INFO371 Lab2

Your name:

Deadline: Wed, Jan 31, 10:30am

Introduction

This homework is pretty long. But don't get desperate just yet! It offloads quite a bit complexity from one of your later homeworks (you do quite a bit of work now!), and it largely repeats related questions, you have to modify your code just a little bit.

Please submit a) your code (notebooks, rmd, whatever) and b) the lab in a final output form (html or pdf). Unlike PS1, you are free to choose either R or python for solving this problem set.

Note: it includes questions you may want to answer on paper instead of computer. You are welcome to do it but please include the result as an image into your final file.

Working together is fun and useful but you have to submit your own work. Discussing the solutions and problems with your classmates is all right but do not copy-paste their solution! Please list all your collaborators below:

1.

2. ...

1 Gradient Ascent

1.1 The Algorithm

Gradient Ascent[1] (GA) is a popular method to find maxima (and minima) of functions, widely used in various machine learning applications. Your task here is to write a GA algorithm that can find the maximum of high-dimensional quadratic function.

The idea with GA is the following:

1. Start somewhere: pick an initial value of the parameter $\mathbf{x}^0 = (x_1^0, x_2^0, \ldots, x_n^0)$.

2. Compute the *gradient* of your objective function $f(\cdot)$ at $\mathbf{x}^0$. Gradient, commonly denoted by $\nabla f(\mathbf{x})$, is just a vector of partial derivatives:

$$\nabla f(\mathbf{x}) \equiv \begin{pmatrix} \frac{\partial}{\partial x_1} f(\mathbf{x}) \\ \frac{\partial}{\partial x_2} f(\mathbf{x}) \\ \cdots \\ \frac{\partial}{\partial x_n} f(\mathbf{x}) \end{pmatrix}. \tag{0.1}$$

   Gradient shows the direction of the steepest ascent of the objective function (can you see that?).

   Compute gradient at $\mathbf{x}^0$, $\nabla f(\mathbf{x}^0)$.

[1] Gradient Descent, the "mirror image" of gradient ascent, is perhaps more widely used.

3. Take a step from $\mathbf{x}^0$ in the gradient direction. The step should be neither too long (you may overshoot) nor too short (it takes too long time to find the maximum). Instead, introduce a *hyper-parameter* ρ (usually called *learning rate*) and take the step of length $\rho \cdot \nabla f(\mathbf{x}^0)$. This will land you into a new place we call $\mathbf{x}^1$:

$$\mathbf{x}^1 = x^0 + \rho \cdot \nabla f(\mathbf{x}_0). \tag{0.2}$$

This is our new best bet for the location of maximum.

4. Are we in the correct place? There are several ways to check:

   (a) Gradient is very small. Say, $|\nabla f(\mathbf{x}^1)| < \epsilon^g$, where $|\cdot|$ mean length of the vector, and ϵ^g is a small number, say 10^{-6}. (Why does this indicate that maximum has been found?)

   (b) Function value does not grow much any more. Say, $f(\mathbf{x}^1) - f(\mathbf{x}^0) < \epsilon^f$ where ϵ^f is another small number.

   (c) You may introduce more conditions, for instance about relative size of gradient.

   Such conditions are called *stopping criteria*. In practice, you always need an additional, bail-out criterion: stop if the process has been repeated too many times already.

5. If we are in correct place, stop here. If not, set $\mathbf{x}^0 \leftarrow \mathbf{x}^1$ and repeat from step 2.

1.2 Implement The 1D GA Algorithm

Now it's your turn!

1. Start slow and manually. Let's look at 1D case with $f(x) = -x^2$. Pick an x^0 (but don't pick $x^0 = 0$), and set $\rho = 0.1$.

   (a) What is the correct location of the maximum of this function?

   (b) What is the gradient vector of the function? What is it's dimension?

   (c) Compute the gradient at x^0, $\nabla f(x^0)$.

   (d) Compute $x^1 = x^0 + \rho \nabla f(x^0)$.

   (e) Did we move closer to the maximum?

2. Now repeat the previous exercise on computer. Choose at least one stopping criterion and it's parameter ϵ, and the bail-out number of iterations, R. Implement the above as an algorithm that at each step prints out the x value, the function's value $f(x)$ and gradient $\nabla f(x)$. At the end it should print the solution, and also how many iterations it took for the loop to converge (finish).

3. Experiment with different starting values, learning rates and stopping parameter values. Comment and explain your findings.

Congrats! You have implemented the 1D gradient ascent! This was easy, right?

1.3 Implement The 2D Version

But now we get more serious: we take a 2D quadratic function From now on, we do everything in matrix form because this scales–there is little difference between 2D and 200D problem if your code is written in matrices. In matrix form the quadratic function can be expressed as

$$f(\mathbf{x}) = -\mathbf{x}' A \mathbf{x} \tag{0.3}$$

where $\mathbf{x} = (x_1, x_2)'$ is a 2×1 matrix and A is a 2×2 matrix. We only look at cases where A is symmetric.

4. Write $\mathbf{x} = (x_1, x_2)'$ and $\mathsf{A} = \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix}$. Show that $f(\mathbf{x}) = -\mathbf{x}'\mathsf{A}\mathbf{x}$ describes a quadratic function.

5. What's the location of it's true maximum?

6. What is the condition that the problem has a single maximum, i.e. $f(\mathbf{x}) \leqslant 0$ for all $\mathbf{x}$?

   Hint: Consult Greene Appedix A7, p834 (available in files/readings).

7. Compute the gradient vector of this function. Use the non-vector form you did under 4 above.

   Note: you may not be familiar with vector derivatives, but in vector form

   $$\nabla f(\mathbf{x}) = -(\mathsf{A}' + \mathsf{A})\mathbf{x} \quad \text{or} \quad -2\mathsf{A}\mathbf{x} \qquad \text{as } \mathsf{A} \text{ is symmetric.} \tag{0.4}$$

   Show that this is indeed true for the 2D case.

8. Now code the algorithm above for 2D case. Do it in matrix form! Show that you get the correct solution if you pick $x^0 = (2, -3)'$ and $\mathsf{A} = \begin{pmatrix} 1 & 2 \\ 2 & 8 \end{pmatrix}$.

   As before, experiment with a few different learning rates and see how does this influence the speed of convergence. Comment and explain your findings.

9. Visualize your algorithm's work:

   - mark all the intermittend points (the $\mathbf{x}^1$-s) on the plot and connect these with lines.
     Note: you may want to save, instead of print, these values in your code now.
   - add the function contours on the plot. In R you can use `graphics::contour` or `ggplot2::geom_contour`. In matplotlib `plt.contour` does a similar job.
     Note: try to use fixed 1:1 aspect ratio. Otherwise the direction of gradient does not coincide with that of the steepest ascent on the figure.
   - Explain what you see.

10. Finally, let's try if your code scales to 5D case without any modifications. Take

    $$\mathsf{A} = \begin{pmatrix} 11 & 4 & 7 & 10 & 13 \\ 4 & 17 & 10 & 13 & 16 \\ 7 & 10 & 23 & 16 & 19 \\ 10 & 13 & 16 & 29 & 22 \\ 13 & 16 & 19 & 22 & 35 \end{pmatrix} \tag{0.5}$$

    (generated as $\frac{1}{2}(\mathsf{A}+\mathsf{A}')+10\cdot\mathsf{I}_5$ where A is a 5×5 matrix of sequence $1 \ldots 25$, and I_5 is the corresponding unit matrix.) Take the initial value $\mathbf{x^0} = \mathbf{c}(10, 20, 30, 40, 50)'$. Play with hyperparameters until you achieve convergence! Show and comment your results. In particular I'd like to see your comments comparing the easiness and speed of getting the 1D, 2D and 5D case to converge.

1.4 Condition Numbers

How easy it is to get your algorithm to converge also depends on how close or far is your matrix from being singular. This can be done using *condition number*.

11. Compute the condition number for matrix A in the example above.

    Hint: consult Greene, Appendix 6.6, p 829.

    Note: a) Greene uses a slightly less common definiton of condition number, typically one uses just the fraction, not square root of it as Greene does. b) the words "largest" and "smallest" should be understood as largest and smallest *in absolute value*.

Now let's return to 2D case. Take a simple singular matrix $\mathsf{A} = \begin{pmatrix} 1 & 1 \\ 1 & 1 \end{pmatrix}$.

12. Show, on computer, that it is singular! Show it without error messages but using relevant linear algebra functions/properties instead.

Let's make it non-singular by adding a "certain amount" α of unit matrix to it: $\mathsf{C} = \mathsf{A} + \alpha \cdot \mathsf{I}_2$. Your task is to analyze and visualize the algorithm's work for three cases: $\alpha \in 100, 1, 0.01$.

13. Run your algorithm for each of these three cases. For each α:
    (a) report the hyperparameters
    (b) compute the corresponding C
    (c) print it's eigenvalues and condition number. Please state which definition of eigenvalues you are using if not following Greene!
    (d) solve the problem using GA
    (e) report the location of maximum and the number of iterations it took
    (f) visualize the process using a figure, similar to the 9 above.

    Note: you may have to set different hyperparameters for different α-s.

14. Comment on your results. How is the convergence speed related to the matrix condition number?

2 How Often Do We Get Big Outliers?

The task in this problem is to conduct a series of MC simulations and see how often do we get outliers of given size. How often do we get "statistically significant" results even if there is nothing significant in our model.

You will generate a large number R (1000 is a good choice) of random samples of given size N (100 is a good choice). The samples are calculated in different ways, such as just normal randoms or means of normal randoms. Each time you will check how often it happens that the values in the sample are way off from the typical values.

2.1 The Easy Task: Normal Sample

1. Pick your number of repetitions R and sample size N. 1000 and 100 are good choices.
2. Now generate a sample of N independent standard normal random variables, and find it's mean. It's almost never exactly 0. How big is it in your case?
3. Compute the 95% confidence intervals for your results. Does the value you got fall into 95% confidence region?

   Hint: if you are unfamiliar with normal distribution and confidence intervals, consult "Openintro Statistics", p 127+

2.2 Get Serious (At Least A Little Serious)

Now repeat the previous exercise R times. At each iteration save your mean value, so at the end you have R mean values (call these m).

4. Print mean-of-means (mean of m-s) and it's empirical 95% confidence region based on your experiment.

   Hint: you have to find the region that covers 95% of your m-s around the center of the distribution. The easiest way to find it is using sample quantiles. Check out the `quantile` function in R, or `np.percentile` function in python.

5. Plot the histogram of your m-s and mark this range on it.

6. Find how many m-s (the actual number) fall out of this range.

7. Extra challenge: can you compute the theoretical confidence intervals for your m-s? Compare the theoretical and empirical intervals. How many results fall out of the theoretical ones?

8. A different extra challenge: run this task in parallel on your multicore laptop.

   Note: as the computations are independent of each other, the parallelization is easy. In fact, similar tasks are called "embarrassingly parallel" problems.

9. Now pick two different N-s. And with "different" I mean *different*, like $100\times$ smaller or larger.

   Compute the mean of means, and it's confidence regions with the new N-s. Compare the results for different N-s.

2.3 Get Nasty (Well, Let Distributions Get Nasty Instead)

Normal distribution has very nice properties, despite it's density function looking a little bit . . . nasty.

But there are many things in our world that are not well approximated by normal distribution. Examples include size of cities, links to webpages, popularity of actors, number of citations of researchers, size of wildfires. . . . All of these are very inequal–there are cities that are enourmous, but most of towns are small. Some researchers have hundreds of thousands of citations but most of them only a few. This kind of outcomes can be described by *Pareto distribution*. There are many ways to define it, but let's do it like this:

$$F(x) = 1 - \left(\frac{x}{x_0}\right)^{-k} \quad \text{and} \quad f(x) = k x_0^k x^{-(k+1)} \qquad \text{where } x \geqslant x_0 \text{ and } k > 0. \tag{0.1}$$

The parameter x_0 (called *location*) tells where the distribution "begins" and k describes how fast it falls for large x values (called *shape*). The expected value for Pareto random variable is

$$\mathbb{E}\, X = \frac{k}{k-1} x_0 \qquad \text{for } k > 1. \tag{0.2}$$

Note that for $k \leqslant 1$ the expected value does not exist! Your task is to explore what the property means in practice.

Set $x_0 = 1$ and pick a $k < 1$. Note: depending on the software you use, it may parameterized differently and you may have to adjust the parameter choice accordingly. If you use `np.random.pareto` (python) or `VGAM::rpareto` (R), the parameterization is as described here.

10. Create a large number of pareto random variables. Make histogram of the results using *log-log scale*! Pick the number in such a fashion that the results look good.

    Comment the shape of the histogram.

    Suggestion: to make the histogram look good, only include values that are less than a limit, say 100.

Pick three different N-s (and I mean *different*. 10, 1000, 1,000,000 are good choices). For each N do the following:

11. Do like in 2.2 – create N pareto random variables. Calculate their mean value. Repeat the process R times and save the results in $\mathfrak{m}$.

12. print the mean and the corresponding 95% empirical confidence intervals (calculated in the same way as above).

13. Compare your findings to those of normal distribution. In particular, I'd like to hear how does the width of the confidence intervals depend on N.