**MACHINE LEARNING (22AIE213)**

**ASSIGNMENT-2**

**NAME: HEMANTH KUMAR REDDY ,**

**HARSHAVARDHAN REDDY**

**REG NO: AIE22134 ,**

**AIE22169**

**Q1.** **Write a function to calculate the Euclidean distance and Manhattan distance between two vectors. The vectors dimension is variable. Please don’t use any distance calculation functions available in Python.**

**Psuedo Code :**

function euc\_distance(vec1, vec2):

distance = 0

for each element i in vec1:

distance += (vec1[i] - vec2[i]) squared

return square root of distance

function manhattan\_distance(vec1, vec2):

distance = 0

for each element i in vec1:

distance += absolute value of (vec1[i] - vec2[i])

return distance

function main():

input vec1 from user as a list of integers

input vec2 from user as a list of integers

euclidean = euc\_distance(vec1, vec2)

manhattan = manhattan\_distance(vec1, vec2)

print "Euclidean distance:", euclidean

print "Manhattan distance:", manhattan

if program is run directly:

call main function

**Code Explanation :**

This python script defines functions to compute Euclidean and Manhattan distances between two vectors, allowing users to input vectors and obtain their respective distances. The euc\_distance() function calculates the Euclidean distance by iterating through elements of the vectors, squaring their differences, summing them, and taking the square root of the sum. Similarly, the manhattan\_distance() function computes the Manhattan distance by summing the absolute differences of corresponding elements. The main() function facilitates user input, performs distance calculations, and displays the results.

**Q2.** **Write a function to implement k-NN classifier. k is a variable and based on that the count of neighbors should be selected.**

**Psuedo Code :**

# Importing necessary module

from collections import Counter

function euclidean\_distance(vector1, vector2):

return square\_root(sum((element1 - element2) \*\* 2 for element1, element2 in zip(vector1, vector2)))

function k\_nearest\_neighbors(training\_data, test\_instance, k=3):

# Calculating distances between test instance and training instances

distances = list of (euclidean\_distance(test\_instance, training\_instance[0]), training\_instance[1]) for each training\_instance in training\_data

# Sorting distances in ascending order

sorted\_distances = sort distances by the first element of each tuple

# Selecting the k nearest labels

k\_nearest\_labels = list of labels for the first k elements in sorted\_distances

# Determining the most common label among the k nearest neighbors

most\_common\_label = the most common label in k\_nearest\_labels

return most\_common\_label

# Initializing an empty list for training data

training\_data = []

num\_train\_instances = input("Enter the number of training instances: ")

# Loop through each training instance to input features and label

for \_ in range(num\_train\_instances):

features = list of floats obtained by splitting input("Enter features (comma-separated values): ") by comma

label = input("Enter the label for this instance: ")

append (features, label) to training\_data

test\_instance = list of floats obtained by splitting input("Enter test instance features (comma-separated values): ") by comma

k\_value = input("Enter the value of k: ")

predicted\_label = k\_nearest\_neighbors(training\_data, test\_instance, k=k\_value)

print "The predicted label for the test instance is:", predicted\_label

**Code Explanation :**

This python script the implements the k-nearest neighbors (k-NN) algorithm for classification. It begins by importing necessary modules and defining functions to calculate Euclidean distance and perform k-NN classification. The training data is initialized as an empty list, and users are prompted to input the features and labels for each training instance. Subsequently, the features for the test instance and the value of k are obtained from the user. The k-NN classification is then performed using the input data, and the predicted label for the test instance is printed.

**Q3. Write a function to convert categorical variables to numeric using label encoding. Don’t use any existing functionalities.**

**Psuedo Code :**

function encoding():

categorical\_data = input("Enter the categorical data using commas: ")

categorical\_list = split categorical\_data by commas

label\_encoding = {} // Dictionary to store label encoding

label\_counter = 0 // Counter for assigning numerical labels

for each category in categorical\_list:

if category is not in label\_encoding:

label\_encoding[category] = label\_counter

increment label\_counter by 1

numerical\_data = [] // List to store numerical representation of data

for each category in categorical\_list:

append label\_encoding[category] to numerical\_data

return numerical\_data

if program is run directly:

numerical\_data = call encoding() function

print "Numerical data:", numerical\_data

**Code Explanation :**

This python script defines a function called encoding() which prompts the user to input categorical data separated by commas. It then splits this input into a list of categories. Within the function, a dictionary called label\_encoding is initialized to store mappings between categories and their corresponding numerical labels, with label\_counter initialized to 0. For each category in the input list, if it's not already in label\_encoding, it's assigned a numerical label starting from 0 and incremented for each new category encountered. The function then generates numerical data by mapping each category to its numerical label. When the script is run directly, it calls the encoding() function to obtain the numerical data and prints it out.

**Q4.** **Write a function to convert categorical variables to numeric using One-Hotencoding. Don’t use any existing functionalities.**

**Psuedo Code :**

function categorical\_data\_input():

print "Enter categorical data separated by commas: "

raed input from user as categorical\_data

return categorical\_data

function onehot\_encoding(categorical\_data):

categories = split categorical\_data by commas

unique\_categories = get unique elements from categories

onehot\_encoding\_data = []

for each category in categories do

encoding = create a list of zeros with length equal to unique\_categories

index = get index of category in unique\_categories

encoding[index] = 1

append encoding to onehot\_encoding\_data

return onehot\_encoding\_data

if \_\_name\_\_ equals "\_\_main\_\_":

categorical\_data = call categorical\_data\_input()

onehot\_encoding\_data = call onehot\_encoding(categorical\_data)

print "One hot encoded data:"

for each encoding in onehot\_encoding\_data do

print encoding

**Code Explanation :**

This Python script comprises two functions: categorical\_data\_input() , onehot\_encoding(categorical\_data). It prompts the user to input categorical data separated by commas, while the latter performs one-hot encoding on this data. In the one-hot encoding function, the input data is split into individual categories, and unique categories are identified. For each category, a one-hot encoded vector is created, where a 1 is placed at the index corresponding to the category's position in the list of unique categories, and 0s elsewhere. The main block ensures the script's execution, obtaining the categorical data, performing one-hot encoding, and displaying the resulting encoded data.