Hands-on training session 2

Hui-Walter models for diagnostic test evaluation

Matt Denwood Giles Innocent

2020-02-18

Introduction

Overview

Date/time:

- 19th February 2020
- 16.00 - 17.00

Teachers:

- Matt Denwood (presenter)
- Giles Innocent

Recap

- Fitting models using MCMC is easy with JAGS / runjags
- But we must *never forget* to check convergence and effective sample size!
- More complex models become easy to implement
    - For example imperfect diagnostic tests
    - But remember to be realistic about what is possible with your data

Recap

- Fitting models using MCMC is easy with JAGS / runjags
- But we must *never forget* to check convergence and effective sample size!
- More complex models become easy to implement
    - For example imperfect diagnostic tests
    - But remember to be realistic about what is possible with your data
- So how do we extend these models to multiple diagnostic tests?

Session 2a: Hui-Walter models for 2 tests and 1 population

Hui-Walter Model

- A particular model formulation that was originally designed for evaluating diagnostic tests in the absence of a gold standard
- Not necessarily (or originally) Bayesian but now usually implemented using Bayesian MCMC
- But evaluating an imperfect test against another imperfect test is a bit like pulling a rabbit out of a hat
    - If we don't know the true disease status, how can we estimate sensitivity or specificity for either test?

Model Specification

```
model{
  Tally ~ dmulti(prob, TotalTests)

  # Test1- Test2-
    prob[1] <- (prev * ((1-se[1])*(1-se[2]))) + ((1-prev) *

  # Test1+ Test2-
    prob[2] <- (prev * ((se[1])*(1-se[2]))) + ((1-prev) * (

  # Test1- Test2+
    prob[3] <- (prev * ((1-se[1])*(se[2]))) + ((1-prev) * (

  # Test1+ Test2+
    prob[4] <- (prev * ((se[1])*(se[2]))) + ((1-prev) * ((
  prev ~ dbeta(1, 1)
  se[1] ~ dbeta(1, 1)
  sp[1] ~ dbeta(1, 1)
  se[2] ~ dbeta(1, 1)
  sp[2] ~ dbeta(1, 1)

  #data# Tally, TotalTests
  #monitor# prev, prob, se, sp
  #inits# prev, se, sp
}
```

```r
twoXtwo <- matrix(c(48, 12, 4, 36), ncol=2, nrow=2)
twoXtwo
```

```
##      [,1] [,2]
## [1,]   48    4
## [2,]   12   36
```

```r
library('runjags')

Tally <- as.numeric(twoXtwo)
TotalTests <- sum(Tally)

prev <- list(chain1=0.05, chain2=0.95)
se <- list(chain1=c(0.5,0.99), chain2=c(0.99,0.5))
sp <- list(chain1=c(0.5,0.99), chain2=c(0.99,0.5))

results <- run.jags('basic_hw.bug', n.chains=2)
```

```
## Warning: You should update the rjags package to version
## (This warning is given once per R session)
```

```
results
```

	Lower95	Median	Upper95	SSeff	psrf
prev	0.311	0.440	0.576	3967	1
prob[1]	0.368	0.461	0.558	13300	1
prob[2]	0.074	0.133	0.204	13181	1
prob[3]	0.018	0.055	0.104	9780	1
prob[4]	0.257	0.344	0.441	13238	1
se[1]	0.821	0.932	1.000	5569	1
se[2]	0.687	0.848	1.000	3111	1
sp[1]	0.748	0.877	1.000	3179	1
sp[2]	0.861	0.948	1.000	5765	1

- Note the wide confidence intervals!

Practicalities

- These models need A LOT of data
  - And/or strong priors for one of the tests
- Convergence is more problematic than usual
- Be **very** vareful with the order of combinations in dmultinom!
- Check your results carefully to ensure they make sense!

Label Switching

How to interpret a test with Se=0% and Sp=0%?

Label Switching

How to interpret a test with Se=0% and Sp=0%?

- The test is perfect - we are just holding it upside down...

Label Switching

How to interpret a test with Se=0% and Sp=0%?

- The test is perfect - we are just holding it upside down...

We can force se+sp >= 1:

```
  se[1] ~ dbeta(1, 1)
  sp[1] ~ dbeta(1, 1)T(1-se[1], )
```

Or:

```
  se[1] ~ dbeta(1, 1)T(1-sp[1], )
  sp[1] ~ dbeta(1, 1)
```

But not both!

This allows the test to be useless, but not worse than useless

Simulating data

Analysing simulated data is useful to check that we can recover parameter values.

```r
se1 <- 0.9; sp1 <- 0.95; sp2 <- 0.99; se2 <- 0.8
prevalence <- 0.5; N <- 100

truestatus <- rbinom(N, 1, prevalence)
Test1 <- rbinom(N, 1, (truestatus * se1) + ((1-truestatus) *
↪  (1-sp1)))
Test2 <- rbinom(N, 1, (truestatus * se2) + ((1-truestatus) *
↪  (1-sp2)))

twoXtwo <- table(Test1, Test2)
twoXtwo
```

```
##      Test2
## Test1  0  1
##     0 38  1
```

Exercise

Modify JAGS code to force tests to be better than useless

Simulate data and recover parameters for:

- N=10, N=100, N=1000

Optional Exercise

Use priors for test1 taken from session 1 and compare the results

Session 2b: Hui-Walter models for 2 tests and N populations

Hui-Walter models with multiple populations

- Basically an extension of the single-population model
- Works best with multiple populations each with differing prevalences
    - These could even be subgroups of individuals within the same population if there are e.g. known risk factors for disease status
    - Remember that the focus is usually to estimate the diagnostic test parameters and not the prevalence in the different populations/subgroups!

Independent intercepts for populations

```
model{
  for(p in 1:Populations){
    Tally[1:4, p] ~ dmulti(prob[1:4, p], TotalTests[p])
    # Test1- Test2- Pop1
      prob[1, p] <- (prev[p] * ((1-se[1])*(1-se[2]))) +
    ((1-prev[p]) * ((sp[1])*(sp[2])))
    ## etc ##
    prev[p] ~ dbeta(1, 1)
  }

  se[1] ~ dbeta(HPSe[1,1], HPSe[1,2])T(1-sp[1], )
  sp[1] ~ dbeta(HPSp[1,1], HPSp[1,2])
  se[2] ~ dbeta(HPSe[2,1], HPSe[2,2])T(1-sp[2], )
  sp[2] ~ dbeta(HPSp[2,1], HPSp[2,2])

  #data# Tally, TotalTests, Populations, HPSe, HPSp
  #monitor# prev, prob, se, sp
  #inits# prev, se, sp
}
```

Auto Hui-Walter

We would usually start with individual-level data in a dataframe:

```r
se1 <- 0.9; sp1 <- 0.95; sp2 <- 0.99; se2 <- 0.8
prevalences <- c(0.1, 0.5, 0.9)
N <- 100

simdata <- data.frame(Population = sample(seq_along(prevalences),
↪  N, replace=TRUE))
simdata$probability <- prevalences[simdata$Population]
simdata$truestatus <- rbinom(N, 1, simdata$probability)
simdata$Test1 <- rbinom(N, 1, (simdata$truestatus * se1) +
↪  ((1-simdata$truestatus) * (1-sp1)))
simdata$Test2 <- rbinom(N, 1, (simdata$truestatus * se2) +
↪  ((1-simdata$truestatus) * (1-sp2)))
```

```r
head(simdata)
```

```
##   Population probability truestatus Test1 Test2
## 1          3         0.9           1     1     0
## 2          1         0.1           0     0     0
## 3          2         0.5           0     0     0
## 4          2         0.5           0     0     0
## 5          1         0.1           0     0     0
## 6          1         0.1           0     1     0
```

[Except that probability and truestatus would not normally be known!]

- The model code and data format for an arbitrary number of populations (and tests) can be determined automatically
- There is a function (provided in the GitHub repo) that can do this for us:

```r
simdata$Population <- factor(simdata$Population,
↪  levels=seq_along(prevalences), labels=paste0('Pop_',
↪  seq_along(prevalences)))

source("autohuiwalter.R")
auto_huiwalter(simdata[,c('Population','Test1','Test2')],
↪  outfile='autohw.bug')
```

```
## The model and data have been written to autohw.bug in th
## You should check and alter priors before running the moc
```

- This generates self-contained model/data/initial values etc:
    - [ignore covse and covsp for now]

```
## ## Auto-generated Hui-Walter model created by script ver
##
## model{
##
##  ## Observation layer:
##
##  # Complete observations (N=100):
##  for(p in 1:Populations){
##      Tally_RR[1:4,p] ~ dmulti(prob_RR[1:4,p], N_RR[p])
##
##      prob_RR[1:4,p] <- se_prob[1:4,p] + sp_prob[1:4,p]
##  }
##
##
```

And can be run directly from R:

```r
results <- run.jags('autohw.bug')
```

```
## Note: The monitored variables 'covse12' and 'covsp12'
## appear to be non-stochastic; they will not be
## included in the convergence diagnostic
## Finished running the simulation
```

```r
results
```

```
##
## JAGS model summary statistics from 20000 samples (chains
##
##              Lower95   Median Upper95     Mean       SD
## se[1]        0.81441  0.92044 0.99704  0.91299 0.050702
## se[2]        0.70145  0.84636  0.9787  0.84086 0.072665
## sp[1]        0.78389  0.88872 0.97862  0.88467  0.05079
## sp[2]        0.90831  0.97034       1  0.96371 0.028238
## prev[1]    0.0061492 0.084139 0.20415 0.093738 0.054871
## prev[2]       0.1555  0.31548 0.50268   0.3207 0.089942
## prev[3]      0.69238  0.84704 0.97197  0.83831  0.07467
## covse12            0        0       0        0        0
## covsp12            0        0       0        0        0
##
```

- Modifying priors must still be done directly in the model file
- The model needs to be re-generated if the data changes
  - But remember that your modified priors will be reset
- There must be a single column for the population (as a factor), and all of the other columns (either factor, logical or numeric) are interpreted as being test results
- The function will soon be included in the runjags package
  - Feedback welcome!

Observation-level model specification

```
model{

  for(i in 1:N){
    Status[i] ~ dcat(prob[i, ])

      prob[i,1] <- (prev[i] * ((1-se[1])*(1-se[2]))) +
                  ((1-prev[i]) * ((sp[1])*(sp[2])))
      prob[i,2] <- (prev[i] * ((se[1])*(1-se[2]))) +
                  ((1-prev[i]) * ((1-sp[1])*(sp[2])))
      prob[i,3] <- (prev[i] * ((1-se[1])*(se[2]))) +
                  ((1-prev[i]) * ((sp[1])*(1-sp[2])))
      prob[i,4] <- (prev[i] * ((se[1])*(se[2]))) +
                  ((1-prev[i]) * ((1-sp[1])*(1-sp[2])))
  intercept ~ dnorm(0, 0.33)
  population_effect[1] <- 0
  for(p in 2:Pops){
    population_effect[p] ~ dnorm(0, 0.1)
  }
  se[1] ~ dbeta(1, 1)T(1-sp[1], )
  sp[1] ~ dbeta(1, 1)
  se[2] ~ dbeta(1, 1)T(1-sp[2], )
  sp[2] ~ dbeta(1, 1)

  #data# Status, N, Population, Pops
  #monitor# intercept, population_effect, se, sp
  #inits# intercept, population_effect, se, sp
}
```

- The main difference is the prior for prevalence in each population
- We also need to give initial values for `intercept` and `population_effect` rather than `prev`, and tell `run.jags` the data frame from which to extract the data (except `N` and `Pops`):

```r
intercept <- list(chain1=-1, chain2=1)
population_effect <- list(chain1=c(NA, 1, -1), chain2=c(NA, -1,
↪  1))
se <- list(chain1=c(0.5,0.99), chain2=c(0.99,0.5))
sp <- list(chain1=c(0.5,0.99), chain2=c(0.99,0.5))

simdata$Status <- with(simdata, factor(interaction(Test1, Test2),
↪  levels=c('0.0','1.0','0.1','1.1')))
N <- nrow(simdata)
Pops <- length(levels(simdata$Population))
glm_results <- run.jags('glm_hw.bug', n.chains=2, data=simdata)
```

```
## Note: The monitored variable 'population_effect[1]'
```

Also like in session 1, the estimates for se/sp should be similar, although this model runs more slowly.

Note: this model could be used as the basis for adding covariates

For a handy way to generate a GLM model see runjags::template.jags

- Look out for integration with autohuiwalter in the near (ish) future…

Practicalities

Need to be very careful with tabulating the data, or use automatically generated code

Works best when populations have very different prevalences

Exercise

Play around with the autohuiwalter function

Notice the model and data and initial values are in a self contained file

Ignore the covse and covsp for now

[There is no particular solution to this exercise!]

Summary

Summary

- Using JAGS / runjags allows us to work with MCMC more easily, safely and efficiently than writing our own sampling algorithms
- But we must *never forget* to check convergence and effective sample size!
- More complex models become easy to implement
  - For example imperfect diagnostic tests
- But just because a model can be defined does not mean that it will be useful for our data
  - We need to be realistic about the information available in the data, what parameters are feasible to estimate, and where we will need to use strong priors