Today, we will continue our exploration of developments in the world of [tidy models](https://github.com/tidymodels), and we will stick with our usual Fama French modeling flow to do so. For new readers who want get familiar with Fama French before diving into this post, see [here](https://rviews.rstudio.com/2018/04/11/introduction-to-fama-french/) where we covered importing and wrangling the data, [here](https://rviews.rstudio.com/2018/05/10/rolling-fama-french/) where we covered rolling models and visualization, and [here](https://rviews.rstudio.com/2018/11/19/many-factor-models/) where we covered managing many models. If you’re into Shiny, [this flexdashboard](http://www.reproduciblefinance.com/shiny/fama-french-three-factor/) might be of interest, as well.

Let’s get to it.

First, we need our data and, as usual, we’ll import data for daily prices of five ETFs, convert them to returns (have a look [here](http://www.reproduciblefinance.com/2017/09/25/asset-prices-to-log-returns/) for a refresher on that code flow), then import the five Fama French factor data and join it to our five ETF returns data. Here’s the code to make that happen (this code was covered in detail in [this post](http://www.reproduciblefinance.com/2018/06/07/fama-french-write-up-part-one/):

symbols <- c("SPY", "EFA", "IJS", "EEM", "AGG")

# The prices object will hold our daily price data.

prices <-

getSymbols(symbols,

src = 'yahoo',

from = "2012-12-31",

to = "2017-12-31",

auto.assign = TRUE,

warnings = FALSE) %>%

map(~Ad(get(.))) %>%

reduce(merge) %>%

`colnames<-`(symbols)

asset\_returns\_long <-

prices %>%

tk\_tbl(preserve\_index = TRUE, rename\_index = "date") %>%

gather(asset, prices, -date) %>%

group\_by(asset) %>%

mutate(daily\_returns = (log(prices) - log(lag(prices)))) %>%

na.omit()

factors\_data\_address <-

"http://mba.tuck.dartmouth.edu/pages/faculty/ken.french/ftp/Global\_5\_Factors\_Daily\_CSV.zip"

factors\_csv\_name <- "Global\_5\_Factors\_Daily.csv"

temp <- tempfile()

download.file(

# location of file to be downloaded

factors\_data\_address,

# where we want R to store that file

temp,

quiet = TRUE)

Global\_5\_Factors <-

read\_csv(unz(temp, factors\_csv\_name), skip = 6 ) %>%

rename(date = X1, MKT = `Mkt-RF`) %>%

mutate(date = ymd(parse\_date\_time(date, "%Y%m%d")))%>%

mutate\_if(is.numeric, funs(. / 100)) %>%

select(-RF)

data\_joined\_tidy <-

asset\_returns\_long %>%

left\_join(Global\_5\_Factors, by = "date") %>%

na.omit()

For today, let’s work with just the SPY data by filtering our data set by asset.

spy\_2013\_2017 <- data\_joined\_tidy %>%

filter(asset == "SPY")

Next, we re-sample this five years’ worth of data into smaller subsets of training and testing sets. This is frequently done by k-fold cross validation (see [here](http://www.reproduciblefinance.com/2019/03/13/rsampling-fama-french/) for an example), where random samples are taken from the data, but since we are working with time series, we will use a time-aware technique. The [rsample](https://cran.r-project.org/package=rsample) package has a function for exactly this purpose, the rolling\_origin() function. We covered this process extensively in this [previous post](http://www.reproduciblefinance.com/2019/03/14/rolling-origin-fama-french/). Here’s the code to make it happen.

rolling\_origin\_spy\_2013\_2017 <-

rolling\_origin(

data = spy\_2013\_2017,

initial = 100,

assess = 1,

cumulative = FALSE

)

rolling\_origin\_spy\_2013\_2017 %>%

dim()

[1] 1159 2

We now have a data object called rolling\_origin\_spy\_2013\_2017 that holds 1159 splits of data. Each split consists of an analysis data set with 100 days of return and factor data, and an assessment data set with one day of return and factor data.

Now, we can start using that collection of data splits to fit a model on the assessment data, and then test our model on the assessment data. That means it’s time to introduce a relatively new addition to the R tool chain, the [parsnip](https://cran.r-project.org/package=parsnip) package.

parsnip is a unified model interface that allows us to create a model specification, set an analytic engine, and then fit a model. It’s a ‘unified’ interface in the sense that we can use the same scaffolding but insert different models, or different engines, or different modes. Let’s see how that works with linear regression.

Recall that [in the previous post](http://www.reproduciblefinance.com/2019/03/14/rolling-origin-fama-french/), we piped our data into a linear model like so:

analysis(rolling\_origin\_spy\_2013\_2017$splits[[1]]) %>%

do(model = lm(daily\_returns ~ MKT + SMB + HML + RMW + CMA,

data = .)) %>%

tidy(model)

# A tibble: 6 x 6

# Groups: asset [1]

asset term estimate std.error statistic p.value

1 SPY (Intercept) 0.000579 0.000338 1.71 8.98e- 2

2 SPY MKT 0.909 0.0739 12.3 2.79e-21

3 SPY SMB -0.495 0.112 -4.43 2.52e- 5

4 SPY HML -0.609 0.208 -2.92 4.38e- 3

5 SPY RMW -0.591 0.259 -2.28 2.47e- 2

6 SPY CMA -0.395 0.206 -1.92 5.81e- 2

Now, we will pipe into the parsnip scaffolding, which will allow us to quickly change to a different model and specification further down in the code.

Since we are running a linear regression, we first create a specification with linear\_reg(), then set the engine with set\_engine("lm"), and finally fit the model with fit(five\_factor\_model, data = one of our splits)

lm\_model <-

linear\_reg() %>%

set\_engine("lm") %>%

fit(daily\_returns ~ MKT + SMB + HML + RMW + CMA,

data = analysis(rolling\_origin\_spy\_2013\_2017$splits[[1]]))

lm\_model

parsnip model object

Call:

stats::lm(formula = formula, data = data)

Coefficients:

(Intercept) MKT SMB HML RMW

0.0005794 0.9086303 -0.4951297 -0.6085088 -0.5910375

CMA

-0.3954515

Now that we’ve fit the model on our test set, let’s see how well it predicted the test set. We can use the predict() function and pass it the results of our parnsip code flow, along with the assessment split.

assessment(rolling\_origin\_spy\_2013\_2017$splits[[1]]) %>%

select(returns) %>%

bind\_cols(predict(lm\_model,

new\_data = assessment(rolling\_origin\_spy\_2013\_2017$splits[[1]])))

# A tibble: 1 x 3

# Groups: asset [1]

asset returns .pred

1 SPY 148. 0.00737

That worked well, but now let’s head to a more complex model and use the [ranger](https://cran.r-project.org/web/packages/ranger/ranger.pdf) package as an engine for a random forest analysis.

To set up the ranger random forest model in parsnip, we first use rand\_forest(mode = "regression", mtry = 3, trees = 100) to create the specification, set\_engine("ranger") to set the engine as the ranger package, and fit(daily\_returns ~ MKT + SMB + HML + RMW + CMA ~ , data = analysis(rolling\_origin\_spy\_2013\_2017$splits[[1]]) to fit the five-factor Fama French model to the 100-day sample in our first split.

# Need to load the packages to be used as the random forest engine

library(ranger)

rand\_forest(mode = "regression", mtry = 3, trees = 100) %>%

set\_engine("ranger") %>%

fit(daily\_returns ~ MKT + SMB + HML + RMW + CMA,

data = analysis(rolling\_origin\_spy\_2013\_2017$splits[[1]]))

parsnip model object

Ranger result

Call:

ranger::ranger(formula = formula, data = data, mtry = ~3, num.trees = ~100, num.threads = 1, verbose = FALSE, seed = sample.int(10^5, 1))

Type: Regression

Number of trees: 100

Sample size: 100

Number of independent variables: 5

Mtry: 3

Target node size: 5

Variable importance mode: none

Splitrule: variance

OOB prediction error (MSE): 1.514654e-05

R squared (OOB): 0.6880896

Notice that ranger gives us an OOB prediction error (MSE) value as part of its return. parsnip returns to us what the underlying engine returns.

Now, let’s apply that random forest regression to all 1159 of our splits (recall that each split consists of 100 days of training data and one day of test data), so we can get an average RMSE. Warning: this will consume some resources on your machine and some time in your day.

To apply that model to our entire data set, we create a function that takes one split, passes it to our parsnip enabled model, and then uses the predict function to attempt to predict our assessment split. The function also allows us to specify the number of trees and the number of variables randomly sampled at each tree split, which is set with the mtry argument.

ranger\_rf\_regress <- function(mtry = 3, trees = 5, split){

analysis\_set\_rf <- analysis(split)

model <-

rand\_forest(mtry = mtry, trees = trees) %>%

set\_engine("ranger") %>%

fit(daily\_returns ~ MKT + SMB + HML + RMW + CMA, data = analysis\_set\_rf)

assessment\_set\_rf <- assessment(split)

assessment\_set\_rf %>%

select(date, daily\_returns) %>%

mutate(.pred = unlist(predict(model, new\_data = assessment\_set\_rf))) %>%

select(date, daily\_returns, .pred)

}

Now we want to pass it our object of 1159 splits, rolling\_origin\_spy\_2013\_2017$splits, and we want the function to iterate over each split. For that we turn to map\_df() from the purrr package, which allows us to iterate over the data object and return a data frame. map\_df() takes the data as an argument and our function as an argument.

ranger\_results <-

map\_df(.x = rolling\_origin\_spy\_2013\_2017$splits,

~ranger\_rf\_regress(mtry = 3, trees = 100, split = .x))

Here are the results. We now have 1159 predictions.

ranger\_results %>%

head()

# A tibble: 6 x 4

# Groups: asset [1]

asset date daily\_returns .pred

1 SPY 2013-05-28 0.00597 0.00583

2 SPY 2013-05-29 -0.00652 -0.00403

3 SPY 2013-05-30 0.00369 0.00658

4 SPY 2013-05-31 -0.0145 -0.0114

5 SPY 2013-06-03 0.00549 0.00119

6 SPY 2013-06-04 -0.00482 0.00202

Notice how the date of each prediction is included since we included it in the select() call in our function. That will come in handy for charting later.

Now, we can use the rmse() function from yardstick to calculate the root mean-squared error each of our predictions (our test sets had only one observation in them because we were testing on one month, so the RMSE is not a complex calculation here, but it would be the same code pattern if we had a larger test set). We can then find the average RMSE by calling summarise(avg\_rmse = mean(.estimate)).

library(yardstick)

ranger\_results %>%

group\_by(date) %>%

rmse(daily\_returns, .pred) %>%

summarise(avg\_rmse = mean(.estimate))

# A tibble: 1 x 1

avg\_rmse

1 0.00253

We have the average RMSE; let’s see if the RMSE were stable over time, first with ggplot.

ranger\_results %>%

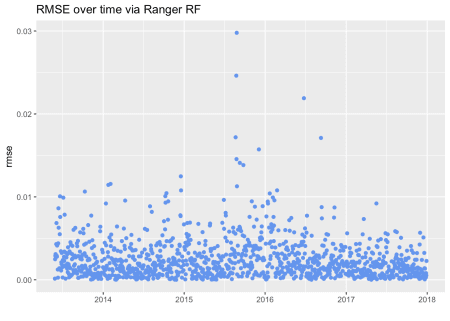
group\_by(date) %>%

rmse(daily\_returns, .pred) %>%

ggplot(aes(x = date, y = .estimate)) +

geom\_point(color = "cornflowerblue") +

labs(y = "rmse", x = "", title = "RMSE over time via Ranger RF")



And with highcharter.

ranger\_results %>%

group\_by(date) %>%

rmse(daily\_returns, .pred) %>%

hchart(., hcaes(x = date, y = .estimate),

type = "point") %>%

hc\_title(text = "RMSE over time via Ranger RF") %>%

hc\_yAxis(title = list(text = "RMSE"))

It looks like our RMSE is relatively stable, except for a period in mid to late 2015.

The amazing power of parsnip is how efficiently we can toggle to another random forest engine. Let’s suppose we wished to use the [randomForest](https://cran.r-project.org/web/packages/randomForest/randomForest.pdf) package instead of ranger. Here’s how we could reconfigure our previous work to use a different engine.

First, we’ll load up the randomForest package, because we need to load the package in order to use it as our engine. Then, we make one tweak to the original ranger\_rf\_regress function, by changing set\_engine("ranger") to set\_engine("randomForest"). That’s all, and we’re now running a random forest model using a different package.

library(randomForest)

randomForest\_rf\_regress <- function(mtry = 3, trees = 5, split){

analysis\_set\_rf <- analysis(split)

model <-

rand\_forest(mtry = mtry, trees = trees) %>%

set\_engine("randomForest") %>%

fit(daily\_returns ~ MKT + SMB + HML + RMW + CMA, data = analysis\_set\_rf)

assessment\_set\_rf <- assessment(split)

assessment\_set\_rf %>%

select(date, daily\_returns) %>%

mutate(.pred = unlist(predict(model, new\_data = assessment\_set\_rf))) %>%

select(date, daily\_returns, .pred)

}

We now have a new function called randomForest\_rf\_regress() that uses randomForest as the engine for our model and can use the same code scaffolding to run that model on our 1159 splits.

randomForest\_results <-

map\_df(.x = rolling\_origin\_spy\_2013\_2017$splits,

~randomForest\_rf\_regress(mtry = 3, trees = 100, split = .x))

randomForest\_results %>%

head()

# A tibble: 6 x 4

# Groups: asset [1]

asset date daily\_returns .pred

1 SPY 2013-05-28 0.00597 0.00609

2 SPY 2013-05-29 -0.00652 -0.00438

3 SPY 2013-05-30 0.00369 0.00597

4 SPY 2013-05-31 -0.0145 -0.00987

5 SPY 2013-06-03 0.00549 0.00134

6 SPY 2013-06-04 -0.00482 0.00118

And we can use the same yardstick code to extract the RMSE.

randomForest\_results %>%

group\_by(date) %>%

rmse(daily\_returns, .pred) %>%

summarise(avg\_rmse = mean(.estimate))

# A tibble: 1 x 1

avg\_rmse

1 0.00252

There’s a lot more to explore in the parsnip package and the tidymodels collection. See you next time when we’ll get into some classification!