Lab 1: Pooled Covid Testing

Stephen R. Proulx, Taom Sakal

1/6/2021

Bayesian Statistical Modeling Winter 2021

Lab Exercise, Week 1

1/6/2021

When is this lab due? Labs are due on the Thursday after they are assigned. However, in many cases you can complete them during the lab period itself. This assignment is due on Thursday, 1/21/2021. You must answer all the problems, which are written in **bold**. Submit homework and labs to gradescope: https://www.gradescope.com/courses/485353

These problems follow up on the example of estimating the percentage of the earth's surface that is water. But first we will briefly go over how to use R Markdown. Feel free to skip this section if you are already wise in its ways.

How to R Markdown (A very very short tutorial)

R Markdown blends text with code. This is a great way to communicate understandable code to others (and to future you once you inevitably forget what you wrote). This is text right now. Below is a code chunk.

```
print("Roses are red,")
```

```
## [1] "Roses are red,"
```

```
print("violets are blue,")
```

```
## [1] "violets are blue,"
```

```
print("R's pretty rad,")
```

```
## [1] "R's pretty rad,"
```

```
print("and Bayes' rule rules.")
```

```
## [1] "and Bayes' rule rules."
```

Put your cursor on one line of the code chunk and press *ctrl-enter* (the actual keystroke may be version specific) to run it. Or place your cursor anywhere in the chunk and press *ctrl-shift-enter* to run the entire thing. You can also press the green play button at the top right of the chunk.

Try creating your own code chunk. You can either type out the code or you can press *ctrl-alt-I* (or *cmd-option-I* for Mac). Below is an empty chunk but try creating your own and running it.

You'll notice the chunks in the code below have names. You can navigate by chunk name in the tiny pop-up menu at the bottom of this editor window.

```
# This chunk has a name!
# These lines with '#' are comments.
# The computer ignores them but humans don't.
# You can use them to clarify code.
#
# It is good coding style to use comments
# to clarify any confusing lines.

2 + 3 + 5  # Addition
```

```
## [1] 10
```

```
2 * 3 * 5  # Multiplication
```

```
## [1] 30
```

```
(2 ^ 3) ^ 5  # Exponentiation
```

```
## [1] 32768
```

There's also a nice document outline button on the top right corner of the editor window. This will let you jump around this R markdown file with ease.

We can save data into a variable to use later. This is the '=' notation. You'll often see '<-' used in R too. We'll try to use '<-' as it is clearer and most style guides recommend it. (But both your TA and professor come from programming traditions where '=' is the norm, so we'll slip up. And many people use the '=' notation, so should be able to read both.)

```
# Three variables
a <- 42
b = 42
a <- TRUE # Overwrite a with a new value
word_variable <- 'words go in quotes!'  # Variable names don't have spaces

# A list of the three variables
list_of_data <- list(a, b, word_variable)
print(list_of_data)
```

```
## [[1]]
## [1] TRUE
##
## [[2]]
## [1] 42
##
## [[3]]
## [1] "words go in quotes!"
```

You can see all active variables in the *Environment* panel on your right. These will stay as they are until you change them.

Finally, you can create a pdf of everything by pressing the Knit button at the top of the editor. You'll need to run the code chunk below to get that button to appear. You'll be turning in the original R Markdown file along with a pdf generated from knitr. Try this now. (If the button just looks like "preview" then choose "Knit to pdf" and save it.)

That's all you need to know for this lab. We've skipped a lot but we'll introduce more concepts throughout the class and labs. As you code try your best to use a consistent style. It makes things more readable for others and is a good habit as you go further into science. The R tidyverse style guide is at https://style.tidyverse.org/.

As you have time you can check it out to learn how to space and indent things. Otherwise explore R Studio. Don't be afraid to press buttons and mess around.

Lab Problem

In a not-so-far-off dystopian future COVID-20 has appeared. It's everywhere and we only have one test for it: a PCR spit test. This test is magical and can perfectly detect any COVID material in any sample of spit.

Sadly, magic is expensive and these tests are limited. While we'd like to administer the test to everybody there's simply not enough. Is there any way to do better than one test per person while still identifying everybody who has COVID?

One potential strategy is to pool k people's spit and test the pool. If even one person in the group has covid then the test will come back positive and we re-test all k people individually to find out who it is. Otherwise if the test came back negative we know nobody has COVID. In this way each pool that is positive generates *k+1* tests but a pool that is negative generates *1* test.

This pooling strategy sounds promising, but does it really use fewer tests than than testing each person individually? And if so what k is the best pool size?

We can answer this through R. First we will simulate real life data by creating a bunch of people and giving some of them COVID. We can then take those people and see how the pooling strategy does, and try the pooling with many values for k.

Create Data

Before we create the data we need to setup R. Running the code chunk below will give us all we need. If you are not on the surver then you might need to install the tidyverse package. In R studio you can do this by going to Tools > Install Packages.

```
library(tidyverse)  # Import the tidyverse package
```

```
## Warning: package 'tidyverse' was built under R version 4.1.2
```

```
## -- Attaching packages --------------------------------------- tidyverse 1.3.2 --
## v ggplot2 3.3.6     v purrr   0.3.4
## v tibble  3.1.7     v dplyr   1.0.9
## v tidyr   1.2.0     v stringr 1.4.0
## v readr   2.1.2     v forcats 0.5.1
```

```
## Warning: package 'ggplot2' was built under R version 4.1.2
```

```
## Warning: package 'tibble' was built under R version 4.1.2
```

```
## Warning: package 'tidyr' was built under R version 4.1.2
```

```
## Warning: package 'readr' was built under R version 4.1.2
```

```
## Warning: package 'dplyr' was built under R version 4.1.2
```

```
## -- Conflicts ------------------------------------------ tidyverse_conflicts() --
## x dplyr::filter() masks stats::filter()
## x dplyr::lag()    masks stats::lag()
```

```
knitr::opts_chunk$set(echo = TRUE)  # ready knitr
```

Now we can simulate some data. We will first create a bunch of pools and fill them with people. We will call the pools "samples," as each are samples of our data.

```
k <- 10  # Number of people in a pool
samps <- 10000  # The total number of pools we test.
```

Some of these pools will have people that test positive and some will not. Because each person in the pool has the same, independent, chance of having COVID we can get the number of sick people in the pool by drawing from a bionomial distribution. You you don't remember what the bionomial distribution is or why we use it see this video: https://www.youtube.com/watch?v=8idr1WZ1A7Q&t=50s&ab_channel=3Blue1Brown. (This video also builds up Bayesian ideas, so it and the channel's other probability videos are worth a watch even if you do remember.)

```
pos_rate <- 0.05  # The rate at which people have COVID. This is now a variable and is saved.
rbinom(1, size = k, prob = pos_rate)  # This randomly picks the number of people with covid in one pool
```

```
## [1] 0
```

The *rbinom(1, size=k,prob=pos_rate)* simulates the number of people in one pool where each person has a *pos_rate* of having covid. As a reminder, something like *rbinom()* is a function. It takes in arguments and gives an output. This one stands for *random binomial* and will return a random draw from a binomial distribution. Try rerunning the *rbinom* line above a couple times. It won't always give the same value.

If we replace 1 with another number then this will give a *vector* of numbers. We can use variables for the function arguements or write in the numbers directly.

```
# Make 42 random draws from the binomial distribution.
rbinom(42, size = k, prob = pos_rate)
```

```
##  [1] 0 1 0 1 1 1 0 0 1 0 1 2 0 2 0 0 0 0 0 0 0 0 1 0 1 0 0 0 0 0 1 0 0 1 0 1 1 1
## [39] 1 0 1 0
```

```
# Make 10 random draws, with positive rate of 50%.
rbinom(10, size = k, prob = 0.5)
```

```
##  [1] 5 5 4 6 9 6 5 8 4 3
```

Problem 1 Make a vector of 200 random draws from a binomial distribution. Let the size of the pools be 100 and the positive rate be 50%.

```
# Write your answer here.
# Make 200 random draws with 100 people in the pool, with positive rate of 50%.
rbinom(200, size = 100, prob = 0.5)
```

```
##   [1] 47 42 52 43 46 48 52 43 49 50 57 56 47 47 51 47 54 53 47 43 54 49 45 48 47
##  [26] 50 52 45 47 49 50 55 53 42 51 46 57 49 50 47 52 56 46 43 55 50 42 47 52 55
##  [51] 45 46 50 44 53 50 51 55 59 46 48 52 51 49 48 47 64 47 44 53 45 49 46 51 51
##  [76] 57 49 50 48 50 42 50 44 42 54 46 59 49 43 46 48 45 47 47 47 53 45 51 50 52
## [101] 50 57 55 50 54 44 48 62 45 50 48 39 51 51 44 51 42 49 49 51 58 64 56 51 40
## [126] 49 55 44 50 47 55 56 51 47 50 46 45 53 43 47 47 43 46 50 55 51 48 52 64 54
## [151] 59 51 42 55 51 58 51 52 44 45 47 50 54 46 57 45 50 53 45 62 56 48 54 55 43
## [176] 44 57 55 55 58 43 45 46 47 53 53 50 52 42 45 45 47 59 54 57 50 46 54 54 51
```

```
# each value is the number of people in the pool that are positive for covid-20
```

Problem 2 We can look up the documentation for a function by putting your cursor on a function and pressing *F1*. Or you can type *?function_name* and run it. The documentation will appear in the lower right-hand panel.

Using the documentation on *rbinom* will pull up documentation for many binomial functions R has. There are four of them, each of which do different things.What are their names?

```
?rbinom

# Write down the three other types of bionomial function's names.
```

```
# dbinom, density
# pbinom, distribution
# qbinom, quantile
```

Playing with the data

Now we can make a list of the number of people with COVID in all 10000 pools and arrange the data in a *tibble*. (A tibble is a more modern dataframe-like object which works well with the tidyverse.)

```
# Make a vector of all the pools,
# where the i'th pool has the number of people with COVID in it.
positive <- rbinom(n = samps, size = k, prob = pos_rate)
head(positive) #look at the data
```

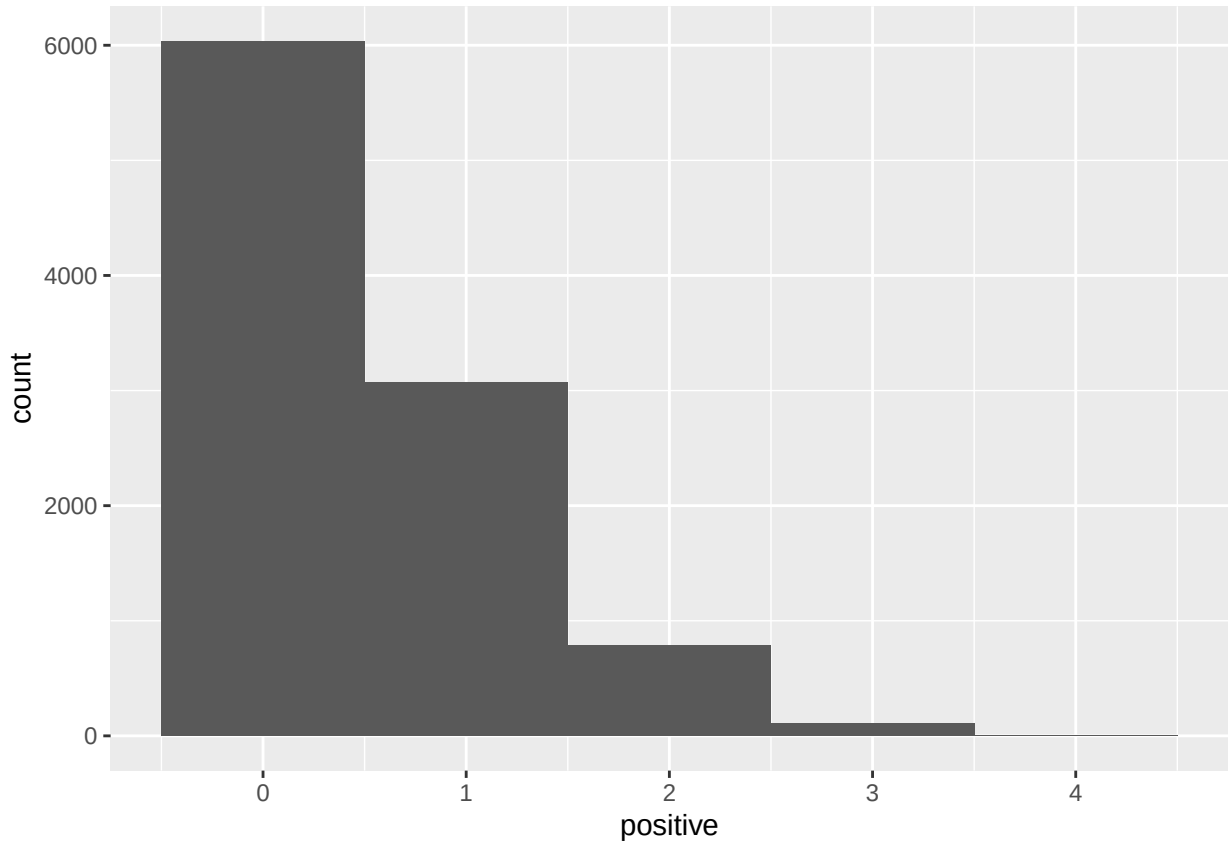
```
## [1] 0 0 0 2 1 1
```

```
# Arrange data in a tibble.
# Look in your Environment panel and click pools to see what this data looks like.
pools <- tibble(positive)
pools
```

```
## # A tibble: 10,000 x 1
##    positive
##       <int>
##  1        0
##  2        0
##  3        0
##  4        2
##  5        1
##  6        1
##  7        0
##  8        2
##  9        1
## 10        1
## # ... with 9,990 more rows
```

Next lets plot the number of people with covid in each pool. We will use the *ggplot2* package (which comes with the tidyverse) for visualization. If you need a refresher on it head here: https://r4ds.had.co.nz/data-visualisation.html

```
# Make a plot and say what data we will plot
ggplot(data = pools, aes(x = positive)) +
  geom_histogram(binwidth = 1)
```




```
# x-axis: number of people who have covid in their pool
# y-axis: count, aka how many pools have that many positive people
```

What we've just made is some simulated pool data that mimics what we expect from real life. We can now use this data to test our pooling strategy. First need to count up how many tests we have to do.

The code below makes a new column with the number of tests we would have to do for each pool. Any pool with at least one person with COVID will test positive, so in that pool we will have to retest everybody and will have a grand total of $1 + 10 = 11$ tests. Otherwise we spent one test for the entire pool, it came back negative, we know nobody has COVID, and we are done.

```
# 'mutate' adds two columns to the data frame, one called 'retest'
# and one called 'tests'. These columns are filled with the values provided.
# You can look at the pools variable before and after running this to see what changed!
pools <- mutate(pools, retest = sign(positive) , tests = 1 + retest * k)

# Take mean number of tests needed
mean(pools$tests) / k
```

```
## [1] 0.4963
```

Problem 3 What does the *sign* function in mutate do? Use the documentation to find out. Write your answer as text below.

< Sign gives the "sign" of your value, if it's positive it's a 1, if its a negative its -1, if it's 0 it's 0 >

Piping

A quick aside for some common syntax: the pipe. This is one of the great boons of R that many other languages don't have. It makes a lot of code more readable. We could rewrite the mutate code above with pipe syntax

```
pools <- pools %>%
  mutate(retest = sign(positive) , tests = 1 + retest * k)
```

We can chain pipes together. The code below takes an imaginary number, then takes the real part of it, then takes the square root, then finally prints it.

```
z = -10 + 3i  # The imaginary number 2 + 3i

z %>%
  Re() %>%  # Take the real part...
  abs() %>% # then take the absolute value...
  log(base = 10)  # then take log base ten.
```

```
## [1] 1
```

```
# Same thing but now save it as the variable a
a <- z %>%
  Re() %>%  # Take the real part...
  abs() %>% # then take the absolute value...
  log(base = 10)  # the take the log base ten.

print(a)
```

```
## [1] 1
```

Piping is encouraged as it is more readable than the typical nested way of writing this.

```
z = -10 + 3i

a = log(abs(Re(z)), base = 10)
```

Problem 4 Take the pipe code and modify it to take the square root after the absolute value but before taking the log. Also use the natural log instead of base 10. You might have to use Google to figure our how to do this. (But half of programming is being great at looking things up.) You should get 1.151293 as your final answer.

```
# Modify this code.

z = -10 + 3i  # The imaginary number 2 + 3i

z %>%
  Re() %>%  # Take the real part...
  abs() %>% # then take the absolute value...
  sqrt() %>% # then take the square root
  log()  # then take the natural log (which is the defaut)
```

```
## [1] 1.151293
```

Functions

Taking the mean number of tests we need per group is a good metric – the lower the mean the more efficient our pooling strategy is. We could rerun the code above to test different values of k and see which is best, but

it is easier to make our own function that takes in a value for k and return the average number of tests per person.

```
# This function takes in a pool size k and
# returns the mean number of tests needed
# to identify everybody that has COVID.

calc_tests <- function(k) {

  pos_rate <- 0.05 # positivity rate of covid
  samps <- 10000 # number of pools we are sampling
  positive = rbinom(n = samps, size = k, prob = pos_rate)  # Make new data for each run
  pools <-  tibble(positive)

  pools <- pools %>%
    mutate(retest = sign(positive) , tests= 1 + retest * k)

  output <- mean(pools$tests) / k # Return the mean return the average number of tests per person which

  output

}
```

We can run this function like so.

```
calc_tests(1)  # Mean tests needed for when pools have two people each, inputting k
```

```
## [1] 1.0501
```

Problem 5 Try changing the pool size in the chunk above to see how the mean number of tests change. Why do the numbers change each time we run it? Why is *calc_tests(1)* not exactly equal to 1?

< In this setup, if you test positive even in a pool of 1, our code says you will retest (even though logically that doesn't make sense, we have no stipulation in the code for a pool size of 1 not requiring a retest), and thus using 2 tests, so we expect a mean higher than 1, and there is a 0.05 chance of being Bd positive, so a value of ~1.05 makes sense >

The best pool size

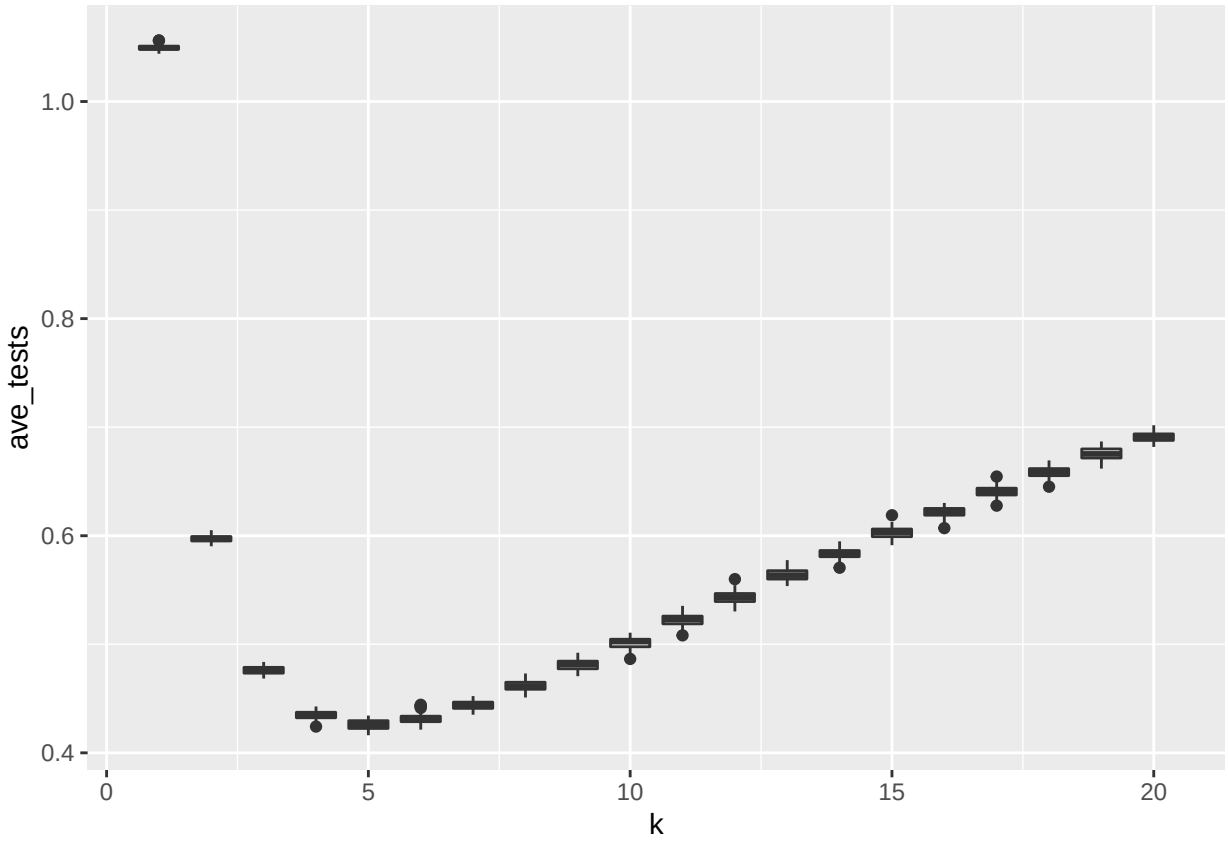
Now armed with the *calc_test* function we can figure out which k is best. This is a simple matter of running through a bunch of possible pool sizes and recording the mean tests needed. Because each run is random we will take the average of 100 runs.

For now you don't need to understand the nitty gritty of the code. But if you're feeling daring then use the documentation, view the variables that are made, and try to reverse engineer the code.

```
# make a new tibble that contains the pool size *k*
# for each replicate and does each replicate 100 times.
keffects <- tibble(k = rep(seq(20), 100))

# Calculate the average number of tests needed
keffects <- keffects %>%
  rowwise() %>%
  mutate(ave_tests = calc_tests(k), poolsize = as.integer(k))  # convert *k* to an integer to get geom_
```

```
ggplot(data = keffects, aes(x = k, y = ave_tests, group = poolsize)) +
  geom_boxplot()
```



Problem 6

Does the pooling strategy use less tests than the naive one-test-per-person strategy? If so, what is the best pool size k to use?

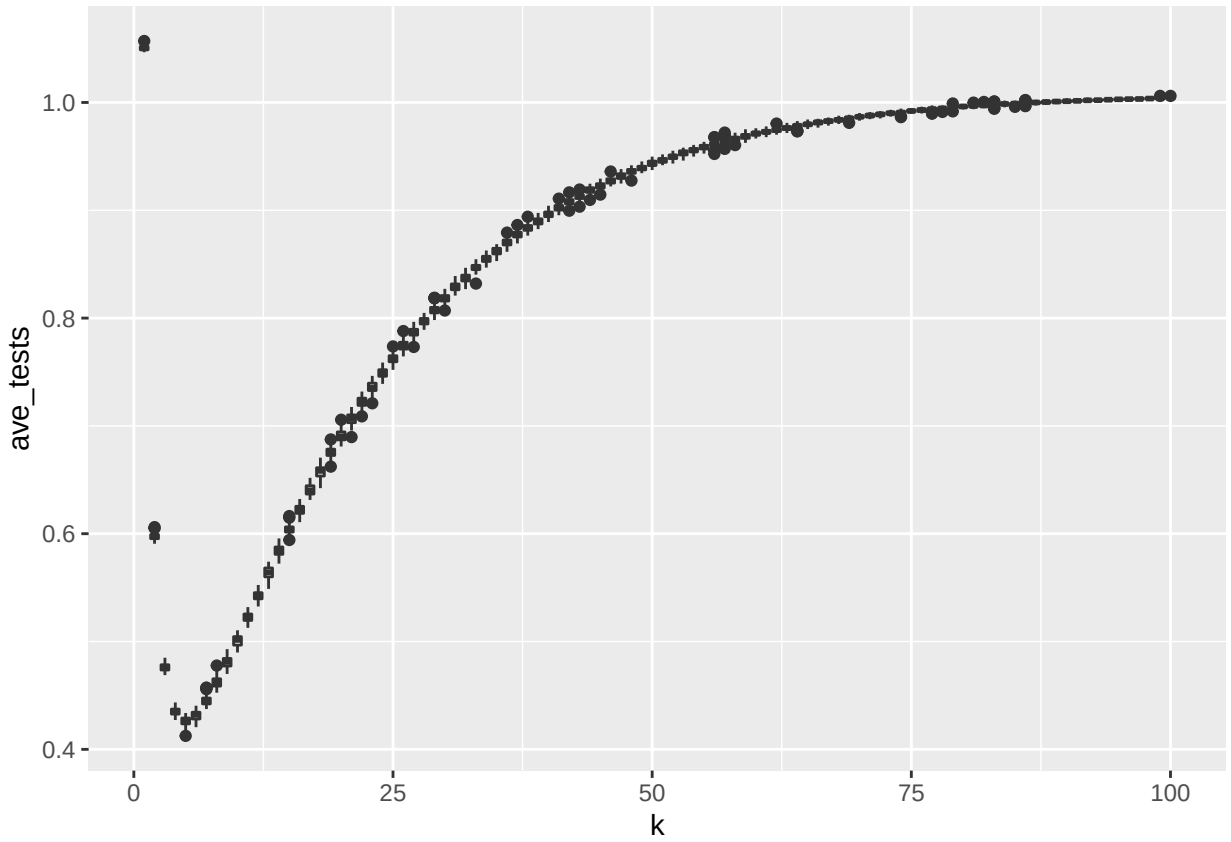
Using one person per test would result in more tests as long as the pool includes more than one person (since using the pooling method for a pool of one would result in more than 1 test per person, since a positive requires a retest). The best pool size, k, is 5 people per pool, as it has the lowest mean number of tests per person

Problem 7 (Bonus) Modify the above code to try up to pool size 100. What do you see? (This will take a few minutes to run!)

```
# make a new tibble that contains the pool size *k*
# for each replicate and does each replicate 100 times.
keffects <- tibble(k = rep(seq(100), 100))

# Calculate the average number of tests needed
keffects <- keffects %>%
  rowwise() %>%
  mutate(ave_tests = calc_tests(k), poolsize = as.integer(k))  # convert *k* to an integer to get geom_


ggplot(data = keffects, aes(x = k, y = ave_tests, group = poolsize)) +
  geom_boxplot()
```



When the pool size is 5 or smaller, the average number of tests per person is lower, and thus, is more efficient. As th epool size grown, it becomes less efficient than a pool of five, but still more efficient than the one test per person, up to a pool of around 80 people. When the pool is above 80 people, it is close to a mean of one test per person, which is then comparable to the classic one test per person.