Hands-on training session 1

Introduction, revision and running basic models

Matt Denwood Giles Innocent

2020-02-12

Course Outline and Practicalities

Overview

Date/time:

- 19th February 2020
- 14.00 - 15.30

Teachers:

- Matt Denwood (presenter)
- Giles Innocent

Preparation

- Have we all looked at the preparation material?
- Do we all have R and JAGS installed?
- Brief demonstration of GitHub and where the files are...

Revision

Bayes Rule

Bayes' theorem is at the heart of Bayesian statistics. This states that:

$$P(\theta|Y) = \frac{P(\theta) \times P(Y|\theta)}{P(Y)}$$

Bayes Rule

Bayes' theorem is at the heart of Bayesian statistics. This states that:

$$P(\theta|Y) = \frac{P(\theta) \times P(Y|\theta)}{P(Y)}$$

Where: θ is our parameter value(s); Y is the data that we have observed; $P(\theta|Y)$ is the posterior probability of the parameter value(s) given the data and priors; $P(\theta)$ is the prior probability of the parameters BEFORE we had observed the data; $P(Y|\theta)$ is the likelihood of the data given the parameters value(s), as discussed above; and $P(Y)$ is the probability of the data, integrated over all parameter space.

Note that $P(Y)$ is rarely calculable except in the simplest of cases, but is a constant for a given model. So in practice we usually work with the following:

$$P(\theta|Y) \propto P(\theta) \times P(Y|\theta)$$

Note that $P(Y)$ is rarely calculable except in the simplest of cases, but is a constant for a given model. So in practice we usually work with the following:

$$P(\theta|Y) \propto P(\theta) \times P(Y|\theta)$$

Our Bayesian posterior is therefore always a combination of the likelihood of the data, and the parameter priors

But for more complex models the distinction between what is 'data' and 'parameters' can get blurred!

MCMC

TODO

- Brief highlight of important points re convergence and effective sample size

Session 1a: Theory and application of MCMC

MCMC in Practice

We have seen that we can write a Metropolis algorithm but this is complex and inefficient

There are a number of general purpose langauages that allow us to define the problem in a faily intuitive manner and leave the details to the software. Among the latter are BUGS (Bayesian inference Using Gibbs Sampling) JAGS (Just another Gibbs Sampler) and STAN (named in honour of Stanislaw Ulam, pioneer of the Monte Carlo method).

JAGS

JAGS is a declarative (non-procedural) programming language. That is the order of statements does not matter. When implemented the interpreted just considers the appropriate line of code to interpret in terms of likelihood and prior. That is why you can only define each variable (LHS) once.

Different ways to run JAGS from R:

- rjags
- runjags
- R2jags
- jagsUI
- See http://runjags.sourceforge.net/quickjags.html (also in the GitHub folder)

A simple JAGS model might look like this:

```
model{
  # Likelihood part:
  Positives ~ dbinom(prevalence, TotalTests)

  # Prior part:
  prevalence ~ dbeta(2, 2)

  # Hooks for automatic integration with R:
  #data# Positives, TotalTests
  #monitor# prevalence
  #inits# prevalence
}
```

There are two model statements:

```
Positives ~ dbinom(prevalence, TotalTests)
```

states that the number of *Positive* test samples is Binomially distributed with probability parameter *prevalence* and total trials *TotalTests*

```
prevalence ~ dbeta(2,2)
```

states that our prior probability distribution for the parameter *prevalence* is Beta(2,2)

These are very similar to the likelihood and prior functions defined in the preparatory exercise

The other lines in this model are automated hooks to allow the runjags package to make sure that the correct R objects are passed to R as data, and that we get samples for the parameter we want to monitor. These statements are ignored by JAGS.

One of the advantages of using something like JAGS or STANis that much of the detail of the coding is shielded from us which means that the code is accessible to a wider variety of audiences. Running this model is much easier (and more efficient) than the Metropolis algorithm code.

To run this model, copy/paste the code above into a new text file called "basicjags.bug" in the same folder as your current working directory.

Then run:

```
library('runjags')
```

```
## Attaching runjags (version 2.0.4-6) and setting user-specified
↪  options
```

```
# data to be retrieved by runjags:
Positives <- 7
TotalTests <- 10

# initial values to be retrieved by runjags:
prevalence <- list(chain1=0.05, chain2=0.95)
```

```
results <- run.jags('basicjags.bug', n.chains=2, burnin=5000,
↪  sample=10000)
```

```
## Loading required namespace: rjags
```

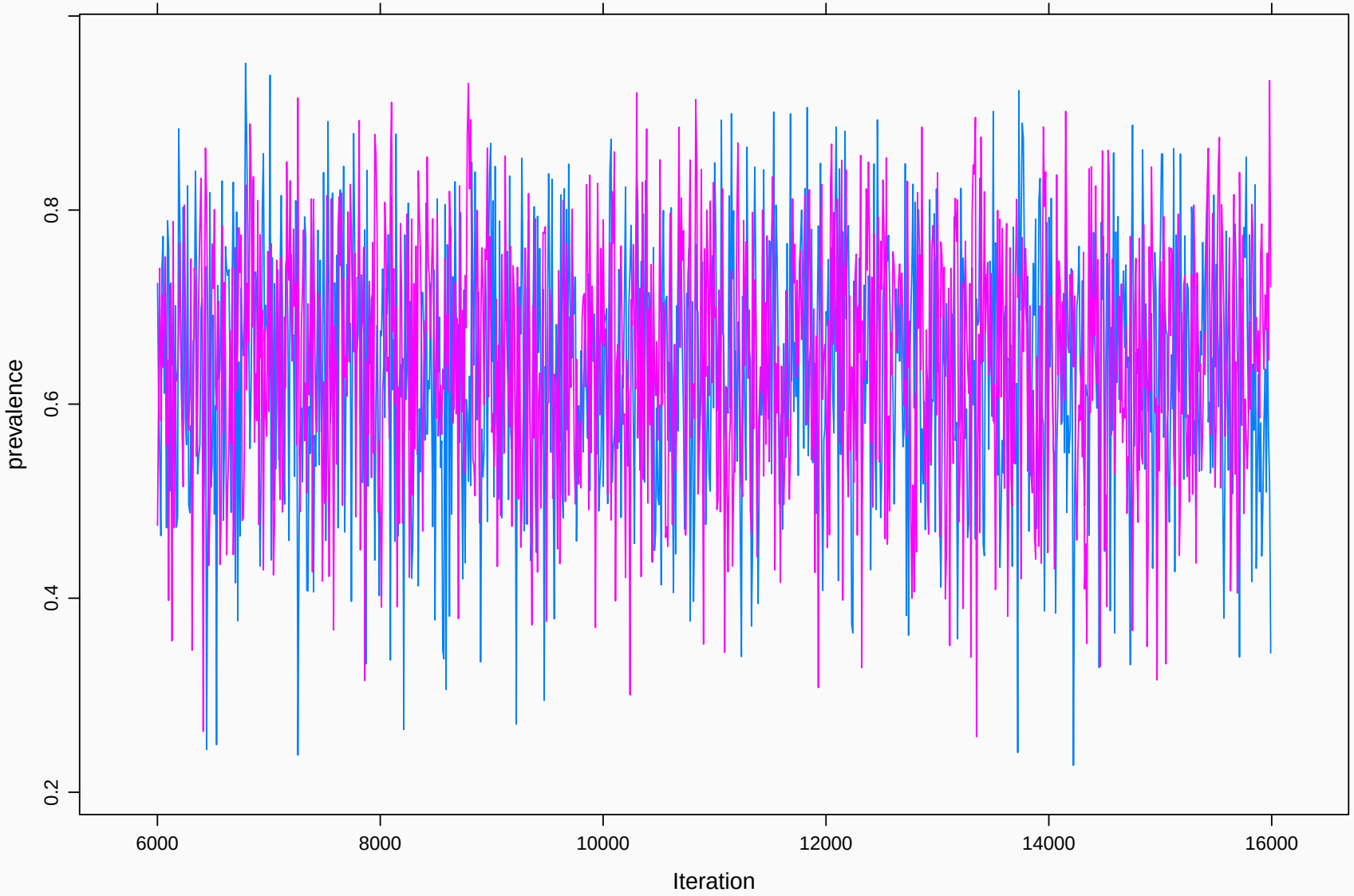
```
## Compiling rjags model...
## Calling the simulation using the rjags method...
## Adapting the model for 1000 iterations...
## Burning in the model for 5000 iterations...
## Running the model for 10000 iterations...
## Simulation complete
## Calculating summary statistics...
## Calculating the Gelman-Rubin statistic for 1
## variables....
## Finished running the simulation
```

You should see some updates then it will be finished

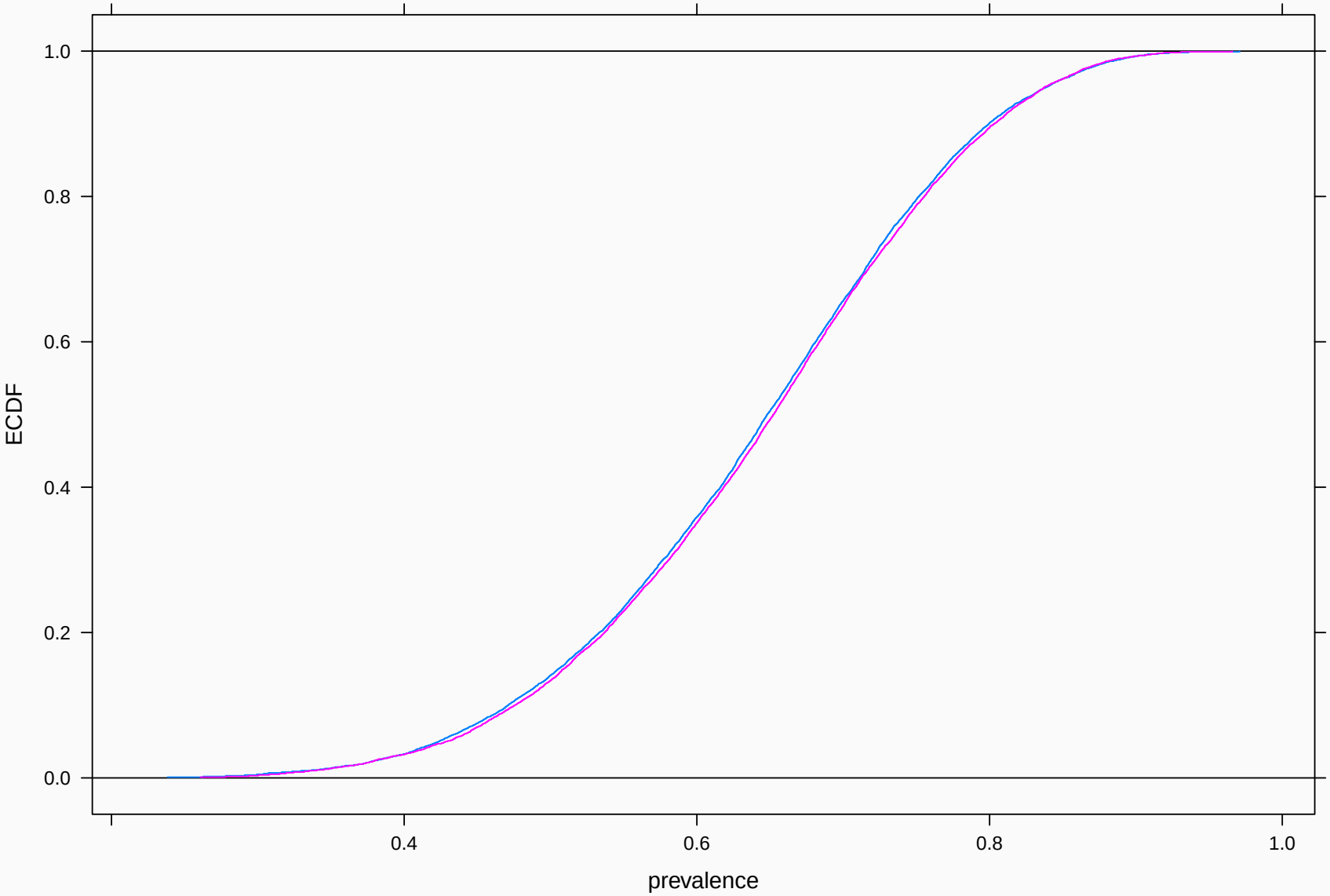
First check the plots for convergence:

```
plot(results)
```

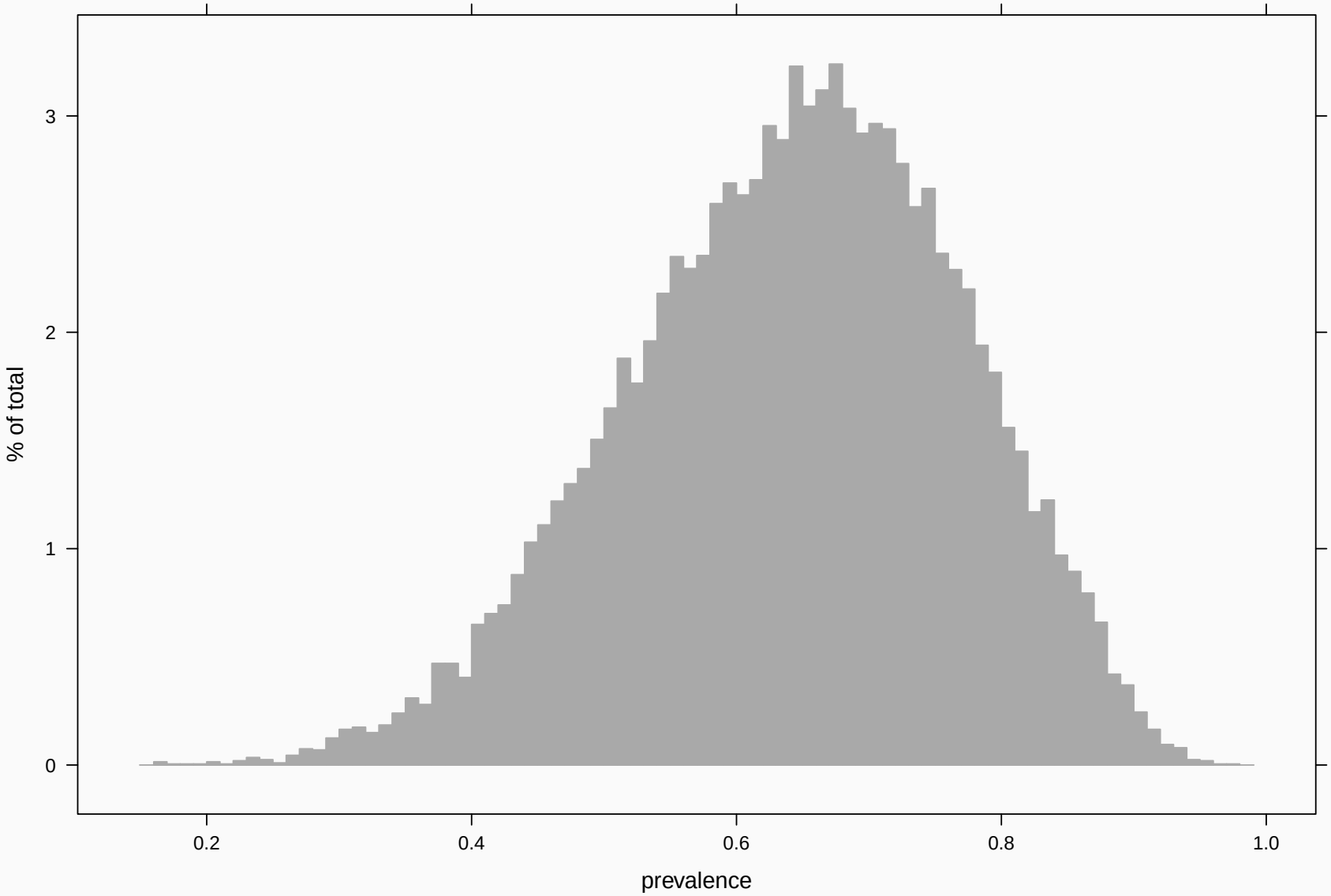
Trace plots: the two chains should be stationary:



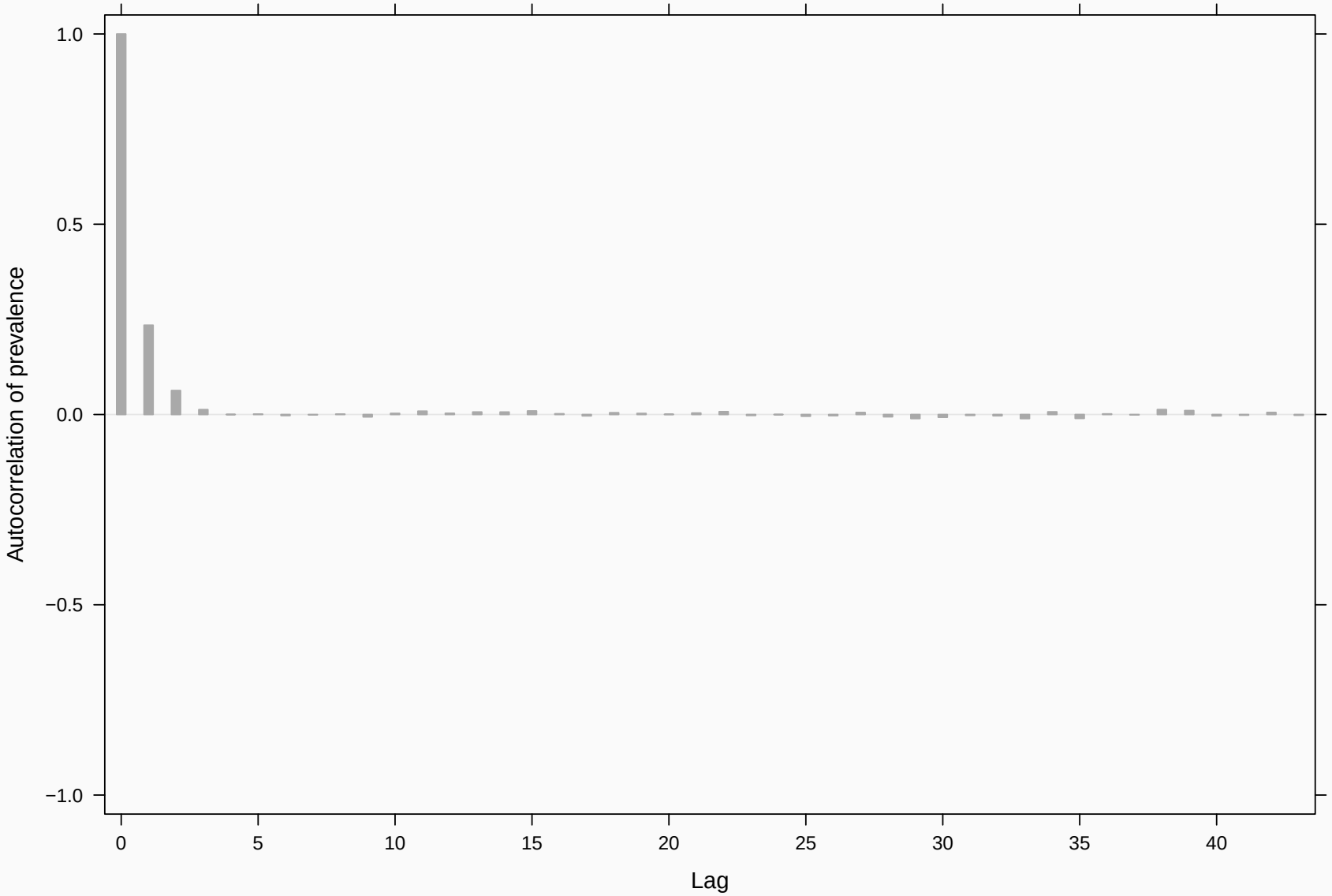
ECDF plots: the two chains should be very close to each other:



Histogram of the combined chains should appear smooth:



Autocorrelation plot tells you how well behaved the model is:



Then check the effective sample size (SSeff) and Gelman-Rubin statistic (psrf):

```
results
```

```
##
## JAGS model summary statistics from 20000 samples (chains = 2;
↪  adapt+burnin = 6000):
##
##            Lower95  Median Upper95    Mean      SD    Mode
## prevalence 0.40394 0.65022 0.87917 0.64246 0.12544 0.66631
##
##                MCerr MC%ofSD SSeff      AC.10   psrf
## prevalence 0.0011377     0.9 12157 0.0029035 1.0002
##
## Total time taken: 0.6 seconds
```

Reminder: we want psrf < 1.05 and SSeff > 1000. If these both look OK then you can use the posterior summary statistics.

Exercise

- Run this model yourself in JAGS
- Change the initial values for the two chains and make sure it doesn't affect the results
- Reduce the burnin length - does this make a difference?
- Change the sample length - does this make a difference?

Optional Exercise

- Change the number of chains to 1 and 4
  - Remember that you will also need to change the initial values
  - What affect does having different numbers of chains have?
  - Try using the `run.jags` argument `method='parallel'` - what affect does this have?

Session 1b: Working with basic models (apparent prevalence)

Other runjags options

There are a large number of other options to runjags. Some highlights:

- The method can be parallel or background or bgparallel
- You can use extend.jags to continue running an existing model (e.g. to increase the sample size)
- You can use coda::as.mcmc.list to extract the underlying MCMC chains
- Use the summary() method to extract summary statistics
    - See '?summary.runjagsand?runjagsclass' for more information

Using embedded character strings

- For simple models we might not want to bother with an external text file. Then we can do:

```
mt <- "
model{
  Positives ~ dbinom(prevalence, TotalTests)
  prevalence ~ dbeta(2, 2)

  #data# Positives, TotalTests
  #monitor# prevalence
  #inits# prevalence
}
"

results <- run.jags(mt, n.chains=2)
```

```
## Finished running the simulation
```

- But this makes the model harder to read, so I would generally

Setting the RNG seed

- If we want to get numerically replicable results we need to add .RNG.name and .RNG.seed to the initial values, and an additional #modules# lecuyer directive:

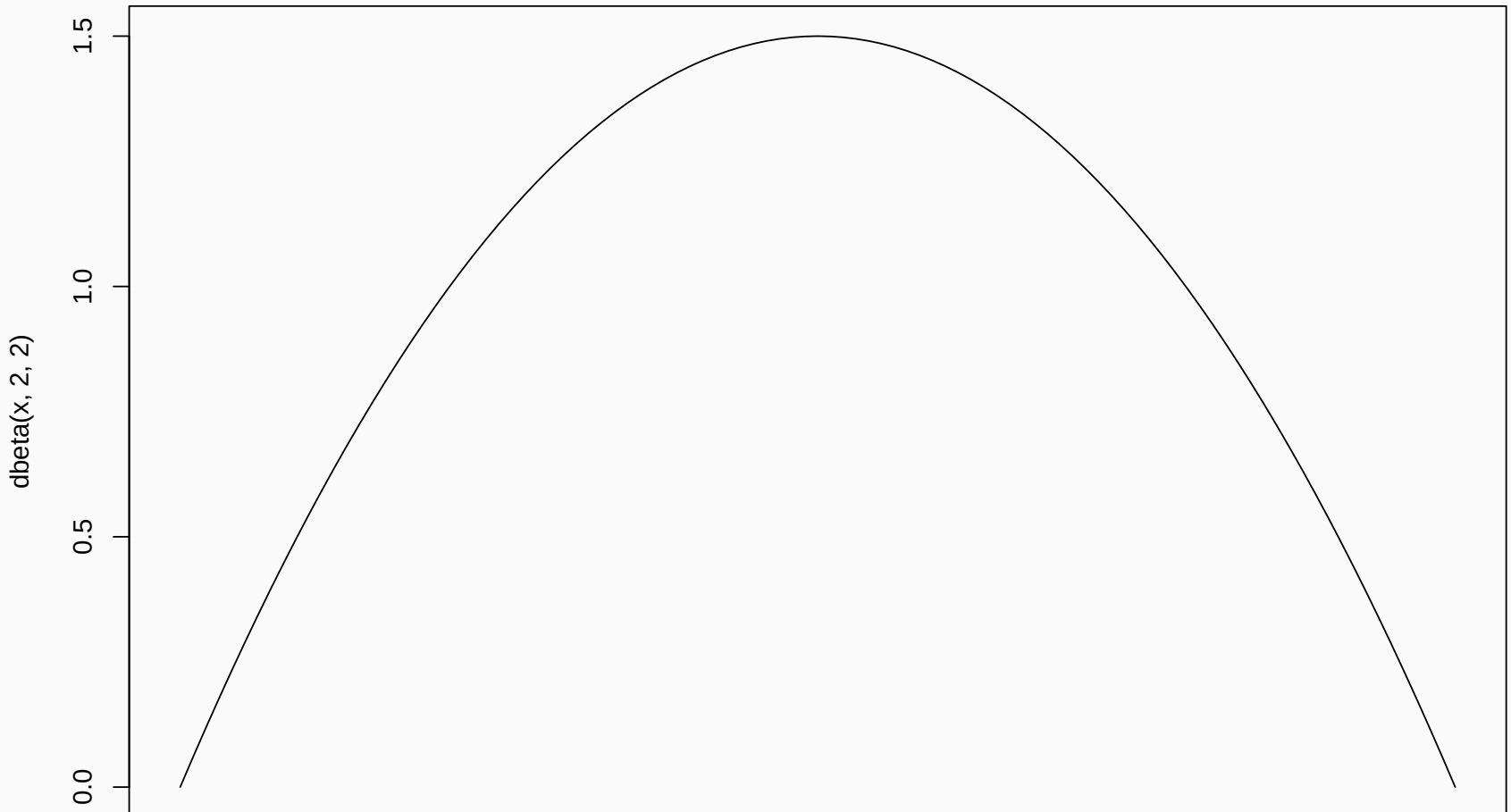
```
mt <- "
model{
  Positives ~ dbinom(prevalence, TotalTests)
  prevalence ~ dbeta(2, 2)

  #data# Positives, TotalTests
  #monitor# prevalence
  #inits# prevalence, .RNG.name, .RNG.seed
  #modules# lecuyer
}
"
.RNG.name <- "lecuyer::RngStream"
.RNG.seed <- list(chain1=1, chain2=2)
results <- run.jags(mt, n.chains=2)
```

A different prior

- A quick way to see what the distribution of a prior looks like is via e.g.:

```
1  curve(dbeta(x, 2, 2), from=0, to=1)
```



- Let's change the prior we are using to dbeta(1,1):

```
model{
  Positives ~ dbinom(prevalence, TotalTests)
  prevalence ~ dbeta(1, 1)

  # Hooks for automatic integration with R:
  #data# Status, TotalTests
  #monitor# prevalence
  #inits# prevalence
}
```

An Equivalent Model

- We could equivalently specify the model as:

```
model{
  # Likelihood part:
  for(i in 1:TotalTests){
    Status[i] ~ dbern(prevalence)
  }

  # Prior part:
  prevalence ~ dbeta(1, 1)

  # Hooks for automatic integration with R:
  #data# Status, TotalTests
  #monitor# prevalence
  #inits# prevalence
}
```

- But we need the data in a different format: a vector of 0/1 rather than total positives!

A GLM Model

- Another option is:

```
model{
  # Likelihood part:
  for(i in 1:TotalTests){
    Status[i] ~ dbern(predicted[i])
    logit(predicted[i]) <- intercept
  }

  # Prior part:
  intercept ~ dnorm(0, 10^-6)

  # Derived parameter:
  prevalence <- ilogit(intercept)

  # Hooks for automatic integration with R:
  #data# Status, TotalTests
  #monitor# intercept, prevalence
  #inits# intercept
}
```

- This is the start of a generalised linear model, where we could add covariates at individual animal level.
- We introduce a new distribution `dnorm()` - notice this is mean and precision, not mean and sd!
- For a complete list of the distributions available see:
  - https://sourceforge.net/projects/mcmc-jags/files/Manuals/4.x/
  - This document is also provided on the GitHub repository
- However, notice that the prior is specified differently...

Exercise

- Run the original version of the model and the GLM version of the model and compare results with the same data
- Now try a larger sample size: e.g. 70 positives out of 100 tests - are the posteriors from the two models more or less similar than before?
- Now try running the GLM model with a prior of `dnorm(0, 0.33)` (and the original data) - does this make a difference?

Optional Exercise

Another way of comparing different priors is to run different models with no data - as there is no influence of a likelihood, the posterior will then be identical to the priors (and the model will run faster).

One way to do this is to make all of the response data (i.e. either Positives or Status) missing. Try doing this for the following three models, and compare the priors for prevalence:

- The original model with prior `prevalence ~ dbeta(1,1)`
- The GLM model with prior `intercept ~ dnorm(0, 10^-6)`
- The GLM model with prior `intercept ~ dnorm(0, 0.33)`

Session 1c: Basics of latent-class models (imperfect test)

Imperfect tests

- Up to now we have ignored issues of diagnostic test sensitivity and specificity
- Usually, however, we do not have a perfect test, so we do not know how many are truly positive or truly negative, rather than just testing positive or negative.
- But we know that:

$$Prev_{obs} = (Prev_{true} \times Se) + ((1 - Prev_{true}) \times (1 - Sp))$$

$$\implies Prev_{true} = \frac{Prev_{obs} - (1 - Sp)}{Se - (1 - Sp)}$$

Model Specification

- We can write this model as:

```
model{
  Positives ~ dbinom(obsprev, TotalTests)
  obsprev <- (prevalence * se) + ((1-prevalence) * (1-sp))

  prevalence ~ dbeta(1, 1)
  se ~ dbeta(1, 1)
  sp ~ dbeta(1, 1)

  #data# Positives, TotalTests
  #monitor# prevalence, obsprev, se, sp
  #inits# prevalence, se, sp
}
```

- And run it:

```
prevalence <- list(chain1=0.05, chain2=0.95)
se <- list(chain1=0.5, chain2=0.99)
sp <- list(chain1=0.5, chain2=0.99)
Positives <- 70
TotalTests <- 100
results <- run.jags('basicimperfect.bug', n.chains=2, burnin=0,
↪  sample=10000)
```

```
## Finished running the simulation
```

[Remember to check convergence and effective sample size!]

	Lower95	Median	Upper95	SSeff	psrf
prevalence	0.058	0.512	0.994	1980	1.000
obsprev	0.605	0.695	0.781	20332	1.000
se	0.166	0.700	1.000	1559	1.001
sp	0.000	0.316	0.862	1582	1.000

- What do these results tell us?

	Lower95	Median	Upper95	SSeff	psrf
prevalence	0.058	0.512	0.994	1980	1.000
obsprev	0.605	0.695	0.781	20332	1.000
se	0.166	0.700	1.000	1559	1.001
sp	0.000	0.316	0.862	1582	1.000

- What do these results tell us?

- We can estimate the observed prevalence quite well
- But not the prevalence, se or sp! The model is unidentifiable.

Priors

- We cannot estimate se, sp and prevalence simultaneously
- We need strong priors for se and sp
- We can use the PriorGen package to generate Beta priors based on published results, for example:

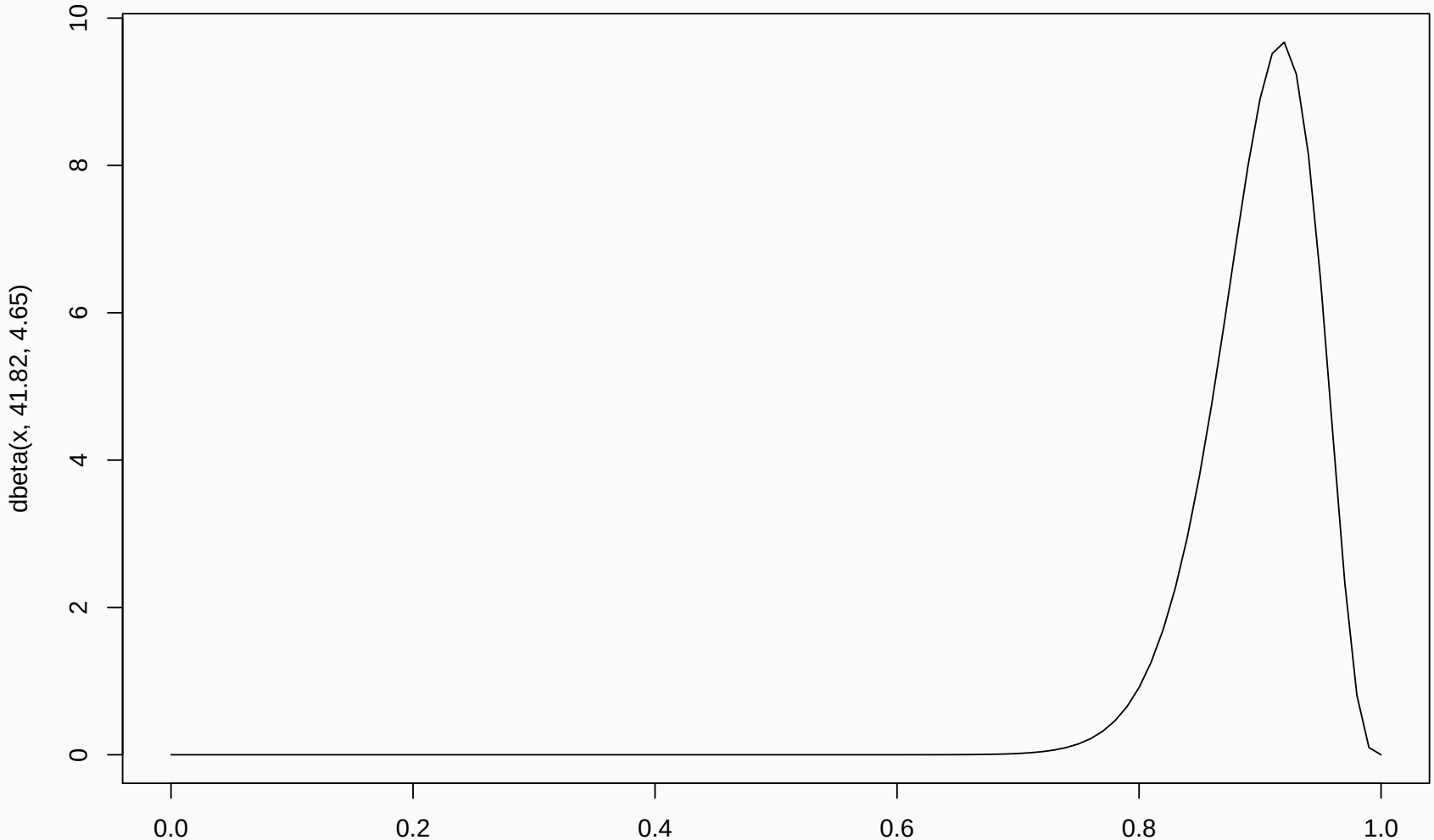
```
PriorGen::findbeta(themean=0.9, percentile = 0.975,
↪  percentile.value = 0.8)
```

```
## [1] "The desired Beta distribution that satisfies the
↪  specified conditions is: Beta( 41.82 4.65 )"
## [1] "Here is a plot of the specified distribution."
## [1] "Descriptive statistics for this distribution are:"
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##  0.6645  0.8754  0.9061  0.9008  0.9332  0.9932
## [1] "Verification: The percentile value 0.8 corresponds to the
↪  0.025 th percentile"
```

```r
qbeta(c(0.025, 0.5, 0.975), 41.82, 4.65)
```

```
## [1] 0.7999144 0.9056630 0.9677283
```

```r
curve(dbeta(x, 41.82, 4.65), from=0, to=1)
```



Exercise

- Find beta distribution priors for sensitivity = 0.9 (95% CI: 0.85 - 0.95) and specificity = 0.95 (95%CI: 0.92-0.97)
- Look at these distributions using curve and qbeta
- Modify the imperfect test model using these priors and estimate prevalence now

Optional Exercise

Run the same model with se and sp fixed to the mean estimate - how does this affect CI for prevalence?

Run the same model with se and sp fixed to 1 - how does this affect estimates and CI for prevalence?

Solution

```
PriorGen::findbeta(themean=0.9, percentile = 0.975,
↪  percentile.value = 0.85)
```

```
## [1] "The desired Beta distribution that satisfies the
↪  specified conditions is: Beta( 148.43 16.49 )"
## [1] "Here is a plot of the specified distribution."
## [1] "Descriptive statistics for this distribution are:"
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##  0.7962  0.8855  0.9018  0.9002  0.9167  0.9719
## [1] "Verification: The percentile value 0.85 corresponds to
↪  the 0.025 th percentile"
```

```
qbeta(c(0.025, 0.5, 0.975), 148.43, 16.49)
```

```
## [1] 0.8500135 0.9016287 0.9408480
```

```
curve(dbeta(x, 148.43, 16.49), from=0, to=1)
```

Summary

TODO