PS 5

Sunday, October 29, 2023 8:09 PM

All the code can be accessed here and executed in google colab or any Jupyter Notebook environment:
https://github.com/DragonBoy25830/caltech-cs-156

1)

```
[10]
Ein_threshold = 0.008
sigma = 0.1
d = 8
```

```
[11]
def lin_reg_error(sigma_in, d_in, N):
    return sigma_in * sigma_in * (1 - (d_in + 1) / N)
```

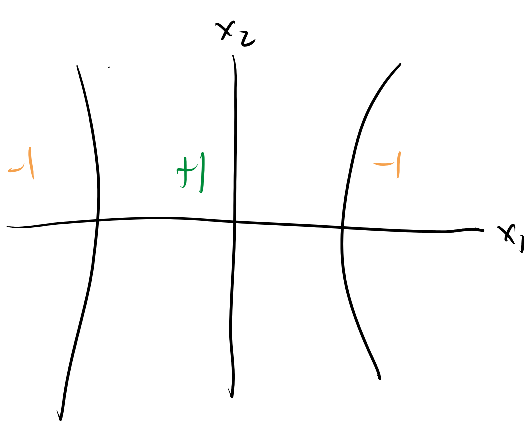
```
[12]
for i in range(1, 1000):
    if lin_reg_error(sigma, d, i) > Ein_threshold:
        print(f"Minimum N: {i}")
        break

Minimum N: 45
```

The above code shows us that the smallest N that leads to the expected in-sample error being greater than 0.008 is 45.

The next answer choice larger than this is 100, so

$\boxed{\text{the answer is c}}$

2)



In a non-linear transformation:

$$g(x) = \text{sign}(\tilde{w}\,\Phi(x))$$

$$= \text{sign}\left(\begin{bmatrix}\tilde{w}_0 & \tilde{w}_1 & \tilde{w}_2\end{bmatrix} \cdot \begin{bmatrix}1 & x_1^2 & x_2^2\end{bmatrix}\right)$$

$= \text{sign}(\tilde{w}_0 + \tilde{w}_1 x_1^2 + \tilde{w}_2 x_2^2)$

If we fix $x_2 = 0$, we see that $g(x) > 0$ when x_1 is large and $g(x) < 0$ when x_1 is small. Assuming that $\tilde{w}_0$ is made sufficiently large to achieve the desired boundary, $\tilde{w}_1 < 0$ to produce the desired decision boundary.

If we fix $x_1 = 0$, we see that $g(x) > 0$ for all x_2. As such $\tilde{w}_2 > 0$.

Since $\tilde{w}_1 < 0$ & $\tilde{w}_2 > 0$, $\boxed{\text{the answer is d.}}$

3) In the transformed space, $d_{vc} \leq \tilde{d} + 1$ where $\tilde{d}$ is the number of features in the transformed space. By simple counting, we see that $\Phi_4 : x$ has 15 features, so

$d_{vc} < 14 + 1 = 15$. The smallest value among the choices not smaller than the VC dimension of a linear model in the transformed space is 15, so $\boxed{\text{the answer is c.}}$

4) $\frac{\partial E}{\partial u}\left[\left(ue^{v}-2ve^{-u}\right)^{2}\right] = 2\left(ue^{v}-2ve^{-u}\right)\left(e^{v}+2ve^{-u}\right)$

$\boxed{\text{The answer is } e.}$

5)

```python
import numpy as np
```
[1] ✓ 0.1s

```python
n = 0.1
uv_array = [(1, 1)]
```
[2] ✓ 0.0s

```python
def partial_Eu(u, v):
    return 2 * (u * np.exp(v) - 2 * v * np.exp(-1 * u)) * (np.exp(v) + 2 * v * np.exp(-1 * u))
```
[3] ✓ 0.0s

```python
def partial_Ev(u, v):
    return 2 * (u * np.exp(v) - 2 * v * np.exp(-1 * u)) * (u * np.exp(v) - 2 * np.exp(-1 * u))
```
[4] ✓ 0.0s

```python
def get_last_uv():
    return uv_array[-1][0], uv_array[-1][1]
```
[5] ✓ 0.0s

```python
def gradient_descent(n_in):
    u_t, v_t = get_last_uv()

    u_t1 = u_t - n_in * partial_Eu(u_t, v_t)
    v_t1 = v_t - n_in * partial_Ev(u_t, v_t)

    uv_array.append((u_t1, v_t1))
```
[6] ✓ 0.0s

```python
def calc_error(u, v):
    return (u * np.exp(v) - 2 * v * np.exp(-1 * u)) ** 2
```
[7] ✓ 0.0s

```python
num_iter = 0

while True:
    u_t, v_t = get_last_uv()

    if calc_error(u_t, v_t) < 1e-14:
        print(num_iter)
        break

    gradient_descent(n)
    num_iter += 1
```
[8] ✓ 0.0s

```
10
```

The above code shows that it takes 10 iterations for the error $E(u,v)$ to fall below 10^{-14}, so $\boxed{\text{the answer is d}}$

6)

```
get_last_uv()
[9]  ✓ 0.0s
...  (0.04473629039778207, 0.023958714099141746)
```

The above code outputs the final (u,v) to be closest to $(0.045, 0.024)$, so $\boxed{\text{the answer is e}}$

7)

```
uv_array = [(1, 1)]
[10]  ✓ 0.0s

def coordinate_descent(n_in):
    u_t, v_t = get_last_uv()

    u_t1 = u_t - n_in * partial_Eu(u_t, v_t)
    v_t1 = v_t - n_in * partial_Ev(u_t1, v_t)

    uv_array.append((u_t1, v_t1))
[11]  ✓ 0.0s

for i in range(15):
    coordinate_descent(n)

    if i == 14:
        u_t, v_t = get_last_uv()
        print(calc_error(u_t, v_t))
[12]  ✓ 0.0s
...  0.13981379199615315
```

The error outputted by the code above is closest to 10^{-1}, so $\boxed{\text{the answer is a}}$

8)

```
import numpy as np
import random as rand
[1]  ✓ 0.0s

num_train_points = 100
num_test_points = 1000
learning_rate = 0.01
[2]  ✓ 0.0s
```

8)

```
import numpy as np
import random as rand
```
[1] ✓ 0.0s

```
num_train_points = 100
num_test_points = 1000
learning_rate = 0.01
```
[2] ✓ 0.0s

```
def generate_target_function():
    x0 = rand.uniform(-1, 1)
    y0 = rand.uniform(-1, 1)
    x1 = rand.uniform(-1, 1)
    y1 = rand.uniform(-1, 1)

    m_f = (y1 - y0) / (x1 - x0)
    b_f = y0 - m_f * x0
    f = [m_f, b_f]

    return f
```
[3] ✓ 0.0s

```
def classify_point(m, b, x, y):
    expected_value = m * x + b

    if y >= expected_value:
        return 1
    else:
        return -1
```
[4] ✓ 0.0s

```
def generate_data(m, b, num_points):
    bias = [1 for _ in range(num_points)]
    x1 = [rand.uniform(-1, 1) for _ in range(num_points)]
    x2 = [rand.uniform(-1, 1) for _ in range(num_points)]

    xn_pre_bias = np.column_stack((x1, x2))
    xn = np.column_stack((bias, xn_pre_bias))
    yn = [classify_point(m, b, x1, x2) for (bias, x1, x2) in xn]

    training_data = np.column_stack((xn, yn))

    return training_data
```
[5] ✓ 0.0s

```
def norm(vec):
    return np.sqrt(np.sum(vec * vec))
```
[6] ✓ 0.0s

```
def SGD(input_weight, training_data):
    perm_training_data = np.random.permutation(training_data)
    xn = np.transpose(np.delete(np.transpose(perm_training_data), 3, 0))
    yn = np.transpose(np.transpose(perm_training_data)[3])

    N = len(xn)

    for i in range(N):
        xi = xn[i]
        yi = yn[i]

        gradient = -1 * (yi * xi) / (1 + np.exp(yi * np.dot(input_weight, xi)))
        input_weight = input_weight - learning_rate * gradient

    return input_weight
```
[7] ✓ 0.0s

```
def calc_cross_entropy_error(testing_data, weight):
    xn = np.transpose(np.delete(np.transpose(testing_data), 3, 0))
    yn = np.transpose(np.transpose(testing_data)[3])

    N = len(xn)

    error = 0

    for i in range(N):
        xi = xn[i]
        yi = yn[i]

        error += np.log(1 + np.exp(-yi * np.dot(weight, xi)))

    return (1 / N) * error
```
[8] ✓ 0.0s

```
Eout_errors = []
epoch_weights = np.array([[0, 0, 0]])
num_epochs_arr = []
```
[9] ✓ 0.0s

```python
def run_experiment():
    m_f, b_f = generate_target_function()
    # generate target function
    train_data = generate_data(m_f, b_f, num_train_points)

    # run SGD
        # run N epochs till subsequent weights are within two decimal places

    prev_weight = epoch_weights[-1]
    weight = SGD(prev_weight, train_data)
    epoch_iter = 1

    while norm(prev_weight - weight) >= 0.01:
        np.append(epoch_weights, [weight], axis=0)
        prev_weight = weight
        weight = SGD(prev_weight, train_data)
        epoch_iter += 1

    num_epochs_arr.append(epoch_iter)
    # generate testing data
    testing_data = generate_data(m_f, b_f, num_test_points)
    # estimate E_out
    Eout_errors.append(calc_cross_entropy_error(testing_data, weight))
```
[12] ✓ 0.0s

```python
for i in range(100):
    run_experiment()
```
[13] ✓ 15.8s

```python
np.mean(Eout_errors)
```
[14] ✓ 0.0s

```
0.10069016921792245
```

The above code outputs a mean E_{out} for $N=100$ of 0.10069 which is closest to 0.100, so the answer is d

9)

```python
np.mean(num_epochs_arr)
```
[15] ✓ 0.0s

```
335.5
```

The above code outputs an average # of epochs to convergence of 335.5 which is closest to 350, so the answer is a.

10) An iteration of PLA is $w_{new} = w_{old} + yx$ where (x,y) is a misclassified point. If the point is classified, then there is no change in w. With SGD, the form is

$W_{new} = W_{old} - \eta \nabla E_{in}(w)$. Assuming the learing rate to be 1, we need $-\nabla E_{in}(w)$ to match yx. The gradient of answer choice e is $-\min(0, y_n x_n)$ & plugging this into $-\nabla E_{in}(w)$ gives us yx & the desired behavior for PLA, so

the answer is e.