3

Data Wrangling

When we begin with a new data science project we will begin with understanding the business problem to be tackled. This includes ensuring all participants in the project understand the goals, the success criteria, and how the results will be deployed into production in the business. We then liaise with the business data technicians to identify the available data. This is followed by a data phase where we work with the business data technicians to access and ingest the data into R. We are then in a position to move on with our journey to the discovery of new insights 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 and then capture a common series of steps that we follow as the data phase of data science. As we progress through the chapter we will build a ***template*** * designed to be reused for journeys through other datasets. As we foreshadowed in Chapter 2 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.

As we will for each chapter we begin with loading the packages required for this chapter. Packages used in this chapter include FSelector (Romanski and Kotthoff, 2016), dplyr (Wickham *et al.*, 2017a), ggplot2 (Wickham and Chang, 2016), lub- ridate (Grolemund *et al.*, 2016), randomForest (Breiman *et al.*, 2015), rattle (Williams, 2017), readr (Wickham *et al.*, 2017b), scales (Wickham, 2016), stringi (Gagolewski *et al.*, 2017), stringr (Wickham, 2017a), tibble (Müller and Wickham, 2017), tidyr (Wickham, 2017b) and magrittr (Bache and Wickham, 2014).

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

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

3.1 Data Ingestion

To begin the data phase of a project we typically need to ingest data into R. For our purposes we will ingest the data from the simplest of sources—a text-based CSV (comma separate value) file. Practically any source format is supported by R through the many packages available. We can find, for example, support for ingesting data directly from Excel spreadsheets or from database

servers.The **weatherAUS** dataset from rattle will serve to illustrate the preparation of data for modelling. Both a CSV file and an R dataset are provided by the package and the dataset is also available directly from the Internet on the rattle web site. We will work with the CSV file as the typical pathway for loading data into R.

Identifying the Data Source

We first identify and record the location of the CSV file to analyse. R is capable of loading data directly from the Internet and so we will illustrate how to load the CSV file from the rattle web site itself. The location of the file (the so-called URL or universal resource locator) will be saved as a string of characters in a variable called dspath—the path to the dataset. This is achieved through the following assignment which we type into our R script file within RStudio. The command is then executed in RStudio by clicking the Run button whilst the cursor is located on the line of code within the script file.

```
# Identify the source location of the dataset.

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

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

Reading the Data

Having identified the source of the dataset we can read the dataset into the memory of the computer using readr::**read_csv()**. This function returns a ***data frame*** (though it is actually an enhanced data frame known as a ***table data frame***) which is the basic data structure used to store a dataset within R. The data is stored as a table consisting of rows (***observations***) and columns (***variables***). We store the dataset (as a data frame) in the computer's memory and reference it by the R variable weatherAUS. It will then be ready to process using R.

```
# Ingest the dataset.

weatherAUS <- 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.
```

Template Variables

To support our goal of creating a reusable template we create a reference to the original dataset using a template (or generic) variable. The new variable will be called ds (short for dataset).

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

ds <- weatherAUS
```

Both ds and weatherAUS will initially reference the same dataset within the computer's memory. As we modify ds those modifications will only affect the data referenced by ds and not the data referenced by weatherAUS. 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. * From here on we no longer refer to the dataset as weatherAUS but as ds. This allows the following steps to be generic—turning the R code into a *template* 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 following code remains essentially unchanged.

The next few steps of our template record the name of the dataset and a generic reference to the dataset.

```
# Prepare the dataset for usage with our template.

dsname <- "weatherAUS"
ds     <- get(dsname)
```

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 original dataset reference and to 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 generic within the 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. Be careful to manage the naming of datasets appropriately to avoid any loss of processed data.

The Shape of the Data

Once the dataset is loaded we seek a basic understanding of the data—its shape. We are interested in the size of the dataset in terms of the number of observations (rows) and variables (columns). We can simply type the variable name that stores the dataset and will be presented with a summary of the actual contents of the dataset.

```
# Print the dataset.

ds

## # A tibble: 138,307 x 24
##          Date Location MinTemp MaxTemp Rainfall Evaporation
##        <date>    <chr>   <dbl>   <dbl>    <dbl>       <chr>
##  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>
## # ... with 138,297 more rows, and 18 more variables:
## #   Sunshine <chr>, 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>
```

The function base::**dim()** will provide dimension information (observations and variables) as will base::**nrow()** and base::**ncol()**. We use rattle::**comcat()** to format numbers with commas.

```
# Basic size information.

dim(ds)  %>% comcat()

## 138,307 24

nrow(ds) %>% comcat()

## 138,307

ncol(ds) %>% comcat()

## 24
```

Data Ingestion

A useful alternative for our initial insight into the dataset is to use tibble::**glimpse()**.

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

glimpse(ds)

## Observations: 138,307
## Variables: 24
## $ Date          <date> 2008-12-01, 2008-12-02, 2008-12-03,...
## $ Location      <chr> "Albury", "Albury", "Albury", "Albur...
## $ MinTemp       <dbl> 13.4, 7.4, 12.9, 9.2, 17.5, 14.6, 14...
## $ MaxTemp       <dbl> 22.9, 25.1, 25.7, 28.0, 32.3, 29.7, ...
## $ Rainfall      <dbl> 0.6, 0.0, 0.0, 0.0, 1.0, 0.2, 0.0, 0...
## $ Evaporation   <chr> NA, NA, NA, NA, NA, NA, NA, NA, NA, ...
## $ Sunshine      <chr> NA, NA, NA, NA, NA, NA, NA, NA, NA, ...
## $ WindGustDir   <chr> "W", "WNW", "WSW", "NE", "W", "WNW",...
## $ WindGustSpeed <int> 44, 44, 46, 24, 41, 56, 50, 35, 80, ...
## $ WindDir9am    <chr> "W", "NNW", "W", "SE", "ENE", "W", "...
## $ WindDir3pm    <chr> "WNW", "WSW", "WSW", "E", "NW", "W",...
## $ WindSpeed9am  <int> 20, 4, 19, 11, 7, 19, 20, 6, 7, 15, ...
## $ WindSpeed3pm  <int> 24, 22, 26, 9, 20, 24, 24, 17, 28, 1...
## $ Humidity9am   <int> 71, 44, 38, 45, 82, 55, 49, 48, 42, ...
## $ Humidity3pm   <int> 22, 25, 30, 16, 33, 23, 19, 19, 9, 2...
## $ Pressure9am   <dbl> 1007.7, 1010.6, 1007.6, 1017.6, 1010...
## $ Pressure3pm   <dbl> 1007.1, 1007.8, 1008.7, 1012.8, 1006...
## $ Cloud9am      <int> 8, NA, NA, NA, 7, NA, 1, NA, NA, NA,...
## $ Cloud3pm      <int> NA, NA, 2, NA, 8, NA, NA, NA, NA, NA...
## $ Temp9am       <dbl> 16.9, 17.2, 21.0, 18.1, 17.8, 20.6, ...
## $ Temp3pm       <dbl> 21.8, 24.3, 23.2, 26.5, 29.7, 28.9, ...
## $ RainToday     <chr> "No", "No", "No", "No", "No", "No", ...
## $ RISK_MM       <dbl> 0.0, 0.0, 0.0, 1.0, 0.2, 0.0, 0.0, 0...
## $ RainTomorrow  <chr> "No", "No", "No", "No", "No", "No", ...
```

Normalizing Variable Names

Next we review the variables included in the dataset. The function base::**names()** will list the names of the variables (columns).

```
# Identify the variables of the dataset.

names(ds)

##  [1] "Date"          "Location"      "MinTemp"
```

```
##  [4] "MaxTemp"       "Rainfall"      "Evaporation"
##  [7] "Sunshine"      "WindGustDir"   "WindGustSpeed"
## [10] "WindDir9am"    "WindDir3pm"    "WindSpeed9am"
## [13] "WindSpeed3pm"  "Humidity9am"   "Humidity3pm"
## [16] "Pressure9am"   "Pressure3pm"   "Cloud9am"
....
```

The names of the variables within the dataset as supplied to us may not be in any particular standard form and may use different conventions. For example, we might see a mix of upper and lower case letters (WindSpeed9AM) or variable names that are very long (Wind_Speed_Recorded_Today_9am) or use sequential numbers to identify each variable (V004 or V010_wind_speed) or use codes (XVn34_windSpeed) or any number of other conventions. Often we prefer and it is convenient to simplify the variable names to ease our processing and to enforce a standard and consistent naming convention for ourselves. This will help us in interacting the data without regular reference to how the variables are named.

A useful convention 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 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. Yet we often see such variable names arising in practise especially when we import data from databases which are often case insensitive.

We use rattle::**normVarNames()** to make a reasonable attempt of converting variables from a dataset into a preferred standard form. The actual form follows the style we introduce in Chapter 11. The following example shows the original names and how they are transformed into a normalized form. Here we make extensive use of the function base::**names()** to work with the variable names. *

Notice the use of the assignment pipe here as introduced in Chapter 2. Recall that the magrittr:: **%** <> **%** operator will pipe the left-hand data to the function on the right-hand side and then return the result to the left-hand side overwriting the original contents of the memory referred to on the left-hand side. In this case, the left-hand side refers to the variable names of the dataset.

3.2 Data Review

Once we have loaded the dataset and had our initial review of its size and cleaned up the variable names the next step is to understand its structure—that is, understand what the data within the dataset looks like.

```
# Review the variables before normalising their names.

names(ds)

##  [1] "Date"          "Location"      "MinTemp"
##  [4] "MaxTemp"       "Rainfall"      "Evaporation"
##  [7] "Sunshine"      "WindGustDir"   "WindGustSpeed"
## [10] "WindDir9am"    "WindDir3pm"    "WindSpeed9am"
## [13] "WindSpeed3pm"  "Humidity9am"   "Humidity3pm"
## [16] "Pressure9am"   "Pressure3pm"   "Cloud9am"
....

# Normalise the variable names.

names(ds) %<>% normVarNames()

# Confirm the results are as expected.

names(ds)

##  [1] "date"           "location"       "min_temp"
##  [4] "max_temp"       "rainfall"       "evaporation"
##  [7] "sunshine"       "wind_gust_dir"  "wind_gust_speed"
## [10] "wind_dir_9am"   "wind_dir_3pm"   "wind_speed_9am"
## [13] "wind_speed_3pm" "humidity_9am"   "humidity_3pm"
## [16] "pressure_9am"   "pressure_3pm"   "cloud_9am"
....
```

Structure

A basic summary of the structure of the dataset is presented using tibble::**glimpse()** as we saw above.

```
# Review the dataset.

glimpse(ds)

## Observations: 138,307
## Variables: 24
## $ date           <date> 2008-12-01, 2008-12-02, 2008-12-0...
## $ location       <chr> "Albury", "Albury", "Albury", "Alb...
## $ min_temp       <dbl> 13.4, 7.4, 12.9, 9.2, 17.5, 14.6, ...
## $ max_temp       <dbl> 22.9, 25.1, 25.7, 28.0, 32.3, 29.7...
## $ rainfall       <dbl> 0.6, 0.0, 0.0, 0.0, 1.0, 0.2, 0.0,...
## $ evaporation    <chr> NA, NA, NA, NA, NA, NA, NA, NA, NA...
....
```

From this summary we see the variable names, their data types and the first few values of the variable. We can see a variety of data types here, ranging from **Date** (date), through **character** (chr) and **numeric** (dbl).

We confirm the data looks as we would expect and begin to gain some insight into the data itself. We might start asking questions such as whether the date values are a sequence of days as we might expect. The first few locations are listed as Albury and so we might ask what the other values are. We see minimum and maximum temperatures and we note the rainfall and

evaporation. We expect each of these to be numeric though we observe that evaporation is reported as a character variable (which we will come back to later). For the sample above we only see missing values for evaporation. For the numerics we will want to understand the distributions of the values of the variables.

Generally our datasets are very large, with many observations (often in the millions) and many variables (sometimes in the thousands). We can't be expected to browse through all of the observations and variables. Instead we can review the contents of the dataset using utils::**head()** and utils::**tail()** to review the top six (by default) and the bottom six observations.

```
# Review the first few observations.

head(ds)

## # A tibble: 6 x 24
##         date location min_temp max_temp rainfall evaporation
##       <date>    <chr>    <dbl>    <dbl>    <dbl>       <chr>
## 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>
....
```

```
# Review the last few observations.

tail(ds)

## # A tibble: 6 x 24
##         date location min_temp max_temp rainfall evaporation
##       <date>    <chr>    <dbl>    <dbl>    <dbl>       <chr>
## 1 2017-01-25    Uluru     21.3     34.2      0.2        <NA>
## 2 2017-01-26    Uluru     23.6     35.5      0.0        <NA>
## 3 2017-01-27    Uluru     24.6     37.9      4.2        <NA>
## 4 2017-01-28    Uluru     24.7     39.2      0.2        <NA>
....
```

We can also have a look at some random observations from the dataset to provide a little more insight. Here we use dplyr::**sample_n()** to randomly select 6 rows from the dataset.

```
set.seed(2)
```

```
# Review a random sample of observations.

sample_n(ds, size=6)

## # A tibble: 6 x 24
##         date     location min_temp max_temp rainfall
##       <date>        <chr>    <dbl>    <dbl>    <dbl>
## 1 2016-04-22      Penrith     13.8     26.8      0.0
## 2 2015-11-10 MountGambier     12.6     19.1      0.8
## 3 2015-05-15     Dartmoor      9.1     13.6      1.4
## 4 2009-09-08      Penrith      6.4     22.4      0.2
....
```

All the time we are building a picture of the data we are looking at. We note the date appears to be a daily sequence starting from December 2008 and ending in 2017. We also note that evaporation is often but not always missing.

3.3 Data Cleaning

Identifying Factors

On loading the dataset into R we can see that a few variables have been identified as having **character** (string of characters) as the values they store. Such variables are often called *categoric* variables. Within R these are usually represented as a data type called **factor** and handled specially by many of the modelling algorithms. Where the **character** data takes on a limited number of possible values we will convert the variable from **character** into **factor** (categoric) so as to take advantage of some special handling.

A **factor** is a variable that can only take on a specific number of known distinct values which we call the ***levels*** of the **factor**. For datasets that we load into R we will not always have examples of all levels of a factor. Consequently it is not always possible to automatically list all of the levels required for the definition of a factor. Thus we load these variables by default as **character** and then convert them to **factor** as required.

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 base::**unique()** values the variable has in the original dataset.

```
# 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 might normally decide to retain this variable as a character data type, but for illustrative purposes we will convert it to a factor.

```
ds$location %<>% as.factor()
```

We next review a base::**table()** of the distribution of the locations.

```
table(ds$location)

##
##         Adelaide           Albany           Albury
##             3047             2894             2894
##     AliceSprings    BadgerysCreek         Ballarat
##             2894             2863             2894
##          Bendigo         Brisbane           Cairns
....
```

Two related variables that are class character that might be better represented as factors are rain_today and rain_tomorrow. We can review the distribution of their values with the following code.

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

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

##     rain_today rain_tomorrow
## No      104498        104493
## Yes      30279         30283
```

Here we dplyr::**select()** from the dataset those variables that start with the string rain_ and then build a base::**table()** over those variables in the subset of the original dataset selected. We use base::**sapply()** to apply base::**table()** to the selected columns since the function takes a single column from the dataset as its argument. This function counts the frequency of the occurence of each value of a variable within the dataset.

We confirm that No and Yes are the only values these two variables have and so 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 convert each variable from character to factor using base::**lapply()**. Note the use of base::**data.frame()** and dplyr::**tbl_df()** to ensure the data is in the correct form to overwrite the original columns in the dataset.

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

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

## [1] "rain_today"    "rain_tomorrow"

# Confirm these are currently character variables.

ds[vnames] %>% sapply(class)

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

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

ds[vnames] %<>%
  lapply(factor) %>%
  data.frame() %>%
  tbl_df() %T>%
  {sapply(., class) %>% print()}

##    rain_today rain_tomorrow
##      "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(starts_with("rain_")) %>%
  sapply(table)

##     rain_today rain_tomorrow
## No      104498        104493
## Yes      30279         30283
```

The three wind direction variables identified as wind_gust_dir, wind_dir_9am and wind_dir_3pm are also **character** variables. We will want to review their distribution of values and can do so in a similar way. Here we dplyr::**select()** from the dataset those variables containing the string _dir and then build a base:: **table()** over those variables in the selected subset of the original dataset. We again use base::**sapply()** with base::**table()** to count the frequency of the occurence of each level of the factors within the dataset.

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

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

##     wind_gust_dir wind_dir_9am wind_dir_3pm
## E            8628         8622         8024
## ENE          7669         7436         7456
## ESE          6894         7157         8000
## N            8922        11339         8520
## NE           6822         7303         7938
## NNE          6260         7763         6332
## NNW          6301         7680         7516
## NW           7791         8404         8306
## S            8670         8241         9359
## SE           8802         8751        10227
## SSE          8707         8573         8857
## SSW          8130         7199         7572
## SW           8554         7930         8878
## W            9577         8128         9733
## WNW          7987         7112         8563
## WSW          8722         6689         9142
```

From the table we notice 16 compass directions. All compass directions are represented and so we will convert these character variables into factors. Notice that the values of the variables are listed in alphabetic order in the above and a simple conversion to a factor will retain the alphabetic order. We might know however that the compass orders the directions in a well-defined manner (from N, NNE, to NW and NNW). With this knowledge we will force the levels to have the appropriate ordering and also let base::**factor()** know that the levels are ordered with ordered=TRUE.

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

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

# 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"
```

```
# Confirm these are currently character variables.

ds[vnames] %>% sapply(class)

## wind_gust_dir  wind_dir_9am  wind_dir_3pm
##   "character"   "character"   "character"

# Convert these variables from character to factor and confirm.

ds[vnames] %<>%
  lapply(factor, levels=compass, ordered=TRUE) %>%
  data.frame() %>%
  tbl_df() %T>%
  {sapply(., class) %>% print()}

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

Again we 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            8922        11339         8520
## NNE          6260         7763         6332
## NE           6822         7303         7938
## ENE          7669         7436         7456
## E            8628         8622         8024
....
```

There are two other variables that have been identified as character data types: evaporation and sunshine. If we look at the dataset we see they have missing values.

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

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

```
# Review their values.

head(ds[cvars])

## # A tibble: 6 x 2
##   evaporation sunshine
##         <chr>    <chr>
## 1        <NA>     <NA>
## 2        <NA>     <NA>
## 3        <NA>     <NA>
## 4        <NA>     <NA>
## 5        <NA>     <NA>
## 6        <NA>     <NA>

sample_n(ds[cvars], 6)

## # A tibble: 6 x 2
##   evaporation sunshine
##         <chr>    <chr>
## 1        <NA>     <NA>
## 2         5.8       12
## 3           2        9
## 4         1.8      8.4
## 5         2.2        0
## 6         4.8        0
```

The heuristic used to determine the data type when readr::**read_csv()** ingests the data only looks at a subset of all the data before it determines the data types. In this case they were missing for the early observations and so in the absence of further information they were represented as character. We need to convert them to numeric.

```
# Check the current class of the variables.

ds[cvars] %>% sapply(class)

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

# Convert to numeric.
```

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

# Confirm the conversion.

ds[cvars] %>% sapply(class)

## evaporation    sunshine
##   "numeric"   "numeric"
```

We have now dealt with all of the character variables converting them to factors or numerics on a case-by-case basis on our understanding of the data.

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 choose to be selective if there are many factors.

```
# Note which variables are categoric.

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

##      location wind_gust_dir  wind_dir_9am  wind_dir_3pm
##             2             8            10            11
##    rain_today rain_tomorrow
##            22            24

# Normalise the levels of all categoric variables.

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

To confirm we can review the categoric variables.

```
glimpse(ds[catc])
```

```
## Observations: 138,307
## Variables: 6
## $ location      <fctr> albury, albury, albury, albury, alb...
## $ wind_gust_dir <ord> w, wnw, wsw, ne, w, wnw, w, w, nnw, ...
## $ wind_dir_9am  <ord> w, nnw, w, se, ene, w, sw, sse, se, ...
## $ wind_dir_3pm  <ord> wnw, wsw, wsw, e, nw, w, w, w, nw, s...
....
```

Ensure Target is a Factor

Many data mining tasks can be expressed as building classification models. For such models we want to ensure the target is categoric. Often it is 0/1 and hence is loaded as numeric. In such cases we could tell our model algorithm 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.

```
# Note the target variable.

target <- "rain_tomorrow"

# Ensure the target is categoric.

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

# Confirm the distribution.

ds[target] %>% table()

## .
##     no    yes
## 104493  30283
```

It is always a good idea to visualise the distribution of the target (and other) variables using ggplot2. We can pipe the dataset into 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 3.1. With some surprise we note that there are missing values in the dataset. We will deal with the missing values (NA) shortly.

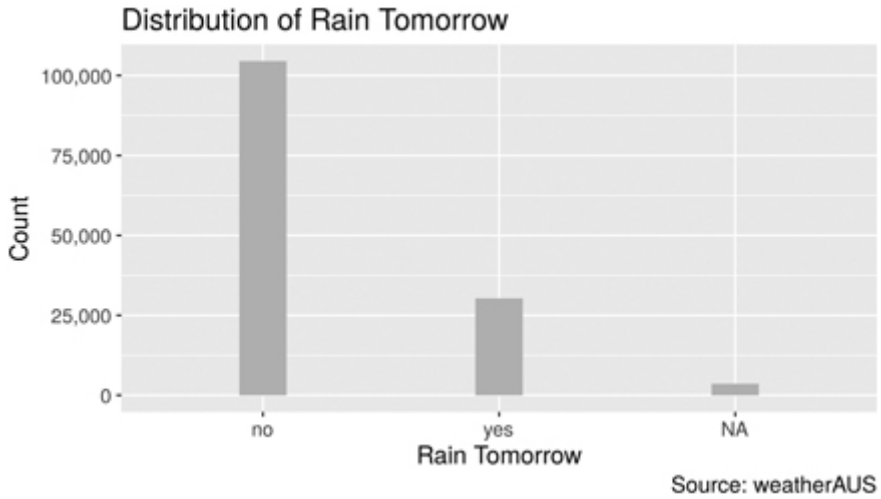

Figure 3.1 *Target variable distribution. Plotting the distribution is useful to gain an insight into the number of observations in each category. As is the case here we often see a skewed distribution.*

```
ds %>%
  ggplot(aes_string(x=target)) +
  geom_bar(width=0.2, fill="grey") +
  theme(text=element_text(size=14)) +
  scale_y_continuous(labels=comma) +
  labs(title   = "Distribution of Rain Tomorrow",
       x       = "Rain Tomorrow",
       y       = "Count",
       caption = "Source: weatherAUS")
```

3.4 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 cleaning, we are in a position to identify the roles played by the variables within the dataset. We record the list of available variables so that we might reference them shortly.

```
# Note the available variables.

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

```
##  [1] "date"           "location"       "min_temp"
##  [4] "max_temp"       "rainfall"       "evaporation"
##  [7] "sunshine"       "wind_gust_dir"  "wind_gust_speed"
## [10] "wind_dir_9am"   "wind_dir_3pm"   "wind_speed_9am"
## [13] "wind_speed_3pm" "humidity_9am"   "humidity_3pm"
## [16] "pressure_9am"   "pressure_3pm"   "cloud_9am"
## [19] "cloud_3pm"      "temp_9am"       "temp_3pm"
## [22] "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 in model building is to identify any patterns among the other observed variables that suggest that it rains the following day.

We also take 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 and the remainder are the variables that will be used to build a model. Another common practise is for the target to be the final column of the dataset.

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

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

```
vars
```

```
##  [1] "rain_tomorrow"   "date"            "location"
##  [4] "min_temp"        "max_temp"        "rainfall"
##  [7] "evaporation"     "sunshine"        "wind_gust_dir"
## [10] "wind_gust_speed" "wind_dir_9am"    "wind_dir_3pm"
## [13] "wind_speed_9am"  "wind_speed_3pm"  "humidity_9am"
## [16] "humidity_3pm"    "pressure_9am"    "pressure_3pm"
....
```

Notice the use of base::**unique()** simply to remove the original occurrence of the target variable.

Another variable that we observe as relating to the outcome rather than to today's observations is risk_mm. From the business context we would learn that this records the amount of rain that fell "tomorrow". We refer to this as a *risk variable*. It is a measure of the impact or risk of the outcome we are predicting (whether it rains 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"
```

Finally 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. We note these two variables as identifiers. Identifiers would not usually be used as independent variables for building predictive analytics models.

```
# Note any identifiers.

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

3.5 Feature Selection

We now move on to identifying variables (features or columns of the dataset) that are irrelevant or inappropriate for modelling.

IDs and Outputs

We start by noting that we should ignore all identifiers and any risk variable (which will be an output variable rather than an input variable). These variables should be ignored in our 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 into a new vector using dplyr::**union()** which performs a set union operation—that is, it joins the two arguments together and removes any repeated variables.

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

id %>%
  union(if (exists("risk")) risk) %T>%
  print() ->
ignore

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

We might also check for variables that have a unique value for every observation. These are often identifiers and if so they are candidates for ignoring.

We begin in the following code block by defining a helper function that given a vector of data it will return the number of unique values found in that vector. This helper function is then

deployed in the following pipeline to identify those vars which have as many unique values as there are rows in the dataset. The pipeline is explained next. This is our first example of defining our own function.

```
# Helper function to count the number of distinct values.

count_unique <- function(x)
{
  x %>% unique() %>% length()
}

# Heuristic for candidate indentifiers to possibly ignore.

ds[vars] %>%
  sapply(count_unique) %>%
  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"
```

We can step through this code line by line to understand its workings. To find the candidate identifiers we retain just the vars from the dataset and pipe this subset of the original dataset ds through to base::**sapply()**. The function base::**sapply()** applies a supplied function to every column of the provided dataset. The function we supply is the helper function we defined. The (x) represents a single column of the dataset at a time. Thus, for each column we identify the base::**unique()** values in that column and then return the base::**length()** of the vector of unique values.

The base::**sapply()** sends the vector of lengths (the number of unique values for each of the columns) on to the next operation through the pipeline. The follow-on operation tests if the calculated number of unique values magrittr::**equals()** the number of rows in the dataset as calculated using base::**nrow()**. The resulting vector of logical values is then piped to base::**which()** to retain those that are TRUE—those that have as many unique values as there are rows in the dataset.

Finally, we extract the base::**names()** of these variables and store them as the variable ids after printing them for information purposes.

We have strung together a series of operations here with each operation piping data on to the next operation. It is worth taking a little time to understand the sequence as a single sentence in the grammar of ***data wrangling*** . As we interact with our data we typically build the sequence adding one extra process at a time in the R Console, confirming the results as we go.

The pipeline has identified no variables as potential identifiers themselves in this dataset; hence, the character(0) result. Below we choose observations from a single location and process that data through the above pipeline to illustrate the selection of date as having a unique value for every observation.

```
# Engineer the data to illustrate identifier selection.

ods <- ds # Take a backup copy of the original dataset.

ds %<>% filter(location=="sydney")

ds[vars] %>%
  sapply(count_unique) %>%
  equals(nrow(ds)) %>%
  which() %>%
  names()

## [1] "date"

ds <- ods # Restore the original dataset.
```

All Missing

We next remove any variable where all of the values are missing. Our pipeline here counts the number of missing values for each variable and then lists the names of those variables that have no values. We introduce another small helper function to count the number of missing values for a vector.

```
# Helper function to count the number of values missing.

count_na <- function(x)
{
  x %>% is.na() %>% sum()
}

# Identify variables with only missing values.

ds[vars] %>%
  sapply(count_na) %>%
  equals(nrow(ds)) %>%
  which() %>%
  names() %T>%
  print() ->
missing

## character(0)

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

ignore %<>% union(missing) %T>% print()

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

Again there are no variables that are completely missing in the **weatherAUS** dataset but in general it is worth checking. Below we engineer a dataset with all missing values for some variables to illustrate the pipeline in action.

```
# Engineer the dataset to illustrate missing columns.

ods <- ds # Take a backup copy of the dataset.

ds %<>% filter(location=="albury")

ds[vars] %>%
  sapply(count_na) %>%
  equals(nrow(ds)) %>%
  which() %>%
  names()

## [1] "evaporation" "sunshine"

ds <- ods # Restore the dataset.
```

Note that it is also adding to our knowledge of this dataset that there are locations for which some variables are not observed or recorded. This may play a role in understanding how to model the data.

Many Missing

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 80% of the values missing:

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

missing.threshold <- 0.8

# Identify variables that are mostly missing.

ds[vars] %>%
  sapply(count_na) %>%
  '>'(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"
```

Here again we identify no variables that have a high proportion of missing observations.

Too Many Levels

Another issue we often come across in our datasets are factors that have very many levels. We might want to ignore such variables (or perhaps group them appropriately). Here we simply identify them and add them to the list of variables to ignore:

```
# Helper function to count the number of levels.

count_levels <- function(x)
{
  ds %>% extract2(x) %>% levels() %>% length()
}

# 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(count_levels) %>%
  '>='(levels.threshold) %>%
  which() %>%
  names() %T>%
  print() ->
too.many

## [1] "location"

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

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

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

The variable location is identified as having too many levels and is thus added to the ignore list though since it is already on that list there is no change to it.

Constants

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

```
# Helper function to test if all values in vector are the same.

all_same <- function(x)
{
  all(x == x[1L])
}

# Identify variables that have a single value.

ds[vars] %>%
  sapply(all_same) %>%
  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"
```

There are no constants found in this dataset.

Correlated Variables

It is often useful to reduce the number of variables we are modelling by identifying and removing highly correlated variables. Such variables will often record the same information but in different ways. Correlated variables can often arise when we combine data from different sources.

First we will identify the numeric variables on which we will calculate correlations. We start by removing the ignored variables from the dataset. We then identify the numeric variables by base::**sapply()**ing the function base::**is.numeric()** to the dataset then find base::**which()** variables are numeric. The variable names are stored into the variable numc.

```
# Note which variables are numeric.

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

##  [1] "min_temp"       "max_temp"       "rainfall"
##  [4] "evaporation"    "sunshine"       "wind_gust_speed"
##  [7] "wind_speed_9am" "wind_speed_3pm" "humidity_9am"
## [10] "humidity_3pm"   "pressure_9am"   "pressure_3pm"
## [13] "cloud_9am"      "cloud_3pm"      "temp_9am"
## [16] "temp_3pm"
```

We can then calculate the correlation between the numeric variables by selecting the numeric columns from the dataset and passing that through to stats::**cor()**. This generates a matrix of pairwise correlations 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. Notice that with diag=TRUE this includes the diagonals of the matrix being set to NA as they will always be perfect correlations (1).

Next we ensure the values are positive using base::**abs()**. We also ensure we have a base::**data.frame()** which we convert to a dplyr::**tbl_df()**. The dataset column names need to be reset appropriately using magrittr::**set_colnames()**. We dplyr::**mutate()** the dataset by adding a new column, then tidyr::**gather()** the dataset. Missing correlations are omitted using stats::**na.omit()**. Finally, the rows are dplyr:: **arrange()**'d with the highest absolute correlations appearing first.

```
# For 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 0.9851472
##  2   pressure_3pm    pressure_9am 0.9620731
##  3       temp_9am        min_temp 0.9069890
##  4       temp_9am        max_temp 0.8936331
##  5       temp_3pm        temp_9am 0.8710598
##  6       max_temp        min_temp 0.7497200
##  7       temp_3pm        min_temp 0.7271346
....
```

This is quite a complex pipeline. It is worth taking time to understand the sequence as a single sentence in the grammar of data wrangling. Importantly it should be noted that we build up such a command sequence interactively, adding one new command in the pipeline sequence at a time until we have our final desired outcome. It is strongly recommended that you replicate building the sequence one step at a time to review and understand the result after each step.

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. We should confirm that the three most highly correlated variables here make intuitive sense.

```
# 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...
## [5] "pressure_3pm" "temp_9am"
```

Removing Variables

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

```
# Check the number of variables currently.

length(vars)

## [1] 24

# Remove the variables to ignore.

vars %<>% setdiff(ignore) %T>% print()

##  [1] "rain_tomorrow"   "min_temp"        "max_temp"
##  [4] "rainfall"        "evaporation"     "sunshine"
##  [7] "wind_gust_dir"   "wind_gust_speed" "wind_dir_9am"
## [10] "wind_dir_3pm"    "wind_speed_9am"  "wind_speed_3pm"
## [13] "humidity_9am"    "humidity_3pm"    "pressure_9am"
## [16] "cloud_9am"       "cloud_3pm"       "rain_today"

# Confirm they are now ignored.

length(vars)

## [1] 18
```

Algorithmic Feature Selection

There are many R packages available to support the preparation of our datasets and over time you will find packages that suit your needs. As you do so they can be added to your version of the template for data wrangling. For example, a useful package is FSelector which provides functions to identify subsets of variables that might be most effective for modelling. We can use this (and other packages) to further assist us in reducing the variables for modelling.

As an example we can use FSelector::**cfs()** to identify a subset of variables to consider for use in modelling by using correlation and entropy.

```
# Construct the formulation of the modelling to undertake.

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

## rain_tomorrow ~ .

# Use correlation search to identify key variables.

cfs(form, ds[vars])

## [1] "sunshine"     "humidity_3pm" "rain_today"
```

Notice the use of the stringi::**%s+%** operator as 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.

A second example lists the variable importance using FSelector::**information.gain()** to advise a useful subset of variables.

```
# Use information gain to identify variable importance.

information.gain(form, ds[vars]) %>%
  rownames_to_column("variable") %>%
  arrange(-attr_importance)

##           variable attr_importance
## 1     humidity_3pm     0.113700324
## 2         sunshine     0.058452293
## 3         rainfall     0.056252382
## 4        cloud_3pm     0.053338637
## 5       rain_today     0.044860122
## 6     humidity_9am     0.039193871
## 7        cloud_9am     0.036895020
## 8     pressure_9am     0.028837842
## 9  wind_gust_speed     0.025299064
## 10        max_temp     0.014812106
## 11    wind_dir_9am     0.008286797
## 12     evaporation     0.006114435
## 13   wind_gust_dir     0.005849817
## 14        min_temp     0.005012095
## 15  wind_speed_3pm     0.004941750
## 16    wind_dir_3pm     0.004717975
## 17  wind_speed_9am     0.003778012
```

The two measures are somewhat consistent in this case. The variables identified by FSelector::**cfs()** are mostly the more important variables identified by FSelector::**information.gain()**. However rain_today is a little further down on the information gain list.

3.6 Missing Data

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 systematic data issue rather than simple randomness. It is important to investigate further why the data is systematically missing. Often this will also lead to a better understanding of the data and how it was collected.

```
# Check the dimensions to start with.

dim(ds)

## [1] 138307     24
```

```
# Identify observations with a missing target.

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

# Check how many are found.

sum(missing_target)

## [1] 3531

# Remove observations with a missing target.

ds %<>% filter(!missing_target)

# Confirm the filter delivered the expected dataset.

dim(ds)

## [1] 134776     24
```

Missing values in the input variables are also an issue for some but not all algorithms. For example, the traditional ensemble model building algorithm randomForest::**randomForest()** omits observations with missing values by default whilst rpart::**rpart()** has a particularly well-developed approach to dealing with missing values.

In the previous section we removed variables with many missing values noting that this is not always appropriate. We may want to instead impute missing values in the data (noting also that it is not always wise to do so as it is effectively inventing data).

Here we illustrate a process for imputing missing values using randomForest::**na.roughfix()**. As the name suggests, this function provides a basic algorithm for imputing missing values. We will demonstrate the process but then restore the original dataset rather than have the imputations 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() %>% comcat()

## 278,293

# Impute missing values.

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

# Confirm that no missing values remain.

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

## 0

# Restore the original dataset.

ds <- ods
```

An alternative might be to remove observations that have missing values. We use stats::**na.omit()** to identify 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 can then remove these observations from the dataset. We start again by keeping a copy of the original dataset to restore later. We also initialise a list of row indicies that we will (omit) from the dataset.

```
# 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] 134776

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

```
## 278,293

# 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() %>% comcat()

## 54,757

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

## [1] 0

# Restore the original dataset.

ds   <- ods
omit <- NULL
```

By restoring the dataset to its original contents to continue with our analysis we are deciding not to omit any observations at this time.

3.7 Feature Creation

Another important task for the data scientist is to create new features from the provided data where appropriate. We will illustrate with two examples. Note that we will often iterate over the feature creation process many times during the life cycle of a project. New derived features will become identified as we gain insights into the data and through our modelling.

Derived Features

From a review of the data we note that each observation has a date associated with it. Unless we are specifically performing time series analysis (and indeed this would be an appropriate analysis to consider for this dataset) some derived features may be useful for our model building rather than using the specific dates as input variables.

Here we add two derived features to our dataset: year and season. The decision to add these was made after we initially began exploring the data and building our initial predictive models. Feedback from our domain expert suggested that the changing pattern over the years is of interest and that predicting rain will often involve seasonal adjustments.

The dataset is dplyr::**mutate()**'d to add these two features. The year is a simple extraction from the date using base::**format()**. To compute the season we extract the month base::**as.integer()** which is then an index to base::**switch()** to a specific season depending on the month.

```
ds %<>%
  mutate(year   = factor(format(date, "%Y")),
         season = format(ds$date, "%m") %>%
                  as.integer() %>%
                  sapply(function(x)
                    switch(x,
                           "summer", "summer", "autumn",
                           "autumn","autumn", "winter",
                           "winter", "winter", "spring",
                           "spring", "spring", "summer")) %>%
                  as.factor()) %T>%
  {select(., date, year, season) %>% sample_n(10) %>% print()}

## # A tibble: 10 x 3
##          date   year season
##        <date> <fctr> <fctr>
##  1 2014-11-13   2014 spring
##  2 2014-10-15   2014 spring
##  3 2010-12-18   2010 summer
##  4 2010-02-06   2010 summer
##  5 2016-01-07   2016 summer
##  6 2015-01-01   2015 summer
##  7 2009-11-22   2009 spring
##  8 2013-09-28   2013 spring
##  9 2009-06-09   2009 winter
## 10 2012-05-27   2012 autumn
```

The final line of code prints a random sample of the original and new features. It is critical to confirm the results are as expected—a sanity check.

The introduced variables will have different roles which are recorded appropriately.

```
vars %<>% c("season")
id   %<>% c("year")
```

Model-Generated Features

In addition to developing features derived from other features we will sometimes find it useful to include model-generated features. A common one is to cluster observations or some aggregation of observations into similar groups. We then uniquely identify each group (for example, by numbering each group) and add this grouping as another column or variable within our dataset.

A cluster analysis (also called segmentation) provides a simple mechanism to identify groups and allows us to then visualise within those groups, for example, rather than trying to visualise the whole dataset at one time. We will illustrate the process of a cluster analysis over the numeric variables within our dataset. The aim of the cluster analysis is to group the locations so that within a group the locations will have similar values for the numeric variables and between the groups the numeric variables will be more dissimilar. The traditional clustering algorithm is stats::**kmeans()**.

In the following code block we specify the number of clusters that we wish to create, storing that as the variable NCLUST. We then select the numc (numeric) variables from the dataset ds, dplyr::**group_by()** the location and then dplyr::**summarise_all()** variables. As an interim step we store for later use the names of the locations. We then continue by removing location from the dataset leaving just the numeric variables. The variables are then rescaled so that all values across the different variables are in the same range. This rescaling is required for cluster analysis so as not to introduce bias due to very differently scaled variables. The stats::**kmeans()** cluster analysis is then performed. We base::**print()** the results and magrittr::**extract2()** just the cluster number for each location.

```
# Reset the random number generator seed for repeatablity.

set.seed(7465)

# Cluster the numeric data per location.

NCLUST <- 5

ds[c("location", numc)] %>%
  group_by(location) %>%
  summarise_all(funs(mean(., na.rm=TRUE))) %T>%
  {locations <<- .$location} %>% # Store locations for later.
  select(-location) %>%
  sapply(function(x) ifelse(is.nan(x), 0, x)) %>%
  as.data.frame() %>%
  sapply(scale) %>%
  kmeans(NCLUST)  %T>%
  print() %>%
  extract2("cluster")->
cluster

## K-means clustering with 5 clusters of sizes 4, 22, 10, 8, 5
##
## Cluster means:
##     min_temp   max_temp     rainfall evaporation    sunshine
## 1 -0.6271436 -0.5345409  0.061972675  -1.2699891 -1.21861982
## 2 -0.3411683 -0.5272989 -0.007762188   0.1137179  0.09919753
....

head(cluster)

## [1] 3 2 2 3 4 2
```

The cluster numbers are now associated with each location as stored within the vector using their base::**names()**. We can then dplyr::**mutate()** the dataset by adding the new cluster number indexed by the location for each observation.

```
# Index the cluster vector by the appropriate locations.

names(cluster) <- locations

# Add the cluster to the dataset.

ds %<>% mutate(cluster="area" %>%
                paste0(cluster[ds$location]) %>%
                as.factor)

# Check clusters.

ds %>% select(location, cluster) %>% sample_n(10)

## # A tibble: 10 x 2
##         location cluster
##           <fctr>  <fctr>
##  1      richmond   area2
##  2         cobar   area3
##  3 mount_gambier   area2
##  4          nhil   area4
##  5        sydney   area2
##  6    launceston   area2
##  7  mount_ginini   area1
##  8    wollongong   area2
##  9    gold_coast   area4
## 10        hobart   area2
```

The introduced variable's role in modelling is recorded appropriately.

```
vars %<>% c("cluster")
```

A quick sanity check will indicate that basically the clustering looks okay, grouping locations that the domain expert agrees are similar in terms of weather patterns.

```
# Check that the clustering looks okay.

cluster[levels(ds$location)] %>% sort()

##      mount_ginini         newcastle           penrith
##                 1                 1                 1
##       salmon_gums            albany            albury
##                 1                 2                 2
##          ballarat           bendigo          canberra
##                 2                 2                 2
....
```

3.8 Preparing the Metadata

Metadata is data about the data. We now record data about our dataset that we use later in further processing and analysis.

Variable Types

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 also identify variables that are the model inputs (also called the independent variables). We record then both as a vector of characters (the variable names) and a vector of integers (the variable indicies).

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

##  [1] "min_temp"       "max_temp"       "rainfall"
##  [4] "evaporation"    "sunshine"       "wind_gust_dir"
##  [7] "wind_gust_speed" "wind_dir_9am"   "wind_dir_3pm"
## [10] "wind_speed_9am" "wind_speed_3pm" "humidity_9am"
## [13] "humidity_3pm"   "pressure_9am"   "cloud_9am"
```

```
## [16] "cloud_3pm"      "rain_today"     "season"
## [19] "cluster"
```

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.

```
inputs %>%
  sapply(function(x) which(x == names(ds)), USE.NAMES=FALSE)%T>%
  print() ->
inputi

##  [1]  3  4  5  6  7  8  9 10 11 12 13 14 15 16 18 19 22 26 27
```

For convenience we also record the number of observations:

```
ds %>%
  nrow() %T>%
  comcat() ->
nobs

## 134,776
```

Next we report on the dimensions of various data subsets primarily to confirm that the dataset appears as we expect:

```
# Confirm various subset sizes.

dim(ds)         %>% comcat()

## 134,776 27

dim(ds[vars])   %>% comcat()

## 134,776 20

dim(ds[inputs]) %>% comcat()

## 134,776 19

dim(ds[inputi]) %>% comcat()

## 134,776 19
```

Sometimes we need to identify the numeric and categoric variables for separate handling. Many cluster analysis algorithms, for example, only deal with numeric variables. Here we identify them both by name (a character string) and by index. Note that 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.

```
# 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"
##  [4] "evaporation"    "sunshine"       "wind_gust_speed"
##  [7] "wind_speed_9am" "wind_speed_3pm" "humidity_9am"
## [10] "humidity_3pm"   "pressure_9am"   "cloud_9am"
## [13] "cloud_3pm"

# Identify the categoric variables by index.

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

## [1]  8 10 11 22 26 27

# Identify the categoric variables by name.

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

## [1] "wind_gust_dir" "wind_dir_9am"  "wind_dir_3pm"
## [4] "rain_today"    "season"        "cluster"
```

3.9 Preparing for Model Building

Our end goal is to build a model based on the data we have prepared. A ***model*** will capture knowledge about the world that the data represents. The final two tasks in preparing our data for modelling are to specify the form of the model to be built and to identify the actual observations from which the model is to be built. For the latter task we will partition the dataset into three

subsets. Whilst this is primarily useful for model building, it may sometimes be useful to explore a random subset of the whole dataset so that our interactions are more interactive, particularly when dealing with large datasets.

Formula to Describe the Model

A ***formula*** is used to identify what it is that we will model from the supplied data. Typically we identify a ***target variable*** which we will model based on other ***input variables*** . We can construct a stats:: **formula()** automatically from a dataset if the first column of the dataset is the target variable and the remaining columns are the input variables. A simple selection of the columns in this order will generate the initial formula automatically. Earlier we engineered the variable vars to be in the required order.

```
ds[vars] %>%
  formula() %>%
  print() ->
form

## rain_tomorrow ~ min_temp + max_temp + rainfall + evaporati...
##     sunshine + wind_gust_dir + wind_gust_speed + wind_dir_...
##     wind_dir_3pm + wind_speed_9am + wind_speed_3pm + humid...
##     humidity_3pm + pressure_9am + cloud_9am + cloud_3pm + ...
##     season + cluster
## <environment: 0x5640bab65ef8>
```

The notation used to express the formula begins with the name of a target (rain_tomorrow) followed by a tilde (~) followed by the variables that will be used to model the target, each separated by a plus (+). The formula here indicates that we aim to build a model that captures the knowledge required to predict the outcome rain_tomorrow from the provided input variables (details of today's weather). This kind of model is called a classification model and can be compared to regression models, for example, which predict numeric outcomes. Specifically, with just two values to be predicted, we call this binary classification. It is generally the simplest kind of modelling task but also a very common task.

Training, Validation and Testing Datasets

Models are built using a machine learning algorithm which learns from the dataset of historic observations. A common methodology for building models is to partition the available data into a ***training dataset*** and a ***testing dataset*** . We will build (or ***train***) a model using the training dataset and then test how good the model is by applying it to the testing dataset. Typically we also introduce a third dataset called the ***validation dataset*** . This is used during the building of the model to assist in tuning the algorithm through trialling different parameters to the machine learning algorithm. In this way we search for the best model using the validation dataset and then obtain a measure of the performance of the final model using the testing dataset.

The original dataset is partitioned randomly into the three subsets. To ensure we can repeatably reproduce our results we will first initiate a random number sequence with a randomly selected seed. In this way we can replicate the examples presented in this book by ensuring the same random subset is selected each time. We will initialise the random number generator with a specific seed using base::**set.seed()**. For no particular reason we choose 42.

```
# Initialise random numbers for repeatable results.

seed <- 42
set.seed(seed)
```

We are now ready to partition the dataset into the two or three subsets. The first is typically a 70% random sample for building the model (the training dataset). The second and third consist of the remainder, used to tune and then estimate the expected performance of the model (the validation and testing datasets).

Rather than actually creating three subsets of the dataset, we simply record the index of the observations that belong to each of the three subsets.

```
# Partition the full dataset into three.

nobs %>%
  sample(0.70*nobs) %T>%
  {length(.) %>% comcat()} %T>%
  {sort(.) %>% head(30) %>% print()} ->
train

## 94,343
##  [1]  1  4  7  9 10 11 12 13 14 15 16 17 18 19 21 22 23 24 26
## [20] 28 31 32 33 35 37 40 41 42 43 45

nobs %>%
  seq_len() %>%
  setdiff(train) %>%
  sample(0.15*nobs) %T>%
  {length(.) %>% comcat()} %T>%
  {sort(.) %>% head(15) %>% print()} ->
validate

## 20,216
##  [1]  2  3  6 20 25 27 29 34 39 44 49 65 66 71 81
```

```
nobs %>%
  seq_len() %>%
  setdiff(union(train, validate)) %T>%
  {length(.) %>% comcat()} %T>%
  {head(.) %>% print(15)} ->
test

## 20,217
## [1]  5  8 30 36 38 48
```

We take a moment here to record the actual target values across the three datasets as these will be used in the evaluations performed in later chapters. A secondary target variable is noted, referred to as the risk variable. This is a measure of the impact or size of the outcome and is common in insurance, financial risk and fraud analysis. For our sample dataset, the risk variable is risk_mm which reports the amount of rain recorded tomorrow.

Notice the correspondence between risk values (the amount of rain) and the target, where 0.0 mm of rain corresponds to the target value no. In particular though also note that small amounts of rain (e.g., 0.2 mm and 0.4 mm) are treated as no rain.

```
# Cache the various actual values for target and risk.

tr_target <- ds[train,][[target]] %T>%
             {head(., 20) %>% print()}

##  [1] no  no  no  no  no  no  no  no  no  no  no  no  no  no
## [15] no  no  no  no  yes no
....

tr_risk   <- ds[train,][[risk]] %T>%
             {head(., 20) %>% print()}

##  [1] 0.0 0.0 0.0 0.0 0.4 0.0 0.2 0.0 0.0 0.0 0.0 0.2 0.2 1.0
## [15] 0.2 0.0 0.0 0.0 1.2 0.0
....

va_target <- ds[validate,][[target]] %T>%
             {head(., 20) %>% print()}
```

```
##  [1] yes no  no  no  no  no  no  no  no  no  yes no  no  no
## [15] no  yes no  no  no  yes
....

va_risk   <- ds[validate,][[risk]] %T>%
             {head(., 20) %>% print()}

##  [1] 9.8 0.0 0.0 0.0 0.0 0.0 0.0 0.6 0.8 0.0 1.6 0.0 0.2 0.0
## [15] 0.0 3.0 0.4 0.0 0.0 5.2
....

te_target <- ds[test,][[target]] %T>%
             {head(., 20) %>% print()}

##  [1] no  no  no  no  no  no  yes no  no  no  yes yes no  no
## [15] no  no  no  no  no  yes
....

te_risk   <- ds[test,][[risk]] %T>%
             {head(., 20) %>% print()}

##  [1] 0.2 0.0 0.0 0.0 0.0 0.0 3.0 0.0 0.0 0.0 1.2 5.8 0.0 0.0
## [15] 0.0 0.0 0.0 0.0 0.0 6.2
....
```

3.10 Save the Dataset

Having transformed our dataset in a variety of ways, cleaned it, wrangled it, and added new variables, we will now save the data and its metadata into a binary RData file. Saving it as a binary compressed file saves both storage space and time on reloading the dataset. Loading a binary dataset is generally faster than loading a CSV file.

```
# Save data into a appropriate folder.

fpath <- "data"

# Timestamp for the dataset.
```

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

## [1] "_20170615"

# Use a fixed timestamp to name our file for convenience here.

dsdate <- "_20170702"

# Filename for the saved dataset.

dsfile <- dsname %s+% dsdate %s+% ".RData"

# Full path to the dataset.

fpath %>%
  file.path(dsfile) %T>%
  print() ->
dsrdata

## [1] "data/weatherAUS_20170702.RData"

# Save relevant R objects to the binary RData file.

save(ds, dsname, dspath, dsdate, nobs,
     vars, target, risk, id, ignore, omit,
     inputi, inputs, numi, numc, cati, catc,
     form, seed, train, validate, test,
     tr_target, tr_risk, va_target, va_risk, te_target, te_risk,
     file=dsrdata)

# Check the resulting file size in bytes.

file.size(dsrdata) %>% comma()

## [1] "5,568,471"
```

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"
##  [5] "nobs"      "vars"      "target"    "risk"
##  [9] "id"        "ignore"    "omit"      "inputi"
## [13] "inputs"    "numi"      "numc"      "cati"
## [17] "catc"      "form"      "seed"      "train"
## [21] "validate"  "test"      "tr_target" "tr_risk"
## [25] "va_target" "va_risk"   "te_target" "te_risk"
```

A call to base:: **load()** returns its result invisibly since we are primarily interested in its side-effect. The side-effect is to read the R binary data from storage and to make it available within our current R session.

3.11 A Template for Data Preparation

Throughout this chapter we have worked towards constructing a standard template for ourselves for a data preparation report. Indeed, whenever we begin a new project the template will provide a useful starting point. The introduction of generic variables facilitates this approach to quickly begin any new analysis.

A template based on this chapter for data preparation is available from https://essentials.togaware.com. There we can find a template that will continue to be refined over time and will incorporate improvements and advanced techniques that go beyond what has been presented here. An automatically derived version including just the R code is available there together with the LaTeX compiled PDF version.

Notice that we would not necessarily perform all of the steps we have presented in this chapter. Normalizing the variable names or imputing imputing missing values, omitting observations with missing values, and so on may well depend on the context of the analysis. We will 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.

3.12 Exercises

Exercise 3.1 Exploring the Weather

We have worked with the Australian **weatherAUS** dataset throughout this chapter. For this exercise we will explore the dataset further.

1. Create a data preparation script beginning with the template available from https://essentials.togaware.com and replicate the data processing performed in this chapter.

2. Investigate the dplyr::**group_by()** and dplyr::**summarise()** functions, combined through a pipeline using magrittr::**%** > **%** to identify regions with considerable variance in their weather observations. The use of stats::**var()** might be a good starting point.

Exercise 3.2 Understanding Ferries

A dataset of ferry crossings on Sydney Harbour is available as https://essentials.togaware.com/ferry.csv. * We will use this dataset to exercise our data template.

1. Create a data preparation script beginning with the template available from https://essentials.togaware.com/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 as appropriate and convert other character variables into factors where it is sensible to do so.
6. Work through the template to prepare and then explore the dataset to identify any issues and to develop initial observations.

Create a report on your investigations and share a narrative to communicate your discoveries from the dataset.

Footnote

* The template will consist 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.
* 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.
* When the name of a variable within a dataset is changed a new copy of that value of that variable is created so that now ds and weatherAUS will now be referring to different datasets in memory.
* The original source of the Ferry dataset is https://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.