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

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

DUE: Tuesday, October 22 by 11:59:59pm

Out October 8, 2019

Questions

This homework assignment will explore some evolutionary computing methods, including one we don't explicitly cover in class but which you saw on the midterm! It also features some introductory concepts on literary theory to prepare us for collaboration with Prof. Elizabeth Davis' Digital Humanities course, and concludes with the same coding problem as in HW3.

1 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 ω.

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 ω 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?

[5pts] 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: You implemented it on the midterm!

[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 Literary Theory **[25pts]**

As part of our collaboration with Prof. Elizabeth Davis' Digital Humanities course, you will soon be placed on teams with students in her class. This will create something of a proto-typical interdisciplinary data science environment: data scientists on one hand (you!), and domain experts on the other (her students). You'll have to work together in order to generate hypotheses, conduct experiments, and answer questions about literature. This is meant as a warm-up exercise.

[5pts] Assume you are given a corpus from Project Gutenberg of 19th century literature. Prof. Davis' students says they are interested in determining whether or not a given piece of literature was written by a male or female. What model would you propose initially to begin answering that question?

[10pts] Using the model you proposed in the previous step, what would a male writer look like under that model, versus a female writer (you're welcome to speculate here, within reason).

[10pts] Not to let mathematical theory take a back seat–an important element of literary analysis (or any NLP task) is to identify common themes. This is often done with an operation like PCA.

Let X be an $N \times d$ matrix of N documents and d words in the corpus. One way to identify topics would be to decompose the corpus matrix into principal components.

Let $Y = X^T$. If the SVD of $Y = U\Sigma V^T$, show that the columns of V are the PCA of X (recall that PCA operates on the covariance Q of X, where $Q = X^T X$).

3 Coding **[40pts]**

In this question, you'll be implementing a slightly simplified version of the MultiRankWalk (MRW) semi-supervised learning algorithm discussed in lecture. The paper is here: https://lti.cs.cmu.edu/sites/default/files/research/reports/2009/cmulti09017.pdf

Yep, this is the same coding question as HW3, verbatim. Literally verbatim. If you already finished it for HW3, just submit it again here (taking care to re-name the files accordingly).

The basic procedure of MRW is similar to other graph-based random walk algorithms such as PageRank. For a graph G defined by the set of vertices V and edges E, the MRW procedure is as follows:

$$\vec{r} = (1-d)\vec{u} + dW\vec{r}$$

where W is the weighted transition matrix of graph G from vertex i to j is given by $W_{ij} = A_{ij}/d_{ii}$, where d_{ii} is the degree of the i^{th} vertex. $\vec{u}$ is the normalized teleportation vector, where $|\vec{u}| = |V|$ and $||\vec{u}||_1 = 1$. d is a constant damping factor, controlling how often random jumps are made.

The value A_{ij} comes from our use of an affinity matrix in representing the graph. **This is a deviation from the MRW paper**, which assumes a simple adjacency matrix. The affinity matrix A will be determined using the radial-basis function kernel, also known as the Gaussian kernel or heat kernel. It has the form $A_{ij} = A_{ji} = e^{-\gamma||\vec{x}_i - \vec{x}_j||^2}$, and is implemented in scikit-learn's `sklearn.metrics.pairwise` module as `rbf_kernel()`. Once you have the affinity matrix A, the diagonal (degree) matrix D can be found by summing the rows of A, i.e. $D_{ii} = \sum_j A_{ij}$. Finally, the weighted transition probability matrix W can be found using A and D and the above formulation.

Your task is to solve for the ranking vector $\vec{r}$ by iteratively substituting $\vec{r}^{t-1}$ with $\vec{r}^{t}$ until convergence or a set number of iterations.

In this implementation, the $\vec{u}$ vector actually functions as a *seed vector*: this identifies vertices that are labeled and function as seeds for the subsequent label-spreading. "Seeds" are labeled data points used to initiate the label-spreading of the MRW algorithm and predict classes for unlabeled data. The original MRW paper cites several methods, including using PageRank to initially rank labeled vertices in terms of preference as seed vertices to MRW. Your code will need to implement both random seed selection, and degree-based seed selection. In the former, you'll randomly pick k labeled data points from each class and use them as seeds. In the latter, you'll rank the labeled vertices of each class by their degree (i.e. sums of the rows of A) and select the top k in each class.

Critically, you will need to perform MRW for **each distinct class c in the data**. Specifically, when initializing the labeled seeds in $\vec{u}$, you need to set each corresponding element $\vec{u}_i = 1$ such that $\vec{y}_i = c$. All other entries of $\vec{u}$ should be 0. Once this step is completed, you will need to normalize $\vec{u}$ such that $||\vec{u}||_1 = 1$. Next, you can proceed with MRW. Finally, you will repeat this process again for all unique labels c in your dataset, so that at the end you'll have a set of ranking vectors $\vec{r}_1, \vec{r}_2, ..., \vec{r}_c$ for each class.

Once you have generated a ranking vector $\vec{r}$ for each class, you'll then assign labels to all your unlabeled data. For the i^{th} vertex, whichever ranking vector $\vec{r}$'s i^{th} element is largest, assign the corresponding class label represented by that ranking vector to the unlabeled data point. Continue for all unlabeled data.

Your code should be able to process: an input file containing the n m-dimensional data points, the number of labeled data points k to use from each class as seeds, whether to choose seeds randomly or by vertex degree, the damping factor d, and an output file to write the predicted classes for all data.

You'll also be provided the boilerplate to read in the necessary command-line parameters:

1. `-i`: a file path to a text file containing the data
2. `-d`: the damping factor (float between 0 and 1)
3. `-k`: number of data points per class to use as seeds
4. `-t`: type of seed selection to use, "random" or "degree"
5. `-e`: the epsilon threshold, or squared difference of $\vec{r}^t$ and $\vec{r}^{t+1}$ to determine convergence
6. `-g`: value of gamma for the pairwise RBF affinity kernel
7. `-o`: a file path to an output file, where the predicted labels will be written

The format of the input file will be tab-delimited, where a single data point will be on one line. The first column will be the labels: any unlabeled data will have a label of -1. Functions are already written in the `assignment4.py-TEMPLATE` file that will handle reading in data and parsing command-line arguments.

The format of the output file should be one label prediction per line; therefore, the number of lines in the input file and the output file should match exactly (so for the labeled data, you can either use the labels you read in from the file or the labels that are predicted from your ranking vectors, though in theory they should be the same). Essentially, fill in the -1 values in your initial label vector, then just write the vector to a text file, such that each element of the vector is on its own line. For your convenience, the ground-truth label files `y_easy.txt` and `y_hard.txt` for the full datasets are provided; you can use these to check how well your code is predicting the -1 labels.

HINT 1: The value of gamma can substantially affect the accuracy of your method. Larger values shrink the neighborhoods and isolate points from each other; smaller values expand the neighborhoods and make everything look the same distance. If in doubt, plot the affinity matrix using `matplotlib.pyplot.imshow`, and you should see a block-diagonal-ish structure. For the easy dataset, try values around 0.5. For the harder dataset, try values in the 10-50 range.

HINT 2: At the same time, adding more seeds per class can help immensely. The default value in the template script is only 1 seeded value per class; while you can still attain high-90s accuracy with proper values of gamma on the hard dataset, it's almost impossible to hit perfect accuracy without increasing the number of seeds.

HINT 3: The two test datasets provided should not require any more than 100 iterations to converge using the default epsilon.

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 4** assessment that is open. When you do, you'll submit two files:

1. `assignment4.py`: the Python script that implements your algorithms, and
2. `assignment4.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 assignment4.tar assignment4.py assignment4.pdf
```

This will create a new file, `assignment4.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 22, 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! But you **can** use the pairwise metrics in scikit-learn, as well as the vector norm in SciPy.
- If you collaborate with anyone, 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.