Introductie tot R

Statistiek en Probabiliteit
Academiejaar 2019-2020

R en RStudio
Dit practicum heeft tot doel om vertrouwd te raken met het statistisch pakket R. Je zal in dit document een beknopte introductie krijgen tot R. Een uitgebreidere intro tot R kan men vinden in het boek R for Data Science `http://r4ds.had.co.nz/`.
R is een freeware programmeertaal, voor meer info kan je op hun website `https://www.r-project.org/` terecht. RStudio (ook freeware) is een integrated development environment (`https://nl.wikipedia.org/wiki/Integrated_development_environment`) voor R, deze maakt het gemakkelijker om met R te werken `https://www.rstudio.com/`.
Via Athena kunnen jullie RStudio openen en werken op de servers van UGent. Als je R en RStudio op je eigen laptop of pc wil installeren, kan je deze op de websites `https://lib.ugent.be/CRAN/` en `https://www.rstudio.com/products/rstudio/download/` downloaden (merk op dat je met RStudio niets kan doen indien je geen R hebt!).

Open nu RStudio. Zoals je kan zien, heeft RStudio 4 vensters. Linksonder zie je de Console, hierin kan je R-code schrijven en laten uitvoeren door op enter te drukken. Handig te weten, is dat je via de pijltjes omhoog en naar beneden je vorige commando's kan oproepen. De meeste output die je aan R vraagt worden door R in de console weergegeven.
Objecten of functies die je hebt aangemaakt kan je zien in de 'global environment' in het scherm rechtsboven. Hier kan men bijvoorbeeld ook datasets importeren. In het tablad "History"kan je reeds uitgevoerde commando's terugvinden.
Gevraagde plots/figuren verschijnen in het scherm rechtsonder, deze plots kan je via 'Export' opslaan op de computer. In het tablad 'Help' kan men informatie over functies en packages opvragen. We kunnen ook rechtstreeks in de console om informatie vragen over bijvoorbeeld de functie 'sum' of package 'stats' via de commando's

- ?sum
- help(sum)
- ??sum
- help.search("sum")

De eerste 2 commando's zullen een hulppagina zoeken van de functie of package, de laatste 2 commando's zoeken alle hulppagina's waar het woord 'sum' in voorkomt.

Als laatste hebben we het scherm linksboven. Het is mogelijk dat dit er nog niet is. Ga in dit geval naar de menubalk en druk op `File -> New File -> R Script`. In dit scherm kan je scripts schrijven van R-code en commentaar. Dit is bijvoorbeeld nuttig als je vaak dezelfde sequentie van code zou moeten uitvoeren, je kan het zien als een recept of werkmethode dat uitgevoerd wordt op een gelijkaardig probleem. Je selecteert gewoon het blok code dat je wil laten uitvoeren en drukt op ctrl + enter. De geselecteerde code wordt dan gekopieerd naar de console en sequentieel uitgevoerd.

Het is vaak nuttig dat je commentaar schrijft bij je code, zodat jij of iemand anders die dit later wil gebruiken, weet wat je code doet. Deze commentaar kan je laten negeren door R als je er een # voor zet (de code krijgt dan een groene kleur in RStudio). Deze scripts kan je dan opslaan op een plaats naar keuze en later openen via de opties onder 'File'. Deze script-bestanden eindigen met '.R'.

Objecten

Je kan objecten creëren in R via een pijltje naar links, dit pijltje bestaat uit een kleiner dan-teken en een minteken. In RStudio is ALT plus - een shortcut[1] voor deze operator. Bijvoorbeeld

```
some.text <- "One does not simply master R in one session"
```

maakt het object 'some.text' dat als inhoud de string 'One does not simply master R in one session' heeft. Je kan de inhoud van een object opvragen door simpelweg de naam van het object in het commandovenster in te geven. Probeer dit in RStudio.
Merk op dat het teken = ook werkt om objecten aan te maken. Doch is het aan te raden dit niet te doen. Het teken = is contextgevoelig en kan in sommige gevallen voor eigenaardige resultaten zorgen. Voor meer info, zie `https://www.rdocumentation.org/packages/base/versions/3.5.1/topics/assignOps`.

Namen van objecten moeten starten met een letter of een punt, indien deze start met een punt, mag het tweede character geen cijfer zijn. Namen mogen enkel bestaan uit letters, cijfers, underscores of punten. Merk op dat R hoofdlettergevoelig is.
Er zijn verschillende types basisobjecten in R:

- numerical
- integer: gehele getallen, specifiek geval van numerical.
- character: een serie van opeenvolgende letters/cijfers/...
  Begin en eind van de string worden aangegeven via dubbele aanhalingstekens.
- logical: een waarde die ofwel waar of vals is
  In R zijn deze constanten bekend als TRUE en FALSE.
- factor: een categorische variabele met meerdere niveaus (levels). Deze factors kunnen al dan niet geordend zijn en je kan de niveaus koppelen aan labels. De labels zijn wat R zal tonen wanneer je het object opvraagt. Indien er geen labels worden meegegeven, zal R de levels hiervoor gebruiken. Meer hierover later.

Via deze basisobjecten kan men omvattende objecten zoals vectoren, matrices, lijsten en dataframes definiëren. Een vector is een opeenvolging van basisobjecten van hetzelfde type. Merk op dat een numerieke waarde of een character per default worden opgeslagen als een vector met lengte 1. Een matrix is in essentie een vector die een extra dimensie toegewezen krijgt. Een lijst is een opeenvolging van willekeurige objecten en een dataframe is een lijst waarvan de elementen vectoren zijn met een gelijke lengte. Dit wordt bijvoorbeeld gebruikt voor een klassieke dataset, de rijen vormen de verschillende observaties en de kolommen zijn de verschillende variabelen.

Opmerking: gebruik spatiëring en haakjes om je code ondubbelzinnig te maken voor R. Zo doen de volgende twee commando's niet hetzelfde:

```
x <- 3
x < -3
```

[1] Een volledige cheat sheet van shortcuts kan je krijgen via ALT plus SHIFT plus K.

Basisfuncties

Indien je het type wil weten van een object, kan je de functie class() gebruiken. Wil je meer weten, bijvoorbeeld over de structuur, kan je gebruik maken van de functie str().
Om te begrijpen hoe je functies moet gebruiken, zullen we kijken naar een voorbeeld. open de hulppagina van de functie `log`. We vinden hier allerlei info over deze functie, waaronder een beschrijving van wat de funtie doet, hoe men de functie moet gebruiken, een beschrijving van de argumenten, voorbeelden, ...
We zien hier dat de standaardvorm van deze functie (log) 1 of 2 argumenten heeft: x en base. Bij base staat er al een `default value` ingevuld, namelijk exp(1), ofwel e. Dit betekent dat tenzij we specifiek een waarde meegeven voor base, het natuurlijk logaritme wordt berekend. De waarde x heeft natuurlijk geen default value, als deze niet meegegeven wordt, geeft R een error. De volgorde waarin je de argumenten geeft is belangrijk, tenzij je de argumenten specifieert. Wat zal de output zijn van de volgende commando's? Ga deze na in R.

```
log(4,2)
log(2,4)
log(4,base=2)
log(base=2,4)
log(2,x=4)
log(base=2,x=4)
```

De gemakkelijkste manier om een vector te maken is via de functie c() van combine of concatenate. Deze zal als uitvoer een vector geven met de meegegeven argumenten aan elkaar 'geplakt'. Meegegeven argumenten mogen ook vectoren zijn. Wat is de uitvoer van het volgende commando? Ga dit na in R.

```
a <- c(1,3,5)
b <- c(1,a,10,a, c(2,4))
b
```

Elementen van een vector kunnen opgevraagd worden aan de hand van vierkante haken. Bijvoorbeeld b[5] geeft het 5e element van de vector b. Merk op dat R dus begint te tellen van 1 en niet 0. Je kan tussen de haken ook een vector met posities meegeven. Wat zal de uitvoer zijn van het volgende commando?

```
b[c(11,5,4,4,9)]
```

Merk op dat het eerste element hiervan de waarde NA (not available) heeft, dit komt omdat R de waarde niet kon berekenen (waarom?). Je kan ook een deelvector opvragen met bepaalde posities verwijderd.

```
b[-c(1,2,6,9)]
```

Je kan tevens waarden aanpassen van je vectoren:

```
b[c(3,1,5)] <- c(2,3,9)
```

De voor jullie vertrouwde functies if(), for() en while() zijn tevens aanwezig in R. Tussen de haken van if en while komt de voorwaarde, voor het coderen van voorwaarden kan je gebruik maken van logische operatoren, later meer hierover. Het gebruik van de functie for gaat als volgt:

```
for(i in 1:100){print(i)} #i zal alle getallen van 1 t.e.m. 100 doorlopen

letters # standaardvector met alle letters van het alfabet
for(x in letters){print(x)} #x zal door alle elementen van de vector letters lopen
```

Vectoren

Er zijn vele shortcuts om vectoren te maken, zo zal

```
1:100
```

een vector maken met de getallen 1 t.e.m 100. Via de functie seq() kan je rekenkundige rijen maken. Deze heeft de argumenten from en to (begin- en eindwaarde) en als laatste argument geef je ofwel het argument by (grootte van de stappen, mag negatief zijn) of length.out (de lengte van de vector) mee.
Maak bijvoorbeeld de vector (10,8,6,4,2,0) op 2 manieren, een keer adhv het argument by en een keer adhv het argument length.out.

Ook de functie rep() zorgt voor vele functionaliteiten. Onderzoek deze aan de hand van de volgende commando's:

```
rep(c(1,2,3), times = 4)
rep(c(1,2,3), each=3)
rep(c(1,2,3), length.out=5)
```

Onderzoek aan de hand van een numerieke vector x wat de volgende functies doen:

```
length(x)
sum(x)
prod(x)
min(x)
max(x)
cumsum(x)
cumprod(x)
mean(x)
```

Numerieke vectoren kunnen ook opgeteld, vermenigvuldigd, etc. worden. R creëert dan een vector waarvan het i-de element gelijk is aan de bewerking van de i-de getallen van de vectoren. Bijvoorbeeld:

```
(1:3)*c(3,7,6)
c(2,4,5)^c(3,2,1)
```

Belangrijk te weten is dat R aan recycling doet, je kan bewerkingen uitvoeren tussen vectoren van ongelijke lengte, zolang de lengte van 1 van beide vectoren een veelvoud is van de andere.

```
(4:9)/(1:2)
c(3,5)+rep(1:3, times=2)
(2:7)*4
```

Begrijp je hoe recycling werkt?
Wat gebeurt er indien de lengtes van de vectoren geen veelvoud zijn van elkaar? Onderzoek dit.

De meeste functies die van toepassing zijn op 1 object, zullen ook bruikbaar zijn voor vectoren:

```
x <- c(-1,4,2)
abs(x)
exp(x)
sqrt(x)
```

Bij het laatste commando krijg je een warning van R: 'NaNs produced'. NaN staat voor 'not a number', dit is een speciaal geval van NA. R weet dat een getal gevraagd is, maar kon deze niet berekenen. In dit geval omdat de vierkantswortel van -1 niet bestaat.

Basisbewerkingen

Stel dat x en y getallen zijn, onderzoek de functionaliteiten van de operatoren in de volgende commando's:

```
x+y
x-y
x*y
x/y
x^y
x**y
x%%y
x%/%y
abs(x)
log(x, base=y)
exp(x)
sqrt(x)
factorial(x)
choose(x,y)
round(x, digits=y)
signif(x, digits=y)
floor(x)
ceiling(x)
```

Booleans en logische operatoren

R kent de volgende logische operatoren om getallen x en y te vergelijken:

```
x==y
x!=y
x>y
x<y
x>=y
x<=y
```

De uitkomst is telkens een boolean. De eerste twee zijn tevens ook bruikbaar om 2 willekeurige objecten te vergelijken.
Zij a en b booleans. Je kan logische uitspraken combineren aan de hand van de volgende operatoren

```
a & b
a | b
!a
xor(a,b)
```

Al deze operatoren zijn tevens van toepassing op vectoren. Handige functies voor logische vectoren zijn any() en all(). any() geeft aan of er minstens 1 waarde waar is en all() geeft aan of alle waarden waar zijn. Ook de functie sum() is bruikbaar voor logische vectoren, deze telt dan hoeveel waarden van de vector gelijk zijn aan TRUE. De functie which() geeft aan welke posities van een logische vector gelijk zijn aan TRUE. Probeer te voorspellen wat het resultaat is van de volgende commando's:

```
a <- c(1,4,6,8,11)
b <- c(4,1,6,9,7)
all(a>b | a<b)
any(a<b & a>=b)
which(xor(a==b,a<=b))
```

Logische vectoren kunnen ook gebruikt worden om subvectoren/selecties te maken. Probeer te begrijpen hoe:

```
c(3,5,6,8)[c(TRUE,TRUE,FALSE,TRUE)]
(1:20)[c(TRUE,FALSE,FALSE)]
a[a>6]
b[a<b & b<8]
```

Merk op bij het tweede voorbeeld dat R hier opnieuw aan recycling doet, ondanks dat de lengtes van de vectoren geen veelvouden van elkaar zijn. Wees altijd kritische tegenover je eigen code en output, fouten in de code zullen soms onopgemerkt voorbij gaan.
Indien de lengte van de meegegeven logische vector langer is, dan zal R NA's geven voor de posities die hij niet kan opvragen.

Er zijn meerdere functies die testen of een object aan een bepaalde voorwaarde voldoet, met resultaat TRUE of FALSE:

- is.na()
- is.nan()
- is.finite()
- is.numeric()
- is.character()
- is.integer()

Test deze als je twijfelt over de functionaliteit.
Opmerking: plus en min oneindig kent R als Inf en -Inf.

Een handige functie die gebruik maakt van een logische uitdrukking, is de functie ifelse(). Deze heeft 3 argumenten nodig: een logische uitdrukking en 2 waarden.

```
b <- c(4,1,6,9,7)
ifelse(b>5,"groter dan 5", "kleiner of gelijk aan 5")
c <- -5:5
ifelse(c>=0,c,0)
```

Uitvoeringstijden

Men heeft de functionaliteiten van R opgebouwd met vectoren centraal, vandaar dat alle basisbewerkingen en meeste functies zeer gemakkelijk bruikbaar zijn voor vectoren. De interne omzetting naar machinetaal is eveneens zeer efficiënt voor bewerkingen op vectoren. De bewerkingen op elk element van de vector worden zo goed als parallel uitgevoerd, in verhouding komt er weinig uitvoeringstijd bij wanneer je de lengte van een vector opdrijft. Beschouw bijvoorbeeld de functie mean(), naïef zouden we verwachten dat R exact hetzelfde doet wanneer we deze schrijven via een for-lus, niets is minder waar. Vergelijk via de volgende code de uitvoertijden tussen beide methoden, vergroot n stelselmatig en bekijk het effect.

```
n <- 100000
a <- rnorm(n) #trekking van n random observaties uit een standaardnormale verdeling

ptm <- proc.time() #proc.time() vraagt de tijd op
som <- 0
for(i in 1:n){
  som <- som+a[i]
}
som/n
proc.time()-ptm

ptm <- proc.time()
mean(a)
proc.time()-ptm
```

Conclusie: vermijd het gebruik van for-lussen indien mogelijk! Dit zal zeer merkbaar zijn in simulatiestudies of bij het verwerken van big data. In latere practica en in projecten zullen we frequent steunen op simulatiestudies, het is dus belangrijk dat je efficiënt leert programmeren in R. Dit kan het verschil betekenen tussen een uitvoertijd van een minuut en een uitvoertijd van enkele uren.

Oefeningen: deel 1

Los alle oefeningen op zonder het gebruik van for-lussen.

1. Maak een vector `vecA` die de nummers 100 t.e.m. 1000 bevat, in stappen van 25.
2. Maak een vector `vecB` met dezelfde lengte en die loopt van 50 tot 5000 in gelijke stappen.
3. Bereken de volgende getallen of vectoren. A_i stelt het i-de getal voor van vector vecA en analoog voor B_i. Laat n de lengte voorstellen van de vectors. Sla de oplossing van elke opgave op, bv als oplossing.a, oplossing.b, ...
   (a) $\sqrt{(A_i^2 + B_i^2)}$ voor $i \in [1, n]$
   (b) $\sqrt{\frac{\sum_{i=1}^{n}(A_i^2 + B_i^2)}{n}}$
   (c) $e^{\left(\frac{A_i - B_i}{A_i + B_i}\right)}$ voor $i \in [1, n]$
   (d) $\sum_{i=5}^{20} ln(|A_{i-1} - A_i|)$
4. Rond de oplossing van d af op 3 cijfers na de comma.
5. Rond de oplossing van d af op 3 beduidende cijfers.
6. Rond de oplossing van b af op 10-tallen.
7. Maak een vector met de natuurlijke getallen wiens kwadraat groter is dan 570 en kleiner dan 9650.
8. Voor welke indices van `vecA` is de waarde groter dan 800 of de wortel kleiner dan 15?
9. Geef alle natuurlijke getallen (≤ 100) waarvoor de rest na deling door 2, 3 en 7 gelijk is aan 1, 0 en 2.

Matrices

```
ea <- 1:10
ea.mat <- matrix(ea,nrow = 2)
ea.mat
ea.mat[1,2] #eerste rij, tweede kolom
ea.mat[,1] #de ganse eerste kolom
```

Matrices zijn speciale vectoren in R, ze hebben een extra dimensie. Matrices creëren in R kan via de functie `matrix`, deze vergt van input een vector die de elementen voorstellen van je matrix en het aantal rijen en/of het aantal kolommen. Standaard worden de elementen van de matrix kolom per kolom gevuld, tenzij je de optie `byrow` op `TRUE` zet.
Elementen, rijen, kolommen en deelmatrices kunnen opgevraagd worden via vierkante haakjes. Hier komen in volgorde de argumenten van de rijen en kolommen, gescheiden door een komma.

```
str(ea.mat)
ea.mat[4:6]
```

Zoals je kan zien, kan je matrices nog steeds behandelen als een normale vector. Die vector geeft de matrix kolom per kolom.

```
ea.mat[,c(3,5)] <- -(1:4)
ea.mat
```

Merk op dat je de elementen meteen kan aanpassen wanneer je deze opvraagt.

```
ea.mat -(1:5) #matrices optellen (met recycling!)
ea.mat * 2 #vermenigvuldigen met een scalair
ea.mat * 2:3 #deze vermenigvuldiging wordt een vermenigvuldiging van 2 vectoren
ea.mat %*% (1:5) #matrixvermenigvuldiging
t(ea.mat)

#Probeer de output van het volgende commando te voorspellen!
t(ea.mat) %*% matrix(c(1,3),nrow=2,ncol=2,byrow=TRUE)
```

Matrices kunnen opgeteld, afgetrokken, vermenigvuldigd en getransponeerd worden. Merk op dat de normale regels voor recycling blijven gelden en dat resultaten zullen versimpeld worden indien nodig (bv van matrix naar vector). Een inverse matrix kan via de functie `solve()`.

Matrices met compatibele dimensies kunnen aan elkaar geplakt worden via de functies `rbind()` en `cbind()`.

```
rbind(ea.mat, 11:15)
cbind(ea.mat, 100, ea.mat)
```

Ook de functies `rownames()`, `colnames()`, `rowSums()`, `colSums()`, `rowMeans()` en `colMeans()` kan je gebruiken op matrices. Hun functionaliteiten zijn voor de hand liggend, probeer ze uit op voorbeeldmatrices indien je toch twijfelt.
Merk op dat `rownames()` en `colnames()` niet enkel dienen om informatie op te vragen!

```
rownames(ea.mat) <- c("eerste rij", "tweede rij")
```

De functie **outer()** is een zeer handige functie om uitproducten te berekenen. Deze is zeer efficiënt geïmplementeerd.

```
outer(1:10,1:5,FUN="+")
```

Arrays

Indien je vectoren in meer dimensies dan twee wil opslaan, kan je terecht bij `Arrays`. Deze werken gelijkaardig aan matrices. Bijvoorbeeld

```
test.array <- array(1:(2*3*4),dim=c(2,3,4))
test.array
```

De arrays worden net zoals matrices standaard kolom per kolom opgevuld. Deelvectoren of matrices worden op een gelijkaardige manier opgevraagd en alle matrix-bewerkingen gelden tevens voor deze objecten.

```
test.array[,,1]%*%t(test.array[,,2])
```

Packages

Gebruikers kunnen hun eigen pakketten aanmaken met onder andere zelfgedefinieerde functies in.
Deze pakketten kan je installeren via `install.packages()`. Deze kunnen dan in het werkgeheugen geladen worden via de functie `library()`. Merk op dat wanneer je via athena werkt, de meeste packages al geïnstalleerd zijn en dus enkel in het werkgeheugen moeten geladen worden. Beschouw bijvoorbeeld het package `dplyr`. Dit pakket is geschreven om data-manipulatie gemakkelijker te maken. Het bezit ook enkele andere handige functies, zoals bijvoorbeeld de functie `near()`.

```
# install.packages("dplyr")
# install.packages ENKEL INDIEN JE R EN RSTUDIO OP JE EIGEN PC GEBRUIKT
library(dplyr)
help(package="dplyr")
?near
sqrt(2)^2==2
near(sqrt(2)^2,2)
```

Lists en data frames

```
lijst <- list("text",ea.mat,5)
lijst
lijst[[1]] #eerste object uit de list
lijst[c(1,2)] #deellijst
lijst[1] #nog steeds een lijst!

class(lijst)
str(lijst)

names(lijst) <- c("text","matrix","getal")
lijst
lijst$matrix
lijst$text
```

Lijsten kunnen meerdere objecten met verschillende classes omvatten. Diens elementen kunnen opgevraagd worden met dubbele vierkante haken. Indien je enkele vierkante haken gebruikt, zal

een deellijst opgegeven worden. Men kan de elementen van deze lijst ook opvragen aan de hand van het dollarteken indien we de elementen een naam geven.

Een data frame is een speciale lijst, deze bevat enkel vectoren die allen even lang zijn. M.a.w. kan je een data frame in matrix-vorm uitdrukken, maar in tegenstelling tot matrices, kunnen de kolommen van type verschillen. Data frames kunnen aangemaakt worden via de functie `data.frame`. Ook kan je matrices omzetten naar data frames via de functie `as.data.frame`.

```
test.df <- data.frame(volgnummer = 1:10, pariteit = c("oneven", "even"),
 score=c(1,5,10,4,6,7,7,4,8,3))
test.df
names(test.df)
test.df$score
test.df$score[c(5,7)] <- c(7,6) #aanpassen scores van volgnummer 5 en 7

test.df[c(1,3)]
test.df[c("pariteit","score")]
test.df[[2]]
test.df[["score"]]
test.df[2,3]

subset(test.df, pariteit=="even", select=-pariteit)
subset(test.df, score %in% 7:10) # variabele %in% vector geeft logische vector

as.data.frame(ea.mat)
```

Merk op dat data frames kunnen behandeld worden als matrix en als lijst.

Andere handige functies voor dataframes te manipuleren zijn `aggregate()` en `gather()`. De functie `aggregate()` dient om een samenvattende statistiek op te vragen voor een variabele volgens bepaalde subgroepen. Het eerste argument is de lijst van variabelen waarvan je de statistiek wil berekenen, het tweede argument is de lijst van variabelen waarvoor de observaties opgedeeld moeten worden en het argument FUN is de naam van de functie die toegepast moet worden. Het resultaat zal opnieuw een data frame zijn met de groeperende variabelen en statistieken van de eerste variabele. Het is ook mogelijke de eerste twee argumenten als formule te geven (bijvoorbeeld $a \sim b + c$, hier zal voor elke verschillende combinatie van waarden voor b en c de functie berekend worden voor a).

```
aggregate(score ~ pariteit, FUN=mean, data=test.df)
#OF
aggregate(test.df["score"],test.df["pariteit"],FUN=mean)
# merk op dat je in de laatste vorm de variabelen als lijst moet meegeven!
```

De functie aggregate zal later handig zijn voor datavisualisaties. Zo ook de functie `gather()`. Deze zit bevat in het pakket `tidyr`. We zullen het nut van deze functie uitleggen aan de hand van een voorbeeld. Stel dat we de volgende dataset beschikbaar hebben:

```
library(tidyr)
regen.df <- data.frame(dag=1:10,
 Brugge=rnorm(10,3),
```

```
 Gent=rnorm(10,2),
 Aalter=rnorm(10,2.5))
regen.df
```

Gegeven is de hoeveelheid regen in Brugge, Gent en Aalter op 10 opeenvolgende dagen. Stel dat we de plaats als variabele willen hebben, dan kan dit via de functie `gather()`.

```
gather(regen.df, plaats, regen, c(Brugge,Gent,Aalter))
```

Het eerste argument is de data frame, het tweede argument is de naam van de nieuwe categorische variabele, het derde argument is de naam van de waarden en het vierde argument zijn de 'variabelen' die gecombineerd moeten worden. Zoals je ziet, worden de namen van de oude variabelen gebruikt als waarden voor de categorische variabele.

Zoals eerder aangehaald, bevat het pakket `dplyr` functies voor datamanipulatie. Een introductie tot dit pakket vind je hier: `https://cran.r-project.org/web/packages/dplyr/vignettes/dplyr.html`.

Factors

Een factor is een speciaal soort character-variabele gebruikt voor categorische data, deze heeft verschillende niveaus (al dan niet geordend). Een voorbeeld van een niet-geordende factor is een keuze van behandeling (behandeling A, behandeling B en placebo/geen behandeling), we kunnen niet zeggen dat $A > B$ of $B > A$. Een voorbeeld van een geordende factor is het hoogst behaalde diploma (diploma lagere school, diploma middelbaar, bachelordiploma, ...). Hier kan je zeggen dat een diploma middelbaar beter is dan een diploma van de lagere school, maar indien we deze hercoderen naar (1, 2, 3, ...), hebben de onderlinge verschillen geen betekenis.

```
dat <- factor(sample(c("A","B","P"),50,replace = T))
dat
table(dat)
```

De functie `table()` geeft een tabel weer van discrete data. Zo kan je ook data frames (of delen hiervan) meegeven voor hoger dimensionale tabellen.

```
df <- data.frame(behandelingsgroep=dat,
         overleven=factor(sample(c("Ja","Nee"),50,replace = T)))
table(df)
```

Alternatief zouden we de verschillende behandelingsgroepen kunnen coderen als 0, 1 en 2.

```
factor(sample(0:2,50, replace=T),levels=0:2,
            labels=c("placebo", "behandeling A","behandeling B"))
```

Op de volgende manier kan je een geordende factor aanmaken:

```
sam <- factor(sample(c("klein","middelmatig","groot"),50,replace = TRUE))
sam
sam <- ordered(sam, levels=c("klein","middelmatig","groot"))
sam
# OF
factor(sample(c("klein","middelmatig","groot"),50,replace = TRUE),
 levels=c("klein","middelmatig","groot"), ordered=TRUE)
```

De functies `sort()` (resp. `order()`) geeft de gesorteerde vector weer (resp. een vector met de gersorteerde volgorde). Deze functies werken tevens ook voor numerieke variabelen.

Werkgeheugen

Je work directory kan je opvragen en aanpassen met de functies `getwd()` en `setwd()`. Hier zullen je bestanden opgeslagen en opgevraagd worden indien je geen specifiek directory path meegeeft. Als je wil weten welke elementen zich bevinden in je work directory, kan dit via `dir()`. Merk op dat je dezelfde functionaliteiten terugvindt onder het tabblad `Files` van het scherm rechtsonder.
Objecten opslaan, kan via de functie `save()`. Indien je het hele werkgeheugen wil opslaan, gebruik je `save.image()`. Deze objecten in het geheugen laden kan via de functie `load()`.
Een lijst van wat er in je werkgeheugen zit, kan je opvragen via de functie `ls()`. Een object verwijderen uit het werkgeheugen kan via de functie `rm()`.

```
getwd()
setwd("H:/statistische gegevensanalyse)
dir()

ls()
save(test.df, file = "Mijn_dataframe.RData")
rm(test.df)
load("Mijn_dataframe.RData")
test.df

save.image(file= "alles.RData")
rm(list=ls())
ls()
load("alles.RData")
ls()
```

Data inlezen

Data laden die niet in RData-formaat zijn opgeslagen, kan via de functie `read.table()`. Deze zal een data frame construeren via de inputfile. Bekijk de verschillende argumenten van deze functie op de hulppagina. Probeer hiermee de data in te lezen van het bestand `cars.txt` (te vinden op Minerva) in je werkgeheugen.

```
help(read.table)
auto <- read.table("cars.txt",header = T)
class(auto)
str(auto)
```

De functie `read.table()` kan veel verschillende data lezen, gegeven dat de argumenten correct gegeven zijn. Er zijn afgeleide functies die specifiek voor één datatype bedoeld zijn, bijvoorbeeld `read.csv()` en `read.csv2()` voor CSV-bestanden. Deze hebben andere basiswaarden voor sommige argumenten.

```
teacher <- read.table("teachers_part.csv", header=TRUE,
  sep = ";", dec =".", comment.char = "")
teacher2 <- read.csv("teachers_part.csv")
all(teacher == teacher2)
```

Om je data te verkennen, kan je gebruik maken van de volgende functies:

```
View(teacher)
dim(teacher)
nrow(teacher)
ncol(teacher)

head(auto)
tail(auto)
summary(auto)
```

In de nieuwere versies van RStudio kan je data gemakkelijker inlezen aan de hand van de `Import Dataset`-knop in het scherm rechtsboven. Je kiest het type bestand, geeft de url/directory pad van het bestand en vervolgens kan je opties aanpassen en meteen een voorbeeld zien van hoe je data frame eruit zal zien. Probeer het bestand `teachers_part.csv` op deze manier in te lezen.

Data genereren, kwantielen, kansen en kansdichtheid

In R kan je ook gemakkelijk kwantielen en kansen bereken van bepaalde distributies of data genereren die van die distributie afkomstig zijn. Voor bekende discrete verdelingen zoals de binomiale verdeling, geometrische verdeling en Poisson-verdeling, kan je dit opzoeken via `?Binomial, ?Geometric` en `?Poisson`. Voor bekende continue verdelingen zoals de normale verdeling, exponentiële verdeling, uniforme verdeling, t-verdeling, gammaverdeling, Chi-kwadraatverdeling en F-verdeling, kan je terecht bij `?Normal, ?Exponential, ?Uniform, ?TDist, ?GammaDist, ?Chisquare` en `?FDist`.

Zo zal bijvoorbeeld de volgende code 100 datapunten genereren, waarvoor het i-de datapunt een verwachtingswaarde van i heeft en een standaarddeviatie van alternerend 1 of 2 heeft.

```
set.seed(123) #Deze seed zorgt ervoor dat iedereen dezelfde waarden zal genereren
rnorm(100, 1:100, c(1,2))
```

Het berekenen van kwantielen, kansen en de kansdichtheid zullen we duiden aan de hand van een binomiale verdeling met 10 worpen en kans 0.5 op succes en een standaardnormale verdeling.

```
#Wat is de kans dat je 4 keer munt gooit bij 10 worpen met een eerlijk muntstuk?
0.5^10*choose(10,4)
dbinom(4, 10, 0.5)

#Wat is de kans dat je minder dan 5 keer munt gooit?
0.5^10*sum(choose(10,0:4))
pbinom(4, 10, 0.5)

#Wat is de kans dat je maximaal 6 keer munt gooit?
pbinom(6, 10, 0.5)

#Wat is de kans dat je minstens 4 keer munt gooit?
0.5^10*sum(choose(10,4:10))
1-pbinom(3, 10, 0.5) #of
pbinom(3, 10, 0.5, lower.tail=FALSE)
```

```
#Hoe vaak zal je munt maximaal gooien in 80% zekerheid (dus met minstens 80% kans)?
qbinom(0.8, 10, 0.5)
# controle:
pbinom(6, 10, 0.5)
pbinom(0:10, 10, 0.5)

#Hoe vaak zal je munt minstens gooien met 90% zekerheid (dus met minstens 90% kans)?
qbinom(0.9, 10, 0.5, lower.tail=FALSE)
# controle:
1-pbinom(3,10,0.5) # Kleiner dan 90%!
1-pbinom(2,10,0.5) # Het antwoord is dus 2!


#Wat is de kansdichtheid voor de waarde 0.5 voor een standaardnormale variabele?
dnorm(0.5)

#Geef een 95% referentie-interval voor een standaardnormale variabele.
(Een 95% referentie-interval voor X is een zo klein mogelijk interval waar
de waarde van X in minstens 95% van de gevallen toe behoort.
Een referentie-interval is tweezijdig tenzij anders vermeld.)
alpha <- (1-0.95)/2
c(qnorm(alpha),qnorm(1-alpha))
#Kan ook korter aangezien de standaardnormale verdeling symmetrisch is rond 0:
c(-1,1)*qnorm(1-alpha)
# controle:
pnorm(qnorm(1-alpha))-pnorm(qnorm(alpha)) # inderdaad 95%

#Geef een 80% referentie-interval voor het aantal keer munt uit 10 worpen.
alpha <- (1-0.8)/2
a <- qbinom(alpha,10,0.5)
a <- ifelse(pbinom(a,10,0.5)==0.1,a+1,a)
b <- qbinom(1-alpha,10,0.5)
c(a,b)
pbinom(b,10,0.5)-pbinom(a-1,10,0.5)
```

Merk op dat het niet voor de hand liggend is dat het berekende interval effectief het gezochte referentie-interval is, er is namelijk niet noodzakelijk een waarde a zodat $\mathrm{P}(X \leq a) = 10\%$ en analoog voor b.
Als $X \sim B(10, 0.5)$ zoeken waarden a (zo groot mogelijk) en b (zo klein mogelijk) zodat

$$\begin{cases} \mathrm{P}(X < a) \leq 10\% \\ \mathrm{P}(X > b) \leq 10\% \end{cases}$$

want dan $\mathrm{P}(a \leq X \leq b) \geq 80\%$.
Equivalent zoeken we de kleinste waarden a en b waarvoor

$$\begin{cases} \mathrm{P}(X \leq a) > 10\% \\ \mathrm{P}(X \leq b) \geq 90\% \end{cases}$$

Functies

Tot hiertoe hebben we al kennisgemaakt met meerdere functies. R laat het toe je eigen functies te definiëren. Dit kan handig zijn als je eenzelfde "script"zou moeten uitvoeren op verschillende objecten. Om gebruiker-gedefinieerde functies te maken, volg je het volgende stramien:

```
myfunction <- function(arg1, arg2, ... ){
statements
return(object)
}
```

Stel dat we bijvoorbeeld een functie willen maken die de verschillen berekent tussen opeenvolgende getallen van een vector:

```
verschil <- function(vector){
l <- length(vector)
return(vector[2:l]-vector[1:(l-1)])
}

#verschillen tussen opeenvolgende kwadraten
verschil((1:10)^2)
```

Indien jouw functie bestaat uit een reeks berekeningen aan de hand van een gegeven waarde, is het meestal nuttig dat jouw functie ook werkt voor vectoren. Dit voor het vermijden van for-lussen.
Stel bijvoorbeeld dat je een functie wil schrijven die de poolcoördinaten berekent gegeven de carthesische coördinaten. De omzettingsformules zijn gegeven door:

- $r \in [0, +\infty[$ en $\theta \in [0, 2\pi[$ (resp. $[0, 360[$) met θ uitgedrukt in radialen (resp. graden)
- $r = \sqrt{x^2 + y^2}$ en $\theta = \arctan(\frac{y}{x}) + \pi \cdot I(x < 0) + 2\pi \cdot I(y < 0 \text{ en } x \geq 0)$
- $(x, y) = (r\cos(\theta), r\sin(\theta))$.

Een goede manier om deze functie te implementeren is dan als volgt:

```
poolc <- function(x,y){
  r <- sqrt(x^2+y^2)
  theta <- atan(y/x) + pi*(x<0) + 2*pi*(y<0 & x>=0)
  return(cbind(r,theta))
}
```

Probeer alle delen van deze functie te begrijpen en overtuig jezelf ervan dat deze code werkt. Test deze bijvoorbeeld voor verschillende inputwaarden. Bijvoorbeeld:

```
poolc(-1,0)
poolc(c(2,1,0,-1),c(0,0,1,0))
```

Is het moeilijk om de functie gevectorizeerd te implementeren, kan men deze alsnog achteraf vectorizeren via de functie `Vectorize()`.
Beschouw bijvoorbeeld de volgende functie:

$$f(x \mid a_1, a_2, \cdots, a_n) = \prod_{k=1}^{n} (x - a_k)$$

Een naïeve implementatie levert

```
productfunctie <- function(x,a){
return(prod(x-a))
}
```

De functie werkt perfect zolang we slechts 1 x-waarde meegeven, maar doet niet wat we willen wanneer een vector van x-waarden wordt meegegeven. Waarom?

```
productfunctie(2,c(1,0))
productfunctie(c(1,2),c(1,0))
productfunctie(c(1,2,3),c(1,0))
```

De functie `Vectorize()` heeft de te vectorizeren functie en een vector met de namen van de te vectorizeren argumenten nodig. Bijvoorbeeld

```
vecprodfunc <- Vectorize(productfunctie, vectorize.args=c("x"))
vecprodfunc(2,c(1,0))
vecprodfunc(c(1,2),c(1,0))
vecprodfunc(c(1,2,3),c(1,0))
```

Vraag je jezelf af hoe je een bepaald object of functie kan recreëren, dan kan je de functie `dput()` gebruiken. Bijvoorbeeld

```
dput(vecprodfunc)
dput(test.df)
```

Het is echter belangrijk te beseffen dat Vectorize de te vectorizeren functie niet noodzakelijk op de meest **efficiënte** manier zal vectorizeren! (Zie oefening deel 2.)

Beschrijvende statistieken

Hieronder staan enkele handige functies die je helpen continue variabelen van een steekproef te analyseren. Gebruik de hulppagina's indien je niet zeker bent over de functionaliteit.

```
set.seed(123)
x <- rnorm(50)
mean(x)
summary(x)
median(x)
max(x)
min(x)
sd(x)
var(x)
IQR(x)
```

Voor covarianties en correlaties kan je terecht bij functies `cov()` en `cor()`.

```
y <- rnorm(50)
test <- data.frame(x,y)
cov(test) #we zien: var(x)=0.857, cov(x,y)=-0.030 en var(y)=0.820
var(test)
var(x,y)
cor(test)
```

Stel dat je functies wil toepassen voor meerdere kolommen, rijen, groepen, ..., dan kan dit via de functies `apply, sapply, tapply, lapply,` ... Stel bijvoorbeeld dat we het product willen berekenen van elke rij of kolom van een matrix, dan gebruiken we de functie `apply`:

```
apply(ea.mat,1,prod)
apply(ea.mat,2,prod)
```

Merk op dat je hier ook zelf gemaakte functies aan kan meegeven. De argumenten van deze functies worden gewoon meegegeven aan de functie `apply`.

```
apply(eamat,2,vecprodfunc,a=c(0,1))
```

Wil je eenzelfde functie toepassen op elk element van een lijst, dan gebruik je de functie `lapply`. Deze geeft een lijst van gelijke lengte terug met alle uitkomsten. `sapply` werkt gelijkaardig, maar zal indien mogelijk de output vereenvoudigen naar een vector of matrix.
Wil je een statistiek bepalen per deelgroep, dan kan dit via `tapply`

```
tapply(test.df$score, test.df$pariteit, mean)
```

Meer informatie over het gebruik van deze functies, vind je hier: `https://www.guru99.com/r-apply-sapply-tapply.html`.

Oefeningen: deel 2

1. Bepaal de kans dat een Poisson(50)-verdeelde variabele groter of gelijk is aan 100.
2. Geef de mediaanwaarde van een $F_{4,5}$-verdeelde variabele.
   (De mediaanwaarde van een variabele X is de waarde c zodat $\mathrm{P}(X \leq c) = 50\%$.)
3. Geef een 95% referentie-interval voor een χ^2_5-verdeelde variabele.
4. Geef een 80% referentie-interval voor een Poisson(3)-verdeelde variabele.
5. Schrijf een functie die voor een Poisson(λ)-verdeelde variabele een $(1-2\alpha)100\%$-referentie-interval geeft ($\alpha \in]0, 0.5[$). Gebruik deze om een 80% referentie-interval te bepalen voor alle $\lambda \in \{0.5, 1, \cdots, 5\}$.
6. Herneem de dataset *teacher* (van het bestand *teachers_part.csv*).
   (a) Bepaal de correlatie tussen variabelen *weight* en *height*.
   (b) Maak een frequentietabel van het aantal kinderen per persoon.
   (c) Bepaal het gemiddelde gewicht voor de personen met kinderen en de personen zonder kinderen.
   (d) Bepaal de standaarddeviatie voor de personen die minstens 1.80m zijn en de personen die kleiner zijn dan 1.80m.
7. Zij X een $n \times p$-matrix en Y een vector van lengte n. Zij

$$\hat{\beta} = \left(X^T X\right)^{-1} X^T Y,$$

   schrijf dan een functie *beta* die gegeven matrix X en vector Y (met correcte dimensies) de vector $\hat{\beta}$ berekent. Vergeet niet de uitvoer terug te transformeren naar een vector (`as.vector`).
   Controleer je functie met $X = matrix(c(rep(1, 5), 1 : 5), ncol = 2)$ en $Y = 6 : 10$, de uitvoer moet gelijk zijn aan de vector $(5, 1)$.

8. Stel $Z \sim N(0,1)$ (lees: Z is normaal verdeeld met verwachtingswaarde 0 en variantie 1) en $Y \mid Z = z \sim N(a + bz, 1)$ (lees: voor gegeven Z is Y normaal verdeeld met verwachtingswaarde $a + bZ$ en variantie 1). Genereer 10000 matrices met 100 rijen observaties van de vorm $(Y, 1, Z)$ voor het geval dat $a = 3$ en $b = 2$ sla deze op in de array *my.array*.

9. Schrijf een functie *beta2* die voor gegeven matrix van voorgaande vorm de functie *beta* uitvoert met Y de eerste kolom en X de matrix zonder de eerste kolom.
   Voer deze *beta2* uit op elke matrix van *my.array*. (hint: apply)

10. De uitvoer van voorgaande operatie is een 2×1000 matrix met schattingen voor a en b, de eerste rij zijn schattingen voor a en de tweede rij zijn schattingen voor b. Maak een dataframe aan met variabelen a en b met als inhoud de voorgaande schattingen en sla deze dataframe op in een RData-bestand.

11. Zoals op het einde van het hoofdstuk over functies vermeld, zal *Vectorize* de te vectorizeren functie niet altijd het meest efficiënt implementeren. Schrijf een functie met behulp van *outer* die sneller werkt dan *vecprodfunc* voor de volgende berekening.

    ```
    system.time(vecprodfunc(1:100000,1:10))
    ```