FRE7241 Algorithmic Portfolio Management

Lecture#4, Fall 2022

Jerzy Pawlowski *jp3900@nyu.edu*

NYU Tandon School of Engineering

September 27, 2022

Centered Price Z-scores

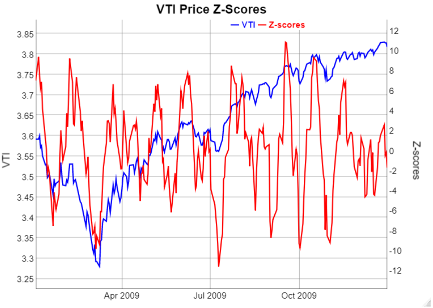
An extreme local price is a price which differs significantly from neighboring prices.

Extreme prices can be identified in-sample using the centered *price z-score* equal to the price difference with neighboring prices divided by the volatility of returns σ_i:

$$z_i = \frac{2 p_i - p_{i-k} - p_{i+k}}{\sigma_i}$$

Where p_{i-k} and p_{i+k} are the lagged and advanced prices.

The lag parameter k determines the scale of the extreme local prices, with smaller k producing larger z-scores for more local price extremes.



```
> # Extract VTI log OHLC prices
> ohlc <- log(rutils::etfenv$VTI)
> nrows <- NROW(ohlc)
> closep <- quantmod::Cl(ohlc)
> retsp <- rutils::diffit(closep)
> # Calculate the centered volatility
> look_back <- 7
> half_back <- look_back %/% 2
> stdev <- roll::roll_sd(retsp, width=look_back, min_obs=1)
> stdev <- rutils::lagit(stdev, lagg=(-half_back))
> # Calculate the z-scores of prices
> pricez <- (2*closep -
+   rutils::lagit(closep, half_back, pad_zeros=FALSE) -
+   rutils::lagit(closep, -half_back, pad_zeros=FALSE))
> pricez <- ifelse(stdev > 0, pricez/stdev, 0)
```

```
> # Plot dygraph of z-scores of VTI prices
> pricets <- cbind(closep, pricez)
> colnames(pricets) <- c("VTI", "Z-scores")
> colnamev <- colnames(pricets)
> dygraphs::dygraph(pricets["2009"], main="VTI Price Z-Scores") %>%
+   dyAxis("y", label=colnamev[1], independentTicks=TRUE) %>%
+   dyAxis("y2", label=colnamev[2], independentTicks=TRUE) %>%
+   dySeries(name=colnamev[1], axis="y", label=colnamev[1], strokeWi
+   dySeries(name=colnamev[2], axis="y2", label=colnamev[2], strokeW
```

Labeling the Tops and Bottoms of Prices

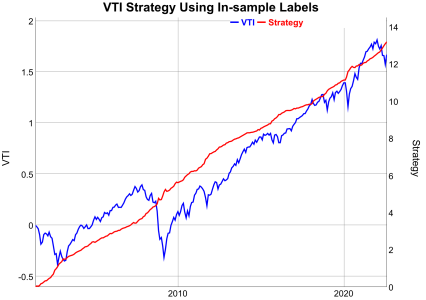
The local tops and bottoms of prices can be labeled approximately in-sample using the z-scores of prices and threshold values.

The local tops of prices represent *overbought* conditions, while the bottoms represent *oversold* conditions.

The labeled data can be used as a response or target variable in machine learning classifier models.

But it's not feasible to classify the prices out-of-sample exactly according to their in-sample labels.

```
> # Calculate thresholds for labeling tops and bottoms
> confl <- c(0.2, 0.8)
> threshv <- quantile(pricez, confl)
> # Calculate the vectors of tops and bottoms
> tops <- zoo::coredata(pricez > threshv[2])
> colnames(tops) <- "tops"
> bottoms <- zoo::coredata(pricez < threshv[1])
> colnames(bottoms) <- "bottoms"
> # Simulate in-sample VTI strategy
> posit <- rep(NA_integer_, nrows)
> posit[1] <- 0
> posit[tops] <- (-1)
> posit[bottoms] <- 1
> posit <- zoo::na.locf(posit)
> posit <- rutils::lagit(posit)
> pnls <- retsp*posit
```


VTI Strategy Using In-sample Labels

```
> # Plot dygraph of in-sample VTI strategy
> wealthv <- cbind(retsp, pnls)
> colnames(wealthv) <- c("VTI", "Strategy")
> endp <- rutils::calc_endpoints(wealthv, interval="months")
> dygraphs::dygraph(cumsum(wealthv)[endp],
+   main="VTI Strategy Using In-sample Labels") %>%
+   dyAxis("y", label="VTI", independentTicks=TRUE) %>%
+   dyAxis("y2", label="Strategy", independentTicks=TRUE) %>%
+   dySeries(name="VTI", axis="y", label="VTI", strokeWidth=2, col="
+   dySeries(name="Strategy", axis="y2", label="Strategy", strokeWi
```

Predictors of Price Extremes

The return volatility and trading volumes may be used as predictors in a classification model, in order to identify *overbought* and *oversold* conditions.

The trailing *volume z-score* is equal to the volume v_i minus the trailing average volumes $\bar{v}_i$ divided by the volatility of the volumes σ_i:

$$z_i = \frac{v_i - \bar{v}_i}{\sigma_i}$$

Trading volumes are typically higher when prices drop and they are also positively correlated with the return volatility.

The *volatility z-score* is equal to the spot volatility v_i minus the trailing average volatility $\bar{v}_i$ divided by the standard deviation of the volatility σ_i:

$$z_i = \frac{v_i - \bar{v}_i}{\sigma_i}$$

Volatility is typically higher when prices drop and it's also positively correlated with the trading volumes.

```
> # Calculate volatility z-scores
> volat <- HighFreq::roll_var_ohlc(ohlc=ohlc, look_back=look_back, s
> meanv <- roll::roll_mean(volat, width=look_back, min_obs=1)
> stdev <- roll::roll_sd(rutils::diffit(volat), width=look_back, mi
> stdev[1] <- 0
> volatz <- ifelse(stdev > 0, (volat - meanv)/stdev, 0)
> colnames(volatz) <- "volat"
> # Calculate volume z-scores
> volumes <- quantmod::Vo(ohlc)
> meanv <- roll::roll_mean(volumes, width=look_back, min_obs=1)
> stdev <- roll::roll_sd(rutils::diffit(volumes), width=look_back, m
> stdev[1] <- 0
> volumez <- ifelse(stdev > 0, (volumes - meanv)/stdev, 0)
> colnames(volumez) <- "volume"
```

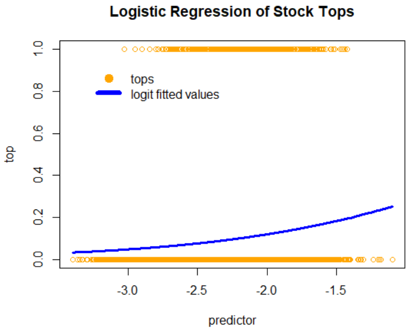
Forecasting Stock Price Tops and Bottoms Using Logistic Regression

The weighted average of the volatility and trading volume z-scores can be used to forecast a stock top (overbought condition) or a bottom (oversold condition).

The residuals are the differences between the actual response values (0 and 1), and the calculated probabilities of default.

The residuals are not normally distributed, so the data is fitted using the *maximum likelihood* method, instead of least squares.

```
> # Define design matrix for tops including intercept column
> predictor <- cbind(volatz, volumez)
> predictor[1, ] <- 0
> predictor <- rutils::lagit(predictor)
> # Fit in-sample logistic regression for tops
> logmod <- glm(tops ~ predictor, family=binomial(logit))
> summary(logmod)
> coeff <- logmod$coefficients
> forecastv <- drop(cbind(rep(1, nrows), predictor) %*% coeff)
> ordern <- order(forecastv)
> # Calculate in-sample forecasts from logistic regression model
> forecastv <- 1/(1+exp(-forecastv))
> all.equal(logmod$fitted.values, forecastv, check.attributes=FALS
> hist(forecastv)
```



```
> x11(width=6, height=5)
> plot(x=forecastv[ordern], y=tops[ordern],
+      main="Logistic Regression of Stock Tops",
+      col="orange", xlab="predictor", ylab="top")
> lines(x=forecastv[ordern], y=logmod$fitted.values[ordern], col="bl
> legend(x="topleft", inset=0.1, bty="n", lwd=6,
+  legend=c("tops", "logit fitted values"),
+  col=c("orange", "blue"), lty=c(NA, 1), pch=c(1, NA))
```

Forecasting Errors of Stock Tops and Bottoms

A *binary classification model* categorizes cases based on its forecasts whether the *null hypothesis* is TRUE or FALSE.

Let the *null hypothesis* be that the data point is not a top: tops = FALSE.

A *positive* result corresponds to rejecting the null hypothesis (tops = TRUE), while a *negative* result corresponds to accepting the null hypothesis (tops = FALSE).

The forecasts are subject to two different types of errors: *type I* and *type II* errors.

A *type I* error is the incorrect rejection of a TRUE *null hypothesis* (i.e. a "false positive"), when tops = FALSE but it's classified as tops = TRUE.

A *type II* error is the incorrect acceptance of a FALSE *null hypothesis* (i.e. a "false negative"), when tops = TRUE but it's classified as tops = FALSE.

```
> # Define discrimination threshold value
> threshold <- quantile(forecastv, confl[2])
> # Calculate confusion matrix in-sample
> confmat <- table(actual=!tops, forecast=(forecastv < threshold))
> confmat
> # Calculate FALSE positive (type I error)
> sum(tops & (forecastv < threshold))
> # Calculate FALSE negative (type II error)
> sum(!tops & (forecastv > threshold))
```

The Confusion Matrix of a Binary Classification Model

The confusion matrix summarizes the performance of a classification model on a set of test data for which the actual values of the *null hypothesis* are known.

	Forecast	
Actual	**Null is FALSE**	**Null is TRUE**
Null is FALSE	True Positive (sensitivity)	False Negative (type II error)
Null is TRUE	False Positive (type I error)	True Negative (specificity)

```
> # Calculate FALSE positive and FALSE negative rates
> confmat <- confmat / rowSums(confmat)
> c(typeI=confmat[2, 1], typeII=confmat[1, 2])
```

Let the *null hypothesis* be that the data point is not a top: `tops = FALSE`.

The *true positive* rate (known as the *sensitivity*) is the fraction of `FALSE` *null hypothesis* cases that are correctly classified as `FALSE`.

The *false negative* rate is the fraction of `FALSE` *null hypothesis* cases that are incorrectly classified as `TRUE` (*type II* error).

The sum of the *true positive* plus the *false negative* rate is equal to 1.

The *true negative* rate (known as the *specificity*) is the fraction of `TRUE` *null hypothesis* cases that are correctly classified as `TRUE`.

The *false positive* rate is the fraction of `TRUE` *null hypothesis* cases that are incorrectly classified as `FALSE` (*type I* error).

The sum of the *true negative* plus the *false positive* rate is equal to 1.

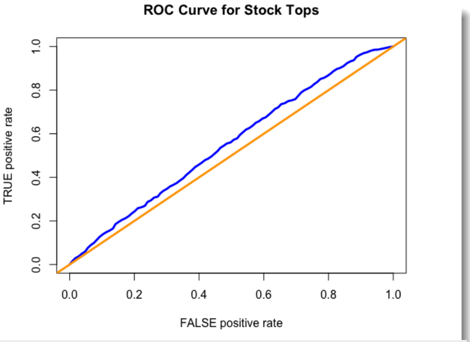
Receiver Operating Characteristic (ROC) Curve for Stock Tops

The *ROC curve* is the plot of the *true positive* rate, as a function of the *false positive* rate, and illustrates the performance of a binary classifier.

The area under the *ROC curve* (AUC) measures the classification ability of a binary classifier.

The *informedness* is equal to the sum of the sensitivity plus the specificity, and measures the performance of a binary classification model.

```
> # Confusion matrix as function of threshold
> confun <- function(actual, forecastv, threshold) {
+     conf <- table(actual, (forecastv < threshold))
+     conf <- conf / rowSums(conf)
+     c(typeI=conf[2, 1], typeII=conf[1, 2])
+   }  # end confun
> confun(!tops, forecastv, threshold=threshold)
> # Define vector of discrimination thresholds
> threshv <- quantile(forecastv, seq(0.01, 0.99, by=0.01))
> # Calculate error rates
> error_rates <- sapply(threshv, confun,
+   actual=!tops, forecastv=forecastv)  # end sapply
> error_rates <- t(error_rates)
> rownames(error_rates) <- threshv
> # Calculate the informedness
> informv <- 2 - rowSums(error_rates)
> plot(threshv, informv, t="l", main="Informedness")
> # Find the threshold corresponding to highest informedness
> threshm <- threshv[which.max(informv)]
> forecastops <- (forecastv > threshm)
```



```
> # Calculate area under ROC curve (AUC)
> error_rates <- rbind(c(1, 0), error_rates)
> error_rates <- rbind(error_rates, c(0, 1))
> truepos <- (1 - error_rates[, "typeII"])
> truepos <- (truepos + rutils::lagit(truepos))/2
> falsepos <- rutils::diffit(error_rates[, "typeI"])
> abs(sum(truepos*falsepos))
> # Plot ROC Curve for stock tops
> plot(x=error_rates[, "typeI"], y=1-error_rates[, "typeII"],
+      xlab="FALSE positive rate", ylab="TRUE positive rate",
+      main="ROC Curve for Stock Tops", type="l", lwd=3, col="blue")
> abline(a=0.0, b=1.0, lwd=3, col="orange")
```

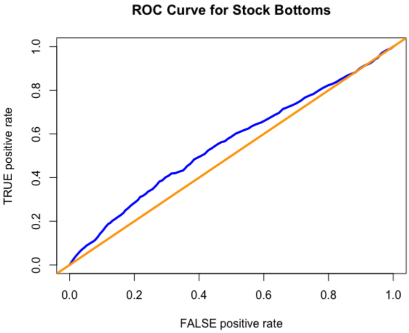
Receiver Operating Characteristic (ROC) Curve for Stock Bottoms

The *ROC curve* is the plot of the *true positive* rate, as a function of the *false positive* rate, and illustrates the performance of a binary classifier.

The area under the *ROC curve* (AUC) measures the classification ability of a binary classifier.

The *informedness* is equal to the sum of the sensitivity plus the specificity, and measures the performance of a binary classification model.

```
> # Fit in-sample logistic regression for bottoms
> logmod <- glm(bottoms ~ predictor, family=binomial(logit))
> summary(logmod)
> # Calculate in-sample forecast from logistic regression model
> coeff <- logmod$coefficients
> forecastv <- drop(cbind(rep(1, nrows), predictor) %*% coeff)
> forecastv <- 1/(1+exp(-forecastv))
> # Calculate error rates
> error_rates <- sapply(threshv, confun,
+   actual=!bottoms, forecastv=forecastv)  # end sapply
> error_rates <- t(error_rates)
> rownames(error_rates) <- threshv
> # Calculate the informedness
> informv <- 2 - rowSums(error_rates)
> plot(threshv, informv, t="l", main="Informedness")
> # Find the threshold corresponding to highest informedness
> threshm <- threshv[which.max(informv)]
> forecastbot <- (forecastv > threshm)
```

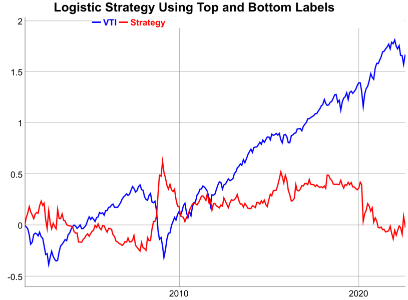


```
> # Calculate area under ROC curve (AUC)
> error_rates <- rbind(c(1, 0), error_rates)
> error_rates <- rbind(error_rates, c(0, 1))
> truepos <- (1 - error_rates[, "typeII"])
> truepos <- (truepos + rutils::lagit(truepos))/2
> falsepos <- rutils::diffit(error_rates[, "typeI"])
> abs(sum(truepos*falsepos))
> # Plot ROC Curve for stock tops
> plot(x=error_rates[, "typeI"], y=1-error_rates[, "typeII"],
+      xlab="FALSE positive rate", ylab="TRUE positive rate",
+      main="ROC Curve for Stock Bottoms", type="l", lwd=3, col="blu
> abline(a=0.0, b=1.0, lwd=3, col="orange")
```

Logistic Tops and Bottoms Strategy In-sample

The logistic strategy forecasts the tops and bottoms of prices, using a logistic regression model with the volatility and trading volumes as predictors.

```
> # Simulate in-sample VTI strategy
> posit <- rep(NA_integer_, NROW(retsp))
> posit[1] <- 0
> posit[forecastops] <- (-1)
> posit[forecastbot] <- 1
> posit <- zoo::na.locf(posit)
> posit <- rutils::lagit(posit)
> pnls <- retsp*posit
```

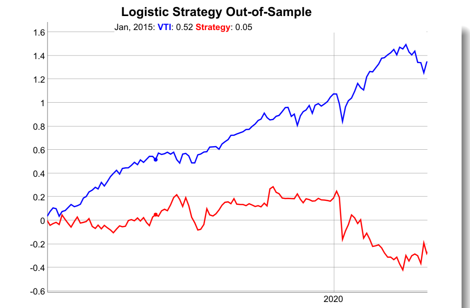


```
> # Calculate the Sharpe and Sortino ratios
> wealthv <- cbind(retsp, pnls)
> colnames(wealthv) <- c("VTI", "Strategy")
> sqrt(252)*sapply(wealthv,
+   function(x) c(Sharpe=mean(x)/sd(x), Sortino=mean(x)/sd(x[x<0])))
> # Plot dygraph of in-sample VTI strategy
> endp <- rutils::calc_endpoints(wealthv, interval="months")
> dygraphs::dygraph(cumsum(wealthv)[endp],
+   main="Logistic Strategy Using Top and Bottom Labels") %>%
+   dyOptions(colors=c("blue", "red"), strokeWidth=2) %>%
+   dyLegend(show="always", width=500)
```

Logistic Tops and Bottoms Strategy Out-of-Sample

The logistic strategy forecasts the tops and bottoms of prices, using a logistic regression model with the volatility and trading volumes as predictors.

```
> insample <- 1:(nrows %/% 2)
> outsample <- (nrows %/% 2 + 1):nrows
> # Fit in-sample logistic regression for tops
> logmod <- glm(tops[insample] ~ predictor[insample, ], family=binor
> fittedv <- logmod$fitted.values
> coefftop <- logmod$coefficients
> # Calculate error rates and best threshold value
> error_rates <- sapply(threshv, confun,
+   actual=!tops[insample], forecastv=fittedv)  # end sapply
> error_rates <- t(error_rates)
> informv <- 2 - rowSums(error_rates)
> threshtop <- threshv[which.max(informv)]
> # Fit in-sample logistic regression for bottoms
> logmod <- glm(bottoms[insample] ~ predictor[insample, ], family=binomial(logit))
> fittedv <- logmod$fitted.values
> coeffbot <- logmod$coefficients
> # Calculate error rates and best threshold value
> error_rates <- sapply(threshv, confun,
+   actual=!bottoms[insample], forecastv=fittedv)  # end sapply
> error_rates <- t(error_rates)
> informv <- 2 - rowSums(error_rates)
> threshbot <- threshv[which.max(informv)]
> # Calculate out-of-sample forecasts from logistic regression mod
> predictout <- cbind(rep(1, NROW(outsample)), predictor[outsample
> forecastv <- drop(predictout %*% coefftop)
> forecastv <- 1/(1+exp(-forecastv))
> forecastops <- (forecastv > threshtop)
> forecastv <- drop(predictout %*% coeffbot)
> forecastv <- 1/(1+exp(-forecastv))
> forecastbot <- (forecastv > threshbot)
```



```
> # Simulate out-of-sample VTI strategy
> posit <- rep(NA_integer_, NROW(outsample))
> posit[1] <- 0
> posit[forecastops] <- (-1)
> posit[forecastbot] <- 1
> posit <- zoo::na.locf(posit)
> posit <- rutils::lagit(posit)
> pnls <- retsp[outsample, ]*posit
> # Calculate the Sharpe and Sortino ratios
> wealthv <- cbind(retsp[outsample, ], pnls)
> colnames(wealthv) <- c("VTI", "Strategy")
> sqrt(252)*sapply(wealthv,
+   function(x) c(Sharpe=mean(x)/sd(x), Sortino=mean(x)/sd(x[x<0])))
> # Plot dygraph of in-sample VTI strategy
> endp <- rutils::calc_endpoints(wealthv, interval="months")
> dygraphs::dygraph(cumsum(wealthv)[endp],
+   main="Logistic Strategy Out-of-Sample") %>%
+   dyOptions(colors=c("blue", "red"), strokeWidth=2) %>%
```

draft: Forecasting Stock Tops and Bottoms Out-of-Sample

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

The method `predict.glm()` produces forecasts for a generalized linear (*glm*) model, in the form of `numeric` probabilities, not the `Boolean` response variable.

The `Boolean` forecasts are obtained by comparing the *forecast probabilities* with a *discrimination threshold*.

Let the *null hypothesis* be that the data point is not a top: `tops = FALSE`.

If the *forecast probability* is greater than the *discrimination threshold*, then the forecast is that the data point is not a top and that the *null hypothesis* is `TRUE`.

The *in-sample forecasts* are just the *fitted values* of the *glm* model.

```
> # Fit logistic regression over training data
> set.seed(1121)  # Reset random number generator
> nrows <- NROW(Default)
> samplev <- sample.int(n=nrows, size=nrows/2)
> traindata <- Default[samplev, ]
> logmod <- glm(formulav, data=traindata, family=binomial(logit))
> # Forecast over test data out-of-sample
> testdata <- Default[-samplev, ]
> forecastv <- predict(logmod, newdata=testdata, type="response")
> # Calculate confusion matrix out-of-sample
> table(actual=!testdata$default,
+ forecast=(forecastv < threshold))
```

Forecasting Returns Using Logistic Regression

The weighted average of the volatility and trading volume z-scores can be used to forecast the sign of future returns.

The residuals are the differences between the actual response values (0 and 1), and the calculated probabilities of default.

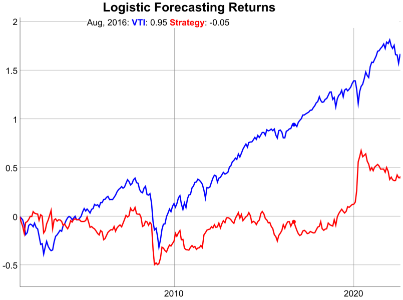
The residuals are not normally distributed, so the data is fitted using the *maximum likelihood* method, instead of least squares.

```
> # Define response and design matrix
> retsf <- rutils::diffit(closep, lagg=5)
> retsf <- drop(coredata(retsf))
> # Fit in-sample logistic regression for positive returns
> retspos <- (retsf > 0)
> logmod <- glm(retspos ~ predictor - 1, family=binomial(logit))
> summary(logmod)
> coeff <- logmod$coefficients
> forecastv <- drop(predictor %*% coeff)
> forecastv <- 1/(1+exp(-forecastv))
> # Calculate error rates
> threshv <- quantile(forecastv, seq(0.01, 0.99, by=0.01))
> error_rates <- sapply(threshv, confun,
+   actual=!retspos, forecastv=forecastv)  # end sapply
> error_rates <- t(error_rates)
> # Calculate the informedness
> informv <- 2 - rowSums(error_rates)
> plot(threshv, informv, t="l", main="Informedness")
> # Find the threshold corresponding to highest informedness
> threshm <- threshv[which.max(informv)]
> forecastpos <- (forecastv > threshm)
> # Fit in-sample logistic regression for negative returns
> retsneg <- (retsf < 0)
> logmod <- glm(retsneg ~ predictor - 1, family=binomial(logit))
> summary(logmod)
> coeff <- logmod$coefficients
> forecastv <- drop(predictor %*% coeff)
> forecastv <- 1/(1+exp(-forecastv))
> # Calculate error rates
> error_rates <- sapply(threshv, confun,
+   actual=!retsneg, forecastv=forecastv)  # end sapply
> error_rates <- t(error_rates)
> # Calculate the informedness
> informv <- 2 - rowSums(error_rates)
> plot(threshv, informv, t="l", main="Informedness")
> # Find the threshold corresponding to highest informedness
> threshm <- threshv[which.max(informv)]
> forecastneg <- (forecastv > threshm)
```

Logistic Forecasting Returns Strategy In-sample

The logistic strategy forecasts the sign of returns, using a logistic regression model with the volatility and trading volumes as predictors.

```
> # Simulate in-sample VTI strategy
> posit <- ifelse(forecastpos, 1, 0)
> posit <- ifelse(forecastneg, -1, posit)
> pnls <- retsp*posit
> # Calculate the Sharpe and Sortino ratios
> wealthv <- cbind(retsp, pnls)
> colnames(wealthv) <- c("VTI", "Strategy")
> sqrt(252)*sapply(wealthv,
+   function(x) c(Sharpe=mean(x)/sd(x), Sortino=mean(x)/sd(x[x<0])))
```

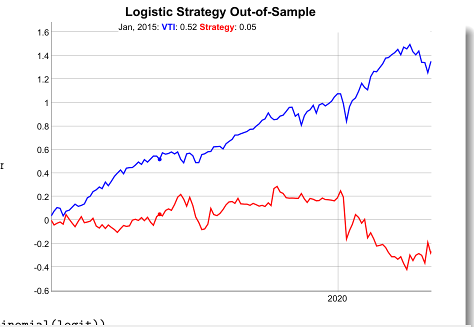


```
> # Plot dygraph of in-sample VTI strategy
> endp <- rutils::calc_endpoints(wealthv, interval="months")
> dygraphs::dygraph(cumsum(wealthv)[endp],
+   main="Logistic Forecasting Returns") %>%
+   dyOptions(colors=c("blue", "red"), strokeWidth=2) %>%
+   dyLegend(show="always", width=500)
```

draft: Logistic Strategy Out-of-Sample

The logistic strategy forecasts the tops and bottoms of prices, using a logistic regression model with the volatility and trading volumes as predictors.

```
> insample <- 1:(nrows %/% 2)
> outsample <- (nrows %/% 2 + 1):nrows
> # Fit in-sample logistic regression for tops
> logmod <- glm(tops[insample] ~ predictor[insample, ], family=binor
> fittedv <- logmod$fitted.values
> coefftop <- logmod$coefficients
> # Calculate error rates and best threshold value
> error_rates <- sapply(threshv, confun,
+   actual=!tops[insample], forecastv=fittedv)  # end sapply
> error_rates <- t(error_rates)
> informv <- 2 - rowSums(error_rates)
> threshtop <- threshv[which.max(informv)]
> # Fit in-sample logistic regression for bottoms
> logmod <- glm(bottoms[insample] ~ predictor[insample, ], family=binomial(logit))
> fittedv <- logmod$fitted.values
> coeffbot <- logmod$coefficients
> # Calculate error rates and best threshold value
> error_rates <- sapply(threshv, confun,
+   actual=!bottoms[insample], forecastv=fittedv)  # end sapply
> error_rates <- t(error_rates)
> informv <- 2 - rowSums(error_rates)
> threshbot <- threshv[which.max(informv)]
> # Calculate out-of-sample forecasts from logistic regression mode
> predictout <- cbind(rep(1, NROW(outsample)), predictor[outsample
> forecastv <- drop(predictout %*% coefftop)
> forecastv <- 1/(1+exp(-forecastv))
> forecastops <- (forecastv > threshtop)
> forecastv <- drop(predictout %*% coeffbot)
> forecastv <- 1/(1+exp(-forecastv))
> forecastbot <- (forecastv > threshbot)
```



```
> # Simulate out-of-sample VTI strategy
> posit <- rep(NA_integer_, NROW(outsample))
> posit[1] <- 0
> posit[forecastops] <- (-1)
> posit[forecastbot] <- 1
> posit <- zoo::na.locf(posit)
> posit <- rutils::lagit(posit)
> pnls <- retsp[outsample, ]*posit
> # Calculate the Sharpe and Sortino ratios
> wealthv <- cbind(retsp[outsample, ], pnls)
> colnames(wealthv) <- c("VTI", "Strategy")
> sqrt(252)*sapply(wealthv,
+   function(x) c(Sharpe=mean(x)/sd(x), Sortino=mean(x)/sd(x[x<0])))
> # Plot dygraph of in-sample VTI strategy
> endp <- rutils::calc_endpoints(wealthv, interval="months")
> dygraphs::dygraph(cumsum(wealthv)[endp],
+   main="Logistic Strategy Out-of-Sample") %>%
+   dyOptions(colors=c("blue", "red"), strokeWidth=2) %>%
```

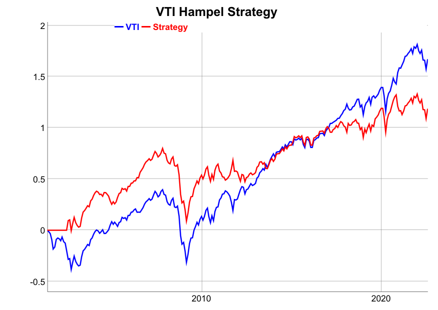
Hampel Filter Strategy

The Hampel filter strategy is a contrarian strategy that uses Hampel z-scores to establish long and short positions.

The Hampel strategy has two meta-parameters: the look-back interval and the threshold level.

The best choice of the meta-parameters can be determined through simulation.

```
> # Calculate VTI percentage returns
> closep <- log(na.omit(rutils::etfenv$prices$VTI))
> retsp <- rutils::diffit(closep)
> # Define look-back window
> look_back <- 11
> # Calculate time series of medians
> medianv <- roll::roll_median(closep, width=look_back)
> # medianv <- TTR::runMedian(closep, n=look_back)
> # Calculate time series of MAD
> madv <- HighFreq::roll_var(closep, look_back=look_back, method="
> # madv <- TTR::runMAD(closep, n=look_back)
> # Calculate time series of z-scores
> zscores <- (closep - medianv)/madv
> zscores[1:look_back, ] <- 0
> tail(zscores, look_back)
> range(zscores)
> # Define threshold value
> threshold <- sum(abs(range(zscores)))/8
> # Simulate VTI strategy
> posit <- rep(NA_integer_, NROW(closep))
> posit[1] <- 0
> posit[zscores < -threshold] <- 1
> posit[zscores > threshold] <- (-1)
> posit <- zoo::na.locf(posit)
> posit <- rutils::lagit(posit)
> pnls <- retsp*posit
```



```
> # Plot dygraph of Hampel strategy pnls
> wealthv <- cbind(retsp, pnls)
> colnames(wealthv) <- c("VTI", "Strategy")
> endp <- rutils::calc_endpoints(wealthv, interval="months")
> dygraphs::dygraph(cumsum(wealthv)[endp],
+   main="VTI Hampel Strategy") %>%
+   dyOptions(colors=c("blue", "red"), strokeWidth=2) %>%
+   dyLegend(show="always", width=500)
```

Vector and Matrix Calculus

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

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

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

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

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

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

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

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

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

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

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

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

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.

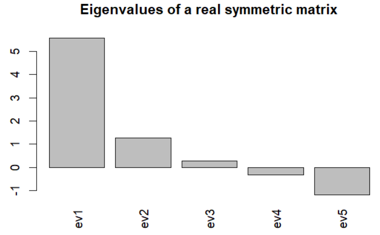
Real symmetric matrices have real *eigenvalues*, and their *eigenvectors* are orthogonal to each other.

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
> matrixv <- matrix(runif(25), nc=5)
> matrixv <- matrixv + t(matrixv)
> # Calculate eigenvectors and eigenvalues
> eigend <- eigen(matrixv)
> eigenvec <- eigend$vectors
> dim(eigenvec)
> # Plot eigenvalues
> barplot(eigend$values, xlab="", ylab="", las=3,
+   names.arg=paste0("ev", 1:NROW(eigend$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*, with $\mathbb{O}^T \mathbb{O} = \mathbb{1}$.

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(eigenvec) %*% eigenvec, digits=4)
> # Diagonalize matrix using eigenvector matrix
> round(t(eigenvec) %*% (matrixv %*% eigenvec), digits=4)
> eigend$values
> # eigen decomposition of matrix by rotating the diagonal matrix
> matrixe <- eigenvec %*% (eigend$values * t(eigenvec))
> # Create diagonal matrix of eigenvalues
> # diagmat <- diag(eigend$values)
> # matrixe <- eigenvec %*% (diagmat %*% t(eigenvec))
> all.equal(matrixv, matrixe)
```

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 that each pair produces a single number,

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).

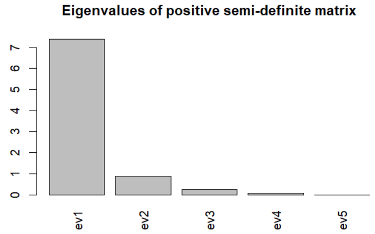
An example of *positive definite* matrices are the covariance matrices of linearly independent variables.

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

Singular Value Decomposition (*SVD*) of Matrices

The *Singular Value Decomposition* (*SVD*) is a generalization of the *eigen decomposition* of square matrices.

The *SVD* of a rectangular matrix $\mathbb{A}$ is defined as the factorization:

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

Where $\mathbb{U}$ and $\mathbb{V}$ are the left and right *singular matrices*, and Σ is a diagonal matrix of *singular values*.

If $\mathbb{A}$ has `m` rows and `n` columns and if `(m > n)`, then $\mathbb{U}$ is an `(m x n)` *rectangular* matrix, Σ is an `(n x n)` *diagonal* matrix, and $\mathbb{V}$ is an `(n x n)` *orthogonal* matrix, and if `(m < n)` then the dimensions are: `(m x m)`, `(m x m)`, and `(m x n)`.

The left $\mathbb{U}$ and right $\mathbb{V}$ singular matrices consist of columns of *orthonormal* vectors, so that $\mathbb{U}^T\mathbb{U} = \mathbb{V}^T\mathbb{V} = \mathbb{1}$.

In the special case when $\mathbb{A}$ is a square matrix, then $\mathbb{U} = \mathbb{V}$, and the *SVD* reduces to the *eigen decomposition*.

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.

```
> # Perform singular value decomposition
> matrixv <- matrix(rnorm(50), nc=5)
> svdec <- svd(matrixv)
> # Recompose matrixv from SVD mat_rices
> all.equal(matrixv, svdec$u %*% (svdec$d*t(svdec$v)))
> # Columns of U and V are orthonormal
> round(t(svdec$u) %*% svdec$u, 4)
> round(t(svdec$v) %*% svdec$v, 4)
```

The Left and Right Singular Matrices

The left $\mathbb{U}$ and right $\mathbb{V}$ singular matrices define rotation transformations into a coordinate system where the matrix $\mathbb{A}$ becomes diagonal:

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

The columns of $\mathbb{U}$ and $\mathbb{V}$ are called the *singular* vectors, and they are only defined up to a reflection (change in sign), i.e. if vec is a singular vector, then so is -vec.

The left singular matrix $\mathbb{U}$ forms the *eigenvectors* of the matrix $\mathbb{A}\mathbb{A}^T$.

The right singular matrix $\mathbb{V}$ forms the *eigenvectors* of the matrix $\mathbb{A}^T\mathbb{A}$.

```
> # Dimensions of left and right matrices
> nleft <- 6 ; nright <- 4
> # Calculate left matrix
> leftmat <- matrix(runif(nleft^2), nc=nleft)
> eigend <- eigen(crossprod(leftmat))
> leftmat <- eigend$vectors[, 1:nright]
> # Calculate right matrix and singular values
> rightmat <- matrix(runif(nright^2), nc=nright)
> eigend <- eigen(crossprod(rightmat))
> rightmat <- eigend$vectors
> singval <- sort(runif(nright, min=1, max=5), decreasing=TRUE)
> # Compose rectangular matrix
> matrixv <- leftmat %*% (singval * t(rightmat))
> # Perform singular value decomposition
> svdec <- svd(matrixv)
> # Recompose matrixv from SVD
> all.equal(matrixv, svdec$u %*% (svdec$d*t(svdec$v)))
> # Compare SVD with matrixv components
> all.equal(abs(svdec$u), abs(leftmat))
> all.equal(abs(svdec$v), abs(rightmat))
> all.equal(svdec$d, singval)
> # Eigen decomposition of matrixv squared
> retsq <- matrixv %*% t(matrixv)
> eigend <- eigen(retsq)
> all.equal(eigend$values[1:nright], singval^2)
> all.equal(abs(eigend$vectors[, 1:nright]), abs(leftmat))
> # Eigen decomposition of matrixv squared
> retsq <- t(matrixv) %*% matrixv
> eigend <- eigen(retsq)
> all.equal(eigend$values, singval^2)
> all.equal(abs(eigend$vectors), abs(rightmat))
```

Inverse of Symmetric 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 $\mathbb{A}^{-1}$ of a *symmetric* square matrix $\mathbb{A}$ can also be expressed as the 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 inverse of *non-symmetric* matrices can be calculated using *Singular Value Decomposition* (*SVD*).

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

```
> # Create random positive semi-definite matrix
> matrixv <- matrix(runif(25), nc=5)
> matrixv <- t(matrixv) %*% matrixv
> # Calculate the inverse of matrixv
> invmat <- solve(a=matrixv)
> # Multiply inverse with matrix
> round(invmat %*% matrixv, 4)
> round(matrixv %*% invmat, 4)
>
> # Calculate eigenvectors and eigenvalues
> eigend <- eigen(matrixv)
> eigenvec <- eigend$vectors
>
> # Perform eigen decomposition of inverse
> inveigen <- eigenvec %*% (t(eigenvec) / eigend$values)
> all.equal(invmat, inveigen)
> # Decompose diagonal matrix with inverse of eigenvalues
> # diagmat <- diag(1/eigend$values)
> # inveigen <-
> #   eigenvec %*% (diagmat %*% t(eigenvec))
```

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.

```
> # Random rectangular matrix: nleft > nright
> nleft <- 6 ; nright <- 4
> matrixv <- matrix(runif(nleft*nright), nc=nright)
> # Calculate generalized inverse of matrixv
> invmat <- MASS::ginv(matrixv)
> round(invmat %*% matrixv, 4)
> all.equal(matrixv, matrixv %*% invmat %*% matrixv)
> # Random rectangular matrix: nleft < nright
> nleft <- 4 ; nright <- 6
> matrixv <- matrix(runif(nleft*nright), nc=nright)
> # Calculate generalized inverse of matrixv
> invmat <- MASS::ginv(matrixv)
> all.equal(matrixv, matrixv %*% invmat %*% matrixv)
> round(matrixv %*% invmat, 4)
> round(invmat %*% matrixv, 4)
> # Perform singular value decomposition
> svdec <- svd(matrixv)
> # Calculate generalized inverse from SVD
> invsvd <- svdec$v %*% (t(svdec$u) / svdec$d)
> all.equal(invsvd, invmat)
> # Calculate Moore-Penrose pseudo-inverse
> invmp <- MASS::ginv(t(matrixv) %*% matrixv) %*% t(matrixv)
> all.equal(invmp, invmat)
```

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

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

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

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

```
> # Create random singular matrix
> # More columns than rows: nright > nleft
> nleft <- 4 ; nright <- 6
> matrixv <- matrix(runif(nleft*nright), nc=nright)
> matrixv <- t(matrixv) %*% matrixv
> # Perform singular value decomposition
> svdec <- svd(matrixv)
> # Incorrect inverse from SVD because of zero singular values
> invsvd <- svdec$v %*% (t(svdec$u) / svdec$d)
> # Inverse property doesn't hold
> all.equal(matrixv, matrixv %*% invsvd %*% matrixv)
```

```
> # Set tolerance for determining zero singular values
> precv <- sqrt(.Machine$double.eps)
> # Check for zero singular values
> round(svdec$d, 12)
> notzero <- (svdec$d > (precv * svdec$d[1]))
> # Calculate regularized inverse from SVD
> invsvd <- svdec$v[, notzero] %*%
+   (t(svdec$u[, notzero]) / svdec$d[notzero])
> # Verify inverse property of matrixv
> all.equal(matrixv, matrixv %*% invsvd %*% matrixv)
> # Calculate regularized inverse using MASS::ginv()
> invmat <- MASS::ginv(matrixv)
> all.equal(invsvd, invmat)
> # Calculate Moore-Penrose pseudo-inverse
> invmp <- MASS::ginv(t(matrixv) %*% matrixv) %*% t(matrixv)
> all.equal(invmp, invmat)
```

Diagonalizing the 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
> unitmat <- matrixv %*% invmat
> round(unitmat, 4)
> round(matrixv %*% invmat, 4)
> round(t(svdec$u) %*% unitmat %*% svdec$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
> matrixv <- matrix(c(1, 2, -1, 2), nc=2)
> vectorv <- c(2, 1)
> # Calculate the inverse of matrixv
> invmat <- solve(a=matrixv)
> invmat %*% matrixv
> # Calculate solution using inverse of matrixv
> solutionv <- invmat %*% vectorv
> matrixv %*% solutionv
> # Calculate solution of linear system
> solutionv <- solve(a=matrixv, b=vectorv)
> matrixv %*% solutionv
```

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 matrix
> matrixv <- matrix(runif(1e4), nc=100)
> matrixv <- t(matrixv) %*% matrixv
> # Calculate eigen decomposition
> eigend <- eigen(matrixv)
> eigenval <- eigend$values
> eigenvec <- eigend$vectors
> # Set tolerance for determining zero singular values
> precv <- sqrt(.Machine$double.eps)
> # If needed convert to positive definite matrix
> notzero <- (eigenval > (precv*eigenval[1]))
> if (sum(!notzero) > 0) {
+   eigenval[!notzero] <- 2*precv
+   matrixv <- eigenvec %*% (eigenval * t(eigenvec))
+ }  # end if
> # Calculate the Cholesky matrixv
> cholmat <- chol(matrixv)
> cholmat[1:5, 1:5]
> all.equal(matrixv, t(cholmat) %*% cholmat)
> # Calculate inverse from Cholesky
> invchol <- chol2inv(cholmat)
> all.equal(solve(matrixv), invchol)
> # Compare speed of Cholesky inversion
> library(microbenchmark)
> summary(microbenchmark(
+   solve=solve(matrixv),
+   cholmat=chol2inv(chol(matrixv)),
+   times=10))[, c(1, 4, 5)]  # end microbenchmark summary
```

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 multiplying them by the *Cholesky* matrix $\mathbb{L}$:

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

```
> # Calculate random covariance matrix
> covmat <- matrix(runif(25), nc=5)
> covmat <- t(covmat) %*% covmat
> # Calculate the Cholesky matrix
> cholmat <- chol(covmat)
> cholmat
> # Simulate random uncorrelated returns
> nassets <- 5
> nrows <- 10000
> retsp <- matrix(rnorm(nassets*nrows), nc=nassets)
> # Calculate correlated returns by applying Cholesky
> retscorr <- retsp %*% cholmat
> # Calculate covariance matrix
> covmat2 <- crossprod(retscorr) /(nrows-1)
> all.equal(covmat, covmat2)
```

Eigenvalues of Singular Covariance Matrices

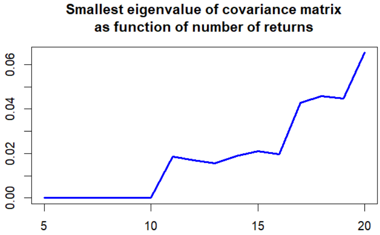
If $\mathbb{R}$ is a matrix of returns (with zero mean) for a portfolio of k assets (columns), over n time periods (rows), then the sample covariance matrix is equal to:

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

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
> nassets <- 10
> nrows <- 100
> set.seed(1121)  # Initialize random number generator
> retsp <- matrix(rnorm(nassets*nrows), nc=nassets)
> # Calculate de-meaned returns matrix
> retsp <- t(t(retsp) - colMeans(retsp))
> # Or
> retsp <- apply(retsp, MARGIN=2, function(x) (x-mean(x)))
> # Calculate covariance matrix
> covmat <- crossprod(retsp) /(nrows-1)
> # Calculate eigenvectors and eigenvalues
> eigend <- eigen(covmat)
> eigend$values
> barplot(eigend$values, # Plot eigenvalues
+   xlab="", ylab="", las=3,
+   names.arg=paste0("ev", 1:NROW(eigend$values)),
+   main="Eigenvalues of covariance matrix")
```


Smallest eigenvalue of covariance matrix
as function of number of returns

```
> # Calculate eigenvectors and eigenvalues
> # as function of number of returns
> ndata <- ((nassets/2):(2*nassets))
> eigenval <- sapply(ndata, function(x) {
+   retsp <- retsp[1:x, ]
+   retsp <- apply(retsp, MARGIN=2, function(y) (y - mean(y)))
+   covmat <- crossprod(retsp) / (x-1)
+   min(eigen(covmat)$values)
+ })  # end sapply
> plot(y=eigenval, x=ndata, t="l", xlab="", ylab="", lwd=3, col="blu
+   main="Smallest eigenvalue of covariance matrix
+   as function of number of returns")
```

Regularized Inverse of Singular Covariance Matrices

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

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

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

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

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

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

```
> # Create rectangular matrix with collinear columns
> matrixv <- matrix(rnorm(10*8), nc=10)
> # Calculate covariance matrix
> covmat <- cov(matrixv)
> # Calculate inverse of covmat - error
> invmat <- solve(covmat)
> # Calculate regularized inverse of covmat
> invmat <- MASS::ginv(covmat)
> # Verify inverse property of matrixv
> all.equal(covmat, covmat %*% invmat %*% covmat)
> # Perform eigen decomposition
> eigend <- eigen(covmat)
> eigenvec <- eigend$vectors
> eigenval <- eigend$values
> # Set tolerance for determining zero singular values
> precv <- sqrt(.Machine$double.eps)
> # Calculate regularized inverse matrix
> notzero <- (eigenval > (precv * eigenval[1]))
> invreg <- eigenvec[, notzero] %*%
+   (t(eigenvec[, notzero]) / eigenval[notzero])
> # Verify that invmat is same as invreg
> all.equal(invmat, invreg)
```

The Bias-Variance Tradeoff of the Regularized Inverse

Removing the very small higher order eigenvalues can also be used to reduce the propagation of statistical noise and improve the signal-to-noise ratio.

Removing a larger number of eigenvalues further reduces the noise, but it increases the bias of the covariance matrix.

This is an example of the *bias-variance tradeoff*.

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

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

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

```
> # Calculate regularized inverse matrix using cutoff
> dimax <- 3
> invmat <- eigenvec[, 1:dimax] %*%
+   (t(eigenvec[, 1:dimax]) / eigend$values[1:dimax])
> # Verify that invmat is same as invreg
> all.equal(invmat, invreg)
```

Shrinkage Estimator of Covariance Matrices

The estimates of the covariance matrix suffer from statistical noise, and those noise 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)
> matrixv <- matrix(rnorm(5e2), nc=5)
> covmat <- cov(matrixv)
> cormat <- cor(matrixv)
> stdev <- sqrt(diag(covmat))
> # Calculate target matrix
> cormean <- mean(cormat[upper.tri(cormat)])
> targetmat <- matrix(cormean, nr=NROW(covmat), nc=NCOL(covmat))
> diag(targetmat) <- 1
> targetmat <- t(t(targetmat * stdev) * stdev)
> # Calculate shrinkage covariance matrix
> alpha <- 0.5
> covshrink <- (1-alpha)*covmat + alpha*targetmat
> # Calculate inverse matrix
> invmat <- solve(covshrink)
```

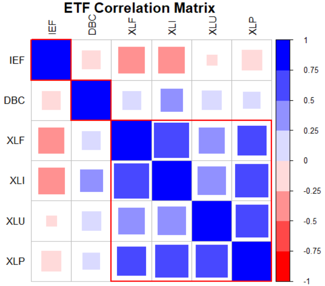
Covariance Matrix of ETF Returns

The covariance matrix $\mathbb{C}$, of the return matrix $\mathbf{r}$ is given by:

$$\mathbb{C} = \frac{(\mathbf{r} - \bar{\mathbf{r}})^T (\mathbf{r} - \bar{\mathbf{r}})}{n - 1}$$

If the returns are *standardized* (de-meaned and scaled) then the covariance matrix is equal to the correlation matrix.

```
> # Select ETF symbols
> symbolv <- c("IEF", "DBC", "XLU", "XLF", "XLP", "XLI")
> # Calculate ETF prices and percentage returns
> pricets <- rutils::etfenv$prices[, symbolv]
> pricets <- zoo::na.locf(pricets, na.rm=FALSE)
> pricets <- zoo::na.locf(pricets, fromLast=TRUE)
> # Calculate log returns without standardizing
> retsp <- rutils::diffit(log(pricets))
> # Calculate covariance matrix
> covmat <- cov(retsp)
> # Standardize (de-mean and scale) the returns
> retsp <- lapply(retsp, function(x) {(x - mean(x))/sd(x)})
> retsp <- rutils::do_call(cbind, retsp)
> round(sapply(retsp, mean), 6)
> sapply(retsp, sd)
> # Alternative (much slower) center (de-mean) and scale the retur
> # retsp <- apply(retsp, 2, scale)
> # retsp <- xts::xts(retsp, zoo::index(pricets))
> # Alternative (much slower) center (de-mean) and scale the retur
> # retsp <- scale(retsp, center=TRUE, scale=TRUE)
> # retsp <- xts::xts(retsp, zoo::index(pricets))
> # Alternative (much slower) center (de-mean) and scale the retur
> # retsp <- t(retsp) - colMeans(retsp)
> # retsp <- retsp/sqrt(rowSums(retsp^2)/(NCOL(retsp)-1))
> # retsp <- t(retsp)
> # retsp <- xts::xts(retsp, zoo::index(pricets))
```


ETF Correlation Matrix

```
> # Calculate correlation matrix
> cormat <- cor(retsp)
> # Reorder correlation matrix based on clusters
> library(corrplot)
> ordern <- corrMatOrder(cormat, order="hclust",
+   hclust.method="complete")
> cormat <- cormat[ordern, ordern]
> # Plot the correlation matrix
> colors <- colorRampPalette(c("red", "white", "blue"))
> x11(width=6, height=6)
> corrplot(cormat, title=NA, tl.col="black", mar=c(0,0,0,0),
+     method="square", col=colors(NCOL(cormat)), tl.cex=0.8,
+     cl.offset=0.75, cl.cex=0.7, cl.align.text="l", cl.ratio=0.25)
> title("ETF Correlation Matrix", line=1)
> # Draw rectangles on the correlation matrix plot
> corrRect.hclust(cormat, k=NROW(cormat) %/% 2,
+   method="complete", col="red")
```

Principal Component Vectors

Principal components are linear combinations of the k return vectors $\mathbf{r}_i$:

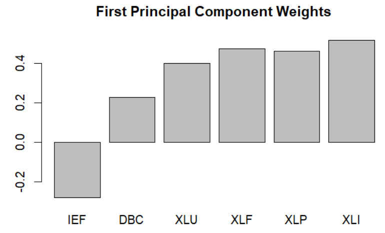
$$\mathbf{pc}_j = \sum_{i=1}^{k} w_{ij}\, \mathbf{r}_i$$

Where $\mathbf{w}_j$ is a vector of weights (loadings) of the *principal component* j, with $\mathbf{w}_j^T \mathbf{w}_j = 1$.

The weights $\mathbf{w}_j$ are chosen to maximize the variance of the *principal components*, under the condition that they are orthogonal:

$$\mathbf{w}_j = \arg\max \left\{ \mathbf{pc}_j^T\, \mathbf{pc}_j \right\}$$
$$\mathbf{pc}_i^T\, \mathbf{pc}_j = 0\ (i \neq j)$$

```
> # create initial vector of portfolio weights
> nweights <- NROW(symbolv)
> weights <- rep(1/sqrt(nweights), nweights)
> names(weights) <- symbolv
> # Objective function equal to minus portfolio variance
> objfun <- function(weights, retsp) {
+   retsp <- retsp %*% weights
+   -sum(retsp^2) + 1e7*(1 - sum(weights^2))^2
+ }  # end objfun
> # Objective for equal weight portfolio
> objfun(weights, retsp)
> # Compare speed of vector multiplication methods
> summary(microbenchmark(
+   transp=(t(retsp[, 1]) %*% retsp[, 1]),
+   sumv=sum(retsp[, 1]^2),
+   times=10))[, c(1, 4, 5)]
```



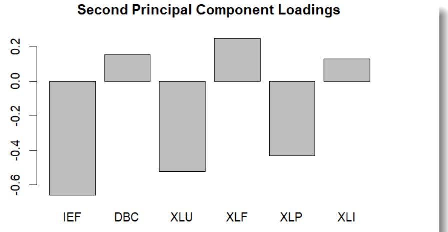
```
> # Find weights with maximum variance
> optiml <- optim(par=weights,
+   fn=objfun,
+   retsp=retsp,
+   method="L-BFGS-B",
+   upper=rep(10.0, nweights),
+   lower=rep(-10.0, nweights))
> # Optimal weights and maximum variance
> weights1 <- optiml$par
> -objfun(weights1, retsp)
> # Plot first principal component weights
> barplot(weights1, names.arg=names(weights1), xlab="", ylab="",
+   main="First Principal Component Weights")
```

Higher Order Principal Components

The second *principal component* can be calculated by maximizing its variance, under the constraint that it must be orthogonal to the first *principal component*.

Similarly, higher order *principal components* can be calculated by maximizing their variances, under the constraint that they must be orthogonal to all the previous *principal components*.

```
> # PC1 returns
> pc1 <- drop(retsp %*% weights1)
> # Redefine objective function
> objfun <- function(weights, retsp) {
+   retsp <- retsp %*% weights
+   -sum(retsp^2) + 1e7*(1 - sum(weights^2))^2 +
+     1e7*(sum(weights1*weights))^2
+ }  # end objfun
> # Find second PC weights using parallel DEoptim
> optiml <- DEoptim::DEoptim(fn=objfun,
+   upper=rep(10, NCOL(retsp)),
+   lower=rep(-10, NCOL(retsp)),
+   retsp=retsp, control=list(parVar="weights1",
+     trace=FALSE, itermax=1000, parallelType=1))
```



```
> # PC2 weights
> weights2 <- optiml$optim$bestmem
> names(weights2) <- colnames(retsp)
> sum(weights2^2)
> sum(weights1*weights2)
> # PC2 returns
> pc2 <- drop(retsp %*% weights2)
> # Plot second principal component loadings
> barplot(weights2, names.arg=names(weights2), xlab="", ylab="",
+   main="Second Principal Component Loadings")
```

Eigenvalues of the Covariance Matrix

The portfolio variance: $\mathbf{w}^T \mathbb{C} \, \mathbf{w}$ can be maximized under the *quadratic* weights constraint $\mathbf{w}^T \mathbf{w} = 1$, by maximizing the *Lagrangian* $\mathcal{L}$:

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

Where λ is a *Lagrange multiplier*.

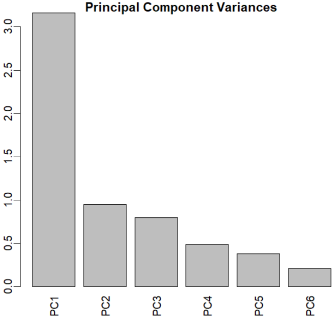
The maximum variance portfolio weights can be found by differentiating $\mathcal{L}$ with respect to $\mathbf{w}$ and setting it to zero:

$$\mathbb{C} \, \mathbf{w} = \lambda \, \mathbf{w}$$

This is the *eigenvalue* equation of the covariance matrix $\mathbb{C}$, with the optimal weights $\mathbf{w}$ forming an *eigenvector*, and λ is the *eigenvalue* corresponding to the *eigenvector* $\mathbf{w}$.

The *eigenvalues* are the variances of the *eigenvectors*, and their sum is equal to the sum of the return variances:

$$\sum_{i=1}^{k} \lambda_i = \frac{1}{1-k} \sum_{i=1}^{k} \mathbf{r}_i^T \, \mathbf{r}_i$$



```
> # Calculate eigenvectors and eigenvalues
> eigend <- eigen(cormat)
> eigend$vectors
> # Compare with optimization
> all.equal(sum(diag(cormat)), sum(eigend$values))
> all.equal(abs(eigend$vectors[, 1]), abs(weights1), check.attribute
> all.equal(abs(eigend$vectors[, 2]), abs(weights2), check.attribute
> all.equal(eigend$values[1], var(pc1), check.attributes=FALSE)
> all.equal(eigend$values[2], var(pc2), check.attributes=FALSE)
> # Eigenvalue equations
> (cormat %*% weights1) / weights1 / var(pc1)
> (cormat %*% weights2) / weights2 / var(pc2)
> # Plot eigenvalues
> barplot(eigend$values, names.arg=paste0("PC", 1:nweights),
+   las=3, xlab="", ylab="", main="Principal Component Variances")
```

Principal Component Analysis Versus *Eigen Decomposition*

Principal Component Analysis (*PCA*) is equivalent to the *eigen decomposition* of either the correlation or the covariance matrix.

If the input time series *are* scaled, then *PCA* is equivalent to the eigen decomposition of the *correlation matrix*.

If the input time series *are not* scaled, then *PCA* is equivalent to the eigen decomposition of the *covariance matrix*.

Scaling the input time series improves the accuracy of the *PCA dimension reduction*, allowing a smaller number of *principal components* to more accurately capture the data contained in the input time series.

The number of *eigenvalues* is equal to the dimension of the covariance matrix.

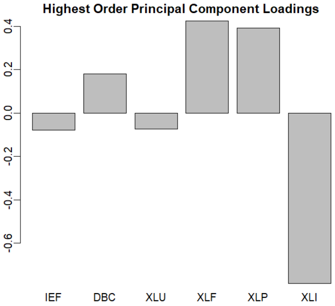
```
> # Eigen decomposition of correlation matrix
> eigend <- eigen(cormat)
> # Perform PCA with scaling
> pcad <- prcomp(retsp, scale=TRUE)
> # Compare outputs
> all.equal(eigend$values, pcad$sdev^2)
> all.equal(abs(eigend$vectors), abs(pcad$rotation),
+     check.attributes=FALSE)
> # Eigen decomposition of covariance matrix
> eigend <- eigen(covmat)
> # Perform PCA without scaling
> pcad <- prcomp(retsp, scale=FALSE)
> # Compare outputs
> all.equal(eigend$values, pcad$sdev^2)
> all.equal(abs(eigend$vectors), abs(pcad$rotation),
+     check.attributes=FALSE)
```

Minimum Variance Portfolio

The highest order *principal component*, with the smallest eigenvalue, has the lowest possible variance, under the *quadratic* weights constraint: $\mathbf{w}^T\mathbf{w} = 1$.

So the highest order *principal component* is equal to the *Minimum Variance Portfolio*.

```
> # Redefine objective function to minimize variance
> objfun <- function(weights, retsp) {
+   retsp <- retsp %*% weights
+   sum(retsp^2) + 1e7*(1 - sum(weights^2))^2
+ }  # end objfun
> # Find highest order PC weights using parallel DEoptim
> optiml <- DEoptim::DEoptim(fn=objfun,
+   upper=rep(10, NCOL(retsp)),
+   lower=rep(-10, NCOL(retsp)),
+   retsp=retsp, control=list(trace=FALSE,
+     itermax=1000, parallelType=1))
> # PC6 weights and returns
> weights6 <- optiml$optim$bestmem
> names(weights6) <- colnames(retsp)
> sum(weights6^2)
> sum(weights1*weights6)
> # Compare with eigend vector
> weights6
> eigend$vectors[, 6]
> # Calculate objective function
> objfun(weights6, retsp)
> objfun(eigend$vectors[, 6], retsp)
```



```
> # Plot highest order principal component loadings
> x11(width=6, height=5)
> par(mar=c(2.5, 2, 2, 3), oma=c(0, 0, 0, 0), mgp=c(2, 0.5, 0))
> barplot(weights6, names.arg=names(weights2), xlab="", ylab="",
+   main="Highest Order Principal Component Loadings")
```

Principal Component Analysis of ETF Returns

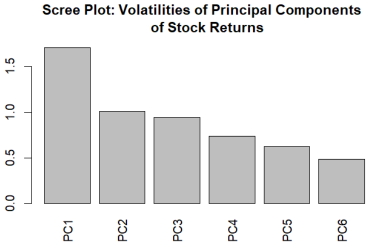
Principal Component Analysis (*PCA*) is a *dimension reduction* technique, that explains the returns of a large number of correlated time series as linear combinations of a smaller number of principal component time series.

The input time series are often scaled by their standard deviations, to improve the accuracy of *PCA dimension reduction*, so that more information is retained by the first few *principal component* time series.

If the input time series are not scaled, then *PCA* analysis is equivalent to the *eigen decomposition* of the covariance matrix, and if they are scaled, then *PCA* analysis is equivalent to the *eigen decomposition* of the correlation matrix.

The function `prcomp()` performs *Principal Component Analysis* on a matrix of data (with the time series as columns), and returns the results as a list of class `prcomp`.

The `prcomp()` argument `scale=TRUE` specifies that the input time series should be scaled by their standard deviations.



A *scree plot* is a bar plot of the volatilities of the *principal components*.

```
> # Perform principal component analysis PCA
> pcad <- prcomp(retsp, scale=TRUE)
> # Plot standard deviations of principal components
> barplot(pcad$sdev, names.arg=colnames(pcad$rotation),
+   las=3, xlab="", ylab="",
+   main="Scree Plot: Volatilities of Principal Components \n of Sto
```

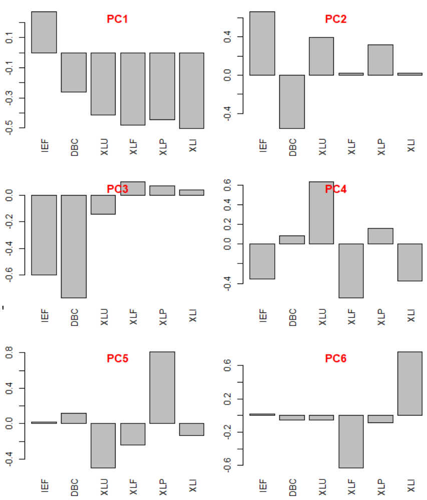
Principal Component Loadings (Weights)

Principal component loadings are the weights of portfolios which have mutually orthogonal returns.

The *principal component* (*PC*) portfolios represent the different orthogonal modes of the return variance.

The *PC* portfolios typically consist of long or short positions of highly correlated groups of assets (clusters), so that they represent relative value portfolios.

```
> # Calculate principal component loadings (weights)
> pcad$rotation
> # Plot barplots with PCA weights in multiple panels
> x11(width=6, height=7)
> par(mfrow=c(nweights/2, 2))
> par(mar=c(3, 2, 2, 1), oma=c(0, 0, 0, 0))
> for (ordern in 1:nweights) {
+   barplot(pcad$rotation[, ordern], las=3, xlab="", ylab="", main='
+   title(paste0("PC", ordern), line=-1, col.main="red")
+ }  # end for
```



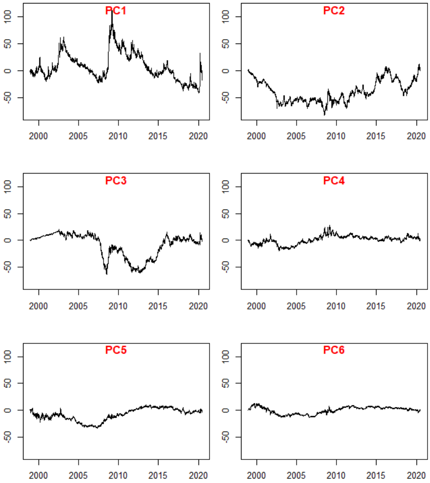
Principal Component Time Series

The time series of the *principal components* can be calculated by multiplying the loadings (weights) times the original data.

The *principal component* time series have mutually orthogonal returns.

Higher order *principal components* are gradually less volatile.

```
> # Calculate products of principal component time series
> round(t(pcad$x) %*% pcad$x, 2)
> # Calculate principal component time series from returns
> dates <- zoo::index(pricets)
> retspca <- xts::xts(retsp %*% pcad$rotation, order.by=dates)
> round(cov(retspca), 3)
> all.equal(coredata(retspca), pcad$x, check.attributes=FALSE)
> pcacum <- cumsum(retspca)
> # Plot principal component time series in multiple panels
> rangev <- range(pcacum)
> for (ordern in 1:nweights) {
+   plot.zoo(pcacum[, ordern], ylim=rangev, xlab="", ylab="")
+   title(paste0("PC", ordern), line=-1, col.main="red")
+ }  # end for
```



Dimension Reduction Using Principal Component Analysis

The original time series can be calculated exactly from the time series of all the *principal components*, by inverting the loadings matrix.

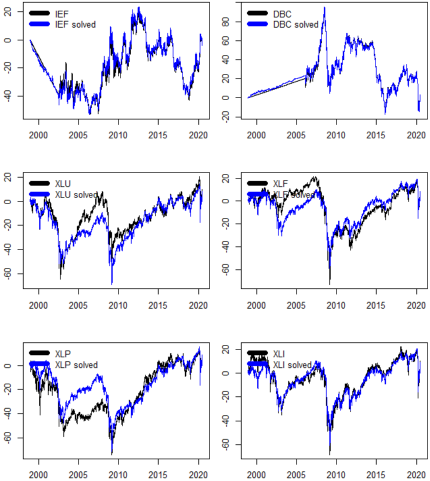
The original time series can be calculated approximately from just the first few *principal components*, which demonstrates that *PCA* is a form of *dimension reduction*.

The *Kaiser-Guttman* rule uses only *principal components* with *variance* greater than 1.

Another rule is to use the *principal components* with the largest standard deviations which sum up to 80% of the total variance of returns.

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

```
> # Invert all the principal component time series
> retspca <- retsp %*% pcad$rotation
> solved <- retspca %*% solve(pcad$rotation)
> all.equal(coredata(retsp), solved)
> # Invert first 3 principal component time series
> solved <- retspca[, 1:3] %*% solve(pcad$rotation)[1:3, ]
> solved <- xts::xts(solved, dates)
> solved <- cumsum(solved)
> retc <- cumsum(retsp)
> # Plot the solved returns
> for (symbol in symbolv) {
+   plot.zoo(cbind(retc[, symbol], solved[, symbol]),
+     plot.type="single", col=c("black", "blue"), xlab="", ylab="")
+   legend(x="topleft", bty="n", legend=paste0(symbol, c("", " solved")),
+    title=NULL, inset=0.0, cex=1.0, lwd=6, lty=1, col=c("black", "blue"))
+ }  # end for
```



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 vector z* is explained by the *predictor x* (also called the *explanatory variable* or *independent variable*).

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
> formulav <- z ~ x + y - 1
> formulav
>
> # Collapse vector of strings into single text string
> paste0("x", 1:5)
> paste(paste0("x", 1:5), collapse="+")
>
> # Create formula from text string
> formulav <- as.formula(
+   # Coerce text strings to formula
+   paste("z ~ ",
+   paste(paste0("x", 1:5), collapse="+")
+   )  # end paste
+ )  # end as.formula
> class(formulav)
> formulav
> # Modify the formula using "update"
> update(formulav, log(.) ~ . + beta)
```

Simple *Linear Regression*

A Simple Linear Regression is a linear model between a *response vector* y and a single *predictor* x, defined by the formula:

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

α and β are the unknown *regression coefficients*.

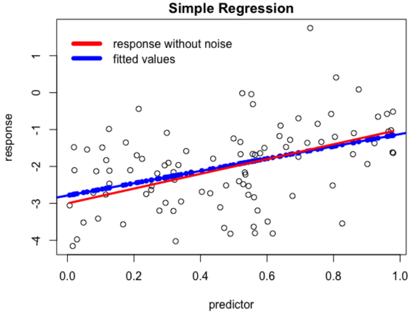
ε_i are the *residuals*, which are usually assumed to be standard normally distributed $\phi(0, \sigma_\varepsilon)$, independent, and stationary.

In the Ordinary Least Squares method (*OLS*), the regression parameters are estimated by minimizing 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 predictor variables, with the index i ranging from 1 to n.



```
> # Define explanatory (predictor) variable
> nrows <- 100
> set.seed(1121)  # Initialize random number generator
> predictor <- runif(nrows)
> noise <- rnorm(nrows)
> # Response equals linear form plus random noise
> response <- (-3 + 2*predictor + noise)
```

The *response vector* and the *predictor matrix* don't have to be normally distributed.

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 can be obtained by manipulating the equation for RSS_{β} as follows:

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

Where $\hat{x} = x - \bar{x} \mathbb{1}$ and $\hat{y} = y - \bar{y} \mathbb{1}$ are the de-meaned variables. Then β is given by:

$$\beta = \frac{\hat{y}^T \hat{x}}{\hat{x}^T \hat{x}} = \frac{\sigma_y}{\sigma_x} \rho_{xy}$$

β is proportional to the correlation coefficient ρ_{xy} between the response and predictor variables.

If the response and predictor 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 *predictor* x: $RSS_{\beta} = -2 \varepsilon^T x = 0$.

The expected value of the *RSS* is equal to the *degrees of freedom* $(n - 2)$ times the variance σ_{ε}^2 of the *residuals* ε_i: $\mathbb{E}[RSS] = (n - 2) \sigma_{\varepsilon}^2$.

```
> # Calculate de-meaned explanatory (predictor) and response vectors
> predzm <- predictor - mean(predictor)
> respzm <- response - mean(response)
> # Calculate the regression beta
> betav <- cov(predictor, response)/var(predictor)
> # Calculate the regression alpha
> alpha <- mean(response) - betav*mean(predictor)
```

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
> formulav <- response ~ predictor
> model <- lm(formulav)  # Perform regression
> class(model)  # Regressions have class lm
[1] "lm"
> attributes(model)
$names
 [1] "coefficients"  "residuals"     "effects"       "rank"
 [5] "fitted.values" "assign"        "qr"            "df.residual"
 [9] "xlevels"       "call"          "terms"         "model"

$class
[1] "lm"
> eval(model$call$formula)  # Regression formula
response ~ predictor
> model$coeff  # Regression coefficients
(Intercept)   predictor
      -2.79        1.67
> all.equal(coef(model), c(alpha, betav),
+   check.attributes=FALSE)
[1] TRUE
```

The *Fitted Values* of Linear Regression

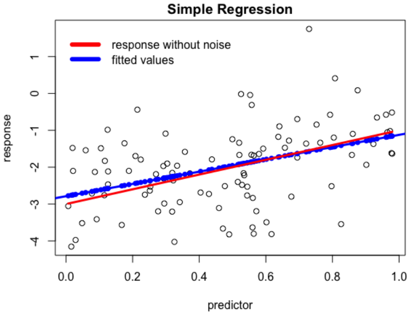
The *fitted values* y_{fit} are the estimates of the *response vector* obtained from the regression model:

$$y_{fit} = \alpha + \beta x$$

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.

```
> fittedv <- (alpha + betav*predictor)
> all.equal(fittedv, model$fitted.values, check.attributes=FALSE)
> x11(width=5, height=4)  # Open x11 for plotting
> # Set plot parameters to reduce whitespace around plot
> par(mar=c(5, 5, 2, 1), oma=c(0, 0, 0, 0))
> # Plot scatterplot using formula
> plot(formulav, xlab="predictor", ylab="response")
> title(main="Simple Regression", line=0.5)
> # Add regression line
> abline(model, lwd=3, col="blue")
> # Plot fitted (predicted) response values
> points(x=predictor, y=model$fitted.values, pch=16, col="blue")
```



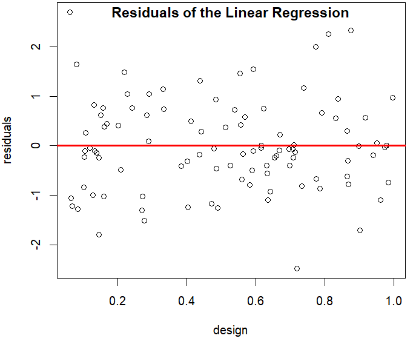
```
> # Plot response without noise
> lines(x=predictor, y=(response-noise), col="red", lwd=3)
> legend(x="topleft", # Add legend
+        legend=c("response without noise", "fitted values"),
+        title=NULL, inset=0.08, cex=0.8, lwd=6,
+        lty=1, col=c("red", "blue"))
```

Linear Regression Residuals

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

$$\varepsilon_i = y_i - y_{fit}$$

```
> # Calculate the residuals
> fittedv <- (alpha + betav*predictor)
> residuals <- (response - fittedv)
> all.equal(residuals, model$residuals, check.attributes=FALSE)
[1] TRUE
> # Residuals are orthogonal to the predictor
> all.equal(sum(residuals*predictor), target=0)
[1] TRUE
> # Residuals are orthogonal to the fitted values
> all.equal(sum(residuals*fittedv), target=0)
[1] TRUE
> # Sum of residuals is equal to zero
> all.equal(mean(residuals), target=0)
[1] TRUE
```



```
> x11(width=6, height=5)  # Open x11 for plotting
> # Set plot parameters to reduce whitespace around plot
> par(mar=c(5, 5, 1, 1), oma=c(0, 0, 0, 0))
> # Extract residuals
> datav <- cbind(predictor, model$residuals)
> colnames(datav) <- c("predictor", "residuals")
> # Plot residuals
> plot(datav)
> title(main="Residuals of the Linear Regression", line=-1)
> abline(h=0, lwd=3, col="red")
```

Standard Errors of Regression Coefficients

The *residuals* are the source of error in the regression model, producing uncertainty in the *response vector* y and in the regression coefficients: $y_i = \alpha + \beta x_i + \varepsilon_i$.

The standard errors of the regression coefficients are equal to their standard deviations, given the *residuals* as the source of error.

Since $\beta = \frac{\hat{y}^T \hat{x}}{\hat{x}^T \hat{x}}$, then its variance is equal to:

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

Since $\alpha = \bar{y} - \beta \bar{x}$, then its variance is equal to:

$$\sigma_\alpha^2 = \frac{\sigma_\varepsilon^2}{n} + \sigma_\beta^2 \bar{x}^2 = \sigma_\varepsilon^2 (\frac{1}{n} + \frac{\bar{x}^2}{\hat{x}^T \hat{x}})$$

```
> # Degrees of freedom of residuals
> degf <- model$df.residual
> # Standard deviation of residuals
> residsd <- sqrt(sum(residuals^2)/degf)
> # Standard error of beta
> betasd <- residsd/sqrt(sum(predzm^2))
> # Standard error of alpha
> alphasd <- residsd*
+   sqrt(1/nrows + mean(predictor)^2/sum(predzm^2))
```

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 the model divided by unexplained variance,

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

```
> modelsum <- summary(model)  # Copy regression summary
> modelsum  # Print the summary to console

Call:
lm(formula = formulav)

Residuals:
   Min     1Q Median     3Q    Max 
-2.133 -0.649  0.106  0.590  3.321 

Coefficients:
            Estimate Std. Error t value Pr(>|t|)    
(Intercept)   -2.787      0.196  -14.20  < 2e-16 ***
predictor      1.665      0.357    4.67  9.8e-06 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 0.988 on 98 degrees of freedom
Multiple R-squared:  0.182,Adjusted R-squared:  0.173 
F-statistic: 21.8 on 1 and 98 DF,  p-value: 9.75e-06
> attributes(modelsum)$names  # get summary elements
 [1] "call"          "terms"          "residuals"     "coefficients"
 [5] "aliased"       "sigma"          "df"            "r.squared"   
 [9] "adj.r.squared" "fstatistic"     "cov.unscaled" 
```

Regression Model Diagnostic Statistics

The *null hypothesis* for regression is that the 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 values exceeding the *t*-statistic, assuming the *null hypothesis* is true.

A small *p*-value means that the regression coefficients are very unlikely to be zero (given the data).

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

If they are not, 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.

Asset returns are very far from normal, so the small *p*-values shouldn't be automatically interpreted as meaning that the regression is statistically significant.

```
> modelsum$coeff
            Estimate Std. Error t value Pr(>|t|)
(Intercept)    -2.79      0.196  -14.20 1.61e-25
predictor       1.67      0.357    4.67 9.75e-06
> # Standard errors
> modelsum$coefficients[2, "Std. Error"]
[1] 0.357
> all.equal(c(alphasd, betasd),
+   modelsum$coefficients[, "Std. Error"],
+   check.attributes=FALSE)
[1] TRUE
> # R-squared
> modelsum$r.squared
[1] 0.182
> modelsum$adj.r.squared
[1] 0.173
> # F-statistic and ANOVA
> modelsum$fstatistic
value numdf dendf
 21.8   1.0  98.0
> anova(model)
Analysis of Variance Table

Response: response
          Df Sum Sq Mean Sq F value  Pr(>F)
predictor  1   21.3   21.25    21.8 9.8e-06 ***
Residuals 98   95.7    0.98
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

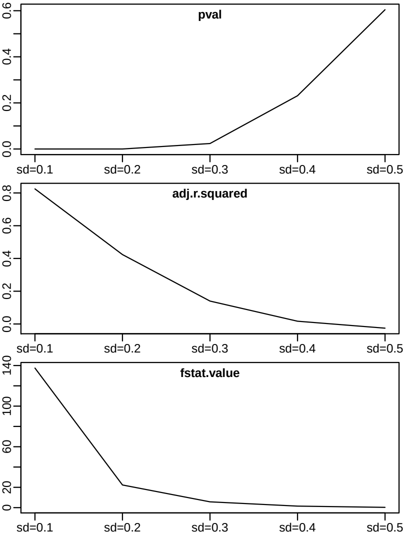
Weak Regression

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

```
> # High noise compared to coefficient
> response <- (-3 + 2*predictor + rnorm(nrows, sd=8))
> model <- lm(formulav)  # Perform regression
> # Values of regression coefficients are not
> # Statistically significant
> summary(model)

Call:
lm(formula = formulav)

Residuals:
    Min      1Q  Median      3Q     Max 
-16.430  -4.325   0.735   4.365  16.720 

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept)    -1.65       1.44   -1.14     0.26
predictor      -1.70       2.62   -0.65     0.52

Residual standard error: 7.25 on 98 degrees of freedom
Multiple R-squared:  0.0043,Adjusted R-squared:  -0.00586 
F-statistic: 0.423 on 1 and 98 DF,  p-value: 0.517
```

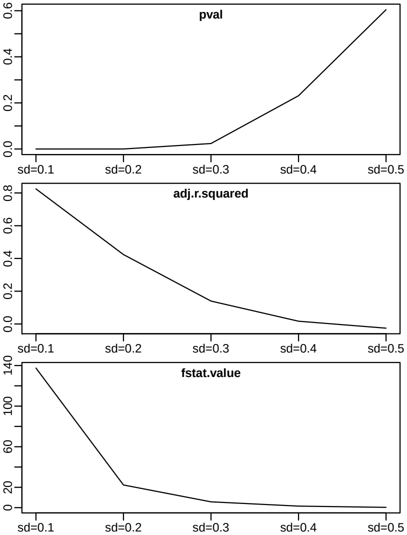
Influence of Noise on Regression

```
> reg_stats <- function(stdev) {  # Noisy regression
+   set.seed(1121)  # initialize number generator
+ # Define explanatory (predictor) and response variables
+   predictor <- rnorm(100, mean=2)
+   response <- (1 + 0.2*predictor +
+   rnorm(NROW(predictor), sd=stdev))
+ # Specify regression formula
+   formulav <- response ˜ predictor
+ # Perform regression and get summary
+   modelsum <- summary(lm(formulav))
+ # Extract regression statistics
+   with(modelsum, 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
> vecsd <- seq(from=0.1, to=0.5, by=0.1)
> names(vecsd) <- paste0("sd=", vecsd)
> statsmat <- t(sapply(vecsd, reg_stats))
> # Plot in loop
> par(mfrow=c(NCOL(statsmat), 1))
> for (it in 1:NCOL(statsmat)) {
+   plot(statsmat[, it], type="l",
+  xaxt="n", xlab="", ylab="", main="")
+   title(main=colnames(statsmat)[it], line=-1.0)
+   axis(1, at=1:(NROW(statsmat)), labels=rownames(statsmat))
+ }  # end for
```



Influence of Noise on Regression Another Method

```
> reg_stats <- function(datav) {  # get regression
+ # Perform regression and get summary
+   colnamev <- colnames(datav)
+   formulav <- paste(colnamev[2], colnamev[1], sep="~")
+   modelsum <- summary(lm(formulav, data=datav))
+ # Extract regression statistics
+   with(modelsum, 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
> vecsd <- seq(from=0.1, to=0.5, by=0.1)
> names(vecsd) <- paste0("sd=", vecsd)
> statsmat <- t(sapply(vecsd, function(stdev) {
+     set.seed(1121)  # initialize number generator
+ # Define explanatory (predictor) and response variables
+     predictor <- rnorm(100, mean=2)
+     response <- (1 + 0.2*predictor +
+ rnorm(NROW(predictor), sd=stdev))
+     reg_stats(data.frame(predictor, response))
+     }))
> # Plot in loop
> par(mfrow=c(NCOL(statsmat), 1))
> for (it in 1:NCOL(statsmat)) {
+   plot(statsmat[, it], type="l",
+  xaxt="n", xlab="", ylab="", main="")
+   title(main=colnames(statsmat)[it], line=-1.0)
+   axis(1, at=1:(NROW(statsmat)),
+  labels=rownames(statsmat))
+ }  # end for
```



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.

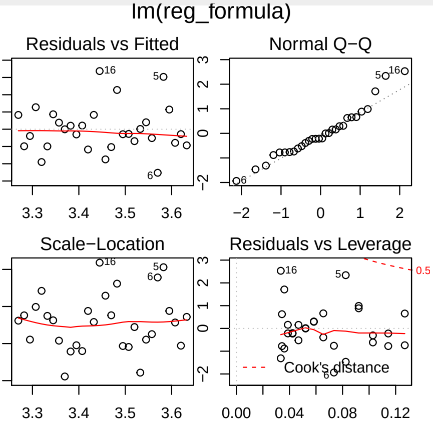
"Residuals vs Leverage" is a scatterplot of the residuals vs. their leverage.

Leverage measures the amount by which the fitted values would change if the response values were shifted by a small amount.

Cook's distance measures the influence of a single observation on the fitted values, and is proportional to the sum of the squared differences between predictions made with all observations and predictions made without the observation.

Points with large leverage, or a Cook's distance greater than 1 suggest the presence of an outlier or a poor model,

```
> par(mfrow=c(2, 2))  # Plot 2x2 panels
> plot(model)  # Plot diagnostic scatterplots
> plot(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
> lmtest::dwtest(model)

Durbin-Watson test

data:  model
DW = 2, p-value = 0.7
alternative hypothesis: true autocorrelation is greater than 0
```

The *Leverage* for Univariate Regression

We can add an extra unit column to the *predictor matrix* $\mathbb{X}$ so that the univariate regression can be written in *homogeneous form* as:

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

With two *regression coefficients*: $\beta = (\alpha, \beta_1)$, and a *predictor matrix* $\mathbb{X}$ with two columns, with the first column equal to a unit vector.

After the second column of the *predictor matrix* $\mathbb{X}$ is de-meaned, its *covariance matrix* is given by:

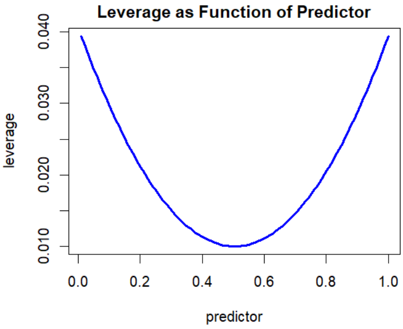
$$\mathbb{X}^T \mathbb{X} = \begin{pmatrix} n & 0 \\ 0 & \sum_{i=1}^n (x_i - \bar{x})^2 \end{pmatrix}$$

And the *influence matrix* $\mathbb{H}$ is given by:

$$\mathbb{H}_{ij} = [\mathbb{X}(\mathbb{X}^T \mathbb{X})^{-1} \mathbb{X}^T]_{ij} = \frac{1}{n} + \frac{(x_i - \bar{x})(x_j - \bar{x})}{\sum_{i=1}^n (x_i - \bar{x})^2}$$

The first term above is due to the influence of the regression intercept α, and the second term is due to the influence of the regression slope β_1.

The diagonal elements of the *influence matrix* $\mathbb{H}_{ii}$ form the *leverage vector*.



```
> # Add unit column to the predictor matrix
> predictor <- cbind(rep(1, nrows), predictor)
> # Calculate generalized inverse of the predictor matrix
> invpred <- MASS::ginv(predictor)
> # Calculate the influence matrix
> influencem <- predictor %*% invpred
> # Plot the leverage vector
> ordern <- order(predictor[, 2])
> plot(x=predictor[ordern, 2], y=diag(influencem)[ordern],
+      type="l", lwd=3, col="blue",
+      xlab="predictor", ylab="leverage",
+      main="Leverage as Function of Predictor")
```

Covariance Matrix of Fitted Values in Univariate Regression

The *fitted values* y_{fit} can be considered to be *random variables* $\hat{y}_{fit}$:

$$\hat{y}_{fit} = \mathbb{H}\hat{y} = \mathbb{H}(y_{fit} + \hat{\varepsilon}) = y_{fit} + \mathbb{H}\hat{\varepsilon}$$

The *covariance matrix* of the *fitted values* $\hat{y}_{fit}$ is:

$$\sigma^2_{fit} = \frac{\mathbb{E}[\mathbb{H}\hat{\varepsilon}(\mathbb{H}\hat{\varepsilon})^T]}{d_{free}} = \frac{\mathbb{E}[\mathbb{H}\,\hat{\varepsilon}\hat{\varepsilon}^T\,\mathbb{H}^T]}{d_{free}} =$$

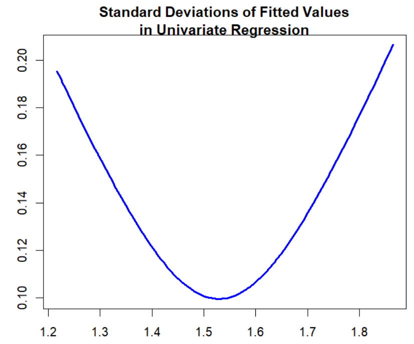
$$\frac{\mathbb{H}\,\mathbb{E}[\hat{\varepsilon}\hat{\varepsilon}^T]\,\mathbb{H}^T}{d_{free}} = \sigma^2_{\varepsilon}\,\mathbb{H} = \sigma^2_{\varepsilon}\,\mathbb{X}(\mathbb{X}^T\mathbb{X})^{-1}\mathbb{X}^T$$

The square of the *influence matrix* $\mathbb{H}$ is equal to itself (it's idempotent): $\mathbb{H}\,\mathbb{H}^T = \mathbb{H}$.

The variance of the *fitted values* σ^2_{fit} increases with the distance of the *predictors* from their mean values.

This is because the *fitted values* farther from their mean are more sensitive to the variance of the regression slope.

```
> # Calculate the influence matrix
> influencem <- predictor %*% invpred
> # The influence matrix is idempotent
> all.equal(influencem, influencem %*% influencem)
```



```
> # Calculate covariance and standard deviations of fitted values
> betas <- invpred %*% response
> fittedv <- drop(predictor %*% betas)
> residuals <- drop(response - fittedv)
> degf <- (NROW(predictor) - NCOL(predictor))
> residvar <- sqrt(sum(residuals^2)/degf)
> fitcovar <- residvar*influencem
> fitsd <- sqrt(diag(fitcovar))
> # Plot the standard deviations
> fitsd <- cbind(fitted=fittedv, stddev=fitsd)
> fitsd <- fitsd[order(fittedv), ]
> plot(fitsd, type="l", lwd=3, col="blue",
+      xlab="Fitted Value", ylab="Standard Deviation",
+      main="Standard Deviations of Fitted Values\nin Univariate Reg
```

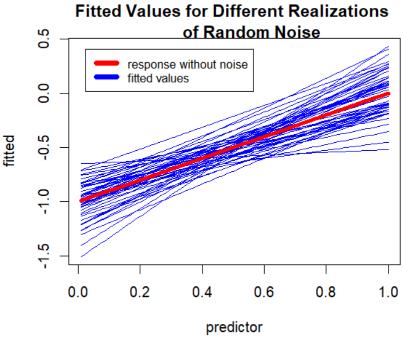
Fitted Values for Different Realizations of Random Noise

The fitted values are more volatile for *predictor* values that are further away from their mean, because those points have higher *leverage*.

The higher *leverage* of points further away from the mean of the *predictor* is due to their greater sensitivity to changes in the slope of the regression.

The fitted values for different realizations of random noise can be calculated using the influence matrix.

```
> # Calculate response without random noise for univariate regressio
> # equal to weighted sum over columns of predictor.
> betas <- c(-1, 1)
> response <- predictor %*% betas
> # Perform loop over different realizations of random noise
> fittedv <- lapply(1:50, function(it) {
+   # Add random noise to response
+   response <- response + rnorm(nrows, sd=1.0)
+   # Calculate fitted values using influence matrix
+   influencem %*% response
+ })  # end lapply
> fittedv <- rutils::do_call(cbind, fittedv)
```



```
> # Plot fitted values
> matplot(x=predictor[,2], y=fittedv,
+ type="l", lty="solid", lwd=1, col="blue",
+ xlab="predictor", ylab="fitted",
+ main="Fitted Values for Different Realizations
+ of Random Noise")
> lines(x=predictor[,2], y=response, col="red", lwd=4)
> legend(x="topleft", # Add legend
+        legend=c("response without noise", "fitted values"),
+        title=NULL, inset=0.05, cex=0.8, lwd=6,
+        lty=1, col=c("red", "blue"))
```

Predictions From *Univariate Regression* Models

The prediction y_{pred} from a regression model is equal to the *response value* corresponding to the *predictor* vector with the new data $\mathbb{X}_{new}$:

$$y_{pred} = \mathbb{X}_{new} \, \beta$$

The variance σ^2_{pred} of the *predicted value* is:

$$\sigma^2_{pred} = \frac{\mathbb{E}[\mathbb{X}_{new} \mathbb{X}_{inv} \hat{\varepsilon} \, (\mathbb{X}_{new} \mathbb{X}_{inv} \hat{\varepsilon})^T]}{d_{free}} =$$

$$\frac{\mathbb{E}[\mathbb{X}_{new} \mathbb{X}_{inv} \hat{\varepsilon} \hat{\varepsilon}^T \mathbb{X}_{inv}^T \mathbb{X}_{new}^T]}{d_{free}} = \sigma^2_{\varepsilon} \mathbb{X}_{new} \mathbb{X}_{inv} \mathbb{X}_{inv}^T \mathbb{X}_{new}^T =$$

$$\sigma^2_{\varepsilon} \, \mathbb{X}_{new} (\mathbb{X}^T \mathbb{X})^{-1} \mathbb{X}_{new}^T = \mathbb{X}_{new} \, \sigma^2_{\beta} \, \mathbb{X}_{new}^T$$

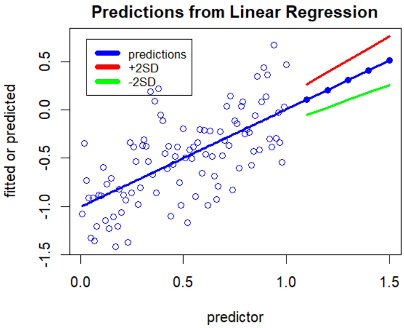
The variance σ^2_{pred} of the *predicted value* is equal to the *predictor* vector multiplied by the *covariance matrix* of the *regression coefficients* σ^2_{β}.

```
> # Inverse of predictor matrix squared
> predictor2 <- MASS::ginv(crossprod(predictor))
> # Define new predictors
> newdata <- (max(predictor[, 2]) + 10*(1:5)/nrows)
> # Calculate the predicted values and standard errors
> predictorn <- cbind(rep(1, NROW(newdata)), newdata)
> predsd <- sqrt(predictorn %*% predictor2 %*% t(predictorn))
> predictv <- cbind(
+   prediction=drop(predictorn %*% betas),
+   stddev=diag(residvar*predsd))
> # Or: Perform loop over predictorn
> predictv <- apply(predictorn, MARGIN=1, function(predictor) {
+   # Calculate predicted values
+   prediction <- predictor %*% betas
+   # Calculate standard deviation
+   predsd <- sqrt(t(predictor) %*% predictor2 %*% predictor)
+   predictsd <- residvar*predsd
+   c(prediction=prediction, stddev=predictsd)
+ })  # end sapply
> predictv <- t(predictv)
```

Confidence Intervals of Regression Predictions

The variables σ_{ε}^2 and σ_y^2 follow the *chi-squared* distribution with $d_{free} = (n - k - 1)$ degrees of freedom, so the *predicted value* y_{pred} follows the *t-distribution*.

```
> # Prepare plot data
> xdata <- c(predictor[,2], newdata)
> xlim <- range(xdata)
> ydata <- c(fittedv, predictv[, 1])
> # Calculate t-quantile
> tquant <- qt(pnorm(2), df=degf)
> predictlow <- predictv[, 1]-tquant*predictv[, 2]
> predicthigh <- predictv[, 1]+tquant*predictv[, 2]
> ylim <- range(c(response, ydata, predictlow, predicthigh))
> # Plot the regression predictions
> plot(x=xdata, y=ydata, xlim=xlim, ylim=ylim,
+      type="l", lwd=3, col="blue",
+      xlab="predictor", ylab="fitted or predicted",
+      main="Predictions from Linear Regression")
> points(x=predictor[,2], y=response, col="blue")
> points(x=newdata, y=predictv[, 1], pch=16, col="blue")
> lines(x=newdata, y=predicthigh, lwd=3, col="red")
> lines(x=newdata, y=predictlow, lwd=3, col="green")
> legend(x="topleft", # Add legend
+        legend=c("predictions", "+2SD", "-2SD"),
+        title=NULL, inset=0.05, cex=0.8, lwd=6,
+        lty=1, col=c("blue", "red", "green"))
```

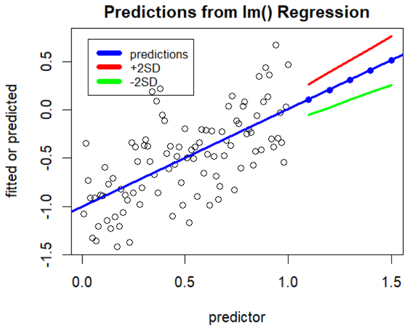


Predictions From *Linear Regression* Using Function `lm()`

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

`predict.lm()` is the predict method for linear models (regressions) produced by the function `lm()`.

```
> # Perform univariate regression
> predictor <- predictor[, 2]
> model <- lm(response ~ predictor)
> # Perform prediction from regression
> newdata <- data.frame(predictor=newdata)
> predictlm <- predict(object=model,
+   newdata=newdata, confl=1-2*(1-pnorm(2)),
+   interval="confidence")
> predictlm <- as.data.frame(predictlm)
> all.equal(predictlm$fit, predictv[, 1])
> all.equal(predictlm$lwr, predictlow)
> all.equal(predictlm$upr, predicthigh)
> plot(response ~ predictor,
+      xlim=range(predictor, newdata),
+      ylim=range(response, predictlm),
+      xlab="predictor", ylab="fitted or predicted",
+      main="Predictions from lm() Regression")
```



```
> abline(model, col="blue", lwd=3)
> with(predictlm, {
+   points(x=newdata$predictor, y=fit, pch=16, col="blue")
+   lines(x=newdata$predictor, y=lwr, lwd=3, col="green")
+   lines(x=newdata$predictor, y=upr, lwd=3, col="red")
+ })  # end with
> legend(x="topleft", # Add legend
+        legend=c("predictions", "+2SD", "-2SD"),
+        title=NULL, inset=0.05, cex=0.8, lwd=6,
+        lty=1, col=c("blue", "red", "green"))
```

Spurious Time Series Regression

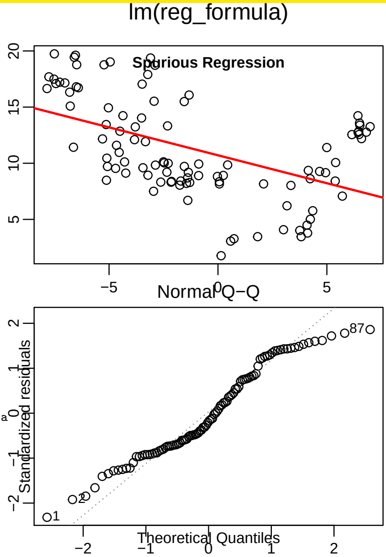
Regression of non-stationary time series creates *spurious* regressions.

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.

```
> predictor <- cumsum(rnorm(100))  # Unit root time series
> response <- cumsum(rnorm(100))
> formulav <- response ˜ predictor
> model <- lm(formulav)  # Perform regression
> # Summary indicates statistically significant regression
> modelsum <- summary(model)
> modelsum$coeff
> modelsum$r.squared
> # Durbin-Watson test shows residuals are autocorrelated
> dw_test <- lmtest::dwtest(model)
> c(dw_test$statistic[[1]], dw_test$p.value)
> plot(formulav, xlab="", ylab="")  # Plot scatterplot using formula
> title(main="Spurious Regression", line=-1)
> # Add regression line
> abline(model, lwd=2, col="red")
> plot(model, which=2, ask=FALSE)  # Plot just Q-Q
```



Multivariate Linear Regression

A *multivariate* linear regression model with k *predictors* 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.

The *residuals* ε_i are assumed to be normally distributed $\phi(0, \sigma_\varepsilon)$, independent, and stationary.

The data consists of n observations, with each observation containing k *predictors* and one *response* value.

The *response vector* y, the *predictor* vectors x_j, and the *residuals* ε are vectors of length n.

The k *predictors* x_j form the columns of the (n, k)-dimensional *predictor matrix* $\mathbb{X}$.

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

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

Where y_{fit} are the *fitted values* of the model.

```
> # Define predictor matrix
> nrows <- 100
> ncols <- 5
> set.seed(1121)  # initialize random number generator
> predictor <- matrix(rnorm(nrows*ncols), ncol=ncols)
> # Add column names
> colnames(predictor) <- paste0("col", 1:ncols)
> # Define the predictor weights
> weights <- sample(3:(ncols+2))
> # Response equals weighted predictor plus random noise
> noise <- rnorm(nrows, sd=5)
> response <- (-3 + 2*predictor %*% weights + noise)
```

Solution of *Multivariate Regression*

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

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

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:

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

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

The matrix $\hat{\mathbb{X}}^{inv}$ is the generalized inverse of the de-meaned *predictor matrix* $\hat{\mathbb{X}}$.

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

```
> # Perform multivariate regression using lm()
> model <- lm(response ~ predictor)
> # Solve multivariate regression using matrix algebra
> # Calculate de-meaned predictor matrix and response vector
> predzm <- t(t(predictor) - colMeans(predictor))
> # predictor <- apply(predictor, 2, function(x) (x-mean(x)))
> respzm <- response - mean(response)
> # Calculate the regression coefficients
> betas <- drop(MASS::ginv(predzm) %*% respzm)
> # Calculate the regression alpha
> alpha <- mean(response) - sum(colSums(predictor)*betas)/nrows
> # Compare with coefficients from lm()
> all.equal(coef(model), c(alpha, betas), check.attributes=FALSE)
[1] TRUE
> # Compare with actual coefficients
> all.equal(c(-1, weights), c(alpha, betas), check.attributes=FALSE
[1] "Mean relative difference: 1.07"
```

Multivariate Regression in Homogeneous Form

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

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

Where the *regression coefficients* β now contain the intercept α: $\beta = (\alpha, \beta_1, \ldots, \beta_k)$, and the *predictor matrix* $\mathbb{X}$ has $k + 1$ columns and n rows.

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

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

The matrix $\mathbb{X}_{inv} = (\mathbb{X}^T \mathbb{X})^{-1} \mathbb{X}^T$ is the generalized inverse of the *predictor matrix* $\mathbb{X}$.

The coefficients β can be interpreted as the projections of the *response vector* y onto the columns of the *predictor matrix* $\mathbb{X}$.

The *predictor matrix* $\mathbb{X}$ maps the *regression coefficients* β into the *response vector* y.

The generalized inverse of the *predictor matrix* $\mathbb{X}_{inv}$ maps the *response vector* y into the *regression coefficients* β.

```
> # Add intercept column to predictor matrix
> predictor <- cbind(rep(1, NROW(predictor)), predictor)
> ncols <- NCOL(predictor)
> # Add column name
> colnames(predictor)[1] <- "intercept"
> # Calculate generalized inverse of the predictor matrix
> invpred <- MASS::ginv(predictor)
> # Calculate the regression coefficients
> betas <- invpred %*% response
> # Perform multivariate regression without intercept term
> model <- lm(response ~ predictor - 1)
> all.equal(drop(betas), coef(model), check.attributes=FALSE)
[1] TRUE
```

The *Residuals* of Multivariate Regression

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

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

Where y_{fit} are the *fitted values* of the model.

The *residuals* are equal to the *response vector* minus the *fitted values*: $\varepsilon = y - y_{fit}$.

The *residuals* ε are orthogonal to the columns of the *predictor matrix* $\mathbb{X}$ (the *predictors*):

$$\varepsilon^T \mathbb{X} = (y - \mathbb{X}(\mathbb{X}^T\mathbb{X})^{-1}\mathbb{X}^T y)^T \mathbb{X} =$$
$$y^T\mathbb{X} - y^T\mathbb{X}(\mathbb{X}^T\mathbb{X})^{-1}\mathbb{X}^T\mathbb{X} = y^T\mathbb{X} - y^T\mathbb{X} = 0$$

Therefore the *residuals* are also orthogonal to the *fitted values*: $\varepsilon^T y_{fit} = \varepsilon^T \mathbb{X}\beta = 0$.

Since the first column of the *predictor matrix* $\mathbb{X}$ is a unit vector, the *residuals* ε have zero mean: $\varepsilon^T \mathbb{1} = 0$.

```
> # Calculate fitted values from regression coefficients
> fittedv <- drop(predictor %*% betas)
> all.equal(fittedv, model$fitted.values, check.attributes=FALSE)
[1] TRUE
> # Calculate the residuals
> residuals <- drop(response - fittedv)
> all.equal(residuals, model$residuals, check.attributes=FALSE)
[1] TRUE
> # Residuals are orthogonal to predictor columns (predictors)
> sapply(residuals %*% predictor, all.equal, target=0)
[1] TRUE TRUE TRUE TRUE TRUE TRUE
> # Residuals are orthogonal to the fitted values
> all.equal(sum(residuals*fittedv), target=0)
[1] TRUE
> # Sum of residuals is equal to zero
> all.equal(sum(residuals), target=0)
[1] TRUE
```

The *Influence Matrix* of Multivariate Regression

The vector $y_{fit} = \mathbb{X}\beta$ are the *fitted values* corresponding to the *response vector* y:

$$y_{fit} = \mathbb{X}\beta = \mathbb{X}(\mathbb{X}^T\mathbb{X})^{-1}\mathbb{X}^T y = \mathbb{X}\mathbb{X}_{inv} y = \mathbb{H} y$$

Where $\mathbb{H} = \mathbb{X}\mathbb{X}_{inv} = \mathbb{X}(\mathbb{X}^T\mathbb{X})^{-1}\mathbb{X}^T$ is the *influence matrix* (or hat matrix), which maps the *response vector* y into the *fitted values* y_{fit}.

The *influence matrix* $\mathbb{H}$ is a projection matrix, and it measures the changes in the *fitted values* y_{fit} due to changes in the *response vector* y.

$$\mathbb{H}_{ij} = \frac{\partial y_i^{fit}}{\partial y_j}$$

The square of the *influence matrix* $\mathbb{H}$ is equal to itself (it's idempotent): $\mathbb{H}\,\mathbb{H}^T = \mathbb{H}$.

```
> # Calculate the influence matrix
> influencem <- predictor %*% invpred
> # The influence matrix is idempotent
> all.equal(influencem, influencem %*% influencem)
[1] TRUE
> # Calculate fitted values using influence matrix
> fittedv <- drop(influencem %*% response)
> all.equal(fittedv, model$fitted.values, check.attributes=FALSE)
[1] TRUE
> # Calculate fitted values from regression coefficients
> fittedv <- drop(predictor %*% betas)
> all.equal(fittedv, model$fitted.values, check.attributes=FALSE)
[1] TRUE
```

Multivariate Regression With de-Meaned Variables

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

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

The intercept α can be substituted with its solution: $\alpha = \bar{y} - \bar{\mathbb{X}}\beta$ to obtain the regression model with de-meaned response and predictor matrix:

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

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

The regression model with a de-meaned *predictor matrix* produces the same *fitted values* (only shifted by their mean) and *residuals* as the original regression model, so it's equivalent to it. has the same influence matrix, and

But the de-meaned regression model has a different *influence matrix*, which maps the de-meaned *response vector* $\hat{y}$ into the de-meaned *fitted values* $\hat{y}_{fit}$.

```
> # Calculate zero mean fitted values
> predzm <- t(t(predictor) - colMeans(predictor))
> fitted_zm <- drop(predzm %*% betas)
> all.equal(fitted_zm,
+   model$fitted.values - mean(response),
+   check.attributes=FALSE)
[1] TRUE
> # Calculate the residuals
> respzm <- response - mean(response)
> residuals <- drop(respzm - fitted_zm)
> all.equal(residuals, model$residuals,
+   check.attributes=FALSE)
[1] TRUE
> # Calculate the influence matrix
> influence_zm <- predzm %*% MASS::ginv(predzm)
> # Compare the fitted values
> all.equal(fitted_zm,
+   drop(influence_zm %*% respzm),
+   check.attributes=FALSE)
[1] TRUE
```

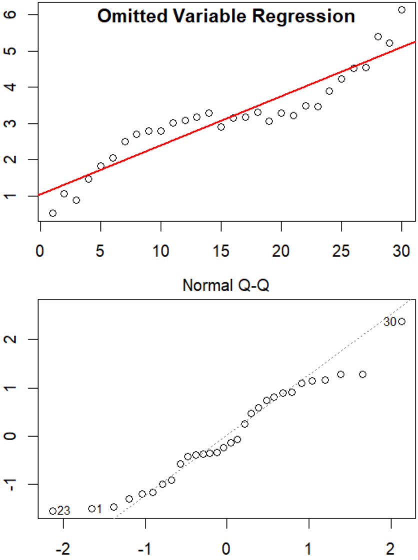
Omitted Variable Bias

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 that the residuals are autocorrelated, which means that the regression coefficients may not be statistically significant (different from zero).

```
> library(lmtest)  # Load lmtest
> # Define predictor matrix
> predictor <- 1:30
> omitv <- sin(0.2*1:30)
> # Response depends on both predictors
> response <- 0.2*predictor + omitv + 0.2*rnorm(30)
> # Mis-specified regression only one predictor
> model_ovb <- lm(response ~ predictor)
> modelsum <- summary(model_ovb)
> modelsum$coeff
> modelsum$r.squared
> # Durbin-Watson test shows residuals are autocorrelated
> lmtest::dwtest(model_ovb)
> # Plot the regression diagnostic plots
> x11(width=5, height=7)
> par(mfrow=c(2,1))  # Set plot panels
> par(mar=c(3, 2, 1, 1), oma=c(1, 0, 0, 0))
> plot(response ~ predictor)
> abline(model_ovb, lwd=2, col="red")
> title(main="Omitted Variable Regression", line=-1)
> plot(model_ovb, which=2, ask=FALSE)  # Plot just Q-Q
```



Regression Coefficients as *Random Variables*

The *residuals* $\hat{\varepsilon}$ can be considered to be *random variables*, with expected value equal to zero $\mathbb{E}[\hat{\varepsilon}] = 0$, and variance equal to σ_{ε}^2.

The variance of the *residuals* is equal to the expected value of the squared *residuals* divided by the number of *degrees of freedom*:

$$\sigma_{\varepsilon}^2 = \frac{\mathbb{E}[\varepsilon^T \varepsilon]}{d_{free}}$$

Where $d_{free} = (n - k)$ is the number of *degrees of freedom* of the *residuals*, equal to the number of observations n, minus the number of *predictors* k (including the intercept term).

The *response vector* y can also be considered to be a *random variable* $\hat{y}$, equal to the sum of the deterministic *fitted values* y_{fit} plus the random *residuals* $\hat{\varepsilon}$:

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

The *regression coefficients* β can also be considered to be *random variables* $\hat{\beta}$:

$$\hat{\beta} = \mathbb{X}_{inv}\hat{y} = \mathbb{X}_{inv}(y_{fit} + \hat{\varepsilon}) =$$

$$(\mathbb{X}^T\mathbb{X})^{-1}\mathbb{X}^T(\mathbb{X}\beta + \hat{\varepsilon}) = \beta + \mathbb{X}_{inv}\hat{\varepsilon}$$

Where β is equal to the expected value of $\hat{\beta}$: $\beta = \mathbb{E}[\hat{\beta}] = \mathbb{X}_{inv}y_{fit} = \mathbb{X}_{inv}y$.

```
> # Regression model summary
> modelsum <- summary(model)
> # Degrees of freedom of residuals
> nrows <- NROW(predictor)
> ncols <- NCOL(predictor)
> degf <- (nrows - ncols)
> all.equal(degf, modelsum$df[2])
[1] TRUE
> # Variance of residuals
> residvar <- sum(residuals^2)/degf
```

Covariance Matrix of the Regression Coefficients

The *covariance matrix* of the *regression coefficients* $\hat{\beta}$ is given by:

$$\sigma^2_\beta = \frac{\mathbb{E}[(\hat{\beta} - \beta)(\hat{\beta} - \beta)^T]}{d_{free}} =$$

$$\frac{\mathbb{E}[\mathbb{X}_{inv}\hat{\varepsilon}(\mathbb{X}_{inv}\hat{\varepsilon})^T]}{d_{free}} = \frac{\mathbb{E}[\mathbb{X}_{inv}\hat{\varepsilon}\hat{\varepsilon}^T\mathbb{X}_{inv}^T]}{d_{free}} =$$

$$\frac{(\mathbb{X}^T\mathbb{X})^{-1}\mathbb{X}^T\mathbb{E}[\hat{\varepsilon}\hat{\varepsilon}^T]\,\mathbb{X}(\mathbb{X}^T\mathbb{X})^{-1}}{d_{free}} =$$

$$(\mathbb{X}^T\mathbb{X})^{-1}\mathbb{X}^T\sigma^2_\varepsilon\mathbb{1}\,\mathbb{X}(\mathbb{X}^T\mathbb{X})^{-1} = \sigma^2_\varepsilon(\mathbb{X}^T\mathbb{X})^{-1}$$

Where the expected values of the squared residuals are proportional to the diagonal unit matrix $\mathbb{1}$:
$\frac{\mathbb{E}[\hat{\varepsilon}\hat{\varepsilon}^T]}{d_{free}} = \sigma^2_\varepsilon\mathbb{1}$

If any of the predictor matrix columns are close to being *collinear*, then the squared predictor matrix becomes singular, and the covariance of their regression coefficients becomes very large.

The matrix $\mathbb{X}_{inv} = (\mathbb{X}^T\mathbb{X})^{-1}\mathbb{X}^T$ is the generalized inverse of the *predictor matrix* $\mathbb{X}$.

```
> # Inverse of predictor matrix squared
> predictor2 <- MASS::ginv(crossprod(predictor))
> # predictor2 <- t(predictor) %*% predictor
> # Variance of residuals
> residvar <- sum(residuals^2)/degf
> # Calculate covariance matrix of betas
> beta_covar <- residvar*predictor2
> # Round(beta_covar, 3)
> betasd <- sqrt(diag(beta_covar))
> all.equal(betasd, modelsum$coeff[, 2], check.attributes=FALSE)
[1] TRUE
> # Calculate t-values of betas
> beta_tvals <- drop(betas)/betasd
> all.equal(beta_tvals, modelsum$coeff[, 3], check.attributes=FALSE)
[1] TRUE
> # Calculate two-sided p-values of betas
> beta_pvals <- 2*pt(-abs(beta_tvals), df=degf)
> all.equal(beta_pvals, modelsum$coeff[, 4], check.attributes=FALSE)
[1] TRUE
> # The square of the generalized inverse is equal
> # to the inverse of the square
> all.equal(MASS::ginv(crossprod(predictor)),
+   invpred %*% t(invpred))
[1] TRUE
```

Covariance Matrix of the Fitted Values

The *fitted values* y_{fit} can also be considered to be *random variables* $\hat{y}_{fit}$, because the *regression coefficients* $\hat{\beta}$ are *random variables*:
$\hat{y}_{fit} = \mathbb{X}\hat{\beta} = \mathbb{X}(\beta + \mathbb{X}_{inv}\hat{\varepsilon}) = y_{fit} + \mathbb{X}\mathbb{X}_{inv}\hat{\varepsilon}$.

The *covariance matrix* of the *fitted values* σ^2_{fit} is:

$$\sigma^2_{fit} = \frac{\mathbb{E}[\mathbb{X}\mathbb{X}_{inv}\hat{\varepsilon}\,(\mathbb{X}\mathbb{X}_{inv}\hat{\varepsilon})^T]}{d_{free}} = \frac{\mathbb{E}[\mathbb{H}\,\hat{\varepsilon}\hat{\varepsilon}^T\,\mathbb{H}^T]}{d_{free}} =$$

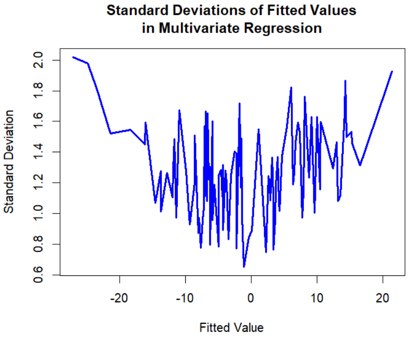
$$\frac{\mathbb{H}\,\mathbb{E}[\hat{\varepsilon}\hat{\varepsilon}^T]\,\mathbb{H}^T}{d_{free}} = \sigma^2_{\varepsilon}\,\mathbb{H} = \sigma^2_{\varepsilon}\,\mathbb{X}(\mathbb{X}^T\mathbb{X})^{-1}\mathbb{X}^T$$

The square of the *influence matrix* $\mathbb{H}$ is equal to itself (it's idempotent): $\mathbb{H}\,\mathbb{H}^T = \mathbb{H}$.

The variance of the *fitted values* σ^2_{fit} increases with the distance of the *predictors* from their mean values.

This is because the *fitted values* farther from their mean are more sensitive to the variance of the regression slope.

```
> # Calculate the influence matrix
> influencem <- predictor %*% invpred
> # The influence matrix is idempotent
> all.equal(influencem, influencem %*% influencem)
```



```
> # Calculate covariance and standard deviations of fitted values
> fitcovar <- residvar*influencem
> fitsd <- sqrt(diag(fitcovar))
> # Sort the standard deviations
> fitsd <- cbind(fitted=fittedv, stddev=fitsd)
> fitsd <- fitsd[order(fittedv), ]
> # Plot the standard deviations
> plot(fitsd, type="l", lwd=3, col="blue",
+      xlab="Fitted Value", ylab="Standard Deviation",
+      main="Standard Deviations of Fitted Values\nin Multivariate R
```

Standard Errors of Time Series Regression

Bootstrapping the regression of asset returns shows that the actual standard errors can be over twice as large as those reported by the function lm().

This is because the function lm() assumes that the data is normally distributed, while in reality asset returns have very large skewness and kurtosis.

```
> # Load time series of ETF percentage returns
> retsp <- rutils::etfenv$returns[, c("XLF", "XLE")]
> retsp <- na.omit(retsp)
> nrows <- NROW(retsp)
> head(retsp)
> # Define regression formula
> formulav <- paste(colnames(retsp)[1],
+   paste(colnames(retsp)[-1], collapse="+"),
+   sep=" ˜ ")
> # Standard regression
> model <- lm(formulav, data=retsp)
> modelsum <- summary(model)
> # Bootstrap of regression
> set.seed(1121)  # initialize random number generator
> bootd <- sapply(1:100, function(x) {
+   samplev <- sample.int(nrows, replace=TRUE)
+   model <- lm(formulav, data=retsp[samplev, ])
+   model$coefficients
+ })  # end sapply
> # Means and standard errors from regression
> modelsum$coefficients
> # Means and standard errors from bootstrap
> dim(bootd)
> t(apply(bootd, MARGIN=1,
+ function(x) c(mean=mean(x), stderror=sd(x))))
```

Predictions From *Multivariate Regression* Models

The prediction y_{pred} from a regression model is equal to the *response value* corresponding to the *predictor* vector with the new data $\mathbb{X}_{new}$:

$$y_{pred} = \mathbb{X}_{new} \, \beta$$

The prediction is a *random variable* $\hat{y}_{pred}$, because the *regression coefficients* $\hat{\beta}$ are *random variables*:

$$\hat{y}_{pred} = \mathbb{X}_{new} \hat{\beta} = \mathbb{X}_{new} (\beta + \mathbb{X}_{inv} \hat{\varepsilon}) = \\ y_{pred} + \mathbb{X}_{new} \mathbb{X}_{inv} \hat{\varepsilon}$$

The variance σ^2_{pred} of the *predicted value* is:

$$\sigma^2_{pred} = \frac{\mathbb{E}[\mathbb{X}_{new} \mathbb{X}_{inv} \hat{\varepsilon} \, (\mathbb{X}_{new} \mathbb{X}_{inv} \hat{\varepsilon})^T]}{d_{free}} = \\ \frac{\mathbb{E}[\mathbb{X}_{new} \mathbb{X}_{inv} \hat{\varepsilon} \hat{\varepsilon}^T \mathbb{X}_{inv}^T \mathbb{X}_{new}^T]}{d_{free}} = \\ \sigma^2_{\varepsilon} \mathbb{X}_{new} \mathbb{X}_{inv} \mathbb{X}_{inv}^T \mathbb{X}_{new}^T = \\ \sigma^2_{\varepsilon} \, \mathbb{X}_{new} (\mathbb{X}^T \mathbb{X})^{-1} \mathbb{X}_{new}^T = \mathbb{X}_{new} \, \sigma^2_{\beta} \, \mathbb{X}_{new}^T$$

The variance σ^2_{pred} of the *predicted value* is equal to the *predictor* vector multiplied by the *covariance matrix* of the *regression coefficients* σ^2_{β}.

```
> # New data predictor is a data frame or row vector
> set.seed(1121)
> newdata <- data.frame(matrix(c(1, rnorm(5)), nr=1))
> colnamev <- colnames(predictor)
> colnames(newdata) <- colnamev
> newdatav <- as.matrix(newdata)
> prediction <- drop(newdatav %*% betas)
> predsd <- drop(sqrt(newdatav %*% beta_covar %*% t(newdatav)))
```

Predictions From *Multivariate Regression* Using `lm()`

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

`predict.lm()` is the predict method for linear models (regressions) produced by the function `lm()`.

In order for `predict.lm()` to work properly, the multivariate regression must be specified using a formula.

```
> # Create formula from text string
> formulav <- paste0("response ~ ",
+   paste(colnames(predictor), collapse=" + "), " - 1")
> # Specify multivariate regression using formula
> model <- lm(formulav, data=data.frame(cbind(response, predictor))
> modelsum <- summary(model)
> # Predict from lm object
> predictlm <- predict.lm(object=model, newdata=newdata,
+    interval="confidence", confl=1-2*(1-pnorm(2)))
> # Calculate t-quantile
> tquant <- qt(pnorm(2), df=degf)
> predicthigh <- (prediction + tquant*predsd)
> predictlow <- (prediction - tquant*predsd)
> # Compare with matrix calculations
> all.equal(predictlm[1, "fit"], prediction)
[1] TRUE
> all.equal(predictlm[1, "lwr"], predictlow)
[1] "Mean relative difference: 0.00185"
> all.equal(predictlm[1, "upr"], predicthigh)
[1] "Mean relative difference: 0.00157"
```

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:

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

$$ESS = (y_{fit} - \bar{y})^T (y_{fit} - \bar{y})$$

$$RSS = (y - y_{fit})^T (y - y_{fit})$$

Since the *residuals* $\varepsilon = y - y_{fit}$ are orthogonal to the *fitted values* y_{fit}, they are also orthogonal to the *fitted* excess values $(y_{fit} - \bar{y})$:

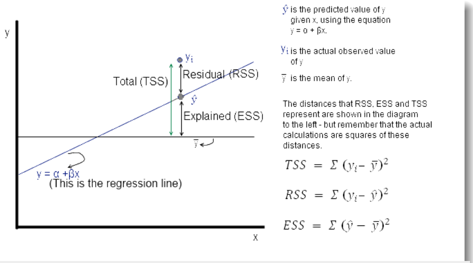
$$(y - y_{fit})^T (y_{fit} - \bar{y}) = 0$$

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

$$TSS = ESS + RSS$$

It also follows that the *RSS* and the *ESS* follow independent *chi-squared* distributions with $(n - k)$ and $(k - 1)$ degrees of freedom.

The degrees of freedom of the *Total Sum of Squares* is equal to the sum of the *RSS* plus the *ESS*: $d_{free}^{TSS} = (n - k) + (k - 1) = n - 1$.



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

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 actual calculations are squares of these distances.

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

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

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

```
> # TSS = ESS + RSS
> tss <- sum((response-mean(response))^2)
> ess <- sum((fittedv-mean(fittedv))^2)
> rss <- sum(residuals^2)
> all.equal(tss, ess + rss)
[1] TRUE
```

R-squared of Multivariate Regression

The *R-squared* is the fraction of the *Explained Sum of Squares* (*ESS*) divided by the *Total Sum of Squares* (*TSS*):

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

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*:

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

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

```
> # Set regression attribute for intercept
> attributes(model$terms)$intercept <- 1
> # Regression summary
> modelsum <- summary(model)
> # Regression R-squared
> rsquared <- ess/tss
> all.equal(rsquared, modelsum$r.squared)
[1] TRUE
> # Correlation between response and fitted values
> cor_fitted <- drop(cor(response, fittedv))
> # Squared correlation between response and fitted values
> all.equal(cor_fitted^2, rsquared)
[1] TRUE
```

Adjusted R-squared of Multivariate Regression

The weakness of *R-squared* is that it increases with the number of predictors (even for predictors which are purely random), so it may provide an inflated measure of the quality of a model with many predictors.

This is remedied by using the *residual variance* ($\sigma_\varepsilon^2 = \frac{RSS}{d_{free}}$) instead of the *RSS*, and the *response variance* ($\sigma_y^2 = \frac{TSS}{n-1}$) instead of the *TSS*.

The *adjusted R-squared* is equal to 1 minus the fraction of the *residual variance* divided by the *response variance*:

$$R_{adj}^2 = 1 - \frac{\sigma_\varepsilon^2}{\sigma_y^2} = 1 - \frac{RSS/d_{free}}{TSS/(n-1)}$$

Where $d_{free} = (n - k)$ is the number of *degrees of freedom* of the *residuals*.

The *adjusted R-squared* is always smaller than the *R-squared*.

The performance of two different models can be compared by comparing their *adjusted R-squared*, since the model with the larger *adjusted R-squared* has a smaller *residual variance*, so it's better able to explain the *response*.

```
> nrows <- NROW(predictor)
> ncols <- NCOL(predictor)
> # Degrees of freedom of residuals
> degf <- (nrows - ncols)
> # Adjusted R-squared
> rsquared_adj <- (1-sum(residuals^2)/degf/var(response))
> # Compare adjusted R-squared from lm()
> all.equal(drop(rsquared_adj), modelsum$adj.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 *F-statistic* random variable:

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

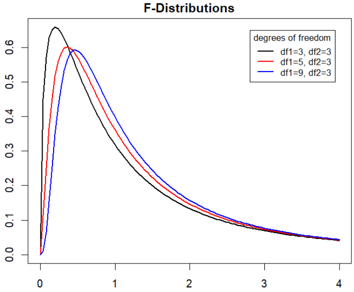
Follows the *F-distribution* with m and n degrees of freedom, with the probability density function:

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

The *F-distribution* depends on the *F-statistic* F and also on the degrees of freedom, m and n.

The function `df()` calculates the probability density of the *F-distribution*.

```
> # Plot three curves in loop
> degf <- c(3, 5, 9)  # Degrees of freedom
> colors <- c("black", "red", "blue", "green")
> for (it in 1:NROW(degf)) {
+ curve(expr=df(x, df1=degf[it], df2=3),
+ xlim=c(0, 4), xlab="", ylab="", lwd=2,
+ col=colors[it], add=as.logical(it-1))
+ }  # end for
```



```
> # Add title
> title(main="F-Distributions", line=0.5)
> # Add legend
> labelv <- paste("df", degf, sep="=")
> legend("topright", inset=0.05, title="degrees of freedom",
+        labelv, cex=0.8, lwd=2, lty=1, col=colors)
```

The *F-test* For the Variance Ratio

Let x and y be independent standard *Normal* variables, and let $\sigma_x^2 = \frac{1}{m-1} \sum_{i=1}^{m} (x_i - \bar{x})^2$ and $\sigma_y^2 = \frac{1}{n-1} \sum_{i=1}^{n} (y_i - \bar{y})^2$ be their sample variances.

The ratio $F = \sigma_x^2 / \sigma_y^2$ of the sample variances follows the *F-distribution* with m and n degrees of freedom.

The *F-test* tests the *null hypothesis* that the *F-statistic* F is not significantly greater than 1 (the variance σ_x^2 is not significantly greater than σ_y^2).

A large value of the *F-statistic* F indicates that the variances are unlikely to be equal.

The function `pf(q)` returns the cumulative probability of the *F-distribution*, i.e. the cumulative probability that the *F-statistic* F is less than the quantile q.

This *F-test* is very sensitive to the assumption of the normality of the variables.

```
> sigmax <- var(rnorm(nrows))
> sigmay <- var(rnorm(nrows))
> fratio <- sigmax/sigmay
> # Cumulative probability for q = fratio
> pf(fratio, nrows-1, nrows-1)
[1] 0.0642
> # p-value for fratios
> 1-pf((10:20)/10, nrows-1, nrows-1)
 [1] 0.500000 0.318150 0.182964 0.096784 0.047876 0.022467 0.010123
 [9] 0.001888 0.000793 0.000329
```

The *F-statistic* for Linear Regression

The performance of two different regression models can be compared by directly comparing their *Residual Sum of Squares* (*RSS*), since the model with a smaller *RSS* is better able to explain the *response*.

Let the *restricted* model have $p1$ parameters with $df1 = n - p1$ degrees of freedom, and the *unrestricted* model have $p2$ parameters with $df2 = n - p2$ degrees of freedom, with $p2 < p1$.

Then the F-statistic F, defined as the ratio of the scaled *Residual Sum of Squares*:

$$F = \frac{(RSS1 - RSS2)/(df1 - df2)}{RSS2/df2}$$

Follows the *F-distribution* with $(p2 - p1)$ and $(n - p2)$ degrees of freedom (assuming that the *residuals* are normally distributed).

If the *restricted* model only has one parameter (the constant intercept term), then $df1 = n - 1$, and its *fitted values* are equal to the average of the *response*: $y_i^{fit} = \bar{y}$, so $RSS1$ is equal to the TSS: $RSS1 = TSS = (y - \bar{y})^2$, so its *Explained Sum of Squares* is equal to zero: $ESS1 = TSS - RSS1 = 0$.

Let the *unrestricted* multivariate regression model be defined as:

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

Where y is the *response*, $\mathbb{X}$ is the *predictor matrix* (with k *predictors*, including the intercept term), and β are the k *regression coefficients*.

So the *unrestricted* model has k parameters ($p2 = k$), and $RSS2 = RSS$ and $ESS2 = ESS$, and then the F-statistic can be written as:

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

The *F-test* for Linear Regression

The *Residual Sum of Squares* $RSS = \varepsilon^T \varepsilon$ and the *Explained Sum of Squares* $ESS = (y_{fit} - \bar{y})^T (y_{fit} - \bar{y})$ follow independent *chi-squared* distributions with $(n - k)$ and $(k - 1)$ degrees of freedom.

Then the *F*-statistic, equal to the ratio of the *ESS* divided by *RSS*:

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

Follows the *F-distribution* with $(k - 1)$ and $(n - k)$ degrees of freedom (assuming that the *residuals* are normally distributed).

```
> # F-statistic from lm()
> modelsum$fstatistic
value numdf dendf
  391     5    94
> # Degrees of freedom of residuals
> degf <- (nrows - ncols)
> # F-statistic from ESS and RSS
> fstat <- (ess/(ncols-1))/(rss/degf)
> all.equal(fstat, modelsum$fstatistic[1], check.attributes=FALSE)
[1] TRUE
> # p-value of F-statistic
> 1-pf(q=fstat, df1=(ncols-1), df2=(nrows-ncols))
[1] 0
```

Regularized Inverse of Rectangular Matrices

The *SVD* of a rectangular matrix $\mathbb{A}$ is defined as the factorization:

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

Where $\mathbb{U}$ and $\mathbb{V}$ are the *singular matrices*, and Σ is a diagonal matrix of *singular values*.

The *generalized inverse* matrix $\mathbb{A}^{-1}$ satisfies the inverse equation: $\mathbb{A}\mathbb{A}^{-1}\mathbb{A} = \mathbb{A}$, and it can be expressed as a product of the *SVD* matrices as follows:

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

If any of the *singular values* are zero then the *generalized inverse* does not exist.

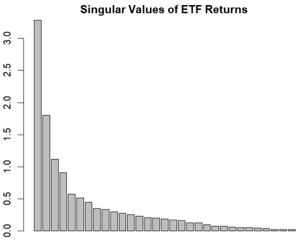
The *regularized inverse* is obtained by removing very small *singular values*:

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

Where $\mathbb{U}_n$, $\mathbb{V}_n$ and Σ_n are the *SVD* matrices without very small *singular values*.

The regularized inverse satisfies the inverse equation only approximately (it has *bias*), but it's often used in machine learning because it has lower *variance* than the exact inverse.

```
> # Calculate ETF returns
> retsp <- na.omit(rutils::etfenv$returns)
> # Perform singular value decomposition
> svdec <- svd(retsp)
> barplot(svdec$d, main="Singular Values of ETF Returns")
```



```
> # Calculate generalized inverse from SVD
> invmat <- svdec$v %*% (t(svdec$u) / svdec$d)
> # Verify inverse property of inverse
> all.equal(zoo::coredata(retsp), retsp %*% invmat %*% retsp)
> # Calculate regularized inverse from SVD
> dimax <- 1:3
> invreg <- svdec$v[, dimax] %*%
+   (t(svdec$u[, dimax]) / svdec$d[dimax])
> # Calculate regularized inverse using RcppArmadillo
> invcpp <- HighFreq::calc_inv(retsp, dimax=3)
> all.equal(invreg, invcpp, check.attributes=FALSE)
> # Calculate regularized inverse from Moore-Penrose pseudo-inverse
> retsq <- t(retsp) %*% retsp
> eigend <- eigen(retsq)
> squared_inv <- eigend$vectors[, dimax] %*%
+   (t(eigend$vectors[, dimax]) / eigend$values[dimax])
> invmp <- squared_inv %*% t(retsp)
> all.equal(invreg, invmp, check.attributes=FALSE)
```

Linear Transformation of the Predictor Matrix

A *multivariate* linear regression model can be transformed by replacing its *predictors* x_j with their own linear combinations.

This is equivalent to multiplying the *predictor matrix* $\mathbb{X}$ by a transformation matrix $\mathbb{W}$:

$$\mathbb{X}_{trans} = \mathbb{X} \, \mathbb{W}$$

The transformed *predictor matrix* $\mathbb{X}_{trans}$ produces the same *influence matrix* $\mathbb{H}$ as the original *predictor matrix* $\mathbb{X}$:

$$\begin{aligned}
&\mathbb{H}_{trans} = \mathbb{X}_{trans}(\mathbb{X}^T_{trans}\mathbb{X}_{trans})^{-1}\mathbb{X}^T_{trans} = \\
&\mathbb{X}\mathbb{W}(\mathbb{W}^T\mathbb{X}^T\mathbb{X}\mathbb{W})^{-1}\mathbb{W}^T\mathbb{X}^T = \\
&\mathbb{X}\mathbb{W}\mathbb{W}^{-1}(\mathbb{X}^T\mathbb{X})^{-1}\mathbb{W}^{T^{-1}}\mathbb{W}^T\mathbb{X}^T = \\
&\mathbb{X}(\mathbb{X}^T\mathbb{X})^{-1}\mathbb{X}^T = \mathbb{H}
\end{aligned}$$

Since the *influence matrix* $\mathbb{H}$ is the same, the transformed regression model produces the same *fitted values* and *residuals* as the original regression model, so it's equivalent to it.

```
> # Define transformation matrix
> trans_mat <- matrix(runif(ncols^2, min=(-1), max=1), ncol=ncols)
> # Calculate linear combinations of predictor columns
> predictor_trans <- predictor %*% trans_mat
> # Calculate the influence matrix
> influence_trans <- predictor_trans %*% MASS::ginv(predictor_trans)
> # Compare the influence matrices
> all.equal(influencem, influence_trans)
[1] TRUE
```

Homework Assignment

Required

- Study all the lecture slides in *FRE7241_Lecture_4.pdf*, and run all the code in *FRE7241_Lecture_4.R*

Recommended

- Download from NYU Classes and read about momentum strategies:
  Moskowitz Time Series Momentum.pdf
  Bouchaud Momentum Mean Reversion Equity Returns.pdf
  Hurst Pedersen AQR Momentum Evidence.pdf