One Page R Data Science
Data Wrangling

Graham.Williams@togaware.com

8th September 2018

Visit https://essentials.togaware.com/onepagers for more Essentials.

20180721

The *business understanding* phase of a data science project aims to understand the business problem and to then liaise with the business data technicians to identify the data available. This is followed by the *data understanding* phase where we work with the business data technicians to access and ingest the data into R. We are then in a position to initiate our journey of discovery driven by the data. By *living and breathing* the data in the context of the business problem we gain our bearings and feed our intuitions as we journey.

In this chapter we present the common series of steps for the data phase of data science. As we progress through the chapter we build a ***template*** designed to be reused for other journeys. As we foreshadowed in Chapter 1 rather than delving into the intricacies of the R language we immerse ourselves into using R to achieve our outcomes, learning more about R as we proceed.

The template consists of programming code that can be reused with little or no modification on a new dataset. The intention is that to get started with a new dataset only a few lines at the top of the template need to be modified. No or only minimal change is then required for the remainder of the code. In many respects the concept of a template is a stepping stone toward writing functions in R.

Through this guide new R commands will be introduced. The reader is encouraged to review the command's documentation and understand what the command does. Help is obtained using the `?` command as in:

```
?read.csv
```

Documentation on a particular package can be obtained using the `help=` option of `library()`:

```
library(help=rattle)
```

This chapter is intended to be hands on. To learn effectively you are encouraged to run R (e.g., RStudio or Emacs with ESS mode) and to replicate the commands. Check that output is the same and that you understand how it is generated. Try some variations. Explore.

1 Packages Used

20180908

Packages used in this chapter include dplyr (Wickham *et al.*, 2018b), FSelector (Romanski and Kotthoff, 2018), ggplot2 (Wickham *et al.*, 2018a), glue (Hester, 2018), lubridate (Spinu *et al.*, 2018), randomForest (Breiman *et al.*, 2018), readr (Wickham *et al.*, 2017), stringi (Gagolewski *et al.*, 2018), stringr (Wickham, 2018), tidyr (Wickham and Henry, 2018), magrittr (Bache and Wickham, 2014), and rattle (Williams, 2018).

```
# Load required packages from local library into R session.

library(rattle)       # normVarNames().
library(readr)        # Efficient reading of CSV data.
library(dplyr)        # Data wrangling, glimpse() and tbl_df().
library(tidyr)        # Prepare a tidy dataset, gather().
library(magrittr)     # Pipes %>% and %T>% and equals().
library(glue)         # Format strings.
library(lubridate)    # Dates and time.
library(FSelector)    # Feature selection, information.gain().
library(stringi)      # String concat operator %s+%.
library(stringr)      # String operations.
library(randomForest) # Impute missing values with na.roughfix().
library(ggplot2)      # Visualise data.
```


2 Data Source

To begin the data phase of a project we will identify a data source. For our purposes we use the simplest of sources—a text-based CSV (comma separated value) file as a typical data source format. *20180908*

The **weatherAUS** dataset from rattle will be used. A binary formatted R dataset is provided by the package but the CSV file for the same dataset is available at https://rattle.togaware.com/weatherAUS.csv.

We first identify and record the location of the CSV file to analyse. R is capable of ingesting data directly from the Internet and so we will illustrate how to do so here. The location of the file (the so-called URL or universal resource location) will be saved as a string in a variable called dspath—the path to the dataset. The following assignment command does this for us. Simply type this into your R script file within RStudio. The command is then executed in RStudio by clicking the Run button whilst the cursor is situated on the line within the script file.

```
# Note the source location of a dataset to ingest into R.

dspath <- "http://rattle.togaware.com/weatherAUS.csv"
```

The assignment operator <- will store the value on the right hand side (which is a string enclosed within quotation marks) into the computer's memory and we can later refer to it as the R variable dspath—we retrieve the string simply by reference to the variable dspath.

By typing the name of the variable (dspath) in the R Console at the > prompt R will respond with the value stored in the variable:

```
dspath
```

```
## [1] "http://rattle.togaware.com/weatherAUS.csv"
```

If not connected to the Internet we can read the data directly from a local copy of the CSV file. The rattle package (once the package has been installed) provides a smaller sample weather.csv. The location of the CSV file within rattle is determined using base::system.file(). Knowing that CSV files are located within the csv sub-directory of the rattle package we generate the string that identifies the file system path to weather.csv.

```
dspath <- system.file("csv", "weather.csv", package="rattle") %T>% print()
```

```
## [1] "/usr/lib/R/site-library/rattle/csv/weather.csv"
```

This is the path to the CSV file on my file system. Your path may well be different depending on where your system installed the rattle package.

Note that this is a considerably smaller subset of the full weatherAUS dataset and ingesting this rather than the full dataset will lead to different results to those presented here.

If you have separately downloaded weatherAUS.csv then you can identify its location. Here we identify that the downloaded file is located in the current working directory.

```
dspath <- "./weatherAUS.csv"
```


3 Data Ingestion

20180721

Having identified the source of the dataset we can ingest the dataset into the memory of the computer using the function `readr::read_csv()`. This function returns an enhanced ***data frame***. A data frame is the basic data structure used to store a dataset within R and the enhanced data frame from the tidyverse adds functionality that improves our interactions with the data frame.

We set up a reference to the data frame's location in the computer's memory by assigning the result of the call to the function `readr::read_csv()` to the R variable `weather`.

```
# Ingest the dataset.

weather <- read_csv(file=dspath)

## Parsed with column specification:
## cols(
##   .default = col_character(),
##   Date = col_date(format = ""),
##   MinTemp = col_double(),
##   MaxTemp = col_double(),
##   Rainfall = col_double(),
##   WindGustSpeed = col_integer(),
##   WindSpeed9am = col_integer(),
##   WindSpeed3pm = col_integer(),
##   Humidity9am = col_integer(),
##   Humidity3pm = col_integer(),
##   Pressure9am = col_double(),
##   Pressure3pm = col_double(),
##   Cloud9am = col_integer(),
##   Cloud3pm = col_integer(),
##   Temp9am = col_double(),
##   Temp3pm = col_double(),
##   RISK_MM = col_double()
## )

## See spec(...)  for full column specifications.
```

As a side effect of calling the function `readr::read_csv()` helpful messages are displayed that identify the data types for each of the variables found in the ingested dataset. We should review these to ensure they match our expectations. If they don't, there are optional arguments to `readr::read_csv()` to inform it otherwise.

Note that the rattle also provides a smaller `rattle::weather` dataset as an R dataset, also named `weather`. Simply by attaching the rattle package from the library a variable called `weather` becomes available. Running the above command will replace the dataset provided by rattle. Having done so we can still access the `weather` dataset provided by rattle using the package prefix as in `rattle::weather`.

4 Data Frame

A data frame can essentially be thought of as a rectangular table of data consisting of rows (***observations***) and columns (***variables***). We can view the structure of such a table as it stores the weatherAUS dataset. Here we choose to display the first 10 observations of the first 6 variables. *20180721*

```
# Display the table structure of the ingested dataset.

weather[1:10,1:6] %>% print.data.frame()
```

```
##          Date Location MinTemp MaxTemp Rainfall Evaporation
## 1  2008-12-01   Albury    13.4    22.9      0.6        <NA>
## 2  2008-12-02   Albury     7.4    25.1      0.0        <NA>
## 3  2008-12-03   Albury    12.9    25.7      0.0        <NA>
## 4  2008-12-04   Albury     9.2    28.0      0.0        <NA>
## 5  2008-12-05   Albury    17.5    32.3      1.0        <NA>
## 6  2008-12-06   Albury    14.6    29.7      0.2        <NA>
## 7  2008-12-07   Albury    14.3    25.0      0.0        <NA>
## 8  2008-12-08   Albury     7.7    26.7      0.0        <NA>
## 9  2008-12-09   Albury     9.7    31.9      0.0        <NA>
## 10 2008-12-10   Albury    13.1    30.1      1.4        <NA>
```

By choosing to pipe the data through `base::print.data.frame()` we request a raw display of the actual data frame.

Next we select 10 random observations, using `dplyr::sample_n()`, of 5 random variables, orchestrated using `dplyr::select()` of a `dplyr::sample()`, with the support of `base::ncol()`.

```
# Display a random selection of observations and variables.

weather %>%
  sample_n(10) %>%
  select(sample(1:ncol(weather), 5)) %>%
  print.data.frame()
```

```
##    WindDir3pm Pressure3pm Temp3pm Evaporation WindDir9am
## 1          NW      1007.3    22.0        <NA>        NNW
## 2         ESE      1023.6    16.6           6         SE
## 3         ENE      1011.0    28.8        <NA>          N
## 4           N      1018.0    23.3         4.8        ENE
## 5          NE      1005.6    30.1         6.2         SE
## 6         NNE          NA    16.1         1.2          N
## 7           W      1019.4    15.6         3.2        WNW
## 8         NNE      1028.3    22.9        <NA>         NE
## 9         ENE      1014.7    27.3         8.2        SSE
## 10          N      1020.8    10.7        <NA>        NNE
```

Observe that this is a tabular form (i.e., it has rows and columns) and that we will generally be working with datasets in such a tabular form.

5 The Shape of the Dataset

Once the dataset is loaded we want to get a basic idea of what it looks like—its shape. Being an extended data frame (what we call a tibble), we can display the data as a tibble simply by printing the data referred to by the variable name. *20180721*

```
# Print the dataset in a human useful way.

weather
```

```
## # A tibble: 145,463 x 24
##    Date       Location MinTemp MaxTemp Rainfall Evaporation Sunshine
##    <date>     <chr>      <dbl>   <dbl>    <dbl> <chr>       <chr>
##  1 2008-12-01 Albury      13.4    22.9      0.6 <NA>        <NA>
##  2 2008-12-02 Albury       7.4    25.1      0   <NA>        <NA>
##  3 2008-12-03 Albury      12.9    25.7      0   <NA>        <NA>
##  4 2008-12-04 Albury       9.2    28        0   <NA>        <NA>
##  5 2008-12-05 Albury      17.5    32.3      1   <NA>        <NA>
##  6 2008-12-06 Albury      14.6    29.7      0.2 <NA>        <NA>
##  7 2008-12-07 Albury      14.3    25        0   <NA>        <NA>
##  8 2008-12-08 Albury       7.7    26.7      0   <NA>        <NA>
##  9 2008-12-09 Albury       9.7    31.9      0   <NA>        <NA>
## 10 2008-12-10 Albury      13.1    30.1      1.4 <NA>        <NA>
## # ... with 145,453 more rows, and 17 more variables: WindGustDir <chr>,
## #   WindGustSpeed <int>, WindDir9am <chr>, WindDir3pm <chr>,
## #   WindSpeed9am <int>, WindSpeed3pm <int>, Humidity9am <int>,
## #   Humidity3pm <int>, Pressure9am <dbl>, Pressure3pm <dbl>, Cloud9am <int>,
## #   Cloud3pm <int>, Temp9am <dbl>, Temp3pm <dbl>, RainToday <chr>,
## #   RISK_MM <dbl>, RainTomorrow <chr>
```

We observe that dataset consists of 145,463 observations of 24 variables. The enhanced nature of the data frame that representing it as a tibble brings to us is that the printout is more informative. The first few observations are shown with a subset of the variables followed by a list of all of the other variables.

6 A Glimpse of the Dataset

A useful alternative to gain some insight into the dataset is through `tibble::glimpse()`. *20180721*

```
# A quick view of the contents of the dataset.

glimpse(weather)
```

```
## Observations: 145,463
## Variables: 24
## $ Date          <date> 2008-12-01, 2008-12-02, 2008-12-03, 2008-12-04, 200...
## $ Location      <chr> "Albury", "Albury", "Albury", "Albury", "Albury", "A...
## $ MinTemp       <dbl> 13.4, 7.4, 12.9, 9.2, 17.5, 14.6, 14.3, 7.7, 9.7, 13...
## $ MaxTemp       <dbl> 22.9, 25.1, 25.7, 28.0, 32.3, 29.7, 25.0, 26.7, 31.9...
## $ Rainfall      <dbl> 0.6, 0.0, 0.0, 0.0, 1.0, 0.2, 0.0, 0.0, 0.0, 1.4, 0....
## $ Evaporation   <chr> NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, ...
## $ Sunshine      <chr> NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, ...
## $ WindGustDir   <chr> "W", "WNW", "WSW", "NE", "W", "WNW", "W", "W", "NNW"...
## $ WindGustSpeed <int> 44, 44, 46, 24, 41, 56, 50, 35, 80, 28, 30, 31, 61, ...
## $ WindDir9am    <chr> "W", "NNW", "W", "SE", "ENE", "W", "SW", "SSE", "SE"...
## $ WindDir3pm    <chr> "WNW", "WSW", "WSW", "E", "NW", "W", "W", "W", "NW",...
## $ WindSpeed9am  <int> 20, 4, 19, 11, 7, 19, 20, 6, 7, 15, 17, 15, 28, 24, ...
## $ WindSpeed3pm  <int> 24, 22, 26, 9, 20, 24, 24, 17, 28, 11, 6, 13, 28, 20...
## $ Humidity9am   <int> 71, 44, 38, 45, 82, 55, 49, 48, 42, 58, 48, 89, 76, ...
## $ Humidity3pm   <int> 22, 25, 30, 16, 33, 23, 19, 19, 9, 27, 22, 91, 93, 4...
## $ Pressure9am   <dbl> 1007.7, 1010.6, 1007.6, 1017.6, 1010.8, 1009.2, 1009...
## $ Pressure3pm   <dbl> 1007.1, 1007.8, 1008.7, 1012.8, 1006.0, 1005.4, 1008...
## $ Cloud9am      <int> 8, NA, NA, NA, 7, NA, 1, NA, NA, NA, NA, 8, 8, NA, N...
## $ Cloud3pm      <int> NA, NA, 2, NA, 8, NA, NA, NA, NA, NA, NA, 8, 8, 7, N...
## $ Temp9am       <dbl> 16.9, 17.2, 21.0, 18.1, 17.8, 20.6, 18.1, 16.3, 18.3...
## $ Temp3pm       <dbl> 21.8, 24.3, 23.2, 26.5, 29.7, 28.9, 24.6, 25.5, 30.2...
## $ RainToday     <chr> "No", "No", "No", "No", "No", "No", "No", "No", "No"...
## $ RISK_MM       <dbl> 0.0, 0.0, 0.0, 1.0, 0.2, 0.0, 0.0, 0.0, 1.4, 0.0, 2....
## $ RainTomorrow  <chr> "No", "No", "No", "No", "No", "No", "No", "No", "Yes...
```

Again we receive a printed summary of the dataset, reporting on the number of observations and variables, but now the table is effectively rotated so that all variables can be listed along with their data type and a selection of their values for the first few observations.

7 Introducing Template Variables

A reference to the original dataset can be created using a template (or generic) variable. The new variable will be called `ds` (short for dataset). *20180721*

```
# Take a copy of the dataset into a generic variable.

ds <- weather
```

Both `ds` and `weather` will now reference the same dataset within the computer's memory. As we modify `ds` those modifications will only affect the data referenced by `ds`. Effectively, an extra copy of the dataset in the computer's memory will start to grow as we change the data from its original form. R avoids making copies of datasets unnecessarily and so a simple assignment does not create a new copy. As modifications are made to one or the other copy of a dataset then extra memory will be used to store the columns that differ between the datasets.

From here on we no longer refer to the dataset as `weather` but as `ds`. This allows the following analyses and processing to be rather generic—turning the R code into a *template* and so requiring only minor modification when used with a different dataset assigned into `ds`.

Often we will find that we can simply load a different dataset into memory, store it as `ds` and the remaining steps of our analyses and processing will essentially work unchanged.

The first few steps of our template are then captured as creating the reference to the dataset and presenting our initial view of the dataset.

```
# Prepare for a templated analysis and processing.

dsname <- "weather"
ds     <- get(dsname)
glimpse(ds)
```

```
## Observations: 145,463
## Variables: 24
## $ Date          <date> 2008-12-01, 2008-12-02, 2008-12-03, 2008-12-04, 200...
## $ Location      <chr> "Albury", "Albury", "Albury", "Albury", "Albury", "A...
## $ MinTemp       <dbl> 13.4, 7.4, 12.9, 9.2, 17.5, 14.6, 14.3, 7.7, 9.7, 13...
## $ MaxTemp       <dbl> 22.9, 25.1, 25.7, 28.0, 32.3, 29.7, 25.0, 26.7, 31.9...
....
```

We are a little tricky here in recording the dataset name in the variable `dsname` and then using the function `base::get()` to make a copy of the dataset reference and link it to the generic variable `ds`. We could simply assign the data to `ds` directly as we saw above. Either way the generic variable `ds` refers to the same dataset. The use of `base::get()` allows us to be a little more generic in our template.

The use of generic variables within a template for the tasks we perform on each new dataset will have obvious advantages but we need to be careful. A disadvantage is that we may be working with several datasets and accidentally overwrite previously processed datasets referenced using the same generic variable (`ds`). The processing of the dataset might take some time and so accidentally losing it is not an attractive proposition. Care needs to be taken to avoid this.

8 Locating Datasets in Memory

We can see that **ds** and **weather** reference the same dataset in memory using dplyr::**location()** and dplyr::**changes()**. *20180721*

```
location(weather)
```

```
## <0x5622da6a4140>
## Variables:
##  * Date:          <0x5622e05a0da0>
##  * Location:      <0x5622e06bcf90>
##  * MinTemp:       <0x5622e07d9180>
##  * MaxTemp:       <0x5622e08f5370>
....
```

```
location(ds)
```

```
## <0x5622da6a4140>
## Variables:
##  * Date:          <0x5622e05a0da0>
##  * Location:      <0x5622e06bcf90>
##  * MinTemp:       <0x5622e07d9180>
##  * MaxTemp:       <0x5622e08f5370>
....
```

```
changes(weather, ds)
```

```
## <identical>
```

This gets rather technical (or geeky), but the strings of digits and characters within the angle brackets are actual memory addresses—that is, they are pointers to a direct location in our computer's memory. The `0x` at the beginning of each identifies that a hexadecimal scheme is used, thus we see digits 0 to 9 and then the letters a, b, c, d, e, and f being used. That is, 16 digits.

The thing to note is that the addresses recorded for **weather** and **ds**, including the addresses where we find the actual variables (columns) within each dataset, are identical. This is confirmed by the call to dplyr::**changes()**.

9 Changing Datasets in Memory

Let's make a change to the weather dataset by simply changing a single cell, changing the value of `MinTemp` (the third variable) for the first observation to `5`. *20180721*

```
weather[1,3] <- 5
```

Notice the divergence of the two datasets. They still share a lot in common, and hence only one copy of that data, but where they diverge, they now use different memory locations.

```
location(weather)
```

```
## <0x5622dc266410>
## Variables:
##  * Date:          <0x5622e05a0da0>
##  * Location:      <0x5622e06bcf90>
##  * MinTemp:       <0x5622e54a1d00>
##  * MaxTemp:       <0x5622e08f5370>
....
```

```
location(ds)
```

```
## <0x5622da6a4140>
## Variables:
##  * Date:          <0x5622e05a0da0>
##  * Location:      <0x5622e06bcf90>
##  * MinTemp:       <0x5622e07d9180>
##  * MaxTemp:       <0x5622e08f5370>
....
```

Using `dplyr::changes()` makes clear the changes.

```
changes(weather, ds)
```

```
## Changed variables:
##           old            new
## MinTemp   0x5622e54a1d00 0x5622e07d9180
##
## Changed attributes:
##           old            new
## row.names 0x5622e3ceb800 0x5622e3cf0760
```

That's an interesting aside, but we now get back to our actual data analysis and processing.

10 Reviewing Variable Names

The names of the variables within the dataset as supplied to us may not be in any particular form and may use different conventions. For example, they may use a mix of upper and lower case letters (`TempToday9AM`) or be very long (`Temperature_Recorded_Today_9am`) or use sequential numbers to identify each variable (`V004` or `V004_rainToday`) or use codes (`XVn34_rain`) or any number of other conventions. Often we prefer to simplify the variable names to ease our processing and thinking and to enforce a standard and consistent naming convention for ourselves.

We use `base::names()` to list the names of the variables within a dataset.

```
# Review the variables to consider normalising their names.

names(ds)
```

```
##  [1] "Date"          "Location"      "MinTemp"       "MaxTemp"
##  [5] "Rainfall"      "Evaporation"   "Sunshine"      "WindGustDir"
##  [9] "WindGustSpeed" "WindDir9am"    "WindDir3pm"    "WindSpeed9am"
## [13] "WindSpeed3pm"  "Humidity9am"   "Humidity3pm"   "Pressure9am"
## [17] "Pressure3pm"   "Cloud9am"      "Cloud3pm"      "Temp9am"
## [21] "Temp3pm"       "RainToday"     "RISK_MM"       "RainTomorrow"
....
```

Notice that the names here use a scheme whereby the initial letter is capitalised and each word within the variable name is also capitalised. That's a reasonable naming scheme and is preferred by some.

11 Normalizing Variable Names

20180721

A convenient convention that I personally prefer is to map all variable names to lowercase. R is case sensitive so that doing this will result in different variable names as far as R is concerned. Such (so called) normalisation is useful when different upper/lower case conventions are intermixed inconsistently in names like `Incm_tax_PyBl`. Remembering how to capitalize when interactively exploring the data with thousands of such variables can be quite a cognitive load for us. Yet we often see such variable names arising in practise especially when we import data from databases which are often case insensitive.

We can use `rattle::normVarNames()` to make a reasonable attempt of converting variables from a dataset into a preferred standard form. The actual form follows a style that is presented in Appendix 6. The example below shows the transformation into a normalised form. We make extensive use of the function `base::names()` to work with the variable names.

```
# Normalise the variable names.

names(ds) %<>% normVarNames() %T>% print()
```

```
##  [1] "date"            "location"        "min_temp"        "max_temp"
##  [5] "rainfall"        "evaporation"     "sunshine"        "wind_gust_dir"
##  [9] "wind_gust_speed" "wind_dir_9am"    "wind_dir_3pm"    "wind_speed_9am"
## [13] "wind_speed_3pm"  "humidity_9am"    "humidity_3pm"    "pressure_9am"
## [17] "pressure_3pm"    "cloud_9am"       "cloud_3pm"       "temp_9am"
## [21] "temp_3pm"        "rain_today"      "risk_mm"         "rain_tomorrow"
....
```

Notice the use of the assignment pipe here as introduced in Chapter 1. We will recall that the `magrittr::%<>%` operator pipes the left-hand data to the function on the right-hand side and then returns the result to the left-hand side overwriting the original contents of the memory referred to on the left-hand side.

12 Effect on Data Storage

20180721

When the names of the variables within a dataset are changed R does not make a complete new copy of the dataset. Instead, the actual data in the column remains in tack whilst the variable itself (`ds`) references a new memory location where the new variable names get noted. The underlying data within the table is unaffected.

```
location(weather)
```

```
## <0x5622dc266410>
## Variables:
##  * Date:          <0x5622e05a0da0>
##  * Location:      <0x5622e06bcf90>
##  * MinTemp:       <0x5622e54a1d00>
##  * MaxTemp:       <0x5622e08f5370>
....
```

```
location(ds)
```

```
## <0x5622e581e0c0>
## Variables:
##  * date:            <0x5622e05a0da0>
##  * location:        <0x5622e06bcf90>
##  * min_temp:        <0x5622e07d9180>
##  * max_temp:        <0x5622e08f5370>
....
```


13 Special Case Variable Name Transformations

20180721

When reviewing the variables of a dataset we often notice other changes that could be made to the variable names. This might be to simplify the variables or to clarify the meaning of the variable. The string processing functions provided by stringr come in handy for such processing.

In the following example we remove the prefix of the variable names where we identify that the prefix consists of all characters up to the first underscore. This is useful where a dataset has prefixed each variable by a sequential number or by some other code and we have no real use of such a prefix in our processing.

```
names(ds) %<>% str_replace("^[^_]*_", "")
```

This will take a variable name like `ab123_tax_payable` and convert it to `tax_payable`.

```
str_replace("ab123_tax_payable", "^[^_]*_", "")
```

```
## [1] "tax_payable"
```

The odd looking characters in the argument to `stringr::str_replace()` are a ***regular expression***. Regular expressions are a very powerful concept and can get quite complex. The reader is referred to the many resources on-line that cover regular expressions. The regular expression is a pattern used to match some part of the variable name. The pattern begins with `^` which anchors the match to the beginning of the variable name. This can be followed by zero or more characters (`*`) that do not match the underscore (`[^_]`)—the `*` specifies that the preceding pattern can be repeated zero or more times. The preceding pattern here is actually a list of characters included between square brackets. Since this list begins with `^` the listed characters are excluded from the matching. That is, the pattern preceding the `*` will match any character that is not an underscore. The third component of the match is then an actual underscore. Combined this regular expression matches any sequence (including an empty sequence) of characters (except for an underscore) that is at the beginning of the variable name and followed by an underscore.

The next argument to `stringr::str_replace()` is the replacement string. In this case we are replacing the matched pattern with an empty string.

The example here is simply one example of very many possible transformations we become used to in cleaning our datasets. The aim in transforming the variable names is to make then easier to use and to understand, both for ourselves and for others.

14 Data Review

Having ingested the dataset and an initial review, normalising the variable names, we are now ready to explore more. In particular, what do the data within the dataset look like. We again gain `tibble::glimpse()` into the dataset: *20180721*

```
# Review the dataset.

glimpse(ds)
```

```
## Observations: 145,463
## Variables: 24
## $ date            <date> 2008-12-01, 2008-12-02, 2008-12-03, 2008-12-04, 2...
## $ location        <chr> "Albury", "Albury", "Albury", "Albury", "Albury", ...
## $ min_temp        <dbl> 13.4, 7.4, 12.9, 9.2, 17.5, 14.6, 14.3, 7.7, 9.7, ...
## $ max_temp        <dbl> 22.9, 25.1, 25.7, 28.0, 32.3, 29.7, 25.0, 26.7, 31...
## $ rainfall        <dbl> 0.6, 0.0, 0.0, 0.0, 1.0, 0.2, 0.0, 0.0, 0.0, 1.4, ...
## $ evaporation     <chr> NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA...
## $ sunshine        <chr> NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA, NA...
## $ wind_gust_dir   <chr> "W", "WNW", "WSW", "NE", "W", "WNW", "W", "W", "NN...
## $ wind_gust_speed <int> 44, 44, 46, 24, 41, 56, 50, 35, 80, 28, 30, 31, 61...
## $ wind_dir_9am    <chr> "W", "NNW", "W", "SE", "ENE", "W", "SW", "SSE", "S...
## $ wind_dir_3pm    <chr> "WNW", "WSW", "WSW", "E", "NW", "W", "W", "W", "NW...
## $ wind_speed_9am  <int> 20, 4, 19, 11, 7, 19, 20, 6, 7, 15, 17, 15, 28, 24...
## $ wind_speed_3pm  <int> 24, 22, 26, 9, 20, 24, 24, 17, 28, 11, 6, 13, 28, ...
## $ humidity_9am    <int> 71, 44, 38, 45, 82, 55, 49, 48, 42, 58, 48, 89, 76...
## $ humidity_3pm    <int> 22, 25, 30, 16, 33, 23, 19, 19, 9, 27, 22, 91, 93,...
## $ pressure_9am    <dbl> 1007.7, 1010.6, 1007.6, 1017.6, 1010.8, 1009.2, 10...
## $ pressure_3pm    <dbl> 1007.1, 1007.8, 1008.7, 1012.8, 1006.0, 1005.4, 10...
## $ cloud_9am       <int> 8, NA, NA, NA, 7, NA, 1, NA, NA, NA, NA, 8, 8, NA,...
## $ cloud_3pm       <int> NA, NA, 2, NA, 8, NA, NA, NA, NA, NA, NA, 8, 8, 7,...
## $ temp_9am        <dbl> 16.9, 17.2, 21.0, 18.1, 17.8, 20.6, 18.1, 16.3, 18...
## $ temp_3pm        <dbl> 21.8, 24.3, 23.2, 26.5, 29.7, 28.9, 24.6, 25.5, 30...
## $ rain_today      <chr> "No", "No", "No", "No", "No", "No", "No", "No", "N...
## $ risk_mm         <dbl> 0.0, 0.0, 0.0, 1.0, 0.2, 0.0, 0.0, 0.0, 1.4, 0.0, ...
## $ rain_tomorrow   <chr> "No", "No", "No", "No", "No", "No", "No", "No", "Y...
```

Observe the variety of data types here, ranging from **Date** (`date`), through **character** (`chr`) and **numeric** (`dbl`).

The data mostly looks as expected though it is odd that `evaporation` and `sunshine` are identified as character. Probably because they seem to be all missing, at least in the first 10 or so observations. We begin question other aspects of the data too. For example, is `date` an ongoing sequence of days as it appears to be here? Does `location` have values other than `Albury`? What is the distribution of the different variables?

These are all questions we will start asking ourselves in the context of "living and breathing" our data. Our aim should be to gleam all we can about the data that we are dealing with. Data science is very much about understanding, not blindly processing. The excitement is in the discovery of patterns in the data and the narrative the data is seeking to tell.

15 Dataset Head and Tail

Datasets can be very large, with many observations (millions) and many variables (thousands). We can't be expected to browse through all of the observations and variables. Instead we might review the contents of the dataset using `utils::head()` and `utils::tail()` to consider the top six (by default) and the bottom six observations. *20180721*

```
# Review the first few observations.

head(ds) %>% print.data.frame()
```

```
##         date location min_temp max_temp rainfall evaporation sunshine
## 1 2008-12-01   Albury     13.4     22.9      0.6        <NA>     <NA>
## 2 2008-12-02   Albury      7.4     25.1      0.0        <NA>     <NA>
## 3 2008-12-03   Albury     12.9     25.7      0.0        <NA>     <NA>
## 4 2008-12-04   Albury      9.2     28.0      0.0        <NA>     <NA>
## 5 2008-12-05   Albury     17.5     32.3      1.0        <NA>     <NA>
## 6 2008-12-06   Albury     14.6     29.7      0.2        <NA>     <NA>
##   wind_gust_dir wind_gust_speed wind_dir_9am wind_dir_3pm wind_speed_9am
## 1             W              44            W          WNW             20
## 2           WNW              44          NNW          WSW              4
## 3           WSW              46            W          WSW             19
....
```

```
# Review the last few observations.

tail(ds) %>% print.data.frame()
```

```
##         date location min_temp max_temp rainfall evaporation sunshine
## 1 2018-07-25    Uluru      7.5     26.9        0        <NA>     <NA>
## 2 2018-07-26    Uluru      5.3     29.7        0        <NA>     <NA>
## 3 2018-07-27    Uluru      3.7     21.5        0        <NA>     <NA>
## 4 2018-07-28    Uluru      3.2     21.6        0        <NA>     <NA>
## 5 2018-07-29    Uluru      3.7     21.8        0        <NA>     <NA>
## 6 2018-07-30    Uluru      2.3     25.6        0        <NA>     <NA>
##   wind_gust_dir wind_gust_speed wind_dir_9am wind_dir_3pm wind_speed_9am
## 1             N              35         ESE          WNW              7
## 2            NW              67           E           NW              7
## 3           SSW              30         WSW          WSW              6
....
```

All the time we are building a picture of the data we are looking at. It is beginning to confirm that `location` has multiple values whilst `date` does appear to be a sequence for each location.

16 Random Observations

It is also useful to review some random observations from the dataset to provide a little more insight. Here we use dplyr::sample_n() to randomly select six rows from the dataset. *20180721*

```
# Review a random sample of observations.

sample_n(ds, size=6) %>% print.data.frame()
```

```
##         date     location min_temp max_temp rainfall evaporation sunshine
## 1 2010-09-29 AliceSprings     11.7     23.0      0.0           8     11.6
## 2 2018-07-20       Darwin     22.9     33.8      0.0         4.2      8.8
## 3 2014-08-29    Newcastle      9.9     18.0      2.4        <NA>     <NA>
## 4 2013-11-11    Melbourne      9.8     19.4      8.0         5.8      5.3
## 5 2010-04-25     Dartmoor      9.1     17.0      4.2         1.8      9.3
## 6 2013-04-18       Hobart      9.2     13.2      0.8           3      4.5
##   wind_gust_dir wind_gust_speed wind_dir_9am wind_dir_3pm wind_speed_9am
## 1            SE              48          SE         <NA>             22
## 2            SE              39         ENE            E              9
## 3          <NA>              NA          SE           SE              4
....
```


17 Characters

On ingesting the dataset into R we observe the variables identified (automatically) as having **character** base::class(). The expected values for such variables are strings of characters. We often call such variables *categoric* variables. Within R these are usually represented as a data type called **factor** and handled specially by many of the modelling algorithms. *20180723*

We can observe some meta data for each of the character variables. Let's first identify the character variables.

```
# Identify the character variables by index.

ds %>%
  sapply(is.character) %>%
  which() %T>%
  print() ->
chari
```

```
##      location   evaporation      sunshine wind_gust_dir  wind_dir_9am
##             2             6             7             8            10
##  wind_dir_3pm    rain_today rain_tomorrow
##            11            22            24
```

```
# Identify the chracter variables by name.

ds %>%
  names() %>%
  '['(chari) %T>%
  print() ->
charc
```

```
## [1] "location"      "evaporation"   "sunshine"      "wind_gust_dir"
## [5] "wind_dir_9am"  "wind_dir_3pm"  "rain_today"    "rain_tomorrow"
```

We will review each one of these in more detail so as to understand how we make use of them in our analyses. In particular we consider which of the variables might be handled as factors.

Where a **character** variable takes on a limited number of possible values we might convert the variable from **character** into **factor** (categoric) so as to take advantage of special handling of factors in R.

In fact, we think of a **factor** as a variable that can only take on a specific number of known distinct values which we call the ***levels*** of the **factor**.

18 Factors

20180908

For datasets that we load into R we will not always have examples of all possible levels of a factor. Consequently it is not always possible to automatically list all of the levels automatically. Be default the tidyverse ingests these variables as **character** so that we can take specific action to convert them to **factor** as required.

We first review the number of unique *levels* for each of the factors.

```
# Observe the unique levels.

ds[charc] %>% sapply(unique)
```

```
## $location
##  [1] "Albury"           "BadgerysCreek"    "Cobar"
##  [4] "CoffsHarbour"     "Moree"            "Newcastle"
##  [7] "NorahHead"        "NorfolkIsland"    "Penrith"
## [10] "Richmond"         "Sydney"           "SydneyAirport"
## [13] "WaggaWagga"       "Williamtown"      "Wollongong"
## [16] "Canberra"         "Tuggeranong"      "MountGinini"
## [19] "Ballarat"         "Bendigo"          "Sale"
## [22] "MelbourneAirport" "Melbourne"        "Mildura"
## [25] "Nhil"             "Portland"         "Watsonia"
## [28] "Dartmoor"         "Brisbane"         "Cairns"
## [31] "GoldCoast"        "Townsville"       "Adelaide"
## [34] "MountGambier"     "Nuriootpa"        "Woomera"
## [37] "Albany"           "Witchcliffe"      "PearceRAAF"
## [40] "PerthAirport"     "Perth"            "SalmonGums"
## [43] "Walpole"          "Hobart"           "Launceston"
## [46] "AliceSprings"     "Darwin"           "Katherine"
## [49] "Uluru"
##
## $evaporation
##   [1] NA     "12"   "14.8" "12.6" "10.8" "11.4" "11.2" "13"   "9.8"  "14.6"
##  [11] "11"   "12.8" "13.8" "16.4" "17.4" "16"   "13.6" "8"    "8.2"  "8.6"
##  [21] "14.2" "15.8" "16.2" "13.4" "14.4" "11.8" "15.6" "15.2" "11.6" "9.6"
##  [31] "6.6"  "0.6"  "6"    "3"    "2"    "5.2"  "9"    "10.2" "10"   "7.4"
##  [41] "8.4"  "9.2"  "9.4"  "12.4" "10.4" "7.2"  "6.8"  "7.6"  "4.4"  "6.4"
....
```

If we decide to convert all of these variables from character into factor, then we can do so using `base::factor()`.

```
# Convert all chracter variables to be factors.

ds[charc] %<>% map(factor)
```

We don't actually do so here as we will consider each character variable in turn to decide how to handle it, especially that we might observe that evaporation and sunshine appear to be numeric.

19 Location

From our review of the data so far we start to make some observations about the character variables. The first is `location`. We note that several locations were reported in the above exploration of the dataset. We can confirm the number of locations by counting the number of `data.table::unique()` values the variable has in the original dataset. *20180723*

```
# How many locations are represented in the dataset.

ds$location %>%
  unique() %>%
  length()
```

```
## [1] 49
```

We may not know in general what other locations we will come across in related datasets and we already have quite a collection of 49 locations. We will retain this variable as a character data type.

Here is a list of locations and their frequencies in the dataset.

```
ds$location %>%
  table()
```

```
## .
##         Adelaide           Albany           Albury     AliceSprings
##             3193             3040             3041             3041
##    BadgerysCreek         Ballarat         Bendigo         Brisbane
##             3010             3041             3041             3194
##           Cairns         Canberra           Cobar     CoffsHarbour
##             3041             3437             3010             3010
##         Dartmoor           Darwin       GoldCoast           Hobart
##             3010             3194             3041             3194
##        Katherine       Launceston       Melbourne MelbourneAirport
##             1579             3041             3194             3010
##          Mildura            Moree     MountGambier      MountGinini
##             3010             3010             3041             3041
##        Newcastle             Nhil       NorahHead    NorfolkIsland
##             3041             1579             3005             3010
##        Nuriootpa       PearceRAAF          Penrith            Perth
##             3009             3009             3040             3193
##     PerthAirport         Portland         Richmond             Sale
##             3009             3010             3010             3010
##       SalmonGums           Sydney    SydneyAirport       Townsville
##             2963             3345             3010             3041
##      Tuggeranong            Uluru       WaggaWagga          Walpole
##             3040             1579             3010             3006
##         Watsonia      Williamtown      Witchcliffe       Wollongong
....
```


20 Evaporation and Sunshine

The next two character variables are: `evaporation`, `sunshine`. It does seem odd that these would be character, expecting bpth to be numeric values. If we look at the top of the dataset we see they have missing values. *20180723*

```
# Note the character remaining variables to be dealt with.

head(ds$evaporation)

## [1] NA NA NA NA NA NA

head(ds$sunshine)

## [1] NA NA NA NA NA NA

# Review other random values.

sample(ds$evaporation, 8)

## [1] NA    NA    "5.4" NA    NA    NA    "1.4" "6.8"

sample(ds$sunshine, 8)

## [1] "8.3"  NA     "10.3" NA     "10.5" NA     "1.2"  NA
```

The heuristic used to determine the data type when ingesting data only looks at a subset of all the data before it determines the data type. In this case the early observations are all missing and so default to character which is general enough to capture all potential values. We need to convert the variables to numeric.

```
# Identify the vairables to process.

cvars <- c("evaporation", "sunshine")

# Check the current class of the variables.

ds[cvars] %>% sapply(class)

## evaporation    sunshine
## "character" "character"

# Convert to numeric.

ds[cvars] %<>% sapply(as.numeric)

# Review some random values.

sample(ds$evaporation, 10)

##  [1] 4.6 1.6 7.8  NA  NA  NA 5.6 4.8 3.0  NA

sample(ds$sunshine, 10)

##  [1]   NA 10.7   NA   NA   NA   NA   NA   NA  2.5   NA
```


21 Wind Directions

The three wind direction variables (`wind_gust_dir`, `wind_dir_9am`, `wind_dir_3pm`) are also identified as **character**. We review the distribution of values here with `dplyr::select()` identifying any variable that `tidyselect::contains()` the string `_dir` and then build a `base::table()` over those variables. *20180723*

```
# Review the distribution of observations across levels.

ds %>%
  select(contains("_dir")) %>%
  sapply(table)
```

```
##     wind_gust_dir wind_dir_9am wind_dir_3pm
## E            9179         9245         8461
## ENE          8164         7936         7849
## ESE          7483         7724         8539
## N            9310        11570         8798
## NE           7055         7670         8256
## NNE          6434         7995         6531
## NNW          6511         7787         7741
## NW           8028         8715         8609
## S            9209         8675         9889
## SE           9424         9305        10948
## SSE          9159         9085         9351
## SSW          8760         7533         8219
## SW           8934         8443         9256
## W            9778         8418        10065
## WNW          8265         7436         8867
## WSW          9040         6897         9508
```

Observe all 16 compass directions are represented and it would make sense to convert this into a factor. Notice that the directions are in alphabetic order and conversion to factor will retain that. Instead we can construct an ordered factor to capture the compass order (from `N`, `NNE`, to `NW` and `NNW`). We note the ordering of the directions here.

```
# Levels of wind direction are ordered compas directions.

compass <- c("N", "NNE", "NE", "ENE",
             "E", "ESE", "SE", "SSE",
             "S", "SSW", "SW", "WSW",
             "W", "WNW", "NW", "NNW")
```


22 Ordered Factor

Given our knowledge that compass directions have an obvious order, we convert the direction variables into an ordered factor. We do so using `ordered=TRUE` with `base::factor()`. *20180723*

```
# Note the names of the wind direction variables.

ds %>%
  select(contains("_dir")) %>%
  names() %T>%
  print() ->
vnames
```

```
## [1] "wind_gust_dir" "wind_dir_9am"  "wind_dir_3pm"
```

```
# Convert these variables from character to factor.

ds[vnames] %<>%
  lapply(factor, levels=compass, ordered=TRUE) %>%
  data.frame() %>%
  tbl_df()
# Confirm they are now factors.

ds[vnames] %>% sapply(class)
```

```
##      wind_gust_dir wind_dir_9am wind_dir_3pm
## [1,] "ordered"     "ordered"    "ordered"
## [2,] "factor"      "factor"     "factor"
```

We can again obtain a distribution of the variables to confirm that all we have changed is the data type.

```
# Verify the distribution has not changed.

ds %>%
  select(contains("_dir")) %>%
  sapply(table)
```

```
##     wind_gust_dir wind_dir_9am wind_dir_3pm
## N            9310        11570         8798
## NNE          6434         7995         6531
## NE           7055         7670         8256
## ENE          8164         7936         7849
## E            9179         9245         8461
## ESE          7483         7724         8539
## SE           9424         9305        10948
## SSE          9159         9085         9351
## S            9209         8675         9889
## SSW          8760         7533         8219
## SW           8934         8443         9256
....
```


23 Rain

The two remaining character variables are: `rain_today`, `rain_tomorrow`. Their distributions are generated by `dplyr::select()`ing from the dataset those variables that start with `rain_` and then build a `base::table()` over those variables. We use `base::sapply()` to apply `base::table()` to the selected columns to count the frequency of the occurrence of each value of a variable within the dataset. *20180723*

```
# Review the distribution of observations across levels.

ds %>%
  select(starts_with("rain_")) %>%
  sapply(table)
```

```
##     rain_today rain_tomorrow
## No      110981        110985
## Yes      31253         31250
```

Noting that `No` and `Yes` are the only values these two variables will take it makes sense to convert them both to factors. We will keep the ordering as alphabetic and so a simple call to `base::factor()` will to convert from character to factor.

```
# Note the names of the rain variables.

ds %>%
  select(starts_with("rain_")) %>%
  names() ->
vnames

# Confirm these are currently character variables.

ds[vnames] %>% sapply(class)
```

```
##    rain_today rain_tomorrow
##   "character"   "character"
```

```
# Convert these variables from character to factor.

ds[vnames] %<>%
  lapply(factor) %>%
  data.frame() %>%
  tbl_df()

# Confirm they are now factors.

ds[vnames] %>% sapply(class)
```

```
##    rain_today rain_tomorrow
##      "factor"      "factor"
```


24 Numeric

Summaries of numeric data are provided using base::summary(). In the following we identify the numeric variables and summarise each. In doing so, as a data scientist, we want to again observe any oddities and to explain them. *20180723*

```
ds %>%
  sapply(is.numeric) %>%
  which() %>%
  names %T>%
  print() ->
numi
```

```
##  [1] "min_temp"       "max_temp"       "rainfall"       "evaporation"
##  [5] "sunshine"       "wind_gust_speed" "wind_speed_9am"  "wind_speed_3pm"
##  [9] "humidity_9am"   "humidity_3pm"   "pressure_9am"   "pressure_3pm"
## [13] "cloud_9am"      "cloud_3pm"      "temp_9am"       "temp_3pm"
## [17] "risk_mm"
```

```
ds[numi] %>%
  summary()
```

```
##     min_temp        max_temp        rainfall        evaporation
##  Min.   :-8.70   Min.   :-4.10   Min.   :  0.000   Min.   : 0.00
##  1st Qu.: 7.40   1st Qu.:17.90   1st Qu.:  0.000   1st Qu.: 2.60
##  Median :11.90   Median :22.50   Median :  0.000   Median : 4.60
##  Mean   :12.04   Mean   :23.14   Mean   :  2.298   Mean   : 5.42
##  3rd Qu.:16.70   3rd Qu.:28.20   3rd Qu.:  0.600   3rd Qu.: 7.20
##  Max.   :33.90   Max.   :48.10   Max.   :371.000   Max.   :82.40
##  NA's   :1545    NA's   :1335    NA's   :3229      NA's   :64843
##     sunshine     wind_gust_speed  wind_speed_9am  wind_speed_3pm
##  Min.   : 0.00   Min.   :  2.00   Min.   : 0.00   Min.   : 0.00
##  1st Qu.: 4.90   1st Qu.: 31.00   1st Qu.: 7.00   1st Qu.:13.00
##  Median : 8.40   Median : 39.00   Median :13.00   Median :19.00
##  Mean   : 7.61   Mean   : 40.01   Mean   :14.01   Mean   :18.64
##  3rd Qu.:10.60   3rd Qu.: 48.00   3rd Qu.:19.00   3rd Qu.:24.00
##  Max.   :14.50   Max.   :135.00   Max.   :87.00   Max.   :87.00
##  NA's   :70820   NA's   :10667    NA's   :2081    NA's   :3407
....
```

Reviewing this information we can make some obvious observations. Temperatures, for example, appears to be in degrees Celsius rather than Fahrenheit. Rainfall looks like millimetres. There are some quite skewed distributions with min and median small but large max values. As data scientists we will further explore the distributions as in Chapter 5.

25 Logical

Above we converted `rain_today` and `rain_tomorrow` to factors. They have just two values as we confirm here, in addition to a small number of missing values (`NA`). *20180723*

```
ds %>%
  select(rain_today, rain_tomorrow) %>%
  summary()
```

```
##  rain_today    rain_tomorrow
##  No  :110981   No  :110985
##  Yes : 31253   Yes : 31250
##  NA's:  3229   NA's:  3228
```

As binary valued factors, and particularly as the values suggest, they are both candidates for being considered as logical variables (sometimes called Boolean). They can be treated as FALSE/TRUE instead of No/Yes and so supported directly by R as class **logical**. Different functions will then treat them as appropriate but not all functions do anything special. If this suits our purposes then the following can be used to perform the conversion to logical.

```
ds %<>%
  mutate(rain_today    = rain_today    == "Yes",
         rain_tomorrow = rain_tomorrow == "Yes")
```

Best to now check that the distribution itself has not changed.

```
ds %>%
  select(rain_today, rain_tomorrow) %>%
  summary()
```

```
##  rain_today      rain_tomorrow
##  Mode :logical   Mode :logical
##  FALSE:110319    FALSE:110316
##  TRUE :31880     TRUE :31877
##  NA's :3261      NA's :3267
```

Observe that the TRUE (`Yes`) values are much less frequent than the FALSE (`No`) values, and we also note the missing values.

The majority of days not having rain can be cross checked with the rainfall variable. In the previous summary of its distribution we note that rainfall has a median of zero, consistent with fewer days of actual rain. As data scientists we perform various cross checks on the hunt for oddities in the data.

As data scientists we will also want to understand why there are missing values. Is it simply some rare failures to capture the observation, or for example is there a particular location not recording rainfall? We would explore that now before moving on.

For our purposes going forward we will retain these two variables as factors. One reason for doing so is that we will illustrate missing value imputation using `randomForest::na.roughfix()` and this functoin does not handle logical data but keeping `rain_tomorrow` as character will allow missing value imputation. Of course we could skip this variable for the imputation.

26 Variable Roles

Now that we have a basic idea of the size and shape and contents of the dataset and have performed some basic data type identification and conversion we are in a position to identify the roles played by the variables within the dataset. First we will record the list of available variables so that we might reference them below. *20180723*

```
# Note the available variables.

vars <- names(ds) %T>% print()
```

```
##  [1] "date"           "location"       "min_temp"       "max_temp"
##  [5] "rainfall"       "evaporation"    "sunshine"       "wind_gust_dir"
##  [9] "wind_gust_speed" "wind_dir_9am"   "wind_dir_3pm"   "wind_speed_9am"
## [13] "wind_speed_3pm" "humidity_9am"   "humidity_3pm"   "pressure_9am"
## [17] "pressure_3pm"   "cloud_9am"      "cloud_3pm"      "temp_9am"
## [21] "temp_3pm"       "rain_today"     "risk_mm"        "rain_tomorrow"
```

By this stage of the project we will usually have identified a business problem that is the focus of attention. In our case we will assume it is to build a predictive analytics model to predict the chance of it raining tomorrow given the observation of today's weather. In this case the variable `rain_tomorrow` is the *target variable*. Given today's observations of the weather this is what we want to predict. The dataset we have is then a *training dataset* of historic observations. The task is to identify any patterns among the other observed variables that suggest that it rains the following day.

```
# Note the target variable.

target <- "rain_tomorrow"

# Place the target variable at the beginning of the vars.

vars <- c(target, vars) %>% unique() %T>% print()
```

```
##  [1] "rain_tomorrow"  "date"           "location"       "min_temp"
##  [5] "max_temp"       "rainfall"       "evaporation"    "sunshine"
##  [9] "wind_gust_dir"  "wind_gust_speed" "wind_dir_9am"   "wind_dir_3pm"
## [13] "wind_speed_9am" "wind_speed_3pm" "humidity_9am"   "humidity_3pm"
## [17] "pressure_9am"   "pressure_3pm"   "cloud_9am"      "cloud_3pm"
## [21] "temp_9am"       "temp_3pm"       "rain_today"     "risk_mm"
....
```

We have taken the opportunity here to move the target variable to be the first in the vector of variables recorded in `vars`. This is common practice where the first variable in a dataset is the target (dependent variable) and the remainder are the variables (the independent variables) that will be used to build a model to predict that target.

27 Risk Variable

20180723

With some knowledge of the data we observe `risk_mm` captures the amount of rain recorded tomorrow. We refer to this as a *risk variable*, being a measure of the impact or risk of the target we are predicting (rain tomorrow). The risk is an output variable and should not be used as an input to the modelling—it is not an independent variable. In other circumstances it might actually be treated as the target variable.

```
# Note the risk variable - measures the severity of the outcome.

risk <- "risk_mm"
```

For this risk variable note that we expect it to have a value of 0 for all observations when the target variable has the value `No`.

```
# Review the distribution of the risk variable for non-targets.

ds %>%
  filter(rain_tomorrow == "No") %>%
  select(risk_mm) %>%
  summary()
```

```
##     risk_mm
##  Min.   :0.00000
##  1st Qu.:0.00000
##  Median :0.00000
##  Mean   :0.07397
##  3rd Qu.:0.00000
##  Max.   :1.00000
```

Interestingly, even a little rain (defined as 1mm or less) is regarded as no rain. That is useful to keep in mind and is a discovery of the data that we might not have expected. As data scientists we should be expecting to find the unexpected.

A similar analysis for the target observations is more in line with expectations.

```
# Review the distribution of the risk variable for targets.

ds %>%
  filter(rain_tomorrow == "Yes") %>%
  select(risk_mm) %>%
  summary()
```

```
##     risk_mm
##  Min.   :  1.10
##  1st Qu.:  2.40
##  Median :  5.20
##  Mean   : 10.19
##  3rd Qu.: 11.60
##  Max.   :371.00
```


28 ID Variables

From our observations so far we note that the variable (`date`) acts as an identifier as does the variable (`location`). Given a `date` and a `location` we have an observation of the remaining variables. Thus we note that these two variables are so-called identifiers. Identifiers would not usually be used as independent variables for building predictive analytics models. *20180723*

```
# Note any identifiers.

id <- c("date", "location")
```

We might get a sense of how this works with the following which will list a random sample of locations and how long the observations for that location have been collected.

```
ds[id] %>%
  group_by(location) %>%
  count() %>%
  rename(days=n) %>%
  mutate(years=round(days/365)) %>%
  as.data.frame() %>%
  sample_n(10)
```

```
##       location days years
## 9       Cairns 3041     8
## 13    Dartmoor 3010     8
## 25   Newcastle 3041     8
## 32       Perth 3193     9
## 45    Watsonia 3010     8
....
```

The data for each location ranges in length from 4 years up to 9 years, though most have 8 years of data.

```
ds[id] %>%
  group_by(location) %>%
  count() %>%
  rename(days=n) %>%
  mutate(years=round(days/365)) %>%
  ungroup() %>%
  select(years) %>%
  summary()
```

```
##      years
##  Min.   :4.000
##  1st Qu.:8.000
##  Median :8.000
##  Mean   :7.918
##  3rd Qu.:8.000
##  Max.   :9.000
```


29 Ignore IDs and Outputs

The identifiers and any risk variable (which is an output variable) should be ignored in any predictive modelling. Always watch out for treating output variables as inputs to modelling—this is a surprisingly common trap for beginners. We will build a vector of the names of the variables to ignore. Above we have already recorded the `id` variables and (optionally) the `risk`. Here we join them together into a new vector using `data.table::union()` which performs a set union operation—that is, it joins the two arguments together and removes any repeated variables. *20180723*

```
# Initialise ignored variables: identifiers and risk.

ignore <- union(id, risk) %T>% print()
```

```
## [1] "date"     "location" "risk_mm"
```

We might also check for any variable that has a unique value for every observation. These are often identifiers and if so they are candidates for ignoring. We select the `vars` from the dataset and pipe through to `base::sapply()` for any variables having only unique values. In our case there are no further candidate identifiers. as indicated by the empty result, `character(0)`.

```
# Heuristic for candidate indentifiers to possibly ignore.

ds[vars] %>%
  sapply(function(x) x %>% unique() %>% length()) %>%
  equals(nrow(ds)) %>%
  which() %>%
  names() %T>%
  print() ->
ids
```

```
## character(0)
```

```
# Add them to the variables to be ignored for modelling.

ignore <- union(ignore, ids) %T>% print()
```

```
## [1] "date"     "location" "risk_mm"
```


30 Ignore Missing

We next remove any variable where all of the values are missing. There are none like this in the weather dataset but in general for other datasets with thousands of variables there may be some. Here we first count the number of missing values for each variable and then list the names of those variables that have no values. *20180723*

```
# Identify variables with only missing values.

ds[vars] %>%
  sapply(function(x) x %>% is.na %>% sum) %>%
  equals(nrow(ds)) %>%
  which() %>%
  names() %T>%
  print() ->
missing
```

```
## character(0)
```

```
# Add them to the variables to be ignored for modelling.

ignore <- union(ignore, missing) %T>% print()
```

```
## [1] "date"     "location" "risk_mm"
```

It is also useful to identify those variables which are very sparse—that have mostly missing values. We can decide on a threshold of the proportion missing above which to ignore the variable as not likely to add much value to our analysis. For example, we may want to ignore variables with more than 70% of the values missing:

```
# Identify a threshold above which proportion missing is fatal.

missing.threshold <- 0.7

# Identify variables that are mostly missing.

ds[vars] %>%
  sapply(function(x) x %>% is.na() %>% sum()) %>%
  '>'(missing.threshold*nrow(ds)) %>%
  which() %>%
  names() %T>%
  print() ->
mostly
```

```
## character(0)
```

```
# Add them to the variables to be ignored for modelling.

ignore <- union(ignore, mostly) %T>% print()
```

```
## [1] "date"     "location" "risk_mm"
```


31 Ignore Excessive Level Variables

Another issue we traditionally come across in our datasets are those factors with very many levels. This is more common when we read data as factors rather than as character, and so this step depends on where the data has come from. Nonetheless We might want to check for and ignore such variables. *20180723*

```
# Identify a threshold above which we have too many levels.

levels.threshold <- 20

# Identify variables that have too many levels.

ds[vars] %>%
  sapply(is.factor) %>%
  which() %>%
  names() %>%
  sapply(function(x) ds %>% extract2(x) %>% levels() %>% length()) %>%
  '>='(levels.threshold) %>%
  which() %>%
  names() %T>%
  print() ->
too.many
```

```
## character(0)
```

```
# Add them to the variables to be ignored for modelling.

ignore <- union(ignore, too.many) %T>% print()
```

```
## [1] "date"     "location" "risk_mm"
```


32 Ignore Constants

We also ignore variables with constant values as they add no extra information to the analysis. *20180723*

```
# Identify variables that have a single value.

ds[vars] %>%
  sapply(function(x) all(x == x[1L])) %>%
  which() %>%
  names() %T>%
  print() ->
constants
```

```
## character(0)
```

```
# Add them to the variables to be ignored for modelling.

ignore <- union(ignore, constants) %T>% print()
```

```
## [1] "date"     "location" "risk_mm"
```


33 Correlated Variables for Numerics

It is often useful to identify highly correlated variables. Such variables will often record the same information but in different ways and often arise when we combine data from different sources. *20180726*

We identify the numeric variables of a dataset by `base::sapply()`ing the function `base::is.numeric()` to find `base::which()` are numeric. Their integer column positions are stored into the variable `numi`.

```
# Note which variables are numeric.

vars %>%
  setdiff(ignore) %>%
  extract(ds, .) %>%
  sapply(is.numeric) %>%
  which() %>%
  names() %T>%
  print() ->
numc
```

```
##  [1] "min_temp"       "max_temp"       "rainfall"       "evaporation"
##  [5] "sunshine"       "wind_gust_speed" "wind_speed_9am" "wind_speed_3pm"
##  [9] "humidity_9am"   "humidity_3pm"   "pressure_9am"   "pressure_3pm"
## [13] "cloud_9am"      "cloud_3pm"      "temp_9am"       "temp_3pm"
```


34 Calculating Correlations

20180726

The correlation is calculated by dplyr::**select()**ing the numeric columns from the dataset and passing that through to stats::**cor()**. This matrix of pairwise correlations is based on only the complete observations so that observations with missing values are ignored.

We set the upper triangle of the correlation matrix to **NA**'s as they are a mirror of the values in the lower triangle and thus redundant. We also set **diag=TRUE** to set the diagonals as **NA** since they will always be perfect correlations.

The processing continues by making all values positive using base::**abs()**. With conversion to base::**data.frame()** then to dplyr::**tbl_df()** the dataset column names need to be reset appropriately using magrittr::**set_colnames()**. We plyr::**mutate()** the dataset with a new column using plyr::**mutate()**, reshape the dataset using tidyr::**gather()** from tidyr and then omit missing correlations using data.table::**na.omit()**. Finally the rows are plyr::**arrange()**'d with the highest absolute correlations appearing first.

```
# For the numeric variables generate a table of correlations

ds[numc] %>%
  cor(use="complete.obs") %>%
  ifelse(upper.tri(., diag=TRUE), NA, .) %>%
  abs %>%
  data.frame %>%
  tbl_df %>%
  set_colnames(numc) %>%
  mutate(var1=numc) %>%
  gather(var2, cor, -var1) %>%
  na.omit %>%
  arrange(-abs(cor)) %T>%
  print() ->
mc
```

```
## # A tibble: 120 x 3
##    var1           var2             cor
##    <chr>          <chr>          <dbl>
##  1 temp_3pm       max_temp       0.985
##  2 pressure_3pm   pressure_9am   0.962
##  3 temp_9am       min_temp       0.908
....
```

35 Dealing with Correlations

From the final result we can identify pairs of variables where we might want to keep one but not the other variable because they are highly correlated. We will select them manually since it is a judgement call. Normally we might limit the removals to those correlations that are 0.90 or more. In our case here the three pairs of highly correlated variables make intuitive sense. *20180726*

```
# Note the correlated variables that are redundant.

correlated <- c("temp_3pm", "pressure_3pm", "temp_9am")

# Add them to the variables to be ignored for modelling.

ignore <- union(ignore, correlated) %T>% print()
```

```
## [1] "date"         "location"     "risk_mm"      "temp_3pm"     "pressure_...
## [6] "temp_9am"
```


36 Removing Ignored Variables

Once we have identified all of the variables to ignore we remove them from our list of variables to use. *20180726*

```
# Check the number of variables currently.

length(vars)

## [1] 24

# Remove the variables to ignore.

vars <- setdiff(vars, ignore)

# Confirm they are now ignored.

length(vars)

## [1] 18
```


37 Feature Selection

The FSelector package provides functions to identify subsets of variables that might be more effective for modelling. We can use this (and other packages) to assist us in reducing the variables that will be useful in our modelling. As we find useful functionality we will add them to our standard template so that for our next dataset we have the functionality readily available. *20180726*

We first use `FSelector::cfs()` to identify a good subset of variables using correlation and entropy. We then list the variable importance using `FSelector::information.gain()` to advise a useful subset of variables. Note that the `stringi::%s+%` operator is a convenience to concatenate strings together to produce a formula that indicates we will model the target variable using all of the other variables of the dataset.

```
# Construct the formulation of the modelling we plan to do.

form <- formula(target %s+% " ~ .") %T>% print()

## rain_tomorrow ~ .

# Use correlation search to identify key variables.

cfs(form, ds[vars])

## [1] "rainfall"     "sunshine"     "humidity_3pm" "cloud_3pm"    "rain_today"

# Use information gain to identify variable importance.

information.gain(form, ds[vars])

##                 attr_importance
## min_temp            0.005965086
## max_temp            0.013679399
## rainfall            0.058867687
## evaporation         0.005233566
## sunshine            0.055563157
## wind_gust_dir       0.006082875
## wind_gust_speed     0.027066589
## wind_dir_9am        0.008660761
## wind_dir_3pm        0.004830997
## wind_speed_9am      0.004388310
## wind_speed_3pm      0.005426372
## humidity_9am        0.037671301
## humidity_3pm        0.111123203
## pressure_9am        0.028254633
## cloud_9am           0.035295285
## cloud_3pm           0.051036499
## rain_today          0.047266724
```

The two measures are consistent in this case in that the variables identified by `FSelector::cfs()` are the more important variables identified by `FSelector::information.gain()`.

38 Missing Targets

Sometimes there may be further operations to perform on the dataset prior to modelling. A common task is to deal with missing values. Here we remove observations with a missing target. As with any missing data we should also analyse whether there is any pattern to the missing targets. This may be indicative of a systemic data issue rather than simply randomly missing values. *20180726*

```
# Check the dimensions to start with.

dim(ds)

## [1] 145463     24

# Identify observations with a missing target.

missing.target <- ds %>% extract2(target) %>% is.na()

# Check how many are found.

sum(missing.target)

## [1] 3228

# Remove observations with a missing target.

ds %<>% filter(!missing.target)

# Confirm the filter delivered the expected dataset.

dim(ds)

## [1] 142235     24
```


39 Missing Values

20180726

Missing values for the variables are an issue for some but not all algorithms. For example randomForest::**randomForest()** omits observations with missing values by default whilst rpart::**rpart()** has a particularly well developed approach to dealing with missing values.

We may want to impute missing values in the data (though it is not always wise to do so). Here we do this using randomForest::**na.roughfix()** from randomForest. This function provides, as the name implies, a rather basic algorithm for imputing missing values. Because of this we will demonstrate the process but then restore the original dataset—we will not want this imputation to be included in our actual dataset.

```
# Backup the dataset so we can restore it as required.

ods <- ds
```

```
# Count the number of missing values.

ds[vars] %>%  is.na() %>% sum()

## [1] 306730

# Impute missing values.

ds[vars] %<>% na.roughfix()

# Confirm that no missing values remain.

ds[vars] %>%  is.na() %>% sum()

## [1] 0
```

As foreshadowed we now restore the dataset with its original contents.

```
# Restore the original dataset.

ds <- ods
```


40 Omitting Observations

An alternative is to remove observations that have missing values. Here `data.table::na.omit()` identifies the rows to omit based on the `vars` to be included for modelling. The list of rows to omit is stored as the `na.action` attribute of the returned object. We then remove these observations from the dataset. *20180726*

Notice we keep a copy of the original dataset and then restore it.

```
# Backup the dataset so we can restore it as required.

ods <- ds

# Initialise the list of observations to be removed.

omit <- NULL
```

```
# Review the current dataset.

ds[vars] %>% nrow()
## [1] 142235
ds[vars] %>% is.na() %>% sum()
## [1] 306730
# Identify any observations with missing values.

mo <- attr(na.omit(ds[vars]), "na.action")

# Record the observations to omit.

omit <- union(omit, mo)

# If there are observations to omit then remove them.

if (length(omit)) ds <- ds[-omit,]

# Confirm the observations have been removed.

ds[vars] %>% nrow()
## [1] 55650
ds[vars] %>% is.na() %>% sum()
## [1] 0
```

```
# Restore the original dataset.

ds <- ods
```


41 Normalise Factors

Some variables will have levels with spaces, and mixture of cases, etc. We may like to normalise the levels for each of the categoric variables. For very large datasets this can take some time and so we may want to be selective. *20180726*

```
# Note which variables are categoric.

ds %>%
  sapply(is.factor) %>%
  which() ->
catc

# Normalise the levels of all categoric variables.

for (v in catc)
  levels(ds[[v]]) %<>% normVarNames()
```


42 Target as a Factor

We often build classification models. For such models we want to ensure the target is categoric. Often it is 0/1 and hence is loaded as numeric. We could tell our model algorithm of choice to explicitly do classification or else set the target using base::as.factor() in the formula. Nonetheless it is generally cleaner to do this here and note that this code has no effect if the target is already categoric. *20180726*

```
# Ensure the target is categoric.

ds[[target]] %<>% as.factor()

# Confirm the distribution.

ds[target] %>% table()
```

```
## .
##     no    yes
## 110985  31250
```

We can visualise the distribution of the target variable using ggplot2. The dataset is piped to ggplot2::ggplot() whereby the target is associated through ggplot2::aes_string() (the aesthetics) with the x-axis of the plot. To this we add a graphics layer using ggplot2::geom_bar() to produce the bar chart, with bars having width= 0.2 and a fill= color of "grey". The resulting plot can be seen in Figure 1.

```
ds %>%
  ggplot(aes_string(x=target)) +
  geom_bar(width=0.2, fill="grey") +
  theme(text=element_text(size=14))
```

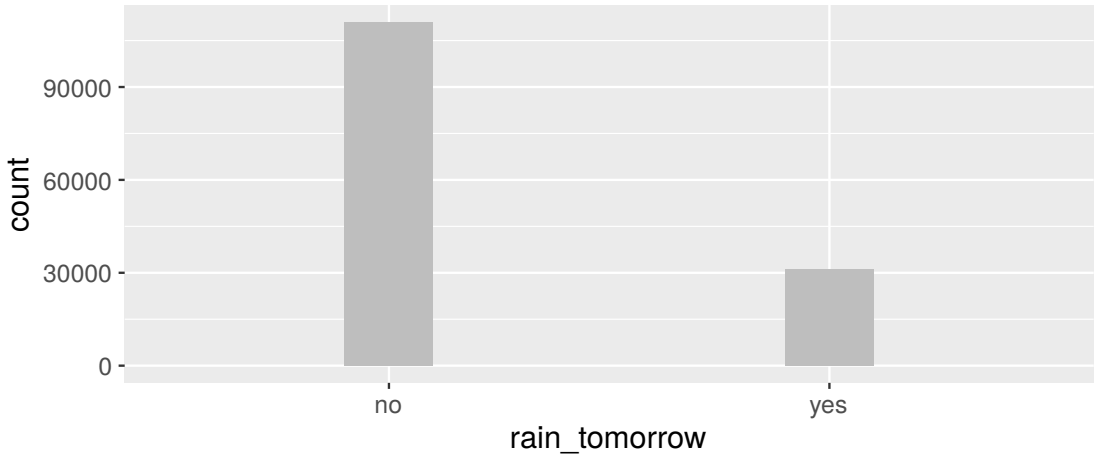


Figure 1: Target variable distribution. Plotting the distribution is useful to gain an insight into the number of observations in each category. As is the case here we often see a skewed distribution.

43 Identify Variable Types

Metadata is data about the data. We now record data about our dataset that we use later in further processing and analysing our data. In one sense the metadata is simply a convenient store. *20180726*

We identify the variables that will be used to build analytic models that provide different kinds of insight into our data. Above we identified the variable roles such as the target, a risk variable and the ignored variables. From an analytic modelling perspective we identify variables that are the model inputs. We record then both as a vector of characters (the variable names) and a vector of integers (the variable indicies).

```
inputs <- setdiff(vars, target) %T>% print()
```

```
##  [1] "min_temp"       "max_temp"       "rainfall"       "evaporation"
##  [5] "sunshine"       "wind_gust_dir"  "wind_gust_speed" "wind_dir_9am"
##  [9] "wind_dir_3pm"   "wind_speed_9am" "wind_speed_3pm" "humidity_9am"
## [13] "humidity_3pm"   "pressure_9am"   "cloud_9am"      "cloud_3pm"
## [17] "rain_today"
```

The integer indices are determined from the `base::names()` of the variables in the original dataset. Note the use of `USE.NAMES=` from `base::sapply()` to turn off the inclusion of names in the resulting vector to keep the result as a simple vector.

```
inputi <- sapply(inputs,
                 function(x) which(x == names(ds)),
                 USE.NAMES=FALSE)
inputi
```

```
##  [1]  3  4  5  6  7  8  9 10 11 12 13 14 15 16 18 19 22
```

For convenience we record the number of observations:

```
nobs <- nrow(ds) %T>% print()
```

```
## [1] 142235
```

Here we simply report on the dimensions of various data subsets primarily to confirm the dataset appear as we expect:

```
dim(ds)
```

```
## [1] 142235     24
```

```
dim(ds[vars])
```

```
## [1] 142235     18
```

```
dim(ds[inputs])
```

```
## [1] 142235     17
```


44 Identify Numeric and Categoric Variables

Identifying numeric and categoric variables may be useful for example for cluster analysis algorithms that only deal with numeric variables. Here we identify them by name (a character string) and by index. When using the index we have to assume the variables remain in the same order within the dataset and all variables are present, otherwise the indicies will get out of sync. *20180726*

```
# Identify the numeric variables by index.

ds %>%
  sapply(is.numeric) %>%
  which() %>%
  intersect(inputi) %T>%
  print() ->
numi
```

```
##  [1]  3  4  5  6  7  9 12 13 14 15 16 18 19
```

```
# Identify the numeric variables by name.

ds %>%
  names() %>%
  extract(numi) %T>%
  print() ->
numc
```

```
##  [1] "min_temp"       "max_temp"       "rainfall"       "evaporation"
##  [5] "sunshine"       "wind_gust_speed" "wind_speed_9am"  "wind_speed_3pm"
##  [9] "humidity_9am"   "humidity_3pm"   "pressure_9am"   "cloud_9am"
## [13] "cloud_3pm"
```

```
# Identify the categoric variables by index and then name.

ds %>%
  sapply(is.factor) %>%
  which() %>%
  intersect(inputi) %T>%
  print() ->
cati
```

```
## [1]  8 10 11 22
```

```
ds %>%
  names() %>%
  extract(cati) %T>%
  print() ->
catc
```

```
## [1] "wind_gust_dir" "wind_dir_9am"  "wind_dir_3pm"  "rain_today"
```


45 Save the Dataset

For large datasets we may want to save it to a binary RData file once we have wrangled it into the right shape and collected the metadata. Loading a binary dataset is generally quicker than loading a CSV file—a CSV file with 2 million observations and 800 variables can take 30 minutes to `utils::read.csv()`, 5 minutes to `base::save()`, and 30 seconds to `base::load()`.

```
# Timestamp for the dataset.

dsdate  <- "_" %s+% format(Sys.Date(), "%y%m%d") %T>% print()

## [1] "_180908"

# Filename for the saved dataset

dsrdata <- dsname %s+% dsdate %s+% ".RData" %T>% print()

## [1] "weather_180908.RData"

# Save relevant R objects to binary RData file.

save(ds, dsname, dspath, dsdate, nobs,
     vars, target, risk, id, ignore, omit,
     inputi, inputs, numi, numc, cati, catc,
     file=dsrdata)
```

Notice that in addition to the dataset (`ds`) we also store the collection of ***metadata***. This begins with items such as the name of the dataset, the source file path, the date we obtained the dataset, the number of observations, the variables of interest, the target variable, the name of the risk variable (if any), the identifiers, the variables to ignore and observations to omit. We continue with the indicies of the input variables and their names, the indicies of the numeric variables and their names, and the indicies of the categoric variables and their names.

Each time we wish to use the dataset we can now simply `base::load()` it into R. The value that is invisibly returned by `base::load()` is a vector naming the R objects loaded from the binary RData file.

```
load(dsrdata) %>% print()

##  [1] "ds"     "dsname" "dspath" "dsdate" "nobs"   "vars"   "target" "risk"
##  [9] "id"     "ignore" "omit"   "inputi" "inputs" "numi"   "numc"   "cati"
## [17] "catc"
```

We place the call to `base::load()` within a call to `(` (i.e., we have surrounded the call with round brackets) to ensure the result of the function call is printed. A call to `base::load()` returns its result invisibly since we are primarily interested in its side-effect. The side-effect is to read to R binary data from disk and to make it available within our current R session.

46 A Template for Data Preparation

Through this chapter we have built a template for data preparation. An actual knitr template based on this chapter for data preparation is available as http://HandsOnDataScience.com/scripts/data.Rnw. An automatically derived version including just the R code is also available as http://HandsOnDataScience.com/scripts/data.R. Notice that we would not necessarily perform all of the steps, such as normalising the variable names, imputing missing values, omitting observations with missing values, and so on. Instead we pick and choose as is appropriate to our situation and specific datasets. Also, some data specific transformations are not included in the template and there may be other transforms we need to perform that we have not covered here. As we discover new tools to support the data scientist we can add them into our own templates.

47 Command Summary

This chapter has introduced, demonstrated and described the following R packages, functions, commands, operators, and datasets:

get *Function from base.* Returns the named dataset.

glimpse *Function from tibble.* Summarise a dataset.

head *Function from utils.* Display the first few rows of a dataset.

names *Function from base.* Column names of a dataset.

normVarNames *Function from rattle.* Normalize variable names.

ncol *Function from base.* Number of columns in a dataset.

nrow *Function from base.* Number of rows in a dataset.

read_csv *Function from readr.* Load data from a CSV file.

sample_n *Function from dplyr.* Random sample of n rows.

sapply *Function from base.* Apply a function to columns of a dataset.

str_replace *Function from stringr.* Replace a string with another.

system.file *Function from base.* Locate a system file.

tail *Function from utils.* Display the last few rows of a dataset.

48 Exercises

Exercise 1 Exploring the Weather

We have worked with the Australian **weatherAUS** dataset throughout this chapter. For this exercise we will explore the dataset further. For each exercise, extend the knitr document with the analyses performed.

1. Create a data preparation script beginning with the template available from http://HandsOnDataScience.com/scripts/data.Rnw and replicating the data processing performed in this chapter.
2. Investigate the `dplyr::group_by()` and `plyr::summarise()` functions, combined through a pipeline using `dplyr::%>%` to identify regions with considerable variance in their weather observations. The use of `stats::var()` might be a good starting point.

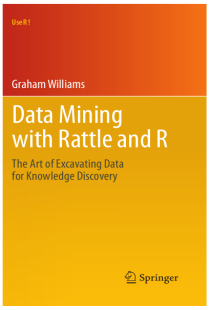
Exercise 2 Understanding Ferries

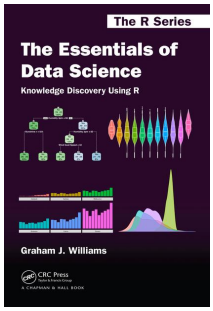
A dataset of ferry crossings on Sydney Harbour is available as http://HandsOnDataScience.com/data/ferry.csv. The original source of the Ferry dataset is http://www.bts.nsw.gov.au/Statistics/Ferry/default.aspx?FolderID=224. The dataset is available under a Creative Commons Attribution (CC BY 3.0 AU) license. We will use this dataset to exercise our data template.

1. Create a data preparation script beginning with the template available from http://HandsOnDataScience/scripts/data.R.
2. Change the sample source dataset within the template to download the ferry dataset into R.
3. Rename the variables to become normalized variable names.
4. Create two new variables from `sub_route`, called `origin` and `destination`.
5. Convert dates.
6. Convert appropriate variables into factors.
7. Work through the template to explore and prepare the dataset.
8. Develop some visualisations.

49 Further Reading

The Rattle book (Williams, 2011), published by Springer, provides a comprehensive introduction to data mining and analytics using Rattle and R. It is available from Amazon. Rattle provides a graphical user interface through which the user is able to load, explore, visualise, and transform data, and to build, evaluate, and export models. Through its Log tab it specifically aims to provide an R template which can be exported and serve as the starting point for further programming with data in R.





The Essentials of Data Science book (Williams, 2017), published by CRC Press, provides a comprehensive introduction to data science through programming with data using R. It is available from Amazon. The book provides a template based approach to doing data science and knowledge discovery. Templates are provided for data wrangling and model building. These serve as generic starting points for programming with data, and are designed to require minimal effort to get started. Visit `https://essentials.togaware.com` for further guides and templates.

50 References

Bache SM, Wickham H (2014). *magrittr: A Forward-Pipe Operator for R*. R package version 1.5, URL https://CRAN.R-project.org/package=magrittr.

Breiman L, Cutler A, Liaw A, Wiener M (2018). *randomForest: Breiman and Cutler's Random Forests for Classification and Regression*. R package version 4.6-14, URL https://CRAN.R-project.org/package=randomForest.

Gagolewski M, Tartanus B, , other contributors; IBM, other contributors; Unicode, Inc (2018). *stringi: Character String Processing Facilities*. R package version 1.2.4, URL https://CRAN.R-project.org/package=stringi.

Hester J (2018). *glue: Interpreted String Literals*. R package version 1.3.0, URL https://CRAN.R-project.org/package=glue.

R Core Team (2018). *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria. URL https://www.R-project.org/.

Romanski P, Kotthoff L (2018). *FSelector: Selecting Attributes*. R package version 0.31, URL https://CRAN.R-project.org/package=FSelector.

Spinu V, Grolemund G, Wickham H (2018). *lubridate: Make Dealing with Dates a Little Easier*. R package version 1.7.4, URL https://CRAN.R-project.org/package=lubridate.

Wickham H (2018). *stringr: Simple, Consistent Wrappers for Common String Operations*. R package version 1.3.1, URL https://CRAN.R-project.org/package=stringr.

Wickham H, Chang W, Henry L, Pedersen TL, Takahashi K, Wilke C, Woo K (2018a). *ggplot2: Create Elegant Data Visualisations Using the Grammar of Graphics*. R package version 3.0.0, URL https://CRAN.R-project.org/package=ggplot2.

Wickham H, François R, Henry L, Müller K (2018b). *dplyr: A Grammar of Data Manipulation*. R package version 0.7.6, URL https://CRAN.R-project.org/package=dplyr.

Wickham H, Henry L (2018). *tidyr: Easily Tidy Data with 'spread()' and 'gather()' Functions*. R package version 0.8.1, URL https://CRAN.R-project.org/package=tidyr.

Wickham H, Hester J, Francois R (2017). *readr: Read Rectangular Text Data*. R package version 1.1.1, URL https://CRAN.R-project.org/package=readr.

Williams GJ (2009). "Rattle: A Data Mining GUI for R." *The R Journal*, **1**(2), 45–55. URL http://journal.r-project.org/archive/2009-2/RJournal_2009-2_Williams.pdf.

Williams GJ (2011). *Data Mining with Rattle and R: The art of excavating data for knowledge discovery*. Use R! Springer, New York.

Williams GJ (2017). *The Essentials of Data Science: Knowledge discovery using R*. The R Series. CRC Press.

Williams GJ (2018). *rattle: Graphical User Interface for Data Science in R*. R package version 5.2.1, URL https://rattle.togaware.com/.

This document, sourced from DataO.Rnw bitbucket revision 291, was processed by KnitR version 1.20 of 2018-02-20 10:11:46 UTC and took 28.7 seconds to process. It was generated by gjw on Ubuntu 18.04.1 LTS.

