Linear Regression

Instructions:

1. Please join our class forum. I recommend using a pseudonymn, rather than your real name, so you will not be embarrassed about asking basic questions. Make sure you explicitly choose to receive e-mail for each post. (Every semester, several students are surprised to discover that their settings were configured to not send them e-mail. A class forum is not very helpful if you ask a question and replies come weeks after the assignmnet deadline.)

2. Visit the main class page. Click on the link to view your scores. Change your password, so you can view your scores in this class. (Initially, your password is the empty string.) If your name does not appear on the list, please e-mail the instructor. (This may happen, for example, if you enroll after the instructor sets it all up.)

3. Download the starter kit for this assignment in C++ or Java.

   You should be comfortable with the following programming topics in whatever language you decide to use:
   - classes and objects,
   - inheritance,
   - initializing member variables,
   - working with vectors of double-precision floating point values,
   - passing by reference as opposed to passing by value, and
   - debugging.

   It is fine with me if you prefer to use some other programming language, but you will be responsible to find or implement components that replace those in my starter kit. Here are some thinks you will need:
   - You will need some data structure for matrices and vectors.
   - You will need to be able to load a matrix from ARFF format. (It is a simple format--just comma-separated-values with a small amount of meta data in the header--so you could easily implement this yourself. You would not need to implement support for strings or dates or sparse formats.)
   - You will need some way to draw random values from a Normal distribution. (Most languages provide this. If you don't have it, you could easily translate my code that does it.)
   - You will need some code to perform convolution of tensors in any number of dimensions.

   If you go to the bother to translate my starter kit, you are welcome to contribute it for the benefit of future classes. If you plan to use another language, please let me know, so I can ensure that my submission server knows how to build and execute code in the language you plan to use. Note that in the past, students who have used Python have discovered that it was much (as in orders of magnitude) slower at performing floating point

operations than they assumed. (No, using numpy does not solve this problem.) This matters because some of our projects require significant amounts (like 40 minutes) of computation, and it took days in Python.

4. Brush up on your linear algebra skills. Make sure you know how to do the following operations:
   - Sum two vectors,
   - Normalize a vector,
   - Compute the inner product (a.k.a. dot product) of two vectors,
   - Compute the outer product of two vectors,
   - Transpose a matrix,
   - Multiply two matrices,
   - Multiply a matrix by a vector.

   For a quick review of these concepts, read Section 5.1 of this book.

5. Add a class named **Layer** to your project. (This is not the simplest possible design for implementing linear regression. It is designed to prepare you for future assignments. So for now, just go with it.) Give this class a member variables named "activation" that can hold a vector of double-precision floating point values. Also, add an abstract method named "activate" that takes two vectors as paramets, as shown in the examples below.

   C++ example:

```
class Layer
{
	Vec activation;

public:
	Layer(size_t inputs, size_t outputs) :
	  activation(outputs)
	{
	}

	virtual ~Layer()
	{
	}

	virtual void activate(const Vec& weights, const Vec& x) = 0;
};
```

   Java example:

```
abstract class Layer
{
	protected Vec activation;

	Layer(int inputs, int outputs)
	{
		activation = new Vec(outputs);
	}

	abstract void activate(Vec weights, Vec x);
}
```

   Note that my instructions are only intended to help you get started. As you begin to

understand this code, it should become yours, and you should feel free to modify it to suit your preferences, coding styles, etc.

6. Add a class named **LayerLinear** that inherits from (a.k.a. extends) your Layer class. Implement the "activate" method to compute the linear equation activation=Mx+b, where x is a vector of size "inputs", and "activation" is a vector of size "outputs". M is a matrix with "outputs" rows and "inputs" columns. b is a vector of size "outputs". The vector "weights" is a big vector containing all of the values needed to fill both M and b. That is, the number of elements in "weights" will be ("outputs" + ("outputs" * "inputs")).

   The following examples show how to chop up the "weights" vector into sub-vectors:

   C++:

   ```
   VecWrapper b(weights, 0, outputs);
   ```

   Java:

   ```
   Vec b = new Vec(weights, 0, outputs);
   ```

   (Note that you will control the values in the "weights" vector, so it does not matter which elements you draw from it to obtain the values for "b" or "M", as long as you are consistent.)

7. Unit testing is an important way to make sure that a portion of code works as expected. No feature is complete until it has been tested. Here is a simple unit test that you can use to make sure your code works:

   ```
   Let x = [0
            1
            2]

   Let M = [1 2 3
            2 1 0]

   Let b = [1
            5]

   Then

   Mx+b = [9
           6]
   ```

8. Add a method named "ordinary_least_squares" to your LayerLinear class that computes "weights" using the equations at the end of Section 4.2.1.2 of this book. This method should accept 3 parameters: a matrix named "X", a matrix named "Y", and an uninitialized vector named "weights" to which the results will be written.

9. Add a unit test for your "ordinary_least_squares" method. It should work like this:
   1. Generate some random weights.
   2. Generate a random feature matrix, X.
   3. Use your LinearLayer.activate to compute a corresponding label matrix, Y.

4. Add a little random noise to Y.
5. Withhold the weights. (That is, don't use them in the next step.)
6. Call your ordinary_least_squares method to generate new weights.
7. Test whether the computed weights are close enough to the original weights. If they differ by too much, throw an exception.

10. Make a new class named "NeuralNet", which holds a collection of "Layer" objects. (For this assignment, it will only hold one layer, but we will add more in future assignments.) Also add a member variable of type Vec to hold the weights of the layers. Make the NeuralNet class inherit from the SupervisedLearner class. Add a method to your NeuralNet class named "predict" that takes a vector of inputs as its parameter, and calls "Layer.activate". Add a method named "train" that uses Ordinary Least Squares to train the weights.

11. Add a method to the SupervisedLearner class that computes sum-squared-error. (Section 2.1.7 covers sum-squared-error.) Note that the SupervisedLearner class already contains a method that counts misclassification. Your method for computing sum-squared-error will be very similar.

12. Add a method to the SupervisedLearner class that performs m-repetitions of n-fold cross-validation. (See Section 2.1.7.2.) Note that the Matrix.copyBlock method may be helpful.

13. Download these features and labels. The labels give the median values of homes in Boston. The features give various statistics about the homes. (See the comments at the top of the file for more details, if you are curious.) You can use the Matrix.loadArff method to load these into data structures. Write code to perform 5 repetitions of 10-fold cross-validation, then print the root-mean-squared-error to the console.

14. Zip up your code. (You may use a tarball if you prefer. Do not use RAR.) Include the datasets because your code needs them to work. Do not include the datasets that came with the starter kit. (Those were just for validating the starter kit, and are not needed for your assignment.) Include the "build.bash" script. (My server will run this to build your code. Don't make it execute your code.) Do not include any generated binary files. Specifically, do not include any files with any of the following extensions: .exe, .class, .suo, .sdf, .ipch, .ncb, .pdb, .o, .obj, .dll, .so. (My server may be configured to reject your submission if you include one of those.) The link to submit your archive is found on the main class page.

FAQ

1. **Q: I get a C++ compiler error about "fenenableexcept" not being found. What should I do?**
   A: Are you building on Windows? You need to add a "WINDOWS" compiler definition to tell it which code is inteded for Windows. Alternatively, you could replace this code

```
#ifdef WINDOWS
        unsigned int cw = _control87(0, 0) & MCW_EM;
        cw &= ~(_EM_INVALID | _EM_ZERODIVIDE | _EM_OVERFLOW);
        _control87(cw,MCW_EM);
#else
        feenableexcept(FE_INVALID | FE_DIVBYZERO | FE_OVERFLOW);
#endif
```

   with this

```
        unsigned int cw = _control87(0, 0) & MCW_EM;
        cw &= ~(_EM_INVALID | _EM_ZERODIVIDE | _EM_OVERFLOW);
        _control87(cw,MCW_EM);
```

2. **Q: I get an error about "alloca" not being found. What should I do?**
   A: Try changing it to "_alloca". (If that works, please let me know.)

3. **Q: I get an error about "ssize_t" not being found. What should I do?**
   A: Try adding this line: "#define ssize_t int". (A better solution is to find the right header file to include for that. If you figure out what it is, please let me know so I can add it to my starter kit.)

4. **Q: I get an error about "M_PI" not being defined. What should I do?**
   A: Try adding these lines:

```
#define M_PI 3.14159265358979323846
#define M_E  2.7182818284590452354
```

   (A better solution is to find the right header file to include for that. If you figure out what it is, please let me know so I can add it to my starter kit.)

5. **Q: Can you give us a simple cross-validation example for sanity-checking purposes?**
   A: Sure.

```
Here is a simple cross-validation example with the Baseline algorithm:

Here is some data:

Features   Labels
--------   ------
0          2
0          4
0          6

We will do 3-fold cross-validation, so there will be
exactly 1 data point in each fold.

Fold 1:
```

```
        Trains on:
            0 -> 4
            0 -> 6
        Average label:
                5
        Tests on:
            0 -> 2
            Error = -3
            Squared error = 9

    Fold 2:
            Trains on:
            0 -> 2
            0 -> 6
        Average label:
                4
        Tests on:
            0 -> 4
            Error = 0
            Squared error = 0

    Fold 3:
            Trains on:
            0 -> 2
            0 -> 4
        Average label:
                3
        Tests on
            0 -> 6
            Error = 3
            Squared error = 9

    Sum-squared error = 18
    Mean-squared error = 6
    Root-mean-squared error = sqrt(6)
```