Practical 2: inferring Covid fatal incidence

During the Covid pandemic it would have been very helpful to know how the number of new infections each day (incidence) was changing. This quantity is what indicates whether the epidemic is under control at any given point. It is much more useful than 'cases' which are recorded some time after infection and do not have a constant relationship with the number of people actually infected at any time. New infections are impossible to observe directly, but the rate at which new fatal infections are occurring can be inferred after the fact, by careful analysis of the data on daily deaths. This is because there is (and was from early in the epidemic), good information on the distribution of time from infection to death from Covid. Post-hoc estimation of this sort is important for trying to work out when control measures were effective, and which may have been excessive.

In this practical you will write code implementing a very simple simulation based method for inferring fatal incidence rates from Covid deaths in English hospitals. Conceptually the method works like this. We start by assigning each victim a guessed time of infection. We then add a random draw from the infection-to-death distribution to each time of infection to get the implied times of death. Initially the resulting distribution of times of death will poorly match the real distribution. So for each time of infection in turn we randomly propose to move it a few days, only accepting the proposed move if the change improves the simulation's fit to the real death time distribution. This process is simply iterated until we match the death time distribution well.

You will work with the Covid death data in the file `engcov.txt`, to be read in using `read.table`. Column `julian` gives the day of the year (Jan 1 2020 is day 1), and column `nhs` gives the number of Covid deaths in English hospitals that day. For this practical, use only the first 150 rows of data. You will use the log normal distribution (see ?dlnorm) as the distribution of infection-to-death durations. Based on the ISARIC study the appropriate parameters are `meanlog`=3.152 and `sdlog`=0.451 (other studies give very similar results). Let d_i denote the actual number of deaths on day i of the year, and d_i^s denote the number of deaths according to simulation. The goodness of fit of the simulation can be judged by a modified Pearson statistic

$$P = \sum_i (d_i - d_i^s)^2 / \max(1, d_i^s)$$

The basic algorithm is as follows.

1. Find the vector of probabilities of each disease duration $1, 2, 3, \ldots 80$ days according to the given log normal density. Normalise the vector so that the probabilities sum to 1. By randomly sampling integers $1, \ldots 80$ with the given probabilities you will be able to randomly generate days from infection to death in the next parts. `sample` can be used for this (with replacement).
2. Create a vector of death day for each of the $n = 29422$ individual fatalities using `rep`. Draw n (integer) values from the distribution of infection-to-death durations. Subtract these from the individual days of death to get a vector, `t0`, of initial guesses for the days of infection.
3. Repeat the following 2 steps `n.rep` times.
4. Generate n new draws from the infection-to-death distribution and add these to the infection times in `t0` to get simulated death days. Compute P. You can use `tabulate` to go from the vector of death days to the number of deaths per day.
5. Work through each element of `t0` in random order (hint: use `sample` again). For each element propose moving it by -4, -2, -1, 1,2 or 4 days chosen at random (but keep its infection to death duration fixed). If P is reduced, accept the move and update P, otherwise leave `t0` and P unchanged.

Note that when implementing this, you should avoid calling `tabulate` or `sample` within the loop over the elements of `t0`. Doing so is not necessary and will slow your code substantially. It is suggested that you work with arrays that assume that infection and death times may occur anywhere from day 1 to day 310 of the year (this is excessive, but avoids problems with 'simulating beyond the end' of an array). Here is what to produce.

- Given the basic algorithm, write a function `deconv(t,deaths,n.rep=100,bs=FALSE,t0=NULL)` implementing it, where `t` will contain the days of the year on which deaths occurred, `deaths` is the number of deaths occurring each day, `n.rep` is as above, `bs` controls bootstrapping (covered below) and `t0` is an optional `t0` vector - when supplied as NULL the function should generate this internally as in step 2 above. The function should return a list containing at least: `n.rep`-vector `P` containing the history of `P`

after each full `t0` update step; a $310 \times$ `n.rep` matrix `inft`, each column of which contains the number of new infections per day, according to the state of `t0` after each full update; and an n vector `t0` containing the final state of `t0`. This specification must be followed exactly, with names, defaults and return object exactly as specified (the return list can contain extra elements if you like).

It is a good idea if your function produces a plot, after each full `t0` update, showing the current estimated incidence per day, the real deaths per day and the current simulated deaths per day. This is useful for checking. You can improve the convergence rate by initially choosing the steps from $\{-8, -4, -2, -1, 1, 2, 4, 8\}$ then (e.g. after 50 steps) from $\{-4, -2, -1, 1, 2, 4\}$ and then (after 75 steps?) from $\{-2, -1, 1, 2\}$.

- Having estimated the fatal incidence trajectory, it is good to quantify its uncertainty. One way to do this is with an approximate bootstrapping approach applied *once the method has converged*. The simplest method treats the real deaths data as estimates of the expected values of Poisson random variables, and then simulates Poisson data with these expected values, to use in place of the real deaths data, before each iteration step fully updating `t0`. Starting in a converged state is easy to do by supplying the returned `t0` from a converged fit, as the `t0` argument to a new fit. Use `bs=TRUE` to signal your function to do the bootstrap steps.
- Finally you should produce a plot showing your estimated incidence trajectory over time, with an illustration of uncertainty, the raw death data trajectory it corresponds to and a vertical line (see `?abline`) illustrating the first day of UK lockdown (day 84, March 24 2020). It would also be good to illustrate the fit of the simulation model to the real death data.

Your code must only use base R, and must not load any other packages. Your code should read in the data file *without specifying a path*. Use `setwd` to set the path to the file on your machine, but comment this out for submission. When cut and pasted, or `sourced` into R, your code should define `deconv`, then run it with `n.rep` set large enough to converge, then run it again to do the bootstrapping and then produce your final plot.

Code must be well commented and start with a brief statement of the proportion of the work contributed by each group member, and what it consisted of. After that, a concise overview comment should explain to someone who knows nothing about all this, what the code is for, and the basic overview. The function should also start with an overview comment briefly explaining inputs outputs, purpose and approach. Remaining comments should seek to explain the purpose, overall structure and approach of the code. Avoid mechanical description of R code (e.g. `a[j] <- b0 ## b0 is put into element j of a` is worse than pointless commenting).

One piece of work - the text file containing your commented R code - is to be submitted for each group of 3 on Learn by 12:00 18th October 2024. You may be asked to supply an invitation to your github repo, so ensure this is in good order. No extensions are available on this course, because of the frequency with which work has to be submitted. So late work will automatically attract a penalty (of 100% after work has been marked and returned). Technology failures will not be accepted as a reason for lateness (unless it is provably the case that Learn was unavailable for an extended period), so aim to submit ahead of time.

Marking Scheme: Full marks will be obtained for code that:

1. does what it is supposed to do, and has been coded in R approximately as indicated, and *exactly* following the specification of function name, inputs and output.
2. is carefully commented, so that someone reviewing the code can easily tell exactly what it is for, what it is doing and how it is doing it without having read this sheet, or knowing anything else about the code. Note that *easily tell* implies that the comments must also be as clear and *concise* as possible. You should assume that the reader knows basic R, but not that they know exactly what every function in R does.
3. is well structured and laid out, so that the code itself, and its underlying logic, are easy to follow.
4. is reasonably efficient (not much more than 3 minutes to run everything)
5. was prepared collaboratively using git and github in a group of 3.
6. contains no evidence of having been copied, in whole or in part, from anyone else (AI counts as anyone else for this purpose), other students on this course, students at other universities (there are now tools to help detect this, including internationally), online sources, ChatGPT etc.
7. includes the comment stating team member contributions.

Individual marks may be adjusted within groups if contributions are widely different.