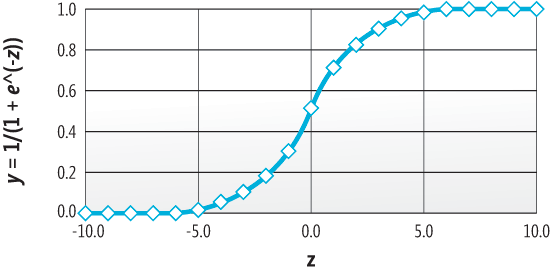
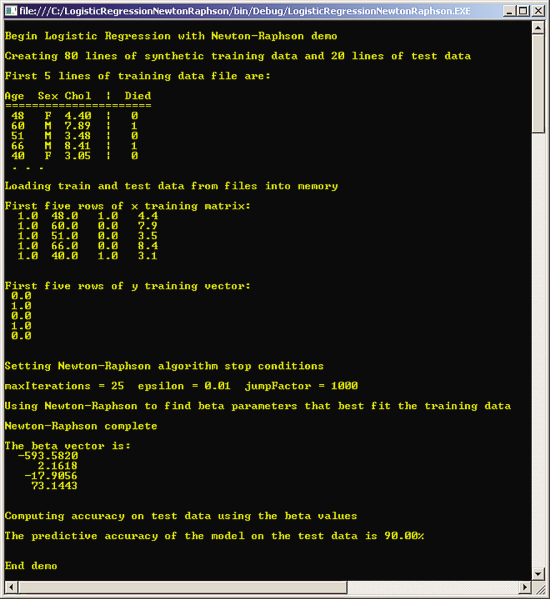
Logistic regression (LR) is a machine-learning technique that can be used to make predictions on data where the dependent variable to be predicted takes a value of 0 or 1. Examples include predicting whether or not a patient will die due to heart disease within a certain number of years (0 = not die, 1 = die), based on the patient’s age, sex and cholesterol level, and predicting whether or not a baseball team will win a game (0 = lose, 1 = win) based on factors such as team batting average and starting pitcher earned run average. Logistic regression assumes that problem data fits an equation that has the form p = 1.0 / (1.0 + e-z) where z = b0 + (b1)(x1) + (b2)(x2) + . . . + (bn)(xn). The x variables are the predictors and the b values are constants that must be determined. For example, suppose you want to predict death from heart disease. Let the predictor variables be x1 = patient age, x2 = patient sex (0 = male, 1 = female) and x3 = patient cholesterol level. And suppose you have somehow determined that b0 = -95.0, b1 = 0.4, b2 = -0.9 and b3 = 11.2. If there’s a 50 year old male patient whose cholesterol level is 6.8, then z = -95.0 + (0.4)(50) + (-0.9)(0) + (11.2)(6.8) = 1.16, and so p = 1.0 / (1.0 + exp(-1.16)) = 0.7613. The p value can loosely be interpreted as a probability, so in this case you’d conclude that the patient has a 0.7613 probability of dying within the specified number of years.

The function 1.0 / (1.0 + e-z) is called the sigmoid function. The domain of possible values for z is all real numbers. The result of the function is a value between 0.0 and 1.0 as shown in **Figure 1**. You can’t assume that the data you’re working with can be modeled by the sigmoid function, but many real-life data sets can in fact be accurately modeled by the function.

  
**Figure 1 The Sigmoid Function**

When using logistic regression, the primary problem is how to determine the b (often called beta) values for the LR equation. In most situations, you have some historical data with known results and use one of several techniques to find the values of beta that best fit the data. After you’ve determined the values of beta, you can use them to make predictions on new data. One of the most common techniques for finding the beta values for a logistic regression equation is called iteratively reweighted least squares (IRLS). IRLS starts with an estimate of the beta values and then iteratively computes a new, better set of betas until some stopping condition is met. There are several techniques that can be used to determine a new, better set of beta values. One of the most common is called Newton-Raphson (NR), which involves finding the calculus derivative of a function—in this case the derivative of the sigmoid function. Because of the close connection between IRLS and Newton-Raphson in logistic regression, the two terms are often used interchangeably.

Although there are plenty of resources available that describe the complex mathematics behind finding logistic regression beta parameters using Newton-Raphson, very few, if any, implementation guides for programmers exist. This article explains exactly how logistic regression with Newton-Raphson works, and how to implement a solution using the C# language. Take a look at the screenshot in **Figure 2** to see where I’m headed.

  
**Figure 2 Logistic Regression with Newton-Raphson**

The demo program begins by generating two synthetic data files. The first is called the training file and consists of 80 lines of age, sex, cholesterol and death data. The training file is used to compute the LR beta values. The second is called the test file and holds 20 lines of data that’s used to evaluate the accuracy of the LR equation with the beta values computed from the training data. The demo program loads the X predictor values of the training data into a matrix, and loads the Y dependent variable values of the data into a vector. Notice that the X training matrix, often called a design matrix, has an additional first column that consists of all 1.0 values, and that the predictor values have all been converted to numeric values. Next, the demo program sets three stopping conditions for the IRLS algorithm, indicated by variables maxIterations, epsilon and jumpFactor. The demo program uses the Newton-Raphson algorithm to estimate the b0, b1, b2 and b3 beta values that best fit the training data. The demo concludes by evaluating how accurate the resulting LR equation with the computed beta values is on the test data. In this example, 18 out of 20 Y values (90 percent) were correctly predicted.

This article assumes you have advanced programming skills and at least an intermediate knowledge of matrix operations and terminology, but doesn’t assume you know anything about logistic regression. The code that produced the screenshot in **Figure 2** is far too large to present in its entirety here, but the complete source code is available at [archive.msdn.microsoft.com/mag201208TestRun](http://archive.msdn.microsoft.com/mag201208TestRun). Because of the complexity of the IRLS/NR algorithm, I’ll focus primarily on key parts of the algorithm rather than on the code itself, so you’ll be able to modify the code to meet your own needs or refactor it to another programming language if you wish.

**Overall Program Structure**

For simplicity, all the code that generated the screenshot in **Figure 2** is contained in a single C# console application. The program structure and Main method, with some WriteLine statements removed, are listed in **Figure 3**.

**Figure 3 Program Structure**

1. **using** System;
2. **using** System.IO;
3. **namespace** LogisticRegressionNewtonRaphson
4. {
5. **class** LogisticRegressionNRProgram
6. {
7. **static** **void** Main(**string**[] args)
8. {
9. **try**
10. {
11. **string** trainFile = "trainFile.txt";
12. **string** testFile = "testFile.txt";
13. MakeRawDataFile(80, 3, trainFile);
14. MakeRawDataFile(20, 4, testFile);
15. Console.WriteLine("First 5 lines of training data file are: \n");
16. DisplayRawData(trainFile, 5);
17. **double**[][] xTrainMatrix = LoadRawDataIntoDesignMatrix(trainFile);
18. **double**[] yTrainVector = LoadRawDataIntoYVector(trainFile);
19. **double**[][] xTestMatrix = LoadRawDataIntoDesignMatrix(testFile);
20. **double**[] yTestVector = LoadRawDataIntoYVector(testFile);
21. **int** maxIterations = 25;
22. **double** epsilon = 0.01;
23. **double** jumpFactor = 1000.0;
24. **double**[] beta = ComputeBestBeta(xTrainMatrix, yTrainVector,
25. maxIterations, epsilon, jumpFactor);
26. Console.WriteLine("Newton-Raphson complete");
27. Console.WriteLine("The beta vector is: ");
28. Console.WriteLine(VectorAsString(beta, **int**.MaxValue, 4, 10));
29. **double** acc = PredictiveAccuracy(xTestMatrix, yTestVector, beta);
30. Console.WriteLine("The predictive accuracy of the model on the test
31. data **is** " + acc.ToString("F2") + "%\n");
32. }
33. **catch** (Exception ex)
34. {
35. Console.WriteLine("Fatal: " + ex.Message);
36. Console.ReadLine();
37. }
38. } // Main
39. **static** **void** MakeRawDataFile(**int** numLines, **int** seed, **string** fileName)
40. **static** **void** DisplayRawData(**string** fileName, **int** numLines)
41. **static** **double**[][] LoadRawDataIntoDesignMatrix(**string** rawDataFile)
42. **static** **double**[] LoadRawDataIntoYVector(**string** rawDataFile)
43. **static** **double** PredictiveAccuracy(**double**[][] xMatrix,
44. **double**[] yVector, **double**[] bVector)
45. **static** **double**[] ComputeBestBeta(**double**[][] xMatrix, **double**[] yVector,
46. **int** maxNewtonIterations, **double** epsilon, **double** jumpFactor)
47. **static** **double**[] ConstructNewBetaVector(**double**[] oldBetaVector,
48. **double**[][] xMatrix,
49. **double**[] yVector, **double**[] oldProbVector)
50. **static** **double**[][] ComputeXtilde(**double**[] pVector, **double**[][] xMatrix)
51. **static** **bool** NoChange(**double**[] oldBvector, **double**[] newBvector, **double** epsilon)
52. **static** **bool** OutOfControl(**double**[] oldBvector, **double**[] newBvector,
53. **double** jumpFactor)
54. **static** **double**[] ConstructProbVector(**double**[][] xMatrix, **double**[] bVector)
55. // Matrix and vector routines:
56. **static** **double** MeanSquaredError(**double**[] pVector, **double**[] yVector)
57. **static** **double**[][] MatrixCreate(**int** rows, **int** cols)
58. **static** **double**[] VectorCreate(**int** rows)
59. **static** **string** MatrixAsString(**double**[][] matrix, **int** numRows,
60. **int** digits, **int** width)
61. **static** **double**[][] MatrixDuplicate(**double**[][] matrix)
62. **static** **double**[] VectorAddition(**double**[] vectorA, **double**[] vectorB)
63. **static** **double**[] VectorSubtraction(**double**[] vectorA, **double**[] vectorB)
64. **static** **string** VectorAsString(**double**[] vector, **int** count, **int** digits, **int** width)
65. **static** **double**[] VectorDuplicate(**double**[] vector)
66. **static** **double**[][] MatrixTranspose(**double**[][] matrix) // assumes
67. matrix **is** not **null**
68. **static** **double**[][] MatrixProduct(**double**[][] matrixA, **double**[][] matrixB)
69. **static** **double**[] MatrixVectorProduct(**double**[][] matrix, **double**[] vector)
70. **static** **double**[][] MatrixInverse(**double**[][] matrix)
71. **static** **double**[] HelperSolve(**double**[][] luMatrix, **double**[] b) // helper
72. **static** **double**[][] MatrixDecompose(**double**[][] matrix, **out** **int**[] perm,
73. **out** **int** tog)
74. } // Program
75. } // ns

Although logistic regression is a complex topic, the code in **Figure 3** is not quite as complicated as it might first appear because most of the methods shown are relatively short helper routines. The two key methods are ComputeBestBeta and ConstructNewBetaVector.

The MakeRawData method generates a file of quasi-random age-sex-cholesterol-death data. Age is a random integer value between 35 and 80, sex is either M or F with equal probability, and cholesterol is a semi-random real value between 0.1 and 9.9 that’s based on the current age value. The death dependent variable is computed using a logistic regression equation with fixed beta values of b0 = -95.0, b1 = 0.4, b2 = -0.9 and b3 = 11.2. So MakeRawData generates data that can definitely be modeled using LR, as opposed to generating purely random data that would likely not follow an LR model.

**Computing a New Beta Vector**

The heart of logistic regression with Newton-Raphson is a routine that computes a new, presumably better, set of beta values from the current set of values. The math is very deep, but fortunately the net result is not too complex. In pseudo-equation form, the update process is given by:

b[t] = b[t-1] + inv(X'W[t-1]X)X'(Y - p[t-1])

Here b[t] is the new (“at time t,” not array indexing) beta vector. On the right-hand side, b[t-1] is the old (“at time t-1”) beta vector. The inv function is matrix inversion. Uppercase X is the design matrix—that is, the values of the predictor variables augmented with a leading column of 1.0s. Uppercase X’ is the transpose of the X design matrix. Uppercase Y is the vector of dependent variable values (recall each value will be 0 or 1). The quantity p[t-1] represents the vector of old predicted probability values for Y (which will consist of values between 0.0 and 1.0). The uppercase W quantity is a so-called weights matrix, which requires a bit of explanation.

The beta update equation and the W matrix are best explained with a concrete example. Suppose for simplicity that the training set consists only of the first five lines of data shown in **Figure 1**. So the design matrix X would be:

1.00    48.00     1.00     4.40

1.00    60.00     0.00     7.89

1.00    51.00     0.00     3.48

1.00    66.00     0.00     8.41

1.00    40.00     1.00     3.05

The Y dependent variable vector would be:

0

1

0

1

0

Let’s assume that the old beta vector of values to be updated, b[t-1], is:

1.00

0.01

0.01

0.01

With these values for X and beta, the old p vector, p[t-1], is:

0.8226

0.8428

0.8242

0.8512

0.8085

Notice that if we presume p values < 0.5 are interpreted as y = 0 and p values >= 0.5 are interpreted as y = 1, the old beta values would correctly predict only two out of five cases in the training data.

The weights matrix W is an m x m matrix where m is the number of rows of X. All the values in the W matrix are 0.0 except for those m values that are on the main diagonal. Each of these values equals the corresponding p value multiplied by 1-p. So for this example, W would be size 5 x 5. The upper-left cell at [0,0] would be (0.8226)(1 - 0.8226) = 0.1459. The cell at [1,1] would be (0.8428)(1 - 0.8428) = 0.1325, and so on. The quantity (p)(1-p) represents the calculus derivative of the sigmoid function.

In practice, the W matrix is not computed explicitly because its size could be huge. If you had 1,000 rows of training data, matrix W would have 1,000,000 cells. Notice the beta update equation has a term W[t-1]X, which means the matrix product of W[t-1] and X. Because most of the values in W[t-1] are zero, most of the matrix multiplication terms are also zero. This allows W[t-1] times X to be computed directly from p[t-1] and X, without explicitly constructing W. Several of the math references that describe IRLS with the NR algorithm for LR use the symbol X~ (X tilde) for the product of W[t-1] and X. See method ComputeXtilde in the code download for implementation details.

Method ConstructNewBetaVector accepts as input parameters the old beta vector, the X design matrix, the Y dependent variable vector and the old probability vector. The method computes and returns the updated beta vector.

The method is implemented like so:

1. **double**[][] Xt = MatrixTranspose(xMatrix);                // X'
2. **double**[][] A = ComputeXtilde(oldProbVector, xMatrix);    // WX
3. **double**[][] B = MatrixProduct(Xt, A);                     // X'WX
4. **double**[][] C = MatrixInverse(B);                         // inv(X'WX)
5. **if** (C == **null**)                                           // inverse failed!
6. **return** **null**;
7. **double**[][] D = MatrixProduct(C, Xt);                     // inv(X'WX)X'
8. **double**[] YP = VectorSubtraction(yVector, oldProbVector); // Y-p
9. **double**[] E = MatrixVectorProduct(D, YP);                 // inv(X'WX)X'(y-p)
10. **double**[] result = VectorAddition(oldBetaVector, E);
11. **return** result;

With the collection of matrix and vector helper methods shown in **Figure 3**, computing a new beta vector is fairly straightforward. Note that the method performs matrix inversion. This is a process that can go wrong in many ways and is a significant weakness of Newton-Raphson.

Continuing the example, matrix Xt (the transpose of X) would be:

 1.00     1.00     1.00     1.00     1.00

48.00    60.00    51.00    66.00    40.00

 1.00     0.00     0.00     0.00     1.00

 4.40     7.89     3.48     8.41     3.05

Matrix A (X~) would be computed from vector p and matrix X by helper method ComputeXtilde as:

0.15     7.00     0.15     0.64

0.13     7.95     0.00     1.05

0.14     7.39     0.00     0.50

0.13     8.36     0.00     1.07

0.15     6.19     0.15     0.47

Intermediate matrix B, representing the product of Xt and X~ (which in turn is XtW[t-1]X) would be:

 0.70    36.90     0.30     3.73

36.90  1989.62    13.20   208.46

 0.30    13.20     0.30     1.11

 3.73   208.46     1.11    23.23

Intermediate matrix C is the inverse of matrix B and would be:

 602.81   -14.43  -110.41    38.05

 -14.43     0.36     2.48    -1.02

-110.41     2.48    26.43    -5.77

  38.05    -1.02    -5.77     3.36

Intermediate matrix D is the product of matrix C and matrix X-transpose and would be computed as:

-33.00    37.01    -0.90   -29.80    31.10

  0.77    -0.96     0.30     0.66    -0.72

  9.52    -7.32    -4.17     4.54    -2.51

 -1.86     3.39    -2.24    -0.98     1.76

Intermediate vector YP is the difference between vectors Y and p[t-1] and would be:

-0.8

 0.2

-0.8

 0.1

-0.8

Intermediate vector E is the product of matrix D and vector YP and holds the increments to be added to the old beta vector. Vector E would be:

 4.1

-0.4

-2.8

 2.3

The new, final beta vector is obtained by adding the values in intermediate vector E to the old values of beta, and in this example would be:

 5.1

-0.3

-2.8

 2.4

With the new values of beta, the new values for the p vector would be:

0.0240

0.9627

0.0168

0.9192

0.0154

If these p values are interpreted as Y = 0 when p < 0.5, then, after only one iteration of Newton-Raphson, the beta values correctly predict all five cases in the test data.

**Knowing When to Stop**

The Newton-Raphson technique for logistic regression iteratively improves the values of the beta vector until some stopping condition is met. It’s surprisingly difficult to know when to stop the iteration. Method ComputeBestBeta handles this task. The code is presented in **Figure 4**.

**Figure 4 Computing the Best Beta Vector**

1. **static** **double**[] ComputeBestBeta(**double**[][] xMatrix, **double**[] yVector,
2. **int** maxIterations, **double** epsilon, **double** jumpFactor)
3. {
4. **int** xRows = xMatrix.Length;
5. **int** xCols = xMatrix[0].Length;
6. **if** (xRows != yVector.Length)
7. **throw** **new** Exception("xxx (error)");
8. // Initialize beta values
9. **double**[] bVector = **new** **double**[xCols];
10. **for** (**int** i = 0; i < xCols; ++i) { bVector[i] = 0.0; }
11. **double**[] bestBvector = VectorDuplicate(bVector);
12. **double**[] pVector = ConstructProbVector(xMatrix, bVector);
13. **double** mse = MeanSquaredError(pVector, yVector);
14. **int** timesWorse = 0;
15. **for** (**int** i = 0; i < maxIterations; ++i)
16. {
17. **double**[] newBvector = ConstructNewBetaVector(bVector, xMatrix,
18. yVector, pVector);
19. **if** (newBvector == **null**)
20. **return** bestBvector;
21. **if** (NoChange(bVector, newBvector, epsilon) == **true**)
22. **return** bestBvector;
23. **if** (OutOfControl(bVector, newBvector, jumpFactor) == **true**)
24. **return** bestBvector;
25. pVector = ConstructProbVector(xMatrix, newBvector);
26. **double** newMSE = MeanSquaredError(pVector, yVector); // Smaller is better
27. **if** (newMSE > mse) // new MSE is worse
28. {
29. ++timesWorse;           // Update got-worse counter
30. **if** (timesWorse >= 4)
31. **return** bestBvector;
32. bVector = VectorDuplicate(newBvector); // Attempt escape
33. **for** (**int** k = 0; k < bVector.Length; ++k)
34. bVector[k] = (bVector[k] + newBvector[k]) / 2.0;
35. mse = newMSE;
36. }
37. **else** // Improved
38. {
39. bVector = VectorDuplicate(newBvector);
40. bestBvector = VectorDuplicate(bVector);
41. mse = newMSE;
42. timesWorse = 0;
43. }
44. } // End main iteration loop
45. **return** bestBvector;
46. }

One of the biggest pitfalls of logistic regression is iterating too many times, resulting in a set of beta values that fit the training data perfectly but don’t fit any other data sets well. This is called over-fitting. Knowing when to stop the training process in logistic regression is part art and part science. Method ComputeBestBeta begins by initializing the beta vector to all 0.0 values, computing the associated p values, and then computing the error between the p values and the Y values. The code in this article uses mean squared error—the average of the sums of squared differences between p and Y. Other possibilities for computing error include average absolute deviation and cross-entropy error.

The main processing loop in ComputeBestBeta repeatedly computes a new beta vector and corresponding error term. The primary stopping condition in the algorithm is a maxIterations parameter. Recall that the Newton-Raphson algorithm computes a matrix inverse, which is susceptible to failing. In this case ComputeBestBeta returns the best beta vector found (which, unfortunately, could be the initial all-zero vector). You might want to throw an exception here instead. Another alternative is to attempt an escape by modifying the new beta values by adjusting them to the average of the previous values and the new values.

The next stop condition checks to see if the change in all the new beta values is smaller than some small value of parameter epsilon, using helper method NoChange. This indicates that Newton-Raphson has converged and, in fact, there’s a good chance your model is now over-fitted. Instead of returning the best beta vector found at this point, you might want to return the best beta vector from an earlier iteration. The next stop condition checks to see if any of the new beta values have jumped by an amount greater than some large value given by parameter jumpFactor. This indicates that Newton-Raphson has possibly spun out of control and you want to throw an exception instead of retuning the best beta vector found.

Another stop condition in ComputeBestBeta checks to see if the new beta values actually produce a larger error than the previous beta values did. In this example, if new betas produce a larger error four times in a row, the algorithm terminates and returns the best beta vector found. You might want to parameterize the value for the maximum number of consecutive increases in error. When an increase in error is detected, the method attempts to escape the situation by changing the values in the current beta vector to the average of the values in the current beta and the values in the newly computed beta.

There’s no single best set of stopping conditions. Every logistic regression problem requires a bit of experimentation to tune the Newton-Raphson algorithm.

**Advantages vs. Disadvantages**

There are several alternatives to Newton-Raphson for estimating the best set of beta values in logistic regression. Newton-Raphson is a deterministic numerical-optimization technique. You can also employ non-deterministic (meaning probabilistic) techniques such as particle swarm optimization, evolutionary algorithms and bacterial foraging optimization.

The primary advantage of Newton-Raphson compared to probabilistic alternatives is that in most situations NR is much faster. But Newton-Raphson has several disadvantages. Because NR uses matrix inversion, the algorithm will fail when encountering a singular matrix during computation. Simple implementations of Newton-Raphson require all data to be in memory, which means there’s a limit to the size of the training and test data sets you can use. Although it’s rare, some datasets might not converge to a set of best beta values at all using NR.

When I use logistic regression, I often employ a two-pronged attack. I first experiment with a Newton-Raphson approach. If that technique fails, I fall back to using particle swarm optimization to find the best set of beta values. It’s important to note that logistic regression isn’t magic, and not all data fits a logistic regression model. Other machine-learning techniques to model data with a binary-dependent variable include neural networks, support vector machines and linear-discriminant analysis.

The explanation of the beta vector update process of the Newton-Raphson algorithm presented in this article and the accompanying code download should get you up and running with logistic regression using NR. Logistic regression is a fascinating, complex topic and can make a valuable addition to your personal skill set.