Part AB

Matan Yarin Shimon: 314669342

Or Yitshak: 208936039

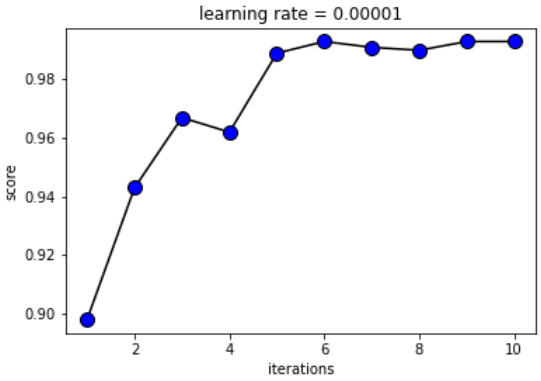
Netanel Levine: 312512619

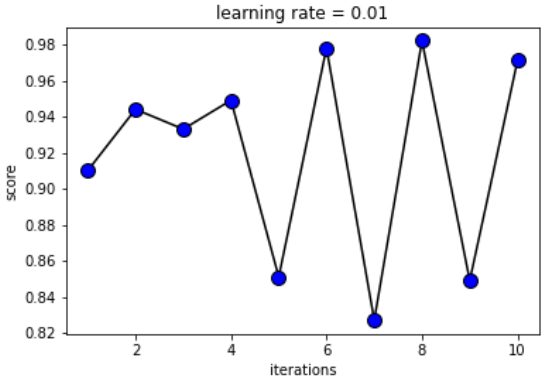
Yahalom Chasid: 208515577

Part A

in this part, we will check how our model performs with a training set in the size of 1000 and a test set in the size of 1000 such that each point in the set gets a value=1 if its y value is greater than 1 and -1 otherwise.

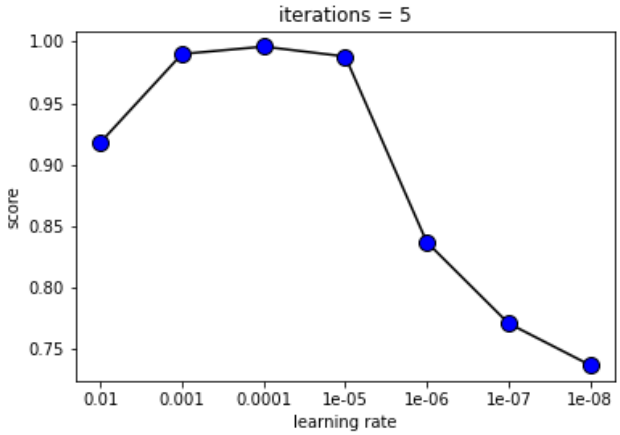
At first, we would like to observe how the 'learning rate' variable and the number of iterations will affect the performance of the algorithm.





From the graphs above we can learn that if the learning rate is too big it will cause big changes in each iteration and will impact the results of the model.

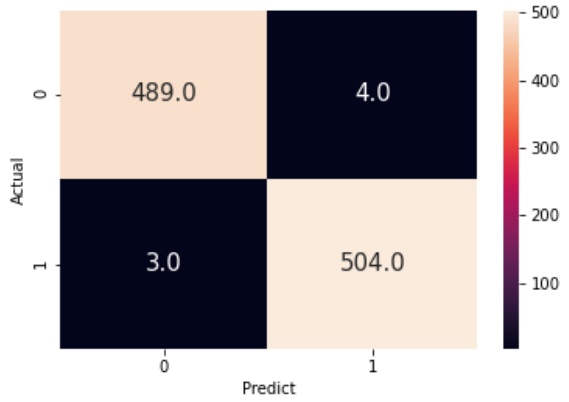
Furthermore, by observing the graph below we can derive that too small a learning rate will need a lot of iterations and won't be good.

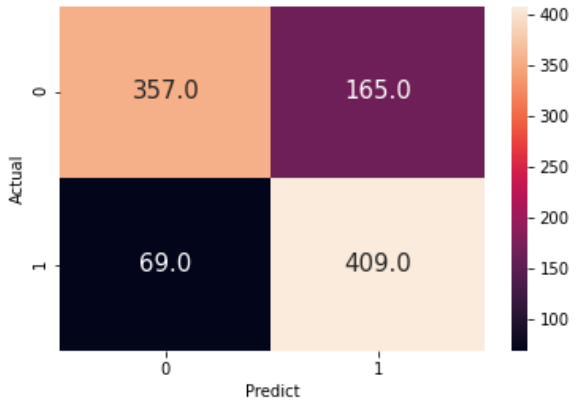


In conclusion, the learning rate should be 0.0001.

Now that we found the right parameters, we will analyze the results of the tuned model.

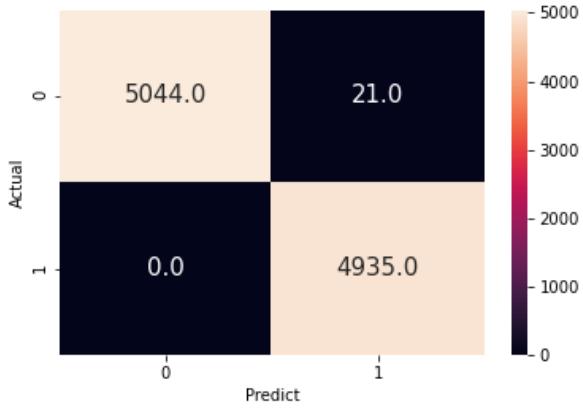
The score (accuracy) of the model Is 99.34 when the training set is balanced and with an extremely unbalanced model the score is 77.85. its confusion matrix is as follows (left - balanced, right – unbalanced):

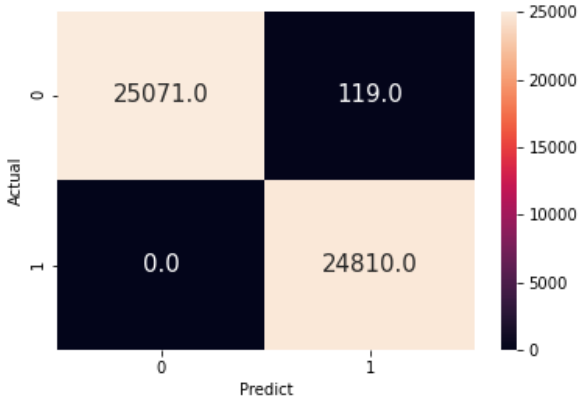




We can see that the score is dramatically affected by the training set.

The model keeps its results also with a bigger set of 10,000 and 50,000.





To summarize, the model performance does not affect by the size of the training set as long its big enough and contains balanced information such that both classes have enough instances.

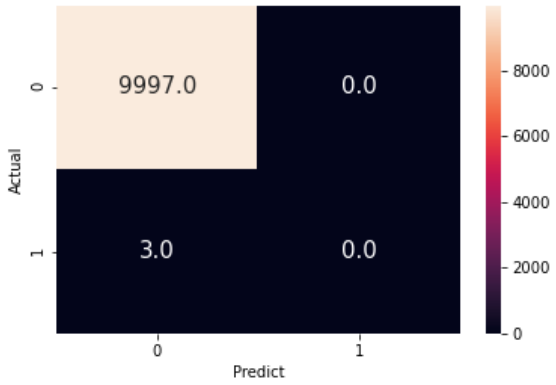
Part B

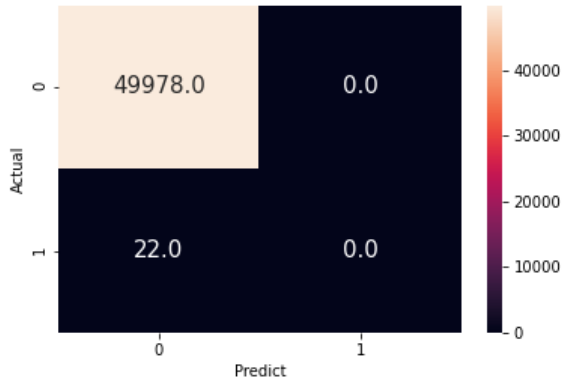
in this part, we will check how our model performs with a training set in the size of 1000 and a test set in the size of 1000 such that each point in the set gets a value of 1 if 4 <= x^2 + y^2 <= 9.

In this part the condition to get value=1 is hard, and we can see that with a set of 1000 instances there is not even one point that gets 1 and the classification is 100% accurate because of it.

But with a bigger set in a size of 10000, we get a few instances with value 1 but because the model didn't have enough instances, he didn't learn how to properly classify them, so it continues to classify them as -1.

The results remain the same even with a bigger set (50000), the model keeps classifying all the points as -1 due to the lack of instances.

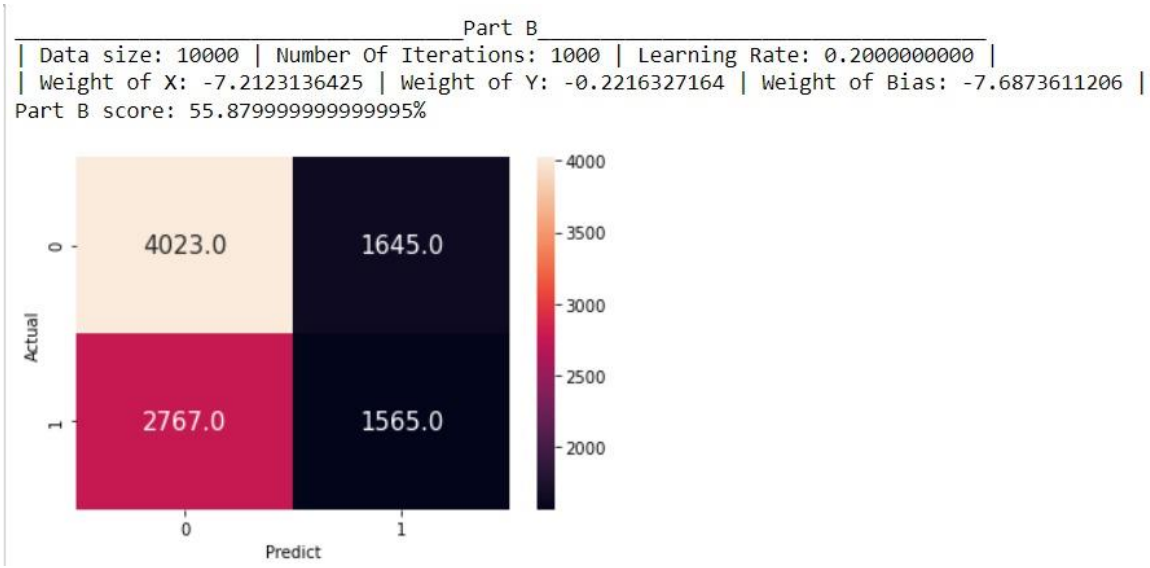




To summarize in this part the algorithm has more trouble classifying the data it may have 99.99% of success but even a dummy classifier that classifies every instance as -1 no matter what will have the same accuracy.

to try to fix the data distribution we gave our model approximately half instances from each type.

In order to get better results we also increased the learning rate:



We can see now that the distribution is more balanced than the model trying to classify more instances as class 1. We assume that the bad accuracy caused by that the condition to be in class 1 is more complicated than in the other class.

The classes:

Point – a simple class that represents a 2D point in the plane.

Adaline – this class contains the implementation of the Adaline algorithm.

It contains the following function:

- Fit – this function receives as input the features of the training set, the actual classes of the training set, the number of iterations, and the alpha parameter and performs the learning progress of the model.
- Predictions – this function receives as input the features of the test set and returns its predictions based on the 'fit' state as output.
- Score – this function receives as input the predictions of the model and the actual classes of the results and calculates the accuracy.

The code:

```
class Point:
    def __init__(self, x, y):
        self._x = x
        self._y = y
        self._value = 1 if y > 1 else -1

    def getX(self):
        return self._x

    def getY(self):
        return self._y

    def getValue(self):
        return self._value

    def setX(self, x):
        self._x = x

    def setY(self, y):
        self._y = y

    def setValue(self, value):
        self._value = value
```

```
import numpy as np
import random


class Adaline:
    def __init__(self):
        self._num_of_iterations = 10
        self._alpha = 0.1
        self._weights = []
        self._error = []

    def getAlpha(self):
        return self._alpha

    def getNumOfIterations(self):
        return self._num_of_iterations

    def getWeights(self):
        return self._weights

    def getErrors(self):
        return self._error

    def setAlpha(self, alpha):
        self._alpha = alpha

    def setNumOfIterations(self, num_of_iterations):
        self._num_of_iterations = num_of_iterations

    def setWeights(self, weights):
        self._weights = weights

    def fit(self, X, Y, num_of_iterations=10, alpha=0.001):
        self.setNumOfIterations(num_of_iterations)
        self.setAlpha(alpha)
        self._weights = [0.5, 0.5]
        bias = random.uniform(0.005, 1)
        bias = 0.2
        train_length = len(X)
        for j in range(num_of_iterations):
            E = 0
            for i in range(train_length):
                y_in = (bias + self._weights[0] * X[i].getX() +
self._weights[1] * X[i].getY()) / 100
                diff = Y[i] - y_in
                if diff != 0:
                    self._weights[0] += alpha * diff * X[i].getX()
                    self._weights[1] += alpha * diff * X[i].getY()
                    bias += alpha * diff
                    E += pow(diff, 2)
            MSE = E / train_length
            if MSE < 0.000000000000001:
                print(j, f'last MSE: {MSE:.14f}')
                break
            # print(f'{MSE:.14f}')
        self._weights.append(bias)
```

```
    def predict(self, X_test):
        # print(self._weights)
        predictions = []
        for i in range(len(X_test)):
            res = self._weights[0] * X_test[i].getX() +
self._weights[1] * X_test[i].getY() + self._weights[2]
            ans = 1 if 4 <= (pow(X_test[i].getY(), 2) +
pow(X_test[i].getX(), 2)) <= 9 else -1
            if res > 0.0:
                predictions.append(1)
            else:
                predictions.append(-1)
        return predictions


    def score(self, predictions, answers):
        correct = sum(predictions[i] == answers[i] for i in
range(len(predictions)))
        return correct / len(predictions)
```

```
import random
import numpy as np
from Adaline import Adaline
from Point import Point
import warnings
warnings.filterwarnings("ignore")


def create_random_points():
    return round(random.randint(-10000, 10000) / 100, 15)


def create_X(num):
    pass


def main():
    data_size = 1000
    num_of_iterations = 1000
    learning_rate = 0.001

    # Part A:
    X_train = []
    X_test = []
    for _ in range(data_size):
        X_train.append(Point(create_random_points(),
create_random_points()))
        X_test.append(Point(create_random_points(),
create_random_points()))
    adalineA = Adaline()
    Y_train = [(1 if point.getY() > 1 else -1) for point in X_train]
    adalineA.fit(X_train, Y_train, num_of_iterations, learning_rate)
    Y_test = [(1 if point.getY() > 1 else -1) for point in X_test]
    predictions = adalineA.predict(X_test)
```

```
    score = adalineA.score(predictions, Y_test)
    # print("res: ", res, "real value: ", test[i].value)
    print("_____Part
A_____")
    print(f'| Data size: {data_size} | Number Of Iterations:
{num_of_iterations} | Learning Rate: {learning_rate:.10f} |')
    print(
        f'| Weight of X: {adalineA.getWeights()[0]:.10f} | Weight of Y:
{adalineA.getWeights()[1]:.10f} | Weight of Bias:
{adalineA.getWeights()[2]:.10f} |')
    print(f'Part A score: {score * 100}%')

    print()

print("_____
_____")

print("_____
_____")

print("_____
_____")
    print()

    # Part B:
    X_train = []
    X_test = []
    for _ in range(data_size):
        X_train.append(Point(create_random_points(),
create_random_points()))
        X_test.append(Point(create_random_points(),
create_random_points()))
    adalineB = Adaline()
    Y_train = [(1 if 4 <= (pow(point.getY(), 2) + pow(point.getX(), 2))
<= 9 else -1) for point in X_train]
    adalineB.fit(X_train, Y_train, num_of_iterations, learning_rate)
    Y_test = [(1 if 4 <= (pow(point.getY(), 2) + pow(point.getX(), 2))
<= 9 else -1) for point in X_test]
    predictions = adalineB.predict(X_test)
    score = adalineB.score(predictions, Y_test)
    # print("res: ", res, "real value: ", test[i].value)
    print("_____Part
B_____")
    print(f'| Data size: {data_size} | Number Of Iterations:
{num_of_iterations} | Learning Rate: {learning_rate:.10f} |')
    print(
        f'| Weight of X: {adalineB.getWeights()[0]:.10f} | Weight of Y:
{adalineB.getWeights()[1]:.10f} | Weight of Bias:
{adalineB.getWeights()[2]:.10f} |')
    print(f'Part B score: {score * 100}%')


if __name__ == '__main__':
    main()
```