Homework 5

Stats 20 Lec 1

Winter 2023

General Guidelines

Please use R Markdown for your submission. Include the following files:

- Your .Rmd file.
- The compiled/knitted HTML document.

Name your .Rmd file with the convention `123456789_stats20_hw0.Rmd`, where `123456789` is replaced with your UID and `hw0` is updated to the actual homework number. Include your first and last name and UID in your exam as well. When you knit to HTML, the HTML file will inherit the same naming convention.

The knitted document should be clear, well-formatted, and contain all relevant R code, output, and explanations. R code style should follow the Tidyverse style guide: https://style.tidyverse.org/.

Any and all course material, including these homework questions, may not be posted online or shared with anyone at any time without explicit written permission by the instructor (Michael Tsiang), even after the quarter is over. Failure to comply is a breach of academic integrity.

Note: All questions on this homework should be done using only functions or syntax discussed in Chapters 1–8 of the lecture notes or Homeworks 1–5. No credit will be given for use of outside functions.

Basic Questions

Collaboration on basic questions must adhere to Level 0 collaboration described in the Stats 20 Collaboration Policy.

Question 1

The objective of this question is to help your understanding of the difference between character vectors and factors.

Recall the type casting functions `as.logical()`, `as.numeric()`, and `as.character()`, which allow us to coerce (or cast) a vector into one of a different mode.

Consider the following commands:

```
char <- c("4", "2", "1", "0")
num <- 0:3
charnum <- data.frame(char, num, stringsAsFactors = TRUE)
```

(a)

Apply `as.numeric()` to `char` and to `charnum$char`. Explain why there is a difference in the results.

(b)

Use the type casting functions to coerce `charnum$char` into a numeric vector that is identical to `as.numeric(char)`.

Question 2

The objective of this question is to give practice with various syntax to work with lists and understand their use and limitations.

Consider the following command:

```
simple_list <- list("vector" = 1:10, "matrix" = matrix(6:1, nrow = 3, ncol = 2))
```

(a)

Give two reasons to explain why the command `simple_list$NULL <- NULL` cannot be used to add a named component to `simple_list` that contains the `NULL` object.

(b)

Add a component to `simple_list` that contains the `NULL` object. Verify that `length(simple_list) == 3` returns `TRUE`.

(c)

Using your updated list `simple_list` from (b), update the `vector` component to contain the `NULL` object.

Note: The `vector` component should only contain `NULL`. It should no longer contain a vector.

Question 3

The objective of this question is to give practice with using attributes and connecting their use and behavior with lists.

In statistics and machine learning, it is often useful/necessary to **rescale** (or **normalize**) the range of data values to a standard interval. The simplest method of rescaling is **min-max scaling** (or min-max normalization), which rescales the range of the data values to the interval $[0, 1]$.

Formally, if $x = (x_1, x_2, \ldots, x_n)$ is a sample of data values, then the min-max scaling to $[0, 1]$ is

$$z_i = \frac{x_i - \min(x)}{\max(x) - \min(x)}, \quad \text{for } i = 1, 2, \ldots, n.$$

The scaled values $z = (z_1, z_2, \ldots, z_n)$ would have a range of $[0, 1]$.

More generally, the min-max scaling to an interval $[a, b]$, with $a < b$, is given by

$$z_i = a + \frac{[x_i - \min(x)](b - a)}{\max(x) - \min(x)}, \quad \text{for } i = 1, 2, \ldots, n.$$

Write a function called `my_scale()` that inputs a numeric vector `x` and outputs a vector such that:

- The output vector contains the min-max scaled values of the input vector.
- The output object contains additional attributes called `a` and `b` that contain, respectively, the original minimum and maximum of the input vector.
- If the input vector does not contain both attributes `a` and `b`, the output vector should be scaled to the range from 0 to 1. If the input vector contains both attributes `a` and `b`, the output vector should be scaled to the range from `a` to `b`.

For any numeric vector `x`, the command `my_scale(my_scale(x))` should represent the same vector of values as `x` (because of the attributes, the objects might not be identical). In other words, the `my_scale()` function is its own inverse function.

Note 1: Attributes of an object that are not inherent to its class (e.g., the `dim` and `dimnames` attributes for `matrix` objects) will automatically be shown when printing the object.

Note 2: This is *not* the same type of scaling as the built-in `scale()` function, which uses z-score normalization (also called standardization). You should not use the `scale()` function for comparison.

Intermediate Questions

Collaboration on intermediate questions must adhere to Level 1 collaboration described in the Stats 20 Collaboration Policy.

Question 4: Writing Exam Questions

The objective of this question is to deepen your understanding of the course material by considering it from a different perspective.

Please write two questions which would be appropriate for the next midterm or the final exam. Assume that the exam will be closed R but with a standard function allow list and a few sheets of notes (like the first midterm).

We will choose at least 1 question to use on a future exam. If your question is chosen, not only will you have an advantage on the exam (you should be capable of correctly answering your own question), you will receive a bonus 3% to your exam score.

(a) Writing a Function

Write a free response question whose solution requires writing a function.

You must explain fully what the expected inputs and outputs are, either giving concrete, specific examples of both including any edge cases of interest, or you must give a clear and concise description of the function's purpose.

The function must be possible to write using only the functions and ideas which are permitted to you from Chapters 1–8 of the lecture notes or Homeworks 1–5.

An ideal, efficient, solution should require no more than 10–15 lines of code, but no fewer than 3.

You must also provide what *you* consider to be an **ideal solution**.

(b) Debugging a Function

Write a free response question whose solution requires debugging a function.

You must write a function to accomplish a task you define.

You must introduce 2–3 errors to the code. The errors should not be spelling or style errors. The best errors should not occur on every input, but rather on certain edge cases.

Each error should be of a different type, requiring a different piece of knowledge to solve.

You must provide the erroneous code, the fixed code, a complete list of the errors, an explanation of why the code produces an error, a brief description of what change needs to be implemented to fix the error, and why that change is a fix.

Question 5

The objective of this question is to give further practice with lists and writing functions with different types of output.

Some additional functions which may be useful:

- `unique()` - returns the unique values of a vector, preserving the order in which they occur.
- `%in%` - a vectorized predicate function which returns, for each element on the left-hand-side, if it is present in the right-hand-side.

(a)

Write a function called `my_unlist()` that inputs a list of vectors `x`, and combines all vector components together into a single vector without the `unlist()` function. If `x` is a factor, the output should be a new factor which combines the levels of all of the factors elements in the list, including any levels which aren't actually present in the factor. For mixed-mode lists (lists with more than one vector type) the output should be of the highest mode hierarchy.

```
my_unlist(list(c(2, 1, 1), c(3, 2, 1), 2))
```

```
[1] 2 1 1 3 2 1 2
```

```
my_unlist(list(factor(c("a", "a", "b", "c")),
               factor(c("b", "c", "e")),
               factor(c("a", "d", "b"))))
```

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

Note: If `x` is a mixed-mode list which includes any factors, the output will still be of the highest mode hierarchy but you must treat all factors as their integer equivalents. See below:

```
my_unlist(list(factor(c("a", "b")), c(1, 2)))
```

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

```
my_unlist(list(factor(c("a", "b")), factor(c("b", "c")), c("a", "b")))
```

```
[1] "1" "2" "1" "2" "a" "b"
```

Optional: Expand upon this problem to handle lists of lists (i.e., recursive lists). It may be helpful to try a *recursive* approach. For more guidance on how recursion works, read the "Notes on Recursion" document in the Required Reading on Bruin Learn.

```
my_unlist(list(list(1:3, 1:4), list(list(1:3, 1:3, list(1:4, 1:5)))))
```

```
 [1] 1 2 3 1 2 3 4 1 2 3 1 2 3 1 2 3 4 1 2 3 4 5
```

(b)

The **statistical mode** of a set of data values is the value or values that appear most often. Using your `my_unlist()` function from (a), write a function called `stat_mode()` that returns all of the statistical modes of an input vector or list of vectors `x`. Include an optional argument `first` with a default value of `FALSE` which indicates if only one mode value (the first encountered) should be returned. The returned mode **must** be of the same type/class as `x`.

For example:

```
stat_mode(c(FALSE, FALSE, TRUE, TRUE, FALSE))
```

```
[1] FALSE
```

```
stat_mode(list(c(2, 1, 1), c(3, 2, 1), 2))
```

```
[1] 2 1
```

```
stat_mode(list(c(2, 1, 1), c(3, 2, 1), 2), first = TRUE)
```

```
[1] 2
```

Note: The first mode is `2` *not* `1`, since the `2` is encountered *first*.

```
stat_mode(list(factor(c("control", "treatment1", "control")),
               factor(c("control", "treatment2"))
               )
          )
```

```
[1] control
Levels: control treatment1 treatment2
```

Hint: If you are unable to get a working `my_unlist()` function from (a), you may use the built-in `unlist()` function.

(c)

Using the `stat_mode()` function from (a), write a function called `df_summary()` that inputs a data frame and outputs a list with the following named components:

- `n_obs`: The number of observations in the data frame.
- `n_var`: The number of variables in the data frame.
- `var_names`: A vector of the variable names in the data frame.
- `column_data`: A list object, where each list item appears in alphabetical order and is a list object with the name of a column which:
  1. contains either the:
  - `class`: class of the column,
  - `min`: minimum,
  - `mean`: mean, and
  - `max`: maximum, of that variable as well as,
  - `na_count`: the number of NA values present in the data for that column
  2. OR which contains the:
  - `class`: class of the column,
  - `modes`: vector of the statistical modes of that variable, as well as
  - `mode_count`: the number of times the modal values are each represented in the data frame.

For instance, if you had the following data frame:

Homework_One	Homework_Two	Homework_Three	Lecture
88	95	NA	Lecture 1
84	90	NA	Lecture 1
93	99	88	Lecture 1
NA	60	23	Lecture 2

The structure of your output list (e.g., if you use the `str()` function on your output list) would look like this:

```
List of 4
 $ n_obs      : int 4
 $ n_var      : int 4
 $ var_names  : chr [1:4] "Homework_One" "Homework_Two" "Homework_Three" "Lecture"
```

```
 $ column_data:List of 4
  ..$ Homework_One  :List of 5
  .. ..$ class   : chr "numeric"
  .. ..$ min     : num 84
  .. ..$ mean    : num 88.3
  .. ..$ max     : num 93
  .. ..$ na_count: int 1
  ..$ Homework_Three:List of 5
  .. ..$ class   : chr "numeric"
  .. ..$ min     : num 23
  .. ..$ mean    : num 55.5
  .. ..$ max     : num 88
  .. ..$ na_count: int 2
  ..$ Homework_Two  :List of 5
  .. ..$ class   : chr "numeric"
  .. ..$ min     : num 60
  .. ..$ mean    : num 86
  .. ..$ max     : num 99
  .. ..$ na_count: int 0
  ..$ Lecture       :List of 3
  .. ..$ class     : chr "character"
  .. ..$ modes     : chr "Lecture 1"
  .. ..$ mode_count: int 3
```

Note: This is the *structure* of the output, not the output list itself.

(d)

Download the `starwars.RData` file from BruinLearn and load it into your workspace.

Side Note: The `starwars.RData` is a modified version of the `starwars` data found in the `dplyr` package. *Do not use the version in `dplyr`.*

Use your `df_summary()` function from part (c) on the `starwars` data, and store the result in your workspace.

(e)

Using *only* the output object from (d), find the most common starships that the characters in the `starwars` data have piloted. Do not refer to the original `starwars` data.

Advanced Questions

Collaboration on advanced questions must adhere to Level 1 collaboration described in the Stats 20 Collaboration Policy.

Note: Advanced Questions are intended for further enrichment and a deeper challenge, so they will not count against your grade if they are not completed or attempted.

Question 6

Imputation is the process of replacing missing values by estimated values. The simplest (far from preferred) method to impute values is to replace missing values by the most typical (or "average") value.

Write a function called `impute()` that will impute missing values from a specified column in a matrix or data frame.

The `impute()` function should have one required argument `x` that specifies the input matrix or data frame and three optional arguments:

- The `col` argument specifies the column or columns in which to impute values. `col` is optional with no default. If `col` is not specified, the function should impute values for all numeric columns.
- The `center` argument specifies what function to use for imputation. Your function should work for any function which returns a single summary statistic for a vector, such as `mean`, `median`, or `max`. This argument should be optional with a default of `mean`.
- The `margin` argument specifies one of two ways to input values:
  - Impute the missing values using the `center` of the observed (non-missing) values in the column.
  - Impute the missing values using the `center` of the observed values in the row.
  - `margin` should be optional with a default of `2`.

Note: `center` should only be computed on the *numeric* values in the column or row.

Optional:

1. You may consider extending `impute()` to compute `center` on the *numeric* variables and use your `stat_mode()` function on the categorical variables.
2. You may consider extending `impute()` to accept a list of vectors for `col`, allowing for multiple, sequential, imputations.

The `impute()` function should return the object `x` with imputed values in the specified column(s).

For example:

```
x_mat
```

```
     [,1] [,2] [,3]
[1,]   NA   10    5
[2,]    7    6    7
[3,]   NA    3    6
[4,]    1    7    3
[5,]   NA   NA    2
```

```
impute(x_mat, col = 1, center = mean, margin = 2)
```

```
     [,1] [,2] [,3]
[1,]    4   10    5
[2,]    7    6    7
[3,]    4    3    6
[4,]    1    7    3
[5,]    4   NA    2
```

```
impute(x_mat, col = 2, center = median, margin = 1)
```

```
     [,1] [,2] [,3]
[1,]   NA   10    5
[2,]    7    6    7
[3,]   NA    3    6
[4,]    1    7    3
[5,]   NA    2    2
```

```
impute(x_mat, col = c(1, 2), center = median, margin = 1)
```

```
     [,1] [,2] [,3]
[1,]  7.5   10    5
[2,]  7.0    6    7
[3,]  4.5    3    6
[4,]  1.0    7    3
[5,]  2.0    2    2
```

The `impute()` function should be able to input an `x` object of any number of rows or columns.