

R Formula Interface

and Model/Design Matrices

DS 6410 | Spring 2024

Rfmla.pdf

```
#-- Required Packages
library(splines)
library(tidyverse)
```

1 Raw input data

The raw input data is often in the form of a data frame (or tibble). For example,

```
#-- Raw Input Data
# cat is categorical with 3 levels: A,B,C
# num is numerical
# y is numerical response variable

Z = tibble(cat=c('A', 'A', 'B', 'B', 'C', 'C'), num=1:6, y=rnorm(6))
Z
#> # A tibble: 6 x 3
#>   cat      num      y
#>   <chr> <int> <dbl>
#> 1 A         1 -1.01
#> 2 A         2 -0.209
#> 3 B         3  0.262
#> 4 B         4  0.374
#> 5 C         5 -0.666
#> 6 C         6  0.914
```

has three columns, `cat` is categorical data, `num` which is numerical data, and `y` which is the outcome variable.

2 Formula in models

The formula interface in R allows you to make transformations of the input data frame automatically. For example, categorical (or factor) columns will generate the appropriate dummy variables.

```
lm(y~cat, data=Z)$coef
#> (Intercept)      catB      catC
#>   -0.6087    0.9266    0.7328
lm(y~cat - 1, data=Z)$coef # remove intercept
#>   catA    catB    catC
#> -0.6087  0.3179  0.1241
```

The default behavior is to convert categorical data to a *factor* and drop the first level.

The formula interface is easy to use:

```
#- numerical data only
lm(y~num, data=Z)$coef
#> (Intercept)      num
#>   -0.8908      0.2386

#- transformations
lm(y~log(num), data=Z)$coef
#> (Intercept)    log(num)
#>   -0.8509      0.7254

#- use I() to make custom functions
lm(y~I(3*num), data=Z)$coef
#> (Intercept)    I(3 * num)
#>   -0.89075      0.07954

#- we have already seen poly()
lm(y~poly(num, degree = 3), data=Z)$coef
#> (Intercept) poly(num, degree = 3)1 poly(num, degree = 3)2
#>   -0.05555      0.99826      -0.23329
#> poly(num, degree = 3)3
#>   0.92124

#- how about B-splines
library(splines)
lm(y~bs(num), data=Z)$coef
#> (Intercept)    bs(num)1    bs(num)2    bs(num)3
#>   -1.1227      3.3290      -0.8128      1.8798

#- two predictors
lm(y~cat + num, data=Z)$coef
#> (Intercept)    catB    catC    num
#>   -1.8536      -0.7332      -2.5869      0.8299
lm(y~cat + num - 1, data=Z)$coef
#>   catA    catB    catC    num
#> -1.8536 -2.5868 -4.4405  0.8299

#- a:b stands for interactions
lm(y~cat + num + cat:num, data=Z)$coef
#> (Intercept)    catB    catC    num    catB:num    catC:num
#>   -1.8066      1.7349      -6.7583    0.7986    -0.6873      0.7812

#- use . to represent everything in data
lm(y~., data=Z)$coef
#> (Intercept)    catB    catC    num
#>   -1.8536      -0.7332      -2.5869      0.8299
lm(y~. - num, data=Z)$coef # use . to include all, then remove some
#> (Intercept)    catB    catC
#>   -0.6087      0.9266      0.7328
```

2.1 model.matrix()

Behind the scenes, `lm()` is calling the function `model.matrix()` to construct the *model matrix* (also known as a *design matrix*). The model matrix is the real valued X matrix used for calculating the coefficients. You have to pass a formula object into `model.matrix()`.

```
fmla = formula(y~num+cat)
model.matrix(fmla, data=Z)
#> (Intercept) num catB catC
```

```
#> 1      1  1  0  0
#> 2      1  2  0  0
#> 3      1  3  1  0
#> 4      1  4  1  0
#> 5      1  5  0  1
#> 6      1  6  0  1
#> attr(,"assign")
#> [1] 0 1 2 2
#> attr(,"contrasts")
#> attr(,"contrasts")$cat
#> [1] "contr.treatment"

fmla = formula(y~num+cat-1) # remove intercept
model.matrix(fmla, data=Z)
#>   num catA catB catC
#> 1   1    1    0    0
#> 2   2    1    0    0
#> 3   3    0    1    0
#> 4   4    0    1    0
#> 5   5    0    0    1
#> 6   6    0    0    1
#> attr(,"assign")
#> [1] 1 2 2 2
#> attr(,"contrasts")
#> attr(,"contrasts")$cat
#> [1] "contr.treatment"
```

Or, if you are good with data manipulation construct the model matrix manually.

```
library(dplyr)
Z %>%
  transmute(
    intercept = 1,
    x1 = num,
    x2 = num^2,
    x3 = ifelse(cat=='B',1,0), x4=ifelse(cat=='C',1,0)
  ) %>%
  as.matrix()
#>   intercept x1 x2 x3 x4
#> [1,]      1  1  1  0  0
#> [2,]      1  2  4  0  0
#> [3,]      1  3  9  1  0
#> [4,]      1  4 16  1  0
#> [5,]      1  5 25  0  1
#> [6,]      1  6 36  0  1
```

Some functions (e.g., `glmnet`) do not take formulas so you will have to pass in the model matrix X directly. Another word of caution, some functions (again like `glmnet`) add the intercept automatically so you should not include a columns of ones.

The function `lm.fit()` fits a linear model from a model matrix:

```
X = model.matrix(formula(y~num+cat), data=Z)
Y = Z$y
lm.fit(x=X, y=Y)$coef
#> (Intercept)      num      catB      catC
#>   -1.8536    0.8299   -0.7332   -2.5869
```

2.2 Comparison

It is always good to compare the approaches just to make sure there are no mistakes.

```
fmla = formula(y~num+cat + I(num^2) + sqrt(num))

#- lm()
beta.lm = lm(fmla, data=Z)$coef

#- lm.fit()
X = model.matrix(fmla, data=Z)
beta.lmfit = lm.fit(X, Z$y)$coef

#- direct matrix operations
beta.eq = solve(t(X) %*% X) %*% t(X) %*% Z$y
# solve(crossprod(X), crossprod(X, Z$y)) # Alternative

#- output
tibble(beta.lm, beta.lmfit, beta.eq)
```

beta.lm	beta.lmfit	beta.eq
-14.1974	-14.1974	-14.1974
-10.9976	-10.9976	-10.9976
0.5629	0.5629	0.5629
-1.2136	-1.2136	-1.2136
0.6875	0.6875	0.6875
23.4996	23.4996	23.4996

3 Appendix: tidymodels

Using the recipe package (part of tidymodels), we can create model matrices.

```
library(tidymodels) # load tidymodels (and recipe package)

# create a `recipe`
rec = recipe(
  y ~ cat + num, # the formula specifies the variables (outcome and predictor)
  data = Z       # the data object provides the variables types
) %>%
  step_dummy(all_nominal_predictors())
```

Notes:

1. The formula specifies the variables (y is the outcome variable, cat and num are the predictor variables).
2. The data provides the variable types. The entire data isn't necessary, I could have used `head(Z)` to limit the amount of data passed around (this is only a concern for large data).
3. The `step_XX()` functions add transformations. In this case, we used `step_dummy()` to create dummy variables for *all nominal predictor variables*.

The last step requires `prep()` and `bake()`. The `prep()` step determines the number and name of the new dummy columns. The `bake()` function, which takes `new_data` will apply the transformations to the data. While this seems a bit verbose for this simple setting, the benefits will be more apparent when we start more complex modeling.

```
rec %>% prep() %>% bake(Z)
#> # A tibble: 6 x 4
#>   num      y cat_B cat_C
#>   <int> <dbl> <dbl> <dbl>
#> 1     1 -1.01     0     0
#> 2     2 -0.209    0     0
#> 3     3  0.262     1     0
#> 4     4  0.374     1     0
#> 5     5 -0.666     0     1
#> 6     6  0.914     0     1
```