Survey analysis

Overview

Ideally, survey respondents represent a completely random sample of the study population. However, this is rarely the case. Selection bias, non-random patterns in who responds to a survey, as well as other biases can influence the demographic makeup of survey respondents to be different than that of the study population. To help combat this large-scale surveys, include NHANES, often include **survey weights**.

In this chapter, we will use survey weights to calculate estimated statistics for an entire population based on a survey sample. Most survey R packages rely on the **survey** package for doing weighted analysis.

Preparation

Packages

```
## load packages from CRAN
pacman::p_load(rio,          # File import
               here,         # File locator
               tidyverse,    # data management + ggplot2 graphics
               tsibble,      # handle time series datasets
               survey,       # for survey functions
               gtsummary,    # wrapper for survey package to produce tables
               apyramid,     # a package dedicated to creating age pyramids
               patchwork,    # for combining ggplots
               ggforce,      # for alluvial/sankey plots
               epikit        # for age_categories
               )
```

Load data

The example dataset used in this section:

- fictional mortality survey data.
- fictional population counts for the survey area.
- data dictionary for the fictional mortality survey data.

This is based off the MSF OCA ethical review board pre-approved survey. The fictional dataset was produced as part of the "R4Epis" project. This is all based off data collected using KoboToolbox, which is a data collection software based off Open Data Kit.

Kobo allows you to export both the collected data, as well as the data dictionary for that dataset. We strongly recommend doing this as it simplifies data cleaning and is useful for looking up variables/questions.

```
# import the survey data
survey_data <- rio::import("survey_data.xlsx")

# import the dictionary into R
survey_dict <- rio::import("survey_dict.xlsx")
```

The first 10 rows of the survey are displayed below.

Show 5 entries Search:

start	end	today	deviceid	date	team_number	villa
				2018-04-15T00:00:00Z		villa
				2018-03-04T00:00:00Z		villa
				2018-04-16T00:00:00Z		villa
				2018-01-23T00:00:00Z		villa
				2018-01-09T00:00:00Z		villa

Showing 1 to 5 of 10 entries Previous 1 2 Next

We also want to import the data on sampling population so that we can produce appropriate weights. This data can be in different formats, however we would suggest to have it as seen below (this can just be typed in to an excel).

```
# import the population data
population <- rio::import("population.xlsx")
```

The first 10 rows of the survey are displayed below.

Show 5 entries Search:

age_group	sex			
0-2	male	0.034	340	district_a
3-14	male	0.1811	1811	district_a
15-29	male	0.138	1380	district_a
30-44	male	0.0808	808	district_a
45+	male	0.0661	661	district_a

Showing 1 to 5 of 10 entries Previous 1 2 Next

For cluster surveys you may want to add survey weights at the cluster level. You could read this data in as above. Alternatively if there are only a few counts, these could be entered as below in to a tibble. In any case you will need to have one column with a cluster identifier which matches your survey data, and another column with the number of households in each cluster.

```
## define the number of households in each cluster
cluster_counts <- tibble(cluster = c("village_1", "village_2", "village_3", "village_4",
                                     "village_5", "village_6", "village_7", "village_8",
                                     "village_9", "village_10"),
                         households = c(700, 400, 600, 500, 300,
                                        800, 700, 400, 500, 500))
```

Clean data

The below makes sure that the date column is in the appropriate format. There are several other ways of doing this, however using the dictionary to define dates is quick and easy.

We also create an age group variable using the `age_categories()` function from **epikit**. In addition, we create a character variable defining which district the various clusters are in. Finally, we recode all of the yes/no variables to TRUE/FALSE variables - otherwise these cant be used by the **survey** proportion functions.

```
## select the date variable names from the dictionary
DATEVARS <- survey_dict %>%
  filter(type == "date") %>%
  filter(name %in% names(survey_data)) %>%
  ## filter to match the column names of your data
  pull(name) # select date vars

## change to dates
survey_data <- survey_data %>%
  mutate(across(all_of(DATEVARS), as.Date))


## add those with only age in months to the year variable (divide by twelve)
survey_data <- survey_data %>%
  mutate(age_years = if_else(is.na(age_years),
                             age_months / 12,
                             age_years))
```

```r
## define age group variable
survey_data <- survey_data %>%
     mutate(age_group = age_categories(age_years,
                                       breakers = c(0, 3, 15, 30, 45)
                                       ))


## create a character variable based off groups of a different variable
survey_data <- survey_data %>%
  mutate(health_district = case_when(
    cluster_number %in% c(1:5) ~ "district_a",
    TRUE ~ "district_b"
  ))


## select the yes/no variable names from the dictionary
YNVARS <- survey_dict %>%
  filter(type == "yn") %>%
  filter(name %in% names(survey_data)) %>%
  ## filter to match the column names of your data
  pull(name) # select yn vars

## change to dates
survey_data <- survey_data %>%
  mutate(across(all_of(YNVARS),
                str_detect,
                pattern = "yes"))
```

```
Warning: There was 1 warning in `mutate()`.
i In argument: `across(all_of(YNVARS), str_detect, pattern = "yes")`.
Caused by warning:
! The `...` argument of `across()` is deprecated as of dplyr 1.1.0.
Supply arguments directly to `.fns` through an anonymous function instead.

  # Previously
  across(a:b, mean, na.rm = TRUE)

  # Now
  across(a:b, \(x) mean(x, na.rm = TRUE))
```

Survey data

There numerous different sampling designs that can be used for surveys. Here we will demonstrate code for: - Stratified - Cluster - Stratified and cluster

As described above (depending on how you design your questionnaire) the data for each level would be exported as a separate dataset. In our example there is one level for households and one level for individuals within those households.

These two levels are linked by a unique identifier. For a Kobo dataset this variable is "_index" at the household level, which matches the "_parent_index" at the individual level. This will create new rows for household with each matching individual, see the handbook section on joining for details.

```
## join the individual and household data to form a complete data set
survey_data <- left_join(survey_data_hh,
                         survey_data_indiv,
                         by = c("_index" = "_parent_index"))


## create a unique identifier by combining indeces of the two levels
survey_data <- survey_data %>%
     mutate(uid = str_glue("{index}_{index_y}"))
```

Observation time

For mortality surveys we want to now how long each individual was present for in the location to be able to calculate an appropriate mortality rate for our period of interest. This is not relevant to all surveys, but particularly for mortality surveys this is important as they are conducted frequently among mobile or displaced populations.

To do this we first define our time period of interest, also known as a recall period (i.e. the time that participants are asked to report on when answering questions). We can then use this period to set inappropriate dates to missing, i.e. if deaths are reported from outside the period of interest.

```
## set the start/end of recall period
## can be changed to date variables from dataset
## (e.g. arrival date & date questionnaire)
survey_data <- survey_data %>%
  mutate(recall_start = as.Date("2018-01-01"),
         recall_end   = as.Date("2018-05-01")
```

```
  )


# set inappropriate dates to NA based on rules
## e.g. arrivals before start, departures departures after end
survey_data <- survey_data %>%
      mutate(
           arrived_date = if_else(arrived_date < recall_start,
                                  as.Date(NA),
                                  arrived_date),
           birthday_date = if_else(birthday_date < recall_start,
                                   as.Date(NA),
                                   birthday_date),
           left_date = if_else(left_date > recall_end,
                               as.Date(NA),
                               left_date),
           death_date = if_else(death_date > recall_end,
                                as.Date(NA),
                                death_date)
           )
```

We can then use our date variables to define start and end dates for each individual. We can use the `find_start_date()` function from **sitrep** to fine the causes for the dates and then use that to calculate the difference between days (person-time).

start date: Earliest appropriate arrival event within your recall period Either the beginning of your recall period (which you define in advance), or a date after the start of recall if applicable (e.g. arrivals or births)

end date: Earliest appropriate departure event within your recall period, or a date before the end of recall if applicable (e.g. departures, deaths)

```
## create new variables for start and end dates/causes
survey_data <- survey_data %>%
     ## choose earliest date entered in survey
     ## from births, household arrivals, and camp arrivals
     find_start_date("birthday_date",
                     "arrived_date",
                     period_start = "recall_start",
                     period_end   = "recall_end",
                     datecol      = "startdate",
                     datereason   = "startcause"
                    ) %>%
```

```
     ## choose earliest date entered in survey
     ## from camp departures, death and end of the study
     find_end_date("left_date",
                   "death_date",
                   period_start = "recall_start",
                   period_end   = "recall_end",
                   datecol      = "enddate",
                   datereason   = "endcause"
                  )


## label those that were present at the start/end (except births/deaths)
survey_data <- survey_data %>%
     mutate(
       ## fill in start date to be the beginning of recall period (for those empty)
       startdate = if_else(is.na(startdate), recall_start, startdate),
       ## set the start cause to present at start if equal to recall period
       ## unless it is equal to the birth date
       startcause = if_else(startdate == recall_start & startcause != "birthday_date",
                            "Present at start", startcause),
       ## fill in end date to be end of recall period (for those empty)
       enddate = if_else(is.na(enddate), recall_end, enddate),
       ## set the end cause to present at end if equall to recall end
       ## unless it is equal to the death date
       endcause = if_else(enddate == recall_end & endcause != "death_date",
                          "Present at end", endcause))


## Define observation time in days
survey_data <- survey_data %>%
  mutate(obstime = as.numeric(enddate - startdate))
```

Weighting

It is important that you drop erroneous observations before adding survey weights. For example if you have observations with negative observation time, you will need to check those (you can do this with the `assert_positive_timespan()` function from **sitrep**. Another thing is if you want to drop empty rows (e.g. with `drop_na(uid)`) or remove duplicates (see handbook section on [De-duplication] for details). Those without consent need to be dropped too.

In this example we filter for the cases we want to drop and store them in a separate data frame - this way we can describe those that were excluded from the survey. We then use the

`anti_join()` function from **dplyr** to remove these dropped cases from our survey data.

> ⚠ Warning
>
> You cant have missing values in your weight variable, or any of the variables relevant to your survey design (e.g. age, sex, strata or cluster variables).

```
## store the cases that you drop so you can describe them (e.g. non-consenting
## or wrong village/cluster)
dropped <- survey_data %>%
  filter(!consent | is.na(startdate) | is.na(enddate) | village_name == "other")

## use the dropped cases to remove the unused rows from the survey data set
survey_data <- anti_join(survey_data, dropped, by = names(dropped))
```

As mentioned above we demonstrate how to add weights for three different study designs (stratified, cluster and stratified cluster). These require information on the source population and/or the clusters surveyed. We will use the stratified cluster code for this example, but use whichever is most appropriate for your study design.

```
# stratified -------------------------------------------------------------------
# create a variable called "surv_weight_strata"
# contains weights for each individual - by age group, sex and health district
survey_data <- add_weights_strata(x = survey_data,
                                  p = population,
                                  surv_weight = "surv_weight_strata",
                                  surv_weight_ID = "surv_weight_ID_strata",
                                  age_group, sex, health_district)

## cluster ---------------------------------------------------------------------

# get the number of people of individuals interviewed per household
# adds a variable with counts of the household (parent) index variable
survey_data <- survey_data %>%
  add_count(index, name = "interviewed")


## create cluster weights
survey_data <- add_weights_cluster(x = survey_data,
                                   cl = cluster_counts,
                                   eligible = member_number,
                                   interviewed = interviewed,
```

```
                                        cluster_x = village_name,
                                        cluster_cl = cluster,
                                        household_x = index,
                                        household_cl = households,
                                        surv_weight = "surv_weight_cluster",
                                        surv_weight_ID = "surv_weight_ID_cluster",
                                        ignore_cluster = FALSE,
                                        ignore_household = FALSE)


# stratified and cluster ----------------------------------------------------
# create a survey weight for cluster and strata
survey_data <- survey_data %>%
  mutate(surv_weight_cluster_strata = surv_weight_strata * surv_weight_cluster)
```

Survey design objects

Create survey object according to your study design. Used the same way as data frames to calculate weight proportions etc. Make sure that all necessary variables are created before this.

There are four options, comment out those you do not use: - Simple random - Stratified - Cluster - Stratified cluster

For this template - we will pretend that we cluster surveys in two separate strata (health districts A and B). So to get overall estimates we need have combined cluster and strata weights.

As mentioned previously, there are two packages available for doing this. The classic one is **survey** and then there is a wrapper package called **srvyr** that makes tidyverse-friendly objects and functions. We will demonstrate both, but note that most of the code in this chapter will use **srvyr** based objects. The one exception is that the **gtsummary** package only accepts **survey** objects.

Survey package

The **survey** package effectively uses **base** *R* coding, and so it is not possible to use pipes (`%>%`) or other **dplyr** syntax. With the **survey** package we use the `svydesign()` function to define a survey object with appropriate clusters, weights and strata.

NOTE: we need to use the tilde (`~`) in front of variables, this is because the package uses the **base** *R* syntax of assigning variables based on formulae.

```r
# simple random ---------------------------------------------------------------
base_survey_design_simple <- svydesign(ids = ~1, # 1 for no cluster ids
                    weights = NULL,              # No weight added
                    strata = NULL,               # sampling was simple (no strata)
                    data = survey_data           # have to specify the dataset
                   )

## stratified -----------------------------------------------------------------
base_survey_design_strata <- svydesign(ids = ~1,  # 1 for no cluster ids
                    weights = ~surv_weight_strata, # weight variable created above
                    strata = ~health_district,     # sampling was stratified by district
                    data = survey_data             # have to specify the dataset
                   )

# cluster ---------------------------------------------------------------------
base_survey_design_cluster <- svydesign(ids = ~village_name, # cluster ids
                    weights = ~surv_weight_cluster, # weight variable created above
                    strata = NULL,                  # sampling was simple (no strata)
                    data = survey_data               # have to specify the dataset
                   )

# stratified cluster ----------------------------------------------------------
base_survey_design <- svydesign(ids = ~village_name,      # cluster ids
                    weights = ~surv_weight_cluster_strata, # weight variable created above
                    strata = ~health_district,             # sampling was stratified by dis
                    data = survey_data                     # have to specify the dataset
                   )
```

Descriptive analysis

In this section we will focus on how to investigate and visualize bias in a sample. We will also look at visualising population flow in a survey setting using alluvial/sankey diagrams.

In general, you should consider including the following descriptive analyses:

- Final number of clusters, households and individuals included

- Number of excluded individuals and the reasons for exclusion
- Median (range) number of households per cluster and individuals per household

Sampling bias

Compare the proportions in each age group between your sample and the source population. This is important to be able to highlight potential sampling bias. You could similarly repeat this looking at distributions by sex.

Note that these p-values are just indicative, and a descriptive discussion (or visualisation with age-pyramids below) of the distributions in your study sample compared to the source population is more important than the binomial test itself. This is because increasing sample size will more often than not lead to differences that may be irrelevant after weighting your data.

```
## counts and props of the study population
ag <- survey_data %>%
  group_by(age_group) %>%
  drop_na(age_group) %>%
  tally() %>%
  mutate(proportion = n / sum(n),
         n_total = sum(n))

## counts and props of the source population
propcount <- population %>%
  group_by(age_group) %>%
    tally(population) %>%
    mutate(proportion = n / sum(n))

## bind together the columns of two tables, group by age, and perform a
## binomial test to see if n/total is significantly different from population
## proportion.
  ## suffix here adds to text to the end of columns in each of the two datasets
left_join(ag, propcount, by = "age_group", suffix = c("", "_pop")) %>%
  group_by(age_group) %>%
  ## broom::tidy(binom.test()) makes a data frame out of the binomial test and
  ## will add the variables p.value, parameter, conf.low, conf.high, method, and
  ## alternative. We will only use p.value here. You can include other
  ## columns if you want to report confidence intervals
  mutate(binom = list(broom::tidy(binom.test(n, n_total, proportion_pop)))) %>%
  unnest(cols = c(binom)) %>% # important for expanding the binom.test data frame
  mutate(proportion_pop = proportion_pop * 100) %>%
  ## Adjusting the p-values to correct for false positives
  ## (because testing multiple age groups). This will only make
  ## a difference if you have many age categories
```

```
  mutate(p.value = p.adjust(p.value, method = "holm")) %>%

  ## Only show p-values over 0.001 (those under report as <0.001)
  mutate(p.value = ifelse(p.value < 0.001,
                          "<0.001",
                          as.character(round(p.value, 3)))) %>%

  ## rename the columns appropriately
  select(
    "Age group" = age_group,
    "Study population (n)" = n,
    "Study population (%)" = proportion,
    "Source population (n)" = n_pop,
    "Source population (%)" = proportion_pop,
    "P-value" = p.value
  )
```

```
# A tibble: 5 x 6
# Groups:   Age group [5]
  `Age group` `Study population (n)` `Study population (%)`
  <chr>                        <int>                  <dbl>
1 0-2                             12                 0.0256
2 3-14                            42                 0.0896
3 15-29                           64                 0.136
4 30-44                           52                 0.111
5 45+                            299                 0.638
# i 3 more variables: `Source population (n)` <dbl>,
#   `Source population (%)` <dbl>, `P-value` <chr>
```

Weighted proportions

This section will detail how to produce tables for weighted counts and proportions, with associated confidence intervals and design effect.

Survey package

We can use the `svyciprop()` function from **survey** to get weighted proportions and accompanying 95% confidence intervals. An appropriate design effect can be extracted using the `svymean()` rather than `svyprop()` function. It is worth noting that `svyprop()` only appears

to accept variables between 0 and 1 (or TRUE/FALSE), so categorical variables will not work.

NOTE: Functions from **survey** also accept **srvyr** design objects, but here we have used the **survey** design object just for consistency

```
## produce weighted counts
svytable(~died, base_survey_design)
```

```
died
     FALSE       TRUE
1406244.43   76213.01
```

```
## produce weighted proportions
svyciprop(~died, base_survey_design, na.rm = T)
```

```
              2.5% 97.5%
died 0.0514 0.0208  0.12
```

```
## get the design effect
svymean(~died, base_survey_design, na.rm = T, deff = T) %>%
  deff()
```

```
diedFALSE  diedTRUE
 3.755508  3.755508
```

We can combine the functions from **survey** shown above in to a function which we define ourselves below, called `svy_prop`; and we can then use that function together with `map()` from the purrr package to iterate over several variables and create a table. See the handbook iteration chapter for details on **purrr**.

```
# Define function to calculate weighted counts, proportions, CI and design effect
# x is the variable in quotation marks
# design is your survey design object

svy_prop <- function(design, x) {

  ## put the variable of interest in a formula
  form <- as.formula(paste0( "~" , x))
```

```r
  ## only keep the TRUE column of counts from svytable
  weighted_counts <- svytable(form, design)[[2]]
  ## calculate proportions (multiply by 100 to get percentages)
  weighted_props <- svyciprop(form, design, na.rm = TRUE) * 100
  ## extract the confidence intervals and multiply to get percentages
  weighted_confint <- confint(weighted_props) * 100
  ## use svymean to calculate design effect and only keep the TRUE column
  design_eff <- deff(svymean(form, design, na.rm = TRUE, deff = TRUE))[[TRUE]]

  ## combine in to one data frame
  full_table <- cbind(
    "Variable"        = x,
    "Count"           = weighted_counts,
    "Proportion"      = weighted_props,
    weighted_confint,
    "Design effect"   = design_eff
    )

  ## return table as a dataframe
  full_table <- data.frame(full_table,
             ## remove the variable names from rows (is a separate column now)
             row.names = NULL)

  ## change numerics back to numeric
  full_table[ , 2:6] <- as.numeric(full_table[, 2:6])

  ## return dataframe
  full_table
}

## iterate over several variables to create a table
purrr::map(
  ## define variables of interest
  c("left", "died", "arrived"),
  ## state function using and arguments for that function (design)
  svy_prop, design = base_survey_design) %>%
  ## collapse list in to a single data frame
  bind_rows() %>%
  ## round
  mutate(across(where(is.numeric), round, digits = 1))
```

```
Variable    Count Proportion X2.5. X97.5. Design.effect
```

```
1     left 701199.1       47.3  39.2   55.5          2.4
2     died  76213.0        5.1   2.1   12.1          3.8
3  arrived 761799.0       51.4  40.9   61.7          3.9
```

Weighted ratios

Similarly for weighted ratios (such as for mortality ratios) you can use `survey:::svyratio`:

Survey package

```
ratio <- svyratio(~died,
          denominator = ~obstime,
          design = base_survey_design)

ci <- confint(ratio)

cbind(
  ratio$ratio * 10000,
  ci * 10000
)
```

```
      obstime    2.5 %   97.5 %
died 5.981922 1.194294 10.76955
```