Query Functions for Public Use Microdata Sample Census API

Katy Kearns Chris Hill

2024-10-02

Table of contents

Introduction

"Our goal is to write functions that will manipulate and process data sets that come from a census API."

This is direct from our project statement and effectively sums up the work below. What follows is a narrative describing some of the trials and tribulations that were overcome in implementing this direction along with the source code that serves as a working product.

The motivation for this work was to develop a user friendly set of functions that allow us to easily query the census API. The user would then receive the response in a format that could be immediately usable in an R session for further analysis. In this case in the form of a tibble.

This process involved building a valid URL from the user-desired information (inputs). These inputs were validated and used to construct the URl that requests user-defined information from the API. Once successful, the API response is then processed into a formatted tibble where each column of data is converted to its appropriate data type.

Though strict public/private functions are not enforced here, we do make note of the four distinct functions intended to be interacted with by a potential user. By building these dedicated interfaces, we can rigorously validate user inputs and hope to provide effective feedback to guide the user if an error occurs.

We have also relied heavily on the separation of concerns between functions to support our collaborative efforts. Small, focused functions that might be reused wherever possible were implemented with great effect and were more easily maintained.

Setting Things Up

Collaboration Workflow

As mentioned above, this project was a collaborative effort. The project is housed in a GIT repository to facilitate this. Because of the small size of our team, we had development success working simply off a main branch in GIT. A development branch was created to explore the option of merging, but it proved to be more difficult than necessary for our workflow.

Libraries

```r
# Load required libraries
library(tidyverse)
library(jsonlite)
library(httr)
library(hms)
```

Primary User Interface Methods

The following two functions are designed to take user input and return a properly formatted and easy to use data table or *tibble*. Before anything further, they will validate each input from the user. They are then passed into functions whose purpose is creating, processing, and handling our API request and response. With user validation checks up front, the helper functions can assume that their inputs are as expected and they can focus on the the more rigorous tasks of transforming user inputs into usable data structures.

We will also introduce two further interface methods that allow for additional features for summarizing numeric and categorical data as well as for plotting. These Generic Class Functions will take the tibble of data processed from the census API request as an input to display meaningful summaries and plots depending on the user's demands.

get_data_tibble_from_census_api

- The first step is to define the inputs we expect from the user. All of our inputs have default settings. As such, this function can be called without arguments and will return census data from the year 2022 from the state of Colorado. In particular, it will return the numeric variables AGEP and PWGTP. As well as the categorical variable SEX.
- No matter the case, we always pass these inputs to various validation functions. We will discuss each of these in turn below.
- With valid input from the user, we then set a chain of events to first build the URL from those inputs, then send that request for information from the census. With the data in it's raw form from the API, we can then work to format it for ease of use.

```r
# User interface to take inputs and return fully processed data tibble
get_data_tibble_from_census_api <- function(year = 2022,
                                            numeric_vars = c("AGEP", "PWGTP"),
                                            categorical_vars = c("SEX"),
                                            geography = "State",
                                            subset = "CO") {

  # validate the user inputs
```

```
  validate_year(year)
  validate_numeric_vars(numeric_vars)
  validate_categorical_vars(categorical_vars)
  validate_geography_and_subset(geography, subset)

  # Send inputs to retrieve data
  build_query_url(year,
                  numeric_vars,
                  categorical_vars,
                  geography,
                  subset) |>
    query_census_with_url()
}
```

query_census_multiple_years

- In addition to the same arguments as the single year version, this function also takes in a range of years
- A loop calls the single year function with each iteration, adding the results as new elements to a list. Each tibble also has a new column designating the year.
- After the loop, bind rows is used to combine all results into one tibble

```
# Function for Querying Multiple Years
query_census_multiple_years <- function(years,
                                numeric_vars = c("AGEP", "PWGTP"),
                                categorical_vars = c("SEX"),
                                geography = "State",
                                subset = "CO") {

  # create empty list to store data frames
  multi_year_list <- list()

  # call the user interface for each year
  for (yr in years) {

    # retrieve single year data tibble
    census_single_yr <- get_data_tibble_from_census_api(yr,
                                                        numeric_vars,
                                                        categorical_vars,
                                                        geography,
```

```
                                                        subset)

    # append year to the tibble
    census_single_yr_tbl <- tibble(Year = yr, census_single_yr)

    # check how many elements are currently in list
    elements <- length(multi_year_list)

    # insert the tbl into the list as the last element
    multi_year_list[[elements + 1]] <- census_single_yr_tbl
  }

  # union of all year-specific results
  census_multi_year_tbl <- bind_rows(multi_year_list)

  # return the final tibble
  return(census_multi_year_tbl)
}
```

Data Processing

Utilities

We define several helper functions that allow us to fetch particular sets of information whenever required. These facilitate readability, reusability, and our approach to separate concerns.

```
get_valid_numeric_vars <- function() {
  c("AGEP", "PWGTP", "GASP", "GRPIP", "JWAP", "JWDP", "JWMNP")
}

get_valid_categorical_vars <- function() {
  c("SEX", "FER", "HHL", "HISPEED", "JWTRNS", "SCH", "SCHL")
}

get_valid_geography_levels <- function() {
  c("All", "Region", "Division", "State")
}
```

get_time_refs

- This function does an API call to retrieve the time codes and their associated time ranges in a JSON string and processes them into a tibble, an undertaking that proved arduous and time-consuming.
- The URL for the API endpoint is actually a .json, so that might be why the data come in differently than the API call for the census data. The values actually came in the form of a list with all 151-286 values as individual elements in that list.
- Bind_rows was used to get those elements into a single tibble, so now each named element was its own column. This data table was transposed using pivot_longer to get just 2 columns, one for the time code and the other for the time range
- Getting from the text time ranges, e.g. "8:50 a.m. to 8:54 a.m." to a nice, clean time (08:52:00) took awhile and a few tries. On the first pass, we assumed both JWAP and JWDP had consistent time ranges, so the first version just took a substring for the first time in the interval, converted it to time, and add 2 minutes to JWAP values and 5 minutes to JWDP values.
- While debugging the issue with time code values changing from factor to numeric, we realized that JWDP had varying intervals. The logic was changed to take both the starting and ending times to compute the midpoint of the interval
- Since we had to repeat the same actions on two different strings to get them both in correct time format, these steps were put into a loop. After that, getting the final time and removing the extraneous columns was pretty straightforward.

```
get_time_refs <- function(time_code) {

  # construct url from the time_code (JWDP or JWAP)
  time_url <- paste0("https://api.census.gov/data/2022/acs/acs1/pums/variables/",
                     time_code, ".json")

  # retrieve data in list form from API, then bind rows to put in 1 by x tibble,
  # then transpose the data to get key-value pair in columns
  times_ref <-
    fromJSON(time_url)$values |>
    bind_rows() |>
    pivot_longer(cols = everything(),
                 names_to = time_code,
                 values_to = "time_range")

  # convert 1st column (JWAP/JWDP) to numeric
  times_ref[[time_code]] <- as.numeric(times_ref[[time_code]])
```

```r
  # filter on the row(s) where JWAP/JWDP == 0, change the value for time_range
  # to missing (it is a string that can't be converted to time, starts with "N/A")
  times_ref$time_range[times_ref[time_code] == 0] <- NA

  # parse the time_range string to find the start and stop times
  times_ref <-
    times_ref |>
    separate_wider_delim(time_range,
                         delim = " to ",
                         names = c("start_time", "end_time"),
                         cols_remove = FALSE)

  # convert new start/end columns to time
  for (col in c("start_time", "end_time")) {
    times_ref[[col]] <-
      times_ref[[col]] |>
      toupper() |>                              # change to upper case
      str_replace_all("[.]", "") |>             # remove periods
      parse_date_time('%I:%M %p', tz = "EST")   # convert to date-time
  }

  # calculate time to the midpoint between the start and end times
  times_ref <-
    times_ref |>
    mutate(midpoint = difftime(end_time, start_time) / 2)

  # assign new clean time code variable as correct time
  times_ref[paste0(time_code, "_clean")] <-
    times_ref$start_time + times_ref$midpoint

  # convert format from date-time to time (reference by [[]] not [])
  times_ref[paste0(time_code, "_clean")] <-
    hms::as_hms(times_ref[[paste0(time_code, "_clean")]])

  # drop extra columns, keeping only 2 (time code, time code clean)
  times_ref <-
    times_ref |>
    select(-time_range, -start_time, -end_time, -midpoint)

  # return final clean ref table
  return(times_ref)
}
```

get_cat_refs

- When it was stated in a message on the discussion board that we should convert all categorical values to their meaningful versions, we realized we needed to create reference tables with an API call
- We were able to utilize our existing get_time_refs function and create a simplified version for the categorical variables.
- This is a helper function for the function that actually performs the conversion, described below

```
get_cat_refs <- function(cat_code) {

  # construct url from the time_code (JWDP or JWAP)
  cat_url <- paste0("https://api.census.gov/data/2022/acs/acs1/pums/variables/",
                    cat_code, ".json")

  # retrieve data in list form from API, then bind rows to put in 1 by x tibble,
  # then transpose the data to get key-value pair in columns
  cat_ref <-
    fromJSON(cat_url)$values |>
    bind_rows() |>
    pivot_longer(cols = everything(),
                 names_to = cat_code,
                 values_to = "description")

  return(cat_ref)
}
```

```
get_cat_refs("REGION") |> knitr::kable(align = 'c')
```

REGION	description
1	Northeast
9	Puerto Rico
3	South
2	Midwest
4	West

get_subset_code

In the construction of the URL, this function will take as arguments the combination of Geography and Subset to return a particular subset's numeric code used in the URL. If the subset is NULL, we simply return an astrix to be used with any geography the user might provide. Otherwise, we need to retrieve the appropriate subset pertaining to the given geography.

- A great opportunity was taken to reuse the get_cat_refs from above here. The regions and divisions had previously been hard-coded within this function, but could now be pulled directly from the API using a single function call.

```
get_subset_code <- function(geography, subset) {

  if (is.null(subset)) {
    return("*")
  }

  geography <- tolower(geography)

  # use get_cat_refs
  region_codes <- get_cat_refs("REGION") |>
    select(description, REGION)

  division_codes <- get_cat_refs("DIVISION") |>
    select(description, DIVISION) |>
    # Division codes come back with unwanted info in parenthesis
    mutate(description = str_remove(description, "\\s*\\(.*?\\)"))

  # Switch based on geography type
  switch(geography,
         "region" = region_codes |>
                    filter(tolower(description) == tolower(subset)) |>
                    pull(REGION),
         "division" = division_codes |>
                    filter(tolower(description) == tolower(subset)) |>
                    pull(DIVISION),
         "state" = get_state_code(subset),
         stop("Invalid geography type"))
}
```

```
get_subset_code("region", "south")
```

```
[1] "3"
```

```
get_subset_code("division", "mountain")
```

```
[1] "8"
```

get_state_code

We have a dedicated helper for states. This function is not only used to retrieve the appropriate numeric code that is to be used in the construction of the URL, it also has a *stop()* within it that will allow us the validate if the the state input returns a valid code.

- Another fantastic use of get_cat_refs. Initially for get_state_code, the numeric codes and their corresponding states were hardcoded as a named vector and contained within. This was refactored however to greatly reduce the code and increased readability and robustness.

```
# Function to get state code from state name or abbreviation
get_state_code <- function(state_input) {

  state_input <- tolower(state_input)

  # Provided state codes
  state_codes <- get_cat_refs("ST")

  state_codes_tibble <- state_codes |>
    separate_wider_delim(description, delim = "/",
                         names = c("state", "abbreviation")) |>
    mutate(state = tolower(state), abbreviation = tolower(abbreviation))

  # Filter down to match input
  result <- state_codes_tibble |>
    filter(state == state_input | abbreviation == state_input) |>
    pull(ST)

  # Return the state code or stop if not found
  if (length(result) == 0) {
    stop("Invalid state name or abbreviation")
  }

  return(result)
}
```

```
get_state_code("North Carolina")
```

```
[1] "37"
```

```
get_state_code("NC")
```

```
[1] "37"
```

Input Validation

We want to be able to fail early and give pertinent information back to the user if inputs are wrong or if an error occurs. These validation checks serve that purpose. If an input is wrong, we can often display to the user which input was incorrect and what the valid options are.

Ensure that the year is between 2010 and 2022

```
# Year must be between 2010 and 2022.
validate_year <- function(year){

  if (!(year %in% 2010:2022))
    stop("Year must be between 2010 and 2022.")
}
```

```
validate_year(2000) |> catch_error()
```

```
Caught error: Year must be between 2010 and 2022.
```

Here we handle the numeric variables. PWGTP and at least one other valid numeric variable must be selected. We have our lists of valid inputs in the utility functions section for reference. We will check against these to ensure each variable in the input is valid.

```
# PWGTP and at least one other valid numeric variable must be selected
validate_numeric_vars <- function(numeric_vars) {

  # Not worried about case
  valid_numeric_vars <- toupper(get_valid_numeric_vars())
  numeric_vars <- toupper(numeric_vars)

  numeric_vars <- intersect(numeric_vars, valid_numeric_vars)
```

```
  if (length(numeric_vars) < 2 || !"PWGTP" %in% numeric_vars) {
    stop("PWGTP and at least one other numeric variable must be selected from: ",
         paste(valid_numeric_vars, collapse = ", "))
  }
}
```

```
validate_numeric_vars(c("Invalid")) |> catch_error()
```

```
Caught error: PWGTP and at least one other numeric variable must be selected from: AGEP, PWGT
```

In a similar fashion, we handle categorical variable inputs. We expect at least one categorical variable from the user. It should be noted, that we are not so much interested in the case of the variables. The URL does have a standard for case, but it does not cause an error in testing. Thus, we allow it for a better user experience.

```
# At least one valid categorical variable must be selected
validate_categorical_vars <- function(categorical_vars) {

  # Not worried about case
  valid_categorical_vars <- toupper(get_valid_categorical_vars())
  categorical_vars <- toupper(categorical_vars)

  categorical_vars <- intersect(categorical_vars, valid_categorical_vars)

  if (length(categorical_vars) < 1) {
    stop("At least one valid categorical variable must be selected from: ",
         paste(valid_categorical_vars, collapse = ", "))
  }
}
```

```
validate_categorical_vars(c("Invalid")) |> catch_error()
```

```
Caught error: At least one valid categorical variable must be selected from: SEX, FER, HHL, H
```

validate_geography_and_subset

Geography and subset are a bit more complicated, but the general idea still applies. We want to check the input against valid lists. Firstly we need to check geography. Given a valid

geography, there are valid subsets that can be accepted. The subset can be NULL for any choice of geography.

Selecting the State geography in particular is worth mentioning. We developed a helper for states because we wanted to allow the user to provide a state name or abbreviation instead of a state code. In this way it is handled slightly different from the other geographies.

- a quick note that get_cat_refs could potentially be implemented here as well to refactor the hardoded region and divisions

```
# Geography & Subset Together
validate_geography_and_subset <- function(geography, subset) {

  # Handle case
  valid_geography_levels <- tolower(get_valid_geography_levels())
  geography <- tolower(geography)

  # Validate the geography
  if (!(geography %in% valid_geography_levels)) {
    stop("Invalid geography level. Must be one of: ",
         paste(valid_geography_levels, collapse = ", "))
  }

  # If geography is "all", subsetting is not allowed
  if (geography == "all" && !is.null(subset)) {
    stop("Subsetting is not allowed when geography is 'All'.")
  }

  valid_region_division_options <- list(
    region = tolower(c("Northeast", "Midwest", "South", "West")),
    division = tolower(c("New England", "Middle Atlantic",
                         "East North Central", "West North Central",
                         "South Atlantic", "East South Central",
                         "West South Central", "Mountain", "Pacific"))
  )

  # Check for region and division
  if (geography %in% c("region", "division")) {
    if (!(tolower(subset) %in% valid_region_division_options[[geography]])) {
      stop("Invalid ", geography, ". Must be one of: ",
           paste(valid_region_division_options[[geography]], collapse = ", "))
    }
  }
```

```
  # Handle State geography
  if (geography == "state") {
    # get_state_code will err out if invalid
    state_code <-  get_state_code(subset)
  }
}
```

```
validate_geography_and_subset("region", "Invalid") |> catch_error()
```

```
Caught error: Invalid region. Must be one of: northeast, midwest, south, west
```

```
validate_geography_and_subset("Invalid", NULL) |> catch_error()
```

```
Caught error: Invalid geography level. Must be one of: all, region, division, state
```

validate_url_response

It was discovered that some URL's do not return data. This caused errors down the line when the functions used to process the data were expecting their own inputs. To handle this issue more gracefully, we built a check on the response from the API.

- We first want to check that the status was a success. If we are dealing with more than just empty data, we can actually display the error message to further diagnose the issue.
- If the request was a success, we also check that the API returned anything at all for us to process. If nothing came back, there is no need to proceed.
- It was also discovered that the API might reject or return an error message even if the request was "successful". In such a case, we shold notify the user of this as well.

```
# Check we got something from the API using GET(URL)
validate_url_response <- function(response) {

  is_success <- http_status(response)$category == "Success"

  response_content <- content(response, as = "text")
  has_content <- !is.null(response_content) && nchar(response_content) > 0

  error_in_content <- str_detect(tolower(response_content),
                                 "error|not found|invalid|rejected")
```

```
  if (!is_success || !has_content || error_in_content) {
    stop("API request failed: ",
           if (!is_success) {
             http_status(census_raw)$message
           } else if (!has_content) {
             "Empty response from API."
           } else {
             "Error returned from API"
           }
         )
  }

  # print to console to show progression in request
  print("API request successful")
}
```

```
validate_url_response(httr::GET("https://api.census.gov/invalid")) |>
  catch_error()
```

```
Caught error: API request failed: Error returned from API
```

Building the URL

Here we reach our first undertaking after validating inputs. We need to construct a URL from the user input.

- We ran into some issues with fetching data from various years. It was discovered that by altering our base url dependent upon the year provided, we were able to query recent years and older records in different manners alleviating the issue. This is seen in the *dataset_type* in the code below.
- We then piece together all the desired variables to append to the base URL.
- Geography and subsets are at the end. Here we very much leverage the helper/utility functions that we defined above.

```
# Build a valid URL for the Census API
build_query_url <- function(year = 2022,
                            numeric_vars = c("AGEP", "PWGTP"),
                            categorical_vars = c("SEX"),
                            geography = "State",
```

```
                            subset = "CO") {

  dataset_type <- ifelse(year == 2021 || year == 2022, "acs1", "acs5")

  base_url <- paste0("https://api.census.gov/data/",
                     year, "/acs/", dataset_type, "/pums?")


  # Handle numeric and categorical inputs
  query_vars <- c(numeric_vars, categorical_vars)
  query_string <- paste0("get=", paste(query_vars, collapse = ","))

  # Handle geography levels ("All" will require no 'for' clause)
  geography_query <- ""

  if (geography != "All") {

    # Subsets need to be numeric codes. If null will return *
    subset <- get_subset_code(geography, subset)

    geography_query <- paste0("for=", gsub(" ", "%20", geography), ":", subset)
  }

  # Concatenate base_url, query_string, and geography_query
  final_url <- paste0(base_url, query_string)

  if (geography_query != "") {
    final_url <- paste0(final_url, "&", geography_query)
  }

  # print to console to show progression in request
  cat("URL: ", final_url)

  return(final_url)
}
```

```
test_url <- build_query_url()
```

```
URL:  https://api.census.gov/data/2022/acs/acs1/pums?get=AGEP,PWGTP,SEX&for=State:08
```

Processing the Response

query_census_with_url

We began with the instructions from the lecture to query the API and transform the JSON string into a data frame with the raw data

- A series of helper functions were developed to keep better track of what each step within this function was doing, which helped to focus on small tasks rather than getting overwhelmed by everything that had to be done to get the final clean data
- The data frame that came out of the JSON-to-data-frame helper had all the data in columns, but everything was character, so a final helper function takes care of cleaning the data

```
query_census_with_url <- function(url) {

  # retrieve data in list form from API
  census_raw <- httr::GET(url)

  # check that data was returned
  validate_url_response(census_raw)

  # call helper function to turn API raw data into a raw tibble
  census_raw_tbl <- json_to_raw_tbl_helper(census_raw)

  # call helper function to clean tibble
  census_clean_tbl <- process_census_data(census_raw_tbl)

  # return final clean tibble
  return(census_clean_tbl)
}
```

json_to_raw_tbl_helper

This is a helper function for query_census_with_url, and puts json stuff into a raw tibble. All variables are character

- The data frame that was created from JSON had the column names in the first row, so a step was included to grab the first row as the column names, and take the rest of the rows as the observations

```
json_to_raw_tbl_helper <- function(census_raw) {

  # convert JSON string raw data to data frame (first row are column names)
  parsed_census <- as.data.frame(fromJSON(rawToChar(census_raw$content)))

  # convert to a tibble, use 1st row from raw df as column names
  census_tbl <- as_tibble(parsed_census[-1,])
  colnames(census_tbl) <- toupper(parsed_census[1,])

  # return final tibble
  return(census_tbl)
}
```

process_census_data

This function is responsible for taking the character data in the tibble, and converting them to the appropriate data types.

- It was straight-forward to coerce categorical fields into factors, and the regular numeric fields into numeric.
- It took a lot of debugging and research to get the time fields right.
  - At first, they were converted to factors since JWAP/JWDP were both also initially considered both numerical and categorical variables.
  - However, we noticed a lot of values were close to midnight. Further investigation found that a lot of values that were actually 0 were getting changed to 1 somewhere along the way.
  - We looked to see if there was something wrong with the join to the time reference table (which provided a time for each time code), and it looked right.
  - However, we discovered that when they were coerced directly from factors to numeric, all of the values changed, since what we were changing were actually the cardinal levels and not the level description. So "0" was changing to 1 because it was the lowest (first) level. This of course meant that *all* of the values were now wrong.
  - Since Dr. Post confirmed that JWAP/JWDP didn't need to be categorical values, we removed the step that was converting them to factors, and the values now converted (from character) as expected.

```
process_census_data <- function(census_data_tbl) {

  # if state is a column, rename as ST
```

```r
if ("STATE" %in% names(census_data_tbl)) {
  names(census_data_tbl)[names(census_data_tbl) == "STATE"] <- "ST"
}

# retrieve valid categorical variables
cat_vars <-
  get_valid_categorical_vars() |>      # get all valid categorical variables
  c("DIVISION", "REGION", "ST") |>     # append regional categories
  intersect(names(census_data_tbl))    # keep only values in the data set

# convert categorical variables to actual descriptive values, and as factors
for (var in cat_vars){
  census_data_tbl[var] <- convert_cat_code_to_description(census_data_tbl[var])
}

# retrieve valid numeric vars, keeping only the ones that exist in
# the input raw data (note JWAP and JWDP will still need to be changed to times)
num_vars <-
  get_valid_numeric_vars() |>
  intersect(names(census_data_tbl))

# turn vars into numeric values in the tibble
for (var in num_vars){
  census_data_tbl[[var]] <- as.numeric(census_data_tbl[[var]])
}

# collect the time variables to convert
time_vars <-
  c("JWAP", "JWDP") |>
  intersect(names(census_data_tbl))

# call helper function to convert time codes to numeric time (won't run if
# time_vars is empty)
for (time_code in time_vars) {
  census_data_tbl <- convert_num_code_to_time(census_data_tbl, time_code)
}

# Assign class for custom methods
class(census_data_tbl) <- c("census", class(census_data_tbl))

# return clean tibble
return(census_data_tbl)
```

```
}
```

convert_num_code_to_time

This is another helper function to take the numerical time codes and change them to a meaningful time

- It starts with another helper function to get a reference tibble that provides the key-value pairs for each time code and time
- It takes the census data and left joins to the time reference table, and replaces the JWAP raw value with the actual time. Using a variable as one of the keys in the "by" argument proved elusive, so we kept it as a natural join. It's not ideal, but it does work. It meant the names of the primary and foreign keys in each table had to match (i.e. JWAP or JWDP), and that no other columns between the two tables could share the same name.
- The final step was to replace the values in JWAP/JWDP with the correct time from the 2nd table, and drop the now duplicate field that had been pulled in (named JWAP_clean or JWDP_clean)

```
convert_num_code_to_time <- function(census_data_tbl, time_code) {

  # get time references from API`
  times_reference <- get_time_refs(time_code)

  # join new JWAP/JWDP to table with proper times
  census_data_tbl <-
    census_data_tbl |>
    left_join(times_reference) # natural join on time code

  # assign the cleaned time values to the JWAP/JWDP column
  census_data_tbl[time_code] <- census_data_tbl[paste0(time_code, "_clean")]

  # Drop the extra column from the time reference table
  census_data_tbl <- census_data_tbl |>
    select(-one_of(paste0(time_code, "_clean")))

  return(census_data_tbl)
}
```

convert_cat_code_to_description

The job of this function is to convert the raw values for categorical variables in the data, and change them to the meaningful descriptive values using a reference table

- Since it appeared sometimes raw data seemed to omit leading zeroes, we decided to remove all leading zeroes to ensure the joins matched values appropriately. This was accomplished by coercing the character values to numeric, and then back to character.

```
# take categorical raw value and convert to descriptive value, and make a factor
convert_cat_code_to_description <- function(data_column) {

  # get the variable name that has to be looked up
  var <- colnames(data_column)

  # get time references from API`
  cat_reference <- get_cat_refs(var)

  # remove leading zeroes (census raw data sometimes have them, sometimes don't)
  cat_reference[[var]] <- as.character(as.numeric(cat_reference[[var]]))

  # remove leading zeroes from the data column too
  data_column[[var]] <- as.character(as.numeric(data_column[[var]]))

  # join new lookup table to the data column with proper values
  data_column <-
    data_column |>
    left_join(cat_reference) # natural join on coded value

  # return new data column with descriptive values, as factor
  return(as.factor(data_column[[2]]))
}
```

Generic Class Functions

Generic functions were created to automatically summarize and plot certain returned data. These are the two additional interfaces that we built for the user. The expect a tibble from a successful API call.

Summary

By default, it will summarize all numeric variables (other than PWGTP) and all categorical variables found in the tibble provided. The user is able to specify the variables they'd like in particular as additional arguments if all are not required.

- Numeric variables will have their weighted sample means and standard deviations calculated.
- Categorical variables will have their level counts displayed.
- Time variables are considered numeric here, but they need to be handled with some care. We convert them from their HMS form into *seconds from midnight* to perform calculations on them. We then convert back to HMS so that they are in a more friendly format.
- There are loops in this function that traverse across the columns of data and summarize them accordingly. Within each iteration of the loop, we utilize *vectorized functions* to process and make calculations across all the observations in the vector of values for that column/variable. Since there are only a small number of variables possible, these loops were left intact for readability. This however, could be a future refactoring opportunity as we might wish to avoid loops in R altogether.

```
# Summary Function for Census Class
summary.census <- function(data,
                           numeric_vars = NULL,
                           categorical_vars = NULL) {

  # Determine the variables that are actually in the dataset
  valid_numeric_vars <- get_valid_numeric_vars()
  valid_categorical_vars <- get_valid_categorical_vars()

  data_names <- toupper(names(data))

  numeric_vars_in_data <- intersect(data_names, valid_numeric_vars)
  categorical_vars_in_data <- intersect(data_names, valid_categorical_vars)

  # Default: Summarize all numeric variables except PWGTP in dataset
  if (is.null(numeric_vars)) {
    numeric_vars <- numeric_vars_in_data[numeric_vars_in_data != "PWGTP"]
  } else {
    # otherwise filter only for those provided
    numeric_vars <- intersect(toupper(numeric_vars), numeric_vars_in_data)
  }
```

```r
# Default: Summarize all categorical variables in dataset
if (is.null(categorical_vars)) {
  categorical_vars <- categorical_vars_in_data
} else {
  # otherwise filter only for those provided
  categorical_vars <- intersect(toupper(categorical_vars),
                                categorical_vars_in_data)
}

weight <- data$PWGTP
summary_list <- list()

# Summarize numeric variables
for (var in numeric_vars) {

  # Check if the variable is a time variable
  is_time_var <- var %in% c("JWAP", "JWDP")

  if (is_time_var) {
    # Convert time to seconds
    numeric_vector <- as.numeric(data[[var]])
  } else if (is.numeric(data[[var]])) {
    numeric_vector <- data[[var]]
  } else {
    stop("Unexpected non-numeric variable: ", var)
  }

  # Calculate weighted mean and standard deviation
  weighted_sample_mean <- sum(numeric_vector * weight, na.rm = TRUE) /
                          sum(weight, na.rm = TRUE)
  sample_sd <- sqrt(sum((numeric_vector^2) * weight, na.rm = TRUE) /
                    sum(weight, na.rm = TRUE) - weighted_sample_mean^2)

  if (is_time_var) {
    # Convert the results back to hms
    weighted_sample_mean <- as_hms(weighted_sample_mean)
    sample_sd <- as_hms(sample_sd)
  }

  # Store the results
  summary_list[[var]] <- list(
    mean = weighted_sample_mean,
```

```
      sd = sample_sd
    )
  }

  # Summarize categorical variables
  for (var in categorical_vars) {

    counts <-  data |> count(.data[[var]])

    # Store the results
    summary_list[[var]] <- list(
      counts = counts
    )
  }

  return(summary_list)
}
```

Plotting

A Generic plot() function was also created for a census class tibble. This requires the user to specify one categorical variable and one numeric variable for plotting purposes.

- Here we check the inputs are in the data and filter out and missing entries before plotting.
- The biggest point of note is the use of sampling. Some of the data sets that come through the API can contain millions of records. Particularly if the region is set to *All*. Due to this circumstance, the plotting was found to take a considerable amount of time. To address this, we simply evaluate the number of rows in the data, and if necessary, take a random sample for plotting purposes.

```
# Plotting Function for Census Class
plot.census <- function(data,
                        numeric_var,
                        categorical_var,
                        sample_size = 100000) {

  # Check User inputs
  for (var in c(numeric_var, categorical_var, "PWGTP")) {
    if (!var %in% names(data)) {
      stop(paste("The variable", var, "is not present in the dataset. ",
                 "Select from: ", paste(names(data), collapse = ", ")))
```

```
    }
  }

  # Remove NA records
  data <- data |>
    filter(!is.na(.data[[numeric_var]]) &
             !is.na(.data[[categorical_var]]))

  # If the dataset is large, take a random sample
  if (nrow(data) > sample_size) {
    message(nrow(data), " records found in dataset. Sampled ",
            sample_size, " rows for plotting.")
    set.seed(123)
    data <- data |> sample_n(sample_size)
  }

  # Plot with ggplot2
  ggplot(data,
         aes(x = get(categorical_var),
             y = get(numeric_var),
             weight = PWGTP)) +
    geom_boxplot(fill = "lightsteelblue", color = "black") +
    labs(
      title = paste("Boxplot of", numeric_var, "by", categorical_var),
      x = categorical_var,
      y = numeric_var
    ) +
    theme_minimal() +
    theme(axis.text.x = element_text(angle = 45, hjust = 1))
}
```

Results

Single Year

First, we demonstrate the output of the main function using default values.

```
test_with_defaults <- get_data_tibble_from_census_api()
```

```
URL:  https://api.census.gov/data/2022/acs/acs1/pums?get=AGEP,PWGTP,SEX&for=State:08[1] "API
```

```
Joining with `by = join_by(SEX)`
Joining with `by = join_by(ST)`
```

```
test_with_defaults |>
  head() |>
  knitr::kable(align = 'c')
```

AGEP	PWGTP	SEX	ST
18	70	Female	Colorado/CO
35	41	Male	Colorado/CO
40	35	Male	Colorado/CO
20	30	Female	Colorado/CO
14	4	Male	Colorado/CO
28	75	Female	Colorado/CO

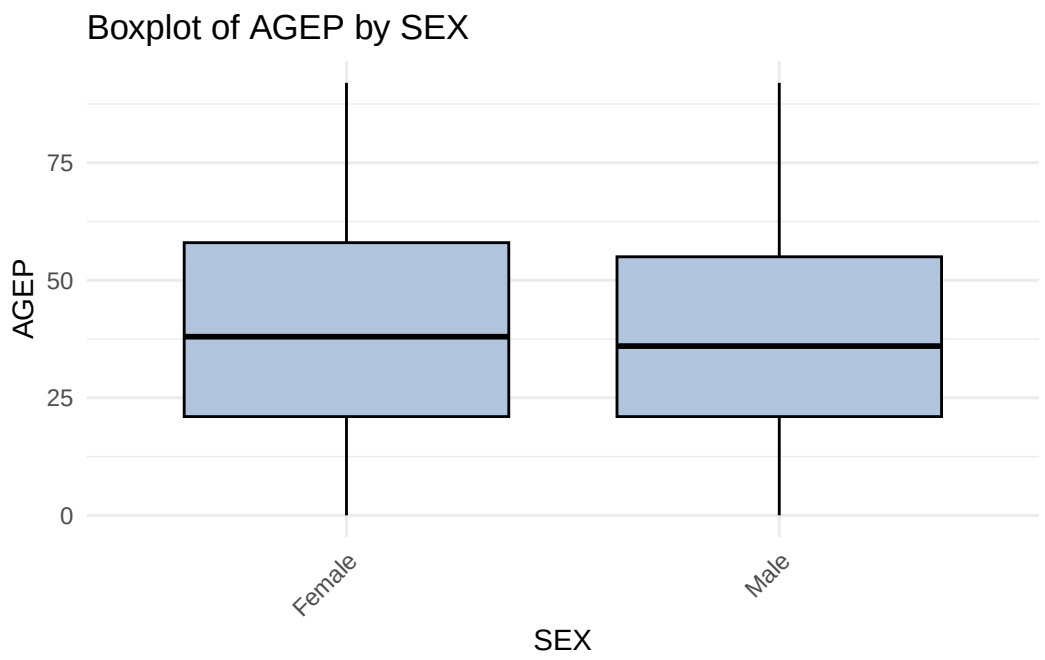
Without supplying specific categorical and numeric values, note how the summary function returns results for all numeric variables except for the weight, along with all categorical variables.

```
summary.census(test_with_defaults)
```

```
$AGEP
$AGEP$mean
[1] 38.80219

$AGEP$sd
[1] 22.42793


$SEX
$SEX$counts
# A tibble: 2 x 2
  SEX        n
  <fct>  <int>
1 Female 29901
2 Male   29940
```

Recall that our plot requires one numeric and one categorical variable, so since our data in this example were generated from defaults, we used Age for the numeric variable, and Sex for the categorical variable.

```
plot.census(test_with_defaults, "AGEP", "SEX")
```



For our investigative example, we have chosen to examine how arrival time to work (JWAP) is related to household language (HHL) in the state of North Carolina for the year 2015.

```
test_with_vars <- get_data_tibble_from_census_api(2015,
                                    c("PWGTP", "JWAP"),
                                    c("HHL"),
                                    "state",
                                    "nc")
```

```
URL:  https://api.census.gov/data/2015/acs/acs5/pums?get=PWGTP,JWAP,HHL&for=state:37[1] "API

Joining with `by = join_by(HHL)`
Joining with `by = join_by(ST)`
Joining with `by = join_by(JWAP)`
```

```
test_with_vars |>
  head() |>
  knitr::kable(align = 'c')
```

PWGTP	JWAP	HHL	ST
3	NA	English Only	North Carolina/NC
3	NA	English Only	North Carolina/NC
5	NA	English Only	North Carolina/NC
6	NA	English Only	North Carolina/NC
13	04:07:00	Spanish	North Carolina/NC
12	NA	Spanish	North Carolina/NC

The summary for our example tibble appears below.

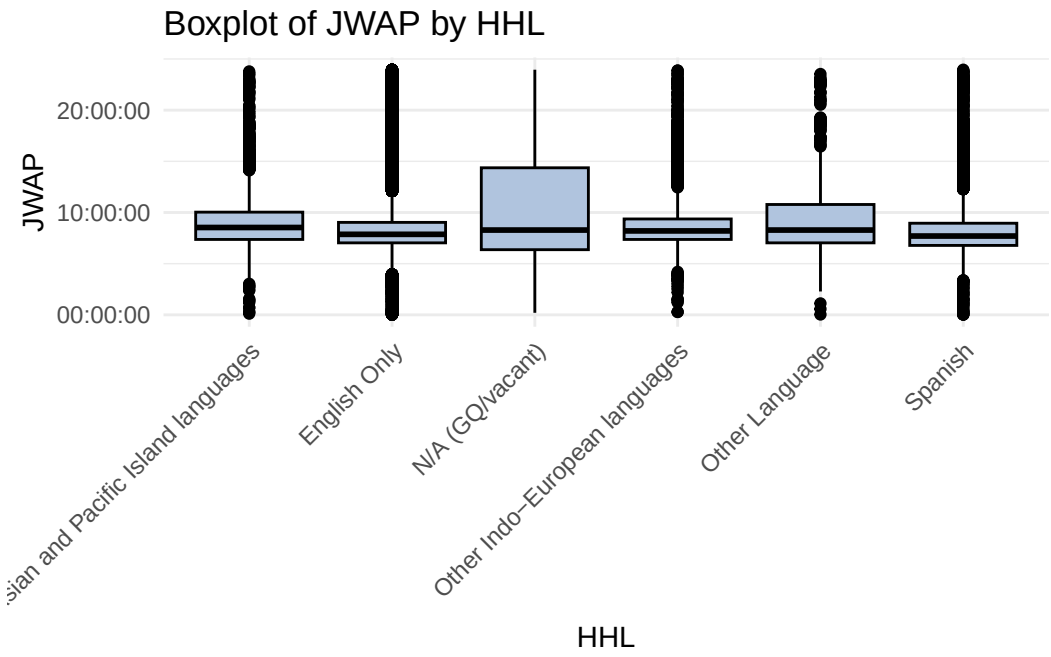
```
summary.census(test_with_vars)
```

```
$JWAP
$JWAP$mean
03:45:33.0226

$JWAP$sd
04:58:32.467112


$HHL
$HHL$counts
# A tibble: 6 x 2
  HHL                                    n
  <fct>                              <int>
1 Asian and Pacific Island languages  8659
2 English Only                      405007
3 N/A (GQ/vacant)                    24922
4 Other Indo-European languages      11492
5 Other Language                      2921
6 Spanish                            33903
```

The boxplots show that the center of the distributions are all very similar, regardless of household language. There is more variation in the interquartile ranges for the different languages.

```
plot.census(test_with_vars, "JWAP", "HHL")
```

```
198175 records found in dataset. Sampled 1e+05 rows for plotting.
```



Multi Year

To showcase our multi-year function, we will view the gross rent as a percentage of household income over the last 12 months (GRPIP) in Wyoming, from 2017 and 2018, by whether they have given birth within the last 12 months (FER). We display an aggregate of the results, so that the stratification by year can be seen.

```
test_multi_year <- query_census_multiple_years(2017:2018,
                                               c("PWGTP", "GRPIP"),
                                               "FER",
                                               geography = "State",
                                               subset = "Wyoming")
```

```
URL:  https://api.census.gov/data/2017/acs/acs5/pums?get=PWGTP,GRPIP,FER&for=State:56[1] "API

Joining with `by = join_by(FER)`
Joining with `by = join_by(ST)`
```

```
URL:  https://api.census.gov/data/2018/acs/acs5/pums?get=PWGTP,GRPIP,FER&for=State:56[1] "API

Joining with `by = join_by(FER)`
Joining with `by = join_by(ST)`
```

```
test_multi_year |>
  group_by(Year, ST, FER) |>
  summarize(mean_GRPIP = sum(PWGTP*GRPIP)/sum(PWGTP)) |>
  knitr::kable(align = 'c')
```

```
`summarise()` has grouped output by 'Year', 'ST'. You can override using the
`.groups` argument.
```

Year	ST	FER	mean_GRPIP
2017	Wyoming/WY	N/A (less than 15 years/greater than 50 years/ male)	7.610905
2017	Wyoming/WY	No	10.687256
2017	Wyoming/WY	Yes	13.154172
2018	Wyoming/WY	N/A (less than 15 years/greater than 50 years/ male)	7.515208
2018	Wyoming/WY	No	10.709837
2018	Wyoming/WY	Yes	11.894927

Conclusion

In the preceding sections, we have outlined our collaborative process for querying API endpoints, validating our inputs, processing the data we receive, and presenting meaningful summaries and plots. Throughout the project, we have utilized the tidyverse suite of packages for their efficiency and their useful data structure, the tibble. We were also mindful of opportunities where the same functions could be reused in multiple places. This minimizes redundant code blocks, enhances readability, and lays groundwork for potential future expansion of the project's scope (a tenet of defensive programming). Encapsulating smaller functions that handle just one or a few tasks inside larger ones simplified and focused our work. This model, plus working with a git repository, enabled us to effectively work as a team towards successful completion of this project.