Project 1: part A and B

first name and T.Z. numbers

second name and T.Z. numbers

Part A:

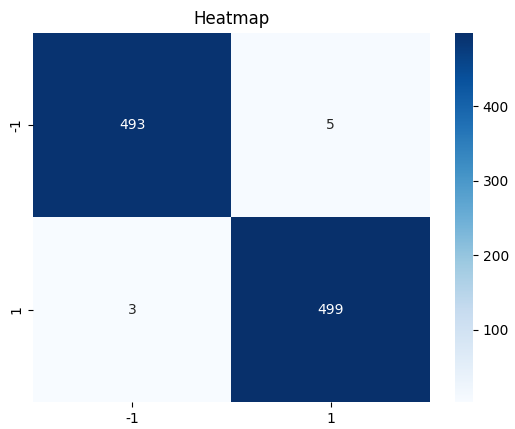
* Dataset:

|  |  |
| --- | --- |
| Class | Number samples |
| Test | |
| -1 | 498 |
| 1 | 502 |
| Train | |
| -1 | 513 |
| 1 | 487 |

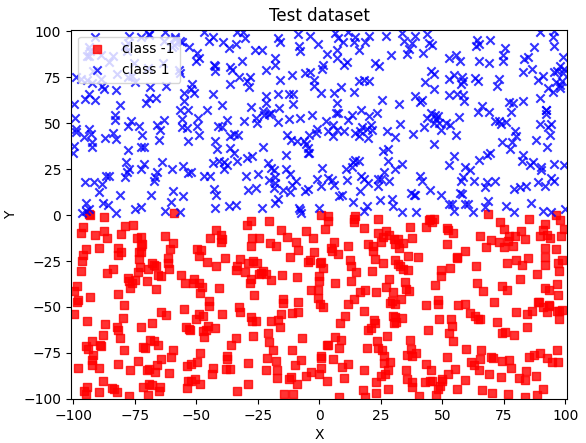
* Classification report:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Precision | Recall | F1-score | Support |
| -1 | 0.99 | 0.99 | 0.99 | 498 |
| 1 | 0.99 | 0.99 | 0.99 | 502 |
|  |  |  |  |  |
| accuracy |  |  | 0.99 | 1000 |
| macro avg | 0.99 | 0.99 | 0.99 | 1000 |
| weighted avg | 0.99 | 0.99 | 0.99 | 1000 |

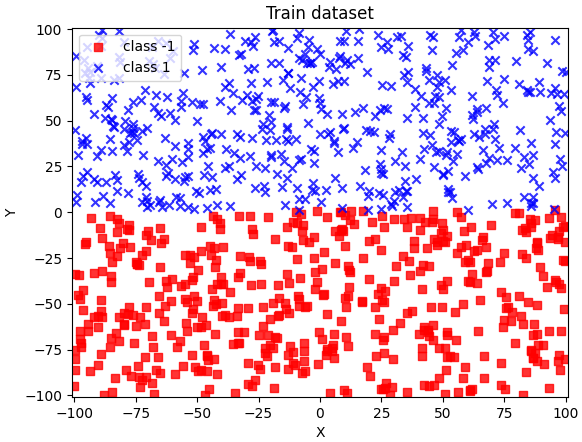
* Heatmap:



* Accuracy score: 99.2%
* Test illustration:



* Train illustration:



* Discussions:

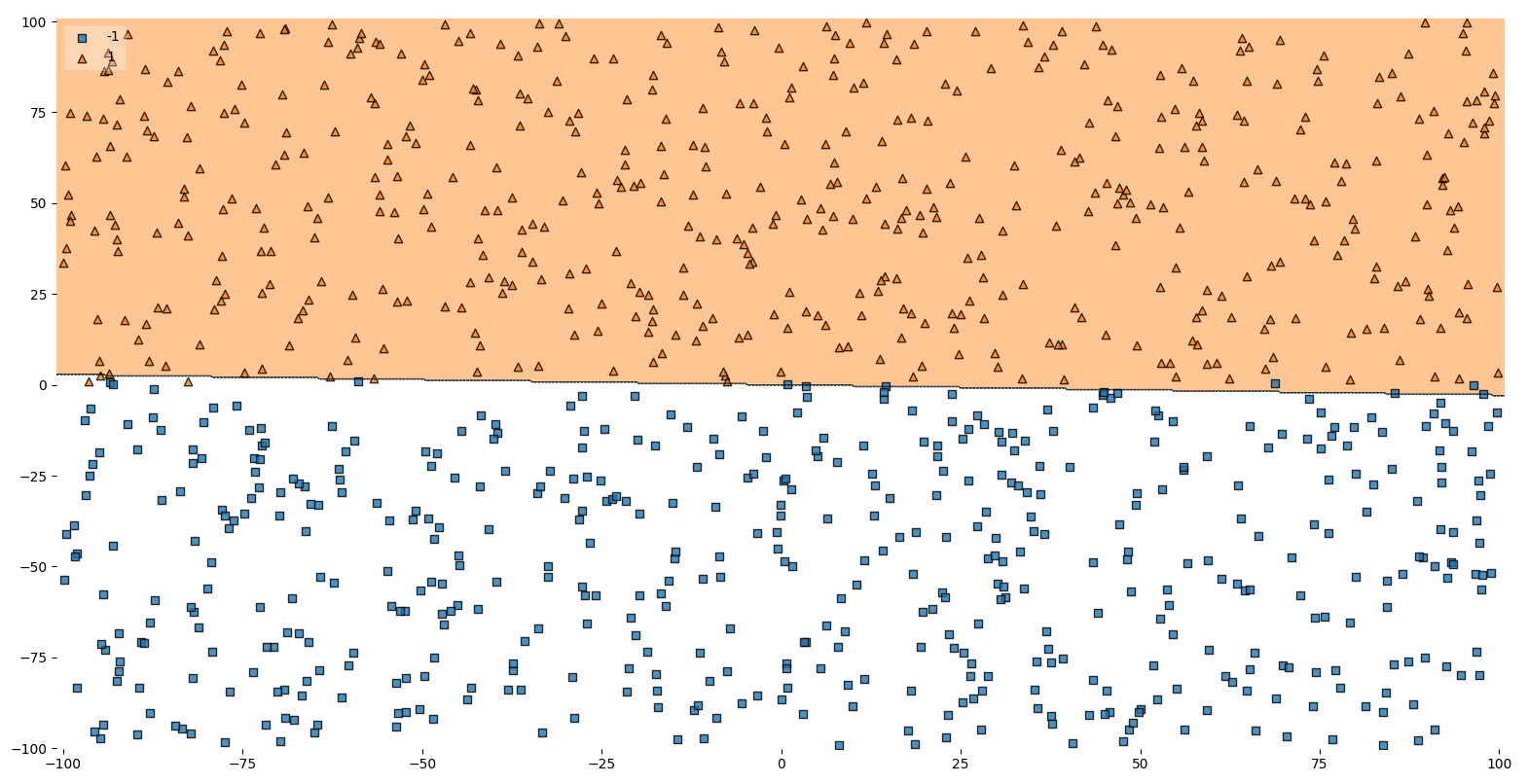
What can you conclude about your results?

*According to our work, the algorithm is capable of finding a line separating two classes. We will see that it cannot in part B.*

Does the accuracy of the result depend on the training set?

*Yes,*

* ~50 (1) / ~50(-1) and a train set of 1000 data points, accuracy score: 99.2%



* 10 (1) / 90 (-1) and a train set of 1000 data points, accuracy score: 98.0%Scatter chart

  Description automatically generated
* ~50 (1) / ~50(-1) and a train set of 500 data points, accuracy score: 97.4%

Chart, scatter chart

Description automatically generated

*Training sample size has a very strong influence on accuracy score. A series of experiments were conducted both in terms of the amount of data and in terms of the percentage of classes, and every time the result was different.*

*How well we train our algorithm depends on the size of the training sample. Since the size of the test sample helps us check the accuracy of our algorithm, we aren't too concerned with the size. As a result, we are satisfied that it includes all possible options, which can be less or more than the size of the training sample.*

* Code:

import random

import pandas as pd

import numpy as np

import seaborn as sns

import matplotlib.pyplot as plt

from matplotlib.colors import ListedColormap

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)

        if (n/100 > 1):

            one\_samples += 1

            data.append([m/100, n/100, 1])

        else:

            zero\_samples += 1

            data.append([m/100, n/100, -1])

    return data

def datasetIllustration(X, y, 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))

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\_\_":

    # 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'])

    # 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()

    figure\_two = plt.figure(2)

    datasetIllustration(X\_test, y\_test)

    plt.title('Test dataset')

    plt.xlabel('X')

    plt.ylabel('Y')

    plt.legend(loc='upper left')

    figure\_two.show()

    input()

    # start algorithm

    aln\_clf = ADAptiveLInearNEuron(n\_iter=3)

    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\_three = plt.figure(3)

    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\_three.show()

    input()

    figure\_four = plt.figure(4)

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

    figure\_four.show()

    input()

Part B:

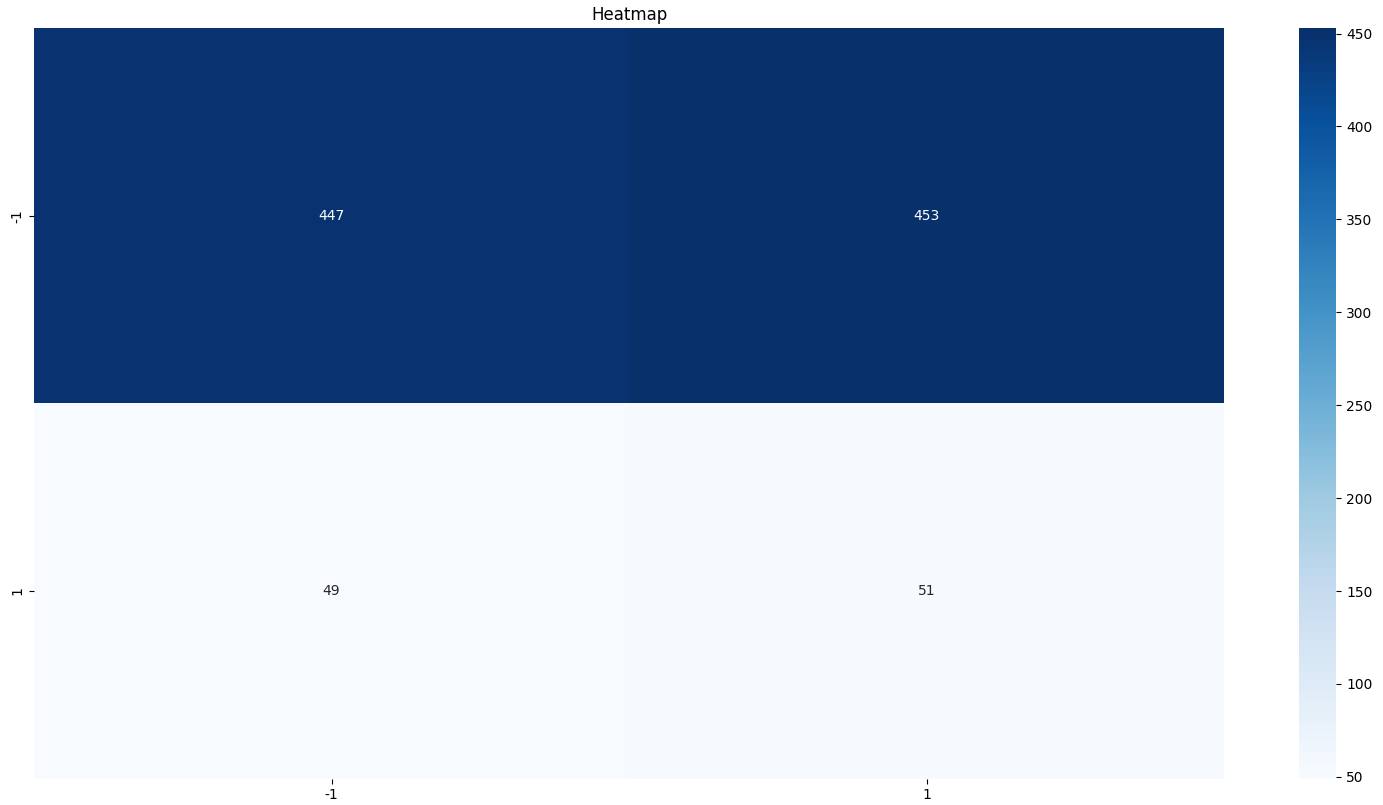
* Dataset:

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

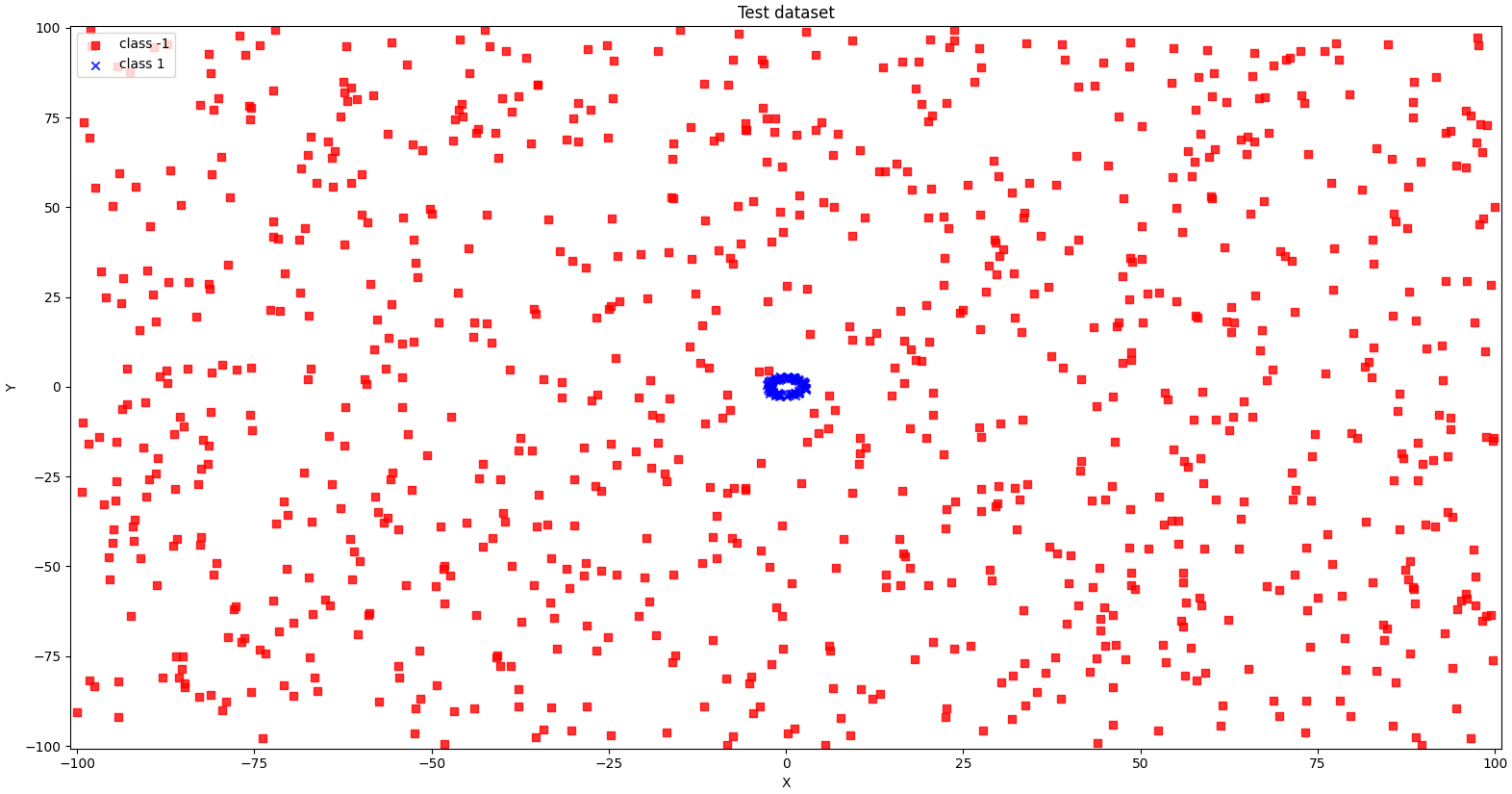
* Classification report:

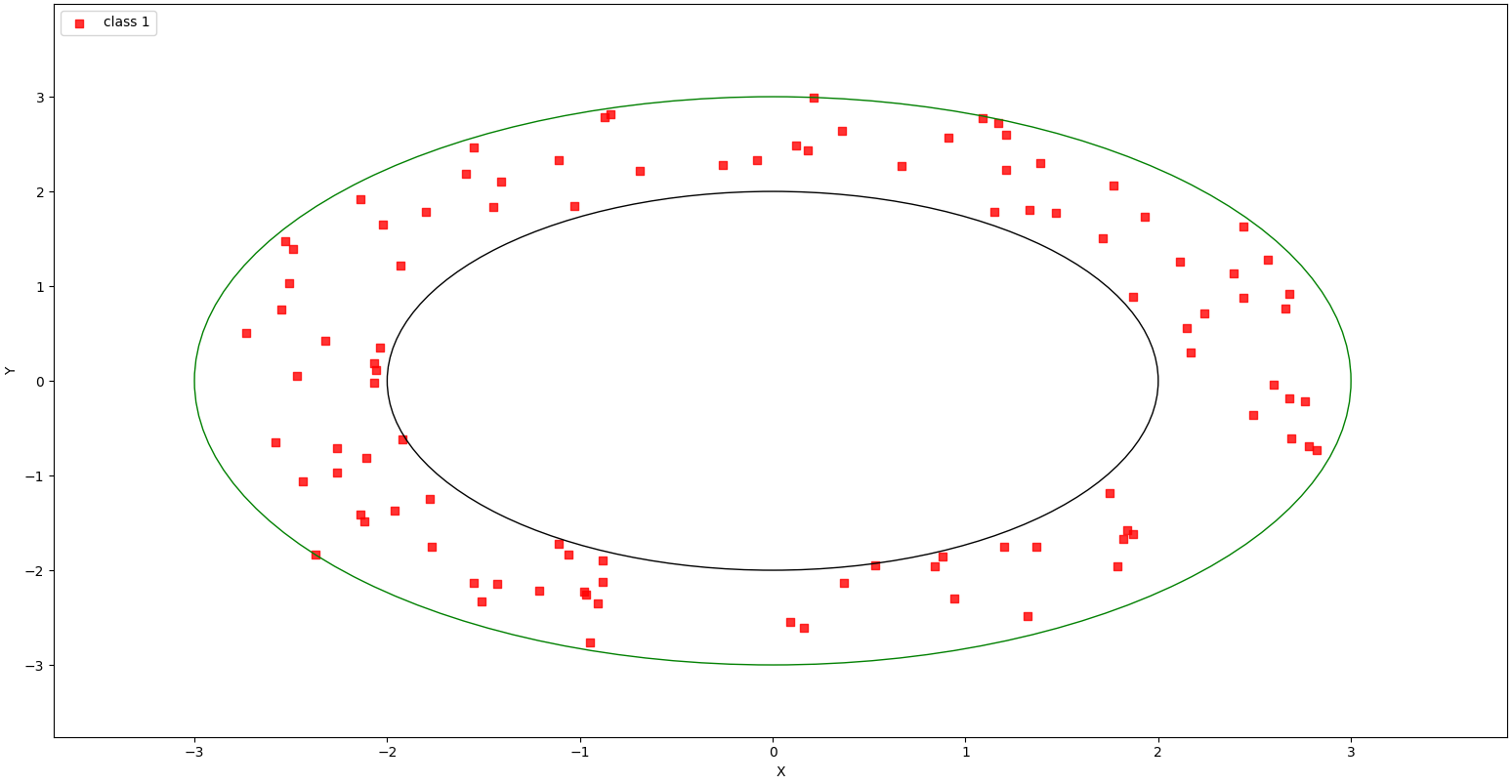
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Precision | Recall | F1-score | Support |
| -1 | 0.90 | 0.50 | 0.64 | 900 |
| 1 | 0.10 | 0.51 | 0.17 | 100 |
|  |  |  |  |  |
| accuracy |  |  | 0.50 | 1000 |
| macro avg | 0.50 | 0.50 | 0.40 | 1000 |
| weighted avg | 0.82 | 0.50 | 0.59 | 1000 |

* Heatmap:

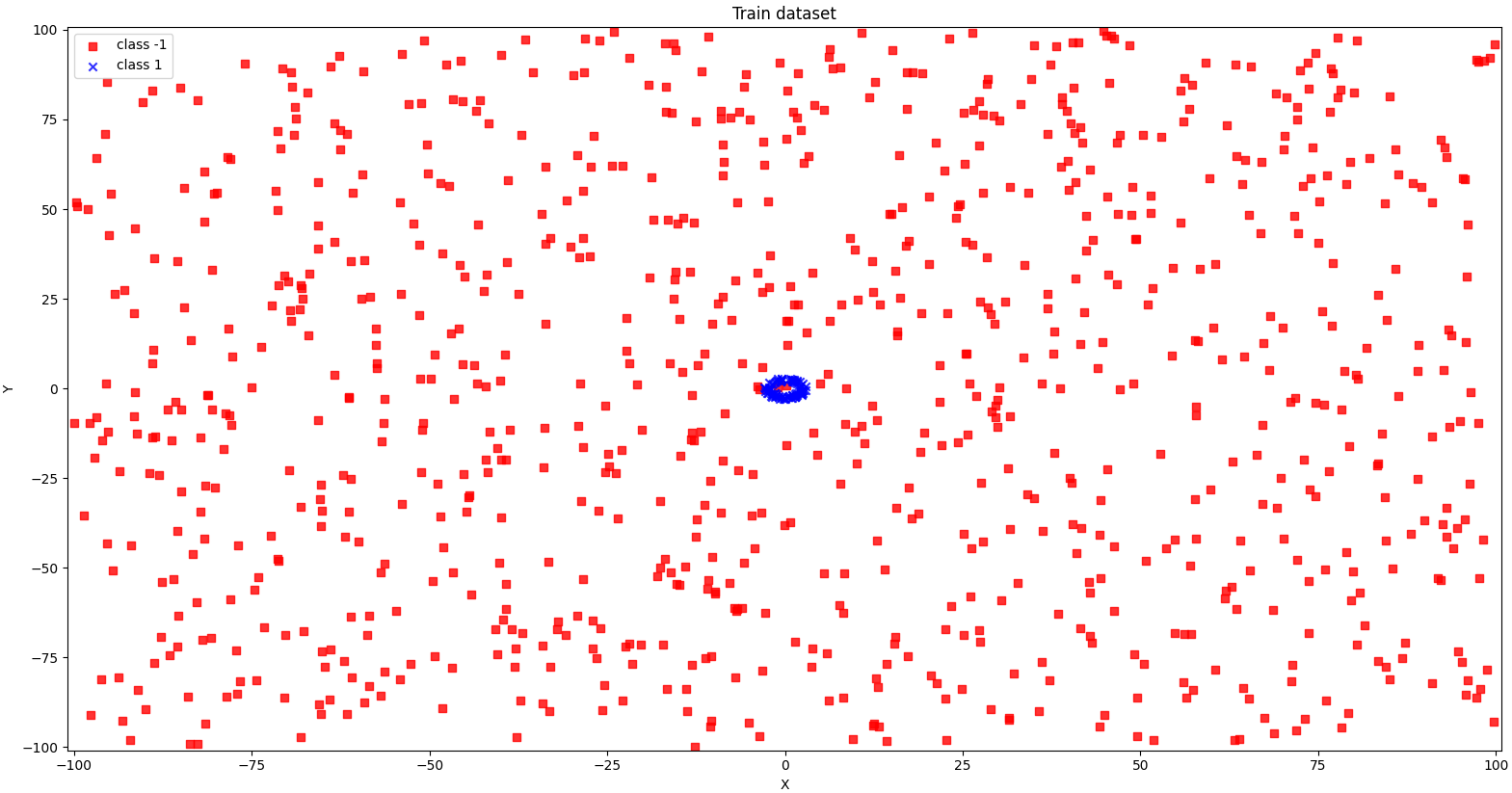


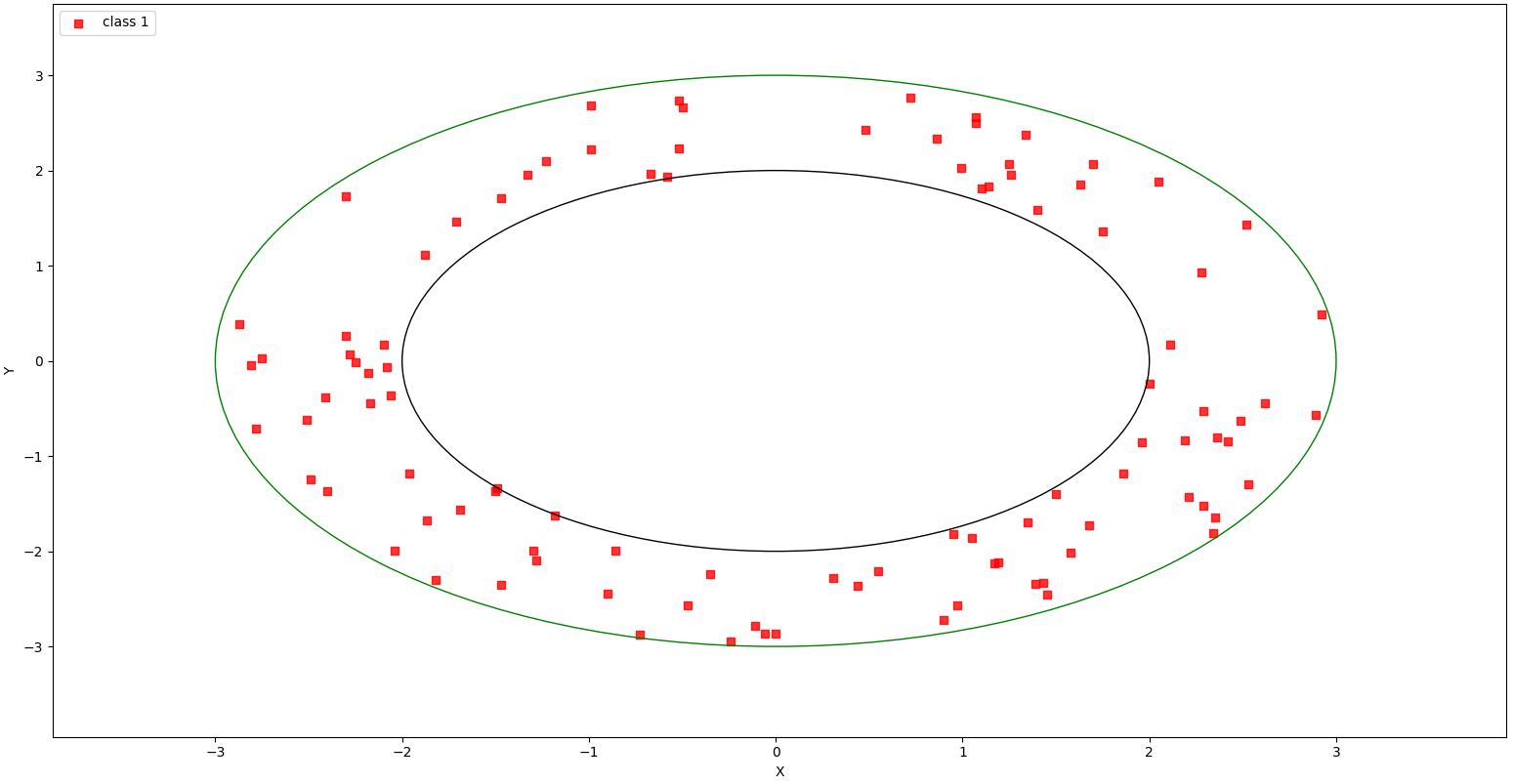
* Accuracy score: 49.8%
* Test illustration:





* Train illustration:

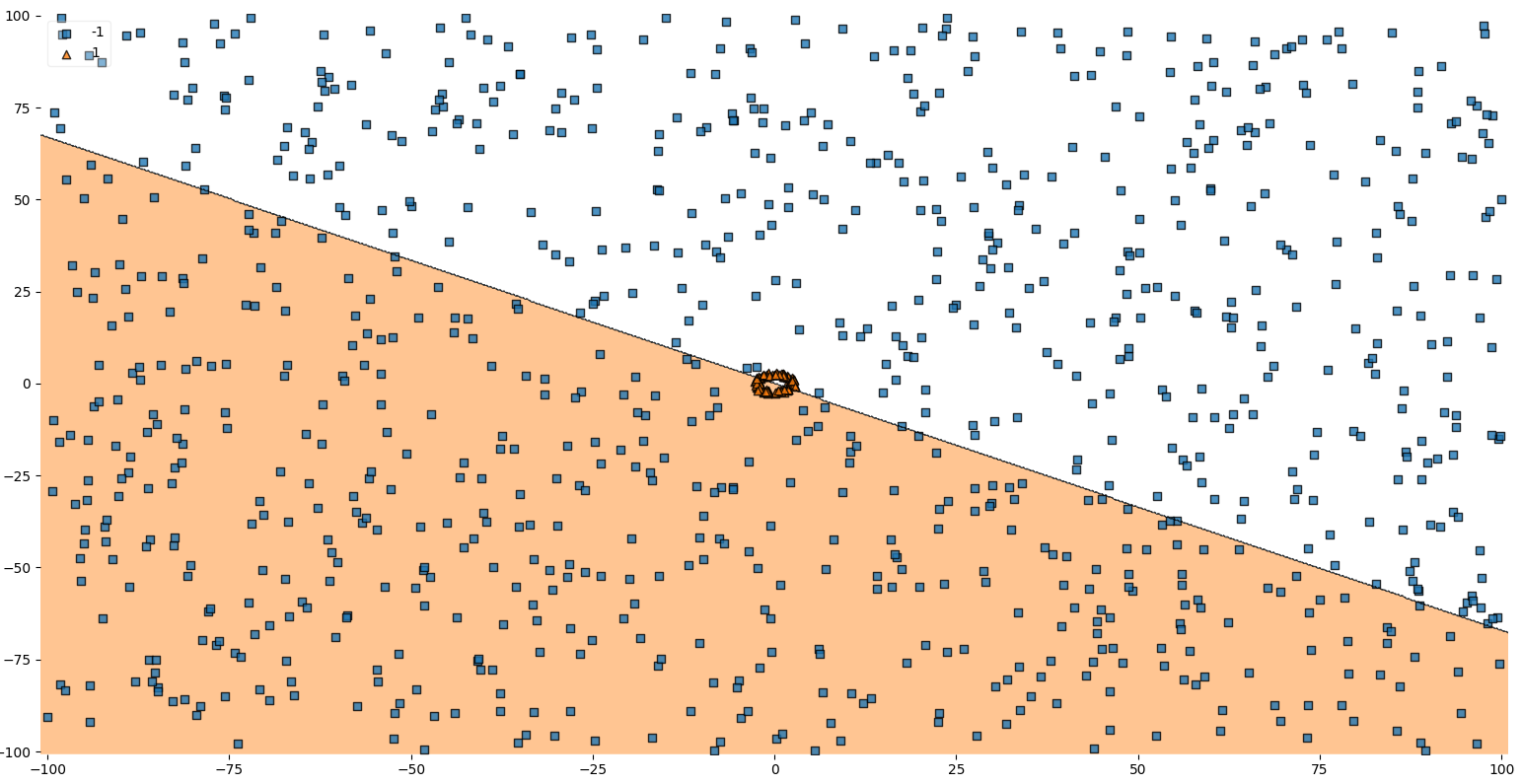




* Discussions:

What are the best results you obtain using an Adaline?

*The best result was 49.8%.*



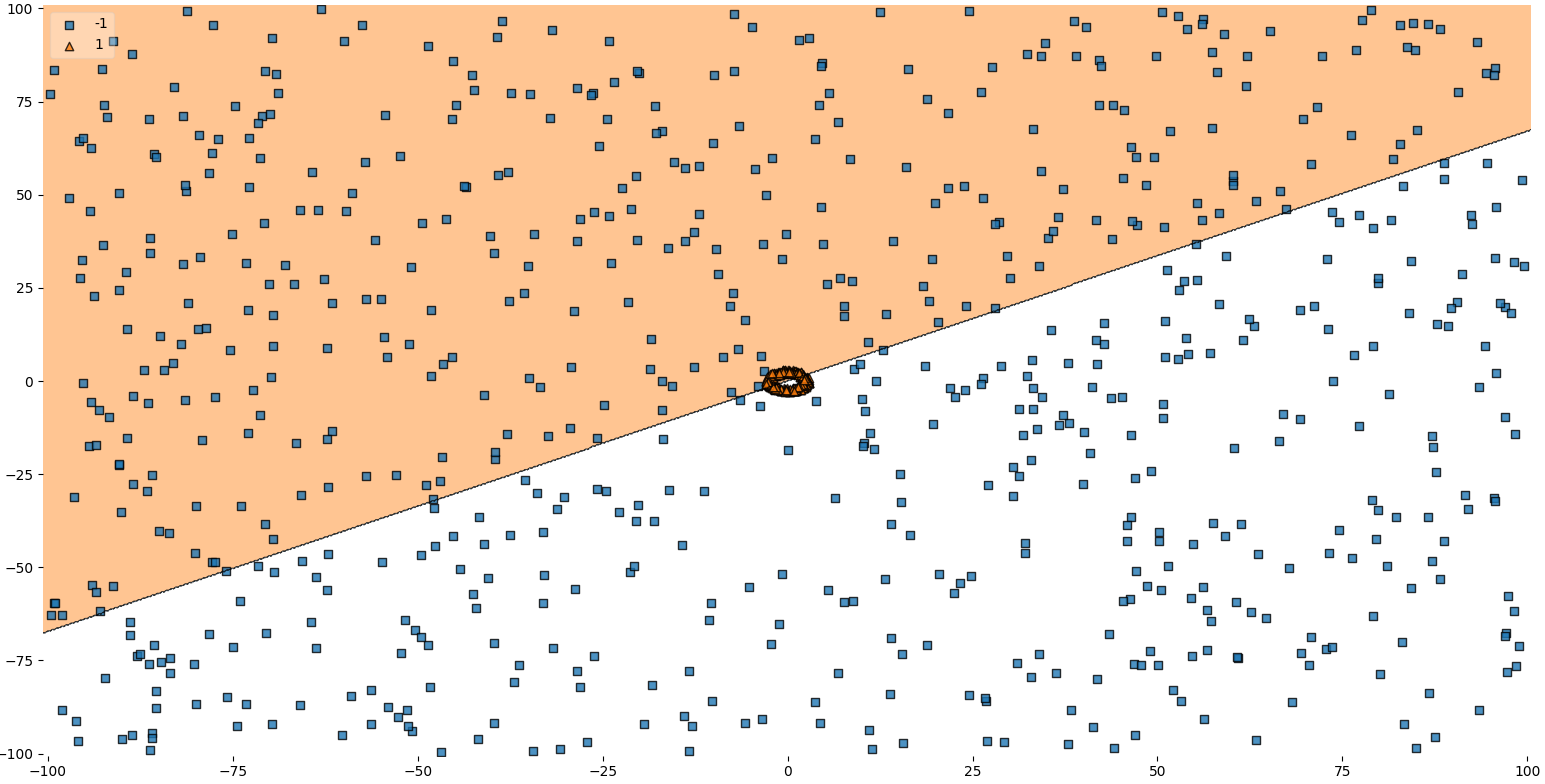
Does the quality of the results change if you use more data?

*According to* [*the Vapnik-Chervonenkis dimension*](https://en.wikipedia.org/wiki/Vapnik%E2%80%93Chervonenkis_dimension)*, no. As an example, in Part A we considered the problem of dividing points on a plane into two classes by a straight line - this is known as a linear classifier. If you have three points that are not on a straight line, then you can divide them into two classes in all possible ways by a straight line, but there is no way to decompose a group of over four points.*

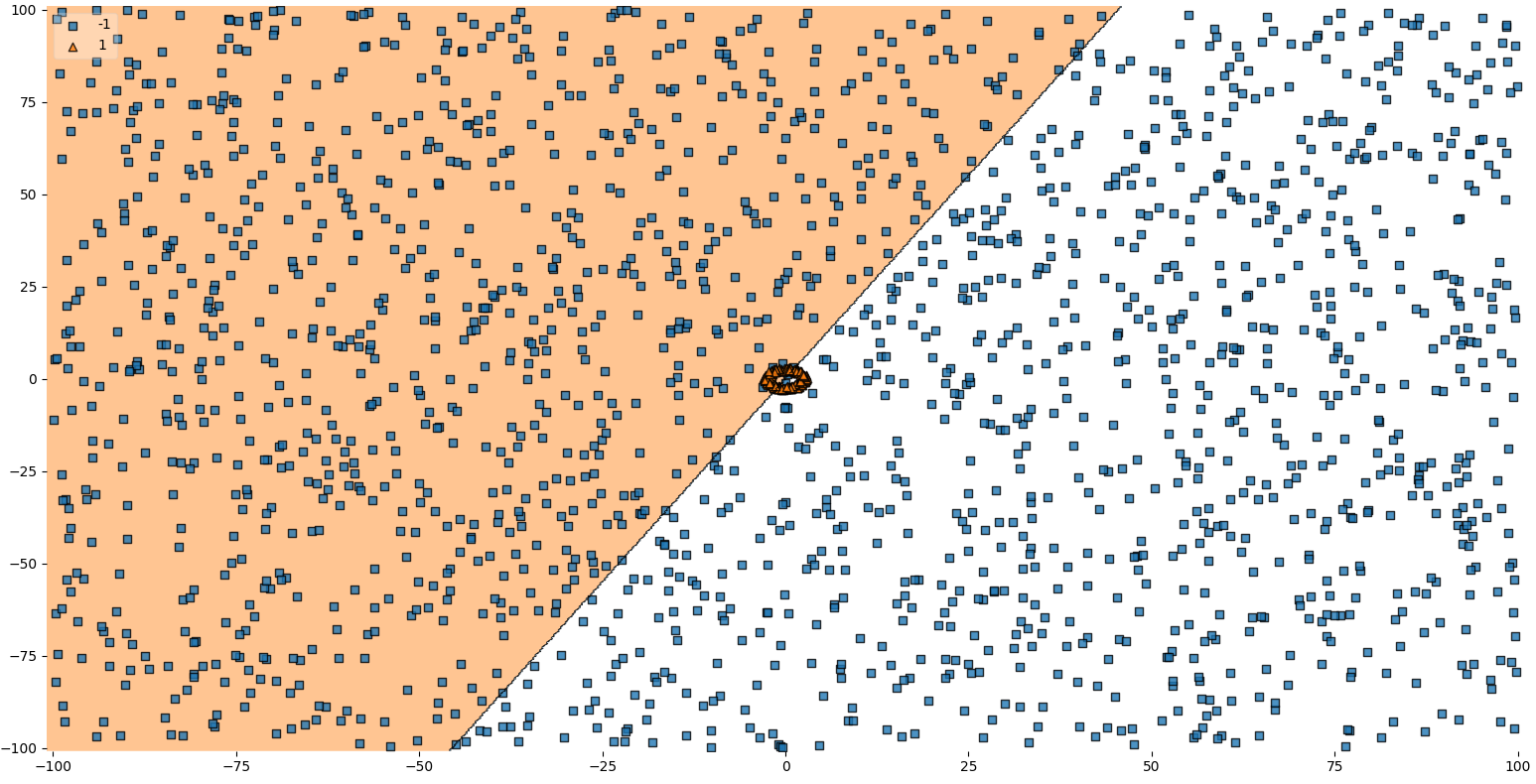
*Diagram

Description automatically generated*

* 300 (1) / 700(-1), accuracy score: 49.7%



* 200 (1) / 1800(-1), accuracy score: 50.9%



* Code:

import random

import pandas as pd

import numpy as np

import seaborn as sns

import matplotlib.pyplot as plt

from matplotlib.colors import ListedColormap

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)

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\_\_":

    # 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()

    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()

    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()

    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()

    # 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()

    figure\_six = plt.figure(6)

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

    figure\_six.show()

    input()