The all-mighty linear model function `lm()` and friends

An R Tutorial by Gaston Sanchez

Contents

1) Reminder: Linear Regression Model

I assume you are familiar with linear regression models, but just in case, let me begin with a brief reminder of some important concepts.

Suppose we have a response variable Y that we want to predict using p explanatory variables $X_1, X_2, \dots, X_p$. In plain vanilla linear models, we depart from the following relationship between the response and the predictors:

$$Y = \beta_0 + \beta_1 X_1 + \beta_2 X_2 + \cdots + \beta_p X_p + \varepsilon$$

where ε is a random error term, assumed uncorrelated from observation to observation, with mean zero and constant variance σ^2.

As usual, we suppose that a data set of $n \geq p+1$ points has been collected. If we denote x_{ij} the i-th value of the variable X_j then the model generates a system of equations linear in $\beta_0, \beta_1, \ldots, \beta_p$ of the form

$$\begin{aligned}
y_1 &= \beta_0 + \beta_1 x_{11} + \beta_2 x_{12} + \cdots + \beta_p x_{1p} + \varepsilon_1 \\
&\vdots \\
y_i &= \beta_0 + \beta_1 x_{i1} + \beta_2 x_{i2} + \cdots + \beta_p x_{ip} + \varepsilon_i \\
&\vdots \\
y_n &= \beta_0 + \beta_1 x_{n1} + \beta_2 x_{n2} + \cdots + \beta_p x_{np} + \varepsilon_n
\end{aligned}$$

1.1) Multiple Linear Model in Matrix Notation

We can express the system of equations of a multiple linear model in vector-matrix form.

$$\mathbf{y} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_i \\ \vdots \\ y_n \end{bmatrix} \quad \mathbf{X} = \begin{bmatrix} 1 & x_{11} & x_{12} & \cdots & x_{1j} & \cdots & x_{1p} \\ 1 & x_{21} & x_{22} & \cdots & x_{2j} & \cdots & x_{2p} \\ \vdots & \vdots & \vdots & & \vdots & & \vdots \\ 1 & x_{i1} & x_{i2} & \cdots & x_{ij} & \cdots & x_{ip} \\ \vdots & \vdots & \vdots & & \vdots & & \vdots \\ 1 & x_{n1} & x_{n2} & \cdots & x_{nj} & \cdots & x_{np} \end{bmatrix} \quad \boldsymbol{\beta} = \begin{bmatrix} \beta_0 \\ \beta_1 \\ \beta_2 \\ \vdots \\ \beta_j \\ \vdots \\ \beta_p \end{bmatrix} \quad \boldsymbol{\varepsilon} = \begin{bmatrix} \varepsilon_1 \\ \varepsilon_2 \\ \vdots \\ \varepsilon_i \\ \vdots \\ \varepsilon_n \end{bmatrix}$$

and note that the system of linear equations can be expressed using matrix notation as

$$\mathbf{y} = \mathbf{X}\boldsymbol{\beta} + \boldsymbol{\varepsilon}$$

- the error vector $\boldsymbol{\varepsilon}$ is a random vector
- the response vector $\mathbf{y}$ is also a random vector
- the $\mathbf{X}$ matrix is of dimension $n \times (p+1)$; its j-th column contains the regressor x_j measured with negligible error
- it is customary to represent the constant vector—first column of matrix $\mathbf{X}$—as $\mathbf{x_0} = \mathbf{1}_n$
- $\mathbf{X}$ is referred to as the **model matrix** or the design matrix;

In this form, $\mathbf{y}$ and $\boldsymbol{\varepsilon}$ are each $n \times 1$ random vectors. The vector $\boldsymbol{\beta}$ is a $(p+1) \times 1$ vector of unknown parameters and $\mathbf{X}$ is an $n \times (p+1)$ matrix of scalars.

1.2) Least Squares for MLR Model

The goal is to estimate the coefficients $\beta_0, \beta_1, \ldots, \beta_p$. The most common estimation method is **ordinary least squares** (OLS). If $\mathbf{X}^\mathsf{T}\mathbf{X}$ is invertible, then the estimated coefficients $\hat{\boldsymbol{\beta}} = \mathbf{b}$ are given by:

$$\mathbf{b} = (\mathbf{X}^\mathsf{T}\mathbf{X})^{-1}\mathbf{X}^\mathsf{T}\mathbf{y}$$

2) Simple Linear Regression

Let's start with a simple linear regression model. For illustration purposes we'll use the data set `mtcars` (a few rows shown below)

```
                   mpg cyl disp  hp drat    wt  qsec vs am gear carb
Mazda RX4         21.0   6  160 110 3.90 2.620 16.46  0  1    4    4
Mazda RX4 Wag     21.0   6  160 110 3.90 2.875 17.02  0  1    4    4
Datsun 710        22.8   4  108  93 3.85 2.320 18.61  1  1    4    1
Hornet 4 Drive    21.4   6  258 110 3.08 3.215 19.44  1  0    3    1
Hornet Sportabout 18.7   8  360 175 3.15 3.440 17.02  0  0    3    2
Valiant           18.1   6  225 105 2.76 3.460 20.22  1  0    3    1
```

The variables in `mtcars` are:

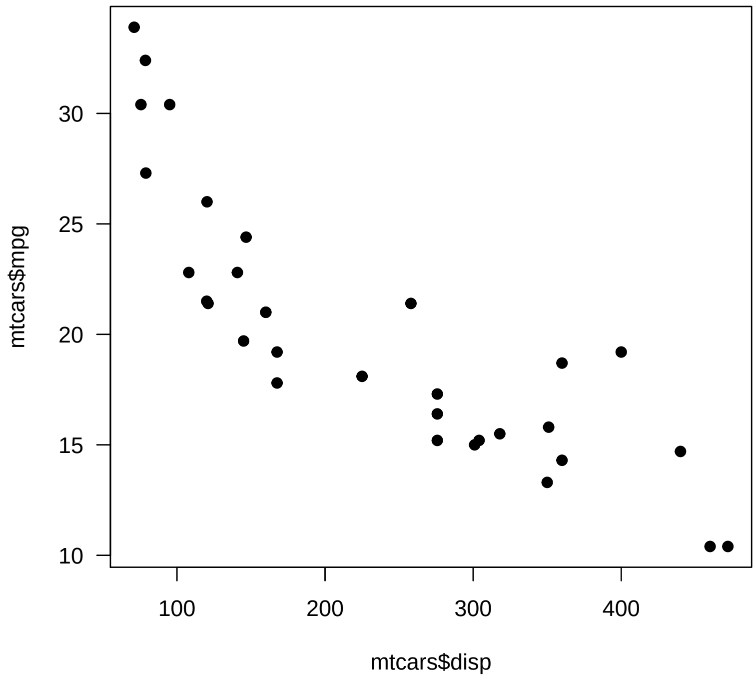
- `mpg`: Miles/(US) gallon
- `cyl`: Number of cylinders
- `disp`: Displacement (cu.in.)
- `hp`: Gross horsepower
- `drat`: Rear axle ratio
- `wt`: Weight (1000 lbs)
- `qsec`: 1/4 mile time
- `vs`: Engine (0 = V-shaped, 1 = straight)
- `am`: Transmission (0 = automatic, 1 = manual)
- `gear`: Number of forward gears
- `carb`: Number of carburetors

Say we want to fit a simple linear model in which miles-per-gallon (`mpg`) is regressed on displacement (`disp`), that is:

$$\texttt{mpg} = \beta_0 + \beta_1 \texttt{disp} + \varepsilon$$

We can take look at the scatterplot between `disp` and `mpg` to get an idea of the direction and form of their relationship:

```
# scatterplot
plot(mtcars$disp, mtcars$mpg, las = 1, pch =19)
```



3) Linear Model Function `lm()`

In R, the function that allows you to fit a regression model via Least Squares is `lm()`, which stands for *linear model*. I should say that this function is a general function that works for various types of linear models such as regression, single stratum analysis of variance, and analysis of covariance.

The main arguments to `lm()` are:

```
lm(formula, data, subset, na.action)
```

where:

- `formula` is the *model formula* (the only required argument)
- `data` is an optional data frame
- `subset` is an index vector specifying a subset of the data to be used (by default all items are used)
- `na.action` is a function specifying how missing values are to be handled (by default missing values are omitted)

3.1) Example: Simple Linear Regression

Let's bring back the simple linear model:

$$\texttt{mpg} = \beta_0 + \beta_1 \texttt{disp} + \varepsilon$$

How do you specify the formula for this model with `lm()`? When the predictor(s) and the response variable are all in a single data frame, you can use `lm()` as follows:

```
# simple linear regression
reg = lm(mpg ~ disp, data = mtcars)
```

The first argument of `lm()` consists of an R formula: `mpg ~ disp`. The tilde, `~`, is the formula operator used to indicate that `mpg` *is predicted* or *described* by `disp`.

The second argument, `data = mtcars`, is used to indicate the name of the data frame that contains the variables `mpg` and `disp`, which in this case is the object `mtcars`. Working with data frames and using this argument is strongly recommended.

3.2) Quick inspection of `lm()` output

The output of `lm()` is an object of class `"lm"`. When you print the `"lm"` objects `reg`, R displays the following information:

```
reg = lm(mpg ~ disp, data = mtcars)
reg
```

```
Call:
lm(formula = mpg ~ disp, data = mtcars)

Coefficients:
(Intercept)         disp
   29.59985     -0.04122
```

Notice that the output contains two parts: `Call:` and `Coefficients:`.

The first part of the output, `Call:`, simply tells you the command used to run the analysis, in this case: `lm(formula = mpg ~ disp, data = mtcars)`.

The second part of the output, `Coefficients:`, shows information about the regression coefficients. The intercept is 29.6, and the other coefficient is -0.0412. Observe the names used by R to display the intercept b_0. While the intercept has the same name `(Intercept)`, the non-intercept term is displayed with the name of the associated variable `disp`.

The printed output of `reg` is very minimalist. However, `reg` contains more information. To see a list of the different components in `reg`, use the function `names()`:

```r
# what's in an "lm" object?
names(reg)
```

```
 [1] "coefficients"  "residuals"     "effects"       "rank"
 [5] "fitted.values" "assign"        "qr"            "df.residual"
 [9] "xlevels"       "call"          "terms"         "model"
```

As you can tell, `reg` contains many more things than just the `coefficients`. In fact, the output of `lm()` is heavily focused on statistical inference, designed to provide results that you can use to form confidence intervals and perform significance tests.

Here's a short description of each of the output elements:

- `coefficients`: a named vector of coefficients.
- `residuals`: the residuals, that is, response minus fitted values.
- `fitted.values`: the fitted mean values.
- `rank`: the numeric rank of the fitted linear model.
- `df.residual`: the residual degrees of freedom.
- `call`: the matched call.
- `terms`: the terms object used.
- `model`: if requested (the default), the model frame used.

To inspect what's in each returned component, type the name of the regression object, `reg`, followed by the `$` dollar operator, followed by the name of the desired component. For example, to inspect the `coefficients` run this:

```r
# regression coefficients
reg$coefficients
```

```
(Intercept)        disp
29.59985476 -0.04121512
```

Likewise, to take a peek at the fitted values use `$fitted.values`

```r
# fitted values
head(reg$fitted.values, n = 8)
```

```
        Mazda RX4     Mazda RX4 Wag        Datsun 710    Hornet 4 Drive
         23.00544          23.00544          25.14862          18.96635
Hornet Sportabout           Valiant        Duster 360         Merc 240D
         14.76241          20.32645          14.76241          23.55360
```

Alternatively, `lm()`—and other similar model fitting functions—have generic helper functions to extract the output elements such as:

- `coef()` to extract the coefficients
- `fitted()` to extract the fitted values
- `residuals()` to extract the residuals

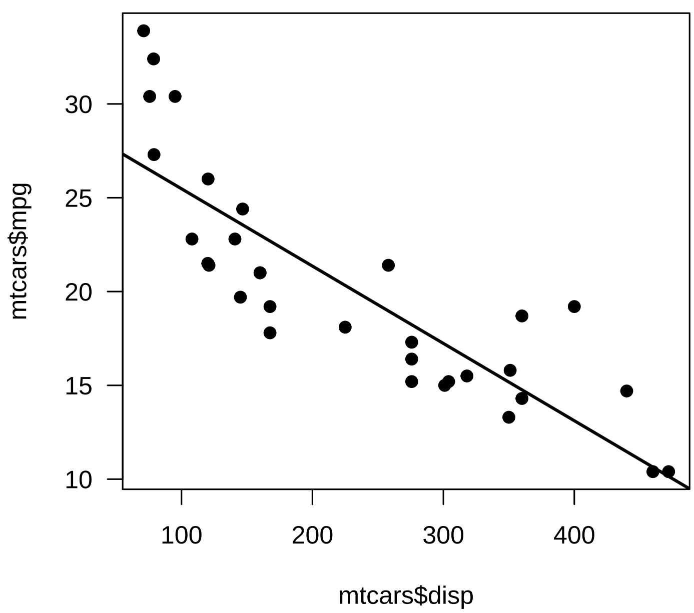
4) Plotting the Regression Line

With a simple linear regression model (i.e. one predictor), once you obtained the `"lm"` object `reg`, you can use it to get a scatterplot with the regression line on it.

4.1) Using `plot()` and `abline()`

The simplest way to achieve this visualization is to first create a scatter diagram with `plot()`, and then add the regression line with the function `abline()`; here's the code in R:

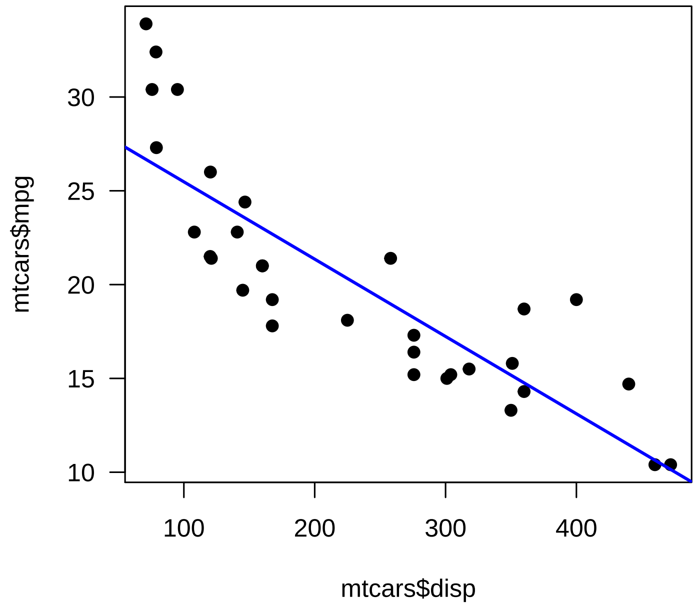
```
# scatterplot with regression line
plot(mtcars$disp, mtcars$mpg, las = 1, pch = 19)
abline(reg, lwd = 2)
```



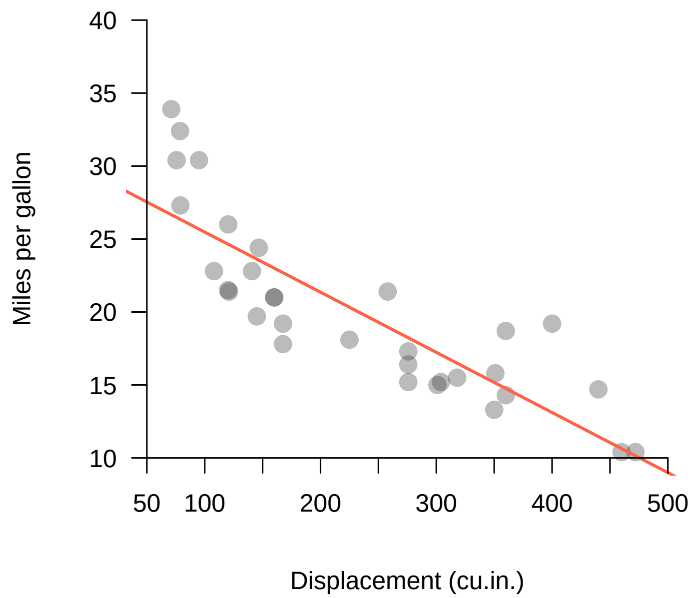
The function `abline()` allows you to add lines to a `plot()`. The good news is that `abline()` recognizes objects of class `"lm"`, and when invoked after a call to `plot()`, it will add the regression line to the plotted chart.

You can tweak some of the `abline()` arguments to increase the width of the line (`lwd`), or change its color (`col`)

```r
# scatterplot with regression line
plot(mtcars$disp, mtcars$mpg, las = 1, pch = 19)
abline(reg, lwd = 2, col = "blue")
```



Here's how to get a nicer plot using low-level plotting functions:




```r
# scatterplot with regression line
plot.new()
plot.window(xlim = c(50, 500), ylim = c(10, 40))
title(xlab = 'Displacement (cu.in.)', ylab = 'Miles per gallon')
points(mtcars$disp, mtcars$mpg, pch = 19, cex = 1.5, col = "#33333355")
```

```r
abline(reg, col = "tomato", lwd = 2)    # regression line
axis(side = 1, pos = 10, at = seq(50, 500, 50))
axis(side = 2, las = 1, pos = 50, at = seq(10, 40, 5))
```

4.2) Regression Line with "ggplot2"

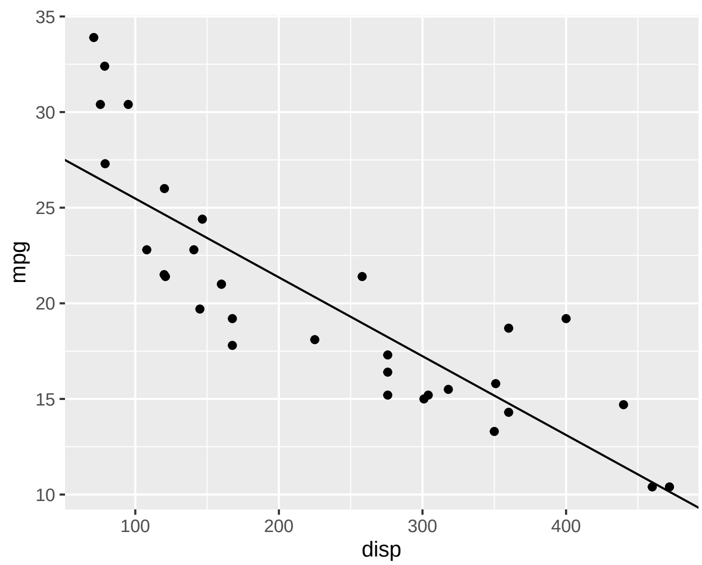
If you prefer to use functions from "ggplot2" to create your graphs, you can also plot a regression line in fairly straightforward manner.

```r
library(ggplot2)
```

4.2.1) Regression line with `geom_abline()`

The corresponding `abline()` function in "ggplot2" is `geom_abline()`. You need to specify the `slope` and the `intercept`

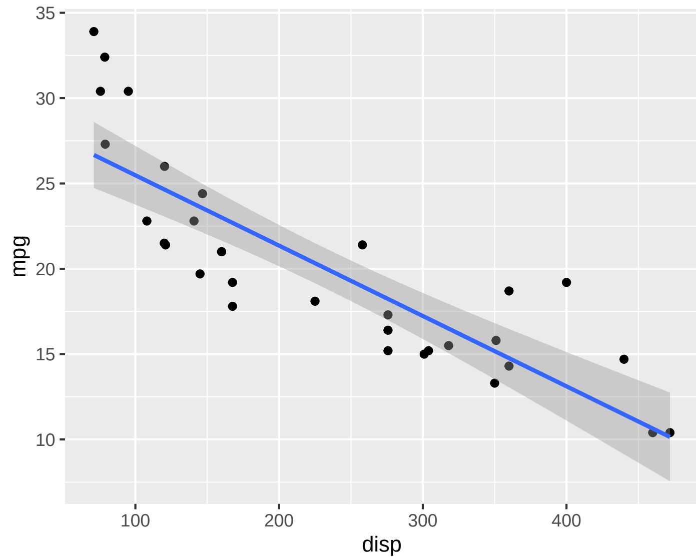
```r
ggplot(data = mtcars, aes(x = disp, y = mpg)) +
  geom_point() +
  geom_abline(slope = reg$coefficients[2], intercept = reg$coefficients[1])
```



4.2.2) Rgeression line with `stat_smooth()`

Another option to graph the line of a simple regression model with `ggplot()` is to use `stat_smooth()` instead of `geom_abline()`. In this case, we don't even have to use an object of class "lm": "ggplot2" will compute the regression output for us and use them to create the graphic.

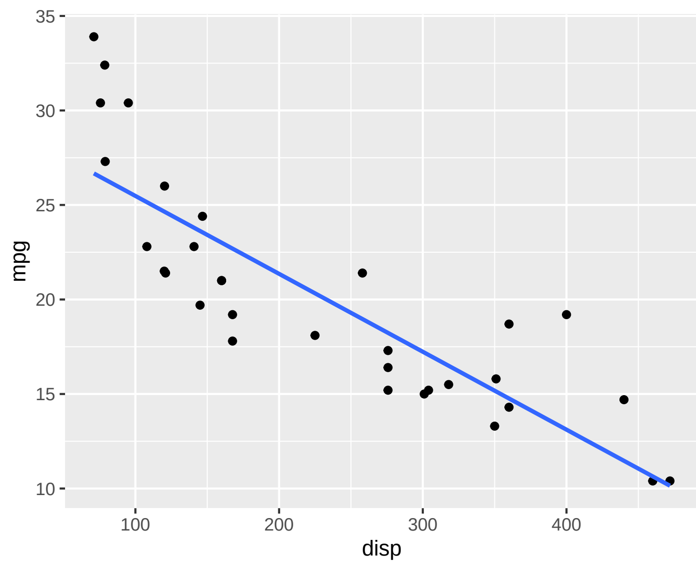
```r
ggplot(data = mtcars, aes(x = disp, y = mpg)) +
  geom_point() +
  stat_smooth(method = "lm")
```



When using `stat_smooth()` we need to set the argument `method = "lm"`. As you can tell, by default `stat_smooth()` displays the regression (in blue) but also a gray "ribbon" surrounding the regression line. This shaded region is the 95% confidence interval of the Standard Error.

To prevent the SE confidence region from being displayed we have to set the argument `se = FALSE`

```r
ggplot(data = mtcars, aes(x = disp, y = mpg)) +
  geom_point() +
  stat_smooth(method = "lm", se = FALSE)
```



5) Example: Multiple Linear Regression

Say we want to fit a **multiple linear model** in which *miles per gallon* (`mpg`) is regressed on *horsepower* (`hp`), *1/4 mile time* (`qsec`) and *weight (1000 lbs)* (`wt`), that is:

$$\texttt{mpg} = \beta_0 + \beta_1 \texttt{hp} + \beta_2 \texttt{qsec} + \beta_3 \texttt{wt} + \varepsilon$$

How can we specify such a model in R?

When the predictor(s) and the response variable are all in a single data frame, you can use `lm()` as follows:

```
# multiple linear regression
reg = lm(mpg ~ hp + qsec + wt, data = mtcars)
reg
```

```

Call:
lm(formula = mpg ~ hp + qsec + wt, data = mtcars)

Coefficients:
(Intercept)           hp         qsec           wt
   27.61053     -0.01782      0.51083     -4.35880
```

6) About Model Formulae

The `formula` declaration in `lm()` was originally introduced as a way to specify linear models, but have since been adopted for so many other purposes. An R formula has the general form:

$$\texttt{response} \sim \texttt{expression}$$

where the left-hand side, `response`, may in some uses be absent and the right-hand side, `expression`, is a collection of terms joined by operators usually resembling an arithmetical expression. The meaning of the right-hand side is context dependent.

The `formula` is interpreted in the context of the argument `data` which must be a list, usually a data frame; the objects named on either side of the formula are looked for first in `data`. If no data frame is provided, then R will search for the specified objects (i.e. variables) in the global environment. So, the following calls to `lm()` are equivalent:

```
# with argument 'data'
lm(mpg ~ disp + hp, data = mtcars)

# without argument 'data'
lm(mtcars$mpg ~ mtcars$disp + mtcars$hp)
```

6.1) The + (plus) operator

Notice that in these cases the `+` indicates *inclusion*, not addition. You can also use `-` which indicates *exclusion*.

As mentioned above, the formula expression `mpg ~ disp` corresponds to the linear model:

$$\texttt{mpg} = \beta_0 + \beta_1 \texttt{disp} + \varepsilon$$

6.2) The . (dot) operator

Another useful syntax when working with formulas is the dot `"."` character. Sometimes you will find the dot `.` as part of a formula declaration, for example:

```
# fit model with all available predictors
reg_all <- lm(mpg ~ ., data = mtcars)
```

The dot `"."` has a special meaning; in `lm()` it means *all the other variables* available in the object `data`. This is very convenient when your model has several variables, and typing all of them becomes tedious. The previous call is equivalent to:

```
# verbose: fit model with all available predictors
reg_all <- lm(mpg ~ cyl + disp + hp + drat + wt + qsec +
              vs + am + gear + carb, data = mtcars)
```

6.3) The : (colon) operator

Another feature of formulas is provided by the colon `":"` operator. This allows you to specify an **interaction term**.

For example, suppose we are interested in modeling `mpg` in terms of `cyl`, `disp`, and `hp`. But we also suspect that `cyl` interacts with `disp`. Algebraically, we could think of the following possible model:

$$\texttt{mpg} = \beta_0 + \beta_1 (\texttt{cyl} \times \texttt{disp}) + \beta_2 \texttt{hp} + \varepsilon$$

This is where `":"` comes handy. You use it to specify an interaction term between two variables in a formula object:

```
# fit model with an interaction term
reg_int1 <- lm(mpg ~ cyl:disp + hp, data = mtcars)
```

6.4) The * (product) operator

Related to the ":" interaction operator, R also provides the "*" operator to handle interactions between variables. The difference is that "*" will also produce individual terms.

For example, suppose we are interested in modeling `mpg` in terms of `cyl`, `disp`, and `hp`, suspecting an interaction bewteen `cyl` and `disp`. This time, though, the model of interest contains individual terms for both `cyl` and `disp`

$$\texttt{mpg} = \beta_0 + \beta_1\texttt{cyl} + \beta_2\texttt{disp} + +\beta_3(\texttt{cyl} \times \texttt{disp}) + \beta_4\texttt{hp} + \varepsilon$$

Instead of using : we should use * between `cyl` and `disp`:

```
# fit model with an interaction, and single terms
reg_int2 <- lm(mpg ~ cyl*disp + hp, data = mtcars)
```

6.5) The I() inhibit function

Another interesting operator is the *inhibit* function `I()`.

When used in a function formula, this operator inhibits the interpretation of operators such as "+", "-", "*" and "^" as formula operators, so they are used as arithmetical operators.

For example, consider the following quadratic model—in terms of `cyl`:

$$\texttt{mpg} = \beta_0 + \beta_1\texttt{cyl} + \beta_2\texttt{cyl}^2 + \varepsilon$$

You could try the following R formulas:

```
reg_qua1 <- lm(mpg ~ cyl + cyl^2, data = mtcars)

reg_qua2 <- lm(mpg ~ cyl + cyl*cyl, data = mtcars)
```

To avoid this confusion, the function `I()` can be used to bracket those portions of a model formula where the operators are used in their arithmetic sense.

In `reg_qua2`, the term `cyl*cyl` indicates an interaction between `cyl` and `cyl`. Which may not necessarily be waht we want to use. So, to tell R that * should be used as the arithmetic product operator instead of the formula interaction operator, we use `I()`:

```
reg_qua <- lm(mpg ~ cyl + I(cyl*cyl), data = mtcars)
```

or equivalently

```
reg_qua <- lm(mpg ~ cyl + I(cyl^2), data = mtcars)
```

7) summary() of "lm" objects

As with many objects in R, you can apply the function `summary()` to an object of class `"lm"`. This will provide, among other things, an extended display of the fitted model. Here's what the output of `summary()` looks like with our object `reg`:

```
# summary of an "lm" object
reg <- lm(mpg ~ disp + hp, data = mtcars)
reg_sum <- summary(reg)
reg_sum
```

```

Call:
lm(formula = mpg ~ disp + hp, data = mtcars)

Residuals:
    Min      1Q  Median      3Q     Max 
-4.7945 -2.3036 -0.8246  1.8582  6.9363 

Coefficients:
             Estimate Std. Error t value Pr(>|t|)    
(Intercept) 30.735904   1.331566  23.083  < 2e-16 ***
disp        -0.030346   0.007405  -4.098 0.000306 ***
hp          -0.024840   0.013385  -1.856 0.073679 .  
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 3.127 on 29 degrees of freedom
Multiple R-squared:  0.7482,	Adjusted R-squared:  0.7309 
F-statistic: 43.09 on 2 and 29 DF,  p-value: 2.062e-09
```

There's a lot going on in the output of `summary()`. So let's examine all the returned pieces.

7.1) Function `Call`

The first part of the output, `Call:`, corresponds to the command that we used to fit the model with `lm()`, in this case: `lm(formula = mpg ~ disp, data = mtcars)`.

7.2) Sumary statistics of `Residuals`

The second part, `Residuals:`, has the 5-number summary of the computed residuals:

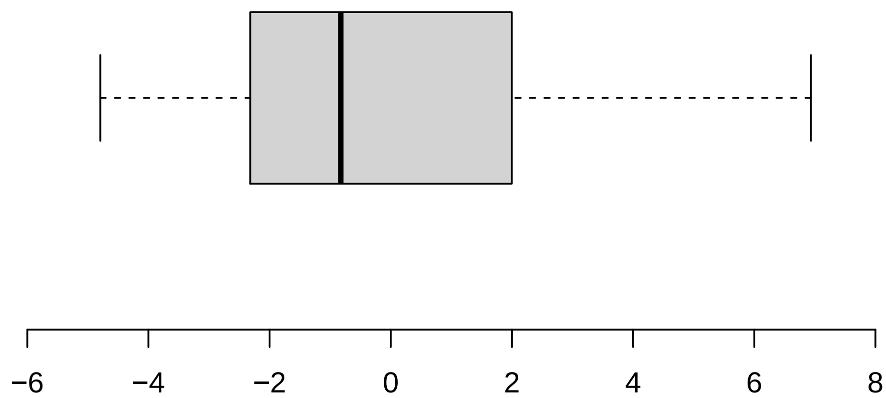
- minimum,
- 1st quartile,
- median,

- 3rd quartile,
- and maximum.

```
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max. 
-4.7945 -2.3036 -0.8246  0.0000  1.8582  6.9363 
```

In case you wonder, you can visualize this numeric summaries with a boxplot

```
# boxplot of residuals
boxplot(residuals(reg), horizontal = TRUE, ylim = c(-6, 8), axes = FALSE)
axis(side = 1)
```



7.3) Table of `Coefficients`

The 3rd part, `Coefficients`, corresponds to a table with five columns, and as many rows as coefficient estimates.

```
               Estimate  Std. Error   t value     Pr(>|t|)
(Intercept) 30.73590425 1.331566129 23.082522 3.262507e-20
disp        -0.03034628 0.007404856 -4.098159 3.062678e-04
hp          -0.02484008 0.013385499 -1.855746 7.367905e-02
```

- The first column has the names of the coefficients: `(Intercept)` and `disp`.
- The second column contains the estimated values.
- The third column has the standard error of the estimates.
- The fourth column has the t-statistic values.
- The fifth column corresponds to the p-values associated to t.

Notice all those asterisks next to the p-values. Both estimates are marked with three stars, indicating a p-value of less than 0.001.

These p-values correspond to tests of the (null) hypothesis:

$$H_0 : \beta_j = 0 \qquad j = 1, 2$$

under the assumptions of the classic linear model (i.e. linearity, normality, homoscedasticity, independence). We'll say more things about the table of coefficients shortly.

7.4) Additional statistics

The 4th and last part is comprised by the last three lines of text. Here there is also a lot going on.

```
Residual standard error: 3.127 on 29 degrees of freedom
Multiple R-squared:  0.7482,    Adjusted R-squared:  0.7309
F-statistic: 43.09 on 2 and 29 DF,  p-value: 2.062e-09
```

We have the following elements:

- Residual Standard Error (RSE) with its degrees of freedom
- Coefficient of determination R^2
- Adjusted R^2
- F-statistic with its degrees of freedom
- And p-value associated to the F-statistic.

Keep in mind that most elements of this 4th part have to do with inferential aspects of a classic linear model.

8) Predictions

In addition to the `summary.lm()` and `print.lm()` functions, `"lm"` objects also have an associated `predict.lm()` function.

Consider the following example of a multiple linear model in which *miles per gallon* (`mpg`) is regressed on *displacement* (`disp`) and *horse power* (`hp`) which would correspond to:

$$\texttt{mpg} = \beta_0 + \beta_1 \texttt{disp} + \beta_2 \texttt{hp} + \varepsilon$$

Here's the code to fit the model with `lm()`

```
# multiple linear regression
reg = lm(mpg ~ disp + hp, data = mtcars)
reg
```

```
Call:
lm(formula = mpg ~ disp + hp, data = mtcars)

Coefficients:
(Intercept)         disp           hp
   30.73590     -0.03035     -0.02484
```

8.1) `predict()` function

Suppose we wished to predict *future* consumption of miles per gallon `mpg`. The first step is to create a new data frame with a variables `disp` and `hp` containing the new values, for example:

```r
# data frame with new data
new_data <- data.frame(
  disp = 200,
  hp = 150,
  row.names = 'new car')

new_data
```

```
        disp  hp
new car  200 150
```

To obtain the predictions we call `predict()` and pass the `"lm"` object and the data frame for the argument `newdata`:

```r
predict(reg, newdata = new_data)
```

```
 new car
20.94064
```

The `predict()` method for `"lm"` objects works by attaching the estimated coefficients to a new model matrix that it constructs using the formula and the new data.

Now let's get predictions with more new observations:

```r
# data frame with more new observations
new_data <- data.frame(
  disp = seq(100, 200, by = 25),
  hp = seq(80, 120, by = 10),
  row.names = paste0('car_', letters[1:5]))

new_data
```

```
      disp  hp
car_a  100  80
car_b  125  90
car_c  150 100
car_d  175 110
car_e  200 120
```

and obtain the predicted `mpg`'s:

```
predict(reg, newdata = new_data)
```

```
   car_a    car_b    car_c    car_d    car_e 
25.71407 24.70701 23.69995 22.69290 21.68584 
```

9) Residual Analysis

As an example, let's take the data `mtcars` in order to regress `mpg` on `hp`

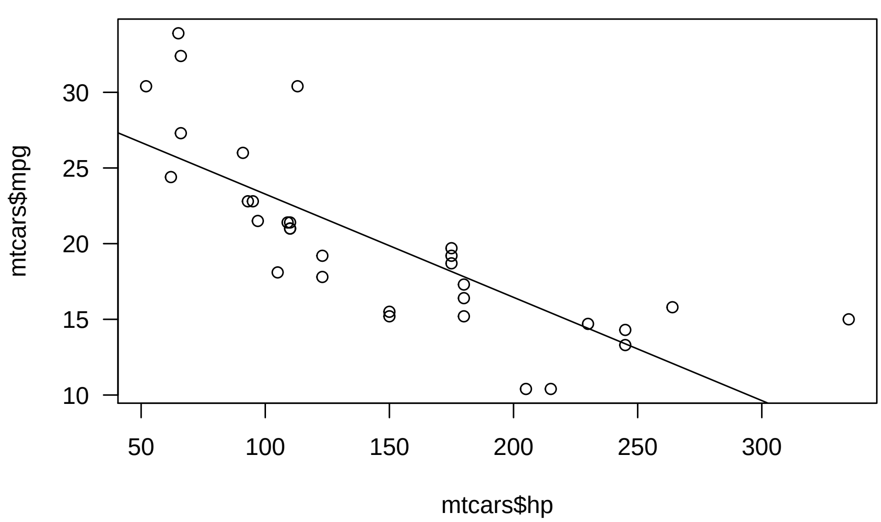
```
reg <- lm(mpg ~ hp, data = mtcars)
reg
```

```
Call:
lm(formula = mpg ~ hp, data = mtcars)

Coefficients:
(Intercept)           hp  
   30.09886     -0.06823  
```

Regression line

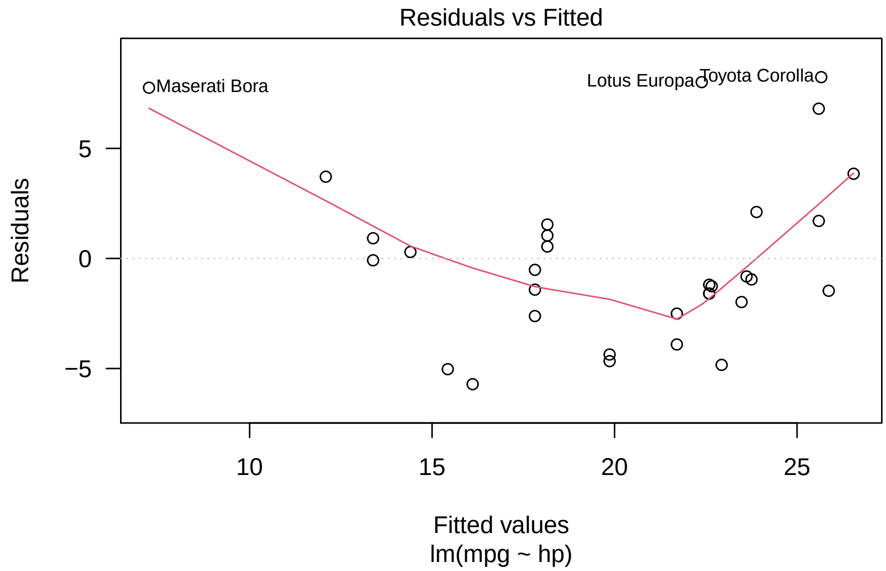
```
plot(mtcars$hp, mtcars$mpg, las = 1)
abline(reg)
```



9.1) Plot of Residuals

The plot below is the default residual plot provided in R when using the `plot()` method on an object of class `"lm"`, choosing the argument `which = 1`

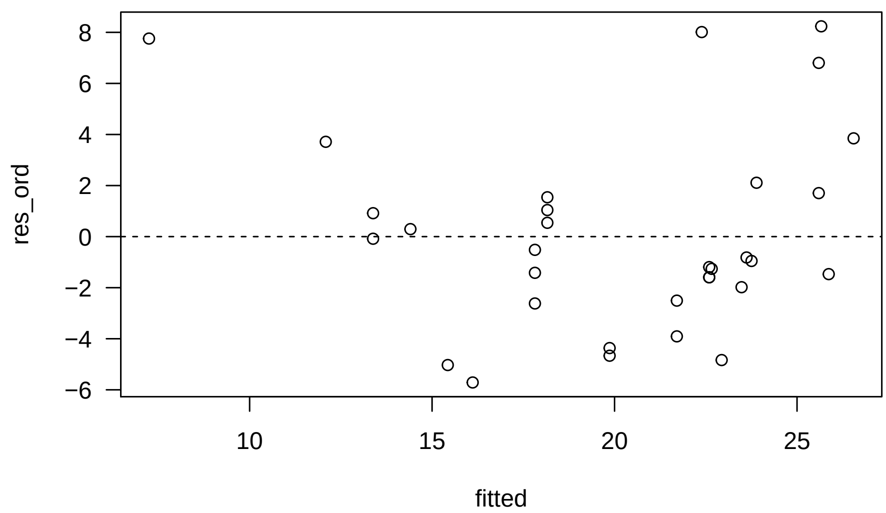
```
# plot of ordinary residuals -vs- fitted values
plot(reg, which = 1, las = 1)
```



You can obtain the same plot, by graphing residuals against fitted values.

```
hat_values <- hatvalues(reg)
fitted <- fitted(reg)
res_ord <- residuals(reg)
res_std <- rstandard(reg)
res_stu <- rstudent(reg)
```

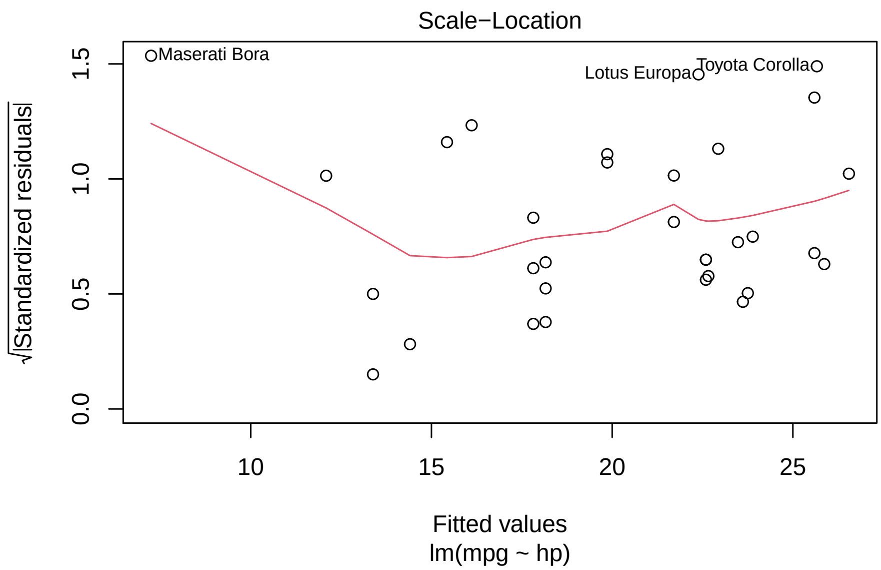
```
plot(fitted, res_ord, las = 1)
abline(h = 0, lty = 2)
```



9.2) Standardized Residuals

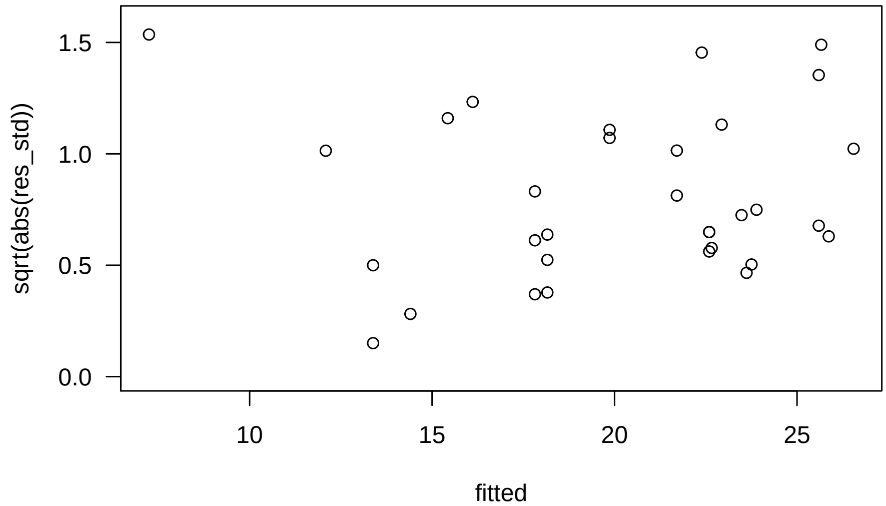
Another common plot is $|\text{Standardized Residuals}|^{1/2}$ against $\hat{y}$

```
# plot of (standardized residuals)^0.5 -vs- fitted values
plot(reg, which = 3)
```



The same plot can be obtained "by hand" like this

```r
plot(fitted, sqrt(abs(res_std)), las = 1, ylim = c(0, 1.6))
```



9.3) Q-Q Plots

The most common way to assess normality of the errors is to look at what is referred to as a *normal probability plot* or *quantile-comparison plot*, most commonly known as a **normal Q-Q plot** of the standardized residuals (for Quantile-Quantile plot). The idea of this plot is to compare the residuals to “ideal” normal observations.

9.3.1) Q-Q plot with standardized residuals

The typical Q-Q plot involves plotting the ordered standardized residuals on the vertical axis against the expected order statistics from a standard normal distribution $\mathcal{N}(0, 1)$ on the horizontal axis.

```r
# qqnorm(res_std, las = 1)
plot(reg, which = 2, las = 1)
```

9.3.2) Q-Q plot with studentized residuals

Another flavor of Q-Q plot involves using studentized residuals instead of standardized residuals. We compare the sample distribution of the **studentized** residuals

```r
qqnorm(rstudent(reg), las = 1, ylab = "Studentized Residuals")
qqline(rstudent(reg), lty = 3)
```

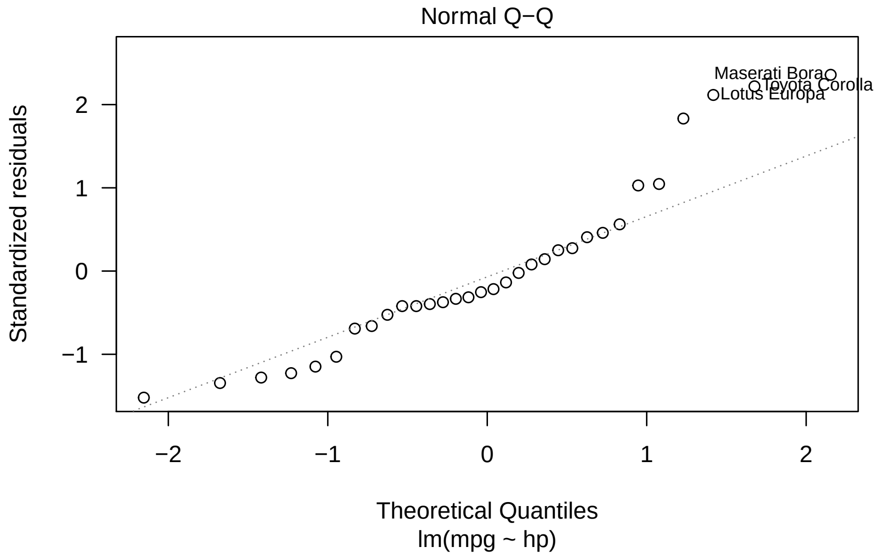


Figure 1: Q-Q plot of standardized residuals

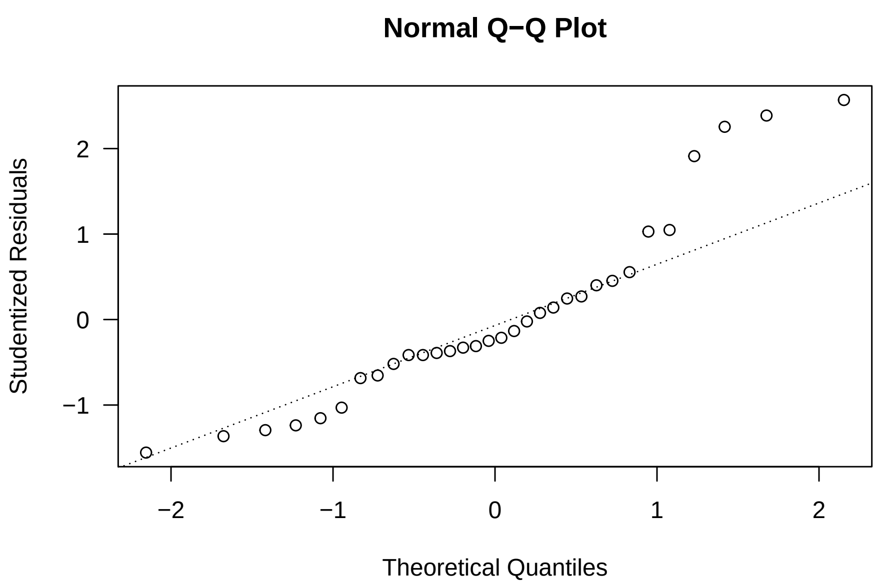


Figure 2: Q-Q plot of studentized residuals