FRE7241 Algorithmic Portfolio Management

Lecture#3, Spring 2018

Jerzy Pawlowski *jp3900@nyu.edu*

NYU Tandon School of Engineering

February 6, 2018

NYU | TANDON SCHOOL OF ENGINEERING

Momentum Strategy for an *ETF* Portfolio

A *momentum* strategy can be *back-tested* for a portfolio of *ETFs* or stocks over a series of *end points* as follows:

1. Trade a single unit (dollar) of capital,
2. Allocate capital to the assets in proportion to their *relative* past performance,
3. Calculate the portfolio weights as the number of units (shares) of each asset,
4. At each *end point* repeat the above and rebalance the asset allocations,
5. Calculate the future out-of-sample portfolio returns in each period (without reinvestment),
6. Calculate the transaction costs (as the bid-offer spread times the asset prices, times the change in weights), and subtract them from the returns,

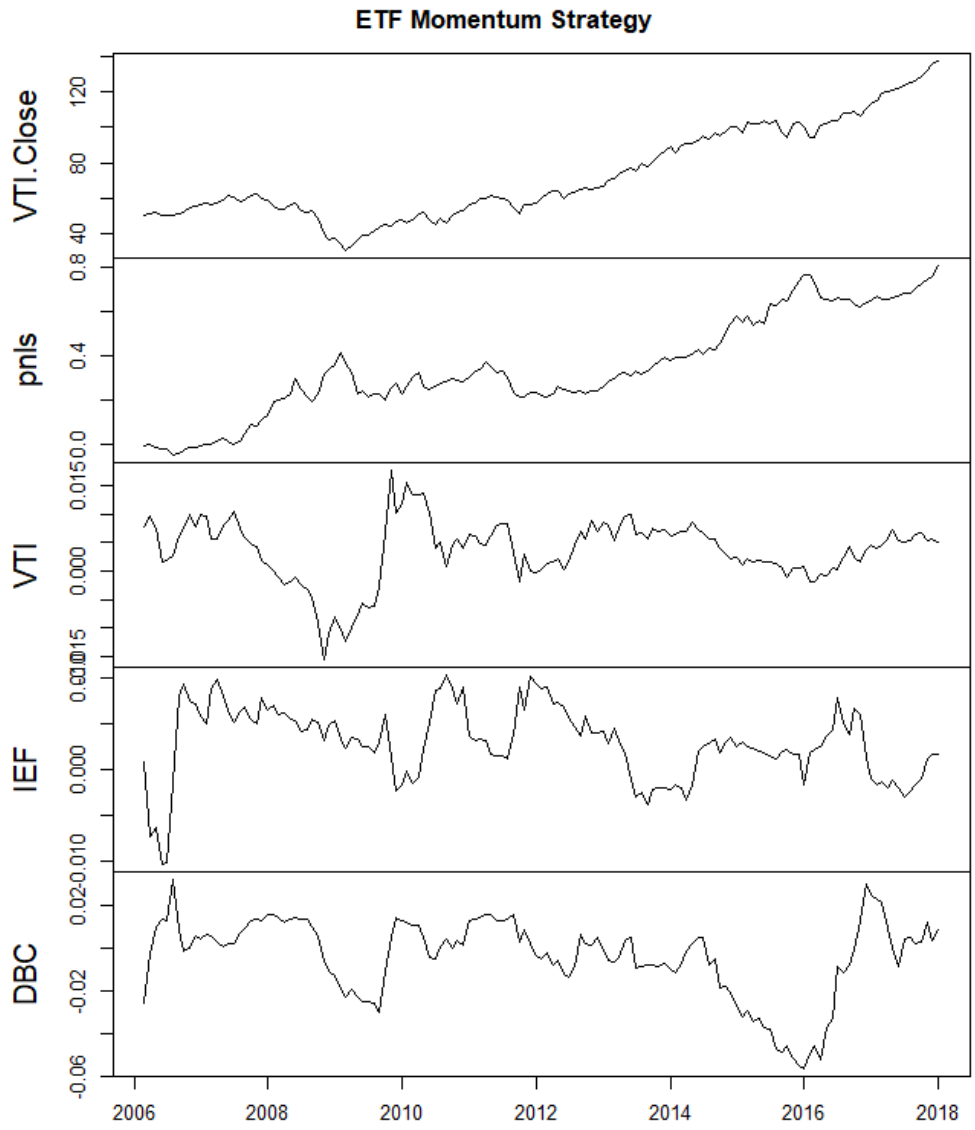
The above points #1, #2 and #3 represent the *momentum* strategy, while points #4, #5 and #6 represent the *back-test* procedures,

```
> # Calculate ETF prices and simple returns
> sym_bols <- c("VTI", "IEF", "DBC")
> price_s <- rutils::env_etf$price_s[, sym_bols]
> price_s <- zoo::na.locf(price_s)
> price_s <- na.omit(price_s)
> re_turns <- rutils::diff_it(price_s)
> # Define look-back and look-forward intervals
> end_points <- rutils::calc_endpoints(re_turns,
+   inter_val="months")
> n_col <- NCOL(re_turns)
> len_gth <- NROW(end_points)
> look_back <- 12
> start_points <- c(rep_len(1, look_back-1),
+   end_points[1:(len_gth-look_back+1)])
> fwd_points <- end_points[c(2:len_gth, len_gth)
> # Perform loop over end-points and calculate a
> agg_fun <-
+   function(re_turns) sum(re_turns)/sd(re_turns
> agg_s <- sapply(1:(len_gth-1), function(it_er)
+   c(back_aggs=sapply(re_turns[start_points[it_
+     fwd_rets=sapply(re_turns[(end_points[it_er
+ })  # end sapply
> agg_s <- t(agg_s)
> # Select look-back and look-forward aggregatio
> back_aggs <- agg_s[, 1:n_col]
> fwd_rets <- agg_s[, n_col+1:n_col]
```

Backtesting the *Momentum* Strategy for an *ETF* Portfolio

The hypothetical *momentum* strategy returns can be calculated by multiplying the forecast portfolio weights times the forward (future) out-of-sample returns,

```
> # Calculate portfolio weights equal to number
> end_prices <- price_s[end_points[-len_gth]]
> weight_s <-
+   back_aggs/rowSums(abs(back_aggs))/end_prices
> weight_s[is.na(weight_s)] <- 0
> colnames(weight_s) <- colnames(re_turns)
> # Calculate profits and losses
> pnl_s <- rowSums(weight_s*fwd_rets)
> pnl_s <- xts(pnl_s, index(end_prices))
> colnames(pnl_s) <- "pnls"
> # Calculate transaction costs
> bid_offer <- 0.001
> cost_s <-
+   0.5*bid_offer*end_prices*abs(rutils::diff_it
> cost_s <- rowSums(cost_s)
> pnl_s <- (pnl_s - cost_s)
> pnl_s <- cumsum(pnl_s)
> # plot momentum strategy with VTI
> cl_ose <- Cl(rutils::env_etf$VTI[index(end_pri
> zoo::plot.zoo(cbind(cl_ose, pnl_s, weight_s),
+   oma = c(3, 1, 3, 0), mar = c(0, 4, 0, 1), nc
+   xlab=NULL, main="ETF Momentum Strategy")
```



Backtesting Functional for *Momentum* Strategy

```
> back_test_ep <- function(re_turns, price_s, agg_fun=sum,
+     look_back=12, re_balance="months", bid_offer=0.001,
+     end_points=rutils::calc_endpoints(re_turns, inter_val=re_balance),
+     with_weights=FALSE, ...) {
+   stopifnot("package:quantmod" %in% search() || require("quantmod", quietly=TRUE))
+   # Define look-back and look-forward intervals
+   n_col <- NCOL(re_turns)
+   len_gth <- NROW(end_points)
+   start_points <- c(rep_len(1, look_back-1), end_points[1:(len_gth-look_back+1)])
+   fwd_points <- end_points[c(2:len_gth, len_gth)]
+   # Perform loop over end-points and calculate aggregations
+   agg_s <- sapply(1:(len_gth-1), function(it_er) {
+     c(back_aggs=sapply(re_turns[start_points[it_er]:end_points[it_er]], agg_fun, ...),  # end
+     fwd_rets=sapply(re_turns[(end_points[it_er]+1):fwd_points[it_er]], sum))  # end sapply
+   })  # end sapply
+   agg_s <- t(agg_s)
+   # Select look-back and look-forward aggregations
+   back_aggs <- agg_s[, 1:n_col]
+   fwd_rets <- agg_s[, n_col+1:n_col]
+   # Calculate portfolio weights equal to number of shares
+   end_prices <- price_s[end_points[-len_gth]]
+   weight_s <- back_aggs/rowSums(abs(back_aggs))/end_prices
+   weight_s[is.na(weight_s)] <- 0
+   colnames(weight_s) <- colnames(re_turns)
+   # Calculate profits and losses
+   pnl_s <- rowSums(weight_s*fwd_rets)
+   pnl_s <- xts(pnl_s, index(end_prices))
+   colnames(pnl_s) <- "pnls"
```

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,

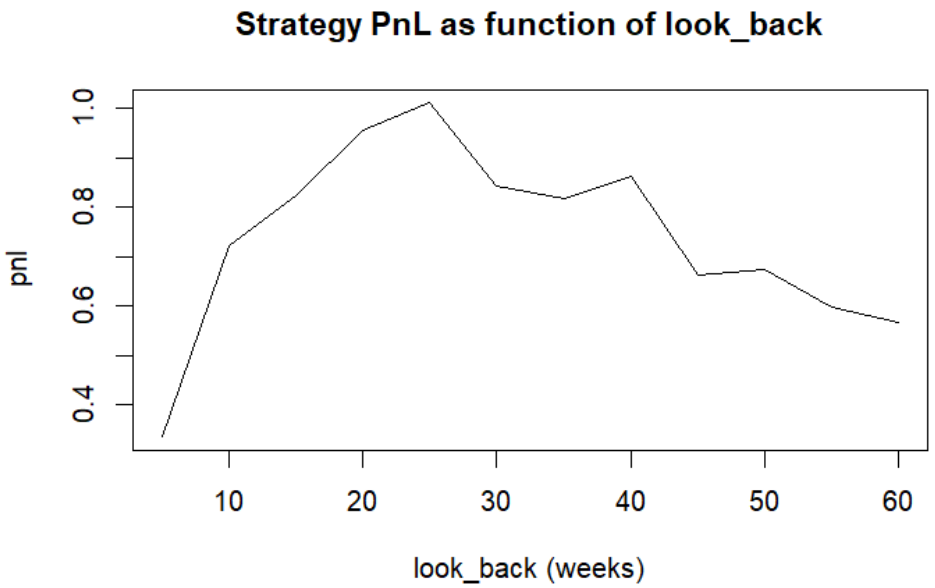
Performing a *back-test* allows finding the optimal *momentum* (trading) strategy parameters, such as the *look-back interval*,

But using a different rebalancing frequency in the *back-test* can produce different values for the optimal trading strategy parameters,

So *back-testing* just redefines the problem of finding (tuning) the optimal trading strategy parameters, into the problem of finding the optimal *back-test* (meta-model) parameters,

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

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



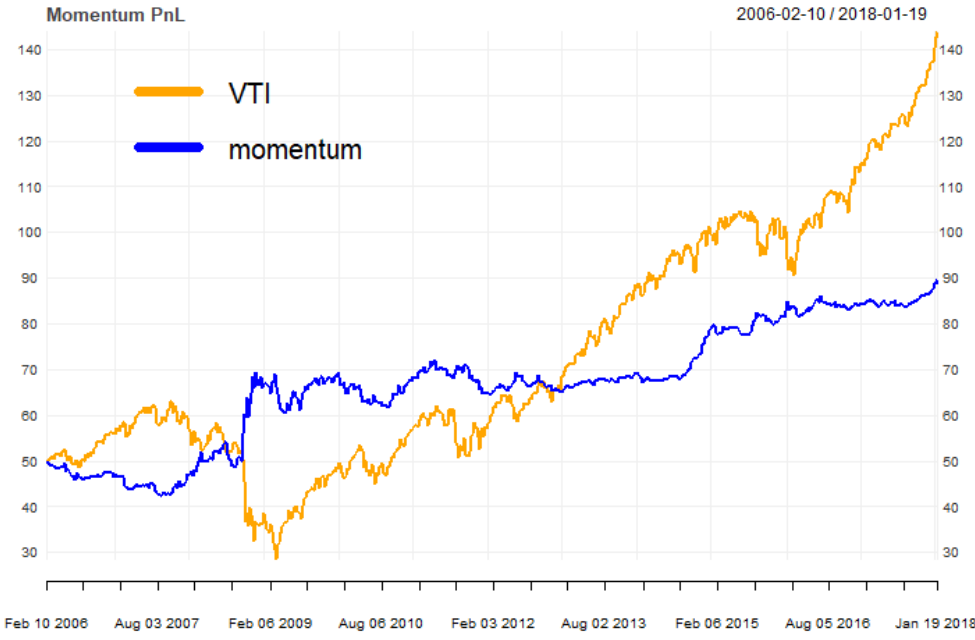

```
> source("C:/Develop/R/lecture_slides/scripts/ba
> look_backs <- seq(5, 60, by=5)
> agg_fun <- function(re_turns) sum(re_turns)/sd
> pro_files <- sapply(look_backs, function(x) {
+   last(back_test_ep(re_turns=re_turns, price_s
+     re_balance="weeks", look_back=x, agg_fun=a
+ })  # end sapply
> plot(x=look_backs, y=pro_files, t="l",
+   main="Strategy PnL as function of look_back"
+   xlab="look_back (weeks)", ylab="pnl")
```

Momentum Strategy Performance

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

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

```
> look_back <- look_backs[which.max(pro_files)]
> pnl_s <- back_test_ep(re_turns=re_turns, price
+   re_balance="weeks", look_back=look_back, agg
+   with_weights=TRUE)
> cl_ose <- Cl(rutils::env_etf$VTI[index(pnl_s)]
> # bind model returns with VTI
> da_ta <- as.numeric(cl_ose[1, ])
> da_ta <- cbind(cl_ose, da_ta*pnl_s[, 1]+da_ta
> colnames(da_ta) <- c("VTI", "momentum")
```



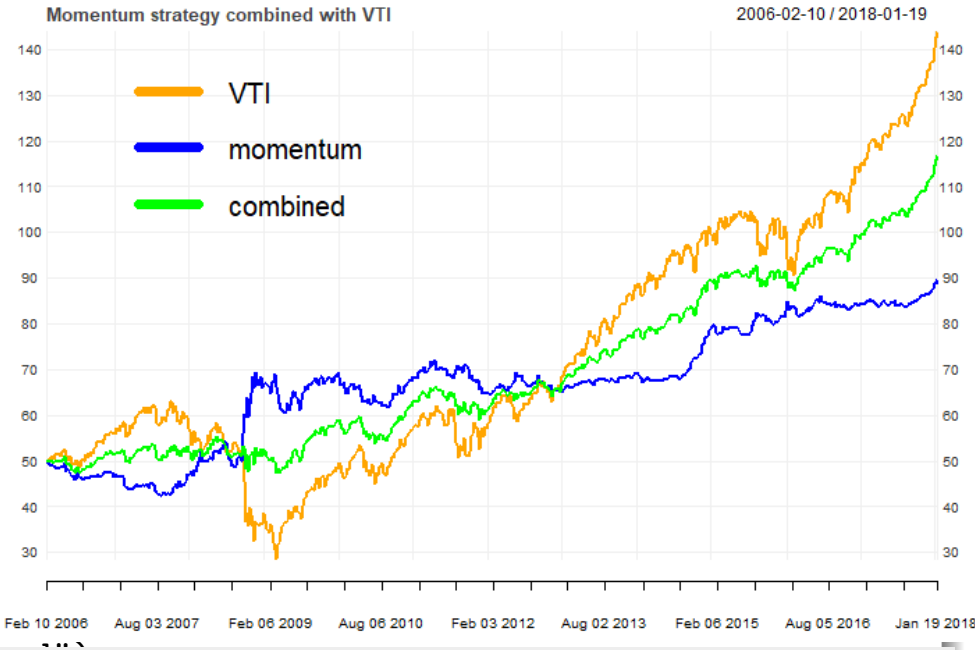

```
> # plot momentum strategy with VTI
> plot_theme <- chart_theme()
> plot_theme$col$line.col <- c("orange", "blue")
> chart_Series(da_ta, theme=plot_theme, lwd=2,
+         name="Momentum PnL")
> legend("topleft", legend=colnames(da_ta),
+   inset=0.1, bg="white", lty=1, lwd=6,
+   col=plot_theme$col$line.col, bty="n")
```

Combining the *Momentum* and Static Strategies

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,



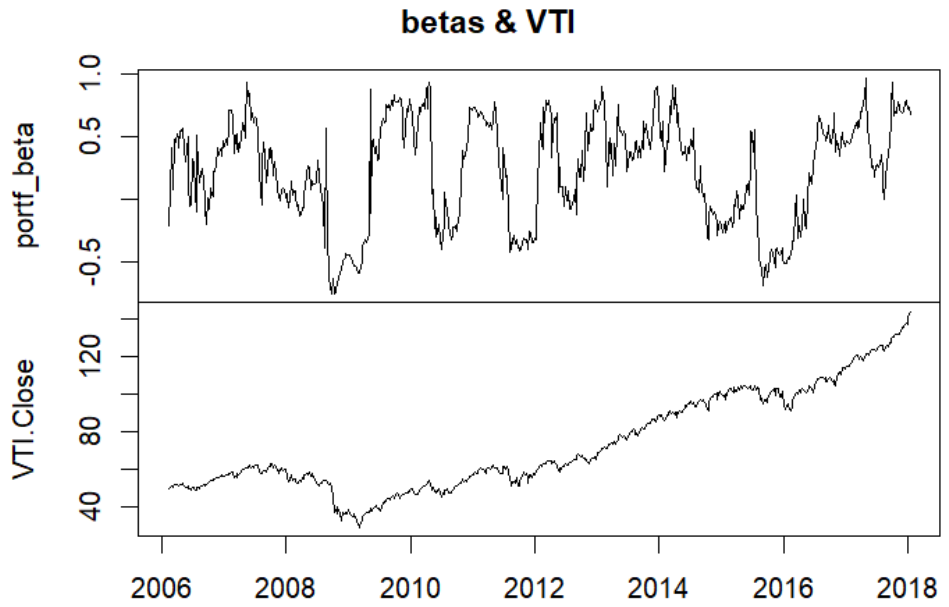

```
> # combine momentum strategy with static
> da_ta <- cbind(da_ta, 0.5* (da_ta[, "VTI"] + d
> colnames(da_ta) <- c("VTI", "momentum", "combi
> # calculate strategy annualized Sharpe ratios
> sapply(da_ta, function(cumu_lative) {
+   x_ts <- na.omit(diff(log(cumu_lative)))
+   sqrt(52)*sum(x_ts)/sd(x_ts)/NROW(x_ts)
+ })  # end sapply
> # calculate strategy correlations
> cor(na.omit(diff(log(da_ta))))
```

```
> # plot momentum strategy combined with VTI
> plot_theme <- chart_theme()
> plot_theme$col$line.col <- c("orange", "blue",
> chart_Series(da_ta, theme=plot_theme,
+        name="Momentum strategy combined with V
> legend("topleft", legend=colnames(da_ta),
+   inset=0.1, bg="white", lty=1, lwd=6,
+   col=plot_theme$col$line.col, bty="n")
```

Momentum Strategy Market Beta

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

```
> # calculate betas
> beta_s <- c(1, rutils::env_etf$capm_stats[
+   match(sym_bols[-1],
+   rownames(rutils::env_etf$capm_stats)),
+   "Beta"])
> names(beta_s)[1] <- sym_bols[1]
> # weights times betas
> weight_s <- price_s[index(pnl_s)]*pnl_s[, -1]
> beta_s <- weight_s %*% beta_s
> beta_s <- xts(beta_s, order.by=index(weight_s)
> colnames(beta_s) <- "portf_beta"
> zoo::plot.zoo(cbind(beta_s, cl_ose),
+   oma = c(3, 1, 3, 0), mar = c(0, 4, 0, 1),
+   main="betas & VTI", xlab="")
```



Momentum Strategy Market Timing

The *market timing* ability of the *momentum* strategy can be measured by performing *linear regression* tests on its returns,

The *market timing* tests are generalizations of the *Capital Asset Pricing Model*,

The *Merton-Henriksson* test measures the ability to forecast the direction of market returns:

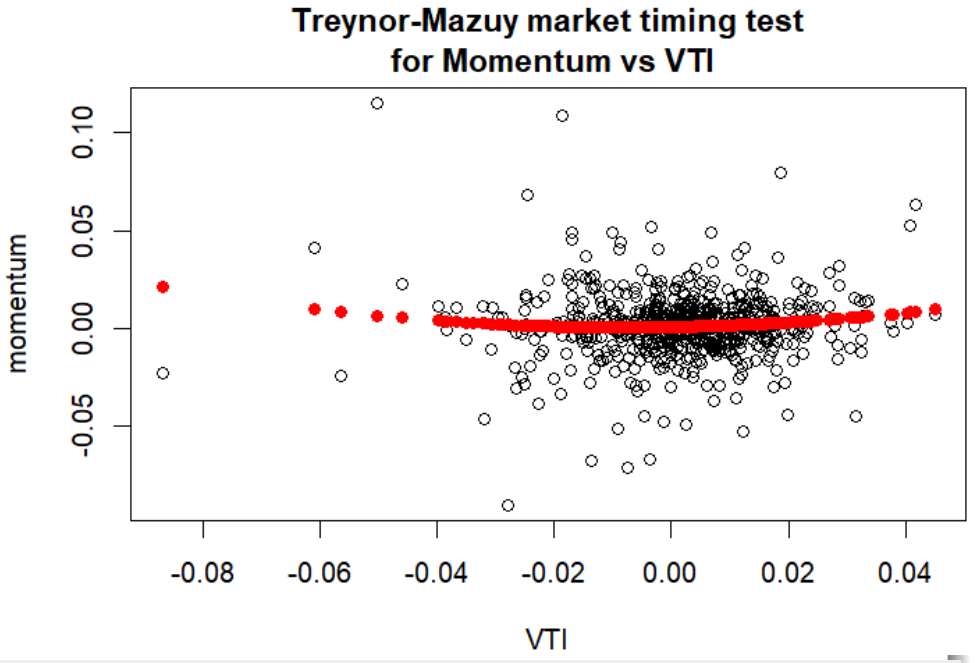
$$R - R_f = \alpha + \beta(R_m - R_f) + \gamma \max(0, R_m - R_f) + \varepsilon$$

The *Treynor-Mazuy* test measures the ability to forecast both the direction and magnitude of future market returns:

$$R - R_f = \alpha + \beta(R_m - R_f) + \gamma(R_m - R_f)^2 + \varepsilon$$

Where R are the model returns, R_m are the market returns, and R_f are the risk-free returns,

```
> momentum_rets <- as.numeric(rutils::diff_it(pr
> vti_rets <- as.numeric(rutils::diff_it(cl_ose
> # Merton-Henriksson test
> vti_ <- cbind(vti_rets, vti_rets+abs(vti_rets
> colnames(vti_) <- c("rets", "sign")
> reg_model <- lm(momentum_rets ~ vti_)
> summary(reg_model)
```



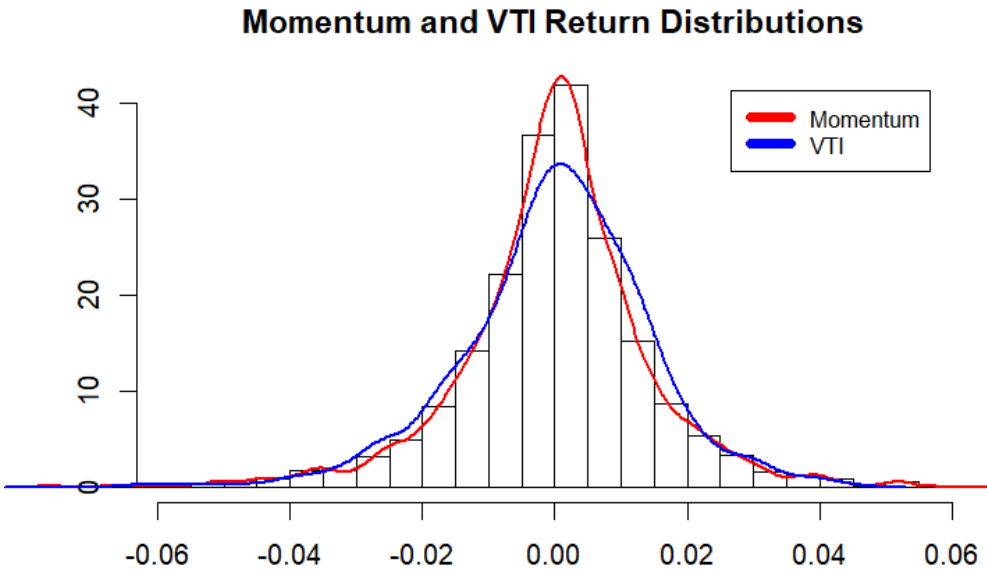

```
> # Treynor-Mazuy test
> vti_ <- cbind(vti_rets, vti_rets^2)
> colnames(vti_) <- c("rets", "squared")
> reg_model <- lm(momentum_rets ~ vti_)
> summary(reg_model)
> # plot scatterplot
> plot(x=vti_rets, y=momentum_rets,
+      xlab="VTI", ylab="momentum")
> title(main="Treynor-Mazuy market timing test\n
> # plot fitted (predicted) response values
> points(x=vti_rets, y=reg_model$fitted.values,
+  pch=16, col="red")
```

Momentum Strategy Returns Skew

The distribution of returns of the *momentum* strategy has about one half of the negative skew compared to *VTI*,

The *momentum* strategy indicates the existence of a market anomaly (outsized profits), because it has a smaller negative skew than the market, while having comparable returns to the market,

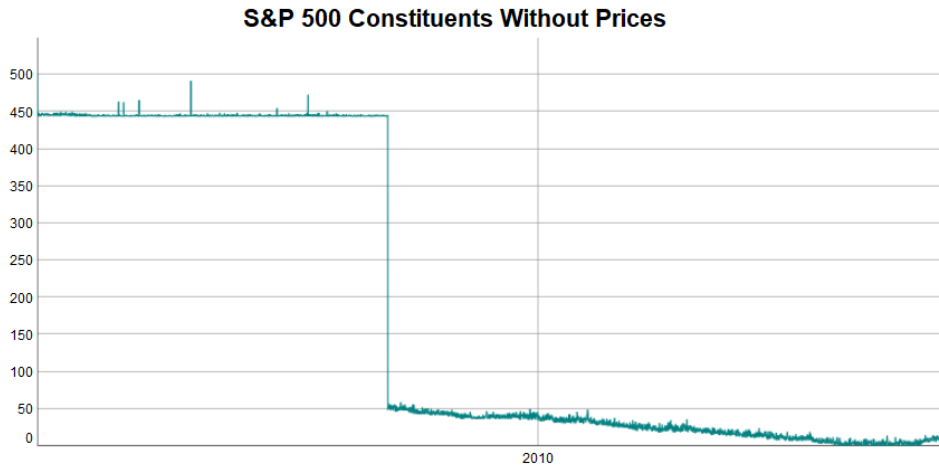
```
> # Normalize the returns
> momentum_rets <-
+   (momentum_rets-mean(momentum_rets))
> momentum_rets <-
+   sd(vti_rets)*momentum_rets/sd(momentum_rets)
> vti_rets <- (vti_rets-mean(vti_rets))
> # calculate ratios of moments
> sapply(2:4, FUN=moments::moment, x=vti_rets)/
+   sapply(2:4, FUN=moments::moment, x=momentum
```




```
> # plot histogram
> x_lim <- 4*sd(momentum_rets)
> hist(momentum_rets, breaks=30,
+   main="Momentum and VTI Return Distributions"
+   xlim=c(-x_lim, x_lim),
+   xlab="", ylab="", freq=FALSE)
> # draw kernel density of histogram
> lines(density(momentum_rets), col='red', lwd=2
> lines(density(vti_rets), col='blue', lwd=2)
> # add legend
> legend("topright", inset=0.05, cex=0.8, title=
+  leg=c("Momentum", "VTI"),
+  lwd=6, bg="white", col=c("red", "blue"))
```

S&P500 Stock Index Constituent Prices

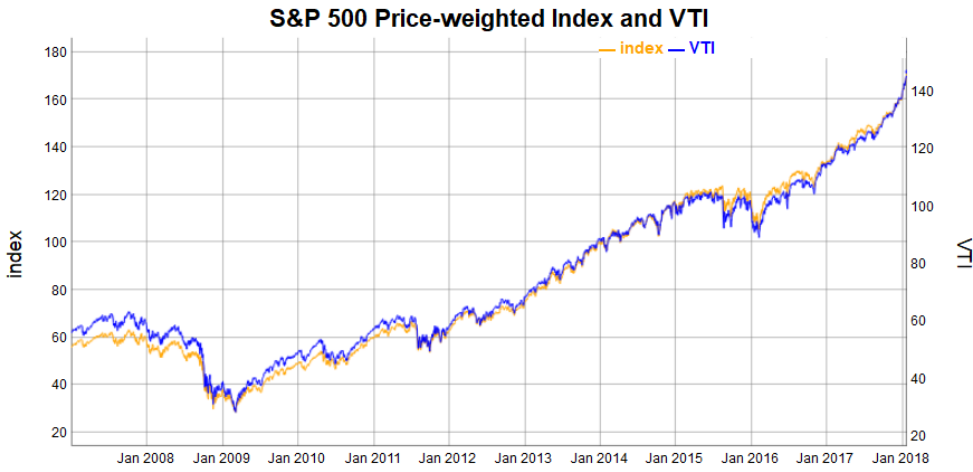
The *S&P500* stock index constituent data is of poor quality before 2007, so we'll select only the data after 2007,




```
> # load S&P500 constituent stock prices
> load("C:/Develop/R/lecture_slides/data/sp500.R
> price_s <- eapply(env_sp500, quantmod::Cl)
> price_s <- rutils::do_call(cbind, price_s)
> # carry forward and backward non-NA prices
> price_s <- zoo::na.locf(price_s)
> price_s <- zoo::na.locf(price_s, fromLast=TRUE
> colnames(price_s) <- sapply(colnames(price_s),
+   function(col_name) strsplit(col_name, split="[.]")[[1]][1])
> # calculate the percentage (log) returns of the S&P500 constituent stocks
> re_turns <- rutils::diff_it(price_s)
> da_ta <- rowSums(re_turns==0)
> da_ta <- xts::xts(da_ta, order.by=index(price_s))
> dygraphs::dygraph(da_ta, main="Number of S&P 500 Constituents Without Prices")
> # select data after 2007
> price_s <- price_s["2007/"]
> re_turns <- re_turns["2007/"]
> n_col <- NCOL(price_s)
> save(price_s,
+   file="C:/Develop/R/lecture_slides/data/sp500_prices.RData")
```

S&P500 Stock Portfolio Index

The price-weighted index of *S&P500* constituents closely follows the VTI *ETF*,

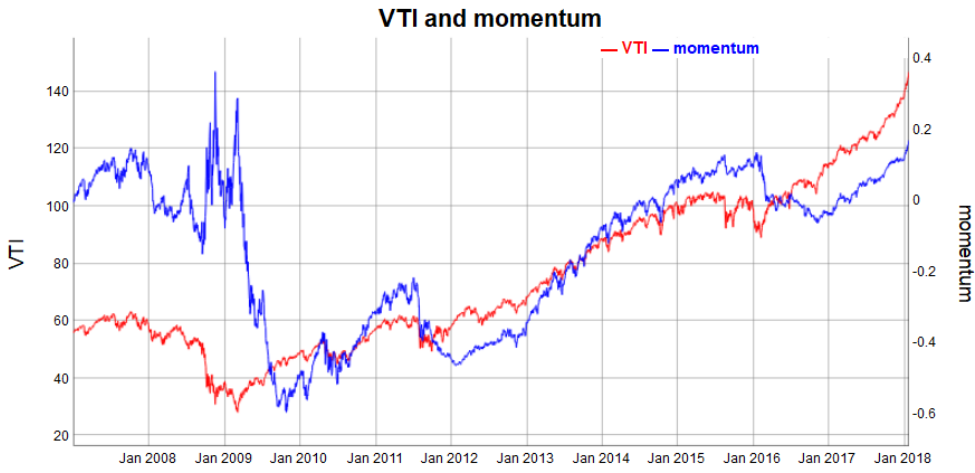



```
> # calculate price weighted index of constituen
> in_dex <- rowSums(price_s)/n_col
> # rescale to VTI
> in_dex <- as.numeric(env_etf$VTI[index(price_s
> in_dex <- xts::xts(in_dex, order.by=index(pric
> colnames(in_dex) <- "index"
> da_ta <- cbind(in_dex, env_etf$VTI[index(price
> col_names <- c("index", "VTI")
> colnames(da_ta) <- col_names
> # plot with VTI
> dygraphs::dygraph(da_ta,
+   main="S&P 500 Price-weighted Index and VTI") %>%
+   dyAxis("y", label=col_names[1], independentTicks=TRUE) %>%
+   dyAxis("y2", label=col_names[2], independentTicks=TRUE) %>%
+   dySeries(col_names[2], axis="y2", col=c("orange", "blue"))
```

Momentum Strategy for *S&P500* Stock Portfolio

A very simple *momentum* strategy for the *S&P500*, is to go long constituents with positive recent performance, and short constituents with negative performance,

This *momentum* strategy does not perform well and suffers from *momentum crashes* when the market rebounds sharply from a recent lows,




```
> # calculate rolling variance of S&P500 portfol
> wid_th <- 252
> vari_ance <- roll::roll_var(re_turns, width=wi
> vari_ance <- zoo::na.locf(vari_ance)
> vari_ance[is.na(vari_ance)] <- 0
> # calculate rolling Sharpe of S&P500 portfoli
> returns_width <- rutils::diff_it(price_s, lagg
> weight_s <- returns_width/sqrt(wid_th*vari_an
> weight_s[vari_ance==0] <- 0
> weight_s[1:wid_th, ] <- 1
> weight_s[is.na(weight_s)] <- 0
> weight_s <- weight_s/rowSums(abs(weight_s))/pi
> weight_s[is.na(weight_s)] <- 0
> weight_s <- rutils::lag_it(weight_s)
> sum(is.na(weight_s))
> # calculate portfolio profits and losses
> pnl_s <- rowSums(weight_s*re_turns)
```

```
> # Calculate transaction costs
> bid_offer <- 0.001
> cost_s <- 0.5*bid_offer*price_s*abs(rutils::di
> cost_s <- rowSums(cost_s)
> pnl_s <- (pnl_s - cost_s)
> pnl_s <- cumsum(pnl_s)
> pnl_s <- xts(pnl_s, order.by=index(price_s))
> pnl_s <- cbind(rutils::env_etf$VTI[, 4], pnl_s
> pnl_s <- na.omit(pnl_s)
> colnames(pnl_s) <- c("VTI", "momentum")
> col_names <- colnames(pnl_s)
> # plot momentum and VTI
> dygraphs::dygraph(pnl_s, main=paste(col_names,
+   dyAxis("y", label=col_names[1], independentT
+   dyAxis("y2", label=col_names[2], independent
+   dySeries(col_names[2], axis="y2", col=c("red
```

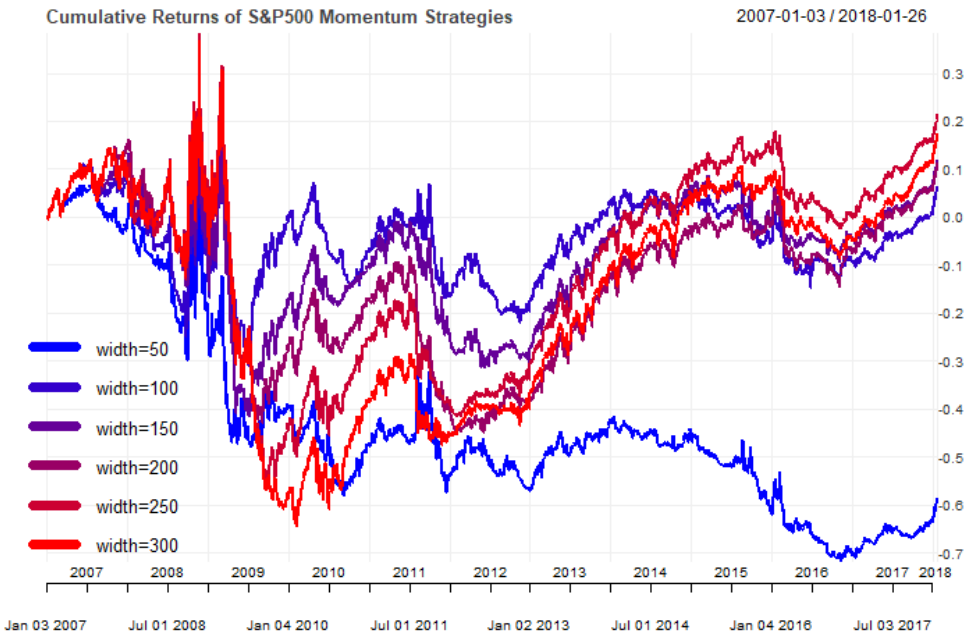
Backtesting Functional for *S&P500* Strategy

```
> back_test_rolling <- function(re_turns, price_s,
+     wid_th=252, bid_offer=0.001, tre_nd=1, ...) {
+   stopifnot("package:quantmod" %in% search() || require("quantmod", quietly=TRUE))
+   # Define look-back and look-forward intervals
+   n_col <- NCOL(re_turns)
+   vari_ance <- roll::roll_var(re_turns, width=wid_th)
+   vari_ance <- zoo::na.locf(vari_ance)
+   vari_ance[is.na(vari_ance)] <- 0
+   # calculate rolling Sharpe of S&P500 portfolio
+   returns_width <- rutils::diff_it(price_s, lagg=wid_th)
+   weight_s <- tre_nd*returns_width/sqrt(wid_th*vari_ance)
+   weight_s[vari_ance==0] <- 0
+   weight_s[1:wid_th, ] <- 1
+   weight_s[is.na(weight_s)] <- 0
+   weight_s <- weight_s/rowSums(abs(weight_s))/price_s
+   weight_s[is.na(weight_s)] <- 0
+   weight_s <- rutils::lag_it(weight_s)
+   # calculate portfolio profits and losses
+   pnl_s <- rowSums(weight_s*re_turns)
+   # Calculate transaction costs
+   bid_offer <- 0.001
+   cost_s <- 0.5*bid_offer*price_s*abs(rutils::diff_it(weight_s))
+   cost_s <- rowSums(cost_s)
+   pnl_s <- (pnl_s - cost_s)
+   pnl_s <- cumsum(pnl_s)
+   pnl_s
+ }  # end back_test_rolling
```

Simulating Multiple *S&P500 Momentum* Strategies

Multiple *S&P500 momentum* strategies can be simulated by calling the function `back_test_rolling()` in a loop over a vector of *width* parameters,

The *momentum* strategies do not perform well, especially the ones with a small *width* parameter,




```
> source("C:/Develop/R/lecture_slides/scripts/ba
> pnl_s <- back_test_rolling(wid_th=252, re_turn
+   price_s=price_s, bid_offer=bid_offer)
> width_s <- seq(50, 300, by=50)
> # perform sapply loop over lamb_das
> pro_files <- sapply(width_s, back_test_rolling
+   price_s=price_s, bid_offer=bid_offer)
> colnames(pro_files) <- paste0("width=", width
> pro_files <- xts(pro_files, index(price_s))
```

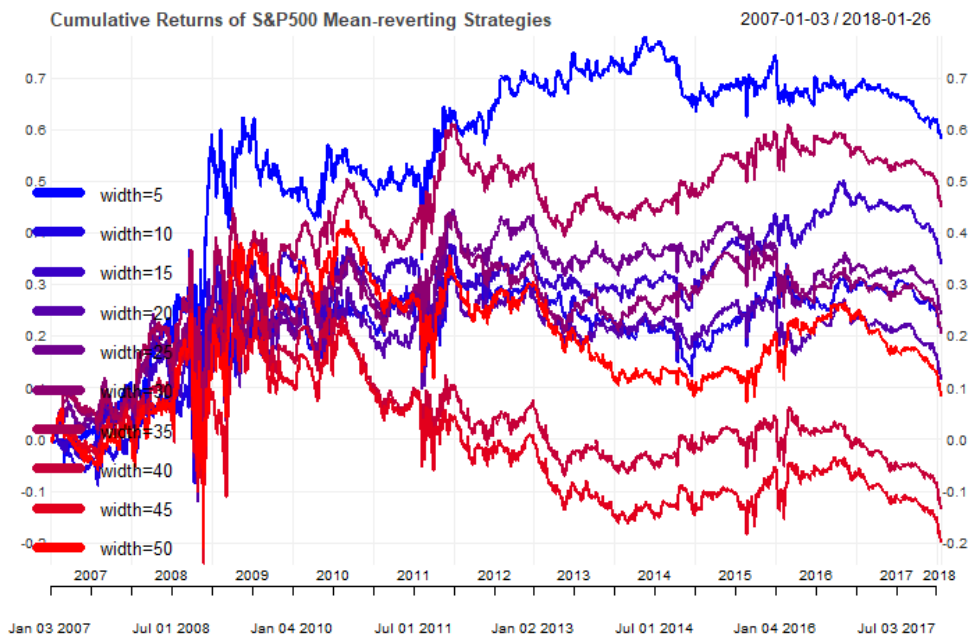
```
> # plot EWMA strategies with custom line colors
> plot_theme <- chart_theme()
> plot_theme$col$line.col <-
+   colorRampPalette(c("blue", "red"))(NCOL(pro_
> chart_Series(pro_files,
+   theme=plot_theme, name="Cumulative Returns o
> legend("bottomleft", legend=colnames(pro_files
+   inset=0.0, bg="white", cex=0.7, lwd=rep(6, N
+   col=plot_theme$col$line.col, bty="n")
```

Simulating Multiple *S&P500 Mean-reverting* Strategies

Multiple *S&P500 mean-reverting* strategies can be simulated by calling the function `back_test_rolling()` in a loop over a vector of *width* parameters,

The *mean-reverting* strategies perform best for small *width* parameters,

```
> width_s <- seq(5, 50, by=5)
> # perform sapply loop over lamb_das
> pro_files <- sapply(width_s, back_test_rolling
+   price_s=price_s, bid_offer=bid_offer, tre_nd
> colnames(pro_files) <- paste0("width=", width_
> pro_files <- xts(pro_files, index(price_s))
```




```
> # plot EWMA strategies with custom line colors
> plot_theme <- chart_theme()
> plot_theme$col$line.col <-
+   colorRampPalette(c("blue", "red"))(NCOL(pro_
> chart_Series(pro_files,
+   theme=plot_theme, name="Cumulative Returns o
> legend("bottomleft", legend=colnames(pro_files
+   inset=0.0, bg="white", cex=0.7, lwd=rep(6, N
+   col=plot_theme$col$line.col, bty="n")
```

Vector and Matrix Calculus

Let $\boldsymbol{v}$ and $\boldsymbol{w}$ be vectors, with $\boldsymbol{v} = \{v_i\}_{i=1}^{i=n}$, and let $\mathbb{1}$ be the unit vector, with $\mathbb{1} = \{1\}_{i=1}^{i=n}$,

Then the inner product of $\boldsymbol{v}$ and $\boldsymbol{w}$ can be written as $\boldsymbol{v}^T\boldsymbol{w} = \boldsymbol{w}^T\boldsymbol{v} = \sum_{i=1}^{n} v_i w_i$,

We can then express the sum of the elements of $\boldsymbol{v}$ as the inner product: $\boldsymbol{v}^T\mathbb{1} = \mathbb{1}^T\boldsymbol{v} = \sum_{i=1}^{n} v_i$,

And the sum of squares of $\boldsymbol{v}$ as the inner product: $\boldsymbol{v}^T\boldsymbol{v} = \sum_{i=1}^{n} v_i^2$,

Let $\mathbb{A}$ be a matrix, with $\mathbb{A} = \left\{A_{ij}\right\}_{i,j=1}^{i,j=n}$,

Then the inner product of matrix $\mathbb{A}$ with vectors $\boldsymbol{v}$ and $\boldsymbol{w}$ can be written as:

$$\boldsymbol{v}^T\mathbb{A}\,\boldsymbol{w} = \boldsymbol{w}^T\mathbb{A}^T\boldsymbol{v} = \sum_{i,j=1}^{n} A_{ij} v_i w_j$$

The derivative of a scalar variable with respect to a vector variable is a vector, for example:

$$\frac{d(\boldsymbol{v}^T\mathbb{1})}{d\boldsymbol{v}} = d_v[\boldsymbol{v}^T\mathbb{1}] = d_v[\mathbb{1}^T\boldsymbol{v}] = \mathbb{1}^T$$

$$d_v[\boldsymbol{v}^T\boldsymbol{w}] = d_v[\boldsymbol{w}^T\boldsymbol{v}] = \boldsymbol{w}^T$$

$$d_v[\boldsymbol{v}^T\mathbb{A}\,\boldsymbol{w}] = \boldsymbol{w}^T\mathbb{A}^T$$

$$d_v[\boldsymbol{v}^T\mathbb{A}\,\boldsymbol{v}] = \boldsymbol{v}^T\mathbb{A} + \boldsymbol{v}^T\mathbb{A}^T$$

Eigenvectors and Eigenvalues of Matrices

The vector w is an *eigenvector* of the matrix $\mathbb{A}$, if it satisfies the *eigenvalue* equation:

$$\mathbb{A}\, w = \lambda\, w$$

Where λ is the *eigenvalue* corresponding to the *eigenvector* w,

The number of *eigenvalues* of a matrix is equal to its dimension,

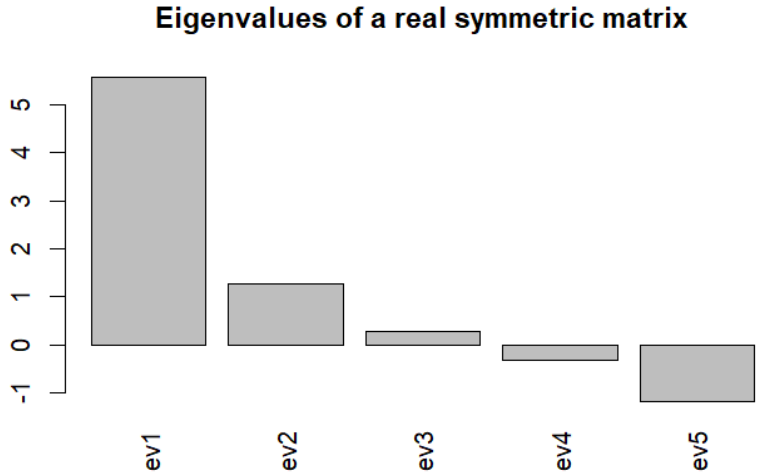
Real symmetric matrices have real *eigenvalues*, and their *eigenvectors* are orthogonal to each other,

The *eigenvectors* can be normalized to 1,

The *eigenvectors* form an *orthonormal basis* in which the matrix $\mathbb{A}$ is diagonal,

The function `eigen()` calculates the *eigenvectors* and *eigenvalues* of numeric matrices,

An excellent interactive visualization of *eigenvectors* and *eigenvalues* is available here:
http://setosa.io/ev/eigenvectors-and-eigenvalues/




```
> # create random real symmetric matrix
> mat_rix <- matrix(runif(25), nc=5)
> mat_rix <- mat_rix + t(mat_rix)
> # calculate eigenvectors and eigenvalues
> ei_gen <- eigen(mat_rix)
> eigen_vec <- ei_gen$vectors
> dim(eigen_vec)
> # plot eigenvalues
> barplot(ei_gen$values,
+   xlab="", ylab="", las=3,
+   names.arg=paste0("ev", 1:NROW(ei_gen$values)),
+   main="Eigenvalues of a real symmetric matrix")
```

Eigen Decomposition of Matrices

Real symmetric matrices have real *eigenvalues*, and their *eigenvectors* are orthogonal to each other,

The *eigenvectors* form an *orthonormal basis* in which the matrix $\mathbb{A}$ is diagonal:

$$\mathbb{D} = \mathbb{O}^T \mathbb{A} \mathbb{O}$$

Where $\mathbb{D}$ is a *diagonal* matrix containing the *eigenvalues* of matrix $\mathbb{A}$, and $\mathbb{O}$ is an *orthogonal* matrix of its *eigenvectors*,

Any real symmetric matrix $\mathbb{A}$ can be decomposed into a product of its *eigenvalues* and its *eigenvectors* (the *eigen decomposition*):

$$\mathbb{A} = \mathbb{O} \mathbb{D} \mathbb{O}^T$$

The *eigen decomposition* expresses a matrix as the product of a rotation, followed by a scaling, followed by the inverse rotation,

```
> # eigenvectors form an orthonormal basis
> round(t(eigen_vec) %*% eigen_vec,
+   digits=4)
> # diagonalize matrix using eigenvector matrix
> round(t(eigen_vec) %*% (mat_rix %*% eigen_vec)
+   digits=4)
> ei_gen$values
> # eigen decomposition of matrix by rotating th
> eigen_decomp <- eigen_vec %*% (ei_gen$values *
> # create diagonal matrix of eigenvalues
> # diago_nal <- diag(ei_gen$values)
> # eigen_decomp <- eigen_vec %*% (diago_nal %*%
> all.equal(mat_rix, eigen_decomp)
```

Orthogonal matrices represent rotations in *hyperspace*, and their inverse is equal to their transpose: $\mathbb{O}^{-1} = \mathbb{O}^T$,

The *diagonal* matrix $\mathbb{D}$ represents a scaling (stretching) transformation proportional to the *eigenvalues*,

The %*% operator performs *inner* (*scalar*) multiplication of vectors and matrices,

Inner multiplication multiplies the rows of one matrix with the columns of another matrix, so

Positive Definite Matrices

Matrices with positive *eigenvalues* are called *positive definite* matrices,

Matrices with non-negative *eigenvalues* are called *positive semi-definite* matrices (some of their *eigenvalues* may be zero),

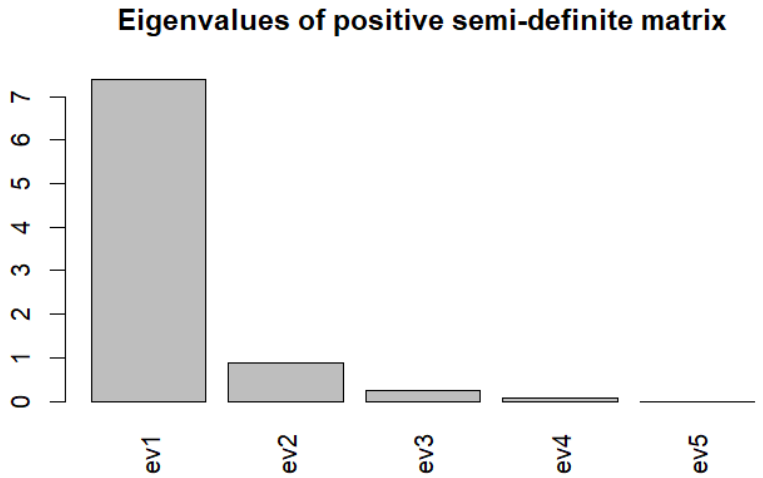
An example of *positive definite* matrices are the covariance matrices of linearly independent variables,

But the covariance matrices of linearly dependent variables have some *eigenvalues* equal to zero, in which case they are *singular*, and only *positive semi-definite*,

All covariance matrices are *positive semi-definite* and all *positive semi-definite* matrices are the covariance matrix of some multivariate distribution,

Matrices which have some *eigenvalues* equal to zero are called *singular* (degenerate) matrices,

For any real matrix $\mathbb{A}$, the matrix $\mathbb{A}^T\mathbb{A}$ is *positive semi-definite*,




```
> # create random positive semi-definite matrix
> mat_rix <- matrix(runif(25), nc=5)
> mat_rix <- t(mat_rix) %*% mat_rix
> # calculate eigenvectors and eigenvalues
> ei_gen <- eigen(mat_rix)
> ei_gen$values
> # plot eigenvalues
> barplot(ei_gen$values, las=3,
+   xlab="", ylab="",
+   names.arg=paste0("ev", 1:NROW(ei_gen$values)),
+   main="Eigenvalues of positive semi-definite
```

Singular Value Decomposition (*SVD*) of Matrices

The *Singular Value Decomposition* (*SVD*) is a generalization of the *eigen decomposition* of *positive semi-definite* matrices,

The *SVD* of a rectangular matrix $\mathbb{A}$ with dimensions `m` rows and `n` columns is defined as the factorization:

$$\mathbb{A} = \mathbb{U}\mathbb{\Sigma}\mathbb{V}^T$$

If (`m > n`), then $\mathbb{U}$ is an (`m x n`) rectangular matrix, $\mathbb{\Sigma}$ is an (`n x n`) diagonal matrix containing the *singular values*, and $\mathbb{V}$ is an (`n x n`) *orthogonal* matrix, and vice versa if (`m < n`),

The (`m x n`) rectangular matrix $\mathbb{U}$ consists of `n` columns of *orthonormal* left-*singular* vectors, with $\mathbb{U}^T\mathbb{U} = \mathbb{1}$,

The columns of the *orthogonal* matrix $\mathbb{V}$ consist of `n` *orthonormal* right-*singular* vectors, with $\mathbb{V}^T\mathbb{V} = \mathbb{1}$,

The *singular* vectors are only defined up to a reflection (sign), i.e. if `vec` is a *singular* vector, then so is `-vec`,

```
> # dimensions of left and right matrices
> n_left <- 6 ; n_right <- 4
> # create random positive semi-definite matrix
> left_mat <- matrix(runif(n_left^2), nc=n_left)
> left_mat <- crossprod(left_mat)
> # or
> left_mat <- left_mat %*% t(left_mat)
> # calculate left eigenvectors
> ei_gen <- eigen(left_mat)
> left_mat <- ei_gen$vectors[, 1:n_right]
> # create random positive semi-definite matrix
> right_mat <- matrix(runif(n_right^2), nc=n_rig
> right_mat <- crossprod(right_mat)
> # or
> right_mat <- right_mat %*% t(right_mat)
> # calculate right eigenvectors and singular va
> ei_gen <- eigen(right_mat)
> right_mat <- ei_gen$vectors
> sing_values <- ei_gen$values
> # compose rectangular matrix
> mat_rix <-
+   left_mat %*% (sing_values * t(right_mat))
> # mat_rix <- left_mat %*% diag(sing_values) %*
```

In the special case when $\mathbb{A}$ is a *positive semi-definite* matrix, the *SVD* reduces to the *eigen decomposition*, with $\mathbb{U} = \mathbb{V}$,

Singular Value Decomposition Using Function svd()

The *SVD* of a rectangular matrix $\mathbb{A}$ with dimensions m rows and n columns is defined as the factorization:

$$\mathbb{A} = \mathbb{U}\Sigma\mathbb{V}^T$$

The (m x n) rectangular matrix $\mathbb{U}$ consists of n columns of *orthonormal* left-*singular* vectors, with $\mathbb{U}^T\mathbb{U} = \mathbb{1}$,

The left-*singular* vectors are the *eigenvectors* of the matrix $\mathbb{A}\mathbb{A}^T$,

The columns of the *orthogonal* matrix $\mathbb{V}$ consist of n *orthonormal* right-*singular* vectors, with $\mathbb{V}^T\mathbb{V} = \mathbb{1}$,

The right-*singular* vectors are the *eigenvectors* of the matrix $\mathbb{A}^T\mathbb{A}$,

The left-*singular* matrix $\mathbb{U}$ combined with the right-*singular* matrix $\mathbb{V}$ define a rotation transformation into a coordinate system where the matrix $\mathbb{A}$ becomes diagonal:

$$\Sigma = \mathbb{U}^T\mathbb{A}\mathbb{V}$$

```
> # perform singular value decomposition
> s_vd <- svd(mat_rix)
> # compare SVD with inputs
> all.equal(abs(s_vd$u), abs(left_mat))
> all.equal(abs(s_vd$v), abs(right_mat))
> all.equal(s_vd$d, ei_gen$values)
```

The function svd() performs *Singular Value Decomposition* (*SVD*) of a rectangular matrix, and returns a list of three elements: the *singular values*, and the matrices of left-*singular* vectors and the right-*singular* vectors,

Inverse of Square Matrices

The inverse of a square matrix $\mathbb{A}$ is defined as a square matrix $\mathbb{A}^{-1}$ that satisfies the equation:

$$\mathbb{A}^{-1}\mathbb{A} = \mathbb{A}\mathbb{A}^{-1} = \mathbb{1}$$

Where $\mathbb{1}$ is the identity matrix,

The inverse $\mathbb{A}^{-1}$ matrix can also be expressed as a product of the inverse of its *eigenvalues* ($\mathbb{D}$) and its *eigenvectors* ($\mathbb{O}$):

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

But *singular* (degenerate) matrices (which have some *eigenvalues* equal to zero) don't have an inverse,

The function `solve()` solves systems of linear equations, and also inverts square matrices,

```
> # create random positive semi-definite matrix
> mat_rix <- matrix(runif(25), nc=5)
> mat_rix <- t(mat_rix) %*% mat_rix
> # calculate the inverse of mat_rix
> in_verse <- solve(a=mat_rix)
> # multiply inverse with matrix
> round(in_verse %*% mat_rix, 4)
> round(mat_rix %*% in_verse, 4)
>
> # calculate eigenvectors and eigenvalues
> ei_gen <- eigen(mat_rix)
> eigen_vec <- ei_gen$vectors
>
> # perform eigen decomposition of inverse
> eigen_inverse <-
+   eigen_vec %*% (t(eigen_vec) / ei_gen$values)
> all.equal(in_verse, eigen_inverse)
> # decompose diagonal matrix with inverse of ei
> # diago_nal <- diag(1/ei_gen$values)
> # eigen_inverse <-
> #   eigen_vec %*% (diago_nal %*% t(eigen_vec))
```

Generalized Inverse of Rectangular Matrices

The generalized inverse of an (`m x n`) rectangular matrix $\mathbb{A}$ is defined as an (`n x m`) matrix $\mathbb{A}^{-1}$ that satisfies the equation:

$$\mathbb{A}\mathbb{A}^{-1}\mathbb{A} = \mathbb{A}$$

The generalized inverse matrix $\mathbb{A}^{-1}$ can be expressed as a product of the inverse of its *singular values* (Σ) and its left and right *singular* matrices ($\mathbb{U}$ and $\mathbb{V}$):

$$\mathbb{A}^{-1} = \mathbb{V}\,\Sigma^{-1}\,\mathbb{U}^T$$

The generalized inverse $\mathbb{A}^{-1}$ can also be expressed as the *Moore-Penrose pseudo-inverse*:

$$\mathbb{A}^{-1} = (\mathbb{A}^T\mathbb{A})^{-1}\mathbb{A}^T$$

In the case when the inverse matrix $\mathbb{A}^{-1}$ exists, then the *pseudo-inverse* matrix simplifies to the inverse: $(\mathbb{A}^T\mathbb{A})^{-1}\mathbb{A}^T = \mathbb{A}^{-1}(\mathbb{A}^T)^{-1}\mathbb{A}^T = \mathbb{A}^{-1}$

The function `MASS::ginv()` calculates the generalized inverse of a matrix,

```
> # create random rectangular matrix
> # case when: n_left > n_right
> n_left <- 6 ; n_right <- 4
> mat_rix <- matrix(runif(n_left*n_right),
+   nc=n_right)
> # calculate generalized inverse of mat_rix
> in_verse <- MASS::ginv(mat_rix)
> round(in_verse %*% mat_rix, 4)
> all.equal(mat_rix,
+     mat_rix %*% in_verse %*% mat_rix)
> # create random rectangular matrix
> # case when: n_left < n_right
> n_left <- 4 ; n_right <- 6
> mat_rix <- matrix(runif(n_left*n_right),
+   nc=n_right)
> # calculate generalized inverse of mat_rix
> in_verse <- MASS::ginv(mat_rix)
> round(mat_rix %*% in_verse, 4)
> # perform singular value decomposition
> s_vd <- svd(mat_rix)
> # calculate generalized inverse from SVD
> svd_inverse <- s_vd$v %*% (t(s_vd$u) / s_vd$d)
> all.equal(svd_inverse, in_verse)
> # calculate Moore-Penrose pseudo-inverse
> mp_inverse <-
+   MASS::ginv(t(mat_rix) %*% mat_rix) %*% t(mat
> all.equal(mp_inverse, in_verse)
```

Generalized Inverse of Singular Matrices

Singular matrices have some *singular values* equal to zero, so they don't have an inverse matrix which satisfies the equation:
$\mathbb{A}^{-1}\mathbb{A} = \mathbb{A}\mathbb{A}^{-1} = \mathbb{1}$,

But if the *singular values* that are equal to zero are removed, then a generalized inverse for *singular* matrices can be specified by:

$$\mathbb{A}^{-1} = \mathbb{V}_n \, \mathbf{\Sigma}_n^{-1} \, \mathbb{U}_n^T$$

Where $\mathbb{U}_n$, $\mathbb{V}_n$ and $\mathbf{\Sigma}_n$ are the *SVD* matrices with rows and columns corresponding to zero *singular values* removed,

```
> # create random singular matrix
> n_left <- 4 ; n_right <- 6
> mat_rix <- matrix(runif(n_left*n_right), nc=n_right)
> mat_rix <- t(mat_rix) %*% mat_rix
> # calculate generalized inverse of mat_rix
> in_verse <- MASS::ginv(mat_rix)
> # verify inverse of mat_rix
> all.equal(mat_rix,
+   mat_rix %*% in_verse %*% mat_rix)
```

```
> # perform singular value decomposition
> s_vd <- svd(mat_rix)
> # set tolerance for determining zero singular
> to_l <- sqrt(.Machine$double.eps)
> # check for zero singular values
> s_vd$d
> not_zero <- (s_vd$d > (to_l * s_vd$d[1]))
> # calculate generalized inverse from SVD
> svd_inverse <-
+   s_vd$v[, not_zero] %*%
+   (t(s_vd$u[, not_zero]) / s_vd$d[not_zero])
> all.equal(svd_inverse, in_verse)
> # calculate Moore-Penrose pseudo-inverse
> mp_inverse <-
+   MASS::ginv(t(mat_rix) %*% mat_rix) %*% t(mat
> all.equal(mp_inverse, in_verse)
```

Diagonalizing Generalized Inverse of Singular Matrices

The left-*singular* matrix $\mathbb{U}$ combined with the right-*singular* matrix $\mathbb{V}$ define a rotation transformation into a coordinate system where the matrix $\mathbb{A}$ becomes diagonal:

$$\Sigma = \mathbb{U}^T \mathbb{A} \mathbb{V}$$

The generalized inverse of *singular* matrices doesn't satisfy the equation:
$\mathbb{A}^{-1}\mathbb{A} = \mathbb{A}\mathbb{A}^{-1} = \mathbb{1}$, but if it's rotated into the same coordinate system where $\mathbb{A}$ is diagonal, then we have:

$$\mathbb{U}^T (\mathbb{A}^{-1}\mathbb{A}) \mathbb{V} = \mathbb{1}_n$$

So that $\mathbb{A}^{-1}\mathbb{A}$ is diagonal in the same coordinate system where $\mathbb{A}$ is diagonal,

```
> # diagonalize the "unit" matrix
> uni_t <- mat_rix %*% in_verse
> round(uni_t, 4)
> round(mat_rix %*% in_verse, 4)
> round(t(s_vd$u) %*% uni_t %*% s_vd$v, 4)
```

Solving Linear Equations Using solve()

A system of linear equations can be defined as:

$$\mathbb{A}\,x = b$$

Where $\mathbb{A}$ is a matrix, b is a vector, and x is the unknown vector,

The solution of the system of linear equations is equal to:

$$x = \mathbb{A}^{-1} b$$

Where $\mathbb{A}^{-1}$ is the *inverse* of the matrix $\mathbb{A}$,

The function solve() solves systems of linear equations, and also inverts square matrices,

The %*% operator performs *inner* (*scalar*) multiplication of vectors and matrices,

Inner multiplication multiplies the rows of one matrix with the columns of another matrix, so that each pair produces a single number:

```
> # define a square matrix
> mat_rix <- matrix(c(1, 2, -1, 2), nc=2)
> vec_tor <- c(2, 1)
> # calculate the inverse of mat_rix
> in_verse <- solve(a=mat_rix)
> in_verse %*% mat_rix
> # calculate solution using inverse of mat_rix
> solu_tion <- in_verse %*% vec_tor
> mat_rix %*% solu_tion
> # calculate solution of linear system
> solu_tion <- solve(a=mat_rix, b=vec_tor)
> mat_rix %*% solu_tion
```

Cholesky Decomposition

The *Cholesky* decomposition of a *positive definite* matrix $\mathbb{A}$ is defined as:

$$\mathbb{A} = \mathbb{L}^T \mathbb{L}$$

Where $\mathbb{L}$ is an upper triangular matrix with positive diagonal elements,

The matrix $\mathbb{L}$ can be considered the square root of $\mathbb{A}$,

The vast majority of random *positive semi-definite* matrices are also *positive definite*,

The function `chol()` calculates the *Cholesky* decomposition of a *positive definite* matrix,

The functions `chol2inv()` and `chol()` calculate the inverse of a *positive definite* matrix two times faster than `solve()`,

```
> # create large random positive semi-definite m
> mat_rix <- matrix(runif(1e4), nc=100)
> mat_rix <- t(mat_rix) %*% mat_rix
> # calculate eigen decomposition
> ei_gen <- eigen(mat_rix)
> eigen_values <- ei_gen$values
> eigen_vec <- ei_gen$vectors
> # set tolerance for determining zero singular
> to_l <- sqrt(.Machine$double.eps)
> # if needed convert to positive definite matri
> not_zero <- (eigen_values > (to_l * eigen_valu
> if (sum(!not_zero) > 0) {
+   eigen_values[!not_zero] <- 2*to_l
+   mat_rix <- eigen_vec %*%
+     (eigen_values * t(eigen_vec))
+ }  # end if
> # calculate the Cholesky mat_rix
> choles_ky <- chol(mat_rix)
> choles_ky[1:5, 1:5]
> all.equal(mat_rix, t(choles_ky) %*% choles_ky)
> # calculate inverse from Cholesky
> chol_inverse <- chol2inv(choles_ky)
> all.equal(solve(mat_rix), chol_inverse)
> # compare speed of Cholesky inversion
> library(microbenchmark)
> summary(microbenchmark(
+   sol_ve=solve(mat_rix),
+   choles_ky=chol2inv(chol(mat_rix)),
```

Simulating Correlated Returns Using Cholesky Matrix

The *Cholesky* decomposition of a covariance matrix can be used to simulate correlated *Normal* returns following the given covariance matrix: $\mathbb{C} = \mathbb{L}^T\mathbb{L}$

Let $\mathbb{R}$ be a matrix with columns of *uncorrelated* returns following the *Standard Normal* distribution,

The *correlated* returns $\mathbb{R}_c$ can be calculated from the *uncorrelated* returns $\mathbb{R}$ by multilying them by the *Cholesky* matrix $\mathbb{L}$:

$$\mathbb{R}_c = \mathbb{L}^T\mathbb{R}$$

```
> # calculate random covariance matrix
> cov_mat <- matrix(runif(25), nc=5)
> cov_mat <- t(cov_mat) %*% cov_mat
> # calculate the Cholesky mat_rix
> choles_ky <- chol(cov_mat)
> choles_ky
> # simulate random uncorrelated returns
> n_assets <- 5
> n_rows <- 10000
> re_turns <- matrix(rnorm(n_assets*n_rows), nc=
> # calculate correlated returns by applying Cho
> corr_returns <- re_turns %*% choles_ky
> # calculate covariance matrix
> cov_returns <- crossprod(corr_returns) / (n_ro
> all.equal(cov_mat, cov_returns)
```

Eigenvalues of the Covariance Matrix

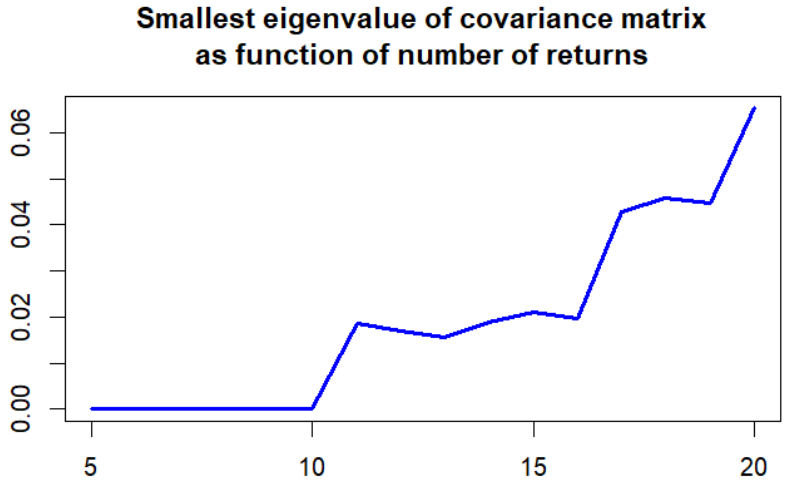
If $\mathbb{R}$ is a matrix of returns (with zero mean) for a portfolio of k assets (columns), over n time periods (rows), then the sample covariance matrix is equal to:

$$\mathbb{C} = \mathbb{R}^T \mathbb{R}/(n-1)$$

If the number of time periods of returns is less than the number of portfolio assets, then the returns are collinear, and the sample covariance matrix is *singular* (some *eigenvalues* are zero),

The function crossprod() performs *inner* (*scalar*) multiplication, exactly the same as the %*% operator, but it is slightly faster,

```
> # simulate random portfolio returns
> n_assets <- 10
> n_rows <- 100
> re_turns <- matrix(rnorm(n_assets*n_rows), nc
> # de-mean the returns
> re_turns <- apply(re_turns, MARGIN=2, functio
> # calculate covariance matrix
> cov_mat <- crossprod(re_turns) / (n_rows-1)
> # calculate eigenvectors and eigenvalues
> ei_gen <- eigen(cov_mat)
> ei_gen$values
> barplot(ei_gen$values, # plot eigenvalues
```




```
> # calculate eigenvectors and eigenvalues
> # as function of number of returns
> n_data <- ((n_assets/2):(2*n_assets))
> e_values <- sapply(n_data, function(x) {
+   re_turns <- re_turns[1:x, ]
+   re_turns <- apply(re_turns, MARGIN=2,
+     function(y) (y-mean(y)))
+   cov_mat <- crossprod(re_turns) / (x-1)
+   min(eigen(cov_mat)$values)
+ })  # end sapply
> plot(y=e_values, x=n_data, t="l",
+   xlab="", ylab="", lwd=3, col="blue",
+   main="Smallest eigenvalue of covariance matr
```

Package *RcppArmadillo* for Fast Linear Algebra

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

Armadillo provides ease of use and speed, with syntax similar to *Matlab*,

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

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

```
> library(RcppArmadillo)
> # source Rcpp functions from file
> Rcpp::sourceCpp(file="C:/Develop/R/lecture_sl
> vec1 <- runif(1e5)
> vec2 <- runif(1e5)
> vec_in(vec1, vec2)
> vec1 %*% vec2
```

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

// Examples of RcppArmadillo functions below

// vec_in() calculates the inner (dot) product of two ve
// It accepts pointers to the two vectors and returns a
//' @export
// [[Rcpp::export]]
double vec_in(const arma::vec& vec1, const arma::vec& ve
  return arma::dot(vec1, vec2);
}  // end vec_in

// mat_2vec_in() calculates the inner (dot) product of a
// with two vectors.
// It accepts pointers to the matrix and vectors, and re
//' @export
// [[Rcpp::export]]
double mat_2vec_in(const arma::vec& vec_tor2, const arma
  return arma::as_scalar(trans(vec_tor2) * (mat_rix * ve
}  // end mat_2vec_in
```

```
> # microbenchmark RcppArmadillo code
> summary(microbenchmark(
+   vec_in=vec_in(vec1, vec2),
+   r_code=(vec1 %*% vec2),
+   times=100))[, c(1, 4, 5)]  # end microbenchm
> # microbenchmark shows:
> # vec_in() is several times faster than %*%, e
> #     expr     mean   median
> # 1 vec_in 110.7067 110.4530
> # 2 r_code 585.5127 591.3575
```

Fast Matrix Algebra Using *RcppArmadillo*

RcppArmadillo functions can be made even faster by operating on pointers to matrices and performing calculations in place, without copying large matrices,

RcppArmadillo functions can be compiled using the same *Rtools* as those for *Rcpp* functions:
https://cran.r-project.org/bin/windows/Rtools/

```
> library(RcppArmadillo)
> # source Rcpp functions from file
> Rcpp::sourceCpp(file="C:/Develop/R/lecture_sli
> mat_rix <- matrix(runif(1e5), nc=1e3)
> # de-mean using apply()
> new_mat <- apply(mat_rix, 2,
+   function(x) (x-mean(x)))
> # de-mean using demean_mat()
> demean_mat(mat_rix)
> all.equal(new_mat, mat_rix)
> # microbenchmark RcppArmadillo code
> summary(microbenchmark(
+   demean_mat=demean_mat(mat_rix),
+   apply=(apply(mat_rix, 2, mean)),
+   times=100))[, c(1, 4, 5)]  # end microbenchmark summary
> # microbenchmark shows:
> # demean_mat() is over 70 times faster than apply()
> #         expr       mean   median
```

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

// Examples of RcppArmadillo functions below

// demean_mat() calculates a matrix with de-meaned colum
// It accepts a pointer to a matrix and operates on the
// It returns the number of columns of the input matrix.
//' @export
// [[Rcpp::export]]
double demean_mat(arma::mat& mat_rix){
  for (unsigned int i = 0; i < mat_rix.n_cols; i++) {
    mat_rix.col(i) -= arma::mean(mat_rix.col(i));
  }  // end for
  return mat_rix.n_cols;
}  // end demean_mat

// inv_mat() calculates the inverse of symmetric positiv
// definite matrix.
// It accepts a pointer to a matrix and operates on the
// It returns the number of columns of the input matrix.
// It uses RcppArmadillo.
//' @export
// [[Rcpp::export]]
double inv_mat(arma::mat& mat_rix){
  mat_rix = arma::inv_sympd(mat_rix);
  return mat_rix.n_cols;
}  // end inv_mat
```

Formula Objects

Formulas in R are defined using the "~" operator followed by a series of terms separated by the "+" operator,

Formulas can be defined as separate objects, manipulated, and passed to functions,

The formula "z ~ x" means the response variable z is explained by the explanatory variable x,

The formula "z ~ x + y" represents a linear model: z = ax + by + c,

The formula "z ~ x - 1" or "z ~ x + 0" represents a linear model with zero intercept: $z = ax$,

The function update() modifies existing formulas,

The "." symbol represents either all the remaining data, or the variable that was in this part of the formula,

```
> # formula of linear model with zero intercept
> lin_formula <- z ~ x + y - 1
> lin_formula
>
> # collapse vector of strings into single text
> paste0("x", 1:5)
> paste(paste0("x", 1:5), collapse="+")
>
> # create formula from text string
> lin_formula <- as.formula(
+   # coerce text strings to formula
+   paste("z ~ ",
+   paste(paste0("x", 1:5), collapse="+")
+   )  # end paste
+ )  # end as.formula
> class(lin_formula)
> lin_formula
> # modify the formula using "update"
> update(lin_formula, log(.) ~ . + beta)
```

Simple *Linear Regression*

A Simple Linear Regression is a linear model between a response variable `z` and a single explanatory variable `x`, defined by the formula:

$$y_i = \alpha + \beta x_i + \varepsilon_i$$

α and β are the unknown regression coefficients,

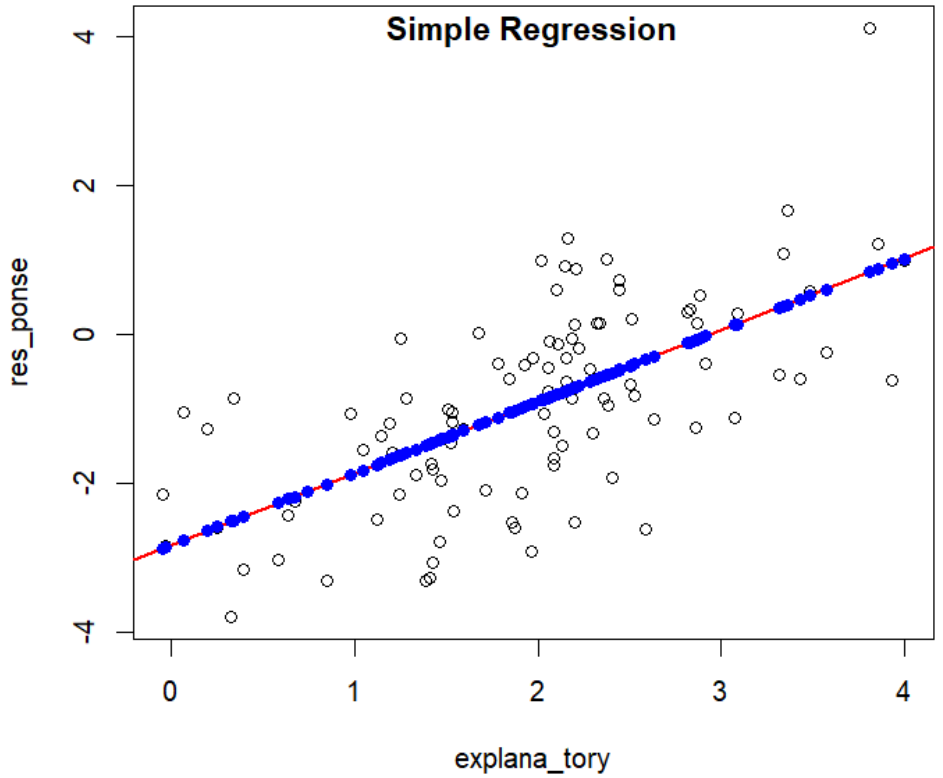
ε_i are the residuals, assumed to be normally distributed, independent, and stationary,

In the Ordinary Least Squares method (*OLS*), the regression parameters are estimated by minimizing the sum of squared residuals, also called the *Residual Sum of Squares* (*RSS*):

$$RSS = \sum_{i=1}^{n} \varepsilon_i^2 = \sum_{i=1}^{n} (y_i - \alpha - \beta x_i)^2$$
$$= (y - \alpha \mathbb{1} - \beta x)^T (y - \alpha \mathbb{1} - \beta x)$$

Where $\mathbb{1}$ is the unit vector, with $\mathbb{1}^T \mathbb{1} = n$ and $\mathbb{1}^T x = x^T \mathbb{1} = \sum_{i=1}^{n} x_i$

The data consists of `n` pairs of observations (x_i, y_i) of the response and explanatory variables, with the index `i` ranging from 1 to `n`.




```
> # define explanatory variable
> len_gth <- 100
> ex_plain <- rnorm(len_gth, mean=2)
> noise <- rnorm(len_gth)
> # response equals linear form plus error terms
> res_ponse <- -3 + ex_plain + noise
```

Solution of *Linear Regression*

The *OLS* solution for the regression coefficients is found by equating the *RSS* derivatives to zero:

$$RSS_\alpha = -2(y - \alpha\mathbb{1} - \beta x)^T \mathbb{1} = 0$$

$$RSS_\beta = -2(y - \alpha\mathbb{1} - \beta x)^T x = 0$$

The solution for α is given by:

$$\alpha = \bar{y} - \beta\bar{x}$$

The solution for β is given by:

$$(y - (\bar{y} - \beta\bar{x})\mathbb{1} - \beta x)^T (x - \bar{x}\mathbb{1}) = 0$$

$$((y - \bar{y}\mathbb{1}) - \beta(x - \bar{x}\mathbb{1}))^T (x - \bar{x}\mathbb{1}) = 0$$

$$((y - \bar{y}\mathbb{1}) - \beta(x - \bar{x}\mathbb{1}))^T (x - \bar{x}\mathbb{1}) = 0$$

$$(y - \bar{y}\mathbb{1})^T (x - \bar{x}\mathbb{1}) - \beta(x - \bar{x}\mathbb{1})^T (x - \bar{x}\mathbb{1}) = 0$$

$$\beta = \frac{(y - \bar{y}\mathbb{1})^T (x - \bar{x}\mathbb{1})}{(x - \bar{x}\mathbb{1})^T (x - \bar{x}\mathbb{1})} = \frac{\sigma_y}{\sigma_x}\rho_{xy}$$

β is proportional to the correlation coefficient ρ_{xy} between the response and explanatory variables,

```
> # calculate de-meaned explanatory and response
> explain_zm <- ex_plain - mean(ex_plain)
> response_zm <- res_ponse - mean(res_ponse)
> # solve for regression beta
> be_ta <- sum(explain_zm*response_zm) / sum(exp
> # solve for regression alpha
> al_pha <- mean(res_ponse) - be_ta*mean(ex_plai
```

If the response and explanatory variables have zero mean, then $\alpha = 0$ and $\beta = \frac{y^T x}{x^T x}$,

The residuals $\varepsilon = y - \alpha\mathbb{1} - \beta x$ have zero mean: $RSS_\alpha = -2\varepsilon^T \mathbb{1} = 0$,

The residuals ε are orthogonal to the explanatory variable `x`: $RSS_\beta = -2\varepsilon^T x = 0$,

Linear Regression Using Function `lm()`

Let the data generating process for the response variable be given as: $z = \alpha_{lat} + \beta_{lat} x + \varepsilon_{lat}$

Where α_{lat} and β_{lat} are latent (unknown) coefficients, and ε_{lat} is an unknown vector of random noise (error terms),

The error terms are the difference between the measured values of the response minus the (unknown) actual response values,

The function `lm()` fits a linear model into a set of data, and returns an object of class `"lm"`, which is a list containing the results of fitting the model:

- call - the model formula,
- coefficients - the fitted model coefficients (α, β_j),
- residuals - the model residuals (response minus fitted values),

The regression residuals are not the same as the error terms, because the regression coefficients are not equal to the coefficients of the data generating process,

```
> # specify regression formula
> reg_formula <- res_ponse ~ ex_plain
> reg_model <- lm(reg_formula)  # perform regres
> class(reg_model)  # regressions have class lm
[1] "lm"
> attributes(reg_model)
$names
 [1] "coefficients"  "residuals"     "effects"
 [4] "rank"          "fitted.values" "assign"
 [7] "qr"            "df.residual"   "xlevels"
[10] "call"          "terms"         "model"

$class
[1] "lm"
> eval(reg_model$call$formula)  # regression for
res_ponse ~ ex_plain
> reg_model$coeff  # regression coefficients
(Intercept)    ex_plain
     -2.824       0.959
> coef(reg_model)
(Intercept)    ex_plain
     -2.824       0.959
> c(al_pha, be_ta)
[1] -2.824  0.959
```

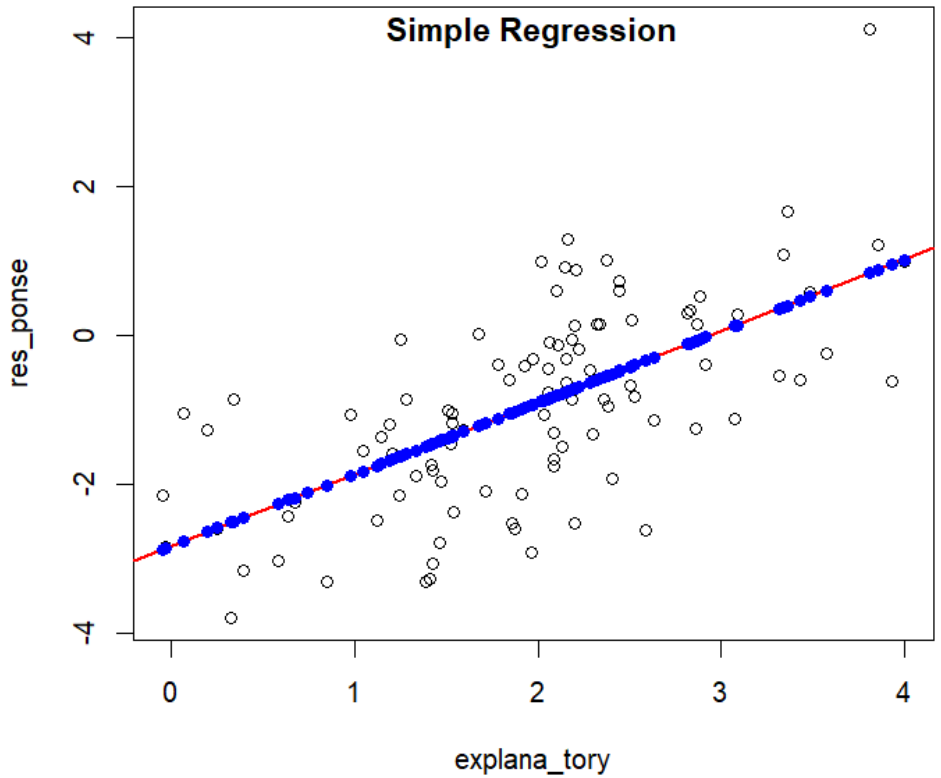
Linear Regression Scatterplot

The generic function `plot()` produces a scatterplot when it's called on the regression formula,

`abline()` plots a straight line corresponding to the regression coefficients, when it's called on the regression object,

The fitted (predicted) values are the values of the response variable obtained from applying the regression model to the explanatory variables,

```
> x11(width=6, height=5)  # open x11 for plottin
> # set plot parameters to reduce whitespace aro
> par(mar=c(5, 5, 1, 1), oma=c(0, 0, 0, 0))
> # plot scatterplot using formula
> plot(reg_formula)
> title(main="Simple Regression", line=-1)
> # add regression line
> abline(reg_model, lwd=2, col="red")
> # plot fitted (predicted) response values
> points(x=ex_plain, y=reg_model$fitted.values,
+        pch=16, col="blue")
```



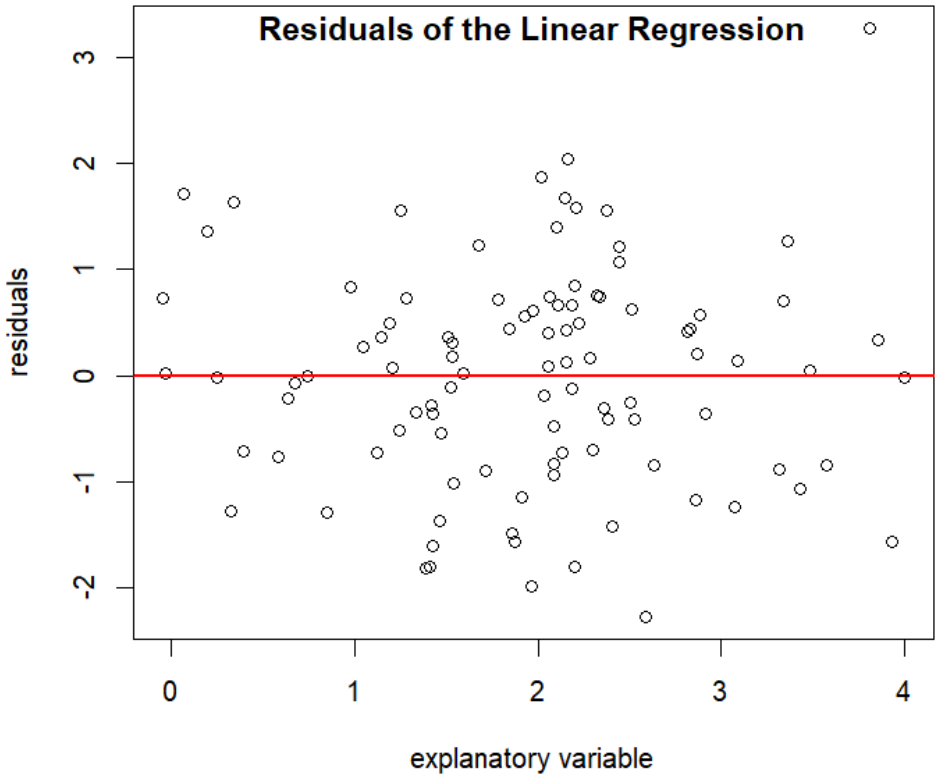
Linear Regression Residuals

The *residuals* of a *linear regression* are defined as the response variable minus the regression fitted values:

$$res_i = y_i - (\alpha + \beta x_i)$$

The *residuals* are the error terms associated with a particular realization of the response and explanatory variables,

The fitted (predicted) values are the values of the response variable obtained from applying the regression model to the explanatory variables,




```
> # sum of residuals = 0
> sum(reg_model$residuals)
> x11(width=6, height=5)  # open x11 for plottin
> # set plot parameters to reduce whitespace aro
> par(mar=c(5, 5, 1, 1), oma=c(0, 0, 0, 0))
> # extract residuals
> resi_duals <- cbind(ex_plain, reg_model$residuals)
> colnames(resi_duals) <- c("explanatory variable", "residuals")
> # plot residuals
> plot(resi_duals)
> title(main="Residuals of the Linear Regression", line=-1)
> abline(h=0, lwd=2, col="red")
```

Linear Regression Summary

The function `summary.lm()` produces a list of regression model diagnostic statistics:

- coefficients: matrix with estimated coefficients, their *t*-statistics, and *p*-values,
- r.squared: fraction of response variance explained by the model,
- adj.r.squared: r.squared adjusted for higher model complexity,
- fstatistic: ratio of variance explained by model divided by unexplained variance,

The regression *null* hypothesis is that the regression coefficients are *zero*,

The *t*-statistic (*t*-value) is the ratio of the estimated value divided by its standard error,

The *p*-value is the probability of obtaining the observed value of the *t*-statistic (and even more extreme values), under the *null* hypothesis,

A small *p*-value is often interpreted as meaning that the regression coefficients are very unlikely to be zero (given the data),

```
> reg_model_sum <- summary(reg_model)  # copy re
> reg_model_sum  # print the summary to console

Call:
lm(formula = reg_formula)

Residuals:
   Min     1Q Median     3Q    Max
-2.268 -0.737  0.037  0.669  3.268

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept)   -2.824      0.245  -11.53  < 2e-16
ex_plain       0.959      0.114    8.42  3.2e-13
---
Signif. codes:
0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 1.03 on 98 degrees of f
Multiple R-squared:  0.42,Adjusted R-squared:  0
F-statistic: 70.9 on 1 and 98 DF,  p-value: 3.16
> attributes(reg_model_sum)$names  # get summary
 [1] "call"          "terms"         "residuals"
 [4] "coefficients"  "aliased"       "sigma"
 [7] "df"            "r.squared"     "adj.r.squa
[10] "fstatistic"    "cov.unscaled"
```

Standard Errors of Regression Coefficients

The regression `summary` is a list, and its elements can be accessed individually,

The standard errors of the regression are the standard deviations of the coefficient estimators, given the residuals as the source of error,

The standard error of β in a simple regression is given by:

$$\sigma_\beta{}^2 = \frac{1}{(n-2)} \frac{E[(\varepsilon^T x)^2]}{(x^T x)^2} = \frac{1}{(n-2)} \frac{E[\varepsilon^2]}{(x^T x)} = \frac{1}{(n-2)} \frac{\sigma_\varepsilon{}^2}{\sigma_x{}^2}$$

The key assumption in the above formula for the standard error and the *p*-value is that the residuals are normally distributed, independent, and stationary,

If the residuals are not normally distributed, independent, and stationary, then the standard error and the *p*-value may be much bigger than reported by `summary.lm()`, and therefore the regression may not be statistically significant,

Market return time series are very far from normal, so the small *p*-values shouldn't be automatically interpreted as meaning that the regression is statistically significant,

```
> reg_model_sum$coeff
             Estimate Std. Error t value Pr(>|t|)
(Intercept)    -2.824      0.245  -11.53 5.97e-20
ex_plain        0.959      0.114    8.42 3.16e-13
> reg_model_sum$r.squared
[1] 0.42
> reg_model_sum$adj.r.squared
[1] 0.414
> reg_model_sum$fstatistic
value numdf dendf
 70.9   1.0  98.0
> # standard error of beta
> reg_model_sum$
+   coefficients["ex_plain", "Std. Error"]
[1] 0.114
> sd(reg_model_sum$residuals)/sd(ex_plain)/
+   sqrt(unname(reg_model_sum$fstatistic[3]))
[1] 0.114
> anova(reg_model)
Analysis of Variance Table

Response: res_ponse
          Df Sum Sq Mean Sq F value  Pr(>F)
ex_plain   1   75.6    75.6    70.9 3.2e-13 ***
Residuals 98  104.4     1.1
---
Signif. codes:
0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

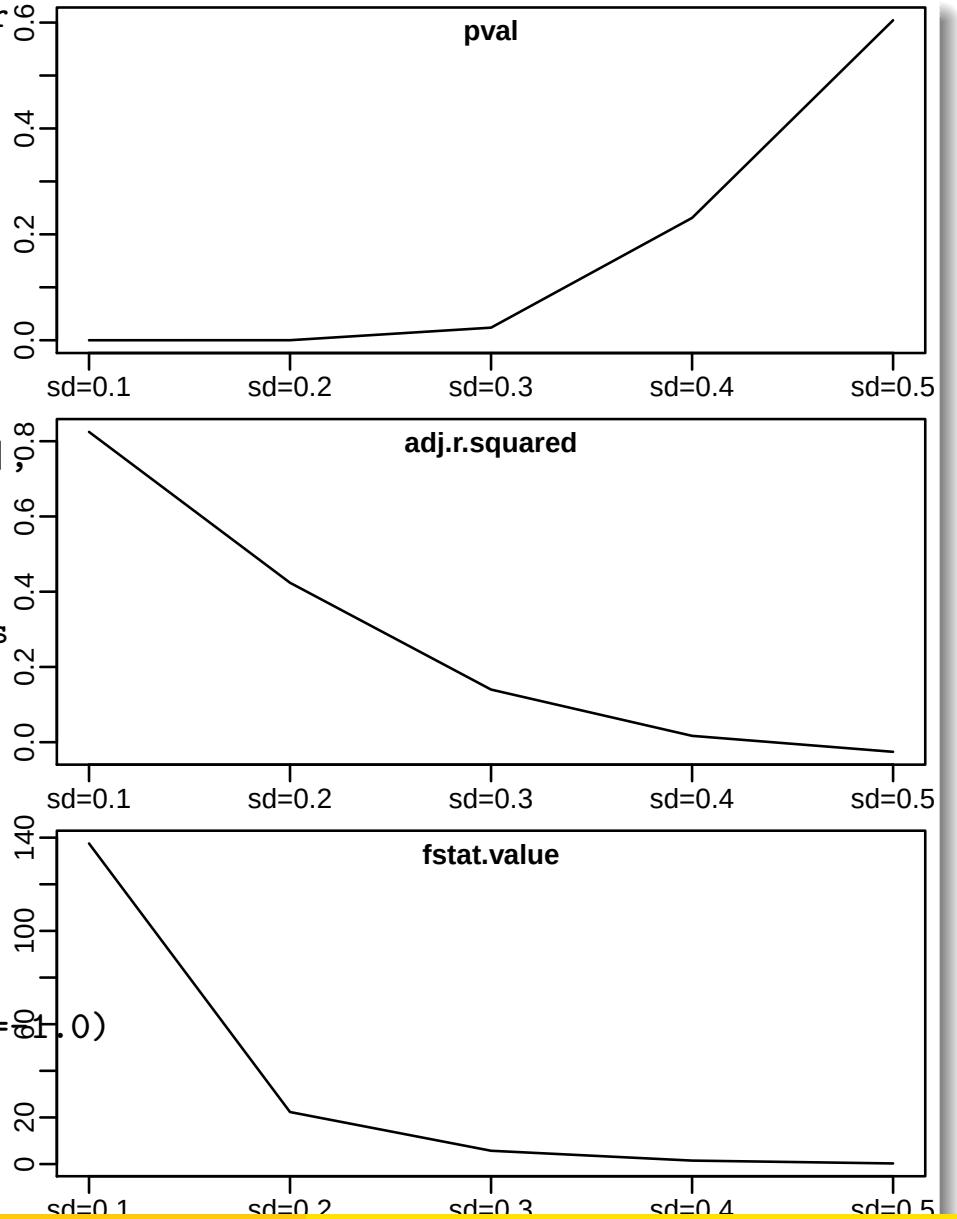
Weak Regression

If the relationship between the response and explanatory variables is weak compared to the error terms (noise), then the regression will have low statistical significance,

```
> # high noise compared to coefficient
> res_ponse <- 3 + ex_plain + rnorm(30, sd=8)
> reg_model <- lm(reg_formula)  # perform regres
> # values of regression coefficients are not
> # statistically significant
> summary(reg_model)

Call:
lm(formula = reg_formula)

Residuals:
    Min      1Q  Median      3Q     Max
-10.815  -3.188  -0.383   1.818  14.938

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept)    2.794      1.361    2.05   0.0427
ex_plain       1.795      0.633    2.84   0.0056
---
Signif. codes:
0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 5.74 on 98 degrees of f
Multiple R-squared:  0.0758,Adjusted R-squared:
F-statistic: 8.04 on 1 and 98 DF,  p-value: 0.00
```

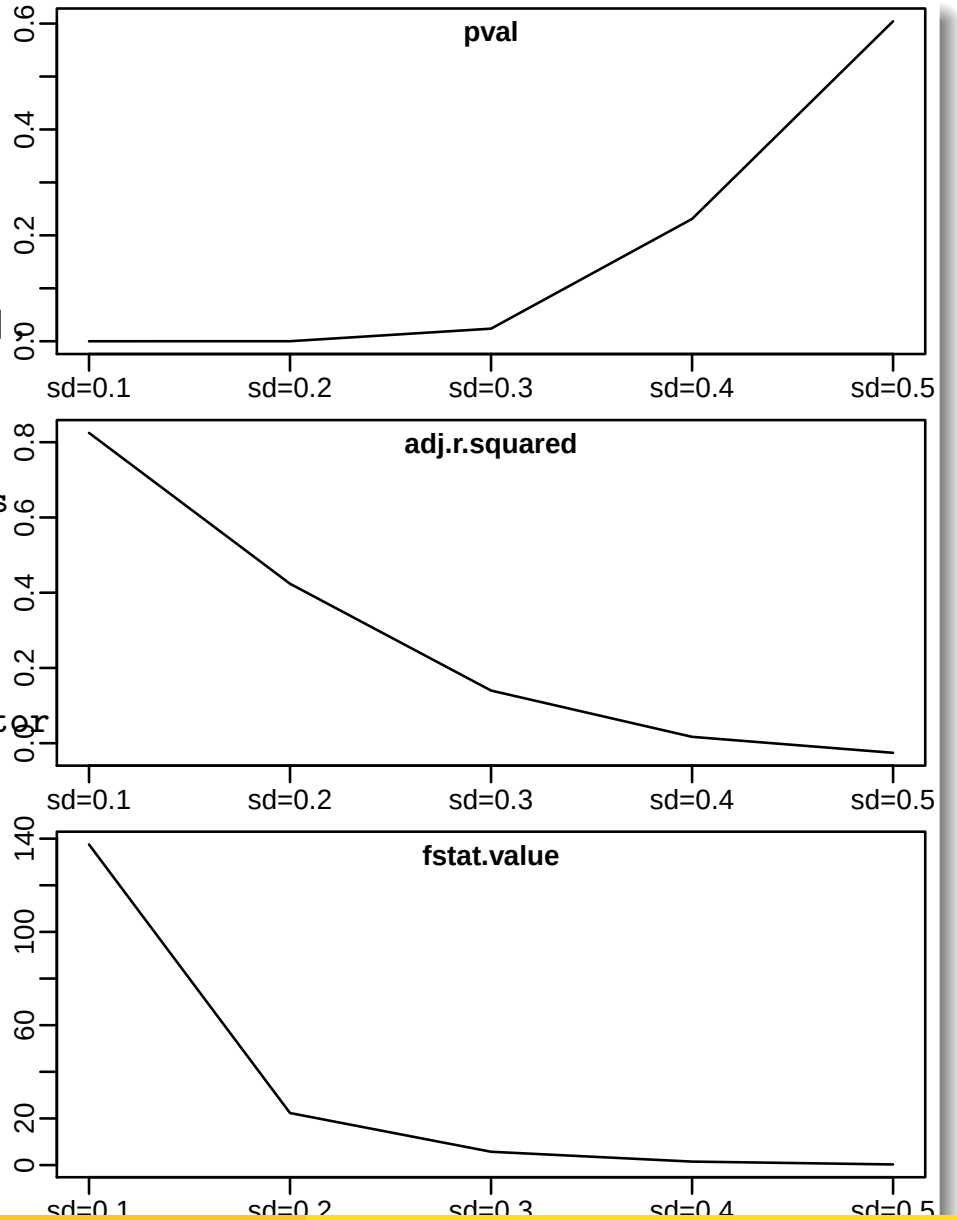
Influence of Noise on Regression

```
> reg_stats <- function(std_dev) {  # noisy regression
+   set.seed(1121)  # initialize number generator
+ # create explanatory and response variables
+   ex_plain <- rnorm(100, mean=2)
+   res_ponse <- 3 + 0.2*ex_plain +
+     rnorm(NROW(ex_plain), sd=std_dev)
+ # specify regression formula
+   reg_formula <- res_ponse ~ ex_plain
+ # perform regression and get summary
+   reg_model_sum <- summary(lm(reg_formula))
+ # extract regression statistics
+   with(reg_model_sum, c(pval=coefficients[2, 4],
+    adj_rsquared=adj.r.squared,
+    fstat=fstatistic[1]))
+ }  # end reg_stats
> # apply reg_stats() to vector of std dev values
> vec_sd <- seq(from=0.1, to=0.5, by=0.1)
> names(vec_sd) <- paste0("sd=", vec_sd)
> mat_stats <- t(sapply(vec_sd, reg_stats))
> # plot in loop
> par(mfrow=c(NCOL(mat_stats), 1))
> for (in_dex in 1:NCOL(mat_stats)) {
+   plot(mat_stats[, in_dex], type="l",
+  xaxt="n", xlab="", ylab="", main="")
+   title(main=colnames(mat_stats)[in_dex], line=-1.0)
+   axis(1, at=1:(NROW(mat_stats)),
+  labels=rownames(mat_stats))
+ }  # end for
```



Influence of Noise on Regression Another Method

```
> reg_stats <- function(da_ta) {  # get regression
+ # perform regression and get summary
+   col_names <- colnames(da_ta)
+   reg_formula <-
+     paste(col_names[2], col_names[1], sep="~")
+   reg_model_sum <- summary(lm(reg_formula,
+                           data=da_ta))
+ # extract regression statistics
+   with(reg_model_sum, c(pval=coefficients[2, 4],
+    adj_rsquared=adj.r.squared,
+    fstat=fstatistic[1]))
+ }  # end reg_stats
> # apply reg_stats() to vector of std dev values
> vec_sd <- seq(from=0.1, to=0.5, by=0.1)
> names(vec_sd) <- paste0("sd=", vec_sd)
> mat_stats <-
+   t(sapply(vec_sd, function (std_dev) {
+     set.seed(1121)  # initialize number generator
+ # create explanatory and response variables
+     ex_plain <- rnorm(100, mean=2)
+     res_ponse <- 3 + 0.2*ex_plain +
+ rnorm(NROW(ex_plain), sd=std_dev)
+     reg_stats(data.frame(ex_plain, res_ponse))
+     }))
> # plot in loop
> par(mfrow=c(NCOL(mat_stats), 1))
> for (in_dex in 1:NCOL(mat_stats)) {
+   plot(mat_stats[, in_dex], type="l",
+  xaxt="n", xlab="", ylab="", main="")
```



Linear Regression Diagnostic Plots

`plot()` produces diagnostic scatterplots for the residuals, when called on the regression object,

The diagnostic scatterplots allow for visual inspection to determine the quality of the regression fit,

"Residuals vs Fitted" is a scatterplot of the residuals vs. the predicted responses,

"Scale-Location" is a scatterplot of the square root of the standardized residuals vs. the predicted responses,

The residuals should be randomly distributed around the horizontal line representing zero residual error,

A pattern in the residuals indicates that the model was not able to capture the relationship between the variables, or that the variables don't follow the statistical assumptions of the regression model,

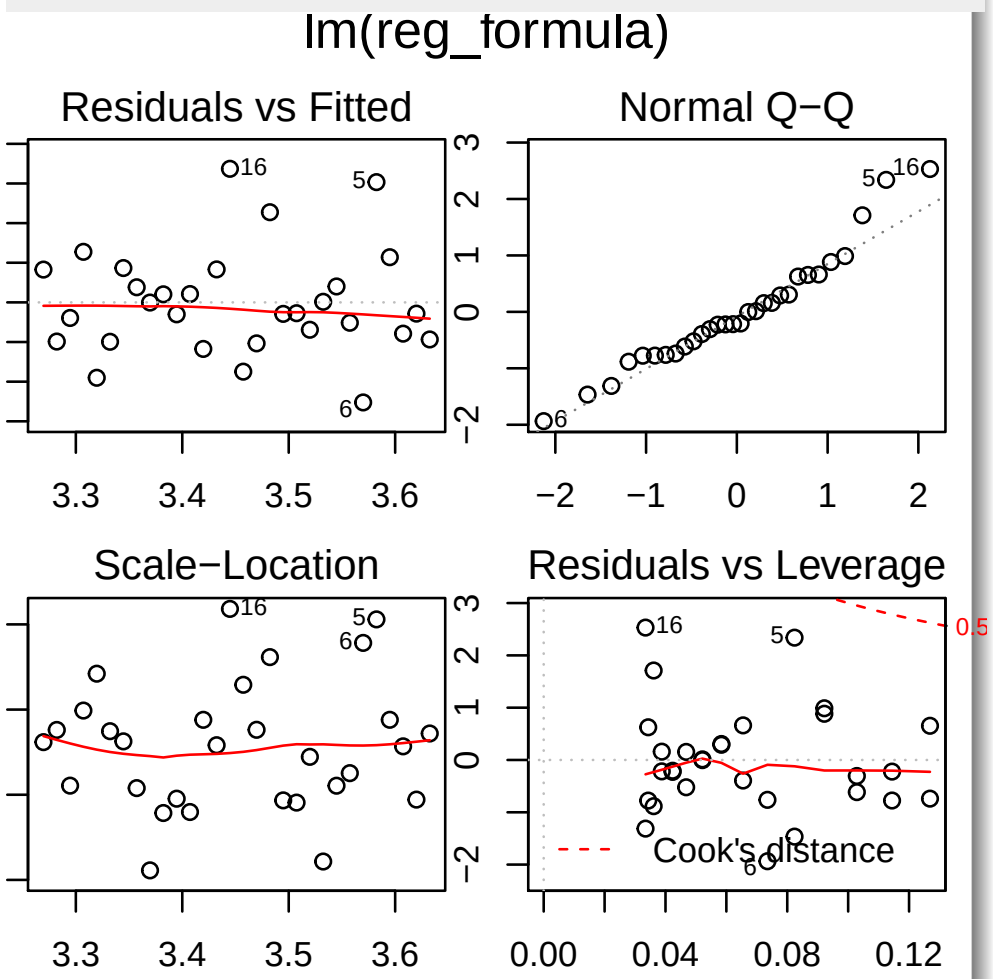
"Normal Q-Q" is the standard Q-Q plot, and the points should fall on the diagonal line, indicating that the residuals are normally distributed,

"Residuals vs Leverage" is a scatterplot of the residuals vs. their leverage,

Leverage measures the amount by which the predicted response would change if the observed response were shifted by a small amount,

Cook's distance measures the influence of a single observation on the predicted values, and is proportional

```
> par(mfrow=c(2, 2))  # plot 2x2 panels
> plot(reg_model)  # plot diagnostic scatterplot
> plot(reg_model, which=2)  # plot just Q-Q
```



Durbin-Watson Test of Autocorrelation of Residuals

The *Durbin-Watson* test is designed to test the *null hypothesis* that the autocorrelations of regression residuals are equal to zero,

The test statistic is equal to:

$$DW = \frac{\sum_{i=2}^{n}(\varepsilon_i - \varepsilon_{i-1})^2}{\sum_{i=1}^{n}\varepsilon_i^2}$$

Where ε_i are the regression residuals,

The value of the *Durbin-Watson* statistic *DW* is close to zero for large positive autocorrelations, and close to four for large negative autocorrelations,

The *DW* is close to two for autocorrelations close to zero,

The *p*-value for the `reg_model` regression is large, and we conclude that the *null hypothesis* is `TRUE`, and the regression residuals are uncorrelated,

```
> library(lmtest)  # load lmtest
> # perform Durbin-Watson test
> dwtest(reg_model)

Durbin-Watson test

data:  reg_model
DW = 2, p-value = 0.5
alternative hypothesis: true autocorrelation is
```

Multivariate *Linear Regression*

A multivariate *linear regression* model with `k` explanatory variables $\{x_j\}$, is defined by the formula:

$$y_i = \alpha + \sum_{j=1}^{k} \beta_j x_{i,j} + \varepsilon_i$$

α and β are the unknown regression coefficients, with α a scalar and β a vector of length `k`,

ε_i are the residuals, assumed to be normally distributed, independent, and stationary, with ε a vector of length `k`,

The response variable y and the `k` explanatory variables $\{x_j\}$ each contain `n` observations, and they form the columns of matrix $\mathbb{X}$,

The response variable y is a vector of length `n`, and the explanatory variable $\mathbb{X}$ is a (n, k)-dimensional matrix,

```
> # define explanatory variable
> len_gth <- 100
> n_var <- 5
> ex_plain <- matrix(rnorm(n_var*len_gth), nc=n_
> noise <- rnorm(len_gth, sd=1.0)
> # response equals linear form plus error terms
> weight_s <- rnorm(n_var)
> res_ponse <- -3 + ex_plain %*% weight_s + nois
```

The *multivariate regression* model can be written in vector notation as:

$$y = \alpha + \mathbb{X}\beta + \varepsilon = \hat{y} + \varepsilon$$
$$\hat{y} = \alpha + \mathbb{X}\beta$$

Where $\hat{y}$ are the *fitted values* of the model,

The *Residual Sum of Squares* (*RSS*) is defined as the sum of the squared residuals:

$$RSS = \varepsilon^T \varepsilon = (y - \hat{y})^T (y - \hat{y}) =$$
$$(y - \alpha + \mathbb{X}\beta)^T (y - \alpha + \mathbb{X}\beta)$$

Solution of Multivariate *Linear Regression*

The *OLS* solution for the regression coefficients is found by equating the *RSS* derivatives to zero:

$$RSS_\alpha = -2(y - \alpha - \mathbb{X}\beta)^T \mathbb{1} = 0$$

$$RSS_\beta = -2(y - \alpha - \mathbb{X}\beta)^T \mathbb{X} = 0$$

The solutions for α and β are given by:

$$\alpha = \bar{y} - \bar{\mathbb{X}}\beta$$

$$RSS_\beta = -2(\tilde{y} - \tilde{\mathbb{X}}\beta)^T \tilde{\mathbb{X}} = 0$$

$$\tilde{\mathbb{X}}^T \tilde{y} - \tilde{\mathbb{X}}^T \tilde{\mathbb{X}}\beta = 0$$

$$\beta = (\tilde{\mathbb{X}}^T \tilde{\mathbb{X}})^{-1} \tilde{\mathbb{X}}^T \tilde{y}$$

Where $\bar{y}$ and $\bar{\mathbb{X}}$ are the column means, and $\tilde{\mathbb{X}} = \mathbb{X} - \bar{\mathbb{X}}$ and $\tilde{y} = y - \bar{y} = \tilde{\mathbb{X}}\beta + \varepsilon$ are the de-meaned variables,

The matrix $\mathbb{C} = \tilde{\mathbb{X}}^T \tilde{\mathbb{X}}/(n-1)$ is the covariance matrix of the matrix $\mathbb{X}$, and it's invertible only if the columns of $\mathbb{X}$ are linearly independent,

The residuals have zero mean: $\varepsilon^T \mathbb{1} = 0$, and they are orthogonal to the explanatory variables: $\varepsilon^T \mathbb{X} = 0$, and variance equal to: $\mathbb{E}[\varepsilon\varepsilon^T] = {\sigma_\varepsilon}^2 \mathbb{1}$

```
> # calculate de-meaned explanatory matrix
> explain_zm <- t(t(ex_plain) - colMeans(ex_plai
> # explain_zm <- apply(explain_zm, 2, function(
> # calculate de-meaned response vector
> response_zm <- res_ponse - mean(res_ponse)
> # solve for regression betas
> beta_s <- MASS::ginv(explain_zm) %*% response_
> # solve for regression alpha
> al_pha <- mean(res_ponse) -
+   sum(colSums(ex_plain)*drop(beta_s))/len_gth
> # multivariate regression using lm()
> reg_model <- lm(res_ponse ~ ex_plain)
> coef(reg_model)
(Intercept)   ex_plain1   ex_plain2   ex_plain3
     -2.893       0.292      -0.208      -1.258
  ex_plain5
     -0.221
> c(al_pha, beta_s)
[1] -2.893  0.292 -0.208 -1.258  0.202 -0.221
> c(-3, weight_s)
[1] -3.000  0.455 -0.232 -1.345  0.344 -0.424
> sum(reg_model$residuals * reg_model$fitted.val
[1] -3.34e-14
```

The residuals are also orthogonal to the *fitted values*: $\varepsilon^T \hat{y} = 0$,

Total Sum of Squares and *Explained Sum of Squares*

The *Total Sum of Squares* (*TSS*) and the *Explained Sum of Squares* (*ESS*) are defined as:

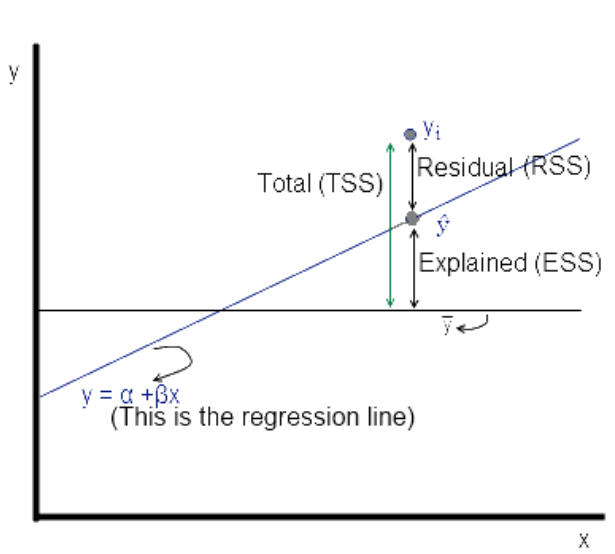
$$TSS = (y - \bar{y})^T (y - \bar{y})$$
$$ESS = (\hat{y} - \bar{y})^T (\hat{y} - \bar{y})$$
$$RSS = (y - \hat{y})^T (y - \hat{y})$$

Since the residuals $\varepsilon = y - \hat{y}$ are orthogonal to the *fitted values* $\hat{y}$, they are also orthogonal to the *fitted* excess values:

$$(y - \hat{y})^T (\hat{y} - \bar{y}) = 0$$

Therefore the *TSS* can be expressed as the sum of the *ESS* plus the *RSS*:

$$TSS = ESS + RSS$$



$\hat{y}$ is the predicted value of y given x, using the equation $y = \alpha + \beta x$.

y_i is the actual observed value of y

$\bar{y}$ is the mean of y.

The distances that RSS, ESS and TSS represent are shown in the diagram to the left - but remember that the actua calculations are squares of these distances.

$$TSS = \Sigma (y_i - \bar{y})^2$$
$$RSS = \Sigma (y_i - \hat{y})^2$$
$$ESS = \Sigma (\hat{y} - \bar{y})^2$$

```
> # calculate fitted values
> fit_ted <- drop(al_pha + ex_plain %*% beta_s)
> all.equal(fit_ted, reg_model$fitted.values, cl
[1] TRUE
> # calculate residuals
> resid_uals <- drop(res_ponse - fit_ted)
> all.equal(resid_uals, reg_model$residuals, check.attributes=FALSE)
[1] TRUE
```

```
> # residuals are orthogonal to fitted values
> sum(resid_uals * fit_ted)
[1] -1.19e-13
> # TSS = ESS + RSS
> t_ss <- (len_gth-1)*var(drop(res_ponse))
> e_ss <- (len_gth-1)*var(fit_ted)
> r_ss <- (len_gth-1)*var(resid_uals)
> all.equal(t_ss, e_ss + r_ss)
[1] TRUE
```

R-squared of Multivariate *Linear Regression*

The *R-squared* is the fraction of the response variance (TSS) that is explained by the model (ESS):

$$R^2 = \frac{ESS}{TSS} = 1 - \frac{RSS}{TSS}$$

The *R-squared* is a measure of the model *goodness of fit*, with *R-squared* close to 1 for models fitting the data very well, and *R-squared* close to 0 for poorly fitting models,

The *R-squared* is equal to the squared correlation between the response and the *fitted values*:

$$\rho_{y\hat{y}} = \frac{(\hat{y} - \bar{y})^T (y - \bar{y})}{\sqrt{TSS \cdot ESS}} = \frac{(\hat{y} - \bar{y})^T (\hat{y} - \bar{y})}{\sqrt{TSS \cdot ESS}} = \sqrt{\frac{ESS}{TSS}}$$

```
> # regression summary
> reg_model_sum <- summary(reg_model)
> # regression R-squared
> r_squared <- e_ss/t_ss
> all.equal(r_squared, reg_model_sum$r.squared)
[1] TRUE
> # correlation between response and fitted valu
> cor_fitted <- drop(cor(res_ponse, fit_ted))
> # squared correlation between response and fit
> all.equal(cor_fitted^2, r_squared)
[1] TRUE
```

Fisher's *F-distribution*

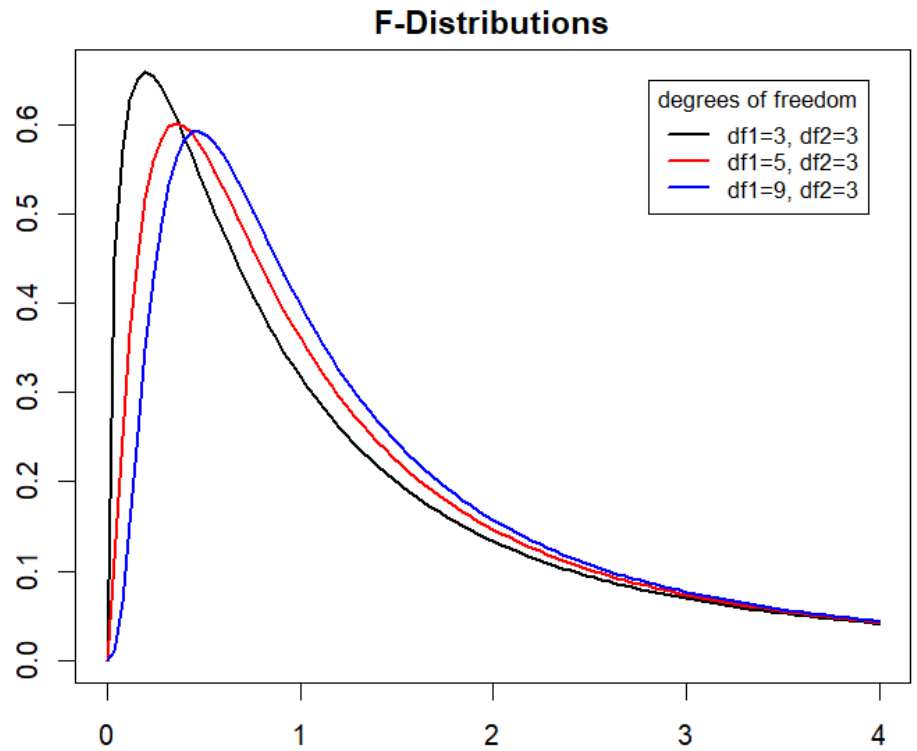
Let χ^2_m and χ^2_n be independent random variables following *Chi-squared* distributions with `m` and `n` degrees of freedom,

Then the random variable:

$$F = \frac{m\,\chi^2_n}{n\,\chi^2_m}$$

Follows the *F-distribution* with `m` and `n` degrees of freedom, with the probability density function:

$$P(F) = \frac{\Gamma((m+n)/2)m^{m/2}n^{n/2}}{\Gamma(m/2)\Gamma(n/2)}\,\frac{F^{n/2-1}}{(m+nF)^{(m+n)/2}}$$




```
> deg_free <- c(3, 5, 9)  # df values
> col_ors <- c("black", "red", "blue", "green")
> lab_els <- paste0("df1=", deg_free, ", df2=3")
> for (in_dex in 1:NROW(deg_free)) {  # plot fo
+ curve(expr=df(x, df1=deg_free[in_dex], df2=3)
+       type="l", xlim=c(0, 4),
+       xlab="", ylab="", lwd=2,
+       col=col_ors[in_dex],
+       add=as.logical(in_dex-1))
+ }  # end for
```

```
> # add title
> title(main="F-Distributions", line=0.5)
> # add legend
> legend("topright", inset=0.05, title="degrees
+        lab_els, cex=0.8, lwd=2, lty=1,
+        col=col_ors)
```

Fisher's *F-test* of Model Significance

The *F-test* can be used to determine if an *unrestricted* model with more parameters is able explain the variance of the *responses* better than a *restricted* model with fewer parameters,

Let y_i be a vector of `n` observations (*responses*), and x_{ij} be a (n, k)-dimensional matrix of explanatory variables,

Let $\hat{y}_i$ be a vector of *fitted values* of the model, using the matrix of explanatory variables x_{ij},

Let the *restricted* model have p_1 parameters, so its *fitted values* have $df_1 = n - p_1$ degrees of freedom, and the *unrestricted* model have p_2 parameters, so its *fitted values* have $df_2 = n - p_2$ degrees of freedom, with $p_1 < p_2$,

Then the *F*-statistic, defined as the ratio:

$$F = \frac{(RSS_1 - RSS_2)/(df_1 - df_2)}{RSS_2/df_2}$$

Follows the *F-distribution* with $(p_2 - p_1)$ and $(n - p_2)$ degrees of freedom (assuming that the residuals are normally distributed),

If the *restricted* regression model has *zero* parameters, then the *fitted values* are all simply equal to the average of the *responses*: $\hat{y}_i = \bar{y}$, with $df_1 = n - 1$, and its *Residual Sum of Squares* is equal to $TSS = (y - \bar{y})^2$,

If the *unrestricted* model has k parameters, then its *Residual Sum of Squares* is equal to $RSS = (y - \hat{y})^2$, with $df_2 = n - k - 1$, and the *F*-statistic is equal to:

$$F = \frac{ESS/k}{RSS/(n - k - 1)}$$

```
> # F-statistic from lm()
> reg_model_sum$fstatistic
value numdf dendf
 37.5   5.0  94.0
> # degrees of freedom of residuals
> deg_free <- len_gth-n_var-1
> # F-statistic from RSS
> f_stat <- e_ss*deg_free/r_ss/n_var
> all.equal(f_stat, reg_model_sum$fstatistic[1],
[1] TRUE
> # p-value of F-statistic
> 1-pf(q=f_stat, df1=len_gth-n_var-1, df2=n_var)
[1] 0.000342
```

Multivariate *Linear Regression* in Homogeneous Form

We can add an extra unit column to the explanatory matrix $\mathbb{X}$ to represent the intercept term, and express the *linear regression* formula in *homogeneous form*:

$$y = \mathbb{X}\beta + \varepsilon$$

Where the β regression coefficients now contain the intercept: $\beta = (\alpha, \beta_1, \ldots, \beta_k)$,

The *OLS* solution for the β coefficients is found by equating the *RSS* derivative to zero:

$$RSS_\beta = -2(y - \mathbb{X}\beta)^T\mathbb{X} = 0$$

$$\mathbb{X}^T y - \mathbb{X}^T\mathbb{X}\beta = 0$$

$$\beta = (\mathbb{X}^T\mathbb{X})^{-1}\mathbb{X}^T y$$

The vector $\hat{y} = \mathbb{X}\beta$ are the *fitted values* of the *responses* in the regression model,

The residuals $\varepsilon = y - \hat{y}$ are orthogonal to the explanatory variables: $\varepsilon^T\mathbb{X} = 0$, and they have zero mean: $\varepsilon^T\mathbb{X}[, 1] = 0$,

```
> # add intercept column to explanatory
> ex_plain <- cbind(rep(1, NROW(ex_plain)), ex_p
> # solve for regression betas
> beta_s <- MASS::ginv(ex_plain) %*% res_ponse
> all.equal(drop(beta_s), coef(reg_model), check
[1] TRUE
> # calculate fitted values
> fit_ted <- drop(ex_plain %*% beta_s)
> all.equal(fit_ted, reg_model$fitted.values, ch
[1] TRUE
> # calculate residuals
> resid_uals <- drop(res_ponse - fit_ted)
> all.equal(resid_uals, reg_model$residuals, che
[1] TRUE
```

The matrix $(\mathbb{X}^T\mathbb{X})^{-1}\mathbb{X}^T$ is the projection matrix onto the explanatory matrix $\mathbb{X}$, and the β coefficients can be interpreted as the coefficients of the projections onto the columns of matrix $\mathbb{X}$,

Standard Errors of Multivariate *Linear Regression*

The standard errors of the regression *betas* are due to the residuals as the source of error:

$$\sigma_\beta{}^2 = \mathbb{E}[(\beta(y) - \beta)(\beta(y) - \beta)^T] =$$
$$\mathbb{E}[(\beta(y) - \beta)^2] = \mathbb{E}[((\mathbb{X}^T\mathbb{X})^{-1}\mathbb{X}^T\tilde{y} - \beta)^2] =$$
$$\mathbb{E}[((\mathbb{X}^T\mathbb{X})^{-1}\mathbb{X}^T(\mathbb{X}\beta + \varepsilon) - \beta)^2] =$$
$$\mathbb{E}[((\mathbb{X}^T\mathbb{X})^{-1}\mathbb{X}^T\varepsilon)^2] =$$
$$\mathbb{E}[((\mathbb{X}^T\mathbb{X})^{-1}\mathbb{X}^T\varepsilon)((\mathbb{X}^T\mathbb{X})^{-1}\mathbb{X}^T\varepsilon)^T] =$$
$$(\mathbb{X}^T\mathbb{X})^{-1}\mathbb{X}^T\mathbb{E}[\varepsilon\varepsilon^T]\,\mathbb{X}(\mathbb{X}^T\mathbb{X})^{-1} =$$
$$(\mathbb{X}^T\mathbb{X})^{-1}\mathbb{X}^T\sigma_\varepsilon{}^2\mathbb{1}\,\mathbb{X}(\mathbb{X}^T\mathbb{X})^{-1} = \sigma_\varepsilon{}^2(\mathbb{X}^T\mathbb{X})^{-1}$$

The key assumption is that the residuals are normally distributed, independent, and stationary, with variance equal to: $\mathbb{E}[\varepsilon\varepsilon^T] = \sigma_\varepsilon{}^2\mathbb{1}$

The matrix $\mathbb{C} = \tilde{\mathbb{X}}^T\tilde{\mathbb{X}}/(n-1)$ is the covariance matrix of the matrix $\mathbb{X}$, and it's invertible only if the columns of $\mathbb{X}$ are linearly independent,

```
> # degrees of freedom of residuals
> deg_free <- len_gth-NCOL(ex_plain)
> # variance of residuals
> resid_var <- sum(resid_uals^2)/deg_free
> # explanatory matrix squared
> explain_squared <- crossprod(ex_plain)
> # explain_squared <- t(ex_plain) %*% ex_plain
> # calculate covariance matrix of betas
> beta_covar <- resid_var*MASS::ginv(explain_squ
> beta_sd <- sqrt(diag(beta_covar))
> all.equal(beta_sd, reg_model_sum$coeff[, 2], c
[1] TRUE
> # calculate t-values of betas
> beta_tvals <- drop(beta_s)/beta_sd
> all.equal(beta_tvals, reg_model_sum$coeff[, 3]
[1] TRUE
> # calculate two-sided p-values of betas
> beta_pvals <- 2*pt(-abs(beta_tvals), df=deg_fr
> all.equal(beta_pvals, reg_model_sum$coeff[, 4]
[1] TRUE
```

Omitted Variable Bias

Omitted Variable Bias occurs in a regression model that omits important predictors,

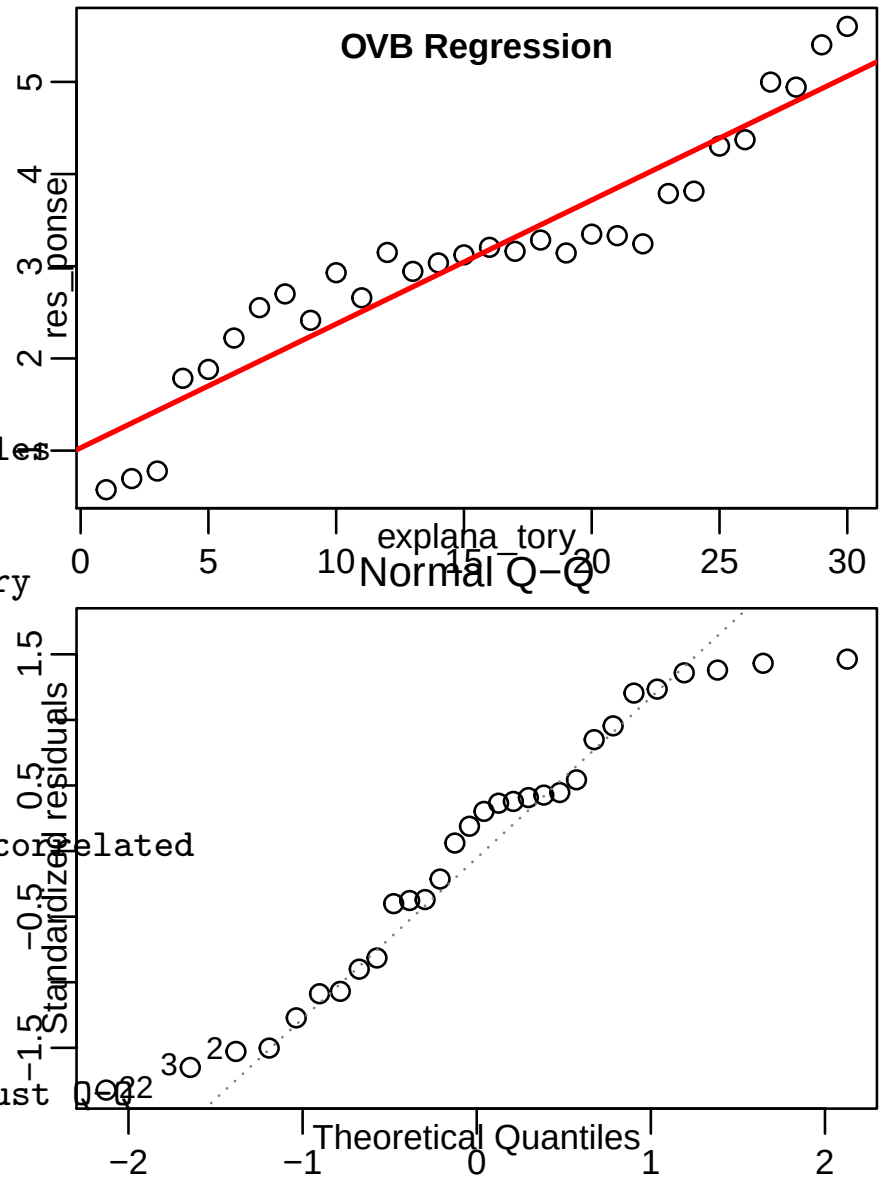
The parameter estimates are biased, even though the *t*-statistics, *p*-values, and *R*-squared all indicate a statistically significant regression,

But the Durbin-Watson test shows residuals are autocorrelated, invalidating other tests,

```
> de_sign <- data.frame(  # design matrix
+   ex_plain=1:30, omit_var=sin(0.2*1:30))
> # response depends on both explanatory variables
> res_ponse <- with(de_sign,
+   0.2*ex_plain + omit_var + 0.2*rnorm(30))
> # mis-specified regression only one explanatory
> reg_model <- lm(res_ponse ~ ex_plain,
+         data=de_sign)
> reg_model_sum <- summary(reg_model)
> reg_model_sum$coeff
> reg_model_sum$r.squared
> # Durbin-Watson test shows residuals are autocorrelated
> dwtest(reg_model)$p.value
```

```
> plot(reg_formula, data=de_sign)
> abline(reg_model, lwd=2, col="red")
> title(main="OVB Regression", line=-1)
> plot(reg_model, which=2, ask=FALSE)  # plot just Q-Q
```

lm(res_ponse ~ explana_tory)



Spurious Time Series Regression

Regression of non-stationary time series creates *spurious* regressions,

The *t*-statistics, *p*-values, and *R*-squared all indicate a statistically significant regression,

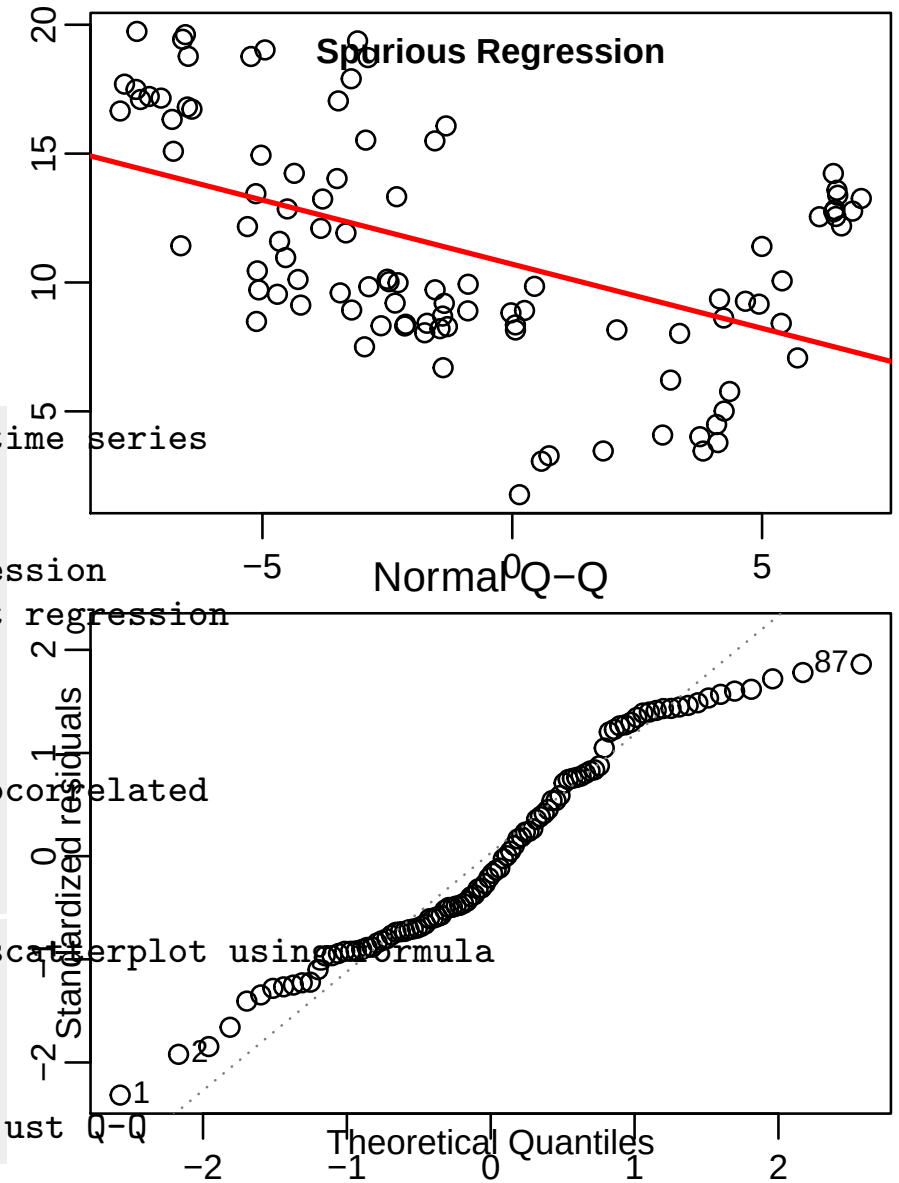
But the Durbin-Watson test shows residuals are autocorrelated, which invalidates the other tests,

The Q-Q plot also shows that residuals are *not* normally distributed,

```
> ex_plain <- cumsum(rnorm(100))  # unit root time series
> res_ponse <- cumsum(rnorm(100))
> reg_formula <- res_ponse ~ ex_plain
> reg_model <- lm(reg_formula)  # perform regression
> # summary indicates statistically significant regression
> reg_model_sum <- summary(reg_model)
> reg_model_sum$coeff
> reg_model_sum$r.squared
> # Durbin-Watson test shows residuals are autocorrelated
> dw_test <- dwtest(reg_model)
> c(dw_test$statistic[[1]], dw_test$p.value)
```

```
> plot(reg_formula, xlab="", ylab="")  # plot scatterplot using formula
> title(main="Spurious Regression", line=-1)
> # add regression line
> abline(reg_model, lwd=2, col="red")
> plot(reg_model, which=2, ask=FALSE)  # plot just Q-Q
```

lm(reg_formula)

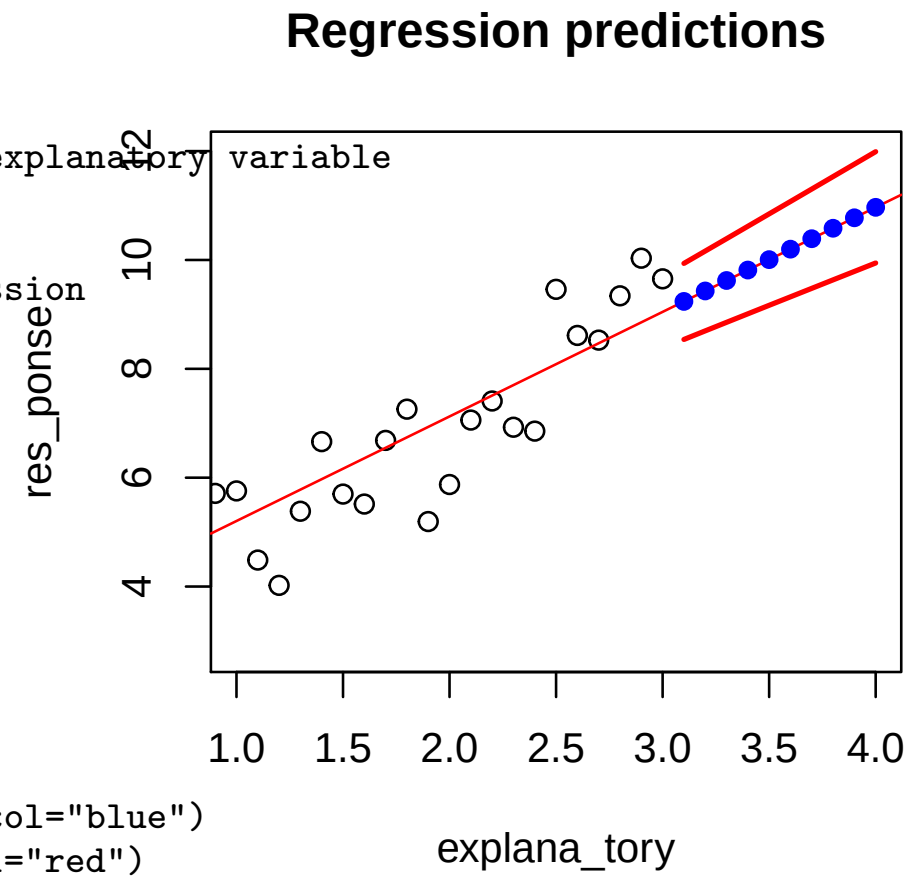


Predictions From *Linear Regression* Models

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

`predict.lm()` is the predict method for linear models (regressions),

```
> ex_plain <- seq(from=0.1, to=3.0, by=0.1)  # explanatory variable
> res_ponse <- 3 + 2*ex_plain + rnorm(30)
> reg_formula <- res_ponse ~ ex_plain
> reg_model <- lm(reg_formula)  # perform regression
> new_data <- data.frame(ex_plain=0.1*31:40)
> predict_lm <- predict(object=reg_model,
+                newdata=new_data, level=0.95,
+                interval="confidence")
> predict_lm <- as.data.frame(predict_lm)
> head(predict_lm, 2)
> plot(reg_formula, xlim=c(1.0, 4.0),
+      ylim=range(res_ponse, predict_lm),
+      main="Regression predictions")
> abline(reg_model, col="red")
> with(predict_lm, {
+   points(x=new_data$ex_plain, y=fit, pch=16, col="blue")
+   lines(x=new_data$ex_plain, y=lwr, lwd=2, col="red")
+   lines(x=new_data$ex_plain, y=upr, lwd=2, col="red")
+ })  # end with
```



Homework Assignment

Required

- Read all the lecture slides in *FRE7241_Lecture_3.pdf*, and run all the code in *FRE7241_Lecture_3.R*