Machine Learning

FRE6871 & FRE7241, Fall 2022

Jerzy Pawlowski *jp3900@nyu.edu*

NYU Tandon School of Engineering

October 30, 2022

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:

$$\begin{aligned} \frac{d(\mathbf{v}^T\mathbb{1})}{d\mathbf{v}} = d_v[\mathbf{v}^T\mathbb{1}] = d_v[\mathbb{1}^T\mathbf{v}] &= \mathbb{1}^T \\ d_v[\mathbf{v}^T\mathbf{w}] = d_v[\mathbf{w}^T\mathbf{v}] &= \mathbf{w}^T \\ d_v[\mathbf{v}^T\mathbb{A}\,\mathbf{w}] &= \mathbf{w}^T\mathbb{A}^T \\ d_v[\mathbf{v}^T\mathbb{A}\,\mathbf{v}] &= \mathbf{v}^T\mathbb{A} + \mathbf{v}^T\mathbb{A}^T \end{aligned}$$

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.

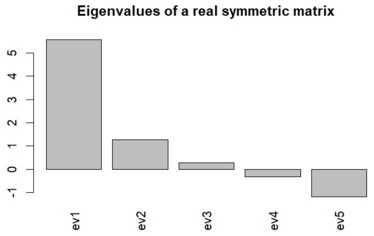
Real symmetric matrices have real *eigenvalues*, and their *eigenvectors* are orthogonal to each other.

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).

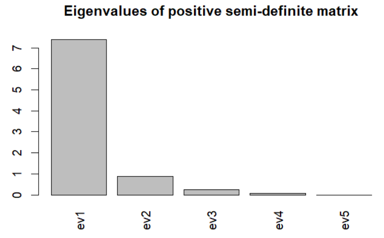
An example of *positive definite* matrices are the covariance matrices of linearly independent variables.

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
> nrows <- 6 ; ncols <- 4
> # Calculate left matrix
> leftmat <- matrix(runif(nrows^2), nc=nrows)
> eigend <- eigen(crossprod(leftmat))
> leftmat <- eigend$vectors[, 1:ncols]
> # Calculate right matrix and singular values
> rightmat <- matrix(runif(ncols^2), nc=ncols)
> eigend <- eigen(crossprod(rightmat))
> rightmat <- eigend$vectors
> singval <- sort(runif(ncols, 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$values[1:ncols], singval^2)
> all.equal(abs(eigend$vectors[, 1:ncols]), 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: nrows > ncols
> nrows <- 6 ; ncols <- 4
> matrixv <- matrix(runif(nrows*ncols), nc=ncols)
> # Calculate generalized inverse of matrixv
> invmat <- MASS::ginv(matrixv)
> round(invmat %*% matrixv, 4)
> all.equal(matrixv, matrixv %*% invmat %*% matrixv)
> # Random rectangular matrix: nrows < ncols
> nrows <- 4 ; ncols <- 6
> matrixv <- matrix(runif(nrows*ncols), nc=ncols)
> # 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: ncols > nrows
> nrows <- 4 ; ncols <- 6
> matrixv <- matrix(runif(nrows*ncols), nc=ncols)
> 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

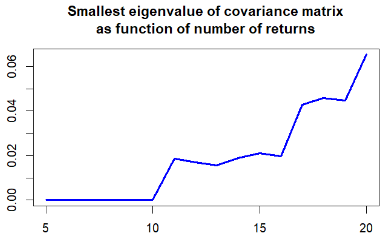
If $\mathbb{R}$ is a matrix of returns (with zero mean) for a portfolio of k assets (columns), over n time periods (rows), then the sample covariance matrix is equal to:

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

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")
```



```
> # 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.

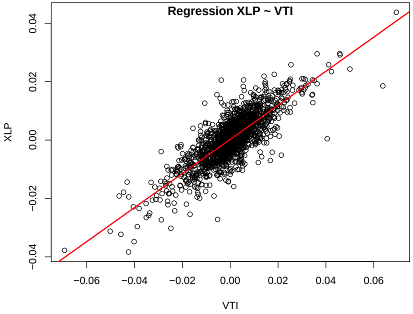
The *shrinkage* estimator reduces the estimate variance at the expense of increasing its bias (known as the *bias-variance tradeoff*).

```
> # 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)
```

draft: Principal Components for two assets

The scaled returns of *XLP* and *VTI* can be expressed as linear combinations of two orthogonal principal components:
The first principal component can be returns of *XLP* and *VTI* are highly correlated because they both share a common factor of market returns.

```
> retsp <- scale(na.omit(rutils::etfenv$returns
+           [, as.character(formulav)[-1]]))
> crossprod(retsp) / NROW(retsp)
> w1 <- sqrt(0.5); w2 <- w1
> foo <- matrix(c(w1, w2, -w2, w1), nc=2)
> t(foo) %*% foo
> # bar <- retsp %*% t(solve(foo))
> (t(bar) %*% bar) / NROW(bar)
>
> covmat <- function(retsp, anglev=0) {
+   w1 <- cos(anglev)
+   w2 <- sin(anglev)
+   matrixv <- matrix(c(w1, -w2, w2, w1), nc=2)
+   pcav <- retsp %*% t(matrixv)
+   (t(pcav) %*% pcav) / NROW(pcav)
+ }  # end covmat
>
> bar <- covmat(retsp, anglev=pi/4)
> crossprod(retsp) / NROW(retsp)
> (t(bar) %*% bar) / NROW(bar)
>
> angles <- seq(0, pi/2, by=pi/24)
> covmat <- sapply(angles, function(anglev)
+   covmat(retsp, anglev=anglev)[1, 1])
> plot(x=angles, y=covmat, t="l")
>
> optiml <- optimize(
+   f=function(anglev)
+     -covmat(retsp, anglev=anglev)[1, 1],
```



```
> # Plot scatterplot of returns
> plot(formulav, data=rutils::etfenv$returns,
+      main="Regression XLP ~ VTI")
> # Add regression line
> abline(regmod, lwd=2, col="red")
```

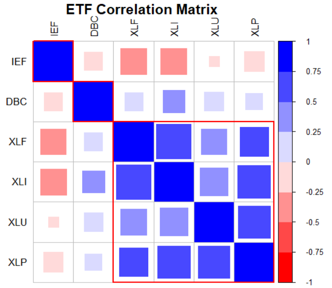
Covariance Matrix of ETF Returns

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
> pricev <- rutils::etfenv$prices[, symbolv]
> pricev <- zoo::na.locf(pricev, na.rm=FALSE)
> pricev <- zoo::na.locf(pricev, fromLast=TRUE)
> # Calculate log returns without standardizing
> retsp <- rutils::diffit(log(pricev))
> # 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(pricev))
> # Alternative (much slower) center (de-mean) and scale the retur
> # retsp <- scale(retsp, center=TRUE, scale=TRUE)
> # retsp <- xts::xts(retsp, zoo::index(pricev))
> # 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(pricev))
```



```
> # 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
> colorv <- 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=colorv(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 $\mathbf{r}_i$:

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

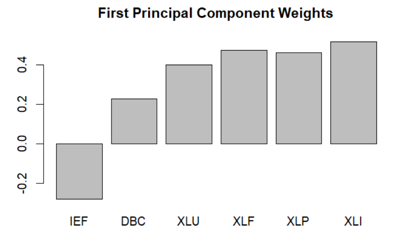
Where $\mathbf{w}_j$ is a vector of weights (loadings) of the *principal component* j, with $\mathbf{w}_j^T \mathbf{w}_j = 1$.

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)]
```



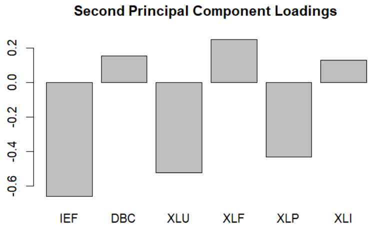
```
> # Find weights with maximum variance
> optiml <- optim(par=weights,
+   fn=objfun,
+   retsp=retsp,
+   method="L-BFGS-B",
+   upper=rep(10.0, nweights),
+   lower=rep(-10.0, nweights))
> # Optimal weights and maximum variance
> weights1 <- optiml$par
> -objfun(weights1, retsp)
> # Plot first principal 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*.

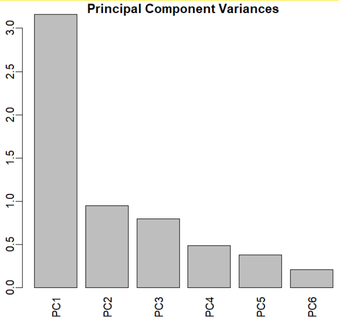
The maximum variance portfolio weights can be found by differentiating $\mathcal{L}$ with respect to $\mathbf{w}$ and setting it to zero:

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

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.

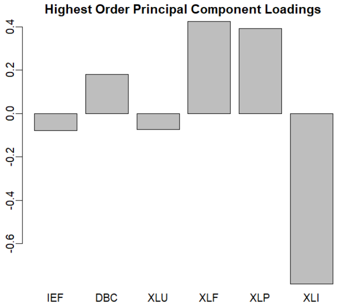
```
> # Eigen decomposition of correlation matrix
> eigend <- eigen(cormat)
> # Perform PCA with scaling
> pcad <- prcomp(retsp, scale=TRUE)
> # Compare outputs
> all.equal(eigend$values, pcad$sdev^2)
> all.equal(abs(eigend$vectors), abs(pcad$rotation),
+     check.attributes=FALSE)
> # Eigen decomposition 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

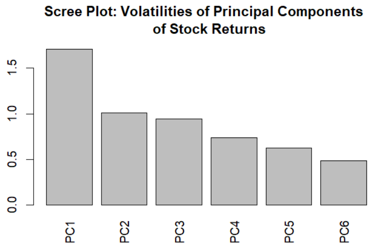
Principal Component Analysis (*PCA*) is a *dimension reduction* technique, that explains the returns of a large number of correlated time series as linear combinations of a smaller number of principal component time series.

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
```

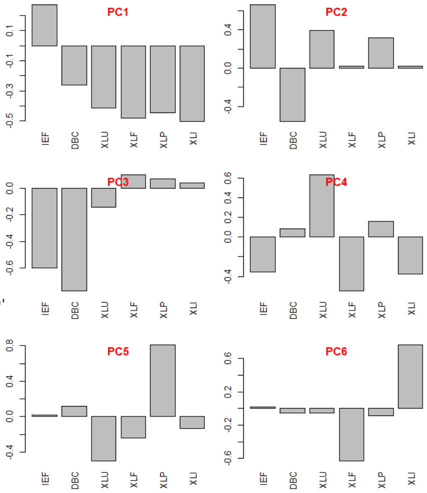
Principal Component Loadings (Weights)

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
```



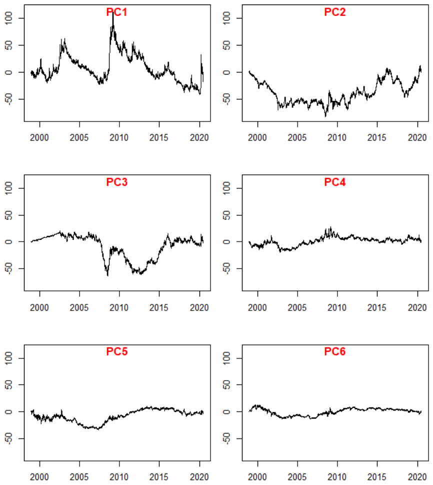
Principal Component Time Series

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(pricev)
> 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

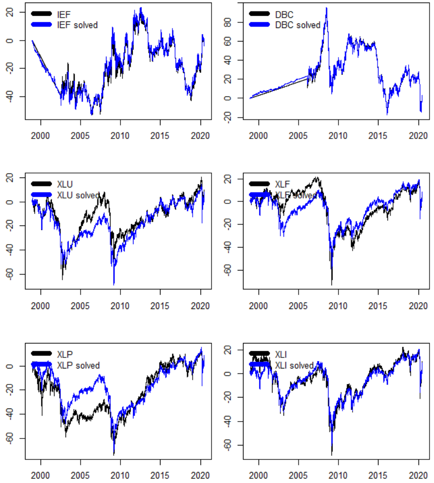
The original time series can be calculated exactly from the time series of all the *principal components*, by inverting the loadings matrix.

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")), y.intersp=0.5,
+    title=NULL, inset=0.0, cex=1.0, lwd=6, lty=1, col=c("black", "blue"))
+ }  # end for
```

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:

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

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

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

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

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*.

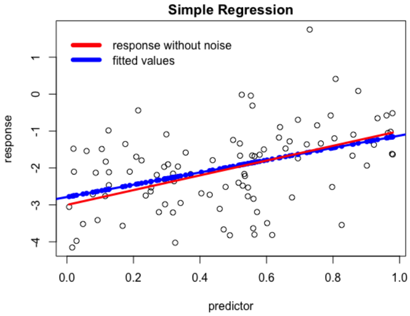
ε_i are the *residuals*, which are usually assumed to be standard normally distributed $\phi(0, \sigma_\varepsilon)$, independent, and stationary.

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

$class
[1] "lm"
> eval(regmod$call$formula)  # Regression formula
response ~ predictor
<environment: 0x121cfb318>
> regmod$coeff  # Regression coefficients
(Intercept)   predictor
      -2.79        1.67
> all.equal(coef(regmod), c(alpha, betav),
+   check.attributes=FALSE)
[1] TRUE
```

The *Fitted Values* of Linear Regression

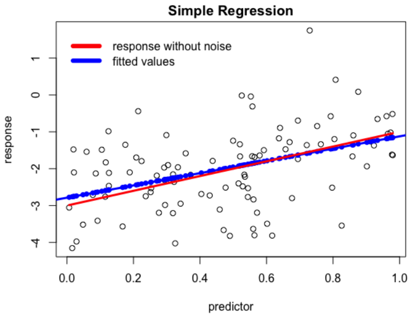
The *fitted values* y_{fit} are the estimates of the *response vector* obtained from the regression model:

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

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



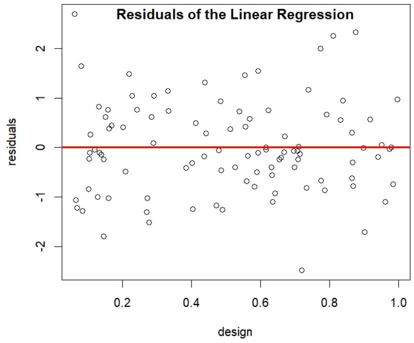
```
> # Plot response without noise
> lines(x=predictor, y=(response-noise), col="red", lwd=3)
> legend(x="topleft", # Add legend
+        legend=c("response without noise", "fitted values"),
+        title=NULL, inset=0.01, cex=1.0, lwd=6, y.intersp=0.5,
+        bty="n", lty=1, col=c("red", "blue"))
```

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, regmod$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, regmod$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 <- regmod$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(regmod)  # 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(regmodsum)$names  # get summary elements

Error in eval(expr, envir, enclos):  object 'regmodsum' not found
```

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(regmod)
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
```

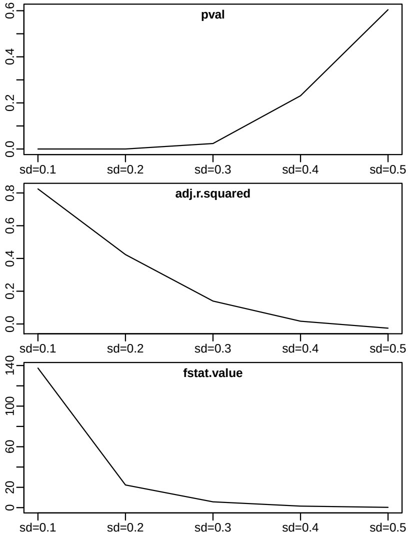
Weak Regression

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))
> regmod <- lm(formulav)  # Perform regression
> # Values of regression coefficients are not
> # Statistically significant
> summary(regmod)

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
```

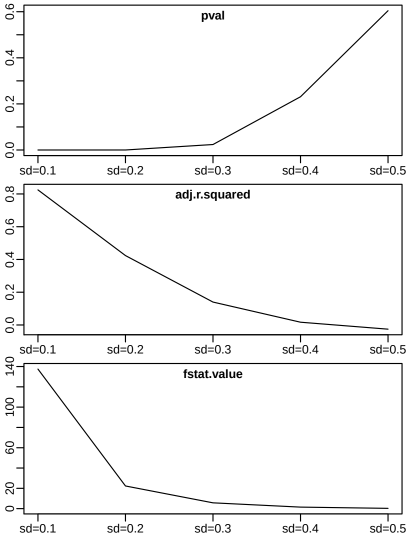
Influence of Noise on Regression

```
> 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(regmodsum, 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(regmodsum, 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.

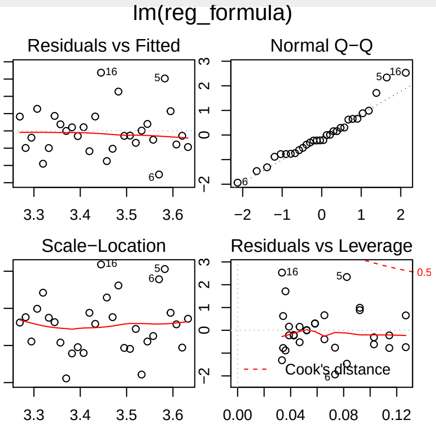
"Residuals vs Leverage" is a scatterplot of the residuals vs. their leverage.

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(regmod)  # Plot diagnostic scatterplots
> plot(regmod, 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(regmod)

Durbin-Watson test

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

draft: Autocorrelated Time Series Regression

Filtering or smoothing a time series containing an error terms over overlapping periods introduces autocorrelations in the error terms of the time series.

Autocorrelations in the error terms introduces autocorrelations of the regression residuals, causing the Durbin-Watson test to fail.

Autocorrelations in the error terms introduce autocorrelations of the regression residuals, causing the Durbin-Watson test to fail.

The failure of the Durbin-Watson test means that the *standard errors* and *p*-values calculated by the regression model are too small, and therefore the regression may not be statistically significant.

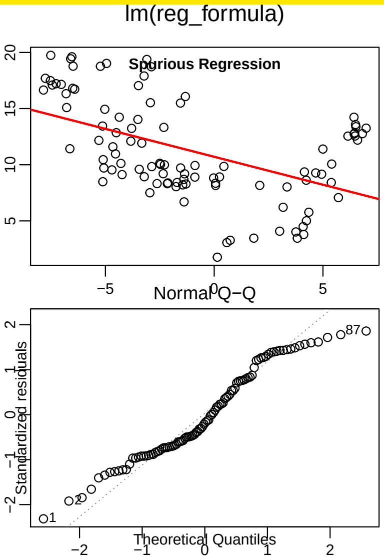
But the failure of the Durbin-Watson test doesn't reject the existence of a linear relationship between the response and predictor variables, it just puts it in doubt.

Links:
https://onlinecourses.science.psu.edu/stat510/node/72
http://stats.stackexchange.com/questions/6469/simple-linear-model-with-autocorrelated-errors-in-r

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



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:

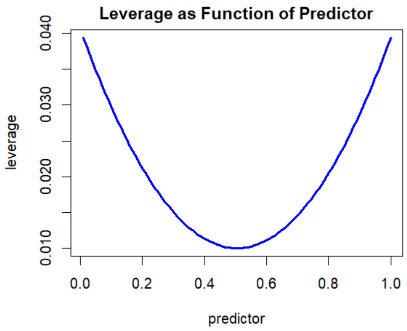
$$\mathbb{X}^T\mathbb{X} = \begin{pmatrix} n & 0 \\ 0 & \sum_{i=1}^n (x_i - \bar{x})^2 \end{pmatrix}$$

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:

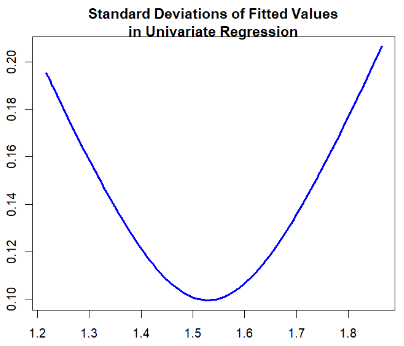
$$\sigma^2_{fit} = \frac{\mathbb{E}[\mathbb{H}\hat{\varepsilon}(\mathbb{H}\hat{\varepsilon})^T]}{d_{free}} = \frac{\mathbb{E}[\mathbb{H}\,\hat{\varepsilon}\hat{\varepsilon}^T\,\mathbb{H}^T]}{d_{free}} =$$
$$\frac{\mathbb{H}\,\mathbb{E}[\hat{\varepsilon}\hat{\varepsilon}^T]\,\mathbb{H}^T}{d_{free}} =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

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
```

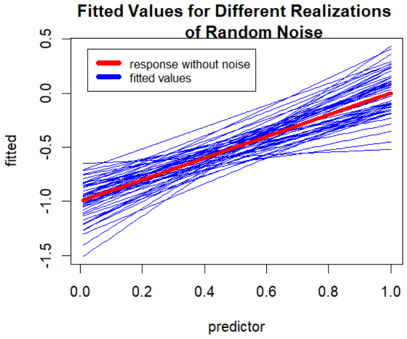
Fitted Values for Different Realizations of Random Noise

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 regressi
> # 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, y.intersp=0.5,
+        bty="n", 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:

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

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

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

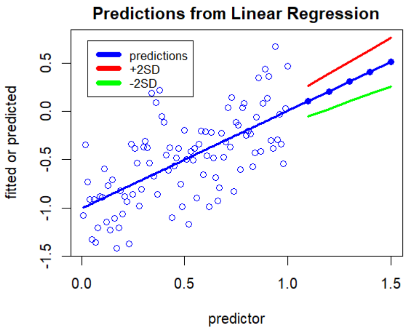
The variance σ^2_{pred} of the *predicted value* is equal to the *predictor* vector multiplied by the *covariance matrix* of the *regression coefficients* σ^2_{β}.

```
> # 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, y.intersp=0.5,
+        bty="n", lty=1, col=c("blue", "red", "green"))
```

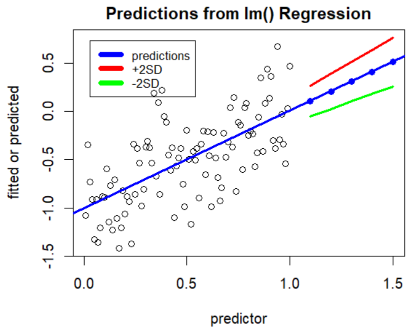


Predictions From *Linear Regression* Using Function `lm()`

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]
> regmod <- 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(regmod, 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, y.intersp=0.5,
+        bty="n", lty=1, col=c("blue", "red", "green"))
```

Spurious Time Series Regression

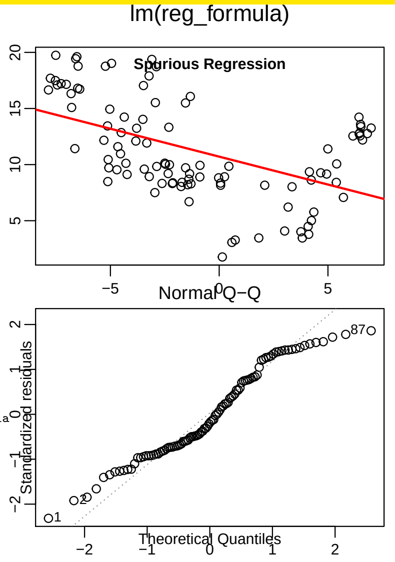
Regression of non-stationary time series creates *spurious* regressions.

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
> regmod <- lm(formulav)  # Perform regression
> # Summary indicates statistically significant regression
> modelsum <- summary(regmod)
> modelsum$coeff
> modelsum$r.squared
> # Durbin-Watson test shows residuals are autocorrelated
> dwtest <- lmtest::dwtest(regmod)
> c(dwtest$statistic[[1]], dwtest$p.value)
> plot(formulav, xlab="", ylab="")  # Plot scatterplot using formula
> title(main="Spurious Regression", line=-1)
> # Add regression line
> abline(regmod, lwd=2, col="red")
> plot(regmod, 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()
> regmod <- 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(regmod), 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
> regmod <- lm(response ~ predictor - 1)
> all.equal(drop(betas), 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
> fittedv <- drop(predictor %*% betas)
> all.equal(fittedv, regmod$fitted.values, check.attributes=FALSE)
[1] TRUE
> # Calculate the residuals
> residuals <- drop(response - fittedv)
> all.equal(residuals, regmod$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, regmod$fitted.values, check.attributes=FALSE)
[1] TRUE
> # Calculate fitted values from regression coefficients
> fittedv <- drop(predictor %*% betas)
> all.equal(fittedv, regmod$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,
+   regmod$fitted.values - mean(response),
+   check.attributes=FALSE)
[1] TRUE
> # Calculate the residuals
> respzm <- response - mean(response)
> residuals <- drop(respzm - fitted_zm)
> all.equal(residuals, regmod$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
```

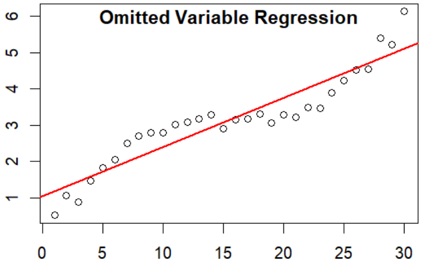
Omitted Variable Bias

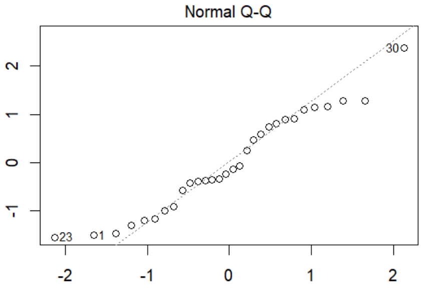
Omitted Variable Bias occurs in a regression model that omits important predictors.

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(regmod_ovb)
> modelsum$coeff
> modelsum$r.squared
> # Durbin-Watson test shows residuals are autocorrelated
> lmtest::dwtest(regmod_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(regmod_ovb, lwd=2, col="red")
> title(main="Omitted Variable Regression", line=-1)
> plot(regmod_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(regmod)
> # 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:

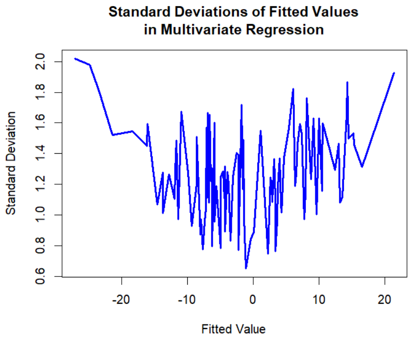
$$\sigma^2_{fit} = \frac{\mathbb{E}[\mathbb{X}\mathbb{X}_{inv}\hat{\varepsilon}\,(\mathbb{X}\mathbb{X}_{inv}\hat{\varepsilon})^T]}{d_{free}} = \frac{\mathbb{E}[\mathbb{H}\,\hat{\varepsilon}\hat{\varepsilon}^T\,\mathbb{H}^T]}{d_{free}} =$$
$$\frac{\mathbb{H}\,\mathbb{E}[\hat{\varepsilon}\hat{\varepsilon}^T]\,\mathbb{H}^T}{d_{free}}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

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 
```

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$returns[, c("XLF", "XLE")]
> retsp <- na.omit(retsp)
> nrows <- NROW(retsp)
> head(retsp)
> # Define regression formula
> formulav <- paste(colnames(retsp)[1],
+   paste(colnames(retsp)[-1], collapse="+"),
+   sep=" ~ ")
> # Standard regression
> regmod <- lm(formulav, data=retsp)
> modelsum <- summary(regmod)
> # Bootstrap of regression
> set.seed(1121)  # initialize random number generator
> bootd <- sapply(1:100, function(x) {
+   samplev <- sample.int(nrows, replace=TRUE)
+   regmod <- lm(formulav, data=retsp[samplev, ])
+   regmod$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:

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

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
> regmod <- lm(formulav, data=data.frame(cbind(response, predictor)
> modelsum <- summary(regmod)
> # 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)
> all.equal(predictlm[1, "lwr"], predictlow)
> all.equal(predictlm[1, "upr"], predicthigh)
```

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

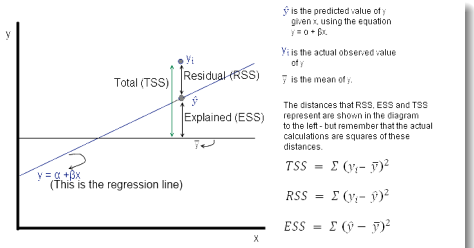
$$(y - y_{fit})^T (y_{fit} - \bar{y}) = 0$$

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((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(regmod$terms)$intercept <- 1
> # Regression summary
> modelsum <- summary(regmod)
> # 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:

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

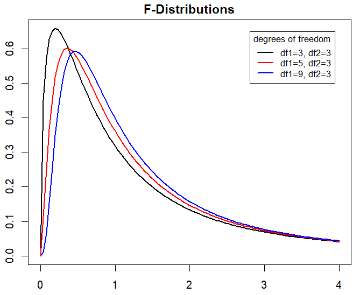
Follows the *F-distribution* with m and n degrees of freedom, with the probability density function:

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

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
> colorv <- 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=colorv[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",
+        y.intersp=0.5, bty="n", labelv, cex=0.8, lwd=2, lty=1, col=
```

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 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 only has 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).

```
> # 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.

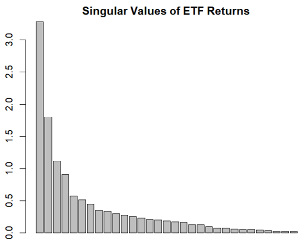
The *regularized inverse* is obtained by removing very small *singular values*:

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

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* $\mathbb{X}$:

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

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
> 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
```

draft: Principal Component Regression

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

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

The matrix $\mathbb{W}$ is chosen so that the columns (principal components) of the *PCA predictor matrix* $\mathbb{X}_{pca}$ form an orthogonal set of predictors.

The *PCA predictor matrix* $\mathbb{X}_{pca}$ produces the same *influence matrix* $\mathbb{H}$ as the original *predictor matrix* $\mathbb{X}$, so *Principal Component Regression* is equivalent to the original regression model.

Principal Component Regression is used to:

- Remove *collinearity* between predictors, by excluding principal components with very small eignevalues,
- Perform *dimension reduction*, by including only principal components with significantly large eignevalues,
- Reduce model overfitting, by reducing the number of predictors and parameters.

```
> # Perform PCA
> pcad <- prcomp(predictor, center=FALSE, scale=FALSE)
> predictorpcr <- pcad$x
> # Principal components are orthogonal to each other
> round(t(predictorpcr) %*% predictorpcr, 2)
    PC1 PC2 PC3  PC4  PC5  PC6
PC1 143   0   0  0.0  0.0  0.0
PC2   0 121   0  0.0  0.0  0.0
PC3   0   0 110  0.0  0.0  0.0
PC4   0   0   0 94.2  0.0  0.0
PC5   0   0   0  0.0 76.4  0.0
PC6   0   0   0  0.0  0.0 67.2
> round(apply(predictorpcr, 2,
+   function(x) c(mean=mean(x), sd=sd(x))), 3)
       PC1   PC2    PC3    PC4   PC5   PC6
mean 0.003 0.233 -0.761 -0.551 0.260 0.110
sd   1.201 1.079  0.726  0.803 0.839 0.816
> # Calculate the PCR influence matrix
> influence_pcr <- predictorpcr %*% MASS::ginv(predictorpcr)
> all.equal(influencem, influence_pcr)
[1] TRUE
```

draft: Principal Component Regression With Shrinkage

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

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

The matrix $\mathbb{W}$ is chosen so that the columns (principal components) of the *PCA predictor matrix* $\mathbb{X}_{pca}$ form an orthogonal set of predictors.

The *PCA predictor matrix* $\mathbb{X}_{pca}$ produces the same *influence matrix* $\mathbb{H}$ as the original *predictor matrix* $\mathbb{X}$, so *Principal Component Regression* is equivalent to the original regression model.

Principal Component Regression is used to:

- Remove *collinearity* between predictors, by excluding principal components with very small eignevalues,
- Perform *dimension reduction*, by including only principal components with significantly large eignevalues,
- Reduce model overfitting, by reducing the number of predictors and parameters.

```
> # Perform PCA
> pcad <- prcomp(predictor, center=FALSE, scale=FALSE)
> predictorpcr <- pcad$x
> # Principal components are orthogonal to each other
> round(t(predictorpcr) %*% predictorpcr, 2)
    PC1 PC2 PC3  PC4  PC5  PC6
PC1 143   0   0  0.0  0.0  0.0
PC2   0 121   0  0.0  0.0  0.0
PC3   0   0 110  0.0  0.0  0.0
PC4   0   0   0 94.2  0.0  0.0
PC5   0   0   0  0.0 76.4  0.0
PC6   0   0   0  0.0  0.0 67.2
> round(apply(predictorpcr, 2,
+   function(x) c(mean=mean(x), sd=sd(x))), 3)
       PC1   PC2    PC3    PC4   PC5   PC6
mean 0.003 0.233 -0.761 -0.551 0.260 0.110
sd   1.201 1.079  0.726  0.803 0.839 0.816
> # Calculate the PCR influence matrix
> influence_pcr <- predictorpcr %*% MASS::ginv(predictorpcr)
> all.equal(influencem, influence_pcr)
[1] TRUE
```

draft: Standard Errors of Principal Component Regression

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

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

The matrix $\mathbb{W}$ is chosen so that the columns (principal components) of the *PCA predictor matrix* $\mathbb{X}_{pca}$ form an orthogonal set of predictors.

The *PCA predictor matrix* $\mathbb{X}_{pca}$ produces the same *influence matrix* $\mathbb{H}$ as the original *predictor matrix* $\mathbb{X}$, so *Principal Component Regression* is equivalent to the original regression model.

Principal Component Regression is used to:

- Remove *collinearity* between predictors, by excluding principal components with very small eignevalues,
- Perform *dimension reduction*, by including only principal components with significantly large eignevalues,
- Reduce model overfitting, by reducing the number of predictors and parameters.

```
> # Perform PCA
> pcad <- prcomp(predictor, center=FALSE, scale=FALSE)
> predictorpcr <- pcad$x
> # Principal components are orthogonal to each other
> round(t(predictorpcr) %*% predictorpcr, 2)
    PC1 PC2 PC3  PC4  PC5  PC6
PC1 143   0   0  0.0  0.0  0.0
PC2   0 121   0  0.0  0.0  0.0
PC3   0   0 110  0.0  0.0  0.0
PC4   0   0   0 94.2  0.0  0.0
PC5   0   0   0  0.0 76.4  0.0
PC6   0   0   0  0.0  0.0 67.2
> round(apply(predictorpcr, 2,
+   function(x) c(mean=mean(x), sd=sd(x))), 3)
       PC1   PC2    PC3    PC4   PC5   PC6
mean 0.003 0.233 -0.761 -0.551 0.260 0.110
sd   1.201 1.079  0.726  0.803 0.839 0.816
> # Calculate the PCR influence matrix
> influence_pcr <- predictorpcr %*% MASS::ginv(predictorpcr)
> all.equal(influencem, influence_pcr)
[1] TRUE
```

draft: Forecasting Using Principal Component Regression

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

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

The matrix $\mathbb{W}$ is chosen so that the columns (principal components) of the *PCA predictor matrix* $\mathbb{X}_{pca}$ form an orthogonal set of predictors.

The *PCA predictor matrix* $\mathbb{X}_{pca}$ produces the same *influence matrix* $\mathbb{H}$ as the original *predictor matrix* $\mathbb{X}$, so *Principal Component Regression* is equivalent to the original regression model.

Principal Component Regression is used to:

- Remove *collinearity* between predictors, by excluding principal components with very small eignevalues,
- Perform *dimension reduction*, by including only principal components with significantly large eignevalues,
- Reduce model overfitting, by reducing the number of predictors and parameters.

```
> # Perform PCA
> pcad <- prcomp(predictor, center=FALSE, scale=FALSE)
> predictorpcr <- pcad$x
> # Principal components are orthogonal to each other
> round(t(predictorpcr) %*% predictorpcr, 2)
    PC1 PC2 PC3  PC4  PC5  PC6
PC1 143   0   0  0.0  0.0  0.0
PC2   0 121   0  0.0  0.0  0.0
PC3   0   0 110  0.0  0.0  0.0
PC4   0   0   0 94.2  0.0  0.0
PC5   0   0   0  0.0 76.4  0.0
PC6   0   0   0  0.0  0.0 67.2
> round(apply(predictorpcr, 2,
+   function(x) c(mean=mean(x), sd=sd(x))), 3)
       PC1   PC2    PC3    PC4   PC5   PC6
mean 0.003 0.233 -0.761 -0.551 0.260 0.110
sd   1.201 1.079  0.726  0.803 0.839 0.816
> # Calculate the PCR influence matrix
> influence_pcr <- predictorpcr %*% MASS::ginv(predictorpcr)
> all.equal(influencem, influence_pcr)
[1] TRUE
```

The *Logistic* Function

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

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

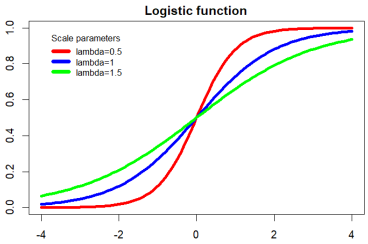
Where λ is the scale (dispersion) parameter.

The *logistic* function is often used as an activation function in neural networks, and logistic regression can be viewed as 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,



```
> lambdas <- 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=lambdas[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", lambdas, sep="="), y.intersp=0.5,
+        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.

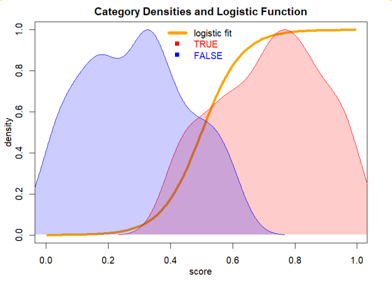
The function glm() fits generalized linear models, including *logistic* regressions.

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.

```
> set.seed(1121)  # Reset random number generator
> # 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
> predictor <- c(sample1, sample2)
> response <- c(logical(100), !logical(100))
> # Perform logit regression
> logmod <- glm(response ~ predictor, family=binomial(logit))
> class(logmod)
> summary(logmod)
```



```
> ordern <- order(predictor)
> plot(x=predictor[ordern], y=logmod$fitted.values[ordern],
+      main="Category Densities and Logistic Function",
+      type="l", lwd=4, col="orange", xlab="predictor", ylab="densit
> densityv <- density(predictor[response])
> densityv$y <- densityv$y/max(densityv$y)
> lines(densityv, col="red")
> polygon(c(min(densityv$x), densityv$x, max(densityv$x)), c(min(de
> densityv <- density(predictor[!response])
> densityv$y <- densityv$y/max(densityv$y)
> lines(densityv, col="blue")
> polygon(c(min(densityv$x), densityv$x, max(densityv$x)), c(min(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=0.5,
+  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 $(1 - p)$.

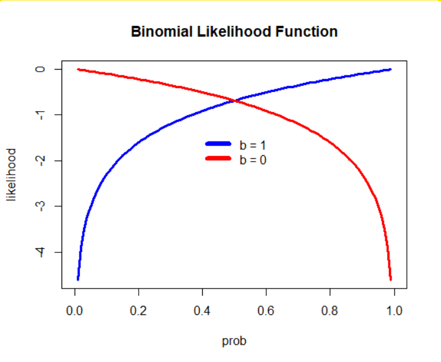
Then b follows the binomial distribution:

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

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=0.5,
+        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
> predictor=cbind(intercept=rep(1, NROW(response)), predictor)
> # Likelihood function of the logistic model
> likefun <- function(coeff, response, predictor) {
+   probs <- plogis(drop(predictor %*% coeff))
+   -sum(response*log(probs) + (1-response)*log((1-probs)))
+ }  # end likefun
> # Run likelihood function
> coeff <- c(1, 1)
> likefun(coeff, response, predictor)
```

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(vectorv, param=25) {
+   sum(vectorv^2 - param*cos(vectorv))
+ }  # end rastrigin
> vectorv <- c(pi/6, pi/6)
> rastrigin(vectorv=vectorv)
> # Draw 3d surface plot of Rastrigin function
> options(rgl.useNULL=TRUE); library(rgl)
> rgl::persp3d(
+   x=Vectorize(function(x, y) rastrigin(vectorv=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=vectorv, 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
+            response=response,
+            predictor=predictor,
+            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$hessian)))
> modelsum <- summary(logmod)
> modelsum$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().

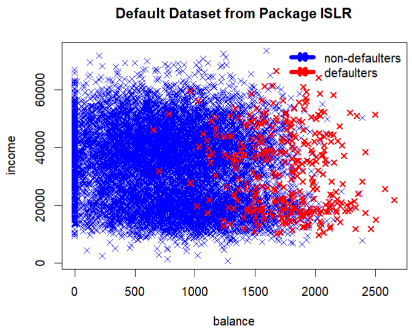
```
> # Coerce the student and default columns into Boolean
> Default <- ISLR::Default
> Default$default <- (Default$default == "Yes")
> Default$student <- (Default$student == "Yes")
> colnames(Default)[1:2] <- c("default", "student")
> attach(Default)  # Attach Default to search path
> # Explore credit 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=0.5, bty="n", col=c("blue", "red"), lty=1, lwd=6, pch=4
```

Boxplots of the Default Dataset

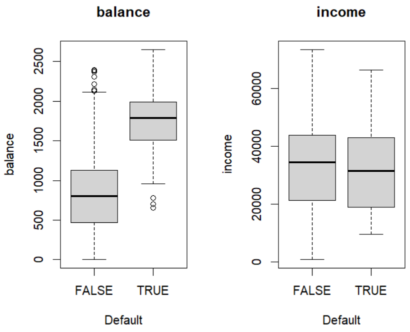
A *Box Plot* (box-and-whisker plot) is a graphical display of a distribution of data:

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")
```

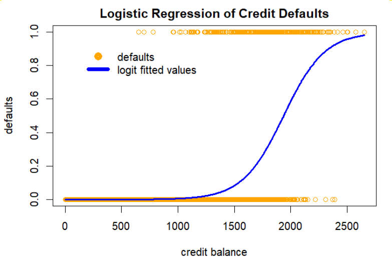
Modeling Credit Defaults Using *Logistic* Regression

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))

Deviance Residuals:
   Min      1Q  Median      3Q     Max
-2.270  -0.146  -0.059  -0.022   3.759

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
```



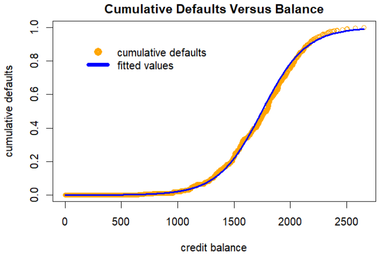
```
> x11(width=6, height=5)
> par(mar=c(4, 4, 2, 2), oma=c(0, 0, 0, 0), mgp=c(2.5, 1, 0))
> plot(x=balance, y=default,
+      main="Logistic Regression of Credit Defaults",
+      col="orange", xlab="credit balance", ylab="defaults")
> ordern <- order(balance)
> lines(x=balance[ordern], y=logmod$fitted.values[ordern], col="blue
> legend(x="topleft", inset=0.1, bty="n", lwd=6, y.intersp=0.5,
+  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=0.5,
+  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
<environment: 0x121cfb318>
> logmod <- glm(formulav, data=Default, family=binomial(logit))
> summary(logmod)

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

Deviance Residuals:
   Min      1Q  Median      3Q     Max
-2.469  -0.142  -0.056  -0.020   3.738

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).

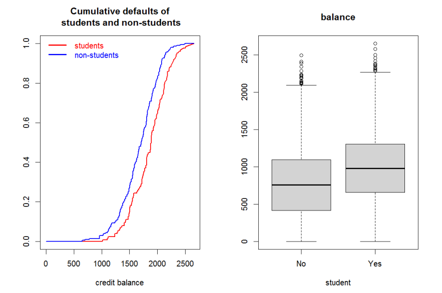
The reason that students are more likely to default is because they have higher credit balances than non-students - which is what the single-factor regression shows.

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
```



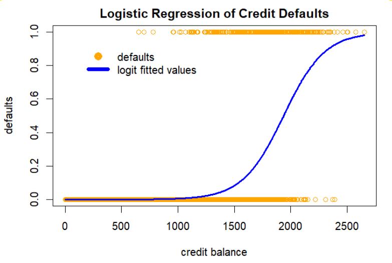
```
> # Calculate cumulative defaults
> cum_defaults <- sapply(balance, function(balv) {
+ c(student=sum(default[student & (balance <= balv)]),
+   non_student=sum(default[!student & (balance <= balv)]))
+ })  # end sapply
> total_defaults <- c(student=sum(student & default),
+       student=sum(!student & 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=0.5,
+  legend=c("students", "non-students"),
+  col=c("red", "blue"), text.col=c("red", "blue"), lwd=3)
> # Balance boxplot for student factor
```

draft: Modeling Credit Defaults Using Student Status

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

Persons who are students are more likely to default because students have higher credit balances.

```
> # Fit logistic regression model
> logmod <- glm(default ~ student, family=binomial(logit))
> summary(logmod)
```



```
> x11(width=6, height=5)
> par(mfrow=c(1,2))  # Set plot panels
> # Balance boxplot
> boxplot(formula=balance ~ default,
+   col="lightgrey", main="balance", xlab="Default")
>
>
> # Plot data points for non-students
> x11(width=6, height=5)
> 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[!student, ])
> # Plot data points for students
> points(income ~ balance, pch=4, lwd=2, col="red",
+  data=Default[student, ])
> # Add legend
> legend(x="topright", bty="n", y.intersp=0.5,
+  legend=c("non-students", "students"),
+  col=c("blue", "red"), lty=1, lwd=6, pch=4)
```

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
> forecastv <- predict(logmod, type="response")
> all.equal(logmod$fitted.values, forecastv)
[1] TRUE
> # Define discrimination threshold value
> threshold <- 0.7
> # Calculate confusion matrix in-sample
> table(actual=!default, forecast=(forecastv < threshold))
       forecast
actual  FALSE TRUE
  FALSE    57  276
  TRUE     12 9655
> # Fit logistic regression over training data
> set.seed(1121)  # Reset random number generator
> nrows <- NROW(Default)
> samplev <- sample.int(n=nrows, size=nrows/2)
> trainset <- Default[samplev, ]
> logmod <- glm(formulav, data=trainset, family=binomial(logit))
> # Forecast over test data out-of-sample
> testset <- Default[-samplev, ]
> forecastv <- predict(logmod, newdata=testset, type="response")
> # Calculate confusion matrix out-of-sample
> table(actual=!testset$default, forecast=(forecastv < threshold))
       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$default,
+ forecast=(forecastv < threshold))
> confmat
       forecast
actual  FALSE TRUE
  FALSE    29  132
  TRUE      9 4830
> # Calculate FALSE positive (type I error)
> sum(!testset$default & (forecastv > threshold))
[1] 9
> # Calculate FALSE negative (type II error)
> sum(testset$default & (forecastv < threshold))
[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).

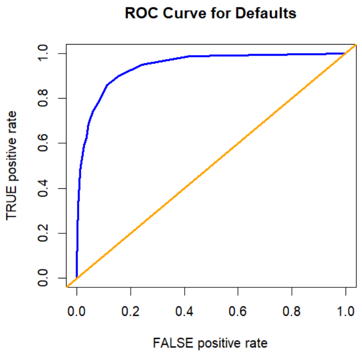
The sum of the *true negative* plus the *false positive* rate is equal to 1.

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, forecastv, threshold) {
+     confmat <- table(actualv, (forecastv < threshold))
+     confmat <- confmat / rowSums(confmat)
+     c(typeI=confmat[2, 1], typeII=confmat[1, 2])
+   }  # end confun
> confun(!testset$default, forecastv, threshold=threshold)
> # Define vector of discrimination thresholds
> threshv <- seq(0.05, 0.95, by=0.05)^2
> # Calculate error rates
> error_rates <- sapply(threshv, confun,
+   actualv=!testset$default, forecastv=forecastv)  # end sapply
> error_rates <- t(error_rates)
> rownames(error_rates) <- threshv
> error_rates <- rbind(c(1, 0), error_rates)
> error_rates <- rbind(error_rates, c(0, 1))
> # Calculate area under ROC curve (AUC)
> 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 Defaults
> x11(width=5, height=5)
> plot(x=error_rates[, "typeI"], y=1-error_rates[, "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")
```

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
> taq
> class(taq)
> colnames(taq)
> sapply(taq, class)
> symbol <- taq$SYM_ROOT[1]
> # Create date-time index
> dates <- paste(taq$DATE, taq$TIME_M)
> # Coerce date-time index to POSIXlt
> dates <- strptime(dates, "%Y%m%d %H:%M:%OS")
> class(dates)
> # Display more significant digits
> # options("digits")
> options(digits=20, digits.secs=10)
> last(dates)
> unclass(last(dates))
> as.numeric(last(dates))
> # Coerce date-time index to POSIXct
> dates <- as.POSIXct(dates)
> class(dates)
> last(dates)
> unclass(last(dates))
> as.numeric(last(dates))
> # Calculate the number of ticks per second
> nsecs <- as.numeric(last(dates)) - as.numeric(first(dates))
> NROW(taq)/(6.5*3600)
> # Select TAQ data columns
> taq <- taq[, .(price=PRICE, volume=SIZE)]
> # Add date-time index
> taq <- cbind(index=dates, taq)
```

Microstructure Noise in High Frequency Data

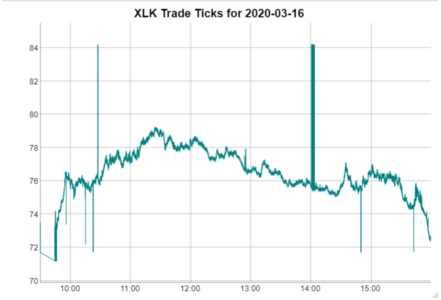
High frequency data contains *microstructure noise* in the form of *price jumps* and the *bid-ask bounce*.

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

Price jumps are often caused by data collection errors, but sometimes they represent actual very large lot trades.

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 fact the mid price is constant.



```
> # Coerce trade ticks to xts series
> xtsv <- xts::xts(taq[, .(price, volume)], taq$index)
> colnames(xtsv) <- paste(symbol, c("Close", "Volume"), sep=".")
> save(xtsv, file="/Users/jerzy/Develop/data/xlk_tick_trades2020_031
> # save(xtsv, file="/Users/jerzy/Develop/data/xlk_tick_trades2020_(
> # Plot dygraph
> dygraphs::dygraph(xtsv$XLK.Close,
+   main="XLK Trade Ticks for 2020-03-16")
> # Plot in x11 window
> x11(width=6, height=5)
> quantmod::chart_Series(x=xtsv$XLK.Close,
+   name="XLK Trade Ticks for 2020-03-16")
```

Removing Microstructure Noise From High Frequency Data

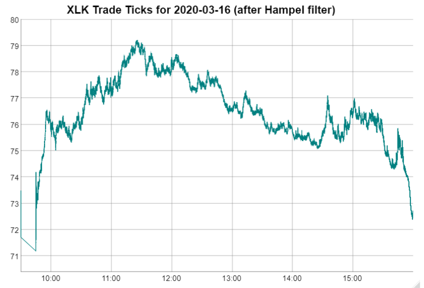
Microstructure noise can be removed from high frequency data by using a *Hampel filter*.

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

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

If the *z-score* exceeds the *threshold value* then it's classified as an *outlier* (jump in prices).

```
> # Calculate centered Hampel filter to remove price jumps
> look_back <- 111
> half_back <- look_back %/% 2
> medianv <- roll::roll_median(taq$price, width=look_back)
> # medianv <- TTR::runMedian(taq$price, n=look_back)
> medianv <- rutils::lagit(medianv, lagg=(-half_back), pad_zeros=F
> madv <- HighFreq::roll_var(matrix(taq$price), look_back=look_bac
> # madv <- TTR::runMAD(taq$price, n=look_back)
> madv <- rutils::lagit(madv, lagg=(-half_back), pad_zeros=FALSE)
> # Calculate Z-scores
> zscores <- (taq$price - medianv)/madv
> zscores[is.na(zscores)] <- 0
> zscores[!is.finite(zscores)] <- 0
> sum(is.na(zscores))
> sum(!is.finite(zscores))
> range(zscores); mad(zscores)
> hist(zscores, breaks=2000, xlim=c(-5*mad(zscores), 5*mad(zscores
```



```
> # Define discrimination threshold value
> threshold <- 6*mad(zscores)
> # Remove price jumps with large z-scores
> bad_ticks <- (abs(zscores) > threshold)
> good_ticks <- taq[!bad_ticks]
> # Calculate number of price jumps
> sum(bad_ticks)/NROW(zscores)
> # Coerce trade prices to xts
> xtsv <- xts::xts(good_ticks[, .(price, volume)], good_ticks$index)
> colnames(xtsv) <- c("XLK.Close", "XLK.Volume")
> # Plot dygraph of the clean lots
> dygraphs::dygraph(xtsv$XLK.Close,
+   main="XLK Trade Ticks for 2020-03-16 (Hampel filtered)")
> # Plot the large lots
> x11(width=6, height=5)
> quantmod::chart_Series(x=xtsv$XLK.Close,
+   name="XLK Trade Ticks for 2020-03-16 (Hampel filtered)")
```

Classifying Data Outliers Using the Hampel Filter

The data points whose absolute *z-scores* exceed a *threshold value* are classified as outliers.

This procedure is a *classifier*, which classifies the prices as either good or bad data points.

If the bad data points are not labeled, then we can add jumps to the data to test the performance of the classifier.

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.

```
> # Define discrimination threshold value
> threshold <- 6*mad(zscores)
> # Calculate number of prices classified as bad data
> isbad <- (abs(zscores) > threshold)
> sum(isbad)
> # Add 200 random price jumps into prices
> set.seed(1121)
> nbad <- 200
> isjump <- logical(NROW(closep))
> isjump[sample(x=NROW(isjump), size=nbad)] <- TRUE
> closep[isjump] <- closep[isjump]*
+   sample(c(0.95, 1.05), size=nbad, replace=TRUE)
> # Plot prices and medians
> dygraphs::dygraph(cbind(closep, medianv), main="VTI median") %>%
+   dyOptions(colors=c("black", "red"))
> # Calculate time series of z-scores
> medianv <- roll::roll_median(closep, width=look_back)
> # medianv <- TTR::runMedian(closep, n=look_back)
> madv <- HighFreq::roll_var(closep, look_back=look_back, method="nc
> # madv <- TTR::runMAD(closep, n=look_back)
> zscores <- (closep - medianv)/madv
> zscores[1:look_back, ] <- 0
> # Calculate number of prices classified as bad data
> isbad <- (abs(zscores) > threshold)
> sum(isbad)
```

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=!isjump, forecast=!isbad)
> sum(isbad)
> # FALSE positive (type I error)
> sum(!isjump & isbad)
> # FALSE negative (type II error)
> sum(isjump & !isbad)
```

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).

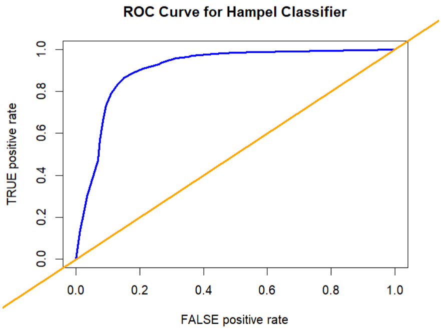
The sum of the *true negative* plus the *false positive* rate is equal to 1.

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, threshold) {
+     confmat <- table(!actualv, !(abs(zscores) > threshold))
+     confmat <- confmat / rowSums(confmat)
+     c(typeI=confmat[2, 1], typeII=confmat[1, 2])
+   }  # end confun
> confun(isjump, zscores, threshold=threshold)
> # Define vector of discrimination thresholds
> threshv <- seq(from=0.2, to=5.0, by=0.2)
> # Calculate error rates
> error_rates <- sapply(threshv, confun,
+   actualv=isjump, zscores=zscores)  # end sapply
> error_rates <- t(error_rates)
> rownames(error_rates) <- threshv
> error_rates <- rbind(c(1, 0), error_rates)
> error_rates <- rbind(error_rates, c(0, 1))
> # Calculate area under ROC curve (AUC)
> 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 Hampel classifier
> x11(width=6, height=5)
> plot(x=error_rates[, "typeI"], y=1-error_rates[, "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")
```

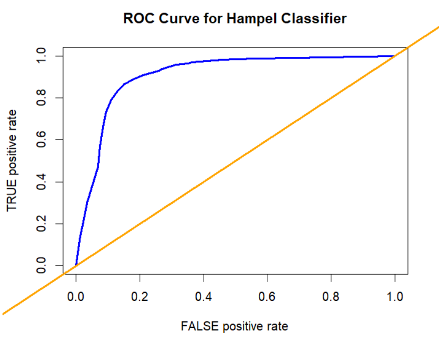
draft: Receiver Operating Characteristic (ROC) Curve

The performance of the Hampel noise classification model depends on the length of the look-back time interval. a binary. *area under the ROC curve* (AUC) is a measure of a binary. The optimal *threshold value* can be determined using *cross-validation*.

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, threshold) {
+     confmat <- table(!actualv, !(abs(zscores) > threshold))
+     confmat <- confmat / rowSums(confmat)
+     c(typeI=confmat[2, 1], typeII=confmat[1, 2])
+   }  # end confun
> confun(isjump, zscores, threshold=threshold)
> # Define vector of discrimination thresholds
> threshv <- seq(from=0.2, to=5.0, by=0.2)
> # Calculate error rates
> error_rates <- sapply(threshv, confun,
+   actualv=isjump, zscores=zscores)  # end sapply
> error_rates <- t(error_rates)
> rownames(error_rates) <- threshv
> error_rates <- rbind(c(1, 0), error_rates)
> error_rates <- rbind(error_rates, c(0, 1))
> # Calculate area under ROC curve (AUC)
> truepos <- (1 - error_rates[, "typeII"])
> truepos <- (truepos + rutils::lagit(truepos))/2
> falsepos <- rutils::diffit(error_rates[, "typeI"])
> abs(sum(truepos*falsepos))
```



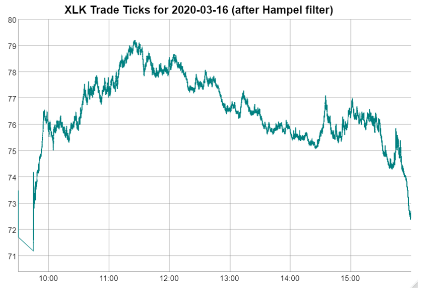
```
> # Plot ROC curve for Hampel classifier
> x11(width=6, height=5)
> plot(x=error_rates[, "typeI"], y=1-error_rates[, "typeII"],
+      xlab="FALSE positive rate", ylab="TRUE positive rate",
+      xlim=c(0, 1), ylim=c(0, 1),
+      main="ROC Curve for Hampel Classifier",
+      type="l", lwd=3, col="blue")
> abline(a=0.0, b=1.0, lwd=3, col="orange")
```

Improved Microstructure Noise Filtering

The filtering of microstructure noise can be improved by calculating the median prices over an interval of only 3 data points.

If the *z-score* exceeds the *threshold value* then it's classified as an *outlier* (jump in prices).

```
> # Calculate centered Hampel filter over 3 data points
> medianv <- roll::roll_median(taq$price, width=3)
> medianv[1:2] <- taq$price[1:2]
> medianv <- rutils::lagit(medianv, lagg=-1, pad_zeros=FALSE)
> madv <- HighFreq::roll_var(matrix(taq$price), look_back=3, method=
> madv <- rutils::lagit(madv, lagg=-1, pad_zeros=FALSE)
> # Calculate Z-scores
> zscores <- ifelse(madv > 0, (taq$price - medianv)/madv, 0)
> range(zscores); mad(zscores)
> madv <- mad(zscores[abs(zscores)>0])
> hist(zscores, breaks=2000, xlim=c(-5*madv, 5*madv))
```



```
> # Define discrimination threshold value
> threshold <- 6*madv
> bad_ticks <- (abs(zscores) > threshold)
> good_ticks <- taq[!bad_ticks]
> # Calculate number of price jumps
> sum(bad_ticks)/NROW(zscores)
> # Coerce trade prices to xts
> xtsv <- xts::xts(good_ticks[, .(price, volume)], good_ticks$index)
> colnames(xtsv) <- c("XLK.Close", "XLK.Volume")
> # Plot dygraph of the clean lots
> dygraphs::dygraph(xtsv$XLK.Close,
+   main="XLK Trade Ticks for 2020-03-16 (Hampel filtered)")
> # Plot the large lots
> x11(width=6, height=5)
> quantmod::chart_Series(x=xtsv$XLK.Close,
+   name="XLK Trade Ticks for 2020-03-16 (Hampel filtered)")
```

draft: Classification Using K-Nearest Neighbor (KNN) Algorithm

The K-nearest neighbor (KNN) algorithm is a supervised learning classification technique.

Normalizing numeric data

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

The method `predict.glm()` produces forecasts for a generalized linear model, in the form of probabilities for the `Boolean` response variable.

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

The *null hypothesis* is that `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.

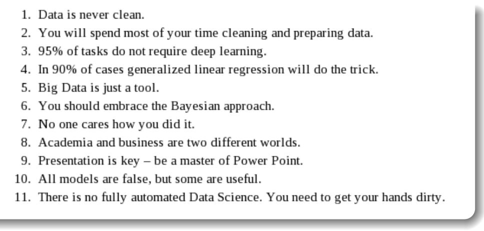
A *type I* error is the incorrect rejection of a `TRUE` *null hypothesis* (i.e. a "false positive"), when good data is classified as bad.

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.

draft: Data Science

Data Science is very important to quantitative finance.

Below is an example of a simulation of the path of *Brownian Motion* crossing a barrier level.

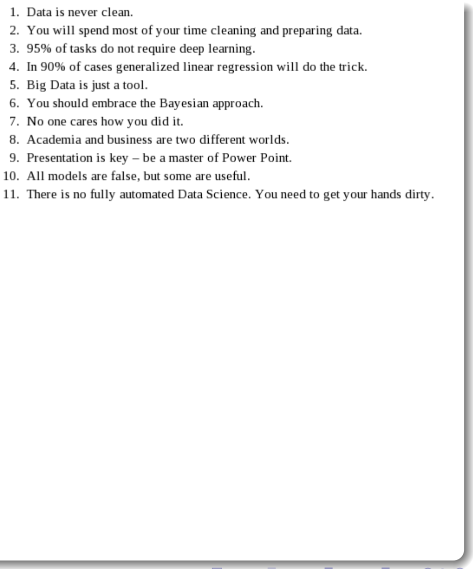


1. Data is never clean.
2. You will spend most of your time cleaning and preparing data.
3. 95% of tasks do not require deep learning.
4. In 90% of cases generalized linear regression will do the trick.
5. Big Data is just a tool.
6. You should embrace the Bayesian approach.
7. No one cares how you did it.
8. Academia and business are two different worlds.
9. Presentation is key – be a master of Power Point.
10. All models are false, but some are useful.
11. There is no fully automated Data Science. You need to get your hands dirty.

draft: Machine Learning

What is Machine Learning? What is Machine Learning? Machine Learning (ML) studies statistical models which can identify patterns in the data and make predictions. ML is closely related to statistics, but with an emphasis on prediction. ML models are divided into supervised learning or unsupervised learning.
Supervised learning models require a training set to calibrate the model parameters. Examples of supervised learning models are linear regression, decision trees, support vector machines (SVM), and neural networks.
Unsupervised learning models don't require a training set. Examples of unsupervised learning models are clustering models, like principal component analysis (PCA) and k-nearest neighbors (KNN). ML models are also divided into classification and regression models.
An example of a regression model is linear regression.
An example of a classification model is logistic regression. ML uses several techniques to calibrate models and improve prediction. First, ML uses cross-validation (backtesting) to determine the optimal model meta-parameters. Second, ML uses estimator shrinkage to achieve a better tradeoff between their bias and variance.
Data Science is very important to quantitative finance.

Below is an example of a simulation of the path of *Brownian Motion* crossing a barrier level.



1. Data is never clean.
2. You will spend most of your time cleaning and preparing data.
3. 95% of tasks do not require deep learning.
4. In 90% of cases generalized linear regression will do the trick.
5. Big Data is just a tool.
6. You should embrace the Bayesian approach.
7. No one cares how you did it.
8. Academia and business are two different worlds.
9. Presentation is key – be a master of Power Point.
10. All models are false, but some are useful.
11. There is no fully automated Data Science. You need to get your hands dirty.