Data preparation

When a new dataset is received, we need to start looking at it to understand what it contains. We have already seen a few approaches, and now we apply them to the `diab01.txt` dataset (available on Learn). The 'fread' function in the data.table package is similar to read.table for delimited text files, but much faster at reading huge files.

```
> library(data.table)
> diab01.dt <- fread("data/diab01.txt", stringsAsFactors = TRUE)
> dim(diab01.dt)
[1] 100  10
> head(diab01.dt)
      PAT AGE SEX  BMI  BP  TC   LDL HDL GLU   Y
1: pat001  59   F 32.1 101 157  93.2  38  87 151
2: pat002  48   M 21.6  87 183 103.2  70  69  75
3: pat003  72   F 30.5  93 156  93.6  41  85 141
4: pat004  24   M 25.3  84 198 131.4  40  89 206
5: pat005  50   M 23.0 101 192 125.4  52  80 135
6: pat006  23   M 22.6  89 139  64.8  61  68  97
```

We can see how different variables are coded by using the `str()` function. Calling `summary()` on the whole data table produces summary statistics for all variables.

```
> str(diab01.dt)
> summary(diab01.dt)
```

We can obtain a scatter plot of each variable against all others, but unless the number of variables is very small, it's not always very helpful.

```
> plot(diab01.dt, cex=0.2)      # cex scales the size of the points
```

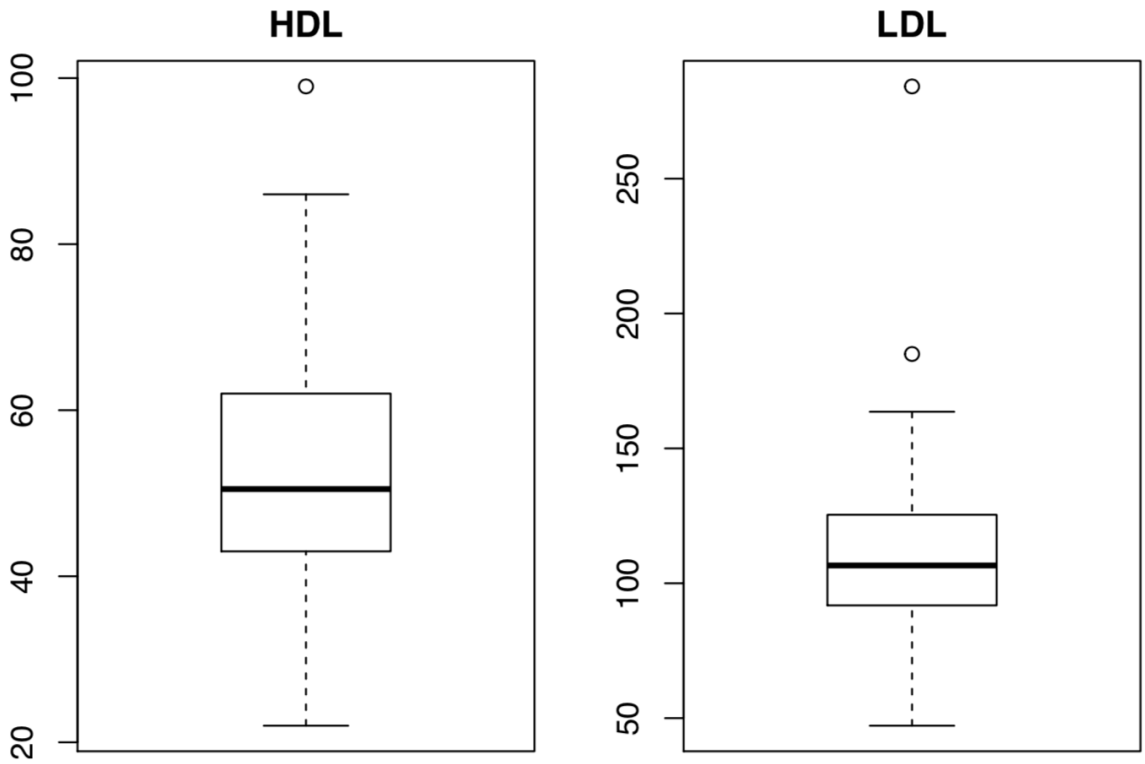
Note that also the patient identifiers were plotted. Usually such identifiers are never used in the data analysis, but they are extremely important to match datasets coming from different sources. We never want to completely detach identifiers from the dataset, as it's very error-prone when we subset the data or try to merge it with other datasets.

Let's plot the distributions of some of these variables.

```
> par(mfrow=c(3,2))           # create a plot with 3 rows and 2 columns
> hist(diab01.dt$AGE, main="Age")
> hist(diab01.dt$BMI, main="BMI")
> hist(diab01.dt$BP, main="Blood pressure")
> hist(diab01.dt$TC, main="Total cholesterol")
> hist(diab01.dt$LDL, main="LDL")
> hist(diab01.dt$HDL, main="HDL")
```

A different way of visualizing variables is by using boxplots, which help to assess the spread of the data, as well as the presence of possible outliers.

```
> par(mfrow=c(1,2), mar=c(2,3,2,1)) # mar changes margins: bottom, left, top, right
> boxplot(diab01.dt$HDL, main="HDL")
> boxplot(diab01.dt$LDL, main="LDL")
```



The thick line in the centre of the box represents the median, and the extremes of the box correspond to the first and third quantiles (25th and 75th percentiles). The two whiskers by default extend to the most extreme data point within 1.5 times the interquartile range of the box. Any data point that exceeds that range appears as a point in the plot: these are possible outliers, but not necessarily so.

Boxplots are a visual way of comparing the summary statistics for two subsets of data. This can be done by using a formula, that is an expression of the type `variable ~ group`. For this to work well, we specify the name of the data table with the `data` argument, otherwise we would have to spell it out twice as `data01$HDL ~ data01$SEX`.

```
> boxplot(HDL ~ SEX, data=diab01.dt, main="HDL stratified by sex")
```

A scatter plot could help in identifying the samples with extreme values. When producing the scatter plot of a single variable, points are plotted in the order they appear in the data table. This may be particularly revealing if the dataset originated from different cohorts (or different batches of measurements), as it may make visually obvious if there are cohort or batch effects in the data.

```
> plot(diab01.dt$LDL, main="Scatter plot of LDL", ylab="LDL")
```

Note that for plots to be most informative, we want them to be properly titled and have readable axis labels (R usually assigns some default ones, but they are not always as good as we'd like). So spend a few seconds to add axis labels (options `xlab` and `ylab`) and title (option `main`) in all your plots.

Missing values

It's very common that datasets contain missing values. These are recorded in R as `NA`.

```
> head(diab01.dt$TC, n=30)
 [1] 157 183 156 198 192 139 160 255 179 180 114  NA 186 186 202 254 207
[18] 214 162 187 156 162 187 210  NA 178 124 158 160 158
```

To count them, we can use the `is.na()` function:

```
> table(is.na(diab01.dt$TC))

FALSE  TRUE
   93     7
```

By default, R will not ignore the missing entries, which is a good thing, so we can become aware of them and decide explicitly how to deal with them. Luckily, in most cases it's very easy to tell R to ignore the `NA`s in a speficic computation.

```
> mean(diab01.dt$TC)              # because of missing values, there's no overall mean
[1] NA
> mean(diab01.dt$TC, na.rm=TRUE)  # the mean is computed only on the observed values
[1] 181.0108
```

In other situations, however, it's not quite as easy. In such cases, we could look at the complete dataset, that is discard all observation with any missing element.

```
> # there are better ways to accomplish this!  (hint - `na.omit() will exclude any row with an NA` )
> diab01.dt.complete <- diab01.dt[ !is.na(BMI) & !is.na(BP) & !is.na(TC) &
+                                  !is.na(LDL) & !is.na(HDL) & !is.na(GLU)]
> table(is.na(diab01.dt.complete))

FALSE
  790
> dim(diab01.dt.complete)        # we lost one fifth of the dataset!
[1] 79 10
```

The other option is imputation. A very simple approach is to impute the missing values to the median value for that variable. Make a copy of the original data before you impute, or alternatively, create a separate column for the imputed value in order to retian the original. When making a copy of data tables, it's very important to understand that a left assignment using `<- or =` will result in a shallow copy, where the resulting structure is a pointer to the original data. To create a deep copy of the data table, you must use the `copy()` function.

```
>
> # Take a deep copy of the original data in order to leave the original intact.
> diab01.dt.imputed <- copy(diab01.dt)
> diab01.dt.imputed[, BMI := ifelse(is.na(BMI), median(BMI, na.rm=T), BMI)]
>
> # Alternatively, create a new column for the imputed result.
> # diab01.dt[, BMI.imp := ifelse(is.na(BMI), median(BMI, na.rm=T), BMI)]
>
> # same for all other variables...
```

We will see later on how to use vector based methods to automate the same operation across a number of variables.

Remember that the validity of the inferences made on the complete dataset depends on the mechanisms of missingness (missing completely at random, missing at random, missing not at random). These mechanisms

also influence the approaches we can use to impute missing values.

Merging datasets

It's very common that medical data comes from different sources. Provided that all sources use the same identifiers, then those can be used to merge the datasets.

```
> diab02.dt <- fread("data/diab02.txt", stringsAsFactors = TRUE)
> str(diab02.dt)
Classes 'data.table' and 'data.frame':  99 obs. of  3 variables:
 $ ID : Factor w/ 99 levels "pat001","pat003",..: 10 4 35 31 15 58 36 87 59 28 ...
 $ TCH: num  6 4 3 2 5 2 6 3 6 3 ...
 $ LTG: num  3.58 4.29 4.83 4.44 4.94 ...
 - attr(*, ".internal.selfref")=<externalptr>
```

First of all, note that the column of identifiers is called differently in the new dataset. Also, you will notice that patients are not listed in the order we had before: this is not a problem, as R can sort this out during the merge. Let's see what's the intersection between identifiers.

```
> length(intersect(diab01.dt$PAT, diab02.dt$ID))
[1] 97
> diab02.dt[!ID %in% diab01.dt$PAT]$ID   # new patients that are not in diab01.dt
[1] pat101 pat102
99 Levels: pat001 pat003 pat004 pat005 pat006 pat007 pat008 ... pat102
> diab01.dt[!PAT %in% diab02.dt$ID]$PAT  # old patients that are not in diab02.dt
[1] pat002 pat009 pat013
100 Levels: pat001 pat002 pat003 pat004 pat005 pat006 pat007 ... pat100
```

So the `diab02.dt` dataset contains 2 patients that were not present in the `diab01.dt` dataset, and 3 patients were in `diab01.dt` but not in `diab02.dt`. Clearly the patients that were in one dataset but not in the other will have missing values corresponding to the variables that they hadn't recorded.

also influence the approaches we can use to impute missing values.

Merging datasets

It's very common that medical data comes from different sources. Provided that all sources use the same identifiers, then those can be used to merge the datasets.

```
> diab02.dt <- fread("data/diab02.txt", stringsAsFactors = TRUE)
> str(diab02.dt)
Classes 'data.table' and 'data.frame':  99 obs. of  3 variables:
 $ ID : Factor w/ 99 levels "pat001","pat003",..: 10 4 35 31 15 58 36 87 59 28 ...
 $ TCH: num  6 4 3 2 5 2 6 3 6 3 ...
 $ LTG: num  3.58 4.29 4.83 4.44 4.94 ...
 - attr(*, ".internal.selfref")=<externalptr>
```

First of all, note that the column of identifiers is called differently in the new dataset. Also, you will notice that patients are not listed in the order we had before: this is not a problem, as R can sort this out during the merge. Let's see what's the intersection between identifiers.

```
> length(intersect(diab01.dt$PAT, diab02.dt$ID))
[1] 97
> diab02.dt[!ID %in% diab01.dt$PAT]$ID   # new patients that are not in diab01.dt
[1] pat101 pat102
99 Levels: pat001 pat003 pat004 pat005 pat006 pat007 pat008 ... pat102
> diab01.dt[!PAT %in% diab02.dt$ID]$PAT  # old patients that are not in diab02.dt
[1] pat002 pat009 pat013
100 Levels: pat001 pat002 pat003 pat004 pat005 pat006 pat007 ... pat100
```

So the `diab02.dt` dataset contains 2 patients that were not present in the `diab01.dt` dataset, and 3 patients were in `diab01.dt` but not in `diab02.dt`. Clearly the patients that were in one dataset but not in the other will have missing values corresponding to the variables that they hadn't recorded.

Let's merge the two datasets using the two columns of identifiers as a way of matching the observations. If we do not specify it, R will match observations from the two datasets using the set of columns that appear in both. In our case, as we know that the identifiers are held in variable `PAT` for `diab01` and variable `ID` for `diab02`, we indicate that explicity with options `by.x` and `by.y`. By default R will create an *inner join* of the two dataset, that is it will keep only the observations that appear in both (the intersection of patients).

```
> diab.dt <- merge(diab01.dt, diab02.dt, by.x="PAT", by.y="ID")
> dim(diab.dt)  # this corresponds to the intersection of patients
[1] 97 12
```

This may not be what we want: if so, we can specify to keep all observations from the first dataset (setting `all.x=TRUE`), or all observations from the second dataset (setting `all.y=TRUE`). For the moment, let's perform a merge so that all observations from both datasets are kept (an *outer join*): by setting `all=TRUE` we imply both `all.x=TRUE` and `all.y=TRUE`.

```
> diab.dt <- merge(diab01.dt, diab02.dt, by.x="PAT", by.y="ID", all=TRUE)
> dim(diab.dt)  # this corresponds to the union of patients
[1] 102  12
```

If in the datasets to be merged there are variables of the same name that are not used in matching observations (that is, they are not listed in the `by` option), both variables are kept in the merged data table, with `.x` and `.y` is appended to their name. To avoid this, you can rename variables using the list `.()` function to rename values before the merge.

```
> diab.alt.dt <- merge(diab01.dt, diab02.dt[,.(PAT=ID, MY1=TCH, MY2=LTG)],
+                      by="PAT", all=TRUE)
> dim(diab.alt.dt)  # this corresponds to the union of patients
[1] 102  12
```

Reusable code

Until now we have worked interactively by inserting commands one at a time from the command prompt. While this works well for seeing hands-on what we are doing, we need to start recording the various steps in a text file (use the `.R` extension) so that they can be recalled in the future with the `source()` command.

Be sure to add comments (starting with the `#` character) to remind yourself what you are meaning to do, especially if you are using complicated commands!

Functions

One of the most important ways of automating some operations is by writing a function that accomplishes that specific task. A function can have any number of arguments, and can return any type of object (but only one object can be returned). This is usually indicated by the `return()` command. The generic way of

```
function.name <- function(arg1, arg2) {
  # body of the function
}
```

For example, the code we used to impute the missing values could be very well incorporated into a simple function.

```
impute.to.median <- function(x) {
  # find which values are missing
  na.idx <- is.na(x)

  # replace NAs with the median computed over the observed values
  x[na.idx] <- median(x, na.rm=TRUE)

  # return the vector with imputed values
  return(x)
}
```

Note that nowhere in the function we had to specify the type of variables that the function applies to. That's because R, as opposed to other languages such as C, is loosely typed, so it will not check your code until the time when it is actually executed.

But you will see that the function fails if it's applied to a non-numeric vector.

```
> impute.to.median(diab01.dt$SEX)
Error in median.default(x, na.rm = TRUE): need numeric data
```

So let's modify the function so that it will return early (without attempting an imputation) if it receives non-numerical data.

```
impute.to.median <- function(x) {
  # check the type of objects we have been given
  if (!(is.numeric(x) || is.integer(x)))
    return(x)

  # find which values are missing
  na.idx <- is.na(x)

  # replace NAs with the median computed over the observed values
  x[na.idx] <- median(x, na.rm=TRUE)

  # return the vector with imputed values
  return(x)
}
```

When the function fails, or cannot handle some of the inputs, we may want to stop and return an error message rather than returning the wrong answer. One way to do this is by using the `stop()` function, which does exactly that.

```
> mean.vector <- function(x) {
+   if (!(is.numeric(x) || is.integer(x)))
+     stop("The function works only for numerical vectors")
+   return(mean(x))
+ }
```

Loops

Many operations are repetitive, in the sense that they need to be performed for all variables, or for all observations, or for a range of parameters and so on.

The `for` loop is often what you need to repeat the same operation over a certain range, but note that it isn't very efficient and should be avoided where you can apply a vector based operation.

```
> for (num in 1:3) {
+   cat("num:", num, "\n")
+ }
num: 1
num: 2
num: 3
```

In R the `for` loop can iterate over almost any list of objects.

```
> for (col in colnames(diab01.dt)) {
+   print(col)
+ }
```

You should be able to see how, for example, we can now quickly impute all missing values in a data table by looping over them. With data tables, all `set*()` operations are generally much more efficient, as they update values by reference.

```
> table(is.na(diab01.dt))

FALSE  TRUE
  977    23
> diab01.dt.imputed <- copy(diab01.dt)   # make a copy to keep the unimputed data around
> for (cols in colnames(diab01.dt.imputed)) {  # using `set()` is a fast way to update huge data tables
+   set(diab01.dt.imputed, j = cols, value = impute.to.median(diab01.dt.imputed[[cols]]))
+ }
> table(is.na(diab01.dt.imputed))

FALSE
 1000
```

So with the `impute.to.median()` function and a small `for` loop we managed to impute most missing values in just a few lines of code!

Sometimes an iteration of a `for` loop must be skipped according to some condition: this can be accomplished by using `next`. Instead, use `break` to interrupt a loop before it reaches its natural termination.

```
> for (idx in 1:100) {
+   if (idx %% 2 == 0) next    # skip even numbers
+   if (idx > 5) break         # stop early despite the loop was set to reach 100
+   cat(idx, "is odd\n")
+ }
1 is odd
3 is odd
5 is odd
```

An alternative to the `for` loop is the function `sapply()`: this allows to apply a given function to each term in the first argument.

```
> abs.vals <- sapply(-5:5, abs)
> abs.vals
 [1] 5 4 3 2 1 0 1 2 3 4 5
```

When applied to a dataframe/data table, `sapply()` operates over each of the columns.

```
> sapply(diab01.dt, class)  # get the class of each column in the data table
      PAT       AGE       SEX       BMI        BP        TC       LDL
 "factor" "integer"  "factor" "numeric" "numeric" "integer" "numeric"
      HDL       GLU         Y
"integer" "integer" "integer"
```

If the function we want to apply is a composite of other functions, we have two options: either to declare the function in the workspace and then use it, or it is possible to define it directly when calling `sapply()` as an *anonymous function*. In the latter case, the function cannot be called again as it doesn't have a name attached to it, so it's convenient only for one-off operations.

```
> num.missing <- function(x) sum(is.na(x))
> sapply(diab01.dt, num.missing)                  # using a named function
PAT AGE SEX BMI  BP  TC LDL HDL GLU   Y
  0   0   0   2   2   7   3   2   7   0
> sapply(diab01.dt, function(z) sum(is.na(z)))    # using an anonymous function
PAT AGE SEX BMI  BP  TC LDL HDL GLU   Y
  0   0   0   2   2   7   3   2   7   0
```

Notice that earlier on we kept a copy of our dataset with unimputed values: this is to allow a quick access to the original dataset, for example to compare the effect of imputation on the distribution of the variables.

```
> num.cols <- which(sapply(diab01.dt,
+                          function(z) (class(z) %in% c("numeric", "integer"))))
> par(mfrow=c(4,2),    # multiple plots in the same image
+     mar=c(2,2,2,1))  # bottom, left, top, right margins
> for (col in num.cols) {
+   plot(density(diab01.dt.imputed[[col]]), col="red",
+        main=colnames(diab01.dt[, ..col]))
+   lines(density(diab01.dt[[col]], na.rm=TRUE), col="black", lty=3)
+ }
```

Assertions

Assertions are checks made to ensure the validity of the assumptions made in the code: it is often the case that datasets change through time intentionally (errors are corrected at source, new observations are added, and so on) or accidentally (a wrong version of a file is read in, a bug is introduced in the code, etc). An assertion can save us a lot of debugging time and give us increased confidence in the correctness of our results.

For example, before we assumed that the order of columns in `diab01.dt` and `diab01.dt.imputed` was the same: since we created one out of the other, we already knew that the order would be identical. But if the two data tables were to undergo different manipulations, it's not guaranteed that the ordering (or indeed the number) of rows and columns would not change.

A simple way of making an assertion is by using `stopifnot()`: in this case, if our condition doesn't hold, the processing will stop at that point (very helpful when using `source()` to run a whole script). Note that multiple conditions can be tested simultaneously, in which case the assertion will fail if any of the conditions is not true.

```
> stopifnot(1 == 3)
Error in eval(expr, envir, enclos): 1 == 3 is not TRUE
> stopifnot(colnames(diab01.dt) == colnames(diab01.dt.imputed))  # no output since it's true
> stopifnot(colnames(diab01.dt) == colnames(diab01.dt.imputed),  # check column ordering
+           diab01.dt$PAT == diab01.dt.imputed$PAT)              # check row ordering
```

Reference classes

R has several class implementations. Here we look at the R6 class package, which is similar to the built-in R classes, but are simpler and smaller. Reference classes allow you to implement object oriented programming in R, enabling self contained, reusable blueprints to write and share your code and methods.

The following R6 class example is a simple regression output reformatter. A class like this could be used again and again when writing papers, supplying data tables for printing directly in R markdown. Through time, your classes can be extended, allowing you and your peers to deliver reproducible research output with consistent methods, formatting Etc.

You may need to install the R6 package in order to run the following code.

install.packages("R6")

```
>
> library(R6)
>
> # Re-run the regression, storing the result
> regr <- lm(Y ~ AGE + SEX + HDL, data=diab01.dt)
>
> # Class definition
> ModelSummary <- R6Class("ModelSummary", public = list(
+   model.store = list(),
+   model.summary.dt = data.table(),
+   model.thresh.set = logical(),
+
+   # The initialize (note spelling) is called by new()
+   initialize = function(model.in) {
+
+     # Input data checks
+     stopifnot(length(model.in$coefficients) >= 2)
+
```

```
+     # Initialise member variables
+     self$model.thresh.set = FALSE
+
+     # Create a data table summarising the results
+     self$model.summary.dt <-
+       data.table(summary(model.in)$coefficients, keep.rownames = TRUE)
+     print(self$model.summary.dt)
+     # Add confidence intervals around the coefficients
+     self$model.summary.dt <- cbind(self$model.summary.dt,
+           confint(model.in))
+
+     # Reformat the output table
+     self$model.summary.dt <-
+       self$model.summary.dt[,
+       .(Name = rn, Estimate = signif(Estimate, 3),
+         `95% CI` = paste0("(", signif(`2.5 %`, 3), ", ",
+       signif(`97.5 %`, 3), ")"), `P-value` = signif(`Pr(>|t|)`, 3) )]
+
+     # Keep a copy of the original model
+     self$model.store <- model.in
+   },
+
+   # Reformat significance level description
+   statsignifdt = function(thresh = 0.05) {
+     stopifnot(is.numeric(thresh))
+
+     if(!self$model.thresh.set) {
+       self$model.summary.dt[, `P-value` := ifelse(`P-value` < thresh,
+                     paste0("<", thresh), as.character(`P-value`))]
+       # Only allow setting of the threshold once
+       self$model.thresh.set = TRUE
+     } else {
+       cat("\nError: P-value threshold has been set previously!\n\n")
+     }
+     return(self$model.summary.dt)
+   },
```

```
+   # The print method is called if the object is viewed
+   print = function() {
+     if(nrow(self$model.summary.dt)>1) {
+       print(self$model.summary.dt)
+     } else {
+       cat("No model data.\n")
+     }
+
+     if(!self$model.thresh.set) {
+       cat("P-value threshold not yet set. Run statsignifdt() method.\n")
+     }
+   })
+ )
>
> # Instantiate an object of class ModelSummary, passing
> # the results of our stored linear regression.
```

```
> mod.desc.obj <- ModelSummary$new(regr)
           rn   Estimate Std. Error   t value     Pr(>|t|)
1: (Intercept) 168.098928 32.9361954  5.103775 1.730632e-06
2:         AGE   1.316011  0.5334562  2.466953 1.543749e-02
3:        SEXM  35.804371 16.2703966  2.200584 3.021509e-02
4:         HDL  -2.200075  0.5313706 -4.140378 7.562832e-05
>
> # Viewing the object in R calls the print method
> mod.desc.obj
          Name Estimate         95% CI  P-value
1: (Intercept)   168.00     (103, 233) 1.73e-06
2:         AGE     1.32  (0.257, 2.38) 1.54e-02
3:        SEXM    35.80    (3.5, 68.1) 3.02e-02
4:         HDL    -2.20 (-3.26, -1.15) 7.56e-05
P-value threshold not yet set. Run statsignifdt() method.
>
> # The internal public objects are available
> print(mod.desc.obj$model.summary.dt)
          Name Estimate         95% CI  P-value
1: (Intercept)   168.00     (103, 233) 1.73e-06
2:         AGE     1.32  (0.257, 2.38) 1.54e-02
3:        SEXM    35.80    (3.5, 68.1) 3.02e-02
4:         HDL    -2.20 (-3.26, -1.15) 7.56e-05
>
> # Member methods may be called, modifying the internal data
> # or providing alternative output Etc.
> mod.desc.obj$statsignifdt(0.001)
>
> # Here we see the result of the method on the last field
> mod.desc.obj
          Name Estimate         95% CI P-value
1: (Intercept)   168.00     (103, 233)  <0.001
2:         AGE     1.32  (0.257, 2.38)  0.0154
3:        SEXM    35.80    (3.5, 68.1)  0.0302
4:         HDL    -2.20 (-3.26, -1.15)  <0.001
```

```
>
> # A feature of our class is that it only allows setting of the threshold once
> mod.desc.obj$statsignifdt(0.005)

Error: P-value threshold has been set previously!
          Name Estimate         95% CI P-value
1: (Intercept)   168.00     (103, 233)  <0.001
2:         AGE     1.32  (0.257, 2.38)  0.0154
3:        SEXM    35.80    (3.5, 68.1)  0.0302
4:         HDL    -2.20 (-3.26, -1.15)  <0.001
```

The class created has three member methods called initialise,

Fitting linear regression models

Let's start by analyzing two samples from a normal distribution. Since these are drawn independently from each other, we do not expect to see any pattern between the two.

```
> set.seed(1)           # initialize the random number generator
> x <- rnorm(100, mean=50, sd=10)
> y <- rnorm(100, mean=75, sd=20)
> cor(x, y)
[1] -0.0009943199
```

To fit a linear regression model, we use `lm()`. The model is specified according to the formula interface, of the form `outcome ~ model`, where `model` consists of the names of the predictors (covariates) we want to include in the model separated by `+`. We do not need to specify an intercept term, as R adds it by default.

```
> regr <- lm(y ~ x)          # linear regression of variables in the workspace
```

The coefficient for variable `x` is very close to zero, which agrees with the correlation coefficient and our expectations. We can extract the regression coefficients from a fitted model using `coef()`.

```
> coef(regr)                 # same as regr$coefficients
 (Intercept)            x
74.352186339 -0.002120772
```

Note that if both variables were standardized (that is, scaled by their respective standard deviations so that they had variance 1), the regression coefficient for `x` would correspond to the correlation coefficient.

```
> x.std <- x / sd(x);  y.std <- y / sd(y)
> coef(lm(y.std ~ x.std))
  (Intercept)         x.std
 3.8810841942 -0.0009943199
```

We can add the regression line to an already open scatter plot by using the `abline()` function. The regression line (in black) coincides almost exactly with the horizontal line drawn at the intercept (in red): effectively, knowing `x` doesn't add any information to what we already knew about `y`.

```
> plot(x, y)
> abline(regr)                          # regression line
> abline(h=coef(regr)[1], col="red")  # horizontal line at the intercept
```

We can extract more information about the fitting of this model by using the `summary()` function.

```
> summary(regr)
```

Among all the output produced, we generally concentrate on the table of hypothesis tests on the regression coefficients. We can limit our output to just that table if necessary.

```
> hyp.tests <- coef(summary(regr))
> hyp.tests
               Estimate Std. Error      t value     Pr(>|t|)
(Intercept) 74.352186339 11.1744506  6.653766602 1.647043e-09
x           -0.002120772  0.2154541 -0.009843269 9.921663e-01
```

As an aside, note that `hyp.test` is of class `matrix`: matrices are two-dimensional vectors and, unlike dataframes/tables, can only store one type of data and do not accept the `$` notation to extract columns (although using column names still works).

The Wald test statistic for the regression coefficients, reported in column labelled `t value`, can be computed manually as the ratio of regression coefficients to standard errors.

```
> tval <- hyp.tests[, "Estimate"] / hyp.tests[, "Std. Error"]
```

These values are compared to the quantiles of a t distribution with $n-p-1$ degrees of freedom, where p is the number of predictors in the model. For the conventional $\alpha = 0.05$ significance level, 95% of the distribution is contained between the 0.025 and 0.975 quantiles. Since the distribution is symmetric, these quantiles only differ by the sign: therefore to compute a p-value it is enough to find the probability corresponding to the absolute value of the test statistic and double it (since we need to account for both tails).

```
> df <- length(y) - 1 - 1                    # one predictor in the model
> qt(c(0.025, 0.975), df)                    # quantiles of a t distribution
[1] -1.984467  1.984467
> 2 * pt(abs(tval), df, lower.tail=FALSE)    # p-values
 (Intercept)            x
1.647043e-09 9.921663e-01
```

In our model we can confidently reject the null hypothesis that the intercept term is zero, as its p-value is well below the standard significance threshold of 0.05. On the other hand, the same doesn't hold for `x`.

```
> hyp.tests[1, 4] < 0.05      # check the p-value of the intercept term
[1] TRUE
> hyp.tests[2, 4] < 0.05      # check the p-value of x
[1] FALSE
```

Realistic data

If we now look at some realistic data, we can start looking for associations between covariates and outcome. In the `diab01.txt` dataset (available on Learn), the outcome of interest is `Y`, which represents a quantitative measure of diabetes status: the higher the score, the more severe is diabetes. Let's start with a simple model in which we study the association between age and diabetes severity.

```
> diab01.dt <- fread("data/diab01.txt", stringsAsFactors = TRUE)
> regr.age <- lm(Y ~ AGE, data=diab01.dt)     # specify the data table of covariates
> coef(summary(regr.age))
             Estimate Std. Error  t value     Pr(>|t|)
(Intercept) 100.5584041 24.1505789 4.163809 6.736605e-05
AGE           0.7202443  0.5052268 1.425586 1.571652e-01
```

We could say that for any additional year of age, our diabetes score increases by 0.7 units. However, the standard error for age is high relative to its regression coefficient: this in turn doesn't allow us to reject the null hypothesis of no age effect, as the p-value (0.157) is above the conventional significance level of 0.05.

We can come to the same conclusions by looking at the 95% confidence intervals for our coefficients: for age, 0 falls within the confidence intervals, so we don't have any support for rejecting the null hypothesis.

```
> confint(regr.age)
                2.5 %    97.5 %
(Intercept) 52.6323663 148.48444
AGE         -0.2823617   1.72285
```

It may be an issue of power: our current sample size of 100 may be be too small to allow us to make confident inferences for the effect sizes we are looking for, and maybe with a larger dataset our conclusions would be different.

Let's now explore the relationship between diabetes severity and other predictors, for example HDL (high density lipoprotein, the so called "good cholesterol"). The easiest approach is to look at the correlation between the two variables: however, given the presence of some missing values in HDL, we need to specify the option `use="pairwise.complete.obs"` to tell `cor()` to use only observations that are not missing in both variabes.

```
> with(diab01.dt, cor(Y, HDL))
[1] NA
> with(diab01.dt, cor(Y, HDL, use="pairwise.complete.obs"))
[1] -0.3199029
```

We see a modest inverse relationship between the two variables, that is when HDL grows, the diabetes score decreases. We can confirm this with a linear regression model: again, given the presence of missing observations, R will perform the analysis of the *complete dataset*, the one that remains after all samples with missing values are removed.

```
> regr.hdl <- lm(Y ~ HDL, data=diab01.dt)
> summary(regr.hdl)

Call:
lm(formula = Y ~ HDL, data = diab01.dt)

Residuals:
     Min       1Q   Median       3Q      Max
-105.371  -52.372   -6.937   38.415  167.067

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept) 216.5354    25.8217   8.386 4.31e-13 ***
HDL          -1.5719     0.4751  -3.308  0.00132 **
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 66.03 on 96 degrees of freedom
  (2 observations deleted due to missingness)
Multiple R-squared:  0.1023,    Adjusted R-squared:  0.09299
F-statistic: 10.94 on 1 and 96 DF,  p-value: 0.001322
```

The negative sign for the HDL coefficient means that there is an inverse association between HDL and the outcome variable: given that `Y` is larger the worse the patient's diabetes status, then a negative coefficient can be interpreted as having a protective effect: an increase in HDL by one unit reduces the diabetes score by 1.57 units.

Given that the p-value for HDL (0.00132) is below the significance threshold, we can say that HDL is significantly associated with the diabetes score. Notice that R adds a number of $*$ according to the magnitude of the p-value, and for HDL we would have rejected the null hypothesis of no effect even if we had set $\alpha = 0.01$.

However, it may be argued that HDL depends on other factors, such as age and sex: by not taking into account these other risk factors, our conclusions may be completely wrong. What we really want to know is if the association with HDL is independent of age and sex, that is: if age and sex were included in the model, would the association change?

To answer this, let's add age and sex to the model with HDL and see what happens.

```
> regr <- lm(Y ~ AGE + SEX + HDL, data=diab01.dt)
> summary(regr)

Call:
lm(formula = Y ~ AGE + SEX + HDL, data = diab01.dt)

Residuals:
    Min      1Q  Median      3Q     Max
-112.59  -43.90  -11.35   42.42  167.53

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept) 168.0989    32.9362   5.104 1.73e-06 ***
AGE           1.3160     0.5335   2.467   0.0154 *
SEXM         35.8044    16.2704   2.201   0.0302 *
HDL          -2.2001     0.5314  -4.140 7.56e-05 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 64.19 on 94 degrees of freedom
  (2 observations deleted due to missingness)
Multiple R-squared:  0.1693,    Adjusted R-squared:  0.1428
F-statistic: 6.387 on 3 and 94 DF,  p-value: 0.0005501
```

We can claim that there is a significant association between HDL and our outcome variable even after adjusting for age and sex. Indeed, the effect size has increased: earlier we saw that one additional unit of HDL would decrease the diabetes score by 1.57, but after taking into account age and sex, the score would decrease by 2.2 units.

Note that in this model, we can now claim that age has a non-zero effect on the diabetes score. This can be interpreted as follows: while age doesn't seem to be able to explain our outcome variable directly, it can explain the residuals produced by having sex and HDL in the model.

It is customary to start looking at associations one variable at a time after adjusting for some known *confounders* (such as age, sex, study cohort, etc), that is variables that are known to affect the outcome or other predictors. Confounders are called like that because if they are ignored, they may change the conclusions of our investigation, sometimes in a drastic manner.

Standardized coefficients

What can we claim about the effect size of the variables in the model? Given that each variable has its own unit measure, it's impossible to compare effect sizes directly. We need to scale variables in such a way that accounts for the spread of each variable.

We accomplish this by dividing each continuous variable by its standard deviation, which produces *standardized coefficients*. Here we use of `:=` and `lapply()` which allows to modify the data table in place across column (lists).

```
> diab01.dt.sd1 <- copy(diab01.dt)
> covar.cols <- colnames(diab01.dt.sd1[, -c("PAT", "SEX", "Y")]) # Exclude non covariate columns
> diab01.dt.sd1[, (covar.cols) := lapply(.SD,
+                 function(x) x / sd(x, na.rm = TRUE)), .SDcols = covar.cols]
> summary(lm(Y ~ AGE + SEX + HDL, data=diab01.dt.sd1))
```

```
Call:
lm(formula = Y ~ AGE + SEX + HDL, data = diab01.dt.sd1)

Residuals:
    Min      1Q  Median      3Q     Max
-112.59  -43.90  -11.35   42.42  167.53

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept)  168.099     32.936   5.104 1.73e-06 ***
AGE           18.015      7.302   2.467   0.0154 *
SEXM          35.804     16.270   2.201   0.0302 *
HDL          -31.044      7.498  -4.140 7.56e-05 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 64.19 on 94 degrees of freedom
  (2 observations deleted due to missingness)
Multiple R-squared:  0.1693,    Adjusted R-squared:  0.1428
F-statistic: 6.387 on 3 and 94 DF,  p-value: 0.0005501
```

This allows us to state that the effect of one standard deviation change in HDL is almost double than what produced by a change of one standard deviation in age (but in opposite directions).

In real-life analyses we generally work with standardized variables, so that comparing effects sizes of different variables is straightforward. When producing a model for clinical use, we may instead want to report equations that are in the original units (unstandardized), as a clinician will not have access to standard deviations.

Design matrix

The design matrix is the matrix of all elements in the predictors and the intercept. To retrieve it, we can use function `model.matrix()` on a fitted regression object or on a model formula. When using a formula it's not necessary to specify the oucome variable, as this does not enter the design matrix.

```
> X <- model.matrix(~ AGE + SEX + HDL, data=diab01.dt)  # same as model.matrix(regr)
> dim(X)
[1] 98  4
```

Note that because of missing values in the HDL variable, the design matrix only contains 98 observations (instead of 100): by default, R fits models on the complete dataset, so it discards any observations containing missing values. To retrieve also the observations with missing values, we need to set an option to control that behaviour (but note that this will affect all other operations unless we reset it).

```
> old.opts <- options(na.action='na.pass')
> X.na <- model.matrix(~ AGE + SEX + HDL, data=diab01.dt)
> dim(X.na)
[1] 100   4
> options(old.opts)       # reset all options to their default values
```

As `SEX` is a categorical variable (with levels `F` and `M`), when it is used in the model, its levels are coded as dummy variables. Given a reference category (in this case, being female), we code all other categores by contrast: this means that we create a binary variable (R assigns the name `SEXM` to it) which codes for being male.

```
> head(diab01.dt[, c("AGE", "SEX")])
   AGE SEX
1:  59   F
2:  48   M
3:  72   F
4:  24   M
5:  50   M
6:  23   M
```

```
> head(model.matrix(~ AGE + SEX, data=diab01.dt))
  (Intercept) AGE SEXM
1           1  59    0
2           1  48    1
3           1  72    0
4           1  24    1
5           1  50    1
6           1  23    1
```

For a categorical variable with k levels, the design matrix contains $k-1$ dummy variables corresponding to all non-reference categories. For example, suppose we categorized patients by age in three strata:

```
> diab01.dt$AGE.CAT <- cut(diab01.dt$AGE, c(0, 40, 60, Inf))
> head(diab01.dt[, c("AGE", "AGE.CAT")])
   AGE  AGE.CAT
1:  59  (40,60]
2:  48  (40,60]
3:  72 (60,Inf]
4:  24   (0,40]
5:  50  (40,60]
6:  23   (0,40]
> head(model.matrix(~ AGE.CAT, data=diab01.dt))     # reference level is (0,40]
  (Intercept) AGE.CAT(40,60] AGE.CAT(60,Inf]
1           1              1               0
2           1              1               0
3           1              0               1
4           1              0               0
5           1              1               0
6           1              0               0
> diab01.dt$AGE.CAT <- NULL      # remove the variable from the table
```

One of the most important outputs from a regression model is the vector of fitted values $\hat{y} = X\hat{\beta}$, where X is our design matrix and $\hat{\beta}$ are the regression coefficients.

```
> X <- model.matrix(regr)            # design matrix
> beta.hat <- coef(regr)             # regression coefficients
> y.hat <- X %*% beta.hat            # matrix multiplication
```

We generally do not need to compute it explicitly, as it is already present in the regression object alongside other helpful quantities.

```
> regr$fitted.values
> regr$model           # complete dataset used in fitting the linear model
> regr$residuals
> all.equal(regr$residuals, regr$model$Y - regr$fitted.values)
```

Performance measures

Different performance measures are printed out when `summary()` is applied to the result of a regression model. If its output is stored to a variable, we can then capture their content by using the `$` operator.

```
> summ.regr <- summary(regr)
> ls(summ.regr)             # list all objects that are stored inside the variable
 [1] "adj.r.squared" "aliased"       "call"          "coefficients"
 [5] "cov.unscaled"  "df"            "fstatistic"    "na.action"
 [9] "r.squared"     "residuals"     "sigma"         "terms"
> summ.regr$coefficients    # same as coef(summ.regr)
             Estimate Std. Error   t value     Pr(>|t|)
(Intercept) 168.098928 32.9361954  5.103775 1.730632e-06
AGE           1.316011  0.5334562  2.466953 1.543749e-02
SEXM         35.804371 16.2703966  2.200584 3.021509e-02
HDL          -2.200075  0.5313706 -4.140378 7.562832e-05
```

An explanation of each of the terms is given in `?summary.lm`. Among these, we are particularly interested in the adjusted R^2, as this penalises the classical R^2 according to the number of predictors used in the model.

```
> summ.regr$sigma          # residual standard error
[1] 64.19219
> summ.regr$r.squared
[1] 0.1693172
> summ.regr$adj.r.squared
[1] 0.1428061
```

These quantities can be computed manually by implementing the formulas shown in class.

$$\text{RSE} = \sqrt{\frac{1}{n-p-1}\varepsilon^T\varepsilon}, \qquad R^2 = \frac{\sum_i(\hat{y}_i - \bar{y})^2}{\sum_i(y_i - \bar{y})^2}, \qquad R^2_{\text{adj}} = 1 - (1-R^2)\frac{n-1}{n-p-1}$$

```
> rse <- function(residuals, num.predictors)
+   sqrt(sum(residuals^2) / (length(residuals) - num.predictors - 1))
> r.squared <- function(y.obs, y.pred)
+   sum((y.pred - mean(y.obs))^2) / sum((y.obs - mean(y.obs))^2)
> adj.r.squared <- function(r.squared, n, num.predictors)
+   1 - (1 - r.squared) * (n - 1) / (n - num.predictors - 1)
```

To visualise the relationship between R^2 and adjusted R^2, let's assume a dataset of size $n = 100$ and $R^2 \in [0.25, 1]$, and plot the adjusted R^2 as a function of R^2 for $p \in \{5, 10, 25, 50\}$.

```
> x.values <- seq(0.25, 1, by=0.05)
> num.predictors <- c(2, 10, 25, 50)
>
> ## first do a plot with the first setting of the variables
> plot(x.values, adj.r.squared(x.values, 100, 2), ylim=c(-0.1, 1),
+      xlab="R^2", ylab="Adjusted R^2", type="l")   # connects points with lines
>
> ## then add all other lines
> for (i in 2:length(num.predictors)) {
+   points(x.values, adj.r.squared(x.values, 100, num.predictors[i]),
+          type="l", col=i)
+ }
```

The plot shows that the adjusted R^2 is always lower, and decreases faster the more predictors are used in the model. If the number of predictors is large enough, relative to the sample size, it also possible for an adjusted R^2 to go negative.