FRE6871 R in Finance

Lecture#4, Spring 2026

Jerzy Pawlowski *jp3900@nyu.edu*

NYU Tandon School of Engineering

February 23, 2026

NYU | TANDON SCHOOL OF ENGINEERING

The *Logistic* Function

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

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

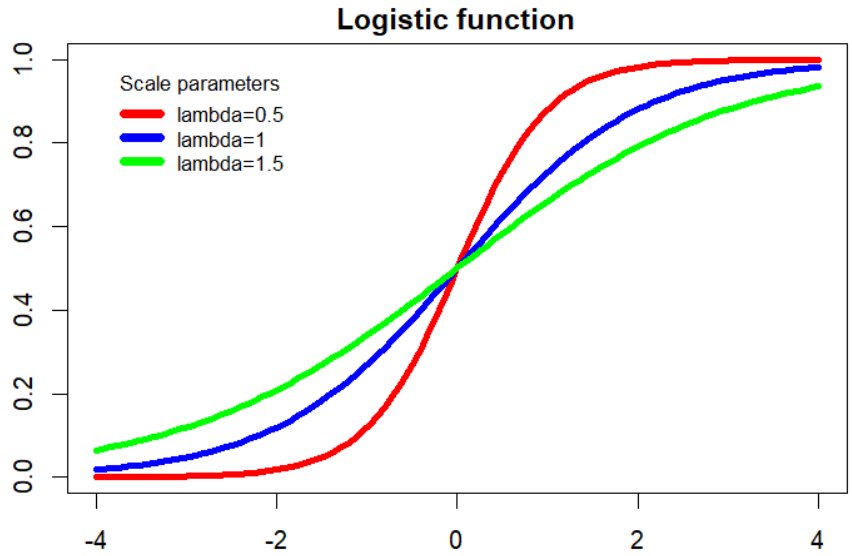
Where λ is the scale (dispersion) parameter.

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

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

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

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




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

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

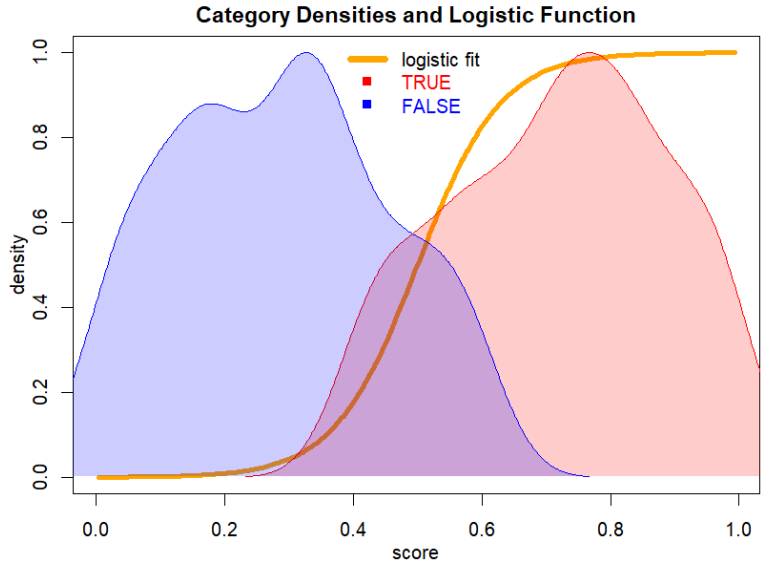
Logistic regression (*logit*) is used when the response are discrete variables (like `factors` or `integers`), when *linear* regression can't be applied.

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

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

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

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




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

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

The Likelihood Function of the Binomial Distribution

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

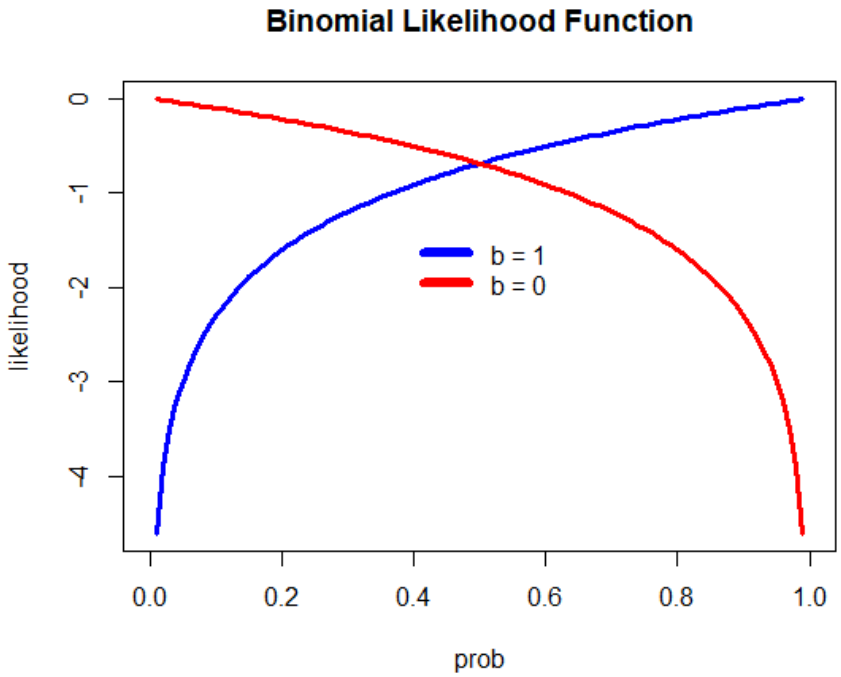
Then the response r follows the binomial distribution:

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

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

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

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




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

The Likelihood Function of the Logistic Model

Let r_i be binomial response variables, with probabilities p_i that depend on the predictor variables s_i through the logistic function:

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

Let's assume that the r_i response and s_i predictor 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*:

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

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

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

Multi-dimensional Optimization Using optim()

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

The argument fn is the objective function to be minimized.

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

The argument par is the initial vector argument value.

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

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

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

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

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

```
> # Rastrigin function with vector argument for optimization
> rastrigin <- function(vecv, param=25) {
+   sum(vecv^2 - param*cos(vecv))
+ }  ##  end rastrigin
> vecv <- c(pi/6, pi/6)
> rastrigin(vecv=vecv)
> # Draw 3d surface plot of Rastrigin function
> options(rgl.useNULL=TRUE); library(rgl)
> rgl::persp3d(
+   x=Vectorize(function(x, y) rastrigin(vecv=c(x, y))),
+   xlim=c(-10, 10), ylim=c(-10, 10),
+   col="green", axes=FALSE, zlab="", main="rastrigin")
> # Render the 3d surface plot of function
> rgl::rglwidget(elementId="plot3drgl", width=400, height=400)
> # Optimize with respect to vector argument
> optim1 <- optim(par=vecv, fn=rastrigin,
+         method="L-BFGS-B",
+         upper=c(4*pi, 4*pi),
+         lower=c(pi/2, pi/2),
+         param=1)
> # Optimal parameters and value
> optim1$par
> optim1$value
> rastrigin(optim1$par, param=1)
```

Maximum Likelihood Calibration of the Logistic Model

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

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

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

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

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

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

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

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

Package *ISLR* With Datasets for Machine Learning

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

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

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

The Default Dataset

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

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

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

This avoids implicit coercion by the function `glm()`.

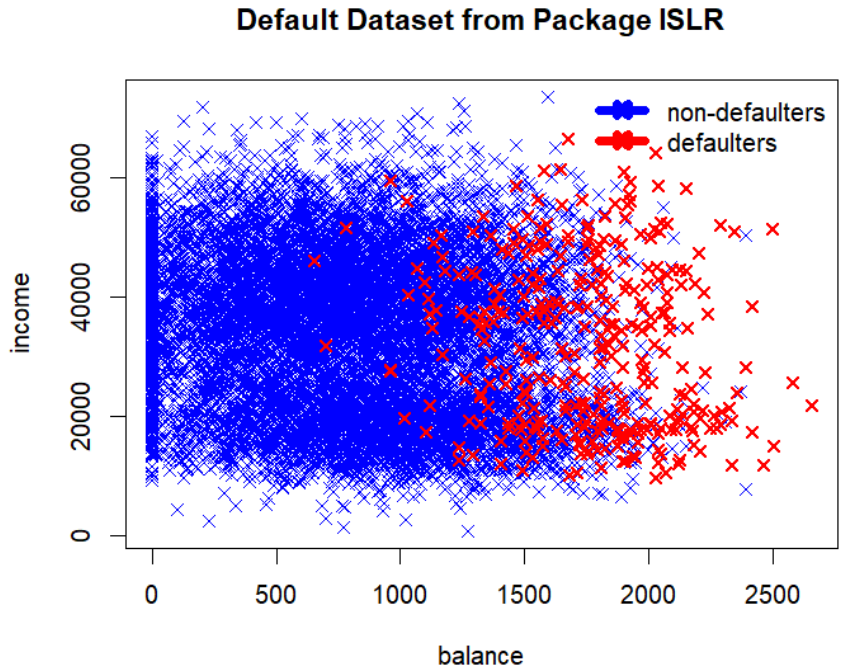
```
> # Coerce the default and student columns to Boolean
> Default <- ISLR::Default
> Default$default <- (Default$default == "Yes")
> Default$student <- (Default$student == "Yes")
> attach(Default)  ##  Attach Default to search path
> # Explore credit default data
> summary(Default)
  default      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

The Dependence of default on The balance and income

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

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

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




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

Boxplots of the Default Dataset

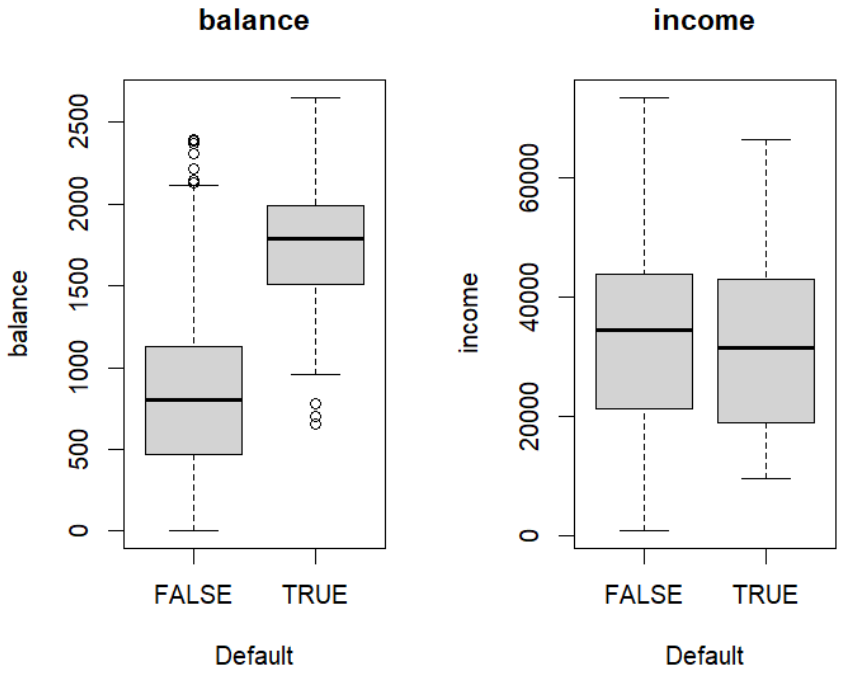
A *Box Plot* (box-and-whisker plot) is a graphical display of a distribution of data:

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])
> # Plot the densities of the balance amounts for defaulters and n
> plot(density(balance[default]),
+      main="Balance Amounts for Defaulters and Non-Defaulters",
+      xlab="balance", ylab="density", col="red", lwd=2)
> lines(density(balance[!default]), col="blue", lwd=2)
> legend(x="topright", inset=0.0, bty="n", lwd=6, y.intersp=0.4,
+  legend=c("non-defaulters", "defaulters"),
+  col=c("blue", "red"))
```

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

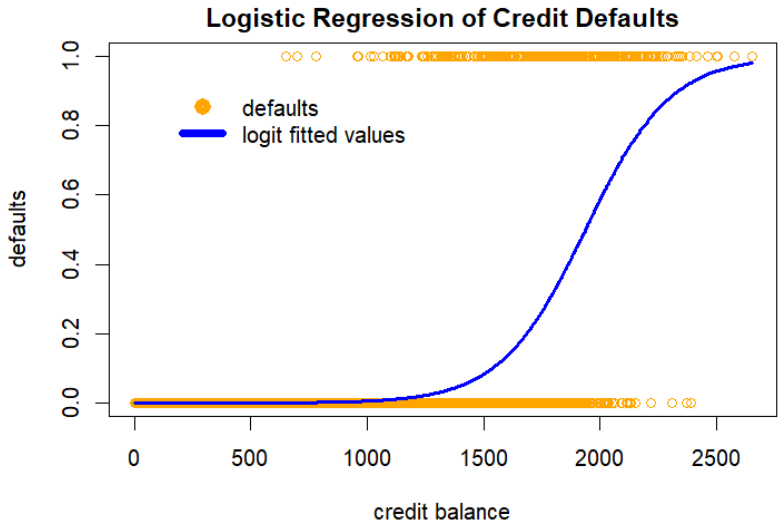
Modeling Credit Defaults Using *Logistic* Regression

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

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

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

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

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

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

(Dispersion parameter for binomial family taken to be 1)

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

Number of Fisher Scoring iterations: 8
```



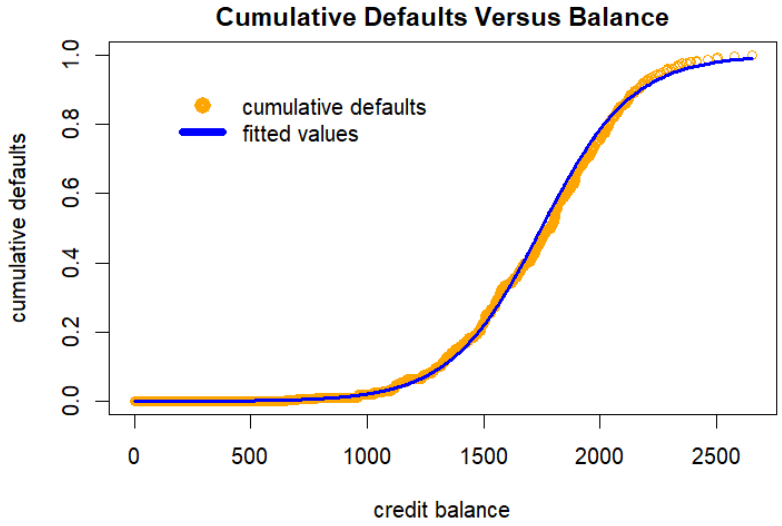

```
> x11(width=6, height=5)
> par(mar=c(4, 4, 2, 2), oma=c(0, 0, 0, 0), mgp=c(2.5, 1, 0))
> plot(x=balance, y=default,
+      main="Logistic Regression of Credit Defaults",
+      col="orange", xlab="credit balance", ylab="defaults")
> ordern <- order(balance)
> lines(x=balance[ordern], y=logmod$fitted.values[ordern],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

Modeling Cumulative Defaults Using *Logistic* Regression

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

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

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




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

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

Multifactor *Logistic* Regression

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

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

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

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

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

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

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

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

(Dispersion parameter for binomial family taken to be 1)

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

Number of Fisher Scoring iterations: 8
```

Modeling Credit Defaults Using Student Status

The `student` column alone can be used to calculate the probability of default using a 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.

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

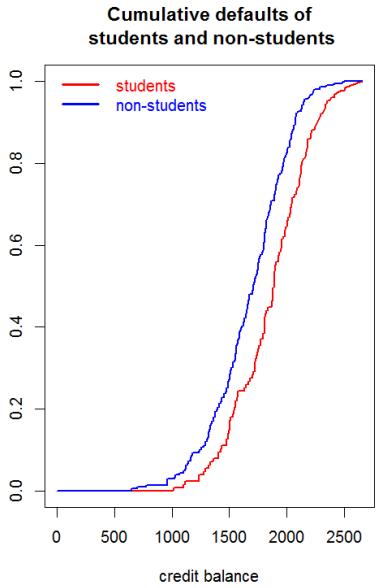
```
> # Fit single-factor logistic model with student as predictor
> studentmod <- glm(default ~ student, family=binomial(logit))
> summary(studentmod)
> # Multifactor coefficient is negative
> logmod$coefficients
> # Single-factor coefficient is positive
> studentmod$coefficients
```

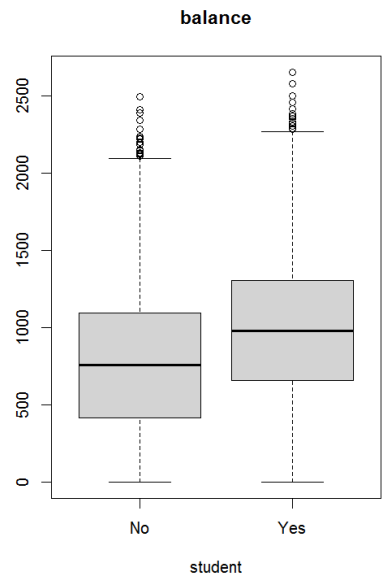
Confounding Variables in Multifactor *Logistic* Regression

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

The plots and boxplots show that students have higher credit balances than non-students.

```
> # Calculate the cumulative defaults
> defcum <- sapply(balance, function(balv) {
+ c(student=sum(default[student & (balance <= balv)]),
+   non_student=sum(default[!student & (balance <= balv)]))
+ })  ##  end sapply
> deftotal <- c(student=sum(student & default),
+       student=sum(!student & default))
> defcum <- t(defcum / deftotal)
> # Plot cumulative defaults
> par(mfrow=c(1,2))  ##  Set plot panels
> ordern <- order(balance)
> plot(x=balance[ordern], y=defcum[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=defcum[ordern, 2], col="blue", lwd=2)
> legend(x="topleft", bty="n", y.intersp=0.4,
+  legend=c("students", "non-students"),
+  col=c("red", "blue"), text.col=c("red", "blue"), lwd=3)
> # Balance boxplot for student factor
> boxplot(formula=balance ~ student,
+   col="lightgrey", main="balance", xlab="Student")
```





Forecasting Credit Defaults using Logistic Regression

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

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

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

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

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

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

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

Forecasting Errors

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

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

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

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

A *type I* error is the incorrect rejection of a `TRUE` *null hypothesis* (i.e. a "false positive"), when there is no default but it's classified as a default.

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

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

The Confusion Matrix of a Binary Classification Model

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

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

```
> # Calculate the FALSE positive and FALSE negative rates
> confmat <- confmat / rowSums(confmat)
> c(typeI=confmat[2, 1], typeII=confmat[1, 2])
  typeI  typeII
0.00186 0.81988
> detach(Default)
```

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

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

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

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

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

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

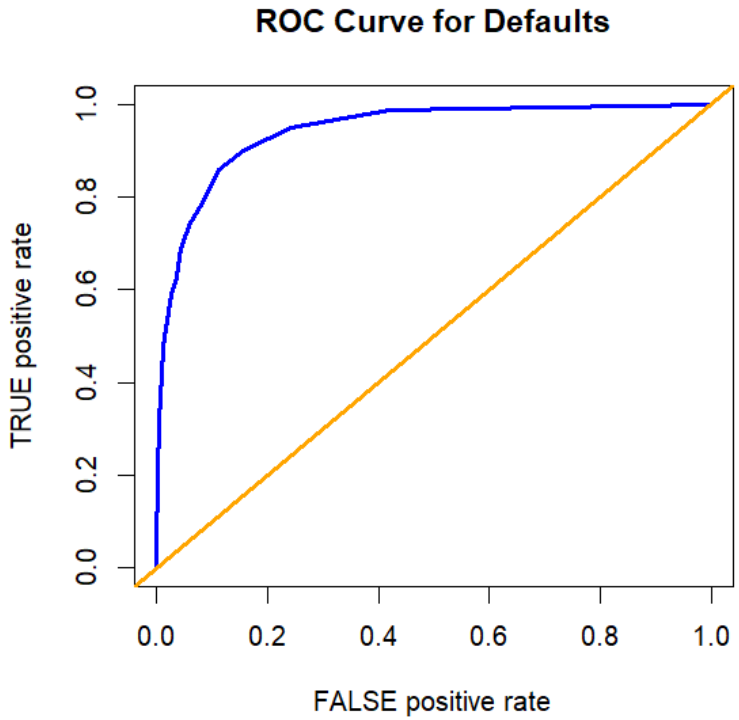
The sum of the *true negative* plus the *false positive* rate is equal to 1.

Receiver Operating Characteristic (ROC) Curve

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

The area under the *ROC curve* (AUC) is a measure of the performance of a binary classification model.

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




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

Types of Bad Data

Possible sources of bad data are: imported data, class coercion, numeric overflow.

Types of bad data:

- `NA` (not available) is a logical constant indicating missing data,
- `NaN` means Not a Number data,
- `Inf` means numeric overflow - divide by zero,

When a function produces `NA` or `NaN` values, then it also produces a *warning* condition, but not an *error*.

`NA` or `NaN` values are not *errors*.

The functions `is.na()` and `is.nan()` test for `NA` and `NaN` values.

Many functions have a `na.rm` parameter to remove `NA`s from input data.

```
> as.numeric(c(1:3, "a"))  ##  NA from coercion
[1]  1  2  3 NA
> 0/0  ##  NaN from ambiguous math
[1] NaN
> 1/0  ##  Inf from divide by zero
[1] Inf
> is.na(c(NA, NaN, 0/0, 1/0))  ##  Test for NA
[1]  TRUE  TRUE  TRUE FALSE
> is.nan(c(NA, NaN, 0/0, 1/0))  ##  Test for NaN
[1] FALSE  TRUE  TRUE FALSE
> NA*1:4  ##  Create vector of Nas
[1] NA NA NA NA
> # Create vector with some NA values
> datav <- c(1, 2, NA, 4, NA, 5)
> datav
[1]  1  2 NA  4 NA  5
> mean(datav)  ##  Returns NA, when NAs are input
[1] NA
> mean(datav, na.rm=TRUE)  ##  remove NAs from input data
[1] 3
> datav[!is.na(datav)]  ##  Delete the NA values
[1] 1 2 4 5
> sum(!is.na(datav))  ##  Count non-NA values
[1] 4
```

Scrubbing Bad Data

The function `complete.cases()` returns `TRUE` if a row has no `NA` values.

```
> # airquality data has some NAs
> head(airquality)
  Ozone Solar.R Wind Temp Month Day
1    41     190  7.4   67     5   1
2    36     118  8.0   72     5   2
3    12     149 12.6   74     5   3
4    18     313 11.5   62     5   4
5    NA      NA 14.3   56     5   5
6    28      NA 14.9   66     5   6
> dim(airquality)
[1] 153   6
> # Number of NA elements
> sum(is.na(airquality))
[1] 44
> # Number of rows with NA elements
> sum(!complete.cases(airquality))
[1] 42
> # Display rows containing NAs
> head(airquality[!complete.cases(airquality), ])
   Ozone Solar.R Wind Temp Month Day
5     NA      NA 14.3   56     5   5
6     28      NA 14.9   66     5   6
10    NA     194  8.6   69     5  10
11     7      NA  6.9   74     5  11
25    NA      66 16.6   57     5  25
26    NA     266 14.9   58     5  26
```

Scrubbing Data Using Carry Forward

Rows containing bad data may be either removed or replaced with an estimated value.

The function `stats::na.omit()` removes individual `NA` values from vectors, and it also removes whole rows of data containing `NA` values from matrices and data frames.

Bad data can also be replaced with the most recent prior values (carry forward good data).

The function `zoo::na.locf()` replaces `NA` values with the most recent non-`NA` values prior to it (*locf* stands for *last observation carry forward*).

Copying the last non-`NA` values forward causes less data loss than removing whole rows of data.

The function `na.locf()` with argument `fromLast=TRUE` replaces `NA` values with non-`NA` values in reverse order, starting from the end.

```
> # Create vector containing NA values
> vecv <- sample(22)
> vecv[sample(NROW(vecv), 4)] <- NA
> # Replace NA values with the most recent non-NA values
> zoo::na.locf(vecv)
 [1]  7  7  7  6  6 18 15  3  9 16 16 12  5 17  8 21 13  1 22 20  2
> # Remove rows containing NAs
> goodair <- airquality[complete.cases(airquality), ]
> dim(goodair)
[1] 111   6
> # NAs removed
> head(goodair)
  Ozone Solar.R Wind Temp Month Day
1    41     190  7.4   67     5   1
2    36     118  8.0   72     5   2
3    12     149 12.6   74     5   3
4    18     313 11.5   62     5   4
7    23     299  8.6   65     5   7
8    19      99 13.8   59     5   8
> # Another way of removing NAs
> freshair <- na.omit(airquality)
> all.equal(freshair, goodair, check.attributes=FALSE)
[1] TRUE
> # Replace NAs
> goodair <- zoo::na.locf(airquality)
> dim(goodair)
[1] 153   6
> # NAs replaced
> head(goodair)
  Ozone Solar.R Wind Temp Month Day
1    41     190  7.4   67     5   1
2    36     118  8.0   72     5   2
3    12     149 12.6   74     5   3
4    18     313 11.5   62     5   4
5    18     313 14.3   56     5   5
6    28     313 14.9   66     5   6
```

Scrubbing Time Series Data

Missing asset prices and returns can be replaced with the most recent prior values (carry forward good data).

But missing asset returns should not be replaced with values from the future. Instead, missing returns should be replaced with zero values.

The function `na.locf.xts()` from package *xts* is faster than `zoo::na.locf()`, but it only operates on time series of class `"xts"`.

```
> # Replace NAs in xts time series
> library(rutils)  ##  load package rutils
> pricev <- rutils::etfenv$prices[, 1]
> head(pricev, 3)
            XLP
1993-01-29  NA
1993-02-01  NA
1993-02-02  NA
> sum(is.na(pricev))
[1] 1490
> pricez <- zoo::na.locf(pricev, fromLast=TRUE)
> pricex <- xts:::na.locf.xts(pricev, fromLast=TRUE)
> all.equal(pricez, pricex, check.attributes=FALSE)
[1] TRUE
> head(pricex, 3)
             XLP
1993-01-29 14.1
1993-02-01 14.1
1993-02-02 14.1
> library(microbenchmark)
> summary(microbenchmark(
+   zoo=zoo::na.locf(pricev, fromLast=TRUE),
+   xts=xts:::na.locf.xts(pricev, fromLast=TRUE),
+   times=10))[, c(1, 4, 5)]  ##  end microbenchmark summary
  expr mean median
1  zoo 30.2   28.4
2  xts 29.1   27.6
```

NULL Values

NULL represents a null object, and is a legitimate value, not bad data.

NULL is often returned by functions whose value is undefined.

NULL can also be used to initialize vectors.

NULL is not the same as NA values or zero-length (empty) vectors.

The functions `numeric()` and `character()` return empty (zero-length) vectors of the specified *type*.

The function `is.null()` tests for NULL values.

Very often variables are initialized to NULL before the start of iteration.

A more efficient way to perform iteration is by pre-allocating the vector.

```
> # NULL values have no mode or type
> c(mode(NULL), mode(NA))
[1] "NULL"    "logical"
> c(typeof(NULL), typeof(NA))
[1] "NULL"    "logical"
> c(NROW(NULL), NROW(NA))
[1] 0 1
> # Check for NULL values
> is.null(NULL)
[1] TRUE
> # NULL values are ignored when combined into a vector
> c(1, 2, NULL, 4, 5)
[1] 1 2 4 5
> # But NA value isn't ignored
> c(1, 2, NA, 4, 5)
[1]  1  2 NA  4  5
> # Vectors can be initialized to NULL
> vecv <- NULL
> is.null(vecv)
[1] TRUE
> # Grow the vector in a loop - very bad code!!!
> for (indeks in 1:5)
+   vecv <- c(vecv, indeks)
> # Initialize empty vector
> vecv <- numeric()
> # Grow the vector in a loop - very bad code!!!
> for (indeks in 1:5)
+   vecv <- c(vecv, indeks)
> # Allocate vector
> vecv <- numeric(5)
> # Assign to vector in a loop - good code
> for (indeks in 1:5)
+   vecv[indeks] <- runif(1)
```

Scrubbing Isolated Spikes In Stock Prices

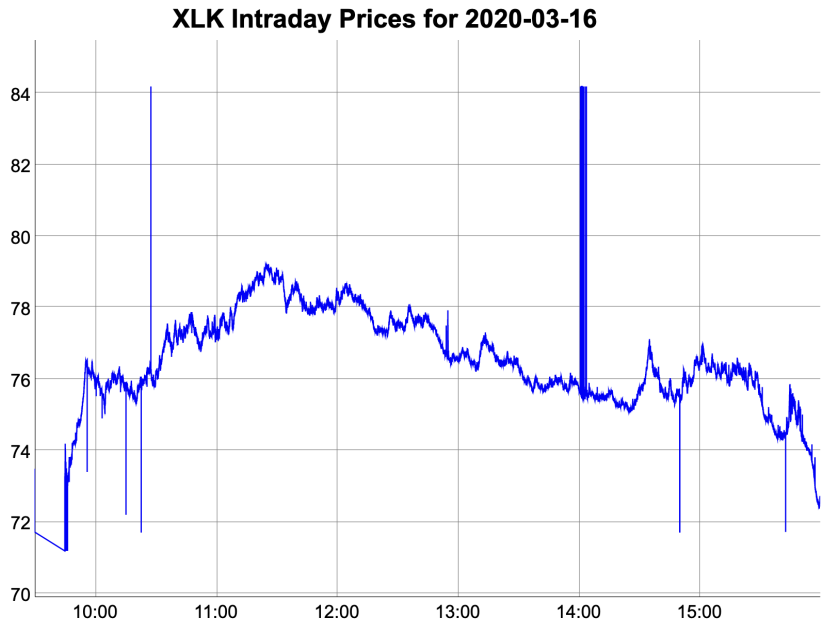
Isolated price spikes in historical (offline) prices can be identified using a *three-point filter* (tri-filter).

The centered *z-score* is equal to the price minus the mean of the neighboring prices, divided by their differences:

$$z_i = \frac{p_i - 0.5(p_{i-1} + p_{i+1})}{p_{i-1} - p_{i+1}}$$

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

The *tri-filter* can remove isolated price spikes, but not consecutive price spikes.




```
> # Load and plot intraday stock prices
> load("/Users/jerzy/Develop/lecture_slides/data/xlk_tick_trades_2(
> pricev <- xlk$price
> dygraphs::dygraph(pricev, main="XLK Intraday Prices for 2020-03-1
+   dyOptions(colors="blue", strokeWidth=1)
> # Calculate the lagged and advanced prices
> pricelag <- rutils::lagit(pricev)
> pricelag[1] <- pricelag[2]
> pricadv <- rutils::lagit(pricev, lagg=-1)
> pricadv[NROW(pricadv)] <- pricadv[NROW(pricadv)-1]
> # Calculate the z-scores
> diffl <- ifelse(abs(pricelag-pricadv) < 0.01, 0.01, abs(pricelag-
> zscores <- (pricev - 0.5*(pricelag+pricadv))/diffl
```

```
> # Z-scores have very fat tails
> range(zscores); mad(zscores)
> madz <- mad(zscores[abs(zscores) > 0])
> hist(zscores, breaks=5000, xlim=c(-2*madz, 2*madz))
> # Scrub the price spikes
> threshv <- 5*madz ##  Discrimination threshold
> indeks <- which(abs(zscores) > threshv)
> pricev[indeks] <- as.numeric(pricev[indeks-1])
> # Plot dygraph of the scrubbed prices
> dygraphs::dygraph(pricev, main="Scrubbed XLK Intraday Prices") %>%
+   dyOptions(colors="blue", strokeWidth=1)
```

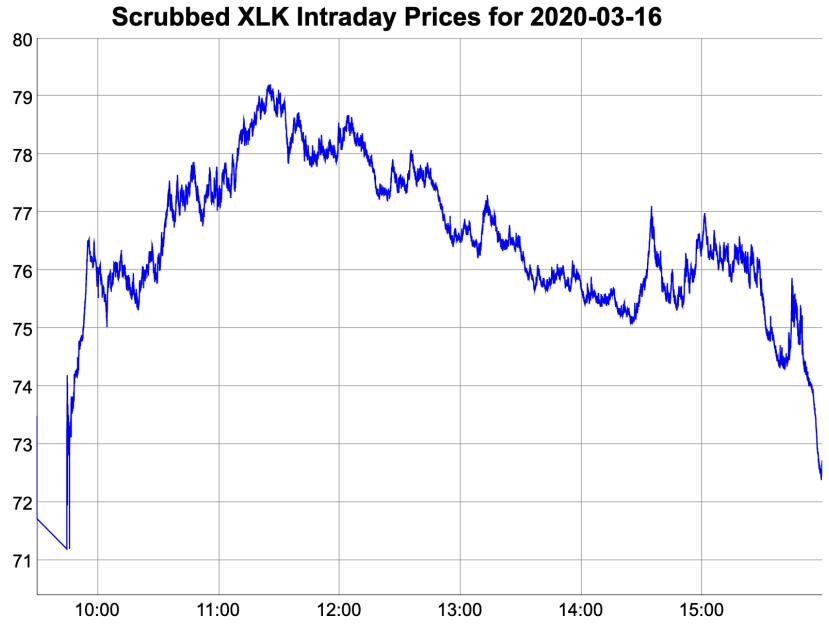
The Hampel Filter For Filtering Price Spikes

The price spikes in prices can be identified more effectively using a *Hampel filter*.

The *z-score* is equal to the price minus the median of the neighboring prices, divided by the median absolute deviation (*MAD*) of prices:

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

The *median* and *MAD* values are more robust to price spikes than the mean and standard deviation, so the *Hampel filter* can remove both isolated and consecutive price spikes.




```
> # Calculate the centered Hampel filter to remove bad prices
> lookb <- 71 ##  Look-back interval
> halfb <- lookb %/% 2 ##  Half-back interval
> pricev <- xlk$price
> # Calculate the trailing median and MAD
> medianv <- HighFreq::roll_mean(pricev, lookb=lookb, method="nonpa
> colnames(medianv) <- c("median")
> madv <- HighFreq::roll_var(pricev, lookb=lookb, method="nonparame
> # madv <- TTR::runMAD(pricev, n=lookb)
> # Center the median and the MAD
> medianv <- rutils::lagit(medianv, lagg=(-halfb), pad_zeros=FALSE)
> madv <- rutils::lagit(madv, lagg=(-halfb), pad_zeros=FALSE)
> # Calculate the Z-scores
> zscores <- ifelse(madv > 0, (pricev - medianv)/madv, 0)
> # Z-scores have very fat tails
> range(zscores); mad(zscores)
> madz <- mad(zscores[abs(zscores) > 0])
> hist(zscores, breaks=5000, xlim=c(-2*madz, 2*madz))
```

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

Classifying Data Outliers Using the Hampel Filter

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

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

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

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

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

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

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

```
> # Add 200 random price spikes to the clean prices
> set.seed(1121, "Mersenne-Twister", sample.kind="Rejection")
> nspikes <- 200
> nrows <- NROW(priceg)
> ispike <- logical(nrows)
> ispike[sample(x=nrows, size=nspikes)] <- TRUE
> priceb <- priceg
> priceb[ispike] <- priceb[ispike]*
+   sample(c(0.999, 1.001), size=nspikes, replace=TRUE)
> # Plot the bad prices and their medians
> medianv <- HighFreq::roll_mean(priceb, lookb=lookb, method="nonpar
> pricem <- cbind(priceb, medianv)
> colnames(pricem) <- c("prices with spikes", "median")
> dygraphs::dygraph(pricem, main="XLK Prices With Spikes") %>%
+   dyOptions(colors=c("red", "blue"))
> # Calculate the z-scores
> madv <- HighFreq::roll_var(priceb, lookb=lookb, method="nonparamet
> zscores <- ifelse(madv > 0, (priceb - medianv)/madv, 0)
> # Z-scores have very fat tails
> range(zscores); mad(zscores)
> madz <- mad(zscores[abs(zscores) > 0])
> hist(zscores, breaks=10000, xlim=c(-4*madz, 4*madz))
> # Identify good prices with small z-scores
> threshv <- 3*madz
> isgood <- (abs(zscores) < threshv)
> # Calculate the number of bad prices
> sum(!isgood)
```

Confusion Matrix of a Binary Classification Model

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

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

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

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

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

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

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

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

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

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

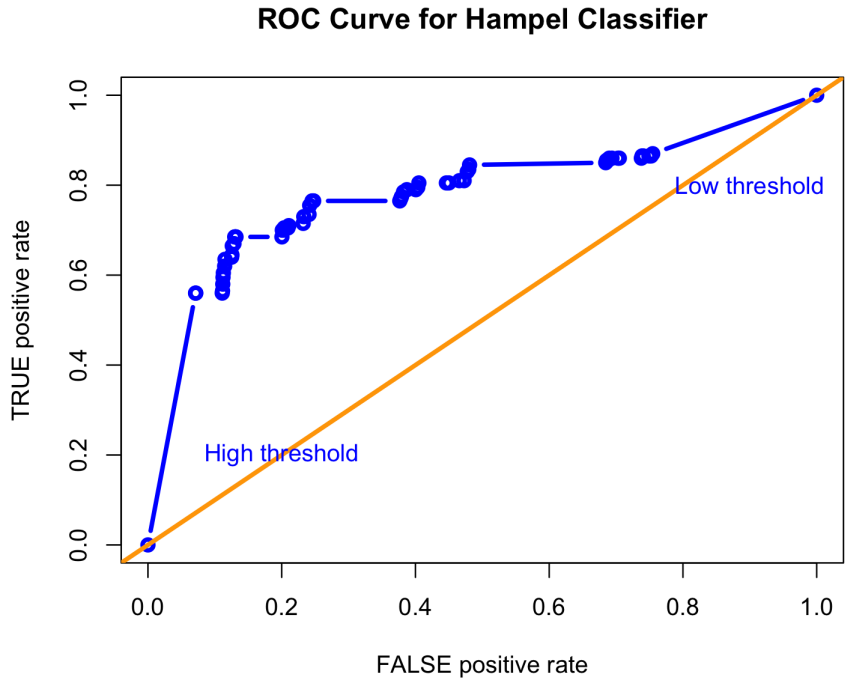
The sum of the *true negative* plus the *false positive* rate is equal to 1.

Receiver Operating Characteristic (ROC) Curve

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

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

The *false positive* and *true positive* rates are higher for lower discrimination threshold values, and lower for higher threshold values.




```
> # Confusion matrix as function of threshold
> confun <- function(actualv, zscores, threshv) {
+     confmat <- table(actualv, (abs(zscores) < threshv))
+     confmat <- confmat / rowSums(confmat)
+     c(typeI=confmat[2, 1], typeII=confmat[1, 2])
+ }  ##  end confun
> confun(!ispike, zscores, threshv=threshv)
> # Define vector of discrimination thresholds
> threshv <- madz*seq(from=0.1, to=3.0, by=0.05)/2
> # Calculate the error rates
> errorr <- sapply(threshv, confun, actualv=!ispike, zscores=zscor
> errorr <- t(errorr)
> rownames(errorr) <- threshv
> errorr <- rbind(c(1, 0), errorr)
> errorr <- rbind(errorr, c(0, 1))
> # Calculate the area under the ROC curve (AUC)
> truepos <- (1 - errorr[, "typeII"])
> truepos <- (truepos + rutils::lagit(truepos))/2
> falsepos <- rutils::diffit(errorr[, "typeI"])
> abs(sum(truepos*falsepos))
```

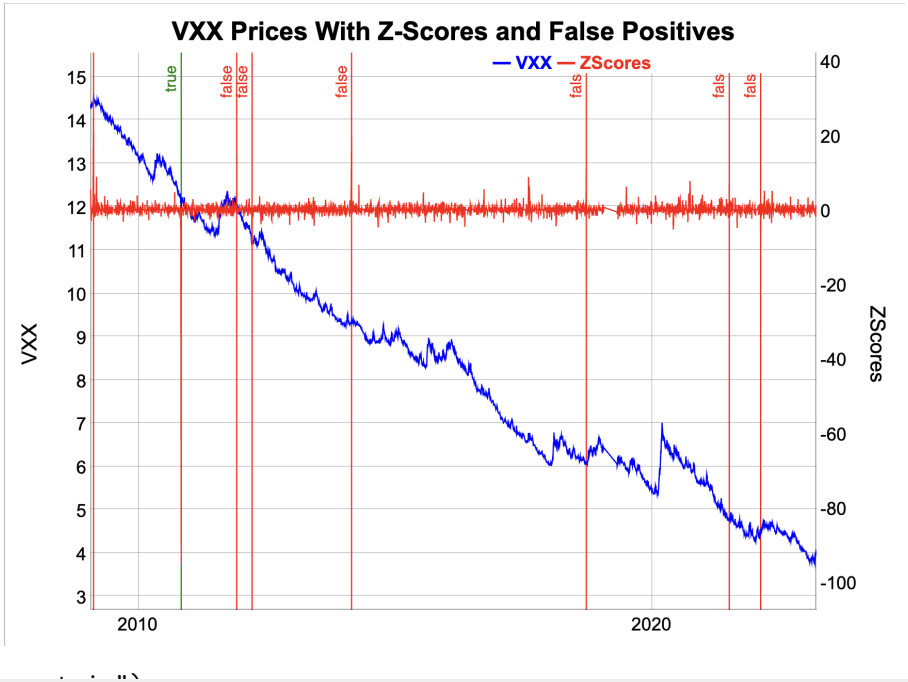
```
> # Plot ROC curve for Hampel classifier
> plot(x=errorr[, "typeI"], y=1-errorr[, "typeII"],
+      xlab="FALSE positive rate", ylab="TRUE positive rate",
+      xlim=c(0, 1), ylim=c(0, 1),
+      main="ROC Curve for Hampel Classifier",
+      type="b", lwd=3, col="blue")
> # Add diagonal line for random classifier
> abline(a=0.0, b=1.0, lwd=3, col="orange")
> # Add text with threshold values
> text(x=0.2, y=0.2, "High threshold", cex=1.0, col="blue")
> text(x=0.9, y=0.8, "Low threshold", cex=1.0, col="blue")
```

Filtering Bad Data From Daily Stock Prices

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

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

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




```
> # Load log VXX prices
> load("/Users/jerzy/Develop/lecture_slides/data/pricevxx.RData")
> nrows <- NROW(pricev)
> # Calculate the centered Hampel filter for VXX
> lookb <- 7 ##  Look-back interval
> halfb <- lookb %/% 2 ##  Half-back interval
> medianv <- HighFreq::roll_mean(pricev, lookb=lookb, method="nonpa
> medianv <- rutils::lagit(medianv, lagg=(-halfb), pad_zeros=FALSE)
> madv <- HighFreq::roll_var(pricev, lookb=lookb, method="nonparame
> madv <- rutils::lagit(madv, lagg=(-halfb), pad_zeros=FALSE)
> zscores <- ifelse(madv > 0, (pricev - medianv)/madv, 0)
> range(zscores); mad(zscores)
> madz <- mad(zscores[abs(zscores) > 0])
> hist(zscores, breaks=100, xlim=c(-3*madz, 3*madz))
> # Define discrimination threshold value
> threshv <- 9*madz
> # Calculate the good prices
> isgood <- (abs(zscores) < threshv)
> sum(!isgood)
> # Dates of the bad prices
> zoo::index(pricev[!isgood])
```

```
> # Calculate the false positives
> falsep <- !isgood
> falsep[which(zoo::index(pricev) == as.Date("2010-11-08"))] <- FALS
> # Plot dygraph of the prices with bad prices
> datam <- cbind(pricev, zscores)
> colnames(datam)[2] <- "ZScores"
> colv <- colnames(datam)
> dygraphs::dygraph(datam, main="VXX Prices With Z-Scores and False
+   dyAxis("y", label=colv[1], independentTicks=TRUE) %>%
+   dyAxis("y2", label=colv[2], independentTicks=TRUE) %>%
+   dySeries(name=colv[1], axis="y", strokeWidth=1, col="blue") %>%
+   dySeries(name=colv[2], axis="y2", strokeWidth=1, col="red") %>%
+   dyEvent(zoo::index(pricev[falsep]), label=rep("false", sum(false
+   dyEvent(zoo::index(pricev["2010-11-08"]), label="true", strokePa
```

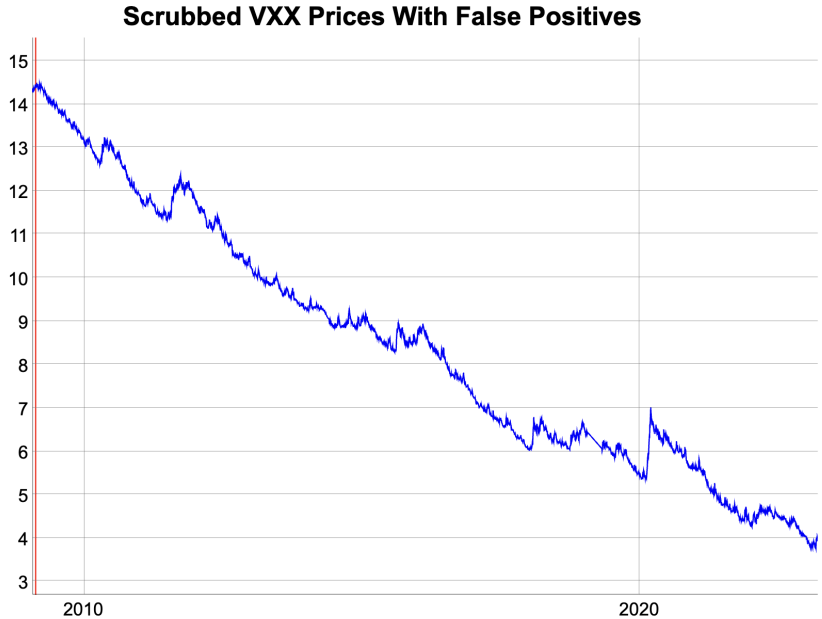
Scrubbing Bad Stock Prices

Bad stock prices can be scrubbed (replaced) with the previous good price using the function `zoo::na.locf()` from the package zoo.

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

Centered *z-scores* are calculated using past and future prices, so they can only be used to scrub historical (offline) prices.

But real time streaming data (online) can only be scrubbed using a one-sided (trailing) filters, which is less effective in identifying bad prices.




```
> # Replace bad stock prices with the previous good prices
> priceg <- pricev
> priceg[!isgood] <- NA
> priceg <- zoo::na.locf(priceg)
> # Calculate the Z-scores
> medianv <- HighFreq::roll_mean(priceg, lookb=lookb, method="nonpa
> medianv <- rutils::lagit(medianv, lagg=(-halfb), pad_zeros=FALSE)
> madv <- HighFreq::roll_var(priceg, lookb=lookb, method="nonparame
> madv <- rutils::lagit(madv, lagg=(-halfb), pad_zeros=FALSE)
> zscores <- ifelse(madv > 0, (priceg - medianv)/madv, 0)
> madz <- mad(zscores[abs(zscores) > 0])
> # Calculate the number of bad prices
> threshv <- 9*madz
> isgood <- (abs(zscores) < threshv)
> sum(!isgood)
> zoo::index(priceg[!isgood])
```

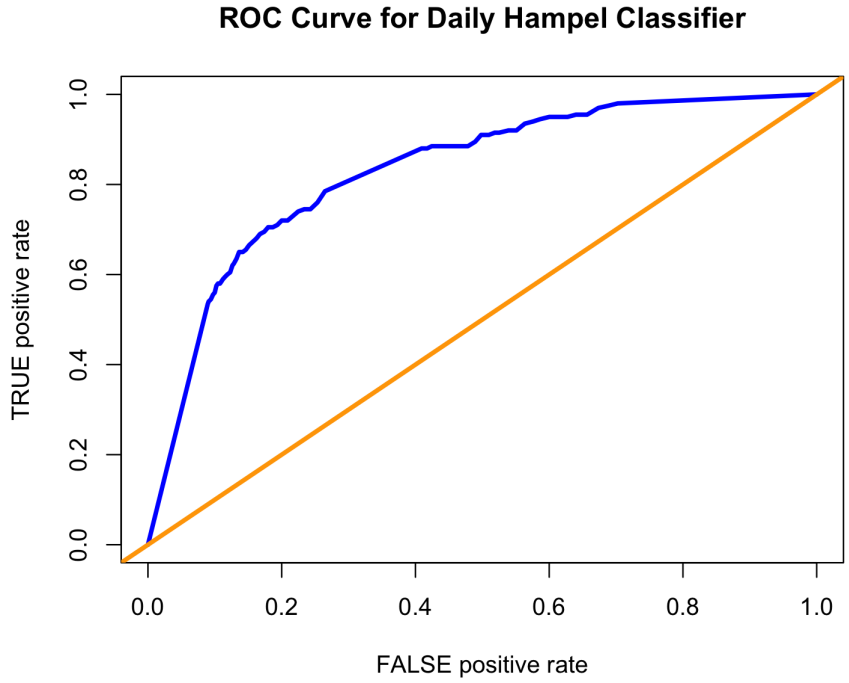
```
> # Calculate the false positives
> falsep <- !isgood
> falsep[which(zoo::index(pricev) == as.Date("2010-11-08"))] <- FALS
> # Plot dygraph of the prices with bad prices
> dygraphs::dygraph(priceg, main="Scrubbed VXX Prices With False Pos
+   dyEvent(zoo::index(priceg[falsep]), label=rep("false",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

ROC Curve for Daily Hampel Classifier

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

The performance of the Hampel noise classification model depends on the length of the look-back time interval.

The optimal *look-back interval* and *threshold value* can be determined using *cross-validation*.




```
> # Add 200 random price spikes to the clean prices
> set.seed(1121, "Mersenne-Twister", sample.kind="Rejection")
> nspikes <- 200
> ispike <- logical(nrows)
> ispike[sample(x=nrows, size=nspikes)] <- TRUE
> priceb <- priceg
> priceb[ispike] <- priceb[ispike]*
+   sample(c(0.99, 1.01), size=nspikes, replace=TRUE)
> # Calculate the Z-scores
> medianv <- HighFreq::roll_mean(priceb, lookb=lookb, method="nonparametric")
> medianv <- rutils::lagit(medianv, lagg=(-halfb), pad_zeros=FALSE)
> madv <- HighFreq::roll_var(priceb, lookb=lookb, method="nonparame
> madv <- rutils::lagit(madv, lagg=(-halfb), pad_zeros=FALSE)
> zscores <- ifelse(madv > 0, (priceb - medianv)/madv, 0)
> madz <- mad(zscores[abs(zscores) > 0])
> # Define vector of discrimination thresholds
> threshv <- madz*seq(from=0.1, to=3.0, by=0.05)/2
> # Calculate the error rates
> errorr <- sapply(threshv, confun, actualv=!ispike, zscores=zscor
> errorr <- t(errorr)
> rownames(errorr) <- threshv
> errorr <- rbind(c(1, 0), errorr)
> errorr <- rbind(errorr, c(0, 1))
```

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

Homework Assignment

Required

- Study all the lecture slides in *FRE6871_Lecture_4.pdf*, and run all the code in *FRE6871_Lecture_4.R*