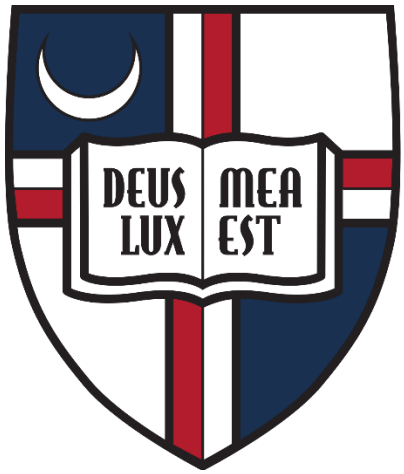
The Catholic University of America
School of Engineering
Department of Electrical Engineering and Computer Science



Project 2 Report
Implementation on Least Mean Square algorithm and the comparison among Rosenblatt's Perceptron, Least Square algorithm, and Least Mean Square algorithm

Student
Loc Tran

Fundamentals of Neural Networks
Dr. Hieu Bui
November 12th, 2020

TABLE OF CONTENTS

1. Least Mean Square Algorithm

The Rosenblatt's Perception algorithm was invented by Widrow and Hoff in 1960. The textbook mentions that this was the first linear adaptive-filtering algorithm for solving problems such as prediction and communication-channel equalization.

In terms of computational complexity, the LMS algorithm's complexity is linear with respect to adjustable parameters, which makes the algorithm computationally efficient, yet the algorithm is effective in performance.

The algorithm is simple to code and therefore easy to build.

Above all, the algorithm is robust with respect to external disturbances.

Purpose: To minimize the instantaneous value of the cost function

$$\mathscr{E}(\hat{\mathbf{w}}) = \frac{1}{2}e^2(n)$$

where e(n) is the error signal measured at time n.

Eventually, we can come up with the final formula for LMS algorithm:

$$\hat{\mathbf{w}}(n+1) = \hat{\mathbf{w}}(n) + \eta\mathbf{x}(n)e(n)$$

The overall step of LMS algorithm is summarized in Table 3.1

TABLE 3.1 Summary of the LMS Algorithm

Training Sample: Input signal vector = $\mathbf{x}(n)$
Desired response = $d(n)$

User-selected parameter: η
Initialization. Set $\hat{\mathbf{w}}(0) = \mathbf{0}$.
Computation. For $n = 1, 2, \ldots$, compute

$$e(n) = d(n) - \hat{\mathbf{w}}^T(n)\mathbf{x}(n)$$
$$\hat{\mathbf{w}}(n+1) = \hat{\mathbf{w}}(n) + \eta\mathbf{x}(n)e(n)$$

2. Frameworks/ Libraries

- Python
- matplotlib: used for visualization, plotting the MSE error figure, generating points to plot the double moon, and plotting the classification line.
- numpy: used for processing dataset with multi-dimensional arrays.

3. Task 1

3.1 Approach for task 1

Step 1: Initialize necessary variables described in the textbook

```
a = 0.99
x0 = 0
eta = 0.02
sigu2 = 0.995
variance = 0.01
```

Code

$$a = 0.99$$
$$\sigma_{\varepsilon}^2 = 0.02$$
$$\sigma_x^2 = 0.995$$

Textbook

Step 2: To generate data, consider a generative model which represents an autoregressive (AR) process of order one defined by:

$$x(n) = ax(n-1) + \varepsilon(n)$$

Where:
a is the only parameter of the model, here a = 0.99
epsilon(n) is computed from a zero-mean white-nose process of variance $\sigma_{\varepsilon}^2 = 0.02$

Code:

```
def get_dataset(a):
    # Generative model x(n) = a*x(n-1) + epsilon(n)
    mean, std_dvt = 0, math.sqrt(variance)
    epsilon = np.random.normal(mean, std_dvt, size=1000000)
    x = [0 for i in range(1000000)]
    x[0] = a * x0 + epsilon[0]
    for i in range(1, 1000000):
        if i == 1:
            x[1] = a * x[0] + epsilon[1]
        else:
            x[i] = a * x[i - 1] + epsilon[i]
    # get the last 5000 data points
    x = x[-5000:]
    return x
```

For 100 iterations, generate the dataset consists of 5000 datapoints.

Step 3: For each iteration, compute the error value and weight value.

Calculate the error values based on the formula:

$$e(n) = d(n) - \mathbf{x}^T(n)\hat{\mathbf{w}}(n)$$

Calculate the weights based on the formula:

$$\hat{\mathbf{w}}(n+1) = \hat{\mathbf{w}}(n) + \eta\mathbf{x}(n)e(n)$$

```
# Compute error: e(n) = d(n) - transpose(w) * x(n)
error[0, :] = x_expected[0, :] - x_predict[0, :]

# Compute weight: w(n+1) = w(n) + eta * x(n) * e(n)
w = w + eta * error[0, :].T * x0

# Loop through the rest of the dataset and compute the corresponding weights for each data point
for n in range(1, n_data):
    w0 = np.append(w0, w)
    x_predict[n, :] = x_expected[n-1, :] * w
    error[n, :] = x_expected[n, :] - x_predict[n, :]
    w = w + eta * error[n, :].T * x_expected[n-1, :]
```

Step 4: the learning-rate parameter ŋ is small, the LMS learning curve in theory is

$$J(n) \approx J_{\min} + \frac{\eta J_{\min}}{2}\sum_{k=1}^{M}\lambda_k + \sum_{k=1}^{M}\lambda_k\left(|v_k(0)|^2 - \frac{\eta J_{\min}}{2}\right)(1-\eta\lambda_k)^{2n} \qquad (3.63)$$

The learning curve in experiment is computed by:

$$J(n) = \mathbb{E}[|e(n)|^2]$$

Code:

```
# LMS learning curve: formula 3.63 in the textbook
J_theory = sigu2*(1-a**2)*(1+(eta/2)*sigu2) + sigu2*(a**2+(eta/2)*(a**2)*sigu2-0.5*eta*sigu2)*(1-eta*sigu2)**(2*t)

# LMS mean square error: formula J = E(e(x)**2) in the textbook
J_experiment = np.mean(np.square(errors), axis=0)
```

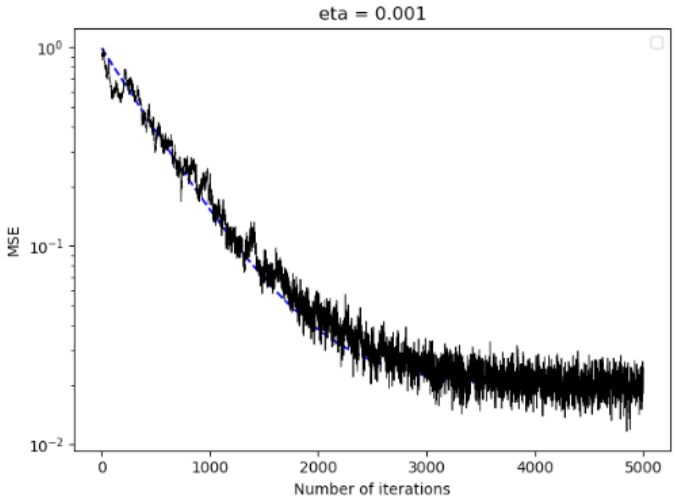
Step 5: Plot the results of the learning curve in theory and in experiment

Code:

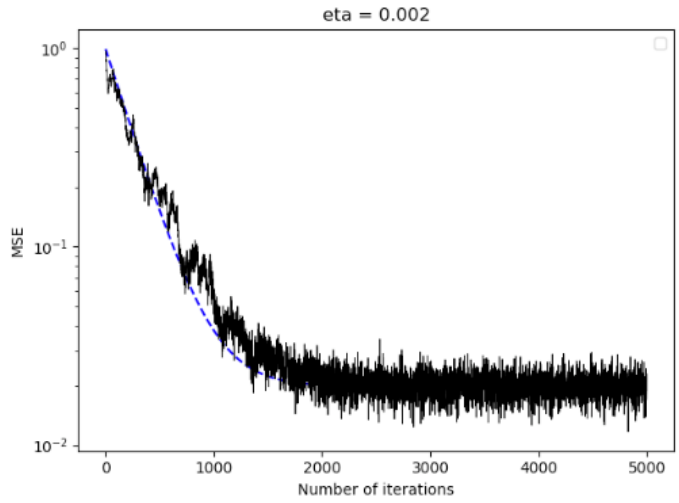
```
def display_result(J_theory, J_experiment):
    plt.legend(loc="upper right")
    plt.semilogy(J_theory, 'b--', label='Theory')
    plt.semilogy(J_experiment, 'k-', label='Experiment', linewidth=0.6)
    plt.title('eta = ' + str(eta))
    plt.xlabel("Number of iterations")
    plt.ylabel("MSE")
    plt.show()
```

3.2 Results for task 1

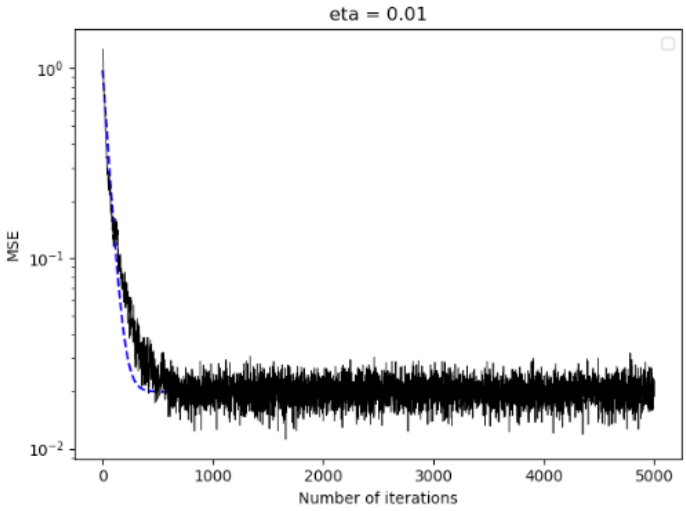
ŋ = 0.001



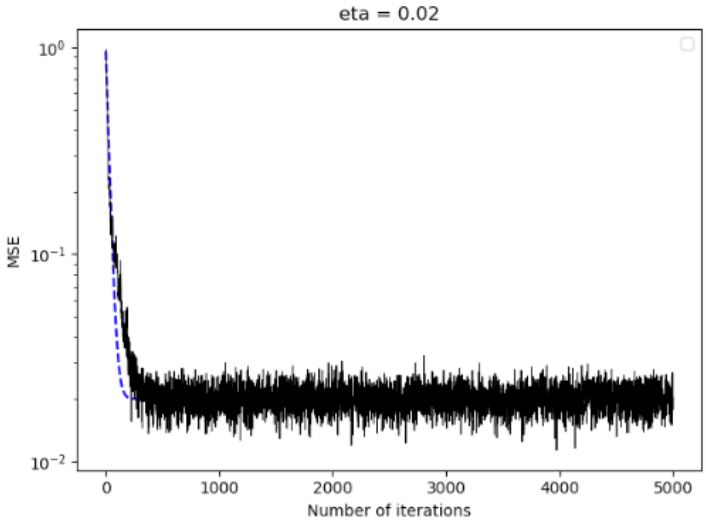
ŋ = 0.002



ŋ = 0.01


eta = 0.01
MSE
Number of iterations
0
1000
2000
3000
4000
5000

ŋ = 0.02

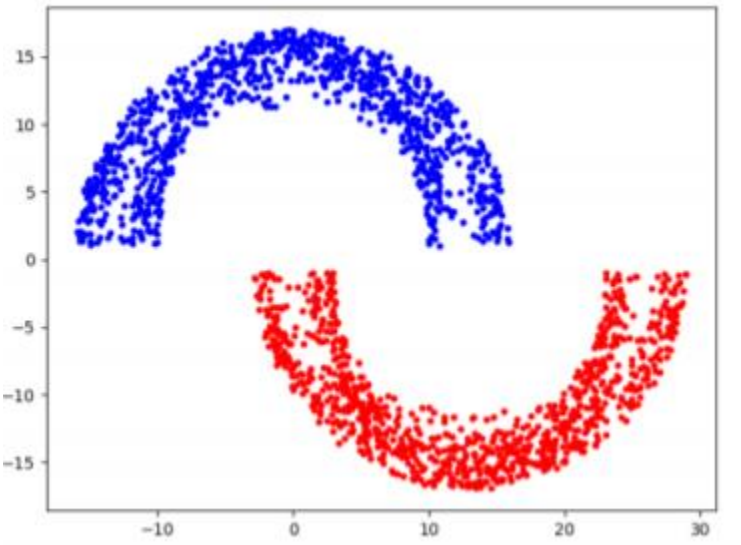

eta = 0.02
MSE
Number of iterations
0
1000
2000
3000
4000
5000

4. Task 2

4.1 Approach

Step 1: Generate moon function as same as previous assignments.

```
def moon(num_points, distance, radius, width):
    points = num_points

    x1 = [0 for _ in range(points)]
    y1 = [0 for _ in range(points)]
    x2 = [0 for _ in range(points)]
    y2 = [0 for _ in range(points)]

    for i in range(points):
        d = distance
        r = radius
        w = width
        a = random() * math.pi
        x1[i] = math.sqrt(random()) * math.cos(a) * (w / 2) + (
                    (-(r + w / 2) if (random() < 0.5) else (r + w / 2)) * math.cos(a))
        y1[i] = math.sqrt(random()) * math.sin(a) * w + (r * math.sin(a)) + d

        a = random() * math.pi + math.pi
        x2[i] = (r + w / 2) + math.sqrt(random()) * math.cos(a) * (w / 2) + (
            (-(r + w / 2)) if (random() < 0.5) else (r + w / 2)) * math.cos(a)
        y2[i] = -(math.sqrt(random()) * math.sin(a) * (-w) + (-r * math.sin(a))) - d
    return [x1, x2, y1, y2]
```



Visualize moon function

Step 2: Generate dataset from moon function and apply normalization on the dataset using: Normalized dataset = (value – mean) / max(value(original dataset)

Code:

```
x1, x2, y1, y2 = moon(total_points, d, 10, 6)
data = []
data.extend([x1[i], y1[i], -1] for i in range(total_points))
data.extend([x2[i], y2[i], 1] for i in range(total_points))
data = np.asarray(data)

# normalize dataset
normalized_data = np.asarray(np.copy(data))
sum_column = np.sum(normalized_data[:, :2], axis=0)
mean_column = np.divide(sum_column, len(normalized_data))
normalized_data[:, 0] = np.subtract(normalized_data[:, 0], mean_column[0])
normalized_data[:, 1] = np.subtract(normalized_data[:, 1], mean_column[1])
max_value = np.amax(abs(normalized_data[:, :2]))
normalized_data[:, 0] = np.divide(normalized_data[:, 0], max_value)
normalized_data[:, 1] = np.divide(normalized_data[:, 1], max_value)
```

Step 3: Train the moon dataset using Least Mean Square algorithm. Instead of applying signum function in project 1 (which produce 0 or 1 value), now keep the original value without using signum function.

For each iteration, compute the MSE value and weight value by the above formula (In task 1) and store the MSE value and weight value to later assessment.
Code:

```
def train(dataset, epochs, eta):
    weights = np.zeros(2)
    mses = []
    for epoch in range(epochs):
        mse = 0.0

        # shuffle the dataset for each epoch
        np.random.shuffle(dataset)
        for row in dataset:
            row_without_label = row[:2]
            row_without_label = np.asarray(row_without_label)
            prediction = np.dot(weights, row_without_label)
            expected = row[-1]
            error = expected - prediction
            mse += error ** 2
            weights = weights + eta*error*row_without_label

        mse = mse / len(dataset)
        mses.append(mse)

        if mse == 0:
            break
    return mses, weights
```

Step 4:

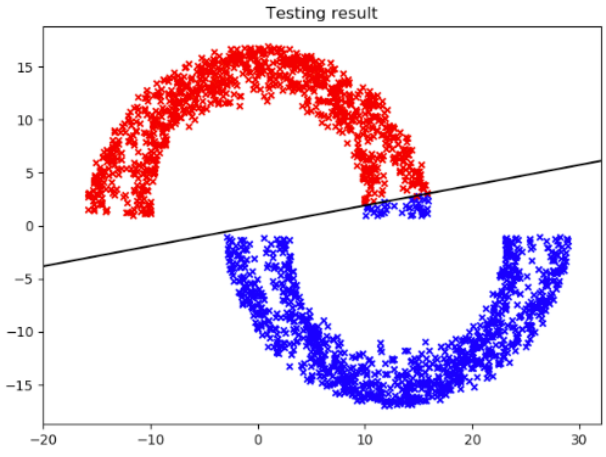
Display the results: learning curve figure and double moon classification figure

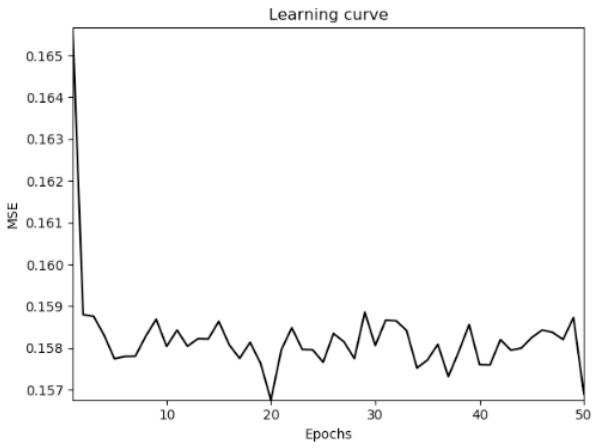
Code:

```
def display_result(weights, mse, dataset):
    label_1_prediction_filter = dataset[:, 0] * weights[0] + dataset[:, 1] * weights[1] >= 0
    label_1_prediction_dataset = dataset[label_1_prediction_filter]
    label_2_prediction_filter = dataset[:, 0] * weights[0] + dataset[:, 1] * weights[1] < 0
    label_2_prediction_dataset = dataset[label_2_prediction_filter]

    n_epoch = 10 if len(mse) <= 10 else len(mse)

    plt.plot(range(1, len(mse)+1), mse, 'k-')
    plt.xlabel("Epochs")
    plt.ylabel("MSE")
    plt.title('Learning curve')
    plt.axis([1, n_epoch, 0, max(mse)])
    plt.ylim(np.min(mse), np.max(mse))
    plt.show()

    x = np.asarray([-20, 32])
    y = (-weights[0] * x)/weights[1]
    plt.plot(x, y, c="k")
    plt.xlim(-20, 32)
    plt.title('Testing result')
    plt.scatter(label_1_prediction_dataset[:, 0], label_1_prediction_dataset[:, 1], c="b", marker='x', s=20)
    plt.scatter(label_2_prediction_dataset[:, 0], label_2_prediction_dataset[:, 1], c="r", marker='x', s=20)
    plt.show()
```

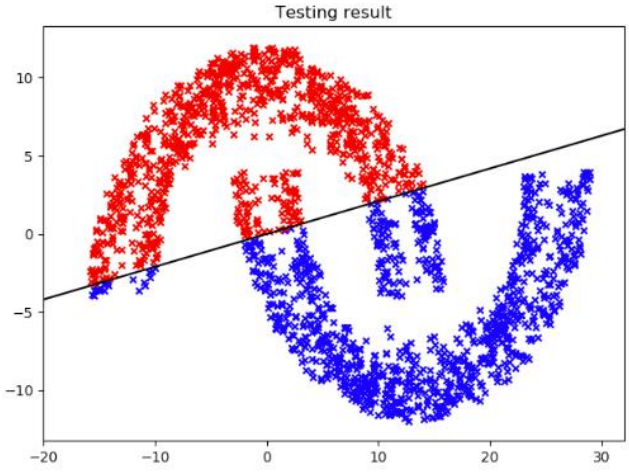
4.2 Result

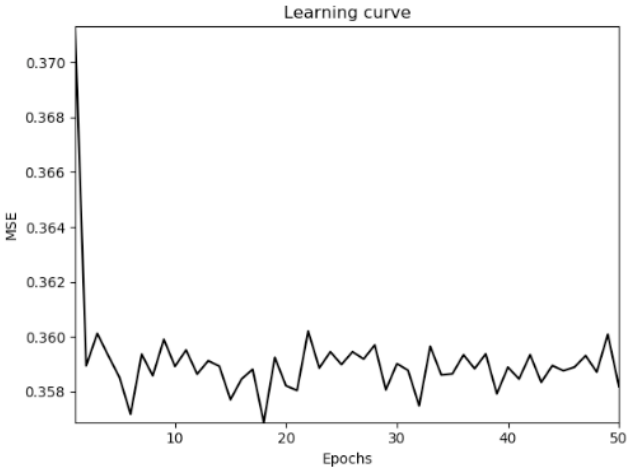
ŋ = 0.01, distance = 1



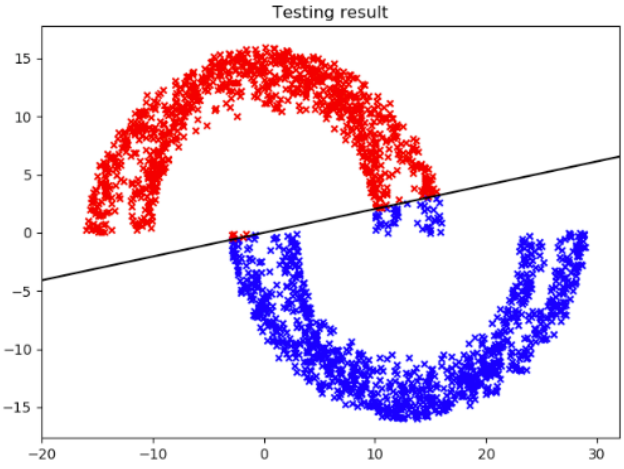


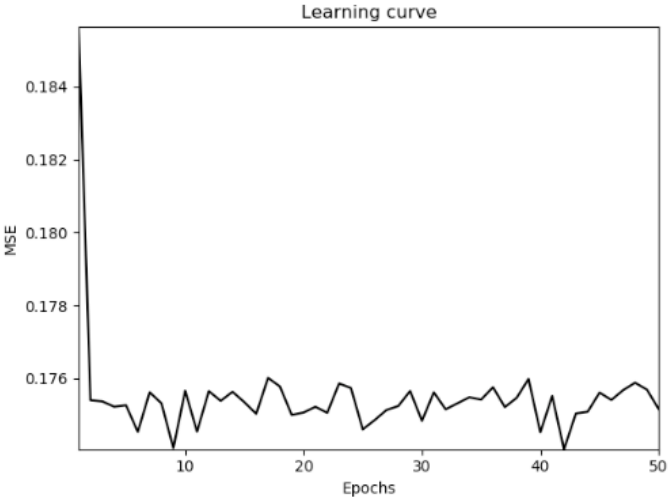
ŋ = 0.01, distance = -4





ŋ = 0.01, distance = 0





5. Task 3

Use the same dataset, then compare the error values among 3 algorithms

Rosenblatt Perceptron algorithm -- MSE: [0.084, 0.02, 0.032, 0.044, 0.016, 0.01, 0.034, 0.03, 0.016, 0.01, 0.01, 0.018, 0.03, 0.016, 0.03, 0.018, 0.02, 0.016, 0.02, 0.008, 0.018, 0.018, 0.018, 0.02, 0.022, 0.022, 0.022, 0.018, 0.022, 0.022, 0.018, 0.022, 0.016, 0.014, 0.008, 0.014, 0.018, 0.02, 0.016, 0.008, 0.014, 0.016, 0.02, 0.014, 0.016, 0.02, 0.016, 0.008, 0.01, 0.016]

Least Square algorithm -- MSE: 0.094

LMS algorithm -- MSE: [0.187 0.178 0.178 0.177 0.177 0.177 0.177 0.177 0.178 0.178 0.177 0.178 0.178 0.177 0.178 0.177 0.178 0.177 0.177 0.177 0.178 0.178 0.178 0.178 0.177 0.178 0.178 0.177 0.178 0.176 0.177 0.177 0.178 0.177 0.177 0.178 0.178 0.178 0.178 0.177 0.178 0.177 0.177 0.178 0.177 0.178 0.178 0.178 0.177 0.177]

Overall:

The errors in Rosenblatt's Perceptron range from 0.008 to 0.084

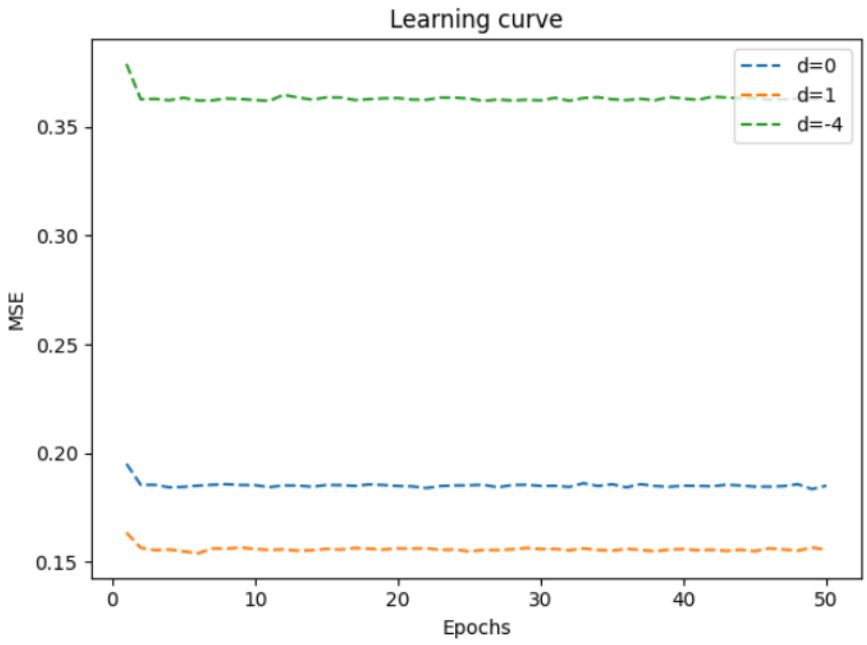
The error in Least Square algorithm is 0.094

The errors in Least Mean Square algorithm range from 0.176 to 0.187

Shortly, among 3 algorithms, Rosenblatt's Perceptron model has the best performance and Least Mean Square algorithm model has the worst performance.

6. Task 4

Plot the pattern-classification learning curves of the least-mean-square algorithm applied to the double-moon configuration for the following values assigned to the distance of separation: d = 1, d = 1, d = -4



Compare the results of the experiment with the corresponding ones obtained using Rosenblatt's perceptron.

```
-------------Distance d=0---------------
Rosenblatt Perceptron algorithm -- MSE:  [0.068 0.022 0.03  0.02  0.034 0.014 0.014 0.012 0.014 0.024 0.026 0.012
 0.008 0.026 0.012 0.006 0.038 0.016 0.012 0.01  0.028 0.014 0.012 0.012
 0.03  0.008 0.02  0.01  0.016 0.026 0.008 0.024 0.008 0.024 0.008 0.026
 0.008 0.024 0.008 0.024 0.012 0.008 0.02  0.01  0.008 0.02  0.012 0.008
 0.024 0.012]
LMS algorithm -- MSE:  [0.1951 0.1854 0.1854 0.1843 0.1845 0.1851 0.1855 0.1857 0.1854 0.1853
 0.1844 0.1852 0.1851 0.1846 0.1854 0.1853 0.1849 0.1856 0.1854 0.1849
 0.1848 0.184  0.1849 0.1852 0.1852 0.1855 0.1843 0.1854 0.1855 0.185
 0.185  0.1845 0.1862 0.1849 0.1857 0.1843 0.1857 0.1849 0.1846 0.1851
 0.185  0.1848 0.1855 0.1851 0.1846 0.1846 0.1849 0.1857 0.1835 0.1851]
-------------Distance d=1---------------
Rosenblatt Perceptron algorithm -- MSE:  [0.004 0.   ]
LMS algorithm -- MSE:  [0.1635 0.1565 0.1554 0.1557 0.1549 0.154  0.1562 0.1561 0.1566 0.156
 0.1555 0.1558 0.1552 0.1554 0.156  0.1557 0.1564 0.1561 0.1557 0.1562
 0.1562 0.1563 0.1556 0.1558 0.1548 0.1555 0.1555 0.1558 0.1564 0.1559
 0.156  0.1554 0.1562 0.1556 0.1552 0.156  0.1555 0.1549 0.1557 0.1559
 0.1554 0.1556 0.1551 0.1557 0.155  0.1563 0.1557 0.1552 0.1566 0.1556]
-------------Distance d=-4---------------
Rosenblatt Perceptron algorithm -- MSE:  [0.146 0.236 0.302 0.238 0.24  0.114 0.25  0.122 0.258 0.248 0.258 0.274
 0.282 0.26  0.248 0.148 0.128 0.236 0.278 0.23  0.254 0.238 0.254 0.128
 0.258 0.27  0.1   0.172 0.272 0.326 0.244 0.236 0.256 0.262 0.132 0.31
 0.146 0.25  0.158 0.274 0.274 0.312 0.238 0.236 0.12  0.104 0.322 0.25
 0.24  0.266]
LMS algorithm -- MSE:  [0.3789 0.3626 0.3628 0.3621 0.3633 0.362  0.3621 0.363  0.3627 0.3623
 0.3619 0.3646 0.3634 0.3624 0.3635 0.3634 0.3622 0.3626 0.363  0.3632
 0.3625 0.3624 0.3634 0.3633 0.3629 0.3618 0.3625 0.362  0.3624 0.3621
 0.3633 0.3619 0.3632 0.3635 0.3626 0.3622 0.3628 0.3621 0.3636 0.3629
 0.3623 0.3638 0.3634 0.3631 0.3632 0.3625 0.3625 0.3631 0.363  0.3626]
```

7. Conclusion

- Learn to implement the computer experiment validating the small learning-rate theory of the LMS algorithm.
- Implement the repeats of the pattern-classification experiment of Section 1.5 on the perceptron, this time using the LMS algorithm.
- The LMS algorithm is simple to formulate and simple to implement.
- The LMS algorithm is effective in performance.
- The LMS algorithm is efficient in complexity (linear law).
- The LMS algorithm is model independent and robust.

8. References

Wikipedia, Least mean squares filter
https://en.wikipedia.org/wiki/Least_mean_squares_filter

Haykin, Simon (2009), Neural Networks and Learning Machines

Wikipedia, Mean squared error

https://en.wikipedia.org/wiki/Mean_squared_error