Stats and JAGS

Sonja Hartnack Valerie Hungerbühler

With some inspiration from Matt Denwood & Giles Innocent from a previous training

outline

- Probability distributions
- Example Binomial
- Maximising a likelihood
- Profiling a likelihood
- Bayesian statistics
- Profiling a posterior
- Summarising a posterior
- Markov chain Monte Carlo
- Metropolis Hastings
- Effective sample size
- Set seed
- JAGS
- Binomial model in JAGS
- Plots for checking convergence

Probability distributions

- Likelihood theory is at the heart of most inferential statistics - it describes the chances of an event happening under certain assumptions, such as the probability distribution and the parameter value(s) that we want to use with that distribution.
- Put simply, a likelihood is the probability of observing our data given the distribution that we use to describe the data generating process.
- For example, what is the likelihood (i.e. probability) of getting 5 heads from 10 tosses of a fair coin?

Probability distributions

- Probability theory can help us decide - we can assume:
  - The probability distribution describing a number of independent coin tosses is called the Binomial distribution
  - In this case, we would use the parameters:
    - Number of coin tosses $= 10$
    - Probability of a head $=$ 'fair' $= 0.5$

Wikipedia tells us that we can calculate the probability of a Binomial distribution as follows:

```
tosses <- 10
probability <- 0.5
heads <- 5
lik.hood_1 <- choose(tosses, heads) * probability^heads *
                (1-probability)^(tosses-heads)
lik.hood_1
```

```
## [1] 0.2460938
```

But R makes our life easier by implementing this using a function called dbinom:

```
lik.hood_2 <- dbinom(heads, tosses, probability)
lik.hood_2
```

```
## [1] 0.2460938
```

These two numbers are the same, which is reassuring.

The most common probability distributions are

- Discrete probability distributions
  - Binomial: dbinom
  - Poisson: dpois
  - Negative binomial: dnbinom
  - Multinomial: dmultinom
- Continuous probability distributions
  - Normal: dnorm
  - Exponential: dexp
  - Gamma: dgamma
  - Beta: dbeta
  - Lognormal: dlnorm
  - Uniform: dunif

Maximising a likelihood

- In the previous example we assumed that we knew the probability of getting a head because the coin was fair (i.e. probability of head $= 50\%$), but typically we would want to estimate this parameter based on the data. One way to do this is via Maximum Likelihood.

Maximising a likelihood

- Let's say that we have observed 7 test positive results from 10 individuals and we want to estimate the prevalence by maximum likelihood. We could do that by defining a function that takes our parameter (prev) as an argument, then calculates the likelihood of the data based on this parameter:

```
likelihood_fun <- function(prev) dbinom(7, 10, prev)
```

Maximising a likelihood

We can now ask the function what the likelihood is for any parameter value that we choose, for example:

```
likelihood_fun(0.8)
```

```
## [1] 0.2013266
```

```
likelihood_fun(0.5)
```

```
## [1] 0.1171875
```

So the data are more likely if the prevalence parameter is 0.8 than if it is 0.5. We could keep doing this for lots of different parameter values until we find the highest likelihood, but it is faster and more robust to use an R function called optimise to do this for us:

Maximising a likelihood

```
optimise(likelihood_fun, interval=c(0, 1), maximum=TRUE)
```

```
## $maximum
## [1] 0.6999843
##
## $objective
## [1] 0.2668279
```

This tells us that the maximum likelihood for this data is 0.267, which corresponds to a parameter value of around 0.7 (or a prevalence of 70%). This is the maximum likelihood.

Maximising a likelihood

We could have also got the same using:

```
model <- glm(cbind(7,10-7) ~  1, family = binomial)
plogis(coef(model))
```

```
## (Intercept)
##         0.7
```

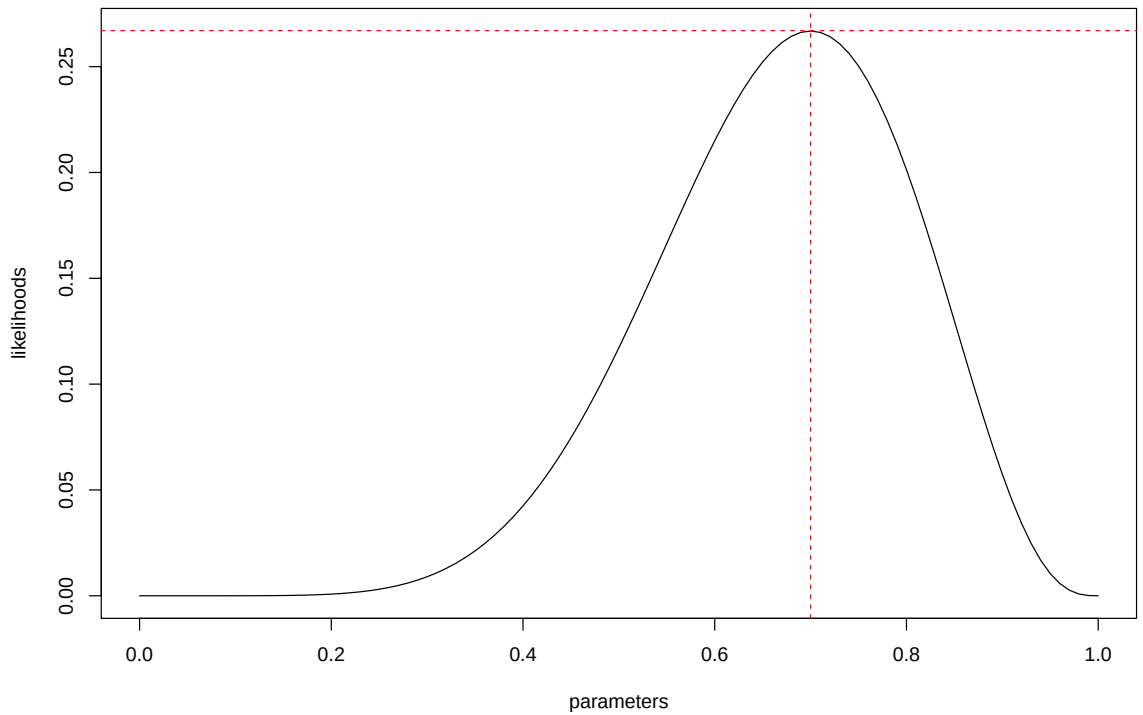
This is really what is happening when you use a (generalised) linear model in R - it optimises the parameter values to give the highest likelihood for the data and model (distribution with predictors) that you have specified.

Profiling a likelihood

The parameters corresponding to the maximum likelihood give the highest probability of observing the data given the parameters, but there are other parameter values under which we could observe the data with almost as high a probability. It is useful to look at the range of parameter values that are consistent with the data, which is why R reports standard errors (and/or confidence intervals) when you run a model.

Profiling a likelihood

But we can also look at the full distribution of the likelihood of the data over a range of parameter values using our function above. The red dashed lines show the maximum likelihood (y axis) with corresponding parameter value (x axis), and the solid line is the likelihood of the data given the parameter value on the x axis. You can see that parameter values near 0.7 have a likelihood that is almost as high as the maximum.



Bayesian statistics

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

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

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.

Bayesian statistics

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:

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

For frequentist statistics we only had the likelihood, but Bayesian statistics allows us to combine this likelihood with a prior to obtain a posterior. There are a lot of advantages of working with a posterior rather than a likelihood, because it allows us to make much more direct (and useful) inference about the probability of our parameters given the data. The cost of working with the posterior is that we must also define a prior, and accept that our posterior is affected by prior that we choose.

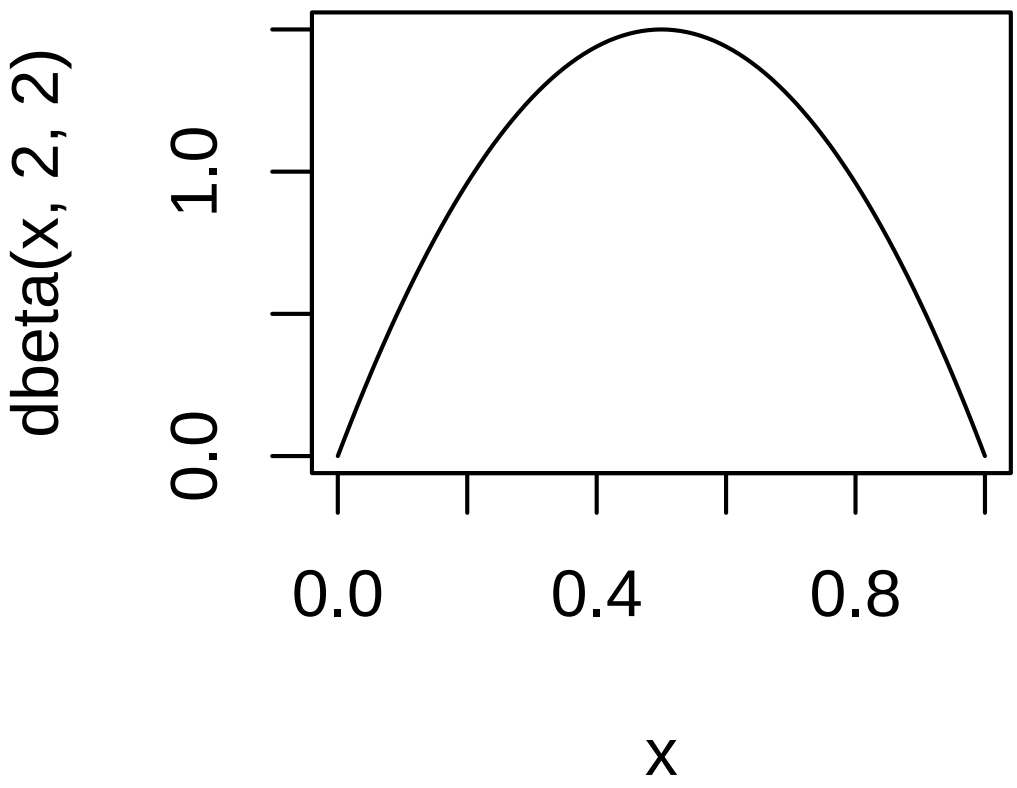
Profiling a Posterior

We can profile a posterior in the same way that we profiled the likelihood before. But first we need to define a prior - let's do this using a function again:

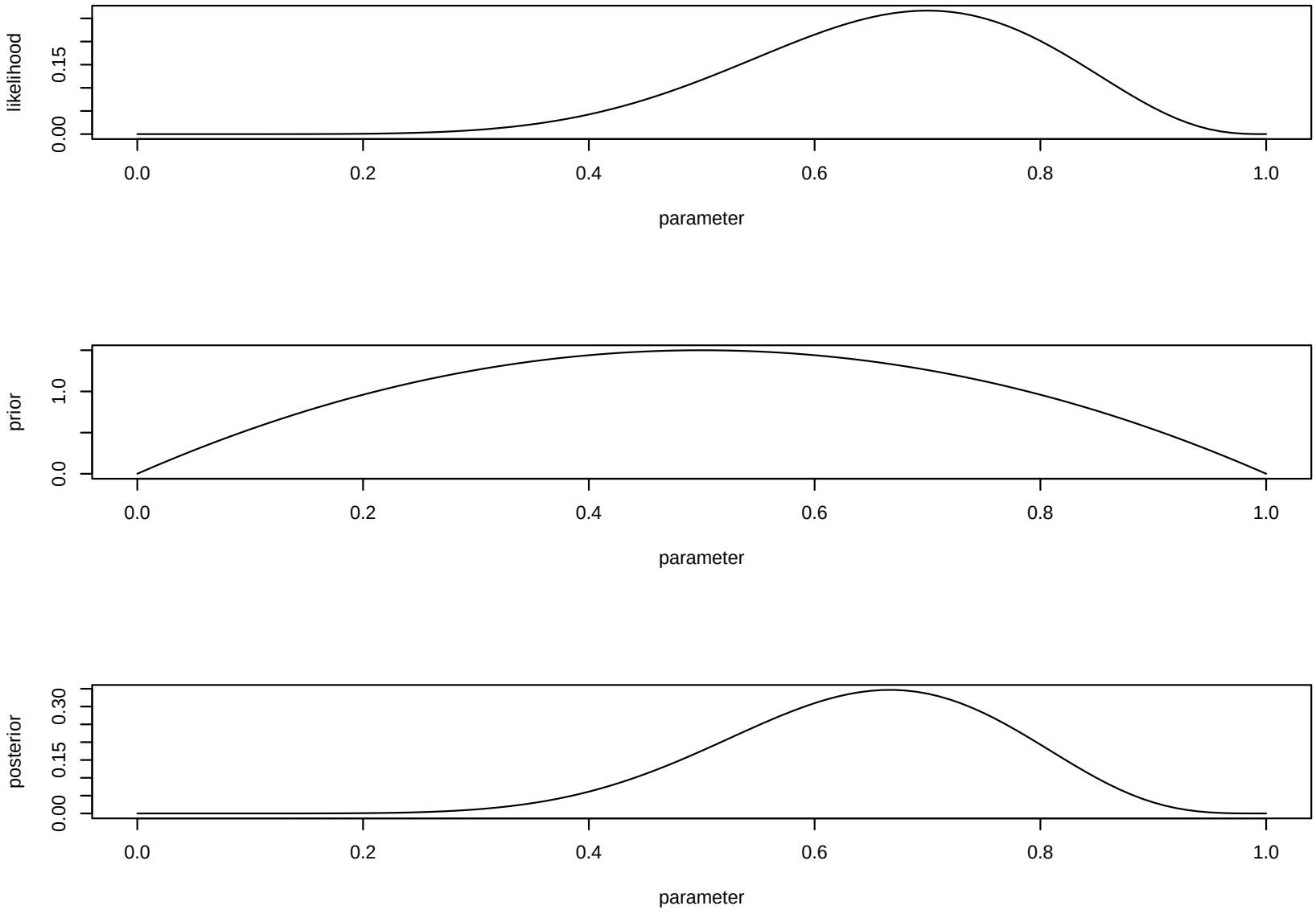
```
prior_fun <- function(prev) dbeta(prev, 2, 2)
```

This $Beta(2, 2)$ prior implies that before we observed the data, we believed that any prevalence value was most likely to be close to 0.5 but any value between 0 and 1 was quite possible.

- A quick way to see the distribution of a prior $Beta(2, 2)$:



Let's repeat the profiling exercise from before but this time consider the prior and posterior as well as the likelihood:



- The shape of the posterior is determined by that of both the likelihood and the prior - as we would expect given that it is simply a combination of the two. Notice also that the posterior will always be more precise (i.e. will have a smaller variance) than both the prior and the likelihood, as it is effectively combining the information available from the priors and the data.
- Try changing the parameters of the Beta distribution used for the prior (for example $Beta(1, 1)$ or $Beta(5, 5)$ or $Beta(5, 2)$) and see what happens to these graphs.

Summarising the Posterior

We will typically want to summarise the posterior distribution. In frequentist statistics we would always report the maximum likelihood value, i.e. the parameter corresponding to the highest likelihood (the point estimate). The equivalent to this in Bayesian statistics is the mode of the posterior, but we often report the median or mean of the posterior instead (the choice between these is somewhat arbitrary). It is also important to report an interval, which represents the degree of uncertainty in the posterior, the credibility or probability interval.

For our prevalence example (with 7/10 positive and a $Beta(2, 2)$ prior), these estimates would be:

```r
a <- 2+7
b <- 2+3
mo <- (a-1)/(a+b-2)
mn <- (a)/(a+b)
md <- qbeta(0.5,a,b)
ci <- TeachingDemos::hpd(qbeta, shape1=a, shape2=b)
```

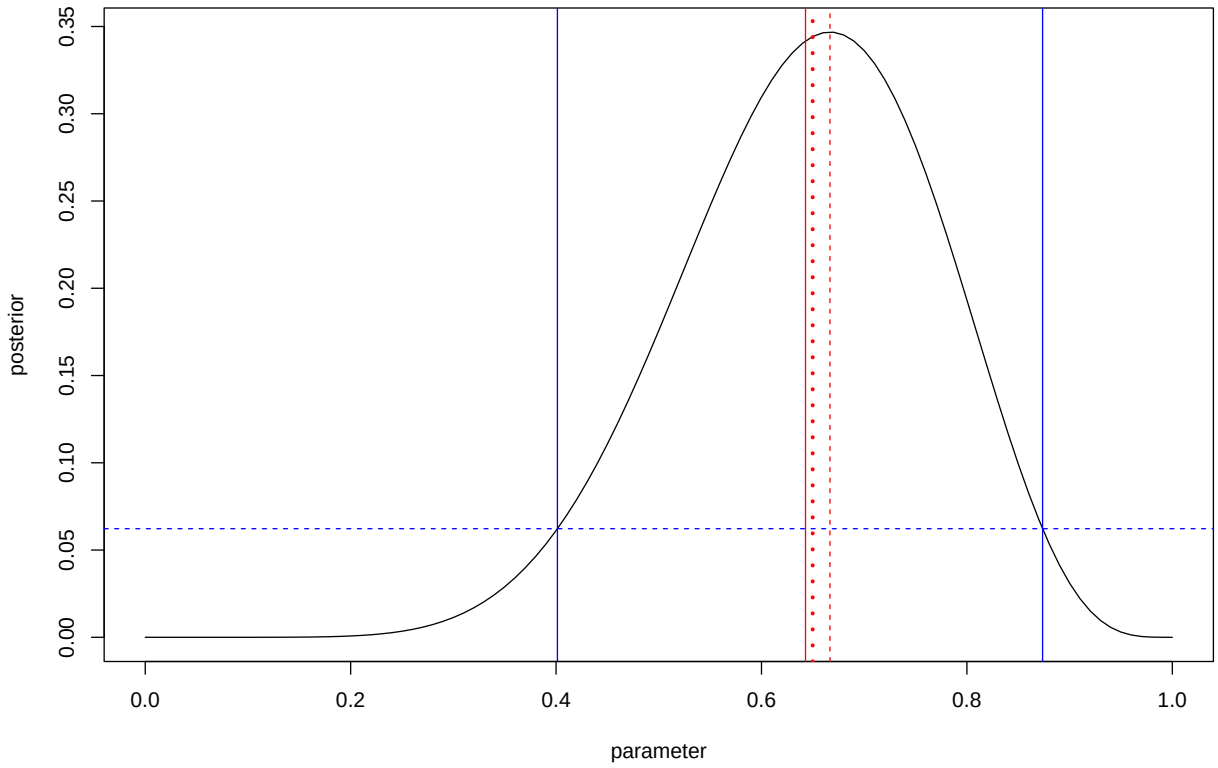
```
## Mode: 0.6666667
```

```
## Mean: 0.6428571
```

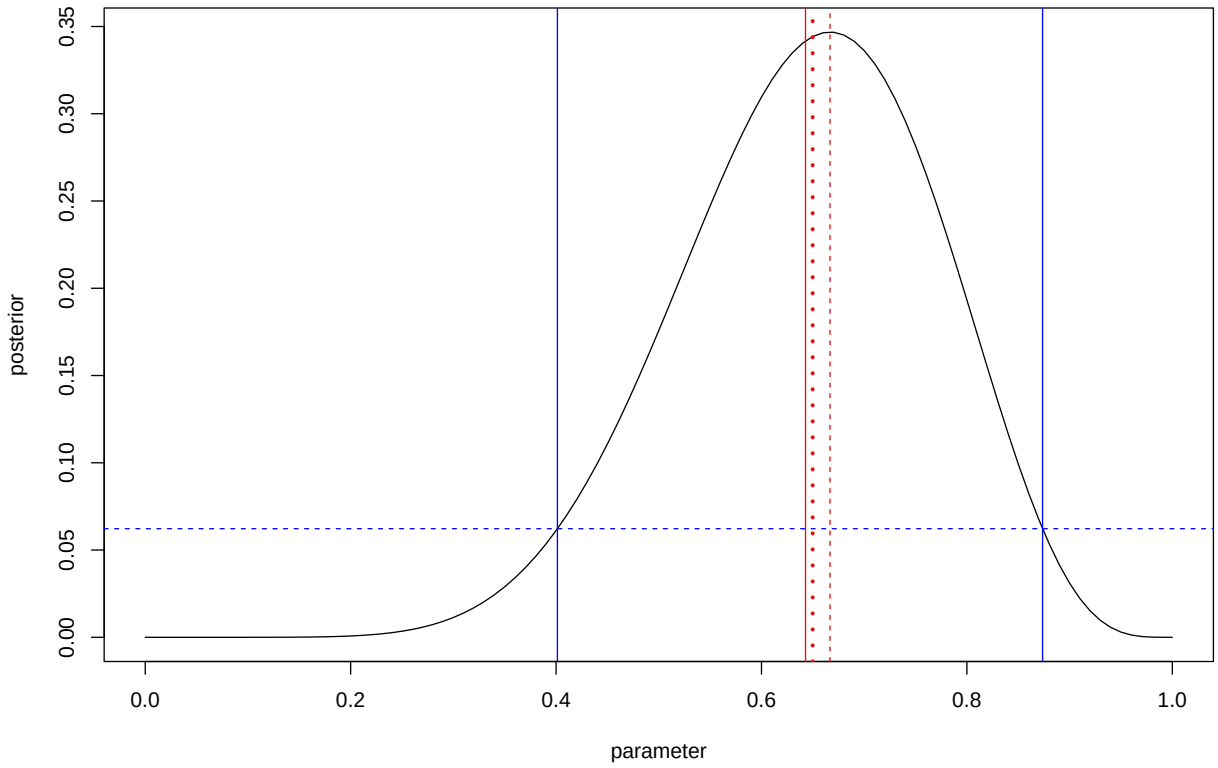
```
## Median: 0.6498366
```

```
## 95% CI: 0.401307 - 0.8736879
```

Don't worry for now about exactly how these are calculated. But we can overlay the estimates onto the plot of the posterior to help us interpret them:



The mean (red solid), median (red dotted) and mode (red dashed) all reflect the "best guess" for our parameter value - typically either the mean or median would be reported. They are usually all quite close to each other - although for a more skewed distribution there will be a bigger difference between them. The number can be interpreted as "my best guess for the prevalence given the data and priors is around 65%".



The blue lines reflect our 95% credible interval calculated using a highest posterior density interval. This interval is used in Bayesian statistics, and ensures that the posterior probability at the lower 95% CI is exactly the same as the posterior probability at the upper 95% CI - i.e. the blue dashed line (intersecting the posterior and 95% CI) is horizontal. This can be interpreted as "I am 95% sure that the prevalence is somewhere between 40% and 87%".

Markov chain Monte Carlo

For simple models like the one we have been working with, we can easily profile the posterior to get the results we need. But we will usually be working with more complicated models with lots of parameters. We could still profile the posterior, but rather than doing this for 101 values of a single parameter, we would have to do it for every combination of possible values for a large number of parameters. This means that the computational complexity of the profiling increases exponentially - for example for 5 parameters (a relatively simple model) we would need to evaluate 101^5 (or 10510100501) combinations of parameter values. This is known as the "curse of dimensionality".

Markov chain Monte Carlo

In order to solve this problem we need a new approach. Rather than evaluating all possible combinations of parameter values, we can sample the most relevant parameter value combinations by focussing on the areas of parameter space close to parameter values that we already know have a high posterior probability. This solves the curse of dimensionality but does introduce a couple of new problems. We can illustrate the approach for our previous example using a very simple form of Markov chain Monte Carlo (MCMC) called a Metropolis algorithm:

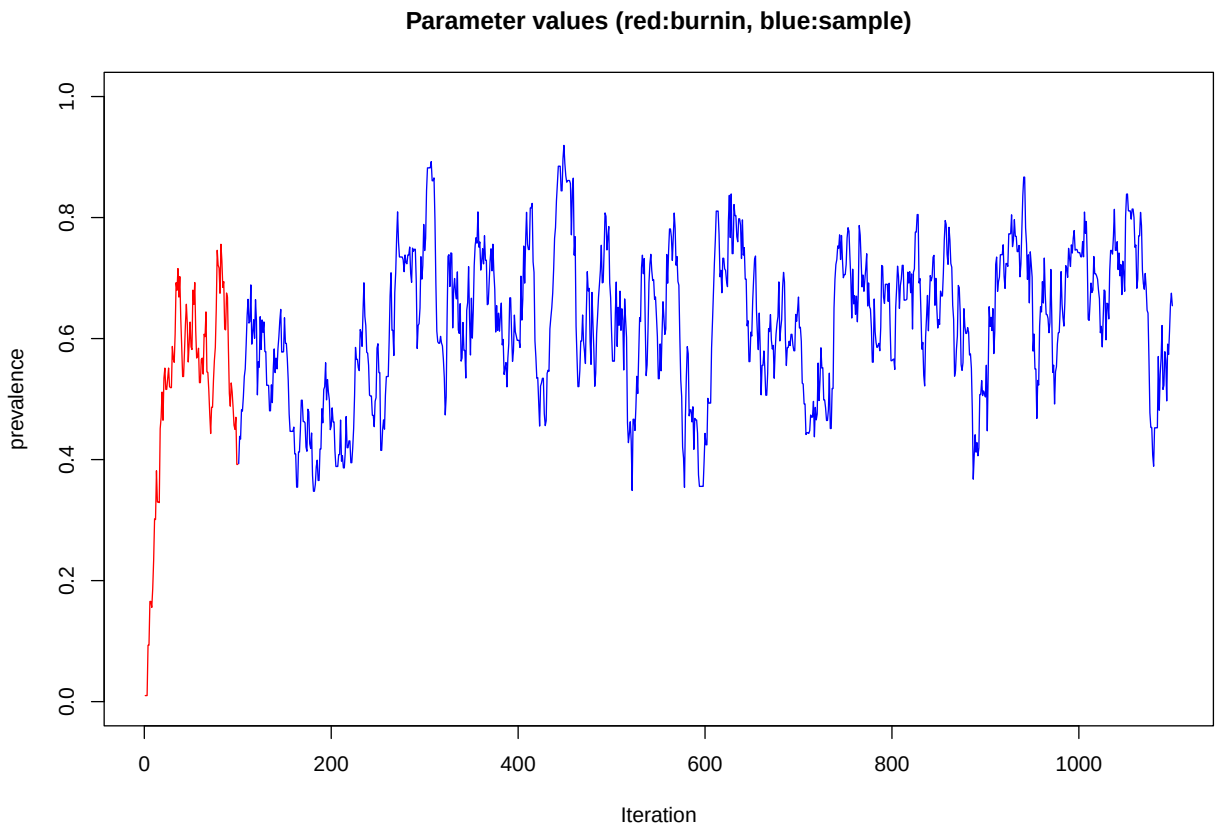
Metropolis Hastings

So what is happening in this process? The Metropolis algorithm is sampling values from the posterior distribution, and the basic idea is that it samples parameter values with frequency proportional to their posterior probability. It does this by sampling parameter values with a kind of random walk across the parameter space, using a normal distribution with pre-specified standard deviation (sigma) as the proposal distribution. The clever bit comes in the accept/reject step based on the ratio of the new posterior to the old posterior probability. Have a quick look through the R code (.Rmd file), but don't worry too much about the details.

Metropolis Hastings

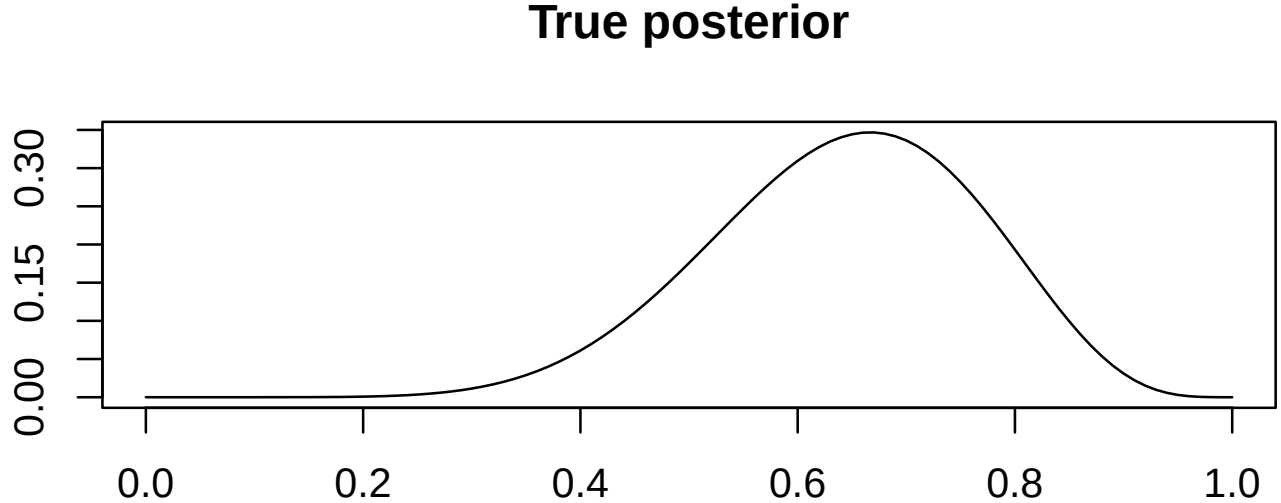
The important part is that we can run this function and obtain results as follows:

```
samples <- metropolis(burnin = 100, sample = 1000,
                      initial_value=0.01)
```



Metropolis Hastings

The algorithm starts an initial value of prevalence=0.01 at iteration 0. At each subsequent iteration (x-axis), a new value of prevalence is chosen based its the posterior probability, although the new value depends to some extent on the value at the last iteration, which you can see on the trace plot as autocorrelation (meaning that consecutive samples are not completely independent). After 1000 iterations, we have a sample of 1000 values from something that we hope is close to the true posterior. We can check this by comparing the profiled posterior to a density plot of the sampled values:



They are similar, but not the same. We can also look at estimates of the mean/median/95% CI based on the samples

```
## [1] 0.6273554
```

```
## [1] 0.632563
```

```
##                  lower     upper
## prevalence 0.3862691 0.8355892
## attr(,"Probability")
## [1] 0.95
```

Effective sample size

Another reason that our samples differed to the true posterior is because the samples are random, and therefore there is some sampling noise. In order to eliminate this we need to make sure we have a sufficient number of samples so that the difference between e.g. the mean of the sample and the true mean of the distribution underlying this sample becomes negligible (or at least small enough for our needs).

Effective sample size

In the previous example we had 1000 iterations, but this does not equal 1000 independent samples because of the autocorrelation. Instead we need to look at the effective sample size:

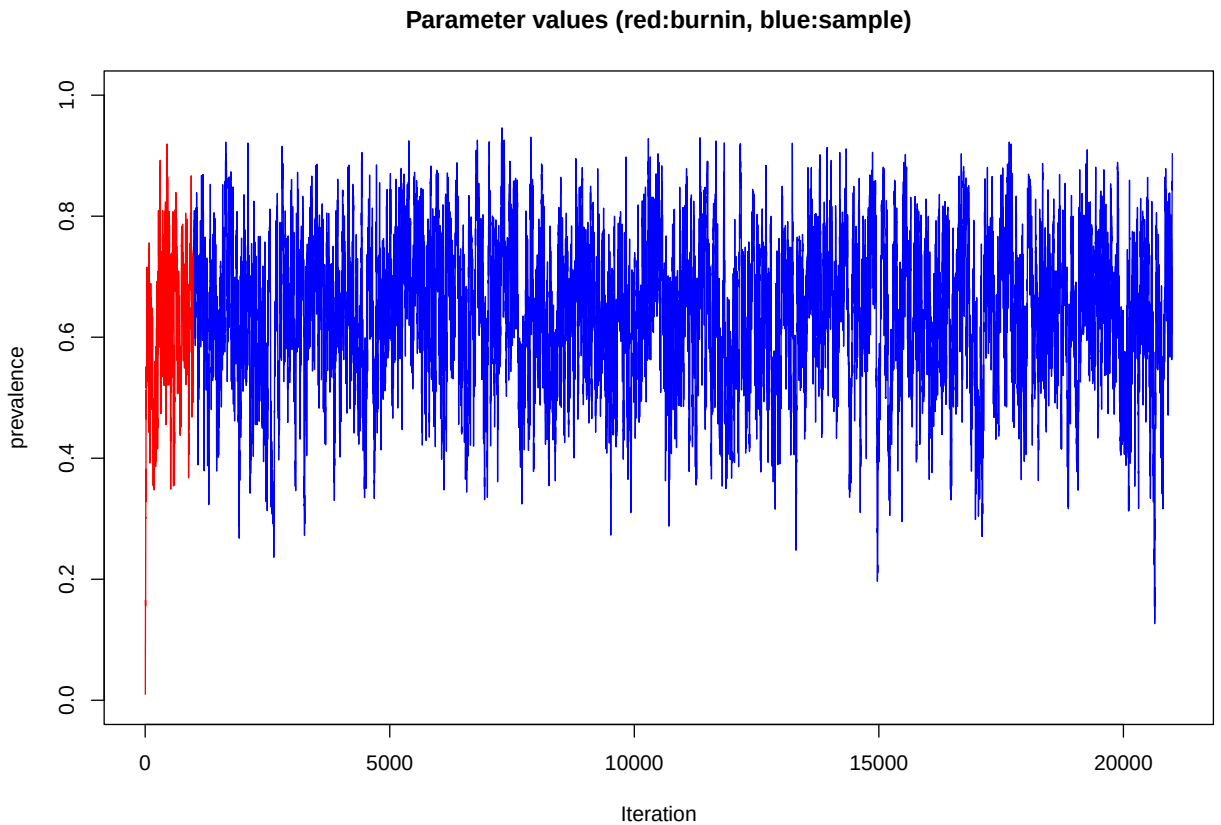
```
## prevalence
##   39.18648
```

So we have the equivalent of approximately 40 independent samples from this posterior, which is not nearly enough to be a good approximation.

Effective sample size

Ideally we would like at least 500 (or even 1000) independent samples, which means running the simulation for a lot more than 1000 iterations. For example:

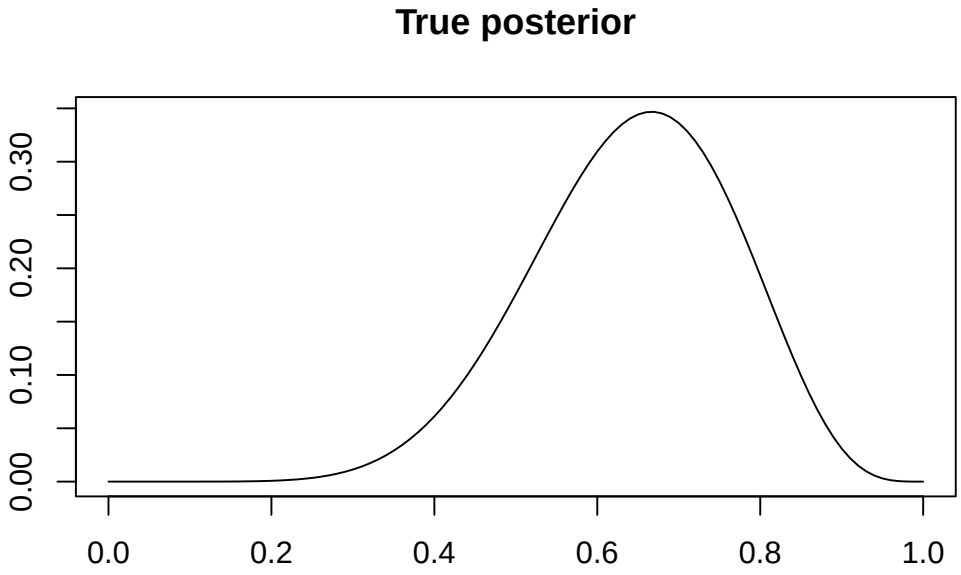
```
samples <- metropolis(burnin = 1000, sample = 20000,
                      initial_value=0.01)
```

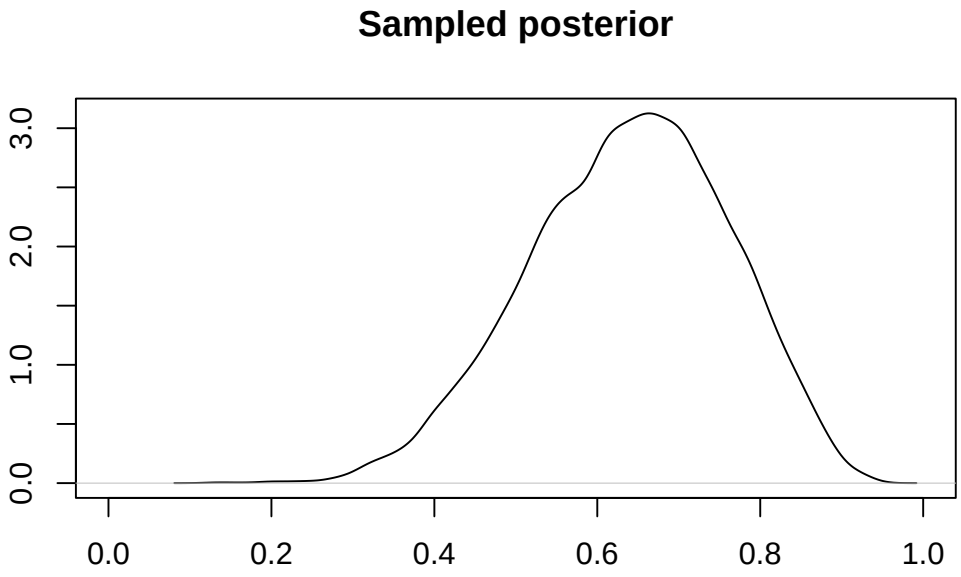


Effective sample size

```
## prevalence
##   670.5198
```

We now have 20000 iterations, which equates to around 670 independent samples (i.e. we need around 30 iterations to get 1 independent sample). This should be enough to get a decent approximation to the true posterior:


True posterior
0.00
0.10
0.20
0.30
0.0
0.2
0.4
0.6
0.8
1.0


Sampled posterior
0.0
1.0
2.0
3.0
0.0
0.2
0.4
0.6
0.8
1.0

Set seed

Among different samples there are potentially differences due to the inherent randomness of the Monte Carlo integration. If we want to ensure we get precisely the same result then we need to set the random number generator seed in R using the set seed function before running the simulation:

```
set.seed(2021-07-14)
samples1 <- metropolis(burnin = 1000, sample = 20000,
                       initial_value=0.99, plot=FALSE)
mean(samples1)
```

```
## [1] 0.6336825
```

```
set.seed(2022-07-14)
samples2 <- metropolis(burnin = 1000, sample = 20000,
                       initial_value=0.99, plot=FALSE)
mean(samples2)
```

```
## [1] 0.6433457
```

Conclusions

In the real world we would very rarely use a Metropolis algorithm, but it is a useful exercise to understand the basic concepts. MCMC as used by JAGS/BUGS is like an extension of the same principle. Crucially, you ALWAYS need to make sure of the following two key points before trusting any estimates made from your sampled posteriors:

1 - Make sure that the chain(s) have converged, and that a sufficient burn-in period has been run before starting to sample. You should look at the potential scale reduction factor (psrf) of the Gelman-Rubin statistic, and at trace plots (more than 1 independent chain) for each parameter to make sure the chains have converged on the stationary posterior. This is extremely important, and can be tricky.

2 - Make sure that you have a sufficient effective sample size. Check the effective sample size (this should be produced by whatever software you are using) to make sure it is over at least 500 (and preferably 1000) for all parameters of interest.

JAGS

- JAGS uses the BUGS language
  - This is a declarative (non-procedural) language
  - The order of statements does not matter
  - The compiler converts our model syntax into an MCMC algorithm with appropriately defined likelihood and prior
  - You can only define each variable once!!!

JAGS

- JAGS uses the BUGS language
  - This is a declarative (non-procedural) language
  - The order of statements does not matter
  - The compiler converts our model syntax into an MCMC algorithm with appropriately defined likelihood and prior
  - You can only define each variable once!!!
- Different ways to run JAGS from R:
  - rjags, runjags, R2jags, jagsUI
- See http://runjags.sourceforge.net/quickjags.html

A simple JAGS model might look like this:

```
model_definition <- "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
}
"
cat(model_definition, file='basicjags.bug')
```

There are two model statements:

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

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

There are two model statements:

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

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

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

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

The other lines in this model:

```
#data# Positives, TotalTests
#monitor# prevalence
#inits# prevalence
```

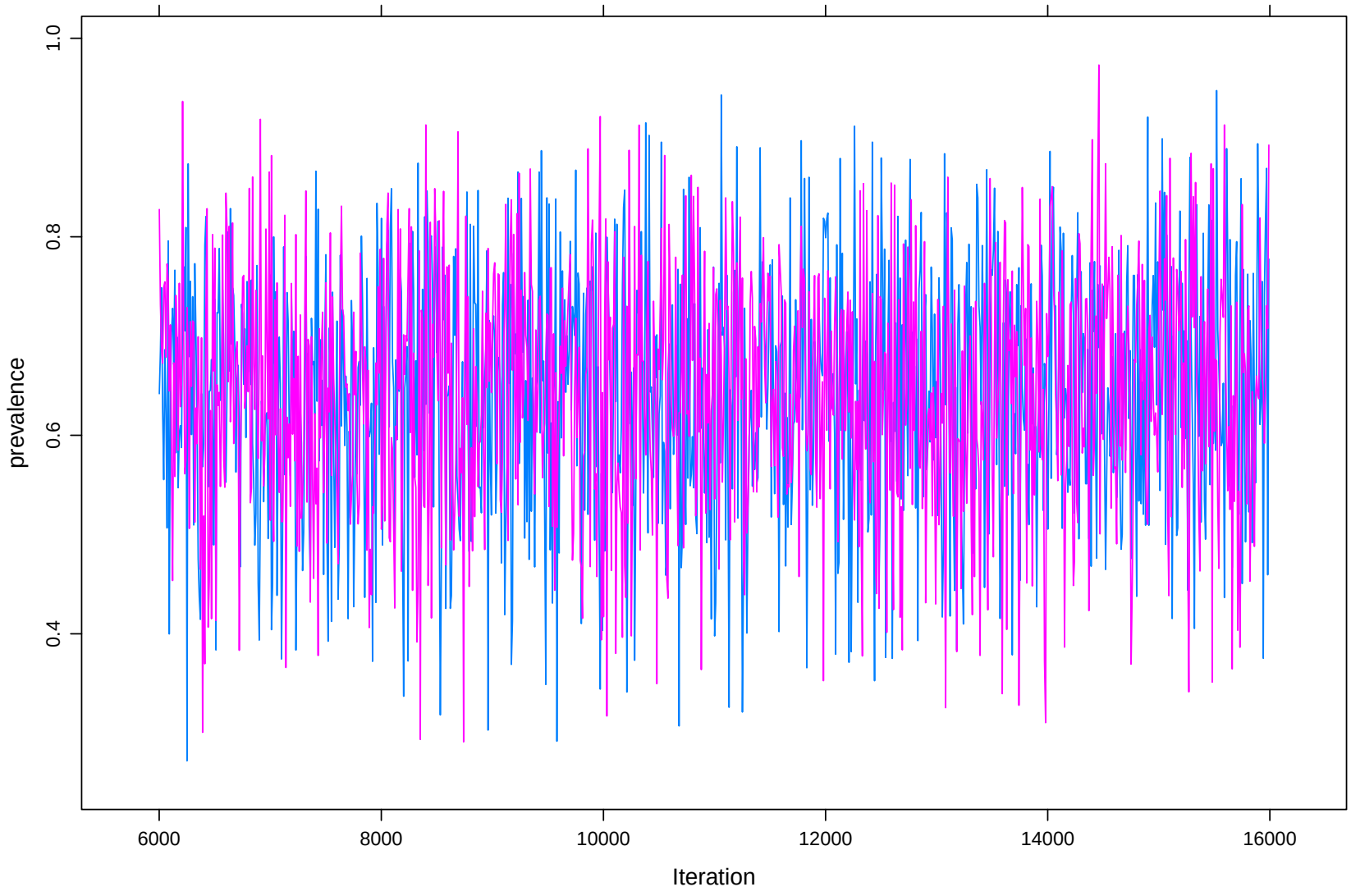
are automated hooks that are only used by runjags.

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:

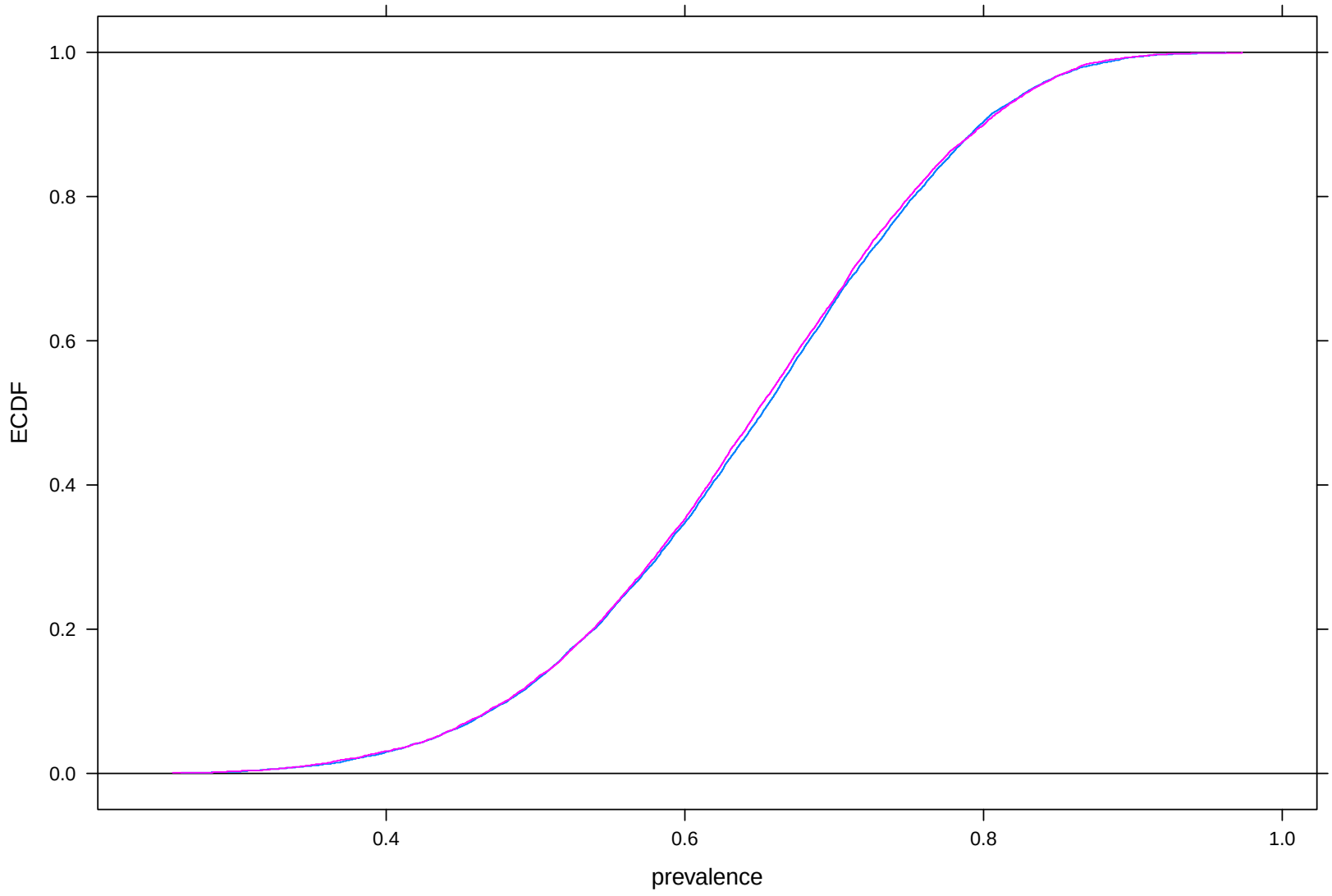
```
library('runjags')

# 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)
```

First check the plots for convergence:

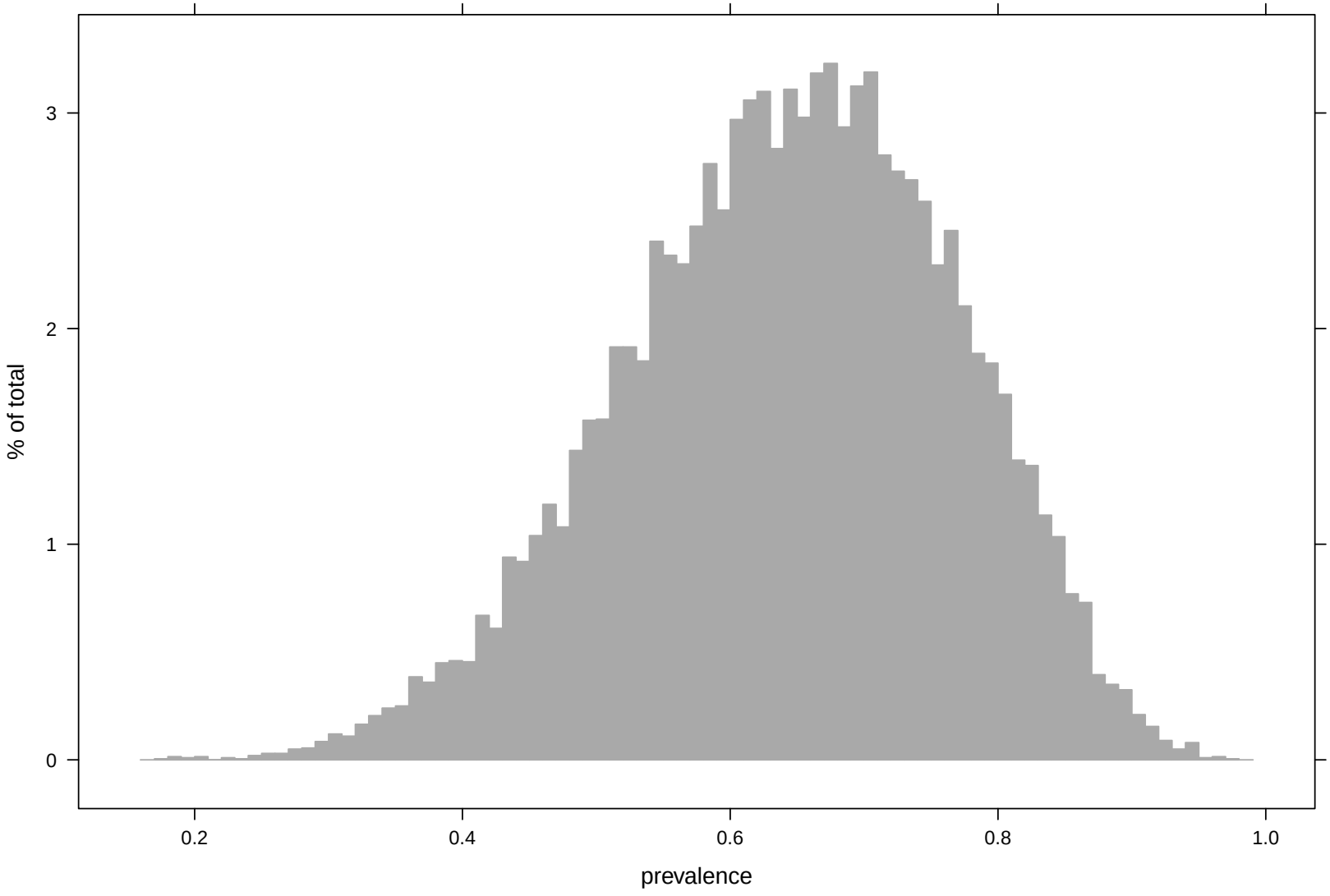
Trace plots: the two chains should be stationary:



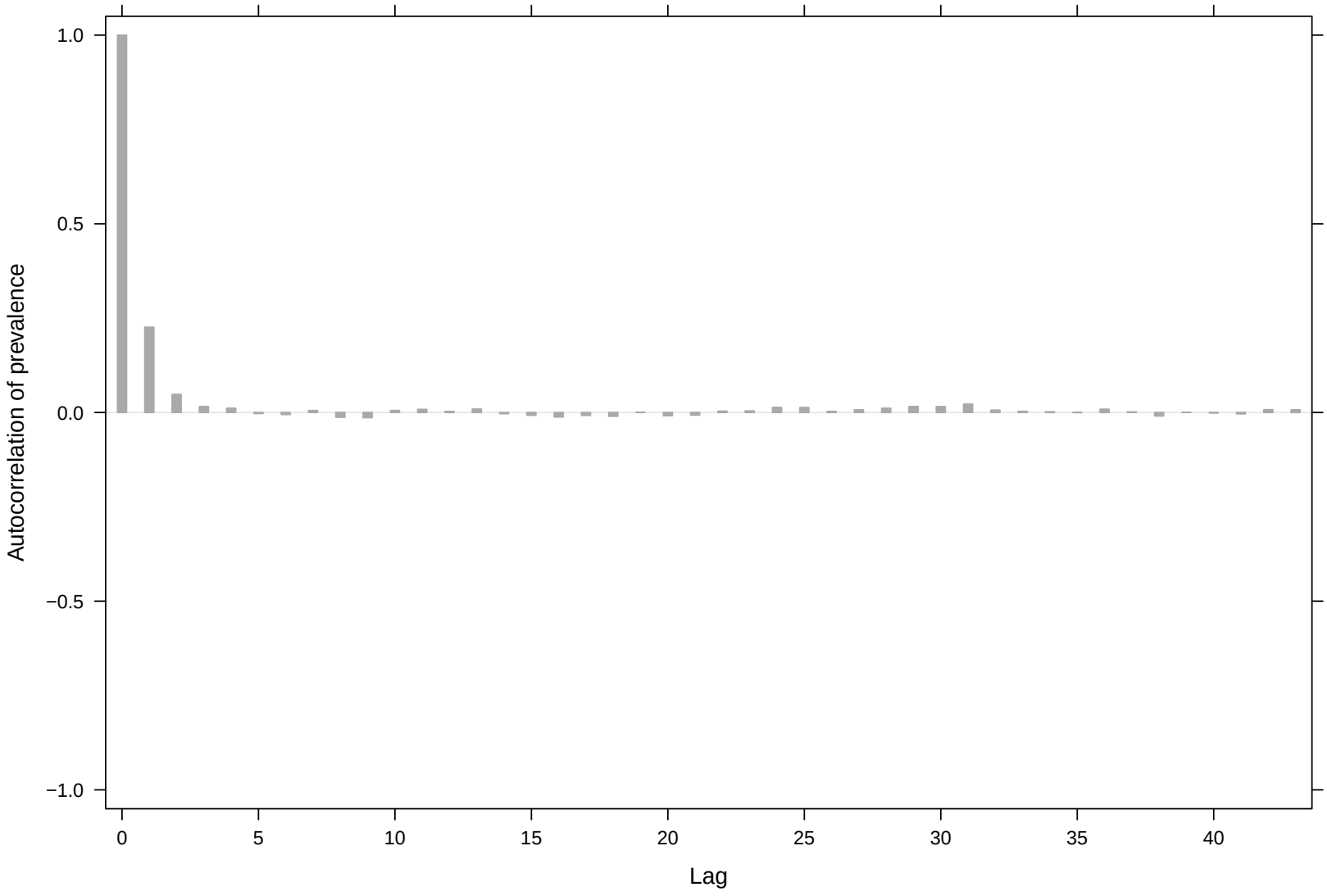
ECDF plots: the two chains should be very close to each other:



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):

```
##
## JAGS model summary statistics from 20000 samples (chains
##
##            Lower95  Median Upper95    Mean      SD Mode
## prevalence 0.40333 0.64942 0.86777 0.64308 0.12234   --
##
##                AC.10 psrf
## prevalence 0.0056884    1
##
## Total time taken: 0.6 seconds
```

Reminder: we want psrf < 1.05 and SSeff > 1000

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)
```

- But I would advise that you stick to using a separate text file!

Setting the RNG seed

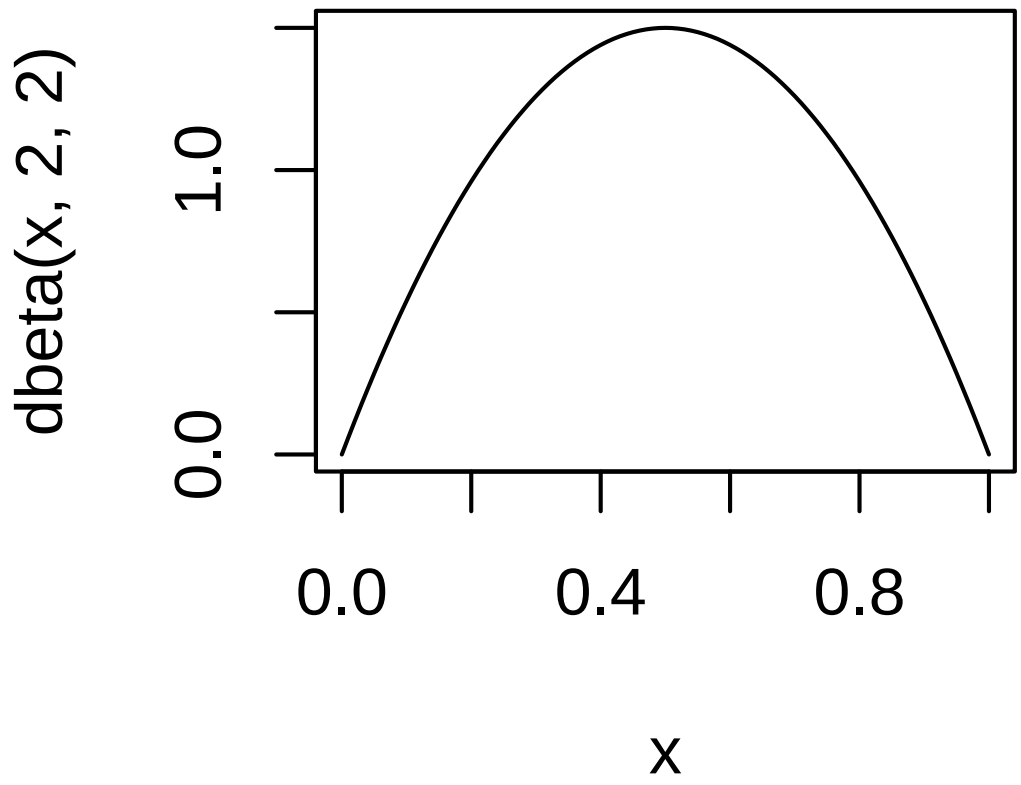
If we want to get numerically replicable results we need to add `.RNG.name` and `.RNG.seed` to the initial values, and an additional `#modules#` lecuyer hook to our basicjags.bug file:

```
.RNG.name <- "lecuyer::RngStream"
.RNG.seed <- list(chain1=1, chain2=2)
results <- run.jags('basicjags.bug', n.chains=2)
```

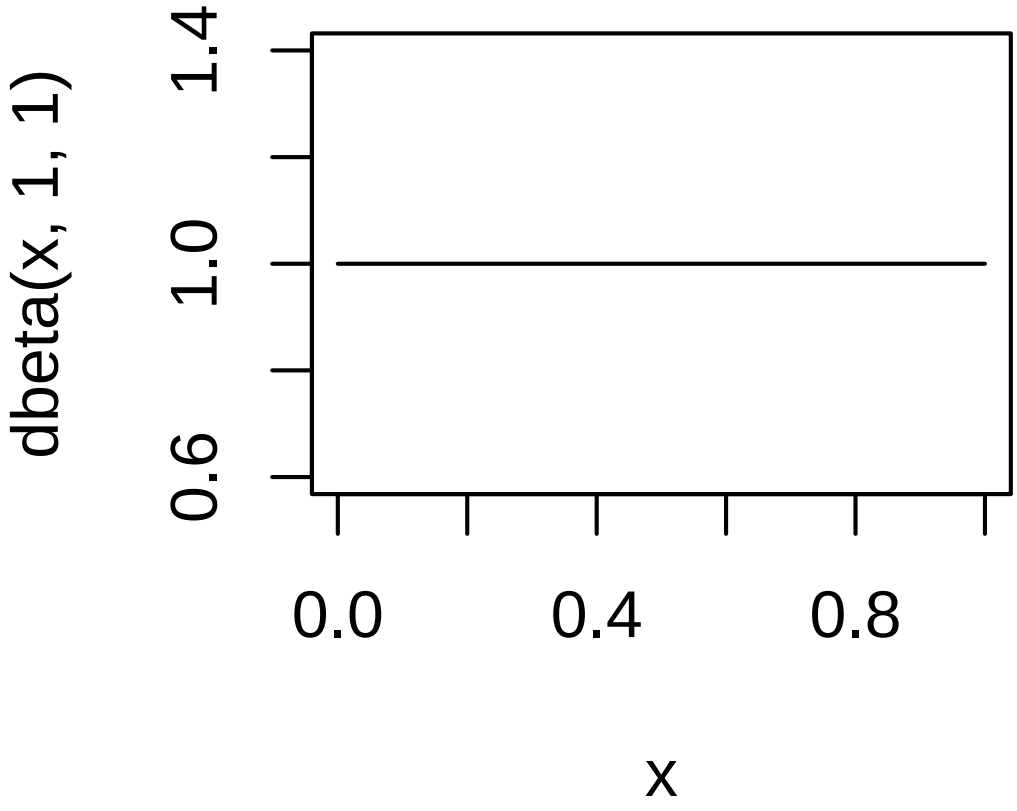
Every time this model is run the results will now be identical

A different prior

- A quick way to see the distribution of a prior:



- A minimally informative prior might be:



- 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# Positives, TotalTests
  #monitor# prevalence
  #inits# prevalence
}
```

An Equivalent Model

- We could equivalently specify an observation-level model:

```
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!