*Rajshahi University of Engineering & Technology*


Course No.: CSE 4204

Course Title: Sessional based on CSE 4203

[Neural Networks and Fuzzy Systems]

Submitted by:

1603001

Shegufa Rob

Date of Submission: 24 July 2022

Topics:

1. Implementing K Nearest Neighbor Classifier
2. Implementing Naïve Bayes Classifier
3. Implementing Single Layer Perceptron Learning Algorithm

## *Title: Implementing K Nearest Neighbor Classifier*

### *Theory:*

First, we are given a dataset which includes class labels. Now, if we are given a new input and its feature values, we can determine the class this new input belongs to using K nearest neighbor (knn).

- For the new input, we need to calculate its distance from each data point. [We can use various distance measures; we will discuss it later.]
- Sort the data points according to the distance values.
- Then, we have to take k points which has smallest distance values and see which class do the majority of these points belong to. That is our class.
- A problem can occur when we need 3 values, but we cannot decide the points because many of them might contain same distance values. In this case, we can do one of the following:
  - i) Take any value randomly.
  - ii) Increase/ decrease k until this problem is solved.
  - iii) Use different distance measure.
  - iv) We can assign weights to the same distant data points. Say, we need r data points, but we have m data points of same distance, then we assign to them weight r/m. For the first k-r data points, we can assign weight value to 1. Then calculate the sum of weights and see which class yields bigger value, that's our desired class.

    [We will follow the fourth procedure in case same distance occurs]

Example:

A dataset is given as below:

| Indices | Height | Weight | Class |
|---|---|---|---|
| 0 | 158 | 58 | B (0) |
| 1 | 158 | 59 | B (0) |
| 2 | 158 | 63 | B (0) |
| 3 | 160 | 64 | R (1) |
| 4 | 163 | 64 | R (1) |
| 5 | 165 | 61 | R (1) |

Now, a person has height 161 & weight 61. We have to determine which class the person belongs to using knn where k=3 & distance function is:

i) City-Block

$$d = \sum_n | new[i] - dp[i] |$$

ii) Euclidian

$$d = \sqrt{\sum_n (new[i] - dp[i])^2}$$

We will see the solution using City-Block distance:

City-Block distance of new data point (161,61) from each data point of our dataset:

| Indices | Class | Distance |
|---|---|---|
| 0 | 0 | 6 |
| 1 | 0 | 5 |
| 2 | 0 | 5 |
| 3 | 1 | 4 |
| 4 | 1 | 5 |
| 5 | 1 | 4 |

Sort the data points based on distance:

| Indices | Class | Distance |
|---|---|---|
| 3 | 1 | 4 |
| 5 | 1 | 4 |
| 1 | 0 | 5 |
| 2 | 0 | 5 |
| 4 | 1 | 5 |
| 0 | 0 | 6 |

As k = 3,
We can take first 2 values here, then for the next 3 data points we have same distance values, so we assign weights r/m (1/3).

| Class | Distance | Weight |
|---|---|---|
| 1 | 4 | 1 |
| 1 | 4 | 1 |
| 0 | 5 | 1/3 |
| 0 | 5 | 1/3 |
| 1 | 5 | 1/3 |

Class 1 = 1+1+1/3 = 2.33
Class 0 = 1/3 + 1/3 = 0.67

Hence, the data point belongs to class 1.

*Code:*

"""K_nearest_neighbor.ipynb

Automatically generated by Colaboratory.

Original file is located at
https://colab.research.google.com/drive/1kcGomqqurs_fV6EKXBCAzAWO8lkSkcld

# **Classification using k nearest neighbor**
"""

```
import numpy as np
import math

data0 = np.array([[158,58,0], [158,59,0],[158,63,0],[160,64,1],[163,64,1],[165,61,1]])
data = data0.copy()

print(data)

new = [161,61]

def city_block_distance(data,new):
  dist = np.zeros((6,3))
  for i in range(6):
    dist[i] = (data[i][2],0,1)

  dp = 0
  for i in data:
    cbd = abs(new[0]-i[0]) + abs(new[1]-i[1])
    dist[dp][1] = cbd
    dp = dp+1
  return dist

def euclidian_distance(data,new):
  dist = np.zeros((6,3))
  for i in range(6):
    dist[i] = (data[i][2],0,1)

  dp = 0
  for i in data:
    ed = math.sqrt((new[0]-i[0])**2+(new[1]-i[1])**2)
    dist[dp][1] = ed
    dp = dp+1
```

```
  return dist

def sort_by_distance(dist):
  sorted_dist = dist[dist[:,1].argsort()]
  return sorted_dist

def finding_same_distance_indices(k,sorted_dist):
  p = k
  while(1):
    if sorted_dist[p][1]==sorted_dist[p-1][1]:
      p=p+1
    else:
      break

  li = p-1
  l = sorted_dist[li][1]

  si = 0
  for i in sorted_dist:
    if i[1] == l:
      break
    si +=1
  #print(li,si)

  m = li-si+1
  r = k-si

  return si,li,r,m

def fixing_weights(sorted_list,si,li,r,m):
  for i in range(si,li+1):
    sorted_dist[i][2] = r/m
  return sorted_dist

def class_det(sorted_dist,li):
  c0 = 0
  c1 = 0
  for i in sorted_dist[0:li+1]:
    if i[0] == 0:
      c0 += i[2]
    else:
      c1 += i[2]
  if c0>c1:
    print("Class 0")
  else:
    print("Class 1")
```

"""**Distance measure: City Block distance**"""

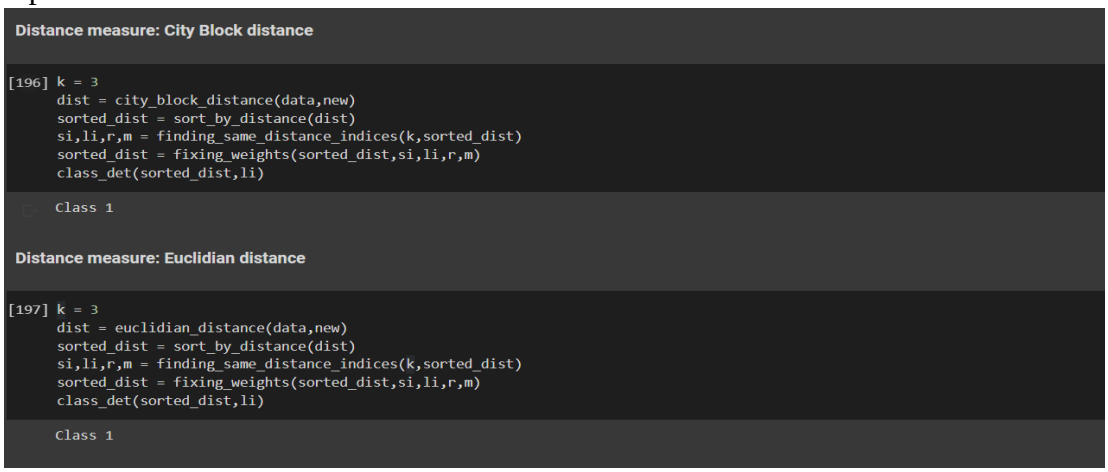
```
k = 3
dist = city_block_distance(data,new)
sorted_dist = sort_by_distance(dist)
si,li,r,m = finding_same_distance_indices(k,sorted_dist)
sorted_dist = fixing_weights(sorted_dist,si,li,r,m)
class_det(sorted_dist,li)
```

"""**Distance measure: Euclidian distance**"""

```
k = 3
dist = euclidian_distance(data,new)
sorted_dist = sort_by_distance(dist)
si,li,r,m = finding_same_distance_indices(k,sorted_dist)
sorted_dist = fixing_weights(sorted_dist,si,li,r,m)
class_det(sorted_dist,li)
```

## *Result:*

The results were as expected. For both measures we got the same class for the given input.




## *Findings:*

i) Easy to implement.

ii) K nearest neighbor classifier can classify a data point easily where we have small dataset. But in case of, bigger dataset, complexity will increase as we will have to calculate distance for each data point.

iii) For smaller values of k, the algorithm might fail in case of too much rogue patterns, bigger values of k will increase complexity. Hence, finding an optimum value of k is a crucial point.

## *Title: Implementing Naïve Bayes Classifier*

### *Theory:*

Naïve Bayes theorem can be expressed as below:

$$P(A|B) = \frac{P(B|A) * P(A)}{P(B)}$$

Naïve Bayes Classifier works similarly. If we have a dataset, then we are given a input with feature vector X = {x1,x2,…..}, we can determine its class using Naïve Bayes Classifier.

We need to calculate

$P(C_i \mid X)$ for each class i. For a pattern X = x1,x2,…..,xn, we can determine the probability of it being in a class C using Naïve Bayes theorem as below:

$$P(C|x1, x2, .., xn) = \frac{P(x1, x2, \dots, xn|C)P(C)}{P(x1, x2, \dots, xn)}$$

A pattern belongs to a class j if

$P(C_j \mid X) > P(C_i \mid X)$, for each class i, j != i

Example:

Say, we have 1000 fruits which could be either ‘banana’, ‘orange’ or ‘other’. The features of the fruits are long, sweet & yellow. Train dataset is as below:

| | Long | Sweet | Yellow | Total |
|---|---|---|---|---|
| Banana (0) | 400 | 350 | 450 | 500 |
| Orange (1) | 0 | 150 | 300 | 300 |
| Other (2) | 100 | 150 | 50 | 200 |
| | | | | 1000 |

Let’s say, we are given a fruit that is long, sweet & yellow. We have to predict which fruit it is using Naïve Bayes Classifier.

So, we need to calculate the values of

P(Banana| long, sweet, yellow) , P(Orange| long, sweet, yellow) , P(Other| long, sweet, yellow) .

The one which yields bigger value, our sample belongs to that class.

*Code:*

```
"""Naive_Bayes_Classifier.ipynb

Automatically generated by Colaboratory.

Original file is located at
    https://colab.research.google.com/drive/1T2xABvFITpQ8H0SnBKeHj96Km1rAUCES

"""

import numpy as np
import math

"""long, sweet, yellow, total, class

0,1,2,3,4
"""

data0 = np.array([[400.0,350.0,450.0,500.0,0.0],[0.0,150.0,300.0,300.0,1.0],[100.0,150.0,50.0,200.0,2.0]])
data = data0.copy()
total = 0
for i in data:
  total += i[3]

for i in data:
  i[0] = i[0]/i[3]
  i[1] = i[1]/i[3]
  i[2] = i[2]/i[3]
  i[3] = i[3]/total
```

```
print(data)


P = np.zeros(3)


j = 0
for i in data:
  P[j] = i[0]*i[1]*i[2]*i[3]
  j+=1
print(P)


max = -1
for i in range(P.size):
  if P[i]>max:
    max = P[i]
    c = i


print("Class "+ str(c))
```

## *Result:*

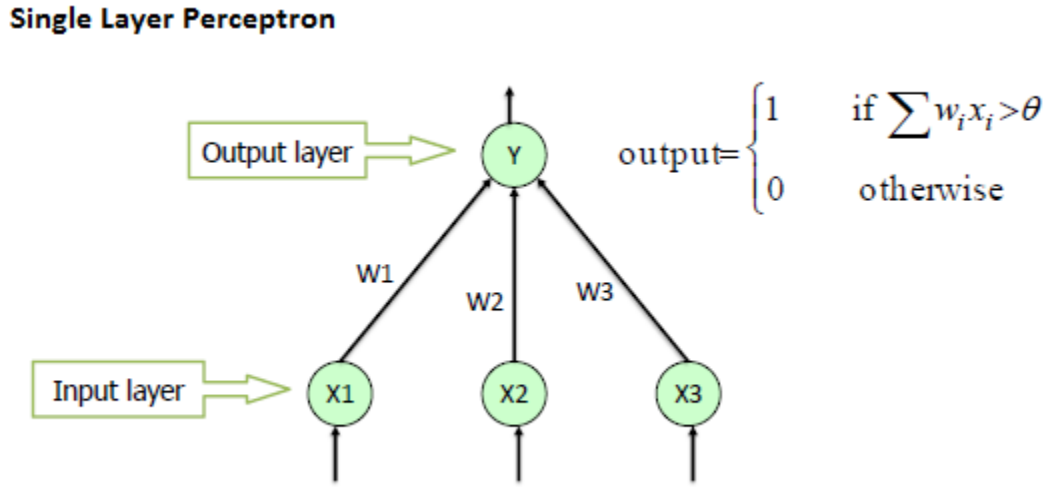
We got the expected result.

```
[61] max = -1
     for i in range(P.size):
       if P[i]>max:
         max = P[i]
         c = i

     print("Class "+ str(c))


     Class 0
```

## *Title: Implementing Single Layer Perceptron Learning Algorithm*

*Theory:*

Perceptron learning algorithm is basically updating weights for input patterns so that the weighted sum if passed through an activation function yield correct output. So, our target is to determine weights for the whole procedure. There are many kinds of activation functions. We will be using binary step function.




[Image taken from https://www.saedsayad.com/artificial_neural_network_bkp.htm ]

The algorithm can be stated as below:

- Initialize weights and threshold value
- Present input and outputs
- Calculate weighted sum : $net = \sum_n xi * wi$
  If net>= threshold value: predicted output = 1
  Else predicted output = 0
- We calculate error measure, delta = given output – predicted output; and update weights using this value:
  wi(t+1) = wi(t) + a*delta*xi
  here, a is adaption rate, 0<=a<=1.

*Code:*

"""Perceptron Learning algorithm.ipynb

Automatically generated by Colaboratory.

Original file is located at

https://colab.research.google.com/drive/1sGrzJ2w5NWQvkJxungCFexwiI4tgWSwD

"""

```
import numpy as np
import math

w = np.array([1.0,2.0])
a = 0.1
th = 2.5

data = np.array([[2.0,1,0],[5,6,1]])

epoch = 0
print("x1 x2 y w1 w2 yp delta w1 w2\n")
while(1):
  error = 0
  epoch +=1
  print(epoch)
  for i in data:
    ws = sum(w*i[0:2])
    if ws>=th:
      yp = 1
    else:
```

```
    yp = 0

  delta = i[2] - yp
  if delta!=0:
    error+=1

  t0 = w[0]
  t1 = w[1]

  w[0] = round(w[0] + a*delta*i[0],2)
  w[1] = round(w[1] + a*delta*i[1],2)

  print(i[0],i[1],i[2],t0,t1,yp,delta, w[0],w[1])
  print("\n")

 if error == 0 :
  print("Final weights are : [" + str(w[0]) + " " + str(w[1]) + "]")
  break
```

*Result:*

We got results as calculated.

```
x1 x2 y w1 w2 yp delta w1 w2

1
2.0 1.0 0.0 1.0 2.0 1 -1.0 0.8 1.9


5.0 6.0 1.0 0.8 1.9 1 0.0 0.8 1.9


2
2.0 1.0 0.0 0.8 1.9 1 -1.0 0.6 1.8


5.0 6.0 1.0 0.6 1.8 1 0.0 0.6 1.8


3
2.0 1.0 0.0 0.6 1.8 1 -1.0 0.4 1.7


5.0 6.0 1.0 0.4 1.7 1 0.0 0.4 1.7


4
2.0 1.0 0.0 0.4 1.7 1 -1.0 0.2 1.6


5.0 6.0 1.0 0.2 1.6 1 0.0 0.2 1.6


5
2.0 1.0 0.0 0.2 1.6 0 0.0 0.2 1.6


5.0 6.0 1.0 0.2 1.6 1 0.0 0.2 1.6


Final weights are : [0.2 1.6]
```

As, we can see, in the epoch=5, the 3rd column (desired output) and the 6th column (predicted output) are same and hence, delta column is 0 for both patterns. Thus, we get final weights from this phase and terminate the process.

*Findings:*

i) Changing the adaption rate, ‘a’, affects the number of iterations needed. For the above problem, when we used a=0.1, we got the final weights in 5 epochs, whereas, when used a=0.2, we got the results in 3 epochs.

ii) Then, again, the initial values of weights and threshold also affect the process. When we decreased the values of weights from [1 2] to [0.5 2], number of iterations decreased from 5 to 3.

iii) When the classes in the problem is linearly separable, single layer perceptron works fine, but the problems where the classes are not linearly separable, single layer perceptron fails such as in XOR problem.