Data Science for Sensory and Consumer Scientists

John Ennis, Julien Delarue, and Thierry Worch

2021-05-06

Contents

Preface

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

Data Science for
Sensory and Consumer Scientists

Who Should Read This Book?

How to Use This Book

Cautions: Don't that Everybody Does

How to Contact Us

Acknowledgements

Chapter 1

Bienvenue!

Why Data Science for Sensory and Consumer Science?

One of the most exciting aspects of being a sensory and consumer scientist is having access to a wide range of data.

Core principles in Sensory and Consumer Science

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

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 Wickham and Grolemund (2016) 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 (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."

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.

2.2 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.

2.2.1 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

[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.

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 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 (see Chapter 17).
- **Shared Learning** Once a team combines standardized reporting with tools for online collaboration such as GitHub (see Appendix A.4), 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 2.1. Each section is described in greater detail below.

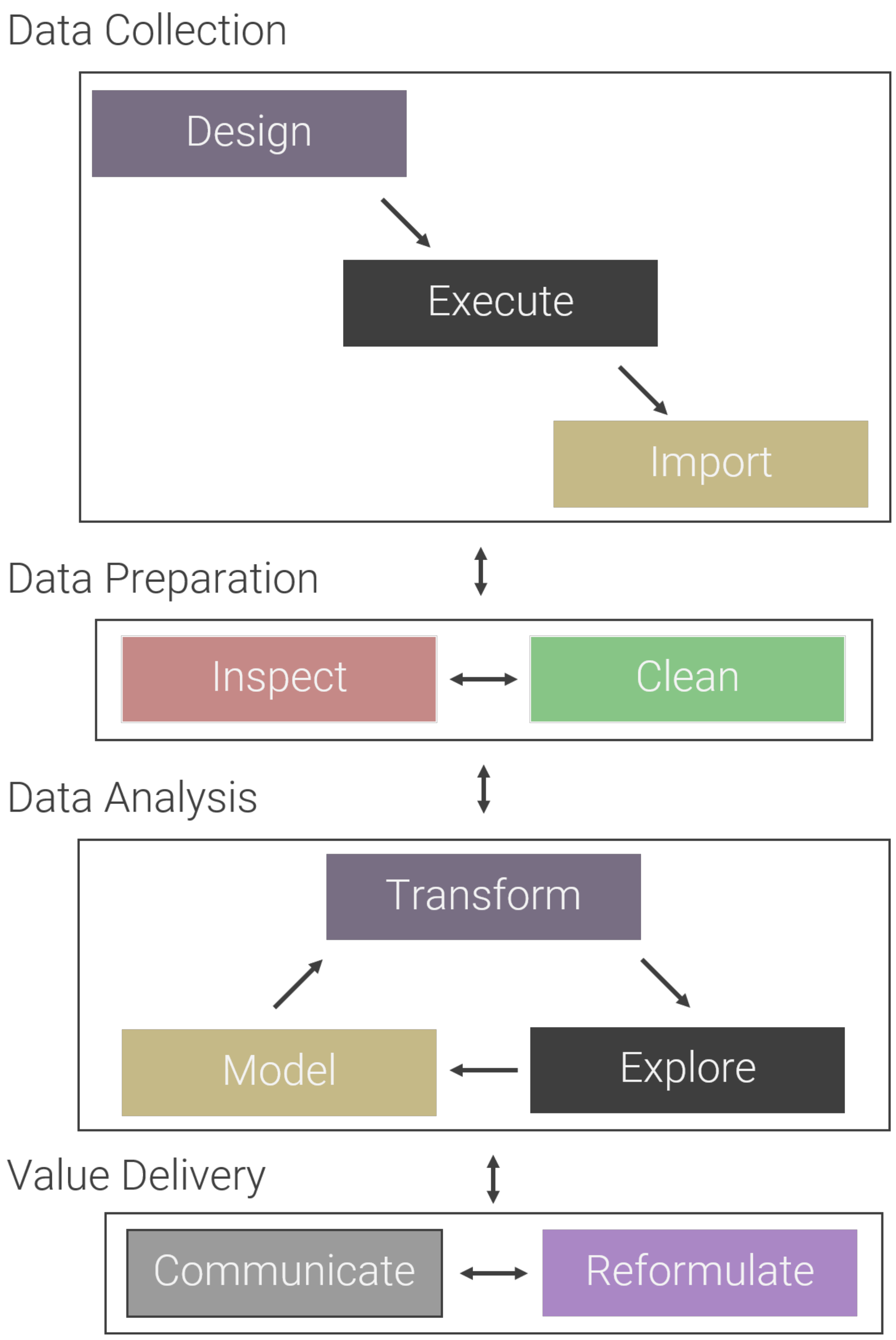

Figure 2.1: Data scientific workflow.

2.3.1 Data Collection

2.3.1.1 Design

Experimental design is an essential part of any scientific investigation. Whether the data

2.3.1.2 Execute

2.3.1.3 Import

2.3.2 Data Preparation

2.3.2.1 Inspect

Goal: Gain familiarity with the data Key Steps: Learn collection details Check data imported correctly Determine data types Ascertain consistency and validity Tabulate and compute other basic summary statistics Create basic plots of key variables of interest

2.3.2.2 Clean

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

2.3.3 Data Analysis

2.3.3.1 Transform

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

2.3.3.2 Explore

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

2.3.3.3 Model

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

2.3.4 Value Delivery

2.3.4.1 Communicate

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

2.3.4.2 Reformulate

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

2.4 Reproducible Research

Discuss benefits

- Time savings
- Collaboration
- Continuous improvement

2.5 How to Learn Data Science

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

We recommend following the instructions in Appendix A to get started.

Chapter 3

Tidy Thinking

This is a test of the footnotes[1].

[1] A great footnote

Chapter 4

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 been liked differently, an ANOVA is performed on a long thin table with (PxC) rows x 3 columns (consumer, sample, and the liking scores).

However, to assess whether consumers have the same preference patterns at the 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 hence essential to learn to manipulate data and transition from one structure to another. After presenting many different ways to transform your data, application through simple examples will be presented[1].

4.1 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

[1]Most of the examples presented in this chapter have no scientific meaning. This has been done on purpose to emphasize the "how to?", not the "why?"

on the statistical unit to consider, and the analyses to perform, data hence need to be manipulated.

4.1.1 Simple Manipulations

The notion of 'simple manipulations' proposed here is completely arbitrary and consists in data transformation that could easily be performed in other software such as Excel (although we strongly recommend performing any sorts of transformation in R)

4.1.1.1 Handling Columns

4.1.1.1.1 Renaming Variables The first simple transformation that one could consider consists in renaming one or multiple variables. This procedure can easily be done using the `rename()` function from the `{dplyr}` package.

In our sensory file, let's recode 'Judge' into 'Panellist', and 'Product' into 'Sample' (here, we apply transformations without saving the results, so without altering the original dataset):

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

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:

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

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.

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

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

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

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

Last but not least (regarding columns transformation) the `select()` function (from the `{dplyr}` package[2]) 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)
```

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 Flavour, 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 'Qtt':

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

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

[2]Many other packages include a function called `select()`, hence creating 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.

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 complex variables' selection 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("Qtt"))
```

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

4.1.1.1.3 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))
```

4.1.1.1.4 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="-")
```

4.1.1.2 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.

4.1.1.2.1 Re-arranging Rows The first step of re-arranging rows is done through the `arrange()` function from the `{dplyr}` package. This function allows sorting the data in the ascending order[3]. 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))
```

4.1.1.2.2 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")`);

[3]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.

- 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))

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.

4.1.1.2.3 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.

4.1.2 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.

4.1.2.1 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
```

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.

4.1.2.2 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.

4.1.3 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).

4.1.3.1 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.

4.1.3.2 Illustrations of Data Manipulation

Let's review the different transformations presented earlier by generating the sensory profiles of the samples through different approaches[4].

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))
```

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.

[4]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...

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)
```

4.1.4 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 8.3.3.

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 5

Data Visualization

5.1 Design Principles

5.2 Table Making

5.3 Chart Making

5.3.1 Philosophy of ggplot2

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

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

5.3.1.1 aesthetics

Provide the most relevant options for `aes()`

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

5.3.1.2 geom_

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

5.3.2 Common Charts

5.3.2.1 Scatter points

`geom_point()`

5.3.2.2 Line charts

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

5.3.2.3 Bar charts

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

5.3.2.4 Distribution

`geom_boxplot()` and `geom_violin()`

5.3.2.5 Text

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

5.3.2.6 Rectangles

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

5.3.2.7 Themes and legend

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

5.3.3 Additional Topics

5.3.3.1 Playing around with axes

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

5.3.3.2 Transposing the plot

`coord_flip()` and `coord_polar()`

5.3.3.3 Splitting plots

`facet_wrap()`, `facet_grid()`

5.3.3.4 Combining plots

{patchwork}

Chapter 6

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.

6.1 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

6.2 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.

```r
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/.

6.3 PowerPoint

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

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

6.3.1 PowerPoint Formatting

PowerPoint Slide Master

6.3.1.1 Importing the Template

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

class(my_file) # checking if correct class

my_file %>%
  layout_summary()
```

6.3.1.2 Creating a PowerPoint Deck

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

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

We cannot load the themes of Office *ex nihilo* returns error

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

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

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

We can even load a template with more than one theme

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

6.3.1.3 Selection Pane

6.3.1.4 Key functions:

`read_pptx(path)` `layout_summary(x)` `layout_properties(x)`

`add_slide(x, layout, master)` `on_slide(x, index)` `slide_summary(x, index)`

6.3.1.5 Example Code

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

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

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

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

6.3.2 Placeholders

6.3.2.1 Placeholders and Shapes

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

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

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

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

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

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

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

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

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

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

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

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

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

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

6.3.2.2 Key functions: ph_with()

6.3.3 Text

6.3.3.1 Working with Text

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

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

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

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

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

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

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

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

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

my_list <- block_list(my_data, blank_line, my_data2)

my_type <- "body"

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

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

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

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

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

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

pptx_obj <- read_pptx()

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

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

6.3.3.2 Key functions:

`fpar()` `ftext()` `fp_text()` `block_list()`

6.3.3.3 Example Code

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

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

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

6.3.4 Tables

6.3.4.1 Basic Code

```
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 = "output/test4.1.pptx")
```

6.3.4.2 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:

6.3.4.3 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)
```

6.3.4.4 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 characteristics
# 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

ft_table
```

6.3.4.5 Demonstration Output

6.3.4.6 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 = "output/test4.7.pptx")
# positions are fixed. We can input exact positions to center the table
```

6.3.5 Charts

6.3.5.1 Adding charts as images

```
# We have preloaded a function to plot the chart.
# the function is using ggplot2 as plotting library
chart_to_plot <- sample_data_list[['Sample 1']] %>%
  make_jar_chart() # code to create a ggplot2 item, we will skip the contents
print(chart_to_plot) # see in Plots Window
```

6.3.5.2 rvg Example

```
# the output is a ggplot2 object

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

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

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

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

pptx_obj <- read_pptx() %>%
  add_slide(layout = "Title and Content", master = "Office Theme") %>%
  ph_with(value = dml(ggobj = chart_to_plot), location =  ph_location(left = 1, top = 
  print(target2 = "output/5.2.rvg2.pptx")

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

6.3.6 mschart Package

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

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

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

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

6.4 Word

6.4.1 Word documents

Formats share similarities

`body_add_*()`

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

`body_add_break()`

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

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

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

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

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. 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

7.1 Products

7.2 Participants

7.3 Design of the consumer test

7.4 Sensory descriptive analysis data

7.5 Summary of the datasets

Chapter 8

Data Collection

8.1 Design

8.2 Execute

8.3 Import

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

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

For other (less common) formats, the reader can find packages that would allow importing their files into R. Particular interest can be given to the package `{rio}` (*rio* stands for *R* *I*nput and *O*utput) which provides an easy solution that

1. can handle a large variety of files,
2. can actually guess the type of file it is,
3. provides tools to import, export, and convert almost any type of data format, including .csv, .xls and .xlsx, or data from other statistical software such as SAS (.sas7bdat and .xpt), SPSS (.sav and .por), or Stata (.dta).

As an alternative, the package `{foreign}` provides functions that allow importing data stored from other statistical software (incl. Minitab, S, SAS, Stata, SPSS, etc.).

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

8.3.1 Importing Structured Excel File

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

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

- The first row within the *Data* sheet of *Sensory Profile.xlsx* contains the headers,
- From the second row 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 dataset thus imported is a `tibble`. Let's take advantage of the printing properties of a `tibble` to evaluate `sensory`:

```
sensory
```

`sensory` is a tibble with 99 rows and 35 columns that includes the `Judge` information (first column, defined as character), the `Product` information (second 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 dataset is not so well organized/structured, and may need to be *decoded*. This is the case for the workbook entitled *TFEQ.xlsx*. For this file:

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

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

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

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

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

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

var_labels
```

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

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

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 performed multiple times over time, the same test has been performed simultaneously in two 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 1. it is very tedious when a larger number of sheets is involved, and 2. it is not automated since the code will no longer run if (say) 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 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")
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 (here session) 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}`:

```
path %>%
  excel_sheets() %>%
  set_names(.) %>%
  map(~read_excel(path, sheet = .)) %>%
  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!

Remarks:

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 datasets are stored in different excel files (rather than different sheets within a file), we could apply a very similar procedure by using `list.files()` from the `{base}` package, together with `pattern = "xlsx"` to limit the search to Excel files.

Chapter 9

Data Preparation

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

9.1 Inspect

9.1.1 Data Inspection

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

```
TFEQ_data
```

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

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

9.1.2 Design Inspection

Evaluate if the design is complete/incomplete Frequencies and cross-frequencies (simple statistics and simple graphs)

9.1.3 Missing Data Inspection

Are there NAs? If yes, are they structured of random?

9.2 Clean

9.2.1 Renaming Variables

renaming columns using `rename()` or `select()`

9.2.2 Handling Data Type

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

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

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

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

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

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

9.2.2.1 Numerical Data

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

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

9.2.2.2 Binary Data

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

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

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

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

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

9.2.2.3 Nominal Data

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

As can be seen, the variable `Area` is the factor version (including its labels) of `Living area`. If we would also consider that `Rurban` should be combined with

`Rural`, and that `Rural` should appear before `Urban`, we can simply modify the code as such:

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

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

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

This approach of renaming and re-ordering factor levels is very important as it simplifies the readability of tables and figures. Some other transformations can be applied to factors thanks to the `{forcats}` package. Particular attention can be given to the following functions:

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

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

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

9.2.3 Converting between Types

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

9.2.3.1 Conditional Renaming?

`mutate()` and `ifelse()`

9.2.4 Handling Missing Values

Ignoring, removing, imputing

9.2.5 Restructuring Data

Presentation of the different shapes of the tables based on objectives

9.2.5.1 Variables Selection and Repositioning

`dplyr::select()` and `dplyr::arrange()`

9.2.5.2 Data Filtering

`dplyr::filter()`

9.2.5.3 Data (Re)Shaping

`pivot_wider()` and `pivot_longer()` `_join()`

9.2.5.4 Preparing Data for FactoMineR and SensoMineR

matrix, data frame, and tibble.

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

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

Chapter 10

Data Analysis

In this chapter, the different techniques presented in 4 will be applied to our biscuit data. 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.

10.1 Sensory Data

Let's start with the analysis of our sensory data. If not done already, import the file *Sensory Profile.xlsx*. To do so, we need the `{here}` and `{readxl}` packages. Since most of the analyses we are going to perform require tools from the `{tidyverse}`, we also load this group of packages.

```r
library(tidyverse)
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)
```

The first analysis often done consists in evaluate 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 `Shiny`, the code would look like this (here we consider the 2-way ANOVA evaluating the Product and Assessor effects):

```
shiny_aov <- lm(Shiny ~ Product + Judge, data=sensory)
anova(shiny_aov)
```

We could duplicate this code for each single attributes, but 1. this would be quite tedious if the number of attributes increase, 2. it is sensitive to the way the variables are being called, meaning that 3. it would usually not be applicable to other dataset. 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 dataset should be in the long and short 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 for each sub-data the ANOVA (the model is then defined as `Score ~ Product + Judge`), and we extract the results of interest using the `{broom}` package[1] 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)
```

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 create a new variable called `mod` which takes the results of the ANOVA. Once the analysis is done, we summarize the info stored in `mod` by converting into a tibble using `{broom}`:

[1]The `{broom}` package is a very useful package that convert statistical objects into tidy tables.

```
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 checking for which attributes there are no 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 decreasingly the `p.value`:

```
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), `Astrigent` (p=0.116), and `Sticky` (p=0.101).

To visualize graphically the results of the ANOVA, a barchart representing the F-value can be generated. We recommend to order the attributes based on their decreasing F-values, and by colour-coding them based on their 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 Attriubtes","(The attributes are sorted according to the F-values)")+
  theme_bw()+
  xlab("")+
  ylab("F-values")+
  coord_flip()
```

It appears that the different 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. Such visualization is often done through Principal Component Analysis (PCA).

In practice, PCA is performed on the mean table crossing the products in rows, and the sensory attributes in columns. In 4, many different approaches to generate such table have been proposed:

```
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 can be submitted to PCA. To do so, the `PCA()` function from the `{FactoMineR}` is used. Few important remarks: The `PCA()` function usually contains only numerical data, the product names being often set as row names in a data frame. This is something that can easily be taken care of through the `column_to_rownames()` function from the `{tibble}` package. Additionally, one might notice that we still have two qualitative variables: `Protein` and `Fiber`. These 2 variables can either be removed prior to running the analysis, or better, projected as supplementary variables through the `quali.sup` parameter from `PCA()`.

```
library(FactoMineR)

senso_pca <- senso_mean %>%
  as.data.frame() %>%
  column_to_rownames(var="Product") %>%
  PCA(., quali.sup=1:2, graph=TRUE)
```

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 any plot from `{FactoMineR}` (and more) as a `ggplot()` object. for `PCA()`, the main function used to re-create the plots are `fviz_pca_ind()` for the products' map, `fviz_pca_var()` for the variables' representation, and `fviz_pca_biplot()` for the biplot combining both. Since we have two qualitative variables, let's take advantage of them by coloring for each of them the products based on their 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)
```

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`.

10.2 Demographic and TFEQ Data

In the *TFEQ.xlsx* file, descriptive (e.g. demographic) and behavioral (e.g. TFEQ) information regarding the consumers is being stored.

A quick look at this file shows that it contains three tabs denoted as *Data*, *Variables*, and *Levels*:

- *Data* contains the data, 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 step as following:

```
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$Name))
```

10.2.1 Demographic Data: Frequency and Proportion

For this demographic data file, we are interested to see the partition of consumers for each of the descriptive variables. More precisely, we would like to know the frequency and proportion (in percentage) attached to each level of `Living area`, `Housing`, `Income range`, and `Occupation`.

To obtain such table, let's start by only selecting these particular columns together with `Judge`. Since the data is coded, let's decode it using the information stored in `demo_lev`: we could automatically recode each variable using `mutate()` and `factor()`. However, we prefer here to use `pivot_longer()` to create a long thin tibble, and then to merge this table with `demo_lev` by `Question` and `Response` using `inner_join()` instead. 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 (`N/sum(N)`)[2].

```
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()
```

To better visualize these different proportions, histograms are often considered. Such histograms can be obtained by splitting `demog_reduced` by `Question` and by creating in each case an histogram using either `N` or `Pct` (we are using `Pct`)[3]. Of course, we automate this across all questions using `map()`.

```
demog_reduced %>%
  split(.$Question) %>%
  map(function(data){

    var = as.character(unlist(data$Question))

    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")+
      xlab("")+
```

[2]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.

[3]Here, the histograms are ordered decreasingly (`reorder`) and are represented horizontally (`coord_flip()`).

```
      ylab("")+
      ggtitle(var)+
      theme_bw()+
      coord_flip()

  })
```

10.2.2 Behavioral Data: TFEQ Traits

In the same data set, consumers also answered some behavioral questions that reflects on their relation to food. These questions can be categorized into three groups: Disinhibitor (variables starting with `D`), Restriction (variables starting with `R`), and Hunger (variables starting with `H`).

Let's start by selecting all these variables. We could use `select()` combined with `starts_with("D")`, `starts_with("R")`, and `starts_with("H")`, but this solution is not satisfying since it can also seelct other variables that would start with any of these letters (e.g. Housing). Since the variables are all well structured (the all start with the letters D, R, or H, followed by a number), the use of a regular expression is more appropriate.

In practice, generating regular expression can be quite complex as it is an abstract concept which follow very specific rules. Fortunately, the package `{RVerbalExpression}` is a great assistant as it generates the regular expression for you thanks to understandable functions. First, we initiate the regular expression by calling the function `rx()` to which we add that we first want any variables that 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 that we need. We can then `dplyr::select()` any variable that fits our regular expression by using the function `matches()`.

```
demographic %>%
  dplyr::select(matches(rdh))
```

Chapter 11

Value Delivery

11.1 Communicate

11.1.1 Know Your Audience

11.1.2 Pick the Correct Format

11.1.3 Storytelling

11.2 Reformulate

Haute Cuisine

Chapter 12

Machine Learning

12.1 Overview

12.2 Key Topics

12.2.1 Model Validation

12.2.2 Unsupervised learning

12.2.2.1 Cluster analysis

12.2.2.2 Factor analysis

12.2.2.3 Principle components analysis

12.2.2.4 t-SNE

12.2.3 Semisupervised learning

12.2.3.1 PLS regression

12.2.3.2 Cluster Characterization

12.2.4 Supervised learning

12.2.4.1 Regression

12.2.4.2 K-nearest neighbors

12.2.4.3 Decision trees

12.2.4.4 Black boxes

12.2.4.4.1 Random forests

12.2.4.4.2 SVMs

12.2.4.4.3 Neural networks

12.2.4.4.4 Computer vision

12.2.5 Interpretability

12.2.5.1 LIME

12.2.5.2 DALEX

12.2.5.3 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 Examples

12.4.1 Data Prep

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

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

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

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

12.4.2 Classification Code

```
library(tidyverse)
library(tidymodels)


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

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

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

summary(data)

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

skimr::skim(data)

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



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

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

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

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


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

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

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

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

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

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

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


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

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

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

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

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

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

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


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

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

tree_spec <-
  decision_tree(
    cost_complexity = tune(),
    tree_depth = tune(),
    min_n = tune()
    ) %>%
  set_mode("classification") %>%
  set_engine("rpart")

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

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

```r
# Check model performance -------------------------------------------------

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

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

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

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

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

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


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

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

final_fit %>% collect_metrics()

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

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

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

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

write_rds(final_model, 'classification_model.rds')


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

model <- read_rds('classification_model.rds')

new_observation <- tibble(
  N1 = 1.8,
  D1 = factor(0),
  D2 = factor(0),
  D3 = factor(1),
  D4 = factor(0),
  D5 = factor(1),
  D6 = factor(0),
  D7 = factor(1),
  D8 = factor(1),
  D9 = factor(1),
  D10 = factor(1),
  D11 = factor(0)
)

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

12.4.3 Regression Code

```
library(tidyverse)
library(tidymodels)
```

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

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

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

summary(data)

skimr::skim(data)

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

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

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

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


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

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

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

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

glmnet_workflow <-
```

```
  workflow() %>%
  add_recipe(glmnet_recipe) %>%
  add_model(glmnet_spec)

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

final_fit %>% collect_metrics()

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

```r
  geom_qq() +
  geom_qq_line()

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

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

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

write_rds(final_model, 'regression_model.rds')

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

model <- read_rds('regression_model.rds')

new_observations <- data[1:2,]
new_observations

predict(model, new_observations)
```

Chapter 13

Text Analysis

13.1 Overview

13.2 Key Topics

13.2.1 Data Sources

13.2.2 Working with Strings

13.2.3 Tokenizing

13.2.4 Lemmatization, stemming, and stop word removal

13.2.5 Part of Speech Tagging

13.3 Common Applications

13.3.1 Frequency counts and summary statistics

13.3.2 Word clouds

13.3.3 Contrast plots

13.3.4 Sentiment analysis

13.3.5 Topic Modeling

13.3.6 Bigrams and word graphs

13.4 Code Examples

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

13.4.1 Statistical entities

What are we considering as statistical entities?

- documents
- sentences
- words
- cleaned words

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

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

Discussion around CATA as a simplified version of text analysis…

13.4.1.1 Notion of tokenization

13.4.1.2 Cleaning the data

Notions of lemmatization, stemming, and stopwords removal

- grouping words
- removing stopwords
- tf-idf

13.4.2 Analysis of Frequencies and term-frequency document

13.4.2.1 Contingency table

Presentation of the tf/contingency table

13.4.2.2 wordclouds

{ggwordclouds}

13.4.2.3 Correspondence Analysis

{FactoMineR} and **{Xplortext}**

13.4.3 Futher Analysis of the words

13.4.3.1 Sentiment Analysis

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

13.4.3.2 Bi-grams and N-grams

Presentation of graph-theory applied to text mining

13.4.3.3 Machine learning

Introduction to machine learning associated to text mining

Chapter 14

Linear Programming

14.1 Overview

14.2 Common Applications

14.2.1 Experimental Design

14.2.2 Sample Selection

14.2.3 Compact Letter Displays

14.2.4 TURF Analysis

14.2.5 Bundle Optimization

14.3 Code Examples

Chapter 15

Pipelines

Chapter 16

Dashboards

Chapter 17

Graph Databases

Appendix A

Apéritifs (Getting Started)

A.1 R

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

A.1.1 Why Learn a Programming Language?

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

A.1.2 Why R?

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

A.2 Why R?

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

A.2.1 Steps to Install R

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

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

A.3 RStudio

A.3.1 Steps to Install RStudio

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

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

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

```
x + y
```

Some recommendations upon installing RStudio:

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

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

A.3.2 Create a Local Project

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

A.3.3 Install and Load Packages

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

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

A.3.4 Run Sample Code

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

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

```
library(sensR)

num_correct <- 15
num_total <- 44

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

print(discrim_res)
```

A.4 Git and GitHub

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

A.4.1 Git

- Install Git
  - Windows
  - macOS
- Register with RStudio

A.4.2 GitHub

- Create a GitHub account
- Register with RStudio

A.5 RAW MATERIAL

A.5.1 Principles

A.5.2 Tools

A.5.2.1 GitHub

A.5.2.2 R scripts

A.5.2.3 RMarkdown

A.5.2.4 Shiny

A.5.3 Documentation

A.5.4 Version control

A.5.5 Online repositories for team collaboration

A.5.6 Building a code base

A.5.6.1 Internal functions

A.5.6.2 Packages

Appendix B

Digestifs (Next Steps)

B.1 Sensory Analysis in R

B.2 Other Recommended Resources

B.2.1 R for Data Science

B.2.2 Hands-On Machine Learning with R

B.2.3 FactoExtra

B.2.4 R Graphics Cookbook

B.2.5 Storytelling with Data

B.2.6 Text Mining wtih R

B.3 Python and Other Languages

Bibliography

Bibliography

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.

Fayyad, U., Piatetsky-Shapiro, G., and Smyth, P. (1996). From data mining to knowledge discovery in databases. *AI magazine*, 17(3):37.

Naur, P. (1974). *Concise survey of computer methods*. Petrocelli Books.

Peng, R. D. (2011). Reproducible research in computational science. *Science*, 334(6060):1226–1227.

Tukey, J. W. (1962). The future of data analysis. *The annals of mathematical statistics*, 33(1):1–67.

Wickham, H. and Grolemund, G. (2016). *R for data science: import, tidy, transform, visualize, and model data.* " O'Reilly Media, Inc.".

Wu, C. F. J. (1997). Statistics = Data Science?