Let’s get to it. Since we’ll be working with R and Python, we start with our usual R setup code chunk to load R packages, but we’ll also load the reticulate package and source a Python script. Here’s what that looks like.

library(tidyverse)

library(tidyquant)

library(riingo)

library(timetk)

library(plotly)

library(roll)

library(slider)

library(reticulate)

riingo\_set\_token("your tiingo token here")

# Python file that holds my tiingo token

reticulate::source\_python("credentials.py")

knitr::opts\_chunk$set(message = FALSE, warning = FALSE, comment = NA)

Note that I set my tiingo token twice: first using riingo\_set\_token() so I can use the riingo package in R chunks and then by sourcing the credentials.py file, where I have put tiingoToken = 'my token'. Now I can use the tiingoToken variable in my Python chunks. This is necessary because we will use both R and Python to pull in data from Tiingo.

Next we will use a Python chunk to load the necessary Python libraries. If you haven’t installed these yet, you can open the RStudio terminal and run pip install. Since we’ll be interspersing R and Python code chunks throughout, I will add a # Python Chunk to each Python chunk and, um, # R Chunk to each R chunk.

# Python chunk

import pandas as pd

import numpy as np

import tiingo

Let’s get to the substance. The goal today is look back at the last 43 years of S&P 500 price history and analyze how the market has performed following a day that sees an extreme return. We will also take care with how we define an extreme return, using rolling volatility to normalize percentage moves.

We will use the mutual fund VFINX as a tradeable proxy for the S&P 500 because it has a much longer history than other funds like SPY or VOO.

Let’s start by passing a URL string from tiingo to the pandas function read\_csv, along with our tiingoToken.

# Python chunk

pricesDF = pd.read\_csv("<https://api.tiingo.com/tiingo/daily/vfinx/prices?startDate=1976-1-1&format=csv&token=>" + tiingoToken)

We just created a Python object called pricesDF. We can look at that object in an R chunk by calling py$pricesDF.

# R chunk

py$pricesDF %>%

head()

We just created a Python object called pricesDF. Let’s reformat the date column becomes the index, in date time format.

# Python chunk

pricesDF = pricesDF.set\_index(['date'])

pricesDF.index = pd.DatetimeIndex(pricesDF.index)

Heading back to R for viewing, we see that the date column is no longer a column – it is the index of the data frame and in pandas the index is more like a label than a new column. In fact, here’s what happens when call the row names of this data frame.

# R chunk

py$pricesDF %>%

head() %>%

rownames()

We now have our prices, indexed by date. Let’s convert adjusted closing prices to log returns and save the results in a new column called returns. Note the use of the shift(1) operator here. That is analogous to the lag(..., 1) function in dplyr.

# Python chunk

pricesDF['returns'] = np.log(pricesDF['adjClose']/pricesDF['adjClose'].shift(1))

Next, we want to calculate the 3-month rolling standard deviation of these daily log returns, and then divide daily returns by the *previous* rolling 3-month volatility in order to prevent look-ahead error. We can think of this as normalizing today’s return by the previous 3-months’ rolling volatility and will label it as stdDevMove.

# Python chunk

pricesDF['rollingVol'] = pricesDF['returns'].rolling(63).std()

pricesDF['stdDevMove'] = pricesDF['returns'] / pricesDF['rollingVol'].shift(1)

Finally, we eventually want to calculate how the market has performed on the day following a large negative move. To prepare for that, let’s create a column of next day returns using shift(-1).

# Python chunk

pricesDF['nextDayReturns'] = pricesDF.returns.shift(-1)

Now, we can filter by the size of the stdDevMove column and the returns column, to isolate days where the standard deviation move was at least 3 and the returns was less than -3%. We use mean() to find the mean next day return following such large events.

# Python chunk

nextDayPerformanceSeries = pricesDF.loc[(pricesDF['stdDevMove'] < -3) & (pricesDF['returns'] < -.03), ['nextDayReturns']].mean()

Finally, let’s loop through and see how the mean next day return changes as we filter on different extreme negative returns or we can call drop tolerances. We will label the drop tolerance as i, set it at -.03 and then run a while loop that decrements down i by .0025 at each pass. In this way we can look at the mean next return following different levels of negative returns.

# Python chunk

i = -.03

while i >= -.0525:

nextDayPerformanceSeries = pricesDF.loc[(pricesDF['stdDevMove'] < -3) & (pricesDF['returns'] < i), ['nextDayReturns']]

print(str(round(i, 5)) + ': ' + str(round(nextDayPerformanceSeries['nextDayReturns'].mean(), 6)))

i -= .0025

It appears that as the size of the drop gets larger and more negative, the mean bounce back tends to get larger.

Let’s reproduce these results in R.

First, we import prices using the riingo\_prices() function from the riingo.

# R chunk

sp\_500\_prices <-

"VFINX" %>%

riingo\_prices(start\_date = "1976-01-01", end\_date = today())

We can use mutate() to add a column of daily returns, rolling volatility, standard deviation move and next day returns.

# R chunk

sp\_500\_returns <-

sp\_500\_prices %>%

select(date, adjClose) %>%

mutate(daily\_returns\_log = log(adjClose/lag(adjClose)),

rolling\_vol = roll\_sd(as.matrix(daily\_returns\_log), 63),

sd\_move = daily\_returns\_log/lag(rolling\_vol),

next\_day\_returns = lead(daily\_returns\_log))

Now let’s filter() on an sd\_move greater than 3 and daily\_returns\_log less than a drop tolerance of -.03.

# R chunk

sp\_500\_returns %>%

na.omit() %>%

filter(sd\_move < -3 & daily\_returns\_log < -.03) %>%

select(date, daily\_returns\_log, sd\_move, next\_day\_returns) %>%

summarise(mean\_return = mean(next\_day\_returns)) %>%

add\_column(drop\_tolerance = scales::percent(.03), .before = 1)

# A tibble: 1 x 2

drop\_tolerance mean\_return

1 3% 0.00625

We used a while() loop to iterate across different drop tolerances in Python, let’s see how to implement that using map\_dfr() from the purrr package.

First, we will define a sequence of drop tolerances using the seq() function.

# R chunk

drop\_tolerance <- seq(.03, .05, .0025)

drop\_tolerance

[1] 0.0300 0.0325 0.0350 0.0375 0.0400 0.0425 0.0450 0.0475 0.0500

Next, we will create a function called outlier\_mov\_fun that takes a data frame of returns, filters on a drop tolerance and gives us the mean return following large negative moves.

# R chunk

outlier\_mov\_fun <- function(drop\_tolerance, returns) {

returns %>%

na.omit() %>%

filter(sd\_move < -3 & daily\_returns\_log < -drop\_tolerance) %>%

select(date, daily\_returns\_log, sd\_move, next\_day\_returns) %>%

summarise(mean\_return = mean(next\_day\_returns) %>% round(6)) %>%

add\_column(drop\_tolerance = scales::percent(drop\_tolerance), .before = 1) %>%

add\_column(drop\_tolerance\_raw = drop\_tolerance, .before = 1)

}

Notice how that function takes two arguments: a drop tolerance and data frame of returns.

Next, we pass our sequence of drop tolerances, stored in a variable called drop\_tolerance to map\_dfr(), along with our function and our sp\_500\_returns object. map\_dfr will iterate through our sequence of drops and apply our function to each one.

# R chunk

map\_dfr(drop\_tolerance, outlier\_mov\_fun, sp\_500\_returns) %>%

select(-drop\_tolerance\_raw)

# A tibble: 9 x 2

drop\_tolerance mean\_return

1 3% 0.00625

2 3% 0.00700

3 3% 0.00967

4 4% 0.0109

5 4% 0.0122

6 4% 0.0132

7 4% 0.0149

8 5% 0.0149

9 5% 0.0162

Have a quick glance up that the results of our Python while() and we should see that the results are consistent.

Alright, let’s have some fun and get to visualizing these results with ggplot and plotly.

# R chunk

(

sp\_500\_returns %>%

map\_dfr(drop\_tolerance, outlier\_mov\_fun, .) %>%

ggplot(aes(x = drop\_tolerance\_raw, y = mean\_return, text = str\_glue("drop tolerance: {drop\_tolerance}

mean next day return: {mean\_return \* 100}%"))) +

geom\_point(color = "cornflowerblue") +

labs(title = "Mean Return after Large Daily Drop", y = "mean return", x = "daily drop") +

scale\_x\_continuous(labels = scales::percent) +

scale\_y\_continuous(labels = scales::percent) +

theme\_minimal()

) %>% ggplotly(tooltip = "text")

Here’s what happens when we expand the upper bound to a drop tolerance of -2% and make our intervals smaller, moving from .25% increments to .125% increments.

# R chunk

drop\_tolerance\_2 <- seq(.02, .05, .00125)

(

sp\_500\_returns %>%

map\_dfr(drop\_tolerance\_2, outlier\_mov\_fun, .) %>%

ggplot(aes(x = drop\_tolerance\_raw, y = mean\_return, text = str\_glue("drop tolerance: {drop\_tolerance}

mean next day return: {mean\_return \* 100}%"))) +

geom\_point(color = "cornflowerblue") +

labs(title = "Mean Return after Large Daily Drop", y = "mean return", x = "daily drop") +

scale\_x\_continuous(labels = scales::percent) +

scale\_y\_continuous(labels = scales::percent) +

theme\_minimal()

) %>% ggplotly(tooltip = "text")

Check out what happens when we expand the lower bound, to a -6% drop tolerance.

# R chunk

drop\_tolerance\_3 <- seq(.02, .06, .00125)

(

sp\_500\_returns %>%

map\_dfr(drop\_tolerance\_3, outlier\_mov\_fun, .) %>%

ggplot(aes(x = drop\_tolerance\_raw, y = mean\_return, text = str\_glue("drop tolerance: {drop\_tolerance}

mean next day return: {mean\_return \* 100}%"))) +

geom\_point(color = "cornflowerblue") +

labs(title = "Mean Return after Large Daily Drop", y = "mean return", x = "daily drop") +

scale\_x\_continuous(labels = scales::percent) +

scale\_y\_continuous(labels = scales::percent) +

theme\_minimal()

) %>% ggplotly(tooltip = "text")

I did not expect that gap upward when the daily drop passes 5.25%.

A quick addendum that if I had gotten my act together and finished this 4 days ago I would not have included, but I’m curious how this last week has compared with other weeks in terms of volatility. I have in mind to visualize weekly return dispersion and that seemed a mighty tall task, until the brand new slider package came to the rescue! slider has a function called slide\_period() that, among other things, allows us to break up time series according to different periodicities.

To break up our returns by week, we call slide\_period\_dfr(., .$date, "week", ~ .x, .origin = first\_monday\_december, .names\_to = "week"), where first\_monday\_december is a date that falls on a Monday. We could use our eyeballs to check a calendar and find a date that’s a Monday or we could use some good ol’ code. Let’s assume we want to find the first Monday in December of 2016.

We first filter our data with filter(between(date, as\_date("2016-12-01"), as\_date("2016-12-31"))). Then create a column of weekday names with wday(date, label = TRUE, abbr = FALSE) and filter to our first value of “Monday”.

# R Chunk

first\_monday\_december <-

sp\_500\_returns %>%

mutate(date = ymd(date)) %>%

filter(between(date, as\_date("2016-12-01"), as\_date("2016-12-31"))) %>%

mutate(day\_week = wday(date, label = TRUE, abbr = FALSE)) %>%

filter(day\_week == "Monday") %>%

slice(1) %>%

pull(date)

Now we run our slide\_period\_dfr() code and it will start on the first Monday in December of 2016, and break our returns into weeks. Since we set .names\_to = "week", the function will create a new column called week and give a unique number to each of our weeks.

# R chunk

sp\_500\_returns %>%

select(date, daily\_returns\_log) %>%

filter(date >= first\_monday\_december) %>%

slide\_period\_dfr(.,

.$date,

"week",

~ .x,

.origin = first\_monday\_december,

.names\_to = "week") %>%

head(10)

# A tibble: 10 x 3

week date daily\_returns\_log

1 1 2016-12-05 00:00:00 0.00589

2 1 2016-12-06 00:00:00 0.00342

3 1 2016-12-07 00:00:00 0.0133

4 1 2016-12-08 00:00:00 0.00226

5 1 2016-12-09 00:00:00 0.00589

6 2 2016-12-12 00:00:00 -0.00105

7 2 2016-12-13 00:00:00 0.00667

8 2 2016-12-14 00:00:00 -0.00810

9 2 2016-12-15 00:00:00 0.00392

10 2 2016-12-16 00:00:00 -0.00172

From here, we can group\_by that week column and treat each week as a discrete time period. Let’s use ggplotly to plot each week on the x-axis and the daily returns of each week on the y-axis, so that the vertical dispersion shows us the dispersion of weekly returns. Hover on the point to see the exact date of the return.

# R chunk

(

sp\_500\_returns %>%

select(date, daily\_returns\_log) %>%

filter(date >= first\_monday\_december) %>%

slide\_period\_dfr(.,

.$date,

"week",

~ .x,

.origin = first\_monday\_december,

.names\_to = "week") %>%

group\_by(week) %>%

mutate(start\_week = ymd(min(date))) %>%

ggplot(aes(x = start\_week, y = daily\_returns\_log, text = str\_glue("date: {date}"))) +

geom\_point(color = "cornflowerblue", alpha = .5) +

scale\_y\_continuous(labels = scales::percent,

breaks = scales::pretty\_breaks(n = 8)) +

scale\_x\_date(breaks = scales::pretty\_breaks(n = 10)) +

labs(y = "", x = "", title = "Weekly Daily Returns") +

theme\_minimal()

) %>% ggplotly(tooltip = "text")

We can also plot the standard deviation of returns for each week.

# R chunk

(

sp\_500\_returns %>%

select(date, daily\_returns\_log) %>%

filter(date >= first\_monday\_december) %>%

slide\_period\_dfr(.,

.$date,

"week",

~ .x,

.origin = first\_monday\_december,

.names\_to = "week") %>%

group\_by(week) %>%

summarise(first\_of\_week = first(date),

sd = sd(daily\_returns\_log)) %>%

ggplot(aes(x = first\_of\_week, y = sd, text = str\_glue("week: {first\_of\_week}"))) +

geom\_point(aes(color = sd)) +

labs(x = "", title = "Weekly Standard Dev of Returns", y = "") +

theme\_minimal()

) %>% ggplotly(tooltip = "text")

That’s all for today! Thanks for reading and stay safe out there.