Linear Regression Using C#

The goal of a linear regression problem is to predict the value of a numeric variable based on the values of one or more numeric predictor variables. For example, you might want to predict the annual income of a person based on his education level, years of work experience and sex (male = 0, female = 1).

The variable to predict is usually called the dependent variable. The predictor variables are usually called the independent variables. When there's just a single predictor variable, the technique is sometimes called simple linear regression. When there are two or more predictor variables, the technique is generally called multiple, or multivariate, linear regression.

A good way to see where this article is headed is to take a look at the demo program in **Figure 1**. The C# demo program predicts annual income based on education, work and sex. The demo begins by generating 10 synthetic data items. Education level is a value between 12 and 16. Work experience is a value between 10 and 30. Sex is an indicator variable where male is the reference value, coded as 0, and female is coded as 1. Income, in thousands of dollars, is in the last column. In a non-demo scenario, you'd probably read data from a text file using a method named something like MatrixLoad.

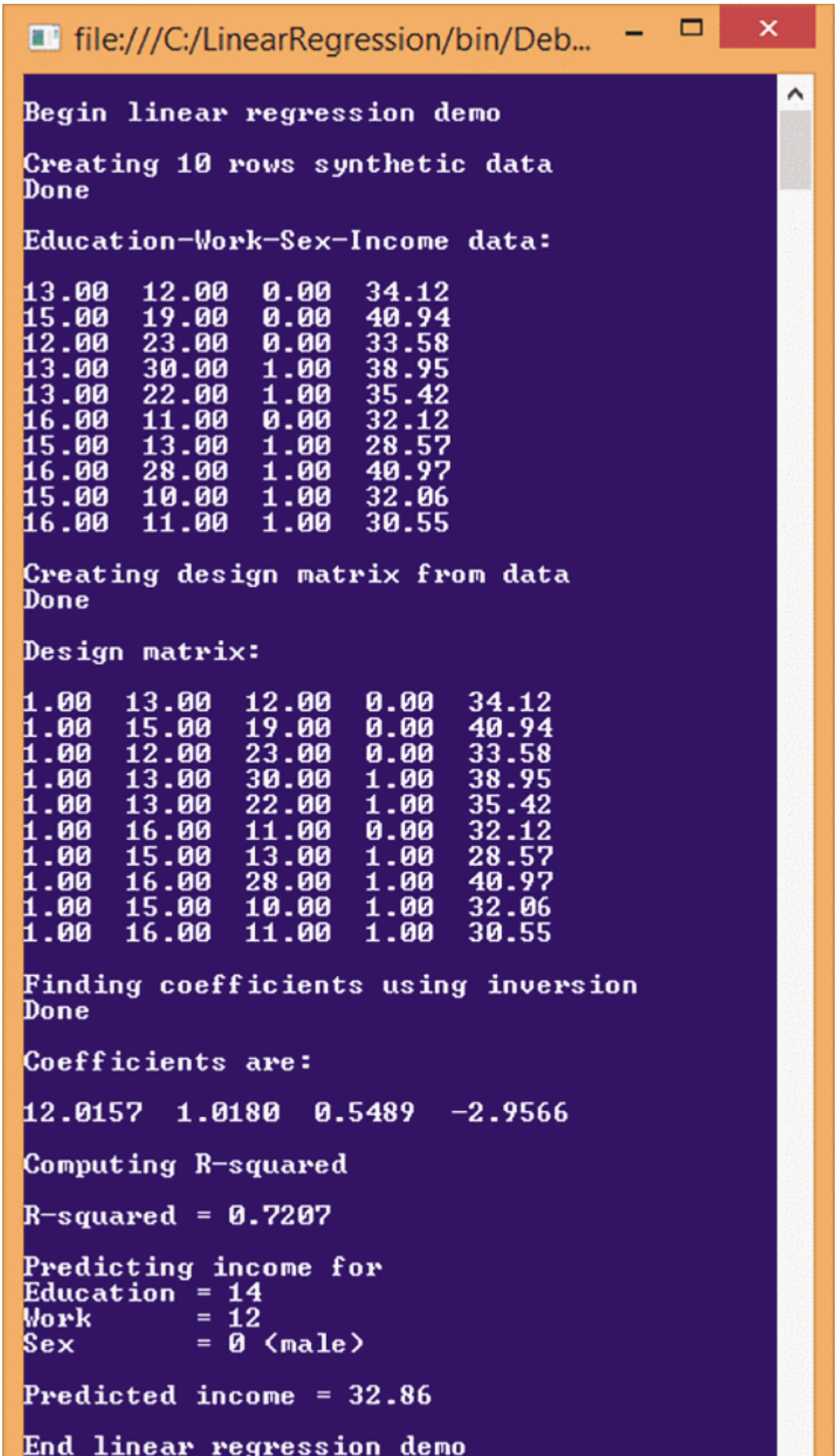
file:///C:/LinearRegression/bin/Deb...
Begin linear regression demo
Creating 10 rows synthetic data
Done
Education-Work-Sex-Income data:
13.00 12.00 0.00 34.12
15.00 19.00 0.00 40.94
12.00 23.00 0.00 33.58
13.00 30.00 1.00 38.95
13.00 22.00 1.00 35.42
16.00 11.00 0.00 32.12
15.00 13.00 1.00 28.57
16.00 28.00 1.00 40.97
15.00 10.00 1.00 32.06
16.00 11.00 1.00 30.55
Creating design matrix from data
Done
Design matrix:
1.00 13.00 12.00 0.00 34.12
1.00 15.00 19.00 0.00 40.94
1.00 12.00 23.00 0.00 33.58
1.00 13.00 30.00 1.00 38.95
1.00 13.00 22.00 1.00 35.42
1.00 16.00 11.00 0.00 32.12
1.00 15.00 13.00 1.00 28.57
1.00 16.00 28.00 1.00 40.97
1.00 15.00 10.00 1.00 32.06
1.00 16.00 11.00 1.00 30.55
Finding coefficients using inversion
Done
Coefficients are:
12.0157 1.0180 0.5489 -2.9566
Computing R-squared
R-squared = 0.7207
Predicting income for
Education = 14
Work = 12
Sex = 0 (male)
Predicted income = 32.86
End linear regression demo

Figure 1 Linear Regression Using C#

After generating the synthetic data, the demo program uses the data to create what's called a design matrix. A design matrix is just the data matrix with a leading column of all 1.0 values added. There are several different algorithms that can be used for linear regression; some can use the raw data matrix while others use a design matrix. The demo uses a technique that requires a design matrix.

After creating the design matrix, the demo program finds the values for four coefficients, (12.0157, 1.0180, 0.5489, -2.9566). The coefficients are sometimes called b-values or beta-values. The first value, 12.0157, is usually called the intercept. It's a constant not associated with any predictor variable. The second, third and fourth coefficient values (1.0180, 0.5489, -2.9566) are associated with education level, work experience and sex, respectively.

The very last part of the output in **Figure 1** uses the values of the coefficients to predict the income for a hypothetical person who has an education level of 14; 12 years of work experience; and whose sex is 0 (male). The predicted income is 32.86, which is calculated like so:

```
income = 12.0157 + (1.0180)(14) + (0.5489)(12) + (-2.9566)(0)
                 = 12.0157 + 14.2520 + 6.5868 + 0
                 = 32.86
```

In other words, to make a prediction using linear regression, the predictor values are multiplied by their corresponding coefficient values and summed. It's very simple. Notice that the leading intercept value (12.0157 in the example) can be considered a coefficient associated with a predictor variable that always has a value of 1. This fact, in part, explains the column of 1.0 values in the design matrix.

The essence of a linear regression problem is calculating the values of the coefficients using the raw data or, equivalently, the design matrix. This is not so easy. The demo uses a technique called closed form matrix inversion, also known as the ordinary least squares method. Alternative techniques for finding the values of the coefficients include iteratively reweighted least squares, maximum likelihood estimation, ridge regression, gradient descent and several others.

In **Figure 1**, before the prediction is made, the demo program computes a metric called the R-squared value, which is also called the coefficient of determination. R-squared is a value between 0 and 1 that describes how well the prediction model fits the raw data. This is sometimes expressed as, "the percentage of variation explained by the model." Loosely interpreted, the closer R-squared is to 1, the better the

prediction model is. The demo value of 0.7207, or 72 percent, would be considered relatively high (good) for real-world data.
This article assumes you have at least intermediate C# programming skills, but doesn't assume you know anything about linear regression. The demo program is too long to present in its entirety, but the complete source code is available in the download that accompanies this article.

Understanding Linear Regression

Linear regression is usually best explained using a diagram. Take a look at the graph in **Figure 2**. The data in the graph represents predicting annual income from just a single variable, years of work experience. Each of the red dots corresponds to a data point. For example, the leftmost data item has work = 10 and income = 32.06. Linear regression finds two coefficients: one intercept and one for the work variable. As it turns out, the values of the coefficients are 27.00 and 0.43.

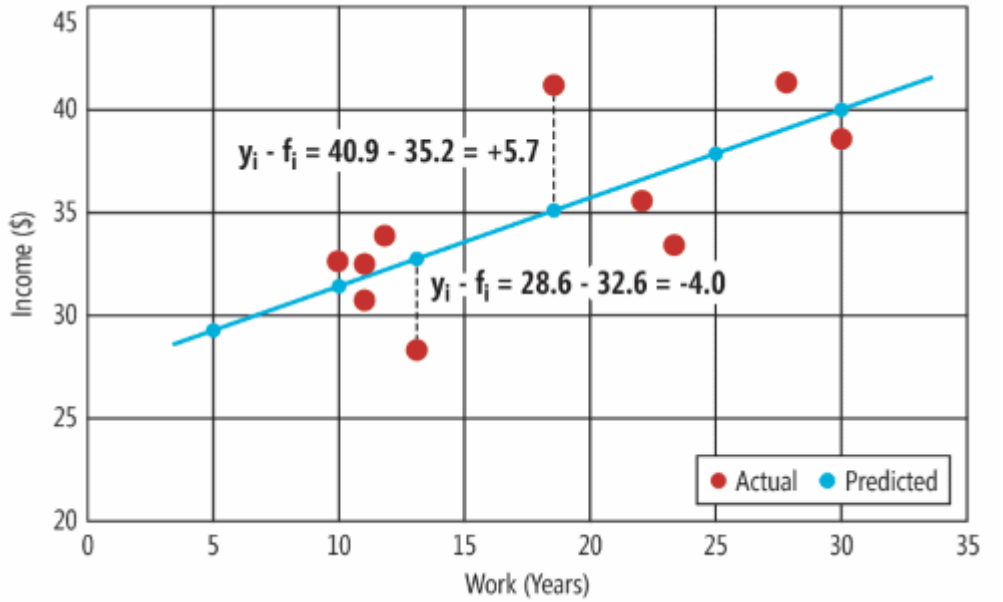

Figure 2 Linear Regression with One Independent Variable

The coefficient values determine the equation of a line, which is shown in blue in **Figure 2**. The line (the coefficients) minimizes the sum of the squared deviations between the actual data points (yi) and the predicted data points (fi). Two of the 10 deviations are shown with dashed lines in **Figure 2**. The first deviation shown is yi - fi = 28.6 - 32.6 = -4.0. Notice that deviations can be positive or negative. If the deviations weren't squared, the negatives and positives could cancel each other out.
The graph in **Figure 2** shows how simple linear regression, with just one independent variable, works. Multivariate linear regression extends the same idea—find coefficients that minimize the sum of squared deviations—using several independent variables.

Expressed intuitively, linear regression finds the best line through a set of data points. This best line can be used for prediction. For example, in **Figure 2**, if a hypothetical person had 25 years of work experience, her predicted income on the blue line would be about 38.

Solving the Least Squares Equation

If a linear regression problem has n predictor variables, then n+1 coefficient values must be found, one for each predictor plus the intercept value. The demo program uses the most basic technique to find the coefficient values. The values of the coefficients are often given using the somewhat intimidating equation shown in **Figure 3**. The equation is not as complicated as it might first appear.

$$\beta = (\mathbf{X}^{T} * \mathbf{X})^{-1} * \mathbf{X}^{T} * \mathbf{Y}$$

Figure 3 Linear Regression Coefficients Solution Using Matrices

The Greek letter beta resembles a script B and represents the coefficient values. Notice that all the letters in the equation are in bold, which in mathematics indicates they represent multi-valued objects (matrices or arrays/vectors) rather than simple scalar values (plain numbers). Uppercase X represents the design matrix. Uppercase X with a T exponent means the transpose of the design matrix. The * symbol means matrix multiplication. The -1 exponent means matrix inversion. Uppercase Y is a column vector (a matrix with one column) of the dependent variable values. Therefore, solving for the values of the coefficients really means understanding matrix operations.

The diagrams in **Figure 4** illustrate matrix transposition, matrix multiplication, and matrix inversion. The transpose of a matrix just swaps rows and columns. For example, suppose you have a 2x3 matrix, that is, one with 2 rows and 3 columns. The transpose of the matrix will be 3x2 where the rows of the original matrix become the columns of the transpose matrix.

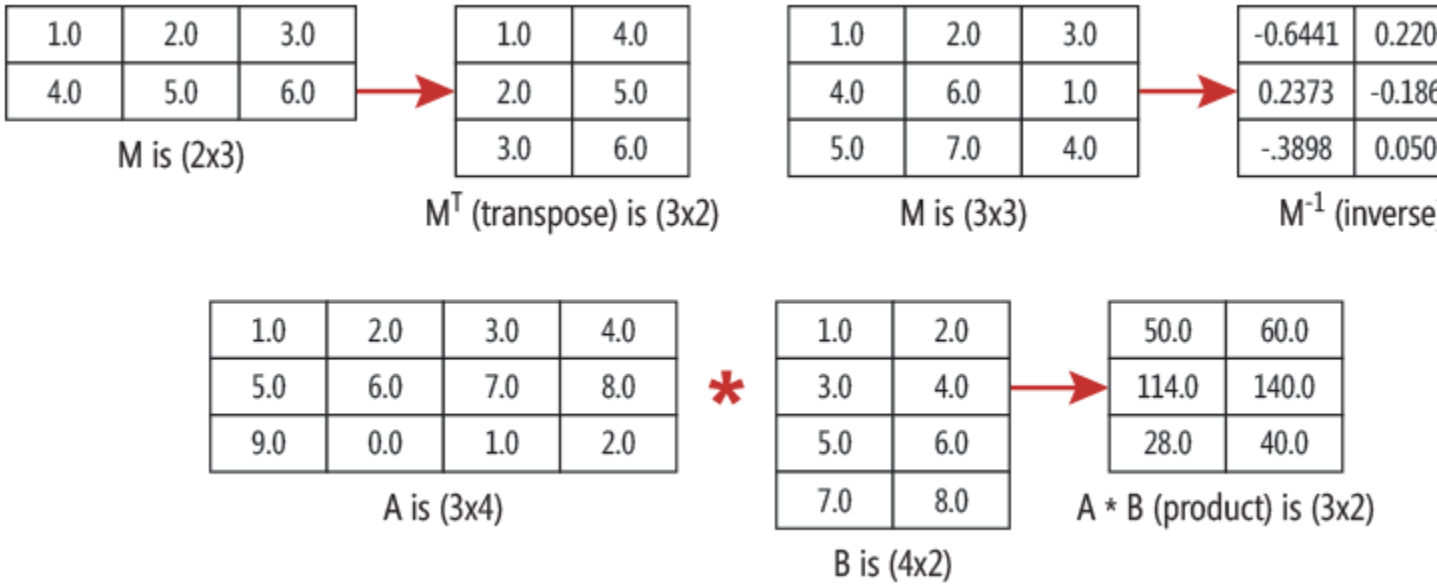

Figure 4 Three Matrix Operations Used to Find Linear Regression Coefficients

Matrix multiplication may seem a bit odd if you haven't encountered it before. If you multiply an (n x m) sized matrix times an (m x p) sized matrix, the result is an (n x p) sized matrix. For example, a 3x4 * a 4x2 matrix has size 3x2. A detailed discussion of matrix multiplication is outside the scope of this article, but once you've seen a few examples, the process is easy to understand and easy to implement in code.

The third matrix operation needed to solve for linear regression coefficient values is matrix inversion, which, unfortunately, is difficult to grasp and difficult to implement. For the purposes of this article, it's enough to know that that the inverse of a matrix is defined only when the matrix has the same number of rows and columns (a square matrix). **Figure 4**shows a 3x3 matrix and its inverse.
There are several algorithms that can be used to find the inverse of a matrix. The demo program uses a technique called Doolittle's decomposition.

To summarize, a linear regression problem with n predictor variables involves finding the values for n+1 coefficients. This can be done using matrices with matrix transposition, matrix multiplication and matrix inversion. Transposition and multiplication are easy, but finding the inverse of a matrix is difficult.

The Demo Program Structure

To create the demo program, I launched Visual Studio and selected the console application project template. I named the project LinearRegression. The program has no significant .NET Framework dependencies so any version of Visual Studio will work.

After the template code loaded into the editor, in the Solution Explorer window, I right-clicked on file Program.cs and renamed it to LinearRegressionProgram.cs. I allowed Visual Studio to automatically rename class Program. At the top of the Editor window, I deleted all using statements except the one referencing the top-level System namespace.

The overall structure of the demo program, with a few minor edits to save space, is presented in **Figure 5**. All the program control logic is in the Main method. The demo program uses a static-method approach rather than an OOP approach.

Figure 5 Linear Regression Demo Program Structure

```
using System;
namespace LinearRegression
{
  class LinearRegressionProgram
  {
    static void Main(string[] args)
    {
      Console.WriteLine("Begin linear regression demo");
      // Generate synthetic data
      // Create design matrix
      // Solve for LR coefficients
      // Calculate R-squared value
      // Do a prediction
      Console.WriteLine("End linear regression demo");
      Console.ReadLine();
    }
    static double Income(double x1, double x2,
      double x3, double[] coef) { . . }
    static double RSquared(double[][] data,
      double[] coef) { . . }
    static double[][] DummyData(int rows,
      int seed) { . . }
    static double[][] Design(double[][] data) { . . }
    static double[] Solve(double[][] design) { . . }
    static void ShowMatrix(double[][] m, int dec) { . . }
    static void ShowVector(double[] v, int dec) { . . }
    // ----------
    static double[][] MatrixTranspose(double[][] matrix)
      { . . }
    static double[][] MatrixProduct(double[][] matrixA,
      double[][] matrixB) { . . }
    static double[][] MatrixInverse(double[][] matrix)
      { . . }
    // Other matrix routines here
  }
} // ns
```

Method Income returns a predicted income from input parameters with values for education level, work experience and sex, using an array of coefficient values.

Method RSquared returns the R-squared value of the model from the data and the coefficients. Method DummyData generates the synthetic data used for the demo.

Method Design accepts a matrix of data and returns an augmented design matrix with a leading column of 1.0 values. Method Solve accepts a design matrix and uses matrix operations to find the linear regression coefficients.

Most of the hard work is done by a set of static methods that perform matrix operations. The demo program defines a matrix in the simplest way possible, as an array of arrays. An alternative is to create a program-defined Matrix class, but in my opinion that approach is unnecessarily complicated. Sometimes ordinary arrays are better than program-defined objects.

Method MatrixTranspose returns the transposition of a matrix. Method MatrixProduct returns the result of multiplying two matrices. Method MatrixInverse returns the inverse of a matrix. The demo has many helper methods. In particular, method MatrixInverse calls helper methods MatrixDuplicate, MatrixDecompose and HelperSolve.

The Solve Method

The heart of the linear regression demo program is method Solve. The method's definition begins with:

```
static double[] Solve(double[][] design)
{
  int rows = design.Length;
  int cols = data[0].Length;
  double[][] X = MatrixCreate(rows, cols - 1);
  double[][] Y = MatrixCreate(rows, 1);
...
```

The single input parameter is a design matrix. An alternative approach you might want to consider is to pass the source data matrix and then have Solve call helper method Design to get the design matrix. Helper method MatrixCreate allocates space for, and returns, a matrix with the specified number of rows and columns. The local X matrix holds the values of the independent variables (with a leading 1.0 value). The local Y matrix has just one column and holds the values of the dependent variable (annual income in the demo).

Next, the cells in matrices X and Y are populated using the values in the design matrix:

```
int j;
for (int i = 0; i < rows; ++i)
{
  for (j = 0; j < cols - 1; ++j)
  {
    X[i][j] = design[i][j];
  }
  Y[i][0] = design[i][j]; // Last column
}
```

Notice that index variable j is declared outside the nested for loops so it can be used to populate the Y matrix. With the X and Y matrices in hand, the linear regression coefficients can be found according to the equation shown in **Figure 3**:

```
...
  double[][] Xt = MatrixTranspose(X);
  double[][] XtX = MatrixProduct(Xt, X);
  double[][] inv = MatrixInverse(XtX);
  double[][] invXt = MatrixProduct(inv, Xt);
  double[][] mResult = MatrixProduct(invXt, Y);
  double[] result = MatrixToVector(mResult);
  return result;
} // Solve
```

In the demo, matrix X has size 10x4 so its transpose, Xt, has size 4x10. The product of Xt and X has size 4x4 and the inverse, inv, also has size 4x4. In general, for a linear regression problem with n independent predictor variables, when using the matrix inversion technique you'll need to find the inverse of a matrix with size (n+1) x (n+1). This means the inversion technique isn't suitable for linear regression problems that have a huge number of predictor variables.

The product of the 4x4 inverse matrix and the 4x10 transpose matrix, invXt in the code, has size 4x10. The product of invXt and the 10x1 Y matrix, mResult ("matrix result") in the code, has size 4x1. These values are the coefficients you need. For convenience, the values in the single column Y matrix are transferred to an ordinary array using helper method MatrixToVector.

Calculating R-Squared

As noted earlier, the R-squared metric is a measure of how well the actual data points fit the computed regression line. In math terms, R-squared is defined as R2 = 1 - (SSres / SStot). The SSres term is usually called the "residual sum of squares." It's the sum of the squared differences between the actual Y values and the predicted Y values, as illustrated in the graph in **Figure 2**. The SStot term is the "total sum of squares." It's the sum of the squared differences between each actual Y value and the mean (average) of all of the actual Y values.

The R-squared metric in linear regression is also called the coefficient of determination and is related to, but different from, another statistical metric named r-squared ("little r-squared"). Interpreting R-squared is a bit tricky and depends on the particular problem domain under investigation. For the natural and social sciences, where data is typically messy and incomplete, an R-squared value of 0.6 or greater is often considered pretty good.

There's an alternative measure of the variance explained by the regression model called the adjusted R-squared. This metric takes into account the number of predictor variables and the number of data items. For most purposes, using the plain R-squared value is good enough to get an idea of the predictive quality of a linear regression model.

Wrapping Up

If you search the Internet for examples of how to perform linear regression using a programming language, you won't find very many references. I think there are two main reasons for this relative lack of information. First, solving for linear regression coefficients using matrix operations is quite difficult, mostly because of the matrix inversion operation. In some ways, I consider the MatrixInverse method of the demo program to be among the most complicated code routines I've ever written. Second, there are plenty of existing standalone tools that can perform linear regression, in particular, the Excel spreadsheet program with its Data Analysis add-in. It's relatively rare to need to directly embed linear regression solution code in a software system.

Linear regression has been studied for decades and there are many ways to extend the technique. For example, you can introduce what are called interaction effects that combine two or more predictor variables. These extensions are sometimes called general linear models to distinguish them from the basic form of linear regression.

In my opinion, linear regression is the "Hello World" technique of classical statistics. There's no clear, universally agreed upon distinction between classical statistics and machine learning, but I tend to think of classical statistics techniques as those that were first studied by mathematicians starting in the early 1900s. In my mind, machine learning techniques, like neural network classification, are those that are more recent, first appearing in the 1950s. Classical statistics linear regression is closely related to a machine learning technique called logistic regression, which has been the topic of several Test Run columns.