Hierarchical Models Made Easy

Peter Solymos and Subhash Lele

July 16, 2016 — Madison, WI — NACCB Congress

Contents

1 About this document **2**

2 About the course **2**

2.1 Where, when? . . . 2

2.2 Description . . . 2

2.3 Organizational structure . . . 3

2.4 Course organizers . . . 3

3 How to prepare **3**

3.1 Install R . . . 3

3.2 Install RStudio . . . 3

3.3 Install JAGS . . . 4

3.4 Install R packages . . . 4

3.5 Check that everything works as expected . . . 4

4 R quickstart **5**

4.1 Basic data structures and operations . . . 5

4.2 Random numbers . . . 14

4.3 MCMC list objects . . . 16

5 Statistical concepts **20**

5.1 Introduction . . . 20

5.2 A simple example . . . 21

5.3 Assumptions . . . 21

5.4 Important properties of likelihood . . . 22

5.5 The maximum likelihood estimator . . . 29

5.6 The sampling distribution of the estimates . . . 30

5.7 Confidence interval and efficiency . . . 31

5.8 Estimated confidence intervals . . . 32

5.9 Coverage . . . 33

5.10 Summary . . . 36

5.11 Bayesian analysis . . . 36

5.12 Apps to illustrate Bayesian analysis . . . 37
5.13 Non-informative priors (objective Bayesian analysis) . . . 38
5.14 Data cloning: How to trick Bayesians into giving Frequentist answers? . . . 39
5.15 A brief theory of data cloning . . . 40
5.16 MCMC . . . 41

1 About this document

This document is a collection of tutorials for the the one-day short course 'Hierarchical models for conservation biologists made easy' at at NACCB congress in Madison, WI, on July 16th. The tutorials are intended to help participants making sure that necessary software is installed ahead of the course. Please spend some time reading the R quickstart, statistical concepts so that we can jump right into the fun part of hierarchical modeling on July 16th.

This introduction is provided in PDF and Rmd file formats. Use RStudio (see installation guide) to open the Rmd (R markdown file format) file to directly access R code used in the PDF file.

2 About the course

We aim this training course towards conservation professionals who need to understand and feel comfortable with modern statistical and computational tools used to address conservation issues. Conservation science needs to be transparent and credible to be able to make an impact and translate information and knowledge into action.

2.1 Where, when?

The congress program is out (see here).

Congress registration is now open here closed.

- The short course date: Saturday, July 16,
- time: 9:00 AM – 5:00 PM,
- location: Hall of Ideas H, Monona Terrace Community and Convention Center.

2.2 Description

Communicating scientific methods and results require a full understanding of concepts, assumptions and implications. However, most ecological data used in conservation decision making are inherently noisy, both due to intrinsic stochasticity found in nature and extrinsic factors of the observation processes. We are often faced with the need to combine multiple studies across different spatial and temporal resolutions. Natural processes are often hierarchical. Missing data, measurement error, soft data provided by expert opinion need to be accommodated during the analysis. Data are often limited (rare species, emerging threats), thus small sample corrections are important for properly quantify uncertainty.

Hierarchical models are useful in such situations. Fitting these models to data, however, is difficult. Advances in the last couple of decades in statistical theory and software development have fortunately made the data analysis easier, although not trivial. In this course, we propose to introduce statistical and computational tools for the analysis of hierarchical models (including tools for small sample inference) specifically in the context of conservation issues.

We will teach both Bayesian and Likelihood based approaches to these models using freely available software developed by the tutors. By presenting both Bayesian and Likelihood based approaches participants will be able to go beyond the rhetorics of philosophy of statistics and use the tools with full understanding of their assumptions and implications. This will help ensure that when they use the statistical techniques, be they Bayesian or Frequentist, they will be able to explain and communicate the results to the managers and general public appropriately.

2.3 Organizational structure

- Introduction and overview of statistical concepts: seminar format (1–1.5 hours), followed by a short break.
- Hierarchical models: hands on training.
- Lunch break (lunch provided) with informal discussions.
- Analyzing data with temporal and spatial dependence, model identifiability: hands on training with short break in the middle.

Participants should bring their own laptops. Open source and free software, and electronic course material will be provided by organizers.

Morning coffee, and afternoon iced tea and lemonade, plus light snack of chips and salsa will be provided.

2.4 Course organizers

- Péter Sólymos
- Subhash R. Lele

Information and course notes are going to be posted on the course website at http://datacloning.org/courses/2016/madison/.

3 How to prepare

Please follow these steps and install necessary software onto your computer that you are going to use at the course. This way we can spend more time on talking about modeling.

3.1 Install R

Follow the instructions at the R website to download and install the most up-to-date base R version suitable for your operating system (the latest R version at the time of writing these instructions is 3.3.1).

3.2 Install RStudio

Having RStudio is not absolutely necessary, but our course material will follow a syntax that is close to RStudio's R markdown notation, so having RStudio will make our life easier. RStudio is also available for different operating systems. Pick the open source desktop edition from here (the latest RStudio Desktop version at the time of writing these instructions is 0.99.902).

3.3 Install JAGS

We will use JAGS during the course because it is robust, easy to install, and cross-paltform available. Download the latest version suitable for your operating system from here (the latest JAGS version at the time of writing these instructions is 3.4.2).

Note: due to recent changes in R's Windows toolchain (which impacts Windows specific installation only), pay attention to matching versions:

- if you are using R 3.3.0 or later then install `JAGS-4.2.0-Rtools33.exe`,
- if you are using R 3.2.4 or earlier then install `JAGS-4.2.0.exe`.

3.4 Install R packages

Once R/RStudio and JAGS is installed, run the following commands in R/RStudio to install the necessary R packages (rjags, dclone, coda, snow, rlecuyer):

```
install.packages(c("rjags","dclone","coda","snow","rlecuyer"))
```

3.5 Check that everything works as expected

Because there are dependencies and version requirements, best to check that everything works. Please run the following code and follow the prompts:

```
check_if_ready_for_the_course <-
function()
{
    if (getRversion() < "2.15.1")
        stop("R >= 2.15.1 required")
    cat("--- R version is",
        as.character(getRversion()), "--- OK\n")

    if (!require(dclone))
        stop("dclone package not installed")
    if (packageVersion("dclone") < "2.1.1")
        stop("dclone >= 2.1.1 required")
    cat("--- dclone package version is",
        as.character(packageVersion("dclone")), "--- OK\n")

    if (!require(coda))
        stop("coda package not installed")
    if (packageVersion("coda") < "0.13")
        stop("coda >= 0.13 required")
    cat("--- coda package version is",
        as.character(packageVersion("coda")), "--- OK\n")

    if (!require(rjags))
        stop("rjags package not installed")
    if (packageVersion("rjags") < "4.4")
        stop("rjags >= 4.4 required")
    cat("--- rjags package version is",
        as.character(packageVersion("rjags")), "--- OK\n")
```

```
    if (!require(snow))
        stop("snow package not installed")
    cat("--- snow package version is",
        as.character(packageVersion("snow")), "--- OK\n")

    if (!require(rlecuyer))
        stop("rlecuyer package not installed")
    cat("--- rlecuyer version is",
        as.character(packageVersion("rlecuyer")), "--- OK\n")

    cat("\n--- YOU ARE READY TO GO!\n\n")
    invisible(NULL)
}
check_if_ready_for_the_course()
```

Congratulations! Now your computer is ready for the course.

Contact Peter Solymos if you still have problems.

4 R quickstart

This section covers the data structures and operators we will use during the short course.

4.1 Basic data structures and operations

R is a great calculator:

```
1 + 2
```

```
## [1] 3
```

Assign a value and print an object:

```
print(x <- 2)
```

```
## [1] 2
```

```
(x = 2) # shorthand for print
```

```
## [1] 2
```

```
x == 2 # logical operator, not assignment
```

```
## [1] TRUE
```

```
y <- x + 0.5
y # another way to print
```

```
## [1] 2.5
```

Logical operators:

```
x == y # equal
```

```
## [1] FALSE
```

```
x != y # not eaqual
```

```
## [1] TRUE
```

```
x < y # smaller than
```

```
## [1] TRUE
```

```
x >= y # greater than or equal
```

```
## [1] FALSE
```

Vectors and sequences:

```
x <- c(1, 2, 3)
x
```

```
## [1] 1 2 3
```

```
1:3
```

```
## [1] 1 2 3
```

```
seq(1, 3, by = 1)
```

```
## [1] 1 2 3
```

```
rep(1, 5)
```

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

```
rep(1:2, 5)
```

```
##  [1] 1 2 1 2 1 2 1 2 1 2
```

```
rep(1:2, each = 5)
```

```
##  [1] 1 1 1 1 1 2 2 2 2 2
```

Vector operations, recycling:

```
x + 0.5
```

```
## [1] 1.5 2.5 3.5
```

```
x * c(10, 11, 12, 13)
```

```
## Warning in x * c(10, 11, 12, 13): longer object length is not a multiple of
## shorter object length
```

```
## [1] 10 22 36 13
```

Indexing vectors, ordering:

```
x[1]
```

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

```
x[c(1, 1, 1)] # a way of repeatig values
```

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

```
x[1:2]
```

```
## [1] 1 2
```

```
x[x != 2]
```

```
## [1] 1 3
```

```
x[x == 2]
```

```
## [1] 2
```

```
x[x > 1 & x < 3]
```

```
## [1] 2
```

```
order(x, decreasing=TRUE)
```

```
## [1] 3 2 1
```

```
x[order(x, decreasing=TRUE)]
```

```
## [1] 3 2 1
```

```
rev(x) # reverse
```

```
## [1] 3 2 1
```

Character vectors, NA values, and sorting:

```
z <- c("b", "a", "c", NA)
z[z == "a"]
```

```
## [1] "a" NA
```

```
z[!is.na(z) & z == "a"]
```

```
## [1] "a"
```

```
z[is.na(z) | z == "a"]
```

```
## [1] "a" NA
```

```
is.na(z)
```

```
## [1] FALSE FALSE FALSE  TRUE
```

```
which(is.na(z))
```

```
## [1] 4
```

```
sort(z)
```

```
## [1] "a" "b" "c"
```

```
sort(z, na.last=TRUE)
```

```
## [1] "a" "b" "c" NA
```

Matrices and arrays:

```
(m <- matrix(1:12, 4, 3))
```

```
##      [,1] [,2] [,3]
## [1,]    1    5    9
## [2,]    2    6   10
## [3,]    3    7   11
## [4,]    4    8   12
```

```r
matrix(1:12, 4, 3, byrow=TRUE)
```

```
##      [,1] [,2] [,3]
## [1,]    1    2    3
## [2,]    4    5    6
## [3,]    7    8    9
## [4,]   10   11   12
```

```r
array(1:12, c(2, 2, 3))
```

```
## , , 1
##
##      [,1] [,2]
## [1,]    1    3
## [2,]    2    4
##
## , , 2
##
##      [,1] [,2]
## [1,]    5    7
## [2,]    6    8
##
## , , 3
##
##      [,1] [,2]
## [1,]    9   11
## [2,]   10   12
```

Attribues:

```r
dim(m)
```

```
## [1] 4 3
```

```r
dim(m) <- NULL
m
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10 11 12
```

```r
dim(m) <- c(4, 3)
m
```

```
##      [,1] [,2] [,3]
## [1,]    1    5    9
## [2,]    2    6   10
## [3,]    3    7   11
## [4,]    4    8   12
```

```
dimnames(m) <- list(letters[1:4], LETTERS[1:3])
m
```

```
##   A B  C
## a 1 5  9
## b 2 6 10
## c 3 7 11
## d 4 8 12
```

```
attributes(m)
```

```
## $dim
## [1] 4 3
##
## $dimnames
## $dimnames[[1]]
## [1] "a" "b" "c" "d"
##
## $dimnames[[2]]
## [1] "A" "B" "C"
```

Matrix indices:

```
m[1:2,]
```

```
##   A B  C
## a 1 5  9
## b 2 6 10
```

```
m[1,2]
```

```
## [1] 5
```

```
m[,2]
```

```
## a b c d
## 5 6 7 8
```

```
m[,2,drop=FALSE]
```

```
##   B
## a 5
## b 6
## c 7
## d 8
```

```
m[2]
```

```
## [1] 2
```

```r
m[rownames(m) == "c",]
```

```
##  A  B  C
##  3  7 11
```

```r
m[rownames(m) != "c",]
```

```
##   A B  C
## a 1 5  9
## b 2 6 10
## d 4 8 12
```

```r
m[rownames(m) %in% c("a", "c", "e"),]
```

```
##   A B  C
## a 1 5  9
## c 3 7 11
```

```r
m[!(rownames(m) %in% c("a", "c", "e")),]
```

```
##   A B  C
## b 2 6 10
## d 4 8 12
```

Lists and indexing:

```r
l <- list(m = m, x = x, z = z)
l
```

```
## $m
##   A B  C
## a 1 5  9
## b 2 6 10
## c 3 7 11
## d 4 8 12
##
## $x
## [1] 1 2 3
##
## $z
## [1] "b" "a" "c" NA
```

```r
l$ddd <- sqrt(l$x)
l[2:3]
```

```
## $x
## [1] 1 2 3
##
## $z
## [1] "b" "a" "c" NA
```

```
l[["ddd"]]
```

```
## [1] 1.000000 1.414214 1.732051
```

Data frames:

```
d <- data.frame(x = x, sqrt_x = sqrt(x))
d
```

```
##   x   sqrt_x
## 1 1 1.000000
## 2 2 1.414214
## 3 3 1.732051
```

Structure:

```
str(x)
```

```
##  num [1:3] 1 2 3
```

```
str(z)
```

```
##  chr [1:4] "b" "a" "c" NA
```

```
str(m)
```

```
##  int [1:4, 1:3] 1 2 3 4 5 6 7 8 9 10 ...
##  - attr(*, "dimnames")=List of 2
##   ..$ : chr [1:4] "a" "b" "c" "d"
##   ..$ : chr [1:3] "A" "B" "C"
```

```
str(l)
```

```
## List of 4
##  $ m  : int [1:4, 1:3] 1 2 3 4 5 6 7 8 9 10 ...
##   ..- attr(*, "dimnames")=List of 2
##   .. ..$ : chr [1:4] "a" "b" "c" "d"
##   .. ..$ : chr [1:3] "A" "B" "C"
##  $ x  : num [1:3] 1 2 3
##  $ z  : chr [1:4] "b" "a" "c" NA
##  $ ddd: num [1:3] 1 1.41 1.73
```

```
str(d)
```

```
## 'data.frame':    3 obs. of  2 variables:
##  $ x     : num  1 2 3
##  $ sqrt_x: num  1 1.41 1.73
```

```
str(as.data.frame(m))
```

```
## 'data.frame':    4 obs. of  3 variables:
##  $ A: int  1 2 3 4
##  $ B: int  5 6 7 8
##  $ C: int  9 10 11 12
```

```
str(as.list(d))
```

```
## List of 2
##  $ x     : num [1:3] 1 2 3
##  $ sqrt_x: num [1:3] 1 1.41 1.73
```

Summary:

```
summary(x)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##     1.0     1.5     2.0     2.0     2.5     3.0
```

```
summary(z)
```

```
##    Length     Class      Mode
##         4 character character
```

```
summary(m)
```

```
##        A               B               C
##  Min.   :1.00   Min.   :5.00   Min.   : 9.00
##  1st Qu.:1.75   1st Qu.:5.75   1st Qu.: 9.75
##  Median :2.50   Median :6.50   Median :10.50
##  Mean   :2.50   Mean   :6.50   Mean   :10.50
##  3rd Qu.:3.25   3rd Qu.:7.25   3rd Qu.:11.25
##  Max.   :4.00   Max.   :8.00   Max.   :12.00
```

```
summary(l)
```

```
##     Length Class  Mode
## m   12     -none- numeric
## x    3     -none- numeric
## z    4     -none- character
## ddd  3     -none- numeric
```

```
summary(d)
```

```
##        x           sqrt_x
##  Min.   :1.0   Min.   :1.000
##  1st Qu.:1.5   1st Qu.:1.207
##  Median :2.0   Median :1.414
##  Mean   :2.0   Mean   :1.382
##  3rd Qu.:2.5   3rd Qu.:1.573
##  Max.   :3.0   Max.   :1.732
```

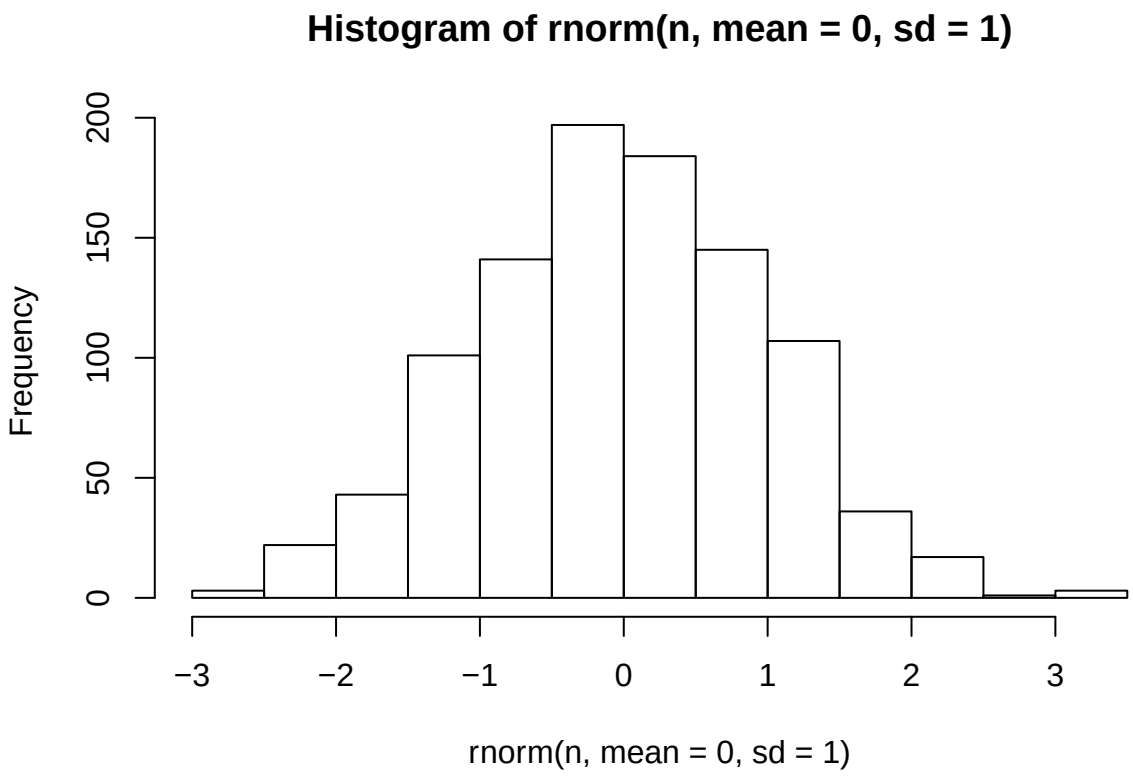
4.1.1 Key concepts

- a matrix is a vector with `dim` attribute, elements are in same mode,
- a data frame is a list where length of elements match and elements can be in different mode.

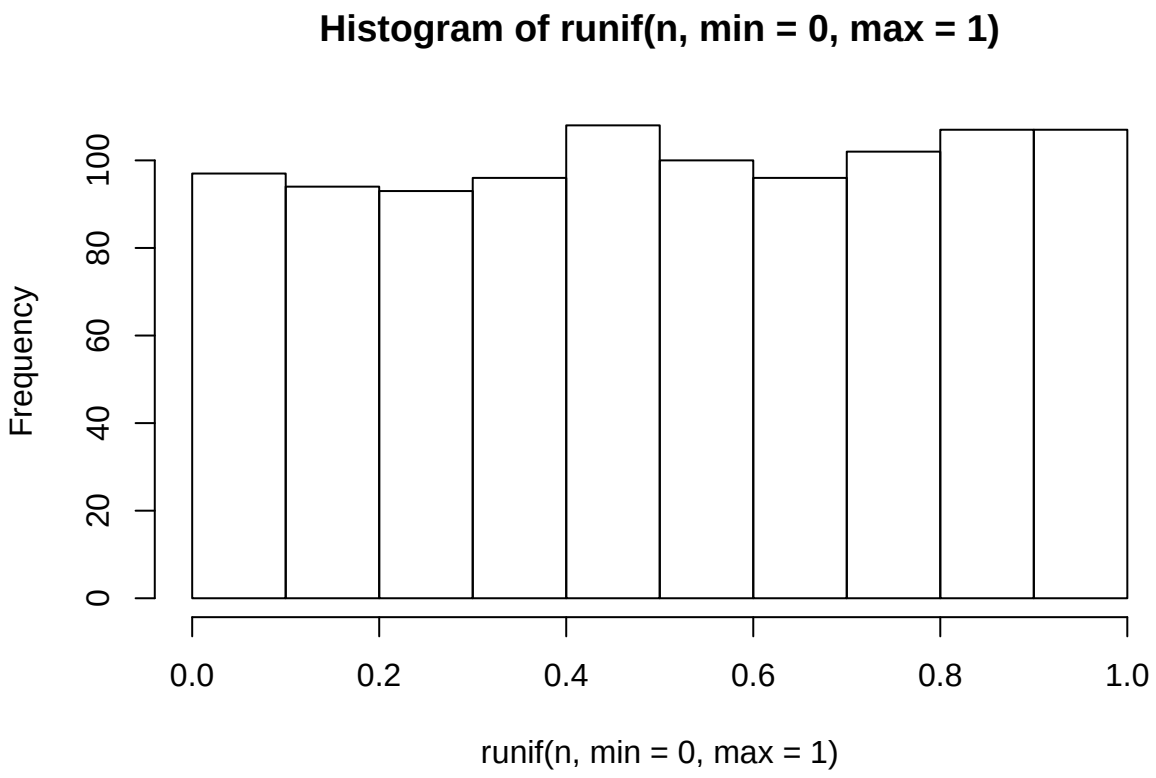
4.2 Random numbers

Random numbers can be generated from a distribution using the `r*` functions where the wildcard (`*`) stands for the abbreviated name of the distribution, for example `rnorm`.

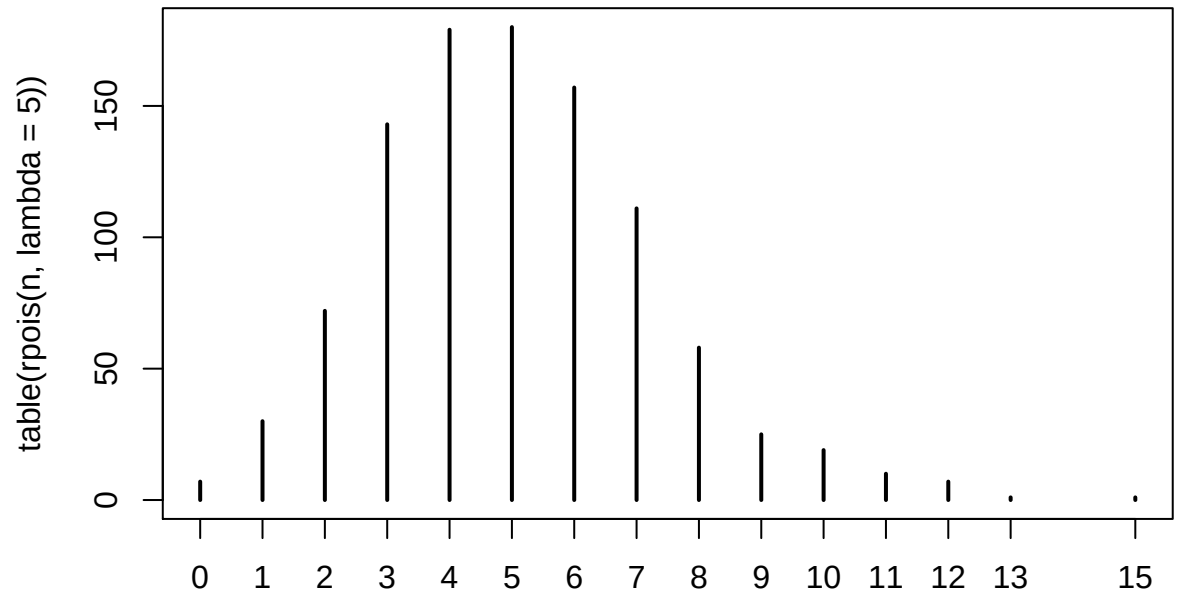
```
n <- 1000
## draw n random numbers from Normal(0, 1)
hist(rnorm(n, mean = 0, sd = 1))
```



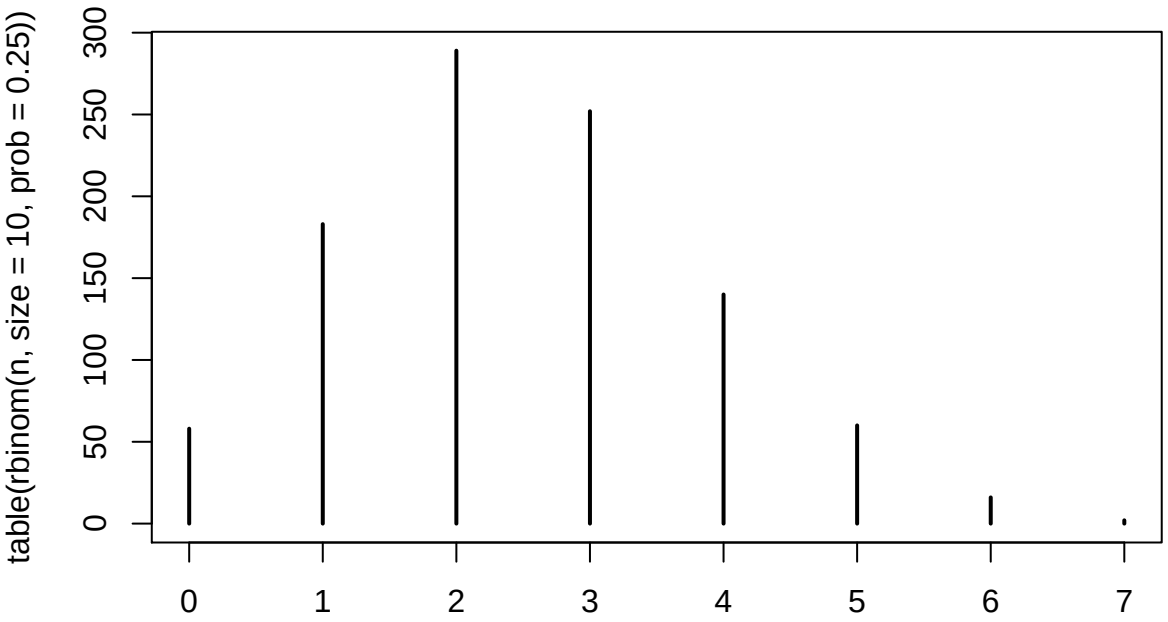

```
## Uniform (not as 'run-if' but as 'r-unif')
hist(runif(n, min = 0, max = 1))
```




```
## Poisson
plot(table(rpois(n, lambda = 5)))
```




```
## Binomial
plot(table(rbinom(n, size = 10, prob = 0.25)))
```



4.3 MCMC list objects

The coda R package defines MCMC list objects as:

- a list where elements are matrices of identical dimensions,
- each list stores posterior sample from an MCMC chain, thus the length of the `mcmc.list` object equals the number of parallel chains,
- each matrix has the dimensions: number of samples (defined by iterations and thinning value), and the number of variables monitored.
- the matrices have some attributes attached to them storing info about the MCMC parameters (start iteration the end iteration and the thinning interval of the chain).

For example if we are monitoring 2 variables, a normally and a uniformly distributed one, the structure might look like this:

```
mcmc <- replicate(3,
    structure(cbind(a = rnorm(n), b = runif(n)),
        mcpar = c(1, n, 1), class = "mcmc"),
    simplify = FALSE)
class(mcmc) <- "mcmc.list"
str(mcmc)
```

```
## List of 3
##  $ : mcmc [1:1000, 1:2] 0.856 2.207 0.071 1.031 -3.045 ...
##   ..- attr(*, "dimnames")=List of 2
##   .. ..$ : NULL
##   .. ..$ : chr [1:2] "a" "b"
##   ..- attr(*, "mcpar")= num [1:3] 1 1000 1
##  $ : mcmc [1:1000, 1:2] 0.611 1.626 0.139 0.792 0.981 ...
##   ..- attr(*, "dimnames")=List of 2
##   .. ..$ : NULL
```

```
##   .. ..$ : chr [1:2] "a" "b"
##   ..- attr(*, "mcpar")= num [1:3] 1 1000 1
##  $ : mcmc [1:1000, 1:2] -0.558 -1.683 0.535 0.333 1.245 ...
##   ..- attr(*, "dimnames")=List of 2
##   .. ..$ : NULL
##   .. ..$ : chr [1:2] "a" "b"
##   ..- attr(*, "mcpar")= num [1:3] 1 1000 1
##  - attr(*, "class")= chr "mcmc.list"
```

Some basic methods for such `mcmc.list` objects are defined in the `coda` and `dclone` packages:

```
library(dclone)
```

```
## Loading required package: coda
```

```
## Loading required package: parallel
```

```
## Loading required package: Matrix
```

```
## dclone 2.1-1      2016-01-11
```

```
summary(mcmc)
```

```
##
## Iterations = 1:1000
## Thinning interval = 1
## Number of chains = 3
## Sample size per chain = 1000
##
## 1. Empirical mean and standard deviation for each variable,
##    plus standard error of the mean:
##
##      Mean     SD Naive SE Time-series SE
## a 0.01913 1.0162 0.018554       0.018698
## b 0.50270 0.2897 0.005289       0.005141
##
## 2. Quantiles for each variable:
##
##       2.5%     25%      50%    75%  97.5%
## a -1.93813 -0.6633 0.006205 0.7251 2.0085
## b  0.02358  0.2470 0.507300 0.7567 0.9744
```

```
str(as.matrix(mcmc))
```

```
##  num [1:3000, 1:2] 0.856 2.207 0.071 1.031 -3.045 ...
##  - attr(*, "dimnames")=List of 2
##   ..$ : NULL
##   ..$ : chr [1:2] "a" "b"
```

```
varnames(mcmc)
```

```
## [1] "a" "b"
```

```
start(mcmc)
```

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

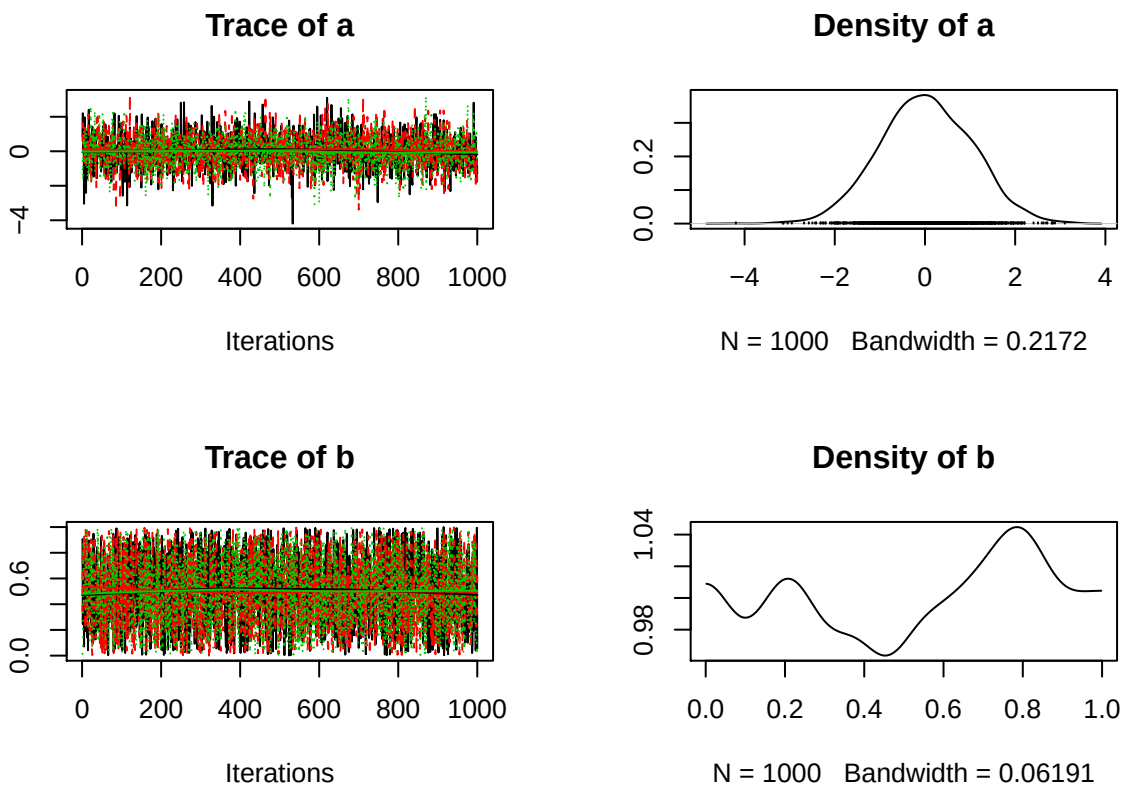
```
end(mcmc)
```

```
## [1] 1000
```

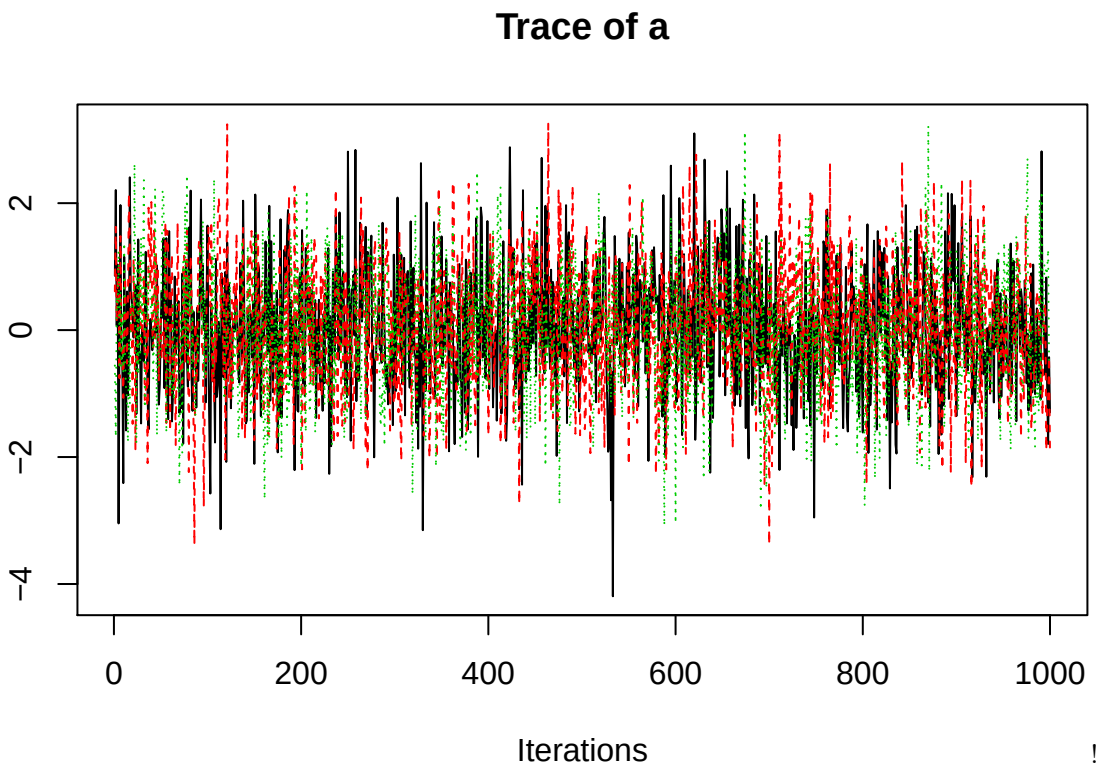
```
thin(mcmc)
```

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

```
plot(mcmc)
```

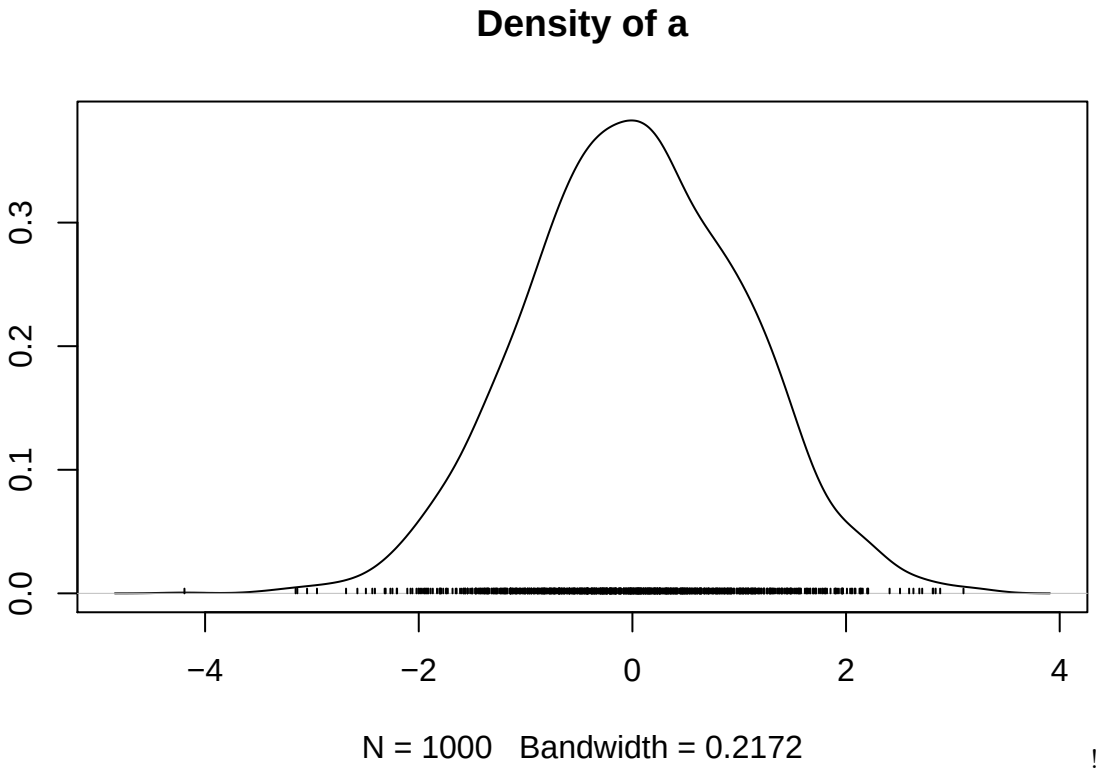



```
traceplot(mcmc)
```



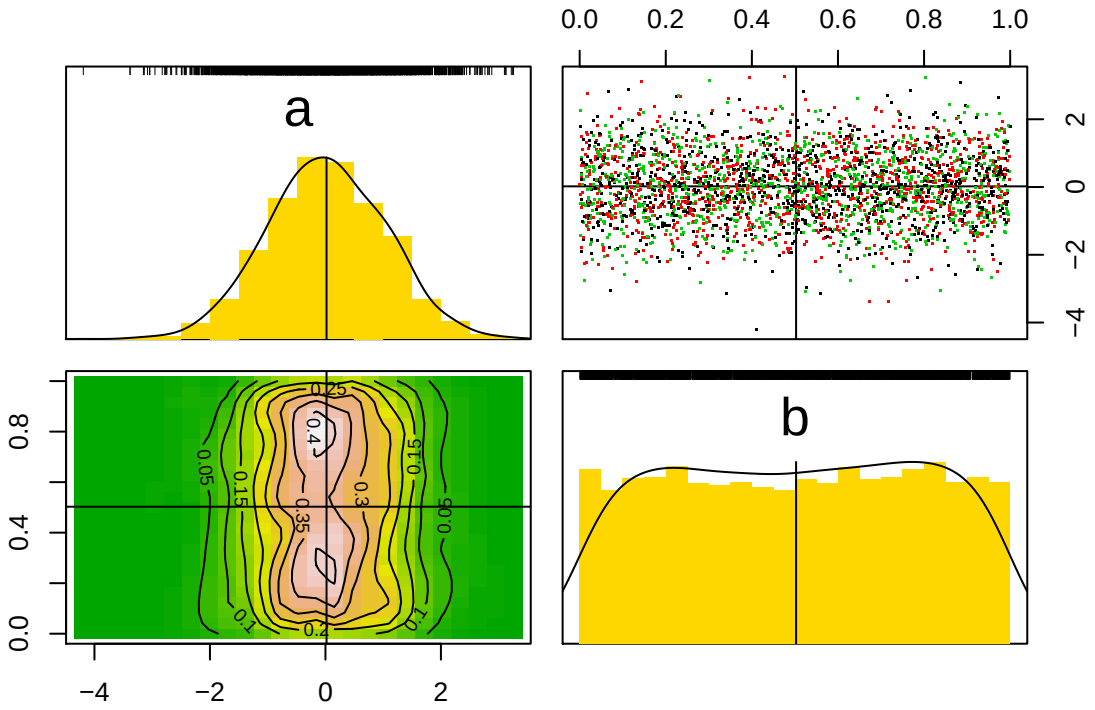
!

```
densplot(mcmc)
```



!

```
pairs(mcmc)
```



5 Statistical concepts

5.1 Introduction

Science, as we envision it, is an interplay between inductive and deductive processes. Francis Bacon, the father of what is known as the scientific method, emphasizes the roles of observations, alternative explanations and tests to choose among various explanations. Bacon saw science as inductive process, moving from the particular to the general.

Karl Popper proposed the doctrine of falsification, which defines what is acceptable as a scientific hypothesis: if a statement cannot be falsified, then it is *not* a scientific hypothesis. This is intrinsically a deductive process. What is common to these different views is that theories need to be probed to assess their correctness. Observations play an important role in such probing.

In most scientific situations, we are interested in understanding the natural processes that have given rise to the observations. Such understanding generally leads to prediction and possibly control of the processes. Traditionally, we formulate our understanding of the processes in terms of mathematical models. These mathematical models may be deterministic or may be stochastic.

It is widely accepted, at least by the statisticians, that stochastic models represent nature more effectively than pure deterministic models. Aside from the natural stochasticity in the system, the observations themselves might have measurement error making it necessary to consider stochastic models to model observations.

One of the consequences of stochastic models is that Popper's theory of falsification does not strictly apply. No data are strictly inconsistent with a stochastic model, except in artificial situations or trivially improper models. Thus, one can only say that the observed data are more likely under one model than the other; or that the strength of evidence for one hypothesis is larger than for an alternative. We cannot outright accept or reject a hypothesis.

Given a set of stochastic models (or, equivalently a set of alternative descriptions of the underlying process), the goal of statistical inference is to choose the model that is best supported by the data. Thus, statistical inference is both deductive (it makes some predictions) and inductive (data determines which model is best supported). An important feature that we demand from all our statistical inference procedures is that with infinite amount of information, the probability of choosing the correct model converges to one.

Another important feature of statistical inference is that it is uncertain. We want to know whether or not our inferential statement, namely the choice of the model, is trustworthy. Quantifying the uncertainty in the inferential statements is a critical issue and has led to different statistical philosophies of inference, in particular the frequentist philosophy and the Bayesian philosophy. Just as numbers without understanding the units are meaningless, statistical inferential statements without proper understanding of the uncertainty are meaningless.

We will discuss the differences in the two approaches to quantify uncertainty in the statistical inference in detail in the context of a simple example later. For the interested researcher, there are several resources available that discuss these issues in depth.

We also do not intend to give a detailed tutorial on the basics of statistical inference. There are many standard reference books for such introduction.

5.2 A simple example

Let us start with a simple occupancy model. We will use this model to introduce various important concepts that will be used throughout the course. We will use it also to introduce some basic commands for analyzing data using the R package dclone.

In conservation biology, one of the first things we want to do is monitor the current status of the population. This can be done in terms of simple presence-absence data answering the question: what is the proportion of occupied sites? If this proportion is high, it may imply that we should not worry too much about the species (if it is something we want to maintain) or may be we want to do some biological control (if it is an invasive species). A simple monitoring procedure would consist of the following steps:

1. Divide the study area into quadrats of equal area. Suppose there are N such quadrats.
2. Take a simple random sample of size n from these.
3. Visit these sites and find out if it is occupied by the species or not.

5.3 Assumptions

It is critical that we state the assumptions underlying the statistical model. In practice, however, we may or may not be able to know whether all the assumptions are fulfilled or not.

1. Quadrats are identical to each other.
2. Occupancy status of one quadrat does not depend on the status of other quadrats.

Mathematically we write this as follows:

$$Y_i \sim Binomial(1, p)$$

(this is also known as the Bernoulli distribution) are independent, identically distributed (*i.i.d.*) random variables.

- Observed data: $ Y_{1}, Y_{2}, ..., Y_{n} $
- Unobserved data: $ Y_{n+1}, Y_{n+2}, ..., Y_{N} $

The probability mass function of the Bernoulli random variable is written as: $P(Y = y) = p^y(1-p)^{1-y}$, where $p \in (0, 1)$ and $y = 0, 1$.

We can now write down the likelihood function. This is proportional to the probability of observing the data at hand:

$$L(p; y_1, y_2, \ldots, y_n) = \prod_{i=1}^{n} p^{y_i}(1-p)^{1-y_i}$$

We take product because observations are assumed to be independent of each other.

5.4 Important properties of likelihood

- Likelihood is a function of the parameter.
- Data are fixed.
- Likelihood is *not* a probability of the parameter taking a specific value. It represents the following quantity: If the parameter is $p = p^*$, then the probability of observing the data at hand is $L(\tilde{p}; y_1, y_2, \ldots, y_n) = \prod_{i=1}^{n} \tilde{p}^{y_i}(1-\tilde{p})^{1-y_i}$.

To demonstrate this we simulate a single data set and vary the parameter value and get a function as represented below:

```
## random numbers from Binomial distribution
## Binomial with size=1 is a Bernoulli distribution
## p value set as 0.3
p <- 0.3
set.seed(1234) # set random seed for reproducibility
(y <- rbinom(n = 1000, size = 1, p = p))
```

```
##    [1] 0 0 0 0 1 0 0 0 0 0 0 0 0 1 0 1 0 0 0 0 0 0 0 0 0 1 0 1 1 0 0 0 0 0
##   [35] 0 1 0 0 1 1 0 0 0 0 0 0 0 0 0 1 0 0 1 0 0 0 0 1 0 1 1 0 0 0 0 1 0 0
##   [69] 0 0 0 1 0 1 0 0 0 0 0 0 0 1 0 0 0 0 0 1 0 0 0 1 0 1 0 0 0 0 0 0 0 1 0 0
##  [103] 0 0 0 0 0 0 0 0 1 0 1 0 0 1 1 0 0 1 1 1 1 1 0 0 0 0 0 0 1 0 0 0 1 0
##  [137] 1 0 0 1 0 1 0 0 0 0 1 0 1 0 0 0 0 1 0 1 0 1 0 0 0 0 0 0 0 0 0 0 1 0
##  [171] 1 0 1 0 0 1 0 0 0 0 0 0 0 0 1 0 1 0 0 1 0 1 0 1 1 1 1 1 1 0 0 0 0 1
##  [205] 0 1 0 0 0 1 0 0 0 1 0 1 0 1 0 1 0 0 0 0 0 0 1 0 0 0 1 0 0 1 0 0 1 0
##  [239] 0 1 0 0 0 0 1 0 1 0 1 1 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 1 0 0 1 0 1
##  [273] 0 1 0 0 1 0 0 0 0 0 1 1 1 0 0 0 0 1 0 0 1 0 0 1 0 0 0 0 0 1 0 0 1 0
##  [307] 0 1 0 1 1 0 0 0 0 0 0 0 0 0 1 0 1 0 0 0 0 0 0 0 1 1 1 1 0 0 0 0 0 1 0
##  [341] 1 0 0 1 0 0 0 0 1 0 0 1 0 1 1 1 0 1 0 1 0 0 0 0 1 1 0 0 1 1 1 0 1 0
##  [375] 0 0 1 0 0 0 0 0 0 0 0 0 0 0 1 1 0 1 1 0 0 1 0 0 0 0 0 0 1 1 0 0 1 0
##  [409] 0 0 1 1 1 0 1 0 0 1 0 0 0 1 0 0 1 0 0 0 0 0 0 1 0 1 0 0 0 1 1 0 1 1
##  [443] 1 0 1 0 1 0 0 0 0 1 1 1 1 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 1 0 1
##  [477] 0 0 1 0 0 0 0 0 0 0 1 1 1 0 0 1 1 0 0 0 0 0 0 0 0 1 1 0 0 1 1 1 0 1
##  [511] 0 1 1 0 0 1 0 0 0 0 1 0 0 1 0 1 0 0 0 0 0 0 0 1 0 1 0 0 1 0 0 0 1 0
##  [545] 1 0 1 1 1 1 0 0 1 0 1 1 0 0 0 1 0 1 0 1 0 1 1 0 0 0 0 1 0 0 0 0 0 0
##  [579] 0 1 1 0 0 0 0 0 0 1 1 0 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 1
##  [613] 0 0 1 0 0 0 1 0 1 0 0 1 0 1 0 1 0 0 1 0 0 0 1 1 0 1 0 0 0 0 0 1 0 0
##  [647] 0 1 0 1 0 0 1 0 0 0 1 0 0 1 0 1 1 0 0 0 1 0 0 1 0 1 0 0 1 1 0 1 0 1
##  [681] 0 0 0 0 1 0 1 0 0 0 1 0 0 0 0 0 0 0 0 1 0 0 1 0 0 0 0 0 0 0 1 1 1 0
##  [715] 0 0 0 0 1 0 1 0 1 0 0 1 0 0 0 1 1 0 0 0 0 1 0 1 1 1 0 0 0 0 0 0 0 0
##  [749] 0 0 0 1 0 1 0 1 0 1 0 0 0 0 0 0 0 0 1 0 1 0 1 1 1 0 0 0 0 0 0 0 1 0
```

```
##  [783] 0 0 0 0 1 0 1 0 1 0 0 0 1 0 0 1 0 1 0 1 0 0 0 0 0 0 0 0 0 0 1 0 0 0
##  [817] 0 1 0 0 0 0 0 0 0 1 0 0 0 0 0 0 1 1 1 0 0 1 0 0 0 0 0 0 0 0 1 0 1 1 1
##  [851] 0 1 1 0 0 0 0 0 1 1 0 0 0 0 0 0 1 0 1 0 0 1 1 1 0 1 0 0 0 0 0 0 0 0 0
##  [885] 0 1 0 0 0 0 1 1 0 0 0 0 0 0 1 1 0 0 1 0 1 0 0 0 1 0 0 0 0 0 0 0 0 1
##  [919] 1 1 0 0 0 0 0 0 1 0 0 0 0 1 0 0 0 0 0 0 0 0 1 0 1 1 0 0 0 0 0 0 1 1
##  [953] 0 1 1 0 1 0 0 0 0 0 1 1 0 0 0 0 0 0 1 1 0 1 0 0 1 0 1 0 1 1 0 0 1 0
##  [987] 0 1 0 0 0 0 1 0 0 0 1 0 1 0
```

```
y1 <- y[1:10] # take only the 1st 10 elements of y

## pt is our p value that we want the Likelihood to be calculated for
pt <- 0.3
## the Likelihood is based on the formula from above
(L <- prod(pt^y1 * (1 - pt)^(1-y1)))
```

```
## [1] 0.01210608
```

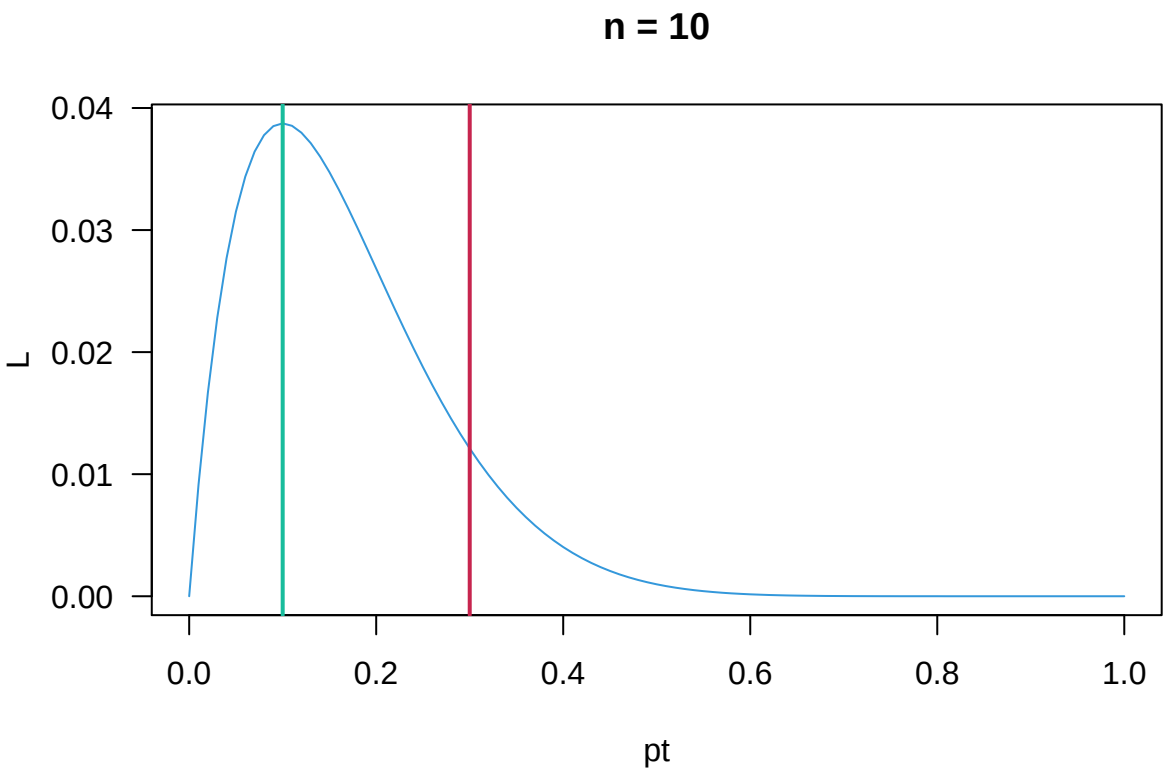
```
## the following statement is equivalent to typing in the formula
## take advantage of bilt-in density functions
prod(dbinom(y1, size = 1, prob = pt))
```

```
## [1] 0.01210608
```

```
## now pt is a vector between 0 and 1
pt <- seq(0, 1, by = 0.01)
## use the sapply function to calculate the likelihood
## using one element of the vector at a time (argument z becomes prob=z)
## by fixing the data y1
L <- sapply(pt, function(z) prod(dbinom(y1, size = 1, prob = z)))
```

Now that we calculated the likelihood function, let us plot it:

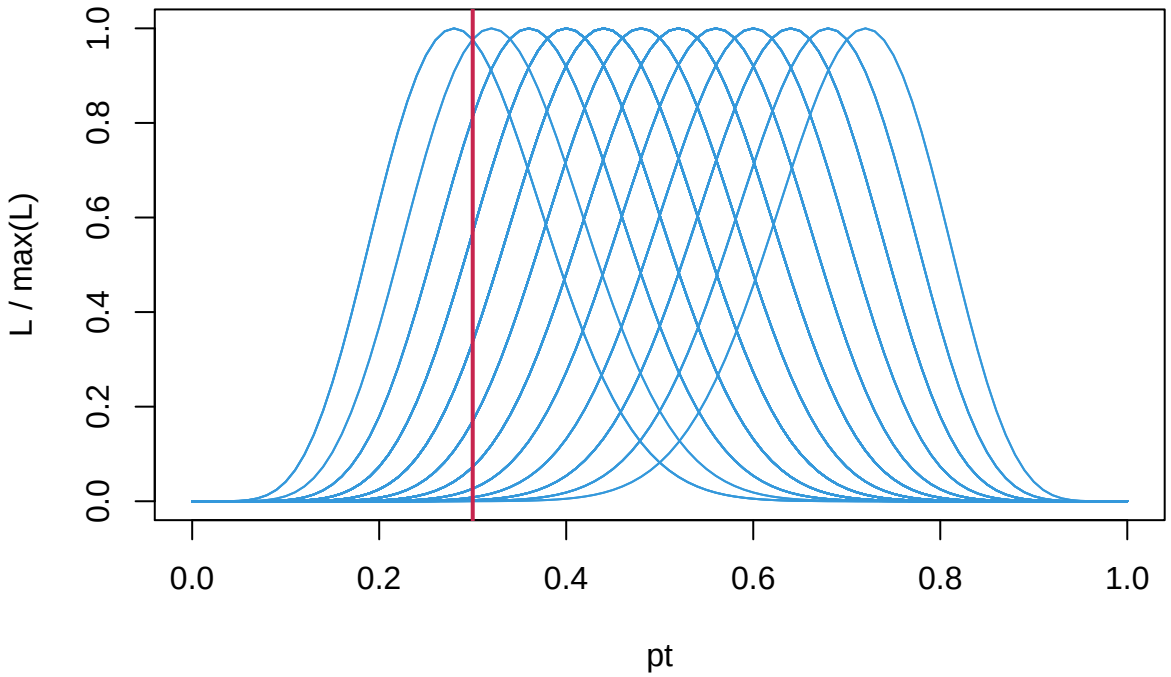
```
op <- par(las=1) # always horizontal axis, store old settings in op
## color palettes for nice figures
flatly <- list(
    "red"="#c7254e",
    "palered"="#f9f2f4",
    "primary"="#2c3e50",
    "success"="#18bc9c",
    "info"="#3498db",
    "warning"="#f39c12",
    "danger"="#e74c3c",
    "pre_col"="#7b8a8b",
    "pre_bg"="#ecf0f1",
    "pre_border"= "#cccccc")
dcpal_reds <- colorRampPalette(c("#f9f2f4", "#c7254e"))
dcpal_grbu <- colorRampPalette(c("#18bc9c", "#3498db"))
## now we plot the Likelihood function
plot(pt, L, type = "l", col = flatly$info,
    main = paste("n =", length(y1)))
abline(v = p, lwd = 2, col = flatly$red) # true value
abline(v = pt[which.max(L)], lwd = 2, col = flatly$success) # ML extimate
```



> As we change the data, the likelihood function changes.

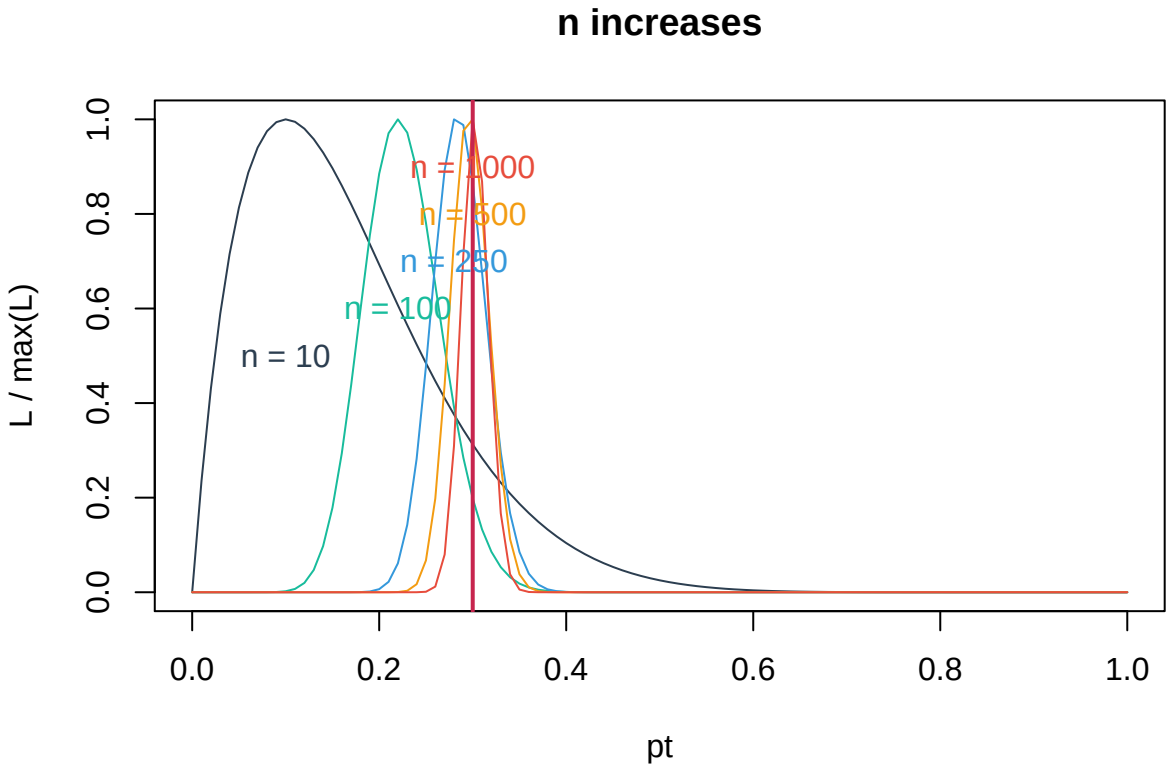
The following code is to demonstrate how the likelihood function changes when we change the data that was simulated under the same parameter values. We keep everything else (e.g. sample size) the same.

```
## function f has a single argument, n: sample size
f <- function(n) {
    y <- rbinom(n = n, size = 1, p = 0.5)
    L <- sapply(pt, function(z) prod(dbinom(y, size = 1, prob = z)))
    L / max(L)
}
## create a blank plot
plot(0, type = "n", main = "n constant, y changes",
     ylim = c(0, 1), xlim = c(0, 1),
     xlab = "pt", ylab = "L / max(L)")
## we simulate an n=25 data set 10 times and
## plot the scaled likelihood function [L / max(L)]
tmp <- replicate(100,
    lines(pt, f(25),
    col = flatly$info))
abline(v = p, lwd = 2, col = flatly$red)
```



As we increase the sample size, the likelihood becomes concentrated around the true value. This property of the maximum likelihood estimator is called *consistency*.

```
## try different sample sizes, data is fixed
## so samples can be nested
nvals <- c(10, 100, 250, 500, 1000)
## scaled likelihood function using different sample sizes
Lm <- sapply(nvals,
    function(n) {
        L <- sapply(pt, function(z)
            prod(dbinom(y[1:n], size = 1, prob = z)))
        L / max(L)
    })
## plot the results
matplot(pt, Lm, type = "l",
    lty = 1, ylab = "L / max(L)", main = "n increases",
    col = unlist(flatly)[3:7])
abline(v = p, lwd = 2, col = flatly$red)
text(apply(Lm, 2, function(z) pt[which.max(z)]),
    0.5+ 0.1*c(0:4), paste("n =", nvals),
    col = unlist(flatly)[3:7])
```



The likelihood value represents the support in the data for a particular parameter value. This is intrinsically a relative concept. How much more support do we have for this parameter value vs. another parameter value. The **likelihood ratio** is a more fundamental concept than the likelihood function itself..

We can now summarize the goals of statistical inference:

1. Given these data, what is the strength of evidence for one hypothesis over the other hypothesis?
2. Given these data, how do we change our beliefs?
3. Given these data, what decision do we make?

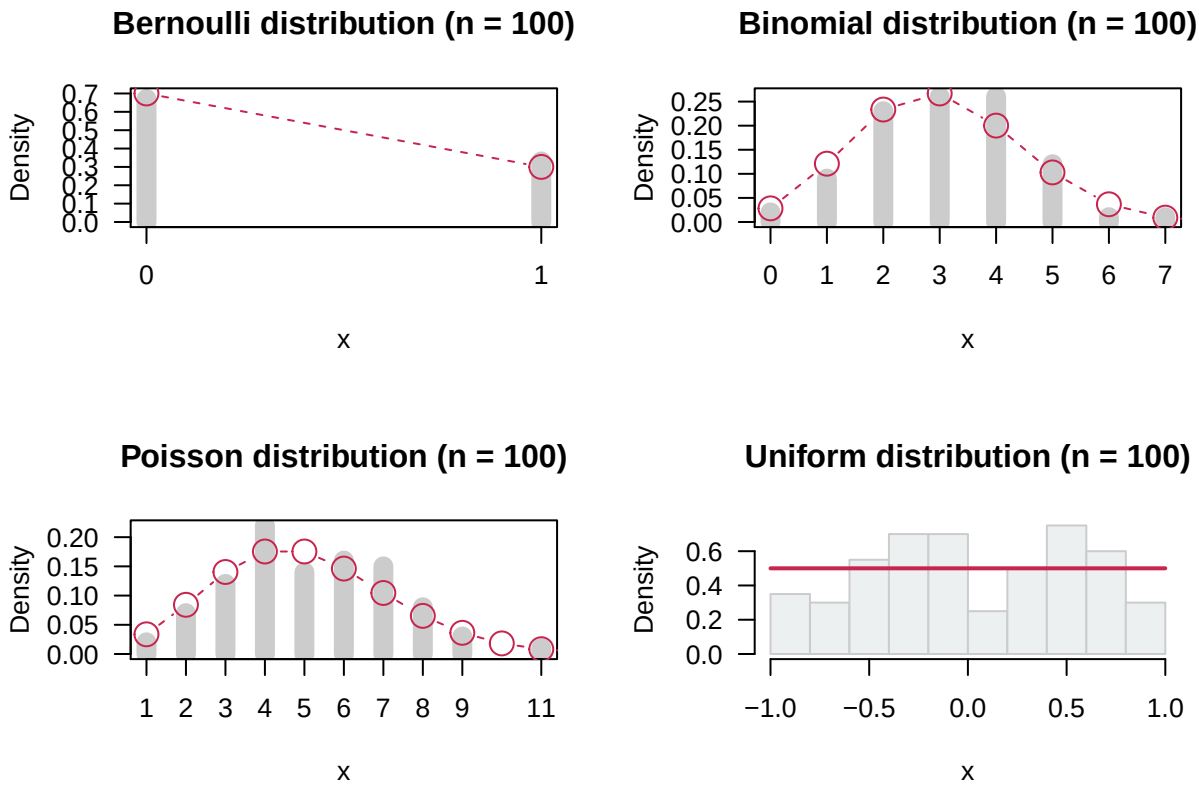
Check out these following functions and play around to see how sample size affects the likelihood function. In order to access them, easiest is to install a half baked (i.e. not yet well documented) package from Github:

```
library(devtools)
install_github("datacloning/dcapps")
```

5.4.1 Commonly used statistical distributions

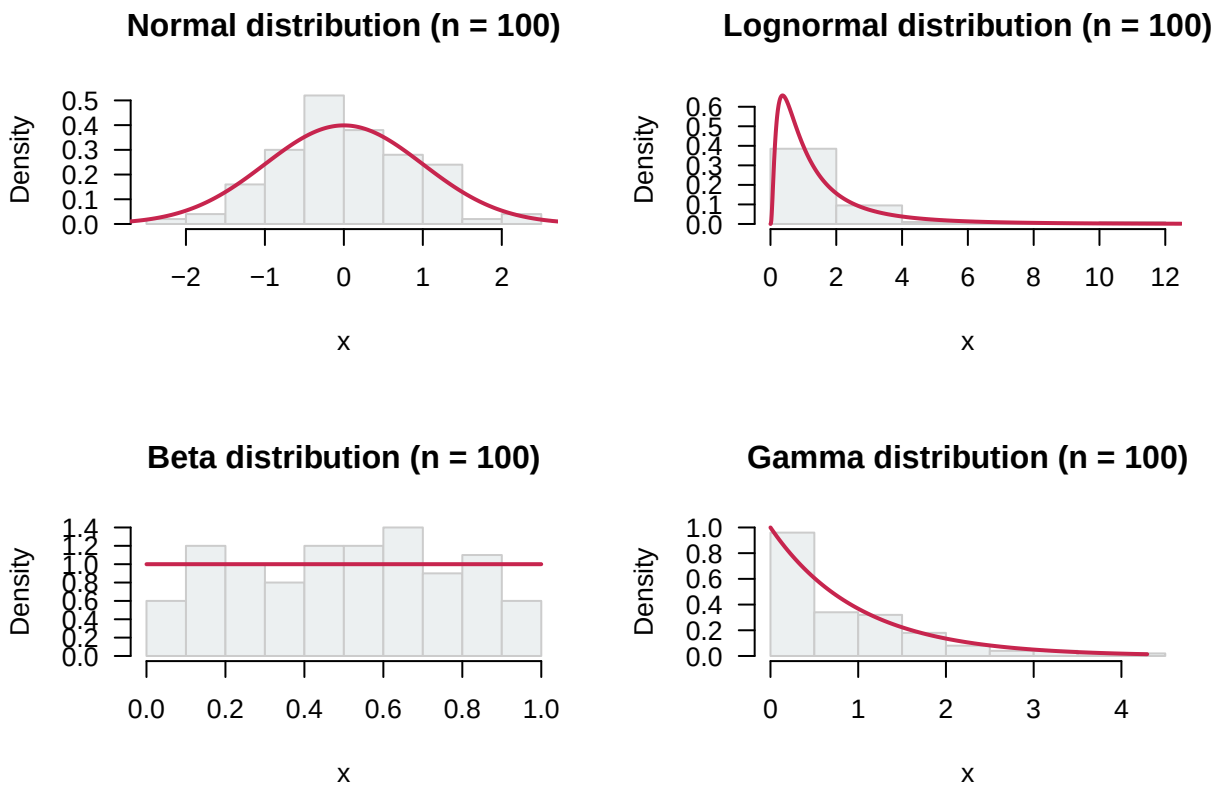
Play with the parameter settings for these distributions:

```
library(dcapps)
op <- par(mfrow = c(2, 2))
distr("Bernoulli", binom_p = 0.3)
distr("Binomial", binom_p = 0.3, binom_size = 10)
distr("Poisson", poisson_lambda = 5)
distr("Uniform", unif_a = -1, unif_b = 1)
```




```
par(op)

op <- par(mfrow = c(2, 2))
distr("Normal", normal_mu = 0, normal_var = 1)
distr("Lognormal", normal_mu = 0, normal_var = 1)
distr("Beta", beta_shape1 = 1, beta_shape2 = 1)
distr("Gamma", gamma_shape = 1, gamma_rate = 1)
```

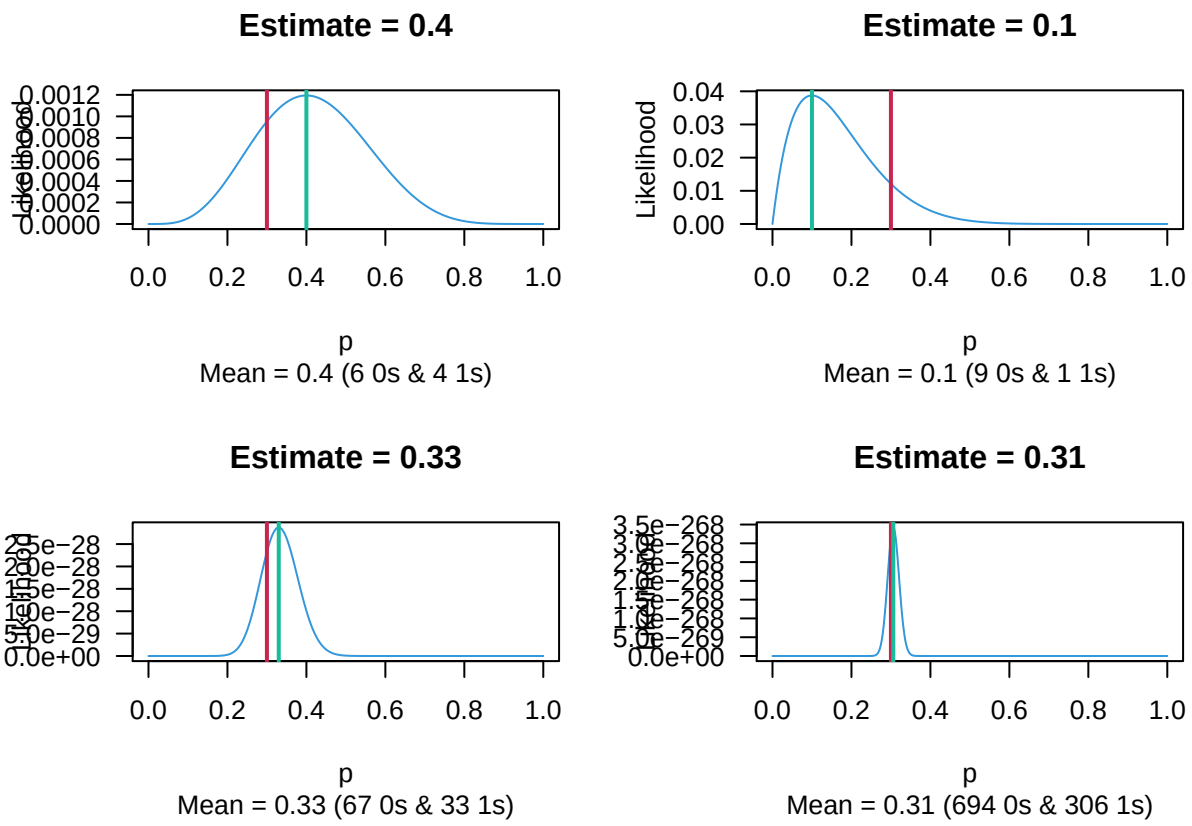



```
par(op)
```

5.4.2 Maximum likelihood estimator

Vary sample size (`n`) and random seed (`seed`) to see how that impacts the maximum likelihood estimator (MLE):

```
op <- par(mfrow = c(2, 2))
mle(p = 0.3, n = 10, seed = 0)
mle(p = 0.3, n = 10, seed = 1234)
mle(p = 0.3, n = 100, seed = 0)
mle(p = 0.3, n = 1000, seed = 0)
```




```
par(op)
```

5.5 The maximum likelihood estimator

Which parameter value has the largest support in the data?

We can use numerical optimization to get the value of a parameter where the likelihood function is maximal. Such a parameter value is called a (point) estimate, while the function we are using to do the estimation (in this case the likelihood function, but there might be other functions, too) is called an estimator.

In numerical optimization, we often find the minimum of the negative of a function, instead of finding the maximum. Also, we use the log likelihood, because the product becomes a sum on the log scale. This is much easier to compute. That is why we ofteh find that programs define the negative log likelihood function as we do below.

```
## this functions simulates observations
sim_fun <- function(n, p) {
    rbinom(n = n, size = 1, p = p)
}
## this function returns the negative log likelihood value
nll_fun <- function(p, y) {
    -sum(dbinom(y, size = 1, prob = p, log = TRUE))
}
```

We use $n = 100$ and $p = 0.5$ for simulating the observation vector y. Then use the one dimensional optimization function, `optimize`. (For multivariate optimization problems, see the `optim` function.)

What is different between using optimization vs. manually setting up a set of values is that optimization starts with a sparse grid first to see what region of the parameter space is of interest. In this region then the search for the minimum (or maximum) is continued with more intensity, i.e. until the difference in subsequent candidate estimates reaches a pre defines tolerance threshold (`tol` argument in `optimize`).

```
n <- 100
p <- 0.3
y <- sim_fun(n, p)
optimize(nll_fun, interval = c(0, 1), y = y)
```

```
## $minimum
## [1] 0.2699928
##
## $objective
## [1] 58.32588
```

Once we can write down the likelihood, we can in principle write a program to calculate the value of the (negative log) likelihood function given some parameter value and the data.

5.6 The sampling distribution of the estimates

Let us revisit now what happened when we kept the sample size fixed but changed the data. In this case, we get different parameter estimates (MLEs) for different data sets. A natural question to ask would be: How much would the answers vary if we have different samples?

In the following program we pre-allocate a vector of length B, we simulate the data B times and store the corresponding MLEs in the object `res`:

```
B <- 1000
res <- numeric(B)
for (i in 1:B) {
    y <- sim_fun(n, p)
    res[i] <- optimize(nll_fun, interval = c(0, 1), y = y)$minimum
}
```

Some summary statistics reveal interesting things: the B estimates now have a *sampling distribution* that can be characterized by its mean and various percentiles:

```
summary(res)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##  0.1700  0.2700  0.3000  0.3009  0.3300  0.4200
```

```
quantile(res, c(0.025, 0.975))
```

```
##      2.5%     97.5%
## 0.2199862 0.4000015
```

5.6.1 Bias and consistency

The bias is defined as the deviation between the estimate and the true parameter values. When the bias converges to 0 with increasing sample size, we say that an estimator is consistent:

```
mean(res - p)
```

```
## [1] 0.0008566297
```

Precision is the corresponding feature of an estimate.

5.7 Confidence interval and efficiency

The 2.5% and 97.5% percentiles of the sampling distribution correspond to the 95% analytical confidence intervals around the true parameter value.

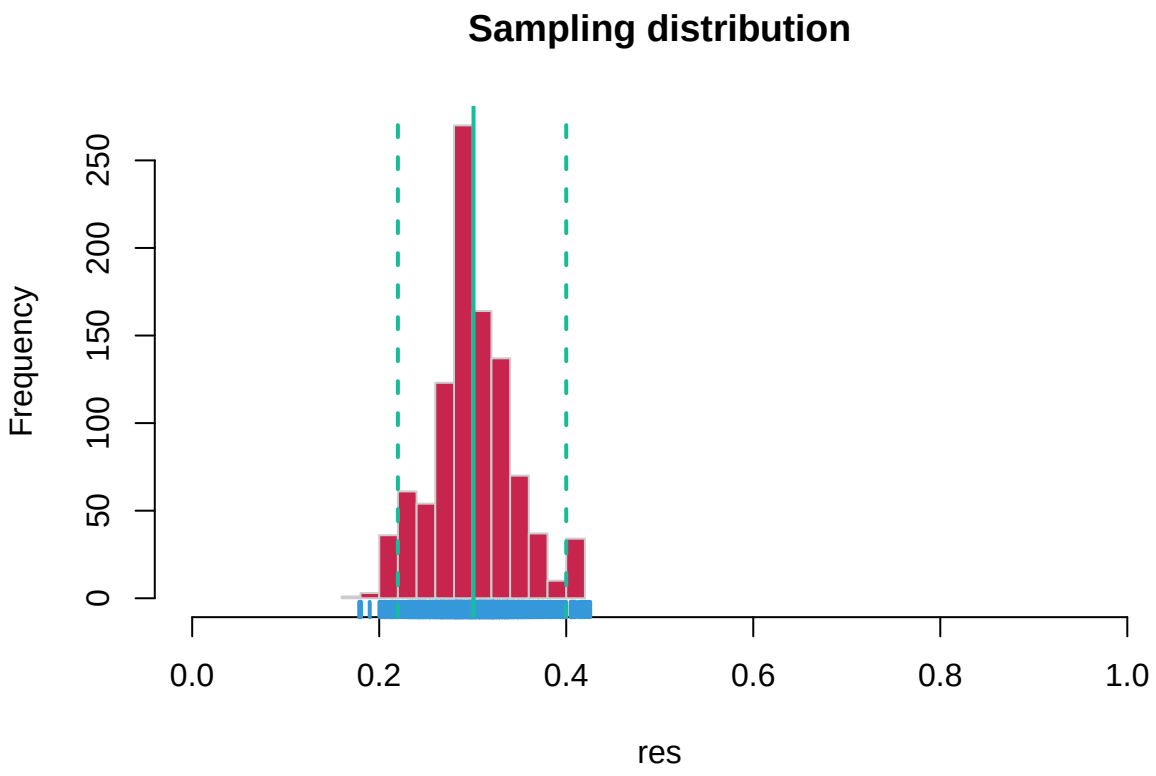
```
level <- 0.95
a <- (1 - level) / 2
a <- c(a, 1 - a)
(ci0 <- quantile(res, a))
```

```
##      2.5%     97.5%
## 0.2199862 0.4000015
```

An estimator is called efficient when the variation in the sampling distribution and the confidence interval gets smaller with increasing sample size. Accuracy is the corresponding feature of an estimate. When the percentiles of the sampling distribution are close to the corresponding analytical confidence intervals, we say that the estimator has nominal coverage.

The following plot shows the sampling distribution of the estimates, the true parameter value, the mean of the estimates, the analytical confidence intervals and the quantiles of the sampling distribution. The values overlap perfectly, that is why the red lines are not visible:

```
hist(res, col = flatly$red, border = flatly$pre_border,
    xlim = c(0, 1), main = "Sampling distribution")
rug(res+runif(B, -0.01, 0.01), col = flatly$info, lwd = 2)
## sampling distribution based sumarry statistics
abline(v = mean(res), lwd = 2, col = flatly$success)
abline(v = quantile(res, c(0.025, 0.975)),
    lwd = 2, col = flatly$success, lty = 2)
```



5.8 Estimated confidence intervals

Of course, in real life, we do not have the luxury of conducting such repeated experiments. So what good are these ideas?

One way to quantify the uncertainty in the estimate is to use asymptotic confidence intervals as we saw above. We called it analytical because for this particular model we could calculate it analytically. This, however, might not be the case in all situation. One can estimate the asymptotic standard error and confidence interval of an estimate:

```
## our data
y <- sim_fun(n, p)
## MLE
(est <- optimize(nll_fun, interval = c(0, 1), y = y)$minimum)
```

```
## [1] 0.3399919
```

```
(ci1 <- qnorm(a, mean = est,
    sd = sqrt(est * (1-est) / n)))
```

```
## [1] 0.2471472 0.4328366
```

We have the MLE. The MLE is kind of close to the true parameter value. So suppose we pretend as if the MLE is the true parameter value, we can get the sampling distribution and the confidence interval. This is the idea behind the parametric bootstrap confidence intervals.

```
B <- 1000
pbres <- numeric(B)
for (i in 1:B) {
    yb <- sim_fun(n, est) # treat est as 'true' value and estimate
    pbres[i] <- optimize(nll_fun, interval = c(0, 1), y = yb)$minimum
}
(ci2 <- quantile(pbres, a))
```

```
##      2.5%     97.5%
## 0.2500143 0.4299993
```

The non-parametric bootstrap is based on a similar principle, but instead of simulating data sets under our initial estimate, we mimic the experiment by resampling the original data set with replacement B times:

```
## we use the same settings and data as for non-parametric bootstrap
npbres <- numeric(B)
for (i in 1:B) {
    yb <- sample(y, replace = TRUE)
    npbres[i] <- optimize(nll_fun, interval = c(0, 1), y = yb)$minimum
}
(ci3 <- quantile(npbres, a))
```

```
##      2.5%     97.5%
## 0.2399808 0.4202492
```

Let us compare the true CI with the estimated CIs:

```
rbind(true = ci0, asy = ci1, pb = ci2, npb = ci3)
```

```
##           2.5%     97.5%
## true 0.2199862 0.4000015
## asy  0.2471472 0.4328366
## pb   0.2500143 0.4299993
## npb  0.2399808 0.4202492
```

5.9 Coverage

We repeat the above experiments multiple times: generate the data, estimate the 95 percent confidence intervals, and check if the interval contains the true value. If the true value is contained between the confidence limits at least 95 percent of the cases, se say the coverage of the estimator is nominal. Here is the code:

```
R <- 100
ci_res <- list()
for (j in 1:R) {
    # cat("run", j, "of", R, "\n") # print run number if you like
    flush.console()
    ## our data
    y <- sim_fun(n, p)
    ## asymptotic CI
    est <- optimize(nll_fun, interval = c(0, 1), y = y)$minimum
```

```
    ci1 <- qnorm(a, mean = est,
        sd = sqrt(est * (1-est) / n))
    ## parametric bootstrap
    pbres <- numeric(B)
    for (i in 1:B) {
        yb <- sim_fun(n, est)
        pbres[i] <- optimize(nll_fun,
            interval = c(0, 1), y = yb)$minimum
    }
    ci2 <- quantile(pbres, a)
    ## non-parametric bootstrap
    npbres <- numeric(B)
    for (i in 1:B) {
        yb <- sample(y, replace = TRUE)
        npbres[i] <- optimize(nll_fun,
            interval = c(0, 1), y = yb)$minimum
    }
    ci3 <- quantile(npbres, a)
    ## store the results
    ci_res[[j]] <- rbind(asy = ci1, pb = ci2, npb = ci3)
}
```

Calculating coverage for the 3 different methods of obtaining the 95% confidence intervals:

```
## a single run
ci_res[[1]]
```

```
##          2.5%     97.5%
## asy 0.2193473 0.4006403
## pb  0.2199862 0.4000015
## npb 0.2199862 0.4000015
```

```
## compare with true p value
ci_res[[1]] - p
```

```
##            2.5%     97.5%
## asy -0.08065273 0.1006403
## pb  -0.08001379 0.1000015
## npb -0.08001379 0.1000015
```

```
## we expect lower CL to be negative
## and upper CL to be positive
sign(ci_res[[1]] - p)
```

```
##     2.5% 97.5%
## asy   -1     1
## pb    -1     1
## npb   -1     1
```

```
## so row sum is 0 if the true value is within CI
coverage <- t(sapply(ci_res, function(z) rowSums(sign(z - p))))
## turn non-0 valies into 1
coverage[coverage != 0] <- 1
## take the complement
coverage <- 1 - coverage
colMeans(coverage)
```

```
##  asy   pb  npb
## 0.95 0.93 0.93
```

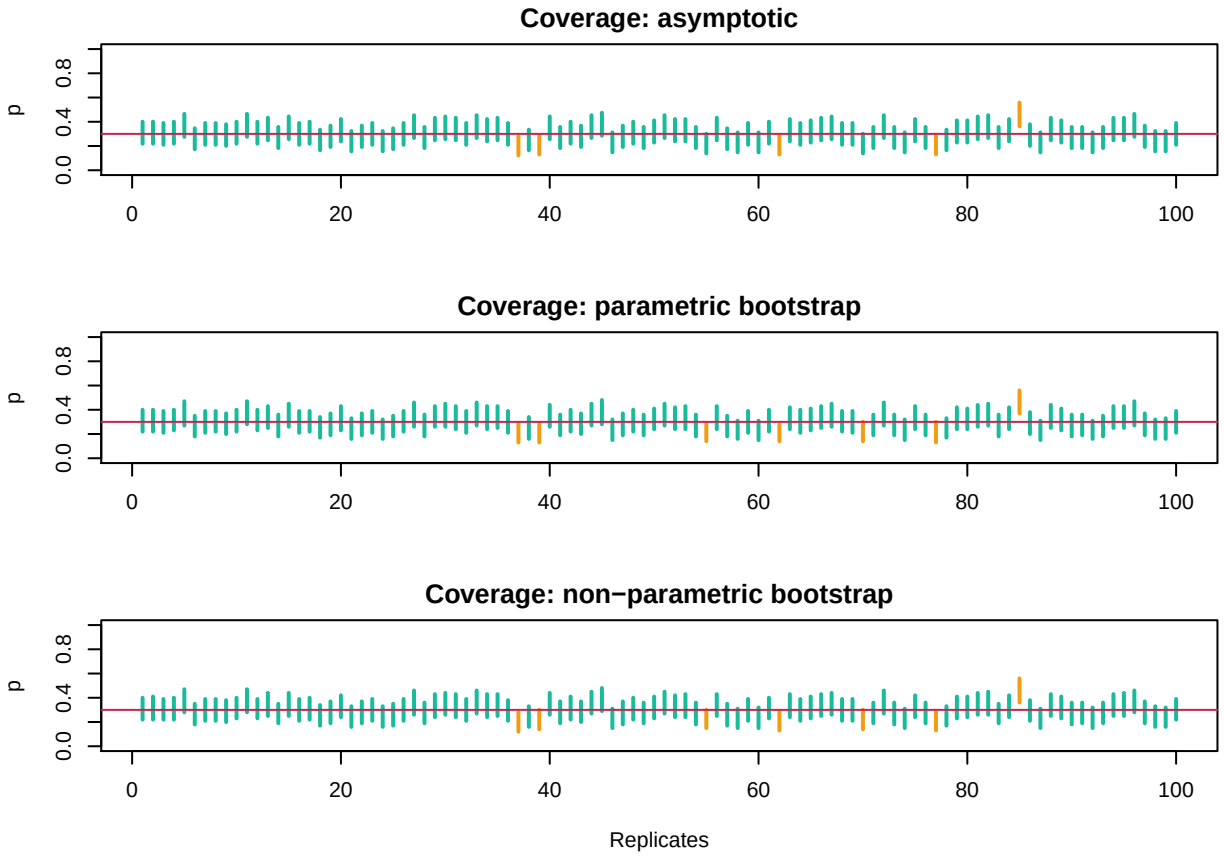
Visualizing the results:

```
op <- par(mfrow = c(3,1), mar = c(4, 4, 2, 1) + 0.1)

plot(0, type = "n", xlim = c(1, R), ylim = c(0, 1),
    main = "Coverage: asymptotic", xlab = "", ylab = "p")
segments(1:R, sapply(ci_res, "[", 1, 1),
    1:R, sapply(ci_res, "[", 1, 2),
    lwd = 2,
    col = ifelse(coverage[,1] == 1, flatly$success, flatly$warning))
abline(h = p, col = flatly$red)

plot(0, type = "n", xlim = c(1, R), ylim = c(0, 1),
    main = "Coverage: parametric bootstrap", xlab = "", ylab = "p")
segments(1:R, sapply(ci_res, "[", 2, 1),
    1:R, sapply(ci_res, "[", 2, 2),
    lwd = 2,
    col = ifelse(coverage[,2] == 1, flatly$success, flatly$warning))
abline(h = p, col = flatly$red)

plot(0, type = "n", xlim = c(1, R), ylim = c(0, 1),
    main = "Coverage: non-parametric bootstrap",
    xlab = "Replicates", ylab = "p")
segments(1:R, sapply(ci_res, "[", 3, 1),
    1:R, sapply(ci_res, "[", 3, 2),
    lwd = 2,
    col = ifelse(coverage[,3] == 1, flatly$success, flatly$warning))
abline(h = p, col = flatly$red)
```




```
par(op)
```

5.10 Summary

This kind of analysis is called the frequentist analysis. We are studying the properties of the inferential statement under the hypothetical replication of the experiment. This analysis tells us about the reliability of the procedure.

The implicit logic is that if the procedure is reliable, we could rely on the inferential statements obtained from only one data set. We choose a procedure that is most reliable.

This is similar to relying more on the blood pressure results from a machine that has small measurement error instead of one with large measurement error.

5.11 Bayesian analysis

One major criticism of the Frequentist approach is that we do not repeat the experiment. What we want to know is: What do the data at hand tell us? Bayesian approach does not quite answer that question but answers a different question: Given these data, how do I change my **beliefs**?

Our goal is to infer about the true parameter value (the true occupancy proportion). Prior distribution, $\pi(\theta)$: This quantifies in probabilistic terms our personal beliefs about the true occupancy rate.

We may believe that it is most likely to be 0.7. Then we consider a distribution with mode at 0.7. We cannot determine the entire distribution from such information. But that is what a Bayesian inference demands. It is a very difficult task but is a necessary task if you want to use the Bayesian approach.

Posterior distribution: This quantifies the beliefs as modified by the data. The mathematical formulation is as follows:

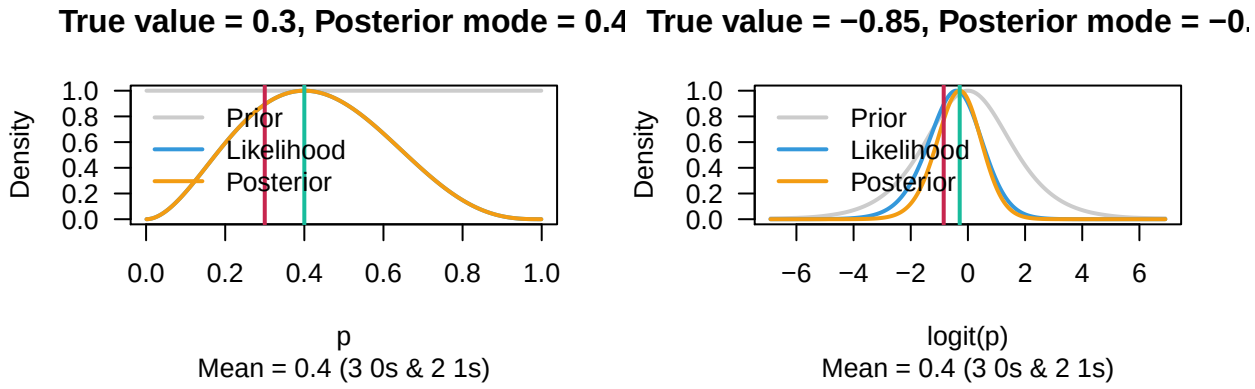
$$\pi(\theta \mid y) = \frac{L(\theta; y)\pi(\theta)}{\int L(\theta; y)\pi(\theta)d\theta}$$

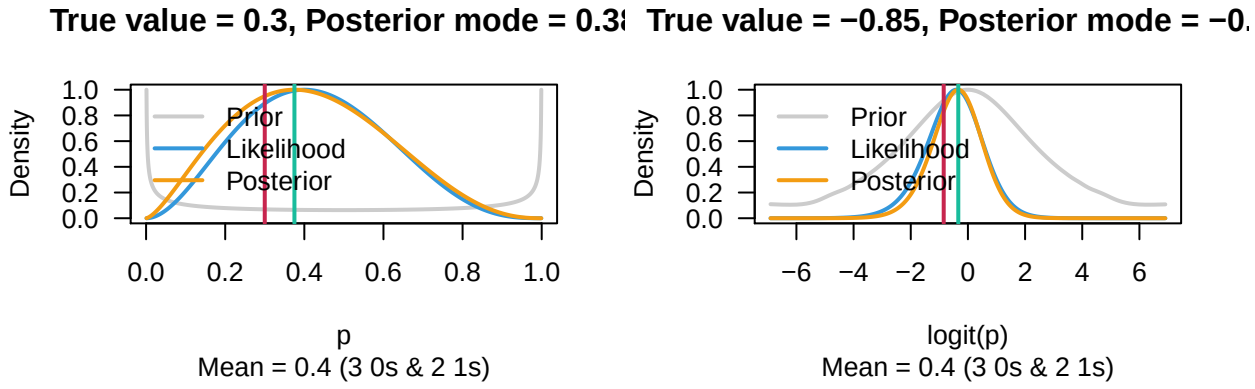
$\pi(\theta)$ is the prior distribution.

5.12 Apps to illustrate Bayesian analysis

Use the dcapps package to play around with different priors for the Bernoulli model. First, try Beta prior:

```
n <- 5
op <- par(mfrow = c(2,2))
beta_prior(p = 0.3, n = n, a = 1, b = 1, scale = "prob")
beta_prior(p = 0.3, n = n, a = 1, b = 1, scale = "logit")
beta_prior(p = 0.3, n = n, a = 0.5, b = 0.5, scale = "prob")
beta_prior(p = 0.3, n = n, a = 0.5, b = 0.5, scale = "logit")
```

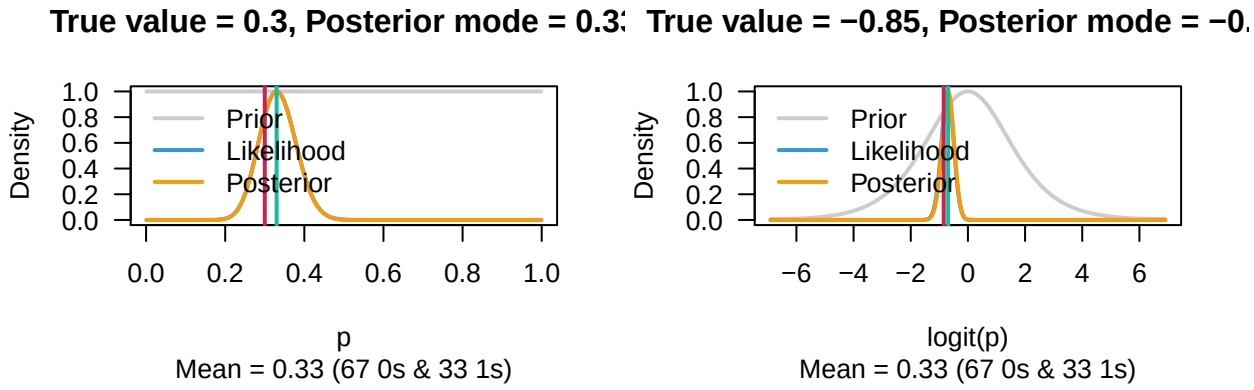


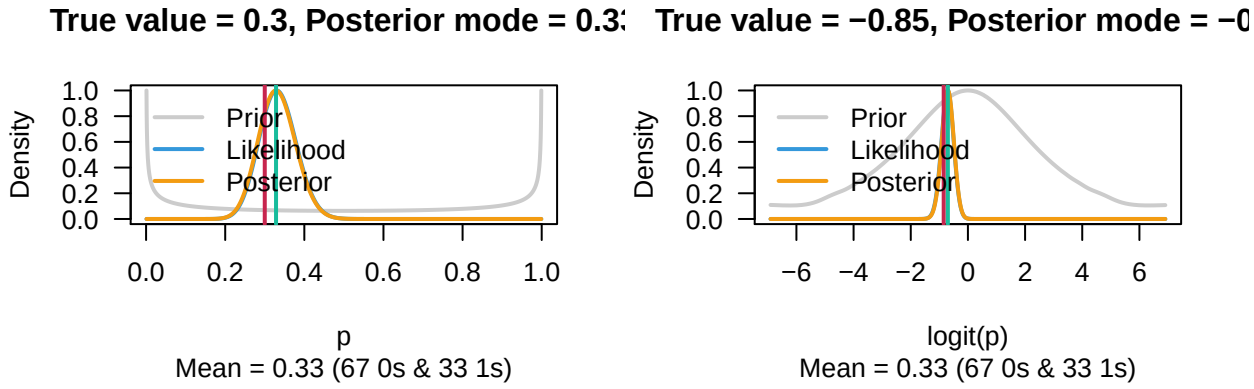



```
par(op)

n <- 100
op <- par(mfrow = c(2,2))
beta_prior(p = 0.3, n = n, a = 1, b = 1, scale = "prob")
beta_prior(p = 0.3, n = n, a = 1, b = 1, scale = "logit")
```

```
beta_prior(p = 0.3, n = n, a = 0.5, b = 0.5, scale = "prob")
beta_prior(p = 0.3, n = n, a = 0.5, b = 0.5, scale = "logit")
```






```
par(op)
```

Beta priors, including Uniform (Beta(1, 1)) and Jeffrey's prior (Beta(0.5, 0.5)) we can see that:

- different priors, same data lead to different posteriors,
- same prior, different data lead to different posteriors,
- as the sample size increases, the posterior is invariant to the prior (eventually degenerate at the true value).

5.13 Non-informative priors (objective Bayesian analysis)

It is clear that priors have an effect on the Bayesian inference. And, they should have an effect. However, this subjectivity is bothersome to many scientists. There is an approach that is euphemistically called an objective Bayesian approach. In this approach, we try to come up with priors that have least influence on the inference (e.g. Bernardo, 1980). There is no unique definition of what we mean by non-informative priors. Various priors have been suggested in the literature. The most commonly used non-informative priors are: Uniform priors and the large variance priors. Other priors are nearly impossible to specify for the complex models that ecologists are faced with.

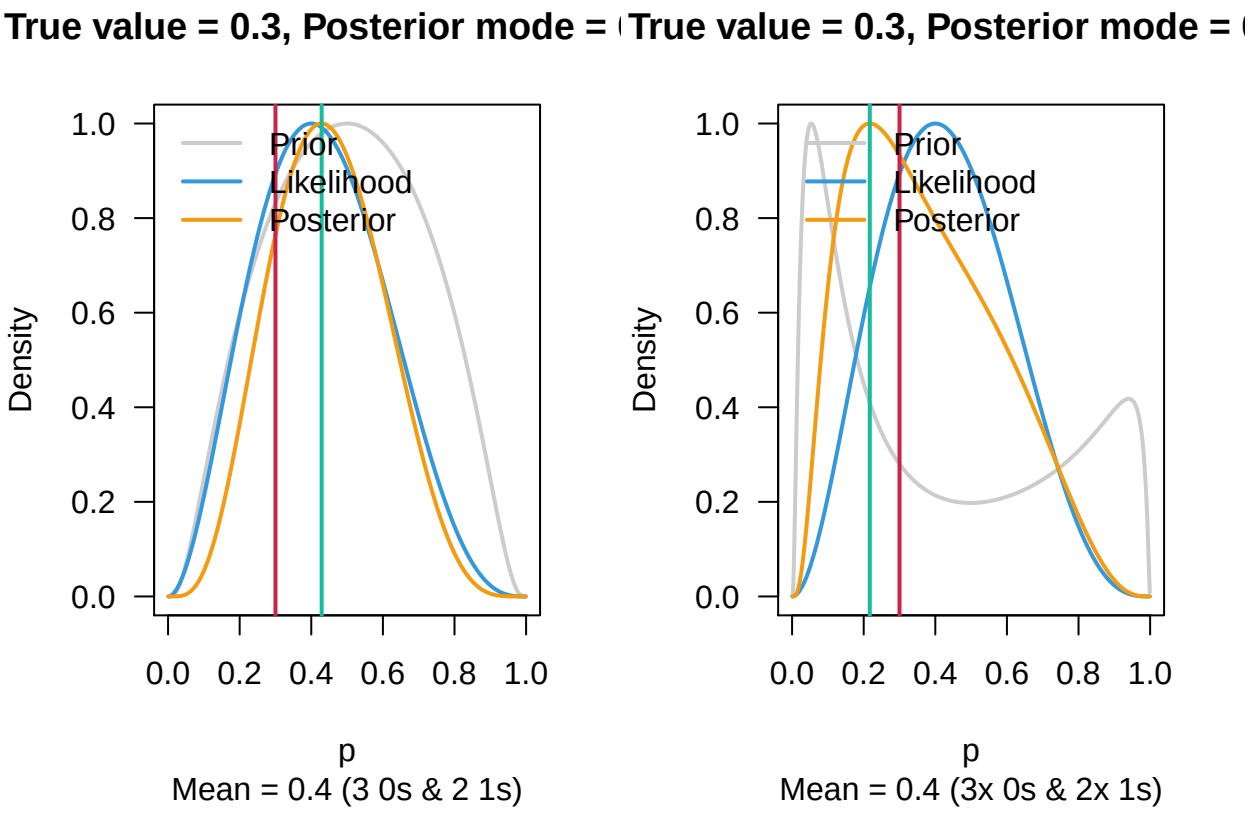
Do these priors affect the inference?

The answer is that "they most certainly do". There is nothing wrong with the priors affecting the inference as long as the researchers can justify their priors.

Use the apps with Beta, Normal, and bimodal priors and check the effects of probability vs. logit scale on the prior and posterior. It is obvious that we get different answers for the same data. Only Jeffrey's prior is not affected.

There is also a function for `normal_prior` and a `bimodel_prior`

```
n <- 5
op <- par(mfrow = c(1,2))
normal_prior(p = 0.3, n = n, mu = 0, var = 1, scale = "prob")
bimodal_prior(p = 0.3, n = n,
    mu1 = -2, var1 = 1, mu2 = 1, var2 = 2, scale = "prob")
```




```
par(op)
```

Fundamentally there is no such thing as 'objective' Bayesian inference. Those who want to use Bayesian inference should simply accept that the inferences are affected by the choice of the priors.

In certain cases (when conjugate prior distributions are used) one can interpret the prior as a set of pseudo-observations. This means that for example in the Bernoulli model and a Beta(a, b) prior distribution, the number of pseudo observations is $a + b - 2$.

5.14 Data cloning: How to trick Bayesians into giving Frequentist answers?

Difference between Frequentist and Bayesian inferential statements: summarize the philosophical differences between the inferential statements made by a Frequentist and a Bayesian in the context of occupancy model.

If the sample size is large, the numerical differences between the Frequentist and Bayesian answers vanish. However, their interpretation is different.

5.15 A brief theory of data cloning

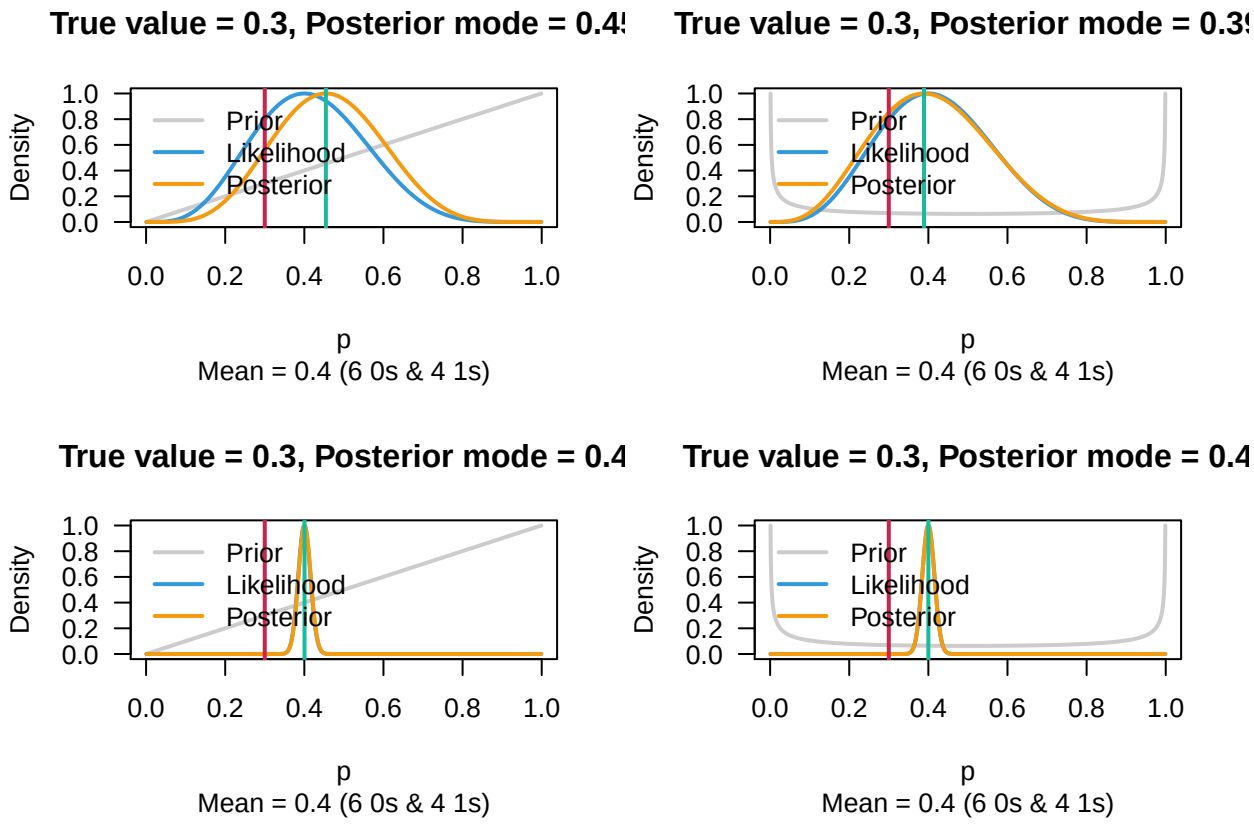
Imagine a hypothetical situation where an experiment is repeated by k different observers, and all k experiments happen to result in exactly the same set of observations, $y^{(k)} = (y, y, \ldots, y)$. The likelihood function based on the combination of the data from these k experiments is $L(\theta, y^{(k)}) = [L(\theta, y)]^k$. The location of the maximum of $L(\theta, y^{(k)})$ exactly equals the location of the maximum of the function $L(\theta, y)$, and the Fisher information matrix based on this likelihood is k times the Fisher information matrix based on $L(\theta, y)$.

One can use MCMC methods to calculate the posterior distribution of the model parameters (θ) conditional on the data. Under regularity conditions, if k is large, the posterior distribution corresponding to k clones of the observations is approximately normal with mean $\hat{\theta}$ and variance $1/k$ times the inverse of the Fisher information matrix. When k is large, the mean of this posterior distribution is the maximum likelihood estimate and k times the posterior variance is the corresponding asymptotic variance of the maximum likelihood estimate if the parameter space is continuous.

Data cloning is a computational algorithm to compute maximum likelihood estimates and the inverse of the Fisher information matrix, and is related to simulated annealing. By using data cloning, the statistical accuracy of the estimator remains a function of the sample size and not of the number of cloned copies. Data cloning does not improve the statistical accuracy of the estimator by artificially increasing the sample size. The data cloning procedure avoids the analytical or numerical evaluation of high dimensional integrals, numerical optimization of the likelihood function, and numerical computation of the curvature of the likelihood function.

Use the following app to see what happens when we clone the data instead of increasing sample size:

```
op <- par(mfrow = c(2, 2))
data_cloning(p = 0.3, n = 10, a = 2, b = 1, K = 1, seed = 0)
data_cloning(p = 0.3, n = 10, a = 0.5, b = 0.5, K = 1, seed = 0)
data_cloning(p = 0.3, n = 10, a = 2, b = 1, K = 100, seed = 0)
data_cloning(p = 0.3, n = 10, a = 0.5, b = 0.5, K = 100, seed = 0)
```




```
par(op)
```

5.16 MCMC

MCMC = Markov chain Monte Carlo

5.16.1 Conventional maximum likelihood estimation

```
m <- glm(formula = y ~ 1, family=binomial("logit"))
summary(m)
```

```
##
## Call:
## glm(formula = y ~ 1, family = binomial("logit"))
##
## Deviance Residuals:
##     Min       1Q   Median       3Q      Max
## -0.8446  -0.8446  -0.8446   1.5518   1.5518
##
## Coefficients:
##             Estimate Std. Error z value Pr(>|z|)
## (Intercept)  -0.8473     0.2182  -3.883 0.000103 ***
## ---
```

```
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## (Dispersion parameter for binomial family taken to be 1)
##
##     Null deviance: 122.17  on 99  degrees of freedom
## Residual deviance: 122.17  on 99  degrees of freedom
## AIC: 124.17
##
## Number of Fisher Scoring iterations: 4
```

```
coef(m)
```

```
## (Intercept)
##  -0.8472979
```

```
exp(coef(m)) / (1 + exp(coef(m)))
```

```
## (Intercept)
##         0.3
```

```
plogis(coef(m))
```

```
## (Intercept)
##         0.3
```

```
mean(y)
```

```
## [1] 0.3
```

```
confint(m)
```

```
## Waiting for profiling to be done...
```

```
##      2.5 %     97.5 %
## -1.2889816 -0.4301396
```

5.16.2 Bayesian model in JAGS

```
library(dclone)
library(rjags)
```

```
## Linked to JAGS 4.0.1
```

```
## Loaded modules: basemod,bugs
```

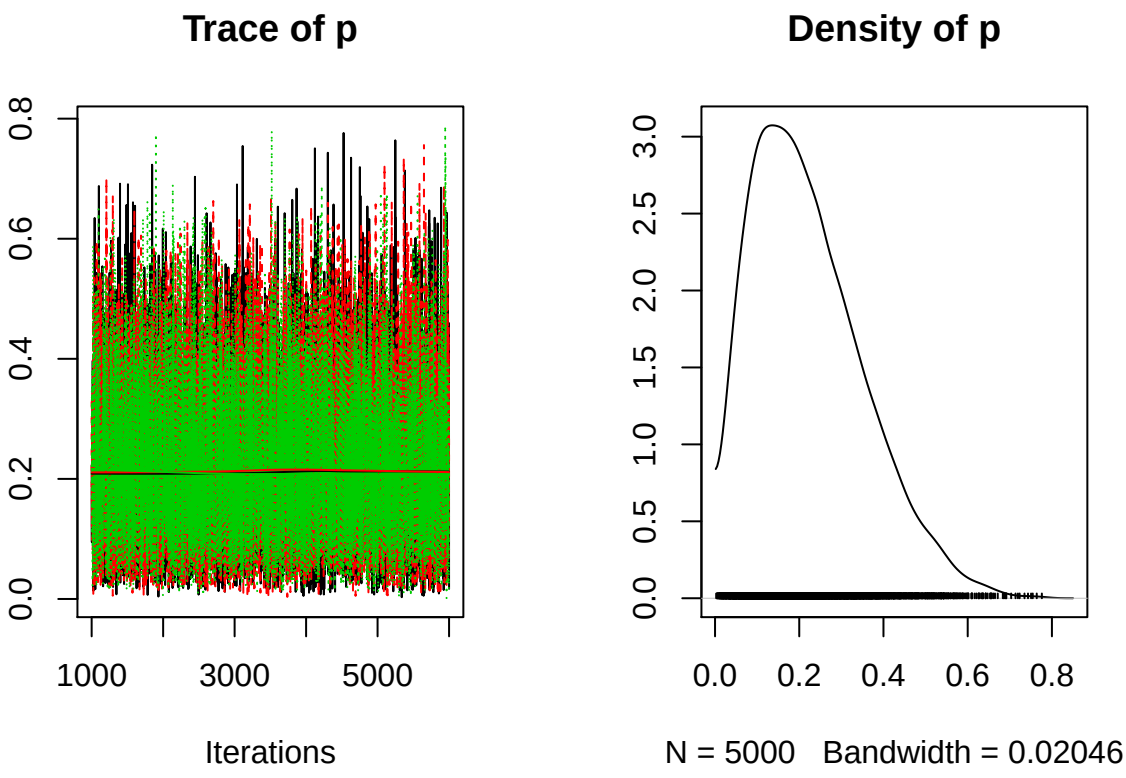
```
model <- custommodel("model {
    for (i in 1:n) {
        #Y[i] ~ dbin(p, 1) # Binomial(N,p)
        Y[i] ~ dbern(p) # Bernoulli(p)
    }
    #p ~ dunif(0.001, 0.999)
    p ~ dbeta(1, 3)
}")
dat <- list(Y = y, n = n)
fit <- jags.fit(data = dat, params = "p", model = model)
```

```
## Compiling model graph
##    Resolving undeclared variables
##    Allocating nodes
## Graph information:
##    Observed stochastic nodes: 5
##    Unobserved stochastic nodes: 1
##    Total graph size: 104
##
## Initializing model
```

```
summary(fit)
```

```
##
## Iterations = 1001:6000
## Thinning interval = 1
## Number of chains = 3
## Sample size per chain = 5000
##
## 1. Empirical mean and standard deviation for each variable,
##    plus standard error of the mean:
##
##           Mean             SD       Naive SE Time-series SE
##       0.222588       0.132062       0.001078       0.001078
##
## 2. Quantiles for each variable:
##
##    2.5%     25%     50%     75%   97.5%
## 0.03064 0.11975 0.20196 0.30408 0.52617
```

```
plot(fit)
```



5.16.3 Data cloning

To make sure that both locations and clones are independent (i.i.d.), it is safest to include and extra dimension and the corresponding loop.

```
model <- custommodel("model {
    for (k in 1:K) {
        for (i in 1:n) {
            Y[i,k] ~ dbin(p, 1)
        }
    }
    logit_p <- log(p/(1-p))
    p ~ dbeta(0.001, 0.999)
}")
dat <- list(Y = dcdim(data.matrix(y)), n = n, K = 1)
dcfit <- dc.fit(data = dat, params = "logit_p", model = model,
    n.clones = c(1,100), unchanged = "n", multiply = "K")
```

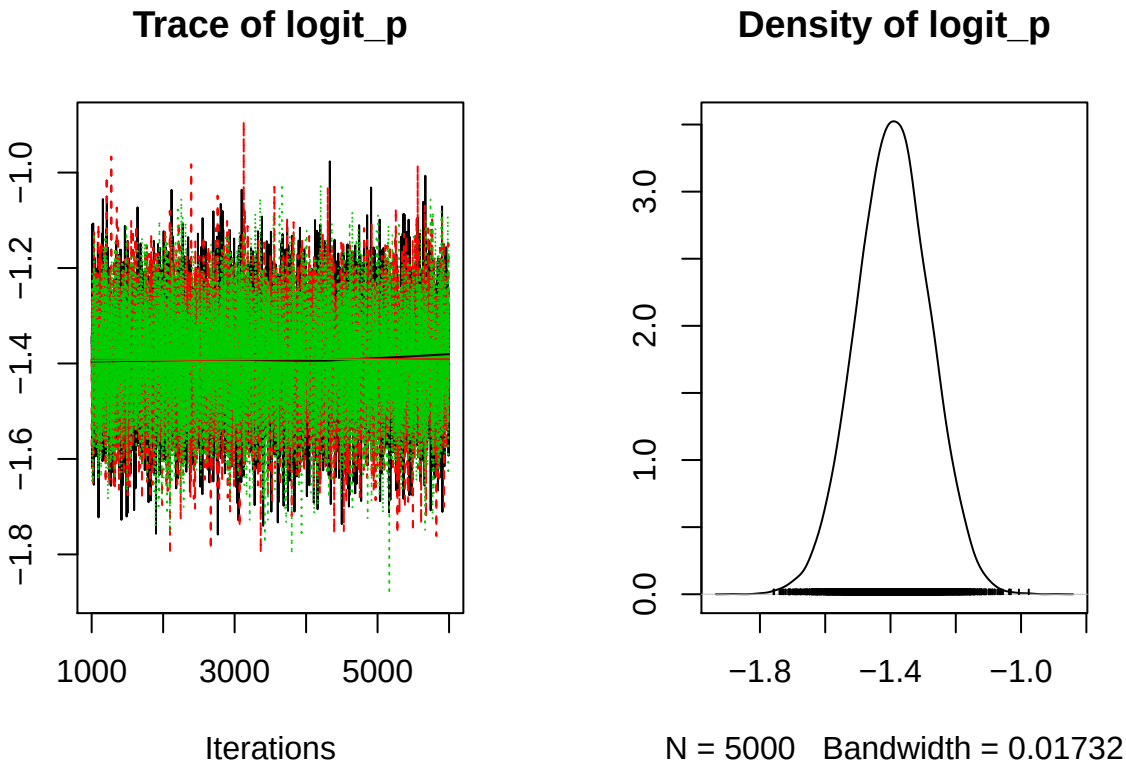
```
##
## Fitting model with 1 clone
##
## Compiling model graph
##    Resolving undeclared variables
##    Allocating nodes
## Graph information:
##    Observed stochastic nodes: 5
##    Unobserved stochastic nodes: 1
```

```
##    Total graph size: 114
##
## Initializing model
##
##
## Fitting model with 100 clones
##
## Compiling model graph
##    Resolving undeclared variables
##    Allocating nodes
## Graph information:
##    Observed stochastic nodes: 500
##    Unobserved stochastic nodes: 1
##    Total graph size: 10509
##
## Initializing model
```

```
summary(dcfit)
```

```
##
## Iterations = 1001:6000
## Thinning interval = 1
## Number of chains = 3
## Sample size per chain = 5000
## Number of clones = 100
##
## 1. Empirical mean and standard deviation for each variable,
##    plus standard error of the mean:
##
##           Mean     SD DC SD.logit_p Naive SE Time-series SE  R hat
## logit_p -1.392 0.1118         1.118 0.000913      0.0009043 0.9999
##
## 2. Quantiles for each variable:
##
##   2.5%    25%    50%    75%  97.5%
## -1.612 -1.467 -1.392 -1.317 -1.175
```

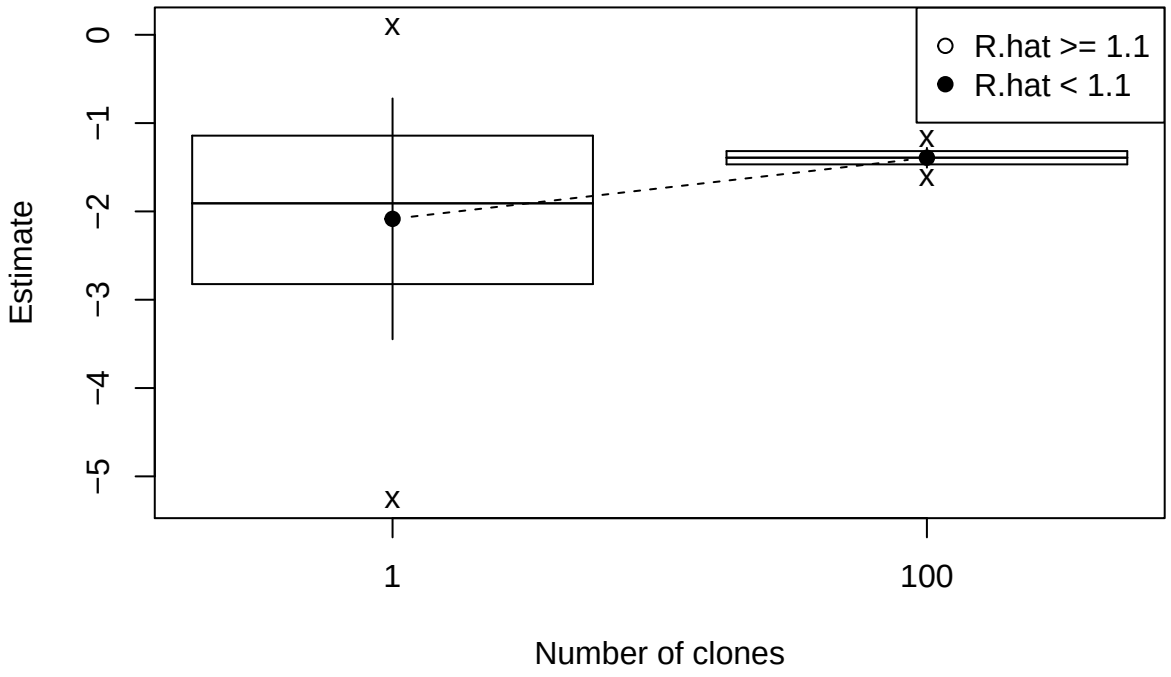
```
plot(dcfit)
```




```
dctable(dcfit)
```

```
## $logit_p
##   n.clones      mean        sd      2.5%       25%       50%       75%
## 1        1 -2.084406 1.3631468 -5.259007 -2.822891 -1.908434 -1.141974
## 2      100 -1.392169 0.1118232 -1.612039 -1.467251 -1.391643 -1.316965
##         97.5%     r.hat
## 1  0.09549493 1.0002098
## 2 -1.17521822 0.9998778
##
## attr(,"class")
## [1] "dctable"
```

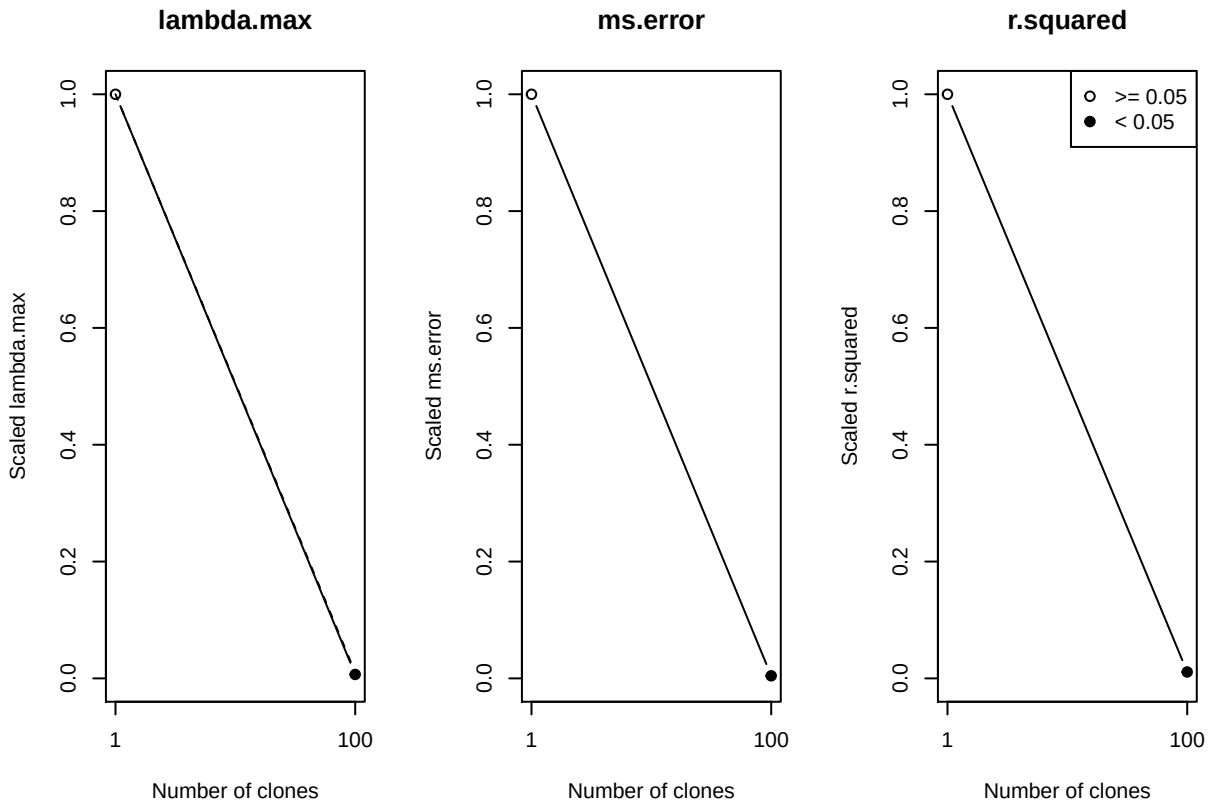
```
plot(dctable(dcfit))
```




```
dcdiag(dcfit)
```

```
##   n.clones lambda.max   ms.error   r.squared     r.hat
## 1        1 1.85816913 0.57716616 0.110979389 1.0002098
## 2      100 0.01250443 0.00253001 0.001213698 0.9998778
```

```
plot(dcdiag(dcfit))
```



5.16.3.1 Modification If locations are treated as i.i.d., it is possible to replicate the vector, so that length becomes `n * K`.

```
model <- custommodel("model {
    for (i in 1:n) {
        Y[i] ~ dbin(p, 1)
    }
    p ~ dunif(0.001, 0.999)
}")
dat <- list(Y = y, n = n)
dcfit <- dc.fit(data = dat, params = "p", model = model,
    n.clones = c(1,2,4,8), multiply = "n")
```

```
##
## Fitting model with 1 clone
##
## Compiling model graph
##    Resolving undeclared variables
##    Allocating nodes
## Graph information:
##    Observed stochastic nodes: 5
##    Unobserved stochastic nodes: 1
##    Total graph size: 109
##
## Initializing model
##
```

```
##
## Fitting model with 2 clones
##
## Compiling model graph
##    Resolving undeclared variables
##    Allocating nodes
## Graph information:
##    Observed stochastic nodes: 10
##    Unobserved stochastic nodes: 1
##    Total graph size: 214
##
## Initializing model
##
##
## Fitting model with 4 clones
##
## Compiling model graph
##    Resolving undeclared variables
##    Allocating nodes
## Graph information:
##    Observed stochastic nodes: 20
##    Unobserved stochastic nodes: 1
##    Total graph size: 424
##
## Initializing model
##
##
## Fitting model with 8 clones
##
## Compiling model graph
##    Resolving undeclared variables
##    Allocating nodes
## Graph information:
##    Observed stochastic nodes: 40
##    Unobserved stochastic nodes: 1
##    Total graph size: 844
##
## Initializing model
```

More in person. Safe travels and see you in Madison!