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

Author: Humaira Zahin Mauni
ID: 16.01.04.012

Abstract
The goal is to implement the Perceptron algorithm in order to find the weights of a linear discriminant function. The data should be checked for linear seperability - and if it is found to be inseparable, should be mapped to a higher dimension using the kernel trick.

Keywords
Linear Classification, Perceptron algorithm, kernel trick, normalization, Linear Discriminant Function

Introduction
The minimum distance to class mean classifier is used to classify unknown data to classes that minimize the distance between the data and the class in the feature space. The Perceptron is an algorithm for supervised learning in binary classification. They consist of functions that can decide whether an input belongs to one class or the other.

Experimental Design

1. Take input from "train.txt" file. Plot all sample points from both classes, but samples from the same class should have the same color and marker. Observe if these two classes can be separated with a linear boundary.
2. Consider the case of a second order polynomial discriminant function. Generate the high dimensional sample points y, as discussed in the class. We shall use the following formula:

   $$y = [x_1^2 \quad x_2^2 \quad x_1 * x_2 \quad x_1 \quad x_2 \quad 1]$$

   Also, normalize any one of the two classes.

3. Use Perceptron Algorithm (both one at a time and many at a time) for finding the weight- coefficients of the discriminant function (i.e., values of w) boundary for your linear classifier in task 2.

   Here α is the learning rate and $0 < \alpha \leq 1$.

   $$\underline{w}(i+1) = \underline{w}(i) + \alpha \underline{\tilde{y}}_m^{(k)} \quad \text{if } \underline{w}^T(i)\, \underline{\tilde{y}}_m^{(k)} \leq 0$$

   (i.e., if $\underline{\tilde{y}}_m^{(k)}$ is misclassified)

   $$= \underline{w}(i) \quad \text{if } \underline{w}^T(i)\, \underline{\tilde{y}}_m^{(k)} > 0$$

4. Three initial weights have to be used (all one, all zero, randomly initialized with seed fixed). For all of these three cases vary the learning rate between 0.1 and 1 with step size 0.1. Create a table which should contain your learning rate, number of iterations for one at a time and batch Perceptron for all of the three initial weights. You also have to create a bar chart visualizing your table data.

Algorithm Implementation

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

## Q1
uploaded = files.upload() #upload train.txt

df_train = pd.read_csv(io.BytesIO(uploaded['train.txt'] ), sep=" " ,  header = None, d
type = 'float64')
df_train.columns = ['x1', 'x2', 'label']
print(df_train)

groups = df_train.groupby('label')

fig, ax = plt.subplots()
for name, group in groups:
    name = 'Train Class ' + str(name)
    ax.plot(group.x1, group.x2, marker='o', linestyle='', ms=12, label=name)
ax.legend()

plt.show()

#Q2

data = np.zeros(shape=(6,6))
for i in range(6):
    data[i][0] = df_train.x1[i]**2
    data[i][1] = df_train.x2[i]**2
    data[i][2] = df_train.x1[i]*df_train.x2[i]
    data[i][3] = df_train.x1[i]
    data[i][4] = df_train.x2[i]
    data[i][5] = 1
print(data)
for i in range(6):
  if df_train.label[i] == 2.0:
```

```
    data[i] = -1 * data[i]
print(data)

# Q3 and Q4
def one_update(learning_rate, activation, weight, data):
  counter = 0
  while counter < 200:
    counter += 1
    #print("\ncounter: ", counter)
    flag = 0
    for i in range(6):
      activation = np.dot(data[i], np.transpose(weight))
      if activation <= 0:
        weight[0] = weight[0] + learning_rate*data[i]
      if activation > 0:
        flag = flag + 1
      #print("i: ", i)
      #print("activation: ", activation)
    if (flag==6): break
  #print("\n\nANSWER:")
  #print("flag : ", flag)
  #print("weight: ", weight)
  #print("counter: ", counter)
  return weight, counter

def batch_update(learning_rate, activation2, weight2, data):
  counter = 0
  temp = 0
  while counter < 200:
    counter += 1
    #print("\ncounter: ", counter)
    flag = 0
    for i in range(6):
      activation2 = np.dot(data[i], np.transpose(weight2))
      if activation2 <= 0:
        temp = temp + data[i]
      if activation2 > 0:
        flag = flag + 1
      #print("i: ", i)
      #print("activation: ", activation2)
    if (flag==6): break
    weight2[0] = weight2[0] + learning_rate*temp
  #print("\n\nANSWER:")
  #print("flag : ", flag)
  #print("weight: ", weight2)
  #print("counter: ", counter)
  return weight2, counter
```

```
results=[]
models=[]
# weight = [1 1 1 1 1 1]
print("\n-----Weight = [1 1 1 1 1 1]----- \n")
for i in range(10):
  learning_rate = 0.1 + (i*0.1)
  weights, iterations = one_update(learning_rate, np.zeros((1,6), dtype=float), np.one
s((1,6), dtype=float), data)
  print("\nLearning rate: ", learning_rate )
  print("\t--single update--\n\tWeights: ", weights, "\n\tIterations: ",iterations)
  results.append(iterations)
  models.append("W1, LR" +str(np.around(learning_rate, decimals=1))+ ", Single")

  weights2, iterations2 = batch_update(learning_rate, np.zeros((1,6), dtype=float), np
.ones((1,6), dtype=float), data)
  print("\n\t--batch update--\n\tWeights: ", weights2,"\n\tIterations: ",iterations2)
  results.append(iterations2)
  models.append("W1, LR" +str(np.around(learning_rate, decimals=1))+ ", Batch")

# weight = [0 0 0 0 0 0]
print("\n-----Weight = [0 0 0 0 0 0]----- \n")
for i in range(10):
  learning_rate = 0.1 + (i*0.1)
  weights, iterations = one_update(learning_rate, np.zeros((1,6), dtype=float), np.zer
os((1,6), dtype=float), data)
  print("\nLearning rate: ", learning_rate )
  print("\t--single update--\n\tWeights: ", weights, "\n\tIterations: ",iterations)
  results.append(iterations)
  models.append("W0, LR" +str(np.around(learning_rate, decimals=1))+ ", Single")

  weights2, iterations2 = batch_update(learning_rate, np.zeros((1,6), dtype=float), np
.zeros((1,6), dtype=float), data)
  print("\n\t--batch update--\n\tWeights: ", weights2,"\n\tIterations: ",iterations2)
  results.append(iterations2)
  models.append("W0, LR" +str(np.around(learning_rate, decimals=1))+ ", Batch")

# weight = random
print("\n-----Weight = random----- \n")
for i in range(10):
  learning_rate = 0.1 + (i*0.1)
  weights, iterations = one_update(learning_rate, np.zeros((1,6), dtype=float), np.ran
dom.sample(size =(1, 6)) , data)
  print("\nLearning rate: ", learning_rate )
  print("\t--single update--\n\tWeights: ", weights, "\n\tIterations: ",iterations)
  results.append(iterations)
  models.append("WR, LR" +str(np.around(learning_rate, decimals=1))+ ", Single")
```

```
  weights2, iterations2 = batch_update(learning_rate, np.zeros((1,6), dtype=float), np
.random.sample(size =(1, 6)) , data)
  print("\n\t--batch update--\n\tWeights: ", weights2,"\n\tIterations: ",iterations2)
  results.append(iterations2)
  models.append("WR, LR" +str(np.around(learning_rate, decimals=1))+ ", Batch")

# barchart


objects = models
y_pos = np.arange(len(objects))
performance = results
plt.xlim(0.0, 200.0)
plt.barh(y_pos, performance, align='center', alpha=0.50)
plt.yticks(y_pos, objects)
plt.ylabel('Hyperparameters')
plt.xlabel('Iterations')
plt.title('Perceptron Iteration Count Comparison')

plt.show()
```

Result Analysis

a) **In task 2, why do we need to take the sample points to a high dimension?**
Ans: Because the data is found to be linearly inseparable. We must map it to a higher dimension in order to find a decision surface that can separate the data points.

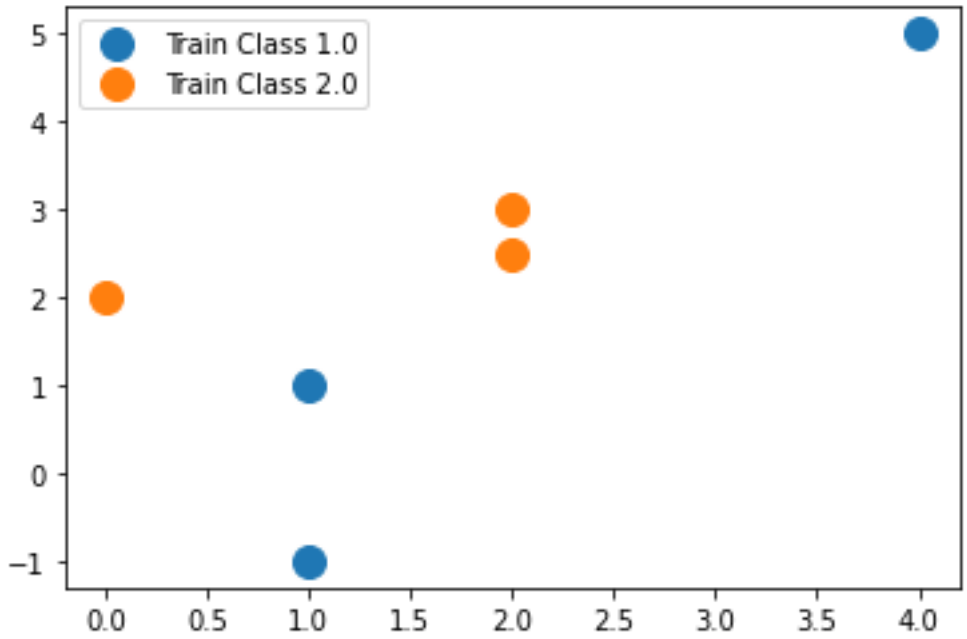

Fig 1: - Graphical representation of the training data in the feature space

a) **In each of the three initial weight cases and for each learning rate, how many updates does the algorithm take before converging?**
Ans: The number iterations in the table below gives us the amount of updates needed.

Initial Weight	Update type	Number of Iterations per Learning Rate									
		0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1.0
Weight = 1	Single	6	147	149	149	141	157	136	136	140	141
	Batch	84	68	92	94	74	73	73	67	67	67
Weight = 0	Single	141	141	141	141	141	141	141	141	141	141
	Batch	61	61	61	61	61	61	61	61	61	61
Weight = Random	Single	134	147	143	136	150	136	149	136	149	136
	Batch	67	94	67	73	67	74	74	74	67	74

Fig 2: - Tabular Representation of Number of Iterations for each variation in Hyperparameter

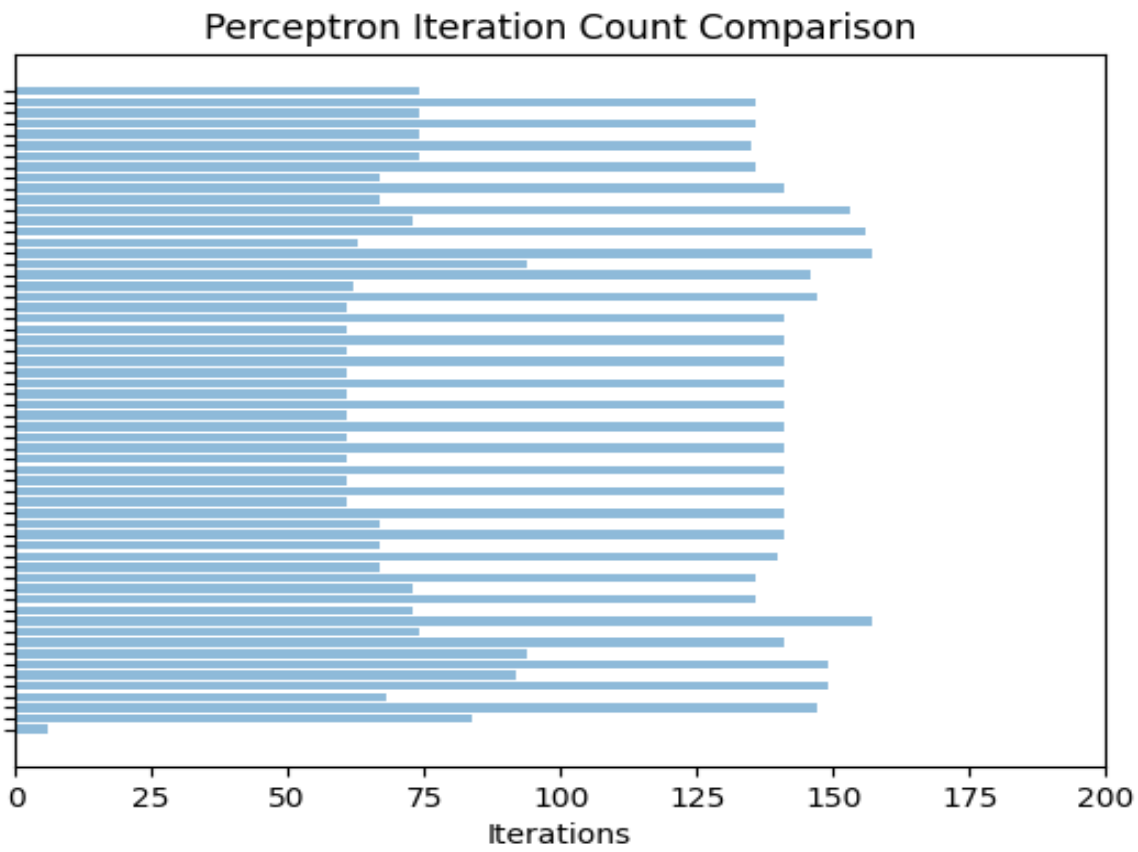

Fig 2: - Bar Graph Comparing Number of Iterations for each variation in Hyperparameter

Conclusion

This model determines whether the training dataset is linearly separable. It then performs the kernel trick, and normalization takes place. Then, it shows us the number of iterations needed for convergence for differing hyperparameters in the Perceptron algorithms, such as different learning rates and initial weights taken.