DATA 609 - Homework 1: Introduction to Optimization and Least Squares

Eddie Xu

Instructions

Please submit a .qmd file along with a rendered pdf to the Brightspace page for this assignment. You may use whatever language you like within your qmd file, I recommend python, julia, or R.

Problem 1: Gradient Descent

(a) Consider the mathematical function defined on $f : \mathbb{R}^2 \to \mathbb{R}$:

$$f(x, y) = (x - 1)^2 + (y + 2)^2,$$

Find the single critical point of this function and show that it is a local minimum (in this case, this will also be a global minimum).

(b) Now consider a new objective function that depends on a parameter b:

$$f(x, y) = x^2 + by^2$$

Here we will look at two different values of b, $b = 3$ and $b = 10$. The global minimum of this function occurs at the point $x^* = 0$, $y^* = 0$ no matter what the value of b. Suppose that we didn't know this and wanted to find the minimum of this function using gradient descent instead of direct calculation.

- First write code to perform the gradient descent algorithm, that is perform the iteration:

$$\mathbf{v}_{n+1} = \mathbf{v}_n - k\nabla f(\mathbf{v}_n),$$

where the vector $\mathbf{v} = \begin{bmatrix} x & y \end{bmatrix}^T$ and k is the learning rate.

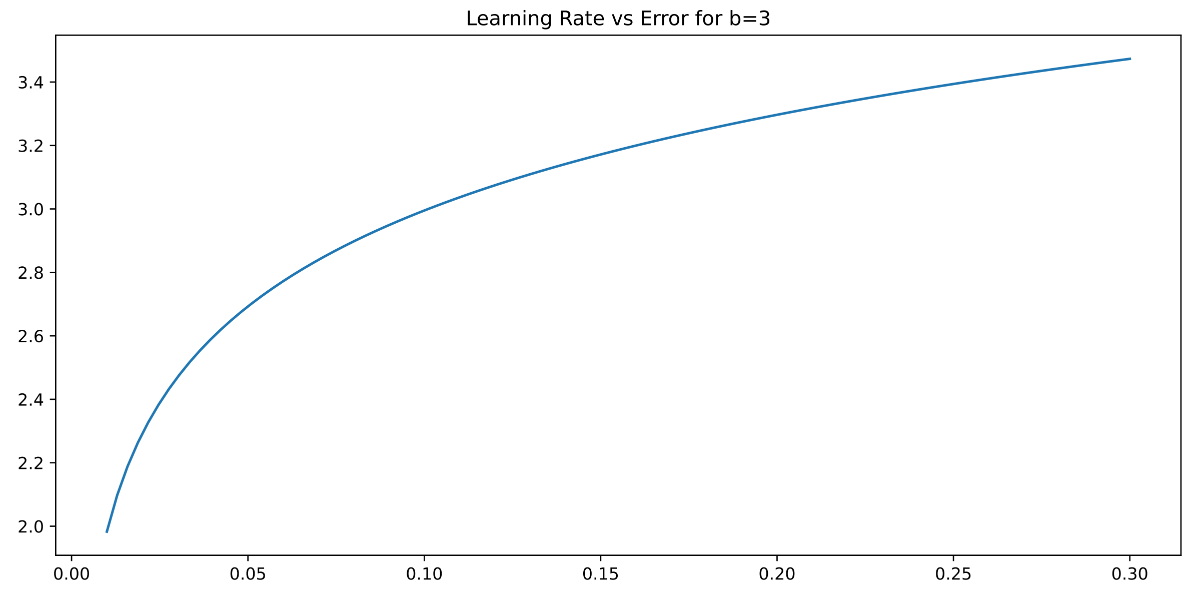
- Then test the performance of your algorithm as a function of the learning rates k by performing 100 iterations of the algorithm for 100 values of k equally spaced between $k = 0.01$ and $k = 0.3$. Start with an initial guess of $\mathbf{v}_0 = \begin{bmatrix} b & 1 \end{bmatrix}^T$. Do this for $b = 3$ and $b = 10$. Make separate plots for $b = 3$ and $b = 10$ of the log base 10 of the error (in this case it is $\sqrt{x_{100}^2 + y_{100}^2}$) for the final value of the iteration versus the value of k. How does learning rate relate to the final value of the error? For which value of b does the algorithm have the ability to converge fastest (have the lowest value of the error at the end)?

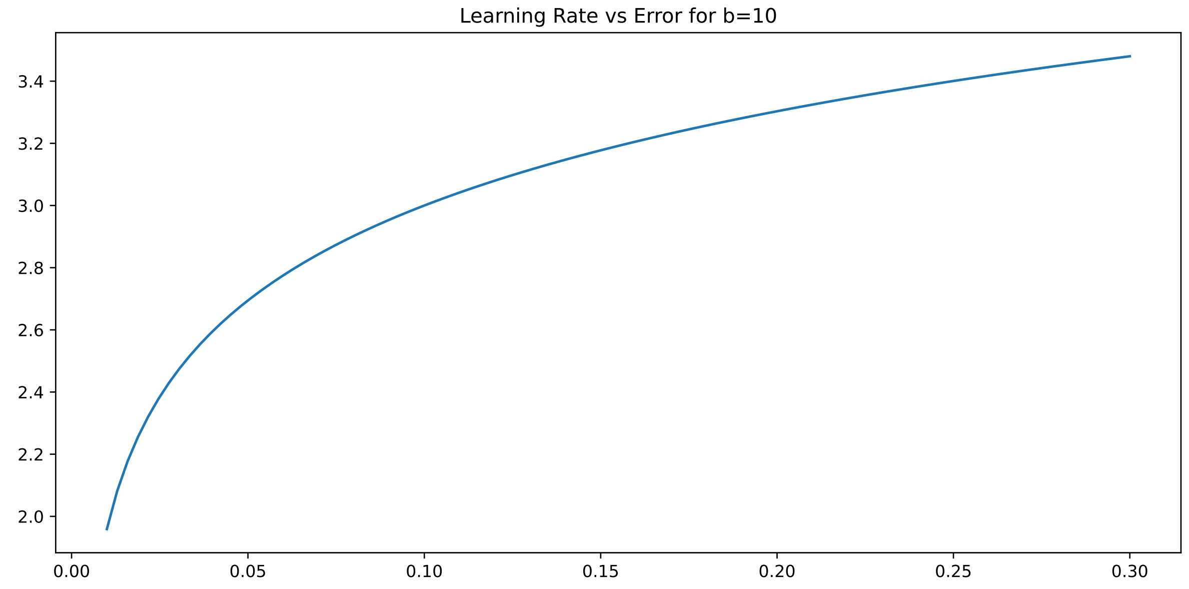
Problem 1 Solution

```
# load dependencies
import sympy as sp
import numpy as np
import matplotlib.pyplot as plt

## Part A
# define the function
x, y = sp.symbols('x y')
formula = (x-1)**2 + (y+2)**2

# find the critical point for the function
df_x = sp.diff(formula, x)
critical_point_x = sp.solve(df_x, x)

df_y = sp.diff(formula, y)
critical_point_y = sp.solve(df_y, y)

print(f'The critical point of x: {critical_point_x} and y: {critical_point_y}')

# find the local minimum
df_dx = sp.diff(df_x, x)
critical_point_x2 = df_dx.subs(x, critical_point_x[0])
print(f'{critical_point_x2} is positive and it is the local minimum')

## Part B
# load dependencies
import matplotlib.pyplot as plt

# define the part b function
def f(x, y, b):
    return x**2 + (b*(y**2))
```

```
# define the derivative of f(x, y) = x^2 + by^2 is d/dx = 2x and d/dy = 2by
def df_dx(x):
    return 2 * x

def df_dy(b, y):
    return 2 * b * y

# define the gradient descent function
def gradient_descent(start_x, start_y, b, k, iterations):

    # initialize the parameter
    x = start_x
    y = start_y
    vector = np.array([x, y])
    grad_array = []

    # run the gradient descent
    for x in range(iterations):
        # calculate the gradients
        gradient_x = df_dx(x)
        gradient_y = df_dy(b, y)
        grad_array = np.array([gradient_x, gradient_y])

        # update the parameter
        vector = vector - k * grad_array

    return vector

# define the error computation function
def compute_error(vector):
    return np.linalg.norm(vector)

# set the parameters
iterations = 100
k_values = np.linspace(0.01, 0.3, 100)
b_values = [3, 10]

# plot the result
fig, ax = plt.subplots(2, 1, figsize=(10, 10))

# test for b (3, 10)
for a, b in enumerate(b_values):
    errors = []

    for k in k_values:
        start_x = b
        start_y = 1

        # run gradient descent
        v_final = gradient_descent(start_x, start_y, b, k, iterations)

        # compute the error
        error = compute_error(v_final)

        # store the error with the log base of 10
        errors.append(np.log10(error))

    # plot
    ax[a].plot(k_values, errors)
    ax[a].set_title(f'Learning Rate vs Error for b={b}')

# show plots
plt.tight_layout()
plt.show()
```

```
The critical point of x: [1] and y: [-2]
2 is positive and it is the local minimum
```





Note: For some combinations of k and b, the algorithm won't converge to the right answer, i.e. the error will grow with time. To make your plot easier to read, don't plot the error for iterations that didn't converge.

- As k increases, for one or both values of b, you will observe a point where the trend of final error versus learning rate reverses direction. Pick a value of k very close to the point where this occurs, and make a contour plot of the function f and the trajectory of the iterations for the gradient descent algorithm for that value of k superimposed over the contour plot. What do you observe?

Note: The differences that you observe here are a special case of a more general phenomenon: the speed of convergence of gradient descent depends on something called the *condition number* of the *Hessian* matrix (the matrix of the 2nd order partial derivatives) of the target function. The condition number for a symmetric matrix is just the ratio of the largest to smallest eigenvalues, in this case the condition number is b (or $1/b$). Gradient descent performs worse and worse the larger the condition number (and large condition numbers are problematic for a wide variety of other numerical methods).

Problem 2: Solving Least Squares Problems

Generate a random 20×10 matrix A and a random 20-vector b (use a Gaussian distribution). Then, solve the least squares problem:

$$\min_{\mathbf{x} \in \mathbb{R}^{10}} \|A\mathbf{x} - \mathbf{b}\|^2$$

in the following ways:

(a) Multiply $\mathbf{b}$ by the Morse-Penrose Pseudoinverse A^+.

(b) Use built in functions to solve the least squares problem (i.e. in python numpy.lstsq, in R lm, and in Julia the backslash operator).

(c) Using the QR factorization of A. This factorization rewrites A as:

$$A = \begin{bmatrix} Q & 0 \end{bmatrix} \begin{bmatrix} R & 0 \end{bmatrix}^T,$$

where Q is an orthonormal matrix and R is upper triangular. The least squares solution equals:

$$\mathbf{x} = R^{-1} Q^T \mathbf{b}$$

(d) Verify that each of these solutions are nearly equal and that the residuals $A\mathbf{x} - \mathbf{b}$ are orthogonal to the vector $A\mathbf{x}$

Problem 2 Solution

```
# load dependencies
import numpy as np

## Part A
# define the matrix
a = np.random.randn(20, 10)
b = np.random.randn(20)

# multiply b by the Morse-Penrose Pseudoinverse a
```

```
x_normal = b @ np.linalg.pinv(a.T)
print('This is the normal solution.')
print(x_normal)

## Part B
# need to transpose a to get the least squares problem to work
x_lstsq = np.linalg.lstsq(a, b)[0]
print('This is the least squares problem solution.')
print(x_lstsq)

## Part C
# compute QR factorization of a
x_q, x_r = np.linalg.qr(a)

# compute tranpose of x_q and b
result_multiply_q = np.dot(x_q.T, b)

# solve the equation
x_qr = np.linalg.solve(x_r, result_multiply_q)
print('This is the QR factorization solution.')
print(x_qr)
```

```
This is the normal solution.
[ 0.0131463   0.24838509 -0.18571563  0.12712951  0.34347529 -0.42071949
 -0.39462928  0.15870199 -0.30568747  0.07353774]
This is the least squares problem solution.
[ 0.0131463   0.24838509 -0.18571563  0.12712951  0.34347529 -0.42071949
 -0.39462928  0.15870199 -0.30568747  0.07353774]
This is the QR factorization solution.
[ 0.0131463   0.24838509 -0.18571563  0.12712951  0.34347529 -0.42071949
 -0.39462928  0.15870199 -0.30568747  0.07353774]
```

```
/var/folders/h4/zjq554hs0b57vqfcrc5738wh0000gn/T/ipykernel_78780/1074645662.py:16: FutureWar

`rcond` parameter will change to the default of machine precision times ``max(M, N)`` where 
To use the future default and silence this warning we advise to pass `rcond=None`, to keep u
```

Problem 3: Iterative Solutions to Least Squares

Although the pseudoinverse provides an exact formula for the least squares solutions, there are some situations in which using the exact solution is computationally difficult, particularly when the matrix A and vector $\mathbf{b}$ have a large number of entries. In this case, AA^T, which is an $m \times m$ matrix if A is $m \times n$, may require an enormous amount of memory. In these cases it may be better to use an approximate solution instead of the exact formula. There are many different approximate methods for solving least squares problems, here we will use an iterative method developed by Richardson.

This method begins with an initial guess $\mathbf{x}^{(0)} = 0$ and calculates successive approximations as follows:

$$\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} - \mu A^T \left(A\mathbf{x}^{(k)} - \mathbf{b} \right)$$

Here μ is a positive parameter that has a similar interpretation to the learning rate for gradient descent. A choice that guarantees convergence is $\mu \leq \frac{1}{\|A\|}$. The iteration is terminated when the change in the residual $\|A^T(Ax^{(k)} - b)\|$ after successive steps is below a user determined threshold, which indicates that the least squares optimality conditions are nearly satisfied.

(a) Suppose that $\mathbf{x}$ is a solution to the least squares problem:

$$\mathbf{x} = A^+\mathbf{b}$$

Show by substitution of the formula for the pseudoinverse that $\mathbf{x}$ is a *fixed point* of the iteration scheme, i.e. that:

$$\mathbf{x} = \mathbf{x} - \mu A^T \left(A\mathbf{x} - \mathbf{b} \right)$$

(b) Generate a random 20×10 matrix A and 20-vector $\mathbf{b}$, and compute the least squares solution $\mathbf{x} = A^+\mathbf{b}$. Then run the Richardson algorithm with $\mu = \frac{1}{\|A\|^2}$ for 500 iterations, and plot $\|\mathbf{x}^{(k)} - \mathbf{x}\|$ to verify that $\mathbf{x}^{(k)}$ is converging to $\mathbf{x}$

Problem 3 Solution

Part A Solution

$$\text{substituting } x = A^+b \text{ to } x = x - \mu A^T \left(A\mathbf{x} - \mathbf{b} \right) A^+b = A^+b - \mu A^T(A(A^+b) - b)$$

$$A^+b = A^+b - \mu A^T(AA^+b - b) = A^+b = A^+b - \mu A^T(b - b) = A^+b = A^+b - \mu A^T(0)$$

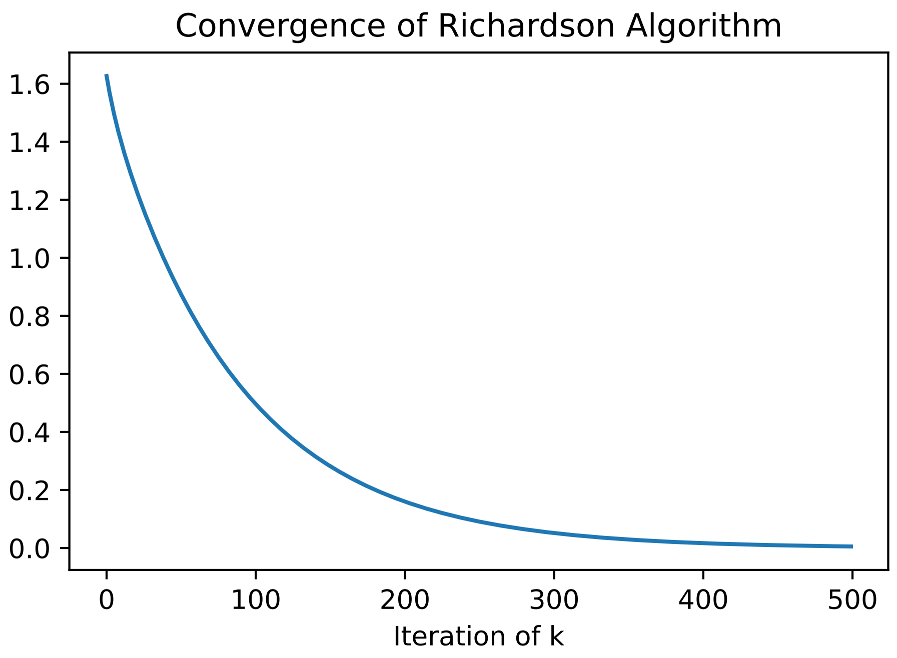
$$A^+b - \mu A^T(0) = A^+b - 0$$

$$A^+b = A^+b$$

```python
# load dependencies
import numpy as np
import matplotlib.pyplot as plt

## Part B
# define the matrix
a = np.random.randn(20, 10)
b = np.random.randn(20)

# compute the least square problem
x_lstsq2 = np.linalg.lstsq(a, b)[0]
print('This is the least squares problem solution.')
print(x_lstsq2)

# define the parameter
mu = 1 / np.linalg.norm(a)**2
x_k = np.zeros(10)
iterations = 500

# store the convergence
convergence = []

# define the richardson algorithm
for i in range(iterations):
    x_k = x_k + mu * a.T.dot(b - a.dot(x_k))
    convergence.append(np.linalg.norm(x_k - x_lstsq2))

# plot
plt.plot(convergence)
plt.xlabel('Iteration of k')
plt.title('Convergence of Richardson Algorithm')
plt.show()
```

```
This is the least squares problem solution.
[ 0.62023471  0.41087762 -0.48372681  0.25631921  0.31781878  1.03471953
  0.28944381  0.74570357  0.08518563 -0.28468751]
```

```
/var/folders/h4/zjq554hs0b57vqfcrc5738wh0000gn/T/ipykernel_78780/3144541300.py:11: FutureWar

`rcond` parameter will change to the default of machine precision times ``max(M, N)`` where 
To use the future default and silence this warning we advise to pass `rcond=None`, to keep u
```


Convergence of Richardson Algorithm
1.6
1.4
1.2
1.0
0.8
0.6
0.4
0.2
0.0
0
100
200
300
400
500
Iteration of k