Maximum Likelihood Estimation With Java and Ruby

Peter von Tessin

November 6, 2005

1 Introduction

Regression Analysis is one of the fundamentals of any data-analysis project. The first thing to do, given the assumption that x might be influencing y is to test this theory using the data. The assumed relationship might be just linear, some more complex time dependencies might be involved or the researcher actually expects a non-linear relationship to exist between x and y. In any case, the assumption of any kind of causal relationship needs to start with a hard look at the data. In this short paper we are present a simple maximum likelihood estimation via Newton-Raphson of a multivariate normal and multivariate logit model using Java and Ruby. The estimation procedure is implemented in both languages to highlight some of the similarities and difference using those two languages in data-analysis projects.

2 Multinomial Normal

The multinomial normal is one of the simplest regression models. It assumes that the random variable y is the linear combination of a number of independent, non-random variables X (combined into a matrix) and a normally distributed vector of random variables u:

$$y = X\beta + uu\tilde{N}(0, \sigma^2) \tag{1}$$

Obviously the classic way of estimating the parameters β and $\sigma^2$is to use least squares, which leads to the following estimator:

$$\hat{\beta} = (X'X)^{-1}X'y \tag{2}$$

2.1 Maximum Likelihood Function for the Multinomial Normal Model

But here we want to use a maximum likelihood estimation of the parameters. Therfore we will start with the Log-Likelihood function for the multinomial normal distribution:

$$\ln L(\beta, \sigma^2|y, X) = -\frac{n}{2}\ln 2\Pi - \frac{n}{2}\ln\sigma^2 - \frac{1}{2\sigma^2}(y - X\beta)'(y - X\beta) \quad (3)$$

The first derivative with respect to β is:

$$\frac{\delta \ln L}{\delta \beta} = \frac{1}{\sigma^2}X'(y - X\beta) \quad (4)$$

and the first derivative with respect to σ^2 is:

$$\frac{\delta \ln L}{\delta \sigma^2} = -\frac{n}{2\sigma^2} + \frac{1}{2^4}(y - X\beta)'(y - X\beta) \quad (5)$$

The matrix of second derivatives, the Hessian (H)is the following:

$$\mathbf{H} = \begin{pmatrix} -(1/\sigma^2)X'X & -(1/\sigma^4)X'(y - X\beta) \\ -(1/\sigma^4)(y - X\beta)'X & n/(2\sigma^4) - (y - X\beta)'(y - X\beta)/\sigma^6 \end{pmatrix} \quad (6)$$

3 Logit Model with Multiple Regressors

The logit model is a non-linear model, which assumes that the random variable y can only be observed as a binary variable, taking the values 0 or 1. This binary variable is assumed to be the observable counterpart of a latent variable, which itself is a linear combination of a number of independent, non-random variables X (combined into a matrix, or rather into a vector of vectors) and a random error term, that follows a Logistic distribution.

3.1 Maximum Likelihood Function for the Logit Model

This leads to the following likelihood function:

$$L =_{i=0}^{n} p_i^{y_i}(1 - p_i)^{1-y_i} \quad (7)$$

where

$$p_i = \frac{1}{1 + \exp(-x_i'\beta)} \quad (8)$$

The first derivative (of the log-likelihood function) is:

$$\frac{\delta L}{\delta \beta} = -\sum_{i=1}^{n}(1-y_i) - \frac{1}{1+\exp(x_i'\beta)}]x_i(9)$$

and the matrix of second derivatives:

$$H = -\sum_{i=1}^{n} p_i(1-p_i)x_i x_i' \tag{10}$$

4 Newton Raphson

The parameter estimation method we are going to use is called Maximum Likelihood, therefore we need to find the parameter values which will maximise the likelihood function. In order to do this, we will use the Newton Raphson algorithm, which is based on a second order Taylor expansion:

$$L(\beta) = L(\beta^*) + \frac{\delta L}{\delta \beta^*}L(\beta^*)(-\beta^*) + \frac{\delta L^2}{\delta\beta\delta\beta}L(^*\frac{\delta L^2}{\delta\beta\delta\beta} \tag{11}$$

5 Java

First thing we will need to do is to build a little framwork in which our code does fit. This could be done much more sophisticated, but we are just going to use a simple template pattern for the model. It is almost to pretentious to call this an implementation of the "template pattern", since it is nothing more than an abstract class, but I just want to show off my knowledge of the these things!

```
package models;
    import Jama.Matrix;
    public interface IModel
    public Matrix firstDerivative(Matrix x, Matrix y, Matrix parameters);
    public Matrix secondDerivative(Matrix x, Matrix y, Matrix parameters);
```

As you can see, we are using the JAMA (Java Matrix) package for the necessary linear algebra. This is a very easy to use package with quite a small footprint (compared to more sophisticated packages like COLT etc.). This will now be implemented first for the normal distribution:

```
public class NormalModel implements IModel
{

    public Matrix firstDerivative(Matrix x, Matrix y, Matrix parameters)
    {
        int n = x.getRowDimension();
        int k = x.getColumnDimension();

        Matrix beta = new Matrix(k, 1);
        for(int i = 0; i < k; i++) {
            beta.set(i, 0, parameters.get(i, 0));
        }
        double sigma2 = parameters.get(k, 0);

        Matrix xBeta = x.times(beta);
        Matrix yMinusXBeta = y.minus(xBeta);

        double sigma4 = sigma2 * sigma2;

        Matrix xte = x.transpose().times(yMinusXBeta);
        Matrix ete = yMinusXBeta.transpose().times(yMinusXBeta);

        Matrix firstDeriv = new Matrix(parameters.getRowDimension(), 1);
        for(int i = 0; i < k; i++) {
            firstDeriv.set(i, 0, xte.get(i, 0) / sigma2);
        }
        firstDeriv.set(k, 0, ete.get(0, 0) / (2.0*sigma4) - n / (2.0 * sigma2));
        return firstDeriv;
    }

    public Matrix secondDerivative(Matrix x, Matrix y, Matrix parameters)
    {

        int n = x.getRowDimension();
        int k = x.getColumnDimension();

        Matrix beta = new Matrix(k, 1);
        for(int i = 0; i < k; i++) {
            beta.set(i, 0, parameters.get(i, 0));
        }
        double sigma2 = parameters.get(k, 0);
```

```
        Matrix xBeta = x.times(beta);
        Matrix yMinusXBeta = y.minus(xBeta);

        Matrix xtx = x.transpose().times(x);
        Matrix ete = yMinusXBeta.transpose().times(yMinusXBeta);
        Matrix xte = x.transpose().times(yMinusXBeta);

        double sigma4 = sigma2 * sigma2;
        double sigma6 = sigma2 * sigma2 * sigma2;

        Matrix secondDeriv = new Matrix(parameters.getRowDimension(),
         parameters.getRowDimension());

        for(int i = 0; i < k; i++) {
            for(int j = 0; j < k; j++) {
                secondDeriv.set(i, j, -xtx.get(i, j) / sigma2);
            }
        }

        for(int i = 0; i < k; i++) {
            secondDeriv.set(i, k, -xte.get(i, 0) / sigma4);
            secondDeriv.set(k, i, -xte.get(i, 0) / sigma4);
        }

        secondDeriv.set(k, k, n / (2.0 * sigma4) - ete.get(0, 0) / sigma6);

        return secondDeriv;
    }

}
```

Obviously using OLS would be much easier:

```
 public Matrix doOLS(Matrix y, Matrix x)
    {

        int n = x.getRowDimension();
        int k = x.getColumnDimension();

        Matrix xtx = x.transpose().times(x);
        Matrix xty = x.transpose().times(y);
```

```
        Matrix beta = xtx.inverse().times(xty);

        Matrix e = y.minus(x.times(beta));
        Matrix ete = e.transpose().times(e);

        Matrix parameters = new Matrix(k + 1, 1);
        for(int i = 0; i < k; i++) {
            parameters.set(i, 0, beta.get(i, 0));
        }
        parameters.set(k, 0, ete.get(0, 0) / (n - k));

        return parameters;
    }
```

but we want to use Maximum Likelihood and therefore we need the first and second derivatives in order to feed them into the the Newton-Raphson alrogithm:

```
public class NewtonRaphson
{
    IModel model;


    public NewtonRaphson(IModel model)
    {
        this.model = model;
    }

    public Matrix run(Matrix x, Matrix y, Matrix startValues)
    {

        int maxNum = 100;
        double eps = 0.01;
        int counter = 0;

        Matrix parameters = startValues.copy();

        do {

            Matrix H = model.secondDerivative(x, y, parameters);
            if(Math.abs(H.det()) < eps) {
                System.out.println("Hessian is singular!");
```

```
                H.print(2,2);
                counter = -1;
                break;
            }
            Matrix Hi = H.inverse();
            Matrix fd = model.firstDerivative(x, y, parameters);
            parameters = parameters.minus(Hi.times(fd));
            counter++;
        } while(counter < maxNum);

        System.out.println("counter: "+counter);
        return parameters;
    }

}
```

It should be obvious that we would only need to replace the Normal-Model with the Logit-Model but all the other code does not change:

```
public class LogitModel implements IModel
{

    public Matrix firstDerivative(Matrix x, Matrix y, Matrix parameters)
    {
        int n = x.getRowDimension();
        int k = parameters.getRowDimension();
        double[][] firstDeriv = new double[k][1];

        for(int i = 0; i < n; i++) {
            double[] row = x.getArray()[i];
            double one = (1.0 - y.get(i,0)) - pPlus(row, parameters);
            for(int j = 0; j < k; j++) {
                firstDeriv[j][0] += -one * row[j];
            }
        }
        return new Matrix(firstDeriv);

    }

    public Matrix secondDerivative(Matrix x, Matrix y, Matrix parameters)
    {
        int n = x.getRowDimension();
```

```
    int k = parameters.getRowDimension();
    double[][] secondDeriv = new double[k][k];

    for(int i = 0; i < n; i++) {
        double[] row = x.getArray()[i];
        double p = pMinus(row, parameters);
        for(int j = 0; j < k; j++) {
            for(int l = 0; l < row.length; l++) {
                secondDeriv[j][l] += -p * (1.0 - p) * row[j] * row[l];
            }
        }
    }

    return new Matrix(secondDeriv);

}


private double pMinus(double[] x, Matrix beta)
{
    double xtb = 0;
    for(int i = 0; i < x.length; i++) {
        xtb += x[i] * beta.get(i, 0);
    }
    double p = 1.0 / (1.0 + Math.exp(-xtb));
    return p;

}

private double pPlus(double[] x, Matrix beta)
{
    double xtb = 0;
    for(int i = 0; i < x.length; i++) {
        xtb += x[i] * beta.get(i, 0);
    }
    double p = 1.0 / (1.0 + Math.exp(xtb));
    return p;

}
```

```
}
```

So the only thing left for us to do is writing a little test, that creates the apropriate kind of data and that would be it:

```
int n = 10000;
int k = 10;
double sigma = 0.1;

Matrix x = Matrix.random(n, k);
Matrix beta = Matrix.random(k, 1);

Matrix e = new Matrix(n, 1);
Random rand = new Random();
for(int i = 0; i < n; i++) {
    e.set(i, 0, rand.nextGaussian() * Math.sqrt(sigma));
}

Matrix y = x.times(beta).plus(e);

Matrix paras_ols = doOLS(y, x);

//making life more difficult for ML
Matrix startValues = new Matrix(k, 1);
for(int i = 0; i < k; i++) {
    double error = 0.05 - 0.1 * Math.random();
    startValues.set(i, 0, paras_ols.get(i, 0) + error);
}

Matrix paras_ml = doML(yBinary, x, startValues);
```

Well, this is fine for the multivariate normal, but obviously for the Logit Model we will have to constrain the dependent variable (but still use OLS with it for starting values)

```
private Matrix getBinaryY(Matrix y) {
    int n = y.getRowDimension();
    double yMax = -Double.MAX_VALUE;
    double yMin = Double.MAX_VALUE;

    for(int i = 0; i < n; i++) {
        double yValue = y.get(i, 0);
```

```
        if(yValue > yMax) {
            yMax = yValue;
        }
        if(yValue < yMin) {
            yMin = yValue;
        }
    }

    double yLimit = yMin + 0.5 * (yMax - yMin);

    Matrix yBinary = new Matrix(n, 1);

    for(int i=0; i<n; i++) {
        if(y.get(i,0)>yLimit) {
            yBinary.set(i,0,1.0);
        }
    }
    return yBinary;
}
```

6 Ruby

Using Ruby instead of Java will lead to much more concise code, but unfortunately also code that (at least on my machine) does run significantly slower. And we would not need to use the Template Pattern for a dynamic language like Ruby. But what we do need is the Matrix file from the Ruby Standard Library:

```
require 'matrix'
require 'mathn'
```

and we have to use quite a roundabout way to put values into a matrix, as the following method will show:

```
# creating a matrix of
# uniformly distributed variables
def random_matrix(n,k)
  rows = Array.new
  n.times do
    row = Array.new(k)
    row.each_index {|i| row[i] = rand}
```

```
    rows<<row
  end
  m = Matrix.rows(rows, true)
end
```

where we have to make sure that all the elements of the array are arrays themselves. But once we have these things in place we can use the following class for the Multivariate Normal model:

```
#multivariate normal model
class Normal

  def first_derivative(x,y,p)
    n = x.row_size
    k = x.column_size

    b = Array.new(k)
    k.times{|i| b[i]=[p[i,0]]}

    beta = Matrix.rows(b)
    sigma2 = p[k,0]

    sigma4=sigma2*sigma2
    e = y-(x*(beta))
    xte = x.transpose*(e)
    ete = e.transpose*(e)

    #rows of the Jacobian
    rows = Array.new(k+1)
    k.times{|i| rows[i] = [xte[i,0]/sigma2]}
    rows[k] =  [ete[0,0]/(2*sigma4) - n/(2*sigma2)]

    fd = Matrix.rows(rows, true)
  end

  def second_derivative(x,y,p)

    n = x.row_size
    k = x.column_size

    b = Array.new(k)
    k.times{|i| b[i]=[p[i,0]]}
```

```
    beta = Matrix.rows(b)

    sigma2 = p[k,0]

    sigma4=sigma2*sigma2
    sigma6 = sigma2*sigma2*sigma2

    e = y-(x*(beta))
    xtx = x.transpose*(x)
    xte = x.transpose*(e)
    ete = e.transpose*(e)

    #rows of the Hessian
    rows = Array.new(k+1)
    k.times do |i|
      row = Array.new(k+1)
      k.times do |j|
        row[j] = -xtx[i,j]/sigma2
      end
      row[k] = -xte[i,0]/sigma4
      rows[i] = row
    end

    last_row = Array.new(k+1)
    k.times do |i|
      last_row[i] = -xte[i,0]/sigma4
    end
    last_row[k] = 2*sigma4 - ete[0,0]/sigma6
    rows[k] = last_row

    sd = Matrix.rows(rows, true)
  end


end
```

and for the Logit Model:

```
class Logit

  def first_derivative(x,y,p)
    n = x.row_size
```

```
  k = x.column_size
  fd = Array.new(k)
  k.times {|i| fd[i] = [0.0]}

  n.times do |i|
    row = x.row(i).to_a
    value1 = (1-y[i,0]) -p_plus(row,p)
    k.times do |j|
      fd[j][0] -= value1*row[j]
    end
  end
  Matrix.rows(fd, true)
end


def second_derivative(x,y,p)
  n = x.row_size
  k = x.column_size
  sd = Array.new(k)

  k.times do  |i|
    arr = Array.new(k)
    k.times{ |j| arr[j]=0.0}
    sd[i] = arr
  end

  n.times do |i|
    row = x.row(i).to_a
    p_m = p_minus(row,p)
    k.times do |j|
      k.times do |l|
        sd[j][l] -= p_m *(1-p_m)*row[j]*row[l]
      end
    end
  end
  Matrix.rows(sd, true)
end

def p_minus(x_row,p)
  value = 0;
  x_row.each_index { |i| value += x_row[i]*p[i,0]}
```

```
    1/(1+Math.exp(-value))
  end

  def p_plus(x_row,p)
    value = 0;
    x_row.each_index { |i| value += x_row[i]*p[i,0]}
    1/(1+Math.exp(value))
  end

end
```

And the Newton-Raphson algorithm would be:

```
# Newton Raphson without
# automatic stopping criteria
def newton_raphson(x,y,start_values, model)
  # deep copy?
  parameters = start_values

  5.times do
    h = model.second_derivative(x,y,parameters)
    if h.singular?
      puts "Hessian is singular!"
    end
    fd = model.first_derivative(x,y,parameters)
    parameters = parameters-(h.inverse*(fd))
  end

  parameters
end
```

which we can test with the following code:

```
PI = 3.14159265

n = 10000
k=10
sigma2 = 0.1

#independent variables
x = random_matrix(n,k)
```

```
#disturbances
gaussian_array = box_mueller(n)
e= Matrix.rows(gaussian_array, true)

#original parameters
beta = random_matrix(k,1)

#dependent variable
y = x*(beta)+(e)

#parameters to estimate
pa = Array.new(k+1)
k.times{|i| pa[i] = [beta[i,0]]}
pa[k] = [sigma2]

paras = Matrix.rows(pa , true)

#classic OLS estimation
#b = (x.transpose*(x)).inverse*(x.transpose*(y))

#maximum likelihood estimation
new_paras = newton_raphson(x,y,paras, Normal.new)
```

The gaussian distributed error terms we are creating with a simple implementation of the Box-Mueller transformation:

```
# creating an array of gaussian
# distributed variables
def box_mueller(n)

  uniform = Array.new(n)
  n.times do |i|
    uniform[i] = [rand,rand]
  end

  gaussian = Array.new(n)
  n.times do |i|
    array = uniform[i]
    x1 = array[0]
    x2 = array[1]
    gaussian[i] = [Math.sqrt(-2*Math.log(x1))*Math.cos(2*PI * x2)]
```

```
  end

  gaussian
end
```

And if we want to test the Logit model, we will again need to turn the dependent variables into a matrix of bivariates:

```
def bivariate(y)
  n = y.row_size
  y_array = y.column(0).to_a
  max = -1000000
  y_array.each {|v| max = v if v>max }
  min =1000000
  y_array.each {|v| min = v if v<min}

  limit = min + 0.5*(max-min)

  y_bivar = Array.new(n)
  n.times do |i|
    if y_array[i]>limit
      y_bivar[i] = [1]
    else
      y_bivar[i] = [0]
    end
  end
  Matrix.rows(y_bivar, true)
end
```

Well, while the code is running fine, at least the document still needs a lot of work!!