Code

Data Wrangling (Data Preprocessing)

Practical assessment 2

Yan Hok Yin

7 Oct 2022

Setup

Hide

```
library(readr)
library(dplyr)
library(magrittr)
library(outliers)
library(MVN)
library(here)
library(tidyr)
library(ggplot2)
library(kableExtra)
library(stringr)
library(splitstackshape)
library(lubridate)
library(editrules)
library(forecast)
library(ggplot2)
```

Student names, numbers and percentage of contributions

Group information

Student name	Student number	Percentage of contribution
Yan Hok Yin	s3952322	33.33%
Sukhum Boondecharak	s3940976	33.33%
Yong Pui Tung	s3934929	33.33%

Executive Summary

The five main processes of data preprocessing are “get, understand, tidy & manipulate, scan, and transform”. All procedures and the collection of tasks required to clean up all types of messy data have been accomplished.

The two data sets are imported into R using the readr package after being downloaded in csv format from Kaggle. After that, we remove the duplicate value that is discovered in the data set and combine the information using left join function by the shared variable between the two data sets. In addition, to understand the variables in the data set and the significance of each value , the dimensions and structure of the combined data “songs” are evaluated. Following Hadley Wickham’s “Tidy Data” principles (Wickham and Grolemund (2016)), brackets in the values of genre column are removed, and each value has been separated into its own cell. When dealing with a huge data frame, it is often preferred to focus solely on a few key variables. As a result, a new data frame named “songs_selected” is created and the data is organized based on the popularity and track name of the songs. In order to make the time data more understandable after the data has been cleaned up, we have created a new variable (duration_min) by mutating the original variable (duration_ms). For missing values, special values, and obvious errors, we have gone through the scanning procedure. The NA value is successfully translated into “uncategorized,” and no special value is discovered. We also use established rules to search for glaring mistakes. The observer outlier in the data set has been identified with the use of boxplot and z-score. We normalize the data at the last stage by using the appropriate transformations (logarithm and square root).

Data

The first data set (unpopular_songs.csv), which contains a single sheet with the label “unpopular songs,” displays over 10,000 unpopular songs discovered on the music streaming service Spotify. The second data set (z_genre_of_artists.csv) contains a single sheet with the name “z genre of artists” that includes information of more than 1,700 artists with the associated genres categorized by Spotify. One song can belong to multiple categories. Both data sets’ updates is completed since August 2022. The popularity rating ranges from 0 to 100, based on the information gathered by Spotify. The data set only displays the songs that are ranked from 0 to 18 which are regarded as unpopular songs.

Both data sets are downloaded from Kaggle at link1 (https://www.kaggle.com/datasets/estienneggx/spotify-unpopular-songs?select=unpopular_songs.csv) and link2 (https://www.kaggle.com/datasets/estienneggx/spotify-unpopular-songs?select=z_genre_of_artists.csv)

Import Data

The “readr” package is needed to import and read the data from the csv files that contain the data sets. For “unpopular_songs.csv” and “z_genre_of_artists.csv”, we load the data into the data frames “unpopular_songs” and “genre” respectively.

Viewing the data sets

We first have a glimpse of the first 6 rows of the two data sets. We use head() function to view the first 6 rows of the both data sets.

Hide

```
# Show the first six rows of "unpopular_songs" dataframe
head(unpopular_songs)
```

danceability	ener...	...	loudness	...	speechiness	acousticness	instrumentalness	live
<dbl>	<dbl>	<dbl>	<dbl>	<dbl>	<dbl>	<dbl>	<dbl>	
0.530	0.770	4	-6.633	0	0.0389	0.284	0.501000	
0.565	0.730	1	-6.063	1	0.0730	0.365	0.000000	
0.427	0.546	4	-8.727	1	0.0849	0.539	0.015200	
0.421	0.531	7	-5.516	1	0.0262	0.706	0.000208	
0.537	0.804	8	-7.378	0	0.1570	0.379	0.000489	
0.710	0.621	9	-7.879	0	0.0329	0.405	0.001900	

6 rows | 1-10 of 17 columns

Hide

```
# Show the first six rows of "genre" data frame
head(genre)
```

artist_name	artist_id	▶
<chr>	<chr>	

artist_name <chr>	artist_id <chr>
James Reeder	5YCUdcLdAbaYZcdZrxNzjU
Cold	0Gw3a3BkWLwsMLFbOBmo6Q
TG	7qwOsGanRCnWgty6lX05P4
Klaxons	2qlAMLpUyBjZgnzuFXXZXI
TG	7qwOsGanRCnWgty6lX05P4
Various Artists	0LyfQWJT6nXafLPZqxe9Of

6 rows | 1-2 of 3 columns

Number of observations

Hide

```
# Check the number of observations in "unpopular_songs"
unpopular_songs %>% count()
```

n <int>
10877

1 row

Hide

```
# Check the number of observations in "genre"
genre %>% count()
```

n <int>
1736

1 row

The count() results reveal that the variables “unpopular songs” and “genre” have 10,877 and 1,736 observations respectively. In order to identify and eliminate any duplicated observations, we use unique () or distinct() function to check the distinct number of observations. After eliminating duplicated observations, the number of unique observations are calculated by count() function.

Identify Duplicated Observation

Hide

```
# Check the distinct number of observations in "unpopular_songs"
unpopular_songs %>% unique() %>% count()
```

n <int>

```
                                                                       n
                                                                   <int>
                                                                   10877
1 row
```

Hide

```
# Check the distinct number of observations in "genre"
genre %>% distinct() %>% count()
```

```
                                                                       n
                                                                   <int>
                                                                    1476
1 row
```

As the row number reduces from 1,736 to 1,476 after using distinct() and count() functions, the result reveals that "genre" contains 1736 - 1476 = 260 duplicated rows.

Remove Duplicated Rows

Therefore, we use distinct() function to remove duplicated rows and then save it as a new data frame called "new_genre".

Hide

```
new_genre <- genre %>% distinct()
```

Merge Data

Finally, we combine the data sets using left join function, connecting by variables "track artist" = "artist name". A new data frame named "songs" is created.

Hide

```
songs <- unpopular_songs %>% left_join(new_genre, by=c("track_artist"="artist_name"))
```

Understand

Data Strucuture and Variable Type

In this part, we will inspect the data structure and variable types. Firstly, we check the dimensions of the data frame using dim() function.

Hide

```
# Checking the dimensions of songs
dim(songs)
```

```
[1] 10887    19
```

The output “10887 19” reveals that the data frame is a 10,887 by 19 data frame. It contains 10,887 rows and 19 columns.

Then we check variable types in the combined data set “songs” using str() function.

Hide

```
# Checking the structure of the data
str(songs)
```

```
spec_tbl_df [10,887 × 19] (S3: spec_tbl_df/tbl_df/tbl/data.frame)
 $ danceability    : num [1:10887] 0.53 0.565 0.427 0.421 0.537 0.71 0.419 0.565 0.54
7 0.533 ...
 $ energy          : num [1:10887] 0.77 0.73 0.546 0.531 0.804 0.621 0.821 0.624 0.56
0.785 ...
 $ key             : num [1:10887] 4 1 4 7 8 9 11 1 0 5 ...
 $ loudness        : num [1:10887] -6.63 -6.06 -8.73 -5.52 -7.38 ...
 $ mode            : num [1:10887] 0 1 1 1 0 0 0 1 1 1 ...
 $ speechiness     : num [1:10887] 0.0389 0.073 0.0849 0.0262 0.157 0.0329 0.0431 0.0
351 0.051 0.0481 ...
 $ acousticness    : num [1:10887] 0.284 0.365 0.539 0.706 0.379 0.405 0.0137 0.00442
0.551 0.591 ...
 $ instrumentalness: num [1:10887] 0.501 0 0.0152 0.000208 0.000489 0.0019 0.00365 0.
221 0.179 0 ...
 $ liveness        : num [1:10887] 0.744 0.237 0.368 0.11 0.323 0.103 0.127 0.108 0.1
37 0.162 ...
 $ valence         : num [1:10887] 0.623 0.511 0.435 0.383 0.543 0.546 0.343 0.655 0.
354 0.521 ...
 $ tempo           : num [1:10887] 120.1 130 78.3 85.1 139.9 ...
 $ duration_ms     : num [1:10887] 225696 158093 167262 236832 239400 ...
 $ explicit        : logi [1:10887] FALSE FALSE FALSE FALSE FALSE FALSE ...
 $ popularity      : num [1:10887] 2 2 2 2 2 2 2 2 2 2 ...
 $ track_name      : chr [1:10887] "No Regrets" "Wild Life" "Fangs" "Afterburner" ...
 $ track_artist    : chr [1:10887] "James Reeder" "James Reeder" "James Reeder" "Jame
s Reeder" ...
 $ track_id        : chr [1:10887] "6f2c4a9lNx8aowZJngv7cJ" "3fTs52jsDzSuVLsifxNKO8"
"6NPafqavrv0icaIHMQnXDy" "3vGmhxveURgmlZStvo0uc1" ...
 $ artist_id       : chr [1:10887] "5YCUdcLdAbaYZcdZrxNzjU" "5YCUdcLdAbaYZcdZrxNzjU"
"5YCUdcLdAbaYZcdZrxNzjU" "5YCUdcLdAbaYZcdZrxNzjU" ...
 $ genre           : chr [1:10887] "[]" "[]" "[]" "[]" ...
 - attr(*, "spec")=
  .. cols(
  ..   danceability = col_double(),
  ..   energy = col_double(),
  ..   key = col_double(),
  ..   loudness = col_double(),
  ..   mode = col_double(),
  ..   speechiness = col_double(),
  ..   acousticness = col_double(),
  ..   instrumentalness = col_double(),
  ..   liveness = col_double(),
  ..   valence = col_double(),
  ..   tempo = col_double(),
  ..   duration_ms = col_double(),
  ..   explicit = col_logical(),
  ..   popularity = col_double(),
  ..   track_name = col_character(),
  ..   track_artist = col_character(),
  ..   track_id = col_character()
  .. )
 - attr(*, "problems")=<externalptr>
```

Data Conversion

Most variables are numeric (e.g. valence, tempo, duration ms, etc). They stand for the music elements of the songs. There are units for some variables, including dB, BPM, and milliseconds. “Explicit” is a unique variable having a logic-based structure since it is captured from the original data as TRUE or FALSE. The remaining parameters (track name, track_artist, track_id, artist_id, and genre) take the data type of character.

Next, we will apply data type conversion. The ranking of songs is represented by popularity. 0 is the lowest rank and 18 is the highest in this data set. Therefore, we change the data type of “popularity” from numeric to ordered factor using factor() function.

Hide

```
# Converting the data type to factor
songs$popularity <- factor(songs$popularity,
                           levels=c("0", "1", "2", "3", "4", "5", "6", "7",
                                    "8", "9", "10", "11", "12", "13", "14",
                                    "15", "16", "17", "18"),
                           ordered = TRUE)
```

We then verify whether the transformation is successful or not using class() function. Transformation is successful if it returns “ordered” “factor”.

Hide

```
class(songs$popularity)
```

```
[1] "ordered" "factor"
```

Furthermore, we transform “mode” from numerical variable to factor. Observations in the mode column are represented by number 0 and 1. We label 0 as “major” and 1 as “minor” to avoid confusion and for easier understanding.

Hide

```
songs$mode <- factor(songs$mode,
                     levels=c("0","1"),
                     labels=c("Major","Minor"))
```

Hide

```
# Check the factor's level
levels(songs$mode)
```

```
[1] "Major" "Minor"
```

Data structure

Finally, we verify the entire data set’s structure once again using str() function.

Hide

```
# Verify the whole data structure again
str(songs)
```

```
spec_tbl_df [10,887 × 19] (S3: spec_tbl_df/tbl_df/tbl/data.frame)
 $ danceability    : num [1:10887] 0.53 0.565 0.427 0.421 0.537 0.71 0.419 0.565 0.54
7 0.533 ...
 $ energy          : num [1:10887] 0.77 0.73 0.546 0.531 0.804 0.621 0.821 0.624 0.56
0.785 ...
 $ key             : num [1:10887] 4 1 4 7 8 9 11 1 0 5 ...
 $ loudness        : num [1:10887] -6.63 -6.06 -8.73 -5.52 -7.38 ...
 $ mode            : Factor w/ 2 levels "Major","Minor": 1 2 2 2 1 1 1 2 2 2 ...
 $ speechiness     : num [1:10887] 0.0389 0.073 0.0849 0.0262 0.157 0.0329 0.0431 0.0
351 0.051 0.0481 ...
 $ acousticness    : num [1:10887] 0.284 0.365 0.539 0.706 0.379 0.405 0.0137 0.00442
0.551 0.591 ...
 $ instrumentalness: num [1:10887] 0.501 0 0.0152 0.000208 0.000489 0.0019 0.00365 0.
221 0.179 0 ...
 $ liveness        : num [1:10887] 0.744 0.237 0.368 0.11 0.323 0.103 0.127 0.108 0.1
37 0.162 ...
 $ valence         : num [1:10887] 0.623 0.511 0.435 0.383 0.543 0.546 0.343 0.655 0.
354 0.521 ...
 $ tempo           : num [1:10887] 120.1 130 78.3 85.1 139.9 ...
 $ duration_ms     : num [1:10887] 225696 158093 167262 236832 239400 ...
 $ explicit        : logi [1:10887] FALSE FALSE FALSE FALSE FALSE FALSE ...
 $ popularity      : Ord.factor w/ 19 levels "0"<"1"<"2"<"3"<..: 3 3 3 3 3 3 3 3 3 3
...
 $ track_name      : chr [1:10887] "No Regrets" "Wild Life" "Fangs" "Afterburner" ...
 $ track_artist    : chr [1:10887] "James Reeder" "James Reeder" "James Reeder" "Jame
s Reeder" ...
 $ track_id        : chr [1:10887] "6f2c4a9lNx8aowZJngv7cJ" "3fTs52jsDzSuVLsifxNKO8"
"6NPafqavrv0icaIHMQnXDy" "3vGmhxveURgmlZStvo0uc1" ...
 $ artist_id       : chr [1:10887] "5YCUdcLdAbaYZcdZrxNzjU" "5YCUdcLdAbaYZcdZrxNzjU"
"5YCUdcLdAbaYZcdZrxNzjU" "5YCUdcLdAbaYZcdZrxNzjU" ...
 $ genre           : chr [1:10887] "[]" "[]" "[]" "[]" ...
 - attr(*, "spec")=
  .. cols(
  ..   danceability = col_double(),
  ..   energy = col_double(),
  ..   key = col_double(),
  ..   loudness = col_double(),
  ..   mode = col_double(),
  ..   speechiness = col_double(),
  ..   acousticness = col_double(),
  ..   instrumentalness = col_double(),
  ..   liveness = col_double(),
  ..   valence = col_double(),
  ..   tempo = col_double(),
  ..   duration_ms = col_double(),
  ..   explicit = col_logical(),
  ..   popularity = col_double(),
  ..   track_name = col_character(),
  ..   track_artist = col_character(),
  ..   track_id = col_character()
  .. )
 - attr(*, "problems")=<externalptr>
```

Tidy & Manipulate Data I

Tidy Data Principles

According to Wickham and Grolemund (2016), there are three interrelated rules which make a data set tidy.

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

For “songs” data set, it obeys the first and second rules as each variable forms its own column and each track forms its own row. However, the order of columns is not very comprehensive as the column of track name is positioned after most variables, such as duration_ms, popularity, etc. This will be tidied up at later stage.

Nonetheless, it does not obey the third rule as each value does not have its own cell when we inspect the values under “genre” column. In order to ensure a consistent form that matches the semantics of the data set with the way it is stored, we need to reshape and tidy the data using various functions.

Tidying “genre” column

The observations must be cleaned up by removing any extra symbols or brackets, and the value order must be rearranged to ensure that each cell has just one value. First, we use the function str_sub to eliminate any undesired [ ] from each value in the genre column. After that, each value is split into its own cell and converted to long format using the cSpilt function. It is converted to long format as we follow Tidy Data Principle 2 that each observation must have its own row. To eliminate the unwanted ’ ’ from the value, str_sub function is applied.

Hide

```
songs$genre <- str_sub(songs$genre, start=2L, end=nchar(songs$genre)-1L)

songs <- cSplit(songs, "genre",",", direction="long")
```

Hide

```
songs$genre <- str_sub(songs$genre, start=2L, end=nchar(songs$genre)-1L)
```

Using select() function

When working with a large data frame, often we want to only assess specific variables. The select() function allows us to select variables. As dplyr functions will not modify inputs, we use the assignment operator to save the result as a new data set called songs_selected.

Hide

```
# Select columns by name using the pipe operator and save it in a new data set
songs_selected <- songs %>% select(popularity, track_name, track_artist, mode, durati
on_ms, genre)
```

Using arrange() function

We then apply arrange() function to order the data set by popularity and track name in ascending order.

Hide

```
# # Order the data set according to two variables
songs_selected %<>% arrange(popularity, track_name)
```

Using filter() function

We filter out songs with popularity equal to 0 and compile them into a new data frame called “unpopular genre” to identify the most unpopular songs within their respective genres. Some songs are not categorized, which contain the value NA. We use drop_na() to drop any NA values.

Hide

```
# Select songs with 0 popularity, then group them by genre
unpopular_genre <- songs_selected %>% filter(popularity == "0") %>% group_by(genre) %
>%
  summarise(number_of_most_unpopular_songs = n())

# Arrange them by descending order
unpopular_genre %>% arrange(desc(number_of_most_unpopular_songs)) %>% drop_na()
```

genre <chr>	number_of_most_unpopular_songs <int>
sleep	120
water	74
environmental	64
gangster rap	51
sound	49
houston rap	43
rap	41
southern hip hop	41
latin pop	35
dirty south rap	34

1-10 of 268 rows Previous **1** 2 3 4 5 6 ... 27 Next

Tidy & Manipulate Data II

After tidying the data, we would like to mutate two new variables from the existing variable duration_ms for easier understanding of the time data. Currently the duration is in millisecond, but we would like to change the unit to second and minute. Here we will use seconds_to_period() function from lubridate package to handle time data.

As the original data set provides the time data in millisecond, we will first divide duration_ms by 1000 first before applying seconds_to_period() function.

Using seconds_to_period() function

Hide

```
# Use seconds_to_period function to transform duration_ms
songlength <- seconds_to_period(songs_selected$duration_ms/1000)

# Checking if the transformation is successful
head(songlength)
```

```
[1] "3M 40.067S"           "5M 10.497S"           "4M 1.62700000000001S" "2M 20.773S"
"3M 40.427S"
[6] "3M 40.427S"
```

Hide

```
# Check the class of songlength
class(songlength)
```

```
[1] "Period"
attr(,"package")
[1] "lubridate"
```

Using mutate() function

We add one new variable using mutate() function from dplyr package.

Hide

```
# Create new variables "duration_min"
# Then drop the existing variable "duration_ms"
songs_selected %<>%
  mutate(songs_selected, duration_min = round(songlength)) %>%
  select (-duration_ms)
```

Then we check if the new variable has been added to the data frame.

Hide

```
# Checking the class of songlength
glimpse(songs_selected)
```

```
Rows: 11,384
Columns: 6
$ popularity   <ord> 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0…
$ track_name   <chr> "¿Comprendes Mendes?", "\"Caro Nome che il mio cor\" From Rigole
tto", "\"Til I Can't Take It Anymore…
$ track_artist <chr> "Control Machete", "Ileana Cotrubas", "London Philharmonic Orche
stra", "Pudge", "Gamma Ray", "Gamma …
$ mode         <fct> Minor, Minor, Minor, Minor, Minor, Minor, Minor, Minor, Minor, M
inor, Minor, Minor, Minor, Minor, Mi…
$ genre        <chr> NA, NA, NA, "wisconsin indie", "german metal", "german power met
al", "hard rock", "melodic metal", "…
$ duration_min <Period> 3M 40S, 5M 10S, 4M 2S, 2M 21S, 3M 40S, 3M 40S, 3M 40S, 3M 40
S, 3M 40S, 3M 40S, 3M 40S, 3M 40S, 3M…
```

The new variable is successfully added to the end of the data frame.

Scan I

In this part, we will scan the data for missing values, special values, and obvious errors. Firstly, we find if there is any missing values.

Scanning for Missing Values

As we read the data from two csv files, any missing value will be represented as NA for integer, < NA > for character variable. To identify missing values in the data frame, we can use is.na() function coupled with colSums() function to find the total missing values in each column in our data frame.

Hide

```
# Identify total number of NAs in each column
colSums(is.na(songs_selected))
```

```
  popularity   track_name track_artist         mode        genre duration_min 
           0            0            0            0          951            0 
```

We found that column “genre” has 951 NA values. As it is not numeric, we cannot recode the missing value using normal subsetting and assignment operations. The best way is to use replace_na() function from tidyr package to replace NA values. As NA means that the song is not categorized, we can replace NAs with character “uncategorized” for a better understanding.

Hide

```
# Replace NA values in "genre"
songs_selected$genre <- replace_na(songs_selected$genre, "uncategorized")

# Check total number of NAs in each column again to confirm there is no missing value
in the data frame
colSums(is.na(songs_selected))
```

```
  popularity   track_name track_artist         mode        genre duration_min 
           0            0            0            0            0            0 
```

Scanning for Special Values

After dealing with missing values, we will check if there is any special value, i.e. -Inf, Inf and NaN. We can use is.finite, is.infinite, or is.nan functions to identify the special values in a data set. As we are checking the entire data frame, we need to use apply family functions. We use sapply function here. It can be applied to a list. As data frames possess the characteristics of both lists and matrices, sapply can be applied to data frames.

We create a new function to check for the sum of infinite or NaN or NA values for numerical column, and the sum of NA values for other columns. We write the function inside sapply() and calculate the total missing values for each column.

Hide

```
# Check every numerical column whether they have infinite or NaN or NA values using a
function
sapply(songs_selected, function(x) {
  if(is.numeric(x)) {sum(is.infinite(x) | is.nan(x) | is.na(x))}
  else {sum(is.na(x))}
    })
```

```
  popularity   track_name track_artist         mode        genre duration_min 
           0            0            0            0            0            0 
```

From the result, there is no missing value or special value.

Scanning for Obvious Error

Next, we will check if there is any obvious inconsistency that may not correspond to a real-world situation. For example, a song duration can not be negative.

We will define a restriction on the duration_min variable using editset functions from editrules package.

Hide

```
(Rule1 <- editset("duration_min > 0"))
```

```


Edit set:
num1 : 0 < duration_min
```

The editset function parses the textual rules and stores them in an editset object. The data set can be checked against these rules using the violatedEdits function. If there is data which violates the rules, it will return a logical array of “TRUE”. Here we will check the total sum.

Hide

```
violatedEdits(Rule1, songs_selected) %>% sum()
```

```
[1] 0
```

The result “0” means that there is no data violating the rule, hence returning all “FALSE” and therefore no “TRUE” is returned.

Scan II

As missing values, special values, and obvious errors have been defined and managed, we will now scan for the outliers.

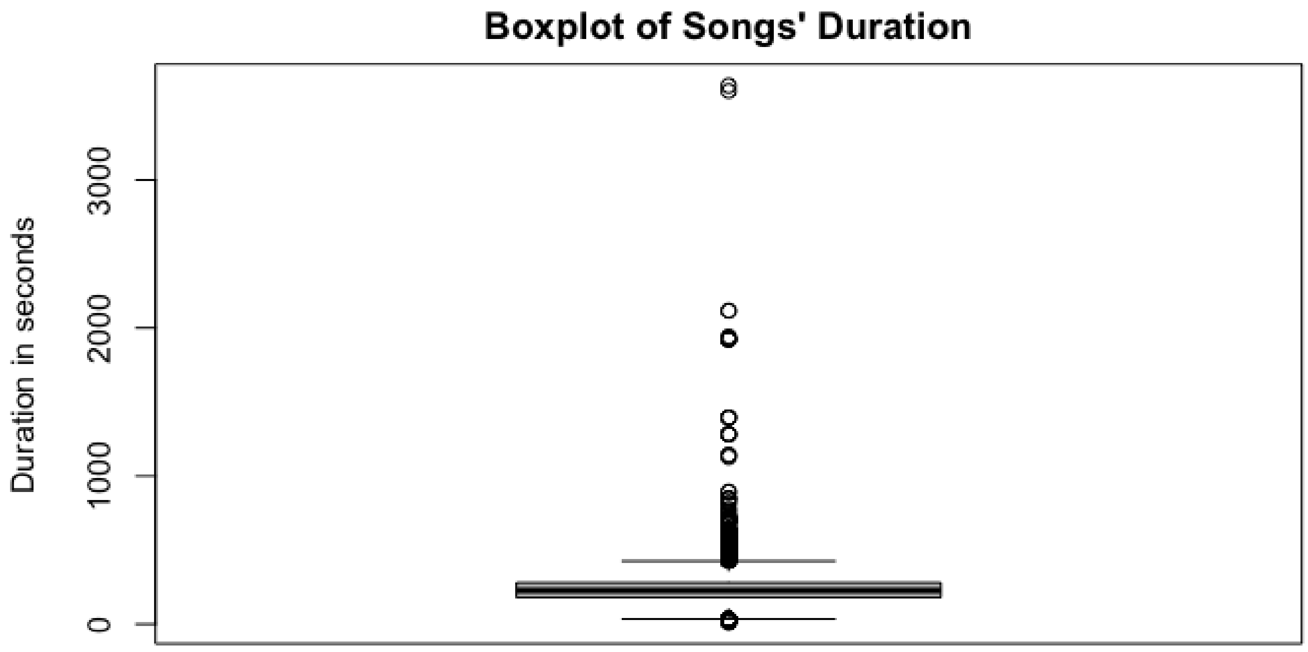
Using boxplot() function

According to Tukey's method of outlier detection, outliers are the values in the data set that are out of the outlier fences, which is a pair of limitations calculated in between −1.5×IQR to 1.5×IQR range of the boxplot.

Given that, we begin with using boxplot() function to get the boxplot of the duration_min variable, which has previously been applied with period_to_seconds() function to approximately convert the variable into seconds.

Hide

```
songs_selected$duration_min %>% period_to_seconds() %>% boxplot(main="Boxplot of Song
s' Duration", ylab="Duration in seconds", col = "grey")
```



Hide

```
#According to the Tukey's method, the duration variable seems to have many outliers
```

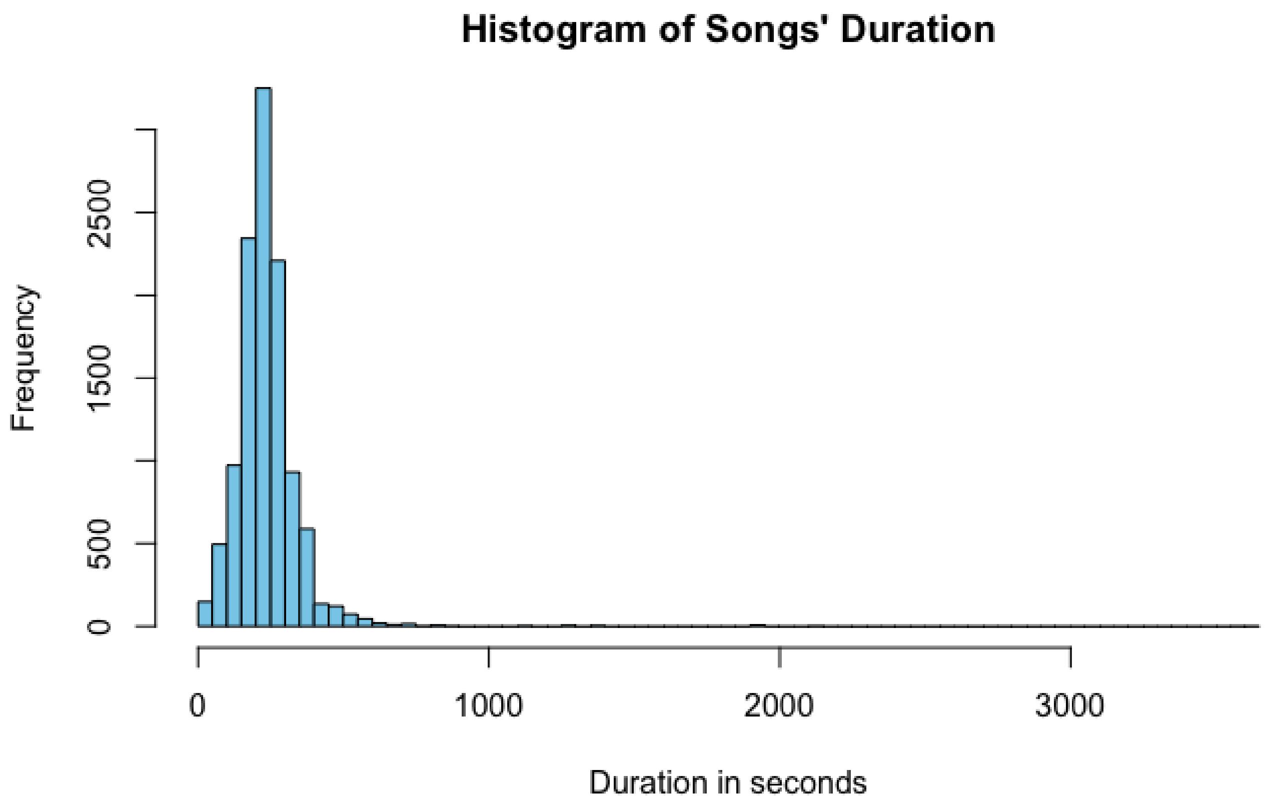
According to the Tukey's method, the duration variable seems to have many outliers.

Using z-score

In order to find how many outliers there are, we will be looking at the z-score using score() function from outliers package. First, we will have to ensure if our distribution is approximately normal. For this, we apply hist() function to see the histogram.

Hide

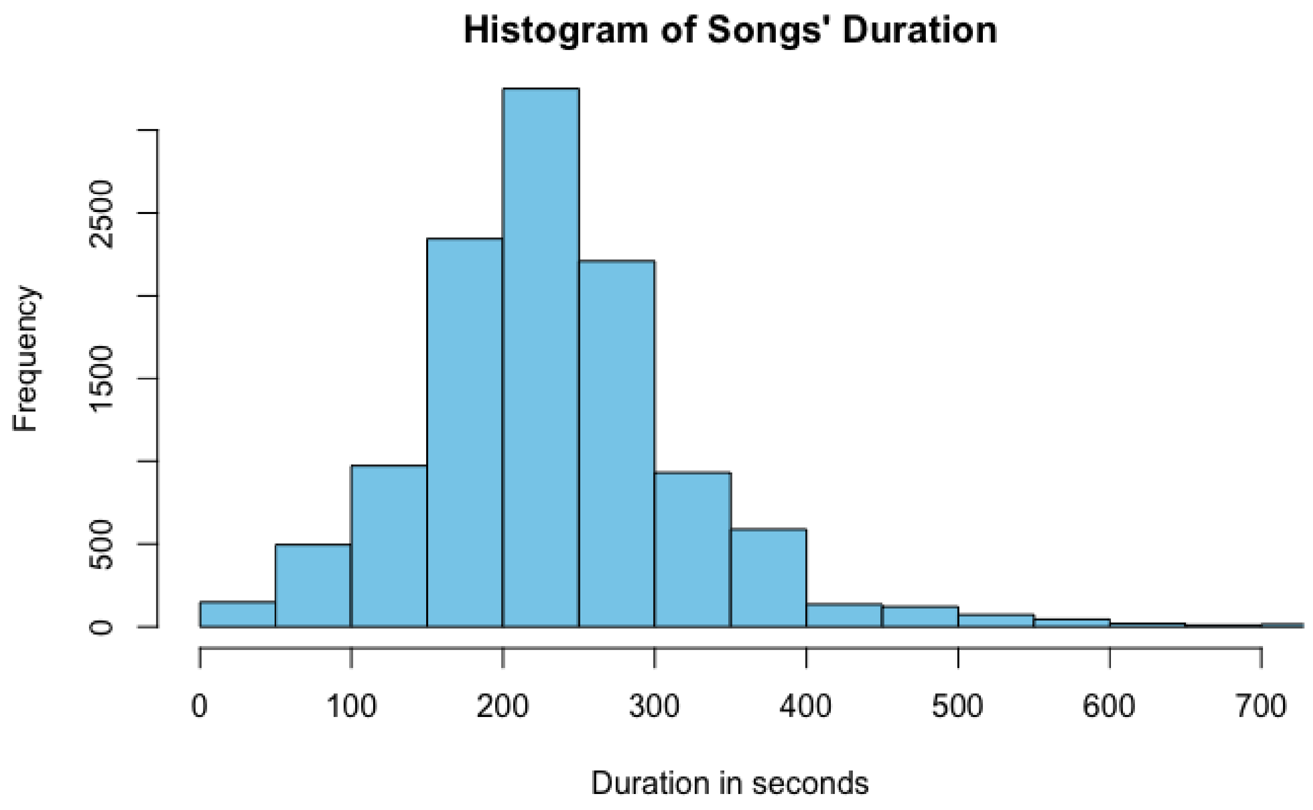
```
songs_selected$duration_min %>% period_to_seconds() %>% hist(main="Histogram of Song
s' Duration", xlab="Duration in seconds", breaks=100, col="skyblue")
```



As strange as it looks, the histogram does not appear normal at first glance. We then realize that the amount from approximately 700 - 1,000 seconds upwards in the x-axis are competitively low compared to the rest on the left-hand side. With the insignificant amount, which can possibly be less than five per cent (5%) assuming from the overall distribution, we then try to set the limit to see the majority of the distribution.

Hide

```
songs_selected$duration_min %>% period_to_seconds() %>% hist(main="Histogram of Song
s' Duration", xlab="Duration in seconds", xlim=c(0,700), breaks=100, col="skyblue")
```



With the requirement of data being normally distributed has been fulfilled, we can now calculate z-score for our data using score() function.

Hide

```
z.scores <- songs_selected$duration_min %>% period_to_seconds() %>% scores(type = "z"
)
z.scores %>% summary()
```

```
    Min.  1st Qu.   Median     Mean  3rd Qu.     Max. 
-1.88659 -0.47167 -0.08654  0.00000  0.34882 28.46304 
```

Using summary() function, we find that the minimum and maximum z-score are -1.88418 and 28.46530 respectively. Lastly, we apply which() function to see the locations of absolute z-score values that are greater than three and count how many they are using length() function.

Hide

```
which(abs(z.scores) >3 )
```

```
 [1]    25   467   724  1788  1789  2334  3025  3026  3027  3609  4098  5172  5736  6
457  6458  6459  6460  6461  6462
[20]  6463  6464  6618  6624  6671  6696  6858  6961  7140  7141  7142  7143  7144  7
145  7146  7845  7930  7977  8003
[39]  8015  8533  9035  9488  9489  9490  9491  9867  9868  9869  9870 10096 10097 10
098 10099 10100 10101 10102 10206
[58] 10207 10208 10209 10476 10477 10478 10479 10480 10802 10803 10804 10811 10812 10
813 10814 10815 10816 10817 10833
[77] 10834 10835 10836 10837 11215 11314
```

Hide

```
length (which(abs(z.scores) >3 ))
```

```
[1] 82
```

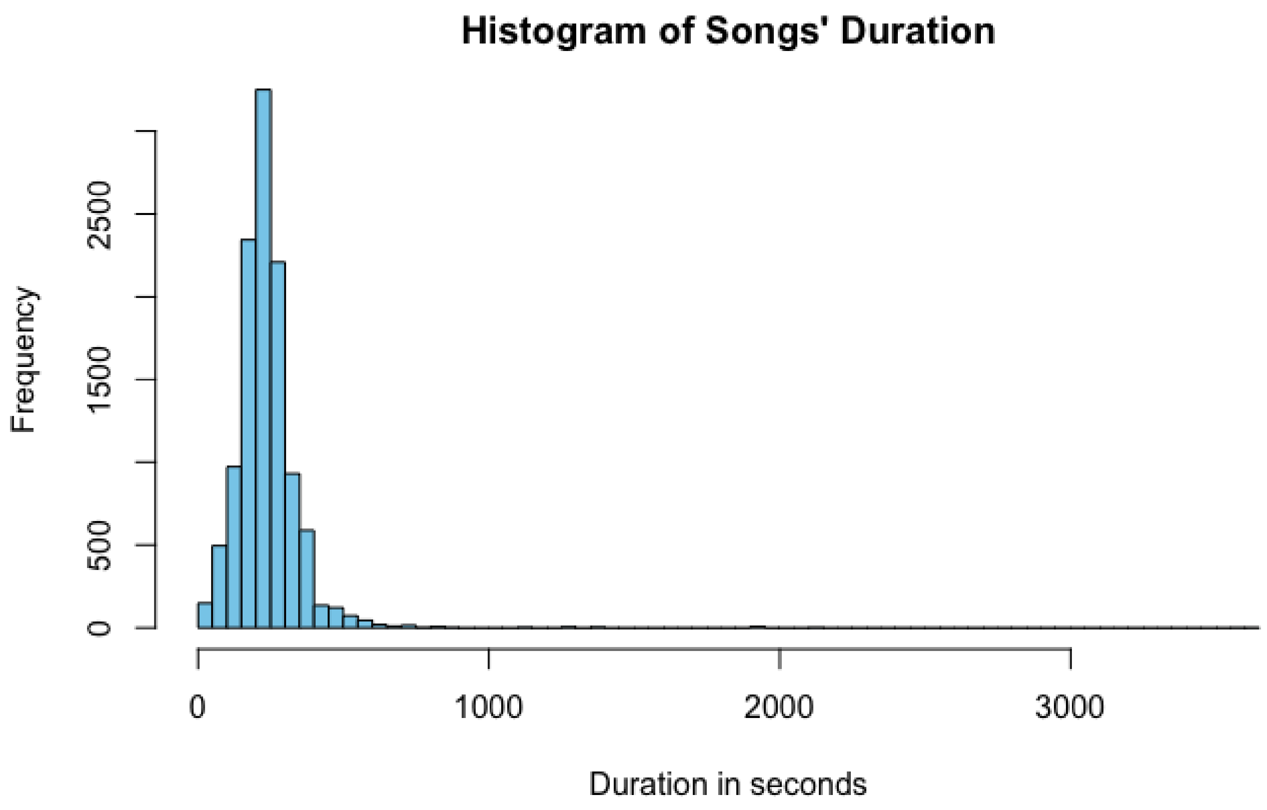
Transform

From the histogram we achieved previously, we can also apply data transformation using mathematical operations to adjust the illustration of the distribution.

First, the histogram for duration_min is totally right-skewed.

Hide

```
songs_selected$duration_min %>% period_to_seconds() %>%
  hist(main="Histogram of Songs' Duration", xlab="Duration in seconds", breaks=100, c
ol="skyblue")
```

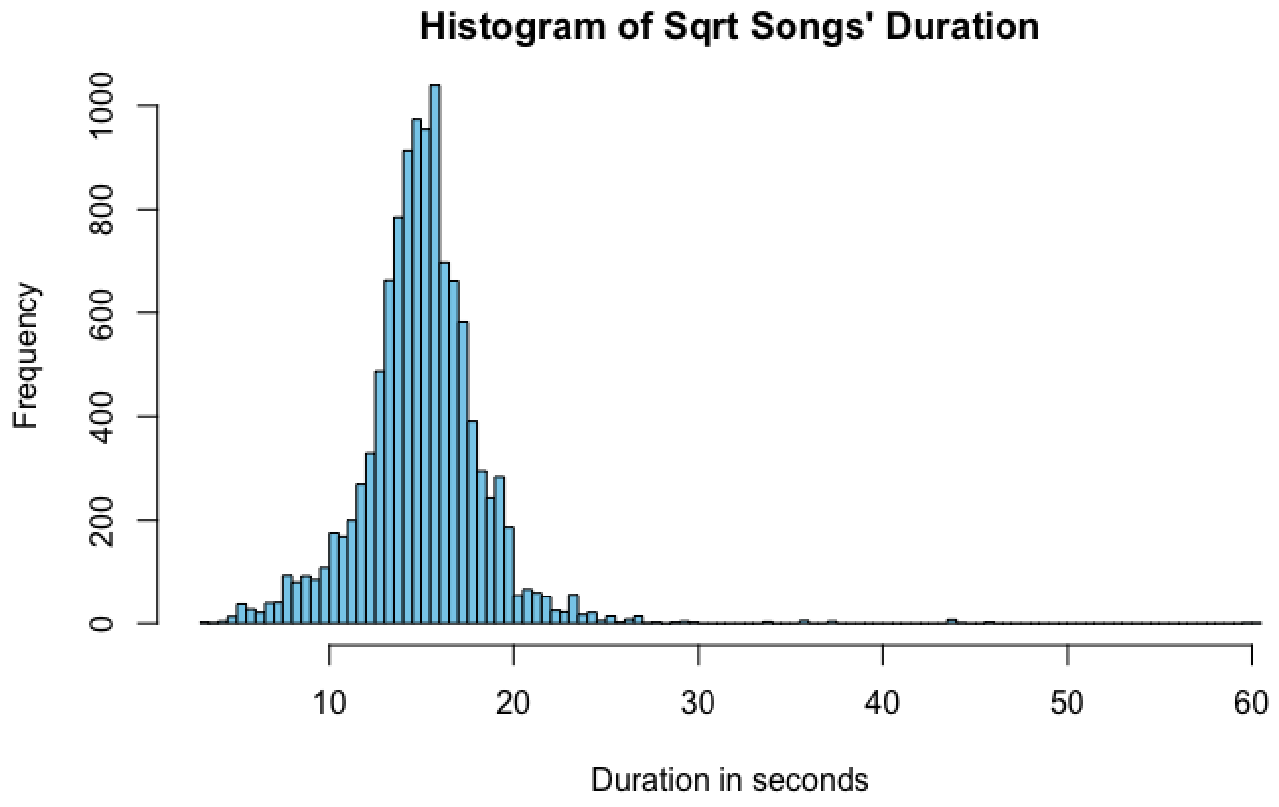


Using sqrt() and log10() functions

Because of the right skewness, we apply sqrt() function that helps reducing right skewness or log10() function to compresses high values and spreads low values by expressing the values as orders of magnitude (Box, George EP, and David R Cox., 1964)

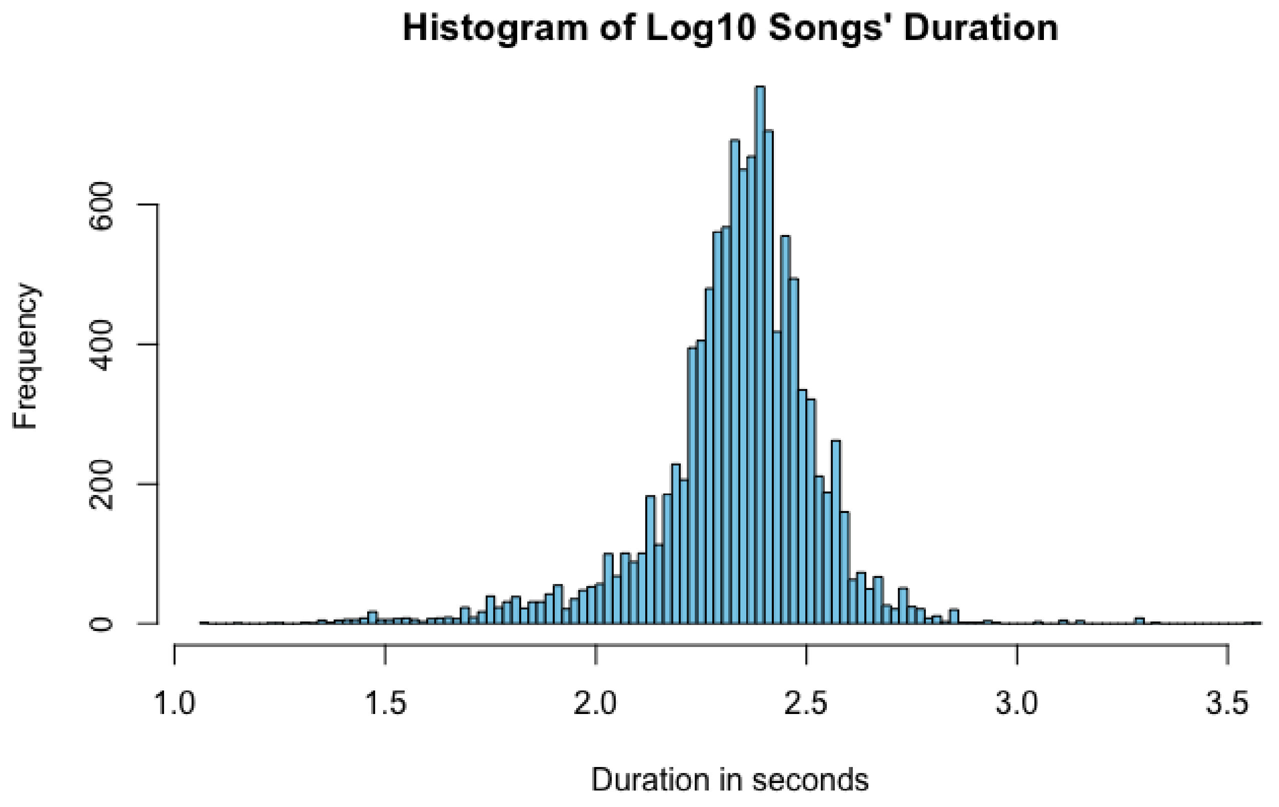
Hide

```
songs_selected$duration_min %>% period_to_seconds() %>% sqrt() %>%
   hist(main="Histogram of Sqrt Songs' Duration", xlab="Duration in seconds", breaks=
100, col="skyblue")
```



Hide

```
songs_selected$duration_min %>% period_to_seconds() %>% log10() %>%
   hist(main="Histogram of Log10 Songs' Duration", xlab="Duration in seconds", breaks
=100, col="skyblue")
```



Alternatively, we can also apply BoxCox transformation using BoxCox() function from forecast package.

Hide

```
Boxcox_duration <- songs_selected$duration_min %>%
  period_to_seconds() %>%
  BoxCox(lambda = "auto")

head(Boxcox_duration, n = 30)
```

```
 [1] 6.355082 6.831987 6.486651 5.750706 6.355082 6.355082 6.355082 6.355082 6.355082
6.355082 6.355082 6.355082 6.355082
[14] 3.546303 3.683906 3.683906 3.683906 3.683906 3.683906 3.683906 6.291134 6.291134
6.291134 6.291134 7.806984 6.542772
[27] 6.542772 6.542772 6.542772 6.304149
```

Hide

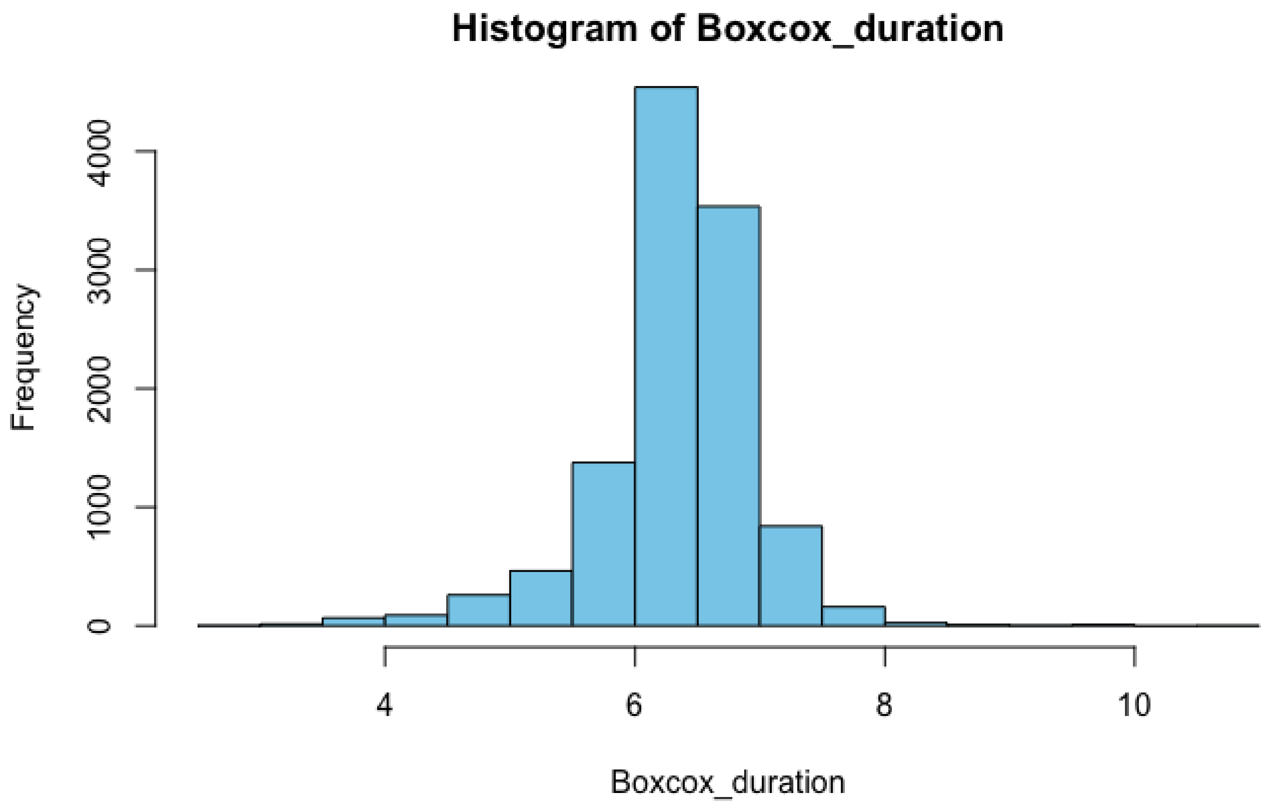
```
# Showing the optimum lambda value
attr(Boxcox_duration,"lambda")
```

```
[1] 0.05924957
```

The values returned from the function are transformed with the best parameter as we set lambda = “auto” and the optimum lambda value is found as 0.05941613. We can also see the distribution of transformed values using hist() function.

Hide

```
# Histogram of Boxcox
hist(Boxcox_duration, col="skyblue")
```



References

Box, George EP, and David R Cox. 1964. “An Analysis of Transformations.” Journal of the Royal Statistical Society. Series B (Methodological), 211–52.

Wickham, Hadley, and Garrett Grolemund. 2016. R for Data Science: Import, Tidy, Transform, Visualize, and Model Data. ” O’Reilly Media, Inc.”.

EstienneGGX, dambs0ap, & Nikita Sharma. (2022, September 4). Spotify unpopular songs (unpopular_songs.csv). Kaggle. Retrieved October 7, 2022, from https://www.kaggle.com/datasets/estienneggx/spotify-unpopular-songs?select=unpopular_songs.csv (https://www.kaggle.com/datasets/estienneggx/spotify-unpopular-songs?select=unpopular_songs.csv)

EstienneGGX, dambs0ap, & Nikita Sharma. (2022, September 4). Spotify unpopular songs (z_genre_of_artists.csv). Kaggle. Retrieved October 7, 2022, from https://www.kaggle.com/datasets/estienneggx/spotify-unpopular-songs?select=z_genre_of_artists.csv (https://www.kaggle.com/datasets/estienneggx/spotify-unpopular-songs?select=z_genre_of_artists.csv)