Übung 1

Aufgaben 1 bis 7

[redacted]

10.10.2022

Aufgabe 1:

Man sollte sich über verschiedene Funktionen und Befehle in R informieren:

1. ***Befrehl [***

```
help("[")
```

```
## starting httpd help server ... done
```

- Eine kurze Recherche mithilfe vom Befehl *help*, zeigt alle Anwendungsfälle von "[". Dieser Operator wird auf Vektoren, Matrizen, Arrays und Listen angewendet, um auf spezielle Einträge explizit zuzugreifen oder diese mit anderen zu ersetzen.

2. ***ordered Argument für factor***

- Erstens müssen wir verstehen was Faktoren sind. Faktoren stellen Datenobjekte dar, die dabei helfen, Daten in Kategorien aufzuteilen und auf zugehöriger Ebene zu speichern. Factoren werden mittels Funktion **factor( )** erzeugt. Argument *ordered* dient daher dazu, diese Ebenen (eng. *levels*) aufsteigend zu ordnen. Folgendes Beispiel veranschaulicht dieses Verhalten:

```
size = c("small", "large", "medium", "large")
#ungeordnet
size_factor <- factor(size)
print(size_factor)
```

```
## [1] small  large  medium large
## Levels: large medium small
```

```
#geordnet
ordered.size <- factor(size, levels = c("small", "medium", "large"), ordered = TRUE)
print(ordered.size)
```

```
## [1] small  large  medium large
## Levels: small < medium < large
```

3. **Funktion, die normalverteilte Daten erzeugt**

- In R gibt es 4 vordefinierten Funktionen, die normalverteilte Daten erzeugen können und zwar: ***dnorm(x, mean, sd), pnorm(x, mean, sd), qnorm(p, mean, sd), rnorm(n, mean, sd)***. Dabei stellt x das ganze Datenset dar, mean das arithmetische Mittel und sd die Standardabweichung bzw. Streuung. Bei Funktion ***qnorm*** ist p Vektor bestehend aus Wahrscheinlichkeiten und bei ***rnorm*** ist n die Anzahl der Werte in der Stichprobe.

4. **Funktion für bedingte logistische Regression**

- Google-Suche weist uns auf Funktion *clogit* hin. Um diese Funktion verwenden zu können muss Library "Survival" geladen werden.

```
library(survival)
help(clogit)
```

- Nach Laden dieser Library können wir diese Funktion an den *help*-Befehl übergeben, wobei im *help-Fenster* folgender Inhalt erscheint:

Conditional logistic regression

Description

Estimates a logistic regression model by maximising the conditional likelihood. Uses a model formula of the form `case.status~exposure+strata(matched.set)`. The default is to use the exact conditional likelihood, a commonly used approximate conditional likelihood is provided for compatibility with older software.

Usage

```
clogit(formula, data, weights, subset, na.action,
 method=c("exact", "approximate", "efron", "breslow"),
 ...)
```

Arguments

`formula`	Model formula
`data`	data frame
`weights`	optional, names the variable containing case weights
`subset`	optional, subset the data
`na.action`	optional na.action argument. By default the global option `na.action` is used.
`method`	use the correct (exact) calculation in the conditional likelihood or one of the approximations
`...`	optional arguments, which will be passed to `coxph.control`

Figure 1: "help-Fenster" nach der Ausführung von "help(clogit)"

5. **Vignetten in R**

- Vignetten in R sind ein Bestandteil von R Packages. Diese Packages bestehen aus verstiedensten Funktionen, die den Nutzer:innen beigebracht und veranschaulicht werden müssen. Dafür sind Vignetten am besten geeignet. RStudio erleichtert das, indem es uns anbietet, diese über Hauptmenü zu erstellen: File -> New File -> R Markdown -> From template -> Package Vignette.

- Jede Vignette besteht aus folgenden drei Teilen: Original Source File, lesbare HTML-Seite oder PDF-Dokument und R-Datei mit R-Code.

6. **Vignetten verfügbar auf meiner R-Installation**

   Die einzige Vignette verfügbar auf meiner R-Installation ist ***Package Vignette***, die über oben angeführte Zeile zugegriffen werden kann.

Aufgabe 2:

```
r <- 1.08
n <- 10
i <- 1 : n
#Summe, wenn n = 10
sum(r^i)
```

```
## [1] 15.64549
```

```
(r^(n+1) - 1)/(r - 1)- 1
```

```
## [1] 15.64549
```

```
n <- 20
i <- 1 : n
#Summe, wenn n = 20
sum(r^i)
```

```
## [1] 49.42292
```

```
(r^(n+1) - 1)/(r - 1)- 1
```

```
## [1] 49.42292
```

```
n <- 30
i <- 1 : n
#Summe, wenn n = 30
sum(r^i)
```

```
## [1] 122.3459
```

```
(r^(n+1) - 1)/(r - 1)- 1
```

```
## [1] 122.3459
```

```
n <- 40
i <- 1 : n
#Summe, wenn n = 40
sum(r^i)
```

```
## [1] 279.781
```

```r
(r^(n+1) - 1)/(r - 1)- 1
```

```
## [1] 279.781
```

```r
#Ergebnisse für alle n zwischen 1 und 100 in einem vector gespeichert
n <- 100
x <- numeric(n)
for(i in 1:n)
{
  x[i]<- sum(r^i)
}
x
```

```
##   [1]    1.080000    1.166400    1.259712    1.360489    1.469328    1.586874
##   [7]    1.713824    1.850930    1.999005    2.158925    2.331639    2.518170
##  [13]    2.719624    2.937194    3.172169    3.425943    3.700018    3.996019
##  [19]    4.315701    4.660957    5.033834    5.436540    5.871464    6.341181
##  [25]    6.848475    7.396353    7.988061    8.627106    9.317275   10.062657
##  [31]   10.867669   11.737083   12.676050   13.690134   14.785344   15.968172
##  [37]   17.245626   18.625276   20.115298   21.724521   23.462483   25.339482
##  [43]   27.366640   29.555972   31.920449   34.474085   37.232012   40.210573
##  [49]   43.427419   46.901613   50.653742   54.706041   59.082524   63.809126
##  [55]   68.913856   74.426965   80.381122   86.811612   93.756540  101.257064
##  [61]  109.357629  118.106239  127.554738  137.759117  148.779847  160.682234
##  [67]  173.536813  187.419758  202.413339  218.606406  236.094918  254.982512
##  [73]  275.381113  297.411602  321.204530  346.900892  374.652964  404.625201
##  [79]  436.995217  471.954834  509.711221  550.488119  594.527168  642.089342
##  [85]  693.456489  748.933008  808.847649  873.555461  943.439897 1018.915089
##  [91] 1100.428296 1188.462560 1283.539565 1386.222730 1497.120549 1616.890192
##  [97] 1746.241408 1885.940720 2036.815978 2199.761256
```

Aufgabe 3:

```r
print("Vector x1:")
```

```
## [1] "Vector x1:"
```

```r
x1 <- rep(c(1, 3, 6, 10, 15), 1:5)
x1
```

```
##  [1]  1  3  3  6  6  6 10 10 10 10 15 15 15 15 15
```

```r
print("Vector x2:")
```

```
## [1] "Vector x2:"
```

```r
x2 <- seq(length = 15, from = 0, to = 1)
x2
```

```
##  [1] 0.00000000 0.07142857 0.14285714 0.21428571 0.28571429 0.35714286
##  [7] 0.42857143 0.50000000 0.57142857 0.64285714 0.71428571 0.78571429
## [13] 0.85714286 0.92857143 1.00000000
```

```
print("Vector x3:")
```

```
## [1] "Vector x3:"
```

```
x3<- seq(from = 0, to = 1, by = 0.05)
x3
```

```
##  [1] 0.00 0.05 0.10 0.15 0.20 0.25 0.30 0.35 0.40 0.45 0.50 0.55 0.60 0.65 0.70
## [16] 0.75 0.80 0.85 0.90 0.95 1.00
```

```
print ("Vector x4:")
```

```
## [1] "Vector x4:"
```

```
x4 <- rep(1, times = 15)
x4
```

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

```
print("Vector x5:")
```

```
## [1] "Vector x5:"
```

```
x5 <- rep(c(1L,2L,1L), times = 5)
x5
```

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

```
print("Vector x6:")
```

```
## [1] "Vector x6:"
```

```
x6 <- rep(c(1,2), times = c(7, 8))
x6
```

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

```
print("Vector x7:")
```

```
## [1] "Vector x7:"
```

```r
arr <- (14:1)
x7 <- numeric(15)
x7[1] <- 15
for(i in 2:15){
  x7[i] = arr[i - 1] + x7[i - 1]
}
x7
```

```
##  [1]  15  29  42  54  65  75  84  92  99 105 110 114 117 119 120
```

```r
print("Vector x8:")
```

```
## [1] "Vector x8:"
```

```r
x8 <-cumprod(1:15)
x8
```

```
##  [1] 1.000000e+00 2.000000e+00 6.000000e+00 2.400000e+01 1.200000e+02
##  [6] 7.200000e+02 5.040000e+03 4.032000e+04 3.628800e+05 3.628800e+06
## [11] 3.991680e+07 4.790016e+08 6.227021e+09 8.717829e+10 1.307674e+12
```

```r
print("Vector x9:")
```

```
## [1] "Vector x9:"
```

```r
x9 <- rev(x7)
x9
```

```
##  [1] 120 119 117 114 110 105  99  92  84  75  65  54  42  29  15
```

```r
print("Vector x10:")
```

```
## [1] "Vector x10:"
```

```r
x10 <- rep(c("a", "b", "c", "d"), times = c(1, 4, 8, 2))
x10
```

```
##  [1] "a" "b" "b" "b" "b" "c" "c" "c" "c" "c" "c" "c" "c" "d" "d"
```

```r
print("Vector x11: Faktor von x5:")
```

```
## [1] "Vector x11: Faktor von x5:"
```

```r
x11 <- factor(x5, ordered = TRUE)
x11
```

```
##  [1] 1 2 1 1 2 1 1 2 1 1 2 1 1 2 1
## Levels: 1 < 2
```

```r
print("Faktorenvektor basierend auf x10:")
```

```
## [1] "Faktorenvektor basierend auf x10:"
```

```r
x12 <- factor(x10, levels = c("a", "b", "c", "d", "e"), ordered = TRUE)
print(x12)
```

```
##  [1] a b b b b c c c c c c c c d d
## Levels: a < b < c < d < e
```

```r
print("Vector x13")
```

```
## [1] "Vector x13"
```

```r
x13 <- factor(as.character(x2), levels = c("less", "sufficient", "good", "planty"), ordered = TRUE)
print(x13)
```

```
##  [1] <NA> <NA> <NA> <NA> <NA> <NA> <NA> <NA> <NA> <NA> <NA> <NA> <NA> <NA> <NA>
## Levels: less < sufficient < good < planty
```

Aufgabe 4:

```r
1L == "1"
```

```
## [1] TRUE
```

Dieser Vegleich gibt true zurück, da folgende Coercion-Ordnung gegeben ist: logical > integer > double > character. Daher wird unser Integer-Wert zu einem Charakter. 1L wird daher zu "1".

```r
-2 < FALSE
```

```
## [1] TRUE
```

Logische Werte werden in Zahlen umgewandelt. 0 steht für FALSE und 1 für TRUE. In diesem Beispiel hätten wir dann -2 < 0, was jedenfalls stimmt. Daher wird TRUE ausgegeben.

```r
"one" < 2
```

```
## [1] FALSE
```

Hier kann kein Vergleich durchgefügrt werden. Einerseits haben wir links ein Wort stehend und andererseit auf der rechten seite eine Zahl. Wenn die Zahl 2 in den Charackter-Typ umgewandelt wird, kann der Vergleich nur in **ASCII**-Code stattfinden. Der erste Buchstabe von "one" ist "o" mit ASCII code 79, ASCII-Code von 2 ist 50. 79 ist größer 50, daher kommt **FALSE** raus.

```r
a <- c(TRUE, TRUE, TRUE)
b <- c(1, 2, 3) #true true true
print(b-2) # -1 0 1
```

```
## [1] -1  0  1
```

```r
as.logical(b - 2)
```

```
## [1]  TRUE FALSE  TRUE
```

```r
a & (b - 2)
```

```
## [1]  TRUE FALSE  TRUE
```

Die Ausgabe liefert TRUE FALSE TRUE, da Elemente von A mit Elementen von B logisch gebunden werden. Im ersten Schritt wird von jedem Eintrag in B 2 abgezogen, was -1 0 1 ergibt. Diese Werte kann man als logische Werte TRUE FALSE TRUE interpretieren. Dann werden Elemente von A und B paarweise logisch gebunden: TRUE&TRUE, TRUE&FALSE, TRUE&TRUE, was im Endeffekt ***True, False, True*** ergibt.

```r
x <- -7
x > 0 & sqrt(x) < 2
```

```
## Warning in sqrt(x): NaNs produced
```

```
## [1] FALSE
```

```r
x <- -7
x > 0 && sqrt(x) < 2
```

```
## [1] FALSE
```

Beide Code-Chunks geben FALSE zurück, da wir Wurzel aus einer negativen Zahl ziehen, was für einen NAN-Wert auf der rechten Seite der Gleichung sorgt. Links von & haben wir eine Null stehen, die logisch als FALSE interpretiert wird. Diese zwei logischen Werte "und-verknüpft" ergeben wiederum FALSE, weil NAN-Wert nicht beachtet wird.

Beide Operatoren & und && stehen für logisches "und". Der einzige Unterschied ist, dass die längere Form von links nach rechts ausgewertet wird, bis Ergebnis ausgerechnet wird. Operator && soll auf Vektoren mit mehr als einem Element angewendet werden. Sonts gibt es eine Warnung oder eine Fehlermeldung, falls davor die IDE-Variable _R_CHECK_LENGTH_1_LOGIC2_ auf true gesetzt war. Das ist seit R 4.2.0 so und in der Zukunft sollte es default werden.

Aufgabe 5:

```r
set.seed(1)

DAT <- sample(LETTERS[1:7], 15, replace = TRUE)
F1 <- factor(DAT, levels = (LETTERS[1:7]))

F2 <- F1
levels(F2) <- rev(levels(F2))

F3 <- rev(factor(DAT, levels = (LETTERS[1:7])))

F4 <- factor(DAT, levels = rev(LETTERS[1:7]))

F1
```

```
##  [1] A D G A B E G C F B C C A E E
## Levels: A B C D E F G
```

```r
F2
```

```
##  [1] G D A G F C A E B F E E G C C
## Levels: G F E D C B A
```

```r
F3
```

```
##  [1] E E A C C B F C G E B A G D A
## Levels: A B C D E F G
```

```r
F4
```

```
##  [1] A D G A B E G C F B C C A E E
## Levels: G F E D C B A
```

- Funktion set.seed() wird in R verwendet, um reproduzierbare Ergebnisse zu erzeugen, wenn man Code schreibt, in dem Varablen erstellt werden, die zufällige Werte annehmen. Argument dieser Funktion ist eine beliebige Zahl.
- DAT ist ein Vektor die mittels **sample** Funktion erzeugt wurde. Das heißt, es wurde ein Sample von der ersten 7 Buchstaben erzeugt, dessen Länge 15 ist und in dem Elemente ersetzt werden (replace = true). Ansonsten sind die Positionen der Elemente und wie oft jedes Element vorkommt reiner Zufall.
- F1 ist ein Faktor, wobei jeder der ersten sieben Buchstaben eine eigene Ebene (Level) darstellt.
- F2 wird danach mit F1 initialisiert. D.h. die beiden haben zumal gleichen Inhalt.
- Ebenen von F2 werden dann die Ebenen von F1 zugewiesen aber in verkehrter Richtung. Statt A, B, C, D, E, F, G, sind die Ebenen von F2 **G, F, E, D, C, B, A.**
- F3 wird Faktor von DAT aber in der umgekehrter Reihenfolge, also invertiert.
- F4 hat den gleichen Inhalt wie F1, aber hier wurde bei dem Argument Levels Funktion levels = **rev**(LETTERS[1:7]) verwendet, was bedeutet, dass die Ebenen (Levels) von F4 gleich wie bei F1 sind aber invertiert von hinten nach vorne.
- F1 und F3 haben gleiche Ebenen (Levels), sprich deren Ordnung ist gleich. Doch sind die Elemente von F1 und F4 verkehrt herum eingefügt. Diese beide Faktoren sind Invertierungen von einander.
- F2 und F4 haben auch gleiche Ebenen,. F1 und F4 haben gleiche Elemente, aber deren Ebenen sind zu einander invert.

Aufgabe 6:

a) Was gibt Funktion dim zurück, wenn sie auf einen atomic Vektor angewendet wird? Was passiert, wenn dim-Attribut einer Matrix bzw. eines Arrays mit NULL installisiert wird?

- Falls dim auf einen atomic Vektor angewendet wird, bekommt dann der Vector Eigenschaften eines multidimensionalen Arrays. Argument von dim muss ein Integer-Vektor sein (oder ein Vektor mit jenen Werten, die in Integer-Werten umgewandelt werden können). Falls kein Wert der **dim**-Funktion übergeben wird oder eine negative Zahl, kann dass nicht akzeptiert werden. Falls der **dim**-Funktion NULL übergeben wird, wird Attribut entfernt.

b)

```
set.seed(1234)
#help("scale")
mat <- rnorm(300, mean = c(1, 2, 3), diag(x = c(1,2,3)))
dim(mat) = c(100, 3)
#Benennung der Zeilen und Spalten
dimnames(mat)[[1]] <- paste0('r', 1:100)
dimnames(mat)[[2]] <- paste('c', 1:3)
mat
```

```
##               c 1         c 2        c 3
## r1   -0.207065749  2.00000000  3.0000000
## r2    2.000000000  3.00000000  1.0000000
## r3    3.000000000  1.00000000  4.7356544
## r4    1.000000000 -1.25818694  3.0000000
## r5    2.554858484  3.00000000  1.0000000
## r6    3.000000000  1.00000000  2.0000000
## r7    1.000000000  2.00000000  6.9886944
## r8    2.000000000 -0.50285779  1.3364728
## r9    6.253323530 -1.18003965  2.0000000
## r10  -1.345697703  2.00000000  3.0000000
## r11   2.000000000  3.00000000  1.0000000
## r12   3.000000000  1.00000000  2.0137857
## r13   1.000000000 -0.68198638  3.0000000
## r14   2.858249378  3.00000000  1.0000000
## r15   3.000000000  1.00000000  2.0000000
## r16   1.000000000  2.00000000  1.6335938
## r17   2.000000000  2.11711842  0.6334761
## r18   4.518167676  0.53410246  2.0000000
## r19   0.425260040  2.00000000  3.0000000
## r20   2.000000000  3.00000000  1.0000000
## r21   3.000000000  1.00000000  3.2965731
## r22   1.000000000  4.89899253  3.0000000
## r23   0.906736288  3.00000000  1.0000000
## r24   3.000000000  1.00000000  2.0000000
## r25   1.000000000  2.00000000  9.2108126
## r26   2.000000000 -0.20592817  0.8466016
## r27   1.306644003  0.14463537  2.0000000
## r28   0.109962171  2.00000000  3.0000000
## r29   2.000000000  3.00000000  1.0000000
## r30   3.000000000  1.00000000 -0.7814019
```

```
## r31   1.000000000  1.43875400  3.0000000
## r32   1.045614600  3.00000000  1.0000000
## r33   3.000000000  1.00000000  2.0000000
## r34   1.000000000  2.00000000  0.8292547
## r35   2.000000000  0.01697977  1.2582618
## r36   0.004840665  0.03148568  2.0000000
## r37   0.223746105  2.00000000  3.0000000
## r38   2.000000000  3.00000000  1.0000000
## r39   3.000000000  1.00000000  1.3658818
## r40   1.000000000 -0.21463639  3.0000000
## r41   2.128917635  3.00000000  1.0000000
## r42   3.000000000  1.00000000  2.0000000
## r43   1.000000000  2.00000000  2.4666301
## r44   2.000000000 -0.75595766  0.8300059
## r45   5.878482177  0.47617188  2.0000000
## r46   0.889714506  2.00000000  3.0000000
## r47   2.000000000  3.00000000  1.0000000
## r48   3.000000000  1.00000000 -0.7446038
## r49   1.000000000  1.00630009  3.0000000
## r50   0.977980988  3.00000000  1.0000000
## r51   3.000000000  1.00000000  2.0000000
## r52   1.000000000  2.00000000  2.4786385
## r53   2.000000000 -2.41809377  1.8502323
## r54   0.266413750  0.41792408  2.0000000
## r55   0.162828320  2.00000000  3.0000000
## r56   2.000000000  3.00000000  1.0000000
## r57   3.000000000  1.00000000  3.3952174
## r58   1.000000000 -0.21777925  3.0000000
## r59   6.831670357  3.00000000  1.0000000
## r60   3.000000000  1.00000000  2.0000000
## r61   1.000000000  2.00000000  4.6499921
## r62   2.000000000 -0.04488603  0.5972680
## r63   3.402264660  0.83769048  2.0000000
## r64   0.509314103  2.00000000  3.0000000
## r65   2.000000000  3.00000000  1.0000000
## r66   3.000000000  1.00000000  1.6168125
## r67   1.000000000  3.12611164  3.0000000
## r68   1.118904255  3.00000000  1.0000000
## r69   3.000000000  1.00000000  2.0000000
## r70   1.000000000  2.00000000 -0.5835836
## r71   2.000000000  7.94345242  0.9468412
## r72   4.378768323  0.22664658  2.0000000
## r73   0.306279753  2.00000000  3.0000000
## r74   2.000000000  3.00000000  1.0000000
## r75   3.000000000  1.00000000  2.5103920
## r76   1.000000000  5.21181926  3.0000000
## r77  -0.896409821  3.00000000  1.0000000
## r78   3.000000000  1.00000000  2.0000000
## r79   1.000000000  2.00000000  8.1178920
## r80   2.000000000 -0.47342564  2.0015133
## r81   4.724267163  1.65658846  2.0000000
## r82  -0.023655723  2.00000000  3.0000000
## r83   2.000000000  3.00000000  1.0000000
## r84   3.000000000  1.00000000  1.0088331
```

```
## r85   1.000000000  7.09798214  3.0000000
## r86   1.969723399  3.00000000  1.0000000
## r87   3.000000000  1.00000000  2.0000000
## r88   1.000000000  2.00000000  4.0666509
## r89   2.000000000  2.89571883 -0.1346080
## r90   0.192154196  0.33036642  2.0000000
## r91   2.102297546  2.00000000  3.0000000
## r92   2.000000000  3.00000000  1.0000000
## r93   3.000000000  1.00000000  3.7564073
## r94   1.000000000  1.98479049  3.0000000
## r95   1.048813842  3.00000000  1.0000000
## r96   3.000000000  1.00000000  2.0000000
## r97   1.000000000  2.00000000  5.9187503
## r98   2.000000000  8.33125334  3.1211171
## r99   0.871679887 -0.13860774  2.0000000
## r100  0.498741939  2.00000000  3.0000000
```

- Zu rechnen war:

```r
X1 <- scale(mat, TRUE, TRUE)
X2 <- scale(mat, TRUE, FALSE)
X3 <- scale(mat, FALSE, TRUE)
X4 <- scale(mat, FALSE, apply(mat, 2, sd))
#X1
#X2
#X3
#X4
```

- Man muss erstmal verstehen wofür die Argumente der ***scale*** Funktion stehen. Nämlich wird zuerst ein Vektor, bzw. eine Matrix, übergeben, der/die zu skalieren ist. Zweiter Argument ist "***center***" und beantwortet die Frage, ob das arithmetische Mittel subtrahiert werden soll (TRUE ist angenommener Wert). Driter Argument ist "***scale***" und gibt an, ob es mit Standardabweichung dividert werden soll (TRUE ist angenommen, wenn kein Parameter übergeben wurde).

- X3 und X4 sind nicht gleich, da bei X4 statt einem logischen Wert (TRUE/FALSE) eine ***apply-*** Funktion übergeben wurde, die für ***jede Spalte*** Streuung berechnet. Was insgesamt **drei numerische** Werte zurückgibt. Es wird bei X3 **scale** auf **TRUE** gestetzt. Diese drei numerischen Werte können insgesamt nicht als WAHR angenommen werden, obwohl einzelne Werte vielleicht TRUE sein mögen (alle Streuungen sind >= 0).

  c) Was passiert, wenn as.matrix auf ein Dataframe angewandet wird, dessen Spalten jeweils einen anderen Typ haben?

- Es wird eine Character-matrix zurückgeliefert, falls die aus keinen nicht-numerichen/logischen oder komplexen Spalten sondern nur **atomic** Spalten besteht.Sonst (im Falle, wenn Matrix auch logische, komplexe oder numeriche Werte hat), wird normale Coercion-Ordnung verwendet (logical < integer < double < complex). Alle Frames, die nur aus logischen Werten bestehen, werden zu einer logical-Matrix zusammengefasst. Gemischte logical-integer-Matrizen ergeben integer-Matrix usw.

Aufgabe 7:

a)

```r
set.seed(1234)
x <- c(1, 3, 4)
y <- sample(1:100, 10)
z <- union(x, y)
x
```

```
## [1] 1 3 4
```

```r
y
```

```
##  [1] 28 80 22  9  5 38 16  4 86 90
```

```r
print("Vereinigung von x und y:")
```

```
## [1] "Vereinigung von x und y:"
```

```r
z
```

```
##  [1]  1  3  4 28 80 22  9  5 38 16 86 90
```

```r
inter <- intersect(x, y)
print("Durchschnitt von x und y:")
```

```
## [1] "Durchschnitt von x und y:"
```

```r
inter
```

```
## [1] 4
```

```r
print("Elemente in y, aber nicht in x:")
```

```
## [1] "Elemente in y, aber nicht in x:"
```

```r
setdiff(y, x)
```

```
## [1] 28 80 22  9  5 38 16 86 90
```

```r
#Untersuchung, ob Jedes element aus x in y gefunden werden kann:
for(i in x){
 print(i %in% y)
}
```

```
## [1] FALSE
## [1] FALSE
## [1] TRUE
```

b)

```r
s <- 1:200
subvec <- s[s%%7 == 0 & s%%2 != 0]
subvec
```

```
##  [1]   7  21  35  49  63  77  91 105 119 133 147 161 175 189
```

c)

```r
s <- 1L:200L
s7 <- s[s%%7 == 0]
s2 <- s[s%%2 != 0]
intersect(s7, s2)
```

```
##  [1]   7  21  35  49  63  77  91 105 119 133 147 161 175 189
```