CS436/536: Introduction to Machine Learning
Homework 2

Name : Ankita Patra
B-number : B01101280
Mail : apatra@binghamton.edu

(1) LFD Exercise 2.3. [5/5/5 points]

1. **Positive Rays** :

- A positive ray starts from a point and extends indefinitely in one direction.
- It can shatter a single point (by placing the threshold anywhere beyond it).
- But it cannot shatter two arbitrary points because there's only one threshold.
- VC dimension = 1.

2. **Positive Intervals**:

- A positive interval ([a, b]) has two boundaries.
- It can shatter any two points (by selecting appropriate intervals).
- But it cannot shatter three arbitrary points since we cannot include only the middle one while excluding the other two.
- VC dimension = 2.

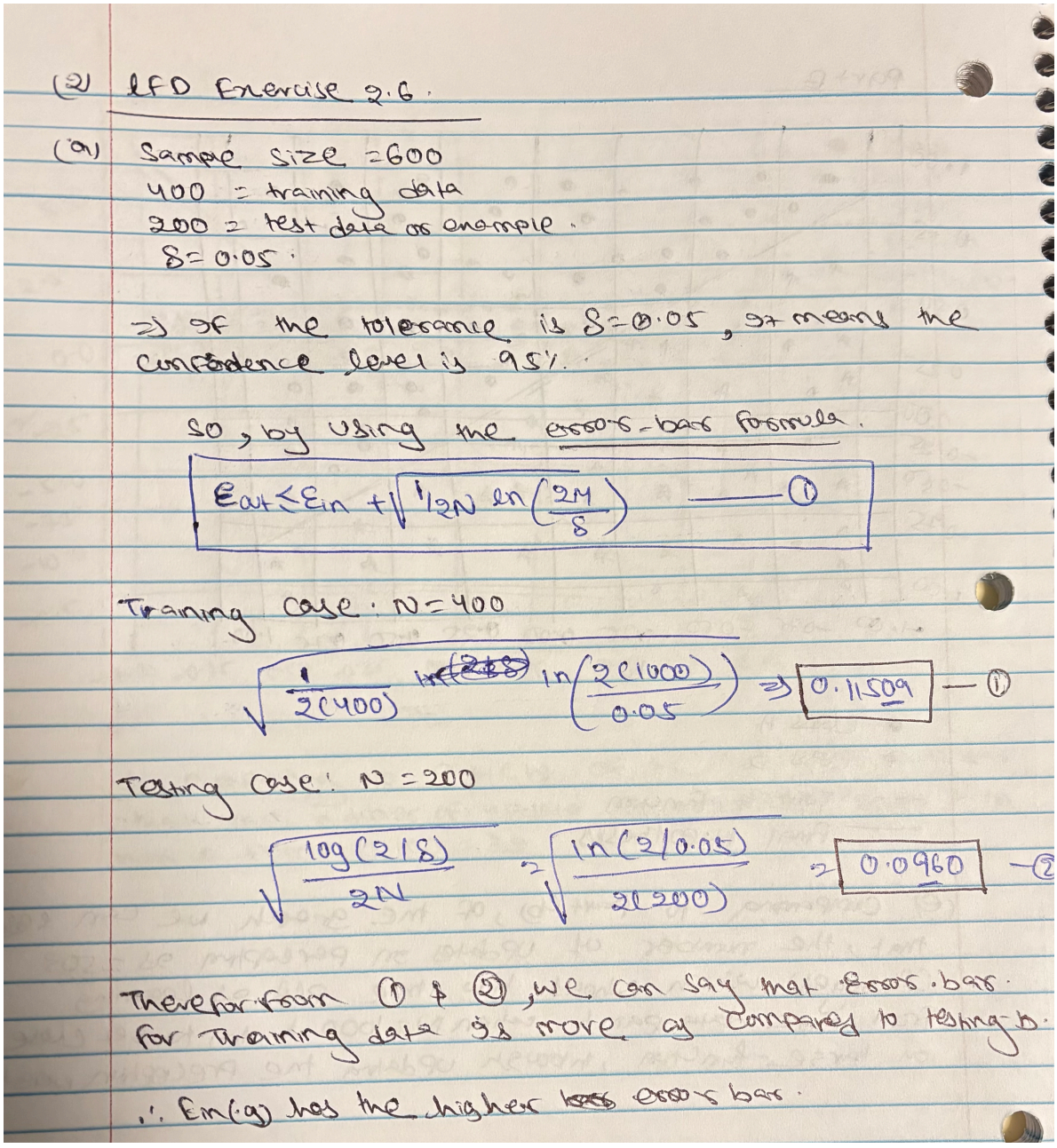
3. **Convex Sets in 2D**:

- Convex sets in 2D are very flexible.
- A convex set can be adjusted to contain or exclude points in a way that could shatter arbitrarily many points (depending on the model).
- If there's no upper bound on the number of points that can be shattered, the VC dimension is **infinite**.

Final Answer:

VC dimension of Positive Rays = 1
VC dimension of Positive Intervals= 2
VC dimension of Convex Sets in 2D = ∞ (infinite)

(2) LFD Exercise 2.6. [8/7 points]



(2) LFD Exercise 2.6.

(a) Sample size = 600
400 = training data
200 = test data as example.
$\delta = 0.05$.

⇒ If the tolerance is $\delta = 0.05$, it means the confidence level is 95%.

So, by using the error-bar formula.

$$E_{out} \leq E_{in} + \sqrt{\frac{1}{2N} \ln\left(\frac{2M}{\delta}\right)} \quad \text{——①}$$

Training case: $N = 400$

$$\sqrt{\frac{1}{2(400)} \ln\left(\frac{2(1000)}{0.05}\right)} \Rightarrow \boxed{0.11509} \quad \text{—①}$$

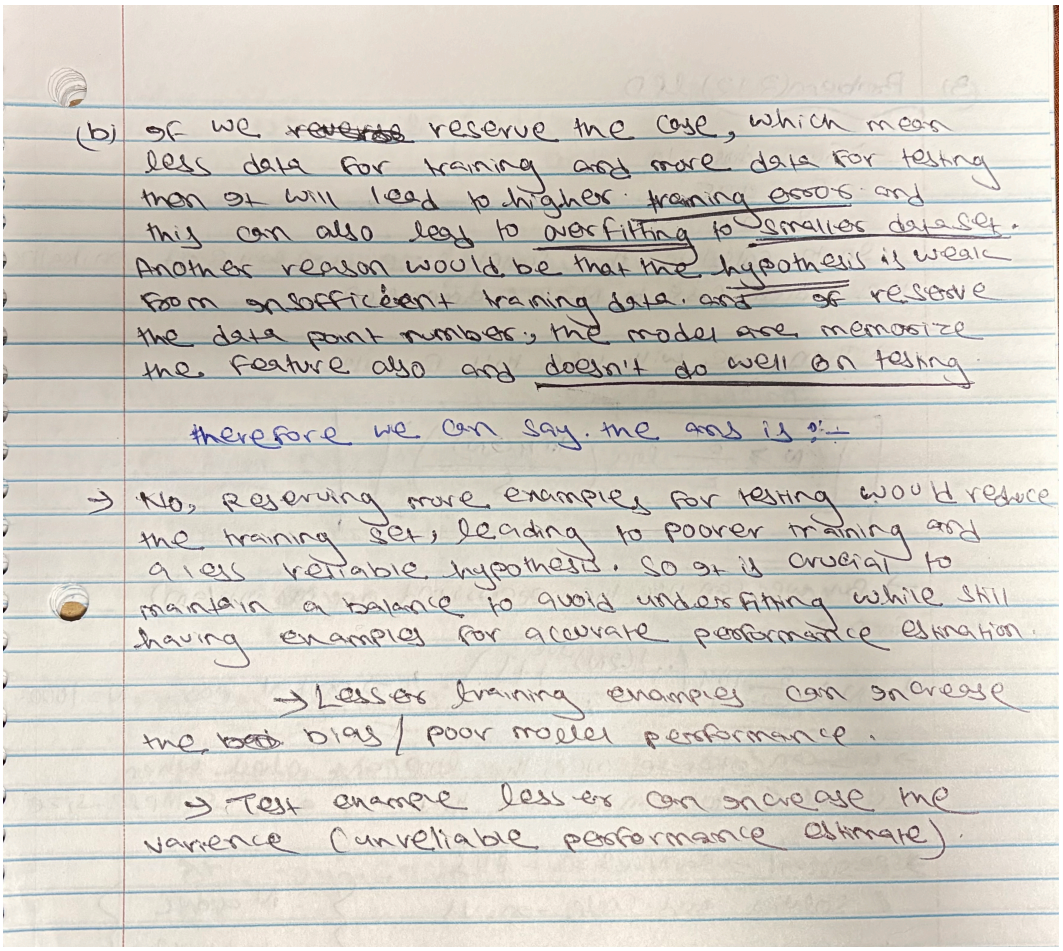
Testing case: $N = 200$

$$\sqrt{\frac{\log(2/\delta)}{2N}} = \sqrt{\frac{\ln(2/0.05)}{2(200)}} = \boxed{0.0960} \quad \text{—②}$$

Therefor from ① & ②, we can say that error bar for training data is more as compared to testing b.

∴ $E_{in}(g)$ has the higher error bar.

b)



(b) If we ~~reverse~~ reserve the case, which mean less data for training and more data for testing then it will lead to higher training error and this can also lead to overfitting to smaller dataset. Another reason would be that the hypothesis is weak from insufficient training data. and if reserve the data point number, the model are memorize the feature also and doesn't do well on testing.

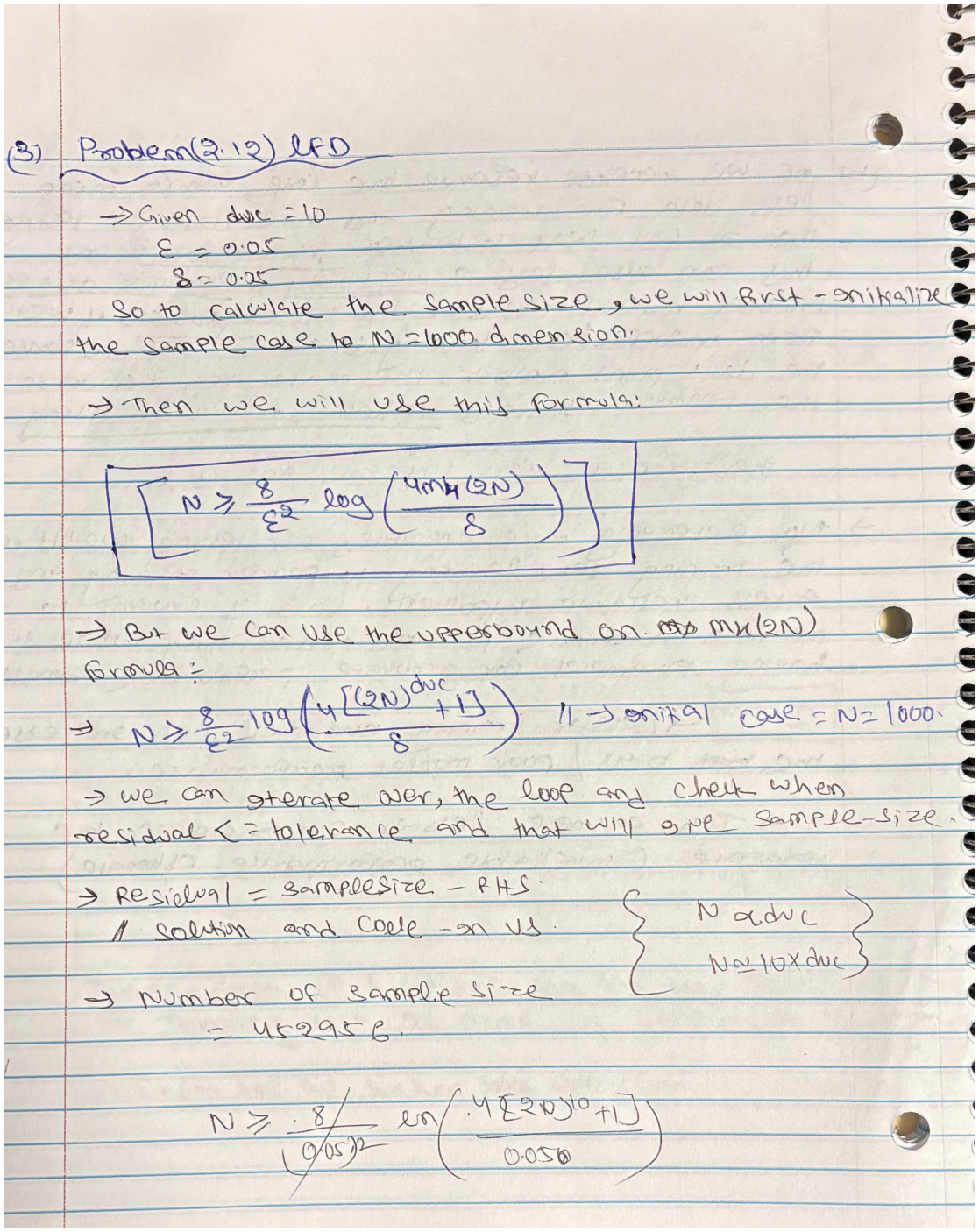
therefore we can say. the ans is :-

→ No, Reserving more examples for testing would reduce the training set, leading to poorer training and a less reliable hypothesis. So it is crucial to maintain a balance to avoid underfitting while still having examples for accurate performance estimation.

→ Lesser training examples can increase the ~~bad~~ bias / poor model performance.

→ Test examples lesser can increase the variance (unreliable performance estimate).

(3) LFD Problem 2.12. [10 points]



(3) Problem (2.12) LFD

→ Given dvc = 10

$\varepsilon = 0.05$

$\delta = 0.05$

So to calculate the sample size, we will first - initialize the sample case to N = 1000 dimension.

→ Then we will use this formula:

$$N \geq \frac{8}{\varepsilon^2} \log\left(\frac{4 m_{\mathcal{H}}(2N)}{\delta}\right)$$

→ But we can use the upperbound on $m_{\mathcal{H}}(2N)$ formula:

→ $N \geq \frac{8}{\varepsilon^2} \log\left(\frac{4[(2N)^{dvc} + 1]}{\delta}\right)$ // → initial case = N = 1000

→ we can iterate over, the loop and check when residual <= tolerance and that will give sample-size.

→ Residual = samplesize - RHS.

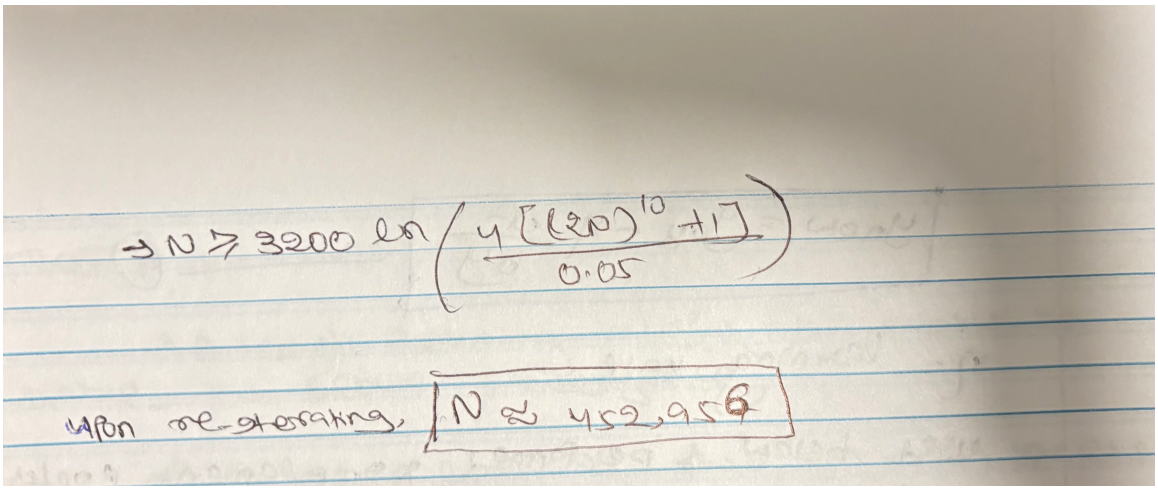
// solution and Code - on VS.

→ Number of sample size = 44296.

N ∝ dvc

N ≈ 10 × dvc

$N \geq \frac{8}{(0.05)^2} \ln\left(\frac{4[2(1000)^{10} + 1]}{0.05}\right)$




```
# Given values
d_vc = 10
error = 0.05
confidence_level = 0.05
starting_input = 1000
# Compute required sample size
Number_of_DATA = Required_sample_size(d_vc, error,
confidence_level, starting_input)
print(f"The Number of data points required is =
{Number_of_DATA}")
```

O/p:

ankitapatra@Mac-1414 Machine Learning % python3 HW2.py
The Number of data points required is = 452956

(4) Gradient Descent on a Simple Function. [60 points]
Consider the function $f(x, y) = 2x2 + y2 + 3\sin(2\pi x)\cos(2\pi y)$.
(a) Implement gradient descent to minimize this function. Run gradient descent starting from the point ($x = 0.1, y = 0.1$). Set the learning rate to $\eta = 0.01$ and the number of iterations to 50. Give a plot that displays how
the function value drops through successive iterations of gradient descent. Repeat this with a learning rate of η
$= 0.1$ and provide a plot of the function value with each iteration. What do you observe?
(b) Obtain the "minimum" value and location of the minimum value of the function you get using gradient descent
with the same learning rate $\eta = 0.01$ and number of iterations (50) as part (a), from the following starting points:
(i) $(0.1, 0.1)$, (ii) $(1, 1)$, (iii) $(0.5, 0.5)$, (iv) $(0.0, 0.5)$, (v) $(-0.5,$
$-0.5)$, (vi) $(-1, 1)$. Write down the minimum
value obtained using gradient descent and the location of the minimum value for each of these starting points.
As you may appreciate, finding the "true" *global* minimum value of an arbitrary function is a hard problem.
Grading Rubrics:
(a) You need to implement python codes for gradient descent algorithm (see pseudocode in ML7 slides) and plotting.
- $\eta = 0.01$: answer correctness 10 points, plots 5 points
- $\eta = 0.1$: answer correctness 5 points, plots 5 points
- Observation: 5 points
Totally 30 points for (a)
(b) Each calculated "minimum value" and "the location" (a 2D vector) starting from (i)-(vi) 5 points, totally 30 points.

Ans :

A)

```
import numpy as np
import matplotlib.pyplot as plt
def f(x, y):
    return 2*x**2 + y**2 + 3*np.sin(2*np.pi*x)*np.cos(2*np.pi*y)
def grad_f(x, y):
    dx = 4*x + 6*np.pi*np.cos(2*np.pi*x)*np.cos(2*np.pi*y)
    dy = 2*y - 6*np.pi*np.sin(2*np.pi*x)*np.sin(2*np.pi*y)
    return np.array([dx, dy])
def gradient_descent(x0, y0, learning_rate, num_iterations):
    x, y = x0, y0
    f_values = []
    for _ in range(num_iterations):
        f_values.append(f(x, y))
        grad = grad_f(x, y)
        x -= learning_rate * grad[0]
```

```
        y -= learning_rate * grad[1]
    return x, y, f_values
# Run gradient descent with η = 0.01
x_min1, y_min1, f_values1 = gradient_descent(0.1, 0.1, 0.01, 50)
# Run gradient descent with η = 0.1
x_min2, y_min2, f_values2 = gradient_descent(0.1, 0.1, 0.1, 50)
# Plot results - COMBINED PLOT
plt.figure(figsize=(8, 6))  # Adjust figure size if needed
plt.plot(f_values1, label='η = 0.01')
plt.plot(f_values2, label='η = 0.1')
plt.xlabel('Iteration')
plt.ylabel('Function Value')
plt.title('Gradient Descent Progress')
plt.grid(True)  # Add grid lines
plt.legend()
plt.show()
# # --- Part B: Find minimum values from different starting
points (COMMENTED OUT) ---
# starting_points = [(0.1, 0.1), (1, 1), (0.5, 0.5), (0.0, 0.5),
(-0.5, -0.5), (-1, 1)]
#
# print("\n--- Part B: Minimum Values from Different Starting
Points ---")
# for i, (x0, y0) in enumerate(starting_points):
#     x_min, y_min, _ = gradient_descent(x0, y0, 0.01, 50)
#     min_value = f(x_min, y_min)
#     print(f"Starting Point {i+1}: ({x0}, {y0})")
#     print(f"Minimum Value: {min_value:.4f}")
#     print(f"Location: ({x_min:.4f}, {y_min:.4f})")
print("\nObservations:")
print("  - Smaller learning rate (η=0.01) leads to a smoother,
more stable descent, but may converge slower.")
print("  - Larger learning rate (η=0.1) can lead to faster
initial descent, but risks overshooting the minimum, resulting
in oscillations.")
```

O/P:

ankitapatra@Ankitas-MacBook-Pro Machine Learning % python3 gradient.py
2025-02-18 16:40:22.419 Python[3112:76797] +[IMKClient subclass]: chose IMKClient_Modern
2025-02-18 16:40:22.419 Python[3112:76797] +[IMKInputSession subclass]: chose IMKInputSession_Modern

A)

Observations:

- Smaller learning rate (η=0.01) leads to a smoother, more stable descent, but may converge slower.
- Larger learning rate (η=0.1) can lead to faster initial descent, but risks overshooting the minimum, resulting in oscillations.

B)

Starting point 1: (0.1, 0.1)
Minimum value: -2.8790846587644263
Location: (-0.24182894588827974, -6.848884023658433e-35)

Starting point 2: (1, 1)
Minimum value: -0.9286447086312597
Location: (0.7252678803578847, 0.9831635693235358)

Starting point 3: (0.5, 0.5)
Minimum value: -2.633242590975637
Location: (0.24181813075114938, 0.4916822590684804)

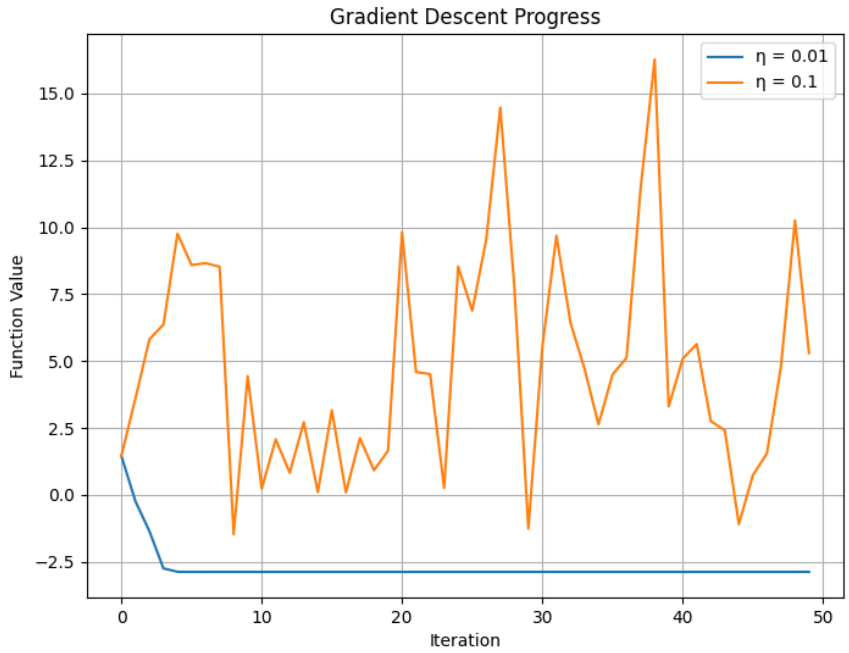
Starting point 4: (0.0, 0.5)
Minimum value: -2.633242590975637
Location: (0.2418181307511494, 0.4916822590684803)

Starting point 5: (-0.5, -0.5)
Minimum value: -1.6660267055389741
Location: (-0.7253691676028068, -0.4915941934004666)

Starting point 6: (-1, 1)
Minimum value: 1.0051615759793315
Location: (-1.2084759495263577, 0.9827889176425988)

O/P plot :

A)

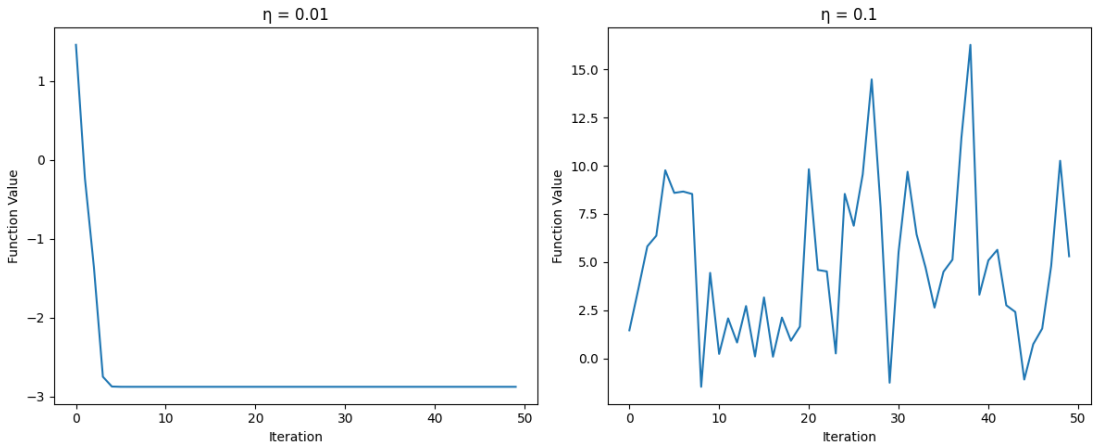


If we can see this in different graph : $\eta = 0.01$, $\eta = 0.1$
Then, here is the code and graph :
4(A)

```
import numpy as np
import matplotlib.pyplot as plt
def f(x, y):
    return 2*x**2 + y**2 + 3*np.sin(2*np.pi*x)*np.cos(2*np.pi*y)
def grad_f(x, y):
    dx = 4*x + 6*np.pi*np.cos(2*np.pi*x)*np.cos(2*np.pi*y)
    dy = 2*y - 6*np.pi*np.sin(2*np.pi*x)*np.sin(2*np.pi*y)
    return np.array([dx, dy])
def gradient_descent(x0, y0, learning_rate, num_iterations):
    x, y = x0, y0
    f_values = []
    for _ in range(num_iterations):
        f_values.append(f(x, y))
        grad = grad_f(x, y)
        x -= learning_rate * grad[0]
        y -= learning_rate * grad[1]
    return x, y, f_values
```

```
# Run gradient descent with η = 0.01
x_min1, y_min1, f_values1 = gradient_descent(0.1, 0.1, 0.01, 50)
# Run gradient descent with η = 0.1
x_min2, y_min2, f_values2 = gradient_descent(0.1, 0.1, 0.1, 50)
# Plot results
plt.figure(figsize=(12, 5))
plt.subplot(121)
plt.plot(f_values1)
plt.title('η = 0.01')
plt.xlabel('Iteration')
plt.ylabel('Function Value')
plt.subplot(122)
plt.plot(f_values2)
plt.title('η = 0.1')
plt.xlabel('Iteration')
plt.ylabel('Function Value')
plt.tight_layout()
plt.show()
starting_points = [(0.1, 0.1), (1, 1), (0.5, 0.5), (0.0, 0.5),
(-0.5, -0.5), (-1, 1)]
for i, (x0, y0) in enumerate(starting_points, 1):
    x_min, y_min, _ = gradient_descent(x0, y0, 0.01, 50)
    min_value = f(x_min, y_min)
    print(f"Starting point {i}: ({x0}, {y0})")
    print(f"Minimum value: {min_value}")
    print(f"Location: ({x_min}, {y_min})")
print()
```

0/p :



Observations:

- Smaller learning rate (η=0.01) leads to a smoother, more stable descent, but may converge slower.
- Larger learning rate (η=0.1) can lead to faster initial descent, but risks overshooting the minimum, resulting in oscillations.

b) ankitapatra@Ankitas-MacBook-Pro Machine Learning % python3 gradient_decent2.py

2025-02-18 16:46:36.046 Python[3157:80630] +[IMKClient subclass]: chose IMKClient_Modern
2025-02-18 16:46:36.046 Python[3157:80630] +[IMKInputSession subclass]: chose IMKInputSession_Modern
Starting point 1: (0.1, 0.1)
Minimum value: -2.8790846587644263
Location: (-0.24182894588827974, -6.848884023658433e-35)

Starting point 2: (1, 1)
Minimum value: -0.9286447086312597
Location: (0.7252678803578847, 0.9831635693235358)

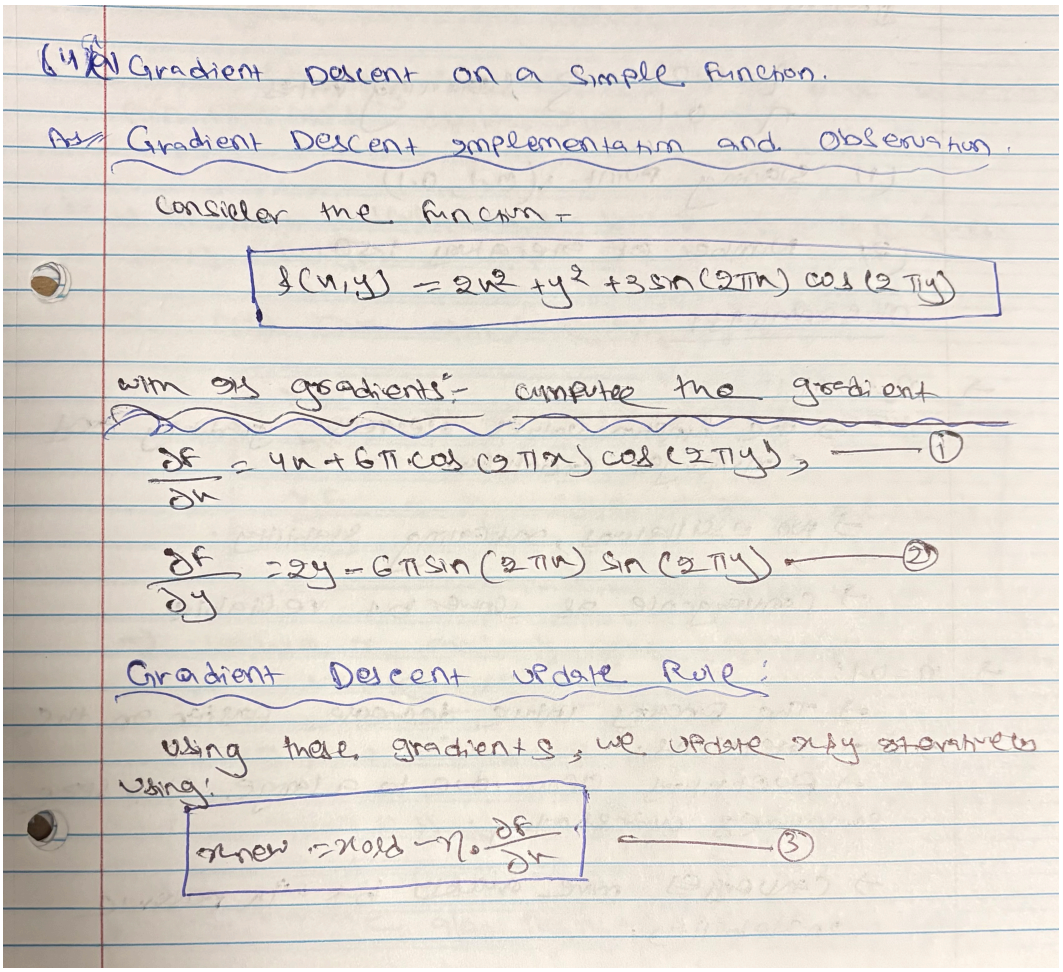
Starting point 3: (0.5, 0.5)
Minimum value: -2.633242590975637
Location: (0.24181813075114938, 0.4916822590684804)

Starting point 4: (0.0, 0.5)
Minimum value: -2.633242590975637
Location: (0.2418181307511494, 0.4916822590684803)

Starting point 5: (-0.5, -0.5)
Minimum value: -1.6660267055389741
Location: (-0.7253691676028068, -0.4915941934004666)

Starting point 6: (-1, 1)
Minimum value: 1.0051615759793315
Location: (-1.2084759495263577, 0.9827889176425988)

Here is my understanding which I implemented in my code for 4(a and b) question .



(4) Gradient Descent on a Simple Function.

Ans Gradient Descent implementation and Observation.

Consider the function -

$$f(x,y) = 2x^2 + y^2 + 3\sin(2\pi x)\cos(2\pi y)$$

with its gradients:- compute the gradient

$$\frac{\partial f}{\partial x} = 4x + 6\pi\cos(2\pi x)\cos(2\pi y), \quad \text{---(1)}$$

$$\frac{\partial f}{\partial y} = 2y - 6\pi\sin(2\pi x)\sin(2\pi y) \quad \text{---(2)}$$

Gradient Descent update Rule:

Using these gradients, we update x,y starvariables using:

$$x_{new} = x_{old} - \eta \cdot \frac{\partial f}{\partial x} \quad \text{---(3)}$$

$$y_{new} = y_{old} - \eta \cdot \frac{\partial f}{\partial y} \quad \text{---------- (1)}$$

η = learning rate.

used below & performed & implement codes & graphs.

$\eta = 0.01$ } learning rates.
$\eta = 0.1$

(1) Starting point : (0.1, 0.1)

(2) Number of iterations : 50.

Observations:

→ $\eta = 0.01$:

→ The function values decrease gradually and converges smoothly.

→ No oscillations, indicating stability.

→ Convergence is slower but reliable.

→ $\eta = 0.1$:

→ The function value decrease faster in the initial iterations.

→ Fluctuations occur due to a large step size, sometimes overshooting.

→ Converges more quickly but with possible instability.

In summary,

→ A small learning rate ($\eta = 0.01$) results in stable but slower convergence.

→ A larger learning rate ($\eta = 0.1$) speeds up convergence but can cause oscillations.

→ Choosing the right learning rate is crucial for efficient optimization.

(b) → We start calculate x & y value with the given point (x_0, y_0)

→ We can compute $\frac{\partial f}{\partial x}$ & $\frac{\partial f}{\partial y}$.

$$\frac{\partial f}{\partial x} = 4x + 6\pi \cos(2\pi x)\cos(2\pi y)$$

$$\frac{\partial f}{\partial y} = 2y - 6\pi \sin(2\pi x)\sin(2\pi y)$$

→ Update x and y using gradient descent update rule:

$$x_{new} = x_{old} - \eta \cdot \frac{\partial f}{\partial x}$$

$$y_{new} = y_{old} - \eta \cdot \frac{\partial f}{\partial y}$$

iterations = 50.

∴ we run the algo then we receive the o/p.

Thank You