COSC 320 - Advanced Data Structures and Algorithm Analysis
Project 1

Dr. Joe Anderson

Due: 13 October 2019

1 Description

You will implement the algorithm discussed in lecture to find the inverse of a matrix, then apply that solution to the problem of linear regression with Ordinary Least Squares.

2 Specifications

Write a `c++` class, `Matrix` (possibly reusing work from Lab 3), to store a matrix of integers. Include in the class methods to perform addition, subtraction, scalar multiplication, transpose, and inverse. Demonstrate the correctness of each of these routines in the program output.

Then, using the operations above, write a program to perform linear regression with Ordinary Least Squares on a set of vectors.

The input for the program will be provided in a data-file, which should be specified via command-line. For example, the command

```
./linearleastsquares myfile.txt
```

should run the program to read the data in `myfile.txt` and store it in a data structure of your choice. The data file will contain vectors of doubles (positive or negative) separated by newlines. There will be one vector per line, and will have values which fit inside a `float` data type. An example (shortened) input file may look like:

```
1.0 1.36353
2.0 0.781852
3.0 8.22072
4.0 26.4652
5.0 17.0021
6.0 23.5086
7.0 21.5244
8.0 29.4199
9.0 28.5146
10.0 13.0951
11.0 26.5581
```

The above is a subset of the data contained in file "`points100.dat`" linked from the course webpage. The linear least squares solution results in $\beta = (1.97942, 4.78616)$ and the graphed solution looks like Figure 1.

You may assume that the input matrix is non-singular.

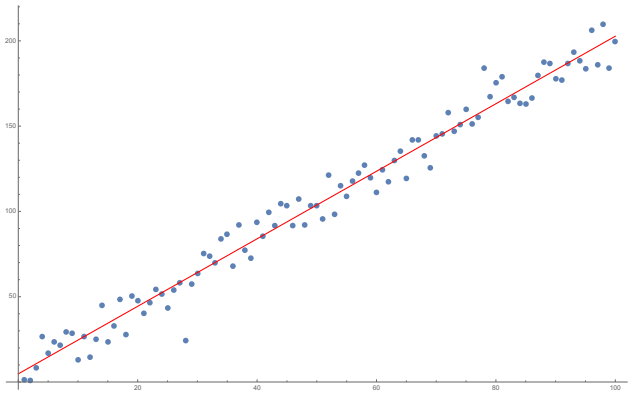

Figure 1: Plotted line of best fit for example file "points100.dat".

2.1 Matrix Inversion

Given an $n \times m$ matrix A, the *inverse* of A is the $m \times n$ matrix A^{-1} so that $AA^{-1} = I_n$, where I_n is the $n \times n$ identity matrix.

For your implementation of inverse, you may assume that A is square $(n \times n)$ and symmetric. To make sure your recursion reaches an acceptable base case, you need to pad the input matrix as

$$A \mapsto \begin{pmatrix} A & 0 \\ 0 & I_k \end{pmatrix}$$

so that $n + k$ is a power of 2. Then note that

$$\begin{pmatrix} A & 0 \\ 0 & I_k \end{pmatrix}^{-1} = \begin{pmatrix} A^{-1} & 0 \\ 0 & I_k \end{pmatrix}.$$

Then, the algorithm to compute A^{-1} is as follows:

1. Divide A into submatrices:
$$A = \begin{pmatrix} B & C^T \\ C & D \end{pmatrix}$$
where B, C, and D are also symmetric and are $(n/2) \times (n/2)$ matrices.
2. Recursively compute B^{-1}.
3. Compute $W = CB^{-1}$ and $W^T = B^{-1}C^T$.
4. Compute $X = WC^T$.
5. Compute $S = D - X$
6. Recursively compute $V = S^{-1}$.
7. Compute $Y = S^{-1}W$ and Y^T.
8. Set $T = -Y^T$ and $U = -Y$.
9. Compute $Z = W^TY$ and set $R = B^{-1} + Z$.
10. Assemble the inverse of A as:
$$A^{-1} = \begin{pmatrix} R & T \\ U & V \end{pmatrix}.$$

Be sure to extract only the top-left $n \times n$ corner, if padding the input matrix was necessary.

Note that the base case of the recursion can be a 1×1 matrix, in which case the inverse is just the 1×1 matrix containing the reciprocal of the single element.

For more information, see the course textbook, section 28.2.

2.2 Least Squares Linear Regression

Given a set of observations $\{x_i = (x_{i,1}, x_{i,2}, x_{i,3}, \ldots, x_{i,m})\}_{i=1}^{n}$ of n points in m dimensions, the problem of linear regression seeks a line that best "approximates" the set of points. In particular, using a variation of the Least Squares method, we re-write the points as the $n \times m$ matrix

$$X = \begin{pmatrix} x_{1,1} & x_{1,2} & \ldots & x_{1,m} \\ x_{2,1} & x_{2,2} & \ldots & x_{2,m} \\ \vdots & & \ddots & \vdots \\ x_{n,1} & x_{n,2} & \ldots & x_{n,m} \end{pmatrix}.$$

Assuming the points come from a noisy linear model (the motivation of "linear" regression), we must modify our mental interpretation to be the following matrix equation:

$$A\beta := \begin{pmatrix} x_{1,1} & x_{1,2} & \ldots & x_{1,m-1} & 1 \\ x_{2,1} & x_{2,2} & \ldots & x_{1,m-1} & 1 \\ \vdots & & \ddots & \vdots & \\ x_{n,1} & x_{n,2} & \ldots & x_{1,m-1} & 1 \end{pmatrix} \begin{pmatrix} \beta_1 \\ \beta_2 \\ \ldots \\ \beta_m \end{pmatrix} = \begin{pmatrix} x_{1,m} \\ x_{2,m} \\ \ldots \\ x_{n,m} \end{pmatrix} =: b$$

where $\beta = (\beta_1, \beta_2, \ldots, \beta_m)$ captures the "independent variable" of each line, and the column of 1's is used to represent the fact that β_m will be a constant added to each equation (the y-intercept in two dimensions). Note that A here is the original matrix of points but with the last column replaced by all 1's; the last column becomes b. Your program should create these two variables, then combine them according to the formula below to calculate $\hat{\beta}$.

It is highly unlikely that such a linear system will have a solution. Instead, we seek to find the vector of m unknowns, $\hat{\beta} = (\beta_1, \beta_2, \ldots, \beta_m)$ that minimizes the squared loss from each of the observed points. This is given by the optimal vector, $\hat{\beta}$

$$\hat{\beta} = \arg\min_{\beta \in \mathbb{R}^m} \sum_{i=1}^{n} |x_{i,m} - \sum_{j=1}^{m} x_{i,j}\beta_j|^2.$$

In other words, the goal is to calculate the "slope" of the line (more precisely, the plane) in $m-1$ dimensional space that most closely matches the given set off points, according to the squared-error model.

Fortunately, the above optimization has a closed form solution that only requires matrix inversion, multiplication, and transposition:

$$\hat{\beta} = (A^T A)^{-1} A^T b.$$

Note here that $A^T A$ is guaranteed to be symmetric and square, simplifying the inversion significantly. Once that is computed, the line of best fit is given by $f(x_1, x_2, \ldots, x_{m-1}) = \beta_m + \sum_{i=1}^{m-1} \beta_i x_i$.

See Section 28.3 of the CLRS textbook for an alternative presentation of the method, in a slightly more general framework. In fact, you can re-formulate the above to calculate the best-fit "shape" for a much broader class of shapes, including, for instance, polynomials!

Important caveats to consider:

1. The above algorithm description is obviously not fully detailed. Be sure to watch for edge cases, take into account matrix operation compatibility, matrix dimensions, and the assumptions needed by the algorithm.

2. **You may not use the `c++` standard library data structures or algorithms.** If you want to use linked lists or related data structures, you should implement a specialized version for this project from scratch. If you are in doubt about whether importing specific outside code is acceptable, consult the instructor.

2.3 Output

Your project should include two executables: one to run the linear least squares procedure and output the optimal vector, $\hat{\beta}$, and one executable to demonstrate correct of the matrix subroutines. Use this to test your program thoroughly!

Demonstrate that the various matrix operations are correct with examples. Include different tests of the linear least squares algorithm. Vary the size of the input and make a note of how the running time changes with relation to the input size.

Record both the time and number of operations performed by the algorithm; count an operation as being any single arithmetic operation on one of the data elements (comparison, addition, multiplication, assignment, etc.).

Include a `Makefile` to compile the code into an executable called `project1`. Be sure to fully document your code.

3 Submission

Submit a `.zip` file called `Project1[LastName].zip` (with `[LastName]` replaced with your own last name) containing your source code, `Makefile`, and documentation, then upload it to the course MyClasses submission page.

Second, print out your code and documentation, to be turned in on the due date (typically at the start of the following Lab).

4 Bonus

If any bonus are completed, be sure to note it in the README file and provide appropriate output to demonstrate its correctness.

1. Everybody's code will be tested and bench-marked rigorously against a large set of input files. The top three programs in terms of efficiency *across all test cases* will receive 10, 5, and 2 bonus points, respectively. You will not have access to the full set of testing data before submission, so test thoroughly, and pay careful attention to time– and space–saving coding techniques!

2. (5 pts) Add a test to your `Matrix` class to test if a given matrix is non-singular.

3. (10 pts) Add a test to verify that a matrix is symmetric, and add a case to invert a non-symmetric square matrix by computing $A^{-1} = (A^T A)^{-1} A^T$, where $A^T A$ is guaranteed to be symmetric.

4. (20 pts) Add an implementation of Strassen's algorithm for matrix multiplication. Include tests and thorough benchmarks (similar to a lab report) to compare it to the basic method using the definition.