CS 457/557: Assignment 1

Due: September 30, 2024, 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
if k > 1 then Split full data set into k folds
for each degree d do
    if k > 1 then
        for each fold F do
            Fit a polynomial of degree d to all data not in F and report training error
            Estimate validation error of fitted model on fold F
        Compute average validation error across the folds
    else
        Fit a polynomial of degree d to all data and report training error
```

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 data from the file named `<FILENAME>` (specified as a `String`)
- `-k <INTEGER>`: Specifies the number of folds for k-fold cross-validation; default is `1`, which should **disable** the use of cross-validation and instead train a model on the full data set (see the **Cross-Validation** section on page 8 for more details on this option)
- `-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
- `-r`: If specified, this flag (which has no arguments) enables randomization of data for splitting the full data set into folds **and** for splitting the training data into mini-batches at the start of each training epoch; if this flag is **not** provided, then the data splitting should be **deterministic** and follow the schemes outlined later in this document (this is to make testing and debugging easier).
- `-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. Filenames may include path information, so do not assume that the files are located in the same directory as your source code. 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). Your program must be able to handle command-line arguments in **any** order (e.g., do **not** assume that the first argument will be `-f`). 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 `a01-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.

Augmenting 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** the 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 (this should be done for all values of d).

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.

Fitting the Training Data

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. 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 02-5) is shown in Algorithm 2. Note that this algorithm assumes that the given training data has already been

augmented as needed to include any higher-order terms, and as such it **does not need to know anything about the degree of the polynomial being fit**.

Algorithm 2 Mini-Batch Gradient Descent

procedure MiniBatchGradientDescent($\{(\mathbf{x}_i, y_i)\}_{i=1}^n$, α, 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 mini-batches of m data points each (**see note below for details)**
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 $\left(w_0^{(t)}, w_1^{(t)}, \ldots, w_p^{(t)}\right)$

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.

For mini-batch construction, if m (the batch size) does not evenly divide n (the size of the training set), then the last mini-batch should have size $n \bmod m$. Additionally, if the `-r` command-line flag is given, these mini-batches should be randomized at the start of each training epoch; otherwise, the first m data points of the training data should be assigned to batch 1, the next m data points to batch 2, and so on (this is to make debugging easier).

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}.

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.

Output without Cross-Validation

The level of output produced by the program is controlled by the `-v` flag (for *verbosity*). The levels range from 1 to 5, with 5 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. (Note that the output changes slightly when cross-validation is used via the `-k` flag; see page 8 for details.)

Output Level 1

Level 1 output prints a summary of the training error 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
Skipping cross-validation.
---------------------------------
* Using model of degree 1
  * Training on all data (1000 examples):
  * Training error:      0.009574
```

The example below tests all degrees from 2 (specified with the `-d` flag) up to 4 (specified with the `-D` flag):

```
shell$ java Driver -f data/sample-p1-d2.txt -d 2 -D 4
Skipping cross-validation.
---------------------------------
* Using model of degree 2
  * Training on all data (1000 examples):
  * Training error:      0.009573

---------------------------------
* Using model of degree 3
  * Training on all data (1000 examples):
  * Training error:      0.011438

---------------------------------
* Using model of degree 4
  * Training on all data (1000 examples):
  * Training error:      0.052017
```

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 example below uses a batch size of 250 for mini-batch gradient descent (`-m 250`), a learning rate of 0.01 (`-a 0.01`), and examines only polynomials of degree 2 (`-d 2`), with a verbosity level

of 2 (`-v 2`):

```
shell$ java Driver -f data/sample-p1-d2.txt -m 250 -d 2 -a 0.01 -v 2
Skipping cross-validation.
---------------------------------
* Using model of degree 2
  * Training on all data (1000 examples):
    * Beginning mini-batch gradient descent
      (alpha=0.010000, epochLimit=10000, batchSize=250)
    * Done with fitting!
      Training took 33ms, 1523 epochs, 6092 iterations (0.0054ms / iteration)
      GD Stop condition: DeltaCost ~= 0
      Model: Y = 7.0029 + 3.0040 X1 + 7.9973 X1^2
  * Training error:      0.009573
```

(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, the cost of the initial model (with all weights set to zero) is printed prior to starting the search. Additionally, during gradient descent, the current cost is printed after every 1000 epochs, and also after the last epoch that gets run (regardless of whether it is a multiple of 1000). The example below uses a batch size of 100, a learning rate of 0.0003, and a verbosity level of 3:

```
shell$ java Driver -f data/sample-p1-d1.txt -m 100 -a 0.0003 -v 3
Skipping cross-validation.
---------------------------------
* Using model of degree 1
  * Training on all data (1000 examples):
    * Beginning mini-batch gradient descent
      (alpha=0.000300, epochLimit=10000, batchSize=100)
      Initial model with zero weights   : Cost =    52.342168726
      After   1000 epochs ( 10000 iter.): Cost =     0.062687788
      After   2000 epochs ( 20000 iter.): Cost =     0.010553233
      After   3000 epochs ( 30000 iter.): Cost =     0.009591781
      After   4000 epochs ( 40000 iter.): Cost =     0.009573951
      After   4651 epochs ( 46510 iter.): Cost =     0.009573640
    * Done with fitting!
      Training took 46ms, 4651 epochs, 46510 iterations (0.0010ms / iteration)
      GD Stop condition: DeltaCost ~= 0
      Model: Y = 7.0019 + 3.0038 X1
  * Training error:      0.009574
```

Output Level 4

Level 4 output also reports the current model (i.e., the values of the weights) whenever the current cost is printed (which happens prior to starting gradient descent, after every 1000 epochs during gradient descent, and after finishing the last epoch of gradient descent):

```
shell$ java Driver -f data/sample-p1-d2.txt -v 4
---------------------------------
* Using model of degree 1
  * Training on all data (1000 examples):
    * Beginning mini-batch gradient descent
      (alpha=0.005000, epochLimit=10000, batchSize=1000)
      Initial model with zero weights   : Cost =   102.233213989;   Model: Y = 0.0000 + 0.0000 X1
      After   1000 epochs (  1000 iter.): Cost =     5.677662980;   Model: Y = 9.6618 + 2.8078 X1
      After   2000 epochs (  2000 iter.): Cost =     5.674310026;   Model: Y = 9.6612 + 2.9047 X1
      After   2852 epochs (  2852 iter.): Cost =     5.674305722;   Model: Y = 9.6612 + 2.9081 X1
    * Done with fitting!
      Training took 26ms, 2852 epochs, 2852 iterations (0.0091ms / iteration)
      GD Stop condition: DeltaCost ~= 0
```

```
    Model: Y = 9.6612 + 2.9081 X1
  * Training error:      5.674306
```

Output Level 5

Level 5 output reports the cost and current model after **every** epoch of gradient descent (instead of just after every 1000 epochs as level 4 does). The example below uses an epoch limit to keep the output manageable (along with modifications to the batch size and learning rate):

```
shell$ java Driver -f data/sample-p2-d2.txt -v 5 -e 8 -m 200 -a 0.05
Skipping cross-validation.
---------------------------------
* Using model of degree 1
  * Training on all data (1000 examples):
    * Beginning mini-batch gradient descent
      (alpha=0.050000, epochLimit=8, batchSize=200)
      Initial model with zero weights   : Cost =    57.307624116;   Model: Y = 0.0000 + 0.0000 X1 + 0.0000 X2
      After      1 epochs (     5 iter.): Cost =    27.961530017;   Model: Y = 2.5812 - 0.4917 X1 + 0.7778 X2
      After      2 epochs (    10 iter.): Cost =    16.499535855;   Model: Y = 4.1058 - 0.9033 X1 + 1.4333 X2
      After      3 epochs (    15 iter.): Cost =    11.635566313;   Model: Y = 5.0064 - 1.2485 X1 + 1.9858 X2
      After      4 epochs (    20 iter.): Cost =     9.326638376;   Model: Y = 5.5385 - 1.5385 X1 + 2.4517 X2
      After      5 epochs (    25 iter.): Cost =     8.087844582;   Model: Y = 5.8530 - 1.7824 X1 + 2.8446 X2
      After      6 epochs (    30 iter.): Cost =     7.348721766;   Model: Y = 6.0389 - 1.9876 X1 + 3.1759 X2
      After      7 epochs (    35 iter.): Cost =     6.873297018;   Model: Y = 6.1488 - 2.1604 X1 + 3.4554 X2
      After      8 epochs (    40 iter.): Cost =     6.553159885;   Model: Y = 6.2139 - 2.3059 X1 + 3.6911 X2
    * Done with fitting!
      Training took 4ms, 8 epochs, 40 iterations (0.1000ms / iteration)
      GD Stop condition: Epoch Limit
      Model: Y = 6.2139 - 2.3059 X1 + 3.6911 X2
  * Training error:      6.553160
```

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.

Additionally, **don't use tabs** (`\t`) in your output! (See here for a discussion of why.)

Cross-Validation

If the `-k` command-line argument with a value greater than 1 is used, then your program should perform k-fold cross-validation to assess the quality of the learned models on data that was not used for training. This will first require that you split the full data set into k folds.

Let s denote the size of the full data set (which gets read from the file). If k evenly divides s, then each fold should have the same size. However, if k does not evenly divide s, then the folds should be as balanced as possible, meaning that any two folds should differ in size by at most one. One way to ensure this is to divide data points from the full data set into folds according to the scheme below:

Fold	Data Points
1	$1,\ k+1,\ 2k+1,\ 3k+1,\ \ldots$
2	$2,\ k+2,\ 2k+2,\ 3k+2,\ \ldots$
3	$3,\ k+3,\ 2k+3,\ 3k+3,\ \ldots$
$\vdots$	
k	$k,\ k+k,\ 2k+k,\ 3k+k,\ \ldots$

That is, each fold j should contain data point j and all data points that are offset from data point j by a multiple of k. (*Hint:* think about how modulus can help you here!)

Regardless of whether k evenly divides s, you should follow the above scheme for splitting data into folds when the `-r` flag is **not specified**, as this will ensure that your folds are the same as mine for testing and debugging purposes. Note that the scheme above **differs** from the mechanism used in mini-batch construction when m, the mini-batch size, does not evenly divide n, the training set size. As such, you will probably want to handle these two approaches differently in your code!

Output with Cross-Validation

With cross-validation, the Level 1 output changes slightly to print a summary of the training and validation errors on each fold, along with the averages of these errors across the folds, for each possible degree. The example below shows the behavior with `-k 4` and `-D 2` (i.e., using 4 folds and testing degree-1 and degree-2 polynomials):

```
shell$ java Driver -f data/sample-p2-d2.txt -k 4 -D 2
Using 4-fold cross-validation.
--------------------------------
* Using model of degree 1
  * Training on all data except Fold 1 (750 examples):
  * Training and validation errors:      5.738617      6.059353

  * Training on all data except Fold 2 (750 examples):
  * Training and validation errors:      5.673204      6.224029

  * Training on all data except Fold 3 (750 examples):
  * Training and validation errors:      5.927985      5.444010

  * Training on all data except Fold 4 (750 examples):
  * Training and validation errors:      5.848045      5.685958

  * Average errors across the folds:     5.796963      5.853337
```

```
---------------------------------
* Using model of degree 2
  * Training on all data except Fold 1 (750 examples):
  * Training and validation errors:      0.009949      0.008272

  * Training on all data except Fold 2 (750 examples):
  * Training and validation errors:      0.009460      0.009985

  * Training on all data except Fold 3 (750 examples):
  * Training and validation errors:      0.009237      0.010393

  * Training on all data except Fold 4 (750 examples):
  * Training and validation errors:      0.009322      0.010314

  * Average errors across the folds:     0.009492      0.009741
```

Increasing the verbosity level simply includes the extra output associated with the training process every time that training is done:

```
shell$ java Driver -f data/sample-p2-d2.txt -k 2 -d 1 -v 4 -m 25
Using 2-fold cross-validation.
---------------------------------
* Using model of degree 1
  * Training on all data except Fold 1 (500 examples):
    * Beginning mini-batch gradient descent
      (alpha=0.005000, epochLimit=10000, batchSize=25)
      Initial model with zero weights   : Cost =    61.355322475;   Model: Y = 0.0000 + 0.0000 X1 + 0.0000 X2
      After    240 epochs (  4800 iter.): Cost =     5.879047160;   Model: Y = 6.4291 - 3.3421 X1 + 4.8354 X2
    * Done with fitting!
      Training took 8ms, 240 epochs, 4800 iterations (0.0017ms / iteration)
      GD Stop condition: DeltaCost ~= 0
      Model: Y = 6.4291 - 3.3421 X1 + 4.8354 X2
  * Training and validation errors:      5.879047      5.803901

  * Training on all data except Fold 2 (500 examples):
    * Beginning mini-batch gradient descent
      (alpha=0.005000, epochLimit=10000, batchSize=25)
      Initial model with zero weights   : Cost =    53.259925756;   Model: Y = 0.0000 + 0.0000 X1 + 0.0000 X2
      After    248 epochs (  4960 iter.): Cost =     5.643722941;   Model: Y = 6.1995 - 2.7807 X1 + 5.0614 X2
    * Done with fitting!
      Training took 4ms, 248 epochs, 4960 iterations (0.0008ms / iteration)
      GD Stop condition: DeltaCost ~= 0
      Model: Y = 6.1995 - 2.7807 X1 + 5.0614 X2
  * Training and validation errors:      5.643723      6.066306

  * Average errors across the folds:     5.761385      5.935104
```