CS 457/557: Assignment 1

Due: September 30, 2022, by 11:00 PM (Central)

Overview

In this assignment, you will write a program that can be used to explore the impact of model and algorithm hyperparameters for linear regression. In doing so, you will need to implement the mini-batch gradient descent algorithm for multiple linear regression and polynomial regression (excluding interaction terms) and k-fold cross-validation to aid in tuning model hyperparameters.

Academic Integrity Policy

All work for this assignment must be your own work and it must be completed **independently**. Using code from other students and/or online sources (e.g., Github) constitutes academic misconduct, as does sharing your code with others either directly or indirectly (e.g., by posting it online). Academic misconduct is a violation of the UWL Student Honor Code and is unacceptable. Plagiarism or cheating in any form may result in a zero on this assignment, a **negative score** on this assignment, failure of the course, and/or additional sanctions. Refer to the course syllabus for additional details on academic misconduct.

You should be able to complete the assignment using only the course notes and textbook along with relevant programming language documentation (e.g., the Java API specification). Use of additional resources is discouraged but not prohibited, provided that this is limited to high-level queries and not assignment-specific concepts. As a concrete example, searching for "how to use a HashMap in Java" is fine, but searching for "gradient descent in Java" **is not**.

Deliverables

You should submit a single compressed archive (either `.zip` or `.tgz` format) containing the following to Canvas:

1. The complete source code for your program. I prefer that you use Java in your implementation, but if you would like to use another language, check with me before you get started. Additional source code requirements are listed below:
   - **Your name must be included in a header comment at the top of each source code file.**
   - Your code should follow proper software engineering principles for the chosen language, including meaningful comments and appropriate code style.
   - Your code must not make use of any non-standard or third-party libraries.
2. A `README` text file that provides instructions for how your program can be compiled (as needed) and run from the command line. **If your program is incomplete, then your `README` should document what parts of the program are and are not working.**

Program Requirements

The general outline of the program is given below:

Algorithm 1 Program Flow

```
Process command-line arguments
Load full data set from file
Split full data set into $k$ folds
for each degree $d$ do
    for each fold $F$ do
        Fit a polynomial of degree $d$ to all data not in $F$
        Estimate validation error of fitted model on fold $F$
    Compute average validation error across the folds
```

The program should be runnable from the command line, and it should be able to process command-line arguments to update various program parameters as needed. The command-line arguments are listed below:

- `-f <FILENAME>`: Reads training data from the file named `<FILENAME>` (specified as a `String`)
- `-k <INTEGER>`: Specifies the number of folds for k-fold cross-validation; default is `5`; using option `-k 1` trains a model on the full data set (see the **Output** section for more details)
- `-d <INTEGER>`: Specifies the smallest polynomial degree to evaluate; default is `1`
- `-D <INTEGER>`: Specifies the largest polynomial degree to evaluate; if not specified, then only evaluate one degree (the degree value specified through the `-d` flag or its default value)
- `-a <DOUBLE>`: Specifies the learning rate in mini-batch gradient descent; default is `0.005`
- `-e <INTEGER>`: Specifies the epoch limit in mini-batch gradient descent; default is `10000`
- `-m <INTEGER>`: Specifies the batch size in mini-batch gradient descent; default is `0`, which should be interpreted as specifying full batch gradient descent
- `-v <INTEGER>`: Specifies a verbosity level, indicating how much output the program should produce; default is `1` (see the **Output** section for details)

The `-f <FILENAME>` option is required; all others are optional. You can assume that your program will only be run with valid arguments (so you do not need to include error checking, though it may be helpful for your own testing). Several example runs of the program are shown at the end of this document.

File Format

The data for your program will be specified in files with lines corresponding to individual examples or data points. Each line in the file will either be a comment, which is indicated by a `#` character at the start of the line, or a record containing the attribute (feature) value(s) and output (target) value of one data point, with values separated by spaces. Example data with two input attributes is shown below:

```
# A simple data set
-0.665034835482 -0.790864319331 1.1002453122
0.272860499087 0.412951452974 9.86951233201
-0.936827710349 0.872424492487 8.6164081852
-0.896057432697 0.0825926706021 4.87709404056
0.418121038902 0.741938247492 11.9406390167
```

Several example input files are provided in the `data.zip` archive on Canvas. **Some** of these input files use a particular naming convention to make it easier (for humans) to identify their data set properties. These names have the form `sample-pA-dB.txt`, where `A` indicates the number of attributes in the data set and `B` indicates the degree of the true function f mapping inputs to outputs. Each of these files also has an associated `sample-pA-dB-no-noise.txt` variant which excludes noise from the output terms. These may be helpful in testing your gradient descent implementation. **DO NOT ASSUME** that every input file will follow this naming convention! In particular, your program should **not** rely on a file's name to determine the number of attributes in the data set.

Fitting the Training Data

In order to fit a polynomial of degree $d > 1$ to a given set of examples (the training set), you will first need to **augment** your training data with additional derived attributes (features) corresponding to the *univariate* higher-order terms (you do **not** need to include interaction terms involving two or more attributes). It is also recommended that you include a zeroth attribute with value 1 for each example in the training data.

For example, to fit a degree 2 polynomial to data that contains three raw attributes labeled X_1, X_2, and X_3, you need to create derived features representing X_1^2, X_2^2, and X_3^2. The values for these derived attributes are just the squared values of the original attributes. An example of this transformation combined with augmented zeroth attribute having value 1 is shown below:

$$\mathbf{x} = (5, 2, 7) \text{ gets transformed to } \mathbf{x}' = (1, 5, 2, 7, 25, 4, 49).$$

The augmented data can then be used in a multiple linear regression context.

Next, you will need to implement the mini-batch gradient descent algorithm for multiple linear regression from scratch. In your implementation, you should not use any vectorized operations or linear algebra libraries (e.g., if you write your program in Python, then you shouldn't use the `numpy` library). The reason for this requirement is because it is important to see all of the details behind the scenes that are required to make gradient descent work. The goal is to find weights that minimize the mean squared error across the training set (i.e., your cost function is the average ℓ_2 loss).

Pseudocode for the mini-batch gradient descent algorithm (adapted from Lecture 02-4, page 25) is shown in Algorithm 2. In mini-batch gradient descent, an **iteration** consists of one set of weight updates using a single mini-batch of data points. An **epoch** consists of one full pass through the training data, meaning that each data point is used once for a weight update. Your implementation should be reasonably efficient and avoid unnecessary work, though it may be helpful to deal with efficiency considerations only after you have a working implementation.

The stopping conditions that you should implement are:

- the number of epochs reaches the specified epoch limit (which defaults to 10000);
- after an epoch, the current cost is less than 10^{-10};
- the change in cost from the start of an epoch to its end is less than 10^{-10}.

Algorithm 2 Mini-Batch Gradient Descent

```
procedure MiniBatchGradientDescent( $\{(\mathbf{x}_i, y_i)\}_{i=1}^{n}$, $\alpha$, $m$ )
            ▷ $\{(\mathbf{x}_i, y_i)\}_{i=1}^{n}$ is the training set with $p+1$ attributes, with zeroth attribute $x_{i,0} = 1$
    $w_0^{(0)}, w_1^{(0)}, \ldots, w_p^{(0)} \leftarrow 0$
    $t \leftarrow 0$                    ▷ $t$ is the iteration counter for number of weight updates
    $e \leftarrow 0$    ▷ $e$ counts number of **epochs**; an epoch is one full pass through the training data
    while stopping conditions not met do
        Divide $\{1, 2, \ldots, n\}$ into random mini-batches of $m$ data points each
            ▷ If $m$ does not evenly divide $n$, then the last mini-batch should have size $n$ mod $m$
        for each mini-batch $B$ do
            for each $k \in \{0, 1, 2, \ldots, p\}$ do
                $w_k^{(t+1)} \leftarrow w_k^{(t)} - \alpha \left[ \frac{1}{|B|} \sum_{i \in B} -2x_{ik} \left( y_i - \sum_{j=0}^{p} w_j^{(t)} x_{ij} \right) \right]$
            $t \leftarrow t + 1$
        $e \leftarrow e + 1$                ▷ an epoch ends once we process all mini-batches
    return $\left( w_0^{(t)}, w_1^{(t)}, \ldots, w_p^{(t)} \right)$
```

Output

The level of output produced by the program is controlled by the `-v` flag (for *verbosity*). The levels range from 1 to 4, with 4 containing the most output; the output produced is additive, meaning that the output for level i should also be produced for any level j with $j \geq i$. The desired output for each level is discussed in more detail below.

Output Level 1

Level 1 output prints a summary of the training and validation errors on each fold during cross-validation, along with the averages of these errors across the folds, for each possible degree. The example below shows default behavior of the program with only the `-f` flag specified:

```
shell$ java Driver -f data/sample-p1-d1.txt
Using 5-fold cross-validation.
----------------------------------
* Using model of degree 1
  * Training on all data except Fold 1:
    Training and validation errors:      0.009622      0.009387

  * Training on all data except Fold 2:
    Training and validation errors:      0.009167      0.011206

  * Training on all data except Fold 3:
    Training and validation errors:      0.009561      0.009636

  * Training on all data except Fold 4:
    Training and validation errors:      0.009596      0.009545

  * Training on all data except Fold 5:
    Training and validation errors:      0.009898      0.008322
  * Average errors across the folds:     0.009569      0.009619

```

The example below tests all degrees from 2 (specified with the `-d` flag) up to 3 (specified with the `-D` flag) using 2-fold cross-validation (specified with the `-k` flag):

```
shell$ java Driver -f data/sample-p1-d2.txt -d 2 -D 3 -k 2
Using 2-fold cross-validation.
----------------------------------
* Using model of degree 2
  * Training on all data except Fold 1:
    Training and validation errors:      0.008739      0.010415

  * Training on all data except Fold 2:
    Training and validation errors:      0.010400      0.008758

  * Average errors across the folds:     0.009569      0.009587

----------------------------------
* Using model of degree 3
  * Training on all data except Fold 1:
    Training and validation errors:      0.010477      0.012037

  * Training on all data except Fold 2:
    Training and validation errors:      0.012412      0.010953

  * Average errors across the folds:     0.011444      0.011495

```

Output Level 2

At output level 2, additional information about the training process is displayed for each model that is learned (fit). This includes:

- the total time to train (fit) the model;
- the number of epochs and iterations used;
- the average time per iteration;
- the stopping condition for gradient descent;
- the final model and weights that were identified;
- the training error and validation error (part of level 1 output).

The example below uses 2 folds (`-k 2`), a batch size of 250 for mini-batch gradient descent (`-m 250`), and examines only polynomials of degree 2 (`-d 2`), with a verbosity level of 2 (`-v 2`):

```
shell$ java Driver -f data/sample-p1-d1.txt -k 2 -m 250 -d 2 -v 2
Using 2-fold cross-validation.
----------------------------------
* Using model of degree 2
  * Training on all data except Fold 1:
    Training took 155ms, 4538 epochs, 9076 iterations (0.0171ms / iteration)
    GD Stop condition: DeltaCost ~= 0
    Model: Y = 7.0018 + 3.0075 X1 + 0.0011 X1^2
    Training and validation errors:      0.008739      0.010418

  * Training on all data except Fold 2:
    Training took 100ms, 4392 epochs, 8784 iterations (0.0114ms / iteration)
    GD Stop condition: DeltaCost ~= 0
    Model: Y = 7.0022 + 3.0005 X1 - 0.0023 X1^2
    Training and validation errors:      0.010399      0.008755

  * Average errors across the folds:     0.009569      0.009587
```

(You can use `System.currentTimeMillis()` to get the current time as a `long` type in Java.)

Output Level 3

Output level 3 adds information about the gradient descent search process itself. Specifically, during gradient descent, the current cost is printed every 1000 epochs, and also in the last epoch before gradient descent stops. The example below shows tests using 2 folds, a degree-2 polynomial, and a batch size of 100:

```
shell$ java Driver -f data/sample-p1-d1.txt -d 2 -k 2 -m 100 -v 3
Using 2-fold cross-validation.
----------------------------------
* Using model of degree 2
  * Training on all data except Fold 1:
    * Beginning mini-batch gradient descent
      (alpha=0.005000, epochLimit=10000, batchSize=100)
      Epoch      0 (iter      0): Cost =    53.203047539
      Epoch   1000 (iter   5000): Cost =     0.008918843
      Epoch   1930 (iter   9650): Cost =     0.008738531
    * Done with fitting!
    Training took 122ms, 1930 epochs, 9650 iterations (0.0126ms / iteration)
    GD Stop condition: DeltaCost ~= 0
    Model: Y = 7.0020 + 3.0075 X1 + 0.0004 X1^2
    Training and validation errors:      0.008739      0.010417

  * Training on all data except Fold 2:
    * Beginning mini-batch gradient descent
      (alpha=0.005000, epochLimit=10000, batchSize=100)
      Epoch      0 (iter      0): Cost =    51.481289913
      Epoch   1000 (iter   5000): Cost =     0.010537188
      Epoch   1860 (iter   9300): Cost =     0.010399144
    * Done with fitting!
    Training took 43ms, 1860 epochs, 9300 iterations (0.0046ms / iteration)
    GD Stop condition: DeltaCost ~= 0
    Model: Y = 7.0024 + 3.0005 X1 - 0.0030 X1^2
    Training and validation errors:      0.010399      0.008756

  * Average errors across the folds:     0.009569      0.009586
```

Output Level 4

Level 4 output also reports the current model (i.e., the value of the weights) after every 1000 epochs of gradient descent.

```
shell$ java Driver -f data/sample-p1-d1.txt -k 2 -v 4
Using 2-fold cross-validation.
----------------------------------
* Using model of degree 1
  * Training on all data except Fold 1:
    * Beginning mini-batch gradient descent
      (alpha=0.005000, epochLimit=10000, batchSize=500)
      Epoch      0 (iter      0): Cost =    53.203047539;   Model: Y = 0.0000 + 0.0000 X1
      Epoch   1000 (iter   1000): Cost =     0.011836950;   Model: Y = 7.0058 + 2.9111 X1
      Epoch   2000 (iter   2000): Cost =     0.008742341;   Model: Y = 7.0023 + 3.0040 X1
      Epoch   2488 (iter   2488): Cost =     0.008738604;   Model: Y = 7.0022 + 3.0068 X1
    * Done with fitting!
    Training took 107ms, 2488 epochs, 2488 iterations (0.0430ms / iteration)
    GD Stop condition: DeltaCost ~= 0
    Model: Y = 7.0022 + 3.0068 X1
    Training and validation errors:      0.008739      0.010414

  * Training on all data except Fold 2:
    * Beginning mini-batch gradient descent
      (alpha=0.005000, epochLimit=10000, batchSize=500)
      Epoch      0 (iter      0): Cost =    51.481289913;   Model: Y = 0.0000 + 0.0000 X1
      Epoch   1000 (iter   1000): Cost =     0.014763683;   Model: Y = 6.9990 + 2.8856 X1
      Epoch   2000 (iter   2000): Cost =     0.010406487;   Model: Y = 7.0014 + 2.9963 X1
      Epoch   2555 (iter   2555): Cost =     0.010400744;   Model: Y = 7.0015 + 2.9999 X1
    * Done with fitting!
    Training took 79ms, 2555 epochs, 2555 iterations (0.0309ms / iteration)
    GD Stop condition: DeltaCost ~= 0
    Model: Y = 7.0015 + 2.9999 X1
    Training and validation errors:      0.010401      0.008758

  * Average errors across the folds:     0.009570      0.009586
```

The output produced by your program does not have to match the above **exactly**, but it should be well-formatted and reasonably easy to read. Additional requirements for displaying numeric quantities include:

- Training and validation errors should be displayed with 6 decimal points.
- Time per iteration and model weights should be displayed with 4 decimal points.
- Costs in gradient descent should be displayed to 9 decimal points.

Additional Output Notes

One additional requirement for output involves corner-case behavior with the `-k` flag. If `-k 1` is specified, then your program should use all of the data for both training and validation (so that the training error and validation error are identical), and it should omit results averaged across the folds. The purpose of this behavior is to make it easy to test a single run of the mini-batch gradient descent algorithm using a single training process. Several examples are shown below with different verbosity levels.

```
shell$ java Driver -f data/sample-p2-d2.txt -k 1 -d 2 -a 0.001 -e 5000 -m 100 -v 1
Skipping cross-validation.
----------------------------------
* Using model of degree 2
  * Training on all data:
    Training and validation errors:      0.009522      0.009522
```

```
shell$ java Driver -f data/sample-p2-d2.txt -k 1 -d 2 -D 4 -a 0.001 -e 5000 -m 100 -v 1
Skipping cross-validation.
----------------------------------
* Using model of degree 2
  * Training on all data:
    Training and validation errors:      0.009522      0.009522

----------------------------------
* Using model of degree 3
  * Training on all data:
    Training and validation errors:      0.016139      0.016139

----------------------------------
* Using model of degree 4
  * Training on all data:
    Training and validation errors:      0.060521      0.060521
```

```
shell$ java Driver -f data/sample-p2-d2.txt -k 1 -d 2 -a 0.001 -e 5000 -m 100 -v 2
Skipping cross-validation.
----------------------------------
* Using model of degree 2
  * Training on all data:
    Training took 462ms, 5000 epochs, 50000 iterations (0.0092ms / iteration)
    GD Stop condition: Epoch Limit
    Model: Y = 4.0007 - 3.0005 X1 + 5.0126 X2 + 8.0046 X1^2 - 1.0005 X2^2
    Training and validation errors:      0.009522      0.009522
```

```
shell$ java Driver -f data/sample-p2-d2.txt -k 1 -d 2 -a 0.001 -e 5000 -m 100 -v 3
Skipping cross-validation.
----------------------------------
* Using model of degree 2
  * Training on all data:
    * Beginning mini-batch gradient descent
      (alpha=0.001000, epochLimit=5000, batchSize=100)
      Epoch      0 (iter      0): Cost =    57.307624116
      Epoch   1000 (iter  10000): Cost =     0.138337151
      Epoch   2000 (iter  20000): Cost =     0.013996070
      Epoch   3000 (iter  30000): Cost =     0.009699722
      Epoch   4000 (iter  40000): Cost =     0.009530082
      Epoch   5000 (iter  50000): Cost =     0.009522456
    * Done with fitting!
    Training took 379ms, 5000 epochs, 50000 iterations (0.0076ms / iteration)
    GD Stop condition: Epoch Limit
    Model: Y = 4.0007 - 3.0005 X1 + 5.0126 X2 + 8.0046 X1^2 - 1.0005 X2^2
    Training and validation errors:      0.009522      0.009522
```

The example below shows the `-k 1` behavior at output level 4 with a data set with one input attribute:

```
shell$ java Driver -f data/sample-p1-d2.txt -k 1 -d 2 -a 0.001 -e 5000 -m 100 -v 4
Skipping cross-validation.
----------------------------------
* Using model of degree 2
  * Training on all data:
    * Beginning mini-batch gradient descent
      (alpha=0.001000, epochLimit=5000, batchSize=100)
      Epoch      0 (iter      0): Cost =   102.233213989;   Model: Y = 0.0000 + 0.0000 X1 + 0.0000 X1^2
      Epoch   1000 (iter  10000): Cost =     0.099697154;   Model: Y = 7.3652 + 2.9837 X1 + 6.9935 X1^2
      Epoch   2000 (iter  20000): Cost =     0.013385503;   Model: Y = 7.0773 + 3.0006 X1 + 7.7910 X1^2
      Epoch   3000 (iter  30000): Cost =     0.009734343;   Model: Y = 7.0181 + 3.0033 X1 + 7.9550 X1^2
      Epoch   4000 (iter  40000): Cost =     0.009579881;   Model: Y = 7.0059 + 3.0039 X1 + 7.9888 X1^2
      Epoch   4971 (iter  49710): Cost =     0.009573375;   Model: Y = 7.0034 + 3.0040 X1 + 7.9956 X1^2
    * Done with fitting!
    Training took 341ms, 4971 epochs, 49710 iterations (0.0069ms / iteration)
    GD Stop condition: DeltaCost ~= 0
    Model: Y = 7.0034 + 3.0040 X1 + 7.9956 X1^2
    Training and validation errors:      0.009573      0.009573
```