Data Science for Sensory and Consumer Scientists

John Ennis, Julien Delarue, and Thierry Worch

2022-05-11

Contents

Digestifs   229

Preface

Welcome to the website for *Data Science for Sensory and Consumer Scientists*, a book in development and under contract for CRC Press.

Who Should Read This Book?

This book is for practitioners or students of sensory and consumer science who
want to participate in the emerging field of computational sensory science. This

book assumes little to no coding experience. Some statistical experience will be helpful to understand the examples discussed.

How to Use This Book

This book is meant to be interactive, with the reader ideally typing and running all of the code presented in the parts labeled "Hors d'Oeuvres" and "Bon Appétit." Computer languages are like human languages in that they need to be practiced to be learned, so we highly recommend the reader actually typing all of the code from these parts, running it, and verifying they receive the results shown in the book. To help with this practice, we have created a special GitHub repository that contains folders called `work_along` and `sample_code`. Please see Appendix 2 for guidance on how to get started with R and GitHub. In the the `sample_code` folder, we have included the code as presented in the book, while in the `work_along` folder we have provided blank files for each chapter that load the libraries[1].

Acknowledgements

---

[1]You may need to load these libraries before the code will run, please see Appendix 2 for more information on this topic as well.

Chapter 1

Bienvenue!

Why Data Science for Sensory and Consumer Science?

Located at the crossroads of biology, social science, and design, sensory and consumer science (SCS) is definitely the tastiest of all sciences. On the menu is a wide diversity of products and experiences that sensory and consumer scientists approach through the lens of human senses. Thanks to a wide set of refined methods, they have access to rich, complex and mouthwatering data. Delightedly, data science empowers them and leads them to explore new flavorful territories.

Core principles in Sensory and Consumer Science

Sensory and consumer science is considered 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.

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,

11

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 and 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 products.
- Descriptive analysis (DA), also referred to as 'sensory profiling', aims at providing both qualitative and quantitative information about product sensory properties.
- Affective tests. This category includes hedonic tests that aim at measuring consumers' liking for the tested products or their preferences among a product set.

Each test category generates its own type of data and related statistical questions in relation to the objectives of the study. Typically, data from difference tests with forced-choice procedures (e.g. triangle test, duo-trio, 2-AFC, etc.) consist in series of binary answers (correct/failed) depending on whether judges successfully picked the odd sample(s) among a set of three or more samples [1]. These data 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, counts, 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.

---

[1] Other procedures like the *different from control* test or the *degree of difference* test generate rating data

Eventually, datasets from hedonic tests consist of hedonic scores (i.e. degrees 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. As in many scientific fields, the development of sophisticated statistical techniques and access to powerful data analysis tools have played an important role in the evolution of sensory & consumer science. Statisticians and data analysts in SCS have developed their own field of research, coined Sensometrics (Schlich (1993), Brockhoff (2011) ,Qannari (2017)). Now then, what makes sensory & consumer science special? And how does it influence the way sensory and consumer data are handled?

Dealing with human diversity

Sensory evaluation attempts to isolate the sensory properties of foods and provides important and useful information about these properties to product developers, food scientists, and managers (Lawless and Heymann (2010)). However, one should bear in mind that these 'sensory properties' actually result from the interaction between the object (the food) and the perceiver of that object (the consumer). In fact, we may very well consider the true object of evaluation in SCS to be mental representations. They are nonetheless very concrete and directly impact behaviors, health and economic decisions (Kahneman and Tversky (2000)). A direct consequence of this is that sensory data depend both on the product to be evaluated and on the subjects who evaluate the product. Because people are different, sensory data are expected to differ accordingly. In its core principle, SCS recognizes the diversity of human beings, biologically, socially and culturally speaking, not to mention the fact that each individual has their own personal history and experience with products. In short, people perceive things differently and like different things. For this reason, SCS only relies on groups of people (i.e. a panel of judges, a sample of consumers) and never on a single individual's response. Yet, sensory and consumer scientists usually collect individual data and analyze them at a refined level of granularity (individual, subgroups) before considering larger groups (specific populations or marketing targets).

This said, sensory and consumer studies must lead to operational recommendations. They are used to make informed decisions on product development,

to launch new product, and sometimes to help define food and health policies. Data science can precisely help sensory and consumer scientists to reach those objectives while taking diversity into account.

For measures of sensory description, sensory and consumer scientists can manage the diversity of human responses to a certain extent by training panels to use a consensual vocabulary, by aligning evaluated concepts and calibrating the quantification of evaluations on scales (Bleibaum (2020)). However, this won't eliminate interindividual differences in sensitivity, simply because we are genetically determined to be different on top of differences due to age, physiological state, illness, etc. Nowadays, as the field becomes more and more consumer-oriented, it becomes clear that the use of several subjects in a panel cannot be assimilated to a mere repetition of individual measurements. Accordingly, sensory methods have been developed to allow panelists to better express their own perceptions and to get a more accurate picture of how people perceive products (Varela and Ares (2012)). These methods yield richer and more complex data that require more advanced analysis techniques to extract relevant and actionable information. Historically, one the first methodological innovations in this direction has been the use of free choice profiling combined to Generalized Procrustes Analysis (Williams and Langron (1984)). Since then, sensory and data analysis methods have multiplied greatly (Delarue et al. (2015)). Naturally, data science has become even more crucial to accompany this evolution.

As regards hedonic tests (liking, acceptance, preference...), the measurements are in essence subjective and participants to such tests are by definition 'untrained' consumers. A constant outcome of these tests is to find important interindividual differences and it is very common to find consumers who have opposite sensory preference patterns. Clustering and segmentation techniques are thus routinely applied to consumer data. One difficulty though, is to link these differences in preferences to other consumer variables, should they be sociodemographic, psychographic, or related to usage and attitudes. Most often, one set of variables is not enough to fully explain preference patterns. In saturated markets however, being able to understand individual needs and preferences is critical should one intend to develop customized products. This makes the understanding of consumer segments even more important. Nowadays, these segments go far beyond sensory preferences and must take into account variables that touch environmental awareness and sustainability dimensions.

Computational Sensory Science

Apéritifs

Chapter 2

Getting Started

2.1 Introduction to R

2.1.1 What is R?

First released in 1995, R is an open-source programming language and software environment. It is a free and powerful statistical software widely used for statistical analysis, graphics representation and reporting. R was created by Ross Ihaka and Robert Gentleman at the University of Auckland, New Zealand and is currently developed by the R Development Core Team. The name "R" was based on Ihaka's and Gentleman's first names and partly a play on the name of the Bell Labs Language S.

R is a scripting language (not a compiled language) that runs the codes or commands in order. It is one of the most popular languages used by statisticians, data analysts, researchers and marketers to retrieve, clean, analyze, visualize and present data. It is currently the 7th most popular programming language in the world and one of the most known in the sensory and consumer field.

2.1.2 Why Learn a Programming Language?

There are several reasons why you should learn a programming language. One of the top reasons is the CONTROL it gives you. Differently from statistical software to analyze/visualize data, you clearly see what is happening, what is behind the scenes, step by step. You are fully under control, and so it ends up teaching you abstract thinking and problem-solving. It allows you an INCREASED CAPABILITY and CONTINUOUS IMPROVEMENT, in the way you play with your data, extract the best you can and continuously improve it. The flexibility it gives you is just incredible!

Allied to this control, the SPEED is also something that stands out. In a few seconds, we run a long and complex script to make some plots, and if you decide to change something like removing a specific attribute or some panelists, for instance, you re-run everything in no time.

Learning a programming language also play an important role in REDUCING ERRORS, and this is in part because you are under control. You can run the script line by line and see what's happening and check if things are working properly in every and each step. It also allows you REPRODUCIBLE RESULTS by embedding script, data sets and results in a single file.

Last, but not least, it IMPROVES COLLABORATION. When user uses a version control, which will be discussed shortly, it makes possible to track changes to source over time and collaboration with multiple developers.

2.1.3   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 `{FactoMineR}` 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.

On top of the main reasons described above, R is well-supported with an active community in the way the extensive online help throughout the numerous forums or websites is readily available. Besides, there are several books, courses and other educational materials to support the user. As mentioned, the universe of available packages is vast, being the programming language with the more specialized tools for sensory analysis. Lastly, R excels at data manipulation and results reporting.

2.1.4   What is RStudio?

RStudio is a powerful and easy way to interact with R programming. It is an Integrated Development Environment (IDE) for R (also Python) that comes

with a multi-panel window setup that provides access to all primary things on a single screen, which means that that all information you need to write code is available in a single window that includes a console, syntax-highlighting editor that supports direct code execution, as well as tools for plotting, history, debugging and workplace management (https://www.rstudio.com/).

Besides the convenience of having all panels in a single screen, we strongly recommend you to go for RStudio instead of R software due to other important reasons such as having a text-editor that is full-proof and full featured, autp completion while writing scripts, many shortcuts and access to all objected created in the Environment tab. Also, the highlighting tool for editor makes coding easier and less error-prone.

2.1.5 Installing R and RStudio

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. We suggest you to download the latest version of R and install it with default options. Note that if you are running R 4.0 or higher, you might need to install Rtools: https://cran.r-project.org/bin/windows/Rtools/

Next you need to install RStudio. To do so, visit the RStudio desktop download page and follow the installation instructions. Download and install the latest version of RStudio with default options and for 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

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
```

2.2 Getting Started in R

2.2.1 Create a Local Project

We strongly recommend to always work in an RStudio project. RStudio projects make it straightforward to divide your work into multiple contexts, each with their own working directory, workspace, history, and source documents. It keeps

all of your R scripts, R markdown documents, R functions and data together in one place The projects keep your files (and activity) organized, help manage your file path (so your computer can find things) and allow for more advanced capabilities. RStudo project allow independence between project, which means that you can open more than one project at the same time and switch very easily between them without fear of them interfering with each other.

To create a new project locally in RStudio, select 'File' –> 'New Project…' from the menu. Typically, a new project is created in a new directory. You can also create a new project by clicking on the 'Project' button in the top right of RStudio and selecting 'New Project…'. Once your new project has been created you will now have a new folder on your computer that contains the basic file structure. You probably want to add folders to better organize all the files and documents, such as a folder for input, output and scripts.

2.2.2   Install and Load Packages

The base installation of R comes with many useful packages that contains many of the functions you will use on a daily basis, however, you will very quickly feel the urge to extend R's capabilities and this is possible by using R packages that can easily be downloaded and install in R. An R package is a collection of functions, data sets, help files and documentation, developed by the community, that extends the capabilities of base R, by improving existing base R functionalities or by adding new ones.

There several packages out there, but we listed above some of the packages we will be consistently be using throughout this book.

- 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]")`. R will download the packages from CRAN and install the onto your computer.

```
install.packages("tidyverse")
```

If a script loads package that are not installed, RStudio will prompt you to install the package Also, note that if you do not have write access on your

computer, you might need IT help to install packages. Additionally, you might need to safelist various R related tools and sites.

Once you have installed a package onto your computer it is not immediately available for you to use until you load it with `library()`.

```
library(tidyverse)
```

2.2.3 R Scripts

You can enter and run your code at the R command. However, you will quickly noticed that although effective and simple, each time you want to execute a set of commands, you will have to re-enter them the the commend line. Additionally, once you close your project or restart the R session, you will loose all the code stream. R scripts are the solution to those issues.

2.2.4 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)
```

2.2.5 Introduction to the {magrittr} and the notion of *pipes*

R is an *evolving* programming language that evolves and expends very rapidly.

If most additions/improvements have a fairly limited reach, the introduction of the {tidyverse} in 2016 by H. Wickham revolutionized the way of scripting in R for many users. At least for us, it had a large impact as we fully embraced its philosophy, as we see its advantage for Data Science and for analyzing our

sensory and consumer data. It is hence no surprise that you'll read and learn a lot about it in this book.

As you may know, the `{tidyverse}` is a grouping of packages dedicated to Data Science, which includes (amongst others) `{readr}` for data importation, `{tibble}` for the data structure, `{stringr}` and `{forcats}` for handling strings and factors, `{dplyr}` and `{tidyr}` for manipulating and tidying data, `{ggplot2}` for data visualization, and `{purrr}` for functional programming. But more importantly, it also includes `{magrittr}`, the package that arguably impacted the most our way of scripting by introducing the notion of *pipes* (defined as `%>%`) as it provides code that is much easier to read and understand.

To illustrate the advantage of the use of pipes, let's use the example provided by H. Wickham in his book *R for Data Science*. It is some code that tells a story about a little bunny names Foo Foo: *Little bunny Foo Foo Went hopping through the forest Scooping up the field mice and bopping them on the head*

If we were meant to tell this story though code, we would start by creating an object name `FooFoo` which is a little bunny:

```r
foo_foo <- little_bunny()
```

To this object, we then apply different functions (we save each step as a different object):

```r
foo_foo_1 <- hop(foo_foo, through=forest)
foo_foo_2 <- scoop(foo_foo_1, up=field_mice)
foo_foo_3 <- bop(foo_foo_2, on=head)
```

One of the main downsides of this approach is that you'll need to create intermediate names for each step. If natural names can be used, this will not be a problem, otherwise it can quickly become a source of error (using the wrong object for instance)! Additionally, such approach may affect your disk memory (your creating a new object in each step), especially if the data set is large.

As an alternative, we could consider running the same code by over-writing the original object:

```r
foo_foo <- hop(foo_foo, through=forest)
foo_foo <- scoop(foo_foo, up=field_mice)
foo_foo <- bop(foo_foo, on=head)
```

If this solution looks neater and more efficient (less thinking, less typing, less memory use), it is more painful to debug, as the entire code should be re-run from the beginning (when `foo_foo` was originally created). Moreover, calling the same object in each step obscures the changes performed in each line.

To these two approaches, we prefer a third one that strings all the functions together without intermediate steps of saving the results. This procedure uses the so-called pipes (defined by `%>%`), which takes automatically as input the output generated by the previous line of code:

```
foo_foo %>%
  hop(through = forest) %>%
  scoop(up = field_mice) %>%
  bop(on = head)
```

This code is easier to read and understand as it focuses more on the verbs (here `hop`, `scoop`, and `bop`) rather than names (`foo_foo_1`, or `foo_foo`). It can be surprising at first, but no worries, by the time you've read this book, you'll be fully familiar with this concept.

When lines are piped, R runs the entire block at once. So how can we understand the intermediate steps that were done, or how can we fix the code if an error occur? The answer to these questions is simple: run back the code bits by bits.

For instance, in this previous example, we could start by printing `foo_foo` (in practice, only select `foo_foo` and run this code only) only to ensure that it is the object that we were supposed to have. If it is the case, we can then extend the selection to the next line by selecting all the code before the pipe (do not select the pipe, else R is expecting additional code after it and will not show results). Repeat this until you found your error, or you ensure that all the steps have been performed correctly.

While reading this book, we advise you to apply this trick to each block of code for you to visualize the steps performed.

Note however that although pipes are very powerful, they are not always the best option:

- A rule of thumb suggests that if you are piping more than 10 lines of code, you're probably better of splitting it into 2 or more blocks (saving results in intermediate step) as this simplifies debugging.
- If some steps require multiple inputs, or provides multiple outputs, pipes should not be used as they usually require a primary object to transform.
- The system of pipes works linearly: if your code requires a complex dependency structure, the pipes should be avoided.

2.3 Version Contrl / 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].

2.3.1   Git

- Install Git
  - Windows
  - macOS
- Register with RStudio

2.3.2   GitHub

- Create a GitHub account
- Register with RStudio

2.3.3   Tips on *how to read this book?*

In this book, we assume that the readers have already some basic knowledge in R. If you are completely new to R, we recommend you reading "R for Data Science" (REF) or looking at some documentation online to get you started with the basics.

Just like with any spoken language, the same message can be said in various ways. The same applies with writing scripts in R, each of us having our own styles, or our own preferences towards certain procedures, packages, functions, etc. In other words, writing scripts is personal. Through this book, we are not trying to impose our way of thinking/proceeding/building scripts, instead we aim in sharing our knowledge built through past experiences to help you find your own style.

But to fully decode our message, you'll need some reading keys. These keys will be described in the next sections.

2.3.3.1   Running code and handling errors

For you to get the most out of this book, you need to understand (and eventually adhere to) our philosophy of scripting, and our way of working. This is why we are providing you with some tips to use, if you're comfortable with them:

1. Create a folder for this book on your computer, and create a script for each chapter in which you re-write yourself the code. If you work with the online version, you could copy/paste the code to go faster, but you may miss some subtleties.
2. Do not be discourage when you get some errors: we all get some regularly. At first, this can be very frustrating, especially when you are not able to fix them quickly. If you get stuck on an error and cannot fix it immediately, take a break and come back later with fresh eyes, you may solve it then. And with time and experience, you'll notice that you can reduce the amount of errors, and will also solve them faster.
3. The more code, the more difficult it is to find errors. This is true whether you use "regular" R-code or pipes. The best way to solve errors in such circumstances is to run the code line by line until you find the error, and understand why the input/output does not match expectations.
4. In the particular case of pipes, debugging errors means that you shouldn't run the entire block of code, but select parts of it and run it by adding in each run a new line. This can either be done by stopping your selection just before the adequate `%>%` sign, or by adding after that `%>%` sign the function `identity()`[1].

2.3.3.2 Printing vs. Saving results

In many examples through this book, we apply changes to certain elements without actually saving them in an R object. This is quite convenient for us as many changes we do are only done for pedagogic reasons, and are not necessarily relevant for our analysis.

Here is an example of such case (REF):

```r
sensory %>%
  rename(Panellist = Judge, Sample = Product)
```

If you run this code, you'll notice that we rename `Judge` to `Panellist`, and `Product` to `Sample`…at least this is what you see on screen. However, if you look at `sensory`, the data set still contains the column `Judge` and `Product` (`Panellist` and `Sample` do not exist!). This is simply because we did not save the changes in any R object.

If we would want to save the element in a new object, we should save the outcome in an element using `<-`:

---

[1] `identity()` is a function that returns as output the input as it is. This function is particularly useful in pipes as you can finish your pipes with it, meaning that you can put any line in comments (starting with '#') without worrying about finishing your pipe with a `%>%`

```
newsensory <- sensory %>%
  rename(Panellist = Judge, Sample = Product)
```

Here, `newsensory` corresponds to `sensory`, but with the new names. Of course, if you would want to overwrite the previous file with the new names, you simply need to ensure that the name of the output is the same as the name of the input. Concretely, we replace here `newsensory` by `sensory`, meaning that the new names are saved in `sensory` (so the old names `Judge` and `Product` and definitely lost). This procedure saves computer memory and does not require you coming up with new names all the time. However, it also means that some changes that you applied may be lost, and if you have a mistake in your code, it is more complicated to find and ultimately solve it (you may need to re-run your entire script).

```
sensory <- sensory %>%
  rename(Panellist = Judge, Sample = Product)
```

To visualize the changes, you would need to type `newsensory` or `sensory` in R. Another (faster) way to visualize it is to put the entire block of code between brackets: Putting code between brackets is equivalent to asking to print the output.

```
(sensory <- sensory %>%
  rename(Panellist = Judge, Sample = Product))
```

Note that if you run all these lines of codes in R, you will get an error stating `Column 'Judge' doesn't exist.` This is a good illustration of a potential error mentioned above: We overwrote the original `sensory` (containing `Judge` and `Product`) with another version in which these columns were already renamed as `Panellist` and `Sample`. So when you re-run this code, you are trying to apply again the same changes to columns that no longer exist, hence the error... This is something that you need to take into consideration when overwriting elements (in this case, you should initialize `sensory` to its original version before trying).

2.3.3.3 Calling Variables

In R, variables can be called in different ways when programming. If the names of variables should be read from the data (e.g. "Product", "products", "samples", etc.), you will often use strings, meaning that the name used will be defined between quotes (e.g. `"Product"`).

Within the `{tidyverse}`, the names of variables that are included within a data set are usually called as it is, without quote:

```
sensory %>%
  dplyr::select(Judge, Product, Shiny)
```

This is true for simple names that do not contain any special characters (e.g. space, `-`, etc.). For names that contain special characters, the use of backticks are required (backticks can also be used with simple names):

```
sensory %>%
  dplyr::select(`Judge`, Product, `Color evenness`).
```

While going through this book, you'll notice that many functions from the `{tidyverse}` sometimes require quotes, and sometimes doesn't. The simple way to know whether quotes are required or not is based on its existence in the data set or not: If the column exists and should be used, no quotes should be used. On the contrary, if the variable doesn't exist and should be created, then quotes should be used.

Let's illustrate this through a simple example involving `pivot_longer()` and `pivot_wider()` successively. For `pivot_longer()`, we create two new variables,one that contains the column names (informed by `names_to`) and one that contains the values (informed by `values_to`). Since these variables are being created, quotes are required for the new names. For `pivot_wider()`, quotes are not needed since the names of the variables to use (`names_from` and `values_from`) are present in the data:

```
sensory %>%
  pivot_longer(Shiny:Melting, names_to="Variables", values_to="Scores") %>%
  pivot_wider(names_from=Variables, values_from=Scores)
```

Hors d'Oeuvres

Chapter 3

Why Data Science?

In this chapter we explain what is data science and discuss why data science is valuable to sensory and consumer scientists. While this book focuses on the aspects of data science that are most important to sensory and consumer scientists, we recommend the excellent text Wickham and Grolemund (2016) for a more general introduction to data science.

3.1 History and Definition

You may have heard that data science was called the "sexiest job of the 21st century" by Harvard Business Review (Davenport and Patil (2012)). But what is data science? Before we give our definition, we provide some brief history for context. For a comprehensive survey of this topic, we recommend Cao (2017).

To begin, 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 [1]. 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" (Naur (1974)). According to Naur, 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 viewpoint arose in statistics, as reflected in 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" (Tukey (1962)).

---

[1] A prize roughly equivalent in prestige to a Nobel prize, but for computer 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, entitled "Statistics = Data Science?," that statistics should be called data science (Wu (1997)).

These two movements[2] 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" (Cleveland (2001)). 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."

A more recent development in the history of data science has been the realization that the standard outputs of data science - such as tables, charts, reports, dashboards, and even statistical models - can be viewed as tools that must be used in the real world in order to be valuable. This realization stems from the influence of the technology sector, where the field of design has focused on improving the ease of use of websites, apps, and devices. To quote Steve Jobs, perhaps the most influential champion of design within the technology space:

> "Design is not just what it looks and feels like. Design is how it works."

Based on this history, we provide our definition of **data science**:

> Data science is 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]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. See Fayyad et al. (1996), for example.

3.2 Benefits of Data Science

Now that we have a working definition of data science, we consider some reasons for sensory and consumer scientists to embrace it. Many of these reasons apply to any modern scientific discipline, yet the fact that sensory and consumer scientists often occupy a central location in their organizations (such as sitting between product development and marketing, for example) means that sensory and consumer scientists must routinely create useful outputs for consumption by a wide variety of stakeholders. Moreover, sensory and consumer data are often diverse, so facility in data manipulation and flexibility in data analysis are especially important skills for sensory scientists.

3.2.1 Reproducible Research

One of the most important ideas in data science is that of reproducible research (cf. Peng (2011)). Importantly, reproducibility in the context of data science does not refer to the repeatability of the experimental results themselves if the experiment were to be conducted again. What is instead meant by reproducible research is the ability to proceed from the input data to the final results in reproducible steps. Ideally, these steps should be well-documented so that any future researcher, including the researcher who originally conducted the work, should be able to determine all choices made in data cleaning, manipulation, and analysis that led to the final results. Since sensory and consumer scientists often work in teams, this clarity ensures that anyone on the team can understand the steps that led to prior results were obtained, and can apply those steps to their own research going forward.

3.2.2 Standardized Reporting

Related to the idea of reproducible research is that of standardized reporting. By following a data-scientific workflow, including automated reporting (see Chapter 6), we can standardize our reporting across multiple projecsts. This standardization has many benefits:

- **Consistent Formatting** When standardized reporting is used, outputs created by a team are formatted consistently regardless of who creates them. This consistency helps consumers of the reports - whether those consumers are executives, clients, or other team members - quickly interpret results.
- **Upstream Data Consistency** Once a standardized workflow is put in place, consistency of data formatting gains a new importance as producers of the report can save significant time by not having to reformat new data. This fact puts pressure on the data collection produce to become

more consistent, which ultimately supports knowledge management (ADD DATABASE REFERENCES).

- **Shared Learning** Once a team combines standardized reporting with tools for online collaboration such as GitHub (see Appendix 2.3), any improvement to reporting (for example, to a table, chart, text output, or even to the reporting format itself) can be leveraged by all members of the team. Thus improvements compound over time, to the benefit of all team members.

3.3   Data Scientific Workflow

A schematic of a data scientific workflow is shown in Figure 3.1. Each section is described in greater detail below.

3.3.1   Data Collection

3.3.1.1   Design

From the standpoint of classical statistics, experiments are conducted to test specific hypotheses and proper experimental design ensures that the data collected will allow hypotheses of interest to be tested (c.f. Fisher (1935)). Sir Ronald Fisher, the father of modern statistics, felt so strongly on this topic that he said:

> "To call in the statistician after the experiment is done may be no more than asking him to perform a postmortem examination: he may be able to say what the experiment died of."

This topic of designed experiments, which are necessary to fully explore causal or mechanistic explanations, is covered extensively in Lawson (2014).

Since Fisher's time, ideas around experimental design have relaxed somewhat, with Tukey arguing in Tukey (1977) that exploratory and confirmatory data analysis can and should proceed in tandem.

> "Unless exploratory data analysis uncovers indications, usually quantitative ones, there is likely to be nothing for confirmatory data analysis to consider.
>
> Experiments and certain planned inquires provide some exceptions and partial exceptions to this rule. They do this because one line of data analysis was planned as a part of the experiment or inquiry. *Even here, however, restricting one's self to the planned analysis – failing to accompany it with exploration – loses sight of the most*
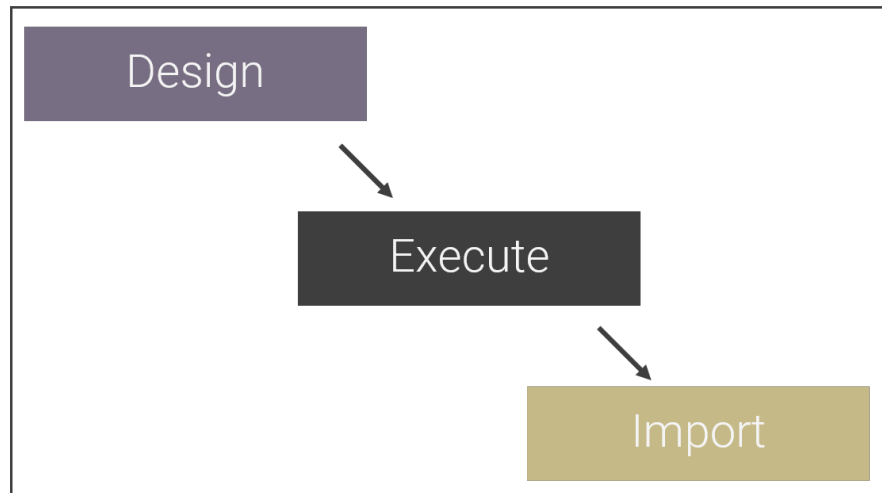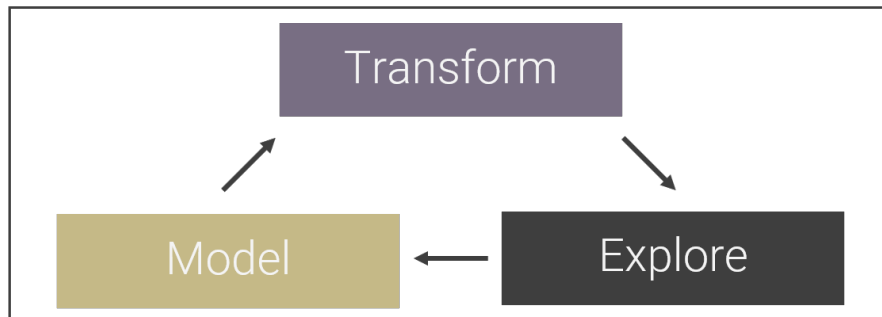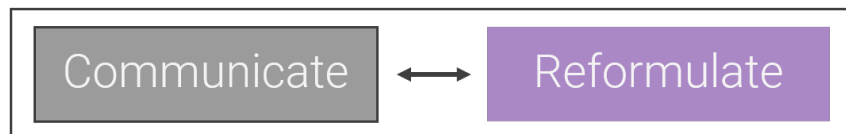
Figure 3.1: Data scientific workflow.

> *interesting results too frequently to be comfortable.* (Emphasis original)"

In this book, we take no strong opinions on this topic, as they belong more properly to the study of statistics than to data science. However, we agree that results from an experiment explicitly designed to test a specific hypothesis should be viewed as more trustworthy than results incidentally obtained. Moreover, as we describe in Chapter 12, well-selected sample sets support more generalizable predictions from machine learning models.

3.3.1.2 Execute

Execution of the actual experiment is a crucial step in the data science workflow, although not one in which data scientists themselves are necessarily involved. Even so, it is imperative that data scientists communicate directly and frequently with the experimenters so that nuances of the data are properly understood for modeling and interpretation.

3.3.1.3 Import

Once the data are collected, they need to find their way into a computer's working memory to be analyzed. This importation process should be fully scripted in code, as we detail in Chapter 8, and raw data files should never be directly edited. This discipline ensures that all steps taken to import the data will be understood later and that the reasoning behind all choices will be documented. Moreover, writing code to import raw data allows for new data to be analyzed quickly in the future as long as the data formatting is consistent. For sensory scientists, who regularly run similar tests, a streamlined workflow for data import and analysis both saves much time and protects against errors.

3.3.2 Data Preparation

Preparing data for analysis typically involves two steps: data inspection and data cleaning.

3.3.2.1 Inspect

In this step, the main goal is to gain familiarity with the data. Under ideal circumstances, this step includes reviewing the study documentation, including the study background, sampling, design, analysis plan, screener (if any), and questionnaire. As part of this step, the data should be inspected to ensure they have been imported properly and relevant data quality checks, such as checks for consistency and validity, should be performed. Preliminary summary tables

and charts should also be preformed at this step to help the data scientist gain familiarity with the data. These steps are discussed in further detail in Section 9.1 of Chapter 9.

3.3.2.2 Clean

Data cleaning is the process of preparing data for analysis. In this step we must identify and correct any errors, and ensure the data are formatted consistently and appropriately for analysis. As part of this step, we will typically tidy our data, a concept that we cover in more detail in Section 4.3. It is extremely important than any changes to the data are made in code with the reasons for the changes clearly documented. This way of working ensures that, a year from now, we don't revisit our analysis to find multiple versions of the input data and not know which version was the one used for the final analysis[3]. We discuss data cleaning in further detail in Section 9.2.

3.3.3 Data Analysis

Data analysis is one of the areas of data science that most clearly overlaps with traditional statistics. In fact, any traditional or computational statistical technique can be applied within the context of data science.

In practice, the dominant cultural difference between the two fields can be summarized as:

- Statistics often focuses on advancing explicit theoretical understanding of an area through parameter estimation within first-principle models.
- Data science often focuses on predictive ability using computational models that are validated empirically using held-out subsets of the data.

Another cultural difference between the two fields is that data science, evolving more directly out of computer science, has been more historically interested in documenting the code used for analysis with the ultimate goal of reproducible research. See Peng (2011) for more information on this topic, for example. This difference is gradually disappearing, however, as statistics more fully embraces a data scientific way of scripting analyses.

Data analysis is covered in greater detail in Chapter 10. The typical steps of data analysis are data transformation, exploration, and modeling, which we review below.

---

[3]Anyone working in the field for more than five years has almost certainly experienced this problem, perhaps even with their own data and reports

3.3.3.1   Transform

Data transformation is slightly different from data preparation. In data preparation, we prepare the raw data for processing in a non-creative way, such as reshaping existing data or storing character strings representing dates as date formatted variables. With data transformation, we create new data for analysis by applying functions to the raw data. These functions can be simple transformations such as inversions or logarithms, or can be summary operations such as computing means and variances, or could be complex operations such as principle components analysis or missing value imputation. In a machine learning context (see Chapter 12), this step is often referred to as "feature engineering." In any case, these functions provide the analyst an opportunity to improve the value of the analysis through skillful choices. Data transformation is covered in more detail in Chapter 10.

3.3.3.2   Explore

Just as data transformation differs slightly from data preparation, data exploration differs slightly from data inspection. When we inspect the data, our goal is to familiarize ourselves with the data and potentially spot errors as we do so. With data exploration, our goal is to begin to understand the results of the experiment and to allow the data to suggest hypotheses for follow-up analyses or future research. The key steps of data exploration are graphical visualizations (covered in Chapter 5) and exploratory analyses (covered in Chapter 10). As we will discuss later in this book, employing automated tools for analysis requires caution; the ease with which we can conduct a wide range of analyses increases the risk that chance results will be regarded as meaningful. In Chapter 12 we will discuss techniques, such as cross-validation, that can help mitigate this risk.

3.3.3.3   Model

At last we reach the modeling step of our workflow, which is the step in which we conduct formal statistical modeling. This step may also include predicitve modeling, which we cover in Chapter 12, as mentioned above. One difference between data science and classical statistics is that this step may feed back into the transform and explore steps, as data scientists are typically more willing to allow the data to suggest new hypotheses for testing (recall Tukey's quotation above). This step is described in further detail in Chapter 10.

3.3.4   Value Delivery

We now arrive at the final stage of the data science workflow, value delivery, which is the stage most influenced by industrial design. Recall the definition we provided above:

- **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.

From this perspective, our product consists of the final results as provided to the intended audience. Consequently, we may need to adjust both the results themselves and the way they are presented according to whether the audience consists of product developers, marketing partners, upper management, or even the general public. Hence, in this stage, we communicate our results and potentially reformulate our outputs so that they will provide maximum value to the intended audience. Although we describe value delivery in more detail in Chapter 11, we briefly review the two steps of value delivery, communicate and reformulate, below.

3.3.4.1 Communicate

The goal of the communication step is to exchange information stemming from our data scientific work. Importantly, communication is a two-way street, so it is just as important to listen in this step as it is to share results. Without feedback from our audience, we won't be able to maximize the impact of our work. We discuss this topic in more detail in Section 11.1, and note that automated reporting, which we cover in Chapter 6 also plays a large role in this step by freeing us to spend more time thinking about the storytelling aspects of our communications.

3.3.4.2 Reformulate

In the final step of our data scientific workflow, we incorporate feedback received during the communication step back into the workflow. This step may involve investigating new questions and revising the way we present results. Since we seek to work in a reproducible manner, the improvements we make to our communication can be committed to code and the lessons these improvements reflect can be leveraged again in the future. It is also important to note that, as we reformulate, we may need to return all the way to the data cleaning step, if we learn during the communication step that some aspect of the data import or initial interpretation needs to be revised. Reformulation is discussed in greater detail in Section 11.2.

3.4 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 stu-

dents 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 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[4]. 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.

3.5 Cautions: Don't that Everybody Does

Don't edit raw data. [ADD MORE]

With these preliminaries completed, and with you (hopefully) sufficiently motivated, let's begin learning data science!

---

[4]We recommend following the instructions in Appendix 2 to get started.

Chapter 4

Data Manipulation

4.1 Chapter Overview

In this section, you will learn how to manipulate and transform your data. This goes from very simple transformation (e.g. renaming columns, sorting tables, relocating variables, etc.) to more complex work (transposing parts of the table, combining/merging tables, etc.).

Such transformations are done through simple examples, including the computation of the sensory profiles (mean scores per product and attribute) of products by following different transformations path. The data set used include *Sensory Profile.xlsx* mainly, as well as made up data set (*Consumer Test.xlsx*) to illustrate how to join different data set.

Most of the functions we use are accessible through the `{tidyverse}`. For importing the data, the usual `{here}` and `{readxl}` are used.

```
library(tidyverse)
library(here)
library(readxl)
```

Note that in many cases, we are not saving the results of the transformation that we do. If you want to apply these changes to your data set, please visit section (APERITIF).

4.2 Why Manipulating Data?

In sensory science, different data collection tools (e.g. different devices, software, methodologies, etc.) may provide the same data in different ways. Also, different statistical analyses may require having the data structured in different formats.

A simple example to illustrate this latter point is the analysis of liking data. Let C consumers provide their hedonic assessments on P samples. To evaluate if samples have received different liking means at the population level, an ANOVA is performed on a long thin table with 3 columns (consumer, sample, and the liking scores), where the combination of CxP is spread in rows.

However, to assess whether consumers have the same preference patterns at the individual level, internal preference mapping or cluster analysis is performed, both these analyses requiring 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 essential to learn to manipulate data and transition from one structure to another.

Different ways to transform your data are illustrated through several simple examples[1].

4.3   Tidying Data

Hadley Wickham (Wickham (2014)) defined 'tidy data' as "data sets that are arranged such that each variable is a column and each observation (or case) is a row." Depending on the statistical unit to consider and the analyses to perform, data may need to be manipulated to be presented in a tidy form.

4.3.1   Simple Manipulations

The notion of 'simple manipulations' proposed here consists in data transformations that could easily be performed in other software such as Excel (using copy-paste, sorting and filtering, creating a pivot table, etc.). However, we strongly recommend performing any sorts of transformation in R as this will reduce the risk of errors, typically be faster, more reliable, and will be reusable if you need to perform the same operations on similar data in the future (including updated versions of the current data set). Moreover, these operations will become easier and more natural for you to use as you familiarize yourself with them.

---

[1]Note that some of the examples presented in this chapter emphasize the "how to?," not the "why?," and are not necessarily chosen to convey scientific meaning

4.3.1.1   Handling Columns

4.3.1.1.1   Renaming Variables   The first simple transformation we consider consists of renaming one or multiple variables. This procedure can easily be done using the `rename()` function from the `{dplyr}` package. In each of the examples below, we use the `names()` function to show just the names of the resulting data set.

In our sensory file[2], let's rename 'Judge' into 'Panellist', and 'Product' into 'Sample' (here we apply transformations without saving the results, so the original data set remains unchanged).

First, here are the original names:

```
sensory %>%
  names()
```

To do so, we indicate in `rename()` that `newname` is replacing `oldname` as following `rename(newname = oldname)`. Additionally, we can apply multiple changes by simply separating them with a `,`:

```
sensory %>%
  rename(Panellist = Judge, Sample = Product) %>%
  names()
```

If this procedure of renaming variables should be applied on many variables following a structured form (e.g. transforming names into snake_case, CamelCase, etc.)   (see https://en.wikipedia.org/wiki/Letter_case#Use_within_programming_languages for more information), the use of the `{janitor}` package comes handy thanks to its `clean_names()` function and the `case` parameter:

```
library(janitor)
sensory %>%
  clean_names(case="snake") %>%
  names()
```

Note that the `{janitor}` package offers many options, and although the transformation is performed here on all the variables, it is possible to apply it on certain variables only.

---

[2]The *sensory* data are imported from the *Sensory Profile.xlsx* file, see Section 8.3 for more on how to import data sets

4.3.1.1.2 Re-Organizing Columns   Another simple transformation consists in re-organizing the data set, either by re-ordering (including removing) columns, or by selecting some rows based on a certain criteria.

For re-ordering columns, `relocate()` is being used. This function allows repositioning a (set of) variable(s) before or after another variable. By re-using the `sensory` data set, let's position all the variables starting with 'Qty' between `Product` and `Shiny`. This can be specified into two different ways:

```
sensory %>%
  relocate(starts_with("Qty"), .after=Product) %>%
  names()

sensory %>%
  relocate(starts_with("Qty"), .before=Shiny) %>%
  names()
```

Another very important function regarding columns transformation is the `select()` function (from the `{dplyr}` package[3]) allows selecting a set of variables, by simply informing the variables that should be kept in the data. Let's limit ourselves in selecting `Judge`, `Product`, and `Shiny`:

```
sensory %>%
  dplyr::select(Judge, Product, Shiny)
```

When a long series of variables should be kept in the same order, the use of the `:` is used. Let's only keep the variables related to Flavor, hence going from `Cereal flavor` to `Warming`:

```
sensory %>%
  dplyr::select(Judge, Product, `Cereal flavor`:Warming)
```

However, when only one (or few) variable needs to be removed, it is easier to specify which one to remove rather than informing all the ones to keep. Such solution is then done using the `-` sign. The previous example can then be obtained using the following code:

```
sensory %>%
  dplyr::select(-c(Shiny, Melting))
```

---

[3]Note that many other packages include a function called `select()`, which could create conflicts. To avoid any risks of errors, we recommend calling it as `dplyr::select()` to formally call `select()` from `{dplyr}`. This avoids any risks of error! Of course, the same procedure applies to any other functions that may suffer from the same issue.

The selection process of variables can be further informed through functions such as `starts_with()`, `ends_with()`, and `contains()`, which all select variables that either starts, ends, or contains a certain character or sequence of character. To illustrate this, let's only keep the variables that starts with 'Qty':

```
sensory %>%
  dplyr::select(starts_with("Qty"))
```

Rather than selecting variables based on their names, we can also select them based on their position (e.g. `dplyr::select(2:5)` to keep the variables that are at position 2 to 5), or following a certain 'rule' using the `where()` function. In that case, let's consider all the variables that are numerical, which automatically removes `Judge` and `Product`:

```
sensory %>%
  dplyr::select(where(is.numeric))
```

Remark: `dplyr::select()` is a very powerful function that facilitates selection of complex variables through very intuitive functions. Ultimately, it can also be used to `relocate()` and even `rename()` variables, as shown in the example below:

```
sensory %>%
  dplyr::select(Panellist = Judge, Sample = Product, Shiny:Sticky, -starts_with("Qty"))
```

More examples illustrating the use of `dplyr::select()` are provided throughout the book.

4.3.1.1.3 Creating Columns In some cases, new variables need to be created from existing ones. Examples of such situations include taking the quadratic term of a sensory attribute to test for curvature, or simply considering a new variables as the sum or the subtraction between two (or more). Such creation of a variable is processed through the `mutate()` function from the `{dplyr}` package. This function takes as inputs the name of the variable to create, and the *formula* to consider. Let's create two new variables, one called `Shiny2` which corresponds to `Shiny` squared up, and one `StiMelt` which corresponds to `Sticky + Melting`. Since we only use these three variables, let's first reduce the data set to these three variables with `select()` to improve readability:

```
sensory %>%
  dplyr::select(Shiny, Sticky, Melting) %>%
  mutate(Shiny2 = Shiny^2,
         StiMelt = Sticky + Melting)
```

Tip: If you want to transform a variable, say by changing its type, or re-writing its content, you can use `mutate()` and assign to the new variable the same name as the original one. This will overwrite the existing column with the new one. To illustrate this, let's transform `Product` from upper case to lower case only. This can be done by mutating `Product` into the lowercase version of `Product` (`tolower(Product)`):

```
sensory %>%
  mutate(Product = tolower(Product))
```

`mutate()` being one of the most important function from the `{dplyr}` package, more examples of its use are presented throughout this book.

Since performing mathematical computations on non-numerical columns is not possible, conditions can easily be added through `mutate()` combined with `across()`. Let's imagine we want to round all variables to 0 decimal, which can only be applied to numerical variables. To do so, we `mutate()` `across()` all variables that are considered `as.numeric` (through `where()`):

```
# round(sensory, 0) returns an error because Judge and Product are characters

sensory %>%
  mutate(across(where(is.numeric), round, 0))
```

In case only a selection of numerical variables should be rounded, we could also replace `where(is.numeric)` by a vector (using `c()`) with the names of the variables to round.

4.3.1.1.4 Merging and Separating columns It can happen that some columns of a data set contain information (strings) that cover different types of information. For instance, we could imagine coding the name of our panelists as *FirstName_LastName* or *Gender_Name*, and we would want to separate them into two columns to make the distinction between the different information, i.e. *FirstName* and *LastName* or *Gender* and *LastName* respectively. In other situations, we may want to merge information present in multiple columns in one.

For illustration, let's consider the information stored in the *Product Info* sheet from *Sensory Profile.xlsx*. This table includes information regarding the cookies, and more precisely their Protein and Fiber content (Low or High).

After importing the data, let's merge these two columns so that both information is stored in one column called `ProtFib`. To do so, we use the `unite()` function from the `{tidyr}` package, which takes as first element the name of the new variables, followed by all the columns to *unite*, and by providing the separation between these elements (here -):

```
file_path <- here("data","Sensory Profile.xlsx")
prodinfo <- read_xlsx(file_path, sheet="Product Info") %>%
  unite(ProtFib, Protein, Fiber, sep="-")
prodinfo
```

By default, `unite()` removes from the data set the individual variables that have been merged. To keep these original variables, the parameter `remove = FALSE` can be used.

Although it is not relevant for combining columns, it is interesting to mention an additional package that can be used to combine elements together. This package is called `{glue}` and provides interesting alternatives to the usual `paste()` and `paste0()` functions.

To reverse the changes (saved here in `prodinfo`) and to separate a column into different variables, the function `separate()` from the `{tidyr}` package is used. Similarly to `unite()`, `separate()` takes as first parameter the name of the variable to split, followed by the names for the different segments generated, and of course the separator defined by `sep`.

In our example, this would be done as following:

```
prodinfo %>%
  separate(ProtFib, c("Protein","Fiber"), sep="-")
```

4.3.1.2 Handling Rows

After manipulating columns, the next logical step is to handle rows. Such operations include three aspects,

1. Re-arranging the rows in a logical way,
2. Filtering entries based on given variables,
3. Splitting the data in sub-groups based on the entries of a variable.

4.3.1.2.1 Re-arranging Rows The first step of re-arranging rows is done through the `arrange()` function from the `{dplyr}` package. This function allows sorting the data in the ascending order[4]. To arrange them in a descending order, the function `desc()` is also required.

Let's re-arrange the data by Judge and Product, the Judge being sorting in an ascending order whereas the product are being sorted in a descending order:

---

[4]For numerical order, this is simply re-arranging the values from the lowest to the highest. For strings, the entries are then sorted alphabetically unless the variable is a factor in which case the order of the levels for that factors are being used.

```
sensory %>%
  arrange(Judge, desc(Product))
```

4.3.1.2.2 Filtering Data To define sub-set of data, the `filter()` function is being used. This function requires providing an argument that is expressed as a *test*, meaning that the outcome should either be TRUE (keep the value) or FALSE (discard the value) when the condition is verified or not respectively. In R, this is expressed by the double '=' sign `==`.

Let's filter the data to only keep the data related to sample `P02`:

```
sensory %>%
  filter(Product == "P02")
```

Other relevant test characters are the following: - `!Product == "P02"` or `Product != "P02"` means different from, and will keep all samples except P02; - `%in% my_vector` keeps any value included within the vector `my_vector` (e.g. `Product %in% c("P01","P02","P03")`)

In some cases, the tests to perform are more complex as they require multiple conditions. There are two forms of conditions:

- `&` (read *and*) is multiplicative, meaning that all the conditions need to be true (`Product == "P02" & Shiny > 40`);
- `|` (read *or*) is additive, meaning that only one of the conditions needs to be true (`Product == "P03" | Shiny > 40`)

As we will see later, this option is particularly useful when you have missing values as you could remove all the rows that contain missing values for a given variable. Since we do not have missing values here, let's create some by replacing all the evaluations for `Shiny` that are larger than 40 by missing values. This is done here through the `ifelse()`, which takes three arguments (in this order): the test to perform (here `Shiny > 40`), the instruction if the test passes (here replace with `NA`), and the instruction if the test doesn't pass (here keep the value stored in `Shiny`).

```
(sensory_na <- sensory %>%
  dplyr::select(Judge, Product, Shiny) %>%
  mutate(Shiny = ifelse(Shiny > 40, NA, Shiny)))
```

In a second step, we filter out all missing values from `Shiny`. In practice, this is done by keeping all the values that are not missing:

```
sensory_na %>%
  filter(!is.na(Shiny))
```

As we can see, this procedure removed 20 rows since the original table had 99 rows and 3 columns, whereas the filtered table only has 79 rows and 3 columns.

4.3.1.2.3 Splitting Data After filtering data, the next logical step is to split data into subsets based on a given variable (e.g. by gender). For such purpose, one could consider using `filter()` by applying it to each subgroup. In a previous example, this is what we have done when we filtered data for sample `P02` only.

Of course, we could get sub-groups of data for each sample by repeating the same procedure for all the other samples. However, this procedure becomes tedious as the number of samples increases. Instead, we prefer to use `split()` which takes as arguments the data and the column to split from:

```
split(sensory, sensory$Product)

# In a pipe, it would be written as (the '.' calls the data):
#
# sensory %>% split(.$Product)
```

This function creates a list of n elements (n being the number of samples), each element corresponding to the data related to one sample. Such list can then be used in automated analyses by performing on each sub-data through the `map()` function, as it will be illustrated later.

4.3.2 Reshaping the Data

Reshaping the data itself is done through pivoting, hence either creating a long and thin table (CREATE FIGURE), or a short and wide table (CREATE FIGURE). This is done through `pivot_longer()` and `pivot_wider()` from {tidyr}.

4.3.2.1 Pivotting Longer

Currently, our `sensory` data table is a table in which we have as many rows as Judge x Product, the different attributes being spread across multiple columns. However, in certain situations, it is relevant to have all the attributes stacked vertically, meaning that the table will have Judge x Product x Attributes rows. Such simple transformation can be done through the `pivot_longer()` function from the {dplyr} package, which takes as inputs the attributes to pivot, the

name of the variables that will contain these names (`names_to`), and the name
of the column that will contain their entries (`values_to`)

```
sensory %>%
  pivot_longer(Shiny:Melting, names_to="Attribute", values_to="Score")
```

This transformation converts a table of 99 rows and 34 columns into a table
with 3168 (99*32) rows and 4 columns. *In the pivoted table, the names of the
variables (stored here in* Attribute*) are in the same order as presented in the*
original table.

TIPS: With `pivot_longer()` and any other function that requires selecting
variables, it is often easier to deselect variables that we do not want to include
rather than selecting all the variables of interest. Throughout the book, both
solutions are being considered.

In case the attribute names are following a standard structure, say "at-
tribute_name modality" as is the case in `sensory` for some attributes, an
additional parameter of `pivot_longer()` becomes handy as it splits the
Attribute variable just created into say 'Attribute' and 'Modality.'

To illustrate this, let's reduce sensory to Judge, Product, and all the variables
that end with odor or flavor (all other variables being discarded). After pivoting
the subset of columns, we automatically split the attribute names into *Attribute*
and *Modality* by informing the separator between names (here, a space):

```
sensory %>%
  dplyr::select(Judge, Product, ends_with("odor"), ends_with("flavor")) %>%
  pivot_longer(-c(Judge,Product), names_to=c("Attribute","Modality"), values_to="Score"
```

This parameter combines both the power of `pivot_longer()` and `separate()`
in one unique process. Note that more complex transformations through the
use of regular expressions (and the `names_pattern` option) can be considered.
More information on this topic is provided in REF CHAPTER TEXTUAL (TO
CHECK IF IT IS THE CASE!).

Note that the `{reshape2}` package provides a similar function called `melt()`,
which comes particularly handy if the entire set of numerical variables should
be pivoted. In case qualitative variables are present in the data set, they will
be kept as *id variables*. If performed on a matrix with row names, then the new
table will have two columns containing the row and column names.

```
melt(sensory)
```

4.3.2.2 Pivotting Wider

The complementary/opposite function to `pivot_longer()` is `pivot_wider()`.
This function pivots data horizontally, hence reducing the number of rows and

increasing the number of columns. In this case, the two main parameters to provide is which column will provide the new column names to create (`name_from`), and what are the corresponding values to use (`values_from`).

From the previous example, we could set `names_from = Attribute` and `values_from = Score` to return to the original format of sensory. However, let's reduce the data set to `Product`, `Judge`, and `Shiny` only, and let's pivot the `Judge` and `Shiny` columns:

```
sensory %>%
  dplyr::select(Judge, Product, Shiny) %>%
  pivot_wider(names_from = Judge, values_from = Shiny)
```

This procedure creates a table with as many rows as there are products, and as many columns as there are panelists (+1 since the product information is in a column, not defined as row names).

These procedures are particularly useful in consumer studies, since `pivot_longer()` and `pivot_wider()` allows restructuring the data for analysis such as ANOVA (`pivot_longer()` output) and preference mapping or clustering (`pivot_wider()` structure).

Important remarks: Let's imagine the sensory test was performed following an incomplete design, meaning that each panelist did not evaluate all the samples. Although the long and thin structure would not show missing values (the entire rows being removed), the shorter and larger version would contain missing values for the products that each panelist did not evaluate. If the user wants to automatically replace these missing values with a fixed value, say, it is possible through the parameter `values_fill` (e.g. `values_fill=0` would replace each missing value with a 0). Additionally, after pivoting the data, if multiple entries exist for a combination row-column, `pivot_wider()` will return a list of elements. In the next Section, an example illustrating such situation and its solution will be presented.

In previous versions of `pivot_wider()`, the order of the new columns did not match the original order as it would re-organize them alphabetically (mainly if the variable specified in `names_from` was of type character). If this situation is fixed with later updates (if this should still happen to you, try to update`{tidyr}`), there was a work around which consists in transforming this variable as a factor (use `fct_inorder()` to maintain the order as shown in the data) before pivoting it. Indeed, if the variable is of type factor, the new columns are re-ordered based on its levels' order. Such trick is great to keep in mind as it can help in many other situations (such as ordering elements on a graphic for instance!)

4.3.3   Transformation that Alters the Data

In some cases, the final table to generate requires altering the data, by (say) computing the mean across multiple values, or counting the number of occurrences of factor levels for instance. In other words, we summarize the information, which also tend to reduce the size of the table. It is hence no surprise that the function used for such data reduction is called `summarise()` (`{dplyr}` package).

4.3.3.1   Introduction to Summary Statistics

In practice, `summarise()` applies a function (whether it is the `mean()`, or a simple count using `n()`) on a set of values. Let's compute the mean on all numerical variables of `sensory`:

```
sensory %>%
  summarise(across(where(is.numeric), mean))
```

As can be seen, the grand mean is computed for each attribute. If multiple functions should be applied, we could perform all the transformation simultaneously as following:

```
sensory %>%
  summarise(across(where(is.numeric), list(min=min, max=max)))
```

In this example, each attribute is duplicated with "_min" and "_max" to provide the minimum and maximum value for each attribute. By using a combination of `pivot_longer()` (or `reshape2::melt()`) with `names_sep` followed by `pivot_wider()`, we could easily restructure such table by showing for each attribute (presented in rows) the minimum and the maximum in two different columns.

By following the same principles, many other functions can be performed, whether they are built-in R or created by the user. Here is a recommendation of interesting descriptive functions to consider with `summarise()`:

- `mean()`, `median()` (or more generally `quantile()`) for the mean and median (or any other quantile);
- `sd()` and `var()` for the standard deviation and the variance;
- `min()`, `max()`, `range()` (provides both the min and max) or `diff(range())` (for the difference between min and max);
- `n()` and `sum()` for the number of counts and the sum respectively.

4.3.3.2 Introduction to *grouping*

It can appear that the interest is not in the grand mean, but in the mean per product (say), or per product and panelist in case the test has been duplicated. In such cases, `summarize()` should aggregate set of values per product, or per product by panelist respectively. Such information can be passed on through `group_by()`[5].

```
sensory %>%
  group_by(Product) %>%
  summarise(across(where(is.numeric), mean)) %>%
  ungroup()
```

This procedure creates a tibble with 11 rows (product) and 33 columns (32 sensory attributes + 1 column including the product information) which contains the mean per attribute for each sample, also known as the sensory profiles of the products.

4.3.3.3 Illustrations of Data Manipulation

Let's review the different transformations presented earlier by generating the sensory profiles of the samples through different approaches[6].

In the previous example, we've seen how to obtain the sensory profile using `summarise() across()` all numerical variables. In case a selection of the attributes should have been done, we could use the same process by simply informing which attributes to transform:

```
sensory %>%
  group_by(Product) %>%
  summarise(across(Shiny:Melting, mean)) %>%
  ungroup()
```

The list of attributes to include can also be stored in an external vector:

```
sensory_attr <- colnames(sensory)[5:ncol(sensory)]
sensory %>%
  group_by(Product) %>%
  summarise(across(all_of(sensory_attr), mean)) %>%
  ungroup()
```

---

[5]We strongly recommend you to `ungroup()` blocks of code that includes `group_by()` once the computations are done to avoid any unexpected results.

[6]It is important to realize that any *data manipulation challenge* can be solved in many different ways, so don't be afraid to think out of the box when solving them...

Remark: It is important to notice that when `group_by()` is being called, the
software will remember the groups unless stated otherwise. This means that
any subsequent transformation performed on the previous table will be done
by product. Such property can be causing unexpected results in case transfor-
mations should be performed across all samples. To avoid such behavior, we
strongly recommend you to apply `ungroup()` as soon as the results per group
have been generated.

A different approach consists in combining `summarise()` to `pivot_longer()`
and `pivot_wider()`. This process requires summarizing only one column by
Product and Attribute:

```
sensory %>%
  pivot_longer(Shiny:Melting, names_to="Attribute", values_to="Scores") %>%
  mutate(Attribute = fct_inorder(Attribute)) %>%
  group_by(Product, Attribute) %>%
  summarise(Scores = mean(Scores)) %>%
  pivot_wider(names_from=Attribute, values_from=Scores) %>%
  ungroup()
```

Note that through this procedure, we transformed `Attribute` into a factor using
`fct_inorder()` to ensure that the double pivoting procedure maintains the
original attributes' order. Without this line of code, the final table would have
the columns reordered alphabetically.

What would happen if we would omit to `summarise()` the data in between the
two pivoting functions? In that case, we also remove Judge which were lost in
the process...

```
sensory %>%
  pivot_longer(Shiny:Melting, names_to="Attribute", values_to="Scores") %>%
  dplyr::select(-Judge) %>%
  pivot_wider(names_from=Attribute, values_from=Scores)
```

As can be seen, each variable is of type list in which each cell contains `dbl [9]`:
This corresponds to the scores provided by the 9 panelists to that product and
that attribute. Since we would ultimately want the mean of these 9 values to
generate the sensory profiles, a solution comes directly within `pivot_wider()`
through the parameter `values_fn` which applies the function provided here on
each set of values:

```
sensory %>%
  pivot_longer(Shiny:Melting, names_to="Attribute", values_to="Scores") %>%
  dplyr::select(-Judge) %>%
  pivot_wider(names_from=Attribute, values_from=Scores, values_fn=mean)
```

4.3.4 Combining Data from Different Sources

It often happens that the data to analyze is stored in different files, and need to be combined or merged. Depending on the situations, different solutions are required.

4.3.4.1 Binding by Rows

Let's start with a simple example where the tables match in terms of variables, and should be combined vertically. To do so, we use the file *excel-scrap.xlsx* which contains a fake example in which 12 assessors evaluated 2 samples on 3 attributes in triplicate, each replication being stored in a different sheet.

The goal here is to read the data stored in the different sheets, and to combine them in one unique file.

To combine the tables vertically, we could use the basic R function `rbind()`. However, we prefer to use `bind_rows()` from `{dplyr}` since it better controls for the columns by ensuring that the order is respected. And in case one table contains a variable that the other tables don't, this variable will be kept and filled in with NAs when the information is missing.

To keep the distinction between the three tables, the parameter `.id` is used. This will create a column called `Session` in this example that will assign a 1 to the first table, a 2 to the second one, and a 3 to the third one (This process is used to avoid losing this information as it was not directly available within the data: If it were, the parameter `.id` could have been ignored).

```r
path <- file.path("data", "excel_scrap.xlsx")

session1 <- read_xlsx(path, sheet=1)
session2 <- read_xlsx(path, sheet=2)
session3 <- read_xlsx(path, sheet=3)

all_data <- bind_rows(session1, session2, session3, .id = "Session")
```

Although this solution works fine, another neater and tidier solution will be presented in 8.3.3.

4.3.4.2 Combining Tables by column(s)

Similarly, tables can be combined horizontally using the corresponding function `cbind()` (`{base}`) and/or `bind_cols()` (`{dplyr}`). In this case, it is better to ensure that the rows' order is identical before combining to avoid mishaps, and the tables should be of the same size.

If that is not the case, mergeing tables should be done using `merge()` (`{base}`), or preferably through the different `*_join()` functions from (`{dplyr}`). Depending on the *merging degree* to consider between tables X and Y, there are four different `*_join()` versions to consider:

- `full_join()` keeps all the cases from X and Y regardless whether they are present in the other table or not (in case they are not present, NAs will be introduced) [it corresponds to `merge(all=TRUE)`];
- `inner_join()` only keeps the common cases, i.e. cases that are present in both X and Y [corresponds to `merge(all=FALSE)`];
- `left_join()` keeps all the cases from X [corresponds to `merge(all.x=TRUE, all.y=FALSE)`];
- `right_join()` keeps all the cases from Y [corresponds to `merge(all.x=FALSE, all.y=TRUE)`];
- `anti_join()` only keeps the elements from X that are not present in Y (this is particularly useful if you have a tibble Y of elements that you would like to remove from X).

The merging procedure requires the users to provide a *key*, i.e. a (set of) variable(s) used to combine the tables. For each unique element defined by the key, a line is being created. When needed, rows of a table are being duplicated. Within the different `*_join()` functions, the key is informed by the `by` parameter, which may contain one or more variables with the same or different names.

To illustrate, let's use the data set called *Consumer Test.xlsx*, which contains three tabs:

```
file_path <- here("data","Consumer Test.xlsx")

excel_sheets(file_path)
```

The three sheets contain the following information, which need to be combined:

- *Biscuits*: The consumers' evaluation of the 10 products and their assessment on liking, hunger, etc. at different moments of the test.
- *Time Consumption*: The amount of cookies and the time required to evaluate them in each sitting.
- *Weight*: The weight associated to each cookie.

Let's start by combining *Time Consumption* and *Weight* so that we can compute the total weight of biscuits eaten by each respondent in each sitting. In this case, the joining procedure is done by `Product` since the weight is only provided for each product. The total weight eaten (`Amount`) is then computed by multiplying the number of cookies eaten (`Nb biscuits`) by `Weight`

```
time <- read_xlsx(file_path, sheet="Time Consumption")
weight <- read_xlsx(file_path, sheet="Weight")

(consumption <- time %>%
  full_join(weight, by="Product") %>%
  mutate(Amount = `Nb biscuits`*Weight))
```

As can be seen, the `Weight` information stored in the *Weight* sheet has been replicated every time each sample has been evaluated by another respondent.

The next step is then to merge this table to `Biscuits`. In this case, since both data set contain the full evaluation of the cookies (each consumer evaluating each product), the joining procedure needs to be done by `Judge` and `Product` simultaneously. A quick look at the data shows two important things:

- In *Biscuits*, the consumer names only contains the numbers whereas in `consumption`, they also contain a J in front of the name: This needs to be fixed as the names need to be identical to be merged, else they will be considered separately and NAs will be introduced. In practice, this will be done by mutating `Consumer` and by pasting a J in front of the number using the function `paste0()`.
- The names that contain the product (`Samples` and `Product`) and consumers (`Consumer` and `Judge`) information are different in both data set. We could rename these columns in one data set to match the other, but instead we will keep the two names and inform it within `full_join()`. This is done through the `by` parameter as following: `"name in dataset 1" = "name in dataset 2"`

```
biscuits <- read_xlsx(file_path, sheet="Biscuits") %>%
  mutate(Consumer = str_c("J",Consumer)) %>%
  full_join(consumption, by=c("Consumer"="Judge", "Samples"="Product"))

biscuits
```

The three data sets are now flawlessly joined into one and can be further analysed.

Chapter 5

Data Visualization

5.1   Design Principles

5.2   Table Making

5.3   Chart Making

"A picture is worth 1000 words". This saying definitely applies to Statistics as well, since visual representation of data often appears clearer than the values themselves stored in a table. It is hence no surprise that R is a powerful tool for graphics.

In practice, there are various ways to build graphics in R. In fact, R itself comes with a powerful way of building graphs through the `plot()` function. An extensive description can be found in (*R Graphics 2nd edition Paul Murrell CRC Press*). Due to its philosophy, its simplicity, and the point of view adopted in this book, we will limit ourselves to graphics built using the `{ggplot2}` package.

5.3.1   Philosophy of `{ggplot2}`

`{ggplot2}` belongs to the `{tidyverse}`, and was developed by H. Wickham and colleagues at RStudio. It is hence no surprise that a lot of the procedures that we are learning throughout this book also applies to `{ggplot2}`. More generally, building graphics with `{ggplot2}` fits very well within the pipes (`%>%`) system from `{magrittr}`. As we will see, `{ggplot2}` also works with a piping system, except that the symbol used is `+` instead of `%>%`.

In practice, `{ggplot2}` is a multi-layer graphical tools, and graphics are built by adding layers to existing graphs. The advantage of such procedure is that

`ggplot` objects are not fixed: They can be printed at any time, and can still be improved by adding other layers if needed. To read more about {`gglot2`} and its philosophy, please refer to http://vita.had.co.nz/papers/layered-grammar.pdflink.

Note that since building graphics is limited to one's imagination, it is not possible to tackle each and every possibilities offered by {`ggplot2`} (and its extensions). For that reason, we limit ourselves to describing the principles of how {`ggplot2`}works. Additionally, we provide in this section and throughout the book examples of graphics that are useful in Sensory and Consumer research. This should be more than sufficient to get you started, and should cover 90% of your daily needs. Still, if that should not be sufficient, we invite you to look into the online documentation or to references such as [REFS].

5.3.2   Getting started with {`ggplot2`}

To use {`ggplot2`}, it needs to be loaded. This can either be done directly using:

```
library(ggplot2)
```

However, if you load the {`tidyverse`} package, this step can be ignored as {`ggplot2`} is included within the list of packages it contains:

```
library(tidyverse)
```

To illustrate the use of {`ggplot2`}, both the sensory data (*Sensory Profile.xlsx*) and the number of biscuit eaten by each respondents (*Consumer Test.xlsx*) are used.

```
library(here)
library(readxl)

# Sensory Profiles Data
file_path <- here("data","Sensory Profile.xlsx")
p_info <- read_xlsx(file_path, sheet="Product Info") %>%
  dplyr::select(-Type)

sensory <- read_xlsx(file_path, sheet="Data") %>%
  inner_join(p_info, by="Product") %>%
  relocate(Protein:Fiber, .after=Product)

# Number of Biscuits Eaten Data
file_path <- here("Data","Consumer Test.xlsx")
```

```
Nbiscuit <- read_xlsx(file_path, sheet="Time Consumption") %>%
  mutate(Product = str_c("P", Product)) %>%
  rename(N = `Nb biscuits`)
```

To initiate a graph, the function `ggplot()` is called. Since the data to be used are stored in `sensory`, `ggplot()` is applied on `sensory`:

```
p <- ggplot(sensory)
```

Running this line of code generates an empty graphic stored in `p`. This is logical since no layers have been added yet. So let's imagine we want to look at the overall relationship between `Sticky` and `Melting`. To evaluate this relationship, a scatter plot with `Sticky` in the x-axis and `Melting` in the y-axis can be created. To do so, two types of information are required:

- the type of visual (here a scatter point);
- the information regarding the data to plot (what should be represented).

Such information can be provided as such:

```
p + geom_point(aes(x=Sticky, y=Melting))
```

As can be seen, a layer that consists of points (defined by `geom_point()`) in which the x-axis coordinates is defined by `Sticky` and the y-axis coordinates by `Melting` defined through aesthetics (or `aes()`) is added to the already existing graph `p` .

5.3.2.1 Introduction to Aesthetics

In the previous example, one can notice that many points are being printed. This surprising result is logical since `sensory` contains the raw sensory data, meaning that there are as many points as there are assessors evaluating products.

Let's color the points per products to see if we can see any patterns. Since the color code is specific to the data (more precisely to `Product`), it should be informed within the aesthetics by adding `colour=Product` within `aes()`:

```
p + geom_point(aes(x=Sticky, y=Melting, colour=Product))
```

As you can see, any parameters provided within `aes()` may depend on a variable (e.g. `colour` in the previous example). If for any reasons, a specific setting should uniformly be applied to all the elements of the graph, then it should be stated outside `aes()`.

Let's illustrate this by providing a simple example in which we change the type of the dots from circle to square using `pch`, and by increasing their size using `cex`:

```
p + geom_point(aes(x=Sticky, y=Melting, colour=Product), pch=15, cex=5)
```

Depending on the `geom_*()` considered, different parameters should be informed within `aes()`. Here is a list of the most common `aes()` you would use:

- `x`, `y`, `z`, provides the coordinates on the X, Y, Z dimensions respectively;
- `colour`/`color`, `fill` controls for the color code[1] that is being applied to the different elements of a graph;
- `group` makes the distinction between points that belong to different groups[2];
- `text`, `label` prints text on the graph;
- `size` controls the size of the element (this should preferably be used with numerical variables).

Examples highlighting various types of aesthetics are presented throughout the book.

5.3.2.2   Introduction to `geom_*()` functions

Since {`ggplot2`} is a multi-layer graph, let's add another layer. For example, the name/code of the panelists associated to each point can be printed.

Such procedure is done thorugh the use of another `geom_*()` function in `geom_text()`[3] which requires in `aes()` the position of the labels (`x` and `y`) as well as the `label` itself.

To avoid having the label overlapping with the point, the text is slightly shifted vertically using `nudge_y`:

```
ggplot(sensory)+
  geom_point(aes(x=Sticky, y=Melting, colour=Product))+
  geom_text(aes(x=Sticky, y=Melting, label=Judge), nudge_y=1)
```

One interesting remark is that some information required in `aes()` is being repeated across the different `geom_*()` used. Such writing can be simplified by providing the `aes()` information that applies to all `geom_*()` to the original `ggplot()` call. The previous code hence can be simplified to:

---

[1]You can also use `alpha` to control for the transparency of the elements by defining values between 0 (completely transparent) to 1 (no transparency)

[2]Note that `colour` and `fill` are specific cases of groups as they additionally provide a visual cue on the groups through the color code

[3]Try using `geom_label()` instead of `geom_text()` to see the difference between these two

```
p <- ggplot(sensory, aes(x=Sticky, y=Melting, label=Judge))+
  geom_point(aes(colour=Product))+
  geom_text(nudge_y=1)
```

With this new code, you'll notice that:

- although `label` is only relevant for `geom_text()`, it can still be provided at the beginning as it will be ignored by `geom_point()` which does not require it;
- `colour` should only be provided within `geom_point()` else the text would also be colored according to `Product` (which we do not want here);
- `nudge_y` is defined outside `aes()` as it applies to all the text.

Since the graphics look at the relationship between two quantitative variables, let's add another layer to the previous graph that shows the regression line between:

```
line_p <- p + geom_smooth(method="lm", formula="y~x", se=FALSE)
```

This code adds a regression line to the graphic. It is built using the `lm()` engine in which the simple linear regression model `y~x` is fitted. This result is somewhat surprising since we have not run any regression yet, meaning that `geom_smooth()` is actually performing the required analysis on the data in the background.

In fact, most `geom_*()` function comes with a statistical process attached to it. This means that on the raw data, the `geom_*()` function calls its `stat_*()` function that runs the corresponding analysis. In the previous example, `geom_smooth()` calls `stat_smooth()`.

Let's illustrate this concept again using another example: bar-charts that is applied on the data stored in `Nbiscuit`. Here, we want to see the distribution (through bar-charts) of number of biscuits eaten per consumer. A quick look at the data shows that some respondents ate portions of the cookies. To simplify the analysis, let's consider the total number of entire cookies eaten: if a respondent has eaten say 3.5 biscuits, it will be rounded down to 3 full cookies.

```
Nbiscuit <- Nbiscuit %>%
  mutate(N = floor(N))
```

To create such distribution, a first solution consists in counting for each product how many respondents ate 0 biscuit, 1 biscuit, 2 biscuits, etc. This is automatically done using `geom_bar` and `stat="count"`. The parameter `position="dodge"` is used to get the results per biscuit side by side rather than stacked up (value by default):

```r
bar_p <- ggplot(Nbiscuit, aes(x=N, fill=Product))+
  geom_bar(stat="count", position="dodge")
```

In the background, this corresponds to grouping the data by `Product`, summarizing the results by counting `N`, and then performing `geom_bar()` in which no transformation is required (we set `stat="identity"`)[4]:

```r
Nbiscuit %>%
  count(Product, N) %>%
  ggplot(aes(x=N, y=n, fill=Product))+
  geom_bar(stat="identity", position="dodge")
```

As can be seen, the two graphics are identical.

5.3.2.3   Making graphs pretty

In the two previous graphs generated (stored in `line_p` and `bar_p`), some features can be changed to produce clearer visualizations. Currently, the background is grey with vertical and horizontal white lines, the legend is positioned on the right side, the axis is defined based on the data itself (and so are the axis titles), there is no title, etc. and all these points (amongst many more) that are being improved in this section.

Let's start with a quick win by completely changing the overall appearance of the graph. To do so, predefined *themes* with pre-set background (with or without lines), axis lines, etc. can be applied. The two themes we use the most are `theme_minimal()` and `theme_bw()` (see https://ggplot2.tidyverse.org/reference/ggtheme.htmllink for a complete list of pre-defined themes.)

Let's start with improving `bar_p` using `theme_minimal()`:

```r
bar_p = bar_p + theme_minimal()
```

Rather than using pre-defined themes (or to complement pre-defined themes), the different parameters of the graph can be controlled through `theme()`.

Next, let's modify the axes by changing their names and by applying more logical breaks. For instance, the limits of the x-axis can be extended to -1 and 11 to ensure that all the histograms are visible, else R removes some and returns a warning: `Removed 10 rows containing missing values`.

---

[4]This code could even be simplified by using `geom_col()` since `geom_col()` corresponds to `geom_bar()` with `stat="identity"` as default.

```
bar_p = bar_p +
  scale_x_continuous(name="Number of Biscuits eaten",
                     breaks=seq(0,10,1),
                     labels=c("None", 1:9, "All of them"),
                     limits=c(-1,11))+
  ylab("Number of Respondents")
```

Last but not least, a title is being added to the graph using `ggtitle()`:

```
bar_p = bar_p +
  ggtitle("Distribution of the number of biscuits eaten","(Results are split per biscuit type)")
```

Let's apply a similar transformation to `line_p`. Here, we are aiming in having a more *realistic* plot using cartesian coordinates, a nice theme, no legend, and a title to the graph.

```
line_p = line_p +
  theme_bw()+
  scale_x_continuous(breaks=seq(0,50,10), limits=c(0,60))+
  scale_y_continuous(breaks=seq(0,50,10), limits=c(0,60))+
  coord_fixed()+
  ggtitle("Relationship between Melting and Sticky", "Biscuits perceived as more sticky tend to b
  guides(colour="none")
```

5.3.3 Common Charts

You have now an overview of the basics of `{ggplot2}` and its philosophy. You'll find plenty of other examples throughout this book to help you develop your skills in building graphics in R.

Since making an exhaustive list of plots that are relevant in sensory and consumer science is out of the scope for this book, it is not going to be further developed here. Yet, here is a summary of the `geom_*()` that could be of interest for you:

- Scatter points:
    - `geom_point()`: create a scatter point (see example…)
- Line charts:
    - `geom_line()`: create a line that connects points (see example…);
    - `geom_smooth()`: add a regression line (see …);
    - `geom_hline()` (resp. `geom_vline()`): add a horizontal (resp. vertical) line using `yintercept` (resp. `xintercept`);

– `geom_segment()`: draw a segment going from ($x;y$) to ($xend;yend$)[5].

- Bar charts:

  – `geom_col()` and `geom_bar()`: produce bar-charts by either using the raw values, or by computing the frequencies first (see example...);
  – `geom_histogram()` and `geom_freqpoly()`: work in a similar way as `geom_bar()` except that it divides the x axis into bins before counting the number of observation in each bin and either represent it as bars (`geom_histogram`) or lines (`geom_freqpoly()`).

- Distribution:

  – `geom_density()`: build the density plot;
  – `geom_boxplot()`: build the well-known boxplot;
  – `geom_violin()`: application of `geom_density()` displayed in `geom_boxplot()` fashion.

- Text and Labels:

  – `geom_text` and `geom_label`: add text to the graph (see example...);
  – the package `{ggrepel}` provides alternative functions (`geom_text_repel()` and `geom_label_repel()`) that re-position labels to avoid overlapping (*repel* stands for *repulsive*).

- Rectangles[6]:

  – `geom_tile()`, `geom_rect`: create area either using its center point (`geom_tile()`) or its four corner (`geom_rect()`) defined by `xmin`, `xmax`, `ymin`, and `ymax` (see example...);
  – `geom_raster()`: high performance alternative to `geom_tile()`/`geom_rect` where all the tiles have the same size.

Besides `geom*()`, a lot of graphical parameters can further be controlled. This includes of course the `theme()` and the `aes()`:

- For pre-defined themes, see example;
- `axis` parameters including its title (`axis.title`), text (`axis.text`), ticks (`axis.ticks`), line (`axis.line`), and all their sub-levels.
- `legend` parameters including its position (`legend.position`), direction (`legend.direction`), its text (`legend.text`, `legend.title`), the design of the box (`legend.box`, `legend.background`) etc.
- `panel` parameters including its background (`panel.background`), the grid lines (`panel.grid`), the border (`panel.border`), etc.

---

[5]This function can also be used to draw arrows through the parameter `arrow` and the function of that same name `arrow()`.

[6]In Sensory and Consumer science, this will often be used for building surface plot responses (e.g. external preference map), and hence is associated to `geom_contour()` to show the different lines.

- `plot` parameters including the different titles (`plot.title`, `plot.subtitle`, `plot.caption`), the background (`plot.backgorund`), etc.

Most of these parameters can be controlled at different levels of granularity:

- overall, e.g. `panel.grid`;
- more detailed, e.g. `panel.grid.major` and `panel.grid.minor`;
- most detailed, e.g. `panel.grid.major.x`, `panel.grid.major.y`, etc.

Depending whether the option to modify is some text, a line, or a rectangle, `element_text()`, `element_line()`, or `element_rect()` would be respectively used to control them. These functions provide general (e.g. `color`) as well as specific options (e.g. `family` and `face` for text, `linetype` for lines etc.) to each type.

Note that if some elements should be left blank, `element_blank()` can be used regardless of the nature of the element.

Let's illustrate these concepts using our previous graph stored in `line_p`. Here, the goal is to remove the grid line, to replace the x and y axis lines by arrows, and to re-position the axis titles to the far end of the axis so that it is next to the arrow head.

```
line_p +
  theme(panel.grid=element_blank(),
        panel.border=element_blank(),
        axis.line=element_line(arrow = arrow(ends = "last", type = "closed")),
        axis.title=element_text(hjust=1))
```

Similarly to the theme, aesthetics can also be adjusted. In previous examples, the x-axis in `bar_p` as adjusted by setting limits, providing breaks and replacing the values by certain labels using `scale_x_continuous()`. Most aesthetics parameters can be controlled by equivalent functions for which the name is using the following structure `scale_*nameaes*_*typescale*`, and where:

- *nameaes* corresponds to any aesthetics including x, y, `colour` or `fill`, `alpha`, etc.
- *typescale* corresponds to the type of scale, where it is `continuous`, `discrete`, or `manual` amongst others.

Such functions fully control how the corresponding aesthetic should behave, by providing the correspondence between a variable level and its color for instance. In the graph saved in `bar_p`, remember that we filled in the bar chart using the product information. Let's imagine that we are particularly interested in biscuit P3, and want to compare it to the rest of the biscuits. We propose to make P3 stand out by filling it in orange, and by setting all the other biscuits in the same gray tone. Such procedure can be done using `scale_fill_manual()`.

```
bar_p +
  scale_fill_manual(values=c("P1"="gray50", "P2"="gray50", "P3"="darkorange", "P4"="gra
                             "P6"="gray50", "P7"="gray50", "P8"="gray50", "P9"="gray50"
```

When multiple aesthetics are being used, the legend might become overwhelming or redundant. It is possible to turn off some of these visuals within the `scale_*()` functions, or by using `guides()` and by setting `nameaes='none'` as shown in the `line_p` example.

5.3.4   Miscealleneous

5.3.4.1   Structuring the axis

By default, `ggplot()` generates plot that fits the data and that fits within the output screen. This means that some graphics might not be perfectly representing the data due to some distortion. In a previous example (`line_p`), the dimensions were made comparable through `coord_fixed()`.

Other transformation can be performed. For instance, the graphic can be transposed using `coord_flip()` as in the following example:

```
bar_p + coord_flip()
```

To conclude this section, and summarize most concepts presented in this chapter, let's introduce the well-known spider plots. The use of such plots are quite polarizing amongst analysts and the reason of this choice here is purely educational, as 1. there are no pre-defined options in `{ggplot2}` that provides such charts, and 2. they present some interesting challenges.

Let's start with deconstructing a spider-plot: a spider plot is a line chart presented in a circular way. So let's start with building a line chart of our sensory profiles (the means are considered here). For more clarity, only two of the samples (`P03` and `POpt`) are represented.

```
# Constructing the Mean Table
sensory_mean <- sensory %>%
  pivot_longer(Shiny:Melting, names_to="Variables", values_to="Scores") %>%
  mutate(Variables = fct_inorder(Variables)) %>%
  group_by(Product, Variables) %>%
  summarize(Mean = mean(Scores)) %>%
  ungroup() %>%
  filter(Product %in% c("P03", "POpt"))

# Building the Line Chart
```

```
spider_line <- ggplot(sensory_mean, aes(x=Variables, y=Mean, colour=Product, linetype=Product))+
  geom_point(pch=20, cex=2)+
  geom_line(aes(group=Product), lwd=1)+
  theme_minimal()+
  xlab("")+
  scale_y_continuous(name="", labels=NULL, limits=c(0,50))+
  scale_colour_manual(values=c("P03"="darkorange", "POpt"="grey50"))+
  scale_linetype_manual(values=c("P03"="solid", "POpt"="dashed"))
```

Next step is to represent this line chart in a circular way. This can be done using `coord_polar()`:

```
spider_line + coord_polar()
```

This already looks like a spider plot! However, a closer look at it highlights a point that needs improvement: There is no connection between the last attribute (`Melting`) and the first one (`Shiny`).

To counter this, the following twofold solution is proposed:

1. Associate each attribute to its position (e.g. `Shiny` is 1, `External color intensity` is 2, until `Melting` which would be 32);
2. Duplicate the last attribute (`Melting`) and associate it to position 0.

```
var <- levels(sensory_mean$Variables)
sensory_mean_pos <- tibble(Variables = c(var[length(var)], var),
                   Position = 0:length(var)) %>%
  full_join(sensory_mean, var_pos, by="Variables")
```

The previous graph is then rebuilt by forcing the position of the attributes on the x-axis using `Position` (`Variables` is used for the labels). Here position 0 is voluntarily omitted (`breaks = 1:length(var)` and `labels = var`), meaning that only the labels going from 1 to the last variable are being showed. However, the x-axis is forced to go from 0 to the number of attributes (`limits = c(0, length(var))`).

```
spiderplot <- ggplot(sensory_mean_pos, aes(x=Position, y=Mean, colour=Product, linetype=Product))
  geom_point(pch=20, cex=2)+
  geom_line(aes(group=Product), lwd=1)+
  theme_minimal()+
  scale_x_continuous(name="", breaks=1:length(var), labels=var, limits=c(0,length(var)))+
  scale_y_continuous(name="", labels=NULL, limits=c(0,50))+
  scale_colour_manual(values=c("P03"="darkorange", "POpt"="grey50"))+
  scale_linetype_manual(values=c("P03"="solid", "POpt"="dashed"))+
  coord_polar()
```

By using this trick, the connection between the first and last attributes is established.

5.3.4.2   Combining plots

When multiple plots should be generated using the same pattern on subset of data, it is possible to generate them automatically using `facet_wrap()` or `facet_grid()`. The difference between these two functions rely in the number of variables to use for the split: In `facet_wrap()`, the graphics are *vectorized*, meaning that each element of the split is represented independently. For `facet_grid()` however, the graphics is represented in a matrix, meaning that two blocks of split variables are required, one defining the columns and one the rows.

An example of `facet_wrap()` is provided in (ref data analysis).

For these two functions, the parameter `scales` is particularly interesting as it allows each separate graph to use its own axis scales (`free` or individually using `free_x`/`free_y`) or not (`fixed`).

To go further, consider also the function `facet_trelliscope()` from the `{trelliscopejs}`. This function generates the same type of graphs as `facet_wrap()` or `facet_grid()` with some powerful twists. After generating the plots, they are still editable thanks to an interactive menu that controls for the *Grid*, *Labels*, *Filter*, and *Sort*. For example, the number of plots to show per row/column can be adjusted, tables with descriptive statistics (e.g. mean, minimum, maximum) can be added under each graph, etc. Moreover, readers that are familiar with the interactivity of `{plotly}` can make great use of it through the `as_plotly = TRUE` parameter, which is then applied to each individual graph!

Such procedure is very handy to produce multiple graphs all at once…when the data allow it. When multiple plot are being generated separately (using different data set, or producing different types of plots), it can still be relevant to combine them all in one. To perform such collage, the package `{patchwork}` becomes handy thanks to its simplicity and flexibility.

`{patchwork}` is a package that allows combining `ggplot()` graphs using *mathematical* operations. To add two elements next to each others, the `+` sign is used. To add two elements on top of each others, they should be separated using `/`. This operation can be combined with `()` to generate fancier collage.

Let's illustrate this by creating a plot with on the left side `spiderplot`, and on the right side `bar_p` on top of `line_p`.

```
library(patchwork)
```

```
p = spiderplot + (bar_p / line_p)
p
```

A general title can be added, as well as tag levels (handy for publications!) using `plot_annotation()`.

```
p + plot_annotation(title = "Example of 'ggplots' I've learned today", tag_levels='a')
```

5.3.5 Few Tips and Tricks

5.3.5.1 Combining data transformation and {ggplot2} grammar

Both the `{tidyverse}` and `{ggplot2}` are using pipes to combine lines of code or layers. However, the pipes themselves are defined differently since `{maggritr}` uses `%>%` whereas `{ggplot2}` uses `+`. It is however possible to combine both systems one after each other, just remember to switch from `%>%` to `+` as you transition from data transformation/tidying to building your graph (see example…).

5.3.5.2 Ordering elements in a plot

When building a graph using a categorical variables, `{ggplot2}` tends to represent the different levels in alphabetical order, especially if the variable is defined as character. Such situation can make the graph more difficult to read, as the categories may not be presented in a logical order (e.g. fall, spring, summer, winter instead of spring, summer, fall, winter).

To ensure that the elements are in the right order, either consider transforming the variables into factor (using `factor()` by indicating the levels order of your choice, or through `fct_inorder()` to keep the order from the file, see example…) or by using a position variable as in the `spiderplot` example. The former option also works for ordering elements in the legend.

If the order of the elements should be changed *within the charts* (simple changes such as reverting the order), it can be done directly within the `geom_*()` function. This is for instance the case with stacked bar chart, in which the order may be reverted using the parameter `position = position_fill(reverse = TRUE)` for instance (suggesting here that the split was defined through `fill` in `aes()`).

5.3.5.3 Fixing overlapping axis text

When `ggplot()` are being built using categorical variables, the labels used on the x-axis are often overlapping (some of the labels being then unreadable).

A first good/easy solution consists in reducing the size of the label, and/or to shortening them as long as it does not affect its readability/clarity. However, this might not always be possible or sufficient, and other adjustment are required. Let's use `spider_line` as illustration to show three possible solutions.

The first option consists in using `theme()` and rotating the labels (here at 45 degrees, but use 90 degrees to get the names vertically). Note that by default, `ggplot()` center the labels: to avoid having them crossing the x-axis line, they are being left-centered using `hjust=1`:

```
spider_line +
  theme(axis.text.x = element_text(angle=45, hjust=1))
```

A second option consists in dodging one every two labels along the x-axis. This option works fine, especially when the labels are not too long. In our example unfortunately, some overlap can still be seen. Note that this option is accessible within `scale_x_discrete()`, not within `theme()` as we would expect:

```
spider_line +
  scale_x_discrete(guide = guide_axis(n.dodge = 2))
```

Last option consists in transposing the graph using `coord_flip()`. This solution works well since labels on the y-axis are written horizontally. However, this option is not always suitable due to conventions: if it is recommended for bar charts, it may not be for line charts for instance.

5.3.5.4   Exporting graphs

There are various ways to save or export `ggplot()` charts. To save these plots to your computer in various formats (e.g. png, pdf, etc.), `ggsave()` can be used. By default, `ggsave()` exports the last plot built and saves it in the location defined by `filename`, in the format defined by `device` (additional information regarding the width, height, dpi etc. can also be configured).

For instance, the `spiderplot` generated earlier[7] can be saved as following:

```
ggsave(filename="spiderplot.png", plot=spiderplot, device="png")
```

As an alternative, `ggplot()` graphs can also be exported in PowerPoint or Word through the `{rvg}` package (see example).

---

[7]Since `spiderplot` is not the last plot generated, it needs to be defined in `plot`.

5.3.5.5 Additional libraries

`{ggplot2}` is a very powerful tool for data visualization. By default, it offers a very large variety of possibilities, which should cover most situations that you would encounter. If not, please search as you will most likely find ggplot extensions in alternative packages.

To help you further, here is a non-exhaustive list of relevant packages:

- `{ggcharts}`: This package provides nice and clear alternatives to some `{ggplot2}` options through simple functions in one line of code, including `bar_chart()`, `line_chart()`, `lollipop_chart()` and `dumbbell_chart()` just to name a few.
- `{graffify}`: This package extends `{ggplot2}` by providing nice and easy functions to help data visualization and linear models for ANOVA. For example, it generates through one function bar chart with error bars through `plot_scatterbar_sd()`, or simultaneous boxplot and scatter plot through `plot_scatterbox()`.
- `{factoextra}`: Although the latest version of `{FactoMineR}` also generates its graphs in `{ggplot2}` by default, `{factoextra}` is a great extension as it is easy to use and provides a wide variety of options to customize your plots.
- `{ggcorrplot}`: There are many packages that propose to visualize graphically tables of correlations, however we particularly like this one for its simplicity and its flexibility.
- `{ggwordcloud}`: It is a great package for building word-clouds as it provides a large degree of control. With this package, the words can either be positioned randomly, by matching a pre-defined shape, etc. But more interestingly, words can also be positioned semi-randomly, hence giving more interpretation power to the final results (for more information, please visit https://lepennec.github.io/ggwordcloud/link).
- `{ggraph}`: This package provides neat solutions to build network visualization in `{ggplot2}`.
- `{performance}`: This package provides pre-defined graphs that allow you evaluating the quality of your models through the single `check_model()` function. See also `{ggside}` if you want to print on the margin of your regression plot the marginal distributions (or density plot) of your different categories.

To learn more about `{ggplot2}` basics, we recommend two additional source of information:

- `{esquisse}`: After loading the package, run the function `esquisser()`. This command opens a window in which you can select your data set (the data set should be available within you R environment), the type of

plot to build, and all the relevant information to build your plot (which variable to be used as x-axis, y-axis, etc.) through an easy-to-use interface. Ultimately, the graph is being generated, but more importantly, the code used to generate the plot is provided. This is hence an educational tool to learn build graphs with ggplot.

- https://www.data-to-viz.com/link provides a wide gallery of graphics sorted by the type of data that you have. Each graphic proposed is illustrated with an example provided in R (often in `{ggplot2}`) and in Python. This website is hence inspirational and educational both at the same time!

Chapter 6

Automated Reporting

6.1 What and why Automated Reporting?

Effective communication of results is amongst the essential duties of sensory scientists...and so is the data collection, data preparation, data analysis etc. Worst, the sometimes tedious mechanics of report production together with the sheer volume of data that many scientists must process combine to make reporting design and nice story-telling an afterthought in too many cases. Although this should preferably not be happening, it is necessary sometimes as presentation deadlines approach and time becomes limited.

Add to this work-load some last-minute changes due to either a change in the data, in the analysis, or maybe an error (e.g. copy/paste the wrong column, etc.) you have just been detected...how can we be sure that the latest version of the report is fully up-to-date, and that all the results/conclusions are correct? As the statistical software (e.g. R) is often separated from the reporting tool (e.g. Microsoft Office), it is easy to miss to transfer updated statistical outputs (e.g. values, tables, figures, etc.) to the report, hence creating inconsistencies.

Let's consider another scenario, in which all the preliminary steps have been successfully done: you are now presenting your report to your manager or clients, and they come with a question such as: "Can we deep-dive into the results by looking at a particular group of consumers (e.g. gender split, or cluster split)?" Do you feel like going through the entire process again?

How would you feel if we would tell you that there is a way to build your report while running the analysis, by using the same script file? This means that in few clicks, say after updating the data to analyze (e.g. filter to the target group of consumers only), your report gets automatically re-generated with all the new updated results. Will you be interesting in such approach?

Such solution seems ideal since it increases efficiency while reducing errors due

to manual processing of results. More importantly, this gain in time and effort allow you designing nicer slides and building a better story.

6.2 Integrating reports within analyses scripts

In this section, let's integrate our report building process within our data analysis. By doing so, we do not focus on building a story yet. Instead, we improve our way of working by exporting directly all the statistical outputs that could[1] be useful for our future story-telling. By doing so, we increase efficiency (especially if code can then be re-used for other studies) by *killing two birds with one stone*: We simultaneously run our analysis, create usable content for our final presentation, while reducing errors due to manual processing.

Since Microsoft Office is often the tool used for sharing results, we will focus our attention in exporting results to Excel, PowerPoint, and Word.

Note that in this chapter on automated reporting, some results (tables, figures) that are being created in one of the first section will be re-used in subsequent sections. In case you do not read this chapter linearly, you might get errors as you might be calling objects that do not exist yet. We advise you to read through the previous sections, find the script where the elements (table or figure) is being generated, run it, and resume your read.

6.2.1 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 offer the possibility to export data and results in Excel, while most data analysis software accept 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 spreadsheets. It is even less a surprise that R provides multiple solutions to import/export results from/to Excel.

For importing Excel files, we have already presented the package `{readxl}` among others (see REF). For exporting results, two complementary packages (yet again, among others!) in terms of ease of use and flexibility in the outcome are proposed: `{writexl}` and `{openxlsx}`.

As its name suggests, `{writexl}` is dedicated to exporting tables to Excel through the `write_xlsx()` function. Its use is very simple as it only takes as inputs the table (or list of tables)[2] to export to the file specified in the `path` parameter.

---

[1] We say *could* as we are in a process of mass-exportation of results, most of them being used for building the story although they would not be kept in the final deck.

[2] List of tables will generate multiple sheets within the same spreadsheet, one table being placed in each sheet.

Let's illustrate this by using our *Sensory Profile.xlsx* file: Let's imagine that we would like to reduce our data set by only considering products that are high in Protein:

```r
library(tidyverse)
library(readxl)
library(writexl)
library(dplyr)

file_path <- file.path("data", "Sensory Profile.xlsx")

product_info <- read_excel(path  = file_path,
                           sheet = "Product Info",
                           range = "A1:D12",
                           col_names = TRUE)

# Selecting Products with High Protein
high_prot <- product_info %>%
  filter(Protein %in% "High") %>%
  pull(Product)

# Filter Data to only keep Products with High Protein
high_prot_data <- read_xlsx(path = file_path,
                  sheet = "Data") %>%
  filter(Product %in% high_prot)

# Exporting Table to Excel
write_xlsx(data,
           path = file.path("output", "High Protein Products only.xlsx"),
           col_names = TRUE)
```

The export of tables using the `{writexl}` package is easy, yet simplistic as it does not allow formatting the tables (except for some minor possibilities for the header), nor does it allow exporting multiple tables within the same sheet. 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 object (e.g. `wb`) using the `createWorkbook()` function. We can add worksheets to `wb` through the `addWorksheet()` function.

```r
library(openxlsx)

# Create workbook object
wb <- openxlsx::createWorkbook()
```

```r
# Add a new worksheet
addWorksheet(wb, sheetName = "Mean", gridLines = FALSE)
```

Note that `addWorksheet()` allows controlling further the appearance of the worksheet: * show/hide grid lines using `gridLines`; * color the 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`).

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.

Let's imagine we want to compute the sensory profiles of the products, and we want to export that into Excel. Rather then simply exporting the results, we want to customize the output a little bit by applying the Excel style named *TabelStyleLight9*:

```r
# Creating the Sensory Profiles with some Product Information
p_info <- read_xlsx(file_path, sheet = "Product Info") %>%
  dplyr::select(-Type)

sensory <- read_xlsx(file_path, sheet="Data") %>%
  inner_join(p_info, by="Product") %>%
  relocate(Protein:Fiber, .after=Product)

senso_mean <- sensory %>%
  pivot_longer(Shiny:Melting, names_to="Attribute", values_to="Score") %>%
  dplyr::select(-Judge) %>%
  pivot_wider(names_from=Attribute, values_from=Score, values_fn=mean)


# Exporting the Results to Excel
writeDataTable(wb,
               sheet = "Mean",
               x = senso_mean,
               startCol = 1,
               startRow = 1,
               colNames = TRUE, rowNames = FALSE,
               tableStyle = "TableStyleLight9")

openXL(wb)
```

At any time, you can visualize the Excel file that is being produced without exporting it yet using `openXL()`. This function comes very handy as it allows you checking that the output looks like what you would wish for.

As can be seen, `writeData()` and `writeDataTable()` give us a lot of control on our export. For instance, we can: * control where to print the data by using `startRow` and `startCol` (or alternatively xy: `xy = c("B",12)` prints the table starting in cell B12), hence allowing exporting multiple tables within the same sheet; * include the row names and column names through `rowNames` and `colNames`; * format the header using `headerStyle` (incl. color of the text and/or background, font, font size, etc.); * apply a specific style to our table using `tableStyle`; * shape the borders using predefined solutions through `borders`, or customizing them with `borderStyle` and `borderColour`; * add a filter to the table using `withFilter`; * convert missing data to "#N/A" or any other string using `keepNA` and `na.string`.

Rather than using some pre-defined formatting as was the case with `tableStyle`, let's consider some more advanced options in which we control (almost) everything. Let's start with setting up the formatting style we would like to apply:

```
# 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" = "0.0")

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

# Header Style (blue background, top/bottom borders, text centered/bold)
headSty <- createStyle(fgFill = "#DCE6F1",
                       border = "TopBottom",
                       halign = "center",
                       textDecoration = "bold")
```

Note that many more formatting options can be configured through:

- `options()` to pre-define number formatting, border colors and style, etc.;
- `modifyBaseFont()` to define the font name and font size;
- `freezePane()` to freeze the first row and/or column of the table;
- `createStyle()` to pre-define a style, or `addStyle()` to apply the styling to selected cells;
- `setColWidths` to control column width;
- `conditionalFormatting()` to format cells based on pre-defined conditions (see next for an example).

Let's export again the sensory profiles in a second sheet after applying these formatting:

```r
addWorksheet(wb, sheetName = "Mean (manual formatting)", gridLines = FALSE)

# Freeze first row and first column
freezePane(wb, sheet=2, firstRow=TRUE, firstCol=TRUE)

# Export the data using writeData
writeData(wb,
          sheet =2,
          x = senso_mean,
          startCol = 1,
          startRow = 1,
          colNames = TRUE, rowNames = FALSE,
          headerStyle = headSty)


openXL(wb)
```

You'll notice that the same table is now presented in a fairly different way.

Let's now consider a third export of the sensory profiles, with an additional twist: for a given variable (i.e. column), the value is colored in red (resp. blue) if it is higher (resp. lower) than its mean. To do so, we need to use conditional formatting.

Let's start with creating two pre-defined parameters called `pos_style` (red) and `neg_style` (blue) using `createStyle()`that we will use to color the different cells. Let's also compute the overall mean per attribute.

```r
# Styles for conditional formatting
pos_style <- createStyle(fontColour = "firebrick3", bgFill = "mistyrose1")
neg_style <- createStyle(fontColour = "navy", bgFill = "lightsteelblue")

# Compute the overall mean
overall_mean <- senso_mean %>%
  summarize(across(where(is.numeric), mean))
```

Let's then create a new worksheet in which we print the data of interest:

```r
# Add a new worksheet
addWorksheet(wb, sheetName = "Conditional Formatting", gridLines=FALSE)

# Write table: note that
writeDataTable(wb,
               sheet = 3,
               x = senso_mean,
               startCol = 1,
```

```
            startRow = 1,
            colNames = TRUE, rowNames = FALSE)
```

Finally, we color the cells according to the rules that was defined earlier. To do so, the decision whether `pos_style` or `neg_style` should be used is defined by the `rule` parameter from the `conditionalFormatting()`[3] function.

```
# Adding formatting to the second column
for (v in 1:ncol(overall_mean)){

  conditionalFormatting(wb,
                        sheet = 3,
                        cols  = v + 3,
                        rows  = 1 + 1:nrow(senso_mean),
                        rule  = paste0(">", overall_mean[1,v]),
                        style = pos_style)

  conditionalFormatting(wb,
                        sheet = 3,
                        cols  = v + 3,
                        rows  = 1 + 1:nrow(senso_mean),
                        rule  = paste0("<", overall_mean[1,v]),
                        style = neg_style)

}

openXL(wb)
```

Few things should be noted:

- We want to run this for each sensory attribute, hence the `for` loop that goes from 1 to the number of columns stored in `overall_mean` (`overall_mean` only contains the overall mean scores for the sensory attributes);
- `senso_mean` however contains 3 extra columns: `Product`, `Protein`, and `Fiber` hence the parameter `cols = v + 3`;
- We apply the formatting to all the rows except the header, hence `rows = 1 + 1:nrow(senso_mean)`;
- Finally, we apply `pos_style` (resp. `neg_style`) if the value is larger (resp. lower) than the overall mean for that attribute using `rule = paste0(">", overall_mean[1,v])`.

---

[3]In `conditionalFormatting()`, you can specify to which `rows` and `cols` the formatting applies. In this example, `cols` takes 2 because the first column contains the row names.

Note that once the spreadsheet is complete, we create the file 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`.

```
saveWorbook(wb, file="temp/excel export.xlsx")
```

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

6.2.2   PowerPoint

6.2.2.1   Creating a PowerPoint Deck

Throughout the years, PowerPoint became one of the main support for presenting results, whether it is in academia, in conference, or in companies. It is hence important for us to show how to generate reports in PowerPoint from R directly. This can be done in different ways, yet we propose to use the `{officer}`[4] package, as its application is vast while still remaining easy to use.

```
library(officer)
```

With `{officer}`, the procedure starts with creating a PowerPoint object (`pptx_obj`) using the `read_pptx()` function.

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

A blank Deck is by default set up with the *Office Theme*. To use a custom theme, we must create a Deck using the PowerPoint software and use it as input. Let's import the *intergral.pptx* template:

```
pptx_obj2 <- read_pptx(file.path("data", "templates", "intergral.pptx"))
```

To inspect the content of the template, we use `layout_summary()`:

```
pptx_obj %>%
  layout_summary()

pptx_obj2 %>%
  layout_summary()
```

---

[4]It should be noted that {officer} contains conflicting function names with other packages we use (e.g. `read_xlsx()`). To ensure you use the right function, call the function from the package of interest (e.g. `readxl::read_xlsx()`).

As can be seen by `layout_summary()`, the template imported (also called **master**, which is defined here as *Office Theme*) proposes 7 types of slides including *Title Slide*, *Title and Content*, *Section Header*, *Two Content*, *Comparison*, *Title Only*, and finally *Blank* (the *intergral.pptx* template has 11 different types of slides). Each of these slides present some pre-defined properties (e.g. a box for text of tables/images, a header etc.). Let's look at the properties of *Title and Content* using `layout_properties()`

```
pptx_obj %>%
  layout_properties() %>%
  filter(name == "Title and Content")
```

This code provides more details about the different sections, their identifier and position of the slide. This information will be particularly useful when we will want to export certain information in some particular areas.

Unfortunately, `{officer}` does not provide a function similar to `openxlsx::openXL()` which allows visualizing the file that is currently being build. Instead, we need to save the document directly on the disk using the `print()` function, which takes as entries the PowerPoint file to export (here `pptx_obj`) and its output location.

6.2.2.2   Adding/Moving/Removing Slides

With `{officer}`, various actions can be done on the slides. The first logical action consists in adding a new slide to a presentation, in which we will later on write some text, tables, figures, etc. Such action can be done using `add_slide()`, in which we inform the type of slide to add, and from which master:

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

This code will add a *Title and Content* slide to your deck.

Additional operations on the slides themselves can be done. In particular, you can re-organize your deck by changing the orders of your slides using `move_slide()`, delete slides that are no longer needed through `remove_slide()`, or modify a pre-existing slides by making it active using `on_slide()` (by default, the last slide created is the active one).

6.2.2.3   Positioning Information to the Slide

On a given slide, any type of content (text, graph, table, etc.) can be exported. To do so, we need to inform where to write what.

As we will see in the next sections, the *what* can be any R element including simple text, tables, figures, etc. So let's ignore it for the moment, and let's focus on the *where*. To inform where to print elements on the slide, the function `ph_with()` (*ph* stands for *placeholder*) is used. In practice, `ph_with()` comes with the parameter `location`, which takes as input a *placeholder location object* pre-defined by the function `ph_location()` or one of its derivative, one of the most useful one being `ph_location_type()`. To do so, simply provide the name stored in the column type from the `layout_properties()` output presented before, as following:

```r
my_data <- c("My functions are:", "ph_with", "ph_location_type")

pptx_obj <- pptx_obj %>%
  ph_with(value = "My first title", location = ph_location_type(type = "title")) %>%
  ph_with(value = my_data, location = ph_location_type(type = 'body'))
```

This will add a title ("My first title") and the text stored in `my_data` to the *body* of the slide (*Title and Content*) created previously.

Other pre-defined alternatives to `ph_location()` include:

- `ph_location_fullsize()` to produce an output that covers the entire slide;
- `ph_location_left()` and `ph_location_right()` to write in the left/right box in *Two Content* types of slide;
- `ph_location_labe()` is similar to `ph_location_type()` except that it uses the label rather than the type.

To gain full control of the exact position of the element to print, `ph_location()` is used. It allows specifying exact positions for content (for left/top/width/height, units are inches):

```r
my_data <- "My new text positioned using ph_location()"

pptx_obj <- pptx_obj %>%
  add_slide(layout = "Title and Content", master = master) %>%
  ph_with(value = my_data, location = ph_location(left = 2, top = 2, width = 3, height
```

To visualize the different steps done so far, let's save the results on our computers:

```r
print(pptx_obj, "temp/my powerpoint export.pptx")
```

6.2.2.4 Exporting Text

In the previous section, we already exported text to slides. Let's go a bit deeper in the process by also showing how to format the text.

By default, each new text item added to a PowerPoint via {officer} is a paragraph object. To further format the paragraph, three main functions are being used: * fpar() (*formatted paragraph*) creates the paragraph; * ftext() (*formatted text*) allows editing the text before pasting into paragraphs. ftext() requires a second argument called prop which contains the formatting properties; * block_list() allows us to wrap multiple paragraphs together.

Let's go through an example to illustrate the use of these functions:

```r
my_prop <- fp_text(color = "red", font.size = 14) # Formatting option
my_text <- ftext("First Line in Red", prop = my_prop) # First line of text, formatted

my_par <- fpar(my_text) # text into a paragraph
blank_line <- fpar("") # other empty paragraph to introduce an empty line

my_par2 <- fpar("Second Line") # second line of text, unformatted
my_list <- block_list(my_par, blank_line, my_par2) # Final block with the two lines of text separ

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

print(pptx_obj, target = "temp/my powerpoint export.pptx")
```

This add an additional slide to our previous PowerPoint deck with our formatted text.

Last element of formatting to consider is the hierarchy in bullet points. Let's add a slide containing three bullet points with a hierarchy so that the 1st and 3rd lines are primary points, and the second line is a secondary point. Such hierarchy can be informed using the level_list parameter, which informs the hierarchy of each element:

```r
text1 <- fpar("FIRST SENTENCE")
text2 <- fpar("second sentence")
text3 <- fpar("THIRD SENTENCE")
my_data <- block_list(text1, text2, text3)

pptx_obj <- pptx_obj %>%
  add_slide(layout = "Title and Content", master = master) %>%
  ph_with(value = my_data, level_list = c(1,2,1), location = ph_location_type(type = 'body'))
```

```r
print(pptx_obj, target = "temp/my powerpoint export.pptx")
```

6.2.2.5   Exporting Tables

Now we know how to export formatted text to slides, let's export tables. This can easily be done by rendering a data frame rather than text as `ph_with()` accepts it and renders it in a default format. Let's use a subset of `senso_mean` for illustration:

```r
ft_data <- senso_mean %>%
  dplyr::select(Product, Salty, Sweet, Sour, Bitter) %>%
  mutate(across(where(is.numeric), round, 2))

pptx_obj <- read_pptx() %>%
  add_slide(layout = "Title and Content", master = master) %>%
  ph_with(value = ft_data, location = ph_location_type(type = "body"))

print(pptx_obj, target = "temp/my powerpoint export.pptx")
```

Although this solution works fine, it does not allow formatting the table as much as we would want. Instead, we prefer to use another package called `{flextable}` which was developed by the same author as `{officer}` (both packages being complementary).

6.2.2.5.1   Introduction to flextable

With `{flextable}`, the procedure starts with creating a *flextable* object (here `ft_table`) using the `flextable()` function.

```r
library(flextable)

ft_table <- ft_data %>%
  arrange(Product) %>%
  flextable()

print(ft_table)
```

By printing the table, it opens in the Rstudio viewer as an html file. This table can then be customized in various different ways through various functions:

- `align()` and `rotate()` controls for the text alignment and its rotation;
- `bold()` and `italic` writes the text in bold or italic;
- `font()` and `fontsize` controls the font type and the size to use;

- `color()` and `bg()` allows changing the color of the text and of the background.

All these functions require informing the rows (parameter `i`) and the columns (`j`) as well as the part (`"body"`, `"header"`, `"footer"`, or `"all"`) to modify.

Additionally, further formatting can be applied to the table itself through the following functions:

- `height()` & `width()` control for the row height and column width;
- `border_outer()`, `border_inner()`, `border_inner_h()` & `border_inner_v()` help design the table by adding borders;
- `autofit()` and `padding()` are used to control the final size of the table.

For illustration, let's apply some of these functions to `ft_table`:

```r
ft_table <- ft_table %>%
  fontsize(size = 11) %>%
  # Formatting the header
  font(fontname = "Roboto", part = "header") %>%
  color(color = "white", part = "header") %>%
  bold(part = "header") %>%
  align(align = "center", part = "header") %>%
  bg(bg = "#324C63", part = "header") %>%
  # Formatting the body
  font(fontname = "Calibri", part = "body") %>%
  bg(i = 1:nrow(ft_data), bg = "#EDEDED") %>%
  # Formatting the last row of the table
  bold(i = nrow(ft_data), j = 1:ncol(ft_data)) %>%
  italic(i = nrow(ft_data), j = ~Product + Salty + Sweet + Sour + Bitter) %>%
  color(i =  nrow(ft_data), j = ~Sour, color = "red") %>%
  color(i =  nrow(ft_data), j = ~Sweet, color = "orange") %>%
  autofit()

# Set up the border style
my_border <- fp_border(color = "black", style = "solid", width = 1)

ft_table <- ft_table %>%
  border_outer(part = "all", border = my_border) %>%
  border_inner(part = "body", border = fp_border(style = "dashed")) %>%
  width(j = 1, width = 1.2) %>%
  height(i = 12, height = 1)

print(ft_table)
```

This is just an overview of the most relevant and used functions in `{flextable}`, yet there are more possibilities. To go further, you can also consider the following functions (amongst many more):

- `merge()` merges vertically or horizontally cells with the same content;
- `compose()`, `as_chunk()`, and `as_paragraph()` works hands in hands to create more complex text formatting (e.g. sentence with parts of the text colored differently, or with sub/superscript);
- `style()` applies a set of formatting properties to the same selection of the rows/columns.

Finally, to export a flextable to a PowerPoint deck, simply export it as we have seen before:

```r
pptx_obj <- pptx_obj %>%
  add_slide(layout = "Title and Content", master = "Office Theme") %>%
  ph_with(value = ft_table, ph_location(left = 2, top = 2, width = 4))

print(pptx_obj, target = "temp/my powerpoint export.pptx")
```

For more details on using **{flextable}** see https://davidgohel.github.io/flextable/.

6.2.2.6 Exporting Plots

The last type of R outputs we want to export to PowerPoint is figures. Before showing how to export them, let's build a simple bar chart from `senso_mean` using {ggplot2}:

```r
chart_to_plot <- senso_mean %>%
  dplyr::select(Product, Salty, Sweet, Sour, Bitter) %>%
  arrange(Product) %>%
  pivot_longer(Salty:Bitter, names_to = 'Attribute', values_to = 'Value') %>%
  ggplot(aes(x = Product, y = Value, fill = Attribute)) +
  geom_col(position = 'dodge')+
  xlab("")+
  theme_bw()
```

To export any `ggplot2` object to PowerPoint, the package `{rvg}` is required. This package provides two graphics devices that produces Vector Graphics outputs in `DrawingML` format for Microsoft PowerPoint with `dml_pptx()` and for Microsoft Excel with `dml_xlsx()`, meaning the the graphics is being 'rebuilt' in PowerPoint or Word. To simplify, the generic `dml()` function is used[5].

---

[5]Depending on the output format, the corresponding function is being called.

```r
library(rvg)

pptx_obj <- pptx_obj %>%
  add_slide(layout = "Title and Content", master = "Office Theme") %>%
  ph_with(value = dml(ggobj = chart_to_plot, editable = TRUE),
          location= ph_location_type(type = 'body'))

print(pptx_obj, target = "temp/my powerpoint export.pptx")
```

For simple graphics (such as line chart, bar charts, etc.), the {mschart} package creates the graphs directly in PowerPoint or Word. These graphics have then the advantage to be interactive. In this case, the ggplot2 graphs are not needed: instead, we use functions such as ms_barchart() to produce them.

```r
library(mschart)

mydata <- senso_mean %>%
  dplyr::select(Product, Salty, Sweet, Sour, Bitter) %>%
  arrange(Product) %>%
  pivot_longer(Salty:Bitter, names_to = 'Attribute', values_to = 'Value')

# Building the barchart using ms_barchart()
my_barchart <- ms_barchart(data = mydata,
                           x = "Product",
                           y = "Value",
                           group = "Attribute")

# The chart is a Powerpoint native object and can be viewed using the preview option in print
print(my_barchart, preview = TRUE)

# To add the object to a powerpoint slide we can use the officer's native ph_with
pptx_obj <- pptx_obj %>%
  add_slide(layout = "Title and Content", master = "Office Theme") %>%
  ph_with(value = my_barchart, location = ph_location_type(type = "body"))

print(pptx_obj, target = "temp/my powerpoint export.pptx")
```

If you open the PowerPoint just exported, on the final slide, you'll find the barchart generated by {mschart}. By clicking the graph, you'll find a 'funnel' icon on the left side, which allows you filter attributes or products, hence making your graph interactive.

At last, {officer} also allows you adding images that are stored on your computer into a PowerPoint deck. This can be done through the external_img() function, which takes as input the location of the file. Like for any other graph,

simply apply this function within `ph_with()` by specifying then the location where the image should be printed.

6.2.3 Word

The process for building Word document directly from R is very similar to the one for PowerPoint, since it is also handled though `{officer}`.

To start a new Word document, the `read_docx()` function is being used. Since Word documents are more *text oriented* than PowerPoint, blocks of text are defined as paragraph. To introduce a new paragraph, the `body_add_par()` function is called. Note that paragraphs are automatically separated by line breaks:

```
docx_obj <- 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(docx_obj, target = "temp/my word export.docx")
```

Of course, it is not required to use the default formatting options from the word document in use. Instead, we can format it directly from R using `body_add_fpar()` to add a formatted text paragraph, or apply pre-defined styles to the previous function suggested (as is the case here with `style = "heading 1"` to set the text as a title of level 1).

```
my_format <- fp_text(font.family = 'Calibri', font.size = 14, bold = TRUE, color = 'blu
my_text <- ftext('Here is another example of text', my_format)
my_par <- fpar(my_text)

docx_obj <- read_docx() %>%
  body_add_par(value = "Document Title", style = "heading 1") %>%
  body_add_par(value = "", style = "Normal") %>%
  body_add_fpar(my_par, style = "Normal")

print(docx_obj, target = "temp/my word export.docx")
```

To export tables or figures, additional functions including `body_add_table()` (for tables) and `body_add_gg()`[6] (for `ggplot2()` figures) are used. These can be combined to `body_add_caption()` to add a caption to your table/figure:

---

[6] Note that `body_add_img()` and `body_add_plot()` can also be used.

```r
table_num <- run_autonum(seq_id = "tab", pre_label = "Table ", bkm = "tables")
figure_num <- run_autonum(seq_id = "fig", pre_label = "Figure ", bkm = "figures")

docx_obj <- docx_obj %>%
  body_add_par(value = "Exporting Tables", style = "heading 2") %>%
  body_add_par(value = "", style = "Normal") %>%
  body_add_par(value = "Here is my first table:", style = "Normal") %>%
  body_add_par(value = "", style = "Normal") %>%
  body_add_table(value = head(mtcars)[,1:4], style = "table_template") %>%
  body_add_caption(block_caption("My first table.", style="centered", autonum=table_num)) %>%
  body_add_par(value = "Exporting Figures", style = "heading 2") %>%
  body_add_par(value = "", style = "Normal") %>%
  body_add_par(value = "Here is my first figure:", style = "Normal") %>%
  body_add_par(value = "", style = "Normal") %>%
  body_add_gg(value = chart_to_plot) %>%
  body_add_caption(block_caption("My first figure.", style="centered", autonum=figure_num))

print(docx_obj, target = "temp/my word export.docx")
```

As can be seen, `body_add_caption()` is combined to `block_caption()`, and can have some automated numbering, as defined previously using `table_num` for tables, and `figure_num` for figures.

Unlike a PowerPoint file that contains separate slides, a word document is a continuous object. Hence, to emphasize a break and add content to a new page, `body_add_break()` needs to be called. Additionally, tables of content can be generated using `body_add_toc()`:

```r
docx_obj <- docx_obj %>%
  body_add_break() %>%
  body_add_par(value = "Conclusion", style = "heading 1") %>%
  body_add_break() %>%
  body_add_par("Table of Contents", style = "heading 1") %>%
  body_add_toc(level = 2)

print(docx_obj, target = "temp/my word export.docx")
```

As can be seen, it is possible to format a nice report in Word directly from R, that integrates text, tables, and figures.

For more information regarding `{officer}`, and on how to export results to Word and PowerPoint, see https://ardata-fr.github.io/officeverse/index.html.

To go further, it is worth mentioning that `{officer}` also allows extracting information from existing reports (Word and PowerPoint). It is however outside the scope of this book, so it will not be further developed here.

=======

6.2.4   Notes on applying corporate branding

You may have noticed that we have been consistent with our approach to export results to reports, regardless of the final output: We start with pre-defining our styling parameters that we then apply to our different tables, slides, paragraphs, etc. This is not a formal rule, yet we strongly recommend you adopting this way of working. Indeed, by creating your different styling parameters at the start of your script file, these lines of code do not interfere with your analyses. At a later stage, you will thank yourself for keeping well-structured code as it gains in clarity, and hence facilitates debugging your code in case of error or changes.

To go one step further, we would recommend you storing all these styling parameters in a separate file you load any time you need them using `source()`. This process **reduces** the size of your script file (increasing clarity), while **harmonizing** all your exports by **centralizing** your **formatting code** in one unique place. The last point is particularly important since any changes required only need to be done once, yet they will be applied to all your reports.

As we have seen, `{officer}` gives you the opportunity to import pre-defined templates (PowerPoint or Word). This is very valuable as your report can easily match your corporate style.

Ultimately, to ensure optimal efficiency, we advise you to spend a bit more time when constructing your report by ensuring that as many details are being taken care of, so that later on, you can spend more time in the story building part and less on the analysis and slide generation. For instance, don't be afraid of mass-exporting results, as it is easier to remove some slides, tables, or figures (in case they are not needed for building your story) then it is to re-generate them at a later stage (in case they are missing).

6.3   Integrating analyses scripts within your reporting tool

As we have just see, we can generate reports in the Microsoft Office suit directly from our R script. Although the results are being showed, the script used to reach these results is completely hidden. Of course, we could add them as text, but the logic would suggest that the researcher can just get back to the script to decode how certain outputs have been obtained.

Let's now change our way of working by proposing an alternative in which we integrate our R analysis directly within a reporting tool. For that, we need to introduce another useful package for reporting and document building: `{rmarkdown}`.

6.3.1 What is `{rmarkdown}`

Markdown is an ecosystem specific to text document, in which authors *script* their reports by controlling various features including:

- paragraphs and inline formatting (e.g. bold, italic, etc.)
- (section) headers
- blocks (code, or quotations)
- (un)numbered lists
- horizontal rules
- tables and figures (including legends)
- LaTeX math expressions, formulas, and theorems
- links, citations, and footnotes

Limiting the creation of Markdown document to this list of elements is more an advantage than a drawback as it suffice to create technical and non-technical documents while still keeping it simple.

In practice, R Markdown provides an authoring framework for data science, as it can be use for both saving/documenting/executing code and generating high quality reports.

Once the document is being created, you can then compile it to build it in the output format of your choice (e.g. word, pdf, html, etc.)

6.3.2 Starting with {rmarkdown}

To start, you need to install the `{rmarkdown}` package using the `install.packages()` function. To load this package, just type `library(rmarkdown)`.

If you intend to build your report in pdf, you also need to install a LaTeX library. For its simplicity, we recommend you installing the TinyTeX library using `install.packages("tinytex")`.

Let's start with a simple example that is provided by RStudio. To start a RMarkdown document, click File > New File > R Markdown… This opens a new window in which you can inform the name of your file, the author name, and the type of report to create (HTML, PDF, or Word). Once set, click OK. A new script file of type *.Rmd* opens.

In this document, there are three components: metadata, text, and code.

The document starts with the metadata. It is easily recognizable as it starts and ends with 3 successive dashes (`---`), and its syntax is YAML (YAML Ain't Markup Language). In this part, information regarding the properties of the final document is being stored. This includes (amongst other) the title, authorship, date, export format, etc. of the final document.

Be aware that indentation matters in YAML, so follow the rules to ensure that your document compiles correctly.

Right after the metadata is the body of document. The syntax for the text is Markdown, and the main features will be presented in the next section. Within the body, computer code can be added, either as a chunk, or within the text.

6.3.3 `{rmarkdown}` through a Simple Example

To illustrate the use of `{rmarkdown}`, let's consider this simple document (inspired from REF):

The top of the document contains the metadata, which (in our case) will generate the report in an HTML document.

Next, we have a first chunk of code that sets the main options on how the code should be handled. If the same philosophy should be applied everywhere, it is handy to set it at the start. However, when different chunks of code should be handled differently, it may be easier to define for each section how it should be handled.

There are mainly four ways to handle code.

The first way is defined here on the code chunk *header* as `include = FALSE`[7]: `include` always run the code, yet it allows printing (`include = TRUE`) or not (`include = FALSE`) the code and its outputs in the final document.

The second option is `echo`. In this code chunk, we automatically set that all the code chunk should be defined as `echo = TRUE`, which means that the code will run and be printed (together with its output) in the document. This seems very similar to `include`, yet it differs from it as `echo = FALSE` runs the code, prints the outputs, but not the code.

If you only want to show some code without running it, the `eval` parameter is used (`eval = FALSE` means that the code will be displayed but will not run). This is useful for displaying example code, or for disabling large or time-consuming chunk of codes without having to set it up as comment.

last, we can control whether outputs should be shown or hidden using `results` (printed output) and `fig.show` (plots). By default, the results are shown, unless it is set as `results = "hide"` or `fig.show = "hide"`.

The document then continues with a section header, which starts with #. The hierarchy of headers is defined by the number of adjacent # (for a header of level 3, starts the header with ###).

In this section, a first paragraph is being written. This is plain text, except for two particular words, one written between two *backticks*, and one written

---

[7]When set manually, this is where you should indicate how to handle each chunk of code

between 2 double *stars*. Here, the *backticks* are used to write text in R font (or as we will see later, to render results from R), whereas the double *stars* write the text in bold (instead of double *stars*, we can also use double *underscore*). For italic, one single *stars* (or single *underscores*) are used.

If the following section and sub-section, we introduce numbered and unnumbered list of elements. For numbered list, starts with a *number* followed by a *.* (numbers will be incremented automatically). For unnumbered list, you can either start with a *dash*, or with *star* for bullet points. For sub-list, indent your marker by pressing the *Tab* key.

In the next section called *Analysis*, we are running our first lines of code.

The first code chunk runs a regression model. In the text under the second chunk of code, we are retrieving automatically a value from R by including a *r* at the starts of two *backticks* followed by the element to retrieve. In our final report, this code will automatically be replaced by the value 3.93.

The second code chunk shows how the results can printed, either directly from R, or in a nicer way using the `knitr::kable()` function.

Finally, the last code chunk of this section creates a plot with a caption, that is automatically numbered.

6.3.4 Creating a document using `{knitr}`

Once the document is ready, you can neat it using the `knit` button. This will create the report in the format of interest (here HTML).

6.3.5 Example of applications

`{rmarkdown}` is a very powerful tool for building report, in particular in the context of reproducible research since it allows sharing code, and running analyses within the report (part of the text around the code can justify the decisions made in terms of analyses to ensure transparency). The latter point is particularly interesting since any change in the data will automatically provide updated results throughout the report, without you having to change them manually.

Its application is various, and can go from report, to teaching material, publication or even books (this book has been written in `{rmarkdown}` and its extension `{bookdown}`), emails, websites, dashboards, surveys etc. Even more interestingly, `{rmarkdown}` can also be combined to `{shiny}` to build interactive reports, dashboards, or teaching materials in which users would (say) import their data set, select the variables to analyze through buttons, chose which analyses and which options to perform, and the results will automatically be generated accordingly.

For more information on `{rmarkdown}` and related packages, please visit (REF)

As mentioned earlier, R Markdown can also be used to generate other types of documents, including presentations. This can be done directly from the `{rmarkdown}` package using *ioslides* presentation (`output: ioslides_presentation` in the metadata), *Slidy* presentation (`output: slidy_presentation`), or *PowerPoint* presentation (`output: powerpoint_presentation` with `reference_doc: my-styles.pptx` to apply your own template) just to name these. It can also be done using additional packages such as the `{xarigan}` package. However, in this section, we deliberately chose to use `{officer}` because of its simplicity and flexibility.

6.4   To go further...

If R allows you saving time by creating your report within your R-script, or by running your analysis within your report document, it cannot communicate the presentation to your partners/clients for you. However, if the report is very standard (say only key results, tables or figures), or running routinely (say in quality control), R could automatically run the analysis as soon as new data is available, build the report, and send it automatically to you, your manager or your colleagues and partners by email.

Such process can be done thanks to the `{blastula}` package (see REF).

Bon Appétit

Chapter 7

Example Project: The Biscuit Study

The dataset that we use as a main example throughout this book comes from a sensory study on biscuits. The study was part of project BISENS funded by the French National Research Agency (ANR, programme ALIA 2008). These biscuits were developed for breakfast consumption and specifically designed to improve satiety. The study was conducted in France with one hundred and seven consumers who tested a total of 10 biscuit recipes (including 9 experimental products varying in their fiber and protein content). Fibers and proteins are known to increase satiety.

The study aimed to measure the liking for these biscuits, its link with eaten quantities and the evolution of hunger sensations over ad libitum consumption. All the volunteers therefore participated to ten morning sessions in order to test every product (one biscuit type per session). After they completed all the sessions, they also filled a questionnaire about food-related personality traits such as cognitive restraint and sensitivity to hunger.

Parallel to this, a panel of nine trained judges performed a descriptive analysis of the biscuits. They evaluated the same 10 products as well as an additional product whose recipe was optimized for liking and satiating properties.

Data from the biscuit study are gathered in three Excel files that can be accessed here [ADD LINK HERE]:

- biscuits_consumer_test.xls

- biscuits_sensory_profile.xls

- biscuits_traits.xls

Table 7.1: Product set for the biscuit study

Product	Protein	Fiber	Type
P01	Low	Low	Trial
P02	Low	High	Trial
P03	High	High	Trial
P04	High	High	Trial
P05	High	Low	Trial
P06	High	Low	Trial
P07	Low	High	Trial
P08	High	Low	Trial
P09	High	High	Trial
P10	Low	Low	Commercial product
POpt	High	Low	Optimized trial

7.1   Products

In total, 11 products were considered in this study. They are all breakfast biscuits with varying contents of proteins and fibers (Table 7.1). Products P01 to P09 are prototypes whereas product P10 is a standard commercial biscuit without enrichment. The eleventh product Popt is an additional optimized biscuit that has been evaluated only by the trained panel for descriptive analysis.

In total, 11 products were considered in this study. They are all breakfast biscuits with varying contents of proteins and fibers. Products P01 to P09 are prototypes whereas product P10 is a standard commercial biscuit without enrichment. The eleventh product Popt is an additional optimized biscuit that has been evaluated only by the trained panel for descriptive analysis.

7.2   Consumer test

7.2.1   Participants

107 women of normal weight (Body Mass Index (BMI): 18-26) participated to the test. All were regular consumers of breakfast biscuits. The biscuits_traits.xls file gives some sociodemographic information (e.g. marital

status, household, income, occupation).

7.2.2 Test design

Consumers evaluated one biscuit type per session. Product evaluation order was randomized across the panel.

The design of the sessions is summarized in Figure 7.1 with main measured variables. After they first rated their appetite sensations using visual analog scales (VAS), the participants tasted and rated one biscuit for liking. They were then served with a box of the same biscuits for *ad libitum* consumption, followed by a new questionnaire regarding their liking, pleasure and appetite sensations.

The liking was measured with two different scales:
1. with a horizontally oriented unstructured linear scale (VAS) anchored with 'I don't like this biscuit at all' (left end) and 'I like this biscuit a lot' (right end) at two times (at the first bite and when stopping their consumption).
2. with a vertically oriented semantic nine-point hedonic scale when stopping their consumption.
VAS scales are frequently used in nutrition studies (REF REF), whereas the nine-point hedonic scale is more popular in sensory and consumer science (REF REF).

Once done, participants were asked about the reason(s) why they stopped eating (6 potential reasons rated with *Likert* scales ranging from *strongly disagree* to to *strongly agree*). They were also asked how much they would like to eat other types of foods (11 food items rated using a VAS).

The time spent in the booth and the number of biscuits eaten by each participant was recorded by the experimenters, as well as the type of drink they selected and the approximate volume they drank during each session. These data are stored in the **Consumer Test.xlsx** file, in the second tab named *Time Consumption*.

7.3 Sensory descriptive analysis data

A panel of 9 trained judges evaluated the 11 products for 32 sensory attributes (8 attributes for aspect, 3 for odor, 12 for flavor, 9 for texture).

For each product, the judges individually reported the perceived intensity of each attribute on an unstructured linear scale. Intensities were automatically converted by the acquisition system into a score ranging from 0 to 60.

7.4 Summary of the datasets

List of the variables in each data sets (or just the size of each data set?).

Appetite (VAS)
  • Hunger
  • Fullness
  • Desire to eat
  • Prospective quantity

  • *Quantity consu*
  • *Duration of co*

t_0

First bite (VAS)
  • Initial overall liking
  • Desire to eat this product

Figure 7.1: General design for the consumer test of the biscuit study. Participants were served with a different set of biscuits every session.

Chapter 8

Data Collection

8.1 Design

8.1.1 Designs of sensory experiments

8.1.1.1 General approach

Sensory and consumer science relies on experiments during which subjects usually evaluate several samples one after the other. This type of procedure is called 'monadic sequential' and is common practice for all three main categories of tests (difference testing, descriptive analysis, hedonic testing). The main advantage of proceeding this way is that responses can be analyzed at the individual level so that analysis and interpretation can account for inter-individual differences, which is a constant feature of sensory data.

However, this type of approach also comes with drawbacks[1] as it may imply order effects and carry-over effects. Fortunately, most of these effects can be controlled with a proper design of experiment (DoE). A good design ensures that order and carry-over effects are not confounded with what you are actually interested to measure (most frequently, the differences between products) by balancing these effects across the panel. However, it is important to note that the design does not eliminate these effects and that each subject in your panel may still experience an order and a carry-over effect, as well as boredom, sensory fatigue, etc.

---

[1]Market researchers would argue that evaluating several products in a row doesn't usually happen in real life and that proceeding this way may induce response biases. They thus advocate the use of pure monadic designs in which participants are only given one sample to evaluate. This corresponds to a between-group design that is also frequently used in fields where only one treatment per subject is possible (drug testing, nutrition studies, etc.).

8.1.1.2   Crossover designs

For any sensory experiment that implies the evaluation of more than one sample, first-order and/or carry-over effects should be expected. That is to say, the evaluation of a sample may affect the evaluation of the next sample even though sensory scientists try to lower such effects by asking panelists to pause between samples and use of appropriate mouth-cleansing techniques (drinking water, eating unsalted crackers, or a piece of apple, etc.). The use of crossover designs is thus highly recommended (Macfie et al. (1989)).

Williams's Latin-Square designs offer a perfect solution to balance carry-over effects. They are very simple to create using the `williams()` function from the `{crossdes}` package. For instance, if you have five samples to test, `williams(5)` would create a 10x5 matrix containing the position at which each of three samples should be evaluated by 10 judges (the required number of judges per design block).

Alternately, the `WilliamsDesign()` function in `{SensoMineR}` allows you to create a matrix of samples (as numbers) with numbered `Judges` as row names and numbered `Ranks` as column names. You only have to specify the number of samples to be evaluated, as in the example below for 5 samples.

```
library(SensoMineR)
wdes_5P10J <- WilliamsDesign(5)
```

Suppose you want to include 20 judges in the experiment, you would then need to duplicate the initial design.

```
wdes_5P20J <- do.call(rbind, replicate(2, wdes_5P10J, simplify=FALSE))
rownames(wdes_5P20J) <- paste("judge", 1:20, sep="")
```

The downside of Williams's Latin square designs is that the number of samples (k) to be evaluated dictates the number of judges. For an even number of samples you must have a multiple of k judges, and a multiple of *2k* judges for an odd number of samples.

As the total number of judges in your study may not always be exactly known in advance (e.g. participants not showing up to your test, extra participants recruited at the last minute), it can be useful to add some flexibility to the design. Of course, additional rows would depart from the perfectly balanced design, but it is possible to optimize them using Federov's algorithm thanks to the `optFederov()` function of the `{AlgDesign}` package, by specifying `augment = TRUE`. For example we can add three more judges to the Williams Latin square design that we just built for *nbP*=5 products and 10 judges, hence leading to a total number of *nbP*=13 judges. Note that this experiment is designed so that each judge will evaluate all the products, therefore the number of samples per judge (*nbR*) equals the number of products (*nbP*).

```
library(SensoMineR)
library(AlgDesign)
nbJ=13
nbP=5
nbR=nbP

wdes_5P10J <- WilliamsDesign(nbP)
tab <- cbind(prod=as.vector(t(wdes_5P10J)), judge=rep(1:nbJ,each=nbR), rank=rep(1:nbR,nbJ))

optdes_5P13J <- optFederov(~prod+judge+rank, data=tab, augment=TRUE, nTrials=nbJ*nbP, rows=1:(nbJ
xtabs(optdes_5P13J$design)
```

In the code above, `xtabs()` is used to arrange the design in a table format that is convenient for the experimenter.

Note that it would also be possible to start from an optimal design and expand it to add one judge at a time. The code below first builds a design for 5 products and 13 judges and then adds one judge to make the design optimal for 5 products and 14 judges.

```
library(AlgDesign)

nbJ=13
nbP=5
nbR=nbP

optdes_5P13J <- optimaldesign(nbP, nbP, nbR)$design
tab <- cbind(prod=as.vector(t(optdes_5P13J)),judge=rep(1:nbJ,each=nbR),rank=rep(1:nbR,nbJ))
add <- cbind(prod=rep(1:nbP,nbR),judge=rep(nbJ+1,nbP*nbR),rank=rep(1:nbR,each=nbP))

optdes_5P14J <- optFederov(~prod+judge+rank,data=rbind(tab,add), augment=TRUE, nTrials=(nbJ+1)*nb
                           rows=1:(nbJ*nbP), nRepeats = 100)
```

8.1.1.3 Balanced incomplete block designs (BIBD)

Sensory and consumer scientists may sometimes consider using incomplete designs, i.e. experiments in which each judge evaluates only a subset of the complete product set (Wakeling and MacFie (1995)). In this case, the number of samples evaluated by each judge remains constant but is lower than the number of products included in the study.

You might want to choose this approach for example if you want to reduce the workload for each panelist and limit sensory fatigue, boredom and inattention. It might also be useful when you cannot "afford" a complete design because of sample-related constraints (limited production capacity, very expensive samples,

etc.). The challenge then, is to balance sample evaluation across the panel as well as the context (i.e. other samples) in which each sample is being evaluated. For such a design you thus want each pair of products to be evaluated together the same number of times.

The `optimaldesign()` function of `{SensoMineR}` can be used to search for a Balanced Incomplete Block Design (BIBD).

```
incompDesign1 <- SensoMineR::optimaldesign(nbPanelist = 30, nbProd= 10, nbProdByPaneli
incompDesign1$design
```

BIBD are only possible for certain combinations of numbers of treatment (products), numbers of blocks (judges), and block size (number of samples per judge). Note that `optimaldesign()` will yield a design even if it is not balanced but it will also generate contingency tables allowing you to evaluate the design's orthogonality, and how well balanced are ranks and carry-over effects.

You can also use the `{crossdes}` package to generate a BIBD with this simple syntax: `find.BIB(trt, b, k, iter)`, with `trt` the number of products, `b` the number of judges, `k` the number of samples per judge, and `iter` the number of iteration. Furthermore, the `isGYD()` functions evaluates whether the incomplete design generated is balanced or not. If the design is a BIBD, you may then use `williams.BIB()` to combine it with a Williams design to balance carry-over effects.

Incomplete balanced designs also have drawbacks. First, from a purely statistical perspective, they are conducive to fewer observations and thus to a lower statistical power. Product and Judge effects are also partially confounded even though the confusion is usually considered as acceptable.

8.1.1.4 Incomplete designs for hedonic tests: Sensory informed designs

One may also be tempted to use incomplete balanced block designs for hedonic tests. However, proceeding this way is likely to induce framing bias. Indeed, each participant will only see part of the product set which would affect their frame of reference if the subset of product they evaluate only covers a limited area of the sensory space.

Suppose you are designing a consumer test of chocolate chip cookies in which a majority of cookies are made with milk chocolate while a few cookies are made with dark chocolate chips. If a participant only evaluates samples that have milk chocolate chips, this participant will not know about the existence of dark chocolate and will potentially give very different scores compared to what they would have if they had a full view of the product category.

To reduce the risks incurred by the use of BIBD, an alternative strategy is to use a sensory informed design. Its principle is to allocate each panelist a

subset of products that best cover the sensory diversity of the initial product set. Pragmatically, this amounts to maximizing the sensory distance between drawn products (Franczak et al. (2015)). Of course, this supposes that one has sensory data to rely on in the first place.

Initialization:
   1. Assign each product a number from 1 to p, where p is the total number of products
   2. Compute the Euclidean distance between each target product using their sensory profile.
   Note: Herein we will refer to this matrix as `dist`.
Construction:
   Create a vector of length p where element is equal to r
   Note: Herein we will refer to this vector as `ct.prod`
      for l in 1:k
         for i in 1:n
            if $l = 1$ Randomly assign panelist i one product such that position l is position balanced.
               if $l \geqslant 2$ {
                  Step 1. Randomly select a panelist, with priority given to a panelist whose first position contains a product that is very similar to the other products.
                  Step 2. Use `dist` to determine which product, j, is the most dissimilar to the product in position 1, or to the centroid of positions 1:$l - 1$ if $l > 2$, of the panelist selected in Step 1.
                  Step 3. Assign product, j, found in Step 2 to position l of the panelist selected in Step 1.
                  }
               Subtract 1 from element j of `ct.prod` where j corresponds to the chosen product.
               if any element of `ct.prod` equals 0, no longer assign that product.
         end for i in 1 : n
      end for l in 1 : k

8.1.2 Product-related designs

Because of their contribution to product development, sensory and consumer scientists often deal with DoE other than sensory designs strictly speaking (see for instance REF REF book Gacula, 2008). Sensory-driven product development is indeed very frequent and implies strong interconnection between the measure of sensory responses and the manipulation of product variables (e.g. ingredients) or process variables (e.g. cooking parameters) (for a review, see REF REF Yu et al. 2018).

In order to get the most of sensory experiments, it is thus essential to ensure that the products or prototypes to be tested will be conducive to sound and conclusive data. First and foremost, as in any experimental science, one wants to avoid confounding effects. In addition to this and to put it more generally, the goal of DoE is to define which trials to run in order to be able to draw reliable conclusions without spending time and resources on unnecessary trials. In other words, one seeks maximum efficiency. This is especially critical in sensory science to limit the number of products to be evaluated and to keep panelists' workload under control.

8.1.2.1 Factorial designs

Full factorial designs are of course commonly used and their application is usually straightforward. They won't be detailed here. However, always in the view of sparing experimental resources, incomplete designs are frequent and require careful preparation. Several strategies can be used to define which experiments to conduct (REF REF book Dean et al. 2018). Ref to Lawson 2015 (Design and analysis of experiments with R).

8.1.2.2 Mixture designs

In many projects (e.g. in the food industry, in the personal care industry), optimizing a product's formula implies adjusting the proportions of its ingredients. In such cases, the proportions are interdependent (the total sum of all components of a mixture must be 100%). Therefore, these factors (the proportions) must be treated as mixture components. Mixture designs are usually represented using ternary diagrams.

The `{mixexp}` package offers a very convenient way to do this. In addition to creating the design, `DesignPoints()` allows to display the corresponding ternary diagram. Below is the example of a simplex-lattice design for 3 components and 3 levels obtained thanks to function `SLD`:

```
library(mixexp)
mdes<-SLD(3,3)
DesignPoints(mdes)
```

Suppose that we want to adjust a biscuit recipe to optimize its sensory proper-
ties, we can design an experiment in which the proportion of ingredients vary.
Let's play with butter, sugar, and flour. All three combined would account for
75% of the dough recipe and the remaining 25% would consist of other ingre-
dients that we won't modify here (eggs, milk, chocolate, etc.). Besides, not
any amount of these three ingredients would make sense (a biscuit with 75% of
butter is not a biscuit, even in Brittany). We thus need to add constraints (ex:
butter varies between 15 and 30% of this blend, sugar varies between 25 and
40%, and flour varies between 30 and 50%). Given this set of constraints, we
can use `mixexp::Xvert` to find the extreme vertices of our design.

However, this design would imply 11 mixtures, which is more than needed to
apply a Scheffé quadratic model (REF REF REF). To reduce the number of
mixtures and still allow fitting a quadratic model, we can use the `optFederov()`
function from `{AlgDesign}` to select a D-optimal subset. Here, let's limit to 9
products.

```
library(mixexp)
library(AlgDesign)
library(tidyverse)

mdes2 <- Xvert(nfac=3, uc=c(.30, .40, .50), lc=c(.15, .25, .30), ndm = 1, plot = FALSE
        mutate_if(is.numeric, round, digits = 3)

MixBiscuits <- optFederov(~ -1 +x1 +x2+ x2 +x1:x2 +x1:x3 +x2:x3 +x1:x2:x3, mdes2, nTria
DesignPoints(MixBiscuits$design, axislabs = c("Butter","Sugar","Flour"), pseudo = TRUE
```

```
## Warning: package 'mixexp' was built under R version 4.0.5

## Warning in system(cmd, intern = intern, wait = wait | intern,
## show.output.on.console = wait, : running command 'C:\WINDOWS\system32\cmd.exe /c
## ftype perl' had status 2

## Warning in system(cmd, intern = intern, wait = wait | intern,
## show.output.on.console = wait, : running command 'C:\WINDOWS\system32\cmd.exe /c
## ftype perl' had status 2

## Warning: package 'daewr' was built under R version 4.0.5

## Warning: package 'ggplot2' was built under R version 4.0.5
```

Once the data are collected we can use the `mixexp::MixModel()` function to fit
a linear model and `mixexp::MixturePlot()` to draw a contour plot. Suppose

Table 8.1: Average liking scores obtained for the biscuits from the mixture design

Product	Butter	Sugar	Flour	Liking
A	0.30	0.25	0.45	7.5
B	0.15	0.40	0.45	5.4
C	0.30	0.40	0.30	5.5
D	0.25	0.25	0.50	7.0
E	0.15	0.35	0.50	6.0
F	0.30	0.32	0.38	8.0
G	0.22	0.40	0.38	5.8
H	0.20	0.30	0.50	6.8
I	0.23	0.33	0.44	7.9

that we obtain average liking scores for our 9 biscuits as given in Table 8.1, this simple code would allow to get a contour plot that shows where would be the optimal area for the biscuit formulation.

```
library(mixexp)
# calculate the cubic mixture model
invisible(capture.output(res <- MixModel(Bmixt, "Liking",
                              mixcomps = c("Butter", "Sugar", "Flour"), model = 4)) )
# plot the model as ternary contour plot
ModelPlot(model = res,
        dimensions = list(x1="Butter", x2="Sugar", x3="Flour"),
        lims = c(0.15, 0.30, 0.25, 0.40, 0.30, 0.50), constraints = TRUE,
        contour = TRUE, cuts = 12, fill = TRUE, pseudo = TRUE,
        axislabs = c("Butter", "Sugar", "Flour"))
```
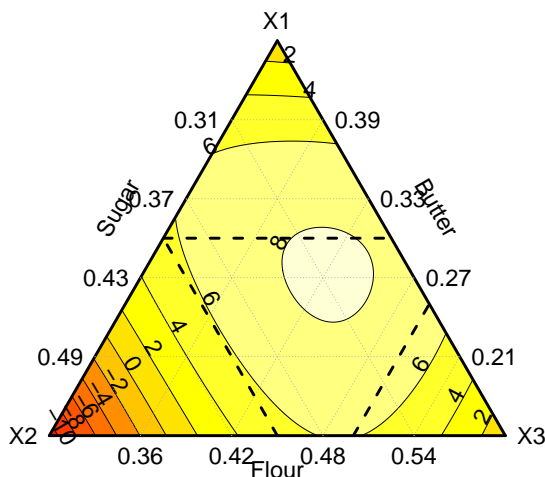
Regardless of the construction of the mixture design, ternary diagrams are easy to plot with packages such as `{ggtern}` or `{Ternary}`. `{ggtern}` is particularly interesting because it builds on `{ggplot2}` and uses the same syntax.

8.1.2.3   Screening designs

Product development is not a monolithic process and in the early stages of a project it could be extremely useful to use a screening design in combination to sensory evaluation to identify most influential factors of interest (REF REF REF). Factorial and mixture designs belong to the product developers' essential toolkit and could serve this purpose. In practice however, they can only include a relatively limited number of factors. By contrast, screening designs are extremely efficient at dealing with many factors, pending some sacrifices on the evaluation of interactions and quadratic effects. Although screening designs are only scarcely used, studies have shown that they could greatly contribute to sensory-led product development, including in nonfood applications of sensory science (REF REF REF).

Plackett-Burman designs are the most commonly used screening designs. They can be easily obtained with `{FrF2}`.

8.1.2.4   Sensory informed designs

Eventually, it is worth mentioning that in some cases sensory properties themselves can be used as factors and implemented in a DoE. Naturally, this implies

that product developers have (1) access to the measure of these properties and (2) can control the level of these properties and their interactions. These requirements are rarely met in food development but can be more easily implemented in some non-food applications. A specific case, is the use of sensory information to make a selection of a subset of products, as described above.

(Include ref to Naes et al REF REF)

8.2 Execute

Sir Ronald Fisher famously said in his presidential address to the first Indian statistical congress (1938): *"To consult the statistician after an experiment is finished is often merely to ask him to conduct a post mortem examination. He can perhaps say what the experiment died of."*

Hopefully, the sections above would have helped the sensory and consumer scientist designing their experiment in a way that would warrant them relevant and meaningful data that are obtained with maximum efficiency.

Fisher continues: *"To utilise this kind of experience the statistician must be induced to use his imagination, and to foresee in advance the difficulties and uncertainties with which, if they are not foreseen, his investigations will be beset."* Fortunately, we can spare the reader some of these imagination efforts and reiterate the fundamental principles of sensory evaluation that should help avoiding major pitfalls[2].

- **Individual evaluation**
  Probably the most important requirement for measurement validity is to perform individual evaluation. Sensory responses are very easily biased when judges can communicate. When this happens, observations cannot be considered independent which would rule out most statistical tests. Although this principle is generally accepted and correctly applied, some situations (such as project team meetings, b2b sample demonstration, tasting events, etc.) may be more challenging in this regard. Individual evaluation is usually ensured by the use of partitioned sensory booths, but it can also be achieved by other means (table-top partitions, curtains, separate tables, separate rooms). There are some cases, in consumer research, where interactions between subjects are allowed or even encouraged because they correspond to real-life situations. But these are exceptions to the rule, and in such cases, observations are to be considered at the group level.

---

[2]For a more detailed description of these principles, we refer the reader to comprehensive sensory evaluation textbooks. See for instance (Lawless and Heymann, 2010), (Civille and Carr, 2016) or (Stone, Bleibaum and Thomas, 2022)

- **Balanced order effects and treatments**
  We already discussed the importance of balancing the evaluation order for first-order and carry-over effects (#LinkAppropriateSection). We cannot overstate how necessary this precaution is to get valid data. On top of having to deal with such effects, sensory scientists sometimes want to test how products are perceived (or liked) under different conditions (e.g. blind vs. branded, with/without nutritional information, in the lab vs. at home, etc.). Choice must then be made between a within-group design (in which participants evaluate the products under the different conditions) and a between-group design (in which participants evaluate the product under one condition only). As often in consumer science, there is no perfect experiment and these two options have pros and cons. For instance, the within-group design would be more powerful and would allow data analysis at the individual level, but it would be more likely to induce response biases. Note that in both cases, participants must be randomly assigned to one group (corresponding either to a given condition, or to the order in which each condition is being experienced, for within-group designs).

- **Separate affective from analytical tasks**
  In sensory evaluation, a clear distinction is usually made between analytical measurements (whereby emphasis is placed on description of sample characteristics or on differences and similarities between samples) and affective measurements (whereby focus is placed on liking, preferences, and emotions that may derive from the consumption of a product). Because the tasks involved in these two types of measurements are very different, the general recommendation is to conduct them separately (and most often, with different people). Proceeding otherwise would risk inducing cognitive biases and collecting skewed - or even meaningless - data. For example, if the goal of a study it to measure how much consumers like a given set of food products, it wouldn't make sense to ask trained panelists rate their liking for the products they have been trained to describe. They can certainly do it, but their judgement of the products is likely to be changed by that training and their extensive exposure to the product. Therefore, they can no longer be considered *normal* consumers. This is relatively commonsense, however the risk of biases can sometimes be more subtle. Indeed, it might be tempting to ask consumers to give their liking for samples and, within the same session, to describe the same samples for a number of attributes. By doing so, you risk changing participants mindset (e.g. by over-focusing on specific attributes) and thus alter liking scores (REF REF) although there is much debate on which type of descriptive tasks would actually lead to biased responses. With this in mind, experimenters might still consider conducting combined measurements for product optimization, especially to get rough estimates of product specifications to target in the first stage of product development. In this objective, Just-about-right (JAR) scales are very popular. They provide a very direct way to optimize products' sensory characteristics. Alternately, one

might expect that 'untrained' consumers cannot be used for descriptive analysis. However in the past few decades, the development of descriptive methods that do not require training and that can be achieved in a single session has made consumer-based descriptive analysis possible and reliable.

- **Controlled evaluation conditions an blind evaluation**

Final word: quantification, scaling, sensory acquisition softwares

```
Bmixt_shuffled <- Bmixt[sample(nrow(Bmixt)), ]
```

8.3 Import

It is a truism, but to analyze data we first 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(x)* file), or software specific (e.g. Stata, SPSS, etc.).

Since it is very common to store the data in Excel spreadsheets (*.xls(x)*) 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, you may find alternative packages that would allow importing your files in 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. Handles a large variety of files,
2. Guess the type of file it is,
3. Provides tools to import, export, and convert almost any type of data format, including *.csv*, *.xls(x)*, 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 `readxl::read_xlsx()` by informing as parameter the `location` of the file and the `sheet` where it is stored. For convenience, we are using the `{here}`[3] package to retrieve the path of the file (stored in `file_path`).

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 that is not data):

- The first row within the *Data* sheet of *Sensory Profile.xlsx* contains the headers,

- From the second row onward, 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 data thus imported is stored as 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 column, defined as character), and the sensory attributes (third column onward, defined as numerical or `dbl`).

8.3.2   Importing Unstructured Excel File

In some cases, the data are not so well organized/structured, and may need to be *decoded*. This is the case for the workbook entitled *TFEQ.xlsx*.

In this file:

---

[3]The package `{here}` is very handy as it provides an easy way to retrieve your file's path (within your working directory) by simply giving the name of the file and folder in which they are stored.

- 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 data set, multiple steps are required:

- Import the variables' name only;
- Import the information regarding the levels;
- Import the data 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")))
```

Ultimately, the data (*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 data 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!).

Alternatively, you can import the data by specifying the range in which the data is being stored (here 'range="A2:BJ108"').

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.

Remark: Other *unstructured* data include the information regarding the levels of a factor as sub-header. In such case, a similar approach is used:

- Start with importing the first rows of the data that contain this information using the parameter `n_max` from 'readxl::read_xlsx'.
- From this subset, extract the column names.

  - For each variable (when information is available), store the additional information as a list of tables that contains the code and their corresponding label.
  - Re-import the data by skipping these rows, and by applying manually the headers to use.

8.3.3   Importing Data Stored in Multiple Sheets

It can happen that the data that needs to be analyzed is stored in different files, or in different sheets within the same file. Such situation could happen if the same test involving the same samples is repeated over time, or has been run simultaneously in different locations, or simply for convenience, your colleague wanted to simplify your task and already split the data based on a variable of interest.

Since the goal here is to highlight the possibilities in R to handle such situations, we propose to use a small fake example where 12 panelists evaluated 2 samples on 3 attributes in 3 sessions, each session being stored in a different sheet in *excel_scrap.xlsx*.

A first approach to tackle this problem could be to import each file separately, and to combine them together using the `bind_rows()` function from the `{dplyr}` package. However, this solution is not optimal since it is very tedious when a larger number of sheets is involved, and it is not automated since the code will no longer run if the number of session changes.

To counterbalance, we first introduce the function `excel_sheets()` from `{readxl}` as it provides all the sheet that are available in the file of interest. This allows us reading all the sheets from that file, regardless of the number of sessions. Second, the function `map()` from the `{purrr}` package comes handy as it applies a function (here `read_xlsx()`) to each element of a list or vector (here, the one obtained from `excel_sheets()`).

```
path <- file.path("data", "excel_scrap.xlsx")
files <- path %>%
  excel_sheets() %>%
  set_names(.) %>%
  map(~read_xlsx(path, sheet = .))
```

As can be seen, this procedure creates a list of tables, with as many elements are there are sheets in the excel file. To convert this list of data tables into one unique data frame, we first extend the previous code and `enframe()` it by informing that the separation was based on `Session`. Once done, the data (stored in `data`) is still nested in a list, and should be *unfolded*. Such operation is done with the `unnest()` function from `{tidyr}`:

```
files %>%
  enframe(name = "Session", value = "data") %>%
  unnest(cols = c(data))
```

This procedure finally returns a tibble with 72 rows and 6 columns, ready to be analyzed!

To go further:

1. Instead of `enframe()`, we could have used `reduce()` from `{purrr}`, or `map()` combined with `bind_rows()`, but both these solutions would then lose the information regarding the `Session` since it is not part of the data set itself.
2. The functions `enframe()` and `unnest()` have their alter-ego in `deframe()` and `nest()` which aim in transforming a data frame into a list of tables, and in nesting data by creating a list-column of data frames.
3. In case the different sets of data are stored in different excel files (rather than different sheets within a file), we could apply a similar procedure by using `list.files()` from the `{base}` package, together with `pattern = "xlsx"` to limit the search to Excel files present in a pre-defined folder.

Chapter 9

Data Preparation

```
library(tidyverse)
```

Data Preparation, which consists of *data inspection* and *data cleaning*, is a critical step before any further *Data Manipulation* or *Data Analysis*. Having a good data preparation procedure ensures a good understanding of the data, and avoids what could be very critical mistakes.

To illustrate the importance of the later point, let's imagine a study in which the samples are defined by their 3-digits code. During importation, R would recognize them as number, and hence defines the *Product* column as numerical. Without inspection and correction, any ANOVA that include the product effect would be replaced by a linear regression (or analysis of covariance) which of course does not provide the results required (although the analysis would run without error). Worst, if this procedure is automated and the *p-value* associated to the product effect is extracted, the conclusions would rely on the wrong analysis! A good data preparation procedure is hence important to avoid such unexpected results.

So what consists of data preparation, and how does that differ from data manipulation? There is clearly a very line between data preparation (and particularly *data cleaning*) and data manipulation, many procedures being used in both (same applies to data manipulation and data analysis for instance). For this book, we decided to follow this rule:

Data Preparation includes all the required steps to ensure that the data is matching its intrinsic nature. These steps include inspecting the data at hand (usually through simple descriptive statistics of the data as a whole, with no *interpretation mindset*) and cleaning the data by eventually correcting importation errors (imputation of missing data is also included here). Although some descriptive

statistics are being produced for data inspection, these analyses have no interpretation value besides ensuring that the data are in the right range, or following the right distribution. For instance, with our sensory data, we would ensure that all our sensory scores are included between 0 and 100 (negative scores would not be permitted), but we would not look at the mean or the distribution of the score per product which would belong to data analyses and would require interpretation (e.g. P01 is sweeter than P02).

The *Data Manipulation* is an optional step that adjust or convert the data into a structure that is usable for further analysis. This of course may lead to *interpretation* of the results as it may involve some analyses.

The *Data Analysis* step ultimately converts the data into results (through values, graphics, tables, etc.) that provide more insights (through interpretation) about the data at hand.

The data used in this chapter includes the *Sensory Profile.xlsx* that you imported in Section 8. Another version of the sensory profiles that includes missing values is also being used (*Sensory Profile (NA).xlsx*).

9.1   Inspect

9.1.1   Data Inspection

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

```
sensory
```

Other informative solutions consists in printing a summary of the data through the `summary()` or `glimpse()` function:

```
summary(sensory)
```

```
glimpse(sensory)
```

These functions provide relevant yet basic views of each variable present in the data including the type of variable, the range, mean, and median, as well as the first values of each variables.

Such view might be sufficient for some first conclusions (e.g. Are my panelists considered as numerical or nominal data? Do I have missing values?), yet it is not sufficient to fully ensure that the data is ready for analysis. For the latter, more extensive analyses can be performed automatically in different ways. These

analyses include looking at the distribution of some variables, or the frequencies of character levels.

A first solution comes from the `{skimr}` package and its `skim()` function. By applying it to data, an automated extended summary is directly printed on screen by separating `character` type variables from `numeric` type variables:

```
library(skimr)
skim(sensory)
```

Another approach consists in generating automatically an html report with some pre-defined analyses using `create_report()` from the `{DataExplorer}` package.

```
library(DataExplorer)
create_report(sensory)
```

Unless specified otherwise through `output_file`, `output_dir`, and `output_format`, the report will be saved as an html file on your active directory as *report.html*. This report provides many statistics on your data, including some simple statistics (e.g. raw counts, percentages), informs you on the structure of your data, as well as on eventual missing data. It also generates graphics to describe your variables (e.g. univariate distribution, correlation and PCA).

Note that the analyses performed to build this report can be called directly within R: `introduce()` and `plot_intro()` generates the first part of the report, whereas `plot_missing()` and `profile_missing()` provide information regarding missing data for instance.

9.1.2 Missing Data

In the previous section on 9.1.1, it can be seen that the data set contain missing values. It concerns for instance the attribute `Light`, for which one missing value has been detected. So how to handle missing values?

9.1.2.1 Ignore missing values

A first solution is to simply *ignore* the presence of missing values, as many analyses handle them well. For instance, an ANOVA could be run for such attributes, and results are being produced:

```
broom::tidy(aov(Light ~ Product + Judge, data=sensory))
```

This solution may work fine when the number of missing values is small, but be aware that it can also provide erroneous results in case they are not handled the way the analyst is expecting them to be handled.

For some other analyses, *ignoring* the presence of missing values may simply provide results. To illustrate this, let's compute the simple mean per product for `Light`

```
sensory %>%
  group_by(Product) %>%
  summarise(Light = mean(Light)) %>%
  ungroup()
```

As can be seen, since `P04` contains missing values, its corresponding mean is defined as `NA`. To enforce the mean to be computed, we need to inform R to remove missing values when computing the mean. This is done by adding within the `mean()` function the parameter `na.rm=TRUE`:

```
sensory %>%
  group_by(Product) %>%
  summarise(Light = mean(Light, na.rm=TRUE)) %>%
  ungroup()
```

Using `na.rm=TRUE` is equivalent to removing rows containing missing values from the data before performing the analysis.

9.1.2.2   Remove missing values

The solution using `na.rm=TRUE` is a first way to handle missing values by removing or ignoring them. However, it is not the only strategy, and we can consider other approaches that are more *sensory and consumer studies* driven.

For instance, missing values can be removed systematically by deleting the entire row of data. Let's take the example of `Sour` which contains 10 missing values:

```
sensory %>%
  filter(!is.na(Sour))
```

Unfortunately, this solution is not satisfactory as it also deletes real data since the dataset went from 99 to 89 rows. This means that for variables that did not have missing values for instance, existing data have been removed.

Another approach suggests to first rotate using `pivot_longer()` the data before removing missing values:

```
sensory %>%
  pivot_longer(Shiny:Melting, names_to="Variables", values_to="Scores") %>%
  filter(!is.na(Scores)) %>%
  group_by(Product, Variables) %>%
  summarize(Means = mean(Scores)) %>%
  ungroup() %>%
  pivot_wider(names_from = Variables, values_from = Means) %>%
  dplyr::select(Product, Sour, Light)
```

If this solution seems satisfactory as the means were computed without using `na.rm=TRUE` for both `Sour` and `Light` (who contained missing values), its use is limited since converting the data to its original format (i.e. performing `pivot_wider()` after `pivot_longer()` without computing the mean in between) will reintroduce the missing values[1].

Also removing missing values has the impact of unbalancing the data. By taking the example of `Light` and `Sour`, let's print the number of panelist evaluating each product:

```
sensory %>%
  pivot_longer(Shiny:Melting, names_to="Variables", values_to="Scores") %>%
  filter(!is.na(Scores),
         Variables %in% c("Light","Sour")) %>%
  group_by(Product, Variables) %>%
  count() %>%
  ungroup() %>%
  pivot_wider(names_from=Variables, values_from=n)
```

Here for example, the only missing value detected for `Light` is related to `P04`. For `Sour`, `P02`, `P07`, and `P09` only have 7 observations out of 9.

Another strategy consists in removing attributes, products, or panelists that have missing data. The procedure presented below show the procedure on how to remove attributes with missing data, but could easily be adapted to panelists or products:

```
sensory_long <- sensory %>%
  pivot_longer(Shiny:Melting, names_to="Variables", values_to="Scores")

(attr_rmv <- sensory_long %>%
  filter(is.na(Scores)) %>%
  pull(Variables) %>%
```

---

[1]Missing values do not need to be visible to exist: Incomplete designs are a good example showing that although the data do not have empty cells, it does contain a lot of missing data (the samples that were not evaluated by each panelist).

```
  unique())

sensory_clean <- sensory_long %>%
  filter(!(Variables %in% attr_rmv)) %>%
  pivot_wider(names_from=Variables, values_from=Scores)
```

This procedure removed the 7 attributes that contained missing values, leading
to a table with 99 rows and 29 columns (instead of 36).

9.1.2.3 Impute missing values

Rather than removing missing data, another (better?) strategy is to impute
missing values. Here again, many strategies can be considered, starting with
replacing them with a fixed value. Such approach is usually not the most suitable
one, yet it can be relevant in certain cases. For instance, in a CATA task, missing
values would often be replaced by 0 (not ticked).

To replace missing values with a fixed value, the `replace_na()` can be used.
When applied to a tibble, this function requires you defining using `list()` the
columns to apply it to, and which values to use (each column being treated
separately).

For convenience, let's apply it to `sensory` by replacing missing values for `Sour`
by the value `888` and for `Light` with `999` (we use these extreme values to track
changes more easily):

```
sensory %>%
  replace_na(list(Sour = 888, Light = 999)) %>%
  dplyr::select(Judge, Product, Sour, Light)
```

When dealing with intensity scale, it is more frequent to replace missing values
by the mean score for that product and attribute. When the test is duplicated,
the mean provided by this *panelist x product x attribute* across the different repe-
titions available is even preferred as it maintains individual variability (e.g. scale
usage) within the scores.

Such approach is then can be done in two steps: 1. Compute the mean (since
we do not have duplicates, we use the mean per product); 2. Combine it to the
data

For simplicity, `sensory_long` is used as starting point:

```
prod_mean <- sensory_long %>%
  group_by(Product, Variables) %>%
  summarize(Mean = mean(Scores, na.rm=TRUE)) %>%
```

```
  ungroup()

sensory_long %>%
  full_join(prod_mean, by=c("Product","Variables")) %>%
  mutate(Scores = ifelse(is.na(Scores), Mean, Scores)) %>%
  dplyr::select(-"Mean") %>%
  pivot_wider(names_from=Variables, values_from=Scores) %>%
  dplyr::select(Judge, Product, Sour, Light)
```

As can be seen, the missing value associated to `J01` for `Light` and `P04` has been replaced by `40.7`. In fact, any missing values related to `P04` and `Light` would automatically be replaced by `40.7` here. For other products (and other attributes), their respective means would be used.

When the model used to impute missing values is fairly simple (here, replacing by the mean correspond to a simple 1-way ANOVA), the imputation can be done directly through the `impute_lm()` function from the `{simputation}` package. To mimic the previous approach, the one-way ANOVA is being used[2]. Here, missing data for both `Sour` and `Light` are being imputed independently using the same model:

```
library(simputation)

sensory %>%
  impute_lm(Sour + Light ~ Product) %>%
  dplyr::select(Judge, Product, Sour, Light)
```

As can be seen, this procedure provides the same results as before, but in less steps!

It can be noted that in some situations, implementing missing values using such ANOVA (or regression) model can lead to aberrations. It is for instance the case when the imputed values falls outside the scale boundaries. To avoid such situations, `{simputation}` also provides other more advanced alternatives including (amongst others) `impute_rf()` which uses random forest to impute the missing values.

Last but not least, imputation of missing values could also be done in a multivariate way, by using the structure of the data (e.g. correlation) to predict the missing values. This is the approach proposed in the `{missMDA}` package. Since our data are numeric, the imputation is done through PCA with the `imputePCA()` function. Note that here, the imputed values are stored in the object `.$completeObs` (here, `sensory` is used):

---

[2]It is worth noticing that the individual differences could also be included by simple adding the Judge effect in the model.

```
library(missMDA)
imputePCA(sensory, quali.sup=1:4, method="EM")$completeObs %>%
  dplyr::select(Judge, Product, Sour, Light)
```

In this case, it can be seen that the missing value for `J01-P04-Light` has been replaced by the value `31.8`.

9.1.2.4 Limitations

As we have seen, there are different ways to implement missing values, and the different algorithms will likely impute them with different values. Hence, the overall results can be affected, and there is no way to know which solution is the most suitable for our study. Still, it is recommended to treat the missing values, and to chose the right strategy that is the most adapted to the data.

However, since most imputation methods involve modeling, applying them to variables with a high missing values rate can introduce bias in the data. Let's consider a situation in which assessors are evaluating half the product set using a BIB. The data hence collected contained 1/2 missing values. By imputing the missing values, each prediction is proportionally based on one unique value. And ultimately, any further analyses on this data would be based on half measured and half *fictive* data.

9.1.3 Design Inspection

As part of the data inspection, it is relevant to check that all the data are in line with their intrinsic nature, look for outliers, etc. Another point of interest - quite specific to sensory and consumer data - is to ensure that the design is well balanced, and handles the first-order and carry-over effects. This step is particularly important for those who analyze the data but were not involved from the start in that study (and hence were not involved when the test was set-up).

Let's show a simple procedure that would check part of the quality of a design. Since neither in *Sensory Profile.xlsx*, nor in *TFEQ.xlsx* the design is not available, the `sensochoc` data stored in the `{SensoMineR}` is used.

To load (and clean) the data, let's run these lines of code:

```
library(SensoMineR)

data(chocolates)

dataset <- sensochoc %>%
```

```
  as_tibble() %>%
  mutate(across(c(Panelist, Session, Rank, Product), as.character))
```

The data consist in 6 products (`Product`) evaluated by 29 panelists (`Panelist`) in duplicates (`Session`). The presentation order is stored in `Rank`.

To evaluate whether the products have been equally presented at each position, a simple cross-count between `Product` and `Rank` is done. This can easily be done using the `xtabs()` function:

```
xtabs(~Product + Rank, data=dataset)
```

Such table can also be obtained using `group_by()` and `count()` to get the results in a tibble:

```
dataset %>%
  group_by(Product) %>%
  count(Rank) %>%
  ungroup() %>%
  pivot_wider(names_from=Rank, values_from=n)
```

As we can see, the design is not perfectly balanced, as `choc2` is evaluated 11 times in the 1st, 4th, and 6th position, but only 7 times in the 5th position.

The next step then consists in evaluating how often each product is assessed before the other products (carry-over effect). Since this information is not directly available in the data, it needs to be added.

Let's start with extracting the information available, i.e. the order of each product for each panelist and session:

```
current <- dataset %>%
  dplyr::select(Panelist, Product, Session, Rank) %>%
  mutate(Rank = as.numeric(Rank))
```

An easy way to add the `Previous` product information as a new column in the data is by replacing `Rank` by `Rank + 1` in `current` (all new positions larger than the number of products is filtered).

```
previous <- current %>%
  rename(Previous = Product) %>%
  mutate(Rank = Rank + 1) %>%
  filter(Rank <= length(unique(dataset$Product)))
```

This new data is merged to `current` by `Panelist`, `Session`, and `Rank`:

```
(cur_prev <- current %>%
  left_join(previous, by=c("Panelist", "Session", "Rank")))
```

As can be seen, the products that are evaluated first get `NA` in `Previous`, and for each rank r (r > 1), `Previous` gets the product that was evaluated at rank r-1.

To evaluate the carry-over effect, the only thing to do is to cross-count `Product` and `Previous` (here, the results are split per `Session`):

```
cur_prev %>%
  group_by(Session, Product, Previous) %>%
  count() %>%
  ungroup() %>%
  mutate(Product = factor(Product, levels=paste0("choc",1:6)),
         Previous = factor(Previous, levels=c("NA",paste0("choc",1:6)))) %>%
  arrange(Previous) %>%
  pivot_wider(names_from=Previous, values_from=n, values_fill=0) %>%
  arrange(Product) %>%
  split(.$Session)
```

As expected, the table shows that a product is never evaluated twice in a row (the diagonal contains 0). Here again, the design is not optimal since `choc1` has been evaluated 3 times before `choc2` and 6 times before `choc5` in the first session.

Finally, the `NA` column refers to the time that products did not have a product tested before, meaning that they were evaluated first.

9.2 Clean

As mentioned in the introduction of this chapter, there is a thin line between *Data Inspection* and *Data Manipulation*, as both steps share many common practices. Here, we are limiting ourselves on handling variables and their type. For a full overview, we encourage the readers to look at (REF DATA MANIPULATION) to see other practices on how to handle data.

9.2.1 Handling Data Type

The data used in this section is stored in *TFEQ.xlsx*. Let's import this file as suggested in (REFERENCE).

```
library(readxl)
library(here)

file_path <- here("Data", "TFEQ.xlsx")

demo_var <- read_xlsx(file_path, sheet="Variables") %>%
  dplyr::select(Code, Name)

demo_lev <- read_xlsx(file_path, sheet="Levels") %>%
  dplyr::select(Question, Code, Levels) %>%
  inner_join(demo_var, by=c("Question"="Code")) %>%
  dplyr::select(-Question)

demographic <- read_xlsx(file_path, sheet="Data", skip=1, col_names=unlist(demo_var$Name))
```

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 data set is stored in 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.

```
demographic
```

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

```
str(demographic)
```

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

9.2.1.1   Numerical Data

Since a large proportion of the research done is quantitative, it is no surprise that our data 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.

9.2.1.2   Binary Data

Another common type that seems 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:

```r
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)
```

9.2.1.3   Nominal Data

Nominal data is any data that is not numerical nor binary. 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. 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 is 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 procedures 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 directly provided by `levels()` and `nlevels()`.

Let's compare a variable set as `factor` and `character` by using a simple hand-made example:

```
(example <- demographic %>%
  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 matches the way they appear in the data. This means that any data collected using a structured scale will often lose its natural order.

When set as `factor`, the factor levels (including their order) are informed, and does not depend necessarily 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. In our case, the levels are read from the data an are automatically being reordered alphabetically (note that J10 and J100 appear before J2 for instance.)

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.

```
example <- demographic %>%
  dplyr::select(Judge) %>%
  mutate(Judge_fct = factor(Judge, str_sort(Judge, numeric=TRUE)))
levels(example$Judge_fct)
```

Now the levels are sorted, let's 'remove' some respondents by only keeping the 20 first ones (J1 to J20), and re-run the previous code:

```
(example_reduced <- example %>%
  filter(Judge %in% paste0("J",1:20)))

unique(example_reduced$Judge)
length(unique(example_reduced$Judge))

levels(example_reduced$Judge_fct)
nlevels(example_reduced$Judge_fct)
```

After filtering some respondents, it can be noticed that the variable set as character only contains 19 elements (J18 doesn't exist in the data), whereas the column set as factor still contains the 107 entries (most of them not having any recordings).

```
example_reduced %>%
  count(Judge, .drop=FALSE)

example_reduced %>%
  count(Judge_fct, .drop=FALSE)
```

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 still use the original count (so 106 here)!

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 `demographic` 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 its levels:

```
example = demographic %>%
  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`. Let's now regroup `Rurban` and `Rural` together under `Rural`, and change the order by ensure that `Rural` appears before `Urban`:

```
example = demographic %>%
  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);
- `fct_inorder()` takes uses the same order as read in the data (particularly useful with `pivot_longer()` for instance);
- `fct_rev()` reverses the order of the levels (particularly useful in graphs).

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;
- `str_sort()` to order alphabetically (or by respecting numbers, as shown previously) its elements.

9.2.2 Converting between Types

When importing data, variables may not always be associated to the right type (remember, each type has its own properties). For example, respondents or products coded numerically would be defined as integers rather than strings after importation. Without manual correction, this would lead to the wrong analyses! Hence, they need to be converted to the right type.

In the following sections, the function `mutate()` is used to 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`). By renaming the new variable with the same name as the original one, we overwrite it by simply changing its type, here.

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 in the factor level order.

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

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

9.2.3 Conclusions

As can be seen, it is very important to verify (and correct!) the type of each variable as it can lead to running the wrong analysis otherwise. When using tibbles, such inspection is quite straightforward since printing the table on screen automatically shows the type of each variable. However, when presented as a matrix or as a data frame, such information is not directly visible:

```
demographic %>%
  as.data.frame()
```

Since each type has its advantages and drawbacks, it is convenient to regularly transit from one to another. This procedure will occur very often in the next sections, which will give you the chance to get accustomed to the procedure.

Chapter 10

Data Analysis

10.1 Chapter Overview

In this chapter, the different techniques presented in 8, 4 and in 5 will be applied to our biscuit data, including *Sensory Profile.xlsx*. The analyses presented are non-exhaustive, but tackle some well-known analyses often used in sensory and consumer science. The following sections are divided based on the type of data to consider and their corresponding analysis.

Logically, most of this chapter is build around `{tidyverse}`, but also includes some more specific packages such as `SensoMineR` and `FactoMineR`.

```
library(tidyverse)
library(here)
library(readxl)
```

10.2 Sensory Data

Let's start with the analysis of our sensory data. If not done already, import the file *Sensory Profile.xlsx*. As usual, we use the `{here}` and `{readxl}` packages for the data importation. Here, we decide to extend a bit the data by also adding some of the products' information:

```
file_path <- here("data", "Sensory Profile.xlsx")
p_info <- read_xlsx(file_path, sheet = "Product Info") %>%
  dplyr::select(-Type)

sensory <- read_xlsx(file_path, sheet="Data") %>%
```

```
  inner_join(p_info, by="Product") %>%
  relocate(Protein:Fiber, .after=Product)
```

Typically, sensory scientists would first seek to determine whether there are differences between samples for the different attributes. This is done through Analysis of Variance (ANOVA) and can be done using the `lm()` or `aov()` functions. If we would want to run the ANOVA for `Sweet`, the code would look like this (here we consider the 2-way ANOVA evaluating the Product and Assessor effects):

```
sweet_aov <- lm(Sweet ~ Product + Judge, data=sensory)
anova(sweet_aov)
```

We could duplicate this code for each single attribute, but this would be quite tedious for large number of attributes. Moreover, this code is sensitive to the way the variables are named, and hence might not be suitable for other data sets. Instead, we propose two solutions, one using `split()` combined with `map()` and one involving `nest_by()` to run this analysis automatically.

For both these solutions, the data should be stored in the long and thin form, which can be obtained using `pivot_longer()`:

```
senso_aov_data <- sensory %>%
  pivot_longer(Shiny:Melting, names_to="Attribute", values_to="Score")
```

From this structure, the first approach consists in splitting the data by attribute. Once done, we run the ANOVA for each data subset (the model is then defined as `Score ~ Product + Judge`) automatically using `map()`[1], and we extract the results of interest using the {broom} package[2]

Ultimately, the results can be combined again using `enframe()` and `unnest()`.

```
senso_aov1 <- senso_aov_data %>%
  split(.$Attribute) %>%
  map(function(data){

    res <- broom::tidy(anova(lm(Score ~ Product + Judge, data=data)))
    return(res)

  }) %>%
  enframe(name="Attribute", value="res") %>%
  unnest(res)
```

---

[1]The `map()` function applies the same function to each element of a list automatically: It is hence equivalent to a `for ()` loop, but in a neater and more efficient way.

[2]The {broom} package is a very useful package that convert statistical objects into tidy tables.

The second approach nests the analysis by attribute (meaning the analysis is done for each attribute separately, a bit like `group_by()`). In this case, we store the results of the ANOVA in a new variable called `mod`.

Once the analysis is done, we summarize the info stored in `mod` by converting it into a tibble using `{broom}`:

```
senso_aov2 <- senso_aov_data %>%
  nest_by(Attribute) %>%
  mutate(mod = list(lm(Score ~ Product + Judge, data=data))) %>%
  summarise(broom::tidy(anova(mod))) %>%
  ungroup()
```

The two approaches return the exact same results.

Let's dig into the results by extracting the attributes that do not show significant differences at 5%. Since the `tidy()` function from `{broom}` tidies the data, all the usual data transformation can be performed. Let's filter only the `Product` effect under `term`, and let's order the `p.value` decreasingly:

```
senso_aov1 %>%
  filter(term == "Product") %>%
  dplyr::select(Attribute, statistic, p.value) %>%
  arrange(desc(p.value)) %>%
  mutate(p.value = round(p.value, 3))
```

As can be seen, the products do not show any significant differences at 5% for 4 attributes: `Cereal flavor` (p=0.294), `Roasted odor` (p=0.193), `Astringent` (p=0.116), and `Sticky` (p=0.101).

Since we ran ANOVAs on multiple attributes, it is useful to visualize the results graphically as a barchart (we use here the F-values). We propose to order the attributes based on their decreasing F-values, and by colour-coding them based on the significance:

```
senso_aov1 %>%
  filter(term == "Product") %>%
  dplyr::select(Attribute, statistic, p.value) %>%
  mutate(Signif = ifelse(p.value <= 0.05, "Signif.", "Not Signif.")) %>%
  mutate(Signif = factor(Signif, levels=c("Signif.", "Not Signif."))) %>%
  ggplot(aes(x=reorder(Attribute, statistic), y=statistic, fill=Signif))+
  geom_bar(stat="identity")+
  scale_fill_manual(values=c("Signif."="forestgreen", "Not Signif."="orangered2"))+
  ggtitle("Sensory Attributes","(The attributes are sorted according to product-wise F-values)")+
  theme_bw()+
  xlab("")+
```

```
  ylab("F-values")+
  coord_flip()
```

It appears that the evaluated biscuits differ the most (top 5) for `Initial hardness`, `Shiny`, `Dairy flavor`, `External color intensity`, and `Thickness`.

Note that as an alternative, we could use the `decat()` function from the `{SensoMineR}` package. This function performs ANOVA on a set of attributes (presented in subsequent columns), followed by some t-test that highlights which samples are significantly more (or less) intense than average for each attribute.

Once the significant differences have been checked, a follow-up analysis consists in visualizing these differences in a multivariate way. Such visualization is often done through Principal Component Analysis (PCA).

In practice, PCA is performed on the sensory profiles, i.e. the mean table crossing the products in rows, and the sensory attributes in columns. Let's start with building such table (see 4 for more examples):

```
senso_mean <- sensory %>%
  pivot_longer(Shiny:Melting, names_to="Attribute", values_to="Score") %>%
  dplyr::select(-Judge) %>%
  pivot_wider(names_from=Attribute, values_from=Score, values_fn=mean)
```

Such table is then submitted to PCA. R proposes many solutions to run such analysis, including the `prcomp()` and `princomp()` functions from the `{stats}` package. However, we prefer to use `PCA()` from the `{FactoMineR}` as it is more complete and it proposes many options that are very useful in sensory and consumer research (e.g. it generates the graphics automatically, and allows projecting supplementary individuals and/or variables).

Note that `PCA()` uses numerical data, the individuals' names (here the product names) being set as row names in a data frame or matrix. However, tibble do not have row names, meaning that this should be taken care of. Fortunately, an easy solution exists by converting the tibble into a data frame (`as.data.frame()`) and by passing the `Product` column into row names with `column_to_rownames(var="Product")`. Additionally, the data also contain two qualitative variables in `Protein` and `Fiber`. These 2 variables can either be removed prior to running the analysis, or better be projected as supplementary through the `quali.sup` parameter from `PCA()`. Finally, since `POpt` is an optimized sample, we are not including it in the analysis per se (it is not contributing to the construction of the dimensions). Instead we project it as supplementary (through `ind.sup`) to illustrate where it would be located on the space if it were.

```
library(FactoMineR)

senso_pca <- senso_mean %>%
  arrange(Product) %>%
  as.data.frame() %>%
  column_to_rownames(var="Product") %>%
  PCA(., ind.sup=nrow(.), quali.sup=1:2, graph=FALSE)
```

The `PCA()` function can generate the plots either in `{base}` R language, or in `{ggplot2}`.

However, we like to use a complementary package called `{factoextra}` which re-creates most plots from `{FactoMineR}` (and more) as a `ggplot()` object. This comes in very handy as you can benefit from the flexibility offered by `ggplot()`.

The score map (the product map) from `PCA()` is created with `fviz_pca_ind()`, whereas the variables' loadings (attribute correlations) is created with `fviz_pca_var()`. `fviz_pca_biplot()` is used to produce the so-alled biplot.

To illustrate this, let's reproduce the product map by coloring the products using the supplementary variables (`Protein` and `Fiber` content). This can easily be done through the `habillage` parameter from `fviz_pca_ind()`, which can either take a numerical value (position) of the name of the qualitative variable.

```
library(factoextra)

fviz_pca_ind(senso_pca, habillage="Protein", repel=TRUE)
fviz_pca_ind(senso_pca, habillage=2, repel=TRUE)
fviz_pca_var(senso_pca)
fviz_pca_biplot(senso_pca)
```

Here, `repel = TRUE` uses `geom_text_repel()` from `{ggrepel}` (rather than `geom_text()` from `{ggplot2}`) to avoid having labels overlapping.

On the first dimension, `P10` is opposed to `P09` and `P03` as it is more intense for attributes such as `Sweet`, and `Dairy flavor` for example, and less intense for attributes such as `Dry in mouth` and `External color intensity`. On the second dimension, `P08`, `P06`, and `POpt` are opposed to `P02` and `P07` as they score higher for `Qty of inclusions`, and `Initial hardness`, and score lower for `RawDough flavor` and `Shiny`.

To go deeper, many more visualizations can be produced. Amongst others, the scree plot (evolution of the eigenvalues across dimensions) to decide how many dimensions to consider, representation of the space on other dimensions (by default, dimension 1 and dimension 2 are created as they explain the maximum variability), or filtering products based on their contribution or quality

of representation can be mentioned. For more information, refer to the examples provided in `?plot.PCA()`, `?fviz_pca()` etc., or in the book (REF BOOK {factoextra}).

10.3   Demographic and Questionnaire Data

The *TFEQ.xlsx* file contains descriptive (i.e. *demographic*) information regarding the consumers and their food-related personality traits (i.e. *TFEQ*). This file has three tabs denoted as *Data*, *Variables*, and *Levels*:

- *Data* contains the data, which is coded;
- *Variables* provides information (e.g. name, information) related to the different variables present in *Data*;
- *Levels* provides information about the different levels each variable can take.

Let's start with importing this data set. The importation is done in multiple steps as following:

```r
file_path <- here("Data", "TFEQ.xlsx")
excel_sheets(file_path)

demo_var <- read_xlsx(file_path, sheet="Variables") %>%
  dplyr::select(Code, Name)

demo_lev <- read_xlsx(file_path, sheet="Levels") %>%
  dplyr::select(Question, Code, Levels) %>%
  inner_join(demo_var, by=c("Question"="Code")) %>%
  dplyr::select(-Question)

demographic <- read_xlsx(file_path, sheet="Data", skip=1, col_names=unlist(demo_var$Nam
```

10.3.1   Demographic Data: Frequency and Proportion

For this demographic data file, let's start by having a look at the partition of consumers for each of the descriptive variables. This is done by computing the frequency and proportion (in percentage) attached to each level of `Living area`, `Housing`, `Income range`, and `Occupation`. To obtain such a table, let's start by selecting only the columns corresponding to these variables together with `Judge`.

Since data from surveys and questionnaires are often coded (here, answer #6 to question `Q10` means "Student", while answer #7 to the same question means

"Qualified worker"), they first need to be decoded. In our case, the key to decode the data is stored in `demo_lev`.

Different strategies to decode the data are possible. One straight-forward strategy consists in automatically decoding each variable using `mutate()` and `factor()`. Another approach is considered here: Let's start with building a long thin tibble with `pivot_longer()` that we merge to `demo_lev` by `Question` and `Response` using `inner_join()`. We prefer this solution here as it is simpler, faster, and independent of the number of variables to decode.

Once done, we can aggregate the results by `Question` and `Levels` (since we want to use the level information, not their code) and compute the frequency (`n()`) and the proportion $(\text{N}/\text{sum(N)})^3$.

```
library(formattable)

demog_reduced <- demographic %>%
  dplyr::select(Judge, `Living area`, Housing, `Income range`, `Occupation`) %>%
  pivot_longer(-Judge, names_to="Question", values_to="Response") %>%
  inner_join(demo_lev, by=c("Question"="Name", "Response"="Code")) %>%
  group_by(Question, Levels) %>%
  summarize(N = n()) %>%
  mutate(Pct = percent(N/sum(N), digits=1L)) %>%
  ungroup()
```

Histograms are a nice way to visualize proportions and to compare them over several variables. Such histograms can be obtained by splitting `demog_reduced` by `Question` and by creating them using either `N` or `Pct` (we are using `Pct`)[4]. Of course, it is automated across all questions using `map()`.

```
demog_reduced %>%
  split(.$Question) %>%
  map(function(data){

    var = data %>%
      pull(Question) %>%
      unique() %>%
      as.character()

    ggplot(data, aes(x=reorder(Levels, Pct), y=Pct, label=Pct))+
      geom_bar(stat="identity", fill="grey50")+
      geom_text(aes(y = Pct/2), colour="white")+
```

---

[3]We use the package {`formattable`} to print the results in percentage using one decimal. As an alternative, we could have used `percent()` from the {`scales`} package.

[4]Here, the histograms are ordered decreasingly (`reorder`) and are represented horizontally (`coord_flip()`).

```
    xlab("")+
    ylab("")+
    ggtitle(var)+
    theme_bw()+
    coord_flip()

})
```

10.3.2   Personality traits and eating behavior: TFEQ data

In the same data set, consumers also answered some questions that reflect on their eating behavior and their relation to food (REF REF). These questions can be categorized into three groups relating to three underlying factors, respectively: Disinhibition (variables starting with `D`), Restriction (variables starting with `R`), and sensitivity to Hunger (variables starting with `H`).

In order to analyze these three factors separately, we first need to select the corresponding variables. As we have seen earlier, such selection could be done by combining `dplyr::select()` to `starts_with("D")`, `starts_with("R")`, and/or `starts_with("H")`. However, this solution is not satisfactory as it also selects other variables that would start with any of these letters (e.g. Housing).

Instead, let's take advantage of the fact that variable names have a recurring pattern (they all start with the letters D, R, or H, followed by a number) to introduce the notion of *regular expression.*

Regular expressions are coded expression that allows finding patterns in names. In practice, generating a regular expression can be quite complex as it is an abstract concept which follows very specific rules. Fortunately, the package `{RVerbalExpression}` is a great assistant as it generates the regular expression for you thanks to understandable functions. To create a regular expression using `{RVerbalExpression}`, we should first initiate it by calling the function `rx()` to which any relevant rules can be added. In our case, the variables must start with any of the letter R, D, or H, followed by a number (or more, as values go from 1 to 21). This can be done using the following code:

```
library(RVerbalExpressions)

rdh <- rx() %>%
  rx_either_of(c("R","D","H")) %>%
  rx_digit() %>%
  rx_one_or_more()

rdh
```

`rdh` contains the regular expression we were looking for.   We can then
`dplyr::select()` any variable that fits our regular expression by using the
function `matches()`.

```
demographic %>%
  dplyr::select(matches(rdh))
```

In order to build a separate frequency table for each of these variables, we create
a function called `myfreq()` which will automatically compute the frequency for
each level, and the corresponding percentage.

```
myfreq <- function(data, info){

  var = unique(unlist(data$TFEQ))
  info <- info %>%
    filter(Name == var)

  res <- data %>%
    mutate(Response = factor(Response, levels=info$Code, labels=info$Levels)) %>%
    arrange(Response) %>%
    group_by(Response) %>%
    summarize(N = n()) %>%
    mutate(Pct = percent(N/sum(N), digits=1L)) %>%
    ungroup()

  return(res)
}
```

We then apply this function to each variable separately using `map()` after piv-
oting all these variables of interest (`pivot_longer()`) and splitting the data by
TFEQ question:

```
TFEQ_freq <- demographic %>%
  dplyr::select(Judge, matches(rdh)) %>%
  pivot_longer(-Judge, names_to="TFEQ", values_to="Response") %>%
  split(.$TFEQ) %>%
  map(myfreq, info=demo_lev) %>%
  enframe(name = "TFEQ", value="res") %>%
  unnest(res) %>%
  mutate(TFEQ = factor(TFEQ, levels=unique(str_sort(.$TFEQ, numeric=TRUE)))) %>%
  arrange(TFEQ)
```

Accordingly, we can create histograms that represent the frequency distribution
for each variable. But let's suppose that we only want to display variables related

to Disinhibition. To do so, we first need to generate the corresponding regular expression (only selecting variables starting with "D") to filter the results before creating the plots:

```r
d <- rx() %>%
  rx_find("D") %>%
  rx_digit() %>%
  rx_one_or_more()

TFEQ_freq %>%
  filter(str_detect(TFEQ, d)) %>%
  ggplot(aes(x=Response, y=Pct, label=Pct))+
  geom_bar(stat="identity" , fill="grey50")+
  geom_text(aes(y = Pct/2), colour="white")+
  theme_bw()+
  theme(axis.text = element_text(hjust=1, angle=30))+
  facet_wrap(~TFEQ, scales="free")
```

Structured questionnaires such as the TFEQ are very frequent in sensory and consumer science. They are used to measure individual patterns as diverse as personality traits, attitudes, food choice motives, engagement, social desirability bias, etc. Ultimately, the TFEQ questionnaire consists in a set of structured questions whose respective answers combine to provide a TFEQ score (actually, three scores, one for Disinhibition, one for Restriction and one for sensitivity to Hunger). This TFEQ scores translate into certain food behavior tendencies.

However, computing the TFEQ scores is slightly more complicated than adding the scores of all TFEQ questions together. Instead, they follow certain rules that are stored in the *Variables* sheet in *TFEQ.xlsx*. For each TFEQ question, the rule to follow is provided by `Direction` and `Value`, and works as following: if the condition provided by `Direction` and `Value` is met, then the respondent gets a 1, else a 0. Ultimately, the TFEQ is the sum of all these evaluations.

Let's start by extracting this information (`Direction` and `Value`) from the sheet *Variables* for all the variables involved in the computation of the TFEQ scores. We store this in `var_drh`.

```r
var_rdh <- read_xlsx(file_path, sheet="Variables") %>%
  filter(str_detect(Name, rdh)) %>%
  dplyr::select(Name, Direction, Value)
```

This information is added to `demographic`.

```r
TFEQ <- demographic %>%
  dplyr::select(Judge, matches(rdh)) %>%
```

```r
  pivot_longer(-Judge, names_to="DHR", values_to="Score") %>%
  inner_join(var_rdh, by=c("DHR"="Name"))
```

Since we need to evaluate each assessors' answer to the TFEQ questions, we create a new variable `TFEQValue` which takes a 1 if the corresponding condition is met, and a 0 otherwise. Such approach is done through `mutate()` combined with a succession of intertwined `ifelse()` functions.[5]

```r
TFEQ_coded <- TFEQ %>%
  mutate(TFEQValue = ifelse(Direction == "Equal" & Score == Value, 1,
                      ifelse (Direction == "Superior" & Score > Value, 1,
                            ifelse(Direction == "Inferior" & Score < Value, 1, 0)))) %>%
  mutate(Factor = ifelse(str_starts(.$DHR, "D"), "Disinhibition",
                      ifelse(str_starts(.$DHR, "H"), "Hunger", "Restriction"))) %>%
  mutate(Factor = factor(Factor, levels=c("Restriction","Disinhibition","Hunger")))
```

Ultimately, we compute the TFEQ Score by summing across all `TFEQValue` per respondent, by maintaining the distinction between each category. Note that the final score is stored in `Total`, which corresponds to sum across categories:

```r
TFEQ_score <- TFEQ_coded %>%
  group_by(Judge, Factor) %>%
  summarize(TFEQ = sum(TFEQValue)) %>%
  mutate(Judge = factor(Judge, levels=unique(str_sort(.$Judge, numeric=TRUE)))) %>%
  arrange(Judge) %>%
  pivot_wider(names_from=Factor, values_from=TFEQ) %>%
  mutate(Total = sum(across(where(is.numeric))))
```

Such results can then be visualized graphically, for instance by representing the distribution of `TFEQ_score` for the 3 categories of questions (or TFEQ-factor):

```r
TFEQ_score %>%
  dplyr::select(-Total) %>%
  pivot_longer(-Judge, names_to="Factor", values_to="Scores") %>%
  ggplot(aes(x=Scores, colour=Factor))+
  geom_density(lwd=1.5)+
  xlab("TFEQ Score")+
  ylab("")+
  ggtitle("Distribution of the Individual TFEQ-factor Scores")+
  theme_bw()
```

---

[5]The function `ifelse()` takes three parameters: 1. the condition to test, 2. the value or code to run if the condition is met, and 3. the value or code to run if the condition is not met.

10.4   Consumer Data

The analysis of consumer data usually involves the same type of analysis as the ones for sensory data (e.g. ANOVA, PCA, etc.), but the way the data is being collected (absence of duplicates) and its underlying nature (affect vs. descriptive) require some adjustments.

Let's start by importing the consumer data that is stored in *Consumer Test.xlsx.* Here, we import two sheets, one with the consumption time and number of biscuits (stored in `Nbiscuit`), and one with different consumer evaluations of the samples (stored in `consumer`)

```r
file_path <- here("Data","Consumer Test.xlsx")

Nbiscuit <- read_xlsx(file_path, sheet="Time Consumption") %>%
  mutate(Product = str_c("P", Product)) %>%
  rename(N = `Nb biscuits`)

consumer <- read_xlsx(file_path, sheet="Biscuits") %>%
  rename(Judge=Consumer, Product=Samples) %>%
  mutate(Judge = str_c("J", Judge), Product = str_c("P", Product)) %>%
  inner_join(Nbiscuit, by=c("Judge", "Product"))
```

Similarly to the sensory data, let's start with computing the mean liking score per product after the first bite (`1stbite_liking`) and at the end of the evaluation (`after_liking`).

```r
consumer %>%
  dplyr::select(Judge, Product, `1stbite_liking`, `after_liking`) %>%
  group_by(Product) %>%
  summarise(across(where(is.numeric), mean))
```

A first glance at the table shows that there are clear differences between the samples (within a liking variable), but little difference between liking variables (within a sample).

Of course, we want to know if differences between samples are significant. We thus need to perform an ANOVA (testing for the product effect) followed up by a paired comparison test (here Tukey's HSD). To do so, the `{agricolae}` package is a good solution, as it is simple to use and has all its built-in tests working in the same way.

```r
library(agricolae)

liking_start <- lm(`1stbite_liking` ~ Product + Judge, data=consumer)
```

```
liking_start_hsd <- HSD.test(liking_start, "Product")$groups %>%
  as_tibble(rownames = "Product")

liking_end <- lm(`after_liking` ~ Product + Judge, data=consumer)
liking_end_hsd <- HSD.test(liking_end, "Product")$groups %>%
  as_tibble(rownames = "Product")
```

Both at the start and at the end of the evaluation, significant differences (at 5%) in liking between samples are observed according to Tukey's HSD test.

To further compare the liking assessment of the samples after the first bite and at the end of the tasting, the results obtained from `liking_start_hsd` and `liking_end_hsd` are combined. We then represent the results in a bar-chart:

```
list(Start = liking_start_hsd %>% rename(Liking=`1stbite_liking`),
     End = liking_end_hsd %>% rename(Liking=`after_liking`)) %>%
  enframe(name = "Moment", value = "res") %>%
  unnest(res) %>%
  mutate(Moment = factor(Moment, levels=c("Start","End"))) %>%
  ggplot(aes(x=reorder(Product, -Liking), y=Liking, fill=Moment))+
  geom_bar(stat="identity", position="dodge")+
  xlab("")+
  ggtitle("Comparison of the liking scores at the start and at the end of the evaluation")+
  theme_bw()
```

As can be seen, the pattern of liking scores across samples is indeed very stable across the evaluation, particularly in terms of rank. At the individual level, such linear relationship is also observed (here for the first 12 consumers):

```
consumer %>%
  dplyr::select(Judge, Product, Start=`1stbite_liking`, End=`after_liking`) %>%
  filter(Judge %in% str_c("J",1:12)) %>%
  mutate(Judge = factor(Judge, levels=unique(str_sort(.$Judge, numeric=TRUE)))) %>%
  ggplot(aes(x=Start, y=End))+
  geom_point(pch=20, cex=2)+
  geom_smooth(method="lm", formula="y~x", se=FALSE)+
  theme_bw()+
  ggtitle("Overall Liking", "(Assessment after first bite vs. end of the tasting)")+
  facet_wrap(~Judge)
```

For your own curiosity, we invite you to re-create the same graph by comparing the Liking score at the end of the evaluation (`after_liking`) with the liking score measured on the 9pt categorical scale (`end_liking 9pt`), and to reflect on the results obtained. Are the consumers consist in their evaluations?

Another interesting relationship to study involves the liking scores[6] and the number of cookies eaten by each consumer. We could follow the same procedure as before, but prefer to add here a filter to only show consumers with a significant regression line at 5%.

Let's start by creating a function called `run_reg()` that runs the regression analysis of the number of biscuits (`N`) in function of the liking score (`Liking`):

```
run_reg <- function(df){
  output <- lm(N ~ Liking, data=df)
  return(output)
}
```

After transforming the data, we apply this function to our data, but we are going to store two sorts of results (as list) in our tibbles: - `lm_obj` which corresponds to the results of the linear model (obtained with `run_reg()``) -glance' which contains some general results of the model incl. R2, the p-value, etc.

```
N_liking <- consumer %>%
  dplyr::select(Judge, Product, Liking=`end_liking 9pt`, N) %>%
  mutate(Liking = 10-Liking) %>%
  group_by(Judge) %>%
  nest() %>%
  ungroup() %>%
  mutate(lm_obj = map(data, run_reg)) %>%
  mutate(glance = map(lm_obj, broom::glance)) %>%
  unnest(glance) %>%
  filter(p.value <= 0.05) %>%
  arrange(p.value) %>%
  mutate(Judge = fct_reorder(Judge, p.value)) %>%
  unnest(data)
```

Ultimately, we can represent the relationship in a line chart:

```
ggplot(N_liking, aes(x=Liking, y=N))+
  geom_point(pch=20, cex=2)+
  geom_smooth(method="lm", formula="y~x", se=FALSE)+
  theme_bw()+
  ggtitle("Number of Biscuits vs. Liking","(Consumers with a significant (5%) regressi
  facet_wrap(~Judge, scales="free_y")
```

---

[6]We would like to remind the reader that the liking scores measured on the categorical scale was reverted since 1 defined "I like it a lot" and 9 "I dislike it a lot". To simplify the readability, this scale is reverted so that 1 corresponds to a low liking score, and 9 to a high liking score (in practice, we will take as value 10-score given).

10.5 Combining Sensory and Consumer Data

10.5.1 Internal Preference Mapping

Now we've analyzed the sensory and the consumer data separately, it is time to combine both data sets and analyzed them conjointly. A first analysis that can then be performed is the internal preference mapping, i.e. a PCA on the consumer liking scores in which the sensory attributes are projected as supplementary.

Such analysis is split in 3 steps: 1. The consumer data is re-organized in a wide format with the samples in rows and the consumers in columns; 2. The sensory mean table is joined to the consumer data (to do so, we need to make sure that the product names perfectly match in the two files); 3. A PCA is performed on the consumer data, the sensory descriptors being projected as supplementary onto that space.

```r
consumer_wide <- consumer %>%
  separate(Product, into = c("P", "Number"), sep = 1) %>%
  mutate(Number = ifelse(nchar(Number) == 1, str_c("0", Number), Number)) %>%
  unite(Product, P, Number, sep="") %>%
  dplyr::select(Judge, Product, Liking=`end_liking 9pt`) %>%
  mutate(Liking = 10-Liking) %>%
  pivot_wider(names_from=Judge, values_from=Liking)

data_mdpref <- senso_mean %>%
  inner_join(consumer_wide, by="Product")

# The first two columns are qualitative and are projected as supplementary through quali.sup=1:2
# The next 32 columns correspond to the sensory attributes which are projected as supplementary
res_mdpref <- data_mdpref %>%
  as.data.frame() %>%
  column_to_rownames(var="Product") %>%
  PCA(., quali.sup=1:2, quanti.sup=3:34, graph=FALSE)

# The sensory product can be coloured based on their Protein content through habillage=1 (1st qua
fviz_pca_ind(res_mdpref, habillage=1)
# For the variables representation, we only show the one with a decent quality of representation
fviz_pca_var(res_mdpref, label="quanti.sup", select.var=list(cos2=0.5), repel=TRUE)
```

As can be seen, the consumers are fairly in agreement as all the consumers (black arrows) are pointed in a similar direction. In overall, they seem to like biscuits that are sweet, with cereal flavor, and fatty/dairy flavor and odor, and dislike biscuits defined as astringent, dry in mouth, uneven and with dark external color.

10.5.2   Consumers Clustering

Although the data show a fairly good agreement between consumers, let's cluster them in more homogeneous groups based on liking.

Various solutions for clustering exist, depending on the type of distance (similarity or dissimilarity), the linkage (single, average, Ward, etc.), and of course the algorithm itself (e.g. AHC, k-means, etc.).

Here, we opt for the Agglomerative Hierarchical Clustering (AHC) with Euclidean distance (dissimilarity) and Ward criterion, as it is fairly common approach in Sensory and Consumer research. Such analysis can be done using `stats::hclust()` or `cluster::agnes()`.

Note that before computing the distance between consumers, it is advised to at least center their liking scores (subtracting their mean liking scores to each of their individual scores) as it allows grouping consumers based on their respective preferences, rather than on their scale usage (otherwise, consumers who scored high on all samples are grouped together, and separated from consumers who scored low on all samples, which isn't so much informative).

Let's start with computing the euclidean distance between each pair of consumers by using the `dist()` function.

```r
consumer_dist <- consumer_wide %>%
  as.data.frame() %>%
  column_to_rownames(var="Product") %>%
  scale(., center=TRUE, scale=FALSE) %>% # scale is a function that center (if TRUE) a
  t(.) %>% # the function t() transpose the table: here we want the consumers in rows
  dist(., method="euclidean")
```

We then run the AHC using the `hclust()` function and the `method = "ward.D2"` parameter, which is the equivalent to the `method = "ward"` for `agnes()`. To visualize the dendrogram, the `factoextra::fviz_dend()` function is used (here we propose to keep visualize the 2-clusters solution by setting `k=2`):

```r
res_hclust <- hclust(consumer_dist, method="ward.D2")

fviz_dend(res_hclust, k=2)
fviz_dend(res_hclust, k=2, type="phylogenic")
```

Since we are satisfied with the 2 clusters solution, we cut the tree (`cutree()` function) at this level using, hence generating a group of 74 and a group of 33 consumers.

```r
res_clust <- cutree(res_hclust, k=2) %>%
  as_tibble(rownames="Judge") %>%
  rename(Cluster = value) %>%
  mutate(Cluster = as.character(Cluster))

res_clust %>%
  count(Cluster)
```

Lastly, we compare visually the preference patterns between clusters by representing in a line chart the average liking score for each product provided by each cluster.

```r
mean_cluster <- consumer %>%
  separate(Product, into = c("P", "Number"), sep = 1) %>%
  mutate(Number = ifelse(nchar(Number) == 1, str_c("0", Number), Number)) %>%
  unite(Product, P, Number, sep="") %>%
  dplyr::select(Judge, Product, Liking=`end_liking 9pt`) %>%
  mutate(Liking = 10-Liking) %>%
  full_join(res_clust, by="Judge") %>%
  group_by(Product, Cluster) %>%
  summarize(Liking = mean(Liking), N=n()) %>%
  mutate(Cluster = str_c(Cluster," (",N,")")) %>%
  ungroup()

ggplot(mean_cluster, aes(x=Product, y=Liking, colour=Cluster, group=Cluster))+
  geom_point(pch=20)+
  geom_line(aes(group=Cluster), lwd=2)+
  xlab("")+
  scale_y_continuous(name="Average Liking Score", limits=c(1,9), breaks=seq(1,9,1))+
  ggtitle("Cluster differences in the appreciation of the Products (using hclust)")+
  theme_bw()
```

It appears that cluster 1 (74 consumers) particularly likes P10, P01, and eventually P05, and has a fairly flat liking pattern otherwise. On the other hand, the cluster 2 (33 consumers) expressed strong rejections towards P04 and P08, and like P10 and P01 the most.

The fact that both clusters agree on the best samples (P10 and P01) goes with our original assumption from the Internal Preference Mapping that the panel of consumers is fairly homogeneous in terms of preferences.

Remark: In the {FactoMineR} package, the HCPC() function also performs AHC but takes as starting point the results of a multivariate analysis (HCPC stands for Hierarchical Clustering on Principal Components). Although results should be identical in most cases, it can happen that results slightly diverge from agnes() and hclust() as it also depends on the number of dimensions kept in

the multivariate analysis and on the treatment of in-between clusters consumers. But more interestingly, `HCPC()` offers the possibility to *consolidate* the clusters by performing k-means on the solution obtained from the AHC (`consol=TRUE`).

10.5.3 Drivers of Liking

When combining sensory and consumer data collected on the same data, it is also relevant to understand which sensory properties of the products drive consumer's liking and disliking. Such evaluation can be done at the panel level, at a group level (e.g. clusters, users vs. non-users, gender, etc.), or even at the individual consumer level. Unless stated otherwise, the computations will be done at the panel level, but could be easily adapted to other levels if needed.

10.5.3.1 Correlation

Let's start by evaluating the simplest relationship between the sensory attributes and overall liking by looking at the correlation. Here, we are combining the average liking score per cluster to the sensory profile of the products. The correlations are the computed using the `cor()` function:

```
data_cor <- mean_cluster %>%
  dplyr::select(-N) %>%
  pivot_wider(names_from=Cluster, values_from=Liking) %>%
  inner_join(senso_mean %>% dplyr::select(-c(Protein, Fiber)), by="Product") %>%
  as.data.frame() %>%
  column_to_rownames(var="Product")

res_cor <- cor(data_cor)
```

Various packages can be used to visualize these correlations. We opt here for the function `ggcorrplot()` from the {ggcorrplot} package as it provides many interesting visualization in {ggplot2}.

Note that this package also comes with the function `cor_pmat()` which return the matrix of p-value associated to the correlations. This matrix of p-value can be used to hide correlations that are not significant at the level defined by the parameter `sig.level`.

```
library(ggcorrplot)

res_cor_pmat <- cor_pmat(data_cor)

ggcorrplot(res_cor, type="full", p.mat=res_cor_pmat, sig.level=0.05, insig="blank", lal
```

By looking at the correlations, the liking scores for cluster 1 (defined as `1 (74)`) are positively correlated with `Overall odor intensity`, `Fatty odor`, `Cereal flavor`, `Fatty flavor`, `Dairy flavor`, `Overall flavor persistence`, `Salty`, `Sweet`, `Warming`, `Fatty in mouth`, and `Melting`. They are also negatively correlated to `External color intensity`, `Astringent`, and `Dry in mouth`. Finally, it can be noted that the correlation between clusters is high with a value of 0.72.

10.5.3.2   Linear and Quadratic Regression

Although the correlation provides a first good idea of which attributes are linked to liking, it only measures linear relationships and it does not allow for inference. To overcome this particular limitations, linear and quadratic regressions can be used.

Let's start by combining the sensory data to the average liking score per product. To simplify the analysis, all the sensory attributes will be structured in the long format alike previous ANOVAs that we have performed. Also, in a quadratic model the quadratic term is evaluated by adding to the model an effect in which the sensory scores have been squared up, a second variable called `Score2 = Score^2` needs to be created.

```r
data_reg <- mean_cluster %>%
  dplyr::select(-N) %>%
  pivot_wider(names_from=Cluster, values_from=Liking) %>%
  inner_join(senso_mean %>% dplyr::select(-c(Protein, Fiber)), by="Product") %>%
  pivot_longer(Shiny:Melting, names_to="Attribute", values_to="Score") %>%
  mutate(Attribute = factor(Attribute, levels=colnames(senso_mean)[4:ncol(senso_mean)])) %>%
  mutate(Score2 = Score^2)
```

We then run both the linear and quadratic regression simultaneously by attribute for cluster 1 (`1 (74)`):

```r
res_reg <- data_reg %>%
  nest_by(Attribute) %>%
  mutate(lin_mod = list(lm(`1 (74)` ~ Score, data = data)),
         quad_mod = list(lm(`1 (74)` ~ Score + Score2, data = data)))
```

We extract the attributes that are significantly (at 5%) linked to liking. To do so, the results stored in `lin_mod` and `quad_mod` need unfolding (`summarize()`) and restructuring (`broom::tidy()`).

```r
lin <- res_reg %>%
  summarise(broom::tidy(lin_mod)) %>%
```

```
  ungroup() %>%
  filter(term == "Score", p.value <= 0.05) %>%
  pull(Attribute) %>%
  as.character()

quad <- res_reg %>%
  summarise(broom::tidy(quad_mod)) %>%
  ungroup() %>%
  filter(term == "Score2", p.value <= 0.05) %>%
  pull(Attribute) %>%
  as.character()
```

These attributes are then represented graphically against the liking Scores.

```
library(ggrepel)

df <- data_reg %>%
  filter(Attribute %in% unique(c(lin,quad)))

p <- ggplot(df, aes(x=Score, y=`1 (74)`, label=Product))+
  geom_point(pch=20, cex=2)+
  geom_text_repel()+
  theme_bw()+
  facet_wrap(~Attribute, scales="free_x")
```

Let's now add a regression line to the model. To do so, `geom_smooth()` is
being used with as `method = lm` combined to `formula = 'y ~ x'` for linear
relationships, and `formula = 'y ~ x + I(x^2)'` for quadratic relationships.
Note that for a given attribute, when both the linear and quadratic models are
significant, the quadratic model is being kept.

```
lm.mod <- function(df, quad){
  ifelse(df$Attribute %in% quad, "y~x+I(x^2)", "y~x")
}
```

We apply this function to our data by applying to each attribute (here we set
`se=FALSE` to remove the confidence intervals around the regression line):

```
p_smooth <- by(df, df$Attribute,
               function(x) geom_smooth(data=x, method=lm, formula=lm.mod(x, quad=quad)

p + p_smooth
```

All attributes except `Astringent` are linearly linked to liking. For `Astringent`,
the curvature is U-shaped: this does not show an effect of saturation as it would

have been represented as an inverted U-shape. Although the quadratic effect shows a better fit than the linear effect, having a linear effect would have been a good predictor as well.

10.5.4   External Preference Mapping

Ultimately, one of the goals of combining sensory and consumer data is to find within the sensory space the area that are liked/accepted by consumers. Since this approach is based on modeling and prediction, it may suggest area of the space with high acceptance potential which are not filled in by products yet. This would open doors to new product development.

To perform such analysis, the External Preference Mapping (PrefMap) could be used amongst other techniques. For more information on the principles of PrefMap, please refer to (ANALYZING SENSORY DATA WITH R or OTHER REFERENCES…).

To run the PrefMap analysis, the `carto()` function from `{SensoMineR}` is being used. This function takes as parameter the sensory space to consider (stored in `senso_pca$ind$coord`, here we will consider dimension 1 and dimension 2), the table of hedonic score (as stored in `consumer_wider`), and the model to consider (here we consider the quadratic model, so we use `regmod=1`).

Since `carto()` requires matrix or data frame with rownames for the analysis, the data needs to be slightly adapted (we also need to ensure that the products are in the same order in both files).

```
senso <- senso_pca$ind$coord[,1:2] %>%
  as_tibble(rownames="Product") %>%
  arrange(Product) %>%
  as.data.frame() %>%
  column_to_rownames(var="Product")

consu <- consumer_wide %>%
  arrange(Product) %>%
  as.data.frame() %>%
  column_to_rownames(var="Product")

library(SensoMineR)
PrefMap <- carto(Mat=senso, MatH=consu, regmod=1, graph.tree=FALSE, graph.corr=FALSE, graph.carto
```

From this map, we can see that the optimal area (dark red) is located on the positive side of dimension 1, between P01, P05, and P10 (as expected by the liking score).

Let's now re-build this plot using `{ggplot2}`.

The sensory space is stored in `senso`, whereas the surface response plot information is split between:

- `PrefMap$f1`: contains the coordinates on dimension 1 in which predictions have be made;
- `PrefMap$f2`: contains the coordinates on dimension 2 in which predictions have be made;
- `PrefMap$depasse`: contains the percentage of consumers that accept a product at each point of the space. This matrix is defined in such a way that `PrefMap$f1` links to the rows of the matrix, and `PrefMap$f2` links to the columns.

Last but not least, `POpt` (which coordinates are stored in `senso_pca$ind.sup$coord`) can be projected on that space in order to see if such sample was well optimized in terms of consumers' liking/preference.

Let's start with preparing the data by transforming everything back into a tibble:

```r
senso <- senso %>%
  as_tibble(rownames="Product")

senso_sup <- senso_pca$ind.sup$coord %>%
  as_tibble(rownames="Product")

dimnames(PrefMap$nb.depasse) <- list(round(PrefMap$f1,2), round(PrefMap$f2,2))
PrefMap_plot <- PrefMap$nb.depasse %>%
  as_tibble(rownames="Dim1") %>%
  pivot_longer(-Dim1, names_to="Dim2", values_to="Acceptance (%)") %>%
  mutate(across(where(is.character), as.numeric))
```

To build the plot, different layers involving different source of data (`senso`, `senso_sup`, and `PrefMap_plot` that is) are required. Hence, the initiation of the plot through `ggplot()` does not specify any data. Instead, the data used in each step are included within the `geom_*()` of interest. In particular, `geom_tile()` (coloring) and `geom_contour()` (contour lines) are used to build the surface plot.

```r
ggplot()+
  geom_tile(data=PrefMap_plot, aes(x=Dim1, y=Dim2, fill=`Acceptance (%)`, color=`Accept
  geom_contour(data=PrefMap_plot, aes(x=Dim1, y=Dim2, z=`Acceptance (%)`), breaks=seq(
  geom_hline(yintercept=0, lty=2)+
  geom_vline(xintercept=0, lty=2)+
  geom_point(data=senso, aes(x=Dim.1, y=Dim.2), pch=20, cex=3)+
  geom_text_repel(data=senso, aes(x=Dim.1, y=Dim.2, label=Product))+
```

```
geom_point(data=senso_sup, aes(x=Dim.1, y=Dim.2), pch=20, col="green", cex=3)+
geom_text_repel(data=senso_sup, aes(x=Dim.1, y=Dim.2, label=Product), col="green")+
scale_fill_gradient2(low="blue", mid="white", high="red", midpoint=50)+
scale_color_gradient2(low="blue", mid="white", high="red", midpoint=50)+
xlab(str_c("Dimension 1(",round(senso_pca$eig[1,2],1),"%)"))+
ylab(str_c("Dimension 2(",round(senso_pca$eig[2,2],1),"%)"))+
ggtitle("External Preference Mapping applied to the biscuits data","(The PrefMap is based on th
theme_bw()
```

As can be seen, POpt is quite far from the optimal area suggested by the
PrefMap. This suggests that prototypes with higher success chances could be
developed.

Chapter 11

Value Delivery

11.1   Communicate

Sensory and consumer scientists often act as consultant whether it be for their own company or for customers. Being able to communicate effectively is perhaps one of the most important skills they should master. Communication is a simple act of transferring information and although undervalued by many, plays a key role in any business's success. Let's start this chapter reminding that there are different ways to communicate, and this process usually includes a combination of two or more types of languages, which are:

- Vocal – the language produced by articulate sounds. It is the language used in client's meetings and presentations for instance.

- Non-verbally – is related to the body language, gestures, and the tone and pitch of voice.

- Written – is the representation of a spoken language in a writing system, like proposals, technical documents, or final reports.

- Visual – is the communication using visual elements, such as the visual quality of presentations or other written documents, including formatting, logo, colors, figures, plots, etc.

It is through effective communication that you will bring potential clients' attention and interest in your company and the services you provide, that will make you truly understand your client's needs, gain their trust and provide the right solutions that will ultimately bring to a long-term partnership. Efficacious communication will be responsible for keeping a friendly relationship and your client's commitment throughout the project development and will also help

you to properly convey the outcomes of a project in a way that surpass and at the very least meet the expectations of your client and opens possibilities for follow-up engagements and/or recommendations.

The ability to communicate accurately, clearly and as intended, is definitely something that consultants should not overlook because although it seems straightforward, it involves a number of skills that may take several years of practice to master. You will find plenty of materials on the Internet and books to help you to understand better and develop your very basic skills for effective communication. We will not focus on that in this chapter, but it is worth highlighting some important aptitudes for vocal communication, which may configure one of the most powerful types of communication with your client:

- Confidence. Being confident makes you be to be seen as an expert on the topic and as you are under the control of the situation. The audience will be more likely to trust, believe, be connected, and give credit to a confident person.

- Passion and Enthusiasm. You must be passionate about what you do and convinced and enthusiastic about the solution you provide. The audience can easily capture that on your vocally and non-verbally language and will be much more interested if they can see and feel your passion.

- Ability to be succinct. No matter how interesting you feel about a topic, you must know that the audience will lose interest after some time, especially if there is a lot of technical and detailed information. Be aware that the attention span of your audience isn't long, so use your time wisely during the presentation, keeping it short and at the point.

- Feeling. This is a skill that definitely one needs time to master, but it is crucial that you pick what is going on with your client, if they seem to be understanding and following you, or if they seem to be confused or not sure about what you are talking about.

In this chapter, we will be focused on four topics that we believe any successful consultant should have in mind which are: Exploratory, Explanatory and Predictive Analysis; Audience Awareness; Method to Communicate; and Storytelling.

11.1.1   Exploratory, Explanatory and Predictive Analysis

As a consultant in the field, you will likely be in a position where you get data from your client and will review it, do the analysis, and ultimately, convey the results. And here is where it is important to make a clear distinction between exploratory and explanatory analysis. Exploratory analysis in the stage you will dig into the data, to understand it, figure out patterns and things that

may be interesting or important to highlight. The explanatory analysis is the ability, from the learnings from the previous step, to select and/or reorganize your data or remake your tables, plots, or charts in a way you can easily convey the message to your audience and make them straightforwardly understand and focus on the things is worthy. Let's discuss that a little bit more.

The hard work starts once you get the data. This is the time you will likely analyze it in multiple ways, make several plots, and look at the data from multiple angles. This is what we call, exploratory analysis! After understanding all the analysis, it may be tempting to show the audience everything, all the steps, decisions, different plots, and approaches you have taken, but do not do that. You don't want to overload your audience making them go through the same tough path you went. Instead of showing your handwork, the robustness of your analysis, and building up your credibility, you will make your audience confused, bored, and lacking interest.

Once you have done all the hard work on data analysis, it is the moment to take some time to stand back and look at the key findings and the message(s) you want to convey. It is important to keep in mind that there is always a balance to find between presenting quantified, accurate, and credible information (i.e. with sufficient details) and presenting information that makes sense, is relevant and that is easily readable and understandable. This challenging phase is what we call explanatory analysis! This is the moment you need to use your ability to translate an extensive, detailed, and complex version of your data analysis to a more concise/holistic version, to a version that will easily and clearly convey the message and highlight the main points. Keep in mind that the explanatory analysis has to be tailored according to your audience (as discussed in the next topic), which means that the way you will present the data analysis and the level of details provided will vary if you are presenting it to a group of experts in the field, including statisticians and mathematicians or in a lecture for a very diverse audience, as in a conference for instance. You need to find the right balance!

Some examples in the field to exemplify the two extremes (too complicated or too simple):

- Factorial maps – The overuse of factorial maps is a common practice in the sensory and consumer science field. It's a great tool to explore data, to make or confirm hypotheses, but maybe not the best to communicate since not so many people can correctly read and interpret them. Therefore, a good approach would be to initially work with the factorial map to interpret and draw conclusions, but then, find another way through tables or alternative charts, that may be simpler and easier, to communicate the findings with your audience.

- Spider Plots – This is the other extreme when consultants can fail but not because they present a very complex and extensive analysis, but because they decide to show the data in such an easy way that puts at

risk important information that should be captured. The use of spider plots is still a common practice that many people can easily understand, but the problem is that this analysis is so simple that can mask sensory complexity.

It is worth noticing that there is a third type of analysis in the data science field, predictive analysis. This is a hot topic in the area that involves techniques such as data modeling, machine learning, AI, and deep learning. Instead of being focused on exploring or explaining the data, predictive analysis is concerned with making successful predictions, in being sure the predictions are accurate. Examples of this approach include face recognition and text-speech. Eventually, some models can be studied to provide insights, but this is not always the case.

11.1.2   Audience Awareness

One of the most important things about being a successful consultant is Audience Awareness! No matter how good you, your team, and the service or product your company offers if you fail to communicate with your target audience. Knowing the audience, the target people you will be, for instance, presenting the outcomes of a project or sending a final report, is just the cornerstone of any successful business. Knowing your audience makes you be better able to connect to them.

In order to know your audience, you must gather some information about them beforehand, such as:

Background. Do they have a sensory science background? Statistical background? Do they have experience in data science, including R language? Do they have experience in automated reporting dashboards, machine learning, etc? If so, are they juniors, specialists, seniors?

- Vocabulary. Will your audience understand very technical terms, or do you need to use simplified terms to convey the same message? This topic is closely related to the audience's background.

- Expectations. What is your audience expecting in a presentation or final report? A short summary of the project's outcomes? A detailed explanation of the statistical analysis including appendixes with further details? Recommendations for follow-up projects? Interpretation and conclusion of the analysis?

- Role. What is the role your audience plays in the project? Are they the decisions makers? Are they the final users of a dashboard, for instance?

In general, according to the profile, the audience will likely fall into one of the three categories: Technical Audience, Management (Decisions Makers), and

General Interest. There is no magical formula on how to deal exactly with each of these types of audience, but in general, based on our experience, we must highlight that one of the key differences are in the focus, language, level of technical and detailed information you will provide for each of those target public. In general, it tends to be necessary a higher level of details and technical information and a lower level of the big picture once you move from your general interest audience to the management, and further down to the technical audience (Figure 11.1). We will further discuss the main differences between each audience above.

11.1.2.1 Technical Audience

The technical audience refers to the ones who will likely have a significant background and experience in or related to the field you are providing consulting service, which means, for instance, Sensory Scientists, Statisticians, Data Analysts, Data Architecture, or your client's IT group. They are the team that will likely be working closely with you throughout the project development, at different stages. This type of audience is usually more exigent and/or engaged and because of its expertise in the field, will likely be expecting a presentation, report, or any other technical document in a higher level of details and with a more technical vocabulary, otherwise, you will sound that you are not an expert in the topic. This audience usually needs a lower sense of the big picture of the project, which means, that they are less interested in the details like the timelines, main outcomes, etc. But be aware, it is very important to be able to distinguish your technical audience. For instance, you should not use sensory technical language to talk to the IT team and vice versa.

11.1.2.2 Management

Although a person in a management position, e.g. Sensory Manager or Director and Principal/Senior Scientist, will likely have a broad experience and background in the field, maybe even higher than your technical audience, they tend to be more interested in the whole picture, which means timelines, progress of the project, potential issues, outcomes, applicability, next steps, etc. A person in a management position has many other projects and roles in a company and will not have time to be involved in the details, instead, they will likely designate a team, which will be your technical audience, to be closely involved. In this case, you should be more concise in a meeting, presentation, or report, for instance. It is advised to keep a certain level of technical language, but it is better to present things in a simpler way and in a lower level of detail than you would do for the technical audience. Additionally, the focus should be different, since as we mentioned, this audience is likely to be more interested in the whole picture instead of the specifics of the project.
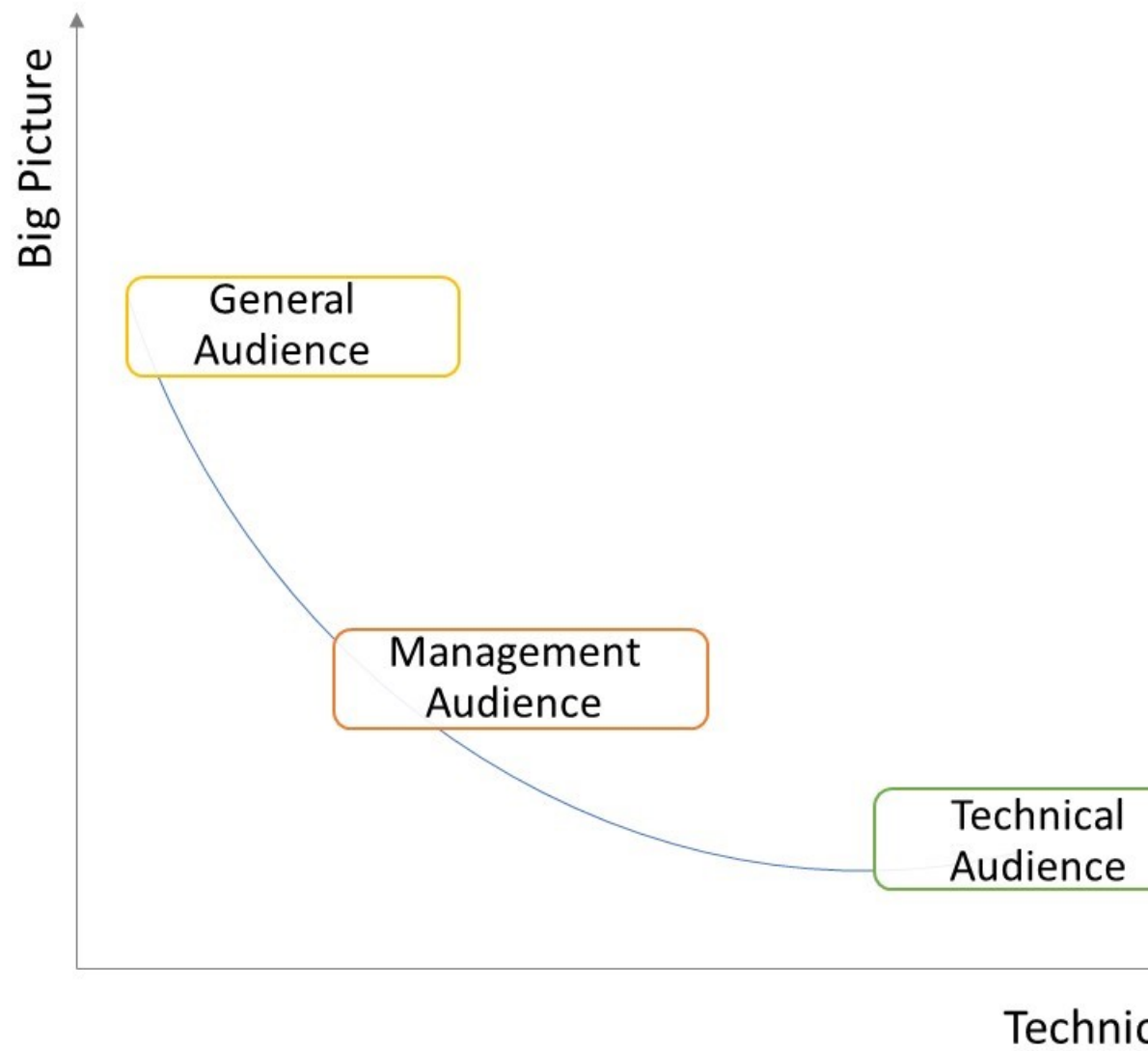
Figure 11.1: Trade-off curve based on the level of technical details and the big picture for each type of audience.

Another distinct type of audience that falls into the management audience would be the executives, as a VP of Research & Regulatory for instance. This public is not necessarily from the field and has even less time and/or background to absorb the specifics. The focus should be the same (whole picture) but with even fewer technical details. The approach and language of this audience tend to be closer to the general interest.

11.1.2.3 General Interest

The general audience usually refers to the ones that will likely be the final users or are somehow related, contributed, or are interested in the project. In this way, this public is usually the least interested in the details and the most interested in the whole picture. The general audience usually refers to a larger group of people with different backgrounds and distinct levels of expertise, for instance, an R&D internship, a Chemistry Researcher, and a Senior Sensory Specialist can be all final users of a dashboard you developed. In this case, to make sure everyone follows you in a training, for instance, you must use less technical language and a lower level of details, otherwise, you will lose part of your audience's attention. But at the same time, you may need to consider covering things that sound obvious to you, you ought to be careful about not skipping topics assuming that everyone knows about that or using certain terms and expressions considering that is evident for all. This can be the most challenging audience to deal with due to its diversity, but in a meeting, training session, or presentation, you should be very attentive and use your feeling to capture what going on and maybe change your position be better to connect with the audience

A valuable tip shared by Cole Nussbaumer, in her book Storytelling with data, is to avoid general audiences, such as the technical and management team at the same time, or general audience such as "anyone related to the field that might be interested in the project". Having a broad audience will put you in a position where you can't communicate effectively to any of them as you would be if the audience was narrowed down.

- Example

We will use the PCA biplot from the biscuit sensory study shown in the previous chapter and point out the main differences in the approach according to the audience. As a quick reminder, 11 breakfast biscuits with varying contents of proteins and fibers were evaluated in this study. Products P01 to P09 are prototypes, product P10 is a standard commercial biscuit without enrichment and the eleventh product (Popt) is an additionally optimized biscuit.

Let's picture the situation is that the R&D team has been developing multiple trials for the biscuit formulation, changing the concentration/ratio of protein and fiber, with the objective to have a product with a sensory profile as close as possible to the commercial biscuit. For this exercise, your role as a consultant

was to support the R&D team designing the study and conducting the analysis and ultimately analyzing ad interpreting the results to make the final conclusion (Figure 11.2).
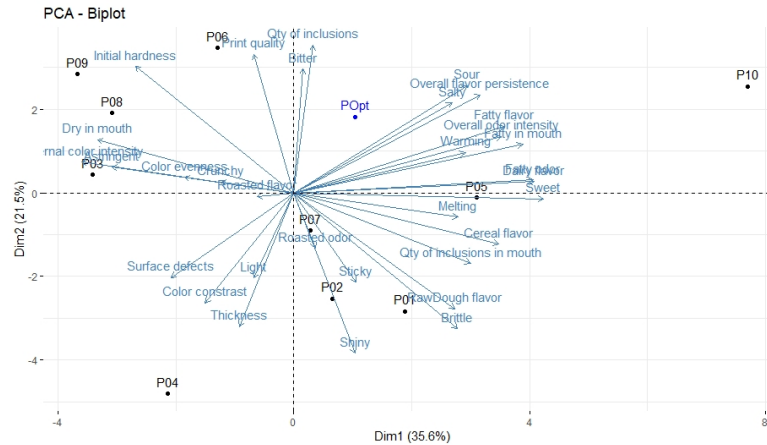

Figure 11.2: PCA Biplot Biscuit Study.

We won't go deep into the interpretation since it's not the focus of this example, but rather point out the approach we would recommend for each type of audience as shown in the table below.

Following those recommendations, your PCA for the management or general audience may look like the Figure 11.4. Note that the PCA was cleaned and simplified for a more straightforward understanding. Specifically, the PC variance explanation and grid line were removed, attributes were slightly moved to avoid overlap, the samples, and attributes that we want less emphasize were given a lighter color and as the opposite, the attributes and samples that we may want to highlight and focus our audience attention were given a different and stronger color, pictures and a more appealing description were used instead of the samples codes and some strategies as to circle the important area/group of samples and attributes helps the audience to focus on what we deem most important to extract from this analysis. In this example, the idea may be to highlight to the audience that the optimized formulation is in fact closer to the commercial one, and to increase even more this similarity, some attributes, like sour, salty, overall flavor persistence, fatty flavor, and fatty in mouth, has to be increased.

11.1.3   Method to Communicate

How will you communicate to your audience? Are you going to deliver a live presentation? Are you going to present a proposal in a live meeting? Or will the communication be a written document you will send through email?

Topics	Technical Audience	Management Audience (Management / Executive)	General Audience
Context/Background Information	The objective of the study, including general and specific objectives Information on the design of the experiment How samples were selected or screened for the study, including a detailed description of each sample Details about the panel such as number of panelists, level of training, and how panelists were trained for the attributes evaluated Details about the test, how attributes were defined, and how and where samples were evaluated	The level of information is somewhere between the Technical Audience and General Audience. In the case of the management audience, it is recommended some level of details you deem important and the use of technical language. Also, be prepared to show or discuss a deeper level of information as requested In the case of an executive audience, the level of information tends to be closer to the general audience	An overall and succinct description of the general objective of the study, samples tested, and type of test performed
Data Exploration	It may be important to show/mention the panel performance evaluation It is important to show the PCA diagnostics, including the Scree Plot and the plots displaying the contribution of each attribute in each dimension		It is not recommended to share the data exploration and analysis done before the PCA. The general audience does not need to understand details such as why two components were chosen for the PCA
Final Results (PCA)	The PCA can be shown in its traditional format without much editing as shown in the picture above Provide a detailed interpretation, including not only the optimized sample, but all trials evaluated		Simple and clear explanation about PCA Remove technical details such as the PC variance explanation Show the PCA in a format that makes the interpretation and visualization easier and more obvious, some examples include: - Use pictures instead of sample code - Use visual resources like circles to delimit groups of similar samples and attributes more relevant - Use different colors, letter sizes, or other visual resources to highlight important information, like an attribute or a sample
Conclusion and Recommendations	The conclusion can be more detailed, and recommendations can be done for further studies		Very clear and succinct conclusion or recommendation. Example: The optimized formulation is close to the commercial but needs to be improved for some attribute

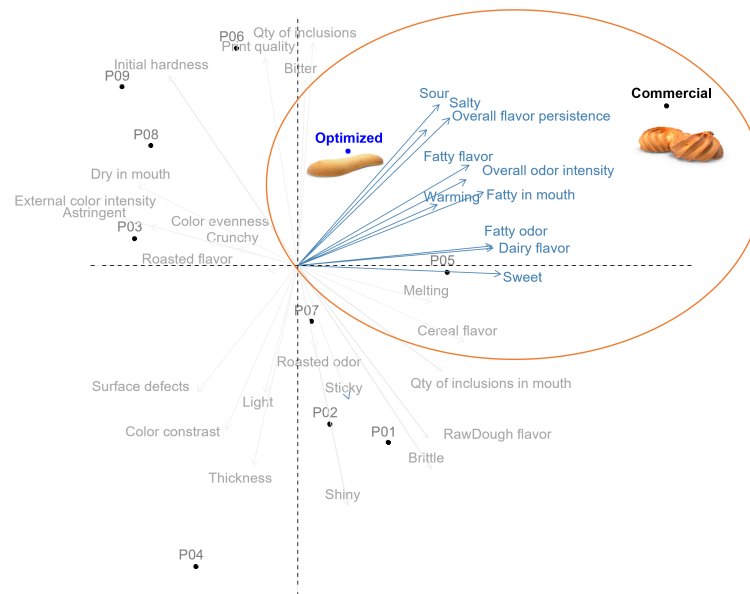Figure 11.3: Exemplification of the approach for each type of audience.

Figure 11.4: PCA Biplot Biscuit Study Modified.

What is the format you will be using to communicate? Word, Excel, or Power-Point? Are you going to send the document in PDF format? Are you going to present a dashboard? Are you going to share R Scripts?

As we will discuss in more detail above, the way and the format you use to communicate to your clients or audience have a huge impact on successful communication, and you should be well aware of that!

11.1.3.1 Consider the Mechanism

You should be aware that the primary method or mechanism you will use to communicate will strongly affect the way your audience effectively gets the information and so you should tailor it accordingly. One of the most important aspects is related to the amount of control you will have over the audience, how they get the information, and hence the level of details needed (Cole Nussbaumer).

In a live presentation, for instance, you are in full control. You can answer questions your audience may have, you can slow down and go into a particular detail you deem important, or you can speed up over repetitive, obvious, or not-so-important topics. In short words, you are the expert there and so, you can easily provide effective communication, and because of that, you don't need to overcrowd your slides or any other document and divert or lose your audience's attention with very detailed information. You can for instance just show a plot or graph and a very simple interpretation or bullet points because you will be covering the nuances and details about that.

In the case the communication will be through a written document in a non-live situation, you have much less control over your audience, on how they will take the information, on whether they will get the main point. In this situation, you will need to be more careful and likely provide a higher level of details to answer or clarify potential questions or doubts your audience may have. In this situation, showing a plot or graph and just a very simple interpretation or bullet points will likely not be enough.

It can be a great idea to merge those two formats, when possible, where you can give time to the audience to consume the information on their own for a while and give the topic thought and a moment where you can discuss it in a live situation, not in this order necessarily. So, for example, let's pretend you have to present a proposal for a client. Instead of sending a dense document to explain all the details and just wait for the client's response, you can make a more concise document, easier to go through if you have a live moment with the client. You can for instance present the proposal initially in a live meeting, where you cover in general all the important topics and details, and then send the written document to the client.

11.1.3.2   Pick the Correct Format

The second point of this topic on the method of communication is related to the correct format to pick. There are certainly many ways for you to communicate with your client - word, excel, or PowerPoint whether in pdf format or not, dashboard, or even scripts – but surely one is the most suitable. Again, there is no universal answer for the best format to pick since it may vary according to clients' requests and the type of project you are dealing with. But there is one thing you should always follow, unless strictly necessary, do not share documents in an editable format. You may use Word to write proposals or final reports, Excel for plots or tables, and PowerPoint for live presentations, and that's totally fine, but never share that in the editable format. We always recommend saving in pdf format to share with your client or audience and this is because of two simple reasons. First, the pdf format cannot be modified! You definitely don't want to take the risks of others changing your document which can lead to misunderstanding, putting you in a delicate situation. Second, the pdf format preserves document formatting which means that it retains the intended format if the file is viewed online or printed. In short words, it is very unprofessional to share documents in editable format.

You may be wondering, so in what type of situation would you share an editable format? When would you share a document in Word, PowerPoint, or Excel? In the situation where you are working with a partner for instance. So, for example, a project that you are working on involves multiple partners and one unique report or presentation. In this case, it may be convenient to share the Word or PowerPoint document for each of the partners to include their inputs. After the document is ready, make sure you carefully review the formatting and save it in pdf before sending it out to the client.

There are two other formats that may be common in the sensory data science field, which are dashboards and scripts. If you are developing a graphical user interface for your client, you will need to deploy the dashboard at some point to a server for your client to be able to access it. The deployment can be done in two ways: web-based, as a simple client web page, or locally, as a locally installed desktop application. The choice should be based on the client's preference.

The last method of communication that is fairly common in the field is R or any other programming language script. It is very common that the client requests the scripts used for a specific project, the text file containing the set of commands and comments you used for instance to develop an automated analysis reporting dashboard. You can share the repository where the scripts are hosted, or you can zip the scripts and share them with your clients. The details should be discussed with the client's IT team since each company has a particular preference. As the scripts should always be available under the client's requests, you should be careful to not display sensitive or confidential information by reusing codes or throughout the comments.

11.1.4 Storytelling

There are basically two ways to communicate with our audience, the first is called conventional rhetoric. A PowerPoint full of facts, filled with bullet points and statistics with a presenter with a formal and memorized speech and using the same voice tone, would be the best way to illustrate the conventional rhetoric style. This way to communicate, which drove the businesses of the past, has a more analytical approach, where statistics, charts, metrics would be dumped on the audience and left to them to digest. There is no need to say that this approach is completely outdated, it clearly fails to stimulate the audience's attention or evoke their energy or emotions.

The second way to communicate, which is the last topic we want to cover in this chapter and also happens to be a critical skill for any successful consultant, is through storytelling! Storytelling is something that we all know, from an early age we were introduced to the notion of narrative structure, which means a clear beginning, middle, and end. The ability of one going throughout this structure to tell us a story is what makes a book, play, or movie grab our attention and evoke our emotional responses, is what makes it interesting! In short words, storytelling is one of the most powerful and effective ways to attract people's attention because we were taught to communicate with stories throughout history. This universal language that everyone can understand has the power to truly engage your audience because it translates abstract facts, numbers, and plots into compelling pictures; it inspires, motivates, and drive actions because it taps into people's emotions.

As described in the book Once Upon an Innovation, by Jean Storlie and Mimi Sherlock, the left side of our brain is linked to more logical and analytical thinking, including data processing, number handling, and statistical interpretations. The right side is linked to expression, emotional intelligence, and imagination, and in our context, will be the part of our brain that will capture the big picture, that will turn data and facts into possibilities and innovative ideas. If you as a consultant overwhelm your audience with analytical information you will reduce their capacity for big picture thinking, you will shut down their capability to generate novel ideas and solutions. We are not saying that numbers, plots, and facts are not important, but that they should be presented in a story narrative format, in a way that will be able to light up the right side of the brain, and this stimulation of both paths is what will trigger unexpected and novel solutions, inspire support and drive changes.

In a real situation, you as a consultant will have many pieces of information that you will collect throughout the journey with a client, from the very first communication until the end of a project. You will have valuable information about your client company's situation, challenges and issues, needs, expectations, potential solutions and/or failed attempts, and final outcomes. Storytelling is the master of tying all together and articulating it into the context of a story in a creative way to engage and persuade your audience. A good story will allow you

to successfully connect with your audience, it will make your audience understand, reflect, and act in a way that plots, numbers, and facts altogether simply can't.

You may be wondering. How exactly should I construct a story? What should be covered in each piece of the narrative? We will provide here a summary of the pieces of a good story and the specifics based on our experience and also on the books Storytelling with Data by Cole Nussbaumer and Beyond Bullet Points by Cliff Atkinson. Both books dedicate a good part to Storytelling making them a great resource on this topic.

11.1.4.1   The beginning (Context)

The key piece of any story is the context, the description of the situation, and surrounding details. This first step is the moment to set up the essential information or background on the topic you will be covering to get everyone on common ground. You should initially spend time to make sure your audience clearly understands the context, why this is important or necessary, and why they are there before diving into actions or results. Subsequently, you will raise the challenges or problems and also the recommended solutions. It is at this very first step that you will first grab your audience's attention. If you fail at this moment, it is very unlike that you will recover their interest in the subsequent steps.

For live presentations, it is strongly recommended to use the first few minutes to be an icebreaker to make everyone feel more comfortable and create a more friendly environment. In order to do so, you can start introducing yourself in case you haven't met everyone yet, you can have a conversation and talk about the latest news, ask about how they and their families are doing, etc. The second piece of advice is to start the presentation by stating bullets of the main points that will be covered, so your audience will have awareness of what you will be talking about.

11.1.4.2   The middle (Action and Impact)

Now is when you get to the crux of your story, it is at this moment that you will explain your solutions or actions and highlight the impacts. You will continue it in a way you will convince your audience of the solution you are proposing or make them clearly understand, agree, and be excited about the outcomes and possibilities of a solution you worked on. You should be careful to retain your audience's attention addressing how they can be part and/or benefited from the solution you are referring to. In the case of a live presentation, pose always confidently, show enthusiasm about what you are talking about, and watch out for hidden clues, try to constantly catch your audience's response/feedback through their expressions and body language.

The content to build out your story at this moment is very dependent on the context of the situation, but from a consultant perspective, it will be likely the moment you will further develop the situation or problem covering relevant information, show some data to illustrate the situation, discuss potential solutions to address a particular topic or present the outcomes of your project.

11.1.4.3 The end (Conclusion)

This is the moment you close your story; it is when you should tie it back to the beginning to somehow recap the problem, highlight the basic idea and conclude the story. You should finish your presentation in a very impactful way, re-emphasizing and repeating your main point, what you want to stick deeply in your audience's mind. Once more, the content at the end of the story can be somehow dependent on the context of the situation, but in a consulting world, it would likely include a conclusion of the topic and also next steps and further recommendations.

11.2 Reformulate

Something important to keep in mind is the follow-up process after a report is sent or a presentation is delivered. The ability to receive feedback and reformulate is undoubtedly a very important and sometimes challenging skill that consultants seeking success should be aware of. It may be challenging since some consultants can be reluctant to feedback because of a misconception that they are the expert in the field and hence, their approach is the best. So, one of the most important rules, regardless of the expertise and knowledge you have in the field, is to be humble! Consultants must understand the idea that: 1) you need to make your client pleased unless you have a strong reason not to do so like ethical reasons or statistical rules and 2) your point of view can be biased over time and your client's request may indeed improve the clarity of an outcome for instance. Or, you can simply be wrong, miss something and have taken not the best approach. It happens! So, be open to feedback and be prepared to reformulate!

Sometimes the client feedback is something very minor, to adjust the scale of a plot, match the color with the company's palette or change the type of plot. In other cases, the feedback will demand a bit more time. It is common that the way you deemed best to present the outcomes is not that clear from your client's view or the set of data or plots you selected did not convey the message you were expecting or in an extreme situation, your client does not agree or ask you to redo an experiment or procedure. In this case, you will need to dedicate more time to address your client's request.

Regardless of the situation, you should motivate and be open mind to your client's feedback and afterward carefully work on that to tackle it all at once.

You definitely want to avoid a situation where your report or presentation be back and forth with your client. It is recommended that you make all possible changes and prepare a convincing explanation for the things that you strongly do not agree with or have a solid reason not to do so. Ideally, you should get back to your client as soon as possible highlighting the changes that were made and explaining the ones not addressed.

Haute Cuisine

Chapter 12

Machine Learning

Machine Learning is recently a hot topic in the sensory and consumer science field, proven to be one of the most game-changing technological advancements to support CPG companies in the development of new products, making a considerable role in speeding up and at the same time, reducing the cost or steps involved in the R&D process. In today's fast-moving and increasingly competitive corporate world, companies that are embracing, adopting and opening their minds to digital transformation and artificial intelligence, moving towards the age of automation, are not one but many steps ahead of their competitors.

Machine Learning (ML) is a branch of artificial intelligence based on the idea that systems can learn from data, that has the capability to evolve. Generally speaking, ML refers to various programming techniques that are able to process large amounts of data and extract useful information from it. It refers to a method of data analysis that build intelligent algorithms that can automatically improve through the experience gained from the data and identify patterns or make decisions with minimal human intervention, without being explicitly programmed. ML focuses on using data and algorithms to mimic the way humans learn, gradually improving their accuracy.

The definition of the objectives or the situation where ML would bring value refers to the very first step of the process. Once that is clear, the next step is to collect data or dig into the historical data sets to understand what information is available and/or has to be obtained. The data varies according to the situation, but it may refers to product composition or formulation, analytical measurements (E.g., pH, color, rheology, instrumental texture, GCMS, etc), sensory attributes (E.g., creaminess, sweetness, bitterness, texture, consistency, etc), or consumer behavior (E.g., frequency of consumption/use, consumption/use situation, etc.), demographics (E.g., age, gender, skin type, etc) and responses (E.g. liking, CATA questions, JAR questions, etc).

The data set size and quality are very important and will directly impact the

model's robustness. There will be specific recommendations on that according to the situation, data type, and objectives. In general, the higher the number of objects, which is usually the number of products or samples, the better, with being 10-14 the minimum recommended. The number of measurements (analytical, sensory and/or consumer measurements) and the number of consumers evaluating the products are also very relevant to the model's quality. In the Sensory and consumer science field, the number of measurements is usually very high, so not something likely to worry about. Regarding the number of consumers, the minimum number recommended would be 100, but the more the better. Regarding data quality, one of the most important aspects, besides the standardization of data collection, refers to the variability of the products/samples. The higher the variability between samples, the broader the space the model will be covering. Additionally, it is strongly recommended the capture the consumers individual differences, not only demographic information, but also JAR questions and conducting the study in a sequential monadic design.

According to the purposes and goals, there are some variations on how to group or classify ML algorithms which can be divided into three prominent methods: supervised learning, unsupervised learning, and reinforcement learning. In this chapter, we will be focusing on the first two approaches.

12.1   Methods Overview

Supervised learning is the most popular type of machine learning that uses labeled data, it is a process that involves providing input data as well as correct output data to the machine learning model. The ultimate goal of the algorithm is to find a mapping function to map input variables with output variables. In supervised learning, models are initially trained using a subset of the data (training data set) and afterwards, the model is tested and validated using the other part of the data (test data set and validation data set). Once this process is done, the model can continuously improve, discovering new patterns and relationships as it trains itself on new data sets.

A very common situation where supervised learning is widely used is to predict consumer responses (E.g., Liking, Perception, Benefit) based on analytical measurements and/or sensory data. In this situation, ML models can provide insights and directions on how to improve product performance and has also the ability to predict consumer response based on analytical and/or sensory data, working as a powerful screening tool where only prototypes or products with the highest potential are moved to the next level, which may be the consumer testing. Another common situation is the use of supervised machine learning to predict product sensory profile or consumer response, based on the formulation or ingredients of a product. In this situation, the ML would be of great support for the developers, who can much easier understand and get clear directions on what has to be changed in the formulation to improve the product sensory

profile or consumer performance.

Unlike supervised learning, unsupervised learning uses unlabeled or untagged data, which means inputs where the output values are not known. In this case, users do not need to supervise the model, instead, the algorithm operates independently to finds patterns and trends, trying to learn from the data distribution the distinguishing features and associations through similarity and dissimilarity measurements. Its ability to discover hidden patterns and get similarities/differences information is what make it the ideal solution for exploratory analysis and consumer segmentation, extracting insights from the data sets.

Unsupervised machine learning is commonly used to segment consumers into groups based on their similarities related to a variety of factors such as shopping or usage behavior, attitudes, interests, liking or preferences. As consumers in the same market segment tend to respond similarly, the segmentation process is a key strategy for companies to understand and tailor effectively their products or marketing approaches for different target groups. Similarly, unsupervised models are also very used to classify group of products in homogeneous analytical, sensory and/or consumer attributes in order to identify different segments in the market.

https://www.ibm.com/cloud/learn/supervised-learning

https://www.javatpoint.com/supervised-machine-learning

https://towardsdatascience.com/11-dimensionality-reduction-techniques-you-should-know-in-2021-dcb9500d388b

https://towardsdatascience.com/dimensionality-reduction-for-machine-learning-80a46c2ebb7e

12.1.1 Unsupervised learning

In the sensory and consumer science field, unsupervised learning models are usually used for Clustering and Dimensionality Reduction.

Clustering is a technique used to discover in a high-dimensional data, groups of observations that are similar to each other and significantly different from the rest. In other words, it is a method that groups unlabeled data based on their similarities and differences in the way that objects with most similarities remains into a group and has less or no similarities with the objects of another group.As previously mentioned, a very common application is to cluster consumers and classify groups of products based on their similarities/dissimilarities. Clustering algorithms can be categorized into a few types, which are Exclusive (E.g. k-means), Hierarchical and Probabilistic clustering (E.g. Gaussian Mixture Model). The first two are the ones most used and well known in the sensory field.

Dimensionality reduction is a technique used to transform a high dimensional space into a low-dimensional space that retains some meaningful properties of the original data. It refers to the process of reducing the number of features/attributes in a data set while maintaining as much of the variation as possible. This method is a important not only as a technique for data visualization, but also as a pre processing step to reduce training time and computational resources, improve ML algorithms performance/accuracy due to less misleading data, avoid problems of overfitting, find latent variables, remove redundant features and noise, among others. There are numerous dimensionality reduction methods that can be used according to the data type and different requirements. The most common and well known dimensionality reduction methods used in the sensory and consumer science field are the ones that apply linear transformations, including Principal Components Analysis (PCA) and Factor Analysis (FA).

12.1.1.1   Clustering and Dimensionality Reduction

Let's start installing/loading the necessary packages and by preparing the dataset for the cluster analysis, with PCA being used as a pre processing method. We will be using a wine dataset, from `rattle` package, that consists of the results of a chemical analysis of wines grown in a specific area of Italy. Three types of wine are represented in the 178 samples, with the results of 13 chemical analyses recorded for each sample.

```r
library(tidyverse)
library(rattle)

# load data

wine.fl <- "http://archive.ics.uci.edu/ml/machine-learning-databases/wine/wine.data"
wine <- read.csv(wine.fl,header = F)

# Names of the variables

wine.names=c("Alcohol", "Malic acid", "Ash", "Alcalinity of ash", "Magnesium",
             "Total phenols", "Flavanoids", "Nonflavanoid phenols", "Proanthocyanins",
             "Color intensity", "Hue", "OD280/OD315 of diluted wines", "Proline")
colnames(wine)[2:14]=wine.names
colnames(wine)[1]="Wine_type"

wine_dataset <- tibble(wine)
```

Dimensionality reduction technique via PCA will be applied before clustering to allow coherent patterns to be detected more clearly. As well known, PCA retains most of the information/variance of the data set and by removing features with

low variance, a more robust clustering can be generated. In other words, this technique reveals distinct groups if data exhibits clustering and by retaining the components with the highest variance, the clusters tends to be more visible.

As the data set being used contain variables with different units/scales, PCA with scaling will be applied. We will be using the {FactoMineR} package to perform the PCA analysis and {factoextra} to easily extract and visualize the results.

```r
library (FactoMineR)
require(factoextra)

pca_results <- PCA(wine_dataset, scale.unit = TRUE)

pca_eig <- fviz_eig(pca_results, addlabels = TRUE, ylim = c(0,50))

pca_plot <- fviz_pca_biplot(pca_results, repel = TRUE,
                            label = "var",
                            col.var = "red",
                            col.ind = "black")

pca_plot

reduced_dataset <- data.frame(pca_results$ind$coord[, 1:2]) %>%
  tibble()
```

The PCA shows that there are apparently 3 distinct groups of wines.

Let's now use its results for the clustering analysis. Although agglomerative hierarchical clustering is more common in the sensory and consumer research, it was already shown in chapter 10, so for this example we decided to use a different solution - k-means clustering. K-means clustering is also a very known unsupervised machine learning algorithm for partitioning a given data set in a way that the total intra-cluster variation is minimized. Both approaches (hierarchical clustering and k-means) are fairly used, being the main different the rule of forming clustering and the requirement to pre-specify the number of cluster to be produced the main different between them. Detailed information about clustering methods and analysis can be found in the book Practical Guide to Cluster Analysis in R (REF).

The first step of k-means clustering is to estimate the optimal number of clusters (k), and this can be conveniently done using the function fviz_nbclust () from {factoextra} package. The plot represents the variance within the clusters and the bend, also called "elbow", indicates that additional clusters beyond the third have little value. In our situation, we will then classify the observations into 3 clusters.

```
fviz_nbclust(reduced_dataset, kmeans, method = "wss")
```

As the k-means algorithm starts with k randomly selected centroids, it's recommend to initially set a seed, through the use of the function `set.seed ()`, to make reproducible results. We will then perform k-means clustering with k = 3. In the k-means function, we can also choose the number of random sets, which means the number of times R will try different random starting assignments. The default is 1, but to get more stable results we will specify nstart = 20. The function `fviz_cluster`, from `{factoextra}` package, will be used for the visualization.

```
set.seed(123)

km_dw <- kmeans(reduced_dataset, 3, nstart = 20)

km_dw

fviz_cluster(
  list(data = reduced_dataset, cluster = km_dw$cluster),
  ellipse.type = "norm",
  geom = "point",
  stand = FALSE
)
```

In the resulting plot, we can easily see the observation represented by points, and the color and ellipse identifying each of the 3 clusters. We suggest you to re-run the codes to define the optimal number of clustering and the cluster analysis without dimensionality reduction. You will see how patterns were detected much more clear when this technique was applied.

12.1.2 Supervised learning

12.1.2.1 Regression

Regression methods approximate the target variable[1] with (usually linear) combination of predictor variables.

There are multiple regression algorithms varying by type of data they can handle, type of target variable and additional aspects like ability to perform dimensionality reduction. We will take a walk through the ones most relevant for sensory and consumer science.

---

[1]This is a bit of simplification. In some cases it is some transformation of combination of predictors that approximates target variable. An example of this is logistic regression.

12.1.2.1.1 Linear regression The simplest and most popular variant is linear regression in which continuous target variable is approximated as linear combination of predictors in a way that minimizes sum of squared estimates of errors (SSE). It can be for example used to predict consumer liking of a product based on it's sensory profile, but user has to keep in mind that linear regression can in some cases return value outside reasonable range of target values. This can be addressed by capping predictions to desired range. Functions in R to apply linear regression are: `lm()` and `glm()` or `parsnip::linear_reg() %>% parsnip::set_engine("lm")` when using `tidymodels` workflow.

12.1.2.1.2 Logistic regression Logistic regression is an algorithm which by use of logistic transformation allows to apply the same approach as linear regression to cases with binary target variables. It can be used in R with `glm(family = "binomial")` or `parsnip::logistic_reg() %>% parsnip::set_engine("glm")` when using `tidymodels` workflow.

12.1.2.1.3 Penalized regression It is often that the data we want to use for modeling have a lot of predictor variables, possibly with lot of high correlations. In such cases linear/logistic regression may become unstable and produce unreasonable predictions. This can be addressed by use of so called penalized regression. It is a special case where instead of minimizing pure error term, algorithm minimizes both error and regression coefficients at the same time. This leads to more stable predictions.

There are three variations of penalized regression and all of them can be accessed via function `glmnet::glmnet()` (β is set of regression coefficients and λ is a parameter to be set by user or determined from cross-validation):

- Ridge regression (L2 penalty) minimizes $SSE + \lambda \sum |\beta|^2$ and drives the coefficients to smaller values
- Lasso regression (L1 penalty) $SSE + \lambda \sum |\beta|$ and forces some of the coefficients to vanish what can be used for variable selection
- Elastic-net regression is a combination of the two previous variants $SSE + \lambda_1 \sum |\beta| + \lambda_2 \sum |\beta|^2$.

Penalized regression can be also ran in `tidymodels` workflow with or `parsnip::linear_reg() %>% parsnip::set_engine("glmnet")`.

12.1.2.1.4 MARS One limitation of all mentioned so far methods is that they assume linear relationship between predictor and target variables. Multivariate adaptive regression spline (MARS) addresses this by modeling non-linearities with piece wise linear function. This gives a nice balance between simplicity and ability to fit complex data, for example Λ-shaped once where there is a maximal point from which function decreases in both directions. In R this model can be accessed via `earth::earth()` function.

12.1.2.1.5 PLS In case of multiple target variables one can apply partial least squares (PLS) regression which, similarly to PCA looks for components that maximizes explained variance of the predictors, but at the same time also maximizes their correlation to target variables. PLS can be applied with `lm()` specifying multiple targets or in `tidymodels` workflow with `plsmod::pls() %>% parsnip::set_engine("mixOmics")`.

12.1.2.2 K-nearest neighbors

A very simple, yet useful and robust algorithm that works for both numeric and nominal target variables is K-nearest neighbors. The idea is that for every new observation we want to predict the algorithms finds K closest points in training set and use either their mean value (for numeric targets) or most frequent value (for nominal targets) as prediction. This algorithm can be used with `kknn::kknn()` function or in `tidymodels` workflow with `parsnip::nearest_neighbor() %>% parsnip::set_engine("kknn")`.

12.1.2.3 Decision trees

Decision tree models the data by splitting the training set in smaller subsets in a way that each split is done by a predictor variable so that it maximizes the difference in target variable between the subsets. One important advantage of decision trees is that they can model complex relationships and interactions between predictors. To use decision tree in R one can use `rpart::rpart()` or in `tidymodels` workflow with `parsnip::decision_tree() %>% parsnip::set_engine("rpart")`.

12.1.2.4 Black boxes

So called black boxes are class of models that have too complex structure to directly interpret relationship between predictor variables and a value predicted by the model. Their advantage usually is ability to model more complicated data than in case of interpretable models, but they have greater risk of overfitting (fitting to noise in training data). Also, lack of clear interpretation may be not acceptable in some business specific use cases. The later problem can be addressed by use of explanation algorithms that will be discussed in later part of this chapter.

12.1.2.4.1 Random forests A random forest is a set of decision trees, each one trained on random subset of observations and/or predictors. The final prediction is an average of individual trees' predictions. This way

12.1.2.4.2 SVMs

12.1.2.4.3 Neural networks

12.1.2.4.4 Computer vision

12.2 Key Topics

12.2.1 Model Validation

12.2.2 Interpretability

12.2.2.1 LIME

12.2.2.2 DALEX

12.2.2.3 IML

12.3 Common Applications

12.3.1 Predicting sensory profiles from instrumental data

12.3.2 Predicting consumer response from sensory profiles

12.3.3 Characterizing consumer clusters

12.4 Code

tidyverse and tidymodel

The tidymodels framework is a collection of packages for modeling and machine learning using tidyverse principles.(https://www.tidymodels.org/) It contains several core packages, including "rsample", "parsnip", "recipes", "workflows", "tune", "yardstick", "broom" and "dials", along with some other specialized packages. These packages will help you build a variety of models in R and check their performances.

```
# install.packages("tidymodels")
library(tidymodels)
```

We will still use the wine data set to build our supervised model using the Tidymodel packages. Let's prepare the data before we go to details for modeling.

We need to mutate the column "Wine_type" to be factors, as this column would
be the target later.

```
# Classification -------------------------------------------------------------

# Data set preparation
wine_classification_dataset <- wine_dataset %>%
  # bind_cols(tibble(Wine_type = km_dw$cluster)) %>% # Our classification label is the
  mutate(Wine_type = as.factor(Wine_type))
```

The function `rsample::initial_split()` takes the original data and saves the
information on how to make the partitions. We set the `strata` argument as
`"Wine_type"` to conduct a stratified split. This ensures our training and test
data sets would keep roughly the same proportions of all Wine types as in the
original data, despite the imbalance we noticed in our class variable. `prop` is set
to 0.7 as we are splitting data to be 70% for training and 30% for testing.

After the initial_split, we can use the training() and testing() functions to obtain
the two data frames for training and testing.

```
initial_split <- initial_split(data = wine_classification_dataset, strata = "Wine_type"

wine_train <- training(initial_split)
wine_testing <- testing(initial_split)
```

Cross-validation is an important step for checking the model quality later. To
do that, we need to use the resampling method to create a series of data sets
similar to the training/testing split before building the model. For each data
set, a subset is used for creating the model, and the other subset is used to
measure the performance. Also noted that resampling is always used with the
training set of the data defined earlier.

We use `vfold_cv` to resample the data. Here we begin with a 5-fold cross-
validation first. `strata` is set to `wind_type` to conduct stratified sampling, so
that each resample is created within the stratification variable.

```
wine_cv <- wine_train %>% vfold_cv(v = 5,strata = Wine_type)
```

Now we have a resampled `wine_cv` for further building the model.

The recipe package is part of tidymodels. It contains a rich set of data ma-
nipulation tools which can be used to preprocess data and to define roles for
each variable (e.g. outcome and predictor). To add a recipe, we simply use the
function `recipe`. `recipe` has two arguments: a formula and the data. Any
variable on the left-hand side of the tilde (~) is considered the model outcome.
In our example, we want to use machine learning model to predict the type of

the wine, therefore `Wine_type` would be the target on the left hand side of the `~`. On the right-hand side of the tilde are the predictors. One can write out all the variables, but an easier option is to use the dot (.) to indicate all other variables as predictors.

```
# Random forest model definition
model_recipe <- wine_train %>%
  recipe(Wine_type ~ .)
```

We decided to use random forest classifier for the wine data.

`rand_forest` has 3 hyper-parameters (`mtry, trees, min_n`) which can be tuned to achieve the best results.

We set all of them as `tune()` right now. Model tuning is the process to find the best values for the parameters. Most often you start with the default values, and change them along the way. Since the model is not executing when created, these parameters can be changed using the tune() function. This provides a simple placeholder for the value.

```
rf_spec <- rand_forest(
  mtry = tune(),
  trees = tune(),
  min_n = tune()) %>%
  set_mode("classification") %>%
  set_engine(engine = "ranger")
```

Let's combine the model and recipe into a single `workflow ()` object to better manage the two R objects.

```
rf_wf <- workflow() %>%
  add_recipe(model_recipe) %>%
  add_model(rf_spec)
```

We created placeholders for tuning hyper-parameters earlier. Now it's time to define the scope of the search and choose the method for searching the parameter space, in this case it is `grid_regular`.

```
params_grid <- rf_spec %>%
  parameters() %>%
  update(mtry = mtry(range = c(1, 2)),
         trees = trees(range = c(10, 200))) %>%
  grid_regular(levels = 5)
```

Now that We have defined the hyper-parameter search range, let's look for the best combination using `tune_grid()` function. We will leave the test set aside,

and use a cross-validation set for this purpose, so that the model during training certainly will not see the data on which we will be making predictions.

We can use `autoplot()` function to take a quick look at the `tuning` object.

Next, we will obtain a reliable estimate of the quality of the model. We first select the best combination of parameters using `select_best()` function. The quality of the model can be estimated with a number of metrics. Here we decided to use roc_auc (Area Under the Receiver Operating Characteristic Curve)

```r
# Best parameters searching

tuning <- tune_grid(
  rf_wf,
  resamples = wine_cv,
  grid = params_grid
)

autoplot(tuning)

params_best <- select_best(tuning, "roc_auc")
```

Now that we have found the best parameters for the model, we can train the final model on the entire training set.

```r
# Finalize model

final_model <- rf_wf %>%
  finalize_workflow(params_best) %>%
  fit(wine_train)
```

A very important part in building machine learning models is to assess the quality of the models. To do so, first we can use the model to predict on the testing data set (`wind_testing`), which the model has not seen yet.

`predict` is a generic function for putting the prediction results from models to convenient tables for further analysis. We combine the `wine_testing` data with the column of predicted Wine_type from the model by adding the `predict(final_model, wine_testing)` in the pipe.

```r
validation_data_pred <- wine_testing %>%
  bind_cols(predict(final_model, .))
```

The `validation_data_pred` is a data frame with testing data with `Wine_type` as the actual wine type, and a `.pred_class` column as the predicted Wind_type at the end. Now we can finally judge the quality of the model by comparing them.

A very pleasant method for visualize the classification results is to use a confusion matrix. Confusion matrix is a table where each row represents instances in the actual class, while each column represents the instances in a predicted class. We can see that in our wine type prediction, it's a 100% match!

```r
cm <- conf_mat(validation_data_pred, Wine_type, .pred_class)

autoplot(cm, type = "heatmap")
```

```r
library(tidyverse)

n_rows <- 100

with(set.seed(1),
     data <- tibble(
       id = sprintf("product_%d", 1:n_rows)
     ) %>%
       mutate(
         sensory_1 = runif(n(), 0, 6),
         sensory_2 = runif(n(), 4, 8),
         sensory_3 = pmax(0, pmin(10, rnorm(n(), 6, 1.5))),
         sensory_4 = runif(n(), 0, 2),
         sensory_5 = runif(n(), 0, 2),
         sensory_6 = pmax(0, pmin(10, rnorm(n(), 3, 1))),
         sensory_7 = runif(n(), 3, 10)
       ) %>%
       mutate(
         liking =
           - 0.2 * abs(sensory_1-2.5)
         + 0.6 * sensory_2
         + 0.4 * sensory_3
         + 0.2 * abs(sensory_4-1)
         - 0.3 * sensory_5
         + 0.1 * sensory_6
         + 0.2 * sensory_7
       ) %>%
       mutate(liking = pmin(9.74, rnorm(n(), 1, 0.1) * liking))
)

summary(data)

# write_rds(data, "data/regression_data.rds")
```

```r
# Tutorial content:
#   1. Split data
#   2. Define model
#   3. Tune hyperparameters
#   4. Visualize results
#   5. Explore model with DALEX and modelStudio


# Install and load libraries -------------------------------------------

# install.packages(c("tidyverse", "tidymodels", "remotes",
#                    "DALEXtra", " modelStudio", "r2d3"))
# remotes::install_github("aigorahub/aigoraOpen")
library(tidyverse)
library(tidymodels)
library(aigoraOpen)


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

# data <- read_rds("data/regression_data.rds")

set.seed(123)

data_split <- initial_split(data)

training_data <- training(data_split)
validation_data <- testing(data_split)

resampling <- vfold_cv(training_data, v = 10)

# Define model --------------------------------------------------------

model_recipe <- training_data %>%
  select(-id) %>% # id shouldn't be used by the model
  recipe(liking ~ .) %>%
  step_earth(all_predictors(), outcome = "liking")

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

```r
model_workflow <- workflow() %>%
  add_recipe(model_recipe) %>%
  add_model(model_spec)


# Tune model hyperparameters (grid search) --------------------------------

set.seed(124)

params_grid <- model_spec %>%
  parameters() %>%
  grid_regular(levels = 10)

tuning <- tune_grid(
  model_workflow,
  resamples = resampling,
  grid = params_grid
)

autoplot(tuning)

params_best <- select_best(tuning, "rmse")


# Zoom in grid search -----------------------------------------------------

set.seed(125)

penalty_levels <- sort(unique(params_grid$penalty))
mixture_levels <- sort(unique(params_grid$mixture))

penalty_new_range <- c(
  penalty_levels[lead(penalty_levels, default = 0) == params_best$penalty],
  penalty_levels[lag(penalty_levels, default = 0) == params_best$penalty]
)
mixture_new_range <- c(
  mixture_levels[lead(mixture_levels, default = 0) == params_best$mixture],
  mixture_levels[lag(mixture_levels, default = 0) == params_best$mixture]
)

params_grid_2 <- model_spec %>%
  parameters() %>%
  update(
    penalty = penalty(log10(penalty_new_range)),
    mixture = mixture(mixture_new_range)
```

```r
  ) %>%
  grid_regular(levels = 10)

tuning_2 <- tune_grid(
  model_workflow,
  resamples = resampling,
  grid = params_grid_2
)

autoplot(tuning_2)

params_best_2 <- select_best(tuning_2, "rmse")


# Finalize model ------------------------------------------------------------

final_model <- model_workflow %>%
  finalize_workflow(params_best_2) %>%
  fit(training_data)

validation_data_pred <- validation_data %>%
  bind_cols(predict(final_model, .))

metric_set(rmse, rsq)(validation_data_pred, liking, .pred)

validation_data_pred %>%
  ggplot(aes(liking, .pred)) +
  geom_point() +
  geom_abline(slope = 1, intercept = 0, linetype = 2)


# Model Studio --------------------------------------------------------------

library(DALEXtra)
library(modelStudio)

explainer <- explain_tidymodels(
  final_model,
  data = training_data %>%
    column_to_rownames("id") %>%
    select(-liking),
  y = training_data$liking
)

resid <- model_performance(explainer)
```

```r
resid
plot(resid)

var_imp <- variable_importance(explainer)
plot(var_imp)

ms <- modelStudio(
  explainer,
  new_observation = validation_data %>%
    column_to_rownames("id") %>%
    select(-liking),
  new_observation_y = round(validation_data$liking, 3)
)

# r2d3::save_d3_html(ms, "output/modelstudio.html")
```

Chapter 13

Text Analysis

Humans exchange information through the use of languages. There is of course a very large number of different languages, each of them having their own specificity. The science that studies languages per se is called *linguistics*: It focuses on areas such as phonetics, phonology, morphology, syntax, semantics, and pragmatics.

Natural Language Processing (NLP) is a sub-field of linguistics, computer science, and artificial intelligence. It connects computers to human language by processing, analyzing, and modeling large amounts of natural language data. One of the main goals of NLP is to "understand" the contents of documents, and to extract accurately information and insights from those documents.

In Sensory and Consumer Research, *Text Analysis* usually refers to NLP.

Since the fields of linguistics and NLP are widely studied, a lot of documentations is already available [REF REF REF?]. The objective of this chapter is to provide sufficient information for you to be familiar with textual data, and to give you the keys to run the most useful analyses in Sensory and Consumer Research. For those who would like to dive deeper into NLP, we recommend reading (Silge and Robinson (2017), Bécue-Bertaut (2019)), and (Hvitfeldt and Silge (2021)) for more advanced techniques.

13.1 Application of Text Analysis in Sensory and Consumer Science

13.1.1 Text analysis as way to describe products

In recent years, open-ended comments have gained interest as it is the fastest, safest, most unbiased way to collect spontaneous data from participants

(Piqueras-fiszman (2015)).

Traditionally, most SCS questionnaires relied primarily on closed questions, to which open-ended questions were added to uncover the consumers' reasons for liking or disliking products. In practice, these open-ended questions were positioned right after liking questions, and aimed at providing some understanding about why a product may or may not be liked, and to give the participants a chance to reduce their frustration by explaining their responses to certain questions. As a result of such practices, these questions were usually not deeply analyzed.

With the development of the so-called *rapid* and consumer-oriented descriptive methods, the benefits of open-ended questions became more apparent as they provide a new way to uncover sensory perception. In practice, respondents are asked to give any terms that describe their sensory perception in addition to their quantitative evaluation of the products by the means of intensity rating or ranking (e.g. Free Choice Profile, Flash Profile), or similarities and dissimilarities assessment (e.g. Free Sorting Task, and Ultra Flash Profile as an extension of Napping). Since the textual responses are now an integral part of the method, its analysis can no longer be ignored.

The importance of open-ended questions increased further as it has been shown that respondents can reliably describe in their own words their full experience (perception, emotion, or any other sort of association) with products. Recently, Mahieu et al. [REF REF REF] showed the benefits of using open-ended questions over CATA[1]. In this study, consumers were asked to describe with their own words both the products they evaluated and what their ideal product would be like. Similarly, Luc et al. [REF REF REF] proposed an alternative to Just About Right (JAR) scale method - called free-JAR - and in which consumers describe the samples using their own words, by still following a JAR terminology (too little, too much, or JAR, etc.).

The inclusion of open-ended questions as one of the primary elements of sensory and consumer tasks blurs the line with other fields, including psychology and sociology where these qualitative methods originated. More recently, advances in the technology (web-scraping, social listening, etc.) opened new doors that brought SCS closer to other fields such as marketing for instance. The amount of data that are collected with such techniques can be considerably larger, but the aim of the analysis stays the same: extracting information from text/comments.

13.1.2   Objectives of Text Analysis

Open-ended comments, and more generally textual responses in questionnaires, are by definition qualitative. This means that the primary analysis should be

---

[1]CATA can be seen as a simplified version of open-comments in the sense that respondents also associate products to words, however they lose the freedom of using their own as they need to select them from a pre-defined list.

qualitative. It could simply consist in reading all these comments and eventually summarizing the information gathered. But as the number of comments increases, such an approach quickly becomes too time and energy consuming for the analysts. How can we transform such qualitative data into quantitative measures? How can we digest and summarize the information contained in these comments without losing the overall meaning of the messages (context)?

One easy solution is to simply count how often a certain word is being used in a given context (e.g. how often the word `sweet` is being associated to each product evaluated). However, if such a solution is a reasonable one to start with, we will show some alternatives that allow going deeper into the understanding of textual inputs. This is the objective of the textual analysis and NLP that we are going to tackle in the next sections.

13.1.3 Warnings

Languages are complex, as many aspects can influence the meaning of a message. For instance, in spoken languages, the intonation is as important as the message itself. In written languages, non-word items (e.g. punctuation, emojis) may also completely change the meaning of a sentence (e.g.irony). Worst, some words have different meanings depending on their use (e.g. *like*), and the context of the message provides its meaning. Unfortunately, the full *context* is only available when analyzed manually (e.g. when the analyst reads all the comments), meaning that automating analyses do not always allow capturing it properly. In practice however, reading all the comments is not a realistic solution. This is why we suggest to automate the analysis to extract as much information as possible, before going back to the raw text to ensure that the conclusions drawn actually match the data.

13.2 Illustration using Sorting Task Data

To illustrate this chapter, the data set that we are using was kindly shared by Dr. Jacob Lahne. It is part of a study that aimed at developing a CATA lexicon for Virginia Hard (Alcoholic) Ciders (REF REF REF.). The data can be found in *cider_data.xlsx*.

13.2.1 Data Pre-processing

Before starting, it is important to mention that there is a large variety of R packages that handle textual data [2]. Among others, we can mention the

---

[2] Readers who are interested in the Reinert textual clustering method (REF REF Reinert 1983) can also have a look at the project IRaMuTeQ (http://www.iramuteq.org/), which is a free software dedicated to text analysis and developed in R and Python.

{tm} package for text mining, {tokenizers} to transform strings into tokens, {SnowballC} for text stemming, {SpacyR} for Natural Language Processing, or {Xplortext} for deep understanding and analysis of textual data.

However, to ensure a continuity with the rest of the book, we will emphasize the use of the {stringr} package for handling strings (here text) combined with the {tidytext} package. Note that {stringr} is part of the tidyverse and thus applies the {tidyverse} philosophy to textual data.

13.2.2   Introduction to working with strings ({stringr})

As mentioned earlier, {stringr} is one of the packages included in the {tidyverse}. This package brings a large set of tools that allow working with strings. Most functions included in {stringr} start with str_*(). Here are some of the most convenient functions:

- str_length() to get the length of the string;
- str_c() to combine multiple strings into one;
- str_detect() to search for a pattern in a string, and str_which() find the position of a pattern within the string;
- str_extract() and str_extract_all() to extract the first (or all) matching pattern from a string;
- str_remove() and str_remove_all() to remove the first (or all) matching pattern from a string;
- str_replace(), str_replace_all(), to replace the first (or all) matching pattern with another one.

It also includes *formatting* options that can be applied to strings, including:

- str_to_upper() and str_to_lower() to convert strings to uppercase or lowercase;
- str_trim() and str_squish() to remove white spaces;
- str_order to order the element of a character vector.

Examples of application of some of these functions will be provided in the next sections.

13.2.3   Tokenization

The analysis of textual data starts with defining the statistical unit of interest, also known as *token*. This can either be a single word, a group of words, a sentence, a paragraph, a whole document etc. The procedure to transform the document into tokens is called *tokenization*.

Let's have a look at our data here:

```r
library(tidyverse)
library(readxl)
library(here)

file_path <- here("data","cider_data.xlsx")
cider_og <- read_xlsx(file_path) %>%
  mutate(sample = as.character(sample))
```

We notice here that for each sample evaluated, respondents are providing a set of descriptions, which can be a single word (e.g. `yeasty`) or a group of words (`like it will taste dry and acidic`). We can also note that this data set is fairly well structured since the responses are separated by a `;` or `,`.

Let's transform this text into tokens using `unnest_tokens()` from the `{tidytext}` package. The function `unnest_tokens()` proposes different options for the tokenization, including by "words", "ngrams", or "sentences" for instance. However, in our example we can build on the data set structure and use a specific character to separate the tokens (here `;`, `,` etc.). For this, we are using "regex" that allows us to specify the different patterns.

```r
(cider <- cider_og %>%
  unnest_tokens(tokens, comments, token="regex", pattern="[;|,|:|.|/]", to_lower=FALSE))
```

This procedure already provides some interesting information as we can readily count word usage and answer questions such as "how often the word *apple* is used to describe each samples?" for instance. However, a deeper look at the data shows some inconsistencies since some words starts with a space, or have capital letters (remember that R is case-sensitive!). Further pre-processing is thus needed.

13.2.4 Simple Transformation (lowercase)

To improve the analysis, let's standardize the text by removing all the white spaces (*irrelevant* spaces in the text, e.g. at the start/end, double spaces, etc.), transforming everything to lower case (note that this could have been done earlier through `unnest_tokens()`), removing some special letters, replacing some misplaced characters etc.[3]

```r
(cider <- cider %>%
  mutate(tokens = str_to_lower(tokens)) %>%
  mutate(tokens = str_trim(tokens)) %>%
```

---

[3]This process is done in iterations: the more you clean your document, the more you find some small things to fix…until you're set!

```
  mutate(tokens = str_squish(tokens)) %>%
  mutate(tokens = str_remove_all(tokens, pattern="[(|)|?|!]")) %>%
  mutate(tokens = str_remove_all(tokens, pattern="[ó|ò]")) %>%
  mutate(tokens = str_replace_all(tokens, pattern="õ", replacement="'")))
```

Let's produce the list of tokens generated here (and its corresponding frequency):

```
cider %>%
  count(tokens) %>%
  arrange(desc(n)) %>%
  View()
```

At first glance, the most used words to describe the ciders are `sweet` (55 occurrences), `fruity` (33 occurrences), and `sour` (32 occurrences). However, a closer look at this list highlights a few things that still need to get tackled:

- The same concept can be described in different ways: `spicy`, `spices`, and `spiced` may all refer to the same concept, yet they are written differently and hence are considered as different tokens. We will see how we can handle this in a later stage.
- Multiple concepts are still joined (and hence considered separately: `sour and sweet` is currently neither associated to `sour`, nor to `sweet`, and we may want to merge them.
- There could be some typos: Is `sweat` a typo and should read `sweet`? Or did that respondent really perceived the cider as `sweat`?
- Although most tokens are made of one (or few) words, some others are defined as a whole sentence (e.g. `this has a very lovely floral and fruity smell`)

13.2.5   Stopwords

Stop words refer to *common* words that do not carry much (if at all) information. In general, stop words include words (in English) such as *I*, *you*, *or*, *of*, *and*, *is*, *has*, etc. It is thus common practice to remove such stop words before any analysis as they would *pollute* the results with unnecessary information.

Building lists of stop words can be tedious. Fortunately, it is possible to find some pre-defined lists, and to eventually adjust them to our own needs by adding and/or removing words. In particular, the package `{stopwords}` contains a comprehensive collection of stop word lists:

```
library(stopwords)
length(stopwords(source="snowball"))
length(stopwords(source="stopwords-iso"))
```

Here, we can see that English Snowball list contains 175 terms, whereas the English list from the Stopwords ISO collection contains 1298 words.

A deeper look at these lists (and particularly to the Stopwords ISO list) shows that certain words including *like*, *not* and *don't* (just to name a few) are considered as stop words. If we would use this list blindly, we would remove these words from our comments.

Although using such list on our current example would have a limited impact on the analysis (most comments are just few descriptive words), it would have a more critical impact on other studies in which consumers give their opinion on samples. Indeed, the analysis of the two following comments *I like Sample A* and *I don't like Sample B* would be lost although they provide some relevant information.

It is therefore important to remember that although a lot of stop words are common, there are also a lot of them that are topic specific, and that should be (or should not be) used in certain contexts. Hence, inspecting and adapting these lists before use is strongly recommended.

For an even better cleaning, let's go one step further and split the remaining tokens into single words by using the space as separator. Then, we can number each token for each assessor using `row_number()` to ensure that we can still recover which words belong to the same token, as defined previously. This information will be specially relevant later when looking at *bigrams*.

```
cider <- cider %>%
  relocate(subject, .before=sample) %>%
  group_by(subject, sample) %>%
  mutate(num = row_number()) %>%
  ungroup() %>%
  unnest_tokens(tokens, tokens, token="regex", pattern=" ")

head(cider)
```

We can see here that for `J1` and `182`, the first token is now separated into three words: `hard`, `cider`, and `smell`.

It appears now that `sweet` appears 91 times, and `apple` 80 times. Interestingly, terms such as `a`, `like`, `the`, `of`, `and` etc. also appear fairly frequently.

Since we have a relatively small text size, let's use the *SnowBall Stopword* list as a start, and look at the terms that our list and this stopword list share:

```
stopword_list <- stopwords(source="snowball")
word_list <- cider %>%
  count(tokens) %>%
  pull(tokens)
```

```
intersect(stopword_list, word_list)
```

As we can see, some words such as *off*, *not*, *no*, *too*, and *very* would automatically be removed. However, such qualifiers are useful in the interpretation of sensory perception, so we would prefer to keep them. We can thus remove them from the original stopword list.

```
word_list[!word_list %in% stopword_list]
```

Conversely, we can look at the words from our data that we would not consider relevant and add them to the list. For instance, words such as *like*, *sample*, *just*, *think*, or *though* do not seem to bring any relevant information here. Hence, we propose to remove these as well:

```
stopword_list <- stopword_list[!stopword_list %in% c("off","no","not","too","very")]
stopword_list <- c(stopword_list, c("accompany","amount","anything","considering","des
                                    "just","like","neither","one","order","others","pr
                                    "sample","seems","something","thank","think","thou
                                    "-"))
```

Finally, we clean our data by removing all the words stored in `stopword_list`. This can easily be done either using `filter()` (we keep tokens that are not contained in `stopword_list`), or by using `anti_join()`[4]:

```
cider <- cider %>%
  anti_join(tibble(tokens = stopword_list), by="tokens")
```

13.2.6   Stemming and Lemmatization

After removing the stop words, we finally have a list of 334 different words. However a closer look at this list shows that it is still not optimal, as for instance `apple` (80 occurrences) and `apples` (24 occurrences) are considered as two separate words although we could argue that they refer to the same concept.

To further *clean* the data, two similar approaches can be considered: **stemming** and **lemmatization**.

The procedure of stemming consists in performing a step-by-step algorithm that reduces each word to its base word (or *stem*). The most used algorithm is the one introduced by REF (Porter, 1980) which is available in the `{SnowballC}` package through the `wordStem()` function:

---

[4]Note that if we were using the original list of stopwords, `anti_join()` can directly be associated to `get_stopwords(source="snowball")`.

```
cider <- cider %>%
  mutate(stem = wordStem(tokens))

cider %>%
  count(stem) %>%
  arrange(desc(n))
```

The stemming reduced further the list to 309 words. Now, `apple` and `apples` have been combined into `appl` (104 occurrences). However, due to the way the algorithm works, the final tokens are no longer English[5] words.

Alternatively, we can *lemmatize* words. Lemmatization is similar to stemming except that it does not cut words to their stems: Instead it uses knowledge about the language's structure to reduce words down to their dictionary form (also called *lemma*). This approach is implemented in the `{spacyr}` package[6] and the `spacy_parse()` function:

```
library(spacyr)

spacy_initialize(entity=FALSE)
lemma <- spacy_parse(cider$tokens) %>%
  as_tibble() %>%
  dplyr::select(tokens=token, lemma) %>%
  unique()

cider <- full_join(cider, lemma, by="tokens")
```

As can be seen, as opposed to stems, lemmas consist in *regular* words. Here, the grouping provides similar number of terms (309 vs. 307) yet some differences can be noticed:

```
cider %>% count(stem)
cider %>% count(lemma)
```

In the case of lemmatization, `acid`, `acidity`, and `acidic` are still considered as separate words whereas they are all grouped under `acid` with the stemming procedure. This particular example shows the advantage and disadvantage of each method, as it may (or may not) group words that are (or are not) meant to be grouped. Hence, the use of lemmatization/stemming procedures should be

---

[5]Different algorithms for different languages exist, so we are not limited to stemming English words.

[6]spaCy is a library written in Python: for the `{spacyr}` package to work, you'll need to go through a series of steps that are described here: (https://cran.r-project.org/web/packages/spacyr/readme/README.html)[https://cran.r-project.org/web/packages/spacyr/readme/README.html]

thought carefully. Depending on their objective, researchers may be interested in the different meanings conveyed by such words as `acid`, `acidity`, and `acidic` and decide to keep them separated, or decide to group them for a more holistic view of the main sensory attributes that could be derived from this text.

It should also be said that neither the lemmatization nor the stemming procedure will combine words that are different but bear similar meanings. For instance, the words `moldy` and `rotten` have been used, and some researchers may decide to group them if they consider them equivalent. This type of grouping can be done case-by-case using `str_replace()`:

```
cider %>%
  count(lemma) %>%
  filter(lemma %in% c("moldy","rotten"))

cider %>%
  mutate(lemma = str_replace(lemma, "moldy", "rotten")) %>%
  count(lemma) %>%
  filter(lemma %in% c("moldy","rotten"))
```

As can be seen here, originally, `moldy` was stated twice whereas `rotten` was stated 5 times. After re-placing `moldy` by `rotten`, the newer version contains 7 occurrences of `rotten` and none of `modly`.

Doing such transformation can quickly be tedious to do directly in R. As an alternative solution, we propose to export the list of words in Excel, create a new column with the new grouping names, and merge the newly acquired names to the previous file. This is the approach we used to create the file entitled *Example of word grouping.xlsx*. In this example, one can notice that we limited the grouping to a strict minimum for most words except `bubble` that we also combined to `bubbly`, `carbonate`, `champagne`, `moscato`, `fizzy`, and `sparkle`:

```
new_list <- read_xlsx("temp/Example of word grouping.xlsx")
cider <- cider %>%
  full_join(new_list, by="lemma") %>%
  mutate(lemma = ifelse(is.na(`new name`), lemma, `new name`)) %>%
  dplyr::select(-`new name`)

cider %>% count(lemma)
```

This last *cleaning* approach reduces further the number of words to 279.

13.3 Text Analysis

Now that the text has been sufficiently cleaned, we can run some analyses and actually compare the samples in the way they have been described by the

respondents. To do so, let's start with simple analyses.

13.3.1 Raw Frequencies and Visualization

In the previous sections, we have already shown how to count the number of occurrences of each word. We can reproduce this and show the top 10 most used words to describe our ciders:

```
cider %>%
  group_by(lemma) %>%
  count() %>%
  arrange(desc(n)) %>%
  filter(n>=10, !is.na(lemma)) %>%
  ggplot(aes(x=reorder(lemma, n), y=n))+
  geom_col()+
  theme_minimal()+
  xlab("")+
  ylab("")+
  theme(axis.line = element_line(colour="grey80"))+
  coord_flip()+
  ggtitle("List of words mentioned at least 10 times")
```

As seen previously, the most mentioned words are `apple`, `sweet`, `fruity`, and `sour`.

Let's now assess the number of time each word has been used to characterize each product.

```
cider %>%
  filter(!is.na(lemma), !is.na(sample)) %>%
  group_by(sample, lemma) %>%
  count() %>%
  ungroup() %>%
  pivot_wider(names_from=lemma, values_from=n, values_fill=0)
```

A first look at the contingency table shows that `apple` has been used 24 times to characterize sample `365` while it has only been used 9 times to characterize sample `401`.

Since the list of terms is quite large, we can visualize these frequencies in different ways: First, we could re-adapt the histogram produced previously overall but per product. This could give a good overview of which words characterize each sample:

```r
prod_term <- cider %>%
  filter(!is.na(lemma), !is.na(sample)) %>%
  group_by(sample, lemma) %>%
  count() %>%
  ungroup() %>%
  split(.$sample) %>%
  map(function(data){

    data %>%
      arrange(desc(n)) %>%
      filter(n>=5) %>%
      ggplot(aes(x=reorder(lemma, n), y=n))+
      geom_col()+
      theme_minimal()+
      xlab("")+
      ylab("")+
      theme(axis.line = element_line(colour="grey80"))+
      coord_flip()+
      ggtitle(paste0("List of words mentioned at least 5 times for ",
                     data %>% pull(sample) %>% unique()))

  })
```

Another approach consists in visualizing the association between the samples and the words in a multiple way using Correspondence Analysis (CA). Since the CA can be sensitive to low frequencies (Add REF), we suggest to only keep terms that were at least mentioned 5 times across all samples, resulting in a shorter frequency table. We then use the `CA()` function from `{FactoMineR}` to build the CA map:

```r
cider_ct <- cider %>%
  filter(!is.na(lemma), !is.na(sample)) %>%
  group_by(sample, lemma) %>%
  count() %>%
  ungroup() %>%
  filter(n >= 5) %>%
  pivot_wider(names_from=lemma, values_from=n, values_fill=0) %>%
  as.data.frame() %>%
  column_to_rownames(var="sample")

library(FactoMineR)
cider_CA <- CA(cider_ct)
```

As can be seen, sample 731 is more strongly associated to alcoholic terms such as `alcohol` or `wine`, and colors (`red`, `green`). Samples 239 and 401 are more

associated to `sour` and `bitter` (and `pear` for `239`), whereas samples `519` and `182` are more frequently described by terms such as `fruity`, and `sweet` (`floral` is also used to characterize `182`).

These frequencies can also be visualized using wordclouds, which can easily be done using the `{ggwordcloud}` package that has the advantage to build such representation in a `{ggplot2}` format. Such wordclouds (here one per product) can be obtained using the following code:

```
cider_wc <- cider %>%
  filter(!is.na(lemma), !is.na(sample)) %>%
  group_by(sample, lemma) %>%
  count() %>%
  ungroup() %>%
  filter(n >= 5)

library(ggwordcloud)
ggplot(cider_wc, aes(x=sample, colour=sample, label=lemma, size=n))+
  geom_text_wordcloud(eccentricity = 2.5)+
  xlab("")+
  theme_minimal()
```

In these wordclouds, we notice that `apple` and `sweet` appear in larger fonts for (almost) all the samples, which can make the comparison quite difficult between samples. Fortunately, the `geom_text_wordcloud()` function provides an interesting parameter in its aesthetics called `angle_group` which allows controlling the position of the words. To illustrate this, let's apply the following rule: for a given sample, if the proportion of association of a word is larger than 1/6 (as we have 6 samples), the word will be printed in the upper part of its wordcloud, and in the lower part otherwise. To facilitate the readability, the color code used follow the same rule.

```
cider_wc %>%
  group_by(lemma) %>%
  mutate(prop = n/sum(n)) %>%
  ungroup() %>%
  ggplot(aes(colour= prop<1/2, label=lemma, size=n, angle_group = prop < 1/2))+
  geom_text_wordcloud(eccentricity = 2.5)+
  xlab("")+
  theme_minimal()+
  facet_wrap(~sample)
```

As can be seen, the term `apple` is more frequently (i.e. more than 1/6) used to characterize samples `182`, `239`, `365`, and `731`. The term `sweet` is more frequently used to characterize samples `182` and `519`. Such conclusions would have been more difficult to reach based on the previous *unstructured* wordcloud.

13.3.2   Bigrams, *n*-grams

In the previous set of analyses, we defined each word as a token. This procedure disconnects words from each others, hence discarding the context of the word. Although this approach is common, it can lead to misinterpretation since a product that would often be associated to (say) *not sweet* would in the end be characterized as *not* and *sweet*. A comparison of samples based on the sole word *sweet* could suggest that the previous product is often characterized as sweet whereas it should be the opposite.

To avoid this misinterpretation, two solutions exist:

1. Replace *not sweet* by *not_sweet*, so that it is considered as one token rather than two;
2. Look at groups of words, i.e. at words within their surroundings.

The latter option leads us to introduce the notion of bi-grams (groups of 2 following words), tri-grams (groups of 3 following words), or more generally *n*-grams (groups of *n* following words). More precisely, we are applying the same frequency count as before except that we are no longer considering one word as a token, but as a sequence of 2, 3, or more generally *n* words as a token. Such grouping can be obtained by the `unnest_tokens()` from `{tidytext}` in which `token='ngrams'`, with `n` defining the number of words to consider.

For simplicity, let's apply this to the original data, although it could be applied to the cleaned version (here we consider bi-grams).

```
cider_2grams <- cider_og %>%
  unnest_tokens(bigrams, comments, token="ngrams", n=2)

cider_2grams %>%
  count(bigrams) %>%
  arrange(desc(n))
```

In our example, `sweet fruity` is the strongest 2-words association. Other relevant associations are `green apple`, `sweet apple`, or `very sweet`. Of course, such bi-grams can also be obtained per product:

```
cider_2grams %>%
  group_by(sample) %>%
  count(bigrams) %>%
  ungroup() %>%
  arrange(desc(n)) %>%
  filter(sample == "182")
```

For sample 182, `not sweet` appears 3 times which can be surprising since it was one of the sample the most associated to `sweet` with 22 occurrences.

13.3.3   Word Embedding

The previous section introduces the concept of context, as words are associated to their direct neighbors. Another approach called *word embedding* goes one step further by looking at connections between words within a certain window: for instance, how often are *not* and *sweet* present together within a window of 3, 5, or 7 words? Such an approach is not presented here as it is only relevant for longer text documents.

In the previous sections, we already introduced the notion of *term frequency* (*tf*), which corresponds to the number of times a word is being used in a document. When a collection of documents are analyzed and compared, it is also interesting to look at the *inverse document frequency* (*idf*), which consists in highlighting words that discriminate between documents by reducing the weight of common words and by increasing the weight of words that are specific to certain documents only. In practice, both concepts are associated (by multiplication) to compute a term's *tf-idf*, which measures the frequency of a term adjusted for its rarity in use.

13.3.4   Sentiment Analysis

In our example, the textual data we analyze is descriptive. In other words, the items that we analyze have no particular valence (i.e. they are neither negative, nor positive). When text data are more spontaneous (e.g. social media such as tweets, or consumers' responses to open-ended questions), they can be the charged with positive or negative connotations. A good way to measure the overall valence of a message is through *Sentiment Analysis*.

To perform *Sentiment Analysis*, we start by deconstructing the message into words (tokenization approach considered previously). Then, in a similar approach to the stop words, we can combine our list of words with a pre-defined list that defines which words should be considered as positive or negative (the rest being neutral). Ultimately, all the scores associated to each message can be summed, hence providing the overall valence score of a message.

To get examples of *sentiment* list, the `get_sentiments()` function from the `{tidytext}` package can be used. This function proposes 4 potential lists: `"bing"`, `"afinn"`, `"loughran"`, and `"nrc"` (REFERENCES). Of course, such lists can be modified and adapted to your own needs in case they do not fit perfectly.

13.4   To go further...

Text Mining and Natural Language Processing is a topic that has been (and is still being) studied for a very long time. Recently, it has made a lot of progress

thanks to the advances in technology, and has gain even more interest with the abundance of text through social media, websites, blogs, etc. It is hence no surprise that a lot of machine learning models use text data (topic modelling, classification of emails to spam, etc.). Even current handy additions to simplify our life are based on text analysis (e.g. suggestions in emails, translation, etc.)

In case you would want to go further on this topic, we strongly recommend the following books:

- Text Mining with R
- Supervised Machine Learning for Text Analysis in R
- Textual Data Science with R
- R for Data Science (introduction to web-scrapping etc.)

Chapter 14

Dashboards

14.1 Objectives

We have certainly been all through situations in which we spent a lot of time analyzing data for our study, built our report and our story, spent time in perfecting our presentation. But when came the day of the presentation to your manager and colleagues, we get questions such as: What would happen if we split the results between users/non users, or between gender for instance? In case you haven't been prepared for this question, and didn't run the analysis up-front, you probably answered something like: "Let me re-run some analyses and I'll update you on that!"

Now imagine that you are in a similar situation, except that when such question arises, you have a tool that can answer live their questions using the data and the same analysis. Wouldn't that bring your discussion to another level? Even better: Imagine you can share this tool with your colleagues for them to use it and answer their own questions, or use it for their own projects, even if they are not good at coding (nor even slightly familiar with R)?

Simply said, this is one of the roles of dashboards, as it brings interactivity to results (tables, graphs) by re-running and updating them when data (e.g. adding or removing data), options, and/or parameters are being altered. In this section, we will show you how to build such dashboard using R and the `{shiny}` package.

14.2   Introduction to Shiny through an Example

14.2.1   What is a Shiny application?

In case you have already been through the visualization chapter, you've already been briefly introduced to some sort of dashboard in R through the `{esquisse}` package. In the next section, we will show you how to build your own dashboard using `{shiny}`.

Shiny is an R package (`{shiny}`) that allows you to directly create from R interactive web applications. The goal of shiny is to 'convert' your R code into an application that can be accessible and used by anyone through their web browser, without having to be familiar with R.

This procedure is made available as `{shiny}` uses some carefully curated set of user interface (UI) functions that generate the HTML, CSS, and JavaScript code needed for most common tasks. In most cases, further knowledge of these languages is not required…unless you want to push your application further. Additionally, it introduces a new way of programming called *reactive programming*, which tracks automatically dependencies between code: When a change in an input is detected, any code that is affected by this change will automatically be updated.

14.2.2   Starting with Shiny

To create your very first shiny application, you can click on R studio in the *new page* icon and select *Shiny Web App…* Once you filled in the relevant information (name, author), you can then decide whether you want to create one unique file (`app.R`) or multiple files (`ui.R` and `server.R`).

Both solutions are equivalent and work the same: In both cases, a `ui()` and a `server()` function are generated. Due to better readability, and to ease its maintenance over time, we recommend to use the single file for short applications, and to use multiple files for larger applications (larger meaning with more code lines).

For our short application, we consider that a single file was more convenient and hence we used that solution.

14.2.2.1   Illustration

For illustration, let's consider a simple application in which we import a data set that contains sensory data, as collected from a trained panel. In this example, the data set structure follows the one in *Sensory Profile.xlsx*.

For simplification purposes, the code developed for this application requires that the data set contains one column called *Judge* (containing the panelist information), one column called *Product* (containing the product information), all the other columns being quantitative (corresponding to the sensory attributes).

The goal of the application is then to compute from this raw data set the sensory profiles of the products (mean table with attributes in rows, and products in columns) that we display on screen. Furthermore, we also represent these sensory profiles graphically in a spider plot or in a circular bar plot. Since the main goal of shiny application is in its reactivity and interactivity, we give the user the opportunity to remove/add/reorder the attributes on the table and plots, and to hide/show products to increase visibility.

Once the graphics match our needs, we also propose to download it as a *.png* file to integrate it in our report.

From a learning perspective, this application introduces you specifically to:

- Importing an external file to the application;
- Create options that that are both independent (type of graph to produce) and depend of the file imported (list of attributes and products);
- Run some analyses (compute the means) and display the results as a table and as a plot;
- Export the graph to your computer as a png file.

14.2.2.2   User Interface

The user interface (UI) is the part of the application that controls what the user sees and controls. In our example, the UI is separated into two parts:

- the left panel which contains the options that the user can manually change;
- the right (or main) panel which contains the outputs.

In the *app.R* file, this information is stored in the `ui()` function, and the two panels can be found in `sidebarPanel()` and in `mainPanel()` respectively.

In `sidebarPanel()`, all the options are set up. These options include `fileInput()` for importing the data set, or `radioButtons()` to control the type of plot to generate. A large list of options exist including `numericInput()`, `sliderInput()`, `textInput()`, `passwordInput()`, `dateInput()`, `selectInput()`, `checkboxInput()` etc. Note that we also extended our library of options by adding `checkboxGroupInput()` from the `{shinyjs}` package[1].

---

[1]To use `{shinyjs}`, you need to load the library and add `useShinyjs(),` at the start of your `ui()` code.

In some cases, the option of interest cannot be defined on the UI side since they depend on the data itself. It is the case for the product or attribute selection, which are then defined using `uiOutput()`. As we will see in the next section, these options are created on the server side.

On `mainPanel()`, `tabsetPanel()` and `tabPanel()` control for the design of the output section. In our example, we create two tabs, one for the table and one for the graph, but we could have opted out to print everything on one page.

In our case, the `mainPanel()` only aims in exporting results computed on the server side. Depending on the type of output generated, the corresponding function should be used. In this case, we use `tableOutput()` to retrieve the table generated with `renderTable()`, `plotOutput()` to retrieve the plot generated with `renderPlot()`, or `downloadButton()` to retrieve the plot generated with `downloadHandler()`. Generally speaking, the `xxxOutput()` function (UI side) is used to retrieve the output generated (server side) from the corresponding `renderXxx()` function.

14.2.2.3   Server

The server side of the application is where all the computations are being performed, including the construction of tables, figures, etc.

Since the options defined on the UI side should affect the computations performed (e.g. our decision on the type of plot to design should affect the plot generated), we need to communicate these decisions to the server, and use them. On the server side, any information (or option) passed to the server side is done through `input$name_option`. In our previous example regarding the type of graph to generate, this is shown as `input$plottype`, as defined by:

```
radioButtons("plottype", "Type of Plot to Draw:",
             choices=c("Spider Plot"="line", "Circular Barplot"="bar"),
             selected="line", inline=TRUE)
```

In this case, if the user select *Spider Plot* (resp. *Circular Barplot*), `input$plottype` will take the value *line* (resp. *bar*).

Reversely, any information that is being build on the server side and that should be passed on the UI part of the application can either be done via the `xxxOutput()`/`renderXxx()` combination presented before (useful for showing results), or by using the `renderUI()`/`uiOutput()` combination (useful for options that are *server-dependent*).

Following a similar communication system than the one from UI to server, the part generated on the server side is stored as `output$name_option` (defined as `renderUI()`) and is captured on the UI side using `uiOutput("name_option")`.

In our example, the latter combination is used for the two options that require reading the data set first, namely the selection of attributes and the selection of products.

```r
# server side:
  output$attribute <- renderUI({

    req(mydata())

    items <- mydata() %>%
      pull(Attribute) %>%
      as.character() %>%
      unique()

    selectInput("attribute", "Select the Attributes (in order) ", choices=items, selected=items,

  })

  output$product <- renderUI({

    req(mydata())
    items <- mydata() %>%
      pull(Product) %>%
      unique()

    checkboxGroupInput("product", "Select the Products to Display:", choices=items, selected=item

  })

# UI side:
  uiOutput("attribute")
  uiOutput("product")
```

Lastly, we have elements that are only relevant on the server side, namely the computation themselves. In our example, these are results of a function called `reactive()`.

Reactivity (and its corresponding `reactive()` function) is a great *lazy* feature of `{shiny}` that was designed so that the computations are only performed when necessary, i.e. when the changes applied in the options affect the computations. This laziness is of great power since only computations that are strictly needed are being performed, hence increasing speed by limiting the computation power required to its minimum.

Let's break this down in a simple example:

```r
mydata <- reactive({

  req(input$datafile)
  data <- readxl::read_xlsx(input$datafile$datapath, sheet=1) %>%
    pivot_longer(-c(Judge, Product), names_to="Attribute", values_to="Score") %>%
    mutate(Attribute = fct_inorder(Attribute), Score = as.numeric(Score)) %>%
    group_by(Product, Attribute) %>%
    summarize(Score = mean(Score)) %>%
    ungroup()

  return(data)

})
```

In this section, we read the file that was selected through the `fileInput()`
option called `datafile` from the UI side. Note that in this case, the path of the
file is stored in the object called `datapath`, meaning that to access this file, we
need to read `input$datafile$datapath`.

Once read (here using `{readxl}`), we do some small transformations to the data
before saving its final version in an object called `mydata`. Since this block of
code only depends on `input$datafile` (UI side), this part will no longer be in
use unless `datafile` is being updated or changed.

For the mean computation, the same procedure applies as well:

```r
mymean <- reactive({

  req(mydata(), input$attribute, input$product)
  mymean <- mydata() %>%
    mutate(across(c("Product", "Attribute"), as.character)) %>%
    filter(Attribute %in% input$attribute) %>%
    mutate(Product = factor(Product, input$product),
           Attribute = factor(Attribute, input$attribute),
           Score = format(round(Score, 2), nsmall=2)) %>%
    pivot_wider(names_from=Product, values_from=Score)

  return(mymean)
})
```

For this `reactive()` block, `mymean` depends on `mydata`, `input$attribute`, and
`input$product`. This means that if `datafile` (read `mydata`), `input$attribute`
and/or `input$product` change, the computations re-run and `mymean` is getting
updated.

For small and simple examples like ours, this domain of reactivity may be suffi-
cient, and would be sufficient in many cases. There are however some few points

that require a bit more explanations.

First, we advise that you use `reactive()` as much as possible: In our example, we could have created the code to build the graph within `renderPlot()`. However, this way of coding is not efficient since it would always be updated, even when it is not necessary. For small examples such as the one proposed here, this may not make much difference, but for larger applications it would have a larger impact. This is why we prefer to create the graphs in a `reactive()` instance, and simply retrieve it for display.

Second, and as you may have seen already, the output of a `reactive()` section can be re-used in other sections. This means that just like in *regular coding*, you can save elements in R object that you can re-use later (e.g. `mydata`, `mymean`, or `myplot`). However, these elements act like functions, meaning that if you want to call them, you should do it as `mydata()` for instance. More generally, let's imagine that `mydata` is a list with two elements (say `mydata$element1` and `mydata$element2`), we would retrieve `element1` as `mydata()$element1`.

Third, let's introduce the function `req()` that is used at the start of almost every block of code on the server side. To do so, let's take the example of `output$attribute` which starts with `req(mydata())`. The `req()` functions aims in requiring the object mentioned (here `mydata()`) before running: if `mydata()` doesn't exist, then `output$attribute` is set as `NULL`. This small line of code comes handy as it avoids returning errors until a data set is being loaded: How to read the list of attributes to consider from data that do not exist yet?

Finally, the application that we are developing here is *over-reactive* as every change we do will create update results. To highlight this, just remove some attributes from the list and you'll see the mean table or graphic being updated. In our small example, this is not too problematic since the application runs fast, but in other instances in which more computation is required, you may not want to wait that each little change done is being processed. To over come this, you can replace `reactive()` by `eventReactive()` by creating a button (e.g. *Run* or *Apply changes*) that will only trigger changes once pushed. This means that changes are only performed on a users' action.

14.2.3   Deploying the Application

To run your application, you can simply do it in two different ways from RStudio: You can either push the *Run app* button on the task bar of your script (a menu allows you to run the app in the Viewer window, or as a new window), or you can use the shortcut *CTRL+SHIFT+ENTER* (on windows). In this case, your computer (and RStudio) will use a virtual local server to build the application.

Unfortunately, this solution is not sufficient in case you really want to share it with colleagues. In such case, you need to publish your app by installing it on a server. Ideally, you have access to a server that can host such application

(check with your IT). If not, you can always create an account on https://www.
shinyapps.io/admin/ and install it there. Note that there is a free registration
option, which comes with limitations (number of applications to install). Also,
before using their services, make sure that their conditions (privacy, protection
of the data, etc.) suit you and your company policy.

14.3   To go further...

Quickly, `{shiny}` became popular and many researchers developed additional
packages and tools to further enhance it. Without being exhaustive, here are a
few tools, widgets, and packages that we often use as they provide interesting
additional features. But don't hesitate to look online for features that answer
your needs!

14.3.1   Personalizing and Tuning your application

If you start building many applications in `{shiny}`, you might get tired of
its default layout. Fortunately, the overall appearance of applications can be
modified easily, especially if you have notion in other programming language
such as CSS and HTML. If not, no worries, there are alternative solutions for
you, including the `{bslib}` package. To change the layout, simply load the
library, and add the following piece of code at the start of your application
(here for using the theme *darkly*):

```
fluidPage(
  theme = bslib::bs_theme(bootswatch = "darkly")
)
```

`{bslib}` also allows you creating your own theme that match your company
style. So do not hesitate to take some time to build it once, and to apply then
to all your applications.

Besides changing the overall theme of your applications, there are certain details
that can make your overall application more appealing and easier to use. In our
short example, you may have noticed that each option and each output is a
separate line. This is sufficient here since the number of options and outputs
are very limited. However, as the application grows in size, this solution is no
longer sustainable as you do not want the users to scroll down a long way to
run adjust all the options.

Instead, you can combine different options (or outputs) on one line. To do so,
`{shiny}` works by defining each panel as a table with as many rows as needed
and 12 columns. When not specified, the whole width (i.e. the 12 columns) of
the screen is used, but this could be controlled through the `column()` function.

In our example, we could have position the product selection and the attribute selection on the same line using:

```
fixedRow(
  column(6, uiOutput("product")),
  column(6, uiOutput("attribute"))
)
```

14.3.2 Upgrading Tables

The default table built in `{shiny}` using the `tableOutput()`/`renderTable()` combination is very handy, yet limited in its layout. Fortunately, many additional packages that create HTML tables have been developed to provide alternative solution to build more flexible tables.

In particular, we recommend the use of the `{DT}` and `{rhandsontable}` packages as they are very simple to use, and yet provides a large variety of powerful design options. Just to name a few, it allows:

- cells or text formatting (e.g. coloring, rounding, adding currencies or other units, etc.);
- merge rows or columns;
- add search/filter fields to the table;
- provide interactivity, that for instance can be connected to graphics;
- include graphics within cells;
- allows manual editing, giving the user the chance to fill in or modify some cells;
- etc.

To build such table, you will need to load the corresponding packages. For `{DT}` tables, you can generate and retrieve them using the functions `renderDataTable()` and `dataTableOutput()`, or its concise forms `renderDT()` and `DTOutput()`[2]. For `{rhandontable}` tables, you can generate and retreve them using `renderRHandsontable()` and `rHandsontableOutput()`.

For more information and examples, please look at https://rstudio.github.io/DT/ and https://jrowen.github.io/rhandsontable/.

14.3.3 Building Dashboard

The example used here illustrates the power of `{shiny}`. However, it is limited to our own data set, meaning that it is study specific. What if we would want

---

[2]the table should be generated using `datatable()`.

to create a dashboard, that is connected to a database for instance and that updates its results as soon as new data is being collected?

This is of course the next step, and `{shiny}` can handle this thanks to the `{shinydashboard}` package.

In its principle, `{shinydashboard}` works in a similar way to `{shiny}` itself, except for the structure of the UI. For `{shinydashboard}`, the UI contains three sections as shown below (the example below generates an empty dashboard.):

```
library(shiny)
library(shinydashboard)

ui <- dashboardPage(
  dashboardHeader(),
  dashboardSidebar(),
  dashboardBody()
)

server <- function(input, output) { }

shinyApp(ui, server)
```

It is then your task to fill in the dashboard by for instance connecting to your data source (e.g. a simple data set, a database, etc.) and to fill in your different tabs with the interactivity and outputs of interest.

For more information, including a comprehensive *Get started* section, we recommend you to visit https://rstudio.github.io/shinydashboard/

14.3.4   Interactive Graphics

Through its way of working, `{shiny}` creates some interactivity to graphics by updating it when changing some options. This is hence done by replacing a static graph by another static graph.

However, R provides other options that creates interactive graphs directly. This can be done thanks to the `{plotly}` library.

`{plotly]` is a library that can be used to build R graphics in general within the R environment: It is not specific to `{shiny}` and can be used outside Shiny applications. To build `{plotly}` visualizations, you can build it directly from scratch using the `plot_ly()` function. It is out of the scope of this book to develop further how `{plotly}` works, mainly because we made the decision to explain in details how `{ggplot2}` works. And fortunately, `{plotly}` provides an easy solution to convert automatically `{ggplot2}` graph to `{plotly}` thanks to the `ggplotly()` function.

For more information, please visit https://plotly.com/r/

14.3.5 Interactive Documents

Ultimately, `{shiny}` can also be combined to other tools such as `{rmarkdown}` to build interactive tutorials, teaching material, etc. This is done by integrating the interactivity of `{shiny}` to propose options and reactive outputs into a text editor through `{rmarkdown}`.

To integrate `{shiny}` in your `{rmarkdown}`, simply add `runtime: shiny` in the YAML metadata of your R markdown document.

14.3.6 Documentation and Books

Thanks to its way of working and its numerous extensions, there is (almost) no limit to applications you can build (except maybe for your imagination?). For inspiration, and to get a better idea of the powerful applications that you can build, have a look at the gallery on the official shiny webpage: https://shiny.rstudio.com/gallery/

In this section, we just introduced you to the main functions available in `{shiny}`, but if you want to go further, there is a whole world for you to explore. Of course, a lot of material is available online, and we would not know where to start to guide you. However, we strongly recommend you to start with the book from Hadley Wickham entitled *Mastering Shiny* (REF) as it is comprehensive and will give you the kick start that you need, and more.

Digestifs

Chapter 15

Conclusion and Next Steps

Congratulations, you've reached the end of this book! ADD COMMENTS

15.1   Sensory Analysis in R

15.2   Other Recommended Resources

15.2.1   R for Data Science

15.2.2   Hands-On Machine Learning with R

15.2.3   FactoExtra

15.2.4   R Graphics Cookbook

15.2.5   Storytelling with Data

15.2.6   Text Mining wtih R

15.3   Databases (Graph and Otherwise)

15.4   Python and Other Languages

Bibliography

Bibliography

Bécue-Bertaut, M. (2019). Textual Data Science with R. *Textual Data Science with R.*

Bleibaum, R. N., editor (2020). *Descriptive Analysis Testing for Sensory Evaluation.* ASTM International, 2nd edition.

Brockhoff, P. B. (2011). Sensometrics.

Cao, L. (2017). Data science: A comprehensive overview. *ACM Computing Surveys*, 50(3):1–42.

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.

Davenport, T. H. and Patil, D. J. (2012). Data scientist. *Harvard business review*, 90(5):70–76.

Delarue, J., Lawlor, B., and Rogeaux, M. (2015). *Rapid Sensory Profiling Techniques and Related Methods: Applications in New Product Development and Consumer Research.* Woodhead Publishing Ltd, Cambridge, UK.

Fayyad, U., Piatetsky-Shapiro, G., and Smyth, P. (1996). From data mining to knowledge discovery in databases. *AI magazine*, 17(3):37.

Fisher, R. A. (1935). The Design of Experiments (Hafner). *New York.*

Franczak, B. C., Browne, R. P., McNicholas, P. D., and Findlay, C. J. (2015). Product selection for liking studies: The sensory informed design. *Food Quality and Preference*, 44:36–43.

Hvitfeldt, E. and Silge, J. (2021). *Supervised Machine Learning for Text Analysis in R.*

Kahneman, D. and Tversky, A. (2000). *Choices, values and frames.* Cambridge University Press and the Russell Sage Foundation, Cambridge.

Lawless, H. T. and Heymann, H. (2010). *Sensory evaluation of food: principles and practices.* Food Science Text Series. Springer New York, 2 edition.

Lawson, J. (2014). *Design and Analysis of Experiments with R.*

Macfie, H. J., Bratchell, N., Greenhoff, K., and Vallis, L. V. (1989). Designs to balance the effect of order of presentation and first-order carry-over effects in hall tests. *Journal of Sensory Studies*, 4(2):129–148.

Naur, P. (1974). *Concise survey of computer methods.* Petrocelli Books.

Peng, R. D. (2011). Reproducible research in computational science. *Science*, 334(6060):1226–1227.

Piqueras-fiszman, B. (2015). Open-ended questions in sensory testing practice. In *Rapid Sensory Profiling Techniques and Related Methods: Applications in New Product Development and Consumer Research*, pages 247–267. Woodhead Publishing Ltd.

Qannari, E. M. (2017). Sensometrics approaches in sensory and consumer research. *Current Opinion in Food Science*, 15:8–13.

Schlich, P. (1993). *Contribution à la sensométrie.* PhD thesis, Université Paris-Sud.

Silge, J. and Robinson, D. (2017). *Text mining with R: A tidy approach.* O'Reilly Media, Inc.

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*, volume 2. Reading, Mass.

Varela, P. and Ares, G. (2012). Sensory profiling, the blurred line between sensory and consumer science. A review of novel methods for product characterization. *Food Research International*, 48(2):893–908.

Wakeling, I. N. and MacFie, H. J. (1995). Designing consumer trials balanced for first and higher orders of carry-over effect when only a subset of k samples from t may be tested. *Food Quality and Preference*, 6(4):299–308.

Wickham, H. (2014). Tidy data. *Journal of statistical software*, 59(1):1–23.

Wickham, H. and Grolemund, G. (2016). *R for data science: import, tidy, transform, visualize, and model data.* " O'Reilly Media, Inc.".

Williams, A. and Langron, S. (1984). The use of Free-choice Profiling for the evaluation of commercial ports. 35(5):558–568.

Wu, C. F. J. (1997). Statistics = Data Science?