Likelihood basics

Allan Clark

Department of Statistical Sciences, University of Cape Town

23 February 2024

Likelihood

- Assume we observe the outcome of many coin tosses. e.g HTTH. . . T
- A data model for each experiment is denoted as

$$X_i \sim Bern(\theta)$$

  where θ is

$$Pr(X_i = H) = \theta.$$

- Here X_i represents the outcome of the i^{th} experiment.
- We assume that θ is constant for all experiments.

We now have the following joint probabilities for the following three cases:

$$Pr(X_1 = H) = \theta$$

$$Pr(X_1 = H,\ X_2 = T) = \theta(1-\theta)$$

$$Pr(X_1 = H,\ X_2 = T,\ X_3 = T) = \theta(1-\theta)(1-\theta)$$

Likelihood

Assume we had observed n experiments. In this case the joint probability mass function is

$$p(x_1, x_2, \ldots, x_n) = p(x_1) \times p(x_2) \times \ldots \times p(x_n).$$

The above mass function is in fact a function of θ and is denoted as

$$\begin{aligned} p(x_1, x_2, \ldots, x_n|\theta) &= p(x_1|\theta) \times p(x_2|\theta) \times \ldots \times p(x_n|\theta) && (1) \\ &= \prod p(x_i|\theta). && (2) \end{aligned}$$

Likelihood

- Above, we assumed that each experiment is independent!
- Equation 2 is termed the likelihood function!

In general,

$$p(x|\theta) = \theta^x(1-\theta)^{1-x}$$

where $x = 0$ or $x = 1$. The likelihood function thus simplifies to

$$p(x_1, x_2, \ldots, x_n|\theta) = \prod_i \theta^{x_i}(1-\theta)^{1-x_i} = \theta^{\sum_i x_i}(1-\theta)^{n-\sum_i x_i} \quad (3)$$

and is often denoted as $L(\boldsymbol{x}, \theta)$.

Maximum likelihood estimation

- Maximum likelihood estimation is the process of using data to estimate the value of θ by maximising the likelihood function.

```
#data
x <- c(0,1,1,1,1,1,1,0,0,1,1,1,0,1,1)

#likelihood function
likelihood <- function(theta, x){
  #returns the likelihood value
  #one could also work with the loglikelihood value
  sx <- sum(x)
  n <- length(x)

  (theta^sx)*( (1-theta)^(n-sx) )
}
```

Maximum likelihood estimation

```r
#optim by default does minimisation.
#fnscale=-1 multiplies 'the'likelihood' by -1

optim(0.5, fn = likelihood, x=x,
      control=list(fnscale=-1),
      method= "Brent", lower=0, upper=1)
```

```
## $par
## [1] 0.7333333
##
## $value
## [1] 0.0001667979
##
## $counts
## function gradient
##       NA       NA
##
## $convergence
## [1] 0
##
## $message
## NULL
```

Maximum likelihood estimation

Notice that the maximum likelihood estimate is

$$\hat{\theta} = 0.7333333 = \frac{11}{15}.$$

If we work with the logarithm of the likelihood function (loglikelihood function) we will get the same estimators!

```
#loglikelihood function
logl <- function(theta, x){
  sx <- sum(x)
  n <- length(x)

  sx*log(theta) + (n-sx)*log(1-theta)
}
```

Maximum likelihood estimation

Notice we obtain the same value!

```r
optim(0.5, fn = logl, x=x,
      control=list(fnscale=-1),
      method= "Brent", lower=0, upper=1)
```

```
## $par
## [1] 0.7333333
##
## $value
## [1] -8.698728
##
## $counts
## function gradient
##       NA       NA
##
## $convergence
## [1] 0
##
## $message
## NULL
```

Exercise - using optim

Assume that you observed $n = 100$ observations from an exponential random variable with rate parameter λ. Assume that $\lambda = 2$. Recall that $f(x|\lambda) = \lambda e^{-\lambda x}$ for $x > 0$.

1. Set the seed to be `1`. Simulate a random data set to replicate the above scenario.
2. Plot the histogram of your data and superimpose the probability density function using the assumed value of λ.
3. Use pencil and paper to derive the likelihood function. Write down the log of the likelihood function.
4. Write a function named `logl` that can be used to calculate the loglikelihood. Name the arguments to your function `lambda` and `xData`. Note `lambda` should be the first argument.
5. Use `R` to estimate the value of `lambda` that maximises the loglikelihood function. Read `optim` for help here!
6. Use your estimated value of λ and superimpose the estimated probability density function on your histogram.

Exercise - using optim

```r
#1.

set.seed(1)
n <- 100
xData <- rexp(n, rate = 2)

cat("\n mean(X) = ", mean(xData))
```

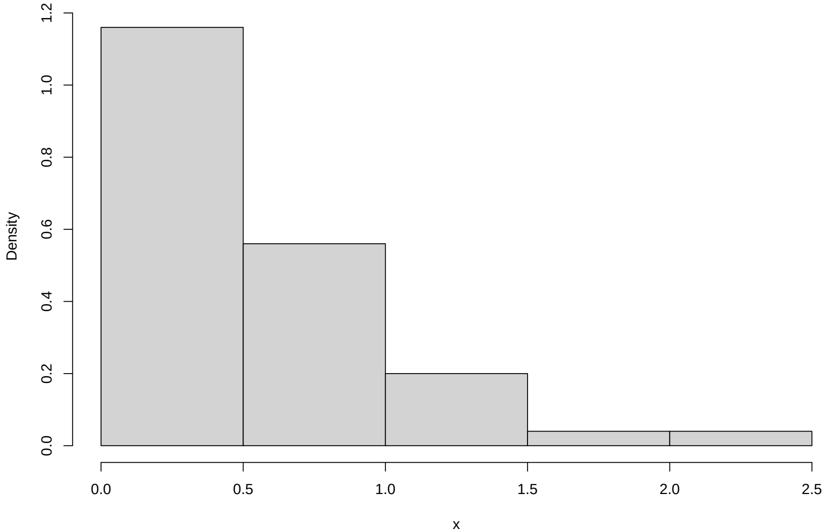
```
##
##  mean(X) =  0.5153382
```

```r
cat("\n var(X) = ", var(xData))
```

```
##
##  var(X) =  0.2192497
```

Exercise - using optim

```r
#2.

hist(xData, main="", xlab="x", prob=TRUE)
```



Exercise - using optim

The loglikelihood function is

$$\text{logl} = n \log(\lambda) - \lambda \sum_i x_i.$$

```
#5.
#First argument must be the parameter of your model.

logl <- function(lambda, xData){
  #return the negative of the loglilihood function
  #lambda = the rate parameter
  #xData = the data used

  n <- length(xData)

  loglp <- n * log(lambda) - lambda * sum(xData)

  return( -loglp )
}
```

Exercise - using optim (there are many different routines that could be used!)

```r
#set a starting value of 5
#this happens to find a solution!
#note we have not told R that the parameter has to be positive
optim(5, fn = logl, xData = xData)
```

```
## Warning in optim(5, fn = logl, xData = xData): one-dimensional optimization
## use "Brent" or optimize() directly

## $par
## [1] 1.94043
##
## $value
## [1] 33.70681
##
## $counts
## function gradient
##       32       NA
##
## $convergence
## [1] 0
##
## $message
## NULL
```

Exercise - using optim - default is "Nelder-Mead"

```r
#set a starting value of 0.5
#this happens to find a solution!
#note we have not told R that the parameter has to be positive
optim(0.5, fn = logl, xData = xData)
```

```
## Warning in optim(0.5, fn = logl, xData = xData): one-dimensional optimizatio
## use "Brent" or optimize() directly

## $par
## [1] 1.940625
##
## $value
## [1] 33.70681
##
## $counts
## function gradient
##       32       NA
##
## $convergence
## [1] 0
##
## $message
## NULL
```

Exercise - using optim method = "Brent"

```r
optim(5, fn = logl, xData = xData, method = "Brent")

#you receive the following error
#Error in optim(5, fn = logl, xData = xData, method = "Brent") :
#'lower' and 'upper' must be finite values
```

There are a number of optimisation methods that could be used. Brent is one such method that can be used to one-dimensional optimisation.

Exercise - using optim (set limits when using Brent)

```r
#for some reason logl(0) does not give an error!
#normally you have to be careful that your function
#evaluates to a finite value over the range of your
#function
optim(5, fn = logl, xData = xData, method = "Brent", lower=0, upper=10)
```

```
## $par
## [1] 1.940473
##
## $value
## [1] 33.70681
##
## $counts
## function gradient
##       NA       NA
##
## $convergence
## [1] 0
##
## $message
## NULL
```

Exercise - using optim

```r
#happens to give the same solution!
#a gradient based method
optim(5, fn = logl, xData = xData, method = "Nelder-Mead")
```

```
## Warning in optim(5, fn = logl, xData = xData, method = "Nelder-Mead"): one-d
## use "Brent" or optimize() directly

## $par
## [1] 1.94043
##
## $value
## [1] 33.70681
##
## $counts
## function gradient
##       32       NA
##
## $convergence
## [1] 0
##
## $message
## NULL
```

Exercise - using optim

```
#happens to give the same solution!
optim(5, fn = logl, xData = xData, method = "L-BFGS-B",
      lower=0, upper=10)

#you get an error
#Error in optim(5, fn = logl, xData = xData, method = "L-BFGS-B",
#lower = 0,  :
#  L-BFGS-B needs finite values of 'fn'
```

Have a look at the bounds of your function. You are getting an error since logl(0) is infinite!

Exercise - using optim (assigning bounds to the parameter)

```r
#set a small lower limit to solve the error
optim(5, fn = logl, xData = xData, method = "L-BFGS-B",
      lower=0.001, upper=10)
```

```
## $par
## [1] 1.940473
##
## $value
## [1] 33.70681
##
## $counts
## function gradient
##        9        9
##
## $convergence
## [1] 0
##
## $message
## [1] "CONVERGENCE: REL_REDUCTION_OF_F <= FACTR*EPSMCH"
```

Exercise - using optim

```
# optim(5, fn = logl, xData = xData, method = "L-BFGS-B",
# lower=0.001, upper=10)
# $par
# [1] 1.940473
# $value
# [1] 33.70681
# $counts
# function gradient
#        9        9
# $convergence
# [1] 0
# $message
# [1] "CONVERGENCE: REL_REDUCTION_OF_F <= FACTR*EPSMCH"
```

- par is the parameter value at which the loglikelihood function is maximised.
- convergence=0 indicates convergence of the algorithm

Exercise - using optimise is another function that can be used for 1d optimisation

```r
optimise(f = logl, xData = xData, lower=0.001, upper=10)
```

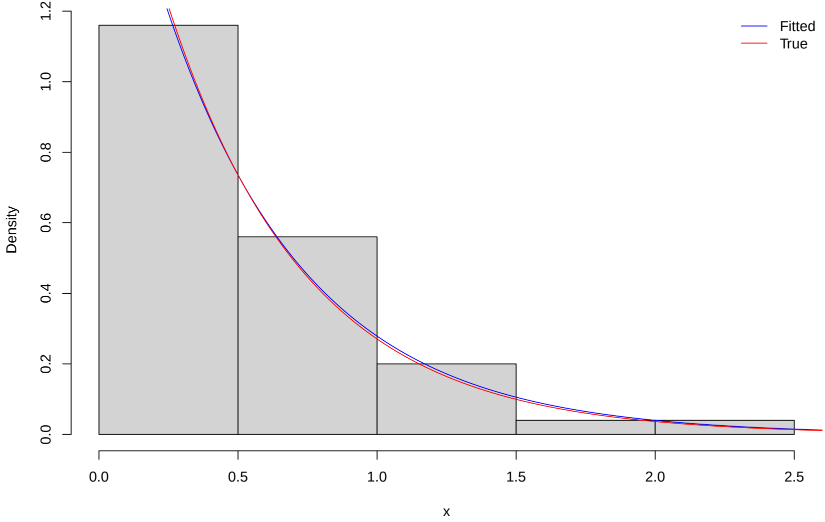
```
## $minimum
## [1] 1.940473
##
## $objective
## [1] 33.70681
```

Take note of what this function gives you! Different functions give you different outputs.

Exercise - using optimise is another function that can be used for 1d optimisation

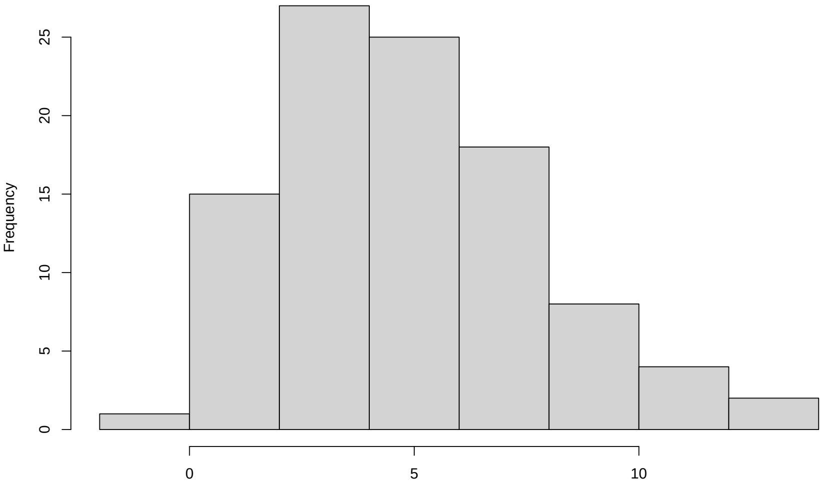
```r
opt <- optimise(f = logl, xData = xData, lower=0.001, upper=10)

hist(xData, main="", xlab="x", prob=TRUE)
xr <- seq(from=0.01, to=3, length.out=500)

#take note how the estimated parameter is extracted from the list.
yr.est <- dexp(xr, rate = opt$minimum)

lines(xr, yr.est, col="blue")

yr.true <- dexp(xr, rate = 2)
lines(xr, yr.true, col="red")

legend("topright", legend = c("Fitted", "True"), col=c("blue", "red"),
       lty=c(1,1), bty="n")
```

Exercise - using optimise is another function that can be used for 1d optimisation



Optimisation using more than one parameter

Assume that we have $n = 100$ observations from a Gaussian distribution with unknown mean (μ) and variance (σ^2). We can easily extend the previous optimisation task to a multi-parameter setting to estimate the unknown parameters.



Optimisation using more than one parameter

The loglikelihood function is

$$l = -\frac{n}{2}\log(\sigma^2) - \frac{1}{2\sigma^2}\sum_i (x_i - \mu)^2.$$

Note we can ignore terms independent of μ and σ^2.

When coding up this loglikelihood function, we set the first argument of our function, logl, to represent a two-dimensional vector and include any additional variables required to undertake the optimisation.

Optimisation using more than one parameter

```
logl <- function(par, xData){
  #return the loglikelihood function for Gaussian data
  #par = c(mu, s)
  #mu = mean(X), s = sd(X)

  #1st param of par argument = mean
  #2nd paramr of par argument = sd
  mu <- par[1]
  s <- par[2]

  #the variance of X
  s2 <- s^2

  logl <- -n*0.5 * log(s2) - 0.5*sum( (xData-mu)^2 )/s2
  return( logl[1] )
}
```

Optimisation using more than one parameter

```
opt <- optim(c(5,5), fn=logl,
             method="L-BFGS-B",
             lower=c(-Inf, 0),
             upper=c(Inf, Inf),
             xData = xData,
             control = list(fnscale = -1))
```

- Take note that a vector has to be supplied as the starting value for the algorithm. ie. c(5,5).
- Note how the lower and upper arguments are specified.
- lower are the lower bounds of the two parameters.
- upper are the upper bounds of the two parameters.
- list(fnscale = -1) implies that the loglikelihood is multiplied by -1.

Optimisation using more than one parameter

```
opt
```

```
## $par
## [1] 4.926770 2.856681
##
## $value
## [1] -154.966
##
## $counts
## function gradient
##       10       10
##
## $convergence
## [1] 0
##
## $message
## [1] "CONVERGENCE: REL_REDUCTION_OF_F <= FACTR*EPSMCH"
```

Optimisation using more than one parameter

```r
#the mean and sd estimates
cat("\n Parameter estimate = ", opt$par)
```

```
##
##  Parameter estimate =  4.92677 2.856681
```

```r
#the value is 0 and thus the algorithm converged
cat("\n Convergence message = ", opt$convergence)
```

```
##
##  Convergence message =  0
```

```r
#check the solutions
cat("\n mean = ", mean(xData))
```

```
##
##  mean =  4.92677
```

Optimisation using more than one parameter

In the above example we constrained the optimisation by using the 'L-BFGS-B' method. For some optimiation problems it might be useful to not use a constrained optimisation routine to maximise the particular likelihood. In this case, it would be useful to transform the constrained parameters so that they are unconstrained. i.e. $-\infty, \infty$.

Specifically, we would transform from σ to σ^* where $\sigma^* = \log(\sigma)$.

Optimisation using more than one parameter

```r
logl <- function(par, xData){
  #return the loglikelihood function for Gaussian data
  #par = c(mu, s*)
  #mu = mean(X), s = log(sd(X))

  #1st param of par argument = mean
  #2nd paramr of par argument = log(sd)
  mu <- par[1]

  #transform from sigma* to sigma
  s <- exp(par[2])

  #the variance of X
  s2 <- s^2

  n <- length(xData)

  logl <- -n*0.5*log(s2) - 0.5*sum( (xData-mu)^2 )/s2
  return( logl[1] )
}
```

Optimisation using more than one parameter

```
opt2 <- optim(c(5,log(5)), fn=logl,
              method="BFGS",
              xData = xData,
              control = list(fnscale = -1))
```

Optimisation using more than one parameter

```r
#the mean and sd estimates
#they are basically the same as before!!!!
#take note the sigma estimate!
cat("\n Parameter estimate = ", c(opt2$par[1],
                                      exp(opt2$par[2])) )
```

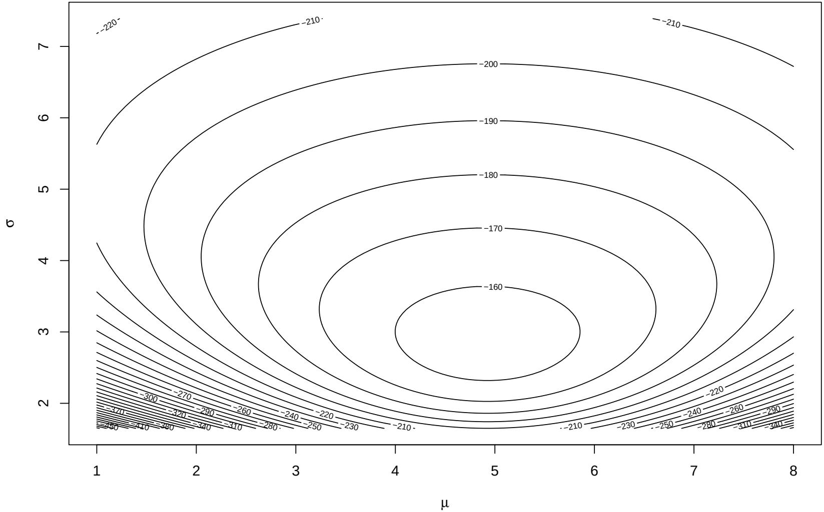
```
##
##  Parameter estimate =  4.927027 2.856644
```

```r
#the value is 0 and thus the algorithm converged
cat("\n Convergence message = ", opt2$convergence)
```

```
##
##  Convergence message =  0
```

Contour plots

Sometimes it is useful to plot a two dimensional (contour) plot that displays the contours of the loglikelihood function to see what the surface looks like.



Contour plots

Above we see that maximum of the loglikelihood function occurs at a value of μ somewhere between 4 and 5, and σ somewhere between 3 and 4.

Contour plots

```
#range of mu and s.star
xr.mu <- seq(from=1, to=8, length.out=100)
xr.s.star <- seq(from=0.5, to=2, length.out=100)

logl.c <- function(mu, s.star, xData){
  #return the loglikelihood function for Gaussian data
  #par = c(mu, s*); mu = mean(X), s = log(sd(X))

  #transform from sigma* to sigma
  s <- exp(s.star)
  #the variance of X
  s2 <- s^2
  n <- length(xData)

  temp <- 0
  for (i in 1:n){ temp <- temp + (xData[i]-mu)^2 }
  logl <- -n*0.5*log(s2) - 0.5*temp/s2

  return( logl[1] )
}
```

Contour plots

We *vectorize* **logl.c** so that we are able to evaluate **logl.c** using the **outer** function. The outer function evaluates a function at all pairs of *xr.mu* and *s.star* and produces a matrix of values.

```
logl.vec <- Vectorize(FUN=logl.c, vectorize.args = c("mu", "s.star"))

outs <- outer(X=xr.mu, Y=xr.s.star, FUN=logl.vec, xData=xData)

#produce the contour plot
contour(xr.mu, exp(xr.s.star), outs,
        nlevel=30, #the number of levels of the plot
        xlab=expression(mu),
        ylab=expression(sigma))
```

Outer function (example)

Lets quickly look at the *outer* function.

```
x1 <- 1:3
x2 <- 5:7

func1 <- function(x,y){
  x*y
}

outer(X=x1, Y=x2, FUN=func1)
```

```
##      [,1] [,2] [,3]
## [1,]    5    6    7
## [2,]   10   12   14
## [3,]   15   18   21
```

Outer function (example)

The *outer* function requires two inputs X and Y and applies these values to your function.

X and Y should be vectors such that all element pairs are evaluated by the *outer* function.

Previously we saw that the first row of the output is 5, 6 and 7.

This is because the first row of the output matrix is 1×5, 1×6 and 1×7. The third row similarly is 3×5, 3×6 and 3×7.

The *outer* function evaluates much faster than using a nested loop to produce the above output and is preferred.