Machine Learning - 1100-ML0ENG (Ćwiczenia informatyczne Z-23/24)

Data Structures in R

Vector

In R, you can create a vector by generating the number or character or logical sequence using any other facilities in R, such as, creating random numbers.

You can also use **sample() function** to generate random vectors. sample takes a sample of the specified size from the specified elements using either with or without replacement.

```
sample(500, 30) #sample 30 numbers from 1 to 500
x11 <- sample(2:10, 4)
x11
x12 <- sample(1:20, 25, rep=TRUE) # sampling be with replacement
x12
```

Because the return value of the **sample()** function is a randomly determined number, if you try this function repeatedly, you'll get different results every time.

```
> sample(500, 30)
 [1] 240 133 427 114 393 321 210  47   2 261 257 105 350 469 252  85 273 367 322 109
[21] 167 154 433  96 325  66  51  44 478 131
# A second time
> sample(500, 30)
 [1] 443 386 396 103  24  20 141 172 363 124 254 372 311 100 483   1 487 307   9  13
[21] 292 274 118 439 320 486  99 228 466 468
```

This is the correct behavior in most cases, but sometimes you may want to get repeatable results every time you run the function.
The set.seed function in R can help when you want to verify or replicate a sample.

*If you provide a seed value, the random-number sequence will be reset to a known state. This is because **R doesn't create truly random numbers, but only pseudo-random numbers**. A pseudo-random sequence is a set of numbers that, for all practical purposes, seem to be random but were generated by an algorithm. When you set a starting seed for a pseudo-random process, R always returns the same pseudo-random sequence. But if you don't set the seed, R draws from the current state of the random number generator-**RNG** (?RNG).*

```
> set.seed(24)
> sample(500, 30)
 [1] 147 113 351 258 329 456 139 377 395 126 297 182 328 488 156 439  96  22 242  67
[21]  46 265 347  66 107 327  37 284 298 483
> set.seed(24)
> sample(500, 30)
 [1] 147 113 351 258 329 456 139 377 395 126 297 182 328 488 156 439  96  22 242  67
[21]  46 265 347  66 107 327  37 284 298 483
```

We were used 24 (but it can be any number you want it to be in). So, if you run the same sample with the same seed in R, you should get the same results. If you run the same sample without a seed, each time, you would get a different result.

M&Ms

If we want to take a sample from things that are not numbers. For example, pretend we are taking M&Ms out of a jar that has blue, green, and red M&Ms, and we want to pretend we're randomly taking M&Ms out of the jar.

```
candy = c("blue","green","red")
sample(candy, 20, replace=TRUE)
```

We can boost some values in the sample

```
jar<- sample(candy, 20, replace=T, prob=c(0.7, 0.2, 0.1))
table(jar)
```

The rnorm() function generates for the random normal distribution with mean=0 and standard deviation sd=1.

```
rnorm(5)
```

The **runif()** function generates random numbers from Uniform distribution

```
runif(10)
runif(n=5, min=10, max=20)
```

Factor

Factors are special types of objects in R. They're neither character vectors nor numeric vectors, although they have some attributes of both. Factors behave a little bit like character vectors in the sense that the unique categories are often text. Factors also behave a little bit like integer vectors because R **encodes the integers as levels**.

```
(w<-rep(c("high", "medium", "low"),c(4,3,5)))
(factor_w<-factor(w))
or
```

```
blood <- factor(c("O", "AB", "AB","O", "AB", "A", "A", "AB", "A"),
                                        levels = c("A", "B", "AB", "O"))
blood
blood[1:2]
blood[-3]
```

Internally a factor is stored as a numeric value associated with each level. This means you can set and investigate the levels of a factor separately from the values of the factor. An attempt to substitute an undefined element in the attributes level give us a message error, and the value of this element will be undefined NA.

```
factor_w[13]<-"moderate"

Warning message:
In `[<-.factor`(`*tmp*`, 3, value = "super") :
  invalid factor level, NA generated

w[13]<-"moderate"
```

Useful functions:

1. **levels** - to look at the levels of a factor;
2. **table, summary** - get a tabular summary of the values of a factor;

```
levels(factor_w)
table(factor_w)
summary(factor_w)
```

The "inverse" function for factor () is a function as.vector()

```
as.vector(factor_wzrost)
as.integer(factor_wzrost)
```

another syntax of factor

```
f1 <- factor(c(2,3,4), levels=1:4)
f1
levels(f1)<-c("bad", "average", "good", "ideal")
f1
```

We can't doing arthmetics

```
f2 <- factor(c(2,3,2,2,3,4), levels=1:5)
print(f2)
x<-f2+2
```

The cut() function creates bins of equal size (by default) in your data and then classifies each element into its appropriate bin.

```
age <- sample(1:100, 16, replace = TRUE)
age
cut(age, c(0, 18, 26, 100))
f.age <-cut(age, c(0, 18, 26, 100))
table(f.age)

f.age.2<-cut(age, 5)
table(f.age.2)
```

List

You can create a list with the **list()** function.

You can use the **list()** function in two ways:

1. to create an unnamed list
2. to create a named list

The difference is small; in both cases, think of a list as a big box filled with a set of bags containing all kinds of different stuff. If these bags are labeled instead of numbered, you have a named list.

Creating an unnamed list is as easy as using the **list()** function and putting all the objects you want in that list between the ().

```
a<-list(1:5, LETTERS[1:15], list("ą","ć","ę"))
```

In order to create a labeled, or named, list, you simply add the labels before the values between the () of the list() function, like this:

```
x <- c(2,3,2,5,4,3,6,7,8)
list.x <- list(mean.value=mean(x),minimum=min(x),maximum=max(x))
```

Function str() gives us an easy way to look at the structure of any object.

```
str(list.x)

List of 3
 $ mean.value : num 4.44
 $ minimum : num 2
 $ maximum: num 8
```

In the case of a named list, you can access the components using the $

```
list.x$maximum
```

For both named and unnamed lists, you can use two other methods to access components in a list:

1. using **[[ ]]** gives you the component itself;
2. using **[ ]** gives you a list with the selected components;

```
list.x[[3]]
list.x[3]
```

or

```
 a[[2]]
[1] "A" "B" "C" "D" "E" "F" "G" "H" "I" "J" "K" "L" "M" "N" "O"

 a[2]
[[1]]
 [1] "A" "B" "C" "D" "E" "F" "G" "H" "I" "J" "K" "L" "M" "N" "O"
```

so a[[2]][10] give us access to 10 element in a vector. And a[2][10] give us NULL.

If we want to add names to unnamed lists, we can use names function.

```
names.new<-c("numbers","letters","polish tails")
names(a)<-names.new
str(a)
a$letters
```

Data frame

You can combine values of the same type into either a vector or a matrix. But **datasets are, in general, built up from different data types.** You can have, for example, the names of your employees, their salaries, and the date they started at your company all in the same dataset. But you can't combine all this data in one matrix without converting the data to character data. So, you need a new data structure to keep all this information together in R. That data structure is a data frame.

To combine a number of vectors into a data frame, you simple add all vectors as arguments to the **data.frame()** function, separated by commas. R will create a data frame with variables that are named the same as the vectors used. Keep in mind that these vectors must have the same length.

```
data1 <- data.frame(LETTERS[1:10],1:10,rep(c(F,T),5))
names(data1) <- c("Initial", "Order", "Even")
```

stringsAsFactors

```
employee <- c("Sheldon Cooper", "Leonard Hofstadter", "Howard Wolowitz")
salary <- c(31000, 23400, 26800)
startdate <- as.Date(c("2017-12-1", "2008-3-25", "2007-3-14"))

employ.data1 <- data.frame(employee, salary, startdate)
str(employ.data1)

employ.data1[4,]<-c(" Rajesh Koothrappali",26900, "2018-01-23")

employ.data2 <- data.frame(employee, salary, startdate, stringsAsFactors = TRUE)
str(employ.data2)

employ.data2[4,]<-c(" Rajesh Koothrappali",26900, "2018-01-23")
```

Data frame is used to load datasets.

Built in datasets

Simple data types

One of the packages in the base R distribution is called **datasets**, and it is entirely filled with example datasets. Many other packages also contain datasets. You can see all the datasets that are available in the packages that you have loaded using the **data** function:

```
data()
str(mtcars)
```

The data was extracted from the 1974 Motor Trend US magazine, and comprises fuel consumption and 10 aspects of automobile design and performance for 32 automobiles (1973–74 models).

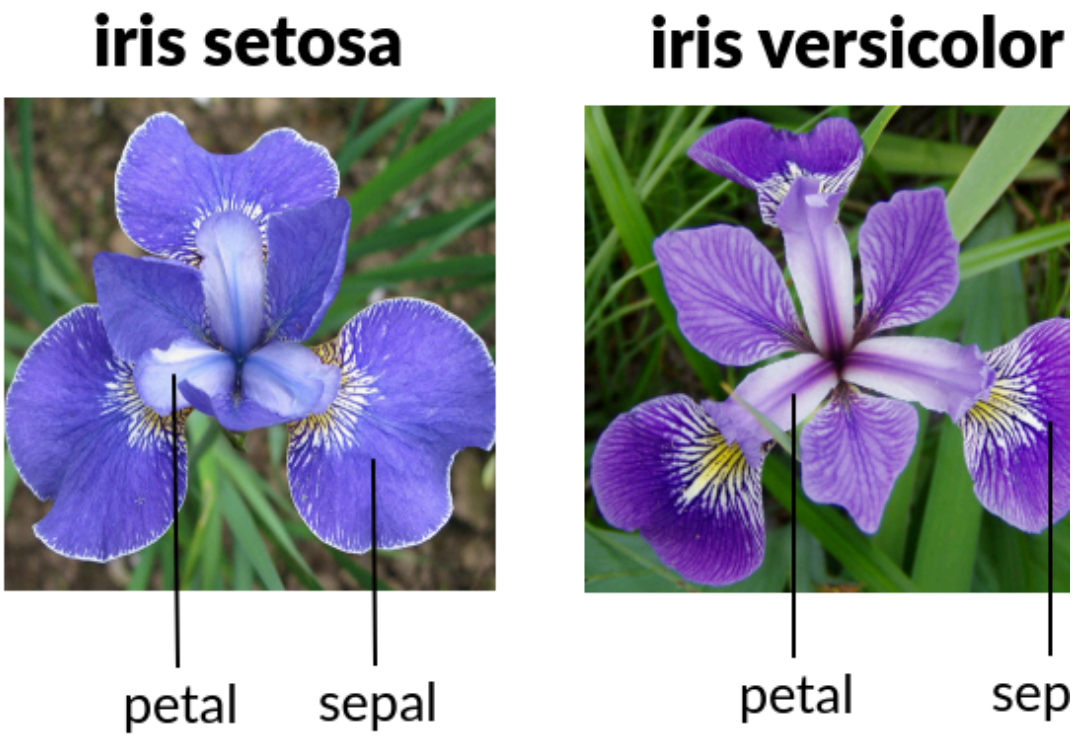
- [, 1] mpg Miles/(US) gallon
- [, 2] cyl Number of cylinders
- [, 3] disp Displacement (cu.in.)
- [, 4] hp Gross horsepower
- [, 5] drat Rear axle ratio

- [, 6] wt Weight (1000 lbs)
- [, 7] qsec 1/4 mile time
- [, 8] vs V/S
- [, 9] am Transmission (0 = automatic, 1 = manual)
- [,10] gear Number of forward gears
- [,11] carb Number of carburetors

Let's prepare the data frame mtcars.

```
cars <- mtcars[c(1, 2, 9, 10)]
str(cars)
cars$am<-as.factor(cars$am)
levels(cars$am)<-c("auto", "manual")
```

Iris flower data set



We consider the iris dataset and we find the frequencies in particular species

```
str(iris)
table(iris$Species)
table(iris$Sepal.Length)
table(iris$Sepal.Length, iris$Species)
```

Last modified: środa, 4 października 2023, 7:52

Accessibility settings

Przetwarzanie danych osobowych

Platformą administruje Komisja ds. Doskonalenia Dydaktyki wraz z Centrum Informatyki Uniwersytetu Łódzkiego Więcej

Informacje na temat logowania

Na platformie jest wykorzystywana metoda logowania za pośrednictwem Centralnego Systemu Logowania.

Studentów i pracowników Uniwersytetu Łódzkiego obowiązuje nazwa użytkownika i hasło wykorzystywane podczas logowania się do systemu USOSweb.

Deklaracja dostępności