Data Science for Sensory and Consumer Scientists

John Ennis, Julien Delarue, and Thierry Worch

2021-09-30

Contents

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'Oeurves" 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 **??** 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].

Cautions: Don't that Everybody Does

How to Contact Us

Acknowledgements

[1] You may need to load these libraries before the code will run, please see Appendix **??** 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 territories.

Core principles in Sensory and Consumer Science

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

Measuring and analyzing human responses

Sensory and consumer science aims at measuring and understanding consumers' sensory perceptions as well as the judgements, emotions and behaviors that may arise from these perceptions. SCS is thus primarily a science of measurement, although a very particular one that uses human beings and their senses as measuring instruments. In other words, sensory and consumer researchers measure and analyze human responses.

To this end, SCS relies essentially on sensory evaluation which comprises a set of techniques that mostly derive from psychophysics and behavioral research. It uses psychological models to help separate signal from noise in collected data [ref O'Mahony, D.Ennis, others?]. Besides, sensory evaluation has developed its own methodological framework that includes most refined techniques for the accurate measurement of product sensory properties while minimizing the potentially biasing effects of brand identity and the influence of other external information on consumer perception [Lawless & Heymann, 2010].

A detailed description of sensory methods is beyond the scope of this book and many textbooks on sensory evaluation methods are available to readers seeking more information. However, just to give a brief overview, it is worth remembering that sensory methods can be roughly divided into three categories, each of them bearing many variants:

- Discrimination tests that aim at detecting subtle differences between two products.
- Descriptive analysis (DA), also referred to as 'sensory profiling', aims at providing both qualitative and quantitative information about product sensory properties.
- Hedonic tests. This category gathers affective tests that aim at measuring consumers' liking for the tested products or their preferences among a product set.

Each of these test categories generates its own type of data and related statistical questions in relation to the objectives of the study. Typically, data from difference tests consist in series of correct/failed binary answers depending on whether judges successfully picked the odd sample(s) among a set of three or more samples. These are used to determine whether the number of correct choices is above the level expected by chance.

Conventional descriptive analysis data consist in intensity scores given by each panelist to evaluated samples on a series of sensory attributes, hence resulting in a product x attribute x panelist dataset (Figure 1). Note that depending on the DA method, quantifying means other than intensity ratings can be used (ranks, frequency, etc.). Most frequently, each panelist evaluates all the samples in the product set. However, the use of balanced incomplete design can also be found when the experimenters aim to limit the number of samples evaluated by each subject.

Eventually, hedonic test datasets consist in hedonic scores (ratings for consumers' degree of liking or preference ranks) given by each interviewed consumer to a series of products. As for DA, each consumer usually evaluates all the samples in the product set, but balanced incomplete designs are sometimes used too. In addition, some companies favor pure monadic evaluation of product (i.e. between-subject design or independent groups design) which obviously result in unrelated sample datasets.

Sensory and consumer researchers also borrow methods from other fields, in particular from sociology and experimental psychology. Definitely a multidisciplinary area, SCS develops in many directions and reaches disciplines that range from genetics and physiology to social marketing, behavioral economics and computational neuroscience. So have diversified the types of data sensory and consumer scientists must deal with.

Computational Sensory Science

Hors d'Oeuvres

Chapter 2

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 **?** for a more general introduction to data science.

2.1 History and Definition

You may have heard that data science was called the "sexiest job of the 21st century" by Harvard Business Review (**?**). 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 **?**.

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" (**?**). 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" (**?**). This movement culminated in 1997 when Jeff Wu proposed during his inaugural lecture upon

[1] A prize roughly equivalent in prestige to a Nobel prize, but for computer science.

becoming the chair of the University of Michigan's statistics department, entitled "Statistics = Data Science?," that statistics should be called data science (**?**).

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" (**?**). 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 **?**, for example.

2.2 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.

2.2.1 Reproducible Research

One of the most important ideas in data science is that of reproducible research (cf. **?**). 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.

2.2.2 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 **??**), 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 (see Chapter **??**).

- **Shared Learning** Once a team combines standardized reporting with tools for online collaboration such as GitHub (see Appendix **??**), 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.

2.3 Data Scientific Workflow

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

2.3.1 Data Collection

2.3.1.1 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. **?**). 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 **?**.

Since Fisher's time, ideas around experimental design have relaxed somewhat, with Tukey arguing in **?** 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 interesting results too frequently to be comfortable.* (Emphasis original)"

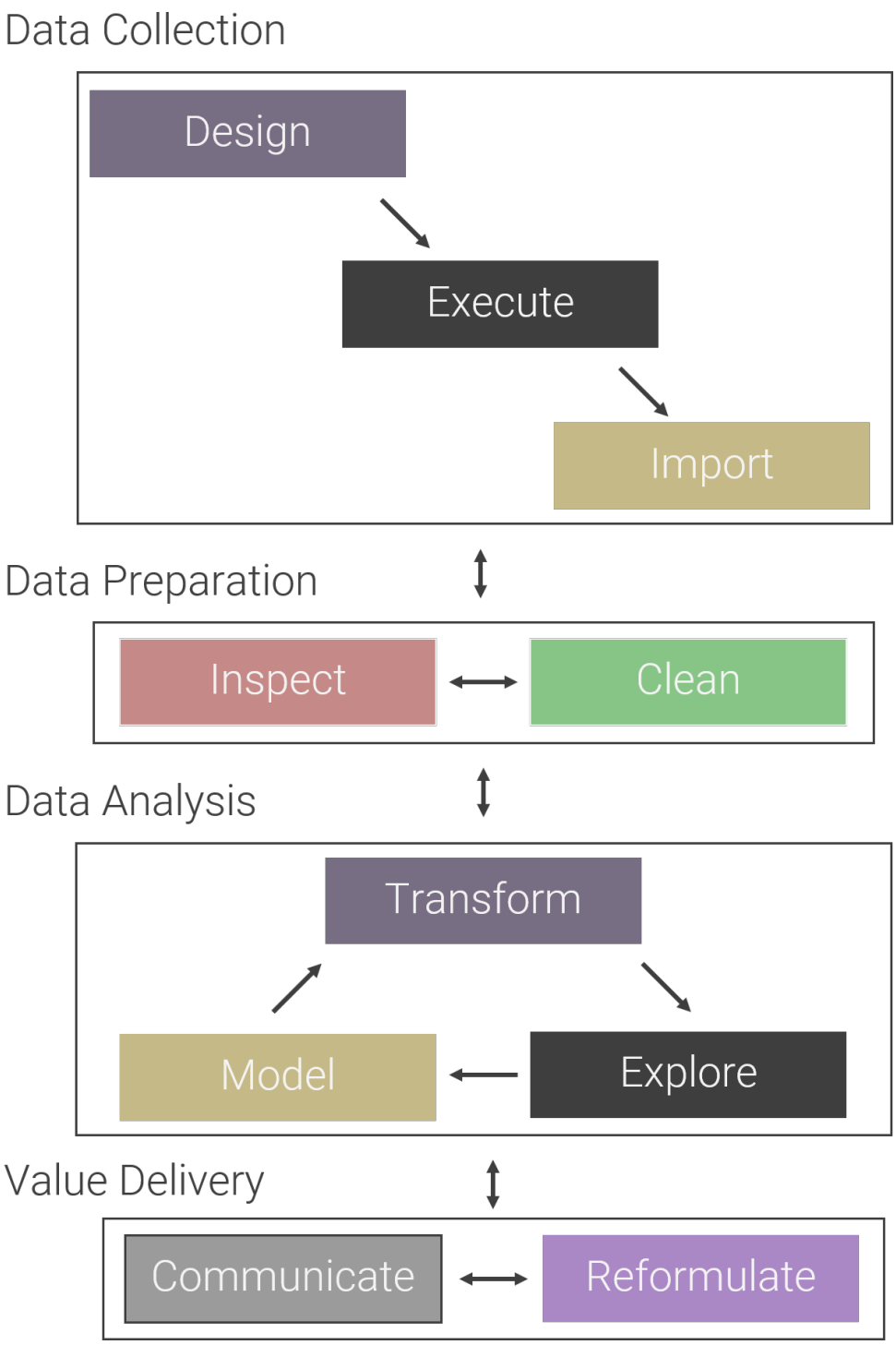

Figure 2.1: Data scientific workflow.

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 will describe in Chapters **??** and **??**, well-selected sample sets support more generalizable predictions from machine learning models.

2.3.1.2 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.

2.3.1.3 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 **??**, 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.

2.3.2 Data Preparation

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

2.3.2.1 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 **??** of Chapter **??**.

2.3.2.2 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 **??**. 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 **??**.

2.3.3 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 **?** 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 **??**. The typical steps of data analysis are data transformation, exploration, and modeling, which we review below.

2.3.3.1 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

[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

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 **??**), 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 **??**.

2.3.3.2 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 **??**) and exploratory analyses (covered in Chapter **??**). 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 **??** we will discuss techniques, such as cross-validation, that can help mitigate this risk.

2.3.3.3 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 **??**, 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 **??**.

2.3.4 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 appear-

ance 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 **??**, we briefly review the two steps of value delivery, communicate and reformulate, below.

2.3.4.1 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 **??**, and note that automated reporting, which we cover in Chapter **??** also plays a large role in this step by freeing us to spend more time thinking about the storytelling aspects of our communications.

2.3.4.2 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 **??**.

2.4 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 students and clients is to get started sooner rather than later, and to accept that

the code you'll write in the future will always be better than the code you'll write today. Also, many of the small details that separate an proficient data scientist from a novice can only 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.

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

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

Chapter 3

Data Manipulation

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 (PxC) rows x 3 columns (consumer, sample, and the liking scores).

However, to assess whether consumers have the same preference patterns at the individual level, internal preference mapping or cluster analysis would be 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. After presenting many different ways to transform your data, we present several simple examples to show these ideas in practice[1].

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

3.1 Tidying Data

Hadley Wickham (**?**) 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.

3.1.1 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, and will be reusable if you need to perform the same operations on similar data in the future (including updated versions of the current dataset). Moreover, these operations will become easier and more natural for you to use as you familiarize yourself with them.

3.1.1.1 Handling Columns

3.1.1.1.1 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 dataset.

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

First, here are the original names:

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

And here are the recoded names:

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

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

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.), 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 was performed to all the variables, it is possible to ignore certain variables for the transformation.

3.1.1.1.2 Re-Organizing Columns Another simple transformation consists in re-organizing the dataset, either by re-ordering (including removing) the 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` dataset, 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 dataset. Let's limit ourselves in selecting `Judge`, `Product`, and `Shiny`:

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

[3]Note that many other packages include a function called `select()`, which could create conflicts. To avoid any risks of errors, we recommend calling the `select()` function using the notation `dplyr::select()` as it formally calls the `select()` function 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.

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 variable to remove rather than informing all the variables 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))
```

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 the `Judge` and `Product` columns:

```
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 `select()` are provided throughout the book.

3.1.1.1.3 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 will only be using these three variables, let's reduce the dataset to these three variables with `select()` first 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, it will be used extensively throughout this book.

Since performing mathematical computation on non-numerical columns is not possible, conditions can easily be added through `mutate()` combined with `across()`. An example could be to round all variables to 0 decimal, which can only be applied to numerical variables:

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

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

3.1.1.1.4 Mergeing 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 Last Name 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 whether 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 the 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` should be set to `FALSE`.

To reverse the changes (saved here in `prodinfo`) and to separate a column into different variables, the function `separate()` from the `{tidyr}` package is required. 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 separated defined by `sep`. In our example, this would be done as following:

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

3.1.1.2 Handling Rows

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

1. by re-arranging the rows in a logical way,
2. by filtering entries based on a given variables,
3. splitting the data in sub-groups based on the entries of a variable.

3.1.1.2.1 Re-arranging Rows The first step of re-arranging rows is done through the `arrange()` function from the `{dplyr}` package. This function al-

lows 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:

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

3.1.1.2.2 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")`);
- for multiple 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. In a second step, we can filter out all missing values from Shiny:

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

[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 of type factor in which case the order of the levels for that factors are being used.

```
sensory_na

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 'clean' table only has 79 rows and 3 columns.

3.1.1.2.3 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. To some extent, this is what we have done when we only filtered data from sample `P02`. To get sub-groups of data for each sample, we could repeat the same procedure for all the other samples. However, this procedure becomes tedious as the number of samples increases. For such task, we prefer the use of the function `split()`, which takes as argument the column to split from:

```
sensory %>%
  split(.$Product)
```

This function creates a list of n elements (n being the number of samples here), each element corresponding to the data related to one sample. From there, automated analyses can be performed to each of the sub-data through the `map()` function, as it will be illustrated later.

3.1.2 Reshaping the Data

Reshaping the data itself is done through pivoting, hence either creating a longer and thinner table (CREATE FIGURE), or a shorter and wider table (CREATE FIGURE). This is done through the `pivot_longer()` and `pivot_wider()` functions from the `{tidyr}` package.

3.1.2.1 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.

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 will be considered.

In case the attribute names are following a standard structure, say "attribute_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 (for clarity, all the other variables are 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", names_se
```

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 `names_pattern` can be considered. More information on this topic is provided in REF CHAPTER TEXTUAL.

3.1.2.2 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 dataset 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 dataset 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 missign 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.

3.1.3 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).

3.1.3.1 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()` 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.

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

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

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.

3.1.3.2 Illustrations of Data Manipulation

Let's review the different transformations presented earlier by generating the sensory profiles of the samples through different approaches[5].

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:

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

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

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

```
sensory_attr <- colnames(sensory)[4:ncol(sensory)]
sensory %>%
  group_by(Product) %>%
  summarise(across(all_of(sensory_attr), mean))
```

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 transformations should be performed across all samples. To avoid such behavior, we strongly recommend you to apply `ungroup()` as soon as the results per group has 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") %>%
  group_by(Product, Attribute) %>%
  summarise(Scores = mean(Scores)) %>%
  pivot_wider(names_from=Attribute, values_from=Scores) %>%
  ungroup()
```

One can notice that through this procedure, the order of the attributes are no longer following the same sequence, and have been ordered in alphabetical order. To maintain the original order, the Attribute column should be transformed into a factor in which the levels are in their original order.

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 cell contains `dbl [9]` corresponding 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 from `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)
```

3.1.4 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.

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.

To combine the tables vertically, we could use the basic R function `rbind()`. However, we prefer the use of `bind_rows()` from the `{dplyr}` package since it better controls for the columns by ensuring that the order is well respected (in case one table contains a variable that the other tables do not, it will keep the variables and allocate NAs when this 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 (we do this here since this information was not available within the tables: If it were, the parameter `.id` could have been ignored).

```
library(here)
library(readxl)

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 **??**.

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 them to avoid mishaps.

Alternatively it is possible to merge tables using `merge()` from `{base}`, or the different `*_join()` functions from the `{dplyr}` package. In that case, the tables do not need to be in the same order, nor from the same size, since the function will handle that.

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) [corresponds to `merge()` with `all=TRUE`];
- `inner_join()` only keeps the common cases, i.e. cases that are present in both X and Y [corresponds to `merge()`with `all=FALSE`];
- `left_join()` keeps all the cases from X [corresponds to `merge()`with `all.x=TRUE` and `all.y=FALSE`];
- `right_join()` keeps all the cases from Y [corresponds to `merge()`with `all.x=FALSE` and `all.y=TRUE`];
- `anti_join()` only keeps the elements from X that do 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 dataset called *Consumer Test.xlsx*, which contains three tabs:

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

library(readxl)
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)

consumption
```

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 dataset contain the full evaluation of the cookies (each consumer evaluating each product), the joining procedure needs to be done by product and by consumer 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 by pasting a J in fron 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 dataset. We could rename these columns in one dataset 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 dataset are now joined in one and could be further processed for some analyses!

Chapter 4

Data Visualization

4.1 Design Principles

4.2 Table Making

4.3 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 also 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 package `{ggplot2}`.

4.3.1 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're 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 system that is similar to pipes, except that the symbol used is `+` instead of `%>%`.

`{ggplot2}` is a multi-layer graphical tools, meaning that the graphics are build by adding layers one at a time to finally build your graphics. This means that

`ggplot` objects can be printed at any time, and yet 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 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. This should be more than sufficient to get you started, and should cover 90% of the graphics that are relevant in sensory and consumer science. But if that should not be the case, we invite you to look into the online documentation or to references such as [REFS].

4.3.2 Getting started with `{ggplot2}`

To use `{ggplot2}`, we need to load this package: 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}`, we will use the sensory dataset stored in *Sensory Profile.xlsx*.

```
library(here)
library(readxl)

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)
```

To initiate a graph, we start by calling the function `ggplot()`. Since the data we want to use is stored in `sensory`, we apply `ggplot()` to `sensory`:

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

If you run this line of code, you'll notice that p contains an empty graphic. This is because we haven't added any layer that is relevant for us yet. So let's imagine we want to look at the overall relationship between `Sticky` and `Melting`. To do so, we want to create a scatter plot with `Sticky` in the x-axis, and `Melting` in the y-axis. To do so, we require to call a `geom_` function that relates to scatter plots, in this case `geom_point()`, by also informing through the aesthetics (defined through `aes()`) what needs to be considered on the x-axis and on the y-axis.

This provides something similar to:

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

A scatter plot is hence generated. In this example, one can notice that many points are being printed: this can easily be explained by the fact that the raw sensory data are being used, meaning that there are as many points as there are assessors evaluating products.

Let's imagine we want to color the points per products to see if we can see patterns. This can be done by informing within the aesthetics that `colour = Product`:

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

It is important to note that parameters can depend on a variable (e.g. `colour` in the previous example) or should be applied to all the elements of the graph. In the first case, the parameters is included within the aesthetics, whereas in the second case, it should be defined outside the aesthetics. Let's illustrate this by providing a simple example in which we change the type of dots (from circle to square) using `pch`, and by increasing their size using `cex` to all the points of the graph:

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

Since `{ggplot2}` is a multi-layer graph, let's add another layer that contains the name of the panellists that is associated to each point. To do so, we could consider `geom_text()`. In that case, we need to inform in `aes()` what corresponds to the x-axis, what correspond to the y-axis, and `label`. To avoid having the label overlapping with the point, we propose to shift slightly the text vertically (using `nudge_y`):

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

initiate graph with ggplot() add geom_ (discussion around geom_ that have some statistical computation integrated...) include aes() what is not an aesthetic - fixed options aes() included in ggplot are passed to every geom, unless stated differently! play around with themes miscalleneous

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

4.3.2.1 aesthetics

Provide the most relevant options for `aes()`

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

4.3.2.2 geom_

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

4.3.3 Common Charts

4.3.3.1 Scatter points

`geom_point()`

4.3.3.2 Line charts

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

4.3.3.3 Bar charts

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

4.3.3.4 Distribution

`geom_boxplot()` and `geom_violin()`

4.3.3.5 Text

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

4.3.3.6 Rectangles

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

4.3.3.7 Themes and legend

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

4.3.4 Additional Topics

4.3.4.1 Playing around with axes

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

4.3.4.2 Transposing the plot

`coord_flip()` and `coord_polar()`

4.3.4.3 Splitting plots

`facet_wrap()`, `facet_grid()`

4.3.4.4 Combining plots

{patchwork}

Chapter 5

Automated Reporting

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

5.1 What is Automated Reporting?

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

5.2 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 an Excel format, and most data analysis tools 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 such spreadsheets. It is even less a surprise that R provides multiple solutions to import/export results from/to Excel.

For the import of datasets, we have already presented the package `{readxl}` among others. 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 the extension of `{readxl}` 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) to export to the file specified in the `path` parameter.

```
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)

#Basic data manipulation: writing a subset into a new xlsx
product_info %>%
  filter(Protein %in% "High") %>%
  write_xlsx(path = file.path("output", "High Protein Products.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 `wb` using the `createWorkbook()` function, to which we add worksheets through the `addWorksheet()` function. On a given worksheet, any table can be exported using `writeData()` or `writeDataTable()`, which controls where to write the

table through the `startRow` and `startCol` options. Through these different functions, many additional formatting procedure can be applied:

- `createWorksheet()` allows:
  - show/hide grid lines using `gridLines`;
  - 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`).
- `writeData()` and `writeDataTable()` allow:
  - controlling where to print the data 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;
  - including the row names and column names through `rowNames` and `colNames`;
  - formatting the header using `headerStyle` (incl. color of the text and/or background, font, font size, etc.);
  - shaping the borders using predefined solutions through `borders`, or customizing them with `borderStyle` and `borderColour`;
  - adding a filter to the table using `withFilter`;
  - converts missing data to "#N/A" or any other string using `keepNA` and `na.string`.
- Additional formatting can be controlled using:
  - `options()` to pre-define number formatting, border colors and style that will be applied automatically to each table;
  - `modifyBaseFont()` to define the font name and font size;
  - `freezePane()` to freeze the first row and/or column of the table using `firstRow = TRUE` and `firstCol = TRUE`;
  - `createStyle()` to pre-define a style, or `addStyle()` to apply the styling to selected cells;
  - `setColWidths` to control column width;
  - `conditionalFormatting()` styling of cells when they meet pre-defined rules, as for instance to highlight significant p-values.

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

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

In the following example, the sensory profiles are exported into a newly created sheet.

To introduce conditional formatting with `{openxlsx}`, the sensory profiles are color coded as following: for each cell, the value is compared to the overall mean computed for that column and is colored in red (resp. blue) if it's higher (resp. lower) than the mean. In practice, the color style is pre-defined in two parameters called `pos_style` (red) and `neg_style` (blue) using `createStyle()`. The decision whether `pos_style` or `neg_style` should be used is defined by the `rule` parameter from the `conditionalFormatting()`[1] function.

```
library(openxlsx)

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

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

# Create a header style in which
  # a blue background is used,
  # borders on top and on the bottom,
  # the text is centered and is bold

headSty <- createStyle(fgFill = "#DCE6F1",
                       border = "TopBottom",
                       halign = "center",
                       textDecoration = "bold")

# Data preparation
senso_mean <- sensory %>%
  group_by(Product) %>%
  summarise(across(where(is.numeric), mean)) %>%
  tibble::column_to_rownames(var = "Product")

overall_mean <- apply(senso_mean, 2, mean)

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

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

[1] 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.

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

# Write table: note that
writeDataTable(wb,
               sheet = "Mean",
               x = senso_mean,
               startCol = 1,
               startRow = 1,
               colNames = TRUE, rowNames = TRUE,
               tableStyle = "TableStyleLight9")

# Freezing the table
freezePane(wb, sheet = "Mean", firstRow = TRUE, firstCol = TRUE)

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

# Adding formatting to the second column
conditionalFormatting(wb,
                      sheet = "Mean",
                      cols  = 2,
                      rows  = 1 + 1:nrow(senso_mean),
                      rule  = paste0(">",overall_mean[2]),
                      style = pos_style)

conditionalFormatting(wb,
                      sheet = "Mean",
                      cols  = 2,
                      rows  = 1 + 1:nrow(senso_mean),
                      rule  = paste0("<",overall_mean[2]),
                      style = neg_style)

setColWidths(wb, sheet = "Mean",
             cols = 1:(1+ncol(senso_mean)), widths = 12)
```

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

TIPS: At any time, you can visualize your file using `openXL()`: This function opens within Excel the temporary file that is currently being built, hence allowing you to double-check that your code matches your expectations.

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

5.3 PowerPoint

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

With `{officer}`, the procedure starts with creating a powerpoint object `pptx_obj` using the `read_pptx()` function, to which we add slides through the `add_slide()` function. On a given worksheet, any type of content (text, graph, table) can be exported using `ph_with()` (ph ~ placeholder) which controls where to write the content through the `location` and `ph_location` options.

Through these different functions, many additional formatting procedure can be applied.

- `add_slide()` with arguments:
  - `layout` type of slide;
  - `master` theme of the deck;

5.3.1 Creating a PowerPoint Deck

Key functions: `read_pptx(path)` `add_slide(x, layout, master)` `layout_summary(x)`

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

class(pptx_obj)

pptx_obj %>%
  layout_summary()

pptx_obj %>%
  print(target = file.path("output","example_1.pptx"))
```

5.3.2 PowerPoint Themes

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.

Loading a template with a custom theme

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

pptx_obj %>%
  layout_summary()

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

5.3.3 Placeholders and Shapes

- `ph_location_type()` allows use predefined placeholders for content:
  - title, body
  - ctrTitle, subTitle, dt, ftr, sldNum

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

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

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

pptx_obj <- pptx_obj %>%
  on_slide(index = 1) # set active slide

slide_summary(pptx_obj) # technical content summary

my_data <- head(mtcars)[,1:4]

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

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

pptx_obj %>%
  print(target = file.path("output","example_2.pptx"))
```

- `ph_location()` allows to specify exact positions for content
  - for left/top/width/height units are inches

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

my_data <- "My text"

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

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

pptx_obj %>%
  print(target = file.path("output","example_3.pptx"))
```

5.3.4 Working with Text

Each new text item added to a PowerPoint via officer is a paragraph object

`fpar()` ("formatted paragraph") creates this object

`block_list()` allows us to wrap multiple paragraphs together

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

```
my_prop <- fp_text(color = "red", font.size = 14)
my_text <- ftext("First Line in Red", prop = my_prop)

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

my_par2 <- fpar("Second Line") # unformatted
my_list <- block_list(my_par, blank_line, my_par2)

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

pptx_obj %>%
  print(target = file.path("output","example_4.pptx"))
```

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

5.4 Tables

`ph_with` accepts a data.frame as value and renders it in a default format

```
ft_data <- senso_mean %>%
  dplyr::select(Salty, Sweet, Sour, Bitter) %>%
  tibble::rownames_to_column() %>%
  rename(Product = rowname) %>%
  mutate(across(Salty:Bitter, as.numeric)) %>%
  bind_rows(summarise(.,
                      across(where(is.numeric), mean),
                      across(where(is.character), ~"Average"))) %>%
  mutate(across(where(is.numeric), round, 2))

pptx_obj <- read_pptx() %>%
  add_slide(layout = "Title and Content", master = "Office Theme") %>%
  ph_with(value = ft_data, location = ph_location_type(type = "body")) %>%
  print(target = file.path("output","table_1.pptx"))
```

5.4.1 Introduction to flextable

With `{flextable}`, the procedure starts with creating a flextable object `ft_table` using the `flextable()` function. Flextable objects are compatible with `{officer}` and therefore are our primary tool for table formatting. Through different functions, many custom formatting procedure can be applied:

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

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

```
library(flextable)

# Create a flextable object
ft_table <- ft_data %>%
  flextable()

# Flextable objects can be found in the Viewer tab of Rstudio
print(ft_table)
```

5.4.2 Formatting examples

```
ft_table <- ft_table %>%
  autofit() %>% # column width
  # alignment of header: we use part argument
  align(align = "center", part = "header") %>%
  # alignment of content: we can use part = "body" or specify exact lines
  align(i = 1:nrow(ft_data), j = 1:ncol(ft_data), align = "center")

print(ft_table)
```

Set font names, sizes and colors

```
ft_table <- ft_table %>%
  # main formatting
  fontsize(size = 11) %>%
  font(fontname = "Calibri") %>% # since no i or j are input, change is for all data
  font(fontname = "Roboto", part = "header") %>% #different font for header
  color(color = "white", part = "header") %>%
  bold(part = "header") %>%
  # format last row
  bold(i = nrow(ft_data), j = 1:ncol(ft_data)) %>% #
  italic(i = nrow(ft_data), j = ~Product + Salty + Sweet + Sour + Bitter) %>% # using
  color(i =  nrow(ft_data), j = ~Sour, color = "red") %>%
  color(i =  nrow(ft_data), j = ~Sweet, color = "orange") %>%
  # background colors
  bg(bg = "#324C63", part = "header") %>% # a custom background for the header
  bg(i = 1:nrow(ft_data), bg = "#EDEDED") # a custom background for some cells

print(ft_table)
```

Set borders and adjust cells heights and widths

```
#BORDERS
# For borders we need to use nested functions (similar to fpar>ftext>fp_text)
# fp_border() is the second level function we will use to specify border"s characteris
# as argument it takes color, style, and width
my_border <- officer::fp_border(color = "black", style = "solid", width = 1)

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

```
  # all measurements are in inches
  width(j = 1, width = 1.2) %>% # column 1 wider
  height(i = 12, height = 1) # last row's height

print(ft_table)
```

5.4.3 Add flextable object to a powerpoint slide

```
# Add table to slide
pptx_obj <- read_pptx() %>%
  add_slide(layout = "Title and Content", master = "Office Theme") %>%
  ph_with(value = ft_table, ph_location(left = 2, top = 2, width = 4)) %>%
  print(target = file.path("output","table_2.pptx"))
```

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

5.5 Charts

5.5.1 Adding charts as images

```
# Using ggplot2 as plotting library
chart_to_plot <- senso_mean %>%
  dplyr::select(Salty, Sweet, Sour, Bitter) %>%
  tibble::rownames_to_column() %>%
  rename(Product = rowname) %>%
  pivot_longer(cols = Salty:Bitter, names_to = 'Attribute', values_to = 'Value') %>%
  ggplot(aes(x = Product, y = Value, fill = Attribute)) +
  geom_col(position = 'dodge')

print(chart_to_plot) #in Plots window of Rstudio
```

5.5.2 rvg package

rvg is providing two graphics devices that produces Vector Graphics outputs in `DrawingML` format for Microsoft PowerPoint with dml_pptx and for Microsoft Excel with dml_xlsx.

```
# body location
library(rvg)

# all items on the chart inside the pptx can be editable
pptx_obj <- read_pptx() %>%
  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(target = file.path("output","rvg_1.pptx"))

#custom location, all units are in inches
pptx_obj <- read_pptx() %>%
  add_slide(layout = "Title and Content", master = "Office Theme") %>%
  ph_with(value = dml(ggobj = chart_to_plot, editable = FALSE),
          location =  ph_location(left = 1, top = 1, width = 8, height = 6))  %>%
  print(target = file.path("output","rvg_2.pptx"))
```

5.5.3 mschart package

```
library(mschart)
# sample dataframe
mydata <- senso_mean %>%
  dplyr::select(Salty, Sweet, Sour, Bitter) %>%
  tibble::rownames_to_column() %>%
  rename(Product = rowname) %>%
  pivot_longer(cols = Salty:Bitter, names_to = 'Attribute', values_to = 'Value')

# syntaxis is similar to ggplot2's aes() with x,y,group
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(my_barchart, preview = TRUE)
#the command will work on machines with a Powerpoint Viewer

# to add the object to a powerpoint slide we can use the officer's native ph_with
pptx_obj <- read_pptx() %>%
  add_slide(layout = "Title and Content", master = "Office Theme") %>%
  ph_with(value = my_barchart, location = ph_location_type(type = "body")) %>%
  print(target = file.path("output","rvg_2.pptx"))
```

5.6 Word

Word documents are created using `read_docx()`.

`body_add_par()` to add a text paragraph. Paragraphs are automatically separated by line breaks.

```
my_doc <- read_docx() %>%
  body_add_par(value = "My Text", style = "Normal") %>%
  body_add_par(value = "Other Text", style = "Normal") %>%
  body_add_par(value = "Conclusion", style = "Normal") %>%
  print(target = file.path("output","doc_1.docx"))
```

Unlike a pptx with separate slides, a word document is a continuous object. To create a new page and continue to write on it `body_add_break()` is used.

```
my_doc <- read_docx() %>%
  body_add_par(value = "My Text", style = "Normal") %>%
  body_add_break() %>%
  body_add_par(value = "Conclusion", style = "Normal") %>%
  print(target = file.path("output","doc_2.docx"))
```

`body_add_fpar()` to add a formatted text paragraph `body_add_table()` to add a table

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

doc <- read_docx() %>%
  body_add_par(value = "Document Title", style = "heading 1") %>%
  body_add_par(value = "", style = "Normal") %>%
  body_add_fpar(my_par, style = "Normal") %>% #formatted paragraph function
  body_add_par(value = "", style = "Normal") %>%
  body_add_table(value = head(mtcars)[, 1:4], style = "table_template" ) %>%
  print(target = file.path("output","doc_3.docx"))
```

Bon Appétit

Chapter 6

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