FRE6871 R in Finance

Lecture#5, Fall 2023

Jerzy Pawlowski *jp3900@nyu.edu*

NYU Tandon School of Engineering

October 10, 2023

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
> # Initialize the random number generator
> set.seed(1121, "Mersenne-Twister", sample.kind="Rejection")
> predm <- matrix(runif(nrows*ncols), ncol=ncols)
> # Add column names
> colnames(predm) <- paste0("pred", 1:ncols)
> # Define the predictor weights
> weightv <- runif(3:(ncols+2), min=(-1), max=1)
> # Response equals weighted predictor plus random noise
> noisev <- rnorm(nrows, sd=2)
> respv <- (1 + predm %*% weightv + noisev)
```

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 centered (de-meaned) variables.

The matrix $\hat{\mathbb{X}}^{inv}$ is the generalized inverse of the centered (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()
> regmod <- lm(respv ~ predm)
> # Solve multivariate regression using matrix algebra
> # Calculate centered (de-meaned) predictor matrix and response ve
> predc <- t(t(predm) - colMeans(predm))
> # predm <- apply(predm, 2, function(x) (x-mean(x)))
> respc <- respv - mean(respv)
> # Calculate the regression coefficients
> betav <- drop(MASS::ginv(predc) %*% respc)
> # Calculate the regression alpha
> alpha <- mean(respv) - sum(colSums(predm)*betav)/nrows
> # Compare with coefficients from lm()
> all.equal(coef(regmod), c(alpha, betav), check.attributes=FALSE)
[1] TRUE
> # Compare with actual coefficients
> all.equal(c(-1, weightv), c(alpha, betav), check.attributes=FALSE
[1] "Mean relative difference: 1.42"
```

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
> predm <- cbind(rep(1, nrows), predm)
> ncols <- NCOL(predm)
> # Add column name
> colnames(predm)[1] <- "intercept"
> # Calculate generalized inverse of the predictor matrix
> predinv <- MASS::ginv(predm)
> # Calculate the regression coefficients
> betav <- predinv %*% respv
> # Perform multivariate regression without intercept term
> regmod <- lm(respv ~ predm - 1)
> all.equal(drop(betav), coef(regmod), 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
> fitv <- drop(predm %*% betav)
> all.equal(fitv, regmod$fitted.values, check.attributes=FALSE)
[1] TRUE
> # Calculate the residuals
> resids <- drop(respv - fitv)
> all.equal(resids, regmod$residuals, check.attributes=FALSE)
[1] TRUE
> # Residuals are orthogonal to predictor columns (predms)
> sapply(resids %*% predm, all.equal, target=0)
[1] TRUE TRUE TRUE TRUE TRUE TRUE
> # Residuals are orthogonal to the fitted values
> all.equal(sum(resids*fitv), target=0)
[1] TRUE
> # Sum of residuals is equal to zero
> all.equal(sum(resids), 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
> infmat <- predm %*% predinv
> # The influence matrix is idempotent
> all.equal(infmat, infmat %*% infmat)
[1] TRUE
> # Calculate fitted values using influence matrix
> fitv <- drop(infmat %*% respv)
> all.equal(fitv, regmod$fitted.values, check.attributes=FALSE)
[1] TRUE
> # Calculate fitted values from regression coefficients
> fitv <- drop(predm %*% betav)
> all.equal(fitv, regmod$fitted.values, check.attributes=FALSE)
[1] TRUE
```

Multivariate Regression With Centered 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 centered (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 centered (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.

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

```
> # Calculate centered (de-meaned) fitted values
> predc <- t(t(predm) - colMeans(predm))
> fittedc <- drop(predc %*% betav)
> all.equal(fittedc,
+   regmod$fitted.values - mean(respv),
+   check.attributes=FALSE)
[1] TRUE
> # Calculate the residuals
> respc <- respv - mean(respv)
> resids <- drop(respc - fittedc)
> all.equal(resids, regmod$residuals,
+   check.attributes=FALSE)
[1] TRUE
> # Calculate the influence matrix
> infmatc <- predc %*% MASS::ginv(predc)
> # Compare the fitted values
> all.equal(fittedc, drop(infmatc %*% respc), check.attributes=FALSE
[1] TRUE
```

Multivariate Regression for Orthogonal Predictors

The generalized inverse can be written as:

$$\mathbb{X}_{inv} = (\mathbb{X}^T \mathbb{X})^{-1} \mathbb{X}^T = \mathbb{C}^{-1} \mathbb{X}^T$$

Where $\mathbb{C} = \mathbb{X}^T \mathbb{X}$ is the matrix of inner products of the predictors $\mathbb{X}$.

If the predictors are orthogonal ($x_i \cdot x_j = 0$ for $i \neq j$, and $x_i \cdot x_i = \sigma_i^2$) then the squared predictor matrix $\mathbb{C}$ is diagonal:

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

And the inverse of the squared predictor matrix $\mathbb{C}^{-1}$ is also diagonal, so the *regression coefficients* can then be written simply as:

$$\beta_i = \frac{x_i \cdot y}{\sigma_i^2}$$

Where $x_i \cdot y$ are the inner products of the predictors x_i times the *response vector* y.

Conversely, if the predictors are *collinear* then their squared predictor matrix is *singular* and the regression is also singular. Predictors are *collinear* if there's a linear combination that is constant.

```
> # Perform PCA of the predictors
> pcad <- prcomp(predm, center=FALSE, scale=FALSE)
> # Calculate the PCA predictors
> predpca <- predm %*% pcad$rotation
> # Principal components are orthogonal to each other
> round(t(predpca) %*% predpca, 2)
> # Calculate the PCA regression coefficients using lm()
> regmod <- lm(respv ~ predpca - 1)
> summary(regmod)
> regmod$coefficients
> # Calculate the PCA regression coefficients directly
> colSums(predpca*drop(respv))/colSums(predpca^2)
> # Create almost collinear predictors
> predcol <- predm
> predcol[, 1] <- (predcol[, 1]/1e3 + predcol[, 2])
> # Calculate the PCA predictors
> pcad <- prcomp(predcol, center=FALSE, scale=FALSE)
> predpca <- predcol %*% pcad$rotation
> round(t(predpca) %*% predpca, 6)
> # Calculate the PCA regression coefficients
> drop(MASS::ginv(predpca) %*% respv)
> # Calculate the PCA regression coefficients directly
> colSums(predpca*drop(respv))/colSums(predpca^2)
```

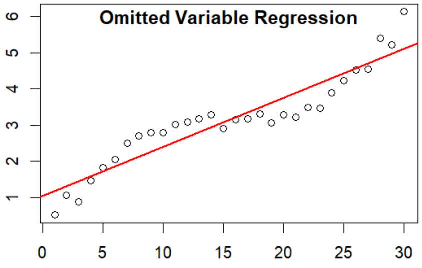
Omitted Variable Bias

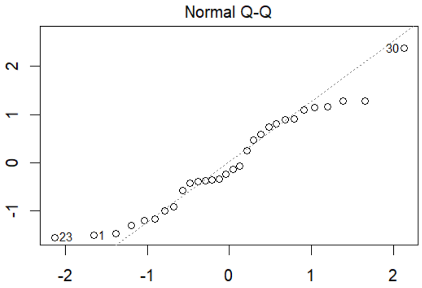
Omitted Variable Bias occurs in a regression model that omits important predictors.

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
> predm <- 1:30
> omitv <- sin(0.2*1:30)
> # Response depends on both predictors
> respv <- 0.2*predm + omitv + 0.2*rnorm(30)
> # Mis-specified regression only one predictor
> modovb <- lm(respv ~ predm)
> regsum <- summary(modovb)
> regsum$coeff
> regsum$r.squared
> # Durbin-Watson test shows residuals are autocorrelated
> lmtest::dwtest(modovb)
> # 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(respv ~ predm)
> abline(modovb, lwd=2, col="red")
> title(main="Omitted Variable Regression", line=-1)
> plot(modovb, 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
> regsum <- summary(regmod)
> # Degrees of freedom of residuals
> nrows <- NROW(predm)
> ncols <- NCOL(predm)
> degf <- (nrows - ncols)
> all.equal(degf, regsum$df[2])
[1] TRUE
> # Variance of residuals
> residsd <- sum(resids^2)/degf
```

Covariance Matrix of the Regression Coefficients

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

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

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

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

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

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 the predictors 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
> pred2 <- MASS::ginv(crossprod(predm))
> # pred2 <- t(predm) %*% predm
> # Variance of residuals
> residsd <- sum(resids^2)/degf
> # Calculate covariance matrix of betas
> betacovar <- residsd*pred2
> # round(betacovar, 3)
> betasd <- sqrt(diag(betacovar))
> all.equal(betasd, regsum$coeff[, 2], check.attributes=FALSE)
[1] TRUE
> # Calculate t-values of betas
> betatvals <- drop(betav)/betasd
> all.equal(betatvals, regsum$coeff[, 3], check.attributes=FALSE)
[1] TRUE
> # Calculate two-sided p-values of betas
> betapvals <- 2*pt(-abs(betatvals), df=degf)
> all.equal(betapvals, regsum$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(predm)), predinv %*% t(predinv))
[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:

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

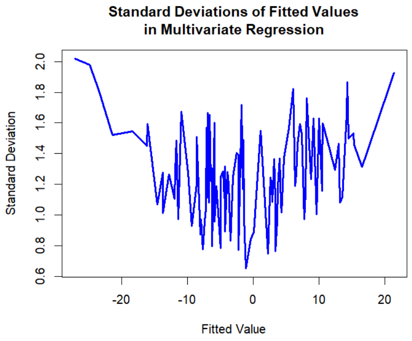
$$\frac{\mathbb{H}\,\mathbb{E}[\hat{\varepsilon}\hat{\varepsilon}^T]\,\mathbb{H}^T}{d_{free}} = \sigma^2_{\varepsilon}\,\mathbb{H} = \sigma^2_{\varepsilon}\,\mathbb{X}(\mathbb{X}^T\mathbb{X})^{-1}\mathbb{X}^T$$

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
> infmat <- predm %*% predinv
> # The influence matrix is idempotent
> all.equal(infmat, infmat %*% infmat)
```



```
> # Calculate covariance and standard deviations of fitted values
> fitcovar <- residsd*infmat
> fitsd <- sqrt(diag(fitcovar))
> # Sort the standard deviations
> fitsd <- cbind(fitted=fitv, stdev=fitsd)
> fitsd <- fitsd[order(fitv), ]
> # 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
> retp <- rutils::etfenv$returns[, c("XLF", "XLE")]
> retp <- na.omit(retp)
> nrows <- NROW(retp)
> head(retp)
> # Define regression formula
> formulav <- paste(colnames(retp)[1],
+   paste(colnames(retp)[-1], collapse="+"),
+   sep=" ~ ")
> # Standard regression
> regmod <- lm(formulav, data=retp)
> regsum <- summary(regmod)
> # Bootstrap of regression
> # Initialize the random number generator
> set.seed(1121, "Mersenne-Twister", sample.kind="Rejection")
> bootd <- sapply(1:100, function(x) {
+   samplev <- sample.int(nrows, replace=TRUE)
+   regmod <- lm(formulav, data=retp[samplev, ])
+   regmod$coefficients
+ })  # end sapply
> # Means and standard errors from regression
> regsum$coefficients
> # Means and standard errors from bootstrap
> dim(bootd)
> t(apply(bootd, MARGIN=1,
+ function(x) c(mean=mean(x), stderror=sd(x))))
```

Forecasts From *Multivariate Regression* Models

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

$$y_f = \mathbb{X}_{new} \, \beta$$

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

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

The variance σ_f^2 of the *forecast value* is:

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

The variance σ_f^2 of the *forecast value* is equal to the *predictor* vector multiplied by the *covariance matrix* of the *regression coefficients* σ_β^2.

```
> # New data predictor is a data frame or row vector
> set.seed(1121, "Mersenne-Twister", sample.kind="Rejection")
> newdata <- data.frame(matrix(c(1, rnorm(5)), nr=1))
> colnamev <- colnames(predm)
> colnames(newdata) <- colnamev
> newdata <- as.matrix(newdata)
> fcast <- drop(newdata %*% betav)
> predsd <- drop(sqrt(newdata %*% betacovar %*% t(newdata)))
```

Forecasts From *Multivariate Regression* Using lm()

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

predict.lm() is the forecasting 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("respv ~ ",
+   paste(colnames(predm), collapse=" + "), " - 1")
> # Specify multivariate regression using formula
> regmod <- lm(formulav, data=data.frame(cbind(respv, predm)))
> regsum <- summary(regmod)
> # Predict from lm object
> fcastlm <- predict.lm(object=model, newdata=newdata,
+    interval="confidence", confl=1-2*(1-pnorm(2)))
> # Calculate t-quantile
> tquant <- qt(pnorm(2), df=degf)
> fcasth <- (fcast + tquant*predsd)
> fcastl <- (fcast - tquant*predsd)
> # Compare with matrix calculations
> all.equal(fcastlm[1, "fit"], fcast)
> all.equal(fcastlm[1, "lwr"], fcastl)
> all.equal(fcastlm[1, "upr"], fcasth)
```

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 $(y_{fit} - \bar{y})$:

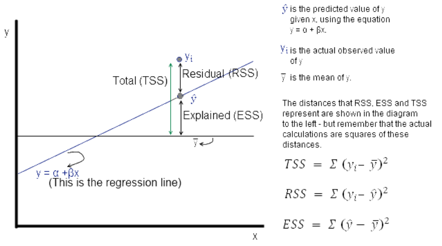
$$(y - y_{fit})^T (y_{fit} - \bar{y}) = 0$$

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*:
$d^{TSS}_{free} = (n - k) + (k - 1) = n - 1$.



$\hat{y}$ 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((respv-mean(respv))^2)
> ess <- sum((fitv-mean(fitv))^2)
> rss <- sum(resids^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*:

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

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

```
> # Set regression attribute for intercept
> attributes(regmod$terms)$intercept <- 1
> # Regression summary
> regsum <- summary(regmod)
> # Regression R-squared
> rsquared <- ess/tss
> all.equal(rsquared, regsum$r.squared)
[1] TRUE
> # Correlation between response and fitted values
> corfit <- drop(cor(respv, fitv))
> # Squared correlation between response and fitted values
> all.equal(corfit^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(predm)
> ncols <- NCOL(predm)
> # Degrees of freedom of residuals
> degf <- (nrows - ncols)
> # Adjusted R-squared
> rsqadj <- (1-sum(resids^2)/degf/var(respv))
> # Compare adjusted R-squared from lm()
> all.equal(drop(rsqadj), regsum$adj.r.squared)
[1] TRUE
```

Fisher's *F-distribution*

Let χ_m^2 and χ_n^2 be independent random variables following *chi-squared* distributions with m and n degrees of freedom.

Then the random variable:

$$F = \frac{\chi_m^2/m}{\chi_n^2/n}$$

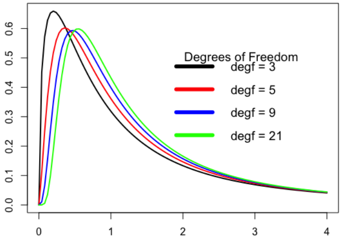
Follows the *F-distribution* with m and n degrees of freedom, with the probability density function:

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

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

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

```
> # Plot four curves in loop
> degf <- c(3, 5, 9, 21)  # Degrees of freedom
> colorv <- c("black", "red", "blue", "green")
> for (indeks in 1:NROW(degf)) {
+   curve(expr=df(x, df1=degf[indeks], df2=3),
+     xlim=c(0, 4), xlab="", ylab="", lwd=2,
+     col=colorv[indeks], add=as.logical(indeks-1))
+ }  # end for
```



```
> # Add title
> title(main="F-Distributions", line=0.5)
> # Add legend
> labelv <- paste("degf", degf, sep=" = ")
> legend("topright", title="Degrees of Freedom", inset=0.0, bty="n",
+         y.intersp=0.4, labelv, cex=1.2, lwd=6, lty=1, col=colorv)
```

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 *null hypothesis* of the *F-test* test is 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 p_1 parameters with $df_1 = n - p_1$ degrees of freedom, and the *unrestricted* model have p_2 parameters with $df_2 = n - p_2$ degrees of freedom, with $p_1 < p_2$.

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

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

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

If the *restricted* model has only one parameter (the constant intercept term), then $df_1 = n - 1$, and its *fitted values* are equal to the average of the *response*: $y_i^{fit} = \bar{y}$, so RSS_1 is equal to the TSS: $RSS_1 = TSS = (y - \bar{y})^2$, so its *Explained Sum of Squares* is equal to zero: $ESS_1 = TSS - RSS_1 = 0$.

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 ($p_2 = k$), and $RSS_2 = RSS$ and $ESS_2 = ESS$, 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).

The *null hypothesis* of the *F-test* test is that the *F-statistic* F is not significantly greater than 1 (the variance of *ESS* is not significantly greater than of *RSS*).

A large value of the *F-statistic* F indicates that the variance of *ESS* is significantly greater than that of *RSS*, and that the regression statistically significant.

```
> # F-statistic from lm()
> regsum$fstatistic
value numdf dendf
 3.37  5.00 94.00
> # Degrees of freedom of residuals
> degf <- (nrows - ncols)
> # F-statistic from ESS and RSS
> fstat <- (ess/(ncols-1))/(rss/degf)
> all.equal(fstat, regsum$fstatistic[1], check.attributes=FALSE)
[1] TRUE
> # p-value of F-statistic
> 1-pf(q=fstat, df1=(ncols-1), df2=(nrows-ncols))
[1] 0.00757
```

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.

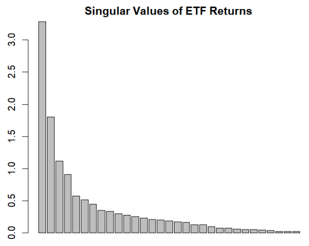
The *regularized inverse* is obtained by removing very small *singular values*:

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

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



```
> # Calculate generalized inverse from SVD
> invmat <- svdec$v %*% (t(svdec$u) / svdec$d)
> # Verify inverse property of inverse
> all.equal(zoo::coredata(retp), retp %*% invmat %*% retp)
> # Calculate regularized inverse from SVD
> dimax <- 1:3
> invreg <- svdec$v[, dimax] %*%
+   (t(svdec$u[, dimax]) / svdec$d[dimax])
> # Calculate regularized inverse using RcppArmadillo
> invcpp <- HighFreq::calc_inv(retp, dimax=3)
> all.equal(invreg, invcpp, check.attributes=FALSE)
> # Calculate regularized inverse from Moore-Penrose pseudo-inverse
> retsq <- t(retp) %*% retp
> eigend <- eigen(retsq)
> inv2 <- eigend$vectors[, dimax] %*%
+   (t(eigend$vectors[, dimax]) / eigend$values[dimax])
> invmp <- inv2 %*% t(retp)
> 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* $\mathbb{X}$:

$$\mathbb{H}_{trans} = \mathbb{X}_{trans}(\mathbb{X}_{trans}^T\mathbb{X}_{trans})^{-1}\mathbb{X}_{trans}^T =$$
$$\mathbb{X}\mathbb{W}(\mathbb{W}^T\mathbb{X}^T\mathbb{X}\mathbb{W})^{-1}\mathbb{W}^T\mathbb{X}^T =$$
$$\mathbb{X}\mathbb{W}\mathbb{W}^{-1}(\mathbb{X}^T\mathbb{X})^{-1}\mathbb{W}^{T^{-1}}\mathbb{W}^T\mathbb{X}^T =$$
$$\mathbb{X}(\mathbb{X}^T\mathbb{X})^{-1}\mathbb{X}^T = \mathbb{H}$$

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
> matv <- matrix(runif(ncols^2, min=(-1), max=1), ncol=ncols)
> # Calculate linear combinations of predictor columns
> predt <- predm %*% matv
> # Calculate the influence matrix of the transformed predictor
> influencet <- predt %*% MASS::ginv(predt)
> # Compare the influence matrices
> all.equal(infmat, influencet)
[1] TRUE
```

Principal Component Regression

In *Principal Component Regression* (*PCR*), the predictor matrix $\mathbb{X}$ is multiplied by the *PCA rotation matrix* $\mathbb{W}$:

$$\mathbb{X}_{pca} = \mathbb{X}\mathbb{W}$$

So that the principal component vectors form the columns of the new predictor matrix.

Since the new *PCR* predictors x_i^{pca} are orthogonal, the regression coefficients are simply:

$$\beta_i = \frac{x_i^{pca} \cdot y}{\sigma_i^2}$$

Where $x_i^{pca} \cdot y$ are the inner products of the *PCR* predictors x_i^{pca} times the *response vector* y, and $\sigma_i^2 = x_i^{pca} \cdot x_i^{pca}$ are the inner products (sum of squares) of the predictors x_i^{pca}.

```
> # Perform PCA of the predictors
> pcad <- prcomp(predm, center=FALSE, scale=FALSE)
> # Calculate the PCA predictors
> predpca <- predm %*% pcad$rotation
> # Principal components are orthogonal to each other
> round(t(predpca) %*% predpca, 2)
> # Calculate the PCA influence matrix
> infmat <- predm %*% MASS::ginv(predm)
> infpca <- predpca %*% MASS::ginv(predpca)
> all.equal(infmat, infpca)
> # Calculate the regression coefficients
> coeffv <- drop(MASS::ginv(predm) %*% respv)
> # Transform the collinear regression coefficients to the PCA
> drop(coeffv %*% pcad$rotation)
> # Calculate the PCA regression coefficients
> drop(MASS::ginv(predpca) %*% respv)
> # Calculate the PCA regression coefficients directly
> colSums(predpca*drop(respv))/colSums(predpca^2)
```

Dimension Reduction Using Principal Component Regression

If the predictor columns are *collinear* then some of the *PCR* predictor squares are zero $\sigma_i^2 = 0$, and the associated regression coefficients are infinite (indeterminate) and should be discarded.

The regression can also become *singular* if the number of rows of the predictor is too small, or is even less than the number of its columns.

The regression can be *regularized* by removing the infinite or very large *PCR* regression coefficients, and transforming the coefficients back to the original predictor coordinates.

This is called *dimension reduction* - excluding the principal components with very small squares.

Dimension reduction can also be applied to reduce model overfitting by reducing the number of effective predictors.

```
> # Create almost collinear predictors
> predcol <- predm
> predcol[, 1] <- (predcol[, 1]/1e3 + predcol[, 2])
> # Calculate the collinear regression coefficients
> coeffv <- drop(MASS::ginv(predcol) %*% respv)
> coeffv
> # Calculate the PCA predictors
> pcad <- prcomp(predcol, center=FALSE, scale=FALSE)
> predpca <- predcol %*% pcad$rotation
> round(t(predpca) %*% predpca, 6)
> # Transform the collinear regression coefficients to the PCA
> drop(coeffv %*% pcad$rotation)
> # Calculate the PCA regression coefficients
> coeffpca <- drop(MASS::ginv(predpca) %*% respv)
> # Calculate the PCA regression coefficients directly
> colSums(predpca*drop(respv))/colSums(predpca^2)
> # Transform the PCA regression coefficients to the original coordi
> drop(coeffpca %*% MASS::ginv(pcad$rotation))
> coeffv
> # Calculate the regression coefficients after dimension reduction
> npca <- NROW(coeffpca)
> drop(coeffpca[-npca] %*% MASS::ginv(pcad$rotation)[-npca, ])
> # Compare with the collinear regression coefficients
> coeffv
> # Calculate the original regression coefficients
> drop(MASS::ginv(predm) %*% respv)
```

Reading *TAQ* Data From .csv Files

Trade and Quote (*TAQ*) data stored in .csv files can be very large, so it's better to read it using the function data.table::fread() which is much faster than the function read.csv().

Each *trade* or *quote* contributes a *tick* (row) of data, and the number of ticks can be very large (hundred of thousands per day, or more).

The function strptime() coerces character strings representing the date and time into POSIXlt *date-time* objects.

The argument format="%H:%M:%OS" allows the parsing of fractional seconds, for example "15:59:59.989847074".

The function as.POSIXct() coerces objects into POSIXct *date-time* objects, with a numeric value representing the *moment of time* in seconds.

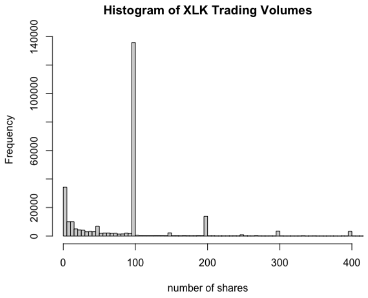
```
> library(HighFreq)
> # Read TAQ trade data from csv file
> taq <- data.table::fread(file="/Users/jerzy/Develop/data/xlk_tick_
> # Inspect the TAQ data in data.table format
> taq
> class(taq)
> colnames(taq)
> sapply(taq, class)
> symbol <- taq$SYM_ROOT[1]
> # Create date-time index
> datev <- paste(taq$DATE, taq$TIME_M)
> # Coerce date-time index to POSIXlt
> datev <- strptime(datev, "%Y%m%d %H:%M:%OS")
> class(datev)
> # Display more significant digits
> # options("digits")
> options(digits=20, digits.secs=10)
> last(datev)
> unclass(last(datev))
> as.numeric(last(datev))
> # Coerce date-time index to POSIXct
> datev <- as.POSIXct(datev)
> class(datev)
> last(datev)
> unclass(last(datev))
> as.numeric(last(datev))
> # Calculate the number of seconds
> nsecs <- as.numeric(last(datev)) - as.numeric(first(datev))
> # Calculate the number of ticks per second
> NROW(taq)/(6.5*3600)
> # Select TAQ data columns
> taq <- taq[, .(price=PRICE, volume=SIZE)]
```

Trading Volumes in High Frequency Data

The trading volumes represent the number of shares traded at a given price.

The histogram of the trading volumes shows that the highest frequencies of trades are for 100 shares and for round lots (trades that are multiples of 100 shares.)

There are also significant frequencies for *odd lots*, with small volumes of less than 100 shares.



```
> # Coerce trade ticks to xts series
> xlk <- xts::xts(taq[, .(price, volume)], datev)
> colnames(xlk) <- c("price", "volume")
> save(xlk, file="/Users/jerzy/Develop/data/xlk_tick_trades2020_0316
> # save(xlk, file="/Users/jerzy/Develop/data/xlk_tick_trades2020_03
> # Plot histogram of the trading volumes
> hist(xlk$volume, main="Histogram of XLK Trading Volumes",
+      breaks=1e5, xlim=c(1, 400), xlab="number of shares")
```

Microstructure Noise in High Frequency Data

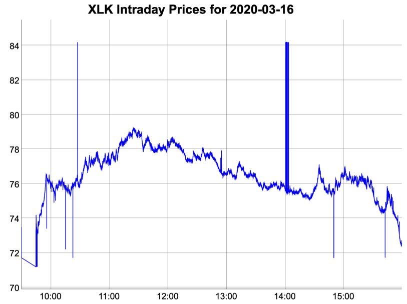
High frequency data contains *microstructure noise* in the form of *price spikes* and the *bid-ask bounce*.

Price spikes are single ticks with prices far away from the average.

Price spikes are often caused by data collection errors, but sometimes they represent actual trades with very large lot (trade) sizes.

The *bid-ask bounce* is the bouncing of traded prices between the bid and ask prices.

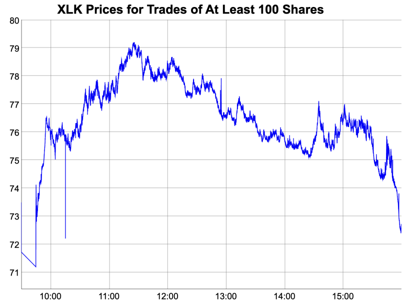
The *bid-ask bounce* creates an illusion of rapidly changing prices, while in reality the mid price is unchanged.



```
> # Plot dygraph
> dygraphs::dygraph(xlk$price, main="XLK Intraday Prices for 2020-03
+   dyOptions(colors="blue", strokeWidth=1)
> # Plot in x11 window
> x11(width=6, height=5)
> quantmod::chart_Series(x=xlk$price, name="XLK Intraday Prices for
```

Microstructure Noise And Trading Volumes in High Frequency Data

The number of the *price spikes* depends on the level of trading volumes, with the number decreasing with higher trading volumes.



```
> # Plot dygraph of trade prices of at least 100 shares
> dygraphs::dygraph(xlk$price[volumv >= 100, ],
+   main="XLK Prices for Trades of At Least 100 Shares") %>%
+   dyOptions(colors="blue", strokeWidth=1)
```

Filtering Microstructure Noise From High Frequency Data

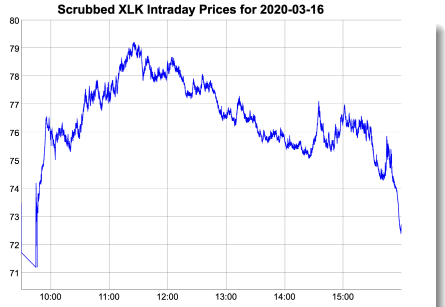
Microstructure noise in high frequency data can be identified using a *Hampel filter*.

The *z-scores* are equal to the prices minus the median of the prices, divided by the median absolute deviation (*MAD*) of prices:

$$z_i = \frac{p_i - \operatorname{median}(\mathbf{p})}{\text{MAD}}$$

If the absolute value of the *z-score* exceeds the *threshold value* then it's classified as *bad data*, and it can be removed or replaced.

```
> # Calculate centered Hampel filter to remove bad prices
> look_back <- 71
> half_back <- look_back %/% 2
> pricev <- xlk$price
> medianv <- roll::roll_median(pricev, width=look_back)
> colnames(medianv) <- c("median")
> # Overwrite leading NA values
> medianv[1:look_back, ] <- pricev[1:look_back, ]
> # medianv <- TTR::runMedian(pricev, n=look_back)
> medianv <- rutils::lagit(medianv, lagg=(-half_back), pad_zeros=F
> madv <- HighFreq::roll_var(pricev, look_back=look_back, method="
> # madv <- TTR::runMAD(pricev, n=look_back)
> madv <- rutils::lagit(madv, lagg=(-half_back), pad_zeros=FALSE)
> # Calculate the Z-scores
> zscores <- ifelse(madv > 0, (pricev - medianv)/madv, 0)
> # Z-scores have very fat tails
> range(zscores); mad(zscores)
> madz <- mad(zscores[abs(zscores) > 0])
> hist(zscores, breaks=50000, xlim=c(-2*madz, 2*madz))
```



```
> # Define discrimination threshold value
> threshv <- 6*madz
> # Identify good prices with small z-scores
> isgood <- (abs(zscores) < threshv)
> # Calculate the number of bad prices
> sum(!isgood)
> # Remove bad prices and calculate time series of scrubbed prices
> priceg <- xlk$price[isgood]
> # Plot dygraph of the scrubbed prices
> dygraphs::dygraph(priceg, main="Scrubbed XLK Intraday Prices") %>%
+   dyOptions(colors="blue", strokeWidth=1)
> # Plot using chart_Series()
> x11(width=6, height=5)
> quantmod::chart_Series(x=priceg,
+   name="Clean XLK Intraday Prices for 2020-03-16")
```

Classifying Data Outliers Using the Hampel Filter

The Hampel filter is a *classifier* which classifies the prices as either good or bad data points.

In order to measure the performance of the Hampel filter, we add price spikes to the clean prices, to see how accurately they're classified.

Let the *null hypothesis* be that the given price is a good data point.

A positive result corresponds to rejecting the *null hypothesis*, while a negative result corresponds to accepting the *null hypothesis*.

The classifications 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 good data is classified as bad.

A *type II* error is the incorrect acceptance of a FALSE *null hypothesis* (i.e. a "false negative"), when bad data is classified as good.

```
> # Add 200 random price spikes to the clean prices
> set.seed(1121, "Mersenne-Twister", sample.kind="Rejection")
> nspikes <- 200
> nrows <- NROW(priceg)
> ispike <- logical(nrows)
> ispike[sample(x=nrows, size=nspikes)] <- TRUE
> priceb <- priceg
> priceb[ispike] <- priceb[ispike]*
+   sample(c(0.999, 1.001), size=nspikes, replace=TRUE)
> # Calculate the z-scores
> medianv <- roll::roll_median(priceb, width=look_back)
> # Plot the bad prices and their medians
> pricem <- cbind(priceb, medianv)
> colnames(pricem) <- c("prices with spikes", "median")
> dygraphs::dygraph(pricem, main="XLK Prices With Spikes") %>%
+   dyOptions(colors=c("red", "blue"))
> # medianv <- TTR::runMedian(priceb, n=look_back)
> madv <- HighFreq::roll_var(priceb, look_back=look_back, method="no
> # madv <- TTR::runMAD(priceb, n=look_back)
> zscores <- ifelse(madv > 0, (priceb - medianv)/madv, 0)
> zscores[1:look_back, ] <- 0
> # Z-scores have very fat tails
> range(zscores); mad(zscores)
> madz <- mad(zscores[abs(zscores) > 0])
> hist(zscores, breaks=10000, xlim=c(-4*madz, 4*madz))
> # Identify good prices with small z-scores
> threshv <- 5*madz
> isgood <- (abs(zscores) < threshv)
> sum(!isgood)
```

Confusion Matrix of a Binary Classification Model

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

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.

Actual \ **Forecast**	**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 confusion matrix
> table(actual=!ispike, forecast=isgood)
> sum(!isgood)
> # FALSE positive (type I error)
> sum(!ispike & isgood)
> # FALSE negative (type II error)
> sum(ispike & isgood)
```

Let the *null hypothesis* be that the given price is a good data point.

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).

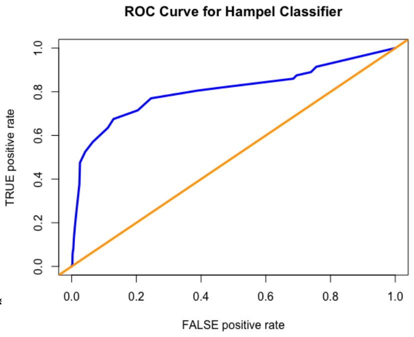
The sum of the *true negative* plus the *false positive* rate is equal to 1.

Receiver Operating Characteristic (ROC) Curve

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.

```
> # Confusion matrix as function of threshold
> confun <- function(actualv, zscores, threshv) {
+     confmat <- table(actualv, (abs(zscores) < threshv))
+     confmat <- confmat / rowSums(confmat)
+     c(typeI=confmat[2, 1], typeII=confmat[1, 2])
+   }  # end confun
> confun(!ispike, zscores, threshv=threshv)
> # Define vector of discrimination thresholds
> threshv <- madz*seq(from=0.1, to=7, by=0.1)^2
> # Calculate error rates
> errorr <- sapply(threshv, confun, actualv=!ispike, zscores=zscores
> errorr <- t(errorr)
> rownames(errorr) <- threshv
> errorr <- rbind(c(1, 0), errorr)
> errorr <- rbind(errorr, c(0, 1))
> # Calculate area under ROC curve (AUC)
> truepos <- (1 - errorr[, "typeII"])
> truepos <- (truepos + rutils::lagit(truepos))/2
> falsepos <- rutils::diffit(errorr[, "typeI"])
> abs(sum(truepos*falsepos))
```



```
> # Plot ROC curve for Hampel classifier
> plot(x=errorr[, "typeI"], y=1-errorr[, "typeII"],
+      xlab="FALSE positive rate", ylab="TRUE positive rate",
+      xlim=c(0, 1), ylim=c(0, 1),
+      main="ROC Curve for Hampel Classifier",
+      type="l", lwd=3, col="blue")
> abline(a=0.0, b=1.0, lwd=3, col="orange")
```

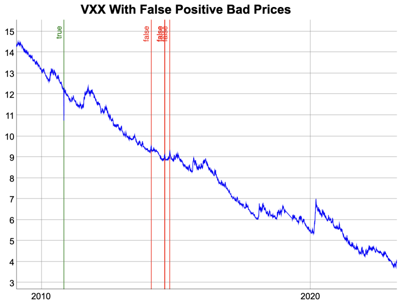
Filtering Bad Data From Daily Stock Prices

Daily stock prices also contain bad data points consisting of mostly single, isolated spikes in prices.

The number of false positives may be too high, so the Hampel filter parameters (the look-back interval and the threshold) need adjustment.

For example, the *VXX* has only one bad price (on 2010-11-08), but the Hampel filter identifies many more than that (which are false positives).

```
> # Calculate the centered Hampel filter for VXX
> pricev <- log(na.omit(rutils::etfenv$prices$VXX))
> medianv <- roll::roll_median(pricev, width=look_back)
> medianv[1:look_back, ] <- pricev[1:look_back, ]
> medianv <- rutils::lagit(medianv, lagg=(-half_back), pad_zeros=FAL
> madv <- HighFreq::roll_var(pricev, look_back=look_back, method="nonparametric")
> madv <- rutils::lagit(madv, lagg=(-half_back), pad_zeros=FALSE)
> zscores <- ifelse(madv > 0, (pricev - medianv)/madv, 0)
> range(zscores); mad(zscores)
> madz <- mad(zscores[abs(zscores) > 0])
> hist(zscores, breaks=100, xlim=c(-3*madz, 3*madz))
```


VXX With False Positive Bad Prices

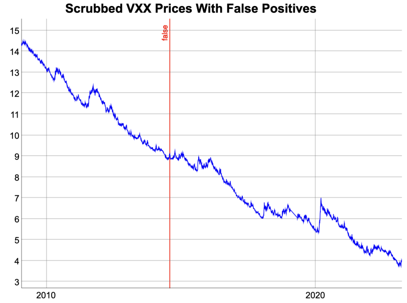
```
> # Define discrimination threshold value
> threshv <- 10*madz
> # Calculate the bad prices
> isgood <- (abs(zscores) < threshv)
> sum(!isgood)
> # Dates of the bad prices
> zoo::index(pricev[!isgood])
> # Calculate the false positives
> isfp <- !isgood
> isfp[which(zoo::index(pricev) == as.Date("2010-11-08"))] <- FALSE
> # Plot dygraph of the prices with bad prices
> dygraphs::dygraph(pricev, main="VXX Prices With False Positives")
+   dyEvent(zoo::index(pricev[isfp]), label=rep("false", sum(isfp)),
+   dyEvent(zoo::index(pricev["2010-11-08"]), label="true", strokePa
+   dyOptions(colors="blue", strokeWidth=1)
```

Scrubbing Bad Stock Prices

Bad stock prices can be scrubbed (replaced) with the previous good price.

But it's incorrect to replace bad prices with the average of the previous good price and the next good price, since that would cause data snooping.

```
> # Dates of the bad prices
> dates <- zoo::index(pricev)
> dateb <- dates[!isgood]
> # Dates of the previous prices
> datep <- c(!isgood[-1], FALSE)
> dates[datep]
> # Replace bad stock prices with the previous good prices
> priceg <- pricev
> priceg[!isgood] <- pricev[datep]
> # Calculate the Z-scores
> medianv <- roll::roll_median(priceg, width=look_back)
> medianv[1:look_back, ] <- priceg[1:look_back, ]
> medianv <- rutils::lagit(medianv, lagg=(-half_back), pad_zeros=F
> madv <- HighFreq::roll_var(priceg, look_back=look_back, method="
> madv <- rutils::lagit(madv, lagg=(-half_back), pad_zeros=FALSE)
> zscores <- ifelse(madv > 0, (priceg - medianv)/madv, 0)
> madz <- mad(zscores[abs(zscores) > 0])
> # Calculate the number of bad prices
> threshv <- 10*madz
> isgood <- (abs(zscores) < threshv)
> sum(!isgood)
> zoo::index(priceg[!isgood])
```



```
> # Calculate the false positives
> isfp <- !isgood
> isfp[which(zoo::index(pricev) == as.Date("2010-11-08"))] <- FALSE
> # Plot dygraph of the prices with bad prices
> dygraphs::dygraph(priceg, main="Scrubbed VXX Prices With False Pos
+   dyEvent(zoo::index(priceg[isfp]), label=rep("false", sum(isfp))
+   dyOptions(colors="blue", strokeWidth=1)
```

The *Logistic* Function

The *logistic* function expresses the probability of a numerical variable ranging over the whole interval of real numbers:

$$p(x) = \frac{1}{1 + \exp(-\lambda x)}$$

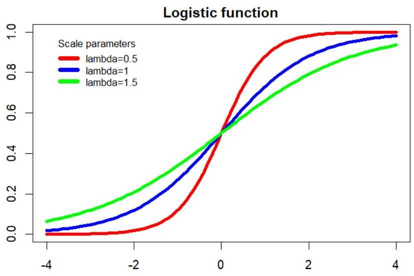
Where λ is the scale (dispersion) parameter.

The *logistic* function is often used as an activation function in neural networks, and logistic regression can be viewed as a perceptron (single neuron network).

The *logistic* function can be inverted to obtain the *Odds Ratio* (the ratio of probabilities for favorable to unfavorable outcomes):

$$\frac{p(x)}{1 - p(x)} = \exp(\lambda x)$$

The function `plogis()` gives the cumulative probability of the *Logistic* distribution,



```
> lambdav <- c(0.5, 1, 1.5)
> colorv <- c("red", "blue", "green")
> # Plot three curves in loop
> for (it in 1:3) {
+   curve(expr=plogis(x, scale=lambdav[it]),
+ xlim=c(-4, 4), type="l", xlab="", ylab="", lwd=4,
+ col=colorv[it], add=(it>1))
+ }  # end for
> # Add title
> title(main="Logistic function", line=0.5)
> # Add legend
> legend("topleft", title="Scale parameters",
+        paste("lambda", lambdav, sep="="), y.intersp=0.4,
+        inset=0.05, cex=0.8, lwd=6, bty="n", lty=1, col=colorv)
```

Performing *Logistic* Regression Using the Function `glm()`

Logistic regression (*logit*) is used when the response are discrete variables (like `factors` or `integers`), when *linear* regression can't be applied.

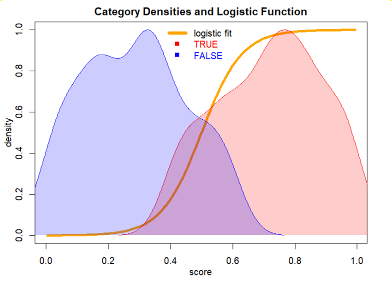
The function `glm()` fits generalized linear models, including *logistic* regressions.

The parameter `family=binomial(logit)` specifies a binomial distribution of residuals in the *logistic* regression model.

The *Mann-Whitney* test *null hypothesis* is that the two samples, x_i and y_i, were obtained from probability distributions with the same median (location).

The function `wilcox.test()` with parameter `paired=FALSE` (the default) calculates the *Mann-Whitney* test statistic and its p-value.

```
> # Initialize the random number generator
> set.seed(1121, "Mersenne-Twister", sample.kind="Rejection")
> # Simulate overlapping scores data
> sample1 <- runif(100, max=0.6)
> sample2 <- runif(100, min=0.4)
> # Perform Mann-Whitney test for data location
> wilcox.test(sample1, sample2)
> # Combine scores and add categorical variable
> predm <- c(sample1, sample2)
> respv <- c(logical(100), !logical(100))
> # Perform logit regression
> logmod <- glm(respv ~ predm, family=binomial(logit))
> class(logmod)
> summary(logmod)
```



```
> ordern <- order(predm)
> plot(x=predm[ordern], y=logmod$fitted.values[ordern],
+      main="Category Densities and Logistic Function",
+      type="l", lwd=4, col="orange", xlab="predictor", ylab="densit
> densv <- density(predm[respv])
> densv$y <- densv$y/max(densv$y)
> lines(densv, col="red")
> polygon(c(min(densv$x), densv$x, max(densv$x)), c(min(densv$y), de
> densv <- density(predm[!respv])
> densv$y <- densv$y/max(densv$y)
> lines(densv, col="blue")
> polygon(c(min(densv$x), densv$x, max(densv$x)), c(min(densv$y), de
> # Add legend
> legend(x="top", cex=1.0, bty="n", lty=c(1, NA, NA),
+  lwd=c(6, NA, NA), pch=c(NA, 15, 15), y.intersp=0.4,
+  legend=c("logistic fit", "TRUE", "FALSE"),
+  col=c("orange", "red", "blue"),
+  text.col=c("black", "red", "blue"))
```

The Likelihood Function of the Binomial Distribution

Let b be a binomial random variable, which either has the value $b = 1$ with probability p, or $b = 0$ with probability $(1 - p)$.

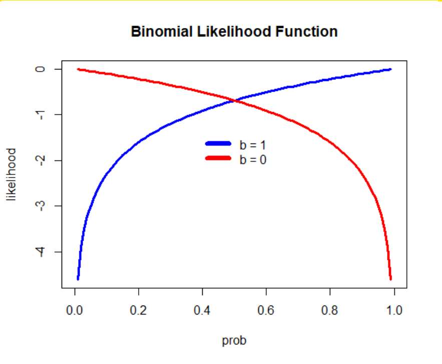
Then b follows the binomial distribution:

$$f(b) = b\,p + (1 - b)\,(1 - p)$$

The *log-likelihood function* $\mathcal{L}(p|b)$ of the probability p given the value b is obtained from the logarithms of the binomial probabilities:

$$\mathcal{L}(p|b) = b\,\log(p) + (1 - b)\,\log(1 - p)$$

The *log-likelihood function* measures how *likely* are the distribution parameters, given the observed values.



```
> # Likelihood function of binomial distribution
> likefun <- function(prob, b) {
+   b*log(prob) + (1-b)*log(1-prob)
+ }  # end likefun
> likefun(prob=0.25, b=1)
> # Plot binomial likelihood function
> curve(expr=likefun(x, b=1), xlim=c(0, 1), lwd=3,
+       xlab="prob", ylab="likelihood", col="blue",
+       main="Binomial Likelihood Function")
> curve(expr=likefun(x, b=0), lwd=3, col="red", add=TRUE)
> legend(x="top", legend=c("b = 1", "b = 0"),
+        title=NULL, inset=0.3, cex=1.0, lwd=6, y.intersp=0.4,
+        bty="n", lty=1, col=c("blue", "red"))
```

The Likelihood Function of the Logistic Model

Let b_i be binomial random variables, with probabilities p_i that depend on the numerical variables s_i through the logistic function:

$$p_i = \frac{1}{1 + \exp(-\lambda_0 - \lambda_1 s_i)}$$

Let's assume that the b_i and s_i values are known (observed), and we want to find the parameters λ_0 and λ_1 that best fit the observations.

The *log-likelihood function* $\mathcal{L}$ is equal to the sum of the individual *log-likelihoods*:

$$\mathcal{L}(\lambda_0, \lambda_1 | b_i) = \sum_{i=1}^{n} b_i \log(p_i) + (1 - b_i) \log(1 - p_i)$$

The *log-likelihood function* measures how *likely* are the distribution parameters, given the observed values.

```
> # Specify predictor matrix
> predm <- cbind(intercept=rep(1, NROW(respv)), predm)
> # Likelihood function of the logistic model
> likefun <- function(coeff, respv, predm) {
+   probs <- plogis(drop(predm %*% coeff))
+   -sum(respv*log(probs) + (1-respv)*log((1-probs)))
+ }  # end likefun
> # Run likelihood function
> coeff <- c(1, 1)
> likefun(coeff, respv, predm)
```

Multi-dimensional Optimization Using optim()

The function optim() performs *multi-dimensional* optimization.

The argument fn is the objective function to be minimized.

The argument of fn that is to be optimized, must be a vector argument.

The argument par is the initial vector argument value.

optim() accepts additional parameters bound to the dots "..." argument, and passes them to the fn objective function.

The arguments lower and upper specify the search range for the variables of the objective function fn.

method="L-BFGS-B" specifies the quasi-Newton *gradient* optimization method.

optim() returns a list containing the location of the minimum and the objective function value.

The *gradient* methods used by optim() can only find the local minimum, not the global minimum.

```
> # Rastrigin function with vector argument for optimization
> rastrigin <- function(vecv, param=25) {
+   sum(vecv^2 - param*cos(vecv))
+ }  # end rastrigin
> vecv <- c(pi/6, pi/6)
> rastrigin(vecv=vecv)
> # Draw 3d surface plot of Rastrigin function
> options(rgl.useNULL=TRUE); library(rgl)
> rgl::persp3d(
+   x=Vectorize(function(x, y) rastrigin(vecv=c(x, y))),
+   xlim=c(-10, 10), ylim=c(-10, 10),
+   col="green", axes=FALSE, zlab="", main="rastrigin")
> # Render the 3d surface plot of function
> rgl::rglwidget(elementId="plot3drgl", width=400, height=400)
> # Optimize with respect to vector argument
> optiml <- optim(par=vecv, fn=rastrigin,
+         method="L-BFGS-B",
+         upper=c(4*pi, 4*pi),
+         lower=c(pi/2, pi/2),
+         param=1)
> # Optimal parameters and value
> optiml$par
> optiml$value
> rastrigin(optiml$par, param=1)
```

Maximum Likelihood Calibration of the Logistic Model

The logistic model depends on the unknown parameters λ_0 and λ_1, which can be calibrated by maximizing the likelihood function.

The function `optim()` with the argument `hessian=TRUE` returns the Hessian matrix.

The Hessian is a matrix of the second-order partial derivatives of the likelihood function with respect to the optimization parameters:

$$H = \frac{\partial^2 \mathcal{L}}{\partial \lambda^2}$$

The Hessian matrix measures the convexity of the likelihood surface - it's large if the likelihood surface is highly convex, and it's small if the likelihood surface is flat.

If the likelihood surface is highly convex, then the coefficients can be determined with greater precision, so their standard errors are small. If the likelihood surface is flat, then the coefficients have large standard errors.

The inverse of the Hessian matrix provides the standard errors of the logistic parameters: $\sigma_{SE} = \sqrt{H^{-1}}$.

```
> # Initial parameters
> initp <- c(1, 1)
> # Find max likelihood parameters using steepest descent optimizer
> optiml <- optim(par=initp,
+         fn=likefun, # Log-likelihood function
+         method="L-BFGS-B", # Quasi-Newton method
+         respv=respv,
+         predm=predm,
+         upper=c(20, 20), # Upper constraint
+         lower=c(-20, -20), # Lower constraint
+         hessian=TRUE)
> # Optimal logistic parameters
> optiml$par
> unname(logmod$coefficients)
> # Standard errors of parameters
> sqrt(diag(solve(optiml$hessian)))
> regsum <- summary(logmod)
> regsum$coefficients[, 2]
```

Package *ISLR* With Datasets for Machine Learning

The package *ISLR* contains datasets used in the book *Introduction to Statistical Learning* by Gareth James, Daniela Witten, Trevor Hastie, and Robert Tibshirani.

The book introduces machine learning techniques using R, and it's a must for advanced finance applications.

```
> library(ISLR)  # Load package ISLR
> # get documentation for package tseries
> packageDescription("ISLR")  # get short description
>
> help(package="ISLR")  # Load help page
>
> library(ISLR)  # Load package ISLR
>
> data(package="ISLR")  # list all datasets in ISLR
>
> ls("package:ISLR")  # list all objects in ISLR
>
> detach("package:ISLR")  # Remove ISLR from search path
```

The Default Dataset

The data frame Default in the package *ISLR* contains credit default data.

The Default data frame contains two columns of categorical data (factors): default and student, and two columns of numerical data: balance and income.

The columns default and student contain factor data, and they can be converted to Boolean values, with TRUE if default == "Yes" and student == "Yes", and FALSE otherwise.

This avoids implicit coercion by the function glm().

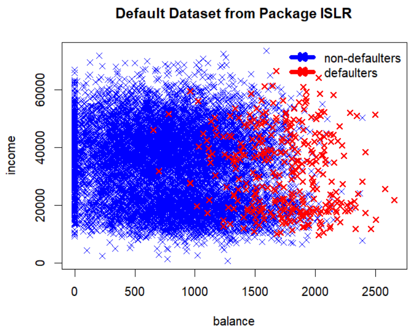
```
> # Coerce the default and student columns to Boolean
> Default <- ISLR::Default
> Default$default <- (Default$default == "Yes")
> Default$student <- (Default$student == "Yes")
> colnames(Default)[1:2] <- c("default", "student")
> attach(Default)  # Attach Default to search path
> # Explore credit default data
> summary(Default)
  default          student          balance         income
 Mode :logical   Mode :logical   Min.   :   0   Min.   :  772
 FALSE:9667      FALSE:7056      1st Qu.: 482   1st Qu.:21340
 TRUE :333       TRUE :2944      Median : 824   Median :34553
                                 Mean   : 835   Mean   :33517
                                 3rd Qu.:1166   3rd Qu.:43808
                                 Max.   :2654   Max.   :73554
> sapply(Default, class)
  default   student   balance    income
"logical" "logical" "numeric" "numeric"
> dim(Default)
[1] 10000     4
> head(Default)
  default student balance income
1   FALSE   FALSE     730  44362
2   FALSE    TRUE     817  12106
3   FALSE   FALSE    1074  31767
4   FALSE   FALSE     529  35704
5   FALSE   FALSE     786  38463
6   FALSE    TRUE     920   7492
```

The Dependence of default on The balance and income

The columns student, balance, and income can be used as *predictors* to predict the default column.

The scatterplot of income versus balance shows that the balance column is able to separate the data points of default = TRUE from default = FALSE.

But there is very little difference in income between the default = TRUE versus default = FALSE data points.



```
> # Plot data points for non-defaulters
> xlim <- range(balance); ylim <- range(income)
> plot(income ~ balance,
+      main="Default Dataset from Package ISLR",
+      xlim=xlim, ylim=ylim, pch=4, col="blue",
+      data=Default[!default, ])
> # Plot data points for defaulters
> points(income ~ balance, pch=4, lwd=2, col="red",
+  data=Default[default, ])
> # Add legend
> legend(x="topright", legend=c("non-defaulters", "defaulters"),
+  y.intersp=0.4, bty="n", col=c("blue", "red"), lty=1, lwd=6, pch=
```

Boxplots of the Default Dataset

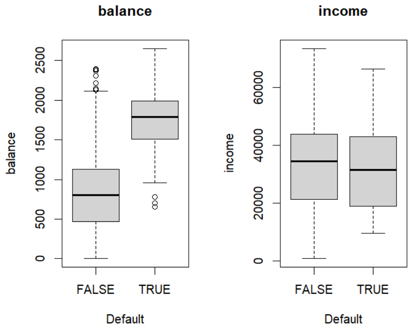
A *Box Plot* (box-and-whisker plot) is a graphical display of a distribution of data:

The *box* represents the upper and lower quartiles,
The vertical lines (whiskers) represent values beyond the quartiles,
Open circles represent values beyond the nominal range (outliers).

The function `boxplot()` plots a box-and-whisker plot for a distribution of data.

`boxplot()` has two `methods`: one for `formula` objects (involving categorical variables), and another for `data frames`.

The *Mann-Whitney* test shows that the `balance` column provides a strong separation between defaulters and non-defaulters, but the `income` column doesn't.



```
> # Perform Mann-Whitney test for the location of the balances
> wilcox.test(balance[default], balance[!default])
> # Perform Mann-Whitney test for the location of the incomes
> wilcox.test(income[default], income[!default])
```

```
> x11(width=6, height=5)
> # Set 2 plot panels
> par(mfrow=c(1,2))
> # Balance boxplot
> boxplot(formula=balance ~ default,
+   col="lightgrey", main="balance", xlab="Default")
> # Income boxplot
> boxplot(formula=income ~ default,
+   col="lightgrey", main="income", xlab="Default")
```

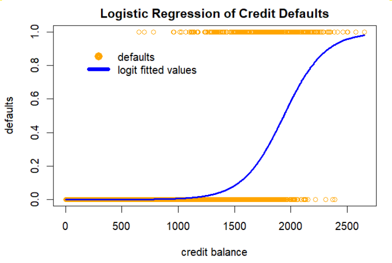
Modeling Credit Defaults Using *Logistic* Regression

The balance column can be used to calculate the probability of default using *logistic* regression.

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.

```
> # Fit logistic regression model
> logmod <- glm(default ~ balance, family=binomial(logit))
> class(logmod)
[1] "glm" "lm"
> summary(logmod)

Call:
glm(formula = default ~ balance, family = binomial(logit))

Coefficients:
              Estimate Std. Error z value Pr(>|z|)
(Intercept) -10.65133    0.36116   -29.5   <2e-16 ***
balance       0.00550    0.00022    24.9   <2e-16 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

(Dispersion parameter for binomial family taken to be 1)

    Null deviance: 2920.6  on 9999  degrees of freedom
Residual deviance: 1596.5  on 9998  degrees of freedom
AIC: 1600

Number of Fisher Scoring iterations: 8
```



```
> x11(width=6, height=5)
> par(mar=c(4, 4, 2, 2), oma=c(0, 0, 0, 0), mgp=c(2.5, 1, 0))
> plot(x=balance, y=default,
+      main="Logistic Regression of Credit Defaults",
+      col="orange", xlab="credit balance", ylab="defaults")
> ordern <- order(balance)
> lines(x=balance[ordern], y=logmod$fitted.values[ordern], col="blue
> legend(x="topleft", inset=0.1, bty="n", lwd=6, y.intersp=0.4,
+  legend=c("defaults", "logit fitted values"),
+  col=c("orange", "blue"), lty=c(NA, 1), pch=c(1, NA))
```

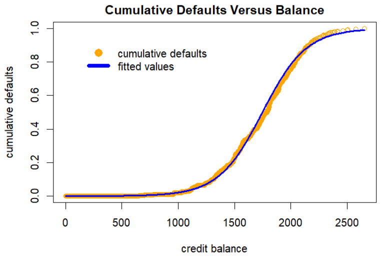
Modeling Cumulative Defaults Using *Logistic* Regression

The function `glm()` can model a *logistic* regression using either a `Boolean` response variable, or using a response variable specified as a frequency.

In the second case, the response variable should be defined as a two-column matrix, with the cumulative frequency of success (`TRUE`) and a cumulative frequency of failure (`FALSE`).

These two different ways of specifying the *logistic* regression are related, but they are not equivalent, because they have different error terms.

```
> # Calculate cumulative defaults
> sumd <- sum(default)
> defaultv <- sapply(balance, function(balv) {
+     sum(default[balance <= balv])
+ })  # end sapply
> # Perform logit regression
> logmod <- glm(cbind(defaultv, sumd-defaultv) ~ balance,
+   family=binomial(logit))
> summary(logmod)
```



```
> plot(x=balance, y=defaultv/sumd, col="orange", lwd=1,
+      main="Cumulative Defaults Versus Balance",
+      xlab="credit balance", ylab="cumulative defaults")
> ordern <- order(balance)
> lines(x=balance[ordern], y=logmod$fitted.values[ordern],
+ col="blue", lwd=3)
> legend(x="topleft", inset=0.1, bty="n", y.intersp=0.4,
+  legend=c("cumulative defaults", "fitted values"),
+  col=c("orange", "blue"), lty=c(NA, 1), pch=c(1, NA), lwd=6)
```

Multifactor *Logistic* Regression

Logistic regression calculates the probability of categorical variables, from the *Odds Ratio* of continuous *predictors*:

$$p = \frac{1}{1 + \exp(-\lambda_0 - \sum_{i=1}^{n} \lambda_i x_i)}$$

The *generic* function `summary()` produces a list of regression model summary and diagnostic statistics:

- coefficients: matrix with estimated coefficients, their z-values, and p-values,
- *Null* deviance: measures the differences between the response values and the probabilities calculated using only the intercept,
- *Residual* deviance: measures the differences between the response values and the model probabilities,

The `balance` and `student` columns are statistically significant, but the `income` column is not.

```
> # Fit multifactor logistic regression model
> colnamev <- colnames(Default)
> formulav <- as.formula(paste(colnamev[1],
+   paste(colnamev[-1], collapse="+"), sep=" ~ "))
> formulav
default ~ student + balance + income
> logmod <- glm(formulav, data=Default, family=binomial(logit))
> summary(logmod)

Call:
glm(formula = formulav, family = binomial(logit), data = Default)

Coefficients:
             Estimate Std. Error z value Pr(>|z|)
(Intercept) -1.09e+01   4.92e-01  -22.08   <2e-16 ***
studentTRUE -6.47e-01   2.36e-01   -2.74   0.0062 **
balance      5.74e-03   2.32e-04   24.74   <2e-16 ***
income       3.03e-06   8.20e-06    0.37   0.7115
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

(Dispersion parameter for binomial family taken to be 1)

    Null deviance: 2920.6  on 9999  degrees of freedom
Residual deviance: 1571.5  on 9996  degrees of freedom
AIC: 1580

Number of Fisher Scoring iterations: 8
```

Confounding Variables in Multifactor *Logistic* Regression

The `student` column alone can be used to calculate the probability of default using single-factor *logistic* regression.

But the coefficient from the single-factor regression is positive (indicating that students are more likely to default), while the coefficient from the multifactor regression is negative (indicating that students are less likely to default).

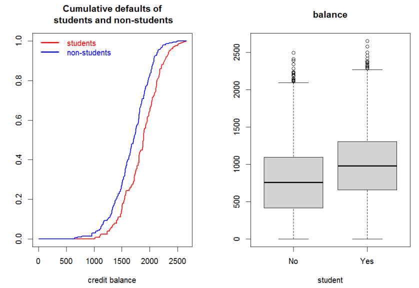
The reason that students are more likely to default is because they have higher credit balances than non-students - which is what the single-factor regression shows.

But students are less likely to default than non-students that have the same credit balance - which is what the multifactor model shows.

The `student` column is a confounding variable since it's correlated with the `balance` column.

That's why the multifactor regression coefficient for `student` is negative, while the single factor coefficient for `student` is positive.

```
> # Fit single-factor logistic model with student as predictor
> glm_student <- glm(default ~ student, family=binomial(logit))
> summary(glm_student)
> # Multifactor coefficient is negative
> logmod$coefficients
> # Single-factor coefficient is positive
> glm_student$coefficients
```



```
> # Calculate cumulative defaults
> cum_defaults <- sapply(balance, function(balv) {
+ c(student=sum(default[student & (balance <= balv)]),
+   non_student=sum(default[!student & (balance <= balv)]))
+ })  # end sapply
> total_defaults <- c(student=sum(student & default),
+       student=sum(!student & default))
> cum_defaults <- t(cum_defaults / total_defaults)
> # Plot cumulative defaults
> par(mfrow=c(1,2))  # Set plot panels
> ordern <- order(balance)
> plot(x=balance[ordern], y=cum_defaults[ordern, 1],
+      col="red", t="l", lwd=2, xlab="credit balance", ylab="",
+      main="Cumulative defaults of\n students and non-students")
> lines(x=balance[ordern], y=cum_defaults[ordern, 2], col="blue", lw
> legend(x="topleft", bty="n", y.intersp=0.4,
+  legend=c("students", "non-students"),
+  col=c("red", "blue"), text.col=c("red", "blue"), lwd=3)
> # Balance boxplot for student factor
```

Forecasting Credit Defaults using Logistic Regression

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 subject will not default: `default = FALSE`.

If the *forecast probability* is *less* than the *discrimination threshold*, then the forecast is that the subject will not default and that the *null hypothesis* is `TRUE`.

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

```
> # Perform in-sample forecast from logistic regression model
> fcast <- predict(logmod, type="response")
> all.equal(logmod$fitted.values, fcast)
[1] TRUE
> # Define discrimination threshold value
> threshv <- 0.7
> # Calculate confusion matrix in-sample
> table(actual=!default, forecast=(fcast < threshv))
       forecast
actual  FALSE TRUE
  FALSE    57  276
  TRUE     12 9655
> # Fit logistic regression over training data
> # Initialize the random number generator
> set.seed(1121, "Mersenne-Twister", sample.kind="Rejection")
> nrows <- NROW(Default)
> samplev <- sample.int(n=nrows, size=nrows/2)
> trainset <- Default[samplev, ]
> logmod <- glm(formulav, data=trainset, family=binomial(logit))
> # Forecast over test data out-of-sample
> testset <- Default[-samplev, ]
> fcast <- predict(logmod, newdata=testset, type="response")
> # Calculate confusion matrix out-of-sample
> table(actual=!testset$default, forecast=(fcast < threshv))
       forecast
actual  FALSE TRUE
  FALSE    29  132
  TRUE      9 4830
```

Forecasting Errors

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 subject will not default: default = FALSE.

A *positive* result corresponds to rejecting the null hypothesis, while a *negative* result corresponds to accepting the null hypothesis.

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 there is no default but it's classified as a default.

A *type II* error is the incorrect acceptance of a FALSE *null hypothesis* (i.e. a "false negative"), when there is a default but it's classified as no default.

```
> # Calculate confusion matrix out-of-sample
> confmat <- table(actual=!testset$default,
+ forecast=(fcast < threshv))
> confmat
       forecast
actual  FALSE TRUE
  FALSE    29  132
  TRUE      9 4830
> # Calculate FALSE positive (type I error)
> sum(!testset$default & (fcast > threshv))
[1] 9
> # Calculate FALSE negative (type II error)
> sum(testset$default & (fcast < threshv))
[1] 132
```

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])
  typeI  typeII
0.00186 0.81988
> detach(Default)
```

Let the *null hypothesis* be that the subject will not default: `default = 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).

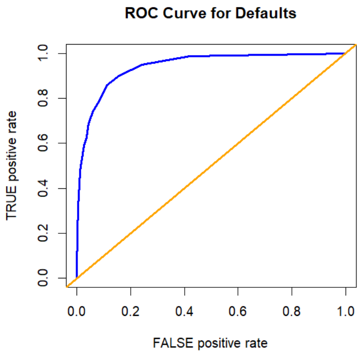
The sum of the *true negative* plus the *false positive* rate is equal to 1.

Receiver Operating Characteristic (ROC) Curve

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) is a measure of the performance of a binary classification model.

```
> # Confusion matrix as function of threshold
> confun <- function(actualv, fcast, threshv) {
+     confmat <- table(actualv, (fcast < threshv))
+     confmat <- confmat / rowSums(confmat)
+     c(typeI=confmat[2, 1], typeII=confmat[1, 2])
+   }  # end confun
> confun(!testset$default, fcast, threshv=threshv)
> # Define vector of discrimination thresholds
> threshv <- seq(0.05, 0.95, by=0.05)^2
> # Calculate error rates
> errorr <- sapply(threshv, confun,
+   actualv=!testset$default, fcast=fcast)  # end sapply
> errorr <- t(errorr)
> rownames(errorr) <- threshv
> errorr <- rbind(c(1, 0), errorr)
> errorr <- rbind(errorr, c(0, 1))
> # Calculate area under ROC curve (AUC)
> truepos <- (1 - errorr[, "typeII"])
> truepos <- (truepos + rutils::lagit(truepos))/2
> falsepos <- rutils::diffit(errorr[, "typeI"])
> abs(sum(truepos*falsepos))
```



```
> # Plot ROC Curve for Defaults
> x11(width=5, height=5)
> plot(x=errorr[, "typeI"], y=1-errorr[, "typeII"],
+      xlab="FALSE positive rate", ylab="TRUE positive rate",
+      main="ROC Curve for Defaults", type="l", lwd=3, col="blue")
> abline(a=0.0, b=1.0, lwd=3, col="orange")
```

Homework Assignment

Required

- Study all the lecture slides in *FRE6871_Lecture_5.pdf*, and run all the code in *FRE6871_Lecture_5.R*

Recommended

- Read about *PCA* in:
  pca-handout.pdf
  pcaTutorial.pdf
- Read about *optimization methods*:
  Bolker Optimization Methods.pdf
  Yollin Optimization.pdf
  Boudt DEoptim Large Portfolio Optimization.pdf