Project 1: part A and B

first name and T.Z. numbers

second name and T.Z. numbers

Part A:

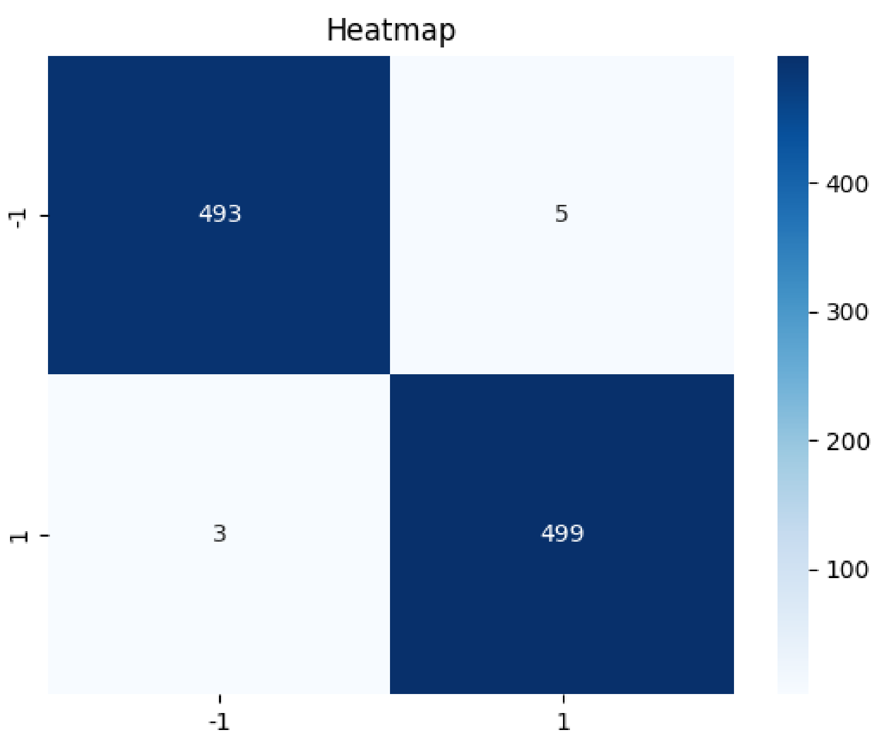
- Dataset:

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

- Classification report:

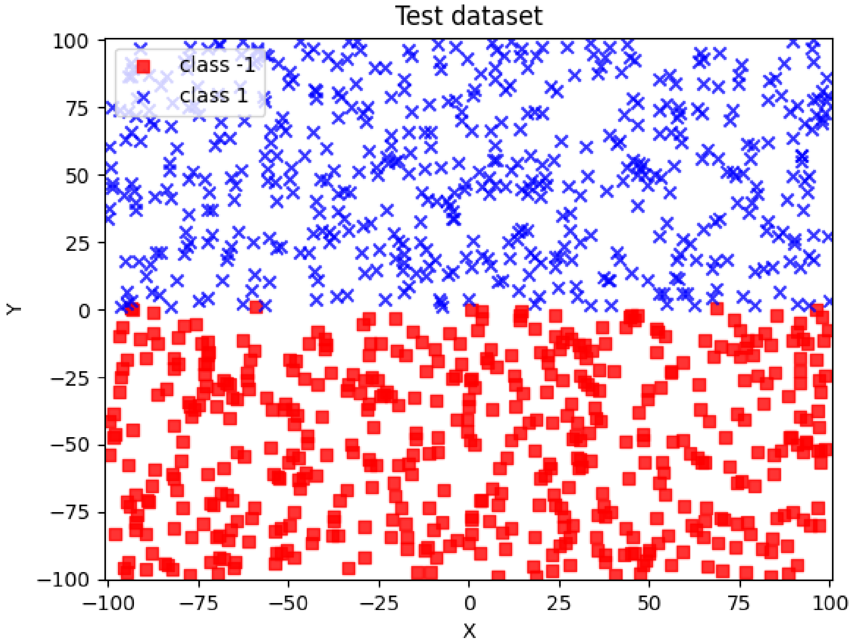
	Precision	Recall	F1-score	Support
-1	0.99	0.99	0.99	498
1	0.99	0.99	0.99	502
accuracy			0.99	1000
macro avg	0.99	0.99	0.99	1000
weighted avg	0.99	0.99	0.99	1000

- Heatmap:

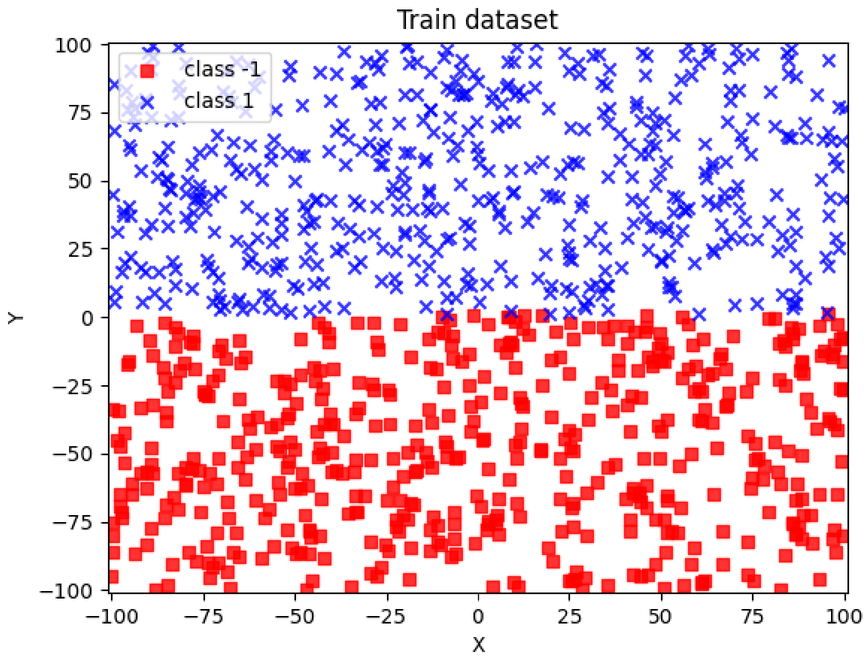


- Accuracy score: 99.2%

- Test illustration:



- Train illustration:



- Discussions:

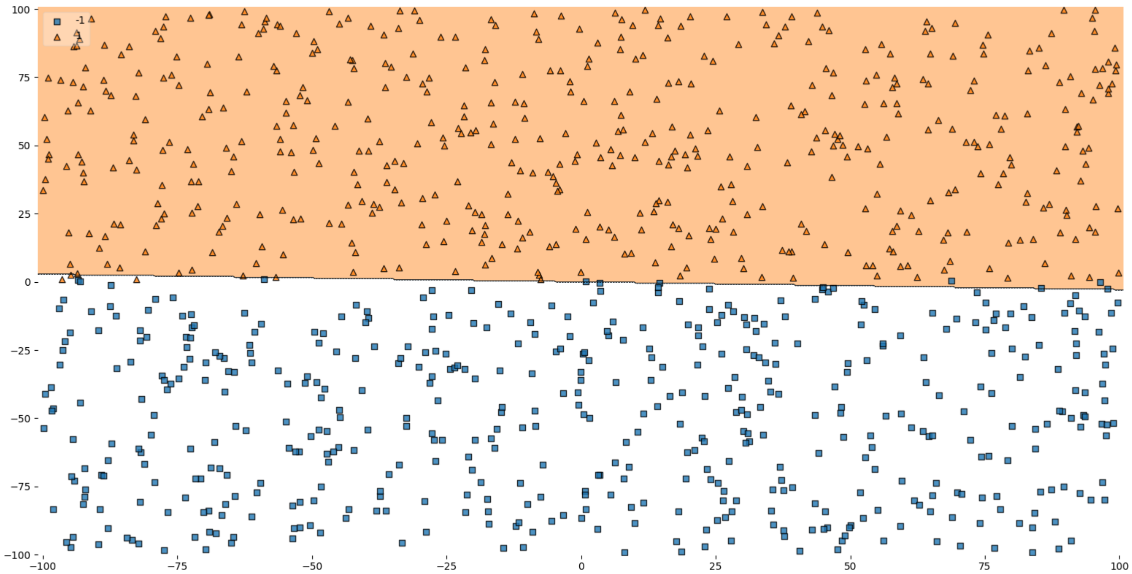
What can you conclude about your results?

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

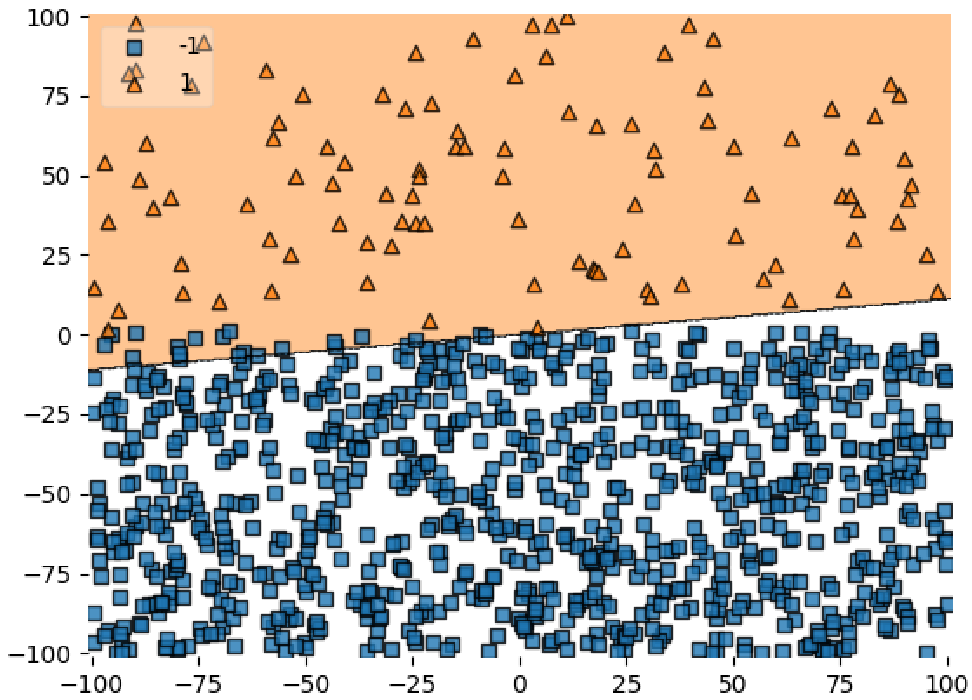
Does the accuracy of the result depend on the training set?

Yes,

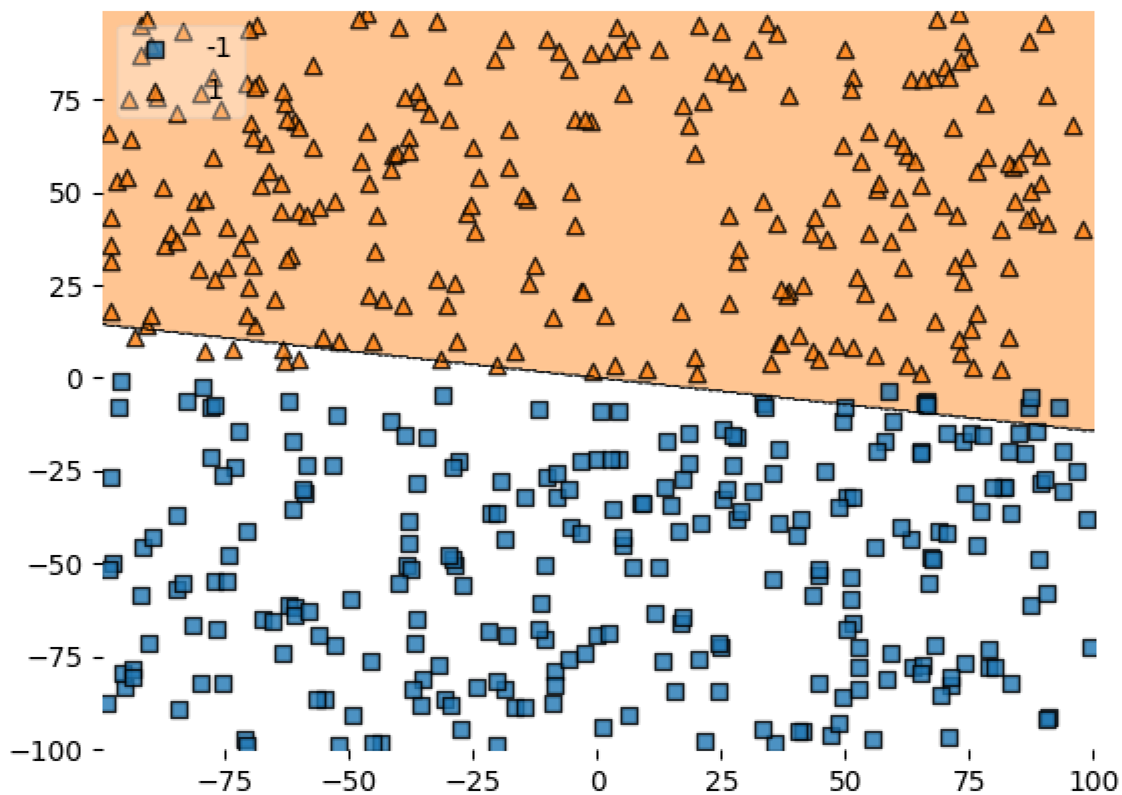
- ~50 (1) / ~50(-1) and a train set of 1000 data points, accuracy score: 99.2%



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



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



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)
```

```python
    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("Enter any char to continue: ")

    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("Enter any char to continue: ")

    # 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("Enter any char to continue: ")
```

```
figure_four = plt.figure(4)
fig = plot_decision_regions(X=X_test, y=y_test, clf=aln_clf, legend=2)
figure_four.show()
input("Enter any char to finish: ")
```

Part B:

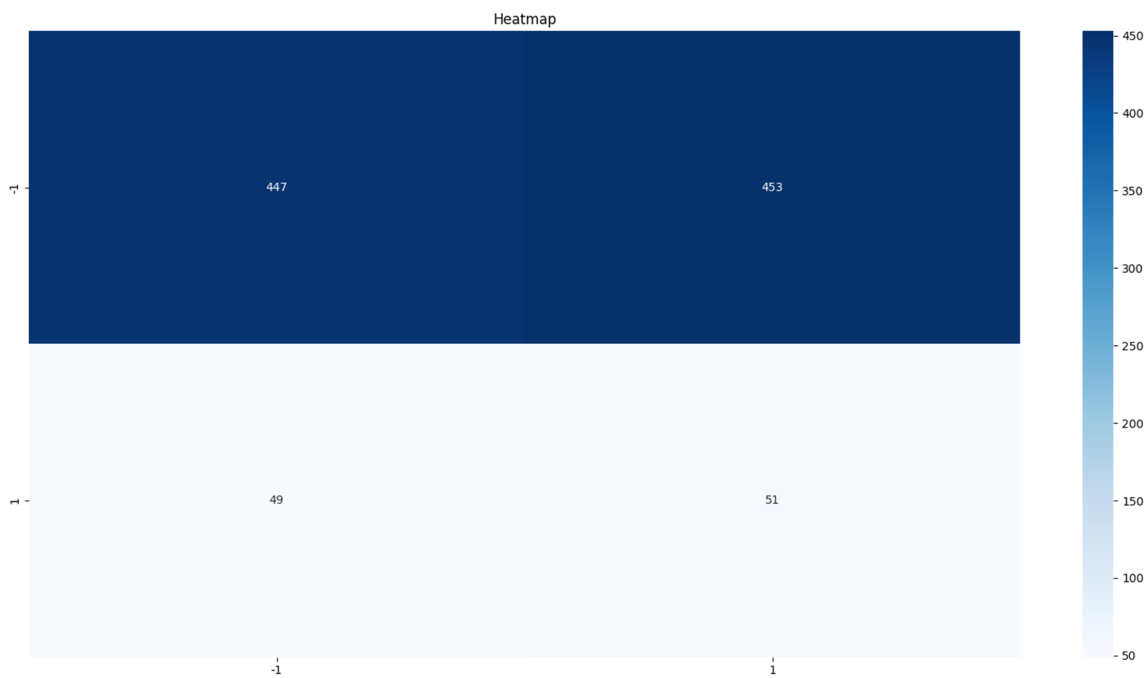
- Dataset:

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

- Classification report:

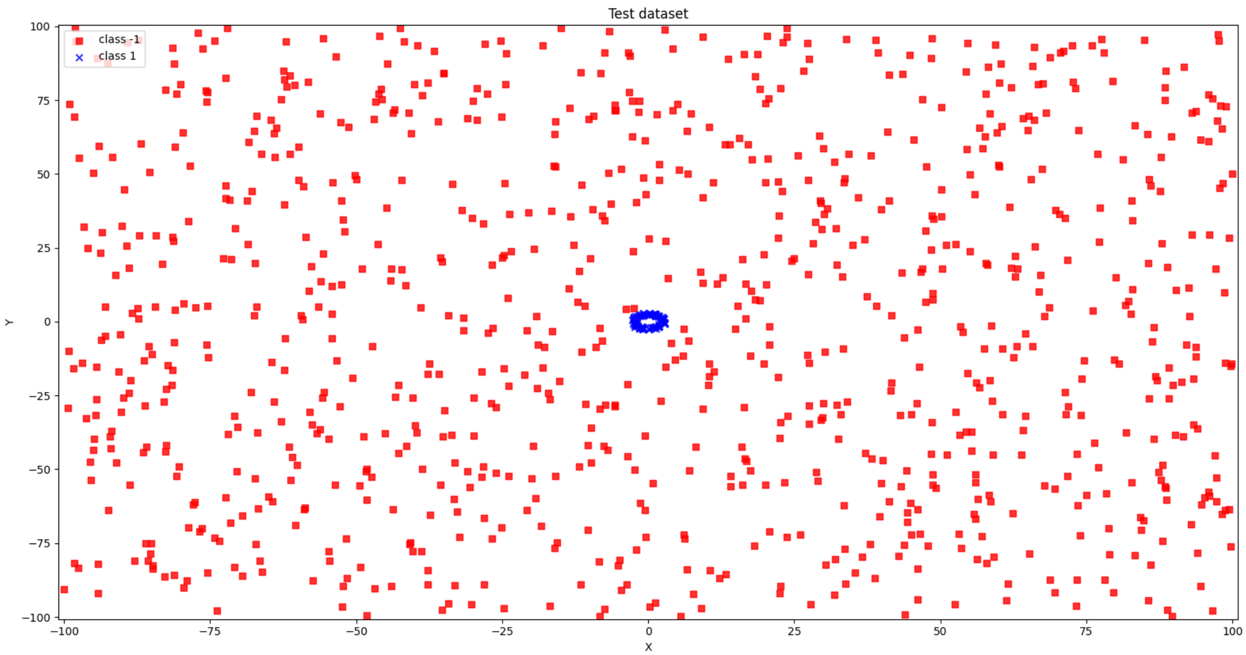
	Precision	Recall	F1-score	Support
-1	0.90	0.50	0.64	900
1	0.10	0.51	0.17	100
accuracy			0.50	1000
macro avg	0.50	0.50	0.40	1000
weighted avg	0.82	0.50	0.59	1000

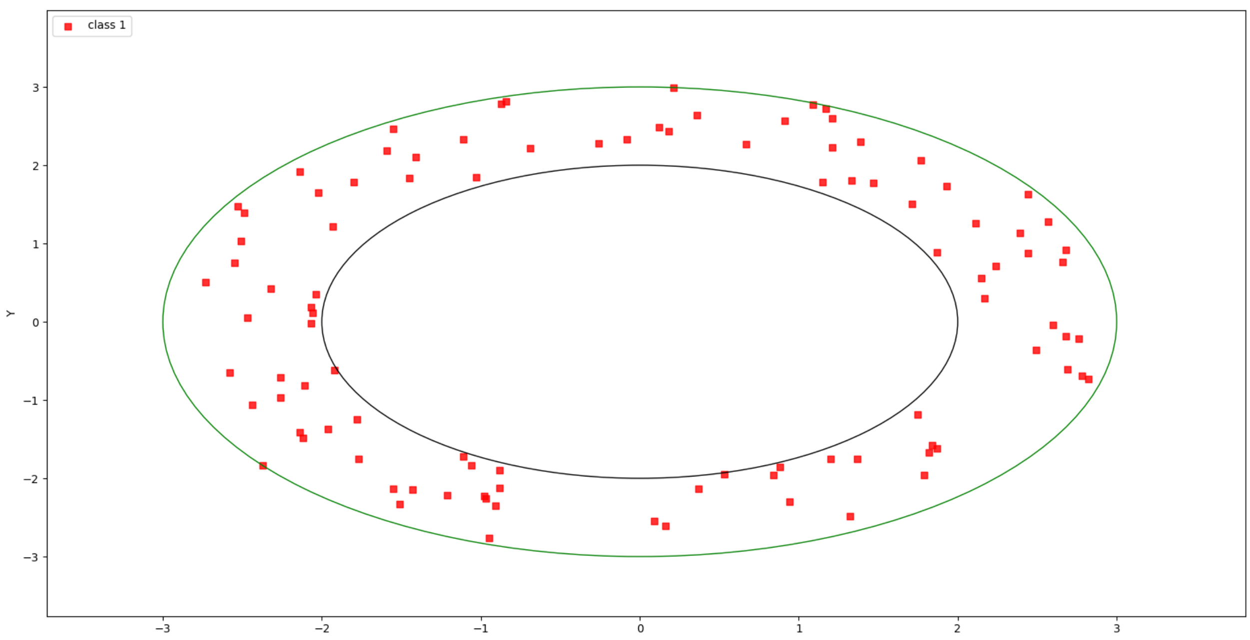
- Heatmap:



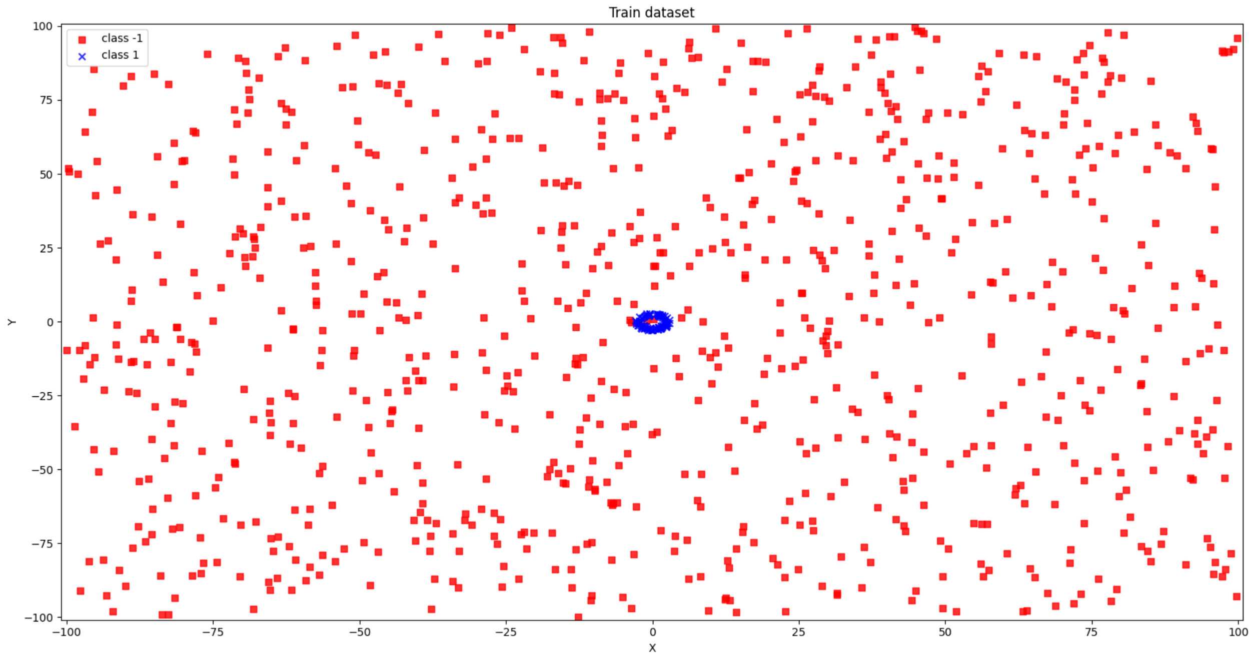
- Accuracy score: 49.8%

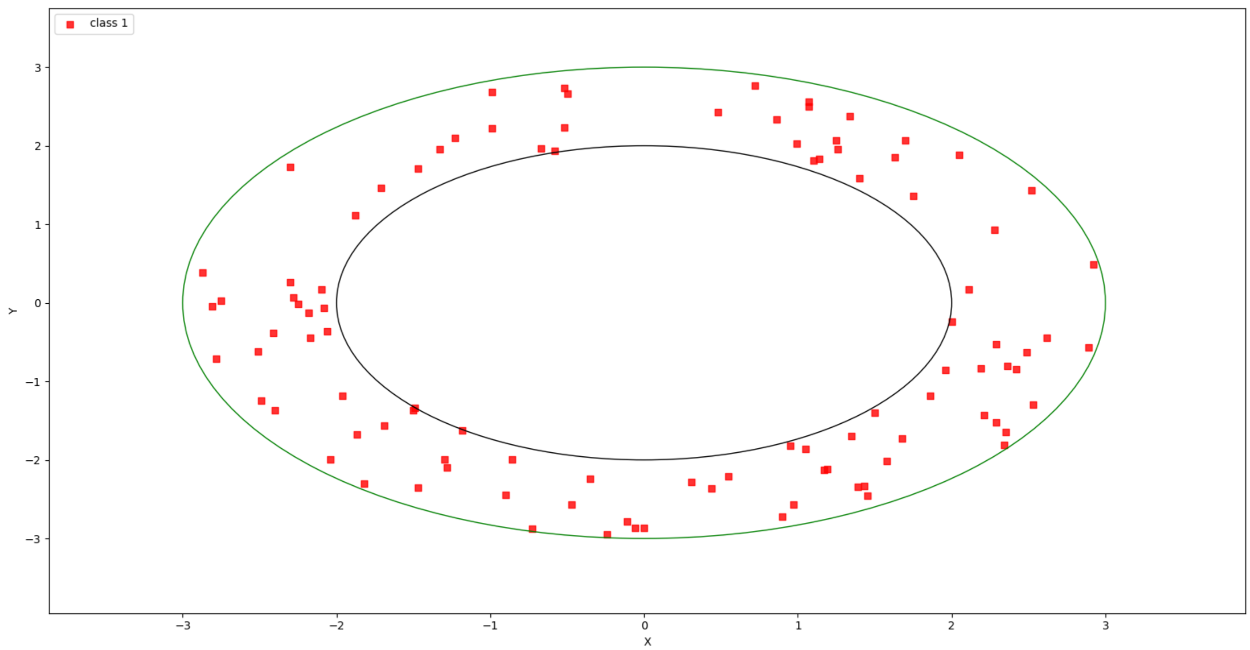
- Test illustration:





- Train illustration:

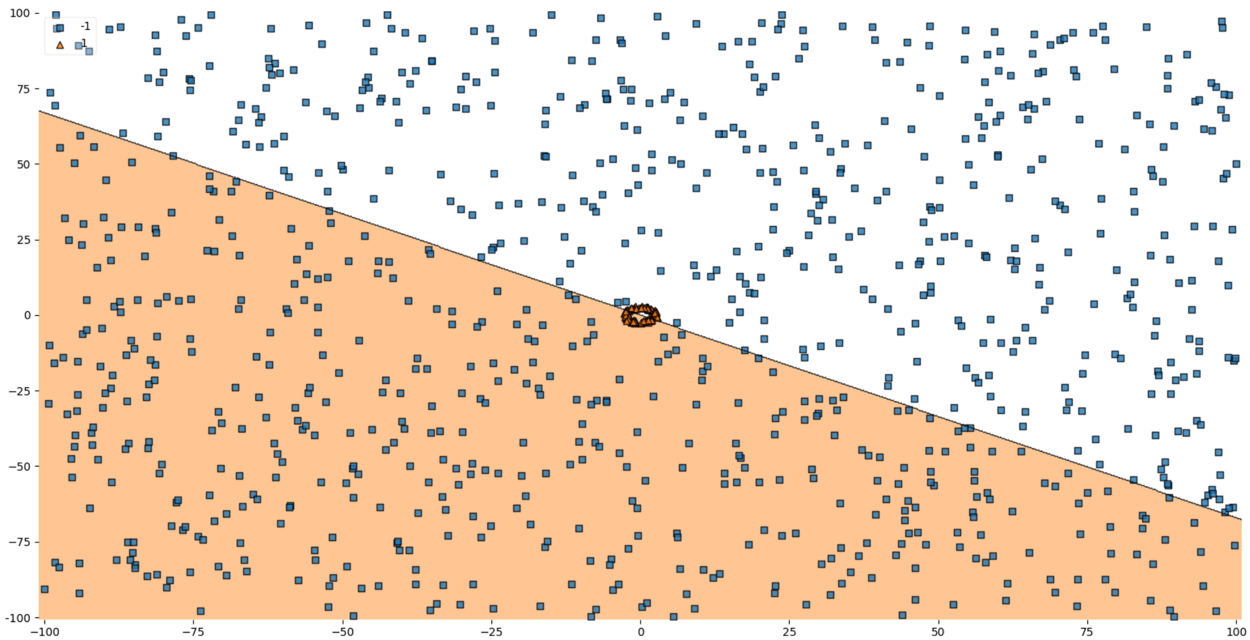




- Discussions:

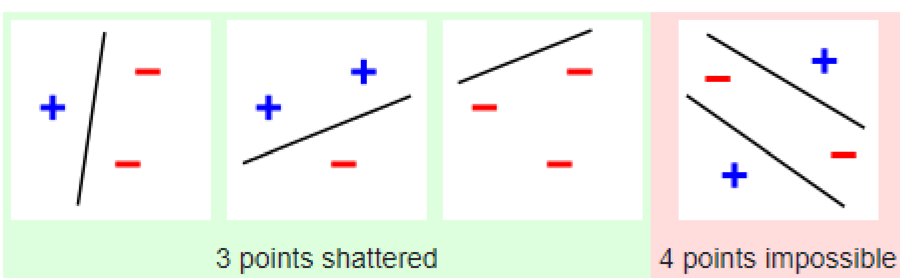
What are the best results you obtain using an Adaline?

The best result was 49.8%.

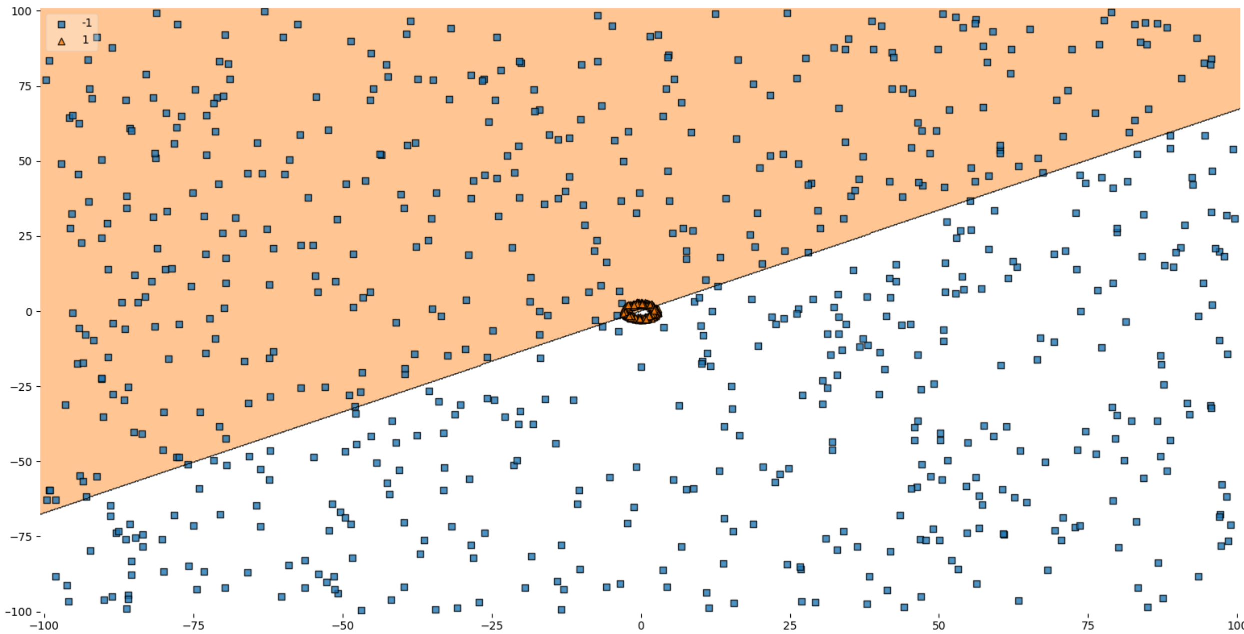


Does the quality of the results change if you use more data?

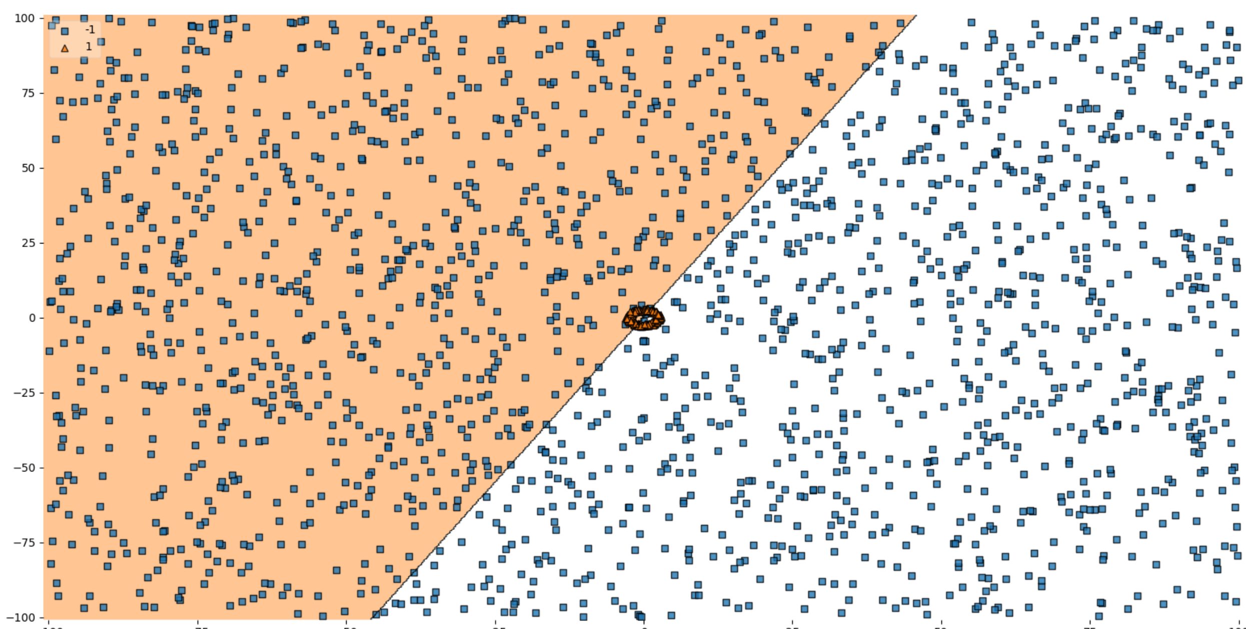
According to [the Vapnik-Chervonenkis dimension](), no. As an example, in Part A we considered the problem of dividing points on a plane into two classes by a straight line - this is known as a linear classifier. If you have three points that are not on a straight line, then you can divide them into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



- 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("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: ")


    # 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: ")
```