Tidyverse and Tidymodels packages

Koji Mizumura

December 22nd,2018 - 2019-01-22

Table of Contents

# Broom

## [Broom by blog post by David Robinson](http://varianceexplained.org/r/broom-intro/)

The concept of “tidy data” offers a powerful framework for data manipulation, analysis and visualization. Popular packages like dplyr, tidyr and ggplot2 take greate advantage of this framework. Please explore several recent posts by others.

But there is an important step in a tidy data workflow that so far has been missing: the **output** of R statistical modeling functions isn’t tidy, meaning it’s difficult to manipulate and recombine in downstream analyses and visualizations. Hadley’s paper makes a convincing statement of this problem.

While model inputs usually require tidy inputs, such attention to detail doesn’t carry over to model outputs. Outputs such as predictions and estimated coefficients aren’t always tidy. This makes it more difficult to combine results from multiple models. For example, in R, the default representation of model coefficients is not tidy because it does not have an explicit variable that records the variable name for each estimate, they are instead recorded as row names. In R, row names must be unique, so combining coefficients from many models (e.g., from bootstrap resamples, or subgroups) requires workarounds to avoid losing important information. This knocks you out of the flow of analysis and makes it harder to combine the results from multiple models. I’m not currently aware of any packages that resolve this problem.

In this [new paper](https://arxiv.org/abs/1412.3565) I introduce the broom package available on CRAN, which bridges the gap from untidy outputs of predictions and estimations to the tidy data we want to work with. It takes the messy output of built-in statistical functions in R, such as lm, nls, kmeans, or t.test as well as popular third-party packages, like gam, glmnet, survival or lme4, and turns them into tidy data frames. This allows the results to be handed to other tidy packages for downstream analysis: they can be recombined using dplyr or visualized using ggplot2.

### Example: linear regression

As a simple example, consider alinear regression on the built-in mtcars dataset:

fit <- lm(mpg~wt+qsec, data=mtcars)  
summary(fit)

##   
## Call:  
## lm(formula = mpg ~ wt + qsec, data = mtcars)  
##   
## Residuals:  
## Min 1Q Median 3Q Max   
## -4.3962 -2.1431 -0.2129 1.4915 5.7486   
##   
## Coefficients:  
## Estimate Std. Error t value Pr(>|t|)   
## (Intercept) 19.7462 5.2521 3.760 0.000765 \*\*\*  
## wt -5.0480 0.4840 -10.430 2.52e-11 \*\*\*  
## qsec 0.9292 0.2650 3.506 0.001500 \*\*   
## ---  
## Signif. codes: 0 '\*\*\*' 0.001 '\*\*' 0.01 '\*' 0.05 '.' 0.1 ' ' 1  
##   
## Residual standard error: 2.596 on 29 degrees of freedom  
## Multiple R-squared: 0.8264, Adjusted R-squared: 0.8144   
## F-statistic: 69.03 on 2 and 29 DF, p-value: 9.395e-12

This summary shows many kinds of statistics describing the regression: coefficient estimates and p-values, information about the residuals, and model statistics like and the F-statistics. But, this format isn’t convenient if you want to combine and compare multiple models, or plot it using ggplot2: you need to turn it into a data frame.

The broom package provides three tidying methods for turning the contents of this object into a data frame, depending on the level of statistics you’re interested in. If you want statistics about each of the coefficients fit by the model, use the tidy() method:

broom::tidy(fit)

## # A tibble: 3 x 5  
## term estimate std.error statistic p.value  
## <chr> <dbl> <dbl> <dbl> <dbl>  
## 1 (Intercept) 19.7 5.25 3.76 7.65e- 4  
## 2 wt -5.05 0.484 -10.4 2.52e-11  
## 3 qsec 0.929 0.265 3.51 1.50e- 3

Note that the rownames are now added as a column, term, meaning that the data can be combined with other models. Note also that the columns have been given names like std\_error and p.value that are more easily accessed than std\_error and Pr(>|t|). This is true of all data frames broom returns: they’re designed so they can be processed in additional steps.

If you are interested in extracting **per-observation information**, such as fitted values and residuals, use the argument method, which adds these to the original data.

fit %>%   
 broom::augment()

## # A tibble: 32 x 11  
## .rownames mpg wt qsec .fitted .se.fit .resid .hat .sigma  
## \* <chr> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>  
## 1 Mazda RX4 21 2.62 16.5 21.8 0.683 -0.815 0.0693 2.64  
## 2 Mazda RX~ 21 2.88 17.0 21.0 0.547 -0.0482 0.0444 2.64  
## 3 Datsun 7~ 22.8 2.32 18.6 25.3 0.640 -2.53 0.0607 2.60  
## 4 Hornet 4~ 21.4 3.22 19.4 21.6 0.623 -0.181 0.0576 2.64  
## 5 Hornet S~ 18.7 3.44 17.0 18.2 0.512 0.504 0.0389 2.64  
## 6 Valiant 18.1 3.46 20.2 21.1 0.803 -2.97 0.0957 2.58  
## 7 Duster 3~ 14.3 3.57 15.8 16.4 0.701 -2.14 0.0729 2.61  
## 8 Merc 240D 24.4 3.19 20 22.2 0.730 2.17 0.0791 2.61  
## 9 Merc 230 22.8 3.15 22.9 25.1 1.41 -2.32 0.295 2.59  
## 10 Merc 280 19.2 3.44 18.3 19.4 0.491 -0.185 0.0358 2.64  
## # ... with 22 more rows, and 2 more variables: .cooksd <dbl>,  
## # .std.resid <dbl>

Finally, glance() computes per-model statistics such as , AIC, BIC:

fit %>%   
 glance()

## # A tibble: 1 x 11  
## r.squared adj.r.squared sigma statistic p.value df logLik AIC BIC  
## \* <dbl> <dbl> <dbl> <dbl> <dbl> <int> <dbl> <dbl> <dbl>  
## 1 0.826 0.814 2.60 69.0 9.39e-12 3 -74.4 157. 163.  
## # ... with 2 more variables: deviance <dbl>, df.residual <int>

The tidy method makes it easy to construct coefficient plots using **ggplot2**:

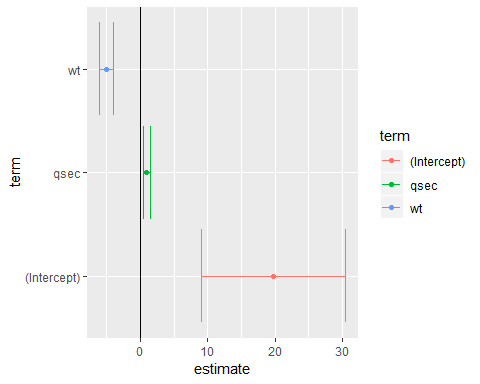
library(ggplot2)

## Warning: package 'ggplot2' was built under R version 3.5.1

td <- fit %>%   
 broom::tidy(conf.int=TRUE)  
  
td %>% head()

## # A tibble: 3 x 7  
## term estimate std.error statistic p.value conf.low conf.high  
## <chr> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>  
## 1 (Intercept) 19.7 5.25 3.76 7.65e- 4 9.00 30.5   
## 2 wt -5.05 0.484 -10.4 2.52e-11 -6.04 -4.06  
## 3 qsec 0.929 0.265 3.51 1.50e- 3 0.387 1.47

ggplot(td, aes(estimate, term, color = term)) +  
 geom\_point() +  
 geom\_errorbarh(aes(xmin = conf.low, xmax = conf.high)) +  
 geom\_vline(xintercept = 0)



When combined with dplyr’s group\_by and do, **broom** also lets you perform regression within groups, such as within automatic and manual cars separately;

library(dplyr)  
  
mtcars %>%   
 group\_by(am) %>%   
 do(broom::tidy(lm(mpg ~ wt, .)))

## # A tibble: 4 x 6  
## # Groups: am [2]  
## am term estimate std.error statistic p.value  
## <dbl> <chr> <dbl> <dbl> <dbl> <dbl>  
## 1 0 (Intercept) 31.4 2.95 10.7 0.00000000601  
## 2 0 wt -3.79 0.767 -4.94 0.000125   
## 3 1 (Intercept) 46.3 3.12 14.8 0.0000000128   
## 4 1 wt -9.08 1.26 -7.23 0.0000169

This is useful for performing regressions or other analyses within each gene, country, or any other kind of division in your tidy dataset.

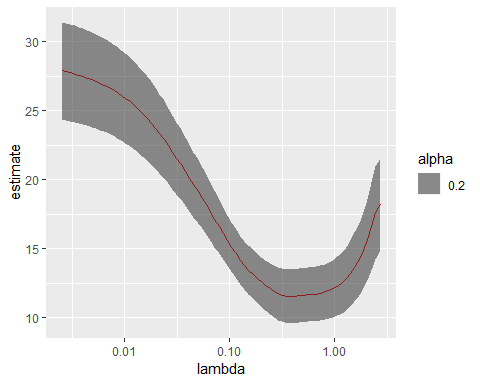
### Using tidiers for visualization with ggplot2

The broom package provides tidying methods for many otherp packages as well. These tidiers serve to connect various statistical models seamlessly with packages like dplyr and ggplot2 . For instance, we could create a LASSO regression with the glmenet package.

library(glmnet)  
set.seed(03-19-2015)  
  
# generate data with 5 real variables and 45 null, on 100 observations  
  
nobs <- 100  
nvar <- 50  
real <- 5  
x <- matrix(rnorm(nobs\*nvar), nobs)  
beta <- c(rnorm(real, 0,1), rep(0, nvar - real))  
y <- c(t(beta) %\*% t(x)) + rnorm(nvar, sd=3)  
  
glmnet\_fit <- cv.glmnet(x,y)

Then, we tidy it with broom and plot it using ggplot2:

tidied\_cv <- glmnet\_fit %>% broom::tidy()  
glance\_cv <- glmnet\_fit %>% broom::tidy()  
  
tidied\_cv %>% ggplot(aes(lambda, estimate))+  
 geom\_line(color="red")+  
 geom\_ribbon(aes(ymin = conf.low, ymax = conf.high, alpha = .2))+  
 scale\_x\_log10()



# geom\_vline(xintercept = glance\_cv$lambda.min) +  
 # geom\_vline(xintercept = glance\_cv$lambda.1se, lty = 2)

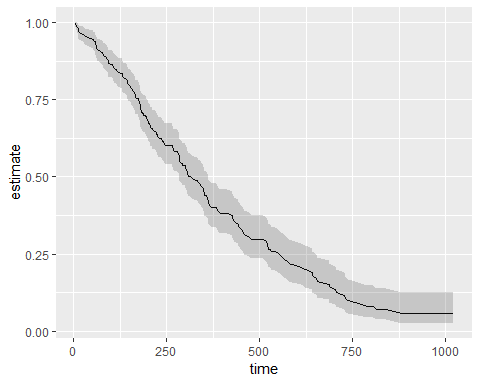
By plotting with ggplot2, rather than relying on glmnet’s built in plotting methods, we gain access to all the tools and framework of the package. This allows us to customize or add attributes, or even to cmplare multiple LASSO cross-validations in the same plot.

The same is true of the [survivial](https://cran.r-project.org/web/packages/survival/index.html) package.

library(survival)

## Warning: package 'survival' was built under R version 3.5.1

surv\_fit <- survfit(coxph(Surv(time, status) ~ age + sex, lung))  
td <- broom::tidy(surv\_fit)  
ggplot(td, aes(time, estimate))+ geom\_line()+  
 geom\_ribbon(aes(ymin=conf.low, ymax=conf.high),alpha=.2)



Others have explored how broom can help visualize [random effects estimated with lme4](https://rstudio-pubs-static.s3.amazonaws.com/38628_54b19baf70b64eb5936a3f1f84beb7da.html). Other packages for which tidiers are implemented include gam, zoo, lfe, and multcomp.

The vignettes for the broom package offer other useful examples, including one on [combining broom and dplyr](https://cran.r-project.org/web/packages/broom/vignettes/broom_and_dplyr.html), a demonstration of [bootstrapping with broom](https://cran.r-project.org/web/packages/broom/vignettes/bootstrapping.html), and a simulation of [k-means clustering](https://cran.r-project.org/web/packages/broom/vignettes/kmeans.html). The broom manuscript offers still more examples.

Tidying model outputs is not an exact science, and it is based on a judgment of the kinds of values a data scientist typically wants out of a tidy analysis (for instance, estimates, test statistics, and p-values). It is my hope that data scientists will propose and contribute their own features feature requests are welcome!) to help expand the universe of tidy analysis tools.

## Visualizing Imer model random effects

I will be exploring the differences between three models:

library(lme4)  
subj\_intercepts\_mod <- lmer(rt ~ A + (1|Subject))  
subjA\_intercepts\_mod <- lmer(rt ~ 1 + (1|Subject:A))  
subj\_slopes\_mod <- lmer(rt ~ A + (A|Subject))

Granted, the second model is rarely encounted in practice. More common would be a nested structure (see this [crossvalidated post](https://stats.stackexchange.com/questions/121504/how-many-random-effects-to-specify-in-lmer))

lmer(rt ~ 1 + (1|Subject) + (1|Subject:A))  
## the above random effects structure is often written as `(1|Subject/A)` the   
## same way `y ~ A + B + A:B` is usually written as `y ~ A \* B`.

Even though the non-nested (1|)

I will work with a modified version of the sleepstudy dataset from lme4.

library(lme4)

## Warning: package 'lme4' was built under R version 3.5.2

library(ggplot2)  
library(reshape2)  
library(dplyr)  
library(broom)  
library(stringr)

head(sleepstudy)

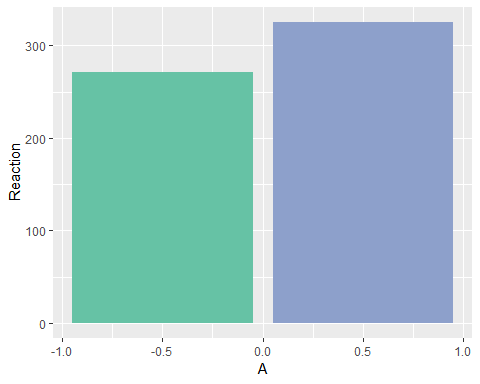
## Reaction Days Subject  
## 1 249.5600 0 308  
## 2 258.7047 1 308  
## 3 250.8006 2 308  
## 4 321.4398 3 308  
## 5 356.8519 4 308  
## 6 414.6901 5 308

example <- sleepstudy %>%   
 mutate(A = ifelse(Days<5, -0.5, 0.5)) %>%   
 select(Subject, A, Reaction)  
  
example %>% head(n=11)

## Subject A Reaction  
## 1 308 -0.5 249.5600  
## 2 308 -0.5 258.7047  
## 3 308 -0.5 250.8006  
## 4 308 -0.5 321.4398  
## 5 308 -0.5 356.8519  
## 6 308 0.5 414.6901  
## 7 308 0.5 382.2038  
## 8 308 0.5 290.1486  
## 9 308 0.5 430.5853  
## 10 308 0.5 466.3535  
## 11 309 -0.5 222.7339

Let’s plot means using ggplot and stat\_summary functions.

base\_plot <- ggplot(example, aes(x=A, y=Reaction))+  
 stat\_summary(aes(fill=factor(A)), fun.y=mean, geom="bar")+  
 scale\_fill\_manual(values=c("#66c2a5", "#8da0cb"))+ #colorbrewer2.rog  
 theme(legend.position = "above")  
  
base\_plot



### Part1: subj\_intercepts\_mod

First, I will fit an lmer model that allows the intercept to vary across subjects.

subj\_intercepts\_mod <- lmer(Reaction ~ A + (1|Subject), data=example)  
# broom::tidy turns fixef(subj\_intercepts\_mod)` into a data.frame  
  
fixed\_params <- subj\_intercepts\_mod %>%   
 broom::tidy(effects="fixed") %>%   
 select(term, estimate)  
  
fixed\_params

## # A tibble: 2 x 2  
## term estimate  
## <chr> <dbl>  
## 1 (Intercept) 299.   
## 2 A 53.8

Although its sometimes helpful to think about model parameters (and you can draw them easily with ggplot::geom\_abline()), I find it more beneficial in a simple design like this to deal in estimates. I will write a little function to speed up this conversion that seems like overkill now but it will come in handy later.

# converts parameters of a `Reaction ~ (Intercept) + A` model into estimates,  
# assumes A is a 0-centered, unit-weighted, dichotomous variable  
  
convert\_parameters\_to\_estimates <- function(tidy\_frame,id\_var="."){  
 tidy\_frame %>%   
 dcast(as.formula(paste(id\_var, "term", sep="~")), value.var = "estimate") %>%   
 mutate(`-0.5` = `(Intercept)` - A/2, `0.5` = `(Intercept)` + A/2) %>%  
 select(-`(Intercept)`, -A) %>%  
 melt(idvars = id\_var, measure.vars = c("-0.5", "0.5"),  
 variable.name = "A", value.name = "Reaction") %>%  
 mutate(A = as.numeric(as.character(A)))  
}  
  
fixed\_estimates <- convert\_parameters\_to\_estimates(fixed\_params)[,c("A","Reaction")]  
  
fixed\_estimates

## A Reaction  
## 1 -0.5 271.6302  
## 2 0.5 325.3856

#sanity check  
example %>%   
 group\_by(A) %>%   
 summarise(Reaction = mean(Reaction)) %>%   
 merge(., fixed\_estimates, by="A", suffixes = c("\_mean", "\_model"))

## A Reaction\_mean Reaction\_model  
## 1 -0.5 271.6302 271.6302  
## 2 0.5 325.3856 325.3856

It is possible to turn parameters from the model into estimates that make sense; now let’s do the same thing with random effects. How will the model’s random effect parameters, when converted to estimates, compare to the average for each subject that we can calculate by hand?

random\_params <- broom::tidy(subj\_intercepts\_mod, effect = "random")  
random\_estimates <- convert\_parameters\_to\_estimates(random\_params, id\_var = "level")  
  
fixed\_slopes\_plot <- base\_plot +   
 geom\_point(data = random\_estimates, shape = 17, size = 3) +  
 geom\_line(aes(group = level), data = random\_estimates)  
fixed\_slopes\_plot

fixed\_slopes\_plot +  
 stat\_summary(aes(group = Subject), fun.y = mean, # means from raw data  
 geom = "point", shape = 19, size = 4, color = "#fc8d62", alpha = 0.6)

Of course, the reason the two sets of points don’t line up is because we are only allowing the subject’s overall Reaction to vary, not the subject’s overall Reaction in each condition. Applying the same slope to each subject, this is the best we can do to account for variance.

base\_plot+  
 geom\_line(aes(group = level), data = random\_estimates)+  
 ## calculate mean Reaction by subject using `stat\_summary`  
 stat\_summary(aes(x=0.0, y=Reaction, group=level), data = random\_estimates, fun.y = mean, geom = "point", shape = 17, size = 3) +  
 stat\_summary(aes(x=0.0, group=Subject), fun.y = mean,  
 geom = "point", shape = 19, size = 4, color = "#fc8d62", alpha = 0.6)

A model that allows intercepts to vary across subjects does just that: it does a great job of estimating overall Reaction for each subject, but it is limited in estimating the effect of A on Reaction.

### Part2: subjA\_intercepts\_mod

We are looking for a way to capture the fact that all of the following by-subject lines don’t have the same slope.

subj\_means\_plot <- base\_plot +  
 stat\_summary(aes(group = Subject), fun.y = mean, geom = "point", shape =19, size =4, color = "#fc8d62")+  
 stat\_summary(aes(group = Subject), fun.y = mean, geom = "line", size = 1.2, color = "#fc8d62")

One way to give the model some flexibility would be to “serve the connection” between the measurements on the left bar from those in the right bar.

example$SubAject <- with(example, paste(Subject, A, sep = ":"))  
subjA\_intercepts\_mod <- lmer(Reaction ~ 1 + (1|SubAject), data = example)

Why make a new, hideously-named variable SubAject? Because if the model can’t understand the relationship between Subject and A, I shouldn’t be able to either! We’ve severed the connection between scores on the left and scores on the right, and given the model more flexibility to estimate the effects.

Of course, nothing is preventing

## [broom and dplyr](https://broom.tidyverse.org/articles/broom_and_dplyr.html)

While broom is useful for summarizing the result of a single analysis in a consistent format, it is really designed for high-throughput applications, where you must combine results from multiple analyses. These could be subgrouped of data, analyses using different models, bootstrap replicates, permutations, and so on. In particular, it plays well with the nest/unnest functions in tidyr and the map function in purrr.

For purrr package, please refer this [RStudio tutorial](https://www.rstudio.com/resources/videos/happy-r-users-purrr-tutorial/).

Let’s try this on a simple dataset, the built-in Orange. We start by coercing Orange to a tibble. This gives a nicer method that will especially useful later on when we start working with list-columns.

library(broom)  
library(tibble)  
  
data("Orange")  
  
  
Orange <- as\_tibble(Orange)  
Orange

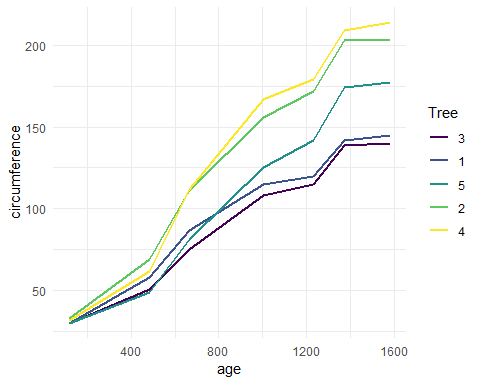
## # A tibble: 35 x 3  
## Tree age circumference  
## \* <ord> <dbl> <dbl>  
## 1 1 118 30  
## 2 1 484 58  
## 3 1 664 87  
## 4 1 1004 115  
## 5 1 1231 120  
## 6 1 1372 142  
## 7 1 1582 145  
## 8 2 118 33  
## 9 2 484 69  
## 10 2 664 111  
## # ... with 25 more rows

This contains 35 observations of three variables: Tree, age and circumference. Tree is a factor with five levels describing five trees. As might be expected, age and circumference are correlated:

cor(Orange$age, Orange$circumference)

## [1] 0.9135189

ggplot2::ggplot(Orange,  
 aes(age, circumference, color=Tree))+  
 geom\_line(size=1)+  
 theme\_minimal()



Suppose you want to test for correlations individually within each tree. You can do this with dplyr’s group\_by:

library(dplyr)  
  
Orange %>%   
 group\_by(Tree) %>%   
 summarise(correlations=cor(age,circumference))

## # A tibble: 5 x 2  
## Tree correlations  
## <ord> <dbl>  
## 1 3 0.988  
## 2 1 0.985  
## 3 5 0.988  
## 4 2 0.987  
## 5 4 0.984

Note that the correlations are much higher than the aggregated one, and furthermore we can now see it is similar across trees. Suppose that instead of simply estimating a correlation, we want to perform a hypothesis test with cor.test:

ct <- cor.test(Orange$age, Orange$circumference)  
ct

##   
## Pearson's product-moment correlation  
##   
## data: Orange$age and Orange$circumference  
## t = 12.9, df = 33, p-value = 1.931e-14  
## alternative hypothesis: true correlation is not equal to 0  
## 95 percent confidence interval:  
## 0.8342364 0.9557955  
## sample estimates:  
## cor   
## 0.9135189

This contains multiple values we could want in our output. Some are vectors of length 1, such as the p-value and the estimate, and some are longer, such as the confidence interval. We can get this into a nicely organized tibble using the tidy function:

ct %>% broom::tidy()

## # A tibble: 1 x 8  
## estimate statistic p.value parameter conf.low conf.high method  
## <dbl> <dbl> <dbl> <int> <dbl> <dbl> <chr>   
## 1 0.914 12.9 1.93e-14 33 0.834 0.956 Pears~  
## # ... with 1 more variable: alternative <chr>

Often, we want to perform multiple tests or fit multiple models, each on different part of the data. In this case, we recommend a nest-map-unnest workflow. For example, suppose we want to perform correlation tests for each different tree. We start by nest ing our data based on the group of interest.

library(tidymodels)  
library(tidyverse)  
  
nested <- Orange %>%   
 nest(-Tree)  
  
nested

## # A tibble: 5 x 2  
## Tree data   
## <ord> <list>   
## 1 1 <tibble [7 x 2]>  
## 2 2 <tibble [7 x 2]>  
## 3 3 <tibble [7 x 2]>  
## 4 4 <tibble [7 x 2]>  
## 5 5 <tibble [7 x 2]>

Then we run a correlation test for each nested tibble using purrr::map:

nested %>%   
 mutate(test = map(data, ~ cor.test(.x$age, .x$circumference)))

## # A tibble: 5 x 3  
## Tree data test   
## <ord> <list> <list>   
## 1 1 <tibble [7 x 2]> <S3: htest>  
## 2 2 <tibble [7 x 2]> <S3: htest>  
## 3 3 <tibble [7 x 2]> <S3: htest>  
## 4 4 <tibble [7 x 2]> <S3: htest>  
## 5 5 <tibble [7 x 2]> <S3: htest>

# i come up with this formula - does this work as well?  
nested %>%   
 mutate(test = map(data, ~ cor.test(.$age, .$circumference)))

## # A tibble: 5 x 3  
## Tree data test   
## <ord> <list> <list>   
## 1 1 <tibble [7 x 2]> <S3: htest>  
## 2 2 <tibble [7 x 2]> <S3: htest>  
## 3 3 <tibble [7 x 2]> <S3: htest>  
## 4 4 <tibble [7 x 2]> <S3: htest>  
## 5 5 <tibble [7 x 2]> <S3: htest>

This results in a list-column of S3 objects. We want to tidy each of the objects, which we can also do with map.

nested %>%   
 mutate(  
 test = map(data, ~ cor.test(.x$age, .x$circumference)), # S3 list-col  
 tidied = map(test, broom::tidy)  
 ) %>%   
 select(Tree, tidied) %>%   
 unnest()

## # A tibble: 5 x 9  
## Tree estimate statistic p.value parameter conf.low conf.high method  
## <ord> <dbl> <dbl> <dbl> <int> <dbl> <dbl> <chr>   
## 1 1 0.985 13.0 4.85e-5 5 0.901 0.998 Pears~  
## 2 2 0.987 13.9 3.43e-5 5 0.914 0.998 Pears~  
## 3 3 0.988 14.4 2.90e-5 5 0.919 0.998 Pears~  
## 4 4 0.984 12.5 5.73e-5 5 0.895 0.998 Pears~  
## 5 5 0.988 14.1 3.18e-5 5 0.916 0.998 Pears~  
## # ... with 1 more variable: alternative <chr>

nested2 <- nested %>%   
 mutate(  
 test = map(data, ~ cor.test(.x$age, .x$circumference)), # S3 list-col  
 tidied = map(test, broom::tidy)  
 )  
   
nested2 %>%   
 select(Tree, tidied) %>%   
 unnest()

## # A tibble: 5 x 9  
## Tree estimate statistic p.value parameter conf.low conf.high method  
## <ord> <dbl> <dbl> <dbl> <int> <dbl> <dbl> <chr>   
## 1 1 0.985 13.0 4.85e-5 5 0.901 0.998 Pears~  
## 2 2 0.987 13.9 3.43e-5 5 0.914 0.998 Pears~  
## 3 3 0.988 14.4 2.90e-5 5 0.919 0.998 Pears~  
## 4 4 0.984 12.5 5.73e-5 5 0.895 0.998 Pears~  
## 5 5 0.988 14.1 3.18e-5 5 0.916 0.998 Pears~  
## # ... with 1 more variable: alternative <chr>

Finally, we want to unnest the tidied data frames so we can see the results in a flat tibble. All together, this looks like:

Orange %>%   
 nest(-Tree) %>%   
 mutate(  
 test = map(data, ~ cor.test(.x$age, .x$circumference)), # S3 list-col  
 tidied = map(test, broom::tidy)  
 ) %>%   
 unnest(tidied, .drop = TRUE)

## # A tibble: 5 x 9  
## Tree estimate statistic p.value parameter conf.low conf.high method  
## <ord> <dbl> <dbl> <dbl> <int> <dbl> <dbl> <chr>   
## 1 1 0.985 13.0 4.85e-5 5 0.901 0.998 Pears~  
## 2 2 0.987 13.9 3.43e-5 5 0.914 0.998 Pears~  
## 3 3 0.988 14.4 2.90e-5 5 0.919 0.998 Pears~  
## 4 4 0.984 12.5 5.73e-5 5 0.895 0.998 Pears~  
## 5 5 0.988 14.1 3.18e-5 5 0.916 0.998 Pears~  
## # ... with 1 more variable: alternative <chr>

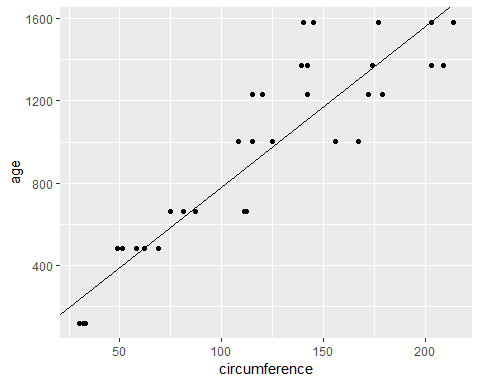
Note that the .drop argument to tidyr::unnest is often useful. This workflow becomes even more useful when applied to regression. Untidy output for a regression looks like:

lm\_fit <- lm(age ~ circumference, data=Orange)  
summary(lm\_fit)

##   
## Call:  
## lm(formula = age ~ circumference, data = Orange)  
##   
## Residuals:  
## Min 1Q Median 3Q Max   
## -317.88 -140.90 -17.20 96.54 471.16   
##   
## Coefficients:  
## Estimate Std. Error t value Pr(>|t|)   
## (Intercept) 16.6036 78.1406 0.212 0.833   
## circumference 7.8160 0.6059 12.900 1.93e-14 \*\*\*  
## ---  
## Signif. codes: 0 '\*\*\*' 0.001 '\*\*' 0.01 '\*' 0.05 '.' 0.1 ' ' 1  
##   
## Residual standard error: 203.1 on 33 degrees of freedom  
## Multiple R-squared: 0.8345, Adjusted R-squared: 0.8295   
## F-statistic: 166.4 on 1 and 33 DF, p-value: 1.931e-14

Orange %>%   
 ggplot()+  
 geom\_point(aes(x=circumference, y=age))+  
 geom\_abline(xintercept=lm\_fit$coefficients[[1]], slope=lm\_fit$coefficients[[2]])

## Warning: Ignoring unknown parameters: xintercept



where we tidy these results, we get multiple rows of output for each model:

broom::tidy(lm\_fit)

## # A tibble: 2 x 5  
## term estimate std.error statistic p.value  
## <chr> <dbl> <dbl> <dbl> <dbl>  
## 1 (Intercept) 16.6 78.1 0.212 8.33e- 1  
## 2 circumference 7.82 0.606 12.9 1.93e-14

Now we can handle multiple regressions at once using exactly the same workflow as before:

Orange %>%   
 nest(-Tree) %>%   
 mutate(  
 fit = map(data, ~ lm(age~circumference, data=.x)),  
 tidied = map(fit, broom::tidy)  
 ) %>%   
 unnest(tidied)

## # A tibble: 10 x 6  
## Tree term estimate std.error statistic p.value  
## <ord> <chr> <dbl> <dbl> <dbl> <dbl>  
## 1 1 (Intercept) -265. 98.6 -2.68 0.0436   
## 2 1 circumference 11.9 0.919 13.0 0.0000485  
## 3 2 (Intercept) -132. 83.1 -1.59 0.172   
## 4 2 circumference 7.80 0.560 13.9 0.0000343  
## 5 3 (Intercept) -210. 85.3 -2.46 0.0574   
## 6 3 circumference 12.0 0.835 14.4 0.0000290  
## 7 4 (Intercept) -76.5 88.3 -0.867 0.426   
## 8 4 circumference 7.17 0.572 12.5 0.0000573  
## 9 5 (Intercept) -54.5 76.9 -0.709 0.510   
## 10 5 circumference 8.79 0.621 14.1 0.0000318

You can just as easily use multiple predictors in the regressions, as shown here on the mtcars dataset. We nest the data into automatic and manual cars (the am column), then peform the regression within each nested tibble.

data("mtcars")  
mtcars <- as\_tibble(mtcars) # to play nicely with list-cols  
mtcars

## # A tibble: 32 x 11  
## mpg cyl disp hp drat wt qsec vs am gear carb  
## \* <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>  
## 1 21 6 160 110 3.9 2.62 16.5 0 1 4 4  
## 2 21 6 160 110 3.9 2.88 17.0 0 1 4 4  
## 3 22.8 4 108 93 3.85 2.32 18.6 1 1 4 1  
## 4 21.4 6 258 110 3.08 3.22 19.4 1 0 3 1  
## 5 18.7 8 360 175 3.15 3.44 17.0 0 0 3 2  
## 6 18.1 6 225 105 2.76 3.46 20.2 1 0 3 1  
## 7 14.3 8 360 245 3.21 3.57 15.8 0 0 3 4  
## 8 24.4 4 147. 62 3.69 3.19 20 1 0 4 2  
## 9 22.8 4 141. 95 3.92 3.15 22.9 1 0 4 2  
## 10 19.2 6 168. 123 3.92 3.44 18.3 1 0 4 4  
## # ... with 22 more rows

mtcars %>%   
 nest(-am) %>%   
 mutate(  
 fit = map(data, ~ lm(wt ~ mpg + qsec + gear, data=.x)),  
 tidied = map(fit, broom::tidy)  
 ) %>%   
 unnest(tidied)

## # A tibble: 8 x 6  
## am term estimate std.error statistic p.value  
## <dbl> <chr> <dbl> <dbl> <dbl> <dbl>  
## 1 1 (Intercept) 4.28 3.46 1.24 0.247   
## 2 1 mpg -0.101 0.0294 -3.43 0.00750   
## 3 1 qsec 0.0398 0.151 0.264 0.798   
## 4 1 gear -0.0229 0.349 -0.0656 0.949   
## 5 0 (Intercept) 4.92 1.40 3.52 0.00309   
## 6 0 mpg -0.192 0.0443 -4.33 0.000591  
## 7 0 qsec 0.0919 0.0983 0.935 0.365   
## 8 0 gear 0.147 0.368 0.398 0.696

What if you want not just the tidy output, but the argument and glance outputs as well, while still performing each regression only once? Since we are using list-columns, we can just fit the model once and use multiple list-columns to store the tidied, glanced and augmented outputs.

regressions <- mtcars %>%   
 nest(-am) %>%   
 mutate(  
 fit = map(data, ~ lm(wt ~ mpg + qsec + gear, data = .x)),  
 tidied = map(fit, broom::tidy),  
 glanced = map(fit, glance),  
 augumented = map(fit, augment)  
 )  
  
regressions %>%   
 unnest(tidied)

## # A tibble: 8 x 6  
## am term estimate std.error statistic p.value  
## <dbl> <chr> <dbl> <dbl> <dbl> <dbl>  
## 1 1 (Intercept) 4.28 3.46 1.24 0.247   
## 2 1 mpg -0.101 0.0294 -3.43 0.00750   
## 3 1 qsec 0.0398 0.151 0.264 0.798   
## 4 1 gear -0.0229 0.349 -0.0656 0.949   
## 5 0 (Intercept) 4.92 1.40 3.52 0.00309   
## 6 0 mpg -0.192 0.0443 -4.33 0.000591  
## 7 0 qsec 0.0919 0.0983 0.935 0.365   
## 8 0 gear 0.147 0.368 0.398 0.696

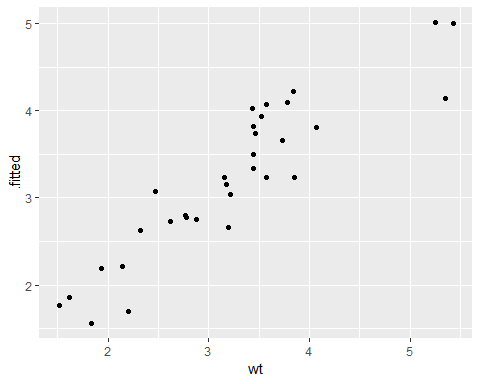
regressions %>%   
 unnest(glanced, .drop=T)

## # A tibble: 2 x 12  
## am r.squared adj.r.squared sigma statistic p.value df logLik  
## <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <int> <dbl>  
## 1 1 0.833 0.778 0.291 15.0 7.59e-4 4 -5.80e-3  
## 2 0 0.625 0.550 0.522 8.32 1.70e-3 4 -1.24e+1  
## # ... with 4 more variables: AIC <dbl>, BIC <dbl>, deviance <dbl>,  
## # df.residual <int>

regressions %>%   
 unnest(augumented)

## # A tibble: 32 x 12  
## am wt mpg qsec gear .fitted .se.fit .resid .hat .sigma  
## <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>  
## 1 1 2.62 21 16.5 4 2.73 0.209 -0.107 0.517 0.304  
## 2 1 2.88 21 17.0 4 2.75 0.152 0.126 0.273 0.304  
## 3 1 2.32 22.8 18.6 4 2.63 0.163 -0.310 0.312 0.279  
## 4 1 2.2 32.4 19.5 4 1.70 0.137 0.505 0.223 0.233  
## 5 1 1.62 30.4 18.5 4 1.86 0.151 -0.244 0.269 0.292  
## 6 1 1.84 33.9 19.9 4 1.56 0.156 0.274 0.286 0.286  
## 7 1 1.94 27.3 18.9 4 2.19 0.113 -0.253 0.151 0.293  
## 8 1 2.14 26 16.7 5 2.21 0.153 -0.0683 0.277 0.307  
## 9 1 1.51 30.4 16.9 5 1.77 0.191 -0.259 0.430 0.284  
## 10 1 3.17 15.8 14.5 5 3.15 0.157 0.0193 0.292 0.308  
## # ... with 22 more rows, and 2 more variables: .cooksd <dbl>,  
## # .std.resid <dbl>

regressions %>%   
 unnest(augumented) %>%   
 ggplot(aes(x=wt, y=.fitted))+  
 geom\_point()



By combining the estimates and p-values across all groups into the same tidy data frame (instead of a list of output model objects), a new class of analyses and visualizations becomes straightforward. This includes

* Sorting by p-value or estimate to find the most significant terms across all tests
* P-value histograms
* Volcano plots comparing p-values to effect size estimates

In each of these cases, we can easily filter, facet, or distinguish based on the term column. In short, this makes the tools of tidy data analysis available for the results of data analysis and models, not just the inputs.

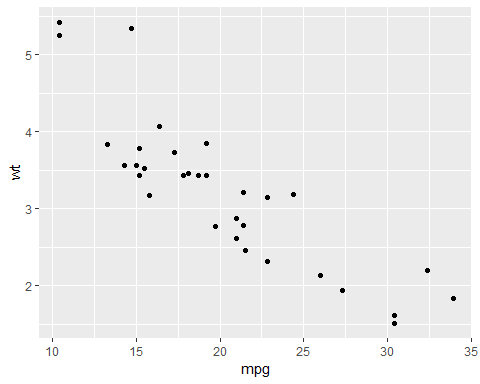
## [Tidy bootstrapping](https://cran.r-project.org/web/packages/broom/vignettes/bootstrapping.html)

Another place where combining model fits in a tidy way becomes useful is when performing bootstrapping or permutation tests. These approach have been explored, for instance, by [Andrew MacDonald here](http://rstudio-pubs-static.s3.amazonaws.com/19698_a4c472606e3c43e4b94720506e49bb7b.html), and [Hadley has explored efficient support for bootstrapping](https://github.com/hadley/dplyr/issues/269) as a potential enhancement to dplyr. broom fits naturally with dplyr in performing these analyses.

Bootstrapping consists of randomly sampling a dataset with replacement, then performing the analysis individually on each bootstrapped replicate. The variation in the resulting estimate is then a reasonable approximation of the variance in our estimate.

Let’s say we want to fit a nonlinear model to the weight/mileage relationship in the mtcars dataset.

library(ggplot2)  
ggplot(mtcars, aes(mpg, wt)) +   
 geom\_point()

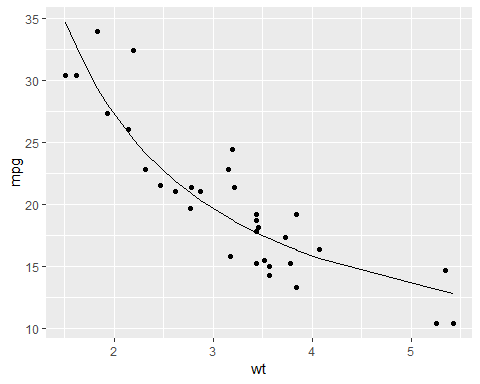


We might use the method of nonlinear least squares (via the nls function) to fit a model.

nlsfit <- nls(mpg~k/wt+b,  
 data = mtcars,  
 start = list(k=1, b=0))  
  
summary(nlsfit)

##   
## Formula: mpg ~ k/wt + b  
##   
## Parameters:  
## Estimate Std. Error t value Pr(>|t|)   
## k 45.829 4.249 10.786 7.64e-12 \*\*\*  
## b 4.386 1.536 2.855 0.00774 \*\*   
## ---  
## Signif. codes: 0 '\*\*\*' 0.001 '\*\*' 0.01 '\*' 0.05 '.' 0.1 ' ' 1  
##   
## Residual standard error: 2.774 on 30 degrees of freedom  
##   
## Number of iterations to convergence: 1   
## Achieved convergence tolerance: 2.877e-08

ggplot(mtcars, aes(wt,mpg))+  
 geom\_point()+  
 geom\_line(aes(y=predict(nlsfit)))



While this does provide a p-value and confidence intervals for the parameters, these are based on model assumptions that may not hold in real data. Bootstrapping is a popular method for providing confidence intervals and predictions that are more robust to the nature of the data.

We cna use the bootstraps function in the rsample package to sample bootstrap replications. First, we construct 100 bootstrap replications of the data, each of which as been randomly sampled with replacement. The resulting object is an rset, which is a dataframe with a column of rsplit objects.

an rsplit object has two main components: an analysis dataset and an assessment dataset, accessible via analysis(rsplit) and asessement(rsplit) respectively. For bootstrap samples, the analysis dataset is the bootstrap sample itself, and the assessment dataset consists of all the out of bag samples.

library(dplyr)  
library(rsample)  
library(broom)  
library(purrr)

set.seed(27)  
boots <- rsample::bootstraps(mtcars, times = 100)  
  
boots %>% head()

## # A tibble: 6 x 2  
## splits id   
## \* <list> <chr>   
## 1 <split [32/12]> Bootstrap001  
## 2 <split [32/15]> Bootstrap002  
## 3 <split [32/10]> Bootstrap003  
## 4 <split [32/10]> Bootstrap004  
## 5 <split [32/12]> Bootstrap005  
## 6 <split [32/12]> Bootstrap006

boots$splits[[1]]

## <32/12/32>

We create a helper function to fit an nls model on each bootstrap sample, and then use purrr::map to apply this to function to all the bootstrap samples at once. Similarly, we create an column of tidy coefficient information by unnesting.

fit\_nls\_on\_bootstrap <- function(split) {  
 nls(mpg ~ k / wt + b, analysis(split), start = list(k = 1, b = 0))  
}  
  
boot\_models <- boots %>%   
 mutate(model = map(splits, fit\_nls\_on\_bootstrap),  
 coef\_info = map(model, broom::tidy))  
  
boot\_coefs <- boot\_models %>%   
 unnest(coef\_info)

The unnested coefficient information contains a summary of each replication combined in a single data frame:

boot\_coefs

## # A tibble: 200 x 6  
## id term estimate std.error statistic p.value  
## <chr> <chr> <dbl> <dbl> <dbl> <dbl>  
## 1 Bootstrap001 k 42.1 3.76 11.2 2.99e-12  
## 2 Bootstrap001 b 5.76 1.43 4.02 3.60e- 4  
## 3 Bootstrap002 k 46.3 3.72 12.4 2.26e-13  
## 4 Bootstrap002 b 4.10 1.43 2.87 7.38e- 3  
## 5 Bootstrap003 k 56.1 3.80 14.7 2.77e-15  
## 6 Bootstrap003 b 0.935 1.31 0.713 4.82e- 1  
## 7 Bootstrap004 k 43.5 3.39 12.8 1.06e-13  
## 8 Bootstrap004 b 4.83 1.33 3.62 1.06e- 3  
## 9 Bootstrap005 k 41.3 3.74 11.0 4.26e-12  
## 10 Bootstrap005 b 5.37 1.31 4.11 2.81e- 4  
## # ... with 190 more rows

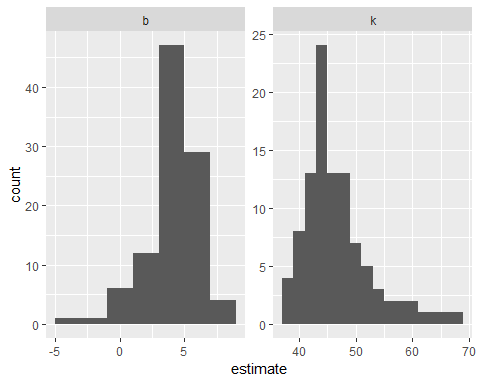
We can then calculate confidence intervals (using what is called the [percentile method](https://www.uvm.edu/~dhowell/StatPages/Randomization%20Tests/ResamplingWithR/BootstMeans/bootstrapping_means.html))

alpha <- .05  
boot\_coefs %>%   
 group\_by(term) %>%   
 summarise(  
 low = quantile(estimate, alpha /2),  
 high = quantile(estimate, 1-alpha/2)  
 )

## # A tibble: 2 x 3  
## term low high  
## <chr> <dbl> <dbl>  
## 1 b -0.695 7.40  
## 2 k 38.7 62.3

or we can use histograms to get a more detailed idea of the uncertainty in each estimate:

ggplot(boot\_coefs, aes(estimate))+  
 geom\_histogram(binwidth = 2)+  
 facet\_wrap(~term, scales = "free")



or we can use augment to visualize the uncertainty in the curve.

boot\_aug <- boot\_models %>%   
 mutate(augmented = map(model,augment)) %>%   
 unnest(augmented)  
  
boot\_aug

## # A tibble: 3,200 x 5  
## id mpg wt .fitted .resid  
## <chr> <dbl> <dbl> <dbl> <dbl>  
## 1 Bootstrap001 21.4 2.78 20.9 0.484  
## 2 Bootstrap001 22.8 2.32 23.9 -1.12   
## 3 Bootstrap001 30.4 1.51 33.6 -3.20   
## 4 Bootstrap001 17.8 3.44 18.0 -0.209  
## 5 Bootstrap001 24.4 3.19 19.0 5.43   
## 6 Bootstrap001 17.3 3.73 17.1 0.243  
## 7 Bootstrap001 22.8 2.32 23.9 -1.12   
## 8 Bootstrap001 21 2.62 21.8 -0.841  
## 9 Bootstrap001 18.7 3.44 18.0 0.691  
## 10 Bootstrap001 14.3 3.57 17.6 -3.26   
## # ... with 3,190 more rows

ggplot(boot\_aug, aes(wt, mpg))+  
 geom\_point()+  
 geom\_line(aes(y=.fitted,group=id),alpha=.2)



With only a few small changes, we could easily perform bootstrapping with other kinds of predictive or hypothetical tesing models, since the tidy functions work for many statistical outputs. As another example, we could use smooth.spline which fits a cubic smoothing spline to data:

fit\_spline\_on\_bootstrap <- function(split){  
 data <- analysis(split)  
 smooth.spline(data$wt, data$mpg, df=4)  
}  
  
boot\_splines <- boots %>%   
 mutate(spline = map(splits, fit\_spline\_on\_bootstrap),  
 aug\_train = map(spline, augment))  
  
splines\_aug <- boot\_splines %>%   
 unnest(aug\_train)  
  
ggplot(splines\_aug, aes(x,y,))+  
 geom\_point()+  
 geom\_line(aes(y = .fitted,group = id), alpha = .2)



## [kmeans with dplyr and broom](https://cran.r-project.org/web/packages/broom/vignettes/kmeans.html)

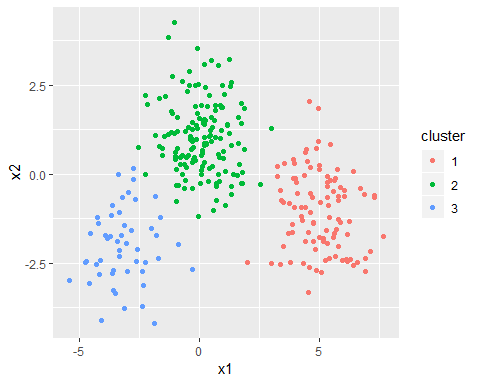
### Tidy k-means clustering

K-means clustering serves as a very useful example of tidy data, and especially the distinction between the three tidying functions: tidy, agument and glance.

Let’s start by generating some random two-dimensional data with three clusters. Data in each cluster will come from a multivariate gaussian distribution with different means for each cluster:

library(dplyr)  
library(ggplot2)  
library(purrr)  
library(tibble)  
library(tidyr)

set.seed(27)  
  
centers <- tibble(  
 cluster = factor(1:3),  
 num\_points = c(100, 150, 50), # the number in each cluster  
 x1 = c(5, 0, -3), # x1 coordinate of cluster center  
 x2 = c(-1, 1, -2) # x2 coordinate of cluster center  
)  
  
labeled\_points <- centers %>%   
 mutate(  
 x1 = map2(num\_points, x1, rnorm),  
 x2 = map2(num\_points, x2, rnorm)  
 ) %>%   
 select(-num\_points) %>%   
 unnest(x1, x2)  
  
ggplot(labeled\_points, aes(x1, x2, color=cluster))+  
 geom\_point()



This is an ideal case for k-means clustering. We’ll use the built-in kmeans function, which accepts a data frame with all numeric columns as its prinary argument.

points <- labeled\_points %>%   
 select(-cluster)  
  
points %>% head()

## # A tibble: 6 x 2  
## x1 x2  
## <dbl> <dbl>  
## 1 6.91 -2.74   
## 2 6.14 -2.45   
## 3 4.24 -0.946  
## 4 3.54 0.287  
## 5 3.91 0.408  
## 6 5.30 -1.58

kclust <- kmeans(points, centers = 3)  
kclust

## K-means clustering with 3 clusters of sizes 51, 101, 148  
##   
## Cluster means:  
## x1 x2  
## 1 -3.14292460 -2.000043  
## 2 5.00401249 -1.045811  
## 3 0.08853475 1.045461  
##   
## Clustering vector:  
## [1] 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2  
## [36] 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2  
## [71] 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 3 3 3 3 3  
## [106] 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3  
## [141] 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3  
## [176] 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 2 3 3 3 3 3 3 3 3 3 3 3  
## [211] 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 1 3  
## [246] 3 3 3 3 3 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1  
## [281] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1  
##   
## Within cluster sum of squares by cluster:  
## [1] 108.8112 243.2092 298.9415  
## (between\_SS / total\_SS = 82.5 %)  
##   
## Available components:  
##   
## [1] "cluster" "centers" "totss" "withinss"   
## [5] "tot.withinss" "betweenss" "size" "iter"   
## [9] "ifault"

summary(kclust)

## Length Class Mode   
## cluster 300 -none- numeric  
## centers 6 -none- numeric  
## totss 1 -none- numeric  
## withinss 3 -none- numeric  
## tot.withinss 1 -none- numeric  
## betweenss 1 -none- numeric  
## size 3 -none- numeric  
## iter 1 -none- numeric  
## ifault 1 -none- numeric

The output is a list of vectors, where each component has a different length. There’s one of length 300: the same as our original dataset. There are number of elements of length 3: withiness, tot.withiness and betweeness and centers is a matrix with 3 rows. And then there are the elements of length 1: totss, tot.withiness, betweenss and iter.

These differing lengths have a deeper meaning when we want to tidy our dataset: they signify that each type of component communicates a *different kind* of information.

* cluster (300 values) contains information about each *point*
* centers, withinss and size (3 values) contain information about each *cluster*
* totss, tot.withinss, betweenss, and iter (1 value) contain information about the *full clustering*

Which of these do we want to extract? There is no right answer: each of them may be interesting to an analyst. Because they communicate entirely different information (not to mention there is no straightfoward way to combine them), they are extracted by separate functions. augment adds the point classifications to the original dataset:

library(broom)  
broom::augment(kclust, points)

## # A tibble: 300 x 3  
## x1 x2 .cluster  
## <dbl> <dbl> <fct>   
## 1 6.91 -2.74 2   
## 2 6.14 -2.45 2   
## 3 4.24 -0.946 2   
## 4 3.54 0.287 2   
## 5 3.91 0.408 2   
## 6 5.30 -1.58 2   
## 7 5.01 -1.77 2   
## 8 6.16 -1.68 2   
## 9 7.13 -2.17 2   
## 10 5.24 -2.42 2   
## # ... with 290 more rows

The tidy function summarizes on a per-cluster level:

broom::tidy(kclust)

## # A tibble: 3 x 5  
## x1 x2 size withinss cluster  
## \* <dbl> <dbl> <int> <dbl> <fct>   
## 1 -3.14 -2.00 51 109. 1   
## 2 5.00 -1.05 101 243. 2   
## 3 0.0885 1.05 148 299. 3

And as it always does, the glance function extracts a single-row summary:

glance(kclust)

## # A tibble: 1 x 4  
## totss tot.withinss betweenss iter  
## <dbl> <dbl> <dbl> <int>  
## 1 3724. 651. 3073. 2

### broom and dplyr for exploratory clustering

While these summaries are useful, they would not have been too difficult to extract out from the dataset yourself. The real power comes from combining these analyses with dplyr.

Let’s say we want to explore theeffects of different choices of k, from 1 to 9, on this clustering. First, cluster the data 9 times, each using a different value of k, then create columns containing the tidied, glanced and augmented data:

kclusts <- tibble(k = 1:9) %>%  
 mutate(  
 kclust = map(k, ~kmeans(points, .x)),  
 tidied = map(kclust, broom::tidy),  
 glanced = map(kclust, glance),  
 augmented = map(kclust, augment, points)  
 )  
  
kclusts

## # A tibble: 9 x 5  
## k kclust tidied glanced augmented   
## <int> <list> <list> <list> <list>   
## 1 1 <S3: kmeans> <tibble [1 x 5]> <tibble [1 x 4]> <tibble [300 x 3]>  
## 2 2 <S3: kmeans> <tibble [2 x 5]> <tibble [1 x 4]> <tibble [300 x 3]>  
## 3 3 <S3: kmeans> <tibble [3 x 5]> <tibble [1 x 4]> <tibble [300 x 3]>  
## 4 4 <S3: kmeans> <tibble [4 x 5]> <tibble [1 x 4]> <tibble [300 x 3]>  
## 5 5 <S3: kmeans> <tibble [5 x 5]> <tibble [1 x 4]> <tibble [300 x 3]>  
## 6 6 <S3: kmeans> <tibble [6 x 5]> <tibble [1 x 4]> <tibble [300 x 3]>  
## 7 7 <S3: kmeans> <tibble [7 x 5]> <tibble [1 x 4]> <tibble [300 x 3]>  
## 8 8 <S3: kmeans> <tibble [8 x 5]> <tibble [1 x 4]> <tibble [300 x 3]>  
## 9 9 <S3: kmeans> <tibble [9 x 5]> <tibble [1 x 4]> <tibble [300 x 3]>

We can turn these into three separate datasets each representing a different type of data: Then tidy the clusterings three ways: using tidy, using augment, and using glance. Each of these goes into a separate dataset as they represent different types of data.

clusters <- kclusts %>%  
 unnest(tidied)

## Warning in bind\_rows\_(x, .id): Unequal factor levels: coercing to character

## Warning in bind\_rows\_(x, .id): binding character and factor vector,  
## coercing into character vector  
  
## Warning in bind\_rows\_(x, .id): binding character and factor vector,  
## coercing into character vector  
  
## Warning in bind\_rows\_(x, .id): binding character and factor vector,  
## coercing into character vector  
  
## Warning in bind\_rows\_(x, .id): binding character and factor vector,  
## coercing into character vector  
  
## Warning in bind\_rows\_(x, .id): binding character and factor vector,  
## coercing into character vector  
  
## Warning in bind\_rows\_(x, .id): binding character and factor vector,  
## coercing into character vector  
  
## Warning in bind\_rows\_(x, .id): binding character and factor vector,  
## coercing into character vector  
  
## Warning in bind\_rows\_(x, .id): binding character and factor vector,  
## coercing into character vector  
  
## Warning in bind\_rows\_(x, .id): binding character and factor vector,  
## coercing into character vector

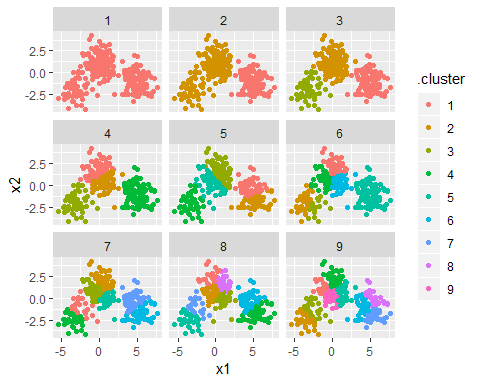
assignments <- kclusts %>%   
 unnest(augmented)

## Warning in bind\_rows\_(x, .id): Unequal factor levels: coercing to character  
  
## Warning in bind\_rows\_(x, .id): binding character and factor vector,  
## coercing into character vector  
  
## Warning in bind\_rows\_(x, .id): binding character and factor vector,  
## coercing into character vector  
  
## Warning in bind\_rows\_(x, .id): binding character and factor vector,  
## coercing into character vector  
  
## Warning in bind\_rows\_(x, .id): binding character and factor vector,  
## coercing into character vector  
  
## Warning in bind\_rows\_(x, .id): binding character and factor vector,  
## coercing into character vector  
  
## Warning in bind\_rows\_(x, .id): binding character and factor vector,  
## coercing into character vector  
  
## Warning in bind\_rows\_(x, .id): binding character and factor vector,  
## coercing into character vector  
  
## Warning in bind\_rows\_(x, .id): binding character and factor vector,  
## coercing into character vector  
  
## Warning in bind\_rows\_(x, .id): binding character and factor vector,  
## coercing into character vector

clusterings <- kclusts %>%   
 unnest(glanced, .drop=TRUE)

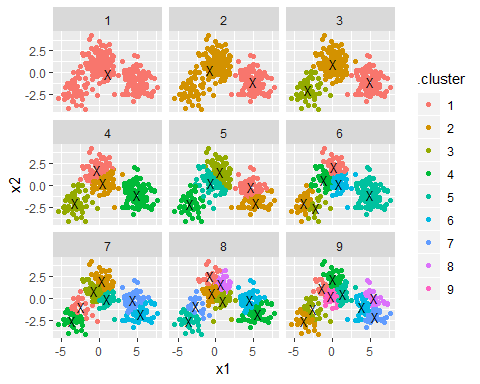
Now we can plot the original points, with each point colored according to the predicted cluster.

p1 <- ggplot(assignments, aes(x1, x2))+  
 geom\_point(aes(color=.cluster))+  
 facet\_wrap(~k)  
  
p1



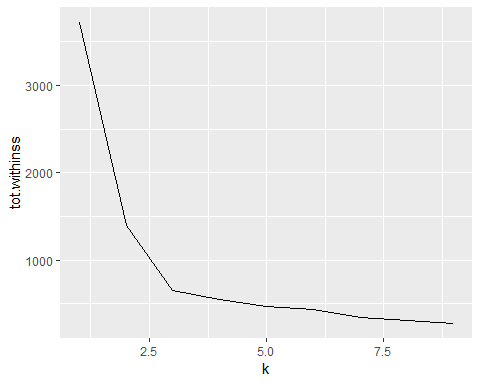
Already we get a good sense of the proper number of clusters (3), and how the k-means algorithm functions when k is too high or too low. We can then add the centers using the data from tidy:

p2 <- p1 + geom\_point(data=clusters, size=3, shape="X")  
p2



The data from glance fits a diffrent but equally important purpose: it lets you view trends of some summary statistics across values of k. Of particular interest is the total within sum of squares, saved in the tot.withinss column.

ggplot(clusterings, aes(k, tot.withinss))+  
 geom\_line()



This represents the variance within clusters. It decreases as k increases, but one can notice a bed (or elborw) right at k=3. This bend indicates that additional clusters beyond the third little value. (See [here](http://web.stanford.edu/~hastie/Papers/gap.pdf) for a more mathematically rigorous interpretation and implementation of this method). Thus, all three methods of tidying data provided by broom are useful for summarizing clustering output.

## Rererence

[Broom Vignette](https://broom.tidyverse.org/)

# rsample

rsample contains a set of functions that can create different types of resamples and corresponding classses for their analysis. The goal is to have modular set of methods that can be used across different R packages for:

* traditional resampling techniques for estimating the sampling distribution of a statistic and
* estimating model performance using a holdout set

The scope of rsample is to provide the basic building blocks for creating and analyzing resamples of a data set but does not include for modeling or calculating statistics. The “Workking with Resample Sets” vignette gives demonstrations of how rsample tools can be used.

install.packages("rsample")  
  
# for the devel version::  
require(devtools)  
devtools::install\_github("tidymodels/rsample")

Note that resampled data sets created by rsample are directly accessible ina resampling object but do not contain much overhead in memory. Since the original data is not modified, R does not make an automatic copy.

For example, creating 50 bootstraps of a data set does not create an object that is 50-fold larger inmemory.

library(rsample)  
library(mlbench)

## Warning: package 'mlbench' was built under R version 3.5.1

library(pryr)

##   
## Attaching package: 'pryr'

## The following objects are masked from 'package:purrr':  
##   
## compose, partial

data(LetterRecognition)  
object\_size(LetterRecognition)

## 2.64 MB

set.seed(35222)  
boots <- rsample::bootstraps(LetterRecognition, times=50)  
  
object\_size(boots)

## 6.69 MB

# object size per resample  
object\_size(boots)/nrow(boots)

## 134 kB

# Fold increase is <<< 50  
as.numeric(object\_size(boots)/object\_size(LetterRecognition))

## [1] 2.529808

## Nested resampling

(A version of this article was originally published in the [*Applied Predictive Modeling* blog](http://appliedpredictivemodeling.com/blog/2017/9/2/njdc83d01pzysvvlgik02t5qnaljnd))

A typical scheme for splitting the data when developing a predictive model is to create an initial split of the data into a training and test set. If resampling is used, it is executed on the training set. A series of binary splits is created. In rsample, we use the term *analysis set* for the data that are used to fit the model and *assessment set* is used to compute performance

A common method for tuning models is grid search where a candidate set of tuning parameters is created. The full set of models for every combination of the tuning parameter grid and the resamples is created. Each time, the assessment data are used to measure performance and the average value is determined for each tuning parameter.

The potential problem is, once we pick the tuning parameter associated with the bet performance, this performance value is usually quoted as the performance of the model. There is a serious potential for *optimization bias* since we uses the same data to tune the model and quote performance. This would result in an optimistinc estimate of performance.

Nested resampling does an additional layer of resampling that separates the tuning activities from the process used to estimate the efficacy of the model. An *outer* resampling scheme is used and, for every split in the outer resample, another full set of resampling splits are created on the original analysis set. For example, if 10-fold cross-validation is used on the outside and 5-fold cross-validation on the inside, a total of 500 models will be fit. The parameter tuning will be conducted 10 times and thebest parameters are determined from the average of the 5 assessment sets. This process occurs 10 times.

We will simulate some regression data to illustrate the methods. The mlbench function mlbench::mlbench.friedman1 can simulate a complex regression data structure from the [original MARS publication](https://scholar.google.com/scholar?hl=en&q=%22Multivariate+adaptive+regression+splines%22&btnG=&as_sdt=1%2C7&as_sdtp=). A training set size of 100 data points are generated as well as a large set that will be used to characterize how well the resampling procedure performed.

library(mlbench)  
sim\_data <- function(n){  
 tmp <- mlbench.friedman1(n, sd=1)  
 tmp <- cbind(tmp$x, tmp$y)  
 tmp <- as.data.frame(tmp)  
 names(tmp)[ncol(tmp)] <- "y"  
 tmp  
}  
  
set.seed(9815)  
train\_dat <- sim\_data(100)  
large\_dat <- sim\_data(10^5)  
  
train\_dat %>% head()

## V1 V2 V3 V4 V5 V6 V7  
## 1 0.1945697 0.6418371 0.9530617 0.7961244 0.2175547 0.9341119 0.1990057  
## 2 0.2281191 0.1046035 0.8010136 0.6872665 0.7132849 0.5495135 0.4451621  
## 3 0.0237278 0.0983911 0.5911238 0.2729825 0.3486822 0.5970778 0.9879835  
## 4 0.7073537 0.8550995 0.8232934 0.5859757 0.5678440 0.1095951 0.6397121  
## 5 0.7844861 0.2843560 0.3885567 0.1277830 0.6481253 0.8126577 0.3983826  
## 6 0.5669030 0.9980239 0.7295834 0.1901266 0.5859250 0.2845121 0.7643961  
## V8 V9 V10 y  
## 1 0.2106022 0.57407261 0.1958863 16.36389  
## 2 0.7320519 0.08607139 0.3788550 12.90426  
## 3 0.6377623 0.56391894 0.9798512 4.12934  
## 4 0.9254958 0.79069775 0.3709754 21.72215  
## 5 0.5251866 0.62025157 0.1792105 11.59917  
## 6 0.2775036 0.82434996 0.1619205 14.79745

train\_dat %>% dim()

## [1] 100 11

To get started, the types of resampling methods need to be specified. This isn’t a large data set, so 5 repeates of 10-fold cross validation will be used as the *outer* resampling method that will be used to generate the estimate of overall performance. To tune the model, it would be good to have precise estimates for each of the values of the tuning parameter so 25 iterations of the bootstrap will be used. fit This means that there will eventually be 5 \* 10 \* 25 = 1250 models that are fit to the data per tuning parameter. These will be discarded once the performance of the model has been quantified.

library(rsample)   
results <- nested\_cv(train\_dat,   
 outside = vfold\_cv(repeats = 5),   
 inside = bootstraps(times = 25))  
results

## [1] "nested\_cv" "vfold\_cv" "rset" "tbl\_df" "tbl"   
## [6] "data.frame"  
## # Nested resampling:  
## # outer: 10-fold cross-validation repeated 5 times  
## # inner: Bootstrap sampling  
## # A tibble: 50 x 4  
## splits id id2 inner\_resamples   
## <list> <chr> <chr> <list>   
## 1 <split [90/10]> Repeat1 Fold01 <tibble [25 x 2]>  
## 2 <split [90/10]> Repeat1 Fold02 <tibble [25 x 2]>  
## 3 <split [90/10]> Repeat1 Fold03 <tibble [25 x 2]>  
## 4 <split [90/10]> Repeat1 Fold04 <tibble [25 x 2]>  
## 5 <split [90/10]> Repeat1 Fold05 <tibble [25 x 2]>  
## 6 <split [90/10]> Repeat1 Fold06 <tibble [25 x 2]>  
## 7 <split [90/10]> Repeat1 Fold07 <tibble [25 x 2]>  
## 8 <split [90/10]> Repeat1 Fold08 <tibble [25 x 2]>  
## 9 <split [90/10]> Repeat1 Fold09 <tibble [25 x 2]>  
## 10 <split [90/10]> Repeat1 Fold10 <tibble [25 x 2]>  
## # ... with 40 more rows

The splitting information for each resample is contained in the split objects. Focusing on the second fold of the first repeat:

results$splits[[2]]

## <90/10/100>

<90/10/100> indicates the number of data in the analysis set, assessment set, and the original data.

Each element of inner\_resamples has its own tibble with the bootstrapping splits.

results$inner\_resamples[[5]]

## # Bootstrap sampling   
## # A tibble: 25 x 2  
## splits id   
## <list> <chr>   
## 1 <split [90/37]> Bootstrap01  
## 2 <split [90/31]> Bootstrap02  
## 3 <split [90/28]> Bootstrap03  
## 4 <split [90/37]> Bootstrap04  
## 5 <split [90/32]> Bootstrap05  
## 6 <split [90/34]> Bootstrap06  
## 7 <split [90/32]> Bootstrap07  
## 8 <split [90/30]> Bootstrap08  
## 9 <split [90/35]> Bootstrap09  
## 10 <split [90/41]> Bootstrap10  
## # ... with 15 more rows

These are self-contained, meaning that the bootstrap sample is aware that it is a sample of a specific 90% of the data:

results$inner\_resamples[[5]]$splits[[1]]

## <90/37/90>

To start, we need to define how the model will be created and measured. For our example, a **radial basis support vector machine** model will be created using the function kernlab::ksvm. This model is generally thought of as having *two* tuning parameters: the SVM cost value and the kernel parameter sigma. For illustration, only the cost value will be tuned and the function kernlab::sigest will be used to estimate sigma during each model fit. This is automatically done by ksvm.

After the model is fit to the analysis set, the root-mean squared error (RMSE) is computed on the assessment set. One important note: for this model, it is critical to center and scale the predictors before computing dot products. We don’t do this operation here because mlbench.friedman1 simulates all of the predictors to be standard uniform random variables.

Our function to fit the model and compute the RMSE is:

library(kernlab)  
  
# `object` will be an `rsplit` object from our `results` tibble  
# `cost` is the tuning parameter  
svm\_rmse <- function(object, cost = 1) {  
 y\_col <- ncol(object$data)  
 mod <- ksvm(y ~ ., data = analysis(object), C = cost)  
 holdout\_pred <- predict(mod, assessment(object)[-y\_col])  
 rmse <- sqrt(mean((assessment(object)$y - holdout\_pred) ^ 2, na.rm = TRUE))  
 rmse  
}  
  
# In some case, we want to parameterize the function over the tuning parameter:  
rmse\_wrapper <- function(cost, object) svm\_rmse(object, cost)

For the nested resampling, a model needs to be fit for each tuning parameter and each bootstrap split. To do this, a wrapper can be created:

library(purrr)  
library(dplyr)  
  
# `object` will be an `rsplit` object for the bootstrap samples  
tune\_over\_cost <- function(object) {  
 results <- tibble(cost = 2 ^ seq(-2, 8, by = 1))  
 results$RMSE <- map\_dbl(results$cost,   
 rmse\_wrapper,  
 object = object)  
 results  
}

Since this will be called across the set of outer cross-validation splits, another wrapper is required:

# `object` is an `rsplit` object in `results$inner\_resamples`   
summarize\_tune\_results <- function(object) {  
 # Return row-bound tibble that has the 25 bootstrap results  
 map\_df(object$splits, tune\_over\_cost) %>%  
 # For each value of the tuning parameter, compute the   
 # average RMSE which is the inner bootstrap estimate.   
 group\_by(cost) %>%  
 summarize(mean\_RMSE = mean(RMSE, na.rm = TRUE),  
 n = length(RMSE))  
}

Now that those functions are defined, we can execute all the inner resampling loops:

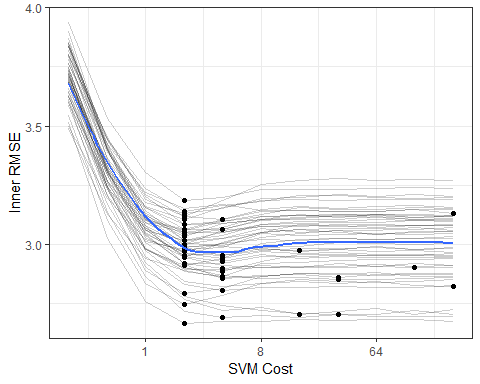
tuning\_results <- map(results$inner\_resamples, summarize\_tune\_results)

tuning\_results is a list of data frames for each of the 50 outer resamples.

Let’s make a plot of the averaged results to see what the relationship is between the RMSE and the tuning parameters for each of the inner bootstrapping operations:

library(ggplot2)  
library(scales)  
  
pooled\_inner <- tuning\_results %>% bind\_rows  
  
best\_cost <- function(dat) dat[which.min(dat$mean\_RMSE),]  
  
p <- ggplot(pooled\_inner, aes(x=cost, y=mean\_RMSE))+  
 scale\_x\_continuous(trans="log2")+  
 xlab("SVM Cost")+ylab("Inner RMSE")  
  
for (i in 1:length(tuning\_results)){  
 p <- p +   
 geom\_line(data=tuning\_results[[i]], alpha=.2)+  
 geom\_point(data=best\_cost(tuning\_results[[i]]), pch = 16)  
}  
  
p <- p+geom\_smooth(data=pooled\_inner, se=FALSE)  
p

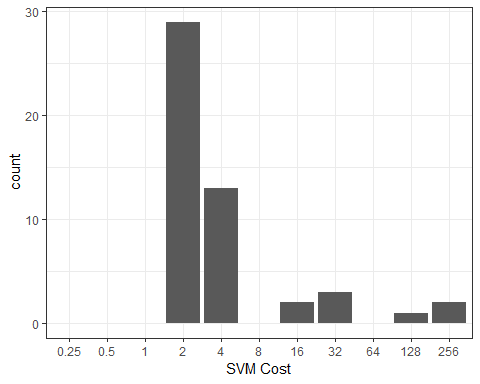
## `geom\_smooth()` using method = 'loess' and formula 'y ~ x'



Each grey line is a separate bootstrap resampling curve created from a different 90% of the data. The blue line is a loess smooth of all the results pooled together.

To determine the best parameter estimate for each of the outer resampling iterations:

cost\_vals <- tuning\_results %>%   
 map\_df(best\_cost) %>%   
 select(cost)  
  
results <- bind\_cols(results, cost\_vals)  
results$cost <- factor(results$cost, levels=paste(2^seq(-2,8,by=1)))  
  
ggplot(results, aes(x=cost))+  
 geom\_bar()+  
 xlab("SVM Cost")+  
 scale\_x\_discrete(drop=FALSE)



Most of the resampling produced and optimal cost values of 2.0 but the distribution is right-skewed due to the flat trend in the resampling profile once the cost value becomes 10 or larger.

Now that we have these estimates, we can compute the outer resampling results for each of the 50 splits using the corresponding tuning parameter value:

results$RMSE <- map2\_dbl(results$splits, results$cost, svm\_rmse)  
summary(results$RMSE)

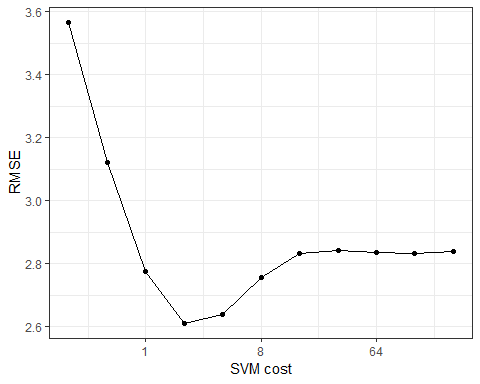
## Min. 1st Qu. Median Mean 3rd Qu. Max.   
## 1.095 2.099 2.636 2.682 3.093 4.177

What is the RMSE estimate for non-nested procedure when only the outer resampling method is used? For each cost value in the tuning grid, 50 SVM models are fit and their RMSE values are averaged. The table of cost values and mean RMSE estimates is used to determine the best cost value. The associated RMSE is the biased estimate.

not\_nested <- map(results$splits, tune\_over\_cost) %>%   
 bind\_rows  
  
outer\_summary <- not\_nested %>%   
 group\_by(cost) %>%   
 summarise(outer\_RMSE = mean(RMSE),  
 n = length(RMSE))  
outer\_summary

## # A tibble: 11 x 3  
## cost outer\_RMSE n  
## <dbl> <dbl> <int>  
## 1 0.25 3.57 50  
## 2 0.5 3.12 50  
## 3 1 2.78 50  
## 4 2 2.61 50  
## 5 4 2.64 50  
## 6 8 2.76 50  
## 7 16 2.83 50  
## 8 32 2.84 50  
## 9 64 2.83 50  
## 10 128 2.83 50  
## 11 256 2.84 50

ggplot(outer\_summary, aes(x = cost, y = outer\_RMSE))+  
 geom\_point()+  
 geom\_line()+  
 scale\_x\_continuous(trans="log2")+  
 xlab("SVM cost")+ylab("RMSE")



The non-nested procedure estimates the RMSE to be 2.61. Both estimates are fairly close.

finalModel <- kernlab::ksvm(y~., data=train\_dat, C = 2)  
large\_pred <- predict(finalModel, large\_dat[, -ncol(large\_dat)])  
sqrt(mean((large\_dat$y - large\_pred)^2, na.rm=TRUE))

## [1] 2.696096

The nested procedure produces a closer estimate to the approximate truth but the non-nested estimate is very similar.

## Recipes with rsample

library(rsample)  
library(recipes)  
library(purrr)

The [recipes](https://topepo.github.io/recipes/) package contains a data preprocessor that can be used to avoid the potentially expensive formula methods as well as providing a richer set of data manipulation tools than base R can provide. This document uses version 0.1.4 of recipes.

In many cases, the preprocessing steps might contain quantities that require statistical estimation of parameters, such as

* signal extraction using principal component analysis
* imputation of missing values
* transformations of individual variables (e.g., Box-Cox transformations)

It is critical that any complex preprocessing steps be contained *inside* of resampling so that the model performance estimates take into account the variability of these steps. Before discussing how rsample can use recipes, let’s look at an example recipe for the Ames housing data.

### An example recipe

For illustration, the Ames housing data will be used. There are sale prices of homes along with various other descriptions for the property:

library(AmesHousing)

## Warning: package 'AmesHousing' was built under R version 3.5.1

ames <- make\_ames()  
names(ames)

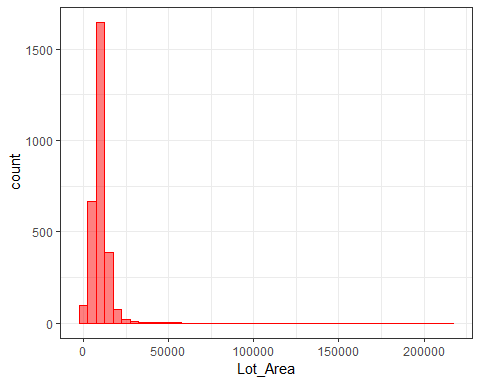
## [1] "MS\_SubClass" "MS\_Zoning" "Lot\_Frontage"   
## [4] "Lot\_Area" "Street" "Alley"   
## [7] "Lot\_Shape" "Land\_Contour" "Utilities"   
## [10] "Lot\_Config" "Land\_Slope" "Neighborhood"   
## [13] "Condition\_1" "Condition\_2" "Bldg\_Type"   
## [16] "House\_Style" "Overall\_Qual" "Overall\_Cond"   
## [19] "Year\_Built" "Year\_Remod\_Add" "Roof\_Style"   
## [22] "Roof\_Matl" "Exterior\_1st" "Exterior\_2nd"   
## [25] "Mas\_Vnr\_Type" "Mas\_Vnr\_Area" "Exter\_Qual"   
## [28] "Exter\_Cond" "Foundation" "Bsmt\_Qual"   
## [31] "Bsmt\_Cond" "Bsmt\_Exposure" "BsmtFin\_Type\_1"   
## [34] "BsmtFin\_SF\_1" "BsmtFin\_Type\_2" "BsmtFin\_SF\_2"   
## [37] "Bsmt\_Unf\_SF" "Total\_Bsmt\_SF" "Heating"   
## [40] "Heating\_QC" "Central\_Air" "Electrical"   
## [43] "First\_Flr\_SF" "Second\_Flr\_SF" "Low\_Qual\_Fin\_SF"   
## [46] "Gr\_Liv\_Area" "Bsmt\_Full\_Bath" "Bsmt\_Half\_Bath"   
## [49] "Full\_Bath" "Half\_Bath" "Bedroom\_AbvGr"   
## [52] "Kitchen\_AbvGr" "Kitchen\_Qual" "TotRms\_AbvGrd"   
## [55] "Functional" "Fireplaces" "Fireplace\_Qu"   
## [58] "Garage\_Type" "Garage\_Finish" "Garage\_Cars"   
## [61] "Garage\_Area" "Garage\_Qual" "Garage\_Cond"   
## [64] "Paved\_Drive" "Wood\_Deck\_SF" "Open\_Porch\_SF"   
## [67] "Enclosed\_Porch" "Three\_season\_porch" "Screen\_Porch"   
## [70] "Pool\_Area" "Pool\_QC" "Fence"   
## [73] "Misc\_Feature" "Misc\_Val" "Mo\_Sold"   
## [76] "Year\_Sold" "Sale\_Type" "Sale\_Condition"   
## [79] "Sale\_Price" "Longitude" "Latitude"

Suppose that we will again fit a simple regression model with the formula:

log10(Sale\_Price)~Neighborhood+House\_Style+Year\_Sold+Lot\_Area

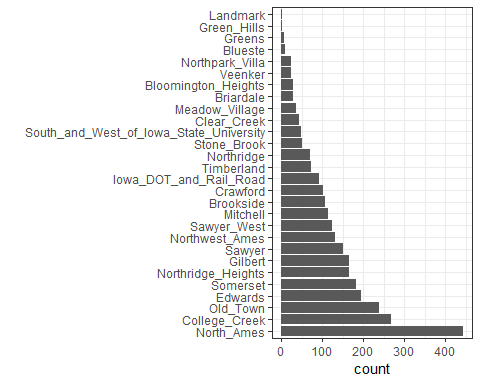
The distribution of the lot size is right-skewed:

library(ggplot2)  
theme\_set(theme\_bw())  
ggplot(ames, aes(x=Lot\_Area))+  
 geom\_histogram(binwidth = 5000, col = "red", fill = "red", alpha = .5)



It might benefit the model if we estimate a transformation of the data using the Box-Cox procedure. Also, note that the frequencies of the neighborhoods can vary:

ggplot(ames, aes(x = Neighborhood))+  
 geom\_bar()+  
 coord\_flip()+  
 xlab("")



When these are resampled, some neighborhood will not be included in the test set and this will result in a column of dummy variables with zero entires. The same is true for the House\_Style variable. We might want to collapse rarely occuring values into other categories.

To define the design matrix, an initial recipe is created:

library(recipes)  
  
rec <- recipe(Sale\_Price ~ Neighborhood + House\_Style + Year\_Sold + Lot\_Area,  
 data = ames) %>%   
 # log the outcome  
 step\_log(Sale\_Price, base = 10) %>%   
 # Collapse rarely occuring jobs into "other"  
 step\_other(Neighborhood, House\_Style, threshold = 0.05) %>%   
 # dummy variables on the qualitative predictors  
 step\_dummy(all\_nominal()) %>%   
 # Unskew a predictor  
 step\_BoxCox(Lot\_Area) %>%   
 # Normalize   
 step\_center(all\_predictors()) %>%   
 step\_scale(all\_predictors())  
  
rec

## Data Recipe  
##   
## Inputs:  
##   
## role #variables  
## outcome 1  
## predictor 4  
##   
## Operations:  
##   
## Log transformation on Sale\_Price  
## Collapsing factor levels for Neighborhood, House\_Style  
## Dummy variables from all\_nominal()  
## Box-Cox transformation on Lot\_Area  
## Centering for all\_predictors()  
## Scaling for all\_predictors()

This recreates the work that the fomula method traditionally uses with the additional steps. While the original data object ames is used in the call, it is only used to define the variables and their characteristics so a single recipe is valid across all resampled versions of the data. The recipe can be estimated on the analysis component of the resample.

If we execute the recipe on the entire data set:

rec\_training\_set <- prep(rec, training=ames)  
rec\_training\_set

## Data Recipe  
##   
## Inputs:  
##   
## role #variables  
## outcome 1  
## predictor 4  
##   
## Training data contained 2930 data points and no missing data.  
##   
## Operations:  
##   
## Log transformation on Sale\_Price [trained]  
## Collapsing factor levels for Neighborhood, House\_Style [trained]  
## Dummy variables from Neighborhood, House\_Style [trained]  
## Box-Cox transformation on Lot\_Area [trained]  
## Centering for Year\_Sold, ... [trained]  
## Scaling for Year\_Sold, ... [trained]

To get the values of the data, the bake function can be used:

# By default, the selector `everything` is used to  
# return all the variables. Other selectors can be used too.  
rec\_training\_set %>%   
 bake(new\_data = ames[1:20,])

## # A tibble: 20 x 14  
## Lot\_Area Year\_Sold Sale\_Price Neighborhood\_Co~ Neighborhood\_Ol~  
## <dbl> <dbl> <dbl> <dbl> <dbl>  
## 1 2.70 1.68 5.33 -0.317 -0.298  
## 2 0.506 1.68 5.02 -0.317 -0.298  
## 3 0.930 1.68 5.24 -0.317 -0.298  
## 4 0.423 1.68 5.39 -0.317 -0.298  
## 5 0.865 1.68 5.28 -0.317 -0.298  
## 6 0.197 1.68 5.29 -0.317 -0.298  
## 7 -1.16 1.68 5.33 -0.317 -0.298  
## 8 -1.12 1.68 5.28 -0.317 -0.298  
## 9 -0.988 1.68 5.37 -0.317 -0.298  
## 10 -0.364 1.68 5.28 -0.317 -0.298  
## 11 0.202 1.68 5.25 -0.317 -0.298  
## 12 -0.244 1.68 5.27 -0.317 -0.298  
## 13 -0.143 1.68 5.26 -0.317 -0.298  
## 14 0.237 1.68 5.23 -0.317 -0.298  
## 15 -0.546 1.68 5.33 -0.317 -0.298  
## 16 3.95 1.68 5.73 -0.317 -0.298  
## 17 0.594 1.68 5.21 -0.317 -0.298  
## 18 0.465 1.68 5.60 -0.317 -0.298  
## 19 1.56 1.68 5.15 -0.317 -0.298  
## 20 0.764 1.68 5.32 -0.317 -0.298  
## # ... with 9 more variables: Neighborhood\_Edwards <dbl>,  
## # Neighborhood\_Somerset <dbl>, Neighborhood\_Northridge\_Heights <dbl>,  
## # Neighborhood\_Gilbert <dbl>, Neighborhood\_Sawyer <dbl>,  
## # Neighborhood\_other <dbl>, House\_Style\_One\_Story <dbl>,  
## # House\_Style\_Two\_Story <dbl>, House\_Style\_other <dbl>

Note that there are fewer dummy variables for Neighborhood and House\_Style than in the data. Also, the above code using prep benefits from the default argument of retain=TRUE, which keeps the processed version of the data set so that we don’t have to reapply the steps to extract the processed values. For the data used to train the recipe, we would have used:

rec\_training\_set %>%   
 juice() %>%   
 head()

## # A tibble: 6 x 14  
## Year\_Sold Lot\_Area Sale\_Price Neighborhood\_Co~ Neighborhood\_Ol~  
## <dbl> <dbl> <dbl> <dbl> <dbl>  
## 1 1.68 2.70 5.33 -0.317 -0.298  
## 2 1.68 0.506 5.02 -0.317 -0.298  
## 3 1.68 0.930 5.24 -0.317 -0.298  
## 4 1.68 0.423 5.39 -0.317 -0.298  
## 5 1.68 0.865 5.28 -0.317 -0.298  
## 6 1.68 0.197 5.29 -0.317 -0.298  
## # ... with 9 more variables: Neighborhood\_Edwards <dbl>,  
## # Neighborhood\_Somerset <dbl>, Neighborhood\_Northridge\_Heights <dbl>,  
## # Neighborhood\_Gilbert <dbl>, Neighborhood\_Sawyer <dbl>,  
## # Neighborhood\_other <dbl>, House\_Style\_One\_Story <dbl>,  
## # House\_Style\_Two\_Story <dbl>, House\_Style\_other <dbl>

The next section will explore recipes and bootstrap resampling for modeling:

library(rsample)  
set.seed(7712)  
bt\_samples <- bootstraps(ames)  
bt\_samples

## # Bootstrap sampling   
## # A tibble: 25 x 2  
## splits id   
## <list> <chr>   
## 1 <split [2.9K/1.1K]> Bootstrap01  
## 2 <split [2.9K/1.1K]> Bootstrap02  
## 3 <split [2.9K/1.1K]> Bootstrap03  
## 4 <split [2.9K/1K]> Bootstrap04  
## 5 <split [2.9K/1.1K]> Bootstrap05  
## 6 <split [2.9K/1.1K]> Bootstrap06  
## 7 <split [2.9K/1.1K]> Bootstrap07  
## 8 <split [2.9K/1K]> Bootstrap08  
## 9 <split [2.9K/1.1K]> Bootstrap09  
## 10 <split [2.9K/1.1K]> Bootstrap10  
## # ... with 15 more rows

bt\_samples$splits[[1]]

## <2930/1076/2930>

### Working with Rsamples

We can add a recipe column to the tibble. recipes has a connivence function called prepper that can be used to call prep but has the split object as the first argument (for easier purring):

library(purrr)  
  
bt\_samples$recipes <- map(bt\_samples$splits, prepper, recipe=rec)  
bt\_samples

## # Bootstrap sampling   
## # A tibble: 25 x 3  
## splits id recipes   
## <list> <chr> <list>   
## 1 <split [2.9K/1.1K]> Bootstrap01 <S3: recipe>  
## 2 <split [2.9K/1.1K]> Bootstrap02 <S3: recipe>  
## 3 <split [2.9K/1.1K]> Bootstrap03 <S3: recipe>  
## 4 <split [2.9K/1K]> Bootstrap04 <S3: recipe>  
## 5 <split [2.9K/1.1K]> Bootstrap05 <S3: recipe>  
## 6 <split [2.9K/1.1K]> Bootstrap06 <S3: recipe>  
## 7 <split [2.9K/1.1K]> Bootstrap07 <S3: recipe>  
## 8 <split [2.9K/1K]> Bootstrap08 <S3: recipe>  
## 9 <split [2.9K/1.1K]> Bootstrap09 <S3: recipe>  
## 10 <split [2.9K/1.1K]> Bootstrap10 <S3: recipe>  
## # ... with 15 more rows

bt\_samples$recipes[[1]]

## Data Recipe  
##   
## Inputs:  
##   
## role #variables  
## outcome 1  
## predictor 4  
##   
## Training data contained 2930 data points and no missing data.  
##   
## Operations:  
##   
## Log transformation on Sale\_Price [trained]  
## Collapsing factor levels for Neighborhood, House\_Style [trained]  
## Dummy variables from Neighborhood, House\_Style [trained]  
## Box-Cox transformation on Lot\_Area [trained]  
## Centering for Year\_Sold, ... [trained]  
## Scaling for Year\_Sold, ... [trained]

Now to fit the model, the fit function only needs the recipe as input. This is because the above code implicitly used the retain=TRUE option in prep. Otherwise, the split objects would also be needed to bake the recipe (as it will in the prediction function below).

fit\_lm <- function(rec\_obj, ...){  
 lm(..., data = juice(rec\_obj, everything()))  
}  
  
bt\_samples$lm\_mod <-   
 map(bt\_samples$recipes,  
 fit\_lm,  
 Sale\_Price ~.  
 )  
  
  
bt\_samples

## # Bootstrap sampling   
## # A tibble: 25 x 4  
## splits id recipes lm\_mod   
## <list> <chr> <list> <list>   
## 1 <split [2.9K/1.1K]> Bootstrap01 <S3: recipe> <S3: lm>  
## 2 <split [2.9K/1.1K]> Bootstrap02 <S3: recipe> <S3: lm>  
## 3 <split [2.9K/1.1K]> Bootstrap03 <S3: recipe> <S3: lm>  
## 4 <split [2.9K/1K]> Bootstrap04 <S3: recipe> <S3: lm>  
## 5 <split [2.9K/1.1K]> Bootstrap05 <S3: recipe> <S3: lm>  
## 6 <split [2.9K/1.1K]> Bootstrap06 <S3: recipe> <S3: lm>  
## 7 <split [2.9K/1.1K]> Bootstrap07 <S3: recipe> <S3: lm>  
## 8 <split [2.9K/1K]> Bootstrap08 <S3: recipe> <S3: lm>  
## 9 <split [2.9K/1.1K]> Bootstrap09 <S3: recipe> <S3: lm>  
## 10 <split [2.9K/1.1K]> Bootstrap10 <S3: recipe> <S3: lm>  
## # ... with 15 more rows

bt\_samples$lm\_mod[[1]] %>% broom::tidy()

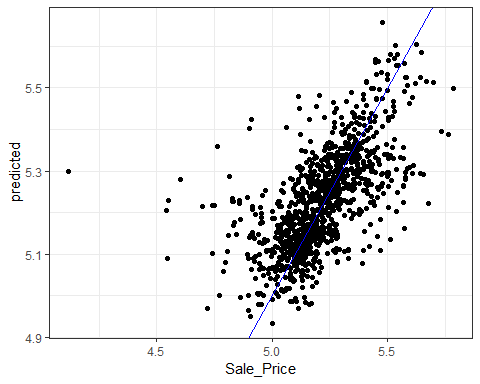
## # A tibble: 15 x 5  
## term estimate std.error statistic p.value  
## <chr> <dbl> <dbl> <dbl> <dbl>  
## 1 (Intercept) 5.22 0.00230 2275. 0.   
## 2 Year\_Sold -0.00242 0.00230 -1.05 2.92e- 1  
## 3 Lot\_Area 0.0723 0.00239 30.3 2.81e-175  
## 4 Neighborhood\_College\_Creek 0.0374 0.00281 13.3 2.91e- 39  
## 5 Neighborhood\_Old\_Town -0.0176 0.00285 -6.18 7.52e- 10  
## 6 Neighborhood\_Edwards -0.00900 0.00270 -3.34 8.54e- 4  
## 7 Neighborhood\_Somerset 0.0504 0.00271 18.6 2.90e- 73  
## 8 Neighborhood\_Northridge\_Heights 0.0734 0.00266 27.6 6.55e-149  
## 9 Neighborhood\_Gilbert 0.0164 0.00276 5.96 2.90e- 9  
## 10 Neighborhood\_Sawyer -0.00647 0.00260 -2.48 1.30e- 2  
## 11 Neighborhood\_Northwest\_Ames 0.0156 0.00261 5.96 2.88e- 9  
## 12 Neighborhood\_other 0.0416 0.00349 11.9 5.70e- 32  
## 13 House\_Style\_One\_Story 0.0162 0.00396 4.10 4.18e- 5  
## 14 House\_Style\_Two\_Story 0.0427 0.00385 11.1 4.93e- 28  
## 15 House\_Style\_other 0.00950 0.00296 3.21 1.33e- 3

To get predictions, the function needs three arguments: the splits (to get the assessment data), the recipe (to process them), and the model. To iterate over these, the function purrr: pmap is used:

pred\_lm <- function(split\_obj, rec\_obj, model\_obj, ...){  
 mod\_data <- bake(  
 rec\_obj,  
 new\_data = assessment(split\_obj),  
 all\_predictors(),  
 all\_outcomes()  
 )  
   
 out <- mod\_data %>% select(Sale\_Price)  
 out$predicted <- predict(model\_obj, newdata = mod\_data %>% select(-Sale\_Price))  
 out  
}  
  
bt\_samples$pred <-   
 pmap(  
 list( #lst  
 split\_obj = bt\_samples$splits,  
 rec\_obj = bt\_samples$recipes,  
 model\_obj = bt\_samples$lm\_mod  
 ),  
 pred\_lm  
 )  
bt\_samples

## # Bootstrap sampling   
## # A tibble: 25 x 5  
## splits id recipes lm\_mod pred   
## <list> <chr> <list> <list> <list>   
## 1 <split [2.9K/1.1K]> Bootstrap01 <S3: recipe> <S3: l~ <tibble [1,076 x ~  
## 2 <split [2.9K/1.1K]> Bootstrap02 <S3: recipe> <S3: l~ <tibble [1,105 x ~  
## 3 <split [2.9K/1.1K]> Bootstrap03 <S3: recipe> <S3: l~ <tibble [1,102 x ~  
## 4 <split [2.9K/1K]> Bootstrap04 <S3: recipe> <S3: l~ <tibble [1,015 x ~  
## 5 <split [2.9K/1.1K]> Bootstrap05 <S3: recipe> <S3: l~ <tibble [1,070 x ~  
## 6 <split [2.9K/1.1K]> Bootstrap06 <S3: recipe> <S3: l~ <tibble [1,058 x ~  
## 7 <split [2.9K/1.1K]> Bootstrap07 <S3: recipe> <S3: l~ <tibble [1,081 x ~  
## 8 <split [2.9K/1K]> Bootstrap08 <S3: recipe> <S3: l~ <tibble [1,048 x ~  
## 9 <split [2.9K/1.1K]> Bootstrap09 <S3: recipe> <S3: l~ <tibble [1,090 x ~  
## 10 <split [2.9K/1.1K]> Bootstrap10 <S3: recipe> <S3: l~ <tibble [1,072 x ~  
## # ... with 15 more rows

bt\_samples$pred[[1]] %>%   
 ggplot(aes(Sale\_Price, predicted))+  
 geom\_point()+  
 geom\_abline(col="blue")



Calculating the RMSE:

library(yardstick)  
  
results <- map\_dfr(bt\_samples$pred, rmse, Sale\_Price, predicted)  
results

## # A tibble: 25 x 3  
## .metric .estimator .estimate  
## <chr> <chr> <dbl>  
## 1 rmse standard 0.135  
## 2 rmse standard 0.137  
## 3 rmse standard 0.133  
## 4 rmse standard 0.127  
## 5 rmse standard 0.130  
## 6 rmse standard 0.126  
## 7 rmse standard 0.126  
## 8 rmse standard 0.136  
## 9 rmse standard 0.131  
## 10 rmse standard 0.140  
## # ... with 15 more rows

mean(results$.estimate)

## [1] 0.1297649

## [Grid search tuning of Keras Models] (<https://github.com/tidymodels/rsample/blob/master/vignettes/Applications/Keras.Rmd>).

Here we demonstrate a single grid search to optimize a tuning parameter of a [keras](https://keras.rstudio.com/index.html) neural network.

The AmesHousing data is used to demonstrate, and there are a number of predictors for these data, but for simplicity, we will see how far we can get by just using the geocodes for the properties as predictors of price. The outcome will be modeled on the log10 scale.

library(AmesHousing)  
library(dplyr)  
ames <- make\_ames() %>%   
 select(Sale\_Price, Longitude, Latitude)

To be consistent with other analysis of thes data, a training/test split is made. However, this article focuses on the training set.

Normally, feature preprocessing should be estimated **within the resampling process** to get generalizable estimates of performance. Here, the two predictors are simply centered and scaled beforehand to avoid complexity in this analysis. However, this is generally a bad idea and the article on [recipe](ttps://topepo.github.io/rsample/articles/Applications/Recipes_and_rsample.html) describes a proper methodology for preprocessing the data.

library(rsample)  
library(dplyr)  
set.seed(4595)  
  
data\_split <- initial\_split(ames,strata = "Sale\_Price")  
  
ames\_train <-   
 training(data\_split) %>%   
 mutate(  
 Sale\_Price = log10(Sale\_Price),  
 Longitude = scale(Longitude, center = TRUE),  
 Latitude = scale(Latitude, center = TRUE)  
 )

To resample the model, simple 10-fold cross-validation is done such that the splits use the outcome as a stratification variable. On average, there should be rfloor(nrow(ames\_train)\*.1 properties in the assessment set and this should be enough to obtain good estimate of the model RMSE.

set.seed(2453)  
cv\_splits <- vfold\_cv(ames\_train, v = 10, strata = "Sale\_Price")

A single layer feed-foward neural network with 10 hidden units will be used to model these data. There are a great many tuning parameters for these models including those for structural aspects (e.g., number of hidden unites, activation type, number of layers), the optimization (momentum dropout rate, etc.) and so on. For simplicity, this article will optimize the number of training epochs (i.e., iterations); basically this is testing for stopping.

A function is needed to compute the model on the analysis set, predict the assessment set, and compute the holdout root mean squared error (in log10 units). The function below constructs the model sequentially and takes the number of epochs as a parameter. The argument split will be used to pass a single elment of cv\_splits$splits. This object will contain the two splits of the data for a single resample. The ellipses (...) will be used to pass arbitrary arguments to keras::fit.

In this function, the seed is set. A few of the model components, such as initializer\_glorot\_uniform and layer\_dropout, use random numbers and their specific seeds are set from the session’s seed. This helps with reproducibility.

library(keras)  
library(yardstick)  
library(purrr)  
  
mlp\_rmse <- function(epoch, split, ...){  
 # set the seed to get reproducible starting values and dropouts  
 set.seed(4109)  
   
 # cleaning the session after the computations have finished  
 # clears memory used by the last trial in preparation for the next iteration  
 on.exit(keras::backend()$clear\_session())  
 # define a single layer MLP with dropout and ReLUs  
 model <- keras\_model\_sequential()  
 model %>%   
 layer\_dense(  
 units = 10,  
 activation = "relu",  
 input\_shape = 2,  
 kernel\_initializer = initializer\_glorot\_uniform()  
 ) %>%   
 layer\_dropout(rate = 0.4) %>%   
 layer\_dense(units = 1, activation = "linear")  
   
 model %>% compile(  
 loss = "mean\_squared\_error",  
 optimizer = optimizer\_rmsprop(),  
 metrics = "mean\_squared\_error"  
 )  
   
 # the data used for modeling (aka the analysis set)  
 geocode <-   
 analysis(split) %>%   
 select(-Sale\_Price) %>%   
 as.matrix()  
   
 model %>% fit(  
 x = geocode,  
 y = analysis(split)[["Sale\_Price"]],  
 epochs = epoch,  
 ...  
 )  
   
 # Now obtain the holdout set for prediction  
 holdout <- assessment(split)  
 pred\_geocode <-   
 holdout %>%   
 select(-Sale\_Price) %>%   
 as.matrix()  
   
 # get predicted values and compute RMSE  
 holdout %>%   
 mutate(predicted = predict(model, pred\_geocode)[,1]) %>%   
 rmse(truth = Sale\_Price, estimate = predicted) %>%   
 pull(.estimate)  
}

Let&s execute the function on the first fold of the data using a batch size of 128 and disable the print/plotting of the optimization:

cv\_splits$splits[[1]]  
  
mlp\_rmse(  
 epoch = 100,  
 cv\_splits$splits[[1]],  
 # options to keras::fit  
 batch\_size = 128,  
 verbose = 0  
)

## [Time series](https://github.com/tidymodels/rsample/blob/master/vignettes/Applications/Time_Series.Rmd)

“[Demo Week: Tidy Forecasting with sweep](http://www.business-science.io/code-tools/2017/10/25/demo_week_sweep.html)” is an excellent article that uses tidy methods with time series. This article uses their analysis with rsample to get performance estimates for future observations using [rolling forecast origin resampling](https://robjhyndman.com/hyndsight/crossvalidation/).

The data are sales of alcholic beverages originally from [the Federal Reserve Bank of St. Louis website](https://fred.stlouisfed.org/series/S4248SM144NCEN).

library(tidymodels)  
data("drinks")  
str(drinks, give.att=FALSE)

## Classes 'tbl\_df', 'tbl' and 'data.frame': 309 obs. of 2 variables:  
## $ date : Date, format: "1992-01-01" "1992-02-01" ...  
## $ S4248SM144NCEN: num 3459 3458 4002 4564 4221 ...

Each row is a month of sales (in millions of US dolloars). Suppose that predictions for one year ahead were needed and the model should use the most recent data from the last 20 years. To setup this resampling scheme:

roll\_rs <- rolling\_origin(  
 drinks,  
 initial = 12\*20,  
 assess = 12,  
 cumulative = FALSE  
)  
  
nrow(roll\_rs)

## [1] 58

roll\_rs

## # Rolling origin forecast resampling   
## # A tibble: 58 x 2  
## splits id   
## <list> <chr>   
## 1 <split [240/12]> Slice01  
## 2 <split [240/12]> Slice02  
## 3 <split [240/12]> Slice03  
## 4 <split [240/12]> Slice04  
## 5 <split [240/12]> Slice05  
## 6 <split [240/12]> Slice06  
## 7 <split [240/12]> Slice07  
## 8 <split [240/12]> Slice08  
## 9 <split [240/12]> Slice09  
## 10 <split [240/12]> Slice10  
## # ... with 48 more rows

Each split element contains the information about the resample:

roll\_rs$splits[[1]]

## <240/12/309>

For plotting, let’s index each split by the first day of the assessment set:

get\_date <- function(x)  
 min(assessment(x)$date)  
  
start\_date <- map(roll\_rs$splits, get\_date)  
roll\_rs$start\_date <- do.call("c", start\_date)  
head(roll\_rs$start\_date)

## [1] "2012-01-01" "2012-02-01" "2012-03-01" "2012-04-01" "2012-05-01"  
## [6] "2012-06-01"

This resampling scheme has 58 splits of the data so that there will be 58 ARIMA models that are fit. To create the models, the auto.arima function from the forecast package is used. The function analysis and assessment return the data frame, so another step converts the data into a ts object called mod\_dat using a function in the timetk package.

library(forecast) # for `auto.arima`  
library(timetk) # for `tk\_ts`   
library(zoo) # for `as.yearmon`  
  
fit\_model <- function(x, ...){  
 # suggested by Matt Dancho:  
 x %>%   
 analysis() %>%   
 # since the first day changes over resamples, adjust it  
 # based on the first date value in the data frame  
 tk\_ts(start = .$date[[1]] %>% as.yearmon(),  
 freq = 12,  
 silent = TRUE) %>%   
 auto.arima(...)  
}

Each model is saved in a new column:

roll\_rs$arima <- map(roll\_rs$splits, fit\_model)  
  
# for example:  
roll\_rs$arima[[1]] %>% broom::tidy()

## # A tibble: 7 x 3  
## term estimate std.error  
## <fct> <dbl> <dbl>  
## 1 ar1 -0.994 0.106   
## 2 ar2 -0.502 0.0810  
## 3 ma1 0.0243 0.111   
## 4 ma2 -0.414 0.125   
## 5 sar1 0.404 0.100   
## 6 sar2 -0.334 0.0759  
## 7 sma1 -0.554 0.0834

(There are some warnings produced by these first regarding extra columns in the data that can be ignored)

Using the model fits, performance will be measured in two ways:

* *interpolation* error will measure how well the model fits to the data that were used to create the model. This is most likely optimistic since no holdout method is used.
* *extrapolation* or *forecast* error evaluates the efficacy of the model on the data from the following year (that were not used in the model fit).

In each case, the mean absolute percent error (MAPE) is the statistic used to characterize the model fits. The interpolation error can be computed from the Arima object. to make things easy, the sweep package’s sw\_glance function is used:

library(sweep)  
  
roll\_rs$interpolation <- map\_dbl(  
 roll\_rs$arima,  
 function(x)  
 sw\_glance(x)[["MAPE"]]  
)  
summary(roll\_rs$interpolation)

## Min. 1st Qu. Median Mean 3rd Qu. Max.   
## 2.84 2.89 2.92 2.93 2.95 3.13

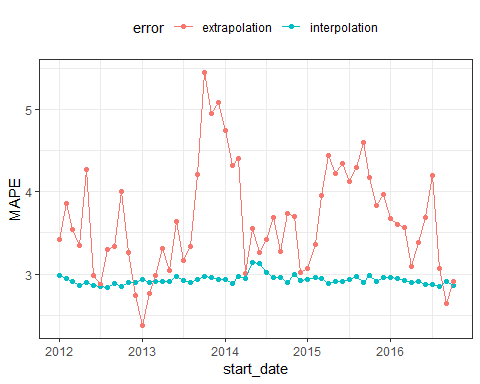
For the extrapolation error, the model and split objects are required. Using these:

get\_extrap <- function(split, mod){  
 n <- nrow(assessment(split))  
 # get asessment data  
 pred\_dat <- assessment(split) %>%   
 mutate(  
 pred = as.vector(forecast(mod, h = n)$mean),  
 pct\_error = ( S4248SM144NCEN - pred ) / S4248SM144NCEN \* 100  
 )  
 mean(abs(pred\_dat$pct\_error))  
}  
  
roll\_rs$extrapolation <-   
 map2\_dbl(roll\_rs$splits, roll\_rs$arima, get\_extrap)  
  
summary(roll\_rs$extrapolation)

## Min. 1st Qu. Median Mean 3rd Qu. Max.   
## 2.37 3.19 3.56 3.65 4.16 5.45

What do these error estimates look like over time?

roll\_rs %>%  
 select(interpolation, extrapolation, start\_date) %>%  
 as.data.frame %>%  
 gather(error, MAPE, -start\_date) %>%  
 ggplot(aes(x = start\_date, y = MAPE, col = error)) +   
 geom\_point() +   
 geom\_line() +   
 theme\_bw() +   
 theme(legend.position = "top")



It is likely that interpolration error is an underestimate to some degree.

It is also worth noting that rolling\_origin() can be used over calendar periods, rather than just over a fixed window size. This is especially useful for irregular series where a fixed window size might not make sense because of missing data points, or because of calender features like different months having a different number of days.

The example below demonstrates this idea by splitting drinks into a nested set of 26 years, and rolling over years rather than months. Note that the end result accomplishes a different task than the original example, in this case, each slice moves forward an entire year, rather than just one month.

# The idea is to nest by the period to roll over,  
# which in this case is the year.  
  
roll\_rs\_annual <- drinks %>%  
 mutate(year = as.POSIXlt(date)$year + 1900) %>%  
 nest(-year) %>%  
 rolling\_origin(  
 initial = 20,   
 assess = 1,   
 cumulative = FALSE  
 )  
  
analysis(roll\_rs\_annual$splits[[1]])

## # A tibble: 20 x 2  
## year data   
## <dbl> <list>   
## 1 1992 <tibble [12 x 2]>  
## 2 1993 <tibble [12 x 2]>  
## 3 1994 <tibble [12 x 2]>  
## 4 1995 <tibble [12 x 2]>  
## 5 1996 <tibble [12 x 2]>  
## 6 1997 <tibble [12 x 2]>  
## 7 1998 <tibble [12 x 2]>  
## 8 1999 <tibble [12 x 2]>  
## 9 2000 <tibble [12 x 2]>  
## 10 2001 <tibble [12 x 2]>  
## 11 2002 <tibble [12 x 2]>  
## 12 2003 <tibble [12 x 2]>  
## 13 2004 <tibble [12 x 2]>  
## 14 2005 <tibble [12 x 2]>  
## 15 2006 <tibble [12 x 2]>  
## 16 2007 <tibble [12 x 2]>  
## 17 2008 <tibble [12 x 2]>  
## 18 2009 <tibble [12 x 2]>  
## 19 2010 <tibble [12 x 2]>  
## 20 2011 <tibble [12 x 2]>

The workflow to access these calender slices is to use bind\_rows() to join each analysis set together.

mutate(  
 roll\_rs\_annual,  
 extracted\_slice = map(splits, ~ bind\_rows(analysis(.x)$data))  
)

## Survival analysis

options(digits = 3)  
library(survival)  
library(purrr)  
library(rsample)  
library(dplyr)  
library(tidyposterior)

## Warning: package 'tidyposterior' was built under R version 3.5.1

##   
## Attaching package: 'tidyposterior'

## The following object is masked from 'package:broom':  
##   
## tidy

library(ggplot2)  
library(tidyr)

In this article, a parameteric analysis of censored daa is conducted and rsample is used to measure the importance of predictors in the model. The data will be used is the NCCTG lung cancer data contained in the survival package:

library(survival)  
str(lung)

## 'data.frame': 228 obs. of 10 variables:  
## $ inst : num 3 3 3 5 1 12 7 11 1 7 ...  
## $ time : num 306 455 1010 210 883 ...  
## $ status : num 2 2 1 2 2 1 2 2 2 2 ...  
## $ age : num 74 68 56 57 60 74 68 71 53 61 ...  
## $ sex : num 1 1 1 1 1 1 2 2 1 1 ...  
## $ ph.ecog : num 1 0 0 1 0 1 2 2 1 2 ...  
## $ ph.karno : num 90 90 90 90 100 50 70 60 70 70 ...  
## $ pat.karno: num 100 90 90 60 90 80 60 80 80 70 ...  
## $ meal.cal : num 1175 1225 NA 1150 NA ...  
## $ wt.loss : num NA 15 15 11 0 0 10 1 16 34 ...

skimr::skim(lung)

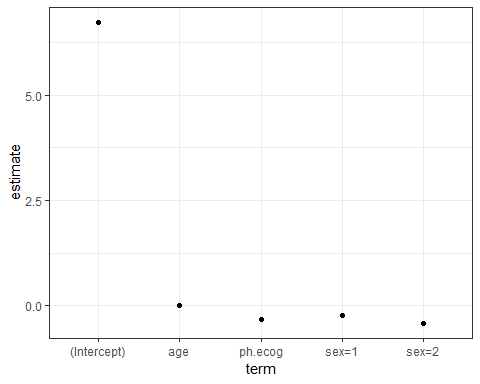
## Skim summary statistics  
## n obs: 228   
## n variables: 10   
##   
## -- Variable type:numeric --------------------------------------------------------------------  
## variable missing complete n mean sd p0 p25 p50 p75  
## age 0 228 228 62.45 9.07 39 56 63 69   
## inst 1 227 228 11.09 8.3 1 3 11 16   
## meal.cal 47 181 228 928.78 402.17 96 635 975 1150   
## pat.karno 3 225 228 79.96 14.62 30 70 80 90   
## ph.ecog 1 227 228 0.95 0.72 0 0 1 1   
## ph.karno 1 227 228 81.94 12.33 50 75 80 90   
## sex 0 228 228 1.39 0.49 1 1 1 2   
## status 0 228 228 1.72 0.45 1 1 2 2   
## time 0 228 228 305.23 210.65 5 166.75 255.5 396.5   
## wt.loss 14 214 228 9.83 13.14 -24 0 7 15.75  
## p100 hist  
## 82 <U+2582><U+2582><U+2585><U+2587><U+2586><U+2587><U+2585><U+2581>  
## 33 <U+2587><U+2582><U+2587><U+2582><U+2582><U+2582><U+2581><U+2581>  
## 2600 <U+2582><U+2585><U+2587><U+2586><U+2581><U+2581><U+2581><U+2581>  
## 100 <U+2581><U+2581><U+2581><U+2583><U+2586><U+2587><U+2587><U+2585>  
## 3 <U+2585><U+2581><U+2587><U+2581><U+2581><U+2583><U+2581><U+2581>  
## 100 <U+2581><U+2582><U+2581><U+2583><U+2587><U+2581><U+2587><U+2583>  
## 2 <U+2587><U+2581><U+2581><U+2581><U+2581><U+2581><U+2581><U+2585>  
## 2 <U+2583><U+2581><U+2581><U+2581><U+2581><U+2581><U+2581><U+2587>  
## 1022 <U+2585><U+2587><U+2586><U+2582><U+2582><U+2582><U+2581><U+2581>  
## 68 <U+2581><U+2582><U+2587><U+2583><U+2582><U+2581><U+2581><U+2581>

status is an indicator for which patients are censored (status=1) or an actual event (status=2). The help file ?survreg has the following model fit:

lung\_mod <- survreg(Surv(time,status)~ ph.ecog + age + strata(sex), data = lung)  
  
#   
summary(lung\_mod)

##   
## Call:  
## survreg(formula = Surv(time, status) ~ ph.ecog + age + strata(sex),   
## data = lung)  
## Value Std. Error z p  
## (Intercept) 6.73235 0.42396 15.88 < 2e-16  
## ph.ecog -0.32443 0.08649 -3.75 0.00018  
## age -0.00581 0.00693 -0.84 0.40193  
## sex=1 -0.24408 0.07920 -3.08 0.00206  
## sex=2 -0.42345 0.10669 -3.97 7.2e-05  
##   
## Scale:  
## sex=1 sex=2   
## 0.783 0.655   
##   
## Weibull distribution  
## Loglik(model)= -1137 Loglik(intercept only)= -1146  
## Chisq= 17.8 on 2 degrees of freedom, p= 0.00014   
## Number of Newton-Raphson Iterations: 5   
## n=227 (1 observation deleted due to missingness)

# coefficient plot  
lung\_mod %>%   
 broom::tidy() %>%   
 ggplot(aes(x=term, y=estimate))+geom\_point()



Note that the stratification on gender only affects the scale parameter: the estimates above are from a log-linear model for the scale parameter even though they are listed with the regression variables for the other parameter. coef gives results that are more clear:

coef(lung\_mod) %>% broom::tidy()

## Warning: 'tidy.numeric' is deprecated.  
## See help("Deprecated")

## # A tibble: 3 x 2  
## names x  
## <chr> <dbl>  
## 1 (Intercept) 6.73   
## 2 ph.ecog -0.324   
## 3 age -0.00581

# coefplot::coefplot(lung\_mod)

To resample these data, it would a good idea to try to maintain the same censoring rate across the splits. To do this, stratified resampling can be used where each analysis/assessment split is conducted within each value of the status indicator. To demonstrate, Monte CArol resampling is used where 75% of the data are in the analysis set. A total of 100 splits are created.

library(rsample)  
set.seed(9666)  
mc\_samp <- mc\_cv(lung, strata="status", times=100)  
  
library(purrr)  
cens\_rate <- function(x) mean(analysis(x)$status==1)  
  
map\_dbl(mc\_samp$splits, cens\_rate) %>%   
 summary()

## Min. 1st Qu. Median Mean 3rd Qu. Max.   
## 0.279 0.279 0.279 0.279 0.279 0.279

To demonstrate the use of resampling with censored data, the parametric model shown above will be fit with different variable sets

## Reference

[Vignettes](https://github.com/tidymodels/rsample/tree/master/vignettes)

# Parsnip

## [introduction](https://tidymodels.github.io/parsnip/)

One issue with different functions available in R that *do the same thing* is that they can have different interfaces and arguments. For example, to fit a random forest classification model, we might have:

# From random forest  
rf\_1 <- randomForest::randomForest(x,y,mtry=12, ntrees=2000, importance=TRUE)  
  
# From ranger  
library(ranger)  
rf\_2 <- ranger(  
 y~.,   
 data = dat,   
 mtry = !2,  
 num.trees = 2000,  
 importance = "impurity"  
)  
  
# From sparklyr  
rf\_3 <- ml\_random\_forest(  
 dat,  
 intercept = FALSE,  
 response = "y",  
 features = names(dat)[names(dat) 1= "y"],  
 col.sample.rate =12,  
 num.trees = 2000  
)

Note that the model syntax is very different and that the argument names (and formals) are also different. This is pain if you go between implementations.

In this example: - the **type** of model is “random forest” - the **mode** of the model is “classification” (as opposed to regression, etc) - the computational **engine** is the name of the R package

The idea of parsnip is to: - separate the definition of a model from its evaluation. - Decouple the model specification from the implementation (whether the implementation is in R, spark or something else). For example, the user would call rand\_forest instead of ranger::ranger or other specific packages. - Harmonize the argument names (e.g., n.trees, ntrees, trees) so that users can remember a single name. This will help across model types too so that trees will be the same argument across random forest as well as boosting or bagging.

Using the example above, the parsnip approach would be:

library(parsnip)  
parsnip::rand\_forest(  
 mtry = 12,  
 trees = 2000  
) %>%   
 set\_engine("ranger", importance = "impurity") %>%   
 fit (y ~., data=dat)

The engine can be easily changed and the mode can be determined when fit is called. to use Spark, the change is simple:

rand\_forest(  
 mtry = 12,  
 trees = 2000  
) %>%   
 set\_engine("spark") %>%   
 fit(y ~ ., data = dat)

### Model list

parsnip contains wrappers for a number of models. For example, the parsnip function rand\_forest() can be used to create a random forest model. The **mode** of a model is realted to its goal. Examples would be regression and classification.

[Ths list of models](https://tidymodels.github.io/parsnip/articles/articles/Models.html) accessible via parsnip is :

* classification: boost\_tree(), decision\_tree(), logistic\_reg(), mars(), mlp(), multinomial\_reg(), nearest\_neighbor(), rand\_forest(), svm\_poly(), svm\_rbf()
* regression: boost\_tree(), decision\_tree(), linear\_reg(), mars(), mlp(), nerarest\_neighbor(), rand\_forest(), surv\_reg(), svm\_reg(), svm\_poly(), svm\_rbf().

How the model is created is related to the *engine*. In many cases, this is an R modeling package. In others, it may be a connection to an externam system (such as SparkorTensorflow).

library(tidyverse)  
readr::write\_csv(ames, path = "C:/Users/kojikm.mizumura/Desktop/ames.csv")

## [Regression](https://github.com/tidymodels/parsnip/blob/master/vignettes/articles/Regression.Rmd)