Tidymodels.org

Koji Mizumura

2020-04-30 - 2020-05-18

Table of Contents

# Build a model {# Build-a-model}

## Introduction

How do you create a statistical model using tidymodels? IN this article, we will walk you through the steps. We start with data for modeling, learn how to specify and train models with different engines using the [parsnip package](parsnip%20package), and understand why these functions are desgined this way.

To use code in this article, you will need to install the following packages; readr, rstatnarm, and tidymodels.

# tictoc::tic()  
pacman::p\_load(tidymodels, readr)  
# tictoc::toc()

## The Sea Urchins data

Let’s use the data from Constable(1993) to explore how three different feeding regimes affect the size of sea urchins over time. The initial size of the sea urchins at the beginning of the experiment probably affects how big they grow as they are fed.

To start, let’s rad our urchins data into R, which we’ll do by providing readr::read\_csv() with a url where our CSV data is located (“<https://tidymodels.org/start/models/urchins.csv>”):

urchins <-   
 # Data were assembled for a tutorial  
 # at https://www.flutterbys.com.au/stats/tut/tut7.5a.html  
 read\_csv("https://tidymodels.org/start/models/urchins.csv") %>%   
 # chnge the names to be a little more verbose  
 setNames(c("food\_regime", "initial\_volume", "width")) %>%   
 # factors are very helpful for modeling, so we convert one column  
 mutate(  
 food\_regime = factor(food\_regime, levels = c("Initial", "Low", "High")))  
#> Parsed with column specification:  
#> cols(  
#> TREAT = col\_character(),  
#> IV = col\_double(),  
#> SUTW = col\_double()  
#> )

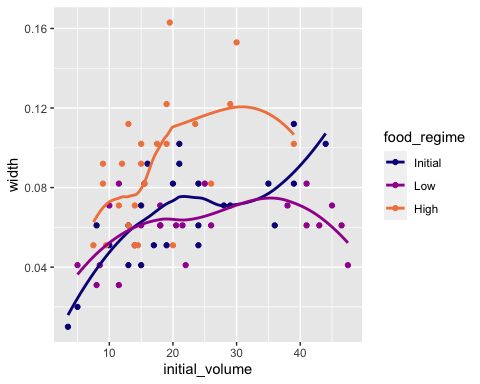
urchins  
## # A tibble: 72 x 3  
## food\_regime initial\_volume width  
## <fct> <dbl> <dbl>  
## 1 Initial 3.5 0.01   
## 2 Initial 5 0.02   
## 3 Initial 8 0.061  
## 4 Initial 10 0.051  
## 5 Initial 13 0.041  
## 6 Initial 13 0.061  
## 7 Initial 15 0.041  
## 8 Initial 15 0.071  
## 9 Initial 16 0.092  
## 10 Initial 17 0.051  
## # … with 62 more rows

The urchins data is a [tibble](https://tibble.tidyverse.org/index.html). If you are new to tibbles, they best place to start is the tibbles chapter in R for Data Science. For each of the 72 urchins, we know their:

* erimental feeding regime group (food\_regime: either Initial, Low, or High),
* size in milliliters at the start of the experiment (initial\_volume), and
* suture width at the end of the experiment (width).

As a first step in modeling, it’s always a good idea to plot the data:

ggplot(urchins,  
 aes(x = initial\_volume,  
 y = width,  
 group = food\_regime,  
 col = food\_regime))+  
 geom\_point()+  
 geom\_smooth(se = FALSE, #method = lm  
 )+  
 # hrbrthemes::theme\_ft\_rc()  
 scale\_color\_viridis\_d(option = "plasma", end = .7)



We can see that urchins that were larger in volume at the start of the experiment tended to have wider sutures at the end, but the slopes of the lines look different so this effect may depend on the feeding regime condition.

## Build and fit a model

A standard two-way analysis of variance (ANOVA) model makes sense for this dataset because we have continuous predictor and a categorical predictor. Since the slopes appear to be different for at least two of the feeding regimes, let’s build a model that allows for two-way interactions. Specifying an R formula with our variables in this way:

width ~ initial\_volume + food\_regime

allows our regression model depending on initial volume to have separate slopes and intercepts for each food regime.

For this kind of model, ordinary least squares is a good initial approach. With tidymodels, we start by specifying the functional form of the model that we want using the parsnip package. Since there is a numeric outcome and the model should be linear with slopes and intercepts, the model type is “linear regression”. We can declare this with:

linear\_reg()  
## Linear Regression Model Specification (regression)

That is pretty underwhelming since, on its own, it doesn’t really do much. However, now that the type of model has been specified, a method for fitting or training the model can be stated using the engine. The engine value is often a mash-up of the software that can be used to fit or train the model as well as the estimation method. For example, to use ordinary least squares, we can set the engine to be lm:

linear\_reg() %>%   
 set\_engine("lm")  
## Linear Regression Model Specification (regression)  
##   
## Computational engine: lm

The documentation page for linear\_reg() lists the possible engines. We’ll save this model object as lm\_mod.

lm\_mod <-   
 linear\_reg() %>%  
 set\_engine("lm")

From here, the model can be estimated or trained using the fit() function.

lm\_fit <-   
 lm\_mod %>%   
 fit(width ~ initial\_volume\*food\_regime, data = urchins)  
  
lm\_fit   
## parsnip model object  
##   
## Fit time: 6ms   
##   
## Call:  
## stats::lm(formula = formula, data = data)  
##   
## Coefficients:  
## (Intercept) initial\_volume   
## 0.0331216 0.0015546   
## food\_regimeLow food\_regimeHigh   
## 0.0197824 0.0214111   
## initial\_volume:food\_regimeLow initial\_volume:food\_regimeHigh   
## -0.0012594 0.0005254

Perhaps our analysis requires a description of the model parameter estimates and their statistical properties. Although the summary() function for lm objects can provide that, it gives the results back in an unwieldy format. Many models have a tidy() method that provides the summary results in a more predictable and useful format (e.g. a data frame with standard column names):

tidy(lm\_fit)  
## # A tibble: 6 x 5  
## term estimate std.error statistic p.value  
## <chr> <dbl> <dbl> <dbl> <dbl>  
## 1 (Intercept) 0.0331 0.00962 3.44 0.00100   
## 2 initial\_volume 0.00155 0.000398 3.91 0.000222  
## 3 food\_regimeLow 0.0198 0.0130 1.52 0.133   
## 4 food\_regimeHigh 0.0214 0.0145 1.47 0.145   
## 5 initial\_volume:food\_regimeLow -0.00126 0.000510 -2.47 0.0162   
## 6 initial\_volume:food\_regimeHigh 0.000525 0.000702 0.748 0.457

## Use a model to predict

This fitted object lm\_fit has the lm model output built-in, which you can access with lm\_fit$fit, but there are some benefits to using the fitted parsnp model object when it comes to predicting.

Suppose that, for a publication, it would be particularly interesting to make a plot of the mean body size for urchins that started the experiment with an initial volume of 20ml. To create such a graph, we start with some new example data that we will make predictions for, to show in our graph:

new\_points <- expand.grid(  
 initial\_volume = 20,  
 food\_regime = c("Initial", "Low", "High")  
)

To get one predicted results, we can use the predict() funtio to find the mean values at 20ml.

It is also important to communicate the variability, so we also need to find the predicted confidence intervals. If we had used lm()to fit the model directly, a few minutes of reading the documentation page for predict.lm() would explain how to do this. However, if we decide to use a different model to estimate urchin size (spoiler: we will!), it is likely that a completely different syntax would be required.

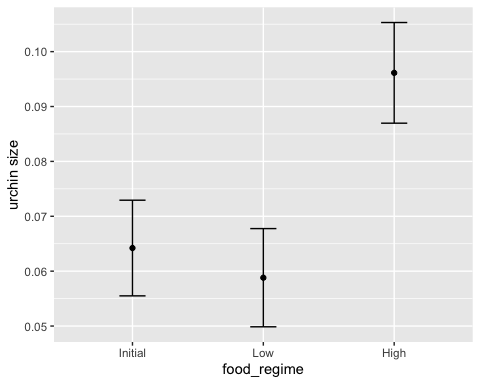
Instead, with tidymodels, the types of predicted values are standardized so that we can use the same syntax to get these values.

First, let’s generate the mean body width values:

mean\_pred <-   
 predict(lm\_fit, new\_data = new\_points)  
  
mean\_pred  
## # A tibble: 3 x 1  
## .pred  
## <dbl>  
## 1 0.0642  
## 2 0.0588  
## 3 0.0961

When making predictions, the tidymodels convention is to always produce a tibble of results with standardized column names. This makes it easy to combine the original data and the predictions in a usable format:

conf\_int\_pred <- predict(lm\_fit,   
 new\_data = new\_points,   
 type = "conf\_int")  
conf\_int\_pred  
## # A tibble: 3 x 2  
## .pred\_lower .pred\_upper  
## <dbl> <dbl>  
## 1 0.0555 0.0729  
## 2 0.0499 0.0678  
## 3 0.0870 0.105  
#> # A tibble: 3 x 2  
#> .pred\_lower .pred\_upper  
#> <dbl> <dbl>  
#> 1 0.0555 0.0729  
#> 2 0.0499 0.0678  
#> 3 0.0870 0.105  
  
# Now combine:   
plot\_data <-   
 new\_points %>%   
 bind\_cols(mean\_pred) %>%   
 bind\_cols(conf\_int\_pred)  
  
# library(hrbrthemes)  
# library(thematic)  
# thematic\_on(  
# bg = "#222222", fg = "white", accent = "#0CE3AC",  
# font = font\_spec("Oxanium", scale = 1.25)  
# )  
  
# and plot:  
ggplot(plot\_data, aes(x = food\_regime)) +   
 geom\_point(aes(y = .pred)) +   
 geom\_errorbar(aes(ymin = .pred\_lower,   
 ymax = .pred\_upper),  
 width = .2) +   
 labs(y = "urchin size")



# Model with a different engine

Every one on your team is happy with that plot except that one person who jut read their first book on Bayesian analysis. They are interested in knowing if the results would be different if the model were estimated using a Bayesian approach. In such an analysis, a prior distribution needs to be declared for each model parameter that represents the possible values of the parameters (before being exposed to the observed data). After some discussion, the group agrees that the priors should be bell-shaped but, since no one has any idea what the range of values should be, to take a conservative approach and make the priors wide using a Cauchy distribution (which is the same as a t-distribution with a single degree of freedom).

The documentation on the rstanarm package shows us that the stan\_glm() function can be used to estimate this model, and that the function arguments that need to be specified are called prior and prior\_intercept. It turns out that linear\_reg() has a stan engine. Since these prior distribution arguments are specific to the Stan software, they are passed as arguments to parsnip::set\_engine(). After that, the same exact fit() call is used:

# see the prior distribution  
prior\_dist <- rstanarm::student\_t(df = 1)  
set.seed(123)  
  
# make the parsnip model  
bayes\_mod <-   
 linear\_reg() %>%   
 set\_engine("stan",  
 prior\_intercept = prior\_dist,  
 prior = prior\_dist)  
  
# train the model  
bayes\_fit <-   
 bayes\_mod %>%   
 fit(width ~ initial\_volume\*food\_regime, data = urchins)  
  
print(bayes\_fit, digits = 5)  
## parsnip model object  
##   
## Fit time: 2.1s   
## stan\_glm  
## family: gaussian [identity]  
## formula: width ~ initial\_volume \* food\_regime  
## observations: 72  
## predictors: 6  
## ------  
## Median MAD\_SD   
## (Intercept) 0.03452 0.00883  
## initial\_volume 0.00150 0.00037  
## food\_regimeLow 0.01805 0.01221  
## food\_regimeHigh 0.01934 0.01367  
## initial\_volume:food\_regimeLow -0.00119 0.00047  
## initial\_volume:food\_regimeHigh 0.00061 0.00065  
##   
## Auxiliary parameter(s):  
## Median MAD\_SD   
## sigma 0.02121 0.00186  
##   
## ------  
## \* For help interpreting the printed output see ?print.stanreg  
## \* For info on the priors used see ?prior\_summary.stanreg

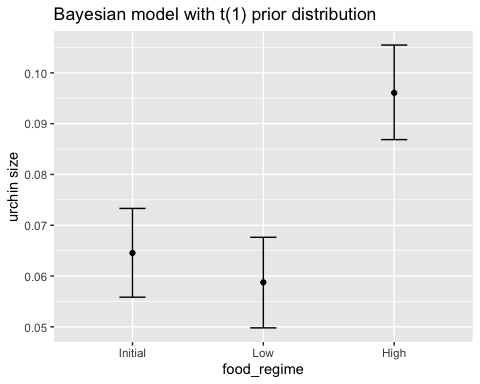
This kind of Bayesian analysis (like many models) involves randomly generated numbers in its fitting procedure. We can use set.seed() to ensure that the same (pseudo-)random numbers are generated each time we run this code. The number 123 isn’t special or related to our data; it is just a “seed” used to choose random numbers.

To update the parameter table, the tidy()method is once again used:

tidy(bayes\_fit, intervals = TRUE)  
## # A tibble: 6 x 5  
## term estimate std.error lower upper  
## <chr> <dbl> <dbl> <dbl> <dbl>  
## 1 (Intercept) 0.0345 0.00883 0.0200 0.0490   
## 2 initial\_volume 0.00150 0.000369 0.000895 0.00212   
## 3 food\_regimeLow 0.0181 0.0122 -0.00181 0.0380   
## 4 food\_regimeHigh 0.0193 0.0137 -0.00317 0.0420   
## 5 initial\_volume:food\_regimeLow -0.00119 0.000472 -0.00199 -0.000413  
## 6 initial\_volume:food\_regimeHigh 0.000610 0.000651 -0.000490 0.00170

A goal of the tidymodels packages is that the interfaces to common tasks are standardized (as seen in the tidy() results above). The same is true for getting predictions; we can use the same code even though the underlying packages use very different syntax:

bayes\_plot\_data <-   
 new\_points %>%   
 bind\_cols(predict(bayes\_fit, new\_data = new\_points)) %>%   
 bind\_cols(predict(bayes\_fit, new\_data = new\_points, type = "conf\_int"))  
  
bayes\_plot\_data  
## initial\_volume food\_regime .pred .pred\_lower .pred\_upper  
## 1 20 Initial 0.06454807 0.05583641 0.07330970  
## 2 20 Low 0.05875854 0.04979481 0.06762928  
## 3 20 High 0.09607671 0.08686153 0.10547756  
  
ggplot(bayes\_plot\_data,   
 aes(x = food\_regime))+  
 geom\_point(aes(y =.pred))+  
 geom\_errorbar(aes(ymin = .pred\_lower,  
 ymax = .pred\_upper),  
 width = .2)+  
 labs(y = "urchin size")+  
 ggtitle("Bayesian model with t(1) prior distribution")



## Why does it work that way?

The extra step of defining the model using a function like linear\_reg() might seem superfluous since a call to lm() is much more succinct. However, the problem with standard modeling functions is that they don’t separate what you want to do from the execution. For example, the process of executing a formula has to happen repeatedly across model calls even when the formula does not change; we can’t recycle those computations.

Also, using the tidymodels framework, we can do some interesting things by incrementally creating a model (instead of using single function call). Model tuning with tidymodels uses the specification of the model to declare what parts of the model should be tuned. That would be very difficult to do if linear\_reg() immediately fit the model.

If you are familiar with the tidyverse, you may have noticed that our modeling code uses the magrittr pipe (%>%). With dplyr and other tidyverse packages, the pipe works well because all of the functions take the data as the first argument. For example:

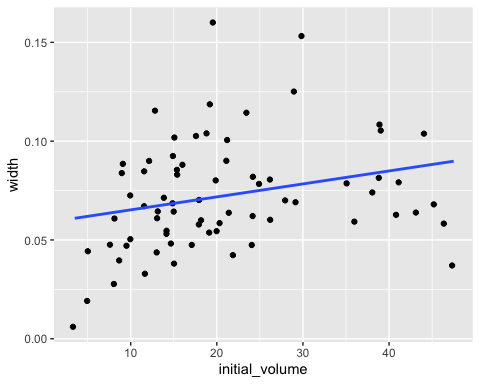
urchins %>%   
 group\_by(food\_regime) %>%   
 summarize(med\_vol = median(initial\_volume))  
## # A tibble: 3 x 2  
## food\_regime med\_vol  
## <fct> <dbl>  
## 1 Initial 20.5  
## 2 Low 19.2  
## 3 High 15

whereas the modeling code uses the pipe to pass around the moel object:

bayes\_mod %>%   
 fit(width ~ initial\_volume \* food\_regime, data = urchins)  
## parsnip model object  
##   
## Fit time: 2s   
## stan\_glm  
## family: gaussian [identity]  
## formula: width ~ initial\_volume \* food\_regime  
## observations: 72  
## predictors: 6  
## ------  
## Median MAD\_SD  
## (Intercept) 0.0 0.0   
## initial\_volume 0.0 0.0   
## food\_regimeLow 0.0 0.0   
## food\_regimeHigh 0.0 0.0   
## initial\_volume:food\_regimeLow 0.0 0.0   
## initial\_volume:food\_regimeHigh 0.0 0.0   
##   
## Auxiliary parameter(s):  
## Median MAD\_SD  
## sigma 0.0 0.0   
##   
## ------  
## \* For help interpreting the printed output see ?print.stanreg  
## \* For info on the priors used see ?prior\_summary.stanreg

This may seem jarring if you have used dplyr, but it is extremely similar to how ggplot2 operates:

ggplot(urchins,  
 aes(initial\_volume, width)) + # returns a ggplot object   
 geom\_jitter() + # same  
 geom\_smooth(method = lm, se = FALSE) # same



# Process your data with recipes

## Introduction

In our [build a model article](#Build-a-model), we learned how to specify and train models with different engines using the [parsnip package](https://tidymodels.github.io/parsnip/). In this article, we will explorer another tidymodels packages recipes, which is designed to help you preprocess your data before traing your model. Recipes are built as a series of preprocess steps, such as:

* converting qualitative predictors to indicate variables(also known as dummy variables)
* transforming data to be on a different scale (e.g., taking the logarithm of a variable)
* transforming whole groups of predictors together,
* extracting key features from raw variables (e.g., getting the day of the week out of a date variable)

and so on. If you are familiar with R’s formula interface, a lot of this might sound familiar and like what a formula already does. Recipes can be used to do many of the same things, but they have a much wider range of possibilities. This article shows how to use recipes for modeling.

To use code in this article, you will need to install the following packages: nycflights13, skimr, and tidymodels.

pacman::p\_load(tidymodels, # recipes package  
 nycflights13, # flight data  
 skimr) # for variables summaries

## The New York City Fligt Data

Let’s use the nycflights13 data to predict whether a plane arrives more than 30 minutes late. This data set contains information on 325,819 flights departing near New York City in 2013. Let’s start by loading the data and making a few changes to the variables:

set.seed(123)  
  
flight\_data <-   
 flights %>%   
 mutate(  
 # Convert the arrivial delay to a factor  
 arr\_delay = ifelse(arr\_delay >= 30, "late", "on\_time"),  
 arr\_delay = factor(arr\_delay),  
 # we will use the date (not date-time) in the recipes below  
 date = as.Date(time\_hour)  
 ) %>%   
 # Include the weather data  
 inner\_join(weather, by = c("origin", "time\_hour")) %>%   
 # ONly rtain the specific columns we will use   
 select(dep\_time, flight, origin, dest, arr\_time, distance,  
 carrier, date, arr\_delay, time\_hour) %>%   
 # Exclude missing data  
 na.omit() %>%   
 # For creating models, it is better to have qualitative columns  
 # Encoded as factors (instead of character strings)  
 mutate\_if(is.character, as.factor)

We can see that about 16% of the flights in this data set arrived more than 30 minutes late.

flight\_data %>%   
 skimr::skim()

Data summary

|  |  |
| --- | --- |
| Name | Piped data |
| Number of rows | 325819 |
| Number of columns | 10 |
| \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ |  |
| Column type frequency: |  |
| Date | 1 |
| factor | 4 |
| numeric | 4 |
| POSIXct | 1 |
| \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ |  |
| Group variables | None |

**Variable type: Date**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| skim\_variable | n\_missing | complete\_rate | min | max | median | n\_unique |
| date | 0 | 1 | 2013-01-01 | 2013-12-30 | 2013-07-03 | 364 |

**Variable type: factor**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| skim\_variable | n\_missing | complete\_rate | ordered | n\_unique | top\_counts |
| origin | 0 | 1 | FALSE | 3 | EWR: 116504, JFK: 108539, LGA: 100776 |
| dest | 0 | 1 | FALSE | 104 | ATL: 16771, ORD: 16507, LAX: 15942, BOS: 14948 |
| carrier | 0 | 1 | FALSE | 16 | UA: 57489, B6: 53715, EV: 50868, DL: 47465 |
| arr\_delay | 0 | 1 | FALSE | 2 | on\_: 273279, lat: 52540 |

**Variable type: numeric**

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| skim\_variable | n\_missing | complete\_rate | mean | sd | p0 | p25 | p50 | p75 | p100 | hist |
| dep\_time | 0 | 1 | 1348.15 | 487.92 | 1 | 907 | 1400 | 1743 | 2400 | ▁▇▆▇▃ |
| flight | 0 | 1 | 1943.54 | 1621.73 | 1 | 544 | 1471 | 3416 | 8500 | ▇▃▃▁▁ |
| arr\_time | 0 | 1 | 1501.81 | 532.18 | 1 | 1104 | 1535 | 1940 | 2400 | ▁▃▇▇▇ |
| distance | 0 | 1 | 1048.18 | 735.86 | 80 | 509 | 888 | 1389 | 4983 | ▇▃▂▁▁ |

**Variable type: POSIXct**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| skim\_variable | n\_missing | complete\_rate | min | max | median | n\_unique |
| time\_hour | 0 | 1 | 2013-01-01 05:00:00 | 2013-12-30 18:00:00 | 2013-07-03 15:00:00 | 6885 |

flight\_data %>%   
 count(arr\_delay) %>%   
 mutate(prop = n/sum(n))  
## # A tibble: 2 x 3  
## arr\_delay n prop  
## <fct> <int> <dbl>  
## 1 late 52540 0.161  
## 2 on\_time 273279 0.839

Before we start building op our recipe, let’s take a quick look at a few specific variables that will be important for both preprocessing and modeling.

First, notice that the variable we created called arr\_delay is a factor variable; it is important that our outcome variable for training a logistic model is a factor.

glimpse(flight\_data)  
## Rows: 325,819  
## Columns: 10  
## $ dep\_time <int> 517, 533, 542, 544, 554, 554, 555, 557, 557, 558, 558, 558,…  
## $ flight <int> 1545, 1714, 1141, 725, 461, 1696, 507, 5708, 79, 301, 49, 7…  
## $ origin <fct> EWR, LGA, JFK, JFK, LGA, EWR, EWR, LGA, JFK, LGA, JFK, JFK,…  
## $ dest <fct> IAH, IAH, MIA, BQN, ATL, ORD, FLL, IAD, MCO, ORD, PBI, TPA,…  
## $ arr\_time <int> 830, 850, 923, 1004, 812, 740, 913, 709, 838, 753, 849, 853…  
## $ distance <dbl> 1400, 1416, 1089, 1576, 762, 719, 1065, 229, 944, 733, 1028…  
## $ carrier <fct> UA, UA, AA, B6, DL, UA, B6, EV, B6, AA, B6, B6, UA, UA, AA,…  
## $ date <date> 2013-01-01, 2013-01-01, 2013-01-01, 2013-01-01, 2013-01-01…  
## $ arr\_delay <fct> on\_time, on\_time, late, on\_time, on\_time, on\_time, on\_time,…  
## $ time\_hour <dttm> 2013-01-01 05:00:00, 2013-01-01 05:00:00, 2013-01-01 05:00…

Second, there are two variables that we don’t want to use as predictors in our model, but that we would like to retain as identification variables that can be used to troubleshoot poorly predicted data points. These are flight, a numeric value, and time\_hour, a date-time value.

Third, there are 104 flights destinations contained in dest and 16 distinc carriers.

flight\_data %>%   
 skimr::skim(dest, carrier)

Data summary

|  |  |
| --- | --- |
| Name | Piped data |
| Number of rows | 325819 |
| Number of columns | 10 |
| \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ |  |
| Column type frequency: |  |
| factor | 2 |
| \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ |  |
| Group variables | None |

**Variable type: factor**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| skim\_variable | n\_missing | complete\_rate | ordered | n\_unique | top\_counts |
| dest | 0 | 1 | FALSE | 104 | ATL: 16771, ORD: 16507, LAX: 15942, BOS: 14948 |
| carrier | 0 | 1 | FALSE | 16 | UA: 57489, B6: 53715, EV: 50868, DL: 47465 |

Because we’ll be using a simple logistic regression model, the variables dest and carrier will be converted to [dummy variables](https://bookdown.org/max/FES/creating-dummy-variables-for-unordered-categories.html). However, some of these values do not occur very frequently and this could complicate our analysis. We’ll discuss specific steps later in this article that we can add to our recipe to address this issue before modeling.

## Data splitting

To get started, let’s split this single dataset into two: a training set and a testing set. We’ll keep most of the rows in the original dataset (subset chosen randomly) in the training set. The training data will be used to fit the model, and the testing set will be used to measure model performance.

To do this, we can use the rsample package to create an object that contains the information on how to split the data, and then two more rsample functions to create data frames for the training and testing sets:

# Fix the random numbers by setting the seed   
# This enables the analysis to be reproducible when random numbers are used   
set.seed(555)  
# Put 3/4 of the data into the training set   
data\_split <- initial\_split(flight\_data, prop = 3/4)  
  
# Create data frames for the two sets:  
train\_data <- training(data\_split)  
test\_data <- testing(data\_split)

## Create recipe and role

To get started, let’s create a recipe for a simple logistic regression model. Before training the model, we can use a recipe to create a few new predictors and conduct some preprocessing required by the model.

Let’s initiate a new recipe:

flights\_rec <-   
 recipe(arr\_delay ~ ., data = train\_data)

The recipe() function as we used it here has two arguments:

* A formula:Any variable on the left-hand side of the tilde (~) is considered the model outcome (here, arr\_delay). On the right-hand side of the tilde are the predictors. Variables may be listed by name, or you can use the dot (.) to indicate all other variables as predictors.
* The data: A recipe is associated with the data set used to create the model. This will typically be the training set, so data = train\_data here. Naming a data set doesn’t actually change the data itself; it is only used to catalog the names of the variables and their types, like factors, integers, dates, etc.

Now we can add roles to this recipe. We can use the update\_role() function to let recipes know that flight and time\_hour are variables with a custom role that we called “ID” (a role can have any character value). Whereas our formula included all variables in the training set other than arr\_delay as predictors, this tells the recipe to keep these two variables but not use them as either outcomes or predictors.

flights\_rec <-   
 recipe(arr\_delay ~., data = train\_data) %>%   
 update\_role(flight, time\_hour, new\_role = "ID")

This step of adding roles to a recipe is optional; the purpose of using it here is that those two variables can be retained in the data but not included in the model. This can be convenient when, after the model is fit, we want to investigate some poorly predicted value. These ID columns will be available and can be used to try to understand what went wrong.

To get the current set of variables and roles, use the summary() function:

summary(flights\_rec)  
## # A tibble: 10 x 4  
## variable type role source   
## <chr> <chr> <chr> <chr>   
## 1 dep\_time numeric predictor original  
## 2 flight numeric ID original  
## 3 origin nominal predictor original  
## 4 dest nominal predictor original  
## 5 arr\_time numeric predictor original  
## 6 distance numeric predictor original  
## 7 carrier nominal predictor original  
## 8 date date predictor original  
## 9 time\_hour date ID original  
## 10 arr\_delay nominal outcome original

## Create features

Now we can start adding steps onto our recipe using the pipe operator. Perhaps it is reasonable for the date of the flight to have an effect on the likelihood of a late arrival. A little bit of **feature engineering** might go a long way to improving our model. How should the date be encoded into the model? The date column has an R date object so including that column “as is” will mean that the model will convert it to a numeric format equal to the number of days after a reference date:

flight\_data %>%   
 distinct(date) %>%   
 mutate(numeric\_date = as.numeric(date))  
## # A tibble: 364 x 2  
## date numeric\_date  
## <date> <dbl>  
## 1 2013-01-01 15706  
## 2 2013-01-02 15707  
## 3 2013-01-03 15708  
## 4 2013-01-04 15709  
## 5 2013-01-05 15710  
## 6 2013-01-06 15711  
## 7 2013-01-07 15712  
## 8 2013-01-08 15713  
## 9 2013-01-09 15714  
## 10 2013-01-10 15715  
## # … with 354 more rows

It’s possible that the numeric date variable is a good option for modeling; perhaps the model would benefit from a linear trend between the log-odds of a late arrival and the numeric date variable. However, it might be better to add model terms derived from the date that have a better potential to be important to the model. For example, we could derive the following meaningful features from the single date variable:

* the day of the week
* the month, and
* whether or not he date corresponds to our recipe:

flight\_rec <-   
 recipe(arr\_delay ~., data = train\_data) %>%   
 update\_role(flight, time\_hour, new\_role = "ID") %>%  
 step\_date(date, features = c("dow", "month")) %>%   
 step\_holiday(date, holidays = timeDate::listHolidays("US")) %>%   
 step\_rm(date)

What do each of these steps do?

* With step\_date(), we created two new factor columns with the appropriate day of the week and the month.
* With step\_holiday(), we created a binary variable indicating whether the current date is a holiday or not. The argument value of timeDate::listHolidays("US) uses the timeDate package to list the 17 standard US holidays.
* With step\_rm(), we remove the original date variable since we no longer wait it in the model.

Next, we will turn our attention to the variable types of our predictors. Because we plan to train a logistic regression model, we know that predictors will ultimately need to be numeric, as opposed to factor variables. In other words, there may be a difference in how we store our data (in factors inside a data frame), and how the underlying equations require them (a purely numeric matrix).

For factors like dest and origin, [standard practice](https://bookdown.org/max/FES/creating-dummy-variables-for-unordered-categories.html) is to convert them into dummy or indicator variables to make them numeric. These are binary values for each level of the factor. For example, our origin variable has values of "EWR", "JFK", and "LGA". The standard dummy variable encoding, shown below, will create two numeric columns of the data that are 1 when the originating airport is "JFK" or "LGA" and zero otherwise, respectively.

But, unlike the standard model formula methods in R, a recipe **does not** automatically create these dummy variables for you; you’ll need to tell your recipe to add this step. This is for two reasons. First, many models do not require numeric predictors, so dummy variables may not always be preferred. Second, recipes can also be used for purposes outside of modeling, where non-dummy versions of the variables may work better. For example, you may want to make a table or a plot with a variable as a single factor. For those reasons, you need to explicitly tell recipes to create dummy variables using step\_dummy():

flights\_rec <-   
 recipe(arr\_delay ~., data = train\_data) %>%   
 update\_role(flight, time\_hour, new\_role = "ID") %>%  
 step\_date(date, features = c("dow", "month")) %>%   
 step\_holiday(date, holidays = timeDate::listHolidays("US")) %>%   
 step\_rm(date) %>%   
 step\_dummy(all\_nominal(), -all\_outcomes())

Here, we did something different than before: instead of applying a step to an individual variable, we used [selectors](https://tidymodels.github.io/recipes/reference/selections.html) to apply this recipe step to several variables at once.

* The first selector, all\_nominal(), selects all variables that are either factors or characters.
* The second selector, -all\_outcomes() removes any outcome variables from this recipe step.

With these two selectors together, our recipe step above translates to:

Create dummy variables for all of the factor or character columns unless they are outcomes.

At this stage in the recipe, this step selects the origin, dest and carrier variables. It also includes two new variables, date\_dow and date\_month, that were created by earlier step\_date().

More generally, the recipe selectors mean that you don’t always have to apply step to individual variables one at a time. Since a recipe knows the *variable type* and role of each column, they can also be selected (or dropped) using this information.

We need one final step to add to our recipe. Since carrier and dest have some infrequently occurring values, it is possible that dummy variables might be created for values that don’t exist in the training set. For example, there is one destination that is only in the test set:

test\_data %>%   
 distinct(dest) %>%   
 anti\_join(train\_data)  
## # A tibble: 1 x 1  
## dest   
## <fct>  
## 1 LEX

When the recipe is applied to the training set, a column is made for LEX but it will contain all zeros. This is a zero-variance predictor that has no information within the column.

While some R functions will not produce an error for such predictors, it usually causes warning and other issue. step\_nz() will remove columns from the data when the training set data has a single value, so it is added to the recipe after step\_dummy().

flights\_rec <-   
 recipe(arr\_delay ~ ., data = train\_data) %>%   
 update\_role(flight, time\_hour, new\_role = "ID") %>%   
 step\_date(date, features = c("dow", "month")) %>%   
 step\_holiday(date, holidays = timeDate::listHolidays("US")) %>%   
 step\_rm(date) %>%   
 step\_dummy(all\_nominal(), -all\_outcomes()) %>%   
 step\_zv(all\_predictors())

Now we have created a specification of what should be done with the data. How do we use the recipe we made?

## Fit a model with a recipe

Let’s use logistic regression to model the flight data. As we saw in Build a Model, we start by building a model specification using the parsnip package.

lr\_mod <-   
 logistic\_reg() %>%   
 set\_engine("glm")

We will want to use our recipe across several steps as we train and test our model. We will

1. **Process the recipe using the training set**: This involves any estimation or calculations based on the training set. For our recipe, the training set will be used to determine which predictors should be converted to dummy variables and which predictors will have zero-variance in the training set, and should be slated for removal.
2. **Apply the recipe to the training set**: We create the final predictor set on the training set.
3. **Apply the recipe to the test set**: We create the final predictor set on the test set. Nothing is recomputed and no information from the test set is used here; the dummy variable and zero-variance results from the training set are applied to the test set.

To simplify this process, we can use a model workflow, which pairs a model and recipe together. This is a straightforward approach because different recipes are often needed for different models, so when a model and recipe are bundled, it becomes easier to train and test workflows. We’ll use the [workflows package](https://tidymodels.github.io/workflows/) from tidymodels to bundle our parsnip model(lr\_mod) with our recipe(flights\_rec)

library(workflows)  
  
flights\_wflow <-   
 workflows::workflow() %>%   
 workflows::add\_model(lr\_mod) %>%   
 workflows::add\_recipe(flights\_rec)  
  
flights\_wflow  
## ══ Workflow ═══════════════════════════════════════════════════════════════════════════  
## Preprocessor: Recipe  
## Model: logistic\_reg()  
##   
## ── Preprocessor ───────────────────────────────────────────────────────────────────────  
## 5 Recipe Steps  
##   
## ● step\_date()  
## ● step\_holiday()  
## ● step\_rm()  
## ● step\_dummy()  
## ● step\_zv()  
##   
## ── Model ──────────────────────────────────────────────────────────────────────────────  
## Logistic Regression Model Specification (classification)  
##   
## Computational engine: glm

Now, there is a single function that can be used to prepare the recipe and train the model from the resulting predictors:

flights\_fit <-   
 flights\_wflow %>%   
 fit(data = train\_data)

This object has the finalized recipe and fitted model objects inside. You may want to extract the model or recipe objects from the workflow. To do this, you can use the helper functions pull\_workflow\_fit() and pull\_workflow\_recipe(). For example, here we pull the fitted model object then use the broom::tidy() function to get a tidy tibble of model coefficients:

flights\_fit %>%   
 pull\_workflow\_fit() %>%   
 tidy()  
## # A tibble: 157 x 5  
## term estimate std.error statistic p.value  
## <chr> <dbl> <dbl> <dbl> <dbl>  
## 1 (Intercept) 6.75 2.71 2.49 1.27e- 2  
## 2 dep\_time -0.00181 0.0000153 -119. 0.   
## 3 arr\_time 0.000399 0.0000119 33.5 2.16e-246  
## 4 distance -0.00128 0.00149 -0.862 3.89e- 1  
## 5 date\_USChristmasDay 1.25 0.172 7.24 4.57e- 13  
## 6 date\_USColumbusDay 0.626 0.158 3.96 7.65e- 5  
## 7 date\_USCPulaskisBirthday 0.963 0.139 6.92 4.56e- 12  
## 8 date\_USDecorationMemorialDay 0.249 0.108 2.29 2.18e- 2  
## 9 date\_USElectionDay 0.692 0.168 4.11 3.91e- 5  
## 10 date\_USGoodFriday 1.43 0.166 8.61 7.45e- 18  
## # … with 147 more rows

## Use a trained workflow to predict

Our goal was to predict whether a plane arrives more than 30 minutes late. We have just:

1. Built the model (lr\_mod),
2. Created a preprocessing recipe (flights\_rec),
3. Bundled the model and recipe (flights\_wflow), and
4. Trained our workflow using a single call to fit().

The next step is to use the trained workflow (flights\_fit) to predict with the unseen test data, which we will do with a single call to predict(). The predict() method applies the recipe to the new data, then passes them to the fitted model.

predict(flights\_fit, test\_data)  
## # A tibble: 81,454 x 1  
## .pred\_class  
## <fct>   
## 1 on\_time   
## 2 on\_time   
## 3 on\_time   
## 4 on\_time   
## 5 on\_time   
## 6 on\_time   
## 7 on\_time   
## 8 on\_time   
## 9 on\_time   
## 10 on\_time   
## # … with 81,444 more rows

Because our outcome variable here is a factor, the output from predict() returns the predicted class: late versus on\_time. But, let’s say we want the predicted class probabilities for each flight instead. To return those, we can specify type = "prob" when we use predict(). We’ll also bind the output with some variables from the test data and save them together:

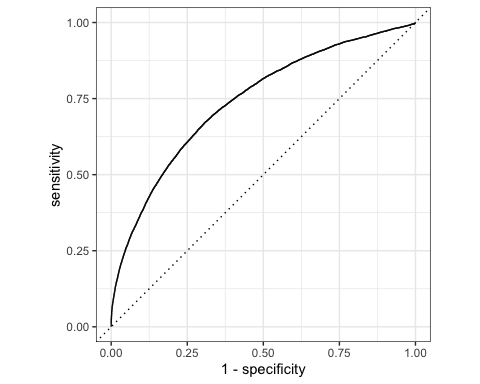
flights\_pred <-   
 predict(flights\_fit, test\_data, type = "prob") %>%   
 bind\_cols(test\_data %>% select(arr\_delay, time\_hour, flight))   
  
# The data look like:   
flights\_pred  
## # A tibble: 81,454 x 5  
## .pred\_late .pred\_on\_time arr\_delay time\_hour flight  
## <dbl> <dbl> <fct> <dttm> <int>  
## 1 0.0500 0.950 on\_time 2013-01-01 05:00:00 1714  
## 2 0.0569 0.943 on\_time 2013-01-01 06:00:00 79  
## 3 0.0570 0.943 on\_time 2013-01-01 06:00:00 301  
## 4 0.0725 0.927 on\_time 2013-01-01 06:00:00 49  
## 5 0.0434 0.957 on\_time 2013-01-01 06:00:00 1187  
## 6 0.0708 0.929 on\_time 2013-01-01 06:00:00 4401  
## 7 0.0545 0.945 on\_time 2013-01-01 06:00:00 1895  
## 8 0.0594 0.941 on\_time 2013-01-01 06:00:00 135  
## 9 0.0593 0.941 on\_time 2013-01-01 06:00:00 4646  
## 10 0.109 0.891 on\_time 2013-01-01 06:00:00 4144  
## # … with 81,444 more rows

Now that we have a tibble with our predicted class probabilities, how will we evaluate the performance of our workflow? We can see from these first few rows that our model predicted these 5 on time flights correctly because the values of .pred\_on\_time are p > .50. But we also know that we have 81,454 rows total to predict. We would like to calculate a metric that tells how well our model predicted late arrivals, compared to the true status of our outcome variable, arr\_delay.

Let’s use the area under the [ROC curve](https://bookdown.org/max/FES/measuring-performance.html#class-metrics) as our metric, computed using roc\_curve() and roc\_auc() from the [yardstick package](https://tidymodels.github.io/yardstick/).

To generate a ROC curve, we need the predicted class probabilities for late and on\_time, which we just calculated in the code chunk above. We can create the ROC curve with these values, using roc\_curve() and then piping to the autoplot() method:

flights\_pred %>%   
 roc\_curve(truth = arr\_delay, .pred\_late) %>%   
 autoplot()



Similarly, roc\_auc() estimates the area under the curve:

flights\_pred %>%   
 roc\_auc(truth = arr\_delay, .pred\_late)  
## # A tibble: 1 x 3  
## .metric .estimator .estimate  
## <chr> <chr> <dbl>  
## 1 roc\_auc binary 0.741

Not too bad! We leave it to the reader to test out this workflow without this recipe. You can use workflows::add\_formula(arr\_delay ~ .) instead of add\_recipe() (remember to remove the identification variables first!), and see whether our recipe improved our model’s ability to predict late arrivals.

# Evaluate your model with resamples

# Introduction

So far, we have built a model and preprocessed data with a recipe. We also introduced [workflows](https://www.tidymodels.org/start/recipes/#fit-workflow) as a way to bundle a [parsnip model](https://tidymodels.github.io/parsnip/) and [recipe](https://tidymodels.github.io/recipes/) together. Once we have a model trained, we need a way yo measure how well that model predicts new dat. This tutorial explains how to characterize model performance based on resampling statistics.

To use code in this article, you need to install the following packages.

library(tidymodels)  
library(modeldata)

## The cell image data

Let’s use data from Hill, LaPan, Li, and Haney (2007), available in the modeldata package, to predict cell image segmentation quality with resampling. To start, we load this data into R:

data(cells, package = "modeldata")  
cells  
## # A tibble: 2,019 x 58  
## case class angle\_ch\_1 area\_ch\_1 avg\_inten\_ch\_1 avg\_inten\_ch\_2 avg\_inten\_ch\_3  
## <fct> <fct> <dbl> <int> <dbl> <dbl> <dbl>  
## 1 Test PS 143. 185 15.7 4.95 9.55  
## 2 Train PS 134. 819 31.9 207. 69.9   
## 3 Train WS 107. 431 28.0 116. 63.9   
## 4 Train PS 69.2 298 19.5 102. 28.2   
## 5 Test PS 2.89 285 24.3 112. 20.5   
## 6 Test WS 40.7 172 326. 654. 129.   
## 7 Test WS 174. 177 260. 596. 124.   
## 8 Test PS 180. 251 18.3 5.73 17.2   
## 9 Test WS 18.9 495 16.1 89.5 13.7   
## 10 Test WS 153. 384 17.7 89.9 20.4   
## # … with 2,009 more rows, and 51 more variables: avg\_inten\_ch\_4 <dbl>,  
## # convex\_hull\_area\_ratio\_ch\_1 <dbl>, convex\_hull\_perim\_ratio\_ch\_1 <dbl>,  
## # diff\_inten\_density\_ch\_1 <dbl>, diff\_inten\_density\_ch\_3 <dbl>,  
## # diff\_inten\_density\_ch\_4 <dbl>, entropy\_inten\_ch\_1 <dbl>,  
## # entropy\_inten\_ch\_3 <dbl>, entropy\_inten\_ch\_4 <dbl>,  
## # eq\_circ\_diam\_ch\_1 <dbl>, eq\_ellipse\_lwr\_ch\_1 <dbl>,  
## # eq\_ellipse\_oblate\_vol\_ch\_1 <dbl>, eq\_ellipse\_prolate\_vol\_ch\_1 <dbl>,  
## # eq\_sphere\_area\_ch\_1 <dbl>, eq\_sphere\_vol\_ch\_1 <dbl>,  
## # fiber\_align\_2\_ch\_3 <dbl>, fiber\_align\_2\_ch\_4 <dbl>,  
## # fiber\_length\_ch\_1 <dbl>, fiber\_width\_ch\_1 <dbl>, inten\_cooc\_asm\_ch\_3 <dbl>,  
## # inten\_cooc\_asm\_ch\_4 <dbl>, inten\_cooc\_contrast\_ch\_3 <dbl>,  
## # inten\_cooc\_contrast\_ch\_4 <dbl>, inten\_cooc\_entropy\_ch\_3 <dbl>,  
## # inten\_cooc\_entropy\_ch\_4 <dbl>, inten\_cooc\_max\_ch\_3 <dbl>,  
## # inten\_cooc\_max\_ch\_4 <dbl>, kurt\_inten\_ch\_1 <dbl>, kurt\_inten\_ch\_3 <dbl>,  
## # kurt\_inten\_ch\_4 <dbl>, length\_ch\_1 <dbl>, neighbor\_avg\_dist\_ch\_1 <dbl>,  
## # neighbor\_min\_dist\_ch\_1 <dbl>, neighbor\_var\_dist\_ch\_1 <dbl>,  
## # perim\_ch\_1 <dbl>, shape\_bfr\_ch\_1 <dbl>, shape\_lwr\_ch\_1 <dbl>,  
## # shape\_p\_2\_a\_ch\_1 <dbl>, skew\_inten\_ch\_1 <dbl>, skew\_inten\_ch\_3 <dbl>,  
## # skew\_inten\_ch\_4 <dbl>, spot\_fiber\_count\_ch\_3 <int>,  
## # spot\_fiber\_count\_ch\_4 <dbl>, total\_inten\_ch\_1 <int>,  
## # total\_inten\_ch\_2 <dbl>, total\_inten\_ch\_3 <int>, total\_inten\_ch\_4 <int>,  
## # var\_inten\_ch\_1 <dbl>, var\_inten\_ch\_3 <dbl>, var\_inten\_ch\_4 <dbl>,  
## # width\_ch\_1 <dbl>

We have data for 2019 cells, with 58 variables. The main outcome variable of interest for us here is called class, which you can see is a factor. But before we jump into predicting the class variable, we need to understand it better. Below is a brief primer on cell image segmentation.

## Predicting image segmentation quality

Some biologists conduct experiments on cells. In drug discovery, a particular type of cell can be treated with either a drug or control and then observed to see what the effect is (if any). A common approach for this kind of measurement is cell imaging. Different parts of the cells can be colored so that the locations of a cell can be determined.

For example, in top panel of this image of five cells, the green color is meant to define the boundary of the cell (coloring something called the cytoskeleton) while the blue color defines the nucleus of the cell.

Using these colors, the cells in an image can be segmented so that we know which pixels belong to which cell. If this is done well, the cell can be measured in different ways that are important to the biology. Sometimes the shape of the cell matters and different mathematical tools are used to summarize characteristics like the size or “oblongness” of the cell.

The bottom panel shows some segmentation results. Cells 1 and 5 are fairly well segmented. However, cells 2 to 4 are bunched up together because the segmentation was not very good. The consequence of bad segmentation is data contamination; when the biologist analyzes the shape or size of these cells, the data are inaccurate and could lead to the wrong conclusion.

A cell-based experiment might involve millions of cells so it is unfeasible to visually assess them all. Instead, a subsample can be created and these cells can be manually labeled by experts as either poorly segmented (PS) or well-segmented (WS). If we can predict these labels accurately, the larger data set can be improved by filtering out the cells most likely to be poorly segmented.

## Back to the cells data

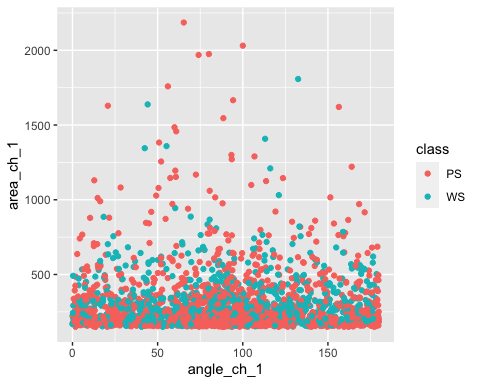
The cells data has class labels for 2019 cells - each is labeled as either poorly sgmented (PS) or well-segmented (WS). Each also has a total of 56 predictors based on automated image analysis measurements.

For example, avg\_inten\_ch\_1 is the mean intensity of the data contained in the nucleus, area\_ch\_1 is the total size of the cell, and so on (some predictors are fairly arcane in nature).

cells  
## # A tibble: 2,019 x 58  
## case class angle\_ch\_1 area\_ch\_1 avg\_inten\_ch\_1 avg\_inten\_ch\_2 avg\_inten\_ch\_3  
## <fct> <fct> <dbl> <int> <dbl> <dbl> <dbl>  
## 1 Test PS 143. 185 15.7 4.95 9.55  
## 2 Train PS 134. 819 31.9 207. 69.9   
## 3 Train WS 107. 431 28.0 116. 63.9   
## 4 Train PS 69.2 298 19.5 102. 28.2   
## 5 Test PS 2.89 285 24.3 112. 20.5   
## 6 Test WS 40.7 172 326. 654. 129.   
## 7 Test WS 174. 177 260. 596. 124.   
## 8 Test PS 180. 251 18.3 5.73 17.2   
## 9 Test WS 18.9 495 16.1 89.5 13.7   
## 10 Test WS 153. 384 17.7 89.9 20.4   
## # … with 2,009 more rows, and 51 more variables: avg\_inten\_ch\_4 <dbl>,  
## # convex\_hull\_area\_ratio\_ch\_1 <dbl>, convex\_hull\_perim\_ratio\_ch\_1 <dbl>,  
## # diff\_inten\_density\_ch\_1 <dbl>, diff\_inten\_density\_ch\_3 <dbl>,  
## # diff\_inten\_density\_ch\_4 <dbl>, entropy\_inten\_ch\_1 <dbl>,  
## # entropy\_inten\_ch\_3 <dbl>, entropy\_inten\_ch\_4 <dbl>,  
## # eq\_circ\_diam\_ch\_1 <dbl>, eq\_ellipse\_lwr\_ch\_1 <dbl>,  
## # eq\_ellipse\_oblate\_vol\_ch\_1 <dbl>, eq\_ellipse\_prolate\_vol\_ch\_1 <dbl>,  
## # eq\_sphere\_area\_ch\_1 <dbl>, eq\_sphere\_vol\_ch\_1 <dbl>,  
## # fiber\_align\_2\_ch\_3 <dbl>, fiber\_align\_2\_ch\_4 <dbl>,  
## # fiber\_length\_ch\_1 <dbl>, fiber\_width\_ch\_1 <dbl>, inten\_cooc\_asm\_ch\_3 <dbl>,  
## # inten\_cooc\_asm\_ch\_4 <dbl>, inten\_cooc\_contrast\_ch\_3 <dbl>,  
## # inten\_cooc\_contrast\_ch\_4 <dbl>, inten\_cooc\_entropy\_ch\_3 <dbl>,  
## # inten\_cooc\_entropy\_ch\_4 <dbl>, inten\_cooc\_max\_ch\_3 <dbl>,  
## # inten\_cooc\_max\_ch\_4 <dbl>, kurt\_inten\_ch\_1 <dbl>, kurt\_inten\_ch\_3 <dbl>,  
## # kurt\_inten\_ch\_4 <dbl>, length\_ch\_1 <dbl>, neighbor\_avg\_dist\_ch\_1 <dbl>,  
## # neighbor\_min\_dist\_ch\_1 <dbl>, neighbor\_var\_dist\_ch\_1 <dbl>,  
## # perim\_ch\_1 <dbl>, shape\_bfr\_ch\_1 <dbl>, shape\_lwr\_ch\_1 <dbl>,  
## # shape\_p\_2\_a\_ch\_1 <dbl>, skew\_inten\_ch\_1 <dbl>, skew\_inten\_ch\_3 <dbl>,  
## # skew\_inten\_ch\_4 <dbl>, spot\_fiber\_count\_ch\_3 <int>,  
## # spot\_fiber\_count\_ch\_4 <dbl>, total\_inten\_ch\_1 <int>,  
## # total\_inten\_ch\_2 <dbl>, total\_inten\_ch\_3 <int>, total\_inten\_ch\_4 <int>,  
## # var\_inten\_ch\_1 <dbl>, var\_inten\_ch\_3 <dbl>, var\_inten\_ch\_4 <dbl>,  
## # width\_ch\_1 <dbl>

The rates of the claases are somewhat imbalanced; there are more poorly segmented cells than well-segmented cells:

cells %>%   
 count(class) %>%   
 mutate(prop = n/sum(n))  
## # A tibble: 2 x 3  
## class n prop  
## <fct> <int> <dbl>  
## 1 PS 1300 0.644  
## 2 WS 719 0.356  
  
cells %>%   
 ggplot(aes(x = angle\_ch\_1, y = area\_ch\_1, col = class))+  
 geom\_point()



## Data splitting

In our previous article, we started by splitting our data. It is common when beginning a modeling project to separate the data set into two partitions:

* The *training set* is used to estimate parameters, compare models and feature engineering techniques, tune models, etc.
* The *test set* is held in reserve until the end of the project, at which point there should only be one or two models under serious consideration. It is used as an unbiased source for measuring final model performance.

There are different ways to these partitions of the data. The most common approach is to use a random sample. Suppose that one quarter of the data were reserved for the test set. Random sampling would randomly select 25% for the test set and use the remainder for the training set. We can use the rsample package for this purpose.

Since random sampling uses random numbers, it is important to set the random number seed. This ensures that the random numbers can be reproduced at a later time (if needed).

The function rsample::initial\_split() takes the original data and saves the information on how to make the partitions. In the original analysis, the authors made their own training/test set and that information is contained in the column case. To demonstrate how to make a split, we’ll remove this column before we make our own split:

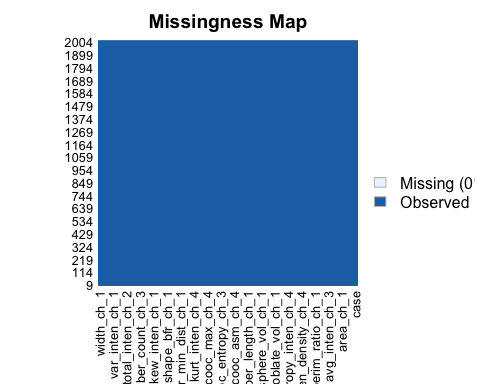
set.seed(123)  
cell\_split <- initial\_split(  
 cells %>% select(-case),  
 strata = class  
)

Here we used the strata argument, which conducts a stratified split. This ensures that, despite the imbalance we noticed in our class variable, our training and test data sets will keep roughly the same proportions of poorly and well-segmented cells as in the original data. After the initial\_split, the training() and testing() functions return the actual data sets.

cell\_train <- training(cell\_split)  
cell\_test <- testing(cell\_split)  
  
nrow(cell\_train)  
## [1] 1515  
nrow(cell\_train) / nrow(cells)  
## [1] 0.7503715  
  
# training set proportions by class  
cell\_train %>%   
 count(class) %>%   
 mutate(prop = n/sum(n))  
## # A tibble: 2 x 3  
## class n prop  
## <fct> <int> <dbl>  
## 1 PS 975 0.644  
## 2 WS 540 0.356  
  
# test set proportions by class  
cell\_test %>%   
 count(class) %>%   
 mutate(prop = n/sum(n))  
## # A tibble: 2 x 3  
## class n prop  
## <fct> <int> <dbl>  
## 1 PS 325 0.645  
## 2 WS 179 0.355

The majority of the modeling work is then conducted on the training set data. To note, there is no missing value on this data, thus no missing data both on Training and Test sets.

cells %>%   
 Amelia::missmap()



dim(cells)  
## [1] 2019 58

## Modeling

[Random forest models](https://bradleyboehmke.github.io/HOML/random-forest.html) are ensembles of [decision trees](https://bradleyboehmke.github.io/HOML/DT.html). A large number of decision tree models are created for the ensemble based on slightly different versions of the training set. When creating the individual decision trees, the fitting process encourages them to be as diverse as possible. The collection of trees are combined into the random forest model and, when a new sample is predicted, the votes from each tree are used to calculate the final predicted value for the new sample. For categorical outcome variables like class in our cells data example, the majority vote across all the trees in the random forest determines the predicted class for the new sample.

One of the benefits of a random forest model is that it is very low maintenance; it requires very little preprocessing of the data and the default parameters tend to give reasonable results. For that reason, we won’t create a recipe for the cells data.

At the same time, the number of trees in the ensemble should be large (in the thousands) and this makes the model moderately expensive to compute.

To fit a random forest model on the training set, let’s use the parsnip package with the ranger engine. We first define the model that we want to create:

# random forest model  
rf\_mod <-   
 rand\_forest(trees = 1000) %>%   
 set\_engine("ranger") %>%   
 set\_mode("classification")  
  
gbm\_mod <-   
 parsnip::boost\_tree() %>%   
 set\_engine("xgboost") %>%   
 set\_mode("classification")

Starting with this parsnip model object, the fit() function can be used with a model formula. Since random forest models use random numbers, we again set the seed prior to computing:

set.seed(234)  
rf\_fit <-   
 rf\_mod %>%   
 fit(class ~., data = cell\_train)  
  
rf\_fit  
## parsnip model object  
##   
## Fit time: 3.2s   
## Ranger result  
##   
## Call:  
## ranger::ranger(formula = formula, data = data, num.trees = ~1000, num.threads = 1, verbose = FALSE, seed = sample.int(10^5, 1), probability = TRUE)   
##   
## Type: Probability estimation   
## Number of trees: 1000   
## Sample size: 1515   
## Number of independent variables: 56   
## Mtry: 7   
## Target node size: 10   
## Variable importance mode: none   
## Splitrule: gini   
## OOB prediction error (Brier s.): 0.1218873

This new rf\_fit object is our fitted model, trained on our training data set. gbm\_fit is created apart from the original article.

gbm\_fit <-   
 gbm\_mod %>%   
 fit(class ~., data = cell\_train)  
  
gbm\_fit  
## parsnip model object  
##   
## Fit time: 192ms   
## ##### xgb.Booster  
## raw: 37.4 Kb   
## call:  
## xgboost::xgb.train(params = list(eta = 0.3, max\_depth = 6, gamma = 0,   
## colsample\_bytree = 1, min\_child\_weight = 1, subsample = 1),   
## data = x, nrounds = 15, verbose = 0, objective = "binary:logistic",   
## nthread = 1)  
## params (as set within xgb.train):  
## eta = "0.3", max\_depth = "6", gamma = "0", colsample\_bytree = "1", min\_child\_weight = "1", subsample = "1", objective = "binary:logistic", nthread = "1", silent = "1"  
## xgb.attributes:  
## niter  
## # of features: 56   
## niter: 15  
## nfeatures : 56

## Estimating performance

During a modeling project, we might create a variety of different models. To choose between them, e need to consider how well these models do, as measured by some performance statistics. In our example in this article, some options we could use are:

* the area under the Receiver Operating Characteristic (ROC) curve, and
* the overall classification accuracy

The ROC curve uses the class probability estimates to give us a sense of performance across the entire set of potential probability cutoffs. Overall accuracy uses the hard class predictions to measure performance. The hard class predictions tell us whether our model predicted PS or WS for each cell. But, behind those predictions, the model is actually estimating a probability. A simple 50% probability cutoff is used to categorize a cell as poorly segmented.

The [yardstick package](https://tidymodels.github.io/yardstick/) has functions for computing both of these measures called roc\_auc() and accuracy().

At first glance, it might seem like a good idea to use the training set data to compute these statistics. (This is actually a very bad idea.) Let’s see what happens if we try this. To evaluate performance based on the training set, we call the predict() method to get both types of predictions (i.e. probabilities and hard class predictions).

rf\_training\_pred <-   
 predict(rf\_fit, cell\_train) %>%   
 bind\_cols(predict(rf\_fit, cell\_train, type = "prob")) %>%   
 # Add the true outcome data back in  
 bind\_cols(cell\_train %>%   
 select(class))

Using the yardstick functions, this model has spectacular results, so spectacular that you might be starting to get suspicious:

rf\_training\_pred %>% # training set predictions  
 roc\_auc(truth = class, .pred\_PS)  
## # A tibble: 1 x 3  
## .metric .estimator .estimate  
## <chr> <chr> <dbl>  
## 1 roc\_auc binary 1.00  
#> # A tibble: 1 x 3  
#> .metric .estimator .estimate  
#> <chr> <chr> <dbl>  
#> 1 roc\_auc binary 1.00  
rf\_training\_pred %>% # training set predictions  
 accuracy(truth = class, .pred\_class)  
## # A tibble: 1 x 3  
## .metric .estimator .estimate  
## <chr> <chr> <dbl>  
## 1 accuracy binary 0.993  
#> # A tibble: 1 x 3  
#> .metric .estimator .estimate  
#> <chr> <chr> <dbl>  
#> 1 accuracy binary 0.993

Now that we have this model with exceptional performance, we proceed to the test set. Unfortunately, we discover that, although our results aren’t bad, they are certainly worse than what we initially thought based on predicting the training set:

rf\_testing\_pred <-   
 predict(rf\_fit, cell\_test) %>%   
 bind\_cols(predict(rf\_fit, cell\_test, type = "prob")) %>%   
 bind\_cols(cell\_test %>%   
 select(class))

# Test set predictions  
rf\_testing\_pred %>%   
 roc\_auc(truth = class, .pred\_PS)  
## # A tibble: 1 x 3  
## .metric .estimator .estimate  
## <chr> <chr> <dbl>  
## 1 roc\_auc binary 0.909  
  
# Test set predictions  
rf\_testing\_pred %>%   
 accuracy(truth = class, .pred\_class)  
## # A tibble: 1 x 3  
## .metric .estimator .estimate  
## <chr> <chr> <dbl>  
## 1 accuracy binary 0.837

# GBM implementation   
# please refer - https://tidymodels.github.io/parsnip/reference/boost\_tree.html  
  
gbm\_training\_pred <-   
 predict(gbm\_fit, cell\_train) %>%   
 bind\_cols(predict(rf\_fit, cell\_train, type = "prob")) %>%   
 # Add the true outcome data back in  
 bind\_cols(cell\_train %>%   
 select(class))  
  
gbm\_training\_pred %>% # training set predictions  
 roc\_auc(truth = class, .pred\_PS)  
## # A tibble: 1 x 3  
## .metric .estimator .estimate  
## <chr> <chr> <dbl>  
## 1 roc\_auc binary 1.00  
#> # A tibble: 1 x 3  
#> .metric .estimator .estimate  
#> <chr> <chr> <dbl>  
#> 1 roc\_auc binary 1.00  
gbm\_training\_pred %>% # training set predictions  
 accuracy(truth = class, .pred\_class)  
## # A tibble: 1 x 3  
## .metric .estimator .estimate  
## <chr> <chr> <dbl>  
## 1 accuracy binary 0.983  
  
gbm\_training\_pred  
## # A tibble: 1,515 x 4  
## .pred\_class .pred\_PS .pred\_WS class  
## <fct> <dbl> <dbl> <fct>  
## 1 PS 0.923 0.0773 PS   
## 2 PS 0.712 0.288 PS   
## 3 WS 0.206 0.794 WS   
## 4 PS 0.910 0.0901 PS   
## 5 PS 0.981 0.0191 PS   
## 6 WS 0.0557 0.944 WS   
## 7 WS 0.104 0.896 WS   
## 8 PS 0.947 0.0525 PS   
## 9 WS 0.266 0.734 WS   
## 10 WS 0.386 0.614 WS   
## # … with 1,505 more rows

### What happended here?

There are several reasons why training set statisics like the ones shown in this section can be unrealistically optimistic:

* Models like random forests, neural networks or and other black-box methods can essentially memorize the training set. Re-predicting that same set should always result in nearly perfect results.
* The training set does not have the capacity to be a good arbiter of performance. It is not an independent piece of information; predicting the training set can only reflect what the model already knows.

To understand that second point better, think about an analogy from teaching. Suppose you give a class a test, then give them answers, then provide the same test. The student scores on the second test do not accurately reflect what they know about the subject; these scores would probably be higher than their results on the first set.

## Resampling to the rescue

Resampling methods, such as cross-validation and the bootstrap, are emprical simulation systems. They create a series of data sets similar to the training/testing split discussed previously; a subset of the data are used for creating the model and a different subset is used to measure performance. Resampling is always used with the training set. This schematic from [Kuhn and Johnson (2019)](https://bookdown.org/max/FES/resampling.html) illustrates data usage for resampling methods:

knitr::include\_graphics("resampling.svg")



In the first level of this diagram, you see what happens when you use rsample::initial\_split(), which splits the original data into training and test sets. THen, the training set is chosen for resampling, and the test set is held out.

Let’s use 10-fold cross-validation (CV) in this example. This method randomly allocates the 1515 cells in the training set to 10 groups of roughly equal size, called “folds”. For the first iteration of resampling, the first fold of about 151 cells are held out for the purpose of measuring performance. This is similar to a test set but, to avoid confusion, we call these data the assessment set in the tidymodels framework.

The other 90% of the data (about 1363 cells) are used to fit the model. Again, this sounds similar to a training set, so in tidymodels we call this data the analysis set. This model, trained on the analysis set, is applied to the assessment set to generate predictions, and performance statistics are computed based on those predictions.

In this example, 10-fold CV moves iteratively through the folds and leaves a different 10% out each time for model assessment. At the end of this process, there are 10 sets of performance statistics that were created on 10 data sets that were not used in the modeling process. For the cell example, this means 10 accuracies and 10 areas under the ROC curve. While 10 models were created, these are not used further; we do not keep the models themselves trained on these folds because their only purpose is calculating performance metrics.

The final resampling estimates for the model are the **averages** of the performance statistics replicates. For example, suppose for our data the results were:

From these resampling statistics, the final estimate of performance for this random forest model would be 0.903 for the area under the ROC curve and 0.833 for accuracy.

These resampling statistics are an effective method for measuring model performance without predicting the training set directly as a whole.

## Fit a model with resampling

To generate these results, the first step is to create a resampling object using resample. There are several resampling methods implemented in rsample; cross-validation folds can be created using vfold\_cv():

set.seed(345)  
folds <- vfold\_cv(cell\_train, v = 10)  
folds  
## # 10-fold cross-validation   
## # A tibble: 10 x 2  
## splits id   
## <named list> <chr>   
## 1 <split [1.4K/152]> Fold01  
## 2 <split [1.4K/152]> Fold02  
## 3 <split [1.4K/152]> Fold03  
## 4 <split [1.4K/152]> Fold04  
## 5 <split [1.4K/152]> Fold05  
## 6 <split [1.4K/151]> Fold06  
## 7 <split [1.4K/151]> Fold07  
## 8 <split [1.4K/151]> Fold08  
## 9 <split [1.4K/151]> Fold09  
## 10 <split [1.4K/151]> Fold10

The list column for splits contains the information on which rows belong in the analysis and assessment sets. There are functions that can be used to extract the individual resampled data called analysis() and assessment().

However, the tune package contains high-level functions that can do the required computations to resample a model for the purpose of measuring performance. You have several options for building an object for resampling:

* Resample a model specification preprocessed with a formula or recipe, or
* Resample a [workflow()](https://tidymodels.github.io/workflows/) that bundles together a model specification and formula/recipe

For this example, let’s use a workflow() that bundles together the random forest model and a formula, since we are not using a recipe. Whichever of these options you use, the syntax to fit\_resamples() is very similar to fit():

rf\_wf <-   
 workflow() %>%  
 add\_model(rf\_mod) %>%  
 add\_formula(class ~ .)  
  
set.seed(456)  
rf\_fit\_rs <-   
 rf\_wf %>%   
 tune::fit\_resamples(folds)  
  
gbm\_wf <-   
 workflow() %>%   
 add\_model(gbm\_mod) %>%   
 add\_formula(class ~ .)  
  
gbm\_fit\_rs <-   
 gbm\_wf %>%   
 tune::fit\_resamples(folds)

rf\_fit\_rs  
## # 10-fold cross-validation   
## # A tibble: 10 x 4  
## splits id .metrics .notes   
## <list> <chr> <list> <list>   
## 1 <split [1.4K/152]> Fold01 <tibble [2 × 3]> <tibble [0 × 1]>  
## 2 <split [1.4K/152]> Fold02 <tibble [2 × 3]> <tibble [0 × 1]>  
## 3 <split [1.4K/152]> Fold03 <tibble [2 × 3]> <tibble [0 × 1]>  
## 4 <split [1.4K/152]> Fold04 <tibble [2 × 3]> <tibble [0 × 1]>  
## 5 <split [1.4K/152]> Fold05 <tibble [2 × 3]> <tibble [0 × 1]>  
## 6 <split [1.4K/151]> Fold06 <tibble [2 × 3]> <tibble [0 × 1]>  
## 7 <split [1.4K/151]> Fold07 <tibble [2 × 3]> <tibble [0 × 1]>  
## 8 <split [1.4K/151]> Fold08 <tibble [2 × 3]> <tibble [0 × 1]>  
## 9 <split [1.4K/151]> Fold09 <tibble [2 × 3]> <tibble [0 × 1]>  
## 10 <split [1.4K/151]> Fold10 <tibble [2 × 3]> <tibble [0 × 1]>

The results are similar to the folds results with some extra columns. The column .metrics contains the performance statistics created from the 10 assessment sets. These can be manually unnested but the tune package contains a number of simple functions that can extract these data:

# df <- tibble(x = c(1, 1, 1, 2, 2, 3), y = 1:6, z = 6:1)  
# df  
# df %>%   
# nest(data = c(y, z)) %>%   
# unnest(data)  
# df %>% chop(c(y, z)) %>%   
# unnest(y)  
  
rf\_fit\_rs %>%   
 unnest(.metrics)  
## # A tibble: 20 x 6  
## splits id .metric .estimator .estimate .notes   
## <list> <chr> <chr> <chr> <dbl> <list>   
## 1 <split [1.4K/152]> Fold01 accuracy binary 0.783 <tibble [0 × 1]>  
## 2 <split [1.4K/152]> Fold01 roc\_auc binary 0.842 <tibble [0 × 1]>  
## 3 <split [1.4K/152]> Fold02 accuracy binary 0.809 <tibble [0 × 1]>  
## 4 <split [1.4K/152]> Fold02 roc\_auc binary 0.894 <tibble [0 × 1]>  
## 5 <split [1.4K/152]> Fold03 accuracy binary 0.849 <tibble [0 × 1]>  
## 6 <split [1.4K/152]> Fold03 roc\_auc binary 0.917 <tibble [0 × 1]>  
## 7 <split [1.4K/152]> Fold04 accuracy binary 0.836 <tibble [0 × 1]>  
## 8 <split [1.4K/152]> Fold04 roc\_auc binary 0.894 <tibble [0 × 1]>  
## 9 <split [1.4K/152]> Fold05 accuracy binary 0.868 <tibble [0 × 1]>  
## 10 <split [1.4K/152]> Fold05 roc\_auc binary 0.906 <tibble [0 × 1]>  
## 11 <split [1.4K/151]> Fold06 accuracy binary 0.841 <tibble [0 × 1]>  
## 12 <split [1.4K/151]> Fold06 roc\_auc binary 0.914 <tibble [0 × 1]>  
## 13 <split [1.4K/151]> Fold07 accuracy binary 0.881 <tibble [0 × 1]>  
## 14 <split [1.4K/151]> Fold07 roc\_auc binary 0.937 <tibble [0 × 1]>  
## 15 <split [1.4K/151]> Fold08 accuracy binary 0.781 <tibble [0 × 1]>  
## 16 <split [1.4K/151]> Fold08 roc\_auc binary 0.889 <tibble [0 × 1]>  
## 17 <split [1.4K/151]> Fold09 accuracy binary 0.815 <tibble [0 × 1]>  
## 18 <split [1.4K/151]> Fold09 roc\_auc binary 0.908 <tibble [0 × 1]>  
## 19 <split [1.4K/151]> Fold10 accuracy binary 0.868 <tibble [0 × 1]>  
## 20 <split [1.4K/151]> Fold10 roc\_auc binary 0.931 <tibble [0 × 1]>  
  
collect\_metrics(rf\_fit\_rs)  
## # A tibble: 2 x 5  
## .metric .estimator mean n std\_err  
## <chr> <chr> <dbl> <int> <dbl>  
## 1 accuracy binary 0.833 10 0.0111   
## 2 roc\_auc binary 0.903 10 0.00842  
collect\_metrics(gbm\_fit\_rs)  
## # A tibble: 2 x 5  
## .metric .estimator mean n std\_err  
## <chr> <chr> <dbl> <int> <dbl>  
## 1 accuracy binary 0.806 10 0.00880  
## 2 roc\_auc binary 0.889 10 0.00783

Think about these values we now have for accuracy and AUC. These performance metrics are now realistic (i.e. lower) than our ill-advised first attempt at computing performance metrics in the section above. If we wanted to try different model types for this data set, we could more confidently compare performance metrics computed using resampling to choose between models. Also, remember that at the end of our project, we return to our test set to estimate final model performance. We have looked at this once already before we started using resampling, but let’s remind ourselves of the results:

rf\_testing\_pred %>% # test set predictions  
 roc\_auc(truth = class, .pred\_PS)  
## # A tibble: 1 x 3  
## .metric .estimator .estimate  
## <chr> <chr> <dbl>  
## 1 roc\_auc binary 0.909  
  
rf\_testing\_pred %>% # test set prediction  
 accuracy(truth = class, .pred\_class)  
## # A tibble: 1 x 3  
## .metric .estimator .estimate  
## <chr> <chr> <dbl>  
## 1 accuracy binary 0.837

The performance metrics from the test set are much closer to the performance metrics computed using resampling than our first (“bad idea”) attempt. Resampling allows us to simulate how well our model will perform on new data, and the test set acts as the final, unbiased check for our model’s performance.

# Tune model parameters

## Introduction

Some model parameters cannot be learned directly from a data set during model training; these kinds of parameters are called **hyperparameters**. Some examples of hyperparameters include the number of predictors hat are sampled at splits in a tree-based model (we call this mtry in tidymodels) or the learning rate in a boosted tree model (we call this learn\_rate). Instead of learning these kinds of hyperparameters during model training, we can estimate the best values for these values by training many models on resampled data sets and exploring how well all these models perform. This process is called **tuning**.

To use code in this article, you will need to install the following packages: mdoeldata, rpart, tidymodels, and vip.

library(tidymodels) # for the tune package, along with the rest of tidymodels  
  
# Helper packages  
library(modeldata) # for the cells data  
library(vip) # for variable importance plots

## The cell image data, revisited

In our previous [evaluate your model with resampling article](https://www.tidymodels.org/start/resampling/), we introduced a dataset of images of cells that were labeled by experts as well-segmented (WS) or poorly segmented (PS). WE trained a random forest model to predict which images are segmented well vs poorly, so that a biologist could filter our poorly segmented cell images in their analysis. We used resamplig to estimate the performance of our model on this data.

data(cells, package = "modeldata")  
cells  
## # A tibble: 2,019 x 58  
## case class angle\_ch\_1 area\_ch\_1 avg\_inten\_ch\_1 avg\_inten\_ch\_2 avg\_inten\_ch\_3  
## <fct> <fct> <dbl> <int> <dbl> <dbl> <dbl>  
## 1 Test PS 143. 185 15.7 4.95 9.55  
## 2 Train PS 134. 819 31.9 207. 69.9   
## 3 Train WS 107. 431 28.0 116. 63.9   
## 4 Train PS 69.2 298 19.5 102. 28.2   
## 5 Test PS 2.89 285 24.3 112. 20.5   
## 6 Test WS 40.7 172 326. 654. 129.   
## 7 Test WS 174. 177 260. 596. 124.   
## 8 Test PS 180. 251 18.3 5.73 17.2   
## 9 Test WS 18.9 495 16.1 89.5 13.7   
## 10 Test WS 153. 384 17.7 89.9 20.4   
## # … with 2,009 more rows, and 51 more variables: avg\_inten\_ch\_4 <dbl>,  
## # convex\_hull\_area\_ratio\_ch\_1 <dbl>, convex\_hull\_perim\_ratio\_ch\_1 <dbl>,  
## # diff\_inten\_density\_ch\_1 <dbl>, diff\_inten\_density\_ch\_3 <dbl>,  
## # diff\_inten\_density\_ch\_4 <dbl>, entropy\_inten\_ch\_1 <dbl>,  
## # entropy\_inten\_ch\_3 <dbl>, entropy\_inten\_ch\_4 <dbl>,  
## # eq\_circ\_diam\_ch\_1 <dbl>, eq\_ellipse\_lwr\_ch\_1 <dbl>,  
## # eq\_ellipse\_oblate\_vol\_ch\_1 <dbl>, eq\_ellipse\_prolate\_vol\_ch\_1 <dbl>,  
## # eq\_sphere\_area\_ch\_1 <dbl>, eq\_sphere\_vol\_ch\_1 <dbl>,  
## # fiber\_align\_2\_ch\_3 <dbl>, fiber\_align\_2\_ch\_4 <dbl>,  
## # fiber\_length\_ch\_1 <dbl>, fiber\_width\_ch\_1 <dbl>, inten\_cooc\_asm\_ch\_3 <dbl>,  
## # inten\_cooc\_asm\_ch\_4 <dbl>, inten\_cooc\_contrast\_ch\_3 <dbl>,  
## # inten\_cooc\_contrast\_ch\_4 <dbl>, inten\_cooc\_entropy\_ch\_3 <dbl>,  
## # inten\_cooc\_entropy\_ch\_4 <dbl>, inten\_cooc\_max\_ch\_3 <dbl>,  
## # inten\_cooc\_max\_ch\_4 <dbl>, kurt\_inten\_ch\_1 <dbl>, kurt\_inten\_ch\_3 <dbl>,  
## # kurt\_inten\_ch\_4 <dbl>, length\_ch\_1 <dbl>, neighbor\_avg\_dist\_ch\_1 <dbl>,  
## # neighbor\_min\_dist\_ch\_1 <dbl>, neighbor\_var\_dist\_ch\_1 <dbl>,  
## # perim\_ch\_1 <dbl>, shape\_bfr\_ch\_1 <dbl>, shape\_lwr\_ch\_1 <dbl>,  
## # shape\_p\_2\_a\_ch\_1 <dbl>, skew\_inten\_ch\_1 <dbl>, skew\_inten\_ch\_3 <dbl>,  
## # skew\_inten\_ch\_4 <dbl>, spot\_fiber\_count\_ch\_3 <int>,  
## # spot\_fiber\_count\_ch\_4 <dbl>, total\_inten\_ch\_1 <int>,  
## # total\_inten\_ch\_2 <dbl>, total\_inten\_ch\_3 <int>, total\_inten\_ch\_4 <int>,  
## # var\_inten\_ch\_1 <dbl>, var\_inten\_ch\_3 <dbl>, var\_inten\_ch\_4 <dbl>,  
## # width\_ch\_1 <dbl>

## Predicting image segmentation, but better

Random forest models are a tree-based ensemble method, and typically perform well with default hyperparameters. However, the accuracy of some other tree-based models, such as boosted tree models or decision tree models, can be sensitive to the values of hyperparameters. In this article, we will train a decision tree model. There are several hyperparameters for **decision tree** models that can be tuned for better performance. Let’s explore:

* complexity parameter (which we call cost\_complexity in tidymodels) for the tree, and
* the maximum tree\_depth

Tuning these hyperparameters can improve model performance because decision tree models are prone to overfitting. This happens because single tree models tend to fit the training data too well — so well, in fact, that they over-learn patterns present in the training data that end up being detrimental when predicting new data.

Tuning the value of cost\_complexity helps by pruning back our tree. It adds a cost, or penalty, to error rates of more complex trees; a cost closer to zero decreases the number tree nodes pruned and is more likely to result in an overfit tree. However, a high cost increases the number of tree nodes pruned and can result in the opposite problem—an underfit tree. Tuning tree\_depth, on the other hand, helps by stopping our tree from growing after it reaches a certain depth. We want to tune these hyperparameters to find what those two values should be for our model to do the best job predicting image segmentation.

We will tune the model hyperparameters to avoid overfitting. Tuning the value of cost\_compexity helps by pruning back our tree. It adds a cost, or penalty, to error rates of more complex trees; a cost closer to zero decreases the number tree nodes pruned and is more likely to result in an overfit tree. However, a high cost increases the number of tree nodes pruned and can result in the opposite problem—an underfit tree. Tuning tree\_depth, on the other hand, helps by stopping our tree from growing after it reaches a certain depth. We want to tune these hyperparameters to find what those two values should be for our model to do the best job predicting image segmentation.

Before we start the tuning process, we split our data into training and testing sets, just like when we trained the model with one default set of hyperparameters. As before, we can use strata = class if we want our training and testing sets to be created using stratified sampling so that both have the same proportion of both kinds of segmentation.

set.seed(123)  
cell\_split <- initial\_split(cells %>%   
 select(-case),  
 strata = class)  
cell\_train <- training(cell\_split)  
cell\_test <- testing(cell\_split)

## Tuning hyperparameters

Let’s start with the parsnip package, using decision\_tree() model with the rpart engine. To tune the decision tree hyperparameters cost\_complexity and tree\_depth, we create a model specification that identifies whichi hyperparameters we plan to tune.

tune\_spec <-   
 decision\_tree(  
 cost\_complexity = tune(),  
 tree\_depth = tune()  
 ) %>%   
 set\_engine("rpart") %>%   
 set\_mode("classification")  
  
tune\_spec  
## Decision Tree Model Specification (classification)  
##   
## Main Arguments:  
## cost\_complexity = tune()  
## tree\_depth = tune()  
##   
## Computational engine: rpart  
  
tune\_gbm\_mod <-   
 parsnip::boost\_tree(  
 mtry = tune(),  
 trees = tune(),  
 tree\_depth = tune(),  
 learn\_rate = tune()  
 ) %>%   
 set\_engine("xgboost") %>%   
 set\_mode("classification")  
  
tune\_gbm\_mod  
## Boosted Tree Model Specification (classification)  
##   
## Main Arguments:  
## mtry = tune()  
## trees = tune()  
## tree\_depth = tune()  
## learn\_rate = tune()  
##   
## Computational engine: xgboost

Think of tune() here as a placeholder. After the tuning process, we will select a single numeric value for each of these hyperparameters. For now, we specify our parsnip model object and identify the hyperparameters we will tune().

We can’t train this specification on a single data set (such as the entire training set) and learn what the hyperparameter values should be, but we can train many models using resampled data and see which models turn out best. We can create a regular grid of values to try using some convenience functions for each hyperparameter:

tree\_grid <- grid\_regular(  
 cost\_complexity(),  
 tree\_depth(),  
 levels = 5  
)  
  
tree\_grid  
## # A tibble: 25 x 2  
## cost\_complexity tree\_depth  
## <dbl> <int>  
## 1 0.0000000001 1  
## 2 0.0000000178 1  
## 3 0.00000316 1  
## 4 0.000562 1  
## 5 0.1 1  
## 6 0.0000000001 4  
## 7 0.0000000178 4  
## 8 0.00000316 4  
## 9 0.000562 4  
## 10 0.1 4  
## # … with 15 more rows  
  
# gbm\_grid <- grid\_regular(  
# mtry(),  
# trees(),  
# tree\_depth(),  
# learn\_rate()  
# )

Here, you can see all 5 vaues of cost\_compexity ranging up to 0.1. These values get repeated for each of the 5 values of tree\_depth.

tree\_grid %>%   
 count(tree\_depth)  
## # A tibble: 5 x 2  
## tree\_depth n  
## <int> <int>  
## 1 1 5  
## 2 4 5  
## 3 8 5  
## 4 11 5  
## 5 15 5

Armed with our grid filled with 25 candidate decision tree models, let’s create cross-validation folds for tuning:

set.seed(234)  
cell\_folds <- vfold\_cv(cell\_train)  
  
cell\_folds  
## # 10-fold cross-validation   
## # A tibble: 10 x 2  
## splits id   
## <named list> <chr>   
## 1 <split [1.4K/152]> Fold01  
## 2 <split [1.4K/152]> Fold02  
## 3 <split [1.4K/152]> Fold03  
## 4 <split [1.4K/152]> Fold04  
## 5 <split [1.4K/152]> Fold05  
## 6 <split [1.4K/151]> Fold06  
## 7 <split [1.4K/151]> Fold07  
## 8 <split [1.4K/151]> Fold08  
## 9 <split [1.4K/151]> Fold09  
## 10 <split [1.4K/151]> Fold10

Tuning in tidymodels requires a resampled object created with the rsample package.

## Model tuning with a grid

We are ready to tune. Let’s use [tune\_grid()](https://tidymodels.github.io/tune/reference/tune_grid.html) to fit models at all the different values we chose for each tuned parameter. THere are sveral options for building the object for tuning:

* Tune a model specification along with a recipe or model, or
* Tune a workflow() that bundles together a model specification and a recipe or model preprocessor.

Here we use a workflow() with a straightfoward formula; if this model required more involved data preprocessing, we could use add\_recipe() intead of add\_formula().

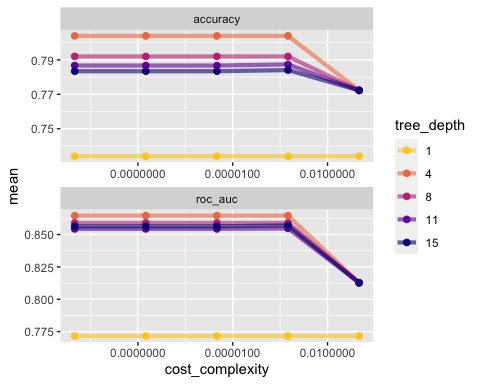
set.seed(345)  
  
tree\_wf <- workflow() %>%  
 add\_model(tune\_spec) %>%  
 add\_formula(class ~ .)  
  
tree\_res <-   
 tree\_wf %>%   
 tune\_grid(  
 resamples = cell\_folds,  
 grid = tree\_grid  
 )  
  
tree\_res  
## # 10-fold cross-validation   
## # A tibble: 10 x 4  
## splits id .metrics .notes   
## <list> <chr> <list> <list>   
## 1 <split [1.4K/152]> Fold01 <tibble [50 × 5]> <tibble [0 × 1]>  
## 2 <split [1.4K/152]> Fold02 <tibble [50 × 5]> <tibble [0 × 1]>  
## 3 <split [1.4K/152]> Fold03 <tibble [50 × 5]> <tibble [0 × 1]>  
## 4 <split [1.4K/152]> Fold04 <tibble [50 × 5]> <tibble [0 × 1]>  
## 5 <split [1.4K/152]> Fold05 <tibble [50 × 5]> <tibble [0 × 1]>  
## 6 <split [1.4K/151]> Fold06 <tibble [50 × 5]> <tibble [0 × 1]>  
## 7 <split [1.4K/151]> Fold07 <tibble [50 × 5]> <tibble [0 × 1]>  
## 8 <split [1.4K/151]> Fold08 <tibble [50 × 5]> <tibble [0 × 1]>  
## 9 <split [1.4K/151]> Fold09 <tibble [50 × 5]> <tibble [0 × 1]>  
## 10 <split [1.4K/151]> Fold10 <tibble [50 × 5]> <tibble [0 × 1]>  
#> # 10-fold cross-validation   
#> # A tibble: 10 x 4  
#> splits id .metrics .notes   
#> \* <list> <chr> <list> <list>   
#> 1 <split [1.4K/152]> Fold01 <tibble [50 × 5]> <tibble [0 × 1]>  
#> 2 <split [1.4K/152]> Fold02 <tibble [50 × 5]> <tibble [0 × 1]>  
#> 3 <split [1.4K/152]> Fold03 <tibble [50 × 5]> <tibble [0 × 1]>  
#> 4 <split [1.4K/152]> Fold04 <tibble [50 × 5]> <tibble [0 × 1]>  
#> 5 <split [1.4K/152]> Fold05 <tibble [50 × 5]> <tibble [0 × 1]>  
#> 6 <split [1.4K/151]> Fold06 <tibble [50 × 5]> <tibble [0 × 1]>  
#> 7 <split [1.4K/151]> Fold07 <tibble [50 × 5]> <tibble [0 × 1]>  
#> 8 <split [1.4K/151]> Fold08 <tibble [50 × 5]> <tibble [0 × 1]>  
#> 9 <split [1.4K/151]> Fold09 <tibble [50 × 5]> <tibble [0 × 1]>  
#> 10 <split [1.4K/151]> Fold10 <tibble [50 × 5]> <tibble [0 × 1]>

Once we have our tuning results, we can both explore them through visualization and then select the best result. THe function collect\_metrics() gives us a tidy tibble with all the rsults. We had 25 candidate models and two metrics accuracy and roc\_auc and we get a row for each metric and model.

tree\_res %>%   
 collect\_metrics()  
## # A tibble: 50 x 7  
## cost\_complexity tree\_depth .metric .estimator mean n std\_err  
## <dbl> <int> <chr> <chr> <dbl> <int> <dbl>  
## 1 0.0000000001 1 accuracy binary 0.734 10 0.00877  
## 2 0.0000000001 1 roc\_auc binary 0.772 10 0.00617  
## 3 0.0000000001 4 accuracy binary 0.804 10 0.00696  
## 4 0.0000000001 4 roc\_auc binary 0.865 10 0.00965  
## 5 0.0000000001 8 accuracy binary 0.792 10 0.0116   
## 6 0.0000000001 8 roc\_auc binary 0.859 10 0.0104   
## 7 0.0000000001 11 accuracy binary 0.787 10 0.0134   
## 8 0.0000000001 11 roc\_auc binary 0.854 10 0.0118   
## 9 0.0000000001 15 accuracy binary 0.783 10 0.0129   
## 10 0.0000000001 15 roc\_auc binary 0.856 10 0.0116   
## # … with 40 more rows

We might get more out of plotting these results:

tree\_res %>%  
 collect\_metrics() %>%  
 mutate(tree\_depth = factor(tree\_depth)) %>%  
 ggplot(aes(cost\_complexity, mean, color = tree\_depth)) +  
 geom\_line(size = 1.5, alpha = 0.6) +  
 geom\_point(size = 2) +  
 facet\_wrap(~ .metric, scales = "free", nrow = 2) +  
 scale\_x\_log10(labels = scales::label\_number()) +  
 scale\_color\_viridis\_d(option = "plasma", begin = .9, end = 0)



We can see that our stubbiest tree, with a depth of 1, is the worst model according to both metrics and across all candidate values of cost\_complexity. Our deepest tree, with a depth of 15, did better. However, the best tree seems to be between these values with a tree depth of 4. The show\_best() function shows us the top 5 candidate models by default:

tree\_res %>%   
 show\_best("roc\_auc")  
## # A tibble: 5 x 7  
## cost\_complexity tree\_depth .metric .estimator mean n std\_err  
## <dbl> <int> <chr> <chr> <dbl> <int> <dbl>  
## 1 0.0000000001 4 roc\_auc binary 0.865 10 0.00965  
## 2 0.0000000178 4 roc\_auc binary 0.865 10 0.00965  
## 3 0.00000316 4 roc\_auc binary 0.865 10 0.00965  
## 4 0.000562 4 roc\_auc binary 0.865 10 0.00965  
## 5 0.0000000001 8 roc\_auc binary 0.859 10 0.0104

We can also use the select\_best() function to pull out the single set of hyperparameter values for our best decision tree model:

best\_tree <-   
 tree\_res %>%   
 select\_best("roc\_auc")  
  
best\_tree  
## # A tibble: 1 x 2  
## cost\_complexity tree\_depth  
## <dbl> <int>  
## 1 0.0000000001 4

These are the values for tree\_depth and cost\_complexity that maximize AUC in this data set of cell images.

## Finalizing our model

We can update (or “finalize”) our workflow object tree\_wf with the values from select\_best().

final\_wf <-   
 tree\_wf %>%   
 finalize\_workflow(best\_tree)  
  
final\_wf  
## ══ Workflow ═══════════════════════════════════════════════════════════════════════════  
## Preprocessor: Formula  
## Model: decision\_tree()  
##   
## ── Preprocessor ───────────────────────────────────────────────────────────────────────  
## class ~ .  
##   
## ── Model ──────────────────────────────────────────────────────────────────────────────  
## Decision Tree Model Specification (classification)  
##   
## Main Arguments:  
## cost\_complexity = 1e-10  
## tree\_depth = 4  
##   
## Computational engine: rpart

## Exploring results

Let’s fit this final model to the training data. What does the decision tree look like?

final\_tree <-   
 final\_wf %>%   
 fit(data = cell\_train)  
  
final\_tree  
## ══ Workflow [trained] ═════════════════════════════════════════════════════════════════  
## Preprocessor: Formula  
## Model: decision\_tree()  
##   
## ── Preprocessor ───────────────────────────────────────────────────────────────────────  
## class ~ .  
##   
## ── Model ──────────────────────────────────────────────────────────────────────────────  
## n= 1515   
##   
## node), split, n, loss, yval, (yprob)  
## \* denotes terminal node  
##   
## 1) root 1515 540 PS (0.64356436 0.35643564)   
## 2) total\_inten\_ch\_2< 47256.5 731 63 PS (0.91381669 0.08618331)   
## 4) total\_inten\_ch\_2< 37166 585 19 PS (0.96752137 0.03247863) \*  
## 5) total\_inten\_ch\_2>=37166 146 44 PS (0.69863014 0.30136986)   
## 10) avg\_inten\_ch\_1< 99.15056 98 14 PS (0.85714286 0.14285714) \*  
## 11) avg\_inten\_ch\_1>=99.15056 48 18 WS (0.37500000 0.62500000)   
## 22) fiber\_align\_2\_ch\_3>=1.47949 20 8 PS (0.60000000 0.40000000) \*  
## 23) fiber\_align\_2\_ch\_3< 1.47949 28 6 WS (0.21428571 0.78571429) \*  
## 3) total\_inten\_ch\_2>=47256.5 784 307 WS (0.39158163 0.60841837)   
## 6) fiber\_width\_ch\_1< 11.19756 329 137 PS (0.58358663 0.41641337)   
## 12) avg\_inten\_ch\_1< 194.4183 254 82 PS (0.67716535 0.32283465) \*  
## 13) avg\_inten\_ch\_1>=194.4183 75 20 WS (0.26666667 0.73333333)   
## 26) total\_inten\_ch\_3>=62458.5 23 9 PS (0.60869565 0.39130435) \*  
## 27) total\_inten\_ch\_3< 62458.5 52 6 WS (0.11538462 0.88461538) \*  
## 7) fiber\_width\_ch\_1>=11.19756 455 115 WS (0.25274725 0.74725275)   
## 14) shape\_p\_2\_a\_ch\_1>=1.225676 300 97 WS (0.32333333 0.67666667)   
## 28) avg\_inten\_ch\_2>=362.0108 55 23 PS (0.58181818 0.41818182) \*  
## 29) avg\_inten\_ch\_2< 362.0108 245 65 WS (0.26530612 0.73469388) \*  
## 15) shape\_p\_2\_a\_ch\_1< 1.225676 155 18 WS (0.11612903 0.88387097) \*

This final\_tree object has the finalized, fitted model object inside. You may want to extract the model object from the workflow. To do this, you can use the helper function pull\_workflow\_fit().

For example, perhaps we woul also like to understand what variables are important in this final model. We can use the [vip](https://koalaverse.github.io/vip/) package to estimate variable importance.

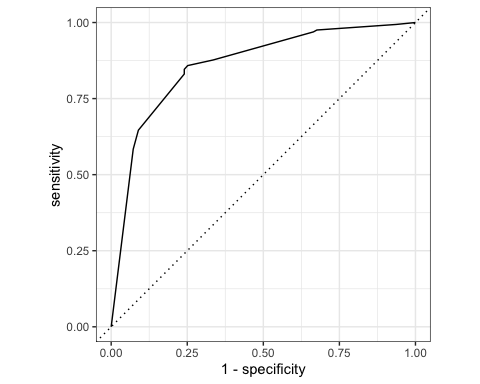
library(vip)  
  
final\_tree %>%   
 pull\_workflow\_fit() %>%   
 vip::vip()

These are the automated image analysis measurements that are the most important in driving segmentation quality predictions

## The last fit

Finally, let’s return to our test data and estimte the model performance we expect to see with new data. We can use the function last\_fit() with our finalized model; this function fits the finalized model on the full training data set and evaluates the finalized model on the testing data.

final\_fit <-   
 final\_wf %>%   
 last\_fit(cell\_split)  
  
final\_fit %>%   
 collect\_metrics()  
## # A tibble: 2 x 3  
## .metric .estimator .estimate  
## <chr> <chr> <dbl>  
## 1 accuracy binary 0.802  
## 2 roc\_auc binary 0.860  
  
final\_fit %>%   
 collect\_predictions() %>%   
 roc\_curve(class, .pred\_PS) %>%   
 autoplot()



The performance metric from the test set indicates that we did not overfit during our tuning procedure.

We leave it to the reader to xplore whether you can tune a different decision tree hyperparameter. You can explore the [reference docs](https://www.tidymodels.org/find/parsnip/#models), or use the args() function to see which parsnip object arguments are available:

args(decision\_tree)  
## function (mode = "unknown", cost\_complexity = NULL, tree\_depth = NULL,   
## min\_n = NULL)   
## NULL

You could tune the other hyperparameter we didn’t use here, min\_n, which sets the minimum n to split at any node. This is another early stopping method for decision trees that can help prevent overfitting. Use this [searchable table](https://www.tidymodels.org/find/parsnip/#model-args) to find the original argument for min\_n in the rpart package (hint). See whether you can tune a different combination of hyperparameters and/or values to improve a tree’s ability to predict cell segmentation quality.

# [A predictive modeling case study](https://www.tidymodels.org/start/case-study/)

## Introduction

Each of the four previous *Get Started* articles has focused on a single task related to modeling. Along the way, we also introduced core packages in the tidymodels ecosystem and some of the key functions you will need to start working with models.

In this final case setudy, we will use all of the previous articles as a foundation to build a predictive model from beggining to end with data on hotel stays.

library(tidymodels)   
  
# Helper packages  
library(readr) # for importing data  
library(vip) # for variable importance plots

## The hotel booking data

Let’s use hotel bookings data from [Antonio, Almeida, and Nunes (2019)](https://doi.org/10.1016/j.dib.2018.11.126) to predict which hotel stays included children and/or babies, based on the other characteristics of the stays such as which hotel the guests stay at, how much they pay, etc. This was also a [#TidyTuesday](https://github.com/rfordatascience/tidytuesday/tree/master/data/2020/2020-02-11) dataset with a [data dictionary](https://github.com/rfordatascience/tidytuesday/tree/master/data/2020/2020-02-11#data-dictionary) you may want to look over to learn more about the variables. We’ll use a slightly edited version of the dataset for this case study.

To start, let’s read our hotel data into R, which we’ll do by providing readr::read\_csv() with a url where our CSV data is located (“<https://tidymodels.org/start/case-study/hotels.csv>”):

hotels <-   
 read\_csv('https://tidymodels.org/start/case-study/hotels.csv') %>%   
 mutate\_if(is.character, as.factor)  
  
dim(hotels)  
## [1] 50000 23

In the original paper, the authors caution that the distribution of many variables (such as number of adults/children, room type, meals bought, country of origin of the guests, and so forth) is different for hotel stays that were canceled versus not canceled. This makes sense because much of that information is gathered (or gathered again more accurately) when guests check in for their stay, so canceled bookings are likely to have more missing data than non-canceled bookings, and/or to have different characteristics when data is not missing. Given this, it is unlikely that we can reliably detect meaningful differences between guests who cancel their bookings and those who do not with this dataset. To build our models here, we have already filtered the data to include only the bookings that did not cancel, so we’ll be analyzing hotel stays only.