`Rsynthpops` Midas Tutorial

5/10/2021

Check back on May 9th for the final version of this document

Background

We'll assume a working knowledge of R, particularly how to read and manipulate data. Our goal is to describe the basic functioning of the package, how to add characteristics to your population that are not already supported, and how to validate your population once it's created.

Setup: PLEASE COMPLETE THESE STEPS PRIOR TO THE TUTORIAL ON MAY 10

- Install the package and any necessary prereqs by running `devtools::install_github("cmhoove14/Rsynthpops")`
- If you don't have one already, get a US census api key at https://api.census.gov/data/key_signup.html. To store your api key, run `tidycensus::census_api_key(your_key, install = TRUE)` replacing `your_key` with the key you obtained from the url above.
- Think about a place from which you want to generate a synthetic population. It could be a county or state where you are currently conducting some modeling work, the county you grew up in, a state you find interesting, or somewhere entirely random! If you choose a state, have that state's two character state abbreviation handy. If it's a county, have the county's 5-digit Federal Information Processing Standard (FIPS) code handy. These are easy to find via a web search with the county's name and "fips code".

Main commands

- `rsp_get_` commands to download raw PUMS, ACS, and census data
- `rsp_process_` commands to convert raw data into household/group quarters and person level seed and target data
- `rsp_synth_` commands to synthesize your population from your target and seed datasets

Key steps

Getting input data

The first decision to be made is which household and person characteristics are to be included in the synthetic population. `Rsynthpops` builds on functions from `tidycensus` to get PUMS and ACS data and adds an additional function to get data on group quarters directly from the census ftp site. Make sure to get and/or supply your census api key in order to use the `tidycensus` download functions.

```
state_use  = "CA"
fips_use   = "06075" # Coffee county fips code
year_use   = 2019
survey_use = "acs5"
pumas_use  = cts_to_pumas %>%
```

```
  filter(STCNTYFP == fips_use) %>%
  pull(PUMA5CE) %>%
  unique()
```

```
# Variables we want to include in the final synthetic population
characteristics <- c(
  "Sex",
  "Age",
  "Race",
  "Ethnicity",
  "Occupation",
  "Grade",
  "School_Type",
  "HH_Income",
  "HH_Type",
  "HH_Size"
)

# Get group quarters data
gq_dat <- rsp_get_gq(
  STATE = state_use,
  LEVEL = "Tract"
)

# Get PUMS data
pums_dat <- rsp_get_pums(
  VARS   = characteristics,
  SURVEY = "acs5",
  STATES = state_use,
  YEAR   = year_use
)

# Get ACS data
acs_dat <- rsp_get_acs(
  VARS   = characteristics,
  SURVEY = "acs5",
  STATES = state_use,
  LEVEL  = "Tract",
  YEAR   = year_use
)
```

Processing data

Now that all the data is downloaded, it needs to be processed to feed into the synthesizer, in this case, the `ipu()` function from the `ipfr` package. Other synthesizers are being considered and tested, but iterative proportional updating is currently the canonical method. `ipu()` requires "seed" and "target" data for every agent characteristics, where the seeds come from individual pums records and targets come from aggregated acs/census records.

```
all_pums_hh_vars <- pums_variables %>%
  filter(year == year_use,
         survey == survey_use,
         level == "housing") %>%
  pull(var_code) %>%
```

```
  unique()

# Determine which variables are household level
pums_vars <- colnames(pums_dat)
hh_vars <- pums_vars[which(pums_vars %in% all_pums_hh_vars)]

# Determine which variables are person level as those leftover from household vars
p_vars <- pums_vars[which(!pums_vars %in% all_pums_hh_vars)]

# Determine fips of areas to synthesize population for
synth_fips <- unique(acs_dat[[1]]$GEOID[which(grepl(fips_use, acs_dat[[1]]$GEOID))])
```

```
gq_seed <- rsp_process_gq_seed(
  pums_dat = pums_dat,
  pumas    = pumas_use
)

gq_tgt  <- rsp_process_gq_tgt(
  gq_dat   = gq_dat[[1]],
  fips_use = fips_use
)

gq_pop <- bind_rows(lapply(
  synth_fips,
  function(i){
    rsp_gq_synth(i, gq_seed, gq_tgt)
  }
))
```

```
hh_seed <- rsp_process_hh_seed(
  pums_dat         = pums_dat,
  pumas            = pumas_use,
  hh_vars          = hh_vars,
  hh_income_breaks = c(-100,50,100,100000)*1000
)

hh_tgt <- rsp_process_hh_tgt(
  acs_hh_dat       = acs_dat,
  chars            = characteristics,
  fips_use         = fips_use,
  hh_income_breaks = c(-100,50,100,100000)*1000
)

p_seed <- rsp_process_p_seed(
  pums_dat     = pums_dat,
  pumas        = pumas_use,
  p_vars       = p_vars,
  p_age_breaks = c(0,10,20,30,40,50,60,70,80,150)
)

p_tgt <- rsp_process_p_tgt(
  acs_p_dat       = acs_dat,
  chars            = characteristics,
  fips_use         = fips_use,
```

```
  p_age_breaks = c(0,10,20,30,40,50,60,70,80,150)
)
```

Synthesizing population

```
synth_pop <- bind_rows(lapply(
  synth_fips,
  function(i){
    cat(i, "\n")
    rsp_hhp_synth(i, hh_seed, p_seed, hh_tgt, p_tgt, gq_pop)
  }
))
```

Validating population

Global comparison of true to synthetic population

```
n_vars <- length(hh_tgt) + length(p_tgt)
n_cts  <- length(synth_cts)

val_df_tgt <- data.frame("ct"    = rep(synth_cts, times = n_vars),
                         "vrbl"  = rep(c(names(hh_tgt), names(p_tgt)), each = n_cts),
                         "pmiss" = NA)

val_df_tgt$pmiss <- apply(val_df_tgt, 1, FUN = function(i){
  ct   = as.character(i[1])
  vrbl = as.character(i[2])

  #cat(vrbl, "\n")

  if(vrbl %in% names(hh_tgt)){

    obs <- lapply(hh_tgt, function(i){
      i %>% filter(geo_tract == ct) %>% dplyr::select(-geo_tract)
    })[[vrbl]] %>%
      pivot_longer(cols = everything(), names_to = vrbl, values_to = "n") %>%
      group_by(!!as.name(vrbl)) %>%
      summarise(n = sum(n))

    # One obs per household
    ct <- synth_pop %>% filter(GEOID == ct)
    ct_hhs <- ct[!duplicated(ct$house_id),]

    gen <- ct_hhs %>%
      group_by(!!as.name(vrbl)) %>%
      dplyr::count()

    class(gen[[vrbl]]) <- class(obs[[vrbl]])

    jnd <- obs %>%
      left_join(gen, by = vrbl, suffix = c("_obs", "_gen")) %>%
      mutate(diff = abs(n_obs - n_gen))
```

```
    out <- sum(jnd$diff, na.rm = T)/sum(jnd$n_obs)

  } else if(vrbl %in% names(p_tgt)){

    obs <- lapply(p_tgt, function(j){
      j %>% filter(geo_tract == ct) %>% dplyr::select(-geo_tract)
    })[[vrbl]] %>%
       pivot_longer(cols = everything(), names_to = vrbl, values_to = "n")

    gen <- synth_pop %>%
      filter(GEOID == ct) %>%
      group_by(!!as.name(vrbl)) %>%
      dplyr::count()

    class(gen[[vrbl]]) <- class(obs[[vrbl]])

    jnd <- obs %>%
      left_join(gen, by = vrbl, suffix = c("_obs", "_gen")) %>%
      mutate(diff = abs(n_obs - n_gen))

    out <- sum(jnd$diff, na.rm = T)/sum(jnd$n_obs)

  } else {

    stop("Variable has no match in target data")

  }

  return(out)

})

val_df_tgt %>%
  ggplot(aes(x = vrbl, y = pmiss, fill = vrbl)) +
    geom_violin() +
    scale_y_continuous(breaks = seq(0,2, by = 0.1)) +
    scale_x_discrete(labels = c("hhincome" = "HH\nIncome",
                                "hhtype"   = "HH\nType",
                                "hhsize"   = "HH\nSize",
                                "grade"    = "Grade",
                                "sex"      = "Sex",
                                "occ_group"= "Occp\nGroup",
                                "age_cat"  = "Age\nCat",
                                "race"     = "Race",
                                "Hispanic" = "Hispanic")) +
    theme_classic() +
    theme(legend.position = "none") +
    labs(x = "Variable",
         y = "Normalized L1",
         title = "L1 normalized error distribution across CTs")

ggsave(here::here("Plots/PctError_Distn_Validation.png"),
       height = 4, width = 5, units = "in")
```

```
high_err_cts <- val_df_tgt %>%
  filter(pmiss > 0.5) %>%
  pull(ct) %>%
  unique()

val_df_tgt %>%
  filter(!ct %in% high_err_cts) %>%
  ggplot(aes(x = vrbl, y = pmiss, fill = vrbl)) +
    geom_violin() +
    scale_y_continuous(breaks = seq(0,2, by = 0.1)) +
    scale_x_discrete(labels = c("hhincome" = "HH\nIncome",
                                "hhtype"   = "HH\nType",
                                "hhsize"   = "HH\nSize",
                                "grade"    = "Grade",
                                "sex"      = "Sex",
                                "occ_group"= "Occp\nGroup",
                                "age_cat"  = "Age\nCat",
                                "race"     = "Race",
                                "Hispanic" = "Hispanic")) +
    theme_classic() +
    theme(legend.position = "none") +
    labs(x = "Variable",
         y = "Normalized L1",
         title = "L1 normalized error distribution across CTs",
         subtitle = " *Large error CTs omitted")

ggsave(here::here("Plots/PctError_Distn_Validation_rmv_high_error.png"),
       height = 4, width = 5, units = "in")
```

Deviation from attribute targets by census tract

Adding new characteristics

`tidycensus` ships with a dataset `pums_variables` where users can inspect potential variables to include in the PUMS download. Below, we see the first few variables available for the 5-year 2019 PUMS data.

```
pums_vars_all <- pums_variables %>%
  filter(year == 2018, survey == "acs5")

pums_vars <- pums_vars_all %>%
  distinct(var_code, var_label, data_type, level)

  head(pums_vars, n = 10)
```

`tidycensus` also has a function `load_variables()` to view aggregate variables reported in the acs.

```
acs_vars<-tidycensus::load_variables(2019, "acs5", cache=FALSE)
head(acs_vars)

acs_vars_subject<-load_variables(year_use, paste0(survey_use, "/subject"), cache=FALSE)
acs_vars_profile<-load_variables(year_use, paste0(survey_use, "/profile"), cache=FALSE)
```

- Inspect ACS and PUMS variables for data pertaining to your desire characteristics

- Determine how to match ACS and PUMS variables
- Process into target and seed characteristics
- Synthesize with new characteristic
- Validate