Environmentally Transmitted Pathogens

Models

Julien Arino

January 2023

Fitting

- General principles
- A simple example – The logistic function
- The issue of parameter identifiability

Fitting

- General principles
- A simple example – The logistic function
- The issue of parameter identifiability

Fitting a model to parameters

Very simplified version of what is done in practice
There are way more elaborate methods. See, e.g.,

- Roda. Bayesian inference for dynamical systems
- Portet. A primer on model selection using the Akaike Information Criterion

Parameter fitting problem

Assume given data points (t_i, y_i), $i = 1, \ldots, N$, with $y_i \in \mathbb{R}^n$ and $t_i \in \mathcal{I} \subset \mathbb{R}$, where $\mathcal{I}$ is some interval of time

Assume model parameters are in a set $\mathcal{P}$

Solution to the ODE is $x(t, p)$ for $t \in \mathcal{I}$ and a given $p \in \mathcal{P}$ (we emphasise the dependent of the solution on the chosen point in parameter space)

Parameter fitting problem

Find $p \in \mathcal{P}$ such that the solution $x(t, p)$ "most closely matches" the data points

Could also be another type of system (discrete-time, continuous-time Markov chain – in which case we would likely use mean solution over a number of realisations, etc.), but in any event, a time series somewhat comparable with the data

Mathematical formulation

We want to minimise the error function

$$E(p) = \sum_{i=1}^{N} \|h(x(t_i)) - y_i\| \tag{1}$$

- $h : \mathbb{R}^n \to \mathbb{R}^n$ is an **observation function**, which selects the relevant part of the solution to match to the data
- The norm is usually the Euclidean norm, but could be different depending on the problem at hand
- Given a parametre p in an (admissible part) of parameter space, compute the solution to the ODE, then deduce $E(p)$
- Use a minimisation algorithm to find a minimum of $E(p)$ while varying p

What are y_i and $h(x(t_i))$ here?

Infectious disease data is ofen in terms of *incidence* (number of new cases per unit time) rather than *prevalence* (number of cases currently present in the population)

In, say, an SIR model with mass action incidence, incidence is βSI, so, using the Euclidean norm

$$E(p) = \sum_{i=1}^{N} (\beta S(t_i) I(t_i) - y_i)^2 \tag{2}$$

where $S(t_i)$ and $I(t_i)$ are the values of the numerical solution to the ODE at the times t_i in the data

Fitting

- General principles
- A simple example – The logistic function
- The issue of parameter identifiability

To illustrate the method, let us use a simple case and fit the logistic equation $N' = rN(1 - N/K)$ to some population data

We use El Salvador, whose population seems to be experiencing growth slowdown. We get population data from the World Bank

We seek values of r and K that minimise $E(p)$ given by (1) with norm the Euclidean norm

The right hand side

This one is quite simple, of course...

```
RHS_logistic <- function(t, x, p) {
    with(as.list(c(x,p)), {
    dN <- r*N*(1-N/K)
    return(list(dN))
    })
}
```

Getting the population data

```
library(wbstats)
get_pop_data <- function(CTRY) {
    pop = wb_data("SP.POP.TOTL", country = CTRY,
                  mrv = 100, return_wide = FALSE)
    pop = pop[,c("date", "value")]
    pop = pop[order(pop$date),]
    pop$date = as.numeric(pop$date)
    pop$value = as.numeric(pop$value)
    return(pop)
}
```

which we call later using

```
pop = get_pop_data("El␣Salvador")
```

Note that if you do not set `return_wide=TRUE`, the column with the return value is named like the indicator (`SP.POP.TOTL` here), otherwise it is `value`

The error function

This is the function that does most of the work

- Takes as input the parameters that vary and any other required parameters
- Computes the solution of the ODE
  - Capitalise on the fact that `ode` returns the solution at `times` that you can specify!
  - If your data is at times, say, $t = 1, 2, 5, 10$, then you can pass `times=c(1,2,5,10)` and get the solution at these times, making the next step easy
- Compute the error

Actually, this does not do most of the work but this is definitely where *you* need to do most of the work

```r
error_fit <- function(p_vary,
                      params,
                      data,
                      method = "rk4") {
    # Anything that changes during optimisation is set here
    params$r = as.numeric(p_vary["r"])
    params$K = as.numeric(p_vary["K"])
    # Set the initial condition
    N0 = data$value[1]
    IC = c(N = N0)
    # Compute the solution
    sol = ode(y = IC, times = data$date, func = RHS_logistic,
              parms = params, method = method)
```

One little trick

If a parameter value or a solution is "not acceptable", one easy way to deal with this is to return an error value of `Inf` (i.e., ∞) and immediately exit the error function

```
if (sol[dim(sol)[1],"time"] < data$date[length(data$date)]) {
    return(Inf)
}
```

would be triggered if, for instance, the numerical integration finished early (because solutions explode, e.g.)

Useful also, e.g., if you want to exclude regions in parameter space. Say you want to find parameters s.t. $\mathcal{R}_0 \leq 10$, then you could return an `Inf` error whenever parameters are s.t. $\mathcal{R}_0 > 10$

Computing the actual error

```
1      diff_values = data$value - sol[,"N"]
2      diff_values_squared = diff_values^2
3      error = sum(diff_values_squared)
4      return(error)
5  }  # END error_incidence
```

Line 1: here, $h(x) = x$, we can simply use $N(t)$ as the observed variable. So we compute the $y_i - N(t_i)$, since we have set `times=data$date` and thus have matching time points

Line 2: square the values (recall that by default, `R` does Hadamard-type operations, so here, squares each entry in the vector), giving a vector with entries $(y_i - N(t_i))^2$

Line 3: sum the elements of the vector, i.e., obtain $\sum_i (y_i - N(t_i))^2$

Set up a last few things

Back in the main code, set values for the parameters, although they will be changed by the optimiser

```
params = list()
params$r = 1
params$K = max(pop$value)
```

Now let an optimiser do the actual work

Here, let us use a genetic algorithm

```
library(GA)
GA = ga(
  type = "real-valued",
  fitness = function(x)
    -error_fit(p_vary = c(r = x[1], K = x[2]),
               params = params,
               data = pop,
               method = "rk4"),
  parallel = TRUE,
  lower = c(0.1, 1000000),
  upper = c(10, 10000000),
  optim = TRUE,
  optimArgs = list(method = "CG"),
  suggestions = c(1, params$K),
  popSize = 500,
  maxiter = 200
)
```

Explaining the algorithm and the call to `GA`

A **genetic algorithm** (GA) mimics evolution selecting for increased fitness.

- A **gene** is a point $p^\star \in \mathcal{P}$ in parameter space, so, here, a given value $(r^\star, K^\star)$ of (r, K)
- The **fitness** of the gene is the value of the function to optimise when evaluated at $p^\star$, i.e., here, $E(p^\star)$ (i.e., `error_fit`)
- A GA "wants to" maximise fitness, so we actually use $-E(p)$

Setting up the gene pool

- Start with a randomly selected population of `popSize` genes (500 here)
- Within these `popSize` genes, one (could be more) is a `suggestion`. Here, I have taken `r=1` and `K=params$K=max(pop$value)`
- Genes other than the suggested ones are selected at random (potentially following some distribution, but by default and here, uniformly) between `lower` and `upper`
- (The order of elements of the gene is important in some places, including `suggestions`, `upper` and `lower`)

At each iteration, up to a maximum `maxiter` (200 here)

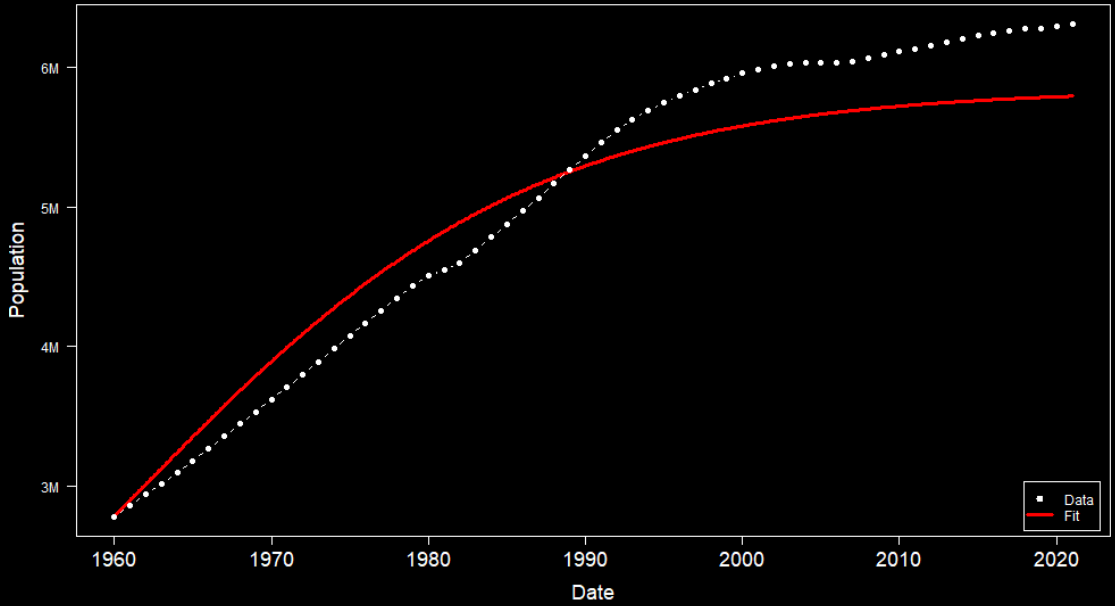
- Compute the fitness of all `popSize` genes
- Retain the genes with highest fitness
- Throw in new genes, some at random like before but others using "constrained randomness", i.e., using analogues of genetic operations such as mutations, crossovers, etc.
- Once the next generation is ready, i.e., there are `popSize` genes, restart

It is possible to specify how much of the existing gene pool to keep, etc.

Two interesting options

1. `parallel=TRUE` (needs libraries `parallel` and `doParallel`) parallelises the code. Each function (fitness) evaluation is completely independent from others, so this algorithm parallelises very nicely, leading to potentially consequential speedups
2. `optim=TRUE` interrupts the GA execution (including parallel component if used) to perform a step of deterministic gradient descent search close to the best value found so far, in case there is no change in best value for a few generation. This allows to potentially fine tune a stochastically found optimum

Note – If you want to limit the number of threads used (to avoid completely bogging down your computer), you can specify a number of threads to use, instead, e.g., `parallel=10`. Currently, you also *must* specify `parallel=124` if you have a CPU with more than 124 threads


6M
5M
4M
3M
Population
1960
1970
1980
1990
2000
2010
2020
Date
Data
Fit

Varying also N_0

Previous graph fits precisely the first data point. We can want to also fit N_0

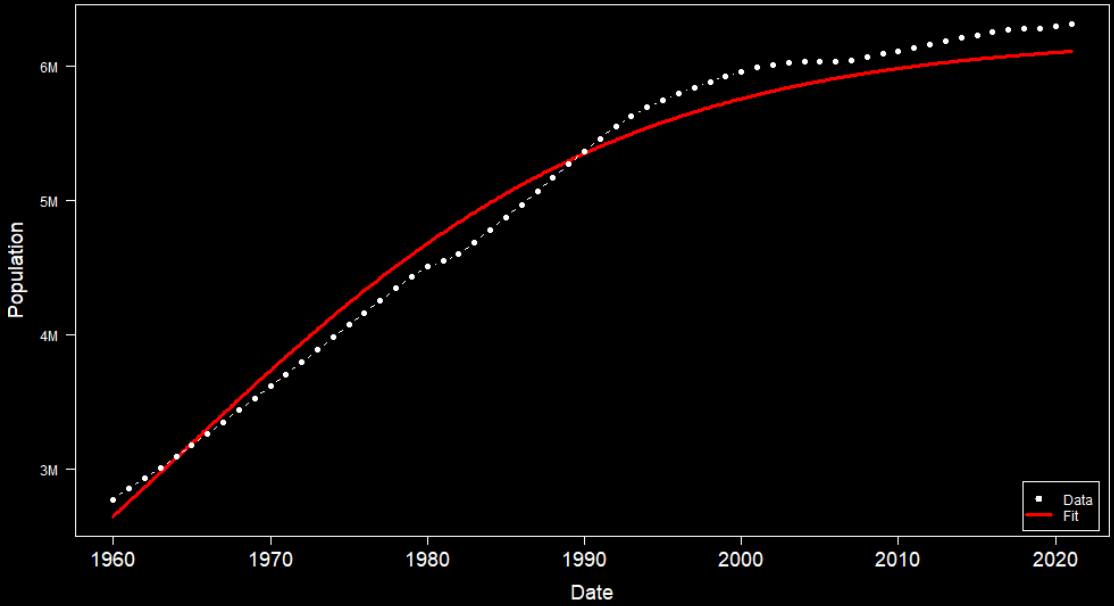
In the function `error_incidence`, add

```
params$N0 = as.numeric(p_vary["N0"])
```

then in the call to `error_incidence` in ga,

```
fitness = function(x)
    -error_incidence(p_vary = c(r = x[1], K = x[2], N0 = x[3]),
                     params = params,
                     data = pop,
                     method = "rk4"),
```

We get a smaller error and something like on the next page


6M
5M
4M
3M
Population
1960
1970
1980
1990
2000
2010
2020
Date
Data
Fit

Fitting

- General principles
- A simple example – The logistic function
- The issue of parameter identifiability

See, e.g., Roda *et al*: Why is it difficult to accurately predict the COVID-19 epidemic?

We seek to minimise the (error) function

$$E(p) = \sum_{i=1}^{N} \|h(x(t_i)) - y_i\| \tag{1}$$

- It is possible (extremely likely with complex models) that several values of p minimise $E(p)$
- It is also possible that the value(s) found for p are only *local* minima
- These problems are linked to the so-called **identifiability** problem