Week 7 Notes Part 2 Design Matrix

Code

Author: Brendan Gongol

Last update: 03 January, 2023

1 Overview

In this Note we introduce the concept of the "design matrix" of a linear model and run a few examples. We also showcase the use of lmFit()function from Bioconductor – a more flexible (at least in some contexts) alternative to lm(), which is also a convenient choice for running multiple linear fits at once.

2 Design Matrix

We have already mentioned before that "linear models" in statistics are linear with respect to the coefficients (parameters) of the dependence, not the data. For instance, $y = ax^2 + bx + c \cdot log(x) + d$ is still a linear model. The rationale for this seemingly counter-intuitive definition is that when we perform regression (fit a model), the data, x, are fixed. We have measured some realization of random variable, and as soon as we have done that, we are dealing with (fixed) numbers. We can square them or log-transform them, they are still numbers. It is the coefficients of the model that we need to solve for, and as long as the equations for those coefficients are linear, we are still in the realm of linear models.

It is instructive to delve a little deeper into how linear models are set up in general, and to discuss a fundamental notion of design matrix.

Suppose we have measured a pair of random variables X,Y and our measurements resulted in realization ("data points") $(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)$. We are fitting a linear model $Y \sim X$ to our data, i.e. $y = ax + b + \varepsilon$. This can be written point-by-point as a system of equations

(1)

$$y_i = ax_i + b + \epsilon_i$$

or

(2)

$$\left.\begin{aligned} y_1 &= ax_1 + b + \epsilon_1 \\ y_2 &= ax_2 + b + \epsilon_2 \end{aligned}\right\} \Rightarrow \begin{bmatrix} y_1 \\ y_2 \\ \ldots \end{bmatrix} = \begin{bmatrix} ax_1 + b \\ ax2 + b \\ \ldots \end{bmatrix} + \begin{bmatrix} \epsilon_1 \\ \epsilon_2 \\ \ldots \end{bmatrix} = \underbrace{\begin{bmatrix} 1 & x_1 \\ 1 & x_2 \\ \ldots & \end{bmatrix}}_{\text{design matrix}} \underbrace{\begin{bmatrix} b \\ a \end{bmatrix}}_{\text{coefficients}} + \begin{bmatrix} \epsilon_1 \\ \epsilon_2 \\ \ldots \end{bmatrix}$$

We are solving the system (2) for coefficients a,b (and our usual criterion is that a pair of values a,b is a solution when it minimizes the sum of squares of residuals, ε_i). The values $x_1, x_2, \ldots$ are fixed numbers. Equation (2) is a linear matrix equation for the model coefficients, and the (fixed numerical) matrix in the equation (2) is known as the design matrix of the model:

(3)

$$\begin{bmatrix} 1 & x_1 \\ 1 & x_2 \\ \ldots & \end{bmatrix}$$

You can easily see by writing it down in a matrix form similar to Eq. (2) that, for instance, for the model y=ax2+bx+c, the (linear) matrix equation for the vector of the coefficients (c,b,a) has the following design matrix:

(4)

$$\begin{matrix} c & b & a \end{matrix}$$
$$\begin{bmatrix} 1 & x_1 & x_1^2 \\ 1 & x_2 & x_2^2 \\ \ldots & \ldots & \ldots \end{bmatrix}$$

Why is the name "design matrix"? It comes from the case of categorical variable(s) (and now we can also finally link together linear models for continuous and categorical variables and understand why and in what sense these are indeed the same type of models using the same analytical and computational machinery). Categorical variables are often related to experimental design (e.g. case/control), hence the name.

Let us consider a linear model $Y \sim X$ with X being now a categorical variable with three levels, α_1, α_2, α_3. As we have discussed in conjunction with ANOVA on categorical variables (Week 5), we can represent different levels with indicators 0/1. The only difference with numerical variable is that categories do not have a scale. For instance, if we had a discreet numerical variable X that can take only values 1,2, 3, we could still try fitting a model y=ax+b, and if measured value of x changes, say, from 1 to 2 or from 1 to 3, then the corresponding (fitted) value of y changes by a or $2a$, respectively (because a change in x from 1 to 3 is twice as large as the change from 1 to 2). With categorical variables, we cannot tell "how far" one level is from the other, so we cannot encode all levels with different numbers (e.g. level 1 = 1, level 2 = 2, level 3 = 3) and use a single "slope" as above. Instead, the design matrix for a categorical linear model can look as follows (for the sake of example, we assume here that we have measured seven data points, and the values $x_1 \ldots x_7$ of X are $(\alpha_1, \alpha_1, \alpha_2, \alpha_2, \alpha_3, \alpha_3, \alpha_3)$:

$a_1 : 100$

$a_2 : 110$

$a_3 : 101$.

(5)

$$\begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \\ y_6 \\ y_7 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 1 & 0 & 0 \\ 1 & 1 & 0 \\ 1 & 1 & 0 \\ 1 & 0 & 1 \\ 1 & 0 & 1 \\ 1 & 0 & 1 \end{bmatrix} \begin{bmatrix} a_1 \\ a_2 \\ a_3 \end{bmatrix} + \begin{bmatrix} \epsilon_1 \\ \epsilon_2 \\ \epsilon_3 \\ \epsilon_4 \\ \epsilon_5 \\ \epsilon_6 \\ \epsilon_7 \end{bmatrix}$$

In the equation above, the fixed, measured values of X are encoded in the design matrix and a_i are the model coefficients we are solving for (so we can say that we have a separate slope ai for each level of the variable X). As you can see, in our example we chose to encode the level α_1 of X as 1,0,0 (first and second rows), and α_2 and α_3 are encoded as (1,1,0) and (1,0,1) respectively. The way we defined the design matrix (5), the model coefficients have the following interpretation: a_1 is the mean value of y when $x = \alpha_1$, a_2 is the difference between mean values of y in the groups $x = \alpha_2$ and $x = \alpha_1$ (so that $a_1 + a_2$ is mean value of y for $x = \alpha_2$), and a_3 is the difference between mean values of y in the groups $x = \alpha_3$ and $x = \alpha_1$ (so that $a_1 + a_3$ is mean value of y for $x = \alpha_3$). You can verify that this is the case by writing down all the equations, datapoint by datapoint (i.e. row by row).

When written down in the form of Eq. (5), the linear model with categorical variable(s) does not differ much from the continuous case (2): we have fixed, numerical design matrix set up in an appropriate way, and we solve a matrix linear equation for coefficients (although we do interpret coefficients in a different way). The design matrix in Eq.(5) is consistent with the examples and linear model fits we studied in Week 5 (patient's BCR/ABL fusion protein subtype). However, as we are going to see momentarily this is not the only possible way to set up a design matrix.

3 lmFit and Examples of Designs

Before looking further into the design matrices, let us download another Bioconductor package:

Hide

```
# if (!require("BiocManager", quietly = TRUE))
#     install.packages("BiocManager")
# BiocManager::install("limma")
library(limma)
```

The package 'limma' contains a number of useful functions, but the one we need right now is `lmFit()`. This function is not very different from familiar `lm()` function from the base package and provides an alternative implementation; it is more flexible, however, and there are some differences, which make it a more convenient choice in some situations. In particular, `lmFit()` can fit multiple models at once (e.g. days-to-remission against each vector of gene expression levels in our `exprs(ALL)` table), and does it more efficiently than an external loop we needed to run in order to fit days-to-remission to each individual gene one by one. Also, `lmFit()` can take a design matrix as an argument, which is useful for our little study here.

Consider the following code:

Hide

```
library(ALL); data(ALL)
fp.df <- data.frame(fp=pData(ALL)[,"fusion protein"],
 g1=exprs(ALL)["1970_s_at",],
 g2=exprs(ALL)["1002_f_at",])
fp.df <- fp.df[!is.na(fp.df$fp),]
table(fp.df$fp)
```

column contains categorical vs data

remove rows in fp column that are NA.

```
##
##      p190 p190/p210      p210
##        17         8         8
```

Hide

```
# try this! This is design matrix used by lm():
model.matrix(fp.df$g1~fp.df$fp)
```

```
##    (Intercept) fp.df$fpp190/p210 fp.df$fpp210
## 1            1                 0            1
## 2            1                 0            0
## 3            1                 0            0
## 4            1                 1            0
## 5            1                 0            0
## 6            1                 0            0
## 7            1                 0            1
## 8            1                 0            0
## 9            1                 1            0
## 10           1                 0            1
## 11           1                 0            0
## 12           1                 0            0
## 13           1                 1            0
## 14           1                 1            0
## 15           1                 0            0
## 16           1                 1            0
## 17           1                 0            1
## 18           1                 0            0
## 19           1                 0            0
## 20           1                 0            1
## 21           1                 1            0
## 22           1                 0            0
## 23           1                 0            0
## 24           1                 1            0
## 25           1                 0            0
## 26           1                 0            0
## 27           1                 0            1
## 28           1                 1            0
## 29           1                 0            1
## 30           1                 0            1
## 31           1                 0            0
## 32           1                 0            0
## 33           1                 0            0
## attr(,"assign")
## [1] 0 1 1
## attr(,"contrasts")
## attr(,"contrasts")$`fp.df$fp`
## [1] "contr.treatment"
```

Hide

```
# the following design has all 1 in the 1st column (p190),
# and an additional 1
# in corresponding cols for p190/p210, p210 (that's default for lm()),
# just the same way we defined design matrix for a 3-level variable in
# our example of Eq.(5) above:
design.lm <- cbind(p190=1,
 p190.p210=as.numeric(fp.df$fp == "p190/p210"),
 p210=as.numeric(fp.df$fp == "p210"))
expr.arrays <- matrix(c(fp.df$g1,fp.df$g2),nrow=2,ncol=33,byrow=T)
# fit result is the mean of p190, and offsets from that mean for
# means of other classes:
lmFit(expr.arrays,design.lm)$coefficients
```

Same result as model.matrix()

```
##          p190  p190.p210       p210
## [1,] 3.705230 0.36144084 0.30549958
## [2,] 3.792347 0.08649919 0.06915255
```

Hide

```
# the following design has 1 in corresponding columns ONLY,
# for each of p190,p190/p210,p210 levels:
design.1 <- cbind(p190=as.numeric(fp.df$fp == "p190"),
 p190.p210=as.numeric(fp.df$fp == "p190/p210"),
 p210=as.numeric(fp.df$fp == "p210"))
# fit results=corresponding means in each class:
lmFit(expr.arrays,design.1)$coefficients
```

```
##          p190 p190.p210    p210
## [1,] 3.705230  4.066671 4.01073
## [2,] 3.792347  3.878846 3.86150
```

In this code we revisit the problem first considered in Week 5: fitting gene expression against the (categorical) variable describing the BCR/ABL fusion protein status. First we pack fusion protein status variable from patient information table `pData(ALL)` alongside with expression levels of two genes into a data frame (purely for convenience). Next, we discard from that data frame all the patients that do not have fusion protein status information (these are missing data from the standpoint of our model anyway). NOTE: while it implements generic linear model, `lmFit()` was written with analysis of microarray data in mind (Bioconductor project!), hence the specific interface design. Also `lmFit()` would not even work with a single gene – it requires at least two (since the fits for each gene are still independent, this is just a bug or at least an oversight by the developers); hence we will use two genes to make `lmFit()` work.

The next command (table) should be familiar to you from previous weeks: it just shows the breakdown of cases (different levels) for the fusion protein status variable. Now the interesting part begins. In Week5 we used the standard function `lm()` in order to fit gene expression to the fusion status categorical variable. The function `model.matrix()` demonstrated in the code above takes the formula we would use for `lm()` and returns the design matrix for that model. The output is not shown in the code fragment (it's 33 lines!), but try this command in your R session and confirm that the design matrix has the form of Eq.(5), with 33 lines

instead of just 7: first column (for p190) is always 1, the second column (corresponding to the level p190.p210) also has 1 when fusion status is p190.p210 and 0 otherwise, and the third column has 1 when fusion status is p210.

We could use the design matrix returned by `model.matrix()` directly with `lmFit()` but in our code we chose to re-create it manually, for the sake of an exercise. Next line creates a matrix `design.lm`, which follows the same convention as described above.

The `lmFit()` command shown next takes as its arguments the following data:

- Matrix of expression levels for each gene we want to fit. The rows are genes, and the columns are "arrays" as they are called in the documentation, or, more generally, cases (in our data cases = patients). The naming convention comes from the notion of running an "array" (a microarray) in each condition/case/patient. Hence one column (multiple gene values)=one patient=one array. Each row is a set of measurements of a particular gene across multiple patients (arrays). This is exactly the format of our `exprs(ALL)` table, so we could fit all the genes at once if we wanted to, without much additional typing. Bioconductor is convenient and reasonably consistent as far asits interfaces and data structures are concerned.
- Design matrix. If we look at Eqs. (2), (5), we note that it is the design matrix that completely determines the right hand side of the equation. If we pass left hand side (values of y, which are gene expression levels in our case), and the design matrix, the equations (2), (5) can be solved for the unknown model coefficients a, b. Design matrix should have one row for each array (case, patient), and the order of those rows should be consistent with the order of columns (patients) in the gene expression data. The values in the design matrix correspond to the levels of the variable we are fitting against.

You can examine the coefficients of the fitted model and observe that the result (for the first gene) is exactly the same as the one we obtained with `lm()` in Week 5. With the choice of the design matrix we made (which corresponds to the default one used by `lm()`), the first coefficient of the fitted model is the mean gene expression in p190 cases, and the other two coefficients are offsets of the means in p190.p210 and p210 cases from that former mean. Since we are fitting multiple genes at once, the resulting fits are printed by row, one for each input gene.

design.1

Next, we are generating a different design matrix. In this case, each column (p190, p190.p210, p210) holds 1 only for the patients with the corresponding fusion type detected and 0 otherwise (print this design matrix in your R session and take a look at it). With this design, each fitted coefficient is now the mean expression level among the corresponding cases, not an offset (compare to Eq.(5) and make sure you understand!). We can see that this is the case by printing the coefficients of the model fitted with the new design matrix (last command in the code fragment).

If you are interested in more in-depth discussion, take a look here (http://www.statsci.org/smyth/pubs/limma-biocbook-reprint.pdf) (there is way more detail in that document than we will ever have time to even touch).

Here we will only mention two important "companions" to the lmFit(). The fit calculated with this function (or actually a collection of fits, since the function fits multiple genes at once against the same design matrix), can be passed to

- `eBayes(fit)`. This function improves and adjusts statistics for each individual gene's fit. Without going into too much detail, we only mention the general idea here. Namely, it was observed that genes with similar expression levels also exhibit comparable variance. Hence, in order to improve the

estimate of the variance for each gene (we need it to compute all kinds of statistics and p-values!), one can try pooling together variance from multiple genes with similar expression levels. This correction is clearly specific to microarray/gene expression data, and it is implemented in `eBayes()`.

- `topTable(fit)`. Selects top (most significant) fits from the collection of fits stored in the fit object. A convenience accessor function: it is possible to select most significant results (smallest p-values) manually, but with this function we do not need to worry about the structure of the object.