Q1) Give a formal description of this application in terms of Task, Experience, and Performance. (Make it a well posed problem)

TASK : Identify best possible next step in the Tic Tac Toe game.

EXPERIENCE : Past game data

PERFORMANCE : Number of games won

Q2) Write a function Initialize() that initializes the board state for tic-tac-toe. Board can be represented as 3x3 matrix in python.

```
import numpy as np
import copy


def create_board():
    """
    This function creates a empty board.(All values in the board is 0. 0 = Empty , 1 = X , 2
    RETURNS : A 3 by 3 numpy array with all the values set to zero.
    """
    return(np.array([[0, 0, 0],
                     [0, 0, 0],
                     [0, 0, 0]]))


def print_board(board):
  """
   This function displays the board.
   INPUT : board - The board to be dispalyed.
   OUTPUT : Displays the board.
  """
  for row in board:
      print(row)



def Initialize_board(board_data,board):
  """
  This function Initilizes the board with the given data.
  INPUT : board_data = Initial value of all the 9 fields.
          board = The board which has to be initilized.
  RETURNS : A 3 by 3 numpy array set to the values board_data.
  """
  for i in range(3) :
     for j in range(3):
       board[i][j] = board_data[i][j]
```

```
  return board


# Create a new empty board and display it
a = create_board()
print_board(a)
```

```
    [0 0 0]
    [0 0 0]
    [0 0 0]
```

```
# Initialize a board (0 => Empty, 1 => X and 2 => 0)
a = Initialize_board([[0,0,1],[0,1,2],[0,0,0]],a)
print_board(a)
```

```
    [0 0 1]
    [0 1 2]
    [0 0 0]
```

Q3) Write a function PossibleMoves(b,p) that shows all the possible moves for a player ‘p’ given the current board state ‘b’.

```
def PossibleMoves(board):
  """
  This function returns the location of the empty cells.
  INPUT : board
  RETURNS : List of location of all the empty cells in the board.
  """
  l = []
  for i in range(len(board)):
     for j in range(len(board)):
         if board[i][j] == 0:
            l.append((i, j))
  return(l)


print(PossibleMoves(a))
print("Possible moves in current board are ",len(PossibleMoves(a)))
print_board(a)
```

```
    [(0, 0), (0, 1), (1, 0), (2, 0), (2, 1), (2, 2)]
    Possible moves in current board are  6
    [0 0 1]
    [0 1 2]
    [0 0 0]
```

Q4) Write a function BestMove(b,p) to find the best move for a player ‘p’ given a board state ‘b’

Given a board state b, find all the PossibleMoves(b,p). For each possible move, Evaluate(b) and

```
def MakeOneMove(board,move,player):
  """
  This function will set one field in the tic tac toe board matrix, with 1 if player is playi
  INPUT : 1) board : board on which changes have to be made
          2) Move : Co-ordinate on which the move has to be made
          3) player : 1 in case the player is playing X
                      2 in case the player is playing 0
  """
  board[move[0]][move[1]] = player


def BestMove(board,player):
  """
  This function checks all the possible moves and returns the best move possible.
  Input : 1) board : board which have to be evaluated
          2) player : 1 in case the player is playing X
                      2 in case the player is playing 0
  """
  possible_moves = PossibleMoves(board)
  score_old = 0

  for move in possible_moves:
     board_copy = copy.deepcopy(board)
     MakeOneMove(board_copy, move, player)
     score = evaluate(board_copy)
     if score > score_old:
       my_move = move
     score_old = score
  print("Best move player {}  is {} ".format(player,my_move))
  print("\n BOARD\n",board)


# Checking for best move for "X" player on board "a".
BestMove(a,1)
```

```
    Best move player 1  is (2, 0)

     BOARD
     [[0 0 1]
     [0 1 2]
     [0 0 0]]
```

Q5) Write a function Evaluate(b) to find the score for board state b.

The features to be considered are

• x1 = # of instances where there are 2 x’s in a ROW with an open subsequent square.

- x2 = # of instances where there are 2 o's in a ROW with an open subsequent square.
- x3 = # of instances where there is an x in a completely open ROW
- x4 = # of instances where there is an o in a completely open ROW
- x5 = # of instances of 3 x's in a ROW (value of 1 signifies end game)
- x6 = # of instances of 3 o's in a ROW (value of 1 signifies end game)

```
def get_feature_value(board):

  """
  This function calculates the feature values of the board.
  INPUT : Board.
  RETURNS : The values corresponding to each feature
  """
  possibilities = []
  for row in board: # Add all the rows ROWS in the list.
         possibilities.append(row)
  for column in board.T: # Add all the columns ROWS in the list.
         possibilities.append(column)
  diagonals =  np.array([[board[0][0],board[1][1],board[2][2]],[board[2][0],board[1][1],boar
  for diagonal in diagonals: # All the diagonal ROWs in the list.
         possibilities.append(diagonal)
      # Initilize all the feature values to 0.
  x1 = 0
  x2 = 0
  x3 = 0
  x4 = 0
  x5 = 0
  x6 = 0
  for possibility in possibilities:
     zeros = 0 # Initilize empty cell count to 0
     Xs = 0 # Initilize "X" count to 0
     Os = 0 # Initilize "0" count to 0
     for entry in possibility:
             if entry == 0:
                zeros += 1
             elif entry == 1:
                 Xs += 1
             elif entry == 2:
                 Os += 1
     if Xs == 2 and zeros == 1:
              x1 += 1
     elif Os == 2 and zeros == 1:
              x2 += 1
     elif Xs == 1 and zeros == 2:
              x3 += 1
     elif Os == 1 and zeros == 2:
              x4 += 1
```

```
        elif Xs == 3:
                 x5 += 1
        elif Os == 3:
                 x6 += 1

    return x1,x2,x3,x4,x5,x6



print_board(a)
```

```
     [0 0 1]
     [0 1 2]
     [0 0 0]
```

```
get_feature_value(a)
```

```
     (1, 0, 3, 0, 0, 0)
```

```
def evaluate(board):
  """
  This function calculate the score of the board.
  INPUT : Board
  RETURNS : Score of the board
  """
  x1,x2,x3,x4,x5,x6 = get_feature_value(board)
  X = np.array([1,x1,x2,x3,x4,x5,x6])
  W = np.array([0.5,0.5,0.5,0.5,0.5,0.5,0.5])
  score_v_cap_b = sum(W.T * X)
  if x5 > 0 : # X wins, score is 100
    print("Game ended, X-Player wins")
    score = 100
  elif x6 > 0 : # 0 wins, score is -100
    print("Game ended, O-Player wins")
    score = -100
  elif (np.count_nonzero(board) == 9) and not((x5 > 0) or (x6 > 0)): # All the fields have be
    print("Game ended, Match draw")
    score = 0
  else:
    score = score_v_cap_b  # match not over yet
  return score



 evaluate(a)
```

```
     2.5
```

```
# Create a new empty board and display it
b = create_board()
print_board(b)
```

```
print_board(b)
```

```
[0 0 0]
[0 0 0]
[0 0 0]
```

```
# Initialize a board (0 => Empty, 1 => X and 2 => 0)
b = Initialize_board([[1,0,0],[2,1,2],[0,0,1]],b)
print_board(b)
```

```
[1 0 0]
[2 1 2]
[0 0 1]
```

```
evaluate(b) # Player X wins
```

```
Game ended, X-Player wins
100
```

```
# Create a new empty board and display it
c = create_board()
print_board(c)
```

```
[0 0 0]
[0 0 0]
[0 0 0]
```

```
# Initialize a board (0 => Empty, 1 => X and 2 => 0)
c = Initialize_board([[0,2,0],[1,2,1],[0,2,0]],c)
print_board(c)
```

```
[0 2 0]
[1 2 1]
[0 2 0]
```

```
evaluate(c) # Player 0 wins
```

```
Game ended, O-Player wins
-100
```

```
# Create a new empty board and display it
d = create_board()
print_board(d)
```

```
[0 0 0]
[0 0 0]
[0 0 0]
```

```
# Initialize a board (0 => Empty, 1 => X and 2 => 0)
```

```
d = Initialize_board([[1,2,1],[2,1,2],[2,1,2]],d)
print_board(d)
```

```
    [1 2 1]
    [2 1 2]
    [2 1 2]
```

```
evaluate(d) # Draw game
```

```
    Game ended, Match draw
    0
```