# **DAY 3**

**DAY 3 ( TIC – TAC - TOE PROBLEM ) :**

**import os**

**import time**

**board = [' ',' ',' ',' ',' ',' ',' ',' ',' ',' ']**

**player = 1**

**########win Flags##########**

**Win = 1**

**Draw = -1**

**Running = 0**

**Stop = 1**

**###########################**

**Game = Running**

**Mark = 'X'**

**#This Function Draws Game Board**

**def DrawBoard():**

**print(" %c | %c | %c " % (board[1],board[2],board[3]))**

**print("\_\_\_|\_\_\_|\_\_\_")**

**print(" %c | %c | %c " % (board[4],board[5],board[6]))**

**print("\_\_\_|\_\_\_|\_\_\_")**

**print(" %c | %c | %c " % (board[7],board[8],board[9]))**

**print(" | | ")**

**#This Function Checks position is empty or not**

**def CheckPosition(x):**

**if(board[x] == ' '):**

**return True**

**else:**

**return False**

**#This Function Checks player has won or not**

**def CheckWin():**

**global Game**

**#Horizontal winning condition**

**if(board[1] == board[2] and board[2] == board[3] and board[1] != ' '):**

**Game = Win**

**elif(board[4] == board[5] and board[5] == board[6] and board[4] != ' '):**

**Game = Win**

**elif(board[7] == board[8] and board[8] == board[9] and board[7] != ' '):**

**Game = Win**

**#Vertical Winning Condition**

**elif(board[1] == board[4] and board[4] == board[7] and board[1] != ' '):**

**Game = Win**

**elif(board[2] == board[5] and board[5] == board[8] and board[2] != ' '):**

**Game = Win**

**elif(board[3] == board[6] and board[6] == board[9] and board[3] != ' '):**

**Game=Win**

**#Diagonal Winning Condition**

**elif(board[1] == board[5] and board[5] == board[9] and board[5] != ' '):**

**Game = Win**

**elif(board[3] == board[5] and board[5] == board[7] and board[5] != ' '):**

**Game=Win**

**#Match Tie or Draw Condition**

**elif(board[1]!=' ' and board[2]!=' ' and board[3]!=' ' and board[4]!=' ' and board[5]!=' ' and board[6]!=' ' and board[7]!=' ' and board[8]!=' ' and board[9]!=' '):**

**Game=Draw**

**else:**

**Game=Running**

**print("Tic-Tac-Toe Game Designed By Sourabh Somani")**

**print("Player 1 [X] --- Player 2 [O]\n")**

**print()**

**print()**

**print("Please Wait...")**

**time.sleep(3)**

**while(Game == Running):**

**os.system('cls')**

**DrawBoard()**

**if(player % 2 != 0):**

**print("Player 1's chance")**

**Mark = 'X'**

**else:**

**print("Player 2's chance")**

**Mark = 'O'**

**choice = int(input("Enter the position between [1-9] where you want to mark : "))**

**if(CheckPosition(choice)):**

**board[choice] = Mark**

**player+=1**

**CheckWin()**

**os.system('cls')**

**DrawBoard()**

**if(Game==Draw):**

**print("Game Draw")**

**elif(Game==Win):**

**player-=1**

**if(player%2!=0):**

**print("Player 1 Won")**

**else:**

**print("Player 2 Won")**

**DAY 3 ( MIN – MAX PROBLEM** ) :

function minimax(node, depth, isMaximizingPlayer, alpha, beta):

if node is a leaf node :

return value of the node

if isMaximizingPlayer :

bestVal = -INFINITY

for each child node :

value = minimax(node, depth+1, false, alpha, beta)

bestVal = max( bestVal, value)

alpha = max( alpha, bestVal)

if beta <= alpha:

break

return bestVal

else :

bestVal = +INFINITY

for each child node :

value = minimax(node, depth+1, true, alpha, beta)

bestVal = min( bestVal, value)

beta = min( beta, bestVal)

if beta <= alpha:

break

return bestVal

DAY 3 ( ALPHA – BETA PROGRAM ) :

**# Player 'O' is max, in this case AI**

**def max(self):**

**# Possible values for maxv are:**

**# -1 - loss**

**# 0 - a tie**

**# 1 - win**

**# We're initially setting it to -2 as worse than the worst case:**

**maxv = -2**

**px = None**

**py = None**

**result = self.is\_end()**

**# If the game came to an end, the function needs to return**

**# the evaluation function of the end. That can be:**

**# -1 - loss**

**# 0 - a tie**

**# 1 - win**

**if result == 'X':**

**return (-1, 0, 0)**

**elif result == 'O':**

**return (1, 0, 0)**

**elif result == '.':**

**return (0, 0, 0)**

**for i in range(0, 3):**

**for j in range(0, 3):**

**if self.current\_state[i][j] == '.':**

**# On the empty field player 'O' makes a move and calls Min**

**# That's one branch of the game tree.**

**self.current\_state[i][j] = 'O'**

**(m, min\_i, min\_j) = self.min()**

**# Fixing the maxv value if needed**

**if m > maxv:**

**maxv = m**

**px = i**

**py = j**

**# Setting back the field to empty**

**self.current\_state[i][j] = '.'**

**return (maxv, px, py)**

**DAY 3 ( DECISION TREE PROBLEM )**

**# Run this program on your local python**

**# interpreter, provided you have installed**

**# the required libraries.**

**# Importing the required packages**

**import numpy as np**

**import pandas as pd**

**from sklearn.metrics import confusion\_matrix**

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

**from sklearn.tree import DecisionTreeClassifier**

**from sklearn.metrics import accuracy\_score**

**from sklearn.metrics import classification\_report**

**# Function importing Dataset**

**def importdata():**

**balance\_data = pd.read\_csv(**

**'https://archive.ics.uci.edu/ml/machine-learning-'+**

**'databases/balance-scale/balance-scale.data',**

**sep= ',', header = None)**

**# Printing the dataswet shape**

**print ("Dataset Length: ", len(balance\_data))**

**print ("Dataset Shape: ", balance\_data.shape)**

**# Printing the dataset obseravtions**

**print ("Dataset: ",balance\_data.head())**

**return balance\_data**

**# Function to split the dataset**

**def splitdataset(balance\_data):**

**# Separating the target variable**

**X = balance\_data.values[:, 1:5]**

**Y = balance\_data.values[:, 0]**

**# Splitting the dataset into train and test**

**X\_train, X\_test, y\_train, y\_test = train\_test\_split(**

**X, Y, test\_size = 0.3, random\_state = 100)**

**return X, Y, X\_train, X\_test, y\_train, y\_test**

**# Function to perform training with giniIndex.**

**def train\_using\_gini(X\_train, X\_test, y\_train):**

**# Creating the classifier object**

**clf\_gini = DecisionTreeClassifier(criterion = "gini",**

**random\_state = 100,max\_depth=3, min\_samples\_leaf=5)**

**# Performing training**

**clf\_gini.fit(X\_train, y\_train)**

**return clf\_gini**

**# Function to perform training with entropy.**

**def tarin\_using\_entropy(X\_train, X\_test, y\_train):**

**# Decision tree with entropy**

**clf\_entropy = DecisionTreeClassifier(**

**criterion = "entropy", random\_state = 100,**

**max\_depth = 3, min\_samples\_leaf = 5)**

**# Performing training**

**clf\_entropy.fit(X\_train, y\_train)**

**return clf\_entropy**

**# Function to make predictions**

**def prediction(X\_test, clf\_object):**

**# Predicton on test with giniIndex**

**y\_pred = clf\_object.predict(X\_test)**

**print("Predicted values:")**

**print(y\_pred)**

**return y\_pred**

**# Function to calculate accuracy**

**def cal\_accuracy(y\_test, y\_pred):**

**print("Confusion Matrix: ",**

**confusion\_matrix(y\_test, y\_pred))**

**print ("Accuracy : ",**

**accuracy\_score(y\_test,y\_pred)\*100)**

**print("Report : ",**

**classification\_report(y\_test, y\_pred))**

**# Driver code**

**def main():**

**# Building Phase**

**data = importdata()**

**X, Y, X\_train, X\_test, y\_train, y\_test = splitdataset(data)**

**clf\_gini = train\_using\_gini(X\_train, X\_test, y\_train)**

**clf\_entropy = tarin\_using\_entropy(X\_train, X\_test, y\_train)**

**# Operational Phase**

**print("Results Using Gini Index:")**

**# Prediction using gini**

**y\_pred\_gini = prediction(X\_test, clf\_gini)**

**cal\_accuracy(y\_test, y\_pred\_gini)**

**print("Results Using Entropy:")**

**# Prediction using entropy**

**y\_pred\_entropy = prediction(X\_test, clf\_entropy)**

**cal\_accuracy(y\_test, y\_pred\_entropy)**

**# Calling main function**

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

**main()**