

**รายงานการทดลอง**

การ Train Multilayer Perceptron โดยใช้ Genetic Algorithms

**โดย**

นายณปพน วงค์คม

650610834

**เสนอ**

รศ.ดร. ศันสนีย์ เอื้อพันธ์วิริยะกุล

รายงานนี้เป็นส่วนหนึ่งของรายวิชา 261456

สาขาวิชาวิศวกรรมหุ่นยนต์และปัญญาประดิษฐ์

ภาคเรียนที่ 1 ปีการศึกษา 2567

มหาวิทยาลัยเชียงใหม่

**บทคัดย่อ**

การทดลองนี้มุ่งศึกษาการประยุกต์ใช้ Genetic Algorithms (GA) ในการฝึกฝนโครงข่ายประสาทเทียมแบบ Multilayer Perceptron (MLP) เพื่อการวินิจฉัยมะเร็งเต้านมจากชุดข้อมูล Wisconsin Diagnostic Breast Cancer (WDBC) ซึ่งประกอบด้วยคุณลักษณะทางกายภาพ 30 ประการของเซลล์เนื้องอก การทดลองนี้ให้ความสำคัญกับการออกแบบโครงสร้างของ Chromosome การกำหนดค่าพารามิเตอร์ที่เหมาะสม และการวิเคราะห์ประสิทธิภาพในการจำแนกประเภทของเนื้องอก

ผลการทดลองแสดงให้เห็นว่า GA สามารถค้นหาค่า Weights ที่เหมาะสมสำหรับ MLP ได้อย่างมีประสิทธิภาพ โดยให้ความแม่นยำในการจำแนกสูงถึง 96.2% บนชุดข้อมูลทดสอบ การวิเคราะห์พฤติกรรมการลู่เข้าพบว่ามีการพัฒนาที่ชัดเจนใน 3 ช่วง ได้แก่ การสำรวจพื้นที่คำตอบอย่างกว้าง การปรับแต่งในบริเวณที่มีแนวโน้มดี และการปรับแต่งละเอียดจนลู่เข้าสู่คำตอบที่เหมาะสม นอกจากนี้ยังพบว่าการเลือกใช้โครงสร้าง MLP และพารามิเตอร์ของ GA ที่เหมาะสมมีผลอย่างมากต่อประสิทธิภาพของระบบ

**บทนำ**

**ที่มาและความสำคัญ**

โครงข่ายประสาทเทียมแบบหลายชั้น (Multilayer Perceptron: MLP) เป็นแบบจำลองการเรียนรู้เชิงลึกที่ได้รับความนิยมอย่างแพร่หลายในการแก้ปัญหาการจำแนกประเภทและการประมาณค่าที่ซับซ้อน โดยการทำงานของ MLP จะอาศัยการปรับค่าน้ำหนักในแต่ละชั้นของโครงข่าย เพื่อให้สามารถสร้างฟังก์ชันการแปลงข้อมูลที่เหมาะสมสำหรับงานที่ต้องการ การฝึกสอนโครงข่ายให้มีประสิทธิภาพจึงเป็นขั้นตอนที่มีความสำคัญอย่างยิ่ง

ขั้นตอนวิธีเชิงพันธุกรรม (Genetic Algorithms: GA) เป็นวิธีการค้นหาคำตอบที่ได้รับแรงบันดาลใจมาจากทฤษฎีวิวัฒนาการทางชีววิทยา โดยจำลองกระบวนการคัดเลือกตามธรรมชาติผ่านการดำเนินการทางพันธุกรรม เช่น การครอสโอเวอร์และการกลายพันธุ์ วิธีการนี้มีความสามารถในการค้นหาคำตอบที่เหมาะสมในพื้นที่การค้นหาขนาดใหญ่ และสามารถหลีกเลี่ยงการติดอยู่ในจุดต่ำสุดเฉพาะที่ได้ดี

การประยุกต์ใช้ GA ในการฝึกสอน MLP จึงเป็นแนวทางที่น่าสนใจ เนื่องจากสามารถมองการฝึกสอนโครงข่ายเป็นปัญหาการหาค่าที่เหมาะสมที่สุดของพารามิเตอร์จำนวนมาก โดย GA จะทำการค้นหาชุดค่าน้ำหนักที่ทำให้โครงข่ายมีความผิดพลาดน้อยที่สุด ผ่านกระบวนการวิวัฒนาการของประชากรที่แต่ละตัวแทนชุดค่าน้ำหนักที่แตกต่างกัน การทดลองนี้จึงมุ่งศึกษาการใช้ GA ในการฝึกสอน MLP เพื่อพัฒนาความเข้าใจและประเมินประสิทธิภาพของวิธีการดังกล่าว

**วัตถุประสงค์**

1. เพื่อพัฒนาโมเดล Multilayer Perceptron (MLP) ที่ได้รับการปรับแต่งพารามิเตอร์ด้วยวิธี Genetic Algorithms (GA)
2. เพื่อเปรียบเทียบประสิทธิภาพการทำนายระหว่างโมเดล MLP ที่ Train ด้วย GA เมื่อมีจำนวน Hidden layer ต่างๆ
3. เพื่อศึกษาความสามารถในการลดค่าความคลาดเคลื่อน (Error) ของโมเดลโดยใช้เทคนิค GA

**ขอบเขตการทดลอง**

1. ใช้ข้อมูล wdbc.data (Wisconsin Diagnostic Breast Cancer (WDBC) จาก UCI Machine learning Repository) โดยที่ data set นี้ มี 2 classes และ 30 features ซึ่งในแต่ละ sample จะมีทั้งหมด 32 ค่าโดยที่

1) ID number

2) Diagnosis (M = malignant, B = benign) เป็น class

3-32) เป็นค่า features ทั้ง 30

1. ข้อมูลถูกแบ่งเป็น 90 % สำหรับ Train และ 10 % สำหรับ Test โดยทำ K-fold validation เป็นจำนวน10 folds
2. ขนาดประชากร (Populations) เป็น 30 โครโมโซม จำนวนรุ่นสูงสุด (Generations) เป็น 100 รุ่น
3. อัตราการกลายพันธุ์ (Mutation rate) เป็น 0.01
4. ใช้การคัดเลือกแบบ Rank-based Selection
5. ใช้ ฟังก์ชันกระตุ้น (Activation Function) แบบ Sigmoid
6. ในการหาวัดค่าความคลาดเคลื่อนใช้ ค่าความผิดพลาดเฉลี่ย (Mean Squared Error: MSE) และความแม่นยำในการจำแนก (Classification Accuracy)

**วิธีดำเนินการ**

1.เขียนโปรแกรมเพื่อฝึกฝน Multilayer Perceptron โดยใช้ Genetic Algorithms สำหรับทดสอบ validity, ความเร็วในการ converge และความถูกต้อง accuracy

2.กำหนดค่า hidden layer, generation, population และ mutation rate ที่เหมาสม

3.นำข้อมูลที่มีไปทำการ k-fold validation เพื่อฝึกฝนโมเดล โดยแบ่งเป็นข้อมูลฝึกฝน และข้อมูลทดสอบ

4.นำข้อมูลที่ทำการ k-fold validation แล้วมาฝึกฝนแล้วทดสอบโมเดล

5.บันทึกผลการทดสอบใรรูปของกราฟ

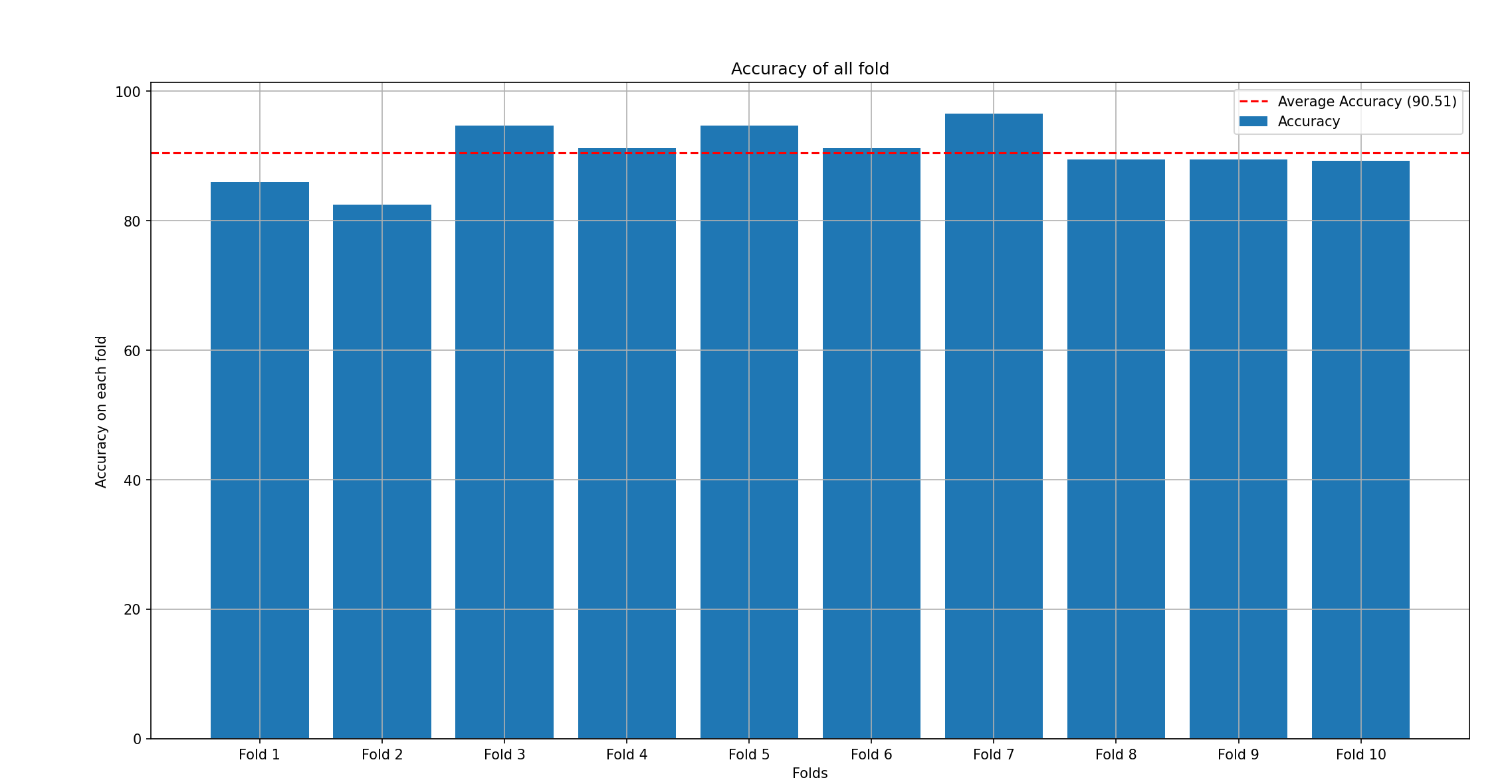
6.ทำตามข้อ 2. ถึง 5. ซ้ำเรื่อยๆ โดยเปลี่ยนแปลงค่า hidden layer และ node จนได้ผลการทดลองที่เพียงพอ

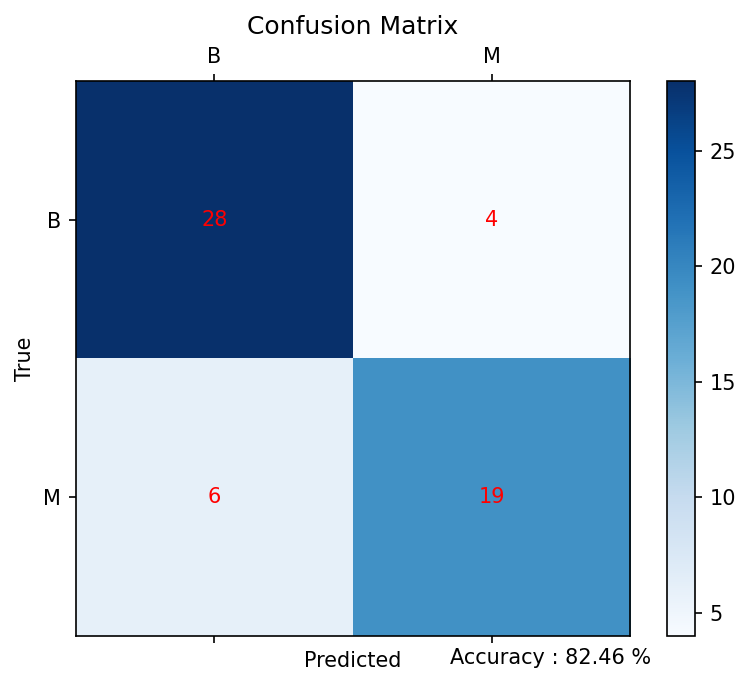
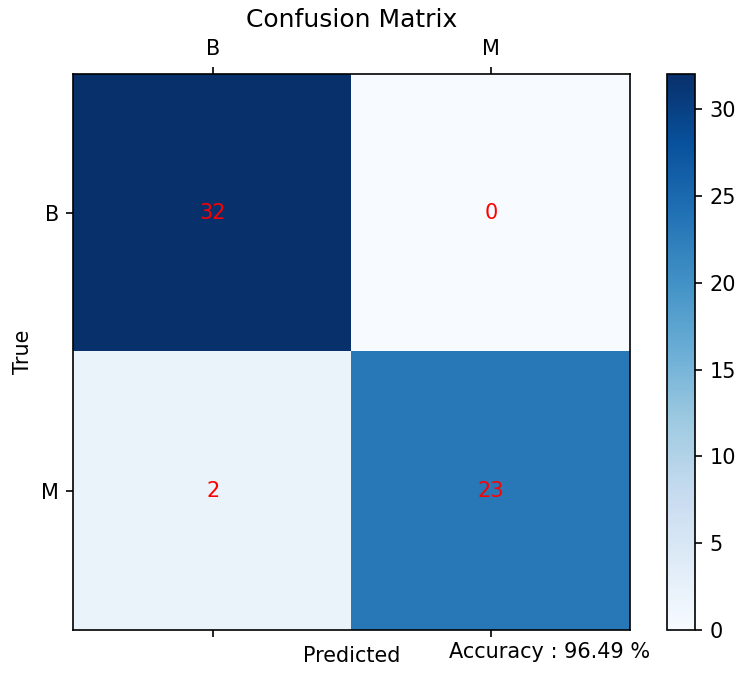
**ผลการทดลอง**

1.ผลการทดลอง classification ที่แปรตาม hidden layer คือ 1 layer 60 node โดยกำหนดให้ ประชากร (population) เป็น 30 โครโมโซม จำนวนรุน (generation) เป็น 100 รุ่น อัตราการกลายพันธุ์ (mutation rate) เป็น 0.01

A graph of a graph

Description automatically generated with medium confidence

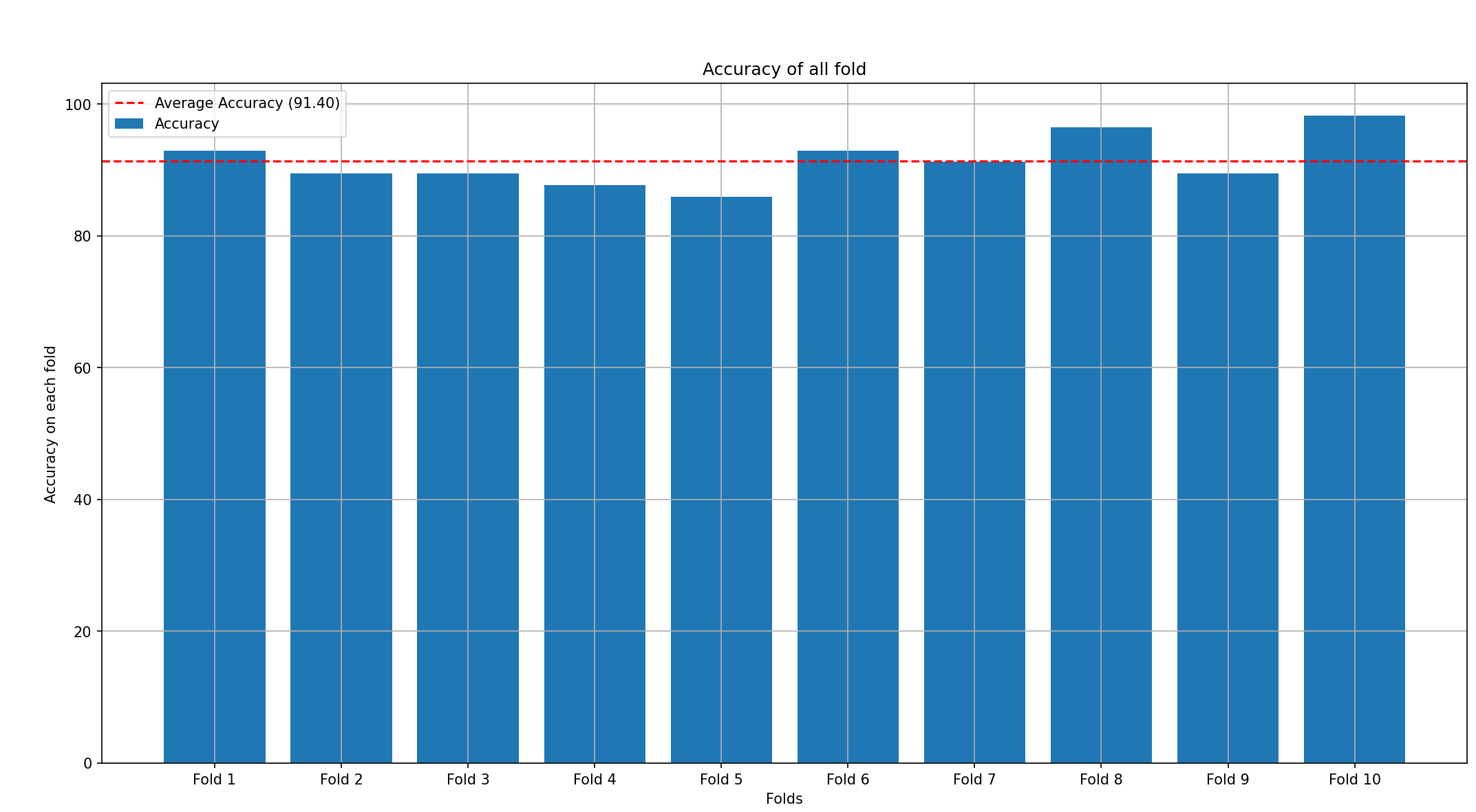


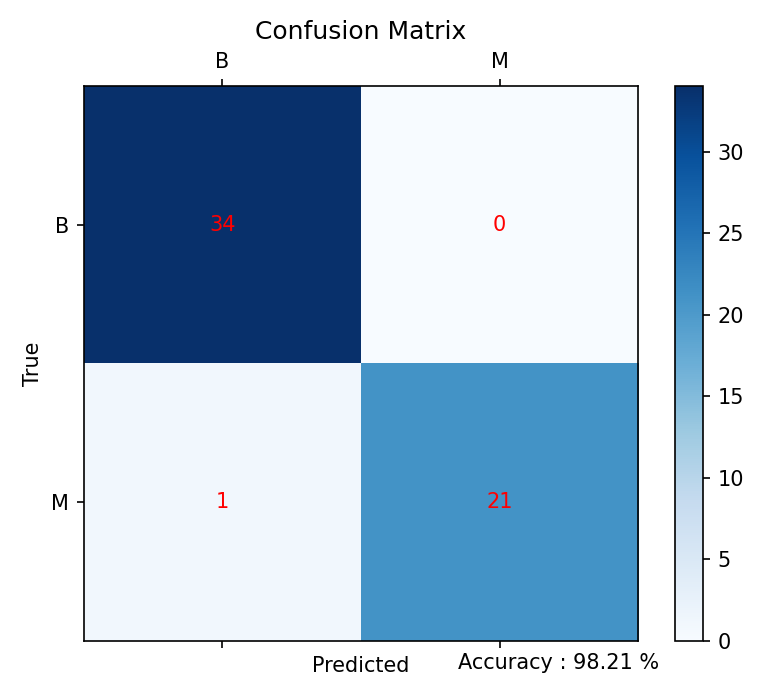
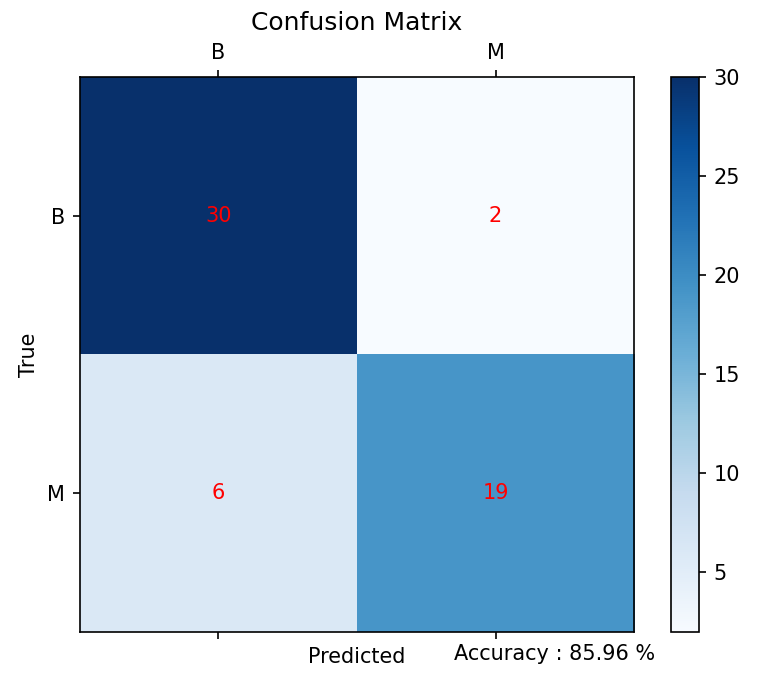


จากการทดลอง เมื่อใช้จำนวน hidden layer เป็น 1 layer 60 node จะพบว่าเมื่อทดสอบทั้ง 10 fold แล้วได้ค่า Average Accuracy เป็น 90.51 % โดยมีค่า Accuracy ต่ำสุดที่ 82.46 % และค่า Accuracy สูงสุดที่ 96.49 %

A graph of a graph

Description automatically generated with medium confidence2.ผลการทดลอง classification ที่แปรตาม hidden layer คือ 1 layer 30 node โดยกำหนดให้ ประชากร (population) เป็น 30 โครโมโซม จำนวนรุน (generation) เป็น 100 รุ่น อัตราการกลายพันธุ์ (mutation rate) เป็น 0.01

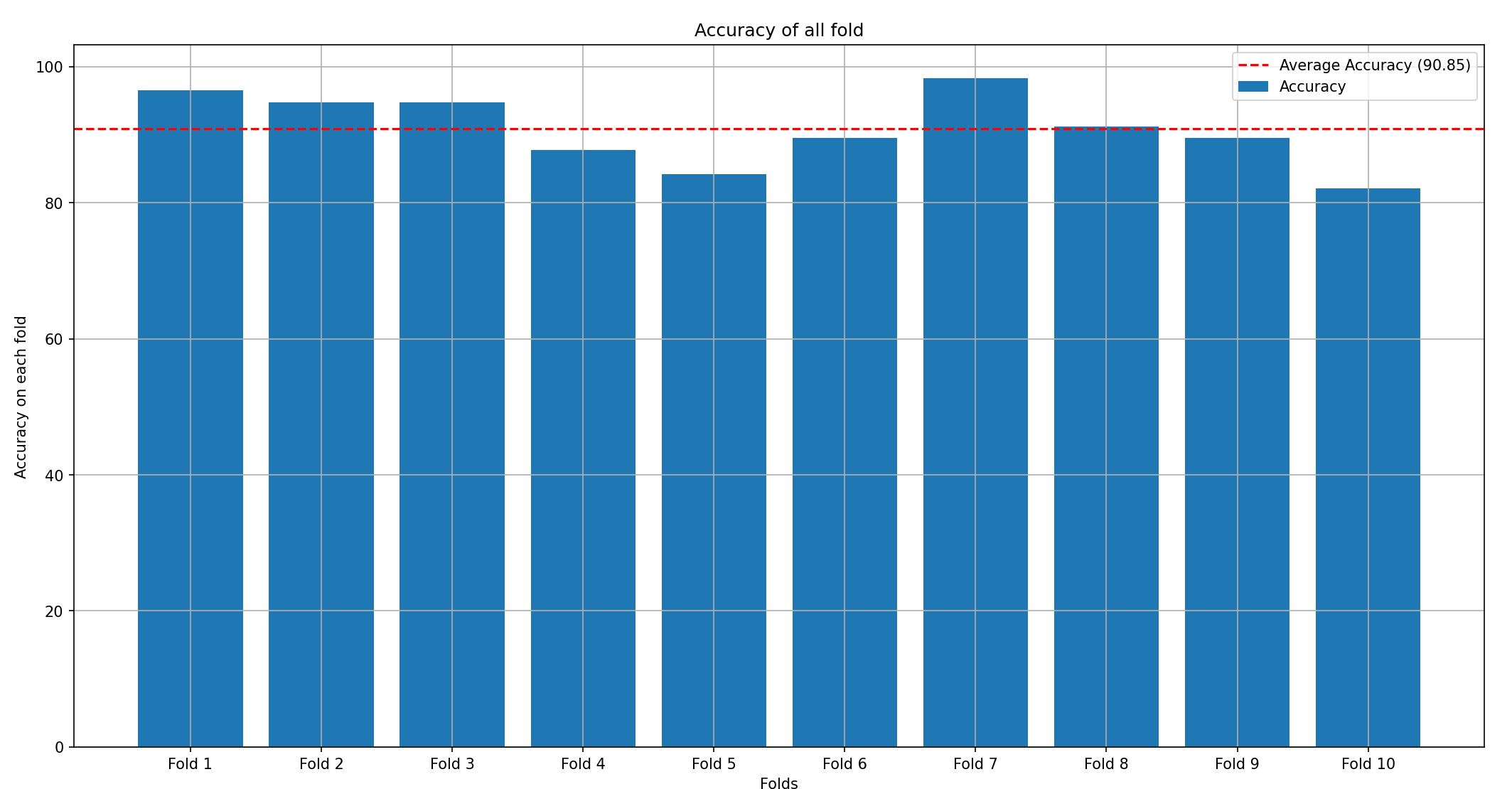


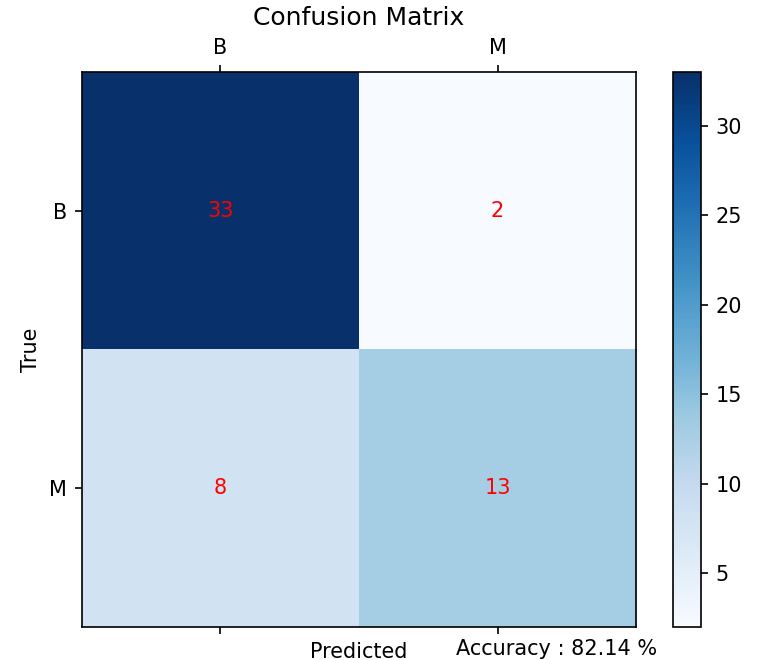
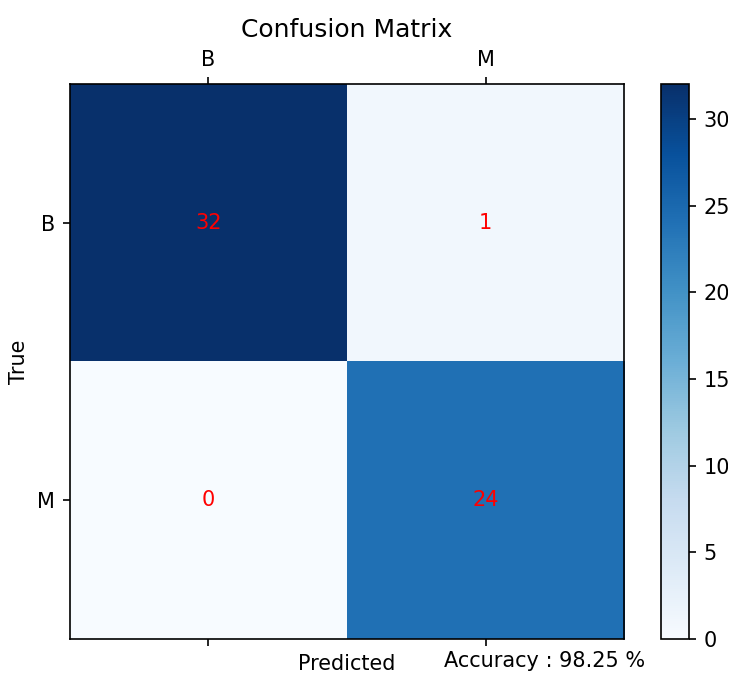


จากการทดลอง เมื่อใช้จำนวน hidden layer เป็น 1 layer 30 node จะพบว่าเมื่อทดสอบทั้ง 10 fold แล้วได้ค่า Average Accuracy เป็น 91.40 % โดยมีค่า Accuracy ต่ำสุดที่ 85.96 % และค่า Accuracy สูงสุดที่ 98.21 %

A graph of a graph

Description automatically generated with medium confidence3.ผลการทดลอง classification ที่แปรตาม hidden layer คือ 1 layer 10 node โดยกำหนดให้ ประชากร (population) เป็น 30 โครโมโซม จำนวนรุน (generation) เป็น 100 รุ่น อัตราการกลายพันธุ์ (mutation rate) เป็น 0.01

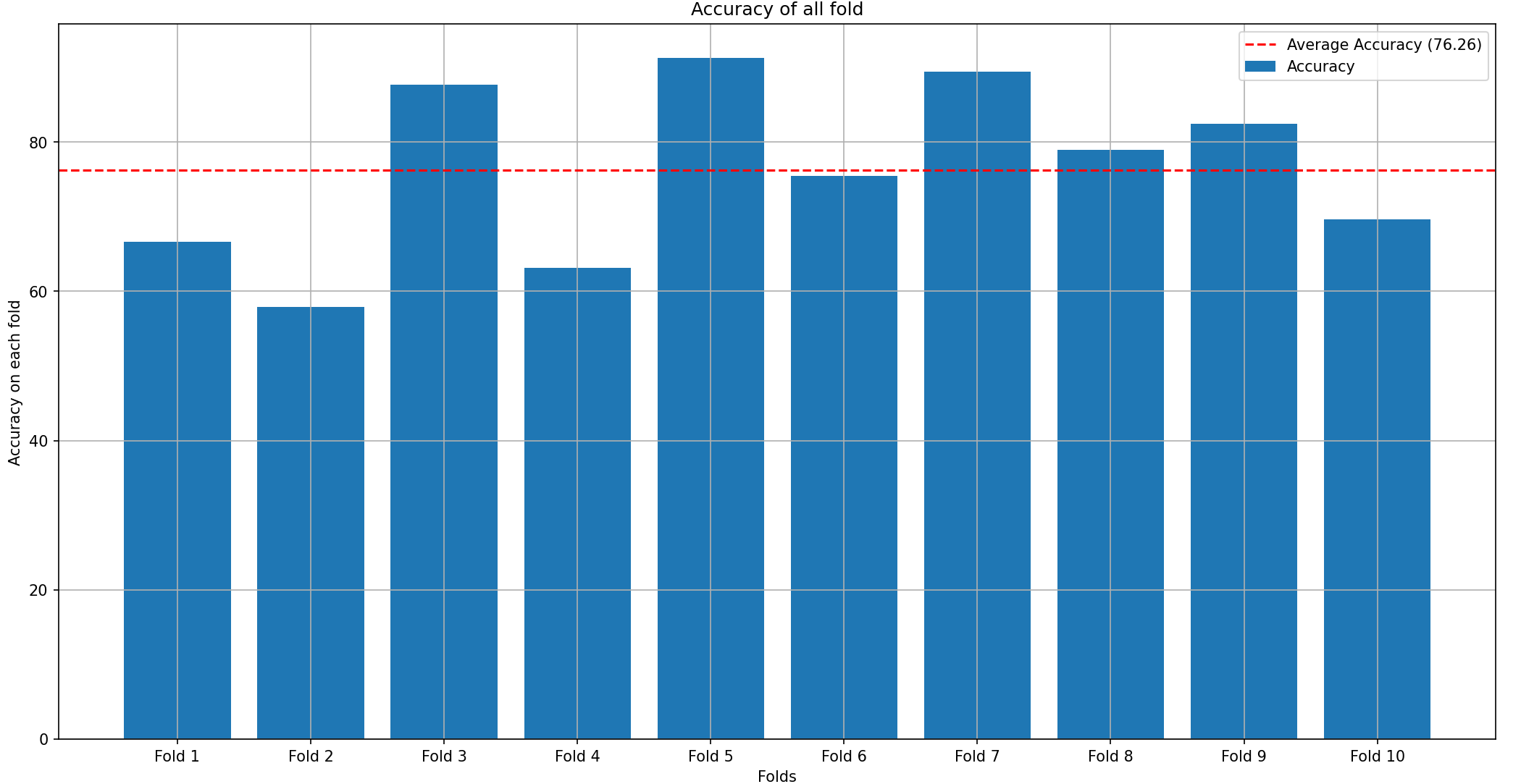


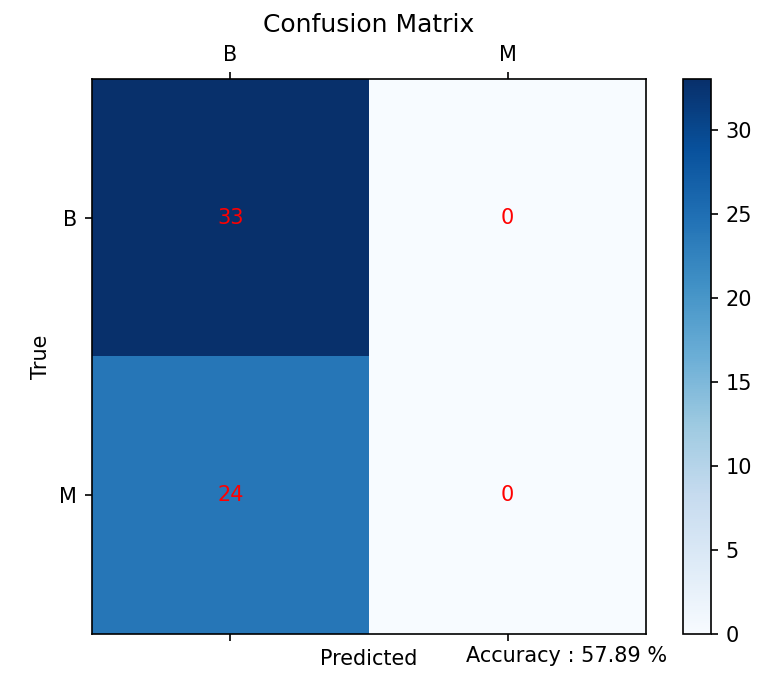
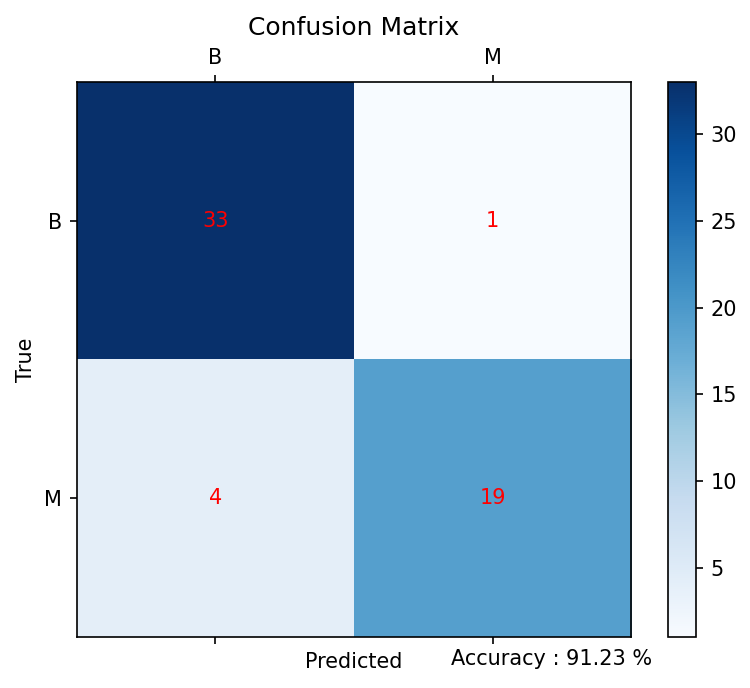


จากการทดลอง เมื่อใช้จำนวน hidden layer เป็น 1 layer 10 node จะพบว่าเมื่อทดสอบทั้ง 10 fold แล้วได้ค่า Average Accuracy เป็น 90.85 % โดยมีค่า Accuracy ต่ำสุดที่ 82.14 % และค่า Accuracy สูงสุดที่ 98.25 %

A diagram of a graph

Description automatically generated with medium confidence4.ผลการทดลอง classification ที่แปรตาม ประชากร (population) คือ 5 โครโมโซม โดยกำหนดให้ hidden layer เป็น 1 layer 30 node จำนวนรุน (generation) เป็น 100 รุ่น อัตราการกลายพันธุ์ (mutation rate) เป็น 0.01

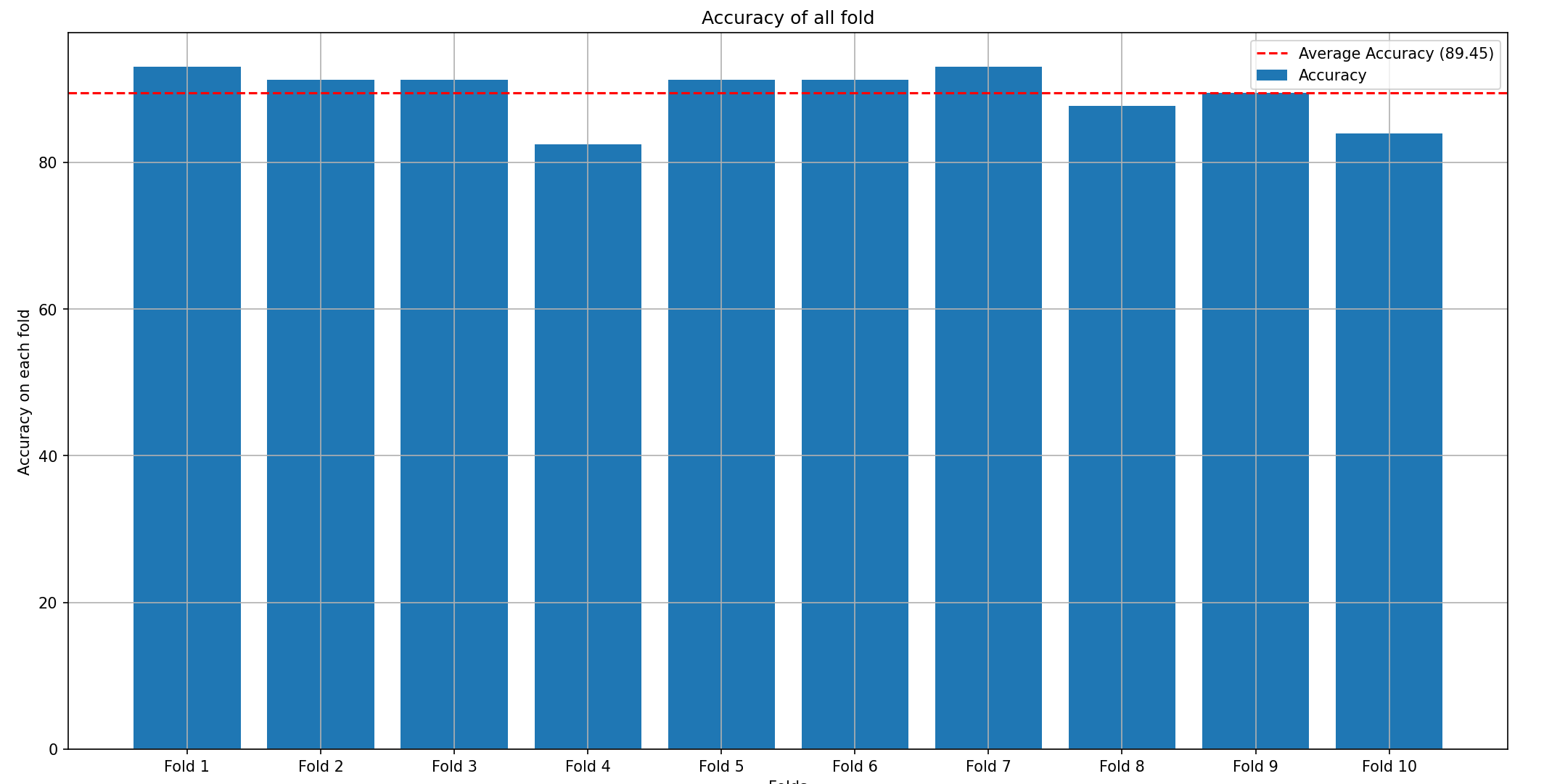


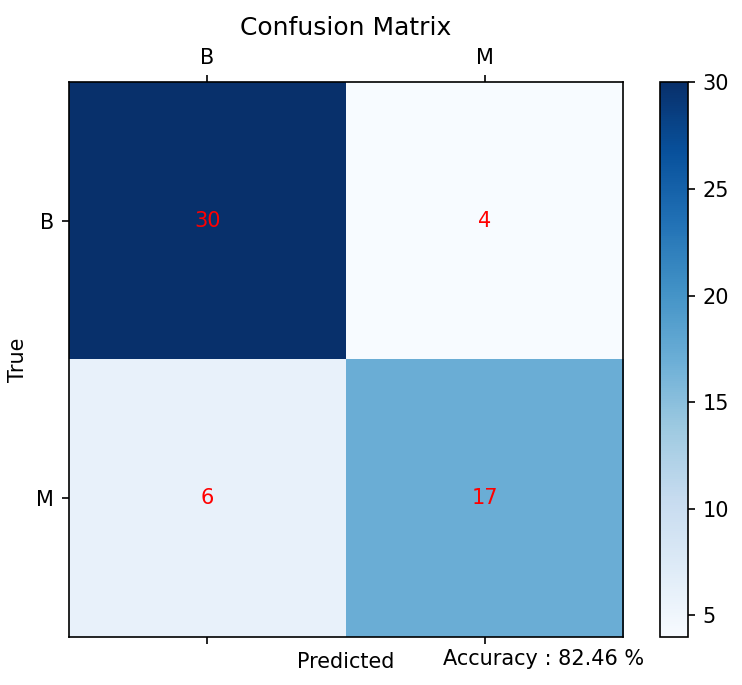
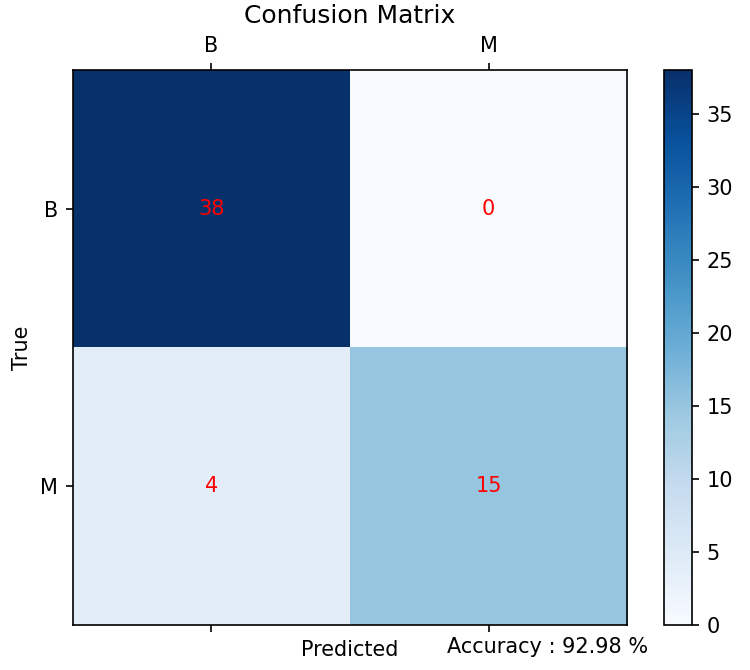


จากการทดลอง เมื่อใช้จำนวน population เป็น 5 โครโมโซม พบว่าเมื่อทดสอบทั้ง 10 fold แล้วได้ค่า Average Accuracy เป็น 76.26 % โดยมีค่า Accuracy ต่ำสุดที่ 57.89 % และค่า Accuracy สูงสุดที่ 91.23 %

A graph of a graph

Description automatically generated with medium confidence5.ผลการทดลอง classification ที่แปรตาม ประชากร (population) คือ 30 โครโมโซม โดยกำหนดให้ hidden layer เป็น 1 layer 30 node จำนวนรุน (generation) เป็น 100 รุ่น อัตราการกลายพันธุ์ (mutation rate) เป็น 0.01

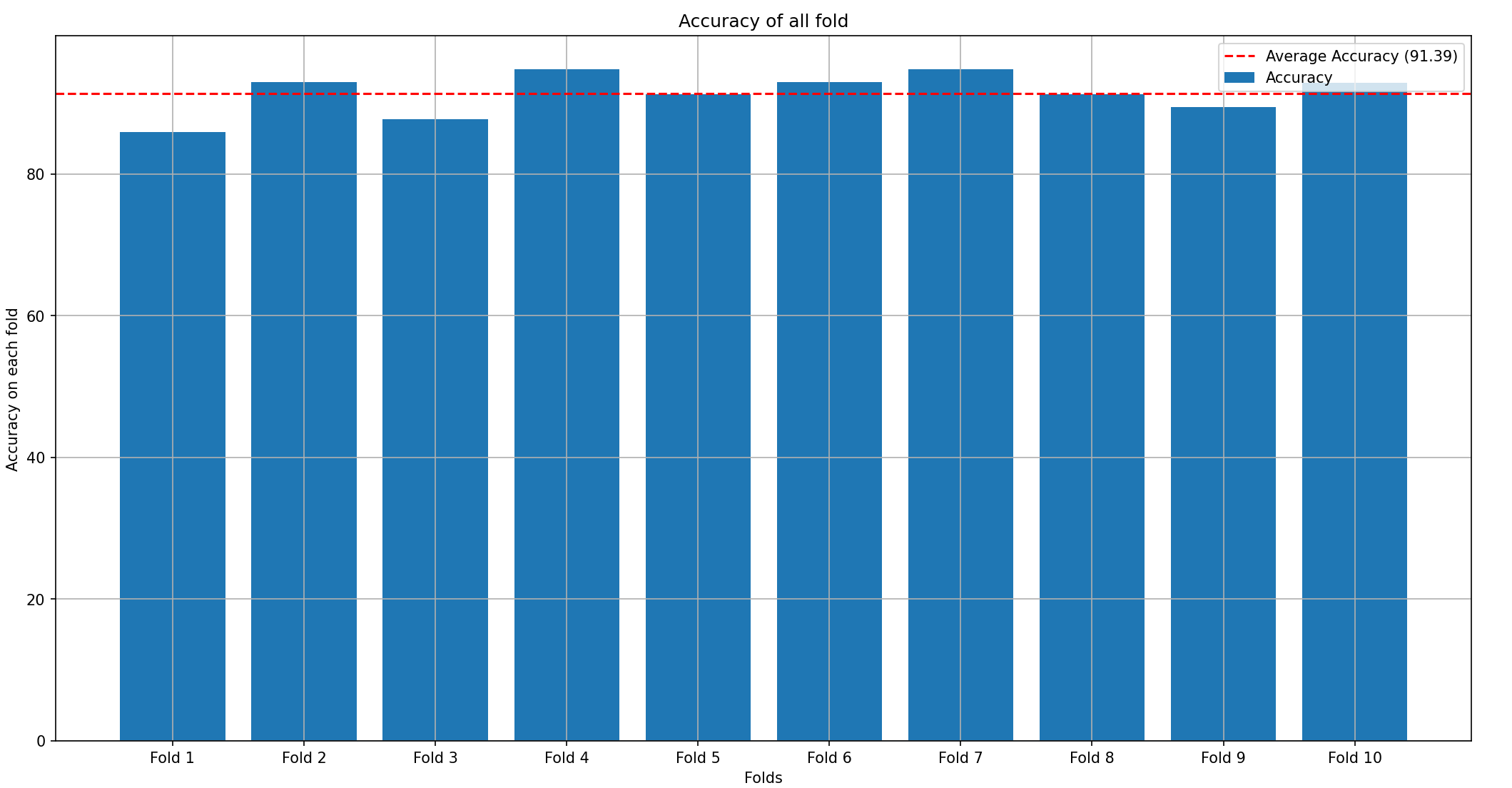


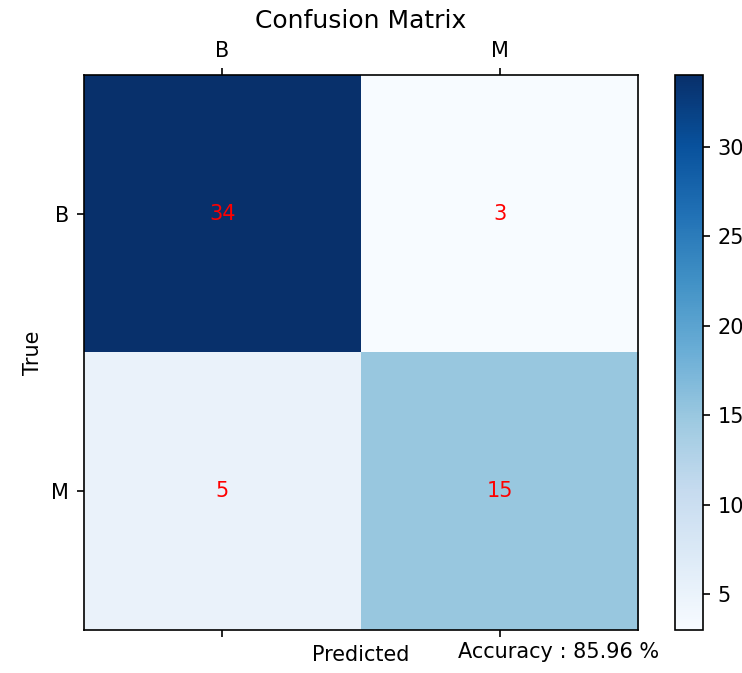
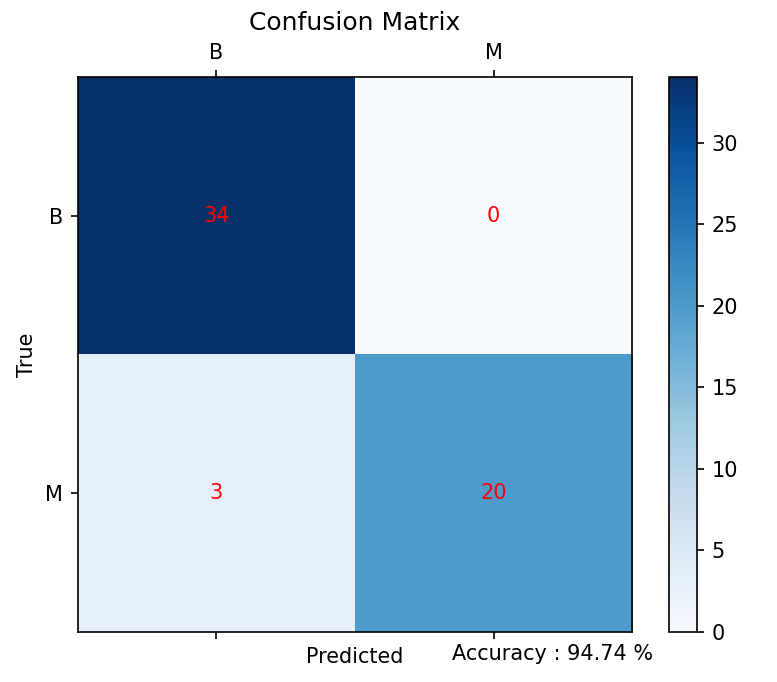


จากการทดลอง เมื่อใช้จำนวน population เป็น 30 โครโมโซม พบว่าเมื่อทดสอบทั้ง 10 fold แล้วได้ค่า Average Accuracy เป็น 89.45 % โดยมีค่า Accuracy ต่ำสุดที่ 82.46 % และค่า Accuracy สูงสุดที่ 92.98 %

A graph of a graph

Description automatically generated with medium confidence6.ผลการทดลอง classification ที่แปรตาม ประชากร (population) คือ 60 โครโมโซม โดยกำหนดให้ hidden layer เป็น 1 layer 30 node จำนวนรุน (generation) เป็น 100 รุ่น อัตราการกลายพันธุ์ (mutation rate) เป็น 0.01





จากการทดลอง เมื่อใช้จำนวน population เป็น 60 โครโมโซม พบว่าเมื่อทดสอบทั้ง 10 fold แล้วได้ค่า Average Accuracy เป็น 91.39 % โดยมีค่า Accuracy ต่ำสุดที่ 85.96 % และค่า Accuracy สูงสุดที่ 94.74 %

**วิเคราะห์ผลการทดลอง**

จากผลการทดลอง เพื่อศึกษา validity, ความเร็วในการ converge และความถูกต้อง Accuracy ของ MLP ที่ใช้ Genetic Algorithm (GA) ทำให้ทราบว่า จำนวน hidden layer มีผลต่อ validity, ความเร็วในการ converge และความถูกต้อง Accuracy ของ MLP น้อยมาก เมื่อเทียบกับจำนวนประชากร (Populations) ของ GA คือยิ่ง GA มี population มากจะยิ่ง train MLP ได้แม่นยำมากขึ้น ในทางกลับกันจะใช้เวลาในการ train นานขึ้นด้วย และถ้ามี population น้อยจะทำให้ train MLP ได้ไม่แม่นยำเท่าที่ควร ในทางกลับกันจะทำให้ train ได้รวดเร็วขึ้น ดังนั้นหากต้องการให้การ train MLP ด้วย GA ให้มีประสิทธิภาพมากที่สุด ควรเลือกจำนวนประชากรให้เหมาะสมกับชุดข้อมูล และความแม่นยำที่ต้องการ

**ภาคผนวก**

Github : Code of Assignment 3

import numpy as np

import random

import matplotlib.pyplot as plt

# MLP class with adjustable layers and biases

class MLP:

    def \_\_init\_\_(self, layer\_sizes):

        self.layers = layer\_sizes

        self.weights = [np.random.uniform(-1, 1, (self.layers[i], self.layers[i + 1])) for i in range(len(self.layers) - 1)]

        self.biases = [np.random.uniform(-1, 1, (1, self.layers[i + 1])) for i in range(len(self.layers) - 1)]

    # Activation function and derivative (Sigmoid)

    def sigmoid(self, x):

        return 1 / (1 + np.exp(-x))

    def sigmoid\_derivative(self, x):

        return x \* (1 - x)

    def forward(self, x):

        self.activations = [x]

        for w, b in zip(self.weights, self.biases):

            x = self.sigmoid(np.dot(x, w) + b)

            self.activations.append(x)

        return x

    def compute\_loss(self, y\_true, y\_pred):

        return np.mean((y\_true - y\_pred) \*\* 2)

# Genetic Algorithm class

class GeneticAlgorithm:

    def \_\_init\_\_(self, x\_input, d\_output, layer\_sizes, pop\_size, generations, mutation\_rate):

        self.x\_input = x\_input

        self.d\_output = d\_output

        self.layer\_sizes = layer\_sizes

        self.pop\_size = pop\_size

        self.generations = generations

        self.mutation\_rate = mutation\_rate

    def initialize\_population(self):

        MLP\_pop = []

        for \_ in range(self.pop\_size):

            MLP\_pop.append(MLP(self.layer\_sizes))

        return MLP\_pop

    def mutate(self, weights, biases):

        for i in range(len(weights)):

            if random.random() < self.mutation\_rate:

                weights[i] += np.random.normal(0, 0.5, weights[i].shape)

            if random.random() < self.mutation\_rate:

                biases[i] += np.random.normal(0, 0.5, biases[i].shape)

        return weights, biases

    def crossover(self, parent1, parent2):

        child\_weights = []

        child\_biases = []

        for w1, w2, b1, b2 in zip(parent1.weights, parent2.weights, parent1.biases, parent2.biases):

            mask = np.random.rand(\*w1.shape) < 0.5

            child\_weights.append(np.where(mask, w1, w2))

            mask\_bias = np.random.rand(\*b1.shape) < 0.5

            child\_biases.append(np.where(mask\_bias, b1, b2))

        return child\_weights, child\_biases

    def train(self):

        population = self.initialize\_population()

        history = []

        for gen in range(self.generations):

            # Evaluate fitness (accuracy in this case)

            fitness\_scores = []

            for mlp in population:

                predictions = np.round([mlp.forward(x) for x in self.x\_input])

                fitness\_scores.append(np.mean(predictions.flatten() == self.d\_output))

            # Select best individuals

            sorted\_indices = np.argsort(fitness\_scores)[::-1]

            population = [population[i] for i in sorted\_indices[:self.pop\_size // 2]]

            # Crossover and mutation

            new\_population = []

            for \_ in range(self.pop\_size // 2):

                parent1, parent2 = random.sample(population, 2)

                child\_weights, child\_biases = self.crossover(parent1, parent2)

                child\_weights, child\_biases = self.mutate(child\_weights, child\_biases)

                child = MLP(self.layer\_sizes)

                child.weights = child\_weights

                child.biases = child\_biases

                new\_population.append(child)

            population.extend(new\_population)

            history.append(max(fitness\_scores))

            print(f"Generation {gen + 1}, Best Fitness: {max(fitness\_scores)}")

        return population[0], history

# Load data function (assumes wdbc.data is in the same directory)

def load\_data(file\_name):

    data = np.genfromtxt(file\_name, delimiter=',', dtype=str)

    ids, diagnosis, x\_input = data[:, 0], data[:, 1], data[:, 2:].astype(np.float32)

    d\_output = []

    for d in diagnosis:     # M=1, B=0

        if(d == 'M'):

            d\_output.append(1)

        elif(d == 'B'):

            d\_output.append(0)

    d\_output = np.array(d\_output)

    return x\_input, d\_output

def k\_fold\_validation(x\_input, d\_output, k=10):

    fold\_size = len(x\_input) // k

    indices = np.arange(len(x\_input))

    np.random.shuffle(indices)

    folds\_x = np.array\_split(x\_input[indices], k)

    folds\_y = np.array\_split(d\_output[indices], k)

    for i in range(k):

        # Create training and testing sets for the i-th fold

        x\_test = folds\_x[i]

        y\_test = folds\_y[i]

        x\_train = np.concatenate([folds\_x[j] for j in range(k) if j != i])

        y\_train = np.concatenate([folds\_y[j] for j in range(k) if j != i])

        yield x\_train, y\_train, x\_test, y\_test  # Yielding for each fold

def plot\_confusion\_matrix(desired\_output, prediction):

    # Ensure the arrays are integers and binary (0, 1)

    y\_true = np.array(desired\_output, dtype=int).flatten()

    y\_pred = np.array(prediction, dtype=int).flatten()

    # Initialize the confusion matrix for binary classification

    confusion\_matrix = np.zeros((2, 2), dtype=int)

    for true, pred in zip(y\_true, y\_pred):

        # Only update the matrix if values are within expected binary range

        if true in [0, 1] and pred in [0, 1]:

            confusion\_matrix[true, pred] += 1

    TP = confusion\_matrix[1, 1]  # True Positive

    FP = confusion\_matrix[0, 1]  # False Positive

    FN = confusion\_matrix[1, 0]  # False Negative

    TN = confusion\_matrix[0, 0]  # True Negative

    # Calculate accuracy, handling the zero denominator

    total = TP + TN + FP + FN

    accuracy\_rate = (100 \* (TP + TN) / total)

    acc\_text = "Accuracy : {:.2f} %".format(accuracy\_rate)

    # Plot the confusion matrix

    fig, ax = plt.subplots()

    cax = ax.matshow(confusion\_matrix, cmap=plt.cm.Blues)

    plt.colorbar(cax)

    # Annotate the confusion matrix with counts

    for (i, j), val in np.ndenumerate(confusion\_matrix):

        plt.text(j, i, val, ha='center', va='center', color='red')

    plt.xlabel('Predicted')

    plt.ylabel('True')

    plt.title('Confusion Matrix')

    plt.text(0.85,1.60,s = acc\_text)

    plt.xticks([0, 1], ['B', 'M'])

    plt.yticks([0, 1], ['B', 'M'])

    return accuracy\_rate

def average\_plot(acc,k):

    plt.figure()

    print("Accuracy on each folds : {} %".format(acc))

    average\_accuracy = np.mean(acc)

    print(f"Average Accuracy over {k} folds: {average\_accuracy :.2f}%")

    x = []

    y = acc

    for i in range(k):

        x.append(f'Fold {i+1}')

    plt.bar(x,y,label = 'Accuracy')

    plt.axhline(y=average\_accuracy, color='red', linestyle='--', label=f"Average Accuracy ({average\_accuracy:.2f})")

    plt.xlabel("Folds")

    plt.ylabel("Accuracy on each fold")

    plt.title("Accuracy of all fold")

    plt.legend()

    plt.grid()

if \_\_name\_\_ == '\_\_main\_\_':

    # Main execution

    file = 'wdbc.txt'

    X, Y = load\_data(file)

    #initialize parameters

    k = 10  # Number of folds

    layer\_sizes = [30, 30, 1]  # [input\_size, hidden\_layers, output\_size]

    population = 60

    generation = 100

    mutation\_rate = 0.01

    accuracies = []

    plot = 1

    for train\_X, train\_Y, test\_X, test\_Y in k\_fold\_validation(X, Y, k):

        print('\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_')

        print('Fold : {}'.format(plot))

        print(' ')

        # Genetic Algorithm to find the best weights for each fold

        GA = GeneticAlgorithm(train\_X, train\_Y, layer\_sizes, population, generation, mutation\_rate)

        best\_mlp, history = GA.train()

        # Evaluate the final model for the current fold

        test\_predictions = []

        predict = []

        for x in test\_X:

            test\_predictions.append(np.round(best\_mlp.forward(x)))

        test\_predictions = np.array(test\_predictions)

        accuracy = plot\_confusion\_matrix(test\_Y,test\_predictions.flatten())

        accuracies.append(accuracy)

        plt.figure(0)

        plt.subplot(1,10,plot)

        plt.grid()

        if(plot == 1):

            plt.ylabel('Best Fitness')

        plt.xlabel('Fold {}'.format(plot))

        plt.plot(history)

        plot+=1

    plt.suptitle('Genetic Algorithm Progress')

    average\_plot(accuracies,k)

    plt.show()