**MACHINE LEARNING (22AIE213)**

**ASSIGNMENT-2**

**NAME: BANDI HARSHITHA REDDY**

**REG NO: AIE22108**

Question 1: 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.

**Pseudo Code:**

function euclidean\_distance(vector1, vector2):

if length of vector1 is not equal to length of vector2:

raise ValueError("Vectors must have the same dimension")

distance\_squared = sum of squares of differences between corresponding elements in vector1 and vector2

return square root of distance\_squared

function manhattan\_distance(vector1, vector2):

if length of vector1 is not equal to length of vector2:

raise ValueError("Vectors must have the same dimension")

distance = sum of absolute differences between corresponding elements in vector1 and vector2

return distance

function get\_vector\_input():

input\_vector1 = input("Enter the first vector: ")

input\_vector2 = input("Enter the second vector: ")

vector1 = convert each element in input\_vector1 to an integer

vector2 = convert each element in input\_vector2 to an integer

return vector1, vector2

if \_\_name\_\_ == "\_\_main\_\_":

vector1, vector2 = get\_vector\_input()

euclidean\_dist = call euclidean\_distance function with vector1 and vector2

manhattan\_dist = call manhattan\_distance function with vector1 and vector2

print("Euclidean Distance:", euclidean\_dist)

print("Manhattan Distance:", manhattan\_dist)

**Explanation:**

Use the Euclidean Distance Calculation to determine whether the vectors are the same size.

Summing the squares of the differences between matching elements in the two vectors will give you the squared Euclidean distance.

Give back the square root of the length squared.

Manhattan Distance Estimator:

Verify that the vectors are the same size.

Summing the absolute differences between matching elements in the two vectors will give you the Manhattan distance.

Give back the Manhattan distance that was calculated.

Obtaining Vector Data:

Request that the user input the elements of two vectors.

Create vectors by converting the input values to integers.

Primary Program:

Obtain user input vectors.

Question 2:Write a function to implement k-NN classifier. k is a variable and based on that the count of neighbors should be selected.

**Pseudo Code:**

Function k\_nearest\_neighbors(train\_data, test\_instance, k):

distances = []

// Step 1: Calculate distances from test\_instance to each point in train\_data

for each train\_point in train\_data:

distance = euclidean\_distance(test\_instance, train\_point[:-1]) // Last element is the class label

distances.append((distance, train\_point))

// Step 2: Sort distances and get the k nearest neighbors

k\_nearest = heapq.nsmallest(k, distances)

// Step 3: Get the class labels of the k nearest neighbors

neighbor\_labels = [neighbor[-1] for \_, neighbor in k\_nearest]

// Step 4: Count the occurrences of each class label

label\_counts = Counter(neighbor\_labels)

// Step 5: Return the class label with the highest count

return class label with highest count from label\_counts

**Explanation:**

train\_data: The training dataset made up of the relevant class labels and characteristics.

test\_instance: The instance for which a projected class label is required.

k: The quantity of closest neighbors to take into account.

Determine Distances: Go through every point in the train\_data (training dataset) iteratively. Determine the Euclidean distance (with the exception of the final element, the class label) between the test\_instance and each point's features. Keep the point and the distance in your file.

To find the k nearest neighbors, sort the distances and choose the closest ones.

Extract Class Labels: Extract the k closest neighbors' class labels.

Class Label Count: Determine how many times each class label appears among the neighbors.

Predict Class Label: Give back the neighboring class label with the highest count.

Question 3: Write a function to convert categorical variables to numeric using label encoding. Don’t use any existing functionalities.

**Pseudo Code:**

Function label\_encode\_categorical(data):

unique\_categories = sorted(set(data))

label\_encoding = {}

for index, category in enumerate(unique\_categories):

label\_encoding[category] = index

return label\_encoding

Function apply\_label\_encoding(data, label\_encoding):

encoded\_data = []

for category in data:

encoded\_data.append(label\_encoding[category])

return encoded\_data

Main program:

categorical\_data = ["red", "blue", "green", "red", "green", "blue", "yellow"]

encoding = label\_encode\_categorical(categorical\_data)

numeric\_labels = apply\_label\_encoding(categorical\_data, encoding)

print("Label Encoding:", encoding)

print("Numeric Labels:", numeric\_labels)

**Explanation:**

Using the label\_encode\_categorical function, distinct categories from the input data are mapped to distinct numeric labels in a dictionary. This guarantees uniform encoding amongst datasets.

Apply\_label\_encoding uses the label encoding dictionary's mapping to replace each category in the incoming data with its matching numeric label.

These functions are shown in the main program by encoding a sample categorical dataset and printing the numeric labels that are produced after the encoding process, along with the label encoding dictionary that results. This demonstrates the practical application of the encoding functions.Question 4: .Write a function to convert categorical variables to numeric using One-Hot encoding. Don’t use any existing functionalities.

**Pseudo Code:**

function get\_unique\_categories(data):

return sorted(set(data))

function one\_hot\_encode\_categorical(data, unique\_categories):

encoded\_data = []

for category in data:

encoded\_vector = list of zeros with length equal to length of unique\_categories

index = index of category in unique\_categories

set the value at index in encoded\_vector to 1

append encoded\_vector to encoded\_data

return encoded\_data

if \_\_name\_\_ == "\_\_main\_\_":

categorical\_data = ["red", "blue", "green", "red", "green", "blue", "yellow"]

unique\_categories = call get\_unique\_categories function with categorical\_data

one\_hot\_encoded\_data = call one\_hot\_encode\_categorical function with categorical\_data and unique\_categories

print("Unique Categories:", unique\_categories)

print("One-Hot Encoded Data:")

for category, encoded\_vector in zip(categorical\_data, one\_hot\_encoded\_data):

print(f"{category}: {encoded\_vector}")

**Explanation:**

(data) get\_unique\_categories:

accepts a categorical data list.

provides a sorted list of distinct categories back.

data, unique categories; one\_hot\_encode\_categorical:

requires a list of unique categories and a list of categorical data.

gives back a set of lists for every categorical value that represent one-hot encoded vectors.

Primary Program:

makes use of a categorical data sample list.

obtains unique categories and encrypts data one-hot.

prints distinct categories along with the one-hot encoded vectors that go with them.