Course : CSC 1103 Programming Methodology Mini-Project 2024

Title : Interactive Tic-Tac-Toe Game For Children

Group : 8

Name	GUID	SIT ID	Scope	Contribution %
Gaddiel Lim Kang Seng	3070572L	2403453	4B (Naive Bayes)	20
Sheila Lim Yann Tsern	3070649L	2401392	4B (Decision Tree) Enhancements of Decision Tree Model	20
Tan Bing Kun Terence	3070672T	2401883	3, 4A Hint System Exit Button	20
Tan Jian Xin	3070651T	2401421	1, 2 Sound Effects Documentation Pseudocode	20
Tang Wei Shan Alicia	3070653T	2401448	4C Grid Animations for Main Screen GUI for Difficulty Selection & AI Selection Modes Back and Quit Buttons Winning Animation	20

Table Of Contents

1. Problem Definition **4**
2. Problem Analysis **6**
2.1. User-Friendly Graphical User Interface (GUI) 6
2.1.1. Raylib 6
2.1.2. Raylib Features 6
2.1.3. Implementing Raylib 7
2.2. Two-Player Mode 7
2.3. Single-Player Mode with AI/ML 7
2.4. Training and Evaluation of ML Models 8
2.5. Hint System 8
2.6. Score Tracking 9
2.7. Execution Instructions 9
3. Pseudocode **10**
3.1. main.c File 10
3.2. Init.c File 15
3.3. Update.c File 17
3.4. Draw.c File 20
3.5. Handle.c File 28
3.6. AI.c File 30
3.7. Check.c File 33
3.8. Hint.c File 35
3.9. Minimax.c File 36
3.10. data_processing.c File 38
3.11. NBmodel.c File 40
3.12. decisiontree.c File 45
4. Function Descriptions **53**
4.1. Game Logic Functions 53
4.2. UI and Animation Functions 53
4.3. Data Processing Functions 54
4.4. AI Functions 54
4.5. Raylib Functions 56
4.6. Standard Functions 57
5. Program Interfaces **58**
6. Implementation of Machine Learning (Naive Bayes) (Easy Mode) **63**
6.1 Extracting and Processing of Dataset 63
6.2 Logic of Naive Bayes 64
6.3 Implementation of the Model 64
6.4 Evaluation of Model 68
6.5 Plots and Results (Naive Bayes) 69
7. Implementation of Machine Learning (Decision Tree) (Easy Mode) **71**
7.1 Extracting and Processing of Dataset 71
7.2 Decision Tree Construction and Training 73
7.3 Splitting the Dataset 74
7.4 Decision Tree Construction and Training 76
7.5 Splitting Nodes and Tree-Building Process (build_tree) 81

7.6 Saving Position Probabilities 82
7.7 Decision Tree Model Prediction dt_predict_best_move 85
7.8 Evaluation of Decision Tree Model and Confusion Matrix evaluate_with_randomness() 87
7.9 Confusion Matrix Plot and Results (Decision Tree) 90
8. Implementation of Minimax Algorithm (Hard Mode) 91
8.1. Purpose and Function 91
8.2. Operation of Minimax Algorithm 91
8.2.1. Recursion 92
8.2.2. Depth-First Search (DFS) Algorithm 92
8.2.3. Maximizing and Minimizing Players 92
8.2.4. Evaluation and Optimal Move Selection 93
8.3. Parameters of Minimax() Function 93
9. Comparison & Interesting Findings / Assumptions 94
9.1. Findings on Dataset 94
9.2. Comparison Between Minimax Algorithm and Decision Tree Model 94
9.2.1 Key Characteristics of Minimax: 94
9.2.2 Key Characteristics of the Decision Tree Model: 94
9.2.3 Limitations 95
9.2.4 Conclusion 95
9.3. Comparison between Minimax Algorithm and Naive Bayes Model 95
10. Enhancements 96
10.1. Minimax (Imperfectness) (Medium Mode) 96
10.1.1 Depth Limitation 96
10.2. Minimax (Alpha-Beta Pruning) 96
10.2.1 Alpha and Beta 97
10.2.2 Operation of Optimized Minimax 97
10.3. Modular Approach 97
10.4. Decision Tree Model Prediction with Randomness predict_with_randomness 98
10.5. Memory Free Allocation for the Decision Tree Model free_tree() 100
11. Gauging of Difficulty Levels 102
11.1. Level of Difficulty for Easy Mode 102
11.2. Level of Difficulty for Medium Mode 103
11.3. Level of Difficulty for Hard Mode 103
12. Appendix 104
Source Codes 104
12.1. Main.c 104
12.2. AI.c 111
12.3. Check.c 115
12.4. Draw.c 117
12.5. Handle.c 127
12.6. Hint.c 130
12.7. Init.c 131
12.8. Minimax.c 133
12.9. Update.c 135
12.10. decisiontree.c 139
12.11. data_processing.c 151

12.12. NBmodel.c 153
12.13. plot_confusion_plot.py 159
12.14. decisiontree.c 161
12.15. confusionmatrix.py (Decision Tree) 175
13. References 179

1. Problem Definition

The primary objective of this project was to develop a 3x3 Tic-Tac-Toe game using the C programming language, integrated with Artificial Intelligence (AI) and Machine Learning (ML) algorithms. The game was specifically designed for a low-memory, power-constrained IoT tablet, intended for use in a nursery childcare setting. The purpose of the game was to support the early development of critical thinking, motor skills, and social interaction in young children.

The game included two distinct modes:

1. Two-Player Mode: This mode was designed to foster social interaction and cooperation among children. Players took turns making moves until a winner was determined or the game ended in a draw.

2. Single-Player Mode: In this mode, players competed against a programmed AI opponent. The AI featured deliberate imperfections to balance the gameplay experience, reducing frustration and allowing children to win occasionally. Multiple difficulty levels were implemented, adjusting the AI's performance to add variety and challenge to the gameplay.

A visually engaging and interactive graphical user interface (GUI) was developed to enhance the user experience. Key features of the GUI included:

- Player Symbols: Distinctive symbols (X and O) were assigned to each player.
- Turn Indicators: Clear visual cues indicated which player's turn it was.
- Winner Declarations: Messages were displayed to announce the winner or a draw outcome.

Sound Effects and Animations:

The game incorporated sound effects and animations to maintain engagement and reward players for their efforts:

- Background Music: A looping, thematic tune was included to create an engaging atmosphere.
- Winning Sound: Celebratory sound effects were played when a player won.
- Confetti Animation: A party popper-style animation was displayed as a reward for winning, encouraging children to continue playing.
- Losing/Draw Sound: Distinct audio feedback was provided for draw and losing outcomes.
- Button Clicks: Clicking sounds were added for interactive buttons such as retry, back, and exit.
- Move Placement Sound: Subtle sound feedback accompanied each player move to enhance interactivity.

Performance Tracking:

To maintain engagement and provide caregivers with insights into children's developmental progress, a performance counter was integrated. This feature tracked the number of wins achieved across different difficulty levels, offering a measure of the child's progress over time.

By balancing gameplay with developmental benefits, the project successfully delivered an engaging and supportive game designed to enhance children’s early learning experiences.

2. Problem Analysis

This section illustrates the approaches the team used to resolve the problems defined in the previous section.

2.1. User-Friendly Graphical User Interface (GUI)

The GUI will be developed using Raylib as it is lightweight and easy to use. The following features are considered:

- Symbols for each player are displayed clearly.
- Turn-taking is indicated during gameplay.
- The game outcome (win, lose or draw) is announced automatically when the winner is detected or all the grids are filled.
- Efficiency is to be considered to ensure smooth operation on resource-constrained devices.

2.1.1. Raylib

Raylib is a simple and easy-to-use open-source GUI library. Raylib supports multiple platforms, technically, any platform that supports C language and OpenGL graphics can run raylib or easily ported to. It is also supported by a huge community to update and maintain the library.

2.1.2. Raylib Features

Raylib has many advantages that make it a great choice for the GUI. Below are the features that raylib offers:

- No external dependencies, all required libraries are included with raylib.
- Multiplatform.
- Written in C code.
- Hardware accelerated with OpenGL.
- Unique OpenGL abstraction layer.
- Powerful fonts module.
- Multiple texture formats support, including compressed formats.
- Full 3D support.
- Flexible materials system, supporting classic maps and PBR maps.
- Supports animated 3D models.
- Supports shader.
- Powerful mathematics module for vector, matrix and quaternion operations.
- Audio loading and playing with streaming support.
- Bindings to over 60 programming languages.
- Free and open source.

With these many great features and few to no downsides, raylib is indeed a great and suitable choice for the program's GUI.

2.1.3. Implementing Raylib

Using raylib for the GUI was relatively easy, only requiring the library to be downloaded, then copy the raylib .h and .a files into the project directory. The library files required for this project wereraylib.h, and libraylib.a, libopengl32.a, libgdi32.a and libwinmm.a respectively.

Once the required files are present in the project directory, the program with raylib functions can be compiled with the .a files linked. The .a files are linked by adding -L./Libraries -lraylib -lopengl32 -lgdi32 -lwinmm into the compile command. By adding the linkage instructions, the program will be able to utilise the functions from raylib.

2.2. Two-Player Mode

In the two-player mode, the gameplay allows two human players to take turns to input on the 3x3 grid. Basic features to be implemented include but are not limited to:

1. Automatic checking of winning combinations.
2. The game outcome (win, lose or draw) announcement when the winner is detected or all the grids are filled.
3. Turn indicators allow players to acknowledge their turns.

In order to satisfy the aforementioned features, the following need to be implemented:

- `CheckWin()`: To iterate through all rows, columns and diagonals to check if three of the same player's symbols are connected in a straight line. If the condition is true, return true, else, return false, indicating this game has a winner ('X' or 'O').
- `CheckDraw()`: To iterate through all rows, columns and diagonals to check if there is an empty cell. If yes, return false, else, return true, indicating this game is a draw.
- A variable `currentPlayerTurn` to store the current player. When `currentPlayerTurn == PLAYER_X_TURN`, use `DrawText()` function from Raylib to draw text to indicate it is player X's turn.

2.3. Single-Player Mode with AI/ML

In the single-player mode, an AI opponent would be implemented using the Minimax Algorithm, chosen for its robust decision-making capabilities. To enhance usability and accessibility, the following features were incorporated:

1. The AI/ML would make its move without any human intervention.
2. Varying difficulty levels were incorporated into the game. When the Minimax Algorithm searches through all the possible moves to use the best one, players have no winning opportunity when against a fully implemented Minimax. To create winning opportunities, modification has been made to limit the search depth of the Minimax Algorithm.
3. The program will notify players of the outcome at the end of each game, ensuring clarity and feedback.
4. Randomise the starting player to increase playability as always having the same player to make the first move can be underwhelming.

In order to satisfy the aforementioned features, the following need to be implemented:

- `HandlePlayerTurn()`: to handle the different player's turn. This function should change the currentPlayerTurn variable after the previous player has made a move and handles other GUI related functions related to players.
- `AITurn()`: this function is to be called in single-player mode, handling the AI moves. A variable isTwoPlayer needs to be declared to store a boolean to check if it is single or two players. If `isTwoPlayer == false`, the `AITurn()` function should be called to handle the AI.
- Different ML models and AI algorithms such as Decision Tree, Naive Bayes and Minimax have been implemented to provide a variety to the single-player gameplay.
- A `gameState` variable to store the current game state. For example, when the game is on the main menu, the `gameState` variable will be `MENU` etc. Introducing this variable will allow the different pages to be displayed to the players, to satisfy the difficulty selection requirement.
- `RandomizeStartingPlayer()`: to randomise the starting player. The function will get a random value from 0 to 1. If the returned value is 0, player X starts first, else, player O starts first.
- `Minimax()`: this function is called to do a complete search of all the possible moves and choose the best move for the computer.
- Added `depthLimit` parameter to the `Minimax()` function to control how many levels the algorithm will search through to make a decision. This will create imperfection in the algorithm which would cause the computer to err so that the player can exploit the opportunity to win.

2.4. Training and Evaluation of ML Models

The Naive Bayes model was trained on an 80:20 split of positive and negative moves to simulate varying difficulty levels. The evaluation was conducted using:

- Training and testing accuracy to assess the model's effectiveness.
- A confusion matrix to identify and analyse misclassifications, ensuring the AI behaved as intended.

2.5. Hint System

A hint button was implemented to enhance the learning and player experience. This feature provides clues to assist players in identifying potential winning moves, fostering problem-solving and strategic thinking.

This function is essentially a Minimax replica. When players use this function, the function will implement a full Minimax Algorithm on the current game state, giving players the next best move available, increasing the chance of winning.

However, to foster a positive learning attitude and environment, a restriction of two hints per game for each player was implemented into the function to avoid abuse of hints. This would allow the players to greatly increase chances of winning, while requiring them to think at the same time.

In order to satisfy the aforementioned features, the following need to be implemented:

- `clearHint()`: this function is called to remove the previous best move.
- `getHint()`: this function is called to activate Minimax to get the best move for the player.
- The hint button would be visible to the player, displaying the remaining chances of hints left. Once the player clicked the hint button, the best move was played, and the remaining chances were reduced by 1. When there were no remaining chances, the button would become unclickable.

2.6. Score Tracking

A score-tracking feature was implemented to monitor player performance over time. This feature recorded the number of wins achieved in each difficulty mode, providing a measurable indicator of progress.

Using the ModeStats structure and the GetCurrentModeStats() function, the system maintained separate statistics for each difficulty level (Medium and Hard) and AI model (Naive Bayes and Decision Tree). Dedicated ModeStats structures, including mediumStats, hardStats, naiveBayesStats, and decisionTreeStats, were used to track wins, losses, and draws for each category.

The GetCurrentModeStats() function was designed to return a pointer to the relevant statistics based on the current game mode and AI model selection. For the Easy mode, it returned either naiveBayesStats or decisionTreeStats, depending on the chosen AI model. For Medium and Hard difficulties, it returned mediumStats or hardStats, respectively.

This tracking system provided players with detailed insights into their performance against different AI opponents and allowed for the evaluation of the effectiveness of various AI strategies. By incorporating this feature, the project successfully met its objectives of creating an educational and engaging game that supported learning and skill development.

2.7. Execution Instructions

Compiled the program using a C compiler with Raylib linked using:

```
gcc -o main main.c DecisionTree_ML/*.c NBmodel/*.c GameFunctions/*.c -I./DecisionTree_ML -I./NBmodel -I./GameFunctions -L./Libraries -lraylib -lopengl32 -lgdi32 -lwinmm
```

Executed the compiled program using:

```
./main
```

3. Pseudocode

This section shows the pseudocode that corresponds to the respective C files, for the functions in the program.

3.1. main.c File

```
BEGIN
    InitWindow(SCREEN_WIDTH, SCREEN_HEIGHT, "Tic-Tac-Toe")
    InitAudioDevice()
    icon ← LoadImage("assets\icon.png")
    SetWindowIcon(icon)
    UnloadImage(icon)

    buttonClickSound ← LoadSound("assets\ButtonClicked.mp3")
    popSound ← LoadSound("assets\Pop.mp3")
    victorySound ← LoadSound("assets\FFVictory.mp3")
    loseSound ← LoadSound("assets\MarioLose/mp3")
    drawSound ← LoadSound("assets\Draw.mp3")
    mainMenuSound ← LoadSound("assets\MainMenu.mp3")
    playSound ← LoadSound("assets\Play.mp3")

    SetSoundVolume(buttonClickSound, 0.4f)
    SetSoundVolume(popSound, 0.4f)
    SetSoundVolume(victorySound, 0.4f)
    SetSoundVolume(loseSound, 0.4f)
    SetSoundVolume(drawSound, 0.4f)
    SetSoundVolume(mainMenuSound, 0.4f)
    SetSoundVolume(playSound, 0.4f)

    InitSymbols()
    InitTitleWords()
    InitConfetti()

    boards[1000][NUM_POSITIONS + 1] ← {0}

    total_records ← 0

    load_data("tic-tac-toe.data", boards, outcomes, &total_records)

    train_size ← 0
    test_size ← 0

    split_data(boards, outcomes, total_records, train_boards, train_outcomes, test_boards, test_outcomes, &train_size, &test_size, RATIO)

    train_NBmodel(&NBmodel, train_boards, train_outcomes, train_size)

    save_NBmodel(&NBmodel, "NBmodel/NBmodel_weights.txt")

    mode ← "w"
    type ← "Training"
```

```
            test_NBmodel("NBmodel/NBmodel_confusion_matrix.txt", mode, type, &NBmodel,
train_boards, train_outcomes, train_size)

    strcpy(mode, "a")
    strcpy(type, "Testing")
    test_NBmodel("NBmodel/NBmodel_confusion_matrix.txt", mode, type, &NBmodel, test_boards,
test_outcomes, test_size)

    growth_Tree(&TDmodel)

    WHILE (!WindowShouldClose())
            IF (gameState == MENU || gameState == DIFFICULTY_SELECT || gameState ==
MODEL_SELECT)
            IF (!IsSoundPlaying(mainMenuSound))
                PlaySound(mainMenuSound)
            ENDIF
            StopSound(playSound)
        ELSEIF (gameState == GAME)
            IF (!IsSoundPlaying(playSound))
                PlaySound(playSound)
            ENDIF
            StopSound(mainMenuSound)
        ELSE
            StopSound(mainMenuSound)
            StopSound(playSound)
        ENDIF

            IF (gameState == MENU || gameState == DIFFICULTY_SELECT || gameState ==
MODEL_SELECT)
            UpdateSymbols()
            UpdateTitleWords()
        ENDIF

        IF (gameState == MENU)
            IF (IsMouseButtonPressed(MOUSE_LEFT_BUTTON))
                mousePos ← GetMousePosition()

            IF (mousePos.x >= SCREEN_WIDTH/2 - 100 && mousePos.x <= SCREEN_WIDTH/2 +
100 &&
                mousePos.y >= SCREEN_HEIGHT/2 + 60 && mousePos.y <= SCREEN_HEIGHT/2
+ 100)
                    PlaySound(buttonClickSound)
                    isTwoPlayer ← false
                    gameState ← DIFFICULTY_SELECT
                    ELSEIF (mousePos.x >= SCREEN_WIDTH/2 - 100 && mousePos.x <=
SCREEN_WIDTH/2 + 100 &&
                    mousePos.y >= SCREEN_HEIGHT/2 + 120 && mousePos.y <= SCREEN_HEIGHT/2
+ 160)
                    PlaySound(buttonClickSound)
                    isTwoPlayer ← true
                    gameState ← GAME
                    InitGame()
```

```
                    ELSEIF (mousePos.x >= SCREEN_WIDTH/2 - 100 && mousePos.x <=
SCREEN_WIDTH/2 + 100 &&
                        mousePos.y >= SCREEN_HEIGHT/2 + 180 && mousePos.y <=
SCREEN_HEIGHT/2 + 220)
                PlaySound(buttonClickSound)
                BREAK
            ENDIF
        ENDIF
    ELSEIF (gameState == GAME)
        UpdateGame(buttonClickSound, popSound, victorySound, loseSound, drawSound)
    ELSEIF (gameState == GAME_OVER)
        UpdateGameOver(buttonClickSound)
    ELSEIF (gameState == DIFFICULTY_SELECT)

        IF (IsMouseButtonPressed(MOUSE_LEFT_BUTTON))
            mousePos ← GetMousePosition()

                IF (mousePos.x >= 20 && mousePos.x <= SCREEN_WIDTH/6 && mousePos.y >= 10
&& mousePos.y <= 40)
                PlaySound(buttonClickSound)
                gameState ← MENU
            ENDIF

            IF (mousePos.x >= SCREEN_WIDTH/2 - BUTTON_WIDTH/2 &&
                mousePos.x <= SCREEN_WIDTH/2 + BUTTON_WIDTH/2)

                IF (mousePos.y >= SCREEN_HEIGHT/2 && mousePos.y <= SCREEN_HEIGHT/2 +
BUTTON_HEIGHT)
                    PlaySound(buttonClickSound)
                    currentDifficulty ← EASY
                    gameState ← MODEL_SELECT
                    InitGame()
                ELSEIF (mousePos.y >= SCREEN_HEIGHT/2 + BUTTON_HEIGHT + 20 &&
                        mousePos.y <= SCREEN_HEIGHT/2 + BUTTON_HEIGHT * 2 + 20)
                    PlaySound(buttonClickSound)
                    currentDifficulty ← MEDIUM
                    gameState ← GAME
                    InitGame()
                ELSEIF (mousePos.y >= SCREEN_HEIGHT/2 + (BUTTON_HEIGHT + 20) * 2 &&
                            mousePos.y <= SCREEN_HEIGHT/2 + (BUTTON_HEIGHT + 20) * 2 +
BUTTON_HEIGHT)
                    PlaySound(buttonClickSound)
                    currentDifficulty ← HARD
                    gameState ← GAME
                    InitGame()
                ENDIF
            ENDIF
        ENDIF
    ELSEIF (gameState == MODEL_SELECT)
        IF (IsMouseButtonPressed(MOUSE_LEFT_BUTTON))
            mousePos ← GetMousePosition()
```

```
            IF (mousePos.x >= 20 && mousePos.x <= SCREEN_WIDTH/6 && mousePos.y >= 10
&& mousePos.y <= 40)
                PlaySound(buttonClickSound)
                gameState ← DIFFICULTY_SELECT
            ENDIF

            nbBtn = SCREEN_WIDTH/2 - BUTTON_WIDTH/2,
                  SCREEN_HEIGHT/2,
                  BUTTON_WIDTH,
                  BUTTON_HEIGHT

            dtBtn = SCREEN_WIDTH/2 - BUTTON_WIDTH/2,
                  SCREEN_HEIGHT/2,
                  BUTTON_WIDTH,
                  BUTTON_HEIGHT

            IF (CheckCollisionPointRec(mousePos, nbBtn))
                PlaySound(buttonClickSound)
                currentModel ← NAIVE_BAYES
                gameState ← GAME
                InitGame()
            ELSEIF (CheckCollisionPointRec(mousePos, dtBtn))
                PlaySound(buttonClickSound)
                currentModel ← DECISION_TREE
                gameState ← GAME
                InitGame()
            ENDIF
        ENDIF
    ENDIF

    BeginDrawing()
    ClearBackground(RAYWHITE)

    SWITCH(gameState)
        CASE MENU
            DrawSymbols()
            DrawTitleWords()
            DrawMenu()
            BREAK
        CASE DIFFICULTY_SELECT
            DrawSymbols()
            DrawDifficultySelect()
            BREAK
        CASE MODEL_SELECT
            DrawSymbols()
            DrawModelSelect()
            BREAK
        CASE GAME
            DrawGame()
            BREAK
        CASE GAME_OVER
            DrawGame()
            DrawGameOver()
```

```
            IF (showPartyAnimation == true)
                UpdateConfetti()
                DrawConfetti()
            ENDIF
            BREAK
        ENDSWITCH
        EndDrawing()
    ENDWHILE

    UnloadSound(buttonClickSound)
    UnloadSound(popSound)
    UnloadSound(victorySound)
    UnloadSound(loseSound)
    UnloadSound(drawSound)
    UnloadSound(mainMenuSound)
    UnloadSound(playSound)
    CloseAudioDevice()
    CloseWindow()
    return 0
END

FUNCTION GetCurrentModeStats()
    IF (currentDifficulty == EASY)
        IF (currentModel == NAIVE_BAYES)
            return &naiveBayesStats
        ELSE
            return &decisionTreeStats
        ENDIF
    ELSE
        IF (currentDifficulty == MEDIUM)
            return &mediumStats
        ELSE
            return &hardStats
        ENDIF
    ENDIF
ENDFUNCTION

FUNCTION RandomizeStartingPlayer()
    IF (GetRandomValue(0, 1) == 0)
        currentPlayerTurn ← PLAYER_X_TURN
    ELSE
        currentPlayerTurn ← PLAYER_O_TURN
    ENDIF
ENDFUNCTION
```

3.2. Init.c File

```
FUNCTION InitTitleWords()
   words ← {"Tic", "-", "Tac", "-", "Toe"}
   startX ← SCREEN_WIDTH / 2 - MeasureText("Tic-Tac-Toe", 40) / 2
   startY ← SCREEN_HEIGHT / 5 + TITLE_GRID_SIZE * 50 + 20
   int spacing ← 10

   FOR i ← 0 to 4 do
      titleWords[i].word ← words[i]
      titleWords[i].position ← (Vector2){ startX, startY }
      titleWords[i].targetPosition ← (Vector2){ startX, startY - 20 }
      titleWords[i].isJumping ← false
      titleWords[i].jumpSpeed ← JUMP_SPEED
      startX ← startX + MeasureText(words[i], 40) + spacing
   ENDFOR
ENDFUNCTION

FUNCTION InitSymbols()
   FOR i = 0 to MAX_SYMBOLS - 1 do
      symbols[i].position ← (Vector2){ GetRandomValue(0, SCREEN_WIDTH), GetRandomValue(-SCREEN_HEIGHT, 0) }
      symbols[i].symbol ← GetRandomValue(0, 1) ? 'X' : 'O'
      symbols[i].rotation ← GetRandomValue(0, 360)
   ENDFOR
ENDFUNCTION

FUNCTION InitConfetti()
   FOR i = 0 to MAX_CONFETTI - 1 do
      confetti[i].position ← (Vector2){ SCREEN_WIDTH - GetRandomValue(30, 70), SCREEN_HEIGHT - GetRandomValue(30, 70)}
      angle ← GetRandomValue(160, 280) * DEG2RAD
      speed ← GetRandomValue(600, 1200)/100.0f
      confetti[i].velocity ← (Vector2){ cos(angle) * speed, sin(angle) * speed }
      SWITCH(GetRandomValue(0, 4))
         CASE 0:
            confetti[i].color ← RED
            BREAK
         CASE 1:
            confetti[i].color ← GREEN
            BREAK
         CASE 2:
            confetti[i].color ← BLUE
            BREAK
         CASE 3:
            confetti[i].color ← YELLOW
            BREAK
         CASE 4:
            confetti[i].color ← Pink
            BREAK
      ENDSWITCH
      confetti[i].size ← GetRandomValue(2, 4)
      confetti[i].active ← true
```

```
      confetti[i].alpha ← 1.0f
      confetti[i].lifetime ← GetRandomValue(150, 200)/100.0f
   ENDFOR
ENDFUNCTION

FUNCTION InitGame()
   hint.hintCountO ← 0
   hint.hintCountX ← 0
   showPartyAnimation ← false
   StopSound(victorySound)
   StopSound(loseSound)
   StopSound(drawSound)
   memset(grid, EMPTY, sizeof(grid))
   gameOver ← false
   winner ← EMPTY
   RandomizeStartingPlayer()
   FOR i = 0 to 2 do
      winningCells[i][0] ← -1
      winningCells[i][1] ← -1
ENDFUNCTION
```

3.3. Update.c File

```
FUNCTION UpdateTitleWords()
   currentWord ← 0
   jumpDelay ← 0.0f

   jumpDelay ← jumpDelay + GetFrameTime()
   IF (jumpDelay > JUMP_DELAY)
      IF (!titleWords[currentWord].isJumping)
         titleWords[currentWord].isJumping ← true
         jumpDelay ← 0.0f
      ENDIF
   ENDIF

   FOR i = 0 to 4 do
      IF (titleWords[i].isJumping)
         titleWords[i].position.y ← titleWords[i].position.y - titleWords[i].jumpSpeed

         IF (titleWords[i].position.y <= titleWords[i].targetPosition.y)
            titleWords[i].jumpSpeed ← -titleWords[i].jumpSpeed
         ENDIF

         IF (titleWords[i].position.y >= SCREEN_HEIGHT / 5 + TITLE_GRID_SIZE * 50 + 20)
            titleWords[i].position.y ← SCREEN_HEIGHT / 5 + TITLE_GRID_SIZE * 50 + 20
            titleWords[i].isJumping ← false
            titleWords[i].jumpSpeed ← JUMP_SPEED
            currentWord ← (currentWord + 1) % 5
         ENDIF
      ENDIF
   ENDFOR
ENDFUNCTION

FUNCTION UpdateSymbols()
   FOR i = 0 to MAX_SYMBOLS - 1 do
      symbols[i].position.y ← symbols[i].position.y + SYMBOL_SPEED
      symbols[i].rotation ← symbols[i].rotation + ROTATION_SPEED
      IF (symbols[i].position.y > SCREEN_HEIGHT)
         symbols[i].position.y ← GetRandomValue(-SCREEN_HEIGHT, 0)
         symbols[i].position.x ← GetRandomValue(0, SCREEN_WIDTH)
         symbols[i].symbol ← IF GetRandomValue(0, 1) ? 'X' : 'O'
         symbols[i].rotation ← GetRandomValue(0, 360)
      ENDIF
   ENDFOR
ENDFUNCTION

FUNCTION UpdateConfetti()
   FOR i = 0 to MAX_CONFETTI - 1 do
      IF (confetti[i].active)
         allInactive ← false
         confetti[i].velocity.x = confetti[i].velocity.x * 0.99f
         confetti[i].velocity.y = confetti[i].velocity.y * 0.99f

         confetti[i].position.x ← confetti[i].position.x + confetti[i].velocity.x * 0.6f
```

```
        confetti[i].position.y ← confetti[i].position.y + confetti[i].velocity.y * 0.6f

        confetti[i].velocity.y ← confetti[i].velocity.y + 0.02f

        confetti[i].velocity.x ← confetti[i].velocity.x + GetRandomValue(-20, 20) / 100.0f
        confetti[i].velocity.y ← confetti[i].velocity.y + GetRandomValue(-20, 20) / 100.0f

        confetti[i].alpha ← confetti[i].alpha - 0.02f
        confetti[i].lifetime ← confetti[i].lifetime - 0.02f

        IF (confetti[i].alpha <= 0 ||
           confetti[i].lifetime <= 0 ||
           confetti[i].position.y > SCREEN_HEIGHT + 50 ||
           confetti[i].position.x < -50 ||
           confetti[i].position.x > SCREEN_WIDTH + 50)
              confetti[i].active ← false
        ENDIF
     ENDIF
  ENDFOR
  IF (allInactive)
     showPartyAnimation ← false
  ENDIF
ENDFUNCTION

FUNCTION UpdateGame(buttonClickSound, popSound, victorySound, loseSound, drawSound,
*model, *TDmodel)
  IF (gameOver) return
  IF (IsMouseButtonPressed(MOUSE_LEFT_BUTTON))
     mousePos ← GetMousePosition()
       IF (mousePos.x >= SCREEN_WIDTH - 80 && mousePos.x <= SCREEN_WIDTH - 10 &&
mousePos.y >= 10 && mousePos.y <= 40)
        PlaySound(buttonClickSound)
        gameState ← MENU
        return
     ENDIF
  ENDIF
  IF (currentPlayerTurn == PLAYER_X_TURN)
     IF (HandlePlayerTurn(popSound, victorySound, loseSound, drawSound))
        PlaySound(popSound)
     ENDIF
  ELSEIF (currentPlayerTurn == PLAYER_O_TURN)
     IF (isTwoPlayer)
        IF (HandlePlayerTurn(popSound, victorySound, loseSound, drawSound))
           PlaySound(popSound)
        ENDIF
     ELSE
        SWITCH(currentDifficulty)
           CASE EASY
              IF (currentModel == NAIVE_BAYES)
                 AITurn(victorySound, loseSound, drawSound, model)
              ELSE
                 AITurnDecisionTree(victorySound, loseSound, drawSound, TDmodel)
              ENDIF
```

```
            BREAK
        CASE MEDIUM
            AITurn(victorySound, loseSound, drawSound, model)
            BREAK
        CASE HARD
            AITurn(victorySound, loseSound, drawSound, model)
            BREAK
      ENDSWITCH
    ENDIF
  ENDIF
ENDFUNCTION

FUNCTION UpdateGameOver(buttonClickSound)
  IF (IsMouseButtonPressed(MOUSE_LEFT_BUTTON))
    mousePos ← GetMousePosition()
    retryBtn ← {
      SCREEN_WIDTH/2 - BUTTON_WIDTH/2,
      SCREEN_HEIGHT/2 + 40,
      BUTTON_WIDTH,
      BUTTON_HEIGHT
    }
    menuBtn ← {
      SCREEN_WIDTH/2 - BUTTON_WIDTH/2,
      SCREEN_HEIGHT/2 + 100,
      BUTTON_WIDTH,
      BUTTON_HEIGHT
    }
    IF (CheckCollisionPointRec(mousePos, menuBtn))
      PlaySound(buttonClickSound)
      gameState ← MENU
      InitGame()
    ELSEIF (CheckCollisionPointRec(mousePos, retryBtn))
      PlaySound(buttonClickSound)
      gameState ← GAME
      InitGame()
    ENDIF
  ENDIF
ENDFUNCTION
```

3.4. Draw.c File

```
FUNCTION DrawConfetti()
   FOR i = 0 to MAX_CONFETTI - 1 do
      IF (confetti[i].active)
         particleColor ← confetti[i].color
         particleColor.a ← confetti[i].alpha*255
         direction ← { -confetti[i].velocity.x * 0.15f, -confetti[i].velocity.y * 0.15f }
         DrawCircle(confetti[i].position.x, confetti[i].position.y, confetti[i].size, particleColor)
         FOR trail = 0 to 7 do
            trailAlpha ← confetti[i].alpha * (1.0f - (trail * 0.14f))
               trailPos ← { confetti[i].position.x + direction.x * trail, confetti[i].position.y + direction.y * trail }
                    DrawCircle(trailPos.x, trailPos.y, confetti[i].size * (1.0f - (trail * 0.12f)), ColorAlpha(particleColor, trailAlpha * 255))
         ENDFOR
      ENDIF
   ENDFOR
ENDFUNCTION

FUNCTION DrawTitleWords()
   FOR i = 0 to 4 do
      DrawText(titleWords[i].word, titleWords[i].position.x, titleWords[i].position.y, 40, BLACK)
   ENDFOR
ENDFUNCTION

FUNCTION DrawSymbols()
   FOR i = 0 tp MAX_SYMBOLS - 1 do
      origin ← {10, 10}
            DrawTextPro(GetFontDefault(), &symbols[i].symbol, symbols[i].position, origin, symbols[i].rotation, 20, 1, symbols[i].symbol == 'X' ? BLUE : RED)
   ENDFOR
ENDFUNCTION

FUNCTION DrawGame()
   isHintHovered ← false
   mousePos ← GetMousePosition()

   FOR i = 0 to GRID_SIZE - 1 do
      FOR j = 0 to GRID_SIZE - 1 do
         cell ← {(j * CELL_SIZE), (i * CELL_SIZE), CELL_SIZE, CELL_SIZE}
         isWinningCell ← false
         IF (gameOver && winner != EMPTY)
            FOR k = 0 to 2 do
               IF (winningCells[k][0] == i && winningCells[k][1] == j)
                  isWinningCell ← true
                  BREAK
               ENDIF
            ENDFOR
         ENDIF

          isHovered ← !gameOver && grid[i][j] == EMPTY && CheckCollisionPointRec(mousePos, cell)
```

```
        IF (isWinningCell)
            IF (!isTwoPlayer && winner == PLAYER_O)
                cellColor ← (Color){255, 200, 200, 255}
            ELSE
                cellColor ← (Color){144, 238, 144, 255}
            ENDIF
        ELSE
            cellColor ← isHovered ? DARKGRAY : LIGHTGRAY
        ENDIF

        DrawRectangleRec(cell, cellColor)

        IF (grid[i][j] == PLAYER_X)
            text ← "X"
            fontSize ← 100
            textWidth ← MeasureText(text, fontSize)
            textHeight ← fontSize * 0.75f
            textX ← cell.x + (CELL_SIZE - textWidth) / 2
            textY ← cell.y + (CELL_SIZE - textHeight) / 2
            DrawText(text, textX, textY, fontSize, BLUE)
        ELSEIF (grid[i][j] == PLAYER_O)
            text ← "O"
            fontSize ← 100
            textWidth ← MeasureText(text, fontSize)
            textHeight ← fontSize * 0.75f
            textX ← cell.x + (CELL_SIZE - textWidth) / 2
            textY ← cell.y + (CELL_SIZE - textHeight) / 2
            DrawText(text, textX, textY, fontSize, RED)
        ENDIF
    ENDFOR
ENDFOR

FOR int = 1 to GRID_SIZE - 1 do
    DrawLine(i * CELL_SIZE, 0, i * CELL_SIZE, SCREEN_HEIGHT, BLACK)
    DrawLine(0, i * CELL_SIZE, SCREEN_WIDTH, i * CELL_SIZE, BLACK)
ENDFOR

hintBtn ← {SCREEN_WIDTH - 80, 10, 70, 30}
*hintText ← "Hint: "
snprintf(hintTextFinal, sizeof(hintTextFinal), "%s%d", hintText, (2-hint.hintCountX))
IF (currentPlayerTurn == PLAYER_X_TURN)
    IF (hint.hintCountX < 2)
            isHintHovered ← (mousePos.x >= SCREEN_WIDTH - 80 && mousePos.x <=
SCREEN_WIDTH - 10 && mousePos.y >= 10 && mousePos.y <= 40)
        DrawButton(hintBtn, hintTextFinal, 20, !gameOver && isHintHovered)
    ELSE
        DrawButton(hintBtn, hintTextFinal, 20, !gameOver && false)
    ENDIF
ENDIF

snprintf(hintTextFinal, sizeof(hintTextFinal), "%s%d", hintText, (2-hint.hintCountO))
IF (currentPlayerTurn == PLAYER_O_TURN)
```

```
      IF (hint.hintCountO < 2)
                isHintHovered ← (mousePos.x >= SCREEN_WIDTH - 80 && mousePos.x <=
SCREEN_WIDTH - 10 && mousePos.y >= 10 && mousePos.y <= 40)
         DrawButton(hintBtn, hintTextFinal, 20, !gameOver && isHintHovered)
      ELSE
         DrawButton(hintBtn, hintTextFinal, 20, !gameOver && false)
      ENDIF
   ENDIF

   quitBtn ← {SCREEN_WIDTH - 80, 10, 70, 30}
   DrawButton(quitBtn, "Quit", 20, !gameOver && isQuitHovered)

   IF (!gameOver && isQuitHovered)
      SetMouseCursor(MOUSE_CURSOR_POINTING_HAND)
   ELSEIF (!gameOver && isHintHovered)
      SetMouseCursor(MOUSE_CURSOR_POINTING_HAND)
   ELSEIF (!gameOver)
      SetMouseCursor(MOUSE_CURSOR_DEFAULT)
   ENDIF

   IF (!gameOver)
      IF (!isTwoPlayer)
         currentStats ← GetCurrentModeStats()

         PRINT(statsText, "Player: %d | AI: %d | Draws: %d",
               currentStats→playerWins,
               currentStats→aiWins,
               currentStats→draws)

         DrawText(statsText, SCREEN_WIDTH/2 - MeasureText(statsText, 20)/2, 10, 20, BLACK)
      ENDIF

      yPos ← isTwoPlayer ? 20 : 40
      IF (currentPlayerTurn == PLAYER_X_TURN)
         turnText ← isTwoPlayer ? "Player X's Turn" : "Your Turn";
         DrawText(turnText, SCREEN_WIDTH/2 - MeasureText(turnText, 30)/2, yPos, 30, BLUE)
      ELSE
         turnText ← isTwoPlayer ? "Player O's Turn" : "AI's Turn";
         DrawText(turnText, SCREEN_WIDTH/2 - MeasureText(turnText, 30)/2, yPos, 30, RED)
      ENDIF
   ENDIF
ENDFUNCTION

FUNCTION DrawMenu()
   titleFontSize ← 40
   buttonFontSize ← 20
   cellSize ← 50
   gridWidth ← TITLE_GRID_SIZE * cellSize
   gridHeight ← TITLE_GRID_SIZE * cellSize
   startX ← SCREEN_WIDTH/2 - gridWidth/2
   startY ← SCREEN_HEIGHT/5

   FOR i = 0 to TITLE_GRID_SIZE - 1 do
```

```
    FOR j = 0 to TITLE_GRID_SIZE - 1 do
        cell = {
            startX + j * cellSize,
            startY + i * cellSize,
            cellSize,
            cellSize
        }
        DrawRectangleLinesEx(cell, 2, BLACK)

        IF (!titleSymbols[i][j].active && GetRandomValue(0, 100) < 2)
            titleSymbols[i][j].symbol ← GetRandomValue(0, 1) ? 'X' : 'O'
            titleSymbols[i][j].alpha ← 0
            titleSymbols[i][j].active ← true
        ENDIF

        IF (titleSymbols[i][j].active)
            titleSymbols[i][j].alpha ← titleSymbols[i][j].alpha + GetFrameTime() * 2
            IF (titleSymbols[i][j].alpha > 1.0f)
                titleSymbols[i][j].alpha ← 0
                titleSymbols[i][j].active ← false
            ENDIF

            symbolColor ← titleSymbols[i][j].symbol == 'X' ? BLUE : RED
            symbolColor.a ← (titleSymbols[i][j].alpha * 255)

            textPos ← {
                cell.x + (cellSize - MeasureText(&titleSymbols[i][j].symbol, 40))/2,
                cell.y + (cellSize - 40)/2
            }
            DrawText(&titleSymbols[i][j].symbol, textPos.x, textPos.y, 40, symbolColor)
        ENDIF
    ENDFOR
ENDFOR

singlePlayerBtn ← {
    SCREEN_WIDTH/2 - BUTTON_WIDTH/2,
    SCREEN_HEIGHT/2 + BUTTON_HEIGHT + 20,
    BUTTON_WIDTH,
    BUTTON_HEIGHT
}

twoPlayerBtn ← {
    SCREEN_WIDTH/2 - BUTTON_WIDTH/2,
    SCREEN_HEIGHT/2 + (BUTTON_HEIGHT + 20) * 2,
    BUTTON_WIDTH,
    BUTTON_HEIGHT
}

exitBtn ← {
    SCREEN_WIDTH/2 - BUTTON_WIDTH/2,
    SCREEN_HEIGHT/2 + (BUTTON_HEIGHT + 20) * 3,
    BUTTON_WIDTH,
    BUTTON_HEIGHT
```

```
    }

    singlePlayerHover ← false
    twoPlayerHover ← false
    exitHover ← false
    HandleButtonHover(singlePlayerBtn, "Single Player", buttonFontSize, &singlePlayerHover)
    HandleButtonHover(twoPlayerBtn, "Two Players", buttonFontSize, &twoPlayerHover)
    HandleButtonHover(exitBtn, "Exit", buttonFontSize, &exitHover)
        SetMouseCursor((singlePlayerHover || twoPlayerHover || exitHover) ?
MOUSE_CURSOR_POINTING_HAND : MOUSE_CURSOR_DEFAULT)
ENDFUNCTION

FUNCTION DrawGameOver()
    titleFontSize ← 40
    buttonFontSize ← 20
    DrawRectangle(0, 0, SCREEN_WIDTH, SCREEN_HEIGHT, (Color){0, 0, 0, 100})

    IF (winner == PLAYER_X)
        resultText ← isTwoPlayer ? "Player X Wins!" : "You win!"
        resultColor ← BLUE

    ELSEIF (winner == PLAYER_O)
        resultText ← isTwoPlayer ? "Player O Wins!" : "You lose!"
        resultColor ← RED

    ELSE
        resultText ← "It's a Draw!"
        resultColor ← DARKGRAY
    ENDIF

    textWidth ← MeasureText(resultText, titleFontSize)
    DrawRectangle(
        SCREEN_WIDTH/2 - textWidth/2 - 10,
        SCREEN_HEIGHT/3 - 10,
        textWidth + 20,
        titleFontSize + 20,
        WHITE
        )
    DrawText(
        resultText,
        SCREEN_WIDTH/2 - textWidth/2,
        SCREEN_HEIGHT/3,
        titleFontSize,
        resultColor
        )
    retryBtn ← {
        SCREEN_WIDTH/2 - BUTTON_WIDTH/2,
        SCREEN_HEIGHT/2 + 40,
        BUTTON_WIDTH,
        BUTTON_HEIGHT
        }
    menuBtn ← {
        SCREEN_WIDTH/2 - BUTTON_WIDTH/2,
```

```
        SCREEN_HEIGHT/2 + 100,
        BUTTON_WIDTH,
        BUTTON_HEIGHT
        }
    mousePos ← GetMousePosition()
    isHoveringMenu ← CheckCollisionPointRec(mousePos, menuBtn)
    isHoveringRetry ← CheckCollisionPointRec(mousePos, retryBtn)
    DrawButton(retryBtn, "Retry", buttonFontSize, isHoveringRetry)
    DrawButton(menuBtn, "Back to Menu", buttonFontSize, isHoveringMenu)
    SetMouseCursor((isHoveringMenu || isHoveringRetry) ? MOUSE_CURSOR_POINTING_HAND
: MOUSE_CURSOR_DEFAULT)
ENDFUNCTION

FUNCTION DrawButton(bounds, *text, fontSize, isHovered)
    vibrationBounds ← bounds

    IF (isHovered)
        buttonVibrationOffset ← sinf(GetTime() * vibrationSpeed) * vibrationAmount
        vibrationBounds.x ← vibrationBounds.x + buttonVibrationOffset
    ENDIF

    DrawRectangleRec(vibrationBounds, isHovered ? GRAY : LIGHTGRAY)
    DrawRectangleLinesEx(vibrationBounds, 2, BLACK)
    DrawText(
        text,
        vibrationBounds.x + (vibrationBounds.width - MeasureText(text, fontSize))/2,
        vibrationBounds.y + (vibrationBounds.height - fontSize)/2,
        fontSize,
        BLACK
    )
ENDFUNCTION

FUNCTION DrawDifficultySelect()
    titleFontSize ← 40
    buttonFontSize ← 20
    title ← "Select Difficulty"
    DrawText(
        title,
        SCREEN_WIDTH/2 - MeasureText(title, titleFontSize)/2,
        SCREEN_HEIGHT/3,
        titleFontSize,
        BLACK
        )

    easyBtn ← {
        SCREEN_WIDTH/2 - BUTTON_WIDTH/2,
        SCREEN_HEIGHT/2,
        BUTTON_WIDTH,
        BUTTON_HEIGHT
    }

    mediumBtn ← {
        SCREEN_WIDTH/2 - BUTTON_WIDTH/2,
```

```
        SCREEN_HEIGHT/2 + BUTTON_HEIGHT + 20,
        BUTTON_WIDTH,
        BUTTON_HEIGHT
    }

    hardBtn ← {
        SCREEN_WIDTH/2 - BUTTON_WIDTH/2,
        SCREEN_HEIGHT/2 + (BUTTON_HEIGHT + 20) * 2,
        BUTTON_WIDTH,
        BUTTON_HEIGHT
    }

    backBtn ← {
        20,
        10,
        SCREEN_WIDTH/6,
        30
    }

    easyHover ← false
    mediumHover ← false
    hardHover ← false
    backHover ← false
    HandleButtonHover(easyBtn, "Easy", buttonFontSize, &easyHover)
    HandleButtonHover(mediumBtn, "Medium", buttonFontSize, &mediumHover)
    HandleButtonHover(hardBtn, "Hard", buttonFontSize, &hardHover)
    HandleButtonHover(backBtn, "Back", buttonFontSize, &backHover)
            SetMouseCursor((easyHover || mediumHover || hardHover || backHover) ?
MOUSE_CURSOR_POINTING_HAND : MOUSE_CURSOR_DEFAULT)
ENDFUNCTION

FUNCTION DrawModelSelect()
    title ← "Select AI Model"
    DrawText(
        SCREEN_WIDTH/2 - MeasureText(title, 40)/2,
        SCREEN_HEIGHT/3,
        40,
        BLACK
    )
    nbBtn ← {
        SCREEN_WIDTH/2 - BUTTON_WIDTH/2,
        SCREEN_HEIGHT/2,
        BUTTON_WIDTH,
        BUTTON_HEIGHT
    }
    dtBtn ← {
        SCREEN_WIDTH/2 - BUTTON_WIDTH/2,
        SCREEN_HEIGHT/2 + BUTTON_HEIGHT + 20,
        BUTTON_WIDTH,
        BUTTON_HEIGHT
    }
    backBtn ← {
        20,
```

```
        10,
        SCREEN_WIDTH/6,
        30
    }
    nbHover ← false
    dtHover ← false
    backHover ← false
    HandleButtonHover(nbBtn, "Naive Bayes", 20, &nbHover)
    HandleButtonHover(dtBtn, "Decision Tree", 20, &dtHover)
    HandleButtonHover(backBtn, "Back", 20, &backHover)
     SetMouseCursor((nbHover || dtHover || backHover) ? MOUSE_CURSOR_POINTING_HAND :
MOUSE_CURSOR_DEFAULT)
ENDFUNCTION
```

3.5. Handle.c File

```
FUNCTION HandleButtonHover(button, *text, fontSize, *isHovered)
   mousePos ← GetMousePosition()
   *isHintHovered ← CheckCollisionPointRec(mousePos, button)
   DrawButton(button, text, fontSize, *isHovered)
   return *isHintHovered
ENDFUNCTION

FUNCTION HandlePlayerTurn(popSound, victorySound, loseSound, drawSound)
   clearHint()
   IF (IsMouseButtonPressed(MOUSE_LEFT_BUTTON))
      mousePos ← GetMousePosition()
      row ← (int)(mousePos.y / CELL_SIZE)
      col ← (int)(mousePos.x / CELL_SIZE)
      IF (mousePos.x >= SCREEN_WIDTH - 80 && mousePos.x <= SCREEN_WIDTH - 10
      && mousePos.y >= 10 && mousePos.y <= 40 && (hint.hintCountX < 2 || hint.hintCountO <2))
         IF (currentPlayerTurn == PLAYER_X_TURN && hint.hintCountX < 2)
            PlaySound(buttonClickSound)
            hint.hintCountX ← hint.hintCountX + 1
            getHint()
            row ← hint.row
            col ← hint.col
         ELSEIF (currentPlayerTurn == PLAYER_O_TURN && hint.hintCountO <2)
            PlaySound(buttonClickSound)
            hint.hintCountO ← hint.hintCountO + 1
            getHint()
            row = hint.row
            row = hint.col
         ELSE
            return false
         ENDIF
      ENDIF
      currentStats ← GetCurrentModeStats()
      IF (row >= 0 && row < GRID_SIZE && col >= 0 && col < GRID_SIZE)
         IF (grid[row][col] == EMPTY)
            grid[row][col] ← (currentPlayerTurn == PLAYER_X_TURN) ? PLAYER_X : PLAYER_O
            IF (CheckWin(grid[row][col]))
               gameOver ← true
               winner ← grid[row][col]
               gameState ← GAME_OVER

               IF (isTwoPlayer)
                  showPartyAnimation ← true
                  InitConfetti()
                  PlaySound(victorySound)
               ELSEIF (!isTwoPlayer && winner == PLAYER_X)
                  showPartyAnimation ← true
                  InitConfetti()
                  currentStats→playerWins ← currentStats→playerWins + 1
                  currentStats→totalGames ← currentStats→totalGames + 1
                  PlaySound(victorySound)
               ELSE
```

```
            showPartyAnimation ← false
            currentStats→aiWins ← currentStats→aiWins + 1
            currentStats→totalGames ← currentStats→totalGames + 1
            PlaySound(loseSound)
          ENDIF

        ELSEIF (CheckDraw())
          gameOver ← true
          gameState ← GAME_OVER
          winner ← EMPTY
          currentStats→draws ← currentStats→draws + 1
          currentStats→totalGames ← currentStats→totalGames + 1
          PlaySound(drawSound)

        ELSE
          currentPlayerTurn ← (currentPlayerTurn == PLAYER_X_TURN) ? PLAYER_O_TURN
: PLAYER_X_TURN
        ENDIF

        return true
      ENDIF
    ENDIF
  ENDIF
  return false
ENDFUNCTION
```

3.6. AI.c File

```
FUNCTION AITurn(victorySound, loseSound, drawSound, *model)
   bestScore ← -1000
   bestRow ← -1
   bestCol ← -1
   IF (currentDifficulty == EASY)
      IF (currentModel == NAIVE_BAYES)
         predict_move(model, grid, &bestRow, &bestCol)
      ELSE
         AITurnDecisionTree()
      ENDIF

   ELSEIF (currentDifficulty == MEDIUM)
      depthLimit ← 4
      FOR i = 0 to GRID_SIZE - 1 do
         FOR j = 0 to GRID_SIZE - 1 do
            IF (grid[i][j] == EMPTY)
               grid[i][j] ← PLAYER_O
               score ← Minimax(grid, false, 0, depthLimit, -1000, 1000)
               grid[i][j] ← EMPTY

               IF (score > bestScore)
                  bestScore ← score
                  bestRow ← i
                  bestCol ← j
               ENDIF
            ENDIF
         ENDFOR
      ENDFOR

   ELSEIF (currentDifficulty == HARD)
      depthLimit ← 9
      FOR i = 0 to GRID_SIZE - 1 do
         FOR j = 0 to GRID_SIZE - 1 do
            IF (grid[i][j] == EMPTY)
               grid[i][j] ← PLAYER_O
               score ← Minimax(grid, false, 0, depthLimit, -1000, 1000)
               grid[i][j] ← EMPTY

               IF (score > bestScore)
                  bestScore ← score
                  bestRow ← i
                  bestCol ← j
               ENDIF
            ENDIF
         ENDFOR
      ENDFOR
   ENDIF

   IF (bestRow != -1 && bestCol != -1)
      grid[bestRow][bestCol] ← PLAYER_O
   ENDIF
```

```
    currentStats ← GetCurrentModeStats()

    IF (CheckWin(PLAYER_O))
        gameOver ← true
        winner ← PLAYER_O
        gameState ← GAME_OVER
        currentStats→aiWins ← currentStats→aiWins + 1
        currentStats→totalGames ← currentStats→totalGames+ 1
        IF (!isTwoPlayer)
            PlaySound(loseSound)
        ELSE
            PlaySound(victorySound)
        ENDIF
    ELSEIF (CheckDraw())
        gameOver ← true
        gameState ← GAME_OVER
        winner ← EMPTY
        currentStats→draws ← currentStats→draws + 1
        currentStats→totalGames ← currentStats→totalGames+ 1
        PlaySound(drawSound)
    ELSE
        currentPlayerTurn ← PLAYER_X_TURN
    ENDIF
ENDFUNCTION

FUNCTION AITurnDecisionTree(victorySound, loseSound, drawSound, *TDmodel)
    bestScore ← -1000
    bestRow ← -1
    bestCol ← -1
    best_prob ← 0.0
    board ← EMPTY 2D ARRAY

    // Convert the grid into a format suitable for the decision tree
    FOR i = 0 to 2 do
        FOR j = 0 to 2 do
            IF grid[i][j] == EMPTY
                board[i][j] ← 'b'
            ELSEIF grid[i][j] == PLAYER_X
                board[i][j] ← 'x'
            ELSEIF grid[i][j] == PLAYER_O
                board[i][j] ← 'o'
            ENDIF
        ENDFOR
    ENDFOR

    print_tree(TDmodel, 2)
    dt_predict_best_move(TDmodel, board, PLAYER_O, &bestRow, &bestCol)

    DO
        row ← GetRandomValue(0, GRID_SIZE - 1)
        col ← GetRandomValue(0, GRID_SIZE - 1)
    WHILE (grid[row][col] != EMPTY)
```

```
    grid[bestRow][bestCol] ← PLAYER_O

    currentStats ← decisionTreeStats

    IF (CheckWin(PLAYER_O))
        gameOver ← true
        winner ← PLAYER_O
        gameState ← GAME_OVER
        currentStats→aiWins ← currentStats→aiWins + 1
        currentStats→totalGames ← currentStats→totalGames + 1
        PlaySound(loseSound)

    ELSE IF (CheckDraw())
        gameOver ← true
        gameState ← GAME_OVER
        winner ← EMPTY
        currentStats→draws ← currentStats→draws + 1
        currentStats→totalGames ← currentStats→totalGames + 1
        PlaySound(drawSound)
    ELSE
        currentPlayerTurn ← PLAYER_X_TURN
    ENDIF
ENDFUNCTION
```

3.7. Check.c File

```
FUNCTION CheckWin(player)
   FOR i = 0 to GRID_SIZE - 1 do
      IF (grid[i][0] == player && grid[i][1] == player && grid[i][2] == player)
         winningCells[0][0] ← i
         winningCells[0][1] ← 0
         winningCells[1][0] ← i
         winningCells[1][1] ← 1
         winningCells[2][0] ← i
         winningCells[2][1] ← 2
         return true
      ENDIF
   ENDFOR
   FOR i = 0 to GRID_SIZE -1 do
      IF (grid[0][i] == player && grid[1][i] == player && grid[2][i] == player)
         winningCells[0][0] ← 0
         winningCells[0][1] ← i
         winningCells[1][0] ← 1
         winningCells[1][1] ← i
         winningCells[2][0] ← 2
         winningCells[2][1] ← i
         return true
      ENDIF
   ENDFOR

   IF (grid[0][0] == player && grid[1][1] == player && grid[2][2] == player)
      winningCells[0][0] ← 0
      winningCells[0][1] ← 0
      winningCells[1][0] ← 1
      winningCells[1][1] ← 1
      winningCells[2][0] ← 2
      winningCells[2][1] ← 2
      return true
   ENDIF
   IF (grid[0][2] == player && grid[1][1] == player && grid[2][0] == player)
      winningCells[0][0] ← 0
      winningCells[0][1] ← 2
      winningCells[1][0] ← 1
      winningCells[1][1] ← 1
      winningCells[2][0] ← 2
      winningCells[2][1] ← 0
      return true
   ENDIF
   return false
ENDFUNCTION

FUNCTION CheckDraw()
   FOR i = 0 to GRID_SIZE - 1 do
      FOR j = 0 to GRID_SIZE - 1 do
         IF (grid[i][j] == EMPTY)
            return false
         ENDIF
```

```
        ENDFOR
    ENDFOR
    return true
ENDFUNCTION
```

3.8. Hint.c File

```
FUNCTION clearHint()
   hint.row ← -1
   hint.col ← -1
ENDFUNCTION

FUNCTION getHint()
   bestScore ← -1000
   bestRow ← -1
   bestCol ← -1
   depthLimit ← 9
   FOR i=0 to GRID_SIZE -1 do
      FOR j = 0 to GRID_SIZE -1 do
         IF (grid[i][j] == EMPTY)
            grid[i][j] ← PLAYER_O
            socre ← Minimax(grid, false, 0, depthLimit, -1000, 1000)
            grid[i][j] ← EMPTY
            IF (score > bestScore)
               bestScore ← score
               bestRow ← i
               bestCol ← j
            ENDIF
         ENDIF
      ENDFOR
   ENDFOR
   IF (bestRow != -1 && bestCol != -1)
      hint.row ← bestRow
      hint.col ← bestCol
   ENDIF
ENDFUNCTION
```

3.9. Minimax.c File

```
FUNCTION Minimax(board, isMaximizing, depth, depthLimit, alpha, beta)
   IF (depth >= depthLimit)
      return 0
   ENDIF
   score ← EvaluateBoard(board)
   IF (score == 10)
      return score - depth
   ENDIF
   IF (sore == -10)
      return score + depth
   ENDIF
   IF (CheckDraw())
      return 0
   ENDIF
   IF (isMaximizing)
      bestScore ← -1000
      FOR i = 0 to GRID_SIZE - 1 do
         FOR j = 0 to GRID_SIZE -1 do
            IF (board[i][j] == EMPTY)
               board[i][j] ← PLAYER_O
                   bestScore ← fmax(bestScore, Minimax(board, false, depth + 1, depthLimit, alpha,
beta))
               board[i][j] ← EMPTY
               alpha ← fmax(alpha, bestScore)
               IF (beta <= alpha)
                  BREAK
               ENDIF
            ENDIF
         ENDFOR
      ENDFOR
      return bestScore
   ELSE
      bestScore ← 1000
      FOR i = 0 to GRID_SIZE - 1 do
         FOR j = 0 to GRID_SIZE - 1 do
            IF (board[i][j] == EMPTY)
               board[i][j] ← PLAYER_X
                    bestScore ← fmin(bestScore, Minimax(board, true, depth + 1, depthLimit, alpha,
beta))
               board[i][j] ← EMPTY
               beta ← fmin(beta, bestScore)
               IF (beta <= alpha)
                  BREAK
               ENDIF
            ENDIF
         ENDFOR
      ENDFOR
      return bestScore
   ENDIF
ENDFUNCTION
```

```
FUNCTION EvaluateBoard(board)
   FOR row = 0 to GRID_SIZE - 1 do
      IF (board[row][0] == board[row][1] && board[row][0] == board[row][2])
         IF (board[row][0] == PLAYER_O)
            return 10
         ELSEIF (board[row][0] == PLAYER_X)
            return -10
         ENDIF
      ENDIF
   ENDFOR
   FOR col = 0 to GRID_SIZE - 1 do
      IF (board[0][col] == board[1][col] && board[0][col] == board[2][col])
         IF (board[0][col] == PLAYER_O)
            return 10
         ELSEIF (board[0][col] == PLAYER_X)
            return -10
         ENDIF
      ENDIF
   ENDFOR

   IF (board[0][0] == board[1][1] && board[0][0] == board[2][2])
      IF (board[0][0] == PLAYER_O)
         return 10
      ELSEIF (board[0][0] == PLAYER_X)
         return -10
      ENDIF
   ENDIF
   IF (board[0][2] == board[1][1] && board[0][2] == board[2][0])
      IF (board[0][2] == PLAYER_O)
         return 10
      ELSEIF (board[0][2] == PLAYER_X)
         return -10
      ENDIF
   ENDIF
   return 0
ENDFUNCTION
```

3.10. data_processing.c File

```
FUNCTION load_data(filename, boards, outcomes, total_records)
   file_ptr refToFile → &filename
   total_records refToInt → &total_records

   OPEN FILE file_ptr
   IF (file_ptr = NULL)
      PRINT "Failed to open file"
   ENDIF

   WHILE (READ FILE *file_ptr)
      line[50] ← *file_ptr
      board[10] ← {line[0] to line[9]}
      outcome[10] ← {line[10] to line[19]}

      boards[total_records] ← board[10]
      outcomes[total_records] ← outcome_index(outcome)
      total_records ← total_records + 1
   ENDWHILE
   CLOSE FILE file_ptr
ENDFUNCTION

FUNCTION split_data(boards, outcomes, total_records, train_boards, train_outcomes,
test_boards, test_outcomes, train_size, test_size, ratio)
   train_size refToInt → &train_size
   test_size refToInt → &test_size

   FOR i = (total_records - 1) down to 1 do
      j ← rand() % (i + 1)

      temp_board ← boards[i]
      boards[i] ← boards[j]
      boards[j] ← temp_board

      temp_outcome ← outcomes[i]
      outcomes[i] ← outcomes[j]
      outcomes[j] ← temp_outcome
   ENDFOR

   target_train_size ← ratio * total_records

   FOR i = 0 to total_records do
      IF (*train_sisze < target_train_size)
         train_boards[*train_size] ← boards[i]
         train_outcomes{*train_size} ← outcomes[i]
         *train_size ← *train_size + 1
      ELSE
         test_boards[*test_size] ← boards[i]
         test_outcomes{*test_size} ← outcomes[i]
         *test_size ← *test_size + 1
      ENDIF
   ENDFOR
```

```
   PRINT "train_boards array:"
   FOR i = 0 to 10 do
      PRINT train_boards[i]
   ENDFOR

   PRINT "train_outcomes array:"
   FOR i = 0 to 10 do
      PRINT train_outcomes[i]
   ENDFOR
ENDFUNCTION

FUNCTION outcome_index(outcome)
   IF (outcome = "positive")
      return POSITIVE
   ELSE
      return NEGATIVE
   ENDIF
ENDFUNCTION
```

3.11. NBmodel.c File

```
FUNCTION train_NBmodel(model, boards, outcomes, size)
   model refToNaiveBayesModel → &NBmodel

   positive_count ← 0
   negative_count ← 0

   x_counts[NUM_POSITIONS][NUM_OUTCOMES] ← {0}
   o_counts[NUM_POSITIONS][NUM_OUTCOMES] ← {0}
   b_counts[NUM_POSITIONS][NUM_OUTCOMES] ← {0}

   FOR i = 0 to size do
      outcome_idx ← outcomes[i]
      IF (outcome_idx = POSITIVE)
         positive_count ← positive_count + 1
      ELSE
         negative_count ← negative_count + 1
      ENDIF

      FOR j = 0 to NUM_POSITIONS do
         IF (boards[i][j] = 'x')
            x_counts[j][outcome_idx] ← x_counts[j][outcome_idx] + 1
         ELSEIF (boards[i][j] = 'o')
            o_counts[j][outcome_idx] ← o_counts[j][outcome_idx] + 1
         ELSE
            b_counts[j][outcome_idx] ← b_counts[j][outcome_idx] + 1
         ENDIF
      ENDFOR
   ENDFOR

   PRINT "X_counts array"
   PRINT " pos     neg"
   FOR i = 0 to NUM_POSITIONS do
      PRINT "Position (i+1) ["
      FOR j = 0 to NUM_OUTCOMES do
         PRINT x_counts[i][j]
      ENDFOR
      PRINT "]"
   ENDFOR

   PRINT "o_counts array"
   PRINT " pos     neg"
   FOR i = 0 to NUM_POSITIONS do
      PRINT "Position (i+1) ["
      FOR j = 0 to NUM_OUTCOMES do
         PRINT o_counts[i][j]
      ENDFOR
      PRINT "]"
   ENDFOR

   PRINT "b_counts array"
   PRINT " pos     neg"
```

```
   FOR i = 0 to NUM_POSITIONS do
      PRINT "Position (i+1) ["
      FOR j = 0 to NUM_OUTCOMES do
         PRINT b_counts[i][j]
      ENDFOR
      PRINT "]"
   ENDFOR

   model→class_probs[POSITIVE] ← positive_count / size
   model→class_probs[NEGATIVE] ← negative_count / size

   FOR i = 0 to NUM_POSITIONS do
      model→ x_probs[i][POSITIVE] ← (x_counts[i][POSITIVE] + 1) / (positive_count + 3)
      model→ x_probs[i][NEGATIVE] ← (x_counts[i][NEGATIVE] + 1) / (negative_count + 3)

      model→ o_probs[i][POSITIVE] ← (o_counts[i][POSITIVE] + 1) / (positive_count + 3)
      model→ o_probs[i][NEGATIVE] ← (o_counts[i][NEGATIVE] + 1) / (negative_count + 3)

      model→ b_probs[i][POSITIVE] ← (b_counts[i][POSITIVE] + 1) / (positive_count + 3)
      model→ b_probs[i][NEGATIVE] ← (b_counts[i][NEGATIVE] + 1) / (negative_count + 3)
   ENDFOR
ENDFUNCTION

FUNCTION save_NBmodel(model, filename)
   model refToNaiveBayesModel → &NBmodel

   file_ptr refToFile → &filename
   OPEN FILE file_ptr
   IF (file_ptr = NULL)
      PRINT "Failed to open file"
   ENDIF

   WRITE "Class Probabilities"
   WRITE "P(Positive):", model→class_probs[POSITIVE]
   WRITE "P(Negative):", model→class_probs[negative]

   FOR i = 0 to NUM_POSITIONS do
      WRITE "Position " (i+1)
      WRITE "P(x | Positive):", model→x_probs[i][POSITIVE]
      WRITE "P(x | Negative):", model→x_probs[i][NEGATIVE]
      WRITE "P(o | Positive):", model→o_probs[i][POSITIVE]
      WRITE "P(o | Negative):", model→o_probs[i][NEGATIVE]
      WRITE "P(b | Positive):", model→b_probs[i][POSITIVE]
      WRITE "P(b | Negative):", model→b_probs[i][NEGATIVE]
   ENDFOR

   CLOSE FILE file_ptr

   PRINT "Model weights saved to " filename
ENDFUNCTION

FUNCTION test_NBmodel(filename, mode, type, model, boards, outcomes, size)
   file_ptr refToFile → &filename
```

```
    model refToNaiveBayesModel → &NBmodel

    true_positive ← 0
    false_positive ← 0
    true_negative ← 0
    false_negative ← 0
    error_count ← 0

    FOR i = 0 to size do
        predicted_outcome ← predict_outcome(model, boards[i])
        IF (outcomes[i] = POSITIVE && predicted_outcome = POSITIVE)
            true_positive ← true_positive + 1
        ELSEIF (outcomes[i] = POSITIVE && predicted_outcome = NEGATIVE)
            false_negative ← false_negative + 1
            error_count ← error_count + 1
        ELSEIF (outcomes[i] = NEGATIVE && predicted_outcome = NEGATIVE)
            true_negative ← true_negative + 1
        ELSE
            false_positive ← false_positive + 1
            error_count ← error_count + 1
        ENDIF
    ENDFOR

    prob_of_error ← error_count / size * 100

    OPEN FILE file_ptr
    IF (file_ptr = NULL)
        PRINT "Failed to open file"
    ENDIF

    IF (type = "Testing)
        WRITE "\n\n"
    ENDIF

    WRITE "Dataset:", type
    WRITE "Accuracy:", (100 - prob_of_error), (size - error_count), size
    WRITE "Error:", prob_of_error, error_count, size
    WRITE "Confusion Matrix"
    WRITE "True Positive:", true_positive
    WRITE "False Positive:", false_positive
    WRITE "True Negative:", true_negative
    WRITE "False Negative:", false_negative

    CLOSE FILE file_ptr
ENDFUNCTION

FUNCTION calculate_probability(model, board, outcome)
    model refToNaiveBayesModel → &NBmodel

    probability ← model→class_probs[outcome]

    FOR i = 0 to NUM_POSITIONS do
        IF (board[i] = 'x')
```

```
            probability ← probability * model→x_probs[i][outcome]
        ELSEIF (board[i] = 'o')
            probability ← probability * model→o_probs[i][outcome]
        ELSE
            probability ← probability * model→b_probs[i][outcome]
        ENDIF
    ENDFOR
    return probability
ENDFUNCTION

FUNCTION predict_outcome(model, board)
    model refToNaiveBayesModel → &NBmodel

    positive_prob ← calculate_probability(model, board, POSITIVE)
    negative_prob ← calculate_probability(model, board, NEGATIVE)

    IF (positive_prob > negative_prob)
        return POSITIVE
    ELSE
        return NEGATIVE
    ENDIF
ENDFUNCTION

FUNCTION predict_move(model, grid[GRID_SIZE][GRID_SIZE], bestRow, bestCol)
    model refToNaiveBayesModel → &NBmodel
    bestRow refToInt → &bestRow
    bestCol refToInt → &bestCol

    best_move ← -1
    best_prob ← 0.0
    k ← 0

    PRINT "AI's Turn"
    PRINT "Game board layout as grid(array) format:"

    FOR i = 0 to GRID_SIZE do
        PRINT "["
        FOR j = 0 to GRID_SIZE do
            IF (grid[i][j] = EMPTY)
                board[k] ← 'b'
                PRINT "b"
            ELSEIF (grid[i][j] = PLAYER_O)
                board[k] ← 'o'
                PRINT "o"
            ELSE
                board[k] ← 'x'
                PRINT "x"
            ENDIF
            k ← k + 1
        ENDFOR
        PRINT "]"
    ENDFOR
```

```
    PRINT "Game board layout as string:"
    PRINT board

    PRINT "Simulated move       Simulated board       Posterior Probabilities"

    FOR i = 0 to NUM_POSITIONS do
        IF (board[i] = 'b')
            temp_board ← board
            temp_board[i] ← 'x'

            positive_prob ← calculate_probability(model, temp_board, POSITIVE)

            IF (positive_prob > best_prob)
                best_prob ← positive_prob
                best_move ← i
            ENDIF

            PRINT i, temp_board, positive_prob
        ENDIF
    ENDFOR

    divide(best_move, 3, bestRow, bestCol)

    PRINT "Best move: %best_move -> (%bestRow, %bestCol)"

    return 0
ENDFUNCTION

FUNCTION divide(dividend, divisor, quo, rem)
    quo refToInt → &quo
    rem refToInt → &rem

    *quo ← dividend / divisor
    *rem ← dividend % divisor
ENDFUNCTION
```

3.12. decisiontree.c File

```
FUNCTION growth_Tree(tree)
    dataset_size, train_size, test_size ← 0
    train_confusion[2][2], test_confusion[2][2] ← {0}
    train_accuracy, test_accuracy AS FLOAT ← 0.0
    train_error_rate, test_error_rate ← 0.0
    correct_train, correct_test ← 0

    srand(time(NULL))

    load_dataset("tic-tac-toe.data", dataset, &dataset_size)

    shuffle_dataset(dataset, dataset_size)

    decision_tree_split_dataset(dataset, dataset_size, train_set, &train_size, test_set, &test_size, 0.8)

    tree ← build_tree(train_set, train_size, 0)

    calculate_position_probabilities(dataset, dataset_size, "DecisionTree_ML/DTweights.txt")

    file ← open("DecisionTree_ML/DTconfusion_matrix.txt", "w")
    IF file IS NOT NULL
        close(file)
    ENDIF

    evaluate_with_randomness(tree, train_set, train_size, train_confusion) → train_accuracy
    correct_train = train_accuracy * train_size
    display_confusion_matrix(train_confusion, "DecisionTree_ML/DTconfusion_matrix.txt", "Training")
    write_accuracy_to_file("DecisionTree_ML/DTconfusion_matrix.txt", "Training", train_accuracy, correct_train, train_size)

    train_error_rate ← calculate_error_rate(tree, train_set, train_size, train_confusion)
    file ← open("DecisionTree_ML/DTconfusion_matrix.txt", "a")
    IF file IS NOT NULL
        WRITE file, "Training Error Rate: train_error_rate"
        close(file)
    ENDIF

    test_accuracy ← evaluate_with_randomness(tree, test_set, test_size, test_confusion)
    correct_test ← int(test_accuracy * test_size)
    display_confusion_matrix(test_confusion, "DecisionTree_ML/DTconfusion_matrix.txt", "Testing")
    write_accuracy_to_file("DecisionTree_ML/DTconfusion_matrix.txt", "Testing", test_accuracy, correct_test, test_size)

    test_error_rate ← calculate_error_rate(tree, test_set, test_size, test_confusion)
    file ← open("DecisionTree_ML/DTconfusion_matrix.txt", "a")
    IF file IS NOT NULL
        WRITE file, "Testing Error Rate: test_error_rate"
        close(file)
```

```
   ENDIF
ENDFUNCTION

FUNCTION load_dataset(filename, dataset, dataset_size)
   file ← open(filename, "r")
   IF (!file)
      PRINT "Failed to open file"
      EXIT(1)
   ENDIF

   dataset_size ← 0
   WHILE (line ← read_line(file)) do
      tokens ← split(line, ",")
      FOR i = 0 to NUM_FEATURES - 1 do
         dataset[dataset_size].features[i] ←
            (tokens[i] == "x") ? 1 : (tokens[i] == "o") ? 2 : 0
      ENDFOR
      dataset[dataset_size].label ←
         (tokens[NUM_FEATURES] == "positive") ? DT_POSITIVE : DT_NEGATIVE
      dataset_size ← dataset_size + 1
   ENDWHILE
   close(file)
ENDFUNCTION

FUNCTION shuffle_dataset(dataset, size)
   FOR i = size - 1 to 1 do
      j ← random(0, i)
       swap(dataset[i], dataset[j])
   ENDFOR
ENDFUNCTION

FUNCTION build_tree(dataset, size, depth)
   positives ← count(dataset, DT_POSITIVE)
   negatives ← count(dataset, DT_NEGATIVE)

   IF (depth >= MAX_DEPTH OR positives == 0 OR negatives == 0)
      leaf ← new DecisionTreeNode
      leaf.is_leaf ← TRUE
      leaf.prediction ← (positives > negatives) ? DT_POSITIVE : DT_NEGATIVE
      return leaf
   ENDIF

   best_feature ← -1, best_threshold ← -1
   best_gini ← 1.0
   left ← ARRAY[MAX_ROWS], right ← ARRAY[MAX_ROWS]
   left_size ← 0, right_size ← 0

   FOR feature_index = 0 to NUM_FEATURES - 1 do
      FOR threshold = 0 to 2 do
         gini ← calculate_gini_index(dataset, size, feature_index, threshold)
         IF (gini < best_gini)
            best_gini ← gini
            best_feature ← feature_index
```

```
          best_threshold ← threshold
        ENDIF
      ENDFOR
    ENDFOR

    WRITE "Splitting at Feature:", best_feature, "Threshold:", best_threshold
      decision_tree_split_data(dataset, size, best_feature, best_threshold, left, &left_size, right,
&right_size)

    node ← new DecisionTreeNode
    node.is_leaf ← FALSE
    node.feature_index ← best_feature
    node.threshold ← best_threshold
    node.left ← build_tree(left, left_size, depth + 1)
    node.right ← build_tree(right, right_size, depth + 1)
    return node
ENDFUNCTION

FUNCTION evaluate_with_randomness(root, dataset, size, confusion_matrix)
    correct_predictions ← 0

    FOR i = 0 TO 1 do
      FOR j = 0 TO 1 do
        confusion_matrix[i][j] ← 0
      ENDFOR
    ENDFOR

    FOR i = 0 TO size - 1 do
      prediction ← predict_with_randomness(root, dataset[i].features)
      actual ← dataset[i].label

      IF (actual == DT_POSITIVE AND prediction == DT_POSITIVE)
        confusion_matrix[0][0] ← confusion_matrix[0][0] + 1
        correct_predictions ← correct_predictions + 1
      ELSE IF (actual == DT_NEGATIVE AND prediction == DT_NEGATIVE)
        confusion_matrix[1][1] ← confusion_matrix[1][1] + 1
        correct_predictions ← correct_predictions + 1
      ELSE IF (actual == DT_NEGATIVE AND prediction == DT_POSITIVE)
        confusion_matrix[1][0] ← confusion_matrix[1][0] + 1
      ELSE IF (actual == DT_POSITIVE AND prediction == DT_NEGATIVE)
        confusion_matrix[0][1] ← confusion_matrix[0][1] + 1
      ENDIF
    ENDFOR
    return correct_predictions / size
ENDFUNCTION

FUNCTION predict_with_randomness(node, features)
    IF node IS NULL
      return DT_NEGATIVE
    ENDIF

    IF node.is_leaf
      RANDOM_VALUE ← random() / RAND_MAX
```

```
        IF RANDOM_VALUE < RANDOMNESS_FACTOR
            return (node.prediction == DT_POSITIVE) ? DT_NEGATIVE : DT_POSITIVE
        ENDIF
        return node.prediction
    ENDIF

    IF features[node.feature_index] ≤ node.threshold
        return predict_with_randomness(node.left, features)
    ELSE
        return predict_with_randomness(node.right, features)
    ENDIF
ENDFUNCTION

FUNCTION display_confusion_matrix(confusion_matrix, filename, dataset_type)
    file ← open(filename, "append")
    IF file IS NULL
        perror("Failed to open confusion matrix file")
        return
    ENDIF

    TP ← confusion_matrix[0][0]
    FP ← confusion_matrix[1][0]
    TN ← confusion_matrix[1][1]
    FN ← confusion_matrix[0][1]

    WRITE(file, "\nDecision Tree ", dataset_type, " Confusion Matrix:\n")
    WRITE "  True Positive:", confusion_matrix[0][0]
    WRITE "  False Positive:", confusion_matrix[1][0]
    WRITE "  True Negative:", confusion_matrix[1][1]
    WRITE "  False Negative:", confusion_matrix[0][1]
    WRITE(file, "\nConfusion Matrix:\n")
    WRITE(file, "                Predicted Positive    Predicted Negative\n")
    WRITE(file, "Actual Positive        ", TP, "                  ", FN, "\n")
    WRITE(file, "Actual Negative        ", FP, "                  ", TN, "\n")
    WRITE(file, "-------------------------------------------------------\n")

    close(file)
ENDFUNCTION

FUNCTION write_accuracy_to_file(filename, dataset_type, accuracy, correct, total)
    file ← open(filename, "append")
    IF file IS NULL
        perror("Failed to open file for writing accuracy")
        return
    ENDIF

    WRITE(file, dataset_type, " Accuracy: ", FORMAT(accuracy * 100, 2), "% (", correct, "/", total, ")\n")
    close(file)
ENDFUNCTION

FUNCTION free_tree(node)
    IF node IS NULL
```

```
      return
   ENDIF
    free_tree(node.left)
    free_tree(node.right)
    free(node)
ENDFUNCTION

FUNCTION calculate_gini_index(dataset, size, feature_index, threshold)
   left ← ARRAY[MAX_ROWS]
   right ← ARRAY[MAX_ROWS]
   left_size ← 0
   right_size ← 0

         decision_tree_split_data(dataset, size, feature_index, threshold, left, &left_size, right,
&right_size)

   IF left_size = 0 OR right_size = 0
      return 1.0
   ENDIF

   positives_left ← 0
   positives_right ← 0

   FOR i = 0 TO left_size - 1 do
      IF left[i].label = DT_POSITIVE
         positives_left ← positives_left + 1
      ENDIF
   ENDFOR

   FOR i = 0 TO right_size - 1 do
      IF right[i].label = DT_POSITIVE
         positives_right ← positives_right + 1
      ENDIF
   ENDFOR

   prob_left ← positives_left / left_size
   gini_left ← 1.0 - (prob_left * prob_left) - ((1.0 - prob_left) * (1.0 - prob_left))

   prob_right ← positives_right / right_size
   gini_right ← 1.0 - (prob_right * prob_right) - ((1.0 - prob_right) * (1.0 - prob_right))

   return ((gini_left * left_size) + (gini_right * right_size)) / size
ENDFUNCTION

FUNCTION decision_tree_split_data(dataset, size, feature_index, threshold, left, left_size, right,
right_size)
   left_size ← 0
   right_size ← 0

   FOR i = 0 TO size - 1 do
      IF dataset[i].features[feature_index] <= threshold
         left[left_size] ← dataset[i]
         left_size ← left_size + 1
```

```
        ELSE
            right[right_size] ← dataset[i]
            right_size ← right_size + 1
        ENDIF
    ENDFOR
ENDFUNCTION

FUNCTION dt_predict_best_move(tree, board, current_player, best_row, best_col)
    IF tree IS NULL
        PRINT "Error: Decision tree is not initialized!"
        return
    ENDIF

    features[NUM_FEATURES]
    max_positive_prob ← -1
    best_row ← -1
    best_col ← -1
    attempts ← 0

    FOR i = 0 TO 2 do
        FOR j = 0 TO 2 do
            IF board[i][j] = 'x'
                features[i * 3 + j] ← 1
            ELSE IF board[i][j] = 'o'
                features[i * 3 + j] ← 2
            ELSE
                features[i * 3 + j] ← 0
            ENDIF
        ENDFOR
    ENDFOR

    FOR attempts = 0 TO 4 do
        temp_row ← -1
        temp_col ← -1

        FOR i = 0 TO 2 do
            FOR j = 0 TO 2 do
                IF board[i][j] = 'b'
                    features[i * 3 + j] ← (current_player = 'x') ? 1 : 2

                    prediction ←  predict_with_randomness(tree, features)

                            IF prediction = DT_POSITIVE AND (max_positive_prob = -1 OR prediction >
max_positive_prob)
                        temp_row ← i
                        temp_col ← j
                        max_positive_prob ← prediction
                    ENDIF

                    features[i * 3 + j] ← 0
                ENDIF
            ENDFOR
        ENDFOR
```

```
        IF temp_row != -1 AND temp_col != -1
            best_row ← temp_row
            best_col ← temp_col
            return
        ENDIF
    ENDFOR

    FOR i = 0 TO 2 do
        FOR j = 0 TO 2 do
            IF board[i][j] = 'b'
                best_row ← i
                best_col ← j
                return
            ENDIF
        ENDFOR
    ENDFOR
ENDFUNCTION

FUNCTION print_tree(node, depth)
    IF node IS NULL
        return
    ENDIF

    IF node.is_leaf
        PRINT "Leaf: Prediction =", node.prediction
    ELSE
        PRINT "Node: Feature =", node.feature_index, ", Threshold =", node.threshold
         print_tree(node.left, depth + 1)
         print_tree(node.right, depth + 1)
    ENDIF
ENDFUNCTION

FUNCTION calculate_position_probabilities(dataset, dataset_size, filename)
    positive_count ← 0
    negative_count ← 0
    position_count[NUM_FEATURES][3][2] ← 0

    FOR i = 0 TO dataset_size - 1 do
        IF dataset[i].label = DT_POSITIVE
            positive_count ← positive_count + 1
        ELSE
            negative_count ← negative_count + 1
        ENDIF

        FOR j = 0 TO NUM_FEATURES - 1 do
            IF dataset[i].features[j] = 1
                position_count[j][0][dataset[i].label] ← position_count[j][0][dataset[i].label] + 1
            ELSE IF dataset[i].features[j] = 2
                position_count[j][1][dataset[i].label] ← position_count[j][1][dataset[i].label] + 1
            ELSE
                position_count[j][2][dataset[i].label] ← position_count[j][2][dataset[i].label] + 1
            ENDIF
```

```
    ENDFOR
  ENDFOR

  file ← open(filename, "w")
  IF file IS NULL
    return
  ENDIF

   write(file, "Class Probabilities:")
   write(file, "  Positive: P(Positive) =", positive_count / dataset_size)
   write(file, "  Negative: P(Negative) =", negative_count / dataset_size)
   write(file, "-------------------------------------------")

  FOR i = 0 TO NUM_FEATURES - 1 do
     write(file, "Position", i + 1, ":")
     write(file, "  Symbol | P(Symbol | Positive) | P(Symbol | Negative)")
     write(file, "  -------|----------------------|----------------------")

    FOR j = 0 TO 2 do
        p_positive ← IF positive_count > 0 position_count[i][j][DT_POSITIVE] / positive_count
ELSE 0.0
       p_negative ← IF negative_count > 0 position_count[i][j][DT_NEGATIVE] / negative_count
ELSE 0.0
      write(file, "  ", symbols[j], "|", p_positive, "|", p_negative)
    ENDFOR

     write(file, "-------------------------------------------")
  ENDFOR

   close(file)
  PRINT "Weights updated and saved to", filename
ENDFUNCTION

FUNCTION calculate_error_rate(root, dataset, size, confusion_matrix)
  error_count ← 0

  FOR i = 0 TO size - 1 do
    prediction ← predict_with_randomness(root, dataset[i].features)
    actual ← dataset[i].label

    IF prediction ≠ actual
      error_count ← error_count + 1
    ENDIF
  ENDFOR

  return (error_count / size) * 100
ENDFUNCTION
```

4. Function Descriptions

This section describes the purposes of the functions.

4.1. Game Logic Functions

`InitGame()`: Initializes the game state and grid.

`HandlePlayerTurn()`: Manages the player's turn and checks for game outcomes.

`HandleButtonHover()`: Manages the hover status of buttons.

`AITurn()`: Handles the AI's turn in single-player mode.

`UpdateGame()`: Updates the game state based on player input and game logic.

`UpdateGameOver()`: Manages the game over state, allowing players to retry or return to the menu.

`CheckWin()`: Checks if a specified player has won the game.

`CheckDraw()`: Checks if the game is a draw.

`clearHint()`: Removing best move from the previous turn.

`getHint()`: Get best move for the current player.

`GetCurrentModeStats()`: Tracks game statistics (wins, losses and draws) for each difficulty mode and AI model. This function provides a clean way to access the correct stats tracker. It is used throughout the game to update and display statistics for the currently active game mode.

4.2. UI and Animation Functions

`DrawButton()`: Draws buttons with optional hover and vibration effects.

`DrawSymbols()`: Draws the falling symbols on the screen.

`DrawTitleWords()`: Draws the animated title words.

`DrawGame()`: Renders the game grid, symbols, and UI elements.

`DrawMenu()`: Draws the interactive main menu that serves as the entry point to different game modes while maintaining visual appeal through animations and interactive elements.

`DrawConfetti()`: Draws the confetti particles on the screen.

`DrawDifficultySelect()`: Draws an intuitive interface for players to select their preferred AI difficulty level while maintaining visual consistency with the rest of the game's UI design.

`DrawModelSelect()`: Draws a clean interface for players to choose between different AI models when playing in Easy mode.

`DrawGameOver()`: Draws a clear and engaging end-game screen that celebrates the outcome while providing intuitive options to continue playing or return to the menu.

`InitConfetti()`: Initializes the confetti particles for the animated UI upon winning.

`InitSymbols()`: Initializes the falling symbols for the animated UI.

`InitTitleWords()`: Initializes the title words for the animated title.

`UpdateConfetti()`: Updates the position and rotation of confetti particles.

`UpdateSymbols()`: Updates the position and rotation of falling symbols.

`UpdateTitleWords()`: Updates the animation state of the title words.

4.3. Data Processing Functions

`load_data()`: Loads data from file.

`split_data()`: Shuffle and split dataset for training and testing of the model.

`outcome_index()`: Convert the string outcome ("positive" or "negative") into the corresponding numerical label (POSITIVE(0) or NEGATIVE(1)).

4.4. AI Functions

`Minimax()`: Implements the Minimax Algorithm for AI decision-making.

`EvaluateBoard()`: Evaluates the board to determine the score for the AI.

`train_NBmodel()`: Trains model with Naive Bayes(NB) algorithm.

`save_NBmodel()`: Saves the weights of the NB model into a text file.

`test_NBmodel()`: Saves the prediction results of the trained NB model into a text file.

`calculate_probability()`: Calculate the posterior probability of a specified outcome based on the given board layout.

`predict_outcome()`: Predicts the outcome of a given board layout.

`predict_move()`: Predicts the next best move based on the given board layout.

`divide()`: Get the quotient and remainder of a given integer.

`growth_Tree()`: Builds, trains, and evaluates a decision tree on a dataset, calculates accuracy and error rates, and writes results to files.

`load_dataset()`: Loads the dataset from a file, parses it into features and labels, and stores it in an array.

`shuffle_dataset()`: Randomly shuffles the dataset to ensure random distribution of samples.

`decision_tree_split_dataset()`: Splits the dataset into training and testing subsets based on a specified ratio.

`build_tree()`: Constructs a decision tree by recursively splitting the dataset using the Gini index and applying depth or purity stopping conditions.

`evaluate_with_randomness()`: Evaluates the decision tree's accuracy with randomized predictions and updates a confusion matrix.

`predict_with_randomness()`: Predicts a label using a decision tree with an optional randomness factor to flip predictions.

`display_confusion_matrix()`: Writes a confusion matrix and associated metrics to a file.

`write_accuracy_to_file()`: Writes the accuracy and classification results for training or testing datasets to a file.

`free_tree()`: Recursively frees memory allocated for the decision tree nodes.

`calculate_gini_index()`: Calculates the Gini index to evaluate the quality of a potential split in the dataset.

`decision_tree_split_data()`: Splits the dataset into left and right branches based on a feature index and threshold.

`dt_predict_best_move()`: Predicts the best move for a player in a tic-tac-toe board using the decision tree model.

`print_tree()`: Recursively prints the structure of the decision tree, including nodes and leaf predictions.

`calculate_position_probabilities()`: Calculates and saves the probabilities of each symbol ('x', 'o', 'b') at each position for positive and negative outcomes.

`calculate_error_rate()`: Calculates the error rate of a decision tree by comparing its predictions to the actual labels.

4.5. Raylib Functions

`InitWindow()`: Initializes the game window.

`CloseWindow()`: Closes the game window.

`WindowShouldClose()`: Check if the application should close (KEY_ESCAPE pressed or windows close icon clicked or “Exit” button clicked).

`BeginDrawing()`: Begins the drawing process.

`EndDrawing()`: Ends the drawing process.

`ClearBackground()`: Clears the screen with a specified background colour, effectively resetting the drawing canvas for the current frame.

`DrawText()`: Draws text (using default font).

`DrawTextPro()`: Draws text using Font and pro parameters (rotation).

`DrawRectangle()`: Draws a color-filled rectangle.

`DrawRectangleRec()`: Draws a rectangle on the screen.

`DrawRectangleLinesEx()`: Draws the outline of a rectangle.

`DrawLine()`: Draws a line.

`GetMousePosition()`: Retrieves the current mouse position.

`IsMouseButtonPressed()`: Checks if a mouse button is pressed.

`CheckCollisionPointRec()`: Checks if a point is within a rectangle.

`PlaySound()`: Plays a sound.

`StopSound()`: Stops playing a sound.

`SetSoundVolume()`: Set the sound volume (between 0.0f (min) to 1.0f (max)).

`LoadSound()`: Loads a sound file.

`UnloadSound()`: Unloads a sound file.

`InitAudioDevice()`: Initializes the audio device.

`CloseAudioDevice()`: Closes the audio device.

`IsSoundPlaying()`: Check if a sound is currently playing.

`LoadImage()`: Load image from file into CPU memory (RAM).

`SetWindowIcon()`: Set icon for window (single image, RGBA 32bit).

`UnloadImage()`: Unload the image from CPU memory (RAM).

`SetMouseCursor()`: Set mouse cursor.

`MeasureText()`: Measures string width for default font.

`GetFontDefault()`: Get the default Font.

`GetRandomValue()`: Get a random value between min and max (both included).

`GetFrameTime()`: Get time in seconds for the last frame drawn (delta time).

`GetTime()`: Get elapsed time in seconds since InitWindow().

4.6. Standard Functions

`sprintf(char *str, const char *format, ...)`: Sends formatted output to a string.

`int snprintf(char *__restrict__ __stream, size_t __n, const char *__restrict__ __format, ...)`: Writes formatted output to a string.

`sinf (float x)`: Computes the sine (specified in radians) of x.

`*memset(void *str, int c, size_t n)`: Returns a pointer to the memory area string.

5. Program Interfaces

This section shows the interfaces of the program.

Main Menu:

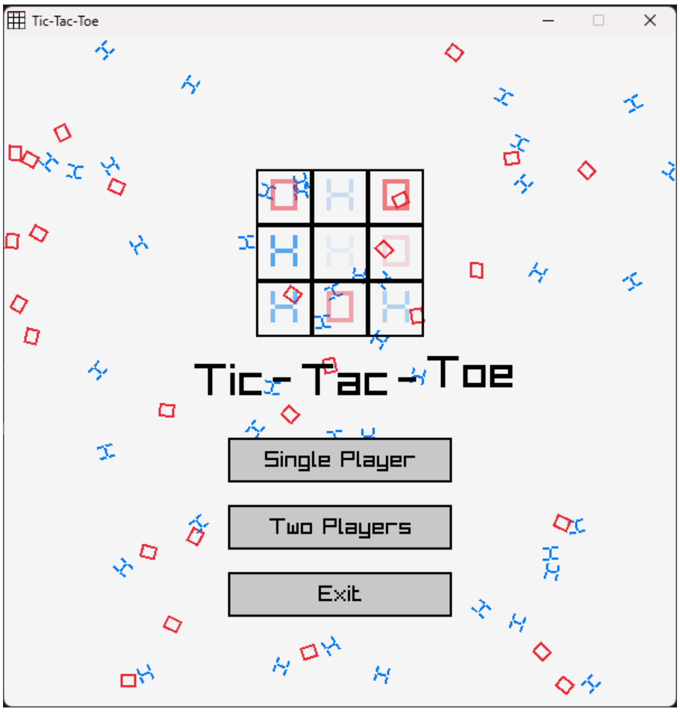


Fig.1: Main Menu

Difficulty Selection:

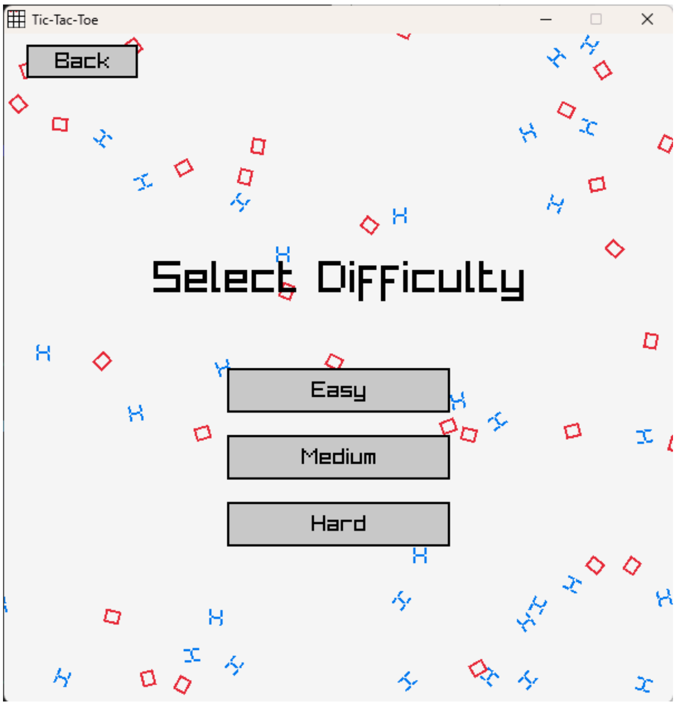


Fig.2: Difficulty Selection

Model Selection for Easy Mode:

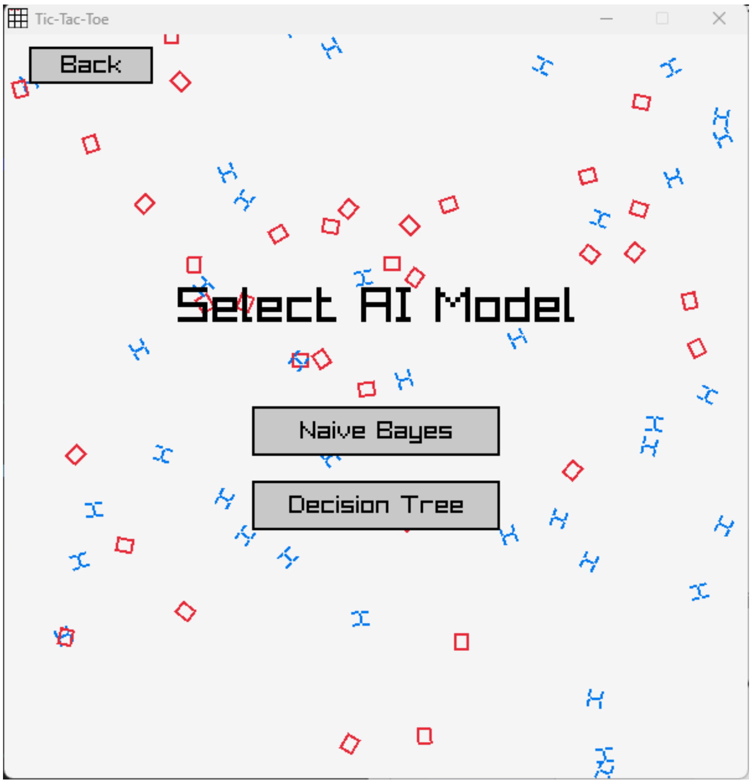


Fig.3: Model Selection

Single-Player Mode:

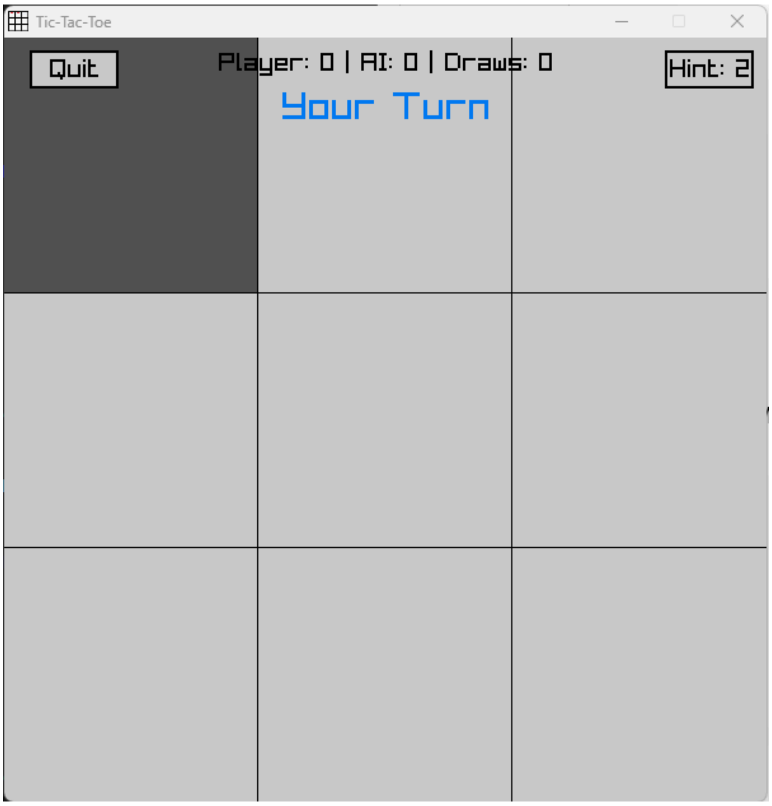


Fig.4: Single-Player Mode

Two-Player Mode:

Fig.5: Two-Player Mode

Game Over Screen:

Fig.6: Game Over

Winning Screen for Single-Player Mode:

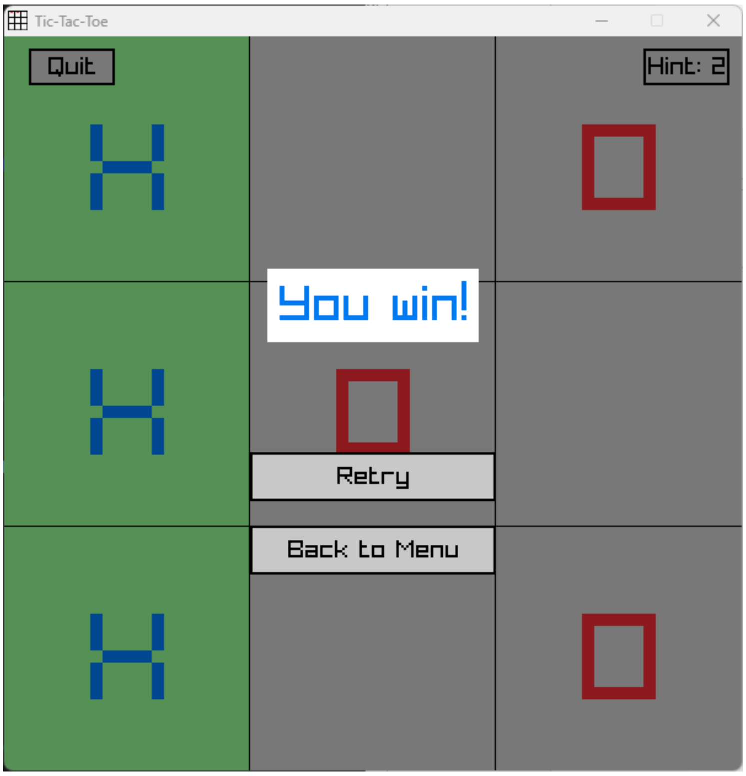


Fig.7: Winning Screen

Losing Screen for Single-Player Mode:

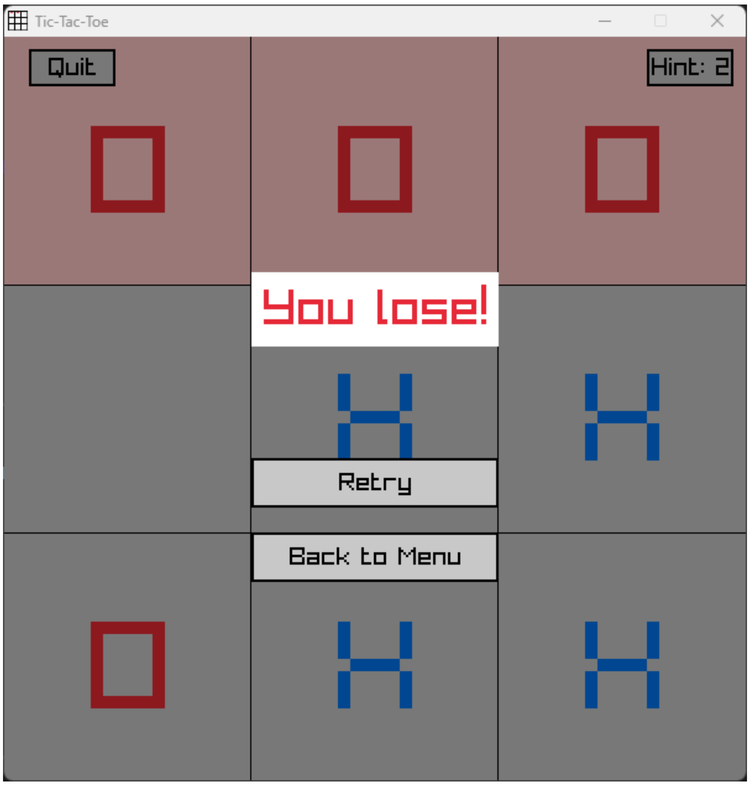


Fig.8: Losing Screen

Draw Screen:

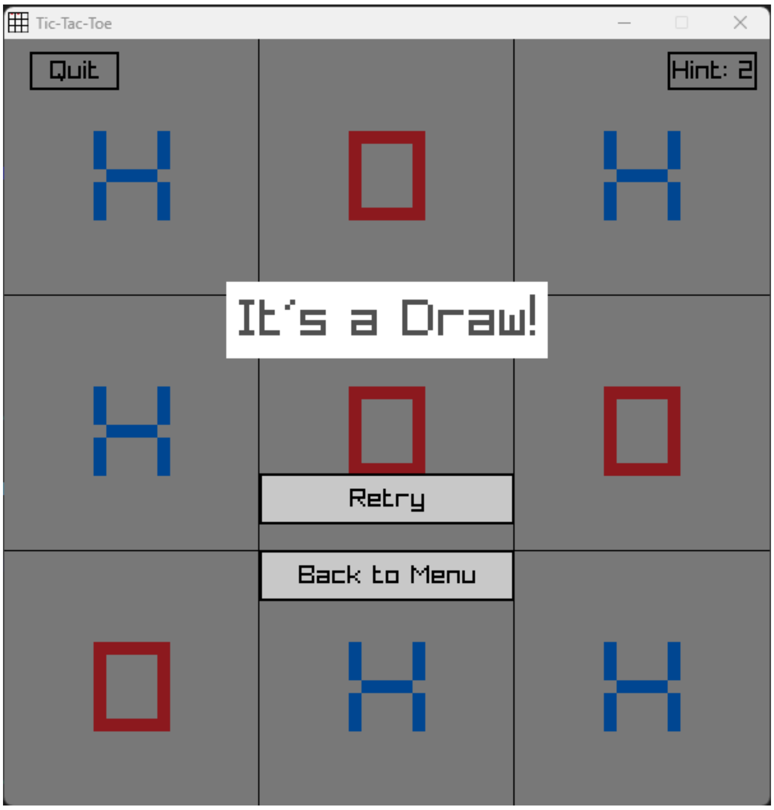


Fig.9: Draw Screen

6. Implementation of Machine Learning (Naive Bayes) (Easy Mode)

This section illustrates the methodology used to implement the Naive Bayes machine learning model into the easy mode.

6.1 Extracting and Processing of Dataset

The provided tic-tac-toe.data dataset would be used for training and testing.

1. Load the dataset into 2 different arrays using `load_data()` function, one containing the features of the Tic-Tac-Toe board layout, and the other containing the respective outcome [“positive” or “negative”].
   a. During the extraction of the outcome array, use the `outcome_index()` function to convert the string outcomes ("positive" or "negative") into the corresponding numerical label (POSITIVE = 0 or NEGATIVE = 1).
   b. This leads to an increase in memory efficiency and faster training speed, as numerical labels consume less memory and computation involving numbers is faster compared to string labels.
2. Split the dataset using `split_data()` function:
   a. Perform random shuffling on the dataset using Fisher-Yates algorithm.

      Iterates through the array backward, swapping the current element with a randomly chosen index from 0 to the current index, ensuring unbiased randomization.
   b. Split the dataset into 80:20 for the training and testing datasets respectively\

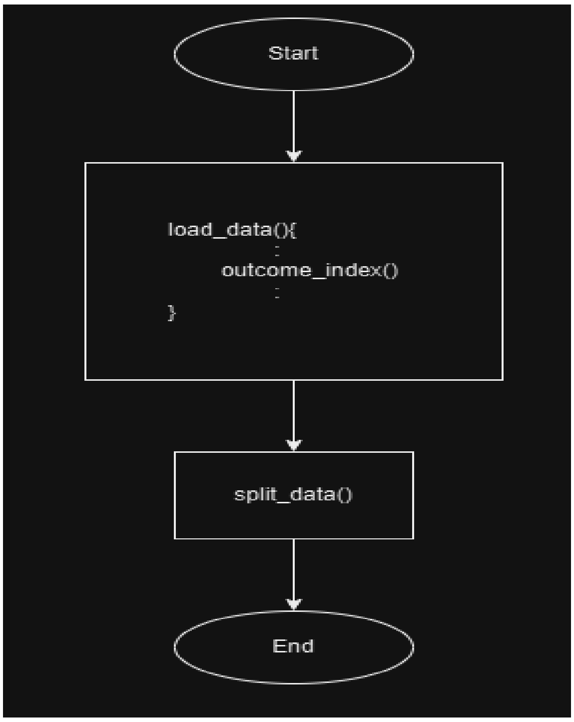


Fig.10: Functions calling order

```
train_boards array:    train_outcomes array:
bxxxbbooo              1
ooxxxoxbb              0
bxoxbobxo              1
xxboxboox              0
bxoxoboxb              1
xxobxoobx              0
oxxoooxxb              1
xbboxoxox              0
xxoxoboox              1
xxxxoobob              0
```

Fig.11: First 10 lines of training datasets

6.2 Logic of Naive Bayes

- It assumes the features are independent of one another given the outcome.
- It uses the Bayes' Theorem to calculate the *Posterior Probability* which is needed to predict the outcome/classes (positive/negative) given the features.

$$P(C|X) = \frac{P(C) \times P(X|C)}{P(X)}$$

a. *P(C|X) Posterior Probability formula*: Probability of the outcome C given the board configuration X
b. *P(C) Prior Probability*: Probability of outcome C (positive/negative) appearing in the dataset
c. *P(X|C) Conditional Probability(Likelihood)*: Probability of a feature X occurring in a given outcome C
d. *P(X) Marginal Probability(Evidence)*: Probability of feature X occurring in the dataset regardless of outcome C
e. However, the dividing of the evidence was not included as it remains constant for all outcome C given the feature X. Instead, comparison of the unnormalized *Posterior Probability* for each outcome was made and the one with the highest value was picked.

Thus, the functions below would be used for computing each of these probabilities.

6.3 Implementation of the Model

1. Train the model with Naive Bayes algorithm using `train_NBmodel()` function.

   1.1. Firstly, it counts the number of occurrences of each feature and each outcome in the dataset.

	x_counts array		o_counts array		b_counts array	
	pos	neg	pos	neg	pos	neg
Position 1	238	93	157	118	113	54
Position 2	176	121	176	84	138	62
Position 3	237	97	158	111	110	48
Position 4	179	124	188	87	138	61
Position 5	289	77	105	151	91	39
Position 6	173	126	186	80	147	66
Position 7	248	97	150	119	113	48
Position 8	182	125	182	77	132	64
Position 9	236	98	153	111	116	55

Fig.12: Arrays containing the number of occurrences of each feature in each position for each outcome

   1.2. Then calculate the *Prior Probability* of each class by taking the count of occurrences of each outcome divided by the size of the dataset

$$P(C) \; = \; \frac{Number\ of\ occurrences\ of\ outcome\ C}{Size\ of\ dataset}$$

1.3. Next calculate the *Conditional Probability* for each feature in each outcome by taking the count of the occurrences of the features in each position of the board in each outcome divided by the count of occurrences of the respective outcome using **Laplace Smoothing** to handle situations where a specific feature did not appear in the training dataset.

For eg, calculating the probability of feature ‘x’ in position 2 for a “positive” outcome:

$$P(X_2|C: "positive") \; = \; \frac{Count\ of\ 'x'\ in\ position\ 2\ in\ class\ "positive" + 1}{Number\ of\ occurrences\ of\ outcome\ "positive"}$$

1.4. Afterwards, save the model by saving its weights (*computed prior and conditional probabilities*) in the text file (“NBmodel_weights”) by calling the `save_NBmodel()` function.

```
Class Probabilities:
P(Positive): 0.655352
P(Negative): 0.344648

Position 1:
P(x | Positive): 0.461386
P(x | Negative): 0.355805
P(o | Positive): 0.300990
P(o | Negative): 0.449438
P(b | Positive): 0.237624
P(b | Negative): 0.194757

Position 2:
P(x | Positive): 0.364356
P(x | Negative): 0.464419
P(o | Positive): 0.368317
P(o | Negative): 0.303371
P(b | Positive): 0.267327
P(b | Negative): 0.232210

Position 3:
P(x | Positive): 0.485149
P(x | Negative): 0.370787
P(o | Positive): 0.287129
P(o | Negative): 0.441948
P(b | Positive): 0.227723
P(b | Negative): 0.187266

Position 4:
P(x | Positive): 0.350495
P(x | Negative): 0.449438
P(o | Positive): 0.372277
P(o | Negative): 0.299625
P(b | Positive): 0.277228
P(b | Negative): 0.250936

Position 5:
P(x | Positive): 0.576238
P(x | Negative): 0.269663
P(o | Positive): 0.233663
P(o | Negative): 0.573034
P(b | Positive): 0.190099
P(b | Negative): 0.157303

Position 6:
P(x | Positive): 0.350495
P(x | Negative): 0.460674
P(o | Positive): 0.376238
P(o | Negative): 0.299625
P(b | Positive): 0.273267
P(b | Negative): 0.239700

Position 7:
P(x | Positive): 0.483168
P(x | Negative): 0.378277
P(o | Positive): 0.297030
P(o | Negative): 0.449438
P(b | Positive): 0.219802
P(b | Negative): 0.172285

Position 8:
P(x | Positive): 0.364356
P(x | Negative): 0.456929
P(o | Positive): 0.362376
P(o | Negative): 0.307116
P(b | Positive): 0.273267
P(b | Negative): 0.235955

Position 9:
P(x | Positive): 0.467327
P(x | Negative): 0.393258
P(o | Positive): 0.310891
P(o | Negative): 0.426966
P(b | Positive): 0.221782
P(b | Negative): 0.179775
```

Fig.13: “NBmodel_weights.txt” text file

2. To calculate the *Posterior Probability* of a specified outcome based on a given board layout, call the `calculate_probability()` function which will perform the Bayes' Theorem formula but skipping the use of division with the *Marginal Probability(Evidence)*. Where it takes the computed *Prior Probability* of that specified outcome multiplied by every computed *Conditional Probability* of the respective features on the given board layout.

$$P(C|X) \;=\; P(C) \times P(X_1|C) \times P(X_1|C) \times \;...\; \times P(X_9|C)$$

For example, if the user set the outcome argument as "positive" and the given board layout is "bxobxobxo", the function would calculate how probable the given board layout would lead to a "positive" outcome.

$$P(''positive{:}\,|X) \;=\; P(''positive{:}) \times P('b'_1|''positive'') \times P('x'_2|''positive'') \times \;...\; \times P('o'_9|''positive'')$$

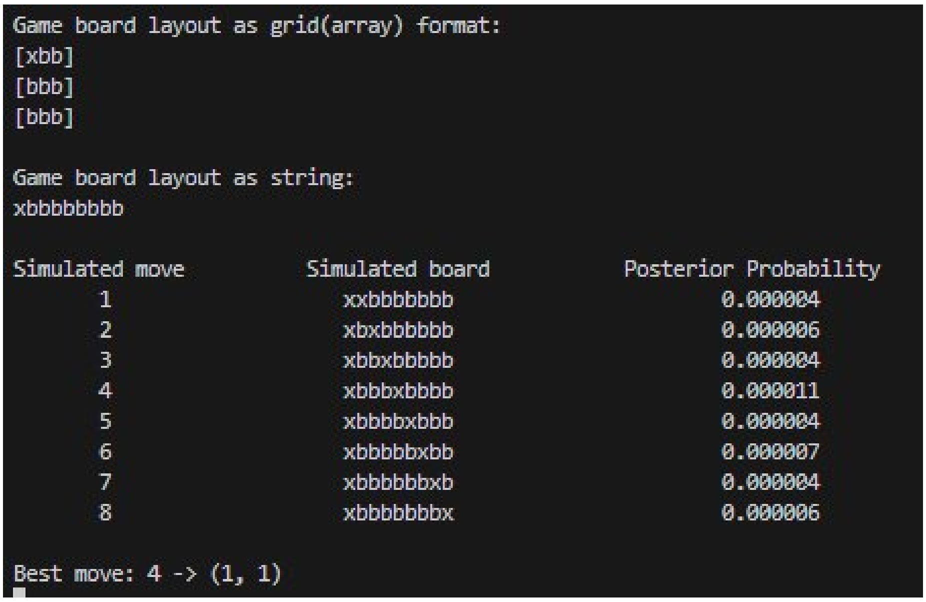
3. Hence, to predict whether a given board layout would lead to either a "positive" or "negative" outcome, the `predict_outcome()` function would calls the `calculate_probability()` function twice to calculate the *Posterior Probability* for both "positive" and "negative" outcome separately, and then returning the higher one.

4. Now that the model is trained with Naive Bayes algorithm, it could be used to play against players by calling the `predict_move()` function:

    4.1. Having it take in the current state of the game (the current board layout) when it is its turn

    4.2. Then simulate a move separately in every available(blank) spot and calculate the probability of winning with that respective move.

    4.3. Ultimately returning (playing) the move that results in the best probability.



```
Game board layout as grid(array) format:
[xbb]
[bbb]
[bbb]

Game board layout as string:
xbbbbbbbb

Simulated move        Simulated board         Posterior Probability
      1                  xxbbbbbbb                   0.000004
      2                  xbxbbbbbb                   0.000006
      3                  xbbxbbbbb                   0.000004
      4                  xbbbxbbbb                   0.000011
      5                  xbbbbxbbb                   0.000004
      6                  xbbbbbxbb                   0.000007
      7                  xbbbbbbxb                   0.000004
      8                  xbbbbbbbx                   0.000006

Best move: 4 -> (1, 1)
```

Fig.14: Logic Process of predict_move() function

5. The `predict_move()` function does this by looping through every *Posterior Probability* of the "positive" outcome (probability of winning) using the `calculate_probability()` function on each separate simulated moves. Then returning the move with the highest probability.

  5.1. The `predict_move()` function simulates the game board as a string. For eg **"obxxoxbbo"**, with the leftmost character being position 1 and rightmost being position 9. After predicting the best move, it will arrive at an integer (0-8) representing one of the positions on the board.

  5.2. However, the GUI reads and writes the board as an array. Such as,

[‘o’ , ‘b’ , ‘x’]
[‘x’ , ‘o’ , ‘x’]
[‘b’ , ‘b’ , ‘o’]

  5.3. Thus, to access position 7 on the board, the two values (2,0), representing the indexes of the row and column of the array respectively.

  5.4. Hence, a loop was created to read in the board from the GUI and convert it into a string, so then the function can resume simulating the moves and get the best move.

For eg:

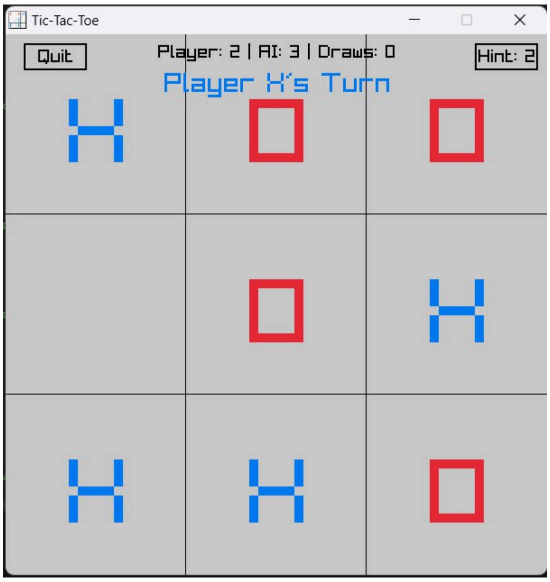


Fig.15: Current board layout (GUI)

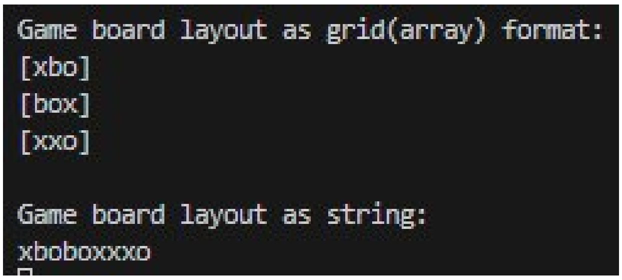


Fig.16: Current board layout in the backend

Note: Even though in position 2 there is a 'b' in the grid format but the GUI shows a move has been made there, it is because the loop only occurs when it is the AI's turn to play but the move has not been made yet. The GUI shows the board after it made its move which is position 2.

5.5. Then call the `divide()` function to convert the best move integer into the row and column indexes of the board.

```
Best move: 8 -> (2, 2)
```

Fig.17: Conversion of integer best_move to row and column indexes of grid

5.6. Return the indexes to the backend to play the move.

5.7. Therefore, successfully implementing the trained Naive Bayes model into the game for players to play against.

6.4 Evaluation of Model

Run **`test_NBmodel()`** function to evaluate the performance of the model. The function will take into account the model's:

a. Count of the True Positive (TP), False Positive (FP), True Negative (TN), False Negative (FN) predicted outcomes made based on any given board layout in both the training and testing dataset.

b. Count of wrong predictions made. [error_count]

c. Using these values we can compute the Accuracy and Probability of error of the trained model.

d. Then save the results in a text file ("NBmodel_confusion_matrix.txt") for plotting of the confusion matrix.

```
Training Dataset:
  Accuracy: 71.54% (548/766)
  Error: 28.46% (218/766)
  Confusion Matrix:
    True Positive: 436
    False Positive: 153
    True Negative: 112
    False Negative: 65


Testing Dataset:
  Accuracy: 70.83% (136/192)
  Error: 29.17% (56/192)
  Confusion Matrix:
    True Positive: 110
    False Positive: 41
    True Negative: 26
    False Negative: 15
```

Fig.18: "NBmodel_confusion_matrix.txt" text file

6.5 Plots and Results (Naive Bayes)

To generate the Confusion Matrix of the Naive Bayes model for evaluation.

1. Ensure Python is installed in your Visual Studio Code(VSC),
2. In the VSC terminal, run these commands to install the necessary libraries to plot the confusion matrix
   a. `pip3 install matplotlib`
   b. `pip3 install seaborn`
   c. `pip3 install numpy`
3. Run the **plot_confusion_matrix.py** file. It would plot the matrix based on the values it reads from the "NBmodel_confusion_matrix.txt" text file. In addition, the matrix plot would be saved as a PNG image under "NBmodel_confusion_matrix.png".

This is the confusion matrix for the predictions made on the training and testing dataset by the Naive Bayes model.

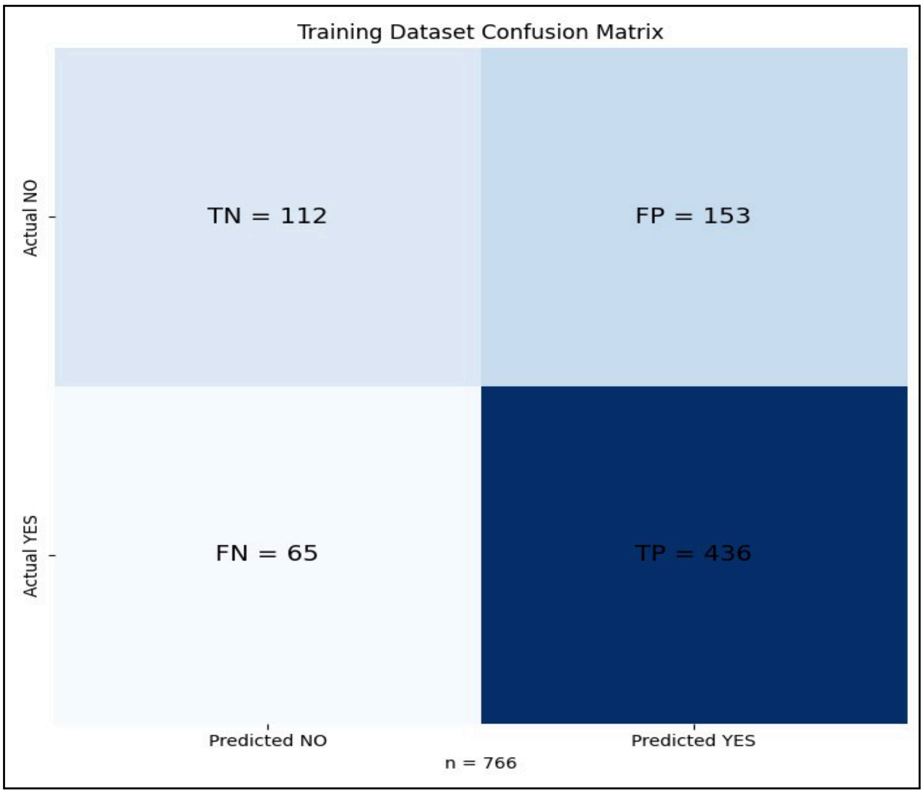


Fig 19: Confusion Matrix of predictions made on Training Dataset

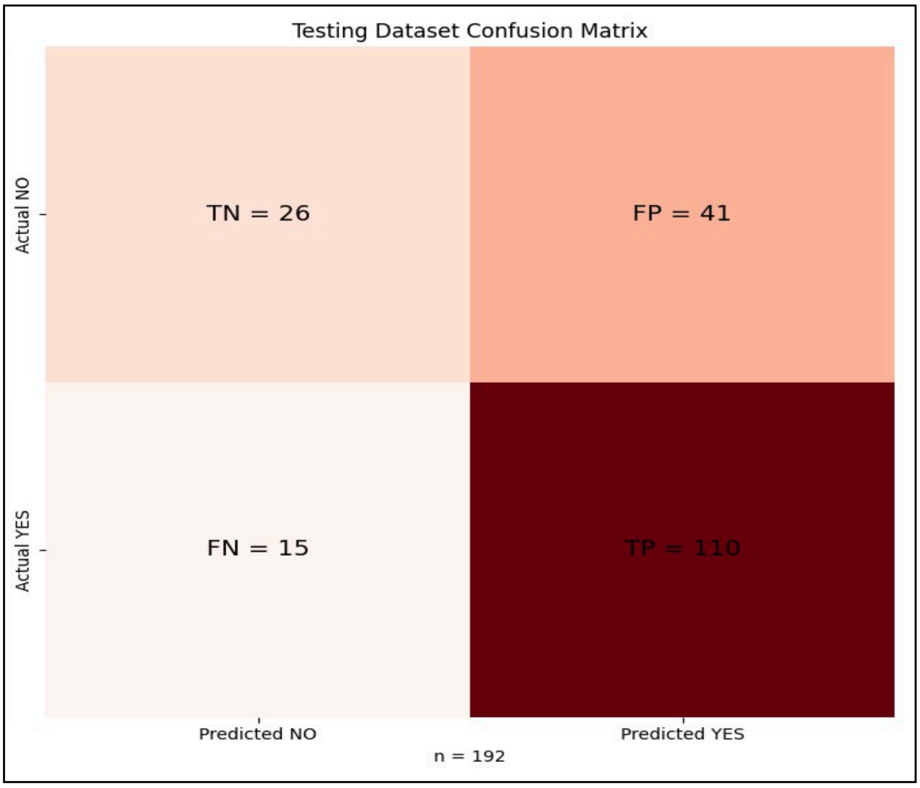


Fig.20: Confusion Matrix of predictions made on Testing Dataset

Definitions of classes:

- True Positive: Number of times the model predicted **correctly** that the outcome of the Tic-Tac-Toe board layout is **positive**.
- False Positive: Number of times the model predicted **wrongly** that the outcome of the Tic-Tac-Toe board layout is **positive**.
- True Negative: Number of times the model predicted **correctly** that the outcome of the Tic-Tac-Toe board layout is **negative**.
- False Negative: Number of times the model predicted **wrongly** that the outcome of the Tic-Tac-Toe board layout is **negative**.

Upon analysing the plot, we can see that the model made:

- 436 + 112 = 548 correct predictions out of 766 outcomes in the training dataset.
- 548/766 X 100 ≈70%
- 110 + 26 = 136 correct predictions out of 192 outcomes in the testing dataset.
- 136/192 X 100 ≈70%

This shows that the model has around 70% accuracy in predicting the correct outcome with any given board layout. This means that when the model is predicting the next best move while playing against players, it would have a 70% chance of choosing the next correct best move, which is sufficient for an easy mode.

7. Implementation of Machine Learning (Decision Tree) (Easy Mode)

This section illustrates the methodology used to implement the Decision Tree machine learning model into the easy mode.

7.1 Extracting and Processing of Dataset

- We used the provided tic-tac-toe.data dataset for training and testing.

1. **Load the dataset into an array using load_dataset() function:**
   - The dataset is loaded into a 2D array, where each row represents a board layout and the last column represents the respective outcome ("positive" or "negative").
   - During loading, the **`load_dataset()`** function converts the string outcomes ("positive" or "negative") into corresponding numerical labels.
   - **Assigned Values**:
     - `#define DT_POSITIVE 1      // Label for outcome "positive = WIN "`
     - `#define DT_NEGATIVE 0      // Label for outcome "negative = LOSE OR DRAW"`
   - **Assign Labels:**
     - (`“positive” = DT_POSITIVE`, `“negative” = DT_NEGATIVE`).
   - **Labels Mapping**:
     - `"positive\n"` → `DT_POSITIVE` (`1`).
     - `"negative\n"` → `DT_NEGATIVE` (`0`).
   - The dataset consists of different board configurations, with each board represented by numerical values for each cell:
     - `1` for an `x`
     - `2` for an `o`
     - `0` for a blank space
   - This representation allows the model to make decisions based on the current state of the board.
   - We store the processed data into an array of `DataRow` structures.

```
Input Line: x,o,x,o,x,b,x,o,b,positive
Output:
features[] = {1, 2, 1, 2, 1, 0, 1, 2, 0}
label = DT_POSITIVE
```

Fig.21: Processed Data Stored In Array

   - This conversion ensures memory efficiency and faster training speed which is crucial for numerical operations as they are much faster compared to string operations which leads to better performance during model training and evaluation.

```
Encoded Training Dataset
[1, 1, 0, 2, 2, 0, 1, 0, 0] DT_POSITIVE
[2, 1, 1, 2, 2, 0, 1, 1, 2] DT_NEGATIVE
[1, 0, 1, 2, 1, 2, 0, 2, 0] DT_POSITIVE
[0, 1, 2, 1, 1, 0, 2, 2, 1] DT_NEGATIVE
[1, 2, 0, 1, 2, 2, 0, 1, 0] DT_POSITIVE
[2, 1, 1, 2, 0, 1, 1, 2, 2] DT_NEGATIVE
[1, 0, 1, 2, 2, 1, 0, 0, 2] DT_POSITIVE
[0, 2, 1, 0, 1, 2, 1, 1, 2] DT_NEGATIVE
[1, 2, 2, 1, 1, 0, 0, 2, 1] DT_POSITIVE
[2, 1, 1, 2, 0, 0, 1, 1, 2] DT_NEGATIVE
```

Fig.22: Encoded Training Dataset

- Each number corresponds to the content of a cell in the Tic-Tac-Toe grid

  `#define NUM_FEATURES 9 (Feature_1 to Feature_9)`

```
Board Position Mapping:
1 | 2 | 3
4 | 5 | 6
7 | 8 | 9
```

Fig.23: Board Position Mapping

- Original Board Representations Input: Encoded Output:

```
Board 1:
x | x |
o | o |
x |   |
Label: DT_POSITIVE
```

Fig.24: Representation Input

```
[1, 1, 0, 2, 2, 0, 1, 0, 0] DT_POSITIVE
```

Fig.25: Encoded Output

2. **Shuffle the dataset using `shuffle_dataset()` functions**
   - Performed a Fisher-Yates shuffle in order to randomize the dataset in an unbiased manner. One of the efficient ways to generate a random permutation of the dataset is by using the Fisher and Yates algorithm so as to make sure that this model is not biased based on the ordering of data. This random shuffling stops over-fitting-the data becomes well distributed, preventing the model from memorizing sequences.

3. **Split the dataset using `decision_tree_split_dataset()` functions**
   - Split the data into 80:20 for the ratio of the training set and test set, respectively, for two subsets in order to evaluate and train. The tree will be grown based on the training set, while the performance and generalization are evaluated on a testing set. The training set offers a varied set of examples that can help the model learn from the decision boundaries. On the other hand, the test set measures how well the model generalizes on unseen data. This ratio may be modified based on the size of the dataset to make certain that decent amounts of data are available for both training and evaluation.

   - **Functions Calling Order**:
     Start
     |
     |----> load_dataset()
     |
     |----> shuffle_dataset()
     |
     |----> decision_tree_split_dataset()
     |
     |----> build_tree()
     |
     |----> evaluate_with_randomness()
     End

7.2 Decision Tree Construction and Training

The Decision Tree gets built using the `build_tree()` function where the training data set is used to recursively construct a binary decision tree to identify the best splits based on features and thresholds which minimizes impurity based on the Gini Index. The tree is constructed until either a stopping condition (pure node or maximum depth) is met, or it perfectly classifies the training data set. Gini Index: Gini Index, an important measure that is used in the Decision Tree Algorithms to give scores to the splits made by partitioning the dataset. Now, the Gini Index can help decision tree trained models to pick the best moves or features so they can form splits that will help Tic-Tac-Toe game strategies make accurate predictions.

Gini Impurity Formula:

$$Gini = 1 - \sum_{i=1}^{c} (pi)^2$$

Where:

- C: Number of classes (e.g., positive and negative outcomes in your case).
- p_i: Proportion of instances belonging to class i in the subset.

For our implementation:

- $C = 2$ (Positive and Negative).
- $p(positive)$ = Proportion of positive labels in the subset.
- $p(negative) = 1 - p(positive)$: Proportion of negative labels in the subset.

The Gini Index in our implementation is calculated separately for two subsets (left and right) and combined into a **weighted average**.

7.3 Splitting the Dataset

- The dataset is divided into two subsets (`left` and `right`) using the `decision_tree_split_data` function based on the **selected feature** and **threshold**.
    i. Feature (`feature_index`)
        a. Each feature represents the **state of a board position** (one of the 9 cells in the 3x3 grid).
            - For example:
                - `feature_index = 1`: The top-left cell.
                - `feature_index = 5`: The center cell.
                - `feature_index = 9`: The bottom-right cell.
        b. Identifies which board position (cell) is most informative for splitting the dataset. For example, splitting based on the **center cell** provides the most information about outcomes.
    ii. Threshold (`threshold`): Defines the rule/evaluation for splitting the dataset based on the selected feature.
        a. `threshold = 0`: Empty cell.
        `threshold = 1`: Cell occupied by 'X'.
        `threshold = 2`: Cell occupied by 'O'.
        b. For example:
        When `threshold = 1` means the dataset is split into:
            - **Left Subset**: Samples where `feature_value ≤ threshold`.
            For example, board states where the center cell is empty or has 'X'.
            - **Right Subset**: Samples where `feature_value > threshold`.
            For example, board states where the center cell has 'O'.

- To achieve more accurate predictions by minimizing impurity and creating the purest possible subsets, the split (feature and threshold) is determined using the **Gini Index.**

Calculation of Gini Index

The `calculate_gini_index` function computes the Gini impurity for a dataset, which is divided into left and right given a feature and its threshold. It also computes the probability of positive labels (`prob_left` and `prob_right`) in each of those subsets. It calculates the quality of the split by measuring the degree to which the class labels are well-separated in the resulting branches, returning the Weighted Averaged Gini Index after splitting the dataset.

- **Check for Empty Branches**: If either of the subsets is empty, it returns a Gini value of 1.0, the worst impurity value possible. This penalizes choosing splits that lead to empty branches.
- **Weighted Averaged Gini Index Range**: [0.0, 1.0].
  - A value of **0.0** indicates a perfectly pure split (no impurity).
  - A value of **1.0** indicates maximum impurity or an invalid split.
- **Calculate Gini for Left and Right Branches**:
  - Calculate the probability of positive labels in both left and right branches for each subset.
  - Use the formula below to calculate Gini impurity for each subset where is the probability of a positive label.
  - Formula for $Gini\ for\ Single\ Subset$

$$Gini = 1 - \sum_{i=1}^{c} (pi)^{2}$$

  - Formula for $Probabulity\ of\ p(positive)$

$$P(positive) = \frac{Number\ of\ Positive\ Labels\ in\ Subset}{Total\ Number\ of\ Samples\ in\ Subset}$$

- **Calculating Gini Impurity for Each Branch which is computed using**
  - Formula for $Gini\ Impurity$:
    - $Gine = 1 - P(postive)^{2} - P(negative)^{2}$

    Where $P\ (negative) = 1 - P(positive)$

- Finally, the function returns the **weighted average of the Gini indices** for both branches to determine the quality of the split.
  - Formula for $Weighted\ Gini\ Index$

$$Gini\ (weighted) = \frac{(Gini(left)\ x\ Size(left)) + (Gini(right)\ x\ Size(right))}{Total\ Size}$$

- To avoid overfitting, the tree grows to a **constrained maximum depth** where no splits are made when either a maximum depth is reached or the data at a node is homogeneous.

In short, the selected **feature** and **threshold** in our decision tree algorithms determine the best criteria for splitting the dataset at a particular node. These criteria are determined by evaluating all possible splits and selecting the one that minimizes impurity, for example, by using Gini Index.

7.4 Decision Tree Construction and Training

```
for (int feature_index = 0; feature_index < NUM_FEATURES; feature_index++) {
        for (int threshold = 0; threshold <= 2; threshold++) {
            // Calculate the Gini impurity for the current split
            float gini = calculate_gini_index(dataset, size, feature_index, threshold);
            if (gini < best_gini) {
                // Update the best Gini impurity, feature, and threshold if this split is better
                best_gini = gini;
                best_feature = feature_index;
                best_threshold = threshold;
            }
        }
    }
```

The process iterates over all the 9 features (`NUM_FEATURES` = 9) which relate to the cells on a Tic-Tac-Toe board. It evaluates three possible thresholds for each of the features: (`0`, `1` and `2`). Next, for each combination of features and thresholds, it calculates the Gini Index of the split that would be created by splitting the dataset based on the feature and threshold using `calculate_gini_index`. If the current split's Gini Index is lower than the previously recorded best, it updates:

- `best_feature` → Feature responsible for the split
- `best_threshold` → Value used to split the feature
- `best_gini` → Gini Index of the split

The splitting process will be illustrated with the following example. For illustration purposes, 5 boards will be used for the splitting process.

Features (each row represents a Tic-Tac-Toe board encoded as integers):

```
[1, 2, 0, 1, 0, 2, 1, 0, 0]  // Board 1
[0, 2, 1, 1, 2, 0, 1, 1, 2]  // Board 2
[1, 0, 0, 2, 1, 0, 0, 2, 1]  // Board 3
[2, 2, 1, 0, 0, 1, 1, 0, 0]  // Board 4
[0, 1, 0, 2, 1, 2, 1, 2, 0]  // Board 5
```

Fig.26: Features used for splitting

Labels (outcomes of each board):

```
[DT_POSITIVE, DT_NEGATIVE, DT_POSITIVE, DT_NEGATIVE, DT_NEGATIVE]
```

Fig.27: Outcome Labels

Select and evaluate all features (`feature_index = 0 to 8`) to find the best split. Firstly, splitting will be based on `feature_index = 4` (center cell). Next, the **Gini Index** for each threshold will be calculated for the chosen `feature_index = 4`.

A. **Case 1**: When Threshold = 0, the split logic works by dividing boards where the center cell is empty (`feature[4] ≤ 0`) and occupied (`feature[4] > 0`).

Splitted Dataset:

```
Features:
[1, 2, 0, 1, 0, 2, 1, 0, 0]  // Board 1
[2, 2, 1, 0, 0, 1, 1, 0, 0]  // Board 4
[0, 1, 0, 2, 0, 2, 1, 2, 0]  // Board 5
Labels:
[DT_POSITIVE, DT_NEGATIVE, DT_NEGATIVE]
```

Fig.28: Left Subset

```
Features:
[0, 2, 1, 1, 2, 0, 1, 1, 2]  // Board 2
[1, 0, 0, 2, 1, 0, 0, 2, 1]  // Board 3
Labels:
[DT_NEGATIVE, DT_POSITIVE]
```

Fig.29: Right Subset

Next, we calculate the Gini Index for both subset.

$Left\ Subset$:

$P(positive) = 1/3,\ P(negative) = 2/3$

$Gini(left) = 1 - (P(positive)^2 + P(negative)^2)$

$= 1 - (1/3^2 + 2/3^2)$

$= 0.444$

$Right\ Subset$:

$P(positive) = 1/2,\ P(negative) = 1/2$

$Gini(right) = 1 - (P(positive)^2 + P(negative)^2)$

$= 1 - (1/2^2 + 1/2^2)$

$= 0.5$

$Gini\ Index$:

$Gini(split) = ((Gini(left)x3) + (Gini(right)x2)/5)$

$= ((0.444x3) + (0.5x2)/5)$

$= 0.467$

```
Left Subset:
Gini(left) = 0.444
Right Subset:
Gini(right) = 0.500
Overall Gini Index (Split):
Gini(split) = 0.467
```

Fig.30: Output of Gini Index

B. **Case 2:** When Threshold = 1, the split logic works by dividing boards where the center cell is empty or 'X' (`feature[4] ≤ 1`) and occupied by 'O'

(`feature[4] > 1`).

Features (each row represents a Tic-Tac-Toe board encoded as integers):

```
[1, 2, 0, 1, 0, 2, 1, 0, 0]  // Board 1
[0, 2, 1, 1, 1, 0, 1, 1, 2]  // Board 2
[1, 0, 0, 2, 1, 0, 0, 2, 1]  // Board 3
[2, 2, 1, 0, 2, 1, 1, 0, 0]  // Board 4
[0, 1, 0, 2, 1, 2, 1, 2, 0]  // Board 5
```

Fig.31: Features

Labels (outcomes of each board):

```
[DT_POSITIVE, DT_NEGATIVE, DT_POSITIVE, DT_NEGATIVE, DT_NEGATIVE]
```

Fig.32: Labels (Outcome of Each Board)

Left Subset (Center ≤ 1):Left

```
Features:
[1, 2, 0, 1, 0, 2, 1, 0, 0]  // Board 1
[0, 2, 1, 1, 1, 0, 1, 1, 2]  // Board 2
[1, 0, 0, 2, 1, 0, 0, 2, 1]  // Board 3
[0, 1, 0, 2, 1, 2, 1, 2, 0]  // Board 5
Labels:
[DT_POSITIVE, DT_NEGATIVE, DT_POSITIVE, DT_NEGATIVE]
```

Fig.33: Left Subset

Right Subset (Center > 1):

```
Features:
[2, 2, 1, 0, 2, 1, 1, 0, 0]  // Board 4
Labels:
[DT_NEGATIVE]
```

Fig.34: Right Subset

Next, the Gini Index is calculated for both subset.

$$
\begin{aligned}
&Left\ Subset: \\
&P(positive) = 1/2,\ P(negative) = 1/2 \\
&Gini(left) = 1 - (P(positive)^2 + P(negative)^2) \\
&\qquad = 1 - (1/2^2 + 1/2^2) \\
&\qquad = 0.5 \\
&Right\ Subset: \\
&P(positive) = 0,\ P(negative) = 1 \\
&Gini(right) = 1 - (P(positive)^2 + P(negative)^2) \\
&\qquad = 1 - (0^2 + 1^2) \\
&\qquad = 0 \\
&Gini\ Index: \\
&Gini(split) = ((Gini(left)x4) + (Gini(right)x1)/5) \\
&\qquad = ((0.5x4) + (0x1)/5) \\
&\qquad = 0.4
\end{aligned}
$$

```
Left Subset:
Gini(left) = 0.500
Right Subset:
Gini(right) = 0
Overall Gini Index (Split):
Gini(split) = 0.400
```

Fig.35: Output of Gini Index

C. **Case 3:** When Threshold = 1, the split logic works by dividing boards where the center cell is empty, 'X', or 'O' (`feature[4] ≤ 2`).

Features (each row represents a Tic-Tac-Toe board encoded as integers):

```
[1, 2, 0, 1, 0, 2, 1, 0, 0]  // Board 1
[0, 2, 1, 1, 2, 0, 1, 1, 2]  // Board 2
[1, 0, 0, 2, 1, 0, 0, 2, 1]  // Board 3
[2, 2, 1, 0, 0, 1, 1, 0, 0]  // Board 4
[0, 1, 0, 2, 1, 2, 1, 2, 0]  // Board 5
```

Fig.36: Selected Features

Labels (outcomes of each board):

```
[DT_POSITIVE, DT_NEGATIVE, DT_POSITIVE, DT_NEGATIVE, DT_NEGATIVE]
```

Fig.37: Selected Labels

Left Subset (Center ≤ 2):Left

Right Subset (Center > 2): No data

```
Features:
[1, 2, 0, 1, 0, 2, 1, 0, 0]  // Board 1
[0, 2, 1, 1, 2, 0, 1, 1, 2]  // Board 2
[1, 0, 0, 2, 1, 0, 0, 2, 1]  // Board 3
[2, 2, 1, 0, 0, 1, 1, 0, 0]  // Board 4
[0, 1, 0, 2, 1, 2, 1, 2, 0]  // Board 5
Labels:
[DT_POSITIVE, DT_NEGATIVE, DT_POSITIVE, DT_NEGATIVE, DT_NEGATIVE]
```

Fig.38: Left Subset

Next, we calculate the Gini Index for both subset.

$Left\ Subset$:

$P(positive) = 2/5,\ P(negative) = 3/5$

$Gini(left) = 1 - (P(positive)^2 + P(negative)^2)$

$= 1 - (2/5^2 + 3/5^2)$

$= 0.48$

$Right\ Subset$:

$P(positive) = 0,\ P(negative) = 0$

$$Gini(right) = 1 - (P(positive)^2 + P(negative)^2)$$
$$= 1 - (0^2 + 0^2)$$
$$= 1\ (empty\ subset)$$

$Gini\ Index$:

$$Gini(split) = ((Gini(left)x5) + (Gini(right)x0)/5)$$
$$= ((0.48x5) + (1x0)/5)$$
$$= 0.48$$

```
Left Subset:
Gini(left) = 0.480
Right Subset:
Gini(right) = 1
Overall Gini Index (Split):
Gini(split) = 0.480
```

Fig.39: Output of Gini Index

(**Notes: This shows the split is valid but doesn't further divide the dataset effectively. Thus, threshold = 2 often does not improve the tree's structure meaningfully.)*

II. Using the calculations above, we select the best split based on the lowest weighted Gini Index. A lower Gini Index indicates better separation between positive and negative outcomes.

Threshold = 0 → Gini = 0.467
Threshold = 1 → Gini = 0.4 (Best split)
Threshold = 2 → Gini = 0.48 (Invalid split as there is no data at right subset)

We concluded that the best split for feature_index = 4 occurs at Threshold = 1 because it has the lowest weighted Gini Index (0.4) resulting in the most effective splitting of the dataset.

The importance of the splitting can conclude in 3 ways:

a. Feature Importance: The chosen feature reveals which board position has the most predictive power regarding the outcome. For example, the center cell (feature_index = 4) is often critical in Tic-Tac-Toe, contributing significantly to the board's outcome.
b. Thresholds Define Splitting Rules: The threshold indicates how the feature is utilized such as:
   - `threshold = 1` → Splits based on whether a cell contains 'x'.
c. Improves predictive power by selecting features and thresholds that reduce impurity, the tree generalizes better to unseen data.

The splitting logic ensures that the decision tree prioritizes the most significant features and thresholds, minimizing dataset impurity at each step. This recursive process continues until the following conditions are satisfied:

- The maximum depth is reached
- All rows in the dataset belong to the same class (pure node)
- A leaf node is created
    - Allocates memory for a new node and stores the best feature and threshold.
    - Recursively calls `build_tree` for the left and right subsets.

7.5 Splitting Nodes and Tree-Building Process `(build_tree)`

During the construction of the decision tree, the dataset is split at each internal node using Gini impurity and leaf nodes based on the feature and threshold that result in the lowest Gini impurity and are labeled based on the majorities of the predictions/outcomes.

I. Logic of the Splitting Nodes

- **Recursively Build Subtrees: Internal Nodes**

```
node->left = build_tree(left, left_size, depth + 1);
node->right = build_tree(right, right_size, depth + 1);
```

The function recursively calls itself to construct the left and right subtrees using their respective subsets of the dataset. With each recursive call, the depth is incremented by one (`depth + 1`). These nodes represent decision points in the tree, where the dataset is split into two subsets. Ideally, each subset becomes more homogeneous as Gini impurity is used as the metric to select the feature and threshold that achieve the best split—maximizing the reduction in impurity.

    - If the node is pure or a maximum depth is reached, a leaf node is created. Leaf nodes are assigned a label based on the majority class.
    - If further splitting is possible, the dataset is split into two subsets based on the **best feature** and **threshold**, and **left** and **right child nodes** are created recursively.

- **Leaf Nodes**

```
int positives = 0, negatives = 0;
for (int i = 0; i < size; i++) {
    if (dataset[i].label == DT_POSITIVE)
        positives++;
    else
        negatives++;
}
```

The function begins by counting the number of positive (`DT_POSITIVE`) and negative (`DT_NEGATIVE`) labels in the current dataset. If a node meets the stopping criteria—either reaching the maximum depth or achieving purity (all points belong to one class)—it is converted into a leaf node. The leaf node is then labeled based on the majority class in the dataset subset at that node.

```
    if (depth >= MAX_DEPTH || positives == 0 || negatives == 0) {
        DecisionTreeNode *leaf = (DecisionTreeNode
*)malloc(sizeof(DecisionTreeNode));
        leaf->is_leaf = 1;
        leaf->prediction = (positives > negatives) ? DT_POSITIVE : DT_NEGATIVE;
        leaf->left = leaf->right = NULL;
        return leaf;
    }
```

The dataset is split recursively at each internal node until a stopping condition is met:

- **i.** Maximum tree depth is reached (`depth >= MAX_DEPTH`) to prevent overfitting.
- **ii.** The node becomes "pure," meaning all data points belong to the same class (`positives == 0 || negatives == 0`), so further splits are unnecessary.
- **iii.** The number of samples at the node falls below the minimum threshold, preventing overfitting to small datasets.

II. **Labeling Leaf Nodes Based on Majority of Predictions**

When a stopping condition is satisfied, a leaf node is generated to represent the final prediction for that branch:

- The label of the leaf node is determined based on the majority class of the dataset that reaches that leaf.
- If the number of positive examples (`positives`) is greater than the number of negative examples (`negatives`), the leaf node is labeled as positive (`DT_POSITIVE`).
- Otherwise, the node is labeled as negative (`DT_NEGATIVE`).

The idea is that each leaf node contains data points that are as homogeneous as possible which improves the accuracy of predictions made using the decision tree. At the end we did **Return Node** to return the newly created decision tree node either a leaf or an internal node to play the move.

7.6 Saving Position Probabilities

After building the decision tree, we then call the function `calculate_position_probabilities()` that computes the weights for every board position using the TicTacToe dataset. It computes the probability of each of the possible symbols (`x`, `o`, `empty`) at each position given the class label (`positive` or `negative`). We write these position-wise probabilities (weights) to a text file, `DTweights.txt`, which could later be used for further analysis and model evaluation.

- **Position Probabilities Calculation**:
  For each board position, the occurrence of symbols (x, o, blank) is calculated for each class (positive or negative). The probabilities are calculated as:
  Where:
  - Conditional probability of symbol given class
  - Total number of possible symbols (x, o, blank)

The function starts by setting up essential counters and data structures.

- The variables positive_count and negative_count are initialized to zero, serving to count positive and negative outcomes respectively. Additionally, a 3D array position_count[NUM_FEATURES][3][2] is initialized to zero to track symbol counts. The dimensions are:
  - NUM_FEATURES = 9 for a standard Tic-Tac-Toe board represents the total number of board positions.
  - 3 represents the possible values of each position (x, o, or empty).
  - 2 represents the game outcome classes (positive or negative).
- The function proceeds to iterate through the dataset to collect counts of each symbol for every board position. For each entry in the dataset, it increases the relevant class counter (positive_count or negative_count) depending on whether the label is DT_POSITIVE or DT_NEGATIVE. Subsequently, for each feature (board position), it updates the corresponding counts in the position_count array.
  - If the feature value is 1 (representing x), it increments the count for x in that position.
  - If the feature value is 2 (representing o), it increments the count for o.
  - If the feature value is 0 (representing an empty position), it increments the count for an empty cell.

After gathering the counts, then we opens weights file (DTweights.txt) in **write mode** ("w") to save the computed probabilities:

- **Class Probabilities** are calculated as the proportion of positive and negative examples in the dataset. Specifically:

$$P(Positive) = \frac{Positive\ Count}{Dataset\ Size} \qquad P(Negative) = \frac{Negative\ Count}{Dataset\ Size}$$

Next, we calculate the **position-wise conditional probabilities** of each symbol (x, o, or empty) for both positive and negative outcomes:

- For each board position (i from 0 to 8), the function iterates through the possible symbols (x, o, and empty).
- For each symbol (j), the function calculates the **conditional probability** given the class:

$$P(Symbol\ j \mid Positive) = \frac{Position\ Count[i][j][DT\ Positive]}{Positive\ Count}$$

$$P(\,Symbol\ j \mid Negative\,) = \frac{Position\ Count[i][j]\left[DT\ NEGATIVE\right]}{Negative\ Count}$$

Example Walkthrough

Consider three data points from the dataset representing Tic-Tac-Toe board states.

```
Data Point 1
Features:
[1, 2, 0, 1, 0, 2, 1, 0, 2]
[x, o, b, x, b, o, x, b, o]
Label: DT_POSITIVE

Data Point 2
Features:
[2, 0, 1, 0, 1, 2, 2, 1, 0]
[o, b, x, b, x, o, o, x, b]
Label: DT_NEGATIVE

Data Point 3
Features:
[1, 1, 2, 2, 0, 1, 0, 2, 1]
[x, o, o, o, b, x, b, o, x]
Label: DT_POSITIVE
```

Fig.40: Datapoints

As the iteration progresses, the `positive_count` is updated to 2, and the `negative_count` to 1. The `position_count` array is adjusted by incrementing the respective elements for each board position and symbol. For instance, in position `1` , if the symbol "x" appears 3 times in positive games, and the total number of positive games is 2, the corresponding probability is calculated.

$$P(\,x\ at\ Position\ 1 \mid Positive\,) \ = \frac{2}{3} = 1.5$$

These probabilities reflect how frequently a specific symbol ("x," "o," or empty) occurs in a given board position, depending on the game's outcome. The computed results are saved in the `DTweights.txt` file in a tabular format, making it easier to interpret. This format provides a clear summary of the dataset's class distribution, offering valuable insights into patterns and tendencies as illustrated in the accompanying image.

```
Class Probabilities:
  Positive: P(Positive) = 0.6534
  Negative: P(Negative) = 0.3466
--------------------------------------------
Position 1:
  Symbol | P(Symbol | Positive) | P(Symbol | Negative)
  -------|----------------------|----------------------
  x      | 0.4712               | 0.3705
  o      | 0.3019               | 0.4398
  b      | 0.2268               | 0.1898
--------------------------------------------
Position 2:
  Symbol | P(Symbol | Positive) | P(Symbol | Negative)
  -------|----------------------|----------------------
  x      | 0.3594               | 0.4608
  o      | 0.3658               | 0.3042
  b      | 0.2748               | 0.2349
--------------------------------------------
Position 3:
  Symbol | P(Symbol | Positive) | P(Symbol | Negative)
  -------|----------------------|----------------------
  x      | 0.4712               | 0.3705
  o      | 0.3019               | 0.4398
  b      | 0.2268               | 0.1898
--------------------------------------------
Position 4:
  Symbol | P(Symbol | Positive) | P(Symbol | Negative)
  -------|----------------------|----------------------
  x      | 0.3594               | 0.4608
  o      | 0.3658               | 0.3042
  b      | 0.2748               | 0.2349
--------------------------------------------
Position 5:
  Symbol | P(Symbol | Positive) | P(Symbol | Negative)
  -------|----------------------|----------------------
  x      | 0.5847               | 0.2771
  o      | 0.2364               | 0.5783
  b      | 0.1789               | 0.1446
--------------------------------------------
```

Fig.41:DTweights.txt

The outputs from the `calculate_position_probabilities()` function play a crucial role in training our decision tree models. These conditional probabilities help the model make smarter decisions by providing deeper insights into the likelihood of different outcomes. By leveraging these probabilities, the model can classify game states more effectively, enhancing both its accuracy and predictive performance.

7.7 Decision Tree Model Prediction `dt_predict_best_move`

The `dt_predict_best_move` function is crafted to predict the best possible move for our AI player in a Tic-Tac-Toe game. It utilizes a trained decision tree model to analyze the current board state and evaluate potential moves. Based on this analysis, the function recommends the optimal move, helping the AI player make a strategic decision.

Inputs

- `DecisionTreeNode *tree`: A pointer to the root of the trained decision tree.
- `char board[3][3]`: A 3x3 character array representing the Tic-Tac-Toe board. The values can be `'x'`, `'o'`, or `'b'` (for blank).
- `char current_player`: A character representing the current player ('x' or 'o').
- `int *best_row, int *best_col`: Pointers to store the row and column of the best move.

```
if (!tree) {
    printf("Error: Decision tree is not initialized!\n");
    return;
}
```

We start by verifying whether the decision tree (`tree`) is properly initialized. If the tree is `NULL`, an error message is displayed, and the function exits without continuing. This safeguard ensures that predictions are made only with a valid and trained decision tree. Additionally, several key variables are initialized to support the prediction process.

- `features[NUM_FEATURES]`: An one-dimensional array to store the board's current features as numerical values where 'x' maps to `1`, 'o' maps to `2`, and blank maps to `0` and allows the decision tree to understand the board's state in numerical (array) form.
- `max_positive_prob`: Initialized to `-1`, it tracks the highest probability for a positive outcome, indicating the best possible move.
- `*best_row` and `*best_col`: Set to `-1` as these variables store the coordinates of the best move found.
- `attempts`: Used to limit the number of attempts to find the best move.

We then attempt to find the best move within a maximum of five iterations. During each iteration, we evaluated all empty cells ('b') on the board and temporarily set the current player's move in the `features` array. Using our trained decision tree, we predicted the outcome of placing the current player's move in that cell. If the prediction indicated a positive outcome that was better than the current best (`max_positive_prob`), we updated the coordinates of the move (`temp_row` and `temp_col`) to reflect this new best option.

After evaluating each move, we reset the `features` array to mark the cell as empty again. This ensured that the board remained unchanged throughout the analysis, preserving its original state. If no positive move was identified after five iterations, we defaulted to selecting a random empty cell as the best move. This fallback strategy guaranteed that a move was always recommended, even when a clearly advantageous option was unavailable or unclear.

By combining predictive modeling with a reliable fallback mechanism, we ensured the function consistently provided a valid move. This approach maximized the potential for the current player to win, simulating intelligent decision-making in a Tic-Tac-Toe game. It allowed the AI to anticipate and make strategic moves based on patterns learned from training data, enhancing its effectiveness and adaptability.

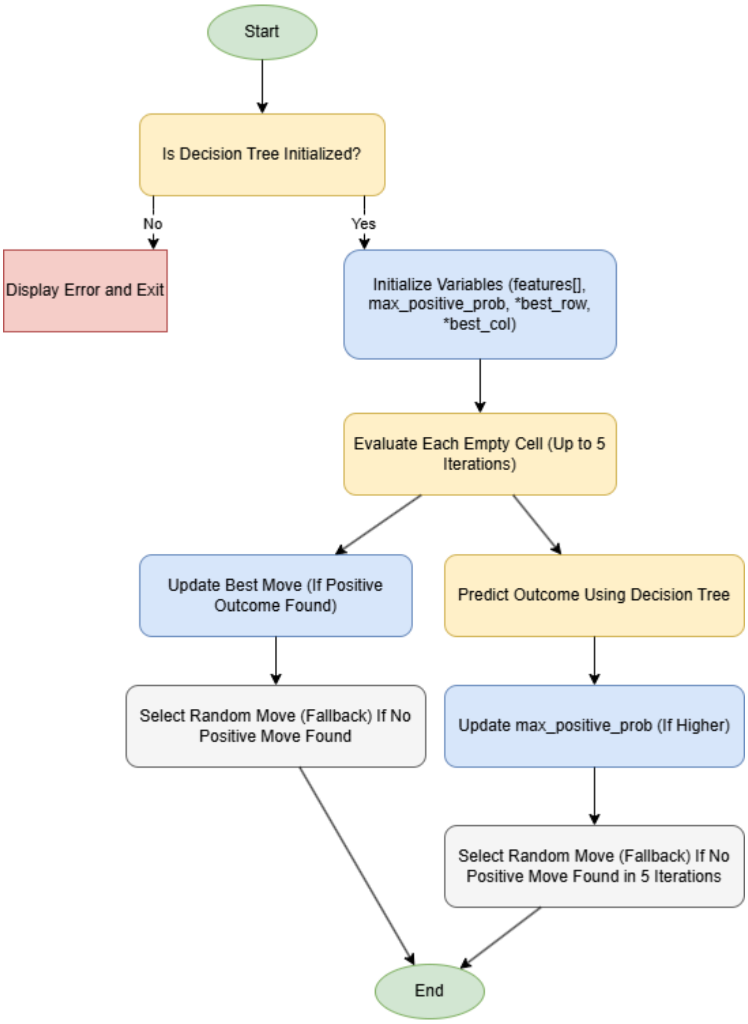


Fig.42: Flowchart of Decision Tree Model Prediction (dt_predict_best_move)

The flowchart above is to show visualisation for the function above.

7.8 Evaluation of Decision Tree Model and Confusion Matrix `evaluate_with_randomness()`

We evaluated our decision tree model by analyzing its performance on both the training and testing datasets. This evaluation process involved measuring key metrics such as accuracy and error rates, along with a detailed examination of the confusion matrices for both datasets. This approach gave us a comprehensive understanding of the metrics used and the formulas applied to assess the quality and reliability of our decision tree model for Tic-Tac-Toe gameplay.

The decision tree was trained and tested using a Tic-Tac-Toe dataset that was split into training and testing sets. To gauge the model's efficacy in predicting gameplay outcomes, we calculated critical performance metrics, including **accuracy**, **error rate**, and **confusion matrices**. These metrics provided valuable insights into the model's ability to make accurate and reliable predictions, ensuring it could effectively simulate intelligent decision-making in the game.

The confusion matrix is used to evaluate the performance of the classification model, which includes True Positive (TP), True Negative (TN), False Positive (FP), and False Negative (FN).

- **True Positive (TP): The number of instances correctly predicted as positive.**
- **True Negative (TN): The number of instances correctly predicted as negative.**
- **False Positive (FP): The number of instances incorrectly predicted as positive.**
- **False Negative (FN): The number of instances incorrectly predicted as negative.**

The following confusion matrices are used in the code:

- Training Confusion Matrix (`train_confusion[2][2]`)
- Testing Confusion Matrix (`test_confusion[2][2]`)

At the start of the evaluation process, all elements of the confusion matrix are initialized to zero. The model then iterates through each data point in both the training and testing datasets. If the predicted label matches the actual label, the corresponding count for either True Positive (TP) or True Negative (TN) is incremented. Conversely, if the predicted label does not match the actual label, the count for either False Positive (FP) or False Negative (FN) is incremented.

The error rate measures the percentage of incorrect predictions made by the decision tree model out of the total predictions during training and testing. In the code, the error rate is calculated using the `calculate_error_rate()` function, which iterates through the dataset and counts the number of incorrect predictions. Accuracy, on the other hand, reflects the percentage of correct predictions made by the model, offering a direct measure of its effectiveness. Together, these metrics provide a clear understanding of the model's performance and reliability in predicting outcomes.

```
float train_accuracy = (float)correct_predictions / train_size;
float test_accuracy = (float)correct_predictions / test_size;
```

The confusion matrix, accuracy and error rate is then saved to the file "`DecisionTree_ML/DTconfusion_matrix.txt`" for both training and testing phases as shown in the image below.

```
Decision Tree Training Confusion Matrix:
    True Positive (TP): 336
    False Positive (FP): 92
    True Negative (TN): 179
    False Negative (FN): 159

Confusion Matrix:
                Predicted Positive    Predicted Negative
Actual Positive              336                   159
Actual Negative               92                   179
---------------------------------------------------------
Training Accuracy: 67.23% (515/766)
Training Error Rate: 32.77%

Decision Tree Testing Confusion Matrix:
    True Positive (TP): 76
    False Positive (FP): 22
    True Negative (TN): 39
    False Negative (FN): 55

Confusion Matrix:
                Predicted Positive    Predicted Negative
Actual Positive               76                    55
Actual Negative               22                    39
---------------------------------------------------------
Testing Accuracy: 69.90% (115/192)
Testing Error Rate: 31.10%
```

Fig.43: DTconfusion_matrix.txt

Training and Testing Results

During the training phase, the decision tree was trained using 80% of the dataset. Training accuracy was calculated as the ratio of correctly classified samples to the total size of the training set. The training error rate offered insights into the number of misclassifications during this phase.

In the testing phase, the decision tree was evaluated on the remaining 20% of the dataset. Testing accuracy measured how well the model generalized to unseen data, while the testing error rate indicated the proportion of prediction errors during evaluation on the test set.

Review of Decision Tree Model

As shown in the results, the training accuracy was 67.23%, and the testing accuracy was 69.90%. These relatively close values suggest that the model performed similarly on both training and testing data, indicating that overfitting was not a significant concern.

The combination of predictive modeling and error analysis ensured that the decision tree was not only capable of making decisions but also able to learn from errors, aiming for improved accuracy in future iterations. The model was successfully evaluated using both training and testing datasets, demonstrating its ability to classify Tic-Tac-Toe board states into positive or negative outcomes. Additionally, the confusion matrix analysis provided valuable insights into the model's performance, highlighting its strengths and identifying areas for improvement.

7.9 Confusion Matrix Plot and Results (Decision Tree)

Installation Instruction

To use the plotting capabilities of the `matplotlib` library, install it using the following command:

```
pip3 install matplotlib
```

To evaluate the Decision Tree model, run the `confusionmatrix.py` file. This script generates a confusion matrix plot using the data from the `DTconfusion_matrix.txt` file. The resulting matrix will be displayed as a plot and saved as a PNG image named `DT_Confusion_Matrix.png`.

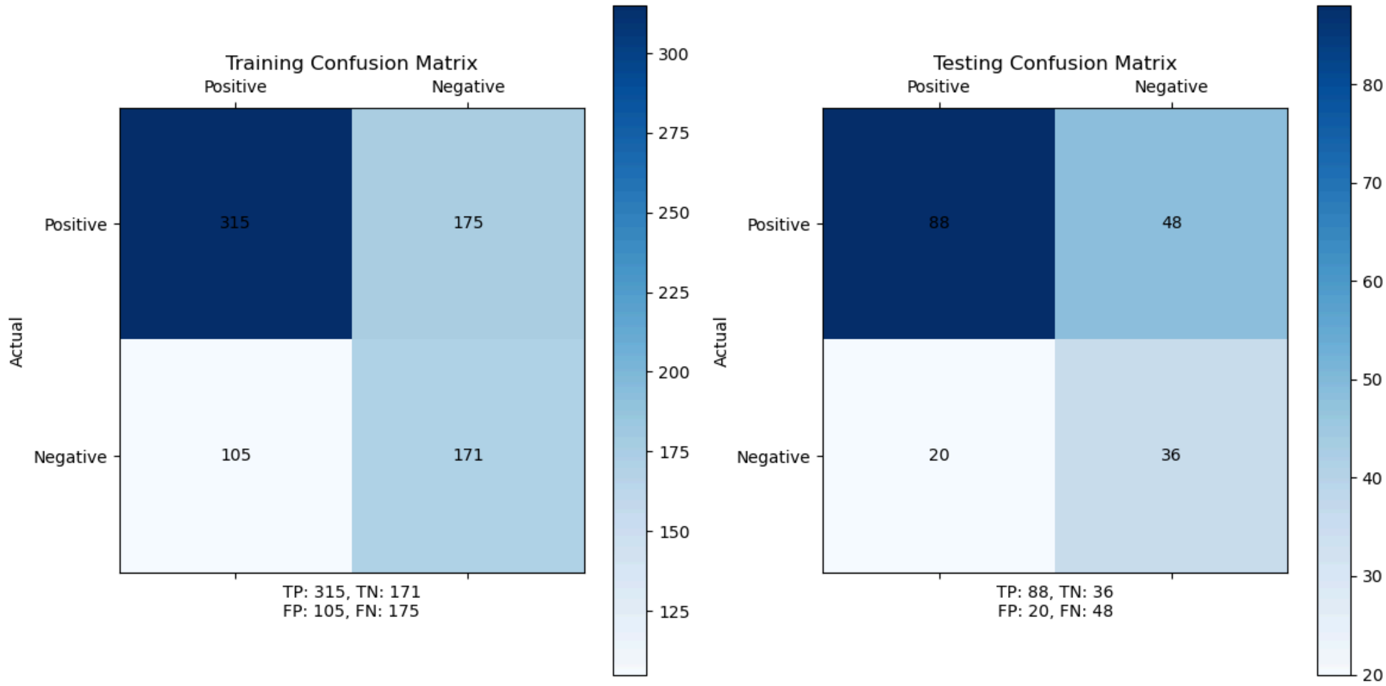


Fig.44: Confusion Matrix Plot and Results for Decision Tree Model

The **False Positives (FP)** are lower compared to **False Negatives (FN)** suggesting the model is more conservative in predicting positives. From my point of view, the trained decision tree model is optimal for our gameplay in easy mode for player to play against to have better chances of winning the game.

With all the implementations above, we have successfully integrated our trained decision tree model into the Tic-Tac-Toe game, allowing players to enjoy a competitive, intelligent AI opponent that offers a challenging yet entertaining experience. We also make enhancements on our decision tree model with **randomness** and **free up the memory allocation** which we mention it detailedly in the upcoming ‘Enhancement’ section.

8. Implementation of Minimax Algorithm (Hard Mode)

This section shows the methodology of implementing the Minimax Algorithm.

8.1. Purpose and Function

The Minimax Algorithm is a decision-making algorithm used in game theory, most commonly in two-player, zero-sum games like Tic-Tac-Toe, Chess, or Checkers. In the context of Tic-Tac-Toe, the algorithm allowed a computer to make the most optimal moves by simulating all possible future moves and selecting the one that led to the best outcome for the player. Below is an image to illustrate an example of the operation of the Minimax Algorithm, its usage of recursion, the application of Depth-First Search (DFS), and the concepts of minimizing and maximizing players.

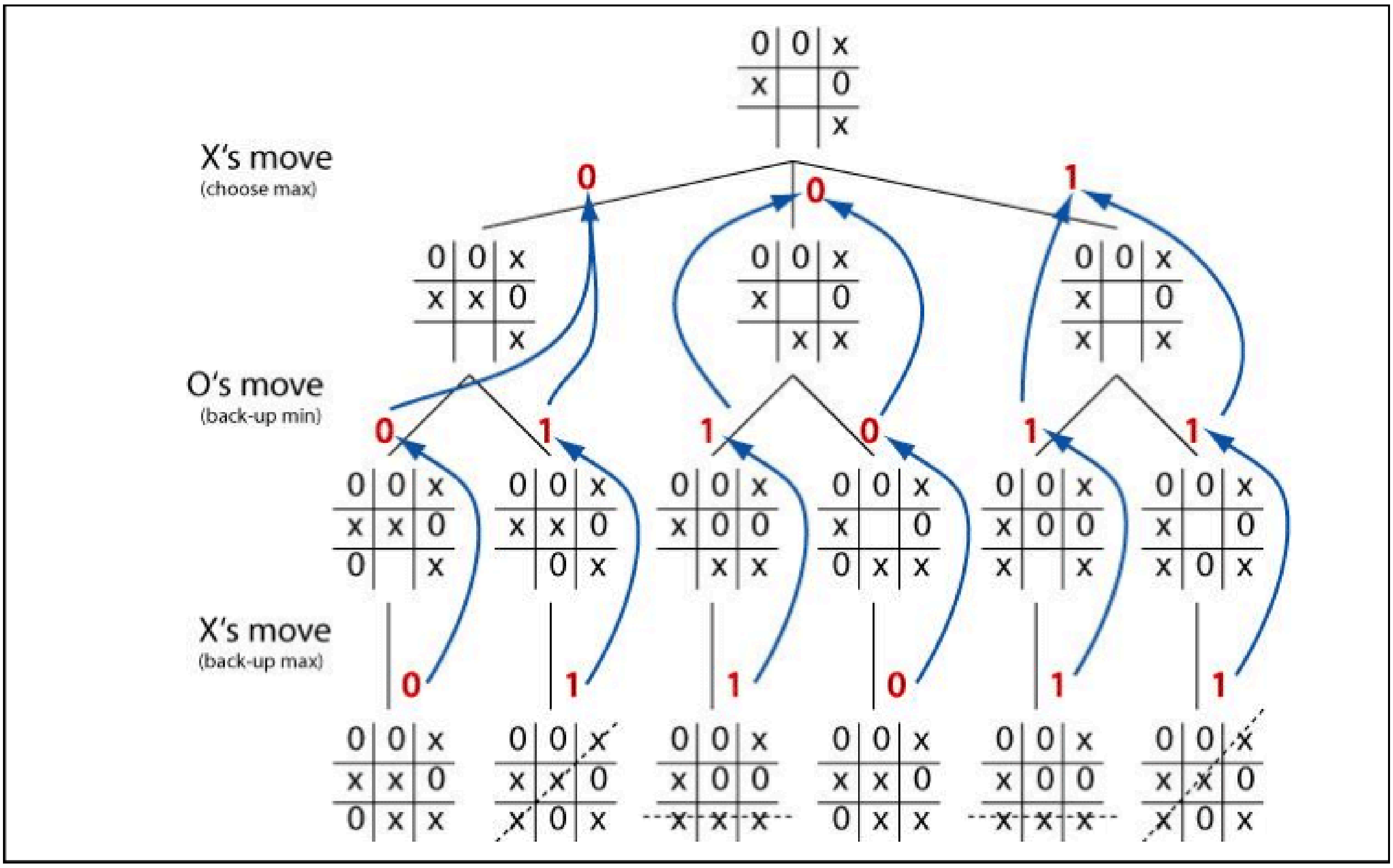


Fig.45: Minimax Diagram

8.2. Operation of Minimax Algorithm

Minimax Algorithm recursively simulated both players' moves, assigning scores based on whether a player won, lost or draws. It will evaluate all possible game states from the current game board, trying to maximize the score for the current player (X) and minimizing the score for the opponent (O). In the game's context, player (X) would be the AI opponent utilising Minimax while player (O) would be the children playing the game. The algorithm would assume that both players would play optimally and return the best possible move for player (X).

8.2.1. Recursion

- Base Case: Occurs when a terminal state is encountered, such as a win, lose or draw. At this stage, the game board would be evaluated with a score assigned to which a win for player (X) was awarded a score of 1 and draw being assigned a score of 0 as shown in Fig.45.

- Recursive Case: Occurs when the terminal state was not encountered and the function would call itself recursively, continuing to simulate each possible move for both players. In each recursive step, the algorithm simulated all valid moves for the current player and evaluated the resulting game states. The recursion continued until the algorithm reached a terminal state, at which point it would backtrack to the previous level and computed the best possible move based on the values returned from the child nodes.

8.2.2. Depth-First Search (DFS) Algorithm

- DFS Traversal in Minimax: The algorithm starts at the root node (the current game state), and recursively explores all possible moves (children). It evaluates each child node's value (win, loss, or draw) by recursively traversing deeper into the tree until a terminal state is reached.

- Backtracking: Once a terminal state is found, the algorithm backtracks to the parent node (the previous game state) and propagates the result (score) back up the tree. The values of the child nodes influence the decision at the parent node, allowing the algorithm to choose the optimal move at each level.

- Effect of DFS on Performance: While DFS ensures that every possible path is explored to its fullest depth, it can also be computationally expensive as the number of possible moves grows exponentially, especially in larger games. In Tic-Tac-Toe, however, the search space is manageable because the game has a limited number of possible moves.

8.2.3. Maximizing and Minimizing Players

- The core concept of Minimax is the alternating roles of the maximizing and minimizing players, which is what gives the algorithm its name. These two roles represent the two players in the game:
    1. Maximizing Player (AI): AI would try to maximize their chances of winning, choosing the best move that maximizes their score. The goal was to select the move that led to the highest possible score.
    2. Minimizing Player (Player): Assuming that the player would play optimally at every turn, they would try to minimize the maximizing player's chances of winning. The algorithm would explore all possible moves for this player and choose the move that minimizes the maximizing player's score. Essentially, the minimizing player tries to block the opponent's winning strategy by choosing the best possible counter-move.

- To summarize, the roles alternate as the algorithm recursively simulates each player's turn, with the maximizing player aiming for the highest score, and the minimizing player working to minimize the maximizing player's advantage.

8.2.4. Evaluation and Optimal Move Selection

Once the DFS traversal has explored all possible moves, the algorithm compares the scores at each level and selects the optimal move.

- Maximizes the score for the maximizing player (AI).
- Minimizes the score for the minimizing player (Player).

The final output of the Minimax Algorithm is the move that guarantees the best possible outcome either win or draw for AI, assuming both players play optimally. In the case where the Player made a suboptimal move is not of concern as that will play in the algorithm's favour.

8.3. Parameters of `Minimax()` Function

1. `board[GRID_SIZE][GRID_SIZE]` is a 3x3 grid, a nested list, where the game state will be represented.
2. `isMaximizing` is a boolean value for the function to decide which player is maximizing or minimizing.
3. `depth` is an integer value to count how deep the function searched for an optimal move.

9. Comparison & Interesting Findings / Assumptions

This section illustrates the comparison between the two machine learning models and the Minimax Algorithm, and some interesting findings.

9.1. Findings on Dataset

It was observed in the tic-tac-toe dataset, that the first 630 lines correspond to positive outcomes while the subsequent 330 lines were negative outcomes. Therefore if the dataset were to immediately be split into the first 80% lines for training and next 20% for testing. Training would be very skewed towards just predicting positive outcomes, and the model would not be able to predict any negative outcomes accurately Hence, the dataset was randomly shuffled the dataset before splitting it into training (80%) and testing (20%) datasets.

9.2. Comparison Between Minimax Algorithm and Decision Tree Model

The Minimax Algorithm is a foundational approach in game theory, widely used for turn-based games like Tic-Tac-Toe. Its primary goal is to identify the optimal move by maximizing the player's chances of winning while minimizing the opponent's potential to gain an advantage. By evaluating all possible game states, the algorithm ensures the best possible decision is made for each move.

9.2.1 Key Characteristics of Minimax:

- **Optimal Play**: The algorithm guarantees the best move, assuming both players make optimal decisions.
- **Recursive Nature**: Minimax relies on a recursive evaluation of game states to determine the ideal move.
- **Complete Search Space Exploration**: It exhaustively evaluates all possible moves and responses, resulting in an optimal solution.

On the other hand, the Decision Tree Model we used was trained on historical Tic-Tac-Toe data to predict the best move based on observed patterns. Its primary function is to classify game states as leading to either positive or negative outcomes.

9.2.2 Key Characteristics of the Decision Tree Model:

- **Learning-Based Approach**: The model leverages a dataset to learn and generalize patterns, allowing it to predict favorable moves based on past data.
- **Randomness Integration**: By incorporating randomness at leaf nodes, the model introduces variability, making its decision-making less predictable.
- **Lower Computational Complexity**: Unlike `Minimax`, decision trees do not evaluate every possible move, making them faster and more efficient in runtime.

9.2.3 Limitations

- Unlike Minimax, the decision tree does not always guarantee the optimal move since it depends on learned patterns rather than a complete exploration of all possible game states.
- The model's effectiveness is heavily reliant on the quality and comprehensiveness of the training data. It may struggle with novel scenarios that are not represented in the dataset.

9.2.4 Conclusion

The Minimax Algorithm excels in delivering optimal and theoretically unbeatable strategies, ensuring perfect play if both players act optimally. However, its predictability and high computational complexity limit its practical application to simpler games or scenarios where variability and spontaneity are not critical.

In contrast, the Decision Tree Model—though potentially suboptimal in some cases—offers unique advantages that enhance gameplay. Its ability to introduce randomness reduces predictability, making the gameplay feel less mechanical and more engaging. Additionally, its lower computational demands make it well-suited for real-time play.

The key difference lies in the trade-off between optimality and variability. Minimax is ideal for scenarios where perfect strategy is crucial, while the decision tree shines in situations where dynamic, human-like play and real-time performance are more desirable.

9.3. Comparison between Minimax Algorithm and Naive Bayes Model

The Naive Bayes model is weaker than the Minimax Algorithm. The Naive Bayes algorithm only trains the model to get the probability of winning in all 9 squares and ranks them in descending order. In order words, it would make a move in the square on the grid that has the next highest probability of winning as the game goes on. For example, based on the NB model weights, the center square always leads to the highest probability of winning, hence it would always try to make a move there. However, if the square is filled by the player's move, then it would make a move in the square with the next highest possibility of winning, which is usually the corners of the grid.

Thus, the NBmodel does not detect the winning move for both the player and itself. Resulting in it not being able to always stop the player from winning or making its winning move when presented with the chance.

10. Enhancements

This section shows the enhancements the team made to increase the efficiency, modularity and readability of the program.

10.1. Minimax (Imperfectness) (Medium Mode)

Introducing error to the algorithm winnability of the player.

10.1.1 Depth Limitation

1. As Minimax Algorithm is known for its depth-first search and accountability of all possible moves to make an optimal move, to counteract this, `depthLimit` variable is introduced.
2. `depthLimit` is an integer representing the number of recursive steps allowed for the function before backtracking.
3. Catering to the needs of the players who would like to challenge and have a fair chance to win, thus increasing engagement and satisfaction, the AI has been modified with an additional parameter to limit the recursion to 4.
4. With a `depthLimit` of 4, the AI evaluates a limited number of moves, resulting in less comprehensive analysis. This reduction in the depth of the search allows the player a higher chance of winning, as the AI's decisions are based on fewer potential outcomes.

10.2. Minimax (Alpha-Beta Pruning)

Below are illustrations designed to aid in explaining the concept of alpha-beta pruning.

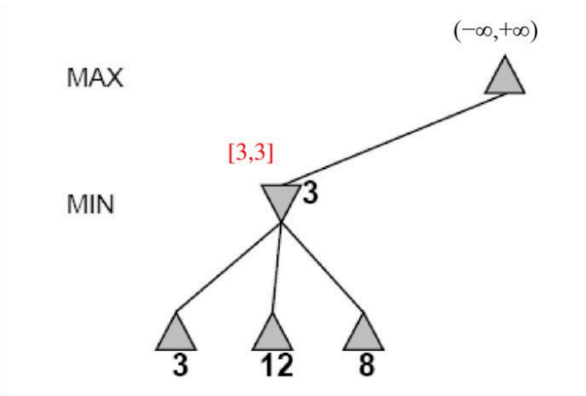


Fig.46: Alpha Beta Pruning 1st node

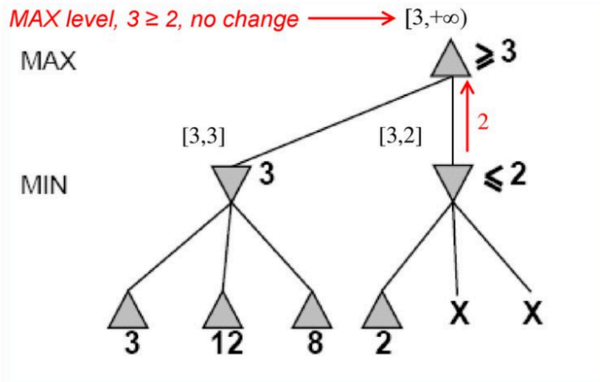


Fig.47: Alpha Beta Pruning 2nd node

10.2.1 Alpha and Beta

1. `alpha` is an integer value representing the best score that the maximizing player could attain which is the highest score. A low number, `-∞`, was set as the initial value in Fig.46. This way any starting score received will be higher and can be used to compare with the subsequent scores.
2. `beta` is an integer value representing the best score that the minimizing player could attain, which is the lowest score. A high number, `∞`, was set as the initial value in Fig.46. This way any starting score received will be lower and can be used to compare with the subsequent scores.

10.2.2 Operation of Optimized Minimax

1. In Fig.47 example, in the left subtree, it can be observed that the minimizing player chose the lowest score 3 and the value is assigned to `beta`. The recursion will backtrack to the maximizing player to choose score 3 as that is the higher score when compared to `-∞`. The value is then assigned to `alpha` when the recursion backtrack again.
2. In Fig.47 , the next terminal node at the right subtree is 2. After the recursion backtracked, the minimizing player would choose 2 as that is the lowest value at that point. Following, `beta` would be compared with 2 to select the lowest of them. In this case, 2 is lower than 3 and `beta` would be updated to 2. Next, the recursion backtrack again and compare if `beta` is smaller or equal to `alpha`.
3. If the current `beta` value is smaller or equal than `alpha`, the remaining branches of the node are pruned, or skipped, because they cannot produce a better outcome as the minimizing player would only select the lowest score.
4. Alpha-Beta Pruning refines the process of the Minimax algorithm by reducing the number of game moves evaluated, improving computational efficiency.

10.3. Modular Approach

The program functions are separated into smaller, specific tasks that are categorized into various parts, as outlined in our Functions Descriptions page (e.g. drawing of UI features, algorithm functions, and game logic functions).

1. This modular approach makes the program more organized, and easier to manage.

2. It also makes the program more scalable, in the sense where changes and expansion of the functions can be developed in the future.
3. It promotes independent development where each function can be tested and debugged separately without affecting others.
4. This structure enhances memory efficiency by localizing memory usage within functions, reducing redundancy through reusable code, and enabling better cache utilization.
5. Thus, through categorizing of the functions, the program becomes more maintainable and adaptable for future improvements and optimizes memory management, ensuring smooth and efficient performance.

Enhancements of Decision Tree Model

10.4. Decision Tree Model Prediction with Randomness `predict_with_randomness`

To enhance the decision-making capabilities of our decision tree model, the `predict_with_randomness` function is to be implemented. This function is a pivotal element in the AI's strategy for Tic-Tac-Toe, as it evaluates the potential moves on the board by combining logical predictions with a hint of randomness. This blend of structure and unpredictability makes the AI's gameplay feel more human-like, less predictable, and more enjoyable for players, fostering a dynamic and engaging gaming experience. When a player makes their move, the current state of the board is passed to the AI for analysis. The board is represented as a feature array, where:

- `1` represents the AI's symbol (`'x'`)
- `2` represents the Player's symbol (`'o'`)
- `0` represents an empty space (`'b'`)

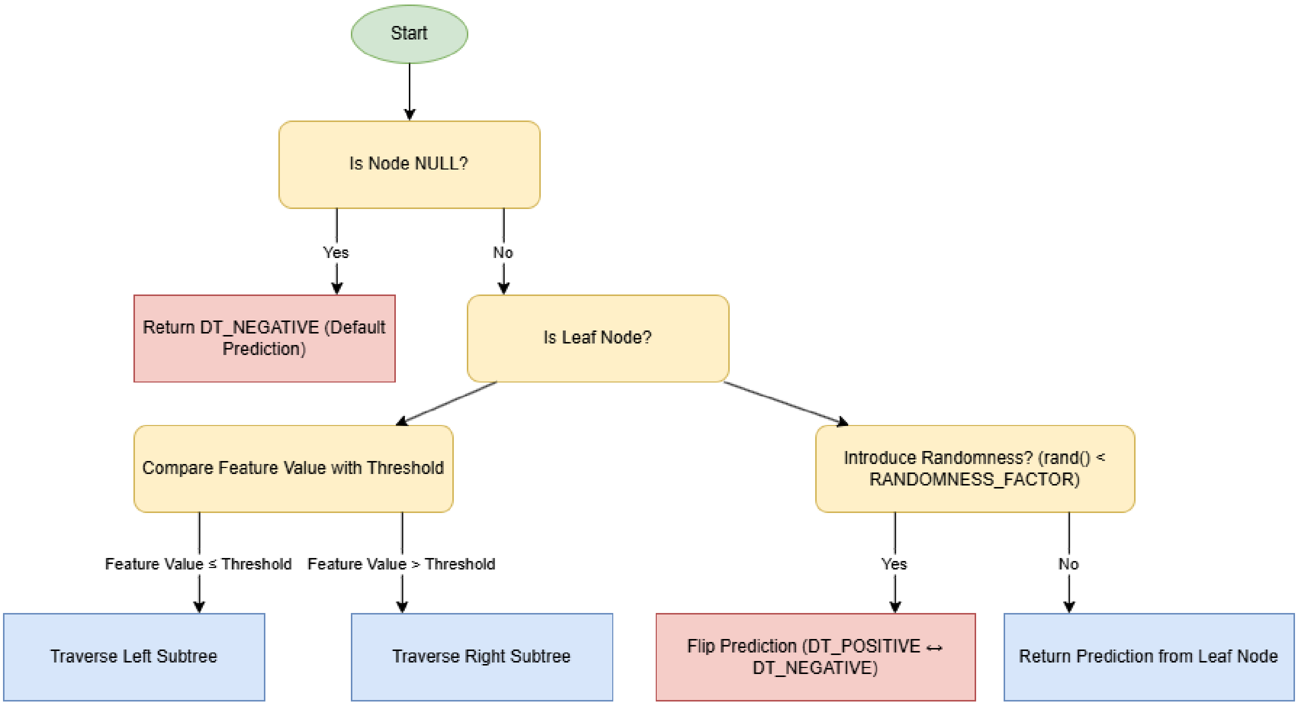


Fig.48: Flowchart for Decision Tree Model Prediction with Randomness

The trained decision tree model evaluates the board by starting at the root of the tree and traversing it based on the board's features. The function examines specific conditions, such as "Is this cell occupied by 'x' or 'o'?" and navigates left or right through the tree accordingly. When a leaf node is reached, the tree provides a prediction: **positive** if the move is favorable, or **negative** if it is unfavorable.

If a move is determined to be favorable, there is a small, controlled chance (determined by the `RANDOMNESS_FACTOR`) that the AI may choose an alternative move to introduce unpredictability. On the other hand, if a move is deemed unfavorable, the AI continues exploring other options to identify a better course of action.

This enhancement through randomness ensures that the AI doesn't play identically in every game, even when presented with the same board state. As a result, the gameplay becomes more dynamic, challenging, and engaging, keeping players entertained and improving the overall experience.

For example, during a game, if it's the player's turn and they place their symbol (`o`) in the center of the board, the board state is updated and passed to the trained decision tree model for evaluation. The AI then uses this updated state to determine its next move, blending strategic prediction with an element of unpredictability to create a more human-like and less repetitive response.

Player's Turn:

```
x | b | o
b | o | b
x | b | b
```

Fig.49: Player's Turn

Now, when it is the AI's turn, our AI analyzes the board using the trained decision tree and identifies potential winning moves (e.g., completing a row or blocking the player's win). At a leaf node, it evaluates whether a move is favorable or not. With slight randomness factor that influence the AI to explore other moves. Then the outcome is our trained decision tree AI places its symbol (`'x'`) strategically such as blocking the player's potential win. Lastly, we have successfully implemented our trained Decision Tree model into our TicTacToe game for players to play against.

AI's Turn:

```
x | b | o
b | o | b
x | b | x
```

Fig.50: AI's Turn

10.5. Memory Free Allocation for the Decision Tree Model `free_tree()`

The decision tree model uses dynamic memory allocation to grow nodes during training, with each node represented by a `DecisionTreeNode` structure that includes pointers to its left and right children. This dynamic allocation, if not managed properly, can lead to memory leaks when the tree is no longer needed. To prevent this, a dedicated function called `free_tree()` was implemented to traverse the tree and free memory systematically to deallocate all nodes in the decision tree in a recursive manner.

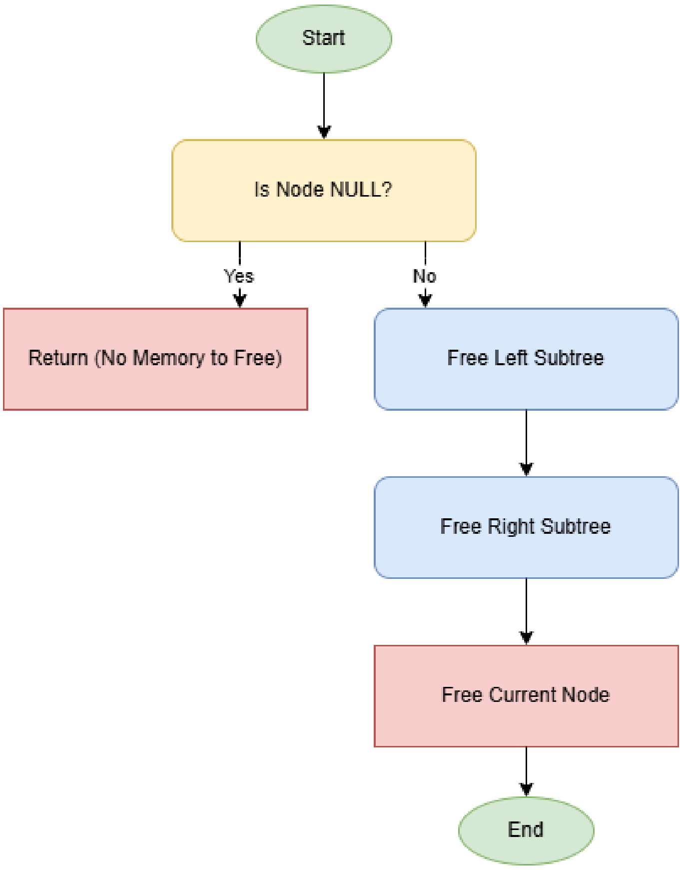


Fig.51: Flowchart of Memory Free Allocation for the Decision Tree Model

```
if (node == NULL) return;
```

The function first checks if the current node is `NULL`. If so, it returns immediately, as there is no memory to free for that path.

```
free_tree(node->left);
free_tree(node->right);
```

It then recursively calls itself to free the left and right subtrees of the current node. This recursive approach ensures a clean and organized way to deallocate memory for all nodes in the tree.

```
free(node);
```

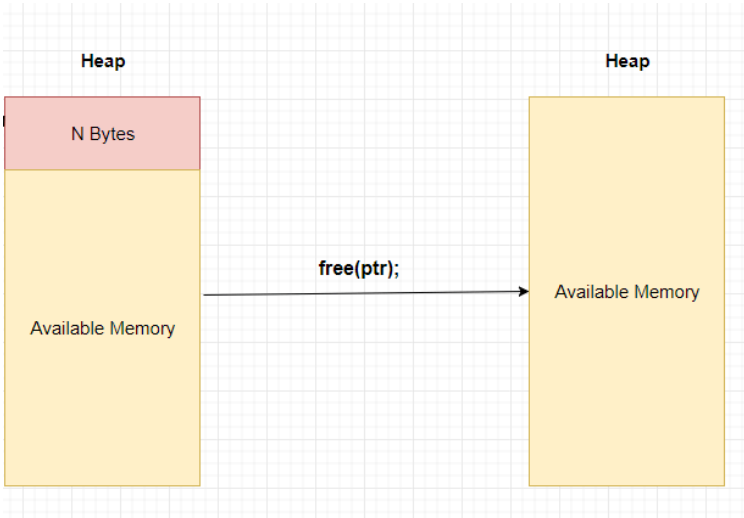


Fig.52: Memory Dellocation

After freeing the left and right subtrees, the function deallocates the memory for the current node itself. In scenarios where the decision tree model is repeatedly trained and tested (e.g., for different datasets or parameter tuning), efficient memory deallocation is crucial. The `free_tree()` function ensures that all memory from previous iterations is cleared before new allocations, preventing memory buildup over time which helps in robustness in repeated training.

11. Gauging of Difficulty Levels

This section shows the number of times the computer wins out of ten games for each difficulty level to gauge the level of each difficulty.

11.1. Level of Difficulty for Easy Mode

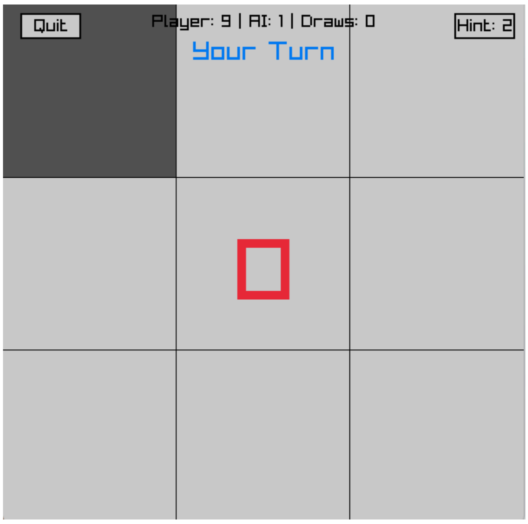


Fig.53: Win Rate for Easy Mode (Naive Bayes)

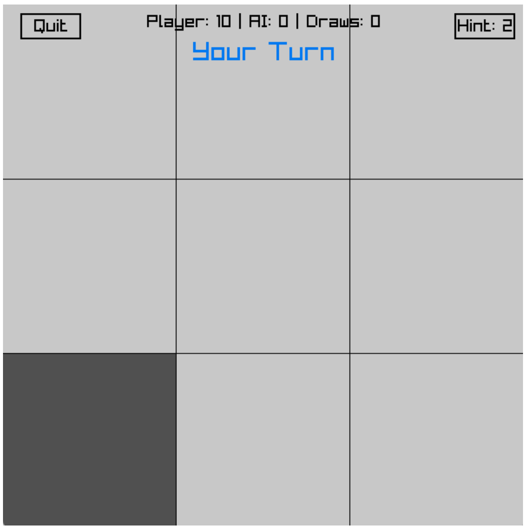


Fig.54: Win Rate for Easy Mode (Decision Tree)

As shown in the figures above, the win rate for easy mode is very high, with 90% for Naive Bayes model and 100% for Decision Tree model. This allows the players to win against the computer easily, building their confidence.

11.2. Level of Difficulty for Medium Mode

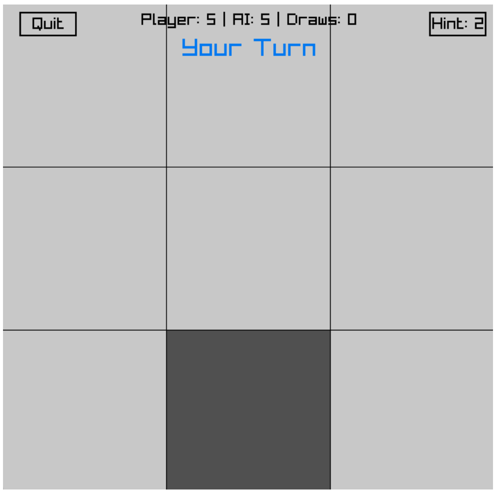


Fig.55: Win Rate for Medium Mode

As shown in the figure above, the win rate for medium mode is 50%, allowing players to win against the computer but at the same time introducing some challenges, to achieve higher playability.

11.3. Level of Difficulty for Hard Mode

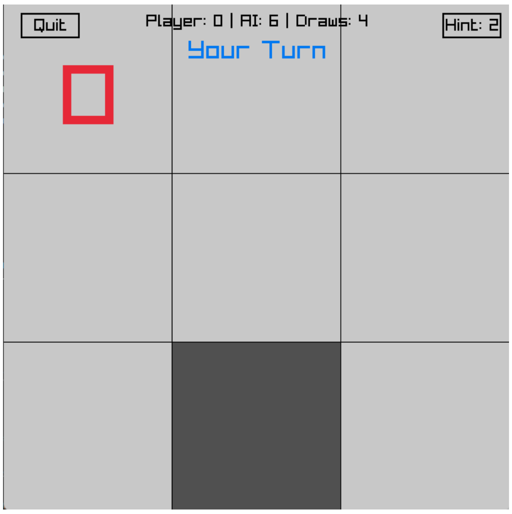


Fig.56: Win Rate for Hard Mode

As shown in the figure, the win rate for the hard mode is 0%. The best the players could achieve is a draw as the minimax algorithm has searched through all possibilities before the players do. The hard mode is essentially unwinnable, but players can learn from the computers' moves and utilise the losing experience to make them better at this game.

12. Appendix

Our Youtube Demo Video: Click here to watch it! [Recorded by: Jian Xin, Edited by: Alicia]

Source Codes

12.1. Main.c

```
#include "main.h"
#include "DecisionTree_ML/decisiontree.h"


// Define the global variables
GridSymbol titleSymbols[TITLE_GRID_SIZE][TITLE_GRID_SIZE];
FallingSymbol symbols[MAX_SYMBOLS];
TitleWord titleWords[5];
Difficulty currentDifficulty = MEDIUM; // Initialize difficulty to a value, doesn't have to be medium
Cell grid[GRID_SIZE][GRID_SIZE]; // Initialize the grid with empty cells
PlayerTurn currentPlayerTurn = PLAYER_X_TURN; // Initialize the current player turn to Player X
bool gameOver = false; // Initialize the game over flag to false
Cell winner = EMPTY; // Initialize the winner to empty
GameState gameState = MENU; // Initialize the game state to menu
bool isTwoPlayer = false; // Flag to check if it's a two-player or single-player game
float titleCellScales[TITLE_GRID_SIZE][TITLE_GRID_SIZE] = {0};
float titleRotations[TITLE_GRID_SIZE][TITLE_GRID_SIZE] = {0};
float titleAnimSpeed = 2.0f; // Animation speed for title cells
float buttonVibrationOffset = 0.0f; // Vibration offset for buttons
float vibrationSpeed = 15.0f; // Speed of vibration, increase this to intensify the vibration
float vibrationAmount = 2.0f; // Amount of vibration
AIModel currentModel = NAIVE_BAYES; // Default to Naive Bayes
int aiWins = 0; // Set aiWins to 0
int totalGames = 0; // Set the total number of games to 0
Confetti confetti[MAX_CONFETTI]; // Set the maximum number of confetti particles
bool showPartyAnimation = false; // Flag to check if the party animation should be shown
bool allInactive = true; // Flag to check if all confetti particles are inactive
struct GetHint hint = { -1, -1, 0, 0}; // Declare hint object to store best move and hint counts for player
int winningCells[3][2] = {{-1,-1}, {-1,-1}, {-1,-1}}; // Store winning cell coordinates

// Initialize the ModeStats structs
ModeStats mediumStats = {0, 0, 0, 0};
ModeStats hardStats = {0, 0, 0, 0};
ModeStats naiveBayesStats = {0, 0, 0, 0};
ModeStats decisionTreeStats = {0, 0, 0, 0};

// Initialize the sound variables
Sound buttonClickSound;
```

```
Sound popSound;
Sound victorySound;
Sound loseSound;
Sound drawSound;
Sound mainMenuSound;
Sound playSound;

// Main function
int main(void)
{
    InitWindow(SCREEN_WIDTH, SCREEN_HEIGHT, "Tic-Tac-Toe");
    InitAudioDevice();  // Initialize audio device

    // Load the icon image
     Image icon = LoadImage("assets\\icon.png");   // Make sure the file path is
correct
    SetWindowIcon(icon);  // Set the window icon
    UnloadImage(icon);  // Unload the image after setting the icon

    buttonClickSound = LoadSound("assets\\ButtonClicked.mp3");  // Load the button
click sound
    popSound = LoadSound("assets\\Pop.mp3");  // Load the pop sound
    victorySound = LoadSound("assets\\FFVictory.mp3");  // Load the victory sound
    loseSound = LoadSound("assets\\MarioLose.mp3");  // Load the lose sound
    drawSound = LoadSound("assets\\Draw.mp3");  // Load the draw sound
    mainMenuSound = LoadSound("assets\\MainMenu.mp3");  // Load the main menu sound
    playSound = LoadSound("assets\\Play.mp3");  // Load the play sound

    // After loading each sound, set its volume (between 0.0f to 1.0f)
    SetSoundVolume(buttonClickSound, 0.4f);  // 40% volume
    SetSoundVolume(popSound, 0.4f);
    SetSoundVolume(victorySound, 0.4f);
    SetSoundVolume(loseSound, 0.4f);
    SetSoundVolume(drawSound, 0.4f);
    SetSoundVolume(mainMenuSound, 0.4f);
    SetSoundVolume(playSound, 0.4f);

    InitSymbols();  // Initialize the falling symbols
    InitTitleWords();  // Initialize the title words
    InitConfetti(); // Initialize the confetti

    // Naive Bayes Machine Learning for easy mode
     char boards[1000][NUM_POSITIONS + 1];        // Array to store attrbutes of
tic-tac-toe.data dataset
     int outcomes[1000];                           // Array to store outcomes of
tic-tac-toe.data dataset
     int total_records = 0;                       // Count for number of lines in
dataset

    // Load data
    load_data("tic-tac-toe.data", boards, outcomes, &total_records);

    // Split data
```

```
    char train_boards[800][NUM_POSITIONS + 1];  // Array for attributes of training
dataset
     int train_outcomes[800];                       // Array for outcomes of training
dataset
     char test_boards[200][NUM_POSITIONS + 1];    // Array for attributes of testing
dataset
     int test_outcomes[200];                        // Array for outcomes of testing
dataset
      int train_size = 0, test_size = 0;              // Count number of lines in
training and testing dataset respectively

       split_data(boards, outcomes, total_records, train_boards, train_outcomes,
test_boards, test_outcomes, &train_size, &test_size, RATIO);

    // Train model
    NaiveBayesModel NBmodel;
    train_NBmodel(&NBmodel, train_boards, train_outcomes, train_size);

    // Save model weights to a file
    save_NBmodel(&NBmodel, "NBmodel/NBmodel_weights.txt");

    // Test model
    char mode[] = "w";
    char type[] = "Training";
       test_NBmodel("NBmodel/NBmodel_confusion_matrix.txt", mode, type, &NBmodel,
train_boards, train_outcomes, train_size);
    strcpy(mode, "a");
    strcpy(type, "Testing");
       test_NBmodel("NBmodel/NBmodel_confusion_matrix.txt", mode, type, &NBmodel,
test_boards, test_outcomes, test_size);
    // End of Machine Learning

    DecisionTreeNode TDmodel;
    growth_Tree(&TDmodel);

    while (!WindowShouldClose())
    {
          if (gameState == MENU || gameState == DIFFICULTY_SELECT || gameState ==
MODEL_SELECT) {
            if (!IsSoundPlaying(mainMenuSound)) {
                PlaySound(mainMenuSound);  // Play main menu sound
            }
            StopSound(playSound);  // Ensure play sound is stopped
        } else if (gameState == GAME) {
            if (!IsSoundPlaying(playSound)) {
                PlaySound(playSound);  // Play play sound
            }
            StopSound(mainMenuSound);  // Ensure main menu sound is stopped
        } else {
              StopSound(mainMenuSound);  // Stop main menu sound when leaving these
states
            StopSound(playSound);  // Stop play sound when leaving the game state
        }
```

```
          if (gameState == MENU || gameState == DIFFICULTY_SELECT || gameState ==
MODEL_SELECT) {
            UpdateSymbols();  // Update the falling symbols
            UpdateTitleWords();  // Update the title words
        }

        if (gameState == MENU) {
            if (IsMouseButtonPressed(MOUSE_LEFT_BUTTON)) {
                Vector2 mousePos = GetMousePosition();
                // Single Player button
                        if (mousePos.x >= SCREEN_WIDTH/2 - 100 && mousePos.x <=
SCREEN_WIDTH/2 + 100 &&
                             mousePos.y >= SCREEN_HEIGHT/2 + 60 && mousePos.y <=
SCREEN_HEIGHT/2 + 100) {
                    PlaySound(buttonClickSound);  // Play sound on button click
                    isTwoPlayer = false;
                      gameState = DIFFICULTY_SELECT;  // go to difficulty selection
instead of game
                }
                // Two Player button
                    else if (mousePos.x >= SCREEN_WIDTH/2 - 100 && mousePos.x <=
SCREEN_WIDTH/2 + 100 &&
                             mousePos.y >= SCREEN_HEIGHT/2 + 120 && mousePos.y <=
SCREEN_HEIGHT/2 + 160) {
                    PlaySound(buttonClickSound);  // Play sound on button click
                    isTwoPlayer = true;
                    gameState = GAME;
                    InitGame();
                }
                // Exit button
                    else if (mousePos.x >= SCREEN_WIDTH/2 - 100 && mousePos.x <=
SCREEN_WIDTH/2 + 100 &&
                             mousePos.y >= SCREEN_HEIGHT/2 + 180 && mousePos.y <=
SCREEN_HEIGHT/2 + 220) {
                    PlaySound(buttonClickSound);  // Play sound on button click
                    break;  // Exit the game loop
                }
            }
        }
        else if (gameState == GAME)
        {
                  UpdateGame(buttonClickSound, popSound, victorySound, loseSound,
drawSound, &NBmodel, &TDmodel);
        }
        else if (gameState == GAME_OVER)
        {
            UpdateGameOver(buttonClickSound);
        }
        else if (gameState == DIFFICULTY_SELECT) {
            if (IsMouseButtonPressed(MOUSE_LEFT_BUTTON)) {
                Vector2 mousePos = GetMousePosition();
```

```
                // Back button
                 if (mousePos.x >= 20 && mousePos.x <= SCREEN_WIDTH/6 && mousePos.y
>= 10 && mousePos.y <= 40) {
                    PlaySound(buttonClickSound);  // Play sound on button click
                    gameState = MENU;
                }

                if (mousePos.x >= SCREEN_WIDTH/2 - BUTTON_WIDTH/2 &&
                    mousePos.x <= SCREEN_WIDTH/2 + BUTTON_WIDTH/2) {
                    // easy button
                    if (mousePos.y >= SCREEN_HEIGHT/2 &&
                        mousePos.y <= SCREEN_HEIGHT/2 + BUTTON_HEIGHT) {
                        PlaySound(buttonClickSound);  // Play sound on button click
                        currentDifficulty = EASY;
                              gameState = MODEL_SELECT;    // go to model selection
instead of game
                        InitGame();
                    }
                    // medium button
                    else if (mousePos.y >= SCREEN_HEIGHT/2 + BUTTON_HEIGHT + 20 &&
                                mousePos.y <= SCREEN_HEIGHT/2 + BUTTON_HEIGHT * 2 +
20) {
                        PlaySound(buttonClickSound);  // Play sound on button click
                        currentDifficulty = MEDIUM; // go to imperfect minimax
                        gameState = GAME;
                        InitGame();
                    }
                    // hard button
                    else if (mousePos.y >= SCREEN_HEIGHT/2 + (BUTTON_HEIGHT + 20) *
2 &&
                             mousePos.y <= SCREEN_HEIGHT/2 + (BUTTON_HEIGHT + 20) *
2 + BUTTON_HEIGHT) {
                        PlaySound(buttonClickSound);  // Play sound on button click
                        currentDifficulty = HARD; // go to perfect minimax
                        gameState = GAME;
                        InitGame();
                    }
                }
            }
        }
        else if (gameState == MODEL_SELECT) {
            if (IsMouseButtonPressed(MOUSE_LEFT_BUTTON)) {
                Vector2 mousePos = GetMousePosition();

                // Back button
                 if (mousePos.x >= 20 && mousePos.x <= SCREEN_WIDTH/6 && mousePos.y
>= 10 && mousePos.y <= 40) {
                    PlaySound(buttonClickSound);  // Play sound on button click
                    gameState = DIFFICULTY_SELECT;
                }

                // Naive Bayes button
                Rectangle nbBtn = {
```

```
                SCREEN_WIDTH/2 - BUTTON_WIDTH/2,
                SCREEN_HEIGHT/2,
                BUTTON_WIDTH,
                BUTTON_HEIGHT
            };

            // Decision Tree button
            Rectangle dtBtn = {
                SCREEN_WIDTH/2 - BUTTON_WIDTH/2,
                SCREEN_HEIGHT/2 + BUTTON_HEIGHT + 20,
                BUTTON_WIDTH,
                BUTTON_HEIGHT
            };

            if (CheckCollisionPointRec(mousePos, nbBtn)) {
                PlaySound(buttonClickSound);
                currentModel = NAIVE_BAYES;
                gameState = GAME;
                InitGame();
            }
            else if (CheckCollisionPointRec(mousePos, dtBtn)) {
                PlaySound(buttonClickSound);
                currentModel = DECISION_TREE;
                gameState = GAME;
                InitGame();
            }
        }
    }

    BeginDrawing();  // Begin drawing
    ClearBackground(RAYWHITE); // Clear the background to white
    if (gameState!=GAME && gameState!=GAME_OVER)
    {
        // resets hintCount when not in game
        hint.hintCountX = 0;
        hint.hintCountO = 0;
    }

    switch(gameState) {
        case MENU:
            DrawSymbols();  // Draw the falling symbols
            DrawTitleWords();  // Draw the jumping title words
            DrawMenu();  // Draw the menu
            break;
        case DIFFICULTY_SELECT:
            DrawSymbols();  // Draw the falling symbols
            DrawDifficultySelect();  // Draw the difficulty selection
            break;
        case MODEL_SELECT:
            DrawSymbols();  // Draw the falling symbols
            DrawModelSelect(); // Draw the model selection
            break;
        case GAME:
```

```
                DrawGame();  // Draw the game
                break;
            case GAME_OVER:
                DrawGame();  // Draw the game
                DrawGameOver();  // Draw the game over screen
                  if (showPartyAnimation == true) {    // If the party animation is
active
                    UpdateConfetti();    // Update the confetti
                    DrawConfetti(); // Draw the confetti
                }
                break;
        }

        EndDrawing();  // End drawing
    }

    UnloadSound(buttonClickSound);  // Unload the button click sound
    UnloadSound(popSound);  // Unload the pop sound
    UnloadSound(victorySound);  // Unload the victory sound
    UnloadSound(loseSound);  // Unload the lose sound
    UnloadSound(drawSound);  // Unload the draw sound
    UnloadSound(mainMenuSound);  // Unload the main menu sound
    UnloadSound(playSound);  // Unload the play sound
    CloseAudioDevice();  // Close the audio device
    CloseWindow();  // Close the window
    return 0;
}

// function to point to the current game mode stats
ModeStats* GetCurrentModeStats() {
    if (currentDifficulty == EASY) {
                return   (currentModel   ==   NAIVE_BAYES)   ?   &naiveBayesStats   :
&decisionTreeStats;
    }
    return (currentDifficulty == MEDIUM) ? &mediumStats : &hardStats;
}

void RandomizeStartingPlayer() { //Randomize starting player
    // 50% chance for each player to start
    if (GetRandomValue(0, 1) == 0) {
        currentPlayerTurn = PLAYER_X_TURN;  // Human starts
    } else {
        currentPlayerTurn = PLAYER_O_TURN;  // AI starts
    }
}

//  gcc  -o  main  main.c  DecisionTree_ML/*.c  NBmodel/*.c  GameFunctions/*.c  -I.
-I./DecisionTree_ML -I./NBmodel -I./GameFunctions -L. -lraylib -lopengl32 -lgdi32
-lwinmm
// ./main
```

12.2. AI.c

```c
#include "main.h"
#include "DecisionTree_ML/decisiontree.h"
extern Cell grid[GRID_SIZE][GRID_SIZE];

// AI's turn using MiniMax algorithms and Machine Learning models
void AITurn(Sound victorySound, Sound loseSound, Sound drawSound, NaiveBayesModel
*model)
{
    int bestScore = -1000;
    int bestRow = -1;
    int bestCol = -1;

    // Ensure this function only applies in medium and hard modes
    if (currentDifficulty == EASY) {
        predict_move(model, grid, &bestRow, &bestCol);
    }

    // Medium mode: use Minimax with limited depth search of 4
    else if (currentDifficulty == MEDIUM) {
        int depthLimit = 4; // Set a depth limit for medium difficulty
        for (int i = 0; i < GRID_SIZE; i++) {
            for (int j = 0; j < GRID_SIZE; j++) {
                if (grid[i][j] == EMPTY) {
                    grid[i][j] = PLAYER_O;
                    int score = Minimax(grid, false, 0, depthLimit, -1000, 1000);
                    grid[i][j] = EMPTY;

                    if (score > bestScore) {
                        bestScore = score;
                        bestRow = i;
                        bestCol = j;
                    }
                }
            }
        }
    }
    // Hard mode: full Minimax search
    else if (currentDifficulty == HARD) {
        int depthLimit = 9; // Full depth for hard mode
        for (int i = 0; i < GRID_SIZE; i++) {
            for (int j = 0; j < GRID_SIZE; j++) {
                if (grid[i][j] == EMPTY) {
                    grid[i][j] = PLAYER_O;
                    int score = Minimax(grid, false, 0, depthLimit, -1000, 1000);
                    grid[i][j] = EMPTY;

                    if (score > bestScore) {
                        bestScore = score;
                        bestRow = i;
                        bestCol = j;
                    }
```

```
                }
            }
        }
    }

    // Ensure a move is made
    if (bestRow != -1 && bestCol != -1) {
        grid[bestRow][bestCol] = PLAYER_O;
    }

    // Get the current mode's stats
    ModeStats* currentStats = GetCurrentModeStats();

    if (CheckWin(PLAYER_O)) {
        gameOver = true;
        winner = PLAYER_O;
        gameState = GAME_OVER;

        // Update the stats counter
        currentStats->aiWins++;
        currentStats->totalGames++;

        // Play sound immediately when a winner is detected
        if (!isTwoPlayer) {
            PlaySound(loseSound);  // Play lose sound for Player O
        } else {
                PlaySound(victorySound);  // Play victory sound for any winner in
two-player mode
        }

    }
    else if (CheckDraw()) {
        gameOver = true;
        gameState = GAME_OVER;
        winner = EMPTY;

        // Update the stats counter
        currentStats->draws++;
        currentStats->totalGames++;

        PlaySound(drawSound);  // Play draw sound
    }
    else {
        currentPlayerTurn = PLAYER_X_TURN;
    }
}

// Decision Tree AI Turn function
void AITurnDecisionTree(Sound victorySound, Sound loseSound, Sound drawSound,
DecisionTreeNode *TDmodel) {
    int bestScore = -1000;       // Initialize best score for evaluating moves
    int bestRow = -1;            // Initialize best row for AI move
    int bestCol = -1;            // Initialize best column for AI move
```

```
    int row, col;                   // Variables for random fallback move
    double best_prob = 0.0;         // Probability of the best move
    char board[3][3];               // Buffer array for board layout

     // Convert the current grid state into a compatible format for the decision
tree
    for (int i = 0; i < 3; i++) {
        for (int j = 0; j < 3; j++) {
            if (grid[i][j] == EMPTY) {
                board[i][j] = 'b';  // Convert EMPTY cells to 'b' (blank)
            } else if (grid[i][j] == PLAYER_X) {
                board[i][j] = 'x';  // Convert PLAYER_X cells to 'x'
            } else if (grid[i][j] == PLAYER_O) {
                board[i][j] = 'o';  // Convert PLAYER_O cells to 'o'
            }
        }
    }

    // Print the decision tree structure for debugging
    print_tree(TDmodel, 2);

    // Use the decision tree to predict the best move for the AI
    dt_predict_best_move(TDmodel, board, PLAYER_O, &bestRow, &bestCol);

    // Fallback logic: Choose a random empty cell if the decision tree fails
    do {
        row = GetRandomValue(0, GRID_SIZE - 1);  // Generate random row index
        col = GetRandomValue(0, GRID_SIZE - 1);  // Generate random column index
    } while (grid[row][col] != EMPTY);           // Ensure the chosen cell is empty

    // Place the AI's move on the grid at the predicted or random position
    grid[bestRow][bestCol] = PLAYER_O;

    // Get the current stats for Easy mode
    ModeStats* currentStats = &decisionTreeStats;

    // Check if the AI's move results in a win
    if (CheckWin(PLAYER_O)) {
        gameOver = true;                // Mark the game as over
        winner = PLAYER_O;              // Set the winner to PLAYER_O
        gameState = GAME_OVER;          // Transition to GAME_OVER state
        currentStats->aiWins++;         // Increment AI win count
        currentStats->totalGames++;     // Increment total games count

        PlaySound(loseSound);           // Play losing sound for the player
    }
    // Check if the game results in a draw
    else if (CheckDraw()) {
        gameOver = true;                // Mark the game as over
        gameState = GAME_OVER;          // Transition to GAME_OVER state
        winner = EMPTY;                 // Set the winner to NONE (draw)
        currentStats->draws++;          // Increment draw count
        currentStats->totalGames++;     // Increment total games count
```

```
        PlaySound(drawSound);           // Play draw sound
    }
    // If the game continues, pass the turn to the player
    else {
        currentPlayerTurn = PLAYER_X_TURN; // Set the turn to PLAYER_X
    }
}
```

12.3. Check.c

```
#include "main.h"

// Declare external variables used in Check.c
extern Cell grid[GRID_SIZE][GRID_SIZE];
extern int winningCells[3][2];

// Function to check if a player has won
bool CheckWin(Cell player) {
    // Check rows
    for (int i = 0; i < GRID_SIZE; i++) {
        if (grid[i][0] == player && grid[i][1] == player && grid[i][2] == player) {
            winningCells[0][0] = i; winningCells[0][1] = 0;
            winningCells[1][0] = i; winningCells[1][1] = 1;
            winningCells[2][0] = i; winningCells[2][1] = 2;
            return true;
        }
    }

    // Check columns
    for (int i = 0; i < GRID_SIZE; i++) {
        if (grid[0][i] == player && grid[1][i] == player && grid[2][i] == player) {
            winningCells[0][0] = 0; winningCells[0][1] = i;
            winningCells[1][0] = 1; winningCells[1][1] = i;
            winningCells[2][0] = 2; winningCells[2][1] = i;
            return true;
        }
    }

    // Check main diagonal
    if (grid[0][0] == player && grid[1][1] == player && grid[2][2] == player) {
        winningCells[0][0] = 0; winningCells[0][1] = 0;
        winningCells[1][0] = 1; winningCells[1][1] = 1;
        winningCells[2][0] = 2; winningCells[2][1] = 2;
        return true;
    }

    // Check anti diagonal
    if (grid[0][2] == player && grid[1][1] == player && grid[2][0] == player) {
        winningCells[0][0] = 0; winningCells[0][1] = 2;
        winningCells[1][0] = 1; winningCells[1][1] = 1;
        winningCells[2][0] = 2; winningCells[2][1] = 0;
        return true;
    }

    return false;
}

// Function to check if the game is a draw
bool CheckDraw() {
    for (int i = 0; i < GRID_SIZE; i++) {
        for (int j = 0; j < GRID_SIZE; j++) {
```

```
                if (grid[i][j] == EMPTY) return false; // If there's an empty cell, it's not a draw
        }
    }
    return true; // All cells are filled
}
```

12.4. Draw.c

```c
#include "main.h"
extern Confetti confetti[MAX_CONFETTI];
extern bool showPartyAnimation;
extern bool gameOver;
extern Cell grid[GRID_SIZE][GRID_SIZE];
extern int winningCells[3][2];
extern struct GetHint hint;
extern bool isTwoPlayer;
extern Cell winner;
extern ModeStats* currentStats;
extern PlayerTurn currentPlayerTurn;

// Draw the confetti
void DrawConfetti() {
    for (int i = 0; i < MAX_CONFETTI; i++) {
        if (confetti[i].active) {
            Color particleColor = confetti[i].color;
            particleColor.a = (unsigned char)(confetti[i].alpha * 255);

            // Longer trails for more visible effect
            Vector2 direction = {
                -confetti[i].velocity.x * 0.15f,  // Increased from 0.1f
                -confetti[i].velocity.y * 0.15f   // Increased from 0.1f
            };

            // Draw main particle
            DrawCircle(
                confetti[i].position.x,
                confetti[i].position.y,
                confetti[i].size,
                particleColor
            );

            // Longer trails with more segments
             for (int trail = 1; trail <= 7; trail++) {  // Increased from 5 to 7
segments
                 float trailAlpha = confetti[i].alpha * (1.0f - (trail * 0.14f));
// Adjusted fade
                Vector2 trailPos = {
                    confetti[i].position.x + direction.x * trail,
                    confetti[i].position.y + direction.y * trail
                };

                DrawCircle(
                    trailPos.x,
                    trailPos.y,
                     confetti[i].size * (1.0f - (trail * 0.12f)),  // Adjusted size
reduction
                    ColorAlpha(particleColor, trailAlpha * 255)
                );
            }
```

```
        }
    }
}

// Drawing of the Game Designs
// Draw the title words
void DrawTitleWords() {
    for (int i = 0; i < 5; i++) {
                            DrawText(titleWords[i].word,    titleWords[i].position.x,
titleWords[i].position.y, 40, BLACK);
    }
}

// Draw the symbols
void DrawSymbols() {
    for (int i = 0; i < MAX_SYMBOLS; i++) {
        Vector2 origin = {10, 10};  // Center of rotation
            DrawTextPro(GetFontDefault(), &symbols[i].symbol, symbols[i].position,
origin, symbols[i].rotation, 20, 1, symbols[i].symbol == 'X' ? BLUE : RED);
    }
}

// Draw the game
void DrawGame() {
    bool isHintHovered = false;
    Vector2 mousePos = GetMousePosition();

    // The grid and pieces
    for (int i = 0; i < GRID_SIZE; i++)
    {
        for (int j = 0; j < GRID_SIZE; j++)
        {
                Rectangle cell = {(float)(j * CELL_SIZE), (float)(i * CELL_SIZE),
(float)CELL_SIZE, (float)CELL_SIZE};

            // Check if this cell is part of the winning combination
            bool isWinningCell = false;
            if (gameOver && winner != EMPTY) {
                for (int k = 0; k < 3; k++) {
                    if (winningCells[k][0] == i && winningCells[k][1] == j) {
                        isWinningCell = true;
                        break;
                    }
                }
            }

            // Check if the mouse is hovering over the cell and the cell is empty
                        bool  isHovered  =  !gameOver  &&  grid[i][j]  ==  EMPTY  &&
CheckCollisionPointRec(mousePos, cell);

            // Draw the cell with appropriate color
            Color cellColor;
            if (isWinningCell) {
```

```
                if (!isTwoPlayer && winner == PLAYER_O) {
                         cellColor = (Color){ 255, 200, 200, 255 }; // Light red
highlight for AI wins
                } else {
                         cellColor = (Color){ 144, 238, 144, 255 }; // Light green
highlight for player wins
                }
            } else {
                cellColor = isHovered ? DARKGRAY : LIGHTGRAY;
            }

            DrawRectangleRec(cell, cellColor);

            if (grid[i][j] == PLAYER_X)
            {
                const char* text = "X";
                float fontSize = 100;
                float textWidth = MeasureText(text, fontSize);
                  float textHeight = fontSize * 0.75f; // Approximate height of the
text
                float textX = cell.x + (CELL_SIZE - textWidth) / 2;
                float textY = cell.y + (CELL_SIZE - textHeight) / 2;
                DrawText(text, textX, textY, fontSize, BLUE);
            }
            else if (grid[i][j] == PLAYER_O)
            {
                const char* text = "O";
                float fontSize = 100;
                float textWidth = MeasureText(text, fontSize);
                  float textHeight = fontSize * 0.75f; // Approximate height of the
text
                float textX = cell.x + (CELL_SIZE - textWidth) / 2;
                float textY = cell.y + (CELL_SIZE - textHeight) / 2;
                DrawText(text, textX, textY, fontSize, RED);
            }
        }
    }

    // Grid lines
    for (int i = 1; i < GRID_SIZE; i++)
    {
        DrawLine(i * CELL_SIZE, 0, i * CELL_SIZE, SCREEN_HEIGHT, BLACK);
        DrawLine(0, i * CELL_SIZE, SCREEN_WIDTH, i * CELL_SIZE, BLACK);
    }

    // Hint button position
    Rectangle hintBtn = {
        SCREEN_WIDTH - 80, 10,  // moved to top right
        70, 30
    };
    // Hint button counts left for player
    const char *hintText = "Hint: ";
    char hintTextFinal[10];
```

```
    // hintCount for player X
        snprintf(hintTextFinal,  sizeof(hintTextFinal),  "%s%d",  hintText,  (2  -
hint.hintCountX)); // hint button text
    if (currentPlayerTurn==PLAYER_X_TURN){
        if (hint.hintCountX < 2) // hint button active when count < 2
        {
                isHintHovered = (mousePos.x >= SCREEN_WIDTH - 80 && mousePos.x <=
SCREEN_WIDTH - 10 && mousePos.y >= 10 && mousePos.y <= 40);
            DrawButton(hintBtn, hintTextFinal, 20, !gameOver && isHintHovered);
        } else
        {
            DrawButton(hintBtn, hintTextFinal, 20, !gameOver && false);
        }
    }
    // hintCount for player O
        snprintf(hintTextFinal,  sizeof(hintTextFinal),  "%s%d",  hintText,  (2  -
hint.hintCountO)); // hint button text
    if (currentPlayerTurn==PLAYER_O_TURN)
    {
        if (hint.hintCountO < 2) // hint button active when count < 2
        {
                isHintHovered = (mousePos.x >= SCREEN_WIDTH - 80 && mousePos.x <=
SCREEN_WIDTH - 10 && mousePos.y >= 10 && mousePos.y <= 40);
            DrawButton(hintBtn, hintTextFinal, 20, !gameOver && isHintHovered);
        } else
        {
            DrawButton(hintBtn, hintTextFinal, 20, !gameOver && false);
        }
    }
    // Quit button position
    Rectangle quitBtn = {
        20, 10,  // moved to top left
        70, 30
    };
     bool isQuitHovered = (mousePos.x >= 20 && mousePos.x <= 90 && mousePos.y >= 10
&& mousePos.y <= 40);
    DrawButton(quitBtn, "Quit", 20, !gameOver && isQuitHovered);

    // Only set cursor for button if we're not in game over state
    if (!gameOver && isQuitHovered) {
        SetMouseCursor(MOUSE_CURSOR_POINTING_HAND);
    } else if (!gameOver && isHintHovered) {
        SetMouseCursor(MOUSE_CURSOR_POINTING_HAND);
    } else if (!gameOver) {
        SetMouseCursor(MOUSE_CURSOR_DEFAULT);
    }

    if (!gameOver) {
        // only display stats for single player mode
        if (!isTwoPlayer) {
            char statsText[100];
            ModeStats* currentStats = GetCurrentModeStats();
```

```
            sprintf(statsText, "Player: %d | AI: %d | Draws: %d",
                    currentStats->playerWins,
                    currentStats->aiWins,
                    currentStats->draws);

            // draw stats in middle above turn display
             DrawText(statsText, SCREEN_WIDTH/2 - MeasureText(statsText, 20)/2, 10,
20, BLACK);
        }

        // turn display indicator
        int yPos = isTwoPlayer ? 20 : 40;  // shift up for 2 player mode

        if (currentPlayerTurn == PLAYER_X_TURN) {
            const char* turnText = isTwoPlayer ? "Player X's Turn" : "Your Turn";
             DrawText(turnText, SCREEN_WIDTH/2 - MeasureText(turnText, 30)/2, yPos,
30, BLUE);
        } else {
            const char* turnText = isTwoPlayer ? "Player O's Turn" : "AI's Turn";
             DrawText(turnText, SCREEN_WIDTH/2 - MeasureText(turnText, 30)/2, yPos,
30, RED);
        }
    }
}

// Draw the menu
void DrawMenu() {
    const int titleFontSize = 40;
    const int buttonFontSize = 20;
    const int cellSize = 50;  // larger cells for better visibility
    const int gridWidth = TITLE_GRID_SIZE * cellSize;
    const int gridHeight = TITLE_GRID_SIZE * cellSize;
    const int startX = SCREEN_WIDTH/2 - gridWidth/2;
    const int startY = SCREEN_HEIGHT/5;

    // Cell animations
    for(int i = 0; i < TITLE_GRID_SIZE; i++) {
        for(int j = 0; j < TITLE_GRID_SIZE; j++) {
            Rectangle cell = {
                startX + j * cellSize,
                startY + i * cellSize,
                cellSize,
                cellSize
            };

            // Draw just the grid lines
            DrawRectangleLinesEx(cell, 2, BLACK);

            // Handle the X and O symbols
            if (!titleSymbols[i][j].active && GetRandomValue(0, 100) < 2) {
                titleSymbols[i][j].symbol = GetRandomValue(0, 1) ? 'X' : 'O';
                titleSymbols[i][j].alpha = 0; // reset to transparent
                titleSymbols[i][j].active = true;
```

```
            }

            if (titleSymbols[i][j].active) {
                titleSymbols[i][j].alpha += GetFrameTime() * 2;
                if (titleSymbols[i][j].alpha > 1.0f) {
                    titleSymbols[i][j].alpha = 0; // reset to transparent
                    titleSymbols[i][j].active = false;
                }

                Color symbolColor = titleSymbols[i][j].symbol == 'X' ? BLUE : RED;
                symbolColor.a = (unsigned char)(titleSymbols[i][j].alpha * 255);

                Vector2 textPos = {
                      cell.x + (cellSize - MeasureText(&titleSymbols[i][j].symbol,
40))/2,
                    cell.y + (cellSize - 40)/2
                };
                    DrawText(&titleSymbols[i][j].symbol, textPos.x, textPos.y, 40,
symbolColor);
            }
        }
    }

    // Button rectangles
    Rectangle singlePlayerBtn = {
        SCREEN_WIDTH/2 - BUTTON_WIDTH/2,
        SCREEN_HEIGHT/2 + BUTTON_HEIGHT + 20,
        BUTTON_WIDTH,
        BUTTON_HEIGHT
    };

    Rectangle twoPlayerBtn = {
        SCREEN_WIDTH/2 - BUTTON_WIDTH/2,
        SCREEN_HEIGHT/2 + (BUTTON_HEIGHT + 20) * 2,
        BUTTON_WIDTH,
        BUTTON_HEIGHT
    };

    Rectangle exitBtn = {
        SCREEN_WIDTH/2 - BUTTON_WIDTH/2,
        SCREEN_HEIGHT/2 + (BUTTON_HEIGHT + 20) * 3,
        BUTTON_WIDTH,
        BUTTON_HEIGHT
    };

    // Check hover states
    bool singlePlayerHover = false;
    bool twoPlayerHover = false;
    bool exitHover = false;

           HandleButtonHover(singlePlayerBtn,   "Single   Player",   buttonFontSize,
&singlePlayerHover);
```

```
              HandleButtonHover(twoPlayerBtn,    "Two    Players",    buttonFontSize,
&twoPlayerHover);
    HandleButtonHover(exitBtn, "Exit", buttonFontSize, &exitHover);

    // Set cursor based on any button hover
    SetMouseCursor((singlePlayerHover || twoPlayerHover || exitHover) ?
        MOUSE_CURSOR_POINTING_HAND : MOUSE_CURSOR_DEFAULT);
}

// Draw the game over screen
void DrawGameOver() {
    const int titleFontSize = 40;
    const int buttonFontSize = 20;

    // Draw semi-transparent overlay
    DrawRectangle(0, 0, SCREEN_WIDTH, SCREEN_HEIGHT, (Color){0, 0, 0, 100});

    // Result Text
    const char* resultText;
    Color resultColor;

    if (winner == PLAYER_X) {
        resultText = isTwoPlayer ? "Player X Wins!" : "You win!";
        resultColor = BLUE;
    } else if (winner == PLAYER_O) {
        resultText = isTwoPlayer ? "Player O Wins!" : "You lose!";
        resultColor = RED;
    } else {
        resultText = "It's a Draw!";
        resultColor = DARKGRAY;
    }

    // Draw result text with background
    int textWidth = MeasureText(resultText, titleFontSize);
    DrawRectangle(
        SCREEN_WIDTH/2 - textWidth/2 - 10,
        SCREEN_HEIGHT/3 - 10,
        textWidth + 20,
        titleFontSize + 20,
        WHITE
    );
    DrawText(resultText,
        SCREEN_WIDTH/2 - textWidth/2,
        SCREEN_HEIGHT/3,
        titleFontSize,
        resultColor
    );

    // Retry Button
    Rectangle retryBtn = {
        SCREEN_WIDTH/2 - BUTTON_WIDTH/2,
        SCREEN_HEIGHT/2 + 40, // Position above the menu button
        BUTTON_WIDTH,
```

```
        BUTTON_HEIGHT
    };

    // Back to Menu Button
    Rectangle menuBtn = {
        SCREEN_WIDTH/2 - BUTTON_WIDTH/2,
        SCREEN_HEIGHT/2 + 100,  // Position below the retry button
        BUTTON_WIDTH,
        BUTTON_HEIGHT
    };

    Vector2 mousePos = GetMousePosition();
    bool isMenuHovered = CheckCollisionPointRec(mousePos, menuBtn);
    bool isRetryHovered = CheckCollisionPointRec(mousePos, retryBtn);

    // Draw buttons with hover effect
    DrawButton(retryBtn, "Retry", buttonFontSize, isRetryHovered);
    DrawButton(menuBtn, "Back to Menu", buttonFontSize, isMenuHovered);

    // Set cursor
    SetMouseCursor((isMenuHovered || isRetryHovered) ? MOUSE_CURSOR_POINTING_HAND :
MOUSE_CURSOR_DEFAULT);
}

// draw the buttons with hover effect
void DrawButton(Rectangle bounds, const char* text, int fontSize, bool isHovered) {
    Rectangle vibrationBounds = bounds;

    // Apply vibration effect to all buttons when hovered
    if (isHovered) {
        buttonVibrationOffset = sinf(GetTime() * vibrationSpeed) * vibrationAmount;
        vibrationBounds.x += buttonVibrationOffset;
    }

    // Draw the button background
     DrawRectangleRec(vibrationBounds, isHovered ? GRAY : LIGHTGRAY); // Draw the
button background with a gray color if hovered

    // Draw the button outline
     DrawRectangleLinesEx(vibrationBounds, 2, BLACK); // Draw the button outline
with a black color

    // Draw the button text
    DrawText(text,
             vibrationBounds.x + (vibrationBounds.width - MeasureText(text,
fontSize))/2, // Center the text horizontally
          vibrationBounds.y + (vibrationBounds.height - fontSize)/2, // Center the
text vertically
        fontSize,
        BLACK
    );
}
```

```
// Draw the difficulty selection screen
void DrawDifficultySelect() {
    const int titleFontSize = 40;
    const int buttonFontSize = 20;

    // Title
    const char* title = "Select Difficulty";
    DrawText(title,
        SCREEN_WIDTH/2 - MeasureText(title, titleFontSize)/2,
        SCREEN_HEIGHT/3,
        titleFontSize,
        BLACK);

    // Button rectangles
    Rectangle easyBtn = {
        SCREEN_WIDTH/2 - BUTTON_WIDTH/2,
        SCREEN_HEIGHT/2,
        BUTTON_WIDTH,
        BUTTON_HEIGHT
    };

    Rectangle mediumBtn = {
        SCREEN_WIDTH/2 - BUTTON_WIDTH/2,
        SCREEN_HEIGHT/2 + BUTTON_HEIGHT + 20,
        BUTTON_WIDTH,
        BUTTON_HEIGHT
    };

    Rectangle hardBtn = {
        SCREEN_WIDTH/2 - BUTTON_WIDTH/2,
        SCREEN_HEIGHT/2 + (BUTTON_HEIGHT + 20) * 2,
        BUTTON_WIDTH,
        BUTTON_HEIGHT
    };

    // Add back button at top left
    Rectangle backBtn = {
        20,                 // Left margin
        10,                 // Top margin
        SCREEN_WIDTH/6,     // Width (100px at 600px screen width)
        30                  // Height
    };

    // Check hover states
    bool easyHover = false;
    bool mediumHover = false;
    bool hardHover = false;
    bool backHover = false;

    // Draw buttons with hover effects
    HandleButtonHover(easyBtn, "Easy", buttonFontSize, &easyHover);
    HandleButtonHover(mediumBtn, "Medium", buttonFontSize, &mediumHover);
    HandleButtonHover(hardBtn, "Hard", buttonFontSize, &hardHover);
```

```
    HandleButtonHover(backBtn, "Back", buttonFontSize, &backHover);

    // Set cursor based on any button hover
    SetMouseCursor((easyHover || mediumHover || hardHover || backHover) ?
        MOUSE_CURSOR_POINTING_HAND : MOUSE_CURSOR_DEFAULT);
}

// Draw the AI model selection screen
void DrawModelSelect() {
    const char* title = "Select AI Model";
    DrawText(title,
        SCREEN_WIDTH/2 - MeasureText(title, 40)/2,
        SCREEN_HEIGHT/3,
        40,
        BLACK);

    Rectangle nbBtn = {
        SCREEN_WIDTH/2 - BUTTON_WIDTH/2,
        SCREEN_HEIGHT/2,
        BUTTON_WIDTH,
        BUTTON_HEIGHT
    };

    Rectangle dtBtn = {
        SCREEN_WIDTH/2 - BUTTON_WIDTH/2,
        SCREEN_HEIGHT/2 + BUTTON_HEIGHT + 20,
        BUTTON_WIDTH,
        BUTTON_HEIGHT
    };

    // back button at top left
    Rectangle backBtn = {
        20,                 // Left margin
        10,                 // Top margin
        SCREEN_WIDTH/6,     // Width (100px at 600px screen width)
        30                  // Height
    };

    // Check hover states
    bool nbHover = false;
    bool dtHover = false;
    bool backHover = false;

    // Draw buttons with hover effects
    HandleButtonHover(nbBtn, "Naive Bayes", 20, &nbHover);
    HandleButtonHover(dtBtn, "Decision Tree", 20, &dtHover);
    HandleButtonHover(backBtn, "Back", 20, &backHover);

    // Set cursor based on any button hover
    SetMouseCursor((nbHover || dtHover || backHover) ? MOUSE_CURSOR_POINTING_HAND :
MOUSE_CURSOR_DEFAULT);
}
```

12.5. Handle.c

```
#include "main.h"
extern Cell grid[GRID_SIZE][GRID_SIZE];
extern struct GetHint hint;

// Handle the button hover
bool  HandleButtonHover(Rectangle  button,  const  char*  text,  int  fontSize,  bool*
isHovered) {
    Vector2 mousePos = GetMousePosition();
    *isHovered = CheckCollisionPointRec(mousePos, button);
    DrawButton(button, text, fontSize, *isHovered);
    return *isHovered;
}

// Handle the player's turn
bool  HandlePlayerTurn(Sound  popSound,  Sound  victorySound,  Sound  loseSound,  Sound
drawSound)
{
    clearHint();
    if (IsMouseButtonPressed(MOUSE_LEFT_BUTTON))
    {
        Vector2 mousePos = GetMousePosition();
        int row = (int)(mousePos.y / CELL_SIZE);
        int col = (int)(mousePos.x / CELL_SIZE);

          // when hint button is clicked, get best move for the player and update
hintCount. If hintCount is 2, button doesnt work
        if (mousePos.x >= SCREEN_WIDTH - 80 && mousePos.x <= SCREEN_WIDTH - 10 &&
                mousePos.y  >=  10  &&  mousePos.y  <=  40  &&  (hint.hintCountX  <  2  ||
hint.hintCountO < 2))
        {
                // Get player turn and update the hint count when hint button is
clicked
            if (currentPlayerTurn == PLAYER_X_TURN && hint.hintCountX < 2)
            {
                PlaySound(buttonClickSound);
                hint.hintCountX+=1; // increment
                getHint(); // Get best move
                row = hint.row; // assign best move to be picked
                col = hint.col; // assign best move to be picked
            } else if (currentPlayerTurn == PLAYER_O_TURN && hint.hintCountO < 2)
            {
                PlaySound(buttonClickSound);
                hint.hintCountO+=1; // increment
                getHint(); // Get best move
                row = hint.row; // assign best move to be picked
                col = hint.col; // assign best move to be picked
            } else {
                return false; // No move was made
            }
        }
```

```
        // When updating stats, use the current mode's counter:
        ModeStats* currentStats = GetCurrentModeStats();

        // check win after a grid is selected
        if (row >= 0 && row < GRID_SIZE && col >= 0 && col < GRID_SIZE)
        {
            if (grid[row][col] == EMPTY)
            {
                 grid[row][col] = (currentPlayerTurn == PLAYER_X_TURN) ? PLAYER_X :
PLAYER_O;
                if (CheckWin(grid[row][col]))
                {
                    gameOver = true;
                    winner = grid[row][col];
                    gameState = GAME_OVER;

                    // Play sound immediately when a winner is detected
                    if (isTwoPlayer) {
                         showPartyAnimation = true; // Show confetti for any winner
in two-player mode
                        InitConfetti(); // trigger confetti animation
                            PlaySound(victorySound); // Play victory sound for any
winner in two-player mode
                    }
                    else if (!isTwoPlayer && winner == PLAYER_X) {
                          showPartyAnimation = true;  // Show party animation only
when human player wins
                        InitConfetti(); // trigger confetti animation
                        currentStats->playerWins++; // Increment player wins
                        currentStats->totalGames++; // Increment total games
                        PlaySound(victorySound); // Play victory sound for Player X
                    }
                    else {
                        showPartyAnimation = false; // No confetti for AI wins
                        currentStats->aiWins++; // Increment AI wins
                        currentStats->totalGames++; // Increment total games
                        PlaySound(loseSound); // Play lose sound for Player O
                    }
                }
                else if (CheckDraw()) { // Check for a draw
                    gameOver = true;
                    gameState = GAME_OVER;
                    winner = EMPTY;  // No winner in a draw
                    currentStats->draws++;  // Increment draws scores
                    currentStats->totalGames++; // Increment total games
                    PlaySound(drawSound);  // Play draw sound
                }
                else {
                         currentPlayerTurn = (currentPlayerTurn == PLAYER_X_TURN) ?
PLAYER_O_TURN : PLAYER_X_TURN; // change player turn
                }
                return true;  // Move was made
            }
```

```
        }
    }
    return false;  // No move was made
}
```

12.6. Hint.c

```c
#include "main.h"
// Declare external variables used in Hint.c
extern struct GetHint hint;
extern Cell grid[GRID_SIZE][GRID_SIZE];

// Clear Hint best move
void clearHint() {
    hint.row = -1;
    hint.col = -1;
}
// Get Hint
void getHint() {
    int bestScore = -1000;
    int bestRow = -1;
    int bestCol = -1;
    int depthLimit = 9; // Full depth
    for (int i = 0; i < GRID_SIZE; i++) {
        for (int j = 0; j < GRID_SIZE; j++) {
            if (grid[i][j] == EMPTY) {
                grid[i][j] = PLAYER_O;
                int score = Minimax(grid, false, 0, depthLimit, -1000, 1000);
                grid[i][j] = EMPTY;

                if (score > bestScore) {
                    bestScore = score;
                    bestRow = i;
                    bestCol = j;
                }
            }
        }
    }
    // save best move
    if (bestRow != -1 && bestCol != -1) {
        hint.row = bestRow;
        hint.col = bestCol;
    }
}
```

12.7. Init.c

```
#include "main.h"
extern struct GetHint hint;
extern Cell grid[GRID_SIZE][GRID_SIZE];
extern int winningCells[3][2];
extern bool showPartyAnimation;
extern bool gameOver;
extern Cell winner;

// Initialize the title words
void InitTitleWords() {
    const char* words[] = {"Tic", "-", "Tac", "-", "Toe"};
    int startX = SCREEN_WIDTH / 2 - MeasureText("Tic-Tac-Toe", 40) / 2;
    int startY = SCREEN_HEIGHT / 5 + TITLE_GRID_SIZE * 50 + 20;
    int spacing = 10;  // Space between words and hyphens

    for (int i = 0; i < 5; i++) {
        titleWords[i].word = words[i];
        titleWords[i].position = (Vector2){ startX, startY };
        titleWords[i].targetPosition = (Vector2){ startX, startY - 20 };
        titleWords[i].isJumping = false;
        titleWords[i].jumpSpeed = JUMP_SPEED;
        startX += MeasureText(words[i], 40) + spacing;
    }
}

// Initialize the symbols
void InitSymbols() {
    for (int i = 0; i < MAX_SYMBOLS; i++) {
              symbols[i].position  =  (Vector2){  GetRandomValue(0,  SCREEN_WIDTH),
GetRandomValue(-SCREEN_HEIGHT, 0) };
        symbols[i].symbol = GetRandomValue(0, 1) ? 'X' : 'O';
        symbols[i].rotation = GetRandomValue(0, 360);  // Random initial rotation
    }
}

// Initialize the confetti
void InitConfetti() {
    for (int i = 0; i < MAX_CONFETTI; i++) {
        // Start all particles from bottom right corner with some variation
        confetti[i].position = (Vector2){
               SCREEN_WIDTH - GetRandomValue(30, 70),   // More variation in start
position
            SCREEN_HEIGHT - GetRandomValue(30, 70)
        };

        // Wider spray pattern (160° to 280° for almost full semicircle)
        float angle = GetRandomValue(160, 280) * DEG2RAD;  // Increased angle range
        float speed = GetRandomValue(600, 1200) / 100.0f;  // Increased speed range
        confetti[i].velocity = (Vector2){
            cos(angle) * speed,
            sin(angle) * speed
```

```c
        };

        // Festive colors for party popper
        switch(GetRandomValue(0, 4)) {
            case 0: confetti[i].color = (Color){255, 50, 50, 255};   // Red
                break;
            case 1: confetti[i].color = (Color){50, 255, 50, 255};   // Green
                break;
            case 2: confetti[i].color = (Color){50, 50, 255, 255};   // Blue
                break;
            case 3: confetti[i].color = (Color){255, 255, 50, 255};  // Yellow
                break;
            case 4: confetti[i].color = (Color){255, 50, 255, 255};  // Pink
                break;
        }

        confetti[i].size = GetRandomValue(2, 4);
        confetti[i].active = true;
        confetti[i].alpha = 1.0f;
        confetti[i].lifetime = GetRandomValue(150, 200) / 100.0f;
    }
}

// initialize the game
void InitGame() {
    // resets hintCount when retry
    hint.hintCountO = 0;
    hint.hintCountX = 0;
    showPartyAnimation = false; // Reset party animation

    // Stop all sounds that might be playing
    StopSound(victorySound);
    StopSound(loseSound);
    StopSound(drawSound);

    // Initialize the grid to EMPTY in a single loop
    memset(grid, EMPTY, sizeof(grid));
    gameOver = false;
    winner = EMPTY;

    // Randomize starting player for both single and two player modes
    RandomizeStartingPlayer();


    // Reset winning cells
    for (int i = 0; i < 3; i++) {
        winningCells[i][0] = -1;
        winningCells[i][1] = -1;
    }
}
```

12.8. Minimax.c

```
#include "main.h"

// minimax algorithm, recursive design
int Minimax(Cell board[GRID_SIZE][GRID_SIZE], bool isMaximizing, int depth, int
depthLimit, int alpha, int beta) {
    if (depth >= depthLimit) return 0; // Return 0 if depth limit is reached
    int score = EvaluateBoard(board);
    if (score == 10) return score - depth; // O (AI) is the maximizing player
    if (score == -10) return score + depth; // X (human) is the minimizing player
    if (CheckDraw()) return 0; // Draw

    if (isMaximizing) {
        int bestScore = -1000; // Initialize the best score to a very low value
        for (int i = 0; i < GRID_SIZE; i++) {
             for (int j = 0; j < GRID_SIZE; j++) { // Iterate through each cell in
the grid
                if (board[i][j] == EMPTY) { // If the cell is empty
                    board[i][j] = PLAYER_O; // Set the cell to PLAYER_O
                      bestScore = fmax(bestScore, Minimax(board, false, depth + 1,
depthLimit, alpha, beta)); // Update the best score
                    board[i][j] = EMPTY; // Reset the cell to EMPTY
                    alpha = fmax(alpha, bestScore); // Update alpha (maximize)
                      if (beta <= alpha) break; // Beta cut-off (prune the branch,
stopping the recursion)

                }
            }
        }
        return bestScore;
    } else {
        int bestScore = 1000; // Initialize the best score to a very high value
        for (int i = 0; i < GRID_SIZE; i++) {
             for (int j = 0; j < GRID_SIZE; j++) { // Iterate through each cell in
the grid
                if (board[i][j] == EMPTY) { // If the cell is empty
                    board[i][j] = PLAYER_X; // Set the cell to PLAYER_X
                      bestScore = fmin(bestScore, Minimax(board, true, depth + 1,
depthLimit, alpha, beta)); // Update the best score
                    board[i][j] = EMPTY; // Reset the cell to EMPTY
                    beta = fmin(beta, bestScore); // Update beta (minimize)
                      if (beta <= alpha) break; // Alpha cut-off (prune the branch,
stopping the recursion)
                }
            }
        }
        return bestScore; // Return the best score
    }
}

// Evaluate the board
int EvaluateBoard(Cell board[GRID_SIZE][GRID_SIZE]) {
```

```
    // Check rows and columns for a win
    for (int row = 0; row < GRID_SIZE; row++) {
        if (board[row][0] == board[row][1] && board[row][0] == board[row][2]) {
            if (board[row][0] == PLAYER_O) return 10;
            else if (board[row][0] == PLAYER_X) return -10;
        }
    }
    for (int col = 0; col < GRID_SIZE; col++) {
        if (board[0][col] == board[1][col] && board[0][col] == board[2][col]) {
            if (board[0][col] == PLAYER_O) return 10;
            else if (board[0][col] == PLAYER_X) return -10;
        }
    }
    // Check diagonals for a win
    if (board[0][0] == board[1][1] && board[0][0] == board[2][2]) {
        if (board[0][0] == PLAYER_O) return 10;
        else if (board[0][0] == PLAYER_X) return -10;
    }
    if (board[0][2] == board[1][1] && board[0][2] == board[2][0]) {
        if (board[0][2] == PLAYER_O) return 10;
        else if (board[0][2] == PLAYER_X) return -10;
    }

    return 0; // No winner
}
```

12.9. Update.c

```
#include "main.h"
extern TitleWord titleWords[5];
extern FallingSymbol symbols[MAX_SYMBOLS];
extern Confetti confetti[MAX_CONFETTI];
extern bool showPartyAnimation;
extern bool gameOver;
extern bool allInactive;
extern int currentWord;
extern int currentModel;
extern Difficulty currentDifficulty;
extern PlayerTurn currentPlayerTurn;
extern bool isTwoPlayer;

// Update the title words
void UpdateTitleWords() {
    static int currentWord = 0;
    static float jumpDelay = 0.0f;

    jumpDelay += GetFrameTime();
    if (jumpDelay > JUMP_DELAY) {  // Delay between each word's jump
        if (!titleWords[currentWord].isJumping) {
            titleWords[currentWord].isJumping = true;
            jumpDelay = 0.0f;
        }
    }

    for (int i = 0; i < 5; i++) {
        if (titleWords[i].isJumping) {
            titleWords[i].position.y -= titleWords[i].jumpSpeed;
            if (titleWords[i].position.y <= titleWords[i].targetPosition.y) {
                   titleWords[i].jumpSpeed = -titleWords[i].jumpSpeed;   // Reverse
direction
            }
             if (titleWords[i].position.y >= SCREEN_HEIGHT / 5 + TITLE_GRID_SIZE *
50 + 20) {
                titleWords[i].position.y = SCREEN_HEIGHT / 5 + TITLE_GRID_SIZE * 50
+ 20;
                titleWords[i].isJumping = false;
                titleWords[i].jumpSpeed = JUMP_SPEED;
                currentWord = (currentWord + 1) % 5;   // Move to the next word
            }
        }
    }
}

// Update the symbols
void UpdateSymbols() {
    for (int i = 0; i < MAX_SYMBOLS; i++) {
        symbols[i].position.y += SYMBOL_SPEED;
        symbols[i].rotation += ROTATION_SPEED;   // Update rotation
        if (symbols[i].position.y > SCREEN_HEIGHT) {
```

```
            symbols[i].position.y = GetRandomValue(-SCREEN_HEIGHT, 0);
            symbols[i].position.x = GetRandomValue(0, SCREEN_WIDTH);
            symbols[i].symbol = GetRandomValue(0, 1) ? 'X' : 'O';
            symbols[i].rotation = GetRandomValue(0, 360);  // Reset rotation
        }
    }
}

// Update the confetti animation
void UpdateConfetti() {
    for (int i = 0; i < MAX_CONFETTI; i++) {
        if (confetti[i].active) {
            allInactive = false;    // Reset the flag

            // Update position with drag effect
            confetti[i].velocity.x *= 0.99f;
            confetti[i].velocity.y *= 0.99f;

            // Increased movement multiplier for wider spread
             confetti[i].position.x += confetti[i].velocity.x * 0.6f;  // Increased from 0.4f
             confetti[i].position.y += confetti[i].velocity.y * 0.6f;  // Increased from 0.4f

            // Reduced gravity for more horizontal movement
            confetti[i].velocity.y += 0.02f;

            // Increased random movement for more spread
                 confetti[i].velocity.x += GetRandomValue(-20, 20) / 100.0f;  // Increased range
                 confetti[i].velocity.y += GetRandomValue(-20, 20) / 100.0f;  // Increased range

            // Slower fade out
            confetti[i].alpha -= 0.002f;
            confetti[i].lifetime -= 0.002f;

            // Increased bounds for off-screen check to allow more spread
            if (confetti[i].alpha <= 0 ||
                confetti[i].lifetime <= 0 ||
                confetti[i].position.y > SCREEN_HEIGHT + 50 ||  // Increased bounds
                confetti[i].position.x < -50 ||                 // Increased bounds
                confetti[i].position.x > SCREEN_WIDTH + 50) {   // Increased bounds
                confetti[i].active = false;
            }
        }
    }

    if (allInactive) {
        showPartyAnimation = false; // Stop the party animation
    }
}
```

```
// Update the game
void UpdateGame(Sound buttonClickSound, Sound popSound, Sound victorySound, Sound
loseSound, Sound drawSound, NaiveBayesModel *model, DecisionTreeNode *TDmodel)
{
    if (gameOver) return;

    // Update quit button position check
    if (IsMouseButtonPressed(MOUSE_LEFT_BUTTON))
    {
        Vector2 mousePos = GetMousePosition();
        if (mousePos.x >= 20 && mousePos.x <= 90 &&
            mousePos.y >= 10 && mousePos.y <= 40)
        {
            PlaySound(buttonClickSound);
            gameState = MENU;
            return;
        }
    }

    // Handle moves based on whose turn it is
    if (currentPlayerTurn == PLAYER_X_TURN)
    {
        // Handle human player X's turn
        if (HandlePlayerTurn(popSound, victorySound, loseSound, drawSound)) {
            PlaySound(popSound);
        }
    }
    else if (currentPlayerTurn == PLAYER_O_TURN)
    {
        if (isTwoPlayer)
        {
            // In 2 player mode, handle human player O's turn
            if (HandlePlayerTurn(popSound, victorySound, loseSound, drawSound)) {
                PlaySound(popSound);
            }
        }
        else
        {
            // In single player mode, handle AI's turn based on difficulty
            switch(currentDifficulty) {
                case EASY:
                    // Use ML models (Naive Bayes or Decision Tree) for EASY mode
                    if (currentModel == NAIVE_BAYES) {
                            AITurn(victorySound, loseSound, drawSound, model);   //
Naive Bayes
                    } else {
                            AITurnDecisionTree(victorySound, loseSound, drawSound,
TDmodel);  // Decision Tree
                    }
                    break;

                case MEDIUM:
                    // Use limited depth Minimax for MEDIUM
```

```
                        AITurn(victorySound, loseSound, drawSound, model);  // This
uses depthLimit = 4
                    break;

                case HARD:
                    // Use full depth Minimax for HARD
                        AITurn(victorySound, loseSound, drawSound, model);  // This
uses depthLimit = 9
                    break;
            }
        }
    }
}

// update the game when its over
void UpdateGameOver(Sound buttonClickSound) {
    if (IsMouseButtonPressed(MOUSE_LEFT_BUTTON)) {
        Vector2 mousePos = GetMousePosition();

        // Retry Button
        Rectangle retryBtn = {
            SCREEN_WIDTH/2 - BUTTON_WIDTH/2,
            SCREEN_HEIGHT/2 + 40,
            BUTTON_WIDTH,
            BUTTON_HEIGHT
        };

        // Back to Menu Button
        Rectangle menuBtn = {
            SCREEN_WIDTH/2 - BUTTON_WIDTH/2,
            SCREEN_HEIGHT/2 + 100,
            BUTTON_WIDTH,
            BUTTON_HEIGHT
        };

        if (CheckCollisionPointRec(mousePos, menuBtn)) {
            PlaySound(buttonClickSound);  // Play sound on button click
            gameState = MENU;
            InitGame();  // Reset the game state
        } else if (CheckCollisionPointRec(mousePos, retryBtn)) {
            PlaySound(buttonClickSound);  // Play sound on button click
            gameState = GAME;
            InitGame();  // Reset the game state for a new game
        }
    }
}
```

12.10. decisiontree.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
#include <time.h>
#include "decisiontree.h"

// Function to build, train, and evaluate the decision tree
void growth_Tree(DecisionTreeNode *tree) {
    DataRow dataset[MAX_ROWS];                                       // Array to
store the dataset
    DataRow train_set[MAX_ROWS], test_set[MAX_ROWS];             // Training and
testing datasets
     int dataset_size = 0, train_size = 0, test_size = 0;            // Sizes of
datasets
      int train_confusion[2][2] = {0}, test_confusion[2][2] = {0};    // Confusion
matrices
    float train_accuracy = 0.0, test_accuracy = 0.0;             // Accuracy for
training and testing
    double train_error_rate = 0.0, test_error_rate = 0.0;          // Error rates
    int correct_train = 0, correct_test = 0;                      // Correctly
classified samples

    // Initialize random seed for shuffling
    srand(time(NULL));

    // Load the dataset from the file
    load_dataset("tic-tac-toe.data", dataset, &dataset_size);

    // Shuffle the dataset to ensure random distribution
    shuffle_dataset(dataset, dataset_size);

    // Split the dataset into training (80%) and testing (20%) sets
       decision_tree_split_dataset(dataset,  dataset_size,  train_set,  &train_size,
test_set, &test_size, 0.8);

    // Build the decision tree using the training data
    tree = build_tree(train_set, train_size, 0);

    // Write position probabilities (weights) to a file
                       calculate_position_probabilities(dataset,      dataset_size,
"DecisionTree_ML/DTweights.txt");

    // Clear the output file before appending results
    FILE *file = fopen("DecisionTree_ML/DTconfusion_matrix.txt", "w");
    if (file) fclose(file);

    // Evaluate the decision tree on the training data
         train_accuracy  =  evaluate_with_randomness(tree,  train_set,  train_size,
train_confusion);
    correct_train = (int)(train_accuracy * train_size);
```

```c
                                              display_confusion_matrix(train_confusion,
"DecisionTree_ML/DTconfusion_matrix.txt", "Training");
        // printf("Training Accuracy: %.2f%% (%d/%d)\n", train_accuracy * 100,
correct_train, train_size);
       write_accuracy_to_file("DecisionTree_ML/DTconfusion_matrix.txt", "Training",
train_accuracy, correct_train, train_size);

    // Calculate error rate for the training data
          train_error_rate = calculate_error_rate(tree, train_set, train_size,
train_confusion);
    // printf("Training Error Rate: %.2f%%\n", train_error_rate);
    file = fopen("DecisionTree_ML/DTconfusion_matrix.txt", "a");
    if (file) {
        fprintf(file, "Training Error Rate: %.2f%%\n", train_error_rate);
        fclose(file);
    }

    // Evaluate the decision tree on the testing data
          test_accuracy = evaluate_with_randomness(tree, test_set, test_size,
test_confusion);
    correct_test = (int)(test_accuracy * test_size);
                                              display_confusion_matrix(test_confusion,
"DecisionTree_ML/DTconfusion_matrix.txt", "Testing");
         // printf("Testing Accuracy: %.2f%% (%d/%d)\n", test_accuracy * 100,
correct_test, test_size);
        write_accuracy_to_file("DecisionTree_ML/DTconfusion_matrix.txt", "Testing",
test_accuracy, correct_test, test_size);

    // Calculate error rate for the testing data
           test_error_rate = calculate_error_rate(tree, test_set, test_size,
test_confusion);
    // printf("Testing Error Rate: %.2f%%\n", test_error_rate);
    file = fopen("DecisionTree_ML/DTconfusion_matrix.txt", "a");
    if (file) {
        fprintf(file, "Testing Error Rate: %.2f%%\n", test_error_rate);
        fclose(file);
    }
}

// Function to load the dataset from a file
void load_dataset(const char *filename, DataRow dataset[], int *dataset_size) {
    // Open the dataset file in read mode
    FILE *file = fopen(filename, "r");
    if (!file) {
        // Print an error message if the file cannot be opened
        perror("Failed to open file");
        exit(1); // Exit the program with an error code
    }
    char line[256];    // Buffer to store each line of the file
    *dataset_size = 0; // Initialize the dataset size to zero
    // Read the file line by line
    while (fgets(line, sizeof(line), file)) {
        // Split the current line into tokens using ',' as the delimiter
```

```c
        char *token = strtok(line, ",");
        for (int i = 0; i < NUM_FEATURES; i++) {
            // Map the token value to corresponding feature representation
            if (strcmp(token, "x") == 0)
                dataset[*dataset_size].features[i] = 1; // Assign 1 for 'x'
            else if (strcmp(token, "o") == 0)
                dataset[*dataset_size].features[i] = 2; // Assign 2 for 'o'
            else
                dataset[*dataset_size].features[i] = 0; // Assign 0 for blank space
            // Move to the next token in the line
            token = strtok(NULL, ",");
        }
        // Assign the label based on the last token in the line
            dataset[*dataset_size].label = (strcmp(token, "positive\n") == 0) ?
DT_POSITIVE : DT_NEGATIVE;
        // Increment the dataset size after processing each line
        (*dataset_size)++;
    }
    // Close the file after reading is complete
    fclose(file);
}

// Function to shuffle the dataset
void shuffle_dataset(DataRow dataset[], int size) {
    for (int i = size - 1; i > 0; i--) {
        int j = rand() % (i + 1); // Generate random index
        DataRow temp = dataset[i]; // Swap elements
        dataset[i] = dataset[j];
        dataset[j] = temp;
    }
}

// Function to split dataset into training and testing sets
void decision_tree_split_dataset(DataRow dataset[], int dataset_size, DataRow
train_set[], int *train_size, DataRow test_set[], int *test_size, float
train_ratio) {
    int train_limit = (int)(dataset_size * train_ratio); // Calculate training data
size
    *train_size = 0;
    *test_size = 0;
    for (int i = 0; i < dataset_size; i++) {
        if (i < train_limit) {
            train_set[(*train_size)++] = dataset[i];
        } else {
            test_set[(*test_size)++] = dataset[i];
        }
    }
}

// Function to build the decision tree with depth limit
DecisionTreeNode *build_tree(DataRow dataset[], int size, int depth) {
    int positives = 0, negatives = 0;
```

```c
    // Count positive and negative labels in the dataset
    for (int i = 0; i < size; i++) {
        if (dataset[i].label == DT_POSITIVE)
            positives++; // Increment positive count for positive labels
        else
            negatives++; // Increment negative count for negative labels
    }

    // Stop conditions: max depth reached or node is pure (only positives or negatives)
    if (depth >= MAX_DEPTH || positives == 0 || negatives == 0) {
        // Allocate memory for a leaf node
        DecisionTreeNode *leaf = (DecisionTreeNode *)malloc(sizeof(DecisionTreeNode));
        leaf->is_leaf = 1; // Mark the node as a leaf
        leaf->prediction = (positives > negatives) ? DT_POSITIVE : DT_NEGATIVE; // Predict the majority class
        leaf->left = leaf->right = NULL; // Leaf nodes have no children
        return leaf; // Return the leaf node
    }

    // Variables to track the best feature and threshold for splitting
    int best_feature = -1, best_threshold = -1;
    float best_gini = 1.0;                    // Initialize the best Gini impurity to the highest value
    DataRow left[MAX_ROWS], right[MAX_ROWS];  // Temporary arrays for storing split datasets
    int left_size = 0, right_size = 0;        // Sizes of left and right subsets

    // Iterate over all features and possible thresholds to find the best split
    for (int feature_index = 0; feature_index < NUM_FEATURES; feature_index++) {
        for (int threshold = 0; threshold <= 2; threshold++) {
            // Calculate the Gini impurity for the current split
            float gini = calculate_gini_index(dataset, size, feature_index, threshold);
            if (gini < best_gini) {
                // Update the best Gini impurity, feature, and threshold if this split is better
                best_gini = gini;
                best_feature = feature_index;
                best_threshold = threshold;
            }
        }
    }

    // Split the dataset into left and right subsets based on the best feature and threshold
    decision_tree_split_data(dataset, size, best_feature, best_threshold, left, &left_size, right, &right_size);

    // Allocate memory for the new decision tree node
    DecisionTreeNode *node = (DecisionTreeNode *)malloc(sizeof(DecisionTreeNode));
```

```
    node->is_leaf = 0;                      // Mark the node as an internal (non-leaf)
node
    node->feature_index = best_feature; // Store the best feature for splitting
    node->threshold = best_threshold;   // Store the best threshold for splitting
    // Recursively build the left subtree using the left subset
    node->left = build_tree(left, left_size, depth + 1);
    // Recursively build the right subtree using the right subset
    node->right = build_tree(right, right_size, depth + 1);

    return node; // Return the newly created decision tree node
}

// Function to evaluate the decision tree with randomness and update the confusion
matrix
float evaluate_with_randomness(DecisionTreeNode *root, DataRow dataset[], int size,
int confusion_matrix[2][2]) {
    int correct_predictions = 0;          // Counter for correct predictions
    // Initialize confusion matrix to zero
    for (int i = 0; i < 2; i++) {         // Iterate through rows of the matrix
        for (int j = 0; j < 2; j++) {     // Iterate through columns of the matrix
            confusion_matrix[i][j] = 0;   // Set each cell to zero
        }
    }
    // Iterate through the dataset to populate the confusion matrix
    for (int i = 0; i < size; i++) {
        int prediction = predict_with_randomness(root, dataset[i].features); // Get
prediction from the decision tree
          int actual = dataset[i].label; // Retrieve the actual label from the
dataset

        if (actual == DT_POSITIVE && prediction == DT_POSITIVE) {
            confusion_matrix[0][0]++; // Increment True Positive (TP)
            correct_predictions++;    // Increment correct predictions count
        } else if (actual == DT_NEGATIVE && prediction == DT_NEGATIVE) {
            confusion_matrix[1][1]++; // Increment True Negative (TN)
            correct_predictions++;    // Increment correct predictions count
        } else if (actual == DT_NEGATIVE && prediction == DT_POSITIVE) {
            confusion_matrix[1][0]++; // Increment False Positive (FP)
        } else if (actual == DT_POSITIVE && prediction == DT_NEGATIVE) {
            confusion_matrix[0][1]++; // Increment False Negative (FN)
        }
    }
    // Return the accuracy as the ratio of correct predictions to the total dataset
size
    return (float)correct_predictions / size; // Calculate accuracy
}

// Function to make predictions with randomness in the decision tree
int predict_with_randomness(DecisionTreeNode *node, int features[]) {
    if (!node) {
        return DT_NEGATIVE; // Default prediction if the node is NULL
    }
    if (node->is_leaf) {
```

```
        // Introduce randomness to the prediction
        if ((float)rand() / RAND_MAX < RANDOMNESS_FACTOR) { // Compare a random value to RANDOMNESS_FACTOR
            return (node->prediction == DT_POSITIVE) ? DT_NEGATIVE : DT_POSITIVE; // Flip the prediction randomly
        }
        return node->prediction; // Return the prediction stored in the leaf node
    }
    // Traverse the decision tree based on the feature threshold
    if (features[node->feature_index] <= node->threshold) { // Compare feature value with threshold
        return predict_with_randomness(node->left, features); // Traverse left subtree if the condition is met
    } else {
        return predict_with_randomness(node->right, features); // Traverse right subtree otherwise
    }
}

// Function to display and log the confusion matrix to a file
void display_confusion_matrix(int confusion_matrix[2][2], const char *filename, const char *dataset_type) {
    FILE *file = fopen(filename, "a"); // Open file in append mode
    if (!file) {
        perror("Failed to open confusion matrix file");
        return;
    }

    // Extract TP, TN, FP, FN from the confusion matrix
    int TP = confusion_matrix[0][0];
    int FP = confusion_matrix[1][0];
    int TN = confusion_matrix[1][1];
    int FN = confusion_matrix[0][1];

    // Print confusion matrix and metrics to console
    /*
    printf("\nDecision Tree %s Confusion Matrix:\n", dataset_type);
    printf("    True Positive (TP): %d\n", TP);
    printf("    False Positive (FP): %d\n", FP);
    printf("    True Negative (TN): %d\n", TN);
    printf("    False Negative (FN): %d\n", FN);
    printf("\nConfusion Matrix:\n");
    printf("                  Predicted Positive    Predicted Negative\n");
    printf("Actual Positive        %10d%20d\n", TP, FN);
    printf("Actual Negative        %10d%20d\n", FP, TN);
    printf("---------------------------------------------------------\n");
    */

    // Write confusion matrix and metrics to file
    fprintf(file, "\nDecision Tree %s Confusion Matrix:\n", dataset_type);
    fprintf(file, "    True Positive (TP): %d\n", TP);
    fprintf(file, "    False Positive (FP): %d\n", FP);
    fprintf(file, "    True Negative (TN): %d\n", TN);
```

```
    fprintf(file, "    False Negative (FN): %d\n", FN);
    fprintf(file, "\nConfusion Matrix:\n");
    fprintf(file, "                 Predicted Positive    Predicted Negative\n");
    fprintf(file, "Actual Positive        %10d%20d\n", TP, FN);
    fprintf(file, "Actual Negative        %10d%20d\n", FP, TN);
    fprintf(file, "----------------------------------------------------------\n");

    fclose(file); // Close the file properly
}

// Function to write accuracy results to a file
void write_accuracy_to_file(const char *filename, const char *dataset_type, float
accuracy, int correct, int total) {
    FILE *file = fopen(filename, "a"); // Open file in append mode to add data
    if (!file) { // Check if the file was opened successfully
        perror("Failed to open file for writing accuracy"); // Print error message
if file open fails
        return; // Exit the function if file cannot be opened
    }

    // Write the dataset type, accuracy percentage, and correct classification
counts to the file
    fprintf(file, "%s Accuracy: %.2f%% (%d/%d)\n", dataset_type, accuracy * 100,
correct, total);
    fclose(file); // Close the file to save changes
}

// Function to free the memory allocated for the decision tree
void free_tree(DecisionTreeNode *node) {
    if (node == NULL) return; // Base case: If the node is NULL, nothing to free,
so return
    // Recursively free memory for the left subtree
    free_tree(node->left);
    // Recursively free memory for the right subtree
    free_tree(node->right);
    free(node); // Free the current node's memory
}

// Function to calculate the Gini index for a potential split
float calculate_gini_index(DataRow dataset[], int size, int feature_index, int
threshold) {
    DataRow left[MAX_ROWS], right[MAX_ROWS]; // Temporary arrays to store left and
right branches
    int left_size = 0, right_size = 0; // Initialize sizes of left and right
branches

    // Split the dataset into left and right branches based on the feature and
threshold
    decision_tree_split_data(dataset, size, feature_index, threshold, left,
&left_size, right, &right_size);

    // If either branch is empty, return the worst Gini index (1.0) to discourage
this split
```

```
    if (left_size == 0 || right_size == 0) return 1.0;
    // Initialize Gini indices for the left and right branches
    float gini_left = 1.0, gini_right = 1.0;
    int positives_left = 0, positives_right = 0; // Counters for positive labels in each branch
    // Count positive labels in the left branch
    for (int i = 0; i < left_size; i++) {
        if (left[i].label == DT_POSITIVE) positives_left++;
    }
    // Count positive labels in the right branch
    for (int i = 0; i < right_size; i++) {
        if (right[i].label == DT_POSITIVE) positives_right++;
    }

    // Calculate the probability of positive labels in the left branch
    float prob_left = (float)positives_left / left_size;
    // Calculate the Gini index for the left branch
      gini_left = 1.0 - (prob_left * prob_left) - ((1.0 - prob_left) * (1.0 - prob_left));
    // Calculate the probability of positive labels in the right branch
    float prob_right = (float)positives_right / right_size;
    // Calculate the Gini index for the right branch
     gini_right = 1.0 - (prob_right * prob_right) - ((1.0 - prob_right) * (1.0 - prob_right));

    // Return the weighted average of the Gini indices for both branches
    return ((gini_left * left_size) + (gini_right * right_size)) / size;
}

// Function to split the dataset into left and right branches
void decision_tree_split_data(DataRow dataset[], int size, int feature_index, int threshold, DataRow left[], int *left_size, DataRow right[], int *right_size) {
    *left_size = 0; // Initialize the size of the left branch to zero
    *right_size = 0; // Initialize the size of the right branch to zero

     // Iterate through the dataset to classify each data point into the left or right branch
    for (int i = 0; i < size; i++) {
           if (dataset[i].features[feature_index] <= threshold) { // Check if the feature value is less than or equal to the threshold
              left[(*left_size)++] = dataset[i]; // Add the data point to the left branch and increment its size
        } else { // Otherwise, add the data point to the right branch
            right[(*right_size)++] = dataset[i]; // Add the data point to the right branch and increment its size
        }
    }
}

// Function to predict the best move for the current player based on the decision tree
void dt_predict_best_move(DecisionTreeNode *tree, char board[3][3], char current_player, int *best_row, int *best_col) {
```

```
    if (!tree) {                  // Check if the decision tree is not initialized
            printf("Error: Decision tree is not initialized!\n"); // Print error
message
        return;                   // Exit the function
    }

     int features[NUM_FEATURES]; // Array to store the board features as numerical
values
     int max_positive_prob = -1; // Variable to track the highest probability for a
positive outcome
     *best_row = -1;               // Initialize the best_row variable to an invalid
value
     *best_col = -1;               // Initialize the best_col variable to an invalid
value
     int attempts = 0;           // Counter to limit the number of attempts to find
the best move

    // Convert the 3x3 board into a feature array
    for (int i = 0; i < 3; i++) { // Loop through each row
        for (int j = 0; j < 3; j++) { // Loop through each column
            if (board[i][j] == 'x') features[i * 3 + j] = 1; // Map 'x' to 1
            else if (board[i][j] == 'o') features[i * 3 + j] = 2; // Map 'o' to 2
            else features[i * 3 + j] = 0; // Map empty cells ('b') to 0
        }
    }

    // Attempt to find the best move within a maximum of 5 iterations
    for (attempts = 0; attempts < 5; attempts++) {
          int temp_row = -1, temp_col = -1; // Temporary variables to store the
coordinates of the current best move

        // Iterate over all cells of the board
        for (int i = 0; i < 3; i++) { // Loop through rows
            for (int j = 0; j < 3; j++) { // Loop through columns
                if (board[i][j] == 'b') { // Check if the current cell is empty
                       // Temporarily set the current player's move in the feature
array
                      features[i * 3 + j] = (current_player == 'x') ? 1 : 2; // Map
'x' to 1 and 'o' to 2

                    // Use the decision tree to predict the outcome of this move
                    int prediction = predict_with_randomness(tree, features);

                       // If the prediction is positive and better than the current
best, update the best move
                       if (prediction == DT_POSITIVE && (max_positive_prob == -1 ||
prediction > max_positive_prob)) {
                        temp_row = i; // Update the row of the best move
                        temp_col = j; // Update the column of the best move
                             max_positive_prob = prediction; // Update the highest
positive probability
                    }
```

```
                    // Reset the feature array for the current cell back to empty
                    features[i * 3 + j] = 0;
                }
            }
        }

        // If a valid positive move is found, update best_row and best_col and exit
the loop
        if (temp_row != -1 && temp_col != -1) {
            *best_row = temp_row; // Set the best move's row
            *best_col = temp_col; // Set the best move's column
            return; // Exit the function
        }
    }

    // If no positive move is found after 5 attempts, choose any random empty cell
    for (int i = 0; i < 3; i++) { // Loop through rows
        for (int j = 0; j < 3; j++) { // Loop through columns
            if (board[i][j] == 'b') { // Check if the cell is empty
                *best_row = i; // Assign the row of the random empty cell
                *best_col = j; // Assign the column of the random empty cell
                return; // Exit the function
            }
        }
    }
}

// Function to recursively print the structure of the decision tree
void print_tree(DecisionTreeNode *node, int depth) {
    if (!node) {
        // Base case: If the node is NULL, return
        return;
    }

    if (node->is_leaf) {
        // Print leaf node details
        // printf("%*sLeaf: Prediction = %d\n", depth * 4, "", node->prediction);
    } else {
        // Print internal node details
            // printf("%*sNode: Feature = %d, Threshold = %d\n", depth * 4, "",
node->feature_index, node->threshold);
        print_tree(node->left, depth + 1); // Recur for the left child
        print_tree(node->right, depth + 1); // Recur for the right child
    }
}

// Function to calculate and save position probabilities for the dataset
void calculate_position_probabilities(DataRow dataset[], int dataset_size, const
char *filename) {
      int positive_count = 0, negative_count = 0; // Counters for the number of
positive and negative samples
      int position_count[NUM_FEATURES][3][2] = {0}; // Array to store counts of
symbols ('x', 'o', empty) for each position and class
```

```c
    // Count occurrences of each symbol ('x', 'o', empty) in every position for
each class
    for (int i = 0; i < dataset_size; i++) {
        if (dataset[i].label == DT_POSITIVE) positive_count++; // Increment
positive count if label is positive
        else negative_count++; // Increment negative count otherwise

        // Iterate over each feature (board position)
        for (int j = 0; j < NUM_FEATURES; j++) {
            if (dataset[i].features[j] == 1)
position_count[j][0][dataset[i].label]++; // Count 'x'
            else if (dataset[i].features[j] == 2)
position_count[j][1][dataset[i].label]++; // Count 'o'
            else position_count[j][2][dataset[i].label]++; // Count empty spaces
        }
    }

    // Open the file to save calculated probabilities
    FILE *file = fopen(filename, "w");
    if (!file) { // Check if the file was successfully opened
        perror("Failed to open file to save weights");
        return; // Exit the function
    }

    // Write the class probabilities to the file
    fprintf(file, "Class Probabilities:\n");
    fprintf(file, "  Positive: P(Positive) = %.4f\n", (double)positive_count /
dataset_size); // Probability of positive class
    fprintf(file, "  Negative: P(Negative) = %.4f\n", (double)negative_count /
dataset_size); // Probability of negative class
    fprintf(file, "--------------------------------------------\n");

    // Write position-wise probabilities to the file
    for (int i = 0; i < NUM_FEATURES; i++) { // Loop through each board position
        fprintf(file, "Position %d:\n", i + 1); // Position label (1-indexed)
        fprintf(file, "  Symbol | P(Symbol | Positive) | P(Symbol | Negative)\n");
        fprintf(file, "  -------|----------------------|----------------------\n");

        const char *symbols[] = {"x", "o", "b"}; // Define symbols corresponding to
feature values
        for (int j = 0; j < 3; j++) { // Loop through symbols
            double p_positive = (positive_count > 0) ?
(double)position_count[i][j][DT_POSITIVE] / positive_count : 0.0; // Probability of
symbol given positive class
            double p_negative = (negative_count > 0) ?
(double)position_count[i][j][DT_NEGATIVE] / negative_count : 0.0; // Probability of
symbol given negative class
            fprintf(file, "  %-6s | %-20.4f | %-20.4f\n", symbols[j], p_positive,
p_negative); // Write probabilities to file
        }
        fprintf(file, "--------------------------------------------\n"); //
Separator line for readability
```

```
    }

    fclose(file); // Close the file after writing
      printf("Weights updated and saved to %s\n", filename); // Notify user of
success
}

// Function to calculate the error rate of the decision tree
double calculate_error_rate(DecisionTreeNode *root, DataRow dataset[], int size,
int confusion_matrix[2][2]) {
    int error_count = 0; // Counter to track the number of incorrect predictions

    // Iterate over the dataset to compare predictions with actual labels
    for (int i = 0; i < size; i++) {
        int prediction = predict_with_randomness(root, dataset[i].features); // Get
the predicted label from the decision tree
        int actual = dataset[i].label; // Get the actual label from the dataset

        if (prediction != actual) { // Check if the prediction is incorrect
            error_count++; // Increment the error count
        }
    }

    // Calculate and return the error rate as a percentage of total samples
    return ((double)error_count / size) * 100;
}
```

12.11. data_processing.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include "main.h"

// Function to load data from file into arrays
void load_data(const char *filename, char boards[][NUM_POSITIONS + 1], int
outcomes[], int *total_records) {
    FILE *file_ptr = fopen(filename, "r");      // Open file of dataset to read
     if (file_ptr == NULL) {                     // Check if its an existing file,
else will send an error
        perror("Failed to open file");
        exit(1);
    }

    char line[50];          // Array buffer for each line in dataset
      // This loop gets each line and store the individual attributes and the
respective outcome in two different arrays, board & outcome
    while (fgets(line, sizeof(line), file_ptr)) {
         char board[NUM_POSITIONS + 1];                    // Array buffer for the 9
attributes per line
        char outcome[10];                                 // Array buffer for outcome
        sscanf(line, "%c,%c,%c,%c,%c,%c,%c,%c,%c,%s",    // Splitting each line into
the 9 different attributes and the outcome
               &board[0], &board[1], &board[2], &board[3],
               &board[4], &board[5], &board[6], &board[7],
               &board[8], outcome);
        strcpy(boards[*total_records], board);                  // Store attributes
into board array
             outcomes[*total_records] = outcome_index(outcome);          // Store
respectively outcome in outcome array
        (*total_records)++;      // count of total number of lines in dataset
    }
    fclose(file_ptr);        // Close file
}

// Function to split dataset into training and testing datasets
void split_data(char boards[][NUM_POSITIONS + 1], int outcomes[], int
total_records, char train_boards[][NUM_POSITIONS + 1], int train_outcomes[], char
test_boards[][NUM_POSITIONS + 1], int test_outcomes[], int *train_size, int
*test_size, float ratio) {
    // Shuffle the dataset using Fisher-Yates algorithm
    srand(time(NULL));
    for (int i = total_records-1; i > 0; i--) {
        int j = rand() % (i + 1);   // Get unique random number

        // Swap boards[i] and boards[j]
        char temp_board[10];
        strcpy(temp_board, boards[i]);
        strcpy(boards[i], boards[j]);
        strcpy(boards[j], temp_board);

        // Swap outcomes[i] and outcomes[j]
        char temp_outcome;
        temp_outcome = outcomes[i];
        outcomes[i] = outcomes[j];
        outcomes[j] = temp_outcome;
    }
```

```c
    int target_train_size = (int)(ratio * total_records);        // Get number of
lines for training dataset, in this case 80% of total_records


    // Loop to separate original dataset into training(80%) and testing(20%)
dataset for machine learning
    for (int i = 0; i < total_records; i++) {
        if (*train_size < target_train_size) { // 80%
            strcpy(train_boards[*train_size], boards[i]);
            train_outcomes[*train_size] = outcomes[i];
            (*train_size)++;
        } else { // 20%
            strcpy(test_boards[*test_size], boards[i]);
            test_outcomes[*test_size] = outcomes[i];
            (*test_size)++;
        }
    }

    // Print out first 10 lines of train_boards array for visualization
    printf("\ntrain_boards array:\n");
    for (int i = 0; i < 10; i++){
        printf("%s\n", train_boards[i]);
    };

    // Print out first 10 lines of train_outcomes array for visualization
    printf("\ntrain_outcomes array:\n");
    for (int i = 0; i < 10; i++){
        printf("%d\n", train_outcomes[i]);
    };
}

// Utility function to convert the string outcome ("positive" or "negative") into
the corresponding numerical label (POSITIVE(0) or NEGATIVE(1)).
int outcome_index(const char *outcome) {
    return (strcmp(outcome, "positive") == 0) ? POSITIVE : NEGATIVE;
}
```

12.12. NBmodel.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "main.h"

// Function to train Naive Bayes model
void train_NBmodel(NaiveBayesModel *model, char boards[][NUM_POSITIONS + 1], int
outcomes[], int size) {
     int positive_count = 0, negative_count = 0;                // Count for the
occurrences of the two different outcomes "positive" and "negative" in the dataset

    // Initialize counts
     int x_counts[NUM_POSITIONS][NUM_OUTCOMES] = {0};         // Array for number of
occurrences of 'x' in every position for both "positive" and "negative" outcome
     int o_counts[NUM_POSITIONS][NUM_OUTCOMES] = {0};         // Array for number of
occurrences of 'o' in every position for both "positive" and "negative" outcome
     int b_counts[NUM_POSITIONS][NUM_OUTCOMES] = {0};         // Array for number of
occurrences of 'b' in every position for both "positive" and "negative" outcome

    // Count occurrences
    for (int i = 0; i < size; i++) {
        int outcome_idx = outcomes[i];
         if (outcome_idx == POSITIVE) positive_count++;       // Counting of number
of times its a 'positive' or 'negative'
          else negative_count++;                              // outcome in the
dataset

         // Loop to count how often 'x', 'o' and blank('b') appear in each position
for each outcome
        for (int j = 0; j < NUM_POSITIONS; j++) {
             if (boards[i][j] == 'x') x_counts[j][outcome_idx]++;         // Count
for 'x'
             else if (boards[i][j] == 'o') o_counts[j][outcome_idx]++;    // Count
for 'o'
             else b_counts[j][outcome_idx]++;                             // Count
for 'b'
        }
    }

    // Print out x_counts array for visualization
    printf("\n    x_counts array\n\n");
    printf("             pos  neg\n");
    for (int i = 0; i < NUM_POSITIONS; i++) {
        printf("Position %d [", i+1);
        for (int j = 0; j < NUM_OUTCOMES; j++){
            printf(" %d ", x_counts[i][j]);
        }
        printf("]\n");
    }

    // Print out o_counts array for visualization
    printf("\n    o_counts array\n\n");
    printf("             pos  neg\n");
    for (int i = 0; i < NUM_POSITIONS; i++) {
        printf("Position %d [", i+1);
        for (int j = 0; j < NUM_OUTCOMES; j++){
            printf(" %d ", o_counts[i][j]);
        }
        printf("]\n");
```

```
    }

    // Print out b_counts array for visualization
    printf("\n    b_counts array\n\n");
    printf("                pos  neg\n");
    for (int i = 0; i < NUM_POSITIONS; i++) {
        printf("Position %d [", i+1);
        for (int j = 0; j < NUM_OUTCOMES; j++){
            printf(" %d ", b_counts[i][j]);
        }
        printf("]\n");
    }

    // Calculate prior probabilities
      model->class_probs[POSITIVE]  =  (double)positive_count  /  size;            //
Probability of outcome being "positive" in the dataset
      model->class_probs[NEGATIVE]  =  (double)negative_count  /  size;            //
Probability of outcome being "negative" in the dataset

    // Calculate conditional probabilities with Laplace smoothing
                for    (int    i    =    0;    i    <    NUM_POSITIONS;    i++)    {
// Loop through each of the 9 position on the board
             model->x_probs[i][POSITIVE]  =  (double)(x_counts[i][POSITIVE]  +  1)  /
(positive_count  +  3);    // Calculate the probability of 'x' appearing in each
position for all "positive" outcomes
             model->x_probs[i][NEGATIVE]  =  (double)(x_counts[i][NEGATIVE]  +  1)  /
(negative_count  +  3);    // Calculate the probability of 'x' appearing in each
position for all "negative" outcomes

             model->o_probs[i][POSITIVE]  =  (double)(o_counts[i][POSITIVE]  +  1)  /
(positive_count  +  3);    // Calculate the probability of 'o' appearing in each
position for all "positive" outcomes
             model->o_probs[i][NEGATIVE]  =  (double)(o_counts[i][NEGATIVE]  +  1)  /
(negative_count  +  3);    // Calculate the probability of 'o' appearing in each
position for all "negative" outcomes

             model->b_probs[i][POSITIVE]  =  (double)(b_counts[i][POSITIVE]  +  1)  /
(positive_count  +  3);    // Calculate the probability of 'b' appearing in each
position for all "positive" outcomes
             model->b_probs[i][NEGATIVE]  =  (double)(b_counts[i][NEGATIVE]  +  1)  /
(negative_count  +  3);    // Calculate the probability of 'b' appearing in each
position for all "negative" outcomes
    }
}

// Function to save model weights
void save_NBmodel(const NaiveBayesModel *model, const char *filename) {
    FILE *file_ptr = fopen(filename, "w");      // Open text file to write
     if  (file_ptr == NULL)  {                         // Check if can open or create
file, else will send an error
        perror("Failed to open file for saving model");
        return;
    }

    // Save the prior probabilities
    fprintf(file_ptr, "Class Probabilities:\n");
    fprintf(file_ptr, "P(Positive): %f\n", model->class_probs[POSITIVE]);
    fprintf(file_ptr, "P(Negative): %f\n\n", model->class_probs[NEGATIVE]);

     // Loop to save each conditional probabilities of each feature ['x', 'o' ,'b']
for each outcome
```

```c
    for (int i = 0; i < NUM_POSITIONS; i++) {
        fprintf(file_ptr, "Position %d:\n", i + 1);
        fprintf(file_ptr, "P(x | Positive): %f\n", model->x_probs[i][POSITIVE]);
        fprintf(file_ptr, "P(x | Negative): %f\n", model->x_probs[i][NEGATIVE]);
        fprintf(file_ptr, "P(o | Positive): %f\n", model->o_probs[i][POSITIVE]);
        fprintf(file_ptr, "P(o | Negative): %f\n", model->o_probs[i][NEGATIVE]);
        fprintf(file_ptr, "P(b | Positive): %f\n", model->b_probs[i][POSITIVE]);
        fprintf(file_ptr, "P(b | Negative): %f\n\n", model->b_probs[i][NEGATIVE]);
    }

    fclose(file_ptr);       // Close file
    printf("\nModel weights saved to %s\n", filename);
}

// Function to test accuracy of model
void test_NBmodel(const char *filename, char mode[], char type[], NaiveBayesModel
*model, char boards[][NUM_POSITIONS + 1], int outcomes[], int size) {
    int true_positive = 0;       // Count of true positives
    int false_positive = 0;      // Count of false positives
    int true_negative = 0;       // Count of true negatives
    int false_negative = 0;      // Count of false negatives
    int error_count = 0;         // Count of prediction errors

    // Loop to count for each of the 4 classes (TP, FP, TN, FN)
    for (int i = 0; i < size; i++) {
        int predicted_outcome = predict_outcome(model, boards[i]);
        if (outcomes[i] == POSITIVE && predicted_outcome == POSITIVE ) {
            true_positive++;
        }
        else if (outcomes[i] == POSITIVE && predicted_outcome == NEGATIVE ) {
            false_negative++;
            error_count++;
        }
        else if (outcomes[i] == NEGATIVE && predicted_outcome == NEGATIVE ) {
            true_negative++;
        }
        else {
            false_positive++;
            error_count++;
        };
    }

    // Calculate probability of error
    double prob_of_error = (double)error_count / size * 100;

    FILE *file_ptr = fopen(filename, mode);     // Open text file to write
     if (file_ptr == NULL) {                        // Check if can open or create
file, else will send an error
        perror("Failed to open file");
        exit(1);
    }

    // Information to write in text file
     if (strcmp(type, "Testing") == 0) fprintf(file_ptr, "\n\n");          // If
writing for Testing dataset, indent two newlines for easier readability

                          fprintf(file_ptr,      "%s      Dataset:\n",      type);
// Type of dataset (Training/Testing)
     fprintf(file_ptr, "  Accuracy: %.2f%% (%d/%d)\n", 100 - prob_of_error, size -
error_count, size);       // Prediction Accuracy of model on dataset
```

```
    fprintf(file_ptr, "   Error: %.2f%% (%d/%d)\n", prob_of_error, error_count,
size);                        // Probability of error of model on dataset
                        fprintf(file_ptr,     "         Confusion     Matrix:\n");
// Confusion Matrix Values
           fprintf(file_ptr,   "          True   Positive:   %d\n",   true_positive);
// Number of True Positive predicted by model
          fprintf(file_ptr,   "          False   Positive:   %d\n",   false_positive);
// Number of False Positive predicted by model
           fprintf(file_ptr,   "          True   Negative:   %d\n",   true_negative);
// Number of True Negative predicted by model
          fprintf(file_ptr,   "          False   Negative:   %d\n",   false_negative);
// Number of False Negative predicted by model

    fclose(file_ptr);        // Close file
}

// Function to calculate the posterior probability of a specified outcome based on
the given board layout.
double  calculate_probability(NaiveBayesModel  *model,  const  char  board[],  int
outcome) {
     double probability = model->class_probs[outcome];         // P(C); Get prior
probability of given/set outcome, "positive" or "negative"

    // Loop to iterate over each position in the board array to update probability
by multiplying with the conditional probability of the feature in that position
    for (int i = 0; i < NUM_POSITIONS; i++) {
        if (board[i] == 'x') probability *= model->x_probs[i][outcome];          //
P(X | C) for 'x'
        else if (board[i] == 'o') probability *= model->o_probs[i][outcome];    //
P(X | C) for 'o'
        else probability *= model->b_probs[i][outcome];                          //
P(X | C) for blank 'b'
    }
    return probability;
}

//  Function  to  predict  outcome  of  given  board  layout;  either  "positive"  or
"negative"
int predict_outcome(NaiveBayesModel *model, const char board[]) {
    double positive_prob = calculate_probability(model, board, POSITIVE);       //
Calculate posterior probability for "positive" outcome
    double negative_prob = calculate_probability(model, board, NEGATIVE);       //
Calculate posterior probability for "negative" outcome
    return (positive_prob > negative_prob) ? POSITIVE : NEGATIVE;               //
Compare both posterior probability and return the outcome with the highest one
}

// Function to find next best move for the NBmodel based on the given board layout
int  predict_move(NaiveBayesModel  *model,  Cell  grid[GRID_SIZE][GRID_SIZE],  int
*bestRow, int *bestCol) {
    int best_move = -1;              // Index of the best move position
    double best_prob = 0.0;          // Probability of best move
    char board[NUM_POSITIONS + 1];   // Buffer array for board layout
    int k = 0;                       // Index of buffer array

    // AI's thinking process visualization
    printf("\nAI's Turn");
    //Print initial grid layout for visualization
    printf("\nGame board layout as grid(array) format:\n");
```

```
    // Loop to translate the board layout in the GUI into an array for the model to read
    for (int i = 0; i < GRID_SIZE; i++) {
        printf("[");         // For visualization
        for (int j = 0; j < GRID_SIZE; j++) {
              if (grid[i][j] == EMPTY) {              // If position is empty, set respective position/index in buffer array to blank 'b'
                board[k] = 'b';
                printf("b");        // For visualization
            }
               else if (grid[i][j] == PLAYER_O){     // If position has 'o', set respective position/index in buffer array to 'o'
                board[k] = 'o';
                printf("o");        // For visualization
            }
            else {
                    board[k] = 'x';                   // If position has 'x', set respective position/index in buffer array to 'x'
                printf("x");        // For visualization
            }
            k++;
        }
        printf("]\n");        // For visualization
    }

    // Print grid as string after conversion for visualization
    printf("\nGame board layout as string:\n");
    printf("%s\n", board);

      // Print the current simulated move and board layout and the respective probability of winning
      printf("\nSimulated move              Simulated board               Posterior Probability\n");

     // Loop over the board layout to compare which available position gives higher probability of winning
    for (int i = 0; i < NUM_POSITIONS; i++) {
        if (board[i] == 'b') {       // Check if position is available
             char temp_board[NUM_POSITIONS + 1];      // Buffer array to simulate a move on current board layout
            strcpy(temp_board, board);
                temp_board[i] = 'x'; // Assume AI is 'x' as 'x' is the winning perspective

                  double positive_prob = calculate_probability(model, temp_board, POSITIVE);       // Calculate posterior probability of a "positive" outcome with the temporary board layout based on the simulated move

             // Check if latest calculated posterior probability is the highest so far, if yes, replace best_prob with that, and best_move with the simulated move
            if (positive_prob > best_prob) {
                best_prob = positive_prob;
                best_move = i;
            }

            // Print the current simulated move and board layout and the respective probability of winning
             printf("        %d                    %s                       %f\n", i, temp_board, positive_prob);
        }
    }
```

```
    // Use divide function to translate best_move into the (x,y) position of the
board layout for the model to place move on GUI.
    // For example, if best_move is 2, it will be translated as (0,2), meaning
first row & third column on board
    divide(best_move, 3, bestRow, bestCol);

    // Print conversion of best_move integer to bestRow index and bestCol index for
visualization
    printf("\nBest move: %d -> (%d, %d)\n", best_move, *bestRow, *bestCol);

    return 0;
}

// Function to get quotient and remainder of an integer
void divide(int dividend, int divisor, int *quo, int *rem){
    *quo = dividend / divisor;      // Compute quotient
    *rem = dividend % divisor;      // Compute remainder
}
```

12.13. plot_confusion_plot.py

```python
import matplotlib.pyplot as plt
import seaborn as sns
import numpy as np

with open("NBmodel/NBmodel_confusion_matrix.txt", 'r') as file:
    content = file.readlines()

# Initialize dictionary to store results
results = {
    "Training": {"Correct Predictions": 0, "Total Predictions": 0, "Accuracy": 0, "Error": 0, "True Positive": 0, "False Positive": 0, "True Negative": 0, "False Negative": 0},
    "Testing": {"Correct Predictions": 0, "Total Predictions": 0, "Accuracy": 0, "Error": 0, "True Positive": 0, "False Positive": 0, "True Negative": 0, "False Negative": 0},
}

# Flag to track whether is training or testing dataset
current_dataset = None

# Iterate through each line
for line in content:
    line = line.strip()  # Remove any extra whitespace or newline characters

    # Identify the dataset (Training or Testing) based on the labels
    if "Training Dataset:" in line:
        current_dataset = "Training"
    elif "Testing Dataset:" in line:
        current_dataset = "Testing"

    # Parse Accuracy and Error
    if "Accuracy:" in line:
        results[current_dataset]["Accuracy"] = float(line.split(":")[1].split("(")[0].strip().replace('%', ''))
        correct, total = map(int, line.split("(")[1].split(")")[0].split("/"))
        results[current_dataset]["Correct Predictions"] = correct
        results[current_dataset]["Total Predictions"] = total
    elif "Error:" in line:
        results[current_dataset]["Error"] = float(line.split(":")[1].split("(")[0].strip().replace('%', ''))

    # Parse confusion matrix values
    elif "True Positive:" in line:
        results[current_dataset]["True Positive"] = int(line.split(":")[1].strip())
    elif "False Positive:" in line:
        results[current_dataset]["False Positive"] = int(line.split(":")[1].strip())
    elif "True Negative:" in line:
        results[current_dataset]["True Negative"] = int(line.split(":")[1].strip())
    elif "False Negative:" in line:
        results[current_dataset]["False Negative"] = int(line.split(":")[1].strip())

# # Print results
# print("Results:")
# for dataset, metrics in results.items():
#     print(f"\n{dataset} Dataset:")
#         print(f"    Accuracy: {metrics['Accuracy']}% ({metrics['Correct Predictions']}/{metrics['Total Predictions']})")
```

```
#     print(f"  Error: {metrics['Error']}%")
#     print("  Confusion Matrix:")
#     print(f"    True Positive: {metrics['True Positive']}")
#     print(f"    False Positive: {metrics['False Positive']}")
#     print(f"    True Negative: {metrics['True Negative']}")
#     print(f"    False Negative: {metrics['False Negative']}")

# Confusion matrix values for prediction on Training dataset
training_cm = np.array([
    [results["Training"]["True Negative"], results["Training"]["False Positive"]],
    [results["Training"]["False Negative"], results["Training"]["True Positive"]]
])

# Confusion matrix values for prediction on Testing dataset
testing_cm = np.array([
    [results["Testing"]["True Negative"], results["Testing"]["False Positive"]],
    [results["Testing"]["False Negative"], results["Testing"]["True Positive"]]
])

# Labels for plots
labels = [
    ['TN', 'FP'],
    ['FN', 'TP']
]

# Set up plots
fig, axes = plt.subplots(1, 2, figsize=(14, 7))

# Plotting Training Confusion Matrix
sns.heatmap(training_cm,         annot=False,         fmt='d',          cmap='Blues',
xticklabels=['Predicted NO', 'Predicted YES'], yticklabels=['Actual NO', 'Actual
YES'], ax=axes[0], cbar=False)
for i in range(training_cm.shape[0]):
    for j in range(training_cm.shape[1]):
        axes[0].text(j + 0.5, i + 0.5, f"{labels[i][j]} = {training_cm[i, j]}",
                     ha='center', va='center', color='black', fontsize=14)
axes[0].set_title('Training Dataset Confusion Matrix')
axes[0].set_xlabel(f"n = {results['Training']['Total Predictions']}")

# Plotting Testing Confusion Matrix
sns.heatmap(testing_cm, annot=False, fmt='d', cmap='Reds', xticklabels=['Predicted
NO',  'Predicted  YES'],  yticklabels=['Actual  NO',  'Actual  YES'],  ax=axes[1],
cbar=False)
for i in range(testing_cm.shape[0]):
    for j in range(testing_cm.shape[1]):
        axes[1].text(j + 0.5, i + 0.5, f"{labels[i][j]} = {testing_cm[i, j]}",
                     ha='center', va='center', color='black', fontsize=14)
axes[1].set_title('Testing Dataset Confusion Matrix')
axes[1].set_xlabel(f"n = {results['Testing']['Total Predictions']}")

# Save plots
plt.savefig("NBmodel/NBmodel_confusion_matrix.png")

# Display plots
plt.tight_layout()
plt.show()
```

12.14. decisiontree.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
#include <time.h>
#include "decisiontree.h"

// Function to build, train, and evaluate the decision tree
void growth_Tree(DecisionTreeNode *tree) {
    DataRow dataset[MAX_ROWS];                                      // Array to
store the dataset
    DataRow train_set[MAX_ROWS], test_set[MAX_ROWS];               // Training and
testing datasets
    int dataset_size = 0, train_size = 0, test_size = 0;           // Sizes of
datasets
    int train_confusion[2][2] = {0}, test_confusion[2][2] = {0};   // Confusion
matrices
    float train_accuracy = 0.0, test_accuracy = 0.0;               // Accuracy for
training and testing
    double train_error_rate = 0.0, test_error_rate = 0.0;          // Error rates
    int correct_train = 0, correct_test = 0;                       // Correctly
classified samples

    // Initialize random seed for shuffling
    srand(time(NULL));

    // Load the dataset from the file
    load_dataset("tic-tac-toe.data", dataset, &dataset_size);

    // Shuffle the dataset to ensure random distribution
    shuffle_dataset(dataset, dataset_size);

    // Split the dataset into training (80%) and testing (20%) sets
      decision_tree_split_dataset(dataset, dataset_size, train_set, &train_size,
test_set, &test_size, 0.8);

    // Build the decision tree using the training data
    tree = build_tree(train_set, train_size, 0);

    // Write position probabilities (weights) to a file
                    calculate_position_probabilities(dataset, dataset_size,
"DecisionTree_ML/DTweights.txt");
```

```
    // Clear the output file before appending results
    FILE *file = fopen("DecisionTree_ML/DTconfusion_matrix.txt", "w");
    if (file) fclose(file);

    // Evaluate the decision tree on the training data
    train_accuracy = evaluate_with_randomness(tree, train_set, train_size, train_confusion);
    correct_train = (int)(train_accuracy * train_size);
    display_confusion_matrix(train_confusion, "DecisionTree_ML/DTconfusion_matrix.txt", "Training");
    // printf("Training Accuracy: %.2f%% (%d/%d)\n", train_accuracy * 100, correct_train, train_size);
    write_accuracy_to_file("DecisionTree_ML/DTconfusion_matrix.txt", "Training", train_accuracy, correct_train, train_size);

    // Calculate error rate for the training data
    train_error_rate = calculate_error_rate(tree, train_set, train_size, train_confusion);
    // printf("Training Error Rate: %.2f%%\n", train_error_rate);
    file = fopen("DecisionTree_ML/DTconfusion_matrix.txt", "a");
    if (file) {
        fprintf(file, "Training Error Rate: %.2f%%\n", train_error_rate);
        fclose(file);
    }

    // Evaluate the decision tree on the testing data
    test_accuracy = evaluate_with_randomness(tree, test_set, test_size, test_confusion);
    correct_test = (int)(test_accuracy * test_size);
    display_confusion_matrix(test_confusion, "DecisionTree_ML/DTconfusion_matrix.txt", "Testing");
    // printf("Testing Accuracy: %.2f%% (%d/%d)\n", test_accuracy * 100, correct_test, test_size);
    write_accuracy_to_file("DecisionTree_ML/DTconfusion_matrix.txt", "Testing", test_accuracy, correct_test, test_size);

    // Calculate error rate for the testing data
    test_error_rate = calculate_error_rate(tree, test_set, test_size, test_confusion);
    // printf("Testing Error Rate: %.2f%%\n", test_error_rate);
    file = fopen("DecisionTree_ML/DTconfusion_matrix.txt", "a");
    if (file) {
        fprintf(file, "Testing Error Rate: %.2f%%\n", test_error_rate);
        fclose(file);
    }
```

```c
}

// Function to load the dataset from a file
void load_dataset(const char *filename, DataRow dataset[], int *dataset_size) {
    // Open the dataset file in read mode
    FILE *file = fopen(filename, "r");
    if (!file) {
        // Print an error message if the file cannot be opened
        perror("Failed to open file");
        exit(1); // Exit the program with an error code
    }
    char line[256];    // Buffer to store each line of the file
    *dataset_size = 0; // Initialize the dataset size to zero
    // Read the file line by line
    while (fgets(line, sizeof(line), file)) {
        // Split the current line into tokens using ',' as the delimiter
        char *token = strtok(line, ",");
        for (int i = 0; i < NUM_FEATURES; i++) {
            // Map the token value to corresponding feature representation
            if (strcmp(token, "x") == 0)
                dataset[*dataset_size].features[i] = 1; // Assign 1 for 'x'
            else if (strcmp(token, "o") == 0)
                dataset[*dataset_size].features[i] = 2; // Assign 2 for 'o'
            else
                dataset[*dataset_size].features[i] = 0; // Assign 0 for blank space
            // Move to the next token in the line
            token = strtok(NULL, ",");
        }
        // Assign the label based on the last token in the line
            dataset[*dataset_size].label  =  (strcmp(token,  "positive\n")  ==  0)  ?
DT_POSITIVE : DT_NEGATIVE;
        // Increment the dataset size after processing each line
        (*dataset_size)++;
    }
    // Close the file after reading is complete
    fclose(file);
}

// Function to shuffle the dataset
void shuffle_dataset(DataRow dataset[], int size) {
    for (int i = size - 1; i > 0; i--) {
        int j = rand() % (i + 1); // Generate random index
        DataRow temp = dataset[i]; // Swap elements
        dataset[i] = dataset[j];
        dataset[j] = temp;
```

```
    }
}

// Function to split dataset into training and testing sets
void decision_tree_split_dataset(DataRow dataset[], int dataset_size, DataRow
train_set[], int *train_size, DataRow test_set[], int *test_size, float
train_ratio) {
    int train_limit = (int)(dataset_size * train_ratio); // Calculate training data
size
    *train_size = 0;
    *test_size = 0;
    for (int i = 0; i < dataset_size; i++) {
        if (i < train_limit) {
            train_set[(*train_size)++] = dataset[i];
        } else {
            test_set[(*test_size)++] = dataset[i];
        }
    }
}

// Function to build the decision tree with depth limit
DecisionTreeNode *build_tree(DataRow dataset[], int size, int depth) {
    int positives = 0, negatives = 0;

    // Count positive and negative labels in the dataset
    for (int i = 0; i < size; i++) {
        if (dataset[i].label == DT_POSITIVE)
            positives++; // Increment positive count for positive labels
        else
            negatives++; // Increment negative count for negative labels
    }

      // Stop conditions: max depth reached or node is pure (only positives or
negatives)
    if (depth >= MAX_DEPTH || positives == 0 || negatives == 0) {
        // Allocate memory for a leaf node
                                DecisionTreeNode *leaf = (DecisionTreeNode
*)malloc(sizeof(DecisionTreeNode));
        leaf->is_leaf = 1; // Mark the node as a leaf
         leaf->prediction = (positives > negatives) ? DT_POSITIVE : DT_NEGATIVE; //
Predict the majority class
        leaf->left = leaf->right = NULL; // Leaf nodes have no children
        return leaf; // Return the leaf node
    }
```

```
    // Variables to track the best feature and threshold for splitting
    int best_feature = -1, best_threshold = -1;
     float best_gini = 1.0;                    // Initialize the best Gini impurity
to the highest value
    DataRow left[MAX_ROWS], right[MAX_ROWS];  // Temporary arrays for storing split
datasets
    int left_size = 0, right_size = 0;        // Sizes of left and right subsets

    // Iterate over all features and possible thresholds to find the best split
    for (int feature_index = 0; feature_index < NUM_FEATURES; feature_index++) {
        for (int threshold = 0; threshold <= 2; threshold++) {
            // Calculate the Gini impurity for the current split
                 float gini = calculate_gini_index(dataset, size, feature_index,
threshold);
            if (gini < best_gini) {
                 // Update the best Gini impurity, feature, and threshold if this
split is better
                best_gini = gini;
                best_feature = feature_index;
                best_threshold = threshold;
            }
        }
    }

     // Split the dataset into left and right subsets based on the best feature and
threshold
      decision_tree_split_data(dataset, size, best_feature, best_threshold, left,
&left_size, right, &right_size);

    // Allocate memory for the new decision tree node
    DecisionTreeNode *node = (DecisionTreeNode *)malloc(sizeof(DecisionTreeNode));
     node->is_leaf = 0;                    // Mark the node as an internal (non-leaf)
node
    node->feature_index = best_feature; // Store the best feature for splitting
    node->threshold = best_threshold;   // Store the best threshold for splitting
    // Recursively build the left subtree using the left subset
    node->left = build_tree(left, left_size, depth + 1);
    // Recursively build the right subtree using the right subset
    node->right = build_tree(right, right_size, depth + 1);

    return node; // Return the newly created decision tree node
}

// Function to evaluate the decision tree with randomness and update the confusion
matrix
```

```c
float evaluate_with_randomness(DecisionTreeNode *root, DataRow dataset[], int size,
int confusion_matrix[2][2]) {
    int correct_predictions = 0;          // Counter for correct predictions
    // Initialize confusion matrix to zero
    for (int i = 0; i < 2; i++) {         // Iterate through rows of the matrix
        for (int j = 0; j < 2; j++) {     // Iterate through columns of the matrix
            confusion_matrix[i][j] = 0;   // Set each cell to zero
        }
    }
    // Iterate through the dataset to populate the confusion matrix
    for (int i = 0; i < size; i++) {
        int prediction = predict_with_randomness(root, dataset[i].features); // Get
prediction from the decision tree
           int  actual  =  dataset[i].label;  //  Retrieve  the  actual  label  from  the
dataset

        if (actual == DT_POSITIVE && prediction == DT_POSITIVE) {
            confusion_matrix[0][0]++; // Increment True Positive (TP)
            correct_predictions++;    // Increment correct predictions count
        } else if (actual == DT_NEGATIVE && prediction == DT_NEGATIVE) {
            confusion_matrix[1][1]++; // Increment True Negative (TN)
            correct_predictions++;    // Increment correct predictions count
        } else if (actual == DT_NEGATIVE && prediction == DT_POSITIVE) {
            confusion_matrix[1][0]++; // Increment False Positive (FP)
        } else if (actual == DT_POSITIVE && prediction == DT_NEGATIVE) {
            confusion_matrix[0][1]++; // Increment False Negative (FN)
        }
    }
    // Return the accuracy as the ratio of correct predictions to the total dataset
size
    return (float)correct_predictions / size; // Calculate accuracy
}

// Function to make predictions with randomness in the decision tree
int predict_with_randomness(DecisionTreeNode *node, int features[]) {
    if (!node) {
        return DT_NEGATIVE; // Default prediction if the node is NULL
    }
    if (node->is_leaf) {
        // Introduce randomness to the prediction
          if  ((float)rand()  /  RAND_MAX  <  RANDOMNESS_FACTOR)  {  //  Compare  a  random
value to RANDOMNESS_FACTOR
              return  (node->prediction == DT_POSITIVE) ? DT_NEGATIVE : DT_POSITIVE;
// Flip the prediction randomly
        }
```

```c
        return node->prediction; // Return the prediction stored in the leaf node
    }
    // Traverse the decision tree based on the feature threshold
      if  (features[node->feature_index]  <=  node->threshold)  {  //  Compare  feature
value with threshold
          return  predict_with_randomness(node->left,  features);  //  Traverse  left
subtree if the condition is met
    } else {
          return  predict_with_randomness(node->right,  features);  //  Traverse right
subtree otherwise
    }
}

// Function to display and log the confusion matrix to a file
void  display_confusion_matrix(int  confusion_matrix[2][2],  const  char  *filename,
const char *dataset_type) {
    FILE *file = fopen(filename, "a"); // Open file in append mode
    if (!file) {
        perror("Failed to open confusion matrix file");
        return;
    }

    // Extract TP, TN, FP, FN from the confusion matrix
    int TP = confusion_matrix[0][0];
    int FP = confusion_matrix[1][0];
    int TN = confusion_matrix[1][1];
    int FN = confusion_matrix[0][1];

    // Print confusion matrix and metrics to console
    /*
    printf("\nDecision Tree %s Confusion Matrix:\n", dataset_type);
    printf("    True Positive (TP): %d\n", TP);
    printf("    False Positive (FP): %d\n", FP);
    printf("    True Negative (TN): %d\n", TN);
    printf("    False Negative (FN): %d\n", FN);
    printf("\nConfusion Matrix:\n");
    printf("                Predicted Positive    Predicted Negative\n");
    printf("Actual Positive        %10d%20d\n", TP, FN);
    printf("Actual Negative        %10d%20d\n", FP, TN);
    printf("---------------------------------------------------------\n");
    */

    // Write confusion matrix and metrics to file
    fprintf(file, "\nDecision Tree %s Confusion Matrix:\n", dataset_type);
    fprintf(file, "    True Positive (TP): %d\n", TP);
```

```
    fprintf(file, "    False Positive (FP): %d\n", FP);
    fprintf(file, "    True Negative (TN): %d\n", TN);
    fprintf(file, "    False Negative (FN): %d\n", FN);
    fprintf(file, "\nConfusion Matrix:\n");
    fprintf(file, "                 Predicted Positive    Predicted Negative\n");
    fprintf(file, "Actual Positive        %10d%20d\n", TP, FN);
    fprintf(file, "Actual Negative        %10d%20d\n", FP, TN);
    fprintf(file, "---------------------------------------------------------\n");

    fclose(file); // Close the file properly
}

// Function to write accuracy results to a file
void write_accuracy_to_file(const char *filename, const char *dataset_type, float
accuracy, int correct, int total) {
    FILE *file = fopen(filename, "a"); // Open file in append mode to add data
    if (!file) { // Check if the file was opened successfully
         perror("Failed to open file for writing accuracy"); // Print error message
if file open fails
        return; // Exit the function if file cannot be opened
    }

      // Write the dataset type, accuracy percentage, and correct classification
counts to the file
     fprintf(file, "%s Accuracy: %.2f%% (%d/%d)\n", dataset_type, accuracy * 100,
correct, total);
    fclose(file); // Close the file to save changes
}

// Function to free the memory allocated for the decision tree
void free_tree(DecisionTreeNode *node) {
     if (node == NULL) return; // Base case: If the node is NULL, nothing to free,
so return
    // Recursively free memory for the left subtree
    free_tree(node->left);
    // Recursively free memory for the right subtree
    free_tree(node->right);
    free(node); // Free the current node's memory
}

// Function to calculate the Gini index for a potential split
float calculate_gini_index(DataRow dataset[], int size, int feature_index, int
threshold) {
     DataRow left[MAX_ROWS], right[MAX_ROWS]; // Temporary arrays to store left and
right branches
```

```
    int left_size = 0, right_size = 0; // Initialize sizes of left and right
branches

    // Split the dataset into left and right branches based on the feature and
threshold
        decision_tree_split_data(dataset, size, feature_index, threshold, left,
&left_size, right, &right_size);

    // If either branch is empty, return the worst Gini index (1.0) to discourage
this split
    if (left_size == 0 || right_size == 0) return 1.0;
    // Initialize Gini indices for the left and right branches
    float gini_left = 1.0, gini_right = 1.0;
    int positives_left = 0, positives_right = 0; // Counters for positive labels in
each branch
    // Count positive labels in the left branch
    for (int i = 0; i < left_size; i++) {
        if (left[i].label == DT_POSITIVE) positives_left++;
    }
    // Count positive labels in the right branch
    for (int i = 0; i < right_size; i++) {
        if (right[i].label == DT_POSITIVE) positives_right++;
    }

    // Calculate the probability of positive labels in the left branch
    float prob_left = (float)positives_left / left_size;
    // Calculate the Gini index for the left branch
     gini_left = 1.0 - (prob_left * prob_left) - ((1.0 - prob_left) * (1.0 -
prob_left));
    // Calculate the probability of positive labels in the right branch
    float prob_right = (float)positives_right / right_size;
    // Calculate the Gini index for the right branch
     gini_right = 1.0 - (prob_right * prob_right) - ((1.0 - prob_right) * (1.0 -
prob_right));

    // Return the weighted average of the Gini indices for both branches
    return ((gini_left * left_size) + (gini_right * right_size)) / size;
}

// Function to split the dataset into left and right branches
void decision_tree_split_data(DataRow dataset[], int size, int feature_index, int
threshold, DataRow left[], int *left_size, DataRow right[], int *right_size) {
    *left_size = 0; // Initialize the size of the left branch to zero
    *right_size = 0; // Initialize the size of the right branch to zero

```

```
    // Iterate through the dataset to classify each data point into the left or
right branch
    for (int i = 0; i < size; i++) {
           if  (dataset[i].features[feature_index]  <=  threshold)  {  // Check if the
feature value is less than or equal to the threshold
               left[(*left_size)++] = dataset[i]; // Add the data point to the left
branch and increment its size
        } else { // Otherwise, add the data point to the right branch
             right[(*right_size)++] = dataset[i]; // Add the data point to the right
branch and increment its size
        }
    }
}

// Function to predict the best move for the current player based on the decision
tree
void   dt_predict_best_move(DecisionTreeNode   *tree,   char   board[3][3],   char
current_player, int *best_row, int *best_col) {
    if (!tree) {                 // Check if the decision tree is not initialized
             printf("Error: Decision tree is not initialized!\n"); // Print error
message
        return;                  // Exit the function
    }

     int features[NUM_FEATURES]; // Array to store the board features as numerical
values
     int max_positive_prob = -1; // Variable to track the highest probability for a
positive outcome
     *best_row = -1;              // Initialize the best_row variable to an invalid
value
     *best_col = -1;              // Initialize the best_col variable to an invalid
value
     int attempts = 0;           // Counter to limit the number of attempts to find
the best move

    // Convert the 3x3 board into a feature array
    for (int i = 0; i < 3; i++) { // Loop through each row
        for (int j = 0; j < 3; j++) { // Loop through each column
            if (board[i][j] == 'x') features[i * 3 + j] = 1; // Map 'x' to 1
            else if (board[i][j] == 'o') features[i * 3 + j] = 2; // Map 'o' to 2
            else features[i * 3 + j] = 0; // Map empty cells ('b') to 0
        }
    }

    // Attempt to find the best move within a maximum of 5 iterations
```

```
    for (attempts = 0; attempts < 5; attempts++) {
           int temp_row = -1, temp_col = -1; // Temporary variables to store the
coordinates of the current best move

        // Iterate over all cells of the board
        for (int i = 0; i < 3; i++) { // Loop through rows
            for (int j = 0; j < 3; j++) { // Loop through columns
                if (board[i][j] == 'b') { // Check if the current cell is empty
                       // Temporarily set the current player's move in the feature
array
                      features[i * 3 + j] = (current_player == 'x') ? 1 : 2; // Map
'x' to 1 and 'o' to 2

                    // Use the decision tree to predict the outcome of this move
                    int prediction = predict_with_randomness(tree, features);

                      // If the prediction is positive and better than the current
best, update the best move
                      if (prediction == DT_POSITIVE && (max_positive_prob == -1 ||
prediction > max_positive_prob)) {
                        temp_row = i; // Update the row of the best move
                        temp_col = j; // Update the column of the best move
                             max_positive_prob = prediction; // Update the highest
positive probability
                    }

                    // Reset the feature array for the current cell back to empty
                    features[i * 3 + j] = 0;
                }
            }
        }

        // If a valid positive move is found, update best_row and best_col and exit
the loop
        if (temp_row != -1 && temp_col != -1) {
            *best_row = temp_row; // Set the best move's row
            *best_col = temp_col; // Set the best move's column
            return; // Exit the function
        }
    }

    // If no positive move is found after 5 attempts, choose any random empty cell
    for (int i = 0; i < 3; i++) { // Loop through rows
        for (int j = 0; j < 3; j++) { // Loop through columns
            if (board[i][j] == 'b') { // Check if the cell is empty
```

```
                *best_row = i; // Assign the row of the random empty cell
                *best_col = j; // Assign the column of the random empty cell
                return; // Exit the function
            }
        }
    }
}

// Function to recursively print the structure of the decision tree
void print_tree(DecisionTreeNode *node, int depth) {
    if (!node) {
        // Base case: If the node is NULL, return
        return;
    }

    if (node->is_leaf) {
        // Print leaf node details
        // printf("%*sLeaf: Prediction = %d\n", depth * 4, "", node->prediction);
    } else {
        // Print internal node details
            // printf("%*sNode: Feature = %d, Threshold = %d\n", depth * 4, "",
node->feature_index, node->threshold);
        print_tree(node->left, depth + 1); // Recur for the left child
        print_tree(node->right, depth + 1); // Recur for the right child
    }
}

// Function to calculate and save position probabilities for the dataset
void calculate_position_probabilities(DataRow dataset[], int dataset_size, const
char *filename) {
      int positive_count = 0, negative_count = 0; // Counters for the number of
positive and negative samples
      int position_count[NUM_FEATURES][3][2] = {0}; // Array to store counts of
symbols ('x', 'o', empty) for each position and class

     // Count occurrences of each symbol ('x', 'o', empty) in every position for
each class
    for (int i = 0; i < dataset_size; i++) {
            if (dataset[i].label == DT_POSITIVE) positive_count++; // Increment
positive count if label is positive
        else negative_count++; // Increment negative count otherwise

        // Iterate over each feature (board position)
        for (int j = 0; j < NUM_FEATURES; j++) {
```

```
                                              if    (dataset[i].features[j]    ==    1)
position_count[j][0][dataset[i].label]++; // Count 'x'
                                        else   if    (dataset[i].features[j]    ==    2)
position_count[j][1][dataset[i].label]++; // Count 'o'
            else position_count[j][2][dataset[i].label]++; // Count empty spaces
        }
    }

    // Open the file to save calculated probabilities
    FILE *file = fopen(filename, "w");
    if (!file) { // Check if the file was successfully opened
        perror("Failed to open file to save weights");
        return; // Exit the function
    }

    // Write the class probabilities to the file
    fprintf(file, "Class Probabilities:\n");
      fprintf(file, "   Positive:  P(Positive)  = %.4f\n",  (double)positive_count  /
dataset_size); // Probability of positive class
      fprintf(file, "   Negative:  P(Negative)  = %.4f\n",  (double)negative_count  /
dataset_size); // Probability of negative class
    fprintf(file, "---------------------------------------------\n");

    // Write position-wise probabilities to the file
    for (int i = 0; i < NUM_FEATURES; i++) { // Loop through each board position
        fprintf(file, "Position %d:\n", i + 1); // Position label (1-indexed)
        fprintf(file, "  Symbol | P(Symbol | Positive) | P(Symbol | Negative)\n");
        fprintf(file, "  -------|----------------------|----------------------\n");

        const char *symbols[] = {"x", "o", "b"}; // Define symbols corresponding to
feature values
        for (int j = 0; j < 3; j++) { // Loop through symbols
                                  double    p_positive    =    (positive_count    >    0)    ?
(double)position_count[i][j][DT_POSITIVE] / positive_count : 0.0; // Probability of
symbol given positive class
                                  double    p_negative    =    (negative_count    >    0)    ?
(double)position_count[i][j][DT_NEGATIVE] / negative_count : 0.0; // Probability of
symbol given negative class
              fprintf(file, "   %-6s | %-20.4f | %-20.4f\n", symbols[j], p_positive,
p_negative); // Write probabilities to file
        }
              fprintf(file,  "---------------------------------------------\n");  //
Separator line for readability
    }
```

```
    fclose(file); // Close the file after writing
      printf("Weights updated and saved to %s\n", filename); // Notify user of
success
}

// Function to calculate the error rate of the decision tree
double calculate_error_rate(DecisionTreeNode *root, DataRow dataset[], int size,
int confusion_matrix[2][2]) {
    int error_count = 0; // Counter to track the number of incorrect predictions

    // Iterate over the dataset to compare predictions with actual labels
    for (int i = 0; i < size; i++) {
        int prediction = predict_with_randomness(root, dataset[i].features); // Get
the predicted label from the decision tree
        int actual = dataset[i].label; // Get the actual label from the dataset

        if (prediction != actual) { // Check if the prediction is incorrect
            error_count++; // Increment the error count
        }
    }

    // Calculate and return the error rate as a percentage of total samples
    return ((double)error_count / size) * 100;
}
```

12.15. confusionmatrix.py (Decision Tree)

```
import matplotlib.pyplot as plt  # Import Matplotlib for plotting
import numpy as np               # Import NumPy for numerical operations
import os                        # Import OS module for file path handling

# Function to plot confusion matrix
def plot_combined_confusion_matrix(
    train_matrix, train_TP, train_TN, train_FP, train_FN,
    test_matrix, test_TP, test_TN, test_FP, test_FN
):
    # Define class labels
    classes = ['Positive', 'Negative']

    # Create a single figure with 2 subplots for training and testing matrices
    fig, axes = plt.subplots(1, 2, figsize=(12, 6))

    # Plot Training Confusion Matrix
     cax1 = axes[0].matshow(train_matrix, cmap="Blues")  # Display the matrix as a
color-coded plot
    fig.colorbar(cax1, ax=axes[0])                          # Add color bar to the plot
      axes[0].set_title("Training Confusion Matrix")          # Set title for the
training subplot
    axes[0].set_xticks([0, 1])                              # Set x-axis ticks
    axes[0].set_yticks([0, 1])                              # Set y-axis ticks
    axes[0].set_xticklabels(classes)                        # Label x-axis ticks
    axes[0].set_yticklabels(classes)                        # Label y-axis ticks

    # Annotate each cell with its value
    for (i, j), val in np.ndenumerate(train_matrix):
        axes[0].text(j, i, f"{val}", ha='center', va='center', color='black')

    # Add textual description of TP, TN, FP, FN for training data
    axes[0].set_xlabel(
        f"TP: {train_TP}, TN: {train_TN}\nFP: {train_FP}, FN: {train_FN}",
        fontsize=10
    )
    axes[0].set_ylabel("Actual")  # Label the y-axis

    # Plot Testing Confusion Matrix
     cax2 = axes[1].matshow(test_matrix, cmap="Blues")  # Display the matrix as a
color-coded plot
    fig.colorbar(cax2, ax=axes[1])                         # Add color bar to the plot
     axes[1].set_title("Testing Confusion Matrix")        # Set title for the testing
subplot
```

```
    axes[1].set_xticks([0, 1])                         # Set x-axis ticks
    axes[1].set_yticks([0, 1])                         # Set y-axis ticks
    axes[1].set_xticklabels(classes)                   # Label x-axis ticks
    axes[1].set_yticklabels(classes)                   # Label y-axis ticks

    # Annotate each cell with its value
    for (i, j), val in np.ndenumerate(test_matrix):
        axes[1].text(j, i, f"{val}", ha='center', va='center', color='black')

    # Add textual description of TP, TN, FP, FN for testing data
    axes[1].set_xlabel(
        f"TP: {test_TP}, TN: {test_TN}\nFP: {test_FP}, FN: {test_FN}",
        fontsize=10
    )
    axes[1].set_ylabel("Actual")  # Label the y-axis

    # Adjust layout to avoid overlap and save the combined image
    plt.tight_layout()
     output_path = os.path.join(os.getcwd(), "DT_Confusion_Matrix.png")   # Define
output path
     plt.savefig(output_path, bbox_inches='tight')                     # Save the
plot to a file
     print(f"Saved combined plot to {output_path}")                     # Notify
user of the saved file


# Function to read confusion matrix and extract TP, TN, FP, FN
def read_confusion_matrix(filename, set_name):
    # Open the confusion matrix file
    with open(filename, "r") as file:
        lines = file.readlines()  # Read all lines from the file

    # Locate the matrix for the specified set (Training or Testing)
    for i, line in enumerate(lines):  # Iterate over each line in the file with its
index
         if f"Decision Tree {set_name} Confusion Matrix" in line:  # Check if the
current line matches the specified set name
            # Extract TP, TN, FP, FN from the respective lines below the header
             TP = int(lines[i + 1].split(":")[-1].strip())  # Extract True Positive
(TP) value from the next line
            FP = int(lines[i + 2].split(":")[-1].strip())  # Extract False Positive
(FP) value from the line after TP
             TN = int(lines[i + 3].split(":")[-1].strip())  # Extract True Negative
(TN) value from the line after FP
```

```
            FN = int(lines[i + 4].split(":")[-1].strip())  # Extract False Negative
(FN) value from the line after TN

            # Extract the rows of the confusion matrix
            row1 = [int(val) for val in lines[i + 8].strip().split()[-2:]]  # Parse
the first row of the matrix (last two values)
            row2 = [int(val) for val in lines[i + 9].strip().split()[-2:]]  # Parse
the second row of the matrix (last two values)
             matrix = np.array([row1, row2])  # Combine the two rows into a NumPy
array representing the matrix

             return matrix, TP, TN, FP, FN  # Return the confusion matrix and the
extracted metrics

    # Return None if the desired section is not found in the file
    return None, None, None, None, None

# Main logic to read confusion matrix and plot
def main():
    # File containing the confusion matrix
    filename = "DTconfusion_matrix.txt"

    # Print the current working directory for debugging
    print("Current Working Directory:", os.getcwd())

    # Read the Training Confusion Matrix
               train_matrix,    train_TP,    train_TN,    train_FP,    train_FN    =
read_confusion_matrix(filename, "Training")
    if train_matrix is None:  # Check if the matrix was successfully read
        print("Training confusion matrix not found in the file.")
        return

    # Read the Testing Confusion Matrix
                 test_matrix,    test_TP,    test_TN,    test_FP,    test_FN    =
read_confusion_matrix(filename, "Testing")
    if test_matrix is None:  # Check if the matrix was successfully read
        print("Testing confusion matrix not found in the file.")
        return

    # Plot the combined confusion matrix for training and testing
    plot_combined_confusion_matrix(
        train_matrix, train_TP, train_TN, train_FP, train_FN,
        test_matrix, test_TP, test_TN, test_FP, test_FN
    )
```

```
    # Print the files present in the current directory for debugging
    print("Files in current folder:", os.listdir(os.getcwd()))

# Entry point of the script
if __name__ == "__main__":
    main()
```

13. References

1. OpenAI. (2024). ChatGPT (Version 4) [Large language model]. OpenAI. https://chat.openai.com

   We have used AI to generate some descriptions for the problem definition and problem analysis.

2. Ralib website: https://www.raylib.com/
3. Raylib cheatsheet: https://www.raylib.com/cheatsheet/cheatsheet.html

4. Fig.10: How minimax AI anticipates your every move and beats you at your own favorite game | by Adam Maj
5. Fig.11: Game-Playing & Adversarial Search - ppt download slide 7
6. Fig.12: Game-Playing & Adversarial Search - ppt download slide 12