AB1:

Project 1 (Parts A + B):

Name: Rotem Halbreich **ID:** 311549364

Name: Ishay Levy **ID:** 318439759

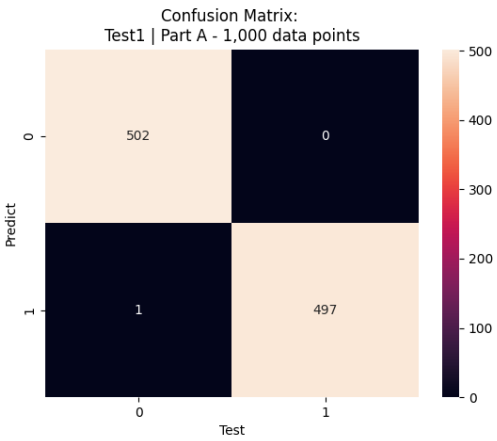
Part A:

A2: *(Tables of part A)*

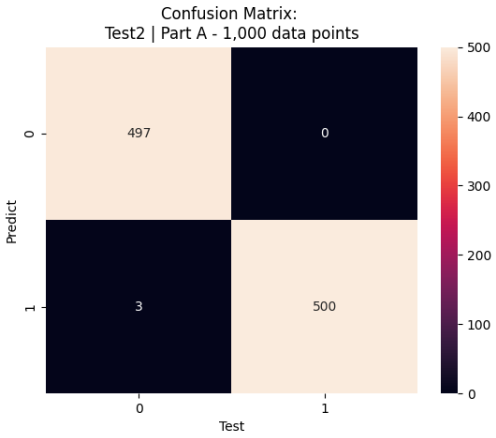
```
################################ PART A ################################

~~~~~~ TEST 1 ~~~~~~
Prediction:  99.9 %
Error:  0.004

~~~~~~ TEST 2 ~~~~~~
Prediction:  99.7 %
Error:  0.004

################################ PART B ################################

~~~~~~ Part B Model ~~~~~~
Prediction:  49.9 %
Error:  0.044
```

Confusion Matrix:

Test 1:

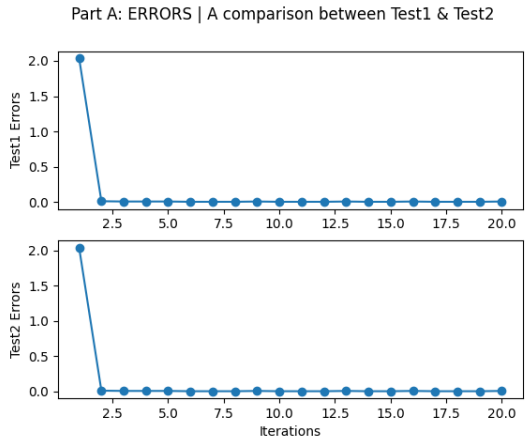


Test 2:

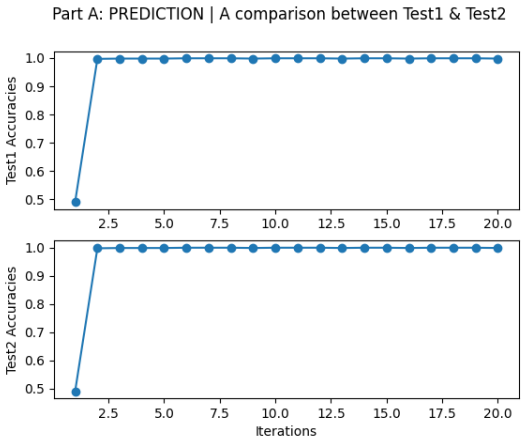


A3: *(discussions and illustrations)*

Errors:



Prediction:



What can you conclude about your results?

Restriction range for the data points such that $value = 1$ only if $y > 1$, else $value = -1$.

There's not a big difference between the Test1 & Test2, we obtained 99.9% prediction for Test1, and 99.7% prediction for Test2, which means that the accuracy is dependent on the training set. The 2 test sets are picked randomly, and therefore we see a little change in the prediction section because both aren't identical, as well as in the confusion matrix. The change we see is not significant because the training set is already big enough (1,000 data points), but if we'll use a bigger training test, we'll get better results.

A4: *(Code and any additional information)*

Implementation of Adaline Algorithm class including Part A & Part B classification in step_function() and create_data():

```
import numpy as np
from numpy import random
import matplotlib.pyplot as plt
from sklearn.metrics import confusion_matrix
import seaborn as sns


class Adaline(object):

    def __init__(self, learning_rate=0.01, n_iter=20, shuffle=True):
        self.learning_rate = learning_rate
        self.n_iter = n_iter
        self.shuffle = shuffle
        self.weights = []
        self.errors = []
        self.accuracies = []


    def fit(self, X, y):
        """
        Fit training data.
        X : Training vectors, X.shape : [#samples, #features]
        y : Target values, y.shape : [#samples]
        :param X:
        :param y:
        :return:
        """
        self.weights = self.random_weights(X, random_state=1) # Initialize weights

        for _ in range(self.n_iter):

            if self.shuffle:
                X, y = self._shuffle(X, y) # Shuffle the data

            predicted = []
            for x in self.activation(X):
                predicted.append(step_function(x))
            output = self.activation(X)
            gradient = 2 * (y - predicted)  # Compute the gradient of error via the weights
            self.weights[1:] += self.learning_rate * X.T.dot(gradient)  # Update weights
            self.weights[0] += self.learning_rate * gradient.sum()  # Update bias
            self.errors.append(((np.array(predicted) - np.array(y)) ** 2).mean())  # Add all errors to a
list
            self.accuracies.append(prediction(predicted, y))  # Add all accuracies to a list

    def random_weights(self, X, random_state: int):
        """
        Creates a vector of random weights
        :param X:
        :param random_state:
        :return:
        """
        rand = np.random.RandomState(random_state)
        self.weights = rand.normal(loc=0.0, scale=0.01, size=1 + X.shape[1])
        return self.weights

    def _shuffle(self, X, y):
        """
        Shuffle training data with np random permutation
        :param X:
        :param y:
        :return:
        """
        rand = np.random.permutation(len(y))
        return X[rand], y[rand]

    def net_input(self, X):
        """
        Computes a net input as dot product
        :param X:
        :return:
        """
        return np.dot(X, self.weights[1:]) + self.weights[0]
```

```
    def activation(self, X):
        """
        Computes a linear activation.
        :param X:
        :return:
        """
        return self.net_input(X)

    def predict(self, X, Part):
        """
        Returns a class label after a unit step
        :param X:
        :param Part:
        :return:
        """
        if Part == 'A':
            return [step_function(x) for x in self.net_input(X)]
        else:
            return np.where(self.net_input(X) >= 1.0, 1, -1)

# Outside the Class

def step_function(data):
    """
    Step function:
    Part A: returns 1 if Y>1, else -> -1
    Part B: returns 1 if if (4 <= X^2 + Y^2 <=9), else -> -1
    """
    if type(data) is np.float64 or type(data) is float or type(data) is np.ndarray:
        if type(data) is np.float64 or type(data) is float:
            return 1 if data > 1 else -1
        else:
            return 1 if data[1] > 1 else -1 # if(Y > 1)
    elif type(data) == tuple or type(data) == list:
        value = (data[0] ** 2) + (data[1] ** 2)
        return 1 if 4 <= value <= 9 else -1 # if(4 <= X^2 + Y^2 <=9)
    else:
        raise ValueError(f"Invalid input {data}")

def prediction(pred, test_label):
    """
    Returns the accuracy
    :param pred:
    :param test_label:
    :return:
    """
    predicted = len(pred)
    count = 0
    for i in range(predicted):
        if pred[i] == test_label[i]:
            count += 1
    return count / predicted

def create_data(n, Part):
    """
    Creates dots & labels lists, dots contains tuples with random values in form: <x, y>,
    labels contains labels with data values: 1 if (Y > 1) else -> -1
    :param n:
    :param Part:
    :return:
    """
    dots = []
    labels = []
    for i in range(n):
        dot = (random.randint(-10000, 10000) / 100, random.randint(-10000, 10000) / 100)
        dots.append(dot)
        if Part == 'A':
            labels.append(step_function(dot[1]))
        else:
            labels.append(step_function(dot))
    return (np.array(dots), np.array(labels))
```

Creation of tables & graphs of Part A:

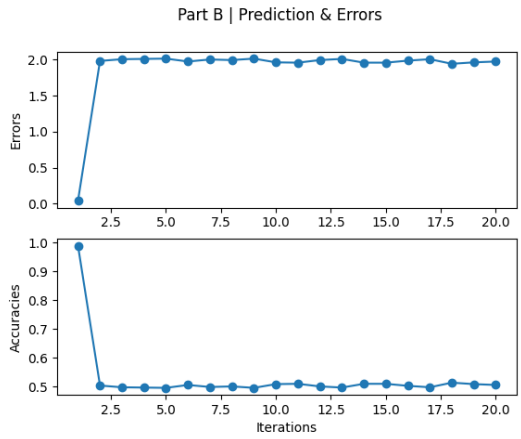
```
def partA():
    print("\n############################### PART A ###############################\n")

    # Create an Adaline classifier & train based on the data
    train_data, train_label = create_data(1000, 'A')

    # Create & Fit a model
    classifier = Adaline()
    classifier.fit(np.array(train_data), np.array(train_label))

    classifier_2 = Adaline()
    classifier_2.fit(np.array(train_data), np.array(train_label))

    # Create data accuracies:
    x1 = range(1, len(classifier.accuracies) + 1)
    y1 = classifier.accuracies
    x2 = range(1, len(classifier_2.accuracies) + 1)
    y2 = classifier_2.accuracies

    fig, (ax1, ax2) = plt.subplots(2, 1)
    fig.suptitle('Part A: PREDICTION | A comparison between Test1 & Test2')

    ax1.plot(x1, y1, 'o-')
    ax1.set_ylabel('Test1 Accuracies')

    ax2.plot(x2, y2, 'o-')
    ax2.set_xlabel('Iterations')
    ax2.set_ylabel('Test2 Accuracies')
    plt.show()

    # Create data errors:
    x1 = range(1, len(classifier.errors) + 1)
    y1 = classifier.errors
    x2 = range(1, len(classifier_2.errors) + 1)
    y2 = classifier_2.errors

    fig, (ax1, ax2) = plt.subplots(2, 1)
    fig.suptitle('Part A: ERRORS | A comparison between Test1 & Test2')

    ax1.plot(x1, y1, 'o-')
    ax1.set_ylabel('Test1 Errors')

    ax2.plot(x2, y2, 'o-')
    ax2.set_xlabel('Iterations')
    ax2.set_ylabel('Test2 Errors')
    plt.show()


    # ______________________________
    # TEST 1 - Prediction & Errors:
    # ------------------------------
    predicted = classifier.predict(np.array(train_data), 'A')

    print("~~~~~~ TEST 1 ~~~~~~")
    print("Prediction: ", prediction(train_label, predicted) * 100, "%")
    print("Error: ", "{:.5}".format(np.array(classifier.errors).min()))

    # Confusion Matrix:
    cm = confusion_matrix(classifier.predict(train_data, 'A'), train_label)
    plt.subplots()
    sns.heatmap(cm, fmt=".0f", annot=True)
    plt.title("Confusion Matrix: \nTest1 | Part A - 1,000 data points")
    plt.xlabel("Test")
    plt.ylabel("Predict")
    plt.show()

    # ______________________________
    # TEST 2 - Prediction & Errors:
    # ------------------------------
    test_data_2, test_label_2 = create_data(1000, 'A')
    predicted_2 = classifier_2.predict(np.array(test_data_2), 'A')

    print("\n~~~~~~ TEST 2 ~~~~~~")
    print("Prediction: ", prediction(test_label_2, predicted_2) * 100, "%")
    print("Error: ", "{:.5}".format(np.array(classifier_2.errors).min()))

    # Confusion Matrix:
    cm = confusion_matrix(classifier.predict(test_data_2, 'A'), test_label_2)
    plt.subplots()
    sns.heatmap(cm, fmt=".0f", annot=True)
    plt.title("Confusion Matrix: \nTest2 | Part A - 1,000 data points")
    plt.xlabel("Test")
    plt.ylabel("Predict")
    plt.show()
```

Part B:

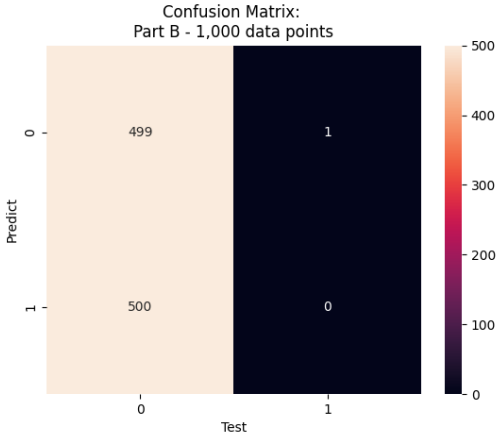
B2:

1,000 points data set:

Errors & Prediction:



Confusion Matrix:

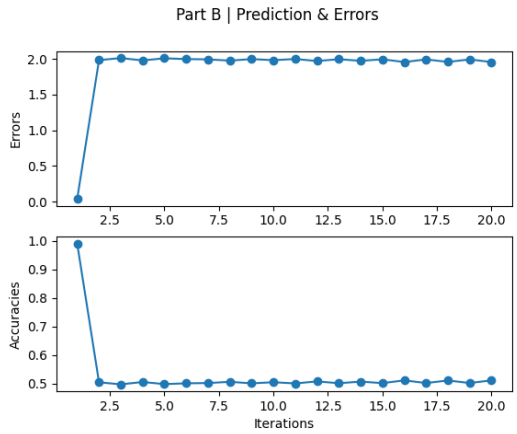


Creation of tables & graphs of Part B:

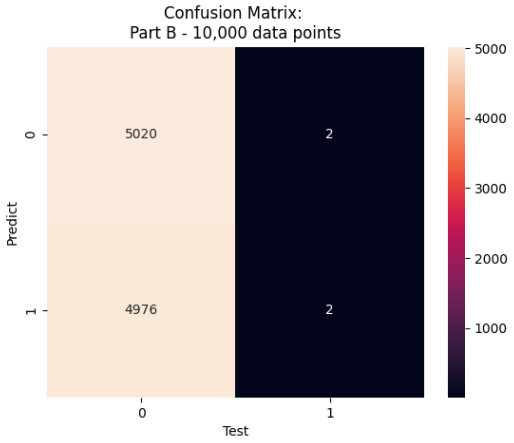
```
def partB():
    print("\n################################ PART B ################################")

    # Create an Adaline classifier & train based on the data
    train_data, train_label = create_data(1000, 'B')

    # Create & Fit a model
    classifier = Adaline()
    classifier.fit(np.array(train_data), np.array(train_label))


    # Create data:
    x1 = range(1, len(classifier.errors) + 1)
    y1 = classifier.errors
    x2 = range(1, len(classifier.accuracies) + 1)
    y2 = classifier.accuracies

    fig, (ax1, ax2) = plt.subplots(2, 1)
    fig.suptitle('Part B | Prediction & Errors')

    ax1.plot(x1, y1, 'o-')
    ax1.set_ylabel('Errors')

    ax2.plot(x2, y2, 'o-')
    ax2.set_xlabel('Iterations')
    ax2.set_ylabel('Accuracies')
    plt.show()

    # ______________________________
    # PART B - Prediction & Errors:
    # ------------------------------
    predicted = classifier.predict(np.array(train_data), 'B')

    print("\n~~~~~~ Part B Model ~~~~~~")
    print("Prediction: ", "{:.5}".format(prediction(train_label, predicted) * 100), "%")
    print("Error: ", "{:.5}".format(np.array(classifier.errors).min()))

    # Confusion Matrix:
    cm = confusion_matrix(classifier.predict(train_data, 'B'), train_label)
    plt.subplots()
    sns.heatmap(cm, fmt=".0f", annot=True)
    plt.title("Confusion Matrix: \nPart B - 1,000 data points")
    plt.xlabel("Test")
    plt.ylabel("Predict")
    plt.show()
```

10,000 points data set:

Errors & Prediction:



Confusion Matrix:



What can you conclude about your results?

Now we change the restriction range for the data points such that $value = 1$ only if

$4 \leq x^2 + y^2 \leq 9$, else $value = -1$.

The best results we obtain using the Adaline Algorithm are around 49.9% with the use of $data\_size = 1{,}000$. The results are a little better when we used 10,000 data points (50.22%), because our model classified all data as "-1". Because there is no average distribution, if we have more data points, we get more values in the "-1" area. In conclusion, our model doesn't get far better while using more data. Adaline is a linear model using a linear activation function (the identity function) and there is no possible way to classified values with high success rates, which are scattered in a non-linear way as a ring, therefore we can see that half of the data points are getting a false positive result which means there's a problem training this non-linear model with Adaline Algorithm.