MCC_2022_W - Project Report

Polynomial regression using ordinary least squares estimation.

Piotr Żeberek | 407663

Introduction

As a result of conducting an experiment we often end up with a large set of data to which we want to assign some kind of dependency. The most basic function expressing connection between independent and dependent variable is polynomial function. Therefore polynomial regression is one of the oldest and most basic form of curve fitting. Clever reader can point out that most relations i.e. in physics are given by more complicated formulas than simple polynomial. However, if this function is infinitely differentiable at some point it can be express at this point as a well known Taylor series which is a perfect example of polynomial function.

The idea of polynomial regression

The problem goes as follows:

> Set of n (x, y) pairs is given. We want to fit mth degree polynomial to this data.

Polynomial of mth degree can by described by $m+1$ coefficients β. For each pair (x, y), following equation can be written setting ϵ as a measure of y's deviation from exact value.

$$y_i = \beta_0 + \beta_1 x_i + \beta_2 x_i^2 + \cdots + \beta_m x_i^m + \epsilon_i \quad \text{where } i = 1, 2, ..., n \tag{1}$$

Expressing x, y and β as vectors

$$\vec{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} \quad \vec{y} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix} \quad \vec{\beta} = \begin{bmatrix} \beta_0 \\ \beta_1 \\ \vdots \\ \beta_m \end{bmatrix} \tag{2}$$

we can write system of linear equations for coefficients β using matrices.

$$\begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ \vdots \\ y_n \end{bmatrix} = \begin{bmatrix} 1 & x_1 & x_1^2 & \cdots & x_1^m \\ 1 & x_2 & x_2^2 & \cdots & x_2^m \\ 1 & x_3 & x_3^2 & \cdots & x_3^m \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_n & x_n^2 & \cdots & x_n^m \end{bmatrix} \begin{bmatrix} \beta_0 \\ \beta_1 \\ \beta_2 \\ \vdots \\ \beta_m \end{bmatrix} + \begin{bmatrix} \epsilon_1 \\ \epsilon_2 \\ \epsilon_3 \\ \vdots \\ \epsilon_n \end{bmatrix} \tag{3}$$

This can be written in a shorter form of

$$\vec{y} = \mathbf{X}\vec{\beta} + \vec{\epsilon} \tag{4}$$

where $\mathbf{X}$ is called Vandermonde matrix.

Taking least squares approach, we want minimize the influence of $\vec{\epsilon}$ by minimizing function $S(\vec{\beta})$ given as

$$S(\vec{\beta}) = \vec{\epsilon}^{\,\mathrm{T}}\vec{\epsilon} = (\vec{y} - \mathbf{X}\vec{\beta})^{\mathrm{T}}(\vec{y} - \mathbf{X}\vec{\beta}) = \|\vec{y} - \mathbf{X}\vec{\beta}\|^2 \tag{5}$$

This minimization problem has a unique solution $\hat{\beta}$, provided that columns of $\mathbf{X}$ matrix are linearly independent.

The solution (estimation for $\vec{\beta}$) is expressed as follows.

$$\hat{\beta} = \left(\mathbf{X}^{\mathrm{T}}\mathbf{X}\right)^{-1}\mathbf{X}^{\mathrm{T}}\vec{y} \tag{6}$$

CUDA implementation

Taking a look at the solution's form we can see that it includes three different kinds of matrices operations: **multiplication**, **transposition** and **inversion**.

Assuming we work with a large set of (x, y) pairs or we want to fit a polynomial of a large degree, we can significantly improve the performance of fitting by implementing algorithms for these operations in CUDA. Additional difficulty is that these algorithms have to work with matrices of any dimensions, not only squares.

Matrix multiplication

At first, a standard naive algorithm was used but it was later replaced with a tiled algorithm utilizing shared memory. Class materials and experience gained during writing of second class report were used.

Transpose of a matrix

Similarly to matrix multiplication, a standard naive algorithm was at some point replaced with a tiled algorithm utilizing shared memory. This was harder than previous algorithm since we did not learn this during classes.

Inverse of a matrix

This was the most complicated algorithm to implement. There are number of ways to take an inverse of a matrix but I decided to use a well know Gauss-Jordan elimination where the inverted matrix is obtained by performing the same set of operation on initial and identity matrices. These operations include row reduction and row subtraction. When all of the operations are finished, initial matrix becomes identity matrix and identity matrix becomes inverted initial matrix. This algorithm is expected to be the most resource consuming and least optimized.

Program execution and output

After compiling the source code using provided Makefile with specified number of SMS, we end up with an executable file called *curveFitting*.

The program has two modes:

- **Demonstration mode** - $\vec{\beta}$ and $\vec{x}$ are generated automatically. The y's vector is then calculated as $\mathbf{X} \cdot \vec{\beta}$ and regression is performed. The regression output $\hat{\beta}$ is expected to be the same as generated $\vec{\beta}$ since no noise is added.
- **File input mode** - (x, y) pairs are read from input file using C++ file stream mechanism. The polynomial's degree is also specified by the user. The regression output $\hat{\beta}$ is then displayed.

For the sake of more user-friendly interface, a simple bash script was written to make running the program easier.

For demo mode, just run below command in terminal.

```
./run.sh
```

For file input mode, script has to be executed as follows.

```
./run.sh <file_name> <polynomial_degree>
```

File input mode demonstration

To demonstrate how the file input mode works, simple *data.txt* file was provided. It contains 1024 (x, y) pairs with noise added to y. This file was generated using a simple C++ program shown below and presenting generating parameters.

```
#include <iostream>
#include <fstream>
#include <cmath>
#include <iomanip>
#include <cstdlib>

int main(){
    double start{-5.0f}, end{5.0f};
    int nSamples{1024};
    int order{8};

    double step = (end - start) / (nSamples - 1);
    double x{0};
    double y{0};

    double coeff[order + 1]{7.4, -3.31, 2.11, 0.78, -0.4, -5, 2.93, -0.001, 0.087};

    std::ofstream file;

    file.open("data.txt");

    srand((unsigned)time(NULL));

    for (int i = 0; i < nSamples; i++){
        x = start + i * step;

        for (int j = 0; j <= order; j++)
            y += coeff[j] * pow(x, j);

        y += (2 * (double)rand() / RAND_MAX - 1) * y * 0.005;

        file << x << std::setprecision(12) << '\t'
        << y << std::setprecision(12) << std::endl;

        y = 0;
    }

    file.close();
}
```

Program output for coefficients (starting with β_0) is:

6.7320 2.2902 2.9573 -1.9802 -0.5843 -4.6989 2.9469 -0.0098 0.0867

As we can see coefficients associated with higher powers of x are very close to exact values. "Lower" coefficients diverge a bit but that is because they are of way less significance than "higher" coefficients. Graphical fitting results (generated with gnuplot) are presented below.

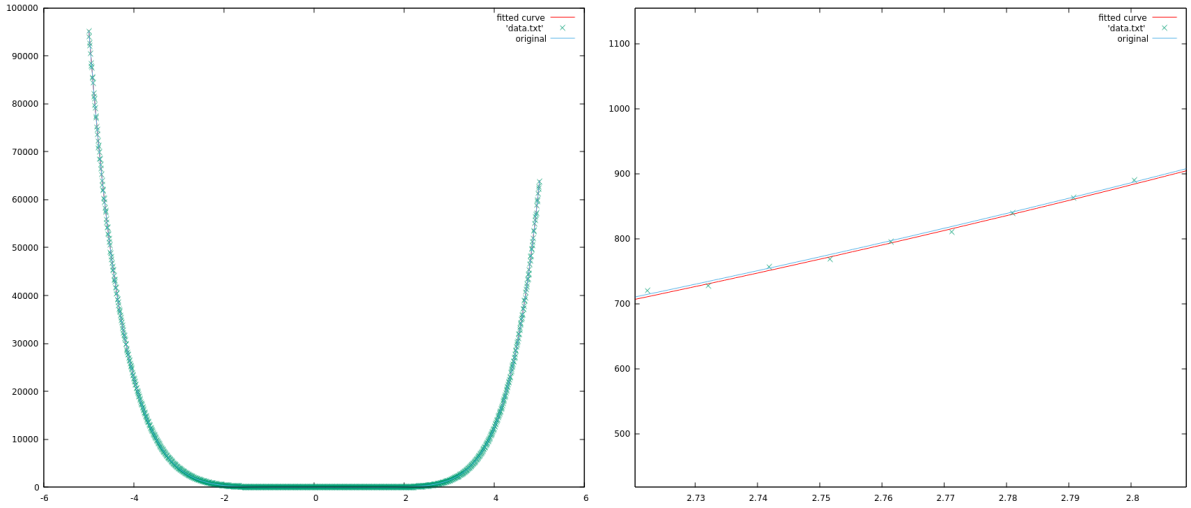

Figure 1: Data points read from file, fitted curve and original one. Zoomed view on the right.

Program limitations and possible improvements

Main limitation of this program is the lost of precision while performing calculation for large degree polynomials. Coefficients start to diverge significantly when doing so. On the contrary, we rarely make use of polynomial regression of i.e. 15th or 20th degree.

Possible improvements to this project are:

- Goodness of fit calculations.
- Dealing with precision problem.
- CPU version for performance comparison.

Summary

Initial idea was to write a program for not only polynomial but any function fitting. It turned out to be too complicated for a graduate student capabilities and time amount he has. Even the polynomial regression was a bit challenging. None the less, program works quite well and can be easily expanded with new features.

GitHub repository: `https://github.com/piotr-zeberek-agh/curveFitting`