Stat 5421 Lecture Notes: Likelihood Computation

Charles J. Geyer

September 13, 2023

1 License

2 R

- The version of R used to make this document is 4.3.1.
- The version of the `rmarkdown` package used to make this document is 2.23.
- The version of the `knitr` package used to make this document is 1.43.
- The version of the `numDeriv` package used to make this document is 2016.8.1.1.
- The version of the `alabama` package used to make this document is 2022.4.1.

3 Statistical Model

As an example, we are going to use the ABO blood group system.

As you all know, you can have blood type A, B, AB, or O. And this governs what blood types you can be safely transfused with.

What this is about is red blood cell surface antigens, which are proteins that stick out from the surface of red blood cells and can be recognized by antibodies (antigens are things recognized by antibodies) causing an immune response if antigens different from one's own are introduced by blood transfusion (or any other way).

Genetically, these proteins are coded for by DNA and each person has two copies of these genes, one from Mom and one from Dad. And each such gene can be in one of three states (alleles): code for the A protein, code for the B protein, or code for no functional protein (that's the type O). Let the probability of the A, B, and O alleles be α, β, and γ, respectively.

We are now going to assume that the genes inherited from the parents are independent. This is called Hardy-Weinberg equilibrium in genetics. It is exactly true for random mating populations, and approximately true in populations that have had large populations sizes for many generations. Anyway, whether it holds for any particular population, we are going to assume it.

The Hardy-Weinberg assumption (statistical independence) says probabilities multiply. This gives us the following table.

	`A`	`B`	`O`
A	α^2	$\alpha\beta$	$\alpha\gamma$

B	$\beta\alpha$	β^2	$\beta\gamma$
O	$\gamma\alpha$	$\gamma\beta$	γ^2

The row labels are genes inherited from Mom, the column labels from Dad.

What can be observed in the lab are just the presence or absence of A or B or both. Thus the observable data has the following table

`type A`	`type B`	`type O`	`type AB`
$\alpha^2 + 2\alpha\gamma$	$\beta^2 + 2\beta\gamma$	γ^2	$2\alpha\beta$

If n_A, n_B, n_O, and n_{AB} are the observed counts of individuals of each blood type, then the log likelihood is

$$l(\alpha, \beta, \gamma) = n_A \log(\alpha^2 + 2\alpha\gamma) + n_B \log(\beta^2 + 2\beta\gamma) + n_O \log(\gamma^2) + n_{AB} \log(2\alpha\beta) \tag{1}$$

Since probabilities must add to one, we have $\alpha + \beta + \gamma = 1$, so there are only two freely variable parameters. We can choose which one to define in terms of the others.

4 Data

Some data found on web attributed to Clarke et al. (1959, *British Medical Journal*, 1, 603–607)

```
nA <- 186
nB <- 38
nAB <- 36
nO <- 284
```

5 Coding the Log Likelihood, Try I

We are going to write a function that evaluates minus the log likelihood because some R optimizers only minimize (not maximize). Minimizing minus the log likelihood is the same as maximizing the log likelihood.

We are going to start with the simplest possible R function that evaluates the log likelihood. This is not actually recommended, but is OK for one-time projects producing throw-away code (no one will ever use it again, even your future self). For the Right Thing (TRT), see the Section 9 below.

R optimizers expect the argument of the function to optimize to be a vector. So the argument of our function will be $\theta = (\alpha, \beta)$. Recall that $\gamma = 1 - \alpha - \beta$.

```
mlogl <- function(theta) {
    alpha <- theta[1]
    beta <- theta[2]
    gamma <- 1 - alpha - beta
    - (nA * log(alpha^2 + 2 * alpha * gamma) +
        nB * log(beta^2 + 2 * beta * gamma) +
        nO * log(gamma^2) +
        nAB * log(2 * alpha * beta))
}
```

6 Starting Values for Likelihood Maximization

Likelihood theory say we find the efficient likelihood estimator (ELE) if we start at a good starting place (so-called root-n-consistent estimates) and go uphill from there.

Method of moment estimators (that set expectations or probabilities equal to their observed values in the data) are always root-n-consistent.

The obvious expectations are those of the cell counts. These give four possible equations to solve for two unknowns.

$$\begin{aligned}
\frac{n_A}{n_A + n_B + n_O + n_{AB}} &= \alpha^2 + 2\alpha\gamma \\
\frac{n_B}{n_A + n_B + n_O + n_{AB}} &= \beta^2 + 2\beta\gamma \\
\frac{n_O}{n_A + n_B + n_O + n_{AB}} &= \gamma^2 \\
\frac{n_{AB}}{n_A + n_B + n_O + n_{AB}} &= 2\alpha\beta
\end{aligned}$$

We start by using the third of these to estimate γ.

```
gamma.twiddle <- sqrt(nO / (nA + nB + nO + nAB))
gamma.twiddle
```

```
## [1] 0.7225364
```

Now with γ estimated, we can use the first two equations to estimate α and β. Let the left-hand side of those equations be denoted p_A and p_B. Then we have the quadratic equation

$$\alpha^2 + 2\alpha\gamma - p_A = 0$$

which has solution

$$\begin{aligned}
\alpha &= \frac{-2\gamma \pm \sqrt{4\gamma^2 + 4p_A}}{2} \\
&= -\gamma \pm \sqrt{\gamma^2 + p_A}
\end{aligned}$$

Because $\sqrt{\gamma^2 + p_A}$ is greater than γ, choosing the $-$ in the $\pm$ gives a negative estimate of the probability, which makes no sense. Hence our estimates are

```
pA <- nA / (nA + nB + nO + nAB)
pB <- nB / (nA + nB + nO + nAB)
alpha.twiddle <- (- gamma.twiddle + sqrt(gamma.twiddle^2 + pA))
beta.twiddle <- (- gamma.twiddle + sqrt(gamma.twiddle^2 + pB))
alpha.twiddle
```

```
## [1] 0.2069638
```

```
beta.twiddle
```

```
## [1] 0.04682164
```

We check whether these sum to one

```
alpha.twiddle + beta.twiddle + gamma.twiddle
```

```
## [1] 0.9763218
```

They don't. There is nothing in the method of moments that forces estimators to be reasonable in this way. Of course, since these are consistent estimators, this sum would get closer and closer to one as sample size goes to infinity. But it does not have to be exactly one for any finite sample size.

Anyway, these are just starting values for optimization. They are good enough (so says the "usual" theory of maximum likelihood estimation).

```
theta.start <- c(alpha.twiddle, beta.twiddle)
```

We will later learn that in a regular full exponential family statistical model we do not need good starting points, any starting point will do. And we will also learn that the multinomial family of distributions is a regular full exponential family. But this model is not the full multinomial model, which has no restrictions on the cell probabilities. It is called a *curved exponential family* because it is not a regular full exponential family. And as such, it does not satisfy the conditions for the theorem that says any starting point will do. Here we need good starting points (like we did with the Cauchy examples in the other likelihood notes).

7 Maximum Likelihood Estimation

We will use R function `nlm` for the optimization.

```
nout <- nlm(mlogl, theta.start)
```

```
## Warning in log(alpha^2 + 2 * alpha * gamma): NaNs produced
## Warning in log(beta^2 + 2 * beta * gamma): NaNs produced
## Warning in nlm(mlogl, theta.start): NA/Inf replaced by maximum positive value
## Warning in log(alpha^2 + 2 * alpha * gamma): NaNs produced
## Warning in log(beta^2 + 2 * beta * gamma): NaNs produced
## Warning in nlm(mlogl, theta.start): NA/Inf replaced by maximum positive value
## Warning in log(alpha^2 + 2 * alpha * gamma): NaNs produced
## Warning in log(beta^2 + 2 * beta * gamma): NaNs produced
## Warning in nlm(mlogl, theta.start): NA/Inf replaced by maximum positive value
```

```
nout$code <= 2 # should be TRUE for solution
```

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

```
nout$estimate
```

```
## [1] 0.22790634 0.06966436
```

R function `nlm` claims it has found a solution even though there were warnings. The issue is that `nlm` does not know that the parameters have to have nonnegative values that sum to less than or equal to one. So it steps outside the parameter space and `NaN` values of our R function `mlogl` result. Eventually, it gets back into the parameter space and finds the solution. But we don't like the warnings, so we redo, starting at the putative solution.

```
nout <- nlm(mlogl, nout$estimate)
nout$code <= 2 # should be TRUE for solution
```

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

```
nout$estimate
```

```
## [1] 0.22790634 0.06966436
```

```
theta.start
```

```
## [1] 0.20696380 0.04682164
```

Now no warnings, and no change in the estimates. It appears the code worked the first time and we didn't need the redo.

We repeat showing the value of `theta.start` to show that the optimization did something. The maximum likelihood estimator (MLE, `nout$estimate`) is different from the method of moments estimator (MOME, `theta.start`). And we know from theory that the MLE are better estimators than any other (asymptotically, for large sample size). Hence, in particular, are better than MOME.

So here are the MLE and MOME

```
theta.hat <- c(nout$estimate, 1 - sum(nout$estimate))
names(theta.hat) <- c("alpha", "beta", "gamma")
theta.hat
```

```
##      alpha       beta      gamma
## 0.22790634 0.06966436 0.70242930
```

```
theta.twiddle <- c(alpha.twiddle, beta.twiddle, gamma.twiddle)
names(theta.twiddle) <- c("alpha", "beta", "gamma")
theta.twiddle
```

```
##      alpha       beta      gamma
## 0.20696380 0.04682164 0.72253638
```

Again, theory says the MLE is better than the MOME. We show both only to show that they are different.

8 Likelihood Inference

8.1 Wald Intervals

We need Fisher information. Here we will use observed Fisher information and let the computer do it for us. We will need to redo the optimization because we forgot to ask the computer to calculate second derivatives.

```
nout <- nlm(mlogl, nout$estimate, hessian = TRUE)
nout$code <= 2 # should be TRUE for solution
```

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

```
nout$estimate
```

```
## [1] 0.22790634 0.06966436
```

```
nout$hessian
```

```
##          [,1]      [,2]
## [1,] 5561.522  1326.034
## [2,] 1326.034 16652.382
```

`nout$hessian` is the observed Fisher information matrix. It is minus the second derivative matrix of the log likelihood because our function evaluates minus the log likelihood.

As an alternative, we could use R package `numDeriv`, which calculates second derivative (Hessian) matrices using a more accurate algorithm.

```
library(numDeriv)
hessian(mlogl, nout$estimate)
```

```
##          [,1]      [,2]
## [1,] 5564.915  1325.674
## [2,] 1325.674 16695.649
```

We'll use this one, but the other is OK too.

```
fisher <- hessian(mlogl, nout$estimate)
```

The asymptotic (large sample approximation) variance of the parameter vector estimate is inverse Fisher information. R function `solve` inverts matrices.

```
fisher.inverse <- solve(fisher)
fisher.inverse
```

```
##               [,1]          [,2]
## [1,]  1.831618e-04 -1.454348e-05
## [2,] -1.454348e-05  6.105063e-05
```

Thus 95% Wald confidence intervals for α and β are given by

```
conf.level <- 0.95
crit <- qnorm((1 + conf.level) / 2)
crit
```

```
## [1] 1.959964
```

```
theta.hat[1] + c(-1, 1) * crit * sqrt(fisher.inverse[1, 1])
```

```
## [1] 0.2013807 0.2544320
```

```
theta.hat[2] + c(-1, 1) * crit * sqrt(fisher.inverse[2, 2])
```

```
## [1] 0.05435020 0.08497852
```

To calculate a confidence interval for γ we use the fact, proved in theory classes and discussed in the notes that accompanied chapter 1 in Agresti, that if a is a nonrandom vector, B is a nonrandom matrix, and X is a random vector having dimensions such that $a + BX$ makes sense, then

$$\text{var}(a + BX) = B\text{var}(X)B^T$$

What we want here is the a and B that turn the estimate of (α, β) into the estimate of (α, β, γ). That is

$$a = \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}$$

$$B = \begin{pmatrix} 1 & 0 \\ 0 & 1 \\ -1 & -1 \end{pmatrix}$$

Let's check.

```
a <- c(0, 0, 1)
b <- matrix(c(1, 0, -1, 0, 1, -1), nrow = 3)
a
```

```
## [1] 0 0 1
```

```
b
```

```
##      [,1] [,2]
## [1,]    1    0
## [2,]    0    1
## [3,]   -1   -1
```

```
as.numeric(a + b %*% nout$estimate)
```

```
## [1] 0.22790634 0.06966436 0.70242930
```

```
theta.hat
```

```
##      alpha       beta      gamma
## 0.22790634 0.06966436 0.70242930
```

So

```
asymp.var.theta.three.dimensional <- b %*% fisher.inverse %*% t(b)
asymp.var.theta.three.dimensional
```

```
##                [,1]          [,2]          [,3]
## [1,]  1.831618e-04 -1.454348e-05 -1.686183e-04
## [2,] -1.454348e-05  6.105063e-05 -4.650715e-05
## [3,] -1.686183e-04 -4.650715e-05  2.151255e-04
```

So our 95% confidence interval for γ is

```
theta.hat[3] + c(-1, 1) * crit * sqrt(asymp.var.theta.three.dimensional[3, 3])
```

```
## [1] 0.6736822 0.7311764
```

8.2 Likelihood Intervals

Now our critical value is based on the chi-square distribution on one degree of freedom (one because we are doing confidence intervals for one parameter at a time and *are not trying for simultaneous coverage*).

```
crit <- qchisq(conf.level, df = 1)
crit
```

```
## [1] 3.841459
```

```
sqrt(crit)
```

```
## [1] 1.959964
```

To find the likelihood-based confidence interval we maximize and minimize the parameter of interest subject to the constraint that the log likelihood is no less than `crit` down from the maximum value.

```
library(alabama)
# alpha lower
confun <- function(theta) crit - 2 * (mlogl(theta) - nout$minimum)
aout <- auglag(nout$estimate, fn = function(theta) theta[1], hin = confun)
```

```
## Min(hin):  3.841459
## Outer iteration:  1
## Min(hin):  3.841459
## par:  0.227906 0.0696644
## fval =   0.2279
##
```

```
## Warning in log(alpha^2 + 2 * alpha * gamma): NaNs produced
```

```
## Warning in log(2 * alpha * beta): NaNs produced
```

```
## Warning in log(alpha^2 + 2 * alpha * gamma): NaNs produced
```

```
## Warning in log(2 * alpha * beta): NaNs produced
```

```
## Warning in log(alpha^2 + 2 * alpha * gamma): NaNs produced
```

```
## Warning in log(2 * alpha * beta): NaNs produced
```

```
## Warning in log(alpha^2 + 2 * alpha * gamma): NaNs produced
```

```
## Warning in log(2 * alpha * beta): NaNs produced
## Warning in log(alpha^2 + 2 * alpha * gamma): NaNs produced
## Warning in log(2 * alpha * beta): NaNs produced
## Warning in log(alpha^2 + 2 * alpha * gamma): NaNs produced
## Warning in log(2 * alpha * beta): NaNs produced
## Warning in log(alpha^2 + 2 * alpha * gamma): NaNs produced
## Warning in log(2 * alpha * beta): NaNs produced
## Warning in log(alpha^2 + 2 * alpha * gamma): NaNs produced
## Warning in log(2 * alpha * beta): NaNs produced
## Warning in log(alpha^2 + 2 * alpha * gamma): NaNs produced
## Warning in log(2 * alpha * beta): NaNs produced
## Warning in log(alpha^2 + 2 * alpha * gamma): NaNs produced
## Warning in log(2 * alpha * beta): NaNs produced
## Warning in log(alpha^2 + 2 * alpha * gamma): NaNs produced
## Warning in log(2 * alpha * beta): NaNs produced
## Warning in log(alpha^2 + 2 * alpha * gamma): NaNs produced
## Warning in log(2 * alpha * beta): NaNs produced
## Warning in log(alpha^2 + 2 * alpha * gamma): NaNs produced
## Warning in log(2 * alpha * beta): NaNs produced
## Warning in log(alpha^2 + 2 * alpha * gamma): NaNs produced
## Warning in log(2 * alpha * beta): NaNs produced
## Warning in log(alpha^2 + 2 * alpha * gamma): NaNs produced
## Warning in log(2 * alpha * beta): NaNs produced
## Warning in log(alpha^2 + 2 * alpha * gamma): NaNs produced
## Warning in log(2 * alpha * beta): NaNs produced
## Warning in log(alpha^2 + 2 * alpha * gamma): NaNs produced
## Warning in log(2 * alpha * beta): NaNs produced
## Warning in log(alpha^2 + 2 * alpha * gamma): NaNs produced
## Warning in log(2 * alpha * beta): NaNs produced
## Warning in log(alpha^2 + 2 * alpha * gamma): NaNs produced
## Warning in log(2 * alpha * beta): NaNs produced
## Warning in log(alpha^2 + 2 * alpha * gamma): NaNs produced
## Warning in log(2 * alpha * beta): NaNs produced
## Warning in log(alpha^2 + 2 * alpha * gamma): NaNs produced
## Warning in log(2 * alpha * beta): NaNs produced
## Warning in log(beta^2 + 2 * beta * gamma): NaNs produced
```

```
## Warning in log(2 * alpha * beta): NaNs produced
## Warning in log(beta^2 + 2 * beta * gamma): NaNs produced
## Warning in log(2 * alpha * beta): NaNs produced
## Warning in log(beta^2 + 2 * beta * gamma): NaNs produced
## Warning in log(2 * alpha * beta): NaNs produced
## Warning in log(beta^2 + 2 * beta * gamma): NaNs produced
## Warning in log(2 * alpha * beta): NaNs produced
## Warning in log(beta^2 + 2 * beta * gamma): NaNs produced
## Warning in log(2 * alpha * beta): NaNs produced
## Warning in log(beta^2 + 2 * beta * gamma): NaNs produced
## Warning in log(2 * alpha * beta): NaNs produced
## Warning in log(beta^2 + 2 * beta * gamma): NaNs produced
## Warning in log(2 * alpha * beta): NaNs produced
## Warning in log(beta^2 + 2 * beta * gamma): NaNs produced
## Warning in log(2 * alpha * beta): NaNs produced
## Warning in log(beta^2 + 2 * beta * gamma): NaNs produced
## Warning in log(2 * alpha * beta): NaNs produced
## Warning in log(alpha^2 + 2 * alpha * gamma): NaNs produced
## Warning in log(beta^2 + 2 * beta * gamma): NaNs produced
## Warning in log(alpha^2 + 2 * alpha * gamma): NaNs produced
## Warning in log(beta^2 + 2 * beta * gamma): NaNs produced

## Warning in log(beta^2 + 2 * beta * gamma): NaNs produced
## Warning in log(alpha^2 + 2 * alpha * gamma): NaNs produced
## Warning in log(2 * alpha * beta): NaNs produced
## Warning in log(alpha^2 + 2 * alpha * gamma): NaNs produced
## Warning in log(2 * alpha * beta): NaNs produced
## Warning in log(alpha^2 + 2 * alpha * gamma): NaNs produced
## Warning in log(2 * alpha * beta): NaNs produced
## Warning in log(alpha^2 + 2 * alpha * gamma): NaNs produced
## Warning in log(2 * alpha * beta): NaNs produced
## Warning in log(alpha^2 + 2 * alpha * gamma): NaNs produced
## Warning in log(2 * alpha * beta): NaNs produced
## Warning in log(alpha^2 + 2 * alpha * gamma): NaNs produced
## Warning in log(2 * alpha * beta): NaNs produced
## Warning in log(beta^2 + 2 * beta * gamma): NaNs produced
## Warning in log(2 * alpha * beta): NaNs produced
```

```
## Warning in log(beta^2 + 2 * beta * gamma): NaNs produced
## Warning in log(2 * alpha * beta): NaNs produced
## Warning in log(beta^2 + 2 * beta * gamma): NaNs produced
## Warning in log(2 * alpha * beta): NaNs produced
## Warning in log(beta^2 + 2 * beta * gamma): NaNs produced
## Warning in log(2 * alpha * beta): NaNs produced
## Warning in log(beta^2 + 2 * beta * gamma): NaNs produced
## Warning in log(2 * alpha * beta): NaNs produced
## Warning in log(beta^2 + 2 * beta * gamma): NaNs produced
## Warning in log(2 * alpha * beta): NaNs produced
## Warning in log(beta^2 + 2 * beta * gamma): NaNs produced
## Warning in log(2 * alpha * beta): NaNs produced
## Warning in log(beta^2 + 2 * beta * gamma): NaNs produced
## Warning in log(2 * alpha * beta): NaNs produced
## Warning in log(beta^2 + 2 * beta * gamma): NaNs produced
## Warning in log(2 * alpha * beta): NaNs produced
## Warning in log(alpha^2 + 2 * alpha * gamma): NaNs produced
## Warning in log(beta^2 + 2 * beta * gamma): NaNs produced
## Warning in log(alpha^2 + 2 * alpha * gamma): NaNs produced
## Warning in log(beta^2 + 2 * beta * gamma): NaNs produced
## Warning in log(alpha^2 + 2 * alpha * gamma): NaNs produced
## Warning in log(2 * alpha * beta): NaNs produced
## Warning in log(alpha^2 + 2 * alpha * gamma): NaNs produced
## Warning in log(2 * alpha * beta): NaNs produced
## Warning in log(alpha^2 + 2 * alpha * gamma): NaNs produced
## Warning in log(2 * alpha * beta): NaNs produced
## Warning in log(alpha^2 + 2 * alpha * gamma): NaNs produced
## Warning in log(2 * alpha * beta): NaNs produced
## Warning in log(alpha^2 + 2 * alpha * gamma): NaNs produced
## Warning in log(2 * alpha * beta): NaNs produced
## Warning in log(alpha^2 + 2 * alpha * gamma): NaNs produced
## Warning in log(2 * alpha * beta): NaNs produced
## Warning in log(beta^2 + 2 * beta * gamma): NaNs produced
## Warning in log(2 * alpha * beta): NaNs produced
## Warning in log(beta^2 + 2 * beta * gamma): NaNs produced
## Warning in log(2 * alpha * beta): NaNs produced
```

```
## Warning in log(beta^2 + 2 * beta * gamma): NaNs produced
## Warning in log(2 * alpha * beta): NaNs produced
## Warning in log(beta^2 + 2 * beta * gamma): NaNs produced
## Warning in log(2 * alpha * beta): NaNs produced
## Warning in log(beta^2 + 2 * beta * gamma): NaNs produced
## Warning in log(2 * alpha * beta): NaNs produced
## Warning in log(beta^2 + 2 * beta * gamma): NaNs produced
## Warning in log(2 * alpha * beta): NaNs produced
## Warning in log(beta^2 + 2 * beta * gamma): NaNs produced
## Warning in log(2 * alpha * beta): NaNs produced
## Warning in log(beta^2 + 2 * beta * gamma): NaNs produced
## Warning in log(2 * alpha * beta): NaNs produced
## Warning in log(beta^2 + 2 * beta * gamma): NaNs produced
## Warning in log(2 * alpha * beta): NaNs produced
## Warning in log(beta^2 + 2 * beta * gamma): NaNs produced
## Warning in log(2 * alpha * beta): NaNs produced
## Warning in log(alpha^2 + 2 * alpha * gamma): NaNs produced

## Warning in log(alpha^2 + 2 * alpha * gamma): NaNs produced
## Warning in log(beta^2 + 2 * beta * gamma): NaNs produced
## Warning in log(alpha^2 + 2 * alpha * gamma): NaNs produced
## Warning in log(beta^2 + 2 * beta * gamma): NaNs produced

## Warning in log(beta^2 + 2 * beta * gamma): NaNs produced
## Warning in log(alpha^2 + 2 * alpha * gamma): NaNs produced
## Warning in log(2 * alpha * beta): NaNs produced
## Warning in log(alpha^2 + 2 * alpha * gamma): NaNs produced
## Warning in log(2 * alpha * beta): NaNs produced
## Warning in log(alpha^2 + 2 * alpha * gamma): NaNs produced
## Warning in log(2 * alpha * beta): NaNs produced
## Warning in log(beta^2 + 2 * beta * gamma): NaNs produced
## Warning in log(2 * alpha * beta): NaNs produced
## Warning in log(beta^2 + 2 * beta * gamma): NaNs produced
## Warning in log(2 * alpha * beta): NaNs produced
## Warning in log(beta^2 + 2 * beta * gamma): NaNs produced
## Warning in log(2 * alpha * beta): NaNs produced
## Warning in log(beta^2 + 2 * beta * gamma): NaNs produced
```

```
## Warning in log(2 * alpha * beta): NaNs produced
## Warning in log(beta^2 + 2 * beta * gamma): NaNs produced
## Warning in log(2 * alpha * beta): NaNs produced
## Warning in log(beta^2 + 2 * beta * gamma): NaNs produced
## Warning in log(2 * alpha * beta): NaNs produced
## Warning in log(beta^2 + 2 * beta * gamma): NaNs produced
## Warning in log(2 * alpha * beta): NaNs produced
## Warning in log(beta^2 + 2 * beta * gamma): NaNs produced
## Warning in log(2 * alpha * beta): NaNs produced
## Warning in log(beta^2 + 2 * beta * gamma): NaNs produced
## Warning in log(2 * alpha * beta): NaNs produced
## Warning in log(beta^2 + 2 * beta * gamma): NaNs produced
## Warning in log(2 * alpha * beta): NaNs produced
## Warning in log(alpha^2 + 2 * alpha * gamma): NaNs produced
## Warning in log(2 * alpha * beta): NaNs produced
## Warning in log(beta^2 + 2 * beta * gamma): NaNs produced
## Warning in log(2 * alpha * beta): NaNs produced
## Warning in log(beta^2 + 2 * beta * gamma): NaNs produced
## Warning in log(2 * alpha * beta): NaNs produced
## Warning in log(beta^2 + 2 * beta * gamma): NaNs produced
## Warning in log(2 * alpha * beta): NaNs produced
## Warning in log(beta^2 + 2 * beta * gamma): NaNs produced
## Warning in log(2 * alpha * beta): NaNs produced
## Warning in log(beta^2 + 2 * beta * gamma): NaNs produced
## Warning in log(2 * alpha * beta): NaNs produced
## Warning in log(beta^2 + 2 * beta * gamma): NaNs produced
## Warning in log(2 * alpha * beta): NaNs produced
## Warning in log(alpha^2 + 2 * alpha * gamma): NaNs produced
```

```
## Warning in log(alpha^2 + 2 * alpha * gamma): NaNs produced
## Warning in log(beta^2 + 2 * beta * gamma): NaNs produced
## Warning in log(alpha^2 + 2 * alpha * gamma): NaNs produced
## Warning in log(beta^2 + 2 * beta * gamma): NaNs produced
## Warning in log(alpha^2 + 2 * alpha * gamma): NaNs produced
## Warning in log(2 * alpha * beta): NaNs produced
## Warning in log(alpha^2 + 2 * alpha * gamma): NaNs produced
## Warning in log(2 * alpha * beta): NaNs produced
```

```
## Outer iteration:  2
## Min(hin):  0.009993892
## par:  0.202212 0.0708439
## fval =   0.2022
##
## Warning in log(alpha^2 + 2 * alpha * gamma): NaNs produced

## Warning in log(alpha^2 + 2 * alpha * gamma): NaNs produced

## Warning in log(alpha^2 + 2 * alpha * gamma): NaNs produced

## Warning in log(2 * alpha * beta): NaNs produced

## Warning in log(alpha^2 + 2 * alpha * gamma): NaNs produced

## Warning in log(2 * alpha * beta): NaNs produced

## Warning in log(alpha^2 + 2 * alpha * gamma): NaNs produced

## Warning in log(2 * alpha * beta): NaNs produced

## Warning in log(alpha^2 + 2 * alpha * gamma): NaNs produced

## Warning in log(2 * alpha * beta): NaNs produced

## Warning in log(alpha^2 + 2 * alpha * gamma): NaNs produced

## Warning in log(2 * alpha * beta): NaNs produced

## Warning in log(alpha^2 + 2 * alpha * gamma): NaNs produced

## Warning in log(2 * alpha * beta): NaNs produced

## Warning in log(alpha^2 + 2 * alpha * gamma): NaNs produced

## Warning in log(2 * alpha * beta): NaNs produced

## Warning in log(alpha^2 + 2 * alpha * gamma): NaNs produced

## Warning in log(2 * alpha * beta): NaNs produced

## Warning in log(alpha^2 + 2 * alpha * gamma): NaNs produced

## Warning in log(2 * alpha * beta): NaNs produced

## Warning in log(beta^2 + 2 * beta * gamma): NaNs produced

## Warning in log(2 * alpha * beta): NaNs produced

## Warning in log(beta^2 + 2 * beta * gamma): NaNs produced

## Warning in log(2 * alpha * beta): NaNs produced

## Warning in log(beta^2 + 2 * beta * gamma): NaNs produced

## Warning in log(2 * alpha * beta): NaNs produced

## Warning in log(beta^2 + 2 * beta * gamma): NaNs produced

## Warning in log(2 * alpha * beta): NaNs produced

## Warning in log(beta^2 + 2 * beta * gamma): NaNs produced

## Warning in log(2 * alpha * beta): NaNs produced

## Warning in log(beta^2 + 2 * beta * gamma): NaNs produced

## Warning in log(2 * alpha * beta): NaNs produced
```

```
## Warning in log(beta^2 + 2 * beta * gamma): NaNs produced
## Warning in log(2 * alpha * beta): NaNs produced
## Warning in log(beta^2 + 2 * beta * gamma): NaNs produced
## Warning in log(2 * alpha * beta): NaNs produced
## Warning in log(alpha^2 + 2 * alpha * gamma): NaNs produced
## Warning in log(beta^2 + 2 * beta * gamma): NaNs produced
## Warning in log(alpha^2 + 2 * alpha * gamma): NaNs produced
## Warning in log(beta^2 + 2 * beta * gamma): NaNs produced

## Warning in log(beta^2 + 2 * beta * gamma): NaNs produced
## Warning in log(alpha^2 + 2 * alpha * gamma): NaNs produced
## Warning in log(2 * alpha * beta): NaNs produced
## Warning in log(alpha^2 + 2 * alpha * gamma): NaNs produced
## Warning in log(2 * alpha * beta): NaNs produced
## Warning in log(beta^2 + 2 * beta * gamma): NaNs produced
## Warning in log(2 * alpha * beta): NaNs produced
## Warning in log(beta^2 + 2 * beta * gamma): NaNs produced
## Warning in log(2 * alpha * beta): NaNs produced
## Warning in log(beta^2 + 2 * beta * gamma): NaNs produced
## Warning in log(2 * alpha * beta): NaNs produced
## Warning in log(alpha^2 + 2 * alpha * gamma): NaNs produced
## Warning in log(beta^2 + 2 * beta * gamma): NaNs produced
## Warning in log(alpha^2 + 2 * alpha * gamma): NaNs produced
## Warning in log(2 * alpha * beta): NaNs produced
## Warning in log(alpha^2 + 2 * alpha * gamma): NaNs produced
## Warning in log(2 * alpha * beta): NaNs produced
## Warning in log(alpha^2 + 2 * alpha * gamma): NaNs produced
## Warning in log(2 * alpha * beta): NaNs produced
## Warning in log(alpha^2 + 2 * alpha * gamma): NaNs produced
## Warning in log(2 * alpha * beta): NaNs produced
## Warning in log(alpha^2 + 2 * alpha * gamma): NaNs produced
## Warning in log(2 * alpha * beta): NaNs produced
## Warning in log(alpha^2 + 2 * alpha * gamma): NaNs produced
## Warning in log(2 * alpha * beta): NaNs produced
## Warning in log(alpha^2 + 2 * alpha * gamma): NaNs produced
## Warning in log(2 * alpha * beta): NaNs produced
## Warning in log(alpha^2 + 2 * alpha * gamma): NaNs produced
```

```
## Warning in log(2 * alpha * beta): NaNs produced
## Warning in log(alpha^2 + 2 * alpha * gamma): NaNs produced
## Warning in log(2 * alpha * beta): NaNs produced
## Warning in log(alpha^2 + 2 * alpha * gamma): NaNs produced
## Warning in log(2 * alpha * beta): NaNs produced
## Warning in log(beta^2 + 2 * beta * gamma): NaNs produced
## Warning in log(2 * alpha * beta): NaNs produced
## Warning in log(alpha^2 + 2 * alpha * gamma): NaNs produced
## Warning in log(beta^2 + 2 * beta * gamma): NaNs produced
## Warning in log(alpha^2 + 2 * alpha * gamma): NaNs produced
## Warning in log(beta^2 + 2 * beta * gamma): NaNs produced
## Warning in log(alpha^2 + 2 * alpha * gamma): NaNs produced
## Warning in log(beta^2 + 2 * beta * gamma): NaNs produced
## Warning in log(alpha^2 + 2 * alpha * gamma): NaNs produced
## Warning in log(2 * alpha * beta): NaNs produced
## Warning in log(alpha^2 + 2 * alpha * gamma): NaNs produced
## Warning in log(2 * alpha * beta): NaNs produced
## Warning in log(alpha^2 + 2 * alpha * gamma): NaNs produced
## Warning in log(2 * alpha * beta): NaNs produced
## Outer iteration:  3
## Min(hin):  1.722997e-06
## par:  0.202142 0.0716726
## fval =   0.2021
##
## Outer iteration:  4
## Min(hin):  5.701391e-07
## par:  0.202142 0.0716726
## fval =   0.2021
##
## Warning in log(alpha^2 + 2 * alpha * gamma): NaNs produced
## Warning in log(alpha^2 + 2 * alpha * gamma): NaNs produced
## Outer iteration:  5
## Min(hin):  3.181802e-07
## par:  0.202142 0.0716726
## fval =   0.2021
##
## Warning in log(alpha^2 + 2 * alpha * gamma): NaNs produced
## Warning in log(alpha^2 + 2 * alpha * gamma): NaNs produced
aout$convergence == 0
## [1] TRUE
```

```
aout$value
```

```
## [1] 0.2021416
```

All of that blather was most annoying. Tell it to STFU.

```
# alpha lower
aout <- suppressWarnings(auglag(nout$estimate, fn = function(theta) theta[1],
    hin = confun))
```

```
## Min(hin):  3.841459
## Outer iteration:  1
## Min(hin):  3.841459
## par:  0.227906 0.0696644
## fval =   0.2279
##
## Outer iteration:  2
## Min(hin):  0.009993892
## par:  0.202212 0.0708439
## fval =   0.2022
##
## Outer iteration:  3
## Min(hin):  1.722997e-06
## par:  0.202142 0.0716726
## fval =   0.2021
##
## Outer iteration:  4
## Min(hin):  5.701391e-07
## par:  0.202142 0.0716726
## fval =   0.2021
##
## Outer iteration:  5
## Min(hin):  3.181802e-07
## par:  0.202142 0.0716726
## fval =   0.2021
##
```

```
aout$convergence == 0
```

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

```
aout$value
```

```
## [1] 0.2021416
```

Need to tell it STFU more forcefully.

```
# alpha lower
aout <- suppressWarnings(auglag(nout$estimate, fn = function(theta) theta[1],
    hin = confun, control.outer = list(trace = FALSE)))
aout$convergence == 0
```

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

```
aout$value
```

```
## [1] 0.2021416
```

Great! Now for the rest.

```
# alpha lower
alpha.lower <- aout$value
# alpha upper
aout <- suppressWarnings(auglag(nout$estimate, fn = function(theta) theta[1],
    hin = confun, control.outer = list(trace = FALSE),
    control.optim = list(fnscale = -1)))
aout$convergence == 0
```

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

```
aout$value
```

```
## [1] 0.2551525
```

```
alpha.upper <- aout$value
```

The `control.optim = list(fnscale = -1)` is what tells R function `auglag` to maximize rather than minimize. This was particularly hard to find in the documentation because it is on the help page for R function `optim`, which is referred to from the help page for R function `auglag`.

And so forth.

```
# beta lower
aout <- suppressWarnings(auglag(nout$estimate, fn = function(theta) theta[2],
    hin = confun, control.outer = list(trace = FALSE)))
aout$convergence == 0
```

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

```
aout$value
```

```
## [1] 0.05538393
```

```
beta.lower <- aout$value
# beta upper
aout <- suppressWarnings(auglag(nout$estimate, fn = function(theta) theta[2],
    hin = confun, control.outer = list(trace = FALSE),
    control.optim = list(fnscale = -1)))
aout$convergence == 0
```

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

```
aout$value
```

```
## [1] 0.08602016
```

```
beta.upper <- aout$value
# gamma lower
aout <- suppressWarnings(auglag(nout$estimate,
    fn = function(theta) 1 - sum(theta),
    hin = confun, control.outer = list(trace = FALSE)))
aout$convergence == 0
```

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

```
aout$value
```

```
## [1] 0.6733144
```

```
gamma.lower <- aout$value
# gamma upper
```

```
aout <- suppressWarnings(auglag(nout$estimate,
    fn = function(theta) 1 - sum(theta),
    hin = confun, control.outer = list(trace = FALSE),
    control.optim = list(fnscale = -1)))
aout$convergence == 0
```

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

```
aout$value
```

```
## [1] 0.7266693
```

```
gamma.upper <- aout$value
```

8.3 Rao (Score) Intervals

These turned out to be just too horrible to do. If we try to derive the Rao pivotal quantity analytically, the calculus is horribly messy. If we try to let the computer do the calculus using R package `numDeriv` then that doesn't work because R function `auglag` in R package `alabama` needs derivatives of the nonlinear constraint function that it tries to calculate by finite-difference approximation. So that means we either need everything done analytically or calculating derivatives of functions containing derivatives by finite-difference approximations of functions themselves defined by finite-difference approximations. And the latter does not work. Computer arithmetic is too sloppy.

However much one may like Rao intervals for simple problems, they are just not worth the effort in complicated problems. And this problem is already too complicated for them, even though it only involves a contingency table with four cells and a curved exponential family with two parameters.

8.4 Summary

8.4.1 Wald Intervals

	lower	upper
alpha	0.2014	0.2544
beta	0.0544	0.0850
gamma	0.6737	0.7312

8.4.2 Likelihood Intervals

	lower	upper
alpha	0.2021	0.2552
beta	0.0554	0.0860
gamma	0.6733	0.7267

These intervals are **not** corrected for simultaneous coverage. See Section 10 below for that.

9 Fixing Up Our R Function

9.1 Error Messages

The first improvement to make to any function is to make it follow the policy of "garbage in, error messages out". And we want the error messages to be understandable and informative. So

```
mlogl <- function(theta) {
    stopifnot(is.numeric(theta))
    stopifnot(is.finite(theta))
    stopifnot(length(theta) == 2)
    alpha <- theta[1]
    beta <- theta[2]
    gamma <- 1 - alpha - beta
    - (nA * log(alpha^2 + 2 * alpha * gamma) +
        nB * log(beta^2 + 2 * beta * gamma) +
        nO * log(gamma^2) +
        nAB * log(2 * alpha * beta))
}
```

We should check everything we reasonably can about our arguments. One might think we should also add

```
    stopifnot(theta >= 0)
    stopifnot(sum(theta) <= 1)
```

But, as we saw, R function `nlm` will call the function supplied to it (which is our function `mlogl` in this handout) when the parameters are outside the sample space (because there is no way to indicate to it what the sample space is). So we do not check these, but rather just suppress the warnings, as shown above, if we don't want to see them.

We had a similar problem with R function `auglag` because it also has the constraint function (argument `hin`) that calls our function `mlogl` and it also does not know what the allowed parameter values are.

9.2 Using a Factory Function

Global variables are evil, as every course on computer programming says. If the project is one-off and not to be (ever, under any circumstances) used as an example (even for one's own future work), then there is nothing wrong with global variables. But code tends to be re-used, even if the re-use is not planned. So avoid global variables.

To do that we don't rewrite our function `mlogl`. Instead we write an R function to write it. A function that makes functions is called a *function factory*. So that is what we are using.

R functions remember the environment in which they were created. That is how our current version of the function finds the variables `nA`, `nB`, `nO`, and `nAB`. It was created in the R global environment, and that is where those variables are.

If one clobbers or removes those variables, our function will not work. When we use a function factory and put those variables in the execution environment of the function factory, then they are where users do not think to mess with them. Most users don't even know functions have environments.

So here is how that looks.

```
mlogl.factory <- function(nA, nB, nO, nAB) {
    stopifnot(is.numeric(nA))
    stopifnot(is.finite(nA))
    stopifnot(round(nA) == nA) # checks integer value
    stopifnot(nA >= 0)
    stopifnot(is.numeric(nB))
    stopifnot(is.finite(nB))
    stopifnot(round(nB) == nB) # checks integer value
    stopifnot(nB >= 0)
    stopifnot(is.numeric(nO))
    stopifnot(is.finite(nO))
    stopifnot(round(nO) == nO) # checks integer value
```

```
    stopifnot(nO >= 0)
    stopifnot(is.numeric(nAB))
    stopifnot(is.finite(nAB))
    stopifnot(round(nAB) == nAB) # checks integer value
    stopifnot(nAB >= 0)
    function(theta) {
        stopifnot(is.numeric(theta))
        stopifnot(is.finite(theta))
        stopifnot(length(theta) == 2)
        alpha <- theta[1]
        beta <- theta[2]
        gamma <- 1 - alpha - beta
        - (nA * log(alpha^2 + 2 * alpha * gamma) +
            nB * log(beta^2 + 2 * beta * gamma) +
            nO * log(gamma^2) +
            nAB * log(2 * alpha * beta))
    }
}
```

An R function returns the value of the last expression if there is no explicit call of R function `return`. The last expression of `mlogl.factory` is the definition of a function that is identical to our function `mlogl` copied from the code above.

Now we use the factory to make a new `mlogl`.

```
mlogl <- mlogl.factory(nA, nB, nO, nAB)
```

And now we don't need the global variables.

```
rm(nA, nB, nO, nAB)
```

We check that our new function `mlogl` works just like the old one.

```
nout.estimate.save <- nout$estimate
nout <- suppressWarnings(nlm(mlogl, theta.start))
nout$code <= 2 # should be TRUE for solution
```

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

```
all.equal(nout$estimate, nout.estimate.save)
```

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

```
# have to redefine because have clobbered above
# global variables are evil
crit <- qchisq(conf.level, df = 1)
aout <- suppressWarnings(auglag(nout$estimate, fn = function(theta) theta[1],
    hin = confun, control.outer = list(trace = FALSE)))
aout$convergence == 0
```

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

```
all.equal(aout$value, alpha.lower)
```

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

How does this work? If we removed the data, where is it?

```
ls(envir = environment(mlogl))
```

```
## [1] "nA"  "nAB" "nB"  "nO"
```

10 Simultaneous Coverage

10.1 Confidence Regions

A confidence interval is a random interval that covers the true unknown (scalar) parameter value with a specified probability. A confidence *region* is a random *region* that covers the true unknown (vector) parameter value with a specified probability.

Confidence regions are not much used, not even mentioned in intro stats books. We are primarily interested in them as a tool for constructing simultaneous confidence intervals. But we develop the general theory here.

Like confidence intervals, confidence regions can be constructed by inverting hypothesis tests. The confidence region is the set of θ_0 that are not rejected by the hypothesis test with hypotheses

$$H_0 : \theta = \theta_0$$
$$H_1 : \theta \neq \theta_0$$

Let c be the appropriate chi-square critical value (the degrees of freedom is the dimension of the parameter space). Then inverting the likelihood ratio test gives the confidence region

$$\{ \, \theta : 2[l(\hat{\theta}) - l(\theta)] \leq c \, \}$$

where $\hat{\theta}$ is the MLE for the alternative hypothesis (the unrestricted MLE).

In the handout titled "Likelihood Inference" the formula for the Wald test statistic using observed Fisher information is

$$W_n' = g(\hat{\theta}_n)^T [\nabla g(\hat{\theta}_n) J_n(\hat{\theta}_n)^{-1} (\nabla g(\hat{\theta}_n))^T]^{-1} g(\hat{\theta}_n)$$

Here we use the special case where $g(\theta) = \theta - \theta_0$, the first derivative of which is the identity matrix. Then we have

$$W_n' = (\hat{\theta}_n - \theta)^T J_n(\hat{\theta}_n)(\hat{\theta}_n - \theta)$$

Then the Wald confidence region is

$$\{ \, \theta : (\hat{\theta}_n - \theta)^T J_n(\hat{\theta}_n)(\hat{\theta}_n - \theta) < c \, \}$$

We have to solve a multivariate quadratic equation to find the boundary of the region.

10.2 Simultaneous Confidence Intervals from Confidence Regions

If R is a confidence region and g is any scalar-valued function whatsoever, then

$$\inf_{\theta \in R} g(\theta) < g(\theta) < \sup_{\theta \in R} g(\theta)$$

is a confidence interval for the parameter $g(\theta)$, and, moreover, the coverage probability is the same as the coverage probability for R, and this is simultaneous for all possible functions g. This is how we are going to derive simultaneous confidence intervals here.

10.3 Likelihood Intervals

To fix up our likelihood intervals for simultaneous coverage, all we need is to use a critical value for two degrees of freedom. Everything else is the same. Why two? Because there are two independently adjustable parameters. The degrees of freedom for simultaneous coverage is always the number of independently adjustable parameters.

We won't show the work just the result. First do

```
crit
```

```
## [1] 3.841459
```

```
crit <- qchisq(conf.level, df = 2)
crit
```

```
## [1] 5.991465
```

and then redo the likelihood intervals shown above (work not shown, but is in the source (Rmd) file).

	lower	upper
alpha	0.1960	0.2621
beta	0.0521	0.0904
gamma	0.6698	0.7330

Comparing with Section 8.3.2 above we see the intervals aren't actually that much wider.

10.4 Wald Intervals

The same trick works for Wald intervals. This is similar to what is called Scheffé's method for linear models, but the same idea works in general. We just use the same critical value, as for the likelihood intervals, but when we use it for Wald intervals, we need to take a square root. We have been using all along the critical value for Wald intervals is the square root of the critical value for the confidence intervals. We just didn't note that explicitly. But a chi-square random variable with one degree of freedom is the square of a standard normal random variable. Now a chi-square random variable on k degrees of freedom is the sum of squares of k independent standard normal random variables.

So we do

```
crit <- sqrt(crit)
crit
```

```
## [1] 2.447747
```

and redo all of our Wald interval calculations (again work not shown but in the Rmd file).

	lower	upper
alpha	0.1948	0.2610
beta	0.0505	0.0888
gamma	0.6665	0.7383

Comparing with Section 8.3.1 above we see the intervals aren't actually that much wider.