Data Science for Sensory and Consumer Scientists

John Ennis, Julien Delarue, and Thierry Worch

2021-01-01

Contents

Preface

Welcome to the website for *Data Science for Sensory and Consumer Scientists*. This book being written in the open and is currently under development.

About the Authors

John Ennis ...

Julien Delarue ...

Thierry Worch ...

Introduction

Chapter 1

Introduction

Sensory and consumer science (SCS) is consider as a pillar of food science and technology and is useful to product development, quality control and market research. Most scientific and methodological advances in the field are applied to food. This book makes no exception as we chose a cookie formulation dataset as a main thread. However, SCS widely applies to many other consumer goods so are the content of this book and the principles set out below.

1.1 Core principles in Sensory and Consumer Science

1.1.1 Measuring and analyzing human responses

Sensory and consumer science aims at measuring and understanding consumers' sensory perceptions as well as the judgements, emotions and behaviors that may arise from these perceptions. SCS is thus primarily a science of measurement, although a very particular one that uses human beings and their senses as measuring instruments. In other words, sensory and consumer researchers measure and analyze human responses. To this end, SCS relies essentially on sensory evaluation which comprises a set of techniques that mostly derive from psychophysics and behavioral research. It uses psychological models to help separate signal from noise in collected data [ref O'Mahony, D.Ennis, others?]. Besides, sensory evaluation has developed its own methodological framework that includes most refined techniques for the accurate measurement of product sensory properties while minimizing the potentially biasing effects of brand identity and the influence of other external information on consumer perception [Lawless & Heymann, 2010]. A detailed description of sensory methods is beyond the scope of this book and many textbooks on sensory evaluation methods are available to

readers seeking more information. However, just to give a brief overview, it is worth remembering that sensory methods can be roughly divided into three categories, each of them bearing many variants: - Discrimination tests that aim at detecting subtle differences between two products. - Descriptive analysis (DA), also referred to as 'sensory profiling', aims at providing both qualitative and quantitative information about product sensory properties. - Hedonic tests. This category gathers affective tests that aim at measuring consumers' liking for the tested products or their preferences among a product set. Each of these test categories generates its own type of data and related statistical questions in relation to the objectives of the study. Typically, data from difference tests consist in series of correct/failed binary answers depending on whether judges successfully picked the odd sample(s) among a set of three or more samples. These are used to determine whether the number of correct choices is above the level expected by chance. Conventional descriptive analysis data consist in intensity scores given by each panelist to evaluated samples on a series of sensory attributes, hence resulting in a product x attribute x panelist dataset (Figure 1). Note that depending on the DA method, quantifying means other than intensity ratings can be used (ranks, frequency, etc.). Most frequently, each panelist evaluates all the samples in the product set. However, the use of balanced incomplete design can also be found when the experimenters aim to limit the number of samples evaluated by each subject. Eventually, hedonic test datasets consist in hedonic scores (ratings for consumers' degree of liking or preference ranks) given by each interviewed consumer to a series of products. As for DA, each consumer usually evaluates all the samples in the product set, but balanced incomplete designs are sometimes used too. In addition, some companies favor pure monadic evaluation of product (i.e. between-subject design or independent groups design) which obviously result in unrelated sample datasets. Sensory and consumer researchers also borrow methods from other fields, in particular from sociology and experimental psychology. Definitely a multidisciplinary area, SCS develops in many directions and reaches disciplines that range from genetics and physiology to social marketing, behavioral economics and computational neuroscience. So have diversified the types of data sensory and consumer scientists must deal with.

1.2 How should sensory and consumer scientists learn data science?

1.3 Caution: Don't that everybody does

1.4 Example projects

Chapter 2

What is Data Science?

In this chapter we explain what is data science.

2.1 History and Definition

Data science has been called the "sexiest job of the 21st century" by Harvard Business Review [insert DJ Patil reference], but what is it? As with all rapidly growing fields, the definition depends on who you ask. Before we give our definition, however, we provide a brief history for context.

To begin, we note that there was a movement in early computer science to call their field "data science." Chief among the advocates for this viewpoint was Peter Naur, winner of the 2005 Turing award. This viewpoint is detailed in the preface to his 1974 book, "Concise Survey of Computer Methods," where he states that data science is "the science of dealing with data, once they have been established." From his perspective, this is the purpose of computer science. This viewpoint is echoed in the statement, often attributed to Edsger Dijkstr, that "Computer science is no more about computers than astronomy is about telescopes."

Interestingly, a similar movement arose in statistics, starting in 1962 with John Tukey's statements that "Data analysis, and the parts of statistics which adhere to it, must … take on the characteristics of science rather than those of mathematics" and that "data analysis is intrinsically an empirical science." This movement culminated in 1997 when Jeff Wu proposed during his inaugural lecture, upon becoming the chair of the University of Michigan's statistics department, that statistics should be called data science.

These two movements came together in 2001 in William S. Cleveland's paper "Data Science: An Action Plan for Expanding the Technical Areas in the Field

of Statistics." In this highly influential monograph, Cleveland makes the key assertion that "The value of technical work is judged by the extent ot which it benefits the data analyst, either directly or indirectly."

[FOOTNOTE: It is worth noting that these two movements were connected by substantial work in the areas of statistical computing, knowledge discovery, and data mining, with important work contributed by Gregory Piatetsky-Shapiro, Usama Fayyad, and Padhraic Smyth among many others.]

Putting this history together, we provide our definition of **data science** as: The intersection of statistics, computer science, and industrial design. Accordingly, we use the following three definitions of these fields:

- **Statistics**: The branch of mathematics dealing with the collection, analysis, interpretation, and presentation of masses of numerical data.
- **Computer Science**: Computer science is the study of processes that interact with data and that can be represented as data in the form of programs.
- **Industrial Design**: The professional service of creating and developing concepts and specifications that optimize the function, value, and appearance of products and systems for the mutual benefit of both user and manufacturer.

Hence data science is the delivery of value through the collection, processing, analysis, and interpretation of data.

2.2 Data Scientific Workflow

A schematic of a data scientific workflow is shown in Figure 2.1. Each section is described in greater detail below.

2.2.1 Data Collection

2.2.1.1 Design

2.2.1.2 Execute

2.2.1.3 Import

2.2.2 Data Preparation

2.2.2.1 Inspect

Goal: Gain familiarity with the data Key Steps: Learn collection details Check data imported correctly Determine data types Ascertain consistency and validity

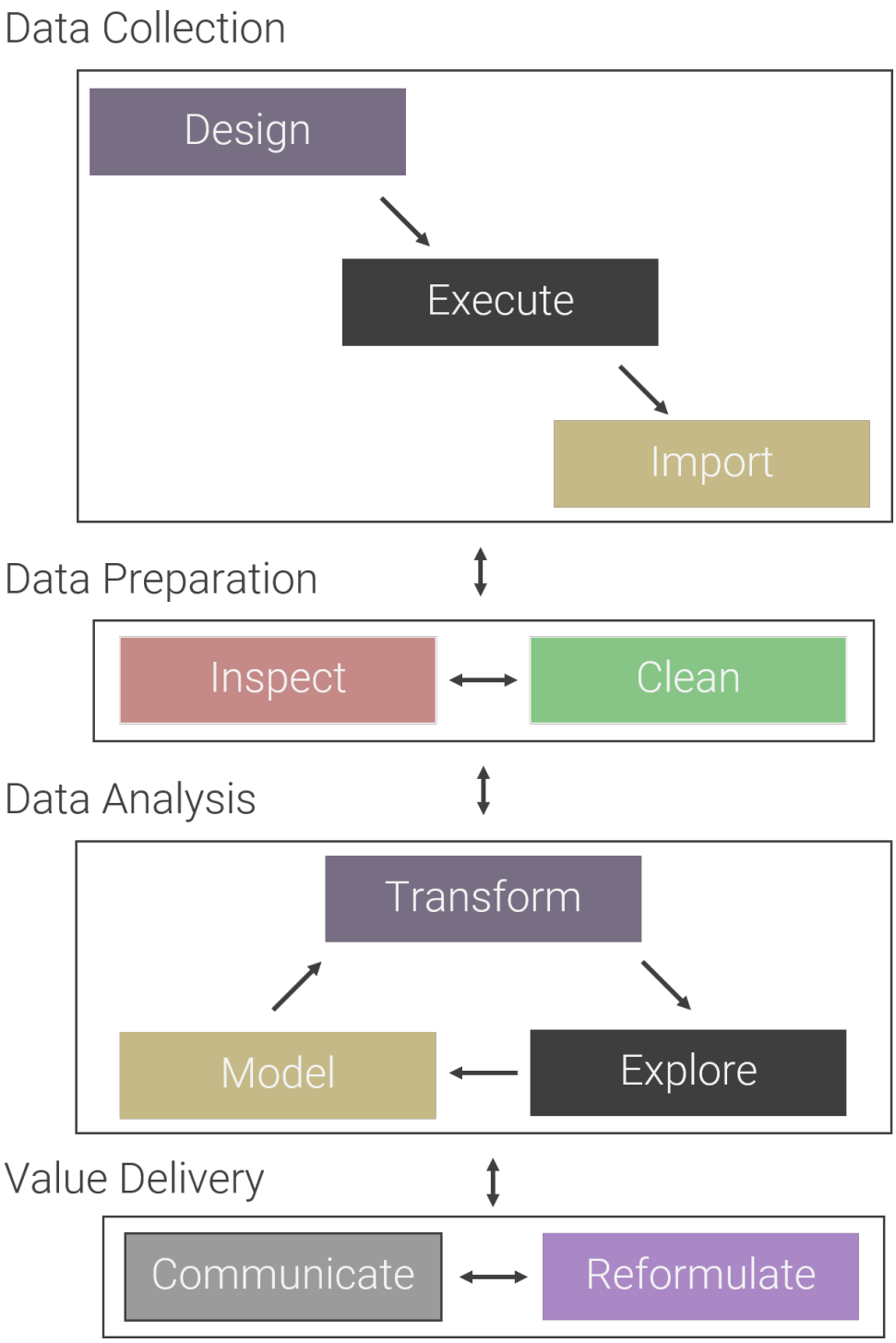

Figure 2.1: Data scientific workflow.

Tabulate and compute other basic summary statistics Create basic plots of key variables of interest

2.2.2.2 Clean

Goal: Prepare data for analysis Key Steps: Remove/correct errors Make data formatting consistent Organize text data Create tidy data (one observation per row) Organize data into related tables Document all choices

2.2.3 Data Analysis

2.2.3.1 Transform

Goal: Adjust data as needed for analysis Key Steps: Create secondary variables Decorrelate data Identify latent factors Engineer new features

2.2.3.2 Explore

Goal: Allow data to suggest hypotheses Key Steps: Graphical visualizations Exploratory analyses Note: Caution must be taken to avoid high false discovery rate when using automated tools

2.2.3.3 Model

Goal: Conduct formal statistical modeling Key Steps: Conduct traditional statistical modeling Build predictive models Note: This step may feed back into transform and explore

2.2.4 Value Delivery

2.2.4.1 Communicate

Goal: Exchange research information Key Steps: Automate reporting as much as possible Share insights Receive feedback Note: Design principles essential to make information accessible

2.2.4.2 Reformulate

Goal: Incorporate feedback into workflow Key Steps: Investigate new questions Revise communications Note: Reformulation make take us back to data cleaning

2.3 Reproducible Research

- Time savings
- Collaboration
- Continuous improvement

2.3.1 Principles

2.3.2 Tools

2.3.2.1 GitHub

2.3.2.2 R scripts

2.3.2.3 RMarkdown

2.3.2.4 Shiny

2.3.3 Documentation

2.3.4 Version control

2.3.5 Online repositories for team collaboration

2.3.6 Building a code base

2.3.6.1 Internal functions

2.3.6.2 Packages

2.4 Benefits of Data Science

2.4.1 Data-Driven Decision Making

2.4.2 Standardized Data Collection

2.4.3 Standardized Reporting

- Especially valuable when there are multiple sites globally

2.4.4 Improved Business Impact

2.5 How to Learn Data Science

Learning data science is much like learning a language or learning to play an instrument - you have to practice. Our advice based on mentoring many students and clients is to get started sooner rather than later, and to accept that the code you'll write in the future will always be better than the code you'll write today. Also, many of the small details that separate an proficient data scientist from a novice can only really be learned through practice as there are too many small details to learn them all in advice. So, starting today, do your best to write at least some code for all your projects. If a time deadline prevents you from completing the analysis in R, that's fine, but at least gain the experience of making an RStudio project and loading the data in R. Then, as time allows, try to duplicate your analyses in R, being quick to search for solutions when you run into errors. Often simply copying and pasting your error into a search engine will be enough to find the solution to your problem. Moreover, searching for solutions is its own skill that also requires practice. Finally, if you are really stuck, reach out to a colleague (or even the authors of this book) for help

2.6 How to Use This Book

We recommend following the instructions in Chapter 3 to get started.

2.7 Common Data Science Tools

We continue our discussion of getting started with R in the next chapter.

Chapter 3

Getting Started

3.1 R

R is an open-source programming language and software environment First released in 1995, R is an open-source implementation of S R was developed by Ross Ihaka and Robert Gentleman The name "R" is partly a play on Ihaka's and Gentleman's first names R is a scripting language (not a compiled language) Lines of R code run (mostly) in order R is currently the 7th most popular programming language in the world

3.1.1 Why Learn a Programming Language?

Control Speed Reduced errors Increased capability Continuous improvement Improved collaboration Reproducible results

3.1.2 Why R?

R originated as a statistical computing language It has a culture germane to sensory science R is well-supported with an active community Extensive online help is available Many books, courses, and other educational material exist The universe of available packages is vast R excels at data manipulation and results reporting R has more specialized tools for sensory analysis than other programming language

3.2 Why R?

For sensory and consumer scientists, we recommend the R ecosystem of tools for three main reasons. The first reason is cultural - R has from its inception been oriented more towards statistics than to computer science, making the feeling of programming in R more natural (in our experience) for sensory and consumer scientists than programming in Python. This opinion of experience is not to say that a sensory and consumer scientist shouldn't learn Python if they are so inclined, or even that Python tools aren't sometimes superior to R tools (in fact, they sometimes are). This latter point leads to our second reason, which is that R tools are typically better suited to sensory and consumer science than are Python tools. Even when Python tools are superior, the R tools are still sufficient for sensory and consumer science purposes, plus there are many custom packages such as SensR, SensoMineR, and FactorMineR that have been specifically developed for sensory and consumer science. Finally, the recent work by the RStudio company, and especially the exceptional work of Hadley Wickham, has lead to a very low barrier to entry for programming within R together with exceptional tools for data manipulation.

3.2.1 Steps to Install R

The first step in this journey is to install R. For this, visit The R Project for Statistical Computing. From there, follow the download instructions to install R for your particular platform.

https://cran.r-project.org/bin/windows/base/ Download the latest version of R Install R with default options You will almost certainly be running 64-bit R Note: If you are running R 4.0 or higher, you might need to install Rtools: https://cran.r-project.org/bin/windows/Rtools/

3.3 RStudio

3.3.1 Steps to Install RStudio

Next you need to install RStudio, which is our recommended integrated development environment (IDE) for developing R code. To do so, visit the RStudio desktop download page and follow the installation instructions.

Once you have installed R and RStudio, you should be able to open RStudio and enter the following into the Console to receive the number "3" as your output:

```
x <- 1
y <- 2
```

```
x + y
```

Some recommendations upon installing RStudio:

- Change the color scheme to dark.
- Put the console on the right.

https://www.rstudio.com/products/rstudio/download/#download Download and install the latest (almost certainly 64-bit) version of RStudio with default options Adjustments: Uncheck “Restore .RData into workspace at startup Select “Never” for “Save workspace to .RData on exit” Change color scheme to dark (e.g. “Idle Fingers”) Put console on right

3.3.2 Create a Local Project

Always work in an RStudio project Projects keep your files (and activity) organized Projects help manage your file path (so your computer can find things) Projects allow for more advanced capabilities later (like GitHub or renv) We cover the use of GitHub in a future webinar For now we create projects locally

3.3.3 Install and Load Packages

As you use R, you will want to make use of the many packages others (and perhaps you) have written Essential packages (or collections): tidyverse, readxl Custom Microsoft office document creation officer, flextable, rvg, openxlsx, extrafont, extrafontdb Sensory specific packages sensR , SensoMineR, FactoMineR, factoextra There are many more, for statistical tests of all varieties, to multivariate analysis, to machine learning, to text analysis, etc.

You only need to install each package once per R version To install a package, you can: Type install.packages(“[package name]”) Use the RStudio dropdown In addition, if a script loads package that are not installed, RStudio will prompt you to install the package Notes: If you do not have write access on your computer, you might need IT help to install packages You might need to safelist various R related tools and sites

3.3.4 Run Sample Code

Like any language, R is best learned first through example then through study We start with a series of examples to illustrate basic principles For this example, we analyze a series of Tetrad tests

Suppose you have 15 out of 44 correct in a Tetrad test Using sensR, it's easy to analyze these data:

```
library(sensR)

num_correct <- 15
num_total <- 44

discrim_res <- discrim(num_correct, num_total, method = "tetrad")

print(discrim_res)
```

3.4 Git and GitHub

Git is a version control system that allows you to revert to earlier versions of your code, if necessary. GitHub is service that allows for online backups of your code and which facilitates collaboration between team members. We highly recommend that you integrate both Git and GitHub into your data scientific workflow. For a full review of Git and GitHub from an R programming perspective, we recommend Happy Git with R by Jenny Bryant. In what follows, we simply provide the minimum information needed to get you up and running with Git and GitHub. Also, for an insightful discussion of the need for version control, please see [Cite bryan2018excuse].

3.4.1 Git

- Install Git
  - Windows
  - macOS
- Register with RStudio

3.4.2 GitHub

- Create a GitHub account
- Register with RStudio

3.5 RAW MATERIAL

3.5.1 Principles

3.5.2 Tools

3.5.2.1 GitHub

3.5.2.2 R scripts

3.5.2.3 RMarkdown

3.5.2.4 Shiny

3.5.3 Documentation

3.5.4 Version control

3.5.5 Online repositories for team collaboration

3.5.6 Building a code base

3.5.6.1 Internal functions

3.5.6.2 Packages

General Skills

Chapter 4

Tidy Thinking

In sensory science, different data collection tools (whether it is device, software, or methodologies) may provide data in different formats. Also, different statistical analyses may require having the same data but structured differently.

A simple example to illustrate this later point is the analysis of liking data. Let C consumers provide their hedonic assessments of P samples. To evaluate if samples have been liked differently, an ANOVA is performed on a long thin table with (PxC) rows x 3 columns (consumer, sample, and the liking scores). However, to assess whether consumers have the same preference patterns at the indidual level, internal preference mapping or cluster analysis would be performed, and both these analyses require as input a short and large table with P rows and C columns.

Another example of data manipulation consists in summarizing data, by for instance computing the mean by product for each sensory attribute (hence creating the so-called sensory profiles), or to generate frequency tables (e.g. proportions of male/female, distribution of the liking scores by sample, contingency table for CATA data, etc.)

For these reasons, it is hence essential to learn to manipulate data and transition from one structure to another one.

For illustration, let's consider some sensory data stored in *Sensory Profile.xlsx*, which consists in 9 panellists evaluating 11 samples on 32 sensory attributes.

We importing this data using the `read_xlsx()` function from the `{readxl}` package (for more information on how to import data, see Section 8).

A first analysis that is commonly done on such table consists in computing the mean per sample for each attribute, hence generating the so-called sensory profiles of the samples. Such table (crossing the samples in rows and the sensory attributes in columns) gives a first impression at the differences between samples across attributes.

The principles of data manipulation will be illustrated here by generating the sensory profiles from the raw data using different approaches. This step consists in reducing the 99x35 table into a 11x32 table, i.e. a table with 11 rows (one per sample) and 32 columns (one per attribute). Note that here, we consider the sample information as row names, not as an extra column in the dataset, else we would have a 11x33 table.

4.0.1 Using the `{tidyverse}`

Although the solutions we just presented are simple and only involve a few lines of code, we propose other solutions using the **{tidyverse}**. This will allow us introducing this philosophy of coding, and will provide you a first contact to some of the functions that we use. It will also show you that each situation can be solved in different ways, hence opening your mind and stimulating your creative imagination.

The **{tidyverse}** operates 5 major transformations on a dataset:

- `select()` allows selecting, renaming, and re-arranging columns of a dataset;
- `filter()` and `arrange()` works on the rows of the dataset by filtering data and rearranging the order;
- `mutate()` adds new columns, which can be the combination of other columns of the dataset;
- `summarise()` summarises the data by providing the statistics of interest on the variables selected (all the variables that are not specified are then removed).

Note that for `mutate()` and `summarise()`, different variant such as `mutate_all()` or `mutate_if()` (resp. `summarise_all()` and `summarise_if()`) allows applying the same transformation to multiple columns (all the columns for `mutate_all()` and `summarise_all()`, or the one that meet a pre-defined condition for `mutate_if()` and `summarise_if()`).

As a first proposition, we use `summarise_all`, meaning that we should only keep the variables that are relevant for the analysis. From `sensory`, we hence select the columns 3 (product), and the block starting from column 4 until the end of the dataset (sensory attributes). Since the mean needs to be computed for each sample separately, the function `group_by()` is used. This function ensures that the results are summarised by product in our case. We then summarise the data by performing the mean on all the variables (product is ignored since it is used to group the results). Ultimately, we use the product names as row names (using `column_to_rownames()`).

```
senso_mean2 <- sensory %>%
  dplyr::select(3,4:ncol(sensory)) %>%
  group_by(product) %>%
  summarise_all(list(~mean(.))) %>%
  column_to_rownames(var="product")
round(senso_mean2, 2)
```

The transformation from the original 11x35 table to the 11x33 table is done through the `group_by()` followed by `summarise_all` functions.

Another way of generating such table consists in not pre-selecting the variables, but in computing the mean only if the variable is numerical. To do so, we use `summarise_if()` and we put the condition that the variable should be numerical to compute the mean (otherwise R will return an error). Here again, we group the results by product.

```
senso_mean3 <- sensory %>%
  group_by(product) %>%
  summarise_if(is.numeric, mean) %>%
  column_to_rownames(var="product")
round(senso_mean3, 2)
```

This solution fits well if all the variables that are numeric should be summarized in the same way (here using the mean). Otherwise, a good solution is to run the analysis on a pre-defined set of variables. Such set can be created manually (we are taking the first 10 sensory attributes here), or automatically if the names of the attributes follow a certain pattern. In such case, the use of functions such as `starts_with()`, `ends_with()`, or using regular expression is of great help!

The mean table is then generated using the `summarise()` function `across()` `all_of()` the variables that were selected.

```
senso_var <- colnames(sensory)[4:13]
senso_mean4 <- sensory %>%
  group_by(product) %>%
  summarise(across(all_of(senso_var), mean)) %>%
  as.data.frame() %>%
  column_to_rownames(var="product")
round(senso_mean4, 2)
```

At last, we propose a solution which in this case is not optimal, but which can be very useful in some others. This solution consists in working on the dataset by permuting all the variables using `pivot_longer()`, hence generating a dataset with 3 relevant columns: one containing the products, one containing all the attributes, and one containing the scores. On this variable, we

`summarise()` the scores by product and by attribute, before re-structuring the data using `pivot_wider()`. Note that the combination `pivot_longer()` and `pivot_wider()` will re-organize automatically the attributes alphabetically. To avoid that, we can transform the column we name `attribute` into a factor with as level order the original order. This procedure maintains the original order.

```
senso_mean5 <- sensory %>%
  pivot_longer(4:ncol(.), names_to="attributes", values_to="scores") %>%
  mutate(attributes = factor(attributes, levels=colnames(sensory)[4:ncol(sensory)])) %
  group_by(product, attributes) %>%
  summarise(scores = mean(scores)) %>%
  pivot_wider(names_from=attributes, values_from=scores) %>%
  column_to_rownames(var="product")
round(senso_mean5, 2)
```

This procedure uses intermediate steps since the structure of the original table (99x35) is transformed by pivoting the attributes from columns to rows: this means that after `pivot_wider()`, the dataset created now contains 99x32=3168 rows and 5 columns (*judge*, *code*, and *product*, one column called *attributes* which contains the attribute names, and one column called *scores* which contain the individual scores). This table is then reduced to a table 11x32=352 rows and 3 columns (*product*, *attributes*, *scores*) through the `group_by()` and `summarise()` process, before being reset as 11x33 table through `pivot_wider()`.

This procedure could slightly be simplified: To generate `senso_mean5`, we computed the mean by `product` and `attributes` across all assessors. If we delete this line of code (i.e. related to `summary`), R will generate a table crossing `product` in rows, `attributes` in columns, in which each cell contains a list of values (here, we have as many values as we have assessors performing the test in each cell). When such situation appears, it is possible to apply automatically a function (here we want the `mean()`) on this sets of values using `values_fn` from `pivot_wider()`.

```
senso_mean6 <- sensory %>%
  pivot_longer(4:ncol(.), names_to="attribute", values_to="scores") %>%
  mutate(attribute = factor(attribute, levels=colnames(sensory)[4:ncol(sensory)])) %>%
  dplyr::select(-judge, -code) %>%
  group_by(product, attribute) %>%
  pivot_wider(names_from=attribute, values_from=scores, values_fn=mean) %>%
  column_to_rownames(var="product")
round(senso_mean6, 2)
```

In a similar way, missing values can be replaced automatically using `values_fill`.

As expected, all these solutions provide the same sensory profiles using different

process. Depending on the situations, some of these processes may be better than others.

Note that multiple analysis can be ordered together. Let's take back the example generating `senso_mean4`, and let's consider 2 other groups of variables, one in which we ask for the median, and one for which we ask the number of measures. This entire table can be generated as such:

```
senso_var1 <- colnames(sensory)[4:13]
senso_var2 <- colnames(sensory)[14:20]
senso_var3 <- colnames(sensory)[21:25]

senso_multifun <- sensory %>%
  group_by(product) %>%
  summarise(across(all_of(senso_var1), mean),
            across(all_of(senso_var2), median),
            across(all_of(senso_var3), length)) %>%
  as.data.frame() %>%
  column_to_rownames(var="product")
round(senso_multifun, 2)
```

Chapter 5

Data Manipulation

Chapter 6

Data Visualization

6.1 Principles

6.2 Table Mechanics

6.3 Chart Mechanics

6.4 Examples

6.5 Raw Material

6.6 Philosophy of ggplot2

Explain the principles of multi-layer graphs through an example.

`aes()`, `geom_()`, `theme()`

6.6.1 aesthetics

Provide the most relevant options for `aes()`

- x, y, z
- group
- color, fill
- text, label
- alpha, size

6.6.2 geom_

Explain the fact that some `geom_()` comes with some stats automatically (e.g. `geom_bar` bins the data)

6.7 Present the most useful graph

6.7.1 Scatter points

`geom_point()`

6.7.2 Line charts

`geom_line()`, `geom_smooth()` `geom_abline()` `geom_hline()` and `geom_vline()` `geom_segment()` and `geom_arrow()`

6.7.3 Bar charts

`geom_bar`, `geom_polygon`, `geom_histogram()`, `geom_freqpoly()` `position="identity"`, `position="dodge"` or `position="fill"`

6.7.4 Distribution

`geom_boxplot()` and `geom_violin()`

6.7.5 Text

`geom_text` and `geom_label` presentation of **{ggrepel}**

6.7.6 Rectangles

`geom_tile()`, `geom_rect`, and `geom_raster()`

6.7.7 Themes and legend

`theme()`, and pre-defined themes like `theme_bw()`, `theme_minimal()`, etc. `ggtitle()` `xlab()`, `ylab()`, or `labs()`

6.8 Interesting addition

6.8.1 Playing around with axes

`coord_fixed()`, `coord_cartesian()`, `coord_trans()` `scale_x_`, `scale_y_`

6.8.2 Transposing the plot

`coord_flip()` and `coord_polar()`

6.8.3 Splitting plots

`facet_wrap()`, `facet_grid()`

6.8.4 Combining plots

{patchwork}

Chapter 7

Automated Reporting

TUTORIAL TITLE Increasing Business Impact through Automated Reporting in R

TUTORIAL DESCRIPTION With this tutorial, we help sensory scientists increase their business impact by leveraging tools for automated report production in R.

ABSTRACT Effective communication of results is among the essential duties of the sensory scientist, but the sometimes tedious mechanics of report production together with the sheer volume of data that many scientists now must process combine to make reporting design an afterthought in too many cases. In this tutorial, we review recent advances in automated report production that liberate resources for scientists to focus on the interpretation and communication of results, while simultaneously reducing errors and increasing the consistency of their analyses. We teach the tutorial through an extended example, cumulatively building an R script that takes participates from receipt of an example dataset to a beautifully-designed and nearly completed PowerPoint presentation automatically and using freely available, open-source packages. Details of how to customize the final presentation to incorporate corporate branding - such as logos, font choices, and color palettes - will also be covered.

TOPICS COVERED INCLUDE - What does automated reporting mean in practice? - Scripting analyses, tables, and charts - Automated production of PowerPoint presentations - Building a "cookbook" of reporting recipes - Font choices and color palettes - Layering storytelling onto an automated report

SOFTWARE PACKAGES We use the R programming environment along with several freely available packages such as those that comprise the tidyverse, extrafont, officer, openxlsx, and RColorBrewer. Before the conference, we will provide a full list of packages and versions required to run the script created during the tutorial. Outline of Section

What is Automated Reporting? [Pull from presentation] Why Script? Save time Reduce errors Collaboration Share code with others Read own code later Explain choices for analysis, table presentation, charts Save steps for result creation Main tools R/RStudio RMarkdown, HTML output, etc. (mention but don't focus) Packages for Microsoft office production Officer suite (PowerPoint, Word) Charts, especially RVG Extract from Word/PowerPoint Index Flextable Images? Packages for formatting extrafont extrafontdb Rcolorbrewer

7.0.1 PowerPoint workflow

Start with template Explain slide master How to adjust choices Internal naming (relevant later) Example Title slides Tables Charts Bullet points Images Layout discussion

7.0.2 Word

Touch on but not in detail Shiny Make dashboards to help non-users (or self)

7.0.3 Excel

Although Excel is not our preferred tool for automated reporting, it is still one of the major ways to access and share data. Most data collection software offers the possibility to export data and/or results in an Excel format, and most data analysis tools accepts Excel format as inputs. With the large use of Excel, it is no surprise that many of our colleagues or clients like to share data and results using such spreadsheets. It is even less a surprise that R provides multiple solutions to import/export results from/to Excel.

For the importation of datasets, we have already presented two packages (**{xlsx}** and **{readxl}**) which provide some interesting options: they will not be detailed any further here.

For exporting results, two complementary packages (amongst others) in terms of ease of use and flexibility in the outcome are considered: **{xlsx}** and **{openxlsx}**.

Besides its option of importing directly Excel files (.xls and .xlsx), the package **{xlsx}** also offers the option to export tables in Excel through the function `write.xlsx()`. `write.xlsx()` works in the same way as `base::write.csv()` (as its name indicates, this function exports tables from R to a .csv file), but since it generates an Excel file (.xls or .xlsx), additional tabs can be added using the `append` option. The procedure for `write.xlsx()` is the following:

- First export the table `x=mydata1` of interest in a new Excel into the tab entitles "mytab1" of the file "myfile.xlsx" using the options

`file="myfile.xlsx"` and `sheetName= "mytab1"`, and set the option `append=FALSE`.

- Then add to `file="myfile.xlsx"` any additional table `x=mydata2` in another tab (e.g. `sheet="mytab2"`) using a similar command, but by setting `append=TRUE`. By doing so, R understands that "myfile.xlsx" should be extended with a second tab called "mytab2" that contains "mytable2". The tables exported may or may not include row names and column namess, and the sheet can automatically be password protected using the `password` option.

```
library(xlsx)

# We propose here to export the 5 first tables in 5 different tabs.
for (v in 1:5){

  if (v==1){
    append=FALSE
  } else {
    append=TRUE
  }

  xlsx::write.xlsx(as.data.frame(demog_freq[[v]]),
                   file="thierry_code/Output/Tables using xlsx.xlsx",
                   sheetName=names(demog_freq)[v],
                   row.names=FALSE, append=append)
}
```

The exportation of tables using the **{xlsx}** package is easy, yet very simplistic as it does not allow formatting the tables, nor does it allow exporting multiple tables in the same tab. For more advanced exporting options, the use of **{openxlsx}** package is preferred as it allows more flexibility in structuring and formatting the Excel output.

With **{openxlsx}**, the procedure starts with creating a workbook `wb` using the `createWorkbook()` function, to which we add worksheets through the `addWorksheet()` function. On a given worksheet, any table can be exported using `writeData()` or `writeDataTable()`, which controls where to write the table through the `startRow` and `startCol` options. Through these different functions, many additional formatting procedure can be applied:

- `createWorksheet()` allows:
  - show/hide grid lines using `gridLines`;
  - colour the specific sheet using `tabColour`;
  - change the zoom on the sheet through `zoom`;
  - show/hide the tab using `visible`;

  - format the worksheet by specifying its size (`paperSize`) and orientation (`orientation`).

- `writeData()` and `writeDataTable()` allow:

  - controlling where to print the data using `startRow` and `startCol` (or alternatively `xy`: `xy=c("B",12)` prints the table in cell B12), hence allowing exporting multiple tables within the same tab;
  - including the row names and column names through `rowNames` and `colNames`;
  - formatting the header using `headerStyle` (incl. colour of the text and/or background, font, font size, etc.) and whether a `filter` should be applied;
  - shaping the borders using predefined solutions through `borders`, or customizing them with `borderStyle` and `borderColour`;
  - adding a filter to the table using `withFilger`;
  - converts missing data to "#N/A" or any other string using `keepNA` and `na.string`.

- Additional formatting can be controlled using:

  - `options()` to predefine number formatting, border colours and style that will be applied automatically to each tables;
  - `modifyBaseFont()` to defined the font and font size;
  - `freezePane()` to freeze the first row and/or column of the table using `firstRow=TRUE` and `firstCol=TRUE`;
  - `createStyle()` to pre-define a style, or `addStyle()` to apply a later stage some styling to some cells;
  - controls the width of the columns using `setColWidths`;
  - `conditionalFormatting()` allows coloring cells when they meet predefined rules, as for instance to highlight significant p-values.

When using **{openxlsx}**, we recommend to use the same procedure as for Word and PowerPoint:

- Start with setting as default the parameters that should be applied to each table;
- Create styles for text or table headers that you save in different elements, and that you apply where needed.

In the following example, the sensory profiles are exported in the first tab, and a set of frequency tables are exported in the second tab. To introduce conditional formatting with **{openxlsx}**, the sensory profiles are color coded as following: For each cell, the value is compared to the overall mean computed for that column and is colored in red (resp. blue) if it's higher (resp. lower) than the mean. In practice, the color style is pre-defined in two parameters

called `pos_style` (red) and `neg_style` (blue) using `createStyle()`. The decision whether `pos_style` or `neg_style` should be used is defined by the `rule` parameter from the `conditionalFormatting()`[1] function.

```
library(openxlsx)

# Pre-define options to control the borders
options("openxlsx.borderColour"="#4F80BD")
options("openxlsx.borderStyle"="thin")

# Automatically set Number formats to 3 values after the decimal
options("openxlsx.numFmt"="0.000")

# Change the font to Calibri size 10
modifyBaseFont(wb, fontSize=10, fontName="Calibri")

# Create a header style in which
  # the text is centered and in bold,
  # borders on top and on the bottom,
  # a blue background is used.
headSty <- createStyle(fgFill="#DCE6F1", border="TopBottom",
                       halign="center", textDecoration="bold")

# Preparing the colouring for the conditional formatting
senso_mean <- sensory %>%
  group_by(product) %>%
  summarise_if(is.numeric, mean) %>%
  column_to_rownames(var="product")
overall_mean <- apply(senso_mean, 2, mean)

pos_style <- createStyle(fontColour="firebrick3", bgFill="mistyrose1")
neg_style <- createStyle(fontColour="navy", bgFill="lightsteelblue")

# Creation of the workbook wb with two tabs called Mean and Frequency
wb <- openxlsx::createWorkbook()
addWorksheet(wb, sheetName="Mean", gridLines=FALSE)
addWorksheet(wb, sheetName="Frequency", gridLines=FALSE)

# Exporting senso_mean to the first tab (first row and first column are frozen)
# A pre-defined Excel design called TableStyleLight9 is used for this table
freezePane(wb, sheet=1, firstRow=TRUE, firstCol=TRUE)
writeDataTable(wb, sheet=1, x=senso_mean,
               colNames=TRUE, rowNames=TRUE,
```

[1] In `conditionalFormatting()`, you can specify to which `rows` and `cols` the formatting applies. In this example, `cols` takes `v+1` because the first column contains the row names.

```
                tableStyle="TableStyleLight9")
for (v in 1:ncol(senso_mean1)){
  conditionalFormatting(wb, 1, cols=v+1, rows=(1:nrow(senso_mean1)+1),
                        rule = paste0(">",overall_mean[[v]]),
                        style = pos_style)
  conditionalFormatting(wb, 1, cols=v+1, rows=(1:nrow(senso_mean1)+1),
                        rule = paste0("<",overall_mean[[v]]),
                        style = neg_style)
}

# Here, we export many different frequency tables in the second tab by:
# 1. adding a title prior to the table
# 2. export the table under its title
# 3. title_r and next_r are used to track where the text is being printed
writeData(wb, 2, x="Frequency Tables", startRow=1, startCol=2)
title_r <- 3
for (v in 1:5){
  writeData(wb, 2, x=names(demog_freq)[v], startRow=max(title_r), startCol=1)
  writeData(wb, 2, x=demog_freq[[v]], startRow=max(title_r)+1, startCol=1,
            rowNames=FALSE, borders="surrounding", headerStyle=headSty)
  next_r <- max(title_r) + 1 + 1 + nrow(demog_freq[[v]]) + 1
  title_r <- c(title_r, next_r)
}

# We apply different style for the main title and for each table's title.
addStyle(wb, 2, style=createStyle(fontSize=14, textDecoration="bold"),
         rows=1, cols=2)
addStyle(wb, 2, style=createStyle(fontSize=12, textDecoration="italic"),
         rows=title_r, cols=rep(1,length(title_r)))

setColWidths(wb, 2, cols=1:10, widths=12)
```

The file is created using `saveWorkbook()` by specifying the name of the workbook `wb` and its path through `file`. In case such workbook already exists, it can be overwritten using `overwrite`. Additionally, the user can visualize the file so far created using `openXL()`.

For more details on using **{openxlsx}** see https://rdrr.io/cran/openxlsx/.

7.1 MORE RAW MATERIAL

```
library(tidyverse)
library(officer)
```

```
library(flextable)
```

7.2 PowerPint Slide Master

7.2.1 Importing the Template

```
my_file <- file.path("input","templates","Tutorial Template.pptx") %>%
                        read_pptx()

class(my_file) # checking if correct class

my_file %>%
  layout_summary()
```

7.2.2 Creating a PowerPoint Deck

```
pptx_obj <- read_pptx () # new empty file
pptx_obj <- pptx_obj %>%
  add_slide(layout  = 'Title and Content',master = "Office Theme")

pptx_obj %>% print(target = "output/first_example.pptx")
```

We can not load the themes of Office *ex nihilo* returns error

```
pptx_obj <- read_pptx() # new empty file
pptx_obj <- pptx_obj %>%
  add_slide(layout = "Title and Content", master = "Integral")
```

However, we can save an empty pptx with the desired theme and use it as a template

```
pptx_obj <- read_pptx(file.path("input","templates","integral.pptx"))
layout_summary(pptx_obj)
```

We can even load a template with more than one theme

```
pptx_obj <- read_pptx(file.path("input","templates","multmasters.pptx"))
layout_summary(pptx_obj)
```

7.2.3 Selection Pane

7.2.4 Key functions: read_pptx(path), layout_summary(x), layout_properties(x), add_slide(x, layout, master), on_slide(x, index), slide_summary(x, index = NULL)

7.2.5 Example Code

```
pptx_obj <- read_pptx() # new empty file

pptx_obj <- pptx_obj %>% # add slide
  add_slide(layout = "Title and Content", master = "Office Theme")

layout_summary(pptx_obj) # contains only basic layouts
layout_properties(pptx_obj) # additional detail

pptx_obj <- pptx_obj %>%
  on_slide(index = 1) # set active slide
slide_summary(pptx_obj) # slide is empty
```

7.3 Placeholders

7.3.1 Placeholders and Shapes

```
# Example 1
my_data <- c("My functions are:","ph_with","ph_location_type")
my_type <- "body"

pptx_obj <- read_pptx() # new empty file

pptx_obj <- pptx_obj %>%
  add_slide(layout = "Title and Content", master = "Office Theme")

pptx_obj <- pptx_obj %>%
  ph_with(value = my_data,location = ph_location_type(type = my_type))

pptx_obj %>%
  print(target = "output/test2.1.pptx")
```

```
# Example 2:
my_data <- head(mtcars)[,1:4]
my_type <- "body"

pptx_obj <- read_pptx() # new empty file
pptx_obj <- pptx_obj %>%
    add_slide(layout = "Title and Content", master = "Office Theme")

pptx_obj <- pptx_obj %>%
  ph_with(value = my_data, location = ph_location_type(type = my_type))

pptx_obj %>%
  print(target = "output/test2.2.pptx")
```

```
# Example 3
# We add a text box item in a custom position
# The same can be done for an image, logo, custom objects, etc.

my_data <- "My text"
my_type <- "body"

pptx_obj <- read_pptx() # new empty file
pptx_obj <- pptx_obj %>%
  add_slide(layout = "Title and Content", master = "Office Theme")

#ph_location is a subfunction which takes as argument
#left/top/width/height, units are inches

pptx_obj <- pptx_obj %>%
  ph_with(value = my_data, location = ph_location(left = 2, top = 2, width = 3, height = 1))

pptx_obj %>%
  print(target = "output/test2.3.pptx")
```

7.3.2 Key functions: ph_with()

7.4 Text

7.4.1 Working with Text

Each new text item added to a PowerPoint via officer is a paragraph object fpar() (“formatted paragraph”) creates this object

```
my_data <- fpar("My text")
my_type <- "body"

pptx_obj <- read_pptx() # new empty file
pptx_obj <- pptx_obj %>%
  add_slide(layout = "Title and Content", master = "Office Theme")

## Add paragraph
pptx_obj <- pptx_obj %>%
  ph_with(value = my_data, location = ph_location_type(type = my_type))

## Try to add a second paragraph
my_data2 <- fpar("My other text")
my_type <- "body"

pptx_obj <- pptx_obj %>%
  ph_with(value = my_data2, location = ph_location_type(type = my_type) )

pptx_obj %>%
  print(target = "output/test3.1.pptx")
## As we see, this code doesn't produce bullet points as we might hope
```

block_list() allows us to wrap multiple paragraphs together

```
my_data <- fpar("My text")
blank_line <- fpar("")
my_data2 <- fpar("My other text")

my_list <- block_list(my_data, blank_line, my_data2)

my_type <- "body"

pptx_obj <- read_pptx() # new empty file
pptx_obj <- pptx_obj %>%
  add_slide(layout = "Title and Content", master = "Office Theme") %>%
  ph_with(value = my_list, location = ph_location_type(type = my_type) )

pptx_obj %>%
  print(target = "output/test3.2.pptx")
```

Use ftext() ("formatted text") to edit the text before pasting into paragraphs. ftext() requires a second argument called prop which contains the formatting properties.

```
my_prop <- fp_text(color = "red", font.size = 16)
my_text <- ftext("Hello", prop = my_prop)

my_par <- fpar(my_text) ## formatted
blank_line <- fpar("")

my_par2 <- fpar("My other text") ## unformatted
my_list <- block_list(my_par,blank_line, my_par2)

pptx_obj <- read_pptx() # new empty file

pptx_obj <- pptx_obj %>%
  add_slide(layout = "Title and Content", master = "Office Theme") %>%
  ph_with(value = my_list, location = ph_location_type(type = my_type) )

pptx_obj %>%
  print(target = "output/test3.3.pptx")
```

7.4.2 Key functions: fpar(), ftext(), fp_text(), block_list()

7.4.3 Example Code

```
my_list <- block_list(
  fpar(ftext("Hello", prop = fp_text(color = "red", font.size = 16))) ,
  fpar(ftext("World", prop = fp_text(color = "blue", font.size = 14))) )

# The hierarchy is:
# block_list > fpar > ftext > fp_text

pptx_obj <- read_pptx() %>%
  add_slide(layout = "Title and Content", master = "Office Theme") %>%
  ph_with(value = block_list(
    fpar(ftext("Hello", prop = fp_text(color = "red", font.size = 16))) ,
    fpar(ftext("World", prop = fp_text(color = "blue", font.size = 14)))),
    ph_location_type(type = "body")
  ) %>%
  print(target = "output/test3.4.pptx")
```

7.5 Tables

7.5.1 Basic Code

```
my_data <- head(mtcars)

pptx_obj <- read_pptx() %>%
  add_slide(layout = "Title and Content", master = "Office Theme") %>%
  ph_with(value = head(mtcars), location = ph_location_type(type = "body")) %>%
  print(target = "output/test4.1.pptx")
```

7.5.2 Introduction to flextable

```
base_table <- tutorial_data[["Age"]] %>%
  table() %>%
  enframe(name = "Age", value = "Count") %>%
  mutate(Count = as.integer(Count)) %>%
  mutate(Percent = format_percent(Count / sum(Count), num_decimals = 2))

base_table
```

flextable: create attractive tables with predefined formatting. Use of %>% is recommended for readability

```
library(flextable)

ft_table <- base_table %>%
  flextable()

ft_table # see preliminary result in Viewer tab of RStudio
```

7.5.3 Demonstration

```
ft_table <- base_table %>%
  flextable() %>%
  autofit() %>% # straightforward, column width
  # ALIGNMENT
  # alignment of header: we use part argument
  align(align = "center", part = "header") %>%
```

```
  # alignment of content: we can use part = "body" or specify exact lines
  align(i = 1:nrow(base_table), j = 1:ncol(base_table), align = "center")

ft_table
```

Set font and characters

```
ft_table = ft_table %>%
# FONT AND CHARACTERS
# each command is independent, there are no nested functions as in officer
bold(i = 1, j = 1:3) %>% # first row, all cols
  italic(i = 3, j = ~Age+Count+Percent) %>% # first row, all cols, using ~ notation

  fontsize(i = 2, size = 16) %>%
  font(fontname = "Calibri") %>% # since no i or j are input, change is for all data
  font(fontname = "Roboto", part = "header") %>% #different font for header
  color(i = 3, j = 2, color = "red") %>%

  # WIDTH AND HEIGHT
  # all measurements are in inches
  width(j = 1, width = 4) %>% # column 1 wider
  height(i = 8, height = 0.5) %>% # row 8 change
  # CELL COLORS (background)
  bg(bg = "#0088FF", part = "header") %>% # a custom background for the header
  bg(i = 7:10, bg = "#C5C5C5") # a custom background for some cells

ft_table
```

Set borders

```
#BORDERS
# For borders we need to use nested functions (similar to fpar>ftext>fp_text)
#fp_border() is the second level function we will use to specify border"s characteristics
# as argument it takes color, style, and width

my_border <- fp_border(color = "black", style = "solid", width = 2)

# We use this second level function inside various main border functions
# border_outer()
# border_inner()
# border_inner_h()
# border_inner_v()
ft_table <- ft_table %>%
  border_outer(part = "all", border = my_border) %>% # using predefined border
```

```
  border_inner(part = "body", border = fp_border(style = "dashed"))

ft_table
```

7.5.4 Demonstration Output

7.5.5 Reformat original example

```
## ft parameters
header_background_color = a_green
body_background_color = a_cream
header_text_col = a_cream
body_text_col = a_dark_grey
total_text_col = a_red

border_solid <- fp_border(color = a_dark_grey, width = 2)
border_dashed <- fp_border(color = a_dark_grey, width = 1, style = "dashed")

ft <- summary_table %>%
  flextable() %>%
  font(fontname = font_name, part = "all") %>%
  fontsize(size = font_size, part = "all") %>%
  bold(part = "header") %>%
  border_remove() %>%
  border_outer(border = border_solid) %>%
  border_inner_v(border = border_solid, part = "header") %>%
  border_inner_h(border = border_dashed, part = "header") %>%
  border_inner_v(border = border_solid, part = "body") %>%
  border_inner_h(border = border_dashed, part = "body") %>%
  bg(bg = header_background_color, part = "header") %>%
  bg(bg = body_background_color, part = "body") %>%
  align(align = "center", part = "all") %>%
  color(color = header_text_col, part = "header") %>%
  color(color = body_text_col, part = "body") %>%
  color(color = total_text_col, part = "body", i = nrow(summary_table)) %>%
  italic(part = "body", i = nrow(summary_table)) %>%
  bold(part = "body", i = nrow(summary_table)) %>%
  autofit(add_w = 1)

ft
# Add table to slide
pptx_obj <- read_pptx() %>%
  add_slide(layout = "Title and Content", master = "Office Theme") %>%
```

```
  ph_with(value = ft, ph_location(left = 2, top = 2, width = 4 )) %>%
  print(target = "output/test4.7.pptx")
# positions are fixed. We can find exact positions to center the table
```

7.5.6 key functions: flextable(), align(), bold(), font(), color(), bg(), height() & width(), border_outer() & border_inner() & border_inner_h() & border_inner_v(), autofit()

Additional function to learn: merge(), compose() & as_chunk(), style()

fix_border_issues()

7.6 Charts

7.6.1 Adding charts as images

```
# We have preloaded a function to plot the chart.
# the function is using ggplot2 as plotting library

chart_to_plot <- sample_data_list[['Sample 1']] %>%
  make_jar_chart() # code to create a ggplot2 item, we will skip the contents
print(chart_to_plot) # see in Plots Window
```

7.6.1.1 rvg Example

```
# the output is a ggplot2 object

# To add this object as a rvg object on a slide, we will use the ph_with_vg
# ph_with_vg replaces the ph_with for a rvg object
# ph_with_vg_at allows to input a precise position for a chart, using the top/left we know alread
# all units are in inches
# argument code requires print(chart), argument type is a specific place on slide ("body" or othe


## IMPORTANT: ph_with_vg is is deprecated.
#old syntaxis ph_with_vg(code = print(chart_to_plot), type = "body")

pptx_obj <- read_pptx() %>%
```

```
  add_slide(layout = "Title and Content", master = "Office Theme") %>%
  ph_with(value = dml(ggobj = chart_to_plot), location  =  ph_location_type(type = 'boo
  print(target = "output/5.2.rvg1.pptx")

## IMPORTANT: ph_with_vg_at is is deprecated.
# old syntaxis ph_with_vg_at(code = print(chart_to_plot), left = 1, top = 1, width = 8

pptx_obj <- read_pptx() %>%
  add_slide(layout = "Title and Content", master = "Office Theme") %>%
  ph_with(value = dml(ggobj = chart_to_plot), location =  ph_location(left = 1, top = 1
  print(target = "output/5.2.rvg2.pptx")

# all items on the chart inside the pptx are now editable, just click on any and see
# the Shape Format tab in PowerPoint
```

7.6.2 mschart Package

```
# sample dataframe
mydata <- sample_data_list[['Sample 1']] %>%
  group_by(Variable,Response) %>% count()

# syntaxis is similar to ggplot2"s aes() with x,y,group
my_barchart <- ms_barchart(data = mydata, x = "Variable", y = "n", group = "Response")

# to add the object to a powerpoint slide we can use the officer"s native ph_with
pptx_obj <- read_pptx() %>%
  add_slide(layout = "Title and Content", master = "Office Theme") %>%
  ph_with(value = my_barchart, location = ph_location_type(type = "body")) %>%
  print(target = "output/5.3.msoffice.pptx")

# if we would open a rvg slide and an ms office slide and click on the slide
# for the rvg slide the only menu that appear is shape format
# While for the msoffice chart we have now the chart design option with all msoffice f
# by using chart_settings() functions one can customise in R the charts
```

7.7 Word

7.7.1 Word documents

Formats share similarities

body_add_*()

```
my_doc <- read_docx() %>%
  body_add_par(value = "My Text", style = "Normal") %>%
  body_add_par(value = "Other Text", style = "Normal") %>%
  body_add_par(value = "Conclusion", style = "Normal") %>%
  print(target = 'output/6.1.mydoc.docx')
```

body_add_break()

```
my_doc <- read_docx() %>%
  body_add_par(value = "My Text", style = "Normal") %>%
  body_add_break() %>%
  body_add_par(value = "Conclusion", style = "Normal") %>%
  print(target = 'output/6.2.mydoc.docx')
```

```
my_format <- fp_text(font.family = 'Calibri', font.size = 14, bold = TRUE, color = 'red')
my_text <- ftext('My dataset is:', my_format)
my_par <- fpar(my_text)

doc <- read_docx() %>%
  body_add_par(value = "Table of content", style = "heading 1") %>%
  body_add_par(value = "", style = "Normal") %>%
  body_add_fpar(my_par, style = "Normal") %>% #formatted paragraph function
  body_add_par(value = "", style = "Normal") %>%
  body_add_table(value = head(mtcars)[, 1:4], style = "table_template" ) %>%
  print(target = 'output/6.3.mydoc.docx')

read_docx() %>% styles_info()
```

Workflow in Practice

Chapter 8

Data Collection

8.1 Background

8.2 Other details

8.3 Importing Data to R

To analyze data, one need *data*. If this data is already available in R, then the analysis can be performed directly. However, in much cases, the data is stored outside the R environment, and needs to be imported.

In practice, the data might be stored in as many format as one can imagine, whether it ends up being a fairly common solution (.txt file, .csv file, or .xls/.xlsx file), or software specific (e.g. Stata, SPSS, etc.). Since it is very common to store the data in Excel spreadsheets (.xlsx) due to its simplicity, the emphasis is on this solution. Fortunately, most generalities presented for Excel files also apply to other formats through `base::read.table()` for .txt files, `base::read.csv()` and `base::read.csv2()` for .csv files, or through the `{read}` package (which is part of the `{tidyverse}`).

For other (less common) formats, the reader can find packages that would allow importing their files into R. Particular interest can be given to the package `{rio}` (*rio* stands for *R I*nput and *O*utput) which provides an easy solution that 1. can handle a large variety of files, 2. can actually guess the type of file it is, and 3. provides tools to import, export, and convert almost any type of data format, including .csv, .xls and .xlsx, or data from other statistical software such as SAS (.sas7bdat and .xpt), SPSS (.sav and .por), or Stata (.dta). As an alternative, the package `{foreign}` provides functions that allow importing

data stored from other statistical software (incl. Minitab, S, SAS, Stata, SPSS, etc.)

Although Excel is most likely one of the most popular way of storing data, there are no `{base}` functions that allow importing such files easily. Fortunately, many packages have been developed in that purpose, including `{XLConnect}`, `{xlsx}`, `{gdata}`, and `{readxl}`. Due to its convenience and speed of execution, we will be using `{readxl}` here.

8.3.1 Importing Structured Excel File

First, let's import the *Sensory Profile.xlsx* workbook using the `readxl::read_xlsx()` file, by informing as parameter the location of the file (informed in `file_path` using the package `{here}`) and the `sheet` we want to read from.

This file is called *structured* as all the relevant information is already stored in the same sheet in a structured way. In other words, no decoding is required here, and there are no 'unexpected' rows or columns (e.g. empty lines, or lines with additional information regarding the data but that is not data):

- The first row within the *Data* sheet of *Sensory Profile.xlsx* contains the headers,
- From the second row onwards, only data is being stored.

Since this data will be used for some analyses, it is assigned data to an R object called `sensory`.

To ensure that the importation went well, we print `sensory` to see how it looks like. Since `{readxl}` has been developed by Hadley Wickham and colleagues, its functions follow the `{tidyverse}` principles and the dataset thus imported is a `tibble`. Let's take advantage of the printing properties of a `tibble` to evaluate `sensory`:

```
sensory
```

`sensory` is a tibble with 99 rows and 35 columns that includes the `Judge` information (first column, defined as character), the `Product` information (second and third columns, defined as character), and the sensory attributes (fourth column onward, defined as numerical or `dbl`).

8.3.2 Importing Unstructured Excel File

In some cases, the dataset is not so well organized/structured, and may need to be *decoded*. This is the case for the workbook entitled *TFEQ.xlsx*. For this file:

- The variables' name have been coded and their corresponding names (together with some other valuable information we will be using in the next chapter) are stored in a different sheet entitled *Variables*;
- The different levels of each variable (including their code and corresponding names) are stored in another sheet entitled *Levels*.

To import and decode this dataset, multiple steps are required:

- Import the variables' name only;
- Import the information regarding the levels;
- Import the dataset without the first line of header, but by providing the correct names obtained in the first step;
- Decode each question (when needed) by replacing the numerical code by their corresponding labels.

Let's start with importing the variables' names from *TFEQ.xlsx* (sheet *Variables*)

In a similar way, let's import the information related to the levels of each variable, stored in the *Levels* sheet. A deeper look at the *Levels* sheet shows that only the coded names of the variables are available. In order to include the final names, `var_names` is joined (using `inner_join`).

```
library(tidyverse)
var_labels <- read_xlsx(file_path, sheet="Levels") %>%
  inner_join(dplyr::select(var_names, Code, Name), by=c(Question="Code"))

var_labels
```

Note: In some cases, this information is directly available in the dataset as subheader: A solution is then to import the first rows of the dataset that contain this information using the parameter `n_max` from 'readxl::read_xlsx". For each variable (when information is available), store that information as a list of tables that contains the code and their corresponding label.

Finally, the dataset (*Data*) is imported by substituting the coded names with their corresponding names. This process can be done by skipping reading the first row of the dataset that contains the coded header (`skip=1`), and by passing `Var_names` as header or column names (after ensuring that the names' sequence perfectly match across the two tables!).

The data has now the proper header, however each variable is still coded numerically. The steps to convert the numerical values with their corresponding labels is shown in Section 9.

Chapter 9

Data Preparation

The data we will be using in this chapter is the one that you imported in Section 8.

9.1 Data Inspection

To inspect the data, different steps can be used. First, since `read_xlsx()` returns a tibble, we can take advantage of its printing properties to get a fill of the data at hand,

```
TFEQ_data
```

Other informative solutions consists in printing a summary of the data through the `summary()` function, or looking at its type and first values using `str()`. However, due to its richness of the outputs, we prefer to use the `skim()` function from the `{skimr}` package.

```
library(skimr)
skim(TFEQ_data)
```

9.1.1 Data Type

In R, the variables can be of different types, going from numerical to nominal to binary etc. This section aims in presenting the most common types (and their properties) used in sensory and consumer studies, and in showing how to transform a variable from one type to another.

Remember that when your dataset is a tibble (as is the case here), the type of each variable is provided as sub-header when printed on screen. This eases the work of the analyst as the variables' type can be assessed at any moment.

In case the dataset is not in a tibble, the use of the `str()` function used previously becomes handy as it provides this information.

In sensory and consumer research, the four most common types are:

- Numerical (incl. integer or `int`, decimal or `dcl`, and double or `dbl`);
- Logical or `lgl`;
- Character or `char`;
- Factor or `fct`.

R still has plenty of other types, for more information please visit: https://tibble.tidyverse.org/articles/types.html

9.1.1.1 Numerical Data

Since a large proportion of the research done is quantitative, it is no surprise that our dataset are often dominated with numerical variables. In practice, numerical data includes integer (non-fractional number, e.g. 1, 2, -16, etc.), or decimal value (or double, e.g. 1.6, 2.333333, -3.2 etc.). By default, when reading data from an external file, R converts any numerical variables to integer unless decimal points are detected, in which case it is converted into double.

Do we want to show how to format R wrt the number of decimals? (e.g. options(digits=2))

9.1.1.2 Binary Data

Another common type that seem to be numerical in appearance, but that has additional properties is the binary type. Binary data is data that takes two possible values (`TRUE` or `FALSE`), and are often the results of a *test* (e.g. is `x>3`? Or is `MyVar` numerical?). A typical example of binary data in sensory and consumer research is data collected through Check-All-That-Apply (CATA) questionnaires.

Note: Intrinsically, binary data is *numerical*, TRUE being assimilated to 1, FALSE to 0. If multiple tests are being performed, it is possible to sum the number of tests that pass using the `sum()` function, as shown in the simple example below:

```
set.seed(123456)
# Generating 10 random values between 1 and 10 using the uniform distribution
x <- runif(10, 1, 10)
```

```
x

# Test whether the values generated are strictly larger than 5
test <- x>5
test

# Counting the number of values strictly larger than 5
sum(test)
```

9.1.1.3 Nominal Data

Nominal data is any data that is not numerical. In most cases, nominal data are defined through text, or strings. It can appear in some situations that nominal variables are still defined with numbers although they do not have a numerical meaning. This is for instance the case when the respondents or samples are identified through numerical codes: In that case, it is clear that respondent 2 is not twice larger than respondent 1 for instance. But since the software cannot guess that those numbers are *identifiers* rather than *numbers*, the variables should be declared as nominal. The procedure explaining how to convert the type of the variables will be explained in the next section.

For nominal data, two particular types of data are of interest:

- Character or `char`;
- Factor or `fct`.

Variables defined as character or factor take strings as input. However, these two types differ in terms of structure of their levels:

- For `character`, there are no particular structure, and the variables can take any values (e.g. open-ended question);
- For `factor`, the inputs of the variables are structured into `levels`.

To evaluate the number of levels, different procedure are required:

- For `character`, one should count the number of unique element using `length()` and `unique()`;
- For `factor`, the levels and the number of levels are direcly provided by `levels()` and `nlevels()`.

Let's compare a variable set as `factor` and `character` by using the `Judge` column from `TFEQ_data`:

```
example <- TFEQ_data %>%
  dplyr::select(Judge) %>%
  mutate(Judge_fct = as.factor(Judge))
summary(example)

unique(example$Judge)
length(unique(example$Judge))

levels(example$Judge_fct)
nlevels(example$Judge_fct)
```

Although `Judge` and `Judge_fct` look the same, they are structurally different, and those differences play an important role that one should consider when running certain analyses, or building tables and graphs.

When set as `character`, the number of levels of a variable is directly read from the data, and its levels' order would either match the way they appear in the data, or are ordered alphabetically. This means that any data collected using a structured scale will lose its natural order.

When set as `factor`, the number and order of the factor levels are informed, and does not depend on the data itself: If a level has never been selected, or if certain groups have been filtered, this information is still present in the data.

To illustrate this, let's re-arrange the levels from `Judge_fct` by ordering them numerically in such a way `J2` follows `J1` rather than `J10`.

```
judge <- str_sort(levels(example$Judge_fct),numeric=TRUE)
judge
levels(example$Judge_fct) <- judge
```

Now the levels are sorted, let's 'remove' some respondents by only keeping the 20 first ones (J1 to J20, as J18 does not exist), and re-run the previous code:

```
example <- TFEQ_data %>%
  dplyr::select(Judge) %>%
  mutate(Judge_fct = as.factor(Judge)) %>%
  filter(Judge %in% paste0("J",1:20))
dim(example)

unique(example$Judge)
length(unique(example$Judge))

levels(example$Judge_fct)
nlevels(example$Judge_fct)
```

After filtering some respondents, it can be noticed that the variable set as character only contains 19 elements, whereas the column set as factor still contains the 107 respondents (most of them not having any recordings). This property can be seen as an advantage or a disadvantage depending on the situation:

- For frequencies, it may be relevant to remember all the options, including the ones that may never be selected, and to order the results logically (use of `factor`).
- For hypothesis testing (e.g. ANOVA) on subset of data (e.g. the data being split by gender), the `Judge` variable set as `character` would have the correct number of degrees of freedom (18 in our example) whereas the variable set as factor would use 106 degrees of freedom in all cases!

The latter point is particularly critical since the analysis is incorrect and will either return an error or worse return erroneous results!

Last but not least, variables defined as factor allow having their levels being renamed (and eventually combined) very easily. Let's consider the `Living area` variable from `TFEQ_data` as example. From the original excel file, it can be seen that it has three levels, `1` corresponding to *urban area*, `2` to *rurban area*, and `3` to *rural area*. Let's start by renaming this variable accordingly:

```
example = TFEQ_data %>%
  mutate(Area = factor(`Living area`, levels=c(1,2,3), labels=c("Urban", "Rurban", "Rural")))

levels(example$Area)
nlevels(example$Area)

table(example$`Living area`, example$Area)
```

As can be seen, the variable `Area` is the factor version (including its labels) of `Living area`. If we would also consider that `Rurban` should be combined with `Rural`, and that `Rural` should appear before `Urban`, we can simply modify the code as such:

```
example = TFEQ_data %>%
  mutate(Area = factor(`Living area`, levels=c(2,3,1), labels=c("Rural", "Rural", "Urban")))

levels(example$Area)
nlevels(example$Area)

table(example$`Living area`, example$Area)
```

This approach of renaming and re-ordering factor levels is very important as it simplifies the readability of tables and figures. Some other transformations can

be applied to factors thanks to the `{forcats}` package. Particular attention can be given to the following functions:

- `fct_reorder`/`fct_reorder2` and `fct_relevel` reorder the levels of a factor;
- `fct_recode` renames the factor levels (as an alternative to `factor` used in the previous example);
- `fct_collapse` and `fct_lump` aggregate different levels together (`fct_lump` regroups automatically all the rare levels).

Although it hasn't been done here, manipulating strings is also possible through the `{stringr}` package, which provides interesting functions such as:

- `str_to_upper`/`str_to_lower` to convert strings to uppercase or lowercase;
- `str_c`, `str_sub` combine or subset strings;
- `str_trim` and `str_squish` remove white spaces;
- `str_extract`, `str_replace`, `str_split` extract, replace, or split strings or part of the strings.

9.1.2 Converting to Other Types

When importing data, variables may not always be associated to the right type. For instance, when respondents or products are numerically coded, they will be defined as integers rather than strings. Additionally, each variable type has its own property. To take full advantage of the different variable types, and to avoid wrong analyses (e.g considering a variable that is numerically coded as numeric when it is not), we need to convert them to other types.

In the following sections, we will `mutate()` a variable to create a new variable that corresponds to the original one after being converted to its new type (as in the previous example with `Area`). In case we want to overwrite a variable by only changing the type, the same name is used within `mutate()`.

Based on our variable types of interest, there are two main conversions to run: - From numerical to character/factor; - From character/factor to numerical.

The conversion from numerical to character or factor is simply done using `as.character()` and `as.factor()` respectively. Note however that `as.factor()` only converts into factors without allowing to chose the order of the levels, nor to rename them. Alternatively, the use of `factor()` allows specifying the `levels` (and hence the order of the levels) and their corresponding `labels`. An example in the use of `as.character()` and `as.factor()` was provided in the previous section when we converted the `Respondent` variables to character and factor. The use of `factor()` was also used earlier when the

variable `Living area` was converted from numerical to factor (called `Area`) with labels.

To illustrate the following points, let's start with creating a tibble with two variables, one containing strings made of numbers, and one containing strings made of text.

```
example <- tibble(Numbers = c("2","4","9","6","8","12","10"),
                  Text = c("Data","Science","4","Sensory","and","Consumer","Research"))
```

The conversion from character to numerical is straight forward and requires the use of the function `as.numeric()`:

```
example %>%
  mutate(NumbersN = as.numeric(Numbers), TextN = as.numeric(Text))
```

As can be seen, when strings are made of numbers, the conversion works fine. However, the text are not converted properly and returns NAs.

Now let's apply the same principle to a variable of the type factor. To do so, we will take the same example but first convert the variables from character to factor:

```
example <- example %>%
  mutate(Numbers = as.factor(Numbers)) %>%
  mutate(Text = factor(Text, levels=c("Data","Science","4","Sensory","and","Consumer","Research")))
```

Let's apply as.numeric() to these variables:

```
example %>%
  mutate(NumbersN = as.numeric(Numbers), TextN = as.numeric(Text))
```

We can notice here that the outcome is not really as expected as the numbers 2-4-9-6-8-12-10 becomes 3-4-7-5-6-2-1, and Data-Science-4-Sensory-and-Consumer-Research becomes 1-2-3-4-5-6-7. The rationale behind this conversion is that the numbers do not reflects the string itself, but the position of that level within the factor level structure.

To convert properly numerical factor levels to number, the variable should first be converted as character:

```
example %>%
  mutate(Numbers = as.numeric(as.character(Numbers)))
```

9.2 Data Organization

Presentation of the different shapes of the tables based on objectives

9.3 Data Manipulation

9.3.1 Type of table

matrix, data frame, and tibble.

how to check the type? `class()` how to test it? `is.data.frame()`, `is.matrix()`, `is_tibble()` how to convert it to another format? (see below)

Note on `{FactoMineR}` and `{SensoMineR}` which require data frames or matrix (not tibble) so introduction to `column_to_rownames()` and `rownames_to_columns()` as well as `as.data.frame()` and `as_tibble()`.

9.3.2 Data organisation

`select()`, `filter()`, `arrange()` `mutate()`

9.3.3 Data re-structuration

`pivot_wider()` and `pivot_longer()` `full_join()`, `inner_join()`, `left_join()` and `right_join()`

`unnest_token()` from **{tidytext}**

9.4 Cleaning and Quality Assessment

9.4.1 Renaming

renaming columns using `rename()` or `select()` renaming elements using `factor()` and **{forcats}**

9.4.2 Recoding

Example of recoding (could be renaming, or replacing NAs, etc.) by combining `mutate()` and `ifelse()`

9.4.3 Handling Missing Values

Removing and replacing NAs

9.4.4 Quality Assessment

Graphics?

Chapter 10

Data Analysis

10.1 Transformation

10.2 Exploration

10.3 Modeling

Chapter 11

Value Delivery

11.1 Design principles

11.2 Scientific inquiry vs storytelling

11.3 Research reformulation

11.4 Interactive reporting

11.5 Excel

11.6 Word

11.7 PowerPoint

11.7.1 Charts

11.7.2 Tables

11.7.3 Bullet Points

11.7.4 Images

11.8 HTML

Additional Topics

Chapter 12

Machine Learning

12.1 Concepts and general workflow (training/test)

12.2 Unsupervised learning

12.2.1 Cluster analysis

12.2.2 Factor analysis

12.2.3 Principle components analysis

12.2.4 t-SNE

12.3 Semisupervised learning

12.3.1 PLS regression

12.4 Supervised learning

12.4.1 Regression

12.4.2 K-nearest neighbors

12.4.3 Decision trees

12.4.4 Black boxes

12.4.4.1 Random forests

12.4.4.2 SVMs

12.4.4.3 Neural networks

12.5 Predictive modeling

```
data <- readr::read_rds('data/masked_data.rds')
nrows <- max(summary(data$Class)) * 2

data_over <- ROSE::ROSE(Class ~ .,
                        data = data %>%
                          mutate(across(starts_with('D'), factor, levels = c(0, 1))),
                        N = nrows, seed = 1)$data

readr::write_rds(data_over, 'data/data_classification.rds')

readxl::read_excel('data/data_regression.xlsx') %>%
  select(-`...1`, -judge, -product, -(steak:V64), -`qtt.drink.(%)`) %>%
  rename(socio_professional = `socio-professional`) %>%
  readr::write_rds('data/data_regression.rds')
```

12.8.2 Classification Code

```
library(tidyverse)
library(tidymodels)


# Load data ---------------------------------------------------------------

data <- read_rds('data/data_classification.rds')

# Inspect the data --------------------------------------------------------

summary(data)

data <- data %>% select(-ID)

skimr::skim(data)

data %>%
  mutate(across(starts_with('D'), factor, levels = c(0, 1))) %>%
  GGally::ggpairs(aes(fill = Class))



# Split data for models ---------------------------------------------------

# Set test set aside
```

```
train_test_split <- initial_split(data)
train_test_split

train_set <- training(train_test_split)
test_set <- testing(train_test_split)

# Split set fot cross-validation
resampling <- vfold_cv(train_set, 10)
resampling


# Fit MARS model ------------------------------------------------------------

usemodels::use_earth(
  Class ~ .,
  data = train_set
  )

earth_recipe <-
  recipe(formula = Class ~ ., data = train_set) %>%
  step_novel(all_nominal(), -all_outcomes()) %>%
  step_dummy(all_nominal(), -all_outcomes()) %>%
  step_zv(all_predictors())

earth_spec <-
  mars(
    num_terms = tune(),
    prod_degree = tune(),
    prune_method = "none"
    ) %>%
  set_mode("classification") %>%
  set_engine("earth")

earth_workflow <-
  workflow() %>%
  add_recipe(earth_recipe) %>%
  add_model(earth_spec)

earth_grid <- tidyr::crossing(num_terms = 2 * (1:6), prod_degree = 1:2)
earth_grid

earth_tune <-
  tune_grid(
    earth_workflow,
    resamples = resampling,
```

```
    # Save predictions for further steps
    control = control_grid(save_pred = TRUE, verbose = TRUE),
    # Test parameters on a grid defined above
    grid = earth_grid
  )


# Check model performance ---------------------------------------------------

earth_tune %>% show_best(n = 10)
earth_tune %>% autoplot()

earth_predictions <- earth_tune %>%
  collect_predictions(parameters = select_best(., 'roc_auc')) %>%
  mutate(model = "MARS")

earth_predictions %>%
  roc_curve(Class, .pred_A) %>%
  autoplot()

earth_predictions %>%
  lift_curve(Class, .pred_A) %>%
  autoplot()

earth_predictions %>%
  pr_curve(Class, .pred_A) %>%
  autoplot()

earth_predictions %>%
  conf_mat(Class, .pred_class) %>%
  autoplot()


# Fit decision tree ---------------------------------------------------------

tree_recipe <-
  recipe(formula = Class ~ ., data = train_set) %>%
  step_novel(all_nominal(), -all_outcomes()) %>%
  step_zv(all_predictors())

tree_spec <-
  decision_tree(
    cost_complexity = tune(),
    tree_depth = tune(),
    min_n = tune()
```

```
    ) %>%
  set_mode("classification") %>%
  set_engine("rpart")

tree_workflow <-
  workflow() %>%
  add_recipe(tree_recipe) %>%
  add_model(tree_spec)

tree_tune <-
  tune_grid(
    tree_workflow,
    resamples = resampling,
    # Save predictions for further steps
    control = control_grid(save_pred = TRUE, verbose = TRUE),
    # Test 20 random combinations of parameters
    grid = 20
  )

# Check model performance ------------------------------------------------

tree_tune %>% show_best(n = 10)
tree_tune %>% autoplot()

tree_predictions <- tree_tune %>%
  collect_predictions(parameters = select_best(., 'roc_auc')) %>%
  mutate(model = "Decision Tree")

tree_predictions %>%
  bind_rows(earth_predictions) %>%
  group_by(model) %>%
  roc_curve(Class, .pred_A) %>%
  autoplot()

tree_predictions %>%
  bind_rows(earth_predictions) %>%
  group_by(model) %>%
  lift_curve(Class, .pred_A) %>%
  autoplot()

tree_predictions %>%
  bind_rows(earth_predictions) %>%
  group_by(model) %>%  pr_curve(Class, .pred_A) %>%
  autoplot()
```

```
tree_predictions %>%
  conf_mat(Class, .pred_class) %>%
  autoplot()


# Let's go with MARS model ------------------------------------------------

final_fit <- earth_workflow %>%
  finalize_workflow(select_best(earth_tune, 'roc_auc')) %>%
  last_fit(train_test_split)

final_fit %>% collect_metrics()

final_fit %>%
  collect_predictions() %>%
  roc_curve(Class, .pred_A) %>%
  autoplot()

final_model <- final_fit %>%
  pluck(".workflow", 1) %>%
  fit(data)

final_model %>%
  pull_workflow_fit() %>%
  vip::vip()

final_model %>%
  pull_workflow_fit() %>%
  pluck("fit") %>%
  summary

write_rds(final_model, 'classification_model.rds')


# Predict something -------------------------------------------------------

model <- read_rds('classification_model.rds')

new_observation <- tibble(
  N1 = 1.8,
  D1 = factor(0),
  D2 = factor(0),
  D3 = factor(1),
  D4 = factor(0),
  D5 = factor(1),
```

```
  D6 = factor(0),
  D7 = factor(1),
  D8 = factor(1),
  D9 = factor(1),
  D10 = factor(1),
  D11 = factor(0)
)

predict(model, new_observation, type = "class")
predict(model, new_observation, type = "prob")
```

12.8.3 Regression Code

```
library(tidyverse)
library(tidymodels)


# Load data ---------------------------------------------------------------

data <- read_rds('data/data_regression.rds')
glimpse(data)

# Inspect the data --------------------------------------------------------

summary(data)

skimr::skim(data)

# Split data for models ---------------------------------------------------

# Set test set aside
train_test_split <- initial_split(data)
train_test_split

train_set <- training(train_test_split)
test_set <- testing(train_test_split)

# Split set fot cross-validation
resampling <- vfold_cv(train_set, 10)
resampling


# Fit glmnet model --------------------------------------------------------
```

```
usemodels::use_glmnet(
  Liking ~ .,
  data = train_set
)

glmnet_recipe <-
  recipe(formula = Liking ~ ., data = train_set) %>%
  step_novel(all_nominal(), -all_outcomes()) %>%
  step_dummy(all_nominal(), -all_outcomes()) %>%
  step_zv(all_predictors()) %>%
  step_normalize(all_predictors(), -all_nominal())

glmnet_spec <-
  linear_reg(penalty = tune(), mixture = tune()) %>%
  set_mode("regression") %>%
  set_engine("glmnet")

glmnet_workflow <-
  workflow() %>%
  add_recipe(glmnet_recipe) %>%
  add_model(glmnet_spec)

glmnet_grid <- tidyr::crossing(penalty = 10^seq(-6, -1, length.out = 20),
                               mixture = c(0.05, 0.2, 0.4, 0.6, 0.8, 1))

glmnet_tune <-
  tune_grid(
    glmnet_workflow,
    resamples = resampling,
    # Save predictions for further steps
    control = control_grid(save_pred = TRUE, verbose = TRUE),
    # Test parameters on a grid defined above
    grid = glmnet_grid
    )

# Check model performance ------------------------------------------------

glmnet_tune %>% show_best(n = 10)
glmnet_tune %>% autoplot()

glmnet_predictions <- glmnet_tune %>%
  collect_predictions(parameters = select_best(., 'rmse')) %>%
  mutate(model = "GLMNet",
         .resid = Liking - .pred)
```

```
glmnet_predictions %>%
  ggplot(aes(sample = .resid)) +
  geom_qq() +
  geom_qq_line()

glmnet_predictions %>%
  ggplot(aes(.pred, Liking)) +
  geom_point() +
  geom_abline(slope = 1, intercept = 0)

glmnet_predictions %>%
  ggplot(aes(.pred, .resid)) +
  geom_point() +
  geom_hline(yintercept = 0)

ggplot(glmnet_predictions, aes(x = .resid)) +
  geom_histogram(aes(y =..density..), fill = 'white', color = 'black') +
  stat_function(fun = dnorm,
                args = list(mean = mean(glmnet_predictions$.resid),
                            sd = sd(glmnet_predictions$.resid)),
                size = 1)

# Fit random forest ------------------------------------------------------

rf_recipe <-
  recipe(formula = Liking ~ ., data = train_set) %>%
  step_novel(all_nominal(), -all_outcomes()) %>%
  step_zv(all_predictors())

rf_spec <-
  rand_forest(
    mtry = tune(),
    min_n = tune(),
    trees = 50
    ) %>%
  set_mode("regression") %>%
  set_engine("ranger", importance = "impurity")

rf_workflow <-
  workflow() %>%
  add_recipe(rf_recipe) %>%
  add_model(rf_spec)

rf_tune <-
  tune_grid(
```

```
    rf_workflow,
    resamples = resampling,
    # Save predictions for further steps
    control = control_grid(save_pred = TRUE, verbose = TRUE),
    # Test 20 random combinations of parameters
    grid = 20
  )

# Check model performance ------------------------------------------------

rf_tune %>% show_best(n = 10)
rf_tune %>% autoplot()

rf_predictions <- rf_tune %>%
  collect_predictions(parameters = select_best(., 'rmse')) %>%
  mutate(model = "Random Forest",
         .resid = Liking - .pred)

rf_predictions %>%
  bind_rows(glmnet_predictions) %>%
  ggplot(aes(sample = .resid)) +
  geom_qq() +
  geom_qq_line() +
  facet_wrap(~model)

rf_predictions %>%
  bind_rows(glmnet_predictions) %>%
  ggplot(aes(.pred, Liking)) +
  geom_point() +
  geom_abline(slope = 1, intercept = 0) +
  facet_wrap(~model)

rf_predictions %>%
  bind_rows(glmnet_predictions) %>%
  ggplot(aes(.pred, .resid)) +
  geom_point() +
  geom_hline(yintercept = 0) +
  facet_wrap(~model)

rf_predictions %>%
  ggplot(aes(x = .resid)) +
  geom_histogram(aes(y =..density..), fill = 'white', color = 'black') +
  stat_function(fun = dnorm,
                args = list(mean = mean(rf_predictions$.resid),
                            sd = sd(rf_predictions$.resid)),
```

```
                size = 1)

# Let's go with rf model ----------------------------------------------

final_fit <- glmnet_workflow %>%
  finalize_workflow(select_best(glmnet_tune, 'rmse')) %>%
  last_fit(train_test_split)

final_fit <- rf_workflow %>%
  finalize_workflow(select_best(rf_tune, 'rmse')) %>%
  last_fit(train_test_split)

final_fit %>% collect_metrics()

final_fit %>%
  collect_predictions() %>%
  mutate(.resid = Liking - .pred) %>%
  ggplot(aes(sample = .resid)) +
  geom_qq() +
  geom_qq_line()

final_model <-  final_fit %>%
  pluck(".workflow", 1) %>%
  fit(data)

final_model %>%
  pull_workflow_fit() %>%
  vip::vip()

# final_model %>%
#   broom::tidy() %>%
#   filter(estimate != 0)

write_rds(final_model, 'regression_model.rds')

# Predict something ---------------------------------------------------

model <- read_rds('regression_model.rds')

new_observations <- data[1:2,]
new_observations

predict(model, new_observations)
```

Chapter 13

Text Analysis

13.1 Data import

13.1.1 Data sources

13.1.2 Tokenizing

13.1.3 Lemmatization, stemming, and stop word removal

13.2 Analysis

13.2.1 Frequency counts and summary statistics

13.2.2 Word clouds

13.2.3 Contrast plots

13.2.4 Sentiment analysis

13.2.5 Bigrams and word graphs

13.3 RAW MATERIAL

Introduction to **{tidytext}** and **{Xplortext}**

13.3.1 Statistical entities

What are we considering as statistical entities?

- documents
- sentences
- words
- cleaned words

Depends on objectives of study and how data are being collected:

- directly from consumers in a CLT (directed questions)
- analysis of social media (e.g. twitter)
- web-scrapping from website

Discussion around CATA as a simplified version of text analysis…

13.3.1.1 Notion of tokenization

13.3.1.2 Cleaning the data

Notions of lemmatization, stemming, and stopwords removal

- grouping words
- removing stopwords
- tf-idf

13.3.2 Analysis of Frequencies and term-frequency document

13.3.2.1 Contingency table

Presentation of the tf/contingency table

13.3.2.2 wordclouds

{ggwordclouds}

13.3.2.3 Correspondence Analysis

{FactoMineR} and **{Xplortext}**

13.3.3 Futher Analysis of the words

13.3.3.1 Sentiment Analysis

Sentiment analysis and its relationship to hedonic statement Introduction to free-JAR?

13.3.3.2 Bi-grams and N-grams

Presentation of graph-theory applied to text mining

13.3.3.3 Machine learning

Introduction to machine learning associated to text mining

Chapter 14

Pipelines

Chapter 15

What's Next?

15.1 Shiny

15.2 Graph Databases

15.3 Sensory Analysis in R

15.4 Learning Resources

15.5 Python

References

Bryan, J. (2018). Happy git and github for the useR. GitHub. Available from https://happygitwithr.com/ Gillespie, C., & Lovelace, R. (2016). Efficient R programming: A practical guide to smarter programming. " O'Reilly Media, Inc.". Grolemund, G. Getting Started with R. Rstudio Support. Available from https://support.rstudio.com/hc/en-us/articles/201141096-Getting-Started-with-R Norman, D. (2013). The design of everyday things: Revised and expanded edition. Basic books. Peng, R., Caffo, B., & Leek, J. Data Science Specialization [Online course]. Available from https://www.coursera.org/specializations/jhu-data-science Peng, R., Caffo, B., & Leek, J. Executive data science specialization [Online course]. Available from https://www.coursera.org/specializations/executive-data-science Wickham, H., & Grolemund, G. (2016). R for data science: import, tidy, transform, visualize, and model data." O'Reilly Media, Inc.". Wickham, H. (2016). ggplot2: elegant graphics for data analysis. Springer. Zuur, A., Ieno, E. N., & Meesters, E. (2009). A Beginner's Guide to R. Springer Science & Business Media.

Breiman, L. (2001). Statistical modeling: The two cultures (with comments and a rejoinder by the author). Statistical science, 16(3), 199-231. Cleveland, W. S. (2001). Data science: an action plan for expanding the technical areas of the field of statistics. International statistical review, 69(1), 21-26. Fayyad, U., Piatetsky-Shapiro, G., & Smyth, P. (1996). From data mining to knowledge discovery in databases. AI magazine, 17(3), 37-37. Naur, P. (1966). The science of datalogy. Communications of the ACM, 9(7), 485. Tukey, J. W. (1962). The future of data analysis. The annals of mathematical statistics, 33(1), 1-67. Tukey, J. W. (1977). Exploratory data analysis. Reading, Mass: Addison-Wesley Pub. Co. Wu, C. F. J. (1997) "Statistics = Data Science?" Lecture notes available online at http://www2.isye.gatech.edu/~jeffwu/presentations/datascience.pdf