Preparation for online workshop

Homework and software installation for the HARMONY training workshop on diagnostic test evaluation

Matt Denwood and Arianna Comin

2022-06-06

Overview

This document will help you to prepare for the HARMONY training workshop on diagnostic test evaluation being held in Uppsala, Sweden from 8th to 10th June 2022. The workshop will be oriented around practical exercises, so it is very important that you are using a computer with the necessary software installed. Although we will try to keep the programming to a minimum during the workshop, you will need to be sufficiently familiar with R so that you can use it for basic tasks. You will also need to download the teaching material from our GitHub site, so you might find it helpful to have created an account with GitHub before the course so that you can access the material more easily during the week (including any updates we make). Finally, the training workshop will start from the assumption that you have a basic understanding of Bayes' theorem, and what the difference is between a likelihood, prior and posterior distribution. The purpose of this document is to ensure that you have the necessary software installed, access to the GitHub repository where the teaching material will be made available, and to give you a brief refresher in the basics of Bayesian statistics.

Software installation

You need to install R (version 4.1.0 or later) from https://cran.r-project.org/ as well as RStudio Desktop (as recent a version as possible) from https://www.rstudio.com/products/rstudio/download/

Please also install the latest versions of the following R packages from CRAN:

```
packages <- c("tidyverse", "PriorGen", "rjags", "runjags",
                    "coda", "TeachingDemos", "knitr", "ggdag")
if(length(find.package(packages)) != length(packages)){
  install.packages(packages)
}
```

You will also need the standalone JAGS software (version 4.3.0 or later) for the course -

download the installer for your platform from here: https://sourceforge.net/projects/mcmc-jags/files/JAGS/4.x/

Note for Windows

You need to match your R and JAGS versions as follows:

- R version 4.1.x: you need the official installer of JAGS version 4.3.0 (and NOT 4.3.1)
- R version 4.2.x: you need the official installer of JAGS version 4.3.1 (and NOT 4.3.0)

Note for macOS

If you are using an M1 (Apple Silicon, aka arm64) mac, then you should use the official installer of JAGS 4.3.1 as this will run natively on both x86_64 and arm64 hardware. If you are using an older Intel mac then you can use either JAGS 4.3.0 or JAGS 4.3.1 (there will not be a difference between the two for your hardware). This is the same irrespective of which version of R you use.

Note for Linux

It is very important that the compiler toolchain matches between R and JAGS - if you are installing from an official repository then this should be taken care of for you, but if you are compiling yourself then you do need to ensure that the same compilers (C++ and Fortran) are used for both.

Note for proficient C++ coders

For the final session, we may (if there is interest) give a brief introduction on how to add functions and distributions to JAGS by creating an R package that creates and loads a new a JAGS module via the C++ interface provided by JAGS. In order to be able to run the examples yourself you will need to have installed the Rcpp package as well as working C++ compilers that match the compiler toolchain used to build R and JAGS (i.e. Rtools for Windows, or Xcode command line tools on macOS). This session will only be for those individuals that are (a) interested and (b) already proficient C++ coders, so if you are not already familiar with C++ and creating R packages then please ignore this note!

Checking installation

To check that you have installed the software correctly please run the following code within R (or RStudio) and make sure that no errors are produced:

```
stopifnot(getRversion() >= "4.1.0")
stopifnot(require('tidyverse'))
stopifnot(require('PriorGen'))
stopifnot(require('rjags'))
```

```
stopifnot(require('runjags'))
stopifnot(require('coda'))
stopifnot(require('TeachingDemos'))
stopifnot(packageVersion("runjags") >= numeric_version("2.2.1-7"))
stopifnot(testjags()$JAGS.available)
stopifnot(numeric_version(testjags()$JAGS.version) >= "4.3.0")
stopifnot(testjags()$rjags.found)
stopifnot(numeric_version(testjags()$rjags.version) >= "4-13")
```

If you have any difficulties installing the software or get errors from the code above, please let us know immediately so that we can resolve these before the start of the course.

GitHub

Basics

GitHub is an online code repository that in it's most basic form stores the version history of any code project you are working on, and allows you to go back in time to a previous version before someone (perhaps you!) introduced a bug. It is particularly useful when collaborating with others because it allows you to see who made a change, when it was made, and what the code looked like before that. It also allows changes from different people to be merged into the same central repository to ensure that nobody gets out of sync with everybody else's code.

We will primarily be using GitHub as a way to disseminate the lecture notes and R/JAGS code for the exercises on course, so you only need to use the most basic features of GitHub (but it is a good thing to learn).

Simple Web Usage

We have created a public repository containing the teaching material for the training workshop. This means that anyone can view the teaching material at any time via the following website: https://github.com/ku-awdc/HarmonyWS3

You should see a number of folders listed with names "Session_0", "Session_1" etc, and within these folders are files like "Session_1.Rmd", "Session_1.pdf" etc. Each of the sessions for the training workshop (including this preparation document) has a series of files: the .Rmd is the original Rmarkdown document, and the .pdf & .html & .R files are created from these. You can click on any of these files to view them, although of the different file formats the .pdf version is probably easiest to download directly from the website. There are two problems with this:

1 - You will likely encounter problems when copy/pasting R code from the PDF files.

2 - If/when any of the files are updated, you will not get an automatic update of the new version.

We therefore recommend that you follow the instructions below to clone the repository directly to your computer.

Creating an Account

If you have never used GitHub before, you should create an account via http://github.com - it is free and easy. Remember to make a note of the username and password that you just created!

Installing Git

Unless you already have Git installed on your system, you will need to install it in order to be able to use Git within RStudio. There are a number of ways of doing this, depending on your system and access privileges, but the easiest method is to install GitHub Desktop from https://desktop.github.com (this will also install Git as a command line tool). If you are unable to install GitHub desktop, then see the following links for some alternative options:

- https://github.com/git-guides/install-git
- https://happygitwithr.com/install-git.html

Cloning a repository inside RStudio

The easiest way to clone the `HarmonyWS3` repository on to your computer is via RStudio. Go to the `File` menu and select `New project`. It will open and wizard menu, giving you three alternative ways to create a project: select `Version Control` and then `GIT`. Then type/paste `https://github.com/ku-awdc/HarmonyWS3.git` into the box where it says `Repository URL`. A suggested local path folder will be created automatically at the bottom but feel free to change this. Then click on `Clone` and wait for the files to download.

Creating the clone copies all of the files currently on the GitHub repository to your computer at the local path specified above. This is where all of the course material will (eventually) be located. You can browse the content using Windows File Explorer or going to the `Files` browser tab in RStudio (usually located in the lower right window). For example, look inside the `'Session_0'` folder and open up the `'Session_Preparation.html'` file - that is a local copy of the same document you are currently reading! But now you also have the PDF version - use this if you want to print the document for some reason, but it is a good idea to stick to using the HTML version if you want to copy/paste R code (you will probably encounter problems with quotation marks if you copy/paste R code from PDF files). Alternatively, you can use the .R files to run the R code directly. The code in each of the three versions should be identical.

Remember to select the option `View in web browser` when opening an html file from inside RStudio.

The next time you want to open the cloned repository, go to the `File` menu inside RStudio and select `Open project`. Locate your project and click `open`.

Modifying Files

Once you have set up the local copy of the repository, you can then add, delete and modify any of the files within your local copy of the repository. Try doing this by deleting the `'Session_Preparation.html'` file. Now go to the `Git` tab in RStudio (usually located in the top right window) where you should see a line appear with a *D* letter in a box to the left hand side - this is telling you that a file has been deleted locally (if you had modified the file rather than deleting it, the box would be blue and contain a *M*). However, you don't want to delete or modify any of the files we are providing in case we update them later. If you do this by mistake, just toggle the box correspoding to the relevant line (the D will move one step to the left), right click and select `Revert` - the file should then be restored. Do this now for the `'Session_Preparation.html'` file and check that it has reappeared. If you do want to modify a file for some reason, we suggest that you copy it and modify the copied version. If you keep the copy inside the same folder then it will appear as a modification (blue M in the box) but you can safely ignore these, or move the copied file outside of the repository folder if you want to keep things simple.

Fetching Updates

An important advantage of GitHub is that we can update a file online (for example to fix a typo or add new material discussed during the workshop), and that update will be immediately available to all of you. But in order to receive the updated files you need to tell RStudio to look for updates. In the `Git` tab in RStudio click on 'Pull' (blue arrow facing downwards) to make sure that any recent changes are synced to your computer. As long as you remember to pull changes regularly then you will always have an up-to-date copy - so do this regularly. Forgetting to do this is the only real potential downside of Git.

More information

That is pretty much all you need to know for the training workshop but there are some good tutorials available, for example: https://happygitwithr.com.

Revision of Basic Principles

Introduction

The aim of this pre-course work is to revise the underlying principles of Bayesian statistics that you will need during the course. It is designed so you can run the code shown and check that your results agree with those shown here. Feel free to play around with the code to see what happens - that is the best way to learn! If you have not encountered a function before then consult the help file for that function with a question mark followed by the function name, for example: `?seq` . Some prior familiarity with R will be assumed for this exercise but if you need to brush up then there are lots of tutorials online e.g. https://www.datacamp.com/courses/free-introduction-to-r

We will assume that you have all worked through this material in advance of the course.

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 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
likelihood_1 <- choose(tosses, heads) * probability^heads *
                (1-probability)^(tosses-heads)
likelihood_1
```

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

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

```
likelihood_2 <- dbinom(heads, tosses, probability)
likelihood_2
```

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

These two numbers are the same, which is reassuring. R also has a LOT of other distributions, but the most common are:

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

If you are not familiar with these distributions, then you can take a look them (and their common uses) on the internet.

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. 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 (prevalence) as an argument, then calculates the likelihood of the data based on this parameter:

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

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:

```
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. We could have also got the same using:

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

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

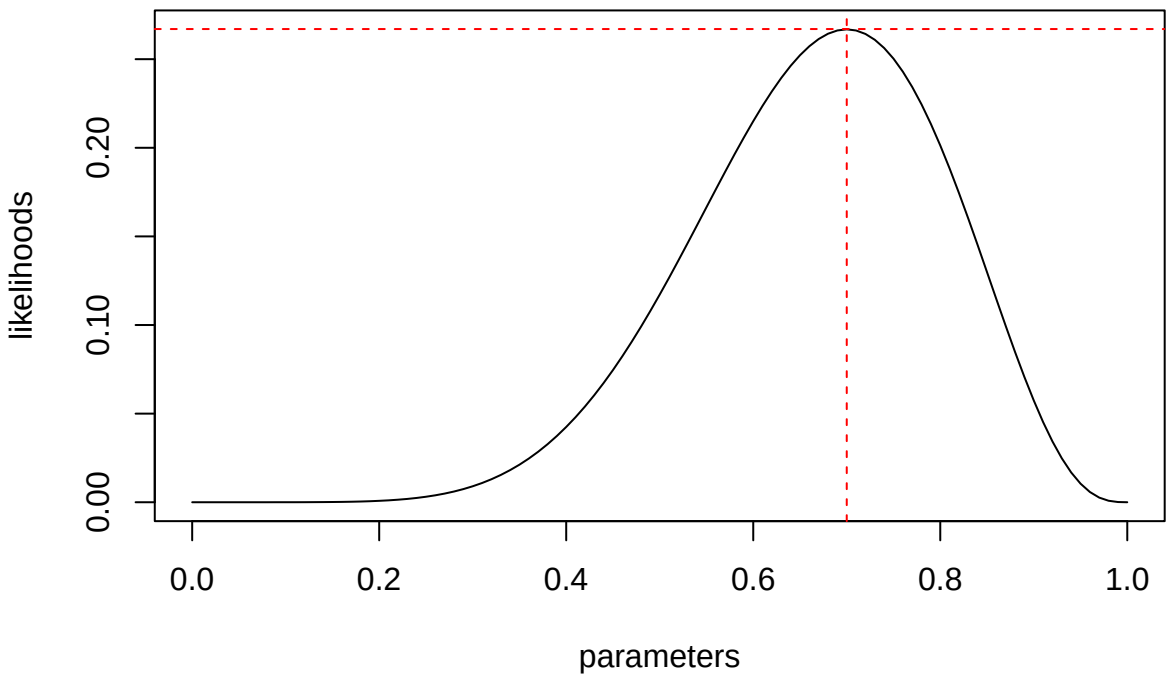
This is really what is happening when you use a (generalised) linear model in R - it optimises the parameter values to give the highest likelihood for the data and model (distribution with

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. 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.

```
parameters <- seq(0, 1, length.out=101)
likelihoods <- numeric(length(parameters))
for(i in 1:length(parameters)){
    likelihoods[i] <- likelihood_fun(parameters[i])
}
plot(parameters, likelihoods, type='l')
abline(h=0.267, lty='dashed', col='red')
abline(v=0.7, lty='dashed', col='red')
```



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:

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

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

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.

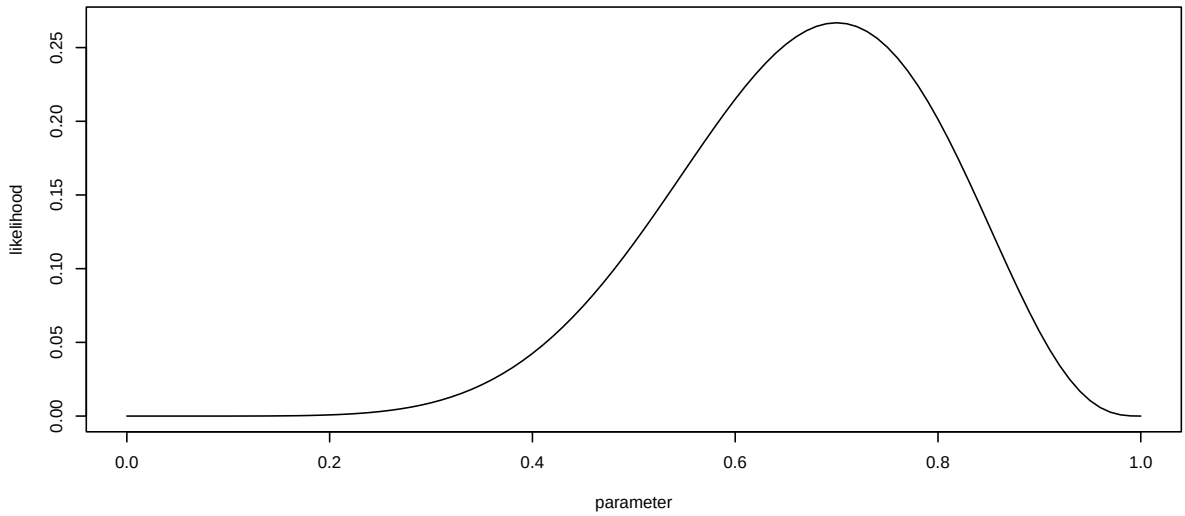
Profiling a Posterior

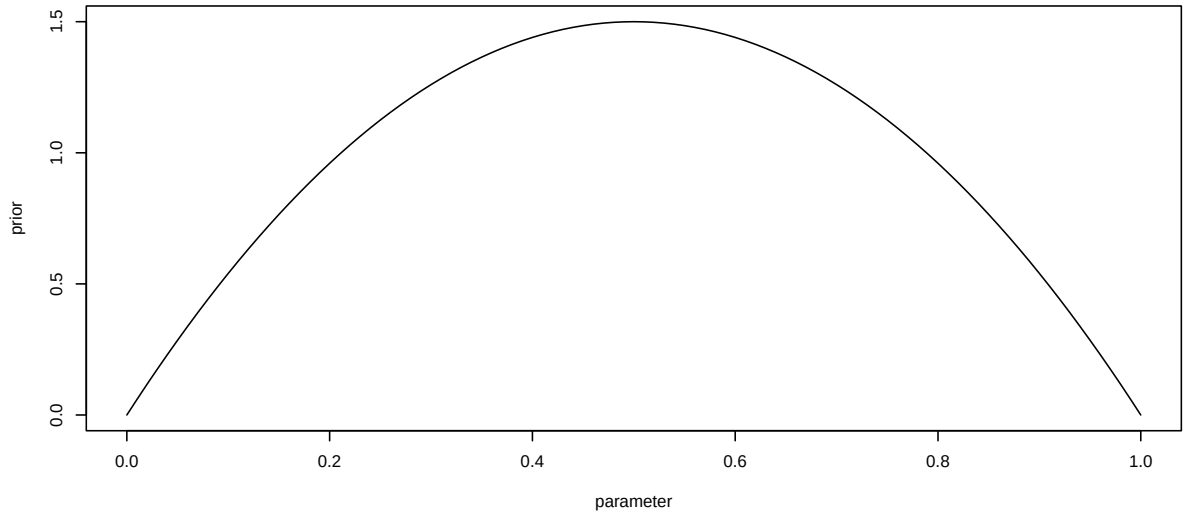
We can profile a posterior in the same way that we profiled the likelihood before. But first we need to define a prior - let's do this using a function again:

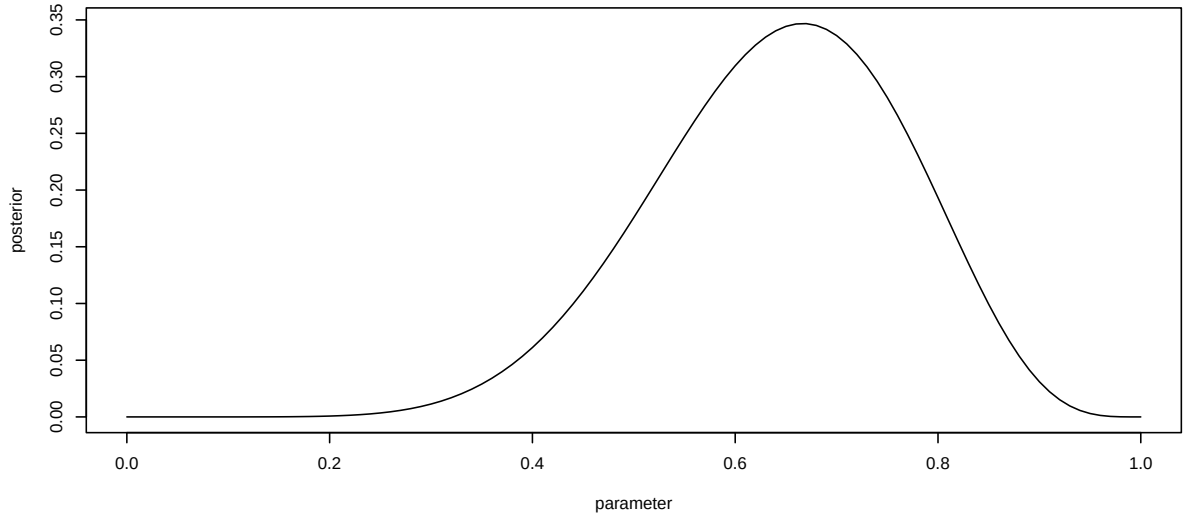
```
prior_fun <- function(prevalence) dbeta(prevalence, 2, 2)
```

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

```
results <- data.frame(parameter=seq(0, 1, length.out=101),
                      likelihood=NA, prior=NA, posterior=NA)
for(i in 1:nrow(results)){
    results$likelihood[i] <- likelihood_fun(results$parameter[i])
    results$prior[i] <- prior_fun(results$parameter[i])
    results$posterior[i] <- results$likelihood[i] * results$prior[i]
}
par(mfrow=c(3,1))
with(results, plot(parameter, likelihood, type='l'))
with(results, plot(parameter, prior, type='l'))
with(results, plot(parameter, posterior, type='l'))
```


likelihood
0.00
0.05
0.10
0.15
0.20
0.25
0.0
0.2
0.4
0.6
0.8
1.0
parameter


prior
0.0
0.5
1.0
1.5
0.0
0.2
0.4
0.6
0.8
1.0
parameter


posterior
0.00
0.05
0.10
0.15
0.20
0.25
0.30
0.35
0.0
0.2
0.4
0.6
0.8
1.0
parameter

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 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 a confidence interval, which represents the degree of uncertainty in the posterior. For our prevalence example (with 7/10 positive and a $Beta(2, 2)$ prior), these estimates would be:

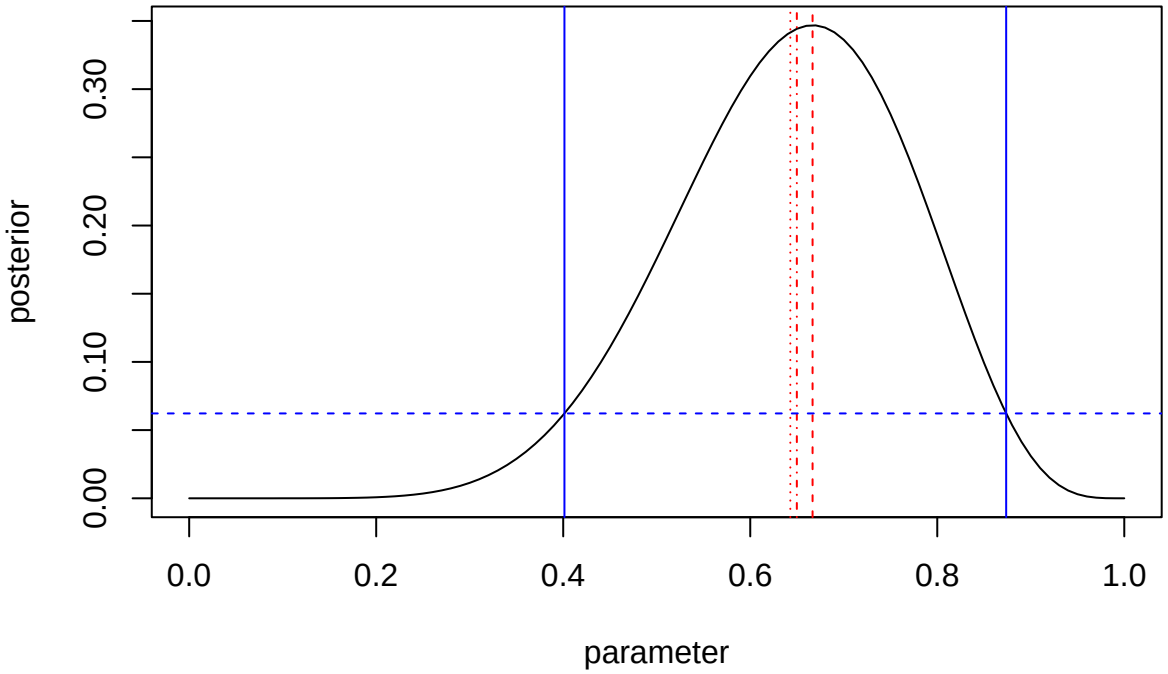
```
## Mode: 0.6666667
```

```
## 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 dotted), median (red dot-dash) and mode (red dashed) all reflect the "best guess" for our parameter value - typically either the mean or median would be reported (I

tend to choose the median but this is a matter of taste). 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% confidence interval, which is actually a 95% credible interval calculated using a highest posterior density interval. This is a particular kind of confidence interval that we use 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".

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 focusing 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:

```
# The coda package has many utilities to work with MCMC objects:
library('coda')

metropolis <- function(burnin = 0, sample = 10000, sigma = 0.05,
               initial_value = 0.05, plot=TRUE){
  stopifnot(initial_value > 0, initial_value < 1)
  stopifnot(sigma > 0)
  burnin <- as.integer(burnin)
  sample <- as.integer(sample)
  stopifnot(burnin >= 0)
  stopifnot(sample > 0)

  # Redefine these to work on the log scale:
  llikelihood_fun <- function(prevalence)
    dbinom(7, 10, prevalence, log=TRUE)
```

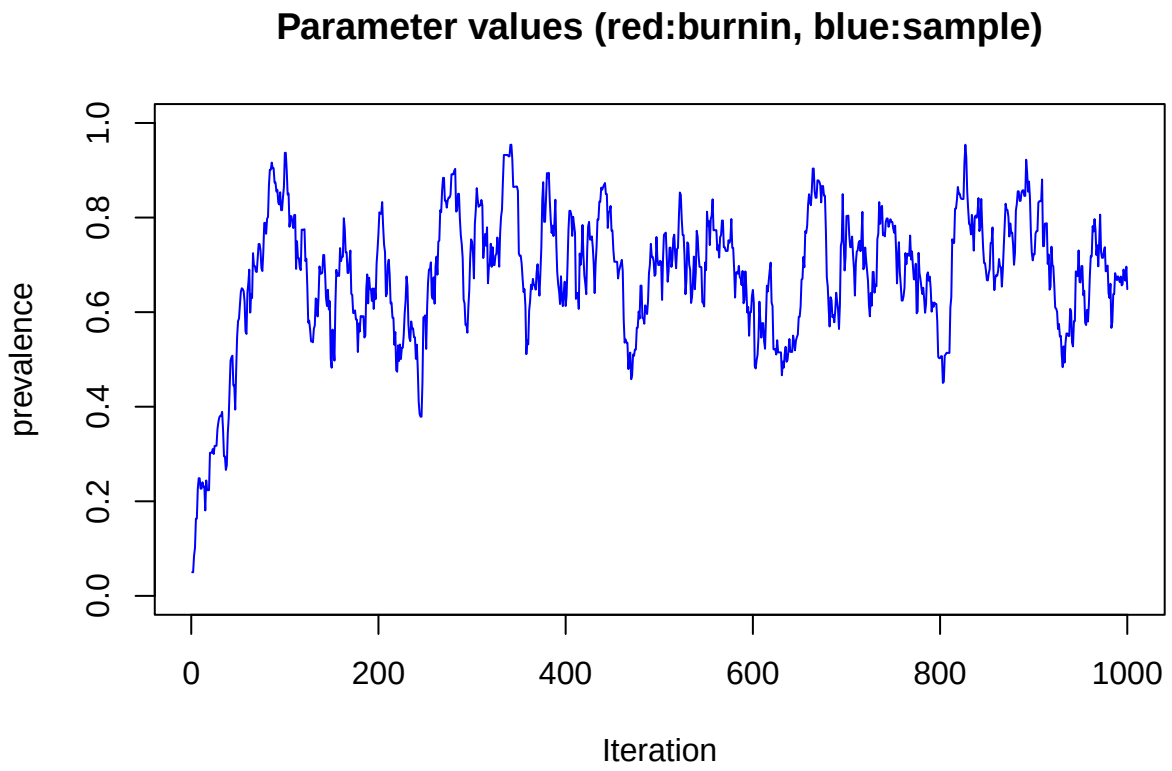
```
  lprior_fun <- function(prevalence)
    dbeta(prevalence, 2, 2, log=TRUE)

  parameters <- numeric(burnin+sample)
  parameters[1] <- initial_value
  current <- initial_value
  post <- llikelihood_fun(current) + lprior_fun(current)
  for(i in 2:(burnin+sample)){
    proposal <- rnorm(1, current, sigma)
    if(proposal > 0 && proposal < 1){
      newpost <- llikelihood_fun(proposal) + lprior_fun(proposal)
      accept <- newpost > post || rbinom(1, 1, exp(newpost-post))
      if(accept){
        current <- proposal
        post <- newpost
      }
    }
    parameters[i] <- current
  }

  if(plot && burnin > 0){
    plot(1:burnin, parameters[1:burnin], type='l', col='red',
         xlim=c(1,burnin+sample), ylim=c(0,1),
         main='Parameter values (red:burnin, blue:sample)',
         ylab='prevalence', xlab='Iteration')
    lines((burnin+1):(burnin+sample), parameters[-(1:burnin)], col='blue')
  }else if(plot){
    plot(1:sample, parameters, type='l', col='blue',
         xlim=c(1,burnin+sample), ylim=c(0,1),
         main='Parameter values (red:burnin, blue:sample)',
         ylab='prevalence', xlab='Iteration')
  }

  parameters <- window(coda::as.mcmc(parameters), start=burnin+1)
  varnames(parameters) <- 'prevalence'

  return(parameters)
}
```

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, but don't worry too much about the details.

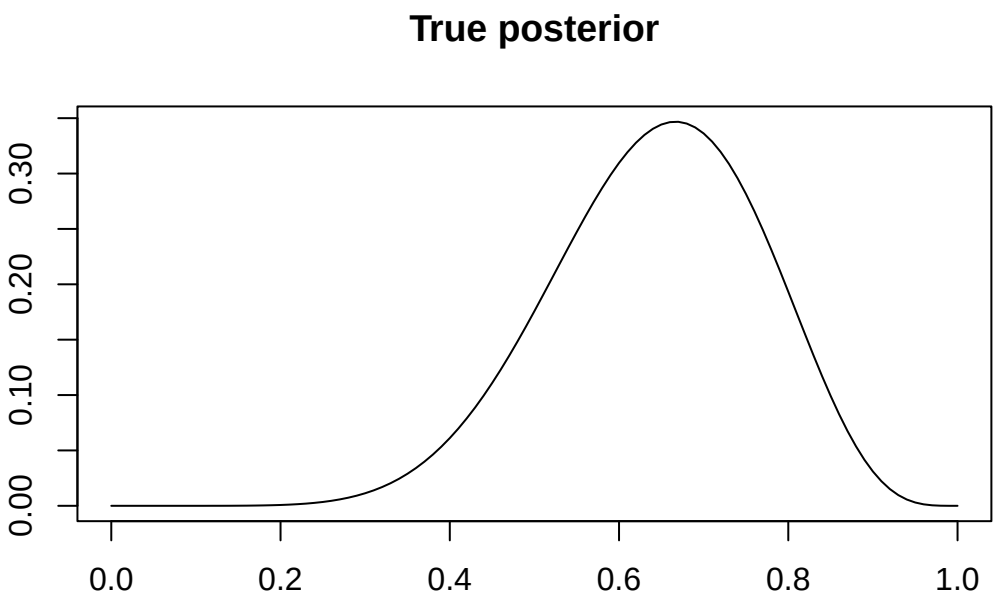
The important part is that we can run this function and obtain results as follows:

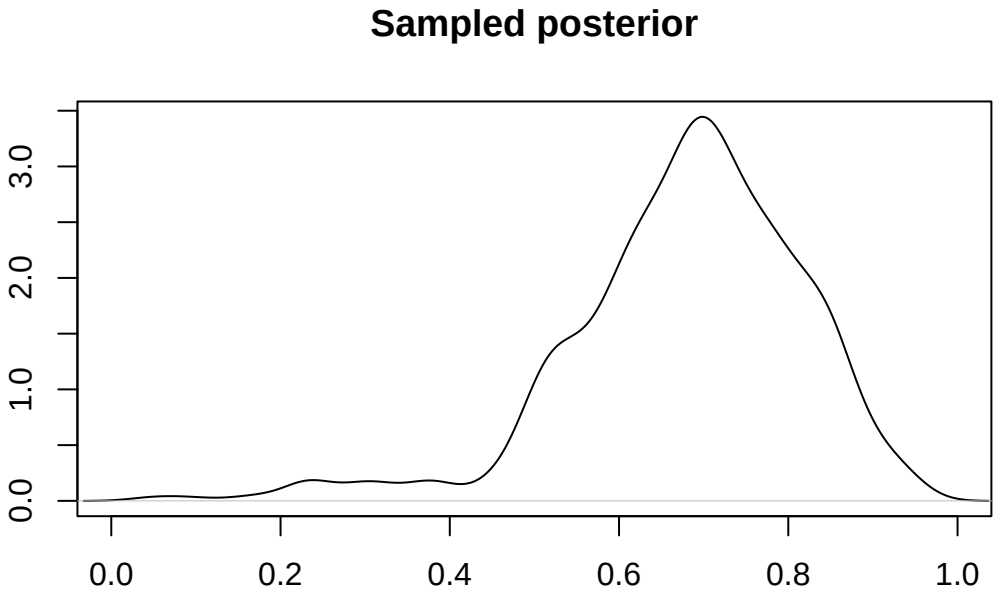
```
samples <- metropolis(burnin = 0, sample = 1000, initial_value=0.05)
```



The algorithm starts an initial value of prevalence=0.05 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:

```
par(mfrow=c(2,1))
with(results, plot(parameter, posterior, type='l',
                   xlim=c(0,1), main='True posterior', ylab=NA, xlab=NA))
plot(density(samples), xlim=c(0,1), main='Sampled posterior',
     ylab=NA, xlab=NA)
```





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

```
# Mean of samples:
mean(samples)
```

```
## [1] 0.6789703
```

```
# Median of samples:
median(samples)
```

```
## [1] 0.6947516
```

```
# 95% CI of samples:
HPDinterval(samples)
```

```
##                lower    upper
## prevalence 0.4451781 0.936942
## attr(,"Probability")
## [1] 0.95
```

And compare these to the true values that we calculated earlier using posterior profiling:

```
## True posterior mean: 0.6428571
```

```
## True posterior median: 0.6498366
```

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

Again, these are close, but not the same. There are two reasons for this:

1 - We did not remove the burnin period

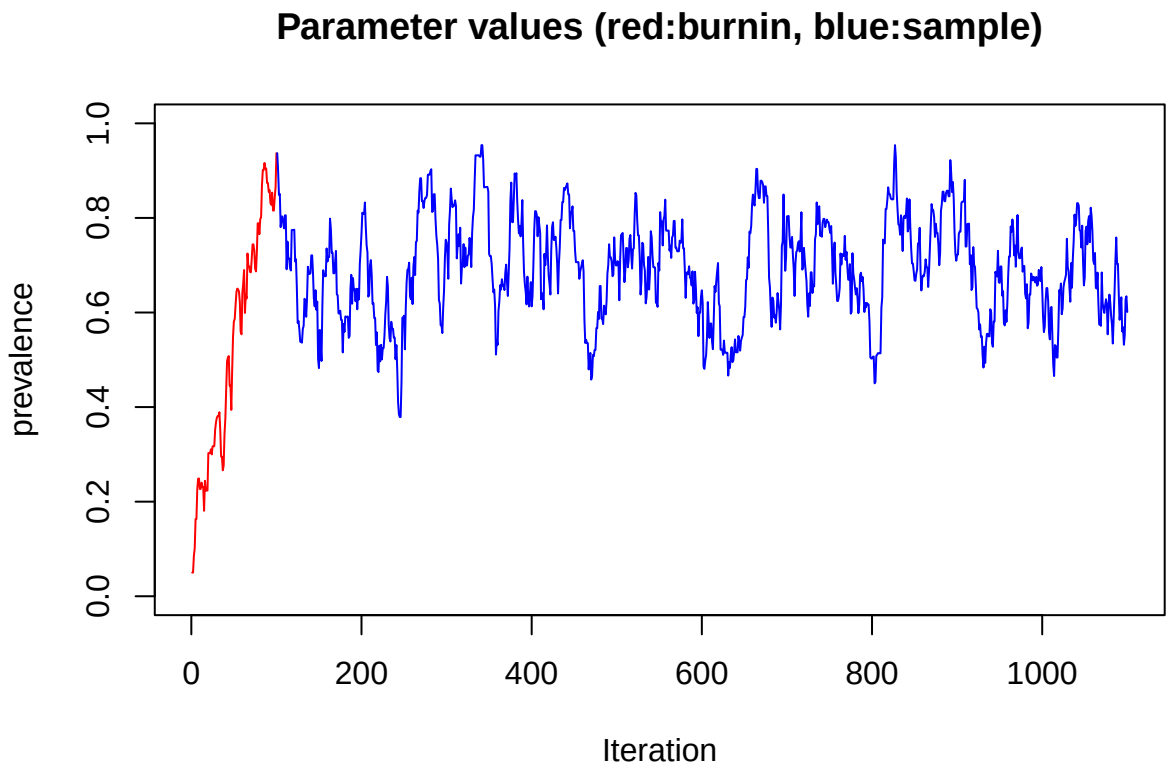
2 - We do not have a sufficiently high effective sample size

Let's look at how to fix these problems.

Burnin

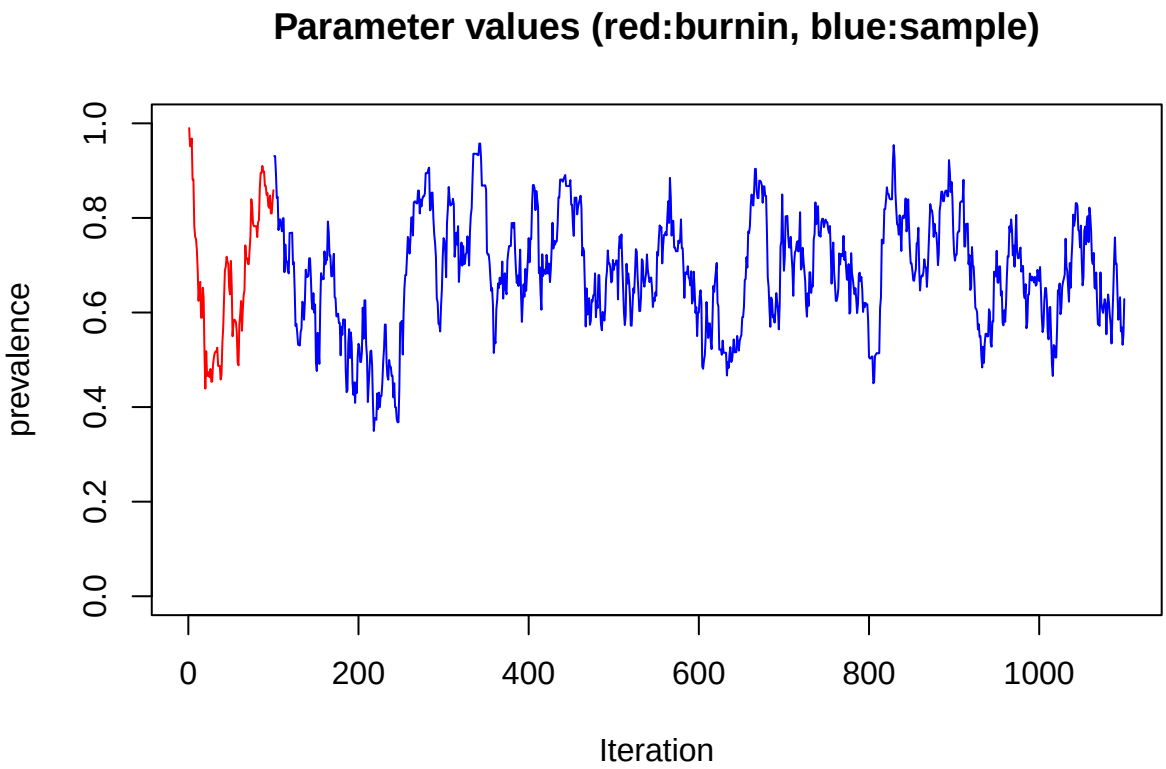
When we ran the simulation last time, we set an initial value of 0.05 and it took a little while for the Metropolis algorithm to move away from this starting value because of the autocorrelation in the chains. This is OK, but we don't actually want to count this "burnin" period as part of the posterior. So we need to tell the algorithm to run for a few iterations before starting to sample from the posterior. For example:

```
samples <- metropolis(burnin = 100, sample = 1000, initial_value=0.05)
```



This time, we told the function to discard the first 100 samples (shown in red) and then take another 1000 samples from the posterior after this point. This means that we can set the initial values to wherever we want, and it won't affect the samples we take (the blue part) even though it will affect the burnin period (the red part). For example if we set an initial value of 0.99:

```
samples <- metropolis(burnin = 100, sample = 1000, initial_value=0.99)
```



The start of the red parts of the chains look quite different (one starts low the other starts high), but by the time it gets to the end of the red part they seem to be moving around in the same area. At this point we say that the chains have converged, which is just a fancy way of saying that it is safe to assume the next samples are from the true posterior. Note that it doesn't matter if we have a burnin period that is too long (e.g. in this case it seems that 50 iterations would have been enough) - it is far safer to remove more than you think is necessary rather than risking not removing enough. It is also impossible to know a priori how long the burnin will take, so if you don't specify a long enough burnin period then you may have to remove the first part of the sampled chains manually. This is another reason to err on the side of caution and specify a longer burnin period than you think will be needed.

Important point: Assessing convergence (and ensuring adequate burnin) is a crucial (and sometimes tricky) part of using MCMC in practice.

Effective sample size

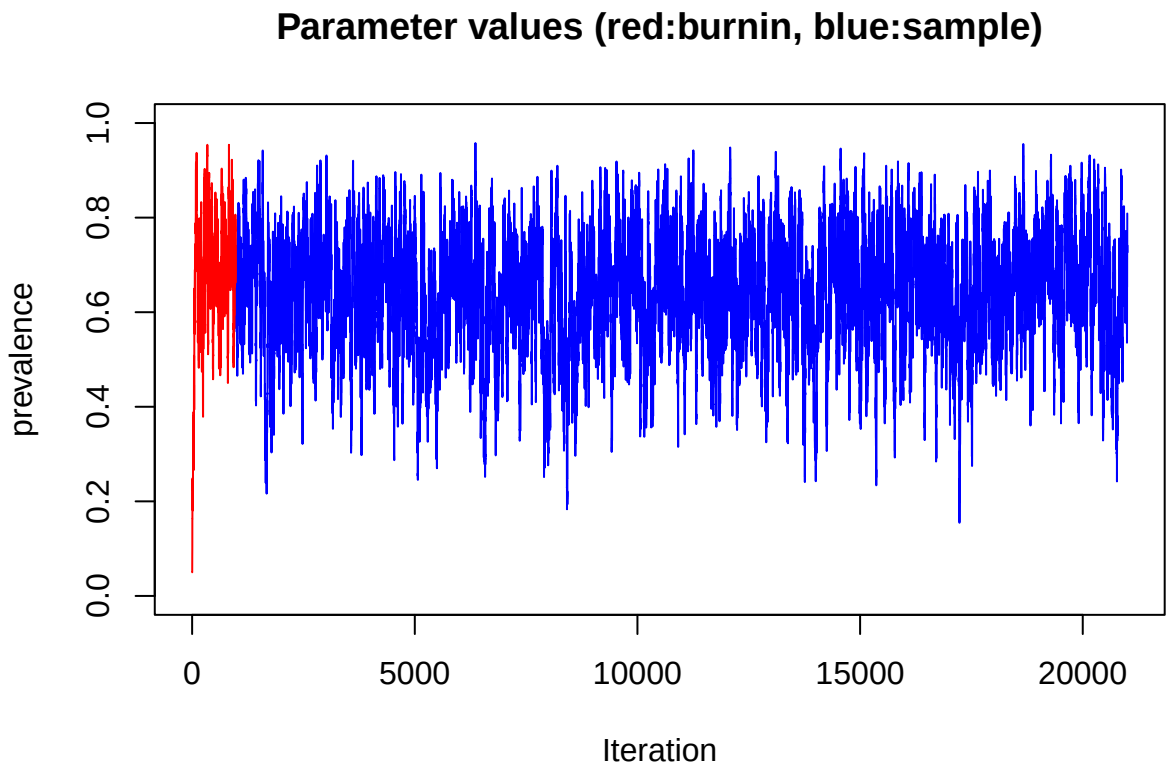
Another reason that our samples differed to the true posterior is because the samples are random, and therefore there is some sampling noise. In order to eliminate this we need to make sure we have a sufficient number of samples so that the difference between e.g. the mean of the sample and the true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

```
# Effective sample size:
effectiveSize(samples)
```

```
## prevalence
##   33.76139
```

So we have the equivalent of approximately 33 independent samples from this posterior, which is not nearly enough to be a good approximation. 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:

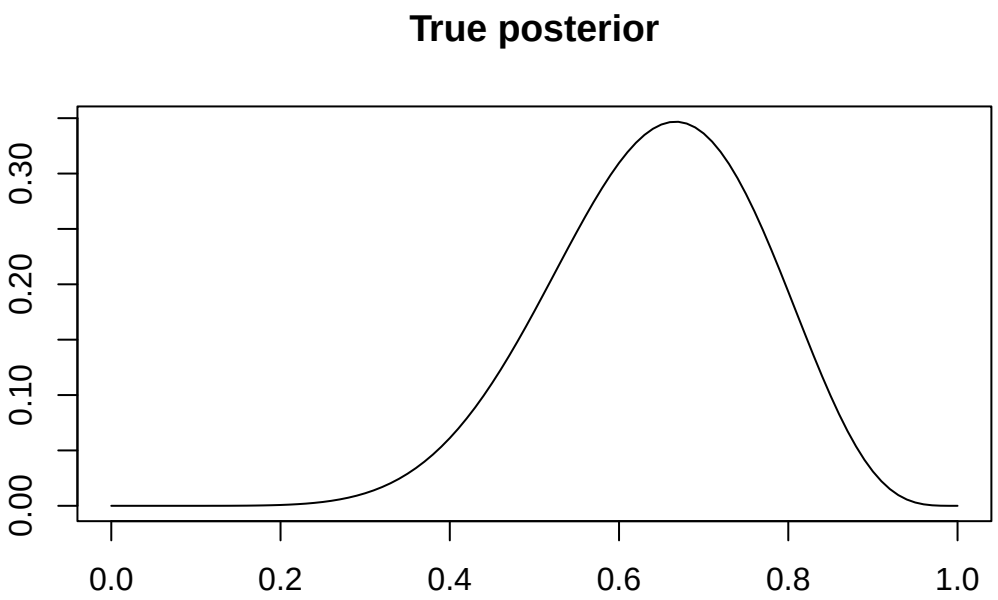
```
samples <- metropolis(burnin = 1000, sample = 20000, initial_value=0.05)
```

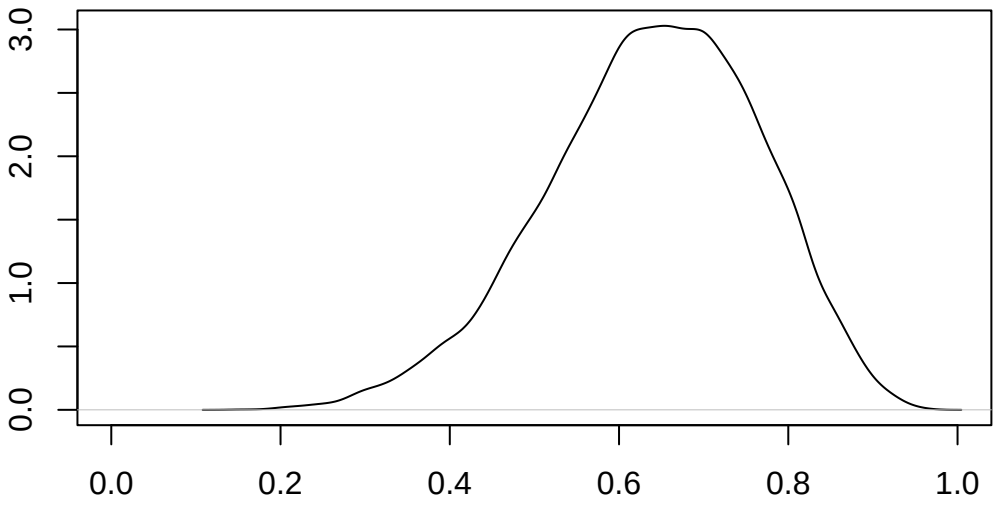



```
# Effective sample size:
effectiveSize(samples)
```

```
## prevalence
##   618.2718
```

We now have 20000 iterations, which equates to around 618 independent samples (i.e. we need around 33 iterations to get 1 independent sample). This should be enough to get a decent approximation to the true posterior:






```
## Mean of samples:  0.6395573
## True posterior mean: 0.6428571
## Median of samples:  0.6469437
## True posterior median: 0.6498366
## 95% CI of samples:  0.3851063 - 0.8663473
## True posterior 95% CI: 0.401307 - 0.8736879
```

Our sampled posterior is now much closer to the true posterior, although there is still a small difference due to the inherent randomness of the Monte Carlo integration. This also means that every time we run this simulation we will get a slightly different answer - 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(2021-06-18)
samples <- metropolis(burnin = 1000, sample = 20000,
                      initial_value=0.99, plot=FALSE)
# Mean of samples:
mean(samples)
```

```
## [1] 0.6396308
```

```
set.seed(2021-06-18)
samples <- metropolis(burnin = 1000, sample = 20000,
                      initial_value=0.99, plot=FALSE)
# Mean of samples:
mean(samples)
```

```
## [1] 0.6396308
```

These two results are numerically identical because the same seed was used. But the following simulation uses a different seed so gives slightly different results:

```
set.seed(2021-06-19)
samples <- metropolis(burnin = 1000, sample = 20000,
                      initial_value=0.99, plot=FALSE)
# Mean of samples:
mean(samples)
```

```
## [1] 0.6381263
```

Important point: MCMC is a numerical approximation to the posterior, so if we want a good approximation then we need to be sure our effective sample size is sufficient.

Exercise 1

Run the code below with different values of *sample*. Make sure you understand the relationship between the number of samples, the effective sample size, and the accuracy of your estimates from the sampled posterior:

```
samples <- metropolis(burnin = 1000, sample = 10000)
# Effective sample size:
effectiveSize(samples)
# Mean of samples:
mean(samples)
# Median of samples:
```

```
median(samples)
# 95% CI of samples:
HPDinterval(samples)
```

Remember the true values for comparison:

```
## True posterior mean: 0.6428571
```

```
## True posterior median: 0.6498366
```

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

Exercise 2

Up to now we have not changed the parameter *sigma*. For the Metropolis algorithm, the value of *sigma* has a big effect on the degree of autocorrelation in the chains. If the value is either too large or too small, then there will be a lot of autocorrelation in the chain, which will result in two things:

1 - The chain might take longer to burn in (particularly if sigma is too small)

2 - The effective sample size will be reduced

Run the following code with different values of sigma and see how it affects the autocorrelation. Make sure you understand the relationship between autocorrelation, the length of the burnin period, and the effective sample size. Here are some examples to get you started:

```
# Large sigma:
samples <- metropolis(sigma=10)
# Autocorrelation:
autocorr(samples, lags=1)
# Effective sample size:
effectiveSize(samples)

# Moderately large sigma:
samples <- metropolis(sigma=1)
# Autocorrelation:
autocorr(samples, lags=1)
# Effective sample size:
effectiveSize(samples)

# Moderately small sigma:
samples <- metropolis(sigma=0.1)
# Autocorrelation:
autocorr(samples, lags=1)
# Effective sample size:
effectiveSize(samples)
```

```
# Small sigma:
samples <- metropolis(sigma=0.01)
# Autocorrelation:
autocorr(samples, lags=1)
# Effective sample size:
effectiveSize(samples)
```

Can you find a value of *sigma* that gives an effective sample size of >= 2000?

Conclusions

In the real world we would very rarely use a Metropolis algorithm, but it is a useful exercise to understand the basic concepts. You can think of all other types of MCMC as used by JAGS/BUGS (and also Hamiltonian Monte Carlo as used by Stan) as being extensions of this 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 burnin period has been run before starting to sample. The two most common methods of doing this are to look at the potential scale reduction factor (psrf) of the Gelman-Rubin statistic, and to look at trace plots (typically of 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. This is relatively easy to do, simply by remembering to 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.

We will reinforce these concepts (and show you how to assess them for models run in JAGS) during the training workshop.