Exp 01

```python
# A* Algorithm

class Graph:
    def __init__(self, adjac_lis):
        self.adjac_lis = adjac_lis

    def get_neighbours(self, v):
        return self.adjac_lis[v]

    def h(self, n):
        # Heuristic function H
        H = {'A': 1, 'B': 1, 'C': 1, 'D': 1}
        return H[n]

    def a_star_algorithm(self, start, stop):
        # Initialize an open set with the start node
        open_lst = set([start])
        # Initialize a closed set as empty
        closed_lst = set([])
        # Initialize a dictionary to store the distance from start to each node
        dist = {}
        dist[start] = 0  # Distance from start to itself is 0
        # Initialize a dictionary to store predecessors for path reconstruction
        prenode = {}
        prenode[start] = start  # Predecessor of start is start itself

        while len(open_lst) > 0:  # Loop until the open set is not empty
            n = None  # Initialize n as None for now
            for v in open_lst:  # Loop through nodes in the open set
                # Update n if a shorter path to n is found among open nodes
                if n is None or dist[v] + self.h(v) < dist[n] + self.h(n):
                    n = v

            if n is None:  # If n is still None, no path is found
                print("Path does not exist")
                return None

            if n == stop:  # If the goal is reached, reconstruct the path
                reconst_path = []
                while prenode[n] != n:
                    reconst_path.append(n)
                    n = prenode[n]
                reconst_path.append(start)
                reconst_path.reverse()
                print("Path found: {}".format(reconst_path))
                return reconst_path

            for (m, weight) in self.get_neighbours(n):  # Loop through neighbors of node n
                # If neighbor not in open or closed set
                if m not in open_lst and m not in closed_lst:
                    open_lst.add(m)  # Add it to the open set
                    prenode[m] = n  # Set its predecessor to n
                    dist[m] = dist[n] + weight  # Update its distance from start
                else:
                    # If a shorter path to m is found
                    if dist[m] > dist[n] + weight:
                        dist[m] = dist[n] + weight  # Update the distance
                        prenode[m] = n  # Update its predecessor
                        # If m was in the closed set
                        if m in closed_lst:
                            closed_lst.remove(m)  # Remove it from the closed set
                            open_lst.add(m)  # Add it to the open set
            open_lst.remove(n)  # Remove n from the open set as it has been evaluated
            closed_lst.add(n)  # Add n to the closed set as it's fully evaluated

        print("Path does not exist")  # If the while loop ends without finding the goal, no path exists
        return None
```

```python
# AO* Algorithm

def Cost(H, condition, weight=1):
    # Initialize dictionary to store costs for paths
    cost = {}

    if 'AND' in condition:
        # Handle 'AND' condition
        AND_nodes = condition['AND']
        Path_A = ' AND '.join(AND_nodes)
        PathA = sum(H[node] + weight for node in AND_nodes)
        cost[Path_A] = PathA

    if 'OR' in condition:
        # Handle 'OR' condition
        OR_nodes = condition['OR']
        Path_B = ' OR '.join(OR_nodes)
        PathB = min(H[node] + weight for node in OR_nodes)
        cost[Path_B] = PathB

    return cost

def update_cost(H, Conditions, weight=1):
    # Get a list of nodes and reverse the order
    Main_nodes = list(Conditions.keys())
    Main_nodes.reverse()

    # Initialize dictionary to track the least cost for each node
    least_cost = {}
    for key in Main_nodes:
        # Get the condition associated with the current node
        condition = Conditions[key]
        # Display the current node and its condition
        print(key, ':', Conditions[key], '>>', Cost(H, condition, weight))

        # Calculate the cost for the current node's condition
        c = Cost(H, condition, weight)
        # Update the heuristic value of the current node to the minimum cost calculated
        H[key] = min(c.values())
        # Store the cost for the current node in the least_cost dictionary
        least_cost[key] = Cost(H, condition, weight)

    return least_cost

def shortest_path(Start, Updated_cost, H):
    # Initialize the path with the starting node
    Path = Start

    if Start in Updated_cost.keys():
        # Find the minimum cost associated with the starting node
        Min_cost = min(Updated_cost[Start].values())
        key = list(Updated_cost[Start].keys())
        values = list(Updated_cost[Start].values())
        Index = values.index(Min_cost)

        # Split the key into individual nodes or paths
        Next = key[Index].split()

        if len(Next) == 1:
            # If the length of Next is 1, it's a single node or path
            Start = Next[0]
            # Recursively find the shortest path
            Path += '<--' + shortest_path(Start, Updated_cost, H)

        else:
            # If the length of Next is more than 1, it represents multiple nodes or paths
            Path += '<--(' + key[Index] + ') '
            Start = Next[0]
            # Recursively find the shortest path for the AND path
            Path += '[' + shortest_path(Start, Updated_cost, H) + ' + '
            Start = Next[-1]
            # Recursively find the shortest path for the remaining path
            Path += shortest_path(Start, Updated_cost, H) + ']'

    return Path
```

```python
#Candidate-Elimination Algorithm

import numpy as np
import pandas as pd

# Loading Data from a CSV File
data = pd.DataFrame(data=pd.read_csv('trainingdata.csv'))
print(data)

# Separating concept features from Target concepts
concepts = np.array(data.iloc[:, 0:-1])
print(concepts)

# Isolating target into a separate DataFrame
# copying last column to target array
target = np.array(data.iloc[:, -1])
print(target)

def learn(concepts, target):
    # Initialise S0 with the first instance from concepts
    # .copy() makes sure a new list is created instead of just pointing to the same memory location
    specific_h = concepts[0].copy()
    print("\nInitialization of specific_h and general_h")
    print(specific_h)

    general_h = [["?" for i in range(len(specific_h))] for i in range(len(specific_h))]
    print(general_h)

    # The learning iterations
    for i, h in enumerate(concepts):
        # Checking if the hypothesis has a positive target
        if target[i] == "Yes":
            for x in range(len(specific_h)):
                # Change values in S & G only if values change
                if h[x] != specific_h[x]:
                    specific_h[x] = '?'
                    general_h[x][x] = '?'

        # Checking if the hypothesis has a negative target
        if target[i] == "No":
            for x in range(len(specific_h)):
                # For negative hypothesis change values only in G
                if h[x] != specific_h[x]:
                    general_h[x][x] = specific_h[x]
                else:
                    general_h[x][x] = '?'

        print("\nSteps of Candidate Elimination Algorithm", i + 1)
        print(specific_h)
        print(general_h)

    # find indices where we have empty rows, meaning those that are unchanged
    indices = [i for i, val in enumerate(general_h) if val == ['?', '?', '?', '?', '?', '?']]
    for i in indices:
        # remove those rows from general_h
        general_h.remove(['?', '?', '?', '?', '?', '?'])

    # Return final values
    return specific_h, general_h

# Apply the learn function to get the final specific_h and general_h
s_final, g_final = learn(concepts, target)

# Display the final specific_h and general_h
print("\nFinal Specific_h:", s_final, sep="\n")
print("\nFinal General_h:", g_final, sep="\n")
```

```python
#ID3 Algorithm

import numpy as np
import math
import csv

# Function to read data from a CSV file
def read_data(filename):
    with open(filename, 'r') as csvfile:
        datareader = csv.reader(csvfile, delimiter=',')
        headers = next(datareader)  # Read the header row
        metadata = []  # List to hold column names
        traindata = []  # List to hold training data
        for name in headers:
            metadata.append(name)  # Store column names in metadata list
        for row in datareader:
            traindata.append(row)  # Store rows of training data
        return metadata, traindata  # Return metadata and training data as tuples

# Node class for building the decision tree
class Node:
    def __init__(self, attribute):
        self.attribute = attribute  # Attribute name for the node
        self.children = []  # List to hold child nodes
        self.answer = ""  # Holds the final classification answer

    def __str__(self):
        return self.attribute  # Returns the attribute name as a string

# Function to create subtables based on column values
def subtables(data, col, delete):
    dict = {}  # Dictionary to hold subtables
    items = np.unique(data[:, col])  # Unique values in the column
    count = np.zeros((items.shape[0], 1), dtype=np.int32)  # Count occurrences of each value

    # Populate subtables based on unique values
    for x in range(items.shape[0]):
        for y in range(data.shape[0]):
            if data[y, col] == items[x]:
                count[x] += 1

    # Fill the dictionary with subtables corresponding to each unique value
    for x in range(items.shape[0]):
        dict[items[x]] = np.empty((int(count[x]), data.shape[1]), dtype="|S32")
        pos = 0
        for y in range(data.shape[0]):
            if data[y, col] == items[x]:
                dict[items[x]][pos] = data[y]
                pos += 1

        if delete:
            dict[items[x]] = np.delete(dict[items[x]], col, 1)  # Delete the column if needed

    return items, dict

# Function to calculate entropy for a set
def entropy(S):
    items = np.unique(S)  # Unique values in the set
    if items.size == 1:  # If only one unique value, entropy is 0
        return 0
    counts = np.zeros((items.shape[0], 1))  # Count occurrences of each unique value
    sums = 0
    # Calculate entropy using the formula and counts
    for x in range(items.shape[0]):
        counts[x] = sum(S == items[x]) / (S.size * 1.0)
    for count in counts:
        sums += -1 * count * math.log(count, 2)  # Entropy formula
    return sums

# Function to calculate gain ratio for a column
def gain_ratio(data, col):
    items, dict = subtables(data, col, delete=False)  # Get subtables for the column
    total_size = data.shape[0]  # Total size of the data

    entropies = np.zeros((items.shape[0], 1))  # Array to hold entropies
```

```
    intrinsic = np.zeros((items.shape[0], 1))  # Array to hold intrinsic information
    # Calculate entropies and intrinsic information
    for x in range(items.shape[0]):
        ratio = dict[items[x]].shape[0] / (total_size * 1.0)
        entropies[x] = ratio * entropy(dict[items[x]][:, -1])  # Entropy for each subtable
        intrinsic[x] = ratio * math.log(ratio, 2)  # Intrinsic information
    total_entropy = entropy(data[:, -1])  # Total entropy of the entire set
    iv = -1 * sum(intrinsic)  # Calculate intrinsic value
    # Calculate gain ratio using entropy and intrinsic value
    for x in range(entropies.shape[0]):
        total_entropy -= entropies[x]
    return total_entropy / iv

# Function to create nodes in the decision tree
def create_node(data, metadata):
    if (np.unique(data[:, -1])).shape[0] == 1:
        node = Node("")
        node.answer = np.unique(data[:, -1])[0]  # Store the answer if only one class remains
        return node
    gains = np.zeros((data.shape[1] - 1, 1))  # Array to hold gain ratios for each column

    # Calculate gain ratio for each column
    for col in range(data.shape[1] - 1):
        gains[col] = gain_ratio(data, col)
    split = np.argmax(gains)  # Find the column with the highest gain ratio
    node = Node(metadata[split])  # Create a node with the split attribute
    metadata = np.delete(metadata, split, 0)  # Remove the split attribute from metadata
    items, dict = subtables(data, split, delete=True)  # Get subtables based on the split attribute
    # Recursively create child nodes
    for x in range(items.shape[0]):
        child = create_node(dict[items[x]], metadata)
        node.children.append((items[x], child))  # Append child nodes to the current node
    return node  # Return the node

# Function to create indentation for tree visualization
def empty(size):
    s = ""
    for x in range(size):
        s += " "
    return s

# Function to print the decision tree
def print_tree(node, level):
    if node.answer != "":
        print(empty(level), node.answer)
        return
    print(empty(level), node.attribute)
    for value, n in node.children:
        print(empty(level + 1), value)
        print_tree(n, level + 2)

# Read data from a CSV file
metadata, traindata = read_data("tennisdata.csv")
data = np.array(traindata)

# Create the decision tree
node = create_node(data, metadata)
print_tree(node, 0)  # Print the decision tree
```

```python
#Backpropagation Algorithm

import numpy as np

X = np.array(([2, 9], [1, 5], [3, 6]), dtype=float)  # X = (hours sleeping, hours studying)
y = np.array(([92], [86], [89]), dtype=float)  # y = score on test

# scale units
X = X / np.amax(X, axis=0)  # maximum of X array
y = y / 100  # max test score is 100

class Neural_Network(object):
    def __init__(self):
        # Parameters
        self.inputSize = 2
        self.outputSize = 1
        self.hiddenSize = 3
        # Weights
        self.W1 = np.random.randn(self.inputSize, self.hiddenSize)  # (3x2) weight matrix from input to hidden layer
        self.W2 = np.random.randn(self.hiddenSize, self.outputSize)  # (3x1) weight matrix from hidden to output layer

    def forward(self, X):
        # forward propagation through our network
        self.z = np.dot(X, self.W1)  # dot product of X (input) and first set of 3x2 weights
        self.z2 = self.sigmoid(self.z)  # activation function
        self.z3 = np.dot(self.z2, self.W2)  # dot product of hidden layer (z2) and second set of 3x1 weights
        o = self.sigmoid(self.z3)  # final activation function
        return o

    def sigmoid(self, s):
        return 1 / (1 + np.exp(-s))  # activation function

    def sigmoidPrime(self, s):
        return s * (1 - s)  # derivative of sigmoid

    def backward(self, X, y, o):
        # backward propagate through the network
        self.o_error = y - o  # error in output
        self.o_delta = self.o_error * self.sigmoidPrime(o)  # applying derivative of sigmoid to output error
        self.z2_error = self.o_delta.dot(self.W2.T)  # z2 error: how much hidden layer weights contributed to output error
        self.z2_delta = self.z2_error * self.sigmoidPrime(self.z2)  # applying derivative of sigmoid to z2 error
        self.W1 += X.T.dot(self.z2_delta)  # adjusting first set (input --> hidden) weights
        self.W2 += self.z2.T.dot(self.o_delta)  # adjusting second set (hidden --> output) weights

    def train(self, X, y):
        o = self.forward(X)
        self.backward(X, y, o)

# Instantiate the neural network
NN = Neural_Network()

# Print initial state
print("\nInput: \n" + str(X))
print("\nActual Output: \n" + str(y))
print("\nPredicted Output: \n" + str(NN.forward(X)))
print("\nLoss: \n" + str(np.mean(np.square(y - NN.forward(X)))))  # mean sum squared loss)

# Train the neural network
NN.train(X, y)
```

```
# Naïve Bayesian classifier

import pandas as pd
from sklearn import tree
from sklearn.preprocessing import LabelEncoder
from sklearn.naive_bayes import GaussianNB

# load data from CSV
data = pd.read_csv('tennisdata.csv')
print("The first 5 values of data are:\n", data.head())

# obtain Train data and Train output
X = data.iloc[:, :-1]
print("\nThe First 5 values of train data are\n", X.head())
y = data.iloc[:, -1]
print("\nThe first 5 values of Train output are\n", y.head())

# Convert them to numbers
le_outlook = LabelEncoder()
X.Outlook = le_outlook.fit_transform(X.Outlook)
le_Temperature = LabelEncoder()
X.Temperature = le_Temperature.fit_transform(X.Temperature)
le_Humidity = LabelEncoder()
X.Humidity = le_Humidity.fit_transform(X.Humidity)
le_Windy = LabelEncoder()
X.Windy = le_Windy.fit_transform(X.Windy)
print("\nNow the Train data is:\n", X.head())

le_PlayTennis = LabelEncoder()
y = le_PlayTennis.fit_transform(y)
print("\nNow the Train output is\n", y)

from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.20)

# Create and train the Naive Bayes classifier
classifier = GaussianNB()
classifier.fit(X_train, y_train)

from sklearn.metrics import accuracy_score
print("Accuracy is:", accuracy_score(classifier.predict(X_test), y_test))
```

```python
# EM Algorithm

from sklearn.cluster import KMeans
from sklearn import preprocessing
from sklearn.mixture import GaussianMixture
from sklearn.datasets import load_iris
import sklearn.metrics as sm
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt

# Load the Iris dataset
dataset = load_iris()
X = pd.DataFrame(dataset.data)
X.columns = ['Sepal_Length', 'Sepal_Width', 'Petal_Length', 'Petal_Width']
y = pd.DataFrame(dataset.target)
y.columns = ['Targets']

# Create a colormap for visualization
plt.figure(figsize=(14, 7))
colormap = np.array(['red', 'lime', 'black'])

# Real Data Plot
plt.subplot(1, 3, 1)
plt.scatter(X.Petal_Length, X.Petal_Width, c=colormap[y.Targets], s=40)
plt.title('Real')

# K-Means Plot
plt.subplot(1, 3, 2)
model = KMeans(n_clusters=3)
model.fit(X)
predY = np.choose(model.labels_, [0, 1, 2]).astype(np.int64)
plt.scatter(X.Petal_Length, X.Petal_Width, c=colormap[predY], s=40)
plt.title('KMeans')

# Gaussian Mixture Model (GMM) Plot
scaler = preprocessing.StandardScaler()
scaler.fit(X)
xsa = scaler.transform(X)
xs = pd.DataFrame(xsa, columns=X.columns)
gmm = GaussianMixture(n_components=3)
gmm.fit(xs)
y_cluster_gmm = gmm.predict(xs)
plt.subplot(1, 3, 3)
plt.scatter(X.Petal_Length, X.Petal_Width, c=colormap[y_cluster_gmm], s=40)
plt.title('GMM Classification')

plt.show()
```

```
 #K-Means Algorithm

from sklearn.datasets import load_iris
from sklearn.neighbors import KNeighborsClassifier
from sklearn.model_selection import train_test_split
import numpy as np

# Load the Iris dataset
dataset = load_iris()

# Split the dataset into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(dataset["data"], dataset["target"], random_state=0)

# Create a K-Nearest Neighbors (KNN) classifier with k=1
kn = KNeighborsClassifier(n_neighbors=1)

# Train the KNN classifier on the training data
kn.fit(X_train, y_train)

# Make predictions on the test data and evaluate the accuracy
for i in range(len(X_test)):
    x = X_test[i]
    x_new = np.array([x])
    prediction = kn.predict(x_new)

    # Print the true target, predicted target, and their corresponding names
    print("TARGET=", y_test[i], dataset["target_names"][y_test[i]], "PREDICTED=", prediction, dataset["target_names"][prediction])

# Print the accuracy of the classifier on the test set
print("Accuracy:", kn.score(X_test, y_test))
```

Exp 09

```
# Locally Weighted Regression Algorithm

import numpy as np
import matplotlib.pyplot as plt

# Radial kernel function
def radial_kernel(x0, X, tau):
    return np.exp(np.sum((X - x0) ** 2, axis=1) / (-2 * tau * tau))

# Local regression function
def local_regression(x0, X, Y, tau):
    # Add bias term
    x0 = np.r_[1, x0]
    X = np.c_[np.ones(len(X)), X]

    # Fit model using normal equations with kernel
    xw = X.T * radial_kernel(x0, X, tau)
    beta = np.linalg.pinv(xw @ X) @ xw @ Y

    # Predict value
    return x0 @ beta

# Generate dataset
n = 1000
X = np.linspace(-3, 3, num=n)
Y = np.log(np.abs(X ** 2 - 1) + .5)

# Jitter X
X += np.random.normal(scale=.1, size=n)

# Domain space for predictions
domain = np.linspace(-3, 3, num=300)

# Function to plot LWR for different bandwidths (tau)
def plot_lwr(tau):
    # Predictions through regression
    predictions = [local_regression(x0, X, Y, tau) for x0 in domain]

    # Plot the dataset and the regression curve
    plt.scatter(X, Y, color='blue', alpha=0.3, s=20)
    plt.plot(domain, predictions, color='red', linewidth=3)
    plt.title(f'Locally Weighted Regression (tau={tau})')
    plt.show()

# Plotting LWR curves for different tau values
plot_lwr(10.)
plot_lwr(1.)
plot_lwr(0.1)
plot_lwr(0.01)
```