Lab 4: Least Squares Regression

Prof. Gaston Sanchez

Stat 154, Fall 2017

Introduction

In this lab, we are going to review the function `lm()` for OLS regression. You will also have the opportunity (challenge?) to write code in R that computes OLS solution. We will focus on obtaining regression coefficients, fitted values (i.e. predicted values), and Residual Sum of Squares (RSS). We leave the inferential aspects for next lab.

Simple Regression

In R, the function that allows us 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, not just simple linear regression.

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)

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

```
# run regression analysis
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.

The example above is a simple linear regression (i.e. only one predictor). To fit a multiple linear model, just include more predictors:

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

Output of `lm()`

Let's consider a simple regression model in which `mpg` is regressed on `disp`:

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

We are storing the output of `lm()` in the object `reg`. Technically, `reg` is an object of class `"lm"`. Let's take a look at `reg`:

```
# default output
reg
```

```
##
## Call:
## lm(formula = mpg ~ disp, data = mtcars)
##
## Coefficients:
## (Intercept)         disp
##    29.59985     -0.04122
```

The first part of the output 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 shows information about the regression coefficients. The intercept is 29.6, and the other coeffcient 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()`:

```
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`. 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:

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

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

For the purposes and scope of this course, the most important output elements of an `"lm"` object are `coefficients`, `residuals`, and `fitted.values`.

About Model Formulae

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

```
response ~ 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` and then searched for in the usual way. 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)
```

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

Inside `lm()`, the formula expression `mpg ~ disp` corresponds to the linear model:

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

In vector-matrix notation, we would have a model expression like this:

$$\underset{n\times 1}{\mathbf{y}} = \underset{n\times 2}{\mathbf{X}} \times \underset{2\times 1}{\boldsymbol{\beta}} + \underset{n\times 1}{\boldsymbol{\varepsilon}}$$

When calling `lm(mpg ~ disp + hp, data = mtcars)`, R will create a **model matrix X** that would conceptually correspond to:

$$\begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ \vdots \\ y_n \end{bmatrix} = \begin{bmatrix} 1 & x_{11} \\ 1 & x_{21} \\ 1 & x_{31} \\ \vdots & \vdots \\ 1 & x_{n1} \end{bmatrix} \begin{bmatrix} \beta_0 \\ \beta_1 \end{bmatrix} + \begin{bmatrix} \varepsilon_1 \\ \varepsilon_2 \\ \varepsilon_3 \\ \vdots \\ \varepsilon_n \end{bmatrix}$$

In other words, R will include a column of ones for the intercept term.

When working with formulas, it is sometimes common to include a dot `.` as part of a formula declaration, for example:

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

The `.` has a special meaning; in `lm()` it means *all the other variables* available in `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)
```

Your turn

- Use `lm()` to regress `mpg` on `disp`. Store the output in an object called `reg1`.
- Now, mean-center the `mtcars` data.
- Re-run `lm()` regressing `mpg` on `disp` using the centered data. Store the output in an object called `reg2`.
- The intercept term that you obtained in `reg2` should be zero, since the used variables are mean-centered. How would you recover the intercept term?
- Now, standardize the `mtcars` data (mean = 0, var = 1).
- Re-run `lm()` regressing `mpg` on `disp` using the standardized data. Store the output in an object called `reg3`.
- From `reg3`, how would you recover the un-standardized $\hat{\beta}_0$ and $\hat{\beta}_1$—like those in `reg1`?
- Find out how to use `lm()` in order to exclude the intercept term β_0, that is, without a coefficient β_0

$$\text{mpg} = \beta_1 \text{disp} + \varepsilon$$

- `mtcars` has a column `am` for *automatic transmission*: 0 = automatic, 1 = manual. Find out how to use the argument `subset` of `lm()` to regress `mpg` on `disp`, with just those cars having automatic transmission.

Summary of an object `"lm"`

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

```
reg_sum <- summary(reg)
reg_sum
```

```
##
## Call:
## lm(formula = mpg ~ disp, data = mtcars)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -4.8922 -2.2022 -0.9631  1.6272  7.2305
##
## Coefficients:
##              Estimate Std. Error t value Pr(>|t|)
## (Intercept) 29.599855   1.229720  24.070  < 2e-16 ***
## disp        -0.041215   0.004712  -8.747 9.38e-10 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 3.251 on 30 degrees of freedom
## Multiple R-squared:  0.7183, Adjusted R-squared:  0.709
## F-statistic: 76.51 on 1 and 30 DF,  p-value: 9.38e-10
```

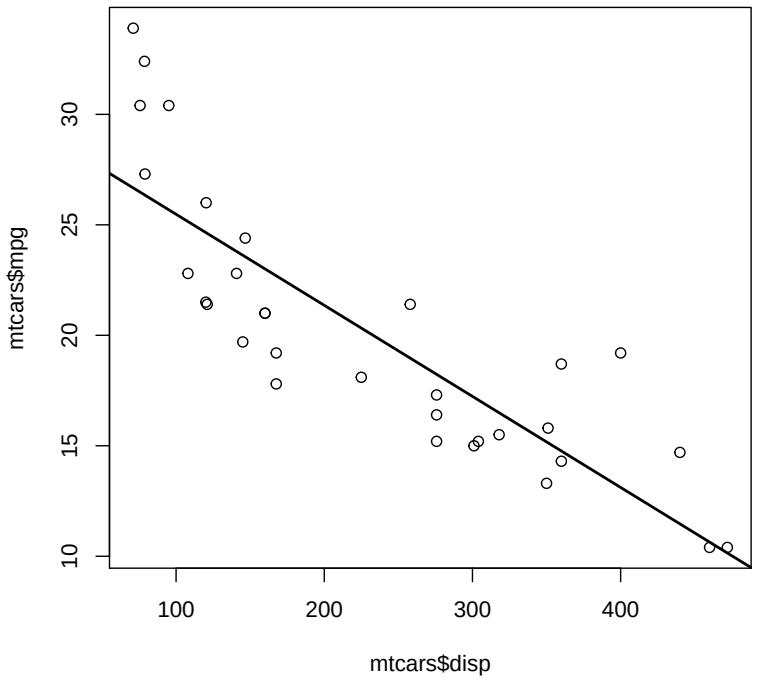
Your turn

- What class of object is `reg_sum`?
- What does `reg_sum` contain?

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. 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:

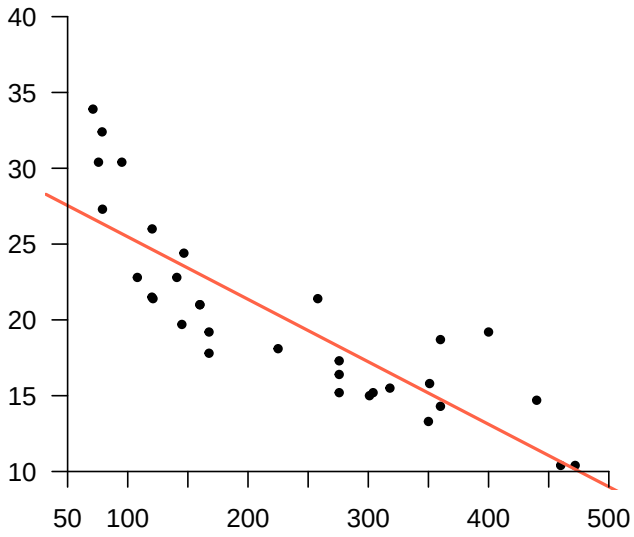
```
# scatterplot with regression line
plot(mtcars$disp, mtcars$mpg)
abline(reg, lwd = 2)
```



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.

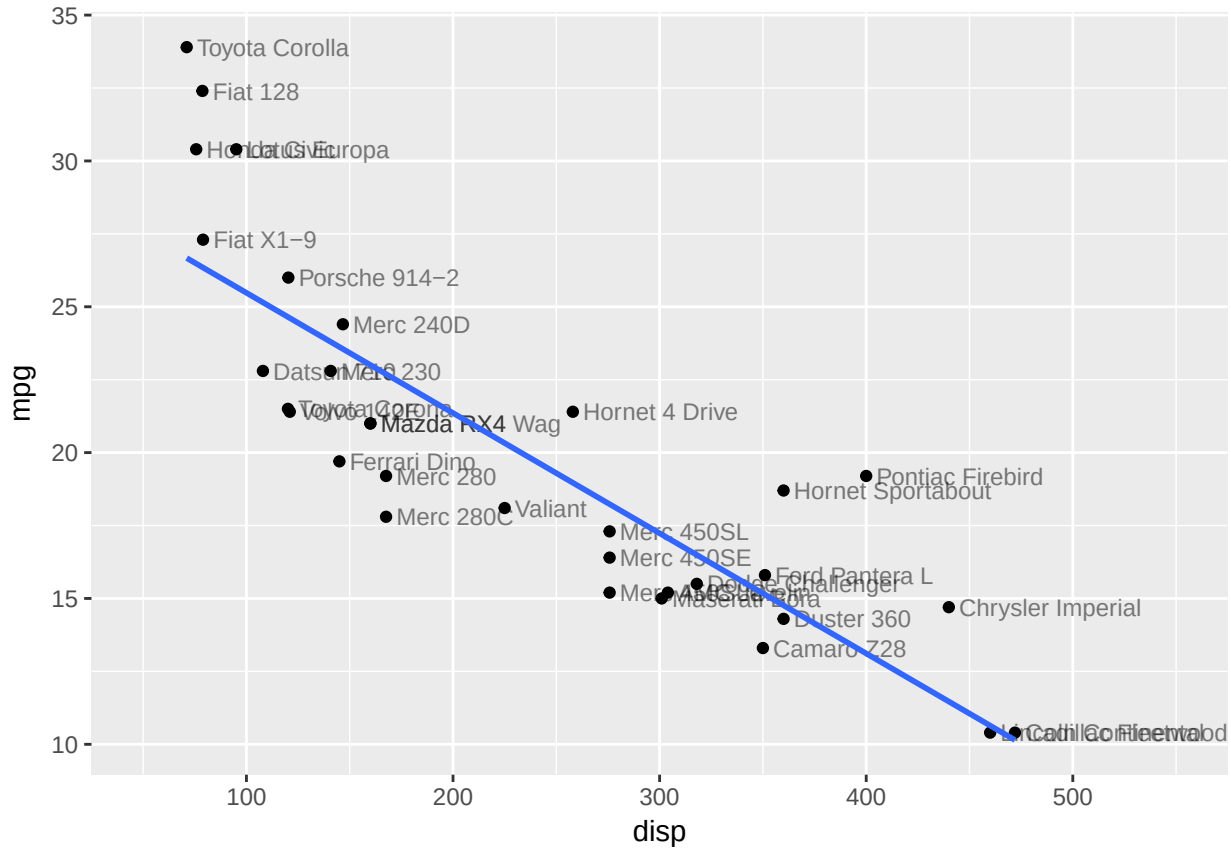
Here's how to get a nicer plot using low-level plotting functions:

```
# scatterplot with regression line
plot.new()
plot.window(xlim = c(50, 500), ylim = c(10, 40))
points(mtcars$disp, mtcars$mpg, pch = 20, cex = 1)
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))
```



Your turn

- Find out how to use functions in the graphcis package `"ggplot2"`, to get a scatterplot with the regression line—see the figure below.



Auxiliary plots

Model assumptions should be checked as far as is possible. The common checks are:

- **a plot of residuals versus fitted values**: for example there may be a pattern in the residuals that suggests that we should be fitting a curve rather than a line;
- **a normal probability**: if residuals are from a normal distribution points should lie, to within statistical error, close to a line.

Your turn: How do you get such plots? The easiest way is to apply `plot()` to the object `"lm"` object, and specify the argument `which`:

- residuals vs fitted model: `plot(reg, which = 1)`
- normal probability plot: `plot(reg, which = 2)`

Interpretation of these plots may require a certain amount of practice. This is specially the case for normal probability plots. Keep in mind that these diagnostic plots are not definitive. Rather, they draw attention to points that require further investigation.

More auxiliary diagnistic tools

Another auxiliar tool is the so-called *Analysis of variance table*, commonly referred to as the *anova* table.

```
reg_anova <- anova(reg)
reg_anova
```

```
## Analysis of Variance Table
##
## Response: mpg
##           Df Sum Sq Mean Sq F value   Pr(>F)
## disp       1 808.89  808.89  76.513 9.38e-10 ***
## Residuals 30 317.16   10.57
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

This table breask down the sum of squares about the mean, for the response variable, in two parts: a part that is accounted for by the deterministic component of the model, and a part attributed to the noise component or residual.

The total sum of squares (about the mean) for the 32 observations is 1126.0471875. Including the variable `disp` rediced this by 808.8884982, giving a residual sum of squares equal to 317.1586893.

This table has the information needed for calculating R^2, also known as the *coefficient of determination* and adjusted R^2. The R^2 statistic is the square of the correlation coefficient, and is the sum of squares due to `disp` divided by the total sum of squares: $R^2 = 0.7183$.

OLS solution

The vector of OLS estimates $\mathbf{b}$ is given by $(\mathbf{X}^\top\mathbf{X})^{-1}\mathbf{X}^\top\mathbf{y}$

```
# beta coefficients
XtXi <- solve(t(X) %*% X)
XtXi %*% t(x) %*% y
```

Although this works, computationally it is not the best way to compute $\mathbf{b}$. Why? Because it is inefficient and be very inaccurate when the predictors are strongly correlated.

A better, but not perfect, way is to use:

```
solve(crossprod(X, X), crossprod(X, y))
```

Here we get the same result as `lm()` because the data are well-behaved. In practice, I recommend to use packaged functions like `lm()` which uses QR decomposition.

QR Decomposition

Any matrix can be written as:

$$\mathbf{X} = \mathbf{QR}$$

where:

- $\mathbf{Q}$ is an $n \times n$ orthogonal matrix
- $\mathbf{R}$ is a $p \times p$ upper triangular matrix

```
QR <- qr(X)
```

To extract the matrices of the decomposition, you have to use the companion functions `qr.Q()` and `qr.R()`

```
Q <- qr.Q(QR)
R <- qr.R(QR)
```

```
f <- t(Q %*% y)
```

Solving $\mathbf{Rb} = \mathbf{f}$ we use the method of backsubstitution:

```
backsolve(R, f)
```