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

# Homework 4: Premature Optimization is $\sqrt{\texttt{evil}}$

DUE: Tuesday, October 31 by 11:59:59pm

Out October 12, 2023

## Questions

This homework assignment will explore some evolutionary computing methods and features more on unsupervised clustering.

### 1 Particle Swarm Optimization [40pts]

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? 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?

**[5pts]** What effects do the random numbers $r_1$ and $r_2$ have?

**[10pts]** One of the greatest advantages of PSO is that it is highly parallelizable. Throughout the iterative process of moving the particles, evaluating them against the objective function, and updating the identified optima, there is only a single step in the entire algorithm that requires synchronization between parallel processes. What step is that? Be specific!

*Hint*: For those who took the midterm, this aspect of PSO made an appearance!

**[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.

## 2 Hierarchical Clustering [20pts]

We spent a little bit of time in class discussing hierarchical clustering, of which–fun fact–graph-based segmentation algorithms (min-cut, norm-cut, and even spectral clustering variants) are often considered a part. We primarily discussed two variants: *top-down*, in which all data points start out in a single large cluster that is continually split until a stopping criterion is reached, and *bottom-up*, in which all data points start out in their own clusters that are continually merged until a stopping criterion is reached.

Here, we'll explore the latter, also known as *agglomerative clustering*. The basic algorithm is as follows:

1. Start with each point in a cluster of its own
2. Until there is only one cluster
   a) Find closest pair of clusters
   b) Merge them
3. Return the tree of cluster-mergers

To convert this procedure into an implementation, one only needs to be able to quantify how "close" two clusters are. While not mentioned in detail, we did discuss metrics that define distance between two clusters, such as single-link, complete-link, and average-link.

In this problem, you'll look at an alternative approach to quantifying the distance between two disjoint clusters, proposed by Joe H. Ward in 1963. We will call it **Ward's metric**.

Ward's metric simply says that the distance between two disjoint clusters, $X$ and $Y$, is how much the sum of squares will increase when we merge them. More formally:

$$\Delta(X, Y) = \sum_{i \in X \cup Y} ||\vec{x}_i - \vec{\mu}_{X \cup Y}||^2 - \sum_{i \in X} ||\vec{x}_i - \vec{\mu}_X||^2 - \sum_{i \in Y} ||\vec{x}_i - \vec{\mu}_Y||^2$$

where $\vec{\mu}_Z$ is the centroid of cluster $Z$, and $\vec{x}_i$ is a data point in our corpus. Here, $\Delta(X, Y)$ is considered the *merging cost* of combining clusters $X$ and $Y$ into one cluster, $X \cup Y$. That is, on each iteration, the two clusters with the lowest *merging cost* is merged using Ward's metric as a distance measure.

**[15pts]** Can you reduce the formula given for $\Delta(X, Y)$ to a simpler form? Provide the simplified formula and the steps to get there. Your formula should be in terms of the cluster sizes (i.e., the number of points in a given cluster, denoted as $n_X$ and $n_Y$) and the distance $||\vec{\mu}_X - \vec{\mu}_Y||^2$ between cluster centroids $\vec{\mu}_X$ and $\vec{\mu}_Y$ (yes, **only** the $n_X$, $n_Y$, $\vec{\mu}_X$, and $\vec{\mu}_Y$ symbols should be in your final equation).

**[5pts]** Assume you are given two *pairs* of clusters $P_1$ and $P_2$. The centers of the two clusters in the $P_1$ pair are farther apart than the pair of centers in $P_2$. Using Ward's metric, does agglomerative clustering *always* choose to merge the two clusters in $P_2$? Why or why not? Justify your answer with a simple example.

## 3 Coding [40pts]

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

Your script `homework4.py` should accept the following required arguments:

1. a file containing training data (same as Homework 1)
2. a file containing training labels (same as Homework 1)
3. a file containing testing data (same as Homework 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 $\mu_i = W_p\left[:, i\right].\text{mean}()$ and $\sigma_i^2 = W_p\left[:, i\right].\text{var}()$. 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 homework4.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 homework4.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 **Homework 4** assessment that is open. When you do, you'll submit two files:

1. `homework4.py`: the Python script that implements your algorithms, and
2. `homework4.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 homework4.tar homework4.py homework4.pdf
```

This will create a new file, `homework4.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 31, 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 Discord server. 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! But you **can** use the pairwise metrics in scikit-learn, as well as the vector norm in SciPy.
- If you collaborate with anyone or anybot, just mention their names in a code comment and/or at the top of your homework writeup.
- Cite any external and/or non-course materials you referenced in working on this assignment.