Group Project Lab Book

Portfilio Optimisation

Edward Davis, Conor Crilly, Conor Newton

https://github.com/edwarddavis1/PortfolioOptimisation

Introduction

Portfolio optimisation is the area of study which works to find the most optimal portfolio of assets for investment which maximises the return while adhering to a specified market risk. In finance, money can be made by investing in assets as the investment is exposed to market risk; with larger risks having the potential to yield larger returns. However, too much risk in an investment could also cause significant losses to an investor. In this project, we will present the R code used to perform portfolio optimisation when used with SP500 data; a stock market index which includes 500 large companies in the United States, covering a significant portion of the equity market in the country.

Getting SP500 Data

To perform portfolio optimisation, we require functions which can get and process SP500 data for selected assets. The function, `get_sp500_data()` allows the user to generate a data frame for selected `stock_names`, and has the (default) option to remove assets which contain missing data. This allows the user to use the data without having to deal with `NULL` data points. This processing of missing data is carried out using the `missing_data()` and `remove_missing_data()` functions. We have utilised data frames to store the data as they allow for quick data manipulation; in `remove_missing_data()`, the identified assets with missing data can be removed from the data by name as data frames can be manipulated conditionally.

```r
#' @param data List of data frames to search for missing rows in
#' @return Returns symbols that could be missing data
missing_data <- function(data) {
  names(data[lapply(data, nrow) != max(unique(sapply(data, nrow)))])
}

#' @param data Remove missing data from this dataset
#' @return Returns the modified data frame
remove_missing_data <- function(data) {
  names_to_delete <- missing_data(data)
  data <- data[names(data) %in% names_to_delete == FALSE]
  return(data)
}

#' Downloads stock data from yahoo finance
#' @param stock_names Vector including the symbols of stocks we want to retrieve
#' @param from Date to get data after
#' @param to Date to get data until
#' @param remove_missing Removes the data with missing rows
#' @return List of dataframes containg data for each sp500 company
```

```r
get_sp500_data <- function(stock_names = sp500_names,
                           from = "2019-11-27",
                           to = Sys.Date(),
                           remove_missing = TRUE,
                           show_output=FALSE,
                           bind=FALSE) {
  data <- list()
  for (i in seq_len(length(stock_names))) {
    if (show_output) print(stock_names[i])
    data[[i]] <- as.data.frame(tidyquant::tq_get(stock_names[i],
                                                 get = "stock.prices",
                                                 from = from,
                                                 to = to))
    names(data)[i] <- stock_names[i]
  }

  if (remove_missing) {
    data <- remove_missing_data(data)
  }
  if (bind) data = bind_rows(data)

  return(data)
}
```

The `portfolio` Class

Member Data

The main part of the code for this project is within the `portfolio` reference class. This method of object orientated programming was selected as it is more manageable when dealing with large problems. A `portfolio` object is created by specifying assets and a time range for evaluation. The object both stores asset stock data from the SP500 and calculates metrics relevant for optimisation schemes. These metrics and data are stored a class member data, which are as follows.

- `assets`: A column vector of symbol characters which specify chosen assets for the portfolio.
- `data`: A data frame containing SP500 data for the selected assets.
- `shares`: A column vector of the number of shares owned for each asset.
- `holdings`: A column vector of holding values for each asset.
- `time`: The number of time steps since portfolio investment.
- `value`: The total portfolio value at the given time.
- `weights`: A column vector of asset weightings.
- `T`: The number of SP500 data points during the time range.
- `investment`: The sum of money to be invested in the portfolio.
- `returns_matrix`: A matrix of assets by asset returns.
- `asset_summary`: A data frame containing risk and return metrics for each asset.

```r
portfolio = setRefClass("portfolio",
                        fields=c(
                          assets = "character",
                          data = "data.frame",
                          shares = "matrix",
                          holdings = "matrix",
                          time = "numeric",
                          value = "numeric",
                          weights = "matrix",
                          T = "numeric",
                          investment = "numeric",
                          returns_matrix = "matrix",
                          asset_summary = "data.frame"
                        ))
```

Class Methods

Constructor

A `portfolio` object is constructed through the use of the reference class method, `initialize()`. By default, the constructor will create a portfolio object including all assets within the SP500, with balanced weights as well as some example investment and date range (specified by `from` and `to` arguments). The constructor then takes the arguments which the user selects and calculates values for each member variable.

```
#' @param from First date of the portfolio range
#' @param to Last date of the portfolio range
initialize = function(assets=sp500_names, weights=NULL, investment=1e4,
           from="2019-11-27", to = Sys.Date()) {
```

If the user decides to specify their own weights, the constructor has checks to make sure that the weights vector is valid; a vector with length equal to the number of assets and whose values sum to one. The function `stop()` is used when these conditions are not met and throws an error specifying which condition the weights failed. If the weights are not specified (by default `weights=NULL`), the weighting is evenly split between the assets.

```
# Check that the selected weighting vector is okay
if (is.null(weights)) const_weight = TRUE
else {
  const_weight = FALSE
  if (sum(weights) != 1) stop("Error: Weights do not add to 1")
  if (length(weights) != n) stop("Error: Weights vector and assets vector are different lengths")
}

# By default have a constant weight portfolio
if (const_weight) .self$weights = as.matrix(rep(1/n, n))
else .self$weights = as.matrix(weights)
```

An additional check made my the constructor is that the selected stock exists in the specified time frame. In the case where the stock does not exist (i.e. the stock price being zero), the shares for that asset will be set to zero (to avoid infinities) and a warning will be thrown, informing the user that the asset will not be useful (due to having zero shares).

```
# Get initial number of shares from the initial stock prices
initial_price = (.self$data %>% filter(symbol == assets[i] & date == date[1]))$adjusted
if (initial_price > 0) temp_shares[i] = .self$holdings[i] / initial_price
# If the stock price is zero throw a warning
else {
  temp_shares[i] = 0
  warning("Warning: Asset does not yet exist!")
}
```

After all initial values of member variables have been calculated, the constructor uses `set_time()` to set the variables to be those at the end of the portfolio date range.

`set_time()`

This function acts as the setter function for the `time` member variable. The portfolio object uses `time` to specify the portfolio value and holdings within the time range. In the constructor, the function is called as `set_time(.self$T)`, which sets the time of the portfolio to be at the last data point in the SP500 data as this is the time we are most interested in (the time where all shares are sold). However, this function exists to provide the user with the freedom to explore the state of the portfolio at different times. To make sure that the user inputs appropriate times, the `stop()` function has been used again to throw errors if the time is not given as an integer (as is the number of time points from the start date) and that said time is within the date range.

```r
# Must be within time range
if (time < 1 | time > .self$T) stop("Error: Time not in range")
# Must be integer
if (time%%1 != 0) stop("Error: Time must be an integer value")
```

When the function calculates the member data, their values are initially stored as temporary variables. By default, the function will update the member data with these temporary variables, but the argument, `update`, can be chosen to be false, which then outputs the portfolio value at that time without updating the member data. This allows the function to be used in the generation of member data time series.

```r
# Update member data if specified
if (update) {
  .self$time = temp_time
  .self$holdings = temp_holdings
  .self$value = temp_value
} else return(temp_value)
```

`set_weights()`

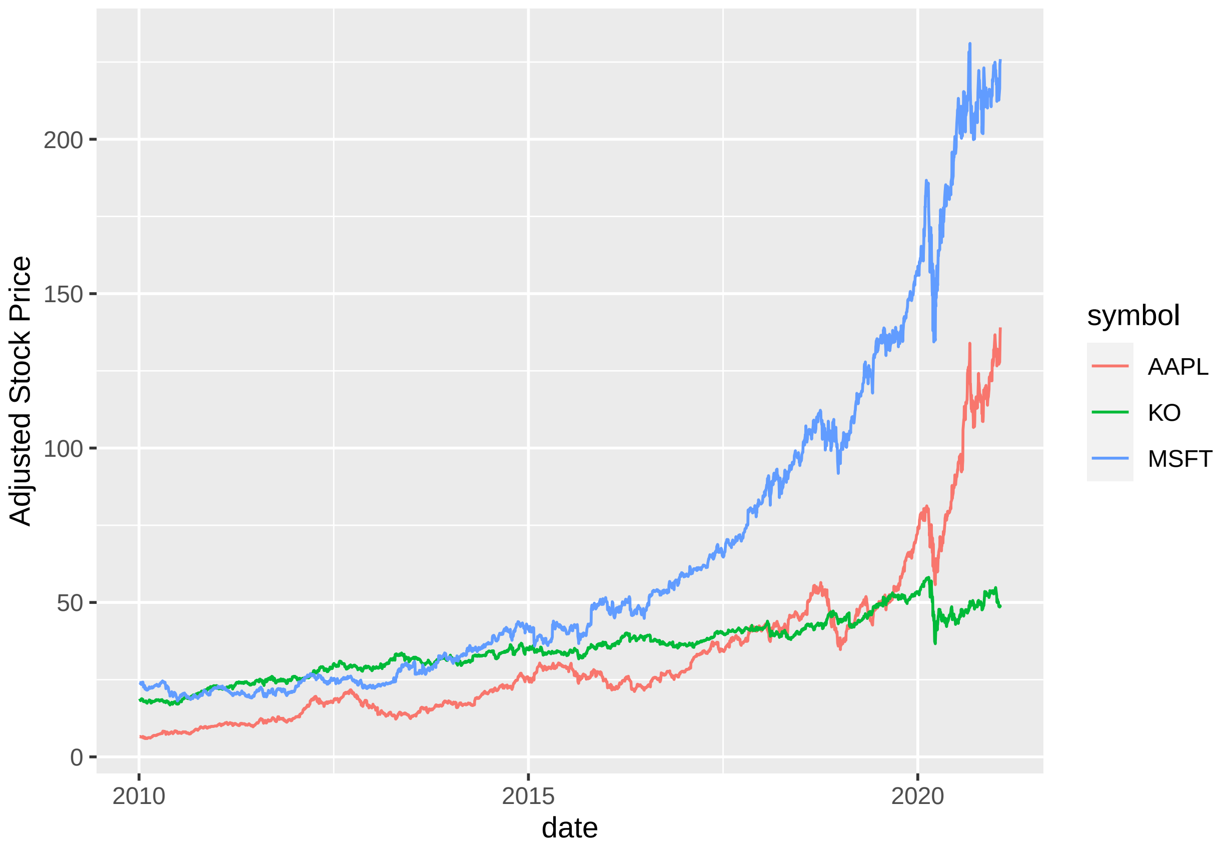
The `set_weights()` method is the setter of the weights member variable, allowing the user to enter their own weightings or reset the weights after an unsuccessful optimisation. The function, like the constructor, throws errors if the entered weights vector is not valid.

```r
set_weights = function(new_weights) {
  # Check weights are okay
  if (sum(new_weights) != 1) stop("Error: Weights do not add to 1")
  if (length(new_weights) != length(.self$assets)) stop("Error: Weights vector and assets vector are di
  .self$weights = as.matrix(new_weights)
}
```

Plotting functions

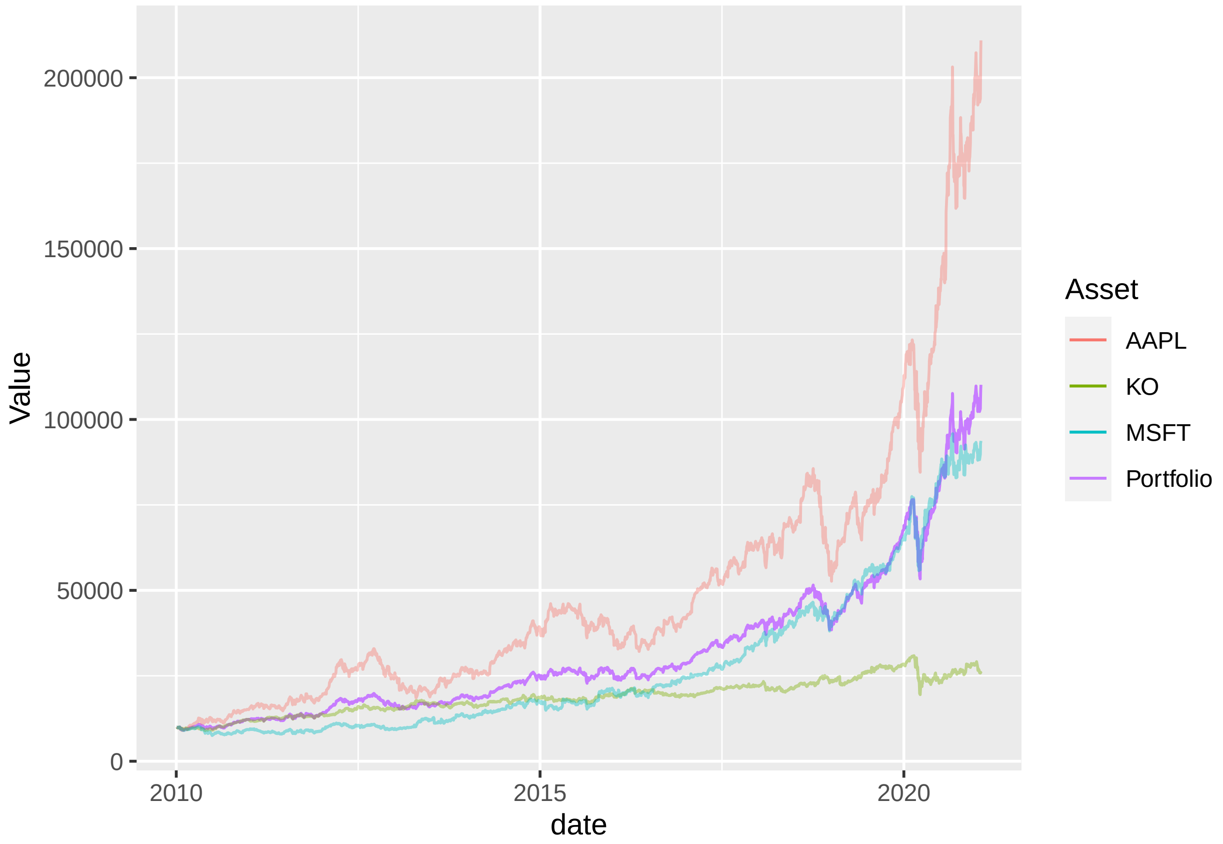
To view how the assets in a portfolio object change over the time range selected, the user can use the `plot_assets()` function, which plots the stock price time series for each asset.

```r
library(tidyquant)
library(tidyverse)
library(magrittr)
library(methods)
library(ggplot2)
library(dplyr)
source('../R/general_functions.R')
source('../R/portfolio_class.R')
ptf = portfolio(c("AAPL", "MSFT", "KO"), from = "2010-01-01")
ptf$plot_assets()
```



However, to illustrate the performance of the portfolio, the `plot_value()` function can be used. This function plots the value (weighted holding) for each asset and, by default, the total value of the portfolio. This is especially useful when trying to work out if a specific weighting vector is successful as an optimal portfolio's value should be above most of the asset values.

```
ptf$plot_value()
```

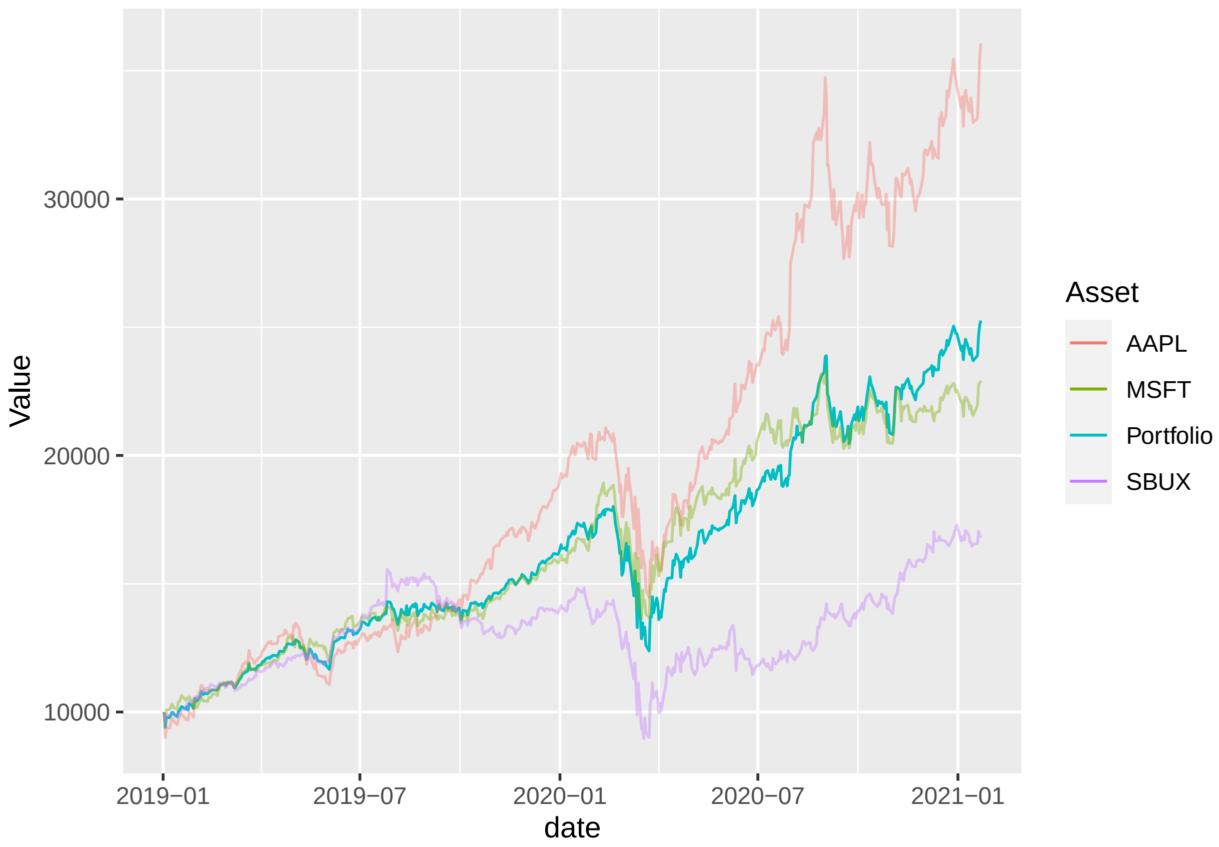


The function has the arguments `inc_portfolio_value`, which can be used to exclude the portfolio value in plotting, and `return_values` which if set to `TRUE` will return a portfolio value time series in a data frame instead of plotting. The latter of which was included in case the time series was required for further plotting; for example, comparing the values over time for different optimisation schemes.

`ls_optimize()`, `mean_variance_optimize()` and `mean_semivariance_optimize()`

The class has three optimisation methods which act to optimise the weights vector. `ls_optimize()` employs a least squares optimisation, and the `mean_variance_optimize()` and `mean_semivariance_optimze()` functions employ two quadratic programming optimisations.

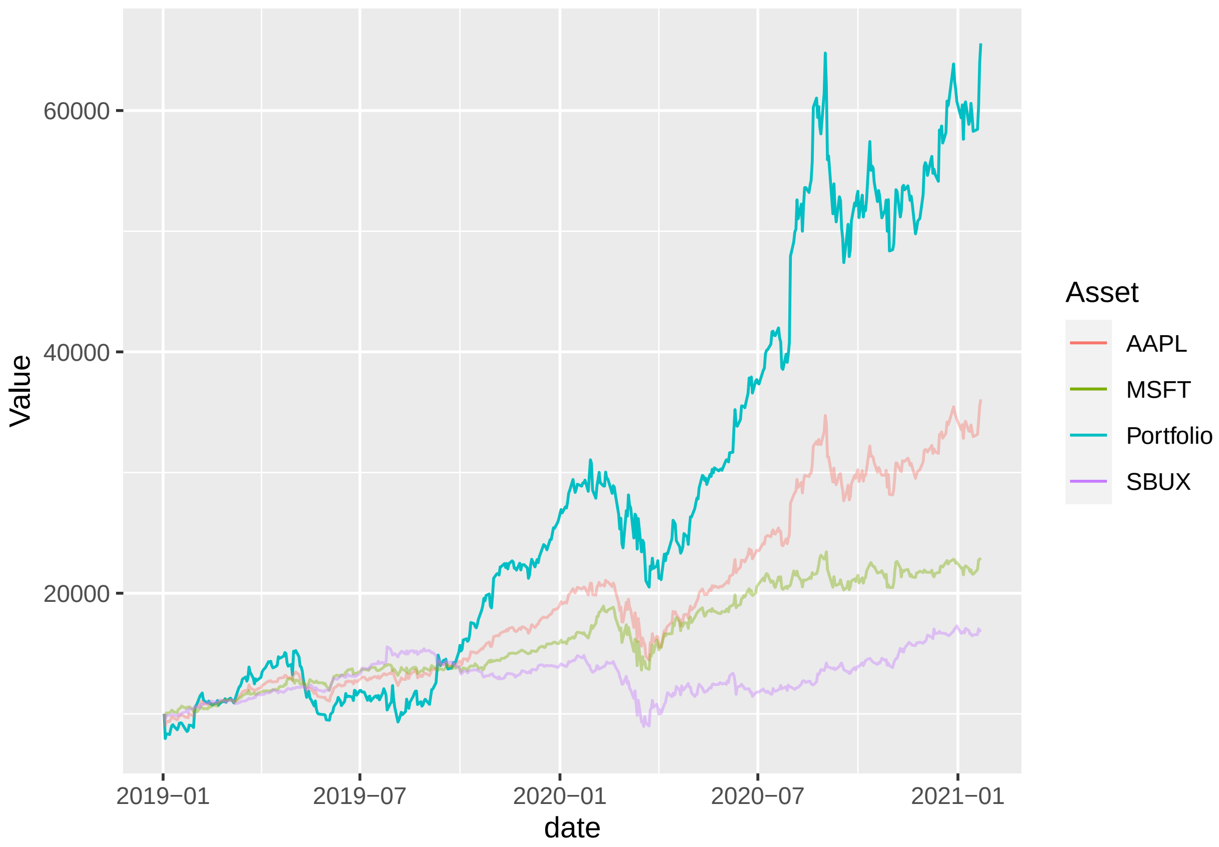
```r
# Equally weighted portfolio value
ptf = portfolio(c("AAPL", "MSFT", "SBUX"), from = "2019-01-01")
equal_weights_value_df = ptf$plot_value(return_values=TRUE)
ptf$plot_value()
```



```
# Least squares optimisation
ptf$ls_optimize(0.5*nrow(ptf$returns_matrix))
```

```
##            [,1]
## [1,]  2.7793182
## [2,] -0.7807553
## [3,] -0.9985629
```

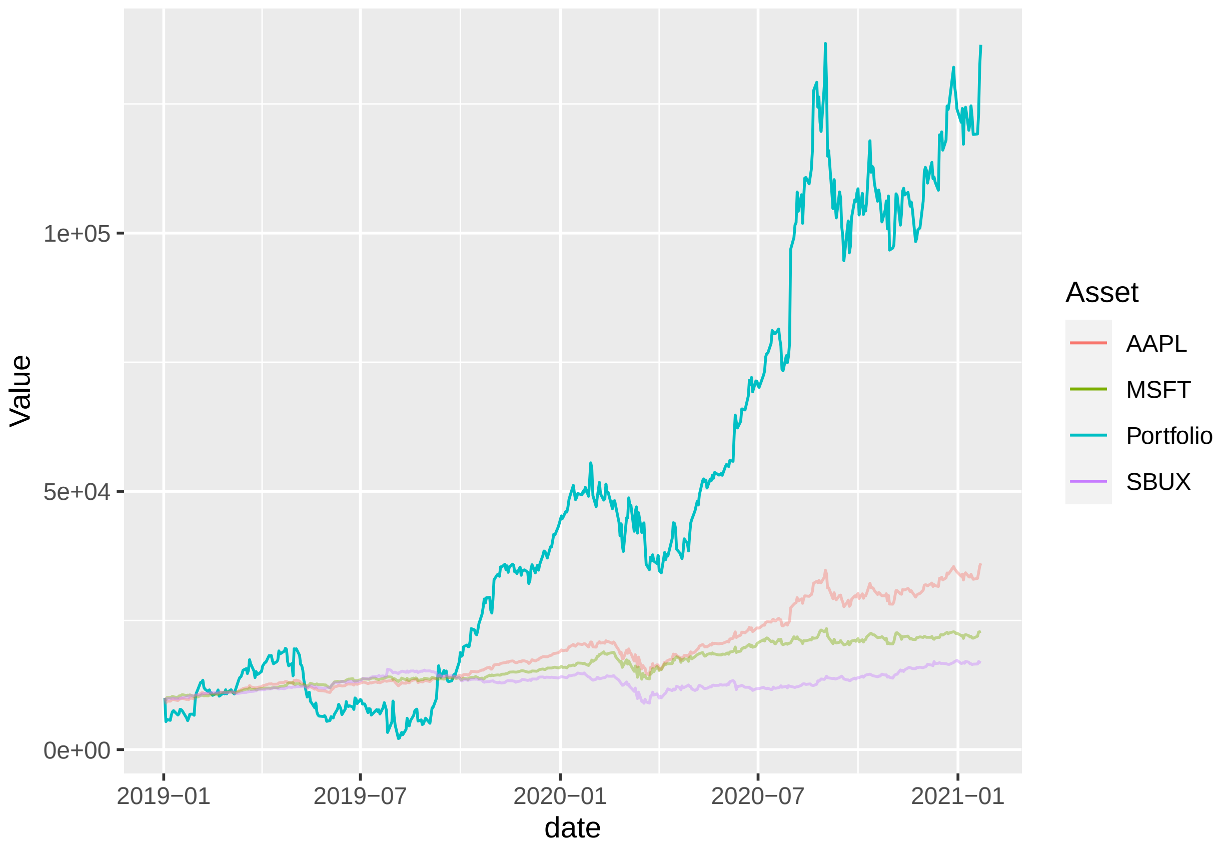
```
ls_value_df = ptf$plot_value(return_values=TRUE)
ptf$plot_value()
```



```
# Mean-variance
ptf$mean_variance_optimize(0.002 * nrow(ptf$returns_matrix), short=TRUE)
```

```
##            [,1]
## [1,]  7.077170
## [2,] -2.730329
## [3,] -3.346841
```

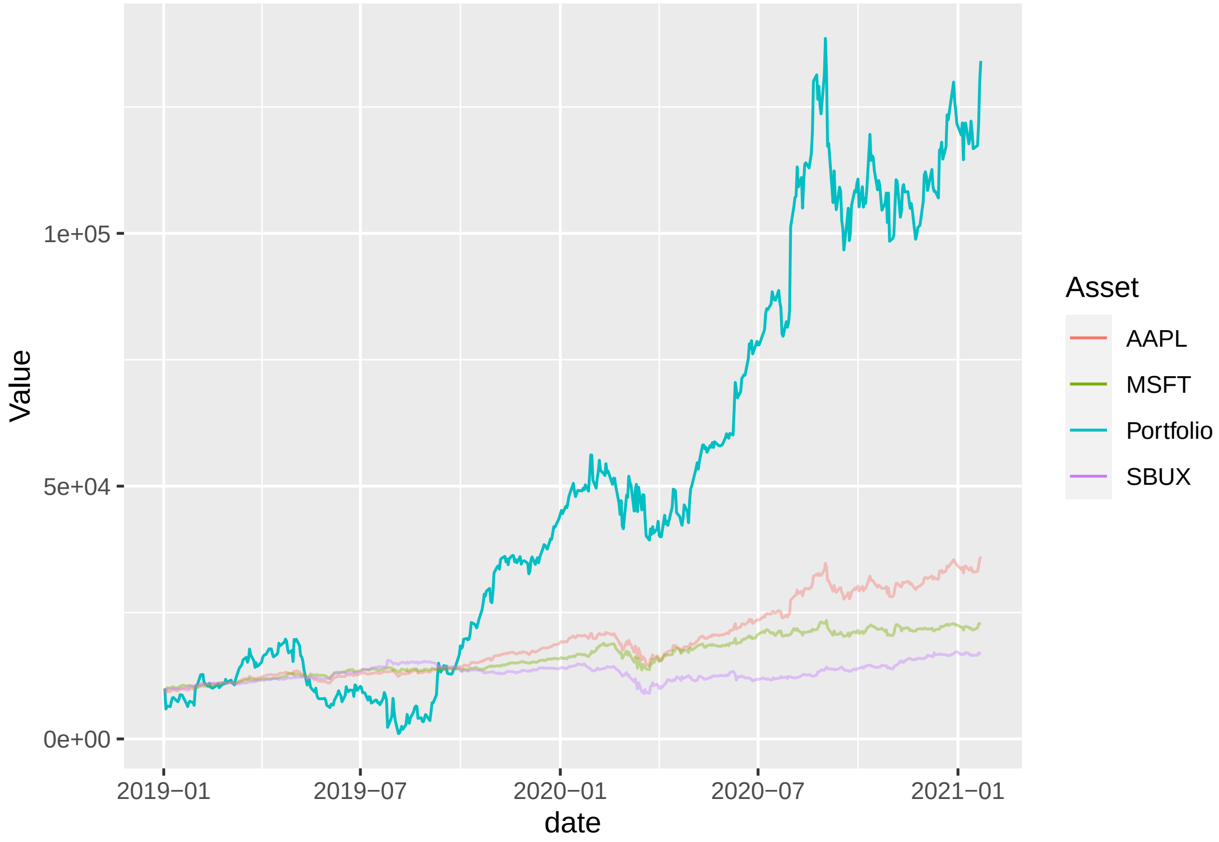
```
mv_value_df = ptf$plot_value(return_values=TRUE)
ptf$plot_value()
```



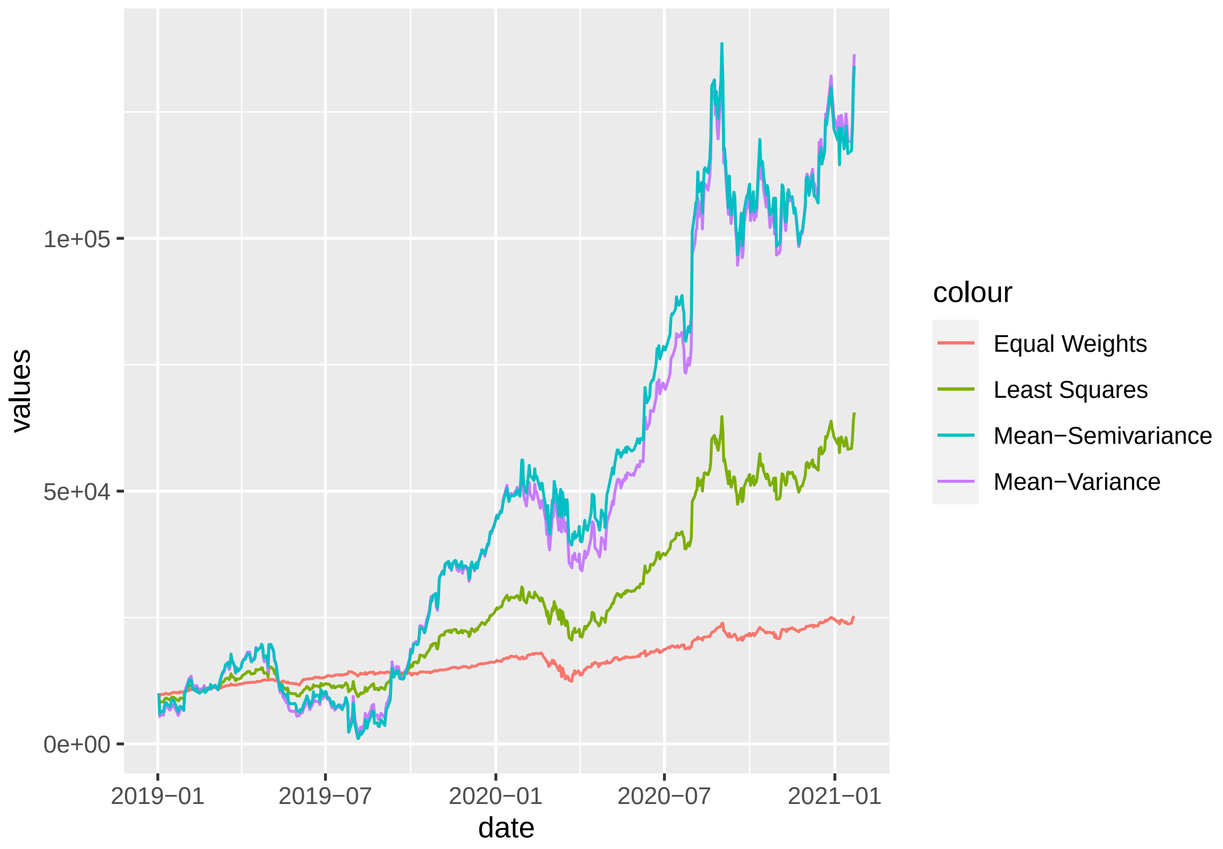
```
# Mean-semivariance
ptf$mean_semivariance_optimize(0.002 * nrow(ptf$returns_matrix), short=TRUE)
```

```
##            [,1]
## AAPL  6.3998057
## MSFT -0.9816175
## SBUX -4.4181882
```

```
msv_value_df = ptf$plot_value(return_values=TRUE)
ptf$plot_value()
```



```
# Compare methods
ggplot() +
  geom_line(data=equal_weights_value_df, aes(x=date, y=values, col="Equal Weights")) +
  geom_line(data=ls_value_df, aes(x=date, y=values, col="Least Squares")) +
  geom_line(data=mv_value_df, aes(x=date, y=values, col="Mean-Variance")) +
  geom_line(data=msv_value_df, aes(x=date, y=values, col="Mean-Semivariance"))
```



Markowitz Bullet

The following code produces the Markowitz Bullet plot in the three risky asset case and is an example of how one can use methods from the portfolio class to conduct portfolio performance analysis. We include three assets - Apple, Starbucks and Microsoft for between 1 January 2018 and 1 July 2020.

```
# generic three asset example
my.portfolio <- portfolio(c('AAPL', 'SBUX', 'MSFT'), from = "2018-01-01",
                          to = "2020-07-01")
# Number of assets in the portfolio
n.assets <- length(my.portfolio$assets)
# Percentage returns
returns.matrix <- my.portfolio$returns_matrix/100
# Variance matrix of returns
sigma <- var(returns.matrix)
#Mean returns
mean.returns <- my.portfolio$asset_summary$avg_percentage_return/100
```

We now find the global minimum variance portfolio which will be useful for comparison later.

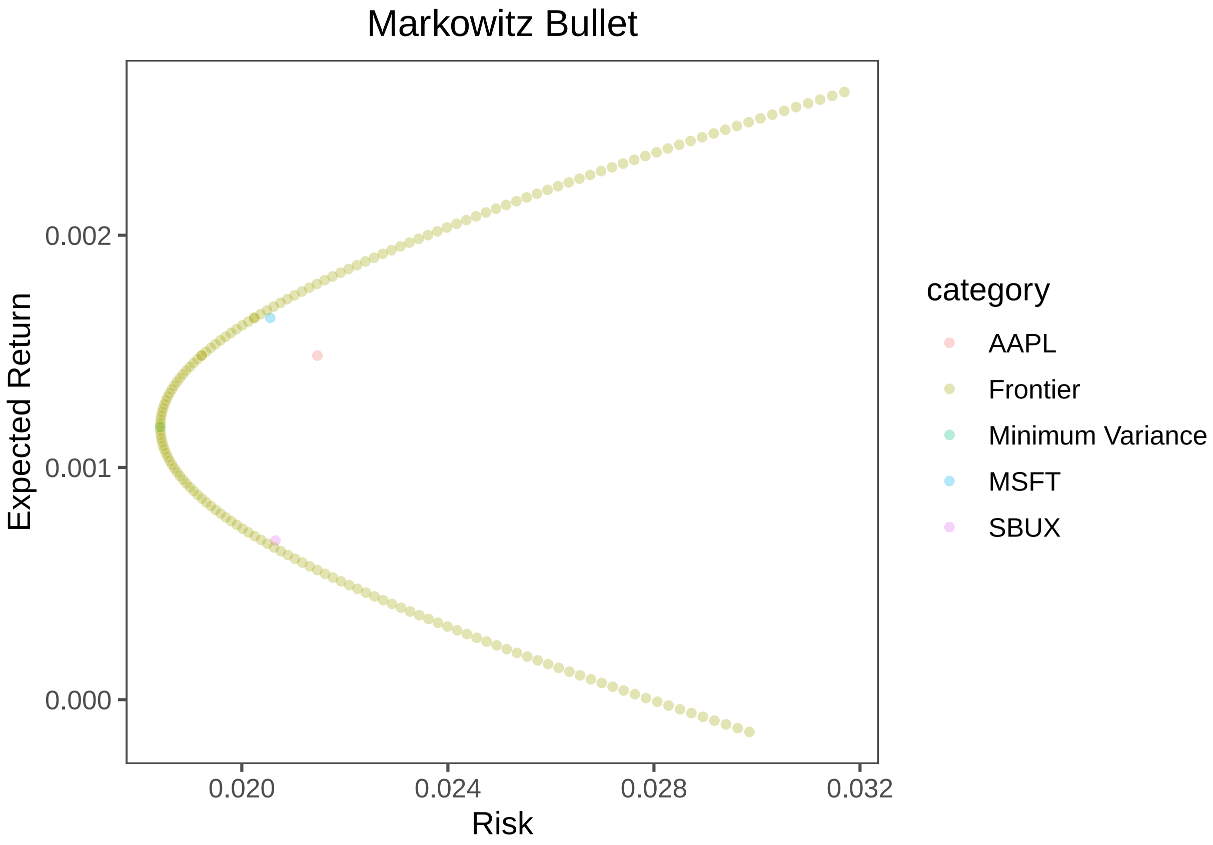
```
# Finding the global minimum variance portfolio - see notes section 1.1.2
get.global.min.var.portfolio <- function(ret.mat) {
  sigma.mat <- 2*var(ret.mat)
  binding.mat <- c(rep(1,ncol(ret.mat)),0)
  rhs.vec <- c(rep(0,ncol(ret.mat)),1)
  lhs.mat <- rbind(cbind(sigma.mat, binding.mat), binding.mat)
  return(solve(lhs.mat, rhs.vec))
}
# Getting the global minimum variance portfolio
global.min.var.portfolio <- get.global.min.var.portfolio(ret.mat =
                                                           my.portfolio$returns_matrix/100)
```

```
## Warning in cbind(sigma.mat, binding.mat): number of rows of result is not a
## multiple of vector length (arg 2)
```

To find the efficient frontier, we need only two efficient portfolios. We can then take convex combinations to find efficient frontier portfolios with arbitrary returns. Efficient portfolios are those that lie on the boundary of the investment opportunity set, with greater expected return than the global minumum variance portfolio. As an example, we can consider finding efficient portfolios with the same return as AAPL and MSFT.

```
# Function to return efficient portfolio with a target return
get.efficient.portfolio <- function(n.assets, mean.returns, target.return, ret.matrix) {
  top.mat <-cbind(2*var(ret.matrix), mean.returns, rep(1,n.assets))
  mid.vec <- c(mean.returns, 0, 0)
  bot.vec <- c(rep(1,3), 0, 0)

  lhs.mat <- rbind(top.mat, mid.vec, bot.vec)

  rhs.vec <- c(rep(0,n.assets), target.return, 1)

  efficient.portfolio <- solve(lhs.mat, rhs.vec)

  return(efficient.portfolio)
}
# Apple and Microsoft returns
AAPL.returns <- mean.returns[my.portfolio$assets == 'AAPL']
MSFT.returns <- mean.returns[my.portfolio$assets == 'MSFT']
# Creating portfolio with same return as Apple
aapl.efficient <- get.efficient.portfolio(n.assets = n.assets,
                                          mean.returns = mean.returns,
                                          target.return = AAPL.returns,
                                          ret.matrix = returns.matrix)
# Creating portfolio with same return as Apple
MSFT.efficient <- get.efficient.portfolio(n.assets = n.assets,
                                          mean.returns = mean.returns,
                                          target.return = MSFT.returns,
                                          ret.matrix = returns.matrix)
# Function to take convex combinations of efficient frontier portfolios
new.efficient.portfolio <- function(alpha, p1.weights, p2.weights, ret.matrix) {
  combined.weights <- alpha*p1.weights + (1-alpha)*p2.weights
  combined.mean <- combined.weights%*%mean.returns
  combined.risk <- sqrt(combined.weights%*%var(ret.matrix)%*%combined.weights)
  return(c(combined.risk, combined.mean))
}
# We can plot these here and compare to the individual assets
plot.markowitz.bullet <- function(ret.mat, av.returns, portfolio.one,
                                  portfolio.two, global.min.var.portfolio) {
  single.asset.sd <- sqrt(diag(var(ret.mat)))
  single.asset.means <- av.returns
```

```r
risk.return.single.assets <- as.data.frame(cbind(single.asset.sd, single.asset.means))

n.assets = ncol(ret.mat)
sigma <- var(ret.mat)
p1.efficient.sd <- sqrt(portfolio.one[1:n.assets]%*%sigma%*%portfolio.one[1:n.assets])
p2.efficient.sd <- sqrt(portfolio.two[1:n.assets]%*%sigma%*%portfolio.two[1:n.assets])

p1.efficient.return <- portfolio.one[1:n.assets]%*%av.returns
p2.efficient.return <- portfolio.two[1:n.assets]%*%av.returns

# Global Minimum Variance Portfolio
mean.global.var.portfolio.return <- global.min.var.portfolio[1:n.assets]%*%av.returns
# Portfolio variance - m*sigma*m
m <- global.min.var.portfolio[1:n.assets]
sd.min.global.var.portfolio.return <- sqrt(m%*%sigma%*%m)

plot.data <- rbind(risk.return.single.assets,
                   c(sd.min.global.var.portfolio.return,
                     mean.global.var.portfolio.return),
                   c(p1.efficient.sd, p1.efficient.return),
                   c(p2.efficient.sd, p2.efficient.return))

alphas <- seq(-10,7, by = 0.1)
means <- rep(0,length(alphas))
risks <- rep(0,length(alphas))

for (i in c(1:length(alphas))) {
  means[i] <- new.efficient.portfolio(alphas[i], MSFT.efficient[1:3],
                                      aapl.efficient[1:3], returns.matrix)[2]
  risks[i] <- new.efficient.portfolio(alphas[i], MSFT.efficient[1:3],
                                      aapl.efficient[1:3], returns.matrix)[1]
}

frontier.data <- as.data.frame(cbind(risks, means))
frontier.plot <- ggplot(data = frontier.data, aes(x = risks, y = means)) + geom_point()

colnames(plot.data) <- colnames(frontier.data)

full.data <- rbind(plot.data, frontier.data)

full.data$category <- rep('Frontier',nrow(full.data))

full.data$category[1:ncol(ret.mat)] <- colnames(returns.matrix)

full.data$category[4] <- "Minimum Variance"

full.data$category %<>% as.factor()

ggplot(data = full.data, aes(x = risks, y = means)) +
  geom_point(aes(col = category), alpha = 0.3, size = 1.2) +
  theme_few() +
  labs(x = "Risk", y = "Expected Return") +
  ggtitle("Markowitz Bullet") +
    theme(plot.title = element_text(hjust = 0.5))


}
plot.markowitz.bullet(ret.mat = returns.matrix, av.returns = mean.returns,
                      portfolio.one = aapl.efficient,
                      portfolio.two = MSFT.efficient,
                      global.min.var.portfolio = global.min.var.portfolio)
```

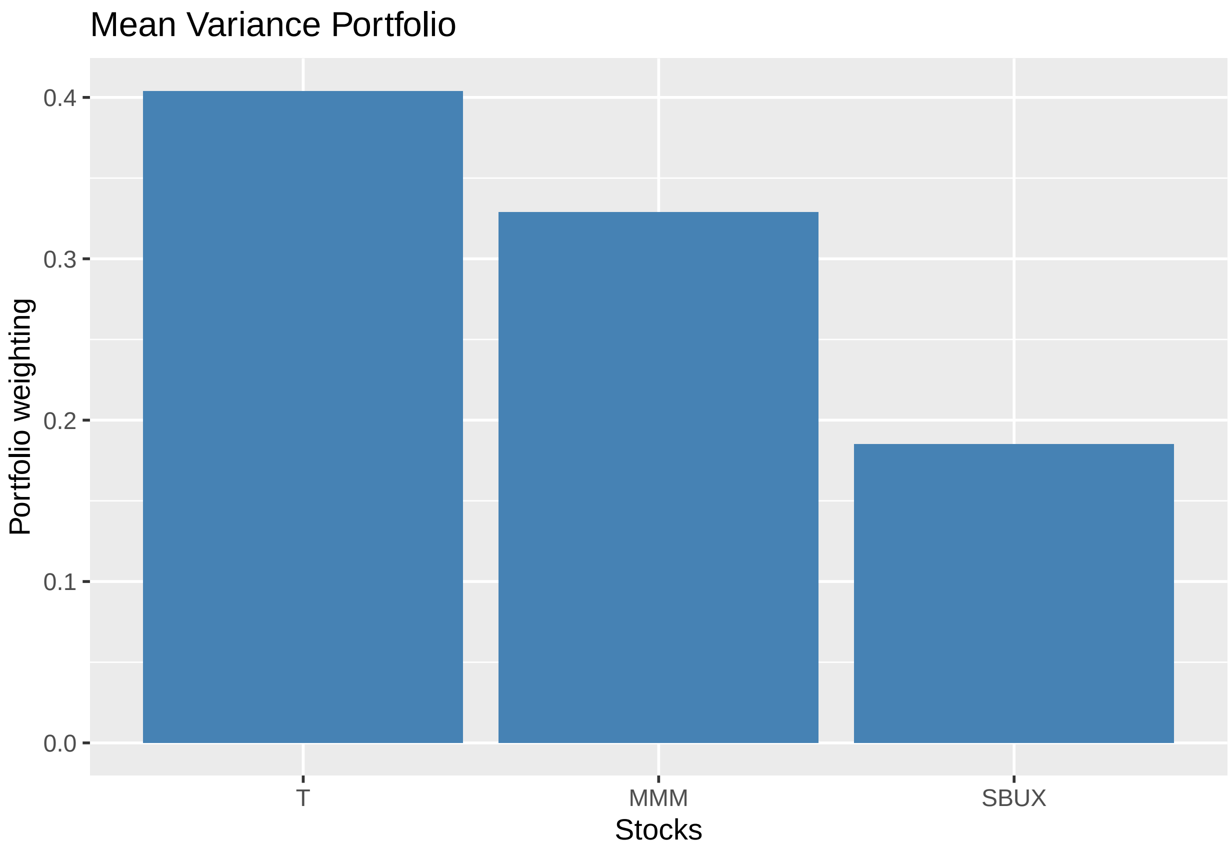


Comparing the Portfolios and Plotting the largest weighted assets

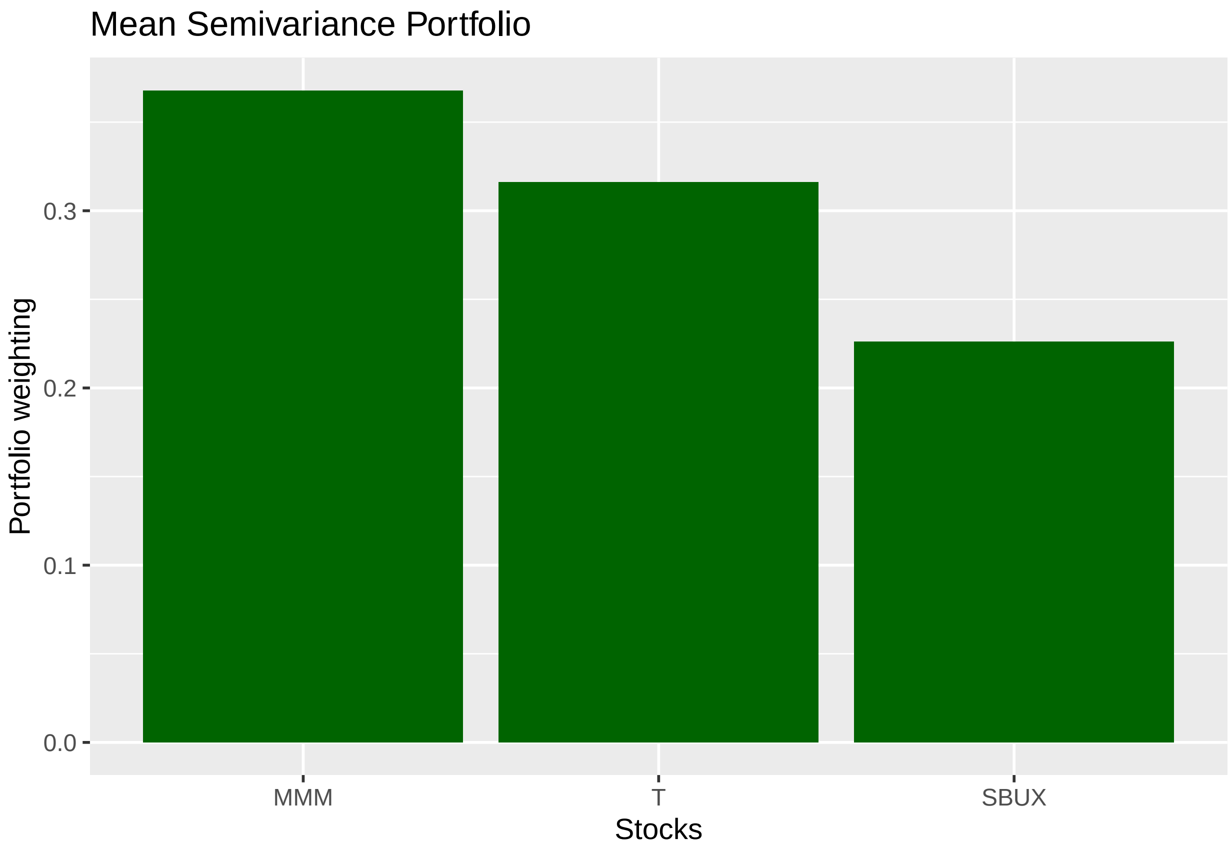
The following is the code we have used to plot the barcharts of the assets with the largest weights in a mean variance portfolio and a mean semivariance portfolio in the statistical methods report. In the following snippet we have taken the liberty of only using 5 stocks and data going back less than a month, this is so that this rmarkdown file renders quickly. In the results of the methods report we have used the whole of the SP500 going back 6 years.

```
source("../R/portfolio_class.R")
source("../R/general_functions.R")

pft1 = portfolio$new(c("AAPL", "SBUX", "T", "MSFT", "A", "MMM"), from = "2021-01-01")
pft2 = portfolio$new(c("AAPL", "SBUX", "T", "MSFT", "A", "MMM"), from = "2021-01-01")

# Create two portfolios using the two different approaches
# w1, w2 will contain the weights of each asset in the portfolios
```

```r
w1 <- pft1$mean_variance_optimize(0.00001, short = FALSE)
w2 <- pft2$mean_semivariance_optimize(0.00001, short = FALSE)

# This function returns a dataframe with the
get_largest_weights <- function(assets, w, n = 10) {
    sorted <- order(w, decreasing = TRUE)[seq_len(n)] # get 10 largest weights
    weights <- w[sorted]
    stocks  <- assets[sorted]
    return(data.frame(stocks, weights))
}


# Plot Barchart for Mean-Variance Portfolio
ggplot() + geom_bar(data = get_largest_weights(pft1$assets, w1, n = 3),
                    aes(x = reorder(stocks, -weights), y = weights),
                    stat = "identity",
                    fill = "steelblue") +
                    ylab("Portfolio weighting") +
                    xlab("Stocks") +
                    ggtitle("Mean Variance Portfolio")
```

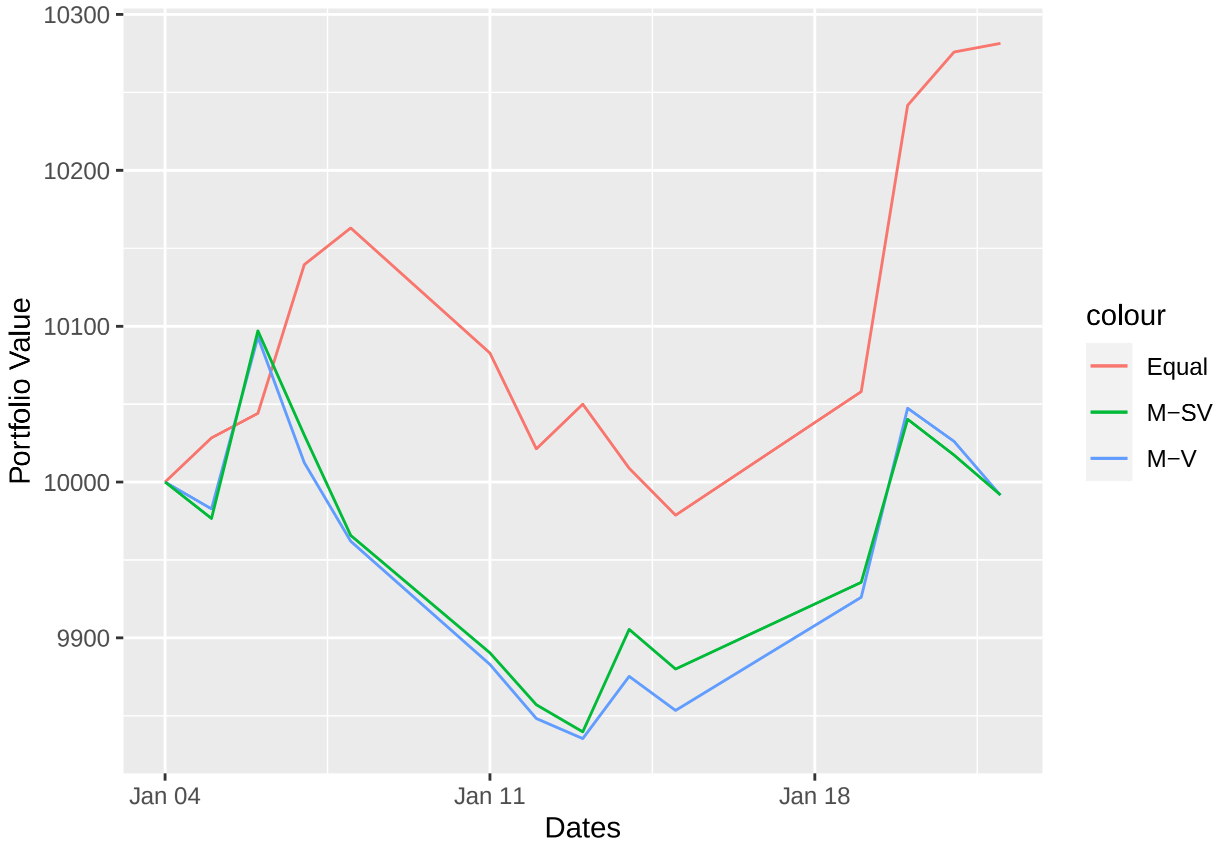


```r
# Plot Barchart for Mean-Semivariance Portfolio
ggplot() + geom_bar(data = get_largest_weights(pft1$assets, w2, n = 3),
                    aes(x = reorder(stocks, -weights), y = weights),
                    stat = "identity",
                    fill = "darkgreen") +
                    ylab("Portfolio weighting") +
                    xlab("Stocks") +
                    ggtitle("Mean Semivariance Portfolio")
```



We can plot the performance of the two portfolios against an equally weighted one as follows.

```
pft3 = portfolio$new(c("AAPL", "SBUX", "T", "MSFT", "A", "MMM"), from = "2021-01-01")

plot_data_equal <- pft3$plot_value(return_values = TRUE)
plot_data_msv <- pft2$plot_value(return_values = TRUE)
plot_data_mv <- pft1$plot_value(return_values = TRUE)
ggplot() + geom_line(data = plot_data_equal, aes(x = date, y = values, color = "Equal")) +
     geom_line(data = plot_data_msv, aes(x = date, y = values, color = "M-V")) +
     geom_line(data = plot_data_mv, aes(x = date, y = values, color = "M-SV")) +
     ylab("Portfolio Value") + xlab("Dates")
```



Testing portfolio on future data

At the end of the statistical methods report for this project we train both a mean variance portfolio and a mean semivariance portfolio on 6 months of historic data and then test the portfolios on the following month of data. The following code snippet does exactly that, but again we use less data here so that the rmarkdown file compiles in a reasonable amount of time.

```
# Matrix to keep track of the three different portfolios
# (M-V, M-SV and equally weighted)
perf <- matrix(0, ncol = 3)

#Starting date
sd = as.Date("2020-01-01")

# Number of days to test for
train_days = 30

# Number of days to train for
test_days = 5

# We repeat the process many times for different training/testing data
for (period in 1:2) {
    # Dates to train portfolios over
    from = (period - 1) * (test_days) + sd
    to   = (period - 1) * (test_days) + train_days + sd
```

```r
    pft1 = portfolio$new(c("AAPL", "SBUX", "T", "MSFT", "A", "MMM"),
                         from = from,
                         to = to)
    pft2 = portfolio$new(c("AAPL", "SBUX", "T", "MSFT", "A", "MMM"),
                         from = from,
                         to = to)

    # Train portfolios
    w1 <- pft1$mean_variance_optimize(0.0001, short = FALSE)
    w2 <- pft2$mean_semivariance_optimize(0.0001, short = FALSE)

    # Make sure weights sum to one, sometimes there is a little error due to
    # optimisation approach
    w1 <- w1/sum(w1)
    w2 <- w2/sum(w2)

    # Make new portfolios just for testing purposes on the
    pft1_test = portfolio$new(c("AAPL", "SBUX", "T", "MSFT", "A", "MMM"),
                         from = to,
                         to = to + test_days,
                         weights = w1)
    pft2_test = portfolio$new(c("AAPL", "SBUX", "T", "MSFT", "A", "MMM"),
                         from = to,
                         to = to + test_days,
                         weights = w2)
    # Equally weighted portfolio for comparison
    pft3_test = portfolio$new(c("AAPL", "SBUX", "T", "MSFT", "A", "MMM"),
                         from = to,
                         to = to + test_days)

    # Calculate the performance of each portfolio over the training period

    eql_values <- pft3_test$plot_value(return_values=TRUE)$values
    mv_values  <- pft1_test$plot_value(return_values=TRUE)$values
    msv_values <- pft2_test$plot_value(return_values=TRUE)$values

    # Performance is percentage change
    equal_perf <- (eql_values[length(eql_values)] - eql_values[1]) / eql_values[1]
    mv_perf <- (mv_values[length(mv_values)] - mv_values[1]) / mv_values[1]
    msv_perf <- (msv_values[length(msv_values)] - msv_values[1]) / msv_values[1]

    perf <- rbind(perf, c(equal_perf, mv_perf, msv_perf))
}

print(perf)
```

```
##              [,1]        [,2]        [,3]
## [1,]  0.000000000 0.000000000 0.000000000
## [2,]  0.022177784 0.020545693 0.018093114
## [3,] -0.001758717 0.003745279 0.001098975
```