Rajshahi University of Engineering & Technology

Department of Computer Science & Engineering

Lab Report

Course No: CSE 4204

Course Title: Sessional Based on CSE 4203

Submitted By

Saifur Rahman

Roll: 1703018

Sec: A

Dept. of CSE, RUET

Submitted To

Dr. Md. Rabiul Islam

Professor

Dept. of CSE, RUET

Date of Submission: 10th June, 2023

Contents

Assignment No. 1

Assignment Name

Implementation of K-Nearest Neighbors (KNN) algorithm from scratch & Comparison with built-in KNN.

Objectives

- To implement the K-Nearest Neighbors (KNN) algorithm.
- To classify data points based on their nearest neighbors.
- To compare the implementation with built-in function.

Problem Definition

The problem is to implement the K-Nearest Neighbors (KNN) classification algorithm to classify individuals as either underweight or normal based on their weight and height. The goal is to create a model that can accurately classify new individuals into these two classes using the KNN approach.

Methodology

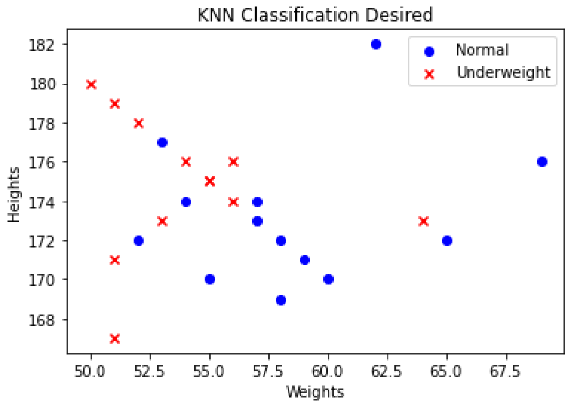
1. Define functions to calculate the Euclidean distance between two points, plot the KNN graph with different classes represented by different markers and colors, to calculate the accuracy of the classification results.
2. Input the data, including weights, heights, and classes (0 for underweight and 1 for normal).
3. Specify the percentage of training data, the number of neighbors (K), and calculate the number of training and testing data points.
4. Iterate through each testing data point:
    a. Calculate the Euclidean distance between the testing point and each training point.
    b. Sort the distances in ascending order.
    c. Determine the class based on the majority of the K nearest neighbors.
    d. Assign the predicted class to the corresponding testing data point.
5. Calculate the accuracy of the classification results by comparing them with the original classes.
6. Plot the desired KNN classification graph with the actual classes.
7. Use the built-in KNN classifier for comparison:
    e. Split the data into training and testing sets.
    f. Create a KNN classifier object and fit it to the training data.
    g. Predict the classes of the testing data.
    h. Calculate the accuracy of the built-in KNN classification results.
8. Plot the desired and actual KNN classification graphs using the built-in KNN classifier.
9. Summarize the results, performance of the built-in KNN and self-implemented KNN classifier.

Implementation

```
# -*- coding: utf-8 -*-
"""KNN Classification

Automatically generated by Colaboratory.

Original file is located at
    https://colab.research.google.com/drive/1V8jPNditGWhe6x72VAzILgTkVFnnFmLS

# Imports
"""

import math
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from sklearn.neighbors import KNeighborsClassifier

"""# Function Definition

Eucledian Distance
"""

def euclideanDist(p, q):
  x1 = p[0]
  y1 = p[1]
  x2 = q[0]
  y2 = q[1]
  return round(math.sqrt((x1 - x2)**2 + (y1 - y2)**2), 2)

"""Plot KNN Graph"""

def plot_knn(classes, weights, heights, title):
  x1 = []
  y1 = []
  x2 = []
  y2 = []
  for i in range(0, len(classes)):
    if classes[i] == 1:
      x1.append(weights[i])
      y1.append(heights[i])
    else:
      x2.append(weights[i])
      y2.append(heights[i])

  plt.scatter(x1, y1, c = 'blue', marker = 'o')
  plt.scatter(x2, y2, c = 'red', marker = 'x')
  plt.title(title)
  plt.legend(['Normal', 'Underweight'])
  plt.xlabel('Weights')
  plt.ylabel('Heights')

"""Accuracy Calculation"""

def accuracy(result, origin):
  print('Result:  ', result)
  print('Original:', origin)
  matched = 0
  non_matched = 0
  for i in range(0, len(result)):
    if result[i] == origin[i]:
      matched = matched + 1
    else:
      non_matched = non_matched + 1
  acc = (matched / (matched + non_matched)) * 100
  print('Accuracy:  ', acc, '%')

"""# Input Data"""

n = 25
weights = [51, 62, 69, 64, 65, 56, 58, 57, 55, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 53, 54, 55, 60, 51, 52]
heights = [167, 182, 176, 173, 172, 174, 169, 173, 170, 180, 179, 178, 177, 176, 175, 176, 174, 172, 171, 173, 174, 175, 170, 171, 172]
classes = [0, 1, 1, 0, 1, 0, 1, 1, 1, 0, 0, 0, 1, 0, 0, 0, 1, 1, 1, 0, 1, 0, 1, 0, 1] # 0 = Underweight, 1 = Normal

"""Dataframe Definition"""

df = pd.DataFrame(list(zip(weights, heights, classes)), columns = ['Weight', 'Height', 'Class'])
copied_df = df.copy() # copied dataframe for accuracy calculation
percentage = 60 # percentage of training data
k = 5 # no of neighbors
df

"""# KNN Classifier

Training & Testing Data
"""

l = math.ceil(n * (percentage / 100)) # no of training data
t = n - l # no of testing data
m = len(df) - t # test data start index
```

```
unknown = df.iloc[l:, 0:2].values.tolist() # test data list
print(unknown)

"""Training"""

for j in range(0, len(unknown)): # testing data

  # Calculate euclidean distance
  dist = []
  for i in range(0, m): # training data
      p = df.iloc[i]
      dist.append([i, euclideanDist(unknown[j], p)]) # index, distance

    # Sort distances
    dist.sort(key= lambda a: a[1]) # sort list by distance

    # Classification
    c1 = 0 # no of nearest neighbors of first class
    c2 = 0 # no of nearest neighbors of second class

    for i in range(0, k): # takes k nearest neighbors under consideration
      if df.iloc[dist[i][0]][2] == 1: # if first class's point is closest
        c1 = c1 + 1
      else:
        c2 = c2 + 1
    print('c1: ', c1, 'c2: ',c2)

    if c1 > c2:
      df.iloc[l][2] = 1 # assigned as class 1
      l = l + 1
    else:
      df.iloc[l][2] = 0 # assigned as class 2
      l = l + 1

df[m:]

copied_df[m:]

"""# Testing"""

result = df.iloc[m:, 2].values.tolist()
origin = copied_df.iloc[m:, 2].values.tolist()
accuracy(result, origin)

"""# Plotting

Desired
"""

plot_knn(classes, weights, heights, 'KNN Classification Desired')

"""Actual"""

updated = df.loc[:, 'Class'].values.tolist()
plot_knn(updated, weights, heights, 'KNN Classification Actual')

"""# Built in KNN"""

X_train = copied_df.loc[:m - 1, ['Weight', 'Height']]
X_test = copied_df.loc[m:, ['Weight', 'Height']]
y_train = copied_df.loc[:m - 1, 'Class']
y_test = copied_df.loc[m:, 'Class']

knn = KNeighborsClassifier(n_neighbors=k)

knn.fit(X_train, y_train)

# Predict on dataset which model has not seen before
y_test = knn.predict(X_test)

built_updated = copied_df.loc[:m - 1, 'Class'].values.tolist()
built_updated = built_updated + list(y_test)

"""Accuracy"""

accuracy(list(y_test), origin)

"""Desired"""
plot_knn(classes, weights, heights, 'Built-in KNN Classification Desired')

"""Actual"""
plot_knn(built_updated, weights, heights, 'Built-in KNN Classification Actual'
```

Results & Performance Analysis



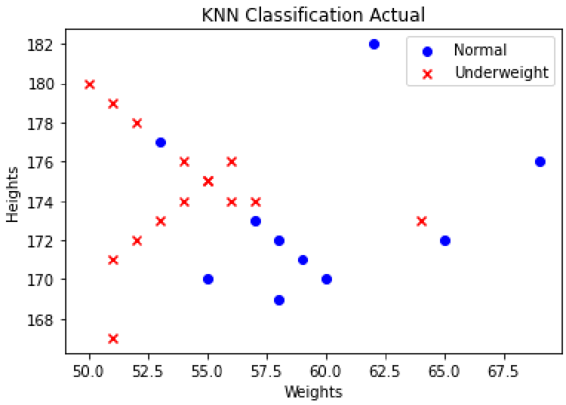
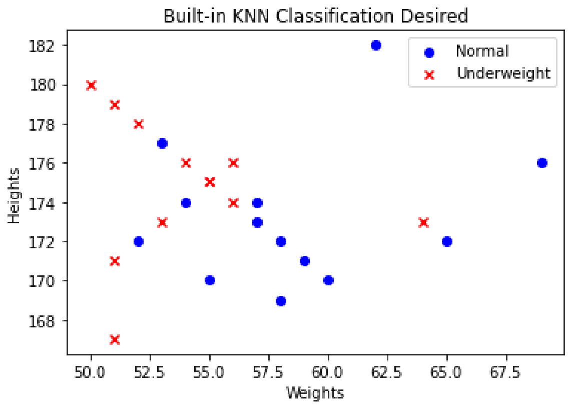


Fig 1.1: Original Data

Fig 1.2: Prediction of KNN

Accuracy of self-implemented KNN: 70%



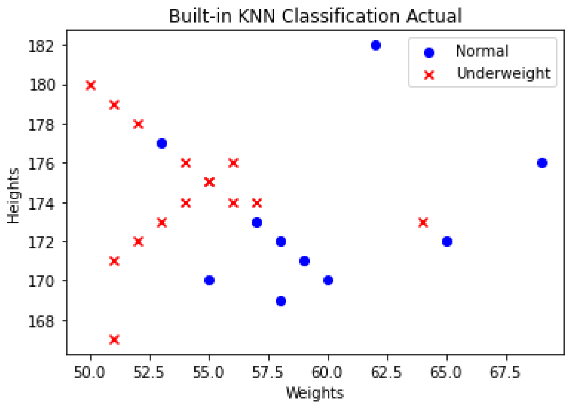


Fig 1.1: Original Data

Fig 1.2: Prediction of built-in KNN

Accuracy of built-in KNN: 70%

Conclusion & Observation

- The accuracy of the implemented KNN algorithm depends on various factors such as the choice of K (number of neighbors) and the quality of the training data.
- Here we can see that both self-implemented and built-in KNN provided same results. But this may vary when large dataset is used for classification.
- Here, the value of K is 5. A different value results different output. But for this dataset, K = 5 provides the optimum results. A small value of K may lead to overfitting.
- KNN is a distant based classification method. KNN will not work properly for datasets where the data points are not well-separated or have overlapping boundaries.
- KNN can suffer from the curse of dimensionality, particularly when dealing with high-dimensional data.

Assignment No. 2

Assignment Name

Implementation of Single Layer Perceptron learning algorithm.

Objectives

- To implement the single-layer perceptron learning algorithm.
- To evaluate its performance on a given dataset.
- To train the perceptron to classify input patterns accurately.

Problem Definition

This is a supervised learning problem which involves implementing a single-layer perceptron learning algorithm to classify binary input patterns. The goal is to train the perceptron to accurately classify the given input patterns into one of two classes: Class 0 and Class 1.

Input Patterns:

The input patterns consist of five binary features (x1, x2, x3, x4, and x5) represented by the values: x1, x2, x3, x4, and x5. Each feature can take on the value of either 0 or 1.

Desired Output:

For each input pattern, there is a corresponding desired output (y) indicating the class label. The desired output is either 0 or 1, representing Class 0 or Class 1, respectively.

The problem is to find the optimal weights (w1, w2, w3, w4, w5) and calculate actual output.

Methodology

1. Initialize the weights (w) and the threshold (T) for the perceptron.
2. Split the dataset into training and testing data.
3. Set the number of iterations (n) for the learning algorithm.
4. Start the learning loop.
5. Calculate the sum of products (sop) for each input pattern in the training data.
6. Apply the thresholding function to determine the actual output (act_y).
7. Compare the actual output (act_y) with the desired output (y).
8. Update the weights (w) based on the error between the actual and desired outputs.
9. After a weight is updated either move the iterator back to top or to starting point of the half where iterator was placed.
10. Repeat steps 5-8 for the specified number of iterations (n).
11. After the learning loop ends, the weights (w) represent the adapted weights.
12. Calculate the actual output (result) for each testing data pattern using the adapted weights.
13. Calculate the accuracy of the perceptron by comparing the actual output (result) with the desired output (origin).

Implementation

```python
# -*- coding: utf-8 -*-
"""5 Bit Single Layer Perceptron Learning Algorithm

Automatically generated by Colaboratory.

Original file is located at
    https://colab.research.google.com/drive/11Eu_BedRYmWegF-ezSyp1PfqhKSBq5si

# Imports
"""

import pandas as pd
import math

"""# Function Definition

Accuracy Calculation
"""

def accuracy(result, origin):
  matched = 0
  non_matched = 0
  for i in range(0, len(result)):
    if result[i] == origin[i]:
      matched = matched + 1
    else:
      non_matched = non_matched + 1
  acc = (matched / (matched + non_matched)) * 100
  print('Accuracy: ', round(acc, 2), '%')

"""# Input Data"""

# input
x1 = [0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1]
x2 = [0, 0, 1, 1, 0, 0, 1, 1, 0, 0, 1, 1, 0, 0, 1, 1, 0, 0, 1, 1, 0, 0, 1, 1, 0, 0, 1, 1, 0, 0, 1, 1]
x3 = [0, 0, 0, 0, 1, 1, 1, 1, 0, 0, 0, 0, 1, 1, 1, 1, 0, 0, 0, 0, 1, 1, 1, 1, 0, 0, 0, 0, 1, 1, 1, 1]
x4 = [0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1]
x5 = [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
# weight
w = [0.1, 0.3, 0.2, 0.3, 0.4]
# output
y = [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
# thresholding function
T = 0.5

"""Dataframe Definition"""

df = pd.DataFrame(list(zip(x5, x4, x3, x2, x1, y)), columns = ['x5', 'x4', 'x3', 'x2', 'x1', 'y'])
test_df = df.copy() # copied dataframe for accuracy calculation
df2 = df.copy()
l = len(df)
b = 5 # no of inputs/ index of class
percentage = 60 # percentage of training data
df

"""# Training

Training & Testing Data
"""

train = math.ceil(l * (percentage / 100)) # no of training data
test = l - train # no of testing data
print(train, test)

"""Learning"""

n = 50 # no of times loop will run
act_y = [0] * (n + 1)
it = 0 # no of iteration
i = 0 # loop iterator

while True:
  it = it + 1
  if i >= train:   # iterate through only train data
    i = 0

  sop = 0
  for k in range(b):
    sop = sop + (df.iloc[i, b - 1 - k] * w[k])
  # actual output according to the thresholding function
  if sop >= T:
    act_y[i] = 1
  else:
```

```
    act_y[i] = 0

  # actual and desired output comparison
  if act_y[i] == df.loc[i, 'y']: # desired and actual matched
    i = i + 1 # no change in weight
  else:
    if df.loc[i, 'y'] == 1 and act_y[i] == 0: # desired 1, actual 0
      for j in range(0, len(w)):
        w[j] = w[j] + df.iloc[i, len(w) - 1 - j] # add input to weight
      i = 0 # startover
    else: # desired 0, actual 1
      for j in range(0, len(w)):
        w[j] = w[j] - df.iloc[i, len(w) - 1 - j] # subtract input from weight
      i = 0 # startover

  if it > n : # loop runs n + 1 times
          break
      print('Adapted weight:', w)
      print('no of iterations: ', it)
      """# Testing"""
      # desired output
      origin = df.loc[train:, 'y'].values.tolist()
      print(origin)

      # actual output
      result = []
      for i in range(train, len(df)):

        sop = 0
        for k in range(b):
          sop = sop + (df.iloc[i, b - 1 - k] * w[k])
        if sop >= T:
          result.append(1)
        else:
          result.append(0)
      print(result)

      """Accuracy Calculation type1"""

      accuracy(result, origin)

      """# Type - 2"""

      n = 50 # no of times loop will run
      act_y = [0] * (n + 1)
      it = 0 # no of iteration
      i = 0 # loop iterator

      while True:
        it = it + 1
        if i >= train: # iterate through only train data
          if i > (train / 2):
              i = math.ceil(train / 2)
          else:
              i = 0
        for k in range(b):
          sop = sop + (df.iloc[i, b - 1 - k] * w[k])

        # actual output according to the thresholding function
        if sop >= T:
          act_y[i] = 1
        else:
          act_y[i] = 0

        # actual and desired output comparison
        if act_y[i] == df2.loc[i, 'y']: # desired and actual matched
          i = i + 1 # no change in weight
        else:
          if df2.loc[i, 'y'] == 1 and act_y[i] == 0: # desired 1, actual 0
            for j in range(0, len(w)):
              w[j] = w[j] + df2.iloc[i, len(w) - 1 - j] # add input to weight

            if i > (train / 2):
              i = math.ceil(train / 2)
            else:
              i = 0 # startover
          else: # desired 0, actual 1
            for j in range(0, len(w)):
              w[j] = w[j] - df2.iloc[i, len(w) - 1 - j] # subtract input from weight
            if i > (train / 2):
              i = math.ceil(train / 2)
            else:
              i = 0 # startover
```

```
        if it > n : # loop runs n + 1 times
            break

    print('Adapted weight:', w)
    print('no of iterations: ', it)

    # actual output type2
    result2 = []
    for i in range(train, len(df2)):
        sop = 0
        for k in range(b):
            sop = sop + (df.iloc[i, b - 1 - k] * w[k])
        if sop >= T:
            result2.append(1)
        else:
            result2.append(0)
    print(result2)

    """Accuracy calculation type 2"""
    accuracy(result2, origin)
```

Results & Performance Analysis

Here two types are considered for performance analysis. While iterating through learning loop, after updating weight the iterator moves back to:

1. The starting point (Type - 1)
2. The starting point of the half in which weight was updated (Type - 2)

For 60% training data,

Accuracy:

Type - 1: 66.67%

Type – 2: 100%

For 80% training data,

Accuracy:

Type - 1: 100%

Type – 2: 100%

Conclusion & Observation

From the assignment, we can observe that the performance of the algorithm depends on following factors:

- Type of data: Dataset with 3 feature results more accurate than dataset with 5 features. As the no. of feature increases the accuracy decreases.
- Train-test splitting: In this dataset, the last values of classes are 1. So, while splitting many patterns of class 1 has been moved to validation set. And no pattern of class 0 has been moved there. That's why type-2 accurately predicted for small training data.

Overall, type – 2 saves much time as the loop does not need to move back all the way to starting point. So, it will be fast learning process. But whether or not it is the best choice for learning that solely depends on how the patterns and classes are sorted, and how the data is split for training.

Assignment No. 3

Assignment Name

Implementation of Multi-layer Perceptron Learning algorithm.

Objectives

- To implement a multi-layer perceptron (MLP) learning algorithm.
- To train the MLP on the given dataset.
- To test the trained MLP.
- To analyze the training and testing results.

Problem Definition

This is a supervised learning problem which involves implementing a multi-layer perceptron learning algorithm to classify input patterns.

Input Patterns:

The input patterns consist of five binary features (x1, x2, x3, x4, and x5) represented by the values of these features: x1, x2, x3, x4, and x5. Each feature value is either 0 or 1.

Desired Output:

For each input pattern, there is a corresponding desired output (y) indicating the class label. The desired output is either 0, 1, or 2 representing Class 0, Class 1, or Class 2, respectively.

The problem is to implement hidden layer in between input and output layers.

Methodology

1. Initialize by specifying the number of input features, hidden layers, and output classes.
2. Initialize the MLP's weights and thresholds randomly.
3. Split the dataset into training and testing subsets.
4. Repeat the following steps until convergence or a maximum number of iterations:
    a. For each input example in the training subset:
    b. Forward propagate the input through the MLP to compute the predicted output.
    c. Calculate the error between the predicted output and the true output label.
    d. Backpropagate the error through the MLP to adjust the weights and thresholds using gradient descent.
5. Evaluate the accuracy of the trained MLP using the testing subset:
    a. For each input example in the testing subset:
    b. Forward propagate the input through the trained MLP to compute the predicted output.
    c. Compare the predicted output with the true output label and calculate the accuracy.
6. Output the accuracy of the trained MLP as the performance measure.

Implementation

```
# -*- coding: utf-8 -*-
"""Multi-layer Perceptron Learning Algorithm

Automatically generated by Colaboratory.

Original file is located at
    https://colab.research.google.com/drive/1THGqlQpXK1Bqb-4FUKBFonJJsGdL3v7x
"""
import pandas as pd
import random
import math
import numpy as np

"""# Input Data"""
# Input
x1 = [0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1]
x2 = [0, 0, 1, 1, 0, 0, 1, 1, 0, 0, 1, 1, 0, 0, 1, 1, 0, 0, 1, 1, 0, 0, 1, 1, 0, 0, 1, 1]
x3 = [0, 0, 0, 0, 1, 1, 1, 1, 0, 0, 0, 0, 1, 1, 1, 1, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 1, 1, 1, 1]
x4 = [0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1]
x5 = [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]

# Output
y = [0, 0, 0, 0, 1, 1, 1, 1, 2, 2, 2, 2, 1, 1, 1, 1, 2, 2, 2, 2, 0, 0, 0, 0, 2, 2, 2, 2, 1, 1, 1, 1]

df = pd.DataFrame(list(zip(x5, x4, x3, x2, x1, y)), columns = ['x5', 'x4', 'x3', 'x2', 'x1', 'y'])
data = df.values.tolist()
print(data)

num_rows = len(data)
num_cols = len(data[0]) - 1
num_unique_values = len(set(y))
print("Number of persons:", num_rows)
print("Number of features:", num_cols)
print("Number of classes:", num_unique_values)

"""Neural Network Definition"""

# input node = 5
inp = 5
hid = 2 # hidden node = 2
out = 3 # output node = 3 : no of classes = 5 - 8
k1 = 1
k2 = 1
N1 = 1
N2 = 1

"""Thresholding functions"""
uh = [0.4, 0.3] # thresholding function
uo = [0.1, 0.3, 0.2] # thresholding function

"""Weights
Wij
"""
rows, cols = (inp, hid) # input 5, hidden 2
w1 = [[0] * cols] * rows
for i in range(0, rows):
  for j in range(0, cols):
    w1[i][j] = round(random.uniform(0, 1), 2)
print(w1)

"""Wjk"""
rows, cols = (hid, out) # hidden 2, output 3
w2 = [[0] * cols] * rows
for i in range(0, rows):
  for j in range(0, cols):
    w2[i][j] = round(random.uniform(0, 1), 2)
print(w2)
"""# Functions"""
def initialization():
  rows, cols = (inp, hid) # input 5, hidden 2
  for j in range(0, cols):
    uh[j] = round(random.uniform(0, 1), 2)
  w1 = [[0] * cols] * rows
  for i in range(0, rows):
    for j in range(0, cols):
      w1[i][j] = round(random.uniform(0, 1), 2)
  #print(w1)
  rows, cols = (hid, out) # hidden 2, output 3
  for j in range(0, cols):
    uo[j] = round(random.uniform(0, 1), 2)
  w2 = [[0] * cols] * rows
```

```
for i in range(0, rows):
    for j in range(0, cols):
        w2[i][j] = round(random.uniform(0, 1), 2)
#print(w2)
return w1, w2, uh, uo
def outputCalculation1(layer1, layer2, input, thres, weight, k):
rows, cols = (layer1, layer2)
net = [0] * cols
sop = 0
for j in range(0, cols):
    sop = 0
    for i in range(len(input) - 1):
        sop = sop + weight[i][j] * input[i]
    net[j] = sop

activ = [0] * cols
for j in range(0, cols):
    activ[j] = net[j] + thres[j]

        Output = [0] * cols
        k = 1 # spread factors
        for j in range(0, cols):
            Output[j] = 1 / ( 1 + math.exp(-1 * k * activ[j]))

        return Output
    def outputCalculation2(layer1, layer2, input, thres, weight, k):
        rows, cols = (layer1, layer2)
        net = [0] * cols
        sop = 0
        for j in range(0, cols):
            sop = 0
            for i in range(0, rows):
                sop = sop + weight[i][j] * input[i]
            net[j] = sop

        activ = [0] * cols
        for j in range(0, cols):
            activ[j] = net[j] + thres[j]
        Output = [0] * cols
        k = 1 # spread factors
        for j in range(0, cols):
            Output[j] = 1 / ( 1 + math.exp(-1 * k * activ[j]))

        return Output
    def adjustOutputWeights(layer1, layer2, Output1, Output2, error, Weight, N, k):
        rows, cols = (layer1, layer2)
        delW = [[0] * cols] * rows
        for j in range(0, rows):
            for k in range(0, cols):
                delW[j][k] = N2 * k2 * error[k] * Output1[j] * Output2[k] * (1 - Output2[k])
                Weight[j][k] = Weight[j][k] + delW[j][k]
        return Weight
    def adjustOutputThres(layer, Output, error, thres, N, k):
        cols = layer
        delu = [0] * cols
        for k in range(0, cols):
            delu[k] = N2 * k2 * error[k] * Output[k] * (1 - Output[k])
            thres[k] = thres[k] + delu[k]
        return thres
    def adjustHiddenWeights(layer1, layer2, layer3, Output, Output1, error, W1, W2, N, k):
        rows, cols, hd = (layer1, layer2, layer3)
        delW = [[0] * cols] * rows
        for i in range(0, rows):
            for j in range(0, cols):
                sop = 0
                for k in range(0, hd):
                    sop = sop + W2[j][k] * error[k]
                delW[i][j] = N1 * k1 * O[i] * O1[j] * (1 - O1[j]) * sop
                W1[i][j] = W1[i][j] + delW[i][j]
        return W1
    def adjustHiddenThres(layer1, layer2, Output, error, W, thres, N, k):
        rows, cols = (layer1, layer2)
        delu = [0] * rows
        for j in range(0, rows):
            sop = 0
```

```
            for k in range(0, cols):
                sop = sop + W[j][k] * error[k]
            delu[j] = N1 * k1 * Output[j] * (1 - Output[j]) * sop
            thres[j] = thres[j] + delu[j]
        return thres

    def sigma(layer, desired, actual):
        sig = [0] * layer
        for k in range(0, layer):
            sig[k] = abs(desired[k] - actual[k])
        return sig
    def error(layer, desired, actual):
        for k in range(0, layer):
            sum = 0
            sum = sum + abs((desired[k] - actual[k])**2)
        Error = 0.5 * sum
        return Error

    """# Algorithm Initialization
    """
    w1, w2, uh, uo = initialization()

    """Train-Test Splitting"""
    p = 40
    m = math.ceil((len(data) * p / 100))
    train = [row[:] for row in data[:m]]
    test = [row[:] for row in data[m:]]
    O = train[0]
    T = [row[5] for row in data[:m]]

    """Input for hidden layer"""
    O1 = outputCalculation1(inp, hid, O, uh, w1, k1)
    print(O1)

    """Input for output layer"""
    #O2 = [0] * out
    O2 = outputCalculation2(hid, out, O1, uo, w2, k2)
    print(O2)

    """Error Calculation"""
    sig = sigma(out, T, O2)
    print(sig)

    """New Output Weights"""

    w2 = adjustOutputWeights(hid, out, O1, O2, sig, w2, N2, k2)
    print(w2)

    """New Output Thresholds"""
    uo = adjustOutputThres(out, O2, sig, uo, N2, k2)
    print(uo)

    """New Hidden Weights"""
    w1 = adjustHiddenWeights(inp, hid, out, O, O1, sig, w1, w2, N1, k1)
    print(w1)
    """New Hidden Thresholds"""
    uh = adjustHiddenThres(hid, out, O1, sig, w2, uh, N1, k1)
    print(uh)

    """Final Error"""
    Error = error(out, T, O2)
    print(Error)

    """# Training & Testing"""

    train = []
    test = []
    P = [50, 60, 70, 80, 90] # training percentage

    print("Training(%)\tAccuracy(%)")
    print("---------------------------")
    center = 7
    for p in P:
    # Training
        m = math.ceil((len(data) * p / 100))
        train = [row[:] for row in data[:m]]
        test = [row[:] for row in data[m:]]
        T = [row[5] for row in data[:m]]
        while(i < m):
            O = train[i]
            w1, w2, uh, uo = initialization()
```

```
            O1 = outputCalculation1(inp, hid, O, uh, w1, k1)
            O2 = outputCalculation2(hid, out, O1, uo, w2, k2)
            sig = sigma(out, T, O2)
            w2 = adjustOutputWeights(hid, out, O1, O2, sig, w2, N2, k2)
            uo = adjustOutputThres(out, O2, sig, uo, N2, k2)
            w1 = adjustHiddenWeights(inp, hid, out, O, O1, sig, w1, w2, N1, k1)
            uh = adjustHiddenThres(hid, out, O1, sig, w2, uh, N1, k1)
            E = error(out, T, O2)
            if(E < 0.33):
                i = i + 1
            else:
                i = 0
                continue
        # Testing
        c = 0
        for t in test:
            O1 = outputCalculation1(inp, hid, t, uh, w1, k1)
            O2 = outputCalculation2(hid, out, O1, uo, w2, k2)
            if(t[-1]==np.round(O2[0])):
                c = c + 1
    print(str(p).center(center),'t', str(round(c/len(test)*100.0, 2)).center(center))
```

Results & Performance Analysis

Training Data (%)	Accuracy (%)
50	25.00
60	33.33
70	44.44
80	66.67
90	100.00

Conclusion & Observation

- As the training percentage increases, the accuracy of the MLP also improves. This indicates that more data is needed for more accurate predictions.
- The accuracy of the MLP shows a steady improvement as the training percentage increases. This suggests that the MLP benefits from having a larger training data.
- A significant jump in accuracy is observed when the training percentage reaches 90%. This indicates that a sufficient amount of training data is needed for perfect accuracy.
- the accuracy remains relatively low for lower training percentages (e.g., 25% and 33.33%) and then starts to increase more rapidly as the training percentage exceeds 60%. This suggests that there is a threshold of training data required for the MLP.
- For different data, MLP may perform differently. It is also observed that for a random dataset, increasing the training data decreases accuracy.
- Train-Test splitting and organization of data has impact on MLP for small datasets.

Assignment No. 4

Assignment Name

Implementation of Kohonen Unsupervised learning algorithm.

Objectives

- To implement the Kohonen algorithm, also known as the self-organizing map (SOM).
- To learn and update the weights of the neural network.
- To find similar patterns after learning.

Problem Definition

Given a set of input patterns (x) and a set of new patterns (y), the goal is to train a Kohonen self-organizing map using the input patterns and update the weight vectors to accurately represent the input patterns in a lower-dimensional space. Once the training is complete, the objective is to determine the most similar patterns from the new patterns based on their distances to the learned weight vectors.

Methodology

1. Generate a set of input patterns.
2. Initialize the weight matrix with random values between 0 and 1.
3. Define no. of neighborhoods, N.
4. Define learning rate, lr.
5. Repeat the following steps until a break condition is met:
    a. Calculate the distances between each input pattern and the weight vectors.
    b. Create a copy of the current weight matrix.
    c. Sort the distances in ascending order and obtain the sorted indices.
    d. Update the neighborhood size based on the learning rate.
        i. N = N - lr * N
    e. Update the weight matrix using the sorted indices and the updated neighborhood size.
        i. $w_{ik} = w_{ik} + lr * (x_{ji} - w_{ik})$
    f. Check for a break condition
        i. by calculating the difference between the current and previous weight matrices. If the sum of differences is below a threshold, break the loop.
        ii. If the no. of neighborhoods is less than or equal to 1. (Referring to itself).
6. Generate a set of new patterns.
7. For each new pattern, calculate its distances to the weight vectors.
8. Find the most similar pattern for each new pattern by obtaining the minimum distance index from the sorted indices.
9. Output the results of the similarity checking.

Implementation

```
# -*- coding: utf-8 -*-
"""Kohonen algorithm
Automatically generated by Colaboratory.
Original file is located at
    https://colab.research.google.com/drive/1u0oP-dmKW1Wm_OFqpWqM9AGo2GBU68q7
"""
import pandas as pd
import random
import math
import numpy as np
"""# Input
Input data
"""
pattern = 100
feat = 10
def pattern_generation(pattern, feat):
  x = []
  for i in range(pattern):
      example = [random.randint(0, 1) for j in range(feat)]
      x.append(example)
  return x
x = pattern_generation(pattern, feat)
lr = 0.1 # learning rate
num_neighborhood = 5 # neighborhood size
print(x)
"""# Functions
Input weights
"""
def initialize_weight():
  rows, cols = (feat, pattern)
  w = [[0] * cols] * rows
  for i in range(0, rows):
    for j in range(0, cols):
      w[i][j] = round(random.uniform(0, 1), 2)
  return w
"""Calculate distance"""
def calculate_distance(x, w):
  d = [0] * pattern
  for j in range(0, pattern):
    sum = 0
    for i in range(0, feat):
      sum = sum + (x[j][i] - w[i][j])**2
    d[j] = sum
  return d
"""Calculate distance for new patterns"""
def calculate_distance_new(p, w, new_pattern):
  d = [0] * pattern
  for j in range(0, pattern):
    sum = 0
    for i in range(0, feat):
      sum = sum + (p[i] - w[i][j])**2
    d[j] = sum
  return d
"""sorted distance"""
def sort_distance(d):
  sorted_d = sorted(enumerate(d), key=lambda x:x[1])
  sorted_values = [x[1] for x in sorted_d]
  sorted_indices = [x[0] for x in sorted_d]
  return sorted_values, sorted_indices
"""Update weight"""
def update_weight(w, num_neighborhood, sorted_indices):
  for i in range(0, feat):
    for j in range(0, int(num_neighborhood)):
      k = sorted_indices[j]
      w[i][k] = w[i][k] + lr * (x[j][i] - w[i][k])
  return w
"""Check for break condition"""
def break_loop_for_weight(w, prev_w):
  dw = [[0] * pattern] * feat
  for i in range(0, feat):
    for j in range(0, pattern):
      dw[i][j] = abs(w[i][j] - prev_w[i][j])
  for i in range(0, feat):
    for j in range(0, pattern):
      if(dw[i][j] < 0.0001):
        dw[i][j] = 0
  sum = 0
  for i in range(0, feat):
    for j in range(0, pattern):
      sum = sum + dw[i][j]
  return sum
"""# Implementation
Learning
"""
w = initialize_weight()
while True:
  d = calculate_distance(x, w)
  prev_w = w.copy()
  sorted_values, sorted_indices = sort_distance(d)
  num_neighborhood = num_neighborhood - lr * num_neighborhood
  if(num_neighborhood <= 1):
    break
```

```
    w = update_weight(w, num_neighborhood, sorted_indices)

    sum = break_loop_for_weight(w, prev_w)
    if sum == 0:
        break
"""Outsiders"""
new_pattern = 10
y = pattern_generation(new_pattern, feat)
print(y)
"""Similarity Checking"""
i = 0
center = 10
print("new pattern\tsimilar with")
print("----------------------------")
for p in y:
    i = i + 1
    d_new = calculate_distance_new(p, w, new_pattern)
    sorted_values, sorted_indices = sort_distance(d_new)
    print(str(i).center(center),'\t', str(sorted_indices[0] + 1).center(center))
```

Results & Performance Analysis

The algorithm trains on random input sequences and tests on random output sequences to identify similarities between testing and training patterns.

Training Pattern = 100, Testing Pattern = 10

New Pattern	Similar With
1	66
2	66
3	60
4	60
5	60
6	77
7	47
8	1
9	77
10	47

Conclusion & Observation

- The learning rate (lr) and neighborhood size (N) affect the convergence and accuracy of the algorithm.
- The algorithm is sensitive to the initial random weights and may yield different results for each run.
- The number of patterns and features in the input data can impact the training process and the accuracy of the results.
- There was no proper evaluation metric found to evaluate the model performance.

Assignment No. 5

Assignment Name

Implementation of Hopfield algorithm.

Objectives

- To implement Hopfield algorithm.
- To learn patterns from the training data without explicit supervision.
- To converge a new pattern into a training pattern.

Problem Definition

Given a set of binary patterns (training data) and a test pattern, the objective is to implement the Hopfield algorithm to train the network using the training patterns and determine which training pattern the test pattern is most similar to. The code calculates the weight matrix based on the training patterns, performs pattern matching to identify the most similar training pattern, and outputs the corresponding index of the matched training pattern for each test pattern.

Methodology

1. Initialize the weight matrix:
    a. Set the diagonal elements of the weight matrix to 0.
    b. Calculate the off-diagonal elements of the weight matrix based on the training data.
        i. w[i][j] = Σ(train[:, i] * train[:, j])
2. Split the data into training and testing sets.
3. Perform pattern matching for each test pattern:
    c. Initialize a counter for matching patterns.
    d. Iterate until convergence:
        i. Update each element of the pattern based on the weight matrix.
        ii. Check if the pattern matches the previous iteration.
        iii. If matched, increment the counter.
        iv. Break the iteration if the counter reaches a threshold.
4. Find the matched training pattern for each test pattern:
    e. Compare each test pattern with the training patterns.
    f. Determine the index of the matched training pattern.
5. Print the test index and the corresponding converged training index.

Implementation

```python
# -*- coding: utf-8 -*-
"""Hopfield Algorithm
Automatically generated by Colaboratory.
Original file is located at
    https://colab.research.google.com/drive/1hXM_Rz-i0CXrPfNhMGsXforCTKApXW2-
"""
import pandas as pd
```

```
import random
import math
import numpy as np

"""# Input Data"""

(pattern, feat) = 24, 8
data = np.random.choice([-1, 1], size=(pattern, feat))
df = pd.DataFrame(data)
#print(df.head())

x = df.values.tolist()
x = np.array(x)
#print(x)

"""# Functions

Initialize weight matrix
"""

def initialize_weight(feat, train):
  rows, cols = (feat, feat)
  w = np.zeros((rows, cols))
  for i in range(0, rows):
    for j in range(0, cols):
      if i == j:
        w[i][j] = 0
      else:
        w[i][j] = np.sum(train[:, i] * train[:, j])
  return w

"""Match row"""

def match_row(train, updated):
  matched_row = None
  i = 0
  for row in train:
      if np.array_equal(row, updated):
          matched_row = row
          break
      i = i + 1
  return i
"""Train-Test Splitting"""

def splitting(percentage, x):
  m = math.ceil(len(x) * percentage / 100) # no of training examples
  train = x[:m][:]
  test = x[m:][:]
  return train, test

"""Pattern matching"""

def matched_pattern(w, test):
  j = 0
  matched = 0
  updated = test[t]
  while True:
    prev_updated = updated
    sum = np.sum(updated * w[j])
    if sum > 0:
      updated[j] = 1
    elif sum < 0:
      updated[j] = -1
    j = j + 1
    j = j % feat

    if np.array_equal(prev_updated, updated):
      matched = matched + 1
    if matched >= 4:
      break
  return updated

"""# Examples"""

percentage = 80
train, test = splitting(percentage, x)

#print("Training Data")
#print(train)

#print("Testing Data")
#print(test)

print(train[:, 0] * train[:, 1])

print(np.sum(train[:, 0] * train[:, 1]))

w = initialize_weight(feat, train)

#print(w)

"""# Implementation"""

      percentage = 60
      center = 10
      train, test = splitting(percentage, x)
      w = initialize_weight(feat, train)
      print('Test Index\tConverged at Train Index')
      print('-------------------------------------')
      for t in range(0, len(test)):
        updated = matched_pattern(w, test)
        i = match_row(train, updated)
print(str(t).center(center), '\t', str(i).center(center))
```

Results & Performance Analysis

In this algorithm, a new pattern similar to already existing pattern is given and it is predicted that which existing pattern is most similar to that new pattern. In other words, if an already existing pattern is slightly changed, the algorithm can detect which pattern it actually was.

Test Pattern	Converged with Training Pattern
0	15
1	15
2	15
3	12
4	15
5	15
6	7
7	15
8	12

Conclusion & Observation

- The Hopfield network can be used for pattern recall and completion after training.
- The network can recover and reconstruct the original stored pattern if a noisy pattern is given.
- There was no proper evaluation metric found to evaluate the performance of Hopfield algorithm.
- The algorithm is sensitive to the initial random weights and may yield different results for each run.