Exercise 1

Exercises 1 to 7

11.10.2023

Contents

Task 1: Use the different help systems available for R (i.e., R homepage, built-in R functions) to find information about:

1. the command [
2. the ordered argument for factor
3. a function to create normally distributed random variables
4. a function that does conditional logistic regression
5. what a vignette is
6. which vignettes are on your R installation available, and access one of them

1. Command: [

To find information about the commands, you can use the built-in help function by placing a "?" in front of the command.

```
?`[`
```

```
## starte den http Server für die Hilfe fertig
```

This will display the documentation for the "[" command, which is used for vectors, matrices, arrays, and lists.

2. ordered Argument for factor

```
?factor
```

Example:

```
# Create a vector of nominal categories
categories <- c("Low", "Medium", "High", "Low", "High", "Medium")

# Create a factor with ordered levels
ordered_factor <- factor(categories, ordered = TRUE, levels = c("Low", "Medium", "High"))

# Print the factor
print(ordered_factor)
```

```
## [1] Low    Medium High   Low    High   Medium
## Levels: Low < Medium < High
```

In the R example, the "c" is used to create a vector.

c stands for "combine" or "concatenate" and is a function in R that is used to create vectors by combining or concatenating elements together. You can use c to create vectors from individual values, lists, or other vectors.

3. a function to create normally distributed random variables

```
?rnorm
```

Example:

```
# Normal distribution of 100 random numbers with a mean of 50 and standard deviation of 2
random_numbers <- rnorm(n = 30, mean = 50, sd = 2)

# print Summary statistics of the generated random numbers
summary(random_numbers)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##   46.56   48.38   49.48   49.98   51.15   55.25
```

```
print(random_numbers)
```

```
##  [1] 48.24577 51.11592 49.41490 50.92192 51.16679 48.03495 51.27153 49.22567
##  [9] 53.54891 47.61231 48.26206 48.89529 47.36885 49.24509 49.73873 49.30899
## [17] 51.28013 48.65160 48.29449 50.38639 51.81184 47.97349 48.92368 50.16423
## [25] 46.56371 54.61170 51.97656 49.53634 55.24643 50.56292
```

4. a function that does conditional logistic regression

```
library(survival)
?clogit
```

The clogit function in R performs conditional logistic regression, analyzing matched case-control data to assess the association between an outcome and one or more predictor variables while accounting for the matching structure.

5. explain what a vignette is

A vignette is a comprehensive, long-form document in R that provides detailed explanations, examples, and usage instructions for a specific package, tool, or topic, serving as a valuable resource for users to understand and effectively use the functionality provided.

6.which vignettes are on your R installation available, and access one of them

with the command "browseVignettes()" you can view all your installed Vignettes:

```
browseVignettes()
```

Task 2: Calculate the sum

```r
r <- 1.08
n <- 10

for (iteration in 1:4) {
  result_sum <- sum(r^(1:n))
  result_formula <- (r^(n+1) - 1)/(r - 1) - 1

  cat("Iteration", n, ": n =", n, "\n")
  cat("Result (Sum):    ", result_sum, "\n")
  cat("Result (Formula):", result_formula, "\n\n")

  n <- n + 10
}
```

```
## Iteration 10 : n = 10
## Result (Sum):     15.64549
## Result (Formula): 15.64549
##
## Iteration 20 : n = 20
## Result (Sum):     49.42292
## Result (Formula): 49.42292
##
## Iteration 30 : n = 30
## Result (Sum):     122.3459
## Result (Formula): 122.3459
##
## Iteration 40 : n = 40
## Result (Sum):     279.781
## Result (Formula): 279.781
```

The cat function in R is used to concatenate and print multiple expressions or strings to the console or a file.

Task 3: Create the following vectors:

```r
x1 <- rep(c(1, 3, 6, 10, 15), c(1, 2, 3, 4, 5))
cat("Vector 1: ", x1, "\n")
```

```
## Vector 1:  1 3 3 6 6 6 10 10 10 10 15 15 15 15 15
```

```r
x2 <- seq(0, 1, by = 1/14)
print(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
```

```r
x3 <- seq(0, 1, by = 0.05)
print(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
```

```r
x4 <- rep(1, times = 15)
cat("Vector 4: ", x4, "\n")
```

```
## Vector 4:  1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
```

```r
x5 <- rep(c(1, 2, 1), times = 5)
cat("Vector 5: ", x5, "\n")
```

```
## Vector 5:  1 2 1 1 2 1 1 2 1 1 2 1 1 2 1
```

```r
x6 <- rep(c(1, 2), each = 7)
cat("Vector 6: ", x6, "\n")
```

```
## Vector 6:  1 1 1 1 1 1 1 2 2 2 2 2 2 2
```

```r
x7 <- cumsum(c(15, 14:1)) #cumsum = cumulative sum
cat("Vector 7: ", x7, "\n")
```

```
## Vector 7:  15 29 42 54 65 75 84 92 99 105 110 114 117 119 120
```

```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
x9 <- rev(x7)
cat("Vector 9: ", x9, "\n")
```

```
## Vector 9:  120 119 117 114 110 105 99 92 84 75 65 54 42 29 15
```

```r
x10 <- rep(c("a", "b", "c", "d"), times = c(1, 4, 7, 2))
cat("Vector 10: ", x10, "\n")
```

```
## Vector 10:  a b b b b c c c c c c c d d
```

```r
x11 <- factor(x5, ordered=TRUE)
cat("Vector 11: ", x11, "\n")
```

```
## Vector 11:  1 2 1 1 2 1 1 2 1 1 2 1 1 2 1
```

```r
x12 <- factor(x10, levels = c("a","b","c","d","e"), ordered=TRUE)
cat("Vector 12: ", x12, "\n")
```

```
## Vector 12:  1 2 2 2 2 3 3 3 3 3 3 3 4 4
```

```r
print(x12)
```

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

```r
#x13:
breaks <- c(0, 0.37, 0.55, 0.96, 1)
labels <- c("less", "sufficient", "good", "plenty")

x13 <- cut(x2, breaks = breaks, labels = labels, ordered=TRUE)
cat("Vector 13: ", x13, "\n")
```

```
## Vector 13:  NA 1 1 1 1 1 2 2 3 3 3 3 3 3 4
```

```r
print(x13)
```

```
##  [1] <NA>       less       less       less       less       less
##  [7] sufficient sufficient good       good       good       good
## [13] good       good       plenty
## Levels: less < sufficient < good < plenty
```

Task 4: Explain

1. Explain the following:

Question	Answer
Why is 1L == "1" TRUE	The comparison 1L == "1" evaluates to TRUE in R because R performs type coercion in this case. 'L' indicates that 1L is an integer literal
Why is -2 < FALSE TRUE	The comparison -2 < FALSE evaluates to TRUE in R because of the way R handles comparisons between numeric and logical values. -2 is less than 0 (the numeric representation of FALSE), which is why -2 < FALSE evaluates to TRUE.
Why is "one" < 2 FALSE	The comparison "one" < 2 evaluates to FALSE in R because you are comparing a character string to a numeric value, and such comparisons are not well-defined in R. When comparing a character string to a numeric value, R typically returns NA (Not Available) because it recognizes that comparing different data types in this way doesn't have a clear meaning. In most cases, this results in a comparison returning FALSE or NA.

2. What does this code return and why?

```
a <- c(TRUE, TRUE, TRUE)
b <- c(1, 2, 3)
a & (b - 2)
```

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

- The vector a is defined as c(TRUE, TRUE, TRUE), which contains three logical TRUE values.
- The vector b is defined as c(1, 2, 3), which contains three numeric values.
- The expression b - 2 subtracts 2 from each element of the vector b, resulting in a new vector with the values (-1, 0, 1).
- The & operator is used to perform element-wise logical AND between the vectors a and (b - 2). It compares the corresponding elements in both vectors.
  - TRUE AND -1 results in **TRUE**.
  - TRUE AND 0 results in **FALSE**.
  - TRUE AND 1 results in **TRUE**.

3. What is the difference between the following chunks of code? Explain

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

```
## Warning in sqrt(x): NaNs wurden erzeugt
```

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

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

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

Explanation:

- square root of a negative number is not defined in the real numbers. Only if you explicitly declare it as a complex number, you won't get a warning.
- The && operator is used for the logical AND operation.
- In the first example, both conditions are evaluated, but the sqrt(x) < 2 part results in NA because the square root of a negative number is not a real number. The & operator combines the $x > 0$ result with the NA, resulting in NA.
- In the second snippet, the $x > 0$ condition is FALSE because -7 is not greater than 0. Since the first condition is FALSE, the && operator short-circuits and does not evaluate the sqrt(x) < 2 condition. The result is FALSE.

So, the key difference is that the & operator performs element-wise comparisons and may return NA, while the && operator avoids evaluating the second condition if the first condition is FALSE.

Task 5: Explain what the following code does:

```
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
```

```
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
```

```
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
```

```
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
```

Explanation:

1. set.seed(1): This sets the random number generator's seed to ensure reproducibility.
2. DAT <- sample(LETTERS[1:7], 15, replace = TRUE): Generating a 15 long vector by randomly selecting elements from the first 7 uppercase letters of the alphabet (A to G). With replacement (mit zurücklegen)
3. F1 <- factor(DAT, levels = (LETTERS[1:7])): creating a factor variable F1 from the DAT vector. It specifies the levels of the factor using the uppercase letters A to G.
4. F2 <- F1: creating a new factor variable F2 that is a copy of F1.
5. levels(F2) <- rev(levels(F2)): reversing the order of the levels in the factor F2. Reversed the levels from G-A
6. F3 <- rev(factor(DAT, levels = (LETTERS[1:7]))): creating a new factor variable F3 by reversing the order of the data. The levels are still the same.
7. F4 <- factor(DAT, levels = rev(LETTERS[1:7])): This line creates a new factor variable F4 directly, using rev to specify the levels in reverse alphabetical order.

Task 6:

1. dim function

When you apply the dim function to an atomic vector in R, it returns NULL. Atomic vectors are one-dimensional and do not have dimensions like matrices or arrays. Therefore, the dim function does not provide dimensions for atomic vectors.

If you assign the dim attribute of a matrix or an array the value NULL, it effectively removes the dimension attribute, and the object will behave like a one-dimensional vector.

For example, consider a matrix:

```
mat <- matrix(1:6, nrow = 2)
cat("The dimensions of mat are:", dim(mat), "\n")
```

```
## The dimensions of mat are: 2 3
```

```
cat("If you assign NULL to the dimension attribute:",dim(mat) <- NULL ,mat)
```

```
## If you assign NULL to the dimension attribute: 1 2 3 4 5 6
```

In this state, mat is no longer considered a matrix but rather a one-dimensional atomic vector.

2. scale function

```
set.seed(1234)

# Create the mean vector and variance-covariance matrix
mv <- c(1, 2, 3) #mv := mean vector
sigma <- diag(c(1, 2, 3))

# Generate random data using rnorm
mat <- rnorm(300, mean= mv, sigma)
dim(mat) = c(100,3)


# Set row and column names
rownames(mat) <- paste0("r", 1:100)
colnames(mat) <- paste0("c", 1:3)

mat
```

```
##                c1          c2         c3
## 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
```

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

x1
```

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

```
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
```

```
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
```

```
x4
```

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

Explanation:

1. Generating Random Data:

- mat <- rnorm(300, mean = mv, sigma): This line generates random data by sampling 300 values from a multivariate normal distribution with the specified mean vector mv and covariance matrix sigma. This creates a 300-element vector.
- dim(mat) = c(100, 3): This reshapes the 300-element vector into a 100x3 matrix, where each row represents an observation, and each column corresponds to a variable.

4. Setting Row and Column Names:

- rownames(mat) <- paste0("r", 1:100): This assigns row names to the matrix, where each row is labeled as "r1," "r2," and so on up to "r100."
- colnames(mat) <- paste0("c", 1:3): This assigns column names to the matrix, labeling the columns as "c1," "c2," and "c3."

3. as.matrix

The behavior of as.matrix depends on the data types present in the data frame.

- If the data frame contains a mix of logical, integer, double, and complex columns.
  Order: logical < integer < double < complex.
  It determines the common data type for the resulting matrix. Example: There are both integers and doubles in a data frame. It will result in a double matrix.
- If there are only only logical columns, it will be converted to a logical matrix.
- If the data frame contains of a mix of logical and integer columns, it will be converted to an integer matrix, and so on for other combinations of data types. This behavior is consistent with R's type coercion rules to ensure a common data type when converting data frames to matrices.

Task 7:

1. Consider the following objects in R:

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

cat("X:                                  ", x,"\n")
```

```
## X:                                  1 3 4
```

```
cat("Y:                                  ", y,"\n")
```

```
## Y:                                  28 80 22 9 5 38 16 4 86 90
```

```
union_xy <- union(x, y)
cat("union of c and y:                   ",union_xy ,"\n")
```

```
## union of c and y:                   1 3 4 28 80 22 9 5 38 16 86 90
```

```
intersection_xy <- intersect(x, y)
cat("intersection of x and y:            ",intersection_xy ,"\n")
```

```
## intersection of x and y:            4
```

```
y_not_in_x <- setdiff(y, x)
cat("Find elements in y but not in x:    ",y_not_in_x ,"\n")
```

```
## Find elements in y but not in x:    28 80 22 9 5 38 16 86 90
```

```
elements_in_y <- x %in% y
cat("Check if each element of x is in y:",elements_in_y ,"\n")
```

```
## Check if each element of x is in y: FALSE FALSE TRUE
```

2. Subvector:

```
s <- 1:200
subvector <- s[s %% 7 == 0 & s %% 2 != 0]
cat("Subvector:",subvector,"\n")
```

```
## Subvector: 7 21 35 49 63 77 91 105 119 133 147 161 175 189
```

Explanation: - s %% 7 == 0 checks if each element of s is divisible by 7 - s %% 2 != 0 checks if each element of s is not divisible by 2 - Combining these two conditions with & (logical AND) ensures that you select elements that meet both criteria. - s[s %% 7 == 0 & s %% 2 != 0] filters the elements of s that satisfy both conditions and creates a subvector of s that is divisible by 7 but not divisible by 2.

3. Another Vector

```
s <- 1:200
s7 <- s[s %% 7 == 0]
s2 <- s[s %% 2 == 0]

# Use set operations to find the subvector
```

```r
subvector <- s7[!(s7 %in% s2)]

# Display the subvector
subvector
```

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

Explanation:

- s7 is created by selecting elements from s that are divisible by 7 using the modulo operator (%%).
- s2 is created by selecting elements from s that are divisible by 2.
- The set operation s7 %in% s2 checks which elements of s7 are also in s2.
- !(s7 %in% s2) negates the result to find elements in s7 that are not in s2, effectively giving a subvector that is divisible by 7 but not divisible by 2.

Feedback

(86/100 Points)

Table of content: Good that you made a table of content, but please don't put too much text in there.

Task 1: If you did not know the function which does conditional logistic regression, how did you find survival::clogit? You were supposed to use help.search("Logistic") or ??Logistic first.
-2

Task 3: x1 <- rep(cumsum(1:5), 1:5) is easier, x7 <- cumsum(15:1) is easier, When creating x13, you should specify the argument include.lowest = TRUE. This way you won't have the NA in the output.
-1

Task 4a: we compare here as.character(2) with "one". It holds "one" > "2" in lexicographic comparison.
-3

Task 6b:
The things you should look at here are:
Be aware of the difference between variance and the standard deviation.
By setting sd = diag(c(1,2,3)), which is a 3x3 matrix, in the rnorm command which creates one dimensional random variables you set two thirds of all entries to have variance 0. This is why two thirds of the entries are either 1, 2 or 3, hence normally distributed with mean 1,2 or 3 respectively and variance 0.
Always check if the random matrix you created actually has the values you want it to have. You can check this by running for example apply(X, 2, mean) and
cov(X) in order to make sure the means of the columns and the covariance matrix is correct.
The solution therefore would be
X <- c(rnorm(100, 1, sqrt(1)), rnorm(100, 2, sqrt(2)), rnorm(100, 3, sqrt(3)))
dim(X) <- c(100, 3)

Task 6b: Explanation why X3 and X4 differ: With X3 we do not center but we set scale = TRUE, which means each column is divided by
the root-mean-square, which is defined by \sqrt{\frac{\sum x_i^2}{n-1}}
Since this is at least as large as the standard deviation
when the column is not centered, this explains why the variances are all smaller than 1
For X4 we also did not center, but we divided "manually" by the standard deviation. This is
the reason why the variances are all 1.

-8