ARTIFICIAL Intelligence Lab: Lab-01

A* Search Algorithm:

⊛ cost fn $f(n) = g(n) + h(n)$ → $g(n)$ = path weight, $h(n)$ = Heuristic value.

open_list: contains All the nodes visited except neighbours visited which are not.

closed_list: contains nodes along with neighbours which are visited.

Algorithm:

1. Initialise start node 'n' and put it in an open list
2. Calculate cost function. $f(n) = g(n) + h(n)$
3. Remove from open list and put in closed list.
   save index of 'n' node with smallest 'f'.
4. Is 'n' target node?
   - Yes → Terminate the Algorithms and use pointers of nodes to get the optimal path.
   - No → Find successive nodes of 'n' which are not in closed list.
5. Find successive nodes of node 'n' which are not in closed list
6. Calculate 'f' for each these nodes.

Program:

```
def aStarAlgo(start_node, stop_node):
    open_set = set(start_node)
    closed_set = set()
    g = {}          # stores distance from starting node
    parents = {}    # It contains adjacency map of all nodes.

    g[start_node] = 0
    parents[start_node] = start_node

    while len(open_set) > 0:
        n = None
        for v in open_set:
            if n == None or g[v] + h(v) < g[n] + h(n):
                n = v
        if n == stop_node or Graph_nodes[n] == None:
            pass
        else:
            for (m, weight) in get_neighbours(n): } ctd.
```

```
if n== stop_node or Graph_nodes[n]==None:
    pass
else:
    for (m, weight) in get_neighbours(n):
        if m not in open_set and m not in closed_set:
            open_set.add(m)
            parents[m] = n
            g[m] = g[n] + weight

        else:
            if g[m] > g[n] + weight:
                g[m] = g[n] + weight
                parents[m] = n

                if m in closed_set:
                    closed_set.remove(m)
                    open_set.add(m)

if n==None:
    print("Path does' not exist!")
    return None

if n== stop_node:
    path = []
    while parents[n] != n:
        path.append(n)
        n = parents[n]
    path.append(start_node)
    path.reverse()
    print('Path found: {}'.format(path))
    return path

open_set.remove(n)
closed_set.add(n)

print('Path does not exist!')
return None
```

```
def get_neighbors(v):
    if v in Graph_nodes:
        return Graph_nodes[v]
    else:
        return None

def h(n):
    H_dist = {
    'A':10, 'B':8, 'C':5, 'D':7, 'E':3, 'F':6, 'G':5, 'H':3, 'I':1, 'J':0 }
    return H_dist[n]

Graph_nodes = {
'A': [('B', 6), ('F', 3)],
'B': [('C', 3), ('D', 2)],
'C': [('D', 1), ('E', 5)],
'D': [('C', 1), ('E', 8)],
'E': [('I', 5), ('J', 5)],
'F': [('G', 1), ('H', 7)],
'G': [('I', 3)],
'H': [('I', 2)],
'I': [('E', 5), ('J', 3)],

}
aStarAlgo('A', 'J')
```

CANDIDATE ELIMINATION [lab-03]

```
import csv
with open('training examples.csv') as f:
    csv_file = csv.reader(f)
    data = list(csv_file)
    specific = data[1][:-1]
    general = [['?' for i in range(len(specific))] for j in range(len(specific))]

    for i in data:
        if i[-1]=="Yes":
            for j in range(len(specific)):
                if i[j] != specific[j]:
                    specific[j] = "?"
                    general[j][j] = "?"
        elif i[-1] == "No":
            for j in range(len(specific)):
                if i[j] != specific[j]:
                    general[j][j] = specific[j]
                else:
                    general[j][j] = "?"
        print("\n Step " + str(data.index(i)+1) + " of CEA :")
        print(specific)
        print(general)

    gh=[]
    for i in general:
        for j in i:
            if j != '?':
                gh.append(i)
                break
    print("\n Final Specific Hypothesis: \n", specific)
    print("\n Final General Hypothesis: \n", gh)
```

Back Propagation Algorithm:

Chethan patel

Lab-05

```
import numpy as np

X = np.array(([2,9],[1,5],[3,6]), dtype=float)
y = np.array(([92],[86],[89]), dtype=float)

X = X/np.amax(X, axis=0)
y = y/100

def sigmoid(x):
    return 1/(1+np.exp(-x))

def derivatives_sigmoid(x):
    return x*(1-x)

epoch = 5000
lr = 0.1
inputlayer_neurons = 2
hiddenlayer_neurons = 3
outputlayer_neurons = 1

wh = np.random.uniform(size=(inputlayer_neurons, hiddenlayer_neurons))
bh = np.random.uniform(size=(1, hiddenlayer_neurons))
wout = np.random.uniform(size=(hiddenlayer_neurons, outputlayer_neurons))
bout = np.random.uniform(size=(1, outputlayer_neurons))

for i in range(epoch):
    #Forward propagation
    hinp1 = np.dot(X, wh)
    hinp = hinp1 + bh
    hlayer_act = sigmoid(hinp)
    outputt
    outinp1 = np.dot(hlayer_act, wout)
    outinp = outinp1 + bout
    output = sigmoid(outinp)
```

```
#Back propagation

    EO = y - output
    outgrad = derivatives_sigmoid(output)
    d_output = EO * outgrad

    EH = d_output.dot(wout.T)
    hiddengrad = derivatives_sigmoid(hlayer_act)
    d_hiddenlayer = EH * hiddengrad.

    wout += hlayer_act.T.dot(d_output) *lr
    wh += X.T.dot(d_hiddenlayer) *lr

print("Input: \n" + str(X))
print("output: \n" + str(y))
print("Predicted output: \n", output)
```

NAIVE BAYESIAN CLASSIFIER (Lab-6) [csv-66]

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

data = pd.read_csv('tennisdata.csv')
print("The First 5 values of the data is: \n", data.head())

X = data.iloc[:, :-1]
print("The Fist 5 values of training data is: \n", X.head())

y = data.iloc[:, -1]
print("The First 5 values of output of training data is: \n", y.head())

le_outlook = LabelEncoder()
X.Outlook = le_outlook.fit_transform(X.Outlook)

le_Temparature = LabelEncoder()
X.Temparature = le_Temparature.fit_transform(X.Temparature)

le_Humidity = LabelEncoder()
X.Humidity = le_Humidity.fit_transform(X.Humidity)

le_windy = LabelEncoder()
X.Windy = le_windy.fit_transform(X.windy)
print("\n Now the training output is\n", X.head())

le_PlayTennis = LabelEncoder()
X.PlayTennis y = le_playTennis.fit_transform(y)
print("\n Now the training data actual output is\n", y)
```

Lab-6 (Naive Bayesian classifier) {Y≈y}

Ctd...

```
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)

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))
```

Lab.9: K-Nearest Neighbor: (Iris data).

Lab-08

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

dataset = load_iris()
print(dataset)
x_train, x_test, y_train, y_test = train_test_split(
        dataset["data"], dataset["target"], random_state=0)

kn = KNeighborsClassifier(n_neighbors=1)
kn.fit(x_train, y_train)

for i in range(len(x_test)):
        x = x_test[i]
        x_new = np.array([x])
        prediction = kn.predict(x_new)
        print("TARGET:", y_test[i],
                dataset["target_names"][y_test[i]],
                "PREDICTED:" prediction,
                dataset["target_names"][prediction])

print(kn.score(x_test, y_test))
```

Lab-09: Locally Weighted Regression

```
import numpy as np
import matplotlib.pyplot as plt

def local_regression(x0, X, Y, tau):
    x0 = [1, x0]
    X = [[1, i] for i in X]
    X = np.asarray(X)
    xw = (X.T) * np.exp(np.sum(X - x0) ** 2, axis = 1) / (-2 * tau))
    Beta = np.linalg.pinv(xw @ X) @ xw @ Y @ x0
    return beta

def draw(tau):
    prediction = [local_regression(x0, X, Y, tau) for x0 in domain]
    plt.plot(domain, prediction, color = 'red')
    plt.plot(X, Y, 'o', color = 'black')
    plt.plot(domain, prediction, color = 'red')
    plt.show()

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

draw(10)
draw(0.1)
draw(0.01)
draw(0.001)
```