CSCI 4360/6360 Data Science II
Department of Computer Science
University of Georgia

# Assignment 3: Premature Optimization is $\sqrt{\texttt{evil}}$

DUE: Tuesday, October 8 by 11:59:59pm

Out September 24, 2019

## Questions

This homework assignment will explore some evolutionary computing methods, including one we don't explicitly cover in class. It also includes a review on basic linear regression techniques. If you're rusty on linear regression, I would strongly recommend checking out *Elements of Statistical Learning*, chapter 3 (on "Linear Methods for Regression"), the full PDF for which is linked from the course website. Sections 3.1 and 3.2 from that chapter should be sufficient (though some of the fundamentals behind general supervised learning can be found in chapter 2).

### 1 Linear Regression **[25pts]**

Assume we are given $n$ training examples $(\vec{x}_1, y_1), (\vec{x}_2, y_2), ..., (\vec{x}_n, y_n)$, where each data point $\vec{x}_i$ has $m$ real-valued features (i.e., $\vec{x}_i$ is $m$-dimensional). The goal of regression is to learn to predict $y$ from $\vec{x}$, where each $y_i$ is also real-valued (i.e. continuous).

The linear regression model assumes that the output $Y$ is a linear combination of input features $X$ plus noise terms $\epsilon$ from a given distribution, with weights on the input features given by $\beta$.

We can write this in matrix form by stacking the data points $\vec{x}_i$ as rows of a matrix $X$, such that $x_{ij}$ is the $j$-th feature of the $i$-th data point. We can also write $Y$, $\beta$, and $\epsilon$ as column vectors, so that the matrix form of the linear regression model is:

$$Y = X\beta + \epsilon$$

where

$$Y = \begin{bmatrix} y_1 \\ \vdots \\ y_n \end{bmatrix}, \epsilon = \begin{bmatrix} \epsilon_1 \\ \vdots \\ \epsilon_n \end{bmatrix}, \beta = \begin{bmatrix} \beta_1 \\ \vdots \\ \beta_m \end{bmatrix}, \text{and } X = \begin{bmatrix} \vec{x}_1 \\ \vdots \\ \vec{x}_n \end{bmatrix},$$

and where $\vec{x}_i = \begin{bmatrix} x_1, x_2, ..., x_n \end{bmatrix}$.

Linear regression seeks to find the parameter vector $\beta$ that provides the best fit of the above regression model. There are lots of ways to measure the goodness of fit; one criteria is to find the $\beta$ that minimizes the squared-error loss function:

$$J(\beta) = \sum_{i=1}^{n} (y_i - \vec{x}_i^T \beta)^2,$$

or more simply in matrix form:

$$J(\beta) = (X\beta - Y)^T (X\beta - Y), \tag{1}$$

which can be solved directly under certain circumstances:

$$\hat{\beta} = (X^T X)^{-1} X^T Y \tag{2}$$

(recall that the "hat" notation $\hat{\beta}$ is used to denote an *estimate* of a true but unknown–and possibly, unknow*able*–value)

When we throw in the $\epsilon$ error term, assuming it is drawn from independent and identically distributed ("i.i.d.") Gaussians (i.e., $\epsilon_i \sim \mathcal{N}(0, \sigma^2)$), then the above solution is also the MLE estimate for $P(Y|X; \beta)$.

All told, then, we can make predictions $\hat{Y}$ using $\hat{\beta}$ ($X$ could be the training set, or new data altogether):

$$\hat{Y} = X\hat{\beta} + \epsilon$$

Now, when we perform least squares regression, we make certain idealized assumptions about the vector of error terms $\epsilon$, namely that each $\epsilon_i$ is i.i.d. according to $\mathcal{N}(0, \sigma^2)$ for some value of $\sigma$. In practice, these idealized assumptions often don't hold, and when they fail, they can outright implode. An easy example and inherent drawback of Gaussians is that they are sensitive to outliers; as a result, noise with a "heavy tail" (more weight at the ends of the distribution than your usual Gaussian) will pull your regression weights toward it and away from its optimal solution.

In cases where the noise term $\epsilon_i$ can be arbitrarily large, you have a situation where your linear regression needs to be *robust* to outliers. Robust methods start by weighting each observation *unequally*: specifically, observations that produce large residuals are down-weighted.

**[15pts]** In this problem, you will assume $\epsilon_1, ..., \epsilon_n$ are i.i.d. drawn from a Laplace distribution (rather than $\mathcal{N}(0, \sigma^2)$); that is, each $\epsilon_i \sim \text{Lap}(0, b)$, where $\text{Lap}(0, b) = \frac{1}{2b}\exp(-\frac{|\epsilon_i|}{b})$.

Derive the loss function $J_{\text{Lap}}(\beta)$ whose minimization is equivalent to finding the MLE of $\beta$ under the above noise model.

*Hint #1*: Recall the critical point above about the form of the MLE; start by writing out $P(Y_i|X_i; \beta)$.

*Hint #2*: Logarithms nuke pesky terms with exponents without changing linear relationships.

*Hint #3*: Multiplying an equation by -1 will switch from "argmin" to "argmax" and vice versa.

**[10pts]** Why do you think the above model provides a more robust fit to data compared to the standard model assuming the noise terms are distributed as Gaussians? Be specific!

## 2 Particle Swarm Optimization **[35pts]**

Particle Swarm Optimization (PSO) is yet another nature-inspired search algorithm that attempts to strike a balance between *exploration* (conducting fast but low-resolution searches of large parameter spaces) with *exploitation* (refining promising but small areas of the total search space).

Here is a Matlab plot of PSO in action: notice how the majority of agents (dots) very quickly gather in the bottom left corner (exploitation) representing the global minimum, but there are nonetheless a few dots that appear elsewhere on the energy landscape (exploration).

Rather than devote a third homework assignment's coding section to yet another document classification scheme, we'll explore PSO from a more theoretical viewpoint.
PSO was introduced in 1995, and was inspired by the movement of groups of animals: insects, birds, and fish in particular. Virtual particles "swarm" the search space using a directed but stochastic algorithm designed to modulate efforts to find the global extremum (exploitation) while avoiding getting stuck in local extrema (exploration). It is relatively straightforward to implement and easy to parallelize; however, it is slow to converge to global optima, and ultimately cannot guarantee convergence to global optima.

Formally: $N$ particles move around the search space $\mathcal{R}^n$ according to a few very simple rules, which are predicated on:

- each particle's individual best position so far, and
- the overall swarm's best position so far

Each particle $i$ has a position $\vec{x}_i$, a velocity $\vec{v}_i$, and an optimal position so far $\vec{p}_i$, where $\vec{x}_i, \vec{v}_i, \vec{p}_i \in \mathcal{R}^n$.

Globally, there is an optimal swarm-level position $\vec{g} \in \mathcal{R}^n$ (the supremum of all $\vec{p}_i$), cognitive and social parameters $c_1$ and $c_2$, and an inertia factor $\omega$.

The update rule for velocity $\vec{v}_i$ at time $t+1$ is as follows:

$$\vec{v}_i(t+1) = \omega\vec{v}_i(t) + c_1 r_1 \left[\vec{p}_i(t) - \vec{x}_i(t)\right] + c_2 r_2 \left[\vec{g}(t) - \vec{x}_i(t)\right]$$

where $r_1, r_2 \sim U(0,1)^n$.

**[15pts]** Explain the effects of the cognitive ($c_1$) and social ($c_2$) parameters on the particle's velocity. What happens when one or both is small (i.e. close to 0)? What happens when one or both is large? What effects do the random numbers $r_1$ and $r_2$ have? Relate the effects of these parameters to their "nature"-based inspiration, if you can.

**[5pts]** The inertia parameter $\omega$ in this formulation is typically started at 1 and decreased slowly on each iteration of the optimization procedure. Why?

**[10pts]** The update rule for position $\vec{x}_i$ at time $t+1$ is as follows:

$$\vec{x}_i(t+1) = \vec{x}_i(t) + \vec{v}_i(t+1)$$

Given an objective function $f$ that can be evaluated using a position vector $\vec{x}_i(t)$, provide Python-like update statements for the best particle-specific estimate $\vec{p}_i(t+1)$, and the best global, swarm-level estimate $\vec{g}(t+1)$. **Note**: for the sake of consistency, let's assume you're searching for the global *minimum* of $f$.

*Hint*: Remember that particle-specific estimates $\vec{p}_i(t)$ are also position vectors.

**[5pts]** Give a *concrete* example of how the PSO formulation described here could be improved (better global estimate in the same amount of time, faster convergence, tighter global convergence bounds, etc); you don't have to provide a specific implementation, but it should be clear how it would work ("more power", therefore, is not a concrete example). Such formulations are easy to find online; I implore you to resist the urge to search! Please keep it brief; I'll stop reading after 2-3 lines.

## 3 Coding **[40pts]**

In this part, you'll re-implement your logistic regression code from Assignment 1 to use a simple genetic algorithm to learn the weights, instead of gradient descent.

Your script `assignment3.py` should accept the following required arguments:

1. a file containing training data (same as Assignment 1)
2. a file containing training labels (same as Assignment 1)
3. a file containing testing data (same as Assignment 1)

It should also be able to accept the following *optional* arguments:

- `-n`: a population size (default: 200)
- `-s`: a per-generation survival rate (default: 0.3)
- `-m`: a mutation rate (default: 0.05)
- `-g`: a maximum number of generations (default: 50)
- `-r`: a random seed (default: -1)

The handout on AutoLab contains a skeleton script with the command-line parsing ready to go. It also contains subroutines that ingest and parse out the data files into NumPy arrays. You'll use the same dataset as before: the training set for your evolutionary algorithm to learn good weights, and the testing set to evaluate the weights.

Your evolutionary algorithm for learning the weights should have a few core components:

**Random population initialization.** You should initialize a full array of weights *randomly* (don't use all 0s!); this counts as a single "person" in the full population. Consequently, initialize $n$ arrays of weights randomly for your full population. You'll evaluate each of these weights arrays independently and pick the best-performing ones to carry on to the next generation.

**Fitness function.** This is a way of evaluating how "good" your current solution is. Fortunately, we have this already: the objective function! You can use the weights to predict the training labels (as you did during gradient descent); the fitness for a set of weights is then the *average classification accuracy*.

**Reproduction.** Once you've evaluated the fitness of your current population, you'll use that information to evolve the "strongest." You'll first take the top $s$%–the $ns$ arrays of weights with the highest fitness scores–and set them aside as the "parents" of the next

generation. Then, you'll "breed" random pairs of these parents parents to produce "children" until you have $n$ arrays of weights again. The breeding is done by simply averaging the two sets of parent weights together.

**Mutation.** Each individual weight has a mutation rate of $m$. Once you've computed the "child" weight array from two parents, you need to determine where and how many of the elements in the child array will mutate. First, flip a coin that lands on heads (i.e., indicates mutation) with probability $m$ (the mutation rate) for each weight $w_i$. Then, for each mutation, you'll generate the new $w_i$ by sampling from a Gaussian distribution with mean and variance set to be the empirical mean and variance of *all* the $w_i$ weights of the *previous* generation. So if $W_p$ is the $n \times |\beta|$ matrix of the previous population of weights, then we can define $\mu_i = W_p[:, i].\text{mean}()$ and $\sigma_i^2 = W_p[:, i].\text{var}()$. Using these quantities, we can then draw our new weight $w_i \sim \mathcal{N}(\mu_i, \sigma_i^2)$.

**Generations.** You'll run the fitness evaluation, reproduction, and mutation repeatedly for $g$ generations, after which you'll take the set of weights from the final population with the highest fitness and evaluate these weights against the testing dataset.

The parent and child populations should be kept *distinct* during reproduction, and only the children should undergo mutation!

Your script should be able to be invoked as follows:

```
> python assignment3.py train.data train.label test.data
```

with the optional parameters then able to be stated at the end. The data files (`train.data` and `test.data`) contain three numbers on each line:

```
<document_id> <word_id> <count>
```

Each row of the data files contains the count of how often a given word (identified by ID) appears in certain documents (also identified by ID). The corresponding labels for the data has only one number per row in the file: the label, 1 or 0, of the document with ID corresponding to the row of the label in the label file. For example, a 0 on the $27^{th}$ line of the label file means the document with ID 27 has the label 0.

After you've found your final weights and used them to make predictions on the test set, your code should print a predicted label (0 or 1) by itself on a single line, *one for each document*–this means a single line of output per unique document ID (or per line in one of the `.label` files). The output will be used to autograde your GA on AutoLab. For example, if the following `test.data` file has four unique document IDs in it, your program should print out four lines, each with a 1 or 0 on it, e.g.:

```
> python assignment3.py train.data train.label test.data
```

```
0
0
1
1
```

Evolutionary programs **will take longer** than logistic regression's gradient descent. I strongly recommend staying under a population size of 300, with no more than about 300 generations. **Make liberal use of NumPy vectorized programming** to ensure your program is running as efficiently as possible. The AutoLab autograder timeout will be extended to about 10 minutes, but you should be able to get reasonable training performance without having to go even half that long.

## Administration

### 1 Submitting

All submissions will go to **AutoLab**. You can access AutoLab at:

- https://autolab.cs.uga.edu

You can submit deliverables to the **Assignment 3** assessment that is open. When you do, you'll submit two files:

1. `assignment3.py`: the Python script that implements your algorithms, and
2. `assignment3.pdf`: the PDF write-up with any questions that were asked

These should be packaged together in a tarball; the archive can be named whatever you want when you upload it to AutoLab, but the files in the archive should be named **exactly** what is above. Deviating from this convention could result in the autograder failing!

To create the tarball archive to submit, run the following command (on a *nix machine):

```
> tar cvf assignment3.tar assignment3.py assignment3.pdf
```

This will create a new file, `assignment3.tar`, which is basically a zip file containing your Python script and PDF write-up. Upload the archive to AutoLab. There's no penalty for submitting as many times as you need to, but keep in mind that swamping the server at the last minute may result in your submission being missed; AutoLab is programmed to close submissions *promptly* at 11:59pm on October 8, so give yourself plenty of time! A late submission because the server got hammered at the deadline will *not* be acceptable (there is a *small* grace period to account for unusually high load at deadline, but I strongly recommend you avoid the problem altogether and start early).

Also, to save time while you're working on the coding portion, you are welcome to create a tarball archive of just the Python script and upload that to AutoLab. Once you get the autograder score you're looking for, you can then include the PDF in the folder, tarball everything, and upload it. AutoLab stores the entire submission history of every student on every assignment, so your autograder (code) score will be maintained and I can just use your most recent submission to get the PDF.

## 2 Reminders

- If you run into problems, ping the `#questions` room of the Slack chat. If you still run into problems, ask me. But please please please, **do NOT** ask Google to give you the code you seek! I will be on the lookout for this (and already know some of the most popular venues that might have solutions or partial solutions to the questions here).
- Prefabricated solutions (e.g. `scikit-learn`, OpenCV) are NOT allowed! You have to do the coding yourself!
- If you collaborate with anyone, just mention their names in a code comment and/or at the top of your homework writeup.
- Explicitly cite any external, non-course materials referenced.