E2M2: Model evaluation and comparison: an example outbreak

Jessica Metcalf; cmetcalf@princeton.edu

Simulating an outbreak

A directly transmitted infection, with a short generation time emerges in a population where everyone is susceptible. Individuals can be in 3 **states**, i.e., susceptible ('S'), infected ('I') and recovered ('R'). We can write the function that defines this process, assuming that the total population size is constant, which means we don't need to keep track of the ('R') compartment. The function is:

```
sir <- function(t,y,parms){
  with(c(as.list(y),parms),{
    dSdt <- -beta*S*I/N
    dIdt <- beta*S*I/N - gamma*I
    list(c(dSdt,dIdt))
  })
}
```

We define a starting population with 1000 susceptible individuals and 2 infected individuals; and a list of parameters that define the processes, β, the transmission rate, and γ, the recovery rate (defined as 1/infectious period). We also assume that the population size, N, is the same as the starting number of susceptible individuals for simplicity:

```
pop.SI <- c(S = 1000,I = 2)       # Starting population structure
values <- c(beta = 1.1,           # Transmission coefficient
            gamma = 1/12,         # Recovery rate
            N=1000)               # Population size (constant)
```

As you know, we can calculate the value of the basic reproduction number $R_0 = \beta/\gamma$, as follows:

```
R0 <- values["beta"]/values["gamma"] # value of R0
R0
```

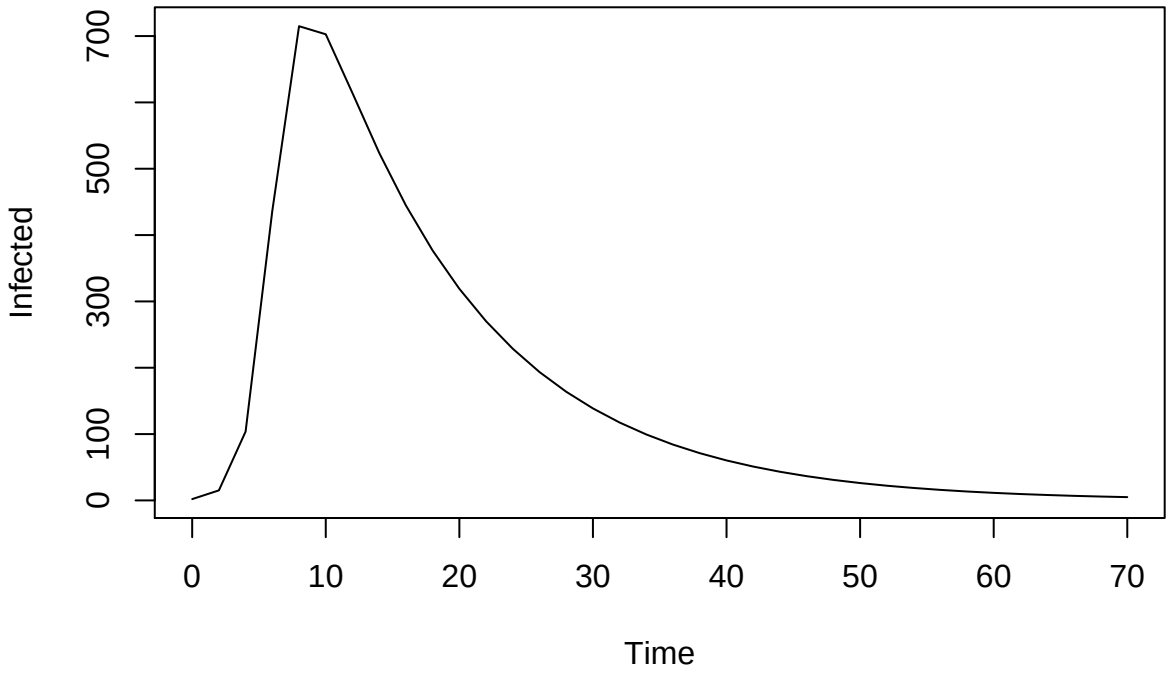
```
## beta
## 13.2
```

As in the structured population exercise example, we can use the function lsoda, here taking as our time-unit 1 day, setting a small time-step (2 days), and running it out across 71 days:

```
library(deSolve)

times <- seq(0,71,by=2)

ts.sir <- data.frame(lsoda(
  y = pop.SI,                   # Initial conditions for population
  times = times,    # Timepoints for evaluation
  func = sir,                   # Function to evaluate
  parms = values                # Vector of parameters
  ))
```

and then we can plot this:

```
plot(ts.sir[,"time"],ts.sir[,"I"], xlab="Time", ylab="Infected", type="l")
```



The epidemic burns itself out (i.e., at the end, the number of infected individuals is zero).

Simulating the observation process

Usually during an outbreak, we only have data on the number of cases - i.e., we have no data on numbers of susceptible or recovered individuals. Furthermore, the probability of observing a case is often less than 1, i.e., it is the outcome of a binomial process. We can simulate something like this by defining the number of cases observed as:

```
cases <- rbinom(nrow(ts.sir),floor(ts.sir[,"I"]),0.7)
```

i.e., we are assuming that 0.7 of the 'true' number of cases are observed in each time-step. Note that the function 'rbinom' requires integers, but our function 'ts.sir' returns continuous variables (i.e., there can be 10.7 individuals). To get around this, we are simply using the function 'floor'.

An underfitted model

An underfitted model might be one that quite simply fits an overall mean across the whole time-series. This would mean that there is **one** parameter in the model. We could simply calculate this mean (using the R function 'mean) or we can use the 'lm' machinery to calculate it, e.g., via:

```
underfit <- lm(cases ~ 1)
```

An overfitted model

An overfitted model might be one that fits a different number of cases for every two subsequent observations (so every four days), i.e.,

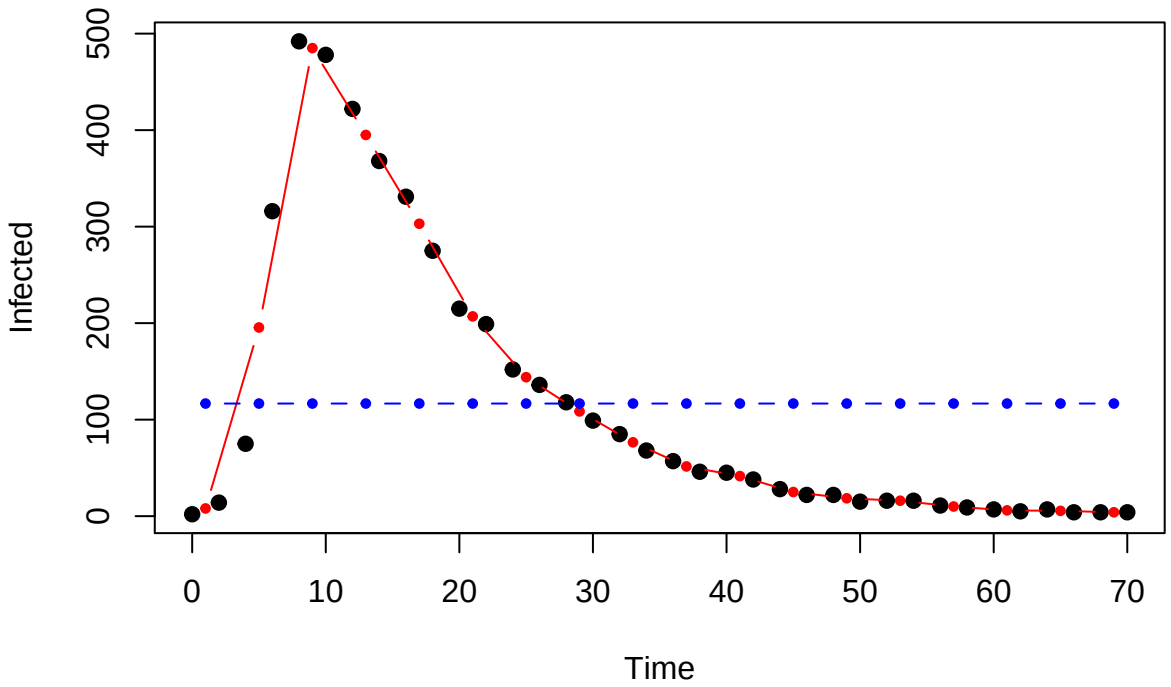
```
## pull out every 2nd measurement
##  - the 'seq' command creates an index 1,3,5, ...
new.time <- ts.sir[seq(1,nrow(ts.sir), by=2),"time"]
## we repeat them twice to have something of same length as the data
## and add 1/2 of the difference between two time-steps, since our average of cases should also reflects
new.time <- rep(new.time,each=2)+0.5*diff(times)[1]

## we fit this using a linear model
overfit <- lm(cases ~ as.factor(new.time)-1)
```

By including "-1", we ensure that R does not fit an overall intercept, so we will have a coefficient estimated for every time-point in 'new.time'. We've done this here just to make things easier for the plotting (below)

Comparison

We can plot both the cases observed, and the under- and over-fitted models for comparison:

```
plot(ts.sir[,"time"],cases, pch=19, xlab="Time", ylab="Infected")
points(unique(new.time),overfit$coeff, type="b", col=2,lty=1,pch=19, cex=0.6)
points(unique(new.time),rep(underfit$coeff,length(unique(new.time))), col=4, pch=19, cex=0.6, type="b")
```



The under-fitted model (blue) is very bad at telling us what is happening both at the epidemic peak and after the epidemic has gone away (and is likely to violate the assumptions of regression). The over-fitted model tells us quite accurately what happens in this outbreak. The problem with the overfitted model is that it is not clear what it adds to our understanding beyond the data - and its very hard to see what you might do with this model in any new circumstances.

Fitting a mechanistic model

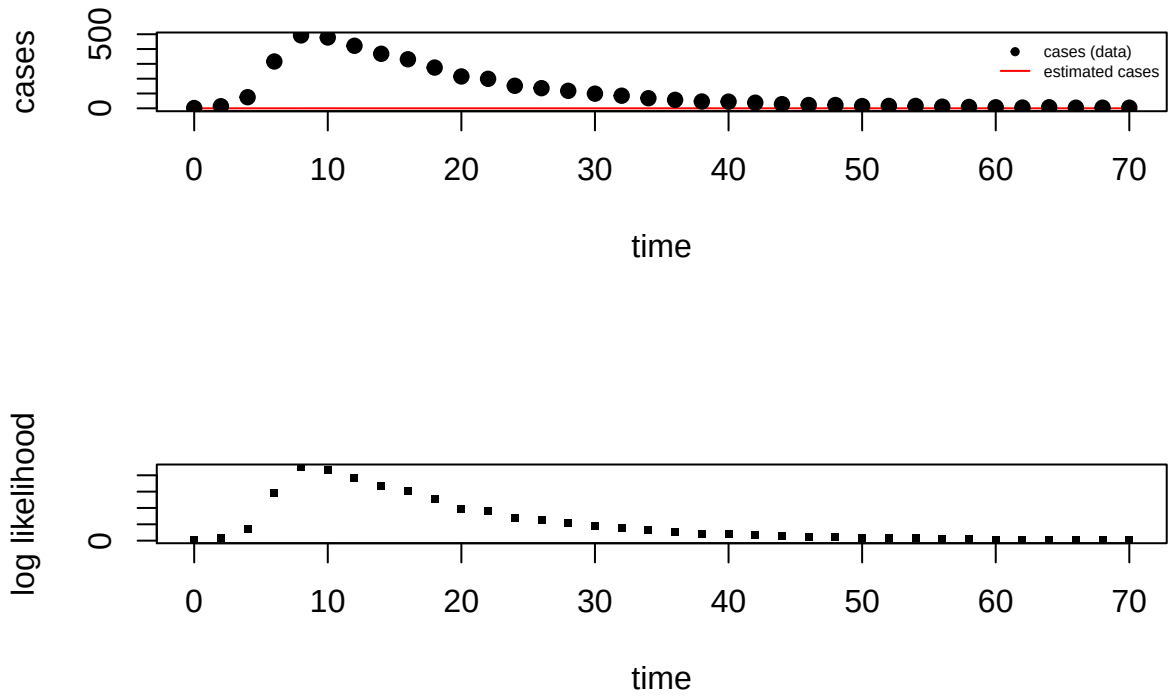
We can also fit a mechanistic model to this same data - we have an advantage here in that we know what the 'true' model is, captured by the function 'sir' above. Of course, this will not be the case in many

realistic situations. We can define a likelihood function for the data, by defining a function that simulates the time-series using our chosen model of the dynamics (here, captured by the function 'sir') and the chosen parameters, then evaluate this against cases, using a binomial likelihood to take into account the observation process:

```
like.cases <- function(par,cases,pop.SI=c(S = 1000,I = 2),do.plot=FALSE){

      #plug 'params' into a named vector (which lsoda needs)
      values <- c(beta = par[1],          # Transmission coefficient
                  gamma = par[2],         # Recovery rate
                  N=as.numeric(pop.SI["S"])) # Assume S=N

      #run our model with these parameters
      ts.sir <- data.frame(lsoda(
          y = pop.SI,                   # Initial conditions for population
          times = times,                # Timepoints for evaluation
          func = sir,                   # Function to evaluate
          parms = values                # Vector of parameters
       ))

      #put obs. prob. on logit scale, to avoid issues where >1 or <0
      rho <- exp(par[3])/(1+exp(par[3]))

     #get the likelihood
     like <- dbinom(cases,floor(ts.sir[,"I"]),
                    rho, ## put obs on logit scale
                    log=TRUE)

     #plot if wanted (this gives us a way to understand problems)
     if (do.plot){
     par(mfrow=c(2,1))    # make 2 rows and 1 column
     plot(ts.sir[,"time"],ts.sir[,"I"]*rho,type="l",col=2,
          ylim=range(c(ts.sir[,"I"]*rho,cases)),xlab="time", ylab="cases")
     legend("topright",legend=c("cases (data)", "estimated cases"),
            pch=c(19,NA),lty=c(NA,1),col=c(1,2), bty="n", cex=0.5)
     points(ts.sir[,"time"],cases,pch=19,col=1)
     plot(ts.sir[,"time"],-like,xlab="time", ylab="log likelihood",
          pch=15, cex=0.5)
     }

     # some edits to make a finite number come out! If more cases are reported
     #  than are observed, the likelihood is -infinity, and we can't add like
     like[like==-Inf] <- -exp(200)
     like[like==Inf] <- exp(200)

     return(-sum(like))

}
```

Note that here, we are assuming that we know the initial conditions (S, I, and N) but we could also be estimating them. We put the likelihood on a log scale, since then we can add up the likelihoods; and the function returns the negative because it is easier to minimize than maximize - we want to maximize the likelihood, of course! To see what this function does, we can write:

```
test1 <- like.cases(par=c(1.1,1/12,-10),cases=cases,
                    pop.SI=c(S = 1000,I = 2), do.plot=TRUE)
```



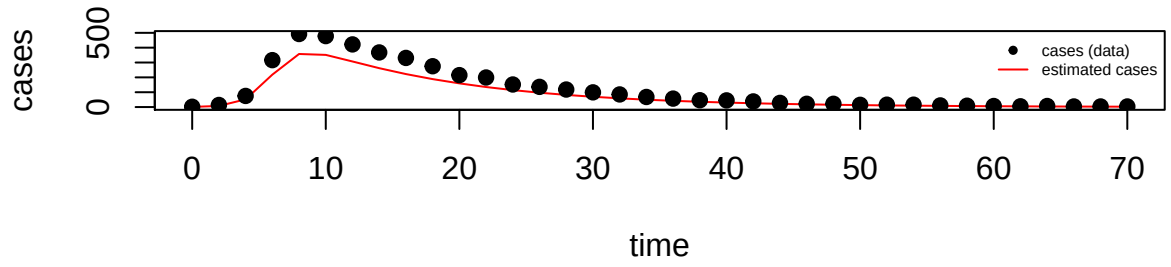

```
test1
```

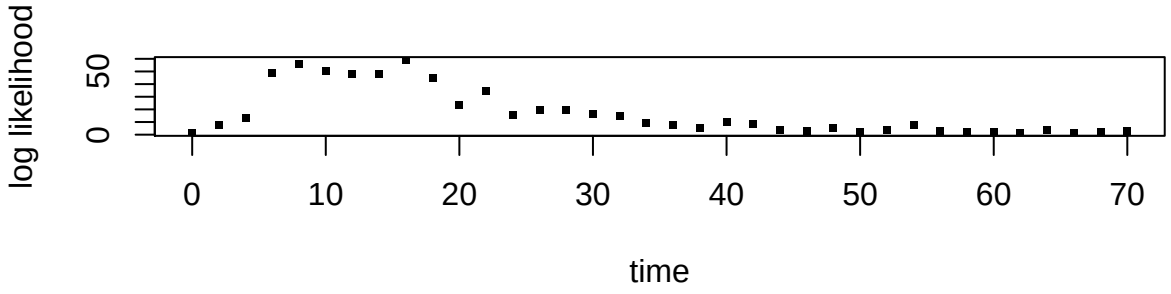
```
## [1] 38481.52
```

and we use the 'do.plot' argument, setting it to 'TRUE' to show us both the predicted trajectory (top plot) and the likelihood obtained (lower plot) at each point. This provides us with a way of figuring out if something is wrong with our function! Note that here, for times where there is no point on the likelihood plot, then the likelihood is not finite (since we make the plot before we make the correction). This happens if we are observing MORE cases than we are predicting, because there is no probability that allows this!

The value returned is the overall likelihood of observing this data if $\beta = 1.1$, $\gamma = 1/12$ and the observation probability is exp(-10)/(1+exp(-10)) (remembering that the observation probability is on the logit scale to constrain it between 0 and 1). Note that you will not get exactly the same value that is printed here since cases are generated stochastically! If we change the observation probability, the likelihood changes...; likewise if we change other parameters.

```
test2 <- like.cases(par=c(1.1,1/12,0),cases=cases,
                    pop.SI=c(S = 1000,I = 2), do.plot=TRUE)
```

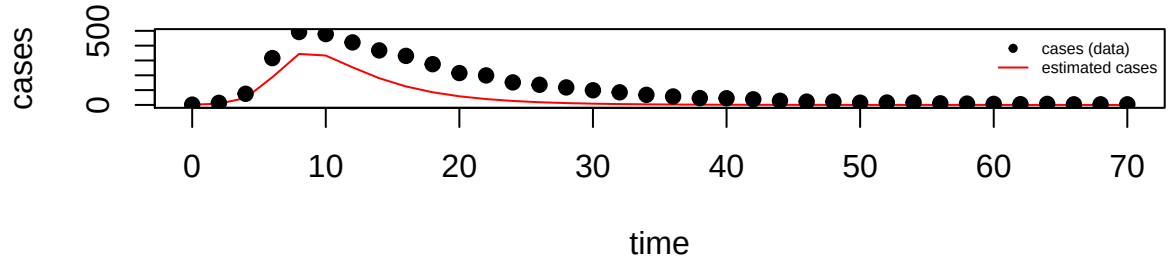


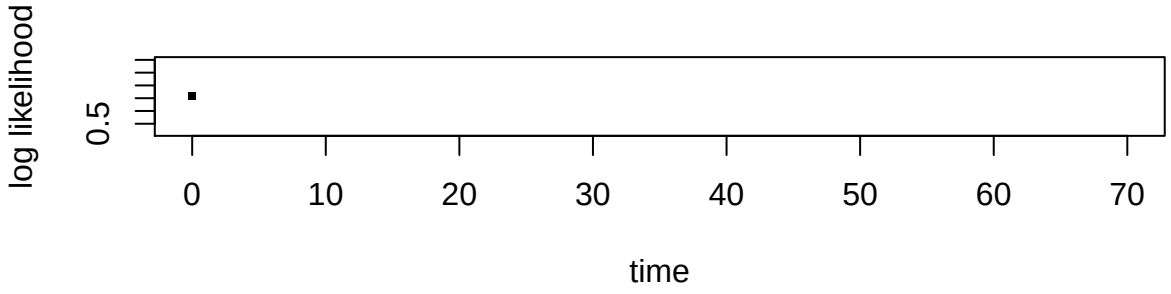



```
test2
```

```
## [1] 606.5676
```

```
test3 <- like.cases(par=c(1.1,1/5,log(0.7/(1-0.7))),cases=cases,
                    pop.SI=c(S = 1000,I = 2), do.plot=TRUE)
```






```
test3
```

```
## [1] 2.529091e+88
```

and so on. As above, you will be getting different values from mine, because 'cases' are generated stochastically.

Clearly, we would like to explore all of parameter space for our three focal parameters to find the most likely, or 'best' model. One option is to use 'optim' in R to do this, giving the function starting parameters that are reasonably close to the 'true' parameters, and setting our chosen method to 'Nelder-Mead':

```
tmp <- optim(par=c(1.0,0.15,log(0.6/(1-0.6))),
             fn=like.cases,cases=cases,do.plot=FALSE,
                      pop.SI=c(S = 1000,I = 2),
             method="Nelder-Mead")
## repeat starting at the output conditions to give more search space
tmp <- optim(par=tmp$par,
             fn=like.cases,cases=cases,do.plot=FALSE,
                      pop.SI=c(S = 1000,I = 2),
             method="Nelder-Mead")

tmp
```

```
## $par
## [1] 1.12745343 0.08070124 0.76494975
##
## $value
## [1] 97.31107
##
## $counts
## function gradient
##      166       NA
##
## $convergence
## [1] 0
##
## $message
## NULL
```

It helps to run it twice to make sure it evaluates a sufficient range. Use '?optim' to understand the output of this model. The key points are that 'par' is the estimated parameters at the function minimum, and 'value' is the value of the function that has been minimized (so in our case, the minimum value of the negative loglikelihood that optim could find). Compare the parameters returned by optim (in 'par') with the ones we originally used to simulate the data:

```
values[1:2]   #This is what we set in the 1st section for beta, gamma
```

```
##       beta      gamma
## 1.10000000 0.08333333
```

```
tmp$par[1:2]  #This is what optim thinks the parameters should be
```

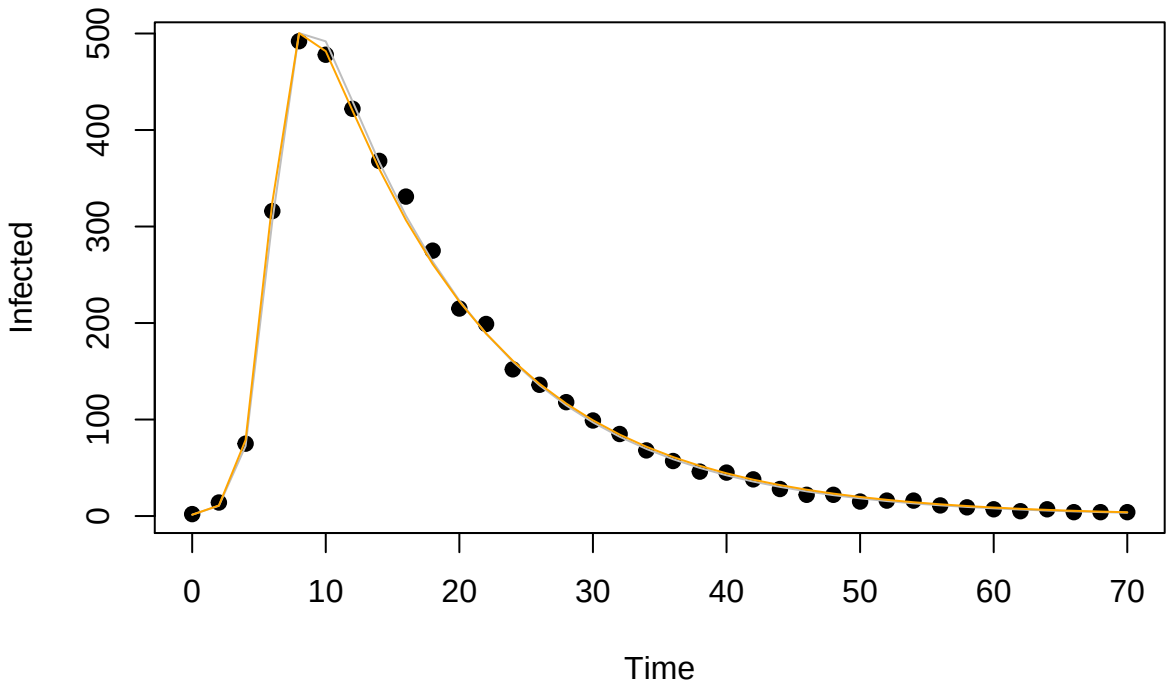
```
## [1] 1.12745343 0.08070124
```

```
#For the observation probability, which we set to 0.7,
exp(tmp$par[3])/(1+exp(tmp$par[3]))  #This is what optim thinks it is
```

```
## [1] 0.6824274
```

It might not be perfect, because the case data is stochastic, but, if we have adequately searched parameter space, it should be close! We can then run the function with these estimated parameters, and plot out results to evaluate the projections of this model (in orange) to the observed (black points), to see how well it does, and also compare it to the 'true' values (i.e., the original model), reduced by the 'true' level of reporting (grey lines):

```
pred.ts.sir <- data.frame(lsoda(
  y = pop.SI,                   # Initial conditions for population
  times = times,    # Timepoints for evaluation
  func = sir,                   # Function to evaluate
  parms = c(beta=tmp$par[1],
            gamma=tmp$par[2],N=1000)                # Vector of parameters
  ))

plot(ts.sir[,"time"],cases, pch=19, xlab="Time", ylab="Infected")
points(ts.sir[,"time"],ts.sir[,"I"]*0.7, type="l", col="grey")
points(pred.ts.sir[,"time"],pred.ts.sir[,"I"]*
         exp(tmp$par[3])/(1+exp(tmp$par[3])),  type="l",col="orange")
```



We can also extract the AIC from this model using the formal definition (2K-2*log(L)) and compare it to the ones obtained from the linear regressions using a nicely built in R function that works with regression models:

```
#underfit
AIC(underfit)
```

```
## [1] 464.6064
```

```
#overfit
AIC(overfit)
```

```
## [1] 385.6596
```

```
#mech fit which has 2 parameters
#remember that our function is designed to minimize! so do subtraction
2*2-2*(-tmp$value)
```

```
## [1] 198.6221
```

Note that instead of using the function 'AIC' you could also calculate it directly using the formula; the function 'logLik' also provides a way of extracting the log likelihood from linear models. Remembering that we want the model with the smallest AIC, the mechanistic model is a clear winner.

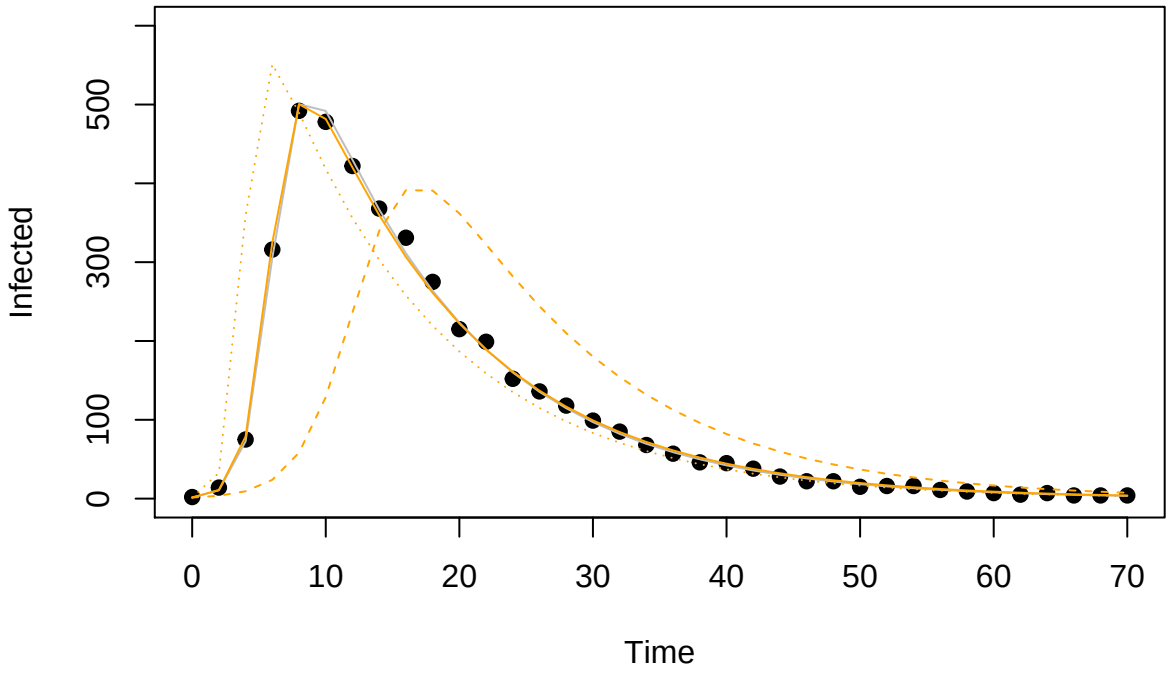
Sensitivity analysis

How much does the outcome hinge on our estimate of a particular value, say β? We can evalute this by comparing the output given values of β:

```
#reduce magnitude by 50% (x by 0.5)
pred.ts.sir.0.5 <- data.frame(lsoda(y = pop.SI,
  times = times,
  func = sir,
  parms = c(beta=0.5*tmp$par[1],gamma=tmp$par[2],N=1000)))

#reduce magnitude by 50% (x by 1.5)
pred.ts.sir.1.5 <- data.frame(lsoda(y = pop.SI,
  times = times,
  func = sir,
  parms = c(beta=1.5*tmp$par[1],gamma=tmp$par[2],N=1000)))
```

and plot these out for comparison (dashed line is reduction of β, dotted line is increase). Note that we could also evaluate their impact on the likelihood, etc.

```
plot(ts.sir[,"time"],cases, pch=19, xlab="Time", ylab="Infected", ylim=c(0,600))
points(ts.sir[,"time"],ts.sir[,"I"]*0.7, type="l", col="grey")
points(pred.ts.sir[,"time"],pred.ts.sir[,"I"]*
         exp(tmp$par[3])/(1+exp(tmp$par[3])),  type="l",col="orange")
points(pred.ts.sir.0.5[,"time"],pred.ts.sir.0.5[,"I"]*
         exp(tmp$par[3])/(1+exp(tmp$par[3])),  type="l",col="orange", lty=2)
points(pred.ts.sir.1.5[,"time"],pred.ts.sir.1.5[,"I"]*
         exp(tmp$par[3])/(1+exp(tmp$par[3])),  type="l",
        col="orange", lty=3)
```



By modifying each of the parameters in turn, we can identify parameters to which output is sensitive, which in turn informs us as to which interventions might be most effective (e.g., here, is it better to reduce transmission, β, or the generation time, γ given the range of what is logistically plausible?). Approaches to completely explore the range of parameter space include Latin Hypercube Sampling, for which there is a package in R

(called 'lhs') for those of you who are interested.

We can also see how robust our model predictions are, the implications of uncertainty in our parameters (if we're not sure whether β is 1.1 or 1.3, how much will this change our predictions?). In more complicated models, it may make sense to drop some parameters (also called 'model simplification') and we can also explore associations between parameters (e.g., find pairs that do similar things). All of this falls within the remit of **sensitivity analysis**.