Problem Set 5

Brynn Woolley

Problem 1 - OOP Programming

Create a class to represent Wald-style normal approximation Confidence Intervals. Do this using S4.

1. For the `waldCI` class, define the following:

    1. A constructor, which takes in a confidence level (0,1) and either a mean and standard error, or a lower and upper bound. This should be a custom constructor, not `new()` or `waldCI()`.

    2. A validator.

    3. A `show` method.

    4. Accessors: `lb`, `ub`, `mean`, `sterr`, `level`.

    5. Setters: `lb`, `ub`, `mean`, `sterr`, `level`. Be sure to validate the resulting `waldCI`.

    6. A `contains` method, returning a logical of whether a value is within a CI.

    7. An `overlap` method, that takes in two `waldCI`'s, and returns a logical of whether the two confidence intervals overlap.

    8. `as.numeric` to return `c(lb, ub)`. (Hint: The second argument of `setGeneric` is not needed when an existing s3 function uses the `.Primitive` function.)

    9. `transformCI` which takes in a `function` and a `waldCI`, and returns the transformed `waldCI` object. Warn the user that only monotonic functions make sense.

```r
## define "waldCI" class
setClass(
  "waldCI",
  slots = list(
    lb = "numeric",       # lower bound
    ub = "numeric",       # upper bound
    mean = "numeric",     # mean
    sterr = "numeric",    # standard error
    level = "numeric"     # confidence level
  ),
  prototype = list(
    lb = NA_real_,
    ub = NA_real_,
    mean = NA_real_,
    sterr = NA_real_,
    level = NA_real_
  ),

  # validity rules
  validity = function(object) {

    if (!is.na(object@sterr) && object@sterr <= 0)
      return("\nStandard error must be positive.")

    if (!is.na(object@lb) && !is.na(object@ub) && object@lb > object@ub)
      return("\nlb cannot exceed ub.")

    if (length(object@level) != 1 || object@level <= 0 || object@level >= 1)
      return("\nLevel must be in (0,1).")

    if (any(!is.finite(c(object@lb, object@ub, object@mean, object@sterr)), na.rm = TRUE))
      return("\nNon-finite values are not allowed.")

    TRUE
  }
)

## constructor
waldCI_create <- function(level, mean=NA, sterr=NA, lb=NA, ub=NA) {
  ## ---------- Early check for (lb, ub) ----------
  if (!is.na(lb) && !is.na(ub)) {
    if (lb > ub)
```

```
      stop("\nlb cannot exceed ub.")
  }

  # if bounds are missing, compute them from mean & sterr
  if (is.na(lb) || is.na(ub)) {
    # Valid only if both mean and sterr are provided
    if (is.na(mean) || is.na(sterr))
      stop("\nProvide either (lb, ub) OR (mean, sterr).")

    # z multiplier for the confidence level.
    z <- qnorm(1 - (1 - level) / 2)

    lb <- mean - z * sterr
    ub <- mean + z * sterr
  } else {
    # If lb and ub are provided, compute mean and sterr from those
    mean  <- (lb + ub) / 2
    sterr <- (ub - mean) / qnorm(1 - (1 - level) / 2)
  }

  # create the object, validator runs automatically
  new("waldCI", lb = lb, ub = ub, mean = mean, sterr = sterr, level = level)
}

## define how a waldCI object prints.
setMethod("show", "waldCI", function(object) {
  cat("Wald CI:\n")
  cat(sprintf("  Level: %.3f\n", object@level))
  cat(sprintf("  Mean: %.4f\n", object@mean))
  cat(sprintf("  StdErr: %.4f\n", object@sterr))
  cat(sprintf("  CI: (%.4f, %.4f)\n", object@lb, object@ub))
})

## accessors
# generic accessors give clean external API.
setGeneric("lb",    function(x) standardGeneric("lb"))
```

```
[1] "lb"
```

```
setGeneric("ub",    function(x) standardGeneric("ub"))
```

```
[1] "ub"
```

```
setGeneric("mean",  function(x) standardGeneric("mean"))
```

```
[1] "mean"
```

```
setGeneric("sterr", function(x) standardGeneric("sterr"))
```

```
[1] "sterr"
```

```
setGeneric("level", function(x) standardGeneric("level"))
```

```
[1] "level"
```

```
# methods that simply return the corresponding slot.
setMethod("lb",    "waldCI", function(x) x@lb)
setMethod("ub",    "waldCI", function(x) x@ub)
setMethod("mean",  "waldCI", function(x) x@mean)
setMethod("sterr", "waldCI", function(x) x@sterr)
setMethod("level", "waldCI", function(x) x@level)

## setters
setGeneric("lb<-", function(x, value) standardGeneric("lb<-"))
```

```
[1] "lb<-"
```

```
setGeneric("ub<-", function(x, value) standardGeneric("ub<-"))
```

```
[1] "ub<-"
```

```
setGeneric("mean<-", function(x, value) standardGeneric("mean<-"))
```

```
[1] "mean<-"
```

```
setGeneric("sterr<-", function(x, value) standardGeneric("sterr<-"))
```

```
[1] "sterr<-"
```

```r
setGeneric("level<-", function(x, value) standardGeneric("level<-"))
```

```
[1] "level<-"
```

```r
# updating slots triggers validation each time.

setReplaceMethod("lb", "waldCI", function(x, value) {
  x@lb <- value
  validObject(x)
  x
})

setReplaceMethod("ub", "waldCI", function(x, value) {
  x@ub <- value
  validObject(x)
  x
})

setReplaceMethod("mean", "waldCI", function(x, value) {
  x@mean <- value
  validObject(x)
  x
})

setReplaceMethod("sterr", "waldCI", function(x, value) {
  x@sterr <- value
  validObject(x)
  x
})

setReplaceMethod("level", "waldCI", function(x, value) {
  x@level <- value
  validObject(x)
  x
})

## contains(): check if value lies inside CI
setGeneric("contains", function(ci, value) standardGeneric("contains"))
```

```
[1] "contains"
```

```
setMethod("contains", c("waldCI", "numeric"),
  function(ci, value) {
    # TRUE if value lies between lb and ub inclusive
    value >= ci@lb & value <= ci@ub
  }
)

## overlap(): check if two CIs intersect
setGeneric("overlap", function(ci1, ci2) standardGeneric("overlap"))
```

```
[1] "overlap"
```

```
setMethod("overlap", c("waldCI", "waldCI"),
  function(ci1, ci2) {
    # overlap occurs when ranges intersect at all
    ci1@lb <= ci2@ub && ci2@lb <= ci1@ub
  }
)

## as.numeric(): return c(lb, ub)
setMethod("as.numeric", "waldCI",
  function(x) c(x@lb, x@ub)
)

## transformCI(): apply function to CI endpoints
setGeneric("transformCI", function(ci, f) standardGeneric("transformCI"))
```

```
[1] "transformCI"
```

```
setMethod("transformCI", c("waldCI", "function"),
  function(ci, f) {

    # warn because valid results require monotonic function
    warning("Only monotonic functions produce valid transformed CIs.")

    new_lb <- f(ci@lb)
    new_ub <- f(ci@ub)

    # if function is decreasing, swap values
    if (new_lb > new_ub) {
```

```
      tmp <- new_lb
      new_lb <- new_ub
      new_ub <- tmp
    }

    # rebuild CI using transformed bounds
    waldCI_create(level = ci@level, lb = new_lb, ub = new_ub)
  }
)
```

2. Use your `waldCI` class to create three objects:

   - `ci1`: (17.2, 24.7), 95%
   - `ci2`: mean: 13, standard error: 2.5, 99%
   - `ci3`: (27.43, 39.22), 75%

```
## ci1: bounds given, level = 0.95
ci1 <- waldCI_create(level = 0.95, lb = 17.2, ub = 24.7)

## ci2: mean & sterr given, level = 0.99
ci2 <- waldCI_create(level = 0.99, mean = 13, sterr = 2.5)

## ci3: bounds given, level = 0.75
ci3 <- waldCI_create(level = 0.75, lb = 27.43, ub = 39.22)
```

Evaluate the following code:

```
ci1
```

```
Wald CI:
  Level: 0.950
  Mean: 20.9500
  StdErr: 1.9133
  CI: (17.2000, 24.7000)
```

```
ci2
```

```
Wald CI:
  Level: 0.990
```

```
  Mean: 13.0000
  StdErr: 2.5000
  CI: (6.5604, 19.4396)
```

```
    ci3
```

```
Wald CI:
  Level: 0.750
  Mean: 33.3250
  StdErr: 5.1245
  CI: (27.4300, 39.2200)
```

```
    as.numeric(ci1)
```

```
[1] 17.2 24.7
```

```
    as.numeric(ci2)
```

```
[1]  6.560427 19.439573
```

```
    as.numeric(ci3)
```

```
[1] 27.43 39.22
```

```
    lb(ci2)
```

```
[1] 6.560427
```

```
    ub(ci2)
```

```
[1] 19.43957
```

```
    mean(ci1)
```

```
[1] 20.95
```

```
    sterr(ci3)
```

```
[1] 5.12453
```

```
    level(ci2)
```

```
[1] 0.99
```

```
    lb(ci2) <- 10.5
    mean(ci3) <- 34
    level(ci3) <- .8
    contains(ci1, 17)
```

```
[1] FALSE
```

```
    contains(ci3, 44)
```

```
[1] FALSE
```

```
    overlap(ci1, ci2)
```

```
[1] TRUE
```

```
    eci1 <- transformCI(ci1, sqrt)
    eci1
```

```
Wald CI:
  Level: 0.950
  Mean: 4.5586
  StdErr: 0.2099
  CI: (4.1473, 4.9699)
```

```
    mean(transformCI(ci2, log))
```

```
[1] 2.659343
```

3. Show that your validator does not allow the creation of invalid confidence intervals:

- negative standard error
- lb > ub
- Infinite bounds
- invalid use of the setters

(Infinite confidence bounds are of course not truly invalid, but we're just going to ignore them for this case.)

```
# negative standard error
waldCI_create(level = 0.95, mean = 10, sterr = -2)
```

```
Error in validObject(.Object): invalid class "waldCI" object:
Standard error must be positive.
```

```
# lb > ub
waldCI_create(level = 0.95, lb = 10, ub = 5)
```

```
Error in waldCI_create(level = 0.95, lb = 10, ub = 5):
lb cannot exceed ub.
```

```
# infinite bounds
waldCI_create(level = 0.95, lb = -Inf, ub = Inf)
```

```
Error in validObject(.Object): invalid class "waldCI" object:
Non-finite values are not allowed.
```

```
# invalid use of the setters
ci <- waldCI_create(level = 0.95, mean = 10, sterr = 1)
lb(ci) <- 20
```

```
Error in validObject(x): invalid class "waldCI" object:
lb cannot exceed ub.
```

Note that there are a lot of choices to be made here. What are you going to store in the class? How are you going to store them (what object types)? How are you going to enforce the function in `transform` being monotonic?

There is no right answer to those questions. Make the best decision you can, and don't be afraid to change it if your decision causes unforeseen difficulties.

You may not use any existing R functions or packages that would trivialize this assignment. (E.g. if you found an existing package that does this, or found a function that checks for overlap between two CIs, that is not able to be used.)

What are you going to store in the class? I store all five components explicitly: the lower bound, upper bound, mean, standard error, and confidence level. Storing everything avoids recomputation and makes accessor and setter methods straightforward.

How are you going to store them (what object types)? Each component is stored as a `numeric` slot. All are length-1 numeric values.

How are you going to enforce the function in `transform` being monotonic? I cannot guarantee monotonicity automatically, so I warn the user that the function must be monotonic. After applying the function to both endpoints, I check whether the transformed values reverse order; if they do, I swap them so the resulting CI remains valid.

Problem 3 - plotly

Repeat [problem set 4, question 3] using plotly.

There is no expectation that you produce the exact same plots as last time. You may of course use your plots as last time, or the ones from the problem set 4 solutions, as inspiration for these plots.

These will be graded similar to last time:

1. Is the type of graph & choice of variables appropriate to answer the question?

2. Is the graph clear and easy to interpret?

3. Is the graph publication ready?

Use the NYTimes Covid data (https://raw.githubusercontent.com/nytimes/covid-19-data/refs/heads/master/rolling-averages/us-states.csv).

This lists daily Covid new cases. For each of the following, produce a publication-ready plot which addresses the question. Use your plot to support an argument for your question.

a. *How many major and minor spikes in cases were there?*

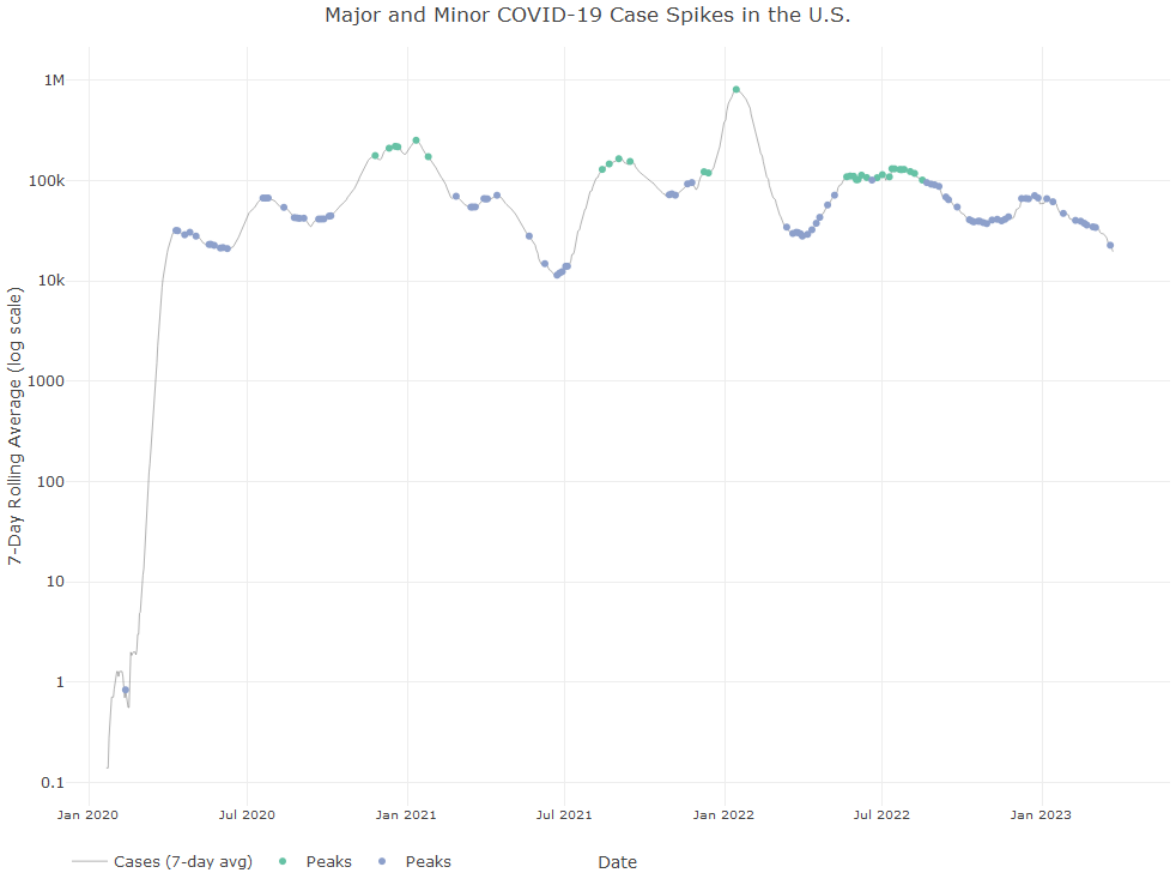
```r
# load data
covid_states <- read_csv(
  "https://raw.githubusercontent.com/nytimes/covid-19-data/refs/heads/master/rolling-avera
)

# aggregate by date
covid_us <- covid_states %>%
  group_by(date) %>%
  summarise(cases_avg = sum(cases_avg, na.rm = TRUE))

# look for peaks
covid_us <- covid_us %>%
  arrange(date) %>%
  mutate(
    prev_avg = lag(cases_avg),
    next_avg = lead(cases_avg),
    is_peak = cases_avg > prev_avg & cases_avg > next_avg
  )

# extract peaks and classify by relative size
peaks <- covid_us %>%
  filter(is_peak) %>%
  mutate(
    peak_type = case_when(
      cases_avg >= quantile(cases_avg, 0.75, na.rm = TRUE) ~ "Major",
      TRUE ~ "Minor"
    )
  ) %>%
  arrange(desc(cases_avg))

peaks %>%
  count(peak_type)
```

```
# A tibble: 2 x 2
  peak_type     n
  <chr>     <int>
1 Major        31
2 Minor        92
```

```r
top_peaks <- peaks %>%
  filter(peak_type == "Major") %>%
```

```r
  slice_max(cases_avg, n = 3)

# plot
p1 <- plot_ly() %>%
  add_lines(
    data = covid_us,
    x = ~date,
    y = ~cases_avg,
    name = "Cases (7-day avg)",
    line = list(width = 0.6, color = "gray")
  ) %>%
  add_markers(
    data = peaks,
    x = ~date,
    y = ~cases_avg,
    color = ~peak_type,
    colors = c("Major" = "firebrick", "Minor" = "steelblue"),
    marker = list(size = 6),
    name = "Peaks"
  ) %>%
  layout(
    title = list(
      text = "Major and Minor COVID-19 Case Spikes in the U.S.",
      x = 0.5,
      y = 0.99
    ),
    xaxis = list(
    title = "Date",
    tickformat = "%b %Y",
    tickfont = list(size = 10),
    title_standoff = 20
    ),
    yaxis = list(
      title = "7-Day Rolling Average (log scale)",
      type = "log"
    ),
    legend = list(
      orientation = "h",
      yanchor = "top",
      y = -0.05   # much closer to the plot
    ),
    margin = list(t = 40)
```

```
  )
# save static version for PDF
plotly::export(p1, file = "p1.png")
```



The figure shows several distinct spikes in U.S. COVID-19 case data between 2020 and 2023. We see major peaks occur in December 2020, January 2021, and January 2022. Smaller waves appeared throughout mid-2021 and late 2022. The log scale highlights both large and moderate increases while preserving proportional differences.

b. *For the states with the highest and lowest overall rates per population, what differences do you see in their trajectories over time?*

```
# identify states with highest & lowest rates
state_rates <- covid_states %>%
  group_by(state) %>%
  summarise(
```

```
    mean_cases_per_100k = mean(cases_avg_per_100k, na.rm = TRUE)
  ) %>%
  arrange(desc(mean_cases_per_100k))

# extract top and bottom states
top_state <- slice_max(state_rates, mean_cases_per_100k, n = 1)
bottom_state <- slice_min(state_rates, mean_cases_per_100k, n = 1)


top_state$state
```

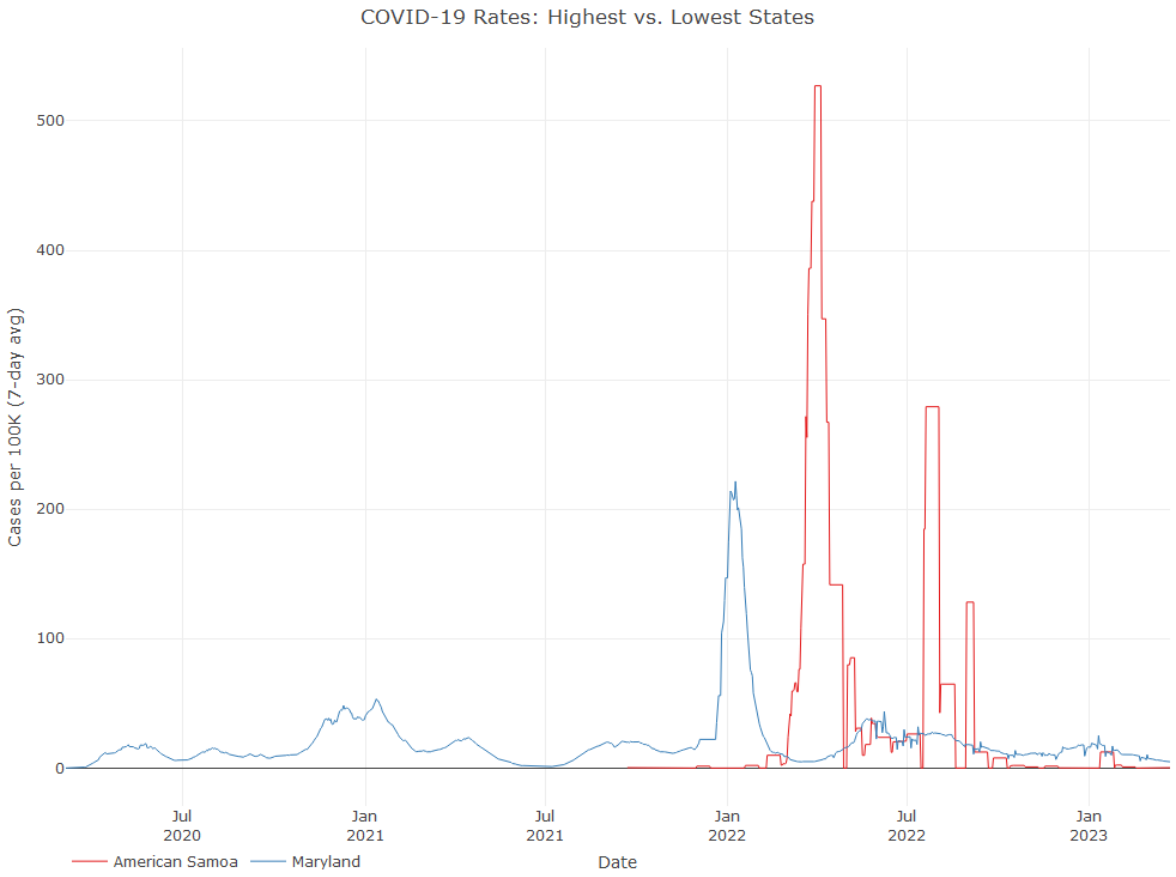
```
[1] "American Samoa"
```

```
bottom_state$state
```

```
[1] "Maryland"
```

```
# plot
p2 <- plot_ly(
  data = covid_states %>%
    filter(state %in% c(top_state$state, bottom_state$state)),
  x = ~date,
  y = ~cases_avg_per_100k,
  color = ~state,
  colors = "Set1",
  type = "scatter",
  mode = "lines",
  line = list(width = 1)
) %>%
  layout(
    title = list(
      text = "COVID-19 Rates: Highest vs. Lowest States",
      x = 0.5
    ),
    xaxis = list(
      title = "Date",
      tickformat = "%b\n%Y"
    ),
    yaxis = list(
      title = "Cases per 100K (7-day avg)"
```

```
    ),
    legend = list(
      orientation = "h",
      yanchor = "top",
      y = -0.05   # much closer to the plot
    ),
    margin = list(t = 40)
  )
# save static version for PDF
plotly::export(p2, file = "p2.png")
```



The state with the highest rate (American Samoa) shows sharp, isolated peaks concentrated in early 2022, reflecting intense but brief outbreaks.

The state with the lowest rate (Maryland), by contrast, maintained a much flatter trajectory throughout the pandemic, with smaller fluctuations and lower sustained case rates.

This highlights how outbreak intensity and timing varied dramatically even after population

adjustment.

c. Identify, to the best of your ability without a formal test, the first five states to experience Covid in a substantial way.

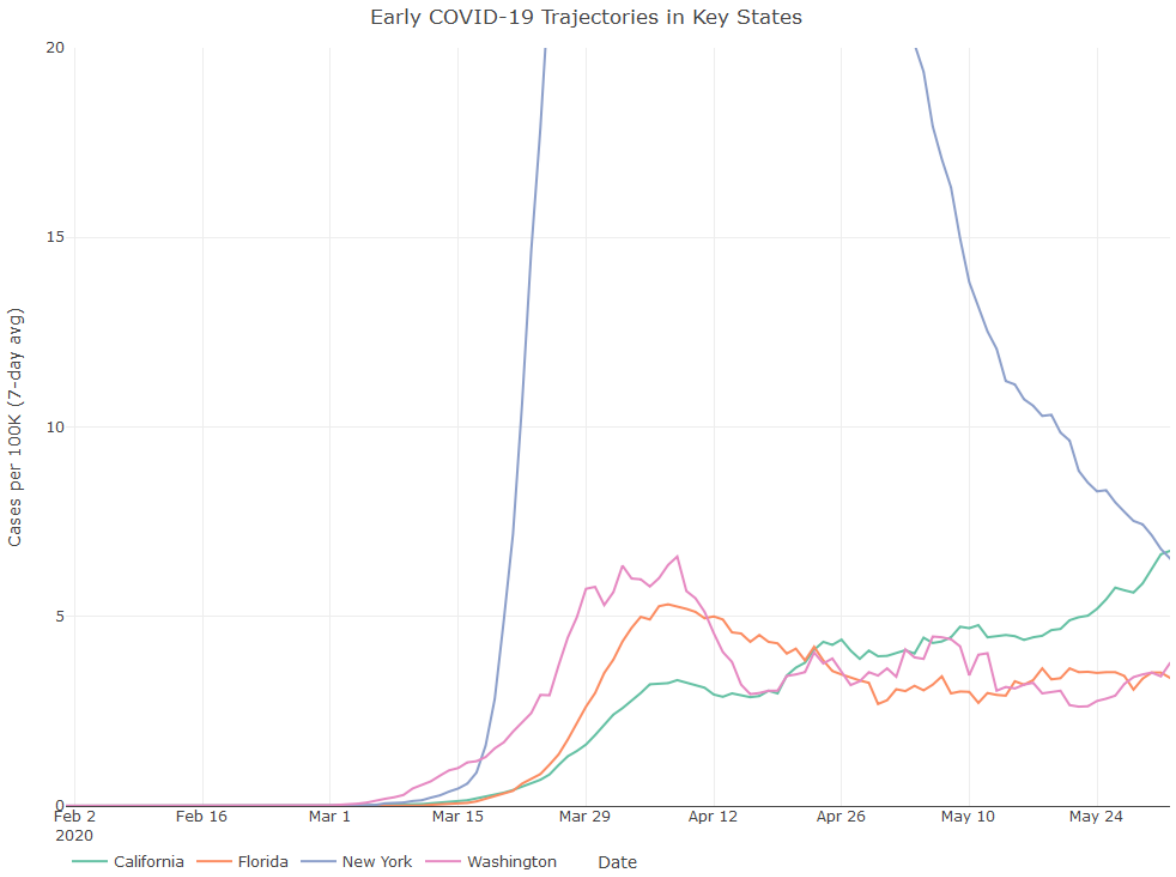
```
## explore data to inform threshold (c) and consecutive days (k)

# examine distribution of daily rates
summary_stats <- covid_states %>%
  summarise(
    min = min(cases_avg_per_100k, na.rm = TRUE),
    q25 = quantile(cases_avg_per_100k, 0.25, na.rm = TRUE),
    median = median(cases_avg_per_100k, na.rm = TRUE),
    mean = mean(cases_avg_per_100k, na.rm = TRUE),
    q75 = quantile(cases_avg_per_100k, 0.75, na.rm = TRUE),
    p90 = quantile(cases_avg_per_100k, 0.9, na.rm = TRUE),
    p95 = quantile(cases_avg_per_100k, 0.95, na.rm = TRUE),
    max = max(cases_avg_per_100k, na.rm = TRUE)
  )
summary_stats
```

```
# A tibble: 1 x 8
    min   q25 median  mean   q75   p90   p95   max
  <dbl> <dbl>  <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
1     0  7.87   15.9  28.0  32.0  60.4  87.8  527.
```

```
# assess persistence of small vs large values
covid_states %>%
  group_by(state) %>%
  mutate(over_1 = cases_avg_per_100k > 1,
         run_id = with(rle(over_1), rep(seq_along(lengths), lengths)),
         run_len = ave(over_1, run_id, FUN = length)) %>%
  ungroup() %>%
  filter(over_1) %>%
  summarise(mean_run = mean(run_len), median_run = median(run_len), max_run = max(run_len)
```

```
# A tibble: 1 x 3
  mean_run median_run max_run
     <dbl>      <dbl>   <int>
1     980.       1092    1103
```

```r
# visualize early trajectories
# exploratory plot of early states
p3 <- plot_ly(
  data = covid_states %>%
    filter(state %in% c("Washington", "California", "New York", "Florida")),
  x = ~date,
  y = ~cases_avg_per_100k,
  color = ~state,
  type = "scatter",
  mode = "lines"
) %>%
  layout(
    title = "Early COVID-19 Trajectories in Key States",
    xaxis = list(
      # use Date or character, NOT as.numeric(...)
      range = c(as.Date("2020-02-01"), as.Date("2020-06-01")),
      # or: range = c("2020-02-01", "2020-06-01"),
      title = "Date"
    ),
    yaxis = list(
      range = c(0, 20),
      title = "Cases per 100K (7-day avg)"
    ),
    legend = list(
      orientation = "h",
      yanchor = "top",
      y = -0.05   # much closer to the plot
    ),
    margin = list(t = 40)
  )

# save static version for PDF
plotly::export(p3, file = "p3.png")
```




```
# quantify baseline noise (pre-March 2020)
baseline_sd <- covid_states %>%
  filter(date < as.Date("2020-03-01")) %>%
  summarise(sd_base = sd(cases_avg_per_100k, na.rm = TRUE))
baseline_sd
```

```
# A tibble: 1 x 1
  sd_base
    <dbl>
1  0.0131
```

The exploratory plot shows that early-hit states such as Washington, California, and New York begin sustained increases above about 1 case per 100K in late February and early March. Before this point, all states exhibit near-zero, non-persistent noise. Visually, the transition above 1 per 100K represents the first clear signal of community spread. These increases

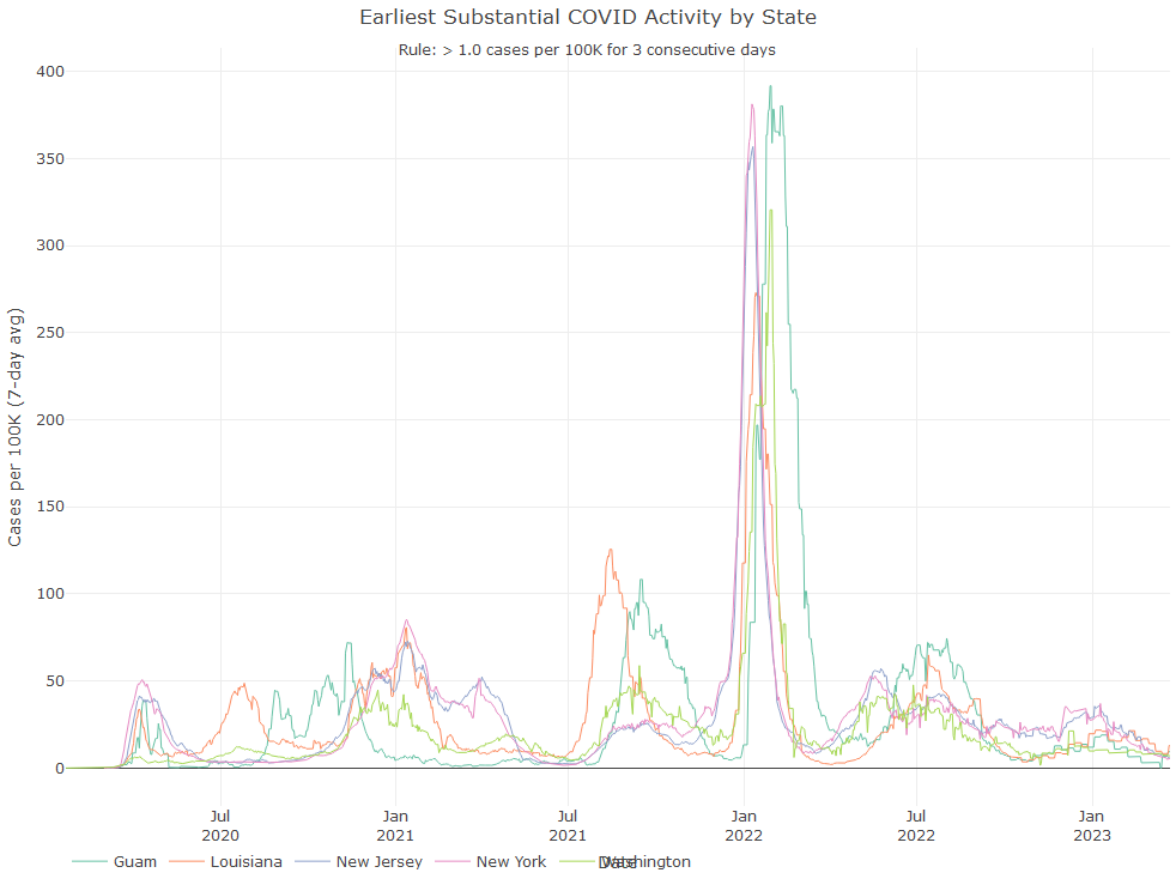
persist for several days, suggesting that a requirement of three consecutive days filters out noise effectively.

```
# threshold and persistence
# apply rule to identify first states
c <- 1
k <- 3
#' Identify first date of sustained COVID activity
#' @param df State-level data frame
#' @param c Threshold for cases_avg_per_100k
#' @param k Minimum consecutive days
#' @return Tibble with first_date

first_sustained <- function(df, c, k) {
  df <- dplyr::arrange(df, date)
  over <- df$cases_avg_per_100k > c
  r <- rle(over)
  run_id <- rep(seq_along(r$lengths), r$lengths)
  run_len <- ave(over, run_id, FUN = length)
  hit <- over & (run_len >= k)
  if (!any(hit, na.rm = TRUE)) return(dplyr::tibble(first_date = as.Date(NA)))
  dplyr::tibble(first_date = min(df$date[hit], na.rm = TRUE))
}

first_dates <- covid_states %>%
  group_by(state) %>%
  group_modify(~ first_sustained(.x, c = c, k = k)) %>%
  ungroup() %>%
  arrange(first_date)

first_five <- slice_head(first_dates, n = 5)

first_five %>%
  knitr::kable(
    col.names = c("State", "First Substantial Case Date"),
    caption = sprintf(
      "First five states by rule: cases_avg_per_100k > %s for %s consecutive days.",
      c, k
    )
  )
```

Table 1: First five states by rule: cases_avg_per_100k > 1 for 3 consecutive days.

State	First Substantial Case Date
Washington	2020-03-16
New York	2020-03-18
Guam	2020-03-19
Louisiana	2020-03-19
New Jersey	2020-03-19

```
# plot
p4 <- plot_ly(
  data = covid_states %>% filter(state %in% first_five$state),
  x = ~date,
  y = ~cases_avg_per_100k,
  color = ~state,
  type = "scatter",
  mode = "lines",
  line = list(width = 1)
) %>%
  layout(
    title = sprintf(
      "Earliest Substantial COVID Activity by State<br><sub>Rule: > %.1f cases per 100K fo
      c, k
    ),
    xaxis = list(title = "Date", tickformat = "%b\n%Y"),
    yaxis = list(title = "Cases per 100K (7-day avg)"),
    legend = list(
      orientation = "h",
      yanchor = "top",
      y = -0.05   # much closer to the plot
    ),
    margin = list(t = 40)
  )
# save static version for PDF
plotly::export(p4, file = "p4.png")
```



Applying the rule of >1 case per 100K for at least 3 consecutive days identifies the five earliest states to experience substantial COVID activity. The final plot shows that these states all exhibit clear, early upward trajectories consistent with the threshold. This supports the rule derived from the exploratory visualization and confirms that these states were the earliest to experience meaningful spread.

GitHub Link

- Repo: https://github.com/brynnwoolley/STATS-506#