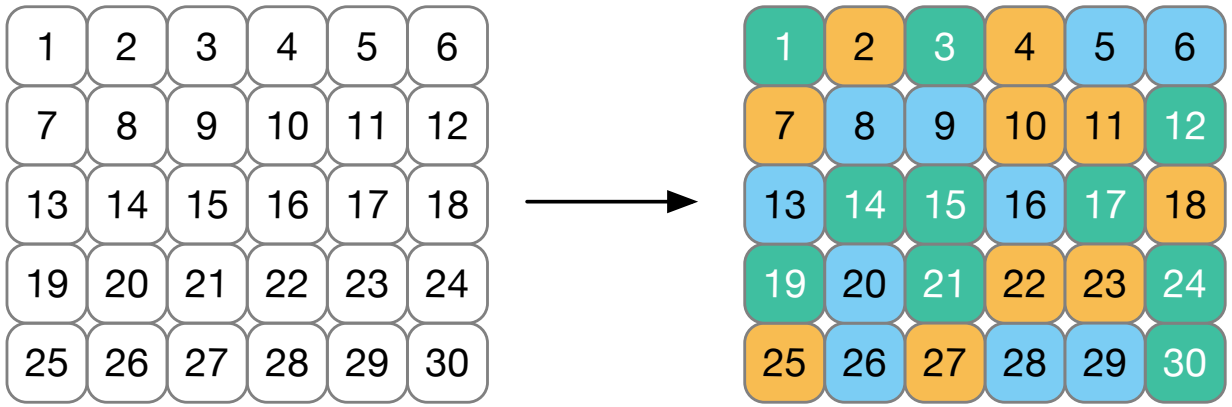
The final performance is based on the hold-out predictions by **averaging** the statistics from the V blocks.

V is usually taken to be 5 or 10 and leave-one-out cross-validation has each sample as a block.

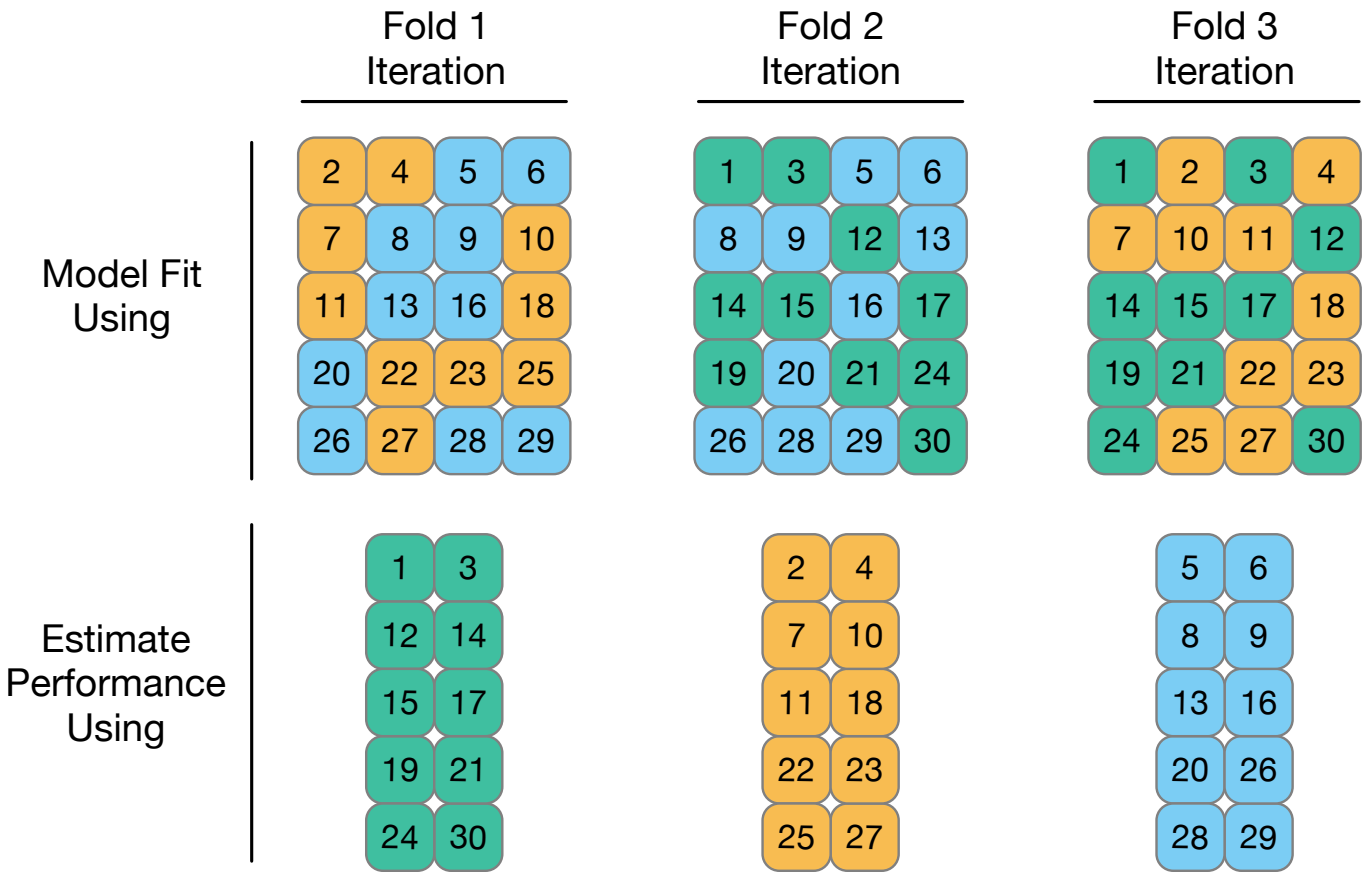
Repeated CV can be used when training set sizes are small. 5 repeats of 10-fold CV averages for 50 sets of metrics.

3-Fold cross-validation with n = 30

Randomly assign each sample to one of three folds



3-Fold cross-validation with n = 30



Resampling results

The goal of resampling is to produce a single estimate of performance for a model.

Even though we end up estimating **V** models (for **V**-fold CV), these models are discarded after we have our performance estimate.

Resampling is basically an empirical simulation system **used to understand how well the model would work on _new data**.

Cross-validating using rsample

rsample has a number of resampling functions built in. One is `vfold_cv()`, for performing V-Fold cross-validation like we've been discussing.

```
set.seed(2453)

cv_splits <- vfold_cv(chi_train) #10-fold is default

cv_splits
```

```
## #  10-fold cross-validation
## # A tibble: 10 × 2
##    splits             id
##    <list>             <chr>
##  1 <split [5115/569]> Fold01
##  2 <split [5115/569]> Fold02
##  3 <split [5115/569]> Fold03
##  4 <split [5115/569]> Fold04
##  5 <split [5116/568]> Fold05
##  6 <split [5116/568]> Fold06
##  7 <split [5116/568]> Fold07
##  8 <split [5116/568]> Fold08
##  9 <split [5116/568]> Fold09
## 10 <split [5116/568]> Fold10
```

Cross-validating Using rsample

- Each individual split object is similar to the `initial_split()` example.
- Use `analysis()` to extract the resample's data used for the fitting process.
- Use `assessment()` to extract the resample's data used for the performance process.

```
cv_splits$splits[[1]]
```

```
## <Analysis/Assess/Total>
## <5115/569/5684>
```

```
cv_splits$splits[[1]] %>%
  analysis() %>%
  dim()
```

```
## [1] 5115   50
```

```
cv_splits$splits[[1]] %>%
  assessment() %>%
  dim()
```

```
## [1] 569  50
```

Time series resampling

Our Chicago data is over time. Regular cross-validation, which uses random sampling, may not be the best idea.

We can emulate our training/test split by making similar resamples.

- Fold 1: Take the first X years of data as the analysis set, the next 2 weeks as the assessment set.

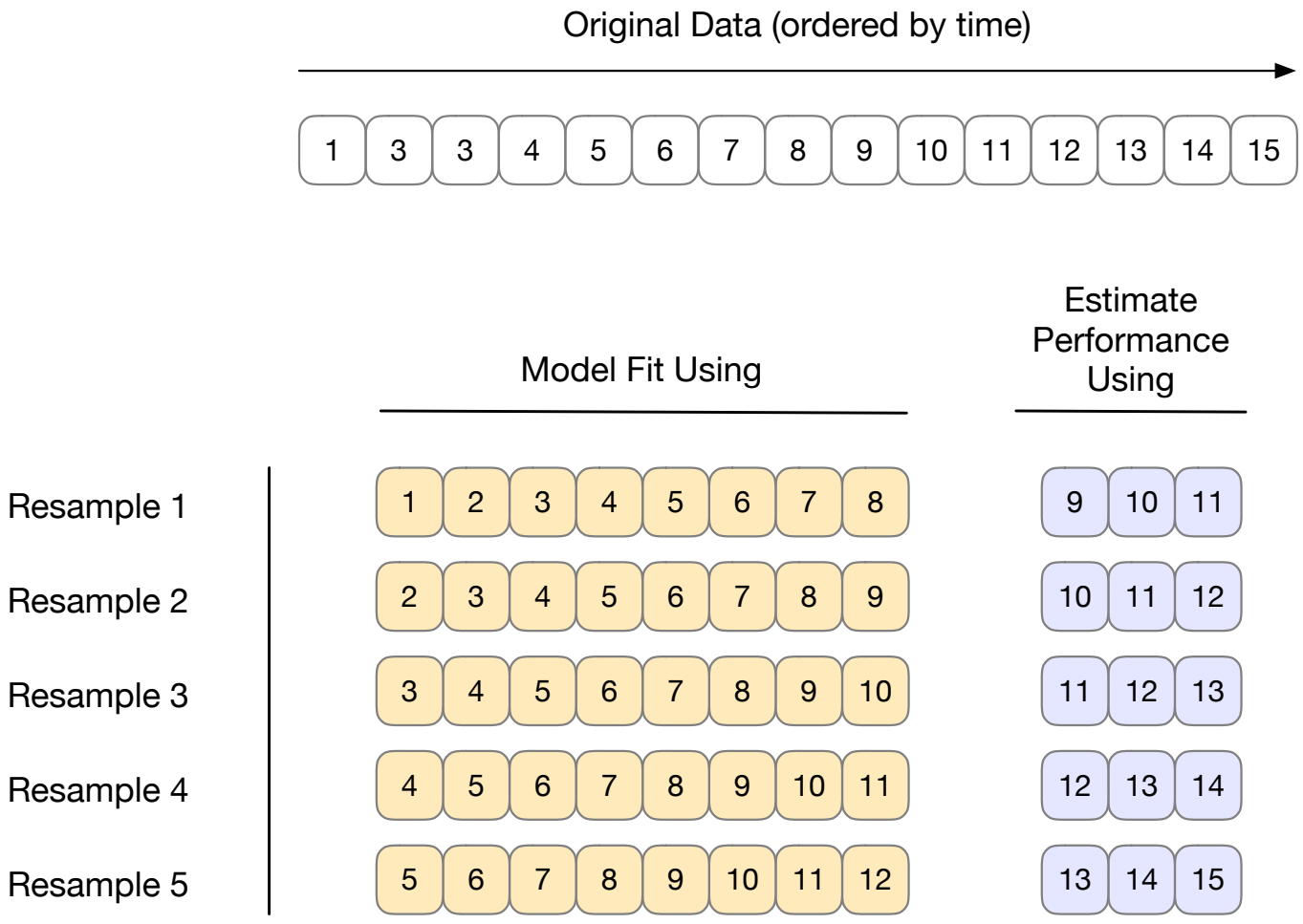
- Fold 2: Take the first X years + 2 weeks of data as the analysis set, the next 2 weeks as the assessment set.

- Fold 3: Take the first X years + 4 weeks of data as the analysis set, the next 2 weeks as the assessment set.

- and so on

Here a small example with a 3 day assessment set

Rolling forecast origin resampling



Using rsample to do this

```
chi_rs <-
  chi_train %>%
  sliding_period(
    index = "date"
  )
```

Use the `date` column to find the date data.

Using rsample to do this

```
chi_rs <-
  chi_train %>%
  sliding_period(
    index = "date",
    period = "week"
  )
```

Our units will be weeks.

Using rsample to do this

```
chi_rs <-
  chi_train %>%
  sliding_period(
    index = "date",
    period = "week",
    lookback = 52 * 15
  )
```

Every analysis set has 15 years of data

Using rsample to do this

```
chi_rs <-
  chi_train %>%
  sliding_period(
    index = "date",
    period = "week",
    lookback = 52 * 15,
    assess_stop = 2
  )
```

Every assessment set has 2 weeks of data

Using rsample to do this

```
chi_rs <-
  chi_train %>%
  sliding_period(
    index = "date",
    period = "week",
    lookback = 52 * 15,
    assess_stop = 2,
    step = 2
  )
```

Increment by 2 weeks so that there are no overlapping assessment sets. For example:

```
chi_rs$splits[[1]] %>% assessment() %>% pluck("date") %>% range()
```

```
## [1] "2016-01-07" "2016-01-20"
```

```
chi_rs$splits[[2]] %>% assessment() %>% pluck("date") %>% range()
```

```
## [1] "2016-01-21" "2016-02-03"
```

Our resampling object

```
chi_rs
```

```
## # Sliding period resampling
## # A tibble: 16 × 2
##    splits             id
##    <list>             <chr>
##  1 <split [5463/14]> Slice01
##  2 <split [5467/14]> Slice02
##  3 <split [5467/14]> Slice03
##  4 <split [5467/14]> Slice04
##  5 <split [5467/14]> Slice05
##  6 <split [5467/14]> Slice06
##  7 <split [5467/14]> Slice07
##  8 <split [5467/14]> Slice08
##  9 <split [5467/14]> Slice09
## 10 <split [5467/14]> Slice10
## 11 <split [5467/14]> Slice11
## 12 <split [5467/14]> Slice12
## 13 <split [5467/14]> Slice13
## 14 <split [5467/14]> Slice14
## 15 <split [5467/14]> Slice15
## 16 <split [5467/11]> Slice16
```

We will fit 16 models on 16 slightly different analysis sets.

Generating the resampling statistics

Let's use the workflow from the last section (`chi_wflow`).

In tidymodels, there is a function called `fit_resamples()` that will do all of this for us:

```
ctrl <- control_resamples(save_pred = TRUE)

chi_res <-
  chi_wflow %>%
  fit_resamples(resamples = chi_rs, control = ctrl)
chi_res
```

```
## # Resampling results
## # Sliding period resampling
## # A tibble: 16 × 5
##    splits             id      .metrics          .notes           .predictions
##    <list>             <chr>   <list>            <list>           <list>
##  1 <split [5463/14]> Slice01 <tibble [2 × 4]> <tibble [0 × 3]> <tibble [14 × 4]>
##  2 <split [5467/14]> Slice02 <tibble [2 × 4]> <tibble [0 × 3]> <tibble [14 × 4]>
##  3 <split [5467/14]> Slice03 <tibble [2 × 4]> <tibble [0 × 3]> <tibble [14 × 4]>
##  4 <split [5467/14]> Slice04 <tibble [2 × 4]> <tibble [0 × 3]> <tibble [14 × 4]>
##  5 <split [5467/14]> Slice05 <tibble [2 × 4]> <tibble [0 × 3]> <tibble [14 × 4]>
##  6 <split [5467/14]> Slice06 <tibble [2 × 4]> <tibble [0 × 3]> <tibble [14 × 4]>
##  7 <split [5467/14]> Slice07 <tibble [2 × 4]> <tibble [0 × 3]> <tibble [14 × 4]>
##  8 <split [5467/14]> Slice08 <tibble [2 × 4]> <tibble [0 × 3]> <tibble [14 × 4]>
```

Getting the results

To obtain the resampling estimates of performance:

```
collect_metrics(chi_res)
```

```
## # A tibble: 2 × 6
##   .metric .estimator  mean     n std_err .config
##   <chr>   <chr>      <dbl> <int>   <dbl> <chr>
## 1 rmse    standard   1.86     16  0.241  Preprocessor1_Model1
## 2 rsq     standard   0.946    16  0.0218 Preprocessor1_Model1
```

To get the holdout predictions:

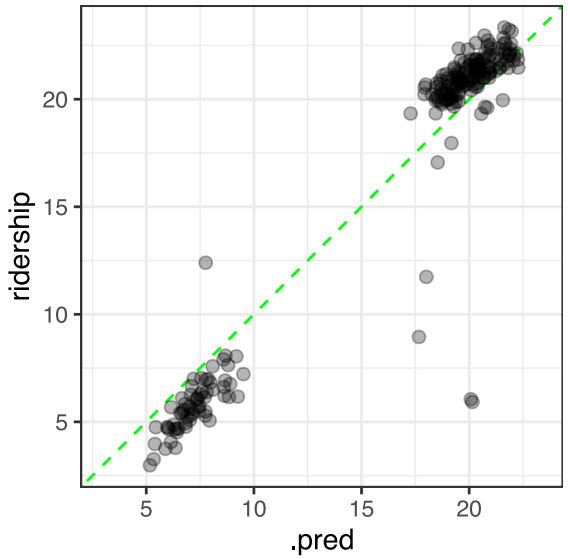
```
chi_pred <- collect_predictions(chi_res)
chi_pred %>% slice(1:4)
```

```
## # A tibble: 4 × 5
##   id      .pred  .row ridership .config
##   <chr>   <dbl> <int>     <dbl> <chr>
## 1 Slice01 20.1   5464     20.4  Preprocessor1_Model1
## 2 Slice01 18.5   5465     20.1  Preprocessor1_Model1
## 3 Slice01  6.84  5466      4.78 Preprocessor1_Model1
## 4 Slice01  5.35  5467      3.26 Preprocessor1_Model1
```

Plot performance

A simple ggplot with a custom `coord_*` can be used.

```
chi_pred %>%
  ggplot(aes(.pred, ridership)) +
  geom_abline(lty = 2, col = "green") +
  geom_point(alpha = 0.3, cex = 2) +
  coord_obs_pred()
```



We can also use the `shinymodels` package to get an interactive version of this plot:

```
library(shinymodels)
explore(chi_res, hover_cols = c(date, ridership))
```

Plotting the metrics over time

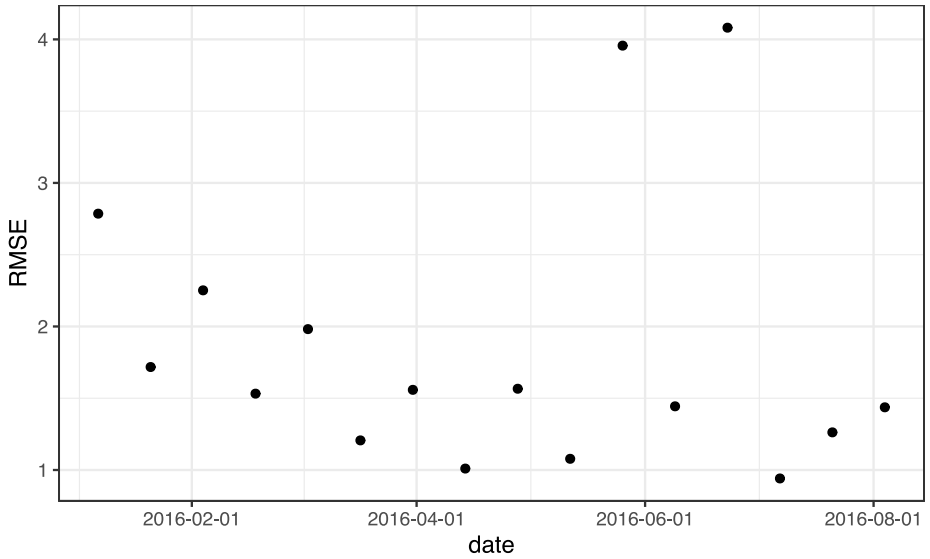
You can get the per-resample metrics and prediction using the `summarize = FALSE` option.

An example function to add them to the results:

```
# Add a date column to time series resampling object metrics
add_date_to_metrics <- function(x, date_col, value = min, ...) {
  res <- collect_metrics(x, summarize = FALSE, ...)
  x %>%
    mutate(
      # Get the assessment set
      holdout = purrr::map(splits, assessment),
      # Keep the date colum
      holdout = purrr::map(holdout, ~ select(.x, all_of(date_col))),
      # Find a date to represent the range
      date = purrr::map(holdout, ~ value(.x[[date_col]]))
    ) %>%
    # date is a nested tibble so unnest then merge with results
   unnest(c(all_of(date_col))) %>%
    select(id, all_of(date_col)) %>%
    full_join(res, by = "id")
}
```

Plotting the metrics over time

```
chi_res %>%
  add_date_to_metrics("date") %>%
  filter(.metric == "rmse") %>%
  ggplot(aes(x = date, y = .estimate)) +
  geom_point() +
  labs(y = "RMSE") +
  scale_x_date(date_breaks = "2 months")
```



Some notes

- These models fits are independent of one another. Parallel processing can be used to significantly speed up the training process.

- The individual models can be saved so you can look at variation in the model parameters or recipes steps.

- If you are interested in a validation set, tidymodels considers that a single resample of the data. Everything else in this chapter works the same.

Tuning parameters

These are model or preprocessing parameters that are important but cannot be estimated directly form the data.

Some examples:

- Tree depth in decision trees.
- Number of neighbors in a K-nearest neighbor model.
- Covariance/correlation matrix structure in mixed models.
- Data distribution in survival models.
- Spline degrees of freedom.

Optimizng tuning parameters

The main approach is to try different values and measure their performance. This can lead us to good values for these parameters.

The main two classes of optimization models are:

- Grid search where a pre-defined set of candidate values are tested.

- Iterative search methods suggest/estimate new values of candidate parameters to evaluate.

Once the value(s) of the parameter(s) are determine, a model can be finalized but fitting the model to the entire training set.

Measuring tuning paramters

We need performance metrics to tell us which candidate values are good and which are not.

Using the test set, or simply re-predicting the training set, are very bad ideas.

Since tuning parameters often control complexity, they can often lead to overfitting.

- This is where the model does very well on the training set but poorly on new data.

Using resampling to estimate performance can help identify parameters that lead to overfitting.

The cost is computational time.

Choosing tuning parameters

Let's take our previous model and add a few changes:

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

chi_rec <-
  recipe(ridership ~ ., data = chi_train) %>%
  step_date(date, features = c("dow", "year")) %>%
  step_holiday(date) %>%
  update_role(date, new_role = "id") %>%
  step_dummy(all_nominal_predictors()) %>%
  step_zv(all_predictors()) %>%
  step_normalize(all_numeric_predictors()) %>%
  step_corr(all_numeric_predictors(), threshold = 0.9)

chi_wflow <-
  workflow() %>%
  add_model(lm_spec) %>%
  add_recipe(chi_rec)
```

Use regularizaed regression

```
lm_spec <-
  linear_reg() %>%
  set_engine("glmnet") #<<

chi_rec <-
  recipe(ridership ~ ., data = chi_train) %>%
  step_date(date, features = c("dow", "year")) %>%
  step_holiday(date) %>%
  update_role(date, new_role = "id") %>%
  step_dummy(all_nominal_predictors()) %>%
  step_zv(all_predictors()) %>%
  step_normalize(all_numeric_predictors()) %>%
  step_corr(all_numeric_predictors(), threshold = 0.9)

chi_wflow <-
  workflow() %>%
  add_model(lm_spec) %>%
  add_recipe(chi_rec)
```

Add model parameters

```
lm_spec <-
  linear_reg(penalty, mixture) %>% #<<
  set_engine("glmnet")

chi_rec <-
  recipe(ridership ~ ., data = chi_train) %>%
  step_date(date, features = c("dow", "year")) %>%
  step_holiday(date) %>%
  update_role(date, new_role = "id") %>%
  step_dummy(all_nominal_predictors()) %>%
  step_zv(all_predictors()) %>%
  step_normalize(all_numeric_predictors()) %>%
  step_corr(all_numeric_predictors(), threshold = 0.9)

chi_wflow <-
  workflow() %>%
  add_model(lm_spec) %>%
  add_recipe(chi_rec)
```

Mark them for tuning

```
lm_spec <-
  linear_reg(penalty = tune(), mixture = tune()) %>% #<<
  set_engine("glmnet")

chi_rec <-
  recipe(ridership ~ ., data = chi_train) %>%
  step_date(date, features = c("dow", "year")) %>%
  step_holiday(date) %>%
  update_role(date, new_role = "id") %>%
  step_dummy(all_nominal_predictors()) %>%
  step_zv(all_predictors()) %>%
  step_normalize(all_numeric_predictors()) %>%
  step_corr(all_numeric_predictors(), threshold = 0.9)

chi_wflow <-
  workflow() %>%
  add_model(lm_spec) %>%
  add_recipe(chi_rec)
```

Remove unneeded step

```
lm_spec <-
  linear_reg(penalty = tune(), mixture = tune()) %>%
  set_engine("glmnet")

chi_rec <-
  recipe(ridership ~ ., data = chi_train) %>%
  step_date(date, features = c("dow", "year")) %>%
  step_holiday(date) %>%
  update_role(date, new_role = "id") %>%
  step_dummy(all_nominal_predictors()) %>%
  step_zv(all_predictors()) %>%
  step_normalize(all_numeric_predictors())


chi_wflow <-
  workflow() %>%
  add_model(lm_spec) %>%
  add_recipe(chi_rec)
```

Add a spline step (just for demonstration)

```
lm_spec <-
  linear_reg(penalty = tune(), mixture = tune()) %>%
  set_engine("glmnet")

chi_rec <-
  recipe(ridership ~ ., data = chi_train) %>%
  step_date(date, features = c("dow", "year")) %>%
  step_holiday(date) %>%
  update_role(date, new_role = "id") %>%
  step_dummy(all_nominal_predictors()) %>%
  step_zv(all_predictors()) %>%
  step_ns(temp, deg_free = tune()) %>%  #<<
  step_normalize(all_numeric_predictors())

chi_wflow <-
  workflow() %>%
  add_model(lm_spec) %>%
  add_recipe(chi_rec)
```

Grid search

This is the most basic (but very effective) way for tuning models.

tidymodels has pre-defined information on tuning parameters, such as their type, range, transformations, etc.

A grid can be created manually or automatically.

The `parameters()` function extracts the tuning parameters and the info.

The `grid_*()` functions can make a grid.

Manual grid - get parameters

```
chi_wflow %>%
  parameters()
```

```
## Warning: `parameters.workflow()` was deprecated in
## Please use `hardhat::extract_parameter_set_dials()`
```

This type of object can be updated (e.g. to change the ranges, etc)

```
## Collection of 3 parameters for tuning
##
##  identifier     type    object
##     penalty  penalty nparam[+]
##     mixture  mixture nparam[+]
##    deg_free deg_free nparam[+]
```

Manual grid - create grid

```
set.seed(2)
grid <-
  chi_wflow %>%
  parameters() %>%
  grid_latin_hypercube(size = 25)

grid
```

This is a type of space-filling design.

It tends to do much better than random grids and is (usually) more efficient than regular grids.

```
## Warning: `parameters.workflow()` was deprecated in
## Please use `hardhat::extract_parameter_set_dials()`
```

```
## # A tibble: 25 × 3
##          penalty mixture deg_free
##            <dbl>   <dbl>    <int>
##  1 0.124           0.309        2
##  2 0.0000000474    0.269        5
##  3 0.0000000101    0.810        4
##  4 0.0000140       0.527       11
##  5 0.172           0.897        3
##  6 0.0000329       0.701        3
##  7 0.00000000610   0.590       13
##  8 0.0141          0.102        6
##  9 0.00513         0.405       13
## 10 0.000667        0.145        1
## # … with 15 more rows
```

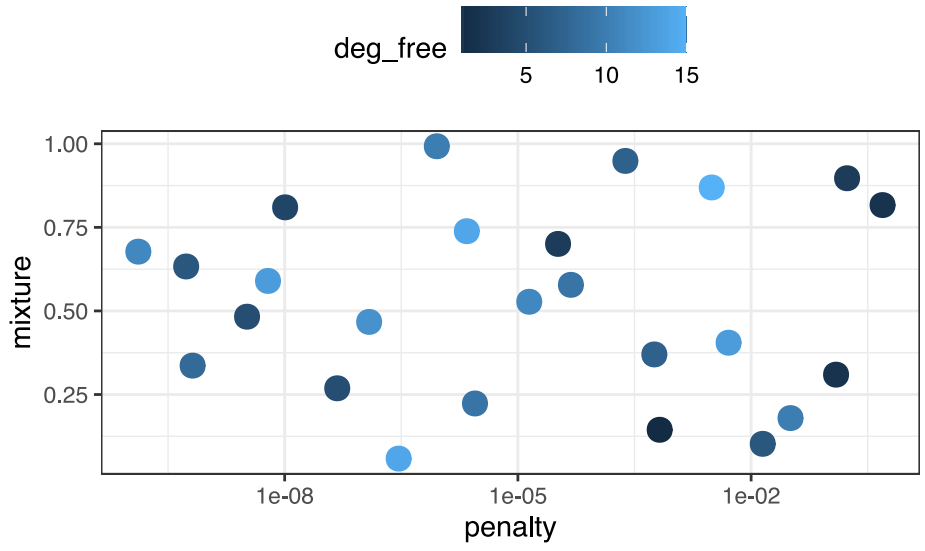
The results

```
set.seed(2)
grid <-
  chi_wflow %>%
  parameters() %>%
  grid_latin_hypercube(size = 25)

grid %>%
  ggplot(aes(penalty, mixture, col = deg_free)) +
  geom_point(cex = 4) +
  scale_x_log10()
```

Note that `penalty` was generated in log-10 units.

```
## Warning: `parameters.workflow()` was deprecated in
## Please use `hardhat::extract_parameter_set_dials()`
```



Grid search

The `tune_*()` functions can be used to tune models.

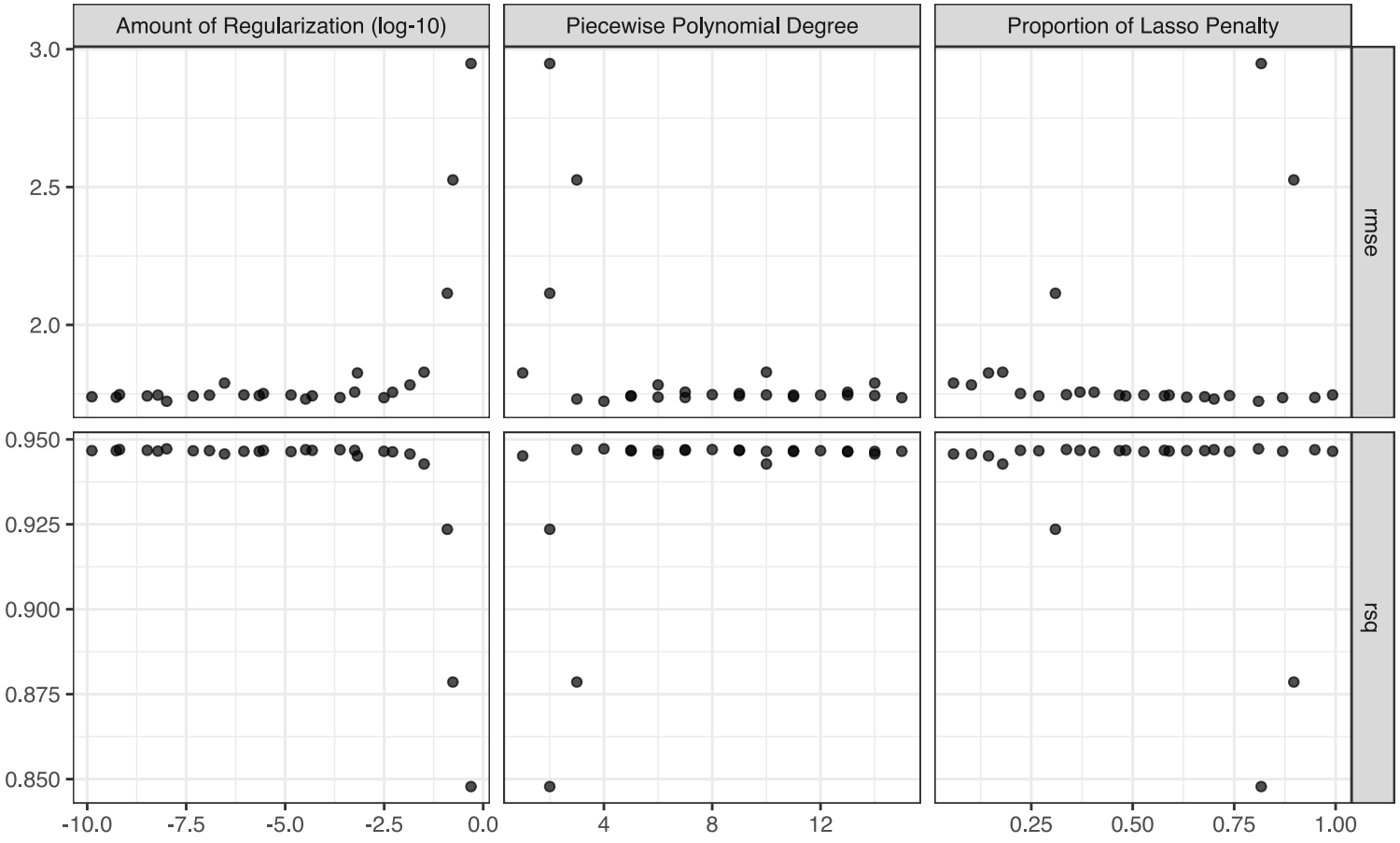
`tune_grid()` is pretty representative of their syntax (and is similar to `last_fit()`):

```
ctrl <- control_grid(save_pred = TRUE)
set.seed(9)
chi_res <-
  chi_wflow %>%
  tune_grid(resamples = chi_rs, grid = grid) # 'grid' = integer for automatic grids
chi_res
```

```
## # Tuning results
## # Sliding period resampling
## # A tibble: 16 × 4
##    splits             id      .metrics          .notes
##    <list>             <chr>   <list>            <list>
##  1 <split [5463/14]> Slice01 <tibble [50 × 7]> <tibble [0 × 3]>
##  2 <split [5467/14]> Slice02 <tibble [50 × 7]> <tibble [0 × 3]>
##  3 <split [5467/14]> Slice03 <tibble [50 × 7]> <tibble [0 × 3]>
##  4 <split [5467/14]> Slice04 <tibble [50 × 7]> <tibble [0 × 3]>
##  5 <split [5467/14]> Slice05 <tibble [50 × 7]> <tibble [0 × 3]>
##  6 <split [5467/14]> Slice06 <tibble [50 × 7]> <tibble [0 × 3]>
##  7 <split [5467/14]> Slice07 <tibble [50 × 7]> <tibble [0 × 3]>
##  8 <split [5467/14]> Slice08 <tibble [50 × 7]> <tibble [0 × 3]>
```

Grid results

```
autoplot(chi_res)
```



ENHANCE

```
autoplot(chi_res, metric = "rmse")  +
  ylim(c(1.7, 1.85))
```

Returning results

```
collect_metrics(chi_res)
```

```
## # A tibble: 50 × 9
##          penalty mixture deg_free .metric .estimator  mean     n std_err .config
##            <dbl>   <dbl>    <int> <chr>   <chr>      <dbl> <int>   <dbl> <chr>
##  1 0.124           0.309        2 rmse    standard   2.11     16  0.266  Prepro…
##  2 0.124           0.309        2 rsq     standard   0.924    16  0.0249 Prepro…
##  3 0.491           0.817        2 rmse    standard   2.95     16  0.402  Prepro…
##  4 0.491           0.817        2 rsq     standard   0.848    16  0.0502 Prepro…
##  5 0.0000000474    0.269        5 rmse    standard   1.74     16  0.241  Prepro…
##  6 0.0000000474    0.269        5 rsq     standard   0.947    16  0.0213 Prepro…
##  7 0.00000000328   0.483        5 rmse    standard   1.74     16  0.241  Prepro…
##  8 0.00000000328   0.483        5 rsq     standard   0.947    16  0.0212 Prepro…
##  9 0.0000000101    0.810        4 rmse    standard   1.72     16  0.241  Prepro…
## 10 0.0000000101    0.810        4 rsq     standard   0.947    16  0.0210 Prepro…
## # … with 40 more rows
```

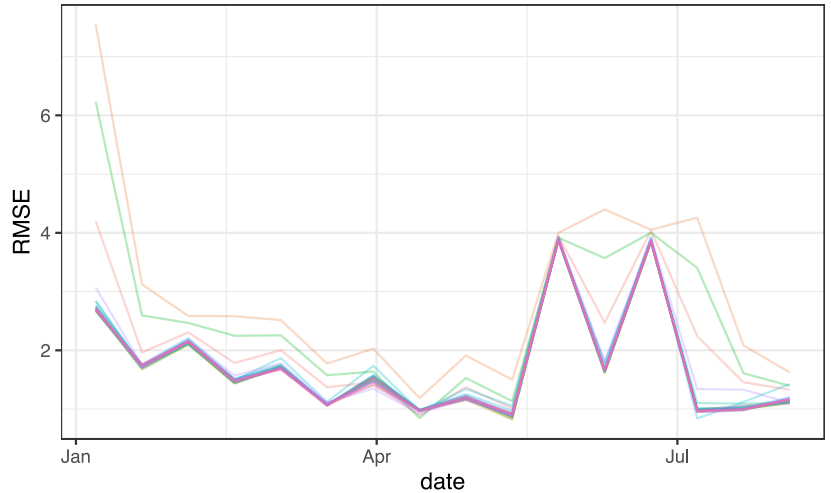
Returning results

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

```
## # A tibble: 800 × 8
##    id      penalty mixture deg_free .metric .estimator .estimate .config
##    <chr>     <dbl>   <dbl>    <int> <chr>   <chr>          <dbl> <chr>
##  1 Slice01   0.124   0.309        2 rmse    standard       4.20  Preprocessor01…
##  2 Slice01   0.124   0.309        2 rsq     standard       0.722 Preprocessor01…
##  3 Slice02   0.124   0.309        2 rmse    standard       1.96  Preprocessor01…
##  4 Slice02   0.124   0.309        2 rsq     standard       0.961 Preprocessor01…
##  5 Slice03   0.124   0.309        2 rmse    standard       2.30  Preprocessor01…
##  6 Slice03   0.124   0.309        2 rsq     standard       0.915 Preprocessor01…
##  7 Slice04   0.124   0.309        2 rmse    standard       1.78  Preprocessor01…
##  8 Slice04   0.124   0.309        2 rsq     standard       0.965 Preprocessor01…
##  9 Slice05   0.124   0.309        2 rmse    standard       2.00  Preprocessor01…
## 10 Slice05   0.124   0.309        2 rsq     standard       0.938 Preprocessor01…
## # … with 790 more rows
```

Metrics over time

```
chi_res %>%
  add_date_to_metrics("date") %>%
  filter(.metric == "rmse") %>%
  ggplot(aes(x = date, y = .estimate,
             group = .config, col = .config)) +
  geom_line(show.legend = FALSE, alpha = .3) +
  labs(y = "RMSE")
```



Picking a parameter combination

You can create a tibble of your own or use one of the `tune::select_*()` functions:

```
show_best(chi_res, metric = "rmse")
```

```
## # A tibble: 5 × 9
##    penalty mixture deg_free .metric .estimator  mean     n std_err .config
##      <dbl>   <dbl>    <int> <chr>   <chr>      <dbl> <int>   <dbl> <chr>
## 1 1.01e- 8   0.810        4 rmse    standard    1.72    16   0.241 Preprocessor…
## 2 3.29e- 5   0.701        3 rmse    standard    1.73    16   0.241 Preprocessor…
## 3 3.11e- 3   0.869       15 rmse    standard    1.74    16   0.242 Preprocessor…
## 4 2.41e- 4   0.949        7 rmse    standard    1.74    16   0.240 Preprocessor…
## 5 5.41e-10   0.633        6 rmse    standard    1.74    16   0.241 Preprocessor…
```

```
smallest_rmse <-
  select_best(chi_res, metric = "rmse")
smallest_rmse
```

```
## # A tibble: 1 × 4
##        penalty mixture deg_free .config
##          <dbl>   <dbl>    <int> <chr>
## 1 0.0000000101   0.810        4 Preprocessor03_Model1
```

Updating the workflow and final fit

```
chi_wflow <-
  chi_wflow %>%
  finalize_workflow(smallest_rmse)

test_res <-
  chi_wflow %>%
  last_fit(split = chi_split)
test_res
```

```
## # Resampling results
## # Manual resampling
## # A tibble: 1 × 6
##   splits            id               .metrics .notes   .predictions .workflow
##   <list>            <chr>            <list>   <list>   <list>       <list>
## 1 <split [5684/14]> train/test split <tibble> <tibble> <tibble>     <workflow>
```

The workflow, fit using the training set:

```
final_chi_wflow <-
  test_res$.workflow[[1]]
```

Two-week test set results

```
collect_metrics(test_res)
```

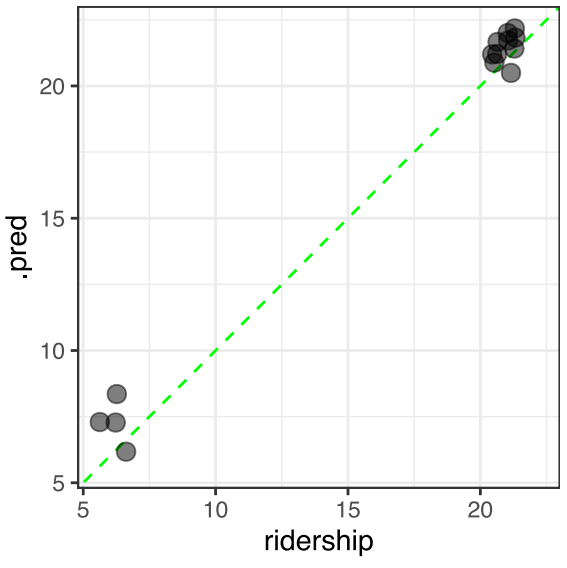
```
## # A tibble: 2 × 4
##   .metric .estimator .estimate .config
##   <chr>   <chr>          <dbl> <chr>
## 1 rmse    standard       0.976 Preprocessor1_Model1
## 2 rsq     standard       0.990 Preprocessor1_Model1
```

```
# Resampling results
show_best(chi_res, metric = "rmse", n = 1)
```

```
## # A tibble: 1 × 9
##        penalty mixture deg_free .metric .estimator  mean     n std_err .config
##          <dbl>   <dbl>    <int> <chr>   <chr>      <dbl> <int>   <dbl> <chr>
## 1 0.0000000101   0.810        4 rmse    standard    1.72    16   0.241 Preproce…
```

Two-week test set results

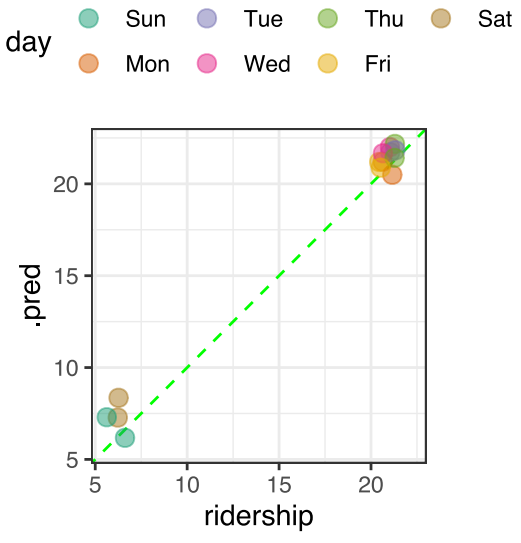
```
test_res %>%
  collect_predictions() %>%
  ggplot(aes(ridership, .pred)) +
  geom_abline(col = "green", lty = 2) +
  geom_point(cex = 3, alpha = 0.5) +
  coord_obs_pred()
```



Two-week test set results

```
library(lubridate)
test_values <-
  final_chi_wflow %>%
  augment(testing(chi_split)) %>%
  mutate(day = wday(date, label = TRUE))

test_values %>%
  ggplot(aes(ridership, .pred, col = day)) +
  geom_abline(col = "green", lty = 2) +
  geom_point(cex = 3, alpha = 0.5) +
  coord_obs_pred() +
  scale_color_brewer(palette = "Dark2")
```



Two-week test set results

```
test_values %>%
  ggplot(aes(date, ridership)) +
  geom_point(aes(col = day),
             cex = 3, alpha = 0.5) +
  geom_line(aes(y = .pred)) +
  scale_color_brewer(palette = "Dark2")
```