MGT4018/MGT4090 Lab 1

Bernd Wurth

Table of contents

1 Introduction

This lab is the same as the SPSS Lab 1 for MGT4018 and MGT4090. We use base R functions as the default. While there are many R packages available, understanding base R operations provides a strong foundation for data analysis.

Alternatives using R packages

Alternatives for achieving the same outcomes using different R packages are provided in green boxes. If you want to explore these alternatives, we need to install and load the necessary R packages. Run this code once to install packages:

```
install.packages(c(
  "tidyverse",  # for data manipulation and visualization
  "gt",         # for creating beautiful tables
  "gtsummary",  # for summary statistics
  "janitor"     # for tabulation functions
))
```

Now load the packages:

```
library(tidyverse)
library(gt)
library(gtsummary)
library(janitor)
```

2 Exercise A: Creating and Exploring a Dataset in R

This lab will guide you through creating a dataset, assigning labels, and conducting basic analyses using R. You will learn how to create variables, enter data, and generate summary tables similar to those you would in SPSS.

2.1 Step A1: Setup

Before starting any work in R, it is important to organize your files. This ensures that your scripts, datasets, and outputs are easy to manage and reproducible. Projects are strongly recommended for better organization and reproducibility, but setting a working directory is an alternative if needed (see also the respective section in `0. Getting Started`).

> **Note**
>
> You can either work through the following steps and copy/paste the respective code into the `Lab_Exercise_A.R` file that you will create in Step 1 or download the R script for Exercise A and follow the instructions below and save the downloaded file in the `scripts` folder that you will create.

2.1.1 Option 1: Create a Project

The best way to organize your work is to create an RStudio project. This keeps all related files in a single folder. Follow these steps:

1. Open RStudio.
2. Go to **File > New Project**.
3. Select **New Directory** and then **New Project**.
4. Choose a location on your computer and give the project a name, e.g., `ResearchMethodsLab`. Avoid spaces in folder and file names.
5. Click **Create Project**. RStudio will open a new session within the project folder.
6. It is good practice to not have you files in this main folder, but create a set of subfolders similar to the following. Navigate to your project folder and create the following subfolders:

```
ResearchMethodsLab/
  data/
  scripts/
  output/
     figures/
     tables/
  docs/
  Lab1.Rproj
```

7. Create a new **R script**: Go to **File > New File > R Script**.
8. Save the script in your `scripts` folder with an appropriate name, e.g., `Lab1_Exercise_A.R`.

Now you are ready to begin your work in R and continue with Step A2!

2.1.2 Option 2: Set Working Directory Manually

If you are not using a project, you will need to set a working directory manually. The working directory is the folder where R looks for files and saves outputs.

1. In your Finder (Mac) or Explorer (Windows), create a new folder for the R Labs (e.g., `ResearchMethodsLab`) as well as the subfolder structure below. Avoid spaces in folder and file names.

```
ResearchMethodsLab/
  data/
  scripts/
  output/
     figures/
     tables/
  docs/
```

2. Open RStudio.
3. Create a new **R script**: Go to **File > New File > R Script**.
4. Save the script in the `scripts` folder that you previously created with an appropriate name, e.g., `Lab_Exercise_A.R`.
5. Set the working directory in your script using the `setwd()` function. Copy the following code into your script, change the path to the older that you created, and execute the script.

```r
setwd("path/to/your/folder")
```

You can check whether the working directory is set to the folder that you want to by using the following code:

```r
# Display the current working directory
getwd()
```

Now you are ready to begin your work in R and continue with Step A2!

2.2 Step A2: Creating or Loading the Dataset

For practice purposes, try both options so you familiarise yourself with creating and loading datasets in R.

2.2.1 Option 1: Create your Dataset

In base R, we will create our vectors and combine them into a data frame. Use the following code to create your variables:

```r
# Create vectors for our data
id <- 1:30

# Create age bands with labels
age_bands <- factor(
  c(1,3,2,4,2,5,5,2,3,1,4,1,3,2,4,2,5,5,2,3,1,4,2,4,2,5,5,2,3,1),
  levels = 1:5,
  labels = c("<21", "21-30", "31-40", "41-50", ">50")
)

# Create gender categories with labels
gender <- factor(
  c(0,1,1,0,0,0,1,1,1,0,9,0,2,1,0,0,1,1,0,0,1,0,0,0,1,1,1,0,9,9),
  levels = c(0,1,2,9),
  labels = c("Male", "Female", "Other", "Prefer not to say")
)
```

> **Note**
>
> In R, we use `factor()` to create categorical variables. This is similar to value labels in SPSS. The `levels` argument specifies the underlying codes, while `labels` provides the human-readable labels.

We now combine the variables that we created in the previous step into a `data.frame`.

```r
# Combine into a data frame
survey_data <- data.frame(
  ID = id,
  AgeBand = age_bands,
  Gender = gender
)
```

You can easily explore and check the basic structure of your data and get a summary:

```r
# View the first few rows using head()
head(survey_data)

# Examine the structure
str(survey_data)

# Get basic summary statistics
summary(survey_data)
```

When working with data in R, saving and reloading data efficiently is crucial for reproducibility and sharing. Two common file formats for saving data in R are **.csv files** and **.rds files**. While both serve the purpose of persisting data for future use, they have distinct differences in terms of structure, use cases, and functionality.

You can use the following code to save your dataset in either format. In line with the best practice around folder structures, the code below works only if you created the `data` folder within your project folder or working directory. **For this tutorial, save your output as a .csv file (without row names).**

```
# Save as R data file
saveRDS(survey_data, "data/survey_data.rds")

# Save as .csv
write_csv(survey_data, "data/survey_data.csv")

# By default, the data is exported with row names. Now to export data without row names we simply have
write_csv(survey_data, "survey_data.csv", row.names=FALSE)
```

A **.csv file** (comma-separated values) is a plain text format commonly used for storing tabular data. Each row in the file corresponds to a row in the dataset, and columns are separated by commas. In general, .csv files are platform-independent, widely supported by software like Excel, and easily shared or imported into other programming environments.

An **.rds file** is a native R binary format used to store a single R object. Unlike .csv, .rds files preserve all attributes and data types, including factors, column classes, and even more complex structures like lists, models, or custom objects. The table below outlines the main differences between saving a data frame in each format.

Table 1:

Feature	CSV	RDS
Format	Plain text	Bin
Portability	Cross-platform and software-independent	R-sp
Metadata	Does not retain R-specific attributes	Ret
Complex objects	Supports tabular data only	Sup
Human-readable	Yes	No
Efficiency	Slower for large datasets	Fast
Use case	Sharing data with non-R users or if the data will be processed in other tools like Excel or Python	Stor

2.2.2 Option 2: Load your Dataset

As an alternative to creating your own dataset, you can download the dataset. Copy this file into your `data` folder and then run the following code:

```
# Load the CSV file stored in the "data" folder
survey_data <- read.csv("data/lab1-exercise-a.csv")
```

You can easily explore and check the basic structure of your data and get a summary:

```
# View the first few rows using head()
head(survey_data)

# Examine the structure
str(survey_data)

# Get basic summary statistics
summary(survey_data)
```

2.3 Step A3: Modifying your Dataset

> **Note**
>
> This step is not needed, the dataset that you created with the code (Step A2, option 1) or downloaded (Step A2, option 2) are already correct, but have a look and get an overview of what you can do with R. We create a copy of the data frame that we use for the remainder of the lab. **Please run the following code before experimenting with possible modifications:**

```r
survey_data_copy <- survey_data
```

Data frames are among the most commonly used data structures in R, especially for storing and analyzing tabular data. Once a data frame is loaded or created, it is often necessary to modify it to suit specific analytical needs. Common modifications include editing specific values, adding or removing columns and rows, and renaming variables. These operations can be performed efficiently using **base R**, without the need for additional packages. Understanding these basic techniques is essential for working effectively with data in R.

2.3.1 Editing Specific Values

You can directly access and modify specific elements of a data frame using **indexing** with square brackets `[ , ]`. The format is `data[row, column]`, where you specify the row and column to edit. You can also use column names for clarity.

```r
# Edit the Gender of the participant with ID 15
survey_data_copy[survey_data_copy$ID == 15, "Gender"] <- "Female"
print(survey_data_copy[survey_data_copy$ID == 15, ])
```

Here, the gender for the participant with ID 15 is updated to "Female."

2.3.2 Adding a Column

To add a new column to a data frame, use the `$` operator or specify the new column name within square brackets.

```r
# Add a new column indicating if the participant is above 30 based on AgeBand
survey_data_copy$Above30 <- survey_data_copy$AgeBand %in% c("31-40", "41-50", ">50")
print(head(survey_data_copy))
```

This new column indicates whether each participant belongs to an age group over 30.

2.3.3 Adding a Row

To append a new row, use the `rbind()` function, which binds rows together.

```r
# Add a new participant to the data frame
new_row <- data.frame(ID = 31, AgeBand = "<21", Gender = "Other", Above30 = FALSE)
survey_data_copy <- rbind(survey_data_copy, new_row)
print(tail(survey_data_copy))
```

2.3.4 Removing a Column

You can remove a column by setting it to `NULL` or by subsetting the data frame without that column.

```r
# Remove the "Above30" column
survey_data_copy$Above30 <- NULL
print(head(survey_data_copy))
```

2.3.5 Removing a Row

To remove a row, subset the data frame, excluding the unwanted row.

```r
# Remove the row where ID is 10
survey_data_copy <- survey_data_copy[survey_data_copy$ID != 10, ]
print(head(survey_data_copy))
```

2.3.6 Renaming Columns

Renaming columns can be done by directly modifying the `names()` of the data frame.

```r
# Rename columns
names(survey_data_copy) <- c("ParticipantID", "AgeGroup", "GenderIdentity")
print(head(survey_data_copy))
```

2.3.7 Combining Data Frames

You can merge two data frames using `merge()` to combine them based on a common column.

```
# Create another data frame with additional information
additional_data <- data.frame(
  ParticipantID = c(1, 2, 3, 4, 5),
  CompletedSurvey = c(TRUE, TRUE, FALSE, TRUE, FALSE)
)

# Merge data frames by the "ParticipantID" column
survey_data_copy <- merge(survey_data_copy, additional_data, by = "ParticipantID", all.x = TRUE)
print(head(survey_data_copy))
```

2.3.8 Summary

These are some of the most common operations for modifying data frames in base R. Mastering these techniques will allow you to efficiently manipulate data frames and tailor them to your analytical needs:

- **Edit specific values:** Modify elements using `data[row, column]`.
- **Add a column:** Use `$` or square brackets.
- **Add a row:** Use `rbind()`.
- **Remove a column:** Set it to `NULL` or subset the data frame.
- **Remove a row:** Subset the data frame excluding the row.
- **Rename columns:** Modify `names()` directly.
- **Combine data frames:** Use `merge()`.

2.4 Step A4: Frequency Tables (Summary Statistics)

Frequency tables are a fundamental tool in data analysis for summarizing and understanding the distribution of categorical variables. They show how often each category appears in a dataset, helping analysts quickly grasp patterns and trends. In business research, frequency tables are commonly used to analyze survey results, customer demographics, or product categories. In R, base functions like `table()` and `prop.table()` provide a straightforward way to create and analyze frequency tables.

This guide introduces frequency tables using the `survey_data` data frame and demonstrates their creation and interpretation.

2.4.1 When and Why to Use Frequency Tables

Frequency tables are particularly useful when:

- You need to summarize **categorical data** (e.g., age groups, gender, product categories).
- You want to identify **dominant categories** or patterns in responses.
- You need a quick check for **data quality** (e.g., missing values or unexpected categories).
- You are preparing data for **visualization** or further statistical analysis.

By condensing raw data into an easy-to-read format, frequency tables simplify decision-making and analysis.

2.4.2 Creating a Basic Frequency Table

The `table()` function in R generates frequency counts for one or more variables. For example, to count the number of participants in each age bandn and for each gender:

```
# Frequency table for AgeBand
age_band_freq <- table(survey_data$AgeBand)

# Frequency table for Gender
gender_freq <- table(survey_data$Gender)

# Display the results
print("Frequency table for AgeBand:")
print(age_band_freq)
print("Frequency table for Gender:")
print(gender_freq)
```

This table displays the count of participants in each age group.

2.4.3 Adding Proportions

Proportions show the relative frequency of each category as a percentage of the total. Use the `prop.table()` function to calculate proportions from a frequency table:

```
# Proportions for AgeBand
age_band_prop <- prop.table(age_band_freq)
print(age_band_prop)
```

The output shows the proportion of participants in each age group. To display percentages, multiply by 100:

```
# Proportions as percentages
age_band_percent <- round(prop.table(age_band_freq) * 100, 2)
print(age_band_percent)
```

This provides a clear picture of the distribution of age groups as percentages. You can also combine the basic frequency table with the proportions and print the combined results:

```
# Frequency table for AgeBand
age_band_freq <- table(survey_data$AgeBand)
# Proportions as percentages for AgeBand
age_band_percent <- round(prop.table(age_band_freq) * 100, 2)

# Combine frequencies and percentages
age_summary <- cbind(
  Frequency = age_band_freq,
  Percentage = age_band_percent
)

# Display the results
print("AgeBand Distribution:")
print(age_summary)
```

2.4.4 Summary Statistics for Frequency Tables

Once you have created a frequency table, you might want to extract specific insights:

- **Mode:** The category with the highest frequency.
- **Rare categories:** Categories with very low frequencies.

```
# Find the most frequent AgeBand
most_frequent_age_band <- names(age_band_freq[which.max(age_band_freq)])
print(paste("Most frequent age band:", most_frequent_age_band))

# Identify rare categories (frequency <= 2)
rare_categories <- names(age_band_freq[age_band_freq <= 2])
print(paste("Rare categories:", paste(rare_categories, collapse = ", ")))
```

2.4.5 Handling Missing or Unexpected Values

Frequency tables are also useful for spotting missing or unexpected values. For instance:

```
# Check for missing or unusual values in Gender
gender_freq <- table(survey_data$Gender, useNA = "ifany")
print(gender_freq)
```

The `useNA` argument ensures that missing values (if any) are included in the table.

2.5 Step A5: Cross-Tabulation

Cross-tabulations, also known as contingency tables, are a powerful tool in data analysis for examining the relationship between two or more categorical variables. They provide a matrix of frequency counts that show how categories of one variable are distributed across categories of another. Cross-tabulations are widely used in

business analytics, such as comparing demographics (e.g., age and gender) or tracking customer behavior across segments.

This guide delves deeper into cross-tabulations using the `survey_data` data frame, illustrating how to create, interpret, and enhance these tables with additional insights.

2.5.1 When and Why to Use Cross Tabulations

Cross-tabulations are particularly useful when:

- You want to **compare distributions** of one variable across levels of another.
- You need to **detect patterns** or relationships between categorical variables.
- You aim to **summarize multivariate data** for clear communication and decision-making.
- You are preparing data for **visualization** (e.g., stacked bar charts or heatmaps).

For example, in a survey analysis, you might want to compare how age groups are distributed by gender.

2.5.2 Creating a Cross Tabulation

In R, the `table()` function is used to create cross-tabulations. For example, to examine the distribution of participants by `AgeBand` and `Gender`:

```
# Create a cross-tabulation of AgeBand and Gender
age_gender_table <- table(survey_data$AgeBand, survey_data$Gender)
print(age_gender_table)
```

The resulting table displays the frequency counts for each combination of `AgeBand` and `Gender`. For instance:

- The value at the intersection of `<21` and `Male` represents the number of participants who are under 21 and identify as male.
- The totals for each row or column provide marginal counts (see **Marginal Totals** below).

2.5.3 Adding Proportions to Cross Tabulations

Frequency counts can be converted to proportions to better understand relative distributions. The `prop.table()` function is used for this purpose.

Proportions for the Entire Table

```
# Proportions for the entire table
age_gender_prop <- prop.table(age_gender_table)
print(round(age_gender_prop * 100, 2))
```

Here, each value represents the percentage of the total participants in that specific category combination.

Row-Wise Proportions

To see proportions within each row (e.g., the percentage of each gender within an age band):

```
# Row-wise proportions
row_prop <- prop.table(age_gender_table, margin = 1)
print(round(row_prop * 100, 2))
```

This output helps answer questions like, "What percentage of individuals in the `31-40` age group identify as `Female`?"

Column-Wise Proportions

To see proportions within each column (e.g., the percentage of each age band within a gender):

```
# Column-wise proportions
col_prop <- prop.table(age_gender_table, margin = 2)
print(round(col_prop * 100, 2))
```

This output helps answer questions like, "What percentage of individuals identifying as `Male` are in the `41-50 age band`?"

2.5.4 Marginal Totals

Adding row and column totals to a cross-tabulation provides a comprehensive overview of the data. Use the `addmargins()` function to include these totals:

```
# Add row and column totals to the table
age_gender_with_totals <- addmargins(age_gender_table)
print(age_gender_with_totals)
```

The row totals show the distribution of participants across age bands, while the column totals display the distribution across gender categories.

2.5.5 Customizing Cross Tabulation Output

For clearer interpretation, you can customize the table's appearance using descriptive labels for rows and columns. Use `dimnames()` to update labels:

```
# Customize row and column names
dimnames(age_gender_table) <- list(
  AgeGroup = levels(survey_data$AgeBand),
  GenderCategory = levels(survey_data$Gender)
)
print(age_gender_table)
```

2.5.6 Detecting Relationships in Cross Tabulations

Cross-tabulations provide a foundation for detecting relationships between variables. While frequency counts and proportions are descriptive, statistical tests such as the **Chi-Square Test** can quantify the association between variables.

```
# Perform a Chi-Square Test of Independence
chi_test <- chisq.test(age_gender_table)
print(chi_test)
```

The test evaluates whether there is a significant association between `AgeBand` and `Gender`. A p-value less than 0.05 suggests that the relationship is statistically significant.

2.5.7 Summary

Cross-tabulations are a versatile tool for analyzing the relationship between categorical variables. Mastering cross-tabulations equips you with the ability to summarize and interpret categorical data effectively, a crucial skill in business research and data analysis. In R, you can:

- Use `table()` to create cross-tabulations.
- Add proportions with `prop.table()` for a deeper understanding of relative distributions.
- Include totals with `addmargins()` for a complete summary.
- Customize, export, or extend your analysis with statistical tests.

2.6 Step A6: Saving Frequency Tables and Cross-Tabulations

When analyzing data, frequency tables and cross-tabulations provide valuable insights into categorical variables and their relationships. To share or store these tables for reporting or further analysis, you can save them as **CSV files** or directly export them in a human-readable format. This guide demonstrates both methods, using `survey_data` for examples.

2.6.1 Saving Frequency Tables

Frequency tables summarize the distribution of a single categorical variable, and you can save them as a CSV file or export them as plain text.

Saving as a CSV File

To save a frequency table as a CSV file, first convert it into a data frame using `as.data.frame()`:

```
# Create a frequency table for AgeBand
age_band_freq <- table(survey_data$AgeBand)
```

```r
# Convert the frequency table to a data frame
age_band_df <- as.data.frame(age_band_freq)

# Save the data frame to a CSV file
write.csv(age_band_df, "output/tables/age_band_frequency.csv", row.names = FALSE)
```

This saves the frequency table as a CSV file named age_band_frequency.csv. Each row in the CSV represents a category and its corresponding frequency.

Exporting to Plain Text

For a more human-readable format, you can save the table directly as plain text using `capture.output()`:

```r
# Export the frequency table to a text file
capture.output(age_band_freq, file = "output/tables/age_band_frequency.txt")
```

The output file, `age_band_frequency.txt`, contains the frequency table as it appears in the R console.

2.6.2 Saving Cross-Tabulations

Cross-tabulations summarize relationships between two categorical variables. Saving them involves similar steps to frequency tables, with additional considerations for row and column labels.

Saving as a CSV File

Like frequency tables, cross-tabulations can be converted into a data frame for CSV export:

```r
# Create a cross-tabulation of AgeBand and Gender
age_gender_table <- table(survey_data$AgeBand, survey_data$Gender)

# Convert the cross-tabulation to a data frame
age_gender_df <- as.data.frame(age_gender_table)

# Save the cross-tabulation to a CSV file
write.csv(age_gender_df, "output/tables/age_gender_cross_tabulation.csv", row.names = FALSE)
```

This saves the cross-tabulation as a CSV file named `age_gender_cross_tabulation.csv`. Each row in the CSV represents a unique combination of `AgeBand` and `Gender`, along with its frequency.

Exporting to Plain Text

To save a formatted version of the cross-tabulation, use `capture.output()`:

```r
# Export the cross-tabulation to a text file
capture.output(age_gender_table, file = "output/tables/age_gender_cross_tabulation.txt")
```

The file `age_gender_cross_tabulation.txt` will display the table as it appears in the R console, retaining its tabular structure.

Adding Totals Before Export

Adding marginal totals to cross-tabulations can make them more informative before saving:

```r
# Add row and column totals
age_gender_with_totals <- addmargins(age_gender_table)

# Save the table with totals as plain text
capture.output(age_gender_with_totals, file = "output/tables/age_gender_with_totals.txt")
```

This ensures that row and column totals are included in the exported table for comprehensive analysis.

2.6.3 Saving Proportions

Proportions provide additional insights and can be saved in the same way as frequency counts. For example:

```r
# Calculate proportions for AgeBand
age_band_prop <- prop.table(age_band_freq)
```

```r
# Convert to a data frame and save as CSV
age_band_prop_df <- as.data.frame(age_band_prop)
write.csv(age_band_prop_df, "output/tables/age_band_proportions.csv", row.names = FALSE)
```

Similarly, cross-tabulation proportions can be saved:

```r
# Calculate proportions for the cross-tabulation
age_gender_prop <- prop.table(age_gender_table)

# Convert to a data frame and save as CSV
age_gender_prop_df <- as.data.frame(age_gender_prop)
write.csv(age_gender_prop_df, "output/tables/age_gender_proportions.csv", row.names = FALSE)
```

2.6.4 Summary

To save frequency tables and cross-tabulations for reporting or analysis:

- **As CSV files**: Use `as.data.frame()` and `write.csv()` to export structured data.
- **As plain text**: Use `capture.output()` for human-readable formats.
- **With proportions**: Save proportions using similar methods to enhance insights.
- **With totals**: Add marginal totals before exporting for a complete overview.

These techniques ensure your tables are accessible for external use, whether for sharing with colleagues, integrating into reports, or storing for reproducibility.

3 Exercise B: Reports and Tables

This exercise focuses on how to produce reports and tables from R using a larger data set.

3.1 Step B1: Setup

You can continue working with the same project or working directory (depending on which option you chose in Step A1). To get started with this exercise, please do the following:

1. Please **download the datatset** survey.csv and save it in the `data` folder within your project folder or working directory .
2. Create a new **R script**: Go to **File > New File > R Script**.
3. Save the script in the `scripts` folder within your project folder or working directory with an appropriate name, e.g., `Lab1_Exercise_B.R`.

3.2 Step B2: