Simulation-based model selection for population biology - STAT 610 - (paper results replication)

Trang Nguyen

Contents

1 1. Context :

In this project, I will replicate the results from the paper "Simulation-based model selection for population biology" by Toni et al. (2009). The goal is to implement the simulation-based model selection approach described in the paper and apply it to a population biology scenario.

The scenario considered is about the spread of different strains of the influenza virus. The data used in the paper comes from influenza A (H3N2) outbreaks that occurred in 1977-1978 and 1980-1981 in Tecomseh, Michigan, (Supplementary table 2) and a second dataset of an influenza B infection outbreak in 1975-1976 and influenza A (H1N1) infection outbreak in 1978-1979 in Seattle, Washington (Supplementary table 3).

Notes: All the codes in this section can be found in the file codes.R

2 2. Questions:

In the paper, the authors focused on two main questions regarding the influenza outbreaks. 1. Can different outbreaks (two periods) of the same strain be described by the same model of disease spread? 2. Can different outbreaks (two periods) of different strains be described by the same model of disease spread?

2.1 Assumptions of the paper :

In the paper, the authors assumed that : - The virus can be spread from the Infected individuals to the Susceptible individuals. - The spread can occur both within households and across the population at large, and these spreads are different.

2.2 Parameters to infer :

In the model, there are three key parameters : - q_c : the probability that a susceptible individual does not get infected from the community. - q_h : the probability that a susceptible individual escapes infection within their household. - w_{js} : the probability that j out of the s susceptibles in a household become infected,

$$w_{js} = \binom{s}{j} w_{jj} (q_c q_h^j)^{s-j}$$

where $w_{0s} = q_{cs}, s = 0, 1, 2, \ldots$ and $w_{jj} = 1 - \sum_{i=0}^{j-1} w_{ij}$.

We want to **infer** q_c and q_h using data from Supplementary table 2 and table 3. In this paper, they considered two models : - for supplementary table 2: one with four parameters (one pair for each outbreak) and one with two parameters (shared between the two outbreaks). - for supplementary table 3: one with four parameters (one pair for each outbreak) and one with three parameters (shared q_h between the two outbreaks, but different q_c for each outbreak).

2.2.1 Predefined parameters :

- Number of particles N = 1000
- Number of replicates for each particle $B_t = 1$ (deterministic simulation)
- Prior distributions of all parameters (hence q_c and q_h) are chosen to be uniform over the range [0, 1].
- Prior distribution over models is uniform (truncated discrete uniform over the number of models M).
- The distance function used to compare simulated and observed data is defined as follows :

$$d(D_0, D^*) = \frac{1}{2}||D_1 - D^*(q_{h1}, q_{c1})||_F + \frac{1}{2}||D_2 - D^*(q_{h2}, q_{c2})||_F$$

$D_0 = D_1 \cup D_2$ is the observed data (the combination of the two outbreaks D_1 and D_2) and D^* is the simulated data from the model. The Frobenius norm is used to measure the difference between the two datasets.

2.2.2 Data

In this part, I will load the data from Supplementary table 2 and table 3 as data frames in R.

```
## Row names for the data frames
nb_infecteds = c("0_infected", "1_infected", "2_infected", "3_infected", "4_infected", "5_infected")

## Infuenza A from 1977-1978
```

```r
suppl2_7778 = data.frame(
    household_1 = c(66, 13, 0, 0, 0 , 0),
    household_2 = c(87, 14, 4, 0, 0, 0),
    household_3 = c(25, 15, 4, 4, 0, 0),
    household_4 = c(22, 9, 9, 3, 1, 0),
    household_5 = c(4, 4, 1, 1, 1, 0))
row.names(suppl2_7778) = nb_infecteds

## Influenza A from 1980-1981
suppl2_8081 = data.frame(
    household_1 = c(44, 10, 0, 0, 0 , 0),
    household_2 = c(62, 13, 9, 0, 0, 0),
    household_3 = c(47, 8, 2, 3, 0, 0),
    household_4 = c(38, 11, 7, 5, 1, 0),
    household_5 = c(9, 5, 3, 1, 0, 1))
row.names(suppl2_8081) = nb_infecteds

## Influenza B from 1975-1976
suppl3_7576 = data.frame(
    household_1 = c(9, 1, 0, 0, 0 , 0),
    household_2 = c(12, 6, 2, 0, 0, 0),
    household_3 = c(18, 6, 3, 1, 0, 0),
    household_4 = c(9, 4, 4, 3, 0, 0),
    household_5 = c(4, 3, 0, 2, 0, 0))
row.names(suppl3_7576) = nb_infecteds

suppl3_7879 = data.frame(
    household_1 = c(15, 11, 0, 0, 0 , 0),
    household_2 = c(12, 17, 21, 0, 0, 0),
    household_3 = c(4, 4, 4, 5, 0, 0))
row.names(suppl3_7879) = nb_infecteds

# Look at the data of table 2
suppl2_7778
```

```
##            household_1 household_2 household_3 household_4 household_5
## 0_infected          66          87          25          22           4
## 1_infected          13          14          15           9           4
## 2_infected           0           4           4           9           1
## 3_infected           0           0           4           3           1
## 4_infected           0           0           0           1           1
## 5_infected           0           0           0           0           0
```

```r
suppl2_8081
```

```
##            household_1 household_2 household_3 household_4 household_5
## 0_infected          44          62          47          38           9
## 1_infected          10          13           8          11           5
## 2_infected           0           9           2           7           3
## 3_infected           0           0           3           5           1
## 4_infected           0           0           0           1           0
## 5_infected           0           0           0           0           1
```

2.3 Functions for epidemic simulation

2.3.1 Implementation of w_{js} and distance function :

Below is the implementation of w_{js}, the distance function, and the outbreak simulation function.

```
# Probability that j out of s susceptibles in a household become infected
w_js = function(j, s, q_c, q_h) {
    # w_0s
    if (j==0){ return (q_c^s) }

    # w_jj for j = 1, 2, ..., s-1
    else if (j==s){
        sum_w_ij = 0

        for (i in 0:(j-1)){
          sum_w_ij = sum_w_ij + w_js(i, j, q_c, q_h)
          }
        return (1 - sum_w_ij)

    # w_js for j = 1, 2, ..., s-1, s different from j
    } else {
        comb = choose(s, j)
        w_jj = w_js(j, j, q_c, q_h)

        return (comb * w_jj * (q_c * q_h^j)^(s - j))
    }
}
# As the paper described the function w_js recursively, I implemented it recursively here.


# Distance function between observed data D_0 and simulated data D_star
distance = function(D_0, D_star) {
    # As D_0 = D1 U D2
    D1 = D_0$D1
    D2 = D_0$D2
    D_star1 = D_star$D1
    D_star2 = D_star$D2

    frob_norm1 = sqrt(sum((D1 - D_star1)^2))
    frob_norm2 = sqrt(sum((D2 - D_star2)^2))

    return (0.5 * frob_norm1 + 0.5 * frob_norm2)
}
```

2.3.2 Outbreak simulation functions:

In the paper: - Supplementary Table 2 summarises the final household outcomes for two influenza A epidemics, in 1977-78 and 1980-81. For each epidemic, the columns correspond to households of a given size (1 to 5 individuals initially susceptible in the household), and the rows give the number of infected individuals in that household at the end of the outbreak (0 to 5). Each entry in the table is the count of households with that combination of household size and final number infected.

- Supplementary Table 3 has the same layout for two different epidemics: an influenza B outbreak in 1975-76 and an influenza A outbreak in 1978-79. Again, columns index household size (number of initially susceptible individuals per household), rows index the final number of infections in the household, and each cell reports how many households of that size experienced that number of infections in the corresponding epidemic.

In the next part, I will implement the outbreak simulation functions based on the above description. Then I will simulate the outbreaks under different models and parameter settings.

```
# Simulate one outbreak given (q_c, q_h) and the observed table for that outbreak
simulate_outbreak = function(q_c, q_h, D_obs) {

  D_obs = as.matrix(D_obs)
  n_sizes = ncol(D_obs)              # total number of susceptibles per household (columns)
  max_j  = nrow(D_obs) - 1           # total number of infected individuals per number of susceptibles

  # Create an empty data matrix to fill in
  # I will fill the table column by column (meaning for each number of susceptibles s = 1, ..., n_sizes
  D_sim = matrix(0, nrow = nrow(D_obs), ncol = ncol(D_obs))

  # For number of susceptibles per household s = 1, ..., n_sizes
  for (s_idx in seq_len(n_sizes)) {
    s = s_idx                        # number of susceptibles in the household
    n_households_s = sum(D_obs[, s_idx]) # number of households with s susceptibles

    if (n_households_s == 0) next # skip if no households having s susceptibles

    # Probabilities for j = 0,...,s w_js function
    j_range = 0:s  # possible number of infected individuals in the household with s susceptibles
    p_js = sapply(j_range, function(j) w_js(j, s, q_c, q_h)) # distribution of having j infected out of


    # numerically robust clipping: ( I encounted some negative probabilities due to numerical errors)
    p_js[p_js < 0] = 0
    total = sum(p_js)
    if (total <= 0) {
      stop("All probabilities zero or negative for s = ", s,
           " with qc = ", q_c, ", qh = ", q_h)
    }
    p_js = p_js / total

    # Make sure that p_js is positive and sums to 1
    if (any(p_js < 0) || abs(sum(p_js) - 1) > 1e-8) {
      stop("Invalid probability distribution p_js")
    }

    # Sample with replacement according to p_js the number of infected individuals for n_households_s h
    samples = sample(j_range, size = n_households_s, replace = TRUE, prob = p_js)


    # Count how many households ended up with j infections (from 0 to max_j)
    counts = table(factor(samples, levels = 0:max_j))

    D_sim[, s_idx] = as.numeric(counts) # fill the matrix
```

```
  }
  return(D_sim)
}
```

Next, I will simulate outbreaks under a given model m and parameter vector theta.

```
# m = 1: 2 parameters (qc, qh) shared
# m = 2: 4 parameters (qc1, qh1, qc2, qh2) or 3 parameters (qc1, qc2, qh) for table 3
simulate_data = function(theta, m, D0, n_params_by_model) {
  if (n_params_by_model[[length(n_params_by_model)]] == 4) {
    if (m == 1) {
      q_c = theta[1]; q_h = theta[2]
      D1_s = simulate_outbreak(q_c, q_h, D0$D1)
      D2_s = simulate_outbreak(q_c, q_h, D0$D2)
    } else if (m == 2) {
      q_c1 = theta[1]; q_h1 = theta[2]
      q_c2 = theta[3]; q_h2 = theta[4]
      D1_s = simulate_outbreak(q_c1, q_h1, D0$D1)
      D2_s = simulate_outbreak(q_c2, q_h2, D0$D2)
    }
  } else if (n_params_by_model[[length(n_params_by_model)]] == 3) {
    if (m == 1) {
      q_c = theta[1]; q_h = theta[2]
      D1_s = simulate_outbreak(q_c, q_h, D0$D1)
      D2_s = simulate_outbreak(q_c, q_h, D0$D2)
    } else if (m == 2) {
      q_c1 = theta[1]; q_c2 = theta[2]; q_h  = theta[3]
      D1_s = simulate_outbreak(q_c1, q_h, D0$D1)
      D2_s = simulate_outbreak(q_c2, q_h, D0$D2)
    }
  }

  return(list(D1 = D1_s, D2 = D2_s))
}
```

3 3. Setup for ABC-SMC algorithm

3.1 Model and prior setup

3.1.1 Prior distributionns

```
# Prior for models
# here we consider only 2 models (M = 2)
prior_model = function(M) {
    return (sample(1:M, size=1, prob=rep(1/M, M)))
}

# Prior distribution for parameter given model m
# n_params_by_model: named vector of number of parameters per model for table 2 or table 3
```

```
prior_theta_parameters = function(m, n_params_by_model) {
    n_par = n_params_by_model[as.character(m)]
    return (runif(n_par, 0, 1))
}
```

3.2 Helper functions : pertubation kernels

In the paper, the authors used a uniform perturbation kernel for parameters KPt. Here, I will call it as pertub_theta.

The kernel is centered at the previous particle, and the scale is set using the previous particles for the same model.

```
# Similar to the paper : KPt

# theta_mat_model: matrix of particles and for T populations (N x T)
# where N is number of particles, T is number of parameters

# theta_prev: previous particle to perturb (vector of length N)
# Returns a new perturbed theta_new, and update the theta_mat_model accordingly.

# theta_star: vector (length d) = centre of the kernel (ancestor)
# theta_prev_mat_model: matrix of previous thetas for this model, N_model x d

perturb_theta = function(theta_star, theta_prev_mat_model) {
  d = length(theta_star)
  sigma = 0.5 * (apply(theta_prev_mat_model, 2, max) -
                   apply(theta_prev_mat_model, 2, min))
  sigma[sigma == 0] = 0.1
  theta_new = numeric(d)
  for (k in seq_len(d)) {
    a = max(0, theta_star[k] - sigma[k])
    b = min(1, theta_star[k] + sigma[k])
    theta_new[k] = runif(1, a, b)
    }
  return (theta_new)
}




# Density of the kernel at theta_new given centre theta_prev.

kernel_theta_density = function(theta_new, theta_prev, theta_prev_mat_model) {
  d = length(theta_prev)
  sigma = 0.5 * (apply(theta_prev_mat_model, 2, max) -
                   apply(theta_prev_mat_model, 2, min))
  sigma[sigma == 0] = 0.1

  dens = 1
  for (p in seq_len(d)) {
    a = max(0, theta_prev[p] - sigma[p])
    b = min(1, theta_prev[p] + sigma[p])
```

```
    if (theta_new[p] < a || theta_new[p] > b) {
      return(0)
    } else {
      dens = dens * (1 / (b - a))    # truncated uniform on [a,b]
    }
  }
  dens
}
```

```
abc_smc_influenza = function(D0,
                             epsilon_schedule,
                             n_params_by_model,
                             N_particles = N,
                             M_models = M) {

  # Initialization
  T_pop   = length(epsilon_schedule)
  max_par = max(n_params_by_model) # maximum number of parameters for all models

  # Save final results in a matrix / list
  models_mat  = matrix(NA_integer_, nrow = N_particles, ncol = T_pop)
  weights_mat = matrix(NA_real_,     nrow = N_particles, ncol = T_pop)
  thetas_list = vector("list", T_pop)  # each [[t]]: N x max_par matrix

  ## ----------------- t = 1: sample directly from priors -----------------
  eps1 = epsilon_schedule[1]
  theta_mat_t1 = matrix(NA_real_, nrow = N_particles, ncol = max_par)

  for (i in seq_len(N_particles)) {
    repeat {
      m_star = prior_model(M_models) # choose the model either 1 or 2
      theta_st = prior_theta_parameters(m_star, n_params_by_model = n_params_by_model) # Given the mode
      D_star = simulate_data(theta_st, m_star, D0, n_params_by_model) # simulate data
      d_val = distance(D0, D_star)

      if (d_val <= eps1) {
        models_mat[i, 1] = m_star
        theta_mat_t1[i, 1:length(theta_st)] = theta_st
        weights_mat[i, 1] = 1  # unnormalised
        break
      }
    }
  }

  # normalise weights at t = 1
  weights_mat[, 1] = weights_mat[, 1] / sum(weights_mat[, 1])
  thetas_list[[1]] = theta_mat_t1

  ## ----------------- t >= 2: SMC steps -----------------
  if (T_pop >= 2) {

    for (t in 2:T_pop) {
      print(paste("Generating population", t, "of", T_pop))
```

```
# next population
epsilon_t = epsilon_schedule[t]

# extract the data from previous population
theta_prev_mat = thetas_list[[t - 1]]   # vector of length N x max_par
model_prev     = models_mat[, t - 1] # vector of length N
w_prev         = weights_mat[, t - 1] # vector of length N

# In case numerical issues give you total weight:
if (sum(w_prev) <= 0) {
  next
}

# even if we have different number of parameters per model,
# we store all particles in a matrix of size N x max_par, padding with NAs
theta_mat_t = matrix(NA_real_, nrow = N_particles, ncol = max_par)
model_t     = integer(N_particles)
w_t         = numeric(N_particles)

# Loop over particles
for (i in seq_len(N_particles)) {
  # print(paste("Generating particle", i, "of", N_particles, "for population", t))
  repeat {
    # From the previous population, we have a sample of (model, theta) particles with weights.
    # We will sample from these to generate new particles for the current population.
    # using the weights from previous population.

    # 1/ Choose an ancestor index across all models
    ancestor_idx = sample(seq_len(N_particles), size = 1, prob = w_prev)
    # print(paste("ancestor_idx", ancestor_idx, "here!"))
    m_prev_star  = model_prev[ancestor_idx]

    # 1b/ Propose new model using kmt
    if (runif(1) < 0.7) {
      m_star = m_prev_star
    } else {
      # with only two models, just flip
      m_star = ifelse(m_prev_star == 1, 2, 1)
    }

    # 2/ Work only with previous particles of this model
    idx_m        = which(model_prev == m_star)

    if (length(idx_m) == 0) {
      next  # go back to the start of repeat{}
    }

    d_m          = n_params_by_model[as.character(m_star)]
    theta_prev_m = theta_prev_mat[idx_m, 1:d_m, drop = FALSE]
    w_prev_m     = w_prev[idx_m]

    if (sum(w_prev_m) <= 0 || any(!is.finite(w_prev_m))) {
```

```
      next
    }

    # 3/ Choose which theta to perturb within this model
    j_idx        = sample(seq_along(idx_m), size = 1, prob = w_prev_m / sum(w_prev_m))
    # print(paste("j_idx", j_idx, "here!"))
    theta_center = theta_prev_m[j_idx, ]

    # 4) Perturb parameters
    theta_st = perturb_theta(theta_center, theta_prev_m)



    # Simulate dataset and compute distance
    D_star = simulate_data(theta_st, m_star, D0, n_params_by_model)
    # print("Here!")

    # print("Data simulated.")
    d_val  = distance(D0, D_star)


    if (d_val <= epsilon_t) {
      model_t[i] = m_star

      theta_full = rep(NA_real_, max_par)
      theta_full[1:length(theta_st)] = theta_st
      theta_mat_t[i, ] = theta_full

      break
    }
  } # repeat for particle i

  ## weight calculation MS3
  # importance of particle i at population t
  m_i = model_t[i]
  d_i = n_params_by_model[as.character(m_i)]
  theta_i_short = theta_mat_t[i, 1:d_i]

  # previous population restricted to same model
  idx_same_model = which(model_prev == m_i)
  theta_prev_m   = theta_prev_mat[idx_same_model, 1:d_i, drop = FALSE]
  w_prev_m       = w_prev[idx_same_model]

  kernel_vals = numeric(length(idx_same_model))
  for (k in seq_along(idx_same_model)) {
    theta_prev_k = theta_prev_m[k, ]
    kernel_vals[k] = kernel_theta_density(theta_i_short,
                                          theta_prev_k,
                                          theta_prev_m)
  }
  denom = sum(w_prev_m * kernel_vals)

  # Prior on m and theta|m are uniform on [0,1] so numerator is constant
```

```
        w_t[i] = ifelse(denom > 0, 1 / denom, 0)
      } # end i particle loop




      # normalise weights
      w_t = w_t / sum(w_t)

      models_mat[, t]  = model_t
      weights_mat[, t] = w_t
      thetas_list[[t]] = theta_mat_t

    } # end t population loop
    # print("....................................")
  }

  return(list(
    models  = models_mat,
    thetas  = thetas_list,
    weights = weights_mat,
    epsilon = epsilon_schedule
  ))
}
```

```
# 8.a Supplementary Table 2
#
D0_suppl2 = list(
  D1 = as.matrix(suppl2_7778),
  D2 = as.matrix(suppl2_8081)
)
epsilon_t = tolerance.sched.t2 # schedule for table 2
res_suppl2 = abc_smc_influenza(
  D0              = D0_suppl2,
  epsilon_schedule = epsilon_t,
  N_particles      = N,
  M_models         = M,
  n_params_by_model = n_params_by_model_t2
)
```

```
## [1] "Generating population 2 of 8"
## [1] "Generating population 3 of 8"
## [1] "Generating population 4 of 8"
## [1] "Generating population 5 of 8"
## [1] "Generating population 6 of 8"
## [1] "Generating population 7 of 8"
## [1] "Generating population 8 of 8"
```

```
# 8.b Supplementary Table 3
D0_suppl3 = list(
  D1 = as.matrix(suppl3_7576),
  D2 = as.matrix(suppl3_7879)
)
```

```r
epsilon_t3 = tolerance.sched.t3 # schedule for table 3
res_suppl3 = abc_smc_influenza(
  D0              = D0_suppl3,
  epsilon_schedule = epsilon_t3,
  N_particles      = N,
  M_models         = M,
  n_params_by_model = n_params_by_model_t3
)
```

```
## [1] "Generating population 2 of 7"
## [1] "Generating population 3 of 7"
## [1] "Generating population 4 of 7"
## [1] "Generating population 5 of 7"
## [1] "Generating population 6 of 7"
## [1] "Generating population 7 of 7"
```

4 4. Results and analysis

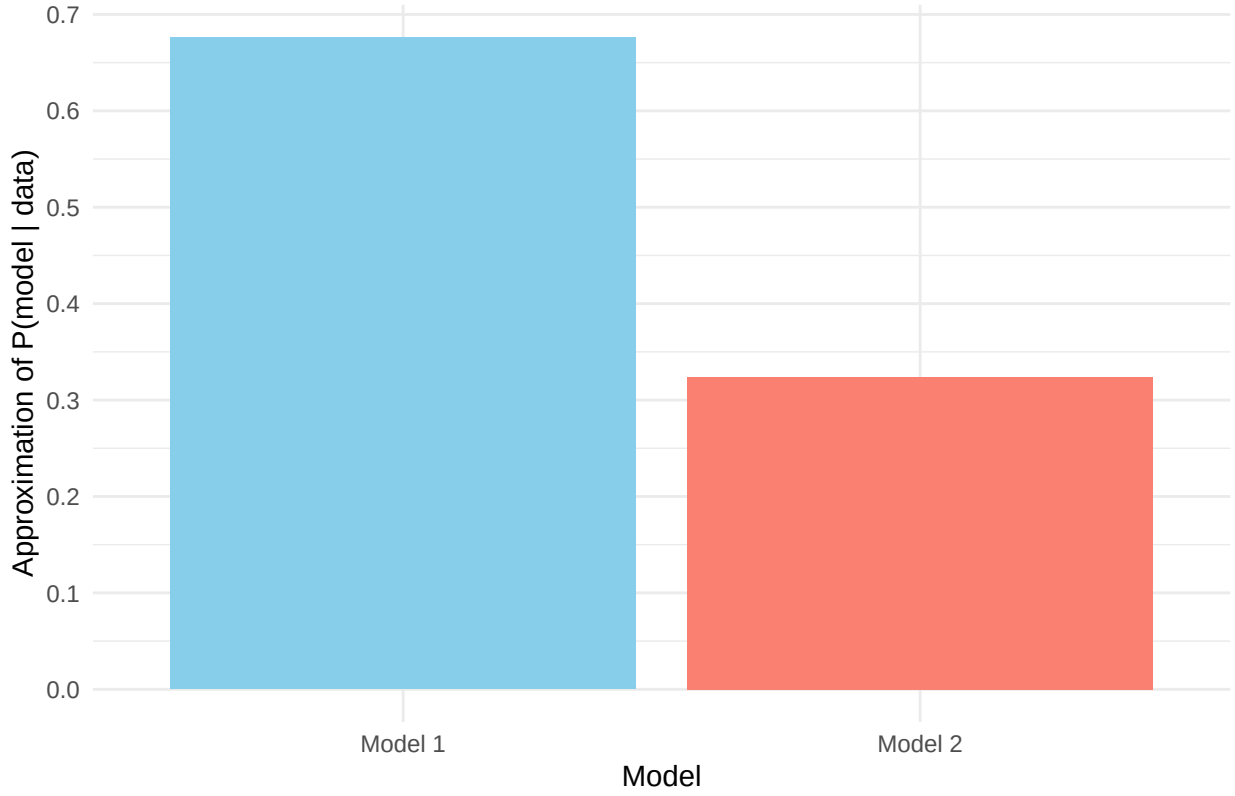
4.1 Reploting figures 3a,b,c,d from the paper

```r
library(ggplot2)
```

```
## Warning: package 'ggplot2' was built under R version 4.4.2
```

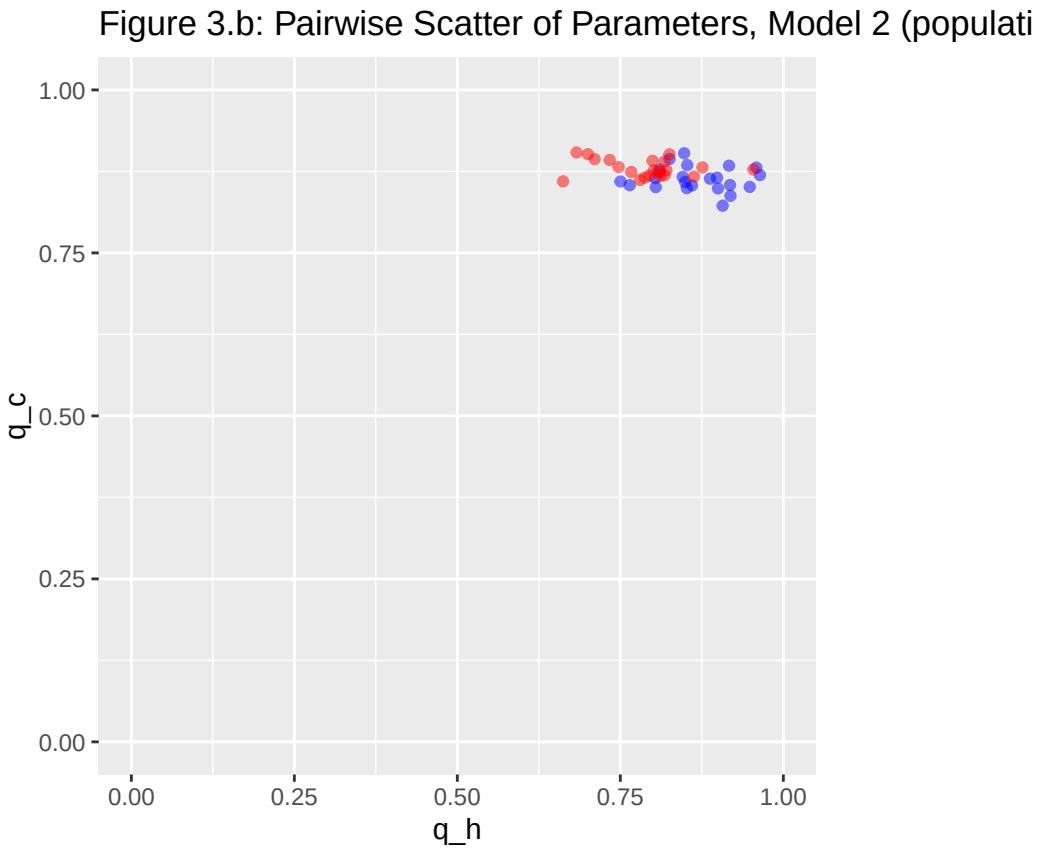
```r
# Supplementary table 2
# Count the number of types of models in total
df_count = as.data.frame(table(res_suppl2$models))

# Normalise the counts to get frequencies
df_count$Var1 = paste("Model", df_count$Var1)
df_count$Freq = df_count$Freq / sum(df_count$Freq)

# Rename columns
colnames(df_count) = c("Model", "P(model | data)")

# Recreate figure 3.a using frequency count
fig3a = ggplot(data = df_count, aes(x = Model, y = `P(model | data)`, fill = Model)) +
  geom_bar(stat = "identity") +
  labs(title = "Figure 3.a: Model Selection Results for Influenza A - 1977-1978 & 1980-1981",
       x = "Model",
       y = "Approximation of P(model | data)") +
  scale_y_continuous(breaks = seq(0, 1, by = 0.1)) +
  scale_fill_manual(values = c("Model 1" = "skyblue", "Model 2" = "salmon")) +
  theme_minimal() +
  theme(legend.position = "none")

print(fig3a)
```

Figure 3.a: Model Selection Results for Influenza A − 1977−1978 & 1980−1




```
# Extract particles for Model 2 where we have 4 parameters
t_last = length(res_suppl2$epsilon)

theta_last  = res_suppl2$thetas[[t_last]]          # N x 4
model_last  = res_suppl2$models[, t_last]          # length N
weight_last = res_suppl2$weights[, t_last]         # length N

# keep only model 2 (4-parameter model)
idx_m2  = which(model_last == 2)
theta_m2  = theta_last[idx_m2, 1:4, drop = FALSE]

qc_1 = theta_m2[, 1]   # q_c1
qh_1 = theta_m2[, 2]   # q_h1
qc_2 = theta_m2[, 3]   # q_c2
qh_2 = theta_m2[, 4]   # q_h2

df_params_model2 = data.frame(qc_1, qh_1, qc_2, qh_2)

fig3b = ggplot() +
  geom_point(data = df_params_model2,
             aes(x = qh_1, y = qc_1),
             alpha = 0.5, colour = "blue") +
  geom_point(data = df_params_model2,
             aes(x = qh_2, y = qc_2),
             alpha = 0.5, colour = "red") +
  coord_cartesian(xlim = c(0, 1), ylim = c(0, 1)) +
```

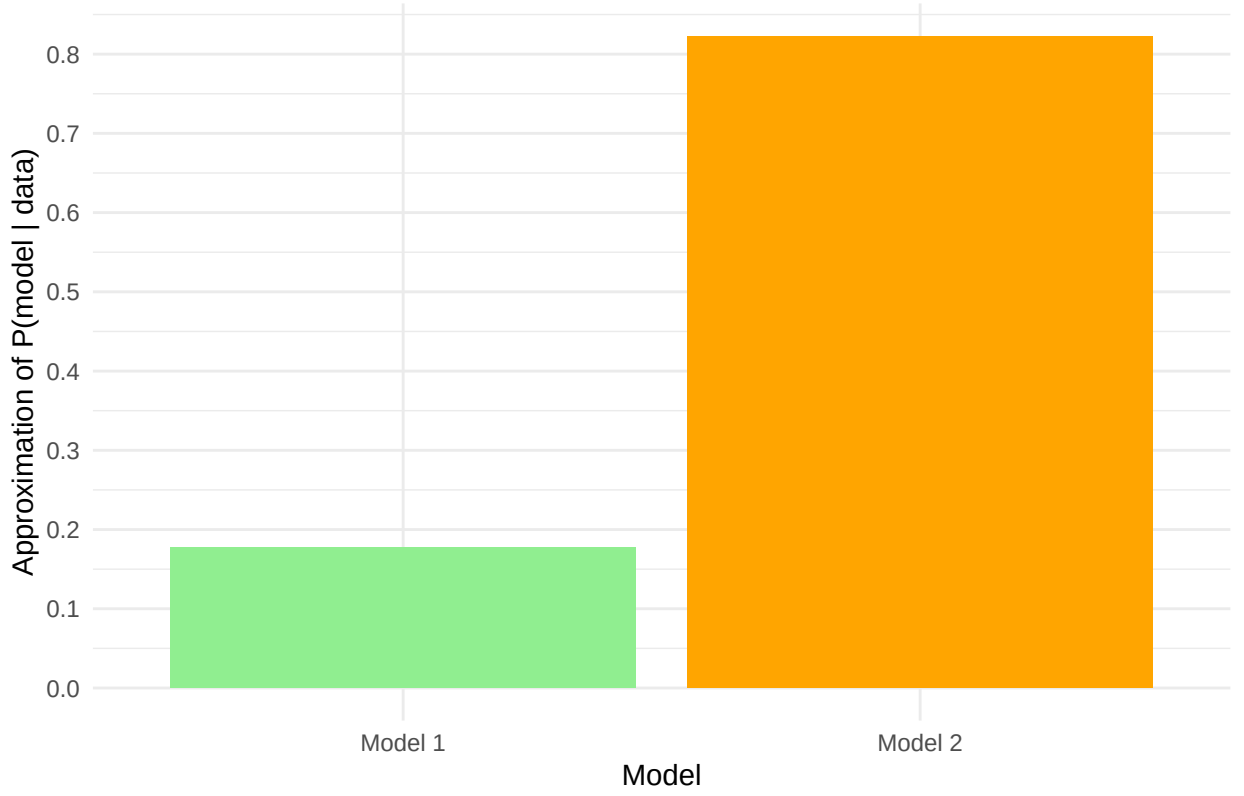
```
  labs(title = "Figure 3.b: Pairwise Scatter of Parameters, Model 2 (population 8)",
       x = "q_h",
       y = "q_c") +
   theme(aspect.ratio=1)
print(fig3b)
```




```
# Supplementary table 3

t_last3 = length(res_suppl3$epsilon)


# Count the number of types of models in total
df_count3 = as.data.frame(table(res_suppl3$models))
# Normalise the counts to get frequencies
df_count3$Var1 = paste("Model", df_count3$Var1)
df_count3$Freq = df_count3$Freq / sum(df_count3$Freq)
# Rename columns
colnames(df_count3) = c("Model", "P(model | data)")
# Recreate figure 3.b using frequency count
fig3c = ggplot(data = df_count3, aes(x = Model, y = `P(model | data)`, fill = Model)) +
  geom_bar(stat = "identity") +
  labs(title = "Figure 3.c: Model Selection Results for Influenza B - 1975-1976 & 1978-1979",
       x = "Model",
       y = "Approximation of P(model | data)") +
  scale_y_continuous(breaks = seq(0, 1, by = 0.1)) +
```

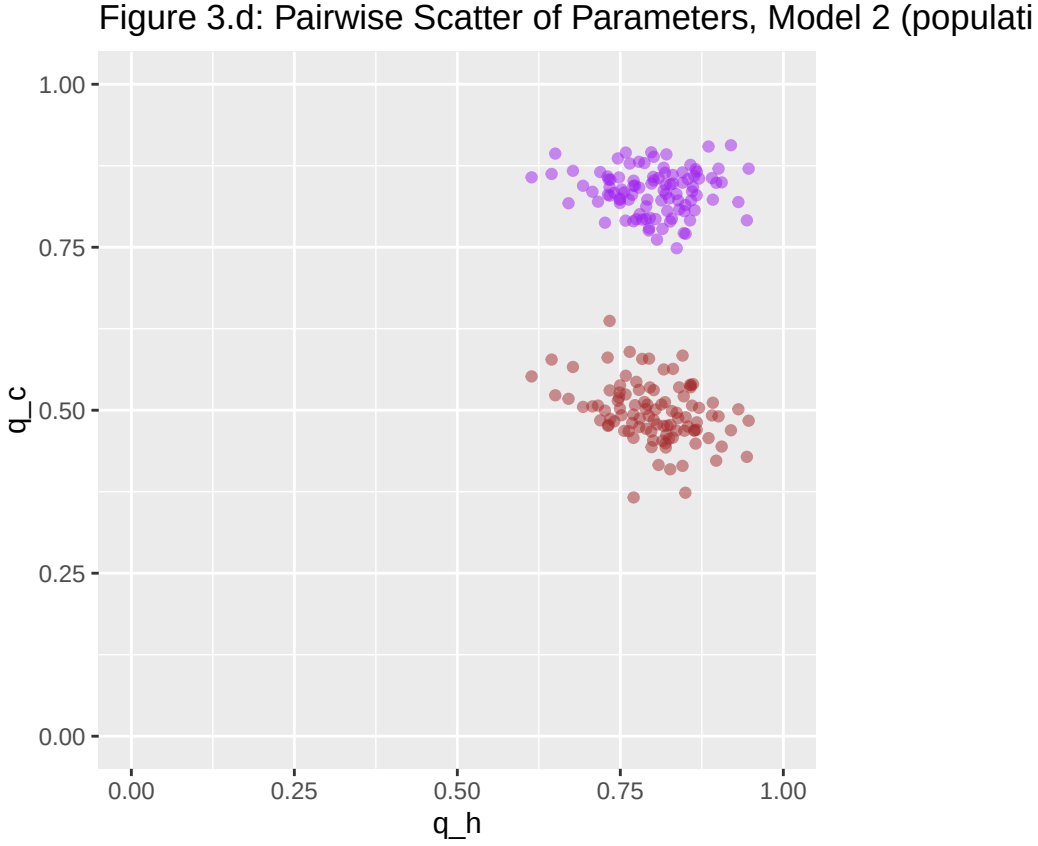
```
  scale_fill_manual(values = c("Model 1" = "lightgreen", "Model 2" = "orange")) +
  theme_minimal() +
  theme(legend.position = "none")
print(fig3c)
```

Figure 3.c: Model Selection Results for Influenza B − 1975−1976 & 1978−1




```
# Extract particles for Model 2 where we have 3 parameters
theta_last3  = res_suppl3$thetas[[t_last3]]          # N x 3
model_last3  = res_suppl3$models[, t_last3]          # length
weight_last3 = res_suppl3$weights[, t_last3]         # length N
# keep only model 2 (3-parameter model)
idx_m2_3  = which(model_last3 == 2)
theta_m2_3  = theta_last3[idx_m2_3, 1:3, drop = FALSE]
qc_1_3 = theta_m2_3[, 1]   # q_c1
qc_2_3 = theta_m2_3[, 2]   # q_c2
qh_3   = theta_m2_3[, 3]   # q_h
df_params_model2_3 = data.frame(qc_1_3, qc_2_3, qh_3)
fig3d = ggplot() +
  geom_point(data = df_params_model2_3,
             aes(x = qh_3, y = qc_1_3),
             alpha = 0.5, colour = "purple") +
  geom_point(data = df_params_model2_3,
             aes(x = qh_3, y = qc_2_3),
             alpha = 0.5, colour = "brown") +
  coord_cartesian(xlim = c(0, 1), ylim = c(0, 1)) +
  labs(title = "Figure 3.d: Pairwise Scatter of Parameters, Model 2 (population 7)",
```

```
       x = "q_h",
       y = "q_c") +
   theme(aspect.ratio=1)
print(fig3d)
```



5 5. Discussion

In this project, I replicated the ABC-SMC method to compare different models and test it on flu outbreak data from households (from a 2009 study by Toni and colleagues). The method looks at two key numbers: q_c (chance of avoiding flu from the community) and q_h (chance of avoiding flu from household members), and it used the final number of sick people in each household size to compare models (shared or non shared parameters). \

For the influenza A data (Table 2 in the supplement), my results match the main finding from the original paper: a more complex model with four parameters (different q_c and q_h values for each of the two A outbreaks) works better than a simpler two-parameter model. This makes sense because the 1977-78 and 1980-81 flu A outbreaks behaved differently, so using separate values for each outbreak gives better results.

For the influenza B/A data (Table 3 in the supplement), my results didn't fully match the published Figure 3c. The original study found no clear winner between two models. But my version shows a stronger preference for model 2 over the other. I think this difference came from my setup (different random seeds, some clipping values of the uniformdistribution). These choices can affect the final model comparisons, especially with small datasets like Table 3.

Even though my Figure 3c results didn't match perfectly, this work shows that the ABC-SMC method works well for comparing models when we can only run simulations and can't calculate probabilities directly. It

also shows that AB-SMC results depend heavily on setup choices (especially the tolerance levels, the choice of priors and the disease models themselves). I think it would be beneficial next to explore more different tolerance settings more carefully and run the analysis multiple times with different random starting points.

The result can be found in the link : https://github.com/TrangNg-Th/STATS-610