HW5

Stats 506 Homework #5

Margaret Miles

Problem #1: **OOP Programming**

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

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

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

   b. A validator.

   c. A show method.

   d. Accessors: `lb`, `ub`, `mean`, `sterr`.

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

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

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

   h. 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.)

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

All `level` arguments should have a reasonable default value. `digits` as well.

```
##' @title create waldCI class using S4
setClass("waldCI",
         slot = c(
           mean = "numeric",
           sterr = "numeric",
           lb = "numeric",
           ub = "numeric",
           level = "numeric"
         ))

##' @title make a constructor
##' @param mean Numeric
```

```r
##' @param sterr Numeric
##' @param lb numeric
##' @param ub numeric
##' @param level numeric
##' @return new `waldCI` object
makeWaldCI <- function(mean = NULL, sterr = NULL,
                       lb = NULL, ub = NULL, level = 0.95){
  # for just mean and sterr
  if (!is.null(mean) && !is.null(sterr)) {
    # make Wald CI
    alpha <- 1 - level
    z <- qnorm(1 - alpha / 2)
    lb <- mean - z * sterr
    ub <- mean + z * sterr
  } else if (!is.null(lb) && !is.null(ub)) {
    # for CIs and level
    # made Wald CI
    mean <- (lb + ub) / 2
    alpha <- 1 - level
    z <- qnorm(1 - alpha / 2)
    sterr <- (ub - mean) / z
  } else {
    stop("Provide either (mean, sterr) or (lb, ub) and level.")
  }

  # Return the S4 object
  new("waldCI",
      mean = mean,
      sterr = sterr,
      lb = lb,
      ub = ub,
      level = level)
}

# validator!!!
##' @title create validator for WaldCI object
##' @param waldCI class object
setValidity("waldCI",  function(object) {
  # non-numeric
  if((!is.numeric(object@mean) || is.null(object@mean)) ||
     (!is.numeric(object@sterr) || is.null(object@sterr)) ||
     (!is.numeric(object@lb) || is.null(object@lb)) ||
     (!is.numeric(object@ub) || is.null(object@ub))){
    stop("A value is non-numeric, please enter only numeric values.")
  }

  # all null
  if(is.null(object@mean) && is.null(object@sterr) && is.null(object@lb) && is.null(objec
    stop("Missing all values, please enter values for CIs or Mean and sterr")
  }
```

```
  # if mean and sterr are null, but so is one of the CIs or level
  if (is.null(object@mean) && is.null(object@sterr) &&
      (is.null(object@lb) || is.null(object@lb) || is.null(object@level))) {
    stop("Missing either lb, ub, or level")
  }

  # if CIs are null but so is mean or sterr
  if (is.null(object@lb) && is.null(object@lb) && (is.null(object@mean) || is.null(object
    stop("Missing either mean or sterr.")
  }

  # if level is not between 0 and 1
  if(object@level > 1 || object@level < 0){
    stop("Level must be inbetween 0 and 1")
  }

  # if error is negative
  if(object@sterr < 0 && !is.null(object@sterr)){
    stop("Standard Error must be non-negative.")
  }

  # lower bound and upper bound error
  if (object@lb >= object@ub)
      return("Lower bound must be less than upper bound.")

  # bounds must be finite
  if (!is.finite(object@lb) || !is.finite(object@ub))
      return("Bounds must be finite numbers.")

  # mean is not in between bounds
  if (!is.null(object@mean)){
    if(object@mean > object@ub || object@mean < object@lb){
      return("Mean cannot exceed the current bounds")
    }
  }

  return(TRUE)

})
```

```
Class "waldCI" [in ".GlobalEnv"]

Slots:

Name:     mean   sterr      lb      ub   level
Class: numeric numeric numeric numeric numeric
```

```
# A `show` method, which takes as arguments `level` and `digits`
##' @title a show method for a `waldCI` object
##' @param object a `waldCI` object
```

```
setMethod("show", "waldCI",
          function(object){
            digits = 3
            cat("Wald-style Confidence Interval\n")
            cat("Level: ")
            cat(object@level)
            cat("\nMean: ", round(object@mean, digits))
            cat("\nStandard Error: ", round(object@sterr,digits))
            cat("\nLower Bound: ", round(object@lb, digits))
            cat("\nUpper Bound: ", round(object@ub, digits))
            cat("\n\n")
            return(invisible(object))
          })

# Accessors
# `lb`, `ub`, `mean`, `sterr`,`level`.

setGeneric("lb", function(object){
  standardGeneric("lb")
})
```

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

```
##' @title Return the lower bound of a `waldCI`
##' @param object A `waldCI` object
##' @return It's lower bound
setMethod("lb", "waldCI", function(object){
  return(slot(object, "lb"))
})

setGeneric("ub", function(object){
  standardGeneric("ub")
})
```

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

```
##' @title Return the upper bound of a `waldCI`
##' @param object A `waldCI` object
##' @return It's upper bound
setMethod("ub", "waldCI", function(object){
  return(slot(object, "ub"))
})

setGeneric("mean", function(object){
  standardGeneric("mean")
})
```

```
Creating a new generic function for 'mean' in the global environment
```

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

```
##' @title Return the mean of a `waldCI`
##' @param object A `waldCI` object
##' @return It's mean
setMethod("mean", "waldCI", function(object){
  return(slot(object, "mean"))
})

setGeneric("sterr", function(object){
  standardGeneric("sterr")
})
```

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

```
##' @title Return the standard error of a `waldCI` object
##' @param object A `waldCI` object
##' @return It's standard error
setMethod("sterr", "waldCI", function(object){
  return(slot(object, "sterr"))
})

setGeneric("level", function(object){
  standardGeneric("level")
})
```

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

```
##' @title Return the lower bound of a `waldCI`
##' @param object A `waldCI` object
##' @return It's lower bound
setMethod("level", "waldCI", function(object){
  return(slot(object, "level"))
})

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

setGeneric("lb<-",
           function(object, value) {
             standardGeneric("lb<-")
           })
```

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

```
##' @title Set the lower bound of a `waldCI`
##' @param object A `waldCI` object
##' @param value New lower bound
##' @return The updated waldCI
```

```
setMethod("lb<-", "waldCI",
  function(object, value) {
    object@lb <- value
    validObject(object) # Re-run validity check
    updated <- makeWaldCI(ub = object@ub, lb = value, level = object@level)
    return(updated)
  }
)

setGeneric("ub<-",
           function(object, value) {
             standardGeneric("ub<-")
           })
```

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

```
##' @title Set the upper bound of a `waldCI`
##' @param object A `waldCI` object
##' @param value New upper bound
##' @return The updated object
setMethod("ub<-", "waldCI",
  function(object, value) {
    object@ub <- value
    validObject(object) # Re-run validity check
    updated <- makeWaldCI(ub = value, lb = object@lb, level = object@level)
    return(updated)
  }
)

setGeneric("mean<-",
           function(object, value) {
             standardGeneric("mean<-")
           })
```

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

```
##' @title Set the mean of a `waldCI`
##' @param object A `waldCI` object
##' @param value mean
##' @return The updated object
setMethod("mean<-", "waldCI",
  function(object, value) {
    object@mean <- value
    validObject(object) # Re-run validity check
    updated <- makeWaldCI(mean = value, sterr = object@sterr, level = object@level)
    return(updated)
  }
)
```

```
setGeneric("sterr<-",
           function(object, value) {
             standardGeneric("sterr<-")
           })
```

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

```
##' @title Set the standard error of a `waldCI`
##' @param object A `waldCI` object
##' @param value New standard error
##' @return The updated object
setMethod("sterr<-", "waldCI",
  function(object, value) {
    object@sterr <- value
    validObject(object) # Re-run validity check
    updated <- makeWaldCI(mean = object@mean, sterr = value, level = object@level)
    return(updated)
  }
)

setGeneric("level<-",
           function(object, value) {
             standardGeneric("level<-")
           })
```

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

```
##' @title Set the confidence level of a `waldCI`
##' @param object A `waldCI` object
##' @param value New level
##' @return The updated object
setMethod("level<-", "waldCI",
  function(object, value) {
    object@level <- value
    validObject(object) # Re-run validity check
    updated <- makeWaldCI(mean = object@mean, sterr = object@sterr, ub = object@ub,
                          lb = object@lb, level = object@level)
    return(updated)
  }
)

# A contains method, returning a logical of whether a value is within a CI.
setGeneric("contains", function(object, value) {
  standardGeneric("contains")
  })
```

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

```
##' @title a contains method
##' @param object A `waldCI` object
##' @param value a value to test whether it is in a CI
##' @return logical
setMethod("contains", "waldCI", function(object, value) {
  value >= object@lb && value <= object@ub
})

# An overlap method, that takes in two waldCI's, and returns a logical of whether the two
setGeneric("overlap", function(object1, object2) {
  standardGeneric("overlap")
  })
```

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

```
##' @title a overlap method
##' @param object1 A `waldCI` object
##' @param object2 another `waldCI` object
##' @return logical
setMethod("overlap", "waldCI", function(object1, object2) {
  return(!(object1@ub < object2@lb || object2@ub < object1@lb))
})



# as.numeric to return c(lb, ub). (Hint: The second argument of setGeneric is not needed

# have to use x
# asked chatGPT why

##' @title an as.numeric to return the lower and upper bound of the object
##' @param x A `waldCI` object
##' @return c(lb, ub) of a `waldCI` object
setMethod("as.numeric", "waldCI", function(x){
  return(c(x@lb, x@ub))
  })

# transformCI which takes in a function and a waldCI, and returns the transformed waldCI
setGeneric("transformCI", function(object, func) {
  standardGeneric("transformCI")
  })
```

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

```
##' @title a transformCI which returns a transformed WaldCI object
##' @param object A `waldCI` object
##' @param func a function
##' @return a transformed `waldCI` object
setMethod("transformCI", c("waldCI", "function"), function(object, func){
  warning("Only monotonic functions are allowed and will make sense.")
```

```
  translb <- func(object@lb)
  transub <- func(object@ub)
  transmean <- func(object@mean)
  # need to calculate SE again
  alpha <- 1 - object@level
  z <- qnorm(1 - alpha / 2)
  transsterr <- (transub - transmean) / z

  if(!is.numeric(translb) || !is.numeric(transub) || !is.numeric(transsterr) || !is.numer
    stop("Values are now non-numeric. Doesn't work.")
  }

  if(is.null(transsterr) || is.null(translb) || is.null(transub) || is.null(transmean)){
    stop("Function produced NULL results.")
  }

  if(is.na(transsterr) || is.na(translb) || is.na(transub) || is.na(transmean)){
    stop("Produced NAs.")
  }

  if(translb == object@lb && transub == object@ub || transmean == object@mean){
    stop("Function didn't make sense. Did not change values.")
  }
  new1 <- makeWaldCI( mean = transmean, sterr = transsterr, lb = translb,
      ub = transub, level = object@level)
  validObject(new1)
  return(new1)
  })
```

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

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

```
# construct from CIs and level
ci1 <- makeWaldCI(lb = 17.2, ub = 24.7, level = 0.95)

# Construct from mean and SE
ci2 <- makeWaldCI(mean = 13, sterr = 2.5, level = 0.99)

# for new CI and level
ci3 <- makeWaldCI(lb = 27.43, ub = 39.22, level = 0.75)

ci1
```

```
Wald-style Confidence Interval
Level: 0.95
Mean:  20.95
```

```
Standard Error:  1.913
Lower Bound:  17.2
Upper Bound:  24.7
```

```
ci2
```

```
Wald-style Confidence Interval
Level: 0.99
Mean:  13
Standard Error:  2.5
Lower Bound:  6.56
Upper Bound:  19.44
```

```
ci3
```

```
Wald-style Confidence Interval
Level: 0.75
Mean:  33.325
Standard Error:  5.125
Lower Bound:  27.43
Upper Bound:  39.22
```

```
print("Prints out CIs as expected")
```

```
[1] "Prints out CIs as expected"
```

```
cat("\n")
```

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

```
[1] 17.2 24.7
```

```
cat("\n")
```

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

```
[1]  6.560427 19.439573
```

```
cat("\n")
```

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

```
[1] 27.43 39.22
```

```
print("Prints out numeric as expected.")
```

```
[1] "Prints out numeric as expected."
```

```
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
```

```
print("Prints accessors as expected.")
```

```
[1] "Prints accessors as expected."
```

```
lb(ci2) <- 10.5
ci2
```

```
Wald-style Confidence Interval
Level: 0.99
Mean:  14.97
Standard Error:  1.735
Lower Bound:  10.5
Upper Bound:  19.44
```

```
print("Changes the lb, recalculates mean and standard error")
```

```
[1] "Changes the lb, recalculates mean and standard error"
```

```
mean(ci3) <- 34
ci3
```

```
Wald-style Confidence Interval
Level: 0.75
Mean:  34
```

```
Standard Error:  5.125
Lower Bound:  28.105
Upper Bound:  39.895
```

```
print("Changes the mean, recalculates lb and ub")
```

```
[1] "Changes the mean, recalculates lb and ub"
```

```
level(ci3) <- .8
ci3
```

```
Wald-style Confidence Interval
Level: 0.8
Mean:  34
Standard Error:  5.125
Lower Bound:  27.433
Upper Bound:  40.567
```

```
print("Changes the level, recalculates the lower and upper bound only")
```

```
[1] "Changes the level, recalculates the lower and upper bound only"
```

```
contains(ci1, 17)
```

```
[1] FALSE
```

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

```
[1] FALSE
```

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

```
[1] TRUE
```

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

```
Warning in transformCI(ci1, sqrt): Only monotonic functions are allowed and
will make sense.
```

```
eci1
```

```
Wald-style Confidence Interval
Level: 0.95
Mean:  4.577
Standard Error:  0.2
```

```
Lower Bound:  4.184
Upper Bound:  4.97
```

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

```
Warning in transformCI(ci2, log): Only monotonic functions are allowed and will
make sense.

[1] 2.706034
```

All of these work as intended and shown above.

c. 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

```
# found try() from ChatGPT
# negative standard error
try(ci4 <- makeWaldCI(mean = 10, sterr = -1, level = 0.95))
```

```
Error in validityMethod(object) : Standard Error must be non-negative.
```

```
try(ci4 <- makeWaldCI(mean = 10, sterr = -10))
```

```
Error in validityMethod(object) : Standard Error must be non-negative.
```

```
# lb > ub
try(ci4 <- makeWaldCI(lb = 47.43, ub = 39.22, level = 0.75))
```

```
Error in validityMethod(object) : Standard Error must be non-negative.
```

```
try(ci4 <- makeWaldCI(lb = 0, ub = 0, level = 0.75))
```

```
Error in validObject(.Object) :
  invalid class "waldCI" object: Lower bound must be less than upper bound.
```

```
# Infinite bounds
try(ci4 <- makeWaldCI(lowerCI = -Inf, upperCI = 10, level = 0.95))
```

```
Error in makeWaldCI(lowerCI = -Inf, upperCI = 10, level = 0.95) :
  unused arguments (lowerCI = -Inf, upperCI = 10)
```

```
try(ci4 <- makeWaldCI(lowerCI = 0, upperCI = Inf, level = 0.95))
```

```
Error in makeWaldCI(lowerCI = 0, upperCI = Inf, level = 0.95) :
  unused arguments (lowerCI = 0, upperCI = Inf)
```

```
# invalid use of the setters

# Attempt to set a negative SE
try(sterr(ci1) <- -5)
```

```
Error in validityMethod(object) : Standard Error must be non-negative.
```

```
# Attempt to set lb > ub
try(lb(ci1) <- 20)
# Attempt to set an infinite upper bound
try(ub(ci1) <- Inf)
```

```
Error in validObject(object) :
  invalid class "waldCI" object: Bounds must be finite numbers.
```

```
# attempt a mean outside of the bounds
try(mean(ci1) <- 35)
```

```
Error in validObject(object) :
  invalid class "waldCI" object: Mean cannot exceed the current bounds
```

```
# trying transform
try(ci4 <- transformCI(ci1, acos))
```

```
Warning in transformCI(ci1, acos): Only monotonic functions are allowed and
will make sense.

Warning in func(object@lb): NaNs produced

Warning in func(object@ub): NaNs produced

Warning in func(object@mean): NaNs produced

Error in transformCI(ci1, acos) : Produced NAs.
```

```
try(ci4 <- transformCI(ci1, tanpi))
```

```
Warning in transformCI(ci1, tanpi): Only monotonic functions are allowed and
will make sense.

Error in validityMethod(object) : Standard Error must be non-negative.
```

```
try(ci4 <- transformCI(ci1, stderr))
```

```
Warning in transformCI(ci1, stderr): Only monotonic functions are allowed and
will make sense.

Error in func(object@lb) : unused argument (object@lb)
```

```
try(ci4 <- transformCI(ci1, sum))
```

```
Warning in transformCI(ci1, sum): Only monotonic functions are allowed and will
make sense.

Error in transformCI(ci1, sum) :
  Function didn't make sense. Did not change values.
```

```
try(ci4 <- transformCI(ci1, ncol))
```

```
Warning in transformCI(ci1, ncol): Only monotonic functions are allowed and
will make sense.

Error in transformCI(ci1, ncol) :
  Values are now non-numeric. Doesn't work.
```

```
try(ci4 <- transformCI(ci1, max))
```

```
Warning in transformCI(ci1, max): Only monotonic functions are allowed and will
make sense.

Error in transformCI(ci1, max) :
  Function didn't make sense. Did not change values.
```

```
try(ci4 <- transformCI(ci1, log2))
```

```
Warning in transformCI(ci1, log2): Only monotonic functions are allowed and
will make sense.
```

```
try(ci4 <- transformCI(ci1, is.object))
```

```
Warning in transformCI(ci1, is.object): Only monotonic functions are allowed
and will make sense.

Error in transformCI(ci1, is.object) :
  Values are now non-numeric. Doesn't work.
```

```
try(ci4 <- transformCI(ci1, cos))
```

```
Warning in transformCI(ci1, cos): Only monotonic functions are allowed and will
make sense.
```

```
try(ci4 <- transformCI(ci1, sin))
```

```
Warning in transformCI(ci1, sin): Only monotonic functions are allowed and will
make sense.

Error in validityMethod(object) : Standard Error must be non-negative.
```

```
try(ci4 <- transformCI(ci1, abs))
```

```
Warning in transformCI(ci1, abs): Only monotonic functions are allowed and will
make sense.

Error in transformCI(ci1, abs) :
  Function didn't make sense. Did not change values.
```

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.)

Hint: It may be useful to define other functions that I don't explicitly ask for.

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:

Is the type of graph & choice of variables appropriate to answer the question? Is the graph clear and easy to interpret? Is the graph publication ready?

Note: This is, intentionally, a very open-ended question. There is no "right" answer. The goal is for you to explore your plotting options, and settle on something reasonable. You can use base R, ggplot, or something else. You'll likely have to look online for resources on plotting beyond what we covered in class.

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.

Based on last homework, assume that the us-states-covid.csv is already downloaded and in the project.

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

```
library(tidyverse)
```

```
── Attaching core tidyverse packages ──────────────────────── tidyverse 2.0.0 ──
✔ dplyr     1.1.4     ✔ readr     2.1.5
✔ forcats   1.0.1     ✔ stringr   1.5.2
✔ ggplot2   4.0.0     ✔ tibble    3.3.0
✔ lubridate 1.9.4     ✔ tidyr     1.3.1
✔ purrr     1.1.0
── Conflicts ────────────────────────────────────────── tidyverse_conflicts() ──
✖ dplyr::filter() masks stats::filter()
✖ dplyr::lag()    masks stats::lag()
ℹ Use the conflicted package (<http://conflicted.r-lib.org/>) to force all conflicts to
become errors
```

```
library(plotly)
```

```
Attaching package: 'plotly'

The following object is masked from 'package:ggplot2':

    last_plot

The following object is masked from 'package:stats':

    filter

The following object is masked from 'package:graphics':

    layout
```

```
covid <- read_csv("us-states-covid.csv")
```

```
Rows: 61942 Columns: 9
── Column specification ────────────────────────────────────────────────────────
Delimiter: ","
chr  (2): geoid, state
dbl  (6): cases, cases_avg, cases_avg_per_100k, deaths, deaths_avg, deaths_a...
date (1): date
```

```
ℹ Use `spec()` to retrieve the full column specification for this data.
ℹ Specify the column types or set `show_col_types = FALSE` to quiet this message.
```

```r
# a major peak is defined as a peak 2SD+
# a minor peak is defined as a peak 1SD+
full_national <- covid %>%
  # sum all cases from the same day
  group_by(date) %>%
  summarize(total_avg_cases = sum(cases_avg, na.rm = TRUE),
            total_cases = sum(cases, na.rm = TRUE),
            .groups = "drop")

# plot(full_national$total_avg_cases)
# plot(full_national$total_cases)

mean_national <- base::mean(full_national$total_avg_cases, na.rm = TRUE)
sd_national   <- sd(full_national$total_avg_cases, na.rm = TRUE)
full_national <- full_national %>%
  mutate(
    z_score  = (total_avg_cases - mean_national) / sd_national
  )

# mark as major and minor peak
full_national <- full_national %>%
  mutate(peak_type = case_when(
    (z_score > 0 & z_score < 2) ~ "minor",
    (z_score > 2) ~ "major",
    TRUE ~ NA))

table(full_national$peak_type, useNA = "ifany")
```

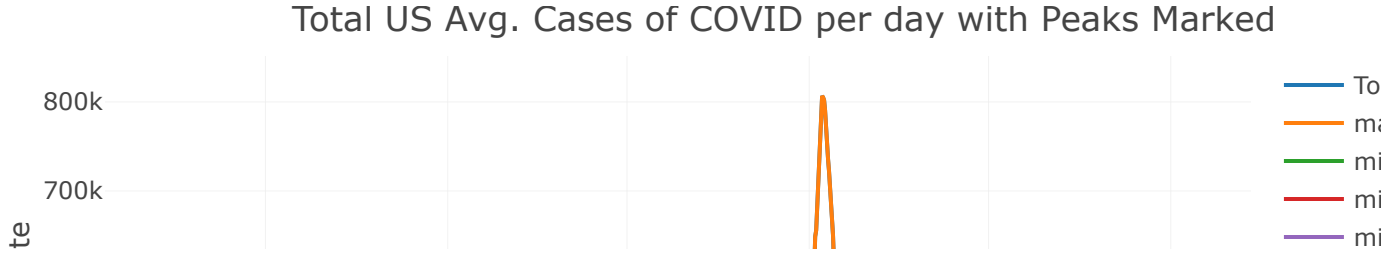
```
major minor  <NA>
   37   337   784
```

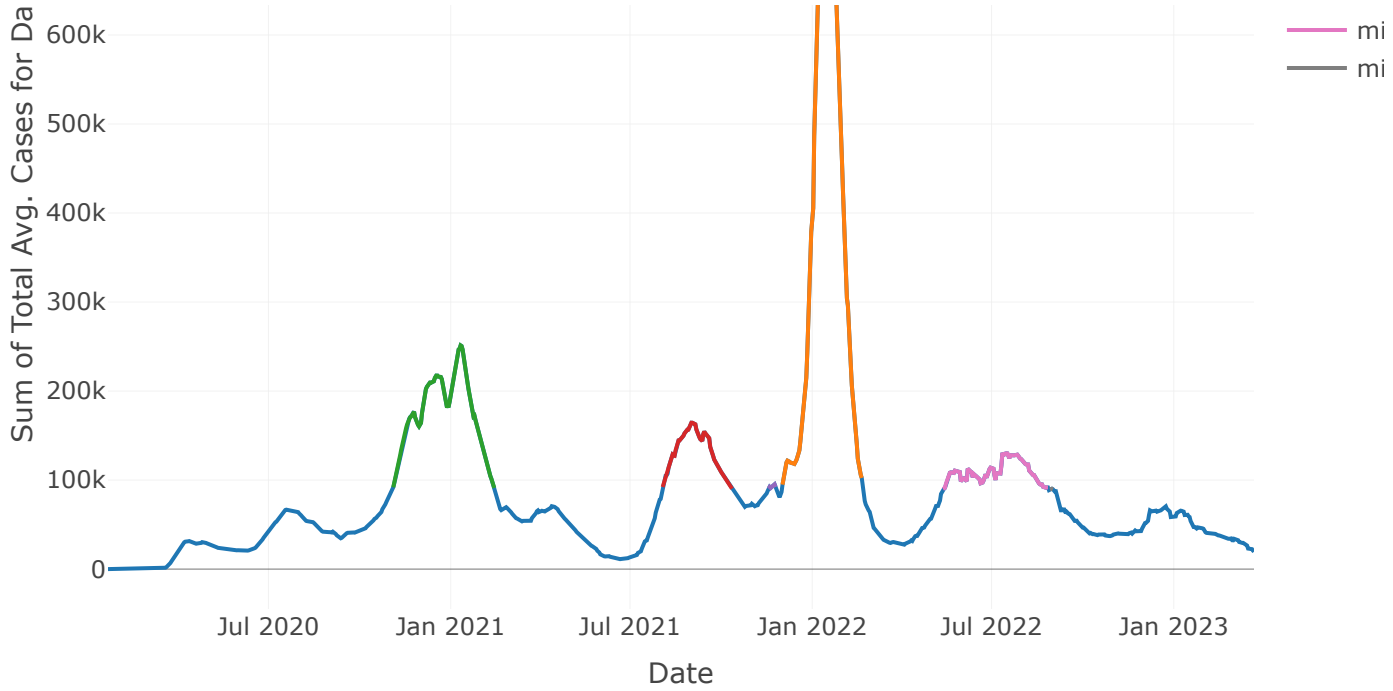
```r
# group major and minors together in the same timing
# Identify peaks and assign interval groups

full_national <- full_national %>%
  # create a change marker for each peak change grouping
  mutate(peak_group = NA,
         peak_change = ifelse((!is.na(peak_type) & is.na(lag(peak_type))), 1, 0))

#table(full_national$peak_type)
#table(full_national$peak_change)
#summary(full_national$peak_change)

full_national <- full_national %>%
  mutate(peak_group = ifelse((peak_change == 1), cumsum(peak_change), NA)) %>%
  # now count all the majors or minors after that in the same group
```

```
  group_by(gr = cumsum(is.na(peak_type))) %>%  # group by stretches between NAs
  mutate(peak_group = ifelse(!is.na(peak_type), first(peak_group[!is.na(peak_group)]), NA
  ungroup()
table(full_national$peak_group)
```

```
  1   2   3   4   5   6
103  71   8  81 107   4
```

```
# replace all minor peaks in group 5 with major because its the same peak
full_national <- full_national %>%
  mutate(peak_type = ifelse(peak_group == 4, "major", peak_type))


# plot using plotly
plot1 <- full_national %>%
  plot_ly(x = ~date, y = ~total_avg_cases, type = 'scatter', mode = 'lines',
          name = 'Total Avg. Cases', hoverinfo = 'x+y') %>%
  layout(title = 'Total US Avg. Cases of COVID per day with Peaks Marked',
         xaxis = list(title = 'Date'),
         yaxis = list(title = 'Sum of Total Avg. Cases for Date'))

# Filter only rows that are part of a peak
# separate major and minor peaks
major_peak <- full_national %>%
  filter(!is.na(peak_group) & peak_type == "major")
minor_peak <- full_national %>%
  filter(!is.na(peak_group) & peak_type == "minor")

# trace the peaks
plot1 <- plot1 %>%
  add_trace(data = major_peak, x = ~date, y = ~total_avg_cases,
      type = 'scatter', mode = 'lines', name = ~peak_type)
# for each minor peak
for(g in 1:6){
  minor_group <- minor_peak %>%
    filter(peak_group == g)
  plot1 <- plot1 %>%
    add_trace(data = minor_group, x = ~date, y = ~total_avg_cases,
      type = 'scatter', mode = 'lines', name = "minor")
}

plot1
```





Based on my results, there is one major peak which was at least 2 standard deviations from the mean and there were five minor peaks which were around 1 standard deviation from the mean.

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

```
# compute each state's overall average case rate per 100k
state_summary <- covid %>%
  group_by(state) %>%
  summarise(avg_rate_per_100k = base::mean(cases_avg_per_100k, na.rm = TRUE))

# identify states with highest and lowest average rates
top_states <- state_summary %>%
  slice_max(avg_rate_per_100k, n = 5) %>%
  select(state)
bottom_states <- state_summary %>%
  slice_min(avg_rate_per_100k, n = 5) %>%
  select(state)

top_states
```

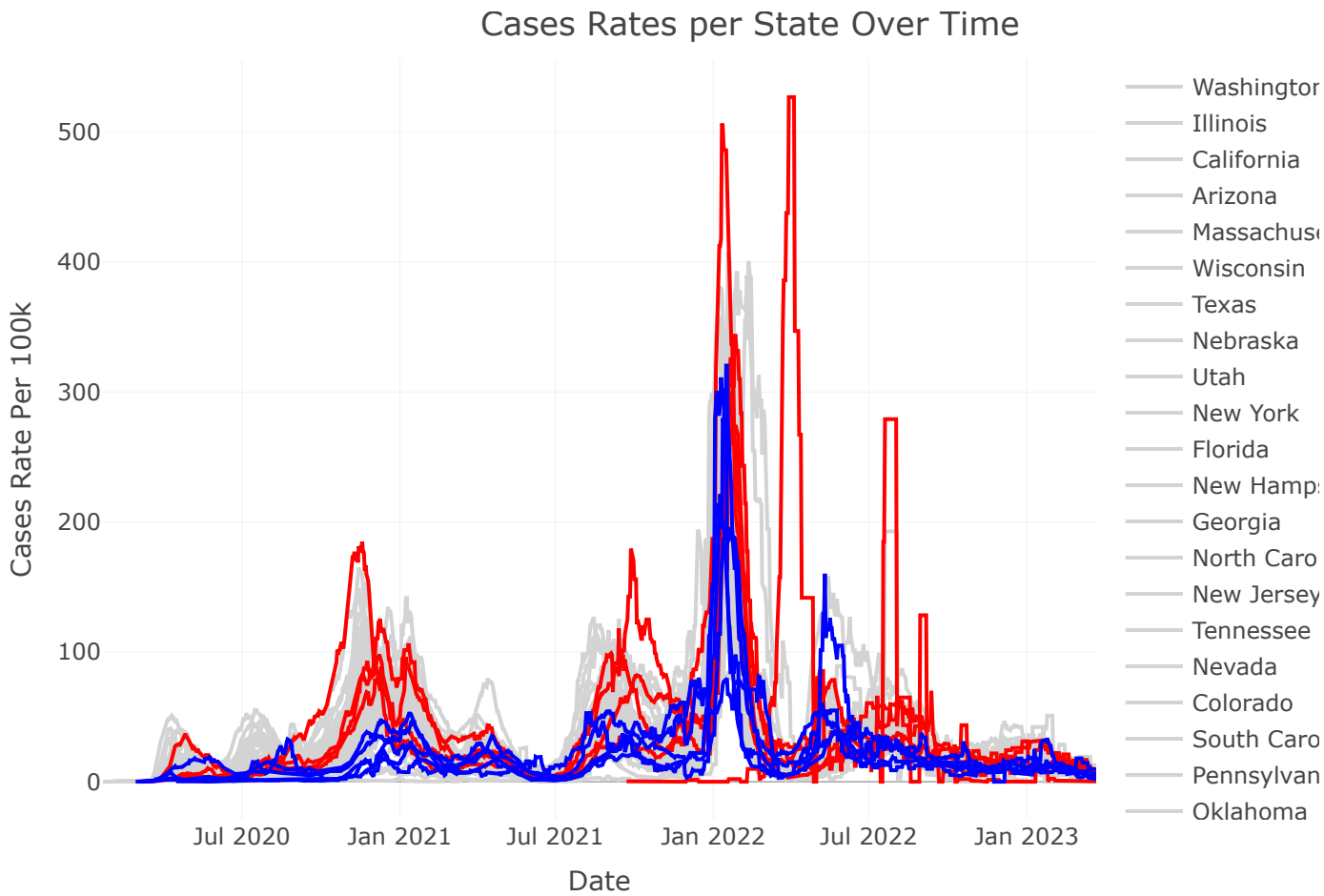
```
# A tibble: 5 × 1
  state
  <chr>
1 American Samoa
2 Rhode Island
3 Alaska
4 Kentucky
5 North Dakota
```

```
bottom_states
```

```
# A tibble: 5 × 1
  state
  <chr>
1 Maryland
2 Maine
3 Oregon
4 Virgin Islands
5 Vermont
```

```
# filter to only states of interest
top_data <- covid %>%
  filter(state %in% top_states$state)
bottom_data <- covid %>%
  filter(state %in% bottom_states$state)

# plot using plotly
# plot first state
state_plot <- covid %>%
  filter(!(state %in% top_states$state)) %>%
  filter(!(state %in% bottom_states$state)) %>%
  filter(state == unique(covid$state)[1])
plot2 <- state_plot %>%
  plot_ly(x = ~date, y = ~cases_avg_per_100k, type = 'scatter', mode = 'lines',
          name = ~state, hoverinfo = 'x+y',
          line = list(color = 'lightgray', width = 2)) %>%
  layout(title = 'Cases Rates per State Over Time',
         xaxis = list(title = 'Date'),
         yaxis = list(title = 'Cases Rate Per 100k'))

# plot all the rest of the  states with rates
for(a in 2:length(unique(covid$state))){
  # pull each state
  state_plot <- covid %>%
    filter(!(state %in% top_states$state)) %>%
    filter(!(state %in% bottom_states$state)) %>%
    filter(state == unique(covid$state)[a])
  # plot that state
  plot2 <- plot2 %>%
    add_trace(data = state_plot, x = ~date, y = ~cases_avg_per_100k, type = 'scatter',
              mode = 'lines', hoverinfo = 'x+y',
              line = list(color = 'lightgray', width = 2))
}

# now overlay top states in red
for(a in 1:length(unique(top_data$state))){
  # pull each state
  state_plot <- top_data %>%
    filter(state == unique(top_data$state)[a])
  # plot that state
  plot2 <- plot2 %>%
```

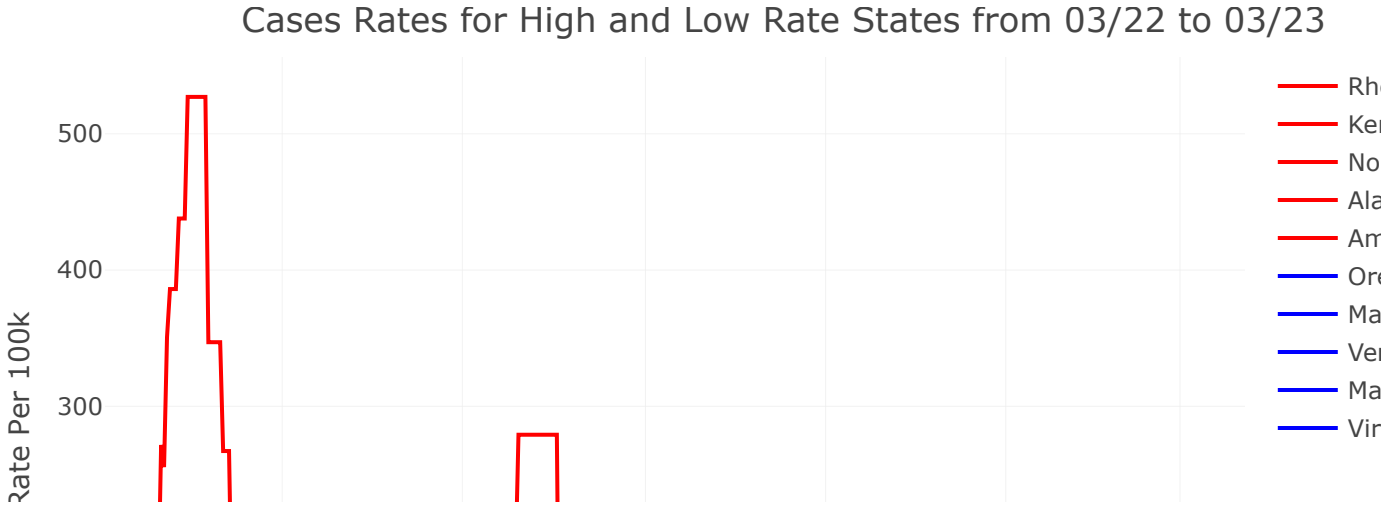
```
    add_trace(data = state_plot, x = ~date, y = ~cases_avg_per_100k, type = 'scatter',
              mode = 'lines', name = ~state, text = ~state, hoverinfo = 'text+x+y',
              line = list(color = 'red', width = 2))
}

# now overlay bottom states in blue
for(a in 1:length(unique(bottom_data$state))){
  # pull each state
  state_plot <- bottom_data %>%
    filter(state == unique(bottom_data$state)[a])
  # plot that state
  plot2 <- plot2 %>%
    add_trace(data = state_plot, x = ~date, y = ~cases_avg_per_100k, type = 'scatter',
              mode = 'lines', name = ~state, text = ~state, hoverinfo = 'text+x+y',
              line = list(color = 'blue', width = 2))
}

plot2
```

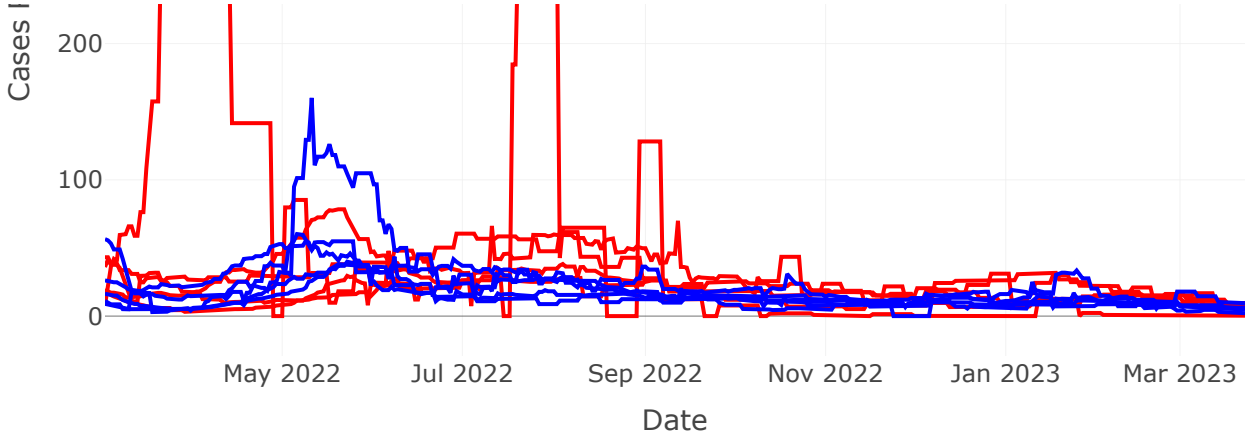


This plot clearly shows that states with high rates peaked early and tended to have larger peaks prior to 2023, but it hrd to see the differences towards the end trajectory so I will plot those too to see

```
state_plot2023 <- top_data %>%
  filter(date > "2022-3-01") %>%
  filter(state == unique(top_data$state)[1])
```

```
plot3 <- state_plot2023 %>%
  plot_ly(x = ~date, y = ~cases_avg_per_100k, type = 'scatter', mode = 'lines',
          name = ~state, text = ~state, hoverinfo = 'text+x+y',
          line = list(color = 'red', width = 2)) %>%
  layout(title = 'Cases Rates for High and Low Rate States from 03/22 to 03/23',
         xaxis = list(title = 'Date'),
         yaxis = list(title = 'Cases Rate Per 100k'))

# now overlay top states in red
for(b in 2:5){
  # pull each state
  state_plot2023 <- top_data %>%
    filter(date > "2022-3-01") %>%
    filter(state == unique(top_data$state)[b])
  # plot that state
  plot3 <- plot3 %>%
    add_trace(data = state_plot2023, x = ~date, y = ~cases_avg_per_100k,
              type = 'scatter', mode = 'lines', name = ~state, text = ~state,
              hoverinfo = 'text+x+y',
              line = list(color = 'red', width = 2))
}

# now overlay bottom states in blue
for(a in 1:5){
  # pull each state
  state_plot2023 <- bottom_data %>%
    filter(date > "2022-3-01") %>%
    filter(state == unique(bottom_data$state)[a])
  # plot that state
  plot3 <- plot3 %>%
    add_trace(data = state_plot2023, x = ~date, y = ~cases_avg_per_100k,
              type = 'scatter', mode = 'lines', name = ~state, text = ~state,
              hoverinfo = 'text+x+y',
              line = list(color = 'blue', width = 2))
}

plot3
```

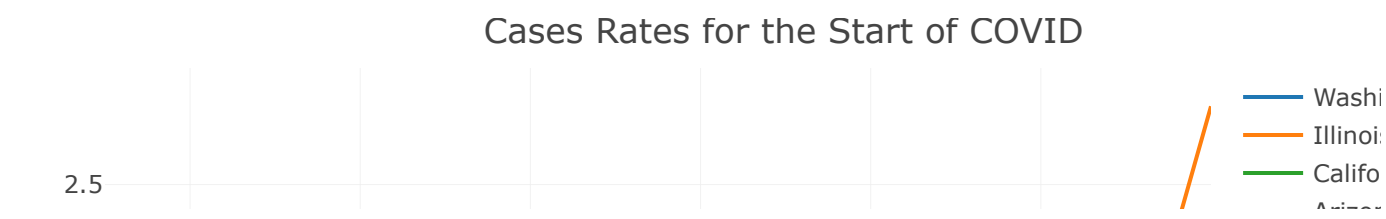


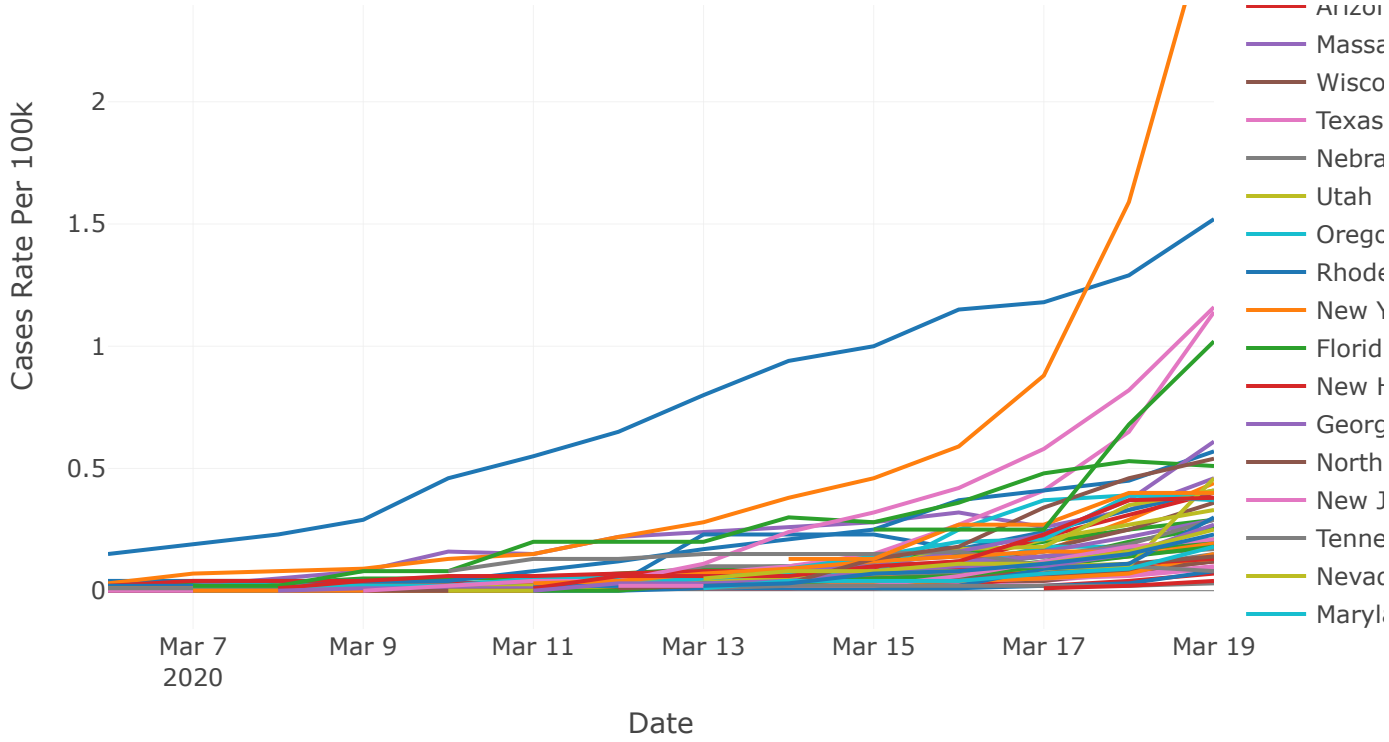


In the last year of data, you can see that the rates of covid actually fluctuate more for more high rate states than low rate states, but you can see overall the trend is the same for both in end trajectories.

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

```
# a substantial way is definitely cases per population or case_avg_per_100k
# zoom up on my plot for the first year of covid
state_rates <- covid %>%
  filter(date < "2020-03-20" & date > "2020-03-05") %>%
  filter(state == unique(covid$state)[1])

plot4 <- state_rates %>%
  plot_ly(x = ~date, y = ~cases_avg_per_100k, type = 'scatter', mode = 'lines',
          name = ~state, text = ~state, hoverinfo = 'text+x+y') %>%
  layout(title = 'Cases Rates for the Start of COVID',
         xaxis = list(title = 'Date'),
         yaxis = list(title = 'Cases Rate Per 100k'))

# now for the rest of the states overlay
for(a in 2:length(unique(covid$state))){
  # pull each state
  state_rates<- covid %>%
    filter(date < "2020-03-20" & date > "2020-03-05") %>%
    filter(state == unique(covid$state)[a])
  # plot that state
  plot4 <- plot4 %>%
    add_trace(data = state_rates, x = ~date, y = ~cases_avg_per_100k, type = 'scatter',
              mode = 'lines', hoverinfo = 'x+y')
}

plot4
```

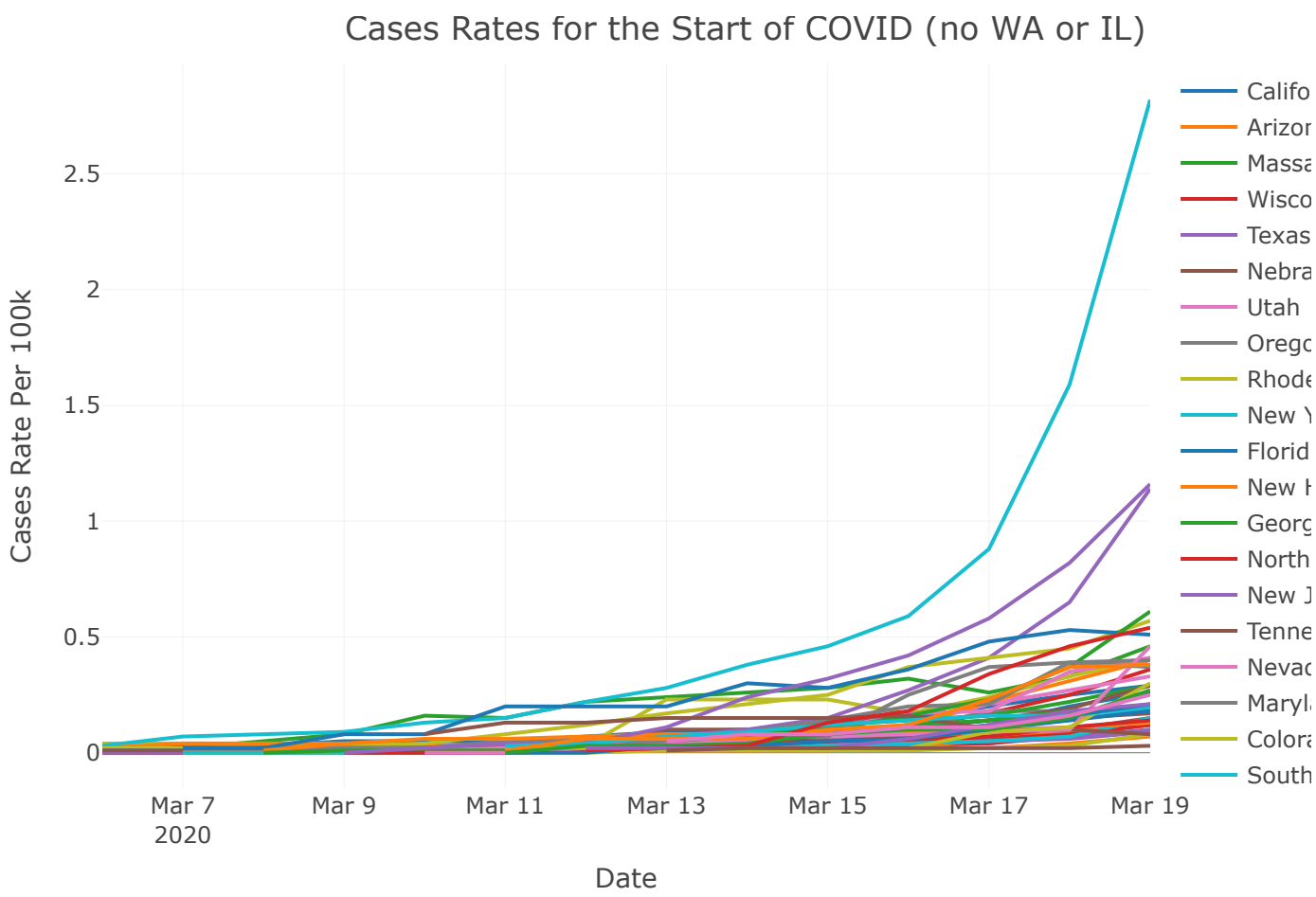




Based on the plot, Washington and Illinois are definitely at the top, but it is hard to tell from the other states, I will remove them to see what other states show up

```
# filtering out washington and Illinois
covid_noILorWA <- covid %>%
    filter(!(state == "Washington")) %>%
   filter(!(state == "Illinois"))

state_rates <- covid_noILorWA %>%
  filter(date < "2020-03-20" & date > "2020-03-05") %>%
  filter(state == unique(covid_noILorWA$state)[1])

plot5 <- state_rates %>%
  plot_ly(x = ~date, y = ~cases_avg_per_100k, type = 'scatter', mode = 'lines',
          name = ~state, text = ~state, hoverinfo = 'text+x+y') %>%
  layout(title = 'Cases Rates for the Start of COVID (no WA or IL)',
         xaxis = list(title = 'Date'),
         yaxis = list(title = 'Cases Rate Per 100k'))

# now for the rest of the states overlay
for(a in 2:47){
  # pull each state
  state_rates<- covid_noILorWA %>%
    filter(date < "2020-03-20" & date > "2020-03-05") %>%
    filter(state == unique(covid_noILorWA$state)[a])
  # plot that state
  plot5 <- plot5 %>%
    add_trace(data = state_rates, x = ~date, y = ~cases_avg_per_100k, type = 'scatter',
              mode = 'lines', text = ~state, hoverinfo = 'text+x+y')
}
```

```
plot5
```



After filtering, I see NY, LA, NJ, DC, CO, and ME with the most significant first rates of covid

```
# now I will plot those states
state_list <- c("Washington", "Illinois", "New York", "Louisiana", "District of Columbia"
state_rates <- covid %>%
  filter(date < "2020-03-20" & date > "2020-03-05") %>%
  filter(state == "Washington")

plot6 <- state_rates %>%
  plot_ly(x = ~date, y = ~cases_avg_per_100k, type = 'scatter', mode = 'lines',
          name = ~state, text = ~state, hoverinfo = 'text+x+y') %>%
  layout(title = 'Early COVID Cases Rates for First 5 States',
         xaxis = list(title = 'Date'),
         yaxis = list(title = 'Cases Rate Per 100k'))

# now for the rest of the states overlay
for(a in 2:length(state_list)){
  # pull each state
  state_rates <- covid %>%
    filter(date < "2020-03-20" & date > "2020-03-05") %>%
    filter(state == state_list[a])
  # plot that state
  plot6 <- plot6 %>%
```

```
    add_trace(data = state_rates, x = ~date, y = ~cases_avg_per_100k, type = 'scatter',
              mode = 'lines', text = ~state, hoverinfo = 'text+x+y')
}

plot6
```