Problem Set 4

Andreas Bender, Philipp Kopper, Sven Lorenz

03 Mai 2021

Transfer

1. Write a function that tests if an integer value is a prime number. Your function should return a logical. Document your function and give it a suitable name. *Hint:* Use the modulo operator `%%`.

```
### Test for prime
###
### The function tests whether a number is a prime number.
### Arguments:
### x:          An interger. Integer, the number for which the test should be performed.
### Returns:    A logical of length one that indicates whether the number is prime or not.
is_prime <- function(number) {

  sum((number %% 1:number) == 0) == 2

}
```

2. Let's make our efforts from last week more convenient.

In last week's transfer task, you found a way to calculate a (in some sense) optimal line that describes the relationship of a flowers petal length and petal width.

In today's task, you should put this functionality into reusable functions that could be applied to any two (continuous) variables from any data set. Make sure that all of your functions are properly documented. You must have worked through last week's problem set to understand this problem set's tasks.

a) Write a function that takes two arguments, `x` (x-coordinate of the points) and `y` (y-coordinate of the points) and returns a vector of length two with the optimal intercept (`[1]`) and slope (`[2]`). Name this function `get_solution()`.

```
### Calculate intercept and slope

### This function finds the intercept (a) and slope (b) of the line through
### a scatter plot that minimizes the quadratic difference between the line
### and the observed values of y.
### Arguments:
### x: a vector. The x-coordinate of the points.
### y: a vector: The y-coordinate of the points.
### Returns: A vector of length two. The first entry is the intercept
### and the second one the slope of the "optimal" line.
get_solution <- function(x, y) {

  X      <- cbind(1, x)
  Xt     <- t(X)
```

```
  XtX    <- Xt %*% X
  invXtX <- solve(XtX)
  Xty    <- Xt %*% y

  unname(invXtX %*% Xty)

}
```

b) Write a function that, given intercept a and slope b, calculates predictions for y based on vector `x`. As before, `x` indicates the x-coordinate of the points. Next to `x` your function will need `a` and `b` as inputs. Name this function `get_predictions()`.

```
### Calculate prediction
###
### This function returns the value of the line on the y-axis, given  x and
### the intercept and slope of the line.
### Arguments:
### x: a vector. The x-coordinate of the points.
### a: A numeric of length 1. The intercept.
### b: A numeric of length 1. The slope.
### Returns: A vector of same length as x. This are the values of the line at
###           x-coordinates provided by vector x.
get_prediction <- function(x, a, b) {

  a + b * x

}
```

c) Write a function that computes the prediction error (sum of errors for each observation). Your function should have arguments `prediction` and `y` as inputs. Name this function `get_error()`

```
### Calculate the sum of squared differences between line and points.
###
### Given a line in the x/y coordinate system, this functions calculates the
### squared difference between the line and the y values (at corresponding
### x-values used to calculate the prediction).
###
### Arguments:
### y: A numeric vector: The y-coordinate of the points.
### x: A numeric vector. Same length as y.
### Returns: A numeric vector of length 1. The calculated error sum of squared differences
get_error <- function(y, prediction) {

  sum((prediction - y)^2)

}
```

d) Discuss whether the names chosen for these functions follow good practice. Rename your functions if necessary.

According to R4DS, 19.3, using "get" in the function name is a good sign that a different name (e.g. a noun might be better). A noun works in the case of `mean`, but `error` for exapmle would be to general. In this case, we can use another suggestion from the book: "If you have a family of functions that do similar things, make sure they have consistent names and arguments. Use a common prefix to indicate that they are connected." In our case this would imply

- `line_solution`,

- `line_prediction`,
- `line_error`

A good alternative to `line_error` would be `sum_of_squared_errors` or `sum_of_squared_differences`, but the relationship to the other functions gets lost.

e) Now, we want to put everything together and visualize the results. Write a function that takes three arguments

- `data`: A data frame.
- `x_name`: The name of the column in `data` that contains the x variable.
- `y_name`: The name of the column in `data` that contains the y variable.

The function should return a named list with elements

- `solution`,
- `prediction` and
- `error`

```
### Line estimator
###
### Given x and y, this function calculates the optimal line (intercept and
### slope), the resulting prediction for y and the error (sum of square
### differences) between line and y.
###
### Arguments:
### data: A data frame.
### x_name: A character of length 1. The name of the variable in data that
###         stores information about the x-coordinates of the points.
### y_name: A character of length 1. The name of the variable in data that
###         stores information about the y-coordinates of the points.
### Returns:
###         - A named list of length 3.
###           - The first entry (solution) contains the intercept and slope
###             (numeric vector of length 2).
###           - The second entry (prediction) contains the prediction
###             (numeric vector of same length as rows in the data set)
###           - The third entry (error) contains the sum of squared differences
###             (numeric vector of length 1)
line_estimator <- function(data, x_name, y_name) {

  x <- data[[x_name]]
  y <- data[[y_name]]

  solution   <- get_solution(x, y)
  prediction <- get_prediction(x, a = solution[1], b = solution[2])
  error      <- get_error(y, prediction)

  list(solution = solution, prediction = prediction, error = error)

}
```

f) Test whether, using your functions, you can reproduce the results you obtained in last weeks exercise.

```
#line_estimator calls all other functions. So that suffices.
line_estimator(iris, "Petal.Width", "Petal.Length")
```

```
## $solution
```

```
##           [,1]
## [1,] 1.083558
## [2,] 2.229940
##
## $prediction
##   [1] 1.529546 1.529546 1.529546 1.529546 1.529546 1.975534 1.752540 1.529546
##   [9] 1.529546 1.306552 1.529546 1.529546 1.306552 1.306552 1.529546 1.975534
##  [17] 1.975534 1.752540 1.752540 1.752540 1.529546 1.975534 1.529546 2.198528
##  [25] 1.529546 1.529546 1.975534 1.529546 1.529546 1.529546 1.529546 1.975534
##  [33] 1.306552 1.529546 1.529546 1.529546 1.529546 1.306552 1.529546 1.529546
##  [41] 1.752540 1.752540 1.529546 2.421522 1.975534 1.752540 1.529546 1.529546
##  [49] 1.529546 1.529546 4.205475 4.428469 4.428469 3.982481 4.428469 3.982481
##  [57] 4.651463 3.313499 3.982481 4.205475 3.313499 4.428469 3.313499 4.205475
##  [65] 3.982481 4.205475 4.428469 3.313499 4.428469 3.536493 5.097451 3.982481
##  [73] 4.428469 3.759487 3.982481 4.205475 4.205475 4.874457 4.428469 3.313499
##  [81] 3.536493 3.313499 3.759487 4.651463 4.428469 4.651463 4.428469 3.982481
##  [89] 3.982481 3.982481 3.759487 4.205475 3.759487 3.313499 3.982481 3.759487
##  [97] 3.982481 3.982481 3.536493 3.982481 6.658409 5.320445 5.766433 5.097451
## [105] 5.989427 5.766433 4.874457 5.097451 5.097451 6.658409 5.543439 5.320445
## [113] 5.766433 5.543439 6.435415 6.212421 5.097451 5.989427 6.212421 4.428469
## [121] 6.212421 5.543439 5.543439 5.097451 5.766433 5.097451 5.097451 5.097451
## [129] 5.766433 4.651463 5.320445 5.543439 5.989427 4.428469 4.205475 6.212421
## [137] 6.435415 5.097451 5.097451 5.766433 6.435415 6.212421 5.320445 6.212421
## [145] 6.658409 6.212421 5.320445 5.543439 6.212421 5.097451
##
## $error
## [1] 33.84475
```

g) Now that you verified the functionality of your wrapper function, its time to recreate the visualizations from last week. To do so, create a new function `plot_line_estimate()`. This function should take the same arguments as the function from subtask e), calculate the optimal line and create

- a scatter plot of x and y and
- add a the optimal line to the plot.

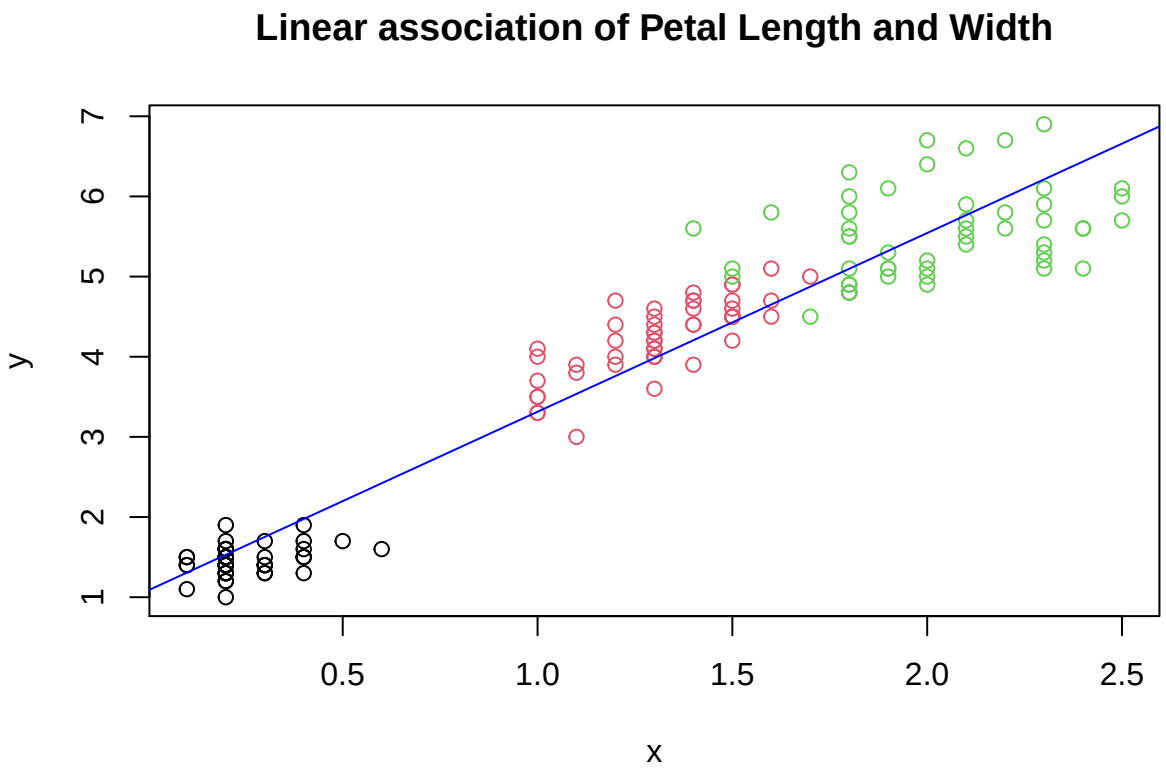
This function should have the same flexibility as the generic `plot` function. That means, you should be able to control the color of the points or the names of the axis labels, etc when calling the function. How can you achieve this without writing a lot of code? Also make sure, that when calling the function, you can change the color of the line.

```
### Line visualization

### Given x and y, this function calculates the optimal line (intercept and
### slope), the resulting prediction for y and the error (sum of squared
### differences) between line and y. Then it creates a scatter plot of x and y
### and adds the estimated line.
### Arguments:
### data: A data frame.
### x_name: A character of length 1. The name of the variable in data that
### stores information about the x-coordinates of the points.
### y_name: A character of length 1. The name of the variable in data that
### stores information about the y-coordinates of the points.
### abline_col: The color of the optimal line.
### ...: Further arguments passed to the generic 'plot' function.
### Returns: A plot.
plot_line_estimate <- function(data, x_name, y_name, abline_col = 1, ...) {
```

```
  # calculate optimal line
  x  <- data[[x_name]]
  y  <- data[[y_name]]
  ab <- get_solution(x, y)

  # create scatter plot, pass ... to the generic plot function
  plot(x = x, y = y, ...) # this is not ideal, because the names of the variables
  # get lost on the x and y axis, but its OK for now.

  # draw line
  abline(ab, col = abline_col)

}
```

j) Test your plot function for the `iris` data set and the columns `Petal.Width` (`x`) and `Petal.Length` (`y`). The optimal line should be blue, there should be a suitable title and the points in the scatter plot should have different color for each species.

```
plot_line_estimate(
  data       = iris,
  x_name     = "Petal.Width",
  y_name     = "Petal.Length",
  abline_col = "blue",
  col        = iris$Species,
  main       = "Linear association of Petal Length and Width")
```



h) In last week's exercise, we said that no straight line can achieve a lower error (according to the definition). However, we might be able to obtain a lower overall error, when we calculate the optimal line for suitable subsets of the data. In our example, a natural candidate is the grouping variable `Species`. Use your previously defined functions to investigate whether the error obtained by calculating the error for

each species separately is lower than calculating it on all data points.

To do so,

- create a subset of the iris data set for each species.
- calculate the "error" for each species.
- sum the three errors and compare the sum to `error_optimal` from last week.

```
## "optimal" error from previous exercise for later reference
error_optimal  <- line_estimator(iris, "Petal.Width", "Petal.Length")$error

#### Now subset solution
## Solution 1:

# Create subsets
iris_setosa     <- iris[iris$Species == "setosa", ]
iris_versicolor <- iris[iris$Species == "versicolor", ]
iris_virginica  <- iris[iris$Species == "virginica", ]

# Calculate line estimate for each species
line_estimate_setosa <- line_estimator(
  data   = iris_setosa,
  y_name = "Petal.Length",
  x_name = "Petal.Width")
line_estimate_versicolor <- line_estimator(
  iris_versicolor,
  y_name = "Petal.Length",
  x_name = "Petal.Width")
line_estimate_virginica <- line_estimator(
  data   = iris_virginica,
  y_name = "Petal.Length",
  x_name = "Petal.Width")
# extract errors for each species
error_setosa     <- line_estimate_setosa$error
error_versicolor <- line_estimate_versicolor$error
error_virginica  <- line_estimate_virginica$error
# overall error based on subsets
error_subsets <- sum(c(error_setosa, error_versicolor, error_virginica))

# compare
error_subsets < error_optimal
```

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

```
error_optimal - error_subsets
```

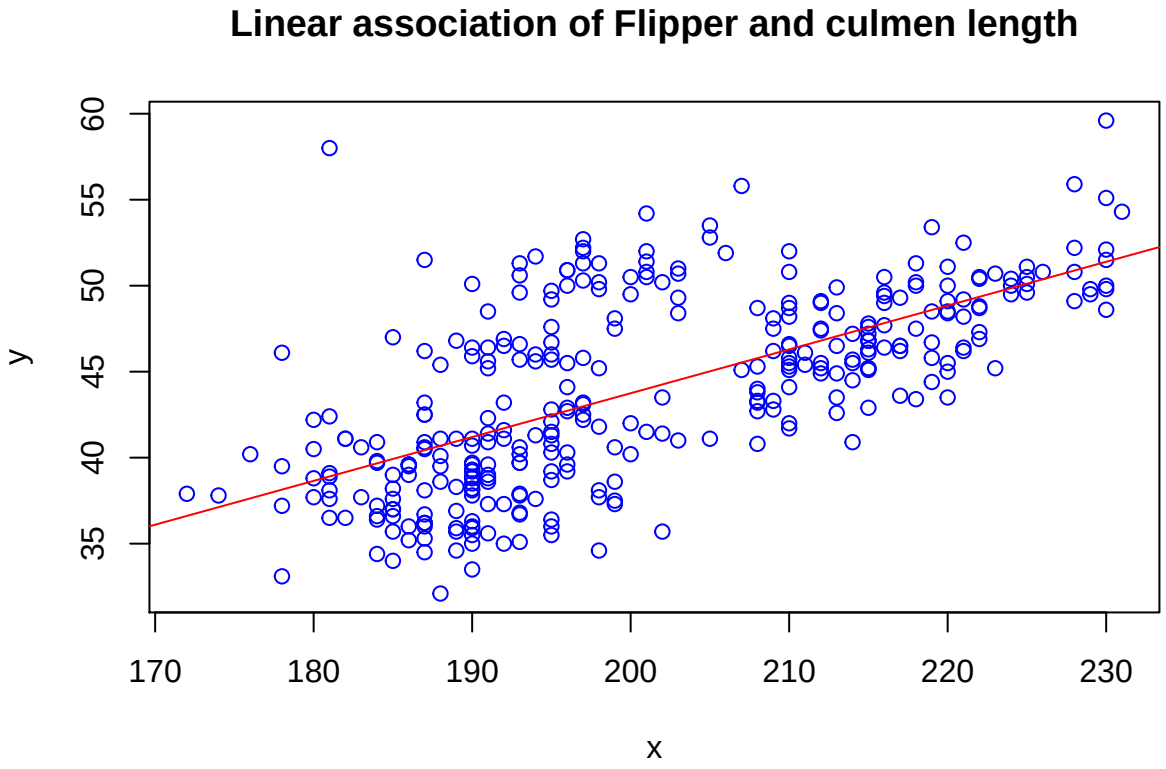
```
## [1] 15.0291
```

```
# Success! We managed to reduce the error!
# However, we now have 3 different lines for each species with different
# intercepts and slopes!

## Solution 2
# put subsets into list
iris_list <- list(iris_setosa, iris_versicolor, iris_virginica)
# alternative: iris_list <- split(iris, iris$Species)
```

```r
# Now use lapply (multiple times)
# First, iterate over all subsets and calculate the error
list_of_estimates <- lapply(iris_list, line_estimator, x_name = "Petal.Width",
  y_name = "Petal.Length")
# the resulting object, estimator list, is once again a list
# the length of the list is 3, since we iterated over 3 subsets
# each entry in the list is itself a list, the output of line_estimator
# to obtain the errors for each species, we need to once again use lapply
# to iterate over the list and extract the error
# To do so, first write a function that extracts the "error" from the output
# of line_estimator
extract_error <- function(estimate) estimate$error
list_of_errors <- lapply(list_of_estimates, extract_error)

error_subsets <- sum(unlist(list_of_errors))
```

j) Test whether your function are robust by applying them to another data set. To do so use the Penguins data set from the first lecture. Omit all missing cases. Use the flipper lengths (`y`) and the culmen lengths (`x`) as columns.

```r
penguins <- readRDS("../penguins.Rds")
penguins <- na.omit(penguins)
plot_line_estimate(
  data       = penguins,
  x_name     = "flipper_length_mm",
  y_name     = "culmen_length_mm",
  abline_col = "red",
  col        = "blue",
  main       = "Linear association of Flipper and culmen length")
```



```r
line_estimator(penguins, "flipper_length_mm", "culmen_length_mm")[c("solution", "error")]
```

```
## $solution
##             [,1]
## [1,] -7.2185580
## [2,]  0.2548247
##
## $error
## [1] 5693.889
```