Adaline-Adaptive Linear Neuron

By Avidan Abitbol 302298963 and Itamar Kraitman 208925578

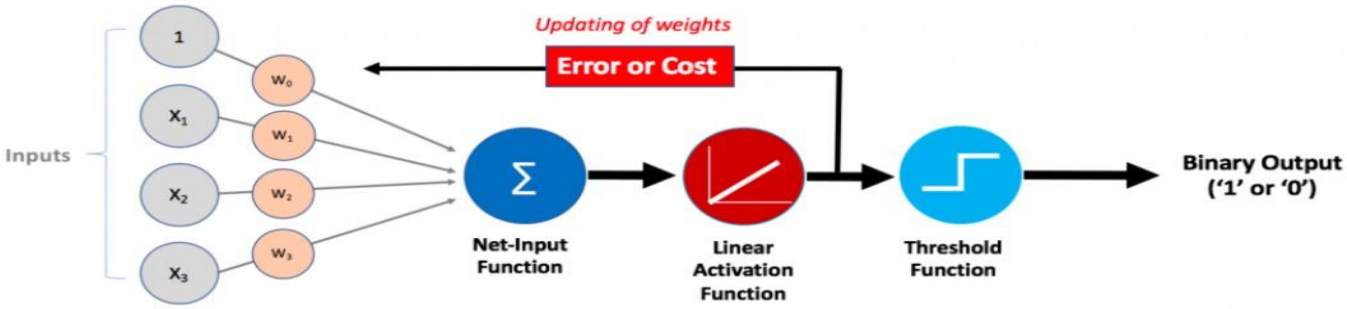


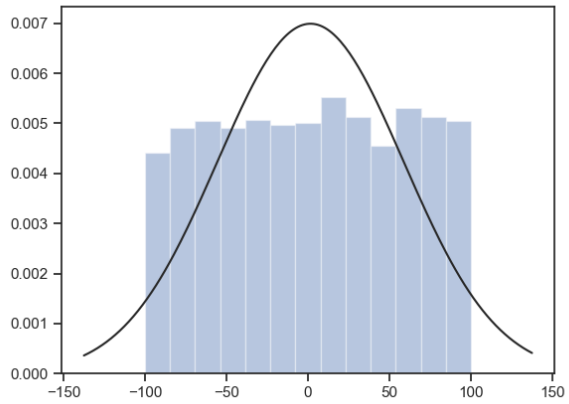
Fig 1. Adaline - Single-layer neural network

Part A

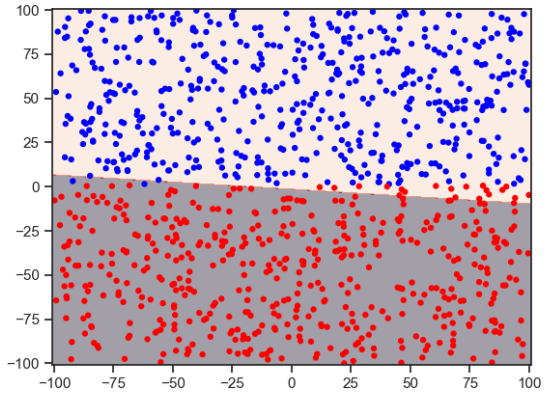
In this part, we created the train dataset according to the following condition: `1` iff y>`1`, and `-1` otherwise. Our first action was to find the best parameters for training the set with.

Firstly, in order to make sure that the train set is well distributed we viewed its distribution and linear separation.

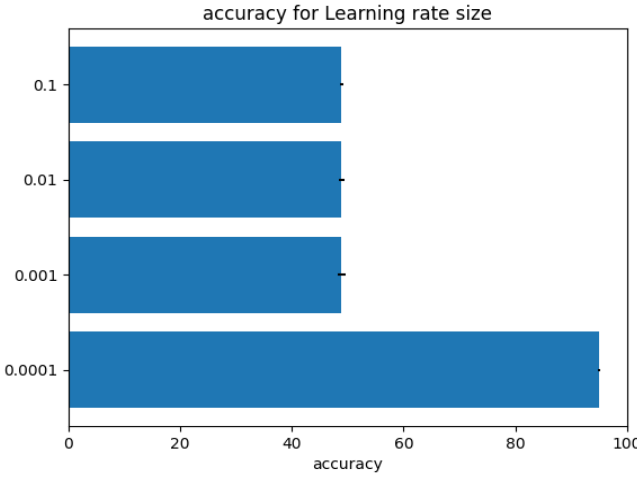
Normal distribution

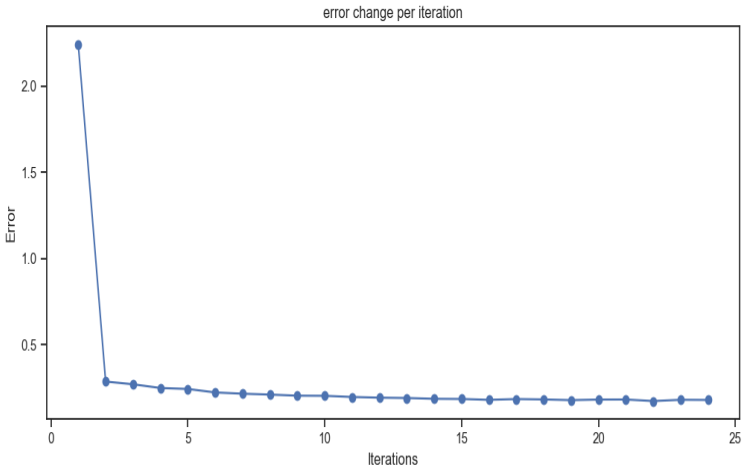


Linear separation between two classes



Secondly, in order to find the most accurate learning rate we made predictions with four different learning rates. As we can see, the learning rate of `0.0001` had the best score. Then we found that the number of iterations of `24` alongside with the learning rate we found had the best result.

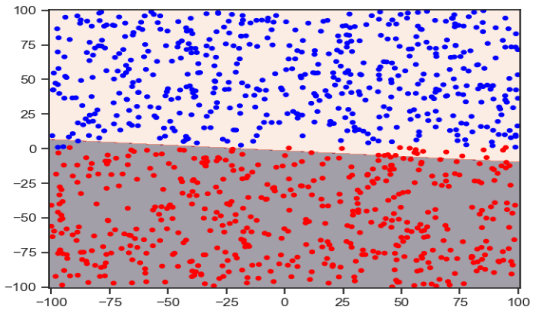


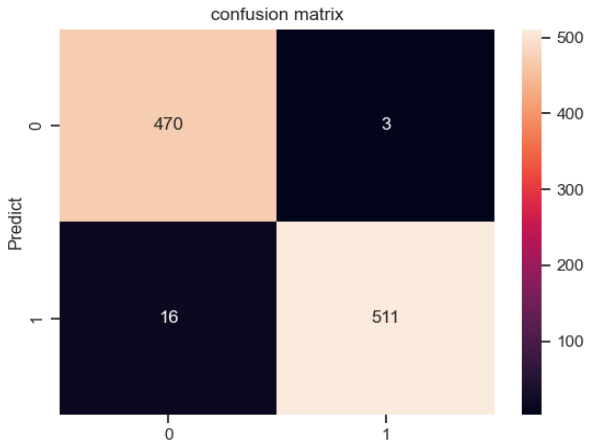


After finding the best parameters for our neuron, we trained the Adaline neuron with those parameters.

By the time the neuron was trained successfully we created two different test sets and made predictions on them with the neuron. The results were as following:

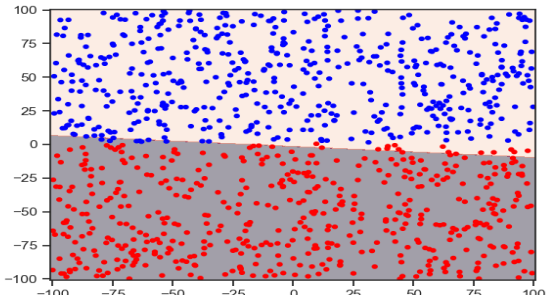
Second test set

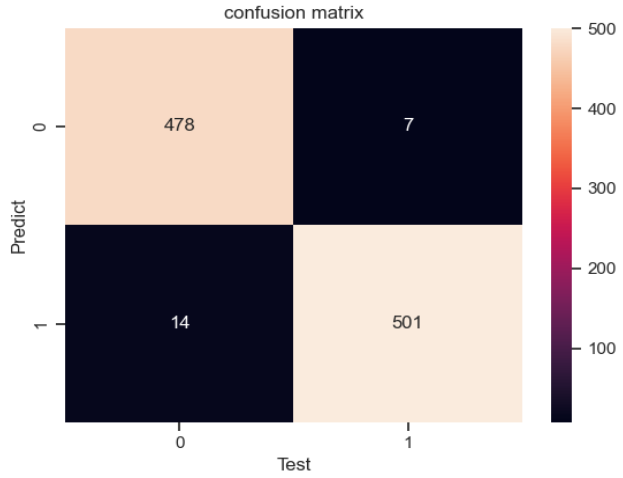




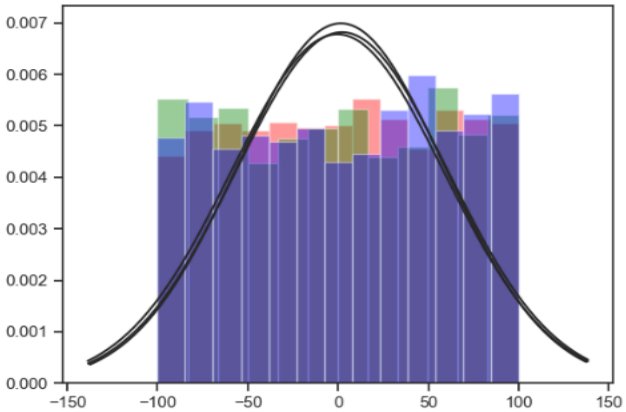
	precision	recall	f1-score	support
-1.0	0.99	0.97	0.98	486
1.0	0.97	0.99	0.98	514
accuracy			0.98	1000
macro avg	0.98	0.98	0.98	1000
weighted avg	0.98	0.98	0.98	1000

first test set



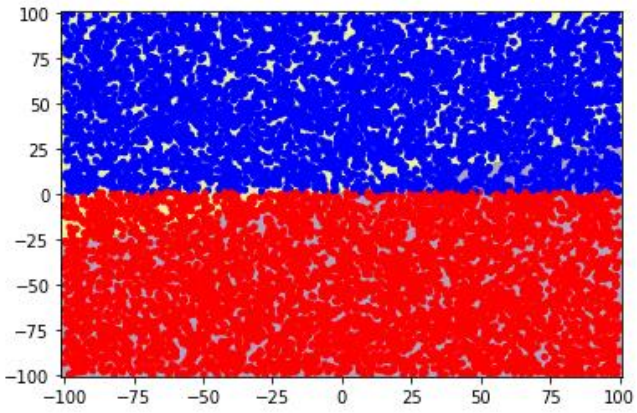


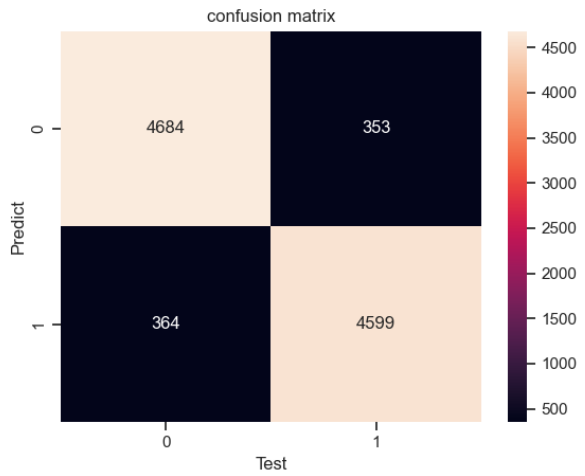
	precision	recall	f1-score	support
-1.0	0.99	0.97	0.98	492
1.0	0.97	0.99	0.98	508
accuracy			0.98	1000
macro avg	0.98	0.98	0.98	1000
weighted avg	0.98	0.98	0.98	1000



After comparing two different test sets, we wanted to find out if a data set of 10,000 samples will get better results than the "regular" set of 1,000 samples , we found that the big data set had some difficulties to make a linear separation on the training set which reflected in the results with lower accuracy than the regular set.

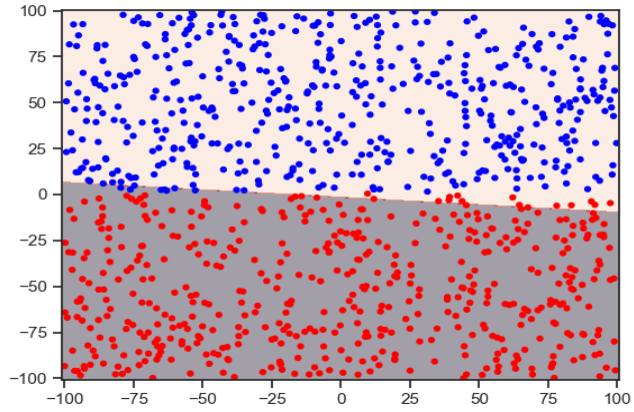
Dataset with 10,000 samples

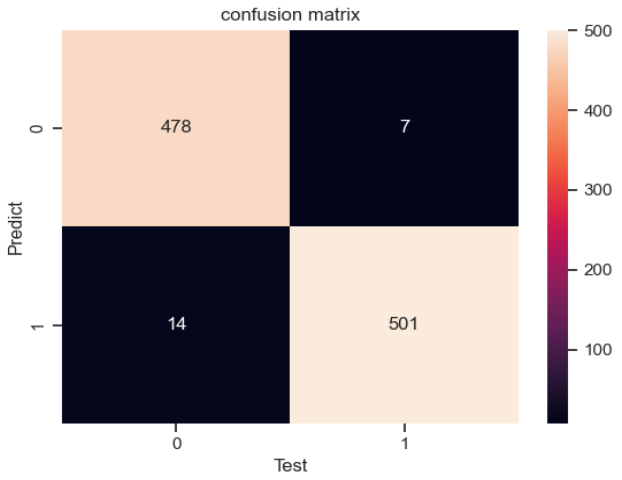




	precision	recall	f1-score	support
-1.0	0.93	0.93	0.93	5048
1.0	0.93	0.93	0.93	4952
accuracy			0.93	10000
macro avg	0.93	0.93	0.93	10000
weighted avg	0.93	0.93	0.93	10000

Dataset with 1,000 samples





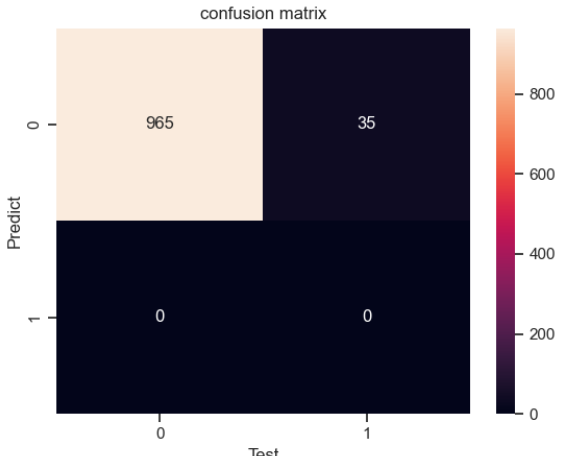
	precision	recall	f1-score	support
-1.0	0.99	0.97	0.98	492
1.0	0.97	0.99	0.98	508
accuracy			0.98	1000
macro avg	0.98	0.98	0.98	1000
weighted avg	0.98	0.98	0.98	1000

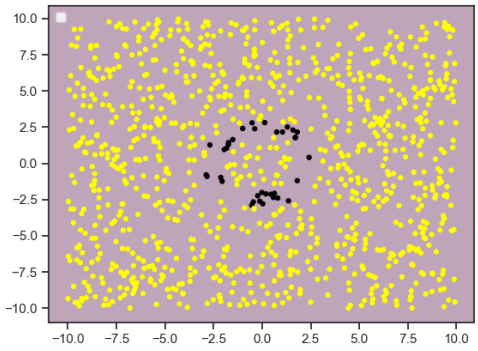
Part B

In this part, we created the train dataset according to the following condition: `1` iff `4 <= x^2 + y^2 <= 9`, and `-1` otherwise. Our first action was to find the bet parameters for training the set with.

In order to find the best parameters, we did as part A. We observed that the best learning rate is the same as Part A – `0.0001` but as can be seen in the following figures, we obtained the best result with size of `10,000` samples- `95.78%` over `95.5%` with size of `1,000`.

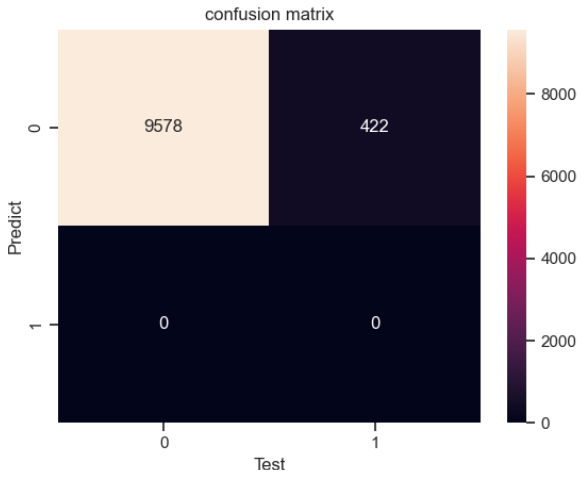
Size of 1,000

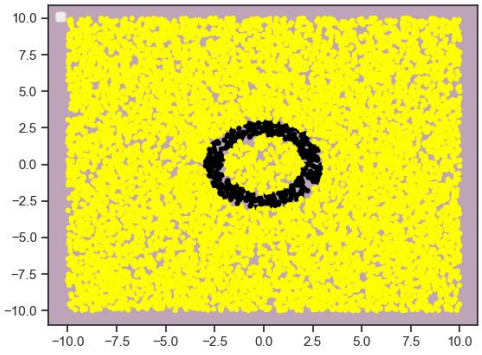




	precision	recall	f1-score	support
-1.0	0.96	1.00	0.98	965
1.0	0.00	0.00	0.00	35
accuracy			0.96	1000
macro avg	0.48	0.50	0.49	1000
weighted avg	0.93	0.96	0.95	1000

Size of 10,000





	precision	recall	f1-score	support
-1.0	0.96	1.00	0.98	9578
1.0	0.00	0.00	0.00	422
accuracy			0.96	10000
macro avg	0.48	0.50	0.49	10000
weighted avg	0.92	0.96	0.94	10000

Summary

In conclusion, we can summarize the process of finding the parameters which will give us the most accurate neuron to three levels:

1. finding the ideal learning rate
2. finding the ideal number of iterations.
3. Finding the ideal size of dataset.

The ideal way is to perform some combination of all parameters.

Adaline neuron course of action

In this research, we used the benefits of the Adaline neuron. But what Adaline is? Adaline or Adaptive Linear Neuron is a single-layer ANN McCulloch–Pitts neuron used in algorithms of machine learning and deep learning. Its course of action is as the following: given an input vector (X) and a weight input (W), compute the result (Y). Then update the weight vector according to the result to increase the amount of correctly predicted outputs in the next iteration. This process is repeated a constant number of iterations (N).

In order to implement the Adaline neuron, we implemented the following methods:

- **fit(X ,y)**
  this method purpose is to train the Adaline neuron with the training set. It takes points dataset; it's labels and returns the trained neuron.

- **_update_weights(sample, target)**
  This method is a private method. Its purpose is to update the weights of the neuron, it takes sample (point) and its target, computes the result, update the weights according it and return the cost.

- **_shuffle(X, y)**
  This method purpose is to shuffle the data before as part of the training process. It takes the data set and it's labels and return the shuffled dataset.

- **net_input(X)**
  This method calculates the net input using matrix multiplication.

- **predict(X)**
  This method makes the prediction on the test set. It takes dataset and return ndarray with the predicted value for each sample.

- **score(X, y)**
  This method computes the score of the model. It takes dataset and its labels and return the score.

Linear Separation and Non-Linear Separation

One of the most important conclusions we conclude from this research is the importance of understanding the concepts of linear separation and non-linear separation. As can be seen, in part A the linear separation is obvious by looking at the visualization and, understandably, there is a tradeoff between the size of the dataset and the score- more samples mean a lower score. While in part B the model is non-linear, therefore it uses different techniques. That's the reason the size of the dataset has a minor impact on the score. For example, in part B the neuron has a large margin of error because the class of "1" has a small range. All of this comes with the idea that one layer neuron can only perform binary classification.

Code

Our code includes two scripts: Adaline neuron implementation and main.

Adaline implementation:

```
class Adaline(object):

    def __init__(self, rate=0.0001, n_iter=24):
        self.l_rate = rate
        self.n_iter = n_iter
        self.weights = []
        self.costs = []

    def fit(self, X, y):
        """
        training the data- 1. adding bias. 2. shuffling. 3. training
        :param X: data
        :param y:label
        :return: trained model
        """
        row = X.shape[0]
        col = X.shape[1]
        # bias
        X = self._bias(X, (row, col))
        # weights
        np.random.seed(1)
        self.weights = np.random.rand(col + 1)
        # training
        for iter in range(self.n_iter):
            # shuffling
            X, y = self._shuffle(X, y)
            cost = []
            for sample, label in zip(X, y):
                cost.append(self._update_weights(sample, label))
            # computing the avg cost and adding to cost list
            avg = sum(cost) / len(y)
            self.costs.append(avg)
        return self

    def _update_weights(self, sample, label):
        """
        updating weights. private method
        :param sample:
        :param target:
        :return: cost
        """
        result = self.net_input(sample)
        error = label - result
        self.weights += self.l_rate * sample.dot(error)
        # cost = 0.5 * (error ** 2)
        return (error ** 2) / 2

    def _shuffle(self, X, y):
        """
```

```
        shuffling data
        :param X: data
        :param y: label
        :return: shuffled data
        """
        per = np.random.permutation(len(y))
        return X[per], y[per]

    def net_input(self, X):
        """
        calculating the net input using matrix multiplication
        :param X: data
        :return: net input
        """
        return X @ self.weights

    def predict(self, X):
        """
        predict sample label
        :param X: data
        :return: 1 or -1
        """
        """ if data and weights are not in the same shape
        we should add bias in order to perform matrix multiplication"""
        if len(X.T) != len(self.weights):
            X = self._bias(X, (X.shape[0], X.shape[1]))
        return np.where(self.net_input(X) > 0.0, 1, -1)

    def score(self, X, y):
        """
        computing the score of the net.
        :param X: data
        :param y: label
        :return: score of net
        """
        wrong_predictions = 0
        # count the wrong predictions
        predicted = self.predict(X)
        for pred, real in zip(predicted, y):
            if pred != real:
                wrong_predictions += 1
        # compute the score
        self.score_ = (len(y) - wrong_predictions) / len(y)
        return self.score_

    def _bias(self,X, size):
        """
        adding bias to the data
        :param X: data
        :param size: size of X
        :return: X after adding bias
        """
        bias = np.ones((size[0], size[1] + 1))
        bias[:, 1:] = X
        X = bias
        return X
```

main script:

```
def create_data(part, is_fit):
    X = np.empty((SIZE, 2), dtype=object)
    if is_fit:
        random.seed(10)
    y = np.zeros(SIZE)
    if part != "A" and part != "B":
        print("invalid input, enter A or B")
        return
    if part == "A":
        for i in range(SIZE):
            X[i, 0] = (random.randint(MIN, MAX) / 100)  # x
            X[i, 1] = (random.randint(MIN, MAX) / 100)  # y
        for i in range(SIZE):
            y[i] = 1 if X[i][1] > 1 else -1
    if part == "B":
        """manually add points to the dataset in order to get label 1,
then fill the other part randomly"""
        for i in range(SIZE):
            X[i, 0] = (random.randint(MIN, MAX) / 1000)  # x
            X[i, 1] = (random.randint(MIN, MAX) / 1000)  # y
        for i in range(SIZE):
            y[i] = 1 if 4 <= (X[i][1] ** 2 + X[i][0] ** 2) <= 9 else -1
    return X, y
```

```
def main_a():
    "Part A"
    #create data
    X, y = create_data("A", True)
    X1, y1 = create_data("A", False)
    X2, y2 = create_data("A", False)

    # data distribution
    sns.distplot(X, fit=sp.stats.norm, kde=False, label="x",
color="red")
    sns.distplot(X1, fit=sp.stats.norm, kde=False, label="x1",
color="green")
    sns.distplot(X2, fit=sp.stats.norm, kde=False, label="x2",
color="blue")

    # fit and predict
    adaline = Adaline().fit(X, y)
    pred1 = adaline.predict(X1)
    pred2 = adaline.predict(X2)

    # classification report
    print(classification_report(y1, pred1))
    print(classification_report(y2, pred2))

    # confusion matrix
    cm = confusion_matrix(pred1, y1)
    plt.subplots()
    sns.heatmap(cm, fmt=".0f", annot=True)
```

```
    plt.title("confusion matrix")
    plt.xlabel("Test")
    plt.ylabel("Predict")

    cm = confusion_matrix(pred2, y2)
    plt.subplots()
    sns.heatmap(cm, fmt=".0f", annot=True)
    plt.title("confusion matrix")
    plt.xlabel("Test")
    plt.ylabel("Predict")

    plt.show()

    # linear separation
    x_min = X[:, 0].min() - 1
    x_max = X[:, 0].max() + 1
    y_min = X[:, 1].min() - 1
    y_max = X[:, 1].max() + 1
    xx1, xx2 = np.meshgrid(np.arange(x_min, x_max, 0.1),
np.arange(y_min, y_max, 0.1), )

    pred = adaline.predict(np.array([xx1.flatten(), xx2.flatten()]).T)
    pred = pred.reshape(xx1.shape)
    colors = ListedColormap(('red', 'blue'))
    # background colors --> showed our prediction
    plt.contourf(xx1, xx2, pred, alpha=0.4, )
    plt.xlim(xx1.min(), xx1.max())
    plt.ylim(xx2.min(), xx2.max())

    plt.scatter(X[:, 0], X[:, 1], marker=".", c=y * 2 - 1, s=50,
cmap=colors)

    plt.show()

    x_min = X1[:, 0].min() - 1
    x_max = X1[:, 0].max() + 1
    y_min = X1[:, 1].min() - 1
    y_max = X1[:, 1].max() + 1
    xx1, xx2 = np.meshgrid(np.arange(x_min, x_max, 0.1),
np.arange(y_min, y_max, 0.1), )

    pred = adaline.predict(np.array([xx1.flatten(), xx2.flatten()]).T)
    pred = pred.reshape(xx1.shape)
    colors = ListedColormap(('red', 'blue'))
    plt.contourf(xx1, xx2, pred, alpha=0.4, )
    plt.xlim(xx1.min(), xx1.max())
    plt.ylim(xx2.min(), xx2.max())

    plt.scatter(X1[:, 0], X1[:, 1], marker=".", c=y1 * 2 - 1, s=50,
cmap=colors)

    plt.show()

    x_min = X2[:, 0].min() - 1
    x_max = X2[:, 0].max() + 1
    y_min = X2[:, 1].min() - 1
    y_max = X2[:, 1].max() + 1
```

```
    xx1, xx2 = np.meshgrid(np.arange(x_min, x_max, 0.1),
np.arange(y_min, y_max, 0.1), )

    pred = adaline.predict(np.array([xx1.flatten(), xx2.flatten()]).T)
    pred = pred.reshape(xx1.shape)
    colors = ListedColormap(('red', 'blue'))
    # background colors --> showed our prediction
    plt.contourf(xx1, xx2, pred, alpha=0.4, )
    plt.xlim(xx1.min(), xx1.max())
    plt.ylim(xx2.min(), xx2.max())

    plt.scatter(X2[:, 0], X2[:, 1], marker=".", c=y2 * 2 - 1, s=50,
cmap=colors)

    plt.show()

    # error-iteration
    fig, ax = plt.subplots(figsize=(12, 5))
    ax.plot(range(1, len(adaline.costs) + 1), adaline.costs,
marker='o')
    ax.set_xlabel('Iterations')
    ax.set_ylabel('Error')
    ax.set_title('error change per iteration')
    plt.show()
```

```
def main_b():
    "Part B"

    # create data
    X, y = create_data("B", True)
    X1, y1 = create_data("B", False)
    X2, y2 = create_data("B", False)

    # data distribution
    sns.distplot(X, fit=sp.stats.norm, kde=False, label="x",
color="red")
    sns.distplot(X1, fit=sp.stats.norm, kde=False, label="x1",
color="green")
    sns.distplot(X2, fit=sp.stats.norm, kde=False, label="x2",
color="blue")
    plt.show()

    # fit and predict
    adaline = Adaline().fit(X, y)
    pred1 = adaline.predict(X1)

    # classification report
    print(classification_report(y1, pred1))

    # confusion matrix
    cm = confusion_matrix(pred1, y1)
    plt.subplots()
    sns.heatmap(cm, fmt=".0f", annot=True)
    plt.title("confusion matrix")
    plt.xlabel("Test")
    plt.ylabel("Predict")
```

```
    plt.show()

    """non-linear separation
        NOTE: SIZE var should be changed to 10,000 when required to
execute with 10,000 samples"""
    x_min = X1[:, 0].min() - 1
    x_max = X1[:, 0].max() + 1
    y_min = X1[:, 1].min() - 1
    y_max = X1[:, 1].max() + 1
    xx1, xx2 = np.meshgrid(np.arange(x_min, x_max, 0.1),
np.arange(y_min, y_max, 0.1), )

    pred = adaline.predict(np.array([xx1.flatten(), xx2.flatten()]).T)
    pred = pred.reshape(xx1.shape)
    colors = ListedColormap(('yellow', 'black'))
    plt.contourf(xx1, xx2, pred, alpha=0.4, )
    plt.xlim(xx1.min(), xx1.max())
    plt.ylim(xx2.min(), xx2.max())

    plt.scatter(X1[:, 0], X1[:, 1], marker=".", c=y1 * 2 - 1, s=50,
cmap=colors)
    plt.legend(loc='upper left')
    plt.show()

    print(accuracy_score(y_true=y1, y_pred=pred1))
```