CompStat/R - Paper 3

Group 2: Carlo Michaelis, Patrick Molligo, Lukas Ruff

06 July 2016

Part I: Linear regression

In this first part of the paper, we will program a function which estimates the unknown parameters β and σ of a (ordinary) linear regression model

$$y = X\beta + \varepsilon, \qquad \varepsilon \sim N(0, \sigma^2 I)$$

by the ordinary least squares (OLS) method. For a given design matrix X and response vector y the OLS estimator is given by

$$\hat{\beta} = (X'X)^{-1}X'y \tag{1}$$

with covariance matrix

$$\mathrm{Var}(\hat{\beta}) = \sigma^2(X'X)^{-1} \tag{2}$$

where σ^2 has to be estimated via the sum of squared residuals (SSR):

$$\hat{\sigma}^2 = \frac{SSR}{df} = \frac{\sum_i (y_i - x_i'\hat{\beta})^2}{n-k}. \tag{3}$$

The term df refers to the degrees of freedom, i.e. the difference between the number of observations n and the number of coefficients k.

Raw implementation

The function `linModEst` is a raw implementation of the OLS estimator. The function takes the response vector y (`y`) and design matrix X (`x`) as arguments and returns a list with the following named elements:

- `coefficients`: the estimated coefficients $\hat{\beta}$
- `vcov`: the estimated covariance matrix $\widehat{\mathrm{Var}(\hat{\beta})}$
- `sigma`: the square root of the estimated scale parameter $\hat{\sigma}^2$
- `df`: the degrees of freedom df

We use equations (1), (2), and (3) for the implementation and compute the inverse of $X'X$ using the `solve` function, which numerically solves the equation

$$(X'X)\,A = I$$

for the matrix $A = (X'X)^{-1}$. To efficiently compute $X'X$ and $X'y$, we use the `crossprod` function.

```r
linModEst <- function(x, y) {
  # Computes the OLS estimator and sample variance assuming a (ordinary) linear
  # regression model.
  #
  # Args:
  #   x: design matrix x
  #   y: response vector y
  #
  # Returns:
  #   A list with the following named elements:
  #     $coefficients: the estimated coefficients
  #     $vcov: the estimated covariance matrix
  #     $sigma: the square root of the estimated variance
  #     $df: the degrees of freedom in the model, i.e. the difference between
  #          the number of rows and columns of x

  # Compute the inverse of (x'x) using the solve- and crossprod-function
  inv <- solve(crossprod(x), diag(nrow = ncol(x)))

  # Compute beta hat, i.e. the estimated coefficients
  coefficients <- inv %*% crossprod(x, y)

  # Compute the degrees of freedom
  df <- nrow(x) - ncol(x)

  # Compute the sample variance via the sum of squared residuals (SSR)
  SSR <- sum((y - x %*% coefficients)^2)
  sigmaSquared <- SSR / df

  # Compute the covariance matrix
  vcov <- sigmaSquared * inv

  # Create named results list to be returned
  results <- list(coefficients, vcov, sqrt(sigmaSquared), df)
  names(results) <- c("coefficients", "vcov", "sigma", "df")

  # Return results
  results
}
```

We test our implementation by computing the linear relationship between heart weight, body weight and sex for the `cats` dataset contained in the package `MASS`. In the following piece of code, `cbind` combines its arguments by columns into a matrix with the number of columns given by the number of arguments and the number of rows given by the greatest length of the given arguments. Shorter arguments are repeated, as long as the matrix number of rows is a multiple of the shorter vector lengths.
Hence, `cbind(1, cats$Bwt, as.numeric(cats$Sex) - 1)` creates a design matrix with an intercept column out of a vector of ones, the variable body weight (`bwt`), and the variable sex (`Sex`), which is converted from a factor into a dummy variable using `as.numeric`. We subtract 1 to receive dummy variable values of 0 and 1, rather than 1 and 2 from the original data. Thus, `cbind` is used to build a proper design matrix of object type `matrix` with an intercept and dummy variable, such that our implementation of `linModEst` works correctly.

```r
# Load cats dataset
data(cats, package = "MASS")

# Compute OLS using our implementation
linModEst(
  x = cbind(1, cats$Bwt, as.numeric(cats$Sex) - 1),
  y = cats$Hwt
)
```

```
## $coefficients
##             [,1]
## [1,] -0.41495263
## [2,]  4.07576892
## [3,] -0.08209684
##
## $vcov
##             [,1]        [,2]        [,3]
## [1,]  0.52900070 -0.20504763  0.06563743
## [2,] -0.20504763  0.08690026 -0.04696312
## [3,]  0.06563743 -0.04696312  0.09244480
##
## $sigma
## [1] 1.457138
##
## $df
## [1] 141
```

We verify our results by comparing them to the output of `R`'s `lm` function:

```r
summary(lm(Hwt ~ Bwt + Sex, data = cats))
```

```
##
## Call:
## lm(formula = Hwt ~ Bwt + Sex, data = cats)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -3.5833 -0.9700 -0.0948  1.0432  5.1016
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept)  -0.4149     0.7273  -0.571    0.569
## Bwt           4.0758     0.2948  13.826   <2e-16 ***
## SexM         -0.0821     0.3040  -0.270    0.788
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 1.457 on 141 degrees of freedom
## Multiple R-squared:  0.6468, Adjusted R-squared:  0.6418
## F-statistic: 129.1 on 2 and 141 DF,  p-value: < 2.2e-16
```

As we can see, our implementation is correct.

Extend implementation

In this section, we write a new function `linMod(formula, data)`, which estimates a linear regression model specified by `formula` and uses our `linModEst` function defined above to estimate the model parameters again by the OLS method. `linMod` returns a list with the following named elements:

- `coefficients`: named vector of the estimated coefficients $\hat{\beta}$
- `vcov`: named estimated covariance matrix $\widehat{\text{Var}(\hat{\beta})}$
- `sigma`: the square root of the estimated scale parameter $\hat{\sigma}^2$
- `df`: the degrees of freedom *df*
- `formula`: the formula that represents the model equation
- `call`: the arguments with which `linMod` was called

Below, we use the `model.frame`, `model.extract`, and `model.matrix` functions, which are very convenient for working with objects of the class `formula`. `model.frame` returns a `data.frame` containing only the variables from its passed `data` argument, which are used in the `formula` expression given. The returned `data.frame` from the `model.frame` function has additional attributes, but these are not needed in our application. With `model.extract`, we are able to extract the response variable from the `data.frame` created by `model.frame`. Moreover, using `model.matrix`, we can create the design matrix (of object class `matrix`) again only from `formula` and `data` arguments. By default, the matrix returned by `model.matrix` includes an intercept and converts factor variables into proper dummy variables (i.e. a factor variable with L levels results in $L-1$ dummy variables). More precisely, the default intercept is taken over from the `formula` object, which then by default adds an intercept term to the model equation, if not specified otherwise. Finally, we use `match.call` to return the call of our function with all the specified arguments by their full names.

```
linMod <- function(formula, data) {
  # Computes the OLS estimator and sample variance assuming a (ordinary) linear
  # regression model with model equation specified by the formula-argument.
  #
  # Args:
  #   formula: a formula specifying the linear model equation
  #   data: a data.frame, list or environment, containing the variables used in
  #         formula
  #
  # Returns:
  #   A list with the following named elements:
  #     $coefficients: named vector of the estimated coefficients
  #     $vcov: named estimated covariance matrix
  #     $sigma: the square root of the estimated variance
  #     $df: the degrees of freedom in the model
  #     $formula: the formula that represents the model equation
  #     $call: the arguments with which the function was called

  # Extract the response variable using the model.extract function on the
  # data.frame returned by model.frame
  y <- model.extract(model.frame(formula, data = data), "response")

  # Create the design matrix using model.matrix, which overtakes an intercept
  # specified in the formula argument by default and converts factor variables into proper
  # dummy variables
  x <- model.matrix(formula, data = data)

  # Use previously defined linModEst for estimation
```

```
  tmp <- linModEst(x, y)

  # Prepare the output
  rownames(tmp$coefficients) <- colnames(x)
  colnames(tmp$vcov) <- colnames(x)
  rownames(tmp$vcov) <- colnames(x)

  # Create results list to be returned
  results <- c(tmp, formula, match.call())
  names(results) <- c("coefficients", "vcov", "sigma", "df", "formula", "call")

  # Return results
  results
}
```

Let's again test our implementation:

```
linMod(Hwt ~ Bwt + Sex, data = cats)
```

```
## $coefficients
##                     [,1]
## (Intercept) -0.41495263
## Bwt          4.07576892
## SexM        -0.08209684
##
## $vcov
##             (Intercept)         Bwt        SexM
## (Intercept)  0.52900070 -0.20504763  0.06563743
## Bwt         -0.20504763  0.08690026 -0.04696312
## SexM         0.06563743 -0.04696312  0.09244480
##
## $sigma
## [1] 1.457138
##
## $df
## [1] 141
##
## $formula
## Hwt ~ Bwt + Sex
##
## $call
## linMod(formula = Hwt ~ Bwt + Sex, data = cats)
```

As we can see, the output has the desired format and the correct results.

Part II: S3 for linear models

In this section we will expand upon our linear regression function using `R`'s object oriented system `S3`. Our goal is to improve the function by returning a more concise output and ultimately replicating the results of the `lm` function. We'll start by redefining `linMod` from Part I and assigning it the class `linMod`. While assigning the a new class, the function operates as *constructor*. As the name implies, this function "constructs" instances of the class `linMod`. So the result will be a list of this type. The assigned class can be used to call a specific method of the generic function `print`.

1. Define the class

First we implement the constructor `linMod` (modified function from Part I). For some additional calculations later (in summary method), we also return the data `x` and the response `y`.

```
linMod <- function(formula, data) {
  # Extract responsevariable and design matrix
  y <- model.extract(model.frame(formula, data = data), "response")
  x <- model.matrix(formula, data = data)

  # Calulcate model, use function from part I
  tmp <- linModEst(x, y)

  # Prepare the output
  rownames(tmp$coefficients) <- colnames(x)
  colnames(tmp$vcov) <- colnames(x)
  rownames(tmp$vcov) <- colnames(x)

  # Create results list to be returned, additionaly add x and y as list
  results <- c(tmp, formula, match.call(), list(x, y))
  names(results) <- c("coefficients", "vcov", "sigma", "df",
                      "formula", "call", "x", "y")

  # Redefine the results to class "linMod"
  # This function has now the role of a constructor
  class(results) <- "linMod"

  # Return instance of class "linMod"
  results
}
```

Now we want to have a look at the structure of our list, which is instance of the `linMod` class:

```
# Call constructur function which constructs an instance of class "linMod"
# Internally it still uses the model calculation from Part I (linModEst)
modelFit <- linMod(Hwt ~ Bwt + Sex, data = cats)

# Verify the structure of modelFit
str(modelFit)
```

```
## List of 8
##  $ coefficients: num [1:3, 1] -0.415 4.0758 -0.0821
##   ..- attr(*, "dimnames")=List of 2
##   .. ..$ : chr [1:3] "(Intercept)" "Bwt" "SexM"
```

```
##   .. ..$ : NULL
##  $ vcov        : num [1:3, 1:3] 0.529 -0.205 0.0656 -0.205 0.0869 ...
##   ..- attr(*, "dimnames")=List of 2
##   .. ..$ : chr [1:3] "(Intercept)" "Bwt" "SexM"
##   .. ..$ : chr [1:3] "(Intercept)" "Bwt" "SexM"
##  $ sigma       : num 1.46
##  $ df          : int 141
##  $ formula     :Class 'formula'  language Hwt ~ Bwt + Sex
##   .. ..- attr(*, ".Environment")=<environment: R_GlobalEnv>
##  $ call        : language linMod(formula = Hwt ~ Bwt + Sex, data = cats)
##  $ x           : num [1:144, 1:3] 1 1 1 1 1 1 1 1 1 1 ...
##   ..- attr(*, "dimnames")=List of 2
##   .. ..$ : chr [1:144] "1" "2" "3" "4" ...
##   .. ..$ : chr [1:3] "(Intercept)" "Bwt" "SexM"
##   ..- attr(*, "assign")= int [1:3] 0 1 2
##   ..- attr(*, "contrasts")=List of 1
##   .. ..$ Sex: chr "contr.treatment"
##  $ y           : Named num [1:144] 7 7.4 9.5 7.2 7.3 7.6 8.1 8.2 8.3 8.5 ...
##   ..- attr(*, "names")= chr [1:144] "1" "2" "3" "4" ...
##  - attr(*, "class")= chr "linMod"
```

2. Print method

Now we'd like to define a printing method for all objects of class `linMod` so that the function returns a more readable and concise output. We want to use a style which is close to the style for printing the `lm` information. To avoid writing a function with a new name, we define a nem method `print.lindMod` of the generic function `print`, assigned to the class `linMod`.

```
print.linMod <- function(x, ...)  {
  # Use cat, sapply and sprintf function to neatly return
  # the function call and coefficients
  #
  # Args:
  #   x: instance of class linMod
  #   ...: passed arguments from generic function to this method
  #
  # Returns:
  #   This function does not retun anything, it just produces an output

  cat("Call:\n", deparse(x$call), "\n\n", "Coefficients:\n",
      sapply(rownames(x$coefficients), sprintf, fmt="%11s"),"\n",
      sapply(round(x$coefficients, digits = 3), sprintf, fmt="%11s"),"\n",
      sep="")
}
```

We simply want to print the model to see that the output is now clear and easy to read and similar to `lm`. It is not necessary to call the defined method above explicitly in some way. `R` will recognize the newly defined `linMod` class and then locate the `linMod` *method* that we have written for the generic `print` function. We just need to call `print`. And becaus `R` automatically calls print, if just enter the object, we just need to call:

```
# Print modelFit
modelFit
```

```
## Call:
## linMod(formula = Hwt ~ Bwt + Sex, data = cats)
##
## Coefficients:
## (Intercept)         Bwt        SexM
##      -0.415       4.076      -0.082
```

The output is now much clearer and user-friendly.

3. Summary method

Next, we define a function, which will add some more information to the `modelFit` object. This function will be a method for our class `linMod` again. We just want to call the `summary` function and we want that `R` automatically detects the class and calls our new method `summary.linMod`.

To build the summary we need to caluclate the *standard error* $\sigma(\hat{\beta})$, the *t value*, the *p value*, the *residuals* and the *coefficient of determination*. We first want to have a look at the calculation for these values.

The *standard error* is calculated by the square root of the diagonal of the covariance matrix $\text{Var}(\hat{\beta})$.

$$\sigma(\hat{\beta}) = \sqrt{\text{diag}\left(\text{Var}(\hat{\beta})\right)} \tag{4}$$

To compute the *t value*, we first need to define our hypothesis. In the case of regression we usually test all parameters against a zero vector, which means, that $\beta_0 = 0$.

$$t = \frac{\beta - \beta_o}{\sigma(\hat{\beta})} = \frac{\beta}{\sigma(\hat{\beta})} \tag{5}$$

For the calculation of the *p value* we need to calculate the propability that our estimator is zero, using the t distribution. In `R` we can use the `pt` function to compute the probability. Mathematically, for a two sided test with symmetric distribution it means that:

$$p = 2 \cdot \mathbb{P}(T \geq t) \tag{6}$$

The *residuals* are simply the difference between the response y and the modeled response $\hat{y}$.

$$\hat{\epsilon} = y - \hat{y} = y - X\hat{\beta} \tag{7}$$

At last, we calculate the *coefficient of determination* R^2 (also known as R-squared), which indicates the proportion of the variance in the dependent variable that is predictable from the independent variable. To calculate this coeffiecient, we need to caluclate the variance of the residuals $\hat{\epsilon}$, which we call RSS, and the variation of y, which we call TSS.

$$R^2 = 1 - \frac{\text{RSS}}{\text{TSS}} \tag{8}$$

In the `summary.linMod` method, we compute all these values, combine them, give a class and return the results as follows:

```r
summary.linMod <- function(x, ...)  {
  # This summary method for ojects of class linMod adds additional information
  # It computes standard errors, t values, p values, residuals and
  # the r squared and adds these information to the existsing information
  #
  # Args:
  #   x: instance of class linMod
  #   ...: passed arguments from generic function to this method
  #
  # Returns:
  # A list with the following named elements:
  #     $coefficients: named matrix of the estimated coefficients,
  #                    their standard errors, t values and p values
  #     $vcov: named estimated covariance matrix
  #     $sigma: the square root of the estimated variance
  #     $df: the degrees of freedom in the model
  #     $formula: the formula that represents the model equation
  #     $call: the arguments with which the function was called
  #     $residuals: the residuals of the regression line
  #     $rSquared: coefficient of determination of the model

  # Compute extra information for coefficients and combine them
  estimates <- x$coefficients
  stdErr <- sqrt(diag(x$vcov))
  tValue <- estimates/stdErr
  pValue <- 2*pt(-abs(tValue),df=(nrow(x$x)-ncol(x$x)))
  x$coefficients <- cbind(estimates, stdErr, tValue, pValue)
  colnames(x$coefficients) <- c("Estimate", "Std. Err.", "t value", "P(>|t|)")

  # Compute residuals on basis of estimated response
  residuals <- x$y - (x$x %*% estimates)
  row.names(residuals) <- NULL  # Remove row numbering

  # Compute r squared of whole model
  rSquared <- 1-(var(residuals)/var(x$y))

  # Combine all result in one list
  results <- c(x, list(residuals=residuals[,1], rSquared=rSquared))

  # Redefine the results to class "summary.linMod"
  # This function has now the role of a constructor
  class(results) <- "summary.linMod"

  # Return results
  results
}
```

After the implementation, we can test the strcture of our new object:

```r
# Call constructur function which constructs an
# instance of class "summary.linMod"
summaryOfMOdelFit <- summary(modelFit)
```

```r
# Verify the structure of summaryOfMOdelFit
str(summaryOfMOdelFit)
```

```
## List of 10
##  $ coefficients: num [1:3, 1:4] -0.415 4.0758 -0.0821 0.7273 0.2948 ...
##   ..- attr(*, "dimnames")=List of 2
##   .. ..$ : chr [1:3] "(Intercept)" "Bwt" "SexM"
##   .. ..$ : chr [1:4] "Estimate" "Std. Err." "t value" "P(>|t|)"
##  $ vcov        : num [1:3, 1:3] 0.529 -0.205 0.0656 -0.205 0.0869 ...
##   ..- attr(*, "dimnames")=List of 2
##   .. ..$ : chr [1:3] "(Intercept)" "Bwt" "SexM"
##   .. ..$ : chr [1:3] "(Intercept)" "Bwt" "SexM"
##  $ sigma       : num 1.46
##  $ df          : int 141
##  $ formula     :Class 'formula'  language Hwt ~ Bwt + Sex
##   .. ..- attr(*, ".Environment")=<environment: R_GlobalEnv>
##  $ call        : language linMod(formula = Hwt ~ Bwt + Sex, data = cats)
##  $ x           : num [1:144, 1:3] 1 1 1 1 1 1 1 1 1 1 ...
##   ..- attr(*, "dimnames")=List of 2
##   .. ..$ : chr [1:144] "1" "2" "3" "4" ...
##   .. ..$ : chr [1:3] "(Intercept)" "Bwt" "SexM"
##   ..- attr(*, "assign")= int [1:3] 0 1 2
##   ..- attr(*, "contrasts")=List of 1
##   .. ..$ Sex: chr "contr.treatment"
##  $ y           : Named num [1:144] 7 7.4 9.5 7.2 7.3 7.6 8.1 8.2 8.3 8.5 ...
##   ..- attr(*, "names")= chr [1:144] "1" "2" "3" "4" ...
##  $ residuals   : num [1:144] -0.737 -0.337 1.763 -0.944 -0.844 ...
##  $ rSquared    : num [1, 1] 0.647
##  - attr(*, "class")= chr "summary.linMod"
```

We can see, that we now have a list of 10 entries, where `$residuals` and `$rSquared` are new, and additionally the coefficient vector is extended.

4. Print method for summary.linMod

In the next step we want to have a method ro print objects of class `summary.linMod` in a self defined way. For this purpose we define a new method `print.summary.linMod` which, again, bases on the generic function `print`. The goal is to have an output which is similar to the `print.summary.lm` method.

```r
print.summary.linMod <- function(x, ...)  {
  # Use cat, sapply and sprintf function to neatly return
  # the function call, the extended coefficients,
  # the residuals and the r squared
  #
  # Args:
  #   x: instance of class summary.linMod
  #   ...: passed arguments from generic function to this method
  #
  # Returns:
  #   This function does not retun anything, it just produces an output

  # Prepare residuals using quantile and mean function
```

```r
# Combine the results in a specific order and name it
resQuant <- quantile(x$residuals)
resMean <- mean(x$residuals)
residuals <- c(resQuant[1:3],resMean,resQuant[4:5])
names(residuals) <- c("Min.","1st Qu.","Median","Mean","3rd Qu.","Max.")

# Prepare the extended coefficients
# First the significant levels are defined, then the colnames are prepared
# and in the last step the values are managed

# Prepare significant levels
pValues <- x$coefficients[,4]
pStars <- sapply(pValues, function(p) {
  if(p < 0.001) {
    return("***")
  } else if(p < 0.01) {
    return("**")
  } else if(p < 0.05) {
    return("*")
  } else if(p < 0.1) {
    return(".")
  } else if(p < 1) {
    return(" ")
  }
})

# Prepare colnames of coeffiecients
coefficientsFormatting <- c("%-11s %8s %9s %7s %7s%4s")
coefficientsNames <- sprintf(coefficientsFormatting,
                             " ",
                             colnames(x$coefficients)[1],
                             colnames(x$coefficients)[2],
                             colnames(x$coefficients)[3],
                             colnames(x$coefficients)[4],
                             " ")

# Prepare values of coeffiecients, using the significant levels
coefficients <- NULL
for(i in 1:nrow(x$coefficients)) {
  coefficients <- paste(coefficients,
                        sprintf(coefficientsFormatting,
                                rownames(x$coefficients)[i],
                                sprintf("%.4f", round(x$coefficients[i,1], 4)),
                                sprintf("%.4f", round(x$coefficients[i,2], 4)),
                                sprintf("%.2f", round(x$coefficients[i,3], 2)),
                                format.pval(x$coefficients[i,4], 2),
                                pStars[i]
                                ),
                        sep="\n")
}

# Print everything using cat
cat("Call:\n", deparse(x$call),"\n\n",
```

```
      "Residuals:\n",
      sapply(names(residuals), sprintf, fmt="%9s"), "\n",
      sapply(sprintf("%.5f", round(residuals, 5)),
             sprintf, fmt="%9s"), "\n\n",
      "Coefficients:\n", coefficientsNames, coefficients, "\n",
      "---\n",
      "Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1\n\n",
      "Multiple R-squared: ", x$rSquared, "\n",
      "More Stats ...",
      sep="")
}
```

If we simply print the object `summaryOfMOdelFit` of class `summary.linMod`, `R` again automatically detects the class and chooses the correct method, defined by us.

```
# Print summaryOfMOdelFit
summaryOfMOdelFit
```

```
## Call:
## linMod(formula = Hwt ~ Bwt + Sex, data = cats)
##
## Residuals:
##      Min.  1st Qu.   Median     Mean  3rd Qu.     Max.
##  -3.58330 -0.97004 -0.09479 -0.00000  1.04322  5.10155
##
## Coefficients:
##             Estimate Std. Err. t value P(>|t|)
## (Intercept)  -0.4150    0.7273   -0.57    0.57
## Bwt           4.0758    0.2948   13.83  <2e-16 ***
## SexM         -0.0821    0.3040   -0.27    0.79
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Multiple R-squared: 0.6468035
## More Stats ...
```

As we can see, the result is beatyfull and very similar to the output of `print.summary.lm`.

5. Plot method

The last method will be a Q-Q plot. The Q-Q plot compares two probability distributions. In our case we want to compare the distribution of the result values `y` with the theoretical quantiles of the standard normal distribution. For that purpose we can use the `qqnorm` function. Additionally we can draw a line of the ideal relation, using the `qqline` function from the `R` base package.

The method will be written for objects of class `linMod`. In this case, we have to compute the residuals again, because the residual calculation before was done for objects of class `summary.linMod`, which is not the input here.

```
plot.linMod <- function(x, ...) {
  # Use qqnorm and qqline to plot a Q-Q plot
  #
```

```
  # Args:
  #   x: instance of class linMod
  #   ...: passed arguments from generic function to this method
  #
  # Returns:
  #   This function does not retun anything, it just produces an output

  # Compute the residuals
  residuals <- x$y - (x$x %*% x$coefficients)

  # Plot residuals against standard normal quantiles
  qqnorm(residuals, main="Normal QQ-Plot of Model Residuals")

  # Additionally draw a line with the ideal relation
  qqline(residuals)
}
```

If we call the generic plot function, `R` will find the correct method again, because `modelFit` is of class `linMod`.

```
# Plot modelFit
plot(modelFit)
```