UBMI-IFC, UNAM
Coyoacan, CDMX

Sequece Characterization Test

Using Genomic-Benchmarks Data

CUSTOM FUNCTIONS ANNEX

Author: Fuentes-Mendez
David Gregorio

Table of contents

Why use Quarto?

Many hurdles came across while trying to present code-cell outputs in *Quarto*, likely stemming from the fact that Quarto is a fairly new notebook format and therefore still has a lot of room for improvement. Despite these limitations, Quarto offers a distinct advantage that made me choose it for the current project: its' easy and intuitive integration of multiple programming languages at once. This capability is particularly valuable for bioinformatics workflows, where transitioning between *Bash*, *R*, and *Python* is often essential for comprehensive analyses.

Note:

Each time the programming language changes, the code-cell will display a header indicating the corresponding language.

Output-Wrapping

Among the initial challenges I noticed with Quarto's output handling was that code-cell text, and sometimes code-cell output, would ignore column width and overflow. First I sought a Quarto-specific solution, such as modifying the YAML or adjusting cell layout options, but none seemed to work and no there was no immediate intention to solve some issues (one example is the case of code-cell text overflowing in the PDF format export: quarto-cli discussion #3693). The best option available I found was to "*chisel*" each code-cells' text myself (so that no line is larger than the column's width) and handle their respective output-wrapping through "*in-cell*" solutions: functions that capture and alter flawed outputs.

An example of such functions are the following 'outputwrap1' and 'outputwrap' functions, which were developed to address the issue of overflowing outputs from 'list_datasets' and 'info_gb'. The former 'outputwrap1', was the initial iteration created to handle this problem. However, even though successful at wrapping 'list_datasets' output, 'info_gb' would print part of its output before 'outputwrap1' could capture it. To solve this, I tried a new approach in 'outputwrap' by calling 'info_gb' directly within the wrapping function, allowing it to capture the entire output. This adjustment turned out to be successful.

Python Code

```
import io, textwrap
from contextlib import redirect_stdout

def outputwrap1(output_func, set_width = 50):
    output = str(output_func)
    wrap_output = textwrap.fill(output,
                                set_width)
    print(wrap_output)

def outputwrap(output_func, width = 50,
               args = (), kwargs = None):
    if kwargs is None:
        kwargs = {}
    output_stream = io.StringIO()
    with redirect_stdout(output_stream):
        func_df = output_func(*args, **kwargs)
    full_output = output_stream.getvalue()
    wrapped_output = textwrap.fill(full_output,
                                   width = width)
    subst_output = wrapped_output.replace(". ", ".\n\n")
    print(subst_output)
    print(func_df)
```

Another instance of the output-wrapping function, but implemented in *R*, is in the following code-cell. Although, unlike its' counterparts in *Python*, this function was specifically designed to manage the overflowing output of some vectors.

R Code

```
#(!) Needs to be updated, has changes in source file
vectwrap <- function(func_out, width = 50) {
  form_out <- format(round(func_out, digits = 2),
                     nsmall = 2)
  wrapped <- strwrap(gsub(',', '', toString(form_out)),
                     width = width)
  wrapped[1] <- paste("\n\n[1]", wrapped[1])
  len_w <- length(wrapped)
  wrapped[2:len_w] <- paste("\n   ", wrapped[2:len_w])
  cat(wrapped)
}
```

Since some outputs are printed next to each other Subsequently, while employing Quarto's figrue layout features, it was noticed how some of them would be printed immediately next to each other (rowwise), so the devised solutions were the following functions, which attach some padding to the input provided and print an horizontal space ('addpadd' and '{hspace}' respectively). within Quarto's cell, the following

```
addpadd <- function(func_out) {
  captout <- capture.output(func_out)
  captout[1] <- paste("\n", captout[1], sep="")
  captout[2:length(captout)] <- paste("\n",
            captout[2:length(captout)], sep="")
  captout <- append(captout, "\n\t")
  cat(captout)
}

hspace <- function()
  knitr::asis_output("\\textcolor{white}
      {\\tiny\\texttt{hi}}\\normalsize")
```

Required Libraries

On the other hand, one inconvenient found in the "*sequence characterization*" step was the necessity to call all the libraries required by 'sequence_characterizer' inside of its' surrounding 'foreach' chunk (even though they are already requested for in the first lines of 'genome-functions.R'). The easiest method I could come up with to solve this is the following 'required_libs' which suppresses the startup messages that, in this context, only clutters the output.

```
required_libs <- function(libs)
  for(lib in libs) suppressPackageStartupMessages(
                     library(lib, character.only = TRUE))
```

Table Customization

Conversely, an inconvenience not related with Quarto was that for some reason the *LaTeX* command '\twocolumn' has incompatibilities with the *longtable* format from 'knitr' (their default type of table). This seemed to be a well documented issue (knitr issues #1348) and the solution also seemed fairly simple (set the longtable argument as FALSE). Thereafter, 'teatable' is a function that does that coupled with a little bit more of table personalization. An unexpected outcome of this workaround is that for some other reason each cell that uses 'teatable' (or 'coffeetable') require to have the Quarto cell feature 'cache' set to 'false', so that it is reloaded each time the notebook is rendered. The first version of 'teatable' was somethin like the code below:

```
teetable <- function(tabl){
  kbl(
    as.data.frame(tabl), booktabs = T, longtable = F) %>%
    kable_styling(position = "center",
                  latex_options = c("striped",
                    "scale_down", "hold_position"))
}
```

The name came from the fact that "kable" might derive from "knitr_table" and I thought that I might as well name my table function(s) after types of furniture. Ultimately it was decided on "teatable" because it was the shortest name I could come up with.

The code below is the final version which displays a colored 'teatable'. The reason behind this was that I liked a lot the result obtained after coloring 'coffeetable'.

```
teatable <- function(tabl, colsize, cat=FALSE,
                     decay=TRUE, pale = TRUE) {
  greens <- c("#a3cfa3", "#b3e6b3", "green!20")
  greens_decay <- c("#8c8c66", "#e6e6cc", "#c2c299")
  decayed_pale <- c("#ccccb3", "#f4f4e2", "#e0e0cc")

  if (!decay) { rowcolors <- greens
  } else { if (!pale) { rowcolors <- greens_decay
    } else { rowcolors <- decayed_pale } }

  ktabl <-
    kbl(tabl, align = 'c',
        booktabs = TRUE, longtable = FALSE) %>%
      kable_styling(position = "center",
                    latex_options=c("striped",
                                    "scale_down",
                                    "hold_position"),
                    stripe_color = rowcolors[3],
                    full_width = F) %>%
      row_spec(seq(2, nrow(tabl), by = 2),
               background = rowcolors[2]) %>%
      row_spec(0, background = rowcolors[1])

  if (!missing(colsize))
    ktabl <- ktabl %>% column_spec(1:ncol(tabl),
                                   width = colsize)
  ktabl <- gsub(x = ktabl, "\\\\midrule",
    "\\\\specialrule\\{0.6pt\\}\\{0.8pt\\}\\{0.6pt\\}")
  ktabl <- gsub(x = ktabl, "\\\\toprule",
    "\\\\specialrule\\{1pt\\}\\{0pt\\}\\{1pt\\}")
  ktabl <- gsub(x = ktabl, "\\\\bottomrule",
    "\\\\specialrule\\{1pt\\}\\{0.6pt\\}\\{0pt\\}")
  ktabl <- gsub(x = ktabl, "\\\\begin\\{table\\}([^\n]*)",
    "\\\\begin\\{table\\}\\1\n\\\\footnotesize")

  if (!cat) {asis_output(ktabl)} else {cat(ktabl)}
}
```

The definition of 'coffeetable' is significantly larger than (and practically does the same as) 'teatable' plus some functions like: row height modification, collapse of columns (i.e. multi-rows) and dynamic coloring of multi-row cells. Therefore, for the sake of readability, it was decided to leave it out of the current Appendix.

One fun-fact about 'coffetable' is that the name actually comes from the amount of coffee I needed to get to this workaround solution, seconds later I noticed the ironic coincidence between tea and coffee, and after that, I changed its' color theme to a brown-scale.

Legend Issues

While plotting with 'cowplot::plot_grid', I noticed it wouldn't display any bottom legend coming from the 'cowplot::get_legend' function. After googling it for a bit, it turned out to be apparently caused by a conflict between 'cowplot' and 'ggplot2=3.5'. Thankfully someone in the following GitHub (cowplot issue #202) issue came up with a solution, which I copyied as 'get_legend_bypass'.

```
get_legend_bypass <- function(plot) {
  legends <- get_plot_component(plot, "guide-box",
                                return_all = TRUE)
  nonzero <- vapply(legends, function(x)
                    !inherits(x,"zeroGrob"), TRUE)
  idx <- which(nonzero)
  if (length(idx) > 0) { return(legends[[idx[1]]])
  } else { return(legends[[1]]) }
}
```

Plotting: Pyramid, Bar, Half-Violin

Now, for the sake of text-space optimization, we'll make some plotting functions (principally pyramid, bar and violin plots), in order to visualize our data.

```
# To plot pyramid-plots
pyrplot_ <- function(cre_data, kmer_labels, title,
                     x_label, y_label, y_breaks) {
  CREs <- c("Enhancer", "Promoter")
  field_order <- filter(cre_data, Type==CREs[1])$Field
  fact_field_order <- factor(cre_data$Field, field_order)

  if (missing(kmer_labels)) kmer_labels <- field_order
```

```r
  ggplot(cre_data) +
    geom_bar(aes(x = fact_field_order,
                 y = ifelse(Type == CREs[1],
                            -Means, Means),
                 fill = paste(Type, "Means")),
             stat = "identity", position = "identity",
             alpha = 0.6, width = 0.7) +
    geom_errorbar(aes(x = fact_field_order,
                      ymin = ifelse(Type == CREs[1],
                                   -Means + StDevs,
                                    Means - StDevs),
                      ymax = ifelse(Type == CREs[1],
                                   -Means - StDevs,
                                    Means + StDevs)),
                  width = 0.5, alpha = 0.6,
                  colour = "black") +
    coord_flip() +
    scale_x_discrete(labels = kmer_labels) +
    scale_y_continuous(breaks = y_breaks,
                       labels = abs(y_breaks)) +
    scale_fill_manual(values = c("turquoise", "coral"),
                      labels = CREs) +
    labs(y = "Means", x = x_label,
         title = title, fill = "CRE Type") +
    theme_minimal() +
    theme(legend.position = "bottom",
          axis.title = element_text(size = 15),
          text = element_text(size = rel(4.25)),
          legend.text = element_text(size = 13),
          legend.title = element_text(size = 13.5),
          axis.text.y = element_text(family = "mono"),
          plot.title = element_text(size = 16, hjust = 0.5)
}

barplot_ <- function(cre_data, y_breaks, y_axis_title="",
                     fill_legend_title="") {
  ggplot(cre_data) +
    geom_bar(aes(x = factor(Type),
                 y = Means, fill = Type),
             stat = "identity", position = "identity",
             alpha = 0.6) +
    geom_errorbar(aes(x = factor(Type),
                      ymin = Means - StDevs,
                      ymax = Means + StDevs),
                  alpha = 0.6, width = 0.5,
                  colour = "black") +
    labs(y = y_axis_title, x = "",
         fill = fill_legend_title) +
    scale_y_continuous(breaks = y_breaks,
                       labels = y_breaks) +
    scale_fill_manual(values = c("turquoise", "coral")) +
    theme_minimal() +
    theme(legend.position = "none",
          text = element_text(size = rel(4.5)),
          axis.title = element_text(size = 16))
}

hvioplot_ <- function(data, y_var, y_label = "",
                     fill_legend_title = "") {
```

```r
  # x_breaks <- seq(-0.12,0.12,0.06)
  ggplot(data, aes(x = 0, y = !!sym(y_var), fill = type)) +
    geom_violinhalf(flip = 1, adjust = 0.25,
                    trim = FALSE, scale = "count",
                    position = position_dodge(width = 0),
                    linewidth = 0.1, alpha = 0.6) +
    theme_minimal() +
    # scale_x_continuous(breaks = x_breaks ,
    #                    labels = abs(x_breaks )) +
    scale_fill_manual(values = c("turquoise", "coral")) +
    labs(x = "", y = y_label, fill = fill_legend_title) +
    theme(legend.position = "none",
          text = element_text(size = rel(4.5)),
          axis.title = element_text(size = 13))
}
```

Principal Components Analysis

```r
simple_pca <- function(data, type, n_comp = 10) {
  # Validate inputs
  if (!is.matrix(data) && !is.data.frame(data)) {
    stop("'data' must be a matrix or data frame.")
  }
  if (length(type) != nrow(data)) {
    stop("'type' must have the same length
         as the number of rows in 'data'.")
  }

  # Scale the data
  scaled_data <- scale(data)

  # Perform PCA
  pca_result <- irlba(scaled_data, nv = n_comp)

  # Scree Plot
  sing_vals <- pca_result$d
  var_expl<- (sing_vals^2) / sum(sing_vals^2) * 100

  scree_df <- data.frame(Component = seq_along(var_expl),
                         Variance = var_expl)

  scree_plot <- ggplot(scree_df, aes(x = Component,
                                     y = Variance)) +
    geom_line(color = "cadetblue", linewidth = 0.5) +
    geom_point(color = "dodgerblue4", size = 2) +
    labs(title = "Scree Plot",
         x = "Principal Component",
         y = "% Explained Variance") +
    theme_minimal() +
    theme(
      plot.title = element_text(size = 16, hjust = 0.5),
      axis.title = element_text(size = 13),
      axis.text = element_text(size = 11))

  # Extract scores
  scores <- pca_result$u %*% diag(pca_result$d)
  # Prepare data for ggplot
  pca_df <- data.frame(PC1 = scores[, 1],
                       PC2 = scores[, 2],
                       Type = type)

  # PCA Plot
  pca_plot <- ggplot(pca_df, aes(x = PC1, y = PC2,
                                 color = Type)) +
    geom_point(alpha = 0.6, size = 2) +
    labs(title = "PCA: First Two Components",
         x = "PC1", y = "PC2") +
    scale_color_manual(values = c("turquoise", "coral")) +
    theme_minimal() +
    theme(
      legend.position = "bottom",
      axis.title = element_text(size = 15),
      text = element_text(size = rel(4.25)),
      legend.text = element_text(size = 13),
      legend.title = element_text(size = 13.5),
      axis.text.y = element_text(family = "mono"),
      plot.title = element_text(size = 16, hjust = 0.5))

  # Return PCA results
  return(list(
    pca_result = pca_result,
    scores = scores,
    var_expl = var_expl,
    scree_plot = scree_plot,
    pca_plot = pca_plot))
}
```