Problem set 4

2025-10-05

In the next problem set, we plan to explore the relationship between COVID-19 death rates and vaccination rates across US states by visually examining their correlation. This analysis will involve gathering COVID-19 related data from the CDC's API and then extensively processing it to merge the various datasets. Since the population sizes of states vary significantly, we will focus on comparing rates rather than absolute numbers. To facilitate this, we will also source population data from the US Census to accurately calculate these rates.

In this problem set we will learn how to extract and wrangle data from the data US Census and CDC APIs.

1. Get an API key from the US Census at https://api.census.gov/data/key_signup.html. You can't share this public key. But your code has to run on a TFs computer. Assume the TF will have a file in their working directory named `census-key.R` with the following one line of code:

```
census_key <- "  "
```

Write a first line of code for your problem set that defines `census_key` by running the code in the file `census-key.R`.

```
## Your code here
census_key <- source("census-key.R")
# Seperate file which I didnt commit
```

2. The US Census API User Guide provides details on how to leverage this valuable resource. We are interested in vintage population estimates for years 2021 and 2022. From the documentation we find that the *endpoint* is:

```
url <- "https://api.census.gov/data/2021/pep/population"
# messy data
# base address for the Census Bureau API endpoint
```

Use the **httr2** package to construct the following GET request.

`https://api.census.gov/data/2021/pep/population?get=POP_2020,POP_2021,NAME&for=state:*&key=Y`

Create an object called `request` of class `httr2_request` with this URL as an endpoint. Hint: Print out `request` to check that the URL matches what we want.

```
library(httr2)
request <- request(url) |> #creates a new HTTP request object
  req_url_query(
    get = "POP_2020,POP_2021,NAME", #name of the query parameter being set.
     ↪  Based on Census API documentation, this is the standard parameter
     ↪  name for specifying the variables to retrieve:the population for
     ↪  2020, the population for 2021, and the name of the geographic area.
    `for` = "state:*", #retreiving geographic level, * is a wildcard that
     ↪  tells the Census API to retrieve data for all states
    key = census_key$value #The Census API requires a key for access (my key
     ↪  I put into the .R file)
  ) #adds query parameters to the URL of the request object. Query
 ↪  parameters are the key=value pairs that come after the ? in a web
 ↪  address.
request
```

3. Make a request to the US Census API using the `request` object. Save the response to and object named `response`. Check the response status of your request and make sure it was successful. You can learn about *status codes* here.

```
response <- request |> req_perform() # contains all the information received
 ↪  from the server, including the status code, content, and headers.
response$status_code
```

```
[1] 200
```

```
#performs or executes the HTTP request. It sends the request to the target
 ↪  server and waits for the server's reply.
#`req_perform()` performs a request, fetching results. It returns an HTTP
 ↪  response on success, or an error if the request fails or has an error
 ↪  status code.
#The status code is a three-digit number that tells  if the request was
 ↪  successful or if an error occurred ( 200 OK, 404 Not Found, 500 Server
 ↪  Error).
```

4. Use a function from the **httr2** package to determine the content type of your response.

```
# Your code here
resp_content_type(response)
```

```
[1] "application/json"
```

```
#`resp_content_type()` returns the content type and subtype, or NA if no
 ↪  header.
#application/json for data
```

5. Use just one line of code and one function to extract the data into a matrix. Hints: 1) Use the `resp_body_json` function. 2) The first row of the matrix will be the variable names and this OK as we will fix in the next exercise.

```
population <- resp_body_json(response, simplifyVector = TRUE)
#This function takes the response object and tries to parse the content of
 ↪  its body as JSON (JavaScript Object Notation).
#USed simply vector: to convert JSON arrays (lists) into simpler R data
 ↪  types, such as vectors or data frames, whenever possible. If this were
 ↪  FALSE, all arrays would be loaded as more complex R lists.
```

6. Examine the `population` matrix you just created. Notice that 1) it is not tidy, 2) the column types are not what we want, and 3) the first row is a header. Convert `population` to a tidy dataset. Remove the state ID column and change the name of the column with state names to `state_name`. Add a column with state abbreviations called `state`. Make sure you assign the abbreviations for DC and PR correctly. Hint: Use the **janitor** package to make the first row the header.

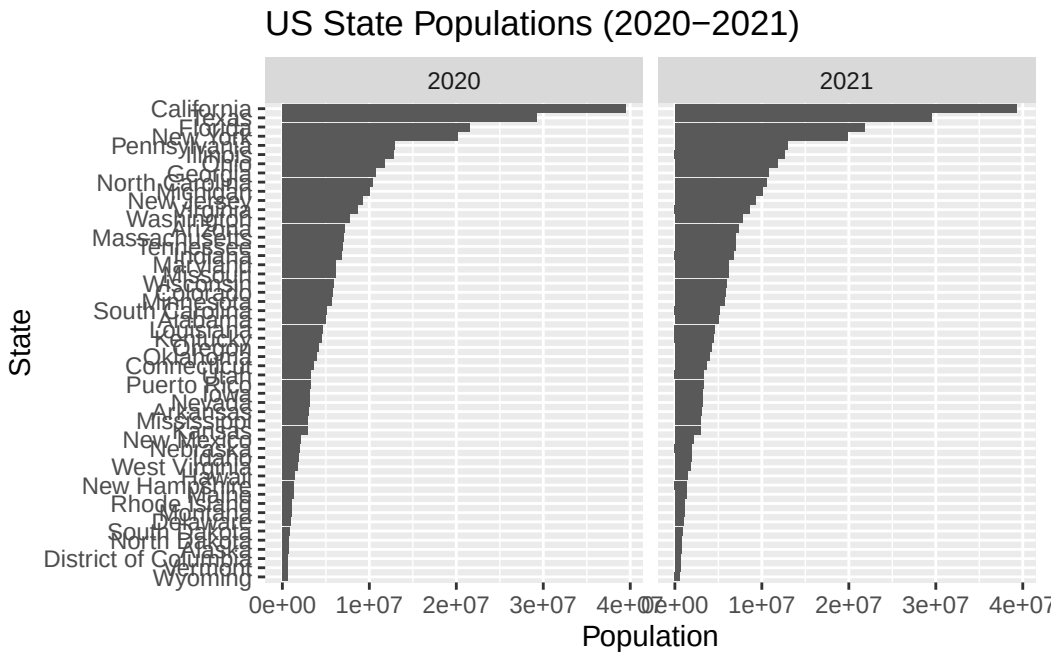
```
library(tidyverse)
library(janitor)
#population <- population |> ## Use janitor row to names function
  # convert to tibble
  # remove stat column
  # rename state column to state_name
  # use pivot_longer to tidy
  # remove POP_ from year
  # parese all relevant colunns to numeric
  # add state abbreviations using state.abb variable
  # use case_when to add abbreviations for DC and PR
```

```r
population <- population |>
  as_tibble() |>
  row_to_names(row_number = 1) |> # janitor function to col names:takes the
   ↪  first row of the data and uses its values to set the column names. The
   ↪  original first row is then removed
  select(-state) |> #removes the column named state from the dataset
  rename(state_name = NAME) |>
  pivot_longer(cols = starts_with("POP"), #selects all columns whose names
   ↪  begin with "POP"
               names_to = "year", #takes the original column names (e.g.,
                ↪  "POP_2020") and moves them into a new column called year
               values_to = "population") |> #takes the values from those
                ↪  original columns and moves them into a new column called
                ↪  population
  #each row represents a unique combination of state_name, year, and
   ↪  population
  mutate(
    year = as.numeric(str_remove(year, "POP_")), # finds the string "POP_"
     ↪  in the year column and removes it, leaving just the year number
     ↪  ("POP_2020" becomes "2020")
    population = as.numeric(population), #converts the population column,
     ↪  which is currently text, into a numeric data type. This is necessary
     ↪  because pivot_longer and other functions sometimes treat numbers as
     ↪  tex
    state = case_when( #creates a new column called state containing the
     ↪  state abbreviations. It works like an "if-else" statement
      state_name == "District of Columbia" ~ "DC", # If the state_name is
↪  "District of Columbia", the abbreviation is set to "DC"
      state_name == "Puerto Rico" ~ "PR",
      TRUE ~ state.abb[match(state_name, state.name)] #For all other cases
       ↪  (TRUE), it finds the corresponding abbreviation.state.name is a
       ↪  built-in R vector with all US state names
    # match(state_name, state.name) finds the position of the state_name
     ↪  from our dataset in the state.name vector.
    #state.abb uses that position to find the correct abbreviation from the
     ↪  built-in state.abb vector
    )
  )
```

7. As a check, make a barplot of states' 2021 and 2022 populations. Show the state names in the y-axis ordered by population size. Hint: You will need to use `reorder` and use

`facet_wrap`.

```
# population |>
  # reorder state
  # assign aesthetic mapping
  # use geom_col to plot barplot
  # flip coordinates
  # facet by year

# y-axis is a better place for long state names
population |>
  ggplot(aes(x = reorder(state_name, population), y = population)) +
  geom_col() +
  coord_flip() +
  facet_wrap(~year) +
  labs(x = "State", y = "Population",
       title = "US State Populations (2020-2021)")
```



8. The following URL:

```
url <- "https://github.com/datasciencelabs/2025/raw/refs/heads/main/data/reg
↪  ions.json"
```

points to a JSON file that lists the states in the 10 Public Health Service (PHS) defined by CDC. We want to add these regions to the `population` dataset. To facilitate this create a data frame called `regions` that has two columns `state_name`, `region`, `region_name`. One of the regions has a long name. Change it to something shorter.

```
library(jsonlite)
library(purrr)
url <- "https://github.com/datasciencelabs/2025/raw/refs/heads/main/data/reg
 ↪  ions.json"
# regions <- use jsonlit JSON parser
# regions <- convert list to data frame. You can use map_df in purrr package

#
regions <- url |>
  fromJSON(simplifyDataFrame = FALSE) |> #False to stop immediately
   ↪  converting the JSON into a data.frame. It reads the data into a list
   ↪  of lists, which allows for more control. Preserved as nested lists
  map_df(as.data.frame)|> # iterates through each element of a list, applies
   ↪  as.data.frame, and combines the results into a single df
  mutate(region_name = ifelse(region_name == "New York and New Jersey,
   ↪  Puerto Rico, Virgin Islands", "(NYC, NJ, PR, VI)", region_name ))|>
  unnest(c(states, region))|> # TA said using unest is fine
  #expands nested list columns into new rows.
  #takes the states and region columns
  # each row represents one state with its region and region_name
  rename(state_name = states) |>
  filter(state_name %in% c(state.name, "District of Columbia", "Puerto
   ↪  Rico")) |>
  select(state_name, region, region_name)

# regions
```

9. Add a region and region name columns to the `population` data frame.

```
# population <-

# merge regions data frame with population so that each state in population
 ↪  gets its corresponding region and region_name
# both data frames have a state_name column, you I use a left join
population <- population |>
  left_join(regions, by = "state_name")
```

```
# Left join over-view:
# population data frame will be updated to include new columns from the
 ↪  regions data frame.
# All rows from the original population data frame are kept.
# For each row in population, the corresponding data from regions is added
 ↪  if the state_name values match.
# If a state_name in the population data frame does not have a match in the
 ↪  regions data frame, the new columns from regions will have NA (missing)
 ↪  values.
```

10. From reading https://data.cdc.gov/ we learn the endpoint `https://data.cdc.gov/resource/pwn4-m3yp.json` provides state level data from SARS-COV2 cases. Use the **httr2** tools you have learned to download this into a data frame. Is all the data there? If not, comment on why.

```
api <- "https://data.cdc.gov/resource/pwn4-m3yp.json"
# cases_raw <-

cases_raw <- request(api) |> req_perform() |>
 ↪  resp_body_json(simplifyDataFrame = TRUE)
#parses the content of the response body from a JSON string into a
 ↪  structured R object. resp_body_json() returns parsed JSON.
# simplifyDataFrame convert the JSON data into a clean, easy-to-use R data
 ↪  frame

# nrow(cases_raw)

# Check if there is any NA in the entire data frame

# We see exactly 1,000 rows. We should be seeing over $52 \times 3$ rows per
 ↪  state.
```

We see exactly 1,000 rows. We should be seeing over 52×3 rows per state.

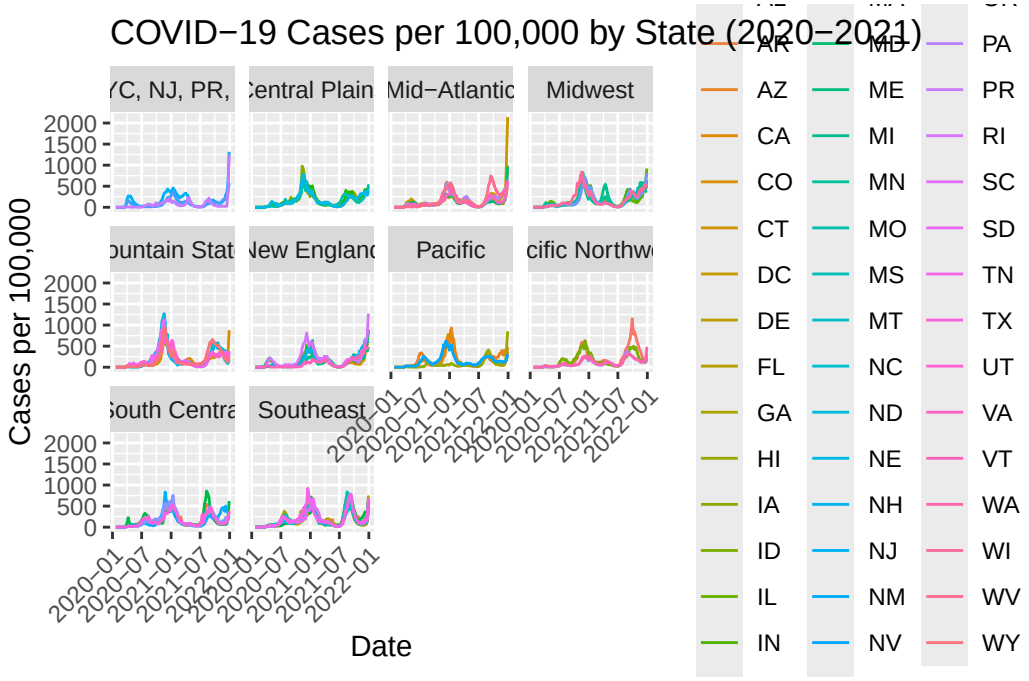
11. The reason you see exactly 1,000 rows is because CDC has a default limit. You can change this limit by adding `$limit=10000000000` to the request. Rewrite the previous request to ensure that you receive all the data. Then wrangle the resulting data frame to produce a data frame with columns `state`, `date` (should be the end date) and `cases`. Make sure the cases are numeric and the dates are in `Date` ISO-8601 format.

```
api <- "https://data.cdc.gov/resource/pwn4-m3yp.json"

cases_raw <-  request(api) |>
  req_url_query(`$limit` = 10000000000) |>  # remove CDC default limit
  req_perform() |>
  resp_body_json(simplifyDataFrame = TRUE) |>
# str(cases_raw) they were all chr, need to wrangle
  mutate(
    date = as.Date(end_date),  # convert end_date to Date format
    cases = as.numeric(new_cases)         # parse_number does the same,  #
     ↪  convert new_cases to numeric
  ) |>
  filter(!is.na(state),!is.na(date),!is.na(cases)) |>
  select(state, date, cases)               # keep only relevant columns

#head(cases_raw)
#str(cases_raw)

# date is end date in df
# cases is new cases in df
```

12. For 2020 and 2021, make a time series plot of cases per 100,000 versus time for each state. Stratify the plot by region name. Make sure to label you graph appropriately.

```
cases_raw |>
  filter(year(date) == 2020 | year(date) == 2021) |>
  left_join(population, by = "state") |>
  filter(!is.na(state),!is.na(date),!is.na(cases),!is.na(population),!is.na(
   ↪  region_name)) |> # because result was creating a seperate plot with
   ↪  NA, since there were NA values.
  ggplot(aes(x = date, y = 10^5*(cases / population) , col = state)) + #
   ↪  y-axis represents the calculated value of cases per 100,000 (cases
   ↪  divided by population, multiplied by 100,000)
  geom_line() + #creating the time series
  facet_wrap(~region_name) +
  labs(
    title = "COVID-19 Cases per 100,000 by State (2020-2021)",
    x = "Date",
    y = "Cases per 100,000"
  ) +
  theme(axis.text.x = element_text(angle = 45, hjust = 1)) #for readbility
   ↪  of x-axis
```

```
Warning in left_join(filter(cases_raw, year(date) == 2020 | year(date) == : Detected an unexp
to-many relationship between `x` and `y`.
i Row 1 of `x` matches multiple rows in `y`.
i Row 103 of `y` matches multiple rows in `x`.
i If a many-to-many relationship is expected, set `relationship =
  "many-to-many"` to silence this warning.
```



13. The dates in the `cases` dataset are stored as character strings. Use the **lubridate** package to properly parse the `date` column, then create a summary table showing the total COVID-19 cases by month and year for 2020 and 2021. The table should have columns for year, month (as month name), and total cases across all states. Order by year and month. Use the **knitr** package and `kable()` function to display the results as a formatted table.

```
library(knitr)
library(lubridate)

cases_raw |>
  filter(year(date) == 2020 | year(date) == 2021) |>
  mutate(month = month(date, label = TRUE), day = day(date), year =
    ↪ year(date)) |> #three new columns for the month, day, and year,
    ↪ extracting each from the date column using lubridate
```

```
  group_by(year, month) |>
  summarise(
    total_cases = sum(cases, na.rm = TRUE) #finding the total cases across
     ↪  all states after selecting what groupings we want
  ) |>
  arrange(year, month) |> #Order by year then month
  kable()
```

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

year	month	total_cases
2020	Jan	11
2020	Feb	68
2020	Mar	68245
2020	Apr	974032
2020	May	650943
2020	Jun	654904
2020	Jul	1989512
2020	Aug	1461283
2020	Sep	1415438
2020	Oct	1628598
2020	Nov	3932646
2020	Dec	7027128
2021	Jan	5808063
2021	Feb	2667511
2021	Mar	2068441
2021	Apr	1773591
2021	May	972915
2021	Jun	493635
2021	Jul	1137440
2021	Aug	3572562
2021	Sep	5027537
2021	Oct	2356302
2021	Nov	2322814
2021	Dec	5615644

14. The following URL provides additional COVID-19 data from the CDC in JSON format:

```
deaths_url <- "https://data.cdc.gov/resource/9bhg-hcku.json"
```

Use **httr2** to download COVID-19 death data from this endpoint. Make sure to remove the default limit to get all available data. Create a clean dataset called `deaths` with columns `state`, `date`, and `deaths` (renamed from the original column name). Ensure dates are in proper Date format and deaths are numeric.

```
# Your code here

death <- request(deaths_url) |>
  req_url_query(`$limit` = 10000000000) |>
  req_perform() |>
  resp_body_json(simplifyDataFrame = TRUE) |>
  drop_na(covid_19_deaths, end_date) |>
  summarize(state = state, death = as.numeric(covid_19_deaths), date =
   ↪  as.Date(end_date)) |>
  select(state, date, death) |>
  filter(!(state == "United States" | state ==  "New York City")) #|>
```

```
Warning: Returning more (or less) than 1 row per `summarise()` group was deprecated in
dplyr 1.1.0.
i Please use `reframe()` instead.
i When switching from `summarise()` to `reframe()`, remember that `reframe()`
  always returns an ungrouped data frame and adjust accordingly.
```

```
 # print()
```

15. Using the `deaths` dataset you created, make a bar plot showing the total COVID-19 deaths by state. Show only the top 10 states with the highest death counts. Order the bars from highest to lowest and use appropriate labels and title.

```
# Your code here
death |>
  group_by(state) |>
  summarize(total_deaths = sum(death), .groups = "drop") |>
  slice_max(total_deaths, n = 10) |>
  ggplot(aes(x = reorder(state, total_deaths, decreasing = TRUE) , y =
   ↪  total_deaths)) +
  geom_col() +
  labs(
```

```
  title = "Top 10 US States by Total COVID-19 Deaths",
  x = "State",
  y = "Total Deaths"
) +
theme(axis.text.x = element_text(angle = 45, hjust = 1)) #fixed x-axis
```