January 2023

Quality Indicators for Administrative Data[1]

User Manual For R Package on GitHub

Sook Kim and Natalie Shlomo
University of Manchester

[1] Funded by the Grant: ‘Methodological Advancements on the Use of Administrative Data in Official Statistics’ ESRC (ES/V005456/1)

Contents

1. Project aims and objectives

The Office for National Statistics (ONS) have strategic priorities on embedding and advancing the use of administrative data into their official statistics processes. Their immediate priority is to use administrative data in the quality assurance of the 2021 census and to develop the production of administrative-based population estimates (ABPEs). A more long-term priority is the Population Statistics Transformation Programme which will feed into a recommendation to Government due in 2023 on the future of census and population statistics. In particular, the objective is to create population characteristic estimates from administrative and integrated data sources.

Motivated by these initiatives, the University of Manchester was successfully awarded an ESRC grant from April 2021 to January 2023 titled: 'Methodological Advancements on the Use of Administrative Data in Official Statistics' (ES/V005456/1). The funded research enabled collaborations with the ONS on researching and developing methods to enhance the use of administrative data in official statistics.

This user manual concerns the sub-project related to developing a quality framework and quantitative measures to assess representativeness and coverage for a single administrative data source. We use univariate and bivariate distributions obtained from high- quality large random probability surveys (such as the UK Annual Population Survey) and compare them to distributions in the administrative data on a common set of variables based on distance metrics between these distributions. In addition, we developed a Representativity (R-) Indicator that is designed for quantifying the representativeness of population groups in the administrative data. Section 2 describes the methods underpinning the R-code hosted in GitHub and Section 3 describes the R-code with a running example of its outputs. We conclude in Section 4 with troubleshooting questions and answers.

2. Quality indicators

In the R-code on GitHub we focus on quality indicators to identify errors arising from coverage and representativeness of a single administrative data source. It is vital that statistical agencies have good quality indicators to ensure the fit of administrative data to the population and to identify those sub-groups that are missing or over-covered, especially when the administrative data contributes to an integrated dataset for multisource processing or to quality assure other data sources, such as surveys and censuses.

2.1 Distance Metrics

In this section we compare distributions obtained from the administrative data with external population auxiliary information, either obtained directly from a census or estimated from a large probability-based random survey.

Denote variable v having categories $h, h = 1, 2 \dots H$ in the administrative dataset having M individuals. Let $\Delta_{h,i}^{v}$ be the 0-1 indicator for individual i being a member of category h in variable v. We can then calculate the counts for category h : $m_h^v = \sum_{i=1}^{M} \Delta_{h,i}^{v}$ and note that $\sum_{h=1}^{H} m_h^v = M$. We can also obtain the probability distribution of variable v having category $h, h = 1, 2 \dots H$ and calculate: $p_h^v = \frac{m_h^v}{M}$. We note that the variable v can also represent a cross-tabulation of two or more variables, for example age group × sex.

Now assume we have equivalent estimates of these distributions from a census, or alternatively from a large probability-based random sample of size n where every individual i in the sample has an associated survey weight w_i . The survey weights typically are calibrated to known population benchmarks and hence sum to the known population size N. In this case, $n_h^v = \sum_{i=1}^{n} w_i \Delta_{h,i}^{v}$ and $\sum_{h=1}^{H} n_h^v = N$. Moreover, the equivalent distribution is $q_h^v = \frac{n_h^v}{N}$.

The entropy measures the uniformity of the probability distributions and hence can be compared when calculated on the administrative data distribution versus the population distribution. The formula for the entropy on the probability distribution $\{p_h^v, h = 1, \dots, H\}$ is: $-\sum_h p_h^v \, log(p_h^v)$ and similarly on the probability distribution $\{q_h^v, h = 1, \dots, H\}$.

We can now use a variety of distance metrics to assess deviations between the distributions $\{p_h^v, h = 1, \dots, H\}$ and $\{q_h^v, h = 1, \dots, H\}$. In this research, we assess three distance metrics: the Indicator of Dissimilarity (Duncan and Duncan, 1955), Hellinger's Distance (HL) and the Kullback-Leibler divergence (KL). The formula for the three distance metrics is given in Table 1.

Table 1: Distance metrics between the distribution calculated from the administrative data $\{p_h^v, h = 1, \dots, H\}$ and the distribution from the population $\{q_h^v, h = 1, \dots, H\}$ on variable v

Distance Metrics	Formula	Standarize
Indicator of Dissimilarity (ID)	$\frac{1}{2}\sum_h \lvert p_h - q_h \rvert$	1-ID
Hellinger's Distance (HL)	$\frac{1}{\sqrt{2}}\sqrt{\sum_h \left(\sqrt{p_h} - \sqrt{q_h}\right)^2}$	1-HL

Kullback-Leibler Divergence (KL)	$\sum_h p_k \, log\left(\frac{p_h}{q_h}\right)$	1-KL

There are subtle differences between these distance metrics. For example, Hellinger's Distance places more weight on the smaller proportions compared to the larger proportions whereas the Indicator of Dissimilarity treats all proportions equally. More work is needed on standardizing the distance metrics into meaningful quality measures. We have yet to determine which distance metric should be used and therefore propose to include all of them in the R-code. This will facilitate more empirical work for future recommendations.

2.2 R-indicators

The R-indicator and its related partial R-indicators were originally designed to assess the representativeness of responses from a survey and are particularly useful as an objective function in the optimization of adaptive survey designs (Schouten, et al. 2009, Schouten and Shlomo, 2017). The R-indicators measure the contrast between those who are missing and not missing in the data and identify those groups that are not represented in the data. Here, we develop the R-indicator and partial R-indicators to assess the representativeness and coverage of an administrative dataset compared to a target population. Recent research by Bianchi, et al. (2019) adapts the R-indicator to the case where only population-based auxiliary information are available instead of sample-based frame information. We draw upon this research and utilize population-based auxiliary information where the population auxiliary information is obtained by weighted survey counts from a large probability-based random sample.

To calculate the population-based R-indicator, denote the response indicator r_i equal to 1 for all units in the administrative dataset. We have information available on the values $\boldsymbol{x_i} = (x_{1,i}, x_{2,i}, \dots, x_{K,i})^T$ of a vector of K auxiliary variables ***X***, for example, sex, age group, geographical region, ethnic minority group and employment status. Therefore, each $x_{k,i}$ is a binary indicator variable. We also assume that values of $\boldsymbol{x_i}$ are observed for all individuals in the administrative dataset so that $\{x_i; i \in r\}$ is observed.

Assume we know $\boldsymbol{x_i}$ at the aggregate level: the population total $\sum_U \boldsymbol{x_i}$ and population cross-products $\sum_U \boldsymbol{x_i}\, \boldsymbol{x_i}^{\boldsymbol{T}}$. This information is known as the population–based auxiliary information. If this information is not available at the population level we can estimate the aggregates and cross-products using a large probability-based random sample, denoted s, where each individual i in the sample has a survey weight w_i. The estimated population-based auxiliary information is then: $\sum_s w_i \boldsymbol{x_i}$ and $\sum_s w_i \boldsymbol{x_i} \boldsymbol{x_i^T}$. We also know the overall total in the population, denoted by N.

Response propensities are defined as the conditional expectation of the response indicator variable r_i given the values of specified variables: $\rho_i \equiv \rho_X(\boldsymbol{x_i})$. In the population-based setting we model the response propensities under an identity link function where the true response propensities satisfy: $\rho_i = \boldsymbol{x_i^T \beta}, i \in U$. For the linear probability model, the estimate of ρ_i in the sample-based scenario is given by: $\hat{\rho}_i^{OLS} = \boldsymbol{x_i^T} (\sum_{\boldsymbol{s'}} d_i \, \boldsymbol{x_i x_i^T})^{\boldsymbol{-1}} \sum_{\boldsymbol{s'}} d_i \, \boldsymbol{x_i} r_i \quad , i \in s'$, where d_i is the design weight and s' denotes the sample under analysis.

In the case of population-based auxiliary information where we know both population totals and cross-products, we note that $\sum_{s'} d_i \boldsymbol{x_i}$ and $\sum_{s'} d_i \boldsymbol{x_i} \, \boldsymbol{x_i}^{\boldsymbol{T}}$ are unbiased estimates for $\sum_U \boldsymbol{x_i}$ and $\sum_U \boldsymbol{x_i} \, \boldsymbol{x_i}^{\boldsymbol{T}}$, respectively and that in large samples we may expect that $\sum_{s'} d_i \boldsymbol{x_i} \approx \sum_U \boldsymbol{x_i}$ and $\sum_{s'} d_i \boldsymbol{x_i} \, \boldsymbol{x_i}^{\boldsymbol{T}} \approx \sum_U \boldsymbol{x_i} \, \boldsymbol{x_i}^{\boldsymbol{T}}$. It follows that, in the population-based setting, we may approximate $\hat{\rho}_i^{OLS}$ by $\hat{\rho}_i^P = \boldsymbol{x_i^T} (\sum_{\boldsymbol{U}} \boldsymbol{x_i x_i^T})^{\boldsymbol{-1}} \sum_{\boldsymbol{r}} d_i \boldsymbol{x_i}, i \in r$ and we refer to the propensities as 'participation' propensities. Note that $\hat{\rho}_i^P$ is computed only on the set of individuals in the administrative data. In addition, in our setting of assessing the representativeness in administrative data, the design weight d_i is the inverse of the coverage weight: $\quad d_i = [M/N]^{-1}$ where M is the number of individuals in the administrative dataset.

In the population-based setting, an estimator for the R-indicator is given by $\hat{R}_{\hat{\rho}^P} = 1 - 2\hat{S}_{\hat{\rho}^P}$ where $\hat{S}_{\hat{\rho}^P}^2 = \frac{N}{N-1} \{ \frac{1}{N} \sum_r d_i \hat{\rho}_i^P - [\frac{1}{N} \sum_r d_i]^2 \}$ and $\hat{\rho}_i^P$ is estimated as above. This estimator of the R-indicator makes the estimator $\hat{S}_{\rho}^2$ linear in $\hat{\rho}_i^P$ which provides an advantage for size bias adjustment computations (although given the large administrative datasets, a size bias adjustment is not needed). Furthermore, we use propensity weighting by $\hat{\rho}_i^{P^{-1}}$ to adjust for coverage bias. The R-indicator measures the variation of the sub-group participation propensities. If the participation propensities are all equal and there is no variation in sub-group participation, the R-indicator would obtain a value of 1.

The unconditional partial R-indicator measures the amount of variation of the participation propensities between the categories of a variable. The larger the between-category variation is, the stronger the relationship is and the stronger the impact of the variable on a lack of representativeness. As earlier, let $\boldsymbol{x_k}$ be one of the components of the vector $\boldsymbol{X}$. The variable $\boldsymbol{x_k}$ is categorical and assume it has H categories. Let m_h denote the weighted respondent size in category h in the administrative data, for $h = 1, 2, \ldots, H$. That means $m_h = \sum_{i \epsilon r} d_i \Delta_{h,i}$ where $\Delta_{h,i}$ is the 0-1 indicator for participating unit i being a member of category h and $\sum_{h=1}^H m_h = N$ given the definition of d_i as the inverse coverage weight. Define $\hat{\bar{\rho}}_h$ the average of the participation propensities in category h of $\boldsymbol{x_k}$ for the units in the administrative dataset and $\hat{\bar{\rho}}$ the overall average participation probability based on the estimated population-based participation propensities $\hat{\rho}_i^P$. The estimate for the unconditional partial R-indicator for variable $\boldsymbol{x_k}$ is: $R_U(\boldsymbol{x_k}) = \sqrt{\frac{1}{N} \sum_{h=1}^H m_h (\hat{\bar{\rho}}_h - \hat{\bar{\rho}})^2}$. The upper bound of the unconditional partial R-indicator is 0.5. The larger the value of the partial R-indicator, the stronger the association of the variable with a lack of representativeness in the administrative dataset. By computing and comparing the unconditional partial indicators for a set of variables it can be established for which variables the relationships are strongest. The unconditional partial R-indicator at the category level h for variable $\boldsymbol{x_k}$ is $R_U\left(\boldsymbol{x_k^h}\right) = \sqrt{\frac{m_h}{N}}\,(\hat{\bar{\rho}}_h - \hat{\bar{\rho}})$ and can assume positive and negative values. Note that at the category-level, a negative sign represents under-representation and a plus sign represents over-representation.

Finally, we note that when producing estimates from the administrative dataset, one should weight each individual i by its inverse participation propensity: $\hat{\rho}_i^{P^{-1}}$ to adjust for coverage bias in the estimates.

3. User Guide on the R-package

This package assumes that there is a Census file to obtain auxiliary population estimates and the administrative datasets under analysis. However, in real settings we would not have a Census microdata to work with, rather we would have a large probability-based survey sample for estimating population distributions. Therefore, we draw a random sample from the Census microdata to support this scenario and obtain auxiliary population totals from weighted sample counts.

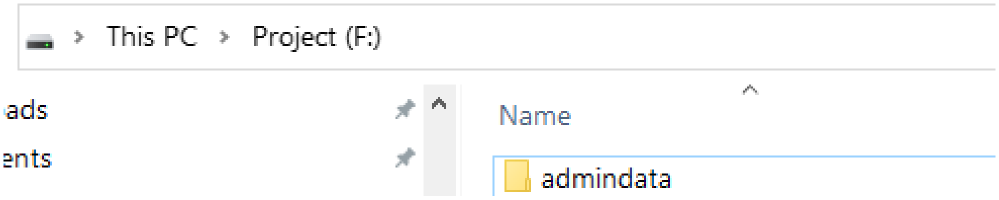
3.1 Download and inspect the contents

3.1.1 Download

Please visit the github site here: qualadmin link. Click on Code at the top-right corner. Then, click on `Download ZIP` to download to your local machine.

Now, the downloaded folder needs to be placed in the designated location. We recommend users decide the appropriate Drive (C, D, E, F, etc) to house the downloaded contents. Then, **create a new folder** called `admindata` in File Explorer of your PC. Users can customise the new folder name as appropriate. This is your **starting path**.

The screenshot showing **starting path**:



Under this Starting path, `F:/admindata`, place the downloaded folder from GitHub. Extract the zip folder as necessary.

As such, `F:/admindata/qualadmin` becomes the MASTER project folder. We'll set it as working directory[1] in RStudio later.

3.1.2 Downloaded contents explained

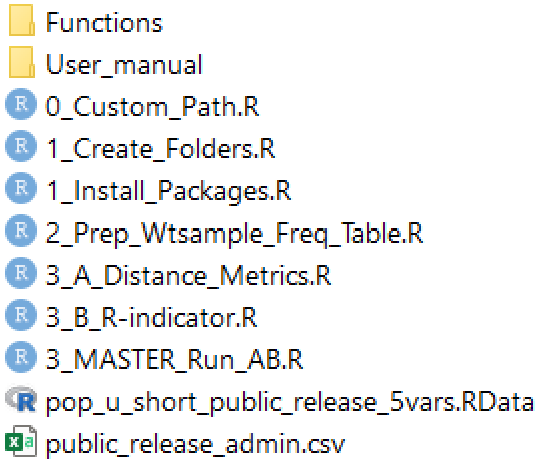
Example datasets

We provide two example data sources.

Data type	File name
Administrative	public_release_admin.csv
Census	pop_u_short_public_release_5vars.Rdata

Folders and R scripts

Under `F:\admindata\qualadmin` folder, you'll be presented with the following contents.



```
Functions
User_manual
0_Custom_Path.R
1_Create_Folders.R
1_Install_Packages.R
2_Prep_Wtsample_Freq_Table.R
3_A_Distance_Metrics.R
3_B_R-indicator.R
3_MASTER_Run_AB.R
pop_u_short_public_release_5vars.RData
public_release_admin.csv
```

The **"User_manual"** folder contains instructions on using the provided R code files.

The users do not need to do anything with the folder titled **"Functions"**. These pre-defined functions are used to either enclose complex procedures or perform repetitive tasks including cleaning and computing quality indicators. There are two files containing pre-defined functions. There is no need to run function files independently.

The functions will be automatically called in when the three main R script files are run: `2_Prep_Wtsample_Freq_Table.R` `3_A_Distance_Metrics.R` `3_B_R-indicator.R`

The `2_Prep_Wtsample_Freq_Table.R` file creates necessary data needed to compute distance metrics and R-indicators. The master file, `3_MASTER_Run_AB.R` runs the above *two* main R script files, (`3_A_Distance_Metrics.R` `3_B_R-indicator.R`) automatically in sequence.

[1] Notice that the terms, folder, directory, and path are used interchangeably in the user manual.

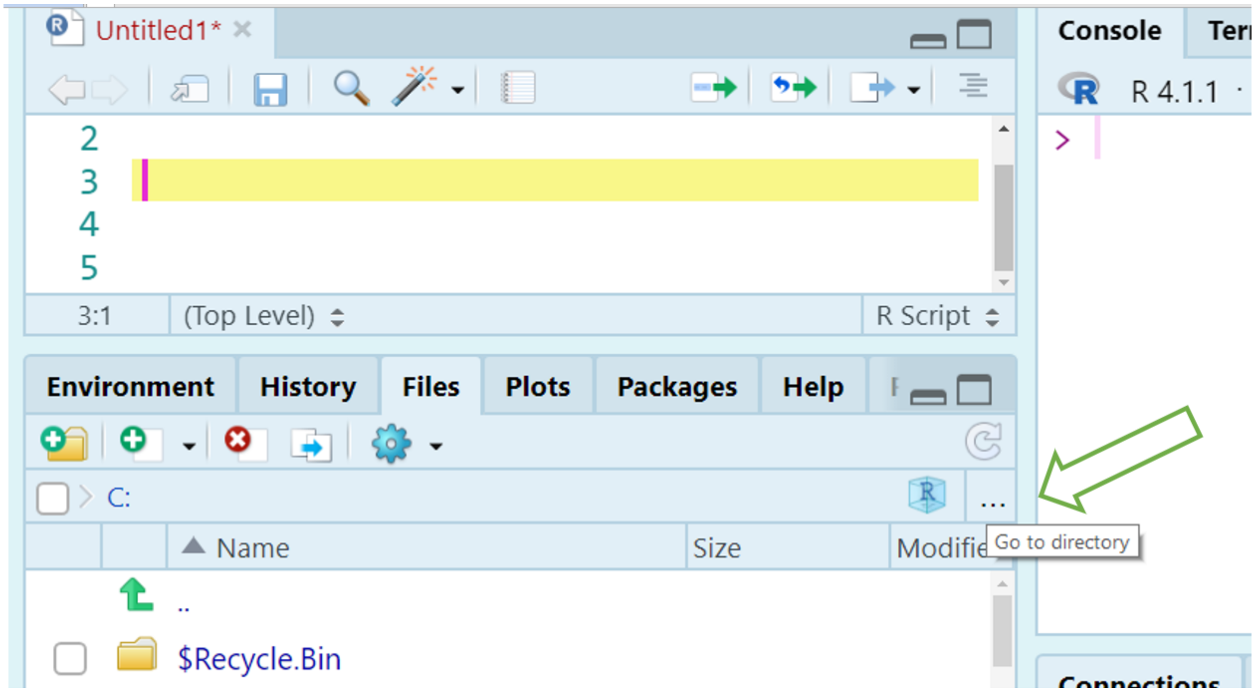
The first three files, `0_Custom_Path.R`, `1_Create_Folders.R` and `1_Install_Packages.R` can be run to get ready to run the above main analysis files, as discussed in the following section.

3.2 Launch RStudio and get ready

3.2.1 Open the entire master folder in RStudio

First, launch RStudio. Then, we need to **open the entire folder** `F:/admindata/qualadmin` where downloaded materials are located.

Unfortunately, RStudio has no feature in the menu, but you could do so by accessing **Files** tab. Click on `...` as shown below.



Then, locate the master folder. In our example, it is `F:/admindata/qualadmin`.

3.2.2 Set custom path

Click open the R script file, `0_Custom_Path.R`. Customise the starting path as needed, and set the path to indicate the master folder. The example code is:

```
# Starting path (CUSTOMISE PLEASE)
setwd("F:/admindata")

# Master project folder (USE AS IT IS)
setwd("./qualadmin")

# Check your current directory
getwd()
```

Please ensure to use a single forward slash / as above. R will print an error when backward slash \ is used in path. For instance,

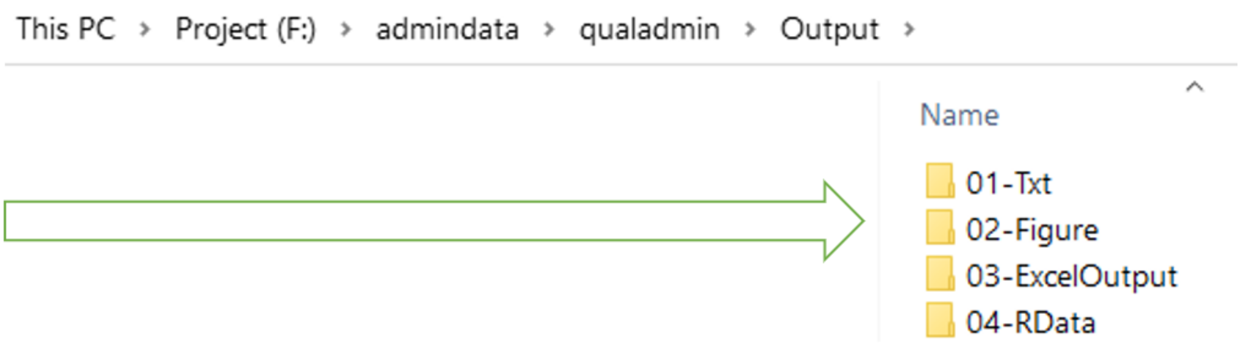
```
    setwd("F:\admindata)`
    Error: '\a' is an unrecognized escape in character string starting ""F:\a
"
```

Please ensure your working directory is set at the master project path throughout the analytical steps.

3.2.3 Automatically create output folders

The three main R script files `2_Prep_Wtsample_Freq_Table.R`, `3_A_Distance_Metrics.R`, `3_B_R-indicator.R` produce outputs. The outputs may be text, figure or in spreadsheet form. For the existing programmes to work, users need to create dedicated output folders.

To do so, please click on the `1_Create_Folders.R` file to open. Then run line by line. The resulting folder structure is provided here:



This PC › Project (F:) › admindata › qualadmin › Output ›

Name

01-Txt
02-Figure
03-ExcelOutput
04-RData

3.2.4 Install packages

The final preparation step is installing packages. Open `1_Install_Packages.R` file, and run line by line.

```
        #-----------------------------------
        # Install packages (Run once)
        #-----------------------------------

        install.packages("ggplot2")
        install.packages("tidyverse")

        install.packages("car")
```

Now, you're all set to proceed with quality measures indicators!

3.5 RUNNING 2_Prep_Wtsample_Freq_Table.R

This code file consists of two parts: generating a weighted sample data, and preparing an auxiliary file for R-indicators. If the Census data or a weighted sample data are available, users consult 3.5.1 Part 1 *A. Use the existing sample data*. For a scenario where these data are unavailable, users can generate the data as shown in 3.5.1 Part 1 *B. Generate a weighted sample data*.

3.5.1 Part 1

Open the `2_Prep_Wtsample_Freq_Table.R` file.

The top of the code file concerns checking the current directory, reading in the pre-defined functions in the R environment and loading relevant R libraries.

```
 getwd()

# Run the code file with functions.
source("Functions/1_Functions.R")

# Define output file folders, path
fn_output_folder_path()

# Disable scientific notation.
options(scipen = 999)

library("tidyverse") # data manipulation
library("ggplot2")   # visualisation
library("janitor")   # cross-tabulation
library("readxl")    # read large csv file
library("writexl")   # export to Excel
```

A. Use the existing weighted sample data

- Step 1: Read in the data.

```
df  <- read_csv("custom_wtsample.csv")

dim(df)       # obs = 1163650 (example)
glimpse(df)   # Quick glance at the data
```

Users decide which variables are to be used for tabulation. For example, users may identify the following five variables.

```
names(df)     # variable names

[1] "geog1"     "sex"       "agecode1"  "eth_code5"
[5] "econg"
```

- Step 2: Declare variables to be tabulated. Here, we declare all five variables using `var` object[2]. Users can customize the variable names here. By running the code below, R automatically saves the total number of variables, 5 in a macro called `maxvar`.

```
    var <- c("geog1", "sex", "agecode1",  # Please customise
          "eth_code5", "econg")
    maxvar <- length(var)                 # No need to customise
    maxvar
    [1] 5
```

- Step 3: Obtain frequency table of categorical variables (count of categories).

This procedure is to assess and calculate *the distribution* of categories in the weighted sample. Users can run the pre-defined function, `fn_maxvar5_freq_table()` to perform the task. The function automatically obtains counts and structure the output in long form, organised by each variable, and by its discrete category. In case of using four variables, users can use `fn_maxvar4_freq_table()` instead[3].

```
        fn_maxvar5_freq_table()
```

- Step 4: Carry out checks to see if the calculated frequency tables are accurate.

```
        freq_table[1:8, 1:9]

##   seq twdigits      n          p oneway v   by1  by2
## 1   1      101 113250 0.09732308      1 1 geog1  01
## 2   2      102 148400 0.12752976      1 1 geog1  02
## 3   3      103 137450 0.11811971      1 1 geog1  03
## 4   4      104 175100 0.15047480      1 1 geog1  04
## 5   5      105  92300 0.07931938      1 1 geog1  05
## 6   6      106 497150 0.42723327      1 1 geog1  06
## 7   7      201 565350 0.48584196      1 2   sex  01
## 8   8      202 598300 0.51415804      1 2   sex  02
```

When we printed the first 8 lines and 10 variables, we can see the count, n, and the corresponding proportion, p by each variable. The following code obtains the total observation size and confirms that the total proportion adds up to 1, for geog1 variable. Here, the total observation size can be viewed as the population size.

```
        # Check whether the total adds up to 1
        sum(freq_table[1:6, "n"])

## [1] 1163650
```

[2] As the var object is treated as global macro, the programme runs automatically using the information stored in global macro, and produces the results.

[3] We do not provide functions for the maximum variable size beyond five. In such cases, users can create their own functions by consulting the provided functions.

```
        sum(freq_table[1:6, "p"])

## [1] 1
```

- Step 5: Rename and save.
- Step 6: Export the output frequency table in Excel with the file name, **Weightedsample_freq_table.xlsx**.
- Step 7: Save the R objects as RData. Done.

B. Generate a weighted sample data

- Step 1: Load a small percentage of Census data. It is called **pop_u_short_public_release_5vars** in the provided example code.

```
        #H-----------------------------------------
        ##> 1. Load Census data
        #H----------------------------------------

        load("pop_u_short_public_release_5vars.RData")
        df  <- pop_u_short_public_release_5vars
        dim(df)    # obs = 1163659
        names(df)
```

In our example Census data, we have 1,163,659 observations with five categorical variables including geography, sex, age groups, ethnic groups, and economic activity status. One can declare which variable to tabulate. Here, we declare all five variables using var object[4].

```
 var <- c("geog1", "sex", "agecode1",
          "eth_code5", "econg")
```

The description of categories, and distribution is shown below.

Variable	Category	Description	N	(%)
Total			1163659	
geog1	1	LA codes	116128	(10.0)
	2	LA codes	150139	(12.9)
	3	LA codes	137520	(11.8)
	4	LA codes	170624	(14.7)
	5	LA codes	90873	(7.8)
	6	LA codes	498375	(42.8)
sex	1	male	564905	(48.5)

[4] As the var object is treated as global macro, the programme runs automatically using the information stored in global macro, and produces the results.

Variable	Category	Description	N	(%)
	2	female	598754	(51.5)
agecode1	1	16-20	82426	(7.1)
	2	21-25	94643	(8.1)
	3	26-30	110296	(9.5)
	4	31-35	120398	(10.3)
	5	36-40	119393	(10.3)
	6	14-45	101711	(8.7)
	7	46-50	94209	(8.1)
	8	51-55	100159	(8.6)
	9	56-60	77799	(6.7)
	10	61-65	65833	(5.7)
	11	66-70	57305	(4.9)
	12	71-75	51263	(4.4)
	13	76-80	43678	(3.8)
	14	81+	44546	(3.8)
eth_code5	1	White	1081812	(93.0)
	2	Mixed/Multiple ethnic groups	10487	(0.9)
	3	Asian/Asian British	46446	(4.0)
	3	Black/African/Caribbean/Black British	16268	(1.4)
	4	Other ethnic group	8646	(0.7)
econg	1	In employment(FT, PT)	689140	(59.2)
	2	Unemployed	27744	(2.4)
	3	Out of workforce	446775	(38.4)

Using this prior information on the population distribution (based on the Census), we can mimic the distribution in a random sample. See the next step.

- Step 2: Then, we draw a random sample 1:50.
- Step 3: From the randomly selected sample (`1163659/50 = 23273`), we then obtain frequency table of categorical variables (count of categories). Users can run the pre-defined function, `fn_maxvar5_freq_table()` to perform the task. The function[5] automatically obtains counts and structure the output in long form, organised by each variable, and by its discrete category.

```
    fn_maxvar5_freq_table()
```

[5] We also provide `fn_maxvar4_freq_table()` for users who declare four categorical variables. We do not provide functions for the maximum variable size beyond five. In such cases, users can create their own functions by consulting the provided functions.

This procedure is to assess and calculate *the distribution* of categories in a random sample (N = 23273). Based on the counts of the randomly selected sample, we multiply the counts by 50. One may wonder why we multiply. As we *reduced* the census sample by drawing a random sample by the 1:50 ratio, we need to *convert* the shrank sample back to the original size (with the priori distribution). This explains why we multiply by 50 (Weighted Sample N = 23273 * 50 = 1163650). This completes the process of generating weighted sample survey data.

- Step 4: Carry out checks to see if the calculated frequency tables are accurate.

```
        freq_table[1:8, 1:9]

##   seq twdigits raw_n      n          p oneway v   by1 by2
## 1   1      101  2265 113250 0.09732308      1 1 geog1  01
## 2   2      102  2968 148400 0.12752976      1 1 geog1  02
## 3   3      103  2749 137450 0.11811971      1 1 geog1  03
## 4   4      104  3502 175100 0.15047480      1 1 geog1  04
## 5   5      105  1846  92300 0.07931938      1 1 geog1  05
## 6   6      106  9943 497150 0.42723327      1 1 geog1  06
## 7   7      201 11307 565350 0.48584196      1 2   sex  01
## 8   8      202 11966 598300 0.51415804      1 2   sex  02
```

When we printed the first 8 lines and 10 variables, we can see the count, n, and the corresponding proportion, p by each variable. The following code obtains the total observation size and confirms that the total proportion adds up to 1, for geog1 variable. Here, the total observation size can be viewed as the population size.

```
        # Check whether the total adds up to 1
        sum(freq_table[1:6, "n"])

## [1] 1163650

        sum(freq_table[1:6, "p"])

## [1] 1
```

- Step 5: Rename and save.

- Step 6: Export the output frequency table in Excel with the file name, **Weightedsample_freq_table.xlsx**.

- Step 7: Save the R objects as RData. Done.

3.5.2 Part 2: Auxiliary file for R-indicators

From the frequency table generated by using the weighted sample, we will prepare an auxiliary file to be used for R-indicator computation procedures. From the previous steps, we identified the population size of 1,116,350 (`popsize = 1163650`) from the weighted sample.

For R-indicator calculations, we need to compute meanpop by variables which are geography, sex, age groups, ethnic groups, and economic activity status.

```
        # Compute meanpop
        auxiliary  <- freq_table %>%
                  filter(oneway == 1) %>%
                  group_by(by1) %>%
                  mutate(meanpop = n / popsize) %>%
                  ungroup()  %>%
                  dplyr::select(seq, count = n, by1,
                   v, by2, meanpop, raw_n)
```

To do so, we first remove two-way and keep the one-way frequency table only. Then, by variable-level (indicated by the variable, by1), we compute meanpop. As seen before, the count of each category is stored in n. The meanpop is obtained by dividing n by popsize. For example, the value of **meanpop** for first category of geog1 is calculated as 113250/1163650 = 0.0973, and the second category of geog1 is 148400/1163650 = 0.1275 and so on.

Finally, we add the popsize in the first row, to complete the Auxiliary file.

Let's look at the Auxiliary file:

```
        print(wtsample_auxiliary_econg)

##    seq   count       by1 v by2     meanpop raw_n     type
## 1    1 1163650     total 0  00 1.000000000     0 wtsample
## 2    2  113250     geog1 1  01 0.097323078  2265 wtsample
## 3    3  148400     geog1 1  02 0.127529756  2968 wtsample
## 4    4  137450     geog1 1  03 0.118119710  2749 wtsample
## 5    5  175100     geog1 1  04 0.150474799  3502 wtsample
## 6    6   92300     geog1 1  05 0.079319383  1846 wtsample
## 7    7  497150     geog1 1  06 0.427233275  9943 wtsample
## 8    8  565350       sex 2  01 0.485841963 11307 wtsample
## 9    9  598300       sex 2  02 0.514158037 11966 wtsample
## 10  10   84600  agecode1 3  01 0.072702273  1692 wtsample
## 11  11   93300  agecode1 3  02 0.080178748  1866 wtsample
## 12  12  111200  agecode1 3  03 0.095561380  2224 wtsample
## 13  13  124250  agecode1 3  04 0.106776092  2485 wtsample
## 14  14  118200  agecode1 3  05 0.101576935  2364 wtsample
## 15  15   99950  agecode1 3  06 0.085893525  1999 wtsample
## 16  16   95800  agecode1 3  07 0.082327160  1916 wtsample
## 17  17   95600  agecode1 3  08 0.082155287  1912 wtsample
## 18  18   78450  agecode1 3  09 0.067417179  1569 wtsample
## 19  19   67600  agecode1 3  10 0.058093069  1352 wtsample
## 20  20   57950  agecode1 3  11 0.049800198  1159 wtsample
## 21  21   50650  agecode1 3  12 0.043526834  1013 wtsample
## 22  22   42250  agecode1 3  13 0.036308168   845 wtsample
## 23  23   43850  agecode1 3  14 0.037683152   877 wtsample
## 24  24 1083250 eth_code5 4  01 0.930907060 21665 wtsample
## 25  25   10450 eth_code5 4  02 0.008980364   209 wtsample
## 26   26   45600 eth_code5 4  03 0.039187041   912 wtsample
## 27   27   16300 eth_code5 4  04 0.014007648   326 wtsample
## 28   28    8050 eth_code5 4  05 0.006917888   161 wtsample
## 29   29  691300     econg 5  01 0.594078976 13826 wtsample
## 30   30   28250     econg 5  02 0.024277059   565 wtsample
## 31   31  444100     econg 5  03 0.381643965  8882 wtsample
```

3.6 RUNNING 3A_Distance_Metrics.R

To calculate distance metrics, we first obtain one-way, and two-way frequency tables of categorical variables, and calculates proportions of each sub-category by each data source. Next, we combine master (benchmark) and admin freq tables. Then, we create domains for quality indicators, and finally compute distance metrics as part of quality indicators. We offer three different types of distance metrics. To allow comparison across the metrics, we standardise the calculations.

Users can also produce a summary table of three types of distance metrics, and visualise the results.

The preliminary step is to ensure we have benchmark data. Simply *source* the previous file as below to update it.

```
        source("2_Prep_Wtsample_Freq_Table.R")
```

Step 1: Read admin data

```
        df  <- read_csv("public_release_admin.csv")

## Rows: 1033664 Columns: 6
## -- Column specification --------------------------------------------------
------
## Delimiter: ","
## dbl (6): person_id, geog1a, sex, agecode1, eth_code5, econg
## 
## i Use `spec()` to retrieve the full column specification for this data.
## i Specify the column types or set `show_col_types = FALSE` to quiet this m
essage.

        tail(df)

## # A tibble: 6 x 6
##   person_id geog1a   sex agecode1 eth_code5 econg
##       <dbl>  <dbl> <dbl>    <dbl>     <dbl> <dbl>
## 1   1033659      1     1        6         1     1
## 2   1033660      6     1        6         1     1
## 3   1033661      6     2        5         1     1
## 4   1033662      5     1       12         1     3
## 5   1033663      1     2        9         1     1
## 6   1033664      6     2        9         1     1
```

The last six observations of the example admin data are shown above. As the person id goes up to 1033664, we can see there are 1033664 observations in admin data. We also notice that geog1a is used instead of *geog1*. As such, we declare all five variables for tabulations.

```
  var <- c("geog1a", "sex", "agecode1",
            "eth_code5", "econg")
```

Step 2: Obtain admin freq tables.

As mentioned in the previous section, we obtain frequency table of categorical variables (count of categories). Users can run the pre-defined function, `fn_maxvar5_freq_table()` to perform the task. The function[6] automatically obtains counts and structure the output in long form, organised by each variable, and by its discrete category.

Once the frequency tables are obtained, we rename the object as `Admin_f_table_one`. Let's inspect `Admin_f_table_one`.

```
        head(Admin_f_table_one)

##   seq twdigits admin_n admin_perc oneway v    by1 by2    by3 by4 by5
## 1   1      101  137993  0.1334989      1 1 geog1a  01 oneway   0   0
## 2   2      102  124051  0.1200110      1 1 geog1a  02 oneway   0   0
## 3   3      103  131176  0.1269039      1 1 geog1a  03 oneway   0   0
## 4   4      104  139867  0.1353119      1 1 geog1a  04 oneway   0   0
## 5   5      105  142304  0.1376695      1 1 geog1a  05 oneway   0   0
## 6   6      106  358273  0.3466049      1 1 geog1a  06 oneway   0   0

       tail(Admin_f_table_one)

##     seq twdigits admin_n   admin_perc oneway v       by1 by2   by3 by4 by5
## 340 340   404501    8557 0.0082783187      2 4 eth_code5  04 econg   5  01
## 341 341   404502     791 0.0007652390      2 4 eth_code5  04 econg   5  02
## 342 342   404503    5007 0.0048439338      2 4 eth_code5  04 econg   5  03
## 343 343   405501    3867 0.0037410609      2 4 eth_code5  05 econg   5  01
## 344 344   405502     255 0.0002466953      2 4 eth_code5  05 econg   5  02
## 345 345   405503    3420 0.0033086187      2 4 eth_code5  05 econg   5  03
```

Step 3: Merge admin + Weighted sample freq tables

```
       fn_merge_one_admin_wtsample_f_table_temp()
```

The code above merges two data sources.

Step 4: Create domains of quality indicators

```
        fn_create_domain_temp()
```

[6] We also provide `fn_maxvar4_freq_table()` for users who wish to declare four categorical variables.

To check the domains, we can use *Janitor* package's `tabyl` function[7]. The function creates 15 domains, including five single variables' domain, and ten bivariate domains.

```
        display_domain %>% tabyl(fct_domain)

##          fct_domain  n     percent
##               geog1  6 0.017391304
##                 sex  2 0.005797101
##            agecode1 14 0.040579710
##           eth_code5  5 0.014492754
##               econg  3 0.008695652
##           geog1:sex 12 0.034782609
##      geog1:agecode1 84 0.243478261
##     geog1:eth_code5 30 0.086956522
##         geog1:econg 18 0.052173913
##        sex:agecode1 28 0.081159420
##       sex:eth_code5 10 0.028985507
##           sex:econg  6 0.017391304
##  agecode1:eth_code5 70 0.202898551
##      agecode1:econg 42 0.121739130
##     eth_code5:econg 15 0.043478261
```

Step 5: Compute distance metrics

Run the functions to compute three types of distance metrics.

```
        fn_unstd_distance_metrics_full()
        fn_unstd_distance_metrics_tidy()
```

Step 6: Standardise distance metrics

Step 7: Reshape, then tidy

From wide form, the outputs have been reshaped to long form. Then, we keep standardised solutions. The results are as follows:

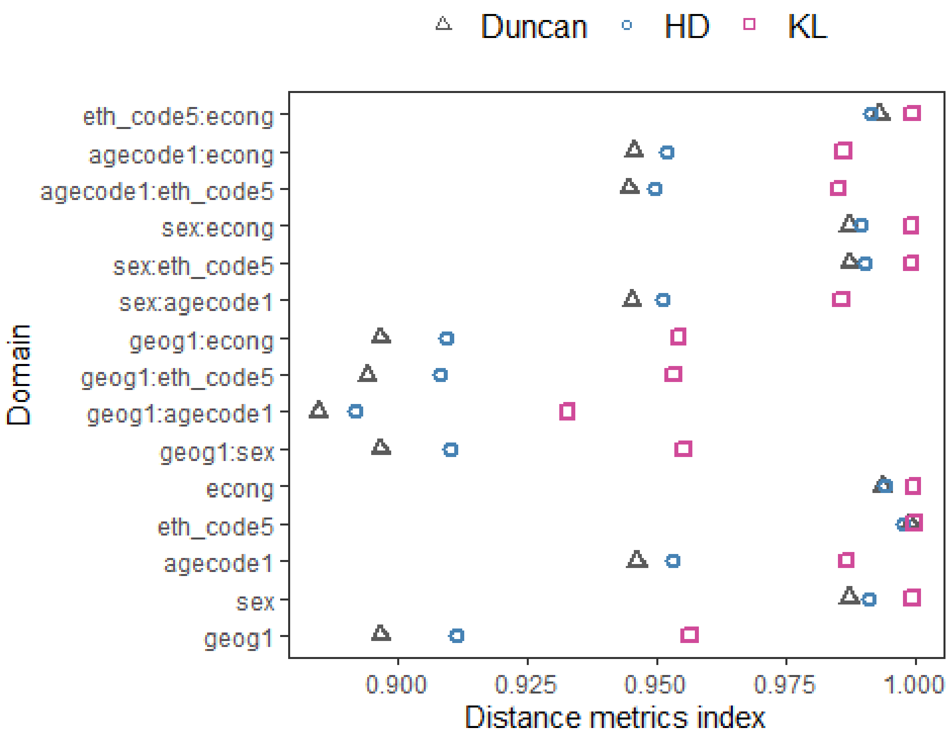
```
      df <- distance_metrics_long
      # std_test(1-Duncan, 1-HD, 1-KL) only
      df <- df %>% filter(std_test_use == 1)

      df[1:9, c(1:2, 4:5, 9)]

## # A tibble: 9 x 5
##   domain_id domain   indicator  index std_test_use
##       <int> <chr>    <chr>      <dbl>        <dbl>
## 1         1 geog1    Std_Duncan 0.897            1
```

[7] This is essentially almost identical to `table(display_domain$fct_domain)`, but the approach by `tabyl` produces percent by default.

```
## 2          1 geog1    Std_t_HD   0.911          1
## 3          1 geog1    Std_t_KL   0.957          1
## 4          2 sex      Std_Duncan 0.987          1
## 5          2 sex      Std_t_HD   0.991          1
## 6          2 sex      Std_t_KL   1.00           1
## 7          3 agecode1 Std_Duncan 0.946          1
## 8          3 agecode1 Std_t_HD   0.953          1
## 9          3 agecode1 Std_t_KL   0.987          1
```

Step 8: Visualise the distance metrics

```
         plot(p)
```



3.7 RUNNING 3B_R-indicator.R

The file computes the overall R-indicator. Users can also proceed with computing **partial** R-indicators by category level, and variable level. The procedure can be computationally extensive. This is noted in the relevant section, so that users can allow some time to execute the code.

On this example, we will prepare the sample and population distributions and calculate an R-indicator for 5 variables: geog1a (6), sex (2), agecode1 (14) and eth_cod5 (5) and econg (3). (Note that in our example dataset, geog1a is from the administrative data, whereas geog1 is in the benchmark data. Users may have consistent variable names).

As part of administrative data preparation, each of the variables should have their categories numbered 1,2,3…. We will use these numbers instead of the original names of the categories because it will enable us to do loops through the data. This is to facilitate building design matrix using dummy variables.

Along with administrative data, we also need benchmark data from Census. Assuming users have no access to Census data, we replace Census with weighted sample counts. The auxiliary data file contains these weighted sample counts.

As such, we will read in both administrative and auxiliary data files separately and compute R-indicators using matrix syntax in R. Due to the complexity of the procedure, we provide defined functions that users can execute with ease in practice. Users can consult `Functions/2_Functions_R-indicators.R` file for more details on the algorithm and operationalisation.

Step 1: Prepare benchmark data

First, load data from the auxiliary data file. The auxiliary file can be derived from the Census, but we use benchmark data in place of Census. It contains the proportions of categories for five variables.

```
load(file = "Output/04-RData/auxiliary.RData")
auxiliary[1:10, ]

##    seq   count      by1 v by2    meanpop raw_n     type
## 1    1 1163650    total 0  00 1.00000000     0 wtsample
## 2    2  113250    geog1 1  01 0.09732308  2265 wtsample
## 3    3  148400    geog1 1  02 0.12752976  2968 wtsample
## 4    4  137450    geog1 1  03 0.11811971  2749 wtsample
## 5    5  175100    geog1 1  04 0.15047480  3502 wtsample
## 6    6   92300    geog1 1  05 0.07931938  1846 wtsample
## 7    7  497150    geog1 1  06 0.42723327  9943 wtsample
## 8    8  565350      sex 2  01 0.48584196 11307 wtsample
## 9    9  598300      sex 2  02 0.51415804 11966 wtsample
## 10  10   84600 agecode1 3  01 0.07270227  1692 wtsample
```

We can see the count of each variable (by1) and the sub-category (by2). The population size is indicated as `1163560` (shown in the first row, under `count` column). We will save this `popsize` as an object to use as a macro variable in programming later.

```
# Obtain popsize from population (Census or benchmark data)
# To be used in programming later
popsize <- auxiliary %>%
           filter( seq == 1) %>%
           summarise(count)

popsize <- popsize[[1]]
popsize
```

```
## [1] 1163650
```

Now, we prepare the data to build design matrix using dummy variables. To do so, from the total number of categories for each variable, we need to remove the **last** category of each categorical variable (`group_by(by1)`). The last category is defined by `max(by2)`and removed accordingly. The first row (`seq == 1`) is not subject to this procedure and kept in the data. As shown in the code below, we keep rows if `lastcat == 0`, while filtering out cases which are the last category. We then drop the indicator for the last category (`dplyr::select(-c(lastcat_, lastcat))`). Here, we explicitly instruct R to use `dplyr` package to access `select` function[8].

```
        col_auxiliary <- auxiliary %>%
             group_by(by1) %>%
             mutate(
              lastcat_ = ifelse(by2 == max(by2), 1, 0),
              lastcat = ifelse(seq == 1, 0, lastcat_)
            ) %>%
             filter(lastcat == 0) %>%
             dplyr::select(-c(lastcat_, lastcat)) %>%
             ungroup()
```

Notice that from 5 variables, geog1a (6), sex (2), agecode1 (14) and eth_cod5 (5) and econg (3), we now have 1+(nvar-1) for each variable so here it is 1+5+1+13+4+3=26 rows. We create a macro, `numcat` to store this information on the total row.

```
nrow(col_auxiliary)

## [1] 26

numcat <- nrow(col_auxiliary)
numcat

## [1] 26
```

To inspect the setup on dummy variables (to be created later), let's produce a cross-tabulation by by1 and by2.

```
  dummychk <- col_auxiliary
  dummychk %>% tabyl(by1, by2)

            00 01 02 03 04 05 06 07 08 09 10 11 12 13
  agecode1   0  1  1  1  1  1  1  1  1  1  1  1  1  1
  econg      0  1  1  0  0  0  0  0  0  0  0  0  0  0
  eth_code5  0  1  1  1  1  0  0  0  0  0  0  0  0  0
  geog1      0  1  1  1  1  1  0  0  0  0  0  0  0  0
```

[8] This is to avoid warning messages from R when R searches for a particular function from two different packages.

```
  sex         0  1  0  0  0  0  0  0  0  0  0  0  0  0
  total       1  0  0  0  0  0  0  0  0  0  0  0  0  0
```

All look good. At this stage, we keep meanpop only, dropping other columns, and print meanpop in a single column (column vector). This column will be transposed before we save it as row vectors. Before reshaping the data, we carefully inspect the values of meanpop.

```
        # Print meanpop
        print(col_auxiliary[1:nrow(col_auxiliary), "meanpop"],
            n = nrow(col_auxiliary))

## # A tibble: 26 x 1
##    meanpop
##      <dbl>
##  1 1
##  2 0.0973
##  3 0.128
##  4 0.118
##  5 0.150
##  6 0.0793
##  7 0.486
##  8 0.0727
##  9 0.0802
## 10 0.0956
## 11 0.107
## 12 0.102
## 13 0.0859
## 14 0.0823
## 15 0.0822
## 16 0.0674
## 17 0.0581
## 18 0.0498
## 19 0.0435
## 20 0.0363
## 21 0.931
## 22 0.00898
## 23 0.0392
## 24 0.0140
## 25 0.594
## 26 0.0243
```

Step 2: Reshape and save benchmark data as row vectors

We use t(col_auxiliary) to transpose and tidy the reshaped data. We then generate ttt as an indicator of benchmark data. We use this indicator for merging with the administrative data later. Save the data as popmean_row_vector.

```
# Transpose to arrange in row vector format.
  temp <- t(col_auxiliary)
  temp <- as.data.frame(temp)
  row.names(temp) <- 1:nrow(temp)

# generate merge id, ttt.
  temp$ttt  <- 0

# Rename variables "popmean1- popmean26"
  names(temp) <- c(paste0("popmean", 1:numcat), "ttt")

# SAVE
  popmean_row_vector <- temp
```

Let's see the row vector names, "popmean1- popmean26" and `ttt`.

```
    names(popmean_temp)

##  [1] "popmean1"  "popmean2"  "popmean3"  "popmean4"  "popmean5"  "popmean6
"
##  [7] "popmean7"  "popmean8"  "popmean9"  "popmean10" "popmean11" "popmean1
2"
## [13] "popmean13" "popmean14" "popmean15" "popmean16" "popmean17" "popmean1
8"
## [19] "popmean19" "popmean20" "popmean21" "popmean22" "popmean23" "popmean2
4"
## [25] "popmean25" "popmean26" "ttt"

    popmean_temp[, 1:5]

##   popmean1   popmean2  popmean3  popmean4  popmean5
## 1        1 0.09732308 0.1275298 0.1181197 0.1504748
```

Now, our benchmark data preparation is completed.

Step 3: Declare variables in administrative data

The next step is to declare variables to be used for computing the R-indicator. These variable names should be from the administrative data.

Users can define their own variables and save as a macro, `var`. For instance, users may use four variables and define as below.

var <- c("geog1a", "sex", "agecode1", "econg")

For demonstration, we use five variables for R-indicator calculations and declared as such.

```
# Customise as needed.
var <- c("geog1a", "sex", "agecode1",
          "eth_code5", "econg")
var

## [1] "geog1a"    "sex"       "agecode1"  "eth_code5" "econg"

# End of custom variables.
```

This concludes steps 1 through 3. Users can customise some setups up to this point.

Step 4: Define macro variables

Along with the defined variables, we need to ensure macro variables are generated correctly. These macro variables are not designed to customise and designed to run without altering the code.

var

variablenum

maxvar

popsize

The total number of variables and the total number of categories of each categorical variables will be stored in a macro called `variablenum` and `maxvar`. We use `var` macro, as we defined earlier, to derive these two macros.

```
variablenum <- length(var) # No need to customise
maxvar <- length(var)      # No need to customise

variablenum ; maxvar

## [1] 5

## [1] 5
```

Earlier, we also generated `popsize` to feed the information on the (total observation) size of the the benchmark data to the programme. The remaining macros, such as respop, piinv and rrate, will be automatically generated by the pre-defined functions.

respop

piinv

rrate

`respop` is the (sample) size of the administrative data. `piinv` refers to the inverse pi. `rrate` is computed by `resppop/popsize`.

Step 5: Compute R-indicators

Once steps 1 through 4 are completed, we are set to carry out computing R-indicators. These procedures are automated via pre-defined functions using complex matrix and data management syntax. Please note that some procedures are computationally intensive.

We first open the corresponding administrative data, and notice the number of rows is 1,033,664.

```
        aa  <- read_csv("public_release_admin.csv")
```

```
## Rows: 1033664 Columns: 6
## -- Column specification --------------------------------------------------
------
## Delimiter: ","
## dbl (6): person_id, geog1a, sex, agecode1, eth_code5, econg
##
## i Use `spec()` to retrieve the full column specification for this data.
## i Specify the column types or set `show_col_types = FALSE` to quiet this m
essage.

        nrow(aa) # 1033664

## [1] 1033664
```

We need several empty objects to hold data to get the functions to work as intended. Note that we have five functions that permit us to compute overall R-indicators, along with additional functions for obtaining partial R-indicators. Each function will generate objects including vvv, pop_respmean, des_pop_respmean, gh, and R_indicators as indicated in the function name. The partial R-indicators can be found in the data object, partial.

```
        df      <- NULL
        between <- NULL
        partial <- NULL
        partialtemp <-  NULL
        fn_r_indicator_1_vvv()
        fn_r_indicator_2_pop_respmean()
        fn_r_indicator_3_des_pop_respmean()
        fn_r_indicator_4_gh()
        fn_R_indicators()

        # Partial R-indicators
        fn_r_indicator_partialtemp()
        fn_r_indicator_domain_order_partial()
```

Let’s go over one function at a time.

> The utility of fn_overall_r_indicator_1_vvv() is to build design matrix. Starting with ensuring that the admin data only contains the declared variables and factorise them, the function defines macro variables, resppop and rrate using the data object, aa. To prepare for design matrix, the code creates dummy variables, using **fastDummies** R library package. Once all dummy variables generated, we need to remove the last category. As such, the function detects the categories of each variable and drops the last category. The details of this step is discussed above.

Let’s see the vvv object, which contains the design matrix with weights. The weights are calculated by the inverse of *rrate* (finalwgt = 1/rrate). The last five columns are printed below.

```
   # fn_overall_r_indicator_1_vvv()
```

```
      from <- ncol(vvv)-4
      vvv[1:8, from: ncol(vvv)]

## # A tibble: 10 x 5
##    des24 des25 des26 finalwgt piinv
##    <int> <int> <int>    <dbl> <dbl>
##  1     0     1     0     1.13     1
##  2     0     1     0     1.13     1
##  3     0     1     0     1.13     1
##  4     0     1     0     1.13     1
##  5     0     1     0     1.13     1
##  6     0     0     0     1.13     1
##  7     0     0     0     1.13     1
##  8     0     1     0     1.13     1
```

The fn_overall_r_indicator_2_pop_respmean() prepares distributions of the administrative data. Using the data object,vvv, we obtain weighted sample counts and produces a row vector called *respmean_row_vector*. Merging the corresponding popmean_row_vector from the benchmark data we prepared at the step 2 earlier, the function combines both mean vectors from the two data sources.

The combined mean vectors are stored at the pop_respmean data object. We will inspect the last five columns.

```
   # fn_overall_r_indicator_2_pop_respmean()

    from <- ncol(pop_respmean)-4
    pop_respmean[, from: ncol(pop_respmean)]

##   respmean25 respmean26 finalwgt piinv ttt
## 1  0.5896214 0.02252997 1.125753     1   0
```

The function, fn_overall_r_indicator_3_des_pop_respmean(), allows us to combine the design matrix with the pop_respmean. We store the data at des_pop_respmean.

```
   # fn_overall_r_indicator_3_des_pop_respmean()

    names(des_pop_respmean)

##  [1] "des1"          "des2"          "des3"          "des4"         
##  [5] "des5"          "des6"          "des7"          "des8"         
##  [9] "des9"          "des10"         "des11"         "des12"        
## [13] "des13"         "des14"         "des15"         "des16"        
## [17] "des17"         "des18"         "des19"         "des20"        
## [21] "des21"         "des22"         "des23"         "des24"        
## [25] "des25"         "des26"         "finalwgt"      "piinv"        
## [29] "popmean1"      "popmean2"      "popmean3"      "popmean4"     
## [33] "popmean5"      "popmean6"      "popmean7"      "popmean8"     
## [37] "popmean9"      "popmean10"     "popmean11"     "popmean12"    
## [41] "popmean13"     "popmean14"     "popmean15"     "popmean16"
## [45] "popmean17"     "popmean18"     "popmean19"     "popmean20"
## [49] "popmean21"     "popmean22"     "popmean23"     "popmean24"
## [53] "popmean25"     "popmean26"     "respmean1"     "respmean2"
## [57] "respmean3"     "respmean4"     "respmean5"     "respmean6"
## [61] "respmean7"     "respmean8"     "respmean9"     "respmean10"
## [65] "respmean11"    "respmean12"    "respmean13"    "respmean14"
## [69] "respmean15"    "respmean16"    "respmean17"    "respmean18"
## [73] "respmean19"    "respmean20"    "respmean21"    "respmean22"
## [77] "respmean23"    "respmean24"    "respmean25"    "respmean26"
## [81] "seq"           "responsesamp1"
```

```
    from <- ncol(des_pop_respmean)-4
    des_pop_respmean[1:6, from: ncol(des_pop_respmean)]
```

```
## # A tibble: 6 x 5
##   respmean24 respmean25 respmean26   seq responsesamp1
##        <dbl>      <dbl>      <dbl> <int>         <dbl>
## 1     0.0139      0.590     0.0225     1             1
## 2     0.0139      0.590     0.0225     2             1
## 3     0.0139      0.590     0.0225     3             1
## 4     0.0139      0.590     0.0225     4             1
## 5     0.0139      0.590     0.0225     5             1
## 6     0.0139      0.590     0.0225     6             1
```

In the `fn_overall_r_indicator_4_gh()`, we compute the difference from the mean vectors and the weight variables. Exerpts of the code from the function below show that the differences are stored in `rsam` and `psam`. By adding *rsam* and *psam* to the intermediate data object, des_pop_respmean, we obtain the gh data. This concludes the pre-matrix preparation part.

```
  # use df for programming.
  df <<-  data.frame(des_pop_respmean)

  # Prep for loop.
  des_col      <<-  c(paste0("des"     , 1:numcat))
  respmean_col <<-  c(paste0("respmean", 1:numcat))
  popmean_col  <<-  c(paste0("popmean" , 1:numcat))

  des      <<-  df[, des_col]
  respmean <<-  df[, respmean_col]
  popmean  <<-  df[, popmean_col]

  rsam <<-  des - respmean
  psam <<-  des - popmean
  temp <<-  data.frame(rsam, psam)

  # Rename variables
  colnames(temp)  <<-  c(paste0("rsam", 1:numcat),
                     paste0("psam", 1:numcat))
  # Combine
  gh <<-  cbind(des_pop_respmean, temp)
```

The `fn_R_indicators()` uses matrix syntax to calculate propensity scores, prior to computing R-indicators. We calculate two kinds of propensity scores – one that used only population information (roipop, or prop_pop) and the other which used a mixture of the response data and the population information (roimix, or prop_mix).

In terms of partial R-indicators, we demonstrate using *prop_mix*. Running `fn_r_indicator_partialtemp()` users can yield partial R-indicators at the variable level and at the category level. The function, `fn_r_indicator_domain_order_partial()` helps us to organise domains.

```
    fn_r_indicator_partialtemp()
    fn_r_indicator_domain_order_partial()
    # View(partial)
```

Step 6: Save in Excel and inspect

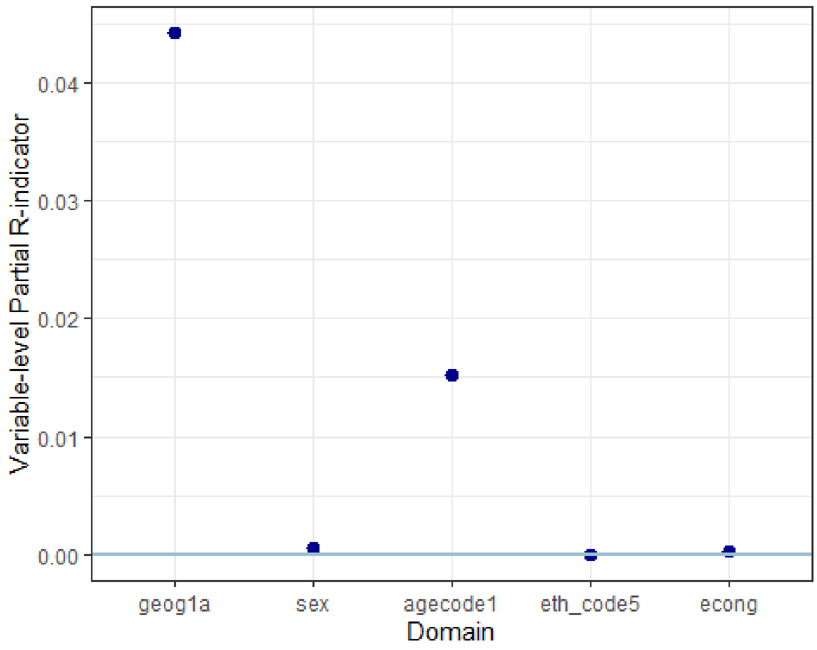
At this stage, users can inspect the output accordingly. Let's have a look. Here, we can see the overall R-indicator is estimated as 0.496 based the administrative data (N=1033664). Looking at the variable-level R-indicator (see rows 4-8), geog1a was seen to have the greatest R-indicator (0.04) compared to econg (0.0002).

```
    partial[1:17, c(1:2, 4, 8:10)]

##    seq     domain         R_indicator  count n_cat domain_n
## 1    1    Overall       0.4960263148     NA  <NA>       NA
## 2    2  mrphatall       0.9597768609     NA  <NA>       NA
## 3    3    resppop 1033664.0000000000     NA  <NA>       NA
## 4    4     geog1a       0.0442418275     NA  <NA>       NA
## 5    5        sex       0.0004983014     NA  <NA>       NA
## 6    6   agecode1       0.0152841196     NA  <NA>       NA
## 7    7  eth_code5       0.0000296805     NA  <NA>       NA
## 8    8      econg       0.0002117090     NA  <NA>       NA
## 9    9       des1                 NA      0     1        0
## 10  10   geog1a_1       0.0879501347 137993     1        1
## 11  11   geog1a_2      -0.0192794816 124051     2        1
## 12  12   geog1a_3       0.0219039079 131176     3        1
## 13  13   geog1a_4      -0.0366161292 139867     4        1
## 14  14   geog1a_5       0.1396946734 142304     5        1
## 15  15   geog1a_6      -0.1216543425 358273     6        1
## 16  16      sex_1      -0.0162005503 489228     1        2
## 17  17      sex_2       0.0153571986 544436     2        2
```

Step 7: Visualising using scatterplots

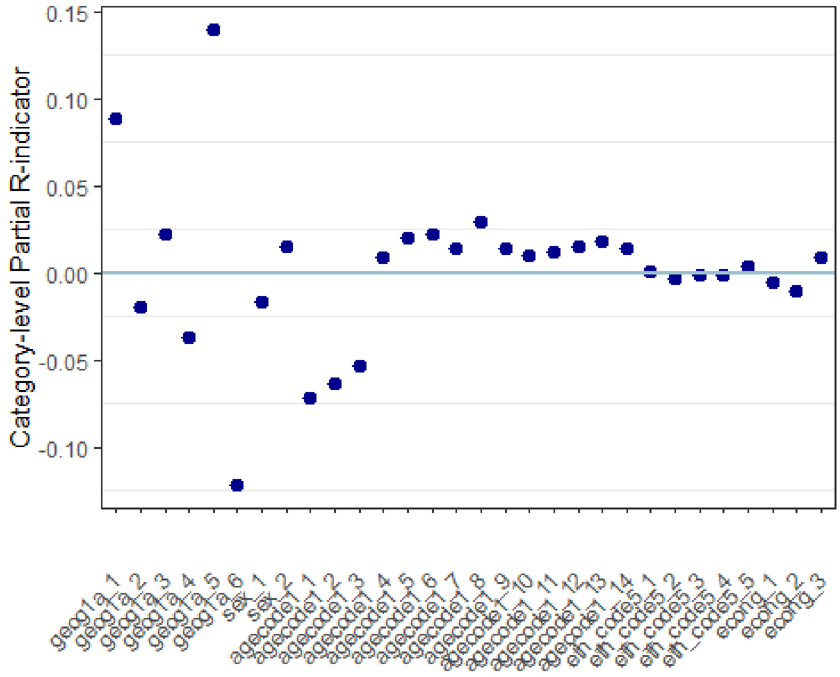
The visualisation of R-indicator by the variable level is shown as an example:

```
plot(p1)
```



And for R-indicator by the category-level:

```
plot(p2)
```



This concludes the manual. Thank you for taking the time reading the material. Please get in touch with any query or errata at fanfurcada@gmail.com.

If you need technical support, please consult the following Troubleshooting Q & A section.

4. Troubleshooting Questions and Answers

4.1 Questions and Answers

How do I know where to customise the code to suit my needs?

Unless indicated as “Customise as needed”, users can run the code as it is. Please consult each code file.

How to use Starting path in multiple machines?

If users plan to use different machines, simply by changing the “starting path”, users can carry out the analysis with minimal disruption. To achieve this, please ensure to use the consistent master project folder name.

What are the commonly used commands?

Most commonly used commands in the tidyverse package are:

```
arrange : sort variables.
bind_rows: append multiple dataframes.
mutate   : manipulate variables, and
           create new variables based on old variables.
select   : order, and keep(drop) variables of interest.
shell.exec: launch a software and opens the target file (Windows PC only)
```

How to free up memory space and speed up RStudio?

You can remove objects that you no longer need.

```
# To remove objects except for certain objects
ls()

keepobjectslist <- c("a", "b", "c")
rm(list = ls()[!ls() %in% keepobjectslist])
ls()
```

I get error messages when a pre-defined function is used.

Users can inspect the codes used in the function, and identify the issues. It is recommended NOT edit the function file directly, as the functions are used repeatedly, and the interlinked sections may not run as expected. Where preferable, users may copy the codes in the function, and use locally with minor tweaks.

How do I modify pre-defined functions?

Users can modify `1_Functions` and `2_Functions_R-indicators.R` under *Functions* folder.

```
# 1_Functions.R
fn_output_folder_path <- function() {

  currentdate <<- Sys.Date()
  txtpath   <<- "Output/01-Txt/"
  figpath   <<- "Output/02-Figure/"
  xlsxpath  <<- "./Output/03-ExcelOutput/"
  Rdatapath <<- "Output/04-RData/"
}
```

We can check how the output folder names are set as path to save the results during the analytical process.

```
fn_output_folder_path()
```

Let's run the function. We can see that xlsxpath is set as `"./Output/03-ExcelOutput/"`.

```
xlsxpath

## [1] "./Output/03-ExcelOutput/"
```

Let's customise the xlsxpath, by renaming the folder name. If we customise `1_Functions.R` file, we can edit the information enclosed in the brackets. Notice that we use `<<-` with functions so that the object created by a function will exist in the global R environment. This is very important.

Alternatively, we could ignore the pre-defined function and just write relevant lines of code and keep it in the main R script file. For instance, we could put output_folder_path at the top of the `2_Prep_Wtsample_Freq_Table.R`. Here, we edited the `xlsxpath`. Notice that `fn_output_folder_path <- function() {   }` is removed.

```
    xlsxpath_2 <- "./Output/03-Excel/"

    xlsxpath_2

## [1] "./Output/03-Excel/"

    #H--------------------------------------
    ## > Step 1. Load Census data
    #H-------------------------------------
    # load("pop_u_short_before_sim_5vars.RData")
```

Notice that we use `<-`. Using `<<-` is not necessary here. Users can remember the usage of `<-` and can modify the functions as appropriate, should the function incur errors.

What approaches are taken in programming?

When loop is used, base R functions were used (table, tapply, etc). For data manipulation, tidyverse package was used extensively. This strategy is partly to improve readability of the code.

To enhance users’ workflow, output files are programmed to launch using the pre-defined functions.

Can I ignore Warning messages?

Some packages alert users with compatibility issues arising from old version. These can be ignored. For example,

```
  library("fastDummies")
  Warning message:
  package 'fastDummies' was built under
  R version 4.1.2

  library(rlist)
  Warning message:
  package 'rlist' was built under R version 4.1.2
```

What version of R is used?

Tested with Windows PC. R version used: 4.1.1 RStudio version: RStudio 2022.12.0 Build 353.

4.2 Troubleshooting

Unused argument error

For example, `sim %>% select(geog1a)` the select command can cause an error:

```
Error in select(., geog1a) :
  unused arguments (geog1a)
```

This maybe due to the conflict in packages.

The error can be fixed by adding the name of the package used, dplyr, explicitly. `sim %>% dplyr::select(geog1a)`

I get errors when computing…

Please inspect zero cells, and ensure 0 (numeric value) is entered for n and perc, as well as admin_n and admin_perc. Errors may occur with NA coding and data attributes(character, factor, numeric).

I am experiencing slowness in computation.

R can be not responsive if memory is full. Please identify bottlenecks and remove them. It may be due to certain commands. For example, `View(object)` command could take a while if the object is huge in size. Unless one should inspect the data, suppress the View command to expedite the computation where possible.

It can also be the case that for loop functions can be slow as well. In some instances, removing objects may help as this procedure can free up memory space. See above commonly used commands for more information.

Error: cannot allocate vector of size xxxx.x Gb

If matrix symbols have entered mistakenly, R shows an error message like this. Please double check whether there are any mistakes. For instance, one may have typed `a*b` instead of `a%*%b`. Users can type `memory.limit()` to check the current memory limit and increase as necessary.

References

Bianchi, Annamaria, Natalie Shlomo, Barry Schouten, Damião N. Da Silva, and Chris Skinner. 'Estimation of Response Propensities and Indicators of Representative Response Using Population-Level Information'. Survey Methodology 45 (2) (2019): 217–47.

Duncan, Otis Dudley, and Beverly Duncan. 'A Methodological Analysis of Segregation Indexes'. American Sociological Review 20, no. 2 (1955): 210–17. https://doi.org/10.2307/2088328.

R Core Team (2022). R: A language and environment for statistical computing. R Foundation for Statistical Computing, Vienna, Austria. URL https://www.R-project.org/.

Schouten, Barry, Fannie Cobben, Jelke Bethlehem. 'Indicators for the Representativeness of Survey Response.. Survey Methodology 35(1) (2009): 101 – 113.

Schouten, Barry, and Natalie Shlomo. 'Selecting Adaptive Survey Design Strata with Partial R-indicators'. International Statistical Review 85(1) (2017): 143-163.

Citation

Please cite this work as:

Kim, Sook & Shlomo, Natalie (2023). "Quality Indicators for Administrative Data User Manual For R Package on GitHub", available at https://github.com/sook-tusk/qualadmin