Parts A and B

Collaborators:

Talia Seada – 211551601, assigned to the morning group.

Yehudit Brickner – 328601018, assigned to the morning group.

Tavor Levine – 315208439, assigned to the evening group.

Noa Nussbaum – 206664278, assigned to the morning group.

Table of contents:

Part A

Tables

First test set results table:

Accuracy score: 99.1%

Confusion matrix:

True Positive	False Positive	True Negative	False Negative
50.60%	0.40%	48.50%	0.50%

Precision: 99%
Recall: 99%
F1-score: 99%

Second test set results table:

Accuracy score: 98.3%

Confusion matrix:

True Positive	False Positive	True Negative	False Negative
47.80%	0.80%	50.50%	0.90%

Precision: 98%
Recall: 98%
F1-score: 98%

Train A:

	x	y	value
0	-55.98	86.51	1
1	-79.33	-16.42	-1
2	-61.37	62.34	1
3	47.28	54.74	1
4	24.39	-31.21	-1
5	-69.25	59.86	1
6	-90.72	27.73	1
7	41.8	99.04	1
8	-99.31	45.94	1
9	-12.73	-25.04	-1
10	93.7	-66.51	-1

First Test A:

	x	y	value
0	51.71	22.32	1
1	-12.47	-54.61	-1
2	-39.01	-97.9	-1
3	10.86	64.75	1
4	51.95	98.14	1
5	-73.52	9.45	1
6	81.62	-86.6	-1
7	24.19	-44.49	-1
8	48.11	38.47	1
9	-48.51	-44.81	-1
10	-22.01	-83.18	-1

Second Test A:

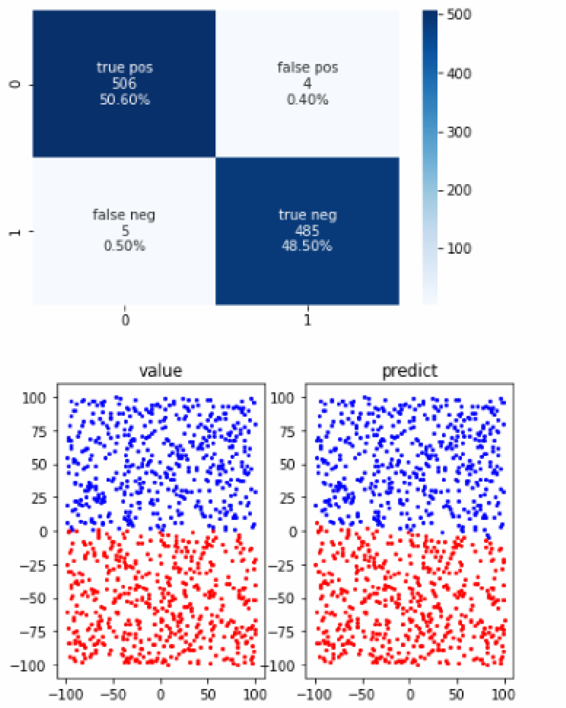
	x	y	value
0	-25.72	21.37	1
1	23.0	-58.61	-1
2	-36.72	-85.66	-1
3	-72.09	-55.16	-1
4	-18.92	65.92	1
5	-31.39	31.3	1
6	-90.08	50.44	1
7	59.71	48.48	1
8	27.95	62.19	1
9	87.8	-37.01	-1
10	31.95	-70.66	-1

Discussions and Illustrations

We tried fitting the Adaline with a bigger training set, it took the same number of iterations of the data to get the MSE below the threshold, but we noticed that the accuracy went down because we were over fitting the network, you can also see that in the picture that the line is not as straight.

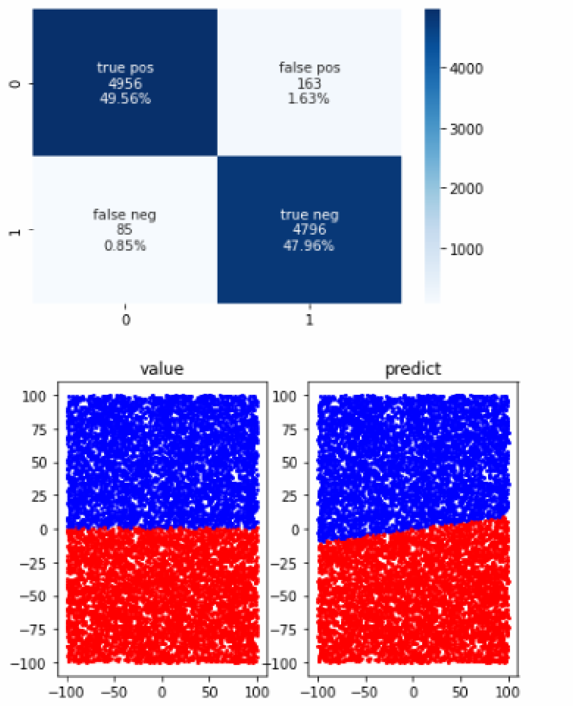
using 1,000 points

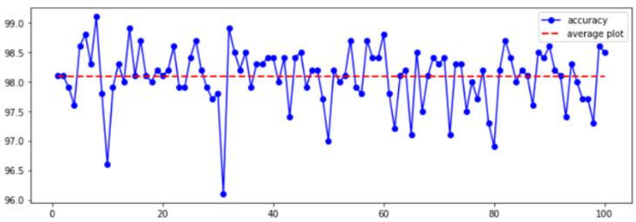
```
First Test:
Acurracy score: 99.1
              precision    recall  f1-score   support

          -1       0.99      0.99      0.99       510
           1       0.99      0.99      0.99       490

    accuracy                           0.99      1000
   macro avg       0.99      0.99      0.99      1000
weighted avg       0.99      0.99      0.99      1000
```

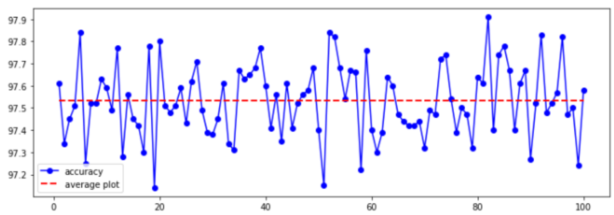


using 10,000 points

```
First Test:
Acurracy score: 97.52
              precision    recall  f1-score   support

          -1       0.98      0.97      0.98      5119
           1       0.97      0.98      0.97      4881

    accuracy                           0.98     10000
   macro avg       0.98      0.98      0.98     10000
weighted avg       0.98      0.98      0.98     10000
```





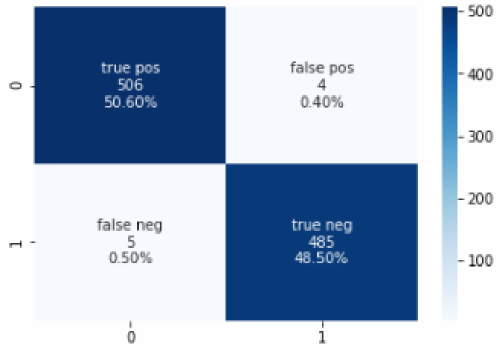


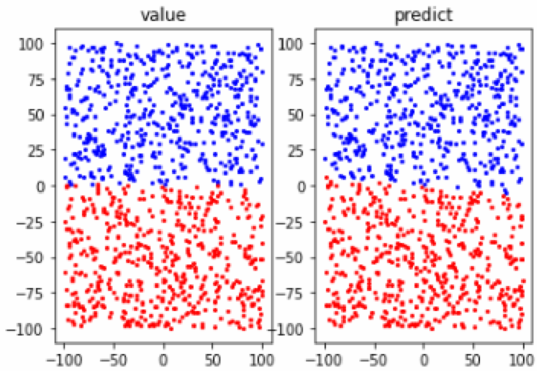
We ran the Adaline algorithm 100 times for both size of data.
You can see in the graphs above that the average accuracy is higher with a smaller training set and testing set.

Additionally, we tried training the Adaline on a different training set and got similar results.

Using the train as a train and test2 for the test

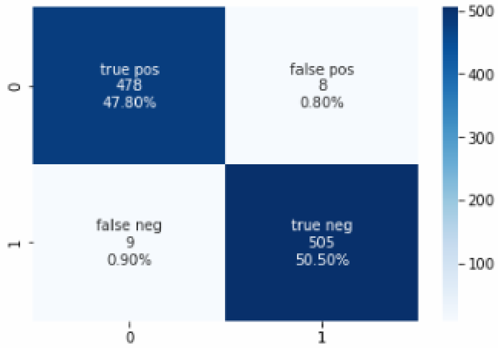
```
First Test:
Acurracy score: 99.1
              precision    recall  f1-score   support

          -1       0.99      0.99      0.99       510
           1       0.99      0.99      0.99       490

    accuracy                           0.99      1000
   macro avg       0.99      0.99      0.99      1000
weighted avg       0.99      0.99      0.99      1000
```

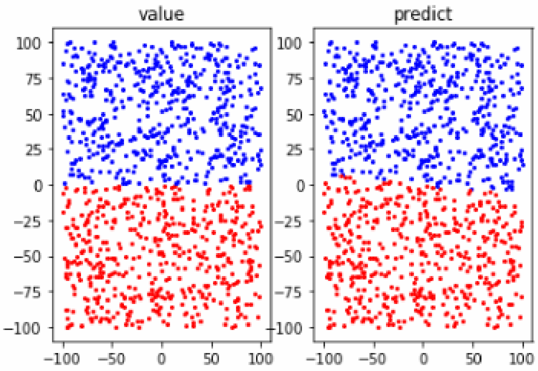




Using test1 as a train and test2 for the test

```
Second Test:
Acurracy score: 98.3
              precision    recall  f1-score   support

          -1       0.98      0.98      0.98       486
           1       0.98      0.98      0.98       514

    accuracy                           0.98      1000
   macro avg       0.98      0.98      0.98      1000
weighted avg       0.98      0.98      0.98      1000
```





Code

Parts A and B:

In [1]:

```
#libraries
import numpy as np
import pandas as pd

import random

#preproccesing
from sklearn.metrics import classification_report,f1_score
from sklearn.metrics import confusion_matrix

# visulization
import matplotlib.pyplot as plt
import seaborn as sns

import warnings
warnings.filterwarnings('ignore')
```

In [2]:

```
class Adaline:
    def __init__(self, learning_rate, train):
        self.learning_rate = learning_rate
        self.train = train

    # this function generates random small weights and bias for the Adaline algorithm
    def _weight_genarate(self):
        weight = [] # [x,y]
        for i in range(2):
            random.seed(i)
            rand = random.uniform(0, 0.01)
            rand = round(rand, 4)
            weight.append(rand)

        # now generate the bias
        random.seed(4)
        bias = random.uniform(0, 1)
        bias = round(bias, 4)
        return weight, bias

    # this function fits the adaline model on the training data
    def fit(self):
        ERR = []
        mse = []
        EPS = 0.001
        # generate weights and bias
        weight, bias = self._weight_genarate()
        oldmse=1
        while(True):
            ERR = []
            # for each row we fix the bias and wights in order to get the minimum error
            for index, row in self.train.iterrows():
                predicted = bias + row['x']/100 * weight[0] + row['y']/100 * weight[1]

                weight[0] = round((weight[0] + self.learning_rate * (row['value'] - predicted) * row['x']/100), 3)
                weight[1] = round((weight[1] + self.learning_rate * (row['value'] - predicted) * row['y']/100), 3)
                bias = round((bias + self.learning_rate * (row['value'] - predicted)), 3)

                # error calculation
                error = (row['value'] - predicted) ** 2
                # if the error is small enough return
                ERR.append(error)

            mse.append(np.sum(ERR))
            if len(mse) >= 2:
                # checking if the error is smaller then eps or if it hasnt changed
                if abs(mse[-1] - mse[-2]) < EPS or abs(mse[-1] - mse[-2])==oldmse :
                    break
            # updating the old mse
            if len(mse)>=2:
                oldmse=abs(mse[-1] - mse[-2])
        return weight, bias

    # this function predicts on a test data and returns the number of correct predictions
    def predict(self, test, weight, bias):
        count = 0
        pred = []
        # for each row we use the activation formula with the weights and bias we returned
        # in the fit function to predict on the test data set
        for index, row in test.iterrows():
            prediction = bias + (row['x'] * weight[0]) + (row['y'] * weight[1])
            if prediction > 0:
                prediction = 1
            else:
                prediction = -1
            pred.append(prediction)

            if prediction == row['value']:
                count += 1
        # now add the prediction list to the data set in order to make comparison
        test['predict'] = pred
        return count

    # this function caculates the acuuracy of the predictions
    def score(self, pred, test):
        acurr = pred / len(test)
        res = round(acurr, 4)
        return res
```

In [3]:
```python
# this function builds the data set for part A of the assighnment
def build_data_partA(i):
    x = []
    y = []
    value = []
    random.seed(i)
    for i in range(1000):
        # generate two random numbers between -10000 to 10000
        randX = random.randint(-10000, 10000)
        randY = random.randint(-10000, 10000)
        x.append(randX / 100)
        y.append(randY / 100)
        # for part A if y > 1 then the value is 1
        if y[i] > 1:
            value.append(1)
        # else the value is -1
        else:
            value.append(-1)

    # make the data frame
    end = {'x': x, 'y': y, 'value': value}
    df = pd.DataFrame(data=end, columns=['x', 'y', 'value'])
    return df
```

In [5]:
```python
# this function plots the values of the actual values of the data compared to the prediction values we predicted
def plotting(test):
    f, ax = plt.subplots(1, 2)
    ax[0].set_title("value")
    ax[1].set_title("predict")

    for index, row in test.iterrows():
        if row['value'] == 1:
            ax[0].plot(row['x'], row['y'], markersize=2, marker="o", color="blue")
        else:
            ax[0].plot(row['x'], row['y'], markersize=2, marker="o", color="red")
        if row['predict'] == 1:
            ax[1].plot(row['x'], row['y'], markersize=2, marker="o", color="blue")
        else:
            ax[1].plot(row['x'], row['y'], markersize=2, marker="o", color="red")
    plt.show()
```

confussion matrix:

a confusion matrix, also known as an error matrix, is a specific table layout that allows visualization of the performance of an algorithm, typically a supervised learning one (in unsupervised learning it is usually called a matching matrix). Each row of the matrix represents the instances in an actual class while each column represents the instances in a predicted class, or vice versa – both variants are found in the literature. The name stems from the fact that it makes it easy to see whether the system is confusing two classes (i.e. commonly mislabeling one as another).

wikipedia - https://en.wikipedia.org/wiki/Confusion_matrix

In [6]:
```python
# this function plots the confussion matrix
def confussion_matrix (cf_matrix):
    group_names = ['true pos', 'false pos', 'false neg', 'true neg']
    group_counts = ["{0:0.0f}".format(value) for value in cf_matrix.flatten()]
    group_percentages = ["{0:.2%}".format(value) for value in cf_matrix.flatten()/np.sum(cf_matrix)]
    labels = [f"{v1}\n{v2}\n{v3}" for v1, v2, v3 in zip(group_names,group_counts,group_percentages)]
    labels = np.asarray(labels).reshape(2,2)
    sns.heatmap(cf_matrix, annot=labels, fmt='', cmap='Blues')
```

In [7]:
```python
def part_a():
    train = build_data_partA(1)
    first_test = build_data_partA(9)
    second_test = build_data_partA(8)

#     train.to_csv(r'C:\Users\talia\NeuroComputation\NeuroComputation\train_A.csv')
#     first_test.to_csv(r'C:\Users\talia\NeuroComputation\NeuroComputation\first_test_A.csv')
#     second_test.to_csv(r'C:\Users\talia\NeuroComputation\NeuroComputation\second_test_A.csv')

    print("First Test:")
    # run Adaline algorithm
    ada = Adaline(0.1, train)
    weight, bias = ada.fit()
    ada_pred = ada.predict(first_test, weight, bias)
    ada_score = ada.score(ada_pred, first_test)
    print("Acurracy score:", ada_score * 100)

    # confusion matrix
    con_mat = confusion_matrix(first_test['value'], first_test['predict'])
    confussion_matrix(con_mat)
    print(classification_report(first_test['value'], first_test['predict']))
    plotting(first_test)

    print("Second Test:")
    # run Adaline algorithm
    ada = Adaline(0.1, train)
    weight, bias = ada.fit()
    ada_pred = ada.predict(second_test, weight, bias)
    ada_score = ada.score(ada_pred, second_test)
    print("Acurracy score:", ada_score * 100)

    # confusion matrix
    con_mat = confusion_matrix(second_test['value'], second_test['predict'])
    confussion_matrix(con_mat)
    print(classification_report(second_test['value'], second_test['predict']))
    plotting(second_test)
```

In [9]:

```python
# this is the main function
def main():
    part = input("Enter the relevant part (A or B): ")

    # part A
    if part == 'A':
        part_a()

    # part B
    elif part == 'B':
        part_b()

    else:
        print("Not Valid")
```

Reminder:

$$Accuracy = \frac{T_p + T_n}{T_p + T_n + F_p + F_n}$$

$$Precision = \frac{T_p}{T_p + F_p}$$

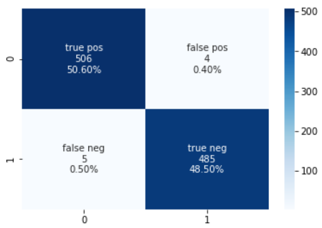
$$Recall = \frac{T_p}{T_p + T_n}$$

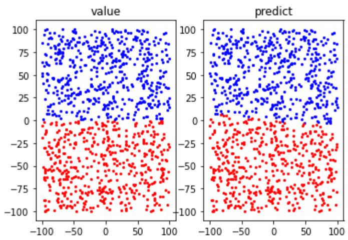
$$F_1 = 2 \cdot \frac{precision \cdot recall}{precision + recall}$$

In [10]:

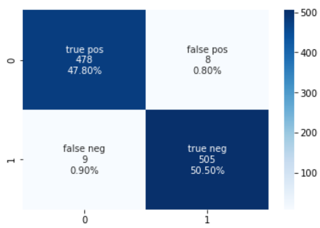
```python
main()
```

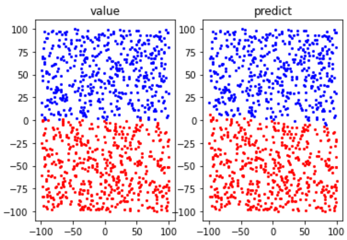
```
Enter the relevant part (A or B): A
First Test:
Acurracy score: 99.1
              precision    recall  f1-score   support

          -1       0.99      0.99      0.99       510
           1       0.99      0.99      0.99       490

    accuracy                           0.99      1000
   macro avg       0.99      0.99      0.99      1000
weighted avg       0.99      0.99      0.99      1000
```






```
Second Test:
Acurracy score: 98.3
              precision    recall  f1-score   support

          -1       0.98      0.98      0.98       486
           1       0.98      0.98      0.98       514

    accuracy                           0.98      1000
   macro avg       0.98      0.98      0.98      1000
weighted avg       0.98      0.98      0.98      1000
```





Part B

Tables

First test set results table:

Accuracy score: 99.9%

Confusion matrix:

True Positive	False Positive	True Negative	False Negative
99.9%	0.00%	0.00%	0.10%

Precision: 100%
Recall: 100%
F1-score: 100%

Second test set results table:

Accuracy score: 99.9%

Confusion matrix:

True Positive	False Positive	True Negative	False Negative
99.9%	0.00%	0.00%	0.10%

Precision: 100%
Recall: 100%
F1-score: 100%

Third test set (where we changed the range for the radius to [5, 50]) results table:

Accuracy score: 49.9%

Confusion matrix:

True Positive	False Positive	True Negative	False Negative
40.10%	40.00%	9.80%	10.10%

Precision: 80%
Recall: 50%
F1-score: 62%

The Adaline is for linear separatable equations classification, thus the accuracy score of the last test is low.

Train B:

	x	y	value
0	51.71	22.32	-1
1	-12.47	-54.61	-1
2	-39.01	-97.9	-1
3	10.86	64.75	-1
4	51.95	98.14	-1
5	-73.52	9.45	-1
6	81.62	-86.6	-1
7	24.19	-44.49	-1
8	48.11	38.47	-1
9	-48.51	-44.81	-1
10	-22.01	-83.18	-1

First Test B:

	x	y	value
0	-22.03	94.19	-1
1	78.33	-57.27	-1
2	21.22	97.89	-1
3	55.33	90.33	-1
4	-78.53	98.44	-1
5	-95.69	53.75	-1
6	-15.02	80.48	-1
7	-23.22	-37.17	-1
8	54.09	77.26	-1
9	80.1	56.09	-1
10	30.13	-50.65	-1

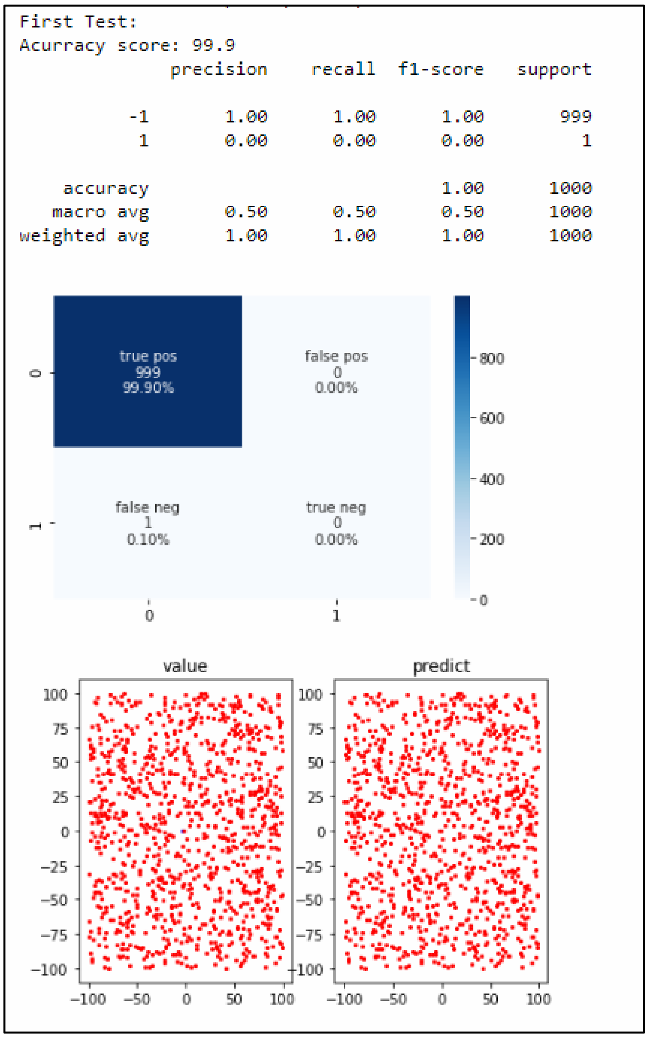
Second Test B:

	x	y	value
0	6.11	-50.57	-1
1	29.37	-84.18	-1
2	-76.27	75.59	-1
3	-69.16	19.82	-1
4	90.96	-81.0	-1
5	66.27	-29.65	-1
6	-87.72	-71.84	-1
7	42.09	37.02	-1
8	-77.11	-21.14	-1
9	-70.28	80.56	-1
10	39.1	-80.64	-1

Discussions and Illustrations

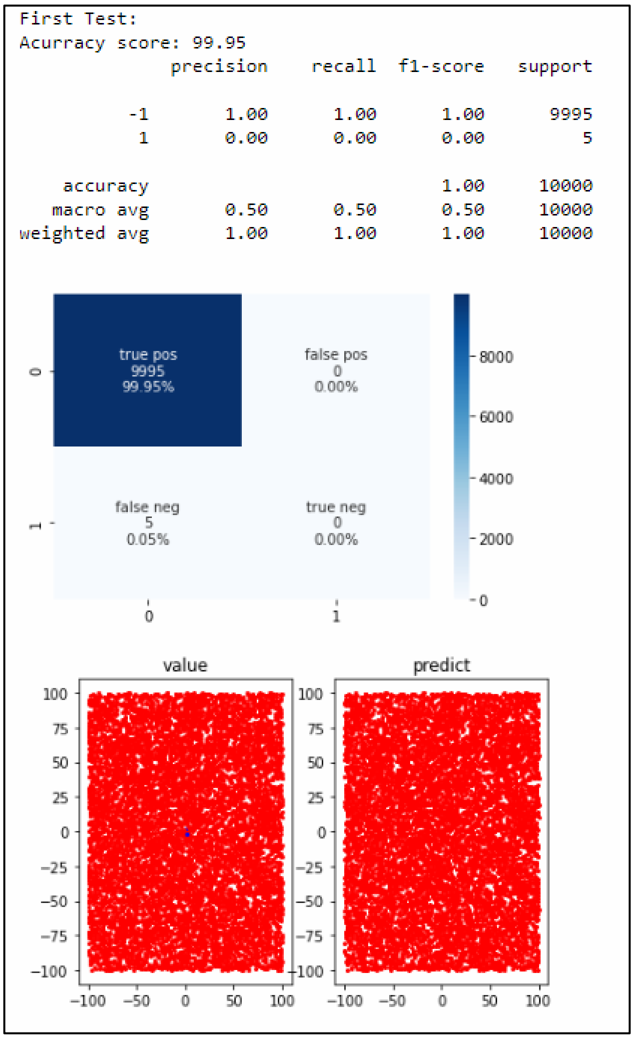
We tried fitting the Adaline with a bigger training set, it took the same number of iterations of the data to get the MSE below the threshold, we did not see a difference in the accuracy.

using 1,000 points

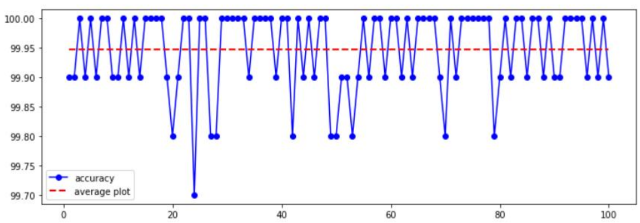

First Test:
Acurracy score: 99.9

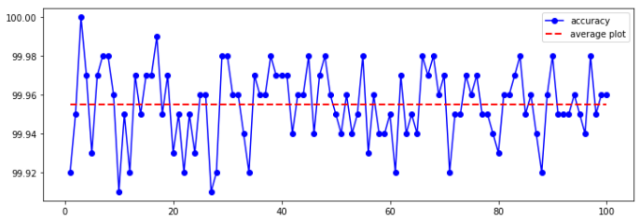
	precision	recall	f1-score	support
-1	1.00	1.00	1.00	999
1	0.00	0.00	0.00	1
accuracy			1.00	1000
macro avg	0.50	0.50	0.50	1000
weighted avg	1.00	1.00	1.00	1000

using 10,000 points


First Test:
Acurracy score: 99.95

	precision	recall	f1-score	support
-1	1.00	1.00	1.00	9995
1	0.00	0.00	0.00	5
accuracy			1.00	10000
macro avg	0.50	0.50	0.50	10000
weighted avg	1.00	1.00	1.00	10000



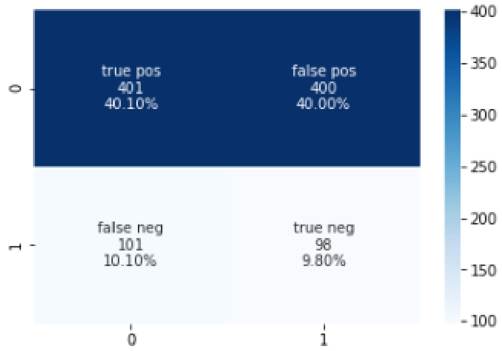


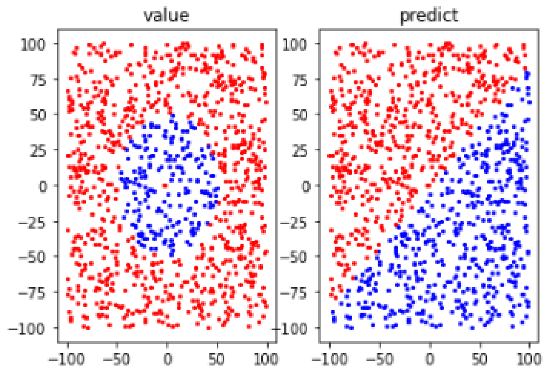
We ran the Adaline algorithm 100 times for both size of data.
You can see in the graphs above that the average accuracy is higher a tiny bit higher with a smaller training set and testing set, but nit anything significant.

We also tried making the circles area a lot bigger using the equation $25 \leq x^2 + y^2 < 2500$ (if the point is inside the circles with radiuses 5 and 50) when we tried to fit this model it did not work well because the Adaline can only find a linear division and the division is not linear.

The reason the Adaline worked with the circles we were given is because the area of those circles is smaller than 0.5% of the total area, Thus the Adaline didn't need to have a line to linearly divide it.

```
First Test:
Acurracy score: 49.9
              precision    recall  f1-score   support

          -1       0.80      0.50      0.62       801
           1       0.20      0.49      0.28       199

    accuracy                           0.50      1000
   macro avg       0.50      0.50      0.45      1000
weighted avg       0.68      0.50      0.55      1000
```

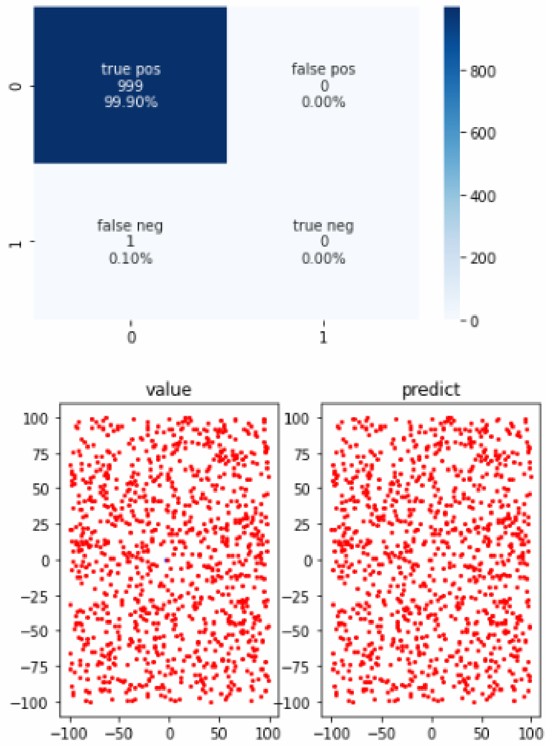




Comparison between two different training sets:

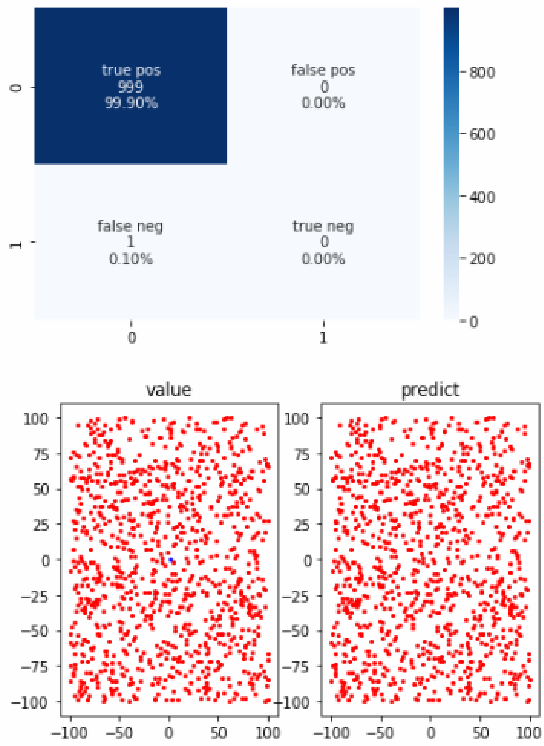
Using the train as a train and test2 for the test

```
First Test:
Acurracy score: 99.9
              precision    recall  f1-score   support

          -1       1.00      1.00      1.00       999
           1       0.00      0.00      0.00         1

    accuracy                           1.00      1000
   macro avg       0.50      0.50      0.50      1000
weighted avg       1.00      1.00      1.00      1000
```



Using test1 as a train and test2 for the test

```
Second Test:
Acurracy score: 99.9
              precision    recall  f1-score   support

          -1       1.00      1.00      1.00       999
           1       0.00      0.00      0.00         1

    accuracy                           1.00      1000
   macro avg       0.50      0.50      0.50      1000
weighted avg       1.00      1.00      1.00      1000
```



Code

Parts A and B:

In [1]:

```
#libraries
import numpy as np
import pandas as pd

import random

#preproccesing
from sklearn.metrics import classification_report,f1_score
from sklearn.metrics import confusion_matrix

# visulization
import matplotlib.pyplot as plt
import seaborn as sns

import warnings
warnings.filterwarnings('ignore')
```

In [2]:

```
class Adaline:
    def __init__(self, learning_rate, train):
        self.learning_rate = learning_rate
        self.train = train

    # this function generates random small weights and bias for the Adaline algorithm
    def _weight_genarate(self):
        weight = [] # [x,y]
        for i in range(2):
            random.seed(i)
            rand = random.uniform(0, 0.01)
            rand = round(rand, 4)
            weight.append(rand)

        # now generate the bias
        random.seed(4)
        bias = random.uniform(0, 1)
        bias = round(bias, 4)
        return weight, bias

    # this function fits the adaline model on the training data
    def fit(self):
        ERR = []
        mse = []
        EPS = 0.001
        # generate weights and bias
        weight, bias = self._weight_genarate()
        oldmse=1
        while(True):
            ERR = []
            # for each row we fix the bias and wights in order to get the minimum error
            for index, row in self.train.iterrows():
                predicted = bias + row['x']/100 * weight[0] + row['y']/100 * weight[1]

                weight[0] = round((weight[0] + self.learning_rate * (row['value'] - predicted) * row['x']/100), 3)
                weight[1] = round((weight[1] + self.learning_rate * (row['value'] - predicted) * row['y']/100), 3)
                bias = round((bias + self.learning_rate * (row['value'] - predicted)), 3)

                # error calculation
                error = (row['value'] - predicted) ** 2
                # if the error is small enough return
                ERR.append(error)

            mse.append(np.sum(ERR))
            if len(mse) >= 2:
                # checking if the error is smaller then eps or if it hasnt changed
                if abs(mse[-1] - mse[-2]) < EPS or abs(mse[-1] - mse[-2])==oldmse :
                    break
            # updating the old mse
            if len(mse)>=2:
                oldmse=abs(mse[-1] - mse[-2])
        return weight, bias

    # this function predicts on a test data and returns the number of correct predictions
    def predict(self, test, weight, bias):
        count = 0
        pred = []
        # for each row we use the activation formula with the weights and bias we returned
        # in the fit function to predict on the test data set
        for index, row in test.iterrows():
            prediction = bias + (row['x'] * weight[0]) + (row['y'] * weight[1])
            if prediction > 0:
                prediction = 1
            else:
                prediction = -1
            pred.append(prediction)

            if prediction == row['value']:
                count += 1
        # now add the prediction list to the data set in order to make comparison
        test['predict'] = pred
        return count

    # this function caculates the acuuracy of the predictions
    def score(self, pred, test):
        acurr = pred / len(test)
        res = round(acurr, 4)
        return res
```

In [4]:
```python
# this function builds the data set for part B of the assighnment
def build_data_partB(i):
    x = []
    y = []
    value = []
    random.seed(i)
    for i in range(1000):
        # generate two random numbers between -10000 to 10000
        randX = random.randint(-10000, 10000)
        randY = random.randint(-10000, 10000)
        x.append(randX / 100)
        y.append(randY / 100)
        # for part A if (4 <= y^2 + x^2 <= 9) then the value is 1
        if 4 <= (y[i] ** 2 + x[i] ** 2) <= 9:
            value.append(1)
        # else the value is -1
        else:
            value.append(-1)

    # make the data frame
    end = {'x': x, 'y': y, 'value': value}
    df = pd.DataFrame(data=end, columns=['x', 'y', 'value'])
    return df
```

In [5]:
```python
# this function plots the values of the actual values of the data compared to the prediction values we predicted
def plotting(test):
    f, ax = plt.subplots(1, 2)
    ax[0].set_title("value")
    ax[1].set_title("predict")

    for index, row in test.iterrows():
        if row['value'] == 1:
            ax[0].plot(row['x'], row['y'], markersize=2, marker="o", color="blue")
        else:
            ax[0].plot(row['x'], row['y'], markersize=2, marker="o", color="red")
        if row['predict'] == 1:
            ax[1].plot(row['x'], row['y'], markersize=2, marker="o", color="blue")
        else:
            ax[1].plot(row['x'], row['y'], markersize=2, marker="o", color="red")
    plt.show()
```

confussion matrix:

a confusion matrix, also known as an error matrix, is a specific table layout that allows visualization of the performance of an algorithm, typically a supervised learning one (in unsupervised learning it is usually called a matching matrix). Each row of the matrix represents the instances in an actual class while each column represents the instances in a predicted class, or vice versa – both variants are found in the literature. The name stems from the fact that it makes it easy to see whether the system is confusing two classes (i.e. commonly mislabeling one as another).

wikipedia - https://en.wikipedia.org/wiki/Confusion_matrix

In [6]:
```python
# this function plots the confussion matrix
def confussion_matrix (cf_matrix):
    group_names = ['true pos', 'false pos', 'false neg', 'true neg']
    group_counts = ["{0:0.0f}".format(value) for value in cf_matrix.flatten()]
    group_percentages = ["{0:.2%}".format(value) for value in cf_matrix.flatten()/np.sum(cf_matrix)]
    labels = [f"{v1}\n{v2}\n{v3}" for v1, v2, v3 in zip(group_names,group_counts,group_percentages)]
    labels = np.asarray(labels).reshape(2,2)
    sns.heatmap(cf_matrix, annot=labels, fmt='', cmap='Blues')
```

In [8]:
```python
def part_b():
    train = build_data_partB(9)
    first_test = build_data_partB(3)
    second_test = build_data_partB(7)

#     train.to_csv(r'C:\Users\talia\NeuroComputation\NeuroComputation\train_B.csv')
#     first_test.to_csv(r'C:\Users\talia\NeuroComputation\NeuroComputation\first_test_B.csv')
#     second_test.to_csv(r'C:\Users\talia\NeuroComputation\NeuroComputation\second_test_B.csv')

    print("First Test:")
    # run Adaline algorithm
    ada = Adaline(0.2, train)
    weight, bias = ada.fit()
    ada_pred = ada.predict(first_test, weight, bias)
    ada_score = ada.score(ada_pred, first_test)
    print("Acurracy score:", ada_score * 100)

    # confusion matrix
    con_mat = confusion_matrix(first_test['value'], first_test['predict'])
    confussion_matrix(con_mat)
    print(classification_report(first_test['value'], first_test['predict']))
    plotting(first_test)

    print("Second Test:")
    # run Adaline algorithm
    ada = Adaline(0.2, train)
    weight, bias = ada.fit()
    ada_pred = ada.predict(second_test, weight, bias)
    ada_score = ada.score(ada_pred, second_test)
    print("Acurracy score:", ada_score * 100)

    # confusion matrix
    con_mat = confusion_matrix(second_test['value'], second_test['predict'])
    confussion_matrix(con_mat)
    print(classification_report(second_test['value'], second_test['predict']))
    plotting(second_test)
```

In [9]:

```python
# this is the main function
def main():
    part = input("Enter the relevant part (A or B): ")

    # part A
    if part == 'A':
        part_a()

    # part B
    elif part == 'B':
        part_b()

    else:
        print("Not Valid")
```

Reminder:

$$Accuracy = \frac{T_p + T_n}{T_p + T_n + F_p + F_n}$$

$$Precision = \frac{T_p}{T_p + F_p}$$

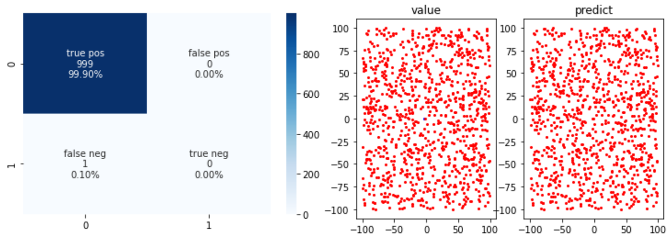
$$Recall = \frac{T_p}{T_p + T_n}$$

$$F_1 = 2 \cdot \frac{precision \cdot recall}{precision + recall}$$

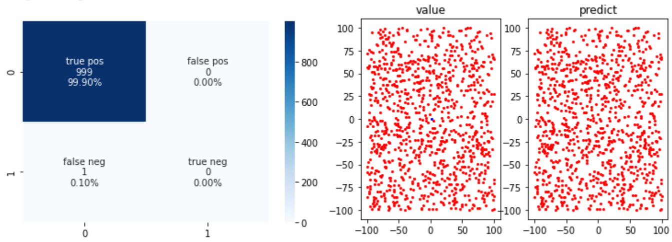
In [11]:

```python
main()
```

```
Enter the relevant part (A or B): B
First Test:
Acurracy score: 99.9
              precision    recall  f1-score   support

          -1       1.00      1.00      1.00       999
           1       0.00      0.00      0.00         1

    accuracy                           1.00      1000
   macro avg       0.50      0.50      0.50      1000
weighted avg       1.00      1.00      1.00      1000
```




```
Second Test:
Acurracy score: 99.9
              precision    recall  f1-score   support

          -1       1.00      1.00      1.00       999
           1       0.00      0.00      0.00         1

    accuracy                           1.00      1000
   macro avg       0.50      0.50      0.50      1000
weighted avg       1.00      1.00      1.00      1000
```

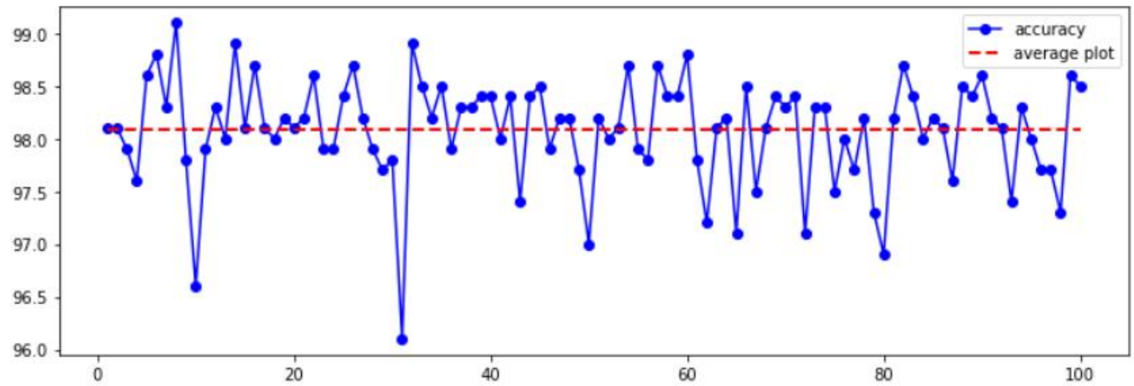


Summary and Discussion

In summary we noticed that it does not matter what training set we use if it is the same size the Adaline model should work. Also, we noticed that if we give that model a trainset that is too big it will over fit the data. And we learned that the model does not work with data that is divided nonlinearly.

We ran a loop 100 times to see how accurate our models are

here are the results for part A:
The average is: 98.081%
The minimum is: 96.1%
The maximum is: 99.1%



And here are the results for part B:
The average is: 99.947%
The min is: 99.7%
The max is: 100%