FRE7241 Algorithmic Portfolio Management

Lecture#6, Spring 2018

Jerzy Pawlowski *jp3900@nyu.edu*

NYU Tandon School of Engineering

March 6, 2018

Rolling Portfolio Optimization Strategy

A *rolling portfolio optimization* strategy consists of rebalancing a portfolio over a vector of 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:

1. Rebalancing frequency (annual, monthly, etc.)
2. Length of look-back interval (sliding or expanding),
3. Scaling of the weights (sum or sum-of-squares),

```
> # sym_bols contains all the symbols in rutils
> sym_bols <- colnames(rutils::env_etf$re_turns)
> sym_bols <- sym_bols[!(sym_bols=="VXX")]
> # Extract columns of rutils::env_etf$re_turns
> re_turns <- rutils::env_etf$re_turns[, sym_bo
> re_turns <- zoo::na.locf(re_turns)
> re_turns <- na.omit(re_turns)
> # Calculate vector of monthly end points and s
> look_back <- 12
> end_points <- rutils::calc_endpoints(re_turns
```

```
> len_gth <- NROW(end_points)
> # sliding window
> start_points <- c(rep_len(1, look_back-1), end
> # expanding window
> start_points <- rep_len(1, NROW(end_points))
> # risk_free is the daily risk-free rate
> risk_free <- 0.03/260
> # Calculate daily excess returns
> ex_cess <- re_turns - risk_free
> # Perform loop over end_points
> portf_rets <- lapply(2:NROW(end_points),
+   function(i) {
+     # subset the ex_cess returns
+     ex_cess <- ex_cess[start_points[i-1]:end_p
+     in_verse <- solve(cov(ex_cess))
+     # calculate the maximum Sharpe ratio portf
+     weight_s <- in_verse %*% colMeans(ex_cess)
+     weight_s <- drop(weight_s/sum(weight_s^2))
+     # subset the re_turns
+     re_turns <- re_turns[(end_points[i-1]+1):e
+     # calculate the out-of-sample portfolio re
+     xts(re_turns %*% weight_s, index(re_turns)
+   }  # end anonymous function
+ )  # end lapply
> portf_rets <- rutils::do_call(rbind, portf_ret
> colnames(portf_rets) <- "portf_rets"
> # Calculate compounded cumulative portfolio re
> portf_rets <- cumsum(portf_rets)
> quantmod::chart_Series(portf_rets,
```

Covariance Matrix Shrinkage Estimator

The estimates of the covariance matrix suffer from statistical errors, and those errors are magnified when the covariance matrix is inverted,

In the *shrinkage* technique the covariance matrix $\mathbb{C}_s$ is estimated as a weighted sum of the sample covariance estimator $\mathbb{C}$ plus a target matrix $\mathbb{T}$:

$$\mathbb{C}_s = (1 - \alpha)\, \mathbb{C} + \alpha\, \mathbb{T}$$

The target matrix $\mathbb{T}$ represents an estimate of the covariance matrix subject to some constraint, such as that all the correlations are equal to each other,

The shrinkage intensity α determines the amount of shrinkage that is applied, with $\alpha = 1$ representing a complete shrinkage towards the target matrix,

The *shrinkage* estimator reduces the estimate variance at the expense of increasing its bias (known as the bias-variance tradeoff),

```
> # create random covariance matrix
> set.seed(1121)
> mat_rix <- matrix(runif(5e2), nc=5)
> cov_mat <- cov(mat_rix)
> cor_mat <- cor(mat_rix)
> std_dev <- sqrt(diag(cov_mat))
> # calculate target matrix
> cor_mean <- mean(cor_mat[upper.tri(cor_mat)])
> tar_get <- matrix(cor_mean, nr=NROW(cov_mat),
> diag(tar_get) <- 1
> tar_get <- t(t(tar_get * std_dev) * std_dev)
> # calculate shrinkage covariance matrix
> al_pha <- 0.5
> cov_shrink <- (1-al_pha)*cov_mat + al_pha*tar_
> # calculate inverse matrix
> in_verse <- solve(cov_shrink)
```

Regularized Inverse of Covariance Matrices

The statistical errors in the covariance matrix are most pronounced in the higher order eigenvalues and eigenvectors,

The *regularization* technique calculates the inverse of the covariance matrix while reducing the effects of statistical errors,

The *regularization* technique involves calculating the inverse of the covariance matrix $\mathbb{C}$ from a limited number of eigenvectors, ignoring the higher order eigenvectors:

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

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

```
> # create random covariance matrix
> set.seed(1121)
> mat_rix <- matrix(runif(5e2), nc=5)
> cov_mat <- cov(mat_rix)
> # perform eigen decomposition
> ei_gen <- eigen(cov_mat)
> eigen_vec <- ei_gen$vectors
> # calculate regularized inverse matrix
> max_eigen <- 2
> in_verse <- eigen_vec[, 1:max_eigen] %*%
+   (t(eigen_vec[, 1:max_eigen]) / ei_gen$values
```

Estimating Rolling Variance Using sapply()

Heteroskedasticity refers to statistical distributions whose variance changes with time,

Empirical *time series* of returns are *heteroskedastic* because their variance changes with time,

The rolling realized variance of a *time series* is a vector given by the estimator:

$$\sigma_i^2 = \frac{1}{k-1} \sum_{j=0}^{k-1} (r_{i-j} - \bar{r}_i)^2$$

$$\bar{r}_i = \frac{1}{k} \sum_{j=0}^{k-1} r_{i-j}$$

Where `k` is the *look-back interval* for performing aggregations over the past,

It's not possible to calculate the rolling variance in `R` using vectorized functions, so it must be calculated using an `apply()` loop,

```
> # VTI percentage returns
> re_turns <- rutils::diff_xts(log(quantmod::Cl(
> # define end points
> end_points <- seq_along(re_turns)
> len_gth <- NROW(end_points)
> look_back <- 51
> # start_points are multi-period lag of end_poi
> start_points <- c(rep_len(1, look_back-1),
+     end_points[1:(len_gth-look_back+1)])
> # define list of look-back intervals for aggre
> look_backs <- lapply(seq_along(end_points),
+   function(in_dex) {
+     start_points[in_dex]:end_points[in_dex]
+ })  # end lapply
> # calculate realized VTI variance in sapply()
> vari_ance <- sapply(look_backs,
+   function(look_back) {
+     ret_s <- re_turns[look_back]
+     sum((ret_s - mean(ret_s))^2)
+ }) / (look_back-1)  # end sapply
> tail(vari_ance)
> class(vari_ance)
> # coerce vari_ance into xts
> vari_ance <- xts(vari_ance, order.by=index(re_
> colnames(vari_ance) <- "VTI.variance"
> head(vari_ance)
```

Estimating Rolling Variance Using Package *roll*

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

- `roll_var()` for *weighted* rolling variance,
- `roll_scale()` for rolling scaling and centering of time series,
- `roll_pcr()` for rolling principal component regressions of time series,

The *roll* functions are about `1,000` times faster than `apply()` loops!

The *roll* functions are extremely fast because they perform calculations in *parallel* in compiled C++ code, using package *Rcpp*,

The *roll* functions accept *xts* time series, and they return *xts*,

```
> # calculate VTI variance using package roll
> library(roll)  # load roll
> vari_ance <-
+   roll::roll_var(re_turns, width=look_back)
> colnames(vari_ance) <- "VTI.variance"
> head(vari_ance)
> sum(is.na(vari_ance))
> vari_ance[1:(look_back-1)] <- 0
> # benchmark calculation of rolling variance
> library(microbenchmark)
> summary(microbenchmark(
+   roll_sapply=sapply(look_backs, function(look
+     ret_s <- re_turns[look_back]
+     sum((ret_s - mean(ret_s))^2)
+   }),
+   ro_ll=roll::roll_var(re_turns, width=look_ba
+   times=10))[, c(1, 4, 5)]
```

Rolling *EWMA* Realized Variance Estimator

Time-varying volatility can be more accurately estimated using an *Exponentially Weighted Moving Average* (*EWMA*) variance estimator,

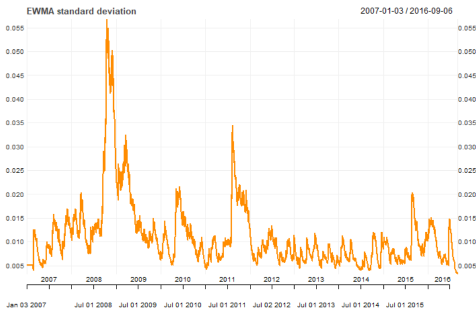
If the *time series* has zero *expected* mean, then the *EWMA realized* variance estimator can be written approxiamtely as:

$$\sigma_i^2 = (1-\lambda) {r_i}^2 + \lambda \sigma_{i-1}^2 = (1-\lambda) \sum_{j=0}^{\infty} \lambda^j {r_{i-j}}^2$$

σ_i^2 is the weighted *realized* variance, equal to the weighted average of the point realized variance for period i and the past *realized* variance,

The parameter λ determines the rate of decay of the *EWMA* weights, with smaller values of λ producing faster decay, giving more weight to recent realized variance, and vice versa,

The function `filter()` calculates the convolution of a vector or time series with a vector of filter coefficients (weights),




```
> # calculate EWMA VTI variance using filter()
> wid_th <- 51
> weight_s <- exp(-0.1*1:wid_th)
> weight_s <- weight_s/sum(weight_s)
> vari_ance <- stats::filter(re_turns^2,
+     filter=weight_s, sides=1)
> vari_ance[1:(wid_th-1)] <- vari_ance[wid_th]
> class(vari_ance)
> vari_ance <- as.numeric(vari_ance)
> x_ts <- xts::xts(sqrt(vari_ance), order.by=in
> # plot EWMA standard deviation
> chart_Series(x_ts,
+   name="EWMA standard deviation")
```

Estimating *EWMA* Variance Using Package *roll*

If the *time series* has non-zero *expected* mean, then the rolling *EWMA* variance is a vector given by the estimator:

$$\sigma_i^2 = \frac{1}{k-1} \sum_{j=0}^{k-1} w_j (r_{i-j} - \bar{r}_i)^2$$

$$\bar{r}_i = \frac{1}{k} \sum_{j=0}^{k-1} w_j r_{i-j}$$

Where w_j is the vector of weights:

$$w_j = \frac{\lambda^j}{\sum_{j=0}^{k-1} \lambda^j}$$

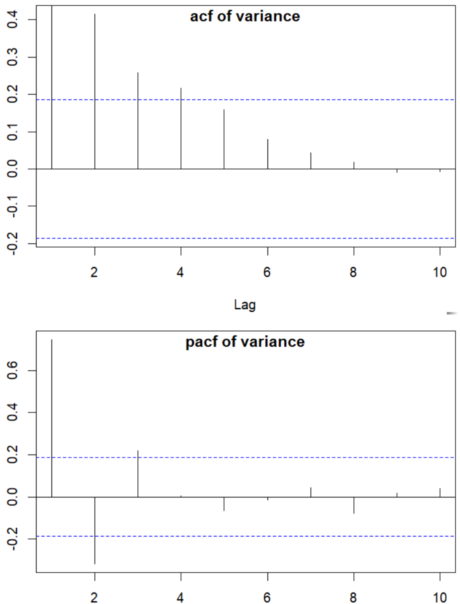
The function `roll_var()` from package *roll* calculates the rolling *EWMA* variance,

```
> # calculate VTI variance using package roll
> library(roll)  # load roll
> vari_ance <- roll::roll_var(re_turns,
+   weights=rev(weight_s), width=wid_th)
> colnames(vari_ance) <- "VTI.variance"
> class(vari_ance)
> head(vari_ance)
> sum(is.na(vari_ance))
> vari_ance[1:(wid_th-1)] <- 0
```

Autocorrelation of Volatility

Variance calculated over non-overlapping intervals has very statistically significant autocorrelations,

```
> # VTI percentage returns
> re_turns <- rutils::diff_xts(log(quantmod::Cl(
> # calculate VTI variance using package roll
> look_back <- 22
> vari_ance <-
+   roll::roll_var(re_turns, width=look_back)
> vari_ance[1:(look_back-1)] <- 0
> colnames(vari_ance) <- "VTI.variance"
> # number of look_backs that fit over re_turns
> n_row <- NROW(re_turns)
> num_agg <- n_row %/% look_back
> end_points <- # define end_points with beginni
+   n_row-look_back*num_agg + (0:num_agg)*look_b
> len_gth <- NROW(end_points)
> # subset vari_ance to end_points
> vari_ance <- vari_ance[end_points]
> # improved autocorrelation function
> acf_plus(coredata(vari_ance), lag=10, main="")
> title(main="acf of variance", line=-1)
> # partial autocorrelation
> pacf(coredata(vari_ance), lag=10, main="", yla
> title(main="pacf of variance", line=-1)
```



GARCH Volatility Model

The *GARCH(1,1)* model is a volatility model defined by two coupled equations:

$$r_i = \mu + \sigma_{i-1}\varepsilon_i$$
$$\sigma_i^2 = \omega + \alpha r_i^2 + \beta\sigma_{i-1}^2$$

Where σ_i^2 is the time-dependent variance, equal to the weighted average of the point *realized* variance ${r_{i-1}}^2$, and the past variance σ_{i-1}^2,

The return process r_i follows a normal distribution with time-dependent variance σ_i^2,

The parameter α is the weight associated with recent realized variance updates, and β is the weight associated with the past variance,

The parameter ω determines the long-term average level of variance, which is given by:

$$\sigma^2 = \frac{\omega}{1 - \alpha - \beta}$$

The sum of α plus β should be less than 1, otherwise the volatility is explosive,

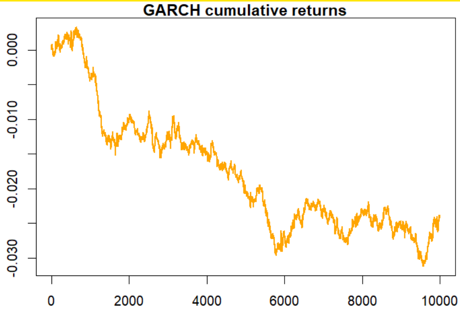
```
> # define GARCH parameters
> om_ega <- 0.01 ; al_pha <- 0.2
> be_ta <- 0.2 ; len_gth <- 1000
> re_turns <- numeric(len_gth)
> vari_ance <- numeric(len_gth)
> vari_ance[1] <- om_ega/(1-al_pha-be_ta)
> re_turns[1] <- rnorm(1, sd=sqrt(vari_ance[1]))
> # simulate GARCH model
> set.seed(1121)  # reset random numbers
> for (i in 2:len_gth) {
+   re_turns[i] <- rnorm(n=1, sd=sqrt(vari_ance[
+   vari_ance[i] <- om_ega + al_pha*re_turns[i]^
+     be_ta*vari_ance[i-1]
+ }  # end for
```

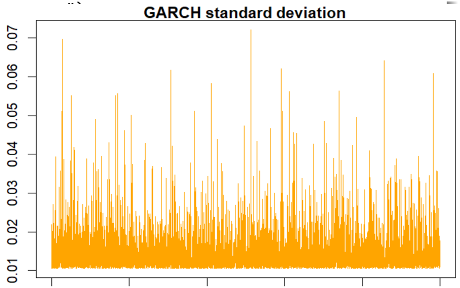
GARCH Volatility Time Series

The *GARCH(1,1)* volatility model exhibits sharp spikes in the volatility, followed by a quick decay of volatility,

But the decay of volatility in the *GARCH* model is faster than what is observed in practice,

```
> # plot GARCH cumulative returns
> plot(cumsum(re_turns/100), t="l",
+   lwd=2, col="orange", xlab="", ylab="",
+   main="GARCH cumulative returns")
> date_s <- seq.Date(from=Sys.Date()-len_gth+1,
+   to=Sys.Date(), length.out=len_gth)
> x_ts <- xts::xts(cumsum(re_turns/100), order.
> dygraphs::dygraph(x_ts, main="GARCH cumulative
> # plot GARCH standard deviation
> plot(sqrt(vari_ance), t="l",
+   col="orange", xlab="", ylab="",
+   main="GARCH standard deviation")
> x_ts <- xts::xts(sqrt(vari_ance), order.by=da
> dygraphs::dygraph(x_ts, main="GARCH standard d
```



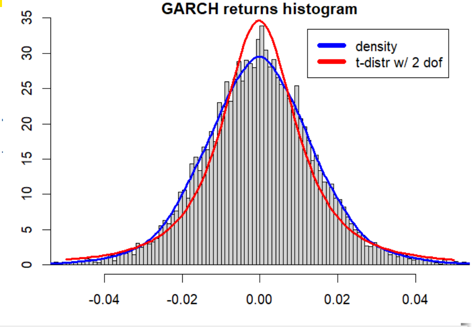


GARCH Model Properties

The parameter α is the weight of the squared realized returns in the variance,

Greater values of α produce a stronger feedback between the realized returns and variance, causing stronger variance spikes and higher kurtosis,

```
> # define GARCH parameters
> om_ega <- 0.0001 ; al_pha <- 0.5
> be_ta <- 0.1 ; len_gth <- 10000
> re_turns <- numeric(len_gth)
> vari_ance <- numeric(len_gth)
> vari_ance[1] <- om_ega/(1-al_pha-be_ta)
> re_turns[1] <- rnorm(1, sd=sqrt(vari_ance[1]))
> # simulate GARCH model
> set.seed(1121)  # reset random numbers
> for (i in 2:len_gth) {
+   re_turns[i] <- rnorm(n=1, sd=sqrt(vari_ance
+   vari_ance[i] <- om_ega + al_pha*re_turns[i]
+     be_ta*vari_ance[i-1]
+ }  # end for
> # calculate kurtosis of GARCH returns
> moments::moment(re_turns, order=4) /
+   moments::moment(re_turns, order=2)^2
> # perform Jarque-Bera test of normality
> tseries::jarque.bera.test(re_turns)
```




```
> # plot histogram of GARCH returns
> histo_gram <- hist(re_turns, col="lightgrey",
+   xlab="returns", breaks=200, xlim=c(-0.05, 0.
+   ylab="frequency", freq=FALSE,
+   main="GARCH returns histogram")
> lines(density(re_turns, adjust=1.5),
+ lwd=3, col="blue")
> optim_fit <- MASS::fitdistr(re_turns,
+   densfun="t", df=2, lower=c(-1, 1e-7))
> lo_cation <- optim_fit$estimate[1]
> sc_ale <- optim_fit$estimate[2]
> curve(expr=dt((x-lo_cation)/sc_ale, df=2)/sc_a
+   type="l", xlab="", ylab="", lwd=3,
+   col="red", add=TRUE)
```

GARCH Model Calibration

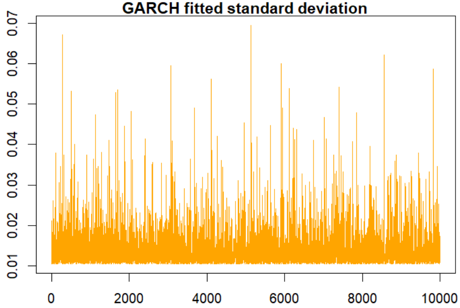
GARCH models can be calibrated on returns using the *maximum-likelihood* method, but it's a complex optimization procedure,

The package *fGarch* contains functions for applying *GARCH* models,

The function `garchFit()` calibrates a *GARCH* model on a time series of returns,

The function `garchFit()` returns an S4 object of class *fGARCH*, with multiple slots containing the *GARCH* model outputs and diagnostic information,

```
> library(fGarch)
> # fit returns into GARCH
> garch_fit <- fGarch::garchFit(data=re_turns)
> # fitted GARCH parameters
> round(garch_fit@fit$coef, 5)
> # actual GARCH parameters
> round(c(mu=mean(re_turns), omega=om_ega,
+   alpha=al_pha, beta=be_ta), 5)
```



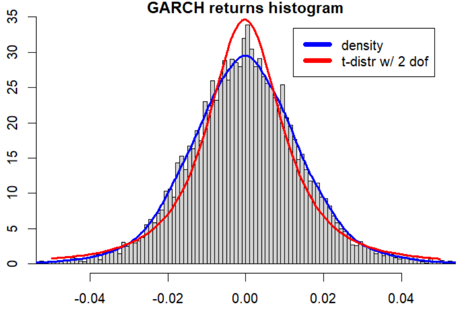

```
> # plot GARCH fitted standard deviation
> plot.zoo(sqrt(garch_fit@fit$series$h), t="l",
+   col="orange", xlab="", ylab="",
+   main="GARCH fitted standard deviation")
```

GARCH Model Simulation

The function garchSpec() from package *fGarch* specifies a *GARCH* model,

The function garchSim() simulates a *GARCH* model,

```
> # specify GARCH model
> garch_spec <- fGarch::garchSpec(
+   model=list(omega=om_ega, alpha=al_pha, beta=
> # simulate GARCH model
> garch_sim <-
+   fGarch::garchSim(spec=garch_spec, n=len_gth)
> re_turns <- as.numeric(garch_sim)
> # calculate kurtosis of GARCH returns
> moments::moment(re_turns, order=4) /
+   moments::moment(re_turns, order=2)^2
> # perform Jarque-Bera test of normality
> tseries::jarque.bera.test(re_turns)
> # plot histogram of GARCH returns
> histo_gram <- hist(re_turns, col="lightgrey",
+   xlab="returns", breaks=200, xlim=c(-0.05, 0
+   ylab="frequency", freq=FALSE,
+   main="GARCH returns histogram")
> lines(density(re_turns, adjust=1.5),
+ lwd=3, col="blue")
```




```
> # fir t-distribution into GARCH returns
> optim_fit <- MASS::fitdistr(re_turns,
+   densfun="t", df=2, lower=c(-1, 1e-7))
> lo_cation <- optim_fit$estimate[1]
> sc_ale <- optim_fit$estimate[2]
> curve(expr=dt((x-lo_cation)/sc_ale, df=2)/sc_a
+   type="l", xlab="", ylab="", lwd=3,
+   col="red", add=TRUE)
> legend("topright", inset=0.05,
+ leg=c("density", "t-distr w/ 2 dof"),
+ lwd=6, lty=c(1, 1),
+ col=c("blue", "red"))
```

Trade and Quote (TAQ) Data

High frequency data is typically formatted as either Trade and Quote (TAQ) data, or Open-High-Low-Close (OHLC) data,

Trade and Quote (TAQ) data contains intraday trades and quotes on exchange-traded stocks and futures,

TAQ data is spaced irregularly in time, with data recorded each time a new trade or quote arrives,

Each row of TAQ data contains both the quote and trade prices, and the corresponding quote size or trade volume: Bid.Price, Bid.Size, Ask.Price, Ask.Size, Trade.Price, and Volume,

```
> # load package HighFreq
> library(HighFreq)
> head(SPY_TAQ)
                    Bid.Price Bid.Size Ask.Price
2014-05-02 00:00:01       188        1       189
2014-05-02 08:00:01       188        1       189
2014-05-02 08:00:02       189        1       189
2014-05-02 08:01:13       188        1       189
2014-05-02 08:01:29       188        1       189
2014-05-02 08:01:52       189        2       189
                    Trade.Price Volume
2014-05-02 00:00:01          NA     NA
2014-05-02 08:00:01         189    100
2014-05-02 08:00:02          NA     NA
2014-05-02 08:01:13          NA     NA
2014-05-02 08:01:29          NA     NA
2014-05-02 08:01:52          NA     NA
```

Open-High-Low-Close (OHLC) Data

Open-High-Low-Close (OHLC) data contains intraday trade prices and trade volumes,

OHLC data is evenly spaced in time, with each row containing the Open, High, Low, and Close prices, and the trade Volume, recorded over the past time interval (called a bar of data),

The Open and Close prices are the first and last trade prices recorded in the time bar,

The High and Low prices are the highest and lowest trade prices recorded in the time bar,

The Volume is the total trading volume recorded in the time bar,

The OHLC data format provides a way of efficiently compressing TAQ data, while preserving information about price levels, volatility (range), and trading volumes,

In addition, evenly spaced OHLC data allows for easier analysis of multiple time series, since the prices for different assets are given at the same moments in time,

```
> # load package HighFreq
> library(HighFreq)
> head(SPY)
                    SPY.Open SPY.High SPY.Low SP
2008-01-02 09:31:00      147      147     147
2008-01-02 09:32:00      147      147     147
2008-01-02 09:33:00      147      147     147
2008-01-02 09:34:00      147      147     147
2008-01-02 09:35:00      147      147     147
2008-01-02 09:36:00      147      147     147
                    SPY.Volume
2008-01-02 09:31:00     591203
2008-01-02 09:32:00     385457
2008-01-02 09:33:00     343700
2008-01-02 09:34:00     863418
2008-01-02 09:35:00     457500
2008-01-02 09:36:00     416708
```

Package *HighFreq* for Managing High Frequency Data

The package *HighFreq* contains functions for managing high frequency time series data, such as:

- converting `TAQ` data to `OHLC` format,
- chaining and joining time series,
- scrubbing bad data,
- managing time zones and alligning time indices,
- aggregating data to lower frequency (periodicity),
- calculating rolling aggregations (VWAP, Hurst exponent, etc.),
- calculating seasonality aggregations,
- estimating volatility, skew, and higher moments,

```
> # install package HighFreq from github
> devtools::install_github(repo="algoquant/HighF
> # load package HighFreq
> library(HighFreq)
> # get documentation for package HighFreq
> # get short description
> packageDescription("HighFreq")
> # load help page
> help(package="HighFreq")
> # list all datasets in "HighFreq"
> data(package="HighFreq")
> # list all objects in "HighFreq"
> ls("package:HighFreq")
> # remove HighFreq from search path
> detach("package:HighFreq")
```

Datasets in Package *HighFreq*

The package *HighFreq* contains several high frequency time series, in *xts* format, stored in a file called `hf_data.RData`:

- a time series called `SPY_TAQ`, containing a single day of `TAQ` data for the *SPY* ETF,
- three time series called `SPY`, `TLT`, and `VXX`, containing intraday 1-minute `OHLC` data for the *SPY*, *TLT*, and *VXX* ETFs,

Even after the *HighFreq* package is loaded, its datasets aren't loaded into the workspace, so they aren't listed in the workspace,

That's because the datasets in package *HighFreq* are set up for *lazy loading*, which means they can be called as if they were loaded, even though they're not loaded into the workspace,

The datasets in package *HighFreq* can be loaded into the workspace using the function `data()`,

The data is set up for *lazy loading*, so it doesn't require calling `data(hf_data)` to load it into the workspace before calling it,

```
> # load package HighFreq
> library(HighFreq)
> # you can see SPY when listing objects in High
> ls("package:HighFreq")
> # you can see SPY when listing datasets in Hig
> data(package="HighFreq")
> # but the SPY dataset isn't listed in the work
> ls()
> # HighFreq datasets are lazy loaded and availa
> head(SPY)
> # load all the datasets in package HighFreq
> data(hf_data)
> # HighFreq datasets are now loaded and in the
> head(SPY)
```

Estimating Volatility of Intraday Time Series

The *close-to-close* estimator depends on `close` prices specified over the aggregation intervals:

$$\hat{\sigma}^2 = \frac{1}{n-1} \sum_{i=1}^{n} (\log(\frac{C_i}{C_{i-1}}) - \bar{r})^2$$
$$\bar{r} = \frac{1}{n} \sum_{i=1}^{n} \log(\frac{C_i}{C_{i-1}})$$

Volatility estimates for intraday time series depend both on the units of returns (per second, minute, day, etc.), and on the aggregation interval (secondly, minutely, daily, etc.),

A minutely time interval is equal to 60 seconds, a daily time interval is equal to `86,400=24*60*60` seconds, etc.),

For example, it's possible to measure returns in minutely intervals in units per second,

The estimated volatility is directly proportional to the measurement units,

For example, the volatility estimated from per minute returns is 60 times the volatility estimated from per second returns,

```
> library(HighFreq)  # load HighFreq
> # minutely SPY returns (unit per minute) singl
> re_turns <- rutils::diff_xts(log(SPY["2012-02-
> # minutely SPY volatility (unit per minute)
> sd(re_turns)
[1] 0.000223
> # minutely SPY returns (unit per second)
> re_turns <- rutils::diff_xts(log(SPY["2012-02-
+   c(1, diff(.index(SPY["2012-02-13"])))
> # minutely SPY volatility scaled to unit per m
> 60*sd(re_turns)
[1] 0.000223
> # minutely SPY returns multiple days no overni
> re_turns <- rutils::diff_xts(log(SPY[, 4]))
> # minutely SPY volatility (unit per minute)
> sd(re_turns)
[1] 0.000726
> # minutely SPY returns (unit per second)
> re_turns <- rutils::diff_xts(log(SPY[, 4])) /
+   c(1, diff(.index(SPY)))
> # minutely SPY volatility scaled to unit per m
> 60*sd(re_turns)
[1] 0.000594
> table(c(1, diff(.index(SPY))))

     1      60     120     180     240     300     360
     1  623570     149      50      17      13      11
   480     540     600     840    1080    1260   63000
```

Volatility as Function of Aggregation Interval

Return volatility depends on the length of the aggregation time interval approximately as the *square root* of the interval:

$$\hat{\sigma} \propto \Delta t^{H/2}$$

Where Δt is the length of the aggregation interval, and H is the *Hurst* exponent,

If prices follow geometric Brownian motion then the volatility is exactly proportional to the *square root* of the interval length (H=1),

If prices are mean-reverting then the volatility grows slower than the *square root* of the interval length (H<1),

If prices are trending then the volatility grows faster than the *square root* of the interval length (H>1),

The length of the daily time interval is often approximated to be equal to 390=6.5*60 minutes, since the trading session is equal to 6.5 hours, and daily volatility is dominated by that of the trading session,

```
> # daily OHLC SPY prices
> SPY_daily <-
+   rutils::to_period(oh_lc=SPY, period="days")
> # daily SPY returns and volatility
> sd(rutils::diff_xts(log(SPY_daily[, 4])))
[1] 0.0148
> # minutely SPY returns (unit per minute)
> re_turns <- rutils::diff_xts(log(SPY[, 4]))
> # minutely SPY volatility scaled to daily inte
> sqrt(6.5*60)*sd(re_turns)
[1] 0.0143
>
> # minutely SPY returns (unit per second)
> re_turns <- rutils::diff_xts(log(SPY[, 4])) /
+   c(1, diff(.index(SPY)))
> # minutely SPY volatility scaled to daily aggr
> 60*sqrt(6.5*60)*sd(re_turns)
[1] 0.0117
>
> # daily SPY volatility
> # including extra time over weekends and holid
> 24*60*60*sd(rutils::diff_xts(log(SPY_daily[, 4
+     c(1, diff(.index(SPY_daily))))
[1] 0.0128
> table(c(1, diff(.index(SPY_daily))))

     1  85500  85980  86340  86400  86460 172740
     1      1      1      5   1242      2      5
```

Range Volatility Estimators of OHLC Time Series

- *Range* volatility estimators utilize the high and low prices, and therefore have lower standard error than the standard *close-to-close* estimator,
- The *Garman-Klass* estimator uses the *low-to-high* price range, but it underestimates volatility because it doesn't account for *close-to-open* price jumps:

$$\hat{\sigma}^2 = \frac{1}{n} \sum_{i=1}^{n} (0.5 \log(\frac{H_i}{L_i})^2 - (2\log 2 - 1)\log(\frac{C_i}{O_i})^2)$$

- The *Yang-Zhang* estimator is the most efficient (has the lowest standard error) among unbiased estimators, and also accounts for *close-to-open* price jumps:

$$\hat{\sigma}^2 = \frac{1}{n-1} \sum_{i=1}^{n} (\log(\frac{O_i}{C_{i-1}}) - \bar{r}_{co})^2 +$$
$$0.134(\log(\frac{C_i}{O_i}) - \bar{r}_{oc})^2 +$$
$$\frac{0.866}{n} \sum_{i=1}^{n} (\log(\frac{H_i}{O_i})\log(\frac{H_i}{C_i}) + \log(\frac{L_i}{O_i})\log(\frac{L_i}{C_i}))$$

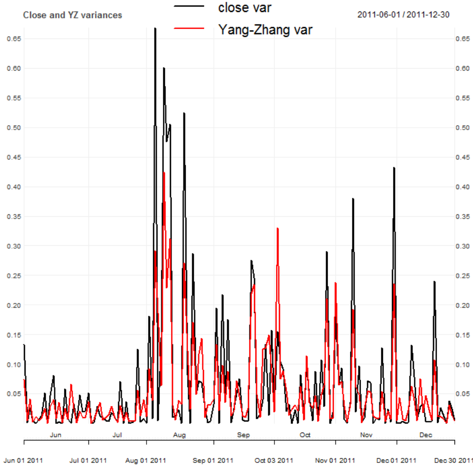
```
> # daily SPY volatility from minutely prices us
> library(TTR)
> sqrt((6.5*60)*mean(na.omit(
+   TTR::volatility(SPY, N=1,
+           calc="yang.zhang"))^2))
> # SPY volatility using package HighFreq
> 60*sqrt((6.5*60)*agg_regate(oh_lc=SPY,
+     weight_ed=FALSE, mo_ment="run_variance",
+     calc_method="yang_zhang"))
```

- Theoretically, the *Yang-Zhang* (*YZ*) and *Garman-Klass-Yang-Zhang* (*GKYZ*) range variance estimators are unbiased and have up to seven times smaller standard errors than the standard close-to-close estimator,
- But in practice, prices are not observed continuously, so the price range is underestimated, and so is the variance when using the *YZ* and *GKYZ* range estimators,
- Therefore in practice the *YZ* and *GKYZ* range estimators underestimate volatility,
- In addition, their standard errors are reduced less than by the theoretical amount, for the same reason,

Comparing *Range* Volatility Estimators

The *Range* volatility estimators follow the standard *Close-to-Close* estimator, except in intervals of high intra-period volatility,

```
> library(HighFreq)  # load HighFreq
> # calculate variance
> var_close <-
+   HighFreq::run_variance(oh_lc=env_etf$VTI,
+                           calc_method="close")
> var_yang_zhang <-
+   HighFreq::run_variance(oh_lc=env_etf$VTI)
> vari_ance <-
+   252*(24*60*60)^2*cbind(var_close, var_yang_z
> colnames(vari_ance) <-
+   c("close var", "Yang-Zhang var")
> # plot
> plot_theme <- chart_theme()
> plot_theme$col$line.col <- c("black", "red")
> x11()
> chart_Series(vari_ance["2011-06/2011-12"],
+   theme=plot_theme, name="Close and YZ varianc
> legend("top", legend=colnames(vari_ance),
+  bg="white", lty=c(1, 1), lwd=c(6, 6),
+  col=plot_theme$col$line.col, bty="n")
```



Dynamic Documents Using *R markdown*

markdown is a simple markup language designed for creating documents in different formats, including *pdf* and *HTML*,

R Markdown is a modified version of *markdown*, which allows creating documents containing *math formulas* and R code embedded in them,

An R document is an *R Markdown* file (with extension .Rmd) containing:

- A *YAML* header,
- Text in *R Markdown* code format,
- Math formulas (equations), delimited using either single "\$" symbols (for inline formulas), or double "\$\$" symbols (for display formulas),
- R code chunks, delimited using either single "`" backtick symbols (for inline code), or triple "```" backtick symbols (for display code),

The packages *rmarkdown* and *knitr* compile R documents into either *pdf*, *HTML*, or *MS Word* documents,

````
---
title: "My First R Markdown Document"
author: Jerzy Pawlowski
date: '`r format(Sys.time(), "%m/%d/%Y")`'
output: html_document
---

```{r setup, include=FALSE}
knitr::opts_chunk$set(echo = TRUE)

# install package quantmod if it can't be loaded success
if (!require("quantmod"))
  install.packages("quantmod")
```

### R Markdown
This is an *R Markdown* document. Markdown is a simple f

One of the advantages of writing documents *R Markdown*

You can read more about publishing documents using *R* h
https://algoquant.github.io/r,/markdown/2016/07/02/Publi

You can read more about using *R* to create *HTML* docum
https://algoquant.github.io/2016/07/05/Interactive-Plots

Clicking the **Knit** button in *RStudio*, compiles the

Example of an *R* code chunk:
```{r cars}
summary(cars)
```

### Plots in *R Markdown* documents

Plots can also be embeded, for example:
```{r pressure, echo=FALSE}
plot(pressure)
````
````

Interactive Charts Using Package *shiny*

The package *shiny* creates interactive plots that display the outputs of live models running in R,

The function `inputPanel()` creates a panel for user input of model parameters,

The function `renderPlot()` renders a plot from the outputs of a live model running in R,

To create a shiny plot, you can first create an .Rmd file, embed the *shiny* code in an R chunk, and then compile the .Rmd file into an *HTML* document, using the *knitr* package,


```
> # R startup chunk
> # ```{r setup, include=FALSE}
> library(shiny)
> library(quantmod)
> inter_val <- 31
> cl_ose <- quantmod::Cl(rutils::env_etf$VTI)
> plot_theme <- chart_theme()
> plot_theme$col$line.col <- c("orange", "blue")
> # ```
> ### end R startup chunk
> inputPanel(
+   sliderInput("lamb_da", label="lambda:",
+     min=0.01, max=0.2, value=0.1, step=0.01)
+ )  # end inputPanel
```

```
> renderPlot({
+   # calculate EWMA prices
+   lamb_da <- input$lamb_da
+   weight_s <- exp(-lamb_da*1:inter_val)
+   weight_s <- weight_s/sum(weight_s)
+   ew_ma <- filter(cl_ose, filter=weight_s, sid
+   ew_ma[1:(inter_val-1)] <- ew_ma[inter_val]
+   ew_ma <- xts(cbind(cl_ose, ew_ma), order.by=
+   colnames(ew_ma) <- c("VTI", "VTI EWMA")
+   # plot EWMA prices
+   ch_ob <- chart_Series(ew_ma, theme=plot_them
+   plot(ch_ob)
+   legend("top", legend=colnames(ew_ma),
+    inset=0.1, bg="white", lty=c(1, 1), lwd=c(2
+    col=plot_theme$col$line.col, bty="n")
```

Homework Assignment

Required

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

Recommended

- Read chapters 1-3: Venables. *An Introduction to R*. URL: http://cran.r-project.org/doc/manuals/r-release/R-intro.pdf
- Read chapters 1, 2, 11: *The Art of R Programming*
- Read: Google. *Style Guide for R*. URL: https://google.github.io/styleguide/Rguide.xml