Bias EM-algorithm mixture of binomials

Han Bossier

29/5/2018

Contents

1 Introduction

In this report, I will check the implementation of an EM-algorithm to estimate the parameters of a mixture of two binomial ditributions. This function is implementend in the `NeuRRoStat` **R** package. We will use Monte-Carlo simulations to check the bias of the algorithm under different starting values. Furthermore, I will look at a two-step procedure in which we first use *naive* starting values in the algorithm, and then use values in close proximity as a *best guess* for a second run of the algorithm.

2 Theory

Consider the situation where we have V observations. In each of these, we observe the amount of successes out of N trials. We assume all V observations are independent. Furthermore, we know that the observations belong either to one class with a probability of success denoted as π_1 and the other observations to another class with probability π_2. Each observations is a realization of a mixture of two Binomial distributions. In this report, we use the E(xpectation) M(aximization) algorithm to estimate the parameters of this mixture density. Let us denote the following parameters:

- λ: the proportion of truly active observations
- π_1: the probability of an observation belonging to the first category
- π_2: the probability of an observation belonging to the second category

Now assume Y is i.i.d. from:

$$\lambda P_1(Y; k, \pi_1) + (1 - \lambda) P_2(Y; k, \pi_2)$$

The complete log-likelihood of the data over all observations (V) is then given as:

$$\text{cst} + \sum_{v=1}^{V} \log \left[\lambda (\pi_1)^{Y_{(v)}} (1 - \pi_1)^{\left(N - Y_{(v)}\right)} + (1 - \lambda)(\pi_2)^{Y_{(v)}} (1 - \pi_2)^{(N - Y_{(v)})} \right],$$

where $Y_{(v)}$ are the amount of successes observed in observation v with $v = 1, \ldots, V$. Then N is the amount of trials used in all observations.

The EM-algorithm is then used to estimate λ, π_1 and π_2.

3 Implementation

The algorithm is implemented in the **R** package `NeuRRoStat`. The function takes a vector of observations (Y), an integer with the amount of trials (N), initial values for the 3 parameters, an optional parameter to specify the maximum amount of iterations and finally the tolerance level. The latter is used when comparing the likelihood of previous and current step in the algorithm. Convergence is assumed when the absolute difference between both log likelihoods is smaller than the tolerance level.

The function looks as follow:

```
function(Y, N, iniL, iniPI1, iniPI2, max.iter = 500, tolerance = 0.001){

  # Give warning if iniPI1 and iniPI2 have the same value
  if(iniPI1 == iniPI2) warning('Using the same starting values for both
                probabilities (pi 1 and pi 2) might lead to bad convergence!')

  ###################
  # Local functions
  ###################

  # Define function weights (using current step in algorithm)
  weight <- function(lambda, pi_1, pi_2, N, Yi){
    # First calculate numinator: lambda times density of binomial
    num <- lambda * dbinom(x = Yi, size = N, prob = pi_1)
    # Now calculate denominator
    denom <- num + ((1 - lambda) * dbinom(x = Yi, size = N, prob = pi_2))
    # Return value
    return(num/denom)
  }

  # Maximalization function that returns updated
  #   value for pi_1, given current values (_c)
  update_pi_1 <- function(lambda_c, pi_1_c, pi_2_c, N, Yi){
    # Weight of current step
    weight_c <- weight(lambda_c, pi_1_c, pi_2_c, N, Yi = Yi)
    # First calculate numinator
    num <- sum(weight_c * Yi)
    # Now calculate denominator
    denom <- sum(weight_c * N)
    # Return value
    return(num/denom)
  }

  # Same for pi_2
  update_pi_2 <- function(lambda_c, pi_1_c, pi_2_c, N, Yi){
    # Weight of current step
    weight_c <- weight(lambda_c, pi_1_c, pi_2_c, N, Yi = Yi)
    # First calculate numinator
    num <- sum(((1 - weight_c) * Yi))
```

```
  # Now calculate denominator
  denom <- sum(((1 - weight_c) * N))
  # Return value
  return(num/denom)
}

# Same for lambda
update_lambda <- function(lambda_c, pi_1_c, pi_2_c, N, Yi){
  mean(weight(lambda_c, pi_1_c, pi_2_c, N, Yi))
}

##################
# EM ALGORITHM  #
##################

# Start for loop of algorithm steps
for(v in 1:max.iter){
  ################################
  ## 1: Initial starting values ##
  ################################
  if(v == 1){
    lambda <- iniL
    pi_1 <- iniPI1
    pi_2 <- iniPI2
  }

  ################################
  ## 2: Conditional expectation ##
  ################################

  # See local functions

  #####################################
  ## 3: Maximise (update parameters) ##
  #####################################

  # Current log likelihood
  logLik_current <- sum((lambda * pi_1^Y * (1 - pi_1)^(N - Y)) +
                          ((1 - lambda)*pi_2^Y * (1 - pi_2)^(N - Y)))

  # Parameter: pi_A1
  pi_1_up <- update_pi_1(lambda, pi_1, pi_2, N, Y)
  # Parameter: pi_I1
  pi_2_up <- update_pi_2(lambda, pi_1, pi_2, N, Y)
  # Paramater: lambda
  lambda_up <- update_lambda(lambda, pi_1, pi_2, N, Y)

  # Now we replace the parameters
  lambda <- lambda_up
  pi_1 <- pi_1_up
  pi_2 <- pi_2_up

  ###################################
```

```
    ## 4: Stopping rule: convergence ##
    ###################################

    # If the absolute difference between the log likelihoods is
    #  smaller than the tolerance, then stop the algorithm
    logLik_update <- sum((lambda * pi_1^Y * (1 - pi_1)^(N - Y)) +
                           ((1 - lambda)*pi_2^Y * (1 - pi_2)^(N - Y)))
    if(abs(logLik_update - logLik_current) < tolerance) break

  }

  # data frame with results
  res <- data.frame('lambda' = lambda, 'PI1' = pi_1, 'PI2' = pi_2,
                    'num.iter' = v)
  return(res)
}
```

4 Simulation

4.1 Details

We use 500 Monte-Carlo simulations to investigate whether we obtain unbiased results. In each, we generate V independent observations where we set $V = 100.000$. In each observation, we have $N = 25$ trials.

```
nsim <- 500
Vobs <- 100000
N <- 25
```

We create data by first specifying a mixing parameter (Z) which puts the observation in class 1 with probability λ. Then, depending on the value of Z, we sample from a binomial distribution with probability of success being π_1 or π_2. Thus we have (as an example):

```
# First generate the mixing parameter (Z)
data.frame(Z = rbinom(n = 50, size = 1, prob = 0.25)) %>%
  # Mutate column with data, depending on mixing parameter
  mutate(Y = ifelse(Z == 1,
            rbinom(n = 1, size = 25, prob = 0.8),
            rbinom(n = 1, size = 25, prob = 0.4)))
```

```
# A tibble: 50 x 2
       Z     Y
   <int> <int>
 1     0    12
 2     0    12
 3     0    12
 4     0    12
 5     1    22
 6     0    12
 7     1    22
 8     1    22
 9     0    12
10     0    12
# ... with 40 more rows
```

4.1.1 True values

We take as true values:

- TRUE $\lambda = 0.05$
- TRUE $\pi_1 = 0.7$
- TRUE $\pi_2 = 0.1$

```
true_lambda <- 0.05
true_pi1 <- 0.7
true_pi2 <- 0.1
```

4.1.2 Simulation settings

We consider 4 simulation settings. In the first, we take as initial starting values:

- START $\lambda = 0.05$
- START $\pi_1 = 0.7$
- START $\pi_2 = 0.1$

These are equal to the true values.

In the second scenario, we have:

- START $\lambda = 0.8$
- START $\pi_1 = 0.1$
- START $\pi_2 = 0.7$

Which are values far from the true values. In the third scenario, we use a two-step procedure:

- START $\lambda_{v_0} = 0.1 \rightarrow \lambda_v = \text{round}(\lambda_{v_0}, 4)$
- START $\pi_{1v_0} = 0.8 \rightarrow \pi_{1v} = \text{round}(\pi_{1v_0}, 4)$
- START $\pi_{2v_0} = 0.05 \rightarrow \pi_{2v} = \text{round}(\pi_{2v_0}, 4)$

Where we use the function round(x) to round the number within brackets to x digits. We thus use an initial guess at step 0 and then start the EM-algorithm with a value close to the one obtained in step 0.

In the final scenario, we use a *robust-sensitivity* procedure. We generate a range of 4 plausible values for each of the parameters. Then we run the EM-algorithm for each combination of those values. When we have 4 values for 3 parameters, we get 64 possible combinations. We then take the median value (i.e. the robustness) of the vector of 64 obtained values for the parameter estimates. The number of iterations is the mean of the observed number of iterations. Here we use the mean as we do want longer iterations to carry more weight. Sensible values in the setting of fMRI data are:

- START $\lambda \in \{0.01, \ldots, 0.3\}$
- START $\pi_1 \in \{0.6, \ldots, 0.95\}$
- START $\pi_2 \in \{0.05, \ldots, 0.4\}$

4.2 Simulation

We now run the full simulation.

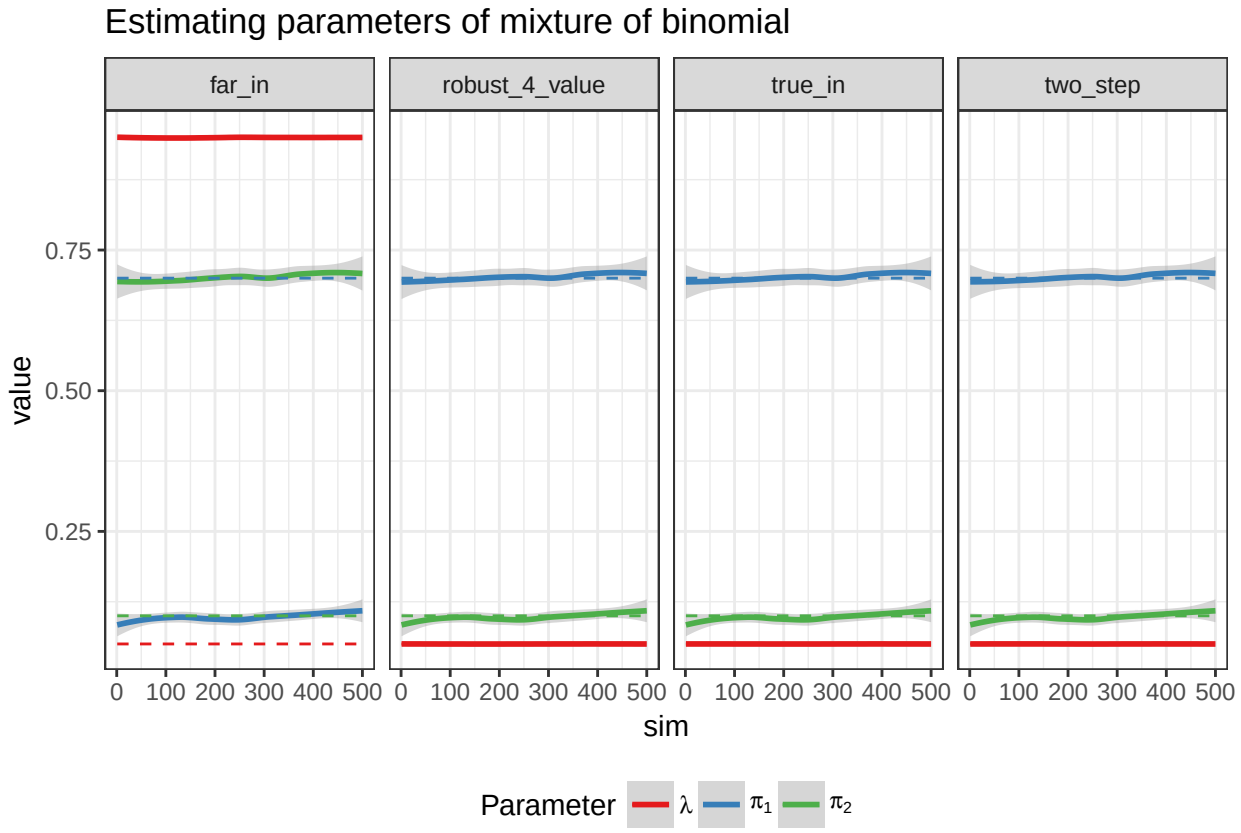
```
# Set seed
set.seed(pi * 3.14)

# Number of scenarios
numScen <- 4
```

```r
# Empty data frame
sim_dat <- data.frame() %>% as_tibble()

# Simulation for loop
for(i in 1:nsim){
  # Create data
  # First generate the mixing parameter (Z)
  TDat_mix <- data.frame(Z = rbinom(n = Vobs, size = 1, prob = true_lambda)) %>%
    # Mutate column with data, depending on mixing parameter
    mutate(Y = ifelse(Z == 1,
              rbinom(n = 1, size = N, prob = true_pi1),
              rbinom(n = 1, size = N, prob = true_pi2)))

  # Now deploy the EM algorithm:
  # Setting 1
  true_in <- NeuRRoStat::EMbinom(Y = TDat_mix$Y, N = N, iniL = true_lambda,
                      iniPI1 = true_pi1, iniPI2 = true_pi2)
  # Setting 2
  far_in <- NeuRRoStat::EMbinom(Y = TDat_mix$Y, N = N, iniL = 0.80,
                      iniPI1 = 0.1, iniPI2 = 0.7)
  # Setting 3
  burn_in <- NeuRRoStat::EMbinom(Y = TDat_mix$Y, N = N, iniL = 0.1,
                      iniPI1 = 0.8, iniPI2 = 0.05)
  two_step <- NeuRRoStat::EMbinom(Y = TDat_mix$Y, N = N,
                          iniL = round(burn_in$lambda, 4),
                      iniPI1 = round(burn_in$PI1, 4),
                      iniPI2 = round(burn_in$PI2, 4))
  # Setting 4
  lams <- seq(0.01,0.3,length.out = 4)
  pi1s <- seq(0.6,0.95, length.out = 4)
  pi2s <- seq(0.05,0.4, length.out = 4)
  combParam <- expand.grid(lams = lams, pi1s = pi1s, pi2s = pi2s)
  avEM <- data.frame()
  for(r in 1:dim(combParam)[1]){
    avEM <- NeuRRoStat::EMbinom(Y = TDat_mix$Y, N = N, iniL = combParam[r, 'lams'],
                    iniPI1 = combParam[r, 'pi1s'], iniPI2 = combParam[r, 'pi2s']) %>%
      bind_rows(avEM, .)
  }
  robust_4_value <- data.frame(lambda = median(avEM$lambda),
                               PI1 = median(avEM$PI1),
                               PI2 = median(avEM$PI2),
                               num.iter = mean(avEM$num.iter))

  # Gather results in data frame
  sim_dat <- data.frame(value = c(t(true_in), t(far_in), t(two_step), t(robust_4_value)),
             trueVal = rep(c(true_lambda, true_pi1, true_pi2, NA), numScen),
             param = rep(rownames(t(true_in)), numScen),
             setting = rep(c('true_in', 'far_in', 'two_step', 'robust_4_value'), each = 4),
             sim = i) %>%
    bind_rows(sim_dat, .)
}
```

5 Results

Let us now plot the results.

```
# Filter out number of iterations
sim_dat %>%
  filter(param != 'num.iter') %>%
    ggplot(., aes(x = sim, y = value, group = param)) +
    geom_smooth(aes(colour = param)) +
    geom_line(aes(x = sim, y = trueVal, colour = param), linetype = 'dashed')+
    facet_grid(. ~ setting) +
    scale_color_manual('Parameter', values = c('#e41a1c', '#377eb8', '#4daf4a'),
                       labels = expression(lambda, pi[1], pi[2])) +
    theme_bw() +
    ggtitle("Estimating parameters of mixture of binomial") +
    theme(legend.position = 'bottom')
```



If the starting values deviate very far from the true values, the algorithm flips the estimates! Potentially, this is mainly driven by the current estimate of λ. For fMRI, we generally know what to expect. Furthermore, we do not seem to find much differences between the 3 methods that produce correct values, starting from estimates close to what we expect. The robust procedure is very slow. Therefor, we suggest to use the two-step procedure, just to be safe.