**Machine Learning Lab : List of Programs**

1)Visualize the n-dimensional data using Scatter plots. Write a program to implement Hill Climbing Algorithm.

2)Visualize the n-dimensional data using 3D surface plots. Write a program to implement the Best First Search (BFS) algorithm.

3)Visualize the n-dimensional data using contour plots. Write a program to implement the A\* algorithm

4)Visualize the n-dimensional data using heat-map. Write a program to implement Min-Max algorithm.

5)Visualize the n-dimensional data using Box-plot. Write a program to implement Alpha-beta pruning algorithm.

6)Write a program to develop the Naive Bayes classifier on Titanic dataset.

7)Write a program to develop the KNN classifier with Euclidean distance and Manhattan distance for the k values as 3 based on split up of training and testing dataset as 70-30 on Glass dataset.

8)Write a program to perform unsupervised K-means clustering techniques on Iris dataset.

9)Write a program to perform agglomerative clustering based on single-linkage, complete linkage criteria.

10) Write a program to develop Principal Component Analysis (PCA) and Linear Discriminant Analysis (LDA) algorithms.

11)Write a Program to develop simple single layer perceptron to implement AND, OR Boolean functions.

1)

# =============================

# Part 1: Visualize N-Dimensional Data (Pairwise Scatter Plots)

# =============================

import pandas as pd

import seaborn as sns

import matplotlib.pyplot as plt

from sklearn.preprocessing import LabelEncoder

# Load and prepare the dataset

df = pd.read\_csv('ToyotaCorolla.csv')[['Price', 'Age\_08\_04', 'KM', 'HP', 'Weight', 'Fuel\_Type']]

# Encode categorical Fuel\_Type

df['Fuel\_Type'] = LabelEncoder().fit\_transform(df['Fuel\_Type'])

# Pairplot to visualize relationships

sns.pairplot(df, hue='Fuel\_Type')

plt.suptitle("Pairwise Scatter Plot of Car Features", y=1.02)

plt.show()

# =============================

# Part 2: Hill Climbing Algorithm (with Visualization)

# =============================

import numpy as np

import matplotlib.pyplot as plt

# Define the function

def f(x):

return np.sin(x) + np.cos(3 \* x)

# Hill Climbing Function

def hill\_climb(start, step=0.1, max\_iter=100):

x = start

path = [x]

for \_ in range(max\_iter):

curr, left, right = f(x), f(x - step), f(x + step)

if left > curr:

x -= step

elif right > curr:

x += step

else:

break

path.append(x)

return x, f(x), path

# Run hill climbing

best\_x, best\_y, path = hill\_climb(start=-5)

# Plotting

x\_vals = np.linspace(-10, 10, 400)

y\_vals = f(x\_vals)

plt.plot(x\_vals, y\_vals, label='f(x)')

plt.plot(path, [f(x) for x in path], 'ro--', label='Hill Climb Path')

plt.title("Hill Climbing on f(x)")

plt.xlabel("x")

plt.ylabel("f(x)")

plt.legend()

plt.grid(True)

plt.show()

print(f"Best x: {best\_x:.2f}, Best f(x): {best\_y:.2f}")

2)

*# Visualize the n-dimensional data using 3D surface plots.*

**import** pandas **as** pd

**import** numpy **as** np

**import** matplotlib.pyplot **as** plt

**import** seaborn **as** sns

data **=** pd**.**read\_csv("ToyotaCorolla.csv")

*#3d surface plot*

x **=** data['KM']

y **=** data['Doors']

z **=** data['Price']

ax **=** plt**.**axes(projection**=**'3d')

ax**.**plot\_trisurf(x,y,z,cmap**=**"jet")

ax**.**set\_title("3D Surface Plot")

plt**.**show()

// BFS

from queue import PriorityQueue

def best\_first\_search(graph, start, goal):

visited = set()

pq = PriorityQueue()

pq.put((0, start))

while not pq.empty():

\_, current = pq.get()

if current == goal:

print(f"Goal {goal} found!")

return

if current not in visited:

print(f"Visiting: {current}")

visited.add(current)

for neighbor, cost in graph[current]:

pq.put((cost, neighbor))

# Example graph: (node: [(neighbor, cost)])

graph = {

'A': [('B', 1), ('C', 3)],

'B': [('D', 1), ('E', 5)],

'C': [('F', 2)],

'D': [],

'E': [('G', 1)],

'F': [('G', 2)],

'G': []

}

best\_first\_search(graph, 'A', 'G')

3)

*#Visualize the n-dimensional data using contour plots.*

**import** pandas **as** pd

**import** numpy **as** np

**import** matplotlib.pyplot **as** plt

**import** seaborn **as** sns

data **=** pd**.**read\_csv("ToyotaCorolla.csv")

*#contour plot*

x **=** data['KM']

y **=** data['Weight']

z **=** data['Price']

plt**.**tricontourf(x, y, z, levels**=**20, cmap**=**'jet')

plt**.**colorbar(label**=**'Price')

plt**.**xlabel('KM')

plt**.**ylabel('Weight')

plt**.**title('Contour Plot')

plt**.**show()

from queue import PriorityQueue

def a\_star\_search(graph, start, goal, heuristic):

pq = PriorityQueue()

pq.put((0 + heuristic[start], 0, start, [])) # f(n), g(n), node, path

visited = set()

while not pq.empty():

f, g, node, path = pq.get()

path = path + [node]

if node == goal:

print(f"Path: {' -> '.join(path)}, Cost: {g}")

return

if node in visited:

continue

visited.add(node)

for neighbor, cost in graph[node]:

if neighbor not in visited:

total\_cost = g + cost

pq.put((total\_cost + heuristic[neighbor], total\_cost, neighbor, path))

# Example

graph = {

'A': [('B', 1), ('C', 4)],

'B': [('D', 2), ('E', 5)],

'C': [('F', 3)],

'D': [],

'E': [('G', 2)],

'F': [('G', 2)],

'G': []

}

heuristic = {'A': 7, 'B': 6, 'C': 5, 'D': 4, 'E': 2, 'F': 3, 'G': 0}

a\_star\_search(graph, 'A', 'G', heuristic)

4)

import seaborn as sns

import pandas as pd

import matplotlib.pyplot as plt

# Sample data

data = pd.DataFrame({

'Math': [90, 80, 70, 60],

'Physics': [85, 78, 72, 60],

'Chemistry': [88, 82, 75, 65]

})

sns.heatmap(data.corr(), annot=True, cmap='coolwarm')

plt.title("Correlation Heatmap")

plt.show()

# MIN-MAX ALGO:

def minmax(depth, isMax):

if depth == 0:

return 0 # Assume draw

if isMax:

return max(minmax(depth-1, False), minmax(depth-2, False))

else:

return min(minmax(depth-1, True), minmax(depth-2, True))

score = minmax(4, True)

print("Min-Max result:", score)

5)

BOX PLOT:

import seaborn as sns

import matplotlib.pyplot as plt

import pandas as pd

# Sample data

df = pd.DataFrame({

'Subject': ['Math', 'Math', 'Physics', 'Physics'],

'Marks': [90, 70, 85, 60]

})

sns.boxplot(x='Subject', y='Marks', data=df)

plt.title("Boxplot of Subject Marks")

plt.show()

#ALPHA-BETA:

**def** alpha\_beta(depth, node\_index, is\_max, scores, alpha, beta, height):

**if** depth **==** height:

**return** scores[node\_index]

**if** is\_max:

best **=** float('-inf')

**for** i **in** range(2):

val **=** alpha\_beta(depth **+** 1, node\_index **\*** 2 **+** i, **False**, scores, alpha, beta, height)

best **=** max(best, val)

alpha **=** max(alpha, best)

*# Prune if beta cutoff*

**if** beta **<=** alpha:

**break**

**return** best

**else**:

best **=** float('inf')

**for** i **in** range(2):

val **=** alpha\_beta(depth **+** 1, node\_index **\*** 2 **+** i, **True**, scores, alpha, beta, height)

best **=** min(best, val)

beta **=** min(beta, best)

*# Prune if alpha cutoff*

**if** beta **<=** alpha:

**break**

**return** best

*# Inputs*

scores **=** [3, 5, 6, 9, 1, 2, 0, **-**1]

height **=** 3

*# Output*

print("Alpha-Beta value:", alpha\_beta(0, 0, **True**, scores, float('-inf'), float('inf'), height))

6)

# Write a program to develop the Naive Bayes classifier on Titanic dataset.

import pandas as pd

from sklearn.model\_selection import train\_test\_split

from sklearn.naive\_bayes import GaussianNB

from sklearn.metrics import accuracy\_score, confusion\_matrix

df = pd.read\_csv("titanic.csv")

df = df[['Pclass', 'Sex', 'Age', 'SibSp', 'Parch', 'Fare', 'Survived']]

df['Age'] = df['Age'].fillna(df['Age'].median())

df['Sex'] = df['Sex'].map({'male': 0, 'female': 1})

X = df.drop('Survived', axis=1)

y = df['Survived']

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.1, random\_state=42)

model = GaussianNB()

model.fit(X\_train, y\_train)

y\_pred = model.predict(X\_test)

accuracy = accuracy\_score(y\_test, y\_pred)

print("Naive Bayes Classifier Accuracy on Titanic dataset: {:.2f}%".format(accuracy \* 100))

cm = confusion\_matrix(y\_test, y\_pred)

print("Confusion Matrix:")

print(cm)

7) KNN:

import pandas as pd

from sklearn.model\_selection import train\_test\_split

from sklearn.neighbors import KNeighborsClassifier

from sklearn.metrics import accuracy\_score, classification\_report

from sklearn.preprocessing import StandardScaler

# Load dataset

data = pd.read\_csv("glass.csv")

# Separate features and target

X = data.drop(columns=['Type']) # all features except 'Type'

y = data['Type']

# Standardize the features

scaler = StandardScaler()

X\_scaled = scaler.fit\_transform(X)

# Split data: 70% training, 30% testing

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X\_scaled, y, test\_size=0.3, random\_state=42)

# Set k = 3

k = 3

# Try both distance metrics

for p\_val, metric\_name in [(2, 'Euclidean'), (1, 'Manhattan')]:

model = KNeighborsClassifier(n\_neighbors=k, metric='minkowski', p=p\_val)

model.fit(X\_train, y\_train)

y\_pred = model.predict(X\_test)

print(f"\nK = {k}, Distance Metric = {metric\_name}")

print("Accuracy:", accuracy\_score(y\_test, y\_pred))

print("Classification Report:\n", classification\_report(y\_test, y\_pred, zero\_division=0))

11)

import numpy as np

class Perceptron:

def \_\_init\_\_(self, input\_size, lr=0.1, epochs=10):

self.weights = np.zeros(input\_size + 1) # bias + weights

self.lr = lr

self.epochs = epochs

def activation\_fn(self, x):

# Step function

return 1 if x >= 0 else 0

def predict(self, X):

x\_with\_bias = np.insert(X, 0, 1) # add bias term

weighted\_sum = np.dot(self.weights, x\_with\_bias)

return self.activation\_fn(weighted\_sum)

def fit(self, X, y):

for \_ in range(self.epochs):

for inputs, label in zip(X, y):

prediction = self.predict(inputs)

error = label - prediction

# update weights and bias

self.weights[1:] += self.lr \* error \* inputs

self.weights[0] += self.lr \* error

X = np.array([[0,0], [0,1], [1,0], [1,1]])

y\_and = np.array([0, 0, 0, 1])

y\_or = np.array([0, 1, 1, 1])

print("Training perceptron for AND function")

perceptron\_and = Perceptron(input\_size=2, lr=0.1, epochs=10)

perceptron\_and.fit(X, y\_and)

for inputs in X:

print(f"Input: {inputs} Output: {perceptron\_and.predict(inputs)}")

print("\nTraining perceptron for OR function")

perceptron\_or = Perceptron(input\_size=2, lr=0.1, epochs=10)

perceptron\_or.fit(X, y\_or)

for inputs in X:

print(f"Input: {inputs} Output: {perceptron\_or.predict(inputs)}")

// OR

import numpy as np

# Define the Sigmoid activation function and its derivative (for backpropagation)

def sigmoid(x):

return 1 / (1 + np.exp(-x))

class Perceptron:

def \_init\_(self, input\_size):

# Initialize the weights with random values and a bias term

self.weights = np.random.rand(input\_size) # Random initialization of weights

self.bias = np.random.rand(1) # Random bias initialization

def forward(self, inputs):

# Weighted sum (dot product) + bias

total\_input = np.dot(inputs, self.weights) + self.bias

# Apply the activation function (sigmoid)

output = sigmoid(total\_input)

return output

def train(self, X, y, epochs=1000, learning\_rate=0.1):

# Training the perceptron with the perceptron learning rule

for epoch in range(epochs):

for i in range(X.shape[0]):

# Forward pass

output = self.forward(X[i])

# Calculate the error (difference between expected and predicted output)

error = y[i] - output

# Update the weights and bias using the perceptron learning rule

self.weights += learning\_rate \* error \* X[i]

self.bias += learning\_rate \* error

# AND and OR dataset

X\_and = np.array([[0, 0], [0, 1], [1, 0], [1, 1]]) # Input for AND/OR functions

y\_and = np.array([0, 0, 0, 1]) # Expected output for AND function

y\_or = np.array([0, 1, 1, 1]) # Expected output for OR function

# Create perceptron instances for AND and OR

perceptron\_and = Perceptron(input\_size=2)

perceptron\_or = Perceptron(input\_size=2)

# Train the perceptrons

perceptron\_and.train(X\_and, y\_and, epochs=1000, learning\_rate=0.1)

perceptron\_or.train(X\_and, y\_or, epochs=1000, learning\_rate=0.1)

# Test the perceptrons

print("AND Function Predictions:")

for i in range(X\_and.shape[0]):

print(f"Input: {X\_and[i]} - Predicted Output: {round(perceptron\_and.forward(X\_and[i])[0])}")

print("\nOR Function Predictions:")

for i in range(X\_and.shape[0]):

print(f"Input: {X\_and[i]} - Predicted Output: {round(perceptron\_or.forward(X\_and[i])[0])}")