Bioinformatics Analysis and Visualisation of Medical Genomics Data

Assignment 1

Alen Lovric

Dependencies

- Pipeline depends on the following packages: 'tidyverse', 'ggrepel', 'remotes', and 'ggpubr' among the rest.

```
# Load libraries
load_libs <- c('tidyverse', 'ggrepel', 'remotes', 'ggpubr', 'kableExtra', "tinytex")
sapply(load_libs, require, character.only = T)
```

Task 4a - Using R example datasets

- **CO2 dataset**

```
# CO2 data set
# Attach data and take a look
data(CO2)
help(CO2)

# Calculate average and median CO2 uptake
# of the plants from Quebec and Mississippi?
co2_summary <- CO2 %>% group_by(Type) %>% summarise(mean = mean(uptake),
                                    median = median(uptake))

# Print table
kable(co2_summary, caption = 'CO2 data set.', format = 'latex', booktabs = T) %>%
            kable_styling(latex_options = c('striped',"HOLD_position"))
```

Table 1: CO2 data set.

Type	mean	median
Quebec	33.54286	37.15
Mississippi	20.88333	19.30

Description: The CO2 data frame has 84 rows and 5 columns of data from an experiment on the cold tolerance of the grass species Echinochloa crus-galli.

Task 4b - Using R example datasets

- **Airway dataset**

```
# Install airway if not already present
# BiocManager::install("airway")
library(airway)
data(airway)

# Extract counts data from the airway assay
counts <- assay(airway) %>% as.data.frame()

# Use tidyverse to calulate no. of expressed and non expressed genes per sample
counts_summary <- counts %>% summarise_all(.funs = list(expressed = function(x) sum(x > 0),
                                                        notExpressed = function(x) sum(x == 0))) %>%
          pivot_longer(cols = everything()) %>%
          separate(name, c('Sample', 'temp'),sep = '_') %>%
          pivot_wider(names_from = 'temp', values_from = 'value')

# Print table
kable(counts_summary, caption = 'Airway dataset.',
      format = 'latex', booktabs = T) %>%
            kable_styling(latex_options = c('striped',"HOLD_position"))
```

Table 2: Airway dataset.

Sample	expressed	notExpressed
SRR1039508	24633	39469
SRR1039509	24527	39575
SRR1039512	25699	38403
SRR1039513	23124	40978
SRR1039516	25508	38594
SRR1039517	25998	38104
SRR1039520	24662	39440
SRR1039521	23991	40111

Description: Number of expressed and non-expressed genes per sample.

Task 5 - Creating R Functions

1. mean-to-median ratio
2. trimmed mean

```
# Create a random numeric vector with fixed seed for reproducibility
set.seed(1)
numval <- sample(1:100, 50)

# Function that calualtes mean-to-median ratio
vectorInfo <- function(x){
    res <- mean(x)/median(x)
    return(cat(paste('mean-to-median ratio:', round(res, 2))))
}

# Call the function using previously created numeric vector
vectorInfo(numval)
```

```
## mean-to-median ratio: 1.03
```

```
# Create trimmed mean function
# Both min and max values are removed from vector before mean calulation
trimmedMean <- function(x){
      temp_x <- x[-c(which.max(x), which.min(x))]
      res <- sum(temp_x)/length(temp_x)
      return(cat(paste('Trimmed mean:', res)))
}

# Call the function using previously created numeric vector
trimmedMean(numval)
```

```
## Trimmed mean: 54.5
```

Why, how, and when not to use pipes.

The use of pipes (%>%) in R, often provided by packages like **magrittr** and widely adopted by the **dplyr** package, has become a popular tool for enhancing the readability and efficiency of R code. However, there are situations where using pipes may not be the best choice, and understanding when not to use them is crucial for writing clean and maintainable R code.

Pipes are incredibly useful when you need to perform a series of operations on a dataset or object, as they allow you to chain together functions in a left-to-right fashion, enhancing code readability. They shine in data manipulation tasks, where you apply multiple transformations to a data frame, for example.

However, here are scenarios where using pipes may not be ideal:

- **Simple Operations:** For basic operations that don't involve chaining multiple functions together, pipes can add unnecessary complexity to the code. In such cases, using standard R syntax is more straightforward and doesn't introduce unnecessary overhead. **Debugging:** While pipes make code more readable, they can make debugging more challenging. When you encounter an error, it can be trickier to pinpoint which step in the pipeline is causing the issue, especially if the data transformations are complex.
- **Non-linear Data Flow:** When your data processing doesn't follow a linear sequence, pipes may not be the best choice. Pipes work best for operations that naturally flow from one step to the next. If your workflow involves branching or looping, using pipes can lead to convoluted code.
- **Performance:** In some cases, using pipes can introduce a small performance overhead compared to writing the code in a more traditional way. While this overhead is usually negligible, it might matter in computationally intensive tasks.
- **Code Style and Team Consistency:** If you're working on a team project, it's essential to maintain code consistency. Some team members may prefer traditional R syntax over pipes, so it's crucial to follow the project's coding guidelines.

Apply-family of functions

The apply-family of functions, including **apply**, **lapply**, **sapply**, and others, play a pivotal role in streamlining and enhancing the efficiency of data analysis and manipulation tasks within my work. These functions provide a concise and flexible way to apply operations across data structures in R, such as matrices, lists, and data frames.

One of the key advantages of these functions is their ability to abstract away the complexities of looping constructs. Traditional for-loops in R can be cumbersome and error-prone, especially when dealing with multidimensional data. The apply functions, on the other hand, allow me to express my intentions more clearly and succinctly, reducing the chances of coding errors.

apply is particularly useful when working with matrices, as it efficiently applies a function to rows or columns, enabling me to calculate row-wise or column-wise statistics or perform custom operations effortlessly. For lists, **lapply** facilitates the application of a function to each element, returning the results as a list—a valuable approach for iterative tasks involving data structures of varying lengths and types.

The **sapply** function is a versatile tool that simplifies the conversion of the output from lapply into a more manageable format, such as a vector or data frame. This simplification is especially useful when I need to work with results in a unified structure.

Furthermore, the apply-family functions promote code modularity and maintainability. By encapsulating specific operations within functions and then applying them using the apply-family, I can create cleaner, more organized code. This modular approach enhances code reusability and allows for easier debugging and testing.

Task 6 - Basic visualization with R

```
library('remotes')
```

Task 7 - Tidybiology package

- Chromosome data set

```
# Install tidybiology
# devtools::install_github("hirscheylab/tidybiology")
library(tidybiology)

# attach chromosome data
data("chromosome")

# A.
# summarize target variables
chromosome_summary <- chromosome %>% select(variations, protein_codinggenes, mi_rna) %>%
               pivot_longer(cols = everything(), names_to = 'Variable') %>%
               mutate(Variable = case_when(Variable == 'mi_rna' ~ 'miRNA',
                                           Variable == 'protein_codinggenes' ~ 'Protein coding genes',
                                           TRUE ~ 'Variations')) %>%
               group_by(Variable) %>%
               summarise(mean = mean(value), median = median(value), max = max(value)) %>%
               mutate(mean = round(mean)) %>%
               arrange(max)


# Print table
kable(chromosome_summary, caption = 'Chromosome dataset.',
      format = 'latex', booktabs = T) %>%
            kable_styling(latex_options = c('striped',"HOLD_position"))
```

Table 3: Chromosome dataset.

Variable	mean	median	max
miRNA	73	75	134
Protein coding genes	850	836	2058
Variations	6484572	6172346	12945965

```r
# B.
# Plot distribution of the chromosome size
# Change plot theme and axis labels
ggplot(chromosome, aes(x = length_mm)) +
        geom_histogram(binwidth = 10, fill = "lightgray", color = "black") +
        labs(x = "Chromosome Length (mm)",
             y = "Frequency") +
        theme_bw() +
        theme(panel.grid.major = element_blank(),
              panel.grid.minor = element_blank())
```

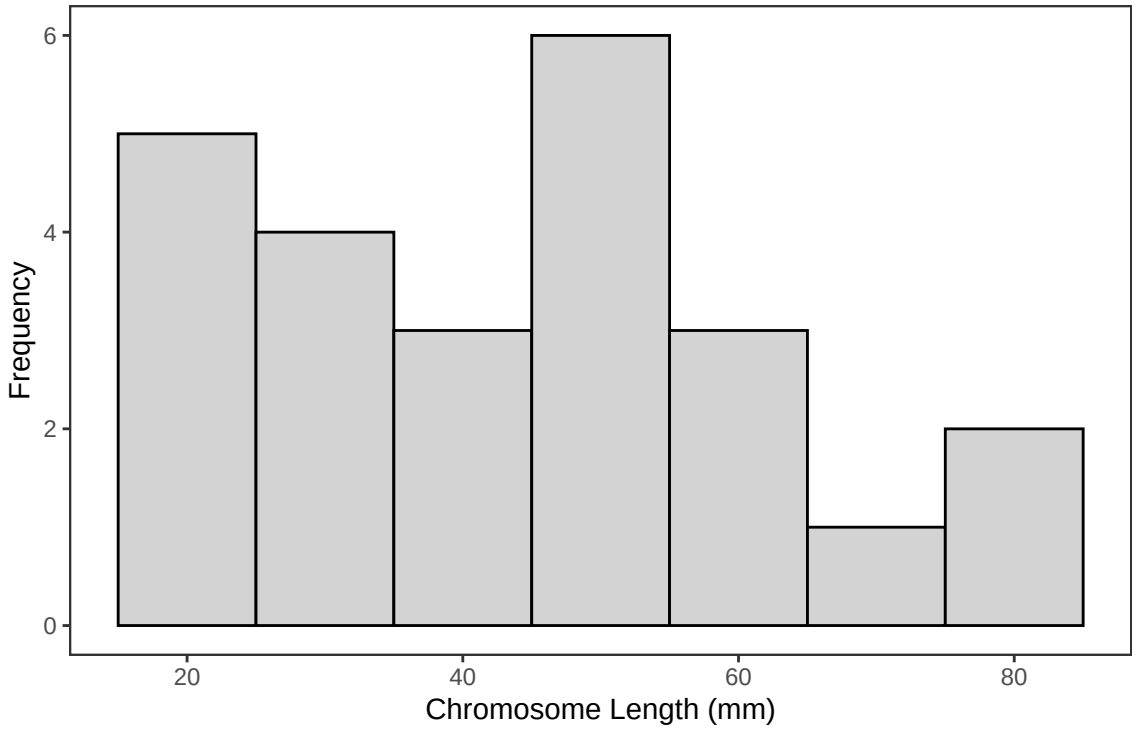

Figure 1: Chromosome data set. Distribution of Chromosome Sizes.

```
# C.
# Plot length and number of protein coding genes
# Add the line and spearman correlation
# Change plot theme and axis labels
ggplot(chromosome, aes(length_mm, protein_codinggenes)) +
        geom_point() +
        geom_smooth(method = 'lm', fill = 'gray', alpha = 0.3) +
        stat_cor(method = 'spearman', cor.coef.name = 'rho') +
        labs(x = 'Chromosome size (mm)', y = 'Protein coding genes') +
        theme_bw() +
        theme(panel.grid.major = element_blank(),
              panel.grid.minor = element_blank())
```

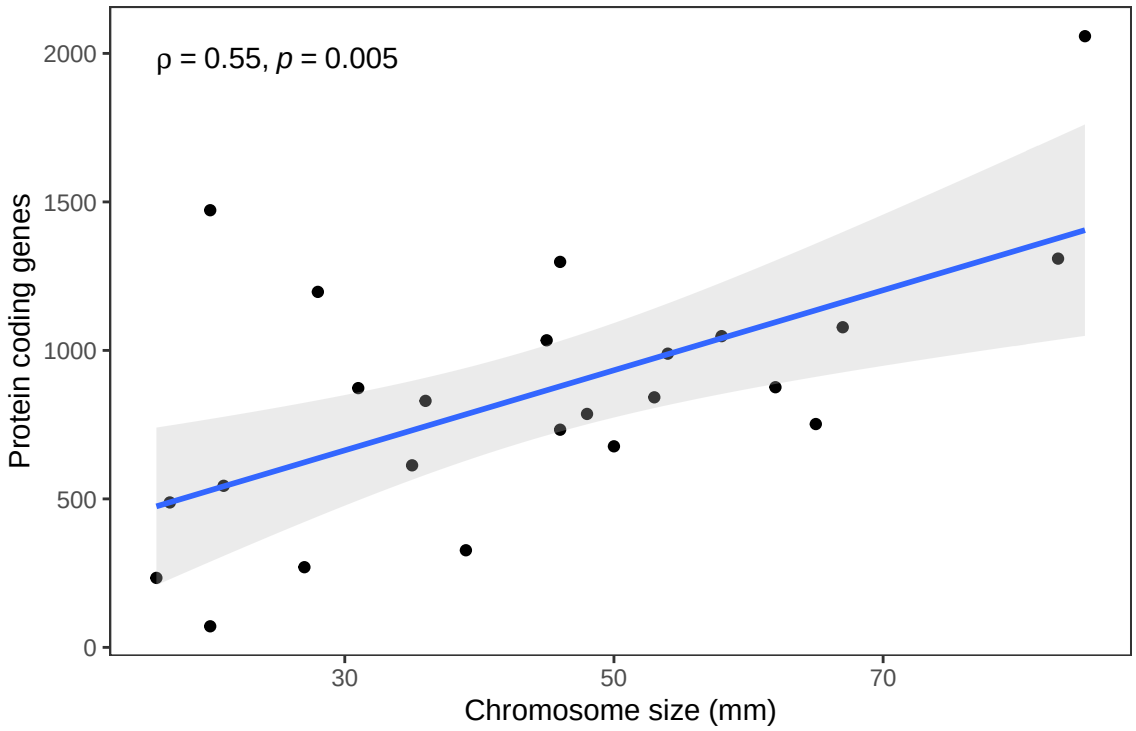

Figure 2: Chromosome data set. Association between chromosome length and number of protein coding genes.

```
# D.
# plot length and number of miRNA
# add the line and spearman correaltion
# change plot theme and axis labels
ggplot(chromosome, aes(mi_rna, protein_codinggenes)) +
        geom_point() +
        stat_cor(lmethod = 'spearman', cor.coef.name = 'rho') +
        geom_smooth(method = 'lm', fill = 'gray', alpha = 0.3) +
        labs(x = 'Protein coding genes', y = 'miRNA') +
        theme_bw() +
        theme(panel.grid.major = element_blank(),
              panel.grid.minor = element_blank())
```

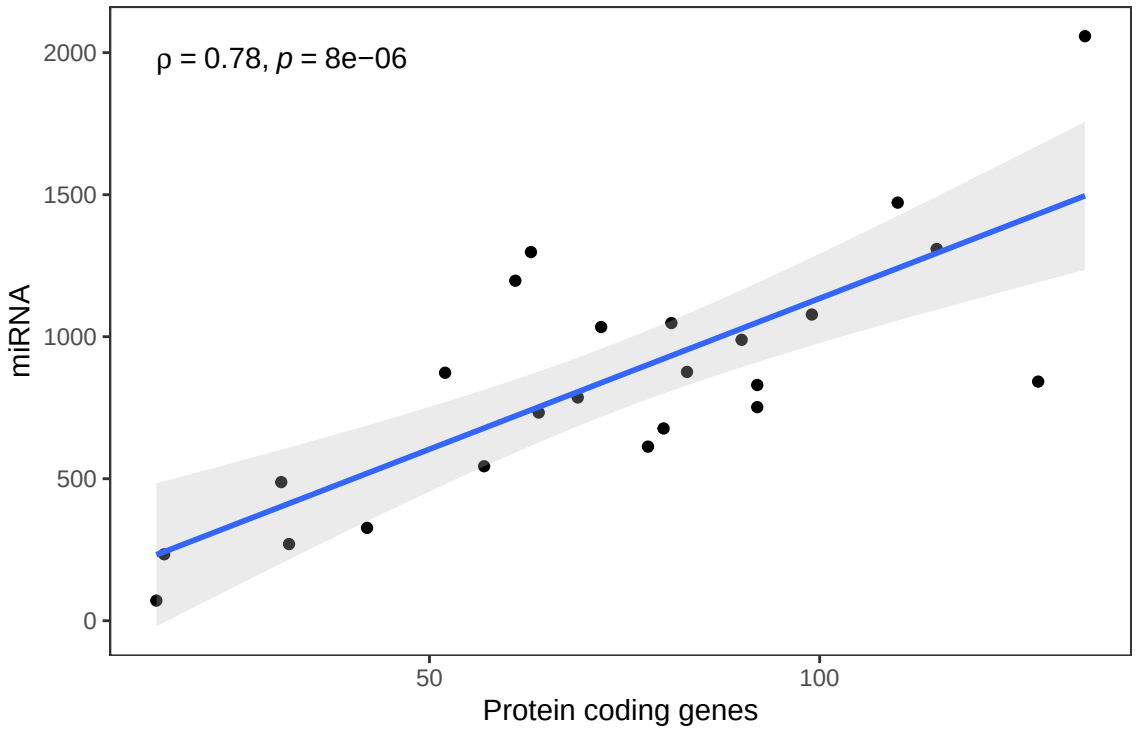

Figure 3: Chromosome data set. Association between number of protein coding genes and miRNA.

Task 7 - Tidybiology package

- Protein data set

```
# Attach the data
data("proteins")

# E.
# Summary stats of protein data
protein_summary <- proteins %>%
          select(length, mass) %>%
          summarise(across(everything(),
                    list(mean = mean, median = median, max = max)))

# Print table
kable(protein_summary %>%
                      pivot_longer(cols = everything()) %>%
                      separate(name, c('Info', 'group')) %>%
                      pivot_wider(names_from = 'group', values_from = 'value'),
              caption = 'Protein data set.',
              format = 'latex', booktabs = T) %>%
      kable_styling(latex_options = c('striped',"HOLD_position"))
```

Table 4: Protein data set.

Info	mean	median	max
length	557.1603	414.0	34350
mass	62061.3791	46140.5	3816030

```
# Plot length vs mass
ggplot(proteins, aes(x = length, y = mass)) +
        geom_point(aes(color = length), size = 1.5) +
        scale_color_gradient(low = "blue", high = "red") +
        labs(title = ,
             x = "Protein length",
             y = "Protein mass") +
        theme_bw() +
        theme(legend.position = "none",
              panel.grid.major = element_blank(),
              panel.grid.minor = element_blank()) +
        geom_text_repel(aes(label = ifelse(mass > max(mass) * 0.20, gene_name, NA)),
                        size = 2.5,
                        max.overlaps = 15) +
        geom_text(data = protein_summary,
                  aes(x = length_mean, y = mass_max, label = "Median protein length"),
                  angle = 90, vjust = -1.5,
                  hjust = 1, size = 2) +
        geom_text(data = protein_summary,
                  aes(x = length_max, y = mass_median, label = "Median protein mass"),
                  vjust = -1, hjust = 1, size = 2) +
        geom_hline(yintercept = protein_summary$mass_median,
```

```
            linetype = "dashed",
            color = "gray") +
  geom_vline(xintercept = protein_summary$length_median,
            linetype = "dashed",
            color = "gray")
```

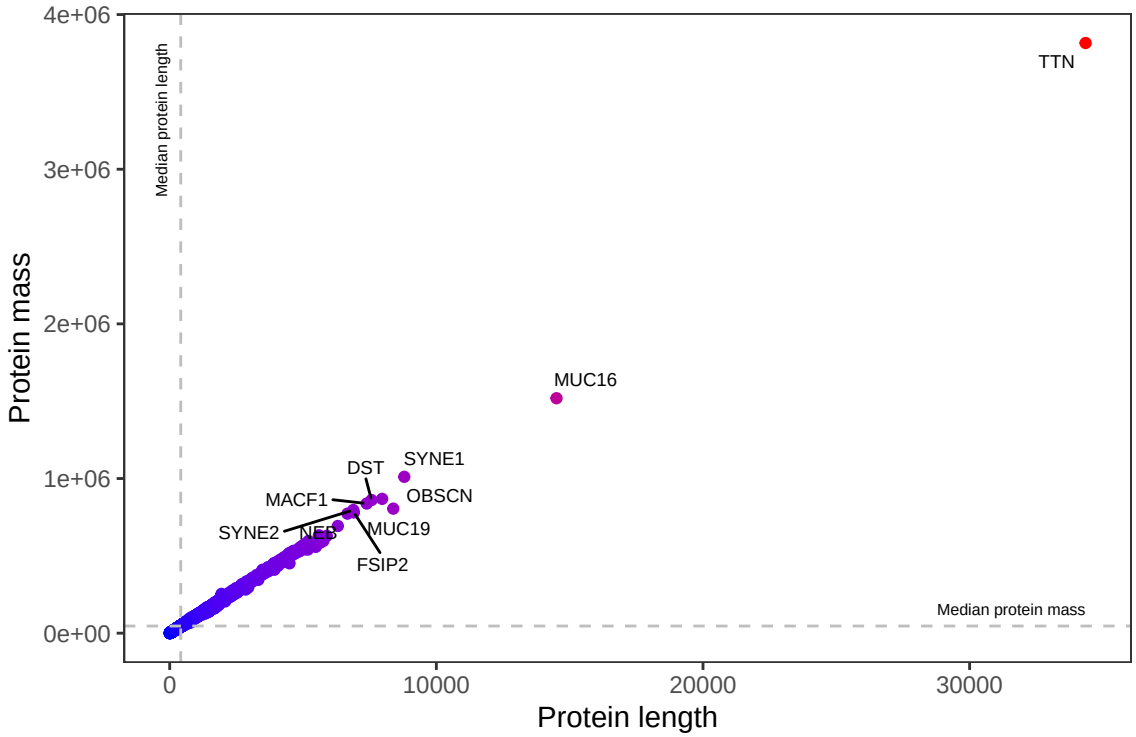

Figure 4: Protein data set. Relationship between Protein Length and Protein Mass.