

**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**](#_rr7tbjnpg3vn)

[**2. Problem Analysis 6**](#_er7ei2kg6dtt)

[2.1. User-Friendly Graphical User Interface (GUI) 6](#_h823i075nnu2)

[2.1.1. Raylib 6](#_r087y6ssw5db)

[2.1.2. Raylib Features 6](#_vpxgdjskkjpd)

[2.1.3. Implementing Raylib 7](#_lba90usvssa9)

[2.2. Two-Player Mode 7](#_1qrky37as28n)

[2.3. Single-Player Mode with AI/ML 7](#_pxcirrwst0h9)

[2.4. Training and Evaluation of ML Models 8](#_gz6r1m79d9d9)

[2.5. Hint System 8](#_ewat0pj3441q)

[2.6. Score Tracking 9](#_de2uq8gsburc)

[2.7. Execution Instructions 9](#_lknavg8oxjr4)

[**3. Pseudocode 10**](#_drya42a3ww92)

[3.1. main.c File 10](#_7gso0x9lqdp6)

[3.2. Init.c File 15](#_jpzw6tbuvkor)

[3.3. Update.c File 17](#_7rpo28rnd0d)

[3.4. Draw.c File 20](#_2veyzzol3nfk)

[3.5. Handle.c File 28](#_agvfsco6tesc)

[3.6. AI.c File 30](#_ar9r4g4y1774)

[3.7. Check.c File 33](#_w2ao5h4lfrei)

[3.8. Hint.c File 35](#_o8mb2m4l6f5p)

[3.9. Minimax.c File 36](#_pa9y3beydn4m)

[3.10. data\_processing.c File 38](#_y8q03vuzq6n7)

[3.11. NBmodel.c File 40](#_tbs6coh0onrl)

[3.12. decisiontree.c File 45](#_inml97moviu4)

[**4. Function Descriptions 53**](#_2rdvy7s13ebq)

[4.1. Game Logic Functions 53](#_56aq00nyzqdd)

[4.2. UI and Animation Functions 53](#_15lrgmrujmde)

[4.3. Data Processing Functions 54](#_ecklte6t796y)

[4.4. AI Functions 54](#_8c0u7qt4nbma)

[4.5. Raylib Functions 56](#_t2pep9ecrdl7)

[4.6. Standard Functions 57](#_zb60em3msk7k)

[**5. Program Interfaces 58**](#_fpwbybkomzk1)

[**6. Implementation of Machine Learning (Naive Bayes) (Easy Mode) 63**](#_86gpqgk463tv)

[6.1 Extracting and Processing of Dataset 63](#_dqg2hv2bmppg)

[6.2 Logic of Naive Bayes 64](#_v0p86n7dliji)

[6.3 Implementation of the Model 64](#_63k5451lreys)

[6.4 Evaluation of Model 68](#_g7wniaovegal)

[6.5 Plots and Results (Naive Bayes) 69](#_r4yyu8hqe2id)

[**7. Implementation of Machine Learning (Decision Tree) (Easy Mode) 71**](#_wnrh643aqrzm)

[7.1 Extracting and Processing of Dataset 71](#_a1njzcskexpk)

[7.2 Decision Tree Construction and Training 73](#_p162354ruvcd)

[7.3 Splitting the Dataset 74](#_32r4jcz4pinc)

[7.4 Decision Tree Construction and Training 75](#_sxl73minvnn3)

[7.5 Splitting Nodes and Tree-Building Process (build\_tree) 80](#_mdj562irclsu)

[7.6 Saving Position Probabilities 82](#_9qzpd82732wn)

[7.7 Decision Tree Model Prediction dt\_predict\_best\_move 84](#_grxz2w4fe68t)

[7.8 Evaluation of Decision Tree Model and Confusion Matrix evaluate\_with\_randomness() 86](#_b8w9mxop8y20)

[7.9 Confusion Matrix Plot and Results (Decision Tree) 89](#_34qrihooe1a6)

[**8. Implementation of Minimax Algorithm (Hard Mode) 90**](#_9p2nsnkhqvn6)

[8.1. Purpose and Function 90](#_gvmc9ptfvdhp)

[8.2. Operation of Minimax Algorithm 90](#_e63501iazk7l)

[8.2.1. Recursion 91](#_39m9cc8r01dz)

[8.2.2. Depth-First Search (DFS) Algorithm 91](#_j0rmwi2pyqvc)

[8.2.3. Maximizing and Minimizing Players 91](#_avic9xavs8iq)

[8.2.4. Evaluation and Optimal Move Selection 92](#_apppndx2rlc7)

[8.3. Parameters of Minimax() Function 92](#_gbfdxh2rvns3)

[**9. Comparison & Interesting Findings / Assumptions 93**](#_im55omatbdbw)

[9.1. Findings on Dataset 93](#_atebztpv2rw3)

[9.2. Comparison Between Minimax Algorithm and Decision Tree Model 93](#_nfv39jvkaocj)

[9.2.1 Key Characteristics of Minimax: 93](#_hnsfm0bqvxez)

[9.2.2 Key Characteristics of the Decision Tree Model: 93](#_lsognq3dxdag)

[9.2.3 Limitations 94](#_5r4xmyu9wxon)

[9.2.4 Conclusion 94](#_y4i38twqg43o)

[9.3. Comparison between Minimax Algorithm and Naive Bayes Model 94](#_9hjcfiysc2lt)

[**10. Enhancements 95**](#_bkksd53ybz71)

[10.1. Minimax (Imperfectness) (Medium Mode) 95](#_gr70fox0drvz)

[10.1.1 Depth Limitation 95](#_w1t3r34m0vpr)

[10.2. Minimax (Alpha-Beta Pruning) 95](#_k6lawm4q6enk)

[10.2.1 Alpha and Beta 96](#_g0431t8nfesc)

[10.2.2 Operation of Optimized Minimax 96](#_fe05go7wj81g)

[10.3. Modular Approach 96](#_76isb5qhhqe)

[10.4. Decision Tree Model Prediction with Randomness predict\_with\_randomness 97](#_10mgrml23b9n)

[10.5. Memory Free Allocation for the Decision Tree Model free\_tree() 99](#_hi2hef23s4ec)

[**11. Gauging of Difficulty Levels 101**](#_srze3uzezlpl)

[11.1. Level of Difficulty for Easy Mode 101](#_7icpv0gdzcaf)

[11.2. Level of Difficulty for Medium Mode 102](#_42ot1djrx3r1)

[11.3. Level of Difficulty for Hard Mode 102](#_n62w8bxhlykd)

[**12. Appendix 103**](#_339gfcq6mmt7)

[Source Codes 103](#_7zmk1i91s3yq)

[12.1. Main.c 103](#_vox4gh1ju2ma)

[12.2. AI.c 110](#_fwfk4irb9sw3)

[12.3. Check.c 114](#_26ioo23d39ah)

[12.4. Draw.c 116](#_bxsch8sgt32j)

[12.5. Handle.c 126](#_aq8he99638s)

[12.6. Hint.c 129](#_ioninw3tid4o)

[12.7. Init.c 130](#_jso8dwd84fv2)

[12.8. Minimax.c 132](#_26f6nvukuz5r)

[12.9. Update.c 134](#_efumgjjril29)

[12.10. decisiontree.c 138](#_p8j1tx83hfs7)

[12.11. data\_processing.c 150](#_c03dl66x0zy4)

[12.12. NBmodel.c 152](#_thsk9jtnqvjd)

[12.13. plot\_confusion\_plot.py 158](#_f7jou9khn1ni)

[12.14. decisiontree.c 160](#_t83fkxw6u6us)

[12.15. confusionmatrix.py (Decision Tree) 174](#_6ida6fyeyzgf)

[**13. References 178**](#_ddtd25crc5pf)

# 

# 

# 

# 

# 

# 

# 

# 

# 

# 

# 

# 

# 

# 

# 

# 

# 

# 

# 

# 

# **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.

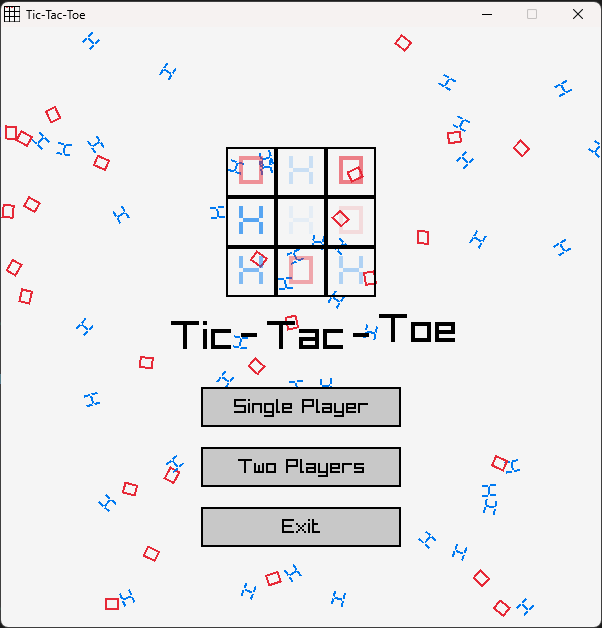
sinf (float x): Computes the sine (specified in radians) of x.

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

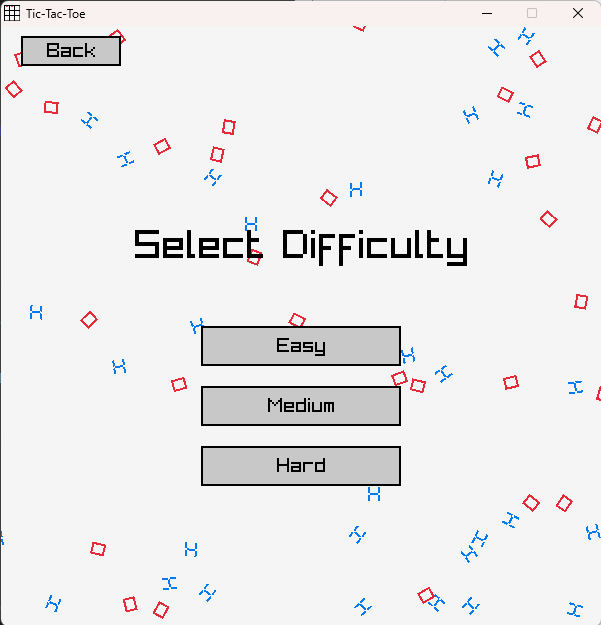
# **5. Program Interfaces**

This section shows the interfaces of the program.

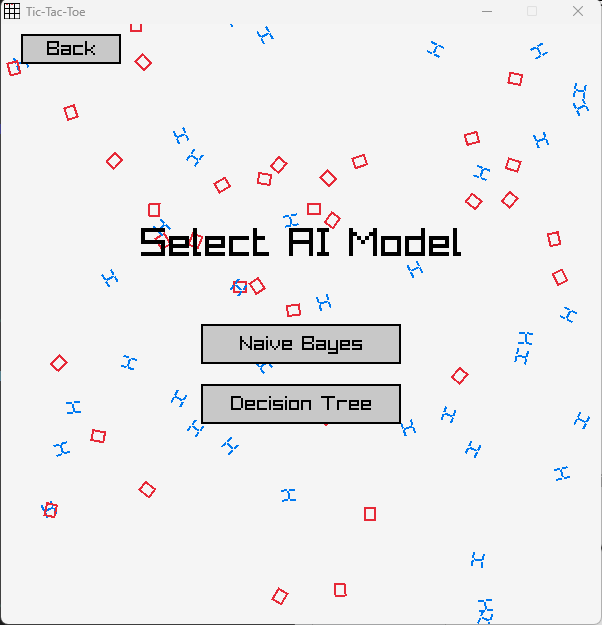
Main Menu:

  
*Fig.1: Main Menu*

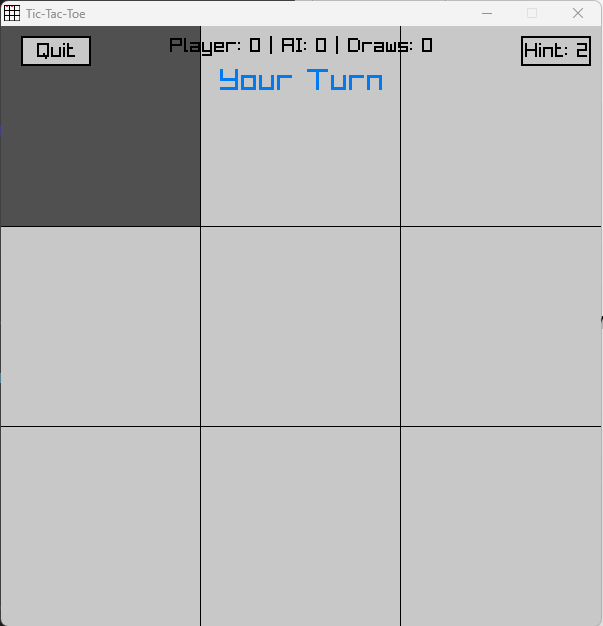
Difficulty Selection:

  
*Fig.2: Difficulty Selection*

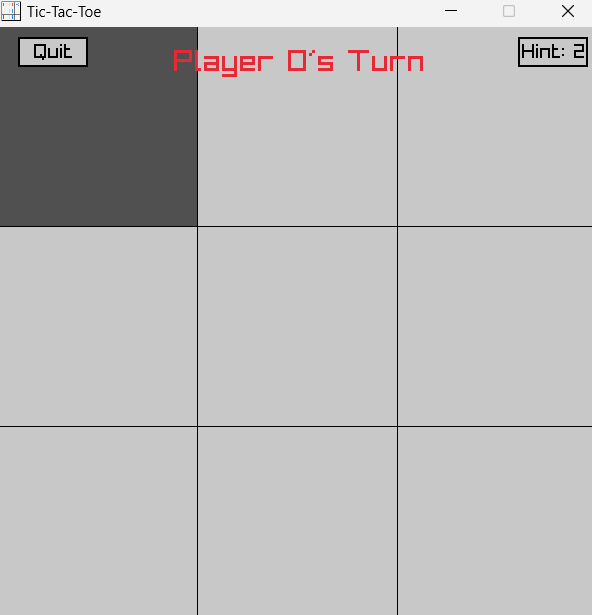
Model Selection for Easy Mode:

  
*Fig.3: Model Selection*

Single-Player Mode:

  
*Fig.4: Single-Player Mode*

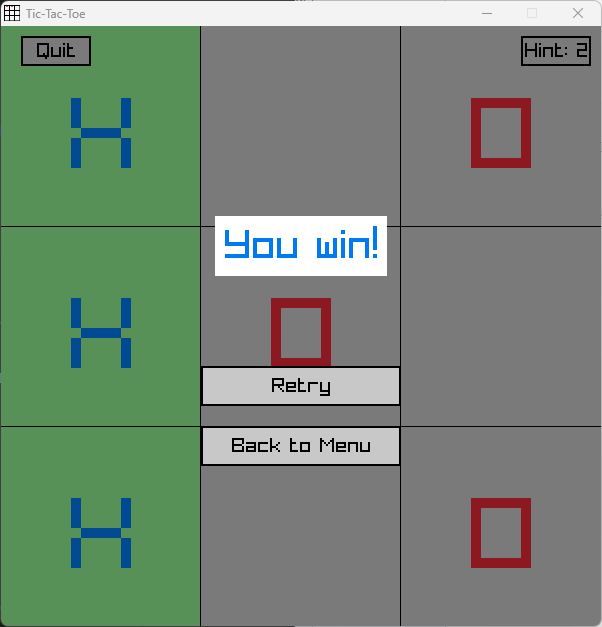
Two-Player Mode:

  
*Fig.5: Two-Player Mode*

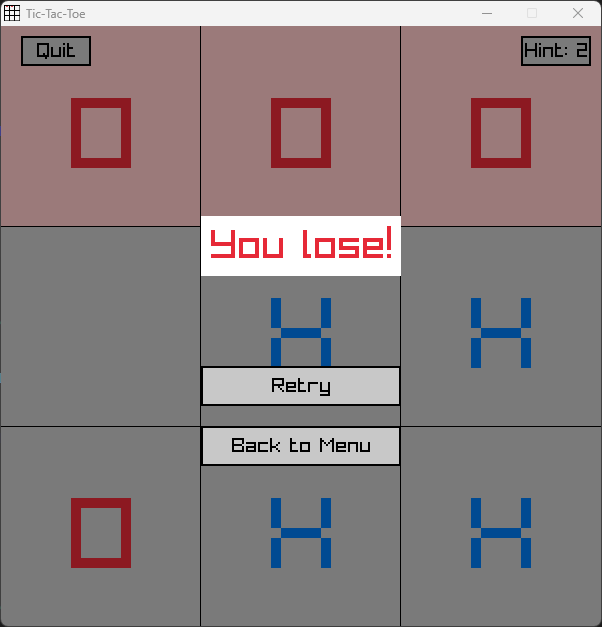
Game Over Screen:

  
*Fig.6: Game Over*

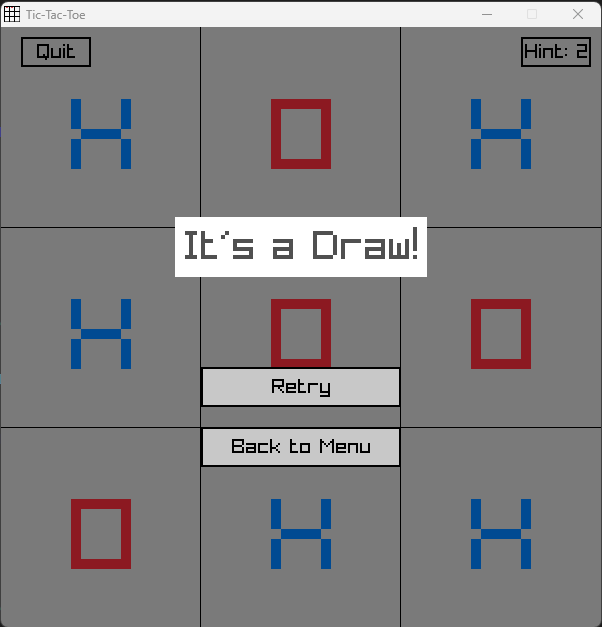
Winning Screen for Single-Player Mode:

  
*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)**

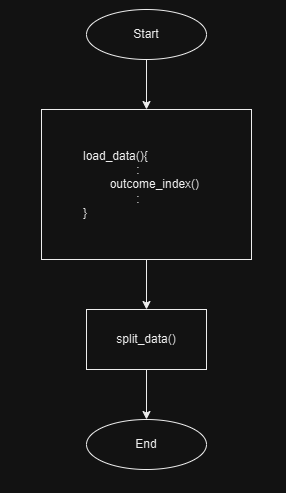
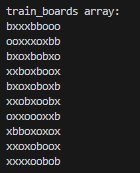
## 6.1 Extracting and Processing of Dataset

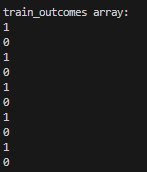
The provided tic-tac-toe.data dataset would be used for training and testing.

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”].
   1. 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).
   2. 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:
   1. 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.

* 1. Split the dataset into 80:20 for the training and testing datasets respectively

Functions calling order: First 10 lines of training datasets:



## **6.2 Logic of Naive Bayes**

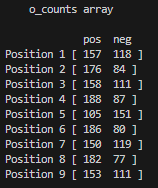
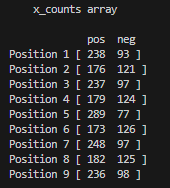
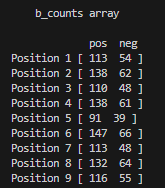
* It assumes the features are independent of one another given the outcome.
* It uses the Bayes’ Theorem to calculate the *Posterior Probability* which is needed to predict the outcome/classes (positive/negative) given the features.
  1. *P(C|X) Posterior Probability formula*: Probability of the outcome C given the board configuration X
  2. *P(C) Prior Probability*: Probability of outcome C (positive/negative) appearing in the dataset
  3. *P(X|C) Conditional Probability(Likelihood)*: Probability of a feature X occurring in a given outcome C
  4. *P(X) Marginal Probability(Evidence)*: Probability of feature X occurring in the dataset regardless of outcome C
  5. 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 Implement**ation **of the** M**odel**

1. Train the model with Naive Bayes algorithm using **train\_NBmodel()**function.
   1. Firstly, it counts the number of occurrences of each feature and each outcome in the dataset.



* 1. Then calculate the *Prior Probability* of each class by taking the count of occurrences of each outcome divided by the size of the dataset
  2. 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:

* 1. 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.

“NBmodel\_weights.txt”:

# 

# 

# 

# 

# 

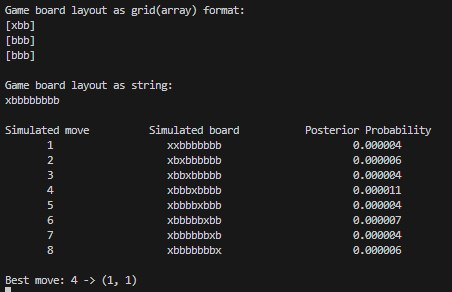
# 

# 

# 

1. 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.

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.

1. 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.
2. Now that the model is trained with Naive Bayes algorithm, it could be used to play against players by:
   1. Having it take in the current state of the game (the current board layout) when it is its turn
   2. Then simulate a move separately in every available(blank) spot and calculate the probability of winning with that respective move.
   3. Ultimately returning (playing) the move that results in the best probability.
3. 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.
   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.
   2. However, the GUI reads and writes the board as an array. Such as,

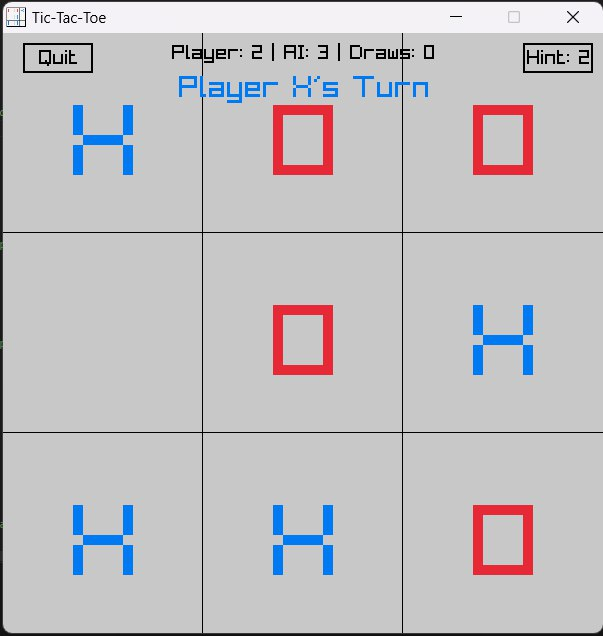
**[‘o’ , ‘b’ , ‘x’]**

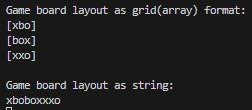
**[‘x’ , ‘o’ , ‘x’]**

**[‘b’ , ‘b’ , ‘o’]**

* 1. 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.
  2. 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:

Current board layout (GUI):

How it looks like in the backend:

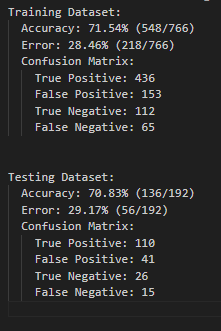
Note: Even though in position 2 there is a ‘b’ in the grid format but the GUI shows a move has been made there, it is because the loop only occurs when it is the AI’s turn to play but the move has not been made yet. The GUI shows the board after it made its move which is position 2.

* 1. Then call the divide()function to convert the best move integer into the row and column indexes of the board.
  2. Return the indexes to the backend to play the move.
  3. 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:

1. 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.
2. Count of wrong predictions made. [error\_count]
3. Using these values we can compute the Accuracy and Probability of error of the trained model.
4. Then save the results in a text file (“NBmodel\_confusion\_matrix.txt”) for plotting of the confusion matrix.

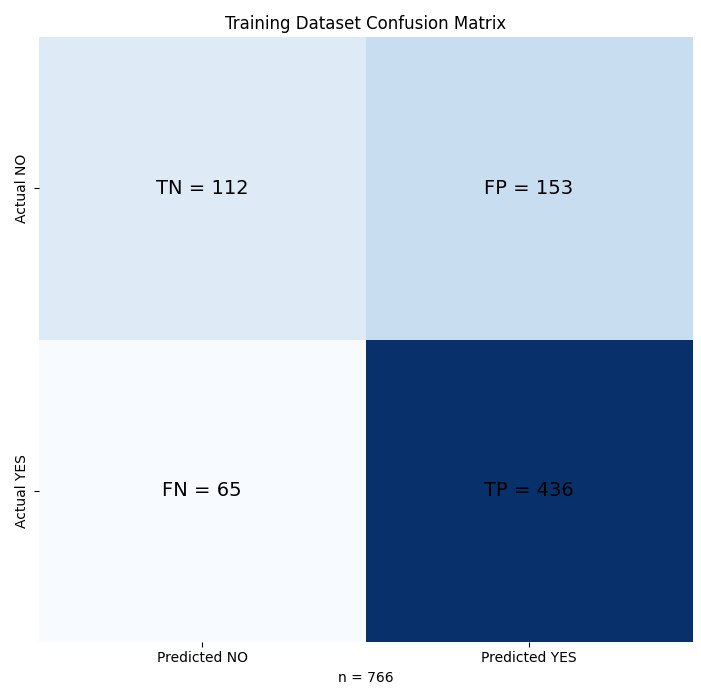
A snapshot of the file:

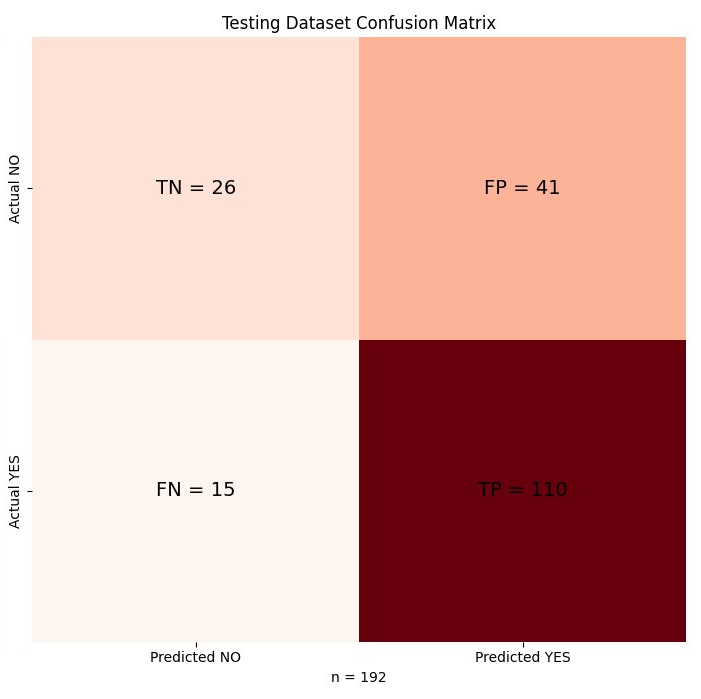
# 

## 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
   1. pip3 install matplotlib
   2. pip3 install seaborn
   3. 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.



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.

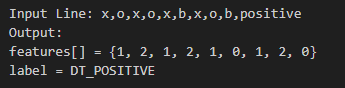
# 7. Implementation of Machine Learning (Decision Tree) (Easy Mode)

## 7.1 **Extracting and Processing of Dataset**

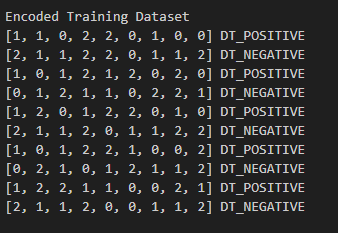
* + We used the provided tic-tac-toe.data dataset for training and testing.

1. **Load the dataset into an array using load\_dataset() function:**

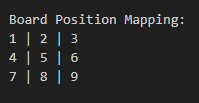
* The dataset is loaded into a 2D array, where each row represents a board layout and the last column represents the respective outcome ("positive" or "negative").
* During loading, the **load\_dataset()** function converts the string outcomes ("positive" or "negative") into corresponding numerical labels.
* **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.

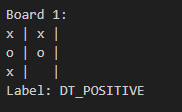


* This conversion ensures memory efficiency and faster training speed which is crucial for numerical operations as they are much faster compared to string operations which leads to better performance during model training and evaluation.



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

* #define NUM\_FEATURES 9 (Feature\_1 to Feature\_9)****

1. Original Board Representations Input: 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.

1. **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.

1. **Functions Calling Order**:

Start

|

|----> load\_dataset()

|

|----> shuffle\_dataset()

|

|----> decision\_tree\_split\_dataset()

|

|----> build\_tree()

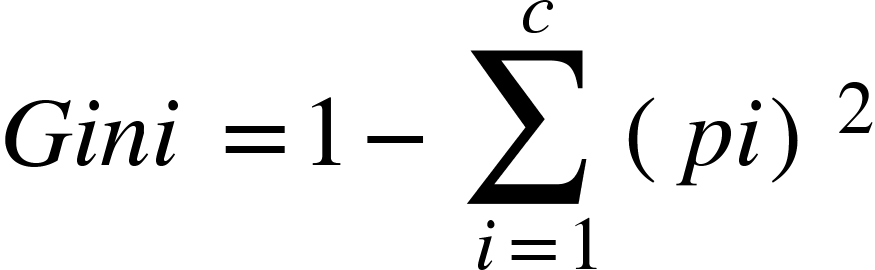
|

|----> 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**:

Where:

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

For our implementation:

* (Positive and Negative).
* ​ Proportion of positive labels in the subset.
* 1 − : 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**.
    1. Feature (feature\_index)
       1. 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.

1. 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.
   * 1. Threshold (threshold): Defines the rule/evaluation for splitting the dataset based on the selected feature.
        1. threshold = 0: Empty cell.

threshold = 1: Cell occupied by 'X'.

threshold = 2: Cell occupied by 'O'.

* + - 1. 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'.

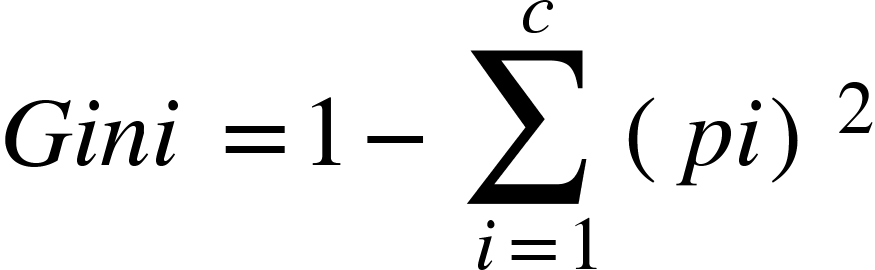
* **Right Subset**: Samples where feature\_value > threshold.

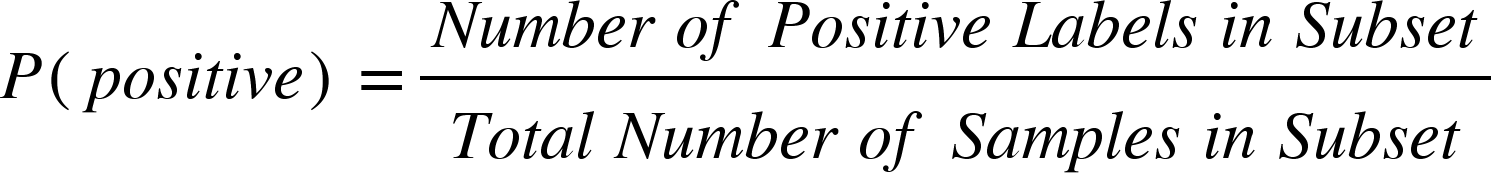
For example, board states where the center cell has 'O'.

* 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.**

1. **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.

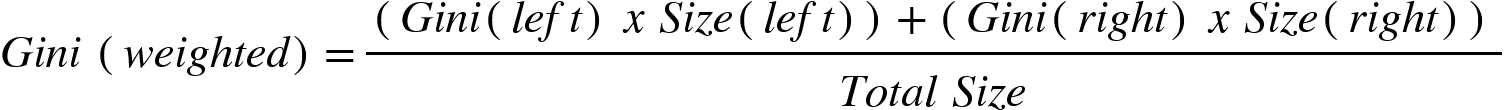
* + **Check for Empty Branches**: If either of the subsets is empty, it returns a Gini value of 1.0, the worst impurity value possible. This penalizes choosing splits that lead to empty branches.
* **Weighted Averaged Gini Index Range**: [0.0, 1.0].
  + A value of **0.0** indicates a perfectly pure split (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 
  + Formula for



* **Calculating Gini Impurity for Each Branch which is computed using**
  + Formula for :

Where

* Finally, the function returns the **weighted average of the Gini indices** for both branches to determine the quality of the split.
  + Formula for



* + 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;

}

}

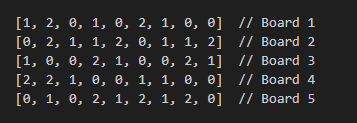
}

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

I will illustrate the splitting process with the following example. For illustration purpose, we take 5 boards for splitting process.

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



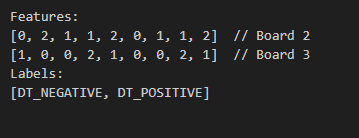
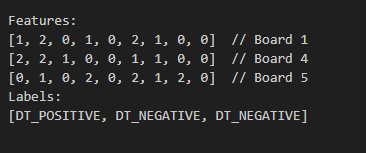
**Labels** (outcomes of each board):



I. Select and evaluate all features (feature\_index = 0 to 8) to find the best split. Now we focus on splitting based on feature\_index = 4 (center cell). Next, for the chosen feature\_index = 4 we calculate the **Gini Index** for each threshold.

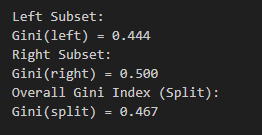
1. **Case 1**: When Threshold = 0, the split logic works by dividing boards where the center cell is empty (feature[4] ≤ 0) and occupied (feature[4] > 0).

**Splited Dataset:**

**Left Subset** (Center ≤ 0):Left **Right Subset** (Center > 0):

Next, we calculate the Gini Index for both subset.

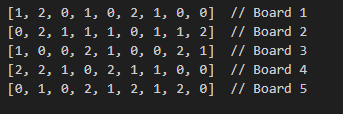
:



1. **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'

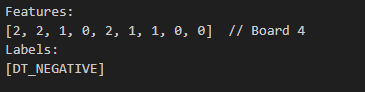
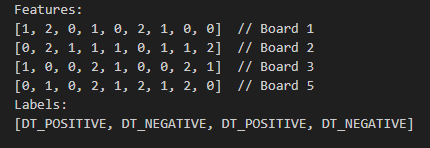
(feature[4] > 1).

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



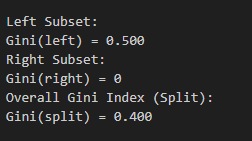
**Labels** (outcomes of each board):



**Left Subset** (Center ≤ 1):Left **Right Subset** (Center > 1):

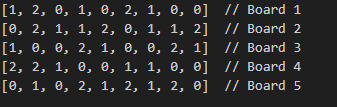
Next, we calculate the Gini Index for both subset.

:



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

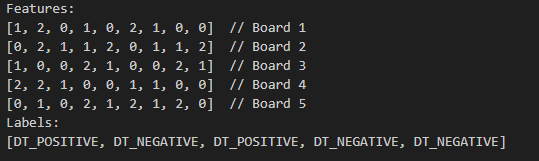
**Features** (each row represents a Tic-Tac-Toe board encoded as integers):



**Labels** (outcomes of each board):

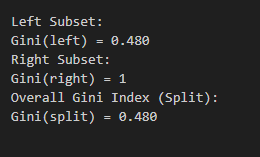


**Left Subset** (Center ≤ 2):Left **Right Subset** (Center > 2): No data

****

Next, we calculate the Gini Index for both subset.

:



(*\*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:

1. 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.
2. Thresholds Define Splitting Rules: The threshold indicates how the feature is utilized such as:
   * threshold = 1 → Splits based on whether a cell contains 'x'.
3. 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:

1. Maximum tree depth is reached (depth >= MAX\_DEPTH) to prevent overfitting.
2. The node becomes "pure," meaning all data points belong to the same class (positives == 0 || negatives == 0), so further splits are unnecessary.
3. 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**

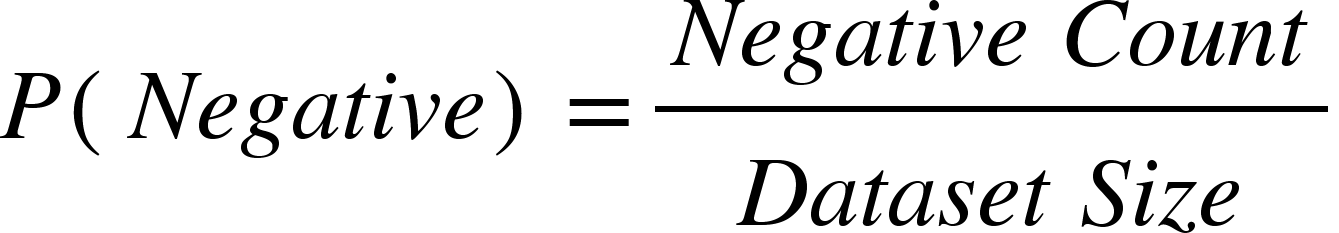
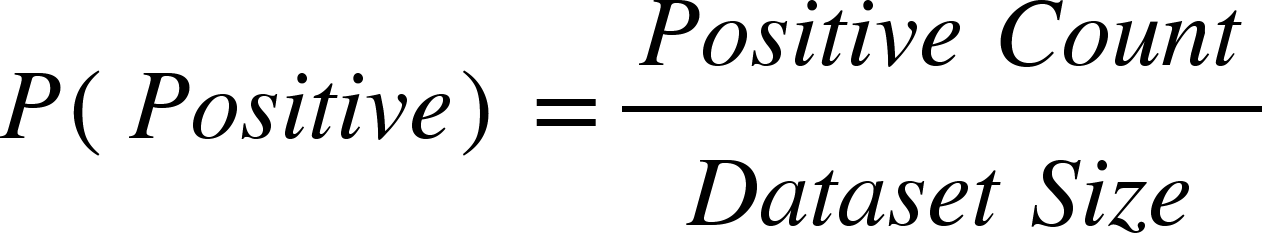
After building the decision tree, we then call the function calculate\_position\_probabilities() that computes the weights for every board position using the TicTacToe dataset. It computes the probability of each of the possible symbols (x, o, empty) at each position given the class label (positive or 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.

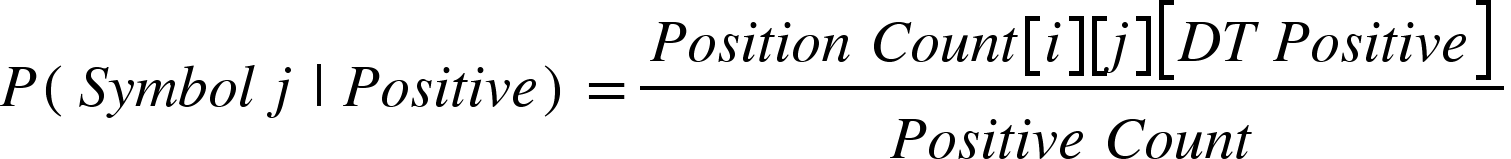
* The variables positive\_count and negative\_count are initialized to zero, serving to count positive and negative outcomes respectively. Additionally, a 3D array position\_count[NUM\_FEATURES][3][2] is initialized to zero to track symbol counts. The dimensions are:
  + NUM\_FEATURES = 9 for a standard 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.

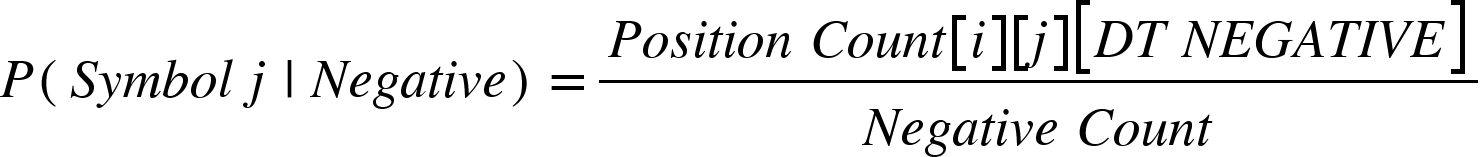
After gathering the counts, then we opens weights file (DTweights.txt) in **write mode** ("w") to save the computed probabilities:

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

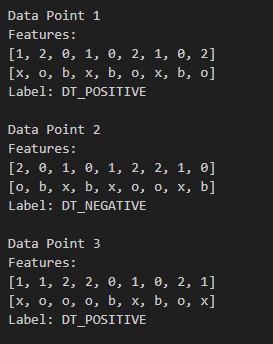
Next, we calculate the **position-wise conditional probabilities** of each symbol (x, o, or empty) for both positive and negative outcomes:

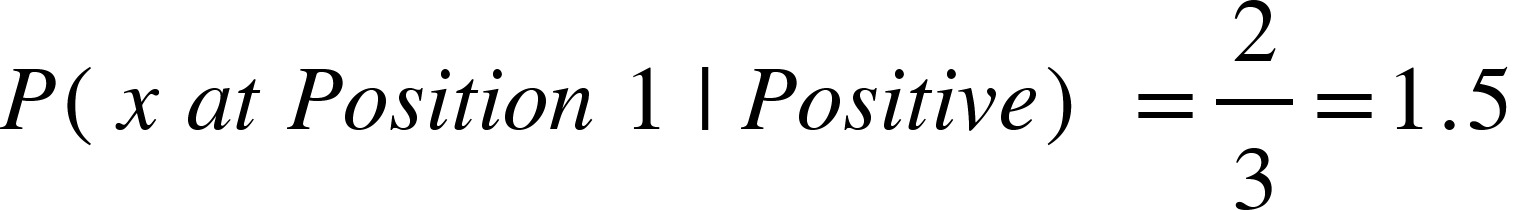
* For each board position (i from 0 to 8), the function iterates through the possible symbols (x, o, and empty).
* For each symbol (j), the function calculates the **conditional probability** given the class:



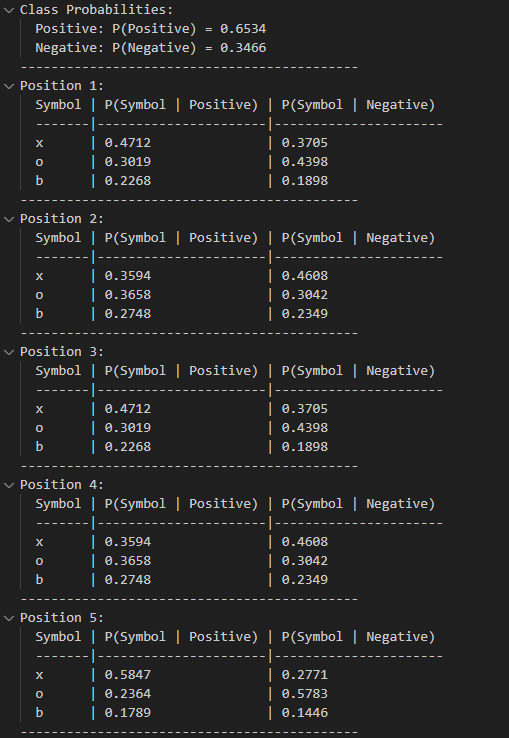


**Example Walkthrough**

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

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.

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.



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;

}

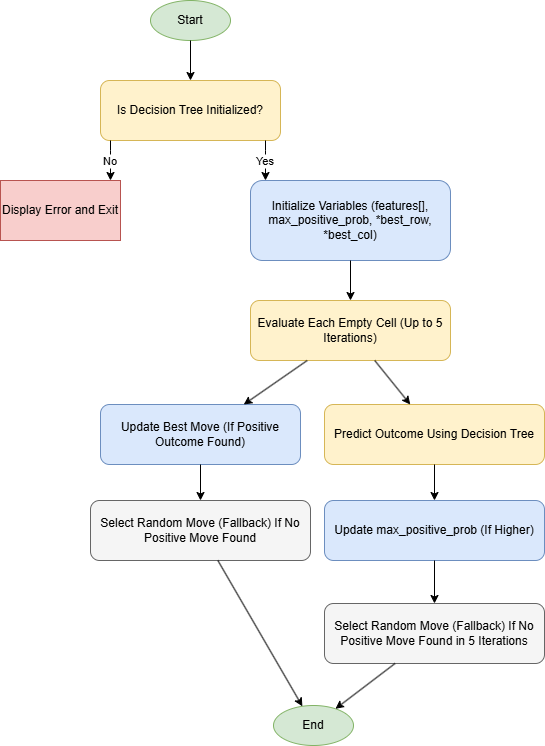
We start by verifying whether the decision tree (tree) is properly initialized. If the tree is NULL, an error message is displayed, and the function exits without continuing. This safeguard ensures that predictions are made only with a valid and trained decision tree. Additionally, several key variables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



The flowchart above is to show for 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])

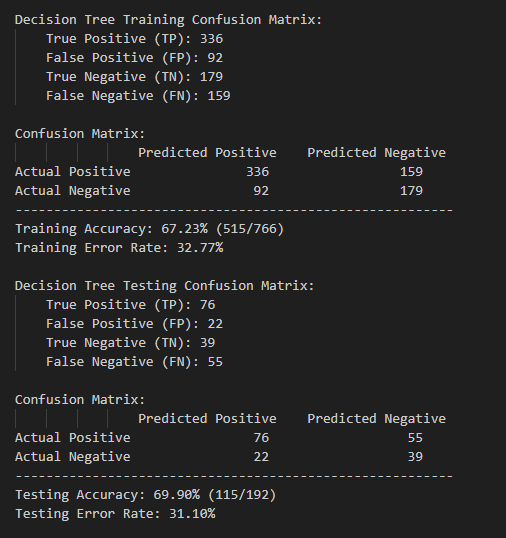
At the start of the evaluation process, all elements of the confusion matrix are initialized to zero. The model then iterates through each data point in both the training and testing datasets. If the predicted label matches the actual label, the corresponding count for either True Positive (TP) or True 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.



**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.

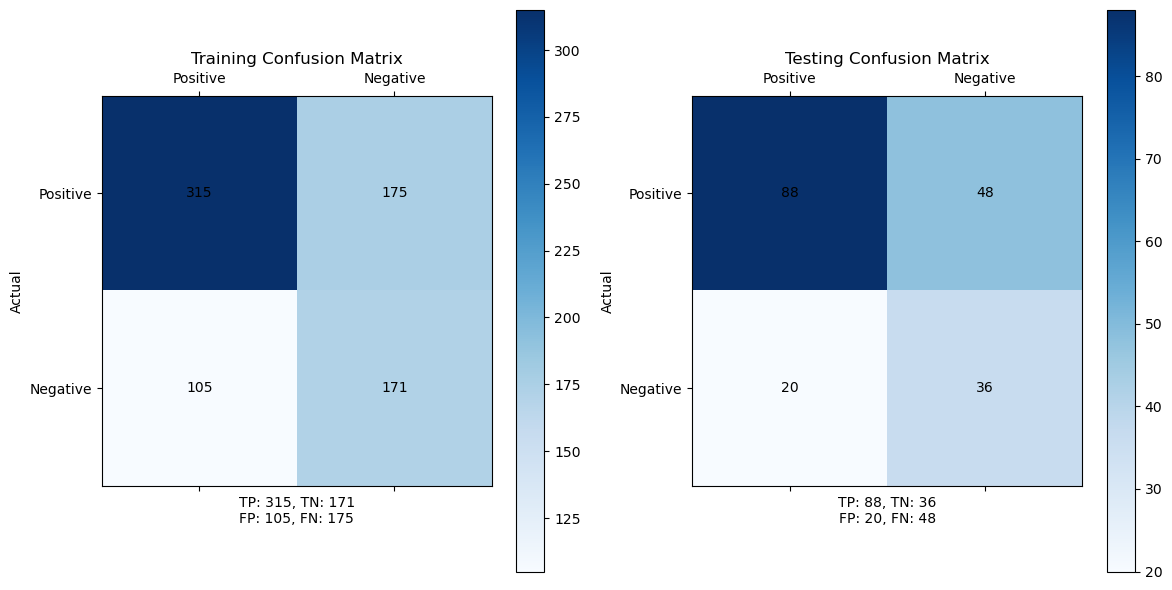
## 7.9 Confusion Matrix Plot and Results (Decision Tree)

**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.

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.

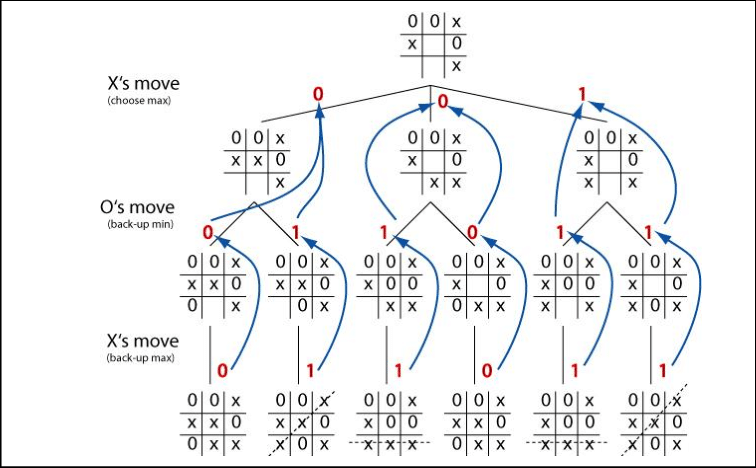
With all the implementations above, we have successfully integrated our trained decision tree model into the Tic-Tac-Toe game, allowing players to enjoy a competitive, intelligent AI opponent that offers a challenging yet entertaining experience. We also make enhancements on our decision tree model with **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.10: 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.10.
* 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.

# **1**0**. 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)

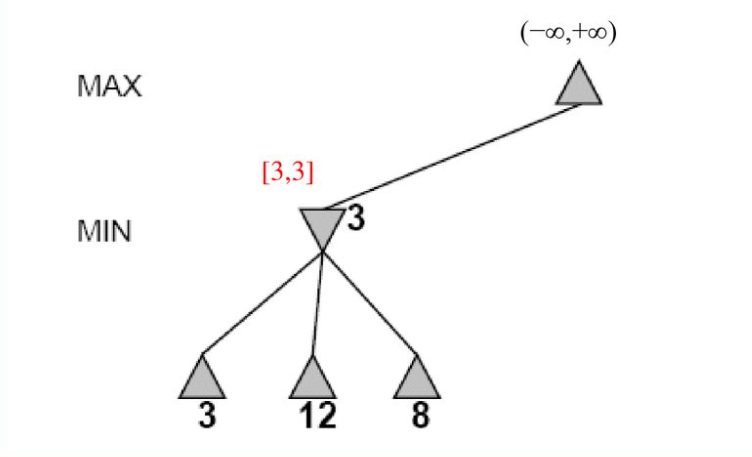
Introducing error to the algorithm winnability of the player.

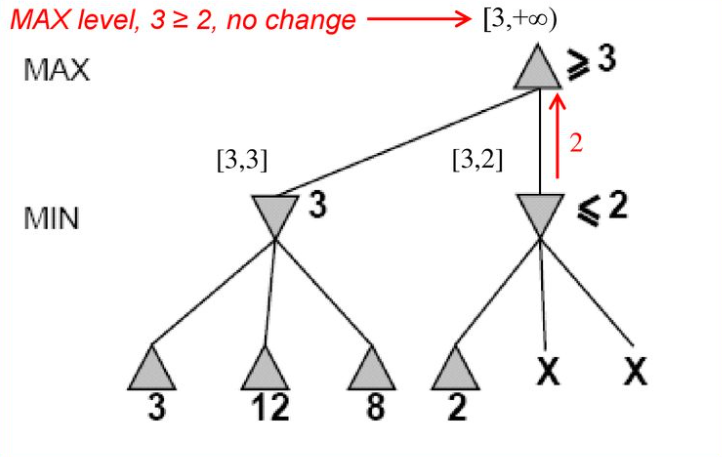
### 10.1.1 Depth Limitation

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.

## **1**0**.**2. **Minimax (Alpha-Beta Pruning)**

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

  
*Fig.11: Alpha Beta Pruning 1st node*

  
*Fig.12: 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.11. 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.11. 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.11 example, in the left subtree, we can see that the minimizing player chose the lowest score 3 and it 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.12 , 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.

## **1**0**.**3. **Modular Approac**h

The program functions are separated into smaller, specific tasks that are categorized into various parts, as outlined in our [Functions Descriptions](#_2rdvy7s13ebq) page (e.g. drawing of UI features, algorithm functions, and game logic functions).

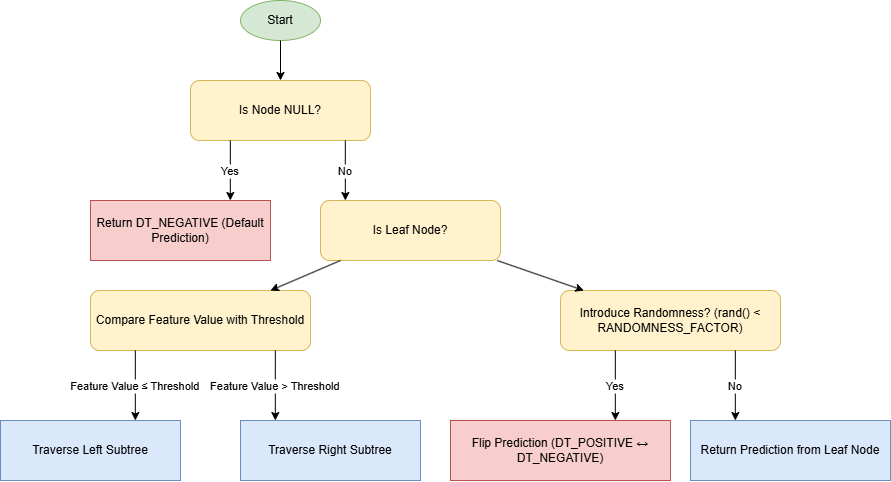
1. This modular approach makes the program more organized, and easier to manage.
2. It also makes the program more scalable, in the sense where changes and expansion of the functions can be developed in the future.
3. It promotes independent development where each function can be tested and debugged 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, we introduced the predict\_with\_randomness function. This function is a pivotal element in the AI's strategy for Tic-Tac-Toe, as it evaluates 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')



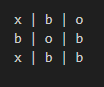
The trained decision tree model evaluates the board by starting at the root of the tree and traversing it based on the board's features. The function examines specific conditions, such as "Is this cell occupied by 'x' or 'o'?" and navigates left or right through the tree accordingly. When a leaf node 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.

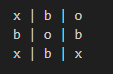
For example, during a game, if it’s the player’s turn and they place their symbol (o) in the center of the board, the board state is updated and passed to the trained decision tree model for evaluation. The AI then uses this updated state to determine its next move, blending strategic prediction with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

**Player's Turn:**



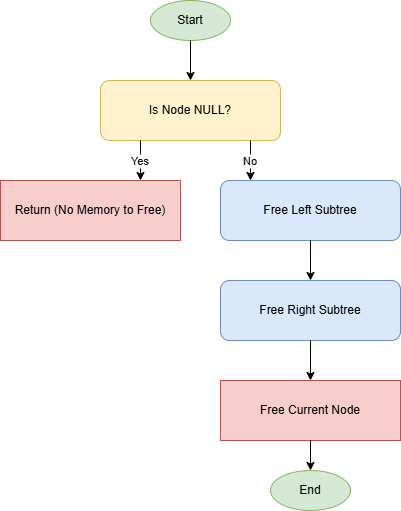
Now, when it is the AI’s turn, our AI analyzes the board using the trained decision tree and identifies potential winning moves (e.g., completing a row or blocking the player's win). At a leaf node, it evaluates whether a move is favorable or not. With slight randomness factor that influence the AI to 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**:



## 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.



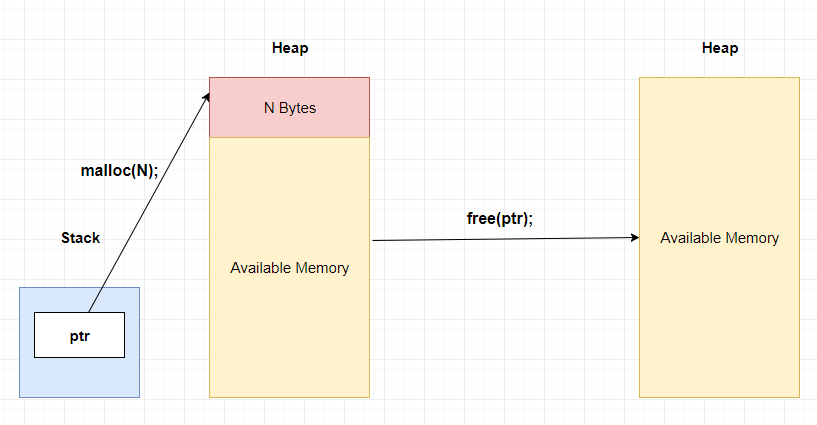
if (node == NULL) return;

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.

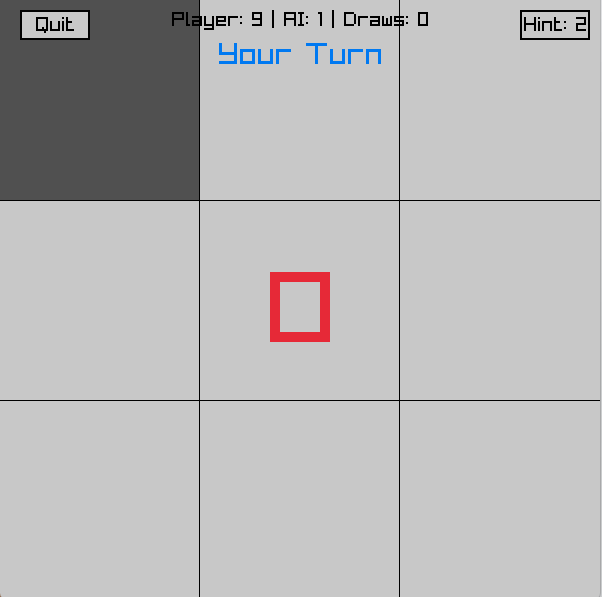
free(node);

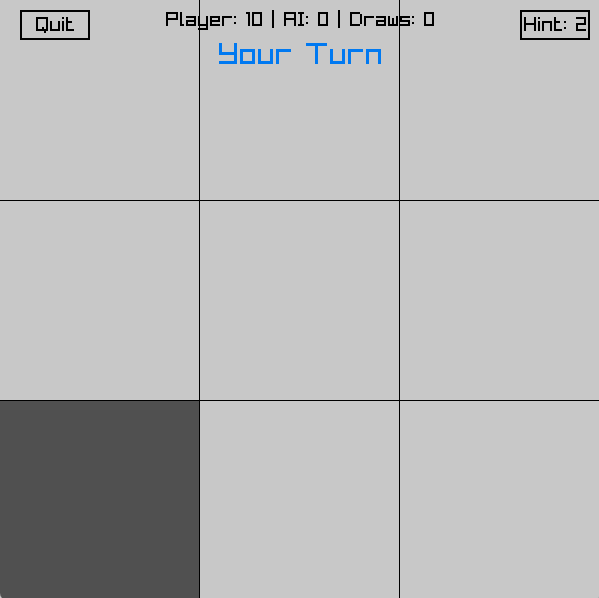
After freeing the left and right subtrees, the function deallocates the memory for the current node itself. In scenarios where the decision tree model is repeatedly trained and tested (e.g., for different datasets or parameter tuning), efficient memory deallocation is crucial. The free\_tree() function 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.

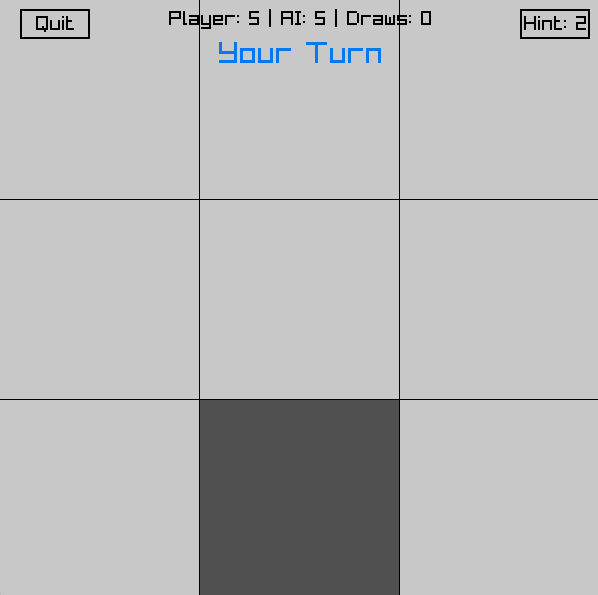
## 11.1. Level of Difficulty for Easy Mode

*****Fig.13: Win Rate for Easy Mode (Naive Bayes)*

  
*Fig.14: 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.

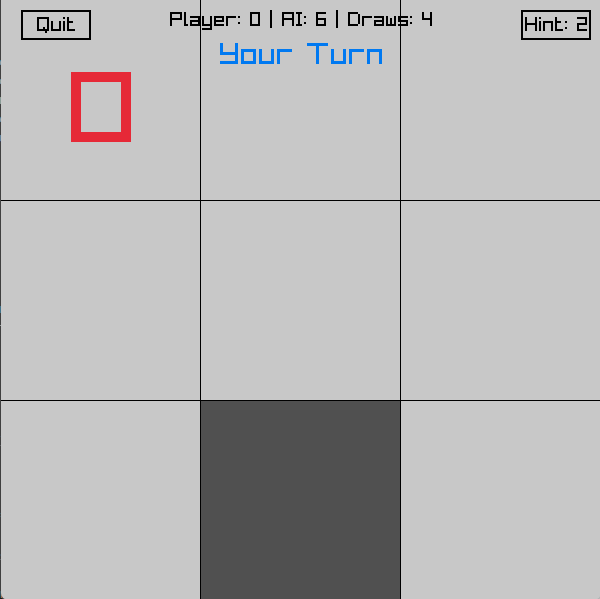
## 11.2. Level of Difficulty for Medium Mode

****

*Fig.15: 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.16: 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.

# 

# **1**2**. Appendix**

Our Youtube Demo Video: [Click here to watch it!](https://youtu.be/sleBS3mQtxM?si=I0wQOy3UFS5fKXpT) [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

#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

#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

#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

};

// 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);

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

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

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

// 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

#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;

}

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

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

**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

#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);

}

}

// 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

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

// 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.

1. Ralib website: <https://www.raylib.com/>
2. Raylib cheatsheet: <https://www.raylib.com/cheatsheet/cheatsheet.html>
3. Fig.10: [How minimax AI anticipates your every move and beats you at your own favorite game | by Adam Maj](https://adam-maj.medium.com/how-minimax-ai-anticipates-your-every-move-and-beats-you-at-your-own-favorite-game-6f5abfc3aef5)
4. Fig.11: [Game-Playing & Adversarial Search - ppt download](https://slideplayer.com/slide/17870816/) slide 7
5. Fig.12: [Game-Playing & Adversarial Search - ppt download](https://slideplayer.com/slide/17870816/) slide 12