Data Wrangling Assessment Task 3: Dataset challenge

Timm Rahrt [redacted]

0. Setup

This Assessment deals with five different datasets of the website `data.ct.gov` *(Ct Data, 2024)*. The primary focus is to answer how to ensure all datasets, with varying structures and format, are transformed into a unified, analysable format, following the Data Wrangling Preprocessing steps.

We will achieve this by:

1. Webscraping and merging multiple datasets from `data.ct.gov`
2. Checking and applying Data Tidy Principles by *Wickham and Grolemund (2016)*
3. Handling missing, `Null` and misleading values
4. Standardising and mutating new variables to enhance the analysis
5. Visualizing and transforming the data

Before we start, we load all necessary libraries:

```
# Un-commend if needed
# install.packages('RSocrata')
# Please load packages required for producing this report
library(tidyverse) # Covers ggplot2, readr, tidyr, dplyr, etc
library(magrittr) # For piping
library(RSocrata) # For webscraping
library(lubridate) # For Date and Time conversion
library(infotheo) # For discretisation techniques, here equal-depth binning
# Seed for reproducibility
set.seed(123)
```

1. Data Description

1.1 Data Description Set 1

The first dataset, accessed via an API using `read.socrata` , focuses on commercial information of different towns in Connecticut *(GitHub, 2024)*. Since the original dataset contains over 1 Million Rows, we apply an `slice_sample` on the complete dataset to extract 10000 randomly selected rows and store the result in `rows_df` *(Dplyr.tidyverse.org, 2024)*. We group by the variable `town` and ensure that only the first occurrence of each town is kept by using `slice(1)` . We `ungroup` this dataset to allow further manipulations to be applied to the entire dataset. Next, we rearrange the variable order by calling `select` and listing `town` before all (`everything`) other variables. Finally, we apply `-c()` with all irrelevant variable names as an `attribute` , to `drop` /delete them.

Our finalised first dataset `sales_data` contains the variables:

- town: Name of the Connecticut town the property is located
- listyear: Year of the properties listing date
- daterecorded: Date when the sale was recorded
- assessedvalue: Assessed Value of the property
- saleamount: Price at which the property was sold
- salesratio: Ratio that compares sale price to assessed value
- propertytype: Type of the Property
- residentialtype: Type of the Residence

```
url <- "https://data.ct.gov/resource/5mzw-sjtu.json"
complete_data <- read.socrata(url)
# Select random rows once
rows_df <- complete_data %>% slice_sample(n = 10000)
# Ensure town appears just once & drop unnecessary variables
sales_data <- rows_df %>%
  group_by(town) %>% slice(1) %>% ungroup() %>%
  select(town, everything(), -c(serialnumber,nonusecode,geo_coordinates.type,geo_co
ordinates.coordinates,remarks,address,opm_remarks))
# Display the first rows
head(sales_data,5)
```

town <chr>	listyear <chr>	daterecorded <dttm>	assessedvalue <chr>	saleamount <chr>	salesratio <chr>	propertyty <chr>
Andover	2012	2013-03-04	49000	59000	0.830508475	NA
Ansonia	2001	2002-07-31	72870	50000	1.4574	NA
Ashford	2005	2006-05-22	116960	234450	0.498869695	NA
Avon	2001	2002-07-01	315890	600000	0.526483333	NA
Barkhamsted	2019	2020-08-03	153030	259000	0.5908	Single Fam

5 rows | 1-7 of 8 columns

1.2 Data Description Set 2

The next dataset contains information on *housing in various towns*. We apply the same webscraping technique as before using the `RSocrata` package and also rearrange the variables order, so that `town` comes first in our newly created dataset `affordable_housing`. Again, we group by `town` and only consider the first available entry using `slice(1)`. `-c()` is used like before, to drop all irrelevant variables.

The finalised dataset `affordable_housing` contains the following variables:

- town: Name of the town
- X_2010_census: Population of the town 2010
- gov_assisted: Number of government-assisted housing units

- tenant_rental_assistance: Rental units receiving rental assistance
- chfa_usda_mortgages: Housing units financed through CHFA (Connecticut Housing Finance Authority) or USDA (United States Department of Agriculture) mortgages
- total_assisted: Total number of assisted housing units in the town.

```
# Import dataset 2, webscrap the dataset using RSocrata
url_2 <- "https://data.ct.gov/resource/3udy-56vi.json"
affordable_housing <- read.socrata(url_2)
# Adjust the dataset and display the first 5 rows
affordable_housing %<>% group_by(town) %>% slice(1) %>% ungroup() %>%
  select(town, everything(), -c(year,code,percent_assisted,deed_restricted))
head(affordable_housing,5)
```

town <chr>	X_2010_census <chr>	gov_assisted <chr>	tenant_rental_assistance <chr>	chfa_usda_mortg <chr>
Andover	1317	18	1	32
Ansonia	8148	349	764	147
Ashford	1903	32	0	36
Avon	7389	244	16	44
Barkhamsted	1589	0	6	23

5 rows

1.3 Data Description Set 3

After the import of the dataset, we save it as `housing_segregation` and `rename` the variable `sub_geography` to `town` as the values are the same as the previous `town` values of the other datasets. We will use `select` for the rearrangement of variables and delete the first twelve rows using `slice(c(1:12))` due to irrelevant entries *(Rdocumentation.org, 2018)*. We sort the dataframe by `town` in ascending order using `arrange` and display the first five observations using `head` on the dataframe. `housing_segragation` displays inequality in the years 2010, 2015 and 2020 across various towns, represented by the Gini Index. An index of `0` equals to perfect equality and `1` to maximal inequality.

- town: Represents the town
- gini_index_2010: The Gini Index for the year 2010
- gini_index_2015: The Gini Index for the year 2015
- gini_index_2020: The Gini Index for the year 2020

```
# Load the third dataset
url_3 <- "https://data.ct.gov/resource/5e73-kfqf.json"
housing_segregation <- read.socrata(url_3)
# Get rid of the first 12 double up rows, rearrange variables and sort by town
housing_segregation %<>%
  rename(town = sub_geography) %>%
  select(town, everything(),-geography) %>% slice(-c(1:12)) %>%  arrange(town)
# Display the first 5 rows
head(housing_segregation,5)
```

	town <chr>	gini_index_2010 <chr>	gini_index_2015 <chr>	gini_index_2020 <chr>
1	Andover	0.394	0.3547	0.3557
2	Ansonia	0.378	0.462	0.4362
3	Ashford	0.347	0.4184	0.4315
4	Avon	0.46	0.4682	0.5141
5	Barkhamsted	0.308	0.3488	0.3479

5 rows

1.4 Data Description Set 4

Our fourth dataset `housing_permits` contains information about the number of housing permits issued in various towns across different years. We need to `rename` towns to `town` , to guarantee equality in the variable name.

This dataset contains various columns like:

- `town` : Name of the town
- `X_1990` to `X_2022` : Number of housing permits issued in the year
- `county` : Name of the county where the town is located

We can see below that this dataset is in wide-format and, without further analysis, violates the Tidy Principle 1. and 2., as values form columns and not every row is one observation.

```
# Load the fourth dataset as per previous technique
url_4 <- "https://data.ct.gov/resource/stm9-38x4.json"
housing_permits <- read.socrata(url_4)
# Rearrange and drop observations
housing_permits %<>% slice(-1) %>% rename(town = towns) %>% arrange(town)
head(housing_permits,4)
```

town <chr>	X_1990 <chr>	X_1991 <chr>	X_1992 <chr>	X_1993 <chr>	X_1994 <chr>	X_1995 <chr>	X_1996 <chr>	X_1997 <chr>

1	Andover	20	16	18	8	10	22	36	26
2	Ansonia	15	19	46	49	34	27	23	16
3	Ashford	24	19	32	52	12	18	14	16
4	Avon	30	39	50	48	60	66	83	144

4 rows | 1-10 of 36 columns

To guarantee this dataset is compatible with the other datasets, we must perform some *tidying of the dataset*. `pivot_longer` changes the format from a wide to long format, where all columns starting with "X_" will be selected `cols = starts_with()` and moved to a new variable called year (`names_to = "year"`) *(R-Packages, 2024)*. `names_prefix` removes *"X_"* and finally, `values_to` moves all values to a new variable named `permits` .

```
# Change the datasets wide format to long format
housing_permits %<>%
  pivot_longer(cols = starts_with("X_"),
               names_to = "year",
               names_prefix = "X_",
               values_to = "permits")
head(housing_permits,3)
```

town <chr>	county <chr>	year <chr>	permits <chr>
Andover	Tolland	1990	20
Andover	Tolland	1991	16
Andover	Tolland	1992	18

3 rows

After successfully transforming the data into long format, we must ensure each town also just represents every `town` once. The below code groups the towns first, county second and performs a `summarise` function on them, which creates two new variables. `total_permits` represents the numerical sum `sum(as.numeric())` of all permits while disregarding all missing values (`na.rm=TRUE`). `peak_year` displays each towns year where the maximum permits were issued, using `which.max()` . We `ungroup` our dataset and `arrange` the `town` in ascending order.

```
# Summarise and create new variables to dispaly unique town's
housing_permits %<>%
  group_by(town, county) %>%
  summarise(total_permits = sum(as.numeric(permits), na.rm = TRUE),
            peak_year = year[which.max(permits)]) %>% ungroup() %>% arrange(town)
```

```
## `summarise()` has grouped output by 'town'. You can override using the
## `.groups` argument.
```

```
head(housing_permits,4)
```

town	county	total_permits	peak_year
<chr>	<chr>	<dbl>	<chr>
Andover	Tolland	357	1996
Ansonia	New Haven	495	1993
Ashford	Windham	488	1993
Avon	Hartford	2046	1998

4 rows

1.5 Data Description Set 5

The last dataset this reports includes contains information about various types of property net values in different towns. The variables are:

- town_name: Name of the town
- residential_net: Net value of residential properties
- apartments_net: Net value of apartment properties
- cip_net: Net value of commercial, industrial, and public properties
- Net value of vacant properties
- total_real_property

It will also be webscraped using `RSocrata`. We save the data with relevant variables in a temporary variable `temp_df` and display the first two observations.

```
# Webscrape data using RSocrata as before and select relevant variables
url_5 <- "https://data.ct.gov/resource/8rr8-a322.json"
temp_df <- read.socrata(url_5)
temp_df <- temp_df[,c(2,4,7,10,13,22)]
head(temp_df,2)
```

	town_na...	residential_net	apartments_net	cip_net	vacant_net	total_real_property
	<chr>	<chr>	<chr>	<chr>	<chr>	<chr>
1	Andover	213711188	1536000	6381500	6133800	228398608
2	Ansonia	903322126	16936400	124242058	0	1044624879

2 rows

As we can see, we are experiencing a similar problem as before, the data is in wide format representing values as variables. To satisfy the `Data Tidying Principles`, we must again transform the dataset into long format using `pivot_longer`. We select all column names and store them in a new variable `type_property` using `names_to`. Like before, we store all values using `values_to` in a newly created variable `property_values`.

```
# Change the datasets format from wide to long
values_property <- temp_df %>%
  pivot_longer(c(residential_net,apartments_net,cip_net,vacant_net,total_real_prope
rty),
               names_to = "type_property",
               values_to = "property_value")
head(values_property, 5)
```

town_name <chr>	**type_property** <chr>	**property_value** <chr>
Andover	residential_net	213711188
Andover	apartments_net	1536000
Andover	cip_net	6381500
Andover	vacant_net	6133800
Andover	total_real_property	228398608

5 rows

As we end up being in the same situation as before, we `summarise` the `mean` of the numeric `property_value`, disregarding missing values, by `town_name`. We rename the variable `town_name` to `town` and display the newly created dataset `avg_property_value`.

```
# Create a new variable `avg_property` to display every town just once
avg_property_value <- values_property %>%
  group_by(town_name) %>%
  summarise(avg_property = mean(as.numeric(property_value), na.rm = TRUE)) %>% ungr
oup() %>% rename(town = town_name)
head(avg_property_value,5)
```

town <chr>	**avg_property** <dbl>
Andover	93440436
Ansonia	340374620
Ashford	106367059
Avon	934401976

Barkhamsted	117875301

5 rows

After the creation of all dataset, we will now merge all datasets to one big dataframe. As the following `left_join` merges two datasets on a common variable, we must identify this variable using the `intersect` function on all five individual datasets *(dplyr.tidyverse.org, 2024)*.

```
# Identify the common variable of all datasets
common_var <- intersect(sales_data %>% names(), affordable_housing %>% names())
common_var <- intersect(common_var, housing_segregation %>% names())
common_var <- intersect(common_var, housing_permits %>% names())
common_var <- intersect(common_var, avg_property_value %>% names())
common_var
```

```
## [1] "town"
```

We can merge all datasets on the variable `town` .

2. Understand

2.1 Merging the datasets

After webscraping and organising the datasets, we will merge them all together. Please note, the data is still untidy and we must also ensure that every variable is represented in their correct type. We join all four datasets together, called `connecticut_df` , using the `left_join` and the previously identified common variable `town` .

```
# Join all datasets to one together
connecticut_df <- sales_data %>%
  left_join(affordable_housing, by = 'town') %>%
  left_join(housing_segregation, by = 'town') %>%
  left_join(housing_permits, by = 'town') %>%
  left_join(avg_property_value, by = 'town')
head(connecticut_df)
```

town <chr>	listyear <chr>	daterecorded <dttm>	assessedvalue <chr>	saleamount <chr>	salesratio <chr>	propertyt <chr>
Andover	2012	2013-03-04	49000	59000	0.830508475	NA
Ansonia	2001	2002-07-31	72870	50000	1.4574	NA
Ashford	2005	2006-05-22	116960	234450	0.498869695	NA
Avon	2001	2002-07-01	315890	600000	0.526483333	NA
Barkhamsted	2019	2020-08-03	153030	259000	0.5908	Single Fa

Beacon Falls	2020	2020-12-22	241220	385000	0.6265	Residenti

6 rows | 1-7 of 20 columns

When inspecting our `connecticut_df`, it becomes obvious that the variables need clearer names. We display all names and their type using the `sapply` function, to gain an overview of their current names and types *(www.rdocumentation.org, 2024)*.

```
# Display just variable names and type
sapply(connecticut_df[0,],typeof)
```

```
##                      town                  listyear              daterecorded
##               "character"               "character"                  "double"
##             assessedvalue                saleamount                salesratio
##               "character"               "character"               "character"
##              propertytype           residentialtype             X_2010_census
##               "character"               "character"               "character"
##              gov_assisted tenant_rental_assistance        chfa_usda_mortgages
##               "character"               "character"               "character"
##            total_assisted            gini_index_2010           gini_index_2015
##               "character"               "character"               "character"
##           gini_index_2020                    county             total_permits
##               "character"               "character"                  "double"
##                 peak_year              avg_property
##               "character"                  "double"
```

The below code will change all variable names to a better, easier understandable version, saved in `new_var_names`. We save the old ones into `old_var_names` using `names`.

```
new_var_names <- c("Towns","Listed Year","Recorded Date","Assessed Value","Sale Amo
unt","Sale Ratio","Property Type","Residential Type","Census 2010","Gov. Assiste
d","Rental Assistance","Financed Units","Total Assisted","Gini 2010","Gini 2015","G
ini 2020", "County","Total Permits","Peak Permit Year","Average Value")
old_var_names <- names(connecticut_df)
rename_ <- setNames(old_var_names,new_var_names)
connecticut_df <- rename(connecticut_df, !!!rename_)
colnames(connecticut_df)
```

```
##  [1] "Towns"             "Listed Year"       "Recorded Date"
##  [4] "Assessed Value"    "Sale Amount"       "Sale Ratio"
##  [7] "Property Type"     "Residential Type"  "Census 2010"
## [10] "Gov. Assisted"     "Rental Assistance" "Financed Units"
## [13] "Total Assisted"    "Gini 2010"         "Gini 2015"
## [16] "Gini 2020"         "County"            "Total Permits"
## [19] "Peak Permit Year"  "Average Value"
```

We are using `!!!` in `rename` to pass the named vector as individual arguments to `rename` *(R-lib.org, 2024)*. As the name are better understandable now, we need to convert the variables in their correct format. We save all numeric variable of `connecticut_df` and call `vapply` method on them, which will iterate through every one of them applying `as.numeric`. `numeric(nrow())` ensures the output of this apply function are numerical variables. We have a look at the result using `glimpse`.

```
# Type conversion numeric data
numeric_var <- c("Assessed Value","Sale Amount","Sale Ratio","Census 2010","Gov. Assisted","Rental Assistance","Financed Units","Total Assisted","Gini 2010","Gini 2015","Gini 2020")
connecticut_df[numeric_var] <- vapply(connecticut_df[numeric_var], as.numeric, numeric(nrow(connecticut_df)))
glimpse(connecticut_df)
```

```
## Rows: 169
## Columns: 20
## $ Towns                 <chr> "Andover", "Ansonia", "Ashford", "Avon", "Barkhams…
## $ `Listed Year`         <chr> "2012", "2001", "2005", "2001", "2019", "2020", "2…
## $ `Recorded Date`       <dttm> 2013-03-04, 2002-07-31, 2006-05-22, 2002-07-01, 2…
## $ `Assessed Value`      <dbl> 49000, 72870, 116960, 315890, 153030, 241220, 6450…
## $ `Sale Amount`         <dbl> 59000, 50000, 234450, 600000, 259000, 385000, 1120…
## $ `Sale Ratio`          <dbl> 0.8305085, 1.4574000, 0.4988697, 0.5264833, 0.5908…
## $ `Property Type`       <chr> NA, NA, NA, NA, "Single Family", "Residential", "C…
## $ `Residential Type`    <chr> NA, NA, NA, NA, "Single Family", "Single Family", …
## $ `Census 2010`         <dbl> 1317, 8148, 1903, 7389, 1589, 2509, 8140, 2044, 73…
## $ `Gov. Assisted`       <dbl> 18, 349, 32, 244, 0, 0, 556, 0, 192, 24, 558, 0, 0…
## $ `Rental Assistance`   <dbl> 1, 764, 0, 16, 6, 4, 50, 2, 26, 0, 106, 2, 3, 77, …
## $ `Financed Units`      <dbl> 32, 147, 36, 44, 23, 46, 142, 13, 154, 9, 341, 28,…
## $ `Total Assisted`      <dbl> 51, 1260, 68, 304, 29, 50, 752, 15, 459, 33, 1005,…
## $ `Gini 2010`           <dbl> 0.394, 0.378, 0.347, 0.460, 0.308, 0.345, 0.387, 0…
## $ `Gini 2015`           <dbl> 0.3547, 0.4620, 0.4184, 0.4682, 0.3488, 0.3540, 0.…
## $ `Gini 2020`           <dbl> 0.3557, 0.4362, 0.4315, 0.5141, 0.3479, 0.4075, 0.…
## $ County                <chr> "Tolland", "New Haven", "Windham", "Hartford", "Li…
## $ `Total Permits`       <dbl> 357, 495, 488, 2046, 330, 779, 2573, 545, 1776, 30…
## $ `Peak Permit Year`    <chr> "1996", "1993", "1993", "1998", "1998", "2005", "2…
## $ `Average Value`       <dbl> 93440436, 340374620, 106367059, 934401976, 1178753…
```

We can see, that more non-numerical variables need to be converted, such as `Recorded Date`. We convert he variable to a date format using the `lubridate` package *(Package 'lubridate' Type Package Title Make Dealing with Dates a Little Easier, 2020)*. `case_when` checks if the data can be converted using a specific format, if not, it applies an alternative format. `TRUE ~` ensures that the remaining cases, which didn't satisfy the conditions, are formatted as defined after the tilde `~ as.Date())`.

```
# Date conversion
connecticut_df %<>% mutate(`Recorded Date` = case_when(is.na(as.Date(`Recorded Date`, format = "%d %b %Y")) ~ as.Date(`Recorded Date`, format = "%d-%b-%Y"), TRUE ~ as.Date(`Recorded Date`, format = "%d %b %Y")))
head(connecticut_df)
```

Towns <chr>	Listed Year <chr>	Recorded Date <date>	Assessed Value <dbl>	Sale Amount <dbl>	Sale Ratio <dbl>	Pr <c
Andover	2012	2013-03-03	49000	59000	0.8305085	NA
Ansonia	2001	2002-07-30	72870	50000	1.4574000	NA
Ashford	2005	2006-05-21	116960	234450	0.4988697	NA
Avon	2001	2002-06-30	315890	600000	0.5264833	NA
Barkhamsted	2019	2020-08-02	153030	259000	0.5908000	Si
Beacon Falls	2020	2020-12-21	241220	385000	0.6265000	Re

6 rows | 1-7 of 20 columns

The last variables we need to adjust are `Proptery Type` and `Residential Type`, as they should be formatted as `factor` variables. We check their `unique` values to ensure the values are ideal for the factor conversion.

```
# Label and level `Property Type` and `Residential Type`
unique(connecticut_df$`Property Type`)
```

```
## [1] NA               "Single Family" "Residential"   "Condo"
## [5] "Vacant Land"   "Three Family"  "Commercial"    "Two Family"
```

```
unique(connecticut_df$`Residential Type`)
```

```
## [1] NA               "Single Family" "Condo"         "Three Family"
## [5] "Two Family"
```

As the variables output a countable amount of unique values, we proceed with the factor conversion. The below code also relabels the levels of each variable to an easier understanding value using `labels`. Finally, we display the two newly created factors using the `select` function.

```
connecticut_df %<>%
  mutate(`Property Type` = factor(`Property Type`,
                                  levels = c("Single Family","Three Family","Cond
o","Residential","Commercial","Vacant Land"),
                                  labels = c("Single Family Property","Three-Family
Property","Condo","Residential Property","Commercial Property","Vacant Land")))

connecticut_df %<>%
  mutate(`Residential Type` = factor(`Residential Type`,
                                  levels = c("Single Family","Condo","Two Family","
Three Family"),
                                  labels = c("Single Household","Condo","Two-Family
Home","Three-Family Home")))
connecticut_df %>%
  select(Towns,`Property Type`,`Residential Type`) %>% head()
```

Towns <chr>	Property Type <fct>	Residential Type <fct>
Andover	NA	NA
Ansonia	NA	NA
Ashford	NA	NA
Avon	NA	NA
Barkhamsted	Single Family Property	Single Household
Beacon Falls	Residential Property	Single Household

6 rows

3.1 Tidy & Manipulate I

Our merged dataset `connecticut_df` should now contain several columns, which all represent a single variable, and rows, which display different observations filtered by the `Towns`. We can double check that our dataset follows the `Tidy Dataset Principles` of Wickham and Grolemund (2016), by inspecting the merged set calling the `str` function.

```
# We inspect the connecticut_df dataset
str(connecticut_df)
```

```
## tibble [169 × 20] (S3: tbl_df/tbl/data.frame)
##  $ Towns             : chr [1:169] "Andover" "Ansonia" "Ashford" "Avon" ...
##  $ Listed Year       : chr [1:169] "2012" "2001" "2005" "2001" ...
##  $ Recorded Date     : Date[1:169], format: "2013-03-03" "2002-07-30" ...
##  $ Assessed Value    : num [1:169] 49000 72870 116960 315890 153030 ...
##  $ Sale Amount       : num [1:169] 59000 50000 234450 600000 259000 ...
##  $ Sale Ratio        : num [1:169] 0.831 1.457 0.499 0.526 0.591 ...
##  $ Property Type     : Factor w/ 6 levels "Single Family Property",..: NA NA NA N
A 1 4 3 1 1 4 ...
##  $ Residential Type : Factor w/ 4 levels "Single Household",..: NA NA NA NA 1 1
2 1 1 1 ...
##  $ Census 2010       : num [1:169] 1317 8148 1903 7389 1589 ...
##  $ Gov. Assisted     : num [1:169] 18 349 32 244 0 0 556 0 192 24 ...
##  $ Rental Assistance: num [1:169] 1 764 0 16 6 4 50 2 26 0 ...
##  $ Financed Units    : num [1:169] 32 147 36 44 23 46 142 13 154 9 ...
##  $ Total Assisted    : num [1:169] 51 1260 68 304 29 50 752 15 459 33 ...
##  $ Gini 2010         : num [1:169] 0.394 0.378 0.347 0.46 0.308 0.345 0.387 0.379
0.375 0.382 ...
##  $ Gini 2015         : num [1:169] 0.355 0.462 0.418 0.468 0.349 ...
##  $ Gini 2020         : num [1:169] 0.356 0.436 0.431 0.514 0.348 ...
##  $ County            : chr [1:169] "Tolland" "New Haven" "Windham" "Hartford" ...
##  $ Total Permits     : num [1:169] 357 495 488 2046 330 ...
##  $ Peak Permit Year : chr [1:169] "1996" "1993" "1993" "1998" ...
##  $ Average Value     : num [1:169] 9.34e+07 3.40e+08 1.06e+08 9.34e+08 1.18e+08
...
```

Lastly, we check if we also fulfilled the third rule of tidy data: *"Each value must have its own cell."*. The below code uses the `vapply` to iterate `is.list` through our variables and their values, returning `TRUE` if any lists are indeed part of the dataset. `vapply` takes an extra argument `logical(1)` to ensure the output is `FALSE` and nothing else *(Rpubs.com)*.

```
# Iterate through all values for lists
standalone_val <- vapply(connecticut_df, is.list, logical(1))
list_val <- names(connecticut_df)[standalone_val]
list_val
```

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

As expected, the result is `character(0)` , which means no `TRUE` or *lists* were found in the values of `connecticut_df` . It is important to note that the dataset is still not *clean* yet, due to missing values and inconsistencies in the variables.

3.2 Tidy & Manipulate II

This section will mutate new variables to `connecticut_df` . We create w new variable `Average Gini` by calculating the mean of all variables starting with "Gini" using `rowMeans` and `starts_with` *(Rdocumentation.org, 2016)*. Once the new variable has been created, we drop the three individual `Gini` columns. We `round` the newly created variable by two digits after zero and display a few selected variables including `Average Gini`.

```
# This is a chunk where you create/mutate at least one variable from existing varia
bles
connecticut_df %<>%
  mutate(`Average Gini` = rowMeans(select(connecticut_df, starts_with("Gini")), na.
rm = TRUE)) %>%
  select(-c(`Gini 2010`,`Gini 2015`,`Gini 2020`))
connecticut_df$`Average Gini` <- round(connecticut_df$`Average Gini`, digits = 2)
connecticut_df %>%
  select(Towns,`Assessed Value`,`Total Assisted`,`Average Gini`,`Sale Ratio`) %>% h
ead(4)
```

Towns <chr>	**Assessed Value** <dbl>	**Total Assisted** <dbl>	**Average Gini** <dbl>	**Sale Ratio** <dbl>
Andover	49000	51	0.37	0.8305085
Ansonia	72870	1260	0.43	1.4574000
Ashford	116960	68	0.40	0.4988697
Avon	315890	304	0.48	0.5264833

4 rows

Next, we would like to include a new factor Category correlating to our `Sale Ratio` variable, displaying if a property is *Low Priced*, *Medium Priced* or *High Priced*. We display basic summary statistics of the variable using `summary`.

```
# Basic summary statistics on the `Sale Ratio` variable
summary(connecticut_df$`Sale Ratio`)
```

```
##     Min.  1st Qu.   Median     Mean  3rd Qu.     Max.
##  0.01211  0.49340  0.63068  0.94179  0.81636 36.68000
```

As would like to categorise `Sale Ratio` by its values, we need to proceed with `equal-depth` binning to equally distribute one-third of the values in `Low Priced`, one in `Medium Priced` and the last third in the `High Priced` category `Price Rubric`. We apply `discretize` from the `infotheo` package to the `Sale Ratio` variable with the `equalfreq` method, which ensures equal distribution across three bins *(Rdocumentation.org, 2022)*. After that, we convert the resulting binned values into a factor with corresponding labels to make it easier understandable. Finally, we display the results and verify the new variable.

```
# Equal depth binning to `Sale Ratio` in three bins
binned <- infotheo::discretize(connecticut_df$`Sale Ratio`, disc = "equalfreq", nbi
ns = 3)
# Convert variable to a factor with labels
connecticut_df$`Price Rubric` <- factor(binned$X,
                              labels = c("Low Priced","Medium Priced","High Price
d"))
# Display the result
connecticut_df %>% select(Towns, `Assessed Value`, `Sale Amount`,`Sale Ratio`,`Pric
e Rubric`) %>% head(3)
```

Towns <chr>	Assessed Value <dbl>	Sale Amount <dbl>	Sale Ratio <dbl>	Price Rubric <fct>
Andover	49000	59000	0.8305085	High Priced
Ansonia	72870	50000	1.4574000	High Priced
Ashford	116960	234450	0.4988697	Low Priced

3 rows

Lastly, we calculate the percentage of assisted housing by dividing `Total Assisted` variable by `Census 2010` and mutate the new variable `Ass. Housing Percentage` , rounded by two digits after zero, to our dataset `connecticut_df` .

```
# Mutate Assited Housing Percentage to the dataframe
connecticut_df %<>%
  mutate(`Ass. Housing Percentage` = round((`Total Assisted` / `Census 2010`) * 10
0,digits = 2))
connecticut_df[,c(1,4:5,9,14,17:ncol(connecticut_df))] %>% head(3)
```

Towns <chr>	Assessed Value <dbl>	Sale Amount <dbl>	Census 2010 <dbl>	County <chr>	Average Value <dbl>	Average <
Andover	49000	59000	1317	Tolland	93440436	
Ansonia	72870	50000	8148	New Haven	340374620	
Ashford	116960	234450	1903	Windham	106367059	

3 rows | 1-8 of 9 columns

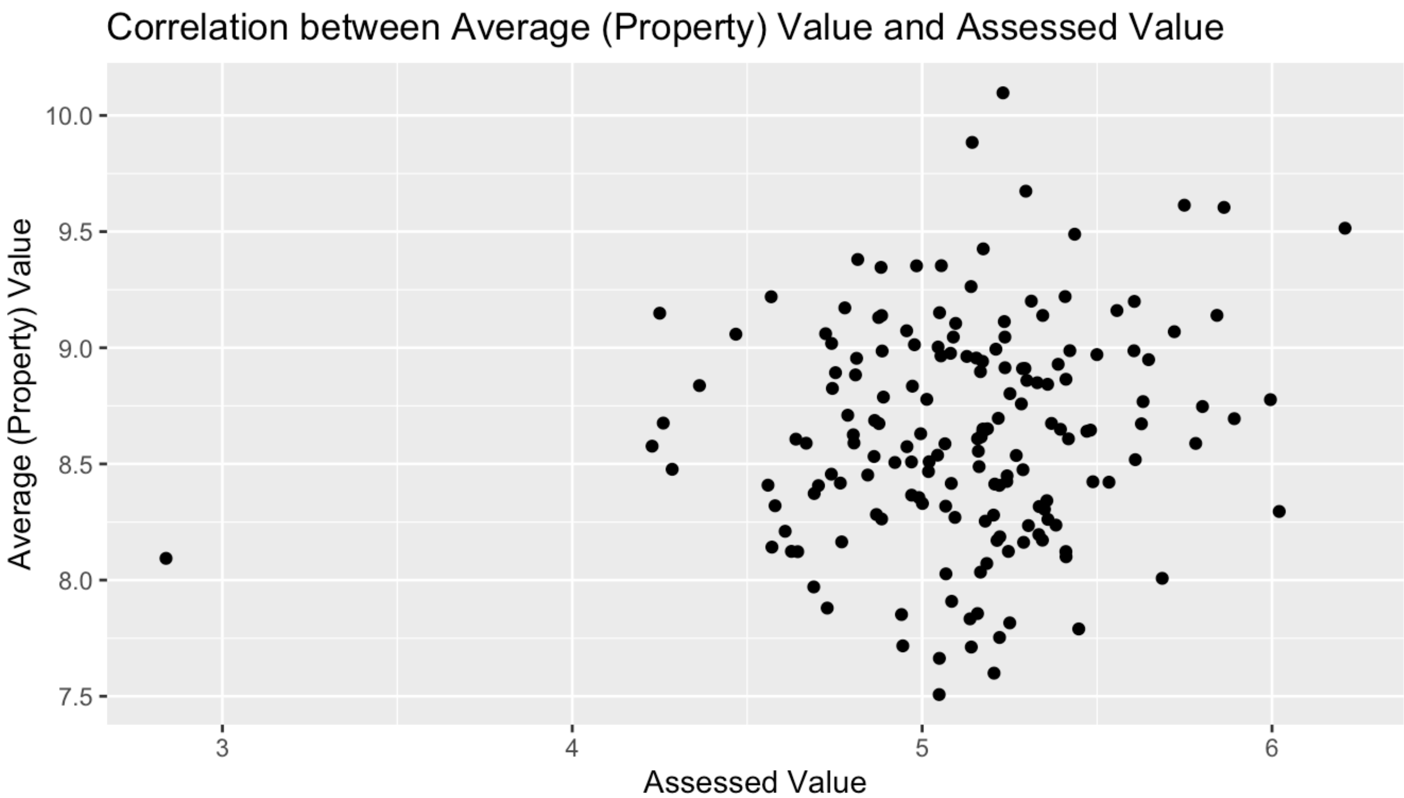
Before we start searching and fixing missing values, we should highlight the variable `Average Value` here. The values seem to be way too high when comparing to the variable `Assessed Value` or `Sale Amount` , let's see below if the variable is just formatted incorrectly but still displays the correct values. To ensure a correlation between the variable `Assessed Value` and `Average (Property) Value` is given, we apply the `log10` function on both variables to transform their values to a comparable scope and *change the scale for better understanding of the variable*. We apply `cor` on both transformed variables with all observations, by using `complete.obs` *(Rdocumentation.org, 2017)*.

```
# Correlation between `Average (Property) Value` and `Assessed Value`
log10_assessed_value <- log10(connecticut_df$`Assessed Value`)
log10_average_value <- log10(connecticut_df$`Average Value`)
cor(log10_assessed_value, log10_average_value, use = "complete.obs")
```

```
## [1] 0.1482207
```

It looks like the variable `Average Value` doesn't have any correlation with `Assessed Value`, therefore we further explore if the variable is incorrect. One way is the use of a `scatterplot`, to visualise the result more clearly. We call the `ggplot2` package, to plot both variables using the scatter plot `geom_point`. Finally, we label the axis and add a title using `labs` to enhance the results understanding.

```
library(ggplot2)
# Let's plot the correlation
ggplot(connecticut_df, aes(x=log10_assessed_value, log10_average_value)) + geom_poi
nt() + labs(title = "Correlation between Average (Property) Value and Assessed Valu
e", x = "Assessed Value", y = "Average (Property) Value")
```



The scatter plot proves again, no correlation exists between `Average Value` and `Assessed Value`, so we drop this variable.

```
# As no correlation is given, we drop the variable
connecticut_df %<>% select(-`Average Value`)
```

4.1 Scan I

4.1.1 Identifying missing and `0` values

This section will clear the dataset from any missing data and `0` values. We will create a new variable and store all `NA` in it. `colSums` in combination with `is.na` goes through all variables of `connecticut_df` and stores `Booleans` in `missing_values` , `TRUE` if values are missing and `FALSE` if not *(Rdocumentation.org)*. When indexing through the new variable and setting the condition to `missing_values > 0` , we receive all variables with missing data.

```
# Scan for missing values
missing_values <- colSums(is.na(connecticut_df))
missing_values[missing_values > 0]
```

```
##    Property Type Residential Type
##               62               62
```

Next, we use the same technique but adjust the parameters of our `Boolean` outputs by asking for a comparison. Are values of `connecticut_df` equal to `0` , `zero_values` will save this value as the logical operator `TRUE` . Any other value will be saved as `FALSE` . We again display all variables of `TRUE` Boolean and the total amount of `0` values.

```
# We can scan for Zero values of numerical variables
zero_values <- colSums(connecticut_df == 0, na.rm = TRUE)
zero_values[zero_values > 0]
```

```
##     Gov. Assisted Rental Assistance
##                24                15
```

4.1.2 Replacing missing and `0` values

After the identifictaion of all missing and `0` values we will replace them. Usually, it is recommended to replace missing qualitative data with their `mode` . As we are dealing with quite a lot of `NA's` for `Property Type` and `Residential Type` (almost 50% of all values), it is better to replace the missing data based on their appearances in the variable, instead of blindly replacing them by one entry. To do this, we create a function `replace_na` which takes a variable as an input. The function will create a new variable `prop` which contains the calculated proportion of each factor within the variable using `prop.table` . `prop.table` (from `dplyr`) takes a table `x` as an input, therefore we have to convert our dataframe to a table using `table` *(Zach, 2021)*. Next, it replaces `NA` values of that variable with random generated values using the `sample` function, while considering each values proportion (`prop=prop`).

Each sample output can occur multiple time due to the attribute `replace=TRUE` within the sample. `size=sum(is.na())` ensures that all missing values of variable `x` are replaced, whereas `return` ensures that we can use the specified amended value for further use and it ends the functions execution.

```
# Replacement of missing data
replace_na <- function(x) {
  proportion <- prop.table(table(x))
  # Replace NA with random sample values based on the proportion of that value with
in the var
  x[is.na(x)] <- sample(names(proportion), size = sum(is.na(x)),replace = TRUE, pro
b = proportion)
  return(x)
}
# Apply the function on our qualitative variables
connecticut_df$`Property Type` <- replace_na(connecticut_df$`Property Type`)
connecticut_df$`Residential Type` <- replace_na(connecticut_df$`Residential Type`)
# Check if we were sucessful
summary(connecticut_df$`Property Type`)
```

```
## Single Family Property  Three-Family Property                  Condo
##                    112                      5                     34
##   Residential Property    Commercial Property            Vacant Land
##                     15                      1                      2
```

```
summary(connecticut_df$`Residential Type`)
```

```
##  Single Household            Condo   Two-Family Home Three-Family Home
##               124               34                 4                 7
```

The output proves whether `Property Type` or `Residential Type` contain any more missing values, their level proportion also looks fairly distributed. The below code will now replace all `0` values of `Gov Assisted` and `Rental Assistance` , but first we display all variables of assisted housing in Connecticut.

```
# Return row with `0` values
connecticut_df %>%
  select(`Gov. Assisted`, `Rental Assistance`,`Financed Units`,`Total Assisted`) %
>%
  filter(`Gov. Assisted` == 0 | `Rental Assistance` == 0 | `Financed Units` == 0 |
`Total Assisted` == 0) %>% head(3)
```

Gov. Assisted <dbl>	Rental Assistance <dbl>	Financed Units <dbl>	Total Assisted <dbl>
32	0	36	68
0	6	23	29

0	4	46	50

3 rows

After inspecting `connecticut_df` variables of assisted housing, we see that `Gov. Assisted`, `Rental Assistance` and `Financed Units` are used to calculate `Total Assisted`. The above rows do look correct as the `0 values` do not seem to be implemented by mistake, as the variable `Total Assisted` is right. *In case we are having incorrect `0` values*, the below will still convert *all* `0` values to `NA` and then use a pre-defined mathematical formula `rules` to calculate and replace the `NA` values using the `validator` and `impute_lr` functions to fulfill the equation correctly.

```
# First we replace all `0` values by `NA`
connecticut_df[connecticut_df == 0] <- NA
# We define the rules as a validator expression
rules <- validate::validator(`Gov. Assisted` + `Rental Assistance` + `Financed Unit
s` == `Total Assisted`,
                             `Gov. Assisted` >= 0,
                             `Rental Assistance` >= 0,
                             `Financed Units` >= 0)
# Now, we use `impute_lr` on `connecticut_df` to fulfill the equation using `rules`
connecticut_df <- deductive::impute_lr(connecticut_df,rules)
connecticut_df %>%
  select(`Gov. Assisted`, `Rental Assistance`,`Financed Units`,`Total Assisted`) %
>%
  filter(`Gov. Assisted` == 0 | `Rental Assistance` == 0 | `Financed Units` == 0 |
`Total Assisted` == 0) %>% tail(3)
```

Gov. Assisted <dbl>	Rental Assistance <dbl>	Financed Units <dbl>	Total Assisted <dbl>
0	7	24	31
0	2	6	8
24	0	32	56

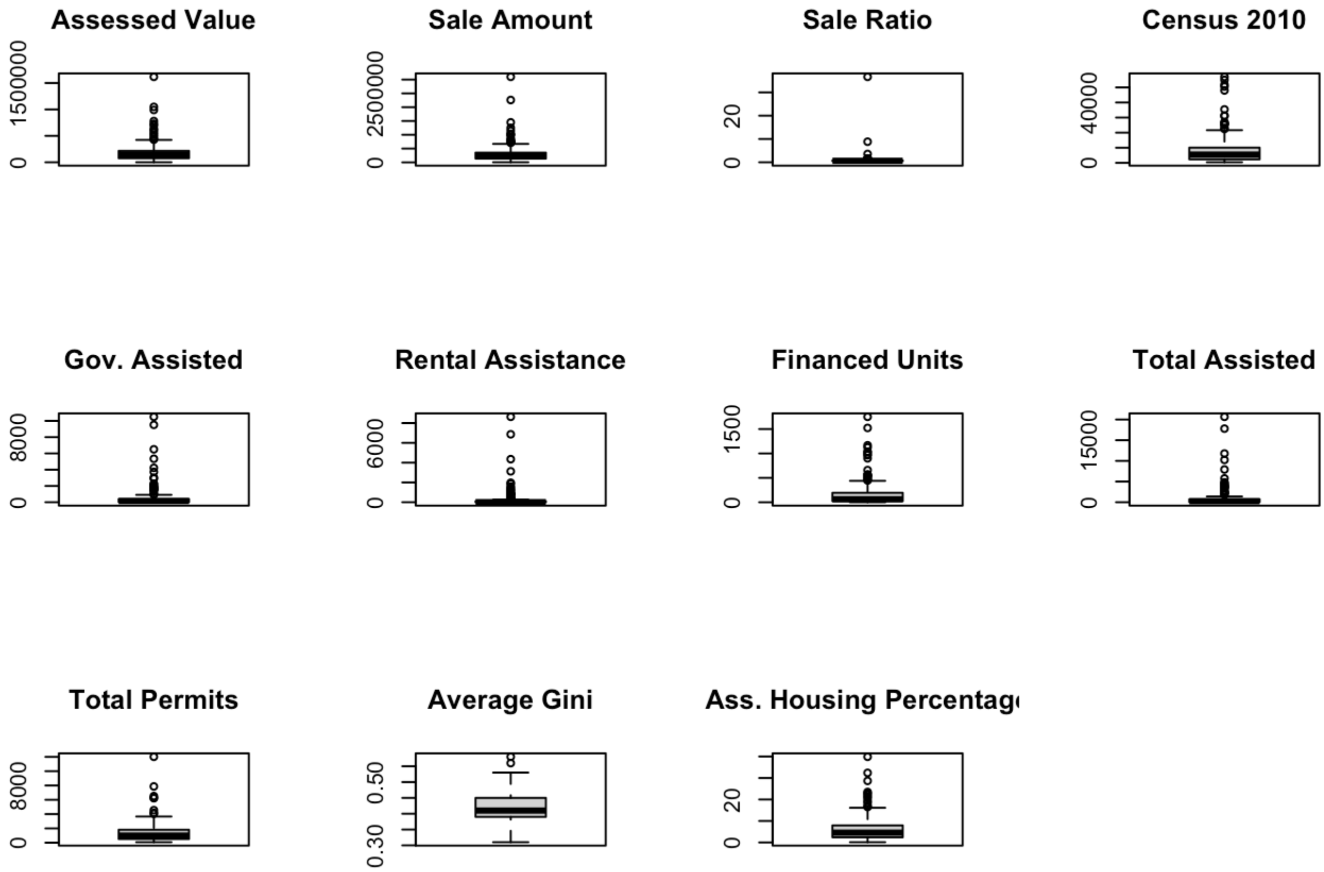
3 rows

The calculation of the last three values are all correct, it seems the `0` values are correct.

4.2 Scan II

This chapter will focus on the identification and standardisation of outliers in the numerical variables. First, we save all numerical variables in a new variable `numeric_var` by iterating the `is.numeric` function over `connecticut_df` using `sapply`. The following `for loop` iterates through each variable of `numeric_var`, creating a `boxplot` for each column, witch their column name as the title `main = col`.

```
# This is a chunk where you scan the numeric data for outliers
numeric_var <- sapply(connecticut_df, is.numeric)
par(mfrow = c(3,4))
for (col in names(connecticut_df)[numeric_var]){
  boxplot(connecticut_df[[col]], main = col)}
```



`par(mfrow))` displays all boxplots in a `3x4` grid. The results show that all variables contain significant upper outlier , as the values are bigger than their `upper fence` . We could use the `summary` statistics on the variables to check if lower outlier exist.

```
# First we check what values are considered upper or lower outlier
AV_q1 <- quantile(connecticut_df$`Assessed Value`, 0.25)
AV_q3 <- quantile(connecticut_df$`Assessed Value`, 0.75)
AV_iqr <- AV_q3 - AV_q1
AV_lower_fence <- AV_q1 - 1.5 * AV_iqr
AV_upper_fence <- AV_q3 + 1.5 * AV_iqr
cat("Lower Fence: ", AV_lower_fence, "\nUpper Fence ", AV_upper_fence)
```

```
## Lower Fence:  -131780
## Upper Fence  423900
```

Even thought the result seems false, the lower fence of `Assessed Value` can indeed be negative. The result simply means that this variable doesn't contain any lower outliers. Another way of testing our dataset for outliers is by applying the `Z-Score Standardisation`. First, we mutate a new variable to our dataset and apply `scale` to the variable, which will standardise all values to the corresponding z-score *(The number of standard deviations a data point is from the mean of a dataset)*. We have to convert the variable into a vector using `as.vector`, otherwise `scale` wouldn't work. We filter and display all values of `Total Permits` with a `Z-Score Total Permit` > 3 *(GeeksforGeeks, 2021)*.

```
connecticut_df %<>%
  mutate(`Z-Score Total Permit` = as.vector(scale(`Total Permits`)))
# Filter out z-scores > 3
connecticut_df %>% select(`Total Permits`, `Z-Score Total Permit`) %>%
  filter(abs(`Z-Score Total Permit`) > 3)
```

Total Permits <dbl>	Z-Score Total Permit <dbl>
7865	4.246816
6516	3.353344
6259	3.183127
6230	3.163920
12014	6.994789

5 rows

As this method of checking outliers can be very repetitive, we define a function `impute_outliers` which will perform the same process as above but replaces all values smaller than `lower_fence` or bigger than `upper_fence` with the *variables mean* (defined in `ifelse` and the *`or`* operator |). We apply the `impute_outliers` function to our numeric variables *four times* using a `for loop` due to the data distribution. Replacing the outlier with the mean changes the distribution of the data, which changes what values are considered outliers. To standardise newly created ones, we must perform the function multiple times as each process recalculates the `IQR` and their `fences`. Applying the method four times stabilises the data and no new outlier are identified.

```
# We save a variable before the transformation, please ignore this for now
before_sale_ratio <- connecticut_df$`Sale Ratio`
# Apply mean replacement for lower_fence and upper_fence outliers
impute_outliers <- function(var){
  q1 <- quantile(var, 0.25, na.rm = TRUE)
  q3 <- quantile(var, 0.75, na.rm = TRUE)
  iqr <- q3 - q1
  lower_fence <- q1 - 1.5 * iqr
  upper_fence <- q3 + 1.5 * iqr
  mean_var <- mean(var, na.rm=TRUE)

  var <- ifelse(var < lower_fence | var > upper_fence, mean_var , var)
  return(var)
}
# Apply the imputation to numeric variables three times to ensure it worked
for (i in 1:4){
  connecticut_df <- connecticut_df %>% mutate_if(is.numeric, impute_outliers)}
```

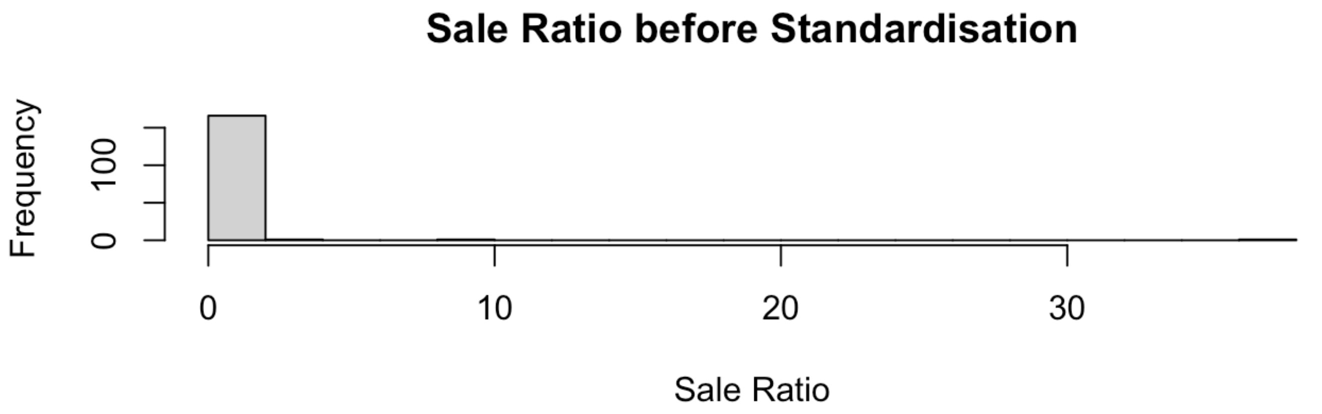
The below code is a copy of the inital `boxplot` creation to check if we were successful and all outliers are standardised.

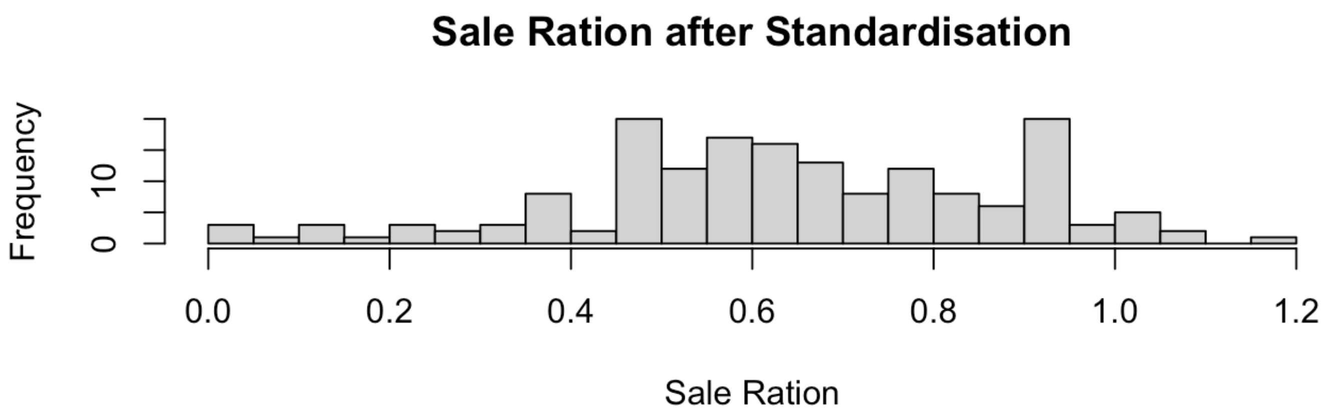
```
# Check if the standardisation worked
numeric_var <- sapply(connecticut_df, is.numeric)
par(mfrow = c(3,4))
for (col in names(connecticut_df)[numeric_var]){
  boxplot(connecticut_df[[col]], main = col)}
```

The new numeric variables don't contain any more outliers. If we consider the scope of the `y-axis` , we can see that the distribution of the data changed drastically *(e.g.* `Rental Assistance` *was originally showing data ranging from 0 to 8000, now 0 to 120)*.

A very convincing effect of the above can be seen below, where we compare the before and after of the variable `Sales Ratio` using a histogram with 20 bins.

```
# Visualise Sale Ratio before and after Standardisation
par(mfrow = c(2,1))
hist(before_sale_ratio, main = "Sale Ratio before Standardisation", xlab = "Sale Ra
tio", breaks = 20)
hist(connecticut_df$`Sale Ratio`, main = "Sale Ration after Standardisation", xlab
= "Sale Ration", breaks=20)
```





Without this standardisation, we couldn't proceed with the transformation the data, to e.g. reduce the skewness even more.

5. Transform

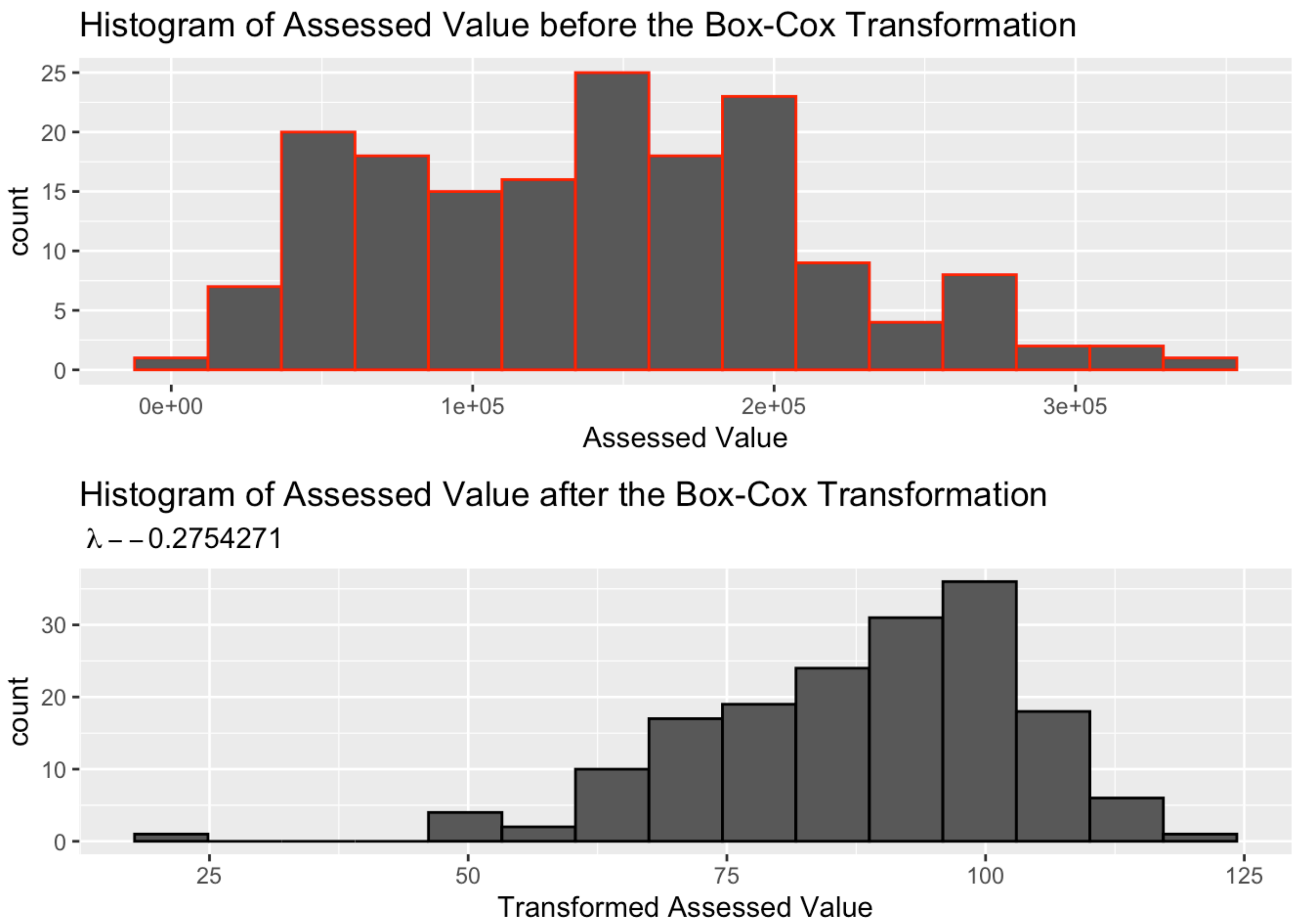
5.1 BoxCox transformation of `Assessed Value`

This section will focus on the transformation of some of `connecticut_df` numerical variables. We start by improving the distribution of the variable `Assessed Value`.

```
# This is a chunk where you apply an appropriate transformation to at least one of
the variables
library(forecast)
```

```
## Registered S3 method overwritten by 'quantmod':
##   method            from
##   as.zoo.data.frame zoo
```

```
boxcox_SR <- BoxCox(connecticut_df$`Assessed Value`, lambda = "auto")
lambda <- attr(boxcox_SR, which = "lambda")
# Defining the histogram before the bxcox transformation using ggplot2
graph1 <- ggplot(connecticut_df, aes(`Assessed Value`)) + geom_histogram(bins = 15,
color = "red") +
  labs(title = "Histogram of Assessed Value before the Box-Cox Transformation")
# Defining the boxcox of Assessed Value using ggplot2
p1 <- ggplot(boxcox_SR %>% as.data.frame())
graph2 <- p1 + geom_histogram(aes(x = .), bins = 15, color = "black") +
  labs(title = "Histogram of Assessed Value after the Box-Cox Transformation",
       subtitle = bquote(~ lambda -- .(lambda)),
       x = "Transformed Assessed Value")
gridExtra::grid.arrange(graph1 ,graph2, nrow = 2) # Display both graphs
```



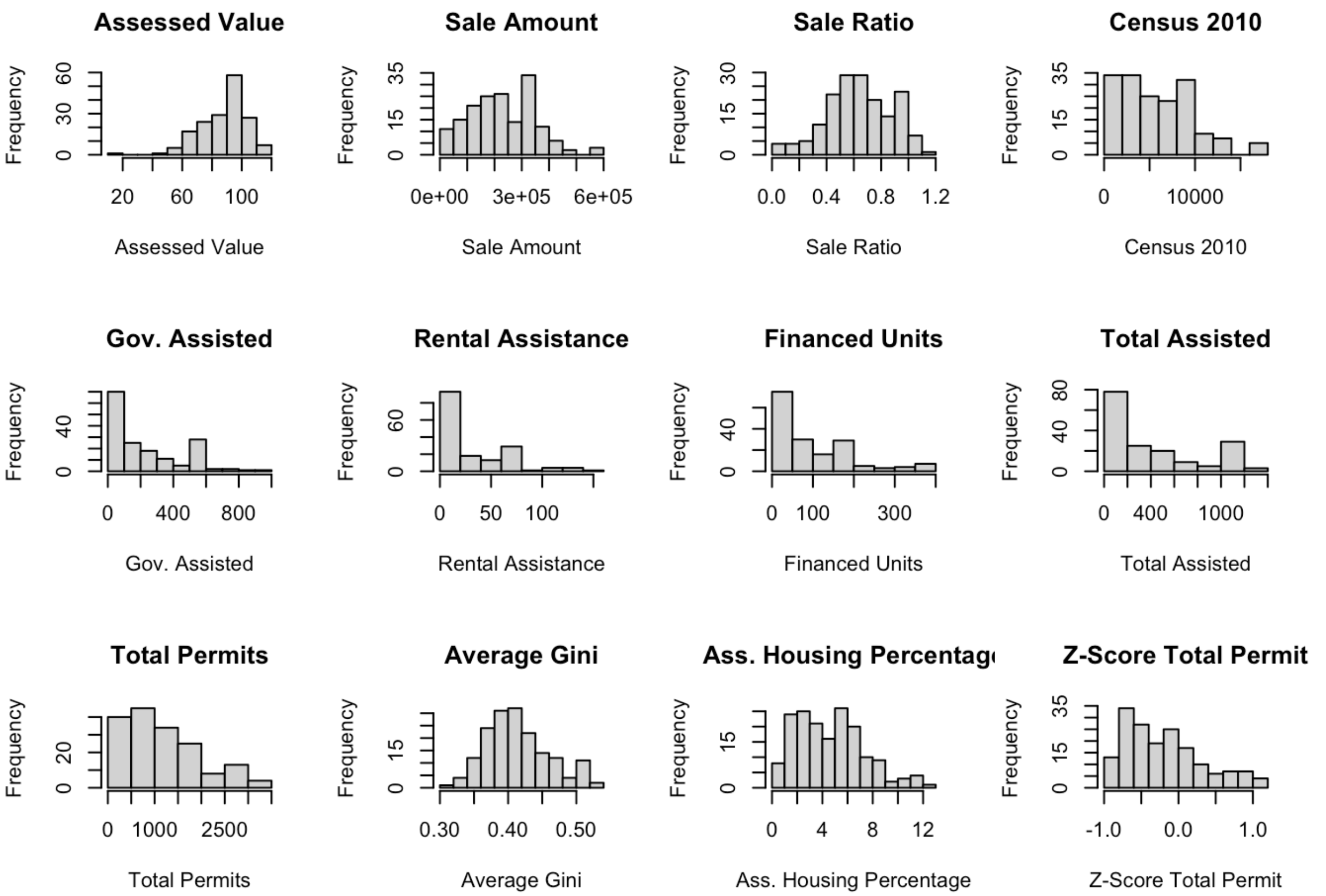
The BoxCox transformation was moderately effective in improving the variables distribution. Before the transformation, `Assessed Value` was distributed irregular, `BoxCox` definitely improved the symmetry visibly. `Lamba` has a value of `-0.2754` , which underlines that this transformation was moderately successful, as the data is now a bit left-skewed *(Rdocumentation.org, 2023)*. Finally, we update the values of `Assessed Value` with the transformed values.

```
# Update `Assessed Value` with the transformed values
connecticut_df$`Assessed Value` <- boxcox_SR
```

5.2 Square-root transformation of `Ass. Housing Percentage`

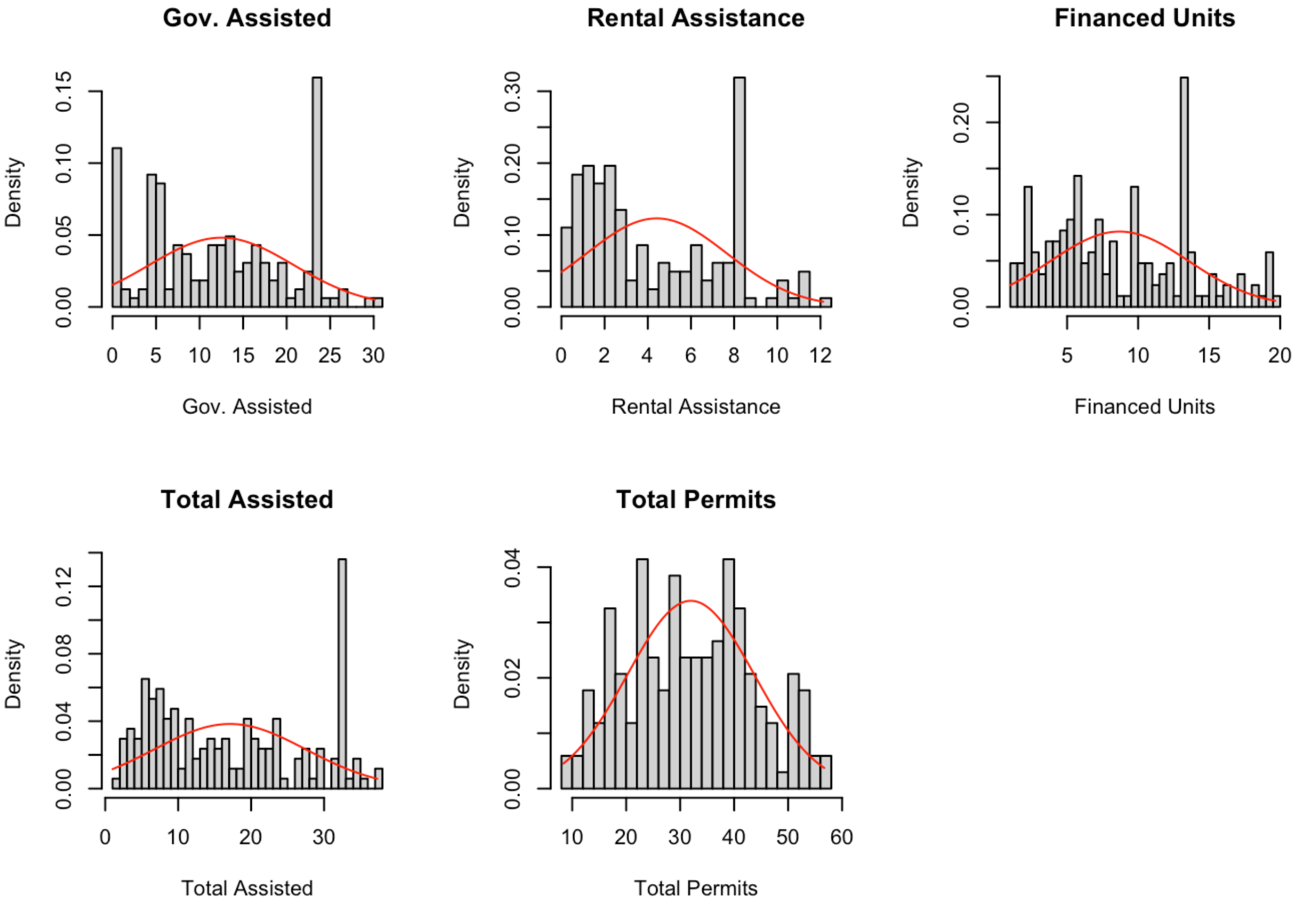
Next, we would like to reduce the skewness of right-skewed data into a more distributed dataset. We use the same `for loop` as before, which displayed the boxplots of the numeric variables, and replace `boxplot` with `hist` to display the variables distribution.

```
# Plot `Total Assisted`
par(mfrow = c(3:4))
for (col in names(connecticut_df)[numeric_var]){
  hist(connecticut_df[[col]], main = col, xlab = col)}
```



Next, we subset all right-skewed variables and apply `sqrt` - the square root transformation - with the goal of an output which is more symmetrical *(Educative, 2024)*.

```
# Subset all right-skewed variables
right_skewed_var <- connecticut_df %>% select(`Gov. Assisted`,`Rental Assistance`,`Financed Units`,`Total Assisted`,`Total Permits`)
# Apply the square root transformation
right_skewed_var %<>% sqrt()
# Display all numerical right-skewed variables to see if their distribution improved
par(mfrow = c(2:3))
for (col in names(right_skewed_var)) {
  hist(right_skewed_var[[col]], freq = FALSE, breaks = 30, main = col, xlab = col)
  # This block displays the normal distribution of each graph
  x <- seq(min(right_skewed_var[[col]], na.rm = TRUE), max(right_skewed_var[[col]], na.rm = TRUE), length.out = 100)
  lines(x, dnorm(x, mean = mean(right_skewed_var[[col]], na.rm = TRUE), sd = sd(right_skewed_var[[col]], na.rm = TRUE)), col = 'red')}
```



The `square root transformation` was a success, especially for variables like `Gov. Assisted`, `Financed Units` and `Total Permits`. The distribution of the other variables also improved slightly. The red line shows each datasets ideal distribution by using `lines()` and calculating each standard distribution individually `sd(right_skewed_var[[col]])` *(Bobbitt, 2023)*.

6. Reflective Journal

In this assessment I tried to answer the question of how I can ensure all datasets with varying structures and format are transformed into a unified, analysable format. I applied the five distinct steps of Data Wrangling, beginning with the Import and Understanding, moving through the Tidying and Manipulation, then onto Scanning, and finally, the Transformation of five different real-world datasets. The objective to answer my initial question was to convert these datasets into a unified, analysable format that would be suitable for in-depth analysis.

Throughout this project, I encountered several challenges, particularly in handling the varying data formats and converting them into a cohesive, mergeable and tidy data structure. The process was further complicated by the requirement to handle missing values, identify and address outliers, and apply appropriate transformation to ensure the integrity of the final dataset. One difficulty was the transformation of right-skewed datasets into more symmetric distributions using the square root transformation. Similarly, the application of a successful BoxCox transformation was challenging, but it provided valuable experiences in applying different transformation techniques and assessing their effectiveness.

Throughout this work, I have gained valuable insights into the complexities of data wrangling and the importance of the step-by-step data pre-processing. It challenged me to apply a wide range of data wrangling techniques, from web scraping and tidying data to handling missing values, outliers, and performing transformations. Each step provided an opportunity to deepen my understanding of data wrangling and refine my approach when working with real-world datasets.

7. Presentation Link

My presentation of this Assessment can be found here (https://rmit-arc.instructuremedia.com/embed/30cf1fef-332c-493f-b54b-dced813ad73a)

8. Sources

- Connecticut Open Data. (2024). Connecticut Open Data. [online] Available at: https://data/ct.gov (https://data/ct.gov) (https://data/ct.gov (https://data/ct.gov)) [Accessed 05 Aug. 2024].
- GitHub. (2024). RSocrata. [online] Available at: https://github.com/Chicago/RSocrata (https://github.com/Chicago/RSocrata) (https://github.com/Chicago/RSocrata (https://github.com/Chicago/RSocrata)) [Accessed 06 Aug. 2024].
- Wickham, H. and Grolemund, G. (2016). R for Data Science: Import, Tidy, Transform, Visualize, and Model Data. 1st Edition. O'Reilly Media [Accessed 08 Aug. 2024].
- Dplyr.tidyverse.org. (2024). Subset rows using their positions — slice. [online] Available at: https://dplyr.tidyverse.org/reference/slice.html (https://dplyr.tidyverse.org/reference/slice.html) [Accessed 08 Aug. 2024].
- Rdocumentation.org. (2018). slice function - RDocumentation. [online] Available at: https://www.rdocumentation.org/packages/dplyr/versions/0.7.6/topics/slice (https://www.rdocumentation.org/packages/dplyr/versions/0.7.6/topics/slice) [Accessed 09 Aug. 2024].
- R-Packages. (2024). Introduction. [online] Available at: https://cran.r-project.org/web/packages/tidyr/vignettes/pivot.html (https://cran.r-project.org/web/packages/tidyr/vignettes/pivot.html) [Accessed 09 Aug. 2024].
- Dplyr.tidyverse.org. (2024). Mutating joins — mutate-joins. [online] Available at:

https://dplyr.tidyverse.org/reference/mutate-joins.html (https://dplyr.tidyverse.org/reference/mutate-joins.html) [Accessed 09 Aug. 2024].

- Rdocumentation.org. (2024). Sapply function - RDocumentation. [online] Available at: https://www.rdocumentation.org/packages/memisc/versions/0.99.31.7/topics/Sapply (https://www.rdocumentation.org/packages/memisc/versions/0.99.31.7/topics/Sapply) [Accessed 10 Aug. 2024].
- R-lib.org. (2024). Splice operator !!! — splice-operator. [online] Available at: https://rlang.r-lib.org/reference/splice-operator.html (https://rlang.r-lib.org/reference/splice-operator.html) [Accessed 10 Aug. 2024].
- Package 'lubridate' Type Package Title Make Dealing with Dates a Little Easier. (2020). Available at: https://cran.r-project.org/web/packages/lubridate/lubridate.pdf (https://cran.r-project.org/web/packages/lubridate/lubridate.pdf) [Accessed 10 Aug. 2024].
- Rpubs.com. (2024). RPubs - lapply, sapply, and vapply. [online] Available at: https://rpubs.com/GilbertTeklevchiev/1113536 (https://rpubs.com/GilbertTeklevchiev/1113536) [Accessed 10 Aug. 2024].
- Rdocumentation.org. (2016). rowMeans function - RDocumentation. [online] Available at: https://www.rdocumentation.org/packages/fame/versions/1.03/topics/rowMeans (https://www.rdocumentation.org/packages/fame/versions/1.03/topics/rowMeans) [Accessed 10 Aug. 2024].
- Rdocumentation.org. (2022). infotheo package - RDocumentation. [online] Available at: https://www.rdocumentation.org/packages/infotheo/versions/1.2.0.1 (https://www.rdocumentation.org/packages/infotheo/versions/1.2.0.1) [Accessed 11 Aug. 2024].
- Rdocumentation.org. (2017). log10 function - RDocumentation. [online] Available at: https://www.rdocumentation.org/packages/SparkR/versions/2.1.2/topics/log10 (https://www.rdocumentation.org/packages/SparkR/versions/2.1.2/topics/log10) [Accessed 11 Aug. 2024].
- Rdocumentation.org. (2024). colSums function - RDocumentation. [online] Available at: https://www.rdocumentation.org/packages/base/versions/3.6.2/topics/colSums (https://www.rdocumentation.org/packages/base/versions/3.6.2/topics/colSums) [Accessed 11 Aug. 2024].
- Zach (2021). How to Use prop.table() Function in R (With Examples). [online] Statology. Available at: https://www.statology.org/r-prop-table/ (https://www.statology.org/r-prop-table/).
- GeeksforGeeks. (2021). Plot Z-Score in R. [online] Available at: https://www.geeksforgeeks.org/plot-z-score-in-r/ (https://www.geeksforgeeks.org/plot-z-score-in-r/) [Accessed 11 Aug. 2024].
- Rdocumentation.org. (2023). boxcox function - RDocumentation. [online] Available at: https://www.rdocumentation.org/packages/EnvStats/versions/2.8.1/topics/boxcox (https://www.rdocumentation.org/packages/EnvStats/versions/2.8.1/topics/boxcox) [Accessed 12 Aug. 2024].
- Educative. (2024). Educative Answers - Trusted Answers to Developer Questions. [online] Available at: https://www.educative.io/answers/how-to-calculate-the-square-root-of-a-number-in-r (https://www.educative.io/answers/how-to-calculate-the-square-root-of-a-number-in-r) [Accessed 12 Aug. 2024].
- Bobbitt, Z. (2023). How to Use lines() Function in R (With Examples). [online] Statology. Available at: https://www.statology.org/lines-function-in-r/ (https://www.statology.org/lines-function-in-r/) [Accessed 13 Aug. 2024].