Implementing the Perceptron algorithm for Finding the weights of a Linear Discriminant function

MD. Saimom Islam
16.02.04.011

January 29, 2021

1 Abstract

Perceptron is an algorithm on linear data points for supervised learning. For non linear data we have to convert it into linear,where the data points are taken from a lower dimension to a higher dimension. With the help of this algorithm we can find the perfect weights of a Linear Discriminant function.

2 Introduction

In perception algorithm our main task is to find the best fitting weights for the data points. At first, the data points must be converted into higher dimension because in lower dimension points are not linearly separable. There are two process of weight updating. One is single update and another is batch update. The clue we are using for updating is:

$$w(i+1) = w(i) + \eta \Sigma y \quad \text{if } w^T(i)y <= 0$$

$$\text{(i:e:; if } y_m \text{ is misclassified)}$$

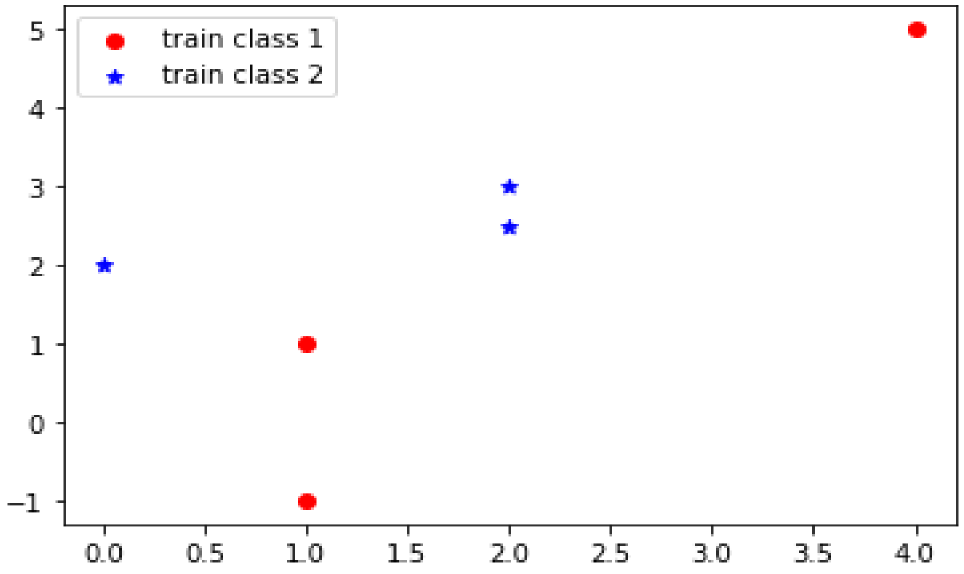
$$= w(i) \text{ ; if } w^T(i)y_m > 0$$

Here η is the learning rate.

3 Implementation

A dataset named ’train_perception.txt’ is given

1. Firstly all sample points from train data have to be plotted from both classes, but samples from the same class should have the same color and marker. For this two numpy arrays was taken to store the classified data according to their class level and plotted them with different marker.



   After plotting the points it is seen that the two classes from train data can not be separated with a linear boundary.

2. Secondly the data is converted into a higher dimension for being linearly separable using the following formula:

$$y = [\, x^2_1 \;\; x^2_2 \;\; x_1 \;\; x_2 \;\; x_1 \;\; x_2 \;\; 1 \,]$$

   Then the class 2 was normalized by negating the data points.

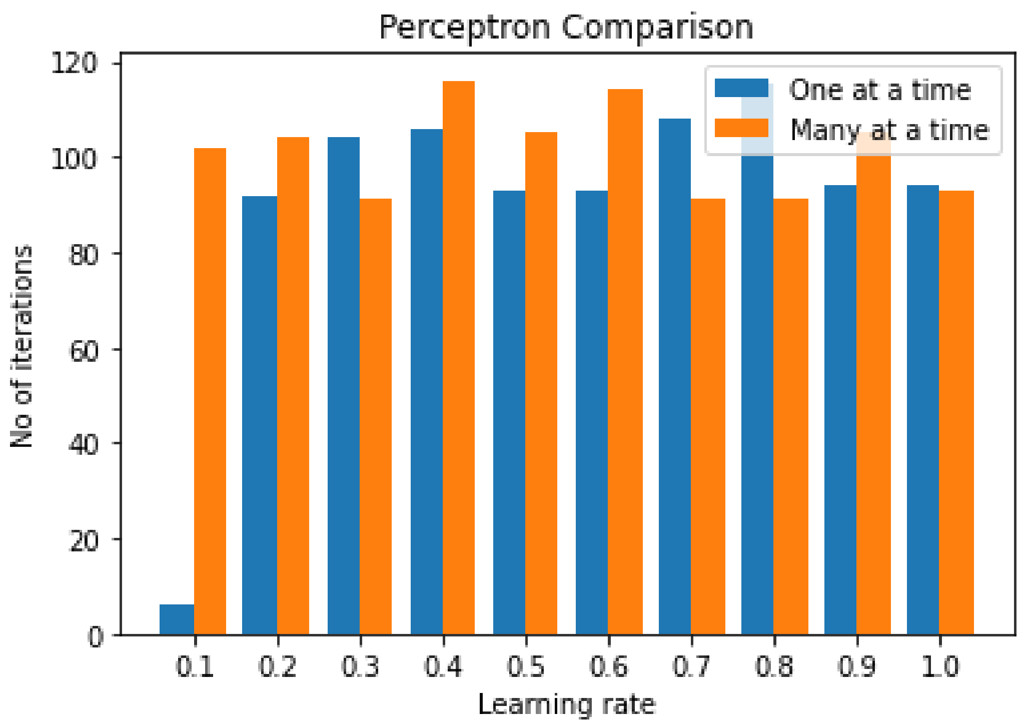
3. Then I have calculated the weights for both single update and batch update for different learning rate between 0.1 and 1.weights were initially zero, random value and also all one.

4. Now after calculating the iterations for both cases we have found three table with weight initiated with all zero's, all one's and randomly. tables and outputs are given below:

   Sample Output (Initial Weight Vector All One):

Alpha (Learning Rate)	One at a time	Many at a time
0.1	6	102
0.2	92	104
0.3	104	91
0.4	106	116
0.5	93	105
0.6	93	114
0.7	108	91
0.8	115	91
0.9	94	105
1.0	94	93

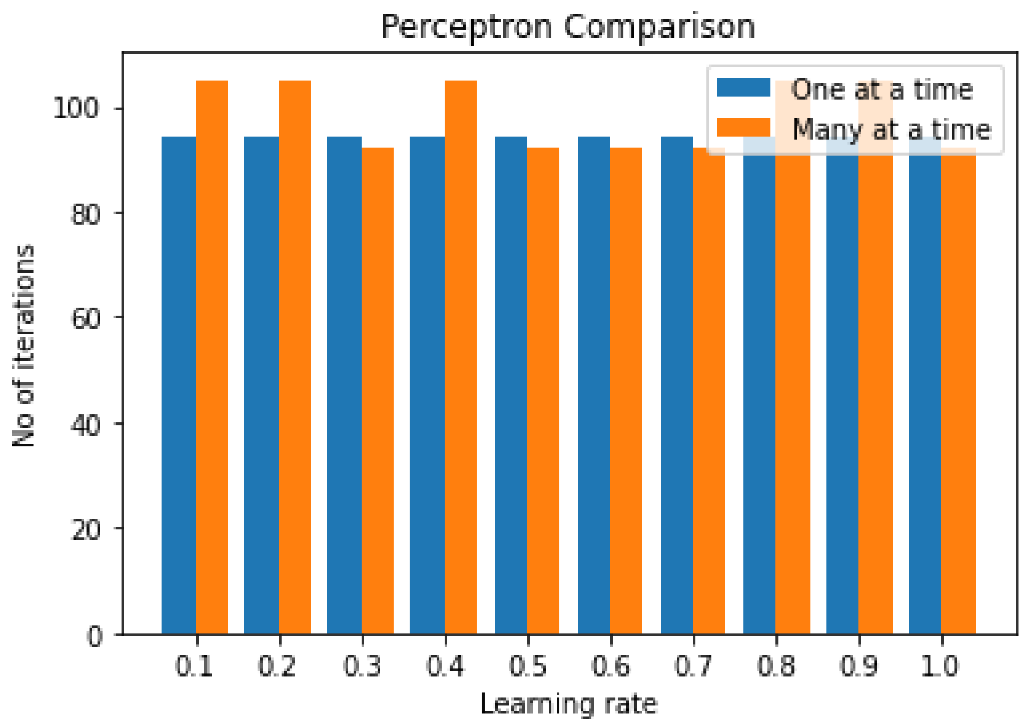
   Bar chart (Initial Weight Vector All One):



Sample Output (Initial Weight Vector All Zero):

Alpha (Learning Rate)	One at a time	Many at a time
0.1	94	105
0.2	94	105
0.3	94	92
0.4	94	105
0.5	94	92
0.6	94	92
0.7	94	92
0.8	94	105
0.9	94	105
1.0	94	92

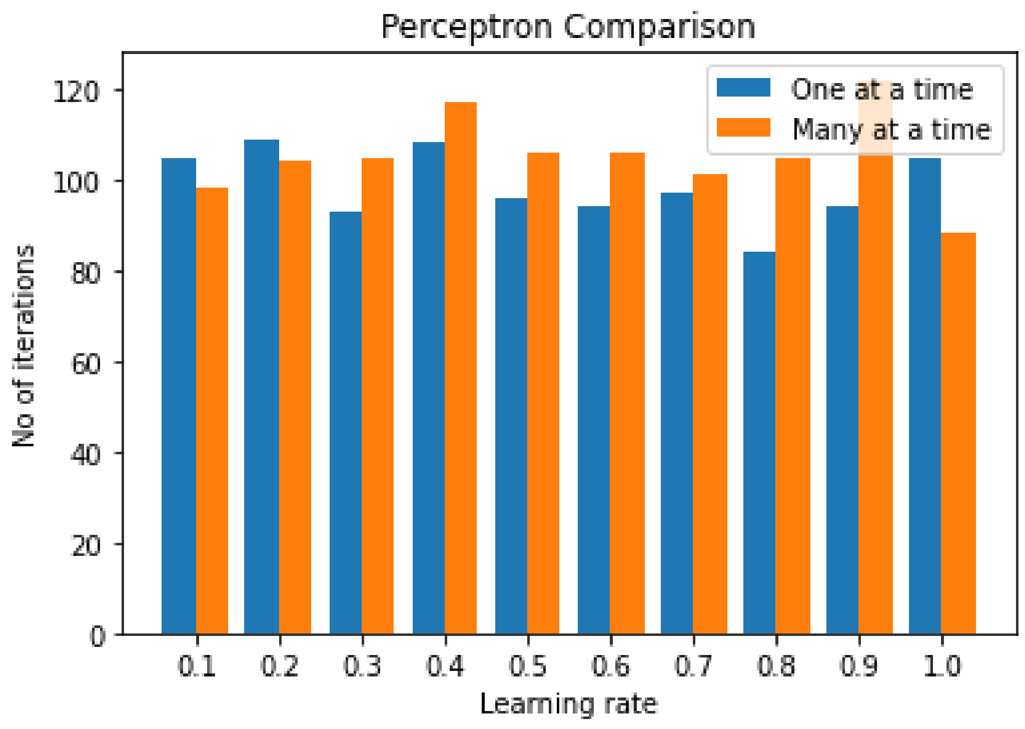
Bar chart (Initial Weight Vector All Zero):



Sample Output (Randomly initialized weight with seed 0):

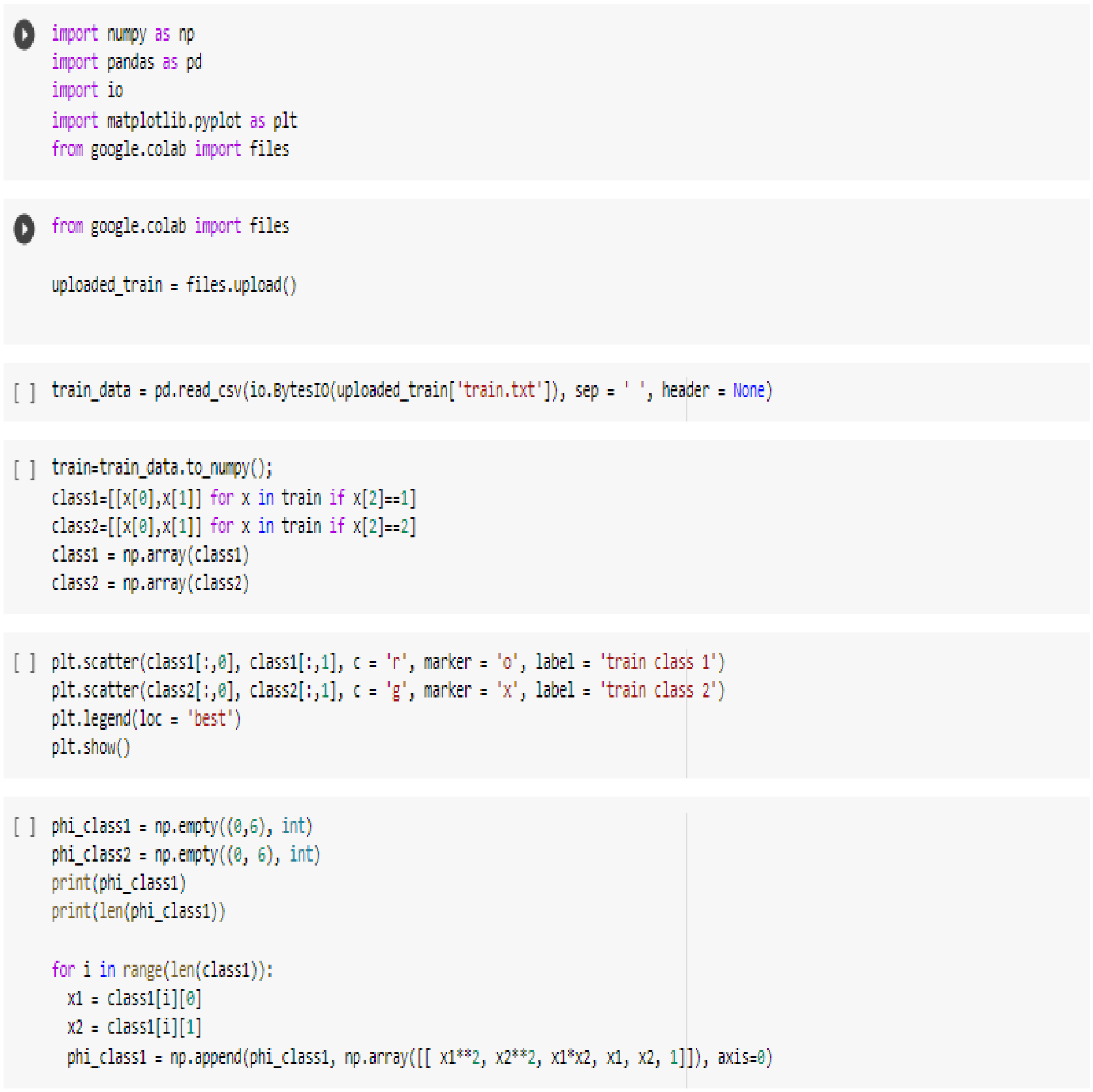
Alpha (Learning Rate)	One at a time	Many at a time
0.1	105	98
0.2	109	104
0.3	93	105
0.4	108	117
0.5	96	106
0.6	94	106
0.7	97	101
0.8	84	105
0.9	94	122
1.0	105	88

Bar chart (Randomly initialized weight with seed 0):



4 Main Code

The Main Code is given below.



```python
import numpy as np
import pandas as pd
import io
import matplotlib.pyplot as plt
from google.colab import files
```

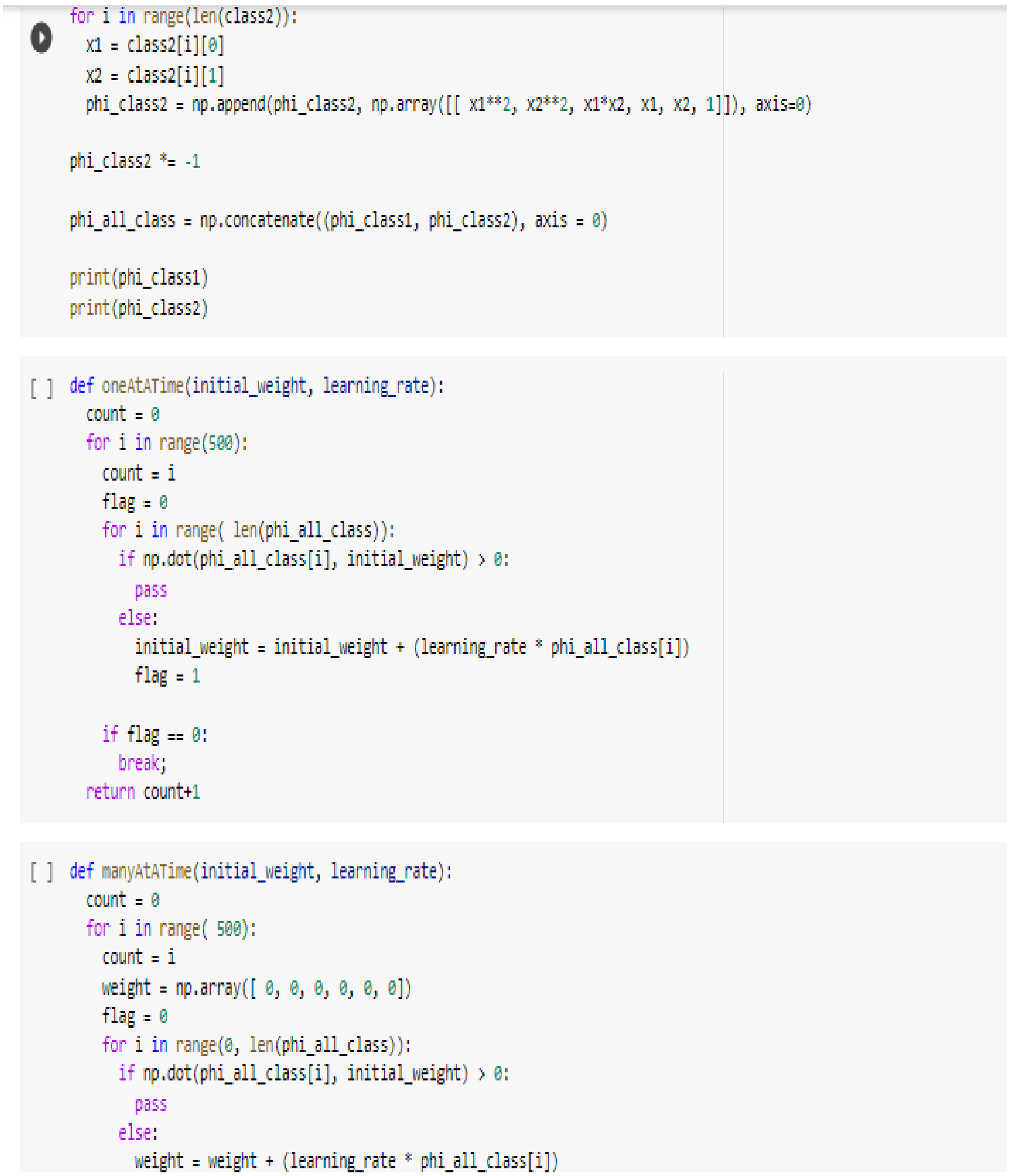
```python
from google.colab import files

uploaded_train = files.upload()
```

```python
train_data = pd.read_csv(io.BytesIO(uploaded_train['train.txt']), sep = ' ', header = None)
```

```python
train=train_data.to_numpy();
class1=[[x[0],x[1]] for x in train if x[2]==1]
class2=[[x[0],x[1]] for x in train if x[2]==2]
class1 = np.array(class1)
class2 = np.array(class2)
```

```python
plt.scatter(class1[:,0], class1[:,1], c = 'r', marker = 'o', label = 'train class 1')
plt.scatter(class2[:,0], class2[:,1], c = 'g', marker = 'x', label = 'train class 2')
plt.legend(loc = 'best')
plt.show()
```

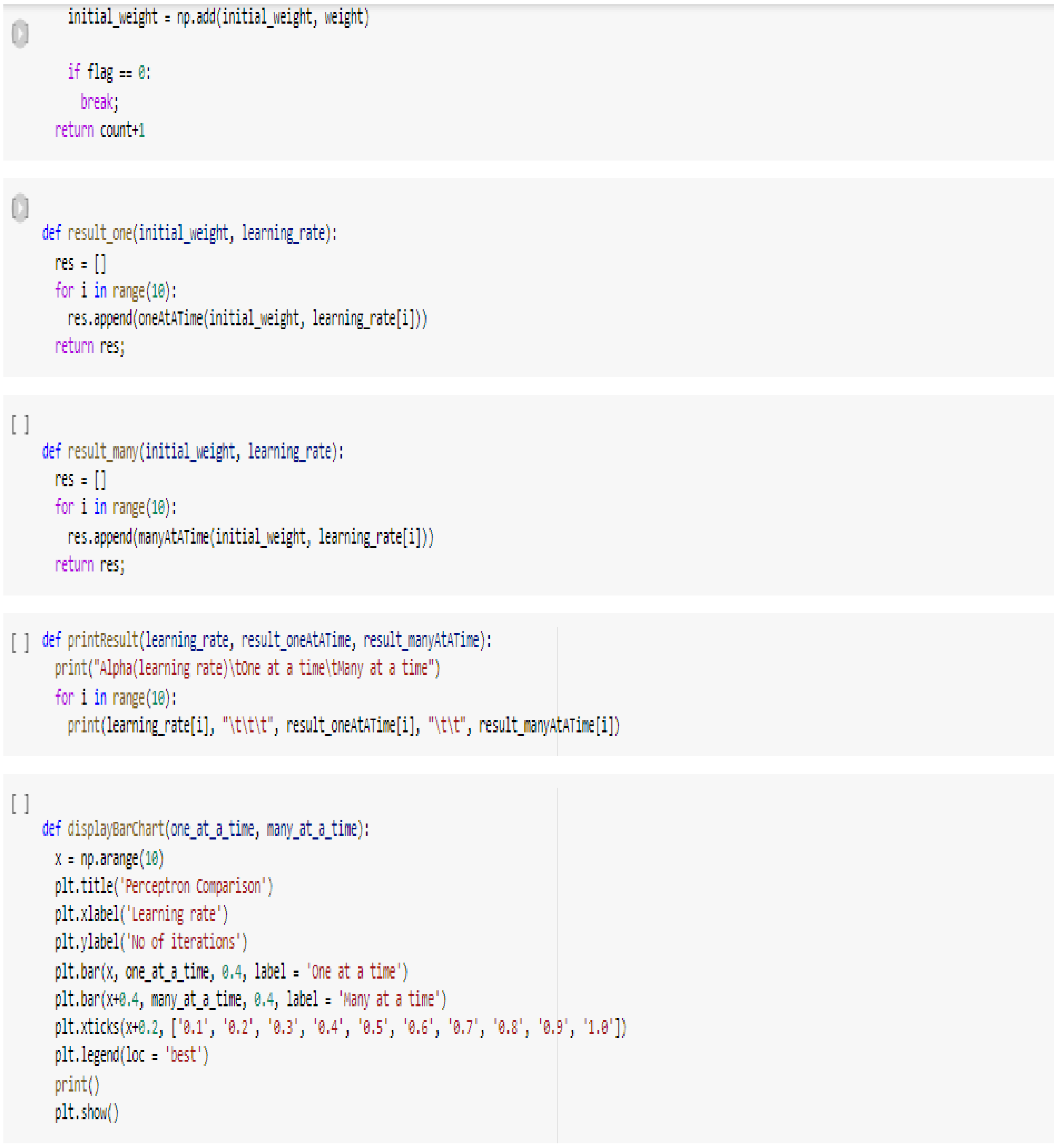
```python
phi_class1 = np.empty((0,6), int)
phi_class2 = np.empty((0, 6), int)
print(phi_class1)
print(len(phi_class1))

for i in range(len(class1)):
  x1 = class1[i][0]
  x2 = class1[i][1]
  phi_class1 = np.append(phi_class1, np.array([[ x1**2, x2**2, x1*x2, x1, x2, 1]]), axis=0)
```



```
for i in range(len(class2)):
  x1 = class2[i][0]
  x2 = class2[i][1]
  phi_class2 = np.append(phi_class2, np.array([[ x1**2, x2**2, x1*x2, x1, x2, 1]]), axis=0)

phi_class2 *= -1

phi_all_class = np.concatenate((phi_class1, phi_class2), axis = 0)

print(phi_class1)
print(phi_class2)
```

```
[ ] def oneAtATime(initial_weight, learning_rate):
      count = 0
      for i in range(500):
        count = i
        flag = 0
        for i in range( len(phi_all_class)):
          if np.dot(phi_all_class[i], initial_weight) > 0:
            pass
          else:
            initial_weight = initial_weight + (learning_rate * phi_all_class[i])
            flag = 1

        if flag == 0:
          break;
      return count+1
```

```
[ ] def manyAtATime(initial_weight, learning_rate):
      count = 0
      for i in range( 500):
        count = i
        weight = np.array([ 0, 0, 0, 0, 0, 0])
        flag = 0
        for i in range(0, len(phi_all_class)):
          if np.dot(phi_all_class[i], initial_weight) > 0:
            pass
          else:
            weight = weight + (learning_rate * phi_all_class[i])
```



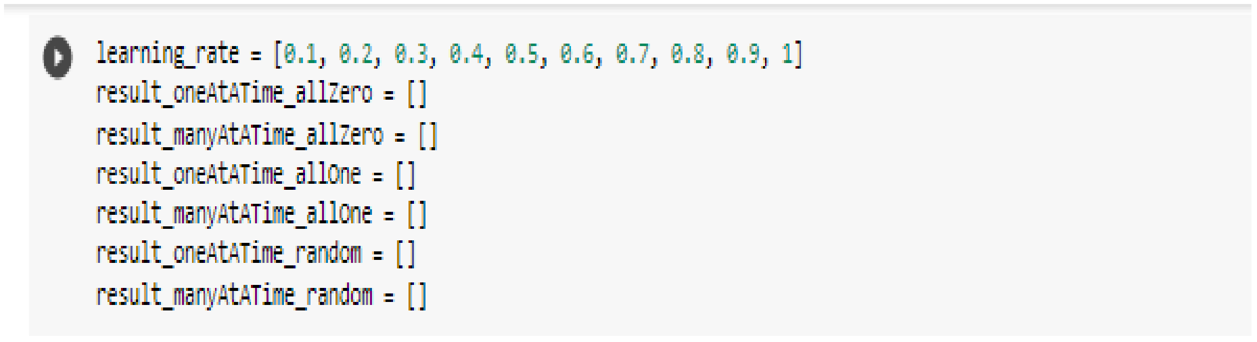
```
      initial_weight = np.add(initial_weight, weight)

      if flag == 0:
        break;
    return count+1
```

```
def result_one(initial_weight, learning_rate):
  res = []
  for i in range(10):
    res.append(oneAtATime(initial_weight, learning_rate[i]))
  return res;
```

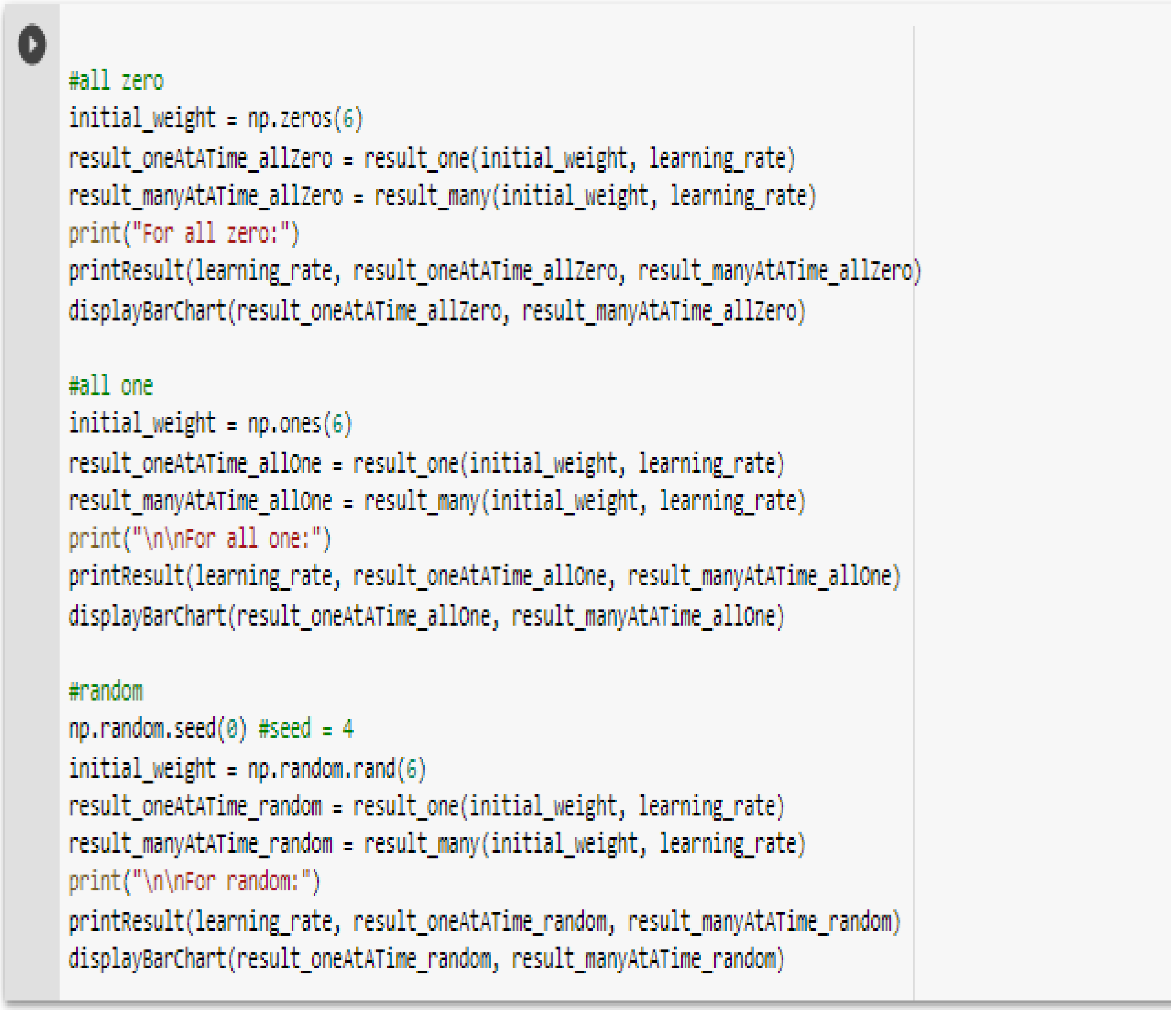
[ ]
```
def result_many(initial_weight, learning_rate):
  res = []
  for i in range(10):
    res.append(manyAtATime(initial_weight, learning_rate[i]))
  return res;
```

[ ]
```
def printResult(learning_rate, result_oneAtATime, result_manyAtATime):
  print("Alpha(learning rate)\tOne at a time\tMany at a time")
  for i in range(10):
    print(learning_rate[i], "\t\t\t", result_oneAtATime[i], "\t\t", result_manyAtATime[i])
```

[ ]
```
def displayBarChart(one_at_a_time, many_at_a_time):
  x = np.arange(10)
  plt.title('Perceptron Comparison')
  plt.xlabel('Learning rate')
  plt.ylabel('No of iterations')
  plt.bar(x, one_at_a_time, 0.4, label = 'One at a time')
  plt.bar(x+0.4, many_at_a_time, 0.4, label = 'Many at a time')
  plt.xticks(x+0.2, ['0.1', '0.2', '0.3', '0.4', '0.5', '0.6', '0.7', '0.8', '0.9', '1.0'])
  plt.legend(loc = 'best')
  print()
  plt.show()
```



```
learning_rate = [0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1]
result_oneAtATime_allZero = []
result_manyAtATime_allZero = []
result_oneAtATime_allOne = []
result_manyAtATime_allOne = []
result_oneAtATime_random = []
result_manyAtATime_random = []
```



```
#all zero
initial_weight = np.zeros(6)
result_oneAtATime_allZero = result_one(initial_weight, learning_rate)
result_manyAtATime_allZero = result_many(initial_weight, learning_rate)
print("For all zero:")
printResult(learning_rate, result_oneAtATime_allZero, result_manyAtATime_allZero)
displayBarChart(result_oneAtATime_allZero, result_manyAtATime_allZero)

#all one
initial_weight = np.ones(6)
result_oneAtATime_allOne = result_one(initial_weight, learning_rate)
result_manyAtATime_allOne = result_many(initial_weight, learning_rate)
print("\n\nFor all one:")
printResult(learning_rate, result_oneAtATime_allOne, result_manyAtATime_allOne)
displayBarChart(result_oneAtATime_allOne, result_manyAtATime_allOne)

#random
np.random.seed(0) #seed = 4
initial_weight = np.random.rand(6)
result_oneAtATime_random = result_one(initial_weight, learning_rate)
result_manyAtATime_random = result_many(initial_weight, learning_rate)
print("\n\nFor random:")
printResult(learning_rate, result_oneAtATime_random, result_manyAtATime_random)
displayBarChart(result_oneAtATime_random, result_manyAtATime_random)
```

6 Question Answering

1. Perception Algorithm is for linear data.for non linear data points it does not give the right output.if I convert it into higher dimension it becomes linear from non linear.and the two class can be saperated easily.Thats why we take it in higher dimension.

2. In section 4.4 there is table where for each initial weight cases and learning rate how many update needs before converging is shown.bar chart is also shown in that section.

7 Conclusion

Perceptron can classify different classes using a linear discriminant function. Here we have learned to implement perceptron after taking the data to a higher dimension where we can separate the classes using a linear line. This is a basic classifier but can separate the data points perfectly using not more than 120 iterations. So we can say that it is much faster for discriminating different classes