FRONT COVER REDACTED

Table of Contents

1 Introduction

This report documents the software engineering processes the team undertook to develop a kid-friendly 3x3 Tic-Tac-Toe game with artificial intelligence in C, for a compute-limited tablet. It covers the problem definition, problem analysis, algorithm design, UI/UX iteration, and performance evaluation.

The team emphasised code quality, with comprehensive in-code documentation via Doxygen[1] comments, reusability with shared functions and header files, and modularity with minimal hardcoding of values and flexible function parameters. There is also a clear separation, both physically and logically, of the different portions of the project.

The final application provides the Tic-Tac-Toe game with a one-player mode with three difficulty levels, a two-player mode, and a kid-friendly and tablet-focused GUI.

[1] https://www.doxygen.nl/

2 Problem Definition

2.1 Tic-Tac-Toe

The core of the Tic-Tac-Toe program is a 3x3 grid, with each cell containing 3 distinct states: empty, “X” representing a player’s move, and “O” representing another player’s move.

The game concludes when either a winning condition is achieved or when the grid runs out of cells for players to place a move, constituting a draw. There are 8 possible winning combinations, which require one of the players to make three moves that constitute: a row, a column, or a diagonal.

2.2 Two-Player Mode

In the two-player mode of the game, the 2 players must be able to place down a move in an alternating fashion. After each move, the program should check for a winning condition or if there are no more empty cells left.

2.3 One-Player Mode

In the one-player mode of the game, a player will be able to place a move before an artificial intelligence (AI) algorithm is called. The algorithm will evaluate the current board and choose what it thinks is the best move for a win in its favour. The player will then be allowed to place a move again, switching back and forth with the algorithm until a winning condition is met or if there are no more empty cells left.

2.4 Minimax Algorithm

Minimax is a recursive depth-first search algorithm that explores an entire game tree for the best possible branch with the highest chance of winning (UC Berkeley, n.d.). The algorithm will evaluate every possible board state by simulating alternating moves until a terminal state (winning condition or draw) is reached.

As it is a perfect algorithm when used for Tic-Tac-Toe, a player can't win in a game against the algorithm. Therefore, mechanisms like limiting the depth must be implemented to make the algorithm less accurate, but are also able to make it run more efficiently (Tuychiyev, 2025).

2.5 Naive Bayes Classifier

A Naive Bayes classifier can be trained with a labelled dataset of terminal game states. It can predict the probability of a positive (win) or negative (draw/loss) outcome by evaluating the current board state against its trained parameters.

By iterating through a board with singular moves, it can evaluate each possible move and find the optimal move which would have the highest probability of a positive outcome.

2.6 GUI

The GUI provides an interactive visual frontend for the Tic-Tac-Toe game that allows players to perform actions with clicks or touches. It should be intuitive, simple, and attractive, as the target audience is children.

It should also provide options for the players to decide on which mode they would like to play, as well as the difficulty level of the AI algorithm to play against in one-player mode.

3 Problem Analysis

3.1 Tic-Tac-Toe

1. Create an empty board at the start of the game.
2. Place a symbol (“X” or “O”) on the board.
    a. The symbol is determined by the current player.
    b. The cell the symbol is placed in is specifiable.
    c. It should check if the cell is empty for a valid move.
3. Alternate current player with the other symbol
4. Check for the winner
    a. Row: all three cells in any row contain the same symbol.
    b. Column: all three cells in any column contain the same symbol.
    c. Diagonal: all three cells along either diagonal contain the same symbol.
    d. Draw: when all cells are filled with no winning condition.
    e. Ongoing: when none of the above conditions are met.

3.2 Two-Player Mode

1. Players select the symbol to go first.
2. Use the logic from the “Tic-Tac-Toe” portion to let the players make their moves in a game loop until the game concludes in a win or a draw.

3.3 One-Player Mode

1. The player selects the difficulty level, corresponding to an AI algorithm.
2. The player selects whether they or an AI algorithm goes first.
3. Use the logic from the “Tic-Tac-Toe” portion to let the player make a move.
4. Call the specified AI algorithm and make the move returned.
5. Alternate between the player and the algorithm until the game concludes in a win or a draw.

3.4 Minimax Algorithm

1. Generate all possible moves from the current board.
    a. Choose a specified limit of random moves; this acts as an imperfection mechanism.
2. Check if the current depth is at the specified limit; this acts as another imperfection mechanism.

    a. If it is, return a score of 0.
3. Use a loop to evaluate each of the generated moves.
4. Check if the game is in a terminal state.
    a. If it is a draw, return a score of 0.
    b. If it is a win, return a score of 9 - current depth of the search.
    c. If it is a loss, return the negative of the winning score.
5. In a recursive loop, alternate the current player and repeat the above steps.
    a. If it is the maximising player (the AI's symbol), keep the move that results in the highest score.
    b. If it is the minimising player (the player's symbol), keep the move that results in the lowest score.
    c. Using alpha-beta pruning, skip the rest of the game tree if the alpha (highest score) is greater than or equal to the beta (lowest score)
6. Return the best score to the start of the game tree to identify the optimal move with the highest overall score.

3.5 Naive Bayes Classifier

1. Dataset Processing
    a. Read the comma-delimited dataset.
    b. Vectorise each line of the dataset into a 3x3 matrix and a binary outcome (positive/negative).
    c. Randomise the vectorised dataset for the training-testing split.
        i. Randomisation is needed as the dataset is sorted by outcome.
    d. Split the first 80% of the dataset for training and the remaining 20% for testing.
2. Model Training
    a. Calculate prior probabilities by computing the occurrences of each outcome. The formula for calculating the prior likelihood is:

$$P(o) = \frac{N(o)}{total\ samples}$$

    b. Calculate the likelihood by computing the occurrences of each cell state given each outcome.

i. Utilise Laplace smoothing such that a non-occurring value will not be unfairly represented. The formula is:

$$P(s|f,o) = \frac{N(s,f,o) + \alpha}{N(f,o) + \alpha \times |S|}$$

c. Store the parameters in a unified structure.
   i. Save to a file for future testing/running.

3. Model Testing
   a. Load trained parameters from a file.
      i. The file is saved from the prior training step.
   b. Loop through each vector from the testing split.
   c. Compute posterior probabilities for a given board by evaluating each cell state against the likelihood. The formula is as follows:

$$score(o) = P(o) \prod_{i=1}^{n} P(s_i|f_i, o)$$

   d. Calculating prior probability with a low prior probability score might lead to an inaccurate underflow value (Gundersen, 2020). To solve that, the posterior probability is computed with the LogSumExp function. The formula is as follows:

$$log \sum exp(o) = m + log \sum exp(o - m), \; m = \max(o)$$

   e. Predict the outcome by choosing the one with a higher probability.
   f. Check the prediction against the actual outcome from the dataset
   g. Once all vectors have been evaluated, display the results using a confusion matrix.

4. Runtime
   a. Generate all possible moves from the current board.
   b. Use a loop to evaluate each of the generated moves.
   c. Similar to the steps in model testing, evaluate each move by computing the probabilities of the board after the move has been made.
   d. Identify the optimal move by choosing the move with the highest probability of a positive outcome.

3.6 GUI

1. Main Menu and Options Menu

    a. Present a screen with the game title.
    b. Display two buttons for one-player mode and two-player mode.
        i. When one-player mode is chosen, transition to a difficulty selection screen with buttons for selecting the difficulty.
    c. Transition to a screen for choosing the starting symbol.
        i. The “O” symbol represents the AI algorithm for one-player mode.
2. Game Board Rendering
    a. Draw a 3x3 Grid with equal-sized cells.
    b. Initially render all the cells as empty; after every move, draw the appropriate symbol (X or O) in the selected cell.
3. Input Handling
    a. Capture click/touch events on the grid and map each event to the board cell.
    b. Before a move is executed, it must be validated to ensure that the selected cell is empty.
    c. Once validated, the move is made with the core Tic-Tac-Toe logic.
4. Result Detection & Display
    a. After each move, use the core logic to determine the current game state: win, draw, or ongoing.
    b. If a win or draw is detected, a message should indicate which symbol has won.

4 Algorithm Design (Pseudocode)

4.1 Tic-Tac-Toe

```
FUNCTION initalise_board(starting_player)
      FOR each cell DO
            cell <- EMPTY
      ENDFOR
      current_player <- starting_player
      last_move <- {-1, -1}
      move_count <- 0
      RETURN board
ENDFUNCTION

FUNCTION find_empty_cells(board)
      count <- 0
      FOR each cell DO
            IF (cell == EMPTY)
                  empty_cells[count++] <- cell
                  count++
            ENDIF
      ENDFOR
      RETURN empty_cells
ENDFUNCTION

FUNCTION make_move(board, move)
      IF (cell[move] != BLANK) THEN
            return FALSE
      IF (current_player <- X) THEN
            cell <- X
```

```
      ELSE
            cell <- O
      ENDIF
      move_count++
      last_move <- cell
      IF (current_player <- X)
            current_player <- O
      ELSE
            current_player <- X
      ENDIF
      RETURN TRUE
ENDFUNCTION

FUNCTION undo_move(board, move)
      IF (last_move == (-1, -1)) THEN
            RETURN FALSE
      ENDIF
      cell[last_move] <- BLANK
      move_count--
      last_move <- {-1, -1}
      IF (current_player <- X)
            current_player <- O
      ELSE
            current_player <- X
      ENDIF
      RETURN TRUE
ENDFUNCTION

FUNCTION check_winner(board)
```

```
      FOR each pattern in win_patterns DO
            a <- board.cell[pattern[0]]
            b <- board.cell[pattern[1]]
            c <- board.cell[pattern[2]]
            IF (a != EMPTY AND a==b AND b==c)
                  RETURN winner
            ENDIF
      ENDFOR
      empty_count <- find_empty_cells(board)
      IF (empty_count > 0) THEN
            RETURN ongoing
      ELSE
            RETURN draw
      ENDIF
ENDFUNCTION
```

4.2 One-Player Mode

```
FUNCTION get_ai_move(board, difficulty)
      IF (move_count == 0 OR difficulty == easy)
            RETURN random_move(board)
      ENDIF
      IF (difficulty == medium)
            RETURN naive_bayes_move(board)
      ENDIF
      RETURN minimax_move(board)
ENDFUNCTION
```

4.3 Minimax Algorithm

```
FUNCTION minimax(board, is_max, alpha, beta, depth)
      IF (depth >= max_depth)
```

```
            RETURN 0
        ENDIF
        result <- check_winner(board)
        IF (result == DRAW)
            RETURN 0
        ENDIF
        IF (result == WIN_X OR result == WIN_O)
            score <- SIZE * SIZE - depth
            RETURN (ai_player == winner) ? score : -score
        ENDIF
        empty_cells <- find_empty_cells(board)
        IF (is_max)
            highest_score <- -10
            FOR each move in empty_cells DO
                make_move(board, move)
                score <- minimax(board, is_max, alpha, beta,
depth + 1)
                undo_move(board, move)
                IF (score > highest_score)
                    highest_score <- score
                ENDIF
                IF (highest_score > alpha)
                    alpha <- highest_score
                ENDIF
                IF (beta <= alpha)
                    BREAK
                ENDIF
            END FOR
            RETURN highest_score
```

```
      ELSE
            lowest_score <- 10
            FOR each move in empty_cells DO
                  make_move(board, move)
                  score <- minimax(board, is_max, alpha, beta,
depth + 1)
                  undo_move(board, move)
                  IF (score < lowest_score)
                        lowest_score <- score
                  ENDIF
                  IF (lowest_score < beta)
                        beta <- lowest_score
                  ENDIF
                  IF (beta <= alpha)
                        BREAK
                  ENDIF
            END FOR
            RETURN lowest_score
      ENDIF
ENDFUNCTION

FUNCTION minimax_best_move(board)
      best_move <- {-1, -1}
      best_score <- -10
      empty_cells <- find_empty_cells(board)
      IF (empty_cells > max_samples)
            empty_cells <- fisher_yates_shuffle(empty_cells)
            empty_cells <- take_first(empty_cells, max_samples)
      ENDIF
```

```
    FOR each move in empty_cells DO
        make_move(board, move)
        score <- minimax(board, FALSE, INT_MIN, INT_MAX, 0)
        undo_move(board, move)
        IF (score > best_score)
            best_score <- score
            best_move <- move
        ENDIF
    END FOR
    RETURN best_move
ENDFUNCTION
```

4.4 Naive Bayes Classifier

4.4.1 Processing Data

```
FUNCTION process_line(line)
    token <- get_token(line, “,”)
    FOR each cell do
        SWITCH(token)
            CASE ‘x’
                vector.cells[cell] <- 1
            CASE ‘o’
                vector.cells[cell] <- 2
            CASE ‘b’
                vector.cells[cell] <- 3
            CASE default
                PRINT invalid
                break
```

```
            token <- get_token(line, “,”)
      END FOR
      token <- get_token(line, “,”)
      IF (token != positive) THEN
            vector.outcome <- 1
      ELSE IF (token != negative) THEN
            vector.outcome <- 2
      ELSE
            PRINT Invalid
            RETURN
      END IF
      RETURN vector
ENDFUNCTION

FUNCTION process_dataset(filepath)
      dataset <- read_file(filepath)
      count <- length(dataset)
      i <- 0
      data_entries[length] <- 0
      WHILE i < length
            line <- dataset[i]
            vector <- process_line(line)
            data_entries[i]
            i <- i + 1
      ENDWHILE
```

```
      RETURN data_entries, count
ENDFUNCTION


FUNCTION shuffle_dataset(data_entries, size)
      IF (size < 2) THEN
            return data_entries
      ENDIF
      i <- 0
      WHILE i < size DO
            j <- i + rand() % (size - i)
            temp_entry <-  data_entries[j]
            data_entries[j] <- data_entries[i]
            data_entries[i] <- temp_entry
            i <- i + 1
      ENDWHILE
      RETURN data_entries
ENDFUNCTION
```

4.4.2 Training

```
FUNCTION train_model(data_entries, size)
      model.prior[2] <- 0, 0
      outcome_count[2] <- 0, 0
      model.likelihood[54] <- 4D array filled with zeroes
(outcomes.size.size.state)
      state_count[54] <- 4D array filled with zeroes
(outcomes.size.size.state)
      i <- 0
```

```
      WHILE i < size DO
            cur_outcome <- data_entry.outcome
            outcome_count.cur_outcome++
            FOR each cell DO
                  state <- data_entry.cells[cell]
                  state_count.outcome.cell.cell.state++
            ENDFOR
      ENDWHILE
      FOR each outcome DO
            model.prior.outcome = (outcome_count.outcome + alpha)
/ size + alpha * outcomes
            FOR each cell DO
                  model.likelihood.outcome.cell.state  <-
(state_count.cell.state + alpha) / outcome_count.outcome + alpha
* state
            ENDFOR
      ENDFOR
      RETURN model
ENDFUNCTION


FUNCTION save_model(model, model_path)
      file <- write_file(model_path)
      file <- model
      RETURN
ENDFUNCTION
```

4.4.3 Testing & Algorithm

```
FUNCTION invert_board(board)
      FOR each cell DO
            IF (cell == X) THEN
                  cell <- O
```

```
            ELSE IF (cell == O) THEN
                  cell <- X
            ENDIF
      ENDFOR
      RETURN
ENDFUNCTION

FUNCTION load_nb_model(model_path)
      file <- read_file(model_path)
      model <- file
      RETURN model
ENDFUNCTION

FUNCTION naive_bayes(board, model)
      log_scores[2] <- 0.0, 0.0
      FOR each outcome DO
            score <- log(model.prior[outcome])
            FOR each cell DO
                  cell_state <- board.cells[cell]
                  score <- score +
model.likelihood.outcome.cell.cell_state
            ENDFOR
            log_scores[outcome] <- score
      ENDFOR
      max_log_score <- max(log_scores)
      total_log_score <- 0
      FOR each outcome DO
            total_log_score <- total_log_score +
exp(log_scores[outcome] - max_log_score)
```

```
      ENDFOR
      log_sum <- max_score + log(total_log_score)
      FOR each outcome DO
            log_scores[outcome] = exp(log_scores[outcome] -
log_sum)
      ENDFOR
      negative_probability <- log_scores[0]
      positive_probability <- log_score[1]
      RETURN negative_probability, positive_probability
ENDFUNCTION

FUNCTION nb_find_move(board_ptr, model)
      best_move <- {-1, -1}
      best_probability <- -1.0
      empty_cells <- find_empty_cells(board)
      if (ai_player == O) THEN
            invert_board(board)
      ENDIF
      FOR each move in empty_cells DO
            make_move(board, move)
            probability <- minimax(board, FALSE, INT_MIN, INT_MAX,
0)
            undo_move(board, move)
            IF (probability > best_probability)
                  best_probability <- probability
                  best_move <- move
            ENDIF
      END FOR
      RETURN best_move
```

5 User Acceptance Testing

5.1 Initial UI

The initial design of the Tic Tac Toe UI was developed after examining several online variants of the game, such as:

- https://playtictactoe.org/
- https://www.mathsisfun.com/games/tic-tac-toe.html
- https://www.cooltictactoe.com/
- https://www.crazygames.com/game/tic-tac-toe

These reference sites provided useful and practical insights that helped shape and influence the foundational look and feel of our early mockups.

The use of large buttons on playtictactoe.org and Maths Is Fun guided our decision to adopt oversized mode-selection and difficulty buttons in the initial UI. It demonstrates that clearly labelled buttons improve player navigation, especially for quick-start games. The team also incorporated bright colours on a dark navy background to ensure strong contrast, following a “Show, Don’t Tell” principle.

Sites like Crazy Games and Cool Tic-Tac-Toe emphasise simple, uncluttered menus with a single key action per screen. This principle reduces the friction of navigation and playing the game by reducing the amount of information per screen. The team settled on separating the initial UI into 4 distinct screens:

- Game Mode Selection (One-Player or Two-Player mode)
- Difficulty Selection (For One-Player Mode)
- Player Selection (“X” or “O” to go first)
- Game Grid

The game grid in our early UI features wide spacing and thick lines, drawing inspiration from websites such as playtictactoe.org and Cool Tic-Tac-Toe, where the grid is visually dominant and easy to view. The wireframe mockup was done in Figma[2] and can be found in Appendix A.

5.2 User Feedback and UI iteration

REDACTED

Figure 1 and 2 - Friends and family testing the team’s application

[2] https://www.figma.com/

5.2.1 Feedback 1

“It was quite troublesome that I had to go back to the main menu and re-select everything to restart another game.”

Changes Made

The team implemented a small pop-up modal that appears when a game concludes. This pop-up displays the winning player or a tie and provides two options: one for returning to the main menu and another to start a new game with the same options.

This feature allows players to quickly choose if they want to for another round or exit back to the main menu, improving the game’s flow and usability. The team consulted the user with the new modal, and they were delighted at how it functioned and looked. A picture of the modal when a game concludes is available in Appendix B.

5.2.2 Feedback 2

“It would be nice if there were an in-game scoreboard to tally wins against a friend or the bot player”

Changes Made

The team implemented an in-game scoreboard to track and display the number of rounds each player has won. The scoreboard is positioned below the game grid, being a prominent and easy-to-read location. As each game concludes, the scoreboard updates automatically.

After consulting the user with a prototype scoreboard in our wireframe, they suggested additional features, like showing the current player. They were satisfied with the final design of the scoreboard, commenting that the game grid still looks good while being more informational. A picture of the scoreboard after a few games is available in Appendix B.

5.2.3 Feedback 3

“The GUI feels quite dark; it looks very techy and minimalist, which I don’t feel is suited for kids”

Changes Made

To improve the game's appeal, particularly for younger audiences, the team enhanced the user interface with additional colour accents and brighter colours

overall. The team also redesigned the buttons and added hover effects to increase the feeling of responsiveness.

The team performed A/B testing with a group of friends, instructing them to choose the variant that they believed would appeal more to adolescents. The majority favoured the newer colours and design, agreeing that it looks more kid-friendly. A picture of the main menu with the adjustments is available in Appendix B.

6 Algorithm Evaluation

6.1 Naive Bayes' Testing Split

6.1.1 Methodology

To evaluate the classification performance of the Naive Bayes model, the team has decided to utilise a confusion matrix alongside accuracy, precision, recall, and F1 score.

The team implemented a program in C for the purpose of evaluating the trained Naive Bayes classifier, using the testing split (20%) of the dataset. It can be run with `./ml-cli stats` after the trained parameters are saved to the default path during training.

Accuracy represents the percentage of correct predictions:

$$Accuracy = \frac{True\ Positive + True\ Negative}{True\ Positive + True\ Negative + False\ Positive + False\ Negative}$$

Precision represents the percentage of correct predictions made for positive outcomes:

$$Precision = \frac{True\ Positive}{True\ Positive + False\ Positive}$$

Recall represents the percentage of all actual positive outcomes that were correctly predicted:

$$Recall = \frac{True\ Positive}{True\ Positive + False\ Negative}$$

F1 calculates the harmonic mean between precision and recall:

$$F1 = \frac{2 \times Precision \times Recall}{Precision + Recall}$$

6.1.2 Results

True Positive	True Negative
112	23
False Positive	**False Negative**

37	20

Table 1 - Confusion Matrix of the trained Naive Bayes classifier

Accuracy	Precision	Recall	F1 Score
70.31	75.17	84.85	79.72

Table 2 - Accuracy, Precision, Recall and F1 Score of the trained Naive Bayes classifier

6.1.3 Analysis

The Naive Bayes classifier manages to attain an overall accuracy of 70.31%, indicating that it can classify ~7 out of 10 boards correctly. It has an F1 score of 79.72%, which demonstrates a good trade-off between false positives and false negatives.

However, it is not perfect, leading to some moves that may not be strategically optimal. These results should also not be taken at face value, as they were evaluated with the dataset of terminal moves. The statistics only show how effective the model is at predicting if a terminal state board results in a positive outcome or a negative outcome, when it would be evaluating non-terminal boards during actual usage.

6.2 Win Rates

This section aims to fairly evaluate the efficacy of the different AI algorithms at winning an actual game. Getting the win rates allows the team to properly assess which algorithm should be assigned to which difficulty level.

6.2.1 Methodology

The team has implemented a benchmarking program in C that is able to record how often each AI algorithm wins and draws when playing against a baseline opponent that picks a move at random. The program is able to simulate thousands of unique games with C's random function in the standard library. 5000 games of Tic-Tac-Toe will be played between the AI algorithm and the baseline opponent, with their outcome recorded (the algorithm's win or a game draw) for evaluation.

The starting player will alternate between the AI algorithm and the baseline opponent to ensure a fair evaluation. The symbol that the AI algorithm plays ("X" or "O") will also be alternated every 2 games for greater fairness.

The program can be run with `./ml-cli benchmark` after the trained Naive Bayes parameters are saved to the default path during training.

6.2.2 Results

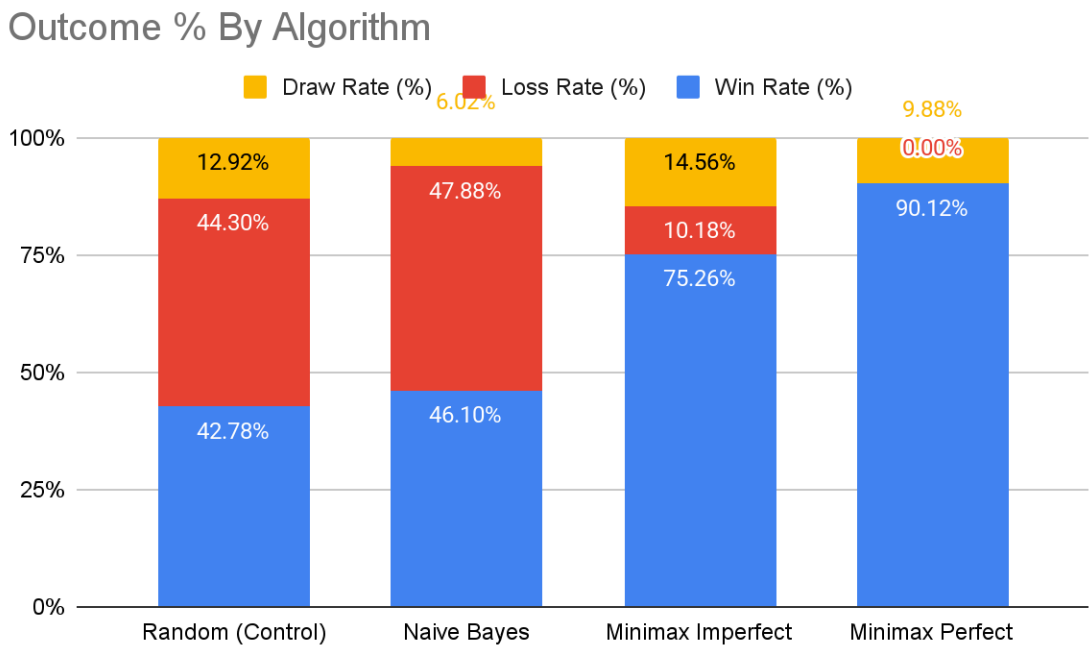


Figure 3 - Outcome of win, loss, and draw rates for each AI algorithm

6.2.3 Analysis

Based on the control statistics (random opponent vs. random opponent), the evaluation can be considered unbiased, as there is only a deviation of ±1.52% between the win rate and loss rate. As the algorithm is random and helps new players to acclimate to the game's rules and conventions, the team chose to assign it to the easy difficulty.

Naive Bayes performs quite poorly with statistics close to the control. It is able to win 46.1% of the time but loses 47.88% of the time to a random opponent. It is noticeably less likely to draw when compared to the control. However, when playing against a human opponent who is not likely to make random moves, it has the potential to perform better in real-world usage. Therefore, the team chose to assign it to the medium difficulty.

Minimax with imperfections has a respectable win rate of 75.26%, with a loss rate of 10.18%. Given its efficiency and accuracy trade-off, it has exceeded the

team's expectations for skill level. Being able to win a quarter of the games would make it a challenging but beatable opponent in real-world usage, leading the team to assign it to the hard difficulty.

The perfect minimax algorithm without surprise manages to win 90.12% of the games, with zero losses. It is also able to minimise the chances of a draw when compared to the control. As such, this algorithm is not implemented into the game as it defeats the purpose of having an opponent, where the player should at least have a small chance of winning against.

6.3 Response Times

To determine the computational efficiency of the AI algorithms, the team has decided to measure the time needed for a move by each of the different algorithms. A lower response time would mean a more efficient algorithm, which will result in lower latency and more responsive gameplay.

6.3.1 Methodology

The same benchmarking program used to record win rates is also able to record response times of the AI algorithms. With the response times, the execution performance of each algorithm can be effectively gauged.

The first move of the game will be random in order to vary the starting positions, such that the computations will be different each run, along with the game tree. The algorithms will then play against themselves until the game concludes, with the time taken to return a move being recorded. Similar to win rate benchmarking, 5000 games will be simulated.

As the computational complexity does not stay the same throughout the game, the team has decided to split the response time for each algorithm by the number of moves left as well.

6.3.2 Results

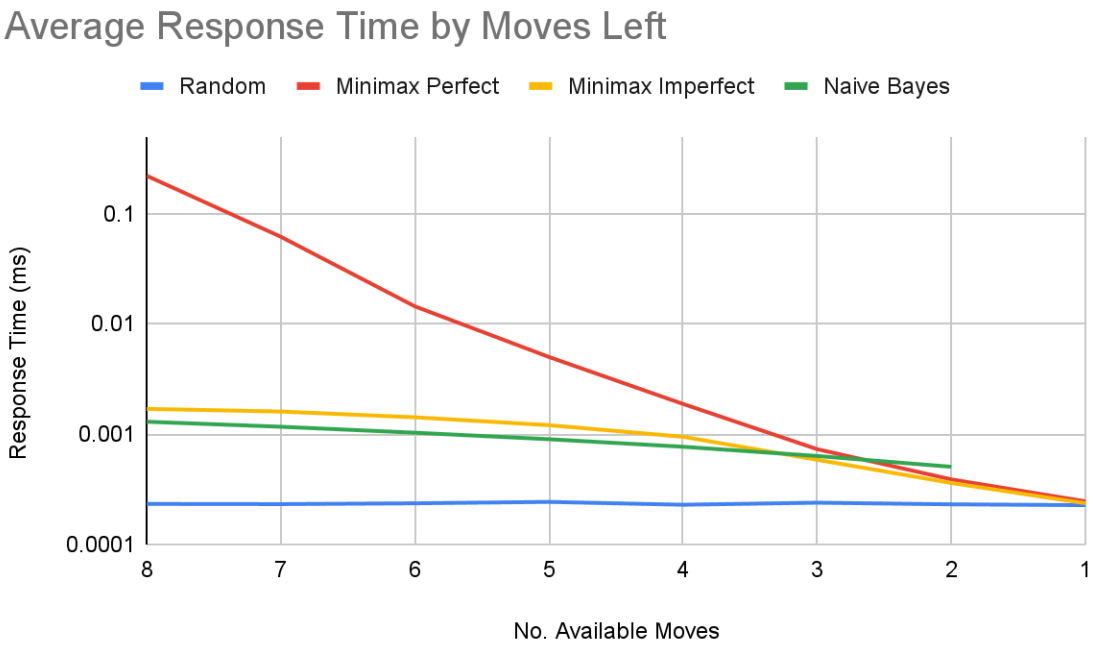


Figure 4 - Average response time of each algorithm by moves, on a logarithmic scale

6.3.3 Analysis

Firstly, the random algorithm that does not need to perform evaluation of the board is able to achieve a flat response time graph, essentially disregarding the number of moves left. This is expected as the algorithm simply generates a random number and chooses an empty cell based on the number generated, with O(1) time complexity.

The next fastest algorithm is the Naive Bayes classifier. Although its graph is not flat, there is not much variation in the number of moves left. The algorithm works by performing the probability calculation of placing a move in each empty cell, then choosing the one with the highest positive outcome probability, so it effectively has the time complexity of O(n).

Before discussing minimax with imperfections, it is noteworthy to analyse how computationally inefficient the original minimax algorithm is, due to its unrestricted depth-first search nature. For it to evaluate a move when there are 8 moves left, it takes >800x longer than the Naive Bayes classifier to do so, which might not seem very pronounced on the graph due to the logarithmic scale. For a perfect algorithm to play the game, it takes up a disproportionate amount of

computation to do so. With alpha-beta pruning implemented, its time complexity is still relatively inefficient at O(n^(9/2)).

Lastly, the minimax algorithm with imperfections is similar to Naive Bayes' efficiency, only slightly worse. This shows that the imperfection mechanisms the team has engineered make the minimax algorithm viable for use, with a low computational requirement and low latency. By tuning the sample moves and depth limit, the team has achieved a good balance between skill level and runtime performance, ultimately ending up with a time complexity of O(n^2). The time complexity is as such, as its depth is limited to 2, and its sample moves are limited to 4.

6.3.4 Cross-Platform Individual Testing

System Specifications & Response Time

REDACTED

Memory and CPU usage

REDACTED

7 Conclusion

In this project, the team developed a complete Tic-Tac-Toe game from scratch. The team started with implementing the basic game rules and a functional two-player mode. Subsequently, a one-player mode with different difficulty levels was implemented using the Minimax algorithm and a Naive Bayes classifier. On top of that, the team designed a user-friendly child-appropriate GUI with user experience iteration, providing an engaging experience for users.

Ultimately, the team gained a deeper understanding into how different components of software engineering, like business logic, data structures and algorithms, and user interface design, are put together. This project allowed the team to discover many insights that will be useful in future academic and professional work.

7.1 Future Work

The final application in the report is able to fulfil the basic functional requirements for both one-player and two-player Tic-Tac-Toe games. However, there are a few aspects which could further improve the application which the team could not explore due to the project's limited timeframe.

Firstly, the team believed that the GUI performance could be improved by using a more streamlined and lower-level library like SDL2, instead of GTK 4. While GTK 4 provides a large variety of widgets and accessibility options, many of these were unnecessary for the game's design, which could have impacted performance. The team did not initially start with SDL2 as the primary GUI framework, as the performance implications of GTK 4 were unknown at the time, and the team presumed it to be a lightweight enough GUI library.

Secondly, the team observed that the Naive Bayes classifier's performance was below expectations; it only achieved a marginally higher win rate compared to the control. This is likely due to its ability to only return a binary classification of a positive or negative outcome, and it lacks awareness of previous or future game states.

The team recommends exploring an alternative ML technique more suited for Tic-Tac-Toe like reinforcement learning (RL), in order to enhance the gameplay difficulty for a more engaging experience. The new ML algorithm can be easily adapted to fit the project's codebase due to its modular nature.

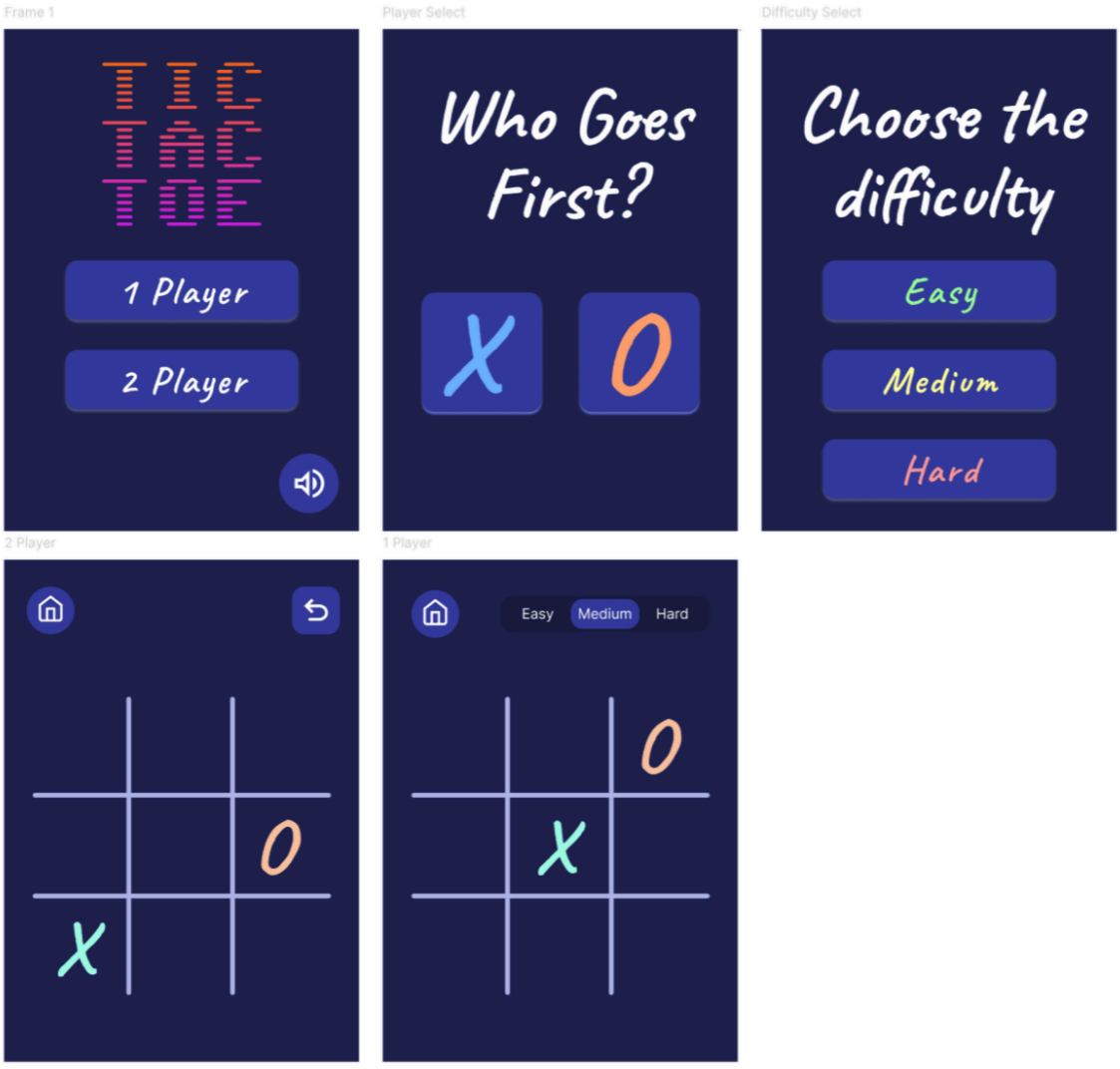
References

Gundersen, G. (2020, 2 9). *The Log-Sum-Exp Trick*. Gregory Gundersen. Retrieved 11 20, 2025, from https://gregorygundersen.com/blog/2020/02/09/log-sum-exp/

Tuychiyev, B. (2025, January 31). *Implementing the Minimax Algorithm for AI in Python*. DataCamp. Retrieved November 20, 2025, from https://www.datacamp.com/tutorial/minimax-algorithm-for-ai-in-python

UC Berkeley. (n.d.). *3.2 Minimax*. Introduction to Artificial Intelligence. Retrieved November 20, 2025, from https://inst.eecs.berkeley.edu/~cs188/textbook/games/minimax.html

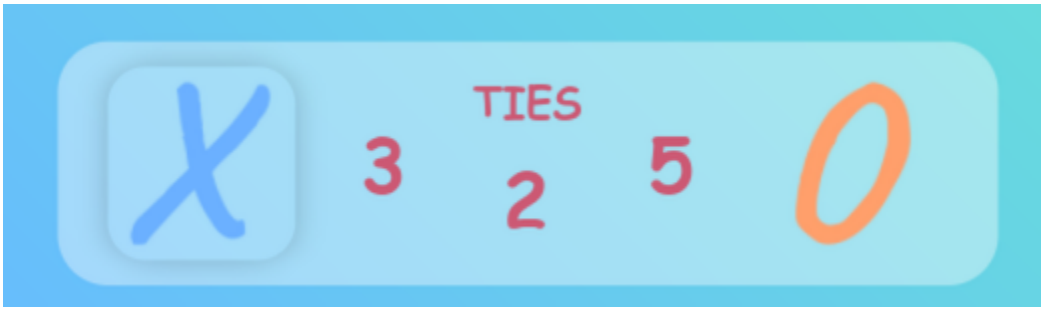
Appendix A - Figma Mockup



Appendix B - Feedback Changes

Feedback 1 - Game Conclude Modal

Feedback 2 - Scoreboard with Current Player Display



Feedback 3 - Main Menu with New Colour Scheme