Data frames: a data structure for statistical analysis

Matteo Fasiolo

Creating a Scoring Table

Imagine we wanted to create a “table” which stores data of different types:

```
student number, name, staff, score
1                teo,     T,  70
2               jack,     T,  80
3               emma,     F,  70
...
```

We cannot use `matrix` as it only allows single type data.

Lists can be used to store data of different types, can we use that?

```r
stu_no <- c(1L, 2L, 3L)
name <- c("teo", "jack", "emma")
staff <- c(T, T, F)
score <- c(70, 80, 70)
score_table <- list(no = stu_no, name = name,
                    staff = staff, score = score)
```

This works:

```r
score_table$name
```

```
[1] "teo"  "jack" "emma"
```

```r
score_table[[2]]
```

```
[1] "teo"  "jack" "emma"
```

```r
score_table$name[1]
```

```
[1] "teo"
```

But recall the data

```
student number, name, staff, score
1                teo,     T,  70
2               jack,     T,  80
3               emma,     F,  70
...
```

To extract all information on Emma we must do:

```
score_table[[1]][3]
```

```
[1] 3
```

```
score_table[[2]][3]
```

```
[1] "emma"
```

```
score_table[[3]][3]
```

```
[1] FALSE
```

(Exercise: can you do this more compactly with `lapply`?)

Data Frames

Point is that `list` does not directly represent tabular format.

A `data.frame` does:

```
score_table <- data.frame(stu_no, name, staff, score)
score_table
```

```
  stu_no name staff score
1      1  teo  TRUE    70
2      2 jack  TRUE    80
3      3 emma FALSE    70
```

Extract info about Emma:

```
score_table[3, ]
```

```
  stu_no name staff score
3      3 emma FALSE    70
```

So `data.frame` can be accesses like a matrix:

```r
score_table[3, 1]
```

```
[1] 3
```

```r
score_table[1:2, ]
```

```
  stu_no name staff score
1      1  teo  TRUE    70
2      2 jack  TRUE    80
```

We can also perform numeric operations on numeric columns:

```r
score_table[ , c(1, 4)] * 2 # Try score_table * 2
```

```
  stu_no score
1      2   140
2      4   160
3      6   140
```

Can also access a data.frame like a list:

```r
score_table$name
```

```
[1] "teo"  "jack" "emma"
```

```r
score_table[[2]]
```

```
[1] "teo"  "jack" "emma"
```

```r
score_table[c(1, 2)]
```

```
  stu_no name
1      1  teo
2      2 jack
3      3 emma
```

If can also mix matrix-like and list-like indexation:

```r
score_table[score_table$name == "emma", ]
```

```
  stu_no name staff score
3      3 emma FALSE    70
```

Using list-like access we add/remove columns (variables):

```
score_table$age <- c(23, 23, 21)
score_table
```

```
  stu_no name staff score age
1      1  teo  TRUE    70  23
2      2 jack  TRUE    80  23
3      3 emma FALSE    70  21
```

```
score_table$age <- NULL
```

Using matrix-like access we can remove or add:

```
( score_table <- score_table[-1, ] )
```

```
  stu_no name staff score
2      2 jack  TRUE    80
3      3 emma FALSE    70
```

```
score_table[3, ] <- list(1, "Jacob", TRUE, 60) #Try c()
```

Summarizing Data Frame

You can also get a summary of your data frame:

```
summary(score_table)
     stu_no         name             staff
 Min.   :1.0   Length:3           Mode :logical
 1st Qu.:1.5   Class :character   FALSE:1
 Median :2.0   Mode  :character   TRUE :2
 Mean   :2.0
 3rd Qu.:2.5
 Max.   :3.0
```

`data.frames` are a basic input for many modelling functions: `lm`, `glm`, `gam`, ...

Case Study: Predicting MNIST Digits.

Let us revisit the TB1 project and how data frame can be useful in this specific application.

We have the following matrices

- Images: $X \in \mathbb{R}^{60000 \times 784}$ ($\sqrt{784} = 28$)
- Labels: $Y \in \{1, 2, 3, 4, 5, 6, 7, 8, 9, 0\}^{60000 \times 1}$
- Testing Images: $TX \in \mathbb{R}^{10000 \times 784}$
- Testing Labels: $TY \in \{1, 2, 3, 4, 5, 6, 7, 8, 9, 0\}^{10000 \times 1}$

We can directly use these matrices within our program (as you did in the TB1 project).

However, by using data frame, we can organize them in a more meaningful way, making our program easily understandable.

Let us download the data

```r
library(dslabs)
# Need internet access!
# Do save(file = "mnist.RData", mnist)
# Then load("mnist.RData")
mnist <- read_mnist()
str(mnist)
```

```
List of 2
 $ train:List of 2
  ..$ images: int [1:60000, 1:784] 0 0 0 0 0 0 0 0 0 0 ...
  ..$ labels: int [1:60000] 5 0 4 1 9 2 1 3 1 4 ...
 $ test :List of 2
  ..$ images: int [1:10000, 1:784] 0 0 0 0 0 0 0 0 0 0 ...
  ..$ labels: int [1:10000] 7 2 1 0 4 1 4 9 5 9 ...
```

We rearrange the data and subsample:

```
sub <- 1:1000
train <- list("images"  = mnist$train$images[sub, ],
              "labels"  = mnist$train$labels[sub],
              "istest" = rep(FALSE, length(sub)))
test <- list("images"  = mnist$test$images[sub, ],
             "labels"  = mnist$test$labels[sub],
             "istest" = rep(TRUE, length(sub)))
```

```r
dat_mnist <- data.frame(
  "images" = I( rbind(train$images, test$images) ),
  "labels" = c(train$labels, test$labels),
  "istest" = c(train$istest, test$istest))

dim(dat_mnist)
```

```
[1] 2000    3
```

```r
dim(dat_mnist$images)
```

```
[1] 2000  784
```

```r
str(dat_mnist)
```

```
'data.frame':   2000 obs. of  3 variables:
 $ images: 'AsIs' int [1:2000, 1:784] 0 0 0 0 0 0 0 0 0 0 .
 $ labels: int  5 0 4 1 9 2 1 3 1 4 ...
 $ istest: logi  FALSE FALSE FALSE FALSE FALSE FALSE ...
```

The `I` function indicates we want the list elements stored in the data frame **as is**.

Without using `I` function, R will break the `images` matrix into its columns.

```
dat_wrong <- data.frame(
  "images" = rbind(train$images, test$images),
  "labels" = c(train$labels, test$labels),
  "istest" = c(train$istest, test$istest))

dim(dat_wrong)
```

```
[1] 2000  786
```

```
str(dat_wrong)
'data.frame':   2000 obs. of  786 variables:
 $ images.1  : int  0 0 0 0 0 0 0 0 0 0 ...
 $ images.2  : int  0 0 0 0 0 0 0 0 0 0 ...
 $ images.3  : int  0 0 0 0 0 0 0 0 0 0 ...
```

Visualizing Images

```
library(imager)

img <- matrix(dat_mnist$images[1, ], nrow = 28)
plot(as.cimg(img), axes = FALSE)
title(dat_mnist$labels[1])
```

For our convenience, let us add the image objects for each digit as a new column to the data frame, so we can easily plot them:

```
cvrt <- function(imgdata){
  img <- imgdata
  img <- matrix(img, nrow = 28, ncol = 28)
  return(as.cimg(img))
}

dat_mnist$pics <- apply(dat_mnist$image, 1, cvrt,
                        simplify = FALSE)
dim(dat_mnist) # now we have 4 columns.
```

```
[1] 2000    4
```

```
str(dat_mnist$pics[1:3])
```

```
List of 3
 $ : 'cimg' num [1:28, 1:28, 1, 1] 0 0 0 0 0 0 0 0 0 0 ...
 $ : 'cimg' num [1:28, 1:28, 1, 1] 0 0 0 0 0 0 0 0 0 0 ...
 $ : 'cimg' num [1:28, 1:28, 1, 1] 0 0 0 0 0 0 0 0 0 0 ...
```

Note the flexibility of the `data.frame` structure:

```
str(dat_mnist)

'data.frame':   2000 obs. of  4 variables:
 $ images: 'AsIs' int [1:2000, 1:784] 0 0 0 0 0 0 0 0 0 0
 $ labels: int  5 0 4 1 9 2 1 3 1 4 ...
 $ error : int  0 0 0 0 0 0 0 0 0 0 ...
 $ pics  :List of 2000
  ..$ : 'cimg' num [1:28, 1:28, 1, 1] 0 0 0 0 0 0 0 0 0 0
  ..$ : 'cimg' num [1:28, 1:28, 1, 1] 0 0 0 0 0 0 0 0 0 0
  ..$ : 'cimg' num [1:28, 1:28, 1, 1] 0 0 0 0 0 0 0 0 0 0
```

The columns are a matrix, two vectors and a list of graphical arrays!

Now each row identifies a digit.

To plot one we do:

```
plot(dat_mnist$pics[[5]], axes = FALSE,
     main = dat_mnist$labels[5])
```

9

Find Neighbours

First, let us write a function produces the 20 nearest neighbours who have the smallest distances to a test image

```
# returns indices of 20 nearest neighbours
find_nei <- function(a, dat){
  train <- dat[dat$istest == FALSE, ]
  n <- nrow(train$images)
  dist <- numeric(n)
  for(i in 1:n){
    dist[i] <- sqrt(sum((a-train$images[i, ])^2))
  }
  return(order(dist)[1:20])
}
```

Where note that

```
order(c(3.5, 1.1, 2))
```

```
[1] 2 3 1
```

Test our function

```r
kk <- 1000 + 2 # 2nd digit in test set
nei <- find_nei(dat_mnist$images[kk, ], dat = dat_mnist)
dat_mnist$labels[nei]
```

```
 [1] 2 2 8 3 2 6 3 3 2 2 2 2 6 1 6 3 1 6 2 1
```

```r
plot(dat_mnist$pics[[kk]], axes = FALSE,
     main = dat_mnist$labels[kk])
```

2

Making Predictions

We want to predict the label based on the majority of the neighbours' labels. For example, here:

```
dat_mnist$labels[nei]
```

```
 [1] 2 2 8 3 2 6 3 3 2 2 2 2 6 1 6 3 1 6 2 1
```

our prediction should be 2.

A useful command here is

```
( tab <- table(dat_mnist$labels[nei]) )
```

```

1 2 3 6 8
3 8 4 4 1
```

```
names( tab )
```

```
[1] "1" "2" "3" "6" "8"
```

We want the label corresponding to the largest count:

```
names(tab)[tab == max(tab)]
```

```
[1] "2"
```

So our prediction function is:

```
predict_lab <- function(a, dat){

  nei <- find_nei(a, dat)

  nei_labs <- dat$labels[nei]

  tab <- table(nei_labs)

  pred <- names(tab)[tab == max(tab)][1]

  return(pred)

}
```

Check if it works:

```r
predict_lab(dat_mnist$images[kk, ], dat = dat_mnist)
```

```
[1] "2"
```

Apply `predict` function to all images.

```r
dat_mnist$preds <- apply(dat_mnist$images, 1,
                         predict_lab, dat = dat_mnist)
```

Exercise: this takes a while, what is the order of complexity of the computation above?

Hint: how many pairs of images are we comparing and what is the complexity of each comparison?

Add error columns:

```
dat_mnist$error <- as.integer(dat_mnist$preds) -
                   as.integer(dat_mnist$labels)
```

```
train_err <- dat_mnist$error[dat_mnist$istest == FALSE]
test_err <- dat_mnist$error[dat_mnist$istest == TRUE]
table(train_err)
```

```
train_err
 -9  -8  -7  -6  -5  -4  -3  -2  -1   0   1   2   3   4   5
  2   2  12  13  13   2  13  24  20 849   7   8   3   5  19
```

```
table(test_err)
```

```
test_err
 -8  -7  -6  -5  -4  -3  -2  -1   0   1   2   3   4   5   6
  7   8  14  15   9   9  39  48 765  12  14   1  10  41   7
```

Mean absolute errors

```r
mean( abs( dat_mnist$error[dat_mnist$istest == FALSE] ) )
```

```
[1] 0.575
```

```r
mean( abs( dat_mnist$error[dat_mnist$istest == TRUE] ) )
```

```
[1] 0.799
```

We do better on the train than on the testing set, why?

Is there a (simple) solution to (nearly) equalise the performance?

Conclusion

A `data.frame` combines data of different types together, forms "datasets".

- Datasets are organized as "observations" and "columns (variables)".
- a `data.frame` can be accessed like a matrix or like a list.

A `data.frame` provides a clean and unified way of managing and accessing datasets in our program.

- Instead of managing several vectors, we only need to manage one data frame for each dataset.