Introduction to R and Data Visualization

2025 Annual AMMNET Meeting, Dakar, Senegal

Presented by Prof. Dennis Muriithi, PhD.

Chuka University - Center for Data Analytics and Modeling, Kenya

Table of Contents

4. **Data Manipulation with `dplyr`**
    - `select()`: Selecting Columns
    - `filter()`: Filtering Rows
    - `mutate()`: Creating New Variables
    - `group_by()` and `summarize()`: Grouped Operations
    - `arrange()`: Sorting Data
5. **Introduction to Data Visualization with `ggplot2`**
    - Grammar of Graphics
    - `ggplot()`: Initializing a Plot
    - `aes()`: Aesthetic Mappings
    - `geom_point()`: Scatter Plots
    - `geom_line()`: Line Plots
    - `geom_bar()`: Bar Charts
    - `geom_histogram()`: Histograms
    - `geom_boxplot()`: Box Plots
6. **Customizing `ggplot2` Visualizations**
    - Adding Titles and Labels
    - Changing Colors and Themes
    - Faceting: Creating Small Multiples
7. **Case Study: Visualizing Malaria Data (Example)**
    - Loading Sample Data
    - Exploring Data Trends
    - Creating Informative Visualizations
8. **Conclusion and Q&A**
    - Key Takeaways
    - Further Resources

- Questions and Discussion

---

1. Introduction to R

What is R?

R is a powerful open-source programming language and software environment primarily used for statistical computing and graphics. It provides a wide variety of statistical (linear and nonlinear modeling, classical statistical tests, time-series analysis, classification, clustering, etc.) and graphical techniques, and is highly extensible. The R language is widely used among statisticians and data miners for developing statistical software and data analysis.

Why R for Data Analysis?

- **Open Source and Free:** R is freely available, making it accessible to everyone without licensing costs.
- **Comprehensive Statistical Capabilities:** R offers an unparalleled collection of statistical techniques and models.
- **Powerful Data Visualization:** R's graphical capabilities, especially with packages like `ggplot2`, are world-class.
- **Vibrant Community and Extensive Packages:** A large and active community contributes to thousands of user-contributed packages, extending R's functionality.
- **Cross-Platform Compatibility:** R runs on various operating systems, including Windows, macOS, and Linux.

RStudio Interface Overview

RStudio is an integrated development environment (IDE) for R. It provides a user-friendly interface that simplifies working with R. Key panes in RStudio include:

- **Console:** Where you type R commands and see their output.
- **Source Editor:** Where you write and save your R scripts.

- **Environment/History:** Shows objects in your current R session and command history.
- **Files/Plots/Packages/Help/Viewer:** Manages files, displays plots, handles packages, provides help documentation, and views web content.

Basic R Syntax and Operations

Let's start with some basic R operations. You can use R as a calculator:

```
# Addition
5 + 3

# Subtraction
10 - 4

# Multiplication
6 * 7

# Division
20 / 5

# Exponentiation
2^3

# Modulo (remainder)
10 %% 3
```

Assigning values to variables:

```
# Assigning a value to a variable
x <- 10
y = 5 # Another way to assign

# Performing operations with variables
x + y
```

---

2. Data Structures in R

R has several fundamental data structures that are crucial for organizing and manipulating data. Understanding these structures is key to effective R programming.

Vectors

Vectors are the most basic R data objects. They are ordered collections of elements of the *same data type*. You can create vectors using the `c()` function (combine).

```r
# Numeric vector
numeric_vector <- c(1, 5, 8, 12)
print(numeric_vector)

# Character vector
character_vector <- c("apple", "banana", "cherry")
print(character_vector)

# Logical vector
logical_vector <- c(TRUE, FALSE, TRUE)
print(logical_vector)

# Accessing elements of a vector
numeric_vector[1] # Access the first element
numeric_vector[c(2, 4)] # Access the second and fourth elements
```

Matrices

A matrix is a two-dimensional rectangular data set. It can be created using the `matrix()` function. All elements in a matrix must be of the *same data type*.

```r
# Create a 2x3 matrix
matrix_data <- matrix(c(1, 2, 3, 4, 5, 6), nrow = 2, ncol = 3, byrow = TRUE)
print(matrix_data)

# Accessing elements of a matrix
matrix_data[1, 2] # Element in row 1, column 2
matrix_data[ , 1] # All elements in column 1
matrix_data[2, ] # All elements in row 2
```

Arrays

Arrays are similar to matrices but can have more than two dimensions. They are used to store data in more than two dimensions. All elements in an array must be of the *same data type*.

```r
# Create a 2x3x2 array
array_data <- array(c(1:12), dim = c(2, 3, 2))
print(array_data)

# Accessing elements of an array
array_data[1, 2, 1] # Element in row 1, column 2, dimension 1
```

Data Frames

A data frame is a list of vectors of equal length. It is the most important data structure in R for storing tabular data. Unlike matrices, different columns in a data frame can have *different data types*.

```r
# Create a data frame
df_data <- data.frame(
  Name = c("Alice", "Bob", "Charlie"),
  Age = c(25, 30, 22),
  IsStudent = c(TRUE, FALSE, TRUE)
)
print(df_data)

# Accessing elements of a data frame
df_data$Name # Access column by name
df_data[1, ] # Access first row
df_data[ , "Age"] # Access column by name (another way)
```

Lists

A list is a generic vector containing other objects. It can contain elements of *different data types* and even other data structures (like vectors, matrices, or data frames).

```r
# Create a list
list_data <- list(
  name = "John Doe",
  age = 30,
  scores = c(95, 88, 92),
  is_active = TRUE,
  matrix_example = matrix(1:4, nrow = 2)
)
print(list_data)

# Accessing elements of a list
list_data$name # Access by name
list_data[[3]] # Access by index
list_data[["scores"]][1] # Access element within a list element
```

---

3. Data Import and Export

One of the first steps in any data analysis project is getting your data into R and, eventually, saving your results. R provides various functions for reading and writing different file formats.

Reading CSV/Text Files

Comma Separated Values (CSV) and plain text files are common formats for sharing data. R makes it easy to read these files using `read.csv()` or `read.table()`.

```
# Create a dummy CSV file for demonstration
write.csv(data.frame(ID = 1:3, Value = c(10, 20, 30)), "dummy_data.csv",
row.names = FALSE)

# Read a CSV file
my_data_csv <- read.csv("dummy_data.csv")
print(my_data_csv)

# Read a text file (e.g., tab-separated)
# Assuming 'my_data.txt' is a tab-separated file
# my_data_txt <- read.table("my_data.txt", header = TRUE, sep = "\t")
# print(my_data_txt)
```

Reading Excel Files

While R has built-in functions for CSV and text files, for Excel files (`.xls` or `.xlsx`), you typically need to install and load a package like `readxl`.

First, install the package (if you haven't already):

```
# install.packages("readxl") # Run this line once to install the package
```

Then, load the package and read the Excel file:

```
library(readxl)

# Create a dummy Excel file for demonstration (requires openxlsx package or
manual creation)
# For simplicity, we'll assume you have an Excel file named 'dummy_excel.xlsx'
# with a sheet named 'Sheet1' and columns 'ColA', 'ColB'

# Example of reading an Excel file
# my_data_excel <- read_excel("dummy_excel.xlsx", sheet = "Sheet1")
# print(my_data_excel)
```

Saving Data

After performing your analysis, you'll often want to save your processed data or results. You can save data frames back to CSV or other formats.

```r
# Save a data frame to a CSV file
write.csv(my_data_csv, "processed_data.csv", row.names = FALSE)

# Save an R object (e.g., a data frame) to an R data file (.RData or .rda)
# This preserves R-specific data types and structures
save(my_data_csv, file = "my_data.RData")

# To load it back later:
# load("my_data.RData")
# print(my_data_csv)
```

---

4. Data Manipulation with `dplyr`

`dplyr` is a powerful and popular R package for data manipulation. It provides a consistent set of verbs that make data wrangling intuitive and efficient. To use `dplyr`, you first need to install and load it.

```r
# install.packages("dplyr") # Run this line once to install the package
library(dplyr)

# Let's create a sample data frame for demonstration
sample_data <- data.frame(
  PatientID = 101:105,
  Age = c(25, 30, 35, 40, 45),
  Gender = c("M", "F", "M", "F", "M"),
  MalariaStatus = c("Positive", "Negative", "Positive", "Negative",
"Positive"),
  Temperature = c(38.5, 36.8, 39.2, 37.1, 38.9)
)
print(sample_data)
```

`select()`: Selecting Columns

The `select()` function allows you to pick variables (columns) from a data frame.

```r
# Select specific columns
selected_cols <- sample_data %>% select(PatientID, Age, MalariaStatus)
print(selected_cols)

# Select all columns except one
all_except_gender <- sample_data %>% select(-Gender)
print(all_except_gender)
```

`filter()`: Filtering Rows

The `filter()` function allows you to choose rows based on their values.

```
# Filter rows where MalariaStatus is 'Positive'
positive_cases <- sample_data %>% filter(MalariaStatus == "Positive")
print(positive_cases)

# Filter rows based on multiple conditions
old_male_patients <- sample_data %>% filter(Age > 30, Gender == "M")
print(old_male_patients)
```

`mutate()`: Creating New Variables

The `mutate()` function adds new variables that are functions of existing variables.

```
# Add a new column for Temperature in Fahrenheit
fahrenheit_temp <- sample_data %>% mutate(TemperatureF = (Temperature * 9/5) +
32)
print(fahrenheit_temp)
```

`group_by()` and `summarize()`: Grouped Operations

These functions are often used together to perform summary operations on grouped data.

```
# Calculate the average temperature by MalariaStatus
avg_temp_by_status <- sample_data %>%
  group_by(MalariaStatus) %>%
  summarize(AverageTemperature = mean(Temperature))
print(avg_temp_by_status)
```

`arrange()`: Sorting Data

The `arrange()` function orders rows by column values.

```
# Arrange data by Age in ascending order
sorted_by_age <- sample_data %>% arrange(Age)
print(sorted_by_age)

# Arrange data by Age in descending order
sorted_by_age_desc <- sample_data %>% arrange(desc(Age))
print(sorted_by_age_desc)
```

---

5. Introduction to Data Visualization with `ggplot2`

`ggplot2` is an elegant and versatile data visualization package in R, based on the "Grammar of Graphics." This grammar provides a structured way to think about and build plots, allowing for highly customizable and informative visualizations. To use `ggplot2`, you first need to install and load it.

```
# install.packages("ggplot2") # Run this line once to install the package
library(ggplot2)

# Let's create a sample dataset for visualization examples
set.seed(123) # for reproducibility
visualization_data <- data.frame(
  Category = factor(rep(c("A", "B", "C"), each = 10)),
  Value = c(rnorm(10, 5, 1), rnorm(10, 7, 1.5), rnorm(10, 6, 0.8)),
  Count = sample(1:100, 30, replace = TRUE),
  Date = seq(as.Date("2024-01-01"), by = "month", length.out = 30)
)
print(head(visualization_data))
```

Grammar of Graphics

The Grammar of Graphics breaks down plots into components: * **Data:** The dataset you want to visualize. * **Aesthetics (aes):** How variables in your data are mapped to visual properties (e.g., x-axis, y-axis, color, size, shape). * **Geometries (geom):** The visual marks that represent data points (e.g., points, lines, bars, boxes). * **Facets:** How to split data into subsets and plot them side-by-side. * **Statistics (stat):** Statistical transformations (e.g., binning for histograms, smoothing for lines). * **Coordinates (coord):** The coordinate system used (e.g., Cartesian, polar). * **Themes:** Overall visual appearance (e.g., fonts, colors, background).

`ggplot()`: Initializing a Plot

Every `ggplot2` plot starts with the `ggplot()` function, where you specify the data and global aesthetic mappings.

```
# Initialize a ggplot object with data
p <- ggplot(data = visualization_data)
print(p) # This will show an empty plot as no geom is added yet
```

`aes()`: Aesthetic Mappings

The `aes()` function maps variables from your dataset to visual properties of the plot. These mappings can be global (in `ggplot()`) or local (in a `geom_` function).

```
# Map 'Value' to x-axis and 'Count' to y-axis
p_aes <- ggplot(data = visualization_data, aes(x = Value, y = Count))
print(p_aes) # Still empty, but mappings are set
```

`geom_point()`: Scatter Plots

`geom_point()` is used to create scatter plots, showing the relationship between two continuous variables.

```
# Scatter plot of Value vs. Count
scatter_plot <- ggplot(data = visualization_data, aes(x = Value, y = Count)) +
  geom_point()
print(scatter_plot)

# Scatter plot with color mapped to Category
scatter_color_plot <- ggplot(data = visualization_data, aes(x = Value, y =
Count, color = Category)) +
  geom_point()
print(scatter_color_plot)
```

`geom_line()`: Line Plots

`geom_line()` is used for visualizing trends over a continuous variable, often time.

```
# Line plot of Value over Date
line_plot <- ggplot(data = visualization_data, aes(x = Date, y = Value)) +
  geom_line()
print(line_plot)

# Line plot with different lines for each Category
line_group_plot <- ggplot(data = visualization_data, aes(x = Date, y = Value,
group = Category, color = Category)) +
  geom_line()
print(line_group_plot)
```

`geom_bar()`: Bar Charts

`geom_bar()` is used to create bar charts, typically for displaying counts of categorical variables. By default, it calculates counts.

```r
# Bar chart of Category counts
bar_chart <- ggplot(data = visualization_data, aes(x = Category)) +
  geom_bar()
print(bar_chart)

# Bar chart of pre-summarized data (e.g., sum of Count by Category)
# First, summarize the data
summary_data <- visualization_data %>%
  group_by(Category) %>%
  summarize(TotalCount = sum(Count))

bar_chart_summarized <- ggplot(data = summary_data, aes(x = Category, y =
TotalCount)) +
  geom_bar(stat = "identity") # Use stat="identity" when y-values are already
counts/sums
print(bar_chart_summarized)
```

`geom_histogram()`: Histograms

`geom_histogram()` displays the distribution of a single continuous variable by dividing the data into bins and counting observations in each bin.

```r
# Histogram of Value
histogram_plot <- ggplot(data = visualization_data, aes(x = Value)) +
  geom_histogram(binwidth = 0.5, fill = "skyblue", color = "black")
print(histogram_plot)
```

`geom_boxplot()`: Box Plots

`geom_boxplot()` is used to visualize the distribution of a continuous variable across different categories, showing median, quartiles, and outliers.

```r
# Box plot of Value by Category
boxplot_plot <- ggplot(data = visualization_data, aes(x = Category, y = Value))
+
  geom_boxplot(fill = "lightgreen")
print(boxplot_plot)
```

---

6. Customizing `ggplot2` Visualizations

`ggplot2` offers extensive options for customizing your plots to make them more informative, aesthetically pleasing, and suitable for your audience. Here, we'll cover adding titles, labels, changing colors, themes, and faceting.

Adding Titles and Labels

Clear titles and labels are essential for making your plots understandable.

```
# Add a main title, subtitle, and axis labels
custom_labels_plot <- ggplot(data = visualization_data, aes(x = Value, y =
Count, color = Category)) +
  geom_point() +
  labs(
    title = "Relationship Between Value and Count by Category",
    subtitle = "Sample Data from 2024",
    x = "Measured Value",
    y = "Observed Count",
    color = "Data Category"
  )
print(custom_labels_plot)
```

Changing Colors and Themes

You can control the colors of your plot elements and apply different themes to change the overall look.

```
# Change point colors manually and use a built-in theme
custom_colors_theme_plot <- ggplot(data = visualization_data, aes(x = Value, y
= Count, color = Category)) +
  geom_point() +
  scale_color_manual(values = c("A" = "red", "B" = "blue", "C" = "darkgreen"))
+
  theme_minimal() # Other themes: theme_classic(), theme_bw(), theme_dark()
print(custom_colors_theme_plot)

# Change fill color for bar chart
bar_chart_custom_fill <- ggplot(data = visualization_data, aes(x = Category)) +
  geom_bar(fill = "purple", color = "black") +
  theme_light()
print(bar_chart_custom_fill)
```

Faceting: Creating Small Multiples

Faceting allows you to create multiple plots, one for each level of a categorical variable, making it easy to compare distributions or relationships across groups.

```r
# Facet by Category using facet_wrap()
faceted_plot_wrap <- ggplot(data = visualization_data, aes(x = Value, y =
Count)) +
  geom_point() +
  facet_wrap(~ Category, scales = "free") # scales = "free" allows independent
axes
print(faceted_plot_wrap)

# Facet by Category using facet_grid() (useful for two categorical variables)
# Let's add another dummy variable for demonstration
visualization_data$Group <- rep(c("X", "Y"), each = 15)

faceted_plot_grid <- ggplot(data = visualization_data, aes(x = Value, y =
Count)) +
  geom_point() +
  facet_grid(Group ~ Category)
print(faceted_plot_grid)
```

---

7. Case Study: Visualizing Malaria Data (Example)

In this section, we will walk through a simplified case study to demonstrate how R and `ggplot2` can be used to visualize malaria-related data. We will create a synthetic dataset for this purpose, mimicking common data points you might encounter in public health.

Loading Sample Data

Let's imagine we have data on malaria cases reported in different regions over several months. We'll create a data frame to simulate this.

```r
# Create a synthetic malaria dataset
set.seed(42) # for reproducibility
malaria_data <- data.frame(
  Month = rep(seq(as.Date("2024-01-01"), by = "month", length.out = 12), each =
3),
  Region = rep(c("North", "Central", "South"), 12),
  Cases = round(runif(36, min = 50, max = 300) +
                rep(c(0, 50, 100), each = 12) + # Add regional baseline
difference
                rep(c(0, 20, -10, 5, 10, 30, 40, 25, 15, 5, -5, -15), 3) # Add
seasonal variation
                ),
  Intervention = sample(c("None", "Bed Nets", "IRS"), 36, replace = TRUE, prob
= c(0.5, 0.3, 0.2))
)

# Ensure cases are non-negative
malaria_data$`Cases[malaria_data`$Cases < 0] <- 0

print(head(malaria_data))
print(tail(malaria_data))
```

Exploring Data Trends

Before visualizing, it's good practice to get a quick overview of the data.

```r
# Summary statistics
summary(malaria_data)

# Check for unique regions
unique(malaria_data$Region)

# Check for unique months
unique(malaria_data$Month)

# Calculate total cases per region
library(dplyr) # Ensure dplyr is loaded
total_cases_by_region <- malaria_data %>%
  group_by(Region) %>%
  summarize(TotalCases = sum(Cases))
print(total_cases_by_region)
```

Creating Informative Visualizations

Now, let's create some visualizations to understand the trends in our synthetic malaria data.

1. Malaria Cases Over Time by Region (Line Plot)

This plot helps us see seasonal trends and differences between regions.

```
line_plot_malaria <- ggplot(malaria_data, aes(x = Month, y = Cases, color =
Region, group = Region)) +
  geom_line() +
  geom_point() +
  labs(
    title = "Malaria Cases Over Time by Region (Synthetic Data)",
    x = "Month",
    y = "Number of Cases",
    color = "Region"
  ) +
  theme_minimal()
print(line_plot_malaria)
```

2. Distribution of Cases by Intervention Type (Box Plot)

This helps us understand if there's a difference in case numbers based on the type of intervention.

```
boxplot_malaria_intervention <- ggplot(malaria_data, aes(x = Intervention, y =
Cases, fill = Intervention)) +
  geom_boxplot() +
  labs(
    title = "Distribution of Malaria Cases by Intervention Type",
    x = "Intervention Type",
    y = "Number of Cases",
    fill = "Intervention"
  ) +
  theme_light()
print(boxplot_malaria_intervention)
```

3. Total Cases by Region (Bar Chart)

A simple bar chart to compare the total number of cases across different regions.

```
bar_chart_total_cases <- ggplot(total_cases_by_region, aes(x = Region, y =
TotalCases, fill = Region)) +
  geom_bar(stat = "identity") +
  labs(
    title = "Total Malaria Cases by Region",
    x = "Region",
    y = "Total Number of Cases",
    fill = "Region"
  ) +
  theme_classic()
print(bar_chart_total_cases)
```

These examples demonstrate how R and `ggplot2` can be used to quickly generate insightful visualizations from public health data, aiding in trend analysis and decision-making.

---

8. Conclusion and Q&A

Key Takeaways

- **R is a powerful, open-source tool** for statistical computing and graphics, widely used in academia and industry.
- **Understanding R's data structures** (vectors, matrices, data frames, lists) is fundamental for effective data handling.
- **`dplyr` simplifies data manipulation** with its intuitive verbs (`select`, `filter`, `mutate`, `group_by`, `summarize`, `arrange`).
- **`ggplot2` provides a robust framework** for creating highly customizable and informative data visualizations based on the Grammar of Graphics.
- **Effective data visualization** is crucial for communicating insights and supporting data-driven decision-making, especially in fields like public health.

Further Resources

- **R for Data Science:** https://r4ds.had.co.nz/
- **ggplot2 Official Website:** https://ggplot2.tidyverse.org/
- **RStudio Cheatsheets:** https://posit.co/resources/cheatsheets/
- **CRAN (The Comprehensive R Archive Network):** https://cran.r-project.org/

Questions and Discussion

Thank you for your participation! We now open the floor for any questions or discussions you may have.
