Fast Backtest Simulations of Trading Strategies

R/Finance Chicago 2022

Jerzy Pawlowski *jp3900@nyu.edu*

NYU Tandon School of Engineering

June 3, 2022

Rolling Portfolio Simulation

We would like to perform a rolling portfolio simulation: At each *end point* in time, select a portfolio in-sample (based on some model) and measure its performance out-of-sample.

A *rolling aggregation* is performed over a vector of *end points* in time. The *start points* are the *end points* lagged by the *look-back interval*.

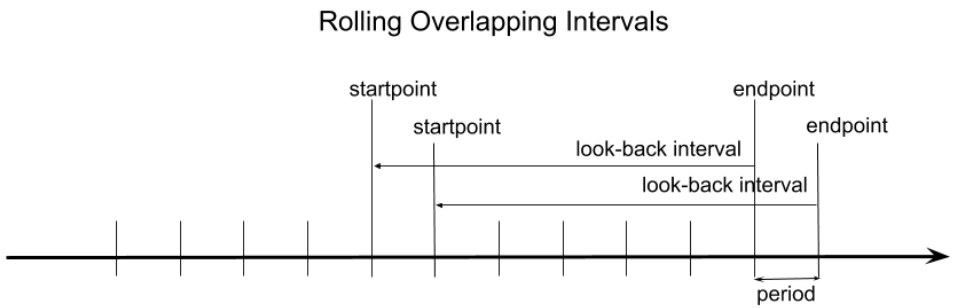
We would like to perform all the matrix algebra and loops in `C++`.

The package *RcppArmadillo* allows calling the high-level *Armadillo* `C++` linear algebra library.

Armadillo is a high-level `C++` linear algebra library, with syntax similar to *Matlab*.

RcppArmadillo functions are often faster than even compiled `R` functions, because they use better optimized `C++` code:
http://arma.sourceforge.net/speed.html

You can learn more about *RcppArmadillo*:
http://arma.sourceforge.net/
http://dirk.eddelbuettel.com/code/rcpp.armadillo.html
https://cran.r-project.org/web/packages/RcppArmadillo/index.html
https://github.com/RcppCore/RcppArmadillo




```
> library(quantmod)
> # Load stock returns
> load("/Users/jerzy/Develop/data/sp500_returns.RData")
> dim(returns)
> # Calculate MSFT percentage returns
> rets <- na.omit(returns$MSFT)
> # Define endp at each point in time
> nrows <- NROW(rets)
> endp <- 1:nrows
> # Start points are multi-period lag of endp
> look_back <- 11
> startp <- c(rep_len(0, look_back), endp[1:(nrows-look_back)])
> head(cbind(startp, endp), 13)
```

Packages for Rolling Aggregations

If possible, always perform loops in C++ and also in *parallel*.

R has several packages for fast rolling aggregations over time series: sums, volatilities, regressions, PCA, etc.

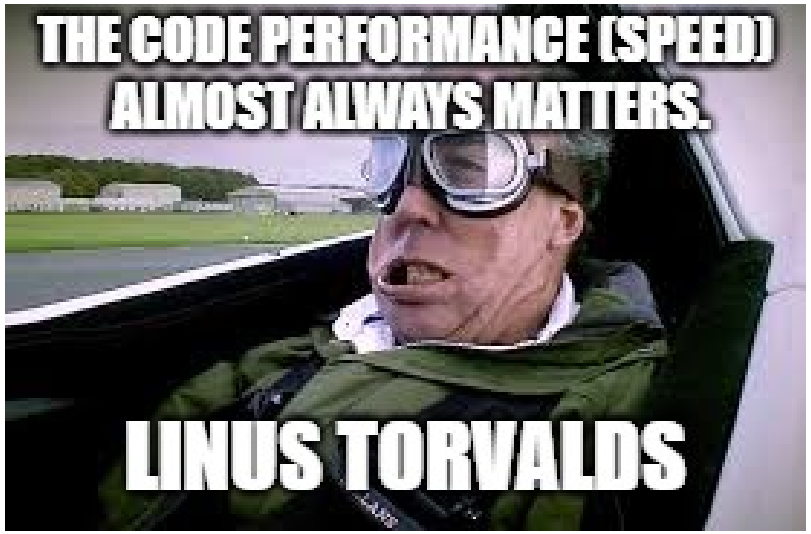
The package *roll* contains functions for calculating *weighted* rolling aggregations over *vectors* and *time series* objects.

The *roll* functions can be 100 times faster than `apply()` loops!

The *roll* functions are extremely fast because they perform calculations in *parallel* in compiled C++ code, using packages *Rcpp*, *RcppArmadillo*, and *RcppParallel*.

The package *RcppRoll* contains functions for calculating *weighted* rolling aggregations over *vectors* and *time series* objects.

The package *matrixStats* contains functions for calculating aggregations over matrix columns and rows, and other matrix computations, such as:




```
> # Calculate rolling MSFT variance using R loop
> varr <- sapply(1:nrows, function(it) {
+   var(rets[startp[it]:endp[it]])
+ })  # end sapply
> # Calculate rolling MSFT variance using package roll
> varcpp <- roll::roll_var(rets, width=(look_back+1))
> all.equal(varr[-(1:look_back)],
+     drop(zoo::coredata(varcpp))[-(1:look_back)])
> # Benchmark calculation of rolling variance
> library(microbenchmark)
> summary(microbenchmark(
+   sapply=sapply(1:nrows, function(it) {
+     var(rets[startp[it]:endp[it]])}),
+   roll=roll::roll_var(rets, width=(look_back+1)),
+   times=10))[, c(1, 4, 5)]
```

Maximum Sharpe Markowitz Portfolio

The *Sharpe* ratio is equal to the ratio of weighted excess returns $\mathbf{w}^T \mu$ divided by the portfolio standard deviation $\sigma = \sqrt{\mathbf{w}^T \mathbb{C} \mathbf{w}}$:

$$SR = \frac{\mathbf{w}^T \mu}{\sigma}$$

Where $\mathbf{w}$ are the portfolio weights and $\mathbb{C}$ is the covariance matrix of returns.

We can calculate the *maximum Sharpe* (Markowitz) portfolio weights by setting the derivative of the *Sharpe* ratio with respect to the weights to zero:

$$\mathbf{w} = \frac{\mathbb{C}^{-1} \mu}{\mathbb{1}^T \mathbb{C}^{-1} \mu}$$

The Markowitz weights are proportional to the inverse of the covariance matrix times the excess returns.

But the returns are difficult to forecast so they have large confidence intervals.

In addition, the covariance matrix of highly correlated returns is either singular or is close to singular, with a large number of very small eigenvalues.

Therefore the Markowitz formula is a noise amplification scheme. See the package *MarkowitzR* for an excellent discussion.

```
> # Select returns after 2000 and overwrite NA values
> rets <- returns["2000/"]
> nstocks <- NCOL(rets)
> rets[1, is.na(rets[1, ])] <- 0
> rets <- zoo::na.locf(rets, na.rm=FALSE)
> dates <- zoo::index(rets)
> # Calculate excess returns
> riskf <- 0.03/252
> retsx <- (rets - riskf)
> # Calculate covariance and inverse matrix
> covmat <- cov(rets)
> # Error: system is singular!
> covinv <- solve(a=covmat)
> # Calculate the Moore-Penrose generalized inverse
> covinv <- MASS::ginv(covmat)
> # Weights of maximum Sharpe portfolio
> weightv <- drop(covinv %*% colMeans(retsx))
> # Combine equal weight and optimal returns
> indeks <- xts::xts(rowMeans(rets), dates)
> pnls <- rets %*% weightv
> pnls <- pnls*sd(indeks)/sd(pnls)
> pnls <- cbind(indeks, pnls)
> colnames(pnls) <- c("Equal Weight", "Optimal")
> # Calculate the out-of-sample Sharpe and Sortino ratios
> sqrt(252)*sapply(pnls,
+   function(x) c(Sharpe=mean(x)/sd(x), Sortino=mean(x)/sd(x[x<0])))
> # Plot the cumulative portfolio returns
> endp <- rutils::calc_endpoints(pnls, interval="months")
> dygraphs::dygraph(cumsum(pnls)[endp], main="Optimal Portfolio Retu
+   dyOptions(colors=c("blue", "red"), strokeWidth=2) %>%
+   dyLegend(width=500)
```

Minimum Variance Portfolio

To avoid the problem of forecasting the returns, we may choose to select a *minimum variance* portfolio.

If the portfolio weights $\mathbf{w}$ are subject to *linear* constraints: $\mathbf{w}^T \mathbb{1} = \sum_{i=1}^{n} w_i = 1$, then the weights that minimize the portfolio variance ($\mathbf{w}^T \mathbb{C}\, \mathbf{w}$) can be found by minimizing the *Lagrangian*:

$$\mathcal{L} = \mathbf{w}^T \mathbb{C}\, \mathbf{w} - \lambda (\mathbf{w}^T \mathbb{1} - 1)$$

Where λ is a *Lagrange multiplier*.

The *minimum variance* portfolio weights are equal to:

$$\mathbf{w} = \frac{\mathbb{C}^{-1} \mathbb{1}}{\mathbb{1}^T \mathbb{C}^{-1} \mathbb{1}}$$

The *minimum variance* portfolio weights are proportional to the inverse of the covariance matrix of returns times the unit vector $\mathbb{1}$.

So the *minimum variance* portfolio still requires the regularization of the inverse covariance matrix.

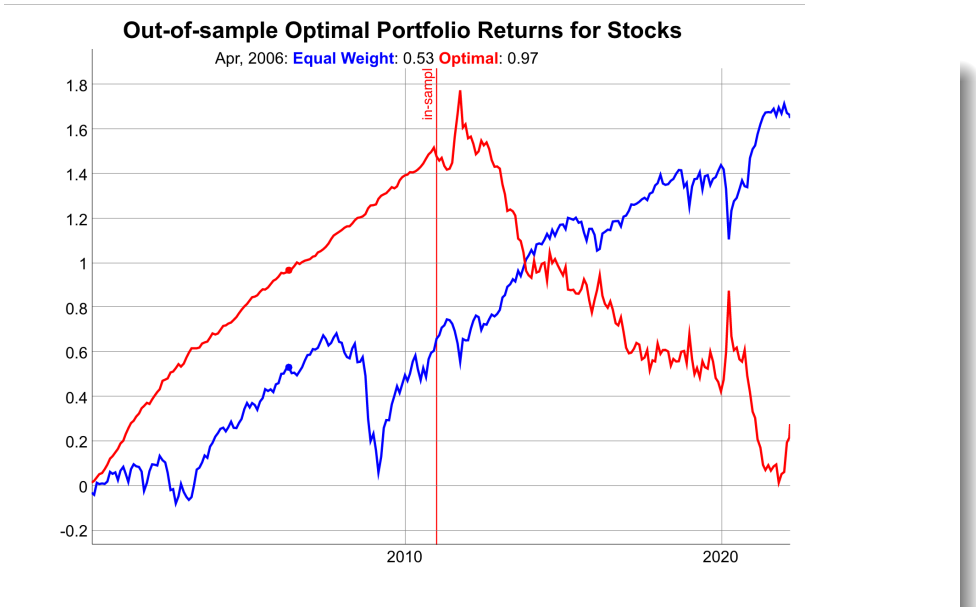
```
> # Weights of minimum variance portfolio
> weightv <- drop(covinv %*% rep(1, nstocks))
> # Combine equal weight and optimal returns
> pnls <- rets %*% weightv
> pnls <- pnls*sd(indeks)/sd(pnls)
> pnls <- cbind(indeks, pnls)
> colnames(pnls) <- c("Equal Weight", "Minimum Variance")
> # Calculate the out-of-sample Sharpe and Sortino ratios
> sqrt(252)*sapply(pnls,
+   function(x) c(Sharpe=mean(x)/sd(x), Sortino=mean(x)/sd(x[x<0])))
> # Plot the cumulative portfolio returns
> endp <- rutils::calc_endpoints(pnls, interval="months")
> dygraphs::dygraph(cumsum(pnls)[endp], main="Minimum Variance Portf
+   dyOptions(colors=c("blue", "red"), strokeWidth=2) %>%
+   dyLegend(width=500)
```

Markowitz Portfolio Out-of-Sample

The Markowitz portfolio is *overfit* in the *in-sample* interval.

Therefore the portfolio underperforms in the *out-of-sample* interval.

The Markowitz portfolio underperforms *out-of-sample* because it has too many degrees of freedom, and also because the covariance matrix has many very small eigenvalues.




```
> # Define in-sample and out-of-sample intervals
> retsis <- rets["/2010"]
> retsos <- rets["2011/"]
> # Maximum Sharpe weights in-sample interval
> covmat <- cov(retsis)
> covinv <- MASS::ginv(covmat)
> weightv <- covinv %*% colMeans(retsx["/2010"])
> names(weightv) <- colnames(rets)
> # Calculate portfolio returns
> insample <- xts::xts(retsis %*% weightv, zoo::index(retsis))
> outsample <- xts::xts(retsos %*% weightv, zoo::index(retsos))
> indeks <- xts::xts(rowMeans(rets), dates)
```

```
> # Combine in-sample and out-of-sample returns
> pnls <- rbind(insample, outsample)
> pnls <- pnls*sd(indeks)/sd(pnls)
> pnls <- cbind(indeks, pnls)
> colnames(pnls) <- c("Equal Weight", "Optimal")
> # Calculate the out-of-sample Sharpe and Sortino ratios
> sqrt(252)*sapply(pnls["2011/"],
+   function(x) c(Sharpe=mean(x)/sd(x), Sortino=mean(x)/sd(x[x<0])))
> # Plot the cumulative portfolio returns
> endp <- rutils::calc_endpoints(pnls, interval="months")
> dygraphs::dygraph(cumsum(pnls)[endp], main="Out-of-sample Optimal
+   dyOptions(colors=c("blue", "red"), strokeWidth=2) %>%
+   dyEvent(zoo::index(last(retsis[, 1])), label="in-sample", stroke
+   dyLegend(width=500)
```

Packages for Portfolio Selection and Optimization

We need to apply some form of regularization to the Markowitz portfolio, such as dimension reduction or shrinkage.

There is good research on portfolio optimization using regularization, dimension reduction, and shrinkage estimators of covariance matrices by *Victor DeMiguel*, *Gianluca De Nard*, and *Olivier Ledoit*.

There are also good `R` packages for portfolio optimization: *RiskPortfolios*, *parma*, *PMwR*, *NMOF*, *fPortfolio*, *MarkowitzR*,

Unfortunately, most of these packages are written in `R` so they can't be called in a `C++` loop. So we'll have to write our own code in `C++`.

So the objective is to: combine modern portfolio analytics, rolling optimization, all written in Armadillo code.

There are also good `R` packages and research on portfolio selection: *riskParityPortfolio*, *Hierarchical Risk Parity*,

Markowitz Portfolio With *Parameter Shrinkage*

The regularization technique of *parameter shrinkage* is designed to reduce the number of parameters in a model, for example in portfolio optimization.

The *parameter shrinkage* technique adds a penalty term to the objective function.

The *elastic net* regularization is a combination of *ridge* and *Lasso* regularization:

$$w_{max} = \arg\max_{w}[\frac{\mathbf{w}^T \mu}{\sigma} - \lambda((1-\alpha)\sum_{i=1}^{n} w_i^2 + \alpha \sum_{i=1}^{n} |w_i|)]$$

The portfolio weights **w** are shrunk to zero as the parameters λ and α increase.

The function `DEoptim()` from package *DEoptim* performs *global* optimization using the *Differential Evolution* algorithm.

But the optimization can be very time consuming.

```
> # Objective with shrinkage penalty
> objfun <- function(weightv, rets, lambda, alpha) {
+   rets <- rets %*% weightv
+   if (sd(rets) == 0)
+     return(0)
+   else {
+     penaltyv <- lambda*((1-alpha)*sum(weightv^2) +
+ alpha*sum(abs(weightv)))
+     -return(mean(rets)/sd(rets) + penaltyv)
+   }
+ }  # end objfun
> # Objective for equal weight portfolio
> weightv <- rep(1, nstocks)
> lambda <- 0.1 ; alpha <- 0.1
> objfun(weightv, rets=rets, lambda=lambda, alpha=alpha)
> # Perform optimization using DEoptim - takes very long on M1 Mac!
> optiml <- DEoptim::DEoptim(fn=objfun,
+   upper=rep(10, nstocks),
+   lower=rep(-10, nstocks),
+   rets=rets,
+   lambda=lambda, alpha=alpha,
+   control=list(trace=FALSE, itermax=100, parallelType=1, packages=
> weightv <- optiml$optim$bestmem/sum(abs(optiml$optim$bestmem))
> weightv <- weightv/sum(weightv)
> names(weightv) <- colnames(rets)
> # Combine equal weight and optimal returns
> pnls <- rets %*% weightv
> pnls <- pnls*sd(indeks)/sd(pnls)
> pnls <- cbind(indeks, pnls)
> colnames(pnls) <- c("Equal Weight", "Optimal")
> # Calculate the out-of-sample Sharpe and Sortino ratios
> sqrt(252)*sapply(pnls,
+   function(x) c(Sharpe=mean(x)/sd(x), Sortino=mean(x)/sd(x[x<0])))
> # Plot the cumulative portfolio returns
> endp <- rutils::calc_endpoints(pnls, interval="months")
> dygraphs::dygraph(cumsum(pnls)[endp], main="Optimal Portfolio Retu
+   dyOptions(colors=c("blue", "red"), strokeWidth=2) %>%
+   dyLegend(width=500)
```

Regularized Inverse of the Covariance Matrix

The covariance matrix of returns $\mathbb{C}$ can be expressed as a product of its *eigenvalues* $\mathbb{D}$ and its *eigenvectors* $\mathbb{O}$:

$$\mathbb{C} = \mathbb{O}\,\mathbb{D}\,\mathbb{O}^T$$

The inverse of the covariance matrix can then be calculated from its *eigen decomposition* as:

$$\mathbb{C}^{-1} = \mathbb{O}\,\mathbb{D}^{-1}\,\mathbb{O}^T$$

If the number of time periods of returns (rows) is less than the number of stocks (columns), then some of the higher order eigenvalues are zero, and the above covariance matrix inverse is singular.

The *regularized inverse* $\mathbb{C}_n^{-1}$ is calculated by removing the zero eigenvalues, and keeping only the first n *eigenvalues*:

$$\mathbb{C}_n^{-1} = \mathbb{O}_n\,\mathbb{D}_n^{-1}\,\mathbb{O}_n^T$$

Where $\mathbb{D}_n$ and $\mathbb{O}_n$ are matrices with the higher order eigenvalues and eigenvectors removed.

The function `MASS::ginv()` calculates the *regularized* inverse of a matrix.

```
> # Calculate covariance matrix
> covmat <- cov(rets)
> # Calculate inverse of covmat - error
> covinv <- solve(covmat)
> # Perform eigen decomposition
> eigend <- eigen(covmat)
> eigenvec <- eigend$vectors
> eigenval <- eigend$values
> # Set tolerance for determining zero singular values
> precv <- sqrt(.Machine$double.eps)
> # Calculate regularized inverse matrix
> notzero <- (eigenval > (precv * eigenval[1]))
> invreg <- eigenvec[, notzero] %*%
+   (t(eigenvec[, notzero]) / eigenval[notzero])
> # Verify inverse property of invreg
> all.equal(covmat, covmat %*% invreg %*% covmat)
> # Calculate regularized inverse of covmat
> covinv <- MASS::ginv(covmat)
> # Verify that covinv is same as invreg
> all.equal(covinv, invreg)
```

Dimension Reduction of the Covariance Matrix

If the higher order singular values are very small then the inverse matrix amplifies the statistical noise in the response matrix.

The technique of *dimension reduction* calculates the inverse of a covariance matrix by removing the very small, higher order eigenvalues, to reduce the propagation of statistical noise and improve the signal-to-noise ratio:

$$\mathbb{C}_{DR}^{-1} = \mathbb{O}_{dimax}\, \mathbb{D}_{dimax}^{-1}\, \mathbb{O}_{dimax}^{T}$$

The parameter `dimax` specifies the number of eigenvalues used for calculating the *dimension reduction inverse* of the covariance matrix of returns.

Even though the *dimension reduction inverse* $\mathbb{C}_{DR}^{-1}$ does not satisfy the matrix inverse property (so it's biased), its out-of-sample forecasts are usually more accurate than those using the actual inverse matrix.

But removing a larger number of eigenvalues increases the bias of the covariance matrix, which is an example of the *bias-variance tradeoff*.

The optimal value of the parameter `dimax` can be determined using *backtesting* (*cross-validation*).

```
> # Calculate in-sample covariance matrix
> covmat <- cov(retsis)
> eigend <- eigen(covmat)
> eigenvec <- eigend$vectors
> eigenval <- eigend$values
> # Calculate dimension reduction inverse of covariance matrix
> dimax <- 3
> covinv <- eigenvec[, 1:dimax] %*%
+   (t(eigenvec[, 1:dimax]) / eigenval[1:dimax])
> # Verify inverse property of inverse
> all.equal(covmat, covmat %*% covinv %*% covmat)
```

Optimal Portfolios Under Zero Correlation

If the correlations of returns are equal to zero, then the covariance matrix is diagonal:

$$\mathbb{C} = \begin{pmatrix} \sigma_1^2 & 0 & \cdots & 0 \\ 0 & \sigma_2^2 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \sigma_n^2 \end{pmatrix}$$

Where σ_i^2 is the variance of returns of asset `i`.

The inverse of $\mathbb{C}$ is then simply:

$$\mathbb{C}^{-1} = \begin{pmatrix} \sigma_1^{-2} & 0 & \cdots & 0 \\ 0 & \sigma_2^{-2} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \sigma_n^{-2} \end{pmatrix}$$

The *minimum variance* portfolio weights are proportional to the inverse of the individual variances:

$$w_i = \frac{1}{\sigma_i^2 \sum_{i=1}^{n} \sigma_i^{-2}}$$

The *maximum Sharpe* portfolio weights are proportional to the ratio of excess returns divided by the individual variances:

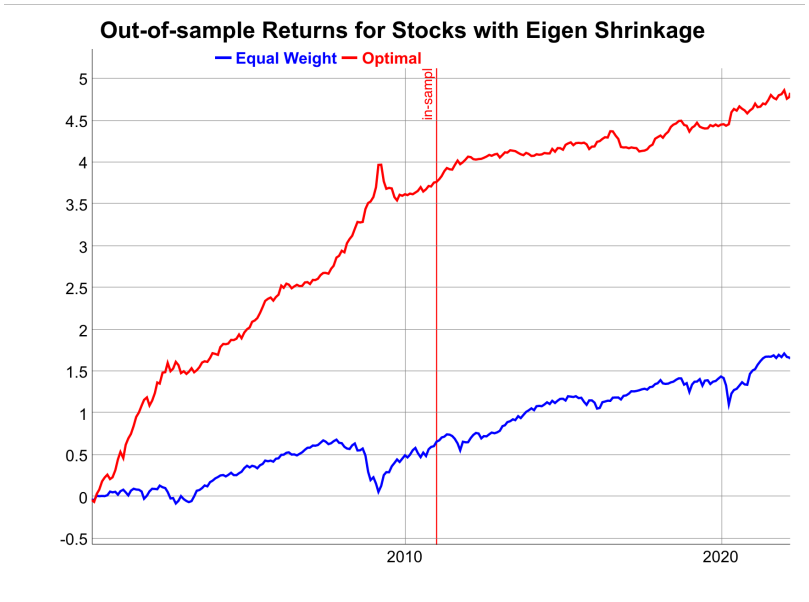
$$w_i = \frac{\mu_i}{\sigma_i^2 \sum_{i=1}^{n} \mu_i \sigma_i^{-2}}$$

Markowitz Portfolio with Dimension Reduction

The *out-of-sample* performance of the Markowitz portfolio is greatly improved by dimension reduction of the inverse covariance matrix.

But the *in-sample* performance is worse because dimension reduction reduces *overfitting*.

```
> # Calculate regularized inverse of covariance matrix
> look_back <- 8; dimax <- 21
> eigend <- eigen(cov(retsis))
> eigenvec <- eigend$vectors
> eigenval <- eigend$values
> covinv <- eigenvec[, 1:dimax] %*%
+   (t(eigenvec[, 1:dimax]) / eigenval[1:dimax])
> # Calculate portfolio weights
> weightv <- covinv %*% colMeans(retsx["/2010"])
> weightv <- drop(weightv/sum(weightv))
> names(weightv) <- colnames(rets)
> # Calculate portfolio returns
> insample <- xts::xts(retsis %*% weightv, zoo::index(retsis))
> outsample <- xts::xts(retsos %*% weightv, zoo::index(retsos))
> indeks <- xts::xts(rowMeans(rets), dates)
```




```
> # Combine in-sample and out-of-sample returns
> pnls <- rbind(insample, outsample)
> pnls <- pnls*sd(indeks)/sd(pnls)
> pnls <- cbind(indeks, pnls)
> colnames(pnls) <- c("Equal Weight", "Optimal")
> # Calculate the out-of-sample Sharpe and Sortino ratios
> sqrt(252)*sapply(pnls["2011/"],
+   function(x) c(Sharpe=mean(x)/sd(x), Sortino=mean(x)/sd(x[x<0])))
> # Plot the cumulative portfolio returns
> endp <- rutils::calc_endpoints(pnls, interval="months")
> dygraphs::dygraph(cumsum(pnls)[endp], main="Out-of-sample Returns
+   dyOptions(colors=c("blue", "red"), strokeWidth=2) %>%
+   dyEvent(zoo::index(last(retsis[, 1])), label="in-sample", stroke
+   dyLegend(width=500)
```

Markowitz Portfolio With Return Shrinkage

To further reduce the statistical noise, the individual returns r_i can be *shrunk* to the average portfolio returns $\bar{r}$:

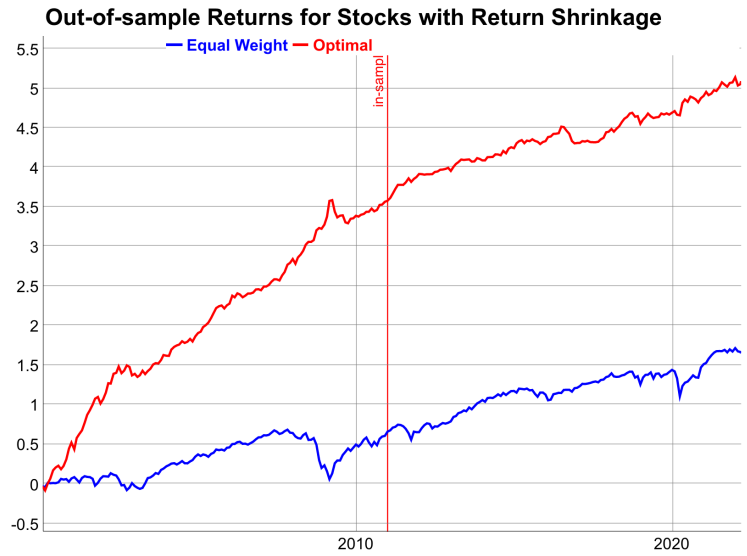
$$r_i' = (1 - \alpha)\, r_i + \alpha\, \bar{r}$$

The parameter α is the *shrinkage* intensity, and it determines the strength of the *shrinkage* of individual returns to their mean.

The weights with return shrinkage are an average of *maximum Sharpe* and *minimum variance* weights.

If $\alpha = 0$ then there is no *shrinkage*, while if $\alpha = 1$ then all the returns are *shrunk* to their common mean: $r_i = \bar{r}$.

The optimal value of the *shrinkage* intensity α can be determined using *backtesting* (*cross-validation*).




```
> # Shrink the in-sample returns to their mean
> alpha <- 0.7
> retsxm <- rowMeans(retsx["/2010"])
> retsxis <- (1-alpha)*retsx["/2010"] + alpha*retsxm
> # Calculate portfolio weights
> weightv <- covinv %*% colMeans(retsxis)
> weightv <- drop(weightv/sum(weightv))
> # Calculate portfolio returns
> insample <- xts::xts(retsis %*% weightv, zoo::index(retsis))
> outsample <- xts::xts(retsos %*% weightv, zoo::index(retsos))
```

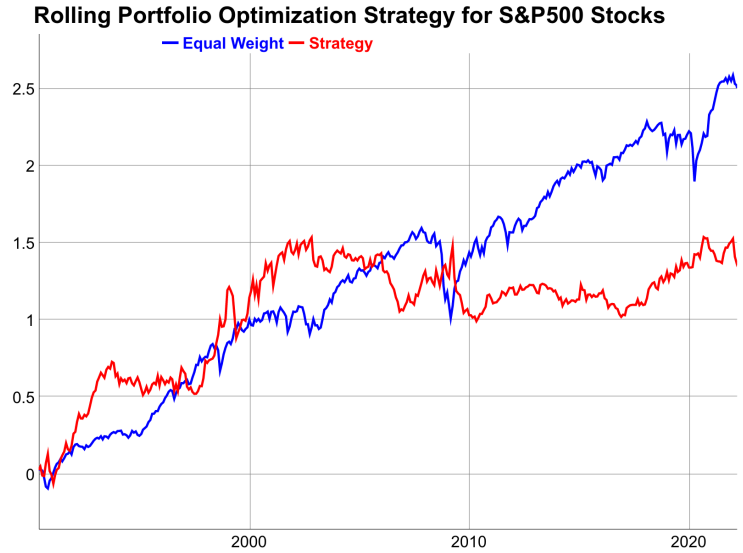
```
> # Combine in-sample and out-of-sample returns
> pnls <- rbind(insample, outsample)
> pnls <- pnls*sd(indeks)/sd(pnls)
> pnls <- cbind(indeks, pnls)
> colnames(pnls) <- c("Equal Weight", "Optimal")
> # Calculate the out-of-sample Sharpe and Sortino ratios
> sqrt(252)*sapply(pnls["2011/"],
+   function(x) c(Sharpe=mean(x)/sd(x), Sortino=mean(x)/sd(x[x<0])))
> # Plot the cumulative portfolio returns
> dygraphs::dygraph(cumsum(pnls)[endp], main="Out-of-sample Returns
+   dyOptions(colors=c("blue", "red"), strokeWidth=2) %>%
+   dyEvent(zoo::index(last(retsis[, 1])), label="in-sample", stroke
+   dyLegend(width=500)
```

Rolling Portfolio Optimization Strategy for S&P500 Stocks

A *rolling portfolio optimization* strategy consists of rebalancing a portfolio over the end points:

1. Calculate the maximum Sharpe ratio portfolio weights at each end point,
2. Apply the weights in the next interval and calculate the out-of-sample portfolio returns.

The strategy parameters are: the rebalancing frequency (annual, monthly, etc.), and the length of look-back interval.




```
> # Overwrite NA values in returns100
> rets <- returns100
> rets[1, is.na(rets[1, ])] <- 0
> rets <- zoo::na.locf(rets, na.rm=FALSE)
> retsx <- (rets - riskf)
> nstocks <- NCOL(rets) ; dates <- zoo::index(rets)
> # Define monthly end points
> endp <- rutils::calc_endpoints(rets, interval="months")
> endp <- endp[endp > (nstocks+1)]
> npts <- NROW(endp) ; look_back <- 12
> startp <- c(rep_len(0, look_back), endp[1:(npts-look_back)])
> # Perform loop over end points - takes very long !!!
> pnls <- lapply(2:npts, function(ep) {
+     # Subset the excess returns
+     insample <- retsx[startp[ep-1]:endp[ep-1], ]
+     covinv <- MASS::ginv(cov(insample))
+     # Calculate the maximum Sharpe ratio portfolio weights
+     weightv <- covinv %*% colMeans(insample)
+     weightv <- drop(weightv/sum(weightv))
+     # Calculate the out-of-sample portfolio returns
+     outsample <- rets[(endp[ep-1]+1):endp[ep], ]
+     xts::xts(outsample %*% weightv, zoo::index(outsample))
+ })  # end lapply
```

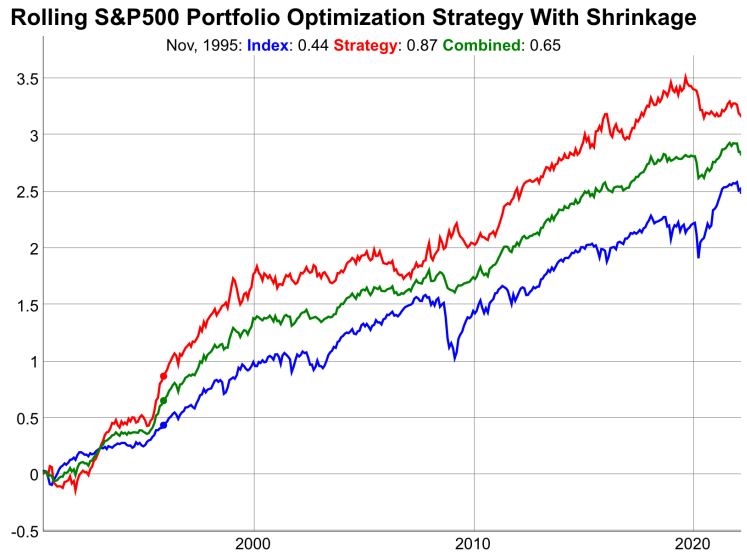
```
> # Calculate returns of equal weight portfolio
> indeks <- xts::xts(rowMeans(rets), dates)
> pnls <- rbind(indeks[paste0("/", start(pnls)-1)], pnls*sd(indeks)/
> # Calculate the Sharpe and Sortino ratios
> wealth <- cbind(indeks, pnls)
> colnames(wealth) <- c("Equal Weight", "Strategy")
> sqrt(252)*sapply(wealth,
+   function(x) c(Sharpe=mean(x)/sd(x), Sortino=mean(x)/sd(x[x<0])))
> # Plot cumulative strategy returns
> dygraphs::dygraph(cumsum(wealth)[endp], main="Rolling Portfolio Op
+   dyOptions(colors=c("blue", "red"), strokeWidth=2) %>%
+   dyLegend(show="always", width=500)
```

Rolling Portfolio Optimization Strategy in C++

The *rolling portfolio optimization* strategy can be improved by applying both dimension reduction and return shrinkage.

The function `back_test()` from package *HighFreq* performs backtest simulations of trading strategies in C++.

```
> # Shift end points to C++ convention
> endp <- (endp - 1)
> endp[endp < 0] <- 0
> startp <- (startp - 1)
> startp[startp < 0] <- 0
> # Specify dimension reduction and return shrinkage
> alpha <- 0.7
> dimax <- 21
> # Perform backtest in Rcpp
> pnls <- HighFreq::back_test(excess=retsx, returns=rets,
+   startp=startp, endp=endp, alpha=alpha, dimax=dimax, method="ma
> pnls <- pnls*sd(indeks)/sd(pnls)
```




```
> # Plot cumulative strategy returns
> wealth <- cbind(indeks, pnls, (pnls+indeks)/2)
> colnames(wealth) <- c("Index", "Strategy", "Combined")
> # Calculate the out-of-sample Sharpe and Sortino ratios
> sqrt(252)*sapply(wealth,
+   function(x) c(Sharpe=mean(x)/sd(x), Sortino=mean(x)/sd(x[x<0])))
> dygraphs::dygraph(cumsum(wealth)[endp], main="Rolling S&P500 Portf
+   dyOptions(colors=c("blue", "red", "green"), strokeWidth=2) %>%
+   dyLegend(show="always", width=500)
```

Strategy Backtesting Using *RcppArmadillo*

```
arma::mat back_test(const arma::mat& excess, // Asset excess returns
                    const arma::mat& returns, // Asset returns
                    arma::uvec startp, // Start points
                    arma::uvec endp, // End points
                    double lambda = 0.0, // Decay factor for averaging the portfolio weights
                    std::string method = "sharpem", // Method for calculating the weights
                    double eigen_thresh = 1e-5, // Threshold level for discarding small singular values
                    arma::uword dimax = 0, // Regularization intensity
                    double confl = 0.1, // Confidence level for calculating the quantiles of returns
                    double alpha = 0.0, // Return shrinkage intensity
                    bool rankw = false, // Rank the weights
                    bool centerw = false, // Center the weights
                    std::string scalew = "voltarget", // Method for scaling the weights
                    double vol_target = 0.01, // Target volatility for scaling the weights
                    double coeff = 1.0, // Multiplier strategy returns
                    double bid_offer = 0.0) {

  double lambda1 = 1-lambda;
  arma::uword nweights = returns.n_cols;
  arma::vec weights(nweights, fill::zeros);
  arma::vec weights_past = ones(nweights)/sqrt(nweights);
  arma::mat pnls = zeros(returns.n_rows, 1);

  // Perform loop over the end points
  for (arma::uword it = 1; it < endp.size(); it++) {
    // cout << "it: " << it << endl;
    // Calculate portfolio weights
    weights = coeff*calc_weights(excess.rows(startp(it-1), endp(it-1)), method, eigen_thresh, dimax,
                                 confl, alpha, rankw, centerw, scalew, vol_target);
    // Calculate the weights as the weighted sum with past weights
    weights = lambda1*weights + lambda*weights_past;
    // Calculate out-of-sample returns
    pnls.rows(endp(it-1)+1, endp(it)) = returns.rows(endp(it-1)+1, endp(it))*weights;
    // Add transaction costs
    pnls.row(endp(it-1)+1) -= bid_offer*sum(abs(weights - weights_past))/2;
    // Copy the weights
    weights_past = weights;
  }  // end for
```

Portfolio Optimization Using *RcppArmadillo*

Fast portfolio optimization using matrix algebra can be implemented using *RcppArmadillo*.

```
arma::vec calc_weights(const arma::mat& returns, // Asset returns
                       std::string method = "maxsharpe", // Method for calculating the weights
                       double eigen_thresh = 1e-5, // Threshold level for discarding small singular values
                       arma::uword dimax = 0, // Regularization intensity
                       double confl = 0.1, // Confidence level for calculating the quantiles of returns
                       double alpha = 0.0, // Return shrinkage intensity
                       bool rankw = false, // Rank the weights
                       bool centerw = false, // Center the weights
                       std::string scalew = "voltarget", // Method for scaling the weights
                       double vol_target = 0.01) { // Target volatility for scaling the weights

  // Initialize
  arma::uword ncols = returns.n_cols;
  arma::vec weights(ncols, fill::zeros);
  // No regularization so set dimax to ncols
  if (dimax == 0)  dimax = ncols;
  // Calculate the covariance matrix
  arma::mat covmat = calc_covar(returns);

  // Apply different calculation methods for weights
  switch(calc_method(method)) {
  case methodenum::maxsharpe: {
    // Mean returns of columns
    arma::vec colmeans = arma::trans(arma::mean(returns, 0));
    // Shrink colmeans to the mean of returns
    colmeans = ((1-alpha)*colmeans + alpha*arma::mean(colmeans));
    // Calculate weights using regularized inverse
    weights = calc_inv(covmat, eigen_thresh, dimax)*colmeans;
    break;
  }  // end maxsharpe
  case methodenum::maxsharpemed: {
    // Median returns of columns
    arma::vec colmeans = arma::trans(arma::median(returns, 0));
    // Shrink colmeans to the median of returns
    colmeans = ((1-alpha)*colmeans + alpha*arma::median(colmeans));
```

Fast Covariance Matrix Inverse Using *RcppArmadillo*

RcppArmadillo can be used to quickly calculate the regularized inverse of a covariance matrix.

The function `calc_inv()` from package *HighFreq* calculates the regularized inverse of matrices in C++.

```
> # Regularized inverse of covariance matrix
> dimax <- 4
> eigend <- eigen(covmat)
> covinv <- eigend$vectors[, 1:dimax] %*%
+   (t(eigend$vectors[, 1:dimax]) / eigend$values[1:dimax])
> # Regularized inverse using RcppArmadillo
> covinv_arma <- calc_inv(covmat, dimax)
> all.equal(covinv, covinv_arma)
> # Microbenchmark RcppArmadillo code
> library(microbenchmark)
> summary(microbenchmark(
+   rcode={eigend <- eigen(cov(covmat))
+     eigend$vectors[, 1:dimax] %*%
+ (t(eigend$vectors[, 1:dimax]) / eigend$values[1:dimax])
+   },
+   cppcode=calc_inv(covmat, dimax),
+   times=100))[, c(1, 4, 5)]  # end microbenchmark summary
```

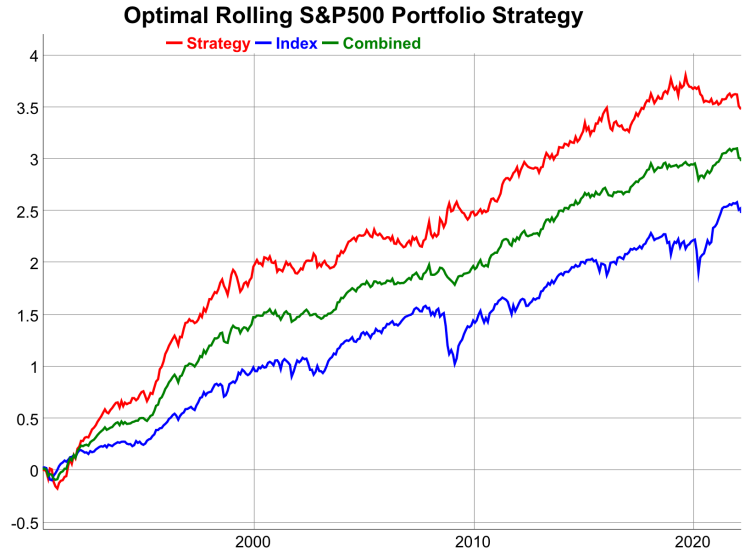
```
arma::mat calc_inv(const arma::mat& tseries,
                   double eigen_thresh = 0.01,
                   arma::uword dimax = 0) {

  // Allocate SVD variables
  arma::vec svdval;  // Singular values
  arma::mat svdu, svdv;  // Singular matrices
  // Calculate the SVD
  arma::svd(svdu, svdval, svdv, tseries);
  // Calculate the number of non-small singular values
  arma::uword svdnum = arma::sum(svdval > eigen_thresh*a

  if (dimax == 0) {
    // Set dimax
    dimax = svdnum - 1;
  } else {
    // Adjust dimax
    dimax = std::min(dimax - 1, svdnum - 1);
  }  // end if

  // Remove all small singular values
  svdval = svdval.subvec(0, dimax);
  svdu = svdu.cols(0, dimax);
  svdv = svdv.cols(0, dimax);

  // Calculate the regularized inverse from the SVD deco
  return svdv*arma::diagmat(1/svdval)*svdu.t();

}  // end calc_inv
```

Determining Regularization Parameters Using Backtesting

The optimal values of the dimension reduction parameter dimax and the return shrinkage intensity parameter α can be determined using *backtesting*.

The best dimension reduction parameter for this portfolio of stocks is equal to dimax=33, which means relatively weak dimension reduction.

The best return shrinkage parameter for this portfolio of stocks is equal to $\alpha = 0.81$, which means strong return shrinkage.



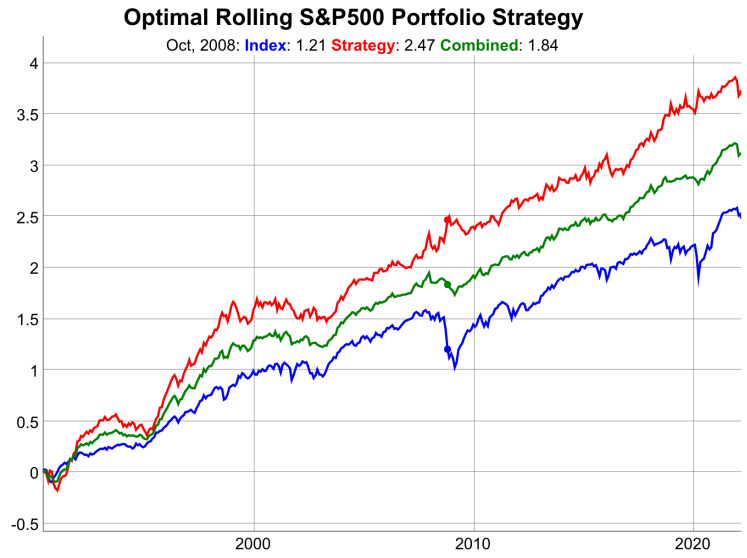

```
> # Perform backtest over vector of return shrinkage intensities
> alphav <- seq(from=0.01, to=0.91, by=0.1)
> pnls <- lapply(alphav, function(alpha) {
+   HighFreq::back_test(excess=retsx, returns=rets,
+   startp=startp, endp=endp, alpha=alpha, dimax=dimax, method="max
+ })  # end lapply
> profilev <- sapply(pnls, sum)
> plot(x=alphav, y=profilev, t="l", main="Rolling Strategy as Funct
+   xlab="Shrinkage Intensity Alpha", ylab="pnl")
> whichmax <- which.max(profilev)
> alpha <- alphav[whichmax]
> pnls <- pnls[[whichmax]]
> # Perform backtest over vector of dimension reduction eigenvals
> eigenvals <- seq(from=3, to=40, by=2)
> pnls <- lapply(eigenvals, function(dimax) {
+   HighFreq::back_test(excess=retsx, returns=rets,
+     startp=startp, endp=endp, alpha=alpha, dimax=dimax, method="maxsharpe")
+ })  # end lapply
> profilev <- sapply(pnls, sum)
> plot(x=eigenvals, y=profilev, t="l", main="Strategy PnL as Function of dimax",
+   xlab="dimax", ylab="pnl")
> whichmax <- which.max(profilev)
> dimax <- eigenvals[whichmax]
> pnls <- pnls[[whichmax]]
```

```
> # Plot cumulative strategy returns
> wealth <- cbind(indeks, pnls, (pnls+indeks)/2)
> colnames(wealth) <- c("Index", "Strategy", "Combined")
> # Calculate the out-of-sample Sharpe and Sortino ratios
> sqrt(252)*sapply(wealth,
+   function(x) c(Sharpe=mean(x)/sd(x), Sortino=mean(x)/sd(x[x<0])))
> dygraphs::dygraph(cumsum(wealth)[endp], main="Optimal Rolling S&P5
+   dyOptions(colors=c("blue", "red", "green"), strokeWidth=2) %>%
+   dyLegend(show="always", width=500)
```

Determining Look-back Interval Using Backtesting

The optimal value of the look-back interval can be determined using *backtesting*.

The optimal value of the look-back interval for this portfolio of stocks is equal to `look_back=9` months, which roughly agrees with the research literature on momentum strategies.




```
> # Perform backtest over look-backs
> look_backs <- seq(from=3, to=12, by=1)
> pnls <- lapply(look_backs, function(look_back) {
+   startp <- c(rep_len(0, look_back), endp[1:(npts-look_back)])
+   startp <- (startp - 1)
+   startp[startp < 0] <- 0
+   HighFreq::back_test(excess=retsx, returns=rets,
+     startp=startp, endp=endp, alpha=alpha, dimax=dimax, method="maxsharpe")
+ })  # end lapply
> profilev <- sapply(pnls, sum)
> plot(x=look_backs, y=profilev, t="l", main="Strategy PnL as Funct
+   xlab="Look-back Interval", ylab="pnl")
> whichmax <- which.max(profilev)
> look_back <- look_backs[whichmax]
> pnls <- pnls[[whichmax]]
> pnls <- pnls*sd(indeks)/sd(pnls)
```

```
> # Plot cumulative strategy returns
> wealth <- cbind(indeks, pnls, (pnls+indeks)/2)
> colnames(wealth) <- c("Index", "Strategy", "Combined")
> # Calculate the out-of-sample Sharpe and Sortino ratios
> sqrt(252)*sapply(wealth,
+   function(x) c(Sharpe=mean(x)/sd(x), Sortino=mean(x)/sd(x[x<0])))
> dygraphs::dygraph(cumsum(wealth)[endp], main="Optimal Rolling S&P5
+   dyOptions(colors=c("blue", "red", "green"), strokeWidth=2) %>%
+   dyLegend(show="always", width=500)
```

Fast PCA Using Package RSpectra

The dimension reduction calculations can be greatly accelerated by performing *partial eigen decomposition* and calculating only the largest eigenvalues.

The function `eigs_sym()` from package *RSpectra* performs partial eigen decomposition of matrices in `C++`.

It calculates the *partial eigen decomposition* using the *Implicitly Restarted Lanczos Method* (IRLM).

The function `eigs_sym()` can be `5` times faster than `eigen()`!

Spectra is a `C++` library for the efficient solving of large scale eigenvalue problems, using the *Eigen* `C++` linear algebra library.

The Spectra library is a `C++` version of the algorithms in the *ARPACK* library.

The ARPACK is a `FORTRAN` library for solving large scale eigenvalue problems.

```
> # Perform eigen decomposition using eigen()
> eigend <- eigen(covmat)
> eigenvec <- eigend$vectors
> eigenval <- eigend$values
> # Perform eigen decomposition using RSpectra
> library(RSpectra)
> dimax <- 5
> eigends = RSpectra::eigs_sym(covmat, k=dimax, which="LM")
> all.equal(eigends$values, eigenval[1:dimax])
> all.equal(abs(eigends$vectors), abs(eigenvec[, 1:dimax]))
> # Compare the speed of eigen() with RSpectra
> library(microbenchmark)
> summary(microbenchmark(
+   eigen=eigen(covmat),
+   RSpectra=RSpectra::eigs_sym(covmat, k=dimax, which="LM"),
+   times=10))[, c(1, 4, 5)]  # end microbenchmark summary
```

Fast Approximate PCA Using Stochastic Gradient Optimization

It's not necessary to perform PCA from scratch at every point in time because the data changes by only a small amount between time periods.

Online PCA algorithms update the PCA with new data, instead of performing PCA from scratch.

The Generalized Hebbian Algorithm (GHA) updates the eigenvectors $\mathbf{u}_t^j$ based on the new data vector $\mathbf{x}_{t+1}$:

$$\mathbf{u}_{t+1}^j = \mathbf{u}_t^j + \gamma \varphi_t^j (\mathbf{x}_{t+1} - \sum_{i=1}^{j} \varphi_t^i \mathbf{u}_t^i)$$

Where the coefficients φ_t^j are the inner products of the new data vector times the eigenvectors:

$$\varphi_t^j = \mathbf{x}_{t+1}^T \mathbf{u}_t^j$$

Where γ is the learning rate (a number between 0 and 1), with larger values placing more weight on the new data (faster learning).

The GHA algorithm was developed as a neural network learning technique. It updates the principal components only approximately using recursion.

The function `ghapca()` from package *onlinePCA* performs approximate PCA of matrices in `C++`.

```
> # Setup data
> ncols <- 10
> nrows <- 1e4
> matv <- matrix(runif(nrows*ncols), nrows, ncols)
> matv <- matv %*% diag(sqrt(12*(1:ncols)))
> # Warmup period
> pcav <- princomp(matv[1:(nrows/2), ])
> meanv <- pcav$center
> pcav <- list(values=pcav$sdev[1:ncols]^2, vectors=pcav$loadings[,
> # Update the PCA recursively
> for (i in (nrows/2+1):nrows) {
+   meanv <- onlinePCA::updateMean(meanv, matv[i, ], n=(i-1))
+   pcav <- onlinePCA::ghapca(lambda=pcav$values, U=pcav$vectors, x=
+ }  # end for
> # Compare PCA methods
> pcaf <- princomp(matv)
> pcaf$sdev^2
> drop(pcav$values)
```

Conclusion

Fast backtest simulations of trading strategies can be developed using the package *RcppArmadillo*.

The simulations can then be visualized using the package *shiny*.

Many thanks to Dirk Eddelbuettel and Romain Francois for developing the excellent *RcppArmadillo* package.

Many thanks to RStudio for developing the excellent *shiny* package.

Thank you!

jpawlowski@machinetrader.io
https://www.machinetrader.io