Complementary Data Suppression in R

Code, Explanation, and Example Implementation

Complementary data suppression is a technique used to protect data privacy in datasets that include small cell counts. For a full explanation of CDS, see the Connecticut State Department of Education's document on methodologies and rules: Data Suppression Guidelines.

Using CSDE's example cutoff values and a dummy dataset, this document walks its reader through the process of carrying out CDS as part of an R pipeline. More information on assumed data format and CDS implementation is detailed throughout the various steps.

If you would like to directly access the fully-commented R script that forms the basis for this narrative explanation, *click here*.

1. Load your libraries, prepare your data, and define your values.

In order to perform any data manipulation, we need to load our relevant libraries.

```
library(tidyverse) # Basic data manipulation
library(here) # NOT NECESSARY for user, needed for loading data into markdown
library(kableExtra)  # For pretty printing
```

For the sake of this explanation, we will be using a dummy dataset. This script assumes that the user's data is in wide format. If that is not the case, you can use `pivot_wider` to easily make the necessary change, as detailed in the code.

```
here() # NOT NECESSARY for user
```

```
[1] "C:/Users/think/OneDrive/Documents/throwaway_repo"
```

```
# Reading in data
tabdata <- read_csv('burner_tabdata.csv')
# Printing to see structure
tabdata |>
  kbl(caption = "Initial Data") |>
  kable_classic(full_width = F, html_font = "Cambria")
```

Initial Data

school	county	hispanic_count	white_count	black_count
abc	tolland	0	50	2
def	tolland	3	80	0
ghi	windham	10	20	23

school	county	hispanic_count	white_count	black_count
jkl	windham	15	15	12
mno	windham	8	34	3
pqr	avon	5	23	5
stu	avon	6	4	6
vwx	hartford	20	0	23
yz	fairfield	30	3	2

Following CSDE guidelines, we will be using a cell count cutoff value of 6 (cells with values equal to five or less will be suppressed). In calculating ratios, we will suppress values with numerators equal to or less than five or denominators equal to or less than 20. These ratios should be calculated and suppressed before the cell count values are examined. We will also define our suppression character to be "*". Because "*" is a special regular expression character (string detection language), we will account for that here.

```
# Suppression values of choice
cell_bound <- 5
numerator_bound <- 5
denominator_bound <- 20

# Suppression character to be used
supp_char <- '*'
# Define regex version of supp_char in case it is a special character
regex_suppchar <- if_else(
  supp_char %in% c('*', '.'),
  paste0('\\', supp_char), # Escape slashes for literal character interpretation
  supp_char)
```

2. Calculate your aggregate fields and check for suppression.

Is your data in long format? If so, now is the time to pivot to wide. Use `pivot_wider`! Because our data is already in wide format, we won't use this. If you need to, uncomment this line of code and fill in the blanks. Check out the documentation and examples if this is new to you.

```
# tabdata <- pivot_wider(tabdata, names_from = name_column, values_from = value_column)
```

Next, we will calculate our ratios and aggregate fields. For this example, we are calculating count totals and ratios.

```
# Carry out suppression of percentages and aggregates before individual numbers
tabdata <- tabdata |>
  mutate(
    total = hispanic_count + white_count + black_count, # Total counts
    hispanic_perc = if_else(
      # Suppression criteria:
      (hispanic_count < numerator_bound) | (total < denominator_bound),
```

```
      # If not met, suppress value:
      supp_char,
      # Else, store value as string.
      as.character(round(hispanic_count / total, 2))), # Rounding for appearance
    black_perc = if_else(
      (black_count < numerator_bound) | (total < denominator_bound),
      supp_char,
      as.character(round(black_count / total, 2))),
    white_perc = if_else(
      (white_count < numerator_bound) | (total < denominator_bound),
      supp_char,
      as.character(round(white_count / total, 2))))
# Current data structure and suppression:
tabdata |>
  kbl(caption = "Initial Data") |>
  kable_classic(full_width = F, html_font = "Cambria")
```

Initial Data

school	county	hispanic_count	white_count	black_count	total	hispanic_perc	black_perc	white_perc
abc	tolland	0	50	2	52	*	*	0.96
def	tolland	3	80	0	83	*	*	0.96
ghi	windham	10	20	23	53	0.19	0.43	0.38
jkl	windham	15	15	12	42	0.36	0.29	0.36
mno	windham	8	34	3	45	0.18	*	0.76
pqr	avon	5	23	5	33	0.15	0.15	0.7
stu	avon	6	4	6	16	*	*	*
vwx	hartford	20	0	23	43	0.47	0.53	*
yz	fairfield	30	3	2	35	0.86	*	*

3. Define your suppression columns.

These are the columns containing the counts we want to apply complementary suppression to. In the case of this example, we are suppressing `white_count`, `black_count`, and `white_count`. We already carried out suppression on our aggregate columns, so those aren't included here.

```
# Columns (COUNTS) to apply complementary suppresion to
supp_cols <- c('white_count', 'black_count', 'hispanic_count')
# Indices of suppression columns, for later manipulation
suppcol_idx <- which(names(tabdata) %in% supp_cols)
```

4. Write your suppression algorithm.

Now we'll write our functions for suppression. If implementation is not of concern to you, copy the code below and skip to step 5.

First, we'll write the function to be called in the main script. This function will perform a first pass on all data values in the suppression columns and suppress values equal to or smaller than our cutoff value (previously defined with cell_bound). Afterwards, this function calls `complementary()`, our recursive function.

```
suppress <- function(df, supp_val, supp_char, supp_cols) {
  # First pass:
     # For all suppression columns (supp_cols), replace any cell value equal to
     # or less than suppression value (supp_val) with suppression character
  df <- df |>
    mutate(across(
      all_of(supp_cols), ~ if_else(
        .x <= supp_val & .x != 0,
        supp_char,
        as.character(.x))))

  # Call recursive function to perform complementary suppression
  df <- complementary(df)
  return(df)
}
```

Recursive functions continually call themselves until reaching a base case, where they have reached the desired condition and break out of the loop. We need a recursive function because we want to repeatedly change our dataframe until we find that each row and column passes the necessary conditions outlined by CSDE: zero or two or more values suppressed in each row and column to prevent values from being discoverable using marginal totals. `complementary()` calls a series of helper functions to perform this task.

```
complementary <- function(df) {
  # Checker dataframe needs to be remade every recursive call
  checker <- checker_df(df)

  # Base case: check_rows() returns NULL AND check_cols() returns NULL
  rows_to_fix <- check_rows(checker)
  cols_to_fix <- check_cols(checker)
  if (length(rows_to_fix) == 0) {
    if (length(cols_to_fix) == 0) {
      return(df) # Base case
    }
    df <- fix_all_cols(df, cols_to_fix)
  } else {
    df <- fix_all_rows(df, rows_to_fix)
    if (length(cols_to_fix != 0)) {
      df <- fix_all_cols(df, cols_to_fix)
    }
  }

  # Recurse with edited dataframe
```

```
  complementary(df)
}
```

Let's define all of our helper functions, starting with `checker_df()`. This function creates a burner dataframe that can be easily used to check whether there are a passing number of suppressed values in each row and column. The dataframe produced by `checker_df()`, `checker`, is passed into `check_rows()` and `check_cols()` to test if the dataframe has reached its base case.

```
# Forms burner dataframe with columns verifying value suppression
checker_df <- function(df) {
  checker <- df |>
    # If a value has been suppressed, replace it with 1.
    # If a value has not been suppressed, replace it with 0.
    mutate_at(supp_cols, ~ as.numeric(str_detect(., regex_suppchar)))

  return (checker)
}

check_rows <- function(checker) {
  row_checker <- checker |>
    # How many cells in a given row have been suppressed?
    mutate(num_supp = rowSums(checker[,suppcol_idx])) #All rows, only suppression columns
  # What are the indices of the rows that need to be fixed (have only 1 value suppressed)?
  fix_rows = which(row_checker$num_supp == 1)
  return (fix_rows)
}

check_cols <- function(checker) {
  # How many cells in a given column have been suppressed?
  colSums(checker[,which(names(checker) %in% supp_cols)])

  # What are the indices of the columns that need to be fixed (have only 1 value suppressed)?
  fix_cols = which(colSums(checker[which(names(checker) %in% supp_cols)]) == 1)
  return (fix_cols)
}
```

If `check_rows()` or `check_cols()` returns a list of column indices that need to be fixed, we call our helper functions to fix them using `fix_all_rows()` and `fix_all_cols()`, which simply serve to iteratively call `fix_row()` and `fix_col()`.

```
# Iterate through all rows to be fixed
fix_all_rows <- function(df, rows_to_fix) {
  # Could be consolidated with map() functions, but left as-is for readability
  for (i in rows_to_fix) {
      df <- fix_row(df, i)
    }
  return(df)
}
```

```r
# Iterate through all columns to be fixed
fix_all_cols <- function(df, cols_to_fix) {
  # Could be consolidated with map() functions, but left as-is for readability
  for (i in cols_to_fix) {
    df <- fix_col(df, i)
  }
  return(df)
}

fix_row <- function(df, i) {
  # Save row to be fixed as vector, remove suppressed values, convert to numeric
  row <- df[i, suppcol_idx]
  num_row <- as.numeric(row[!(row %in% c(supp_char, "0"))])

  # Randomly choose the index of one cell equal to the minimum row value to suppress
  # Subtract 1 to switch from 1-based indexing to 0-based indexing for data subsetting
  poss_idx <- which(row == min(num_row))
  if (length(poss_idx) != 1) {
    idx_to_supp <- sample(poss_idx, size = 1) - 1
  } else {
    idx_to_supp <- poss_idx - 1 }

  # Suppress selected cell with suppression character
  # Second argument enables proper indexing into larger dataframe:
      #idx_to_supp = index of column within relevant suppression columns
      #suppcol_idx[1] = starter index of first relevant suppression column
  df[i, idx_to_supp + suppcol_idx[1]] <- supp_char
  return(df)
}

fix_col <- function(df, i) {
  # Save column to be fixed as vector, remove suppressed values, convert to numeric
  col <- df[,i]
  num_col <- as.numeric(col[!(col %in% c(supp_char, "0"))])

  # Randomly choose the index of one cell equal to the minimum column value to suppress
  poss_idx <- which(col == min(num_col))
  if (length(poss_idx) != 1) {
    idx_to_supp <- sample(poss_idx, size = 1)
  } else {
    idx_to_supp <- poss_idx}

  # Suppress selected cell with suppression character
  # Second argument enables proper indexing into larger dataframe:
      #i = column being fixed
      #idx_to_supp = row of column to be suppressed
  df[idx_to_supp, i] <- supp_char
  return(df)
}
```

5. Call complementary suppression function.

Let's use our call stack! Since all of our values are defined from step 1, we can plug-and-chug with our new functions.

```
# Call initial function, which engages all other helper functions
suppressed_tabdata <- suppress(tabdata, cell_bound, supp_char, supp_cols)

# Print!
suppressed_tabdata |>
  kbl(caption = "Initial Data") |>
  kable_classic(full_width = F, html_font = "Cambria")
```

Initial Data

school	county	hispanic_count	white_count	black_count	total	hispanic_perc	black_perc	white_perc
abc	tolland	0	*	*	52	*	*	0.96
def	tolland	*	*	0	83	*	*	0.96
ghi	windham	10	20	23	53	0.19	0.43	0.38
jkl	windham	15	15	12	42	0.36	0.29	0.36
mno	windham	*	34	*	45	0.18	*	0.76
pqr	avon	*	23	*	33	0.15	0.15	0.7
stu	avon	6	*	*	16	*	*	*
vwx	hartford	20	0	23	43	0.47	0.53	*
yz	fairfield	30	*	*	35	0.86	*	*

Notes:

This code could be easily optimized and made concise. Excessive helper functions were used to improve readability and ease of functionality. Inspiration for implementation taken from open-source dar-tool, a Python tool performing complementary suppression.