CS 211: Computer Architecture, Fall 2018
Programming Assignment 2: One-Shot Learning and Sudoku
(100 points + 25 Extra credit points)
Due Date: October 23th, 5pm

Instructor: Prof. Santosh Nagarakatte

This assignment is designed to provide you some experience writing programs with the C programming language. There are three parts to this assignment. The third part is extra credit.

Part 1: One-Shot Learning (50 points)

In the first part, you will write a C program that implements simple "one-shot" machine learning algorithm for predicting house prices in your area.

There is significant hype and excitement around artificial intelligence (AI) and machine learning. CS 211 students will get a glimpse of AI/ML by implementing a simple machine learning algorithm to predict house prices based on historical data.

For example, the price of the house (y) can depend on certain attributes of the house: number of bedrooms (x_1), total size of the house (x_2), number of baths (x_3), and the year the house was built (x_4). Then, the price of the house can be computed by the following equation:

$$y = w_0 + w_1.x_1 + w_2.x_2 + w_3.x_3 + w_4.x_4 \quad (1)$$

Given a house, we know the attributes of the house (*i.e.*, x_1, x_2, x_3, x_4). However, we don't know the weights for these attributes: w_0, w_1, w_2, w_3 and w_4. *The goal of the machine learning algorithm in our context is to learn the weights for the attributes of the house from lots of training data.*

Let's say we have N examples in your training data set that provide the values of the attributes and the price. Let's say there are K attributes. We can represent the attributes from all the examples in the training data as a Nx$(K+1)$ matrix as follows, which we call X:

$$\begin{bmatrix} 1, & x_{0,1}, & x_{0,2}, & x_{0,3}, & x_{0,4} \\ 1, & x_{1,1}, & x_{1,2}, & x_{1,3}, & x_{1,4} \\ 1, & x_{2,1}, & x_{2,2}, & x_{2,3}, & x_{2,4} \\ 1, & x_{3,1}, & x_{3,2}, & x_{3,3}, & x_{3,4} \\ .. & & & & \\ 1, & x_{n,1}, & x_{n,2}, & x_{n,3}, & x_{n,4} \end{bmatrix}$$

where n is $N-1$. We can represent the prices of the house from the examples in the training data as a Nx1 matrix, which we call Y.

$$\begin{bmatrix} y_0 \\ y_1 \\ .. \\ y_n \end{bmatrix}$$

Similarly, we can represent the weights for each attribute as a $(K+1)$x1 matrix, which we call W.

[
w_0
w_1
..
w_k
]

The goal of our machine learning algorithm is to learn this matrix from the training data.

Now in the matrix notation, entire learning process can be represented by the following equation, where X, Y, and W are matrices as described above.

$$X.W = Y \tag{2}$$

Using the training data, we can learn the weights using the below equation:

$$W = (X^T.X)^{-1}.X^T.Y \tag{3}$$

where X^T is the transpose of the matrix X, $(X^T.X)^{-1}$ is the inverse of the matrix $X^T.X$.

Your main task in this part to implement a program to read the training data and learn the weights for each of the attributes. You have to implement functions to multiply matrices, transpose matrices, and compute the inverses of the matrix. You will use the learned weights to predict the house prices for the examples in the test data set.

Want to learn more about One-shot Learning? The theory behind this learning is not important for the purposes of this class. The algorithm you are implementing is known as linear regression with least square error as the error measure. The matrix $((X^T.X)^{-1}.X^T)$ is also known as the pseudo-inverse of matrix X. If you are curious, you can learn more about this algorithm at `https://www.youtube.com/watch?v=FIbVs5GbBlQ&hd=1`.

Computing the Inverse using Gauss-Jordan Elimination

To compute the weights above, your program has to compute the inverse of matrix. There are numerous methods to compute the inverse of a matrix. We want you to implement a **specific method for computing the inverse of a matrix known as Guass-Jordan elimination**, which is described below. If you compute inverse using any other method, you will risk losing all points for this part.

An inverse of a square matrix A is another square matrix B, such that $A.B = B.A = I$, where I is the identity matrix.

Gauss-Jordan Elimination for computing inverses

Below, we give a sketch of Gauss-Jordan elimination method. Given a matrix A whose inverse needs to be computed, you create a new matrix A_{aug}, which is called the augmented matrix of A, by concatenating identity matrix with A as shown below.

Let say matrix A, whose inverse you want to compute is shown below:

```
[
  1  2  4
  1  6  7
  1  3  2
]
```

The augmented matrix (A_{aug}) of A is:

```
[
  1   2   4   1   0   0
  1   6   7   0   1   0
  1   3   2   0   0   1
]
```

The augmented matrix essentially has the original matrix and the identity matrix. Next, we perform row operations on the augmented matrix so that the original matrix part of the augmented matrix turns into an identity matrix.

The valid row operations to compute the inverse (for this assignment) are:

- You can divide the entire row by a constant
- You can subtract a row by another row
- You can subtract a row by another row multiplied by a constant

However, you are not allowed to swap the rows. In the traditional Gauss-Jordan elimination method you are allowed to swap the rows. For simplicity, we do not allow you to swap the rows.

Let's see this method with the above augmented matrix A_{aug}.

- Our goal is to transform A part of the augmented matrix into an identity matrix.

  Since $A_{aug}[1][0]! = 0$, we will subtract the first row from the second row because we want to make $A_{aug}[1][0] = 0$. Hence, we perform the operation $R_1 = R_1 - R_0$, where R_1 and R_0 represents the second and first row of the augmented matrix. Augmented matrix A_{aug} after $R_1 = R_1 - R_0$

  $$\begin{bmatrix} 1 & 2 & 4 & 1 & 0 & 0 \\ 0 & 4 & 3 & -1 & 1 & 0 \\ 1 & 3 & 2 & 0 & 0 & 1 \end{bmatrix}$$

- Now we want to make $A_{aug}[1][1] = 1$. Hence, we perform the operation $R_1 = R_1/4$. The augmented matrix A_{aug} after $R_1 = R_1/4$ is:

  $$\begin{bmatrix} 1 & 2 & 4 & 1 & 0 & 0 \\ 0 & 1 & \frac{3}{4} & \frac{-1}{4} & \frac{1}{4} & 0 \\ 1 & 3 & 2 & 0 & 0 & 1 \end{bmatrix}$$

- Next, we want to make $A_{aug}[2][0] = 0$. Hence, we perform the operation $R_2 = R_2 - R_0$. The augmented matrix A_{aug} after $R_2 = R_2 - R_0$ is:

  $$\begin{bmatrix} 1 & 2 & 4 & 1 & 0 & 0 \\ 0 & 1 & \frac{3}{4} & \frac{-1}{4} & \frac{1}{4} & 0 \\ 0 & 1 & -2 & -1 & 0 & 1 \end{bmatrix}$$

- Next, we want to make $A_{aug}[2][1] = 0$. Hence, we perform the operation $R_2 = R_2 - R_1$. The augmented matrix A_{aug} after $R_2 = R_2 - R_1$ is:

  $$\begin{bmatrix} 1 & 2 & 4 & 1 & 0 & 0 \\ 0 & 1 & \frac{3}{4} & \frac{-1}{4} & \frac{1}{4} & 0 \\ 0 & 0 & \frac{-11}{4} & \frac{-3}{4} & \frac{-1}{4} & 1 \end{bmatrix}$$

- Now, we want to make $A_{aug}[2,2] = 1$, Hence, we perform the operation $R_3 = R_3 * \frac{-4}{11}$. Then, A_{aug} is:

  $$\begin{bmatrix} 1 & 2 & 4 & 1 & 0 & 0 \\ 0 & 1 & \frac{3}{4} & \frac{-1}{4} & \frac{1}{4} & 0 \\ 0 & 0 & 1 & \frac{3}{11} & \frac{1}{11} & \frac{-4}{11} \end{bmatrix}$$

- Next, we want to make $A_{aug}[1,2] = 0$, Hence, we perform the operation $R_1 = R_1 - \frac{3}{4} * R_2$. Then, A_{aug} is:

$$\begin{bmatrix} 1 & 2 & 4 & 1 & 0 & 0 \\ 0 & 1 & 0 & \frac{-5}{11} & \frac{2}{11} & \frac{3}{11} \\ 0 & 0 & 1 & \frac{3}{11} & \frac{1}{11} & \frac{-4}{11} \end{bmatrix}$$

- Next, we want to make $A_{aug}[0,2] = 0$, Hence, we perform the operation $R_0 = R_0 - 4 * R_2$. Then, A_{aug} is:

$$\begin{bmatrix} 1 & 2 & 0 & \frac{1}{11} & \frac{-4}{11} & \frac{16}{11} \\ 0 & 1 & 0 & \frac{-5}{11} & \frac{2}{11} & \frac{3}{11} \\ 0 & 0 & 1 & \frac{3}{11} & \frac{1}{11} & \frac{-4}{11} \end{bmatrix}$$

- Next, we want to make $A_{aug}[0,1] = 0$, Hence, we perform the operation $R_0 = R_0 - 2 * R_1$. Then, A_{aug} is:

$$\begin{bmatrix} 1 & 0 & 0 & \frac{9}{11} & \frac{-8}{11} & \frac{10}{11} \\ 0 & 1 & 0 & \frac{-5}{11} & \frac{2}{11} & \frac{3}{11} \\ 0 & 0 & 1 & \frac{3}{11} & \frac{1}{11} & \frac{-4}{11} \end{bmatrix}$$

- At this time, the A part of the augmented matrix is an identity matrix. Hence, the inverse of A matrix is:

$$\begin{bmatrix} \frac{9}{11} & \frac{-8}{11} & \frac{10}{11} \\ \frac{-5}{11} & \frac{2}{11} & \frac{3}{11} \\ \frac{3}{11} & \frac{1}{11} & \frac{-4}{11} \end{bmatrix}$$

Your goal is to write a program to compute the inverse of a matrix to perform one-shot learning.

Input/Output specification

Usage interface

Your program for this part will be executed as follows:

```
./first <train-data-file-name> <test-data-file-name>
```

where `<train-data-file-name>` is the name of the training data file with attributes and price of the house. You can assume that the training data file will exist and that it is well structured. The `<test-data-file-name>` is the name of the test data file with attributes of the house. You have to predict the price of the house for each entry in the test data file.

Input specification

The input to the program will be a training data file and a test data file.

Structure of the training data file

The first line in the training file will be an integer that provides the number of attributes (K) in the training set. The second line in the training data file will be an integer (N) providing the number of training examples in the training data set. The next N lines represent the N training examples. Each line for the example will be a list of comma-separated `double precision` floating point values. The first K double precision values represent the values for the attributes of the house. The last double precision value in the line represents the price of the house.

An example training data file (train1.txt) is shown below:

```
4
7
3.000000,1.000000,1180.000000,1955.000000,221900.000000
3.000000,2.250000,2570.000000,1951.000000,538000.000000
2.000000,1.000000,770.000000,1933.000000,180000.000000
4.000000,3.000000,1960.000000,1965.000000,604000.000000
3.000000,2.000000,1680.000000,1987.000000,510000.000000
4.000000,4.500000,5420.000000,2001.000000,1230000.000000
3.000000,2.250000,1715.000000,1995.000000,257500.000000
```

In the example above, there are 4 attributes and 7 training data examples. Each example has values for the attributes and last value is the price of the house. To illustrate, consider the training example below

```
3.000000,1.000000,1180.000000,1955.000000,221900.000000
```

The first attribute has value 3.000000, the second attribute has value 1.000000, third attribute has value 1180.000000, and the fourth attribute has value 1955.000000. The price of the house for these set of attributes is provided as the last value in the line: 221900.000000

Structure of the test data file

The first line in the training file will be an integer (M) that provides the number of test data points in the file. Each line will have K attributes. The value of K is defined in the training data file. Your goal is predict the price of house for each line in the test data file. The next M lines represent the M test points for which you have to predict the price of the house. Each line will be a list of comma-separated double precision floating point values. There will be K double precision values that represent the values for the attributes of the house.

An example test data file (test1.txt) is shown below:

```
2
3.000000,2.500000,3560.000000,1965.000000
2.000000,1.000000,1160.000000,1942.000000
```

It indicates that you have to predict the price of the house using your training data for 2 houses. The attributes of each house is listed in the subsequent lines.

Output specification

Your program should print the price of the house for each line in the test data file. Your program should not produce any additional output. If the price of the house is a fractional value, then your program should round it to the nearest integer, which you can accomplish with the following printf statement:

```
printf("%0.0lf\n", value);
```

where value is the price of the house and its type is `double` in C.

Your program should predict the price of the entry in the test data file by substituting the attributes and the weights (learned from the training data set) in Equation (1).

A sample output of the execution when you execute your program as shown below,

```
./first train1.txt test1.txt
```

should be

```
737861
203060
```

Part 2: Sudoku (50 points)

In Part 2 of this assignment, you will write a C program that implements a simple Sudoku solver (in the third part you can implement a complex Sudoku solver for extra credit). Sudoku is a simple logic puzzle where the objective is to fill a 9x9 grid of cells with each cell containing a number from the set 1-9 while fulfilling the following constraints:

- Each number is unique in its row and column.
- Each number is unique in its subgrid where a subgrid is defined by cutting the 9x9 into 9 non-overlapping 3x3 grids (top left, top, top right, left, middle, right, bottom left, bottom, bottom right).
- Each row, column, and subgrid have all numbers from 1-9 present.

A Sudoku is defined as solved when all of its cells in the 9x9 grid are filled with numbers with each cell satisfying the constraints above. A Sudoku is defined as unsolvable when there is no configuration where all the cells can be filled without breaking the constraints. A Sudoku will start partially completed with some numbers placed in the 9x9 grid. This will allow the solver to determine where to place new numbers in order to find the solution for the given Sudoku. For this part, we will only be looking at Sudoku grids with one unique solution.

In this part, we will look at Sudoku grids wehre there will always be a cell that you can fill with 100% certanity with an unique according to the constraints described above.

Part 3: Extra Credit (25 points)

It is not always possible to fill at least one element with 100% certainity in each step for many Sudoku grids. Sometimes the next step to be taken cannot be known and instead, a guess must be taken in order to move forward, such as perhaps filling in a cell randomly. For extra credit on this assignment, you can implement an algorithm that solves these more complex Sudoku grids. A possible algorithm to do so would be to solve as much of the Sudoku as can be done so confidently. Once no more nodes can be filled with 100% certainty, continue by taking the first unsolved cell (let's call this cell A) and filling it with a number that satisfies the constraints and continue solving as usual from there. If a solution is found, then you are done. Otherwise, if no moves can be taken and the board has not yet been filled than you must backtrack to the state where you guessed the value of cell A and either guess a new value for cell A or perhaps choose a new cell altogether. This is a simple backtracking algorithm. There are many algorithms to solve these complex Sudoku and it is entirely up to you to decide which to implement.

Input format for Part 2 and Part 3

Your program will read in the data from a file given to the program as an argument from the command line. The file format will be a 9x9 grid with each number in the same row separated by a tab. Blank cells will be represented with a '-'.

An example Sudoku grid test1.text is given below:

```
-  -  -  5  8  2  -  -  -
-  -  5  -  -  -  2  -  -
9  2  -  1  -  4  -  -  5
5  -  8  -  -  1  7  -  -
-  3  -  -  -  -  -  2  -
-  -  9  7  -  -  1  -  3
4  -  -  9  -  5  -  1  6
-  -  1  -  -  -  9  -  -
-  9  -  2  1  -  -  -  -
```

A sample Sudoku grid test2.txt is given below:

```
1  -  3  -  -  -  8  4  1
-  -  -  -  -  -  5  -  2
2  -  -  -  -  -  6  9  3
-  -  -  3  -  9  -  -  4
-  -  -  4  -  -  -  -  5
-  -  -  -  -  -  3  -  6
-  -  -  -  -  -  -  -  7
-  -  -  -  -  -  -  -  8
3  2  -  4  5  6  7  8 -
```

You can assume that all inputs will be valid meaning the spacing will be correct and only the '-' character and numbers 1-9 will be present in the test files. The preset numbers in the test may not satisfy the constraints of Sudoku as in the example test2.txt above resulting in an unsolvable Sudoku.

Output format:

For a Sudoku grid that has a solution, you should print out the 9x9 grid with each number in the same row separated by a tab and with each row on a new line. There should be no trailing spaces or tabs on any line of output.

A sample execution is given below:

```
./second test1.txt
```

```
3  1  4  5  8  2  6  7  9
7  8  5  6  9  3  2  4  1
9  2  6  1  7  4  3  8  5
5  6  8  3  2  1  7  9  4
1  3  7  4  6  9  5  2  8
2  4  9  7  5  8  1  6  3
4  7  2  9  3  5  8  1  6
6  5  1  8  4  7  9  3  2
8  9  3  2  1  6  4  5  7
```

For a Sudoku grid that has no solution, your program should print "no-solution" and nothing else. For part 2, Sudoku grids that have a solution but cannot be found without guesses are considered unsolvable.

Structure of your submission folder

All files must be included in the pa2 folder. The pa2 directory in your tar file must contain 2 directories or 3 directories (if decide to do the extra credit part). The name of the directories should be named first through second or first through third (in lower case). Each directory should contain a c source file, a header file(if you use it), and a Makefile. For example, the subdirectory first will contain, first.c, first.h (if you create one), and Makefile (the names are case sensitive).

Hints and suggestions

- You are allowed to use functions from standard libraries but you cannot use third-party libraries downloaded from the Internet (or from anywhere else). If you are unsure whether you can use something, ask us.

- We will compile and test your program on the iLab machines so you should make sure that your program compiles and runs correctly on these machines. You must compile all C code using the gcc compiler with the `-Wall -Werror -fsanitize=address` flags.
- You should test your program with the autograder provided with the assignment.

Submission

You have to e-submit the assignment using Sakai. Your submission should be a tar file named `pa2.tar`. To create this file, put everything (three folders) that you are submitting into a directory named `pa2`. Then, `cd` into the directory containing `pa2` (that is, `pa2`'s parent directory) and run the following command:

```
tar cvf pa2.tar pa2
```

To check that you have correctly created the tar file, you should copy it (`pa2.tar`) into an empty directory and run the following command:

```
tar xvf pa2.tar
```

This should create a directory named `pa2` in the (previously) empty directory.

Grading guidelines

The grading will be automatically graded using the autograder.

Automated grading phase

This phase will be based on programmatic checking of your program using the autograder. We will build a binary using the Makefile and source code that you submit, and then test the binary for correct functionality and efficiency against a set of inputs.

- We should be able build your program by just running `make`.
- Your program should follow the format specified above for both both the parts.
- Your program should **strictly** follow the input and output specifications mentioned. **Note: This is perhaps the most important guideline: failing to follow it might result in you losing all or most of your points for this assignment. Make sure your program's output format is *exactly* as specified. Any deviation will cause the automated grader to mark your output as "incorrect". REQUESTS FOR RE-EVALUATIONS OF PROGRAMS REJECTED DUE TO IMPROPER FORMAT WILL NOT BE ENTERTAINED.**
- **We will check all solutions pair-wise from all sections of this course to detect cheating using `moss` software and related tools**. If two submissions are found to be similar, they will instantly be awarded zero points and reported to office of student conduct. See Rutgers CS's academic integrity policy at: `https://www.cs.rutgers.edu/academic-integrity/introduction`.

Autograder

We provide the AutoGrader to test your assignment. AutoGrader is provided as pa2_autograder.tar. Executing the following command will create the pa2_autograder folder.

```
tar xvf pa2_autograder.tar
```

There are two modes available for testing your assignment with the PA2 AutoGrader.

First mode

Testing when you are writing code with a `pa2` folder

1. Lets say you have a `pa2` folder with the directory structure as described in the assignment.
2. Copy the folder to the directory of the pa2_autograder
3. Run the pa2_autograder with the following command
   python pa2_autograder.py
   It will run the test cases and print your scores.

Second mode

This mode is to test your final submission (i.e, pa2.tar)

1. Copy pa2.tar to the pa2_autograder directory
2. Run the pa2_autograder.py with pa2.tar as the argument. The command line is:

   python pa2_autograder.py pa2.tar