Least-Squares Fitting of Data with Polynomials

David Eberly
Geometric Tools, LLC
http://www.geometrictools.com/

Created: October 17, 2005
Last Modified: February 9, 2008

Contents

1 Fitting with Standard Polynomials

Given a set of n points $\{(x_k, y_k)\}_{k=0}^{n-1}$ with $x_{k+1} > x_k$ for all k, the goal is to fit the data with a polynomial of degree d, namely,

$$y = \sum_{i=0}^{d} c_i x^i$$

The least-squares method chooses the coefficients c_i to minimize the squared error function

$$E(c_0, \ldots, c_d) = \sum_{k=0}^{n} \left(\sum_{i=0}^{d} c_i x_k^i - y_k \right)^2$$

This is a nonnegative quadratic function with respect to the coefficients and obtains its global minimum when the gradient of E is zero; that is, the partial derivatives of E must be zero:

$$0 = \frac{\partial E}{\partial c_j} = 2 \sum_{k=0}^{n-1} \left(\sum_{i=0}^{d} c_i x_k^i - y_k \right) x_k^j$$

for $0 \le j \le d$. The equation simplifies to

$$\sum_{i=0}^{d} \left(\sum_{k=0}^{n-1} x_k^{j+i} \right) c_i = \sum_{k=0}^{n-1} x_k^j y_k$$

This is a linear system of $d+1$ equations in $d+1$ unknowns and is of the form

$$A^{\mathrm{T}} A \mathbf{c} = A^{\mathrm{T}} \mathbf{y} \tag{1}$$

where $A = [a_{ki}]$ is an $n \times (d+1)$ matrix whose general entry is $a_{ki} = x_k^i$, $\mathbf{c} = [c_j]$ is a $(d+1) \times 1$ column vector, and $\mathbf{y} = [y_k]$ is an $n \times 1$ column vector.

The coefficient matrix of the system is

$$A^{\mathrm{T}} A = \begin{bmatrix} \sum_{k=0}^{n-1} 1 & \sum_{k=0}^{n-1} x_k & \cdots & \sum_{k=0}^{n-1} x_k^d \\ \sum_{k=0}^{n-1} x_k & \sum_{k=0}^{n-1} x_k^2 & \cdots & \sum_{k=0}^{n-1} x_k^{d+1} \\ \vdots & \vdots & \ddots & \vdots \\ \sum_{k=0}^{n-1} x_k^d & \sum_{k=0}^{n-1} x_k^{d+1} & \cdots & \sum_{k=0}^{n-1} x_k^{2d} \end{bmatrix}$$

The assumption that the x_k are increasing guarantees that $A^{\mathrm{T}} A$ is invertible, so the coefficients of the polynomial are

$$\mathbf{c} = \left(A^{\mathrm{T}} A \right)^{-1} A^{\mathrm{T}} \mathbf{y}$$

Although this is a concise mathematical formulation, a computer implementation using a fixed-size floating-point number can suffer from numerical problems when inverting the coefficient matrix. The inversion is ill-conditioned when $|x_0| = \min_k |x_k|$ or $|x_{n-1}| = \max_k |x_k|$ is large and the degree d is large.

In some cases, a preconditioning of the data points will suffice to give a reasonably accurate set of coefficients. Map the input points by translating the x-values and then uniformly scale the x- and y-values so that the translated and scaled x-values lie in the interval $[0, 1]$. Such a transformation is

$$(u_k, v_k) = \left(\frac{x_k - x_0}{x_{n-1} - x_0}, \frac{y_k}{x_{n-1} - x_0} \right)$$

The transformed data points are fit with a polynomial

$$v = \sum_{i=0}^{d} c_i u^i$$

using the least-squares method. The conditioning of the matrix A is sometimes better by using the transformation approach, but not all the time. The evaluation of the polynomial at an x value is illustrated by the pseudocode, where the degree, coefficients, and data points are assumed to be globally accessible:

```
float EvaluatePolynomial (float x)
{
    float u = (x - x[0])/(x[n-1] - x[0]);
    float v = c[degree];
    for (int i = degree-1; i >= 0; i--)
    {
        v *= u;
        v += c[i];
    }
    y = (x[n-1] - x[0])*v;
}
```

Another attempt to solve the matrix system uses the Cholesky decomposition. The matrix $A^{\mathrm{T}}A$ is factored into BB^{T}, where B is lower triangular (and B^{T} is upper triangular). The matrix B is then inverted using LU methods to produce $B^{\mathrm{T}}\mathbf{c} = B^{-1}A^{\mathrm{T}}\mathbf{y}$. The matrix B^{T} is similarly inverted to solve for $\mathbf{c} = B^{-\mathrm{T}}B^{-1}A^{\mathrm{T}}\mathbf{y}$. Even this method can suffer from numerical problems with fixed-size floating-point numbers.

Alternatively, a computationally expensive method is to use exact rational arithmetic, where the data points have floating-point components that are exactly represented as rational numbers.

2 Fitting with Orthogonal Polynomials

Another way to view the fitting problem is to use *different* polynomials than $1, x, x^2, \ldots, x^d$, call them $p_i(x)$ for $0 \le i \le d$ with $\text{Degree}(p_i) = i$. The idea is to obtain a set of equations like Equation (1), but where $A^{\mathrm{T}}A$ is a diagonal matrix. In this case, the problems with numerical round-off errors are eliminated (other than the potential ones associated with divisions).

The polynomial we want to fit is of the form

$$y = \sum_{i=0}^{d} c_i p_i(x) \tag{2}$$

The error function to minimize is

$$E(c_0, \ldots, c_d) = \sum_{k=0}^{n} \left(\sum_{i=0}^{d} c_i p_i(x_k) - y_k \right)^2$$

Computing the partial derivatives and setting to zero,

$$0 = \frac{\partial E}{\partial c_j} = 2\sum_{k=0}^{n-1}\left(\sum_{i=0}^{d} c_i p_i(x_k) - y_k\right) p_j(x_k)$$

for $0 \le j \le d$. The equation simplifies to

$$\sum_{i=0}^{d}\left(\sum_{k=0}^{n-1} p_i(x_k)p_j(x_k)\right) c_i = \sum_{k=0}^{n-1} p_j(x_k)y_k$$

If we can choose the polynomials so that

$$\sum_{k=0}^{n-1} p_i(x_k)p_j(x_k) = 0, \quad i \neq j \tag{3}$$

then the set of equations have the easily computed solution

$$c_i = \frac{\sum_{k=0}^{n-1} p_i(x_k)y_k}{\sum_{k=0}^{n-1} p_i^2(x_k)} \tag{4}$$

The condition in Equation (3) is an *orthogonality condition*. A set of polynomials satisfying the condition are said to be *orthogonal polynomials*. Notice that the orthogonality condition depends on the data points themselves. This is different from the standard polynomial fitting where $1, x, \ldots, x^d$ are chosen independently of the input data. Thus, the fitting with orthogonal polynomials may be viewed as a *data-driven method*.

There are a variety of ways to generate orthogonal polynomials. One method is illustrated next. Choose

$$p_0(x) = 1 \tag{5}$$

which is a constant polynomial (degree 0) for all x. Also choose

$$p_1(x) = x - a_1, \quad a_1 = \frac{1}{n}\sum_{k=0}^{n-1} x_k \tag{6}$$

It is easy to verify the orthogonality of the two polynomials:

$$\sum_{k=0}^{n-1} p_0(x_k)p_1(x_k) = \sum_{k=0}^{n-1}(x_k - a_1) = \left(\sum_{k=0}^{n-1} x_k\right) - na_1 = na_1 - na_1 = 0$$

The remaining polynomials are generated by a second-order recurrence relationship,

$$p_{m+1}(x) = (x - a_{m+1})p_m(x) - b_m p_{m-1}(x), \quad m \ge 1 \tag{7}$$

where

$$a_{m+1} = \frac{\sum_{k=0}^{n-1} x_k p_m^2(x_k)}{\sum_{k=0}^{n-1} p_m^2(x_k)}, \quad b_m = \frac{\sum_{k=0}^{n-1} p_m^2(x_k)}{\sum_{k=0}^{n-1} p_{m-1}^2(x_k)}$$

The expressions for a_{m+1} and b_m may be computed from three equations obtained by multiplying Equation (7) by $p_{m-1}(x)$, by $p_m(x)$, and by $p_{m+1}(x)$, and then substituting in x_k and summing over k. In the process of doing so, you must use the orthogonality condition of Equation (3).

The problems of computing the coefficients in Equation (4) and of evaluating the polynomial in Equation (2) both reduce to having to evaluate the orthogonal polynomials of Equations (5), (6), and (7). The pseudocode for this is listed next. It is assumed that a_{m+1} and b_m are globally accessible.

```
float EvaluateOrthogonal (int power, float x)
{
    float p0 = 1;
    if (power == 0)
    {
        return p0;
    }

    float p1 = x - a[1];
    if (power == 1)
    {
        return p1;
    }

    float p2;
    for (int m = 2; m <= power; m++)
    {
        p2 = (x - a[m])*p1 - b[m-1]*p0;
        p0 = p1;
        p1 = p2;
    }
    return p2;
}

void ComputeAB (int n, float x[])
{
    a[1] = 0;
    for (k = 0; k < n; k++)
    {
        a[1] += x[k];
    }
    a[1] /= n;

    for (m = 1; m < degree; m++)
    {
        float sum0 = 0, sum1 = 0, sum2 = 0;
        for (k = 0; k < n; k++)
        {
            float tmp0 = EvaluateOrthogonal(m-1,x[k]);
            float tmp1 = EvaluateOrthogonal(m,x[k]);
            sum0 += tmp0*tmp0;
            sum1 += tmp1*tmp1;
            sum2 += x[k]*tmp1*tmp1;
        }
        a[m+1] = sum2/sum1;
        b[m] = sum1/sum0;
    }
}
```

The function `ComputeAB` should be called first to initialize the a_m and b_m values that are used in general polynomial evaluations.