Defining a likelihood function

The likelihood function is related to the probability distribution function, but its view point is turned around. Take the binomial distribution:

$$p(k;p,n) = P(X = k) = \binom{n}{k} p^k (1-p)^{n-k}$$

This is written as a function of k with parameter values p and n (the sample size). Assuming you know how big your sample is – you did count the data – we still have p unknown. The probability distribution function allows you to compute probabilities with p is known. In statistics though – p is unknown, whereas the sample data is known.

A sample is usually assumed to be a *random sample* where each number is assumed to be a realization *indpendent* random variables. To the layman, independent means multiply, so the probability distribution of taking m realizations of the binomial would have distribution:

$$p(k_1,k_2,....,k_m;p,n) = \binom{n}{k_1} p^{k_1}(1-p)^{n-k_1} \cdots \binom{n}{k_m} p^{k_m}(1-p)^{n-k_m}.$$

The likelihood method at its basic level just reverses how we view the above function. Rather than thinking the values k_1, ... depend on p, we think instead knowledge of the k influences what we think p is. So we write instead

$$L(p) = p(k1,k2,....,k_m;p,n)$$

The likelihood method for estimating parameters has one choose the value of p that maximizes this probability – that is roughly it is the most likely value of p for the given data.

The logarithm

The logarithm is a mathematical function with three very useful properties for us:

1. It turns products into sums
2. It makes really small numbers into much bigger negative numbers

3. It takes positive numbers to the entire real line

The first two properties are why we consider the negative log likelihood instead – the small numbers are now large and the products that come from independence are replaced with mathematically easier sums. The penalty? We need to minimize instead of maximize, but this just means we need to keep this straight – the work is no harder.

For example, the negative log likelihood above in R becomes:

```
> Lk <- function(p) prod(dbinom(k, prob=p, size=n))
```

This assumes the data is in the variable k and n is the variable n

For example, here is some simulated data:

```
> p <- .25; n <- 25
> k <- rbinom(20, prob=p, size=n)
```

To graph `Lk` we have one issue – it isn't vectorized so making a bunch of values at once requires an extra step. Then we can graph `Lk` with

```
> ps <- seq(0.01, 0.99, by=.01)
> plot(ps, sapply(ps, Lk), type="l")
```

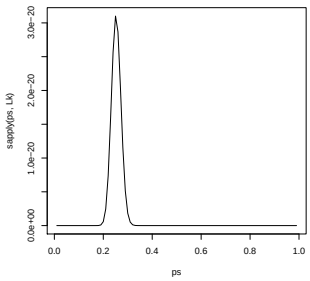

Figure 1: Graph of Likelihood function

Question 0.1. The function `sapply` simply calls the specified function for each value in the vector of numbers and packages the answers up into as tidy an output as possible. In the above, it is a data vector. We could aslo have done similar things with a loop. Here is how:

```
> ys <- numeric(length(ps))
> for(i in 1:length(ps)) {
+   ys[i] <- Lk(ps[i])
+ }
```

Do so, then check that plotting the `ps` against the `ys` gives the same graphic.

For this graph we look to maximize, however note the y axis – the numbers are tiny. The issue here is the computer is then sensitive to round off error. So, we take logs:

```
> negLL <- function(p) -log(Lk(p))
```

But that is not good either, rather we do the math first, then write. This turns products into sums, so we have:

```
> negLL <- function(p) -sum( log(dbinom(k, size=n, prob=p)))
```

Or using an argument for `dbinom`:

```
> negLL <- function(p) -sum(dbinom(k, size=n, prob=p, log=TRUE))
```

Question 0.2. Verify that all three of the styles above give the same answers by computing `negLL(0.5)` for each. Did it work?

Plotting `negLL` gives a graph where the minimum value is of interest

```
> plot(ps, sapply(ps, negLL), type="l")
```

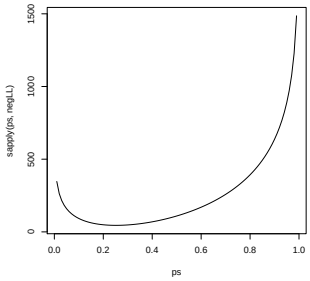

Figure 2: Plot of negative log likelihood

Using optim to find minimum values

We can eyeball the minimum point, but to get values we need to be able to optimize the value.

Optimization in most cases can be tricky – it often requires good estimates for the parameters. Most computer algorithms implemented are much more complicated than possible, as often they first use some slow method to get close, then a fast method to finish off. In fact, there are many methods for doing optimization – a fact that may need to be considered to get an answer in some cases.

The basic R function for optimization we consider here is `optim`. Later, the `mle2` function will be illustrated. Here we see how to use optim to get the minimum value of the negative log likelihood graph.

We need a good guess for the intial parameters. Eyeballing from the graph may lead you to guess a value. We will take 0.2.

Next, we need an R function of a single variable which may be a vector of values. We are already there in this case.

To use the default optization method then is simply:

```
> out <- optim(par=c(0.2), negLL)
```

Did it work? (It doesn't always.) We can check, does this give a value of 0:

```
> out$convergence

[1] 0
```

So far so good, the actual parameter estimated is found with

```
> out$par

[1] 0.2519922
```

Using mle2 to do optimization

We can use the `mle2` function to do the work above. The call is straightforward, but a bit different. The initial values can be specified, or found from the *default* values of the function. To use this approach we wrap our function as follows:

Otherwise, we can pass in the starting values to `mle2` through its `start` argument.

```
> f <- function(p=0.5) negLL(p)
```

Then we call the optimization with

```
> library(bbmle)
> out <- mle2(f)
> out

Call:
mle2(minuslogl = f)

Coefficients:
        p
0.2520009

Log-likelihood: -44.91
```

The output has the estimated coefficient shown. The different here is that the `out` value has various *methods* defined for it that make getting values out familiar to R users.

For example, There is randomness involved here in the estimate. How to account for that? Often a point estimate has a confidence interval associated with it. Here we have that:

```
> confint(out)

    2.5 %    97.5 %
0.2152764 0.2912618
```

A summary of the coefficients is given by:

```
> summary(out)

Maximum likelihood estimation

Call:
mle2(minuslogl = f)

Coefficients:
  Estimate Std. Error z value     Pr(z)
p 0.252001   0.019416  12.979 < 2.2e-16 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

-2 log L: 89.82914
```

There p-value, `Pr(z)` is for a hypothesis test that p is 0, so it is no suprise that it is small. What we see also is the standard error. The advantage of likelihood methods is that asymptotically things are normally distributed, so standard errors contain a lot of information.

Question 0.3. Explain why there is no surprise above.

multiple parameters

Many problems have more than one parameter. A toy case is the normal distribution with both a mean and standard devi-

ation. Imagine we have a random sample, which we generate with:

```
> mu=10; sigma=1
> x <- rnorm(25, mu, sigma)
```

What is the likelihood function? Similar to above, only it depends on two values:

```
> negLL <- function(mu, sigma) -sum(dnorm(x, mu, sigma, log=TRUE))
```

However, we have one catch that proves useful in many cases. Optimization can run into issues when boundaries are involved, such as $\sigma > 0$ as it is a standard deviation. Often a transformation proves useful. In this case, taking logs can work. So instead of `sigma` we write our function in terms of the log of `sigma`:

```
> negLL <- function(mu=0, lsigma=0) -sum(dnorm(x, mu, exp(lsigma), log=TRUE))
```

Not really harder to do – but you need to remember to do it.

Visualizing such a function requires a more complicated graphic. We use a contour plot. To make these we have to genearate x and y values and then the z values. Here is one way. Suppose we look for means in the area 5 to 15 and log sigma in the range -1 to 1:

```
> x <- seq(5, 15, length=100)
> y <- seq(-1, 1, length=100)
> d <- expand.grid(x,y); names(d) <- c("x","y")
> d$negLL <- sapply(1:nrow(d), function(i) negLL(d$x[i], d$y[i]))
```

This bit is tricky. We make a data frame of all possible combiations of the x and y variables. Then we want to call the function on these values. To do this, we have some fancy R command which for each row, does exactly that. We do this, as our negative log likelihood isn't vectorized.

Once done, we can plot using `contourplot` from the `lattice` package:

```
> library(lattice)
> print(contourplot(negLL ~ x + y, data=d))
```

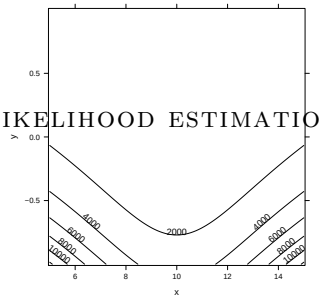

Figure 3: Contourplot of negative log likelihood. Parameter values are where this shows minimum value. Not so easy to see on this graphic.

Not too great, we should narrow down the range of values we looked at.

Question 0.4. Do that – narrow down the viewing window to make a better contour plot.

Anyways, the contour plot shows us roughly where the value is, the optimization routine tells us exactly.

Since we wrote the negative log likelihood function to have default values, the call to `mle2` works exactly as before.

The shortest call to `mle2` is to just pass in the *objective function* as done here. There are other options. For example, one can pass in a formula to specify a model, one can specify starting values, the algorithm for optimization, a list of fixed parameter values (so these are not optimized over), etc.

```
> out <- mle2(negLL)

> coef(out)

      mu   lsigma
9.999738 1.070227

> summary(out)

Maximum likelihood estimation

Call:
mle2(minuslogl = negLL)

Coefficients:
       Estimate Std. Error z value     Pr(z)
mu     9.999738   0.291604  34.292 < 2.2e-16 ***
lsigma 1.070227   0.070717  15.134 < 2.2e-16 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

-2 log L: 497.8141
```

Question 0.5. Fit the above as shown. Then make a second fit, only this time call

```
> out.1 <- mle2(negLL, method="L-BFGS-B")
```

Is there a difference? This just uses a different method to do the optimization.

How do we do? We can ask if the paramters – known in this case – sit in the confidence intervals:

```
> confint(out)
```

```
           2.5 %    97.5 %
mu     9.4229469 10.577053
lsigma 0.9376736  1.215435
```

But the one is for the log of sigma. We exponentiate to get back to the original scale:

```
> exp(confint(out)[2, ])
```

```
   2.5 %   97.5 %
2.554033 3.371759
```

Some examples and questions

Lets look at a data set from the `emdbook` package on pine cones for a certain type of fir. This is time series data, but we will not consider that. Rather we look at the simple relationship between total number of cones `TOTCONES` and the size of the tree measured through the "diameter at breast height" `DBH`.

This is a favorite variable often accompanied with a side story of how Europe and America measure "breast" at a different height.

First, we look at the distribution of cones. To this we fit an exponential model using likelihood.

We first plot a histogram. Based on the graph we estimate the mean to be around 100, so the rate is $1/100$:

```
> library(emdbook)
> data(FirDBHFec)
> hist(FirDBHFec$TOTCONES, prob=TRUE)
> curve(dexp(x, rate=1/100), add=TRUE)
```

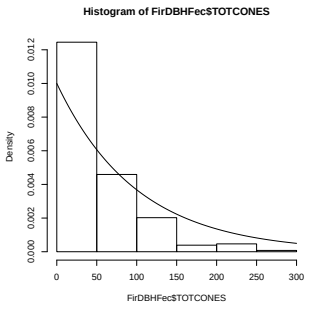

Figure 4: Histogram of total number of cones for all trees. The exponential curve was drawn with parameter found by first guessing the mean (around 100) then remembering that R uses the rate or the reciprocal of the mean.

To fit this with `mle2` we have:

```
> negLL <- function(lr=log(1/100))
+   -sum(dexp(x, rate=exp(lr), log=TRUE))
```

(We reparameterized the rate to avoid having a "boundary" at 0.) This data has missing values which will cause problems. As such, we reduce to the case where there are values.

```
> ind <- !is.na(FirDBHFec$TOTCONES)
> x <- FirDBHFec$TOTCONES[ind]
```

Now we can call `mle2`

```
> out <- mle2(negLL)
```

Our estimated mean is

```
> 1/exp(coef(out))

      lr
49.89576
```

This is a toy model, the likelihood function can be approached analytically and the answer for this distribution will always just be related to the sample mean.

Question 0.6. The number of total cones might also have a Poisson distribution, rather than an exponential. These are related, but one is discrete so is commonly used for such counts. Fit the total cone data using a Poisson distribution and compare rates. (The parameter `lambda` is the mean, not the reciprocal of the mean.)

Question 0.7. Make a plot of the negative log likelihood function you found above. Does it have a flat bottom or is it more "peaked?" Why is this question even of interest?

Question 0.8. Make a histogram of the data, then compare with the values of a confidence interval based on the Poisson fit. (You will need to take the exponential of the output from `confint`.)

We can use likelihood to do regression models. Imagine we assume a slight variation to the standard regression model, rather than assume normally distributed error terms, we use a double exponential. The density for this is similar to what we just used. The rate will depend on the mean which in turn depends in our model on the diameter at breast height. As such, a likelihood function for this is:

The bit `y-(a+b*x)` models the error term by an exponential. The double bit is why the `(1/2)` is there along with the `abs` function.

```
> negLL <- function(lr=log(1/50), a=.75, b=15)
+   -sum( (1/2)*dexp(abs(y - (a + b*x)), rate=exp(lr),
+                    log=TRUE))
```

Again we address the issue with `NA` values

```
> d <- subset(FirDBHFec, select=c("TOTCONES","DBH"))
> d <- d[complete.cases(d),]
> y <- d$TOTCONES; x <- d$DBH

> out <- mle2(negLL)
```

Now we compare our fit with the standard regression model:

```
> plot(TOTCONES ~ DBH, data=d)
> abline(lm(TOTCONES ~ DBH, data=d),
+        lty=2, lwd=2, col="red")
> abline(a=coef(out)['a'], b=coef(out)['b'],
+        lwd=2, col="blue")
```

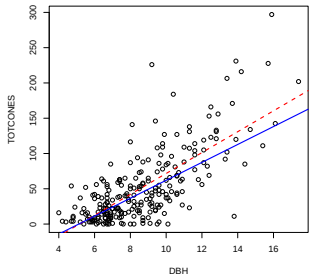

Figure 5: Regression fits. The dashed line with an assumption of normally distributed error terms, the solid line with double-exponential assumption.

Question 0.9. Does the regression model approach above give the same answer if normal error terms were assumed. Investigate. A possible function to consider is:

```
> negLL <- function(ls=log(20), a=0, b = 55)
+   -sum(dnorm(y - (a + b*x), sd=exp(ls), log=TRUE))
```

Question 0.10. We can't visualize this negative log likelihood with a contour plot, as it has 3 variables – not 2. The `profile` method computes profiles – as described in the text. These are used to give confidence intervals. To see a graphic, one has

```
> plot(profile(out))
```

where `out` holds the output of a call to `mle2`.

These graphs indicate confidence intervals for the parameters computed by accounting for changes to all the parameters (unlike slices). The `95%` should correspond to that from `confint`. Does it?