Project 1 - API Census Data

Patrick Seebold

Setting up an API for Census Data

In this project, I will build an API to allow a user to access census data from online. First, let's grab the appropriate packages.

```
library('jsonlite') # for handling json data from our APIs
```

```
Warning: package 'jsonlite' was built under R version 4.3.3
```

```
library('tidyverse') # for those beautiful tibble functions
```

```
-- Attaching core tidyverse packages ------------------------ tidyverse 2.0.0 --
v dplyr     1.1.2     v readr     2.1.4
v forcats   1.0.0     v stringr   1.5.0
v ggplot2   3.4.2     v tibble    3.2.1
v lubridate 1.9.2     v tidyr     1.3.0
v purrr     1.0.1
-- Conflicts ------------------------------------------ tidyverse_conflicts() --
x dplyr::filter()  masks stats::filter()
x purrr::flatten() masks jsonlite::flatten()
x dplyr::lag()     masks stats::lag()
i Use the conflicted package (<http://conflicted.r-lib.org/>) to force all conflicts to becor
```

```
library('lubridate') # for dealing with the time variables
library('ggplot2') # for plotting
```

Before we start writing the functions that let our users specify custom APIs, let's go ahead and test out the 'usual method' covered in class to get make an API call. This will let us confirm that we can get the data we want without a key and investigate the format it returns our data in

```r
d = httr::GET(url =
    "https://api.census.gov/data/2022/acs/acs1/pums?get=SEX,PWGTP,MAR&SCHL=24")
str(d)
```

```
List of 10
 $ url        : chr "https://api.census.gov/data/2022/acs/acs1/pums?get=SEX,PWGTP,MAR&SCHL=24
 $ status_code: int 200
 $ headers    :List of 12
  ..$ cache-control               : chr "max-age=60, must-revalidate"
  ..$ access-control-allow-origin : chr "*"
  ..$ access-control-allow-methods: chr "GET,POST"
  ..$ access-control-allow-headers: chr "Origin, X-Requested-With, Content-Type, Accept"
  ..$ referrer-policy             : chr "same-origin"
  ..$ content-security-policy     : chr "default-src 'self'; img-src 'self' data:; object-src
  ..$ x-content-type-options      : chr "nosniff"
  ..$ content-type                : chr "application/json;charset=utf-8"
  ..$ date                        : chr "Sat, 05 Oct 2024 17:05:50 GMT"
  ..$ strict-transport-security   : chr "max-age=31536000"
  ..$ set-cookie                  : chr "TS010383f0=01283c52a4837b30cb4a6f1cdfe85d9af7ed04ed
  ..$ transfer-encoding           : chr "chunked"
  ..- attr(*, "class")= chr [1:2] "insensitive" "list"
 $ all_headers:List of 1
  ..$ :List of 3
  .. ..$ status : int 200
  .. ..$ version: chr "HTTP/1.1"
  .. ..$ headers:List of 12
  .. .. ..$ cache-control               : chr "max-age=60, must-revalidate"
  .. .. ..$ access-control-allow-origin : chr "*"
  .. .. ..$ access-control-allow-methods: chr "GET,POST"
  .. .. ..$ access-control-allow-headers: chr "Origin, X-Requested-With, Content-Type, Accept
  .. .. ..$ referrer-policy             : chr "same-origin"
  .. .. ..$ content-security-policy     : chr "default-src 'self'; img-src 'self' data:; obje
  .. .. ..$ x-content-type-options      : chr "nosniff"
  .. .. ..$ content-type                : chr "application/json;charset=utf-8"
  .. .. ..$ date                        : chr "Sat, 05 Oct 2024 17:05:50 GMT"
  .. .. ..$ strict-transport-security   : chr "max-age=31536000"
  .. .. ..$ set-cookie                  : chr "TS010383f0=01283c52a4837b30cb4a6f1cdfe85d9af7e
  .. .. ..$ transfer-encoding           : chr "chunked"
  .. .. ..- attr(*, "class")= chr [1:2] "insensitive" "list"
 $ cookies    :'data.frame':    1 obs. of  7 variables:
  ..$ domain    : chr "#HttpOnly_.api.census.gov"
  ..$ flag      : logi TRUE
```

```
  ..$ path      : chr "/"
  ..$ secure    : logi TRUE
  ..$ expiration: POSIXct[1:1], format: "Inf"
  ..$ name      : chr "TS010383f0"
  ..$ value     : chr "01283c52a4837b30cb4a6f1cdfe85d9af7ed04ed7729c49694cc09292245688851c123
 $ content    : raw [1:937508] 5b 5b 22 53 ...
 $ date       : POSIXct[1:1], format: "2024-10-05 17:05:50"
 $ times      : Named num [1:6] 0 0.00386 0.10514 0.28417 0.54276 ...
  ..- attr(*, "names")= chr [1:6] "redirect" "namelookup" "connect" "pretransfer" ...
 $ request    :List of 7
  ..$ method    : chr "GET"
  ..$ url       : chr "https://api.census.gov/data/2022/acs/acs1/pums?get=SEX,PWGTP,MAR&SCHL=
  ..$ headers   : Named chr "application/json, text/xml, application/xml, */*"
  .. ..- attr(*, "names")= chr "Accept"
  ..$ fields    : NULL
  ..$ options   :List of 2
  .. ..$ useragent: chr "libcurl/7.84.0 r-curl/5.0.1 httr/1.4.6"
  .. ..$ httpget  : logi TRUE
  ..$ auth_token: NULL
  ..$ output    : list()
  .. ..- attr(*, "class")= chr [1:2] "write_memory" "write_function"
  ..- attr(*, "class")= chr "request"
 $ handle     :Class 'curl_handle' <externalptr>
 - attr(*, "class")= chr "response"
```

Looks like it worked! We now have a sample dataset in json format. We can use the jsonlite package to process this into a tibble, which will be more user-friendly for the our tidyverse methods.

```
parsed = fromJSON(rawToChar(d$content))
head(parsed) # check that the conversion out of JSON format worked
```

```
     [,1]  [,2]    [,3]  [,4]
[1,] "SEX" "PWGTP" "MAR" "SCHL"
[2,] "2"   "6"     "5"   "24"
[3,] "2"   "23"    "2"   "24"
[4,] "1"   "23"    "3"   "24"
[5,] "1"   "80"    "5"   "24"
[6,] "1"   "16"    "1"   "24"
```

Great, we got the data into a character matrix using the method we discussed in class! However, we want it to be in a tibble, not in a matrix. Let's go ahead and make a function that will let us convert GET() output into tibble format:

```
tibble_maker = function(input_url, tibble_name){
  d = httr::GET(input_url) # get the data from the URL
  parse = fromJSON(rawToChar(d$content)) # convert this our of JSON format
  colnames(parse) = parse[1,] # names are in the first row, so grab these
  tibble_name = as_tibble(parse, .name_repairs = 'unique') # specify rownames are
                                                           # in matrix
  tibble_name = tibble_name[-1,]# drop the row which has the tibble names in it
  return(tibble_name)
}
```

We needed to do some extra edits in our function to make sure that we get the variable names in the right place, but after some trial and error, we got it! Now we test the function to confirm it works:

```
tibble_maker(
   "https://api.census.gov/data/2022/acs/acs1/pums?get=SEX,PWGTP,MAR&SCHL=24",
   test_tibble)
```

```
# A tibble: 44,079 x 4
   SEX   PWGTP MAR   SCHL
   <chr> <chr> <chr> <chr>
 1 2     6     5     24
 2 2     23    2     24
 3 1     23    3     24
 4 1     80    5     24
 5 1     16    1     24
 6 1     107   3     24
 7 2     10    5     24
 8 1     22    1     24
 9 2     127   5     24
10 2     46    5     24
# i 44,069 more rows
```

Making the customizable API Function:

Great, we now have the tibble_maker() function which will take API census data and convert it into the desired tibble format. Next, we can proceed to work on the function that allows a user to specify how to adjust their API call. We want to include:

- Year (2022 as default, 2010 - 2022 as options - note that 2020 is not available and thus excluded)

- Numeric Variables (AGEP, PWGTP as default. Options are AGEP, GASP, GRPIP, JWAP, JWDP, and JWMNP. PWGTP and one other Numeric variable must be returned)
- Categorical Variables (SEX as default. Options are FER, HHL, HISPEED, JWTRNS, SCH, SCHL, and SEX. One categorical variable must be returned)
- Location - STATE:19 as default, options are ALL, REGION, DIVISION, STATE.

```
one_year_api = function(year = 2022, num = 'AGEP', cat = 'SEX', loc_type= "STATE",
                        loc_num = "19"){

  # set the accepted values for each type of variable
  num_list = c('AGEP', 'GASP', 'GRPIP', 'JWAP', 'JWDP','JWMNP')
  cat_list = c('FER', 'HHL', 'HISPEED', 'JWTRNS', 'SCH', 'SCHL','SEX')
  loc_list = c('ALL', 'REGION','DIVISION', 'STATE','')

  # check that provided values match options
  if (year != 2020){
    if (between(year,2010,2022)){
      print('Year is in range')} else {
        stop('Year must be between 2010 - 2022, excluding 2020')
      }
    }else {
        stop('Data not available for 2020. Please select other year 2010 - 2022')
    }

  if (num %in% num_list){
    print('Numerical variable selected')} else {
      stop('Make sure numerical variable is in list: AGEP, GASP, GRPIP, JWAP,
           JWDP, JWMNP')
    }

  if (cat %in% cat_list){
    print('Categorical variable selected')} else {
      stop('Make sure the (1) categorical variable is in list: FER, HHL, HISPEED,
           JWTRNS, SCH, SCHL,SEX')
    }

  if (loc_type%in% loc_list){
    print('Location variable selected')} else {
      stop('Location must be (1) from list: ALL, DIVISION, REGION, STATE')
  }
```

```
  # now that we have everything appropriately selected, we can write the API call
  if (loc_type== "ALL"){ # if we don't specify location, API call is simpler
  api_url = as.character(paste('https://api.census.gov/data/',
                               year,'/acs/acs1/pums?get=',num,',',
                               cat,',PWGTP', sep=''))
  } else { # if location != ALL, set location at end of API call
  api_url = as.character(paste('https://api.census.gov/data/',
                               year,'/acs/acs1/pums?get=',num,',',
                               cat,',PWGTP','&for=',tolower(loc_type),
                               ":", loc_num, sep=''))
  }
  returned_tibble = tibble_maker(api_url, test_tibble)
  return(returned_tibble)
}
test = one_year_api()
```

```
[1] "Year is in range"
[1] "Numerical variable selected"
[1] "Categorical variable selected"
[1] "Location variable selected"
```

```
str(test)
```

```
tibble [33,586 x 4] (S3: tbl_df/tbl/data.frame)
 $ AGEP : chr [1:33586] "18" "69" "64" "22" ...
 $ SEX  : chr [1:33586] "1" "2" "1" "1" ...
 $ PWGTP: chr [1:33586] "36" "80" "25" "10" ...
 $ state: chr [1:33586] "19" "19" "19" "19" ...
```

Making the Helper Functions

Cool, now we have a function that gives us some customizable options for calling from the census API! However, our current tibble_maker function won't handle all the cool conversions we want to be able to do. Let's write some functions that will handle the conversions of interest, which we can then integrate into our updated tibble_maker() function.

First, let's handle our numeric variables. We want to make all of these variables are changed into numerics, and JWAP/JWDP need to be converted to minutes since midnight (using interval midpoints for each given interval). The following functions will handle these changes:

```
# This function generates the comparison table for JWAP/JWDP based on which type
# the user specifies. This comparison df will be used for converting API data to
# the appropraite minutes-from-midnight form

generate_minute_conversion_reference = function(type){ # change given vector to time
  # first, grab the json appropriate for JWDP or JWAP, as specified by type input
  if (type == 'JWAP'){
      temp = httr::GET(
        "https://api.census.gov/data/2022/acs/acs1/pums/variables/JWAP.json")
    } else if (type == 'JWDP'){
      temp = httr::GET(
        "https://api.census.gov/data/2022/acs/acs1/pums/variables/JWDP.json")
    }
    #turn reference json  into a list
    temp_list = temp$content |> rawToChar() |>jsonlite::fromJSON()
    #grab just the names and values
    j = temp_list$values$item
    #reorder just so it is clearer
    j_values = j[sort(names(j))]
    parsed_j = c() # initialize an empty vector
    df = c()
    for (i in 1:length(j_values)){
      df[i] = j_values[[i]]
    }
    df = data.frame(df)
    df = df[-1,]
    df = data.frame(df)
    parsed_j = separate_wider_delim(df, cols = everything(), delim = " ",
                                    names_sep = "")
    parsed_j = parsed_j[-c(2,3,5)]
    df=c()
    df$a = hm(parsed_j$df1) # convert to hour minute notation!
    df$b = hm(parsed_j$df4)
    df = as.data.frame(df)
    df$dif = df$b - df$a # calculate the number of mins in each interval

    df$interval_length = as.numeric(df$dif) # make values numeric for easier math
    df$interval_length = df$interval_length/60 # put back into minutes
    df$interval_midpoint = df$interval_length/2 # this gives us the midpoint
    for (i in 1:nrow(df)){ # loop through data, calculating midnight from midnight
      if (i == 1){
        df$minutes_from_md_to_interval_start[i] = 0
```

```
        df$minutes_from_md_to_interval_midpoint[i] = df$interval_midpoint[i]
      } else {
      df$minutes_from_md_to_interval_start[i] = 1 +
        df$minutes_from_md_to_interval_start[i-1] +
        df$interval_length[i-1]
      df$minutes_from_md_to_interval_midpoint[i] =
        df$minutes_from_md_to_interval_start[i] + df$interval_midpoint[i]
      }
    }
    df = as.data.frame(df$minutes_from_md_to_interval_midpoint)
    colnames(df) = 'mins'
   # df = as.numeric(df) # make sure this is the right type!
    return(df)
} # this returns a reference df that we will use for converting JWAP/JWDP
  # values to minutes since midnight form


# this function takes the JWAP/JWDP column in our API data along with the
# comparison df generated from generate_minute_conversion_reference()
convert_minutes = function(input_col, comp){
    for (i in 1:length(input_col)){
      if (input_col[i] == 0){
        next # if the value is 0, we keep it at 0
      } else {
        val = input_col[i]
        input_col[i] = comp$mins[val]  # if value != 0, we replace value with the
                                       # appropriate number of minutes from
                                       # midnight in our reference df
      }
    }
  return(input_col)
}

# together, these functions let us convert our JWAP/JWDP values into minutes since
# midnight. Then, they simply need to replace the pre-existing variable!
```

Let's make one more helper function for all of our categorical variables. This will just take in the categorical variable given and convert the levels of that variable into the appropriate factor levels:

```
# convert factors to appropraite levels
convert_factors = function(input_col, var_name){ # change given vector to time
  # first, grab the json appropriate for variable
    temp = httr::GET(paste0(
      "https://api.census.gov/data/2022/acs/acs1/pums/variables/",
      var_name, ".json"))
    #turn reference json  into a list
    temp_list = temp$content |> rawToChar() |>jsonlite::fromJSON()
    #grab just the names and values
    j = temp_list$values$item
    #reorder just so it is clearer
    j_values = j[sort(names(j))]
    val = c() # kick up an empty vector
    for (i in 1:length(j_values)){
      val[i] = j_values[[i]]
    }
    if (var_name != "SEX"){
      output = factor(input_col, levels = as.character(0:(length(j_values)-1)),
                      labels = val)
    } else { # SEX has no 0 value, so we need to specify this separately
      output = factor(input_col, levels = c(1,2), labels = val)
    }
    return(output)
}

# This function will convert our categorical variables into lovely factors, using
# the names drawn from the online census data!
```

Updating our tibble_maker() function with helper functions

Now that we've written our helper functions for converting time and factor levels, we can look to update our tibble_maker() function. Our plan will be to update the tibble_maker() function so that we can still use the one_year_api() as it was above without further edits.

```
# let's recreate our tibble_maker() function with added helper function capability
tibble_maker = function(input_url, tibble_name){
  d = httr::GET(input_url) # get the data from the URL
  parse = fromJSON(rawToChar(d$content)) # convert this our of JSON format
  colnames(parse) = parse[1,] # names are in the first row, so grab these
  tibble_name = as_tibble(parse, .name_repairs = 'unique') # specify rownames
  tibble_name = tibble_name[-1,]# drop the row which has the tibble names in it
```

```
  # make sure our numeric types are correct before we continue; factors will be
  # converted later with our helper function
  tibble_name[[1]] = as.numeric(tibble_name[[1]])
  tibble_name[[3]] = as.numeric(tibble_name[[3]])

  # this is where we add the new components to adjust each var type!
  if ("JWAP" %in% names(tibble_name)){
    comp = generate_minute_conversion_reference("JWAP") # make comparison df
    tibble_name$JWAP = convert_minutes(tibble_name$JWAP, comp)
  }
  if ("JWDP" %in% names(tibble_name)){ # use same function for both time vars
    comp = generate_minute_conversion_reference("JWDP") # make comparisons df
    tibble_name$JWDP = convert_minutes(tibble_name$JWDP, comp)
  }

  cat_name = names(tibble_name[2])
  tibble_name[2] = convert_factors(tibble_name[[2]], cat_name)

  # set the unique class of our tibble for future functions:
  class(tibble_name) <- c("census", class(tibble_name))
  return(tibble_name)
}
# setup test url, with JWAP and SCHL as our num and cat variables
input_url =
  "https://api.census.gov/data/2022/acs/acs1/pums?get=JWAP,SCHL,PWGTP&for=state:19"
test_tibble = tibble_maker(input_url, test_tibble)
head(test_tibble) # looks good!
```

```
# A tibble: 6 x 4
   JWAP SCHL                                              PWGTP state
  <dbl> <fct>                                             <dbl> <chr>
1     0 Some college, but less than 1 year                   36 19
2     0 Regular high school diploma                          80 19
3     0 12th grade - no diploma                              25 19
4   907 1 or more years of college credit, no degree         10 19
5     0 Professional degree beyond a bachelor's degree      100 19
6     0 Master's degree                                      71 19
```

Create multi-year function

Awesome, now our function converts specified variables into the right kinds! Now let's make it so we can call multiple years at once. We'll assume that our users will be interested in comparing the same variables across years, so we will not build in the ability to search for different variables across different years. We'll output our data into a single tibble with the year as a new variable column:

```
l = list() # we'll put our one year dfs in here before combining
multi_year_api = function(years = c(2022), num = 'AGEP', cat = 'SEX',
                          loc_type= "STATE", loc_num = "19"){
                         # specify years as c(year1,year2,etc); 2022 by default
  x = length(years)
  tib = c()
  for (i in 1:x){
    tib = one_year_api(year = years[i])#, num, cat, loc_type, loc_num)
    tib$year = years[i] # add in the year column
    l[[i]] = tib # store these outside the loop
    }
  final_tibble = do.call(rbind, l) #use do.call so we can get all years in one tib!
}

# let's do a quick test:
test=multi_year_api(c(2022,2021,2019))
```

```
[1] "Year is in range"
[1] "Numerical variable selected"
[1] "Categorical variable selected"
[1] "Location variable selected"
[1] "Year is in range"
[1] "Numerical variable selected"
[1] "Categorical variable selected"
[1] "Location variable selected"
[1] "Year is in range"
[1] "Numerical variable selected"
[1] "Categorical variable selected"
[1] "Location variable selected"
```

```
nrow(test)
```

```
[1] 98879
```

```
# success! We won't print this out (very long!!)
```

Make Summary/Plotting Functions

Now, we have our completed functions. Finally let's look at some person-level outcomes using the summary/plotting functions provided. We gave our tibble_maker() outputs a new 'census' class, which we will use in our upcoming functions:

```
# set up the census summary function:
summary.census = function(tibble, num_var = "col1", cat_var = "col2"){
  if (num_var == "col1"){     # We use the known order of the tibble here to make
    num_var = names(tibble[1])# default values the num & cat variables in our input
  }
  if (cat_var == "col2"){
    cat_var = names(tibble[2])
  }
  mean = sum(tibble[[num_var]]*tibble$PWGTP)/sum(tibble$PWGTP)
  sd = sqrt(sum(tibble[[num_var]]^2*tibble$PWGTP)/(sum(tibble$PWGTP)-mean^2))
  cat_summary = table(tibble[[cat_var]])
  return(list(mean, sd, cat_summary))
}

plot.census = function(tibble, num_var = "col1", cat_var = "col2"){
  if (num_var == "col1"){     # We use the known order of the tibble here to make
    num_var = names(tibble[1])# default values the num & cat variables in our input
  }
  if (cat_var == "col2"){
    cat_var = names(tibble[2])
  }
  ggplot(tibble,
    aes(x = get(cat_var), y = get(num_var), weight = PWGTP)) +
    geom_boxplot()
}


data = one_year_api(year = 2011)
```

```
[1] "Year is in range"
[1] "Numerical variable selected"
[1] "Categorical variable selected"
[1] "Location variable selected"
```

```
summary.census(data)
```

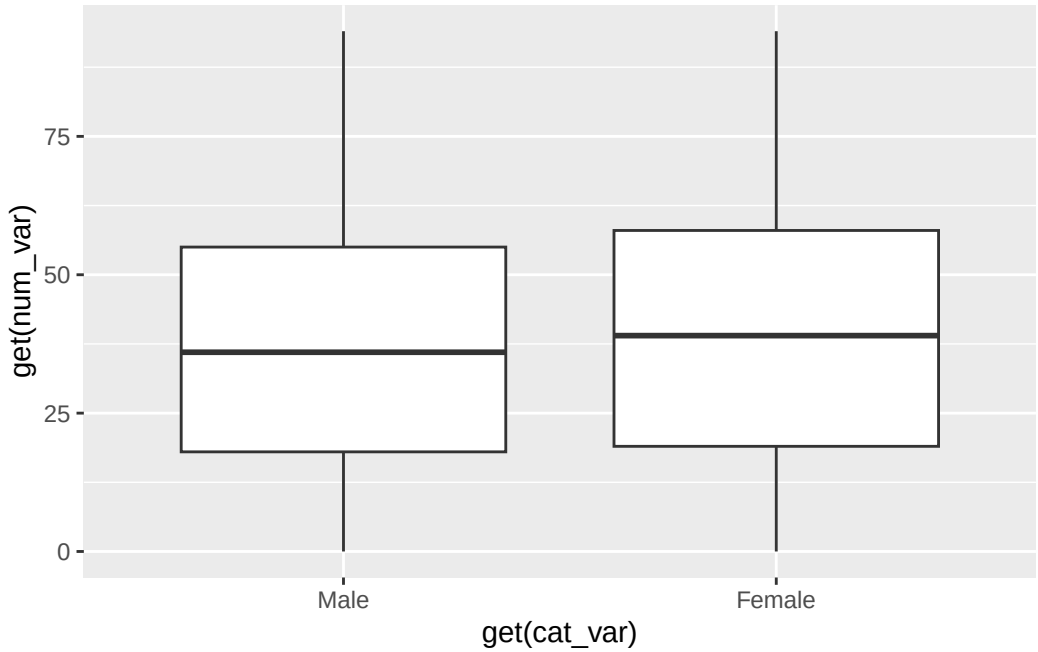
```
[[1]]
[1] 38.41486

[[2]]
[1] 45.07772

[[3]]

  Male Female
 15481  15984
```

```
plot.census(data)
```



From our example data above, we can see that the average age of our participants is ~38.4, and that the number of Males and Females is approximately an even split. Plotting these variables lets us visualize those trends more easily; although the Female boxplot is a little bit higher than the Male boxplot, the two look very similar.

In Conclusion

In conclusion, we have now produced a set of functions that will allow our users to submit a query to the online census database, retrieve data for select variables, and then plot/summarize those variables. This takes much of the heavy lifting out of a user's experience, as they need to only understand a small number of function inputs to successfully get outputs of interest!