Data Handling and Data Mining

Fixing The Sparrow Data Set

-

BASIC STATISTICS FOR BIOLOGISTS

Erik Kusch
PhD Student
Aarhus University
Department of Bioscience
Section for Ecoinformatics & Biodiversity
Center for Biodiversity and Dynamics in a Changing World (BIOCHANGE)
Ny Munkegade 116, Building 1540
8000 Aarhus
Denmark
email: erik@i-solution.de

Summary:

Welcome to our first "real" practical experience in `R`. The following notes present you with an example of how data handling (also known as data cleaning) can be done. Obviously, the possibility for flaws to occur in any given data set are seemingly endless and so the following, tedious procedure should be thought of as less of an recipe of how to fix common flaws in biological data sets but make you aware of how important proper data collection and data entry is.

Contents

1. Preparing Our Procedure

The following three sections are what I consider to be *essential* parts of the preamble to any `R`-based analysis. I highly recommend clearly indicating these bits in your code.

More often than not, you will use variations of these code chunks whether you are working on data handling, data exploration or full-fledged statistical analyses.

1.1 Necessary Steps For Reproducibility

Reproducibility is the be-all and end-all of any statistical analysis, particularly in light of the peer-review process in life sciences.

```
rm(list = ls())  # clearing environment
Dir.Base <- getwd()  # soft-coding our working directory
Dir.Data <- paste(Dir.Base, "Data", sep = "/")  # soft-coding our data directory
```

Once you get into highly complex statistical analyses, you may wish to break up chunks of your analysis into separate documents. To ensure that remnants of an earlier analysis or analysis chunk do not influence the results of your current analysis, you may wish to *empty* `R`'s cache (*Environment*) before attempting a new analysis. This is achieved via the command `rm(list=ls())`.

Next, you *need* to remember the importance of *soft-coding* for the sake of reproducibility. One of the worst offences to the peer-review process in `R`-based statistics is the erroneous hard-coding of the working directory. The `getwd()` function shown above solves this exact problem. However, for this workaround to function properly you need to open the code document of interest by double-clicking it within its containing folder.

When using the `xlsx` package or any *Excel*-reliant process via `R`, your code will automatically run a Java process in the background. By default the Java engine is limited as far as RAM allocation goes and tends to fail when faced with enormous data sets. The workaround `options(java.parameters = "-Xmx8g")` gets rid of this issue by allocation 8 GBs of RAM to Java.

1.2 Packages

Packages are `R`'s way of giving you access to a seemingly infinite repository of functions.

```
# function to load packages and install them if they haven't been installed yet
install.load.package <- function(x) {
    if (!require(x, character.only = TRUE))
        install.packages(x)
    require(x, character.only = TRUE)
}
package_vec <- c("dplyr"  # we need this package to fix the most common data errors
)
sapply(package_vec, install.load.package)
```

```
## dplyr
##  TRUE
```

Using the above function is way more sophisticated than the usual `install.packages()` + `library()` approach since it automatically detects which packages require installing and only install these thus not overwriting already installed packages.

1.3 Loading The Data

Loading data is crucial to any analysis in `R`. Period.

`R` offers a plethora of approaches to data loading and you will usually be taught the `read.table()` command in basic biostatistics courses. However, I have found to prefer the functionality provided by the `xlsx` package since most data recording is taking place in Excel. As this package is dependant on the installation of Java and `RJava`, we will settle on the base `R` function `read.csv()`.

```
Data_df_base <- read.csv(file = paste(Dir.Data, "/SparrowData.csv", sep = ""), header = TRUE)
Data_df <- Data_df_base  # duplicate and save initial data on a new object
```

Another trick to have up your sleeve (if your RAM enables you to act on it) is to duplicate your initial data onto a new object once loaded into `R`. This will enable you to easily remedy mistakes in data treatment without having to reload your initial data set from the data file.

2. Inspecting The Data

Once the data is loaded into `R`, you *need to inspect* it to make sure it is ready for use.

2.1 Assessing A Data Frame in `R`

Most, if not all, data you will ever load into `R` will be stored as a `data.frame` within `R`. Some of the most important functions for inspecting data frames ("df" in the following) in base `R` are the following four:

- `dim(df)` returns the dimensions (Rows × Columns)of the data frame
- `head(df)` returns the first 6 rows of the data frame by default (here changed to 4)
- `tail(df)` returns the last 6 rows of the data frame by default (here changed to 4)
- `View(df)` opens nearly any `R` object in a separate tab for further inspection. Since we are dealing with an enormous data set here, I will exclude this function for now to save you from printing unnecessary pages.

```
dim(Data_df)
```

```
## [1] 1068   21
```

```
head(Data_df, n = 4)
```

```
##   NA.    Site Index Latitude Longitude     Climate Population.Status Weight Height
## 1   1 Siberia    SI       60       100 Continental            Native  34,05     13
## 2   2 Siberia    SI       60       100 Continental            Native  34,86     14
## 3   3 Siberia    SI       60       100 Continental            Native  32,34     13
## 4   4 Siberia    SI       60       100 Continental            Native  34,78     15
##   Wing.Chord Colour    Sex Nesting.Site Nesting.Height Number.of.Eggs Egg.Weight Flock
## 1        6.7  Brown   Male           NA             NA             NA         NA     B
## 2        6.8   Grey   Male           NA             NA             NA         NA     B
## 3        6.6  Black Female        Shrub           35.6              1       3.21     C
## 4        7.0  Brown Female        Shrub          47.75              0         NA     E
##   Home.Range Flock.Size Predator.Presence Predator.Type
## 1      Large         16               Yes         Avian
## 2      Large         16               Yes         Avian
## 3      Large         14               Yes         Avian
## 4      Large         10               Yes         Avian
```

```
tail(Data_df, n = 4)
```

```
##        NA.           Site Index Latitude Longitude Climate Population.Status Weight Height
## 1065 1065 Falkland Isles    FI      -52       -59 Coastal        Introduced  34.25     15
## 1066 1066 Falkland Isles    FI      -52       -59 Coastal        Introduced  31.76     13
## 1067 1067 Falkland Isles    FI      -52       -59 Coastal        Introduced  31.48     12
## 1068 1068 Falkland Isles    FI      -52       -59 Coastal        Introduced  31.94     13
##      Wing.Chord Colour  Sex Nesting.Site Nesting.Height Number.of.Eggs Egg.Weight Flock
## 1065        7.0   Grey Male         <NA>           <NA>           <NA>       <NA>     A
## 1066        6.7   Grey Male         <NA>           <NA>           <NA>       <NA>     A
## 1067        6.6  Black Male         <NA>           <NA>           <NA>       <NA>     C
## 1068        6.7   Grey Male         <NA>           <NA>           <NA>       <NA>     A
##      Home.Range Flock.Size Predator.Presence Predator.Type
## 1065      Large         19               Yes         Avian
## 1066      Large         19               Yes         Avian
## 1067      Large         18               Yes         Avian
## 1068      Large         19               Yes         Avian
```

When having an initial look at the results of `head(Data_df)` and `tail(Data_df)` we can spot two important things:

- NAs in head and tail of our data set are stored differently. This is a common problem with biological data sets and we will deal with this issue extensively in the next few sections of this document.
- Due to our data loading procedure we ended up with a redundant first column that is simply showing the respective row numbers. However, this is unnecessary in R and so we can delete this column as seen below.

```
Data_df <- Data_df[, -1]  # eliminating the erroneous first column as it is redundant
dim(Data_df)  # checking if the elimination went right
```

```
## [1] 1068   20
```

2.2 The Summary() Function

As already stated in our seminar series, the summary() function is *invaluable* to data exploration and data inspection. However, it is only partially applicable as it will not work flawlessly on every class of data. Examples of this are shown below.

The weight data contained within our data frame should be numeric and thus pose no issue to the summary() function. However, as shown in the next section, it is currently of type factor which leads the summary() function to work improperly.

```
summary(Data_df$Weight)
```

```
##   31.01   29.11   29.45   31.04   31.66   32.33   21.75    23.3   23.75   29.36   29.51 
##       6       5       5       5       5       5       4       4       4       4       4 
##   29.53   29.86    29.9   29.93   30.04   30.22   30.44   30.63    30.7   31.03   31.19 
##       4       4       4       4       4       4       4       4       4       4       4 
##   31.28   31.37   31.42   31.48   31.54   31.72    32.2   32.27   32.34   32.37   32.68 
##       4       4       4       4       4       4       4       4       4       4       4 
##   33.09   21.69   22.38   22.45   22.55   22.73    22.8   23.23    28.8   28.86   28.98 
##       4       3       3       3       3       3       3       3       3       3       3 
##   29.16    29.3   29.33    29.5   29.54   29.57   29.58   29.69   29.82   29.84   29.89 
##       3       3       3       3       3       3       3       3       3       3       3 
##   29.95   30.01   30.05   30.12    30.3   30.38   30.53   30.57   30.59   30.66   30.67 
##       3       3       3       3       3       3       3       3       3       3       3 
##   30.68   30.69   30.71    30.8   30.83   30.95   31.05   31.18   31.22    31.3   31.38 
##       3       3       3       3       3       3       3       3       3       3       3 
##   31.53   31.55   31.63   31.71   31.77   31.93   31.99   32.05   32.11   32.29   32.32 
##       3       3       3       3       3       3       3       3       3       3       3 
##   32.63   32.66   32.72    33.1   33.44   20.41      21   21.12   21.19   21.31   21.68 
##       3       3       3       3       3       2       2       2       2       2       2 
## (Other) 
##     736 
```

The height data within our data set, on the other hand, is stored correctly as class numeric. Thus the summary() function performs flawlessly.

```
summary(Data_df$Height)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max. 
##       1      14      15      15      16     135 
```

Making data inspection more easy, one may which to automate the use of the summary() function. However, this only makes sense, when every data column is presenting data in the correct class type. Therefore, we will first fix the column classes and then use the summary() command.

3. Data Cleaning Workflow

3.1 Identifying Problems

Indentifying most problems in any data set you may ever encounter comes down to mostly two manifestations of inadequate data entry or handling:

1. Types/Classes
Before even opening a data set, we should know what kind of data classes we expect for every variable (for example, height records as a `factor` don't make much sense). Problems with data/variable classes can have lasting influence on your analyses and so we need to test the class for each variable (column) individually. Before we alter any column classes, we will first need to identify columns whose classes need fixing. Doing so is as easy applying the `class()` function to the data contained within every column of our data frame separately.
`R` offers multiple functions for this but I find the `lapply()` function to perform flawlessly as shown below. Since `lapply()` returns a `list` of class identifiers and these don't translate well to paper, I have opted to transform the list into a named character vector using the `unlist()` command. One could also use the `str()` function.

```
unlist(lapply(Data_df, class))
```

```
##              Site             Index          Latitude         Longitude           Climate
##          "factor"          "factor"         "numeric"         "numeric"          "factor"
## Population.Status            Weight            Height        Wing.Chord            Colour
##          "factor"          "factor"         "numeric"         "numeric"          "factor"
##               Sex      Nesting.Site    Nesting.Height    Number.of.Eggs        Egg.Weight
##          "factor"          "factor"          "factor"          "factor"          "factor"
##             Flock        Home.Range        Flock.Size Predator.Presence     Predator.Type
##          "factor"          "factor"         "numeric"          "factor"          "factor"
```

For further inspection, one may want to combine the information obtained by using the `class()` function with either the `summary()` function (for all non-numeric records) or the `hist` function (particularly useful for numeric records).

2. Contents/Values
Typos and the like will always lead to some data that simply doesn't make sense given the context of your project. Sometimes, errors like these are salvageable but doing so can be a very difficult process. Before we alter any column contents, we will first need to identify columns whose contents need fixing, however. Doing so is as easy applying an automated version of `summary()` to the data contained within every column of our data frame separately after having fixed possibly erroneous data classes.

3.2 Fixing The Problems

Fixing the problems in our data sets always comes down to altering data classes, altering faulty values or removing them entirely.
To make sure we fix all problems, we may often wish to enlist the `summary()` function as well as the `hist()` function for data inspection and visualisation.

Before we alter any column contents, we will first need to identify columns whose contents need fixing.

4. Our Data

4.1 Site

Variable Class Expectation: `factor` (only 11 possible values)

4.1.1 Identifying Problems

Let's asses our Site records for our *Passer domesticus* individuals and check whether they behave as expected:

```
class(Data_df$Site)
```

```
## [1] "factor"
```

```
summary(Data_df$Site)
```

```
##      Australia         Belize Falkland Isles  French Guiana      Louisiana       Manitoba
##             88            105             69            250             81             68
##        Nunavut        Reunion        Siberia  South America United Kingdom
##             64             95             66            114             68
```

Indeed, they do behave just like we'd expect them to.

4.1.2 Fixing Problems

We don't need to fix anything here.

4.2 Index

Variable Class Expectation: `factor` (only 11 possible values)

4.2.1 Identifying Problems

Let's asses our Index records for our *Passer domesticus* individuals and check whether they behave as expected:

```
class(Data_df$Index)
```

```
## [1] "factor"
```

```
summary(Data_df$Index)
```

```
##  AU  BE  FG  FI  LO  MA  NU  RE  SA  SI  UK
##  88 105 250  69  81  68  64  95 114  66  68
```

Indeed, they do behave just like we'd expect them to. Pay attention that thes shortened index numbers lign up with the numbers of site records!

4.2.2 Fixing Problems

We don't need to fix anything here.

4.3 Latitude

Variable Class Expectation: `numeric` (Latitude is inherently continuous)

4.3.1 Identifying Problems

Let's asses our Latitude records for our *Passer domesticus* individuals and check whether they behave as expected:

```
class(Data_df$Latitude)
```

```
## [1] "numeric"
```

```
table(Data_df$Latitude)  # use this instead of summary due to station-dependency here
```

```
##
## -51.75    -25  -21.1      4   14.6  17.25     31     54     55     60     70
##     69     88     95    250    114    105     81     68     68     66     64
```

Indeed, they do behave just like we'd expect them to.

4.3.2 Fixing Problems

We don't need to fix anything here.

4.4 Longitude

Variable Class Expectation: `numeric` (Longitude is inherently continuous)

4.4.1 Identifying Problems

Let's asses our Longitude records for our *Passer domesticus* individuals and check whether they behave as expected:

```
class(Data_df$Longitude)
```

```
## [1] "numeric"
```

```
table(Data_df$Longitude)  # use this instead of summary due to station-dependency here
```

```
##
##    -97    -92    -90 -88.75 -59.17  -57.7    -53     -2   55.6    100    135
##     68     81     64    105     69    114    250     68     95     66     88
```

Indeed, they do behave just like we'd expect them to.

4.4.2 Fixing Problems

We don't need to fix anything here.

4.5 Climate

Variable Class Expectation: `factor` (three levels: coastal, semi-coastal, continental)

4.5.1 Identifying Problems

Let's asses our Climate records for our *Passer domesticus* individuals and check whether they behave as expected:

```
class(Data_df$Climate)
```

```
## [1] "factor"
```

```
summary(Data_df$Climate)
```

```
##      Coastal  Continental Semi-Coastal
##          846          154           68
```

Indeed, they do behave just like we'd expect them to.

4.5.2 Fixing Problems

We don't need to fix anything here.

4.6 Population Status

Variable Class Expectation: `factor` (two levels: native, introduced)

4.6.1 Identifying Problems

Let's asses our Population Status records for our *Passer domesticus* individuals and check whether they behave as expected:

```
class(Data_df$Population.Status)
```

```
## [1] "factor"
```

```
summary(Data_df$Population.Status)
```

```
## Introduced     Native
##        934        134
```

Indeed, they do behave just like we'd expect them to.

4.6.2 Fixing Problems

We don't need to fix anything here.

4.7 Weight

Variable Class Expectation: `numeric` (weight is a continuous metric)

4.7.1 Identifying Problems

Let's asses our Weight records for our *Passer domesticus* individuals and check whether they behave as expected:

```
class(Data_df$Weight)
```

```
## [1] "factor"
```

```
summary(Data_df$Weight)
```

```
##   31.01   29.11   29.45   31.04   31.66   32.33   21.75    23.3   23.75   29.36   29.51 
##       6       5       5       5       5       5       4       4       4       4       4 
##   29.53   29.86    29.9   29.93   30.04   30.22   30.44   30.63    30.7   31.03   31.19 
##       4       4       4       4       4       4       4       4       4       4       4 
##   31.28   31.37   31.42   31.48   31.54   31.72    32.2   32.27   32.34   32.37   32.68 
##       4       4       4       4       4       4       4       4       4       4       4 
##   33.09   21.69   22.38   22.45   22.55   22.73    22.8   23.23    28.8   28.86   28.98 
##       4       3       3       3       3       3       3       3       3       3       3 
##   29.16    29.3   29.33    29.5   29.54   29.57   29.58   29.69   29.82   29.84   29.89 
##       3       3       3       3       3       3       3       3       3       3       3 
##   29.95   30.01   30.05   30.12    30.3   30.38   30.53   30.57   30.59   30.66   30.67 
##       3       3       3       3       3       3       3       3       3       3       3 
##   30.68   30.69   30.71    30.8   30.83   30.95   31.05   31.18   31.22    31.3   31.38 
##       3       3       3       3       3       3       3       3       3       3       3 
##   31.53   31.55   31.63   31.71   31.77   31.93   31.99   32.05   32.11   32.29   32.32 
##       3       3       3       3       3       3       3       3       3       3       3 
##   32.63   32.66   32.72    33.1   33.44   20.41      21   21.12   21.19   21.31   21.68 
##       3       3       3       3       3       2       2       2       2       2       2 
## (Other) 
##     736 
```

Obviously, something is wrong.

4.7.2 Fixing Problems

As seen above, weight records are currently stored as factor which they shouldn't. So how do we fix this?

Firstly, let's try an intuitive `as.numeric()` approach which attempts to convert all values contained within a vector into numeric records.

```
Data_df$Weight <- as.numeric(Data_df_base$Weight)
summary(Data_df$Weight)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max. 
##       1     206     351     345     494     687 
```

Apparently, this didn't do the trick since weight data values (recorded in g) below 13 and above 40 are highly unlikely for *Passer domesticus*.

Sometimes, the `as.numeric()` can be made more powerful by handing it data of class `character`. To do so, simply combine `as.numeric()` with `as.character()` as shown below.

```
Data_df$Weight <- as.numeric(as.character(Data_df_base$Weight))
summary(Data_df$Weight)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.    NA's 
##      19      26      30      29      32     420      66 
```

That still didn't resolve our problem. Weight measurements were taken for all study organisms and so there shouldn't be any `NA`s and yet we find 66.

Interestingly enough this is the exact same number as observations available for Siberia. A closer look at the data frame shows us that weight data for Siberia has been recorded with commas as decimal delimiters whilst the rest of the data set utilises dots.

Fixing this is not necessarily difficult but it is an erroneous issue for data handling which comes up often and is easy to avoid. Getting rid of the flaws is as simple as using the `gsub()` function contained within the `dplyr` package.

```
Data_df$Weight <- as.numeric(gsub(pattern = ",", replacement = ".", x = Data_df_base$Weight))
summary(Data_df$Weight)
```

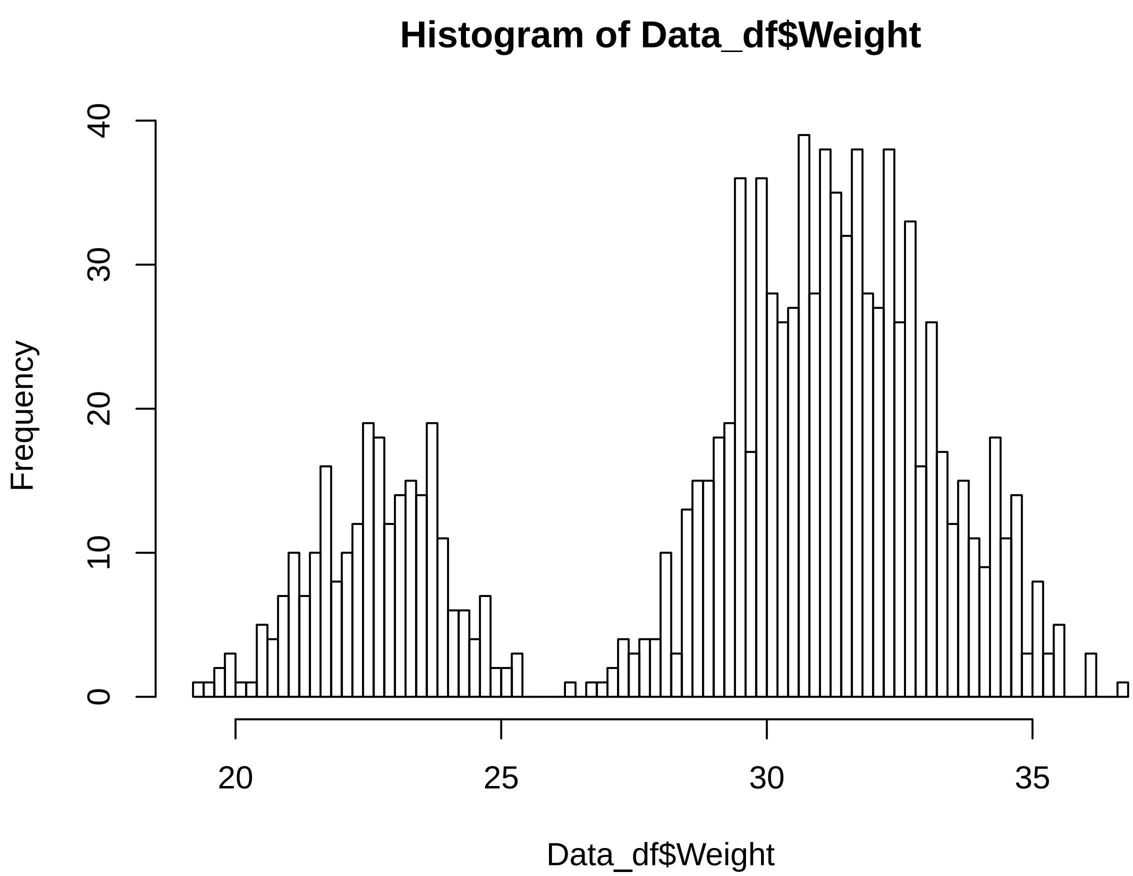
```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##      19      28      31      30      32     420
```

There is one data record left hat exceeds the biologically viable span for body weight records of *Passer domesticus*. This data record holds the value 420. Since this is unlikely to be a simple mistake of placing the decimal delimiter in the wrong place (both 4.2 and 42 grams are also not feasible weight records for house sparrows), we have to delete the weight data record in question:

```
Data_df$Weight[which(Data_df_base$Weight == 420)] <- NA
summary(Data_df$Weight)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.    NA's
##      19      28      31      29      32      37       1
```

```
hist(Data_df$Weight, breaks = 100)
```



We finally fixed it!

4.8 Height

Variable Class Expectation: `numeric` (height is a continuous metric)

4.8.1 Identifying Problems

Let's asses our Height records for our *Passer domesticus* individuals and check whether they behave as expected:

```
class(Data_df$Height)
```

```
## [1] "numeric"
```

```
summary(Data_df$Height)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##       1      14      15      15      16     135
```

Again, some of our data don't behave the way the should (a 135.4 or 1.35 cm tall sparrow are just absurd).

4.8.2 Fixing Problems

Height (or "Length") records of *Passer domesticus* should fall roughly between 10cm and 22cm. Looking at the data which exceed these thresholds, it is apparent that these are generated simply through misplaced decimal delimiters. So we fix them as follows and use a histogram to check if it worked.

```
Data_df$Height[which(Data_df$Height < 10)]  # decimal point placed wrong here
```

```
## [1] 1.4 1.4
```

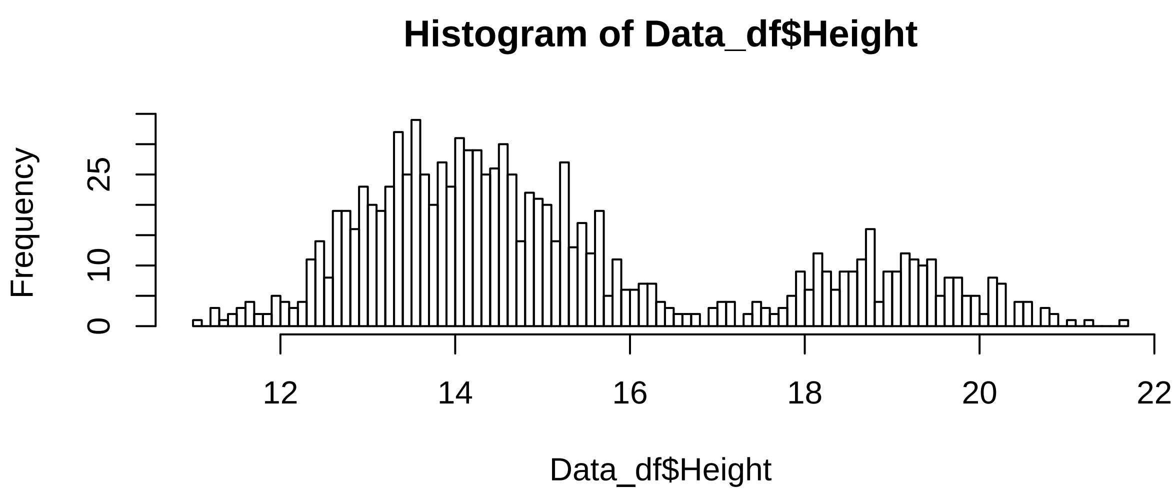
```
Data_df$Height[which(Data_df$Height < 10)] <- Data_df$Height[which(Data_df$Height <
    10)] * 10  # FIXED IT!
Data_df$Height[which(Data_df$Height > 22)]  # decimal point placed wrong here
```

```
## [1] 127 135
```

```
Data_df$Height[which(Data_df$Height > 22)] <- Data_df$Height[which(Data_df$Height >
    22)]/10  # FIXED IT!
summary(Data_df$Height)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##    11.1    13.5    14.5    15.2    16.2    21.7
```

```
hist(Data_df$Height, breaks = 100)
```



We finally fixed it!

4.9 Wing Chord

Variable Class Expectation: `numeric` (wing chord is a continuous metric)

4.9.1 Identifying Problems

Let's asses our Wing Chord records for our *Passer domesticus* individuals and check whether they behave as expected:

```
class(Data_df$Wing.Chord)
```

```
## [1] "numeric"
```

```
summary(Data_df$Wing.Chord)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##     6.4     6.8     7.0     7.3     7.4     9.0
```

Indeed, they do behave just like we'd expect them to.

4.9.2 Fixing Problems

We don't need to fix anything here.

4.10 Colour

Variable Class Expectation: `factor` (three levels: black, grey, brown)

4.10.1 Identifying Problems

Let's asses our Colour records for our *Passer domesticus* individuals and check whether they behave as expected:

```
class(Data_df$Colour)
```

```
## [1] "factor"
```

```
summary(Data_df$Colour)
```

```
##                  Black          Bright black                 Brown                  Grey
##                    356                     1                   298                   412
## Grey with black spots
##                      1
```

Some of the colour records are very odd.

4.10.2 Fixing Problems

The colour records "Bright black" and "Grey with black spots" should be "Grey". Someone clearly got too eager on the assignment of colours here. The fix is as easy as identifying the data records which are "too precise" and overwrite them with the correct assignment:

```
Data_df$Colour[which(Data_df$Colour == "Bright black")] <- "Grey"
Data_df$Colour[which(Data_df$Colour == "Grey with black spots")] <- "Grey"
Data_df$Colour <- droplevels(Data_df$Colour)  # drop unused factor levels
summary(Data_df$Colour)  # FIXED IT!
```

```
## Black Brown  Grey
##   356   298   414
```

We finally fixed it!

4.11 Sex

Variable Class Expectation: `factor` (two levels: male and female)

4.11.1 Identifying Problems

Let's asses our Climate records for our *Passer domesticus* individuals and check whether they behave as expected:

```
class(Data_df$Sex)
```

```
## [1] "factor"
```

```
summary(Data_df$Sex)
```

```
## Female   Male
##    524    544
```

Indeed, they do behave just like we'd expect them to.

4.11.2 Fixing Problems

We don't need to fix anything here.

4.12 Nesting Site

Variable Class Expectation: `factor` (two levels: shrub and tree)

4.12.1 Identifying Problems

Let's asses our Nesting Site records for our *Passer domesticus* individuals and check whether they behave as expected:

```
class(Data_df$Nesting.Site)
```

```
## [1] "factor"
```

```
summary(Data_df$Nesting.Site)
```

```
## Ground     NA  Shrub   Tree   NA's
##      1    498    292    231     46
```

4.12.2 Fixing Problems

One individual is recording to be nesting on the ground. This is something house sparrows don't do. Therefore, we have to assume that this individual is not even a *Passer domesticus* to begin with.

The only way to solve this is to remove all observations pertaining to this individual:

```
Data_df <- Data_df[-which(Data_df$Nesting.Site == "Ground"), ]
summary(Data_df$Nesting.Site)
```

```
## Ground     NA  Shrub   Tree   NA's
##      0    498    292    231     46
```

We just deleted a data record. This affects the flock size of the flock it belongs to (basically, this column contains hard-coded values) which we are going to deal with later.
Still, there are manually entered `NA` records present which we have to get rid of. These can be fixed easily without altering column classes and simply making use of logic by indexing their dependencies on other column values. The nesting site for a data record where sex reads "Male" has to be `NA`.

```
Data_df$Nesting.Site[which(Data_df$Sex == "Male")] <- NA
Data_df$Nesting.Site <- droplevels(Data_df$Nesting.Site)  # drop unused factor levels
summary(Data_df$Nesting.Site)  # FIXED IT!
```

```
## Shrub  Tree  NA's
##   292   231   544
```

4.13 Nesting Height

Variable Class Expectation: `numeric` (continuous records in two clusters corresponding to shrubs and trees)

4.13.1 Identifying Problems

Let's asses our Nesting Height records for our *Passer domesticus* individuals and check whether they behave as expected:

```
class(Data_df$Nesting.Height)
```

```
## [1] "factor"
```

```
summary(Data_df$Nesting.Height)
```

```
##      NA   36.01   50.74   50.85   58.32 1001.73  1005.5 1006.56 1008.47 1009.31 1010.54
##     498       2       2       2       2       1       1       1       1       1       1
## 1012.02 1012.53 1013.39 1015.71 1019.11 1024.55 1024.87  1029.9 1031.27 1046.35 1048.34
##       1       1       1       1       1       1       1       1       1       1       1
## 1053.22 1053.43 1053.71 1057.81 1059.16 1063.79 1064.28  1064.7 1068.52 1069.93 1075.86
##       1       1       1       1       1       1       1       1       1       1       1
## 1077.88 1084.29 1088.43 1090.58 1094.36   11.78 1103.09 1113.41 1115.33 1124.95 1128.81
##       1       1       1       1       1       1       1       1       1       1       1
## 1134.07 1136.43 1146.02 1146.18 1151.29 1152.71 1163.69 1167.03 1169.04 1181.56 1198.31
##       1       1       1       1       1       1       1       1       1       1       1
##   12.22 1207.51 1208.59 1228.62 1241.25 1246.93 1247.78  1254.7 1257.61 1257.78 1258.52
##       1       1       1       1       1       1       1       1       1       1       1
## 1259.49 1261.85 1264.52 1294.14 1298.12   13.21  1301.4 1304.03 1307.51 1310.76  1311.9
##       1       1       1       1       1       1       1       1       1       1       1
## 1315.18 1315.22 1318.44 1323.93 1324.25 1329.65 1343.61 1345.61 1354.22 1354.49 1368.18
##       1       1       1       1       1       1       1       1       1       1       1
## 1385.62 1390.81   14.69   14.71 1406.22 1407.76   141.9  1417.2 1428.51 1429.67 (Other)
##       1       1       1       1       1       1       1       1       1       1     422
##    NA's
##      46
```

There are obviously some issues here.

4.13.2 Fixing Problems

Nesting height is a clear example of a variable that should be recorded as `numeric` and yet our data frame currently stores them as factor.

Our first approach to fixing this, again, is using the `as.numeric()` function.

```
summary(as.numeric(Data_df$Nesting.Height))
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.    NA's
##       1       1      15     135     269     521      46
```

Clearly, something went horribly wrong here. When taking a closer look, the number of 1s is artificially inflated. This is due to the `NA`s contained within the data set. These are currently stored as characters since they have been entered into the Excel sheet itself. The `as.numeric()` function transforms these into 1s.

One way of circumventing this issue is to combine the `as.numeric()` function with the `as.character()` function.

```
Data_df$Nesting.Height <- as.numeric(as.character(Data_df$Nesting.Height))
summary(Data_df$Nesting.Height)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.    NA's
##      12      42      65     481     951    1951     544
```

This quite clearly fixed our problems.

4.14 Number of Eggs

Variable Class Expectation: `numeric` (no a priori knowledge of levels)

4.14.1 Identifying Problems

Let's asses our Number of Eggs records for our *Passer domesticus* individuals and check whether they behave as expected:

```
class(Data_df$Number.of.Eggs)
```

```
## [1] "factor"
```

```
summary(Data_df$Number.of.Eggs)
```

```
##    NA     0     1    10     2     3     4     8     9  NA's
##   498    46    79    16   106   130    36    16    94    46
```

One very out of the ordinary record is to be seen.

4.14.2 Fixing Problems

Number of eggs is another variable which should be recorded as `numeric` and yet is currently stored as factor.

Our first approach to fixing this, again, is using the `as.numeric()` function.

```
summary(as.numeric(Data_df$Number.of.Eggs))
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.    NA's
##       1       1       2       3       6       9      46
```

Again, this didn't do the trick. The number of 1s might be inflated and we expect exactly 544 (number of males) `NA`s since number of eggs have only been recorded for female house sparrows.

We already know that improperly stored `NA` records are prone to causing an inflation of data records of value 1. We also remember that head and tail of our data frame hold different types of `NA` records. Let's find out who entered `NA`s correctly:

```
unique(Data_df$Site[which(is.na(Data_df$Egg.Weight))])
```

```
## [1] Falkland Isles
## 11 Levels: Australia Belize Falkland Isles French Guiana Louisiana Manitoba ... United Kingdom
```

The code above identifies the sites at which proper `NA` recording has been done. The Falkland Isle team did it right (`NA` fields in Excel were left blank). Fixing this is actually a bit more challenging and so we do the following:

```
# make everything into characters
Data_df$Number.of.Eggs <- as.character(Data_df$Number.of.Eggs)
# writing character NA onto actual NAs
Data_df$Number.of.Eggs[which(is.na(Data_df$Number.of.Eggs))] <- "  NA"
# make all character NAs into proper NAs
Data_df$Number.of.Eggs[Data_df$Number.of.Eggs == "  NA"] <- NA
# make everything numeric
Data_df$Number.of.Eggs <- as.numeric(as.character(Data_df$Number.of.Eggs))
summary(Data_df$Number.of.Eggs)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.    NA's
##       0       2       3       4       4      10     544
```

We did it!

4.15 Egg Weight

Variable Class Expectation: `numeric` (another weight measurement that needs to be continuous)

4.15.1 Identifying Problems

Let's asses our Egg Weight records for our *Passer domesticus* individuals and check whether they behave as expected:

```
class(Data_df$Egg.Weight)
```

```
## [1] "factor"
```

```
summary(Data_df$Egg.Weight)
```

```
##      NA    2.59    2.75    2.71    2.55    2.69    2.83    2.63    2.66    2.72    2.84
##     544      14      10       9       8       8       8       7       7       7       7
##    2.98    2.05     2.6    2.64    2.77     2.8    2.81    2.86    2.89    1.96    2.04
##       7       6       6       6       6       6       6       6       6       5       5
##    2.45    2.52    2.57    2.62    2.68    2.74    2.79    2.93    2.97    3.03    3.17
##       5       5       5       5       5       5       5       5       5       5       5
##    1.94    1.95    2.11    2.13    2.14    2.17    2.18    2.21    2.28    2.34    2.36
##       4       4       4       4       4       4       4       4       4       4       4
##    2.37    2.51    2.58    2.67     2.7    2.78    2.82    2.85    2.87    2.91    2.92
##       4       4       4       4       4       4       4       4       4       4       4
##    2.99    3.23    1.86     1.9    1.93    2.08    2.19    2.27    2.29    2.46    2.47
##       4       4       3       3       3       3       3       3       3       3       3
##    2.53    2.61    2.65    2.73    2.76     2.9    2.94    2.95       3    3.04    3.14
##       3       3       3       3       3       3       3       3       3       3       3
##    3.15    3.21    3.25     3.3    3.39    1.87    1.91    1.92    1.99       2    2.01
##       3       3       3       3       3       2       2       2       2       2       2
##    2.03    2.06     2.1    2.12    2.15    2.16    2.23    2.26    2.39    2.41 (Other)
##       2       2       2       2       2       2       2       2       2       2      69
##    NA's
##      46
```

4.15.2 Fixing Problems

Egg weight should be recorded as `numeric` and yet is currently stored as factor. Our first approach to fixing this, again, is using the `as.numeric()` function again.

```
summary(as.numeric(Data_df$Egg.Weight))
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.    NA's
##       1       1       1      38      80     156      46
```

Something is wrong here. Not enough `NA`s are recorded. We expect exactly 590 `NA`s (Number of males + Number of Females with zero eggs). Additionally, there are way too many 1s. Our problem, again, lies with the way the `NA`s have been entered into the data set from the beginning and so we use the following fix again.

```
# make everything into characters
Data_df$Egg.Weight <- as.character(Data_df$Egg.Weight)
# writing character NA onto actual NAs
Data_df$Egg.Weight[which(is.na(Data_df$Egg.Weight))] <- "  NA"
# make all character NAs into proper NAs
Data_df$Egg.Weight[Data_df$Egg.Weight == "  NA"] <- NA
# make everything numeric
Data_df$Egg.Weight <- as.numeric(as.character(Data_df$Egg.Weight))
summary(Data_df$Egg.Weight)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.    NA's
##       2       2       3       3       3       4     590
```

4.16 Flock

Variable Class Expectation: `factor` (each sparrow was assigned to one particular flock)

4.16.1 Identifying Problems

Let's asses our Flock records for our *Passer domesticus* individuals and check whether they behave as expected:

```
class(Data_df$Flock)
```

```
## [1] "factor"
```

```
summary(Data_df$Flock)
```

```
##   A   B   C   D   E
## 194 244 214 186 229
```

Indeed, they do behave just like we'd expect them to.

4.16.2 Fixing Problems

We don't need to fix anything here.

4.17 Home Range

Variable Class Expectation: `factor` (three levels: small, medium, large)

4.17.1 Identifying Problems

Let's asses our Home Range records for our *Passer domesticus* individuals and check whether they behave as expected:

```
class(Data_df$Home.Range)
```

```
## [1] "factor"
```

```
summary(Data_df$Home.Range)
```

```
##  Large Medium  Small
##    269     99    699
```

Indeed, they do behave just like we'd expect them to.

4.17.2 Fixing Problems

We don't need to fix anything here.

4.18 Flock Size

Variable Class Expectation: `numeric` (continuous measurement of how many sparrows are in each flock - measured as integers)

4.18.1 Identifying Problems

Let's asses our Flock Size records for our *Passer domesticus* individuals and check whether they behave as expected:

```
class(Data_df$Flock.Size)
```

```
## [1] "numeric"
```

```
summary(Data_df$Flock.Size)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##       7      16      19      26      31      58
```

Indeed, they do behave just like we'd expect them to.

4.18.2 Fixing Problems

We don't need to fix anything here.

4.19 Predator Presence

Variable Class Expectation: `factor` (two levels: yes and no)

4.19.1 Identifying Problems

Let's asses our Predator Presence records for our *Passer domesticus* individuals and check whether they behave as expected:

```
class(Data_df$Predator.Presence)
```

```
## [1] "factor"
```

```
summary(Data_df$Predator.Presence)
```

```
##  No Yes
## 357 710
```

Indeed, they do behave just like we'd expect them to.

4.19.2 Fixing Problems

We don't need to fix anything here.

4.20 Predator Type

Variable Class Expectation: `factor` (three levels: Avian, Non-Avian, and `NA`)

4.20.1 Identifying Problems

Let's asses our Predator Type records for our *Passer domesticus* individuals and check whether they behave as expected:

```
class(Data_df$Predator.Type)
```

```
## [1] "factor"
```

```
summary(Data_df$Predator.Type)
```

```
##     Avian      Hawk        NA Non-Avian
##       240       250       357       220
```

Something doesn't sit well here.

4.20.2 Fixing Problems

Someone got overly eager when recording Predator Type and specified the presence of a hawk instead of taking down "Avian". We fix this as follows:

```
Data_df$Predator.Type[which(Data_df$Predator.Type == "Hawk")] <- "Avian"
summary(Data_df$Predator.Type)
```

```
##     Avian      Hawk        NA Non-Avian
##       490         0       357       220
```

This fixed it but there are still manually entered `NA` records present which we have to get rid of. These can be fixed easily without altering column classes and simply making use of logic by indexing their dependencies on other column values. The predator type for a data record where predator presence reads "No" has to be `NA`.

```
Data_df$Predator.Type[which(Data_df$Predator.Presence == "No")] <- NA
Data_df$Predator.Type <- droplevels(Data_df$Predator.Type)  # drop unused factor levels
summary(Data_df$Predator.Type)  # FIXED IT!
```

```
##     Avian Non-Avian      NA's
##       490       220       357
```

4.21 Redundant Data

Our data contains redundant columns (i.e.: columns whose data is present in another column already). These are (1) Flock Size (data contained in Flock column) and (2) Flock.Size (data contained in Index column). The fix to this is as easy as removing the columns in question.

```
Data_df <- within(Data_df, rm(Flock.Size, Site))
dim(Data_df)
```

```
## [1] 1067   18
```

Fixed it!

By doing so, we have gotten rid of our flock size problem stemming from the deletion of a data record. You could also argue that the columns `Site` and `Index` are redundant. We keep both for quality-of-life when interpreting our results (make use of `Sites`) and coding (make use os `Index`).

5. Saving The Fixed Data Set

We fixed out entire data set! The data set is now ready for use.

Keep in mind that the data set I provided you with was relatively clean and real-world messy data sets can be far more difficult to clean up.

Before going forth, we need to save it. **Attention:** don't overwrite your initial data file!

5.1 Final Check

Before exporting you may want to ensure that everything is in order and do a final round of data inspection. This can be achieved by running the automated `summary()` command from earlier again as follows. I am not including the output here to save some space.

```
for (i in 1:dim(Data_df)[2]) {
    print(colnames(Data_df)[i])
    print(summary(Data_df[, i]))
    print("-----------------------------------------------------")
}
```

Everything checks out. Let's save our final data frame.

5.2 Exporting The Altered Data

Since Excel is readily available for viewing data outside of R, I like to save my final data set in excel format as can be seen below. Additionally, I recommend saving your final data frame as an RDS file. These are `R` specific data files which you will not be able to alter outside of `R` thus saving yourself from accidentally changing records when only trying to view your data. On top of that, RDS files take up less space than either Excel or TXT files do.

```
# saving in excel sheet
write.csv(Data_df, file = paste(Dir.Data, "/SparrowData_FIXED.csv", sep=""))
# saving as R data frame object
saveRDS(Data_df, file = paste(Dir.Data, "/1 - Sparrow_Data_READY.rds", sep=""))
```