Forecasting Frameworks

Dexter Barrows
March 18, 2016

1 Data Setup

This section will focus on taking the stochastic SIR model from the previous section, truncating the synthetic data output from realizations of that model, and seeing how well IF2 and HMCMC can reconstruct out-of-sample forecasts.

An example of a simulated system with truncated data can be seen in Figure [1] below.

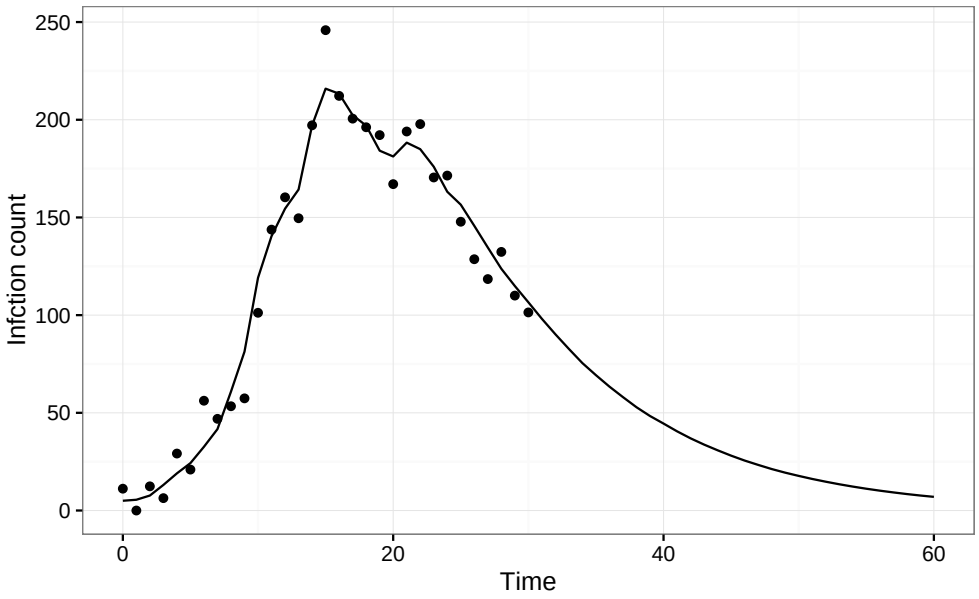

Figure 1: Infection count data truncated at $T = 30$. The solid line shows the true underlying system states, and the dots show those states with added observation noise. Parameters used were $R_0 = 3.0$, $r = 0.1$, $\eta = .05$, $\sigma_{proc} = 0.5$, and additive observation noise was drawn from $\mathcal{N}(0, 10)$.

In essence we want to be able to give either IF2 of HMCMC only the data points and have it reconstruct the entirety of the true system states.

2 IF2

For IF2, we will take advantage of the fact that the particle filter will produce state estimates for every datum in the time series given to it, as well as producing parameter maximum likelihood point estimates. Both of these sources of information will be used to produce forecasts by parametric bootstrapping using the final parameter estimates from the particle swarm after the last IF2 pass, then using the newly generated parameter sets along with the system state point estimates from the first fitting to simulate the systems forward into he future.

We will truncate the data at half the original time series length (to $T = 30$), and fit the model as previously described.

First, we can see the state estimates for each time point produced by the last IF2 pass in Figure [2] below.

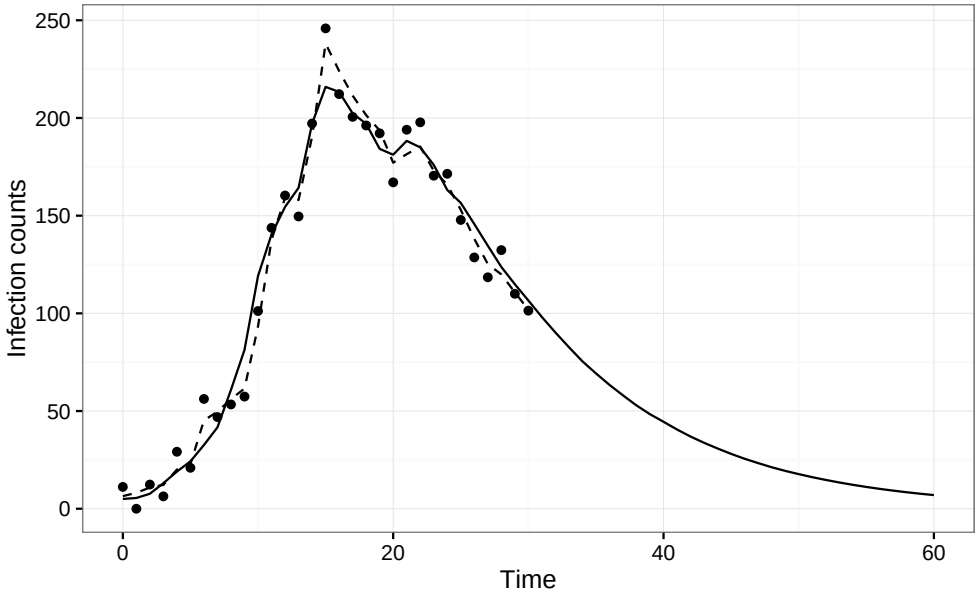

Figure 2: Infection count data truncated at $T = 30$ from Figure [1] above. The dashed line shows IF2's attempt to reconstruct the true underlying state from the observed data points.

Recall that IF2 is not trying to generate parameter estimation densities, but rather produce a point estimate. Since we wish to determine the approximate distribution of each of the parameters in addition to the point estimate, we must turn to another method, parametric bootstrapping.

2.1 Parametric Bootstrapping

The goal of the parametric bootstrap is use an initial density sample θ^* to generate further samples $\theta_1, \theta_2, ..., \theta_M$. It works by using θ to generate artificial data sets $D_1, D_2, ..., D_M$ to which we can refit our model of interest and generate new parameter sets.

[I'm still trying to dig up a good paper that talks about applicability to dynamical systems, there will be a paragraph here about it.]

An algorithm for parametric bootstrapping using IF2 and our stochastic SIR model is shown in Algorithm [1].

Algorithm 1: Parametric Bootstrap

Input : Forward simulator $S(\theta)$, data set D

/* Initial fit */

$\theta^* \leftarrow IF2(D)$

/* Generate artificial data sets */

for $i = 1 : M$ **do**

  $D_i \leftarrow S(\theta^*)$

/* Fit to new data sets */

for $i = 1 : M$ **do**

  $\theta_i \leftarrow IF2(D_i)$

Output: Distribution samples $\theta_1, \theta_2, ..., \theta_M$

2.2 IF2 Forecasts

Using the parameter sets $\theta_1, \theta_2, ..., \theta_M$ and the point estimate of the state provided by the initial IF2 fit, we can use a normal bootstrap to produce estimates of the future state. A plot showing a projection of the data from the previous plots can be seen in Figure [3].

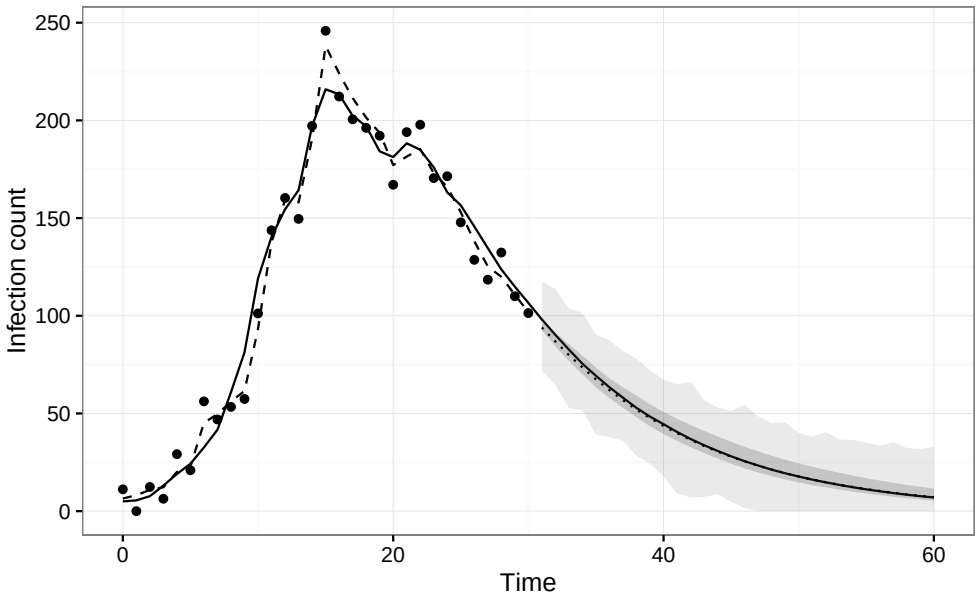

Figure 3: Forecast produced by the IF2 / parametric bootstrapping framework. The dotted line shows the mean estimate of the forecasts, the dark grey ribbon shows the centre 95th quantile of the true state estimates, and the lighter grey ribbon shows the centre 95th quantile of the true state estimates with added observation noise drawn from $\mathcal{N}(0, \sigma)$.

We can define a metric to gauge forecast effectiveness by calculating the SSE and dividing that value by the number of values predicted to get the average squared error per point. For the data in Figure [3] the value was $\overline{SSE} = 1.67$.

3 HMCMC

For HMCMC we can use a simpler bootstrapping approach. We do not get state estimates directly from the RStan fitting due to the way we implemented the model, but we can construct them using the process noise latent variables. Once we've done this we can forward simulate the system from the state estimate into the future.

As before we fit the stochastic SIR model to the partial data, but now perform bootstrapping as described above, and obtain the plot in Figure [4].

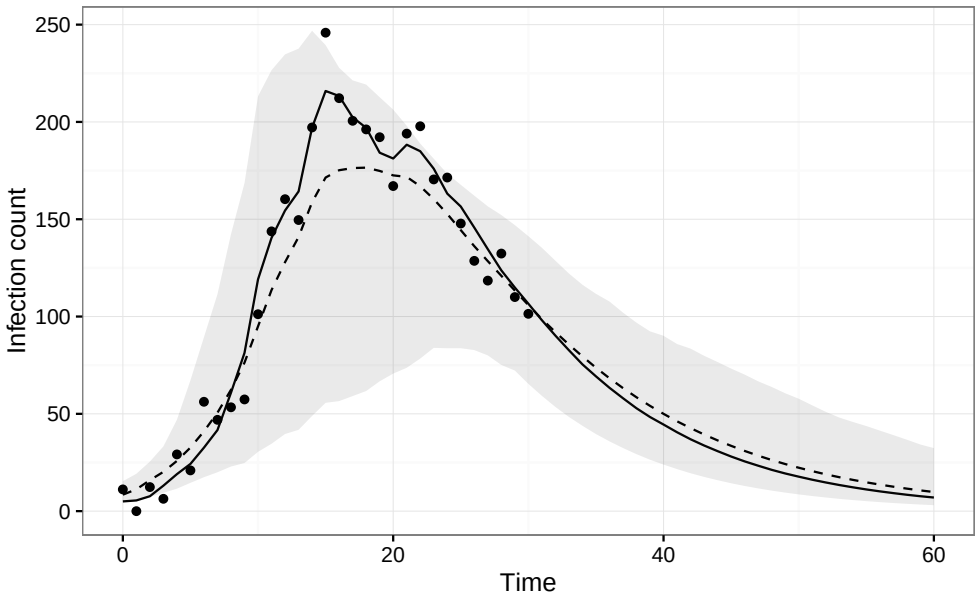

Figure 4: Forecast produced by the HMCMC / bootstrapping framework with $M = 200$ trajectories. The dotted line shows the mean estimate of the forecasts, and the grey ribbon shows the centre 95th quantile.

And as before we can evaluate the averaged SSE of the forecast for the data shown, giving $\overline{SSE} = 20.27$.

4 Truncation vs. Error

Of course the above mini-comparison only shows one truncation value for one trajectory. Really, we need to know how each method performs on average given different trajectories and truncation amounts. In effect we wish to "starve" each method of data and see how poor the estimates become with each successive data point loss.

Using each method, we can fit the stochastic SIR model to successively smaller time series to see the effect of truncation on forecast averaged SSE. This was performed with 10 new trajectories drawn for each of the desired lengths. The results are shown in Figure [5].

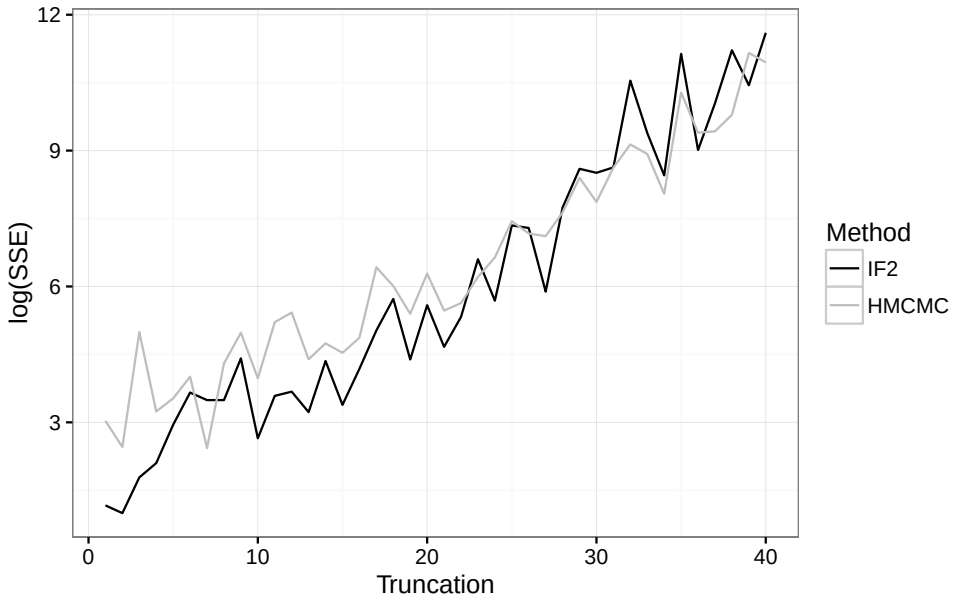

Figure 5: Error growth as a function of data truncation amount. Both methods used 200 bootstrap trajectories. Note that the y-axis shows the natural log of the averaged SSE, not the total SSE.

IF2 and HMCMC perform very closely, with IF2 maintaining a small advantage up to a truncation of about 25-30 data points.

Since the parametric bootstrapping approach used by IF2 requires a significant number of additional fits, its computational cost is significantly higher than the simpler bootstrapping approach used by the HMCMC framework, about 35.5x as expensive. However the now much longer running time can somewhat alleviated by parallelizing the parametric bootstrapping process; as each of the parametric bootstrap fittings in entirely independent, this can be done without a great deal of additional effort. The code used here has this capability, but it was not utilised in the comparison so as to accurately represent total computational cost, not potential running time.

Appendices

A IF2 Parametric Bootstrapping Function

The parametric bootstrapping machinery used to produce forecasts.

```
# Dexter Barrows
#
# IF2 parametric bootstrapping function

library(foreach)
library(parallel)
library(doParallel)
library(Rcpp)

if2_paraboot ← function(if2data_parent, T, Tlim, steps, N, nTrials, if2file
    , if2_s_file, stoc_sir_file, NP, nPasses, coolrate) {

  source(stoc_sir_file)

  if (nTrials < 2)
    ntrials ← 2

  # unpack if2 first fit data
  # ...parameters
  paramdata_parent ← data.frame( if2data_parent$paramdata )
  names(paramdata_parent) ← c("R0", "r", "sigma", "eta", "berr", "Sinit", "
      Iinit", "Rinit")
  parmeans_parent ← colMeans(paramdata_parent)
  names(parmeans_parent) ← c("R0", "r", "sigma", "eta", "berr", "Sinit", "
      Iinit", "Rinit")
  # ...states
  statedata_parent ← data.frame( if2data_parent$statedata )
  names(statedata_parent) ← c("S","I","R","B")
  statemeans_parent ← colMeans(statedata_parent)
  names(statemeans_parent) ← c("S","I","R","B")


  ## use parametric bootstrapping to generate forcasts
  ##
  trajectories ← foreach( i = 1:nTrials, .combine = rbind, .packages = "
      Rcpp") %dopar% {

    source(stoc_sir_file)

    ## draw new data
    ##

    pars ← with( as.list(parmeans_parent),
                 c(R0 = R0,
```

```
                r = r,
                N = N,
                eta = eta,
                berr = berr) )

  init_cond ← with( as.list(parmeans_parent),
                    c(S = Sinit,
                      I = Iinit,
                      R = Rinit) )

  # generate trajectory
  sdeout ← StocSIR(init_cond, pars, Tlim + 1, steps)
  colnames(sdeout) ← c('S','I','R','B')

  # add noise
  counts_raw ← sdeout[,'I'] + rnorm(dim(sdeout)[1], 0, parmeans_parent[['
      sigma']])
    counts     ← ifelse(counts_raw < 0, 0, counts_raw)

    ## refit using new data
    ##

    rm(if2) # because stupid things get done in packages
    sourceCpp(if2file)
    if2time ← system.time( if2data ← if2(counts, Tlim+1, N, NP, nPasses,
        coolrate) )

    paramdata ← data.frame( if2data$paramdata )
  names(paramdata) ← c("R0", "r", "sigma", "eta", "berr", "Sinit", "Iinit
      ", "Rinit")
  parmeans ← colMeans(paramdata)
  names(parmeans) ← c("R0", "r", "sigma", "eta", "berr", "Sinit", "Iinit"
      , "Rinit")

  ## generate the rest of the trajectory
  ##

  # pack new parameter estimates
  pars ← with(  as.list(parmeans),
                c(R0 = R0,
                r = r,
                N = N,
                eta = eta,
                berr = berr) )
  init_cond ← c(S = statemeans_parent[['S']],
                  I = statemeans_parent[['I']],
                  R = statemeans_parent[['R']])

  # generate remaining trajectory part
  sdeout_future ← StocSIR(init_cond, pars, T-Tlim, steps)
  colnames(sdeout_future) ← c('S','I','R','B')

  return ( c( counts = unname(sdeout_future[,'I']),
              parmeans,
```

```
                  time = if2time[['user.self']]) )


  }

  return(trajectories)

}
```

B RStan Forward Simulator

The code used to reconstruct the state estimates, then project the trajectory forward past data.

```
StocSIRstan ← function(y, pars, T, steps, berrvec, bveclim) {

  out ← matrix(NA, nrow = (T+1), ncol = 4)

  R0 ← pars[['R0']]
  r ← pars[['r']]
  N ← pars[['N']]
  eta ← pars[['eta']]
  berr ← pars[['berr']]

  S ← y[['S']]
  I ← y[['I']]
  R ← y[['R']]

  B0 ← R0 * r / N
  B ← B0

  out[1,] ← c(S,I,R,B)

  h ← 1 / steps

  for ( i in 1:(T*steps) ) {

      if (i <= bveclim) {
        B ← exp( log(B) + eta*(log(B0) - log(B)) + berrvec[i])
      } else {
          B ← exp( log(B) + eta*(log(B0) - log(B)) + rnorm(1, 0, berr))
      }

    BSI ← B*S*I
    rI ← r*I

    dS ← -BSI
    dI ← BSI - rI
    dR ← rI

```

```
    S ← S + h*dS  #newInf
    I ← I + h*dI  #newInf - h*dR
    R ← R + h*dR  #h*dR

    if (i %% steps == 0)
      out[i/steps+1,] ← c(S,I,R,B)

  }

  return(out)

}
```