Homework 6

Instructions

There are six exercises below. You are required to provide five solutions, with the same options for choosing languages as with the last exercise. You can provide solutions in two languages for one exercise only (for example, Ex. 1,2,3,5 in R and Ex. 1 in SAS is acceptable, Ex. 1,2,3 in SAS and Ex. 1,2 in R is not).

Warning I will continue restricting the use of external libraries in R, particularly `tidyverse` libraries. You may choose to use `ggplot2`, but take care that the plots you produce are at least as readable as the equivalent plots in base R. You will be allowed to use whatever libraries tickle your fancy in the midterm and final projects.

Reuse

For many of these exercises, you may be able to reuse functions written in prior homework. Define those functions here. I'm also including data vectors that can be used in some exercises.

```
CaloriesPerServingMean <- c(268.1, 271.1, 280.9, 294.7, 285.6, 288.6, 384.4)
CaloriesPerServingSD <- c(124.8, 124.2, 116.2, 117.7, 118.3, 122.0, 168.3)
Year <- c(1936, 1946, 1951, 1963, 1975, 1997, 2006)
```

Exercise 1

Part a.

Write a function or macro named `print.cohen.d`. This function should accept a single parameter, `d` that will be assumed to represent Cohen's d. The function should :

1. Print `d` to the console. The `cat` function in R will generally produce a nicer looking output.
2. Print one of the following messages:

- `Effect is small` (if $d \leq 0.2$)
- `Effect is large` (if $d \geq 0.8$)
- `Effect is medium` (otherwise)

3. This function does not need an explicit return value.

Part b.

If you name the function `print.cohen.d`, R will automatically call (or *dispatch*) this function when printing an instance of class `cohen.d` (a variable `a` such that `class(a)=='cohen.d'`). To create an instance of this class, modify your Cohen's d function with code of the form

```
cohen.d <- function( .... ) {
  ret <- ... calculate a d from m_1, s_1, m_2, s_2
  class(ret) <- 'cohen.d'
```

```
  return(ret)
}
```

You will not be able to perform this step in SAS.

You may want to visit the supplementary lecture from the `Week 3 Functions and Macros` module, titled `Objects in R` to understand the `class` function.

Part c.

Repeat the calculations for Cohen's d from Homework 3, but this time invoking the `print.cohen.d` function. You will need to call this function explicitly in SAS. In R, this function should be automatically dispatched if you simply call `cohen.d` without assigning the results to a variable; that is, the three lines below should produce equivalent output

```
print.cohen.d(cohen.d(....))
print(cohen.d(....))
cohen.d(....)
```

1936 versus 2006

1936 versus 1997

1997 versus 2006

Exercise 2

Part a.

Write a function or macro to compute mean, standard deviation, skewness and kurtosis from a single vector of numeric values. You can use the built-in mean function, but must use one (and only one) for loop to compute the rest. Be sure to include a check for missing values. Note that computationally efficient implementations of moments take advantage of $(Y_i - \bar{Y})^4 = (Y_i - \bar{Y}) \times (Y_i - \bar{Y})^3$, etc.

See https://www.itl.nist.gov/div898/handbook/eda/section3/eda35b.htm for formula for skewness and kurtosis. This reference gives several definitions for both skewness and kurtosis, you only need to implement one formula for each. Note that for computing skewness and kurtosis, standard deviation is computed using N as a divisor, not $N - 1$.

Your function should return a list with `Mean`, `SD`, `Skewness` and `Kurtosis`. If you use IML, you will need to implement this as a subroutine and use call by reference; include these variables in parameter list.

Part b.

Test your function by computing moments for `POM` from `lacanne.csv`, for `ELO` from `elo.csv` or the combine observations from `SiRstvt`.

If you wish, compare your function results with the `skewness` and `kurtosis` in the `moments` package.

Exercise 2

Consider Newton's method to find a minimum or maximum value attained by a function over an interval. Given a function f, we wish to find

$$\max_{x \in [a,b]} f(x)$$

Start with an initial guess, x_0, then generate a sequence of guesses using the formula

$$x_{k+1} = x_k - \frac{f'(x_k)}{f''(x_k)}$$

where f' and f'' are first and second derivatives. We won't be finding derivatives analytically, instead, we will be using numerical approximations (*central finite differences*), given by

$$f' \approx \frac{f(x + \frac{h}{2}) - f(x - \frac{h}{2})}{h}$$
$$f'' \approx \frac{f(x+h) - 2f(x) + f(x-h)}{h^2}$$

where h is some arbitrary small value.

We will work with likelihood function $L(x; \mu, \sigma^2) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$. Let $\mu = m_{1936}$ be the mean Calories per Serving from 1936, and let $\sigma = s_{1936}$ be the corresponding standard deviation. We will wish to find the x_* that maximizes

$$\max L(x; m_{1936}, s^2_{1936})$$

Let the initial guess be $x_0 = 180$ and let $h = 0.1$. Calculate 10 successive x_k, saving each value in a vector. Print the final x_k. Why does this value maximize the likelihood function?

Part b.

Plot the sequence of x versus iteration number (k) as the independent variable. Add a horizontal line corresponding to m_{1936}. How many iterations are required until $|x_{k+1} - x_k| < 10^{-6}$ (It should be less than 10)?

Exercise 4

Consider the Trapezoidal Rule for integration. From "Analysis by Its History" (https://books.google.com/books/about/Analysis_by_Its_History.html?id=E2IhMXPZMNIC)

> On the interval $[x_i, x_{i+1}]$ the function $f(x)$ is replaced by a straight line passing through $(x_i, f(x_i))$ and $(x_{i+1}, f(x_{i+1}))$. The integral between x_i and x_{i+1} is then approximated by the trapezoidal area $h \cdot (f(x_i) + f(x_{i+1}))/2$ and we obtain

$$\int_a^b f(x)dx = F(x) \approx \sum_{i=0}^{N-1} \frac{h}{2} \left(f(x_i) + f(x_{i+1})\right)$$

We will calculate the integral for the normal pdf

$$\int_{-3}^{3} L(x;\mu,\sigma^2)dx = \int_{-3}^{3} \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(x-\mu)^2}{2\sigma^2}} dx$$

with $\mu = 0$ and $sigma = 1$, using your `norm.pdf` function. We will do this by creating a sequence of approximations, each more precise than the preceding approximation.

Part a.

Calculate a first approximation of step size $h_0 = 1$, using the sequence of $x_i = \{-2.0, -1.0, 0.0, 1.0, 2.0\}$. Let this approximation be F_0. Print the first approximation.

Part b.

Continue to calculate a series of approximations $F_0, F_1, F_2, \ldots$ such that F_{k+1} improves on F_k by increasing N. Do this by decreasing the step size by 2, $h_{k+1} = h_k/2$. Thus, the sequence used to calculate F_1 will be of the form $x_i = \{-2.0, -1.5, -1.0, \ldots, 1.5, 2.0\}$

Calculate the first 10 approximations in the series and print the final approximation.

Part c.

Plot the successive approximations F_i against iteration number (you will need to define an array to store each approximation). Add a horizontal line for the expected value (`pnorm(2, lower.tail = TRUE)-pnorm(-2, lower.tail = TRUE)`). Set y-axis limits for this plot to be $[0.92, 0.96]$ to best view the progression of approximations.

It is common practice to terminate a sequence of approximations when the difference between successive approximations is less than some small value. What is the difference between your final two approximations (It should be less than 10^{-6})?

Exercise 5

In this exercise, we will use run-time profiling and timing to compare the speed of execution for different functions or calculations. In the general, the algorithm will be

1. Write a loop to execute a large number of iterations. I find 10^6 to be useful; you might start with a smaller number as you develop your code.
2. In this loop, call a function or perform a calculation. You don't need to use or print the results, just assign the result to a local variable.
3. Repeat 1 and 2, but with a different function or formula.
4. Repeat steps 1-3 10 times, saving the time of execution for each pair of the 10 tests. Calculate mean, standard deviation and effect size for the two methods tested.

If you choose R, I've included framework code using `Rprof`; I've included framework code for IML in the SAS template.

Test options

- In homework 3, you were given two formula for the Poisson pmf,

$$\begin{aligned} f(x;\lambda) &= \frac{e^{-\lambda}\lambda^x}{x!} \\ &= exp(-\lambda)(\frac{1}{x!})exp[x \times log(\lambda)] \end{aligned}$$

Compare the computational time of these two formula.

- Create a sequence `x` of numbers -3 to 3 of length 10^6 or so. In the first test, determine the amount of time it takes to compute 10^5 estimates of `norm.pdf` by visiting each element of `x` in a loop. In the second test, simply pass `x` as an argument to `norm.pdf`. Do R or IML optimize vector operations?
- The mathematical statement $\sqrt{x}$ can be coded as either `sqrt(x)` or `x^(1/2)`. Similarly, e^x can be written as $exp(1)^x$ or $exp(x)$. These pairs are mathematically equivalent, but are they computationally equivalent? Write two test loops to compare formula with either $\sqrt{x}$ or e^x of some form (the normal pdf, perhaps).

I find $10^5 - 10^7$ give useful results, and that effect size increases with the number iterations; there is some overhead in the loop itself that contributes relatively less with increasing iterations.

```
Rprof("test1")
## iteration code 1
Rprof(NULL)
Rprof("test2")
## iteration code 2
Rprof(NULL)
#execution time for test 1
summaryRprof("test1")$sampling.time
#execution time for test 2
summaryRprof("test2")$sampling.time
#functions called in test 1
summaryRprof("test1")$by.self
#functions called in test 2
summaryRprof("test2")$by.self
}
```

Exercise 6

Write an improved Poisson pmf function, call this function `smart.pois`, using the same parameters `x` and `lamba` as before, but check x for the following conditions.

1. If x is negative, return a missing value (`NA`, `.`).
2. If x is non-integer, truncate x then proceed.
3. If x is too large for the factorial function, return the smallest possible numeric value for your machine.

What x is too large? You could write a loop (outside your function) to test the return value of `factorial` against `Inf` while calling `factorial` with increasing values. You can then hardcode this value in your function.

You can reuse previously tested code writing this function as a wrapper for a previously written `pois.pmf` and call that function only after testing the for specified conditions.

Test this function by repeating the plots from Homework 4, Ex 6. How is the function different than `dpois`?