Project 1: part C and D

first name and T.Z. numbers

second name and T.Z. numbers

Part C:

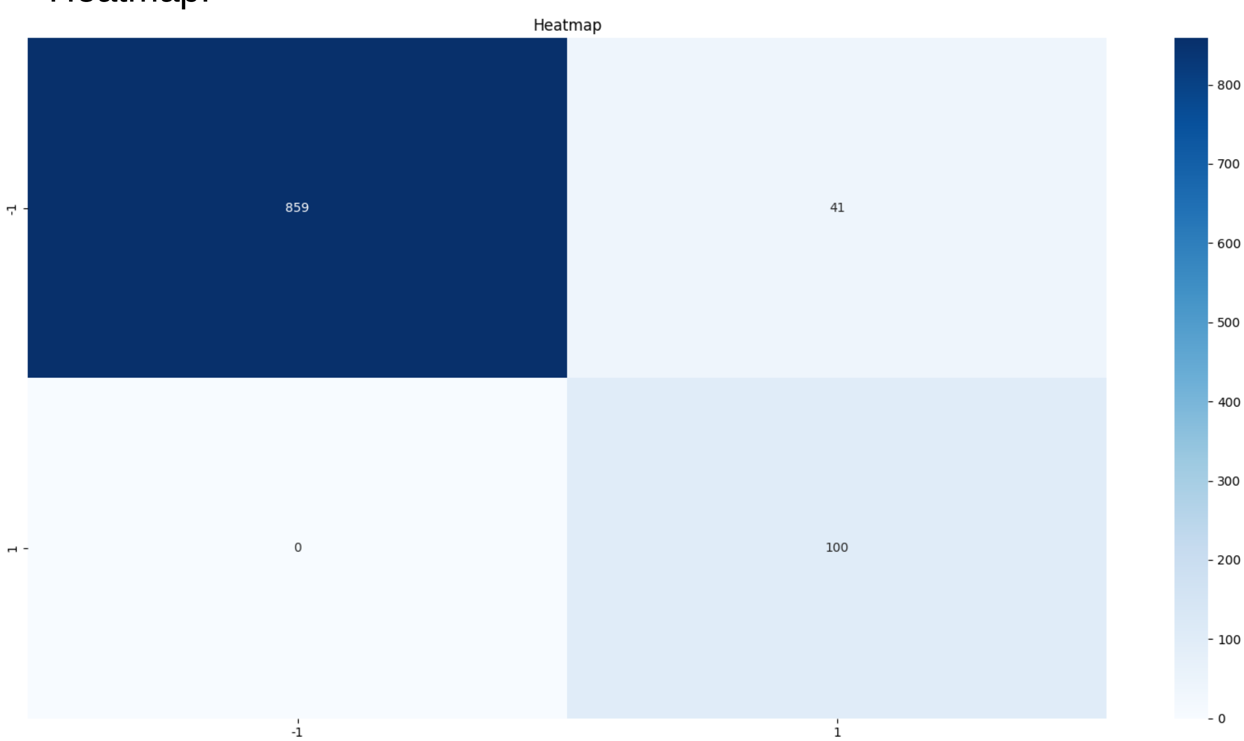
- Dataset:

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

- Classification report:

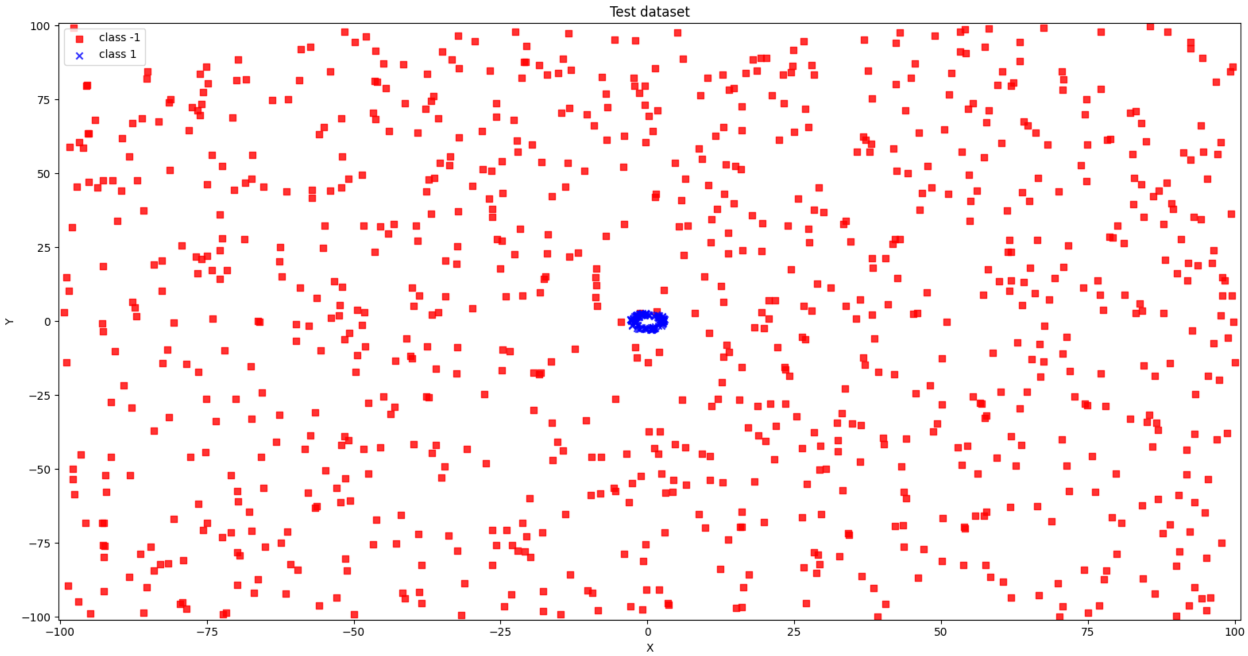
	Precision	Recall	F1-score	Support
-1	1.00	0.95	0.98	900
1	0.71	1.00	0.83	100
accuracy			0.96	1000
macro avg	0.85	0.98	0.90	1000
weighted avg	0.97	0.96	0.96	1000

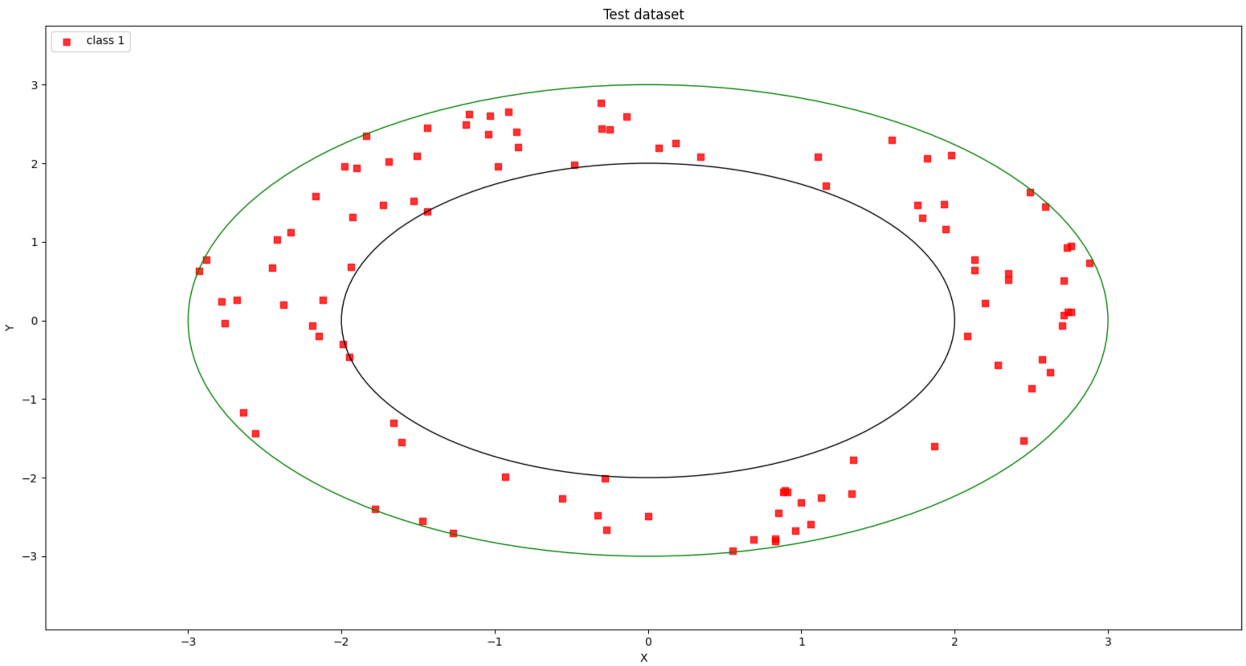
- Heatmap:



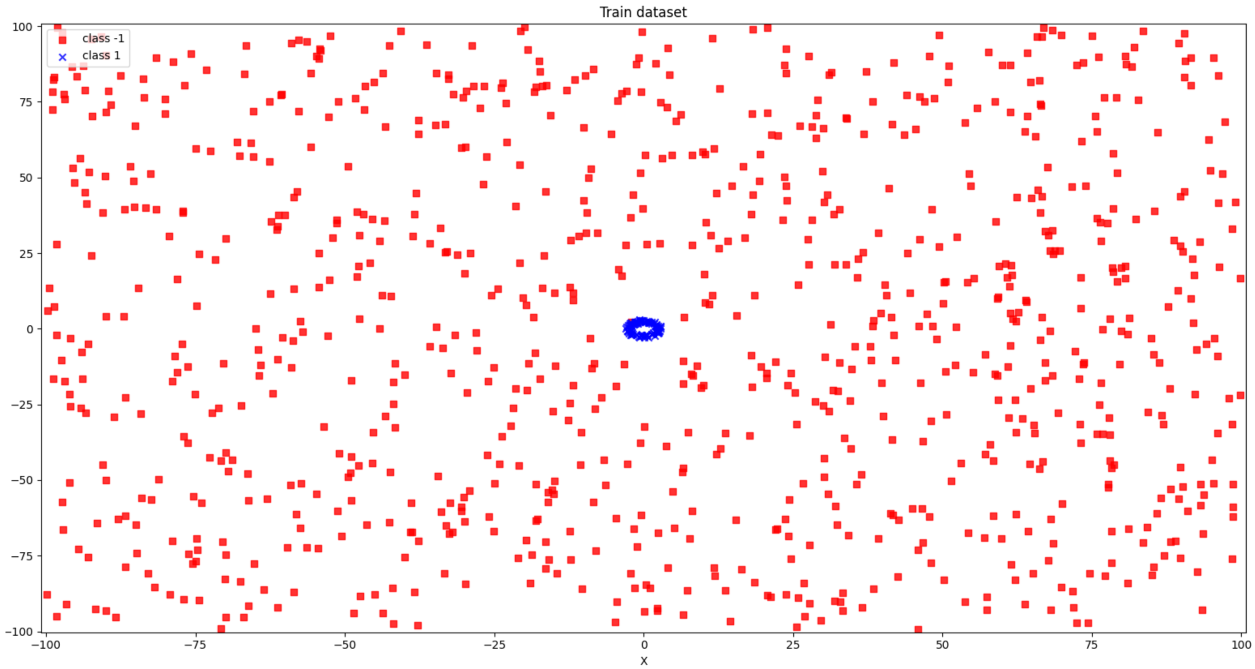
- Accuracy score: 95.9%

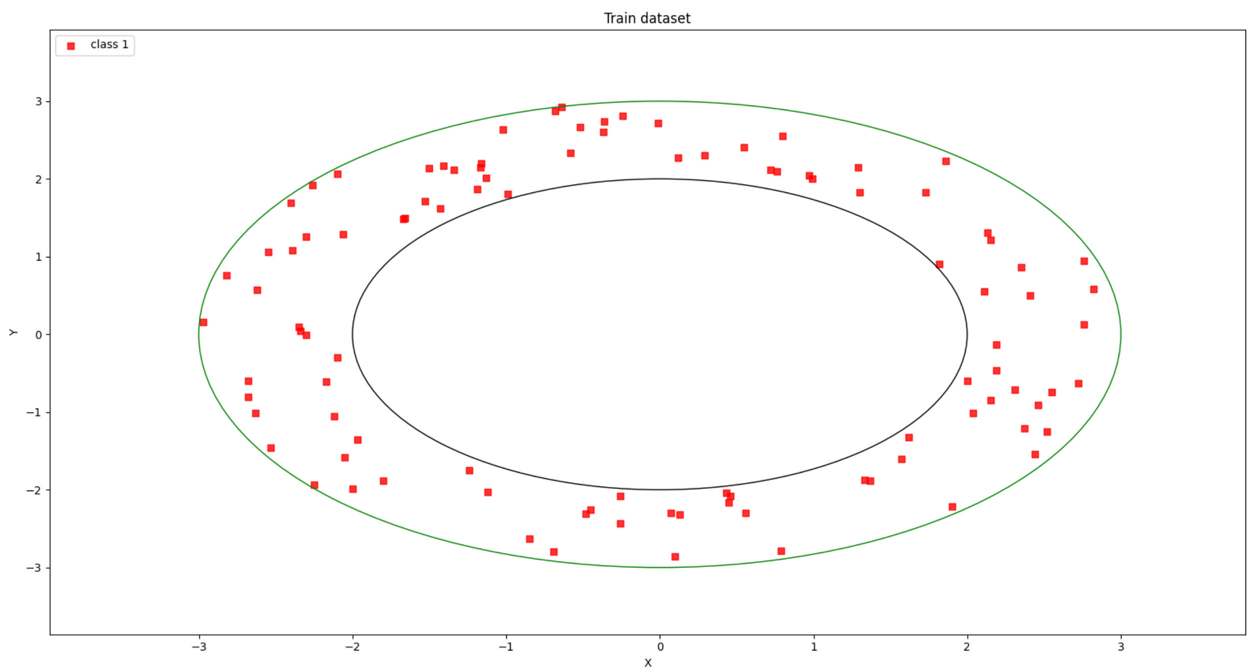
- Test illustration:





- Train illustration:





- Code:

```
import random
import pandas as pd
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt

from random import seed
from math import exp
from matplotlib.colors import ListedColormap
from sklearn.preprocessing import MinMaxScaler
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')
```

```python
    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)
```

```python
            update_weights(network, row, l_rate)

# Initialize a network
def initialize_network(n_inputs, n_hidden, n_outputs):
    network = list()
    hidden_layer = [{'weights':[random.random() for i in range(n_inputs +
1)]} for i in range(n_hidden)]
    network.append(hidden_layer)
    output_layer = [{'weights':[random.random() for i in range(n_hidden +
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__":
    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 = MinMaxScaler()
    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 = 0.3
    n_epoch = 500
    n_hidden = 8
    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)
    print("network", network)

    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:

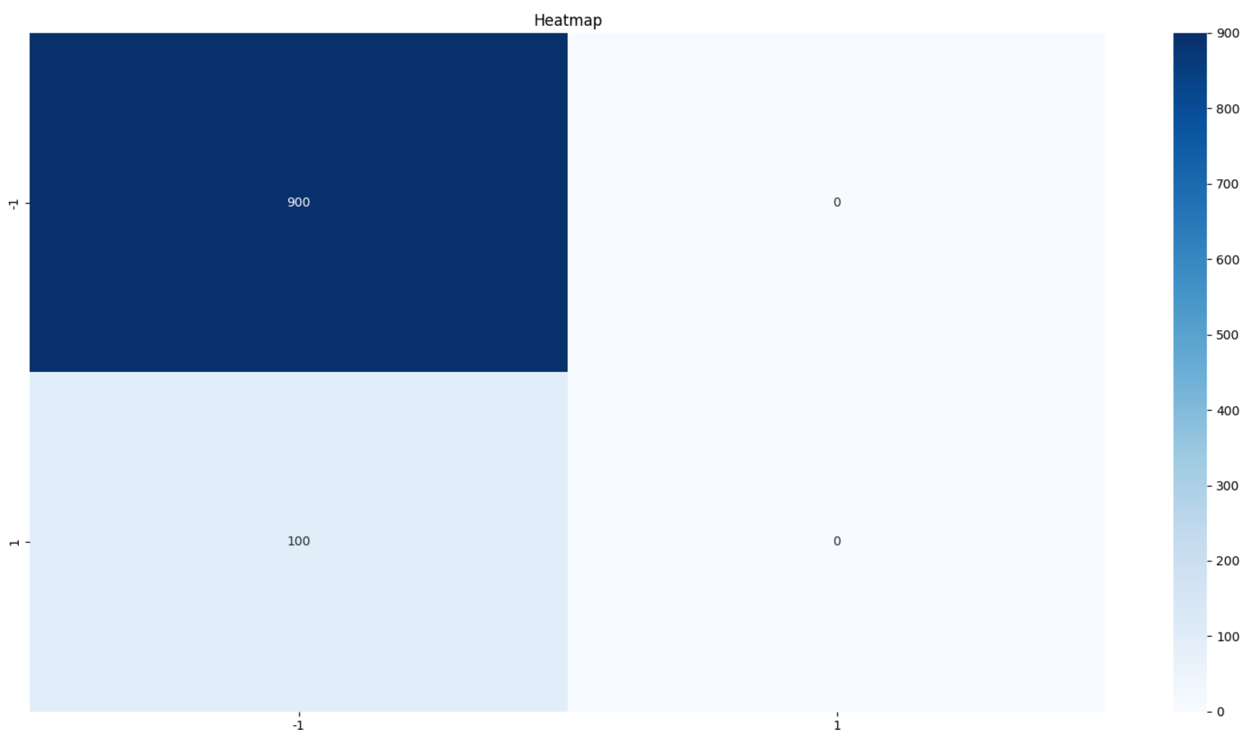
- Dataset:

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

- Classification report:

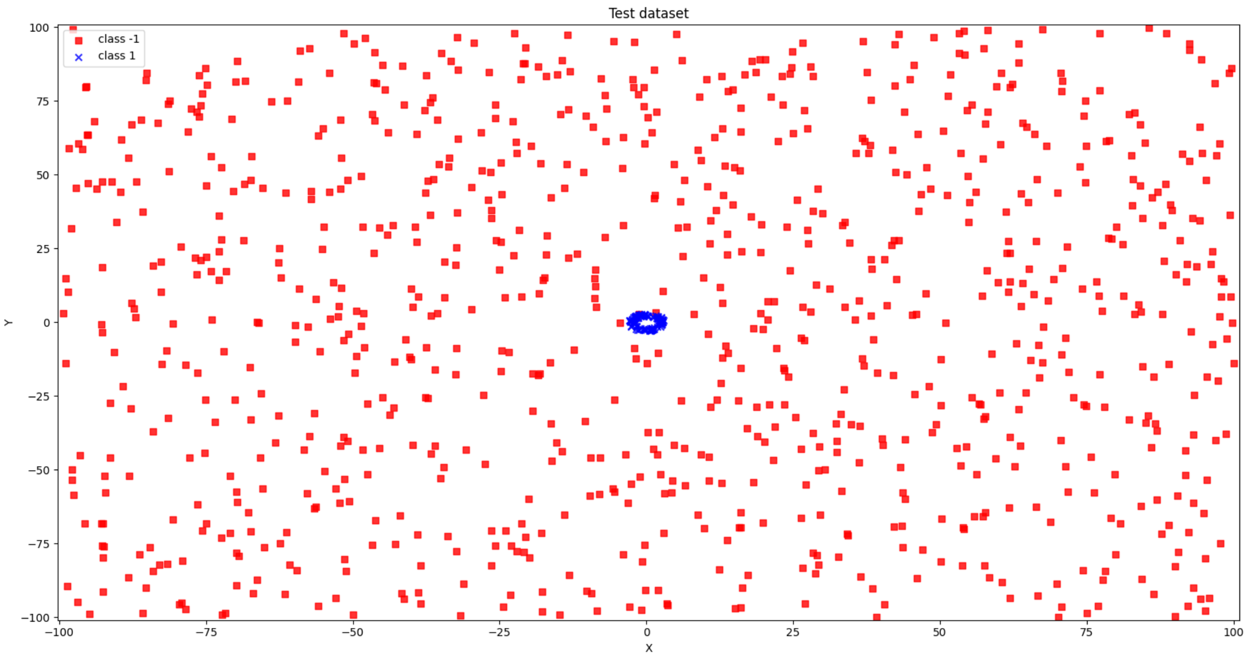
	Precision	Recall	F1-score	Support
-1	0.90	1.00	0.95	900
1	0.00	0.00	0.00	100
accuracy			0.90	1000
macro avg	0.45	0.50	0.47	1000
weighted avg	0.81	0.90	0.85	1000

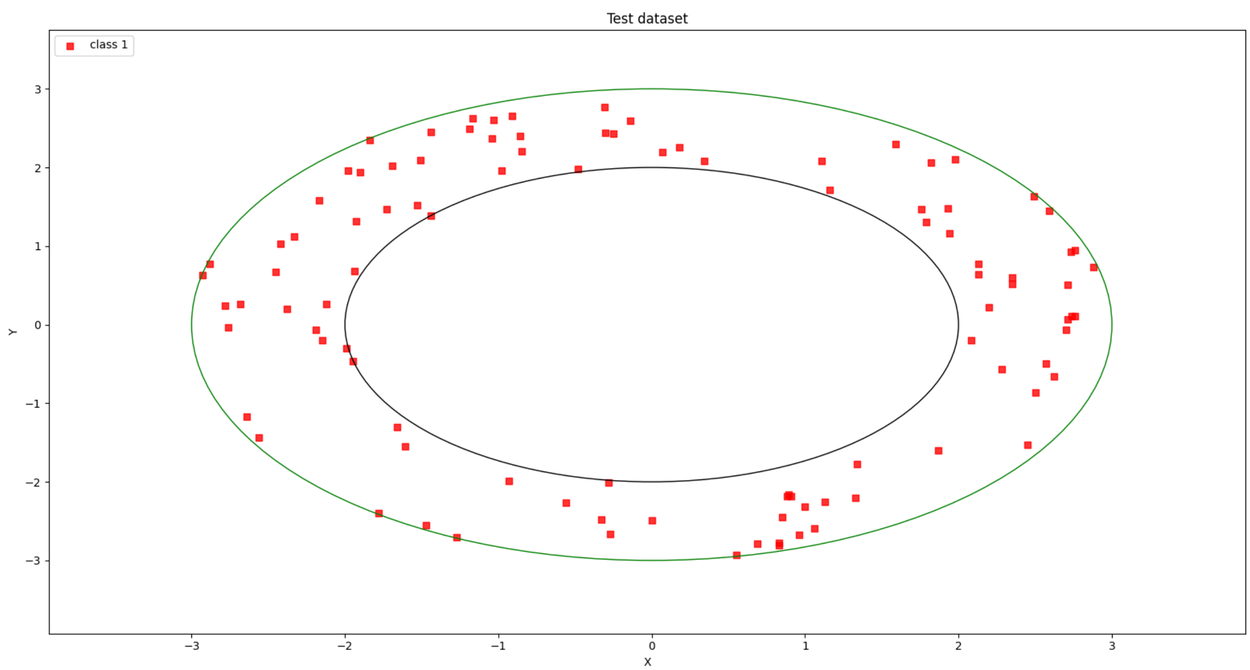
- Heatmap:



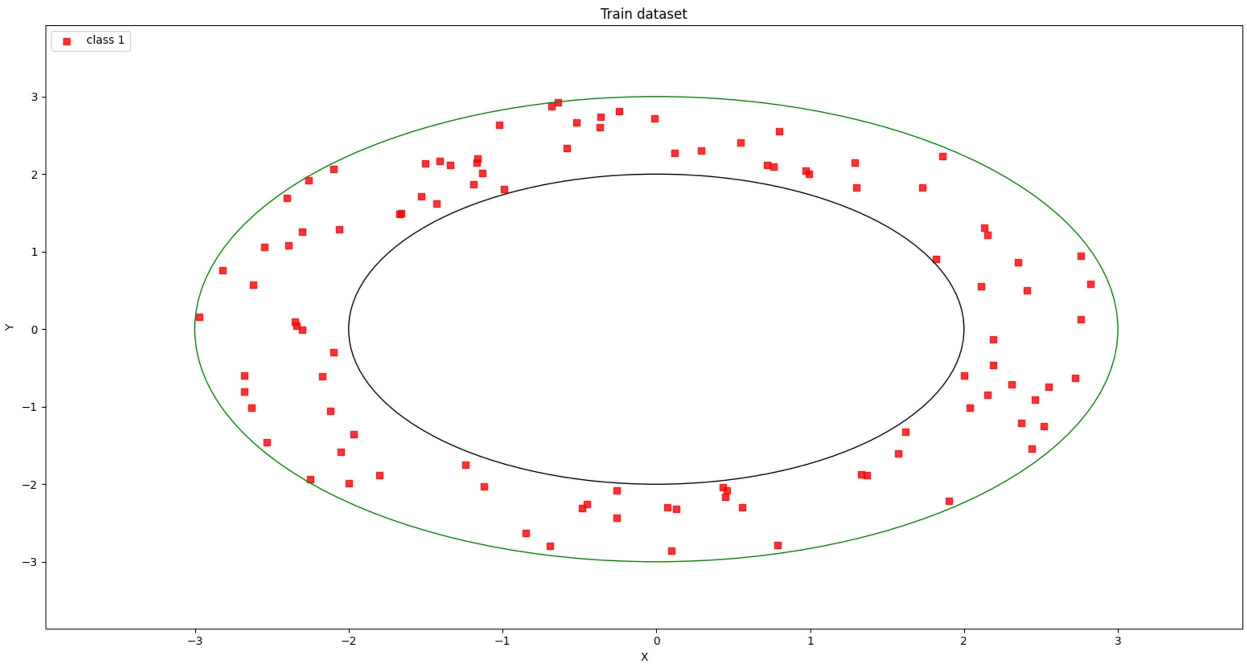
- Accuracy score: 90.0%

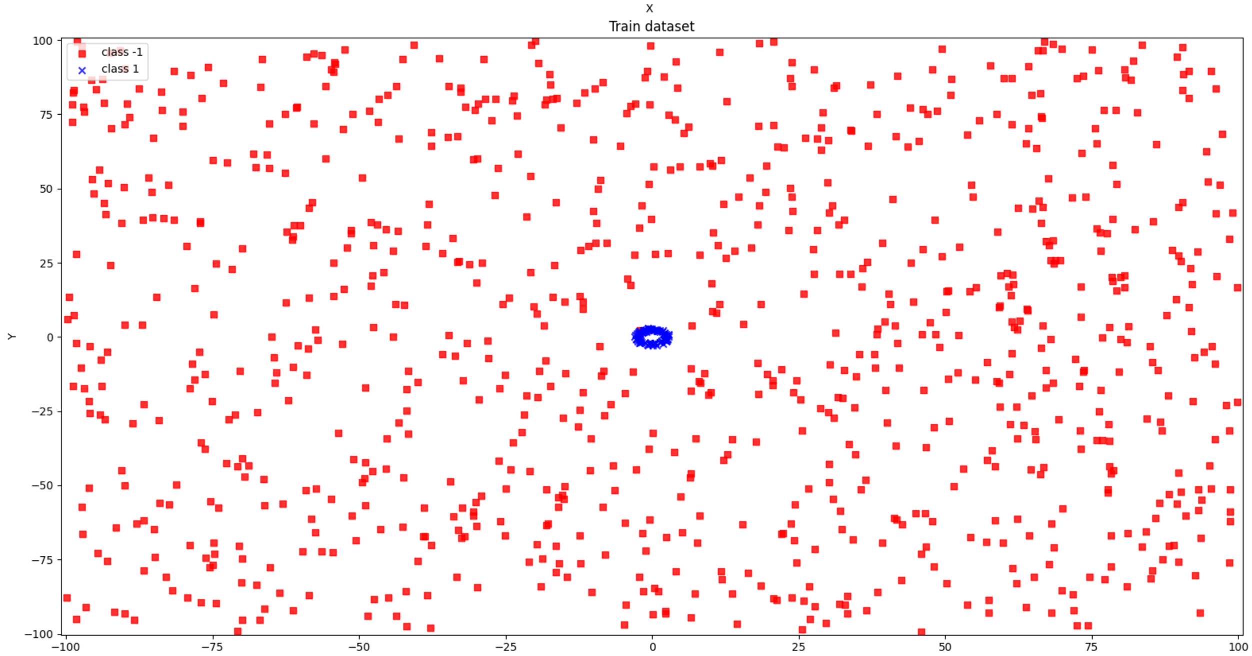
- Test illustration:





- Train illustration:





- Discussions:

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

However, Adaline was still unable to identify the classes despite getting 90%.

- Code:

```
import random
import pandas as pd
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt

from random import seed
from math import exp
from matplotlib.colors import ListedColormap
from sklearn.preprocessing import MinMaxScaler
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')
```

```python
    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()
    hidden_layer = [{'weights':[random.random() for i in range(n_inputs +
1)]} for i in range(n_hidden)]
    network.append(hidden_layer)
    output_layer = [{'weights':[random.random() for i in range(n_hidden +
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__":
    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 = MinMaxScaler()
    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 = 0.3
    n_epoch = 500
    n_hidden = 8
    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)
    print("network", network)

    predictions = list()
    for row in dataset_test:
        prediction = predict(network, row)
        predictions.append(prediction)

    data = []
    for row in dataset_train:
        outputs = forward_propagate(network, row)
        if (row[2]==1):
            data.append([int(outputs[0]*100), int(outputs[1]*100), 1])
        else:
            data.append([int(outputs[0]*100), int(outputs[1]*100), -1])

    df_train_backpropagation = pd.DataFrame(data, columns = ['x', 'y',
'label'])
    df_train_backpropagation.to_csv('out_train_backpropagation.csv',
index=False)

    data = []
    for row in dataset_test:
        outputs = forward_propagate(network, row)
        if (row[2]==1):
            data.append([int(outputs[0]*100), int(outputs[1]*100), 1])
        else:
            data.append([int(outputs[0]*100), int(outputs[1]*100), -1])


    df_test_backpropagation = pd.DataFrame(data, columns = ['x', 'y',
'label'])
    df_test_backpropagation.to_csv('out_test_backpropagation.csv',
index=False)
```

```python
    X_train = np.stack([df_train_backpropagation['x'],
df_train_backpropagation['y']]).T
    y_train = np.stack(df_train_backpropagation['label'])

    X_test = np.stack([df_test_backpropagation['x'],
df_test_backpropagation['y']]).T
    y_test = np.stack(df_test_backpropagation['label'])

    # start algorithm
    aln_clf = ADAptiveLInearNEuron(eta = 0.01, n_iter = 15)
    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: ")
```