# EE5907 Pattern Recognition: Assignment #1

XU NUO, A0313771H

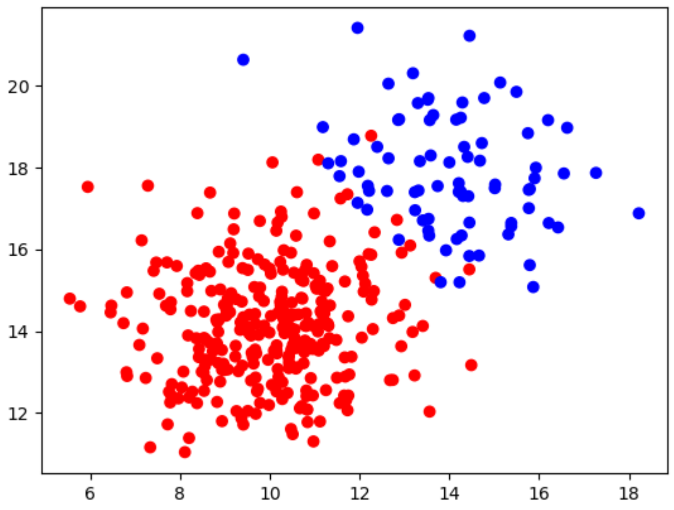
**Part 1. Classification with MLP Updated by Backpropagation Following Gradient Descent**

**a. Data Generation**

```python
# Code used to generate 2D data following a normal distribution
# visualize it as a scatter plot with color-coded classes, then save the dataset
# Import necessary libraries
import matplotlib.pyplot as plt
import numpy as np
import scipy.io as io

N1 = 300 # Number of samples in class 1
N2 = 80 # Number of samples in class 2
K = 2 # Feature dimension
sigma = 2 # Data dispersion

# Generate corresponding data belonging to class 1
mean1 = (10, 14)
cov1 = np.array([[sigma, 0], [0, sigma]])
X1 = np.random.multivariate_normal(mean1, cov1, N1)
c1 = ['red'] * len(X1)

# Generate corresponding data belonging to class 2
mean2 = (14, 18)
cov2 = np.array([[sigma, 0], [0, sigma]])
X2 = np.random.multivariate_normal(mean2, cov2, N2)
c2 = ['blue'] * len(X2)

# Data concatenation
X = np.concatenate((X1, X2))
color = np.concatenate((c1, c2))

# Define the class labels for generated data points.
T = []
for n in range(0, len(X)):
    if n < len(X1):
        T.append(0)
    else:
        T.append(1)

# Plot X1 and X2
plt.scatter(X[:, 0], X[:, 1], marker = 'o', c = color)
plt.show()
```

```
# Save the generated data
np.save('class1.npy', X1)
np.save('class2.npy', X2)
io.savemat('class1.mat', {'class1': X1})
io.savemat('class2.mat', {'class2': X2})
```

For data generation, I used the code provided by the assignment, namely 'Generate_data.py'. I added comments in the code to facilitate understanding and reading. Each data point has two features and is normally distributed. Class 1 has 300 data points with a mean of (10, 14), marked as red dots in the figure; Class 2 has 80 data points with a mean of (14, 18), marked as blue dots in the figure. The figure is shown below:

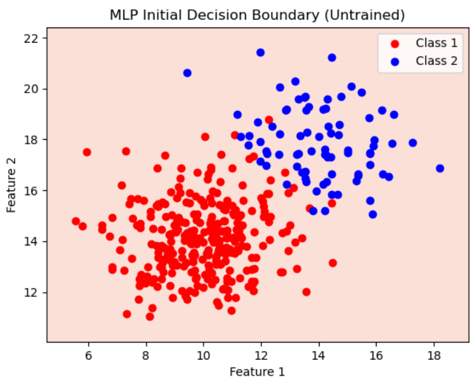


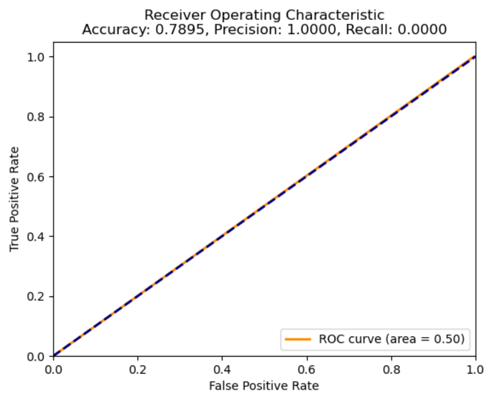
**b. Multi-Layer Perceptron Implementation (Untrained)**

```
# Import necessary libraries
import torch
import numpy as np
import matplotlib.pyplot as plt
from sklearn.metrics import accuracy_score, precision_score, recall_score, roc_curve,
auc

# Define an MLP class to implement a neural network for binary classification task
class MLP:
    def __init__(self):
        # Manually define weights, biases, and intermediate variables
        self.h2 = None
        self.z2 = None
        self.h1 = None
        self.z1 = None
        # Constrain initial weights to avoid gradient vanishing
        self.W1 = torch.randn(2, 3) * 0.1 # Two input neurons/One hidden layer with three
neurons
        self.b1 = torch.zeros(1, 3) # The bias of hidden layer
        self.W2 = torch.randn(3, 1) * 0.1 # One output neuron
        self.b2 = torch.zeros(1, 1) # The bias of output layer

    def forward(self, x):
        # Perform manual forward propagation from scratch
        self.z1 = x @ self.W1 + self.b1 # Calculate the input of hidden layer
        self.h1 = torch.relu(self.z1) # Activate hidden layer
        self.z2 = self.h1 @ self.W2 + self.b2 # Calculate the input of output layer
        self.h2 = torch.sigmoid(self.z2) # Activate output layer
        return self.h2

# Instantiate the MLP
mlp = MLP()

# Load the pre-generated data and create corresponding labels
X1 = np.load('class1.npy')
X2 = np.load('class2.npy')
X_concat = np.concatenate((X1, X2))
Y_concat = np.concatenate((np.zeros(X1.shape[0]), np.ones(X2.shape[0])))

# Generate a grid of coordinates based on the two features X1 and X2
x_min, x_max = X_concat[:, 0].min() - 1, X_concat[:, 0].max() + 1
y_min, y_max = X_concat[:, 1].min() - 1, X_concat[:, 1].max() + 1
res = 0.01
xx, yy = np.meshgrid(np.arange(x_min, x_max, res), np.arange(y_min, y_max, res))
grid_points = np.c_[xx.ravel(), yy.ravel()].astype(np.float32)
grid_tensor = torch.tensor(grid_points)

# Use the untrained MLP model to infer the classification results of the grid
coordinates
with torch.no_grad(): # Disable gradient calculation
    Z = mlp.forward(grid_tensor).numpy()
    Z = Z.reshape(xx.shape)

# Plot the decision boundary
plt.contourf(xx, yy, Z, levels=[0, 0.5, 1], cmap = "coolwarm_r", alpha = 0.3)
'''
'Contourf' is used to draw contour plots, and 'levels' to define the contour level.
'Cmap' is used to represent the mapping of colors and 'alpha' represents transparency
'''
```

```python
# Plot the feature points X1 and X2
plt.scatter(X1[:, 0], X1[:, 1], color = 'red', label = "Class 1")
plt.scatter(X2[:, 0], X2[:, 1], color = 'blue', label = "Class 2")

plt.legend()
plt.title("MLP Initial Decision Boundary (Untrained)")
plt.xlabel("Feature 1")
plt.ylabel("Feature 2")
plt.show(block = False)

# Use the untrained MLP model to infer the classification results of the input features
with torch.no_grad():
    X_concat_tensor = torch.tensor(X_concat, dtype = torch.float32)
    Y_pred = mlp.forward(X_concat_tensor).numpy()
    Y_pred_class = (Y_pred > 0.5).astype(int) # Convert probability to categorical labels

# Calculate classification metrics
accuracy = accuracy_score(Y_concat, Y_pred_class)
precision = precision_score(Y_concat, Y_pred_class, zero_division = 1)
recall = recall_score(Y_concat, Y_pred_class, zero_division = 1)

# Calculate and plot the ROC curve
fpr, tpr, thresholds = roc_curve(Y_concat, Y_pred)
roc_auc = auc(fpr, tpr)

plt.figure()
plt.plot(fpr, tpr, color = 'darkorange', lw = 2, label = 'ROC curve (area = %0.2f)' %
roc_auc)
plt.plot([0, 1], [0, 1], color = 'navy', lw = 2, linestyle = '--')
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.05])
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title(
    f"Receiver Operating Characteristic\nAccuracy: {accuracy:.4f}, Precision:
{precision:.4f}, Recall: {recall:.4f}")
plt.legend(loc = "lower right")
plt.show()
```

In this part, I defined an MLP as required by the assignment. It has two inputs, a hidden layer with three neurons, and an output. I initialized the weights of the hidden and output layers following normal distribution and initialized the biases of the hidden and output layers to 0. I also implemented forward propagation from scratch based on the principles of neural networks. ReLU and Sigmoid are used as activation functions for the hidden layer and output layer respectively. The decision boundary and ROC curve plotted for the input data without training the model are shown in the figures below:





When the model is not trained, due to the random distribution of initial weights, the model has not yet learned how to classify, all data will be randomly classified, so the decision boundary is also random.
According to the results of this code run, all data are classified as Class 1, and the decision boundary is not within the given coordinate range. The accuracy of the model's classification is about 0.79, the precision is 1, and the recall is 0.
Accuracy is defined as the proportion of correctly predicted samples to all samples. All samples in the figure are predicted to be Class 1, which gives

$$Accuracy = \frac{TP + TN}{TP + FP + TN + FN} = \frac{300}{380} \approx 0.79$$

Precision is defined as the proportion of samples that are actually positive among those predicted by the model to be positive samples. All samples in the figure are predicted to be Class 1 whose ground-truth output is 0, which means that the model doesn’t predict the sample as Class 2 at all, which gives

$$Precision = \frac{TP}{TP + FP} = \frac{0}{0 + 0}$$

Although the code sill outputs a precision of 1 in this case, the precision is not measurable in reality.
Recall is defined as the proportion of samples that the model correctly predicts to be positive among all samples that are actually positive, which gives

$$Recall = \frac{TP}{TP + FN} = \frac{0}{0 + 80} = 0$$

The ROC curve is a diagonal line, and the AUC is 0.5, indicating that the model has the same effect as a random classifier and cannot effectively distinguish between Class 1 and Class 2.

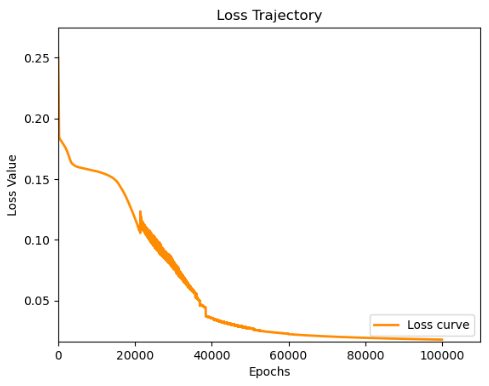
### c. Multi-Layer Perceptron Implementation (Trained)

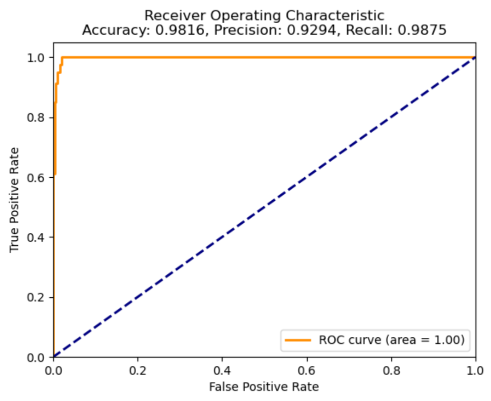
```
# Import necessary libraries
import torch
import numpy as np
import matplotlib.pyplot as plt
from sklearn.metrics import accuracy_score, precision_score, recall_score, roc_curve,
auc
```

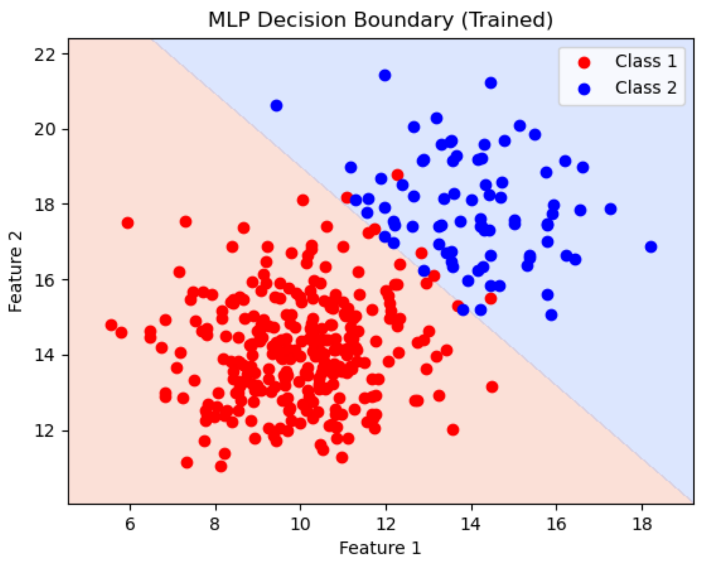
```python
# Define an MLP class to implement a neural network for binary classification task
class MLP:
    def __init__(self):
        # Manually define weights, biases, and intermediate variables
        self.z1 = None
        self.h1 = None
        self.z2 = None
        self.h2 = None
        # Constrain initial weights to avoid gradient vanishing
        self.W1 = torch.randn(2, 3) * 0.1 # Two input neurons/One hidden layer with three
neurons
        self.b1 = torch.zeros(1, 3) # The bias of hidden layer
        self.W2 = torch.randn(3, 1) * 0.1 # One output neuron
        self.b2 = torch.zeros(1, 1) # The bias of output layer

    def forward(self, x):
        # Perform manual forward propagation from scratch
        self.z1 = x @ self.W1 + self.b1 # Calculate the input of hidden layer
        self.h1 = torch.relu(self.z1) # Activate hidden layer
        self.z2 = self.h1 @ self.W2 + self.b2 # Calculate the input of output layer
        self.h2 = torch.sigmoid(self.z2) # Activate output layer
        return self.z1, self.h1, self.z2, self.h2

# Derivative calculation function of the ReLU function
def relu_derivative(x):
    return (x > 0).float()

# Derivative calculation function of the Sigmoid function
def sigmoid_derivative(x):
    sig = torch.sigmoid(x)
    return sig * (1 - sig)

# Perform manual backpropagation from scratch
def back_propagation(model, x, y, lr):
    z1, h1, z2, h2 = model.forward(x)
    y_predict = h2
    loss_value = (y_predict - y) ** 2 # Use squared error as the loss function
    d_loss = 2 * (y_predict - y) # Calculate the gradient of the loss with respect to the
output

    delta_out = d_loss * sigmoid_derivative(z2) # Calculate the output layer error
    dW2 = h1.T @ delta_out / len(x) # Calculate the gradient of the output layer weights
    db2 = torch.mean(delta_out, dim = 0)
    delta_hidden = delta_out @ model.W2.T * relu_derivative(z1) # Calculate the hidden
layer error
    dW1 = x.T @ delta_hidden / len(x) # Calculate the gradient of the hidden layer weight
    db1 = torch.mean(delta_hidden, dim = 0)

    # Update the weights and biases
    model.W2 -= lr * dW2
    model.b2 -= lr * db2
    model.W1 -= lr * dW1
    model.b1 -= lr * db1

    return loss_value.mean().item()

# Load the pre-generated data and create corresponding labels
X1 = np.load('class1.npy')
X2 = np.load('class2.npy')
X_concat = np.concatenate((X1, X2))
Y_concat = np.concatenate((np.zeros(X1.shape[0]), np.ones(X2.shape[0])))
X_tensor = torch.tensor(X_concat, dtype = torch.float32)
Y_tensor = torch.tensor(Y_concat, dtype = torch.float32).view(-1, 1)

# Instantiate the MLP
mlp = MLP()

# Define hyperparameters
epochs = 100000
learning_rate = 0.01
loss_list = []

# Model training
for epoch in range(epochs):
    loss = back_propagation(mlp, X_tensor, Y_tensor, learning_rate)
    loss_list.append(loss)
    if epoch % 10000 == 0:
        print(f'Epoch {epoch}, Loss: {loss:.4f}')

# Generate a grid of coordinates based on the two features X1 and X2
x_min, x_max = X_concat[:, 0].min() - 1, X_concat[:, 0].max() + 1
y_min, y_max = X_concat[:, 1].min() - 1, X_concat[:, 1].max() + 1
res = 0.01
xx, yy = np.meshgrid(np.arange(x_min, x_max, res), np.arange(y_min, y_max, res))
grid_points = np.c_[xx.ravel(), yy.ravel()].astype(np.float32)
grid_tensor = torch.tensor(grid_points)
```

```
# Use the trained MLP model to infer the classification results of the grid coordinates
_, _, _, grid_pred = mlp.forward(grid_tensor)
Z = grid_pred.detach().numpy().reshape(xx.shape)

# Plot the decision boundary
plt.contourf(xx, yy, Z, levels = [0, 0.5, 1], cmap = "coolwarm_r", alpha=0.3)

# Plot the feature points X1 and X2
plt.scatter(X1[:, 0], X1[:, 1], color = 'red', label = "Class 1")
plt.scatter(X2[:, 0], X2[:, 1], color = 'blue', label = "Class 2")
plt.legend()
plt.title("MLP Decision Boundary (Trained)")
plt.xlabel("Feature 1")
plt.ylabel("Feature 2")
plt.show()

# Data classification
_, _, _, y_pred = mlp.forward(X_tensor)
Y_pred = y_pred.detach().numpy()
Y_pred_class = (Y_pred > 0.5).astype(int)

# Calculate classification metrics
accuracy = accuracy_score(Y_concat, Y_pred_class)
precision = precision_score(Y_concat, Y_pred_class, zero_division = 1)
recall = recall_score(Y_concat, Y_pred_class, zero_division = 1)

# Calculate and plot the ROC curve
fpr, tpr, thresholds = roc_curve(Y_concat, Y_pred)
roc_auc = auc(fpr, tpr)

plt.figure()
plt.plot(fpr, tpr, color = 'darkorange', lw=2, label = 'ROC curve (area = %0.2f)' %
roc_auc)
plt.plot([0, 1], [0, 1], color = 'navy', lw=2, linestyle = '--')
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.05])
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title(f"ROC Curve\nAccuracy: {accuracy:.4f}, Precision: {precision:.4f}, Recall:
{recall:.4f}")
plt.legend(loc = "lower right")

# Plot the loss curve
plt.figure()
plt.plot(range(epochs), loss_list, color = 'darkorange', lw = 2, label = 'Loss curve')
plt.xlim([0.0, epochs * 1.1])
plt.ylim([min(loss_list) * 0.9, max(loss_list) * 1.1])
plt.xlabel('Epochs')
plt.ylabel('Loss Value')
plt.title('Loss Trajectory')
plt.legend(loc = "lower right")
plt.show()
```

Based on the existing code, I defined the derivative functions of ReLU and Sigmoid functions, used squared error as the loss function and implemented manual backpropagation from scratch to update the weights and bias values to achieve the purpose of model training. The loss trajectory, ROC curve and the new decision boundary are plotted as shown below.







The decision boundary of the untrained MLP is random and has no classification ability. After the model is trained, it can classify sample points very accurately, with a clear decision boundary that conforms to the distribution of the samples. The accuracy of the model's classification now is about 0.98, the precision is about

0.93, and the recall is about 0.99. This proves that the weights and biases are optimized by gradient descent to form a more reasonable feature map.

The ROC curve of the untrained MLP is close to $y = x$, proving that the classification ability of the model is equivalent to random guessing. The ROC curve after training is obviously bent to the upper left, and the AUC is increased to close to 1, indicating that the trained MLP has better classification ability and can effectively distinguish Class 1 and Class 2.

**Some questions and guesses about Part 1.**

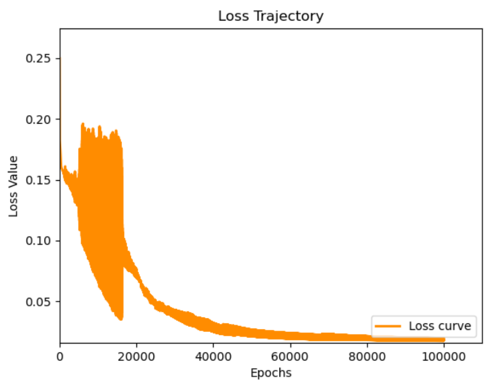
In the code, I defined the learning rate as 0.01 and epoch as 100,000. But I did not initially set such a small learning rate and such a large epoch.

I set the learning rate to the most common 0.1 at first, and the number of epochs to less than 5000 rounds. Because I thought this classification was relatively simple and didn't require so many iterations. However, when the learning rate was set to 0.1, the loss value would continue to oscillate instead of gradually decreasing. I think the violent oscillation is caused by the learning rate being too large, which leads to the gradient update step being too long and skipping the optimal point. So, I reduced the learning rate and this improved a lot.

Thousands of epochs still failed to make the model converge, and even the model classification was still completely wrong. When I adjusted the training cycle to tens of thousands of rounds, I was able to complete the classification. As for the reasons for slow training, I think there are several reasons:

1. Low learning rate.
2. The weight update value is the mean of the number of sample points, resulting in a small update value.
3. The sigmoid function causes the gradient to disappear.
4. The categories of the sample points are unbalanced, and the direction of weight update is limited.

The loss curve when the learning rate is 0.1 is shown in the figure below. It can be clearly seen that there is a violent oscillation and a slow downward trend.



For MLP with hidden layers, the decision boundary should be nonlinear, but from the results, the decision boundary is still linear. I think this is because the sample points themselves have very little overlap, and the linear decision boundary is sufficient to complete the classification task with a very low loss value.

**Part 2. Classification with RBF Network with Random RBF Centers**

**a. RBF Network with 3 Random RBF Centers**

```
# Import necessary libraries
import numpy as np
```

```python
import matplotlib.pyplot as plt
from sklearn.metrics import accuracy_score, precision_score, recall_score

# Set the seed of the NumPy random number generator to ensure reproducibility of results
np.random.seed(40)

# Load the pre-generated data and create corresponding labels.
X1 = np.load('class1.npy')
X2 = np.load('class2.npy')
X = np.concatenate((X1, X2))
Y = np.concatenate((np.zeros(X1.shape[0]), np.ones(X2.shape[0])))

num_centers = 3 # Number of RBF centers
random_indices = np.random.choice(len(X), num_centers, replace = False) # Randomly
select corresponding indices from all X as RBF centers without replacement
rbf_centers = X[random_indices]
sigma = 1  # The Sigma parameter of the Gaussian RBF kernel

# Define Gaussian RBF function
def rbf_function(x, rbf_center, sig):
    return np.exp(-np.linalg.norm(x - rbf_center, axis = 1) ** 2 / (2 * sig ** 2))

# Calculate the RBF values for each data point to all RBF centers
H = np.zeros((len(X), num_centers)) # Create the hidden layer output matrix of the RBF
network with shape (380, 3)
for i in range(num_centers):
    H[:, i] = rbf_function(X, rbf_centers[i], sigma)

# Calculate the weight matrix using the standard solution of Least Squares Estimation
# Use pseudoinverse matrix to increase robustness
W = np.linalg.pinv(H.T @ H) @ H.T @ Y

# Print RBF centers
print("RBF centers:")
for i, center in enumerate(rbf_centers):
    print(f"center {i + 1}: {center}")

# Data classification
Y_pred = H @ W
Y_pred_class = (Y_pred > 0.5).astype(int)

# Calculate classification metrics
accuracy = accuracy_score(Y, Y_pred_class)
precision = precision_score(Y, Y_pred_class)
```

```
recall = recall_score(Y, Y_pred_class)

# Generate a grid of coordinates based on the two features X1 and X2
x_min, x_max = X[:, 0].min() - 1, X[:, 0].max() + 1
y_min, y_max = X[:, 1].min() - 1, X[:, 1].max() + 1
res = 0.01
xx, yy = np.meshgrid(np.arange(x_min, x_max, res), np.arange(y_min, y_max, res))
grid_points = np.c_[xx.ravel(), yy.ravel()]

# Calculate the RBF network's predictions for the grid points
H_grid = np.zeros((len(grid_points), num_centers))
for i in range(num_centers):
    H_grid[:, i] = rbf_function(grid_points, rbf_centers[i], sigma)
Z = H_grid @ W
Z = Z.reshape(xx.shape)

# Plot the decision boundary
plt.contourf(xx, yy, Z, levels = np.linspace(Z.min(), Z.max(), 50), cmap = "coolwarm_r",
alpha = 0.3)
plt.contour(xx, yy, Z, levels = [0.5], colors = 'k', linestyles = '-', linewidths = 2)
plt.scatter(X1[:, 0], X1[:, 1], color = 'red', label = "Class 1")
plt.scatter(X2[:, 0], X2[:, 1], color = 'blue', label = "Class 2")

for i, center in enumerate(rbf_centers):
    plt.scatter(center[0], center[1], color = 'black', marker = 'x', s = 100) # Plot RBF
centers
    circle = plt.Circle((center[0], center[1]), sigma * 2, color = 'black', fill = False,
linestyle = '--', alpha = 0.5)
    # Draw the influence range of each RBF center with a dashed circle, radius 2 * sigma
    plt.gca().add_patch(circle)
    plt.text(center[0] + 0.1, center[1] + 0.1, f"w = {W[i]:.4f}", fontsize = 12) #
Annotate weight values

plt.legend()
plt.title(
    f"RBF Network Decision Boundary (3 Centers)\nAccuracy: {accuracy:.4f}, Precision:
{precision:.4f}, Recall: {recall:.4f}")
plt.xlabel("Feature 1")
plt.ylabel("Feature 2")
plt.show()
```

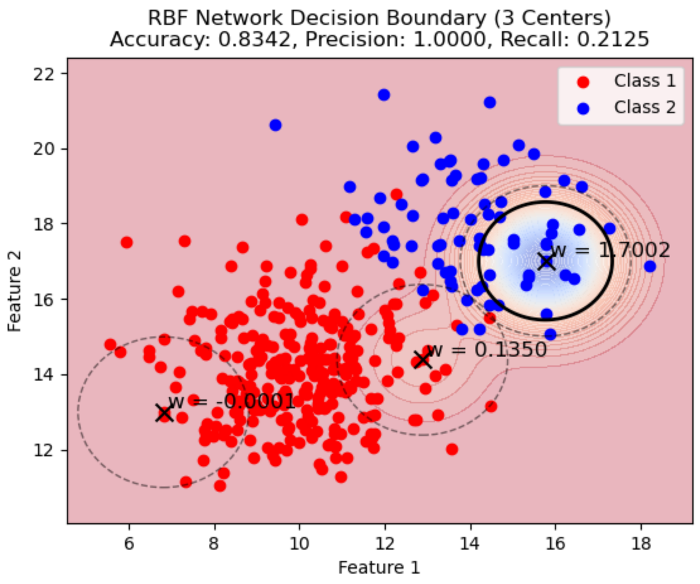
In this part, I randomly selected 3 points from all the sample points as the RBF centers and set the sigma of the Gaussian function to 1. After that, the RBF was defined according to the formula

$$\phi(x) = e^{-\frac{||x-c||^2}{2\sigma^2}}$$

and the sample classification was predicted based on the distance from each sample point to the RBF centers and the weights using the standard solution of the least squares method

$$W = (X^T X)^{-1} X^T Y = (X^T X)^{+} X^T Y \quad (if\ not\ invertible)$$

The figure is shown below:



The black crosses in the figure represent 3 randomly selected RBF centers. The black dotted circles represent the influence range of the RBF kernel function and the black solid circle is the decision boundary. The contour line represents the value of the RBF network output ranging from 0 to 1. The marked *w* value is the weight of each RBF neuron. The classification accuracy is about 0.83, the precision is 1 and the recall is about 0.21.

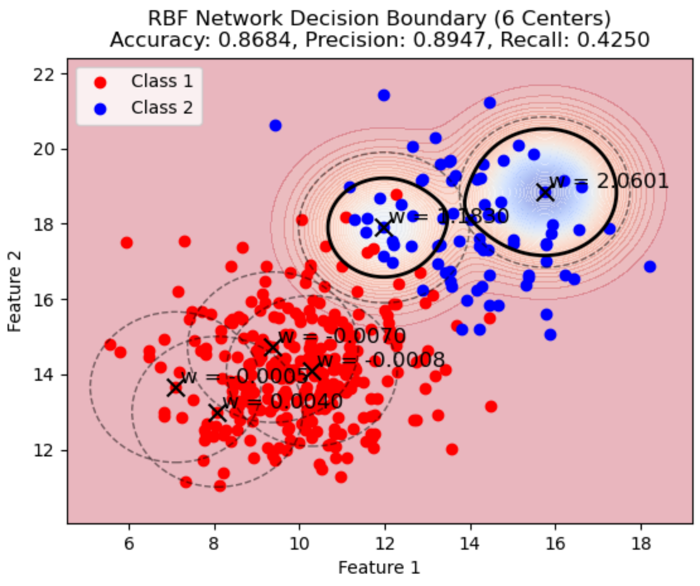
The RBF center in the upper right corner $w = 1.7002$ dominates the classification decision, and its weight is much larger than the other two centers, indicating that it contributes the most to the classification. Its influence is large, causing the blue area to be more concentrated around it. The middle one $w = 0.1350$ has less influence. This center is located at the boundary of the red and blue sample points. It has limited contribution to classification because the center affects both red and blue points, resulting in its weight being weakened. The RBF center in the lower left corner $w = -0.0001$ has almost no contribution to the classification.

Compared with the updated MLP, the classification performance of the RBF network is relatively worse. If trained sufficiently, MLP can find the decision boundary of the data well. It can also achieve non-linear decision boundary through hidden layers. However, the RBF network relies on the selection of RBF centers. In this case, we only have 3 centers and they are randomly chosen. They can't cover the data distribution well.

In terms of training speed, the RBF network has a fast training speed because it only needs to calculate the least squares method once. MLP requires multiple rounds of training, but can learn data distribution more fully, and is suitable for tasks with higher data complexity.

**b. RBF Network with 6 Random RBF Centers**

There is almost no difference in the code between 3 RBF centers and 6 RBF centers. The figure is shown below:



The accuracy of the classification is about 0.87, the precision is about 0.89, the recall is about 0.43. The improved accuracy indicated that after adding RBF centers, the model's ability to fit the data has been enhanced. The improved recall means that more Class 2 are correctly classified and the RBF network is no longer heavily biased towards Class 1. Compared to 3 centers, the influence of the entire decision area is more balanced using 6 centers. However, since the RBF centers are randomly selected, the actual effective centers are still few. It would be better to use K-means to select highly representative points.