Hausübung I

FoRt 2018/19

Fabian Scheipl

- Ihre Lösung muss den gesamten R-Code für die Beantwortung jeder Aufgabe enthalten sowie, wo nötig, ausformulierte Antworten.
- Benutzen Sie **keine externen Pakete** außer den in den Aufgabenstellungen angegebenen.
- Ihr Code muss die Richtlinien in Hadley Wickham's **style guide** befolgen und die in der Vorlesung besprochenen Prinzipien für sauberen, lesbaren & sinnvoll **dokumentierten** Code befolgen. Code in Ihrer Lösung wird mit dem Paket `lintr` auf sauberen Stil überprüft – das sollten Sie auch tun.
- Alle Fehlermeldungen die Ihr Code produziert sollten selbst verfasst sein (sprich: Fehler sein, die sie in Ihrem Code antizipiert haben).
- Die evtl. in der Angabe abgedruckten Grafiken müssen Sie nicht exakt reproduzieren, sie dienen primär dazu Ihnen zu ermöglichen die Korrektheit Ihrer Ergebnisse zu prüfen. Übereinstimmung von Farben, Achsenabschnitte, Legenden etc. Ihrer Lösung mit denen in der Angabe sind nicht Grundlage der Bewertung außer explizit in der Aufgabe angegeben.
- Speichern Sie Ihre Lösungen in **UTF8-Encoding** ab.
- Speichern Sie Ihre Lösung **unter den in der Angabe angebenen Dateinamen** ab. Ihre Abgabe wird automatisiert überprüft und dafür müssen die Dateinamen exakt so sein wie spezifiziert.
- Packen Sie alle Dateien die zu Ihrer Lösung gehören (falls nötig: mit einer `README`-Datei die erklärt welches File was tut) in *ein* `<Nachname>-<Vorname>.zip`, `.7z` oder `.tar.gz`-Archiv und laden Sie dieses Archiv auf der Moodle-Seite des Kurses als Hausaufgabe hoch.

Nur Lösungen, die vor dem Abgabedatum dort hochgeladen werden können gewertet werden. Die Bewertung berücksichtigt

1. ob sie einen sinnvollen, ernsthaften Versuch unternommen haben die jeweilige Teilaufgabe zu lösen.
2. ob ihr Code technisch korrekt und ausführbar ist.
3. ob ihr Code sauber entworfen (funktionale Kapselung, *DRY*, keine *code smells*, defensive Programmierung, etc.), sinnvoll kommentiert, gut lesbar und korrekt formatiert im Sinne des obigen Styleguides ist.

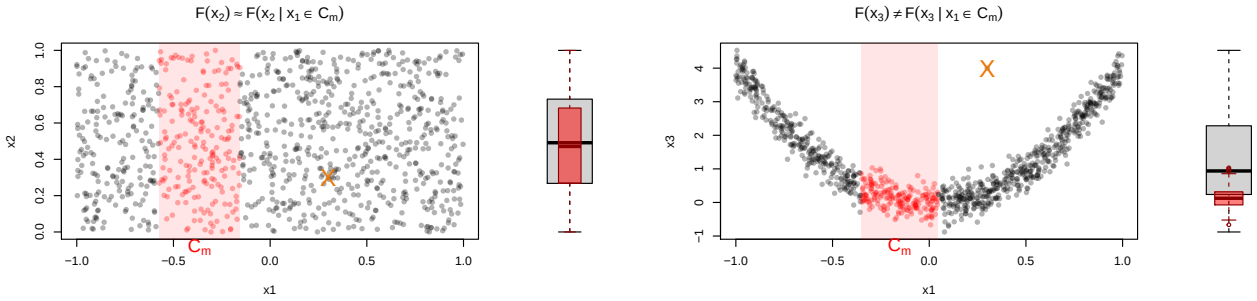

Figure 1: Links Unterraum $S = \{X_1, X_2\}$ mit niedrigem Kontrast (keine Abhängigkeit), rechts Unterraum $S = \{X_1, X_3\}$ mit hohem Kontrast (starke Abhängigkeit). Je ein Beispiel-Intervall C_m mit $\sqrt{\alpha} = 0.2$ in rot, Ausreisser-Wert in Orange. Graue Boxplots für marginale, unbedingte Verteilungen von X_2 (links) bzw. X_3 (rechts), rote Boxplots für bedingte Verteilungen gegeben $X_1 \in C_m$.

1: Spast mit Kontrast [50 Pkt.]

Wir entwerfen die Implementation (für einen Teil) des *HiCS*-Algorithmus (**Hi**gh **C**ontrast **S**ubspaces). Eine ausführliche Beschreibung dieses Algorithmus finden Sie hier.

Kurz zusammengefasst ist *HiCS* eine Methode mit der man in hochdimensionalen Datenräumen $X = (X_1, X_2, ..., X_p) \in \mathbb{R}^p$ niedrigdimensionalere Unterräume (in unserem Fall: zweidimensionale Unterräume (X_i, X_j)) identifizieren kann, in denen Abhängigkeitsstrukturen zwischen den Variablen vorliegen.
In solchen niedrigdimensionalen Unterräumen sind dann Ausreißer mit herkömmlichen Verfahren einfach zu identifizieren, was in hochdimensionalen Räumen aus Gründen[1] meist nicht der Fall ist. Figure 1 veranschaulicht die Grundidee des Algorithmus.

Der *HiCS*-Algorithmus nutzt aus, dass für unabhängige Zufallsvariablen X_i, X_j $F(X_i) = F(X_i|X_j)$ bzw. $F(X_j) = F(X_j|X_i)$ gilt. Um also interessante Unterräume $S = \{X_i, X_j\}$ mit möglichst stark abhängigen Variablen X_i, X_j zu identifizieren berechnet man für Variablen-Paare im Datensatz den "Abstand" zwischen der marginalen Verteilung $F(X_i)$ und der bedingten Verteilung $F(X_i|X_j)$ (bzw. zwischen $F(X_j)$ und $F(X_j|X_i)$).

Dieser "Abstand" $contrast(S)$ wird geschätzt in dem man für jeden Unterraum S M-mal die folgenden 3 Schritte wiederholt:

- auswürfeln auf welche der beiden Variable in $S = \{X_i, X_j\}$ bedingt werden soll,
- zufällig ein Intervall ("*slice*") $C_m = [l_m, u_m], m = 1, ..., M$ aus dem Wertebereich der bedingenden Variable auswählen welches einen vorher festgelegten Anteil $\sqrt{\alpha} \in (0, 1)$ der Daten enthält,
- Abweichung $d_m(S, C_m) = deviation(\hat{F}(x_i|x_j \in C_m), \hat{F}(x_i))$ berechnen (bzw. $deviation(\hat{F}(x_j|x_i \in C_m), \hat{F}(x_j))$, je nach Ergebnis in Schritt 1).

Der Subspace-Kontrast $contrast(S)$ ist dann der Mittelwert über die M berechneten Abweichungen:

$$contrast(S) = M^{-1} \sum_{m=1}^{M} d_m(S, C_m)$$

Als $deviation(\hat{F}(x_i|x_j \in C_m), \hat{F}(x_i))$ bietet sich "1 - p-Wert" von Tests auf Verteilungsgleichheit an, also z.B. die Kolmogoroff-Smirnov- oder Cramèr-von Mises-Tests für $H_0 : F(x_i) = F(x_i|x_j \in C_m)$.

---

[1] Der *curse of dimensionality* bewirkt u.a. dass alles von allem total weit weg ist, weil hochdimensionale Räume einfach so verdammt GROSS sind und distanz-basierte Verfahren zur Ausreißerentdeckung also versagen. Auch dichte-basierte Verfahren scheiden aus, da Dichten auf hochdimensionalen Trägern gar schwierig zu schätzen sind. Ausserdem sind Punkte, die nur im Bezug auf manche der vielen Dimensionen merkwürdig liegen schwierig zu entdecken...

Gesucht ist eine Funktion `get_contrasts` mit folgender Spezifikation:

```
# Computes 2D-subspace contrasts for (all, or at most <max_spaces>) pairs
# of numeric columns of <data>.
#
# Inputs:
# data: a matrix or a data.frame with at least 2 numeric columns
# spaces: (optional) 2 x <no. of subspaces> numeric matrix giving the pairs
#   of columns that define the subspaces for which the contrasts should be
#   computed. If not supplied, all possible subspaces (up to at most
#   <max_spaces>, randomly selected) are used.
# deviation: which distance measure for empirical cumulative distribution
#   functions to use. Either "ks" for 1 - p-value of the KS-Test
#   (default, see stats::ks.test), "cvm" for 1 - p-Value of
#   the Cramer-von Mises-Test (see goftest::cvm.test), or "tw"
#   for 1 - p-value of Welch's t-test (see stats::t.test) or a user-defined
#   function with an argument <conditional> which is a numeric
#   vector containing the observations in the conditioning slice and an argument
#   <marginal> which is a numeric vector containing all observations.
#   The function should return the dissimilarity of the two distributions, i.e.,
#   higher values for subspaces with stronger dependencies.
# slice: proportion of data to condition on, defaults to 0.2 (= square root of alpha).
# draws: how often to repeat the conditioning on random "slices" of the
#   conditioning variable, defaults to 100 (= M)
# max_spaces: how many 2D subspaces to investigate at most if
#   \code{spaces} is not supplied. Defaults to 4950, i.e. choose(100, 2),
#    the number of 2D subspaces for a 100D dataset.
# seed: (optional) random generator seed
#
# Output:
#   a \code{data.frame} with the column numbers of the investigated subspaces
#   in the first two columns (named "dim_1", "dim_2"") and the contrast values
#   in the third column (named "deviation"),
#   rows sorted so the highest contrast subspaces are in the first rows.
get_contrasts <- function(data, spaces = NULL, deviation = c("ks", "cvm", "tw"),
                          slice = 0.2, draws = 1e2, max_spaces = 4950,
                          seed = NULL) {
    library(checkmate)
    library(goftest)
    # TODO
}
```

a) [15 Pkt.] Skizzieren Sie, so wie wir das im Kurs am Beispiel der Stability Selection und des Promillerechners durchexerziert haben, eine "Top Down"-Lösung für den Entwurf der oben spezifizierten `get_contrasts` Funktion. Gehen Sie bei Ihrem Entwurf 2 "Level" tief, also zerlegen Sie den in `get_contrasts` zu implementierenden Algorithmus in Unterfunktionen/Schritte und die entsprechenen Unterfunktionen jeweils in ihre einzelnen Schritte bzw. Unter-Unterfunktionen. Dokumentieren Sie für jede Unter- und Unterunterfunktion die Aufgabe, die diese Funktion löst sowie die jeweiligen In- und Outputs ähnlich detailliert wie in der obigen Spezifikation von `get_contrasts`. **In dieser Aufgabe ist kein lauffähiger Code gefragt oder gewünscht**, sondern ein nachvollziehbarer, sauber dokumentierter und durchdachter Programmentwurf in Pseudo-Code, der den HiCS-Algorithmus in klar voneinander abgetrennte logische Schritte und Subroutinen unterteilt.
Speichern Sie Ihre Lösung unter `hics-design.R` oder `hics-design.Rmd` oder `hics-design.txt`.

b) [30 Pkt.] Implementieren Sie den in a) erarbeiteten Entwurf für `get_contrasts`. Benutzen Sie **keine**

Zusatz-Pakete außer `checkmate` und `goftest`. Benutzen Sie exakt die selbe Funktionsdefinition (Argumentnamen und Defaultwerte) wie in der Spezifikation der Aufgabe angeben.
Speichern Sie Ihre Lösung in einer Datei `hics-get-contrasts.R`.
Achten Sie dabei, wie immer, darauf

- die goldenen Regeln *Keep it simple!* und *Don't repeat yourself!* zu befolgen,
- die Prinzipien von Top-Down-Programmierung konsequent anzuwenden,
- lesbaren, klar strukturierten, sauber formatierten Code zu schreiben,
- mögliche Bedienerfehler und extreme Datensituationen zu antizipieren und durch entsprechendes *input checking* und defensives Programmieren abzufangen,
- *code smells* wie z.B. *conditional complexity* oder *speculative generality* zu vermeiden,
- ausreichend viel und sinnvoll zu kommentieren,
- einigermaßen effizienten Code zu schreiben.

Ihre Implementation muss (möglichst viele der) folgenden Test-Cases (s. `hics-get-contrasts-tests.R`) bewältigen:

```
set.seed(1212)
source("hics-get-contrasts.R")

#simple 2D datasets:
grid <- as.matrix(expand.grid(x = 1:10, y = 1:10)) + .1 * matrix(rnorm(200), 100, 2)
grid <- grid[sample(100),]
line <- cbind(x = 1:100, y = 1:100)
fuzzy_line <- line + 10 * matrix(rnorm(200), 100, 2)
examples <- cbind(line, fuzzy_line, grid)

#------------------------------------------------------------------------------
# CHECK: yield sensible results for defaults:
isTRUE(get_contrasts(grid)$deviation < .2)
isTRUE(get_contrasts(line)$deviation > .9)
isTRUE(get_contrasts(fuzzy_line)$deviation <
         get_contrasts(line)$deviation)

contrasts_examples <- get_contrasts(examples)
# all subspaces are covered:
nrow(contrasts_examples) == choose(6, 2)
# 1st column has smaller column numbers, 2nd the higher ones:
all(sort(unique(contrasts_examples[, 1])) == 1:(ncol(examples) - 1))
all(sort(unique(contrasts_examples[, 2])) == 2:ncol(examples))
# rows are sorted high to low according to deviation:
all(rank(contrasts_examples$deviation) == 15:1)
#"line" has highest contrast, "grid" has lowest contrast
all(contrasts_examples[1, 1:2] == c(1, 2))
all(contrasts_examples[15, 1:2] == c(5, 6))
# default deviation is 1 - p-value:
all(contrasts_examples$deviation > 0 & contrasts_examples$deviation < 1)

#------------------------------------------------------------------------------
# CHECK: yields reproducible results:
identical(get_contrasts(examples, seed = 12121),
          get_contrasts(examples, seed = 12121))

#------------------------------------------------------------------------------
# CHECK: cleans up inputs:
```

```
get_contrasts(cbind(grid, NA, 1), seed = 12121) # should trigger warning(s)
identical(get_contrasts(cbind(grid, NA, 1), seed = 12121),
          get_contrasts(grid, seed = 12121))

#------------------------------------------------------------------------------
# CHECK: accepts (and cleans up) data.frames:
df_examples <- cbind(as.data.frame(examples),
                     some_factor = gl(10, 10))
# this should give a warning:
get_contrasts(df_examples, seed = 12121)
# irrelevant columns are ignored/removed:
identical(get_contrasts(df_examples, seed = 12121),
          get_contrasts(examples, seed = 12121))

#------------------------------------------------------------------------------
# CHECK: implements different types of deviation measures correctly:
set.seed(122)
# three_d has strong correlation for (1,2), medium for (1,3), zero for (2, 3)
sigma <- matrix(c(1,.9, .4, .9, 1, 0, .4, 0, 1), 3, 3)
three_d <- mvtnorm::rmvnorm(500, sigma = sigma)
# same order of subspaces for different deviations:
!identical(get_contrasts(three_d, deviation = "cvm", seed = 12121),
           get_contrasts(three_d, deviation = "ks", seed = 12121))
identical(get_contrasts(three_d, deviation = "cvm", seed = 12121)[,1:2],
          get_contrasts(three_d, deviation = "ks", seed = 12121)[,1:2])
!identical(get_contrasts(three_d, deviation = "cvm", seed = 12121),
           get_contrasts(three_d, deviation = "tw", seed = 12121))
identical(get_contrasts(three_d, deviation = "cvm", seed = 12121)[,1:2],
          get_contrasts(three_d, deviation = "tw", seed = 12121)[,1:2])
# accepts user defined functions for deviation:
ks_user <- function(conditional, marginal) {
  1 - ks.test(conditional, marginal)$p.value
}
identical(get_contrasts(three_d, deviation = ks_user, seed = 12121),
          get_contrasts(three_d, seed = 12121))


#------------------------------------------------------------------------------
# CHECK: implements max_spaces arg correctly:
get_contrasts(examples, max_spaces = 2)
#above should give an informative warning that not all sub-spaces are explored.
isTRUE(nrow(get_contrasts(examples, max_spaces = 7)) == 7L)
#reproducible results:
identical(get_contrasts(examples, seed = 12121, max_spaces = 3),
            get_contrasts(examples, seed = 12121, max_spaces = 3))
highdim <- matrix(runif(1e4), ncol = 1e3, nrow = 10)
isTRUE(nrow(get_contrasts(highdim, draws = 1)) == choose(100, 2))

#------------------------------------------------------------------------------
# CHECK: implements spaces arg correctly:
identical(get_contrasts(three_d, spaces = combn(1:3, 2), seed = 12121),
          get_contrasts(three_d, seed = 12121))
identical(get_contrasts(three_d, spaces = rbind(2, 3))[1, -3],
```

```
          c(2, 3))
get_contrasts(three_d, spaces = cbind(c(3, 2), c(2, 3), c(2, 3)))
# above should give an informative warning...
# ... and remove redundant spaces:
identical(get_contrasts(three_d, spaces = rbind(2, 3), seed = 12121),
          get_contrasts(three_d, spaces = cbind(c(3, 2), c(2, 3), c(2, 3)),
                        seed = 12121))

#------------------------------------------------------------------------------
# FAILS: the calls below should all fail with INFORMATIVE, precise error messages
get_contrasts(as.character(grid))
get_contrasts(matrix(1, 10, 10))
get_contrasts(data.frame(gl(10, 10), gl(5, 20)))

get_contrasts(line, slice = "a")
get_contrasts(line, slice = 1*NA)
get_contrasts(line, slice = 0)
get_contrasts(line, slice = 2)
get_contrasts(line, slice = c(1, 2))

get_contrasts(line, draws = "a")
get_contrasts(line, draws = 1*NA)
get_contrasts(line, draws = 0.5)
get_contrasts(line, draws = c(1, 2))

get_contrasts(line, deviation = "sk")
bs <- function(x1, x2) {
  cat("bs!")
}
get_contrasts(line, deviation = bs)

get_contrasts(examples, max_spaces = "a")
get_contrasts(examples, max_spaces = NA)
get_contrasts(examples, max_spaces = -1)

get_contrasts(examples, spaces = expand.grid(1:2, 1:3))
get_contrasts(examples, spaces = combn(1:7, 2))
```

c) [5 Pkt.] Der HiCS-Algorithmus ist *embarrassingly parallel* – parallelisieren Sie Ihren Lösung aus b) unter Benutzung von `foreach`, `doParallel`, `doRNG` oder dem `future.apply` Paket ("oder", nicht "und"!). Auch diese Implementation – nennen Sie sie `get_contrasts_parallel()` – sollte natürlich alle obigen Tests bestehen.
Die Parallelisierung kann hier an unterschiedlichen Stellen ansetzen. Begründen Sie in entsprechenden Kommentaren im Code für welche Variante Sie sich warum entschieden haben.
Speichern Sie Ihre Lösung in einer Datei `hics-get-contrasts-parallel.R`.

Hinweis: Falls Sie in b) an der Implementation von $contrast(S)$ scheitern können Sie für $contrast(S)$ mit $S = (X_i, X_j)$ ersatzweise und zum Preis eines signifikanten Punktverlusts auch einfach die Spearman-Korrelation zwischen X_i und X_j benutzen.

2: F%in%stervalle [20 Pkt.]

In dieser Aufgabe implementieren wir neue Funktionalität für die im Paket `intervals` definierten S4-Klassen für Intervalle über $\mathbb{R}$ bzw. $\mathbb{Z}$.

Wir implementieren eine generische infix-Funktion `%in%` und entsprechende Methoden so, dass:

- das normale Verhalten von `%in%` für gewöhnliche *inputs*, wie es in der Dokumentation zu `?match` beschrieben wird, nicht beeinflusst wird.
- Aufrufe `x %in% y` für numerische vektoren `x` und Objekte `y`, die den durch `intervals` definierten S4-Klassen angehören, einen logischen Vektor zurückliefern der angibt ob die Werte in `x` in dem/den durch `y` definierten Intervall(en) liegen (s.a. Tests weiter unten in Teilaufgabe c).

Speichern Sie den Code für die Lösung aller 3 Teilaufgaben in einem File `intervals-in.R` ab.
Die Textantwort zu a) können Sie dort als Kommentar anlegen.

a) [3 Pkt] Die erste Funktions-Definition, die benötigt wird um die oben angegeben Spezifikation zu implementieren, ist:

```
`%in%` <- function(x, table) UseMethod("%in%", object = table)
```

Kopieren Sie obige Zeile in Ihre Lösungsdatei. Erläutern Sie in Ihren eigenen Worten was diese Zeile genau tut. Inwiefern weicht die obige Funktionsdefinition von der typischen Anwendung für `UseMethod` ab? (Warum) ist das hier nötig?

b) [5 Pkt] Schreiben Sie jetzt *eine* zusätzliche Funktion, die sicherstellt dass der gesamte Funktionsumfang des in `base` definierten `%in%`-Operators für entsprechende Inputs unverändert erhalten bleibt. Benutzen Sie den folgenden Code (s. `intervals-in-tests-1.R`) um Ihre Implementation zu überprüfen:

```
# TESTS: %in% for base-S3-objects

source("intervals-in.R")
# make base-version of %in% available for direct comparisons:
`%base_in%` <- base::`%in%`

all.equal(1:5 %in% c(1, 3, 5, 9),
          1:5 %base_in% c(1, 3, 5, 9))
all.equal(c("a", "b", "c") %in% c("a", "b"),
          c("a", "b", "c") %base_in% c("a", "b"))
all.equal(c("a", "b", "c") %in% list(c("a", "b", "c"), "a"),
          c("a", "b", "c") %base_in% list(c("a", "b", "c"), "a"))
```

c) [12 Pkt] Schreiben Sie jetzt zusätzliche Funktionen, so dass der folgende Test-Code (s. `intervals-in-tests-2.R`) die gewünschten Outputs erzeugt:

```
# TESTS: %in% for S4-intervals

source("intervals-in.R")
library(intervals)

#-------------------------------------------------------------------------------
# set up example data:

(ints <- Intervals(cbind(0:1, 1:2)))
(ints_open <- Intervals(cbind(0:1, 1:2), closed = FALSE))
(sets <- Intervals(cbind(c(-5, 0, 5), c(0, 5, Inf)),
                   closed = c(TRUE, FALSE), type = "Z"))
(empty <- Intervals(c(0, 0), closed = FALSE))
```

```r
(na <- Intervals(c(0, NA), closed = TRUE))

#------------------------------------------------------------------------------
#CHECK: basic functionality:

all.equal(1 %in% ints[1], TRUE)
all.equal(1 %in% ints_open[1], FALSE)
all.equal(NA %in% ints[1], NA)
all.equal(NA %in% na, NA)
all.equal(0 %in% na, TRUE)
all.equal(0 %in% empty, FALSE)

#------------------------------------------------------------------------------
#CHECK: vector inputs:

all.equal(c(-1, .5, 1, 0) %in% ints[1],
          c(FALSE, TRUE, TRUE, TRUE))
all.equal(c(-1, .5, 1, 0) %in% ints_open[1],
          c(FALSE, TRUE, FALSE, FALSE))
all(c(.5, 1.1) %in% ints)
all.equal(.5 %in% ints,
          c(TRUE, FALSE))
all.equal(5 %in% sets,
          c(FALSE, FALSE, TRUE))
all.equal(0.5 %in% sets,
          c(FALSE, FALSE, FALSE))

#------------------------------------------------------------------------------
#CHECK: dealing with unsuitable/weird inputs:

"a" %in% ints #should fail or return FALSE? explain pros & cons of your decision.

# The calls below should fail with informative error messages:
c(.5, 1.1, 2) %in% ints #should fail
matrix(0, 2, 2) %in% ints #should fail
list(0, 2, 2) %in% ints #should fail

#------------------------------------------------------------------------------
#CHECK: deal with open and closed intervals:

(ints_all <- c(ints, ints_open))

all.equal(1 %in% ints_all,
          c(TRUE, TRUE, FALSE, FALSE))
all.equal(c(0, 2, 0, 2) %in% ints_all,
          c(TRUE, TRUE, FALSE, FALSE))
all.equal(c(-1, 1, 0.5, 2) %in% ints_all,
          c(FALSE, TRUE, TRUE, FALSE))
```

3: *Wrecked & Tangled* [30 Pkt.]

In dieser Aufgabe implementieren wir eine neue `S4`-Klasse für achsenparallele Rechtecke, die auf die im Paket `intervals` definierten `S4`-Klassen für Intervalle aufsetzt.

a) [10 Pkt.] Implementieren Sie eine S4-Klasse `Rectangles` die (Mengen von) achsenparallelen Rechtecken auf $\mathbb{R} \times \mathbb{R}$ implementiert.
Jedes einzelne Rechteck wird intern repräsentiert durch 2 geschlossene `Intervals` über $\mathbb{R}$, welche die horizontale und vertikale Ausdehnung des Rechtecks definieren. Schreiben Sie eine Konstruktorfunktion `Rectangles()` und implementieren Sie alle nötigen Validitätsprüfungen für die Klasse.
Speichern Sie Ihre Lösung in einer Datei `intervals-rectangles-class.R`.
Überprüfen Sie Ihre Implementation (mindestens) mit untenstehendem Testcode (s. `intervals-rect-tests-1.R`):

```
library(intervals)
source("intervals-rectangles-class.R")

#-------------------------------------------------------------------------------
# CHECK:  Rectangles creates a Rectangles S4-object,
#         with slots x and y of class Intervals.
test_rect <- Rectangles(Intervals(c(0, 1)), Intervals(c(1, 2)))
is(test_rect, "Rectangles")
is(test_rect@x, "Intervals")
is(test_rect@y, "Intervals")
identical(test_rect@x@.Data, t(matrix(c(0, 1))))
identical(test_rect@y@.Data, t(matrix(c(1, 2))))

#-------------------------------------------------------------------------------
# CHECK: constructor works for Intervals & corresponding vector or matrix inputs:
unit_rect <- Rectangles(Intervals(c(0, 1)), Intervals(c(0, 1)))
identical(
  unit_rect,
  Rectangles(c(0, 1), c(0, 1)))
identical(
  Rectangles(Intervals(rbind(0:1, 1:2)), Intervals(rbind(0:1, 1:2))),
  Rectangles(rbind(0:1, 1:2), rbind(0:1, 1:2)))

#-------------------------------------------------------------------------------
# FAILS:
# The following calls to Rectangles should fail with INFORMATIVE & precise
#   error messages:
Rectangles(c(-1, -2), c(0, 1))
Rectangles(Intervals(c(0, 1), closed = FALSE), Intervals(c(0, 1)))
Rectangles(Intervals(c(0, 1), type = "Z"), Intervals(c(0, 1)))
Rectangles(Intervals(c(0, 1), type = "Z", closed),
           Intervals(c(0, 1), type = "Z"))
```

b) [7 Pkt.] Implementieren Sie `show` und `size` Methoden für Objekte der in a) implementierten S4-Klasse `Rectangles`.

1. `show` definiert die Textrepräsentation von S4-Objekten, orientieren Sie sich bei Ihrer Implementation an der `show`-Methode für die Klasse `Intervals_virtual`.
2. `size` berechnet die Flächeinhalte der in einem `Rectangles`-Objekt enthaltenen Rechtecke.

Speichern Sie Ihre Lösung in einer Datei `intervals-rectangles-methods.R`.
Überprüfen Sie Ihre Implementation (mindestens) mit untenstehendem Testcode (s. `intervals-rect-tests-2.R`):

```
library(intervals)
source("intervals-rectangles-class.R")
source("intervals-rectangles-methods.R")

#-------------------------------------------------------------------------------

rect <- Rectangles(
  x = rbind(c(0, 1), c(-1, 2), c(0, 1)),
  y = rbind(c(0, 2), c(1, 2), c(-Inf, Inf)))
rect_na <- Rectangles(c(0, NA), c(1, 2))
rect_line <- Rectangles(c(0, 0), c(1, 2))

#-------------------------------------------------------------------------------
# CHECK: show-method works as expected:
rect
```

```
## Object of class Rectangles with 3 rectangles:
## [0, 1] x [0, 2]
## [-1, 2] x [1, 2]
## [0, 1] x [-Inf, Inf]
```

```
#-------------------------------------------------------------------------------
# CHECK: size-method works as expected:
all.equal(size(rect), c(2, 3, Inf))
```

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

```
all.equal(size(rect_na), NA_real_)
```

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

```
all.equal(size(rect_line), 0)
```

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

c) [13 Pkt.] Implementieren Sie alles was nötig ist um eine Funktion `overlap` zu definieren, die für zwei `Rectangles`-Objekte elementweise die Überlappung der entsprechenden Einträge als neues `Rectangles`-Objekt zurückgibt. Achten Sie auch hier wieder auf das "DRY"- und "KIS"-Prinzip.
Wenn alle Ecken der Rechtecke bekannt sind ist $\text{overlap}(R_1, R_2) = R_1 \cap R_2$, für $R_i = [x_{i1}, x_{i2}] \times [y_{i1}, y_{i2}]$.
Komplikation: Wenn die Koordinaten der Ecken teilweise `NA`, also unbekannt, sind soll $\text{overlap}(R_1, R_2)$ die Teilmenge aus $\mathbb{R}^2$ definieren, die *sicher* $\subseteq R_1 \cap R_2$ ist. Die leere Menge wird durch 2 `NA` Intervalle repräsentiert. (Vgl. Testcode.)
Speichern Sie Ihre Lösung in einer Datei `intervals-rectangles-overlap.R`.
Überprüfen Sie Ihre Implementation (mindestens) mit untenstehendem Testcode (s. `intervals-rect-tests-3.R`):

```
library(intervals)
source("intervals-rectangles-class.R")
source("intervals-rectangles-methods.R")
source("intervals-rectangles-overlap.R")

#-------------------------------------------------------------------------------

rect <- Rectangles(
  x = rbind(c(0, 1), c(-1, 2), c(0, 1)),
  y = rbind(c(0, 2), c(1, 2), c(-Inf, Inf)))
rect_empty <- Rectangles(c(NA_real_, NA), c(NA_real_, NA))
```

```
#------------------------------------------------------------------------------
# CHECK: basic functionality
# ([0, 1]×[0, 2]) ∩ ([0, 2]×[1, 2]) = ([0, 1]×[1, 2])
identical(
  overlap(Rectangles(c(0, 1), c(0, 2)),
          Rectangles(c(0, 2), c(1, 2))),
  Rectangles(c(0, 1), c(1, 2)))
# ([0, 1]×[0, 1]) ∩ ([.3, .5]×[.3, .5]) = ([.3, .5]×[.3, .5])
identical(
  overlap(Rectangles(c(0, 1), c(0, 2)),
          Rectangles(c(.3, .5), c(.3, .5))),
  Rectangles(c(.3, .5), c(.3, .5)))
# ([0, 1]×[0, 2]) ∩ ([2, 3]×[4, 6]) = ∅
identical(overlap(Rectangles(c(0, 1), c(0, 2)),
                  Rectangles(c(2, 3), c(4, 6))),
          rect_empty)
# overlap is symmetric:
identical(
  overlap(Rectangles(c(0, 1), c(0, 2)),
          Rectangles(c(0, 2), c(1, 2))),
  overlap(Rectangles(c(0, 2), c(1, 2)),
          Rectangles(c(0, 1), c(0, 2))))
identical(
  overlap(Rectangles(c(0, 1), c(0, 2)),
          Rectangles(c(.3, .5), c(.3, .5))),
  overlap(Rectangles(c(.3, .5), c(.3, .5)),
          Rectangles(c(0, 1), c(0, 2))))
identical(
  overlap(Rectangles(c(0, 1), c(0, 2)),
          Rectangles(c(2, 3), c(4, 6))),
  overlap(Rectangles(c(0, 1), c(0, 2)),
          Rectangles(c(2, 3), c(4, 6))))

#------------------------------------------------------------------------------
# CHECK: basic functionality for sets of rectangles
identical(
  overlap(rect, rect),
  rect)
identical(
  overlap(Rectangles(c(-Inf, Inf), c(-Inf, Inf)), rect),
  rect)
identical(
  overlap(Rectangles(c(-Inf, Inf), c(-Inf, Inf)), rect),
  overlap(rect, Rectangles(c(-Inf, Inf), c(-Inf, Inf))))

#------------------------------------------------------------------------------
# CHECK: deals with NAs sensibly:
# returns the *smallest* rectangle that is guaranteed to be in the intersection:
# ([0, 1]×[0, 1]) ∩ ([?, .5]×[.5, 1]) ⊇ ([.5, .5]×[.5, 1])
identical(
  overlap(Rectangles(c(0, 1), c(0, 1)),
          Rectangles(c(NA, .5), c(.5, 1))),
          Rectangles(c(.5, .5), c(.5, 1)))
```

```
# ([0, 1]×[0, 2]) ∩ ([2, ?]×[1, 2]) = ∅ (no overlap at all in x direction)
identical(
  overlap(Rectangles(c(0, 1), c(0, 2)),
          Rectangles(c(2, NA), c(1, 2))),
  rect_empty)
# ([0, 1]×[0, ?]) ∩ ([?, .5]×[.5, 1]) = ∅ (!)
# possibly no overlap on y-axis, and then no overlap at all --> empty set
identical(
  overlap(Rectangles(c(0, 1), c(0, NA)),
          Rectangles(c(NA, .5), c(.5, 1))),
  rect_empty)

#-------------------------------------------------------------------------------
# FAILS:
# The following calls should fail with INFORMATIVE & precise error messages:
overlap(rect)
overlap(rect, cbind(c(0, 0),c(1, 1)))
overlap(c(0,1), cbind(c(0, 0),c(1, 1)))
overlap(rect,
        Rectangles(cbind(c(0, 0),c(1, 1)), cbind(c(0, 0),c(1, 1))))
```