

Lab 6 & 7 Fitting models to data, optimisation and bayesian statistics

Ben Bolker, modified at several places by Bob Douma and Alejandro Morales. Bayesian part by AM

16 November 2022

Contents

1	Learning goals	1
2	Fitting models to data	2
3	Fitting parameters of made-up data	3
3.1	Finding the maximum likelihood estimate of the paramaters	4
4	Maximum likelihood and continuous covariates	6
5	Maximum likelihood with continous and categorical predictors	7
6	Advanced topics	8
6.1	Likelihood surface	8
6.2	Optimisation problems and assessing the confidence limits of parameter estimates	9
6.3	Bayesian parameter estimation: negative binomial	14
7	Hints for choosing deterministic functions and stochastic functions	22

1 Learning goals

You will learn how to:

1. Program the likelihood function of a model.
2. Estimate the parameters of a model through maximum likelihood, including models with continuous and categorical covariates.
3. Estimate the confidence intervals of the model parameters through profiling and the quadratic approximation.
4. Estimate parameters in a Bayesian framework and how parameter uncertainty can be assessed

In case of time constraints, focus on sections 2-5. If you want something challenging do 6,7 and 8 as well.

2 Fitting models to data

Fitting a model to data through likelihood requires that you take five steps:

1. Specify how the dependent variable depends on the independent variable, i.e. specify a function that describes how the mean of y depends on the value of x .
2. Specify a probability distribution to describe the deviations of the observations from the mean
3. Specify a function that calculate the negative log likelihood (NLL) based on the data and the parameter values.
4. Choose the parameters of the deterministic model and the probability model such that the negative log likelihood is lowest.
5. Compare the likelihood of alternative models (change the deterministic function or the stochastic function) and compare with AIC(c) or BIC which model is most parsimonious.

For example to calculate the NLL of a linear model and a normal distribution the following function works:

```
nll = function(par,y,x){  
  a = par[1]  
  b = par[2]  
  sd = par[3]  
  # this calculates the mean y for a given value of x:  
  #the deterministic function  
  mu = a+b*x  
  # this calculates the likelihood of the function given the probability  
  # distribution, the data and mu and sd  
  nll = -sum(dnorm(y,mean=mu,sd=sd,log=T))  
  return(nll)  
}
```

Notice that the function takes three arguments: a vector with parameters, a vector with y values and a vector with x values. Inside the vector `par`, three values are stored: `a`, `b` and `sd`. Next, the mean given x is calculated with `mu=a+b*x`. The `nll` returns the Negative LogLikelihood of the data (y) given a normal distribution with mean `mu` (vector!) and a standard deviation `sd`. The `log=T` returns the log of the probability densities.

Next we call an optimisation function to find the maximum likelihood estimate

```
par=c(a=1,b=1,c=1) # initial parameters  
  
# y represents the data, x the independent variable  
  
opt1 = optim(par=nll,x=x,y=y,hessian=TRUE)
```

The optimization result is a list with elements:

- the best-fit parameters (`opt1$par`, with parameter names because we named the elements of the starting vector—see how useful this is?);}
- the minimum negative log-likelihood (`opt1$value`);
- information on the number of function evaluations (`opt1$counts`; the `gradient` part is NA because we didn't specify a function to calculate the derivatives (and the Nelder-Mead algorithm wouldn't have used them anyway)
- information on whether the algorithm thinks it found a good answer `opt1$convergence`, which is zero if R thinks everything worked and uses various numeric codes (see `?optim` for details) if something goes wrong;
- `opt1$message` which may give further information about the when the fit converged or how it failed to converge;
- because we set `hessian=TRUE`, we also get `opt1$hessian`, which gives the (finite difference approximation of) the second derivatives evaluated at the MLE.

It can also be done through `mle2`

```
nll.mle = function(a,b,sd){  
  # this calculates the mean y for a given value of x: the deterministic function  
  mu = a+b*x  
  # this calculates the likelihood of the function given the probability  
  # distribution, the data and mu and sd  
  nll = -sum(dnorm(y,mean=mu,sd=sd,log=T))  
  return(nll)  
}  
  
# the data should be supplied through data and the parameters through list().  
mle2.1 = mle2(nll.mle,start=list(a=1,b=1,sd=1),data=data.frame(x,y))  
summary(mle2.1)
```

3 Fitting parameters of made-up data

The simplest thing to do to convince yourself that your attempts to estimate parameters are working is to simulate the “data” yourself and see if you get close to the right answers back. Set the random seed to 1001 so we get identical answers across R sessions.

3.1 Finding the maximum likelihood estimate of the parameters

Lab Exercise 3.1

Take the steps below

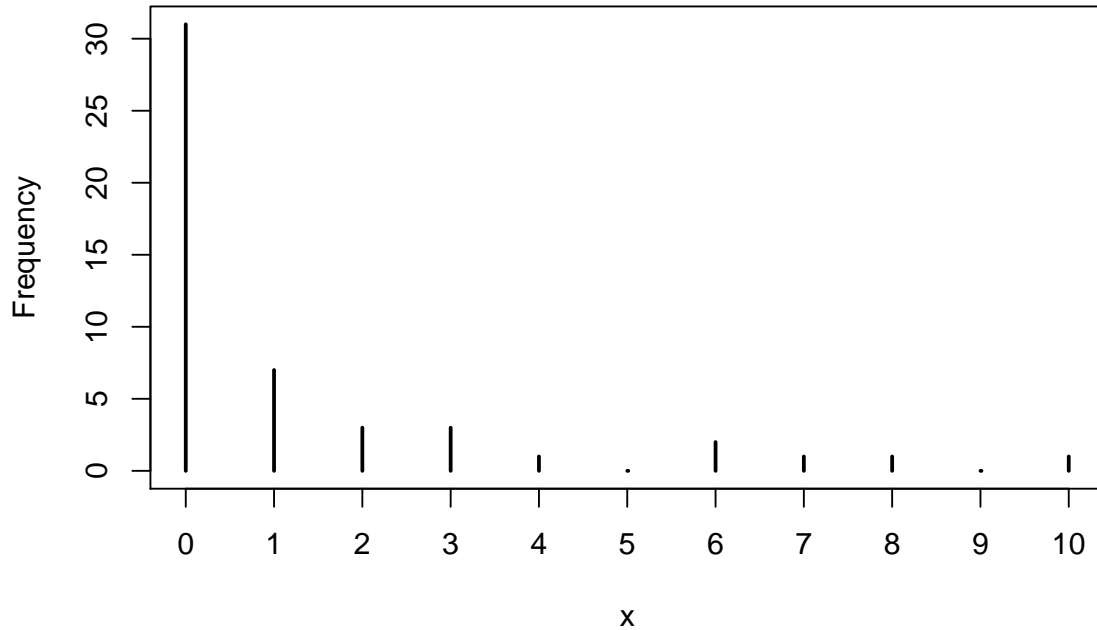
1. Generate 50 values from a negative binomial (`rnbinom`) with $\mu = 1$, $k = 0.4$. Save the values in variables in case we want to use them again later.
2. Plot the numbers in a frequency diagram
3. Next, define the negative log-likelihood function for a simple draw from a negative binomial distribution: the first parameter, `par`, will be the vector of parameters, and the second parameter, `dat`, will be the vector with simulated values.
4. Calculate the negative log-likelihood of the data for the parameter values with which you generated the numbers. Combine these parameter values into the vector `par` with `c()` to pass them to the negative log-likelihood function. Naming the elements in the parameter vector is optional but can help avoid mistakes if the number of parameters is large (e.g. `par = c(mu = 1, k = 2)`).
5. Calculate the NLL of parameter values that are far from the values that were used to generate the data ($\mu = 10$, $k = 10$)
6. Calculate the maximum likelihood estimate (MLE)? Use `optim` with the default options (Nelder-Mead simplex method) and the method-of-moments estimates as the starting estimates (`par`): `opt1 = optim(fn=NLLfun1, par=c(mu=mu.mom, k=k.mom), hessian=TRUE)`
7. What is the difference in NLL between the MLE estimates and the NLL derived at 5?

The Likelihood Ratio Test would say, however, that the difference in likelihoods would have to be greater than $\chi^2_2(0.95)/2$ (two degrees of freedom because we are allowing both μ and k to change). This can be done through `ldiff=nll.true-nll.mom` and `qchisq(0.95,df=2)/2`. So — better, but not significantly better at $p = 0.05$. `pchisq(2*ldiff,df=2,lower.tail=FALSE)` would tell us the exact p -value if we wanted to know.)

```
set.seed(1001)
mu.true=1
k.true=0.4

x = rnbinom(50,mu=mu.true,size=k.true)

plot(table(factor(x,levels=0:max(x))),
     ylab="Frequency",xlab="x")
```



```

# this function calculate the NLL of the data given the set of parameters defined in p
NLLfun1 = function(p,dat=x) {
mu=p[1]
k=p[2]
-sum(dnbinom(x,mu=m, size=k, log=TRUE))
}

# the NLL of the data given the parameter values that were used to generate the data
nll.true=NLLfun1(c(mu=mu.true,k=k.true))

nll.true

## [1] 72.64764
NLLfun1(c(mu=10,k=10))

## [1] 291.4351

m = mean(x)
v = var(x)
# calculate parameters through method of moments
mu.mom = m
k.mom = m/(v/m-1)

```

```

# find MLE estimate of the parameters given the data
opt1 = optim(fn=NLLfun1,par=c(mu=mu.mom,k=k.mom),hessian=TRUE)
coef(opt1)

## NULL

# NLL at MLE
opt1$value

## [1] 71.79646

# compare with nll.true

# significantly different?
ldiff=nll.true-opt1$value; ldiff

## [1] 0.8511813

# no significant difference (which is what we would expect in 95% the generated datasets)
pchisq(2*ldiff,df=2,lower.tail=FALSE)

## [1] 0.4269103

```

The minimum negative log-likelihood (`round(opt1$value,2)`) is better than the NLL of the model with the true parameters (`round(nll.true,2)`), but all of these are within the LRT cutoff, i.e. the negative log likelihoods differ by less than 1.92. Remember that the cut-off is based on the Likelihood Ratio Test that states that twice the difference in the log-likelihood between the simpler and more complex model will follow a χ^2 distribution with n degrees of freedom. n is the number of parameters that are fixed to a specific value. The cut-off value for a χ^2 with 1 degree of freedom is 3.84. The value of 1.92 is derived from $3.84/2$ because we evaluate the difference in log Likelihood and not twice the difference. In other words, we could also multiply all the logLikelihood surface by two and find the 3.84 cutoff.

4 Maximum likelihood and continuous covariates

The following exercise has the purpose to learn you how to fit a model to data when we have a single covariate.

Lab Exercise 4.1

1. Take the second dataset (shapes2.csv from shapes.xlsx), use a michaelis-menten as deterministic function, and a normal distribution as stochastic model. Tweak the function in the first three grey boxes (above) such that it accomodates the michaelise menten and the normal distribution.
hint: In a previous exercise you have eyeballed the parameter values of the functions, you can use these as starting values.
hint: In case you get convergence problems, further adapt your starting values, or choose a different optimizer. For example Nelder-Mead is a robust one, e.g. `method = "Nelder-Mead"`.
2. Change the determinstic function for a possible alternative determinstic function, and fit this new model to the data. Remember that in Lab 3 you have proposed multiple deterministic functions for this dataset.
3. Compare the likelihoods of the data given both models
4. Apply model selection criteria and conclude which model fits that data best.
5. Does the model makes sense from a biological perspective?

5 Maximum likelihood with continous and categorical predictors

Sometimes you want to fit the same model to different groups (males/females, treatment/control etc.). The easiest way is to separately fit the model to the subsets, but this makes it very difficult to assess whether the fitted parameters for both groups are comparable. A more elegant method is explained below.

We use the fifth dataset of the six datasets you have worked with earlier on (shapes5.csv or the fifth sheet from shapes.xlsx). Assume that the function was generated by a decreasing exponential function $ae^{(-bx)}$ and you want to the values of a and b . The dataset has three columns that are relevant: the independent variable x , the dependent variable y , and a dummy variable $group$ indicating to which group the observation belongs to. We want to test whether we can justify a different a and b for the two groups.

This is how the NLL function would look like assuming no grouping:

```
read.csv("shapes5.csv") # and select fifth dataset
# test dataset five for differences between groups
nll0 = function(par,dat){
  a = par[1]
  b = par[2]
  ymean = a*exp(-b*dat$x)
  nll = -sum(dpois(dat$y,lambda=ymean,log=T))
  return(nll)
}

par=c(4,0.2)
```

```
opt1 = optim(par=par,fn=nll0,dat=dat)
```

Lab Exercise 5.1

1. Fit the above model to the data without considering differences between groups in a and b .
2. Adjust the likelihood function such that it can accomodate for different values of b depending on the group an observation belong to.

Use the following pseudocode to achieve this and/or check page 305 for in inspiration or go back to Lab 1 section 11.1.2. a. Adapt the likelihood function such that the parameter b depends on the group. b. Adjust the starting values so it contains multiple starting values for b

3. Estimate the parameters a and b when letting b depend on the group. Compare the negative loglikelihood of this model with the model fitted in question 1. Which has a better fit?
4. Apply model selection techniques (Likelihood ratio test, AIC or BIC) to select the most parsimonious model. Are the models nested? Which model is preferred?

Lab Exercise 5.2

To practice model fitting a little bit more, you could repeat the above procedure for the other 4 datasets from shapes.xlsx.

Pick a dataset, go back to the Lab 3 Question 2.1 and Lab 4 Question 2.1 and list the stochastic model and the deterministic function and the eyeballed parameters that you thought were appropriate for this dataset. Next write a negative loglikelihood function, and use mle2 or optim to obtain the maximum likelihood estimates for the parameters. If you have practised sufficiently, you can move on with the advanced topics below.

6 Advanced topics

6.1 Likelihood surface

To find the likelihood surface follow the steps below (background information can be found in Bolker Ch. 6). This exercise continues on exercise #3.1 (Lab 3) where you used the negative binomial to generate 50 numbers and fitted back the parameters.

Lab Exercise 6.1

For the likelihood surface:

1. Set up vectors of μ and k values. Let's try μ from 0.4 to 3 in steps of 0.05 and k from 0.01 to 0.7 in steps of 0.01.
2. Set up a matrix to hold the results, The matrix for the results will have rows corresponding to μ and columns corresponding to k :
3. Run **for** loops to calculate and store the values. Use a **for** nested in another one
4. Drawing a contour using the function 'contour'. Change the argument **nlevels** to 100 to get a better view of the likelihood surface
5. Add the MLE estimates in the contour plot (use 'points'). Additionally, add the parameter values that were used to generate the data, and the parameter values that were obtained with the method of moments.

6.2 Optimisation problems and assessing the confidence limits of parameter estimates

Fitting a model to data requires you to specify a relationship between variables. After specifying this relationship we need to fit parameters of this model that best fits the data. This fitting is done through computer algorithms (optimizers). However, sometimes it may be hard to fit a model to data. After having found the best fitting model, you want to assess how certain you are about the parameter estimates. For assessing the uncertainty of model parameters several methods exist that have pros and cons.

If you feel comfortable with fitting models to data you are ready for a more challenging exercise. If you do not feel comfortable yet, go back to question 5.2 and practise a bit more.

This exercise has two purposes. First you will learn that an innocent looking function can be challenging to fit. Second, you will learn to assess the uncertainty in the parameter values. For assessing the uncertainty in the parameter estimates there are two methods: the profiling method and the quadratic approximation. Bolker recommends to use the likelihood profile for assessing the uncertainty in the parameters because this one is more accurate than the approximation based on the Hessian matrix.

1. Take the first dataset of the six datasets you have worked with earlier on. Assume that the function was generated by the monomolecular function $a(1 - e^{-bx})$. Fit this model with normally distributed errors through this data with **mle2** and **optim** method **Nelder-Mead**. Choose four different starting points of the optimisation: **start_a** = **c(5,10,20,30)**, **start_b** = **c(0.001,0.005,0.01,0.1)** and compare the NLL of those four optimisations. Plot the curves into the plot with data and try to understand what happened. You can set the σ to 3.
2. To understand the behaviour of the optimisation routine we will plot the likelihood surface over a range of values of a and b . For a choose a number of parameter values in the range of 0-40 and for b choose a number of values in the range 0.1-10. Calculate for each combination the NLL and plot the NLL surface using **contour** plot. For more insight into the functioning of what the optimisation method did, you can add the starting points that you gave to **mle2** and the best fitting points, use **points()** for

this. Do you have a clue why the optimisation did not find the minimum point in the landscape? Now zoom in and choose values for b in the range of 0.001-0.03 and check again the NLL surface.

hint: See Bolker Lab 6 for inspiration on coding.

hint: You can use a for a double for-loop to run over all parameters

hint: Store the NLL results in a matrix (you can make a 100x100 matrix by `matrix(NA,nrow=100,ncol=100)`).

3. Calculate the confidence intervals of the parameters through constructing the likelihood profile. Consult page 106 of Bokler or Lab 6 for how to calculate the confidence intervals based on the likelihood profile. Use the following pseudocode to achieve this:
 - a. Adapt the likelihood function such that one parameter is not optimised but chosen by you, say parameter a .
 - b. Vary a of a range and optimise the other parameters.
 - c. Plot the NLL as a function of parameter a .
 - d. Find the values of a that enclose $-L + \chi^2(1 - \alpha)/2$. In R this can be done through `qchisq(0.95,1)/2`.
 - e. Compare your results with the results from the R function `confint()`. `confint()` uses the profiling method along with interpolation methods.
4. (*time permitting*) Calculate the confidence intervals through the quadratic approximation. Take the following steps to achieve this:
 - a. Get the standard error of the parameter estimates through `vcov`. Note that `vcov` return the variance/covariance matrix
 - b. Calculate the interval based on the fact that the 95% limits are 1.96 (`qnorm(0.975,0,1)`) standard deviation units away from the mean.
5. (*time permitting*) Plot the confidence limits of the both method and compare the results. Is there a big difference between the methods?
6. To assess the uncertainty in the predictions from the model you can construct population prediction intervals (PPIs, see 7.5.3 Bolker). Population prediction intervals shows the interval in which a new observation will likely fall. To construct the PPI take the following steps
 - a. Simulate a number of parameter values taken the uncertainty in the parameter estimates into account.

hint: If the fitted mle object is called `mle2.obj`, then you can extract the variance-covariance matrix by using `vcov(mle2.obj)`. You can extract the mean parameter estimates by using `coef(mle2.obj)`. Now you are ready to simulate 1000 combinations of parameter values through `z = mvrnorm(1000,mu=coef(mle2.obj),Sigma=vcov(mle2.obj))`. `mvrnorm` is a function to randomly draw values from a multivariate normal distribution.
 - b. Predict the mean response based on the simulated parameter values and the values of x

hint: make a for-loop and predict for each simulated pair of parameter values the mean for a given x . Thus $\mu = z[i,1]*(1-\exp(-z[i,2]*x))$

- c. Draw from a normal distribution with a mean that was predicted in the previous step and the sd that you simulated in step a.

hint: $\text{pred} = \text{rnorm}(\text{length}(\mu), \text{mean}=\mu, \text{sd}=z[i,3])$. Store pred in a matrix with each simulated dataset in a separate row.

- d. Calculate for each value of x the 2.5% and the 97.5% quantiles

hint: If the predictions are stored in a matrix mat , you can use $\text{apply}(\text{mat}, 2, \text{quantile}, 0.975)$ to get the upper limit.

```
shapes1= read.csv("shapes1.csv")
plot(shapes1)

nll.mle = function(a,b,sd){
  # this calculates the mean y for a given value of x: the deterministic function
  mu = a*(1-exp(-b*shapes1$x))
  # this calculates the likelihood of the function given the probability
  # distribution, the data and mu and sd
  nll = -sum(dnorm(shapes1$y,mean=mu,sd=sd,log=T))
  return(nll)
}

library(bbmle)

# Try 4 different starting points
mle2.1 = vector("list", 4)
start_a = c(5,10,20,30)
start_b = c(0.001,0.005,0.01,0.1)
for(i in 1:4) {
  mle2.1[[i]] = mle2(nll.mle,start=list(a=start_a[i],b = start_b[i], sd=1), method="Nelder-M")
}

# Check the best fit (in this case it is 3rd starting point)
for(i in 1:4) {
  print(logLik(mle2.1[[i]]))
}

# Extract the best fit for the rest of the analysis
best_mle2.1 = mle2.1[[3]]
summary(best_mle2.1)
logLik(best_mle2.1)
confint(best_mle2.1)
coef(best_mle2.1)
```

```

plot(shapes1)
curve(coef(best_mle2.1)[1]*(1-exp(-coef(best_mle2.1)[2]*x)),add=T)
curve(coef(mle2.1[[1]])[1]*(1-exp(-coef(mle2.1[[1]])[2]*x)),add=T, col = 2)

# likelihood surface
a1 = seq(0,40,length.out = 100)
b1.1 = seq(0.001,0.03,length.out=100)
b1.2 = seq(0.1,10,length.out=100)

nll.grid = expand.grid(a1,b1.1)
nll.grid$NLL = NA
no = 0
# Construct first contour
for (i in 1:length(a1)){
  for (j in 1:length(b1.1)){
    no = no + 1
    nll.grid[no,1] = a1[i]
    nll.grid[no,2] = b1.1[j]
    nll.grid[no,3] = nll.mle(a=a1[i],b=b1.1[j],sd=2.06)
  }
}
library(reshape2)
z1.1 = as.matrix(dcast(nll.grid,Var1~Var2)[-1])

# Construct second contour
no = 0
for (i in 1:length(a1)){
  for (j in 1:length(b1.2)){
    no = no + 1
    nll.grid[no,1] = a1[i]
    nll.grid[no,2] = b1.2[j]
    nll.grid[no,3] = nll.mle(a=a1[i],b=b1.2[j],sd=2.06)
  }
}
z1.2 = as.matrix(dcast(nll.grid,Var1~Var2)[-1])

# Plot the two contours
par(mfrow = c(2,1), mar = c(0,4,1,1), las = 1)
contour(a1,b1.2,z1.2,nlevels = 20, xaxt = "n", yaxt = "n", ylim = c(0,9))
axis(2, seq(1,9,2))
points(start_a[4],start_b[4],pch=4, col = 4, lwd = 2)
points(coef(mle2.1[[1]])[1],coef(mle2.1[[1]])[2],pch=19, col = 2)
points(coef(mle2.1[[2]])[1],coef(mle2.1[[2]])[2],pch=19, col = 3)
points(coef(mle2.1[[4]])[1],coef(mle2.1[[4]])[2],pch=19, col = 4)
contour(a1,b1.2,z1.2,levels=120,col=2,add=T)

```

```

par(mar = c(3.5,4,0.5,1))
contour(a1,b1.1,z1.1,nlevels = 20)
points(coef(best_mle2.1)[1],coef(best_mle2.1)[2],pch=19)
points(start_a[1],start_b[1],pch=4, col = 2, lwd = 2)
points(start_a[2],start_b[2],pch=4, col = 3, lwd = 2)
points(start_a[3],start_b[3],pch=4, col = 1, lwd = 2)
contour(a1,b1.1,z1.1,levels=120,col=2,add=T)

# profile
nll.mle1 = function(a,sd){
  # this calculates the mean y for a given value of x: the deterministic function
  mu = a*(1-exp(-b*x))
  # this calculates the likelihood of the function given the probability
  # distribution, the data and mu and sd
  nll = -sum(dnorm(y,mean=mu,sd=sd,log=T))
  return(nll)
}

nll = numeric(length(b1.1))
for (i in 1:length(b1.1)){
  b = b1.1[i]
  mle.21 = mle2(nll.mle1,start=list(a=25,sd=7.96),data=data.frame(x=shapes1$x,y=shapes1$y),m
  nll[i] = -logLik(mle.21)
}
par(mfrow = c(1,1))
plot(nll~ b1.1,type="l",xlim=c(0.008,0.012), ylim = c(117,125))
which.min(nll)

# cutoff
-logLik(best_mle2.1) + qchisq(0.95, 1)/2
which(nll < 119.852)
b1.1[c(23,35)]

plot(nll~ b1.1,type="l",xlim=c(0.0070,0.012),ylim=c(116,125))
abline(v=c(0.00744,0.01096),lty=2)
abline(v=0.008968,lty=1,lwd=2)
abline(v=c(0.00738,0.01103),lty=2,col="red")

se.mu = sqrt(diag(solve(best_mle2.1@details$hessian))[2])
b + c(-1,1)*qnorm(0.975) * se.mu
confint(best_mle2.1)

abline(v=c(0.007177,0.0107589),col="blue")

```

6.3 Bayesian parameter estimation: negative binomial

In this section we will practice parameter estimation using the Bayesian method on the same negative binomial example as before (Section 6.1). The purpose of this exercise is to gain intuition of how Markov Chain Monte Carlo (MCMC) algorithms work and better understand the differences (and similarities) between maximum likelihood and Bayesian parameter estimation. The MCMC algorithm implemented below can be useful for relatively simple models such as the ones covered in this course. For more complex data analysis we recommend to use dedicated R packages that implement more powerful (and automated) algorithms. A list of such packages can be found in the task view on Bayesian Inference (<https://cran.r-project.org/web/views/Bayesian.html>).

6.3.1 From Bayes rule to log posterior

The aim of Bayesian analysis is to estimate the parameters of a model conditional on observed data ($P(\theta|D)$, known as *posterior distribution*) given the likelihood ($L(\theta|D) = P(D|\theta)$) and *prior distributions* of the parameters ($P(\theta)$), according to Bayes rule:

$$P(\theta|D) = \frac{P(D|\theta)P(\theta)}{P(D)}$$

Details on Bayes rule are given in section 4.3 and 6.2.2 of the book. Note that the only unknown in the right hand side of Bayes rule is $P(D)$. However, we know that $P(D) = \int P(D|\theta)P(\theta)d\theta$. Therefore, in order to calculate the posterior distribution, we could calculate this integral. Any integration method would work, but integration will not be feasible for a large number of parameters. In practice, a more popular approach is to generate samples from the posterior distribution, while avoiding the integral. This is achieved by so-called Markov Chain Monte Carlo (MCMC) algorithms. These algorithms will provide a random sample from the posterior distribution given a formulation of the problem as:

$$\log(P(\theta|D)) \propto \log(P(D|\theta)) + \log(P(\theta)) = \mathcal{L} + \log(P(\theta))$$

Where \mathcal{L} is the positive log-likelihood and \propto means “proportional to”. These algorithms work with logarithms for the same reason as in maximum likelihood estimation (i.e., to avoid numerical instability due to very large or very small numbers that would result from multiplication).

The first step of Bayesian parameter estimation is to build a function that calculates the log-posterior density for every parameter value. We will use the example of the negative binomial from section 6.1. This example fits a negative binomial distribution parameterized by its mean (μ) and size (k) both of which have to be positive.

In a Bayesian approach, we need to assign prior probabilities to each of the parameters, which means choosing a distribution and its parameters, based on prior knowledge. Of course, without a context, it is not possible to specify meaningful prior distributions (and this is

arguably the hardest step in any Bayesian analysis), but for the sake of this exercise let's assume that we can represent our prior beliefs with Normal distributions centered around 0 and with a standard deviation of 2 (in practice only half of these prior distributions are being used as μ and k are positive, but that is fine). This essentially means that we are 99% certain that μ and k will be lower than 4.6, prior to seeing any data.

We have to construct a function that can return the sum of the log-likelihood and log-prior densities for a given combination of μ and k in order to use MCMC (remember, this is not the exact log-posterior because of the unknown normalizing constant):

```
LPfun1 = function(p, dat = x) {
  # Mean and size of the negative binomial (use exp to force them to be positive)
  mu = exp(p[1])
  k = exp(p[2])
  # Logarithm of the prior distributions on mu and k
  # (0 and 2 are parameters chosen by the user, they represent prior beliefs)
  lp_mu = dnorm(mu, 0, 2, log = TRUE)
  lp_k = dnorm(k, 0, 2, log = TRUE)
  log_prior = lp_mu + lp_k
  # Log-likelihood of the data under the model
  LL = sum(dnbinom(dat, mu=mu, size=k, log=TRUE))
  # Sum of the log-likelihood and the log-prior
  LL + log_prior
}
```

The main difference between `LPfun1` and `NLLfun1` created in section 6.1 is that the new function includes the log-prior densities of μ and k (`lp_mu` and `lp_k`, respectively) and that it returns the sum of log-likelihood + log-prior densities.

6.3.2 Sampling from posterior: Metropolis-Hastings algorithm

Below is a simple version of the Metropolis-Hastings algorithm (section 7.3.1 of the book), with a multivariate Normal proposal distribution (**you need to install package `mvtnorm` first!**). Note that this function is written in a generic fashion, that is, it will work with any user-defined function that is assigned to the first argument (`model`) and any data required by said function is passed through the `...` argument (this is the strategy is used in many R functions, including `optim`).

The inputs of the MH function (see below for code) are:

- **model**: Function that calculates the non-normalized log-posterior (i.e. `LPfun1`).
- **init**: Initial values for the parameters. The closer to the “true” values the faster the MCMC algorithm will converge to the posterior distribution.
- **sigma**: Variance-covariance matrix of the proposal distribution used to calculate jumps in parameter space.
- **niter**: Number of iterations the algorithm will run for.
- **burn**: Fraction of iterations that will be used as burn-in (check section 7.3.2). These iterations will not be used for analysis but are required for convergence of the MCMC

algorithm.

- **seed**: Seed for pseudo-random number generator that allows reproducing results.

The algorithm keeps track of all the parameter values it visits and stores them in the variable **chain**. Each iteration, it proposes new values for each parameter (**proposal**) sampled from a multivariate Normal distribution centered at the current values. The probability of accepting the proposal is equal to the exponent of the difference in log posterior densities (**paccept**, see Equation 7.3.2 in the book, taking into account that the proposal distribution is symmetric). If the proposal is accepted, then it is added to the **chain** and becomes the new **current** values (i.e., the algorithm “moves” to that location). After the run is finished, the values in **chain** are split between the burn-in samples and after burn-in. The variable **acceptance** calculates the fraction of jumps that were accepted (do not confuse with the probability of accepting an individual jump!). The higher this number is, the more efficient the algorithm is in exploring the posterior distribution.

```
library(mvtnorm)

## Warning: package 'mvtnorm' was built under R version 4.0.5

MH = function(model, init, Sigma = diag(init/10), niter = 3e4, burn = 0.5,
              seed = 1134, ...) {
  # To make results reproducible you should set a seed (change among chains!!!)
  set.seed(seed)
  # Pre-allocate chain of values
  chain = matrix(NA, ncol = length(init), nrow = niter)
  # Chain starts at init
  current = init
  lp_current = model(current, ...)
  # Iterate niter times and update chain
  for(iter in 1:niter) {
    # Generate proposal values from multivariate Normal distribution
    proposal = rmvnorm(1, mean = current, sigma = Sigma)
    # Calculate probability of acceptance (proposal distribution is symmetric)
    lp_proposal = model(proposal, ...)
    paccept = min(1, exp(lp_proposal - lp_current))
    # Accept the proposal... or not!
    # If accept, update the current and lp_current values
    accept = runif(1) < paccept
    if(accept) {
      chain[iter,] = proposal
      lp_current = lp_proposal
      current = chain[iter,]
    } else {
      chain[iter,] = current
    }
  }
  # Calculate the length of burn-in
```



```

nburn = floor(niter*burn)
# Calculate final acceptance probability after burn-in (fraction of proposals accepted)
acceptance = 1 - mean(duplicated(chain[-(1:nburn),]))
# Package the results
list(burnin = chain[1:nburn,], sample = chain[-(1:nburn),],
      acceptance = acceptance, nburn = nburn)
}

```

So let's tackle the negative binomial problem with the algorithm above. First, let's regenerate the data that was used in the previous section:

```

set.seed(1001)
mu.true=1
k.true=0.4
x = rnbinom(50,mu=mu.true,size=k.true)

```

Now we can run MH with some values. I want to make the point that choosing a good proposal distribution matters for an efficient MCMC algorithm. So let's start with a variance-covariance matrix that is not reasonable (because it is too wide):

```
Sigma = diag(c(10,10))
```

Now we can run MH combined with the LPfun1 function and some initial values:

```

init = log(c(1,1))
bay1 = MH(LPfun1, init, Sigma, burn = 0.3, dat = x)

```

The first result you want to check is the acceptance probability to see how succesful proposals were:

```
bay1$acceptance
```

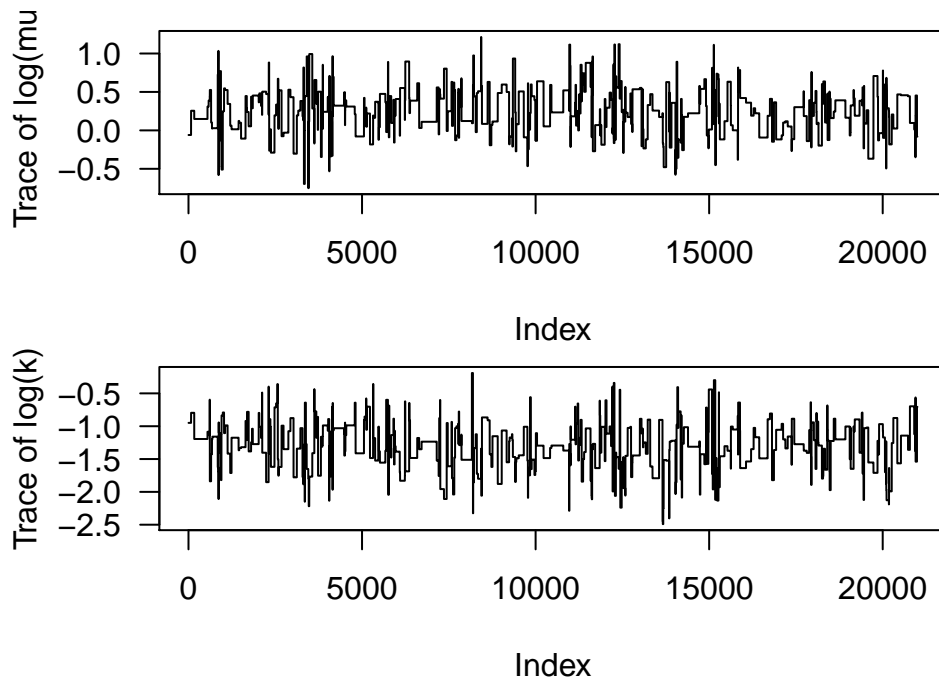
```
## [1] 0.01957143
```

This is terrible! 98% of the proposed values were rejected so it would take really long to get a representative sample from the posterior distribution. The next step is usually to take a look at the traces of the values sampled by the MCMC (noticed that the sampling was done on the log transformation of the parameters as they are positive):

```

par(mfrow = c(2,1), mar = c(4,4,0.5,0.5), las = 1)
plot(bay1$sample[,1], t = "l", ylab = "Trace of log(mu)")
plot(bay1$sample[,2], t = "l", ylab = "Trace of log(k)")

```



The low probability of acceptance means that the traces look like “squiggly lines”, getting stuck at different values for hundreds of iterations (i.e., horizontal sections in the traces). This slows down the effective sampling and can introduce biases in the estimates (unless the algorithm runs for very long).

Note that more modern MCMC algorithms (that R packages specialized on Bayesian statistics will use internally) will automatically *tune* the proposal distribution or even use alternatives methods to propose values that are more robust. However a poor man’s tuned MCMC may suffice for this introduction (and for simple models with few parameters) and it works as follows:

1. Calculate the value that maximizes the posterior distribution using `optim` (*a.k.a* Maximum A Posteriori estimate or MAP for short).
2. Estimate the variance-covariance matrix of the posterior distribution using the Hessian matrix returned by `optim` (analogous to what we do for maximum likelihood estimation).
3. Run MH using the above results as the values for `init` and `sigma`, respectively.

The reason why this works better is because points 1 and 2 will often give a good first approximation of the posterior distribution, especially for large data (in which case the posterior distribution approaches a Normal distribution). This means that MH will be sampling from a distribution similar to the target distribution and hence a higher proportion of proposals will be accepted (intuitively, fewer values that are far in the tails of the posterior distribution will be proposed). This approach can be implemented as:

```
mapfit = optim(fn = LPfun1, par = log(c(1,1)),
              hessian = TRUE, method = "BFGS",
              control = list(fnscale = -1), dat = x)
```

```

Sigma = solve(-mapfit$hessian)
init = mapfit$par
bay2 = MH(LPfun1, init, Sigma, burn = 0.3)

```

Notice that we should use `control = list(fnscale = -1)` because we want to maximize the posterior probability, not minimize it. That is also the reason why I add a negative sign in front of the Hessian as in `solve(-mapfit$hessian)` (in previous examples we were minimizing the negative log likelihood and we did not include the negative sign in front of the Hessian).

We can see that the matrix `Sigma` obtained from the Hessian around the MAP estimate is different from the one assumed in the first MH run (specifically, the variances are much smaller):

```

Sigma

##              [,1]      [,2]
## [1,] 0.078897091 0.002920995
## [2,] 0.002920995 0.122861528

```

These lower variances mean that the Markov chain does not wander far into the tails of the posterior distribution but rather remains in the area of high probability. Thus, the new run has a higher acceptance probability:

```

bay2$acceptance

```

```

## [1] 0.5599524

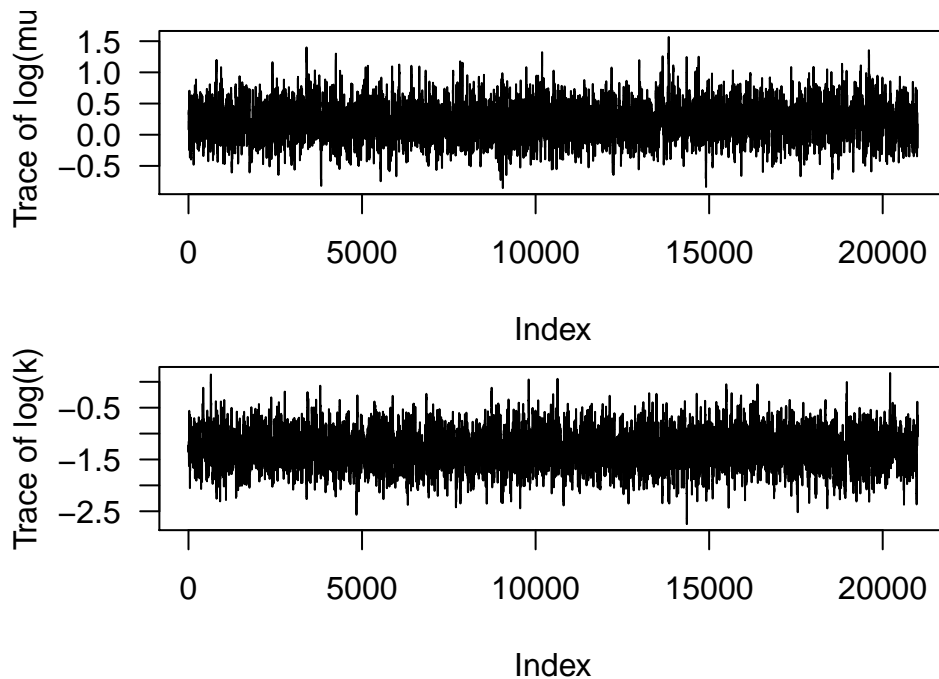
```

Now this is the good. 56% of the time the candidates will be accepted, ensuring that the chain samples efficiently from the posterior. The traces will approach white noise (these are often called “fuzzy caterpillars” in the community):

```

par(mfrow = c(2,1), mar = c(4,4,0.5,0.5), las = 1)
plot(bay2$sample[,1], t = "l", ylab = "Trace of log(mu)")
plot(bay2$sample[,2], t = "l", ylab = "Trace of log(k)")

```

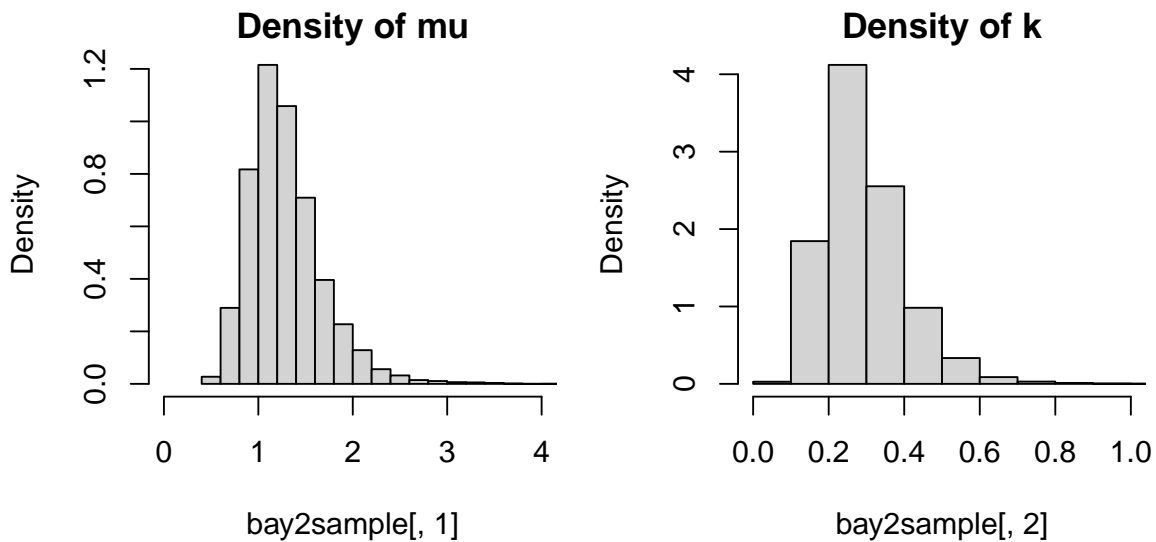


At this point we would normally calculate more diagnostics to build up more confidence on the results of the MCMC chains, but we will keep it simple in this introduction. The object `bay2$sample` contains a random sample from the posterior from which we can calculate several properties. First, remember that we took the logarithm of the parameters to avoid negative values, so we need to undo this transformation in the result:

```
bay2sample = exp(bay2$sample)
```

We can visualize the estimates for each parameter using `density` (more common) or `hist` (easier to interpret):

```
par(mfrow = c(1,2), mar = c(4,4,1.5,1))
hist(bay2sample[,1], main = "Density of mu", freq = F, xlim = c(0,4))
hist(bay2sample[,2], main = "Density of k", freq = F, xlim = c(0,1))
```



One striking feature is that the distributions are not symmetric, they have a longer tail to the right. This is typical of positive parameters that are close to 0. A consequence of this is that the mean, median and mode of the distributions will differ (though in this case not so much). Let's compare all the estimates we have so far for the negative binomial fitted to these data:

```
map = exp(mapfit$par)
meanp = colMeans(bay2sample)
medianp = c(median(bay2sample[,1]), median(bay2sample[,2]))
cbind(map, meanp, medianp,
      mom = c(mu.mom, k.mom),
      mle = opt1$par,
      true = c(mu.true, k.true))

##           map      meanp  medianp      mom      mle true
## mu 1.2209093 1.2897290 1.222248 1.2600000 1.2602356 1.0
## k  0.2876172 0.2926038 0.274538 0.3778531 0.2884793 0.4
```

For this model, data and priors, all estimates are quite similar to each other across different methods of estimation. The reason is because there is sufficient data (50 points for 2 parameters is quite some data...) such that the priors have a negligible effect.

The 95% credible intervals (analogous to 95% confidence intervals) can be calculated with the `quantile` function applied directly to the sample from the posterior:

```
t(apply(bay2sample, 2, quantile, probs = c(0.025, 0.975)))

##           2.5%      97.5%
## [1,] 0.7078096 2.2138543
## [2,] 0.1348619 0.5519341
```

7 Hints for choosing deterministic functions and stochastic functions

1. Deterministic functions

- dataset 1

light response curve. There are a number of options of functions to choose from, depending on

$\frac{ax}{(b+x)}$, $a(1 - e^{(-bx)})$, $\frac{1}{2\theta}(\alpha I + p_{max} - \sqrt{(\alpha I + p_{max})^2 - 4\theta I p_{max}})$ see page 98. A parameter d can be added in all cases to shift the curve up or down. The y represents net photosynthesis $\mu\text{molCO}_2/\text{m}^2\text{s}$

- dataset 2

The dataset describes a functional responses. Bolker mentions four of those

$$\min(ax, s) \quad \frac{ax}{(b+x)}, \quad \frac{ax^2}{(b^2+x^2)}, \quad \frac{ax^2}{(b+cx+x^2)}$$

The y is measured in grams prey eaten per unit time.

-dataset 3

Allometric relationships generally have the form ax^b . The y represent the total number of cones produced.

-dataset 4

This could be logistic growth $n(t) = \frac{K}{1+(\frac{K}{n_0})e^{-rt}}$ or the gompertz function $f(x) = e^{-ae^{-bx}}$. The y represent the population size (numbers).

-dataset 5

What about a negative exponential? ae^{-bx} or a power function ax^b . The y represent a number per unit area.

-dataset 6

Species reponse curves are curves that describe the probability of presence as a function of some factor. A good candidate good be a unimodal response curve. You could take the equation of the normal distribution without the scaling constant: e.g. $ae^{-\frac{(x-\mu)^2}{2\sigma^2}}$. The y represent presence or absence of the species (no units).

2. Stochastic functions/Probability distributions

-dataset 1 y represents real numbers and both positive and negative numbers occur. This implies that we should choose a continuous probability distribution. In addition, the numbers seems unbound. Within the family of continuous probability distributions, the normal seems a good candidate distribution because this one runs from $-\infty$ to $+\infty$. In contrast the Gamma and the Lognormal only can take positive numbers, so these distributions cannot handle the negative numbers. In addition, the beta distribution is not a good candidate because it runs from 0-1.

-dataset 2 y represents real numbers and only positive numbers occur. The data represents a functional response (intake rate of the predator), and it is likely that you can only measure positive numbers (number of prey items per unit of time). This implies that we should choose a continuous probability distribution. Within the family of continuous probability distributions, the Gamma and the Lognormal could be taken as candidate distributions because they can only take positive numbers (beware that the Gamma cannot take 0). However, you could try to use a normal as well.

-dataset 3 y seems represents counts (this is the cone dataset that is introduced in ch. 6.). Given that it contains counts we can pick a distribution from the family of discrete distributions. The Poisson and the Negative Binomial could be good candidates to describe this type of data.

-dataset 4 y represents population size over time. From looking at the data, they seems to represent counts. Given that it contains counts we can pick a distribution from the family of discrete distributions. The Poisson and the Negative Binomial could be good candidates to describe this type of data.

-dataset 5 No information is given on y . The data clearly seems to represent counts. Thus the same reasoning applies here as to the two previous datasets.

-dataset 6 The data (y) represents species occurrences (presence/absence). The binomial model would be a good model to predict the probability of presence.