P02 – Datenstrukturen

19. April 2021

Contents

Hinweise zur Abgabe:

Erstelle pro Aufgabe eine R-Code-Datei und benenne diese nach dem Schema `P<Woche>-<Aufgabe>.R` also hier `P02-1.R`, `P02-2.R`, `P02-3.R` und `P02-4.R` Schreibe den Code zur Lösung einer Aufgabe in die jeweilige Datei.

Es ist erlaubt (aber nicht verpflichtend) zu zweit abzugeben. Abgaben in Gruppen von drei oder mehr Personen sind nicht erlaubt. Diese Gruppierung gilt nur für die Abgabe der Programmierprobleme, nicht für die Live-Übungen.

Bei Abgaben zu zweit gibt nur eine der beiden Person ab. Dabei müssen in **jeder** abgegebenen Datei in der **ersten Zeile** als Kommentar **beide** Namen stehen also zB

```
# Ada Lovelace, Charles Babbage

1+1
# ...
```

Die Abgabe der einzelnen Dateien (kein Archiv wie `.zip`) erfolgt über Moodle im Element namens `P02`. Die Abgabe muss bis spätestens Sonntag, 25. April 2021, 23:59 erfolgen.

1 Testen der Struktur von Funktionsargumenten (15 Punkte)

Erweitere die Funktion `lsq()`, sodass ihre Argumente auf sinnvolle Struktur getestet werden.

```
lsq <- function(X, y) {
  # TODO
  A <- t(X) %*% X
  solve(A, t(X) %*% y)
}
```

Nutze dazu die Funktionen `stop()` oder `stopifnot()`.

- `stop("asdf")` beendet die Funktion und erzeugt die Fehlermeldung `asdf`.
- `stopifnot(expr)`: Falls `expr` nicht zu einem `logical` Vektor mit ausschließlich `TRUE`-Einträgen evaluiert, wird der Fehler `expr is not TRUE` erzeugt.
- Mit `stopifnot("asdf" = x > 0)` wird die Fehlermeldung auf `asdf` geändert.

```r
x <- c(-1, 1)
stopifnot(x > 0)
## Error: x > 0 are not all TRUE
stopifnot("x must be positive" = x > 0)
## Error: x must be positive
```

Achtung: Die Synatx `stopifnot("Fehlermeldung" = expr)` ist erst ab R Version 4 verfügbar.

Die Spezifikation für die Argumente `X` und `y` der Funktion `lsq()` lautet:

- Weder `X` noch `y` haben Länge 0.
- `X` ist eine numerische Matrix.
- `y` ist eine numerische Matrix mit einer Spalte oder ein numerischer atomarer Vektor.
- Weder `X` noch `y` enthalten `NA` Werte.
- Die Dimensionen von `X` und `y` sind kompatibel, sodass `t(X) %*% y` berechnet werden kann.
- `t(X) %*% X` ist invertierbar

"Numerisch" soll hier `integer` oder `double` bedeuten.

```r
lsq(matrix(1:6, nrow=3), 1:3)
##      [,1]
## [1,]    1
## [2,]    0
lsq(matrix(runif(6), nrow=3), matrix(runif(3), ncol=1))
##             [,1]
## [1,] -0.01111489
## [2,]  0.53856952
lsq(matrix(letters[1:6] , nrow=2), 1:3)
## Error in lsq(matrix(letters[1:6], nrow = 2), 1:3): X must be numeric
lsq(matrix(1:6, nrow=3), list(1,2,3))
## Error in lsq(matrix(1:6, nrow = 3), list(1, 2, 3)): y must be numeric
lsq(1:6, 1:3)
## Error in lsq(1:6, 1:3): X must be matrix
lsq(matrix(1:6, nrow=3), array(1:3, dim=c(1,1,3)))
## Error in lsq(matrix(1:6, nrow = 3), array(1:3, dim = c(1, 1, 3))): y must be matrix or vector
lsq(matrix(1:6, nrow=3), 1:4)
## Error in lsq(matrix(1:6, nrow = 3), 1:4): dimensions of X and y do not fit
lsq(matrix(1:6, nrow=3), matrix(1:3, nrow=1))
## Error in lsq(matrix(1:6, nrow = 3), matrix(1:3, nrow = 1)): dimensions of X and y do not fit
lsq(matrix(1:6, nrow=3), matrix(1:6, nrow=3))
## Error in lsq(matrix(1:6, nrow = 3), matrix(1:6, nrow = 3)): y is not allowed to have more than 1 col
lsq(matrix(double(0), nrow=0, ncol=0), matrix(double(0), nrow=0, ncol=0))
## Error in lsq(matrix(double(0), nrow = 0, ncol = 0), matrix(double(0), : y is not allowed to have mor
lsq(matrix(1:6, nrow=3), c(1,NA,3))
## Error in lsq(matrix(1:6, nrow = 3), c(1, NA, 3)): y may not contain NA
lsq(matrix(c(1:5, NA), nrow=3), 1:3)
## Error in lsq(matrix(c(1:5, NA), nrow = 3), 1:3): X may not contain NA
lsq(matrix(c(1,1,2,1,1,2), nrow=3), 1:3)
## Error in lsq(matrix(c(1, 1, 2, 1, 1, 2), nrow = 3), 1:3): det(t(X) %*% X) must not be 0
```

2 matrix() (15 Punkte)

Schreibe eine Funktion `my_matrix()`, die das Verhalten von `matrix()` (in Teilen) imitiert. Die Funktionen `matrix()`, `rownames()`, `colnames()` dürfen dabei nicht verwendet werden.

```r
my_matrix <- function(vec, nrow=NULL, ncol=NULL, colnames=NULL, rownames=NULL) {
  # TODO
}
```

`vec` ist der Vektor der Einträge der Matrix.

Die Argumente `nrow` und `ncol` sollen sich verhalten, wie bei `matrix()`, dh es muss nur eines der beiden Argumente angegeben werden. Beide können auch angegeben werden, wenn die Länge von `vec` gleich 1 ist oder dem Produkt von `nrow` und `ncol` entspricht.

Falls entweder `colnames` oder `rownames` oder beide gesetzt sind, soll die Matrix entsprechende Spalten oder Zeilennamen bekommen, wobei die Dimension der Zeilen den Namen `rows` und die Dimension der Spalten den Namen `columns` bekommt.

Teste alle Argumente auf kompatible Länge bzw Werte mit `stopifnot()`, siehe Aufgabe 1. Die Fehlerausgaben müssen nicht umbenannt werden. Auf korrekten Typ oder Klasse muss nicht getestet werden.

Der Rückgabewert ist eine Matrix der entsprechenden Dimension und ggf mit Zeilen- und Spaltennamen.

```r
my_matrix(1:6)
## Error in my_matrix(1:6): at least one of nrow, ncol has to be specified
my_matrix(1:6, ncol=1)
##      [,1]
## [1,]    1
## [2,]    2
## [3,]    3
## [4,]    4
## [5,]    5
## [6,]    6
my_matrix(1:6, ncol=2)
##      [,1] [,2]
## [1,]    1    4
## [2,]    2    5
## [3,]    3    6
my_matrix(1:6, ncol=3)
##      [,1] [,2] [,3]
## [1,]    1    3    5
## [2,]    2    4    6
my_matrix(1:6, ncol=6)
##      [,1] [,2] [,3] [,4] [,5] [,6]
## [1,]    1    2    3    4    5    6
my_matrix(1:6, ncol=4)
## Error in my_matrix(1:6, ncol = 4): incompatible length
my_matrix(1:6, nrow=2)
##      [,1] [,2] [,3]
## [1,]    1    3    5
## [2,]    2    4    6
my_matrix(1:6, nrow=7)
## Error in my_matrix(1:6, nrow = 7): incompatible length
my_matrix(1:6, ncol=2, nrow=2)
## Error in my_matrix(1:6, ncol = 2, nrow = 2): incompatible length
my_matrix(1:6, ncol=2, nrow=3)
##      [,1] [,2]
## [1,]    1    4
## [2,]    2    5
## [3,]    3    6
```

```r
my_matrix(1:6, ncol=2, nrow=1)
## Error in my_matrix(1:6, ncol = 2, nrow = 1): incompatible length
my_matrix(0, ncol=3, nrow=2)
##      [,1] [,2] [,3]
## [1,]    0    0    0
## [2,]    0    0    0
my_matrix(1:6, ncol=3, colnames=LETTERS[1:3])
##       columns
##        A B C
##   [1,] 1 3 5
##   [2,] 2 4 6
my_matrix(1:6, ncol=3, colnames=LETTERS[1:2])
## Error in my_matrix(1:6, ncol = 3, colnames = LETTERS[1:2]): length of colnames must be ncol
my_matrix(1:6, ncol=3, rownames=letters[24 + 1:2])
##
## rows [,1] [,2] [,3]
##    y    1    3    5
##    z    2    4    6
my_matrix(1:6, ncol=3, colnames=LETTERS[1:3], rownames=letters[24 + 1:2])
##     columns
## rows A B C
##    y 1 3 5
##    z 2 4 6
```

Bemerkung: Die Funktion `matrix()` hat keine Argumente `colnames` oder `rownames`, aber das Argument `dimnames`. Siehe auch `?colnames`.

3 factor() und tibble() (15 Punkte)

Schreibe die Funktionen `my_tibble()` und `my_factor()`, die analog zu `tibble::tibble()` und `factor()` Tibbles bzw Faktoren erzeugen, ohne die Originale zu nutzen (auch `data.frame()` soll nicht genutzt werden).

Das Testen der Argumente auf kompatible Struktur ist für diese Aufgabe nicht nötig.

`my_tibble()` hat ein Argument namens `data`, welches eine Liste der Spalten des Ausgabe-Tibbles ist.

`my_factor()` hat ein Argument namens `data`. Wir nehmen an, dass `data` ein `character`-Vektor ist.

```r
my_tibble <- function(data) {
  # TODO
}

my_factor <- function(data) {
  # TODO
}
```

```r
library(tibble) # nur zur Ausgabe auf Konsole
my_tb <- my_tibble(list(x=1:3, y=letters[1:3]))
my_tb
## # A tibble: 3 x 2
##       x y
## * <int> <chr>
## 1     1 a
## 2     2 b
## 3     3 c
```

```r
tb <- tibble(x=1:3, y=letters[1:3])
identical(tb, my_tb)
## [1] TRUE


my_fac <- my_factor(c("a", "b", "a", "a", "c", "c"))
my_fac
## [1] a b a a c c
## Levels: a b c
fac <- factor(c("a", "b", "a", "a", "c", "c"))
identical(fac, my_fac)
## [1] TRUE
```

4 Collatz-Problem (15 Punkte)

Die Collatz-Sequenz beginnend mit $x_1 \in \mathbb{N}$ ist definiert durch

$$x_{i+1} := \begin{cases} x_i/2 & \text{falls } x_i \text{ gerade,} \\ 3x_i + 1 & \text{falls } x_i \text{ ungerade.} \end{cases}$$

Die Collatz-Vermutung besagt, dass diese Folge irgendwann $x_n = 1$ erreicht.

Schreibe eine Funktion `collatz()` zur Berechnung von Collatz-Sequenzen.

`collatz()` hat zwei Argumente: `x` und `max_iter`.

`x` ist der Startwert x_1. Falls `x` nicht zu einem einzelnen `integer`-Wert gewandelt werden kann, wird eine Fehlermeldung ausgegeben.

`max_iter` gibt die maximale Länge der Ausgabe-Sequenz an. Würde diese überschritten werden, bricht die Funktion ab.

Der Rückgabewert ist eine Liste mit zwei Elementen. Das erste Element trägt den Namen `seq` und enthält einen `integer`-Vektor der Collatz-Sequenz $x_1, \ldots, x_n = 1$ oder $x_1, \ldots, x_m$ mit m gleich `max_iter` falls $n > m$. Das zweite Element trägt den Namen `len` und ist ein einzelner `integer`. Der Wert ist n, falls $n \leq m$, sonst der `NA`-Wert.

```r
collatz <- function(x, max_iter) {
  # TODO
}
```

```r
str(collatz(1, 1e4))
## List of 2
##  $ seq: int 1
##  $ len: int 1
str(collatz(2, 1e4))
## List of 2
##  $ seq: int [1:2] 2 1
##  $ len: int 2
str(collatz(3, 1e4))
## List of 2
##  $ seq: int [1:8] 3 10 5 16 8 4 2 1
##  $ len: int 8
str(collatz(3, 5))
## List of 2
##  $ seq: int [1:5] 3 10 5 16 8
```

```
##  $ len: int NA
str(collatz("4", 1e4))
## List of 2
##  $ seq: int [1:3] 4 2 1
##  $ len: int 3
str(collatz("four", 1e4))
## Warning in collatz("four", 10000): NAs introduced by coercion
## Error in collatz("four", 10000): Cannot interprete x as integer
str(collatz(1:5, 1e4))
## Error in collatz(1:5, 10000): Length of x must be 1
str(collatz(5.0, 1e4))
## List of 2
##  $ seq: int [1:6] 5 16 8 4 2 1
##  $ len: int 6
str(collatz(5.1, 1e4))
## List of 2
##  $ seq: int [1:6] 5 16 8 4 2 1
##  $ len: int 6
str(collatz(5.9, 1e4))
## List of 2
##  $ seq: int [1:6] 5 16 8 4 2 1
##  $ len: int 6
```