Vol-Chapter

Volatility

```
#webshot::install_phantomjs()
library(tidyquant)
library(tidyverse)
library(timetk)
library(tibbletime)
library(highcharter)

knitr::opts_chunk$set(message=FALSE, warning=FALSE)

load("~/reproducible-fin-chapters/book-data.Rdata")
```

Welcome to our chapter focused on portfolio volatility, variance and standard deviation. I realize that it's a lot more fun to ~~fantasize about~~ analyze stock returns which is why television shows and websites constantly update the daily market returns and give them snazzy green and red colors. But good ol'volatility is quite important in its own right, especially to finance geeks, aspiring finance geeks and institutional investors. If you are, might become, or might ever work with/for any of those, this chapter should at least serve as a jumping off point.

A quick word of warning that this chapter begins at the beginning with portfolio standard deviation and builds up to more complex work. R users with experience in the world of volatility may wish to skip a few sections or head straight to the Shiny sections. That said, I would humbly offer a couple of benefits to the R code that awaits us.

First, volatility is important, possibly more important than returns. I don't think any investment professional looks back on hours spent pondering volatility as a waste of time.

Second, as always, we have an eye on making our work reproducible and reusable. We'll make it exceedingly clear how we derive our final data visualizations on portfolio volatility. It's a good template for other visualization derivations, even if standard deviation is old hat for you.

Introduction to Volatility

We will start with the textbook equation for the standard deviation of a multi-asset portfolio.

Here is the equation for sd:

$$Standard\ Deviation = \sqrt{\sum_{t=1}^{n}(x_i - \overline{x})^2/n}$$

- First, we assign the weights of each asset.
- Then, we isolate and assign returns of each asset.
- Next, we plug those weights and returns into the equation for portfolio standard deviation, which involves the following:
- Take the weight squared of each asset times it's variance and sum those weighted variance terms.
- Then we take the covariance of each asset pair, multiplied by 2 times the weight of the first asset times the weight of the second asset.
- Sum together the covariance terms and the weighted variance terms.
- This gives us the portfolio variance.

- Then take the square root to get the standard deviation.

```
w_1 <- w[1]
w_2 <- w[2]
w_3 <- w[3]
w_4 <- w[4]
w_5 <- w[5]

asset1 <- asset_returns_xts[,1]
asset2 <- asset_returns_xts[,2]
asset3 <- asset_returns_xts[,3]
asset4 <- asset_returns_xts[,4]
asset5 <- asset_returns_xts[,5]


sd_by_hand <-
  # Important, don't forget to take the square root!
  sqrt(
  # Our weighted variance terms.
  (w_1^2 * var(asset1)) + (w_2^2 * var(asset2)) + (w_3^2 * var(asset3)) +
  (w_4^2 * var(asset4)) + (w_5^2 * var(asset5)) +
  # Our weighted covariance terms
  (2 * w_1 * w_2 * cov(asset1, asset2)) +
  (2 * w_1 * w_3 * cov(asset1, asset3)) +
  (2 * w_1 * w_4 * cov(asset1, asset4)) +
  (2 * w_1 * w_5 * cov(asset1, asset5)) +
  (2 * w_2 * w_3 * cov(asset2, asset3)) +
  (2 * w_2 * w_4 * cov(asset2, asset4)) +
  (2 * w_2 * w_5 * cov(asset2, asset5)) +
  (2 * w_3 * w_4 * cov(asset3, asset4)) +
  (2 * w_3 * w_5 * cov(asset3, asset5)) +
  (2 * w_4 * w_5 * cov(asset4, asset5))
  )

# I want to print the percentage, so multiply by 100.
sd_by_hand_percent <- round(sd_by_hand * 100, 2)
```

Writing that equation out was painful but at least we won't be forgetting it any time soon. Our result is a monthly portfolio returns standard deviation of 2.67%.

Now let's turn to the less verbose matrix algebra path and confirm that we get the same result.

First, we will build a covariance matrix of returns using the `cov()` function.

```
# Build the covariance matrix.
covariance_matrix <- cov(asset_returns_xts)
covariance_matrix
```

```
##               SPY          EFA           IJS          EEM           AGG
## SPY  7.293930e-04 6.899253e-04  8.185514e-04 0.0006970811 -3.959920e-06
## EFA  6.899253e-04 1.066615e-03  6.443392e-04 0.0010575342  4.640306e-05
## IJS  8.185514e-04 6.443392e-04  1.568491e-03 0.0006717294 -7.299985e-05
## EEM  6.970811e-04 1.057534e-03  6.717294e-04 0.0017682417  1.037519e-04
## AGG -3.959920e-06 4.640306e-05 -7.299985e-05 0.0001037519  7.400944e-05
```

Have a look at the covariance matrix.

AGG, the US bond ETF, has a negative or very low covariance with the other ETFs and it should make a nice volatility dampener. Interestingly, the covariance between EEM and EFA is quite low as well. Our painstakingly written-out equation above is a good reminder of how low covariances affect total portfolio standard deviation.

Back to our calculation, let's take the square root of the transpose of the weights vector times the covariance matrix times the weights vector. To perform matrix multiplcation, we use `%*%`.

```
sd_matrix_algebra <- sqrt(t(w) %*% covariance_matrix %*% w)

sd_matrix_algebra_percent <- round(sd_matrix_algebra * 100, 2)

sd_by_hand_percent
```

```
##      SPY
## SPY 2.67
```

```
sd_matrix_algebra_percent
```

```
##      [,1]
## [1,] 2.67
```

Have a look at our two standard deviation calculations Thankfully, these return the same result so we don't have to sort through the by-hand equation again.

And, finally, we can use the built-in `StdDev()` function from the `PerformanceAnalytics` package. It takes two arguments, returns and weights: `StdDev(asset_returns_xts, weights = w)`.

```
portfolio_sd_xts_builtin <- StdDev(asset_returns_xts, weights = w)


portfolio_sd_xts_builtin_percent <- round(portfolio_sd_xts_builtin * 100, 2)
```

We now have:

```
sd_by_hand_percent
```

```
##      SPY
## SPY 2.67
```

```
sd_matrix_algebra_percent
```

```
##      [,1]
## [1,] 2.67
```

```
portfolio_sd_xts_builtin_percent
```

```
##      [,1]
## [1,] 2.67
```

Now let's head to the tidyverse and explore the code flow. We will start with the portfolio returns object `portfolio_returns_tq_rebalanced_monthly` and we want to calculate the lifetime standard deviation. We use the `summarise()` function from `dplyr` and then the base function `sd()`. We will also perform the calculation with our own equation `sqrt(sum((returns - mean(returns))^2)/(nrow(.)-1)))`.

```
portfolio_sd_tidy_builtin_percent <-
  portfolio_returns_tq_rebalanced_monthly %>%
  summarise(sd = sd(returns),
            sd_byhand = sqrt(sum((returns - mean(returns))^2)/(nrow(.)-1))) %>%
  mutate(sd = round(sd, 4) * 100,
         sd_byhand = round(sd_byhand, 4) * 100)  %>%
```

```
  select(sd, sd_byhand)

portfolio_sd_tidy_builtin_percent
```

```
## # A tibble: 1 x 2
##      sd sd_byhand
##   <dbl>     <dbl>
## 1  2.67      2.67
```

Here are our standard deviation calculations thus far:

- by-hand calculation = 2.67%
- matrix algebra calculation = 2.67%
- xts built in function calculation = 2.67%
- tidy built in in function calculation = 2.67%
- tidy by hand calculation = 2.67%

That was quite a lot of work to confirm that 5 calculations are equal to each other but there are a few benefits.

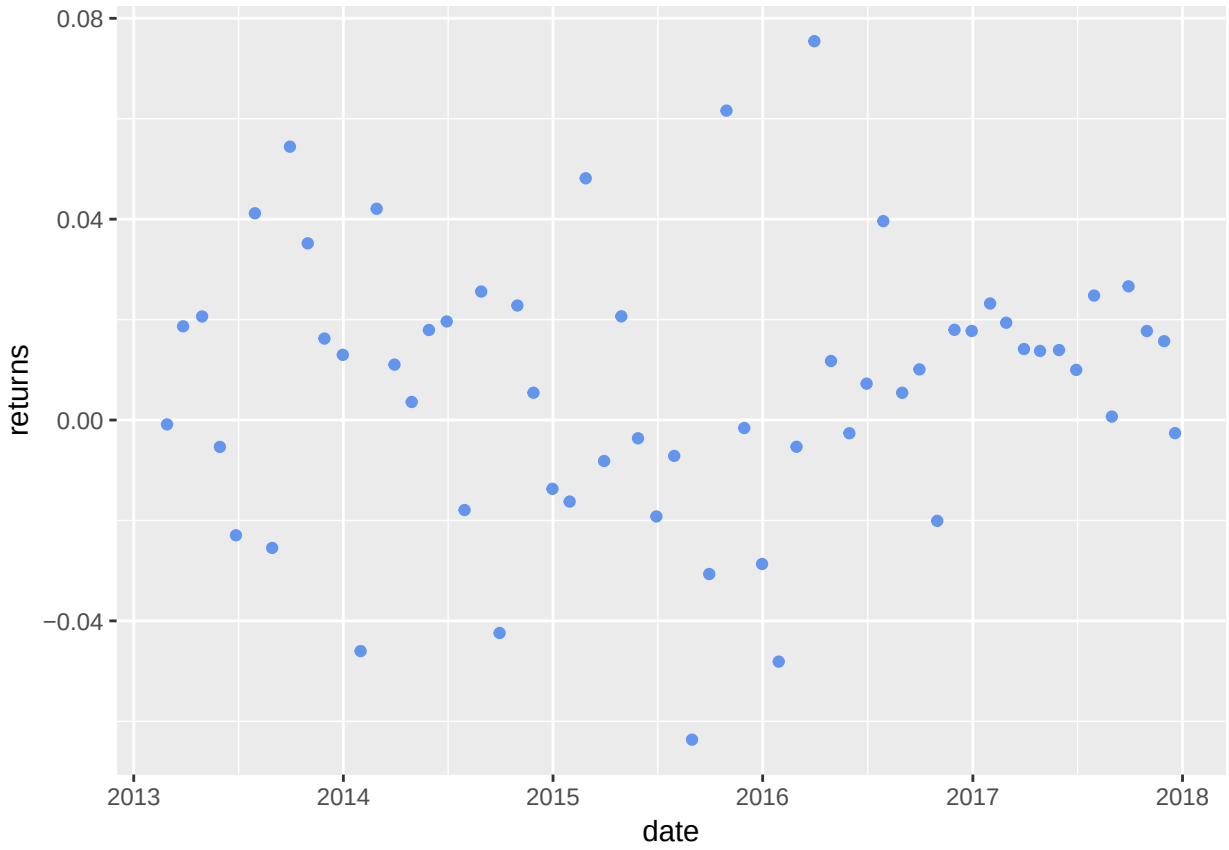
First, while it was tedious, we should feel comfortable with calculating portfolio standard deviations in various ways and starting from different object types.

More importantly, as our work gets more complicated and we build custom functions, we'll want to rely on the built-in `StdDev` function and the built-in tidy workflow. Our tedious work here, where we used different flows and functions by choice, will facilitate our future work when we need to solve harder coding challenges.

Beyond helping our own work, it can help our colleagues to understand the more complex work if we can point back to this starter code flow on standard deviation.

Visualize

```
library(scales)
portfolio_returns_tq_rebalanced_monthly %>%

  ggplot(aes(x = date, y = returns)) +

  geom_point(color = "cornflowerblue") +
  scale_x_date(breaks = pretty_breaks(n = 8))
```



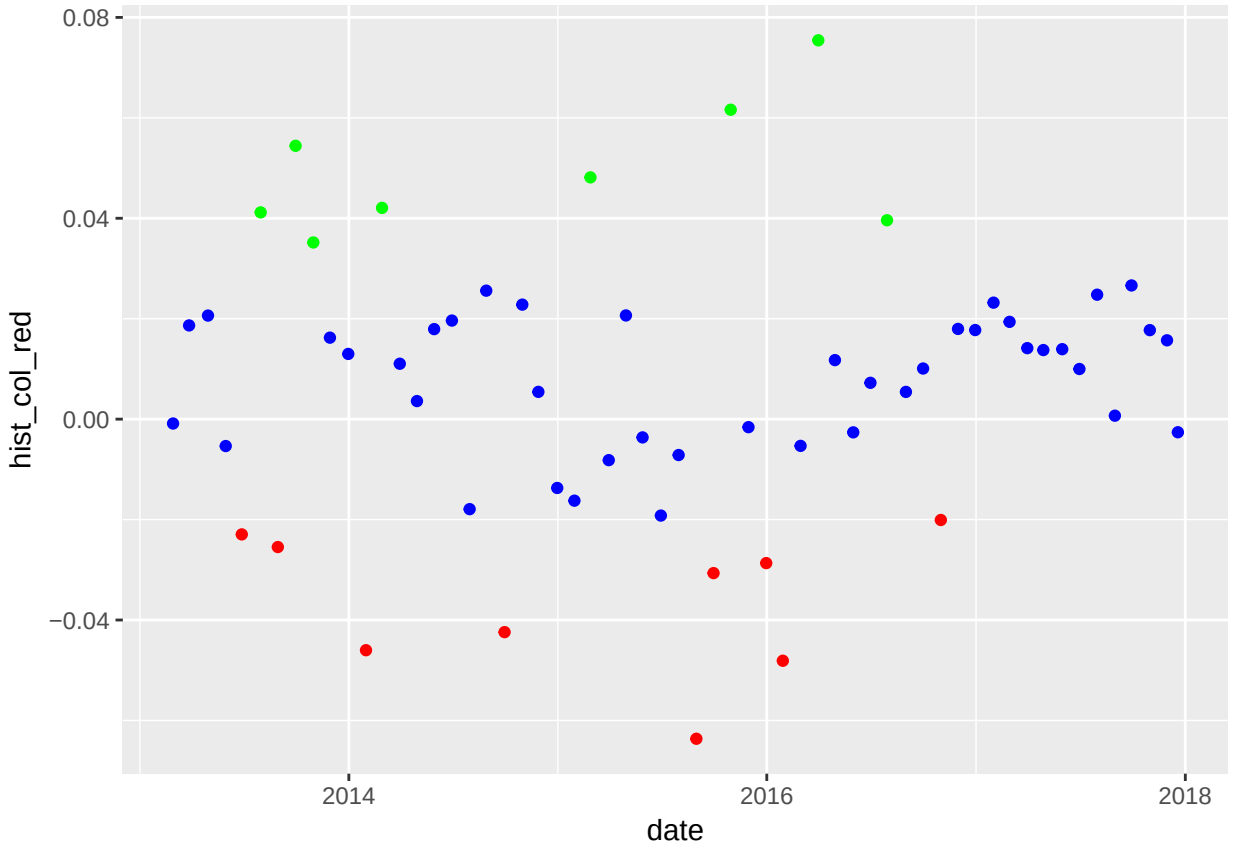
Hard to pick up much here though we can notice that the year 2017 had consistently positive monthly returns. Let's add some more indicators. It might be nice to have a different color for any monthly returns are, say, one standard deviation away from the mean. First, we will create an indicator for the mean return with `sd(portfolio_returns_tq_rebalanced_monthly$returns)` and one for the standard deviation with `mean(portfolio_returns_tq_rebalanced_monthly$returns)`. We will call the variables `mean_plot` and `sd_plot`.

```
sd_plot <- sd(portfolio_returns_tq_rebalanced_monthly$returns)
mean_plot <- mean(portfolio_returns_tq_rebalanced_monthly$returns)
```

We want to shade the scatter points according to some logic, in this case their distance from the standard deviation of returns. Accordingly, we'll use `mutate()` to create three columns of returns...

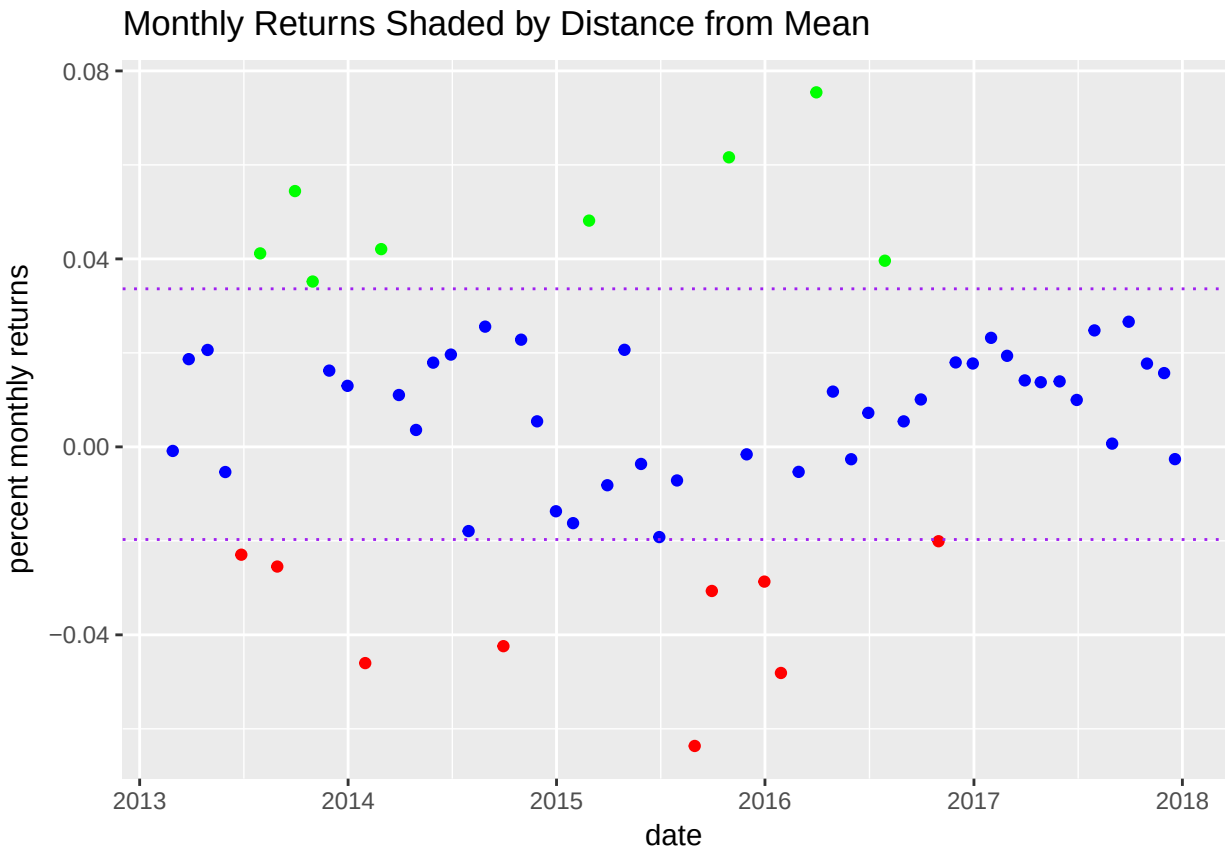
```
portfolio_returns_tq_rebalanced_monthly %>%
  mutate(hist_col_red =
           ifelse(returns < (mean_plot - sd_plot),
                  returns, NA),
         hist_col_green =
           ifelse(returns > (mean_plot + sd_plot),
                  returns, NA),
         hist_col_blue =
           ifelse(returns > (mean_plot - sd_plot) &
                  returns < (mean_plot + sd_plot),
                  returns, NA)) %>%

  ggplot(aes(x = date)) +

  geom_point(aes(y = hist_col_red),
```

```
              color = "red") +

  geom_point(aes(y = hist_col_green),
              color = "green") +

  geom_point(aes(y = hist_col_blue),
              color = "blue")
```




```
portfolio_returns_tq_rebalanced_monthly %>%
  mutate(hist_col_red =
           ifelse(returns < (mean_plot - sd_plot),
                  returns, NA),
         hist_col_green =
           ifelse(returns > (mean_plot + sd_plot),
                  returns, NA),
         hist_col_blue =
           ifelse(returns > (mean_plot - sd_plot) &
                  returns < (mean_plot + sd_plot),
                  returns, NA)) %>%

  ggplot(aes(x = date)) +

  geom_point(aes(y = hist_col_red),
              color = "red") +

  geom_point(aes(y = hist_col_green),
              color = "green") +
```

```
  geom_point(aes(y = hist_col_blue),
             color = "blue") +

  geom_hline(yintercept = (mean_plot + sd_plot), color = "purple", linetype = "dotted") +
  geom_hline(yintercept = (mean_plot-sd_plot), color = "purple", linetype = "dotted") +
  scale_x_date(breaks = pretty_breaks(n = 8)) +
ylab("percent monthly returns") +
  ggtitle("Monthly Returns Shaded by Distance from Mean")
```



This is showing us returns over time and whether they fall below or above one standard deviation from the mean. One thing that jumps out is how many red or green circles we see after 2017. 0! That's zero monthly returns that are least one standard deviation from the mean during calendar year 2017 (Is 2017 a year of very low volatility or did we construct an amazingly stable porftolio? Check out the VIX during 2017!). When we get to rolling volatility, we should see this reflected as a low rolling volatility through 2017. If we something different, we need to investigate.

Rolling Volatility

We have already calculated the volatility for the entire life of the portfolio but that's not quite adequate to understand the volatility for this collection of assets.

We might miss a 3-month or 6-month period where the volatility spiked or plummeted or did both. And the longer our portfolio life, the more likely we are to miss something important. If we had 10 or 20 years of data and we calculated the standard deviation for the entire historyy, we could, or most certainly would, fail to notice a period in which volatility was very high, and hence we would fail to ponder the probability that it could occur again.

Imagine a portfolio which had a standard deviation of returns for each 6-month period of 3% and it never changed. Now imagine a portfolio whose volatility fluctuated every few 6-month periods from 0% to 6% . We might find a 3% standard deviation of monthly returns over a 10-year sample for both of these, but those two portfolios are not exhibiting the same volatility. The rolling volatility of each would show us the differences and then we could hypothesize about the past causes and future probabilities for those differences. We might also want to think about dynamically rebalancing our portfolio to better manage volatility if we are seeing large spikes in the rolling windows.

Our least difficult task is calculating the rolling standard deviation of portfolio returns.

In the `xts` world, we use a direct call to `rollapply` for this and need to choose a number of months for the rolling window.

```
window <- 6

port_rolling_sd_xts <- rollapply(portfolio_returns_xts_rebalanced_monthly,
                                 FUN = sd,
                                 width = window) %>% na.omit()
```

The `tidyquant` packages has a nice way to apply the rolling function to data frames. We use `tq_mutate` and supply `mutate_fun = rollapply` as our mutation function argument, then `FUN = sd` as the nested function beneath it.

```
port_rolling_sd_tidy <-
  portfolio_returns_tq_rebalanced_monthly %>%
  tq_mutate(mutate_fun = rollapply,
            width = window,
            FUN = sd,
            col_rename = ("rolling_sd")) %>%
  select(date, rolling_sd) %>%
  na.omit()
```

Take a quick peak to confirm consistent results.

```
head(port_rolling_sd_xts)
```

```
##               returns
## 2013-07-31 0.02274361
## 2013-08-30 0.02666069
## 2013-09-30 0.03357664
## 2013-10-31 0.03496031
## 2013-11-29 0.03380581
## 2013-12-31 0.02814225
```

```
head(port_rolling_sd_tidy)
```

```
## # A tibble: 6 x 2
##         date rolling_sd
##       <date>      <dbl>
## 1 2013-07-31 0.02274361
## 2 2013-08-30 0.02666069
## 3 2013-09-30 0.03357664
## 4 2013-10-31 0.03496031
## 5 2013-11-29 0.03380581
## 6 2013-12-31 0.02814225
```

We now have an `xts` object called `port_rolling_sd_xts` and a tibble object called `port_rolling_sd_tidy`. They contain the 6-month rolling standard deviations of portfolio.

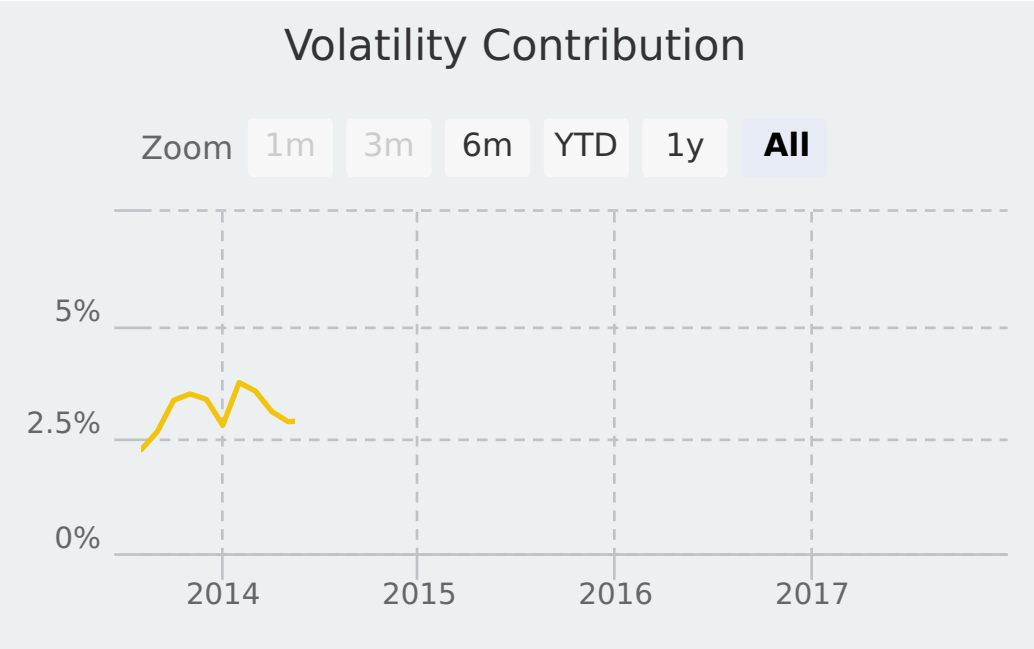
At the outset of this section, we opined that rolling volatilty might some insight that is obscured by the total volatility. Visualizing the rolling standard deviation should help to illuminate this.

Let's start with the `xts` object and `highcharter`.

First, we will convert to our data to rounded percentages for ease of charting.

```
port_rolling_sd_xts_hc <- round(port_rolling_sd_xts, 4) * 100
```

```
highchart(type = "stock") %>%
    hc_title(text = "Volatility Contribution") %>%
    hc_add_series(port_rolling_sd_xts_hc) %>%
    hc_add_theme(hc_theme_flat()) %>%
    hc_yAxis(
      labels = list(format = "{value}%"),
             opposite = FALSE) %>%
    hc_navigator(enabled = FALSE) %>%
    hc_scrollbar(enabled = FALSE)
```



Maybe we should add a flag to highlight this event. We can also add flags for the maximum SPY volatility, maximum and minimum portfolio rolling volatility and might as well include a line for the mean rolling volatility of SPY to practice adding horizontal lines.

We will use two methods for adding flags. First, we'll hard code the date for the flag as "2016-04-29", which is the date when rolling SPY volatility dipped below the portfolio.

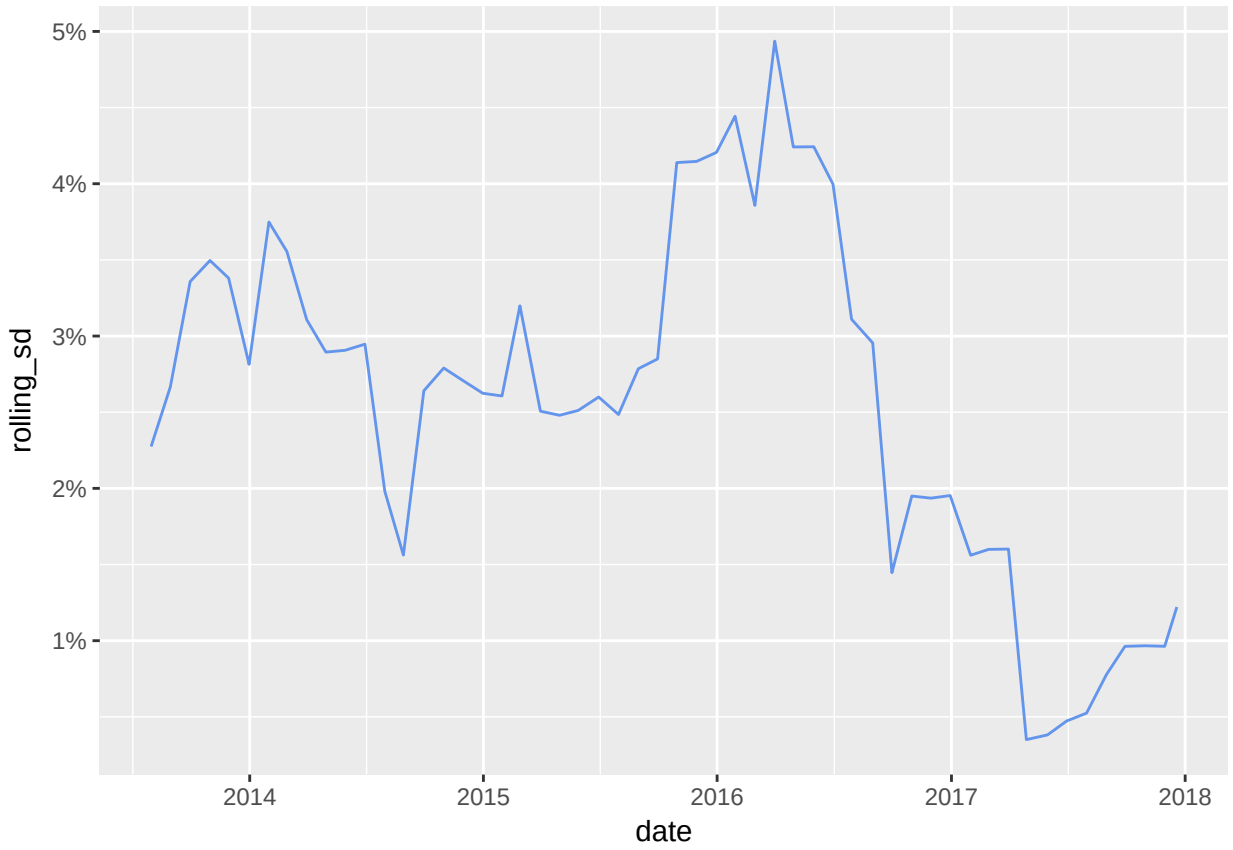
Second, we'll set a flag with the date `as.Date(index(port_rolling_sd_xts_hc[which.max(port_rolling_sd_xts_hc)]),format = "%Y-%m-%d")` which looks like a convoluted mess but is adding a date for whenever the rolling portfolio standard deviation hit its maximum.

```
port_max_date <- as.Date(index(port_rolling_sd_xts_hc[which.max(port_rolling_sd_xts_hc)]),
                         format = "%Y-%m-%d")
port_min_date <- as.Date(index(port_rolling_sd_xts_hc[which.min(port_rolling_sd_xts_hc)]),
                         format = "%Y-%m-%d")
```

```
highchart(type = "stock") %>%
  hc_title(text = "Portfolio Rolling Volatility") %>%
  hc_yAxis(title = list(text = "Volatility"),
           labels = list(format = "{value}%"),
           opposite = FALSE) %>%
  hc_add_series(port_rolling_sd_xts_hc,
                name = "Portf Volatility",
                color = "cornflowerblue",
                id = "Port") %>%
   hc_add_series_flags(port_max_date,
                       title = c("Max Rolling Vol"),
                       text = c("maximum rolling volatility."),
                       id = "Port") %>%
  hc_add_series_flags(port_min_date,
                       title = c("Min Rolling Vol"),
                       text = c("min rolling volatility."),
                       id = "Port") %>%
  hc_navigator(enabled = FALSE) %>%
  hc_scrollbar(enabled = FALSE)
```

Let's do the same with our data frame and ggplot.

```
port_rolling_sd_tidy %>%
  ggplot(aes(x = date)) +
  geom_line(aes(y = rolling_sd), color = "cornflowerblue") +
  scale_y_continuous(labels = scales::percent) +
  scale_x_date(breaks = pretty_breaks(n = 8))
```



Note that we didn't do any work to change our decimal to percentage format. When we called `scale_y_continuous(labels = scales::percent)` it did that work for us by adding the % sign and multiplying by 100.

Do these visualizations add to our understanding of this portfolio? Well, we can see a spike in rolling volatility in early 2016 followed by a consistently falling vol through to mid 2017.

It's remarkable how rolling volatility has absolutely plunged since early-to-mid 2016.

Shiny App

Now let's wrap all of that work into a Shiny app that allows a user to choose a 5-asset portfolio and chart rollling volatility of different widths.

The app is availalbe online here:

www.reproduciblefinance.com/shiny/portfolio-volatility-shiny-app/

Our input sidebar is almost identical to the previous apps on portfolio returns and dollar growth. For the full writeup on the input sidebar, see this section.

One difference is we let the user choose a rolling window with a `numerInput()`. The code chunk below is part of our sidebar.

```
fluidRow(
  column(5,
  numericInput("window", "Window", 12, min = 3, max = 36, step = 1))
)
```

Next we lean on our previous work to calculate portfolio returns and save them as `portfolio_returns_tq_rebalanced_monthly`. Then we pass that object on to a `dplyr` piped flow to calculate rolling standard deviation.

```
port_rolling_sd_tidy <-
  portfolio_returns_tq_rebalanced_monthly %>%
  tq_mutate(mutate_fun = rollapply,
            width = window,
            FUN = sd,
            col_rename = ("rolling_sd")) %>%
  select(date, rolling_sd) %>%
  na.omit()
```

We now have the object `port_rolling_sd_tidy` available for downstream visualizations. We will start in with highcharter and that requires changing to an `xts` object first.

```
renderHighchart({

  port_rolling_sd_tidy_hc <-
    port_rolling_sd_tidy() %>%
    tk_xts(date_col = date) %>%
    round(., 4) * 100


  highchart(type = "stock") %>%
    hc_title(text = "Portfolio Rolling Volatility") %>%
    hc_yAxis(title = list(text = "Volatility"),
           labels = list(format = "{value}%"),
           opposite = FALSE) %>%
    hc_add_series(port_rolling_sd_tidy_hc,
                  name = "Portfolio Vol",
                  color = "cornflowerblue",
                  id = "Port") %>%
    hc_add_theme(hc_theme_flat()) %>%
    hc_navigator(enabled = FALSE) %>%
    hc_scrollbar(enabled = FALSE)
```

Let's also add a flag for the min and max rolling volatility. We create the flags with the following:

```
  port_max_date <- as.Date(index(port_rolling_sd_tidy_hc[which.max(port_rolling_sd_tidy_hc)]),
                           format = "%Y-%m-%d")
  port_min_date <- as.Date(index(port_rolling_sd_tidy_hc[which.min(port_rolling_sd_tidy_hc)]),
                           format = "%Y-%m-%d")
```

Then add to the chart with the two `hc_add_series_flags()` function calls below:

```
   hc_add_series_flags(port_max_date,
                       title = c("Max Rolling Vol"),
                       text = c("maximum rolling volatility."),
                       id = "Port") %>%
  hc_add_series_flags(port_min_date,
                       title = c("Min Rolling Vol"),
                       text = c("min rolling volatility."),
                       id = "Port")
```

Now on to `ggplot()`. We don't need to alter the original `port_rolling_sd_tidy` reactive and can pass it straight into a piped `dplyr/ggplot` flow.

```
renderPlot({
  port_rolling_sd_tidy() %>%
    ggplot(aes(x = date)) +
    geom_line(aes(y = rolling_sd), color = "cornflowerblue") +
    scale_y_continuous(labels = scales::percent) +
    ggtitle("Portfolio Rolling Vol") +
    ylab("volatility") +
    scale_x_date(breaks = pretty_breaks(n = 8)) +
    theme(plot.title = element_text(hjust = 0.5))
})
```

Component Contribution

Now we want to break that total portfolio volatility into its constituent parts and investigate how each asset contributes to the volatility. Why might we want to do that?

For our own risk management purposes, we might want to ensure that our risk hasn't got too concentrated in one asset. Not only might this lead a less diversified portfolio than we thought we had, but it also might indicate that our initial assumptions about a particular asset were wrong, or at least, they have become less right as the asset has changed over time.

Similarly, if this portfolio is governed by a mandate from, say, an institutional client, that client might have a preference or even a rule that no asset or sector can rise above a certain threshold risk contribution. That institutional client might require a report like this from each of their outsourced managers, so they can sum the constituents.

As in the previous section on volatility, we need to build the covariance matrix and calculate portfolio standard deviation.

```
covariance_matrix <- cov(asset_returns_xts)

# Square root of transpose of the weights cross prod covariance matrix returns
# cross prod weights gives portfolio standard deviation.
sd_portfolio <- sqrt(t(w) %*% covariance_matrix %*% w)
```

Now let's start to look at the individual components.

The percentage contribution of asset i is defined as: (marginal contribution of asset i * weight of asset i) / portfolio standard deviation

Let's find the marginal contribution first. To do so, take the cross product of the weights vector and the covariance matrix divided by the portfolio standard deviation.

```
# Marginal contribution of each asset.
marginal_contribution <- w %*% covariance_matrix / sd_portfolio[1, 1]
```

Now multiply marginal contribution of each asset by the weights vector to get total contribution. We can then sum the asset contributions and make sure it's equal to total portfolio standard deviation.

```
# Component contributions to risk are the weighted marginal contributions
component_contribution <- marginal_contribution * w

# This should equal total portfolio vol, or the object `sd_portfolio`
components_summed <- rowSums(component_contribution)
```

The summed components are 0.0266747 and the matrix calculation is 0.0266747.

To get to percentage contribution of each asset, we divide each asset's contribution by total portfolio standard deviation.

```
# To get the percentage contribution, divide component contribution by total sd.
component_percentages <- component_contribution / sd_portfolio[1, 1]
```

Let's port this to a tibble for ease of presentation, and we'll append `by_hand` to the object because we did the calculations step-by-step.

```
percentage_tibble_by_hand <-
  tibble(symbols, w, as.vector(component_percentages)) %>%
  rename(asset = symbols, 'portfolio weight' = w, 'risk contribution' = `as.vector(component_percentages`

percentage_tibble_by_hand
```

```
## # A tibble: 5 x 3
##   asset `portfolio weight` `risk contribution`
##   <chr>              <dbl>               <dbl>
## 1   SPY               0.25         0.231033557
## 2   EFA               0.25         0.275510935
## 3   IJS               0.20         0.226681481
## 4   EEM               0.20         0.263378281
## 5   AGG               0.10         0.003395745
```

As you might have guessed, we used `by_hand` in the object name because we could have used a pre-built R function to do all this work. The `StdDev` function from PerformanceAnalytics will run this same calculation if we pass in the weights and set `portfolio_method = "component"` (recall that if we set `portfolio_method = "single"`, the function will return the total portfolio standard deviation, as we saw in a previous section).

Let's confirm that the pre-built function returns the same results.

```
# Confirm component contribution to volality.
component_sd_pre_built <- StdDev(asset_returns_xts, weights = w,
                                 portfolio_method = "component")
component_sd_pre_built
```

```
## $StdDev
##            [,1]
## [1,] 0.02667474
##
## $contribution
##          SPY          EFA          IJS          EEM          AGG
## 6.162760e-03 7.349183e-03 6.046670e-03 7.025547e-03 9.058063e-05
##
## $pct_contrib_StdDev
##         SPY         EFA         IJS         EEM         AGG
## 0.231033557 0.275510935 0.226681481 0.263378281 0.003395745
```

That function returns a list and one of the elements is `$pct_contrib_StdDev`, which is the percentage contribution of each asset. Let's move it to a `tibble` for ease of presentation.

```
# Port to a tibble.
percentages_tibble_pre_built <-
  component_sd_pre_built$pct_contrib_StdDev %>%
  tk_tbl(preserve_index = FALSE) %>%
  mutate(asset = symbols) %>%
  rename('risk contribution' = data) %>%
  select(asset, everything())
```

Has our work checked out? Is `percentages_tibble_pre_built` showing the same result as `component_percentages_tibble_by_hand`?

Compare the two objects

```
percentages_tibble_pre_built
```

```
## # A tibble: 5 x 2
##   asset `risk contribution`
##   <chr>               <dbl>
## 1   SPY         0.231033557
## 2   EFA         0.275510935
## 3   IJS         0.226681481
## 4   EEM         0.263378281
## 5   AGG         0.003395745
```

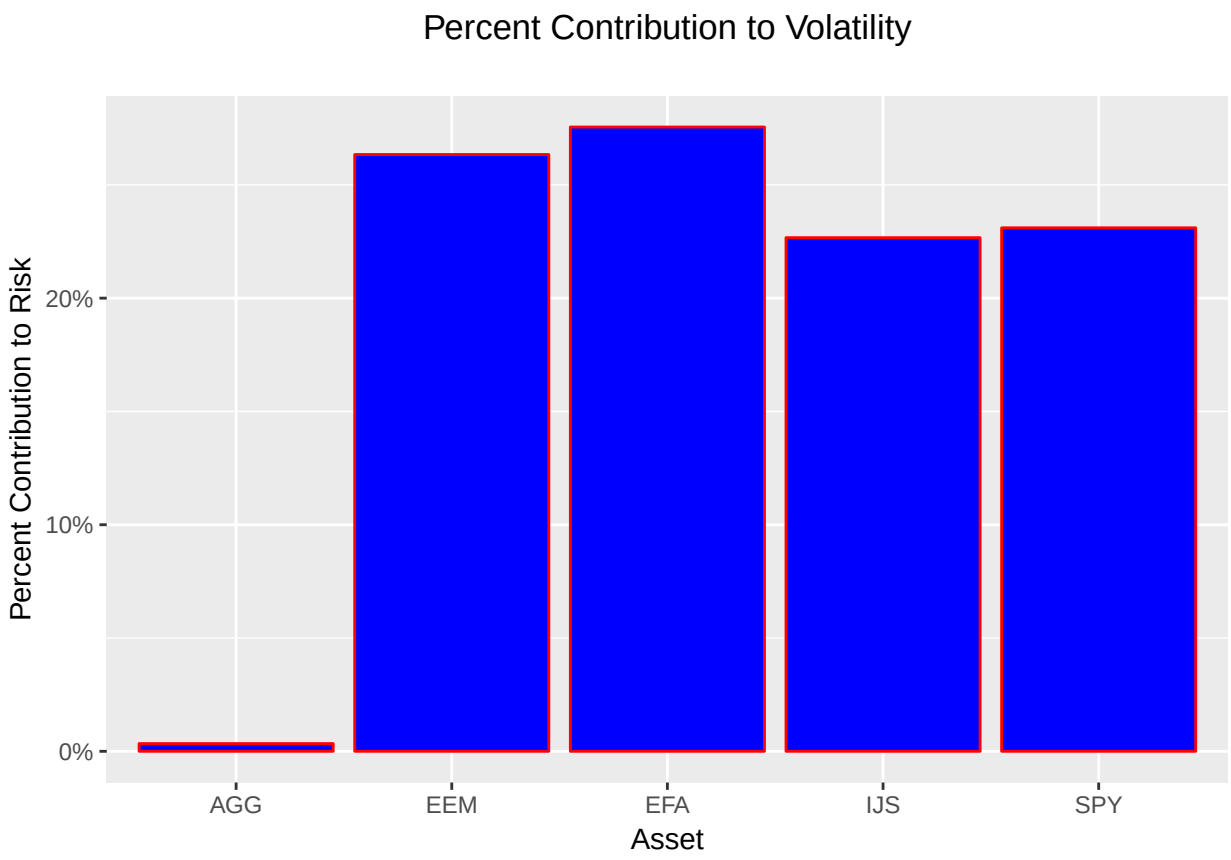
```
percentage_tibble_by_hand
```

```
## # A tibble: 5 x 3
##   asset `portfolio weight` `risk contribution`
##   <chr>              <dbl>               <dbl>
## 1   SPY               0.25         0.231033557
## 2   EFA               0.25         0.275510935
## 3   IJS               0.20         0.226681481
## 4   EEM               0.20         0.263378281
## 5   AGG               0.10         0.003395745
```

Huzzah - our findings seem to be consistent!

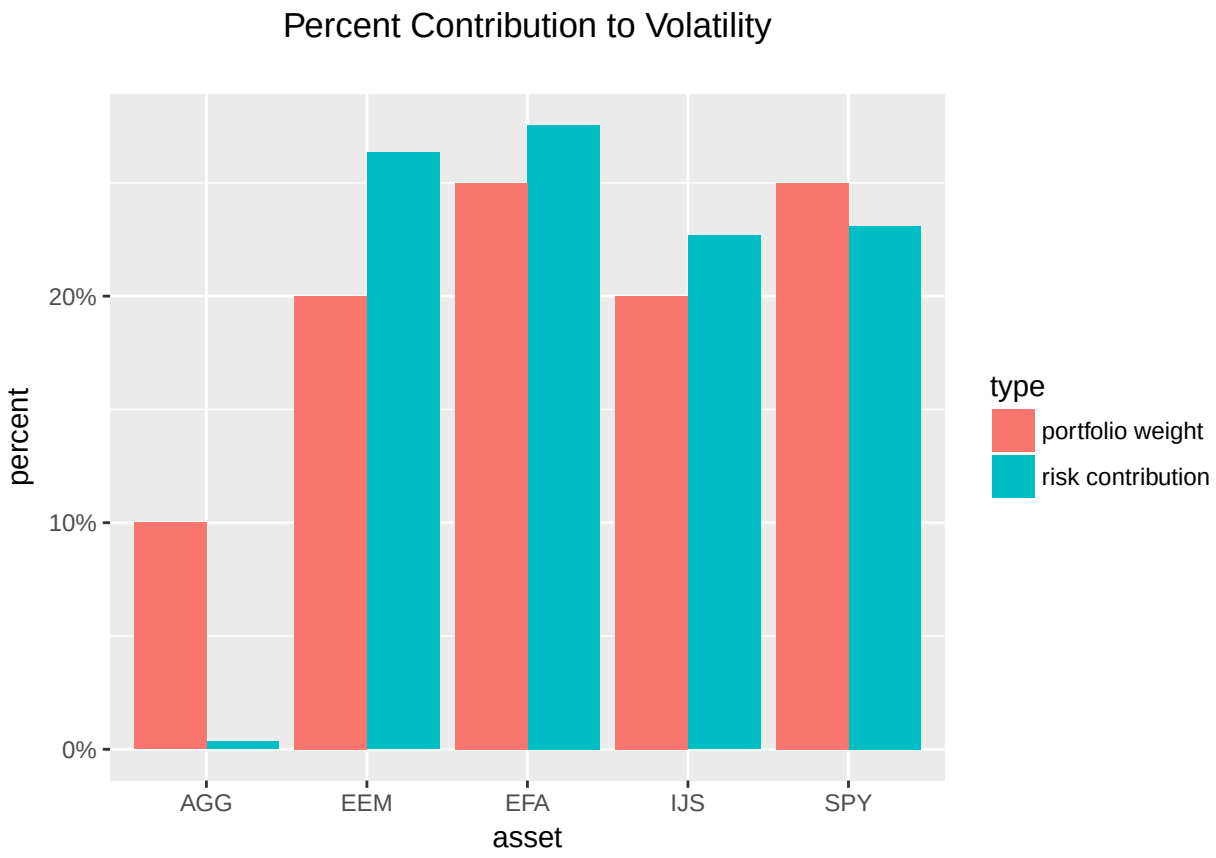
Our substantive work is done but let's turn to `ggplot` for some visualizing.

```
component_percent_plot <-
  ggplot(percentage_tibble_by_hand, aes(asset, `risk contribution`)) +
  geom_col(fill = 'blue', colour = 'red') +
  scale_y_continuous(labels = scales::percent) +
  ggtitle("Percent Contribution to Volatility",
          subtitle = "") +
  theme(plot.title = element_text(hjust = 0.5)) +
  theme(plot.subtitle = element_text(hjust = 0.5)) +
  xlab("Asset") +
  ylab("Percent Contribution to Risk")

component_percent_plot
```



How about a chart that compares weights to risk contribution. First we'll need to gather our tibble to long format, then call `ggplot`.

```
# gather
percentage_tibble_by_hand_gather <-
  percentage_tibble_by_hand %>%
  gather(type, percent, -asset)

# built ggplot object
plot_compare_weight_contribution <-
  ggplot(percentage_tibble_by_hand_gather, aes(x = asset, y = percent, fill = type)) +
  geom_col(position='dodge') +
  scale_y_continuous(labels = scales::percent) +
  ggtitle("Percent Contribution to Volatility",
          subtitle = "") +
  theme(plot.title = element_text(hjust = 0.5)) +
  theme(plot.subtitle = element_text(hjust = 0.5))

plot_compare_weight_contribution
```



It looks like AGG, a bond fund, has done a good job as a volatility dampener. It has a 10% allocation but contributes almost zero to volatility. We're ignoring returns for now.

The largest contributor to the portfolio volatility has been EEM, an emerging market ETF, but have a look at the EEM chart and note that it's own absolute volatility has been quite low.

```
EEM_sd <- StdDev(asset_returns_xts$EEM)

EEM_sd_overtime <-
  round(rollapply(asset_returns_xts$EEM, 20, function(x) StdDev(x)), 4) * 100

highchart(type = "stock") %>%
  hc_title(text = "EEM Volatility") %>%
  hc_add_series(EEM_sd_overtime, name = "EEM Vol") %>%
  hc_yAxis(labels = list(format = "{value}%"), opposite = FALSE) %>%
  hc_navigator(enabled = FALSE) %>%
  hc_scrollbar(enabled = FALSE)
```

EEM has contributed 37% to portfolio volatility, but it hasn't been very risky over this time period. It's standard deviation has been 0.0420505. Yet, it is still the riskiest asset in our portfolio. Perhaps this is a safe portfolio? Or perhaps we are in a period of very low volatility (Indeed, that is the case. See this post for a visualization on the VIX and volatility since 2008).

That's all for today. The next post will be about determining the rolling contribution of each asset and it's not for the faint of heart.

Rolling Component Contribution

First, we need to build a function to calculate the rolling contribution of each asset. The previous sectio told us the total contribution of each asset over the life of the portfolio, but it did not help us understand risk components over time. As we discussed in our section on rolling portfolio volatility, there's reasons to care about rolling.

Our goal is to create a function that takes (1) a `data.frame` of asset returns and calculates the rolling contributions of each asset to volatility, based on a (2) starting date index and a (3) window, for a portfolio (4) with specified weights of each asset. We will need to supply four arguments to the function, accordingly.

Here's the logic I used to construct that function (feel free to eviscerate this logic and replace it with something better).

1. Assign a start date and end date based on the window argument. If we set window = 6, we'll be calculating 6-month rolling contributions.
2. Use `filter()` to subset the original `data.frame` down to one window. I label the subsetted data frame as `interval_to_use`. In our example, that interval is a 6-month window of our original data frame.
3. Now we want to pass that `interval_to_use` object to `StdDev()` but it's not an `xts` object. We need to convert it and label it `returns_xts`.
4. Before we call `StdDev()`, we need weights. Create a weights object called `w` and give the value from the argument we supplied to the function.
5. Pass the `returns_xts` and `w` to `StdDev()`, and set `portfolio_method = "component"`.
6. We now have an object called `results_as_xts`. What is this? It's the risk contributions for each asset during the first 6-month window of our weighted portfolio.
7. Convert it back to a `tibble` and return.
8. We now have the risk contributions for the 6-month period that started on the first date, because we default to `start = 1`. If we wanted to get the standard deviation for a 6-month period that started on

the second date, we could set `start = 2`, etc.

```
my_interval_sd <- function(returns_df, start = 1, window = 6, weights){

  # First create start date.
  # For now let's always start at the first date for which we have data.
  start_date <- returns_df$date[start]

  # Next an end date.
  end_date <-  returns_df$date[c(start + window)]

  # Filter on start and end date.
  interval_to_use <- returns_df %>% filter(date >= start_date & date < end_date)

  # Convert to xts so can use built in Performance Analytics function.
  returns_xts <- interval_to_use %>% tk_xts(date_var = date)

  # Portfolio weights.
  w <- weights

  # Pass xts object to function.
  results_as_xts <- StdDev(returns_xts, weights = w, portfolio_method = "component")

  # Convert results to tibble.
  results_to_tibble <- tk_tbl(t(results_as_xts$pct_contrib_StdDev)) %>%
    mutate(date = ymd(end_date)) %>%
    select(date, everything())
}
```

We're halfway there, though, because we need to apply that function starting at the first date in our `returns_df` object, and keep applying it to successive date indexes until the date that is 6-months before the final date. Why end there? Because there is no rolling 6-month contribution that starts only 1, 2, 3, months ago!

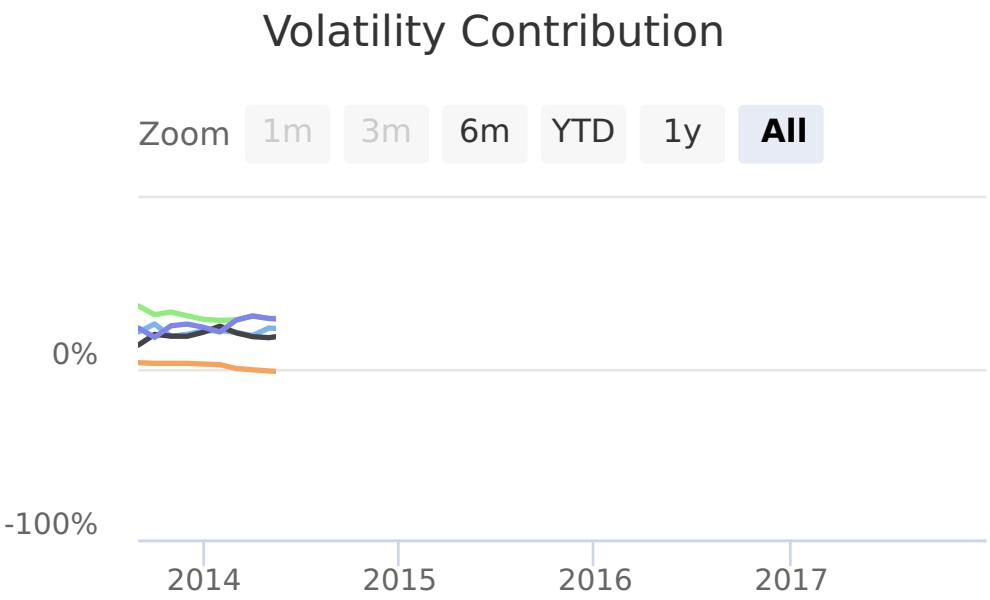
We will invoke `map_df()` to apply our function to date 1, then save the result to a `data.frame`, then apply our function to date 2, and save to that same `data.frame`, and so on until we tell it stop at the at index that is 6 months before the last date index.

```
window <- 6

rolling_vol_contributions <-
  map_df(1:(nrow(asset_returns_dplyr_byhand) - window),
         my_interval_sd, returns_df = asset_returns_dplyr_byhand,
         window = window, weights = w) %>%
  mutate(date = ymd(date)) %>%
  select(date, everything()) %>%
  # Convert back to xts so we can use highcharter for visualizations.
  tk_xts(date_var = date) %>%
  round(., 4) *100


portfolio_vol_components_tidy <-
    map_df(1:(nrow(asset_returns_dplyr_byhand)-window),
           my_interval_sd, returns_df = asset_returns_dplyr_byhand,
           weights = w, window = window) %>%
    mutate_all(funs(round(., 3))) %>%
```

```
    mutate(date = ymd(date)) %>%
    select(date, everything())
```

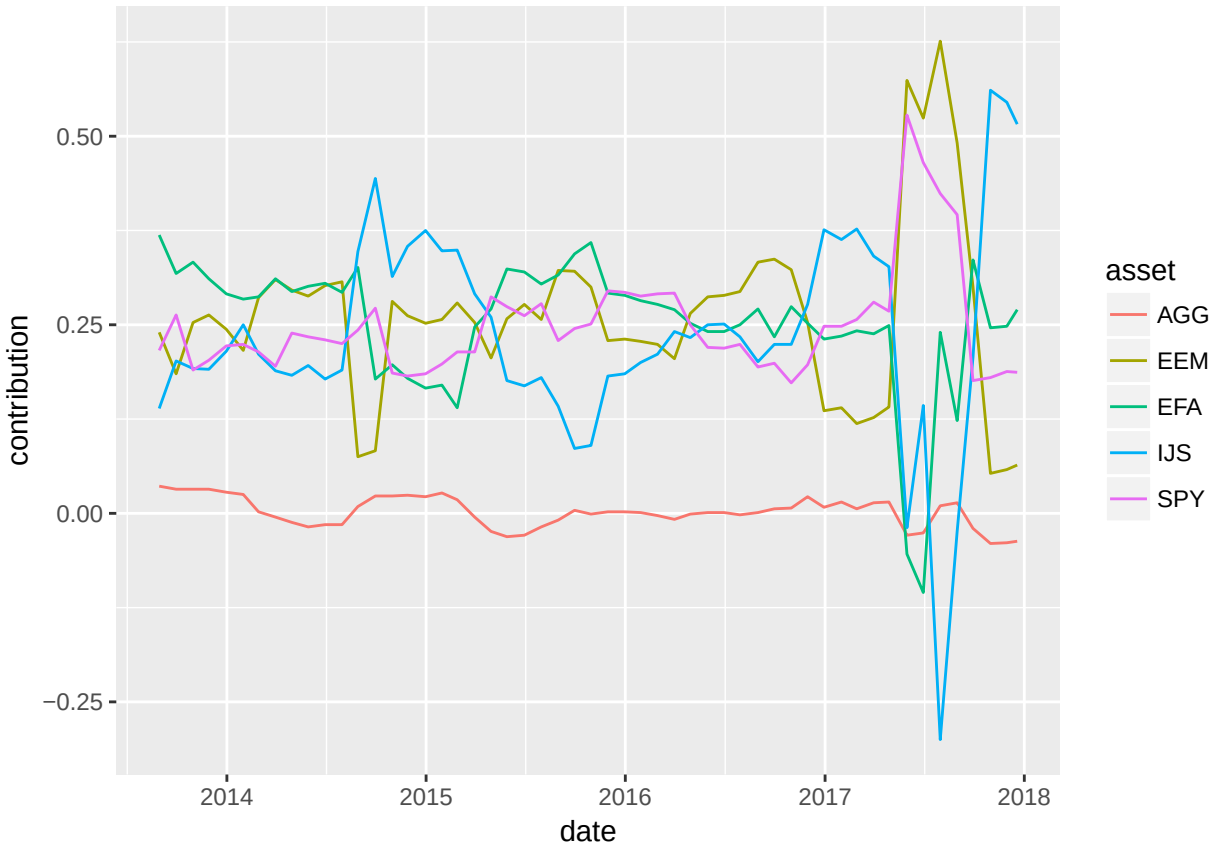
Now we can visualize with `highcharter` by adding the rolling conbtribution of each asset to a chart.

```
highchart(type = "stock") %>%
hc_title(text = "Volatility Contribution") %>%
hc_add_series(rolling_vol_contributions$SPY, name = "SPY", id = "SPY") %>%
hc_add_series(rolling_vol_contributions$IJS, name = "IJS") %>%
hc_add_series(rolling_vol_contributions$EFA, name = "EFA" )%>%
hc_add_series(rolling_vol_contributions$AGG, name = "AGG") %>%
hc_add_series(rolling_vol_contributions$EEM, name = "EEM") %>%
hc_yAxis(labels = list(format = "{value}%"), max = 100, opposite = FALSE) %>%
hc_navigator(enabled = FALSE) %>%
hc_scrollbar(enabled = FALSE)
```



For some reason, EEM rolling volatility spiked in June. Did something roil emerging markets in the preceding months? And IJS vol plunged, which seems unusual for a small-cap fund. AGG and EFA have had stable volatilities.

```
portfolio_vol_components_tidy %>%
  gather(asset, contribution, -date) %>%
  group_by(asset) %>%
  ggplot(aes(x = date)) +
  geom_line(aes(y = contribution, color = asset))
```



To shiny with component contribution