Plant Genetics Data Analysis with R - Baku

A Practical Guide for Breeders

Your Name / Breeding Institute Baku (ICARDA Collaboration)

2025-06-09

Table of contents

1 Welcome – Plant Genetics Data Analysis with R

2 Welcome to the Plant Genetics Data Analysis Course

Target Audience: Breeders in Baku, Azerbaijan with minimal prior data analysis or programming experience. Collaboration with ICARDA.

Goal: To provide a practical and understandable introduction to analyzing common breeding and genomic data types using the R programming language.

This course covers fundamental concepts in genetics and statistics relevant to breeding programs, alongside hands-on R coding sessions. We aim to build your confidence in handling your own data and interpreting results.

Please use the navigation menu (Table of Contents) to move through the course modules.

Let's begin!

3 Meet Your Instructors & the ICARDA Bioinformatics Unit

The ICARDA Bioinformatics Unit is at the forefront of applying cutting-edge computational biology to address agricultural challenges in dry areas. Our multidisciplinary team specializes in genomics, data science, AI, and high-performance computing to empower your data analysis journey.

Photo	Name	Details
	**Zakaria Kehel**	Research Team Leader – Genetic Resources (GRS) & Morocco Interim Country Manager Genetic Innovation Ph.D. from the School of Agricultural and Forestry Engineering at the University of Córdoba in Spain.

Photo	Name	Details
	**Tamara Ortiz**	Bioinformatician, started February 2024 MS in Bioinformatics, NYU Tandon School of Engineering (2025–2026) BE in Bioengineering, UTEC – Universidad de Ingeniería y Tecnología (2018–2023)

Alsamman M. Alsamman Bioinformatician, started January 2021 PhD in Genetics, Faculty of Agriculture, Ain Shams University; MSc in Genetics, Faculty of Agriculture, Cairo University; BSc in Biotechnology with a Bioinformatics Minor, Faculty of Agriculture, Al-Azhar University.

Part I

Introduction and Setup

4 Welcome and Course Overview

4.1 Hello Baku Breeders!

Welcome to this introductory course on data analysis for plant genetics, a collaboration with ICARDA. We are excited to guide you through the essential tools and concepts needed to make sense of your valuable breeding data using R.

4.2 Course Objectives

- Learn the fundamentals of the R programming language for data tasks.
- Understand basic concepts of genomic data.
- Perform basic data loading, cleaning, and quality control.
- Grasp key genetic concepts like allele frequency and relatedness (kinship).
- Understand the idea behind marker-trait association studies (GWAS).
- Get introduced to tools like GIGWA and basic AI applications.

4.3 Course Structure

This course is divided into several modules, starting with setup and R basics, moving through data handling and genetic concepts, and ending with analysis methods and tools. Each module includes explanations and practical R exercises.

No prior programming experience is required!

5 Setting Up Your Environment: R and RStudio

5.1 Why R and RStudio?

- **R:** A powerful, free programming language specifically designed for statistical computing and graphics. Widely used in academia and industry for data analysis, including genomics and breeding.
- **RStudio:** An excellent, free Integrated Development Environment (IDE) for R. It makes using R much easier with features like code highlighting, plot viewing, package management, and project organization.

5.2 Installation Steps

1. **Install R:** Go to CRAN (the Comprehensive R Archive Network) and download the latest version for your operating system (Windows, macOS, Linux). Follow the installation instructions.
2. **Install RStudio:** Go to the Posit website and download the free RStudio Desktop version for your operating system. Install it after installing R.
3. **Install Quarto:** Go to Quarto's website and download and install Quarto for your system. RStudio often bundles Quarto, but installing the latest version is good practice.
4. **(For PDF Output) Install LaTeX:** Open RStudio, go to the Console panel, and type the following commands one by one, pressing Enter after each:

```
# Run these lines in the R Console
install.packages("tinytex")
# Run only once if you don't have it
# tinytex::install_tinytex()
Run only once to install LaTeX distribution
# This might take a few minutes. If it fails, consult TinyTeX documentation or ask instructo
```

```r
# Installing R Packages for the Course
# We will use several add-on packages in R. You only need to install packages *once*. Use the
# --- Run this code chunk in the R Console --- List of packages we will likely need:

# Packages in Cran
cran_packages <- c(
  "tidyverse", "readxl", "writexl", "readr", "qqman", "vcfR", "QBMS", "adegenet",
  "ade4", "ggiraph", "ggpubr", "plotly", "poppr", "reactable",
  "rnaturalearth", "scatterpie", "snpReady", "viridis", "tibble",
  "ggplot2", "reshape2", "forcats", "dplyr", "sp", "scales", "htmltools",
  "ASRgenomics", "statgenGWAS", "gplots"
)

# Bioconductor Packages
bioc_packages <- c("rrBLUP", "LEA")

# Installing Cran Packages
installed <- rownames(installed.packages())
missing <- setdiff(cran_packages, installed)
if (length(missing)) {
  message("Installing missing CRAN packages: ", paste(missing, collapse = ", "))
  install.packages(missing)
}

# Installing Bioconductor Packages
if (!requireNamespace("BiocManager", quietly = TRUE)) {
  install.packages("BiocManager")
}
installed <- rownames(installed.packages())
missing <- setdiff(bioc_packages, installed)
if (length(missing)) {
  message("Installing missing Bioconductor packages: ", paste(missing, collapse = ", "))
  BiocManager::install(missing)
}
```

5.3 Loading Libraries and Functions

```r
# You can use library() to load any single package
# We will load all libraries using lapply()
invisible(
```

```
  suppressPackageStartupMessages(
    lapply(c(cran_packages, bioc_packages), library, character.only = TRUE)
  )
)
```

5.4 Quick RStudio Tour

(We will cover this live, but key windows include: Console, Script Editor/Notebook, Environment/History, Files/Plots/Packages/Help/Viewer/Projects). Familiarize yourself with these panes.

Part II

R Programming Fundamentals

6 Module 1.1: Introduction to R - Your Breeding Data Analysis Tool

6.0.1 Introduction to R

R is a powerful language for data manipulation, visualization, and statistical analysis. Think of R as a versatile calculator for data.

- **What is R?** Think of R as a powerful, specialized calculator combined with a programming language. It's designed specifically for handling data, performing statistical analyses, and creating informative graphs.
- **Why R for Breeding?**
  - **Free & Open Source:** Anyone can use it without cost.
  - **Powerful for Data:** Excellent at handling the types of large datasets we generate in breeding (phenotypes, genotypes).
  - **Cutting-Edge Statistics:** Many new statistical methods (like those for genomic selection or GWAS) are first available as R packages.
  - **Great Graphics:** Create publication-quality plots to visualize your results.
  - **Large Community:** Lots of help available online and specialized packages for genetics and breeding (like `rrBLUP` which we might see later).
- **R vs. Excel:** Excel is great for data entry and simple summaries, but R is much better for complex analysis, automation, reproducible research, and handling very large datasets.

Try these examples in the RStudio Console:

```
# Basic arithmetic
2 + 5
```

```
[1] 7
```

```
10 - 3
```

```
[1] 7
```

```r
4 * 8
```

```
[1] 32
```

```r
100 / 4
```

```
[1] 25
```

```r
# Order of operations (like standard math)
5 + 2 * 3   # Multiplication first
```

```
[1] 11
```

```r
(5 + 2) * 3 # Parentheses first
```

```
[1] 21
```

```r
# Built-in mathematical functions
sqrt(16)    # Square root
```

```
[1] 4
```

```r
log(10)     # Natural logarithm
```

```
[1] 2.302585
```

```r
log10(100)  # Base-10 logarithm
```

```
[1] 2
```

6.1 Variables: Storing Information

Variables are used to store information in R. You can think of them as containers for data. In R, you can create variables using the assignment operator `<-`. You can also use `=` for assignment, but `<-` is more common in R.

Use the `<-` operator to assign and manipulate variables:

```r
# Assign the value 5 to variable x
x <- 5

# Assign the result of 10 + 3 to variable y
y <- 10 + 3

# Print the value of x
x
```

```
[1] 5
```

```r
# Use variables in calculations
z <- x + y
# Print the value of z
z
```

```
[1] 18
```

```r
# Assign the name of a variety to a variable
best_variety <- "ICARDA_Gold" # Text needs quotes ""

# Print name
print(best_variety)
```

```
[1] "ICARDA_Gold"
```

```r
# We can also concatenate text like this
print(paste("The best variety is", best_variety))
```

```
[1] "The best variety is ICARDA_Gold"
```

7 Module 1.2: R Data Types and Structures - The Building Blocks

7.1 Introduction: Types and Structures

Think of data like building blocks:

- **Data Types:** The *kind* of block (e.g., numeric brick, text brick, true/false switch).
- **Data Structures:** How you *organize* those blocks (e.g., a single row of bricks, a flat grid, a complex box holding different things).

Understanding these is fundamental to working with data in R.

7.2 Basic Data Types

R needs to know what *kind* of information it's dealing with.

1. **Numeric:** Represents numbers. Can be integers (whole numbers) or doubles (with decimals). Used for measurements like yield, height, counts.

```
yield <- 75.5     # Double (decimal)
num_plots <- 120  # Integer (whole number)
class(yield)      # Check the type
```

```
[1] "numeric"
```

```
class(num_plots)  # Often stored as 'numeric' (double) by default
```

```
[1] "numeric"
```

2. **Character:** Represents text (strings). Always enclose text in double (") or single (') quotes. Used for IDs, names, descriptions.

```r
variety_name <- "ICARDA_RustResist"
plot_id <- 'Plot_A101'
class(variety_name)
```

```
[1] "character"
```

3. **Logical:** Represents TRUE or FALSE values. Often the result of comparisons. Crucial for filtering data.

```r
is_resistant <- TRUE
yield > 80 # This comparison results in a logical value
```

```
[1] FALSE
```

```r
class(is_resistant)
```

```
[1] "logical"
```

4. **Factor:** Special type for categorical data (variables with distinct levels or groups). R stores them efficiently using underlying numbers but displays the text labels. Very important for statistical models and plotting.

```r
# Example: Different locations in a trial
locations <- c("Baku", "Ganja", "Baku", "Sheki", "Ganja")
location_factor <- factor(locations)

print(location_factor) # Shows levels
```

```
[1] Baku  Ganja Baku  Sheki Ganja
Levels: Baku Ganja Sheki
```

```r
class(location_factor)
```

```
[1] "factor"
```

```r
levels(location_factor) # See the unique categories
```

```
[1] "Baku"  "Ganja" "Sheki"
```

7.3 Key Data Structures

How R organizes collections of data:

1. **Vector:** The most basic structure! A sequence (ordered list) containing elements **of the same data type**. Created using `c()` (combine function).

```
# Vector of plot yields (numeric)
plot_yields <- c(75.5, 81.2, 78.9, 85.0)
# Vector of variety names (character)
plot_varieties <- c("ICARDA_Gold", "Local_Check", "ICARDA_Gold", "ICARDA_RustResist")
# Vector of resistance status (logical)
plot_resistance <- c(TRUE, FALSE, TRUE, TRUE)

plot_yields[1]       # Access the first element (Indexing starts at 1!)
```

```
[1] 75.5
```

```
plot_yields[2:4]     # Access elements 2 through 4
```

```
[1] 81.2 78.9 85.0
```

```
length(plot_yields)  # Get the number of elements
```

```
[1] 4
```

Important: If you mix types in `c()`, R will force them into a single common type (usually character).

```
mixed_vector <- c(10, "VarietyA", TRUE)
print(mixed_vector) # All become character strings!
```

```
[1] "10"       "VarietyA" "TRUE"
```

```
class(mixed_vector)# Example: Small genotype matrix (Individuals x SNPs)
```

```
[1] "character"
```

2. **Matrix:** A two-dimensional grid (rows and columns) where all elements **must be of the same data type**. Useful for genotype data (0,1,2 are all numeric).

```r
# Example: Small genotype matrix (Individuals x SNPs)
 genotype_data <- matrix(c(0, 1, 2, 1, 1, 0), nrow = 2, ncol = 3, byrow = TRUE)
 rownames(genotype_data) <- c("Line1", "Line2")
 colnames(genotype_data) <- c("SNP1", "SNP2", "SNP3")
 print(genotype_data)
```

```
      SNP1 SNP2 SNP3
Line1    0    1    2
Line2    1    1    0
```

```r
 class(genotype_data)
```

```
[1] "matrix" "array"
```

```r
 dim(genotype_data) # Get dimensions (rows, columns)
```

```
[1] 2 3
```

```r
 genotype_data[1, 2] # Access element row 1, column 2
```

```
[1] 1
```

3. **Data Frame: The most important data structure for breeders!** Like a spreadsheet or table in R.
    - It's a collection of vectors (columns) of equal length.
    - **Crucially, columns can be of different data types!** (e.g., character ID, numeric yield, factor location).
    - Rows represent observations (e.g., plots, plants, samples).
    - Columns represent variables (e.g., ID, traits, treatments).

```r
# Create a simple breeding trial data frame
trial_data <- data.frame(
 PlotID = c("A101", "A102", "B101", "B102"),
 Variety = factor(c("ICARDA_Gold", "Local_Check", "ICARDA_RustResist", "ICARDA_Gold")),
 Yield_kg_plot = c(5.2, 4.5, 6.1, 5.5),
 Is_Resistant = c(TRUE, FALSE, TRUE, TRUE)
)

print(trial_data)
```

```
  PlotID           Variety Yield_kg_plot Is_Resistant
1   A101       ICARDA_Gold           5.2         TRUE
2   A102       Local_Check           4.5        FALSE
3   B101 ICARDA_RustResist           6.1         TRUE
4   B102       ICARDA_Gold           5.5         TRUE
```

```
class(trial_data)
```

```
[1] "data.frame"
```

```
str(trial_data)       # Structure: Shows types of each column - VERY USEFUL!
```

```
'data.frame':   4 obs. of  4 variables:
 $ PlotID       : chr  "A101" "A102" "B101" "B102"
 $ Variety      : Factor w/ 3 levels "ICARDA_Gold",..: 1 3 2 1
 $ Yield_kg_plot: num  5.2 4.5 6.1 5.5
 $ Is_Resistant : logi  TRUE FALSE TRUE TRUE
```

```
head(trial_data)      # Show first few rows
```

```
  PlotID           Variety Yield_kg_plot Is_Resistant
1   A101       ICARDA_Gold           5.2         TRUE
2   A102       Local_Check           4.5        FALSE
3   B101 ICARDA_RustResist           6.1         TRUE
4   B102       ICARDA_Gold           5.5         TRUE
```

```
summary(trial_data)   # Summary statistics for each column
```

```
   PlotID                      Variety  Yield_kg_plot   Is_Resistant
 Length:4           ICARDA_Gold      :2   Min.   :4.500   Mode :logical
 Class :character   ICARDA_RustResist:1   1st Qu.:5.025   FALSE:1
 Mode  :character   Local_Check      :1   Median :5.350   TRUE :3
                                         Mean   :5.325
                                         3rd Qu.:5.650
                                         Max.   :6.100
```

```
# Access columns using $
trial_data$Yield_kg_plot
```

```
[1] 5.2 4.5 6.1 5.5
```

```
mean(trial_data$Yield_kg_plot) # Calculate mean of a column
```

```
[1] 5.325
```

(We will work extensively with data frames).

4. **List:** A very flexible container that can hold *any* collection of R objects (vectors, matrices, data frames, even other lists), and they don't have to be the same type or length. Often used to return complex results from functions.

```
analysis_results <- list(
   description = "Yield Trial - Baku 2023",
   raw_data = trial_data, # Include the data frame
   significant_snps = c("SNP101", "SNP504"), # A character vector
   model_parameters = list(threshold = 0.05, method = "MLM") # A nested list
)
print(analysis_results$description)
```

```
[1] "Yield Trial - Baku 2023"
```

```
print(analysis_results$raw_data) # Access the data frame inside the list
```

```
  PlotID           Variety Yield_kg_plot Is_Resistant
1   A101       ICARDA_Gold           5.2         TRUE
2   A102       Local_Check           4.5        FALSE
3   B101 ICARDA_RustResist           6.1         TRUE
4   B102       ICARDA_Gold           5.5         TRUE
```

8 Module 1.3: Basic Operations in R

Now that we know about data types and structures, let's see how to manipulate them.

8.1 Arithmetic Operations (Review)

Works on numbers and numeric vectors/matrices element-wise.

```
mixed_vector <- c(10, "VarietyA", TRUE)
print(mixed_vector) # All become character strings!
```

```
[1] "10"       "VarietyA" "TRUE"
```

```
class(mixed_vector)
```

```
[1] "character"
```

8.2 Logical Comparisons and Operators

Used to ask TRUE/FALSE questions about our data. Essential for filtering.

- **Comparison Operators:**
  - `>` : Greater than
  - `<` : Less than
  - `>=`: Greater than or equal to
  - `<=`: Less than or equal to
  - `==`: **Exactly equal to** (TWO equal signs! Very common mistake to use just one `=`)
  - `!=`: Not equal to
- **Logical Operators (Combine TRUE/FALSE):**
  - `&` : AND (both sides must be TRUE)

- `|` : OR (at least one side must be TRUE)
- `!` : NOT (reverses TRUE to FALSE, FALSE to TRUE)

```
yield <- 5.2
min_acceptable_yield <- 5.0
variety <- "ICARDA_Gold"

# Comparisons
yield > min_acceptable_yield # Is yield acceptable? TRUE
```

```
[1] TRUE
```

```
variety == "Local_Check"    # Is it the local check? FALSE
```

```
[1] FALSE
```

```
variety != "Local_Check"    # Is it NOT the local check? TRUE
```

```
[1] TRUE
```

```
# On vectors
plot_yields <- c(5.2, 4.5, 6.1, 5.5)
plot_yields > 5.0 # Which plots yielded above 5.0? [TRUE FALSE TRUE TRUE]
```

```
[1]  TRUE FALSE  TRUE  TRUE
```

```
plot_varieties <- c("ICARDA_Gold", "Local_Check", "ICARDA_RustResist", "ICARDA_Gold")
plot_varieties == "ICARDA_Gold" # Which plots are ICARDA_Gold? [TRUE FALSE FALSE TRUE]
```

```
[1]  TRUE FALSE FALSE  TRUE
```

```
# Combining conditions
# Find plots where yield > 5.0 AND variety is ICARDA_Gold
(plot_yields > 5.0) & (plot_varieties == "ICARDA_Gold") # [TRUE FALSE FALSE TRUE]
```

```
[1]  TRUE FALSE FALSE  TRUE
```

```
# Find plots where yield > 6.0 OR variety is Local_Check
(plot_yields > 6.0) | (plot_varieties == "Local_Check") # [FALSE TRUE TRUE FALSE]
```

```
[1] FALSE  TRUE  TRUE FALSE
```

8.3 Vectorization: R's Superpower

Many R operations are **vectorized**, meaning they automatically apply to each element of a vector without needing you to write a loop. This makes R code concise and efficient. We've already seen this with arithmetic (`plot_yields + 0.5`) and comparisons (`plot_yields > 5.0`).

Functions like `mean()`, `sum()`, `min()`, `max()`, `sd()` (standard deviation), `length()` also work naturally on vectors:

```
plot_yields <- c(5.2, 4.5, 6.1, 5.5)

mean(plot_yields)
```

```
[1] 5.325
```

```
sd(plot_yields)
```

```
[1] 0.6652067
```

```
sum(plot_yields > 5.0) # How many plots yielded > 5.0? (TRUE=1, FALSE=0)
```

```
[1] 3
```

```
length(plot_yields) # How many plots?
```

```
[1] 4
```

8.4 Working with Data Frames (Indexing and Filtering)

This is crucial for selecting specific data from your tables.

Let's use the `trial_data` data frame from the previous section:

```
trial_data <- data.frame(
  PlotID = c("A101", "A102", "B101", "B102"),
  Variety = factor(c("ICARDA_Gold", "Local_Check", "ICARDA_RustResist", "ICARDA_Gold")),
  Yield_kg_plot = c(5.2, 4.5, 6.1, 5.5),
  Is_Resistant = c(TRUE, FALSE, TRUE, TRUE)
)
```

1. **Accessing Columns:** Use `$` (most common) or `[[ ]]`. `trial_data$Variety` or `trial_data[["Variety"]]`
2. **Accessing Rows/Columns/Cells using `[row, column]`:**

```
# Get the value in Row 2, Column 3
trial_data[2, 3] # Should be 4.5
```

```
[1] 4.5
```

```
# Get the entire Row 1 (returns a data frame)
trial_data[1, ]
```

```
  PlotID     Variety Yield_kg_plot Is_Resistant
1   A101 ICARDA_Gold           5.2         TRUE
```

```
# Get the entire Column 2 (Variety column, returns a vector/factor)
trial_data[, 2]
```

```
[1] ICARDA_Gold       Local_Check       ICARDA_RustResist ICARDA_Gold
Levels: ICARDA_Gold ICARDA_RustResist Local_Check
```

```
# Get Columns 1 and 3 (PlotID and Yield)
trial_data[, c(1, 3)] # Use c() for multiple column indices
```

```
  PlotID Yield_kg_plot
1   A101           5.2
2   A102           4.5
3   B101           6.1
4   B102           5.5
```

```
trial_data[, c("PlotID", "Yield_kg_plot")] # Can also use column names
```

```
  PlotID Yield_kg_plot
1   A101           5.2
2   A102           4.5
3   B101           6.1
4   B102           5.5
```

3. **Filtering Rows Based on Conditions (VERY IMPORTANT):** Use a logical condition inside the `row` part of the square brackets.

```
# Select rows where Yield_kg_plot is greater than 5.0
high_yield_plots <- trial_data[trial_data$Yield_kg_plot > 5.0, ]
print(high_yield_plots)
```

```
  PlotID           Variety Yield_kg_plot Is_Resistant
1   A101       ICARDA_Gold           5.2         TRUE
3   B101 ICARDA_RustResist           6.1         TRUE
4   B102       ICARDA_Gold           5.5         TRUE
```

```
# Select rows where Variety is "ICARDA_Gold"
icarda_gold_plots <- trial_data[trial_data$Variety == "ICARDA_Gold", ]
print(icarda_gold_plots)
```

```
  PlotID     Variety Yield_kg_plot Is_Resistant
1   A101 ICARDA_Gold           5.2         TRUE
4   B102 ICARDA_Gold           5.5         TRUE
```

```
# Select rows where Variety is "ICARDA_Gold" AND yield > 5.0
# (We generated the logical vector for this earlier)
condition <- (trial_data$Variety == "ICARDA_Gold") & (trial_data$Yield_kg_plot > 5.0)
print(condition) # Shows [TRUE FALSE FALSE TRUE]
```

```
[1]  TRUE FALSE FALSE  TRUE
```

```r
selected_plots <- trial_data[condition, ]
print(selected_plots)
```

```
  PlotID     Variety Yield_kg_plot Is_Resistant
1   A101 ICARDA_Gold           5.2         TRUE
4   B102 ICARDA_Gold           5.5         TRUE
```

```r
# Select rows where the variety is resistant
resistant_plots <- trial_data[trial_data$Is_Resistant == TRUE, ] # Or just trial_data[trial_
print(resistant_plots)
```

```
  PlotID           Variety Yield_kg_plot Is_Resistant
1   A101       ICARDA_Gold           5.2         TRUE
3   B101 ICARDA_RustResist           6.1         TRUE
4   B102       ICARDA_Gold           5.5         TRUE
```

8.5 apply() collection

The apply() family of functions lets us apply a function to the rows or columns in a matrix or data frame, a list or a vector.

```r
# Imagine we have a data matrix of plot yield values for different varieties.
# Each row represents a variety and each column a yield measurement for each trial
plot_trials <- matrix(c(5.2, 4.5, 6.1, 5.5, 4, 6.6, 7, 5.1, 5.3), nrow = 3, ncol = 3)

# We calculate the mean for each variety (each row), 1 means function is run on rows, 2 would
apply(plot_trials, 1, mean)
```

```
[1] 5.900000 4.533333 6.000000
```

Exercise: Select the data for the 'Local_Check' variety from the `trial_data` data frame. Calculate its yield.

9 Module 1.4: Reading and Writing Data

So far, we've created data inside R. But usually, your breeding data exists in external files, like Excel spreadsheets or CSV files. We need to get this data *into* R and save our results *out* of R.

9.1 Common Data File Formats

- **CSV (Comma Separated Values - `.csv`):** Plain text file where columns are separated by commas. Very common, easily readable by many programs (including R and Excel). **Often the best format for sharing data.**
- **TSV (Tab Separated Values - `.tsv`):** Similar to CSV, but uses tabs to separate columns.
- **Excel Files (`.xls`, `.xlsx`):** Native Microsoft Excel format. Can contain multiple sheets, formatting, formulas. Requires specific R packages to read/write.
- **Text Files(`.txt`):** Additionally, data can be saved as a simple text file. This file type can support comma or tab separated values. You would simply need to specify your separator when reading the file.

9.2 Paths, Working Directory, and RStudio Projects (Best Practice!)

R needs to know *where* to find your files.

- **Working Directory:** The default folder location R looks in. You can see it with `getwd()` and set it with `setwd("path/to/folder")`, but **setting it manually is usually bad practice** because it makes your code non-portable.
- **Absolute Path:** The full path from the root of your computer (e.g., `"C:/Users/YourName/Documents/Bre`
  Avoid this! It breaks if you move folders or share your code.
- **Relative Path & RStudio Projects (RECOMMENDED):**
  1. Organize your work using an **RStudio Project**. Create one via `File -> New Project -> Existing Directory...` and select your main course folder (`course_project_baku`).

2. When you open the `.Rproj` file, RStudio automatically sets the working directory to that project folder.
3. Keep your data files *inside* the project folder, ideally in subdirectories like `data/raw` (original data) or `data/example` (cleaned data for examples).
4. Refer to files using **relative paths** starting from the project root, like `"data/example/phenotypes.csv"`. This makes your analysis reproducible and easy to share!

9.3 Reading Data into R

We'll use functions from the `readr` (for CSV/TSV) and `readxl` (for Excel) packages. Make sure they are installed (see Module 1.1).

```
# Load the necessary libraries
library(readr)
library(readxl)
library(dplyr) # for glimpse

# --- Reading a CSV file ---
# Assumes you have a file 'sample_phenotypes.csv' in the 'data/example' folder
# relative to your project root.
pheno_file_path <- "data/sample_phenotypes.csv"

# Check if file exists before trying to read (good habit)
if (file.exists(pheno_file_path)) {
  # Use read_csv from the readr package (generally preferred)
  phenotype_data <- read_csv(pheno_file_path)

  print("CSV data loaded successfully:")
  head(phenotype_data)   # Look at the first 6 rows
  glimpse(phenotype_data) # See column names and data types

} else {
  print(paste("Error: Phenotype file not found at", pheno_file_path))
  phenotype_data <- NULL # Set to NULL if file not found
}

# Note: Base R has read.csv() - it works but readr::read_csv() is often faster
# and handles data types more consistently (e.g., doesn't default strings to factors).

# --- Reading an Excel file ---
```

```
# Assumes you have 'sample_trial.xlsx' in 'data/example'
excel_file_path <- "data/sample_trial.xlsx" # You'll need to create this file

if (file.exists(excel_file_path)) {
  # See what sheets are in the workbook
  excel_sheets(excel_file_path)

  # Read data from a specific sheet (e.g., "YieldData")
  # yield_data_excel <- read_excel(excel_file_path, sheet = "YieldData")

  # Or read by sheet number (first sheet is 1)
  # yield_data_excel <- read_excel(excel_file_path, sheet = 1)

  # print("Excel data loaded:")
  # glimpse(yield_data_excel)

} else {
  print(paste("Warning: Example Excel file not found at", excel_file_path))
}
```

- **Always inspect your data after loading!** Use `head()`, `str()`, `glimpse()`, `summary()`. Did R read the column names correctly? Are the data types what you expected (numeric, character, etc.)?

9.4 Writing Data out of R

After cleaning data or performing analysis, you'll want to save results.

```
# Load the necessary libraries
library(readr)
library(readxl)
library(dplyr) # for glimpse

# --- Reading a CSV file ---
# Assumes you have a file 'sample_phenotypes.csv' in the 'data/example' folder
# relative to your project root.
pheno_file_path <- "data/sample_phenotypes.csv"

# Check if file exists before trying to read (good habit)
if (file.exists(pheno_file_path)) {
  # Use read_csv from the readr package (generally preferred)
```

```
  phenotype_data <- read_csv(pheno_file_path)

  print("CSV data loaded successfully:")
  head(phenotype_data)   # Look at the first 6 rows
  glimpse(phenotype_data) # See column names and data types

} else {
  print(paste("Error: Phenotype file not found at", pheno_file_path))
  phenotype_data <- NULL # Set to NULL if file not found
}

# Note: Base R has read.csv() - it works but readr::read_csv() is often faster
# and handles data types more consistently (e.g., doesn't default strings to factors).

# --- Reading an Excel file ---
# Assumes you have 'sample_trial.xlsx' in 'data/example'
excel_file_path <- "data/sample_trial.xlsx" # You'll need to create this file

if (file.exists(excel_file_path)) {
  # See what sheets are in the workbook
  excel_sheets(excel_file_path)

  # Read data from a specific sheet (e.g., "YieldData")
  # yield_data_excel <- read_excel(excel_file_path, sheet = "YieldData")

  # Or read by sheet number (first sheet is 1)
  # yield_data_excel <- read_excel(excel_file_path, sheet = 1)

  # print("Excel data loaded:")
  # glimpse(yield_data_excel)

} else {
  print(paste("Warning: Example Excel file not found at", excel_file_path))
}
```

Exercise: If you have a simple Excel file with some breeding data (e.g., Plot ID, Variety, Yield), try reading it into R using `read_excel()`. Inspect the loaded data frame using `glimpse()`.

10 Why Visualize Your Data?

"A picture is worth a thousand words" - this is especially true for data! Plots help us to:

- **Explore:** Understand the distribution of your traits (e.g., `Yield`, `Height`). See relationships between variables. Identify patterns.
- **Diagnose:** Spot potential problems like outliers (strange values) or unexpected groupings. Check assumptions of statistical models.
- **Communicate:** Clearly present your findings to colleagues, managers, or in publications.

10.1 Introducing `ggplot2`: The Grammar of Graphics

R has basic plotting functions, but we will focus on the **`ggplot2`** package, which is part of the `tidyverse`. It's extremely powerful and flexible for creating beautiful, publication-quality graphics.

`ggplot2` is based on the **Grammar of Graphics**. The idea is to build plots layer by layer:

1. **`ggplot()` function:** Start the plot. You provide:
    - `data`: The data frame containing your variables.
    - `mapping = aes(...)`: **Aesthetic mappings**. This tells `ggplot` *how variables in your data map to visual properties* of the plot (e.g., map `Yield` to the y-axis, `Height` to the x-axis, `Variety` to color).
2. **`geom_` functions:** Add geometric layers to actually *display* the data. Examples:
    - `geom_point()`: Creates a scatter plot.
    - `geom_histogram()`: Creates a histogram.
    - `geom_boxplot()`: Creates box-and-whisker plots.
    - `geom_line()`: Creates lines.
    - `geom_bar()`: Creates bar charts.
3. **Other functions:** Add labels (`labs()`), change themes (`theme_bw()`, `theme_minimal()`), split plots into facets (`facet_wrap()`), customize scales, etc. Each function also allows you to edit aesthetic characteristics such as size, color, etc.

10.2 Let's Make Some Plots!

First, load the necessary libraries:

```
# Load necessary libraries
library(ggplot2)
library(dplyr) # Often used with ggplot2 for data prep
```

Now, let's create a sample breeding data frame for plotting.

```
set.seed(123) # for reproducible random numbers
breeding_plot_data <-
  tibble(
    PlotID = paste0("P", 101:120),
    Variety = factor(rep(c("ICARDA_A", "ICARDA_B", "Check_1", "Check_2"), each = 5)),
    Location = factor(rep(c("Baku", "Ganja"), each = 10)),
    Yield = rnorm(20, mean = rep(c(6, 7, 5, 5.5), each = 5), sd = 0.8),
    Height = rnorm(20, mean = rep(c(90, 110, 85, 88), each = 5), sd = 5)
  )

# Take a quick look at the data structure
glimpse(breeding_plot_data)
```

```
Rows: 20
Columns: 5
$ PlotID   <chr> "P101", "P102", "P103", "P104", "P105", "P106", "P107", "P108~
$ Variety  <fct> ICARDA_A, ICARDA_A, ICARDA_A, ICARDA_A, ICARDA_A, ICARDA_B, I~
$ Location <fct> Baku, Baku, Baku, Baku, Baku, Baku, Baku, Baku, Baku, Baku, G~
$ Yield    <dbl> 5.551619, 5.815858, 7.246967, 6.056407, 6.103430, 8.372052, 7~
$ Height   <dbl> 84.66088, 88.91013, 84.86998, 86.35554, 86.87480, 101.56653, ~
```

10.2.1 1. Scatter Plot: Relationship between Yield and Height

See if taller plants tend to have higher yield in this dataset.

```
# 1. ggplot(): data is breeding_plot_data, map Height to x, Yield to y
# 2. geom_point(): Add points layer
# 3. labs() and theme_bw(): Add labels and theme
plot1 <-
  ggplot(data = breeding_plot_data, mapping = aes(x = Height, y = Yield)) +
```

```
  geom_point() +
  labs(
    title = "Relationship between Plant Height and Yield",
    x = "Plant Height (cm)",
    y = "Yield (kg/plot)",
    caption = "Sample Data"
  ) +
  theme_bw() # Use a clean black and white theme

# Display the plot
plot1
```

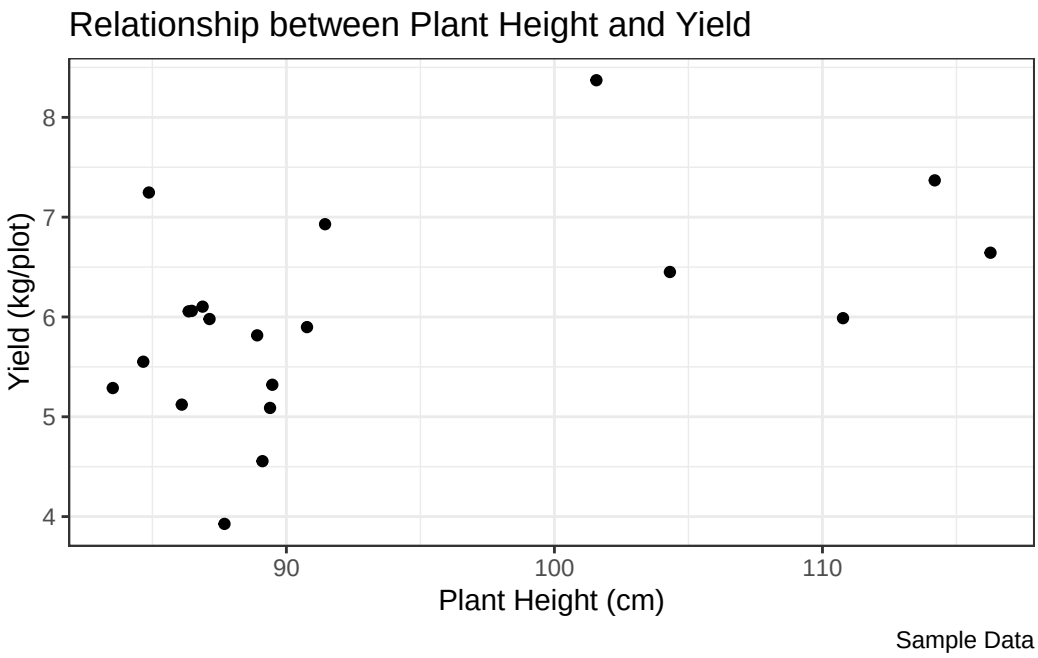


Figure 10.1: Relationship between Plant Height and Yield.

Let's color the points by Variety:

```
# Map 'color' aesthetic to the Variety column
# Adjust point size and transparency for better visibility
plot2 <-
  ggplot(data = breeding_plot_data, mapping = aes(x = Height, y = Yield, color = Variety)) +
  geom_point(size = 2.5, alpha = 0.8) + # Make points slightly bigger, semi-transparent
  labs(
```

```
    title = "Height vs. Yield by Variety",
    x = "Plant Height (cm)",
    y = "Yield (kg/plot)"
  ) +
  theme_minimal() # Use a different theme

# Display the plot
plot2
```

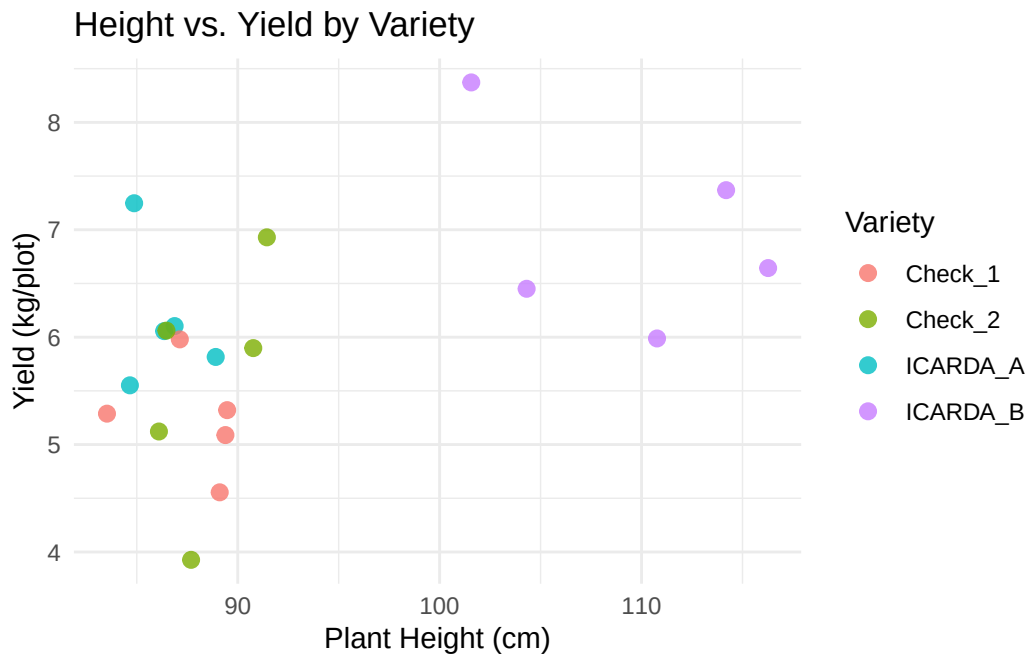


Figure 10.2: Height vs. Yield by Variety, colored by Variety.

10.2.2 2. Histogram: Distribution of Yield

See the frequency of different yield values.

```
# 1. ggplot(): data, map Yield to x-axis
# 2. geom_histogram(): Add histogram layer. Adjust 'binwidth' or 'bins'.
# 3. labs() and theme_classic(): Add labels and theme
plot3 <-
  ggplot(data = breeding_plot_data, mapping = aes(x = Yield)) +
  geom_histogram(binwidth = 0.5, fill = "lightblue", color = "black") + # Specify binwidth, 
  labs(
```

```
    title = "Distribution of Plot Yields",
    x = "Yield (kg/plot)",
    y = "Frequency (Number of Plots)"
  ) +
  theme_classic()

# Display the plot
plot3
```

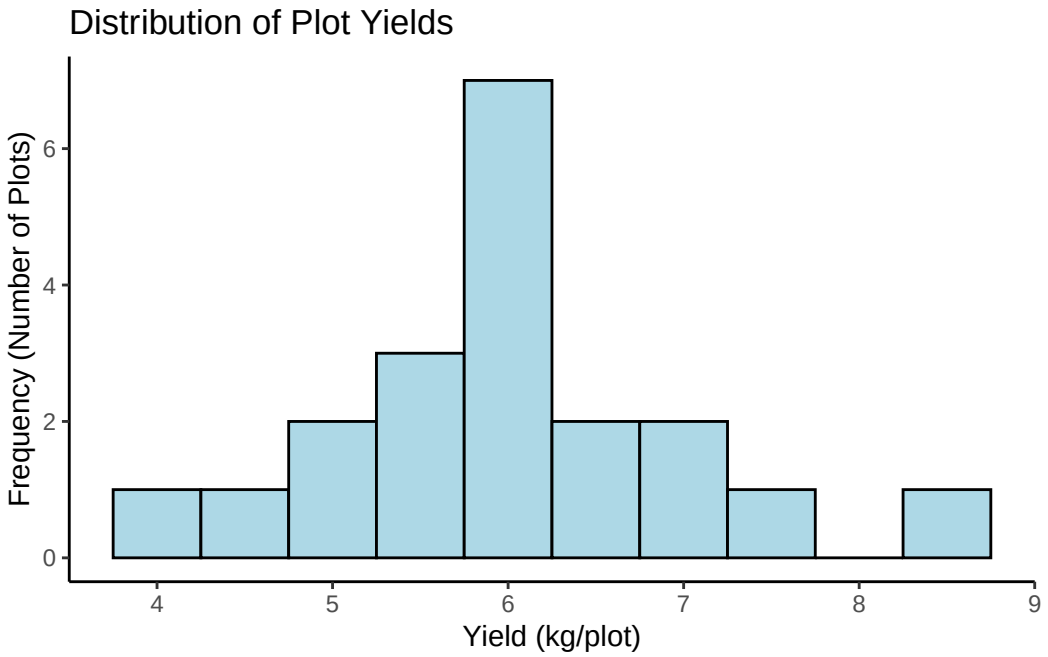


Figure 10.3: Distribution of Plot Yields.

10.2.3 3. Box Plot: Compare Yield across Locations

Are yields different in Baku vs. Ganja? Box plots are great for comparing distributions across groups.

```
# 1. ggplot(): data, map Location (categorical) to x, Yield (numeric) to y
# 2. geom_boxplot(): Add boxplot layer. Map 'fill' to Location for color.
# 3. labs() and theme_light(): Add labels and theme
# 4. theme(): Customize theme elements (e.g., remove legend)
plot4 <-
  ggplot(data = breeding_plot_data, mapping = aes(x = Location, y = Yield, fill = Location))
```

```
  geom_boxplot() +
  labs(
    title = "Yield Comparison by Location",
    x = "Location",
    y = "Yield (kg/plot)"
  ) +
  theme_light() +
  theme(legend.position = "none") # Hide legend if coloring is obvious from x-axis

# Display the plot
plot4
```

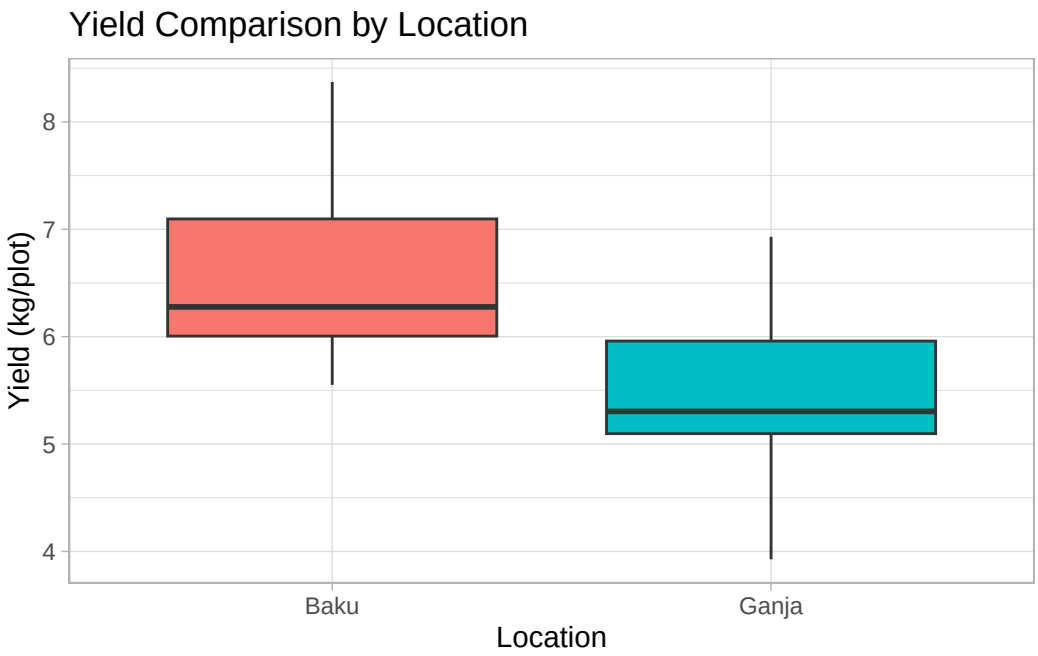


Figure 10.4: Yield Comparison by Location.

Box plot anatomy: The box shows the interquartile range (IQR, middle 50% of data), the line inside is the median, whiskers extend typically 1.5*IQR, points beyond are potential outliers.

10.3 Saving Your Plots

Use the `ggsave()` function after you've created a ggplot object (like plot1, plot2, etc.).

```r
# Make sure the 'output/figures' directory exists
# The 'recursive = TRUE' creates parent directories if needed
output_dir <- "output/figures"
if (!dir.exists(output_dir)) {
  dir.create(output_dir, recursive = TRUE)
}

# Save the height vs yield scatter plot (plot2)
ggsave(
  filename = file.path(output_dir, "height_yield_scatter.png"), # Use file.path for robust pa
  plot = plot2,                                 # The plot object to save
  width = 7,                                    # Width in inches
  height = 5,                                   # Height in inches
  dpi = 300                                     # Resolution (dots per inch)
)

# You can save in other formats too, like PDF:
# ggsave(
#   filename = file.path(output_dir, "yield_distribution.pdf"),
#   plot = plot3,
#   width = 6,
#   height = 4
# )
```

10.4 Exercise

Create a box plot comparing Plant Height (`Height`) across the different Varieties (`Variety`) in the `breeding_plot_data`. Save the plot as a PNG file named `height_variety_boxplot.png` in the `output/figures` directory.

```r
# Exercise: Box plot comparing Plant Height across Varieties
plot5 <-
  ggplot(data = breeding_plot_data, mapping = aes(x = Variety, y = Height, fill = Variety)) +
  geom_boxplot() +
  labs(
    title = "Plant Height Comparison by Variety",
    x = "Variety",
    y = "Plant Height (cm)"
   ) +
  theme_light() +
  theme(legend.position = "none")
```

```
# Display the new plot
plot5
```

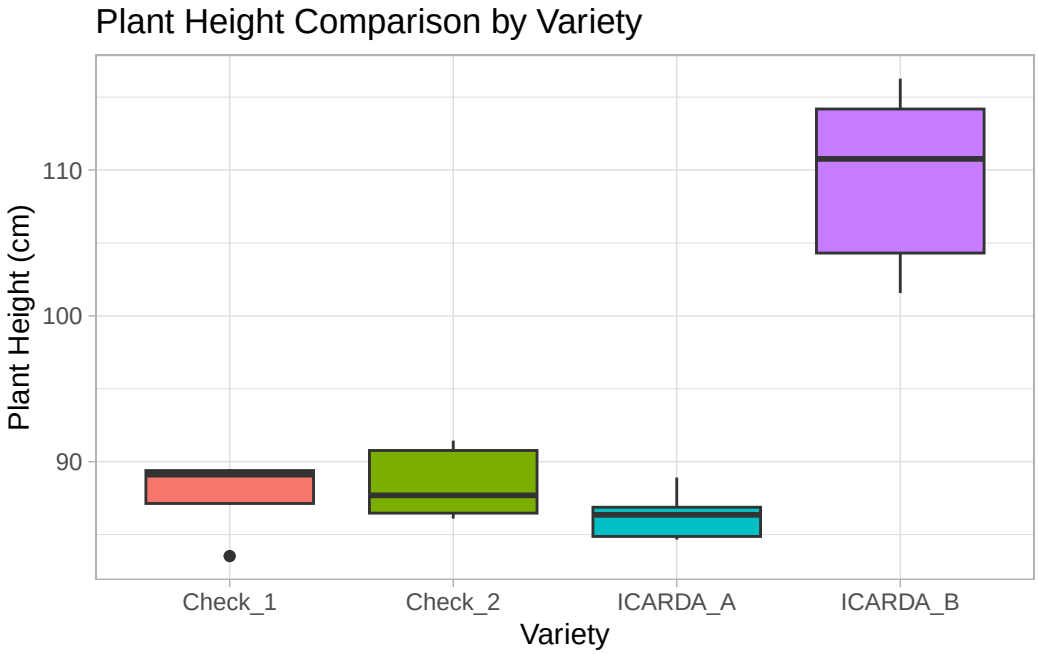


Figure 10.5: Plant Height Comparison by Variety.

```
# Ensure output directory exists
output_dir <- "output/figures"
if (!dir.exists(output_dir)) {
  dir.create(output_dir, recursive = TRUE)
}

# Save the box plot as a PNG file
ggsave(
  filename = file.path(output_dir, "height_variety_boxplot.png"),
  plot = plot5,
  width = 7,
  height = 5,
  dpi = 300
)
```

Part III

Handling Breeding Data

11 Module 2.1: Loading Breeding Data - ICARDA Barley Example

11.1 Introduction to the Dataset

In this module, we'll learn how to load typical phenotypic data into R. We'll use a real-world example: data from a study on **275 barley accessions conducted at ICARDA in 2019**. This dataset contains various measurements related to agronomic traits, grain quality, and morphological characteristics.

Why this dataset? * It's representative of the kind of multi-trait data breeders work with. * It allows us to practice loading, inspecting, and performing basic summaries on realistic data. * This data comes from ICARDA's valuable work in crop improvement for dry areas.

Column Descriptions (Partial List - full list would be in a data dictionary): * `Taxa`: The identifier for each barley accession (genotype). * `Area`: Grain area (e.g., mm^2). * `B_glucan`: Beta-glucan content (%), a quality trait. * `DTH`: Days to Heading (days), an agronomic trait. * `Fe`: Iron content in grain (ppm), a nutritional trait. * `FLA`: Flag Leaf Area (cm^2). * `GY`: Grain Yield (e.g., t/ha or kg/plot - units should always be known!). * `PH`: Plant Height (cm). * `Protein`: Grain protein content (%). * `TKW`: Thousand Kernel Weight (grams). * `Zn`: Zinc content in grain (ppm). * *(And many others related to grain morphology and plant characteristics…)*

Our goal is to load this data (which is typically stored in a file like a CSV or Excel sheet) into an R data frame so we can start analyzing it.

11.2 Setting Up: Libraries and File Path

First, we need to load the R packages that help us read data. Even though we have previously installed and loaded all packages we will need, in case you are only focusing on reading data, the `readr` package (part of `tidyverse`) is excellent for reading text files like CSVs.

Remember our RStudio Project setup! We will assume the data file is saved in the `data/` subfolder of our project.

```r
# Load the necessary libraries
# 'tidyverse' includes 'readr' (for read_csv) and 'dplyr' (for glimpse, etc.)
library(tidyverse)
```

```
-- Attaching core tidyverse packages ------------------------ tidyverse 2.0.0 --
v dplyr     1.1.4     v readr     2.1.5
v forcats   1.0.0     v stringr   1.5.1
v ggplot2   3.5.2     v tibble    3.2.1
v lubridate 1.9.4     v tidyr     1.3.1
v purrr     1.0.4
-- Conflicts ------------------------------------------ tidyverse_conflicts() --
x dplyr::filter() masks stats::filter()
x dplyr::lag()    masks stats::lag()
i Use the conflicted package (<http://conflicted.r-lib.org/>) to force all conflicts to becor
```

11.3 Reading the CSV File

Let's say our barley data is stored in a CSV file named `icarda_barley_2019_pheno.csv`.

```r
# Define the path to our data file (relative to the project root)
barley_data_file_path <- "data/icarda_barley_2019_pheno.csv"

# Check if the file exists (good practice!)
if (file.exists(barley_data_file_path)) {
  # Use read_csv() from the readr package to load the data
  barley_pheno_data <- read_csv(barley_data_file_path)

  print("ICARDA Barley Phenotype data loaded successfully!")
} else {
  print(paste("ERROR: File not found at:", barley_data_file_path))
  print("Please make sure 'icarda_barley_2019_pheno.csv' is in the 'data/example' folder.")
  # If the file isn't found, we'll create an empty placeholder to avoid later errors in the c
  barley_pheno_data <- tibble() # Creates an empty tibble (tidyverse data frame)
}
```

11.4 First Look: Inspecting the Loaded Data

It's **CRUCIAL** to always inspect your data immediately after loading it to make sure it looks correct.

1. **`head()`**: Shows the first few rows (default is 6).
2. **`dim()`**: Shows the dimensions (number of rows, number of columns).
3. **`glimpse()`** (from `dplyr`): A great way to see column names, their data types, and the first few values. Better than `str()` for tibbles.
4. **`summary()`**: Provides basic summary statistics for each column (Min, Max, Mean, Median, Quartiles for numeric; counts for character/factor).

11.5 Understanding Data Types in Our Barley Data

When `glimpse()` runs, you'll see types like: * `Taxa`: Should be `<chr>` (character) as it's an identifier. * `Area`, `B_glucan`, `DTH`, `GY`, `PH`, etc.: Should mostly be `<dbl>` (double-precision numeric) as they are measurements.

If `read_csv` misinterprets a numeric column as character (e.g., if there's a text entry like "missing" in a numeric column), you'll need to clean that data or specify column types during import using the `col_types` argument in `read_csv()`. (We'll cover data cleaning later).

11.6 Quick Summary of a Specific Trait

Let's say we are interested in Grain Yield (`GY`).

```
# Make sure the data and the 'GY' column exist
if (nrow(barley_pheno_data) > 0 && "GY" %in% names(barley_pheno_data)) {
  # Access the GY column
  yield_values <- barley_pheno_data$GY

  # Calculate some basic statistics
  mean_yield <- mean(yield_values, na.rm = TRUE) # na.rm=TRUE ignores missing values in calcu
  min_yield <- min(yield_values, na.rm = TRUE)
  max_yield <- max(yield_values, na.rm = TRUE)
  sd_yield <- sd(yield_values, na.rm = TRUE)

  print(paste("Average Grain Yield (GY):", round(mean_yield, 2)))
  print(paste("Minimum Grain Yield (GY):", round(min_yield, 2)))
  print(paste("Maximum Grain Yield (GY):", round(max_yield, 2)))
  print(paste("Standard Deviation of GY:", round(sd_yield, 2)))

  # How many accessions do we have yield data for (non-missing)?
  num_yield_obs <- sum(!is.na(yield_values))
```

```
  print(paste("Number of accessions with GY data:", num_yield_obs))
} else if (nrow(barley_pheno_data) > 0) {
  print("Column 'GY' not found in the loaded data.")
}
```

```
[1] "Average Grain Yield (GY): 1.19"
[1] "Minimum Grain Yield (GY): 0.5"
[1] "Maximum Grain Yield (GY): 2.47"
[1] "Standard Deviation of GY: 0.31"
[1] "Number of accessions with GY data: 275"
```

Exercise: 1. Load the `icarda_barley_2019_pheno.csv` file into R. 2. Use `glimpse()` to check the column names and data types. 3. Calculate and print the average Plant Height (`PH`) from the dataset. Remember to handle potential missing values (`na.rm = TRUE`).

This module has shown you the first critical step: getting your valuable field data into R. In the next modules, we'll learn how to clean, manipulate, and visualize this data.

```
# Exercise
# Inspect data and data types
glimpse(barley_pheno_data)
```

```
Rows: 275
Columns: 24
$ Taxa        <chr> "G1", "G2", "G3", "G4", "G5", "G7", "G8", "G9", "G12", "G1~
$ Area        <dbl> 22.72177, 23.07877, 19.93612, 23.32083, 19.54859, 22.96829~
$ B_glucan    <dbl> 6.848584, 7.430943, 4.012621, 6.091926, 7.307811, 7.267738~
$ Circularity <dbl> 1.887915, 1.780070, 1.555492, 2.300471, 1.896487, 1.752493~
$ Diameter    <dbl> 5.366522, 5.403555, 5.034355, 5.420909, 4.981714, 5.422541~
$ DTH         <dbl> 75.89584, 70.17532, 74.19461, 74.39462, 77.66037, 72.65420~
$ Fe          <dbl> 29.67685, 31.59088, 34.15825, 31.44544, 30.31127, 30.61605~
$ FLA         <dbl> 16.586760, 7.966124, 7.592350, 18.753322, 16.532845, 18.40~
$ FLH         <dbl> 82.77116, 62.36145, 61.81653, 69.77661, 64.38496, 78.08801~
$ GpS         <dbl> 44.70886, 25.25222, 25.72167, 66.23116, 54.63221, 26.63211~
$ GWS         <dbl> 1.6156277, 1.0563208, 0.9516462, 2.5202010, 1.2044795, 1.0~
$ GY          <dbl> 1.3747605, 1.3735596, 0.9054266, 0.7614383, 0.8827177, 2.0~
$ HW          <dbl> 57.65487, 64.41055, 69.41048, 55.35590, 53.65940, 64.98411~
$ Length      <dbl> 10.542981, 10.106793, 8.565790, 11.920853, 9.781558, 9.968~
$ Length_Wid  <dbl> 3.313525, 3.013035, 2.524445, 3.833381, 3.305421, 2.978634~
$ PdH         <dbl> 91.09761, 63.65600, 68.75482, 69.50663, 64.10002, 80.59519~
$ PdL         <dbl> 8.1320940, 1.0341930, 6.5555050, 0.9639243, -0.6416490, 2.~
```

```
$ Perimeter   <dbl> 29.07010, 28.33654, 24.57919, 32.21194, 27.02846, 28.03643~
$ PH          <dbl> 96.76140, 70.56720, 77.16486, 79.22446, 71.78375, 88.70248~
$ Protein     <dbl> 14.06372, 14.16090, 15.23294, 14.34877, 14.46224, 14.31356~
$ SL          <dbl> 5.343668, 6.804813, 8.074976, 10.039594, 7.526454, 8.31457~
$ TKW         <dbl> 28.48964, 35.00164, 31.95561, 27.27054, 24.64983, 33.83401~
$ width       <dbl> 3.251696, 3.439623, 3.455571, 3.204069, 3.000305, 3.429043~
$ Zn          <dbl> 31.21179, 34.21217, 25.50724, 32.24918, 33.58773, 33.92710~
```

```r
# Calculating average Plant Height
plant_heights <- barley_pheno_data$PH # extracting plant height column
mean_height <- mean(plant_heights, na.rm = TRUE)

# Printing average Plant Height
print(paste("Average Plant Height (PH):", round(mean_height, 2)))
```

```
[1] "Average Plant Height (PH): 74.63"
```

12 Module 2.2: Data Quality Control and Filtering

12.1 Quality Control

In this module we will briefly describe the idea of data quality control, a common practice that allows us to identify errors or anomalies within our data before performing any posterior analyses.

12.2 Filtering Duplicates

In plant breeding we will typically have multi-trait or gene data collected by accession. In a data frame, we may have an ID or Taxa column, followed by columns with the trait or gene information. Before performing any type of analyses, it is important to identify and filter duplicates if there are any, as these could skew our results. For this example we will work with a generated file based on the data set used for Module 9.1. In this case, * `Taxa` is our genotype identifier column.

```
# Load the necessary libraries
# 'tidyverse' includes 'readr' (for read_csv) and 'dplyr' (for glimpse, etc.)
library(tidyverse)
```

```
-- Attaching core tidyverse packages ------------------------ tidyverse 2.0.0 --
v dplyr     1.1.4     v readr     2.1.5
v forcats   1.0.0     v stringr   1.5.1
v ggplot2   3.5.2     v tibble    3.2.1
v lubridate 1.9.4     v tidyr     1.3.1
v purrr     1.0.4
-- Conflicts ------------------------------------------ tidyverse_conflicts() --
x dplyr::filter() masks stats::filter()
x dplyr::lag()    masks stats::lag()
i Use the conflicted package (<http://conflicted.r-lib.org/>) to force all conflicts to beco
```

```
# We start by loading our raw data file
barley_data_file_path <- "data/icarda_barley_2019_pheno_raw.csv" # Defining our file path
barley_pheno_data <- read_csv(barley_data_file_path) # Loading into data frame
```

```
# We filter the data frame to keep only one entry for each ID
data <- barley_pheno_data[!duplicated(barley_pheno_data),]
data
```

```
# A tibble: 275 x 24
   Taxa   Area B_glucan Circularity Diameter   DTH    Fe   FLA   FLH   GpS   GWS
   <chr> <dbl>    <dbl>       <dbl>    <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
 1 G1     22.7     6.85        1.89     5.37  75.9  29.7 16.6   82.8  44.7 1.62
 2 G2     23.1     7.43        1.78     5.40  70.2  31.6  7.97  62.4  25.3 1.06
 3 G3     19.9     4.01        1.56     5.03  74.2  34.2  7.59  61.8  25.7 0.952
 4 G4     23.3     6.09        2.30     5.42  74.4  31.4 18.8   69.8  66.2 2.52
 5 G5     19.5     7.31        1.90     4.98  77.7  30.3 16.5   64.4  54.6 1.20
 6 G7     23.0     7.27        1.75     5.42  72.7  30.6 18.4   78.1  26.6 1.08
 7 G8     24.2     6.52        1.90     5.51  73.1  34.5 13.2   66.8  51.4 2.46
 8 G9     22.5     6.61        1.90     5.34  74.0  33.5 12.7   70.0  38.2 1.39
 9 G12    21.5     7.12        1.62     5.23  77.2  32.3 11.6   61.3  31.3 1.20
10 G13    19.3     7.07        1.88     4.98  76.6  31.7 17.3   62.8  43.5 1.31
# i 265 more rows
# i 13 more variables: GY <dbl>, HW <dbl>, Length <dbl>, Length_Wid <dbl>,
#   PdH <dbl>, PdL <dbl>, Perimeter <dbl>, PH <dbl>, Protein <dbl>, SL <dbl>,
#   TKW <dbl>, width <dbl>, Zn <dbl>
```

12.3 Filtering by Missing Data

Many times, we will want to remove rows with missing data. Missing values can distort certain calculations. Although some functions in R have an option to automatically remove NA's before performing the calculation, many times it is necessary to remove them beforehand.

```
# We will work with our duplicate filtered data frame from last section.
# complete.cases() allows us to filter any rows with missing values
data <- data[complete.cases(data),]
data
```

```
# A tibble: 266 x 24
   Taxa   Area B_glucan Circularity Diameter   DTH    Fe   FLA   FLH   GpS   GWS
   <chr> <dbl>    <dbl>       <dbl>    <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
```

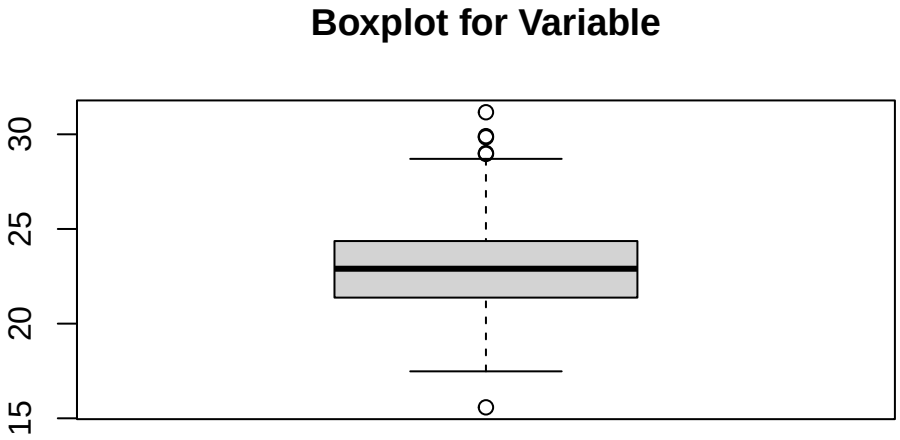
```
 1 G1     22.7     6.85        1.89     5.37  75.9  29.7 16.6   82.8  44.7 1.62
 2 G2     23.1     7.43        1.78     5.40  70.2  31.6  7.97  62.4  25.3 1.06
 3 G3     19.9     4.01        1.56     5.03  74.2  34.2  7.59  61.8  25.7 0.952
 4 G4     23.3     6.09        2.30     5.42  74.4  31.4 18.8   69.8  66.2 2.52
 5 G5     19.5     7.31        1.90     4.98  77.7  30.3 16.5   64.4  54.6 1.20
 6 G7     23.0     7.27        1.75     5.42  72.7  30.6 18.4   78.1  26.6 1.08
 7 G8     24.2     6.52        1.90     5.51  73.1  34.5 13.2   66.8  51.4 2.46
 8 G9     22.5     6.61        1.90     5.34  74.0  33.5 12.7   70.0  38.2 1.39
 9 G12    21.5     7.12        1.62     5.23  77.2  32.3 11.6   61.3  31.3 1.20
10 G13    19.3     7.07        1.88     4.98  76.6  31.7 17.3   62.8  43.5 1.31
# i 256 more rows
# i 13 more variables: GY <dbl>, HW <dbl>, Length <dbl>, Length_Wid <dbl>,
#   PdH <dbl>, PdL <dbl>, Perimeter <dbl>, PH <dbl>, Protein <dbl>, SL <dbl>,
#   TKW <dbl>, width <dbl>, Zn <dbl>
```

12.4 Identifying and Filtering Outliers

Identifying and filtering outliers helps improve the accuracy and reliability of our results. There are many sources of outliers or anomalies, human error, measurement errors, or simply nature doing its thing. However, even if we don't want to remove these data points, it is important to identify them first to then decide what the following steps will be. Outliers can skew important statistical measures and correlations, leading to misleading conclusions. Moreover, many machine learning algorithms, such as regression or PCA, are very sensitive to outliers. For examples, outliers in PCA can influence the principal components and clusterization, leading to misclassification.

For the following example we will analyse the * `Taxa` in our filtered data set.

```
# We can use a simple boxplot to explore the overall structure of our data and identify outli
areaBP <- boxplot(data$Area, main = "Boxplot for Variable")
```




```
# Saving our boxplot in an object allows us to extract the outlier values as a vector
areaBP$out
```

```
[1] 31.16480 28.96363 29.90218 29.84108 29.00766 15.57887
```

```
# We can then choose to remove these outlier entries from our data frame
# The idea here is that we keep the rows where the `Area` value is not in our outliers vecto
dataArea <- data[! data$Area %in% areaBP$out,]
dataArea
```

```
# A tibble: 260 x 24
   Taxa   Area B_glucan Circularity Diameter   DTH    Fe   FLA   FLH   GpS   GWS
   <chr> <dbl>    <dbl>       <dbl>    <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
 1 G1     22.7     6.85        1.89     5.37  75.9  29.7 16.6   82.8  44.7 1.62
 2 G2     23.1     7.43        1.78     5.40  70.2  31.6  7.97  62.4  25.3 1.06
 3 G3     19.9     4.01        1.56     5.03  74.2  34.2  7.59  61.8  25.7 0.952
 4 G4     23.3     6.09        2.30     5.42  74.4  31.4 18.8   69.8  66.2 2.52
 5 G5     19.5     7.31        1.90     4.98  77.7  30.3 16.5   64.4  54.6 1.20
 6 G7     23.0     7.27        1.75     5.42  72.7  30.6 18.4   78.1  26.6 1.08
 7 G8     24.2     6.52        1.90     5.51  73.1  34.5 13.2   66.8  51.4 2.46
 8 G9     22.5     6.61        1.90     5.34  74.0  33.5 12.7   70.0  38.2 1.39
 9 G12    21.5     7.12        1.62     5.23  77.2  32.3 11.6   61.3  31.3 1.20
10 G13    19.3     7.07        1.88     4.98  76.6  31.7 17.3   62.8  43.5 1.31
# i 250 more rows
# i 13 more variables: GY <dbl>, HW <dbl>, Length <dbl>, Length_Wid <dbl>,
#   PdH <dbl>, PdL <dbl>, Perimeter <dbl>, PH <dbl>, Protein <dbl>, SL <dbl>,
#   TKW <dbl>, width <dbl>, Zn <dbl>
```

Part IV

Core Genomic Concepts

13 Module 3.1: Genotype Data

13.1 Introduction

Genotype data refers to the genetic makeup, in this case of crops, at specific loci across the genome. This data allows us to associate genetic differences with traits of agronomic interest and regional information.

- The **genotype** refers to the specific combination of alleles at a given location. Depending on the ploidy of the crop, we will have two (diploids) or more alleles per locus.
- **SNPs (Single Nucleotide Polymorphisms)** are positions across the genome where variations exist between individuals.
  - *Example:* Three different crop variants may have homozygous A/A, heterozygous A/G and homozygous G/G at a specific locus. This can also be coded as 0, 1 and 2. 0 represents homozygous for the reference allele, 1 represents heterozygous, and 2 represents homozygous for the alternate allele.

13.2 Formats

SNP data can be stored in different formats and file types, depending on the platform or program used. We will briefly discuss the most common file types.

- **VCF (Variant Call Format - `.vcf`):** Standard format for SNPs and variants from sequencing. Contains metadata, IDs, calls, positions and other information.
  - GT = Genotype
  - 0 = REF; 1 = ALT
  - 0/0 or 1/1= homozygous; 0/1 or 1/0 = heterozygous

```
#CHROM  POS     ID            REF   ALT QUAL    FILTER  INFO    FORMAT  G1  G2  G3
1H      7253074 SCRI_RS_1929    A     C   .     PASS        .     GT        1/1 1/1 0/0
```

```
# Read vcf file
vcf <- read.vcfR("data/Barley.vcf", verbose = FALSE)
```

```r
# Glimpse vcf
head(vcf)
```

```
[1] "***** Object of class 'vcfR' *****"
[1] "***** Meta section *****"
[1] "##fileformat=VCFv4.1"
[1] "##FILTER=<ID=PASS,Description=\"All filters passed\">"
[1] "##FORMAT=<ID=DP,Number=1,Type=Integer,Description=\"Approximate read  [Truncated]"
[1] "##FORMAT=<ID=DV,Number=.,Type=Integer,Description=\"Read depth of the [Truncated]"
[1] "##FORMAT=<ID=GT,Number=1,Type=String,Description=\"Genotype\">"
[1] "##INFO=<ID=MQ,Number=1,Type=Float,Description=\"RMS Mapping Quality\">"
[1] "First 6 rows."
[1]
[1] "***** Fixed section *****"
     CHROM POS      ID REF ALT QUAL  FILTER
[1,] "1H"  "144018" NA "A" "G" "999" "NA"
[2,] "1H"  "147155" NA "T" "C" "999" "NA"
[3,] "1H"  "166336" NA "C" "T" "999" "NA"
[4,] "1H"  "173286" NA "T" "C" "999" "NA"
[5,] "1H"  "253434" NA "C" "T" "999" "NA"
[6,] "1H"  "253481" NA "C" "T" "999" "NA"
[1]
[1] "***** Genotype section *****"
     FORMAT     ICARDA_G0011 ICARDA_G0012 ICARDA_G0013 ICARDA_G0014
[1,] "GT:DP:DV" "0/0:33:0"   "0/0:23:0"   "0/0:18:0"   "0/0:23:0"
[2,] "GT:DP:DV" "0/0:8:0"    "0/0:8:0"    "0/0:10:0"   "0/0:6:0"
[3,] "GT:DP:DV" "0/0:16:0"   "0/0:9:0"    "0/0:9:0"    "0/0:5:0"
[4,] "GT:DP:DV" "0/0:18:0"   "0/0:18:0"   "0/0:14:0"   "0/0:10:0"
[5,] "GT:DP:DV" "1/1:12:12"  "1/1:19:19"  "1/1:12:11"  "1/1:10:10"
[6,] "GT:DP:DV" "0/0:12:0"   "0/0:17:0"   "0/0:12:0"   "0/0:10:0"
     ICARDA_G0015
[1,] "0/0:15:0"
[2,] "0/0:8:0"
[3,] "0/0:9:0"
[4,] "0/0:10:0"
[5,] "1/1:12:12"
[6,] "0/0:11:0"
[1] "First 6 columns only."
[1]
[1] "Unique GT formats:"
[1] "GT:DP:DV"
[1]
```

```
# Turn into matrix
vcfMatrix <- extract.gt(vcf)
```

- **PLINK (`-.ped`, `.map`) or Binary PLINK (`-.bed`, `.bim`, `.fam`)**
  - `.ped`: Pedigree/genotype data (tab delimited)
  - `.map`: SNP mapping information
  - `.bed`: Binary genotype matrix
  - `.bin`: SNP information
  - `.fam`: Sample information
- **HapMap (`-.hmp.txt`):** Used in TASSEL, header includes metadata, positions and genotypes encoded as allele pairs (A/A, A/G, etc.).
- **Numeric Matrix (-.csv, .txt):** SNPs in columns and genotypes in rows (or vice versa), data encoded as 0, 1 and 2 for homozygous for reference allele, heterozygous, and homozygous for alternate allele.

```
# Load SNP data matrix
matrix <- read.table("data/BarleyMatrix.txt", sep = "\t", header = TRUE, row.names = 1, chec
```

```
# The vcf matrix we obtained can also be turned into this type of format
matrixNum <- vcfToNumericMatrix(vcfMatrix)
```

14 Module 3.2: Using the QBMS Package to Query Genotypic Data

We have explored how to import genotypic data from files into R, but we can also retrieve it directly from online databases, such as **Gigwa**, using **ICARDA's QBMS package**.

```
# Loading QBMS library
library(QBMS)

# Configuring the connection
set_qbms_config(url = "https://gigwa.icarda.org:8443/gigwa/", engine = "gigwa")

login_gigwa("Tamara", "-Zy2gn5ijQcW!EE")
```

Once logged in, we can use the `gigwa_list_dbs()` function to view all available data bases.

```
gigwa_list_dbs()
```

```
 [1] "BarleySubData"              "Barley_Hvulgare2"
 [3] "Barley_Hvulgare3"           "Barley_Hvulgare4"
 [5] "Barley_Hvulgare5"           "Barley_MegaProject1"
 [7] "Barley_MegaProject1_public" "Cactus_Copuntia1"
 [9] "Chickpea_Carietinum1"       "Chickpea_Carietinum2"
[11] "Chickpea_Carietinum3"       "Chickpea_Carietinum4"
[13] "Faba_Vfaba1"                "GrassPea_Lsativus1"
[15] "GrassPea_Lsativus2"         "Musa_Macuminata1"
[17] "WheatDurum_Tdurum1"         "WheatDurum_Tdurum2"
[19] "WheatDurum_Tdurum3"         "WheatDurum_Tdurum4"
[21] "WheatDurum_Tdurum5"         "WheatDurum_Tdurum6"
[23] "WheatDurum_Tdurum7"         "WheatDurum_Tdurum8"
[25] "WheatWild_Tspp1"            "Wheat_Taestivum1"
[27] "Wheat_Taestivum2"           "Wheat_Taestivum3"
[29] "Wheat_Taestivum4"           "Wheat_Taestivum5"
```

For this example we will be choosing the *"BarleySubData"* database.

```
# To set a data base
gigwa_set_db("BarleySubData")
```

Once we have defined a data base, we have to define a project and a run. We can do this the following way.

```
# To view available projects
gigwa_list_projects()
```

```
      studyName
1 BarleySubData
```

```
# To set a project
gigwa_set_project("BarleySubData")

# To view available runs
gigwa_list_runs()
```

```
  variantSetName
1           Run1
```

```
# To set a run
gigwa_set_run("Run1")
```

Once we have defined a data base, a project and a run, there are many tools we can use to extract relevant information.

1. `gigwa_get_samples()`: Retrieves a list of samples associated with defined GIGWA project
2. `gigwa_get_sequences()`: Retrieves a list of chromosomes associated with defined GIGWA project
3. `gigwa_get_markers()`: Retrieves a list of SNP variants from selected GIGWA run. We can define the following parameters:
    - `start`: starting position of query
    - `end`: ending position of query
    - `chrom`: chromosome
    - `simplify`: defaults as TRUE, returns data in HapMap format with columns for rs#, alleles, chromosome and position

4. `gigwa_get_allelematrix()`: Retrieves a two-dimensional matrix of genotype data from the defined GIGWA run.
    - `samples`: optional list of sample IDs, if NULL, all samples are included
    - `start`: starting position of query
    - `end`: ending position of query
    - `chrom`: chromosome
    - `snps`: list of SNP variants to filter
    - `simplify`: defaults as TRUE, returns data in numeric coding (0, 1, 2 for diploids)
5. `gigwa_get_metadata()`: Retrieves associated metadata (if available)

```
# Get a list of all samples in the selected run
samples <- gigwa_get_samples()
```

```
  |
  |                                                                      |   0%
  |
  |======================================================================| 100%
```

```
# Get sequence list
chroms <- gigwa_get_sequences()

# Get markers
markers <- gigwa_get_markers()
```

```
  |
  |                                                                      |   0%
  |
  |====                                                                  |   5%
  |
  |=======                                                               |  11%
  |
  |===========                                                           |  16%
  |
  |===============                                                       |  21%
  |
  |==================                                                    |  26%
  |
```

```
  |======================                                                |  32%
  |
  |=========================                                             |  37%
  |
  |============================                                          |  42%
  |
  |================================                                      |  47%
  |
  |===================================                                   |  53%
  |
  |=======================================                               |  58%
  |
  |==========================================                            |  63%
  |
  |==============================================                        |  68%
  |
  |=================================================                     |  74%
  |
  |====================================================                  |  79%
  |
  |========================================================              |  84%
  |
  |============================================================          |  89%
  |
  |================================================================      |  95%
  |
  |======================================================================| 100%
```

```
# Get genotypic matrix
#marker_matrix <- gigwa_get_allelematrix()
```

15 Module 3.3: Data Quality Control and Filtering for SNP Data

Filtering SNP data is important for genetic and genomic studies in order to improve data quality, optimize resources and avoid noise by removing non-informative markers, and account for missing data. The most common filtering criteria we will focus on is **call rate**, **missing data** and **MAF (Minor Allele Frequency)**.

15.1 Call Rate

Call rate refers to the **percentage of non-missing data for a specific SNP marker**. It is calculated by dividing the number of non-missing individuals / total number of individuals for each marker. The filtering threshold will depend on the specific data set or investigation; however, common thresholds tend to be 0.9 or 0.95.]

```
# Load SNP data matrix
matrix <- read.table("data/BarleyMatrix.txt", sep = "\t", header = TRUE, row.names = 1, chec
```

```
# considering our n x m matrix with n markers and m individuals
# defining our threshold
call_rate <- 0.9

# filtering our matrix
filtered_matrix <- matrix[which(rowMeans(!is.na(matrix)) > call_rate),]
```

15.2 Missing data

Just as we can filter markers with too much missing data, we can filter individuals with too many missing values. Our missing data threshold will refer to the **percentage of non-missing data for each individual**.

```r
# defining our threshold
na_ind <- 0.8

# filtering our matrix
filtered_matrix <- filtered_matrix[,which(colMeans(!is.na(filtered_matrix)) > na_ind)]
```

15.3 MAF

Minor allele frequency refers to the **frequency of the least common allele for a particular SNP marker** in a given population. It is commonly used as a filtering criteria as it allows you to exclude markers that contribute little to population-level analyses and helps reduce noise.

For example, if I have a marker which is homozygous 0 (`AA`), heterozygous 1 (`AG`), and homozygous 2 (`GG`), each individual contributes 2 alleles. We then count the minor alleles across all individuals to obtain MAF.

```r
# calculating MAF for all markers
mafFreq <- apply(filtered_matrix, 1, function(row) {
  row <- row[!is.na(row)]
  maf <- sum(row) / (2 * length(row))
  maf <- min(maf, 1 - maf)
  maf
})

# filtering matrix according to maf
filtered_matrix <- filtered_matrix[mafFreq > 0.01,]
```

15.4 General Filtering

To simplify all previous steps, we can use the `filterData()` function from our package. The function allows you to define thresholds for call rate, MAF and missing individuals. By setting `stats = TRUE`, you can also get a data frame with statistics for the number of filtered markers and individuals by criteria.

```r
filter <- filterData(matrix, call_rate = 0.9, maf = 0.01, na_ind = 0.8)
```

Part V

Population Structure and Relatedness

16 Module 4.1: Kinship and Relatedness

Identifying relatedness between individuals is important to ensure samples are independent, as not accounting for kinship may distort posterior analyses such as GWAS or population structure. Kinship coefficients can help control confounding affects in association studies and can help infer subpopulations when studying structure. Moreover, not only may individuals be related, we can sometimes find duplicates of the same individual, which can skew posterior diversity estimates. Overall, studying kinship allows us to maintain the quality standard of our data.

16.1 Kinship

Kinship refers to the **genetic relatedness between individuals**, and it is a measure of **how much of their genomes two individuals share due to common ancestry**. Kinship is often evaluated by calculating a **kinship matrix**. We can use the `kinshipMatrix()` function from the ICARDA package to do this. This function has the option to choose the method we want to use to calculate the matrix by defining the `method` parameter. By default it it set to `"vanRaden"`, but we can choose between `"astle"`, `"IBS"`, and `"identity"`. We can also choose to save the matrix locally as a text file by defining the `save` parameter, which is set to `FALSE` by default.

```
# Importing our genotypic data
raw_matrix <- read.table("data/BarleyMatrix.txt", sep = "\t", header = TRUE, row.names = 1, c
```

```
# Filtering
matrix <- filterData(raw_matrix, call_rate = 0.9, maf = 0.01, na_ind = 0.8)

# We transpose our matrix (to have individuals as rows and makers as columns for posterior an
matrix <- t(matrix)

# Calculating kinship matrix using ICARDA package
kinshipMat <- kinshipMatrix(matrix, method = "vanRaden", save = FALSE)
```

```
# Generating a heatmap from our kinship matrix as an image in our working directory
kinshipHeatmap(kinshipMat, file = "output/figures/heatmap.png")
```

16.2 Duplicates

We can use the results from our kinship matrix to identify potential duplicates within our data. The existence of duplicates in a data set can mean different things. A sample may be genotyped multiple times or accidentally re-entried as a new individual sample and it is important to identify these errors. However, we can also find cases where the samples belong to different individuals but present no genetic variation, which can hint towards inbred lines or clonal lines. Moreover, duplicates inflate sample sizes, which can give us false confidence on GWAS or other statistical analyses.

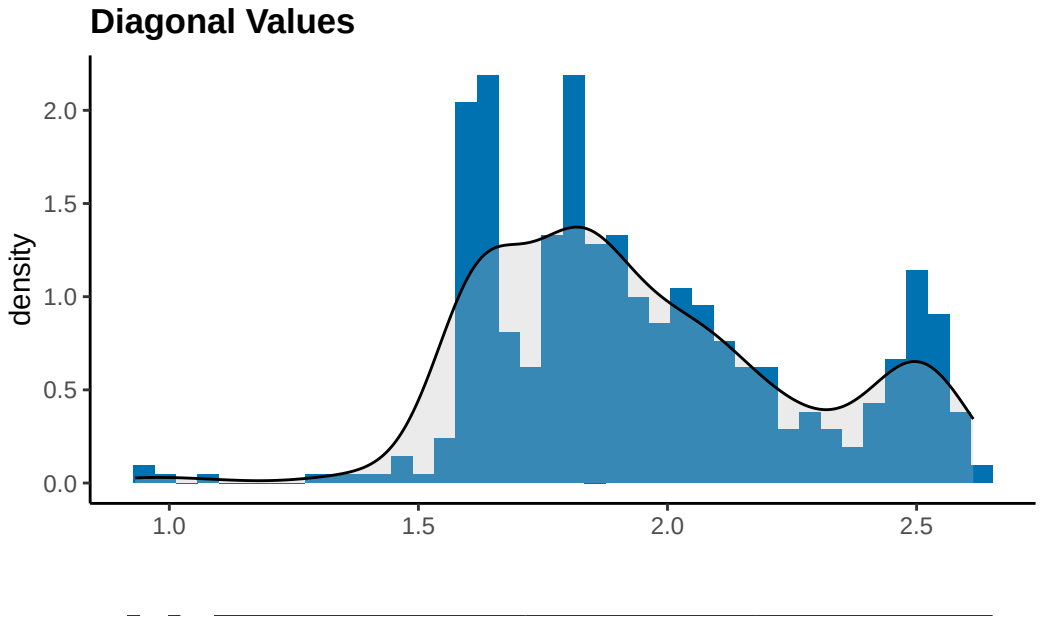
```
# Identifying duplicates by setting a similarity threshold and using the ICARDA package
duplicates <- kinshipDuplicates(matrix, threshold = 0.99, kinship = kinshipMat)

# Printing potential duplicates along with their kinship and correlation
duplicates$potentialDuplicates
```

```
        Indiv.A      Indiv.B    Value      Corr
1  ICARDA_G1416 ICARDA_G0226 1.623916 0.9988232
2  ICARDA_G0052 ICARDA_G0043 1.784647 0.9984566
3  ICARDA_G0140 ICARDA_G0059 1.840917 0.9983279
4  ICARDA_G0110 ICARDA_G0098 1.808388 0.9982951
5  ICARDA_G0119 ICARDA_G0098 1.807323 0.9980452
6  ICARDA_G0291 ICARDA_G0217 1.622423 0.9979949
7  ICARDA_G0298 ICARDA_G0296 1.606878 0.9979506
8  ICARDA_G0258 ICARDA_G0234 1.897229 0.9979442
9  ICARDA_G1421 ICARDA_G0301 1.613681 0.9979219
10 ICARDA_G0119 ICARDA_G0110 1.805239 0.9976256
11 ICARDA_G0232 ICARDA_G0230 1.658003 0.9976119
12 ICARDA_G1418 ICARDA_G0014 1.632860 0.9971155
13 ICARDA_G0473 ICARDA_G0137 1.973008 0.9971065
14 ICARDA_G0588 ICARDA_G0097 1.918720 0.9967419
15 ICARDA_G0887 ICARDA_G0146 1.833113 0.9966084
16 ICARDA_G1420 ICARDA_G0278 1.623460 0.9957251
17 ICARDA_G0257 ICARDA_G0218 1.672619 0.9955905
18 ICARDA_G0275 ICARDA_G0259 1.590987 0.9953467
19 ICARDA_G1201 ICARDA_G0627 1.816801 0.9951241
20 ICARDA_G0301 ICARDA_G0245 1.601896 0.9947260
```

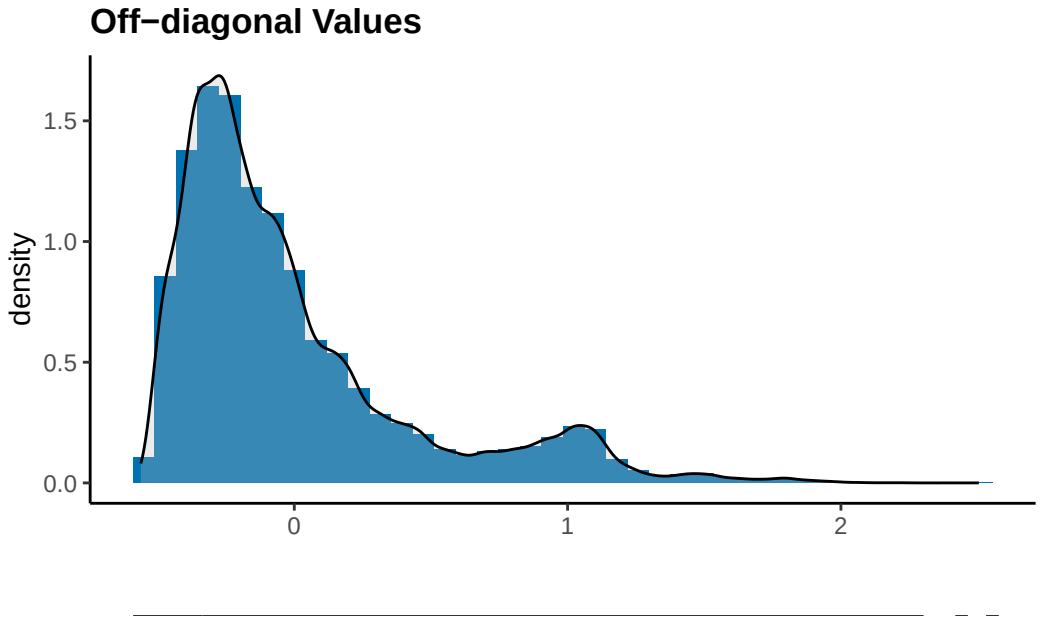
```
21 ICARDA_G1421 ICARDA_G0245 1.601728 0.9947013
22 ICARDA_G0244 ICARDA_G0234 1.884887 0.9945220
23 ICARDA_G0297 ICARDA_G0242 1.618348 0.9942201
24 ICARDA_G0814 ICARDA_G0146 1.825129 0.9938527
25 ICARDA_G1423 ICARDA_G0297 1.616111 0.9937178
26 ICARDA_G0258 ICARDA_G0244 1.879864 0.9935735
27 ICARDA_G1409 ICARDA_G0214 1.624520 0.9935612
28 ICARDA_G0889 ICARDA_G0146 1.820499 0.9934813
29 ICARDA_G0846 ICARDA_G0146 1.817510 0.9926027
30 ICARDA_G0889 ICARDA_G0887 1.816931 0.9924443
31 ICARDA_G0207 ICARDA_G0034 1.604201 0.9923286
32 ICARDA_G0501 ICARDA_G0500 1.862038 0.9923112
33 ICARDA_G0863 ICARDA_G0146 1.814894 0.9920920
34 ICARDA_G0503 ICARDA_G0498 2.504639 0.9920558
35 ICARDA_G0887 ICARDA_G0814 1.820125 0.9920372
36 ICARDA_G1383 ICARDA_G0983 1.692738 0.9916451
37 ICARDA_G0887 ICARDA_G0846 1.813754 0.9914600
38 ICARDA_G0887 ICARDA_G0863 1.811138 0.9909474
39 ICARDA_G1377 ICARDA_G0884 1.992010 0.9904147
40 ICARDA_G0795 ICARDA_G0146 1.808636 0.9903712
41 ICARDA_G1269 ICARDA_G0067 2.228613 0.9901741
```

```
# Printing histograms with the distribution of diagonal and off-diagonal values
duplicates$plots
```

```
[[1]]
```




```
[[2]]
```



If we decide we want to filter our data according to the potential duplicates we identified, we can use the `kinshipFilter()` function from the ICARDA package. This will give us a filtered version of our SNP matrix and kinship matrix.

```
filteredMatrix <- kinshipFilter(matrix, duplicates$potentialDuplicates, kinshipMat)
```

17 Module 4.2: Population Structure

18

Part VI

Marker-Trait Association (GWAS)

19

20

21

Part VII

Tools and Advanced Concepts

22

23