Assignment 6 for Statistical Computing and Empirical Methods:

Henry Reeve

Introduction

This document describes your fifth assignment for Statistical Computing and Empirical Methods (Unit EMATM0061) on the MSc in Data Science. Before starting the assignment it is recommended that you first watch video lecture 6 entitled “Tidy data and iteration”.

Begin by creating an Rmarkdown document with html output. You will need to load the Tidyverse library.

```
library(tidyverse)
```

1 Missing data and iteration

In this task we investigate the effect of missing data and imputation on plots.

The following function performs imputation by mean.

```
impute_by_mean<-function(x){

  mu<-mean(x,na.rm=1) # first compute the mean of x

  impute_f<-function(z){ # coordinate-wise imputation
    if(is.na(z)){
      return(mu) # if z is na replace with mean
    }else{
      return(z) # otherwise leave in place
    }
  }
  return(map_dbl(x,impute_f)) # apply the map function to impute across vector
}
```

Create a function called “impute_by_median” which imputes missing values based on the median of the sample, rather than the mean.

You can test your function on the following sample vector:

```
v<-c(1,2,NA,4)
impute_by_median(v)
```

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

Next generate a data frame with two variables "x" and "y". For our first variable "x" we have a sequence $(x_1, x_2, \ldots, x_n)$ where $x_1 = 0$, $x_n = 10$ and for each $i = 1, \ldots, n-1$, $x_{i+1} = x_i + 0.1$. For our second variable "y" we set $y_i = 5 \times x_i + 1$ for $i = 1, \ldots, n$. Generate data of this form and place within a data frame called "df_xy".

```
df_xy%>%head(5)
```

```
##     x   y
## 1 0.0 1.0
## 2 0.1 1.5
## 3 0.2 2.0
## 4 0.3 2.5
## 5 0.4 3.0
```

The **map2()** function is similar to the **map()** function but iterates over two variables in parallel rather than one. You can learn more here https://purrr.tidyverse.org/reference/map2.html (https://purrr.tidyverse.org/reference/map2.html). The following simple example shows you how **map2_dbl()** can be combined with the **mutate()** function.

```
df_xy%>%
  mutate(z=map2_dbl(x,y,~.x+.y))%>%
  head(5)
```

```
##     x   y   z
## 1 0.0 1.0 1.0
## 2 0.1 1.5 1.6
## 3 0.2 2.0 2.2
## 4 0.3 2.5 2.8
## 5 0.4 3.0 3.4
```

We will now use **map2_dbl()** to generate a new data frame with missing data.

First create a function "sometimes_missing" with two variables "index" and "value". The function should return NA if index is divisible by 5 and returns value otherwise.

Your function should produce the following outputs:

```
sometimes_missing(14,25)
```

```
## [1] 25
```

```
sometimes_missing(15,25)
```

```
## [1] NA
```

Next generate a new data frame called “df_xy_missing” with two variables “x” and “y”, but some missing data. For the first variable “x” we have a sequence $(x_1, \cdots, x_n)$, which is precisely the same as with “df_xy”. For the second variable “y” we have a sequence $(\tilde{y}_1, \cdots, \tilde{y}_n)$ where $\tilde{y}_i = \mathrm{NA}$ if i is divisible by 5 and $\tilde{y}_i = y_i$ for i not divisible by 5. To generate the dataframe “d_xy_missing” you may want to make use of the functions **row_number()**, **map2_dbl()**, **mutate()** as well as **sometimes_missing()**.

Check that the first ten rows of your data frame are as follows:

```
df_xy_missing%>%
  head(10)
```

```
##      x   y
## 1  0.0 1.0
## 2  0.1 1.5
## 3  0.2 2.0
## 4  0.3 2.5
## 5  0.4  NA
## 6  0.5 3.5
## 7  0.6 4.0
## 8  0.7 4.5
## 9  0.8 5.0
## 10 0.9  NA
```

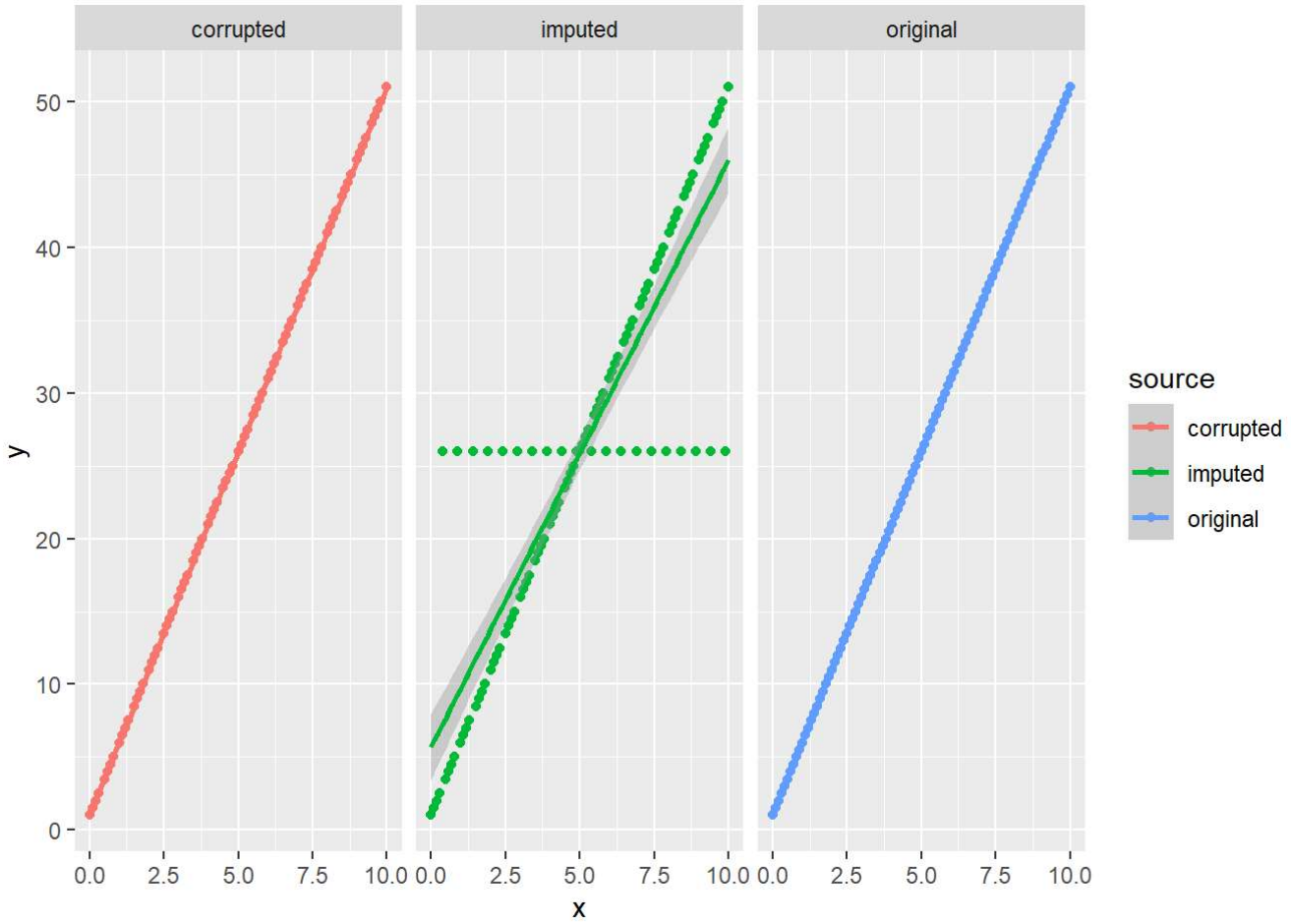
Create a new data frame “df_xy_imputed” with two variables “x” and “y”. For the first variable “x” we have a sequence $(x_1, \cdots, x_n)$, which is precisely the same as with “df_xy”. For the second variable “y” we have a sequence $(y'_1, \cdots, y'_n)$ which is formed from $(\tilde{y}_1, \cdots, \tilde{y}_n)$ by imputing any missing values with the median. To generate “df_xy_imputed” from “df_xy_missing” by applying a combination of the functions **mutate** and **impute_by_median()**.

Combine the dataframes df_xy, df_xy_missing and df_xy_impute within a single dataframe called df_combined, along with an additional column indicating the source of the data.

```
df_xy<-df_xy%>%
  mutate(source="original")

df_xy_missing<-df_xy_missing%>%
  mutate(source="corrupted")

df_xy_impute<-df_xy_impute%>%
  mutate(source="imputed")

df_combined<-rbind(df_xy,df_xy_missing,df_xy_impute)
```

Plot the original data, the corrupted data and the imputed data together together with a trend line for each sample.

```
ggplot(df_combined,aes(x=x,y=y,color=source))+geom_point()+
  facet_wrap(~source)+geom_smooth(method="lm")
```



Do the imputed values y_i' give reasonable estimates of the true values y_i?

2 Tidying data with pivot functions

In this task you will read in data from a spreadsheet and apply some data wrangling tasks to tidy that data.

First download the excel spreadsheet entitled “HockeyLeague.xlsx”. The excel file contains two spreadsheets - one with the wins for each team and one with the losses for each team. To read this spreadsheet into R we shall make use of the **readxl** library. You may need to install the library:

```
install.packages("readxl")
```

The following code shows how to read in a sheet within an excel file as a data frame. You will need to edit the “folder_path” variable to be the directory which contains your copy of the spreadsheet.

```
library(readxl) # load the readxl library

folder_path<-"your_folder_path..." # set this to the name of the directory containing "Ho
ckeyLeague.xlsx"

file_name<-"HockeyLeague.xlsx" # set the file name

file_path<-paste(folder_path,file_name,sep="") # create the file_path

wins_data_frame<-read_excel(file_path,sheet="Wins") # read of a sheet from an xl file
```

Inspect the first 3 rows of the first five columns:

```
wins_data_frame %>%
  select(1:5)%>%
  head(3)
```

```
## # A tibble: 3 x 5
##   ...1   `1990`   `1991`   `1992`   `1993`
##   <chr>  <chr>    <chr>    <chr>    <chr>
## 1 Ducks  30 of 50 11 of 50 30 of 50 12 of 50
## 2 Eagles 24 of 50 12 of 50 37 of 50 14 of 50
## 3 Hawks  20 of 50 22 of 50 33 of 50 11 of 50
```

A cell value of the form “a of b” means that a games were won out of a total of b for that season. For example, the element for the “Ducks” row of the “1990” column is “30 of 50” meaning that 30 out of 50 games were won that season.

Is this tidy data?

Now apply your data wrangling skills to transform the “wins_data_frame” data frame object into a data frame called “wins_tidy” which contains the same information but has just four columns entitled “Team”, “Year”, “Wins”, “Total”. The “Team” column should contain the team name, the “Year” column should contain the year, the “Wins” column

should contain the number of wins for that season and the “Total” column the total number of games for that season. The first column should be of character type and the remaining columns should be of integer type. You can do this by combining the following functions: **rename()**, **pivot_longer()**, **mutate()** and **separate()**.

You can check the shape of your data frame and the first five rows as follows:

```
wins_tidy%>% dim() # check the dimensions
```

```
## [1] 248   4
```

```
wins_tidy%>%head(5) # inspect the top 5 rows
```

```
## # A tibble: 5 x 4
##   Team   Year  Wins Total
##   <chr> <int> <int> <int>
## 1 Ducks  1990    30    50
## 2 Ducks  1991    11    50
## 3 Ducks  1992    30    50
## 4 Ducks  1993    12    50
## 5 Ducks  1994    24    50
```

The “HockeyLeague.xlsx” also contains a sheet with the losses for each team by season. Apply a similar procedure to read the data from this sheet and transform that data into a dataframe called “losses_tidy” with four columns: “Team”, “Year”, “Losses”, “Total” which are similar to thos in the “wins_tidy” data frame except for the “Losses” column gives the number of losses for a given season and team, rather than the number of losses.

You may notice that the number of wins plus the number of losses for a given team, in a given year does not add up to the total. This is because some of the games are neither wins nor losses but draws. That is, for a given year the number of draws is equal to the total number of games minus the sum of the wins and losses.

Now combine your two data frames, “wins_tidy” and “losses_tidy”, into a single data frame entitled “hockey_df” which has 248 rows and 9 columns: A “Team” column which gives the name of the team as a character, the “Year” column which gives the season year, the “Wins” column which gives the number of wins for that team in the given year, the “Losses” column which gives the number of losses for that team in the given year and the “Draws” column which gives the number of draws for that team in the given year, the “Wins_rt” which gives the wins as a proportion of the total number of games (ie. Wins/Total) and similarly the “Losses_rt” and the “Draws_rt” which gives the losses and

draws as a proportion of the total, respectively. To do this you can make use of the **mutate()** function. You may also want to utilise the **across()** function for a slightly neater solution.

The top five rows of your data frame should look as follows:

```
hockey_df%>% head(5)
```

```
## # A tibble: 5 x 9
##   Team   Year  Wins Total Losses Draws Wins_rt Losses_rt Draws_rt
##   <chr> <int> <int> <int>  <int> <int>   <dbl>     <dbl>    <dbl>
## 1 Ducks  1990    30    50     20     0    0.6       0.4      0
## 2 Ducks  1991    11    50     37     2    0.22      0.74     0.04
## 3 Ducks  1992    30    50      1    19    0.6       0.02     0.38
## 4 Ducks  1993    12    50     30     8    0.24      0.6      0.16
## 5 Ducks  1994    24    50      7    19    0.48      0.14     0.38
```

To conclude this task generate a summary data frame which displays, for each team, the median win rate, the mean win rate, the median loss rate, the mean loss rate, the median draw rate and the mean draw rate. The number of rows in your summary should equal the number of teams. These should be sorted in descending order or median win rate. You may want to make use of the following functions: **select()**, **group_by()**, **across()**, **arrange()**.

```
hockey_df%>%
  select(-Wins,-Draws,-Losses)%>%
  group_by(Team)%>%
  summarise(across(starts_with(c("Wins","Losses","Draws")),list(md=median,mn=mean),
                   .names="{substring(.col,1,1)}_{.fn}"))%>%
  arrange(desc(W_md))
```

```
## # A tibble: 8 x 7
##   Team         W_md  W_mn  L_md  L_mn  D_md  D_mn
##   <chr>       <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
## 1 Eagles      0.45  0.437 0.25  0.279 0.317 0.284
## 2 Penguins    0.45  0.457 0.3   0.310 0.133 0.232
## 3 Hawks       0.417 0.388 0.233 0.246 0.32  0.366
## 4 Ducks       0.383 0.362 0.34  0.333 0.25  0.305
## 5 Owls        0.32  0.333 0.3   0.33  0.383 0.337
## 6 Ostriches   0.3   0.309 0.4   0.395 0.267 0.296
## 7 Storks      0.3   0.284 0.22  0.283 0.48  0.433
## 8 Kingfishers 0.233 0.245 0.34  0.360 0.4   0.395
```

3 Most correlated variables

This data wrangling task is more challenging. Only complete this task if you have sufficient time.

The objective is to investigate, for each numerical variable within a data set, which other numerical variables have the largest correlation (in absolute value).

In lecture 6 we introduced the following function called “max_cor_var”. The function entitled “max_cor_var” takes as input a data frame “df” and a column name “col_name”. It then extracts the variable with name col_name and determines which other numerical variables within the data set have the highest correlation (in absolute value) with that variable. It then returns a data frame containing the name of the variable “var_name” and the corresponding correlation “cor”. Begin by making sure you understand the structure of the function.

```
max_cor_var<-function(df,col_name){ # function to determine the variable with maximal cor
relation

  v_col<-df%>%select(all_of(col_name)) # extract variable based on col_name

  df_num<-df%>%
    select_if(is.numeric)%>%
    select(-all_of(col_name)) # select all numeric variables excluding col_name

  correlations<-unlist(map(df_num,
                           function(x){cor(x,v_col,use="complete.obs")})) # compute corre
lations with all other numeric variables

  max_abs_cor_var<-names(which(abs(correlations)==max(abs(correlations)))) # extract the
variable name
  cor<-as.double(correlations[max_abs_cor_var]) # compute the correlation

  return(data.frame(var_name=max_abs_cor_var,cor=cor)) # return dataframe
}
```

Next generate a new function called “top_correlates_by_var” which takes input a data frame “df” and outputs a data frame with a single row. The column names of this output data frame should coincide with the names of the numerical columns within the input dataframe “df”. For each column name, the value should be equal to variable name corresponding to the numerical variable which has the highest level of correlation (in absolute value) to the variable with that column name, but is not equal to it.

You can test your function as follows. By using the Palmer penguins data set you should obtain the following output.

```
library(palmerpenguins)

penguins%>%
  top_correlates_by_var()
```

```
## # A tibble: 1 x 5
##   bill_length_mm   bill_depth_mm    flipper_length_mm body_mass_g    year
##   <chr>            <chr>            <chr>             <chr>          <chr>
## 1 flipper_length_~ flipper_length_~ body_mass_g       flipper_lengt~ flipper_le~
```

Next use a combination of the functions **group_by()**, **nest()**, **mutate()**, **select()**, **unnest()** together with your new function **top_correlates_by_var()** to determine those variables with highest correlation, broken down by species of penguin. Your results should be as follows.

```
## # A tibble: 3 x 5
## # Groups:   species [3]
##   species   bill_length_mm bill_depth_mm  flipper_length_mm body_mass_g
##   <fct>     <chr>          <chr>          <chr>             <chr>
## 1 Adelie    body_mass_g    body_mass_g    body_mass_g       bill_depth_mm
## 2 Gentoo    body_mass_g    body_mass_g    bill_depth_mm     bill_depth_mm
## 3 Chinstrap bill_depth_mm  bill_length_mm body_mass_g       flipper_length_mm
```