An R Approach to Data Cleaning and Wrangling for Education Research

Juho Kopra, Santtu Tikka, Merja Heinäniemi, Sonsoles López-Pernas, and Mohammed Saqr

1 Introduction

When analyzing data, it is crucial that the data is in a suitable format for the tools you will be using. This makes data wrangling essential. Data preparation and cleaning, such as extracting information from raw data or removing erroneous measurements, must also be done before data is ready for analysis. Data wrangling often takes up the majority of the time spent on analysis, sometimes up to 80%. To reduce the amount of work required, it is beneficial to use tools that follow the same design paradigm to minimize the time spent on data wrangling. The `tidyverse` [1] programming paradigm is currently the most popular approach for this in R.

The `tidyverse` has several advantages that make it preferable over other alternatives such as simply using base R for your data wrangling needs. All packages in the `tidyverse` follow a consistent syntax, making it intuitive to learn and use new `tidyverse` packages. This consistency also makes the code more easier to read, and maintain, and reduces the risk of errors. The `tidyverse` also has a vast range of readily available packages that are actively maintained, reducing the need for customized code for each new data wrangling task. Further, these packages integrate seamlessly with one another, facilitating a complete data analysis pipeline.

J. Kopra (✉) · S. López-Pernas · M. Saqr
School of Computing, University of Eastern Finland, Kuopio, Finland
e-mail: juho.kopra@uef.fi

S. Tikka
Department of Mathematics and Statistics, University of Jyväskylä, Finland

M. Heinäniemi
Institute of Biomedicine, University of Eastern Finland, Kuopio, Finland

M. Saqr, S. López-Pernas (eds.), *Learning Analytics Methods and Tutorials*,
https://doi.org/10.1007/978-3-031-54464-4_4

To fully realize the benefits of the `tidyverse` programming paradigm, one must first understand the key concepts of tidy data and pivoting. Tidy data follows three rules:

1. Each variable must have its own column.
2. Each observation must have its own row.
3. Each value must have its own cell.

Let's consider examples of tidy data. For instance, if you have data from a Moodle course where two attempts of an exam for each student are located in a single column. This example data violates the first rule, because there are two variables in a single column instead of separate columns for each variable. What is needed to make this data tidy is to pivot it to a longer format. The pivoted data would have two rows for each student, both of which are different observations (exam attempts 1 and 2). Thus the pivoted data would not conflict with second rule.

Data can also be too long, but in practice, this is much more rare. This can occur if two or more variables are stored on a single column across multiple rows. A key indicators of this is if the different rows of the same column have different measurements units (e.g. lb vs. kg). It may also occur that your raw data has multiple values in a single cell. In these cases, it is necessary to split the cells to extract the necessary information. In a simple case, where you have two values systematically in a one cell, the values can be easily separated into their own columns.

Overall, using `tidyverse` and understanding the key concepts of tidy data and pivoting can streamline the data analysis process and make code easier to work with and maintain. The rest of this chapter will guide readers through the process of data cleaning and wrangling with R in the field of learning analytics. We demonstrate how data can be grouped and summarized, how to select and transform variables of interest, and how data can be rearranged, reshaped and joined with other datasets. We will strongly rely on the `tidyverse` programming paradigm for a consistent and coherent approach to data manipulation, with a focus on tidy data.

2 Reading Data into R

Data files come in many formats, and getting your data ready for analysis can often be a daunting task. The `tidyverse` offers much better alternatives to base R functions for reading data, especially in terms of simplicity and speed when reading large files. Additionally, most of the file input functions in the `tidyverse` follow a similar syntax, meaning that the user does not have to master every function for reading every type of data individually.

Often, just before the data can be read into R, user must specify the location of data files by setting a working directory. Perhaps most useful way to do that is to create a project in RStudio and then create a folder called "data" within the project folder. Data files can be put into that folder and user can refer to those files just by telling R-functions relative path of data file (e.g. "data/Final%20Dataset.csv") while the project takes care of the rest of the path. A more traditional way of setting this, which also works without RStudio, is by using a command such as `setwd("/home/Projects/LAproject/data/Final%20Dataset.csv")`. Here, a function called `setwd()` is used to set up a folder into location mentioned in a character string given as its argument. A `getwd()` lists current working directory, which can also be seen in RStudio just above the Console output.

Some of the most common text data formats are comma-separated files or semicolon-separated files, both of which typically have the file extension .csv. These files can be read into R using the `readr` [2] package and the functions `read_csv()` and `read_csv2()`, respectively. For instance, we can read a comma-separated file R as follows

```r
library("readr")
url <- "https://github.com/lamethods/data/raw/main/2_moodleEdify/"
lms <- read_csv(paste0(url, "Final%20Dataset.csv"))
```

Functions in `readr` provide useful information about how the file was read into R, which can be used to assess if the input was successful and what assumptions about the data were made during the process. In the printout above, the `read_csv()` function tells us the number of rows and columns in the data and the column specification, i.e., what type of data is contained within each column. In this case, we have 17 columns with `character` type of data, and 4 columns of `double` type of data. Functions in `readr` try to guess the column specification automatically, but it can also be specified manually when using the function. For more information about this dataset, please refer to Chapter 2 in this book [3].

Data from Excel worksheets can be read into R using the `import()` function from the `rio` [4] package. We will use synthetic data generated based on a real blended course of learning analytics for the remainder of this chapter. These data consist of three Excel files which we will first read into R.

```r
library("rio")
url <- "https://github.com/lamethods/data/raw/main/1_moodleLAcourse/"
events <- import(paste0(url, "Events.xlsx"), setclass = "tibble")
results <- import(paste0(url, "Results.xlsx"), setclass = "tibble")
demographics <- import(paste0(url, "Demographics.xlsx"), setclass = "tibble")
```

The data files contain information on students' Moodle events, background information such as their name, study location and employment status, and various grades they've obtained during the course. For more information about the dataset, please refer to Chapter 2 in this book [3]. These data are read in the `tibble` [5] format, a special type of `data.frame` commonly used by `tidyverse` packages. We also load the `dplyr` [6] package which we will use for various tasks throughout this chapter.

```r
library("dplyr")
```

3 Grouping and Summarizing Data

Instead of individual-level data metrics, we may be interested in specific groups as specified by some combination of data values. For example, we could compute the number of students studying in each location by gender. To accomplish this, we need to start by creating a grouped dataset with the function `group_by()`. To compute the number of students, we can use the `summarise()` function which we already used previously in Chapter 1 and the function `count()`, which simply returns the number of observations in each category of its argument.

```
demographics |>
  group_by(Gender) |>
  count(Location)
```

```
# A tibble: 4 x 3
# Groups:   Gender [2]
  Gender Location      n
  <chr>  <chr>     <int>
1 F      On campus    55
2 F      Remote       10
3 M      On campus    51
4 M      Remote       14
```

The column `n` lists the number of students in each group. When a `tibble` that contains grouped data is printed into the console, the grouping variable and the number of groups will be displayed in the console below the dimensions. Next, we will compute the total number of Moodle events of each student, which we will also use in the subsequent sections.

```
events_summary <- events |>
  group_by(user) |>
  tally() |>
  rename(Frequency.Total = n)
events_summary
```

```
# A tibble: 130 x 2
  user      Frequency.Total
  <chr>               <int>
1 00a05cc62             417
2 042b07ba1             918
3 046c35846             199
4 05b604102             199
5 0604ff3d3             436
# i 125 more rows
```

Here, the function `tally()` simply counts the number of number rows in the data related to each student, reported in the column `n` which we rename to `Frequency.Total` with the `rename()` function. We could also count the number of events by event type for each student

```
events |>
  group_by(user, Action) |>
  count(Action)
```

```
# A tibble: 1,439 x 3
# Groups:   user, Action [1,439]
  user     Action           n
  <chr>    <chr>        <int>
1 00a05cc62 Applications     2
2 00a05cc62 Assignment     121
3 00a05cc62 Course_view    103
4 00a05cc62 Feedback         7
5 00a05cc62 General         10
# i 1,434 more rows
```

4 Selecting Variables

In the `tidyverse` paradigm, selecting columns, i.e., variables from data is done using the `select()` function. The `select()` function is very versatile, allowing the user to carry out selections ranging from simple selection of a single variable to highly complicated selections based on multiple criteria. The most basic selection selects only a single variable in the data based on its name. For example, we can select the employment statuses of students as follows

```
demographics |>
  select(Employment)
```

```
# A tibble: 130 x 1
  Employment
  <chr>
1 None
2 None
3 None
4 None
5 Part-time
# i 125 more rows
```

Note that using `select()` with a single variable is not the same as using the `$` symbol to select a variable, as the result is still a `tibble`, `select()` simply produces a subset of the data, where only the selected columns are present. Select is more similar to `subset()` in base R, which can accomplish similar tasks as `select()` and `filter()` in the `tidyverse`. However, we do not recommend using `subset()`, as it may not work correctly when the working environment has variables that have the same name as columns in the data, which can lead to undesired outcomes.

To extract the values of the selected column as a vector, we can use the function `pull()`. We use the `head()` function here to limit the console output to just the first few values of the vector (default is 6 values).

```
demographics |>
  pull(Employment) |>
  head()
```

```
[1] "None"      "None"      "None"      "None"      "Part-time" "Part-time"
```

The `select()` function syntax supports several operations that are similar to base R. We can select ranges of consecutive variables using `:`, complements using `!`, and combine selections using `c()`. The following selections illustrate some of these features:

```
demographics |>
  select(user:Origin)
```

```
# A tibble: 130 x 4
  user      Name   Surname  Origin
  <chr>     <chr>  <chr>    <chr>
1 6eba3ff82 Amanda Mora     Costa Rica
2 05b604102 Lian   Abdullah Yemen
3 111422ee7 Bekim  Krasniqi Kosovo
4 b4658c3a9 Yusuf  Kaya     Turkey
5 e6ec47f29 Zoran  Babić    Serbia
# i 125 more rows
```

```
demographics |>
  select(!Gender)
```

```
# A tibble: 130 x 7
  user      Name   Surname Origin     Birthdate Location  Employment
  <chr>     <chr>  <chr>   <chr>      <chr>     <chr>     <chr>
```

```
1 6eba3ff82 Amanda Mora     Costa Rica 28.2.1998  On campus None
2 05b604102 Lian   Abdullah Yemen      19.11.1996 On campus None
3 111422ee7 Bekim  Krasniqi Kosovo     30.1.1999  On campus None
4 b4658c3a9 Yusuf  Kaya     Turkey     16.6.1998  On campus None
5 e6ec47f29 Zoran  Babić    Serbia     29.10.1998 On campus Part-time
# i 125 more rows
```

```
demographics |>
  select(c(user, Surname))
```

```
# A tibble: 130 x 2
  user      Surname
  <chr>     <chr>
1 6eba3ff82 Mora
2 05b604102 Abdullah
3 111422ee7 Krasniqi
4 b4658c3a9 Kaya
5 e6ec47f29 Babić
# i 125 more rows
```

In the first selection, we select all variables starting from user on the left to Origin on the right. In the second, we select all variables except Gender. In the third, we select both user and Surname variables.

Sometimes, our selection might not be based directly on the variable names themselves as in the examples above but instead on vectors that contain the names of columns we wish to select. In such cases, we can use the function all_of(). We can consider the intersections or unions of such selections using & and |, respectively.

```
cols_a <- c("user", "Name", "Surname")
cols_b <- c("Surname", "Origin")
demographics |>
  select(all_of(cols_a))
```

```
# A tibble: 130 x 3
  user      Name   Surname
  <chr>     <chr>  <chr>
1 6eba3ff82 Amanda Mora
2 05b604102 Lian   Abdullah
3 111422ee7 Bekim  Krasniqi
4 b4658c3a9 Yusuf  Kaya
5 e6ec47f29 Zoran  Babić
# i 125 more rows
```

```
demographics |>
  select(all_of(cols_a) & all_of(cols_b))
```

```
# A tibble: 130 x 1
  Surname
  <chr>
1 Mora
2 Abdullah
3 Krasniqi
4 Kaya
5 Babić
# i 125 more rows
```

```
demographics |>
  select(all_of(cols_a) | all_of(cols_b))
```

```
# A tibble: 130 x 4
  user      Name   Surname  Origin
  <chr>     <chr>  <chr>    <chr>
1 6eba3ff82 Amanda Mora     Costa Rica
2 05b604102 Lian   Abdullah Yemen
3 111422ee7 Bekim  Krasniqi Kosovo
4 b4658c3a9 Yusuf  Kaya     Turkey
5 e6ec47f29 Zoran  Babić    Serbia
# i 125 more rows
```

Often the names of data variables follow a similar pattern, and these patterns can be used to construct selections. Selections based on a prefix or a suffix in the variable name can be carried out with the functions `starts_with()` and `ends_with()`, respectively. The function `contains()` is used to look for a specific substring in the names of the variables, and more complicated search patterns can be defined with the function `matches()` that uses regular expressions (see `?tidyselect::matches` for further information).

```
results |>
  select(starts_with("Grade"))
```

```
# A tibble: 130 x 13
  Grade.SNA_1 Grade.SNA_2 Grade.Review Grade.Group_self Grade.Group_All
        <dbl>       <dbl>        <dbl>            <dbl>           <dbl>
1           0           0         6.67                5               4
2           8          10         6.67                1               3
```

```
3           10          10         10                   10             9.11
4            5           5          0                    1             4   
5           10          10         10                   10             9.18
# i 125 more rows
# i 8 more variables: Grade.Excercises <dbl>, Grade.Project <dbl>,
#   Grade.Literature <dbl>, Grade.Data <dbl>, Grade.Introduction <dbl>,
#   Grade.Theory <dbl>, Grade.Ethics <dbl>, Grade.Critique <dbl>
```

```
results |>
  select(contains("Data"))
```

```
# A tibble: 130 x 1
  Grade.Data
       <dbl>
1          4
2          3
3          5
4          3
5          5
# i 125 more rows
```

So far, our selections have been based on variable names, but other conditions for selection are also feasible. The general-purpose helper function `where()` is used to select those variables for which a function provided to it returns `TRUE`. For example, we could select only those columns that contain `character` type data or `double` type data.

```
results |>
  select(where(is.character))
```

```
# A tibble: 130 x 1
  user
  <chr>
1 6eba3ff82
2 05b604102
3 111422ee7
4 b4658c3a9
5 e6ec47f29
# i 125 more rows
```

```
results |>
  select(where(is.double))
```

```
# A tibble: 130 x 14
  Grade.SNA_1 Grade.SNA_2 Grade.Review Grade.Group_self Grade.Group_All
        <dbl>       <dbl>        <dbl>            <dbl>           <dbl>
1           0           0         6.67                5            4
2           8          10         6.67                1            3
3          10          10        10                  10            9.11
4           5           5         0                   1            4
5          10          10        10                  10            9.18
# i 125 more rows
# i 9 more variables: Grade.Excercises <dbl>, Grade.Project <dbl>,
#   Grade.Literature <dbl>, Grade.Data <dbl>, Grade.Introduction <dbl>,
#   Grade.Theory <dbl>, Grade.Ethics <dbl>, Grade.Critique <dbl>, Final_grade
<dbl>
```

5 Filtering Observations

In contrast to selection which relates to obtaining a subset of the columns of the data, filtering refers to obtaining a subset of the rows. In the `tidyverse`, data filtering is carried out with the `dplyr` package function `filter()`, which should not be confused with the base R `filter()` function in the `stats` package. As we have attached the `dplyr` package, the base R `filter()` function is masked, meaning that when we write code that uses `filter()`, the `dplyr` version of the function will automatically be called.

Filtering is often a much simpler operation than selecting variables, as the filtering conditions are based solely on the values of the data variables. Using `filter()` is analogous to the base R subset operator `[`, but the filtering condition is given as an argument to the `filter()` function instead. It is good to remind that in R a single equal sign (=) is merely for arguments of function calls, while double equal sign (==) is needed for comparison of two values. And example of filter:

```
demographics |>
  filter(Origin == "Bosnia") |>
  select(Name, Surname)
```

```
# A tibble: 2 x 2
  Name  Surname
  <chr> <chr>
1 Hamza Hodžić
2 Davud Delić
```

The code above first filters our student demographics data to only those students whose country of origin is Bosnia. Then, we select their first and last names.

Multiple filtering conditions can be refined and combined using base R logical operators, such as & and `|`.

```
demographics |>
  filter(Gender == "F" & Location == "Remote")
```

```
# A tibble: 10 x 8
  user       Name    Surname      Origin      Gender Birthdate  Location Employment
  <chr>      <chr>   <chr>        <chr>       <chr>  <chr>      <chr>    <chr>
1 d93f7f0d3 Zahra   Gul          Afghanistan F      22.11.1999 Remote   None
2 93d1f2e82 Louise  Bernard      France      F      5.9.1998   Remote   Part-time
3 417892918 Miora   Rakotomalala Madagascar  F      9.12.1995  Remote   None
4 f98e6e2b8 Linda   Mensah       Ghana       F      7.2.1991   Remote   None
5 590846fe3 Lucija  Horvat       Croatia     F      22.7.1998  Remote   None
# i 5 more rows
```

Here, we filtered our data to female students who are studying remotely. The same result could also be obtained by using the `filter()` function two times

```
demographics |>
  filter(Gender == "F") |>
  filter(Location == "Remote")
```

```
# A tibble: 10 x 8
  user       Name    Surname      Origin      Gender Birthdate  Location Employment
  <chr>      <chr>   <chr>        <chr>       <chr>  <chr>      <chr>    <chr>
1 d93f7f0d3 Zahra   Gul          Afghanistan F      22.11.1999 Remote   None
2 93d1f2e82 Louise  Bernard      France      F      5.9.1998   Remote   Part-time
3 417892918 Miora   Rakotomalala Madagascar  F      9.12.1995  Remote   None
4 f98e6e2b8 Linda   Mensah       Ghana       F      7.2.1991   Remote   None
5 590846fe3 Lucija  Horvat       Croatia     F      22.7.1998  Remote   None
# i 5 more rows
```

This type of approach may improve the readability of your code especially when there are several independent filtering conditions to be applied simultaneously.

Filters can naturally be based on numeric values as well. For example, we could select those students whose final grade is higher than 8.

```
results |>
  filter(Final_grade > 8)
```

```
# A tibble: 58 x 15
  user      Grade.SNA_1 Grade.SNA_2 Grade.Review Grade.Group_self Grade.Group_All
  <chr>           <dbl>       <dbl>        <dbl>            <dbl>           <dbl>
1 111422ee7          10          10        10                  10            9.11
2 e6ec47f29          10          10        10                  10            9.18
3 4951e7386          10          10         7                   9            8.56
4 9d744e5bf          10          10        10                  10            9.29
5 0ef305578          10          10         9.33               10            8.56
# i 53 more rows
# i 9 more variables: Grade.Excercises <dbl>, Grade.Project <dbl>,
#   Grade.Literature <dbl>, Grade.Data <dbl>, Grade.Introduction <dbl>,
#   Grade.Theory <dbl>, Grade.Ethics <dbl>, Grade.Critique <dbl>, Final_grade <dbl>
```

Similarly, we could select students based on their total number of Moodle events.

```
events_summary |>
  filter(Frequency.Total > 100 & Frequency.Total < 500)
```

```
# A tibble: 44 x 2
  user      Frequency.Total
  <chr>               <int>
1 00a05cc62             417
2 046c35846             199
3 05b604102             199
4 0604ff3d3             436
5 0af619e4b             268
# i 39 more rows
```

6 Transforming Variables

In the best-case scenario, our data is already in the desired format after it has been read into R, but this is rarely the case with real datasets. We may need to compute new variables that were not present in the original data, convert measurements to different units, or transform text data into a numeric form. In the `tidyverse`, data transformations are carried out by the `mutate()` function of the `dplyr` package. This function can be used to transform multiple variables at the same time or to construct entirely new variables. The syntax of the function is the same in both cases: first, the name of the variable should be provided followed by an R expression that defines the variable. The transformed data is not automatically assigned to any variable, enabling transformations to be used as temporary variables within a chain of piped operations.

As a simple example, we could convert the students' locations into a factor variable.

```
demographics |>
  mutate(Location = factor(Location))
```

```
# A tibble: 130 x 8
  user      Name   Surname Origin     Gender Birthdate Location  Employment
  <chr>     <chr>  <chr>   <chr>      <chr>  <chr>     <fct>     <chr>
1 6eba3ff82 Amanda Mora    Costa Rica F      28.2.1998 On campus None
```

```
2 05b604102 Lian   Abdullah Yemen      F      19.11.1996 On campus None
3 111422ee7 Bekim  Krasniqi Kosovo     M      30.1.1999  On campus None
4 b4658c3a9 Yusuf  Kaya     Turkey     M      16.6.1998  On campus None
5 e6ec47f29 Zoran  Babić    Serbia     M      29.10.1998 On campus Part-time
# i 125 more rows
```

As we see from the `tibble` printout, the `Location` variable is a factor in the transformed data as indicated by the `<fct>` heading under the variable name. Note that the original `demographics` data was not changed, as we did not assign the result of the computation.

The gender and employment status of the students could also be used as factors, which we could do in a single `mutate()` call

```r
demographics |>
  mutate(
    Gender = factor(Gender),
    Location = factor(Location),
    Employment = factor(Employment)
  )
```

```
# A tibble: 130 x 8
  user      Name   Surname  Origin     Gender Birthdate  Location  Employment
  <chr>     <chr>  <chr>    <chr>      <fct>  <chr>      <fct>     <fct>
1 6eba3ff82 Amanda Mora     Costa Rica F      28.2.1998  On campus None
2 05b604102 Lian   Abdullah Yemen      F      19.11.1996 On campus None
3 111422ee7 Bekim  Krasniqi Kosovo     M      30.1.1999  On campus None
4 b4658c3a9 Yusuf  Kaya     Turkey     M      16.6.1998  On campus None
5 e6ec47f29 Zoran  Babić    Serbia     M      29.10.1998 On campus Part-time
# i 125 more rows
```

However, writing out individual identical transformations manually is cumbersome when the number of variables is large. For such cases, the `across()` function can be leveraged, which applies a function across multiple columns. This function uses the same selection syntax that we already familiarized ourselves with earlier to define the columns that will be transformed. To accomplish the same three transformations into a factor format, we could write

```r
demographics |>
  mutate(across(c(Gender, Location, Employment), factor))
```

```
# A tibble: 130 x 8
  user      Name   Surname  Origin     Gender Birthdate  Location  Employment
  <chr>     <chr>  <chr>    <chr>      <fct>  <chr>      <fct>     <fct>
1 6eba3ff82 Amanda Mora     Costa Rica F      28.2.1998  On campus None
2 05b604102 Lian   Abdullah Yemen      F      19.11.1996 On campus None
3 111422ee7 Bekim  Krasniqi Kosovo     M      30.1.1999  On campus None
4 b4658c3a9 Yusuf  Kaya     Turkey     M      16.6.1998  On campus None
5 e6ec47f29 Zoran  Babić    Serbia     M      29.10.1998 On campus Part-time
# i 125 more rows
```

The first argument to the `across()` function is the selection that defines the variables to be transformed. The second argument defines the transformation, in this case, a function, to be used.

Working with dates can often be challenging. When we read the student demographic data into R, the variable `Birthdate` was assumed to be a `character` type variable. If we would like to use this variable to e.g., compute the ages of the students, we need to first convert it into a proper format using the `as.Date` function. Since the dates in the data are not in any standard format, we must provide the format manually. Afterwards, we can use the `lubridate` [7] package to easily compute the ages of the students, which we will save into a new variable called `Age`. We will also construct another variable called `FullName` which formats the first and last names of the students as `"Last, First"`.

```
library("lubridate")
demographics |>
  mutate(
    Birthdate = as.Date(Birthdate, format = "%d.%m.%Y"),
    Age = year(as.period(interval(start = Birthdate, end = date("2023-03-12")))),
    FullName = paste0(Surname, ", ", Name)
  ) |>
  select(Age, FullName)
```

```
# A tibble: 130 x 2
    Age FullName
  <dbl> <chr>
1    25 Mora, Amanda
2    26 Abdullah, Lian
3    24 Krasniqi, Bekim
4    24 Kaya, Yusuf
5    24 Babić, Zoran
# i 125 more rows
```

The computation of the ages involves several steps. First, we construct a time interval object with the `interval()` function from the birthdate to the date for which we wish to compute the ages. Next, the `as.period()` function converts this interval into a time duration, from which we lastly get the number of years with the `year()` function.

Suppose that we would like to construct a new variable `AchievingGroup` that categorizes the students into top 50% achievers and bottom 50% achievers based on their final grade on the course. We leverage two functions from the `dplyr` package to construct this new variable: `case_when()` and `ntile()`. The function `case_when()` is used to transform variables based on multiple sequential conditions. The function `ntile()` has two arguments, a vector `x` and an integer `n`, and it splits `x` into `n` equal-sized groups based on the ranks of the values in `x`.

```r
results <- results |>
  mutate(
    AchievingGroup = factor(
      case_when(
        ntile(Final_grade, 2) == 1 ~ "Low achiever",
        ntile(Final_grade, 2) == 2 ~ "High achiever"
      )
    )
  )
```

The syntax of `case_when()` is very simple: we describe the condition for each case followed by `~` after which we define the value that the case should correspond to. We assign the result of the computation to the `results` data, as we will be using the `AchievingGroup` variable in later chapters.

We would also like to categorize the students based on their activity level, i.e., the number of total Moodle events. Our goal is to create three groups of equal size consisting of low activity, moderate activity and high activity students. The approach we applied to categorizing the achievement level of the students is also applicable for this purpose. We name our new variable as `ActivityGroup`, and we assign the result of the computation, as we will also be using this variable in later chapters.

```r
events_summary <- events_summary |>
  mutate(
    ActivityGroup = factor(
      case_when(
        ntile(Frequency.Total, 3) == 1 ~ "Low activity",
        ntile(Frequency.Total, 3) == 2 ~ "Moderate activity",
        ntile(Frequency.Total, 3) == 3 ~ "High activity"
      )
    )
  )
```

7 Rearranging Data

Sometimes we may want to reorder the rows or columns of our data, for example in alphabetical order based on the names of students on a course. The `arrange()` function from the `dplyr` package orders the rows of the by the values of columns selected by the user. The values are sorted in ascending order by default, but the order can be inverted by using the `desc()` function if desired. The variable order in the selection defines how ties should be broken when duplicate values are encountered in the previous variables of the selection. For instance, the following code would arrange the rows of our `demographics` data by first comparing the surnames of the students, and then the given names for those students with the same surname. Missing values are placed last in the reordered data.

```
demographics |>
  arrange(Surname, Name)
```

```
# A tibble: 130 x 8
  user      Name   Surname  Origin     Gender Birthdate  Location  Employment
  <chr>     <chr>  <chr>    <chr>      <chr>  <chr>      <chr>     <chr>
1 ba76ebfab Bismah Abbasi   Pakistan   F      2.4.1996   Remote    Full-time
2 05b604102 Lian   Abdullah Yemen      F      19.11.1996 On campus None
3 d2c3ce9a4 Amir   Ait      Morocco    M      19.6.1997  On campus None
4 68a377c82 Saliha Akmatova Kyrgyzstan F      19.5.1999  Remote    None
5 7e2726f3c Kazi   Akter    Bangladesh M      22.12.1992 On campus None
# i 125 more rows
```

A descending order based on both names can be obtained by applying the `desc()` function.

```
demographics |>
  arrange(desc(Surname), desc(Name))
```

```
# A tibble: 130 x 8
  user      Name    Surname  Origin    Gender Birthdate Location  Employment
  <chr>     <chr>   <chr>    <chr>     <chr>  <chr>     <chr>     <chr>
1 a48165ad5 Liam    Zambrano Ecuador   M      4.4.1998  On campus None
2 1115dae61 Poema   Wong     Tahiti    F      22.1.1999 Remote    Part-time
3 0ef305578 Jack    White    Australia M      22.4.1995 Remote    None
4 f753ce9bf Dechen  Wangmo   Bhutan    F      29.4.1999 On campus None
5 f87eaa00c Prasert Wang     Thailand  M      9.4.1997  On campus None
# i 125 more rows
```

Column positions can be changed with the `relocate()` function of the `dplyr` package. Like `arrange()`, we first select the column or columns we wish to move into a different position in the data. Afterwards, we specify the position where the columns should be moved to in relation to positions of the other columns. In our `demographics` data, the user ID column `user` is the first column. The following code moves this column after the `Employment` column so that the `user` column becomes the last column in the data.

```r
demographics |>
  relocate(user, .after = Employment)
```

```
# A tibble: 130 x 8
  Name   Surname  Origin      Gender Birthdate  Location  Employment user
  <chr>  <chr>    <chr>       <chr>  <chr>      <chr>     <chr>      <chr>
1 Amanda Mora     Costa Rica F      28.2.1998  On campus None       6eba3ff82
2 Lian   Abdullah Yemen       F      19.11.1996 On campus None       05b604102
3 Bekim  Krasniqi Kosovo      M      30.1.1999  On campus None       111422ee7
4 Yusuf  Kaya     Turkey      M      16.6.1998  On campus None       b4658c3a9
5 Zoran  Babić    Serbia      M      29.10.1998 On campus Part-time  e6ec47f29
# i 125 more rows
```

The mutually exclusive arguments `.before` and `.after` of `relocate()` specify the new column position in relation to columns that were not selected. These arguments also support the `select()` function syntax for more general selections.

8 Reshaping Data

Broadly speaking, tabular data typically take one of two formats: wide or long. In the wide format, there is one row per subject, where the subjects are identified by an identifier variable, such as the `user` variable in our Moodle data, and multiple columns for each measurement. In the long format, there are multiple rows per subject, and the columns describe the type of measurement and its value. For example, the `events` data is in a long format containing multiple Moodle events per student, but the `results` and `demographics` data are in a wide format with one row per student.

In the previous section, we constructed a summary of the users' Moodle events in total and of different types. The latter data is also in a long format with multiple rows per subject, but we would instead like to have a column for each event type with one row per user, which means that we need to convert this data into a wide format. Conversion between the two tabular formats is often called *pivoting*, and the corresponding functions `pivot_wider()` and `pivot_longer()` from the `tidyr` [8] package are also named according to this convention. We will create a wide format data of the counts of different event types using the `pivot_wider()` function as follows

```r
library("tidyr")
events_types <- events |>
  group_by(user, Action) |>
  count(Action) |>
  pivot_wider(
    names_from = "Action",
    names_prefix = "Frequency.",
    values_from = "n",
    values_fill = 0
  )
events_types
```

```
# A tibble: 130 x 13
# Groups:   user [130]
  user      Frequency.Applications Frequency.Assignment Frequency.Course_view
  <chr>                      <int>                <int>                 <int>
1 00a05cc62                      2                  121                   103
2 042b07ba1                      0                   62                   294
3 046c35846                      0                   41                    53
4 05b604102                      0                   44                    49
5 0604ff3d3                      0                    9                   119
# i 125 more rows
# i 9 more variables: Frequency.Feedback <int>, Frequency.General <int>,
#   Frequency.Group_work <int>, Frequency.Instructions <int>,
#   Frequency.La_types <int>, Frequency.Practicals <int>, Frequency.Social <int>,
#   Frequency.Ethics <int>, Frequency.Theory <int>
```

Here, we first specify the column name that the names of the wide format data should be taken from in the long format data with `names_from`. In addition, we specify a prefix for the new column names using `names_prefix` that helps to distinguish what these new columns will contain, but in general, the prefix is optional. Next, we specify the column that contains the values for the new columns with `values_from`. Because not every student necessarily has events of every type, we also need to specify what the value should be in cases where there are no events of a particular type by using `values_fill`. As we are considering the frequencies of the events, it is sensible to select 0 to be this value. We save the results to `events_types` as we will use the event type data in later sections and chapters.

9 Joining Data

Now that we have computed the total number of events for each student and converted the event type data into a wide format, we still need to merge these new data with the demographics and results data. Data merges are also called *joins*, and the dplyr package provides several functions for different kinds of joins. Here, we will use the left_join() function that will preserve all observations of the first argument.

```
left_join(demographics, events_summary, by = "user")
```

```
# A tibble: 130 x 10
  user      Name  Surname Origin Gender Birthdate Location Employment Frequency.Total
  <chr>     <chr> <chr>   <chr>  <chr>  <chr>     <chr>    <chr>                <int>
1 6eba3ff~ Aman~ Mora    Costa~ F      28.2.1998 On camp~ None                   312
2 05b6041~ Lian  Abdull~ Yemen  F      19.11.19~ On camp~ None                   199
3 111422e~ Bekim Krasni~ Kosovo M      30.1.1999 On camp~ None                   532
4 b4658c3~ Yusuf Kaya    Turkey M      16.6.1998 On camp~ None                   246
5 e6ec47f~ Zoran Babić   Serbia M      29.10.19~ On camp~ Part-time              356
# i 125 more rows
# i 1 more variable: ActivityGroup <fct>
```

In essence, the above left join adds all columns from events_summary to demographics whenever there is a matching value in the by column. To continue, we can use additional left joins to add the remaining variables from the results data, and the Moodle event counts of different types from events_types to have all the student data in a single object.

```
all_combined <- demographics |>
  left_join(events_types, by = "user") |>
  left_join(events_summary, by = "user") |>
  left_join(results, by = "user")
all_combined
```

```
# A tibble: 130 x 37
  user       Name   Surname  Origin     Gender Birthdate  Location  Employment
  <chr>      <chr>  <chr>    <chr>      <chr>  <chr>      <chr>     <chr>
1 6eba3ff82 Amanda Mora     Costa Rica F      28.2.1998  On campus None
2 05b604102 Lian   Abdullah Yemen      F      19.11.1996 On campus None
3 111422ee7 Bekim  Krasniqi Kosovo     M      30.1.1999  On campus None
4 b4658c3a9 Yusuf  Kaya     Turkey     M      16.6.1998  On campus None
5 e6ec47f29 Zoran  Babić    Serbia     M      29.10.1998 On campus Part-time
# i 125 more rows
# i 29 more variables: Frequency.Applications <int>, Frequency.Assignment <int>,
#   Frequency.Course_view <int>, Frequency.Feedback <int>, Frequency.General <int>,
#   Frequency.Group_work <int>, Frequency.Instructions <int>,
#   Frequency.La_types <int>, Frequency.Practicals <int>, Frequency.Social <int>,
#   Frequency.Ethics <int>, Frequency.Theory <int>, Frequency.Total <int>,
#   ActivityGroup <fct>, Grade.SNA_1 <dbl>, Grade.SNA_2 <dbl>, ...
```

We will use this combined dataset in the following chapters as well.

10 Missing Data

Sometimes it occurs that learning analytics data has cells for which the values are missing for some reason. The Moodle event data which we have utilized in this chapter does not naturally contain missing data. Thus, to have an example, we need to create a data which does. Second, handling of missing data is a vast topic of which we can only discuss some of the key points very briefly from a practical perspective. For a more comprehensive overview, we recommend reading [9] and [10] for a hands on approach. A short overview of missingness can be found in [11].

The code below will create missing values randomly to each column of `events_types` data (`user` column is an exception). To do that, we use the `mice` [12] package which also has methods for the handling of missing data. Unfortunately, `mice` is not part of the `tidyverse`. For more information about `mice`, a good source is `miceVignettes` at https://www.gerkovink.com/miceVignettes/. Now, let's create some missing data.

```
library("mice")
set.seed(44)
events_types <- events_types |>
  rename(
    "Ethics" = "Frequency.Ethics",
    "Social" = "Frequency.Social",
    "Practicals" = "Frequency.Practicals"
  )
ampute_list <- events_types |>
  ungroup(user) |>
  select(Ethics:Practicals)|>
  as.data.frame() |>
  ampute(prop = 0.3)
events_types_mis <- ampute_list$amp |>
  as_tibble()
events_types_mis[2, "Practicals"] <- NA
```

Above, we also rename the variables that contain the frequencies of Moodle events related to ethics, social and practicals into `Ethics`, `Social` and `Practicals`, respectively. Let's now see some of the values of `events_types_mis`

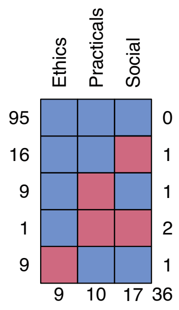
```
events_types_mis
```

```
# A tibble: 130 x 3
  Ethics Social Practicals
   <int>  <int>      <int>
1     NA     12         89
```

```
2      14      NA          NA
3       0       0          47
4       0       0          48
5       0       0          61
# i 125 more rows
```

We can see that now the data contains `NA` values in some of the cells. These are the cells in which a missing value occurs, meaning that a value for those measurements has not been recorded. A missing data pattern, that is how missing of one variable affects missingness of other variables, can be show as:

```
md.pattern(events_types_mis, rotate.names = TRUE)
```



Above, each red square indicates a missing value while blue squares stand for observed ones. We can see that there are 95 complete rows, 10 for which `Practicals` are missing, 17 have missingness on `Social` and 9 are missing on `Ethics`. Also, one row has two red squares indicating a missing value on both `Social` and `Practicals`.

Let's now discuss options of handling missing data briefly. There are four classes of statistical methods for analyzing data with missing values: complete case (CC) methods, weighting methods, imputation methods, and model-based methods. The simplest of these is complete case analysis, which leaves missing values out of the analysis and only uses observations with all variables recorded. This can be done with the `tidyr` [8] package function `drop_na()`:

```
events_types_mis |>
  drop_na()
```

```
# A tibble: 95 x 3
  Ethics Social Practicals
   <int>  <int>      <int>
1      0      0         47
```

```
2       0       0          48
3       0       0          61
4       0      24         102
5       4      18          71
# i 90 more rows
```

We can see that after using this method, our data has only 95 rows as those were the rows without any columns having missing values. This made our data much smaller! If there are a lot of missing values, the data may become too small to use for practical purposes.

A more novel group of methods are imputation methods. One of the options is using single imputation (SI) where the mean of each variable will determine the imputed value. The single mean imputation can be done as follows:

```
imp <- mice(events_types_mis, method = "mean",
m = 1, maxit = 1 , print = FALSE)
complete(imp) |>
  head()
```

```
     Ethics   Social Practicals
1  7.553719 12.00000   89.00000
2 14.000000 15.64602   74.80833
3  0.000000  0.00000   47.00000
4  0.000000  0.00000   48.00000
5  0.000000  0.00000   61.00000
6  0.000000 24.00000  102.00000
```

We can see from above that the imputed values are not integers anymore. However, if we aim to estimate means or regression coefficients (see Chapter 5 [13] for details) that is not a problem. One of the problems with mean imputation is that the variance and standard error estimates will become downward biased. A mean of `Ethics` for mean imputation is:

```
fit <- with(imp, lm(Ethics ~ 1))
summary(fit)
```

```
# A tibble: 1 x 6
  term        estimate std.error statistic  p.value  nobs
  <chr>          <dbl>     <dbl>     <dbl>    <dbl> <int>
1 (Intercept)     7.55     0.811      9.32 4.20e-16   130
```

Next, let’s briefly have a look at how we can utilize multiple imputation (MI) which is an improvement over single imputation. The multiple imputation approach generates more than one imputation thus creating many complete data sets for us.

For each of these datasets, we can perform any analysis that we are interested in. After the analysis, one must pool the results from the impured datasets to get the final result. Here, we utilize a method called predictive mean matching (`method = "pmm"` in the code below), which uses the neighbour values of data as imputations.

```
imp2 <- mice(events_types_mis, method = "pmm",
m= 10, maxit = 100, print = FALSE)
fit2 <- with(imp2, lm(Ethics ~ Practicals))
pool_fit <- pool(fit2)
# Multiple imputation
summary(pool_fit)
```

```
         term   estimate  std.error statistic       df     p.value
1 (Intercept) 1.62080372 2.18285149 0.7425167 101.7304 0.459485402
2  Practicals 0.08049328 0.02616765 3.0760606 106.0802 0.002668431
```

```
# Complete cases
summary(lm(Ethics ~ Practicals, events_types_mis))["coefficients"]
```

```
$coefficients
              Estimate Std. Error  t value   Pr(>|t|)
(Intercept) 2.15459162 2.12235051 1.015191 0.31226283
Practicals  0.06220884 0.02605001 2.388054 0.01865793
```

```
# Without missingness
summary(lm(Ethics ~ Practicals, events_types))["coefficients"]
```

```
$coefficients
              Estimate Std. Error   t value     Pr(>|t|)
(Intercept) 1.05409529 2.17821479 0.4839262 0.6292651258
Practicals  0.08891892 0.02590313 3.4327482 0.0008053447
```

From the results above, we can see that in this particular case the multiple imputation performs well in comparison to CC approach. The regression coefficient for full data without any missing values is 0.089, and it is 0.080 for multiple imputation, while complete case analysis gives 0.062. As all of them have very similar standard errors, this yields that MI and full data give statistically significant p-values for significance level 0.01, while CC does not.

11 Correcting Erroneous Data

Let's imagine that our data has an error on the surname variable `Surname` and that all the names ending with "sen" should end with "ssen". What we can do is that we can use regular expressions to detect the erroneous rows and we can also use them to replace the values. Let's first figure out which last names contain a name ending with "sen". We can use a function `str_detect()` to return TRUE/FALSE for each row from `stringr` [14] package within a `filter()` function call. We define `pattern = "sen$"` where $ indicates the end of the string.

```
library("stringr")
demographics |>
  filter(str_detect(string = Surname, pattern = "sen$")) |>
  pull(Surname)
```

```
[1] "Nielsen"  "Johansen" "Joensen"  "Jansen"   "Olsen"
```

After pulling the filtered surnames, there seems to be five surnames ending with "sen". Next, let's try to replace "sen" with "ssen". On the next row we filter just as previously to limit output.

```
demographics |>
  mutate(Surname = str_replace(
    string = Surname, pattern = "sen$", replacement = "ssen")
  ) |>
  filter(str_detect(string = Surname, pattern = "sen$")) |>
  pull(Surname)
```

```
[1] "Nielssen"  "Johanssen" "Joenssen"  "Janssen"   "Olssen"
```

Thus, the following code updates the data so that all the surnames ending with "sen" now end with "ssen" instead.

```
demographics <- demographics |>
  mutate(Surname = str_replace(
    string = Surname, pattern = "sen$", replacement = "ssen")
  )
```

12 Conclusion and Further Reading

Data wrangling is one of the most important steps in any data analysis pipeline. This chapter introduced the `tidyverse`, tidy data, and several commonly used R packages for data manipulation and their use in basic scenarios in the context of learning analytics. However, the `tidyverse` is vast and can hardly be fully covered in a single chapter. We refer the reader to additional resources such as those found on the tidyverse website at https://www.tidyverse.org/learn/ and the book "R for Data Science" by Hadley Wicham and Garret Grolemund. The book is free to use and readily available online at https://r4ds.had.co.nz/.

References

1. Wickham H, Averick M, Bryan J, Chang W, McGowan LD, François R, Grolemund G, Hayes A, Henry L, Hester J, Kuhn M, Pedersen TL, Miller E, Bache SM, Müller K, Ooms J, Robinson D, Seidel DP, Spinu V, Takahashi K, Vaughan D, Wilke C, Woo K, Yutani H (2019) Welcome to the tidyverse. J Open Source Softw 4:1686. https://doi.org/10.21105/joss.01686
2. Wickham H, Hester J, Bryan J (2024) readr: read rectangular text data. R package version 2.1.5. https://CRAN.R-project.org/package=readr
3. López-Pernas S, Saqr M, Conde J, Del-Río-Carazo L (2024) A broad collection of datasets for educational research training and application. In: Saqr M, López-Pernas S (eds) Learning analytics methods and tutorials: a practical guide using R. Springer, Berlin
4. Chan C, Leeper T, Becker J, Schoch D (2023) rio: a swiss-army knife for data file I/O. https://cran.r-project.org/package=rio
5. Müller K, Wickham H (2023) tibble: simple data frames. R package version 3.2.1. https://CRAN.R-project.org/package=tibble
6. Wickham H, François R, Henry L, Müller K, Vaughan D (2023) dplyr: a grammar of data manipulation. R package version 1.1.4. https://CRAN.R-project.org/package=dplyr
7. Grolemund G, Wickham H (2011) Dates and times made easy with lubridate. J Stat Softw 40(3):1–25. https://www.jstatsoft.org/v40/i03/
8. Wickham H, Vaughan D, Girlich M (2023) tidyr: tidy messy data. R package version 1.3.0. https://CRAN.R-project.org/package=tidyr
9. Little RJA, Rubin DB (2020) Statistical analysis with missing data, 3rd edn. Wiley, Hoboken
10. van Buuren S (2018) Flexible imputation of missing data. Chapman et Hall/CRC Press, Boca Raton. https://doi.org/10.1201/9780429492259
11. Kopra J (2018) Statistical modelling of selective non-participation in health examination surveys. Report/University of Jyväskylä, Department of Mathematics and Statistics
12. van Buuren S, Groothuis-Oudshoorn K (2011) mice: multivariate imputation by chained equations in R. J Stat Softw 45:1–67. https://doi.org/10.18637/jss.v045.i03
13. Tikka S, Kopra J, Heinäniemi M, López-Pernas S, Saqr M (2024) Introductory statistics with R for educational researchers. In: Saqr M, López-Pernas S (eds). Springer, Berlin
14. Wickham H (2023) stringr: simple, consistent wrappers for common string operations. R package version 1.5.1. https://CRAN.R-project.org/package=stringr