A Laboratory on Data Manipulation and Pre-processing with R the Tidyverse

Section 1: Introduction to Data Wrangling: A Methodological Framework

1.1 The Imperative of Clean Data

Data in its initial state is often described as "messy" or "dirty," characterized by a variety of issues including *missing values*, *structural inconsistencies*, *formatting errors*, and *outliers*. The process of *transforming this raw, inconsistent data into a structured, reliable format suitable* for analysis is known as data wrangling or data cleaning. It is a critical and often time-consuming component of the data science workflow, frequently consuming up to 80% of a data analyst's time.

The quality of any statistical model or analytical insight is fundamentally constrained by the quality of the input data. Inaccurate, incomplete, or improperly formatted data can lead to misleading conclusions, biased models, and ultimately, flawed decision-making. Therefore, a systematic and principled approach to data cleaning and pre-processing is an indispensable skill for any data scientist. This laboratory session is designed to equip you with both the conceptual framework and the practical tools to perform these essential tasks effectively within the R programming environment.

1.2 Introducing the Tidyverse

To navigate the challenges of data wrangling, we will utilize the ***tidyverse***, a collection of R packages designed specifically for data science.

1.2.1 Core Principles of the Tidyverse

Tidyverse aims to facilitate the "conversation" between an analyst and their data. This is achieved through three core principles:

1. **Consistency:** Tidyverse functions are designed to be consistent. For example, the first argument to any major function is always a data frame, and function names are typically verbs that describe their action (e.g., filter, select). This consistency reduces the cognitive load on the user, making the tools easier to learn and remember.
2. **Composability:** Tidyverse functions are simple and designed to do one thing well. Complex data manipulation tasks are solved not by a single, monolithic function with dozens of arguments, but by composing a sequence of these simple functions. This is made possible by the
   pipe operator (%>%), which allows for the creation of readable, sequential data processing pipelines. This approach mirrors the way an analyst thinks about a problem: as a series of discrete, logical steps.
3. **Tidy Data:** The entire ecosystem is built around the concept of "tidy data." A dataset is

considered tidy if it adheres to three rules:
    - Each variable has its own column.
    - Each observation has its own row.
    - Each value has its own cell.

This standardized structure is the target state for our data wrangling efforts. By converting messy data into a tidy format, we make it vastly easier to manipulate, model, and visualize.

1.2.2 The Tidyverse Ecosystem

While the tidyverse contains many packages, our focus in this lab will be on a core set that work together seamlessly:

- **dplyr**: The grammar of data manipulation, providing the core "verbs" for filtering, selecting, arranging, and summarizing data.
- **tidyr**: Tools for tidying data, such as converting between wide and long formats and handling missing values.
- **stringr**: A consistent and powerful set of tools for working with character strings, essential for cleaning text data.
- **ggplot2**: A declarative system for creating graphics, built on the "grammar of graphics". While not the focus of this lab, it is a key part of the tidyverse workflow for visualizing data at every stage of the cleaning process.

1.3 Lab Session Objectives

By the conclusion of this laboratory session, you will be able to:

1. Synthesize a complex, realistic dataset in R that contains common data quality issues.
2. Employ a systematic, diagnostic-driven workflow to identify and characterize issues.
3. Master the core data manipulation verbs of the dplyr package to subset, reorder, and transform data frames.
4. Implement robust strategies for handling missing values, including both exclusion and statistically sound imputation methods.
5. Utilize functions from stringr and dplyr to clean and standardize inconsistent textual and categorical data.
6. Apply normalization and transformation techniques to prepare numeric data for statistical modeling.
7. Construct a complete, end-to-end data cleaning pipeline by composing multiple functions with the pipe operator.

Section 2: Synthesizing the Laboratory Data: A Clinical Trial Dataset

2.1 Rationale for a Synthetic Dataset

In a pedagogical setting, working with a pre-existing, real-world dataset can sometimes obscure the learning objectives. The data's flaws may be too subtle, too complex, or not

directly aligned with the techniques being taught.

We will synthesize our own dataset from first principles. This approach offers a distinct advantage: we can *intentionally introduce a representative sample of the most common data quality problems such as missing values, inconsistent text formatting, and skewed distributions*—that directly map to the cleaning and pre-processing techniques this lecture aims to teach.

Furthermore, by constructing the dataset ourselves, we gain a perfect understanding of its underlying structure and the "ground truth" of its distributions. This allows us to precisely evaluate the effectiveness of our cleaning and transformation methods. The process also serves as a practical introduction to R's powerful functions for generating random data from various statistical distributions.

2.2 Understanding R's Data Generation Functions

R provides a comprehensive and consistent suite of functions for working with probability distributions. This consistency is embodied in the d/p/q/r naming convention, where a one-letter prefix is combined with the root name of a distribution (e.g., norm for the normal distribution). Understanding this convention is key to mastering data generation in R.

- **d for Density:** This function calculates the probability density (for continuous distributions) or probability mass (for discrete distributions) at a specific point. For example, dnorm(0) gives the height of the standard normal curve at its peak.
- **p for Probability:** This function calculates the cumulative distribution function (CDF), which is the probability of observing a value less than or equal to a given point. For example, pnorm(0) returns 0.5, as half of the standard normal distribution lies to the left of zero.
- **q for Quantile:** This is the inverse of the CDF. It takes a probability (a value between 0 and 1) and returns the corresponding quantile. For example, qnorm(0.975) returns approximately 1.96, the critical value for a 95% confidence interval.
- **r for Random:** This function generates random numbers (deviates) from the specified distribution. This is the function we will primarily use to construct our dataset.

For our dataset, we will employ the following specific r functions:

- **rnorm(n, mean, sd)**: Generates n random numbers from a normal distribution with a specified mean and standard deviation (sd).
- **rbinom(n, size, prob)**: Generates n random numbers from a binomial distribution, representing the number of "successes" in size trials with a prob probability of success. For our purposes, with size = 1, it is ideal for generating binary outcomes (0 or 1).
- **runif(n, min, max)**: Generates n random numbers from a uniform distribution between a min and max value, where every value in the range is equally likely.
- **rexp(n, rate)**: Generates n random numbers from an exponential distribution, which is a continuous distribution that is inherently right-skewed. This is useful for modeling

phenomena where small values are common and large values are rare.
- **rpois(n, lambda)**: Generates n random numbers from a Poisson distribution, which models the number of events occurring in a fixed interval of time or space, given an average rate of occurrence (lambda).

2.3 Dataset Construction: "Project Minerva Clinical Trial"

To provide a realistic context for our data wrangling tasks, we will frame our synthetic dataset as the output from **"Project Minerva,"** a fictional clinical trial. This trial investigates the efficacy of a new drug, "Novacrine," compared to a placebo for a specific condition. The data is collected from multiple hospital sites, which introduces plausible sources for data entry errors and inconsistencies. This narrative transforms the cleaning process from a series of abstract mechanical steps into a logical problem-solving exercise, where each cleaning task has a clear real-world motivation.

The following R code generates the raw dataset, minerva_raw_df. The set.seed() function is used to ensure that the random numbers generated are the same for everyone, making the results of this lab fully reproducible.

R

```
# Load necessary libraries for the entire session
# install.packages(c("tidyverse", "moments")) # Run this line if you haven't installed them
library(tidyverse)
# tidyverse is a collection of packages for data science,
# including tibble for creating data frames and dplyr for manipulating them
library(moments)

# simulates a "raw" or "dirty" dataset for a fictional clinical trial

# Set a seed for reproducibility
# the random numbers generated in the script will be the exact same every time it's run.
# This is crucial for getting consistent results.
set.seed(42)

# Define the number of patients for the dataset
num_patients <- 500

# --- Define ALL helper variables outside the tibble ---
# vectors (lists) that hold potential values
gender_pool <- c("Male", "Female", "male ", "F", "female", NA, "M")
site_pool <- c(" Site A ", "Site B", " Site C")
```

```
# The tibble() function from the tidyverse package is used here to create a modern,
# enhanced data frame called a "tibble." Think of it as a constructor for a spreadsheet-like table
# where each argument defines a column. It's generally preferred over the base R data.frame()
# because it has better printing methods, is more consistent, and doesn't automatically change variable types.
# --- Generate the Dataset ---
minerva_raw_df <- tibble(
  # Each argument inside tibble() creates a new column.

  # patient_id: A unique identifier for each patient
  patient_id = 1:num_patients,

  # treatment_group: Randomly assign patients
  treatment_group = ifelse(rbinom(num_patients, 1, 0.5) == 1, "Novacrine", "Placebo"),

  # age: Patient age, normally distributed
  age = round(rnorm(num_patients, mean = 55, sd = 8)),

  # gender: Sample from the pre-defined gender_pool vector
  gender = sample(gender_pool, num_patients, replace = TRUE),

  # biomarker_1: A normally distributed measurement with NAs
  biomarker_1 = {
    # it generates 500 normally distributed values.
    b1 <- rnorm(num_patients, mean = 120, sd = 20)
     # Then, it randomly selects 20 positions in b1 and replaces their values with NA to simulate missing data.
    b1[sample(1:num_patients, 20)] <- NA
    b1
  },

  # biomarker_2: A right-skewed measurement
  biomarker_2 = rexp(num_patients, rate = 0.1) + 5,

  # adverse_events: A count of events with data entry errors
  adverse_events = {
     # It generates 500 integer values from a Poisson distribution, which is often used for count data
    # (like the number of events occurring in an interval).
    ae <- rpois(num_patients, lambda = 0.5)
      # It then simulates data entry errors by randomly inserting an impossible value (-1) in 5
```

```
places.
  ae[sample(1:num_patients, 5)] <- -1
  ae
 },

 # hospital_site: Sample from the pre-defined site_pool vector
 hospital_site = sample(site_pool, num_patients, replace = TRUE)
)

# Verify the result (it will now work)
print(head(minerva_raw_df))
```

This code generates a tibble (a modern data frame) with 500 observations and 8 variables, each containing specific, intentional flaws that we will address in the subsequent sections.

Section 3: The dplyr Grammar of Data Manipulation

With our raw dataset created, we now turn to the tools for its manipulation. The dplyr package provides a "grammar" for data manipulation, consisting of a small set of core functions or "verbs" that can be combined to solve complex problems.

3.1 The Pipe Operator (%>%): The Syntax of Composition

The cornerstone of the tidyverse workflow is the **pipe operator**, %>%, which comes from the magrittr package. The pipe allows you to write a sequence of operations in a linear, readable fashion, from left to right. The fundamental logic of the pipe is simple:

x %>% f(y) is equivalent to f(x, y). It takes the object on its left-hand side (LHS) and passes it as the first argument to the function on its right-hand side (RHS).

Consider a simple task: selecting the age column from our dataset and then calculating the mean.

Without the pipe (nested functions):

```
R

mean(select(minerva_raw_df, age)$age, na.rm = TRUE)
```

This code is difficult to read because it must be interpreted from the inside out.

Without the pipe (intermediate variables):

```R
age_column <- select(minerva_raw_df, age)
mean_age <- mean(age_column$age, na.rm = TRUE)
```

This is more readable but clutters the workspace with temporary variables.

With the pipe:

```R
minerva_raw_df %>%
  select(age) %>%
  pull(age) %>%
  mean(na.rm = TRUE)
```

This version is both concise and highly readable. It can be interpreted as a series of steps: "Take the minerva_raw_df, *then* select the age column, *then* pull it out as a vector, *then* calculate the mean." This sequential, verb-based syntax is what makes the tidyverse so intuitive and powerful.

Note on the base R pipe (|>): As of R version 4.1.0, a native pipe operator, |>, is available in base R. While similar, the magrittr pipe (%>%) offers a richer feature set, such as the use of the . placeholder, which is prevalent in existing code and tutorials. For this lab, we will use the magrittr pipe.

3.2 Core Verbs: The Five Foundational Tools

The following five verbs form the foundation of dplyr and cover the vast majority of single-table data manipulation tasks.

3.2.1 filter(): Subsetting Rows

The filter() function allows you to select a subset of rows based on their values. The conditions are specified using logical expressions.

Example: Find all patients in the 'Novacrine' treatment group who are over 65 years old.

```R
minerva_raw_df %>%
  filter(treatment_group == "Novacrine" & age > 65)
```

Logical operators such as == (equal to), != (not equal to), >, <, >=, <=, & (and), | (or), and %in%

(is in a set of values) are used to construct these conditions.

3.2.2 arrange(): Ordering Rows

The arrange() function reorders the rows of a data frame based on the values in one or more columns. By default, it sorts in ascending order. To sort in descending order, wrap the column name in the

desc() helper function.

Example: Sort the dataset first by treatment group alphabetically, and then by age from oldest to youngest within each group.

R

```
minerva_raw_df %>%
  arrange(treatment_group, desc(age))
```

3.2.3 select(): Subsetting Columns

The select() function allows you to choose columns by their names. This is useful for focusing on a relevant subset of variables or for reordering columns.

Example: Create a new data frame with only the patient identifier and the two biomarker readings.

R

```
minerva_raw_df %>%
  select(patient_id, biomarker_1, biomarker_2)
```

select() also works with several helper functions for more powerful selections, such as starts_with("bio"), ends_with("_id"), contains("site"), and everything() (to select all columns, useful for reordering). You can also deselect columns by prefixing them with a minus sign (e.g.,

select(-gender, -age)).

3.2.4 mutate(): Creating and Modifying Columns

The mutate() function is used to add new columns that are functions of existing columns, or to modify existing columns. It is one of the most versatile verbs and is central to data cleaning and feature engineering.

Example: Create a new column biomarker_1_log that contains the natural logarithm of biomarker_1.

```R
minerva_raw_df %>%
  mutate(biomarker_1_log = log(biomarker_1))
```

You can create multiple new columns in a single mutate() call, separating each new column definition with a comma.

3.2.5 summarise() with group_by(): The Power of Aggregation

The summarise() (or summarize()) function collapses a data frame down to a single row containing summary statistics. On its own, it provides a global summary.

Example: Calculate the overall average age and the total number of patients in the trial.

```R
minerva_raw_df %>%
  summarise(
    mean_age = mean(age, na.rm = TRUE),
    n_patients = n()
  )
```

The true power of summarise() is unleashed when it is combined with group_by(). The group_by() function changes the scope of the other dplyr verbs from operating on the entire dataset to operating on it group-by-group.

Example: Calculate the mean age and number of patients for each treatment group separately.

```R
minerva_raw_df %>%
  group_by(treatment_group) %>%
  summarise(
    mean_age = mean(age, na.rm = TRUE),
    n_patients = n()
  )
```

This combination is a fundamental pattern in data analysis, allowing for the calculation of group-wise statistics with remarkable efficiency and clarity.

Section 4: A Systematic Data Cleaning Workflow

Data cleaning should not be a haphazard process. It is a systematic, diagnostic-driven workflow that can be broken down into distinct phases. We will follow a professional

methodology: first, we explore the data to diagnose its problems; second, we apply targeted treatments to fix those problems; and third, we re-evaluate to ensure the fixes were successful. This iterative approach ensures a thorough and justifiable cleaning process.

4.1 Phase 1: Initial Exploration and Diagnosis

Before altering the data, we must first understand its structure and identify its flaws.

4.1.1 Structural Overview with str()

The str() function (short for "structure") provides a compact, one-line summary for each column in the data frame. It is often the very first function one should run on a new dataset.

```
R

str(minerva_raw_df)
```

The output of str() reveals critical information at a glance:

- The overall dimensions of the data (200 observations, 8 variables).
- The name of each column.
- The data type of each column (e.g., num for numeric, chr for character).
- The first few values of each column, which can reveal formatting issues like the leading/trailing whitespace in hospital_site.

4.1.2 Statistical Summary with summary()

The summary() function provides descriptive statistics for each variable. The output is intelligently tailored to the data type of each column.

```
R

summary(minerva_raw_df)
```

For numeric columns like age and biomarker_1, summary() provides the "five-number summary" (minimum, 1st quartile, median, 3rd quartile, maximum) plus the mean. This output is a powerful diagnostic tool:

- **Missing Values:** The count of NAs is explicitly listed for each variable (e.g., in biomarker_1).
- **Erroneous Values:** The minimum value for adverse_events is listed as -1, which is biologically impossible and indicates a data entry error.
- **Potential Outliers:** Extreme minimum or maximum values can suggest outliers that may require further investigation.

For character columns like gender, summary() provides the length, class, and mode.

4.1.3 Quantifying Missingness with is.na()

While summary() gives a count of missing values per column, it is often useful to get a more direct and programmable assessment. The is.na() function returns a logical vector of the same size as its input, with TRUE for every NA value and FALSE otherwise. Combining this with

sum() (which treats TRUE as 1 and FALSE as 0) allows us to count missing values precisely.

Example: Count the total number of missing values in the entire dataset.

R

```
sum(is.na(minerva_raw_df))
```

Example: Count the number of missing values in each column.

R

```
colSums(is.na(minerva_raw_df))
```

This diagnostic phase has confirmed several issues: biomarker_1 and gender have missing values; adverse_events has impossible negative values; and gender and hospital_site have inconsistent text formatting.

4.2 Phase 2: Handling Missing Values (NA)

Now that we have identified the missing values in biomarker_1 and gender, we must decide on a strategy to handle them. The choice of strategy is context-dependent and has significant statistical implications.

4.2.1 Strategy 1: Exclusion of Missing Values

The simplest approach is to remove observations with missing data.

- **Listwise Deletion:** The tidyr::drop_na() function removes any row that contains an NA in any of the specified columns. If no columns are specified, it removes rows with an NA in *any* column. This is a powerful but potentially dangerous method. While it guarantees a complete dataset, it can lead to a significant loss of data and statistical power. More critically, if the missingness is not completely random, it can introduce systematic bias into the analysis.
  Example: Remove all rows with any missing values.
  R
  ```
  minerva_raw_df %>% drop_na()
  ```

- **Targeted Deletion:** A more nuanced approach is to remove rows only if a value is missing in a column that is critical for a specific analysis.
  Example: Remove rows only if biomarker_1 is missing.
  R
  ```
  minerva_raw_df %>% drop_na(biomarker_1)
  ```

4.2.2 Strategy 2: Imputation of Missing Values

Imputation is the process of replacing missing data with substituted values. This can preserve sample size but must be done carefully to avoid distorting the data's natural distribution.

- **Mean/Median Imputation:** A common simple method is to replace NAs with the mean or median of the non-missing values in that column. This is easily accomplished with mutate() and ifelse(). It is crucial to include the argument na.rm = TRUE when calculating summary statistics like mean() or median(), as these functions will otherwise return NA if any NAs are present in their input.
  Example: Impute missing biomarker_1 values with the overall mean of biomarker_1.
  R
  ```
  minerva_raw_df %>%
   mutate(biomarker_1_imputed = ifelse(is.na(biomarker_1),
                               mean(biomarker_1, na.rm = TRUE),
                               biomarker_1))
  ```

- **Advanced Imputation (Group-wise Mean):** A more statistically robust approach is to impute missing values based on subgroups within the data. For our clinical trial, it is more plausible that a patient's biomarker level is similar to others in the same treatment group. We can implement this using a combination of group_by() and mutate(). This method better preserves the underlying structure of the data compared to using the global mean.[68]
  Example: Impute missing biomarker_1 values with the mean of their respective treatment_group.
  R
  ```
  minerva_imputed_df <- minerva_raw_df %>%
   group_by(treatment_group) %>%
   mutate(biomarker_1 = ifelse(is.na(biomarker_1),
                         mean(biomarker_1, na.rm = TRUE),
                         biomarker_1)) %>%
   ungroup() # It's good practice to ungroup after the operation
  ```

4.3 Phase 3: Correcting Inconsistent and Unstructured Data

This phase addresses the formatting errors in our character-based columns and corrects the erroneous numeric data.

4.3.1 Text Standardization with stringr

The stringr package provides a consistent and powerful set of functions for string manipulation, which integrate seamlessly within dplyr::mutate().

- **Trimming Whitespace:** The str_trim() function removes leading and trailing whitespace

from a character string. This is essential for cleaning the hospital_site column.
- **Standardizing Case:** The str_to_lower(), str_to_upper(), or str_to_title() functions can be used to enforce a consistent case, which is necessary for the gender column.

Example: Clean both hospital_site and gender in one mutate call.

R

```
minerva_text_cleaned_df <- minerva_imputed_df %>%
 mutate(
  hospital_site = str_trim(hospital_site),
  gender = str_to_lower(gender)
 )
```

4.3.2 Recoding Categorical Variables with case_when()

After standardizing the text, the gender column still contains multiple representations (e.g., "m", "male"). The dplyr::case_when() function is the ideal tool for recoding these into a consistent set of values. It is a vectorized equivalent of a series of if-else if-else statements and is far more readable than nested ifelse() calls.

The syntax is a series of condition ~ value formulas. The first TRUE condition determines the output value. A final TRUE ~ "default_value" can be used as a catch-all else case.

Example: Consolidate the gender column.

R

```
minerva_recoded_df <- minerva_text_cleaned_df %>%
 mutate(
  gender = case_when(
   gender %in% c("m", "male") ~ "Male",
   gender %in% c("f", "female") ~ "Female",
   TRUE ~ NA_character_ # Recode any other values (including original NAs) to NA
  )
 )
```

4.3.3 Converting to Factors

Once character columns are clean and consistent, they should be converted to factors. A factor is R's special data type for categorical variables. It stores the data as integers, with an associated set of character labels for the levels. This is more memory-efficient and, critically, ensures that statistical modeling functions (like

lm() or glm()) correctly interpret the variable as categorical rather than continuous.

Example: Convert relevant columns to factors using mutate() and across().

```R
minerva_factored_df <- minerva_recoded_df %>%
  mutate(across(c(treatment_group, gender, hospital_site), as.factor))
```

4.3.4 Correcting Erroneous Numeric Data

Our diagnostic phase revealed impossible negative values in the adverse_events column. We can correct these using mutate() and ifelse(). A reasonable strategy is to assume these were data entry errors and should be treated as missing data (NA), which could then be imputed if necessary.

Example: Replace negative adverse_events with NA.

```R
minerva_corrected_df <- minerva_factored_df %>%
  mutate(adverse_events = ifelse(adverse_events < 0, NA_integer_, adverse_events))
```

After this phase, our data is considered "clean." All missing values have been handled, inconsistencies have been resolved, and data types are correctly set.

Section 5: Data Pre-processing for Analysis and Modeling

Data pre-processing is the final step before analysis or modeling. While cleaning focuses on correcting errors and inconsistencies, pre-processing focuses on transforming the data into a format that is optimal for the specific algorithm or statistical test to be used.

5.1 Rationale for Pre-processing

Many statistical models and machine learning algorithms have underlying assumptions about the data they receive. For instance, linear regression assumes a linear relationship between predictors and the outcome and normally distributed residuals. Distance-based algorithms like K-Nearest Neighbors (KNN) can be biased by features with large scales. Pre-processing aims to transform the data to better meet these assumptions or requirements, often leading to improved model performance and more reliable results.

5.2 Transforming Skewed Data

5.2.1 Diagnosing Skewness

Skewness is a measure of the asymmetry of a probability distribution. A distribution is right-skewed (positive skew) if the tail on the right side is longer, and left-skewed (negative

skew) if the left tail is longer. Our

biomarker_2 variable was intentionally generated from an exponential distribution, which is right-skewed.

We can diagnose this both visually and statistically.

- **Visual Diagnosis:** A histogram created with ggplot2 will clearly show the long tail to the right.
- **Statistical Diagnosis:** The skewness() function from the moments package provides a numerical measure. A value greater than 1 or less than -1 typically indicates substantial skewness.[85]

Example: Check the skewness of biomarker_2.

R

```
# Visual
ggplot(minerva_corrected_df, aes(x = biomarker_2)) +
  geom_histogram(bins = 30, fill = "skyblue", color = "black") +
  theme_minimal() +
  ggtitle("Distribution of Biomarker 2 (Original)")

# Statistical
skewness(minerva_corrected_df$biomarker_2, na.rm = TRUE)
```

5.2.2 Logarithmic Transformation

For right-skewed data, the most common transformation is the **logarithmic transformation**. Taking the logarithm of the values compresses the range of the larger values more than the smaller values, effectively "pulling in" the long right tail and making the distribution more symmetric.

It is important to note that the logarithm is undefined for zero and negative numbers. If the data contains zeros, a small constant (e.g., 1) should be added to all values before taking the log (i.e., log(x + 1)).

Example: Apply a log transformation to biomarker_2 and visualize the result.

R

```
minerva_transformed_df <- minerva_corrected_df %>%
  mutate(biomarker_2_log = log(biomarker_2))

# Visualizing the transformed data
ggplot(minerva_transformed_df, aes(x = biomarker_2_log)) +
  geom_histogram(bins = 30, fill = "lightgreen", color = "black") +
```

```
  theme_minimal() +
  ggtitle("Distribution of Biomarker 2 (Log-Transformed)")
```

The resulting histogram should appear much more symmetric, closer to a normal distribution.

5.3 Feature Scaling: Normalization and Standardization

Feature scaling is the process of transforming numeric features to a common scale. This is essential for algorithms that are sensitive to the magnitude of feature values, ensuring that no single feature dominates the analysis simply because its values are larger.

5.3.1 Method 1: Min-Max Normalization

Min-Max normalization rescales the data to a fixed range, typically . The transformation is performed using the formula:

$$x_{new} = \frac{x_{max} - x_{min}}{x - x_{min}}$$

This method is useful when the data has a known, bounded range and for algorithms like neural networks that expect inputs in the range. However, it is sensitive to outliers, as a single extreme value can drastically shrink the range of the other data points.

Implementation: A custom function can be written, or the calculation can be done directly within mutate().

R

```
minerva_normalized_df <- minerva_transformed_df %>%
  mutate(age_normalized = (age - min(age)) / (max(age) - min(age)))
```

5.3.2 Method 2: Z-Score Standardization

Z-score standardization (or simply "standardization") rescales the data to have a mean of 0 and a standard deviation of 1. The formula is:

$$z = \frac{x - \mu}{\sigma}$$

where μ is the mean and σ is the standard deviation of the feature. This method is generally more robust to outliers than Min-Max normalization and is preferred for algorithms like Principal Component Analysis (PCA) and many forms of regression.

Implementation: R's built-in scale() function performs Z-score standardization directly.

R

```
minerva_standardized_df <- minerva_transformed_df %>%
  mutate(age_standardized = as.numeric(scale(age)))
```

Note: The scale() function returns a matrix object, so it is often necessary to coerce it back to a numeric vector using as.numeric().

Section 6: The Complete Pipeline: From Raw to Analysis-Ready

6.1 Synthesizing the Workflow

We have now explored the individual steps of diagnosing, cleaning, and pre-processing our data. The true elegance and power of the tidyverse lies in its ability to chain these individual operations into a single, cohesive, and highly readable pipeline. This approach not only produces the desired clean dataset but also serves as a reproducible script that documents the entire wrangling process from start to finish.

6.2 The Code

The following code block represents the complete, end-to-end pipeline. It starts with the minerva_raw_df and applies all the necessary transformations in a logical sequence, assigning the final, analysis-ready result to a new data frame, minerva_clean_df.

R

```
minerva_clean_df <- minerva_raw_df %>%

  # --- Phase 1: Correct Erroneous and Inconsistent Data ---

  # Correct impossible values in adverse_events (replace -1 with NA)
  mutate(adverse_events = ifelse(adverse_events < 0, NA_integer_, adverse_events)) %>%

  # Clean and standardize text columns
  mutate(
    hospital_site = str_trim(hospital_site),
    gender = str_to_lower(gender)
  ) %>%

  # Recode gender into consistent categories
  mutate(
    gender = case_when(
      gender %in% c("m", "male") ~ "Male",
      gender %in% c("f", "female") ~ "Female",
      TRUE ~ NA_character_
    )
  ) %>%

  # --- Phase 2: Handle Missing Values ---
```

```r
# Impute missing biomarker_1 values with the mean of their treatment group
group_by(treatment_group) %>%
mutate(
  biomarker_1 = ifelse(is.na(biomarker_1), mean(biomarker_1, na.rm = TRUE), biomarker_1)
) %>%
ungroup() %>%

# For any remaining NAs (e.g., in gender or adverse_events), remove the entire row.
# This is a final, decisive step after targeted imputation.
drop_na() %>%

# --- Phase 3: Pre-processing for Modeling ---

# Apply log transformation to the skewed biomarker_2
mutate(biomarker_2_log = log(biomarker_2)) %>%

# Apply Z-score standardization to numeric predictors
mutate(
  age_z = as.numeric(scale(age)),
  biomarker_1_z = as.numeric(scale(biomarker_1))
) %>%

# Convert categorical columns to factors
mutate(across(c(treatment_group, gender, hospital_site), as.factor)) %>%

# --- Phase 4: Final Selection and Reordering ---

# Select and reorder columns for a clean final dataset
select(
  patient_id,
  treatment_group,
  age,
  age_z,
  gender,
  hospital_site,
  biomarker_1,
  biomarker_1_z,
  biomarker_2,
  biomarker_2_log,
  adverse_events
)
```

6.3 Verification

Finally, we inspect the structure and summary of our new minerva_clean_df to confirm that all data quality issues have been resolved.

```R
# Check the structure of the final, clean dataset
str(minerva_clean_df)

# Get a statistical summary of the final, clean dataset
summary(minerva_clean_df)
```

The output should now show a clean, well-structured data frame:

- No NA values.
- gender, hospital_site, and treatment_group are factors with consistent levels.
- adverse_events has a minimum value of 0 or greater.
- New transformed and standardized columns (biomarker_2_log, age_z, biomarker_1_z) are present and correctly calculated.

This final data frame is now truly analysis-ready, and the pipeline that created it is a transparent, reproducible record of every decision made during the data wrangling process.

Laboratory Exercises

Please use the original minerva_raw_df as the starting point for each exercise.

Exercise 1: Filtering & Arranging

Create a new data frame named site_a_males that contains only male patients from 'Site A'. The final data frame should be sorted by the age column in descending order (oldest to youngest).

Exercise 2: Mutating & Selecting

Create a new column in minerva_raw_df named biomarker_ratio which is calculated as the ratio of biomarker_1 to biomarker_2. Then, create a new data frame named biomarker_data that contains only the patient_id column and your new biomarker_ratio column.

Exercise 3: Grouping & Summarising

Calculate the mean age and the median of biomarker_1 for each unique combination of treatment_group and hospital_site. Your final output should be a tibble with four columns: treatment_group, hospital_site, mean_age, and median_biomarker_1. Remember to handle missing values in your calculations.

Exercise 4: Missing Value Imputation

Create a new data frame named imputed_data. In this data frame, perform the following two imputation steps in order:

1. Impute the missing gender values with the statistical mode (the most frequent value). *Hint: You will need to calculate the mode first.*
2. After imputing gender, impute the missing biomarker_1 values with the median biomarker_1 value calculated *separately for each gender*.

Exercise 5: Text Cleaning & Recoding

Using minerva_raw_df, create a new data frame site_info_df. This data frame should have two new columns:

1. cleaned_site: A corrected version of the hospital_site column with all leading/trailing whitespace removed.
2. site_region: A new column created using case_when() where 'Site A' is mapped to 'East',

'Site B' is mapped to 'West', and 'Site C' is mapped to 'North'.

Exercise 6: Normalization & Transformation

Create a new data frame named processed_numeric_df from minerva_raw_df. This data frame should contain the original patient_id and age columns, along with two new columns:

1. biomarker_2_log: The result of applying a natural log transformation to the biomarker_2 column.
2. age_scaled: The result of applying Z-score standardization to the age column.

Capstone Exercise 7: Full Pipeline

Write a single, chained sequence of dplyr commands starting from minerva_raw_df to perform the following transformations in order:

1. Remove all rows where biomarker_1 is NA.
2. Correct the negative values in adverse_events by replacing them with 0.
3. Standardize the gender column to have two levels, "Male" and "Female", handling all inconsistencies. Any values that cannot be mapped to these should become NA.
4. Standardize the hospital_site column by removing whitespace.
5. Create a new logical variable is_high_risk which is TRUE if a patient's age is greater than 65 AND they have had more than 0 adverse_events, and FALSE otherwise.
6. Arrange the final data frame first by treatment_group (alphabetically) and then by patient_id (ascending).
7. Store the final result in a new data frame named minerva_final_df.

Appendix A: Table of dplyr Join Functions

While this lab focuses on single-table manipulation, a crucial part of data wrangling involves combining data from multiple sources. The dplyr package provides a consistent and powerful set of functions for this purpose, known as joins. The following table summarizes the four main "mutating joins," which add columns from a second table (y) to a primary table (x) based on matching key columns.

Function	Rows Kept from x (Left Table)	Rows Kept from y (Right Table)	Use Case
inner_join(x, y)	Only rows with a match in y.	Only rows with a match in x.	Keep only observations that are present in both tables.
left_join(x, y)	All rows.	Only rows with a match in x.	Keep all information from the primary table (x) and add matching information from y. This is the most common type of join.
right_join(x, y)	Only rows with a match in y.	All rows.	Keep all information from the secondary table (y) and add matching information from x.
full_join(x, y)	All rows.	All rows.	Keep all observations from both tables, filling non-matches with NA. Useful for identifying mismatches between tables.