Operace nad prvky vektorů, seznamů a matic

Michal Kvasnička

Často je potřeba provést nějakou operaci s každým prvkem vektoru, seznamu, matice apod. Většina programovacích jazyků k tomu používá cykly. R však nabízí lepší možnosti: mnohé funkce jsou v R vektorizované, tj. pracují s každým prvkem vektoru zvlášť (např. funkce `sin()`). V ostatních případech nabízí R funkce typu `apply()`. Jedná se o funkcionály, tj. funkce, které jako vstup berou jiné funkce (a data).

Operace nad vektory

Základní funkce

Základní funkce pro provádění operací nad vektory je funkce `lapply(x, f, ...)`. Její první parametr je vektor (atomický nebo seznam) a druhý parametr je funkce. Funkce `lapply()` spustí funkci `f` na každý prvek vektoru `x` a výsledky poskládá do seznamu. Prvek vektoru `x` je vždy prvním parametrem funkce `f`; další parametry zadané do `lapply()` se předají jako další parametry funkci `f`. Funkce `f` nemusí být pojmenovaná.

Příklady:

Chceme zjistit délku jednotlivých vektorů v seznamu:

```
v <- list(1, 1:2, 1:3, 1:4, 1:5)
lapply(v, length)
```

```
## [[1]]
## [1] 1
##
## [[2]]
## [1] 2
##
## [[3]]
## [1] 3
##
## [[4]]
## [1] 4
##
## [[5]]
## [1] 5
```

Chceme zjistit, jaký datový typ mají jednotlivé sloupce datasetu (data.frame je zároveň tabulka i seznam; funkce `lapply()` s ním zachází jako se seznamem sloupců):

```
df <- data.frame(x = 1:6,
                 y = c(rnorm(1:5), NA),
                 z = c(NA, letters[1:4], NA),
                 stringsAsFactors = FALSE)
df
```

```
##   x           y    z
## 1 1  0.02208284 <NA>
## 2 2 -1.94408728    a
## 3 3  0.60180021    b
## 4 4  0.56831947    c
```

```
## 5 5 -1.87997891    d
## 6 6          NA <NA>
```

```
lapply(df, class)
```

```
## $x
## [1] "integer"
##
## $y
## [1] "numeric"
##
## $z
## [1] "character"
```

Chceme zjistit, kolik obsahuje každý sloupec datasetu hodnot `NA`:

```
lapply(df, function(x) sum(is.na(x)))  # používá anonymní funkci
```

```
## $x
## [1] 0
##
## $y
## [1] 1
##
## $z
## [1] 2
```

Chceme vytvořit několik vektorů náhodných čísel o různých délkách:

```
v1 <- lapply(1:5, rnorm)  # normální rozložení, mean = 0, sd = 1
v1
```

```
## [[1]]
## [1] -0.6784571
##
## [[2]]
## [1] -0.3936059  2.2730785
##
## [[3]]
## [1]  0.7768281 -1.2030095 -0.4940104
##
## [[4]]
## [1]  1.10492478 -0.17638848  0.43992224 -0.01932104
##
## [[5]]
## [1] -0.11315327 -0.67668375 -1.25424193 -0.24910243  0.08951088
```

```
v2 <- lapply(1:5, rnorm, mean = 10, sd = 10)  # normální rozložení, mean = 10, sd = 10
                                              # extra parametry předány do rnorm()
v2
```

```
## [[1]]
## [1] 13.24493
##
## [[2]]
```

```
## [1] 6.306235 5.275554
##
## [[3]]
## [1] -1.7012568  0.8480395 18.1188554
##
## [[4]]
## [1] -3.155310 18.963901  1.943644 13.198381
##
## [[5]]
## [1]  35.694246  15.889956   4.969437 -10.896509  -5.530693
```

Zjednodušení výsledku

Často nechceme výsledek jako seznam, ale chceme jej zjednodušit na vektor nebo matici. K tomu slouží dvě funkce: `sapply()` a `vapply()`. Funkce `sapply(x, f, ...)` nejdříve spustí `lapply()`, a pak se pokusí její výsledek zjednodušit:

- pokud je výsledkem `lapply()` seznam vektorů o délce 1, vrátí vektor
- pokud je výsledkem seznam vektorů o stejných délkách, ale vyšších než 1, vrátí matici
- jinak vrátí seznam

Počet hodnot `NA` v datasetu tak zjistíme jako (názvy hodnot odpovídají jménům sloupců datasetu):

```
sapply(df, function(x) sum(is.na(x)))  # používá anonymní funkci
```

```
## x y z
## 0 1 2
```

Funkce `sapply()` se dobře hodí pro interaktivní práci; pro psaní skriptů je poněkud nebezpečná, protože pokud nedokáže seznam zjednodušit na vektor nebo matici, bez varování vrátí seznam. Proto je ve skriptech bezpečnější použít funkci `vapply(x, f, fv, ...)`, která dělá totéž co `sapply()`, ale navíc umožňuje zadat datový typ výsledku. Typ výsledku se zadává jako vektor příkladů (tj. celé číslo např. jako `1L`, reálné číslo jako `numeric(1)` nebo 1.0 apod.) Pokud se konverze nezdaří, vydá funkce `vapply()` chybové hlášení.

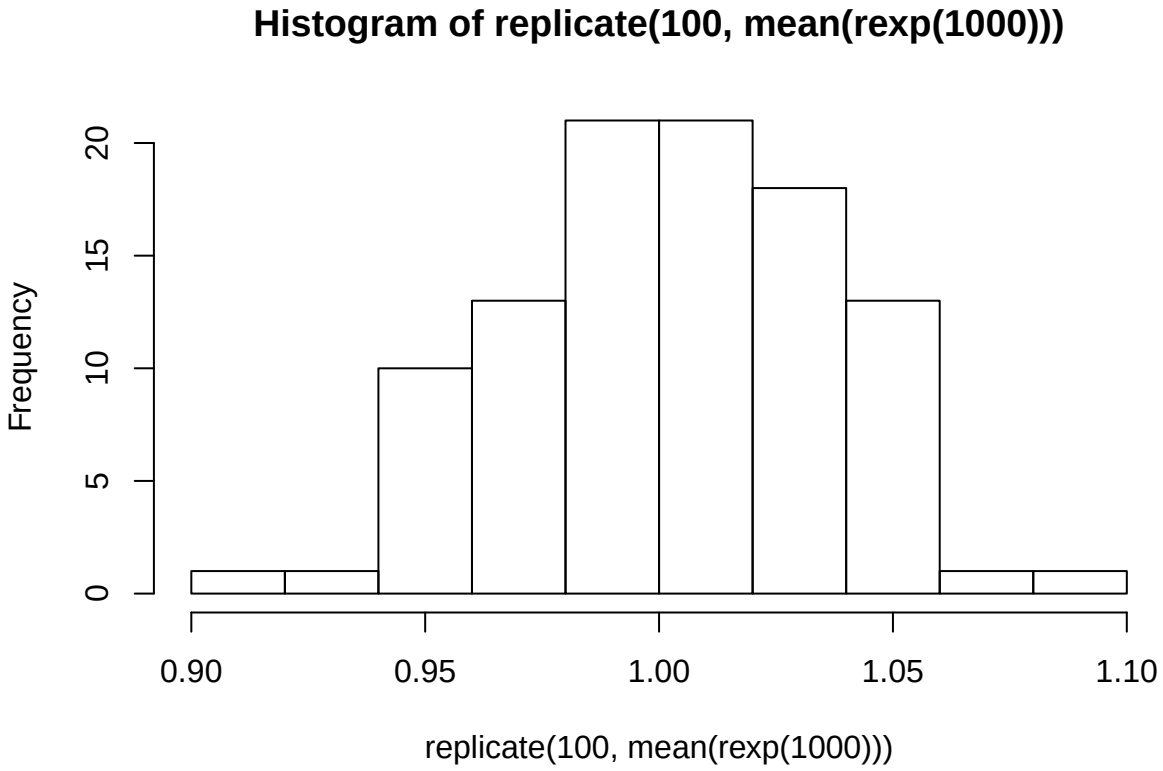
Totéž, co předchozí příklad:

```
vapply(df, function(x) sum(is.na(x)), 1L)
```

```
## x y z
## 0 1 2
```

Speciálním případem je situace, kdy potřebujete mnohokrát vyhodnotit nějaký výraz, který typicky zahrnuje náhodná čísla. K tomu slouží funkce `replicate(n, e)`, která n-krát vyhodnotí výraz e, který typicky obsahuje generátor náhodných čísel. Následující kód spočítá sto průměrů tisíce náhodných čísel s exponenciálním rozdělením a vykreslí z něj histogram:

```
hist(replicate(100, mean(rexp(1000))))
```



Funkce `replicate()` se stejně jako `sapply()` snaží výsledek zjednodušit na vektor nebo matici. To však lze vypnout pomocí parametru `simplify = FALSE` (příklad níže).

Poznámka: Funkce `lapply()`, `sapply()` a `vapply()` pracují s každým prvkem samostatně a izolovaně, takže nezáleží na pořadí, v jakém jsou tyto prvky zpracovány. To, mimo jiné, umožňuje paralelizaci, tj. zpracování každého prvku na jiném jádře počítače nebo jiném prvku počítačového klastru. Pokud jsou data tak velká, že se to vyplatí, je možné použít ekvivalenty `apply()` funkcí z balíku **foreach** (např. s backendem z balíku **doParallel**).

Práce nad sloupci datasetu

Někdy chceme provést nějakou operaci se všemi sloupci datasetu tak, aby výsledkem byl opět dataset. Výše uvedené funkce se k tomu zdánlivě nehodí: `lapply()` vrátí seznam, zatímco `sapply()` a `vapply()` matici. To je však možné obejít dvěma způsoby: 1) vzniklý seznam převést na `data.frame` explicitní konverzí nebo 2) výsledku vnutit subsetováním strukturu `data.frame` (funguje jen při úpravě dat "in place").

Rekněme, že máte dataset, který obsahuje pouze číselné sloupce a že každý z nich chcete vydělit 1 000 (možná v důsledku měnové reformy):

```
df <- data.frame(income = c(1.3, 1.5, 1.7) * 1e6,
                 rent = c(500, 450, 580) * 1e3,
                 loan = c(0, 250, 390) * 1e9)
df
```

```
##    income   rent    loan
## 1 1300000 500000 0.0e+00
## 2 1500000 450000 2.5e+11
## 3 1700000 580000 3.9e+11
```

```
# výsledek je seznam
lapply(df, function(x) x/ 1000)
```

```
## $income
## [1] 1300 1500 1700
##
## $rent
## [1] 500 450 580
##
## $loan
## [1] 0.0e+00 2.5e+08 3.9e+08
```

```
# výsledek při explicitní konverzi
as.data.frame(lapply(df, function(x) x/ 1000))
```

```
##   income rent    loan
## 1   1300  500 0.0e+00
## 2   1500  450 2.5e+08
## 3   1700  580 3.9e+08
```

```
#
df[,] <- lapply(df, function(x) x/ 1000)
df
```

```
##   income rent    loan
## 1   1300  500 0.0e+00
## 2   1500  450 2.5e+08
## 3   1700  580 3.9e+08
```

(V našem případě by samozřejmě stačilo i `df / 1000`.)

Selekce prvků podle funkcí

Funkce `Filter(f, x)` vrátí seznam prvků vektoru `x`, pro který vrací logická funkce `f` hodnotu `TRUE`.

Funkce `Find(f, x)` vrátí první prvek vektoru `x`, pro který vrací logická funkce `f` hodnotu `TRUE`.

Funkce `Position(f, x)` vrátí indexy prvků vektoru `x`, pro který vrací logická funkce `f` hodnotu `TRUE`.

```
Filter(is.character, df)    # vrací seznam, tj. zde data.frame
```

```
## data frame with 0 columns and 3 rows
```

```
Find(is.character, df)      # vrací prvek data.frame, tj. zde vektor
```

```
## NULL
```

```
Position(is.character, df)  # třetí sloupec data.frame
```

```
## [1] NA
```

Reduce

Funkce `Reduce(f, x)` redukuje vektor na jednu hodnotu, a to tak, že rekurzivně volá funkci `f` vždy na dva argumenty – nejdříve zkombinuje první a druhý prvek vektoru `x`, pak výsledek tohoto výpočtu se třetím prvkem vektoru `x` atd.

Součet prvků vektoru (funguje proto, že `+` je ve skutečnosti binární funkce `+(a, b)`; pomocí backticks je možné i zavolat i jako funkci):

```r
Reduce(`+`, 1:3)  # totéž, co ((1 + 2) + 3) = 6
```

```
## [1] 6
```

Funkce `Reduce()` umožňuje použít funkci, která bere jen dva argumenty na mnoho argumentů. Např. funkce `intersect()` bere dvě množiny a vrací jejich průnik. Pokud chceme najít průnik pěti množin, stačí použít `Reduce()`, které rekurzivně počítá průnik předchozího výsledku a další množiny (příklad převzat z AdvR, s. 220):

```r
l <- replicate(5, sample(1:10, 15, replace = T), simplify = FALSE)
str(l)
```

```
## List of 5
##  $ : int [1:15] 8 2 5 5 9 3 6 1 1 5 ...
##  $ : int [1:15] 10 5 8 7 10 2 4 2 1 3 ...
##  $ : int [1:15] 6 1 6 6 8 9 6 3 7 5 ...
##  $ : int [1:15] 8 5 6 3 6 4 9 2 10 6 ...
##  $ : int [1:15] 1 6 9 1 9 8 4 7 8 10 ...
```

```r
Reduce(intersect, l)
```

```
## [1] 8 9 3 4
```

Funkci lze také použít k volání funkce na proměnné uložené v seznamu. Typické je např. spojit datasety uložené v seznamu pomocí funkce `rbind()`:

```r
l <- list(data.frame(x = 1:3, y = letters[1:3]),
          data.frame(x = 4:7, y = letters[4:7]),
          data.frame(x = 8:9, y = letters[8:9]))
Reduce(rbind, l)
```

```
##   x y
## 1 1 a
## 2 2 b
## 3 3 c
## 4 4 d
## 5 5 e
## 6 6 f
## 7 7 g
## 8 8 h
## 9 9 i
```

Komplexní příklad 1

Někdy potřebujete načíst velké množství tabulek z jednotlivých `.csv` souborů, zkontrolovat jejich konzistenci a spojit je dohromady. To dělá následující kód, který představuje zjednodušenou verze části jednoho mého projektu:

```r
# adresář s daty
DATADIR <- file.path("..", "testdata")
# seznam jmen souborů, které splňují určitou masku danou regulárním výrazem
PRODUCT_FILES <- dir(DATADIR, "products_.*\\.csv\\.gz", full.names = TRUE)
# řetězec s dvaceti "c" (vnucený typ načítaných souborů ve funkci read_csv2())
PRODUCT_FILE_MASK <- paste(rep("c", 20), collapse = "")
```

```
# načtení jednotlivých souborů
product_data <- lapply(PRODUCT_FILES,
                       function(x) read_csv2(file = x, col_types = PRODUCT_FILE_MASK))
# kontrola, že všechny datasety mají stejnou strukturu
product_col_names <- lapply(product_data, colnames)
product_col_names_test <- sapply(product_col_names,
                                 function(x) all.equal(product_col_names[[1]], x))
if (!(all(identical(product_col_names_test,
                    rep(TRUE, length = length(product_col_names))))))
    stop("Product data sets have different columns!")
# spojení jednotlivých tabulek do jedné
product_data <- Reduce(rbind, product_data)
```

Vlastní spojení dat je možné provést efektivněji pomocí funkce z balíku **dplyr**:

```
product_data <- dplyr::bind_rows(product_data)  # pokud není balík načten, musíme funkci zavolat pln
```

nebo pomocí funkce `do.call()`. Obě varianty jsou ve skutečnosti efektivnější než `Reduce()`, protože alokují paměť efektivněji (ideálně jen jednou).

Komplexní příklad 2

Někdy chceme odhadnout mnoho ekonometrických modelů, které se liší buď použitým tvarem modelu, nebo použitými daty. Zde se podíváme na první případ.

```
# seznam formulí, které popisují modely (viz lekce o ekonometrii)
formulas <- list(mpg ~ disp,
                 mpg ~ I(1 / disp),
                 mpg ~ disp + wt,
                 mpg ~ I(1 / disp) + wt)
# modely odhaduje funkce lm(); její parametry jsou
# 1. formule, která popíše tvar modelu
# 2. data
# zde odhadneme model pro všechny formule a uložíme do seznamu
models <- lapply(formulas, function(f) lm(f, data = mtcars))
# vypíšeme modely
lapply(models, summary)
```

```
## [[1]]
##
## Call:
## lm(formula = f, 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
##
##
## [[2]]
##
## Call:
## lm(formula = f, data = mtcars)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -3.7383 -1.8216 -0.0751  1.0487  4.6105
##
## Coefficients:
##              Estimate Std. Error t value Pr(>|t|)
## (Intercept)   10.7520     0.7994   13.45 3.06e-14 ***
## I(1/disp)   1557.6739   114.8935   13.56 2.49e-14 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 2.295 on 30 degrees of freedom
## Multiple R-squared:  0.8597, Adjusted R-squared:  0.855
## F-statistic: 183.8 on 1 and 30 DF,  p-value: 2.494e-14
##
##
## [[3]]
##
## Call:
## lm(formula = f, data = mtcars)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -3.4087 -2.3243 -0.7683  1.7721  6.3484
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept) 34.96055    2.16454  16.151 4.91e-16 ***
## disp        -0.01773    0.00919  -1.929  0.06362 .
## wt          -3.35082    1.16413  -2.878  0.00743 **
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 2.917 on 29 degrees of freedom
## Multiple R-squared:  0.7809, Adjusted R-squared:  0.7658
## F-statistic: 51.69 on 2 and 29 DF,  p-value: 2.744e-10
##
##
## [[4]]
##
## Call:
## lm(formula = f, data = mtcars)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -2.7085 -1.5121 -0.6191  1.5142  4.2314
##
## Coefficients:
##              Estimate Std. Error t value Pr(>|t|)
## (Intercept)   19.0242     3.4520   5.511 6.13e-06 ***
```

```
## I(1/disp)   1142.5600   199.8427   5.717 3.47e-06 ***
## wt            -1.7976     0.7327  -2.453   0.0204 *
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 2.124 on 29 degrees of freedom
## Multiple R-squared:  0.8838, Adjusted R-squared:  0.8758
## F-statistic: 110.3 on 2 and 29 DF,  p-value: 2.788e-14
```

```
# získáme koeficient determinace odhadnutých modelů
sapply(models, function(m) summary(m)$r.squared)
```

```
## [1] 0.7183433 0.8596865 0.7809306 0.8838038
```

Pokud máme odhad modelů uložený v seznamu, můžeme jej samozřejmě v R vypsat i do pěkné tabulky (jak, to uvidíte později):

Table 1:

	Dependent variable:			
	mpg			
	(1)	(2)	(3)	(4)
disp	−0.041***		−0.018*	
	(0.005)		(0.009)	
I(1/disp)		1,557.674***		1,142.560***
		(114.894)		(199.843)
wt			−3.351***	−1.798**
			(1.164)	(0.733)
Constant	29.600***	10.752***	34.961***	19.024***
	(1.230)	(0.799)	(2.165)	(3.452)
Observations	32	32	32	32
R^2	0.718	0.860	0.781	0.884
Note:				*p<0.1; **p<0.05; ***p<0.01

Funkce `do.call()`

Funkce `do.call(f, args)` volá funkci `f` s argumenty uloženými v seznamu `args`. Funkce může být dokonce zadána jako řetězec obsahující jméno funkce.

Spojení datasetů:

```
product_data <- do.call(rbind, product_data)
```

Volání funkce `mean()`, pokud jsou její parametry uložené v seznamu:

```
l <- list(c(1, 3, 5, NA, 9), na.rm = TRUE)
mean(l[[1]])                # průměr z prvního pruku l
```

```
## [1] NA
```

```r
mean(l[[1]], na.rm = TRUE)  # průměr z prvního pruku l, NA se vynechává
```

```
## [1] 4.5
```

```r
do.call(mean, l)            # totéž -- mean(c(l[[1]], l[[2]]))
```

```
## [1] 4.5
```

```r
do.call("mean", l)          # totéž
```

```
## [1] 4.5
```

Speciality

Pokud potřebujete paralelně iterovat nad dvěma nebo více vektory, můžete použít funkce `Map()` a `mapply()`, viz dokumentace.

K vektorizaci obecné funkce slouží funkce `Vectorize()`, viz dokumentace.

Balík purrr

Balík **purrr* poskytuje nástroje pro pokročilé funkcionální programování, viz https://blog.rstudio.org/2015/09/29/purrr-0-1-0/, https://blog.rstudio.org/2016/01/06/purrr-0-2-0/ a https://github.com/hadley/purrr.

Operace nad maticemi a poli

Pro přímou práci s maticemi a poli slouží funkce `apply(X, margin, f, ...)`, kde `X` je tabulka o dvou nebo více rozměrech, `margin` je rozměr (řádek je `1`, sloupec `2` atd.) a funkce, která se použije na daný rozměr tabulky. Další parametry se předají funkci `f`.

```r
m <- matrix(1:20, nrow = 4)
m
```

```
##      [,1] [,2] [,3] [,4] [,5]
## [1,]    1    5    9   13   17
## [2,]    2    6   10   14   18
## [3,]    3    7   11   15   19
## [4,]    4    8   12   16   20
```

```r
apply(m, 1, mean)  # průměrná čísla na řádcích
```

```
## [1]  9 10 11 12
```

```r
apply(m, 2, mean)  # průměrná čísla ve sloupcích
```

```
## [1]  2.5  6.5 10.5 14.5 18.5
```

(Pro sčítání a průměrování řádků matic poskytuje R hotové funkce: `rowSums()`, `colSums()`, `rowMeans()` a `colMeans()`, které jsou navíc optimalizované a rychlejší než použití `apply()`. Podle mě je však zbytečné si je pamatovat.)

Upoutávka na dplyr

Někdy je potřeba vektory, matice a datasety rozdělit podle nějaké externí proměnné (typicky faktoru) nebo na ně aplikovat nějaké funkce po skupinách daných nějakou externí proměnnou (opět typicky faktorem). Dají se k tomu použít funkce `tapply()` a `split()`. Ve většině případů se však takové operace provádí nad datasety, kde je možné použít mnohem komfortnější funkce z balíku **dplyr**, kterým se budeme zabývat později.

Srovnání s cykly

R poskytuje i klasické cykly `for()`, `while()` a `repeat()` (viz kapitola o funkcích a dokumentace). Ve většině případů však nejsou potřeba a je lepší použít funkce typu `apply()`: jednak jsou většinou mnohem rychlejší, jednak je výsledný kód podstatně čitelnější.

Existují však tři typické případy, kdy je užitečné použít klasické cykly:

- náhrada prvků vektorů, matic a datasetů "na místě"
- rekurzivní výpočty
- cyklus s neznámým počtem opakování (`while` a `repeat`)

Příklady viz Hadley Wickham: *Advanced R*, oddíl 11.6.

Domácí úkol

Upravte skript `hw06.R` tak, že v něm vytvoříte funkci `smartScale(df)` takovou, že vezme `data.frame df` a vrátí nový dataset takový, že

- všechny numerické sloupce budou normalizované, tj. bude z nich odečtena střední hodnota a výsledek bude podělen směrodatnou odchylkou (k normalizaci využijte funkci `scale()`)
- všechny ne-numerické sloupce zůstanou stejné

Poznámka: otestujte, zda vaše funkce funguje správně pro celá i reálná čísla, faktory, logické hodnoty i řetězce.