Project 1: part C and D

first name and T.Z. numbers

second name and T.Z. numbers

Part C:

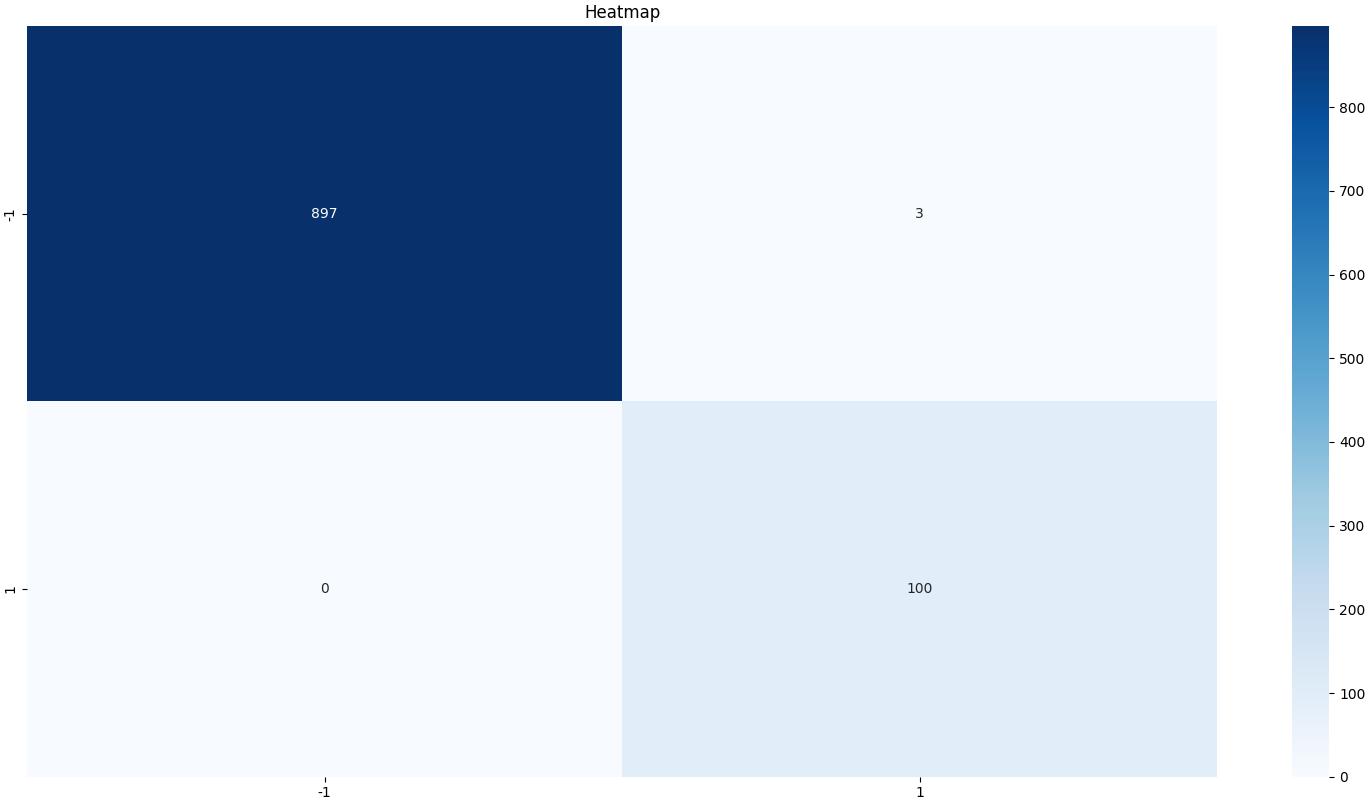
* Dataset:

|  |  |
| --- | --- |
| Class | Number samples |
| Test | |
| -1 | 900 |
| 1 | 100 |
| Train | |
| -1 | 900 |
| 1 | 100 |

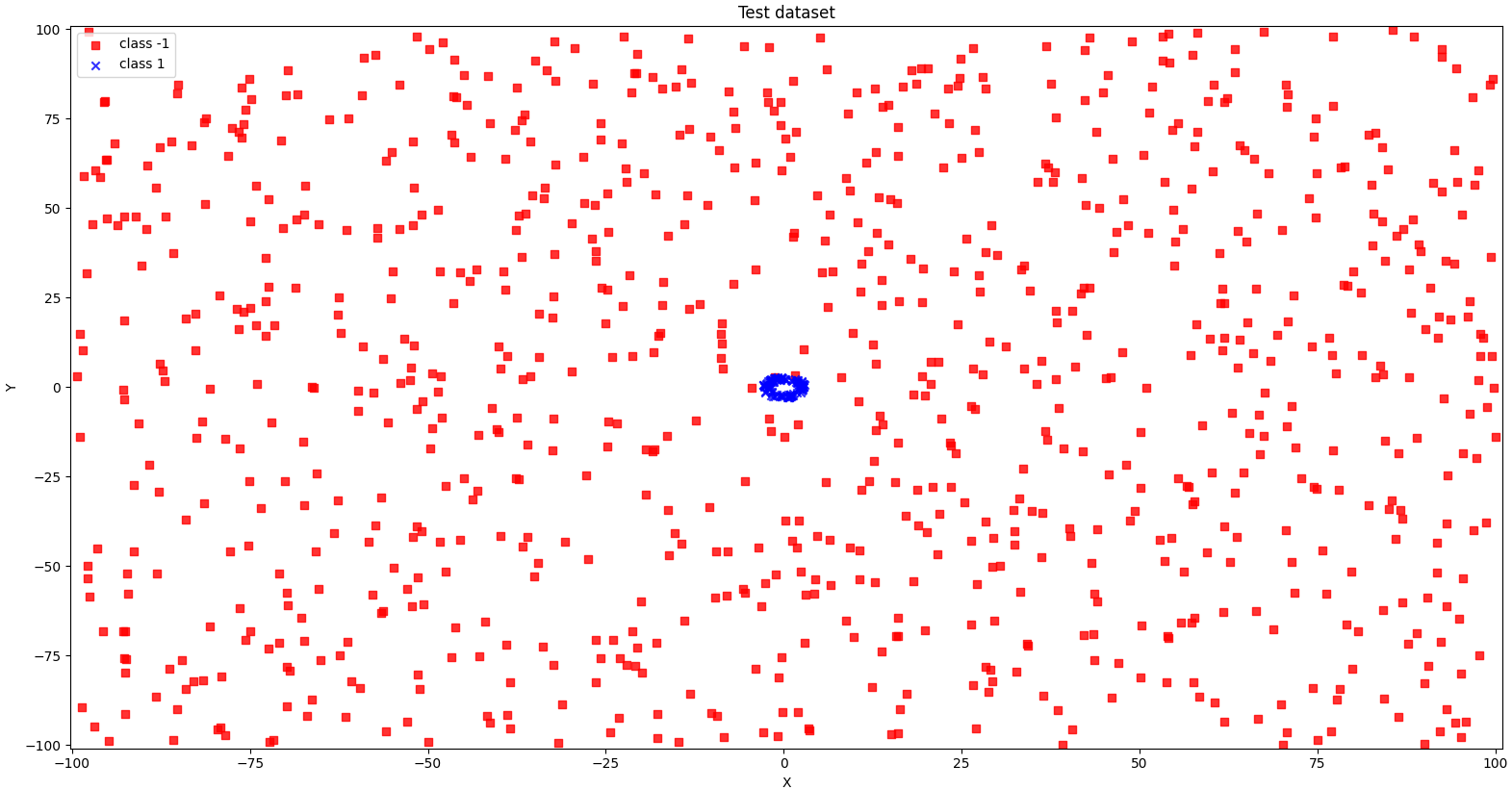
* Classification report:

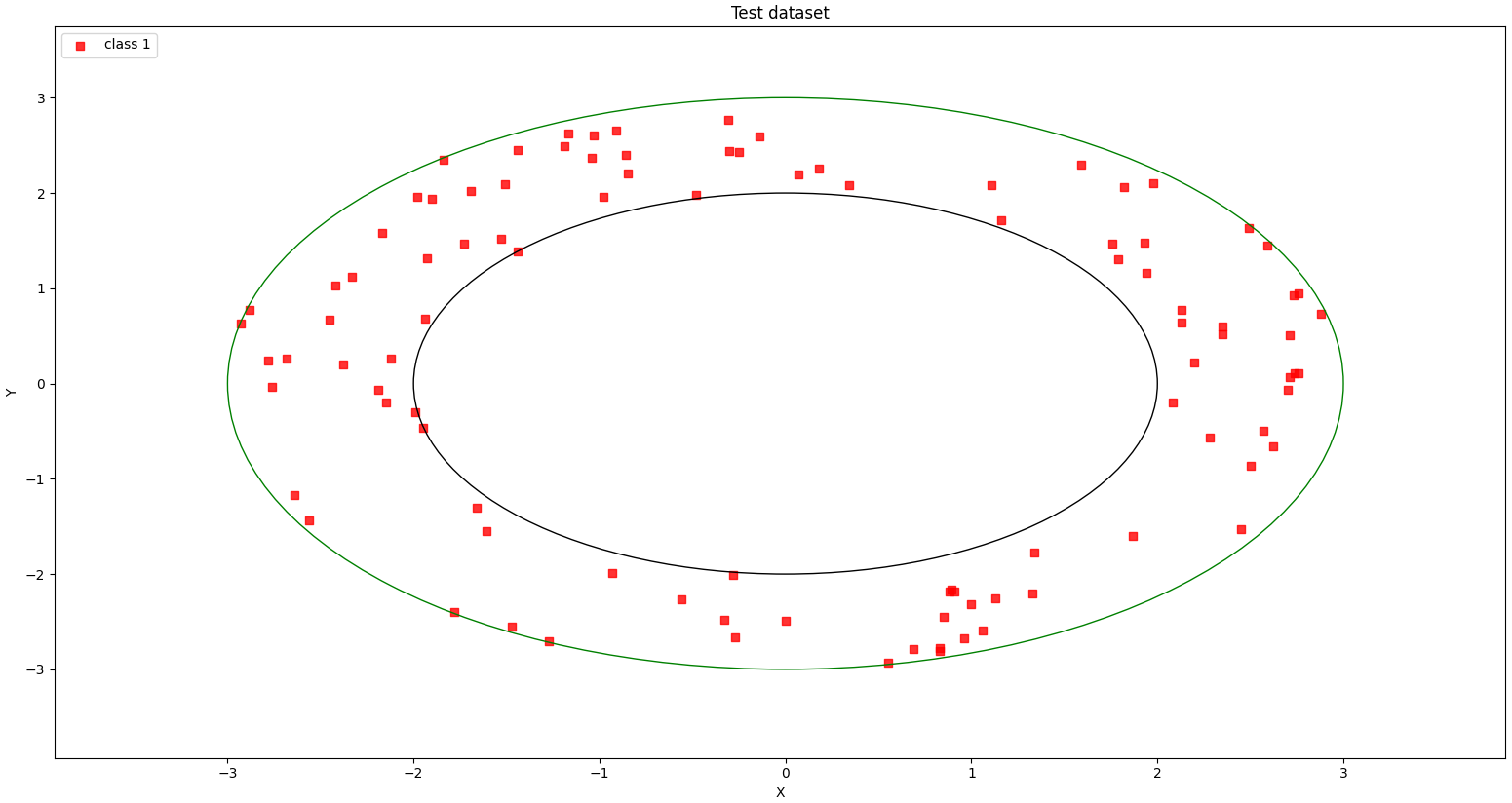
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Precision | Recall | F1-score | Support |
| -1 | 1.00 | 1.00 | 1.00 | 900 |
| 1 | 0.97 | 1.00 | 0.99 | 100 |
|  |  |  |  |  |
| accuracy |  |  | 1.00 | 1000 |
| macro avg | 0.99 | 1.00 | 0.99 | 1000 |
| weighted avg | 1.00 | 1.00 | 1.00 | 1000 |

* Heatmap:

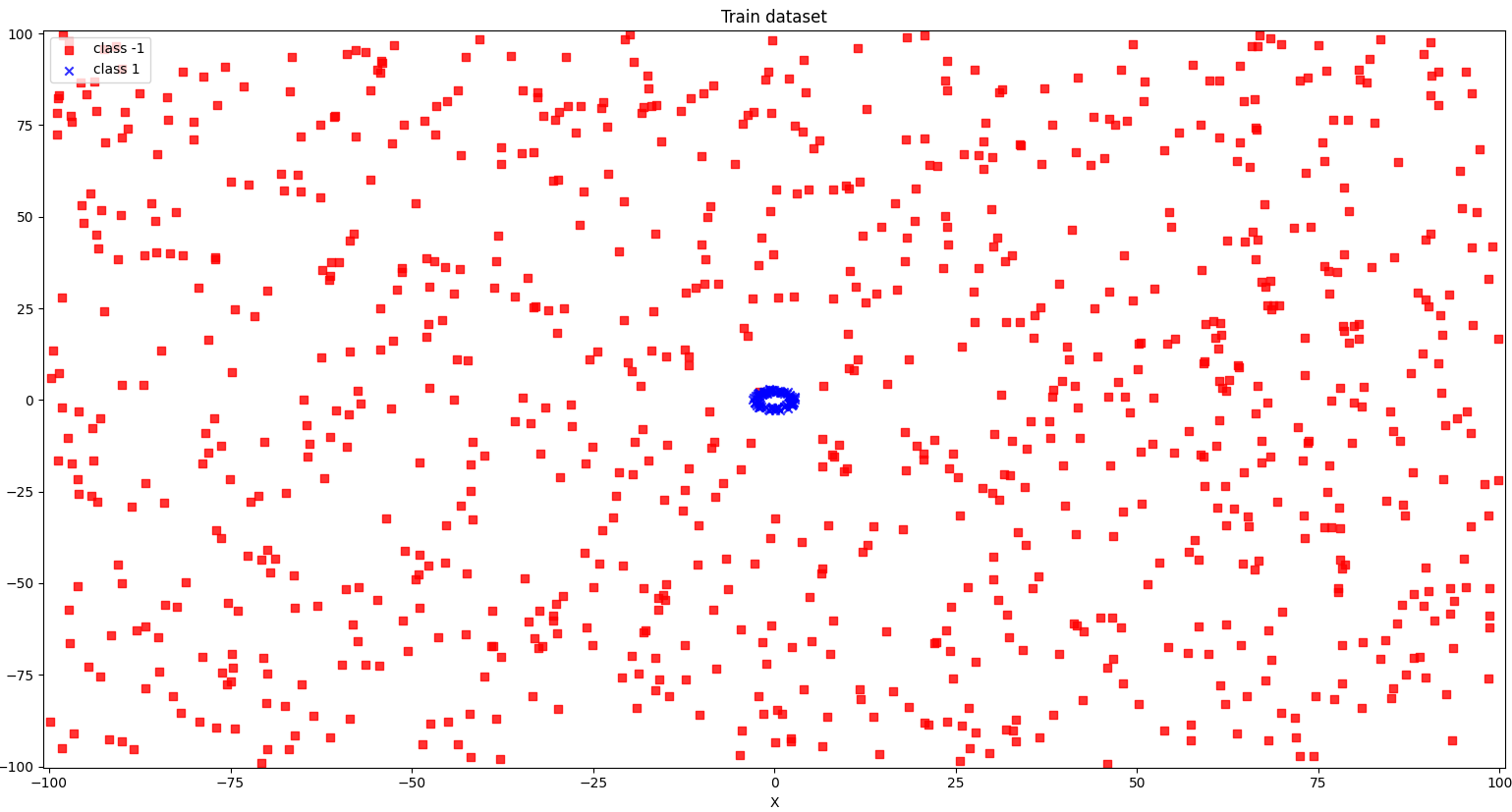


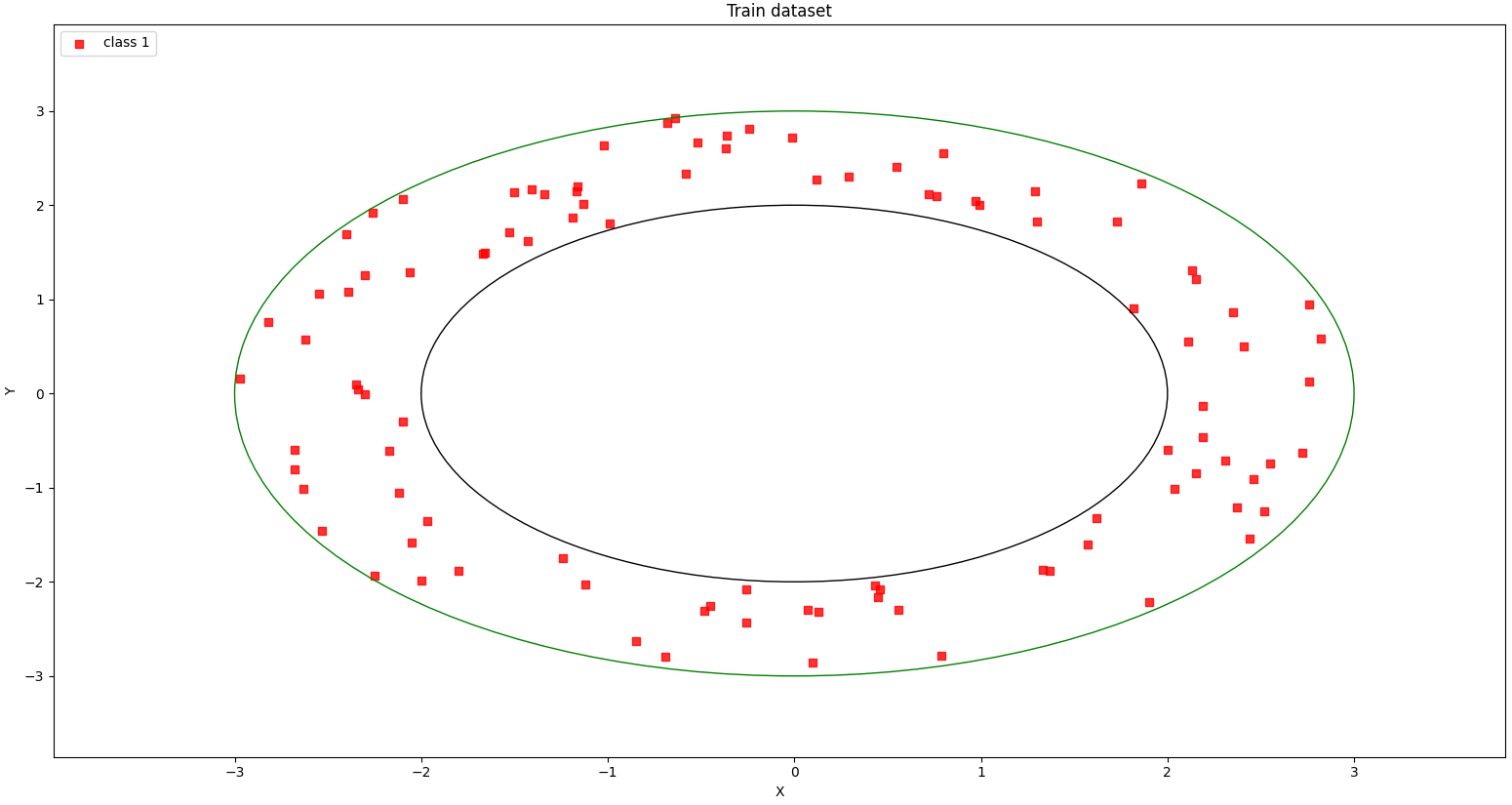
* Accuracy score: 99.7%
* Test illustration:





* Train illustration:





* Code:

import random

import pandas as pd

import numpy as np

import seaborn as sns

import matplotlib.pyplot as plt

from math import exp

from matplotlib.colors import ListedColormap

from sklearn.preprocessing import StandardScaler

from sklearn.metrics import classification\_report, confusion\_matrix, accuracy\_score

max\_limit = 10000

min\_limit = -10000

num\_samples = 1000

def generateDataset():

    one\_samples = 0

    zero\_samples = 0

    data = []

    while  (one\_samples + zero\_samples ) < num\_samples:

        n = random.randint(min\_limit, max\_limit)

        m = random.randint(min\_limit, max\_limit)

        x = m/100

        y = n/100

        circle = pow(x, 2) + pow(y, 2)

        if (circle <= 9 and circle >= 4):

            one\_samples += 1

            data.append([x, y, 1])

        elif zero\_samples < 900:

            zero\_samples += 1

            data.append([x, y, -1])

    return data

def datasetIllustration(X, y, show\_circle=False, resolution=0.02):

    # setup marker generator and color map

    markers = ('s', 'x', 'o', '^', 'v')

    colors = ('red', 'blue', 'lightgreen', 'gray', 'cyan')

    cmap = ListedColormap(colors[:len(np.unique(y))])

    # plot the decision surface

    x1\_min, x1\_max = X[:,  0].min() - 1, X[:, 0].max() + 1

    x2\_min, x2\_max = X[:, 1].min() - 1, X[:, 1].max() + 1

    xx1, xx2 = np.meshgrid(np.arange(x1\_min, x1\_max, resolution),

    np.arange(x2\_min, x2\_max, resolution))

    plt.xlim(xx1.min(), xx1.max())

    plt.ylim(xx2.min(), xx2.max())

    # plot class samples

    for idx, cl in enumerate(np.unique(y)):

        plt.scatter(x=X[y == cl, 0], y=X[y == cl, 1],

        alpha=0.8, c=cmap(idx),

        marker=markers[idx], label='class ' + str(cl))

    # circles

    if show\_circle:

        circle9 = plt.Circle((0, 0), 2, color='black', fill=False)

        circle4 = plt.Circle((0, 0), 3, color='green', fill=False)

        plt.gca().add\_patch(circle4)

        plt.gca().add\_patch(circle9)

# Calculate neuron activation for an input

def activate(weights, inputs):

    activation = weights[-1]

    for i in range(len(weights)-1):

        activation += weights[i] \* inputs[i]

    return activation

# Transfer neuron activation

def transfer(activation):

    return 1.0 / (1.0 + exp(-activation))

# Forward propagate input to a network output

def forward\_propagate(network, row):

    inputs = row

    for layer in network:

        new\_inputs = []

        for neuron in layer:

            activation = activate(neuron['weights'], inputs)

            neuron['output'] = transfer(activation)

            new\_inputs.append(neuron['output'])

        inputs = new\_inputs

    return inputs

# Calculate the derivative of an neuron output

def transfer\_derivative(output):

    return output \* (1.0 - output)

# Backpropagate error and store in neurons

def backward\_propagate\_error(network, expected):

    for i in reversed(range(len(network))):

        layer = network[i]

        errors = list()

        if i != len(network)-1:

            for j in range(len(layer)):

                error = 0.0

                for neuron in network[i + 1]:

                    error += (neuron['weights'][j] \* neuron['delta'])

                errors.append(error)

        else:

            for j in range(len(layer)):

                neuron = layer[j]

                errors.append(neuron['output'] - expected[j])

        for j in range(len(layer)):

            neuron = layer[j]

            neuron['delta'] = errors[j] \* transfer\_derivative(neuron['output'])

# Update network weights with error

def update\_weights(network, row, l\_rate):

    for i in range(len(network)):

        inputs = row[:-1]

        if i != 0:

            inputs = [neuron['output'] for neuron in network[i - 1]]

        for neuron in network[i]:

            for j in range(len(inputs)):

                neuron['weights'][j] -= l\_rate \* neuron['delta'] \* inputs[j]

            neuron['weights'][-1] -= l\_rate \* neuron['delta']

# Train a network for a fixed number of epochs

def train\_network(network, train, l\_rate, n\_epoch, n\_outputs):

    for epoch in range(n\_epoch):

        for row in train:

            outputs = forward\_propagate(network, row)

            expected = [0 for i in range(n\_outputs)]

            expected[int(row[-1])] = 1

            backward\_propagate\_error(network, expected)

            update\_weights(network, row, l\_rate)

# Initialize a network

def initialize\_network(n\_inputs, n\_hidden, n\_outputs):

    network = list()

    n\_hidden2 = n\_hidden \* 2

    hidden\_layer1 = [{'weights':[random.random() for i in range(n\_inputs + 1)]} for i in range(n\_hidden)]

    network.append(hidden\_layer1)

    hidden\_layer2 = [{'weights':[random.random() for i in range(n\_hidden + 1)]} for i in range(n\_hidden2)]

    network.append(hidden\_layer2)

    output\_layer = [{'weights':[random.random() for i in range(n\_hidden2 + 1)]} for i in range(n\_outputs)]

    network.append(output\_layer)

    return network

# Make a prediction with a network

def predict(network, row):

    outputs = forward\_propagate(network, row)

    return outputs.index(max(outputs))

# Backpropagation Algorithm With Stochastic Gradient Descent

def back\_propagation(train, test, l\_rate, n\_epoch, n\_hidden):

    n\_inputs = len(train[0]) - 1

    n\_outputs = len(set([row[-1] for row in train]))

    network = initialize\_network(n\_inputs, n\_hidden, n\_outputs)

    train\_network(network, train, l\_rate, n\_epoch, n\_outputs)

    predictions = list()

    for row in test:

        prediction = predict(network, row)

        predictions.append(prediction)

    return(predictions)

if \_\_name\_\_ == "\_\_main\_\_":

    random.seed(1)

    # generate dataset for train and test

    train\_data = generateDataset()

    test\_data = generateDataset()

    df\_train = pd.DataFrame(train\_data, columns = ['x', 'y', 'label'])

    df\_train.to\_csv('out\_train.csv', index=False)

    df\_test = pd.DataFrame(test\_data, columns = ['x', 'y', 'label'])

    df\_test.to\_csv('out\_test.csv', index=False)

    X\_train = np.stack([df\_train['x'], df\_train['y']]).T

    y\_train = np.stack(df\_train['label'])

    X\_test = np.stack([df\_test['x'], df\_test['y']]).T

    y\_test = np.stack(df\_test['label'])

    df\_test\_filtered = df\_test[df\_test['label'] == 1]

    coordinates\_test = np.stack([df\_test\_filtered['x'], df\_test\_filtered['y']]).T

    labels\_test = np.stack(df\_test\_filtered['label'])

    df\_train\_filtered = df\_train[df\_train['label'] == 1]

    coordinates\_train = np.stack([df\_train\_filtered['x'], df\_train\_filtered['y']]).T

    labels\_train = np.stack(df\_train\_filtered['label'])

    # illustration

    figure\_one = plt.figure(1)

    datasetIllustration(X\_train, y\_train)

    plt.title('Train dataset')

    plt.xlabel('X')

    plt.ylabel('Y')

    plt.legend(loc='upper left')

    figure\_one.show()

    input("Enter any char to continue: ")

    figure\_two = plt.figure(2)

    datasetIllustration(coordinates\_train, labels\_train, show\_circle=True)

    plt.title('Train dataset')

    plt.xlabel('X')

    plt.ylabel('Y')

    plt.legend(loc='upper left')

    figure\_two.show()

    input("Enter any char to continue: ")

    figure\_three = plt.figure(3)

    datasetIllustration(X\_test, y\_test)

    plt.title('Test dataset')

    plt.xlabel('X')

    plt.ylabel('Y')

    plt.legend(loc='upper left')

    figure\_three.show()

    input("Enter any char to continue: ")

    figure\_four = plt.figure(4)

    datasetIllustration(coordinates\_test, labels\_test, show\_circle=True)

    plt.title('Test dataset')

    plt.xlabel('X')

    plt.ylabel('Y')

    plt.legend(loc='upper left')

    figure\_four.show()

    input("Enter any char to continue: ")

    # normalize input variables

    scaler = StandardScaler()

    df\_train[['x', 'y']] = scaler.fit\_transform(df\_train[['x', 'y']])

    df\_test[['x', 'y']] = scaler.fit\_transform(df\_test[['x', 'y']])

    df\_train['label\_2'] = np.where(df\_train['label']==1, int(1), int(0))

    df\_test['label\_2'] = np.where(df\_test['label']==1, 1, 0)

    dataset\_train = np.stack([df\_train['x'], df\_train['y'], df\_train['label\_2']]).T

    dataset\_test = np.stack([df\_test['x'], df\_test['y'], df\_test['label\_2']]).T

    y\_test = np.stack(df\_test['label\_2'])

    # evaluate algorithm

    l\_rate = 0.1

    n\_epoch = 5000

    n\_hidden = 4

    n\_inputs = 2

    n\_outputs = 2

    # Backpropagation Algorithm

    network = initialize\_network(n\_inputs, n\_hidden, n\_outputs)

    train\_network(network, dataset\_train, l\_rate, n\_epoch, n\_outputs)

    predictions = list()

    for row in dataset\_test:

        prediction = predict(network, row)

        predictions.append(prediction)

    # results

    accuracy = accuracy\_score(y\_test, predictions)

    print("accuracy score: {0:.2f}%".format(accuracy\*100))

    print(classification\_report(y\_test, predictions))

    reps = {1: 1, 0: -1}

    y\_test = [reps.get(x,x) for x in y\_test]

    predictions = [reps.get(x,x) for x in predictions]

    figure\_five = plt.figure(5)

    cf\_matrix = confusion\_matrix(y\_test, predictions)

    heatmap = sns.heatmap(cf\_matrix, annot=True, cmap='Blues', fmt='g', xticklabels=np.unique(y\_test), yticklabels=np.unique(y\_test))

    plt.title('Heatmap')

    figure\_five.show()

    input("Enter any char to continue: ")

Part D:

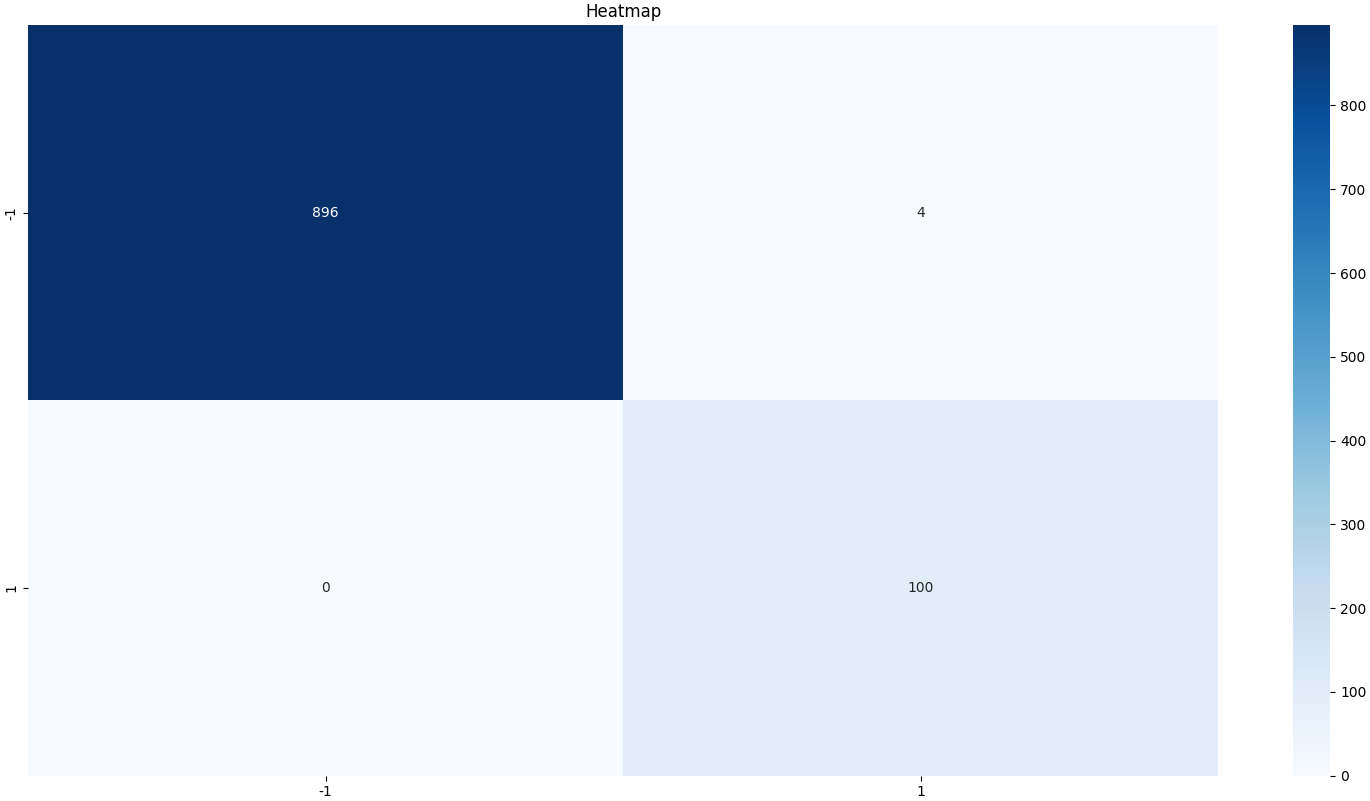
* Dataset:

|  |  |
| --- | --- |
| Class | Number samples |
| Test | |
| -1 | 900 |
| 1 | 100 |
| Train | |
| -1 | 900 |
| 1 | 100 |

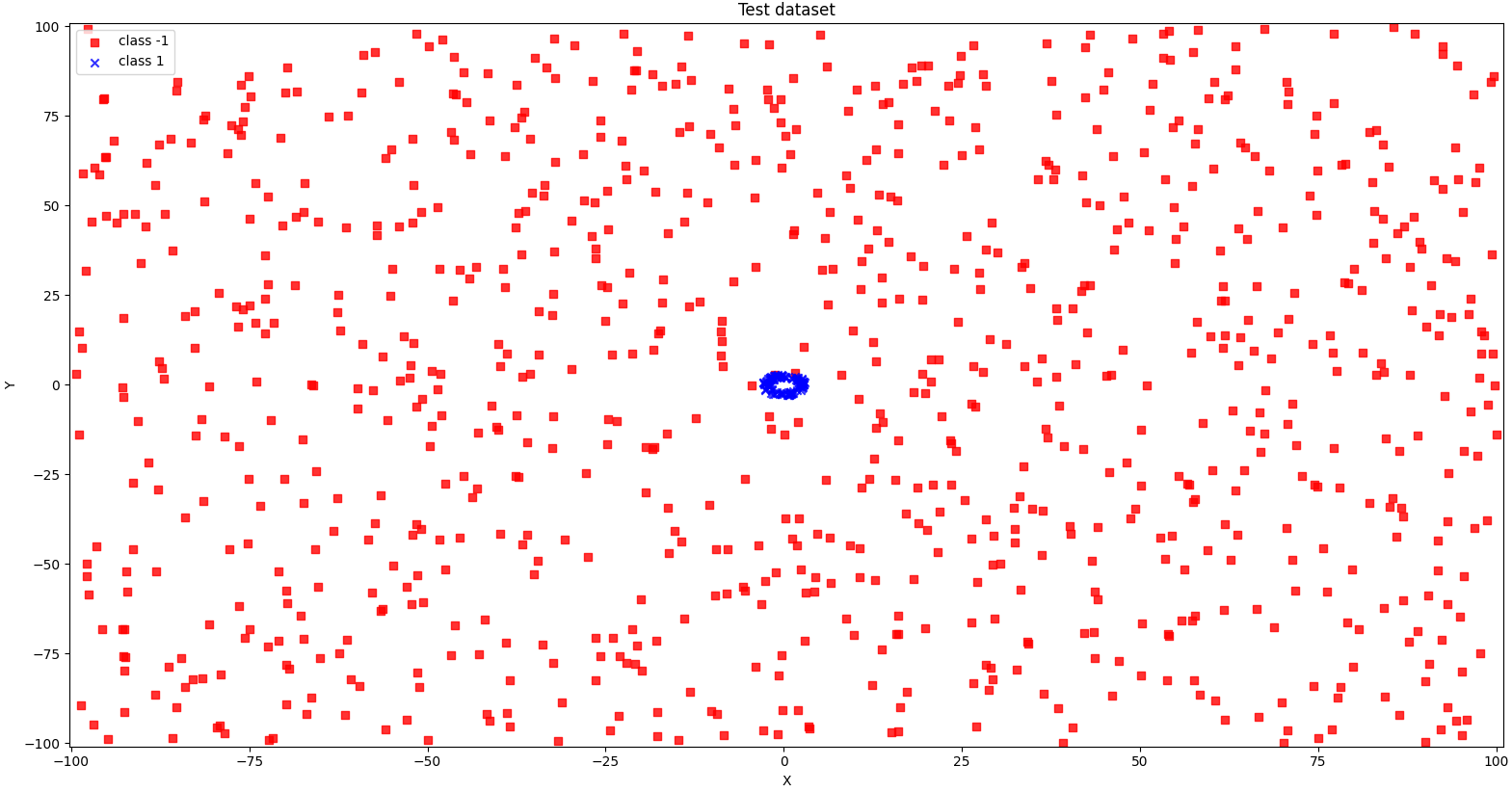
* Classification report:

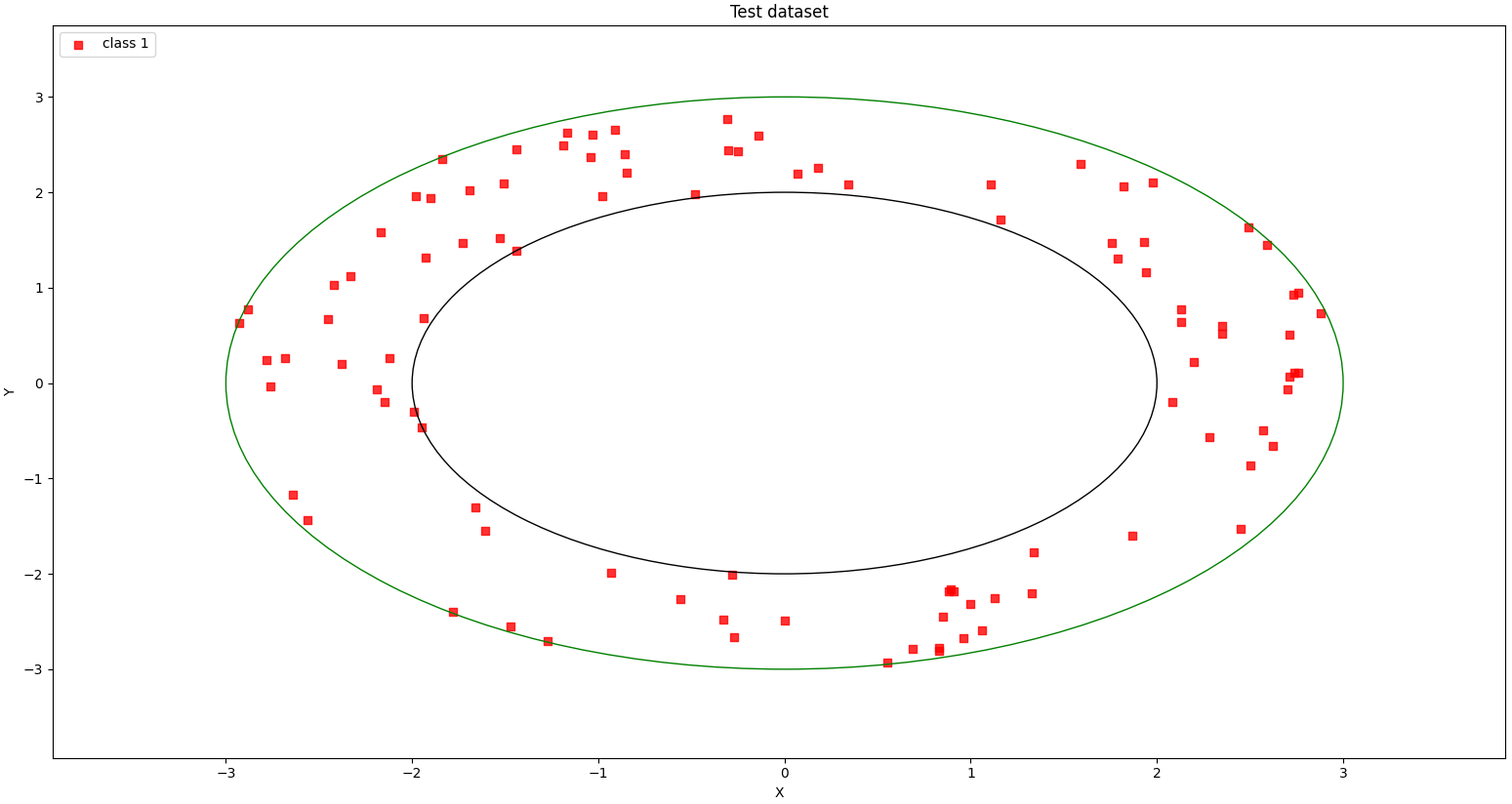
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Precision | Recall | F1-score | Support |
| -1 | 1.00 | 1.00 | 1.00 | 900 |
| 1 | 0.96 | 1.00 | 0.98 | 100 |
|  |  |  |  |  |
| accuracy |  |  | 1.00 | 1000 |
| macro avg | 0.98 | 1.00 | 0.99 | 1000 |
| weighted avg | 1.00 | 1.00 | 1.00 | 1000 |

* Heatmap:

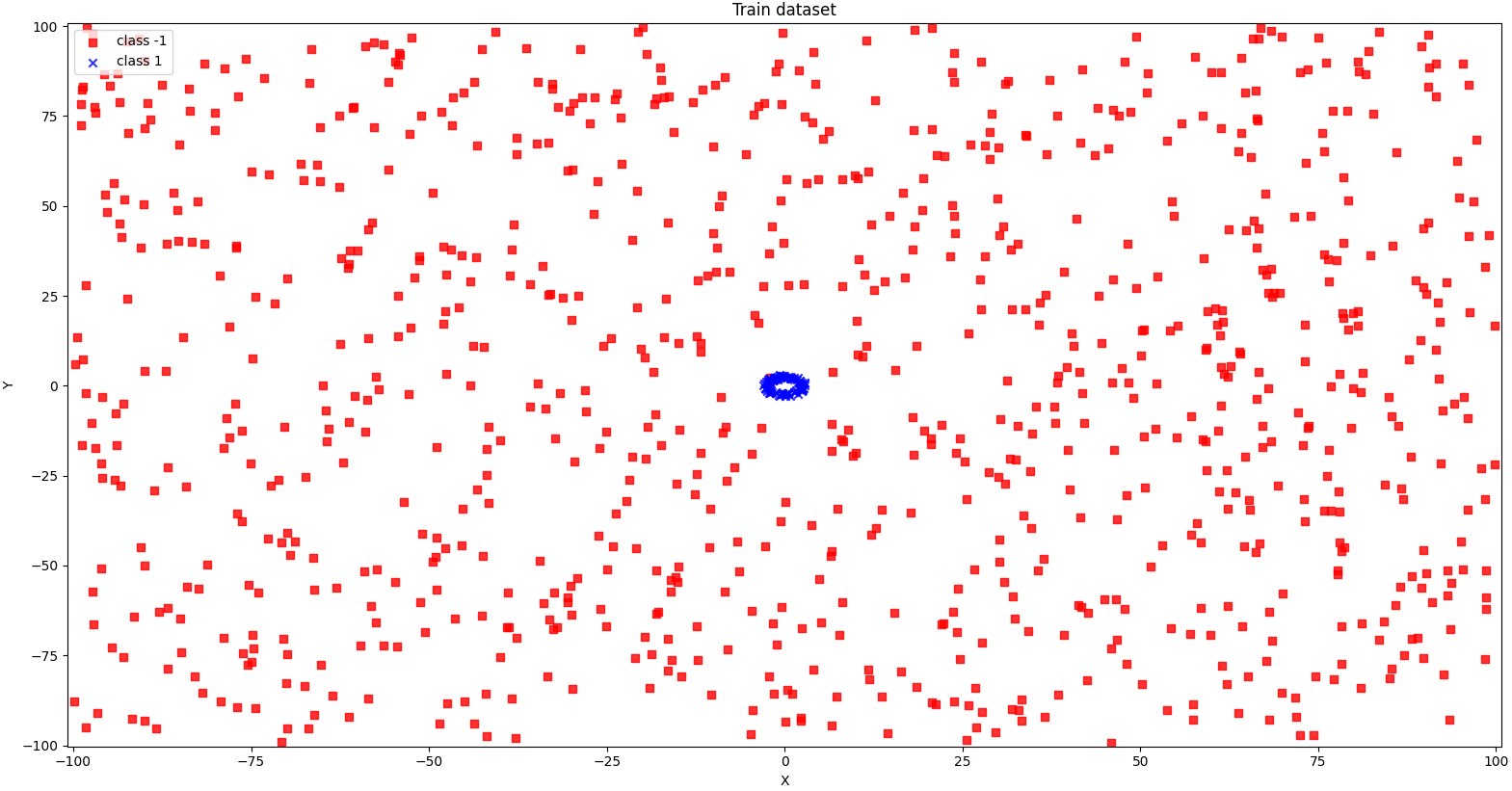
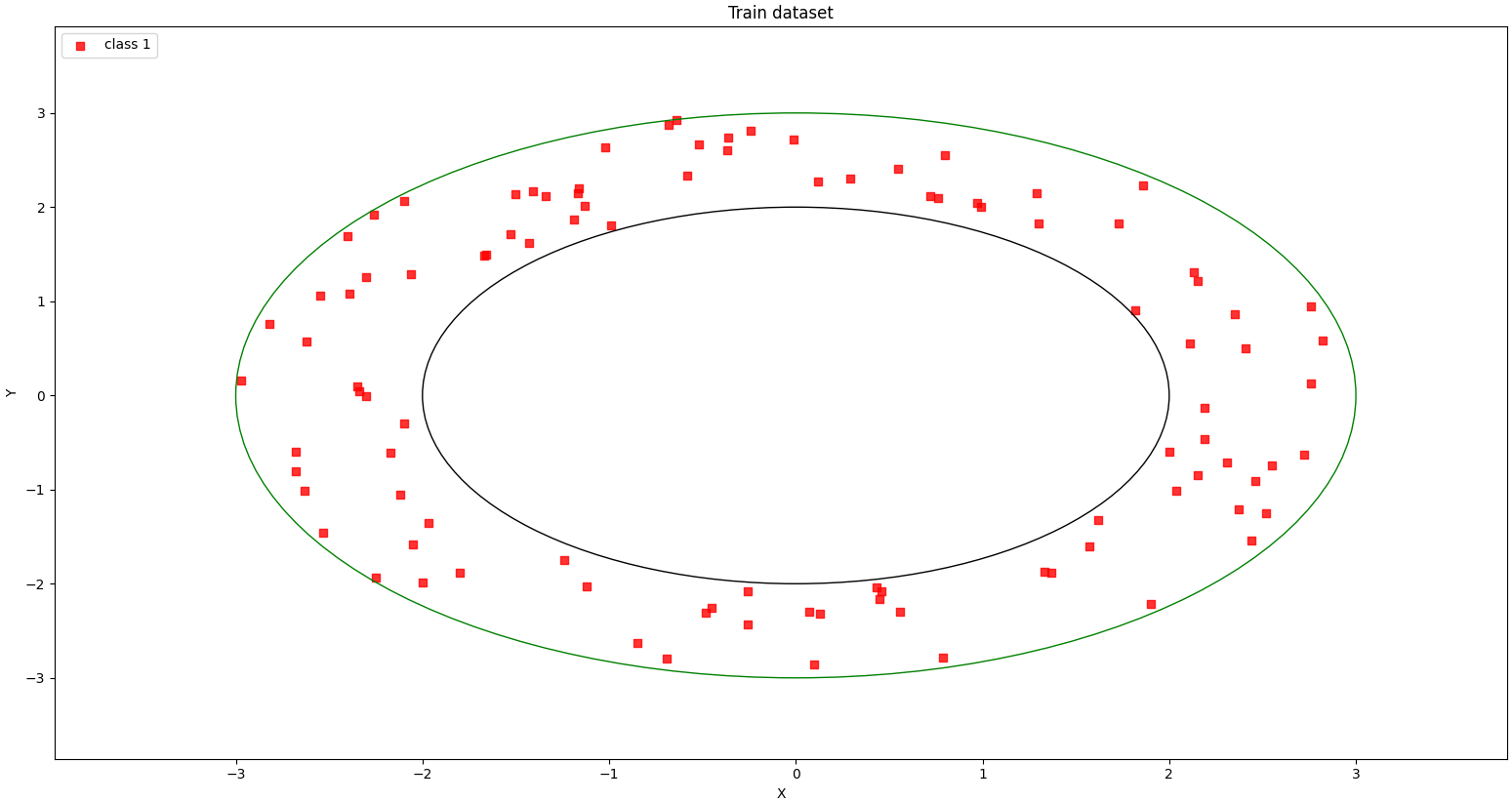


* Accuracy score: 99.6%
* Test illustration:

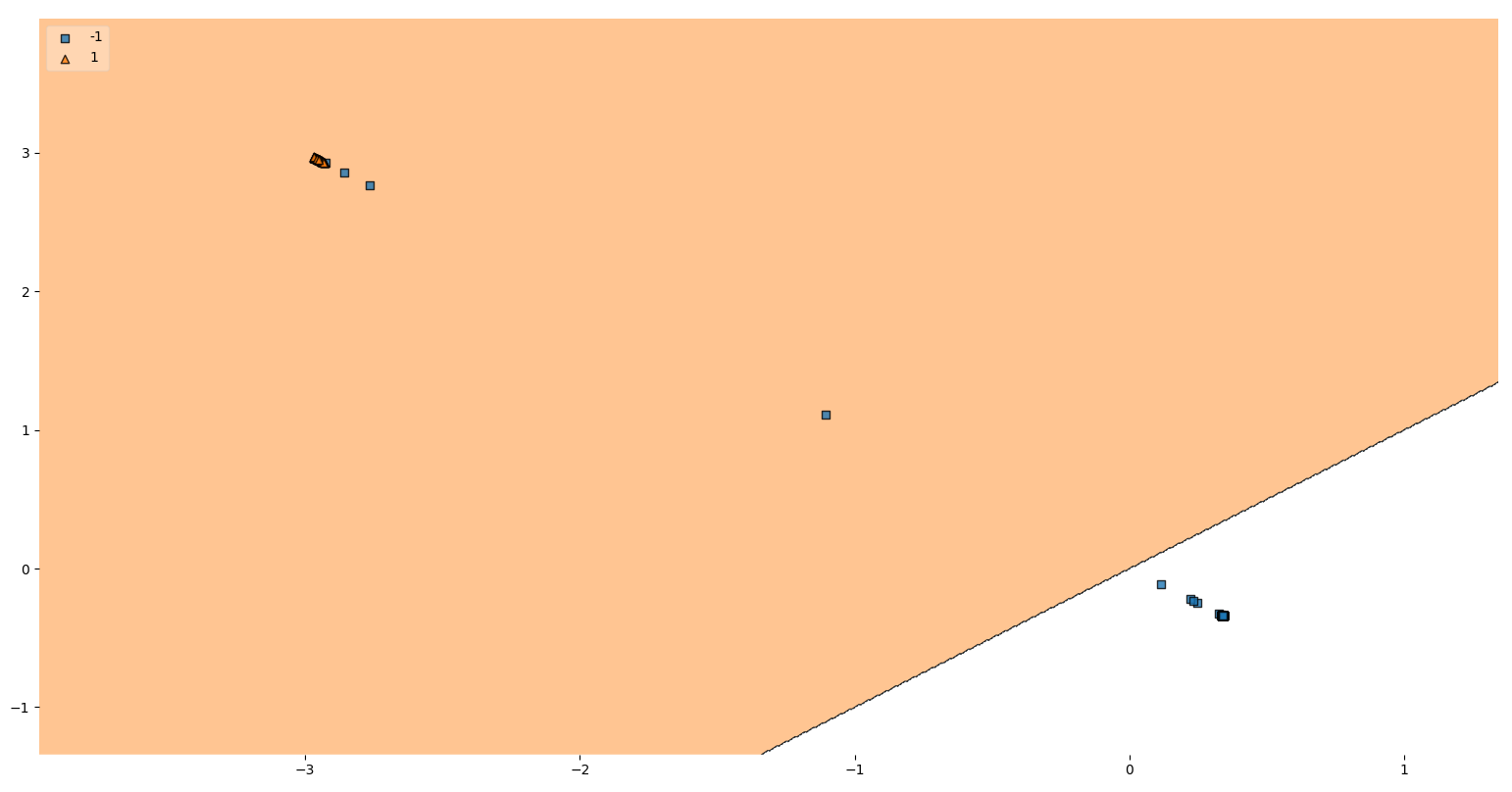




* Train illustration:



* Discussions:



Draw whatever conclusions you think are appropriate from your results and report them.

*The Adaline was almost as accurate as the backpropagation. Based on the upper graph and the results, we can understand that the output layer in the backpropagation can be replaced by the Adaline.*

* Code:

import random

import pandas as pd

import numpy as np

import seaborn as sns

import matplotlib.pyplot as plt

from math import exp

from matplotlib.colors import ListedColormap

from sklearn.preprocessing import StandardScaler

from sklearn.metrics import classification\_report, confusion\_matrix, accuracy\_score

from mlxtend.plotting import plot\_decision\_regions

max\_limit = 10000

min\_limit = -10000

num\_samples = 1000

def generateDataset():

    one\_samples = 0

    zero\_samples = 0

    data = []

    while  (one\_samples + zero\_samples ) < num\_samples:

        n = random.randint(min\_limit, max\_limit)

        m = random.randint(min\_limit, max\_limit)

        x = m/100

        y = n/100

        circle = pow(x, 2) + pow(y, 2)

        if (circle <= 9 and circle >= 4):

            one\_samples += 1

            data.append([x, y, 1])

        elif zero\_samples < 900:

            zero\_samples += 1

            data.append([x, y, -1])

    return data

def datasetIllustration(X, y, show\_circle=False, resolution=0.02):

    # setup marker generator and color map

    markers = ('s', 'x', 'o', '^', 'v')

    colors = ('red', 'blue', 'lightgreen', 'gray', 'cyan')

    cmap = ListedColormap(colors[:len(np.unique(y))])

    # plot the decision surface

    x1\_min, x1\_max = X[:,  0].min() - 1, X[:, 0].max() + 1

    x2\_min, x2\_max = X[:, 1].min() - 1, X[:, 1].max() + 1

    xx1, xx2 = np.meshgrid(np.arange(x1\_min, x1\_max, resolution),

    np.arange(x2\_min, x2\_max, resolution))

    plt.xlim(xx1.min(), xx1.max())

    plt.ylim(xx2.min(), xx2.max())

    # plot class samples

    for idx, cl in enumerate(np.unique(y)):

        plt.scatter(x=X[y == cl, 0], y=X[y == cl, 1],

        alpha=0.8, c=cmap(idx),

        marker=markers[idx], label='class ' + str(cl))

    # circles

    if show\_circle:

        circle9 = plt.Circle((0, 0), 2, color='black', fill=False)

        circle4 = plt.Circle((0, 0), 3, color='green', fill=False)

        plt.gca().add\_patch(circle4)

        plt.gca().add\_patch(circle9)

# Calculate neuron activation for an input

def activate(weights, inputs):

    activation = weights[-1]

    for i in range(len(weights)-1):

        activation += weights[i] \* inputs[i]

    return activation

# Transfer neuron activation

def transfer(activation):

    return 1.0 / (1.0 + exp(-activation))

# Forward propagate input to a network output

def forward\_propagate(network, row):

    inputs = row

    for layer in network:

        new\_inputs = []

        for neuron in layer:

            activation = activate(neuron['weights'], inputs)

            neuron['output'] = transfer(activation)

            new\_inputs.append(neuron['output'])

        inputs = new\_inputs

    return inputs

# Calculate the derivative of an neuron output

def transfer\_derivative(output):

    return output \* (1.0 - output)

# Backpropagate error and store in neurons

def backward\_propagate\_error(network, expected):

    for i in reversed(range(len(network))):

        layer = network[i]

        errors = list()

        if i != len(network)-1:

            for j in range(len(layer)):

                error = 0.0

                for neuron in network[i + 1]:

                    error += (neuron['weights'][j] \* neuron['delta'])

                errors.append(error)

        else:

            for j in range(len(layer)):

                neuron = layer[j]

                errors.append(neuron['output'] - expected[j])

        for j in range(len(layer)):

            neuron = layer[j]

            neuron['delta'] = errors[j] \* transfer\_derivative(neuron['output'])

# Update network weights with error

def update\_weights(network, row, l\_rate):

    for i in range(len(network)):

        inputs = row[:-1]

        if i != 0:

            inputs = [neuron['output'] for neuron in network[i - 1]]

        for neuron in network[i]:

            for j in range(len(inputs)):

                neuron['weights'][j] -= l\_rate \* neuron['delta'] \* inputs[j]

            neuron['weights'][-1] -= l\_rate \* neuron['delta']

# Train a network for a fixed number of epochs

def train\_network(network, train, l\_rate, n\_epoch, n\_outputs):

    for \_ in range(n\_epoch):

        for row in train:

            \_ = forward\_propagate(network, row)

            expected = [0 for \_ in range(n\_outputs)]

            expected[int(row[-1])] = 1

            backward\_propagate\_error(network, expected)

            update\_weights(network, row, l\_rate)

# Initialize a network

def initialize\_network(n\_inputs, n\_hidden, n\_outputs):

    network = list()

    n\_hidden2 = n\_hidden \* 2

    hidden\_layer1 = [{'weights':[random.random() for i in range(n\_inputs + 1)]} for i in range(n\_hidden)]

    network.append(hidden\_layer1)

    hidden\_layer2 = [{'weights':[random.random() for i in range(n\_hidden + 1)]} for i in range(n\_hidden2)]

    network.append(hidden\_layer2)

    output\_layer = [{'weights':[random.random() for i in range(n\_hidden2 + 1)]} for i in range(n\_outputs)]

    network.append(output\_layer)

    return network

# Make a prediction with a network

def predict(network, row):

    outputs = forward\_propagate(network, row)

    return outputs.index(max(outputs))

# Backpropagation Algorithm With Stochastic Gradient Descent

def back\_propagation(train, test, l\_rate, n\_epoch, n\_hidden):

    n\_inputs = len(train[0]) - 1

    n\_outputs = len(set([row[-1] for row in train]))

    network = initialize\_network(n\_inputs, n\_hidden, n\_outputs)

    train\_network(network, train, l\_rate, n\_epoch, n\_outputs)

    predictions = list()

    for row in test:

        prediction = predict(network, row)

        predictions.append(prediction)

    return(predictions)

class ADAptiveLInearNEuron(object):

    """

    ADALINE classifier.

    Parameters

    -----------

    eta  - learning rate (between 0.0 and 1.0). The default value is 0.01.

    n\_iter - the actual number of iterations before reaching the stopping criterion. The default value is 15.

    """

    def \_\_init\_\_(self, eta = 0.01, n\_iter = 15):

        self.eta = eta

        self.n\_iter = n\_iter

    def fit(self, X, y):

        """

        Fit training data (Gradient Descent).

        Parameters

        -----------

        X - training data.

        y - target values.

        Attributes

        -----------

        weights - the weight vector.

        errors - number of misclassifications in every epoch.

        Returns

        -----------

        Returns an instance of self.

        """

        self.weights = np.zeros(1 + X.shape[1])

        for \_ in range(self.n\_iter):

            output\_model = self.net\_input(X)

            errors = (y - output\_model)

            # update rule

            self.weights[1:] += self.eta \* X.T.dot(errors)

            self.weights[0] += self.eta \* errors.sum()

        return self

    def net\_input(self, X):

        """

        Calculate net input, sum of weighted input signals.

        y = SUM(X\*w) + theta  [https://en.wikipedia.org/wiki/ADALINE]

        Parameters

        -----------

        X - the input vector.

        Attributes

        -----------

        weights - the weight vector.

        weights[0] (theta) - some constant.

        Returns

        -----------

        Return the output of the model.

        """

        return np.dot(X, self.weights[1:]) + self.weights[0]

    def activation(self, X):

        """ Compute linear activation """

        return self.net\_input(X)

    def predict(self, X):

        """ Return class label after unit step """

        return np.where(self.activation(X) >= 0.0, 1, -1)

if \_\_name\_\_ == "\_\_main\_\_":

    random.seed(1)

    # generate dataset for train and test

    train\_data = generateDataset()

    test\_data = generateDataset()

    df\_train = pd.DataFrame(train\_data, columns = ['x', 'y', 'label'])

    df\_train.to\_csv('out\_train.csv', index=False)

    df\_test = pd.DataFrame(test\_data, columns = ['x', 'y', 'label'])

    df\_test.to\_csv('out\_test.csv', index=False)

    X\_train = np.stack([df\_train['x'], df\_train['y']]).T

    y\_train = np.stack(df\_train['label'])

    X\_test = np.stack([df\_test['x'], df\_test['y']]).T

    y\_test = np.stack(df\_test['label'])

    df\_test\_filtered = df\_test[df\_test['label'] == 1]

    coordinates\_test = np.stack([df\_test\_filtered['x'], df\_test\_filtered['y']]).T

    labels\_test = np.stack(df\_test\_filtered['label'])

    df\_train\_filtered = df\_train[df\_train['label'] == 1]

    coordinates\_train = np.stack([df\_train\_filtered['x'], df\_train\_filtered['y']]).T

    labels\_train = np.stack(df\_train\_filtered['label'])

    # illustration

    figure\_one = plt.figure(1)

    datasetIllustration(X\_train, y\_train)

    plt.title('Train dataset')

    plt.xlabel('X')

    plt.ylabel('Y')

    plt.legend(loc='upper left')

    figure\_one.show()

    input("Enter any char to continue: ")

    figure\_two = plt.figure(2)

    datasetIllustration(coordinates\_train, labels\_train, show\_circle=True)

    plt.title('Train dataset')

    plt.xlabel('X')

    plt.ylabel('Y')

    plt.legend(loc='upper left')

    figure\_two.show()

    input("Enter any char to continue: ")

    figure\_three = plt.figure(3)

    datasetIllustration(X\_test, y\_test)

    plt.title('Test dataset')

    plt.xlabel('X')

    plt.ylabel('Y')

    plt.legend(loc='upper left')

    figure\_three.show()

    input("Enter any char to continue: ")

    figure\_four = plt.figure(4)

    datasetIllustration(coordinates\_test, labels\_test, show\_circle=True)

    plt.title('Test dataset')

    plt.xlabel('X')

    plt.ylabel('Y')

    plt.legend(loc='upper left')

    figure\_four.show()

    input("Enter any char to continue: ")

    # normalize input variables

    scaler = StandardScaler()

    df\_train[['x', 'y']] = scaler.fit\_transform(df\_train[['x', 'y']])

    df\_test[['x', 'y']] = scaler.fit\_transform(df\_test[['x', 'y']])

    df\_train['label\_2'] = np.where(df\_train['label']==1, int(1), int(0))

    df\_test['label\_2'] = np.where(df\_test['label']==1, 1, 0)

    dataset\_train = np.stack([df\_train['x'], df\_train['y'], df\_train['label\_2']]).T

    dataset\_test = np.stack([df\_test['x'], df\_test['y'], df\_test['label\_2']]).T

    y\_test = np.stack(df\_test['label\_2'])

    # evaluate algorithm

    l\_rate = 0.1

    n\_epoch = 5000

    n\_hidden = 4

    n\_inputs = 2

    n\_outputs = 2

    # Backpropagation Algorithm

    network = initialize\_network(n\_inputs, n\_hidden, n\_outputs)

    train\_network(network, dataset\_train, l\_rate, n\_epoch, n\_outputs)

    data = []

    for row in dataset\_train:

        outputs = forward\_propagate(network, row)

        data.append([outputs[0], outputs[1], 1 if row[2] == 1 else -1])

    df\_train\_backpropagation = pd.DataFrame(data, columns = ['node\_1', 'node\_2', 'label'])

    df\_train\_backpropagation.to\_csv('out\_train\_backpropagation.csv', index=False)

    data = []

    for row in dataset\_test:

        outputs = forward\_propagate(network, row)

        data.append([outputs[0], outputs[1], 1 if row[2] == 1 else -1])

    df\_test\_backpropagation = pd.DataFrame(data, columns = ['node\_1', 'node\_2', 'label'])

    df\_test\_backpropagation.to\_csv('out\_test\_backpropagation.csv', index=False)

    df\_train\_backpropagation[['node\_1', 'node\_2']] = scaler.fit\_transform(df\_train\_backpropagation[['node\_1', 'node\_2']])

    df\_test\_backpropagation[['node\_1', 'node\_2']] = scaler.fit\_transform(df\_test\_backpropagation[['node\_1', 'node\_2']])

    X\_train = np.stack([df\_train\_backpropagation['node\_1'], df\_train\_backpropagation['node\_2']]).T

    y\_train = np.stack(df\_train\_backpropagation['label'])

    X\_test = np.stack([df\_test\_backpropagation['node\_1'], df\_test\_backpropagation['node\_2']]).T

    y\_test = np.stack(df\_test\_backpropagation['label'])

    # start algorithm

    aln\_clf = ADAptiveLInearNEuron(eta = 0.1, n\_iter = 25)

    aln\_clf.fit(X\_train, y\_train)

    aln\_predictions = aln\_clf.predict(X\_test)

    # results

    accuracy = accuracy\_score(y\_test, aln\_predictions)

    print("accuracy score: {0:.2f}%".format(accuracy\*100))

    print(classification\_report(y\_test, aln\_predictions))

    figure\_five = plt.figure(5)

    cf\_matrix = confusion\_matrix(y\_test, aln\_predictions)

    heatmap = sns.heatmap(cf\_matrix, annot=True, cmap='Blues', fmt='g', xticklabels=np.unique(y\_test), yticklabels=np.unique(y\_test))

    plt.title('Heatmap')

    figure\_five.show()

    input("Enter any char to continue: ")

    figure\_six = plt.figure(6)

    fig = plot\_decision\_regions(X=X\_test, y=y\_test, clf=aln\_clf, legend=2)

    figure\_six.show()

    input("Enter any char to finish: ")