FRE6871 R in Finance

Lecture#5, Spring 2025

Jerzy Pawlowski *jp3900@nyu.edu*

NYU Tandon School of Engineering

April 21, 2025

NYU | TANDON SCHOOL OF ENGINEERING

Lists

Lists are a type of vector that contain elements of different *types*.

Lists are recursive object types, meaning each list element can contain other vectors or lists.

The function `list()` creates a list from a list of vectors.

`list()` creates a named list from a list of symbol-value pairs.

The function `is.list()` returns `TRUE` if its argument is a list, and `FALSE` otherwise.

The function `unlist()` collapses a list with atomic elements into a vector (which can cause type coercion).

```
> # Create a list with two elements
> listv <- list(c("a", "b"), 1:4)
> listv
[[1]]
[1] "a" "b"

[[2]]
[1] 1 2 3 4
> c(typeof(listv), mode(listv), class(listv))
[1] "list" "list" "list"
> # Lists are also vectors
> c(is.vector(listv), is.list(listv))
[1] TRUE TRUE
> NROW(listv)
[1] 2
> # Create named list
> listv <- list(first=c("a", "b"), second=1:4)
> listv
$first
[1] "a" "b"

$second
[1] 1 2 3 4
> names(listv)
[1] "first"  "second"
> unlist(listv)
 first1  first2 second1 second2 second3 second4
    "a"     "b"     "1"     "2"     "3"     "4"
```

Subsetting Lists

Lists can be subset (indexed) using:

- the "[" operator (returns sublist),
- the "[[" operator (returns an element),
- the "$" operator (for named listv only),

Partial name matching allows subsetting with partial name, as long as it can be resolved.

```
> listv[2]  # Extract second element as sublist
$second
[1] 1 2 3 4
> listv[[2]]  # Extract second element
[1] 1 2 3 4
> listv[[2]][3]  # Extract third element of second element
[1] 3
> listv[[c(2, 3)]]  # Third element of second element
[1] 3
> listv$second  # Extract second element
[1] 1 2 3 4
> listv$s  # Extract second element - partial name matching
[1] 1 2 3 4
> listv$second[3]  # Third element of second element
[1] 3
> listv <- list()  # Empty list
> listv$a <- 1
> listv[2] <- 2
> listv
$a
[1] 1

[[2]]
[1] 2
> names(listv)
[1] "a" ""
```

Coercing Vectors Into Lists Using `as.list()`

The function `as.list()` coerces vectors and other objects into lists.

`as.list()` returns a list with the same elements as the vector.

`list()` called on a vector returns a single element equal to the vector.

```
> # Convert vector elements to list elements
> as.list(1:3)
[[1]]
[1] 1

[[2]]
[1] 2

[[3]]
[1] 3
> # Convert whole vector to single list element
> list(1:3)
[[1]]
[1] 1 2 3
```

Data Frames

Data frames are 2-D objects (like matrices), but their columns can be of different *types*.

Data frames can be thought of as listv of vectors of the same length.

The function `data.frame()` creates a *data frame* from vectors assigned to column names.

```
> dframe <- data.frame(  # Create a data frame
+                 type=c("rose", "daisy", "tulip"),
+                 color=c("red", "white", "yellow"),
+                 price=c(1.5, 0.5, 1.0)
+               )  # end data.frame
> dframe
   type  color price
1  rose    red   1.5
2 daisy  white   0.5
3 tulip yellow   1.0
> dim(dframe)  # Get dimension attribute
[1] 3 3
> colnames(dframe)  # Get the colnames attribute
[1] "type"  "color" "price"
> rownames(dframe)  # Get the rownames attribute
[1] "1" "2" "3"
> class(dframe)  # Get object class
[1] "data.frame"
> typeof(dframe)  # Data frames are listv
[1] "list"
> is.data.frame(dframe)
[1] TRUE
>
> class(dframe$type)  # Get column class
[1] "character"
> class(dframe$price)  # Get column class
[1] "numeric"
```

Subsetting Data Frames

Data frames can be subset in a similar way to listv and matrices.

Depending on how a data frame is subset, the result can be either a data frame or a vector.

Extracting a single column from a data frame produces a vector.

The data frame class attribute can be preserved by using the parameter `"drop=FALSE"`.

Extracting a single row from a data frame produces a data frame.

The function `unlist()` applied to a single row extracted from a data frame coerces it to a vector.

```
> dframe[, 3]  # Extract third column as vector
[1] 1.5 0.5 1.0
> dframe[[3]]  # Extract third column as vector
[1] 1.5 0.5 1.0
> dframe[3]  # Extract third column as data frame
  price
1   1.5
2   0.5
3   1.0
> dframe[, 3, drop=FALSE]  # Extract third column as data frame
  price
1   1.5
2   0.5
3   1.0
> dframe[[3]][2]  # Second element from third column
[1] 0.5
> dframe$price[2]  # Second element from "price" column
[1] 0.5
> is.data.frame(dframe[[3]]); is.vector(dframe[[3]])
[1] FALSE
[1] TRUE
> dframe[2, ]  # Extract second row
   type color price
2 daisy white   0.5
> dframe[2, ][3]  # Third element from second column
  price
2   0.5
> dframe[2, 3]  # Third element from second column
[1] 0.5
> unlist(dframe[2, ])  # Coerce to vector
   type   color   price
"daisy" "white"   "0.5"
> is.data.frame(dframe[2, ]); is.vector(dframe[2, ])
[1] TRUE
[1] FALSE
```

Data Frames and Factors

By default `data.frame()` does not coerce `character` vectors to `factors`, so no need for the option `stringsAsFactors=FALSE`.

The function `options()` sets global *options*, that determine how `R` computes and displays its results.

If the global `option stringsAsFactors=FALSE` is set, then `character` vectors will not be coerced to `factors` in all subsequent `data frame` operations.

The default is `stringsAsFactors=FALSE` since `R` version `4.0`.

```
> dframe <- data.frame(  # Create a data frame
+                 type=c("rose", "daisy", "tulip"),
+                 color=c("red", "white", "yellow"),
+                 price=c(1.5, 0.5, 1.0),
+                 row.names=c("flower1", "flower2", "flower3")
+               )  # end data.frame
> dframe
         type  color price
flower1  rose    red   1.5
flower2 daisy  white   0.5
flower3 tulip yellow   1.0
> class(dframe$type)  # Get column class
[1] "character"
> class(dframe$price)  # Get column class
[1] "numeric"
> # Set option to not coerce character vectors to factors - that was
> options("stringsAsFactors")
$stringsAsFactors
NULL
> options(stringsAsFactors=FALSE)
> options("stringsAsFactors")
$stringsAsFactors
[1] FALSE
```

Exploring Data Frames

The function `str()` displays the structure of an `R` object.

The functions `head()` and `tail()` display the first and last rows of an `R` object.

```
> str(dframe)  # Display the object structure
'data.frame': 3 obs. of  3 variables:
 $ type : chr  "rose" "daisy" "tulip"
 $ color: chr  "red" "white" "yellow"
 $ price: num  1.5 0.5 1
> dim(cars)  # The cars data frame has 50 rows
[1] 50  2
> head(cars, n=5)  # Get first five rows
  speed dist
1     4    2
2     4   10
3     7    4
4     7   22
5     8   16
> tail(cars, n=5)  # Get last five rows
   speed dist
46    24   70
47    24   92
48    24   93
49    24  120
50    25   85
```

Sorting Vectors

The function `sort()` returns a vector sorted into ascending order.

A permutation is a re-ordering of the elements of a vector.

The permutation index specifies how the elements are re-ordered in a permutation.

The function `order()` calculates the permutation index to sort a given vector into ascending order.

Applying the function `order()` twice: `order(order())`, calculates the permutation index to sort the vector from ascending order into its unsorted (original) order.

So the permutation index produced by: `order(order())` is the reverse of the permutation index produced by: `order()`.

`order()` can take several vectors as input, to break any ties.

Data frames can be sorted on any column.

```
> # Create a named vector of student scores
> scorev <- sample(round(runif(5, min=1, max=10), digits=2))
> names(scorev) <- c("Angie", "Chris", "Suzie", "Matt", "Liz")
> # Sort the vector into ascending order
> sort(scorev)
Angie   Liz Suzie  Matt Chris
 3.02  3.09  6.42  9.15  9.41
> # Calculate index to sort into ascending order
> order(scorev)
[1] 1 5 3 4 2
> # Sort the vector into ascending order
> scorev[order(scorev)]
Angie   Liz Suzie  Matt Chris
 3.02  3.09  6.42  9.15  9.41
> # Calculate the sorted (ordered) vector
> sortv <- scorev[order(scorev)]
> # Calculate index to sort into unsorted (original) order
> order(order(scorev))
[1] 1 5 3 4 2
> sortv[order(order(scorev))]
Angie Chris Suzie  Matt   Liz
 3.02  9.41  6.42  9.15  3.09
> scorev
Angie Chris Suzie  Matt   Liz
 3.02  9.41  6.42  9.15  3.09
> # Examples for sort() with ties
> order(c(2, 1:4))  # There's a tie
[1] 2 1 3 4 5
> order(c(2, 1:4), 1:5)  # There's a tie
[1] 2 1 3 4 5
```

Sorting Data Frames

Data frames can be sorted on any one of its columns.

```
> # Create a vector of student ranks
> rankv <- c("fifth", "fourth", "third", "second", "first")
> # Reverse sort the student ranks according to students
> rankv[order(order(scorev))]
[1] "fifth"  "first"  "third"  "second" "fourth"
> # Create a data frame of students and their ranks
> rosterdf <- data.frame(score=scorev,
+   rank=rankv[order(order(scorev))])
> rosterdf
      score   rank
Angie  3.02  fifth
Chris  9.41  first
Suzie  6.42  third
Matt   9.15 second
Liz    3.09 fourth
> # Permutation index on price column
> order(dframe$price)
[1] 2 3 1
> # Sort dframe on price column
> dframe[order(dframe$price), ]
        type  color price
flower2 daisy  white   0.5
flower3 tulip yellow   1.0
flower1  rose    red   1.5
> # Sort dframe on color column
> dframe[order(dframe$color), ]
        type  color price
flower1  rose    red   1.5
flower2 daisy  white   0.5
flower3 tulip yellow   1.0
```

Coercing Data Frames Into Matrices Using as.matrix()

The function as.matrix() coerces vectors and data frames into matrices.

Coercing a data frame into a matrix causes coercion of numeric values into character.

as.matrix() coerces vectors into single column matrices, as opposed to matrix(), which produces a matrix.

```
> as.matrix(dframe)
        type    color    price
flower1 "rose"  "red"    "1.5"
flower2 "daisy" "white"  "0.5"
flower3 "tulip" "yellow" "1.0"
> vecv <- sample(9)
> matrix(vecv, ncol=3)
     [,1] [,2] [,3]
[1,]    1    3    8
[2,]    5    4    2
[3,]    6    9    7
> as.matrix(vecv, ncol=3)
      [,1]
 [1,]    1
 [2,]    5
 [3,]    6
 [4,]    3
 [5,]    4
 [6,]    9
 [7,]    8
 [8,]    2
 [9,]    7
```

Coercing Matrices Into Data Frames

The generic function `as.data.frame()` coerces matrices and other objects into data frames.

The method `as.data.frame.matrix()` coerces only matrices into data frames.

`as.data.frame.matrix()` is about 50% faster than `as.data.frame()`, because it skips extra `R` code in `as.data.frame()` needed for argument validation, error checking, and method dispatch.

As a general rule, calling generic functions is slower than directly calling individual methods, because generic functions must execute extra `R` code for method dispatch.

The function `data.frame()` can also be used to coerce matrices into data frames, but is much slower than even `as.data.frame()`.

`as.data.frame()` is about three times faster than `data.frame()`, because it doesn't require extra `R` code in `data.frame()` needed for handling different types of vectors, and for method dispatch.

```
> library(microbenchmark)
> # Call method instead of generic function
> as.data.frame.matrix(matv)
> # A few methods for generic function as.data.frame()
> sample(methods(as.data.frame), size=4)
> # Function method is faster than generic function
> summary(microbenchmark(
+   as_dframem=as.data.frame.matrix(matv),
+   as_dframe=as.data.frame(matv),
+   dframe=data.frame(matv),
+   times=10))[, c(1, 4, 5)]  # end microbenchmark summary
```

Coercing Matrices Into Lists

Matrices can be coerced into lists in at least two different ways.

Matrices can be first coerced into a data frame, and then into a list using function `as.list()`.

Matrices can be directly coerced into a list using function `lapply()`.

Using `lapply()` is the faster of the two methods, because `lapply()` is a *compiled* function.

```
> # lapply is faster than coercion function
> summary(microbenchmark(
+   aslist=as.list(as.data.frame.matrix(matv)),
+   lapply=lapply(seq_along(matv[1, ]),
+     function(indeks) matv[, indeks]),
+   times=10))[, c(1, 4, 5)]  # end microbenchmark summary

Error in h(simpleError(msg, call)):  error in evaluating the
argument 'object' in selecting a method for function 'summary':
object 'matv' not found
```

The iris Data Frame

The iris data frame is included in the datasets base package.

iris contains sepal and petal dimensions of 50 flowers from 3 species of iris.

The function unique() extracts unique elements of an object.

sapply() applies a function to a list or a vector of objects and returns a vector.

sapply() performs a loop over the list of objects, and can replace "for" loops in R.

```
> # ?iris  # Get information on iris
> dim(iris)
[1] 150   5
> head(iris, 2)
  Sepal.Length Sepal.Width Petal.Length Petal.Width Species
1          5.1         3.5          1.4         0.2  setosa
2          4.9         3.0          1.4         0.2  setosa
> colnames(iris)
[1] "Sepal.Length" "Sepal.Width"  "Petal.Length" "Petal.Width"  "Spe
> unique(iris$Species)  # List of unique elements of iris
[1] setosa     versicolor virginica
Levels: setosa versicolor virginica
> class(unique(iris$Species))
[1] "factor"
> # Find which columns of iris are numeric
> sapply(iris, is.numeric)
Sepal.Length  Sepal.Width Petal.Length  Petal.Width      Species
        TRUE         TRUE         TRUE         TRUE        FALSE
> # Calculate means of iris columns
> sapply(iris, mean)  # Returns NA for Species
Sepal.Length  Sepal.Width Petal.Length  Petal.Width      Species
        5.84         3.06         3.76         1.20           NA
```

The mtcars Data Frame

The mtcars data frame is included in the datasets base package, and contains design and performance data for 32 automobiles.

```
> # ?mtcars  # mtcars data from 1974 Motor Trend magazine
> # mpg   Miles/(US) gallon
> # qsec   1/4 mile time
> # hp  Gross horsepower
> # wt  Weight (lb/1000)
> # cyl   Number of cylinders
> dim(mtcars)
[1] 32 11
> head(mtcars, 2)
              mpg cyl disp  hp drat   wt qsec vs am gear carb
Mazda RX4      21   6  160 110  3.9 2.62 16.5  0  1    4    4
Mazda RX4 Wag  21   6  160 110  3.9 2.88 17.0  0  1    4    4
> colnames(mtcars)
 [1] "mpg"  "cyl"  "disp" "hp"   "drat" "wt"   "qsec" "vs"   "am"
[11] "carb"
> head(rownames(mtcars), 3)
[1] "Mazda RX4"     "Mazda RX4 Wag" "Datsun 710"
> unique(mtcars$cyl)  # Extract list of car cylinders
[1] 6 4 8
> sapply(mtcars, mean)  # Calculate means of mtcars columns
    mpg     cyl    disp      hp    drat      wt    qsec      vs
 20.091   6.188 230.722 146.688   3.597   3.217  17.849   0.438   0.
   carb
  2.812
```

The Cars93 Data Frame

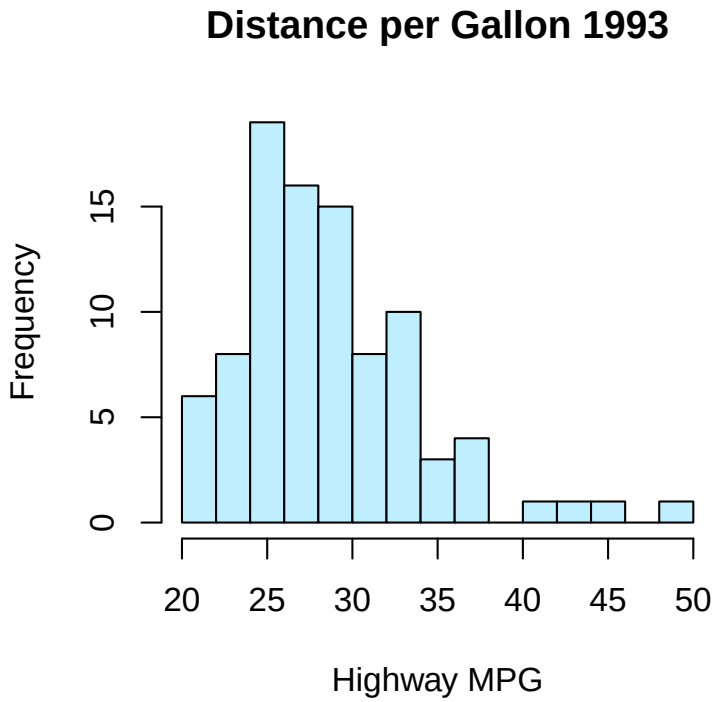
The Cars93 data frame is included in the MASS package, and contains design and performance data for 93 automobiles.

The function hist() calculates and plots a histogram, and returns its data *invisibly*.

The parameter breaks is the number of cells of the histogram.

"FD" stands for the Freedman-Diaconis rule for calculating histogram breaks,

```
> library(MASS)
> # ?Cars93  # Get information on Cars93
> dim(Cars93)
> head(colnames(Cars93))
> # head(Cars93, 2)
> unique(Cars93$Type)  # Extract list of car types
> # sapply(Cars93, mean)  # Calculate means of Cars93 columns
> # Plot histogram of Highway MPG using the Freedman-Diaconis rule
> hist(Cars93$MPG.highway, col="lightblue1",
+      main="Distance per Gallon 1993", xlab="Highway MPG", breaks="FD")
```



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]  3  3 10 18 18  5 12  8 11  6 22 21 15 15  7 19  1 20  2  2 17
> # 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)
            VEU
1993-01-29  NA
1993-02-01  NA
1993-02-02  NA
> sum(is.na(pricev))
[1] 3552
> 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)
             VEU
1993-01-29 30.9
1993-02-01 30.9
1993-02-02 30.9
> 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 30.0   25.8
2  xts 24.6   24.2
```

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)
```

The *Logistic* Function

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

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

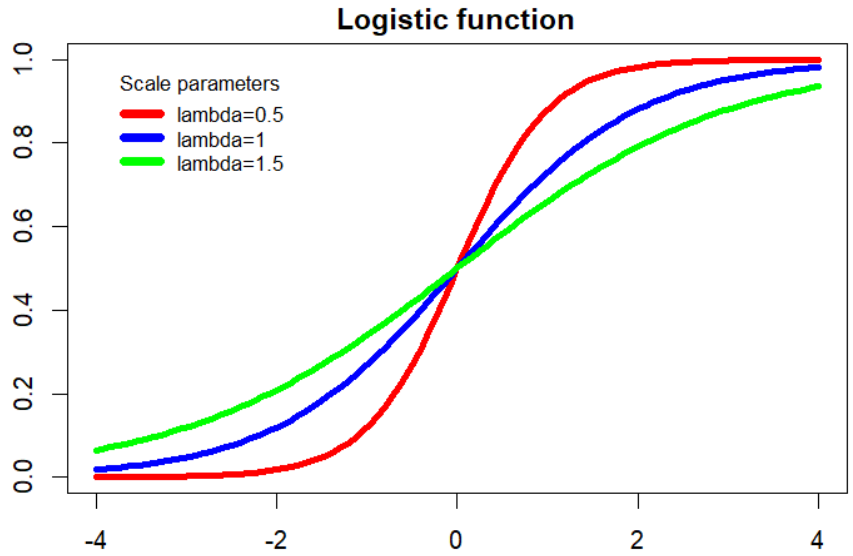
Where λ is the scale (dispersion) parameter.

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.

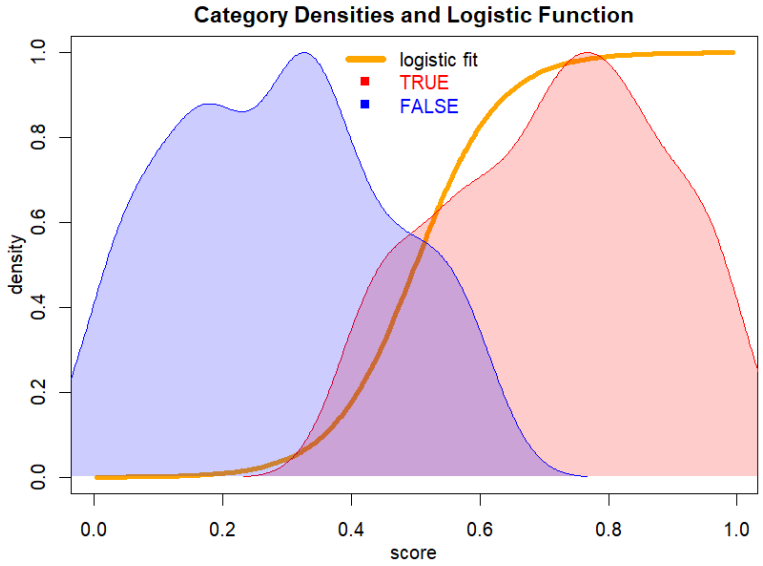
The function `glm()` fits generalized linear models, including *logistic* regressions.

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 $(1 - p)$.

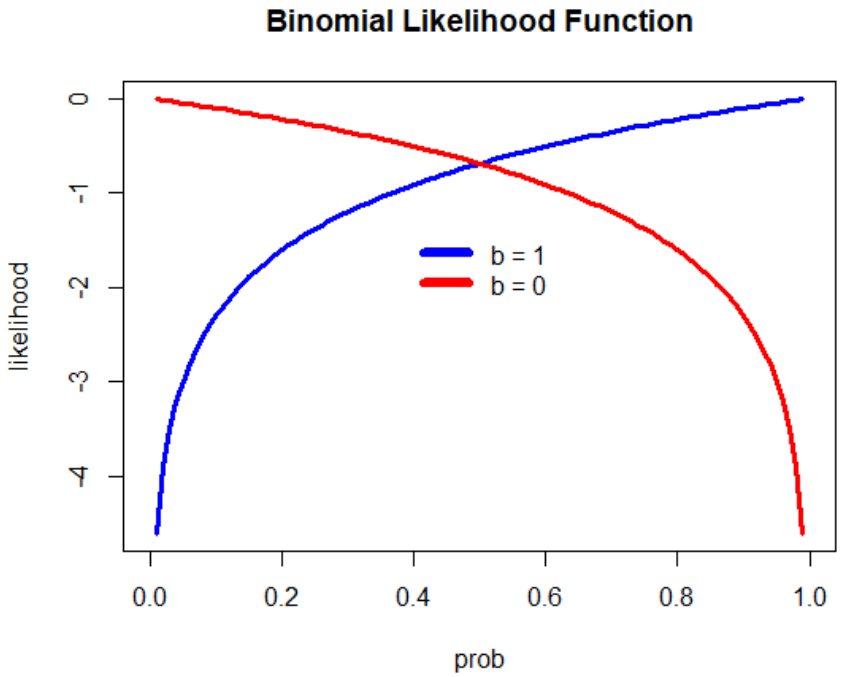
Then the response r follows the binomial distribution:

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

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()`.

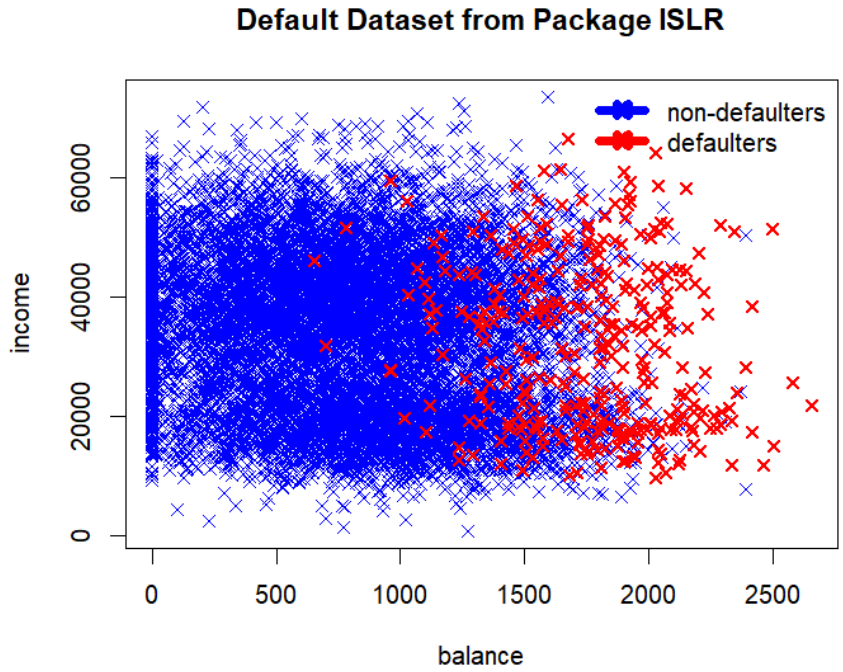
```
> # Coerce the default and student columns to Boolean
> Default <- ISLR::Default
> Default$default <- (Default$default == "Yes")
> Default$student <- (Default$student == "Yes")
> attach(Default)  # Attach Default to search path
> # Explore credit default data
> summary(Default)
  default          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

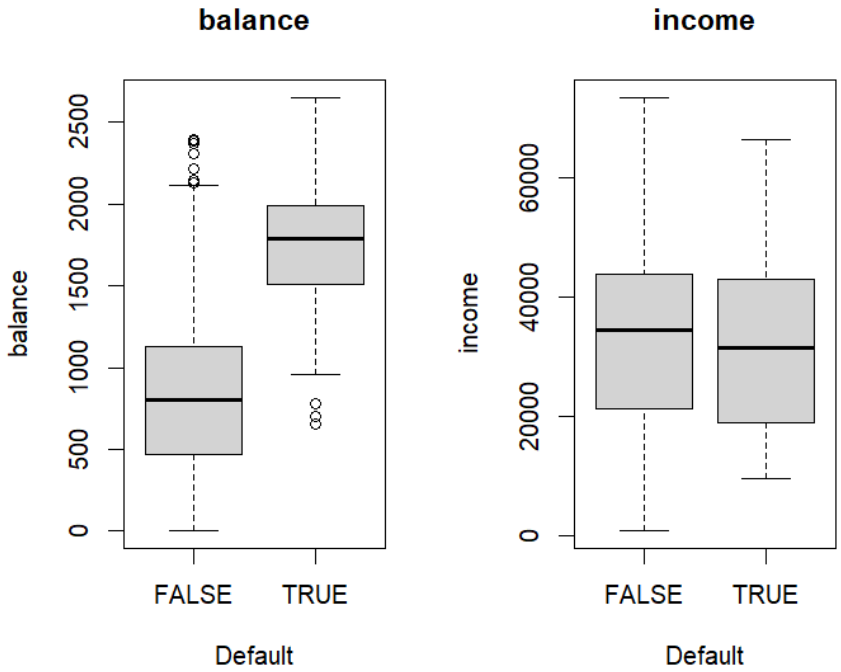
A *Box Plot* (box-and-whisker plot) is a graphical display of a distribution of data:

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])
```

```
> 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")
```

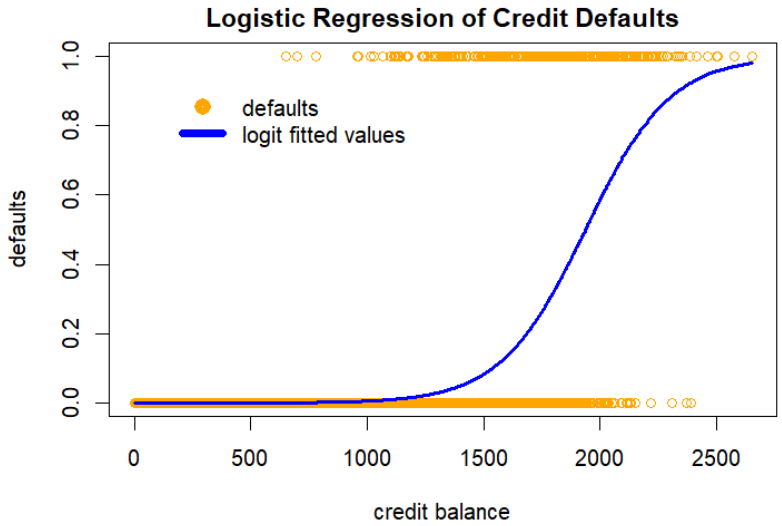
Modeling Credit Defaults Using *Logistic* Regression

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
```



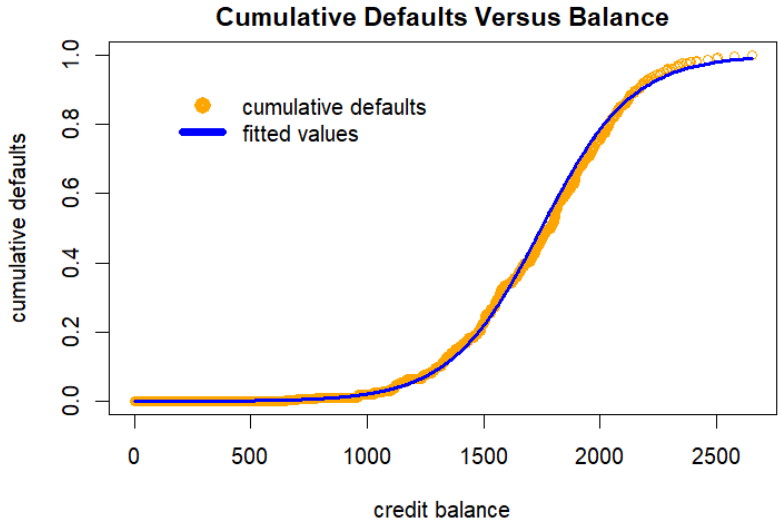

```
> x11(width=6, height=5)
> par(mar=c(4, 4, 2, 2), oma=c(0, 0, 0, 0), mgp=c(2.5, 1, 0))
> plot(x=balance, y=default,
+      main="Logistic Regression of Credit Defaults",
+      col="orange", xlab="credit balance", ylab="defaults")
> ordern <- order(balance)
> lines(x=balance[ordern], y=logmod$fitted.values[ordern], 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
```

Confounding Variables in Multifactor *Logistic* Regression

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

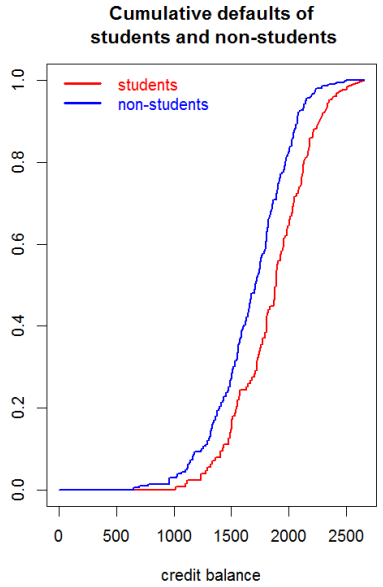
But the coefficient from the single-factor regression is positive (indicating that students are more likely to default), while the coefficient from the multifactor regression is negative (indicating that students are less likely to default).

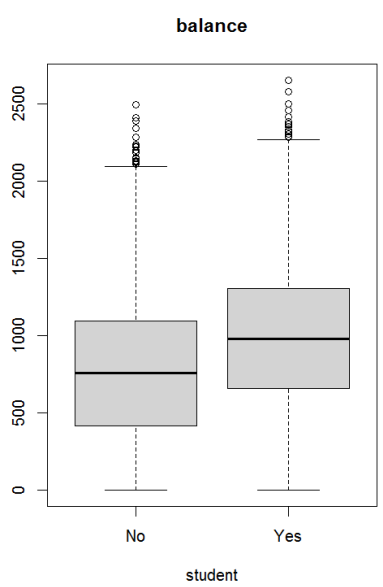
The reason that students are more likely to default is because they have higher credit balances than non-students - which is what the single-factor regression shows.

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
> logmodstud <- glm(default ~ student, family=binomial(logit))
> summary(logmodstud)
> # Multifactor coefficient is negative
> logmod$coefficients
> # Single-factor coefficient is positive
> logmodstud$coefficients
```

```
> # 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
```

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.

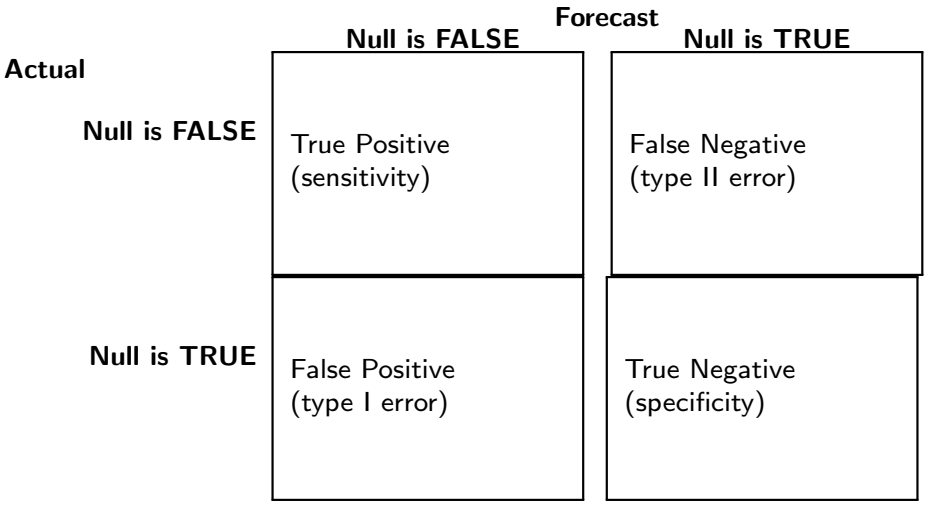
A *type I* error is the incorrect rejection of a `TRUE` *null hypothesis* (i.e. a "false positive"), when there is no default but it's classified as a default.

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.




```
> # 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).

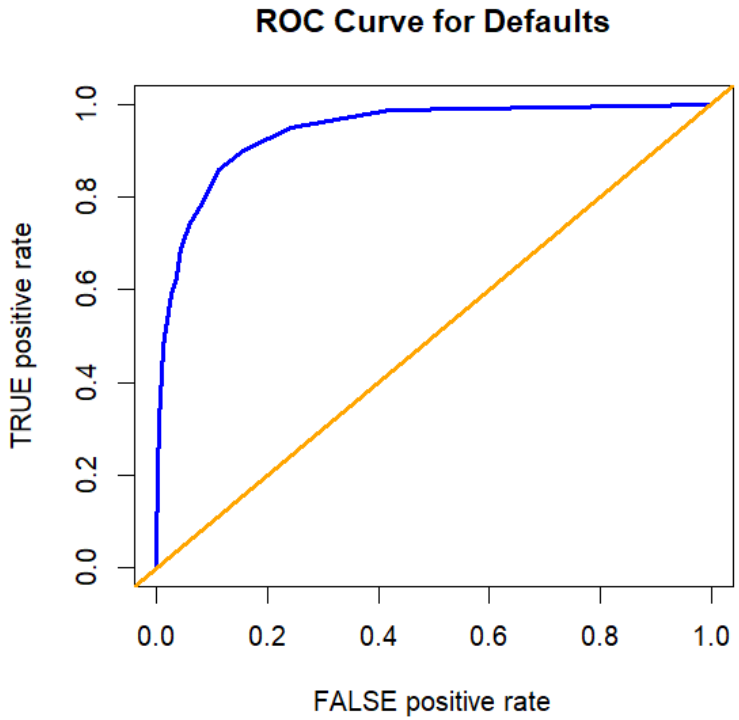
The sum of the *true negative* plus the *false positive* rate is equal to 1.

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")
```

Homework Assignment

Required

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

Recommended

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