FRE7241 Algorithmic Portfolio Management

Lecture#6, Fall 2021

Jerzy Pawlowski *jp3900@nyu.edu*

NYU Tandon School of Engineering

October 19, 2021

NYU | TANDON SCHOOL OF ENGINEERING

Fitting Time Series to Autoregressive Models

An *autoregressive* process *AR(n)* for the time series of returns r_i:

$$r_i = \varphi_1 r_{i-1} + \varphi_2 r_{i-2} + \ldots + \varphi_n r_{i-n} + \xi_i = \sum_{j=1}^{n} \varphi_j r_{i-j} + \xi_i$$

Can be solved as a *multivariate* linear regression, with the *response* equal to r_i, and the columns of the *design matrix* equal to the lags of r_i.

An intercept term can be added to the above formula by adding a unit column to the regression design matrix.

Adding the intercept term produces slightly different coefficients, depending on the mean of the returns.

The function `stats::ar.ols()` fits an *AR(n)* model, but it produces slightly different coefficients than linear regression, because it uses a different calibration procedure.

```
> # Specify AR process parameters
> n_rows <- 1e3
> co_eff <- matrix(c(0.1, 0.39, 0.5)); n_coeff <- NROW(co_eff)
> set.seed(1121); in_nov <- matrix(rnorm(n_rows))
> # ari_ma <- filter(x=in_nov, filter=co_eff, method="recursive")
> # Simulate AR process using HighFreq::sim_ar()
> ari_ma <- HighFreq::sim_ar(coeff=co_eff, innov=in_nov)
> # Fit AR model using ar.ols()
> ar_fit <- ar.ols(ari_ma, order.max=n_coeff, aic=FALSE)
> class(ar_fit)
> is.list(ar_fit)
> drop(ar_fit$ar); drop(co_eff)
> # Define design matrix without intercept column
> de_sign <- sapply(1:n_coeff, rutils::lag_it, in_put=ari_ma)
> # Fit AR model using regression
> design_inv <- MASS::ginv(de_sign)
> coeff_fit <- drop(design_inv %*% ari_ma)
> all.equal(drop(ar_fit$ar), coeff_fit, check.attributes=FALSE)
```

The Standard Errors of the *AR(n)* Coefficients

The *standard errors* of the fitted *AR(n)* coefficients are proportional to the standard deviation of the fitted residuals.

Their *t*-values are equal to the ratio of the fitted coefficients divided by their standard errors.

```
> # Calculate the regression residuals
> fit_ted <- drop(de_sign %*% coeff_fit)
> residual_s <- drop(ari_ma - fit_ted)
> # Variance of residuals
> var_resid <- sum(residual_s^2)/(n_rows-NROW(coeff_fit))
> # Design matrix squared
> design_2 <- crossprod(de_sign)
> # Calculate covariance matrix of AR coefficients
> co_var <- var_resid*MASS::ginv(design_2)
> coeff_fitd <- sqrt(diag(co_var))
> # Calculate t-values of AR coefficients
> coeff_tvals <- drop(coeff_fit)/coeff_fitd
```

Order Selection of *AR(n)* Model

Order selection means determining the *order parameter* n of the *AR(n)* model that best fits the time series.

The order parameter n can be set equal to the number of significantly non-zero *partial autocorrelations* of the time series.

The order parameter can also be determined by only selecting coefficients with statistically significant t-values.

Fitting an *AR(n)* model can be performed by first determining the order n, and then calculating the coefficients.

The function `stats::arima()` calibrates (fits) an *ARIMA* model to a univariate time series.

The function `auto.arima()` from the package *forecast* performs order selection, and calibrates an *AR(n)* model to a univariate time series.

```
> # Fit AR(5) model into AR(3) process
> de_sign <- sapply(1:5, rutils::lag_it, in_put=ari_ma)
> design_inv <- MASS::ginv(de_sign)
> coeff_fit <- drop(design_inv %*% ari_ma)
> # Calculate t-values of AR(5) coefficients
> residual_s <- drop(ari_ma - drop(de_sign %*% coeff_fit))
> var_resid <- sum(residual_s^2)/(n_rows-NROW(coeff_fit))
> co_var <- var_resid*MASS::ginv(crossprod(de_sign))
> coeff_fitd <- sqrt(diag(co_var))
> coeff_tvals <- drop(coeff_fit)/coeff_fitd
> # Fit AR(5) model using arima()
> arima_fit <- arima(ari_ma, order=c(5, 0, 0), include.mean=FALSE)
> arima_fit$coef
> # Fit AR(5) model using auto.arima()
> library(forecast)  # Load forecast
> arima_fit <- forecast::auto.arima(ari_ma, max.p=5, max.q=0, max.d=
> # Fit AR(5) model into VTI returns
> re_turns <- drop(zoo::coredata(na.omit(rutils::etf_env$re_turns$VT
> de_sign <- sapply(1:5, rutils::lag_it, in_put=re_turns)
> design_inv <- MASS::ginv(de_sign)
> coeff_fit <- drop(design_inv %*% re_turns)
> # Calculate t-values of AR(5) coefficients
> residual_s <- drop(re_turns - drop(de_sign %*% coeff_fit))
> var_resid <- sum(residual_s^2)/(n_rows-NROW(coeff_fit))
> co_var <- var_resid*MASS::ginv(crossprod(de_sign))
> coeff_fitd <- sqrt(diag(co_var))
> coeff_tvals <- drop(coeff_fit)/coeff_fitd
```

The Yule-Walker Equations

To lighten the notation we can assume that the time series r_i has zero mean $\mathbb{E}[r_i] = 0$ and unit variance $\mathbb{E}[r_i^2] = 1$. ($\mathbb{E}$ is the expectation operator.)

Then the *autocorrelations* of r_i are equal to: $\rho_k = \mathbb{E}[r_i r_{i-k}]$.

If we multiply the *autoregressive* process *AR(n)*: $r_i = \sum_{j=1}^{n} \varphi_j r_{i-j} + \xi_i$, by r_{i-k} and take the expectations, then we obtain the Yule-Walker equations:

$$\begin{pmatrix} \rho_1 \\ \rho_2 \\ \rho_3 \\ \vdots \\ \rho_p \end{pmatrix} = \begin{pmatrix} 1 & \rho_1 & \dots & \rho_{n-1} \\ \rho_1 & 1 & \dots & \rho_{n-2} \\ \rho_2 & \rho_1 & \dots & \rho_{n-3} \\ \vdots & \vdots & \ddots & \vdots \\ \rho_{n-1} & \rho_{n-2} & \dots & 1 \end{pmatrix} \begin{pmatrix} \varphi_1 \\ \varphi_2 \\ \varphi_3 \\ \vdots \\ \varphi_n \end{pmatrix}$$

The Yule-Walker equations relate the *autocorrelation coefficients* ρ_i with the coefficients of the *AR(n)* process φ_i.

The Yule-Walker equations can be solved for the *AR(n)* coefficients φ_i using matrix inversion.

```
> # Compute autocorrelation coefficients
> ac_f <- acf(ari_ma, lag=10, plot=FALSE)
> ac_f <- drop(ac_f$acf)
> acf1 <- ac_f[-NROW(ac_f)]
> # Define Yule-Walker matrix
> yule_walker <- sapply(2:9, function(lagg) {
+   c(acf1[lagg:1], acf1[2:(NROW(acf1)-lagg+1)])
+ })  # end sapply
> yule_walker <- cbind(acf1, yule_walker, rev(acf1))
> # Generalized inverse of Yule-Walker matrix
> yule_walker_inv <- MASS::ginv(yule_walker)
> # Solve Yule-Walker equations
> coeff_yw <- drop(yule_walker_inv %*% ac_f[-1])
> round(coeff_yw, 5)
> coeff_fit
```

Forecasting Autoregressive Processes

An *autoregressive* process *AR(n)*:

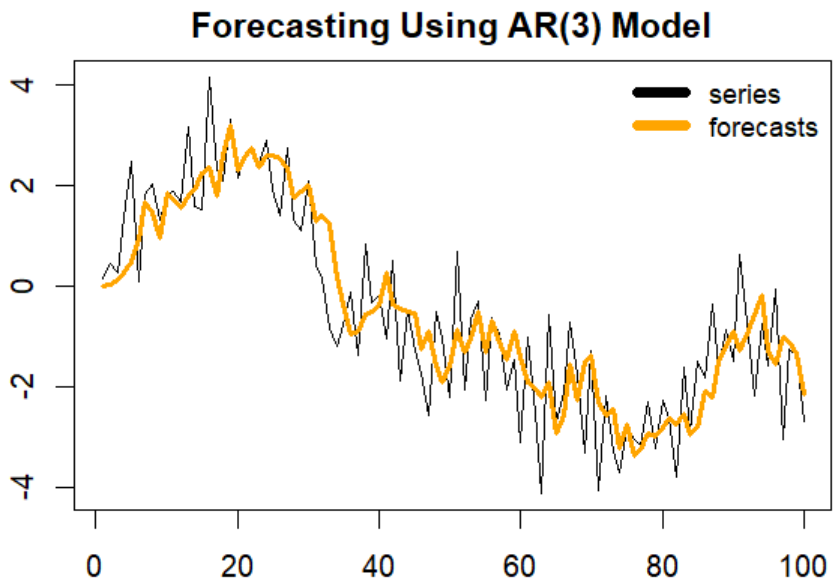
$$r_i = \varphi_1 r_{i-1} + \varphi_2 r_{i-2} + \ldots + \varphi_n r_{i-n} + \xi_i$$

Can be simulated using the function `filter()` with the argument `method="recursive"`.

Filtering can be performed even faster by directly calling the compiled C++ function `stats:::C_rfilter()`.

The one step ahead *forecast* f_i is equal to the *convolution* of the time series r_i with the *AR(n)* coefficients:

$$f_i = \varphi_1 r_{i-1} + \varphi_2 r_{i-2} + \ldots + \varphi_n r_{i-n}$$




```
> n_rows <- 1e2
> co_eff <- c(0.1, 0.39, 0.5); n_coeff <- NROW(co_eff)
> set.seed(1121); in_nov <- rnorm(n_rows)
> # Simulate AR process using filter()
> ari_ma <- filter(x=in_nov, filter=co_eff, method="recursive")
> ari_ma <- as.numeric(ari_ma)
> # Simulate AR process using C_rfilter()
> arima_fast <- .Call(stats:::C_rfilter, in_nov, co_eff,
+   double(n_rows + n_coeff))
> all.equal(ari_ma, arima_fast[-(1:n_coeff)],
+   check.attributes=FALSE)
```

```
> # Forecast AR(3) process using loop in R
> forecast_s <- numeric(NROW(ari_ma)+1)
> forecast_s[1] <- 0
> forecast_s[2] <- co_eff[1]*ari_ma[1]
> forecast_s[3] <- co_eff[1]*ari_ma[2] + co_eff[2]*ari_ma[1]
> for (it in 4:NROW(forecast_s)) {
+   forecast_s[it] <- ari_ma[(it-1):(it-3)] %*% co_eff
+ }  # end for
> # Plot with legend
> x11(width=6, height=5)
> par(mar=c(3, 3, 2, 1), oma=c(0, 0, 0, 0), mgp=c(2, 1, 0))
> plot(ari_ma, main="Forecasting Using AR(3) Model",
+   xlab="", ylab="", type="l")
> lines(forecast_s, col="orange", lwd=3)
> legend(x="topright", legend=c("series", "forecasts"),
+  col=c("black", "orange"), lty=1, lwd=6,
+  cex=0.9, bg="white", bty="n")
```

Fast Forecasting of Autoregressive Processes

The one step ahead *forecast* f_i is equal to the *convolution* of the time series r_i with the *AR(n)* coefficients:

$$f_i = \varphi_1 r_{i-1} + \varphi_2 r_{i-2} + \ldots + \varphi_n r_{i-n}$$

The above *convolution* can be quickly calculated by using the function `filter()` with the argument `method="convolution"`.

The convolution can be calculated even faster by directly calling the compiled `C++` function `stats:::C_cfilter()`.

The forecasts can also be calculated using the design matrix multiplied by the *AR(n)* coefficients.

```
> # Forecast using filter()
> filter_fast <- filter(x=ari_ma, sides=1,
+   filter=co_eff, method="convolution")
> filter_fast <- as.numeric(filter_fast)
> # Compare excluding warmup period
> all.equal(forecast_s[-(1:n_coeff)], filter_fast[-(1:(n_coeff-1))],
+     check.attributes=FALSE)
> # Filter using C_cfilter() compiled C++ function directly
> filter_fast <- .Call(stats:::C_cfilter, ari_ma, filter=co_eff,
+                sides=1, circular=FALSE)
> # Compare excluding warmup period
> all.equal(forecast_s[-(1:n_coeff)], filter_fast[-(1:(n_coeff-1))],
+     check.attributes=FALSE)
> # Filter using HighFreq::roll_conv() Rcpp function
> filter_fast <- HighFreq::roll_conv(matrix(ari_ma), matrix(co_eff))
> # Compare excluding warmup period
> all.equal(forecast_s[-(1:n_coeff)], filter_fast[-(1:(n_coeff-1))],
+     check.attributes=FALSE)
> # Define predictor matrix for forecasting
> predic_tor <- sapply(0:(n_coeff-1), function(lagg) {
+   rutils::lag_it(ari_ma, lagg=lagg)
+ })  # end sapply
> # Forecast using predictor matrix
> filter_fast <- c(0, drop(predic_tor %*% co_eff))
> # Compare with loop in R
> all.equal(forecast_s, filter_fast, check.attributes=FALSE)
```

Forecasting Using predict.Arima()

The forecasts of the *AR(n)* process can also be calculated using the function `predict()`.

The function `predict()` is a *generic function* for forecasting based on a given model.

The *method* `predict.Arima()` is *dispatched* by `R` for calculating predictions from *ARIMA* models produced by the function `stats::arima()`.

The *method* `predict.Arima()` returns a prediction object which is a `list` containing the predicted value and its standard error.

The function `stats::arima()` calibrates (fits) an *ARIMA* model to a univariate time series, using the *maximum likelihood* method (which may give slightly different coefficients than the linear regression model).

```
> # Fit ARIMA model using arima()
> arima_fit <- arima(ari_ma, order=c(3,0,0), include.mean=FALSE)
> arima_fit$coef
> co_eff
> # One-step-ahead forecast using predict.Arima()
> pre_dict <- predict(arima_fit, n.ahead=1)
> # Or directly call predict.Arima()
> # pre_dict <- predict.Arima(arima_fit, n.ahead=1)
> # Inspect the prediction object
> class(pre_dict)
> names(pre_dict)
> class(pre_dict$pred)
> unlist(pre_dict)
> # One-step-ahead forecast using matrix algebra
> fore_cast <- drop(ari_ma[n_rows:(n_rows-2)] %*% arima_fit$coef)
> # Compare one-step-ahead forecasts
> all.equal(pre_dict$pred[[1]], fore_cast)
> # Get information about predict.Arima()
> ?stats:::predict.Arima
```

The Forecasting Residuals

The *forecasting residuals* ε_i are equal to the differences between the actual values r_i minus their *forecasts* f_i: $\varepsilon_i = r_i - f_i$.

Accurate forecasting of an *AR(n)* process requires knowing its coefficients.

If the coefficients of the *AR(n)* process are known exactly, then its *in-sample residuals* ε_i are equal to its *innovations* ξ_i: $\varepsilon_i = r_i - f_i = \xi_i$.

In practice, the *AR(n)* coefficients are not known, so they must be fitted to the empirical time series.

If the *AR(n)* coefficients are fitted to the empirical time series, then its *residuals* are *not* equal to its *innovations*.

```
> # Calculate the in-sample forecasting residuals
> residual_s <- (ari_ma - forecast_s[-NROW(forecast_s)])
> # Compare residuals with innovations
> all.equal(in_nov, residual_s, check.attributes=FALSE)
> plot(residual_s, t="l", lwd=3, xlab="", ylab="",
+      main="ARIMA Forecast Errors")
```

Fitting and Forecasting Autoregressive Models

In practice, the *AR(n)* coefficients are not known, so they must be fitted to the empirical time series first, before forecasting.

Forecasting using an autoregressive model is performed by first fitting an *AR(n)* model to past data, and calculating its coefficients.

The fitted coefficients are then applied to calculating the *out-of-sample* forecasts.

The model fitting procedure depends on two unknown *meta-parameters*: the order n of the *AR(n)* model and the length of the look-back interval (`look_back`).

```
> # Define AR process parameters
> n_rows <- 1e3
> co_eff <- c(0.5, 0.0, 0.0); n_coeff <- NROW(co_eff)
> set.seed(1121); in_nov <- rnorm(n_rows)
> # Simulate AR process using C_rfilter()
> ari_ma <- .Call(stats:::C_rfilter, in_nov, co_eff,
+   double(n_rows + n_coeff))[-(1:n_coeff)]
> # Define order of the AR(n) forecasting model
> or_der <- 5
> # Define predictor matrix for forecasting
> de_sign <- sapply(1:or_der, rutils::lag_it, in_put=ari_ma)
> colnames(de_sign) <- paste0("pred_", 1:NCOL(de_sign))
> # Add response equal to series
> de_sign <- cbind(ari_ma, de_sign)
> colnames(de_sign)[1] <- "response"
> # Specify length of look-back interval
> look_back <- 100
> # Invert the predictor matrix
> rang_e <- (n_rows-look_back):(n_rows-1)
> design_inv <- MASS::ginv(de_sign[rang_e, -1])
> # Calculate fitted coefficients
> coeff_fit <- drop(design_inv %*% de_sign[rang_e, 1])
> # Calculate forecast
> drop(de_sign[n_rows, -1] %*% coeff_fit)
```

Backtesting Autoregressive Forecasting Models

Backtesting is the simulation of a model on historical data to test its forecasting accuracy.

The autoregressive forecasting model can be *backtested* by calculating forecasts over either a *rolling* or an *expanding* look-back interval.

If the start date is fixed at the first row then the look-back interval is *expanding*.

The coefficients of the *AR(n)* process are fitted to past data, and then applied to calculating out-of-sample forecasts.

The *backtesting* procedure allows determining the optimal *meta-parameters* of the forecasting model: the order *n* of the *AR(n)* model and the length of look-back interval (`look_back`).

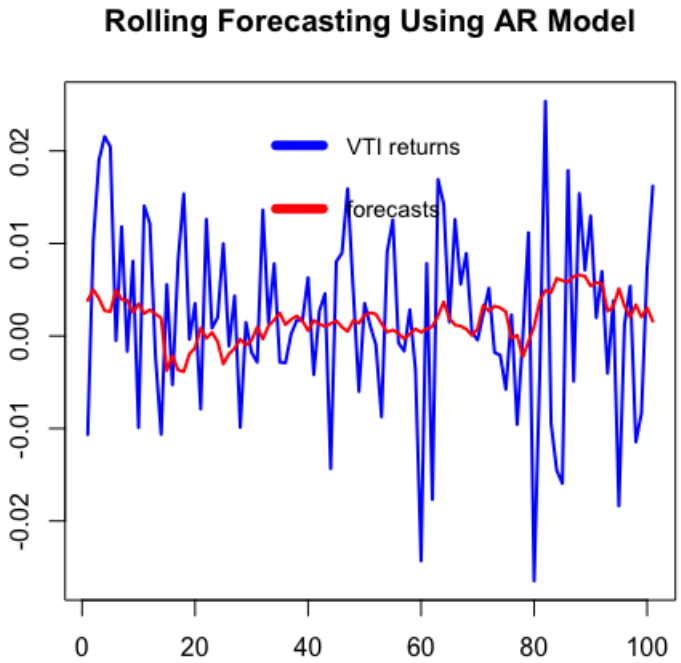
```
> # Calculate a vector of daily VTI log returns
> re_turns <- na.omit(rutils::etf_env$re_turns$VTI)
> date_s <- index(re_turns)
> re_turns <- as.numeric(re_turns)
> n_rows <- NROW(re_turns)
> # Define predictor as a rolling sum
> n_agg <- 5
> predic_tor <- rutils::roll_sum(re_turns, look_back=n_agg)
> # Shift the res_ponse forward out-of-sample
> res_ponse <- rutils::lag_it(predic_tor, lagg=(-n_agg))
> # Define predictor matrix for forecasting
> order_max <- 5
> predic_tor <- sapply(1+n_agg*(0:order_max), rutils::lag_it,
+                in_put=predic_tor)
> predic_tor <- cbind(rep(1, n_rows), predic_tor)
> # Define de_sign matrix
> de_sign <- cbind(res_ponse, predic_tor)
> # Perform rolling forecasting
> look_back <- 100
> forecast_s <- sapply((look_back+1):n_rows, function(end_p) {
+   # Define rolling look-back range
+   start_p <- max(1, end_p-look_back)
+   # Or expanding look-back range
+   # start_p <- 1
+   rang_e <- start_p:(end_p-1)
+   # Invert the predictor matrix
+   design_inv <- MASS::ginv(de_sign[rang_e, -1])
+   # Calculate fitted coefficients
+   coeff_fit <- drop(design_inv %*% de_sign[rang_e, 1])
+   # Calculate forecast
+   drop(de_sign[end_p, -1] %*% coeff_fit)
+ })  # end sapply
> # Add warmup period
> forecast_s <- c(rep(0, look_back), forecast_s)
```

The Accuracy of the Autoregressive Forecasting Model

The accuracy of a forecasting model can be measured using the *mean squared error* and the *correlation*.

The mean squared error (*MSE*) of a forecasting model is the average of the squared forecasting residuals ε_i, equal to the differences between the actual values r_i minus the *forecasts* f_i: $\varepsilon_i = r_i - f_i$:

$$\text{MSE} = \frac{1}{n}\sum_{i=1}^{n}(r_i - f_i)^2$$




```
> # Mean squared error
> mean((re_turns - forecast_s)^2)
> # Correlation
> cor(forecast_s, re_turns)
> # Plot forecasting series with legend
> x11(width=6, height=5)
> par(mar=c(3, 3, 2, 1), oma=c(0, 0, 0, 0))
> plot(forecast_s[(n_rows-look_back):n_rows], col="red",
+      xlab="", ylab="", type="l", lwd=2,
+      main="Rolling Forecasting Using AR Model")
> lines(re_turns[(n_rows-look_back):n_rows], col="blue", lwd=2)
> legend(x="top", legend=c("re_turns", "forecasts"),
+  col=c("blue", "red"), lty=1, lwd=6,
```

Backtesting Function for the Forecasting Model

The *meta-parameters* of the *backtesting* function are the order *n* of the *AR(n)* model and the length of the look-back interval (`look_back`).

```
> # Define backtesting function
> sim_forecasts <- function(res_ponse, predic_tor=res_ponse, n_agg=5
+                 or_der=5, look_back=100) {
+   n_rows <- NROW(res_ponse)
+   # Define predictor as a rolling sum
+   predic_tor <- rutils::roll_sum(res_ponse, look_back=n_agg)
+   # Shift the res_ponse forward out-of-sample
+   res_ponse <- rutils::lag_it(predic_tor, lagg=(-n_agg))
+   # Define predictor matrix for forecasting
+   predic_tor <- sapply(1+n_agg*(0:or_der), rutils::lag_it,
+                  in_put=predic_tor)
+   predic_tor <- cbind(rep(1, n_rows), predic_tor)
+   # Define de_sign matrix
+   de_sign <- cbind(res_ponse, predic_tor)
+   # Perform rolling forecasting
+   forecast_s <- sapply((look_back+1):n_rows, function(end_p) {
+     # Define rolling look-back range
+     start_p <- max(1, end_p-look_back)
+     # Or expanding look-back range
+     # start_p <- 1
+     rang_e <- start_p:(end_p-1)
+     # Invert the predictor matrix
+     design_inv <- MASS::ginv(de_sign[rang_e, -1])
+     # Calculate fitted coefficients
+     coeff_fit <- drop(design_inv %*% de_sign[rang_e, 1])
+     # Calculate forecast
+     drop(de_sign[end_p, -1] %*% coeff_fit)
+   })  # end sapply
+   # Add warmup period
+   forecast_s <- c(rep(0, look_back), forecast_s)
+   rutils::roll_sum(forecast_s, look_back=n_agg)
+ }  # end sim_forecasts
> # Simulate the rolling autoregressive forecasts
> forecast_s <- sim_forecasts(re_turns, or_der=5, look_back=100)
> c(mse=mean((re_turns - forecast_s)^2), cor=cor(re_turns, forecast_
```

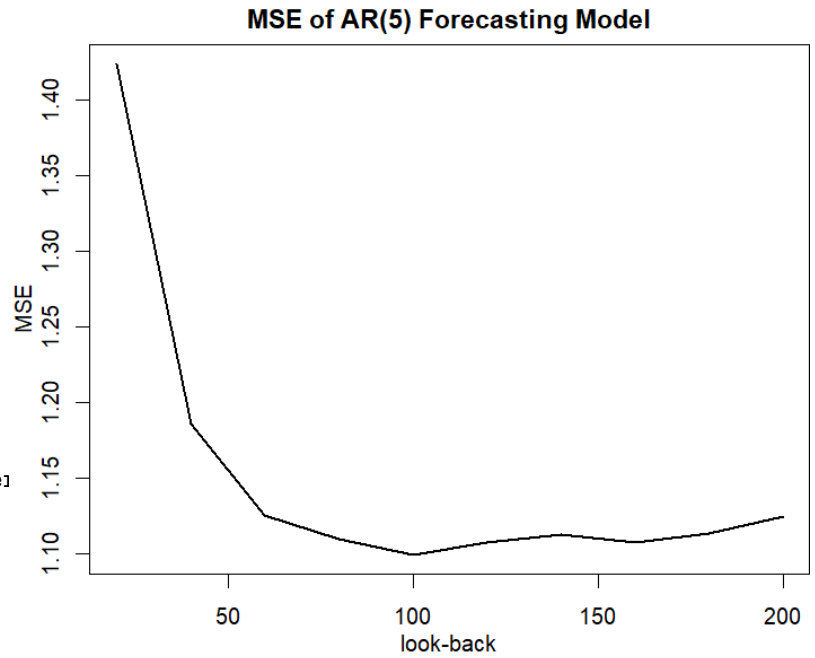
The Optimal Parameters of the Forecasting Model

The *backtesting* function can be used to find the optimal *meta-parameters* of the autoregressive forecasting model.

The accuracy of the forecasting model depends on the order n of the $AR(n)$ model and on the length of the look-back interval (`look_back`).

The two *meta-parameters* can be chosen by minimizing the *MSE* of the model forecasts in a *backtest* simulation.

```
> look_backs <- seq(20, 200, 20)
> back_tests <- sapply(look_backs, back_test, se_ries=ari_ma, or_de
> back_tests <- t(back_tests)
> rownames(back_tests) <- look_backs
> # Plot forecasting series with legend
> plot(x=look_backs, y=back_tests[, 1],
+   xlab="look-back", ylab="MSE", type="l", lwd=2,
+   main="MSE of AR(5) Forecasting Model")
```

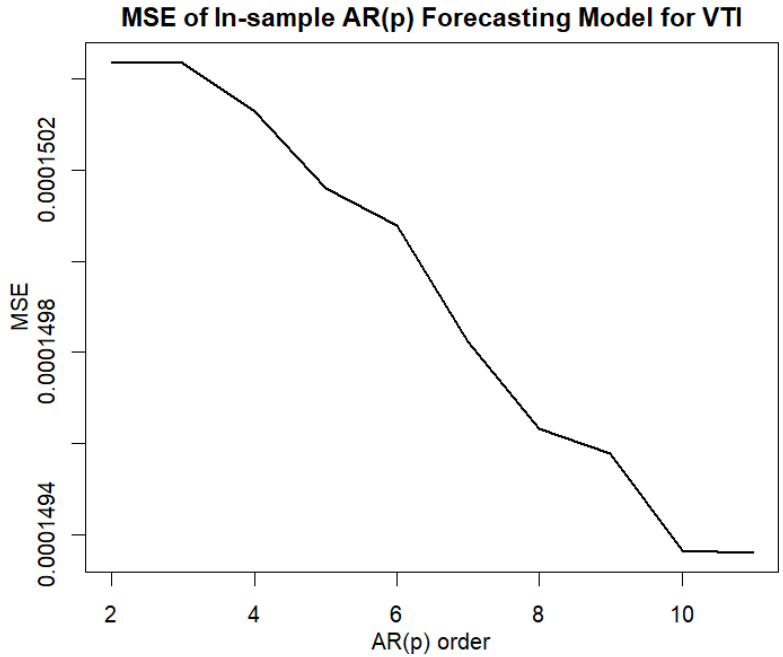


In-sample Forecasting Using Autoregressive Models

In-sample forecasting consists of first fitting an *AR(n)* model to the data, and calculating its coefficients.

The *in-sample* forecasts are calculated by multiplying the response vector of returns by the fitted coefficients.

The mean squared errors (*MSE*) of the *in-sample* forecasts decrease steadily with the increasing order parameter *n* of the *AR(n)* model.




```
> # Calculate a vector of daily VTI log returns
> vt_i <- na.omit(rutils::etf_env$re_turns$VTI)
> date_s <- index(vt_i)
> vt_i <- as.numeric(vt_i)
> n_rows <- NROW(vt_i)
> # Define predictor matrix for forecasting
> order_max <- 5
> predic_tor <- sapply(1:order_max, rutils::lag_it, in_put=vt_i)
> predic_tor <- cbind(rep(1, n_rows), predic_tor)
> colnames(predic_tor) <- paste0("pred_", 1:NCOL(predic_tor))
> res_ponse <- vt_i
> # Calculate forecasts as function of the AR order
> forecast_s <- lapply(2:NCOL(predic_tor), function(or_der) {
+   # Calculate fitted coefficients
+   in_verse <- MASS::ginv(predic_tor[, 1:or_der])
+   co_eff <- drop(in_verse %*% res_ponse)
+   # Calculate in-sample forecasts of vt_i
+   drop(predic_tor[, 1:or_der] %*% co_eff)
+ })  # end lapply
> names(forecast_s) <- paste0("p=", 2:NCOL(predic_tor))
```

```
> # Calculate mean squared errors
> ms_e <- sapply(forecast_s, function(x) {
+   c(mse=mean((vt_i - x)^2), cor=cor(vt_i, x))
+ })  # end sapply
> ms_e <- t(ms_e)
> rownames(ms_e) <- names(forecast_s)
> # Plot forecasting MSE
> x11(width=6, height=5)
> par(mar=c(3, 3, 2, 1), oma=c(0, 0, 0, 0), mgp=c(2, 1, 0))
> plot(x=2:NCOL(predic_tor), y=ms_e[, 1],
+   xlab="AR(n) order", ylab="MSE", type="l", lwd=2,
+   main="MSE of In-sample AR(n) Forecasting Model for VTI")
```

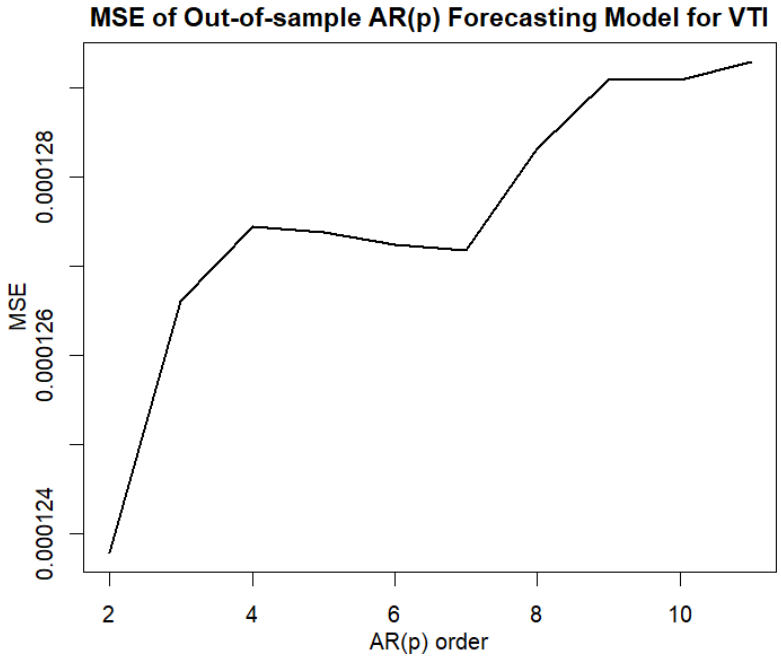
Out-of-sample Forecasting Using Autoregressive Models

Out-of-sample forecasting consists of first fitting an *AR(n)* model to the training data, and calculating its coefficients.

The *out-of-sample* forecasts are calculated by multiplying the *out-of-sample* response vector of returns by the fitted coefficients.

The mean squared errors (*MSE*) of the *out-of-sample* forecasts increase steadily with the increasing order parameter *n* of the *AR(n)* model.

The reason for the increasing out-of-sample MSE is *overfitting* of the coefficients to the training data.




```
> in_sample <- 1:(n_rows %/% 2)
> out_sample <- (n_rows %/% 2 + 1):n_rows
> # Calculate forecasts as function of the AR order
> forecast_s <- lapply(2:NCOL(predic_tor), function(or_der) {
+   # Calculate fitted coefficients
+   in_verse <- MASS::ginv(predic_tor[in_sample, 1:or_der])
+   co_eff <- drop(in_verse %*% res_ponse[in_sample])
+   # Calculate out-of-sample forecasts of vt_i
+   drop(predic_tor[out_sample, 1:or_der] %*% co_eff)
+ })  # end lapply
> names(forecast_s) <- paste0("p=", 2:NCOL(predic_tor))
```

```
> # Calculate mean squared errors
> ms_e <- sapply(forecast_s, function(x) {
+   c(mse=mean((vt_i[out_sample] - x)^2), cor=cor(vt_i[out_sample],
+ })  # end sapply
> ms_e <- t(ms_e)
> rownames(ms_e) <- names(forecast_s)
> # Plot forecasting MSE
> plot(x=2:NCOL(predic_tor), y=ms_e[, 1],
+   xlab="AR(n) order", ylab="MSE", type="l", lwd=2,
+   main="MSE of Out-of-sample AR(n) Forecasting Model for VTI")
```

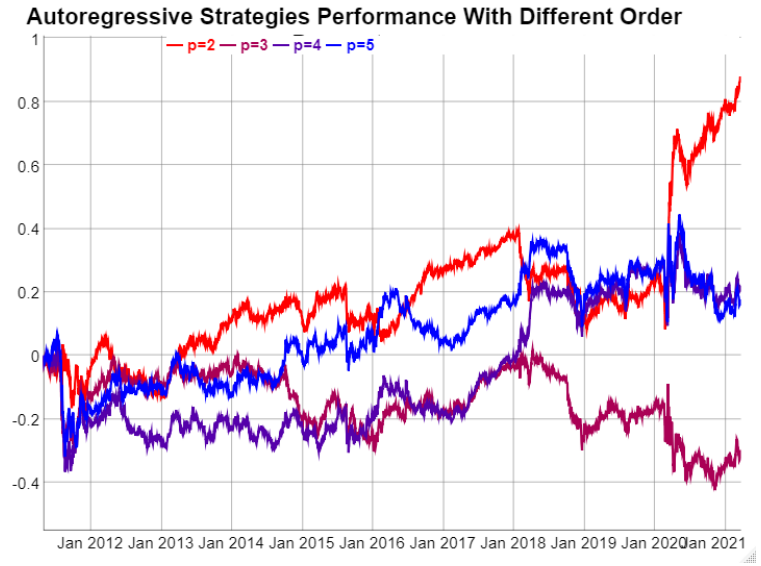
Autoregressive Strategy Out-of-sample Performance

The autoregressive strategy invests a dollar amount of *VTI* equal to the sign of the forecasts.

The performance of the autoregressive strategy is better with a smaller order parameter *n* of the *AR(n)* model.

Decreasing the order parameter of the autoregressive model is a form of *shrinkage* because it reduces the number of predictive variables.

```
> # Calculate out-of-sample PnLs
> pnl_s <- sapply(forecast_s, function(x) {
+   cumsum(sign(x)*vt_i[out_sample])
+ })  # end sapply
> colnames(pnl_s) <- names(forecast_s)
> pnl_s <- xts::xts(pnl_s, date_s[out_sample])
```




```
> # Plot dygraph of out-of-sample PnLs
> color_s <- colorRampPalette(c("red", "blue"))(NCOL(pnl_s[, 1:4]))
> col_names <- colnames(pnl_s[, 1:4])
> dygraphs::dygraph(pnl_s[, 1:4],
+   main="Autoregressive Strategies Performance With Different Order
+   dyOptions(colors=color_s, strokeWidth=2) %>%
+   dyLegend(width=500)
```

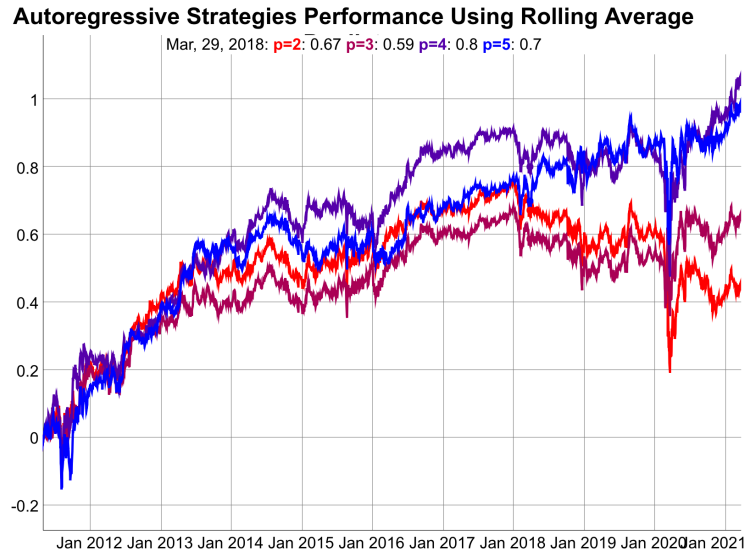
Autoregressive Strategy Using Rolling Average Returns

The *out-of-sample* forecasts can be improved by using the rolling average of the returns as a predictor.

This is because the average of returns has a lower *variance*.

But the average also has a higher *bias* because it includes returns that may be unrelated to the present.

Using the rolling average of returns as a predictor reduces the forecast variance at the expense of increasing its bias (known as the *bias-variance tradeoff*).




```
> # Define predictor as a rolling mean
> n_agg <- 5
> predic_tor <- roll::roll_mean(vt_i, width=n_agg, min_obs=1)
> res_ponse <- vt_i
> # Define predictor matrix for forecasting
> predic_tor <- sapply(1+n_agg*(0:order_max), rutils::lag_it,
+                in_put=predic_tor)
> predic_tor <- cbind(rep(1, n_rows), predic_tor)
> # Calculate forecasts as function of the AR order
> forecast_s <- lapply(2:NCOL(predic_tor), function(or_der) {
+   in_verse <- MASS::ginv(predic_tor[in_sample, 1:or_der])
+   co_eff <- drop(in_verse %*% res_ponse[in_sample])
+   drop(predic_tor[out_sample, 1:or_der] %*% co_eff)
+ })  # end lapply
> names(forecast_s) <- paste0("p=", 2:NCOL(predic_tor))
```

```
> # Calculate out-of-sample PnLs
> pnl_s <- sapply(forecast_s, function(x) {
+   cumsum(sign(x)*vt_i[out_sample])
+ })  # end sapply
> colnames(pnl_s) <- names(forecast_s)
> pnl_s <- xts::xts(pnl_s, date_s[out_sample])
> # Plot dygraph of out-of-sample PnLs
> dygraphs::dygraph(pnl_s[, 1:4],
+   main="Autoregressive Strategies Performance Using Rolling Averag
+   dyOptions(colors=color_s, strokeWidth=2) %>%
+   dyLegend(width=500)
```

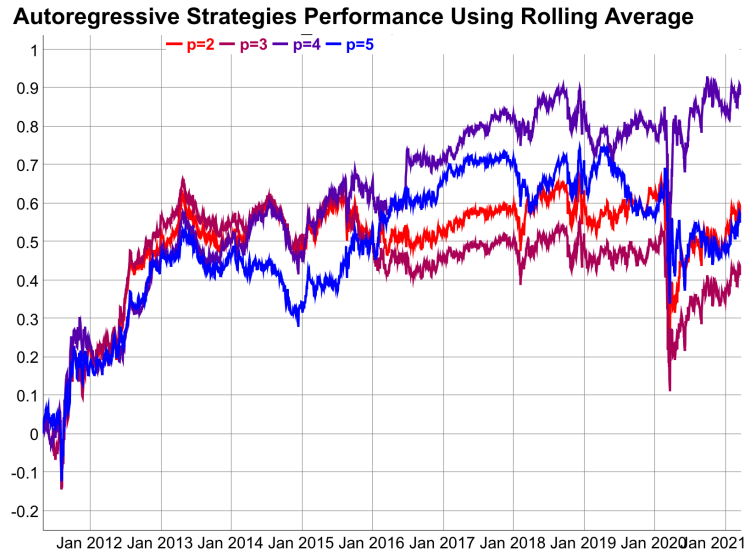
Autoregressive Strategy Using Rolling Average Forecasts

The *out-of-sample* forecasts can be further improved by using the average of past forecasts.

This is because the average of forecasts has a lower *variance*.

But the average also has a higher *bias* because it includes past forecasts that may be unrelated to the present.

Using the rolling average of past forecasts reduces the forecast variance at the expense of increasing its bias (known as the *bias-variance tradeoff*).




```
> # Calculate out-of-sample PnLs
> pnl_s <- sapply(forecast_s, function(x) {
+   x <- roll::roll_mean(x, width=n_agg, min_obs=1)
+   cumsum(sign(x)*vt_i[out_sample])
+ })  # end sapply
> colnames(pnl_s) <- names(forecast_s)
> pnl_s <- xts::xts(pnl_s, date_s[out_sample])
```

```
> # Plot dygraph of out-of-sample PnLs
> dygraphs::dygraph(pnl_s[, 1:4],
+   main="Autoregressive Strategies Performance Using Rolling Averag
+   dyOptions(colors=color_s, strokeWidth=2) %>%
+   dyLegend(width=500)
```

Backtesting Autoregressive Forecasting Models

Backtesting is the simulation of a model on historical data to test its forecasting accuracy.

The autoregressive forecasting model can be *backtested* by calculating forecasts over either a *rolling* or an *expanding* look-back interval.

If the start date is fixed at the first row then the look-back interval is *expanding*.

The coefficients of the *AR(n)* process are fitted to past data, and then applied to calculating out-of-sample forecasts.

The *backtesting* procedure allows determining the optimal *meta-parameters* of the forecasting model: the order *n* of the *AR(n)* model and the length of look-back interval (`look_back`).

```
> # Calculate a vector of daily VTI log returns
> vt_i <- na.omit(rutils::etf_env$re_turns$VTI)
> date_s <- index(vt_i)
> vt_i <- as.numeric(vt_i)
> n_rows <- NROW(vt_i)
> # Define predictor as a rolling mean
> n_agg <- 5
> predic_tor <- roll::roll_mean(vt_i, width=n_agg, min_obs=1)
> # Shift the res_ponse forward out-of-sample
> res_ponse <- vt_i
> # Define predictor matrix for forecasting
> order_max <- 5
> predic_tor <- sapply(1+n_agg*(0:order_max), rutils::lag_it,
+                in_put=predic_tor)
> predic_tor <- cbind(rep(1, n_rows), predic_tor)
> # Define de_sign matrix
> de_sign <- cbind(res_ponse, predic_tor)
> # Perform rolling forecasting
> look_back <- 100
> forecast_s <- sapply((look_back+1):n_rows, function(end_p) {
+   # Define rolling look-back range
+   start_p <- max(1, end_p-look_back)
+   # Or expanding look-back range
+   # start_p <- 1
+   rang_e <- start_p:(end_p-1)
+   # Invert the predictor matrix
+   design_inv <- MASS::ginv(de_sign[rang_e, -1])
+   # Calculate fitted coefficients
+   co_eff <- drop(design_inv %*% de_sign[rang_e, 1])
+   # Calculate forecast
+   drop(de_sign[end_p, -1] %*% co_eff)
+ })  # end sapply
> # Add warmup period
> forecast_s <- c(rep(0, look_back), forecast_s)
```

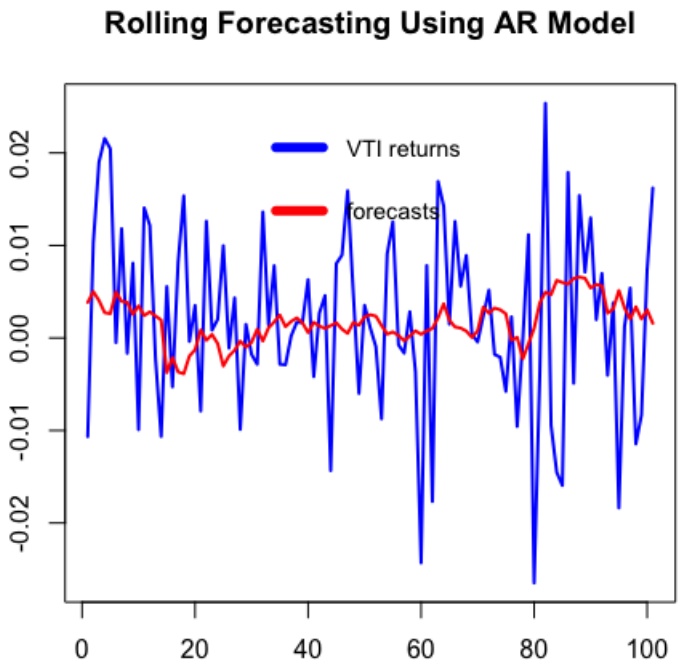
The Accuracy of the Autoregressive Forecasting Model

The accuracy of a forecasting model can be measured using the *mean squared error* and the *correlation*.

The mean squared error (*MSE*) of a forecasting model is the average of the squared forecasting residuals ε_i, equal to the differences between the actual values r_i minus the *forecasts* f_i: $\varepsilon_i = r_i - f_i$:

$$\text{MSE} = \frac{1}{n}\sum_{i=1}^{n}(r_i - f_i)^2$$

```
> # Mean squared error
> mean((vt_i - forecast_s)^2)
> # Correlation
> cor(forecast_s, vt_i)
> # Plot forecasting series with legend
> x11(width=6, height=5)
> par(mar=c(3, 3, 2, 1), oma=c(0, 0, 0, 0))
> plot(vt_i[(n_rows-look_back):n_rows], col="blue",
+      xlab="", ylab="", type="l", lwd=2,
+      main="Rolling Forecasting Using AR Model")
> lines(forecast_s[(n_rows-look_back):n_rows], col="red", lwd=2)
> legend(x="top", legend=c("VTI returns", "forecasts"),
+  col=c("blue", "red"), lty=1, lwd=6,
+  cex=0.9, bg="white", bty="n")
```



Backtesting Function for the *AR* Forecasting Model

The *meta-parameters* of the *backtesting* function are the order *n* of the *AR(n)* model and the length of the look-back interval (`look_back`).

```
> # Define backtesting function
> sim_forecasts <- function(res_ponse, predic_tor=res_ponse, n_agg=5
+                 or_der=5, look_back=100) {
+   n_rows <- NROW(res_ponse)
+   # Define predictor as a rolling mean
+   predic_tor <- roll::roll_mean(vt_i, width=n_agg, min_obs=1)
+   # Define predictor matrix for forecasting
+   predic_tor <- sapply(1+n_agg*(0:or_der), rutils::lag_it,
+                  in_put=predic_tor)
+   predic_tor <- cbind(rep(1, n_rows), predic_tor)
+   # Define de_sign matrix
+   de_sign <- cbind(res_ponse, predic_tor)
+   # Perform rolling forecasting
+   forecast_s <- sapply((look_back+1):n_rows, function(end_p) {
+     # Define rolling look-back range
+     start_p <- max(1, end_p-look_back)
+     # Or expanding look-back range
+     # start_p <- 1
+     rang_e <- start_p:(end_p-1)
+     # Invert the predictor matrix
+     design_inv <- MASS::ginv(de_sign[rang_e, -1])
+     # Calculate fitted coefficients
+     co_eff <- drop(design_inv %*% de_sign[rang_e, 1])
+     # Calculate forecast
+     drop(de_sign[end_p, -1] %*% co_eff)
+   })  # end sapply
+   # Add warmup period
+   forecast_s <- c(rep(0, look_back), forecast_s)
+   roll::roll_mean(forecast_s, width=n_agg, min_obs=1)
+ }  # end sim_forecasts
> # Simulate the rolling autoregressive forecasts
> forecast_s <- sim_forecasts(vt_i, or_der=5, look_back=100)
> c(mse=mean((vt_i - forecast_s)^2), cor=cor(vt_i, forecast_s))
```

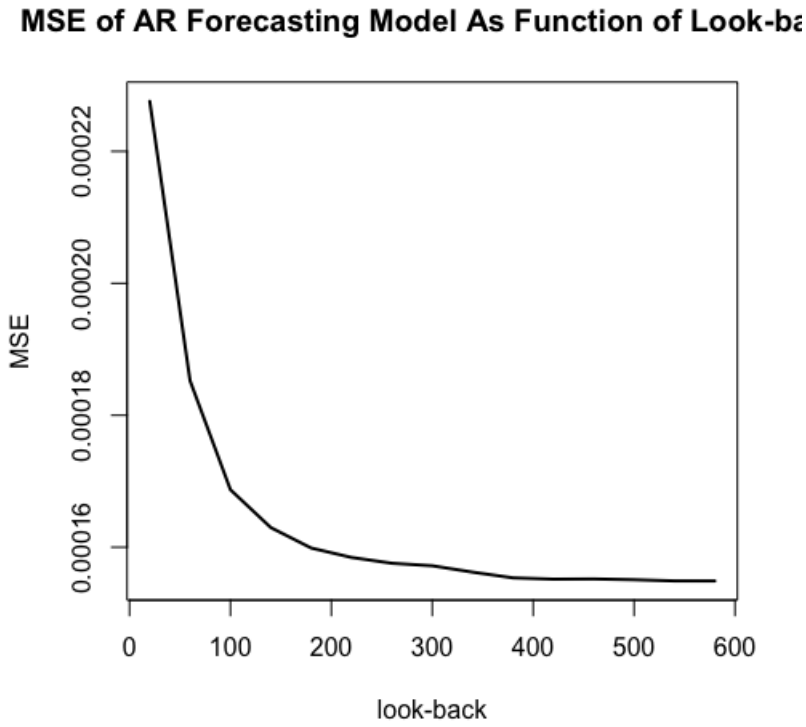
The Dependence On the Look-back Interval

The *backtesting* function can be used to find the optimal *meta-parameters* of the autoregressive forecasting model.

The accuracy of the forecasting model depends on the order *n* of the *AR(n)* model and on the length of the look-back interval (`look_back`).

The two *meta-parameters* can be chosen by minimizing the *MSE* of the model forecasts in a *backtest* simulation.

The accuracy of the forecasting model increases steadily with longer look-back intervals (`look_back`), because more data improves the estimates of the autoregressive coefficients.




```
> look_backs <- seq(20, 600, 40)
> library(parallel)  # Load package parallel
> # Calculate number of available cores
> n_cores <- detectCores() - 1
> # Initialize compute cluster under Windows
> clus_ter <- makeCluster(n_cores)
> # clusterExport(clus_ter, c("star_t", "bar_rier"))
> # Perform parallel loop under Windows
> forecast_s <- parLapply(clus_ter, look_backs, sim_forecasts, res
+                   predic_tor=vt_i, n_agg=5, or_der=5)
> # Perform parallel bootstrap under Mac-OSX or Linux
> forecast_s <- mclapply(look_backs, sim_forecasts, res_ponse=vt_i
+   predic_tor=vt_i, n_agg=5, or_der=5, mc.cores=n_cores)
```

```
> # Calculate mean squared errors
> ms_e <- sapply(forecast_s, function(x) {
+   c(mse=mean((vt_i - x)^2), cor=cor(vt_i, x))
+ })  # end sapply
> ms_e <- t(ms_e)
> rownames(ms_e) <- look_backs
> # Select optimal look_back interval
> look_back <- look_backs[which.min(ms_e[, 1])]
> # Plot forecasting MSE
> plot(x=look_backs, y=ms_e[, 1],
+   xlab="look-back", ylab="MSE", type="l", lwd=2,
+   main="MSE of AR Forecasting Model As Function of Look-back")
```

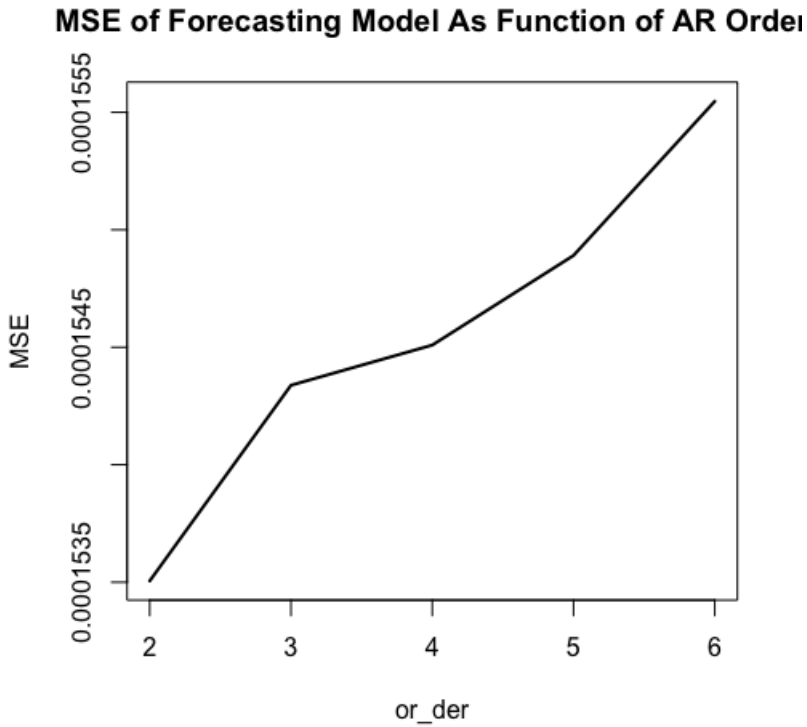
The Dependence On the Order Parameter

The *backtesting* function can be used to find the optimal *meta-parameters* of the autoregressive forecasting model.

The accuracy of the forecasting model depends on the order *n* of the *AR(n)* model and on the length of the look-back interval (`look_back`).

The two *meta-parameters* can be chosen by minimizing the *MSE* of the model forecasts in a *backtest* simulation.

The accuracy of the forecasting model increases steadily with longer look-back intervals (`look_back`), because more data improves the estimates of the autoregressive coefficients.




```
> order_s <- 2:6
> library(parallel)  # Load package parallel
> # Calculate number of available cores
> n_cores <- detectCores() - 1
> # Initialize compute cluster under Windows
> clus_ter <- makeCluster(n_cores)
> # clusterExport(clus_ter, c("star_t", "bar_rier"))
> # Perform parallel loop under Windows
> forecast_s <- parLapply(clus_ter, order_s, sim_forecasts, res_por
+                   predic_tor=vt_i, n_agg=5, look_back=look_back)
> stopCluster(clus_ter)  # Stop R processes over cluster under Winc
> # Perform parallel bootstrap under Mac-OSX or Linux
> forecast_s <- mclapply(order_s, sim_forecasts, res_ponse=vt_i,
+   predic_tor=vt_i, n_agg=5, look_back=look_back, mc.cores=n_cores
```

```
> # Calculate mean squared errors
> ms_e <- sapply(forecast_s, function(x) {
+   c(mse=mean((vt_i - x)^2), cor=cor(vt_i, x))
+ })  # end sapply
> ms_e <- t(ms_e)
> rownames(ms_e) <- order_s
> # Select optimal order parameter
> or_der <- order_s[which.min(ms_e[, 1])]
> # Plot forecasting MSE
> plot(x=order_s, y=ms_e[, 1],
+   xlab="or_der", ylab="MSE", type="l", lwd=2,
+   main="MSE of Forecasting Model As Function of AR Order")
```

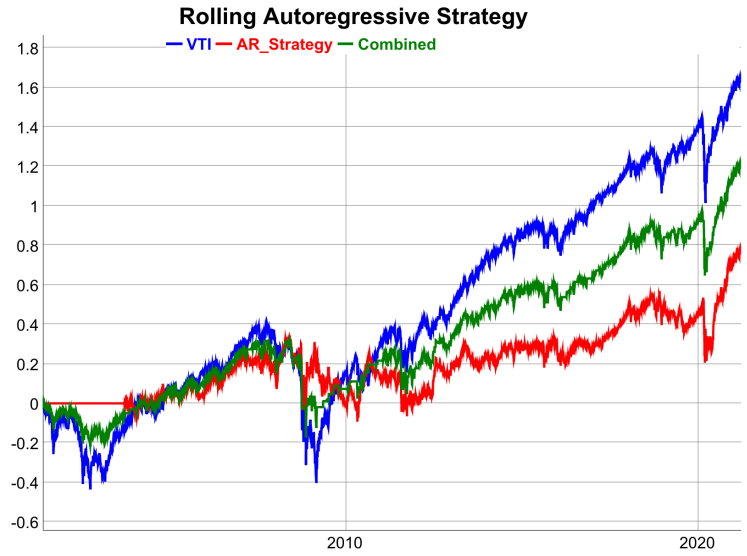
Performance of the Rolling Autoregressive Strategy

The return forecasts are calculated just before the close of the markets, so that trades can be executed before the close.

The autoregressive strategy is dominated by a few periods with very large returns, without producing profits for the remaining periods.

Using the return forecasts as portfolio weights produces very large weights in periods of high volatility, and creates excessive risk.

To reduce excessive risk, a binary strategy uses portfolio weights equaly to the sign of the forecasts.




```
> # Simulate the rolling autoregressive forecasts
> forecast_s <- sim_forecasts(vt_i, or_der=or_der, look_back=look_
> # Calculate strategy PnLs
> pnl_s <- sign(forecast_s)*vt_i
> pnl_s <- cbind(vt_i, pnl_s, (vt_i+pnl_s)/2)
> colnames(pnl_s) <- c("VTI", "AR_Strategy", "Combined")
> cor(pnl_s)
> # Annualized Sharpe ratios of VTI and AR strategy
> pnl_s <- xts::xts(pnl_s, date_s)
> sqrt(252)*sapply(pnl_s, function (x) mean(x)/sd(x))
```

```
> # Plot the cumulative strategy PnLs
> dygraphs::dygraph(cumsum(pnl_s), main="Rolling Autoregressive Stra
+   dyOptions(colors=c("blue", "red", "green"), strokeWidth=2) %>%
+   dyLegend(show="always", width=500)
```

Interest Rate Yield Curve and Stock Returns

Daily stock returns have insignificant correlations with the daily changes in interest rates, with the possible exception of the 10-year bond yield.

And these correlations change significantly over time.

```
> # Load constant maturity Treasury rates
> load(file="/Users/jerzy/Develop/lecture_slides/data/rates_data.RI
> # Combine rates into single xts series
> rate_s <- do.call(cbind, as.list(rates_env))
> # Sort the columns of rate_s according bond maturity
> name_s <- colnames(rate_s)
> name_s <- substr(name_s, start=4, stop=10)
> name_s <- as.numeric(name_s)
> indeks <- order(name_s)
> rate_s <- rate_s[, indeks]
> # Align rates dates with VTI prices
> clos_e <- quantmod::Cl(rutils::etf_env$VTI)
> colnames(clos_e) <- "VTI"
> n_rows <- NROW(clos_e)
> date_s <- zoo::index(clos_e)
> rate_s <- na.omit(rate_s[date_s])
> clos_e <- clos_e[zoo::index(rate_s)]
> date_s <- zoo::index(clos_e)
```

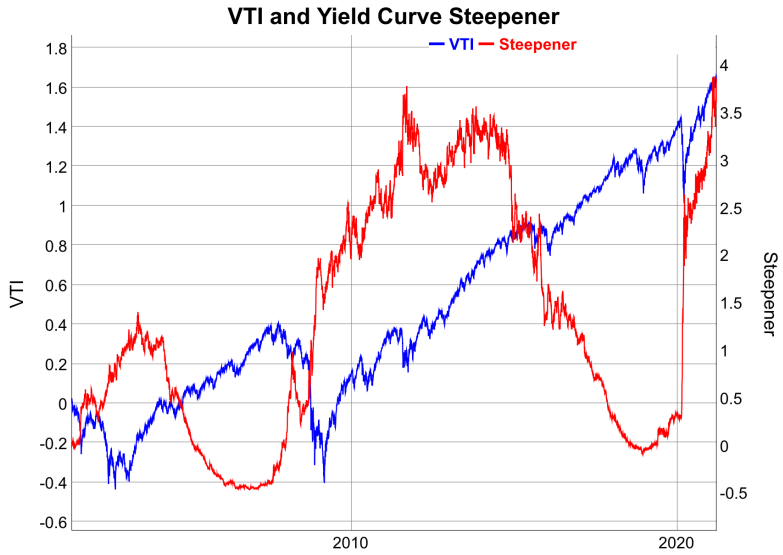
```
> # Calculate VTI returns and IR changes
> re_turns <- rutils::diff_it(log(clos_e))
> rates_diff <- rutils::diff_it(log(rate_s))
> # Regress VTI returns versus the lagged rate differences
> predic_tor <- rutils::lag_it(rates_diff)
> mod_el <- lm(re_turns ~ predic_tor)
> summary(mod_el)
> # Regress VTI returns before and after 2012
> summary(lm(re_turns["/2012"] ~ predic_tor["/2012"]))
> summary(lm(re_turns["2012/"] ~ predic_tor["2012/"]))
```

Yield Curve Principal Components and Stock Returns

The principal components of the interest rate yield curve can also be used as predictors of stock indices.

The second principal component describes the steepening and flattening of the yield curve, and it's an indicator of investor risk appetite. So it's also related to bullish and bearish market periods.

```
> # Calculate PCA of rates correlation matrix
> ei_gen <- eigen(cor(rates_diff))
> rates_pca <- -rates_diff %*% ei_gen$vectors
> colnames(rates_pca) <- paste0("PC", 1:6)
> # Define predictor as the YC PCAs
> predic_tor <- rutils::lag_it(rates_pca)
> mod_el <- lm(re_turns ~ predic_tor)
> summary(mod_el)
```



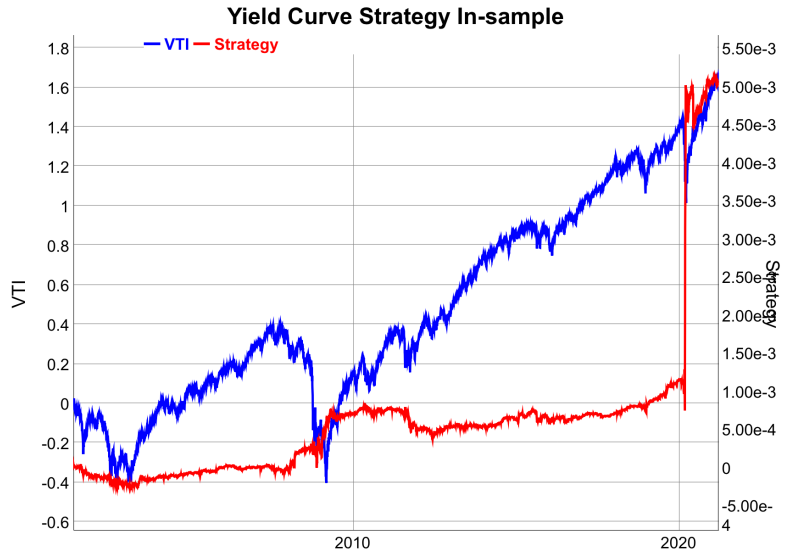

```
> # Plot YC steepener principal component with VTI
> da_ta <- cbind(re_turns, rates_pca[, 2])
> colnames(da_ta) <- c("VTI", "Steepener")
> col_names <- colnames(da_ta)
> dygraphs::dygraph(cumsum(da_ta), main="VTI and Yield Curve Steeper
+   dyAxis("y", label=col_names[1], independentTicks=TRUE) %>%
+   dyAxis("y2", label=col_names[2], independentTicks=TRUE) %>%
+   dySeries(name=col_names[1], axis="y", label=col_names[1], stroke
+   dySeries(name=col_names[2], axis="y2", label=col_names[2], strok
```

Stock Return Forecasts In-Sample

For in-sample forecasts, the training set and the test set are the same. The model is calibrated on the data that is used for forecasting.

Although it's not realistic to achieve the in-sample performance, it's useful because it provides insights into how the model works.

The in-sample strategy performs well in periods of high volatility, but otherwise it's flat.




```
> # Define predictor with intercept term
> predic_tor <- rutils::lag_it(rates_diff)
> predic_tor <- cbind(rep(1, NROW(predic_tor)), predic_tor)
> colnames(predic_tor)[1] <- "intercept"
> # Calculate inverse of predictor
> in_verse <- MASS::ginv(predic_tor)
> # Calculate coefficients from response and inverse of predictor
> res_ponse <- re_turns
> co_eff <- drop(in_verse %*% res_ponse)
> # Calculate forecasts in-sample
> forecast_s <- (predic_tor %*% co_eff)
> pnl_s <- forecast_s*re_turns
> # Calculate in-sample factors
> factor_s <- (predic_tor * co_eff)
> apply(factor_s, 2, sd)
```

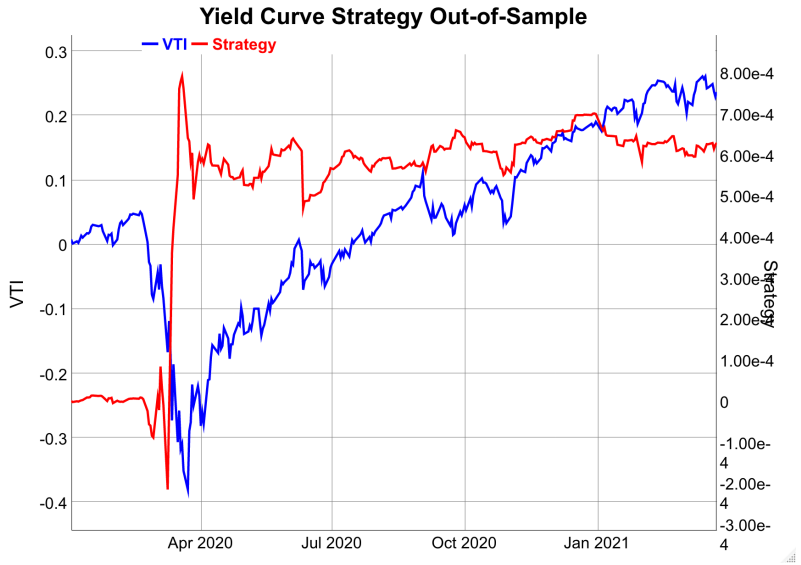
```
> # Plot dygraph of in-sample IR strategy
> weal_th <- cbind(re_turns, pnl_s)
> colnames(weal_th) <- c("VTI", "Strategy")
> col_names <- colnames(weal_th)
> dygraphs::dygraph(cumsum(weal_th), main="Yield Curve Strategy In-s
+   dyAxis("y", label=col_names[1], independentTicks=TRUE) %>%
+   dyAxis("y2", label=col_names[2], independentTicks=TRUE) %>%
+   dySeries(name=col_names[1], axis="y", col="blue", strokeWidth=2)
+   dySeries(name=col_names[2], axis="y2", col="red", strokeWidth=2)
+   dyLegend(show="always", width=500)
```

Stock Return Forecasts Out-of-Sample

For out-of-sample forecasts, the training set and the test set are separate. The model is calibrated on the training data, and forecasts are calculated using the test data.

The out-of-sample strategy performs well in periods of high volatility, but otherwise it's flat.

```
> # Define in-sample and out-of-sample intervals
> in_sample <- (date_s < as.Date("2020-01-01"))
> out_sample <- (date_s >= as.Date("2020-01-01"))
> # Calculate inverse of predictor in-sample
> in_verse <- MASS::ginv(predic_tor[in_sample, ])
> # Calculate coefficients in-sample
> co_eff <- drop(in_verse %*% res_ponse[in_sample, ])
> # Calculate forecasts and pnls out-of-sample
> forecast_s <- (predic_tor[out_sample, ] %*% co_eff)
> pnl_s <- forecast_s*re_turns[out_sample, ]
```



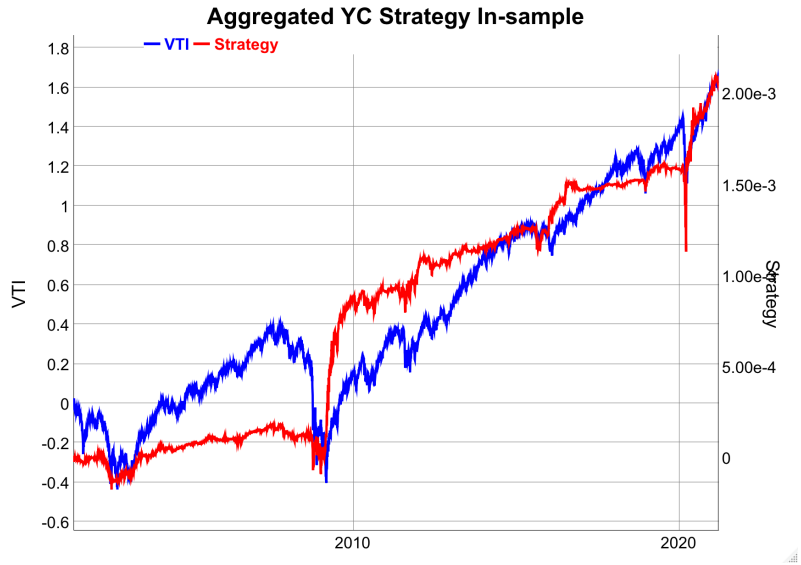

```
> # Plot dygraph of out-of-sample IR PCA strategy
> weal_th <- cbind(re_turns[out_sample, ], pnl_s)
> colnames(weal_th) <- c("VTI", "Strategy")
> col_names <- colnames(weal_th)
> dygraphs::dygraph(cumsum(weal_th), main="Yield Curve Strategy Out-
+   dyAxis("y", label=col_names[1], independentTicks=TRUE) %>%
+   dyAxis("y2", label=col_names[2], independentTicks=TRUE) %>%
+   dySeries(name=col_names[1], axis="y", col="blue", strokeWidth=2)
+   dySeries(name=col_names[2], axis="y2", col="red", strokeWidth=2)
+   dyLegend(show="always", width=500)
```

Forecasts Using Aggregated Predictor

Aggregating the predictor reduces its noise and increases the significance of correlations.

The optimal aggregation number can be found by maximizing the regression t-values.

```
> # Find optimal n_agg for predictor
> n_aggs <- 5:100
> tvalues <- sapply(n_aggs, function(n_agg) {
+   predic_tor <- roll::roll_mean(rates_diff, width=n_agg, min_obs=1
+   predic_tor <- cbind(rep(1, NROW(predic_tor)), predic_tor)
+   predic_tor <- rutils::lag_it(predic_tor)
+   mod_el <- lm(res_ponse ~ predic_tor - 1)
+   model_sum <- summary(mod_el)
+   max(abs(model_sum$coefficients[, 3][-1]))
+ })  # end sapply
> n_aggs[which.max(tvalues)]
> plot(n_aggs, tvalues, t="l", col="blue", lwd=2)
> # Calculate aggregated predictor
> n_agg <- 53
> predic_tor <- roll::roll_mean(rates_diff, width=n_agg, min_obs=1)
> predic_tor <- rutils::lag_it(predic_tor)
> predic_tor <- cbind(rep(1, NROW(predic_tor)), predic_tor)
> mod_el <- lm(res_ponse ~ predic_tor - 1)
> summary(mod_el)
```



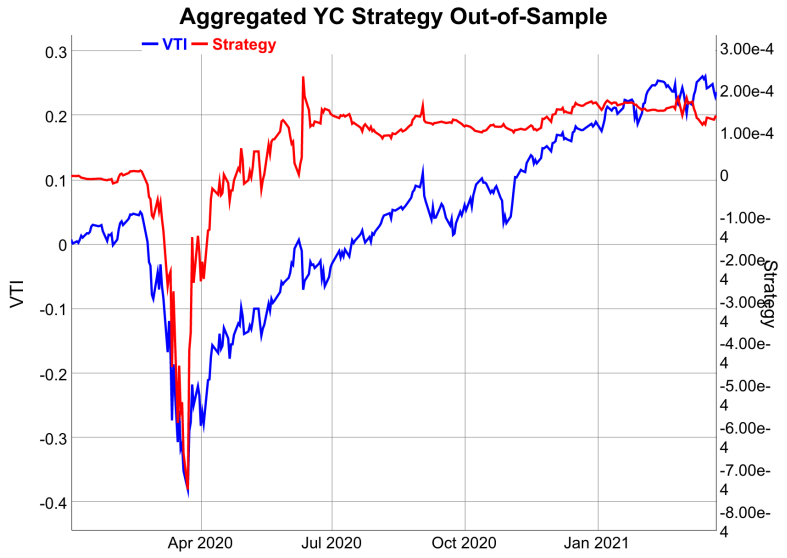

```
> # Calculate forecasts in-sample
> in_verse <- MASS::ginv(predic_tor)
> co_eff <- drop(in_verse %*% res_ponse)
> forecast_s <- (predic_tor %*% co_eff)
> pnl_s <- forecast_s*re_turns
> # Plot dygraph of in-sample IR strategy
> weal_th <- cbind(re_turns, pnl_s)
> colnames(weal_th) <- c("VTI", "Strategy")
> col_names <- colnames(weal_th)
> dygraphs::dygraph(cumsum(weal_th), main="Aggregated YC Strategy In
+   dyAxis("y", label=col_names[1], independentTicks=TRUE) %>%
+   dyAxis("y2", label=col_names[2], independentTicks=TRUE) %>%
+   dySeries(name=col_names[1], axis="y", col="blue", strokeWidth=2)
+   dySeries(name=col_names[2], axis="y2", col="red", strokeWidth=2)
+   dyLegend(show="always", width=500)
```

Aggregated Forecasts Out-of-Sample

For out-of-sample forecasts, the training set and the test set are separate. The model is calibrated on the training data, and forecasts are calculated using the test data.

The out-of-sample strategy performs well in periods of high volatility, but otherwise it's flat.

```
> # Define in-sample and out-of-sample intervals
> in_sample <- (date_s < as.Date("2020-01-01"))
> out_sample <- (date_s >= as.Date("2020-01-01"))
> # Calculate forecasts and pnls out-of-sample
> in_verse <- MASS::ginv(predic_tor[in_sample, ])
> co_eff <- drop(in_verse %*% res_ponse[in_sample, ])
> forecast_s <- (predic_tor[out_sample, ] %*% co_eff)
> pnl_s <- forecast_s*re_turns[out_sample, ]
```

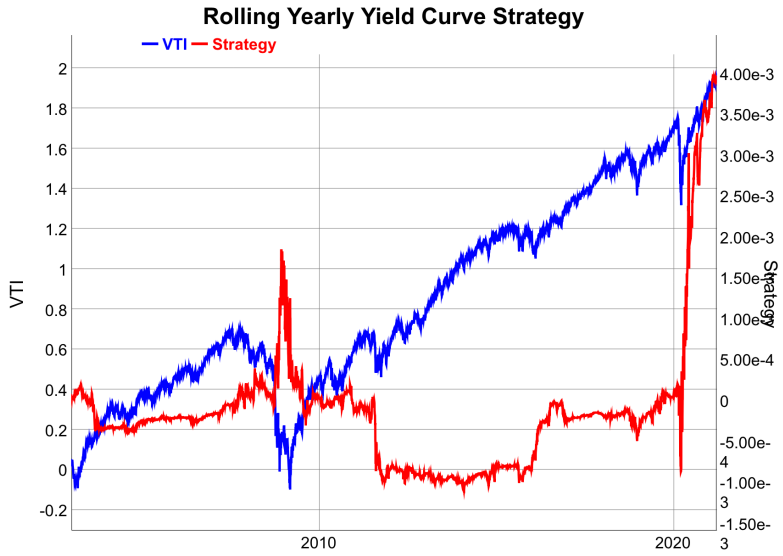



```
> # Plot dygraph of out-of-sample YC strategy
> weal_th <- cbind(re_turns[out_sample, ], pnl_s)
> colnames(weal_th) <- c("VTI", "Strategy")
> col_names <- colnames(weal_th)
> dygraphs::dygraph(cumsum(weal_th), main="Aggregated YC Strategy Ou
+   dyAxis("y", label=col_names[1], independentTicks=TRUE) %>%
+   dyAxis("y2", label=col_names[2], independentTicks=TRUE) %>%
+   dySeries(name=col_names[1], axis="y", col="blue", strokeWidth=2)
+   dySeries(name=col_names[2], axis="y2", col="red", strokeWidth=2)
+   dyLegend(show="always", width=500)
```

Rolling Yearly Yield Curve Strategy

In the rolling yearly yield curve strategy, the model is recalibrated at the end of every year using a training set of the past 2 years. The coefficients are applied to perform out-of-sample forecasts in the following year.

The rolling yearly strategy performs well in periods of high volatility, but otherwise it's flat.




```
> # Define yearly dates
> format(date_s[1], "%Y")
> year_s <- paste0(seq(2001, 2022, 1), "-01-01")
> year_s <- as.Date(year_s)
> # Perform loop over yearly dates
> pnl_s <- lapply(3:(NROW(year_s)-1), function(i) {
+   # Define in-sample and out-of-sample intervals
+   in_sample <- (date_s > year_s[i-2]) & (date_s < year_s[i])
+   out_sample <- (date_s >= year_s[i]) & (date_s < year_s[i+1])
+   # Calculate coefficients in-sample
+   in_verse <- MASS::ginv(predic_tor[in_sample, ])
+   # in_verse <- HighFreq::calc_inv(predic_tor[in_sample, ], eige
+   co_eff <- drop(in_verse %*% res_ponse[in_sample, ])
+   # Calculate forecasts and pnls out-of-sample
+   forecast_s <- (predic_tor[out_sample, ] %*% co_eff)
+   forecast_s*re_turns[out_sample, ]
+ })  # end lapply
> pnl_s <- do.call(rbind, pnl_s)
```

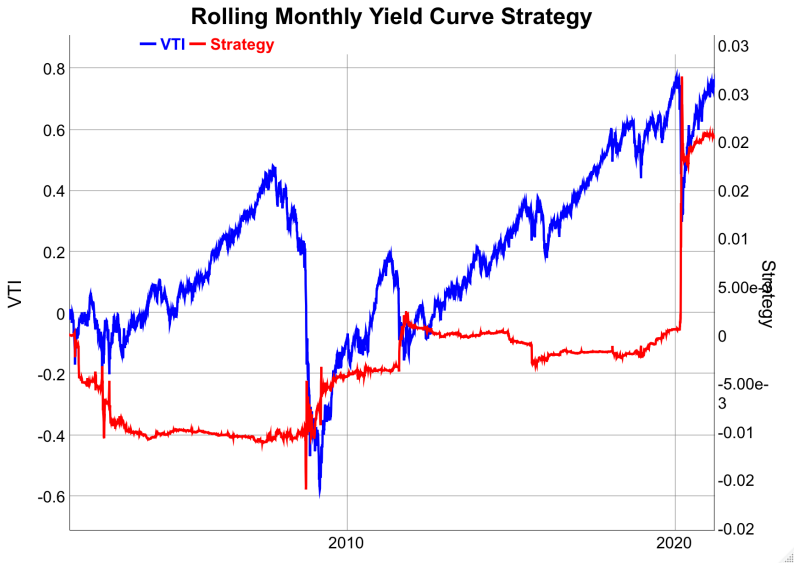
```
> # Plot dygraph of rolling yearly IR strategy
> weal_th <- cbind(re_turns[zoo::index(pnl_s),], pnl_s)
> colnames(weal_th) <- c("VTI", "Strategy")
> col_names <- colnames(weal_th)
> dygraphs::dygraph(cumsum(weal_th), main="Rolling Yearly Yield Curv
+   dyAxis("y", label=col_names[1], independentTicks=TRUE) %>%
+   dyAxis("y2", label=col_names[2], independentTicks=TRUE) %>%
+   dySeries(name=col_names[1], axis="y", col="blue", strokeWidth=2)
+   dySeries(name=col_names[2], axis="y2", col="red", strokeWidth=2)
+   dyLegend(show="always", width=500)
```

Rolling Monthly Yield Curve Strategy

In the rolling monthly yield curve strategy, the model is recalibrated at the end of every month using a training set of the past 11 months. The coefficients are applied to perform out-of-sample forecasts in the following month.

Research shows that looking back roughly a year provides the best out-of-sample forecasts.

The rolling monthly strategy performs better than the yearly strategy, but mostly in periods of high volatility, and otherwise it's flat.



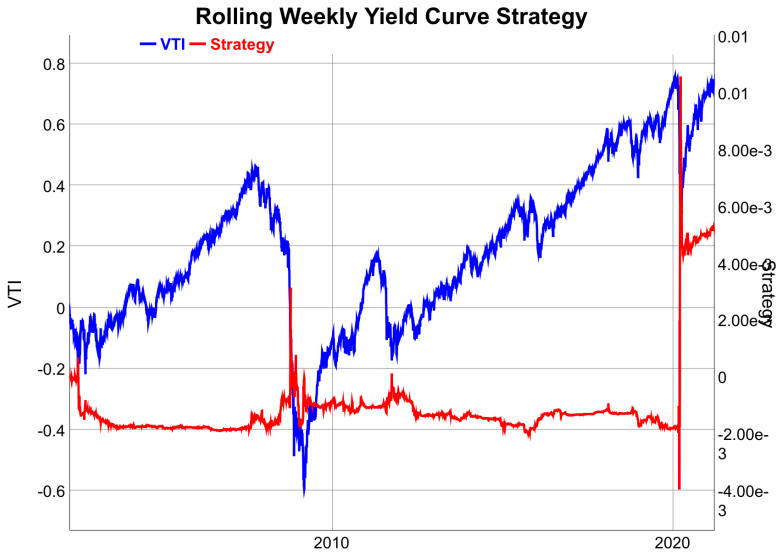

```
> # Define monthly dates
> format(date_s[1], "%m-%Y")
> format(date_s[NROW(date_s)], "%m-%Y")
> month_s <- seq.Date(from=as.Date("2001-05-01"), to=as.Date("2021-
> # Perform loop over monthly dates
> pnl_s <- lapply(12:(NROW(month_s)-1), function(i) {
+   # Define in-sample and out-of-sample intervals
+   in_sample <- (date_s > month_s[i-11]) & (date_s < month_s[i])
+   out_sample <- (date_s > month_s[i]) & (date_s < month_s[i+1])
+   # Calculate forecasts and pnls out-of-sample
+   in_verse <- MASS::ginv(predic_tor[in_sample, ])
+   # in_verse <- HighFreq::calc_inv(predic_tor[in_sample, ], eige
+   co_eff <- drop(in_verse %*% res_ponse[in_sample, ])
+   forecast_s <- (predic_tor[out_sample, ] %*% co_eff)
+   forecast_s*re_turns[out_sample, ]
+ })  # end lapply
> pnl_s <- do.call(rbind, pnl_s)
```

```
> # Plot dygraph of rolling monthly IR strategy
> weal_th <- cbind(re_turns[zoo::index(pnl_s),], pnl_s)
> colnames(weal_th) <- c("VTI", "Strategy")
> col_names <- colnames(weal_th)
> dygraphs::dygraph(cumsum(weal_th), main="Rolling Monthly Yield Cu
+   dyAxis("y", label=col_names[1], independentTicks=TRUE) %>%
+   dyAxis("y2", label=col_names[2], independentTicks=TRUE) %>%
+   dySeries(name=col_names[1], axis="y", col="blue", strokeWidth=2
+   dySeries(name=col_names[2], axis="y2", col="red", strokeWidth=2
+   dyLegend(show="always", width=500)
```

Rolling Weekly Yield Curve Strategy

In the rolling weekly yield curve strategy, the model is recalibrated at the end of every week using a training set of the past 50 weeks. The coefficients are applied to perform out-of-sample forecasts in the following week.

The rolling weekly strategy performs worse than the monthly strategy.




```
> # Define weekly dates
> week_s <- seq.Date(from=as.Date("2001-05-01"), to=as.Date("2021-04
> # Perform loop over weekly dates
> pnl_s <- lapply(51:(NROW(week_s)-1), function(i) {
+   # Define in-sample and out-of-sample intervals
+   in_sample <- (date_s > week_s[i-50]) & (date_s < week_s[i])
+   out_sample <- (date_s > week_s[i]) & (date_s < week_s[i+1])
+   # Calculate forecasts and pnls out-of-sample
+   in_verse <- MASS::ginv(predic_tor[in_sample, ])
+   # in_verse <- HighFreq::calc_inv(predic_tor[in_sample, ], eige
+   co_eff <- drop(in_verse %*% res_ponse[in_sample, ])
+   forecast_s <- (predic_tor[out_sample, ] %*% co_eff)
+   forecast_s*re_turns[out_sample, ]
+ })  # end lapply
> pnl_s <- do.call(rbind, pnl_s)
```

```
> # Plot dygraph of rolling weekly IR strategy
> weal_th <- cbind(re_turns[zoo::index(pnl_s),], pnl_s)
> colnames(weal_th) <- c("VTI", "Strategy")
> col_names <- colnames(weal_th)
> dygraphs::dygraph(cumsum(weal_th), main="Rolling Weekly Yield Curv
+   dyAxis("y", label=col_names[1], independentTicks=TRUE) %>%
+   dyAxis("y2", label=col_names[2], independentTicks=TRUE) %>%
+   dySeries(name=col_names[1], axis="y", col="blue", strokeWidth=2)
+   dySeries(name=col_names[2], axis="y2", col="red", strokeWidth=2)
+   dyLegend(show="always", width=500)
```

Momentum Strategy for an *ETF* Portfolio

Momentum strategies can be *backtested* by specifying the portfolio rebalancing frequency, the formation period, and the holding period:

- Specify a portfolio of *ETFs*, stocks, or other assets, and a time series of their returns,
- Specify *end points* for the portfolio rebalancing frequency,
- Specify *look-back* intervals for portfolio formation, and *look-forward* intervals for portfolio holding,
- Specify a performance function to calculate the past performance of the assets,
- Calculate the past performance over the *look-back* formation intervals,
- Calculate the portfolio weights from the past performance,
- Calculate the future returns over the *look-forward* holding intervals,
- Calculate the out-of-sample momentum strategy returns by applying the portfolio weights to the future returns,
- Calculate the transaction costs and subtract them from the strategy returns.

```
> # Extract ETF returns
> sym_bols <- c("VTI", "IEF", "DBC")
> re_turns <- rutils::etf_env$re_turns[, sym_bols]
> re_turns <- na.omit(re_turns)
> # Or, select rows with IEF data
> # re_turns <- re_turns[index(rutils::etf_env$IEF)]
> # Copy over NA values
> # re_turns[1, is.na(re_turns[1, ])] <- 0
> # re_turns <- zoo::na.locf(re_turns, na.rm=FALSE)
```

Look-back and Look-forward Intervals

Performance aggregations are be calculated over a vector of overlapping in-sample *look-back* intervals attached at *end points*.

For example, aggregations at monthly *end points* over overlapping 12-month *look-back* intervals.

An example of a data aggregation are the cumulative past returns at each *end point*.

The variable `look_back` is equal to the number of *end points* in the *look-back* interval.

The *start points* are the *end points* lagged by the length of the *look-back* interval.

The *look-back* intervals are spanned by the vectors of *start points* and *end points*.

Performance aggregations are also be calculated over non-overlapping out-of-sample *look-forward* intervals.

The *look-forward* intervals should not overlap with the *look-back* intervals, in order to avoid data snooping.

```
> # Define end of month end points
> end_p <- rutils::calc_endpoints(re_turns, inter_val="months")
> end_p <- end_p[-1]
> n_rows <- NROW(end_p)
> date_s <- zoo::index(re_turns)[end_p]
> # Start points equal end points lagged by 12-month look-back inter
> look_back <- 12
> start_p <- c(rep_len(1, look_back-1),
+   end_p[1:(n_rows - look_back + 1)])
> # Calculate matrix of look-back intervals
> look_backs <- cbind(start_p, end_p)
> colnames(look_backs) <- c("start", "end")
> # Calculate matrix of look-forward intervals
> look_fwds <- cbind(end_p + 1, rutils::lag_it(end_p, -1))
> look_fwds[n_rows, ] <- end_p[n_rows]
> colnames(look_fwds) <- c("start", "end")
> # Inspect the intervals
> head(cbind(look_backs, look_fwds))
> tail(cbind(look_backs, look_fwds))
```

Momentum Portfolio Weights

The portfolio weights of *momentum* strategies are calculated based on the past performance of the assets.

The weights are scaled to limit the portfolio *leverage* and its market *beta*.

The portfolio weights of *momentum* strategies can be scaled in several different ways.

To limit the portfolio leverage, the weights can be scaled so that the sum of squares is equal to 1: $\sum_{i=1}^{n} w_i^2 = 1$.

The weights can also be de-meaned, so that their sum is equal to zero, to create long-short portfolios with small betas: $\sum_{i=1}^{n} w_i = 0$.

```
> # Define performance function as Sharpe ratio
> perform_ance <- function(re_turns) sum(re_turns)/sd(re_turns)
> # Calculate past performance over look-back intervals
> pas_t <- apply(look_backs, 1, function(ep) {
+   sapply(re_turns[ep[1]:ep[2]], perform_ance)
+ })  # end sapply
> pas_t <- t(pas_t)
> pas_t[is.na(pas_t)] <- 0
> # Weights are proportional to past performance
> weight_s <- pas_t
> # weight_s[weight_s < 0] <- 0
> # Scale weight_s so sum of squares is equal to 1.
> weight_s <- weight_s/sqrt(rowSums(weight_s^2))
> # Or scale weight_s so sum is equal to 1
> # weight_s <- weight_s/rowSums(weight_s)
> # Set NA values to zero
> weight_s[is.na(weight_s)] <- 0
> sum(is.na(weight_s))
```

Backtesting the Momentum Strategy

Backtesting is the testing of the accuracy of a forecasting model using simulation on historical data.

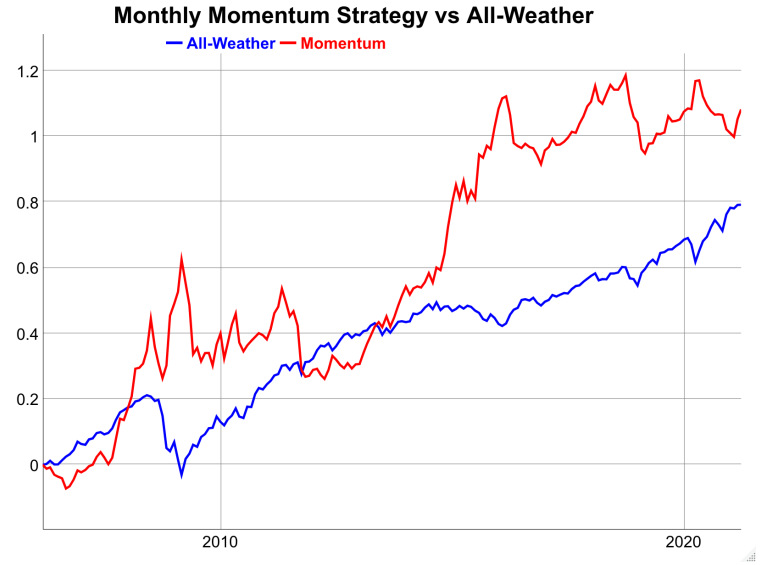
Backtesting is a type of *cross-validation* applied to time series data.

Backtesting is performed by *training* the model on past data and *testing* it on future out-of-sample data.

The *training* data is specified by the *look-back* intervals (`pas_t`), and the model forecasts are applied to the future data defined by the *look-forward* intervals (`fu_ture`).

The out-of-sample *momentum* strategy returns can be calculated by multiplying the `fu_ture` returns by the forecast *ETF* portfolio weights.

The momentum returns are lagged so that they are attached to the end of the future interval, instead of at its beginning.




```
> # Calculate future out-of-sample performance
> fu_ture <- apply(look_fwds, 1, function(ep) {
+   sapply(re_turns[ep[1]:ep[2]], sum)
+ })  # end sapply
> fu_ture <- t(fu_ture)
> fu_ture[is.na(fu_ture)] <- 0
> tail(fu_ture)
```

```
> # Calculate the momentum pnls
> pnl_s <- rowSums(weight_s*fu_ture)
> # Lag the future and momentum returns to proper dates
> fu_ture <- rutils::lag_it(fu_ture)
> pnl_s <- rutils::lag_it(pnl_s)
> # The momentum strategy has low correlation to stocks
> cor(pnl_s, fu_ture)
> # Define all-weather benchmark
> weights_aw <- c(0.30, 0.55, 0.15)
> all_weather <- fu_ture %*% weights_aw
> # Calculate the wealth of momentum returns
> weal_th <- xts::xts(cbind(all_weather, pnl_s), order.by=date_s)
> colnames(weal_th) <- c("All-Weather", "Momentum")
> cor(weal_th)
> # Plot dygraph of the momentum strategy returns
> dygraphs::dygraph(cumsum(weal_th), main="Monthly Momentum Strategy
+   dyOptions(colors=c("blue", "red"), strokeWidth=2) %>%
+   dyLegend(show="always", width=500)
```

Backtesting Functional for Momentum Strategy

```
> backtest_momentum <- function(returns,
+                 perform_ance=function(re_turns) (sum(re_turns)/sd(re_turns)),
+                 look_back=12, re_balance="months", bid_offer=0.001,
+                 end_p=rutils::calc_endpoints(re_turns, inter_val=re_balance)[-1],
+                 with_weights=FALSE, ...) {
+   stopifnot("package:rutils" %in% search() || require("rutils", quietly=TRUE))
+   # Define look-back and look-forward intervals
+   n_rows <- NROW(end_p)
+   start_p <- c(rep_len(1, look_back-1), end_p[1:(n_rows-look_back+1)])
+   # Calculate look-back intervals
+   look_backs <- cbind(start_p, end_p)
+   # Calculate look-forward intervals
+   look_fwds <- cbind(end_p + 1, rutils::lag_it(end_p, -1))
+   look_fwds[n_rows, ] <- end_p[n_rows]
+   # Calculate past performance over look-back intervals
+   pas_t <- t(apply(look_backs, 1, function(ep) sapply(re_turns[ep[1]:ep[2]], perform_ance)))
+   pas_t[is.na(pas_t)] <- 0
+   # Calculate future performance
+   fu_ture <- t(apply(look_fwds, 1, function(ep) sapply(re_turns[ep[1]:ep[2]], sum)))
+   fu_ture[is.na(fu_ture)] <- 0
+   # Scale weight_s so sum of squares is equal to 1
+   weight_s <- pas_t
+   weight_s <- weight_s/sqrt(rowSums(weight_s^2))
+   weight_s[is.na(weight_s)] <- 0  # Set NA values to zero
+   # Calculate momentum profits and losses
+   pnl_s <- rowSums(weight_s*fu_ture)
+   # Calculate transaction costs
+   cost_s <- 0.5*bid_offer*cumprod(1 + pnl_s)*rowSums(abs(rutils::diff_it(weight_s)))
+   pnl_s <- (pnl_s - cost_s)
+   if (with_weights)
+     rutils::lag_it(cbind(pnl_s, weight_s))
+   else
+     rutils::lag_it(pnl_s)
+ }  # end backtest_momentum
```

Optimization of Momentum Strategy Parameters

The performance of the *momentum* strategy depends on the length of the *look-back interval* used for calculating the past performance.

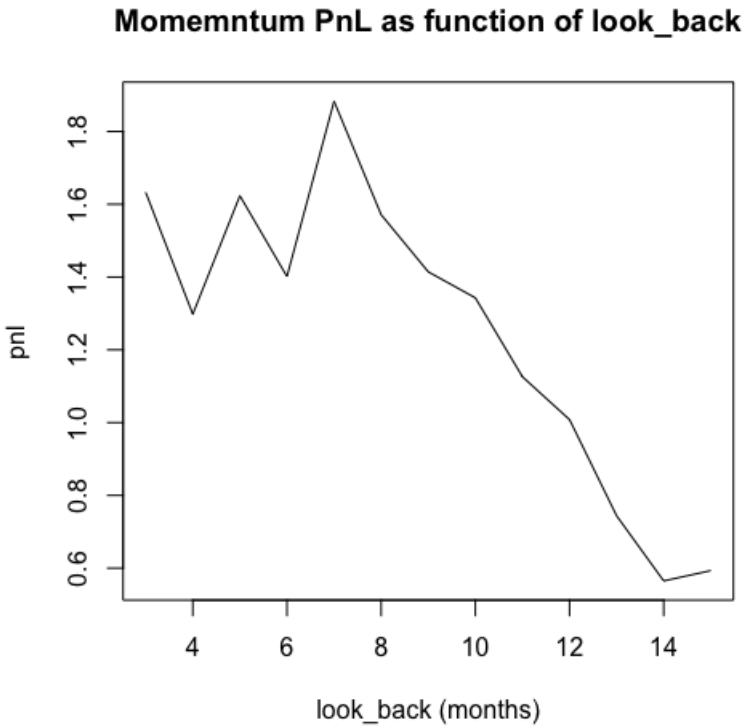
Performing a *backtest* allows finding the optimal *momentum* (trading) strategy parameters, such as the *look-back interval*.

But using a different rebalancing frequency in the *backtest* can produce different values for the optimal trading strategy parameters.

So *backtesting* just redefines the problem of finding (tuning) the optimal trading strategy parameters, into the problem of finding the optimal *backtest* (meta-model) parameters.

But the advantage of using the *backtest* meta-model is that it can reduce the number of parameters that need to be optimized.

Performing many *backtests* on multiple trading strategies risks identifying inherently unprofitable trading strategies as profitable, purely by chance (*p-value hacking*).

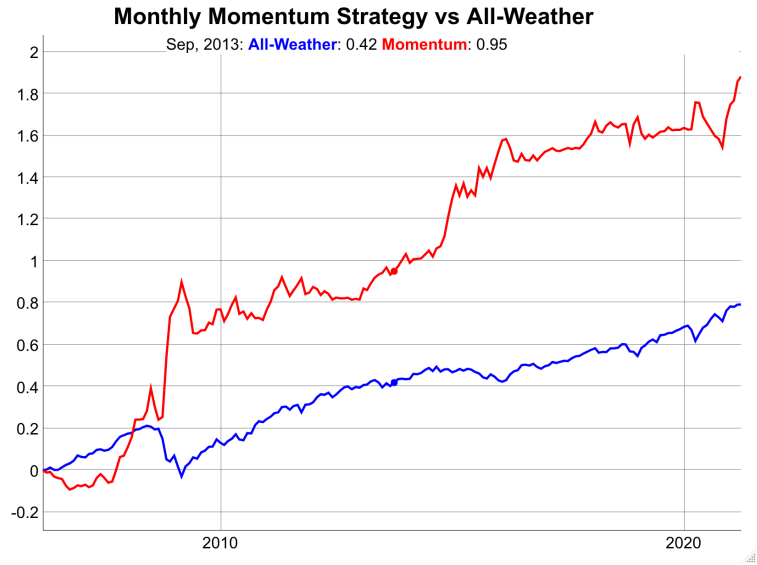



```
> source("C:/Develop/lecture_slides/scripts/back_test.R")
> look_backs <- seq(3, 15, by=1)
> perform_ance <- function(re_turns) sum(re_turns)/sd(re_turns)
> pro_file <- sapply(look_backs, function(look_back) {
+   pnl_s <- backtest_momentum(returns=re_turns, end_p=end_p,
+     look_back=look_back, perform_ance=perform_ance)
+   sum(pnl_s)
+ })  # end sapply
> # Plot momemntum PnLs
> x11(width=6, height=5)
> plot(x=look_backs, y=pro_file, t="l",
+   main="Momemntum PnL as function of look_back",
+   xlab="look_back (months)", ylab="pnl")
```

Optimal ETF Momentum Strategy Performance

The hypothetical out-of-sample *momentum* strategy returns can be calculated by multiplying the `fu_ture` returns by the forecast *ETF* portfolio weights.

The *training* data is specified over the *look-back* intervals, and the forecast weights are applied to the future data defined by the *look-forward* intervals.




```
> # Optimal look_back
> look_back <- look_backs[which.max(pro_file)]
> pnl_s <- backtest_momentum(returns=re_turns,
+   look_back=look_back, end_p=end_p,
+   perform_ance=perform_ance, with_weights=TRUE)
> tail(pnl_s)
> # Calculate the wealth of momentum returns
> ret_mom <- pnl_s[, 1]
> weal_th <- xts::xts(cbind(all_weather, ret_mom), order.by=date_s
> colnames(weal_th) <- c("All-Weather", "Momentum")
> cor(weal_th)
```

```
> # Plot dygraph of the momentum strategy returns
> dygraphs::dygraph(cumsum(weal_th), main="Monthly Momentum Strategy
+   dyOptions(colors=c("blue", "red"), strokeWidth=2) %>%
+   dyLegend(show="always", width=500)
> # Or
> plot_theme <- chart_theme()
> plot_theme$col$line.col <- c("orange", "blue")
> quantmod::chart_Series(cumsum(weal_th), theme=plot_theme, lwd=2,
+        name="Momentum PnL")
> legend("topleft", legend=colnames(weal_th),
+   inset=0.1, bg="white", lty=1, lwd=6,
+   col=plot_theme$col$line.col, bty="n")
```

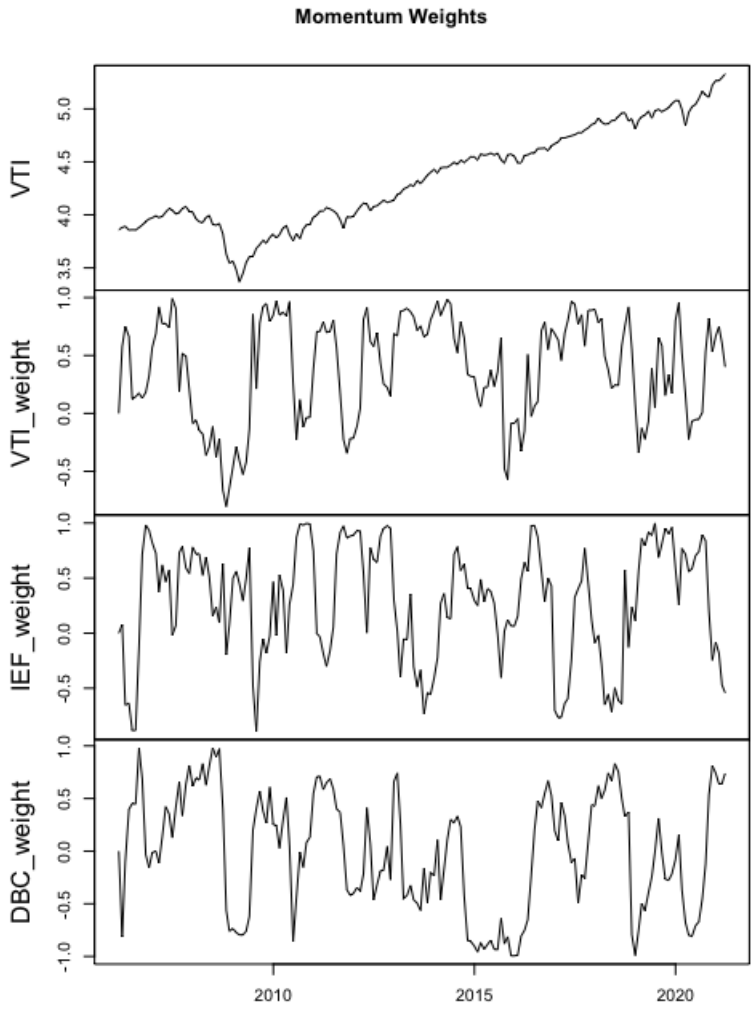
Time Series of Momentum Portfolio Weights

In *momentum* strategies, the portfolio weights are adjusted over time to be proportional to the past performance of the assets.

This way *momentum* strategies switch their weights to the best performing assets.

The weights are scaled to limit the portfolio *leverage* and its market *beta*.

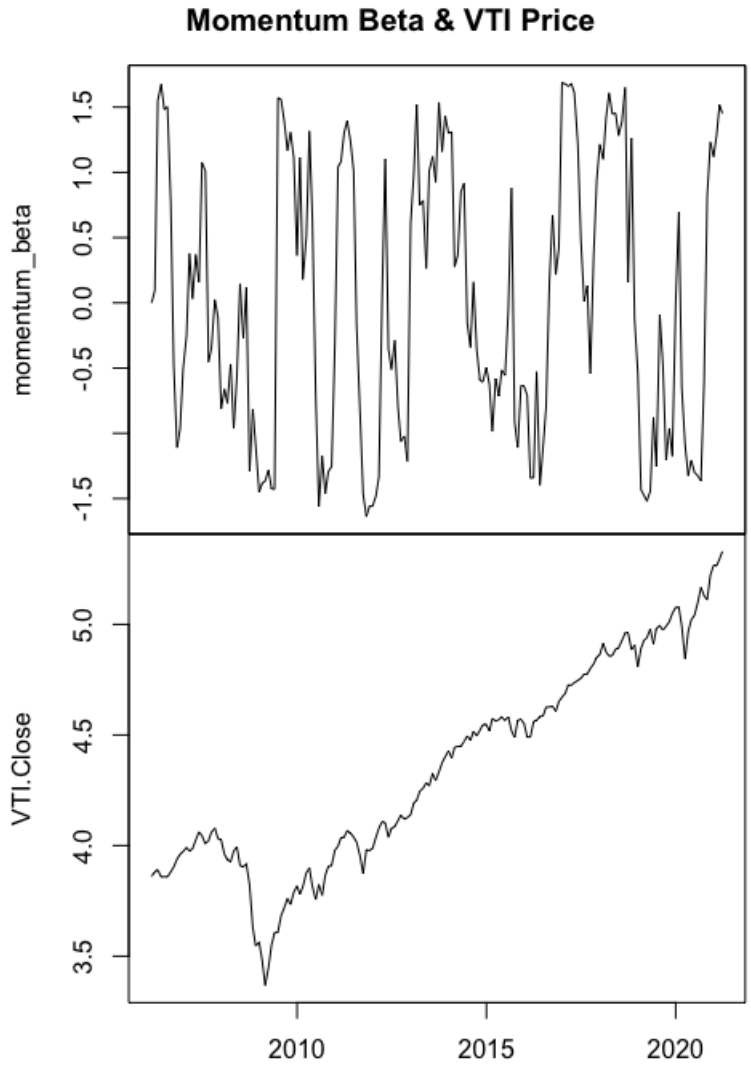
```
> # Plot the momentum portfolio weights
> weight_s <- pnl_s[, -1]
> vt_i <- log(quantmod::Cl(rutils::etf_env$VTI[date_s]))
> colnames(vt_i) <- "VTI"
> da_ta <- cbind(vt_i, weight_s)
> da_ta <- na.omit(da_ta)
> colnames(da_ta)[2:NCOL(pnl_s)] <- paste0(colnames(weight_s), "_we
> zoo::plot.zoo(da_ta, xlab=NULL, main="Momentum Weights")
```



Momentum Strategy Market Beta

The *momentum* strategy market beta can be calculated by multiplying the *ETF* betas by the *ETF* portfolio weights.

```
> # Calculate ETF betas
> betas_etf <- sapply(re_turns, function(x)
+   cov(re_turns$VTI, x)/var(x))
> # Momentum beta is equal weights times ETF betas
> beta_s <- weight_s %*% betas_etf
> beta_s <- xts::xts(beta_s, order.by=date_s)
> colnames(beta_s) <- "momentum_beta"
> da_ta <- cbind(beta_s, vt_i)
> zoo::plot.zoo(da_ta,
+   oma = c(3, 1, 3, 0), mar = c(0, 4, 0, 1),
+   main="Momentum Beta & VTI Price", xlab="")
```

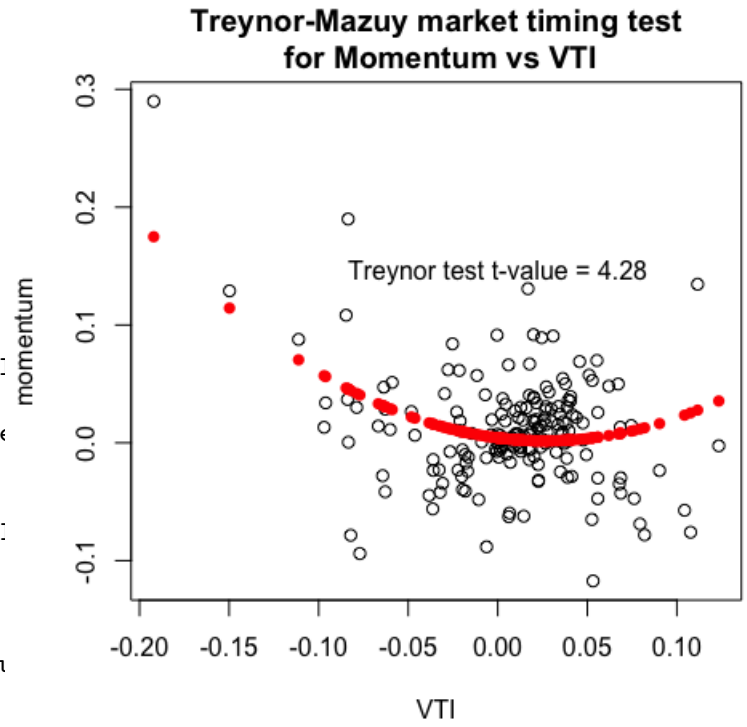


Momentum Strategy Market Timing Skill

Market timing skill is the ability to forecast the direction and magnitude of market returns.

The *Treynor-Mazuy* test shows that the *momentum* strategy has significant *market timing* skill.

```
> # Merton-Henriksson test
> vt_i <- rutils::diff_it(vt_i)
> de_sign <- cbind(VTI=vt_i, 0.5*(vt_i+abs(vt_i)), vt_i^2)
> colnames(de_sign)[2:3] <- c("merton", "treynor")
> mod_el <- lm(ret_mom ~ VTI + merton, data=de_sign); summary(mod_el
> # Treynor-Mazuy test
> mod_el <- lm(ret_mom ~ VTI + treynor, data=de_sign); summary(mod_e
> # Plot residual scatterplot
> plot.default(x=vt_i, y=ret_mom, xlab="VTI", ylab="momentum")
> title(main="Treynor-Mazuy market timing test\n for Momentum vs VTI
> # Plot fitted (predicted) response values
> points.default(x=vt_i, y=mod_el$fitted.values, pch=16, col="red")
> residual_s <- mod_el$residuals
> text(x=0.0, y=max(residual_s), paste("Treynor test t-value =", rou
```



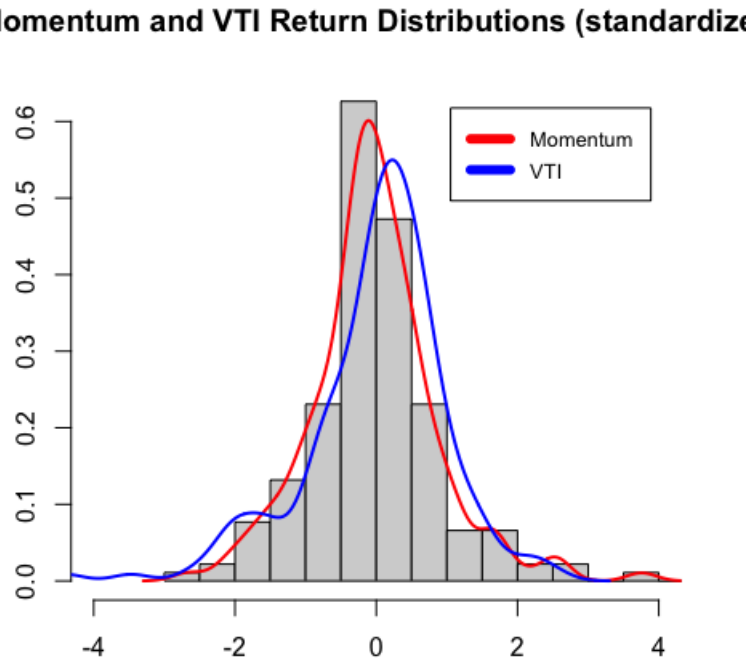
Skewness of Momentum Strategy Returns

Most assets with *positive returns* suffer from *negative skewness*.

The *momentum* strategy returns have more positive skewness compared to the negative skewness of *VTI*.

The *momentum* strategy is a genuine *market anomaly*, because it has both positive returns and positive skewness.

```
> # Standardize the returns
> ret_mom_std <- (ret_mom-mean(ret_mom))/sd(ret_mom)
> vt_i <- (vt_i-mean(vt_i))/sd(vt_i)
> # Calculate skewness and kurtosis
> apply(cbind(ret_mom_std, vt_i), 2, function(x)
+   sapply(c(skew=3, kurt=4),
+     function(e) sum(x^e)))/n_rows
```



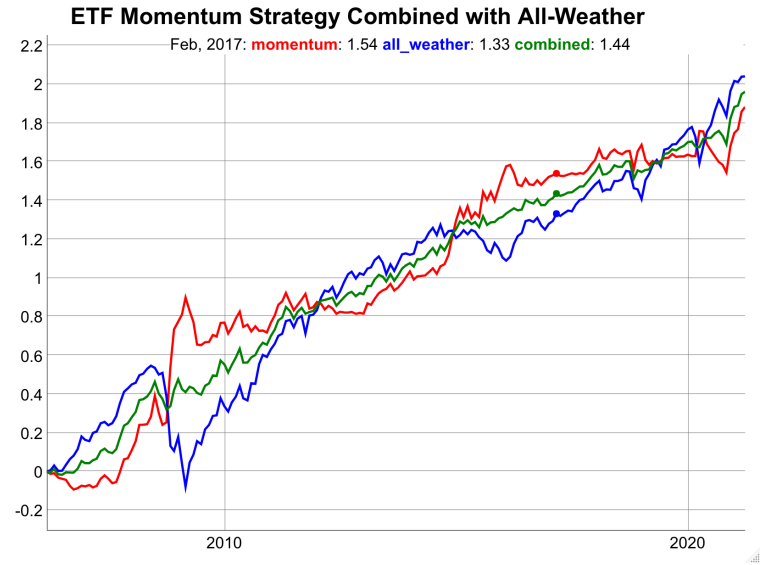

```
> # Plot histogram
> hist(ret_mom_std, breaks=30,
+   main="Momentum and VTI Return Distributions (standardized",
+   xlim=c(-4, 4),
+   xlab="", ylab="", freq=FALSE)
> # Draw kernel density of histogram
> lines(density(ret_mom_std), col='red', lwd=2)
> lines(density(vt_i), col='blue', lwd=2)
> # Add legend
> legend("topright", inset=0.05, cex=0.8, title=NULL,
+  leg=c("Momentum", "VTI"),
+  lwd=6, bg="white", col=c("red", "blue"))
```

Combining Momentum with the *All-Weather* Portfolio

The *momentum* strategy has attractive returns compared to a static buy-and-hold strategy.

But the *momentum* strategy suffers from draw-downs called *momentum crashes*, especially after the market rallies from a sharp-sell-off.

This suggests that combining the *momentum* strategy with a static buy-and-hold strategy can achieve significant diversification of risk.




```
> # Combine momentum strategy with all-weather
> all_weather <- sd(ret_mom)*all_weather/sd(all_weather)
> weal_th <- cbind(ret_mom, all_weather, 0.5*(ret_mom + all_weather)
> colnames(weal_th) <- c("momentum", "all_weather", "combined")
> # Calculate strategy annualized Sharpe ratios
> apply(weal_th, MARGIN=2, function(x) {
+   sqrt(12)*sum(x)/sd(x)/NROW(x)
+ })  # end apply
> # Calculate strategy correlations
> cor(weal_th)
> # Calculate cumulative wealth
> weal_th <- xts::xts(weal_th, date_s)
```

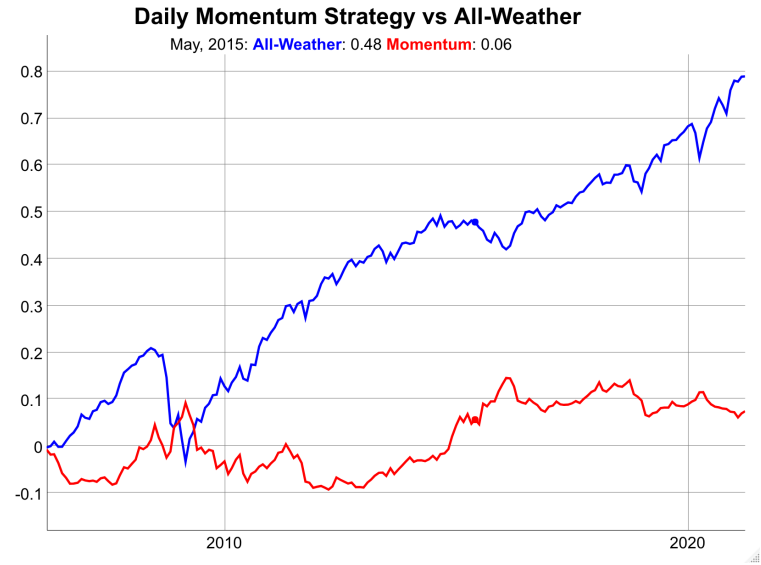
```
> # Plot ETF momentum strategy combined with All-Weather
> dygraphs::dygraph(cumsum(weal_th), main="ETF Momentum Strategy Co
+   dyOptions(colors=c("red", "blue", "green"), strokeWidth=2) %>%
+   dyLegend(show="always", width=500)
> # Or
> plot_theme <- chart_theme()
> plot_theme$col$line.col <- c("green", "blue", "red")
> quantmod::chart_Series(weal_th, theme=plot_theme,
+        name="ETF Momentum Strategy Combined with All-Weather")
> legend("topleft", legend=colnames(weal_th),
+   inset=0.1, bg="white", lty=1, lwd=6,
+   col=plot_theme$col$line.col, bty="n")
```

Momentum Strategy With Daily Rebalancing

A momentum strategy with *daily* rebalancing can't be practically backtested using `apply()` loops because they are too slow.

The package *roll* contains extremely fast functions for calculating rolling aggregations using compiled `C++` code.

The momentum strategy with *daily* rebalancing performs worse than the strategy with *monthly* rebalancing.




```
> # Calculate rolling variance
> look_back <- 252
> vari_ance <- roll::roll_var(re_turns, width=look_back, min_obs=1)
> vari_ance[1, ] <- 1
> # Calculate rolling Sharpe
> pas_t <- roll::roll_mean(re_turns, width=look_back, min_obs=1)
> weight_s <- pas_t/sqrt(vari_ance)
> weight_s <- weight_s/sqrt(rowSums(weight_s^2))
> weight_s <- rutils::lag_it(weight_s)
> sum(is.na(weight_s))
> # Calculate momentum profits and losses
> pnl_s <- rowMeans(weight_s*re_turns)
```

```
> # Calculate transaction costs
> bid_offer <- 0.001
> cost_s <- 0.5*bid_offer*rowSums(abs(rutils::diff_it(weight_s)))
> pnl_s <- (pnl_s - cost_s)
> # Define all-weather benchmark
> weights_aw <- c(0.30, 0.55, 0.15)
> all_weather <- re_turns %*% weights_aw
> # Calculate the wealth of momentum returns
> weal_th <- xts::xts(cbind(all_weather, pnl_s), order.by=index(re_t
> colnames(weal_th) <- c("All-Weather", "Momentum")
> cor(weal_th)
> # Plot dygraph of the momentum strategy returns
> dygraphs::dygraph(cumsum(weal_th)[date_s], main="Daily Momentum St
+   dyOptions(colors=c("blue", "red"), strokeWidth=2) %>%
+   dyLegend(show="always", width=500)
```

Backtesting Functional for Daily Momentum Strategy

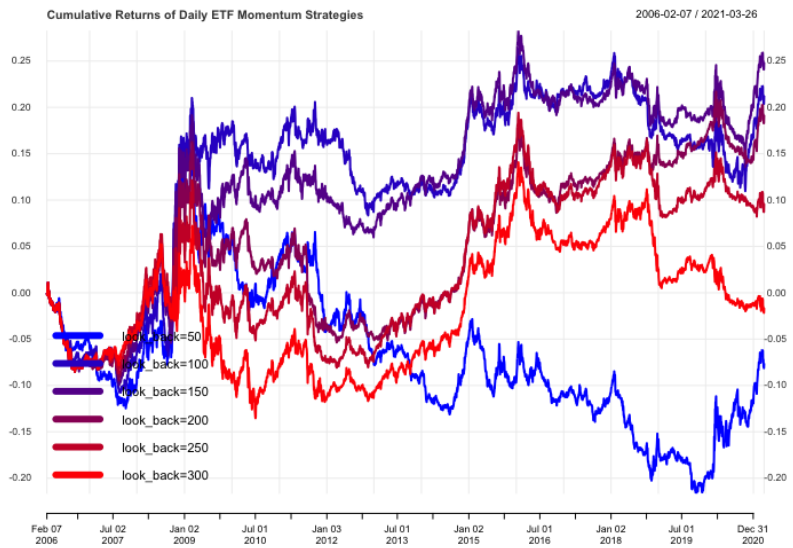
```
> # If trend=(-1) then it backtests a mean reverting strategy
> momentum_daily <- function(returns, look_back=252, bid_offer=0.001, trend=1, ...) {
+   stopifnot("package:quantmod" %in% search() || require("quantmod", quietly=TRUE))
+   # Calculate rolling variance
+   vari_ance <- roll::roll_var(returns, width=look_back, min_obs=1)
+   vari_ance[1, ] <- 1
+   vari_ance[vari_ance <= 0] <- 1
+ # Calculate rolling Sharpe
+   pas_t <- roll::roll_mean(returns, width=look_back, min_obs=1)
+   weight_s <- pas_t/sqrt(vari_ance)
+   weight_s <- weight_s/sqrt(rowSums(weight_s^2))
+   weight_s <- rutils::lag_it(weight_s)
+   # Calculate momentum profits and losses
+   pnl_s <- trend*rowMeans(weight_s*returns)
+   # Calculate transaction costs
+   cost_s <- 0.5*bid_offer*rowSums(abs(rutils::diff_it(weight_s)))
+   (pnl_s - cost_s)
+ }  # end momentum_daily
```

Multiple Daily ETF Momentum Strategies

Multiple daily ETF *momentum* strategies can be backtested by calling the function `backtest_rolling()` in a loop over a vector of *look-back* parameters.

The *momentum* strategies do not perform well, especially the ones with a small *look-back* parameter.

```
> # Simulate a daily ETF momentum strategy
> source("C:/Develop/lecture_slides/scripts/back_test.R")
> weal_th <- momentum_daily(returns=re_turns, look_back=252,
+   bid_offer=bid_offer)
> # Perform sapply loop over look_backs
> look_backs <- seq(50, 300, by=50)
> weal_th <- sapply(look_backs, momentum_daily,
+   returns=re_turns, bid_offer=bid_offer)
> colnames(weal_th) <- paste0("look_back=", look_backs)
> weal_th <- xts::xts(weal_th, index(re_turns))
> tail(weal_th)
```



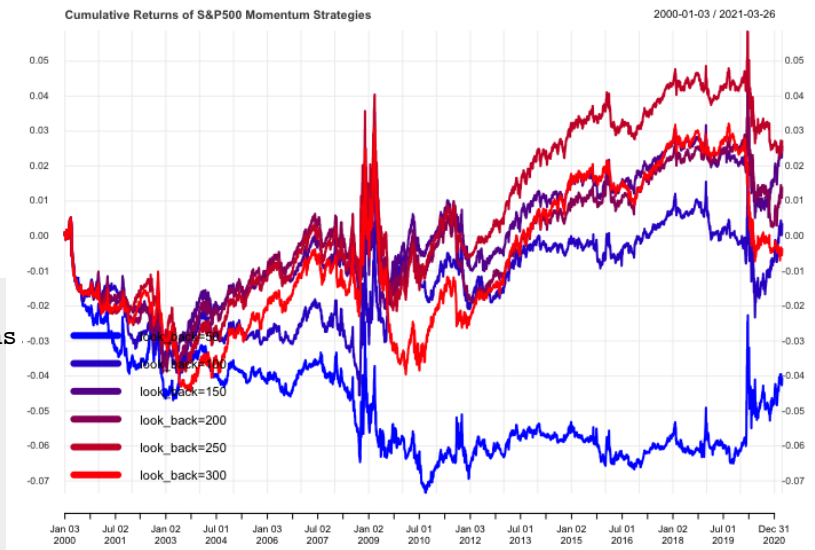

```
> # Plot EWMA strategies with custom line colors
> plot_theme <- chart_theme()
> plot_theme$col$line.col <-
+   colorRampPalette(c("blue", "red"))(NCOL(weal_th))
> quantmod::chart_Series(cumsum(weal_th),
+   theme=plot_theme, name="Cumulative Returns of Daily ETF Momentum
> legend("bottomleft", legend=colnames(weal_th),
+   inset=0.02, bg="white", cex=0.7, lwd=rep(6, NCOL(re_turns)),
+   col=plot_theme$col$line.col, bty="n")
```

Backtesting Multiple S&P500 Momentum Strategies

Multiple *S&P500 momentum* strategies can be backtested by calling the function `backtest_rolling()` in a loop over a vector of *look-back* parameters.

The *momentum* strategies do not perform well, especially the ones with a small *look-back* parameter.

```
> # Load daily S&P500 percentage stock returns.
> load(file="/Users/jerzy/Develop/lecture_slides/data/sp500_returns
> # Overwrite NA values in returns_100
> returns_100 <- returns_100["2000/"]
> returns_100[1, is.na(returns_100[1, ])] <- 0
> returns_100 <- zoo::na.locf(returns_100, na.rm=FALSE)
> # Simulate a daily S&P500 momentum strategy.
> # Perform sapply loop over look_backs
> look_backs <- seq(50, 300, by=50)
> weal_th <- sapply(look_backs, momentum_daily,
+   returns=returns_100, bid_offer=0)
> colnames(weal_th) <- paste0("look_back=", look_backs)
> weal_th <- xts::xts(weal_th, index(returns_100))
```




```
> # Plot daily S&P500 momentum strategies with custom line colors
> plot_theme <- chart_theme()
> plot_theme$col$line.col <- colorRampPalette(c("blue", "red"))(NCOL
> quantmod::chart_Series(cumsum(weal_th),
+   theme=plot_theme, name="Cumulative Returns of S&P500 Momentum St
> legend("bottomleft", legend=colnames(weal_th),
+   inset=0.02, bg="white", cex=0.7, lwd=rep(6, NCOL(re_turns)),
+   col=plot_theme$col$line.col, bty="n")
```

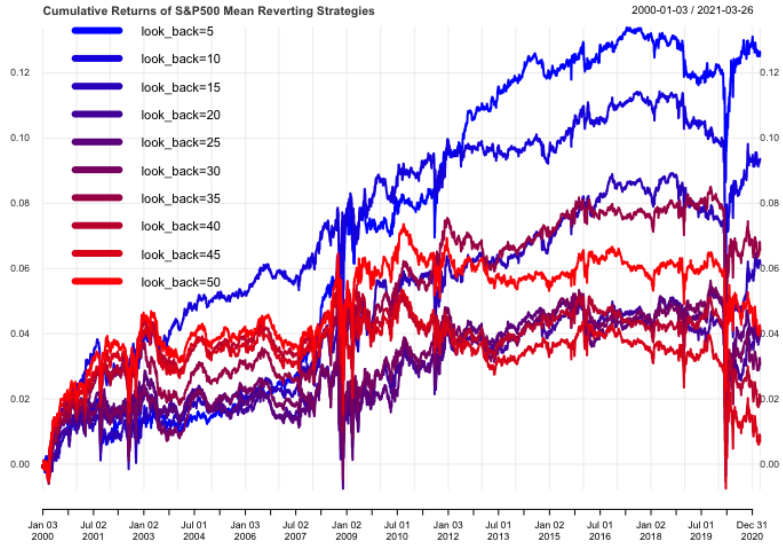
Backtesting Multiple S&P500 *Mean Reverting* Strategies

Multiple *S&P500 mean reverting* strategies can be backtested by calling the function `backtest_rolling()` in a loop over a vector of *look-back* parameters.

The *mean reverting* strategies for the *S&P500* constituents perform the best for short *look-back* parameters.

The *mean reverting* strategies had their best performance prior to the 2008 financial crisis.

```
> # Perform sapply loop over look_backs
> look_backs <- seq(5, 50, by=5)
> weal_th <- sapply(look_backs, momentum_daily,
+   returns=returns_100, bid_offer=0, trend=(-1))
> colnames(weal_th) <- paste0("look_back=", look_backs)
> weal_th <- xts::xts(weal_th, index(returns_100))
```

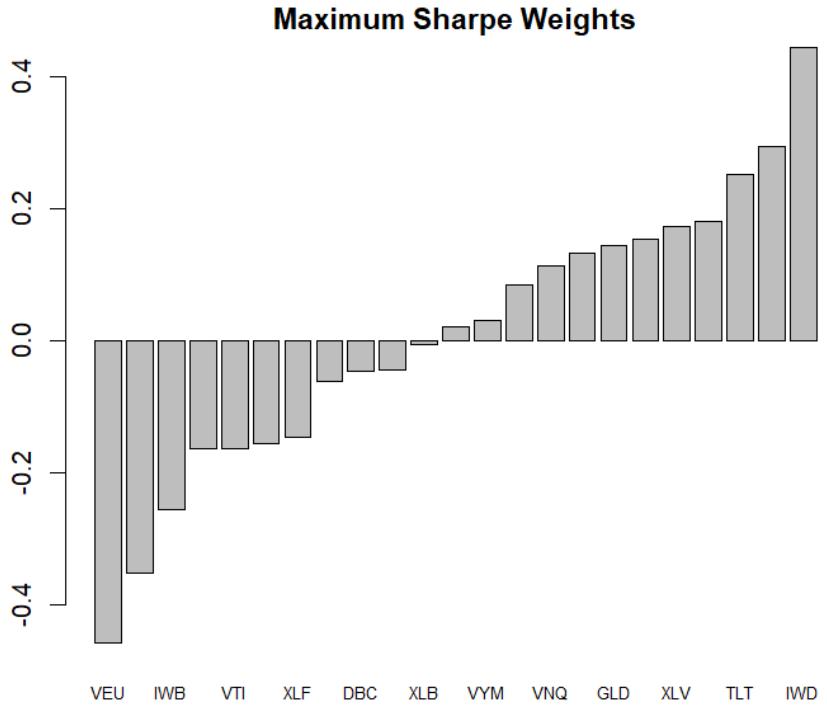


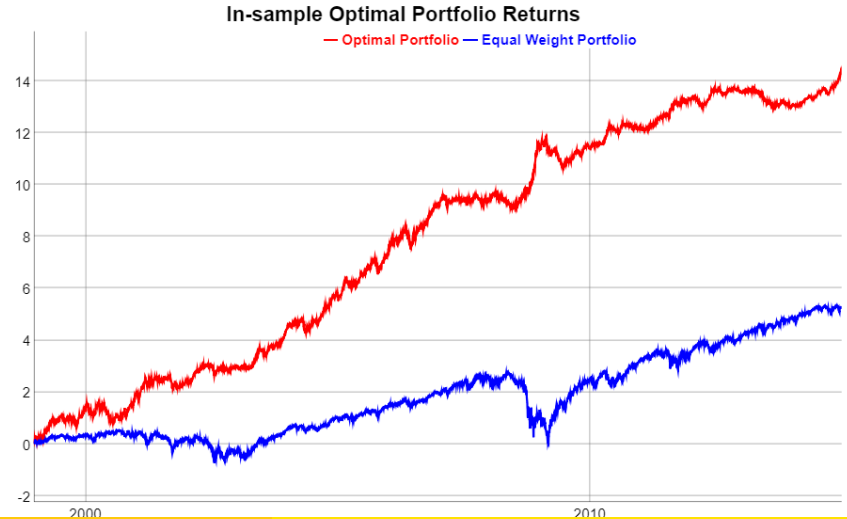

```
> # Plot EWMA strategies with custom line colors
> plot_theme <- chart_theme()
> plot_theme$col$line.col <- colorRampPalette(c("blue", "red"))(NCOL
> quantmod::chart_Series(cumsum(weal_th),
+   theme=plot_theme, name="Cumulative Returns of S&P500 Mean Revert
> legend("topleft", legend=colnames(weal_th),
+   inset=0.05, bg="white", cex=0.7, lwd=rep(6, NCOL(re_turns)),
+   col=plot_theme$col$line.col, bty="n")
```

Portfolio Optimization In-sample

The *ETF* portfolio is overweight bond ETFs, for example *TLT* and *VYM*.

```
> # Select all the ETF symbols except "VXX", "SVXY" and "MTUM"
> sym_bols <- colnames(rutils::etf_env$re_turns)
> sym_bols <- sym_bols[!(sym_bols %in% c("VXX", "SVXY", "MTUM"))]
> # Extract columns of rutils::etf_env$re_turns and overwrite NA va
> re_turns <- rutils::etf_env$re_turns[, sym_bols]
> n_assets <- NCOL(re_turns)
> # re_turns <- na.omit(re_turns)
> re_turns[1, is.na(re_turns[1, ])] <- 0
> re_turns <- zoo::na.locf(re_turns, na.rm=FALSE)
> # Returns in excess of risk-free rate
> risk_free <- 0.03/252
> ex_cess <- (re_turns - risk_free)
> # Maximum Sharpe weights in-sample interval
> in_verse <- MASS::ginv(cov(re_turns["/2014"]))
> weight_s <- in_verse %*% colMeans(ex_cess["/2014"])
> weight_s <- drop(weight_s/sqrt(sum(weight_s^2)))
> names(weight_s) <- colnames(re_turns)
> # Plot portfolio weights
> x11(width=6, height=5)
> par(mar=c(3, 3, 2, 1), oma=c(0, 0, 0, 0), mgp=c(2, 1, 0))
> barplot(sort(weight_s), main="Maximum Sharpe Weights", cex.names=(
> # Calculate portfolio returns
> rets_is <- re_turns["/2014"]
> portf_is <- xts::xts(rets_is %*% weight_s, index(rets_is))
> in_dex <- xts::xts(rowSums(rets_is)/sqrt(n_assets), index(rets_is)
> portf_is <- portf_is*sd(in_dex)/sd(portf_is)
> # Plot cumulative portfolio returns
> weal_th <- cumsum(cbind(portf_is, in_dex))
> colnames(weal_th) <- c("Optimal Portfolio", "Equal Weight Portfoli
> dygraphs::dygraph(weal_th, main="In-sample Optimal Portfolio Retur
+   dyOptions(colors=c("red", "blue"), strokeWidth=2) %>%
+   dyLegend(width=500)
```





Portfolio Optimization Strategy for ETFs

The *portfolio optimization* strategy invests in the best performing portfolio in the past *in-sample* interval, expecting that it will continue performing well *out-of-sample*.

The *portfolio optimization* strategy consists of:

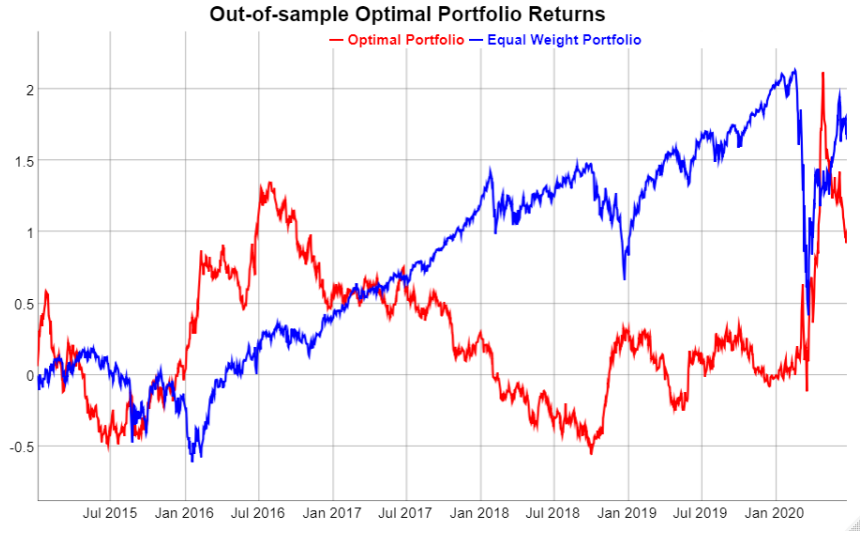
1. Calculating the maximum Sharpe ratio portfolio weights in the *in-sample* interval,
2. Applying the weights and calculating the portfolio returns in the *out-of-sample* interval.

The optimal portfolio weights $\mathbf{w}$ are calculated using the past in-sample excess returns $\mu = \mathbf{r} - r_f$ (in excess of the risk-free rate r_f):

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

This strategy has performed well for *ETF* portfolios because of the consistent performance of bond ETFs, like *TLT* and *VYM*.

```
> # Out-of-sample portfolio returns
> rets_os <- re_turns["2015/"]
> portf_os <- xts::xts(rets_os %*% weight_s, index(rets_os))
> in_dex <- xts::xts(rowSums(rets_os)/sqrt(n_assets), index(rets_os))
> portf_os <- portf_os*sd(in_dex)/sd(portf_os)
```

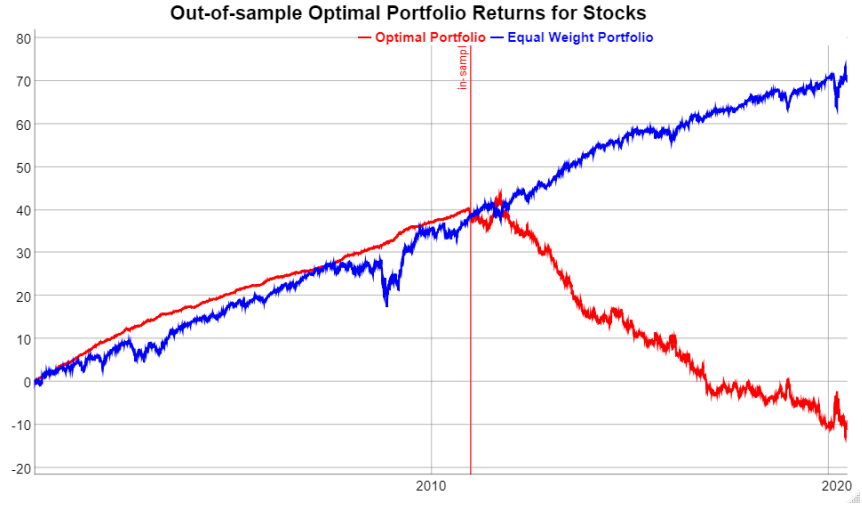



```
> # Plot cumulative portfolio returns
> weal_th <- cumsum(cbind(portf_os, in_dex))
> colnames(weal_th) <- c("Optimal Portfolio", "Equal Weight Portfoli
> dygraphs::dygraph(weal_th, main="Out-of-sample Optimal Portfolio P
+   dyOptions(colors=c("red", "blue"), strokeWidth=2) %>%
+   dyLegend(width=500)
```

Portfolio Optimization Strategy for Stocks

The *portfolio optimization* strategy for stocks is *overfitted* in the *in-sample* interval.

Therefore the strategy completely fails in the *out-of-sample* interval.




```
> load("C:/Develop/lecture_slides/data/sp500_returns.RData")
> # Overwrite NA values in re_turns
> re_turns <- re_turns["2000/"]
> n_assets <- NCOL(re_turns)
> re_turns[1, is.na(re_turns[1, ])] <- 0
> re_turns <- zoo::na.locf(re_turns, na.rm=FALSE)
> risk_free <- 0.03/252
> ex_cess <- (re_turns - risk_free)
> rets_is <- re_turns["/2010"]
> rets_os <- re_turns["2011/"]
> # Maximum Sharpe weights in-sample interval
> cov_mat <- cov(rets_is)
> in_verse <- MASS::ginv(cov_mat)
> weight_s <- in_verse %*% colMeans(ex_cess["/2010"])
> weight_s <- drop(weight_s/sqrt(sum(weight_s^2)))
> names(weight_s) <- colnames(re_turns)
> # Calculate portfolio returns
> portf_is <- xts::xts(rets_is %*% weight_s, index(rets_is))
> portf_os <- xts::xts(rets_os %*% weight_s, index(rets_os))
> in_dex <- xts::xts(rowSums(re_turns)/sqrt(n_assets), index(re_turns))
```

```
> # Plot cumulative portfolio returns
> weal_th <- rbind(portf_is, portf_os)
> weal_th <- weal_th*sd(in_dex)/sd(weal_th)
> weal_th <- cumsum(cbind(weal_th, in_dex))
> colnames(weal_th) <- c("Optimal Portfolio", "Equal Weight Portfoli
> dygraphs::dygraph(weal_th, main="Out-of-sample Optimal Portfolio P
+   dyOptions(colors=c("red", "blue"), strokeWidth=2) %>%
+   dyEvent(index(last(rets_is[, 1])), label="in-sample", strokePatt
+   dyLegend(width=500)
```

Regularized Inverse of Singular Covariance Matrices

The *regularization* technique allows calculating the inverse of *singular* covariance matrices while reducing the effects of statistical noise.

If the number of time periods of returns is less than the number of assets (columns), then the covariance matrix of returns is *singular*, and some of its *eigenvalues* are zero, so it doesn't have an inverse.

The *regularized* inverse $\mathbb{C}_n^{-1}$ is calculated by removing the higher order eigenvalues that are almost zero, 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.

```
> # Create rectangular matrix with collinear columns
> ran_dom <- matrix(rnorm(10*8), nc=10)
> # Calculate covariance matrix
> cov_mat <- cov(ran_dom)
> # Calculate inverse of cov_mat - error
> in_verse <- solve(cov_mat)
> # Calculate regularized inverse of cov_mat
> in_verse <- MASS::ginv(cov_mat)
> # Verify inverse property of mat_rix
> all.equal(cov_mat, cov_mat %*% in_verse %*% cov_mat)
> # Perform eigen decomposition
> ei_gen <- eigen(cov_mat)
> eigen_vec <- ei_gen$vectors
> eigen_val <- ei_gen$values
> # Set tolerance for determining zero singular values
> to_l <- sqrt(.Machine$double.eps)
> # Calculate regularized inverse matrix
> not_zero <- (eigen_val > (to_l * eigen_val[1]))
> reg_inverse <- eigen_vec[, not_zero] %*%
+   (t(eigen_vec[, not_zero]) / eigen_val[not_zero])
> # Verify inverse property of mat_rix
> all.equal(in_verse, reg_inverse)
```

Regularized Inverse of the Covariance Matrix

For a portfolio of *S&P500* stocks, the number return columns is very large, which may make the covariance matrix of returns simgular.

Removing the very small higher order eigenvalues also reduces the propagation of statistical noise and improves the signal-to-noise ratio.

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

Even though the *regularized* inverse $\mathbb{C}_n^{-1}$ does not satisfy the matrix inverse property, its out-of-sample forecasts may be more accurate than those using the actual inverse matrix.

The parameter `max_eigen` specifies the number of eigenvalues used for calculating the *regularized* inverse of the covariance matrix of returns.

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

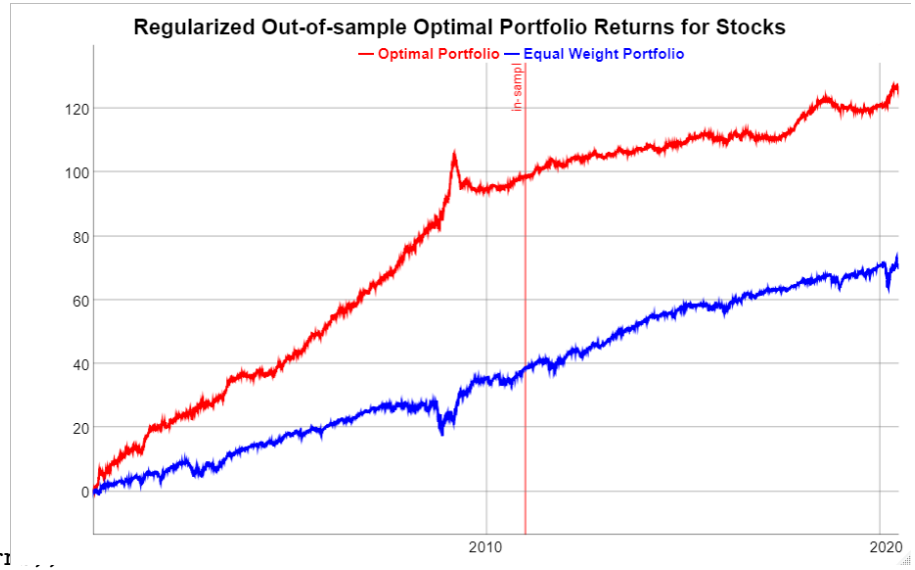
```
> # Calculate in-sample covariance matrix
> cov_mat <- cov(rets_is)
> ei_gen <- eigen(cov_mat)
> eigen_vec <- ei_gen$vectors
> eigen_val <- ei_gen$values
> # Calculate regularized inverse of covariance matrix
> max_eigen <- 21
> in_verse <- eigen_vec[, 1:max_eigen] %*%
+   (t(eigen_vec[, 1:max_eigen]) / ei_gen$values[1:max_eigen])
```

Regularized Portfolio Optimization Strategy for Stocks

The *out-of-sample* performance of the *portfolio optimization* strategy is greatly improved by regularizing the inverse of the covariance matrix.

The *in-sample* performance is worse because regularization reduces *overfitting*.

```
> # Calculate portfolio weights
> weight_s <- in_verse %*% colMeans(ex_cess["/2010"])
> weight_s <- drop(weight_s/sqrt(sum(weight_s^2)))
> names(weight_s) <- colnames(re_turns)
> # Calculate portfolio returns
> portf_is <- xts::xts(rets_is %*% weight_s, index(rets_is))
> portf_os <- xts::xts(rets_os %*% weight_s, index(rets_os))
> in_dex <- xts::xts(rowSums(re_turns)/sqrt(n_assets), index(re_turn
```




```
> # Plot cumulative portfolio returns
> weal_th <- rbind(portf_is, portf_os)
> weal_th <- weal_th*sd(in_dex)/sd(weal_th)
> weal_th <- cumsum(cbind(weal_th, in_dex))
> colnames(weal_th) <- c("Optimal Portfolio", "Equal Weight Portfoli
> dygraphs::dygraph(weal_th, main="Regularized Out-of-sample Optimal
+   dyOptions(colors=c("red", "blue"), strokeWidth=2) %>%
+   dyEvent(index(last(rets_is[, 1])), label="in-sample", strokePatt
+   dyLegend(width=500)
```

Optimal Portfolio Weights With Return Shrinkage

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

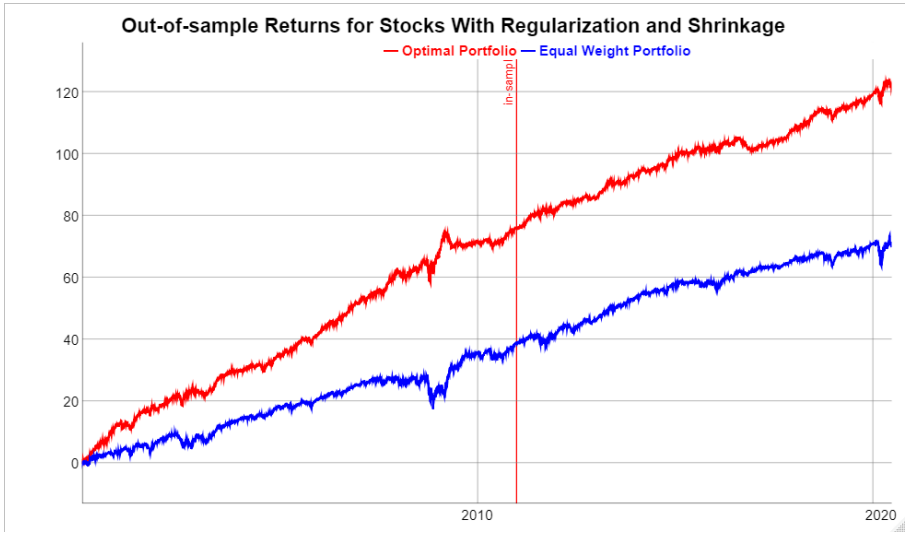
$$r_i = (1 - \alpha)\, r_i + \alpha\, \bar{r}$$

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

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
> rets_mean <- colMeans(rets_is) - risk_free
> al_pha <- 0.7
> rets_mean <- (1 - al_pha)*rets_mean + al_pha*mean(rets_mean)
```




```
> # Calculate portfolio weights
> weight_s <- in_verse %*% rets_mean
> weight_s <- drop(weight_s/sqrt(sum(weight_s^2)))
> # Calculate portfolio returns
> portf_is <- xts::xts(rets_is %*% weight_s, index(rets_is))
> portf_os <- xts::xts(rets_os %*% weight_s, index(rets_os))
> # Plot cumulative portfolio returns
> weal_th <- rbind(portf_is, portf_os)
> weal_th <- weal_th*sd(in_dex)/sd(weal_th)
> weal_th <- cumsum(cbind(weal_th, in_dex))
> colnames(weal_th) <- c("Optimal Portfolio", "Equal Weight Portfoli
> dygraphs::dygraph(weal_th, main="Out-of-sample Returns for Stocks
+   dyOptions(colors=c("red", "blue"), strokeWidth=2) %>%
+   dyEvent(index(last(rets_is[, 1])), label="in-sample", strokePatt
+   dyLegend(width=500)
```

Fast Covariance Matrix Inverse Using *RcppArmadillo*

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

```
// Rcpp header with information for C++ compiler
// [[Rcpp::depends(RcppArmadillo)]]
#include <RcppArmadillo.h>  // include Rcpp C++ header files
using namespace std;
using namespace Rcpp; // use Rcpp C++ namespace
using namespace arma;

//' @export
// [[Rcpp::export]]
arma::mat calc_inv(const arma::mat& mat_rix, const arma::uword& max_eigen) {
  arma::mat eigen_vec;
  arma::vec eigen_val;

  arma::eig_sym(eigen_val, eigen_vec, cov(mat_rix));
  eigen_vec = eigen_vec.cols(eigen_vec.n_cols-max_eigen, eigen_vec.n_cols-1);
  eigen_val = 1/eigen_val.subvec(eigen_val.n_elem-max_eigen, eigen_val.n_elem-1);

  return eigen_vec*diagmat(eigen_val)*eigen_vec.t();

}  // end calc_inv
```

```
> library(RcppArmadillo)
> # Source Rcpp functions from file
> Rcpp::sourceCpp("C:/Develop/lecture_slides/scripts/calc_weights.cpp")
> # Create random matrix of returns
> mat_rix <- matrix(rnorm(300), nc=5)
> # Regularized inverse of covariance matrix
> max_eigen <- 4
> ei_gen <- eigen(cov(mat_rix))
> cov_inv <- ei_gen$vectors[, 1:max_eigen] %*%
+   (t(ei_gen$vectors[, 1:max_eigen]) / ei_gen$values[1:max_eigen])
> # Regularized inverse using RcppArmadillo
> cov_inv_arma <- calc_inv(mat_rix, max_eigen)
> all.equal(cov_inv, cov_inv_arma)
> # Microbenchmark RcppArmadillo code
```

Portfolio Optimization Using *RcppArmadillo*

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

```
// Fast portfolio optimization using matrix algebra and RcppArmadillo
// Rcpp header with information for C++ compiler
// [[Rcpp::depends(RcppArmadillo)]]
#include <RcppArmadillo.h>  // include Rcpp C++ header files
using namespace std;
using namespace Rcpp; // use Rcpp C++ namespace
using namespace arma;

//' @export
// [[Rcpp::export]]
arma::vec calc_weights(const arma::mat& re_turns,
                       const std::string& typ_e = "max_sharpe",
                       int max_eigen = 1,
                       const double& pro_b = 0.1,
                       const double& al_pha = 0.0,
                       const bool scal_e = true) {
  // Initialize
  arma::vec weight_s(re_turns.n_cols);
  if (max_eigen == 1)  max_eigen = re_turns.n_cols;

  // Calculate weights depending on typ_e
  if (typ_e == "max_sharpe") {
    // Mean returns by columns
    arma::vec mean_cols = arma::trans(arma::mean(re_turns, 0));
    // Shrink mean_cols to the mean of re_turns
    mean_cols = ((1-al_pha)*mean_cols + al_pha*arma::mean(mean_cols));
    // Apply regularized inverse
    weight_s = calc_inv(re_turns, max_eigen)*mean_cols;
  } else if (typ_e == "max_sharpe_median") {
    // Mean returns by columns
    arma::vec mean_cols = arma::trans(arma::median(re_turns, 0));
    // Shrink mean_cols to the mean of re_turns
    mean_cols = ((1-al_pha)*mean_cols + al_pha*arma::median(mean_cols));
    // Apply regularized inverse
    weight_s = calc_inv(re_turns, max_eigen)*mean_cols;
```

Strategy Backtesting Using *RcppArmadillo*

Fast backtesting of strategies can be implemented using *RcppArmadillo*.

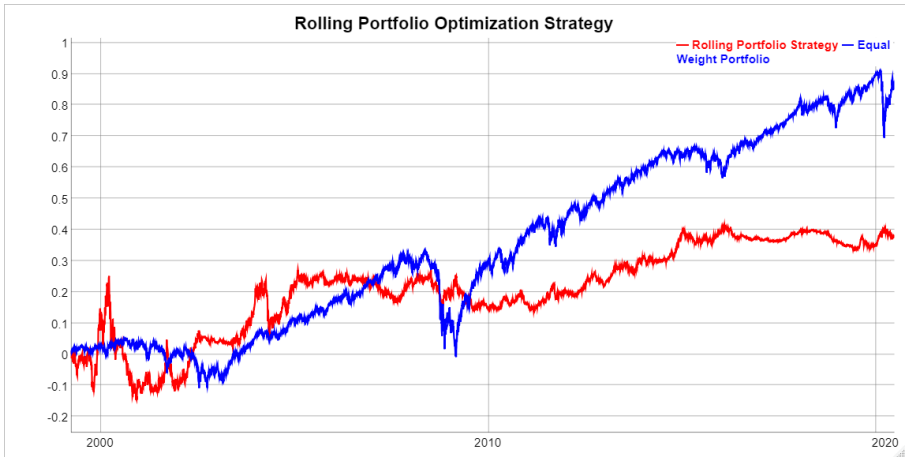
```
// [[Rcpp::depends(RcppArmadillo)]]
#include <RcppArmadillo.h>  // include Rcpp C++ header files
using namespace std;
using namespace Rcpp; // use Rcpp C++ namespace
using namespace arma;

arma::mat back_test(const arma::mat& ex_cess, // Portfolio excess returns
                    const arma::mat& re_turns, // Portfolio returns
                    const arma::uvec& start_p,
                    const arma::uvec& end_p,
                    const std::string& typ_e = "max_sharpe",
                    const arma::uword& max_eigen = 1,
                    const double& pro_b = 0.1,
                    const double& al_pha = 0,
                    const bool& scal_e = true,
                    const double& co_eff = 1.0,
                    const double& bid_offer = 0.0) {
  arma::vec pnl_s = zeros(re_turns.n_rows);
  arma::vec weights_past = zeros(re_turns.n_cols);
  arma::vec weight_s(re_turns.n_cols);

  // Perform loop over the end_p
  for (arma::uword it=1; it < end_p.size(); it++) {
    // cout << "it: " << it << endl;
    // Calculate portfolio weights
    weight_s = co_eff*calc_weights(ex_cess.rows(start_p(it-1), end_p(it-1)), typ_e, max_eigen, pro_b, al_pha, s
    // Calculate out-of-sample returns
    pnl_s.subvec(end_p(it-1)+1, end_p(it)) = re_turns.rows(end_p(it-1)+1, end_p(it))*weight_s;
    // Add transaction costs
    pnl_s.row(end_p(it-1)+1) -= bid_offer*sum(abs(weight_s - weights_past))/2;
    weights_past = weight_s;
  }  // end for
  // Return the strategy returns
  return pnl_s;
}  // end back_test
```

Rolling Portfolio Optimization Strategy

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




```
> # Calculate vector of monthly end points and start points
> end_p <- rutils::calc_endpoints(re_turns, inter_val="months")
> end_p <- end_p[end_p > 2*NCOL(re_turns)]
> n_rows <- NROW(end_p)
> look_back <- 24
> start_p <- c(rep_len(0, look_back-1),
+        end_p[1:(n_rows-look_back+1)])
```

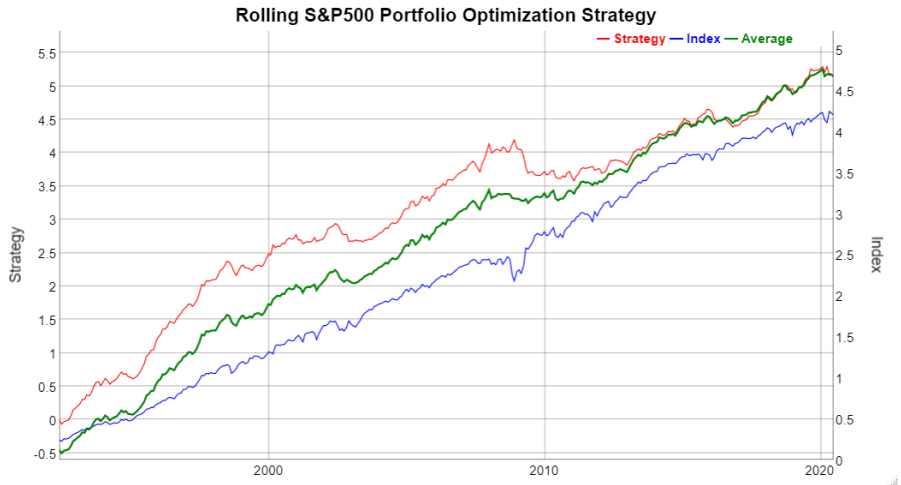
```
> # Perform loop over end points
> rets_portf <- lapply(2:n_rows, function(i) {
+     # Subset the ex_cess returns
+     ex_cess <- ex_cess[start_p[i-1]:end_p[i-1], ]
+     in_verse <- MASS::ginv(cov(ex_cess))
+     # Calculate the maximum Sharpe ratio portfolio weights
+     weight_s <- in_verse %*% colMeans(ex_cess)
+     weight_s <- drop(weight_s/sqrt(sum(weight_s^2)))
+     # Calculate the out-of-sample portfolio returns
+     re_turns <- re_turns[(end_p[i-1]+1):end_p[i], ]
+     xts::xts(re_turns %*% weight_s, index(re_turns))
+ })  # end lapply
> rets_portf <- rutils::do_call(rbind, rets_portf)
> # Plot cumulative strategy returns
> in_dex <- xts::xts(rowSums(re_turns)/sqrt(n_assets), index(re_turn
> weal_th <- cumsum(na.omit(cbind(rets_portf, in_dex*sd(rets_portf)/
> colnames(weal_th) <- c("Rolling Portfolio Strategy", "Equal Weight
> dygraphs::dygraph(weal_th, main="Rolling Portfolio Optimization St
+   dyOptions(colors=c("red", "blue"), strokeWidth=2) %>%
+   dyLegend(show="always", width=500)
```

Rolling Portfolio Optimization Strategy for S&P500

In the rolling portfolio optimization strategy the portfolio weights are adjusted to their optimal values at every end point.

A portfolio optimization is performed using past data, and the optimal portfolio weights are applied out-of-sample in the next interval.

The weights are scaled to match the volatility of the equally weighted portfolio, and are kept constant until the next end point.



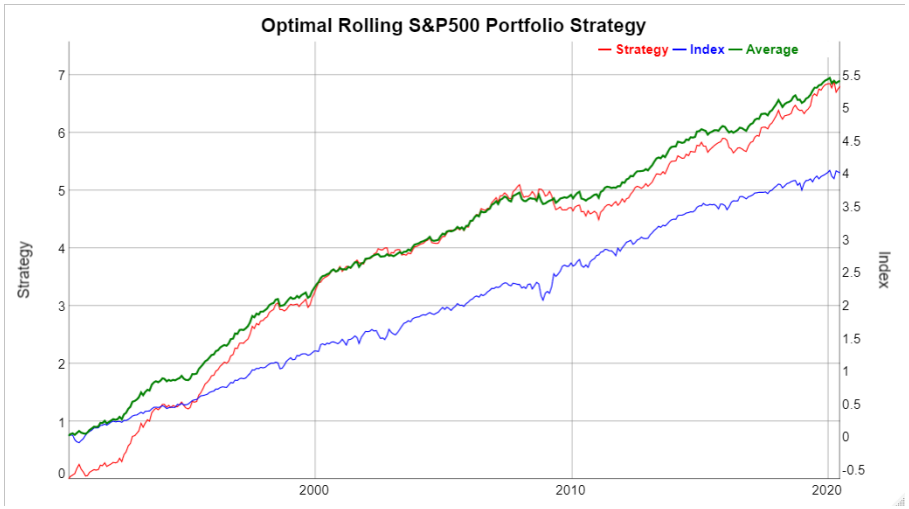

```
> load("C:/Develop/lecture_slides/data/sp500_returns.RData")
> # Overwrite NA values in re_turns
> returns_100[1, is.na(returns_100[1, ])] <- 0
> returns_100 <- zoo::na.locf(returns_100, na.rm=FALSE)
> n_cols <- NCOL(returns_100) ; date_s <- index(returns_100)
> # Define monthly end points
> end_p <- rutils::calc_endpoints(returns_100, inter_val="months")
> end_p <- end_p[end_p > (n_cols+1)]
> n_rows <- NROW(end_p) ; look_back <- 12
> start_p <- c(rep_len(0, look_back-1), end_p[1:(n_rows-look_back+1
> end_p <- (end_p - 1)
> start_p <- (start_p - 1)
> start_p[start_p < 0] <- 0
> al_pha <- 0.7 ; max_eigen <- 21
```

```
> # Perform backtest in Rcpp
> pnl_s <- HighFreq::back_test(typ_e="max_sharpe",
+   ex_cess=returns_100, returns=returns_100,
+   start_p=start_p, end_p=end_p,
+   al_pha=al_pha, max_eigen=max_eigen)
> # Calculate returns on equal weight portfolio
> in_dex <- xts::xts(rowMeans(returns_100), index(returns_100))
> # Plot cumulative strategy returns
> weal_th <- cbind(pnl_s, in_dex, (pnl_s+in_dex)/2)
> weal_th <- cumsum(na.omit(weal_th))
> col_names <- c("Strategy", "Index", "Average")
> colnames(weal_th) <- col_names
> dygraphs::dygraph(weal_th[end_p], main="Rolling S&P500 Portfolio
+   dyAxis("y", label=col_names[1], independentTicks=TRUE) %>%
+   dyAxis("y2", label=col_names[2], independentTicks=TRUE) %>%
+   dySeries(name=col_names[1], axis="y", col="red", strokeWidth=1)
+   dySeries(name=col_names[2], axis="y2", col="blue", strokeWidth=
+   dySeries(name=col_names[3], axis="y2", col="green", strokeWidth=
```

Determining Strategy Parameters Using Backtesting

The optimal values of the parameters `max_eigen` and α can be determined using *backtesting*.

```
> # Perform backtest over alphas
> alpha_s <- seq(from=0.01, to=0.91, by=0.1)
> pnl_s <- lapply(alpha_s, function(al_pha) {
+   HighFreq::back_test(typ_e="max_sharpe",
+   ex_cess=returns_100, returns=returns_100,
+   start_p=start_p, end_p=end_p,
+   al_pha=al_pha, max_eigen=max_eigen)
+ })  # end lapply
> pro_file <- sapply(pnl_s, sum)
> plot(x=alpha_s, y=pro_file, t="l", main="Strategy PnL as Function
+   xlab="Shrinkage Intensity Alpha", ylab="pnl")
> al_pha <- alpha_s[which.max(pro_file)]
> pnl_s <- pnl_s[[which.max(pro_file)]]
> # Perform backtest over max_eigens
> max_eigens <- seq(from=3, to=40, by=2)
> pnl_s <- lapply(max_eigens, function(max_eigen) {
+   HighFreq::back_test(typ_e="max_sharpe",
+     ex_cess=returns_100, returns=returns_100,
+     start_p=start_p, end_p=end_p,
+     al_pha=al_pha, max_eigen=max_eigen)
+ })  # end lapply
> pro_file <- sapply(pnl_s, sum)
> plot(x=max_eigens, y=pro_file, t="l", main="Strategy PnL as Funct
+   xlab="Max_eigen", ylab="pnl")
> max_eigen <- max_eigens[which.max(pro_file)]
> pnl_s <- pnl_s[[which.max(pro_file)]]
```

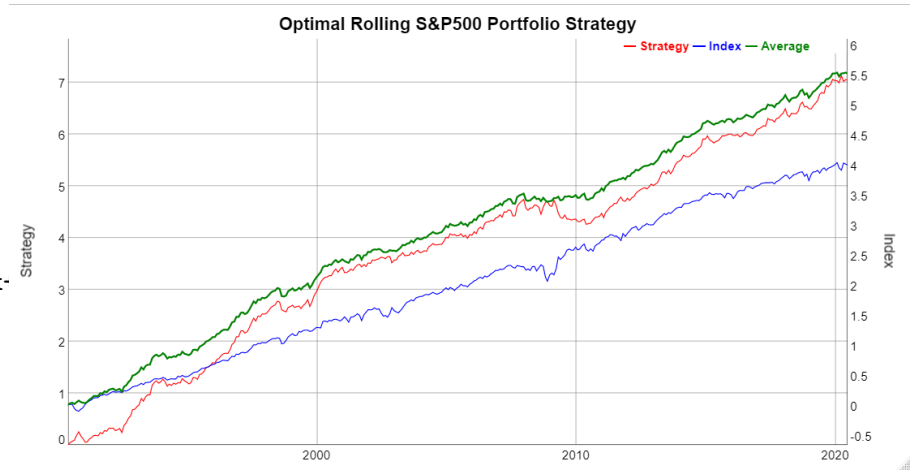



```
> # Plot cumulative strategy returns
> weal_th <- cbind(pnl_s, in_dex, (pnl_s+in_dex)/2)
> weal_th <- cumsum(na.omit(weal_th))
> col_names <- c("Strategy", "Index", "Average")
> colnames(weal_th) <- col_names
> dygraphs::dygraph(weal_th[end_p], main="Optimal Rolling S&P500 Por
+   dyAxis("y", label=col_names[1], independentTicks=TRUE) %>%
+   dyAxis("y2", label=col_names[2], independentTicks=TRUE) %>%
+   dySeries(name=col_names[1], axis="y", col="red", strokeWidth=1)
+   dySeries(name=col_names[2], axis="y2", col="blue", strokeWidth=
+   dySeries(name=col_names[3], axis="y2", col="green", strokeWidth=
```

Determining Look-back Interval Using Backtesting

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

```
> # Perform backtest over look-backs
> look_backs <- seq(from=3, to=24, by=1)
> pnl_s <- lapply(look_backs, function(look_back) {
+   start_p <- c(rep_len(0, look_back-1), end_p[1:(n_rows-look_back-
+   start_p <- (start_p - 1)
+   start_p[start_p < 0] <- 0
+   HighFreq::back_test(typ_e="max_sharpe",
+     ex_cess=returns_100, returns=returns_100,
+     start_p=start_p, end_p=end_p,
+     al_pha=al_pha, max_eigen=max_eigen)
+ })  # end lapply
> pro_file <- sapply(pnl_s, sum)
> plot(x=look_backs, y=pro_file, t="l", main="Strategy PnL as Funct
+   xlab="Look-back Interval", ylab="pnl")
> look_back <- look_backs[which.max(pro_file)]
> pnl_s <- pnl_s[[which.max(pro_file)]]
```




```
> # Plot cumulative strategy returns
> weal_th <- cbind(pnl_s, in_dex, (pnl_s+in_dex)/2)
> weal_th <- cumsum(na.omit(weal_th))
> col_names <- c("Strategy", "Index", "Average")
> colnames(weal_th) <- col_names
> dygraphs::dygraph(weal_th[end_p], main="Optimal Rolling S&P500 Por
+   dyAxis("y", label=col_names[1], independentTicks=TRUE) %>%
+   dyAxis("y2", label=col_names[2], independentTicks=TRUE) %>%
+   dySeries(name=col_names[1], axis="y", col="red", strokeWidth=1)
+   dySeries(name=col_names[2], axis="y2", col="blue", strokeWidth=
+   dySeries(name=col_names[3], axis="y2", col="green", strokeWidth=
```

Homework Assignment

Required

Study all the lecture slides in `FRE7241_Lecture_6.pdf`, and run all the code in `FRE7241_Lecture_6.R`

Recommended

- Read about *estimator shrinkage*:
  Aswani Regression Shrinkage Bias Variance Tradeoff.pdf
  Blei Regression Lasso Shrinkage Bias Variance Tradeoff.pdf
- Read about *optimization methods*:
  Bolker Optimization Methods.pdf
  Yollin Optimization.pdf
  DEoptim Introduction.pdf
  Ardia DEoptim Portfolio Optimization.pdf
  Boudt DEoptim Portfolio Optimization.pdf
  Boudt DEoptim Large Portfolio Optimization.pdf
  Mullen Package DEoptim.pdf
- Read about *momentum*:
  Bouchaud Momentum Mean Reversion Equity Returns.pdf