TicTacToe

v1.0

Generated by Doxygen 1.12.0

1 TicTacToe

1.1 TIC TAC TOE - CSC1103 & CSC1104 Project

1.1.1 Installation Instructions (Linux)

To get started with the project, ensure you have the following installed:

Important

> Ensure that you follow all the instruction in the link below!

1. **Setup Docker's apt repository**

```
 # Add Docker's official GPG key:
 sudo apt-get update
 sudo apt-get install ca-certificates curl
 sudo install -m 0755 -d /etc/apt/keyrings
 sudo curl -fsSL https://download.docker.com/linux/debian/gpg -o /etc/apt/keyrings/docker.asc
 sudo chmod a+r /etc/apt/keyrings/docker.asc
 # Add the repository to Apt sources:
 echo \
  "deb [arch=$(dpkg --print-architecture) signed-by=/etc/apt/keyrings/docker.asc]
https://download.docker.com/linux/debian \
  $(. /etc/os-release && echo "$VERSION_CODENAME") stable" | \
sudo tee /etc/apt/sources.list.d/docker.list > /dev/null
sudo apt-get update
```

2. **Installing xhost(X11)**

```
sudo apt-get install x11-apps
sudo apt-get install x11-xserver-utils
```

1. **Installing Docker Desktop**

```
sudo apt-get install docker-ce docker-ce-cli containerd.io docker-buildx-plugin docker-compose-plugin
```

Remarks

> Adding this command in startup.sh will make your life easier.

```
systemctl --user start docker-desktop
```

1.1.1.1 Building the Project via Docker (Linux)

After setting up WSL2 and Docker, you can choose to either load a Docker image or build the Docker image yourself.

1.1.1.1.1 [OPTIONAL] Loading Docker Image

```
sudo docker load -i FILE_NAME.tar
```

1.1.1.1.2 Build and Run TicTacToe Application on Docker (Linux)

```
sudo ./run_docker.sh
```

Warning

> **Warning for Docker Building:**
>
> If `compile.sh` or `run_docker.sh` is not found and it's clearly in the directory, run the following command to convert it to Unix line endings:
> ```
> dos2unix SCRIPT_NAME.sh
> ```

Remarks

> **Saving Docker Image**
> ```
> ./run_script.sh -s
> ```

1.1.2 Installation Instructions (MacOS)

To get started with the project, ensure you have the following installed:

1. **Docker Desktop** Select based on your specs.
   - `Install Docker Desktop`
   - Select "Download for Mac - Intel Chip"
   - Select "Download for Mac - Apple Silicon"
2. **Installation Docker Desktop**
   - Go to "Downloads" and double click "Docker.dmg"
   - Drag and Drop "Docker" to "Application"
   - "Launch Pad" either search or swipe to find "Docker"
3. **Verify Docker Desktop**
   - Launch the application and select "Recommended Settings" OR
   - Type the command in terminal "docker --version"
4. **Verify Docker Image and Containers** Now create a new directory/folder and "cd" to that folder. inside terminal use the following command and wait for "Download complete"
   - docker pull hello-world
   - Verify docker image in "Docker Desktop"

Under the action tab click on "Run" OR From terminal type the following.

```
docker run hello-world
```

To verifiy container. Switch to the containers tab. Under the name tab and click on the "highlighted blue" and view the logs. OR Use the following command.

```
docker ps -a
```

1.1.2.1 Run TicTacToe Application on Docker (MacOS)

Important

> ∗∗Pre-requisite application to be install.
>
> ∗∗BEFORE BUILDING DOCKER IMAGE OR CONTAINER.

1. Install brew
   - `Install brew`
2. install xquartz from terminal
   ```
   brew install --cask xquartz
   ```
3. xquarts settings
   - setting -> security -> allow connections from network clients
   - Restart xquartz
4. Terminal command
   ```
   xhost + 127.0.0.1
   ```
5. Build and Run Docker
   ```
   docker-compose up --build -d
   ```

1.1.3 Installation Instructions (Windows)

To get started with the project, ensure you have the following installed:

1. **Docker Desktop**
   - `Install Docker Desktop`
2. **WSL2 (Windows Subsystem for Linux)**
   - Follow `Microsoft's guide to install WSL2`
   - Install Ubuntu with the following command:
     ```
     wsl --install -d Ubuntu
     ```
   - Set Ubuntu as your default WSL distribution:
     ```
     wsl --set-default Ubuntu
     ```
   - Check default: `wsl -l -v`

1.1.3.1 Building the Project via Docker (Windows)

After setting up WSL2 and Docker, you can choose to either load a Docker image or build the Docker image yourself.

1.1.3.1.1 [OPTIONAL] Loading Docker Image

```
docker load -i FILE_NAME.tar
```

1.1.3.1.2 Build and Run TicTacToe Application on Docker (Windows)

```
./run_docker.sh
```

Warning

> **Warning for Docker Building:** If `compile.sh` or `run_docker.sh` is not found and it's clearly in the directory, run the following command to convert it to Unix line endings:
> ```
> dos2unix SCRIPT_NAME.sh
> ```

Remarks

> **Saving Docker Image**
> ```
> ./run_script.sh -s
> ```

1.1.3.2 Build and Run the Project in Windows (w/o Docker)

Attention

> **Not recommended**, unless you know what you're doing and install the right packages. (refer Dockerfile)
> ```
> ./compile.sh
> ```

1.1.4 BASIC REQUIREMENTS (BOTH)

- ☑ GUI (GTK)
- ☑ 2 Player Mode
- ☑ 1 Player Mode ("Perfect" Minimax)
- ☑ Winning Logic
- ☑ GUI indication when player WIN (e.g, blinking))

1.1.5 [^1]PM-CSC1103 REQUIRMENTS

- ☑ Improve Minimax memory usage
- ☑ Implement ML Algorithm (80:20)

  [TIP] Linear regression, Navie bayes, Neural network and Reinforcement learning
- ☑ Plot the confusion matrix for the training and testing accuracy
- ☐ Calculate the number of times the computer wins as a gauge of difficulty level.

1.1.6 [^2]COA-CSC1104 REQUIRMENTS

☑ Replace one function with assembly

> [TIP] Use inline assembly code in C source file, or linking separate C and assembly object files.

[^1]: PROGRAMMING METHODOLOGY. [^2]: COMPUTER ORGANIZATION AND ARCH.

2 Doxygen Documentation Generation and Viewing Instructions

This guide provides instructions for generating, extracting, and viewing the Doxygen documentation in HTML and LaTeX formats.

2.1 Steps to Generate and View the Documentation:

2.1.1 1. Run the Doxygen Configuration File:

- Locate the `Doxyfile` in the project directory.
- Run Doxygen using the following command in your terminal: `bash doxygen Doxyfile`
- This will generate two folders in ./docs:
  - `html` (for HTML documentation)
  - `latex` (for LaTeX documentation)

2.1.2 2. View the HTML Documentation:

- Navigate to the `html` folder generated by Doxygen.
- Locate the file named `index.html`.
- Open `index.html` in your preferred web browser:
  - Double-click the file to open it using the default web browser.
  - Alternatively, right-click the file and select `Open With` to choose a specific browser (e.g., Chrome, Firefox, Edge).

2.1.2.1 Recommended Browsers:

- Google Chrome (latest version)
- Mozilla Firefox (latest version)
- Microsoft Edge (latest version)
- Safari (for macOS users)

2.1.3 3. Generate and View LaTeX Documentation:

- Navigate to the `latex` folder generated by Doxygen.
- Compile the LaTeX files into a PDF:
  - Ensure you have a LaTeX distribution installed, such as TeX Live, MiKTeX, or MacTeX.
  - Run the following commands in the terminal: `bash cd latex make`
  - This will produce a PDF file (e.g., `refman.pdf`) in the `latex` folder.
- Open the PDF using a PDF viewer like Adobe Acrobat Reader or any other suitable application.

2.2 Troubleshooting:

- **Doxygen Command Not Found**:
  - Ensure Doxygen is installed and added to your system's PATH.
  - Download Doxygen from `https://www.doxygen.nl/download.html` if not already installed.
- **LaTeX Make Errors**:
  - Ensure a LaTeX distribution is installed and properly set up.
  - Check the `make` command output for details on missing dependencies.

3 Data Structure Index

3.1 Data Structures

Here are the data structures with brief descriptions:

4 File Index

4.1 File List

Here is a list of all files with brief descriptions:

5 Data Structure Documentation

5.1 BoardState Struct Reference

Stores the current state of the Tic-Tac-Toe board along with the best move.
`#include <minimax.h>`

Data Fields

- int board [3][3]
- struct Position bestMove

5.1.1 Detailed Description

Stores the current state of the Tic-Tac-Toe board along with the best move.
< Include macro definitions
This structure holds a 3x3 board array and the best move associated with that board state. It is used for storing and comparing previous board states and their corresponding optimal moves.
Definition at line 29 of file minimax.h.

5.1.2 Field Documentation

bestMove

```
struct Position bestMove
```

The best move for the bot
Definition at line 32 of file minimax.h.

board

```
int board[3][3]
```

The Tic-Tac-Toe board
Definition at line 31 of file minimax.h.
The documentation for this struct was generated from the following file:

- header/minimax.h

5.2 BtnPos Struct Reference

Stores the position of a button in the game grid.
`#include <main.h>`

Data Fields

- int pos [2]

5.2.1 Detailed Description

Stores the position of a button in the game grid.
This structure contains an array `pos[2]` that holds the row and column of the button in the grid.
Definition at line 80 of file main.h.

5.2.2 Field Documentation

pos

```
int pos[2]
```

Array to hold row and column
Definition at line 82 of file main.h.
The documentation for this struct was generated from the following file:

- header/main.h

5.3 Dataset Struct Reference

Structure to hold a Tic-Tac-Toe board state and its outcome.
`#include <importData.h>`

Data Fields

- char grid [3][3]
- char outcome [9]

5.3.1 Detailed Description

Structure to hold a Tic-Tac-Toe board state and its outcome.
This structure represents the board state as a 3x3 grid and the outcome as a string.
Definition at line 28 of file importData.h.

5.3.2 Field Documentation

grid

```
char grid[3][3]
```

3x3 grid representing the board state
Definition at line 30 of file importData.h.

outcome

```
char outcome[9]
```

Outcome of the board state
Definition at line 31 of file importData.h.
The documentation for this struct was generated from the following file:

- header/importData.h

5.4 PlayerMode Struct Reference

Stores the current game mode and its textual representation.
`#include <main.h>`

Data Fields

- char txt [2]
- int mode

5.4.1 Detailed Description

Stores the current game mode and its textual representation.
This structure holds the mode of the game (e.g., player vs player, player vs bot) and a textual representation of the mode for display.
Definition at line 92 of file main.h.

5.4.2 Field Documentation

mode

```
int mode
```

Integer value representing the current game mode
Definition at line 95 of file main.h.

txt

```
char txt[2]
```

Textual representation of the current game mode
Definition at line 94 of file main.h.
The documentation for this struct was generated from the following file:

- header/main.h

5.5 Position Struct Reference

Represents a position on the Tic-Tac-Toe grid.

```
#include <macros.h>
```

Data Fields

- int row
- int col

5.5.1 Detailed Description

Represents a position on the Tic-Tac-Toe grid.
Definition at line 65 of file macros.h.

5.5.2 Field Documentation

col

```
int col
```

Column index (0-2)
Definition at line 68 of file macros.h.

row

```
int row
```

Row index (0-2)
Definition at line 67 of file macros.h.
The documentation for this struct was generated from the following file:

- header/macros.h

6 File Documentation

6.1 Documentation/README.md File Reference

6.2 header/elapsedTime.h File Reference

Provides functionality for measuring elapsed time for profiling purposes.

```
#include <macros.h>
#include <sys/time.h>
```

Functions

- void startElapseTime ()

  Starts the elapsed time tracking.
- void stopElapseTime (char ∗str)

  Stops the elapsed time tracking and outputs the result.

6.2.1 Detailed Description

Provides functionality for measuring elapsed time for profiling purposes.

Author

jacktan-jk

Version

1.0

Date

2024-11-13

Copyright

Copyright (c) 2024

This header file declares functions and macros to track the start and stop times of operations, enabling performance measurement. The code is conditionally compiled based on the `DISABLE_ELAPSED` macro, allowing profiling code to be included or excluded as needed.
Definition in file elapsedTime.h.

6.2.2 Function Documentation

startElapseTime()

```
void startElapseTime ()
```

Starts the elapsed time tracking.
Captures the current time and stores it in `gStartTime` to mark the beginning of an elapsed time measurement.
Only operates if `DISABLE_ELAPSED` is not defined, allowing conditional compilation for performance tracking.
Definition at line 9 of file elapsedTime.c.

stopElapseTime()

```
void stopElapseTime (
             char * str)
```

Stops the elapsed time tracking and outputs the result.
Calculates the time elapsed since `startElapseTime` and outputs it in seconds using the provided label.

Parameters

str	Label describing the operation or section being timed.

Only operates if `DISABLE_ELAPSED` is not defined, and outputs timing information through `PRINT_DEBUG` for profiling and debugging.
Definition at line 17 of file elapsedTime.c.

6.3 elapsedTime.h

Go to the documentation of this file.

```

#ifndef ELAPSED_TIME_H
#define ELAPSED_TIME_H

#include <macros.h>
#include <sys/time.h>

void startElapseTime();

void stopElapseTime(char *str);

#endif // ELAPSED_TIME_H
```

6.4 header/importData.h File Reference

Header file for handling dataset import and manipulation for Tic-Tac-Toe game data.

`#include <macros.h>`

Data Structures

- struct Dataset

  Structure to hold a Tic-Tac-Toe board state and its outcome.

Macros

- #define RES_PATH "./resources/"
- #define DATA_PATH "tic-tac-toe.data"
- #define TRAIN_PATH "training-"
- #define TEST_PATH "testing-"

Functions

- int readDataset (const char *filename, bool split)

  Reads a dataset from a file and optionally randomizes entries for training and testing.
- int splitFile ()

  Splits the dataset into training and testing files with an 80-20 ratio.
- void getRandomNo (int random[DATA_SIZE])

  Generates an array of unique random integers within the range of the dataset size.
- int getTrainingData (struct Dataset **d)

  Retrieves the training data from a file and returns its length.
- int getTestingData (struct Dataset **d)

  Retrieves the testing data from a file and returns its length.

6.4.1 Detailed Description

Header file for handling dataset import and manipulation for Tic-Tac-Toe game data.

Author

jacktan-jk

Version

1.0

Date

2024-11-12

This file contains function declarations and structures to read, split, and manage Tic-Tac-Toe game data used for training and testing.

Definition in file importData.h.

6.4.2 Macro Definition Documentation

DATA_PATH

```
#define DATA_PATH "tic-tac-toe.data"
```

Name of the primary dataset file
Definition at line 18 of file importData.h.

RES_PATH

```
#define RES_PATH "./resources/"
```

Path to resources directory
Definition at line 17 of file importData.h.

TEST_PATH

```
#define TEST_PATH "testing-"
```

Prefix for testing data file
Definition at line 20 of file importData.h.

TRAIN_PATH

```
#define TRAIN_PATH "training-"
```

Prefix for training data file
Definition at line 19 of file importData.h.

6.4.3 Function Documentation

getRandomNo()

```
void getRandomNo (
             int random[DATA_SIZE])
```

Generates an array of unique random integers within the range of the dataset size.
This function populates an array with unique random integers between 0 and `DATA_SIZE - 1`. It ensures that each integer appears only once by checking a `check` array to track used indices. This can be used for randomizing the order of data for splitting purposes.

Parameters

random	Array to store the generated unique random integers.

See also

DATA_SIZE

Definition at line 143 of file importData.c.

getTestingData()

```
int getTestingData (
             struct Dataset ** d)
```

Retrieves the testing data from a file and returns its length.
This function zeroes out the `data` array for the length of the testing set, reads the dataset from the specified `testingFile`, and assigns the `data` pointer to the provided dataset pointer. Returns the length of the testing data loaded.

Parameters

d	Pointer to a dataset pointer that will reference the loaded testing data array.

Returns

The number of testing entries loaded (i.e., `len_test`).

See also

readDataset, testingFile

Definition at line 175 of file importData.c.

getTrainingData()

```
int getTrainingData (
            struct Dataset ** d)
```

Retrieves the training data from a file and returns its length.

This function initializes the `data` array to zero for the length of the training set, reads the dataset from the specified `trainingFile`, and assigns the `data` pointer to the provided dataset pointer. Returns the length of the training data loaded.

Parameters

d	Pointer to a dataset pointer that will reference the loaded training data array.

Returns

The number of training entries loaded (i.e., `len_train`).

See also

readDataset, trainingFile

Definition at line 167 of file importData.c.

readDataset()

```
int readDataset (
            const char * filename,
            bool split)
```

Reads a dataset from a file and optionally randomizes entries for training and testing.

Opens a file to read each line as a Tic Tac Toe board state, populating a grid structure where 'x', 'o', and 'b' represent the Bot, Player 1, and empty cells, respectively. Each board state is followed by an outcome that is stored within the dataset. If `split` is true, entries are randomized using an array of unique indices for shuffling.

Parameters

filename	The name of the dataset file to read.
split	Boolean indicating whether to randomize entries for dataset splitting.

Returns

int SUCCESS (0) if reading is successful, BAD_PARAM (-5) if the file cannot be opened, or the return value of `splitFile()` if `split` is enabled.

See also

getRandomNo, splitFile

Definition at line 44 of file importData.c.

splitFile()

`int splitFile ()`

Splits the dataset into training and testing files with an 80-20 ratio.

This function separates the dataset into two parts: 80% for training and 20% for testing. The training portion is written to `trainingFile`, and the testing portion is written to `testingFile`. Each entry consists of a 3x3 grid representing the Tic Tac Toe board and the outcome of that board.

Returns

int SUCCESS (0) if both files are written successfully, BAD_PARAM (-5) if either file cannot be opened.

See also

data, trainingFile, testingFile

Definition at line 90 of file importData.c.

6.5 importData.h

Go to the documentation of this file.

```
#ifndef IMPORTDATA_H
#define IMPORTDATA_H

#include <macros.h>

// changed to 100 for testing. make sure to chg back
#define RES_PATH "./resources/"
#define DATA_PATH "tic-tac-toe.data"
#define TRAIN_PATH "training-"
#define TEST_PATH "testing-"
struct Dataset
{
    char grid[3][3];
    char outcome[9];
};

int readDataset(const char *filename, bool split);

int splitFile();

void getRandomNo(int random[DATA_SIZE]);

int getTrainingData(struct Dataset **d);

int getTestingData(struct Dataset **d);

#endif // IMPORTDATA_H
```

6.6 header/macros.h File Reference

Header file containing macros, constants, and structure definitions for the Tic-Tac-Toe game.

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include <time.h>
#include <string.h>
```

Data Structures

- struct Position

  Represents a position on the Tic-Tac-Toe grid.

Macros

- #define PLAY 0x69
- #define TIE 0xFF

- #define WIN 0xAA
- #define SUCCESS 0
- #define ERROR -1
- #define BAD_PARAM -5
- #define MODE_2P 0
- #define MODE_MM 1
- #define MODE_ML 2
- #define EMPTY 0
- #define PLAYER1 1
- #define BOT 2
- #define DATA_SIZE 958
- #define CLASSES 2
- #define DEBUG 1
- #define MINIMAX_GODMODE
- #define DISABLE_LOOKUP
- #define DISABLE_ELAPSED
- #define DISABLE_ASM
- #define PRINT_DEBUG(...)

6.6.1 Detailed Description

Header file containing macros, constants, and structure definitions for the Tic-Tac-Toe game.

Author

jacktan-jk

Version

1.0

Date

2024-11-13

Copyright

Copyright (c) 2024

This file defines various constants for game modes, player identifiers, error codes, and debugging options. It also includes the `Position` structure for grid positions in the game.
Definition in file macros.h.

6.6.2 Macro Definition Documentation

BAD_PARAM

```
#define BAD_PARAM -5
```

Bad parameter error
Definition at line 30 of file macros.h.

BOT

```
#define BOT 2
```

Bot (Player 2)
Definition at line 40 of file macros.h.

CLASSES

```
#define CLASSES 2
```

Number of outcome classes
Definition at line 44 of file macros.h.

DATA_SIZE

```
#define DATA_SIZE 958
```

Dataset size

Definition at line 43 of file macros.h.

DEBUG

```
#define DEBUG 1
```

Enable debug messages

Definition at line 47 of file macros.h.

DISABLE_ASM

```
#define DISABLE_ASM
```

Disable ASM functions

Definition at line 51 of file macros.h.

DISABLE_ELAPSED

```
#define DISABLE_ELAPSED
```

Disable Elapsed time function

Definition at line 50 of file macros.h.

DISABLE_LOOKUP

```
#define DISABLE_LOOKUP
```

Disable Minimax lookup table

Definition at line 49 of file macros.h.

EMPTY

```
#define EMPTY 0
```

Empty cell

Definition at line 38 of file macros.h.

ERROR

```
#define ERROR -1
```

Error

Definition at line 29 of file macros.h.

MINIMAX_GODMODE

```
#define MINIMAX_GODMODE
```

Minimax god mode toggle

Definition at line 48 of file macros.h.

MODE_2P

```
#define MODE_2P 0
```

Two-player mode

Definition at line 33 of file macros.h.

MODE_ML

```
#define MODE_ML 2
```

Machine Learning mode

Definition at line 35 of file macros.h.

MODE_MM

```
#define MODE_MM 1
```

Minimax mode

Definition at line 34 of file macros.h.

PLAY

```
#define PLAY 0x69
```

Player move

Definition at line 25 of file macros.h.

PLAYER1

```
#define PLAYER1 1
```

Player 1

Definition at line 39 of file macros.h.

PRINT_DEBUG

```
#define PRINT_DEBUG(
                ...)
```

Value:

```
printf(__VA_ARGS__);
```

Definition at line 54 of file macros.h.

SUCCESS

```
#define SUCCESS 0
```

Success

Definition at line 28 of file macros.h.

TIE

```
#define TIE 0xFF
```

Tie state

Definition at line 26 of file macros.h.

WIN

```
#define WIN 0xAA
```

Winning state

Definition at line 27 of file macros.h.

6.7 macros.h

Go to the documentation of this file.

```
00001
00015 #ifndef MACROS_H
00016 #define MACROS_H
00017
00018 #include <stdio.h>
00019 #include <stdlib.h>
00020 #include <stdbool.h>
00021 #include <time.h>
00022 #include <string.h>
00023
00024 // Constants for game states and player identifiers
00025 #define PLAY 0x69
00026 #define TIE 0xFF
00027 #define WIN 0xAA
00028 #define SUCCESS 0
00029 #define ERROR -1
00030 #define BAD_PARAM -5
00032 // Game modes
00033 #define MODE_2P 0
00034 #define MODE_MM 1
```

```
#define MODE_ML 2
// Player identifiers
#define EMPTY 0
#define PLAYER1 1
#define BOT 2
// Data constants
#define DATA_SIZE 958
#define CLASSES 2
// Debugging and configuration options
#define DEBUG 1
#define MINIMAX_GODMODE 0
#define DISABLE_LOOKUP   0
#define DISABLE_ELAPSED 0
#define DISABLE_ASM     0
#if DEBUG
#define PRINT_DEBUG(...) printf(__VA_ARGS__);
#else
#define PRINT_DEBUG(...)
#endif

struct Position
{
    int row;
    int col;
};

#endif // MACROS_H
```

6.8 header/main.h File Reference

Header file for the Tic-Tac-Toe game logic.

```
#include <gtk/gtk.h>
#include <macros.h>
#include <minimax.h>
#include <ml-naive-bayes.h>
#include <elapsedTime.h>
```

Data Structures

- struct BtnPos

  Stores the position of a button in the game grid.
- struct PlayerMode

  Stores the current game mode and its textual representation.

Functions

- int doBOTmove ()

  Executes the bot's move based on the current game mode.
- int chkPlayerWin ()

  Checks the current game board for a win or tie.
- void clearGrid ()

  Clears the game board and resets the player's turn.
- void updateScoreBtn (gpointer data)

  Updates the score display on the button.
- void on_btnGrid_clicked (GtkWidget ∗widget, gpointer data)

  Callback function for handling button clicks on the game grid.
- void on_btnScore_clicked (GtkWidget ∗widget, gpointer data)

  Handles button click for score.
- void showWin ()

  Clears the winning positions and resets the grid.

6.8.1 Detailed Description

Header file for the Tic-Tac-Toe game logic.

Author

jacktan-jk

Version

1.0

Date

2024-11-12

Copyright

Copyright (c) 2024

This file contains function declarations, structure definitions, and constants for handling game logic, button interactions, and the bot's behavior.
Definition in file main.h.

6.8.2 Function Documentation

chkPlayerWin()

```
int chkPlayerWin ()
```

Checks the current game board for a win or tie.
This function checks all possible win conditions:

- Diagonals
- Rows
- Columns

If there is a winning line, it marks the winning positions and returns WIN. If there are no winning conditions and the board is full, it returns TIE. If there are unclicked positions left, it returns PLAY.

Returns

WIN if there is a winner, TIE if the game is a tie, PLAY if the game is still ongoing.

See also

iBoard, iWinPos

Definition at line 234 of file main.c.

clearGrid()

```
void clearGrid ()
```

Clears the game board and resets the player's turn.
This function is used to reset the game board for a new round. It clears the labels on the buttons in the grid and resets the internal board state (iBoard) to 0. It also sets the player turn back to player 1.

- Sets all button labels in the `btnGrid` to an empty string.
- Resets all values in the `iBoard` array to 0, indicating no moves.
- Resets `isPlayer1Turn` to `true`, indicating it's Player 1's turn.

See also

iBoard
btnGrid
isPlayer1Turn

Definition at line 29 of file main.c.

doBOTmove()

```
int doBOTmove ()
```

Executes the bot's move based on the current game mode.
In MM mode:

- Performs a minimax move.
- 20% chance of the minimax randomly selects a position.

In ML mode, the bot uses machine learning to determine the best position.
The function also measures and logs the time taken for the minimax move.

Returns

SUCCESS if the bot's move was made successfully.

See also

playerMode, isMLAvail, iBoard, findBestMove, getBestPosition, btnGrid

Definition at line 181 of file main.c.

on_btnGrid_clicked()

```
void on_btnGrid_clicked (
             GtkWidget * widget,
             gpointer data)
```

Callback function for handling button clicks on the game grid.
This function handles the logic for a player's move when a button in the game grid is clicked. It updates the game state, checks for a winner or tie, and updates the score display. It also handles player turns, Bot moves (if applicable), and resets the game board when the game state changes.

Parameters

widget	The GtkWidget that was clicked (the button in the grid).
data	Additional data passed to the callback (usually the score display data).

- If the game state is not `PLAY`, the game will be reset, and the score updated.
- If the clicked button already has a label, the function returns early (no action is taken).
- If the clicked button is empty, the move is recorded in the `iBoard` array (Player 1 or MM or ML).
- After each move, the game checks for a win or tie condition using `chkPlayerWin()`.
- If Player 1 or Player 2 wins, the score is updated, and the win condition is shown.
- If the game ends in a tie, the tie score is updated.
- If the game is in **2P** mode, turns alternate between Player 1 and Player 2.
- In **MM mode**, the Minimax will automatically make a move after Player 1's turn.
- In **ML mode**, the dataset is re-read and initialized after the game ends.

See also

iBoard, isPlayer1Turn, iPlayer1_score, iPlayer2_score, iTie_score

playerMode, updateScoreBtn, chkPlayerWin, doBOTmove, showWin

PLAY, TIE, WIN

Definition at line 60 of file main.c.

on_btnScore_clicked()

```
void on_btnScore_clicked (
             GtkWidget * widget,
             gpointer data)
```

Handles button click for score.

Toggles the player mode and updates the displayed score.

Parameters

widget	The widget that triggered the event.
data	Additional data passed to the callback.

See also

playerMode, isMLAvail, isPlayer1Turn, updateScoreBtn, clearGrid

Definition at line 131 of file main.c.

showWin()

```
void showWin ()
```

Clears the winning positions and resets the grid.

Iterates over the win positions and clears any displayed labels, resetting the grid to its initial state.

See also

iWinPos, btnGrid

Definition at line 159 of file main.c.

updateScoreBtn()

```
void updateScoreBtn (
             gpointer data)
```

Updates the score display on the button.

This function updates the label on a score button to display the current scores for Player 1, Player 2, and Ties. It changes the text formatting depending on which player's turn it is, highlighting the active player.

Parameters

data	A gpointer (usually a button widget) that is used to update the label.

- The function checks if it's Player 1's turn and updates the score display with a bold label for Player 1, or Player 2’s turn with Player 2's score in bold.
- The button text is updated using `gtk_button_set_label()`, and the label markup is updated using `gtk_label_set_markup()`.
- The score includes Player 1's score, Player 2's score, the tie count, and the current game mode (`playerMode.txt`).

See also

iPlayer1_score, iTie_score, iPlayer2_score, playerMode

Definition at line 42 of file main.c.

6.9 main.h

Go to the documentation of this file.

```c

#ifndef MAIN_H // Start of include guard
#define MAIN_H
#include <gtk/gtk.h>

#include <macros.h>
#include <minimax.h>
#include <ml-naive-bayes.h>
#include <elapsedTime.h>

/*===============================================================================================
GLOBAL DECLARATION
===============================================================================================*/
typedef struct
{
    int pos[2];
} BtnPos;

struct PlayerMode
{
    char txt[2];
    int mode;
};

int doBOTmove();

int chkPlayerWin();

void clearGrid();

void updateScoreBtn(gpointer data);

void on_btnGrid_clicked(GtkWidget *widget, gpointer data);

void on_btnScore_clicked(GtkWidget *widget, gpointer data);

void showWin();

#endif // MAIN_H  // End of include guard
```

6.10 header/minimax.h File Reference

Header file for the Tic-Tac-Toe Minimax algorithm.

```c
#include <macros.h>
#include <elapsedTime.h>
```

Data Structures

- struct BoardState

  Stores the current state of the Tic-Tac-Toe board along with the best move.

Macros

- #define FILE_BESTMOV "resources/bestmove.txt"
- #define MAX_BOARDS 10000

Functions

- int max (int a, int b)

  Returns the maximum of two integers.
- int min (int a, int b)

  Returns the minimum of two integers.
- struct Position findBestMove (int board[3][3])

  Finds the best move for the bot in the Tic-Tac-Toe game.
- int minimax (int board[3][3], int depth, bool isMax)

  Implements the Minimax algorithm to evaluate the best move for the bot.
- int evaluate (int b[3][3])

Evaluates the current board state to determine if there is a winner.

- bool isMovesLeft (int board[3][3])

  Checks if there are any moves left on the board.
- bool checkAndUpdateBestMove (int board[3][3], struct Position ∗bestMove, struct BoardState boardStates[ ], int count)

  Checks if the current board configuration exists in the lookup table and updates the best move.
- void writeBestMoveToFile (int board[3][3], struct Position bestMove)

  Appends the current board state and the best move to a file.
- int loadBoardStates (struct BoardState boardStates[ ])

  Loads board states and their best moves from a file.
- void printFileContents ()

  Prints the contents of the best move file.

6.10.1 Detailed Description

Header file for the Tic-Tac-Toe Minimax algorithm.

Author

jacktan-jk

Version

1.0

Date

2024-11-12

Copyright

Copyright (c) 2024

This file contains function declarations, structure definitions, and constants for implementing the Minimax algorithm in a Tic-Tac-Toe game. It includes functions for evaluating board states, calculating the best move for the bot, and checking if any moves are left on the board.
Definition in file minimax.h.

6.10.2 Macro Definition Documentation

FILE_BESTMOV

```
#define FILE_BESTMOV "resources/bestmove.txt"
```

Path to the file storing best moves
Definition at line 35 of file minimax.h.

MAX_BOARDS

```
#define MAX_BOARDS 10000
```

Maximum number of boards to store in memory
Definition at line 36 of file minimax.h.

6.10.3 Function Documentation

checkAndUpdateBestMove()

```
bool checkAndUpdateBestMove (
             int board[3][3],
             struct Position * bestMove,
             struct BoardState boardStates[],
             int count)
```

Checks if the current board configuration exists in the lookup table and updates the best move.

This function compares the current board with previously saved board states in the `boardStates` array. If a matching board configuration is found, it updates the provided `bestMove` structure with the best move associated with that board state. The function returns true if a match is found and the move is updated, and false if no match is found in the lookup table.

Parameters

board	The current Tic Tac Toe board to check against the saved states.
bestMove	A pointer to the `Position` structure where the best move will be stored if a match is found.
boardStates	An array of `BoardState` structures containing previously saved board configurations and their best moves.
count	The number of saved board states in the `boardStates` array.

Returns

`true` if a matching board configuration is found and the best move is updated, `false` otherwise.

See also

BoardState, Position

Definition at line 419 of file minimax.c.

evaluate()

```
int evaluate (
            int b[3][3])
```

Evaluates the current board state to determine if there is a winner.

This function checks the Tic-Tac-Toe board for winning conditions, i.e., it checks rows, columns, and diagonals for three consecutive marks (either `BOT` or `PLAYER1`). It returns a score based on the result:

- +10 if the `BOT` wins.
- -10 if `PLAYER1` wins.
- 0 if there is no winner yet (no winner in rows, columns, or diagonals).

Parameters

b	A 3x3 array representing the Tic-Tac-Toe board.

Returns

The evaluation score:

- +10 for a `BOT` win,
- -10 for a `PLAYER1` win,
- 0 if there is no winner.

See also

BOT, PLAYER1

Definition at line 220 of file minimax.c.

findBestMove()

```
struct Position findBestMove (
              int board[3][3])
```

Finds the best move for the bot in the Tic-Tac-Toe game.
This function first checks if the best move is already stored in memory by looking through previous board states. If the move is found, it is returned. If no best move is found in memory, it traverses all the empty cells on the board, evaluates the potential moves using the minimax algorithm, and returns the optimal move.

Parameters

board	A 3x3 array representing the current Tic-Tac-Toe board.

Returns

The best move for the bot as a struct Position containing the row and column.

See also

minimax, loadBoardStates, checkAndUpdateBestMove, writeBestMoveToFile

Definition at line 71 of file minimax.c.

isMovesLeft()

```
bool isMovesLeft (
              int board[3][3])
```

Checks if there are any moves left on the board.
This function determines if there are any empty cells left on a 3x3 board. It uses one of the following implementations based on compilation options:

- A standard C implementation when `DISABLE_ASM` is defined.
- Optimized assembly instructions for either AArch64 or x86 platforms.

The choice of implementation depends on the platform and preprocessor directives.

Parameters

board	A 3x3 array representing the board state. Each cell should contain: • `EMPTY` (typically 0) if the cell is empty. • Any non-zero value if the cell is occupied.

Returns

`true` if there are empty cells; otherwise, `false`.

Note

If `DISABLE_ASM` is defined, the function uses pure C logic.

For AArch64 platforms, the function uses assembly instructions for efficient scanning.

For x86 platforms, the function uses assembly instructions for efficient scanning.

Example usage:

```
int board[3][3] = {
    {1, 2, 0},
    {0, 1, 2},
    {2, 1, 0}
};
bool movesLeft = isMovesLeft(board);
// movesLeft will be true as there are empty cells (0s).
```

Definition at line 270 of file minimax.c.

loadBoardStates()

```
int loadBoardStates (
            struct BoardState boardStates[])
```

Loads board states and their best moves from a file.

This function attempts to open a file containing saved board states and the corresponding best move for each state. If the file does not exist, a new file is created. It reads the board configurations and the best move for each board, storing them in the provided `boardStates` array.

Each line in the file represents one board state. The board is stored as a 3x3 grid, where 'x' denotes the BOT's move, 'o' denotes PLAYER1's move, and empty spaces are represented as ' ' (empty). The best move for each board is also saved in the file.

Parameters

boardStates	An array of `BoardState` structures to store the loaded board states.

Returns

The number of boards loaded from the file. If the file does not exist, it returns 0 and creates a new file.

See also

BoardState, FILE_BESTMOV

Definition at line 366 of file minimax.c.

max()

```
int max (
            int a,
            int b)
```

Returns the maximum of two integers.

This function computes the maximum of two integers using either:

- A standard C implementation when `DISABLE_ASM` is defined.
- Optimized assembly instructions for either AArch64 or x86 platforms.

The choice of implementation depends on the platform and preprocessor directives.

Parameters

a	The first integer to compare.
b	The second integer to compare.

Returns

The greater of the two integers.

Note

If `DISABLE_ASM` is defined, the function uses pure C logic.

For AArch64 platforms, the function uses assembly instructions with conditional selection.

For x86 platforms, the function uses assembly instructions with register manipulation and branching.

Example usage:

```
int result = max(10, 20);
// result now holds the value 20.
```

Definition at line 5 of file minimax.c.

min()

```
int min (
            int a,
            int b)
```

Returns the minimum of two integers.

This function computes the minimum of two integers using either:

- A standard C implementation when `DISABLE_ASM` is defined.
- Optimized assembly instructions for either AArch64 or x86 platforms.

The choice of implementation depends on the platform and preprocessor directives.

Parameters

a	The first integer to compare.
b	The second integer to compare.

Returns

The smaller of the two integers.

Note

If `DISABLE_ASM` is defined, the function uses pure C logic.

For AArch64 platforms, the function uses assembly instructions with conditional selection.

For x86 platforms, the function uses assembly instructions with register manipulation and branching.

Example usage:

```
int result = min(10, 20);
// result now holds the value 10.
```

Definition at line 38 of file minimax.c.

minimax()

```
int minimax (
            int board[3][3],
            int depth,
            bool isMax)
```

Implements the Minimax algorithm to evaluate the best move for the bot.

The function recursively evaluates all possible moves using the Minimax algorithm. It returns the best score for the current player (maximizer or minimizer) based on the game state. The algorithm chooses the optimal move for the bot and evaluates the game state at each depth. The depth is capped if Minimax Godmode is not enabled. If there are no moves left or the game is over, it returns the evaluation score.

Parameters

board	A 3x3 array representing the current Tic-Tac-Toe board.
depth	The current depth in the game tree.
isMax	Boolean flag indicating whether it is the maximizer's turn (bot) or the minimizer's turn (player).

Returns

The best score for the current move based on the evaluation function.

See also

evaluate, isMovesLeft, max, min

Definition at line 136 of file minimax.c.

printFileContents()

```
void printFileContents ()
```

Prints the contents of the best move file.
This function reads and prints the contents of the file that stores the best moves for various board states.

writeBestMoveToFile()

```
void writeBestMoveToFile (
             int board[3][3],
             struct Position bestMove)
```

Appends the current board state and the best move to a file.
This function writes the current Tic Tac Toe board state to a file, encoding the board as a sequence of characters where 'o' represents Player 1, 'x' represents the Bot, and 'b' represents an empty cell. After writing the board state, it appends the best move (row and column) for the current board to the same file.

Parameters

board	The current Tic Tac Toe board to write to the file.
bestMove	The best move to be made, represented by its row and column indices.

See also

Position, BoardState

Definition at line 436 of file minimax.c.

6.11 minimax.h

Go to the documentation of this file.

```

#ifndef MINIMAX_H // Start of include guard
#define MINIMAX_H

#include <macros.h>
#include <elapsedTime.h>

struct BoardState
{
    int board[3][3];
    struct Position bestMove;
};

#define FILE_BESTMOV "resources/bestmove.txt"
#define MAX_BOARDS 10000
int max(int a, int b);

int min(int a, int b);

struct Position findBestMove(int board[3][3]);

int minimax(int board[3][3], int depth, bool isMax);

int evaluate(int b[3][3]);

bool isMovesLeft(int board[3][3]);

bool checkAndUpdateBestMove(int board[3][3], struct Position *bestMove, struct BoardState
      boardStates[], int count);

void writeBestMoveToFile(int board[3][3], struct Position bestMove);

int loadBoardStates(struct BoardState boardStates[]);

void printFileContents();

#endif // MINIMAX_H // End of include guard
```

6.12 header/ml-naive-bayes.h File Reference

Header file for Naive Bayes classifier functions for Tic-Tac-Toe outcome prediction.

```
#include <macros.h>
#include <importData.h>
```

Macros

- #define TRAINING_DATA_SIZE 0.8 ∗ DATA_SIZE
- #define TESTING_DATA_SIZE 0.2 ∗ DATA_SIZE
- #define CLASSES 2

Functions

- int assignMoveIndex (char move)

  Assigns an index to each move ("x", "o", or "b").
- void calculateProbabilities (int dataset_size)

  Calculates the probabilities for each class and conditional probabilities with Laplace smoothing.
- void resetTrainingData ()

  Resets the training data and associated statistics for a fresh training cycle.
- int initData ()

  Initializes the training data and model statistics.
- int predictOutcome (struct Dataset board)

  Predicts the outcome of a given Tic Tac Toe board based on previously calculated probabilities.
- void calcTrainErrors ()

  Calculates the training errors and the probability of error.
- void calcConfusionMatrix ()

  Calculates the confusion matrix and error probability for the testing dataset.
- struct Position getBestPosition (int grid[3][3], char player)

  Determines the best position for the bot to make a move based on the highest probability.
- int getTruthValue (char ∗str1)

  Returns an integer value representing a truth value based on input.
- void assignCMValue (int actual, int predicted)

  Updates the confusion matrix based on actual and predicted outcomes.
- void debugDataset (struct Dataset ∗data, int len)

  Debug function to display dataset contents.

6.12.1 Detailed Description

Header file for Naive Bayes classifier functions for Tic-Tac-Toe outcome prediction.

Author

jacktan-jk

Version

1.0

Date

2024-11-13

Copyright

Copyright (c) 2024

This file contains function declarations, structure definitions, and global variables for implementing the Naive Bayes classification model for predicting outcomes in a Tic-Tac-Toe game. It includes functions for initializing training data, calculating probabilities, predicting outcomes based on the trained model, calculating error rates, and updating the confusion matrix. The model is trained using a dataset of game board states and outcomes, and is used to predict the outcome of new game states.

Definition in file ml-naive-bayes.h.

6.12.2 Macro Definition Documentation

CLASSES

```
#define CLASSES 2
```

Number of possible outcome classes (positive/negative)
Definition at line 26 of file ml-naive-bayes.h.

TESTING_DATA_SIZE

```
#define TESTING_DATA_SIZE 0.2 * DATA_SIZE
```

Size of the testing dataset (20%)
Definition at line 25 of file ml-naive-bayes.h.

TRAINING_DATA_SIZE

```
#define TRAINING_DATA_SIZE 0.8 * DATA_SIZE
```

Size of the training dataset (80%)
Definition at line 24 of file ml-naive-bayes.h.

6.12.3 Function Documentation

assignCMValue()

```
void assignCMValue (
             int actual,
             int predicted)
```

Updates the confusion matrix based on actual and predicted outcomes.
This function updates the confusion matrix counters for true positives, false negatives, false positives, and true negatives. It checks the actual and predicted outcomes and increments the appropriate counter in the confusion matrix.
If either the actual or predicted value is ERROR, an error is logged.

Parameters

actual	The actual outcome value (1 for positive, 0 for negative).
predicted	The predicted outcome value (1 for positive, 0 for negative).

See also

cM, ERROR

Definition at line 340 of file ml-naive-bayes.c.

assignMoveIndex()

```
int assignMoveIndex (
             char move)
```

Assigns an index to each move ("x", "o", or "b").
This function maps the board move characters to their corresponding integer values:

- 'x' is mapped to the BOT.
- 'o' is mapped to PLAYER1.
- 'b' is mapped to EMPTY. If the character does not match any of the valid moves, -1 is returned.

Parameters

move	The character representing the move ('x', 'o', or 'b').

Returns

int The integer corresponding to the move:

- BOT for 'x',
- PLAYER1 for 'o',
- EMPTY for 'b',
- ERROR for invalid input.

See also

BOT, PLAYER1, EMPTY, ERROR

Definition at line 21 of file ml-naive-bayes.c.

calcConfusionMatrix()

```
void calcConfusionMatrix ()
```

Calculates the confusion matrix and error probability for the testing dataset.

This function evaluates the model's performance by calculating the confusion matrix based on actual and predicted outcomes. It iterates through the testing data, compares actual outcomes with predicted ones, and updates the confusion matrix values. The number of prediction errors and the probability of error are also computed.

See also

cM, test_PredictedErrors, probabilityErrors, getTruthValue, predictOutcome

Definition at line 373 of file ml-naive-bayes.c.

calcTrainErrors()

```
void calcTrainErrors ()
```

Calculates the training errors and the probability of error.

This function evaluates the model's performance on the training dataset by comparing predicted outcomes with actual ones. It updates the count of prediction errors and computes the probability of error based on the number of errors and the size of the training dataset.

See also

train_PredictedErrors, probabilityErrors, getTruthValue, predictOutcome

Definition at line 424 of file ml-naive-bayes.c.

calculateProbabilities()

```
void calculateProbabilities (
             int dataset_size)
```

Calculates the probabilities for each class and conditional probabilities with Laplace smoothing.

This function calculates:

- The class probabilities for positive and negative outcomes.
- The conditional probabilities for each move ('x', 'o', 'b') at each position on the board, given the class (positive or negative) with Laplace smoothing applied.

The Laplace smoothing is used to prevent zero probabilities for moves that may not have been observed in the training data. The resulting probabilities are printed for debugging purposes.

Parameters

dataset_size	The total number of samples in the dataset used for probability calculation.

See also

positive_count, negative_count, positiveMoveCount, negativeMoveCount

Definition at line 36 of file ml-naive-bayes.c.

debugDataset()

```
void debugDataset (
            struct Dataset * data,
            int len)
```

Debug function to display dataset contents.

This function prints the details of the provided dataset, including the grid values and the corresponding outcomes. It is primarily used for debugging purposes and is not currently in use within the code.

Parameters

data	Pointer to the dataset to be printed.
len	The length of the dataset (number of entries).

See also

PRINT_DEBUG

Definition at line 451 of file ml-naive-bayes.c.

getBestPosition()

```
struct Position getBestPosition (
            int grid[3][3],
            char player)
```

Determines the best position for the bot to make a move based on the highest probability.

This function evaluates all empty positions on the Tic Tac Toe grid and calculates the probability of the bot winning (either as 'x' or 'o') using the pre-calculated move probabilities from the training data. The bot chooses the position with the highest probability of winning, where the move is either 'x' or 'o' depending on the current player. It returns the best position for the bot to make its move.

Parameters

grid	The current state of the Tic Tac Toe game board.
player	The current player, either 'x' or 'o'.

Returns

A struct `Position` representing the row and column of the best move for the bot. If no valid move is found, it returns an error indicator.

See also

positive_count, negative_count, positiveMoveCount, negativeMoveCount

Definition at line 156 of file ml-naive-bayes.c.

getTruthValue()

```
int getTruthValue (
            char * str1)
```

Returns an integer value representing a truth value based on input.

This function converts a string input into a corresponding integer truth value. The input string is compared against "positive" and "negative" to determine the output:

- Returns `1` for "positive".
- Returns `0` for "negative".
- Returns `-1` for any other input, and logs an error message for invalid cases.

Parameters

str1	A pointer to the input string to be evaluated.

Returns

int The corresponding truth value:

- `1` for "positive".
- `0` for "negative".
- `−1` for invalid inputs.

See also

PRINT_DEBUG

Definition at line 406 of file ml-naive-bayes.c.

initData()

```
int initData ()
```

Initializes the training data and model statistics.

This function resets the training data, then retrieves the training dataset for model training. It processes the dataset to count occurrences of positive and negative outcomes and updates the move counts for each grid position based on the data. Afterward, it calculates training errors and updates the confusion matrix.

If the initial dataset is empty, it attempts to load the data again.

See also

resetTrainingData, getTrainingData, calcTrainErrors, calcConfusionMatrix

Definition at line 273 of file ml-naive-bayes.c.

predictOutcome()

```
int predictOutcome (
            struct Dataset board)
```

Predicts the outcome of a given Tic Tac Toe board based on previously calculated probabilities.

This function calculates the probabilities of a positive (Player 1 wins) or negative (Bot wins) outcome for a given board state by multiplying the conditional probabilities of each move in the grid with the class probabilities. The prediction is made based on which outcome (positive or negative) has the higher probability.

If the calculated probabilities are zero, indicating that the outcome cannot be predicted with the available data, the function returns -1.

Parameters

board	The current Tic Tac Toe board whose outcome needs to be predicted.

Returns

1 if the predicted outcome is positive (Player 1 wins), 0 if negative (Bot wins), and -1 if the outcome cannot be predicted.

See also

positiveClassProbability, negativeClassProbability, positiveMoveCount, negativeMoveCount, assignMoveIndex

Definition at line 88 of file ml-naive-bayes.c.

resetTrainingData()

`void resetTrainingData ()`

Resets the training data and associated statistics for a fresh training cycle.

This function resets all relevant variables used in the machine learning model's training process. It clears the outcome counts, resets the move count arrays for each grid position, and reinitializes the confusion matrix. Additionally, it clears the prediction error counters, ensuring that the model starts with a clean state.

See also

positive_count, negative_count, positiveMoveCount, negativeMoveCount, cM, test_PredictedErrors, train_PredictedErrors

Definition at line 247 of file ml-naive-bayes.c.

6.13 ml-naive-bayes.h

Go to the documentation of this file.

```

#ifndef ML_NAIVE_BAYES_H
#define ML_NAIVE_BAYES_H

#include <macros.h>
#include <importData.h>

#define TRAINING_DATA_SIZE 0.8 * DATA_SIZE
#define TESTING_DATA_SIZE 0.2 * DATA_SIZE
#define CLASSES 2
int assignMoveIndex(char move);

void calculateProbabilities(int dataset_size);

void resetTrainingData();

int initData();

int predictOutcome(struct Dataset board);

void calcTrainErrors();

void calcConfusionMatrix();

struct Position getBestPosition(int grid[3][3], char player);

int getTruthValue(char *str1);

void assignCMValue(int actual, int predicted);

void debugDataset(struct Dataset *data, int len);

#endif // ML_NAIVE_BAYES_H
```

6.14 mainpage.md File Reference

6.15 src/elapsedTime.c File Reference

`#include <elapsedTime.h>`

Functions

- void startElapseTime ()

  Starts the elapsed time tracking.
- void stopElapseTime (char *str)

  Stops the elapsed time tracking and outputs the result.

Variables

- struct timeval gTime
- double gStartTime
- double gEndTime

6.15.1 Function Documentation

startElapseTime()

`void startElapseTime ()`

Starts the elapsed time tracking.

Captures the current time and stores it in `gStartTime` to mark the beginning of an elapsed time measurement.

Only operates if `DISABLE_ELAPSED` is not defined, allowing conditional compilation for performance tracking.

Definition at line 9 of file elapsedTime.c.

stopElapseTime()

```
void stopElapseTime (
             char * str)
```

Stops the elapsed time tracking and outputs the result.

Calculates the time elapsed since `startElapseTime` and outputs it in seconds using the provided label.

Parameters

str	Label describing the operation or section being timed.

Only operates if `DISABLE_ELAPSED` is not defined, and outputs timing information through `PRINT_DEBUG` for profiling and debugging.

Definition at line 17 of file elapsedTime.c.

6.15.2 Variable Documentation

gEndTime

`double gEndTime`

Holds the end time in seconds for the timed section.

Definition at line 6 of file elapsedTime.c.

gStartTime

`double gStartTime`

Holds the start time in seconds for the timed section.

Definition at line 5 of file elapsedTime.c.

gTime

`struct timeval gTime`

Stores the current time values for elapsed time calculation.

Definition at line 4 of file elapsedTime.c.

6.16 elapsedTime.c

Go to the documentation of this file.

```
00001 #include <elapsedTime.h>
00002
00003 #if !(DISABLE_ELAPSED)
00004 struct timeval gTime;
00005 double gStartTime;
00006 double gEndTime;
00007 #endif
00008
00009 void startElapseTime()
00010 {
00011 #if !(DISABLE_ELAPSED)
00012     gettimeofday(&gTime, NULL);
00013     gStartTime = gTime.tv_sec + 1.0e-6 * gTime.tv_usec;
00014 #endif
00015 }
00016
00017 void stopElapseTime(char *str)
00018 {
```

```
#if !(DISABLE_ELAPSED)
    gettimeofday(&gTime, NULL);
    gEndTime = gTime.tv_sec + 1.0e-6 * gTime.tv_usec;
    PRINT_DEBUG("[ELAPSED] %s -> took %f seconds \n\n", str, (double)(gEndTime - gStartTime));
#endif
}
```

6.17 src/importData.c File Reference

`#include <importData.h>`

Functions

- int readDataset (const char ∗filename, bool split)

  Reads a dataset from a file and optionally randomizes entries for training and testing.
- int splitFile ()

  Splits the dataset into training and testing files with an 80-20 ratio.
- void getRandomNo (int random[DATA_SIZE])

  Generates an array of unique random integers within the range of the dataset size.
- int getTrainingData (struct Dataset ∗∗d)

  Retrieves the training data from a file and returns its length.
- int getTestingData (struct Dataset ∗∗d)

  Retrieves the testing data from a file and returns its length.

Variables

- int len_train = 0

  Global variable to store the number of training dataset entries.
- int len_test = 0

  Global variable to store the number of testing dataset entries.
- int randomNo [DATA_SIZE]

  Global array to store unique random indices for dataset splitting.
- struct Dataset data [DATA_SIZE]

  Global array to store the dataset.
- const char ∗ trainingFile = RES_PATH "" TRAIN_PATH "" DATA_PATH

  Global variable to store the path for the training dataset file.
- const char ∗ testingFile = RES_PATH "" TEST_PATH "" DATA_PATH

  Global variable to store the path for the testing dataset file.

6.17.1 Function Documentation

getRandomNo()

```
void getRandomNo (
            int random[DATA_SIZE])
```

Generates an array of unique random integers within the range of the dataset size.

This function populates an array with unique random integers between 0 and `DATA_SIZE - 1`. It ensures that each integer appears only once by checking a `check` array to track used indices. This can be used for randomizing the order of data for splitting purposes.

Parameters

random	Array to store the generated unique random integers.

See also

DATA_SIZE

Definition at line 143 of file importData.c.

getTestingData()

```
int getTestingData (
            struct Dataset ** d)
```

Retrieves the testing data from a file and returns its length.

This function zeroes out the `data` array for the length of the testing set, reads the dataset from the specified `testingFile`, and assigns the `data` pointer to the provided dataset pointer. Returns the length of the testing data loaded.

Parameters

d	Pointer to a dataset pointer that will reference the loaded testing data array.

Returns

The number of testing entries loaded (i.e., `len_test`).

See also

readDataset, testingFile

Definition at line 175 of file importData.c.

getTrainingData()

```
int getTrainingData (
            struct Dataset ** d)
```

Retrieves the training data from a file and returns its length.

This function initializes the `data` array to zero for the length of the training set, reads the dataset from the specified `trainingFile`, and assigns the `data` pointer to the provided dataset pointer. Returns the length of the training data loaded.

Parameters

d	Pointer to a dataset pointer that will reference the loaded training data array.

Returns

The number of training entries loaded (i.e., `len_train`).

See also

readDataset, trainingFile

Definition at line 167 of file importData.c.

readDataset()

```
int readDataset (
            const char * filename,
            bool split)
```

Reads a dataset from a file and optionally randomizes entries for training and testing.

Opens a file to read each line as a Tic Tac Toe board state, populating a grid structure where 'x', 'o', and 'b' represent the Bot, Player 1, and empty cells, respectively. Each board state is followed by an outcome that is stored within the dataset. If `split` is true, entries are randomized using an array of unique indices for shuffling.

Parameters

filename	The name of the dataset file to read.
split	Boolean indicating whether to randomize entries for dataset splitting.

Returns

int SUCCESS (0) if reading is successful, BAD_PARAM (-5) if the file cannot be opened, or the return value of `splitFile()` if `split` is enabled.

See also

getRandomNo, splitFile

Definition at line 44 of file importData.c.

splitFile()

```
int splitFile ()
```

Splits the dataset into training and testing files with an 80-20 ratio.

This function separates the dataset into two parts: 80% for training and 20% for testing. The training portion is written to `trainingFile`, and the testing portion is written to `testingFile`. Each entry consists of a 3x3 grid representing the Tic Tac Toe board and the outcome of that board.

Returns

int SUCCESS (0) if both files are written successfully, BAD_PARAM (-5) if either file cannot be opened.

See also

data, trainingFile, testingFile

Definition at line 90 of file importData.c.

6.17.2 Variable Documentation

data

```
struct Dataset data[DATA_SIZE]
```

Global array to store the dataset.

This array holds the Tic-Tac-Toe board states and their corresponding outcomes.

Definition at line 38 of file importData.c.

len_test

```
int len_test = 0
```

Global variable to store the number of testing dataset entries.

This variable tracks the size of the testing dataset after splitting.

Definition at line 36 of file importData.c.

len_train

```
int len_train = 0
```

Global variable to store the number of training dataset entries.

This variable tracks the size of the training dataset after splitting.

Definition at line 35 of file importData.c.

randomNo

```
int randomNo[DATA_SIZE]
```

Global array to store unique random indices for dataset splitting.

This array stores randomized indices used to split the dataset into training and testing subsets.

Definition at line 37 of file importData.c.

testingFile

```
const char* testingFile = RES_PATH "" TEST_PATH "" DATA_PATH
```

Global variable to store the path for the testing dataset file.

This variable holds the full path to the testing dataset file for reading and writing.

Definition at line 42 of file importData.c.

trainingFile

```
const char* trainingFile = RES_PATH "" TRAIN_PATH "" DATA_PATH
```

Global variable to store the path for the training dataset file.

This variable holds the full path to the training dataset file for reading and writing.

Definition at line 41 of file importData.c.

6.18 importData.c

Go to the documentation of this file.

```
00001 #include <importData.h>
00002
00035 int len_train = 0;
00036 int len_test = 0;
00037 int randomNo[DATA_SIZE];
00038 struct Dataset data[DATA_SIZE];
00039
00040 // to write to directory before
00041 const char *trainingFile = RES_PATH "" TRAIN_PATH "" DATA_PATH;
00042 const char *testingFile = RES_PATH "" TEST_PATH "" DATA_PATH;
00043
00044 int readDataset(const char *filename, bool split)
00045 {
00046     FILE *file = fopen(filename, "r");
00047     if (!file)
00048     {
00049         PRINT_DEBUG("[ERROR] Error opening file.\n");
00050         return BAD_PARAM;
00051     }
00052
00053     if (split)
00054     {
00055         // get an array of random int where each position is different
00056         getRandomNo(randomNo);
00057     }
00058
00059     char line[100];
00060     for (int i = 0; i < DATA_SIZE && fgets(line, sizeof(line), file); i++)
00061     {
00062         // Get first token with delimiter being ","
00063         char *token = strtok(line, ",");
00064         for (int row = 0; row < 3; row++)
00065         {
00066             for (int col = 0; col < 3; col++)
00067             {
00068                 if (token != NULL)
00069                 {
00070                     data[split ? randomNo[i] : i].grid[row][col] = token[0];
00071                     token = strtok(NULL, ",");
00072                 }
00073             }
00074         }
00075
00076         if (token != NULL)
00077         {
00078             strncpy(data[split ? randomNo[i] : i].outcome, token, sizeof(data[split ? randomNo[i] :
      i].outcome) - 1);
00079         }
00080     }
00081     fclose(file);
00082
00083     if (split)
00084     {
00085         return splitFile();
00086     }
00087     return SUCCESS;
00088 }
00089
00090 int splitFile()
00091 {
00092     // get 80% and 20% respectively
00093     int eighty = len_train = 0.8 * DATA_SIZE;
00094     len_test = 0.2 * DATA_SIZE;
00095
00096     // write into training dataset
00097     FILE *trainFile;
00098     trainFile = fopen(trainingFile, "w");
00099     if (!trainFile)
00100     {
00101         PRINT_DEBUG("[ERROR] Error opening file.\n");
00102         return BAD_PARAM;
00103     }
00104
```

```c
    for (int i = 0; eighty > i; i++)
    {
        for (int row = 0; 3 > row; row++)
        {
            for (int col = 0; 3 > col; col++)
            {
                fprintf(trainFile, "%c,", data[i].grid[row][col]);
            }
        }
        fprintf(trainFile, "%s\n", data[i].outcome);
    }

    fclose(trainFile);

    // write into testing dataset
    FILE *testFile;
    testFile = fopen(testingFile, "w");
    if (!testFile)
    {
        PRINT_DEBUG("[ERROR] Error opening file.\n");
        return BAD_PARAM;
    }

    for (int i = eighty; DATA_SIZE > i; i++)
    {
        for (int row = 0; 3 > row; row++)
        {
            for (int col = 0; 3 > col; col++)
            {
                fprintf(testFile, "%c,", data[i].grid[row][col]);
            }
        }
        fprintf(testFile, "%s\n", data[i].outcome);
    }
    fclose(testFile);
    return SUCCESS;
}

void getRandomNo(int random[DATA_SIZE])
{
    int count = 0;
    srand(time(NULL));

    // initialize all to 0 for proper check
    int check[DATA_SIZE];
    for (int i = 0; DATA_SIZE > i; i++)
    {
        check[i] = 0;
    }

    while (DATA_SIZE > count)
    {
        int randNo = rand() % DATA_SIZE;
        if (check[randNo] == 0)
        {
            check[randNo] = 1;
            random[count] = randNo;
            count++;
        }
    }
}

int getTrainingData(struct Dataset **d)
{
    memset(data, 0, len_train * sizeof(struct Dataset));
    readDataset(trainingFile, false);
    *d = data;
    return len_train;
}

int getTestingData(struct Dataset **d)
{
    memset(data, 0, len_test * sizeof(struct Dataset));
    readDataset(testingFile, false);
    *d = data;
    return len_test;
}
```

6.19 src/main.c File Reference

```c
#include <main.h>
```

Functions

- void clearGrid ()

  Clears the game board and resets the player's turn.
- void updateScoreBtn (gpointer data)

  Updates the score display on the button.
- void on_btnGrid_clicked (GtkWidget ∗widget, gpointer data)

  Callback function for handling button clicks on the game grid.
- void on_btnScore_clicked (GtkWidget ∗widget, gpointer data)

  Handles button click for score.
- void showWin ()

  Clears the winning positions and resets the grid.
- int doBOTmove ()

  Executes the bot's move based on the current game mode.
- int chkPlayerWin ()

  Checks the current game board for a win or tie.
- int main (int argc, char ∗argv[ ])

  Initializes and runs the Tic-Tac-Toe GTK application.

Variables

- int iPlayer1_score = 0

  Global variable to track Player 1's score.
- int iPlayer2_score = 0

  Global variable to track Player 2's or Bot's (Minimax/ML) score.
- int iTie_score = 0

  Global variable to track the number of ties/draws.
- int iGameState = PLAY

  Global variable to track the current game state.
- int iBoard [3][3]

  Global 2D array representing the Tic-Tac-Toe game board.
- int iWinPos [3][3]

  Global 2D array to track winning positions on the board.
- bool isPlayer1Turn = true

  Global flag indicating if it's Player 1's turn.
- bool isMLAvail = true

  Global flag indicating if Machine Learning mode is available. This is set to false if the ML data file is missing, disabling the ML game mode.
- struct PlayerMode playerMode = {"2P", MODE_2P}

  Global structure to track the current game mode.
- GtkWidget ∗ btnGrid [3][3]

  Global 2D array of buttons corresponding to the game grid.

6.19.1 Function Documentation

chkPlayerWin()

```
int chkPlayerWin ()
```

Checks the current game board for a win or tie.
This function checks all possible win conditions:

- Diagonals
- Rows

- Columns

If there is a winning line, it marks the winning positions and returns WIN. If there are no winning conditions and the board is full, it returns TIE. If there are unclicked positions left, it returns PLAY.

Returns

WIN if there is a winner, TIE if the game is a tie, PLAY if the game is still ongoing.

See also

iBoard, iWinPos

Definition at line 234 of file main.c.

clearGrid()

```
void clearGrid ()
```

Clears the game board and resets the player's turn.
This function is used to reset the game board for a new round. It clears the labels on the buttons in the grid and resets the internal board state (iBoard) to 0. It also sets the player turn back to player 1.

- Sets all button labels in the `btnGrid` to an empty string.
- Resets all values in the `iBoard` array to 0, indicating no moves.
- Resets `isPlayer1Turn` to `true`, indicating it's Player 1's turn.

See also

iBoard
btnGrid
isPlayer1Turn

Definition at line 29 of file main.c.

doBOTmove()

```
int doBOTmove ()
```

Executes the bot's move based on the current game mode.
In MM mode:

- Performs a minimax move.
- 20% chance of the minimax randomly selects a position.

In ML mode, the bot uses machine learning to determine the best position.
The function also measures and logs the time taken for the minimax move.

Returns

SUCCESS if the bot's move was made successfully.

See also

playerMode, isMLAvail, iBoard, findBestMove, getBestPosition, btnGrid

Definition at line 181 of file main.c.

main()

```
int main (
            int argc,
            char * argv[ ])
```

Initializes and runs the Tic-Tac-Toe GTK application.
This function initializes GTK, creates the main window, and sets up the game grid, score display, and buttons. It also handles the setup for the game mode and ML availability. The game board and score are displayed, and event listeners are attached to buttons.

Parameters

argc	The number of arguments passed to the program.
argv	The list of arguments passed to the program.

Returns

SUCCESS if the program runs successfully.

See also

initData, on_btnScore_clicked, on_btnGrid_clicked, btnGrid

Definition at line 302 of file main.c.

on_btnGrid_clicked()

```
void on_btnGrid_clicked (
            GtkWidget * widget,
            gpointer data)
```

Callback function for handling button clicks on the game grid.
This function handles the logic for a player's move when a button in the game grid is clicked. It updates the game state, checks for a winner or tie, and updates the score display. It also handles player turns, Bot moves (if applicable), and resets the game board when the game state changes.

Parameters

widget	The GtkWidget that was clicked (the button in the grid).
data	Additional data passed to the callback (usually the score display data).

- If the game state is not `PLAY`, the game will be reset, and the score updated.
- If the clicked button already has a label, the function returns early (no action is taken).
- If the clicked button is empty, the move is recorded in the `iBoard` array (Player 1 or MM or ML).
- After each move, the game checks for a win or tie condition using `chkPlayerWin()`.
- If Player 1 or Player 2 wins, the score is updated, and the win condition is shown.
- If the game ends in a tie, the tie score is updated.
- If the game is in **2P** mode, turns alternate between Player 1 and Player 2.
- In **MM mode**, the Minimax will automatically make a move after Player 1's turn.
- In **ML mode**, the dataset is re-read and initialized after the game ends.

See also

iBoard, isPlayer1Turn, iPlayer1_score, iPlayer2_score, iTie_score
playerMode, updateScoreBtn, chkPlayerWin, doBOTmove, showWin
PLAY, TIE, WIN

Definition at line 60 of file main.c.

on_btnScore_clicked()

```
void on_btnScore_clicked (
            GtkWidget * widget,
            gpointer data)
```

Handles button click for score.
Toggles the player mode and updates the displayed score.

Parameters

widget	The widget that triggered the event.
data	Additional data passed to the callback.

See also

playerMode, isMLAvail, isPlayer1Turn, updateScoreBtn, clearGrid

Definition at line 131 of file main.c.

showWin()

```
void showWin ()
```

Clears the winning positions and resets the grid.
Iterates over the win positions and clears any displayed labels, resetting the grid to its initial state.

See also

iWinPos, btnGrid

Definition at line 159 of file main.c.

updateScoreBtn()

```
void updateScoreBtn (
            gpointer data)
```

Updates the score display on the button.
This function updates the label on a score button to display the current scores for Player 1, Player 2, and Ties. It changes the text formatting depending on which player's turn it is, highlighting the active player.

Parameters

data	A gpointer (usually a button widget) that is used to update the label.

- The function checks if it's Player 1's turn and updates the score display with a bold label for Player 1, or Player 2’s turn with Player 2's score in bold.
- The button text is updated using `gtk_button_set_label()`, and the label markup is updated using `gtk_label_set_markup()`.
- The score includes Player 1's score, Player 2's score, the tie count, and the current game mode (`player↩ Mode.txt`).

See also

iPlayer1_score, iTie_score, iPlayer2_score, playerMode

Definition at line 42 of file main.c.

6.19.2 Variable Documentation

btnGrid

```
GtkWidget* btnGrid[3][3]
```

Global 2D array of buttons corresponding to the game grid.
Definition at line 19 of file main.c.

iBoard

`int iBoard[3][3]`

Global 2D array representing the Tic-Tac-Toe game board.
Definition at line 11 of file main.c.

iGameState

`int iGameState = PLAY`

Global variable to track the current game state.
Game states:

- PLAY: The game is ongoing.
- TIE: The game ended in a tie.
- WIN: A player has won the game.

Definition at line 10 of file main.c.

iPlayer1_score

`int iPlayer1_score = 0`

Global variable to track Player 1's score.
Definition at line 7 of file main.c.

iPlayer2_score

`int iPlayer2_score = 0`

Global variable to track Player 2's or Bot's (Minimax/ML) score.
Definition at line 8 of file main.c.

isMLAvail

`bool isMLAvail = true`

Global flag indicating if Machine Learning mode is available. This is set to false if the ML data file is missing, disabling the ML game mode.
Definition at line 15 of file main.c.

isPlayer1Turn

`bool isPlayer1Turn = true`

Global flag indicating if it's Player 1's turn.
Definition at line 14 of file main.c.

iTie_score

`int iTie_score = 0`

Global variable to track the number of ties/draws.
Definition at line 9 of file main.c.

iWinPos

`int iWinPos[3][3]`

Global 2D array to track winning positions on the board.
Definition at line 12 of file main.c.

playerMode

```
struct PlayerMode playerMode = {"2P", MODE_2P}
```

Global structure to track the current game mode.
Fields:

- txt: Text representation of the current mode (e.g., "2P", "MM", "ML").
- mode: Integer value representing the current game mode.

Player modes:

- MODE_2P: Player vs Player mode.
- MODE_MM: Minimax Bot mode.
- MODE_ML: Machine Learning Bot mode.

Definition at line 17 of file main.c.

6.20 main.c

Go to the documentation of this file.

```
00001 #include <main.h>
00002 
00003 /*==========================================================================================
00004 GLOBAL DECLARATION
00005 ========================================================================================*/
00006 
00007 int iPlayer1_score = 0;
00008 int iPlayer2_score = 0;
00009 int iTie_score = 0;
00010 int iGameState = PLAY;
00011 int iBoard[3][3];
00012 int iWinPos[3][3];
00013 
00014 bool isPlayer1Turn = true;
00015 bool isMLAvail = true;
00016 
00017 struct PlayerMode playerMode = {"2P", MODE_2P};
00018 
00019 GtkWidget *btnGrid[3][3];
00020 
00021 /*==========================================================================================
00022 END OF GLOBAL DECLARATION
00023 ========================================================================================*/
00024 
00025 /*==========================================================================================
00026 GUI FUNCTIONS
00027 ========================================================================================*/
00028 
00029 void clearGrid()
00030 {
00031     isPlayer1Turn = true;
00032     for (int i = 0; i < 3; i++)
00033     {
00034         for (int j = 0; j < 3; j++)
00035         {
00036             gtk_button_set_label(GTK_BUTTON(btnGrid[i][j]), ""); // Clear the button labels
00037             iBoard[i][j] = 0;
00038         }
00039     }
00040 }
00041 
00042 void updateScoreBtn(gpointer data)
00043 {
00044     // Update the score display
00045     char score_text[100];
00046     if (isPlayer1Turn == true)
00047     {
00048         snprintf(score_text, sizeof(score_text), "<b>Player 1 (O): %d</b>   |   TIE: %d   |   Player 2
      (X): %d   |   [%s]   ", iPlayer1_score, iTie_score, iPlayer2_score, playerMode.txt);
00049     }
00050     else
00051     {
00052         snprintf(score_text, sizeof(score_text), "Player 1 (O): %d   |   TIE: %d   |   <b>Player 2
      (X): %d</b>   |   [%s]   ", iPlayer1_score, iTie_score, iPlayer2_score, playerMode.txt);
00053     }
00054     gtk_button_set_label(GTK_BUTTON(data), score_text); // Update the score button label
00055     gtk_label_set_markup(GTK_LABEL(gtk_bin_get_child(GTK_BIN(data))), score_text);
00056 }
```

```c


// Callback function for button clicks
void on_btnGrid_clicked(GtkWidget *widget, gpointer data)
{
    const char *current_label = gtk_button_get_label(GTK_BUTTON(widget));
    BtnPos *btnPos = (BtnPos *)g_object_get_data(G_OBJECT(widget), "button-data");

    if (iGameState != PLAY)
    {
        iGameState = PLAY;
        clearGrid();
        updateScoreBtn(data);
        return;
    }

    if (strcmp(current_label, "") != 0)
    {
        return;
    }

    iBoard[btnPos->pos[0]][btnPos->pos[1]] = isPlayer1Turn ? PLAYER1 : BOT; // O (1), X(2), BOT is the same as player 2

    // Update the button text, for example, with an "O"
    gtk_button_set_label(GTK_BUTTON(widget), isPlayer1Turn ? "O" : "X");

    int retVal = chkPlayerWin();

    if (retVal == PLAY)
    {
        isPlayer1Turn = !isPlayer1Turn;
        updateScoreBtn(data);

        if (playerMode.mode == MODE_2P)
        {
            return;
        }

        doBOTmove();
        retVal = chkPlayerWin();
    }

    if (retVal == WIN)
    {
        showWin();
        PRINT_DEBUG("[DEBUG] GAME RESULT -> %s Win\n", isPlayer1Turn ? "Player 1" : playerMode.mode == MODE_2P ? "Player 2"
: "BOT");
        isPlayer1Turn ? iPlayer1_score++ : iPlayer2_score++;
        iGameState = WIN;
    }

    if (retVal == TIE)
    {
        PRINT_DEBUG("[DEBUG] GAME RESULT -> TIE\n");
        iTie_score++;
        iGameState = TIE;
    }

    if (playerMode.mode != MODE_2P)
    {
        isPlayer1Turn = !isPlayer1Turn;
    }

    if (isMLAvail && playerMode.mode == MODE_ML)
    {
        if (retVal == WIN || retVal == TIE)
        {
            readDataset(RES_PATH "" DATA_PATH, true);
            initData();
        }
    }
    updateScoreBtn(data);
}

void on_btnScore_clicked(GtkWidget *widget, gpointer data)
{
    playerMode.mode = (playerMode.mode > 1 ? MODE_2P : ++playerMode.mode);
    switch (playerMode.mode)
    {
    case MODE_MM:
        strncpy(playerMode.txt, "MM", sizeof(playerMode.txt));
        break;

    case MODE_ML:
        if (isMLAvail)
        {
            strncpy(playerMode.txt, "ML", sizeof(playerMode.txt));
            break;
        }

    default:
        playerMode.mode = MODE_2P;
        strncpy(playerMode.txt, "2P", sizeof(playerMode.txt));
    }
    PRINT_DEBUG("playerMode: %d\n", playerMode.mode);
    isPlayer1Turn = true;
    iPlayer1_score = iPlayer2_score = iTie_score = 0;

    clearGrid();
    updateScoreBtn(data);
}

void showWin()
{
    for (int i = 0; i < 3; i++)
    {
        for (int j = 0; j < 3; j++)
        {
            if (iWinPos[i][j] != WIN)
            {
                gtk_button_set_label(GTK_BUTTON(btnGrid[i][j]), "");
            }
        }
    }
    memset(iWinPos, 0, sizeof(iWinPos));
}
/*===============================================================================================
END OF GUI FUNCTIONS
===============================================================================================*/

/*===============================================================================================
LOGIC FUNCTIONS
===============================================================================================*/

int doBOTmove()
{
    struct Position botMove;
    if (playerMode.mode == MODE_MM)
    {
        startElapseTime();
#if !(MINIMAX_GODMODE)
        if (rand() % 100 < 80)
#endif
        {
            botMove = findBestMove(iBoard);
        }
#if !(MINIMAX_GODMODE)
        else
        {
            startElapseTime();
            int randRow = rand() % 3;
            int randCol = rand() % 3;
            bool bIsDone = false;

            while (!bIsDone)
            {
                if (iBoard[randRow][randCol] == EMPTY)
                {
                    PRINT_DEBUG("Random Move -> R:%d C:%d\n", randRow, randCol);
                    botMove.row = randRow;
                    botMove.col = randCol;
                    bIsDone = !bIsDone;
                }
                else
                {
                    randRow = rand() % 3;
                    randCol = rand() % 3;
                }
            }
            stopElapseTime("Minimax Random Move");
        }
#endif
        stopElapseTime("Minimax Move");
    }
    else // ML mode, sets ML as default if for some reason playermode.mode has expected value.
    {
        if (isMLAvail)
        {
            botMove = getBestPosition(iBoard, ’x’);
        }
    }
```

```c

    iBoard[botMove.row][botMove.col] = BOT;
    gtk_button_set_label(GTK_BUTTON(btnGrid[botMove.row][botMove.col]), "X");
    return SUCCESS;
}

int chkPlayerWin()
{
    // check both dia
    if (iBoard[0][0] == iBoard[1][1] && iBoard[1][1] == iBoard[2][2] && iBoard[0][0] != 0)
    {
        iWinPos[0][0] = iWinPos[1][1] = iWinPos[2][2] = WIN;
        return WIN;
    }

    if (iBoard[0][2] == iBoard[1][1] && iBoard[1][1] == iBoard[2][0] && iBoard[0][2] != 0)
    {
        iWinPos[0][2] = iWinPos[1][1] = iWinPos[2][0] = WIN;
        return WIN;
    }

    // check rows and col
    for (int i = 0; i < 3; i++)
    {
        // Check rows
        if (iBoard[i][0] == iBoard[i][1] && iBoard[i][1] == iBoard[i][2] && iBoard[i][0] != 0)
        {
            iWinPos[i][0] = iWinPos[i][1] = iWinPos[i][2] = WIN;
            return WIN;
        }
        // Check columns
        if (iBoard[0][i] == iBoard[1][i] && iBoard[1][i] == iBoard[2][i] && iBoard[0][i] != 0)
        {
            iWinPos[0][i] = iWinPos[1][i] = iWinPos[2][i] = WIN;
            return WIN;
        }
    }

    // check for unclicked grid, if none left then tie
    for (int i = 0; i < 3; i++)
    {
        for (int j = 0; j < 3; j++)
        {
            if (iBoard[i][j] == 0)
            {
                return PLAY;
            }
        }
    }

    return TIE;
}

/*================================================================================================
END OF LOGIC FUNCTIONS
================================================================================================*/

/*================================================================================================
MAIN
*Init GUI interface and global variable/objects
================================================================================================*/

int main(int argc, char *argv[])
{
    int retVal = SUCCESS;
    srand(time(NULL));

    retVal = initData();
    if (retVal != SUCCESS) // disable ML
    {
        isMLAvail = false;
    }

    GtkWidget *window;
    GtkWidget *grid;
    GtkWidget *score_button;

    // Initialize GTK
    gtk_init(&argc, &argv);

    // Create a new window
    window = gtk_window_new(GTK_WINDOW_TOPLEVEL);
    gtk_window_set_title(GTK_WINDOW(window), "Tic-Tac-Toe");
    gtk_window_set_default_size(GTK_WINDOW(window), 250, 950);
    g_signal_connect(window, "destroy", G_CALLBACK(gtk_main_quit), NULL);

    // Create a box to hold the grid and score button with padding
    GtkWidget *box = gtk_box_new(GTK_ORIENTATION_VERTICAL, 0);
    gtk_container_set_border_width(GTK_CONTAINER(box), 50); // Set padding

    // Create a grid to hold the btnGrid
    grid = gtk_grid_new();
    gtk_box_pack_start(GTK_BOX(box), grid, TRUE, TRUE, 0); // Add grid to the box

    // Set the grid to expand
    gtk_grid_set_row_homogeneous(GTK_GRID(grid), TRUE);
    gtk_grid_set_column_homogeneous(GTK_GRID(grid), TRUE);

    // Set CSS styles
    GtkCssProvider *css_provider = gtk_css_provider_new();
    gtk_css_provider_load_from_data(css_provider,
                                    "window { background-color: black; }\n"
                                    "button { background-color: black; color: white; border: 3px solid white; font-size: 24px; background-image: none; }\n"
                                    "button:pressed { background-color: darkgray; }\n",
                                    -1, NULL);
    gtk_style_context_add_provider_for_screen(gdk_screen_get_default(),
                                              GTK_STYLE_PROVIDER(css_provider),
                                              GTK_STYLE_PROVIDER_PRIORITY_USER);

    // Create a button for the score display
    score_button = gtk_button_new_with_label("");
    gtk_label_set_markup(GTK_LABEL(gtk_bin_get_child(GTK_BIN(score_button))), "<b>Player 1 (O): 0</b> |   TIE: 0   |   Player 2 (X): 0   |  [2P]  ");
    g_signal_connect(score_button, "clicked", G_CALLBACK(on_btnScore_clicked), score_button);
    gtk_grid_attach(GTK_GRID(grid), score_button, 0, 3, 3, 1); // Attach score button below the grid

    // Create the 9 btnGrid and add them to the grid
    for (int i = 0; i < 3; i++)
    {
        for (int j = 0; j < 3; j++)
        {
            btnGrid[i][j] = gtk_button_new_with_label("");

            BtnPos *data = g_new(BtnPos, 1); // Allocate memory for the structure
            data->pos[0] = i;                    // Store row
            data->pos[1] = j;                    // Store column

            // Set the structure as data on the button
            g_object_set_data(G_OBJECT(btnGrid[i][j]), "button-data", data);

            g_signal_connect(btnGrid[i][j], "clicked", G_CALLBACK(on_btnGrid_clicked), score_button); // Pass score_button as data
            gtk_grid_attach(GTK_GRID(grid), btnGrid[i][j], j, i, 1, 1); // Attach btnGrid to the grid
        }
    }

    // Make the btnGrid expand to fill the available space
    for (int i = 0; i < 3; i++)
    {
        gtk_widget_set_vexpand(btnGrid[i][0], TRUE);
        gtk_widget_set_hexpand(btnGrid[i][0], TRUE);
    }
    gtk_widget_set_vexpand(score_button, TRUE);
    gtk_widget_set_hexpand(score_button, TRUE);

    // Add the box to the window
    gtk_container_add(GTK_CONTAINER(window), box);

    // Show everything
    gtk_widget_show_all(window);

    // Start the GTK main loop
    gtk_main();

    return SUCCESS;
}
/*
===============================================================================================
END OF MAIN
===============================================================================================
*/
```

6.21 src/minimax.c File Reference

`#include <minimax.h>`

Functions

- int max (int a, int b)

  Returns the maximum of two integers.
- int min (int a, int b)

  Returns the minimum of two integers.
- struct Position findBestMove (int board[3][3])

  Finds the best move for the bot in the Tic-Tac-Toe game.
- int minimax (int board[3][3], int depth, bool isMax)

  Implements the Minimax algorithm to evaluate the best move for the bot.
- int evaluate (int b[3][3])

  Evaluates the current board state to determine if there is a winner.
- bool isMovesLeft (int board[3][3])

  Checks if there are any moves left on the board.
- int loadBoardStates (struct BoardState boardStates[ ])

  Loads board states and their best moves from a file.
- bool checkAndUpdateBestMove (int board[3][3], struct Position ∗bestMove, struct BoardState boardStates[ ], int count)

  Checks if the current board configuration exists in the lookup table and updates the best move.
- void writeBestMoveToFile (int board[3][3], struct Position bestMove)

  Appends the current board state and the best move to a file.

Variables

- int depthCounter = 0

6.21.1 Function Documentation

checkAndUpdateBestMove()

```
bool checkAndUpdateBestMove (
            int board[3][3],
            struct Position * bestMove,
            struct BoardState boardStates[ ],
            int count)
```

Checks if the current board configuration exists in the lookup table and updates the best move.
This function compares the current board with previously saved board states in the `boardStates` array. If a matching board configuration is found, it updates the provided `bestMove` structure with the best move associated with that board state. The function returns true if a match is found and the move is updated, and false if no match is found in the lookup table.

Parameters

board	The current Tic Tac Toe board to check against the saved states.
bestMove	A pointer to the `Position` structure where the best move will be stored if a match is found.
boardStates	An array of `BoardState` structures containing previously saved board configurations and their best moves.
count	The number of saved board states in the `boardStates` array.

Returns

`true` if a matching board configuration is found and the best move is updated, `false` otherwise.

See also

BoardState, Position

Definition at line 419 of file minimax.c.

evaluate()

```
int evaluate (
              int b[3][3])
```

Evaluates the current board state to determine if there is a winner.

This function checks the Tic-Tac-Toe board for winning conditions, i.e., it checks rows, columns, and diagonals for three consecutive marks (either BOT or PLAYER1). It returns a score based on the result:

- +10 if the BOT wins.
- -10 if PLAYER1 wins.
- 0 if there is no winner yet (no winner in rows, columns, or diagonals).

Parameters

b	A 3x3 array representing the Tic-Tac-Toe board.

Returns

The evaluation score:

- +10 for a BOT win,
- -10 for a PLAYER1 win,
- 0 if there is no winner.

See also

BOT, PLAYER1

Definition at line 220 of file minimax.c.

findBestMove()

```
struct Position findBestMove (
              int board[3][3])
```

Finds the best move for the bot in the Tic-Tac-Toe game.

This function first checks if the best move is already stored in memory by looking through previous board states. If the move is found, it is returned. If no best move is found in memory, it traverses all the empty cells on the board, evaluates the potential moves using the minimax algorithm, and returns the optimal move.

Parameters

board	A 3x3 array representing the current Tic-Tac-Toe board.

Returns

The best move for the bot as a struct Position containing the row and column.

See also

minimax, loadBoardStates, checkAndUpdateBestMove, writeBestMoveToFile

Definition at line 71 of file minimax.c.

isMovesLeft()

```
bool isMovesLeft (
                int board[3][3])
```

Checks if there are any moves left on the board.

This function determines if there are any empty cells left on a 3x3 board. It uses one of the following implementations based on compilation options:

- A standard C implementation when `DISABLE_ASM` is defined.
- Optimized assembly instructions for either AArch64 or x86 platforms.

The choice of implementation depends on the platform and preprocessor directives.

Parameters

board	A 3x3 array representing the board state. Each cell should contain: • `EMPTY` (typically 0) if the cell is empty. • Any non-zero value if the cell is occupied.

Returns

`true` if there are empty cells; otherwise, `false`.

Note

If `DISABLE_ASM` is defined, the function uses pure C logic.

For AArch64 platforms, the function uses assembly instructions for efficient scanning.

For x86 platforms, the function uses assembly instructions for efficient scanning.

Example usage:

```
int board[3][3] = {
    {1, 2, 0},
    {0, 1, 2},
    {2, 1, 0}
};
bool movesLeft = isMovesLeft(board);
// movesLeft will be true as there are empty cells (0s).
```

Definition at line 270 of file minimax.c.

loadBoardStates()

```
int loadBoardStates (
                struct BoardState boardStates[])
```

Loads board states and their best moves from a file.

This function attempts to open a file containing saved board states and the corresponding best move for each state. If the file does not exist, a new file is created. It reads the board configurations and the best move for each board, storing them in the provided `boardStates` array.

Each line in the file represents one board state. The board is stored as a 3x3 grid, where 'x' denotes the BOT's move, 'o' denotes PLAYER1's move, and empty spaces are represented as ' ' (empty). The best move for each board is also saved in the file.

Parameters

boardStates	An array of `BoardState` structures to store the loaded board states.

Returns

The number of boards loaded from the file. If the file does not exist, it returns 0 and creates a new file.

See also

BoardState, FILE_BESTMOV

Definition at line 366 of file minimax.c.

max()

```
int max (
              int a,
              int b)
```

Returns the maximum of two integers.
This function computes the maximum of two integers using either:

- A standard C implementation when `DISABLE_ASM` is defined.
- Optimized assembly instructions for either AArch64 or x86 platforms.

The choice of implementation depends on the platform and preprocessor directives.

Parameters

a	The first integer to compare.
b	The second integer to compare.

Returns

The greater of the two integers.

Note

If `DISABLE_ASM` is defined, the function uses pure C logic.

For AArch64 platforms, the function uses assembly instructions with conditional selection.

For x86 platforms, the function uses assembly instructions with register manipulation and branching.

Example usage:

```
int result = max(10, 20);
// result now holds the value 20.
```

Definition at line 5 of file minimax.c.

min()

```
int min (
              int a,
              int b)
```

Returns the minimum of two integers.
This function computes the minimum of two integers using either:

- A standard C implementation when `DISABLE_ASM` is defined.
- Optimized assembly instructions for either AArch64 or x86 platforms.

The choice of implementation depends on the platform and preprocessor directives.

Parameters

a	The first integer to compare.
b	The second integer to compare.

Returns

The smaller of the two integers.

Note

If DISABLE_ASM is defined, the function uses pure C logic.

For AArch64 platforms, the function uses assembly instructions with conditional selection.

For x86 platforms, the function uses assembly instructions with register manipulation and branching.

Example usage:

```
int result = min(10, 20);
// result now holds the value 10.
```

Definition at line 38 of file minimax.c.

minimax()

```
int minimax (
            int board[3][3],
            int depth,
            bool isMax)
```

Implements the Minimax algorithm to evaluate the best move for the bot.

The function recursively evaluates all possible moves using the Minimax algorithm. It returns the best score for the current player (maximizer or minimizer) based on the game state. The algorithm chooses the optimal move for the bot and evaluates the game state at each depth. The depth is capped if Minimax Godmode is not enabled. If there are no moves left or the game is over, it returns the evaluation score.

Parameters

board	A 3x3 array representing the current Tic-Tac-Toe board.
depth	The current depth in the game tree.
isMax	Boolean flag indicating whether it is the maximizer's turn (bot) or the minimizer's turn (player).

Returns

The best score for the current move based on the evaluation function.

See also

evaluate, isMovesLeft, max, min

Definition at line 136 of file minimax.c.

writeBestMoveToFile()

```
void writeBestMoveToFile (
            int board[3][3],
            struct Position bestMove)
```

Appends the current board state and the best move to a file.

This function writes the current Tic Tac Toe board state to a file, encoding the board as a sequence of characters where 'o' represents Player 1, 'x' represents the Bot, and 'b' represents an empty cell. After writing the board state, it appends the best move (row and column) for the current board to the same file.

Parameters

board	The current Tic Tac Toe board to write to the file.
bestMove	The best move to be made, represented by its row and column indices.

See also

Position, BoardState

Definition at line 436 of file minimax.c.

6.21.2 Variable Documentation

depthCounter

```
int depthCounter = 0
```

Definition at line 3 of file minimax.c.

6.22 minimax.c

Go to the documentation of this file.

```
00001 #include <minimax.h>
00002 
00003 int depthCounter = 0;
00004 
00005 int max(int a, int b)
00006 {
00007 #if (DISABLE_ASM)
00008     return (a > b) ? a : b;
00009 #else
00010     int result;
00011 #ifdef __aarch64__
00012     __asm__(
00013         "mov %w0, %w1;"       // Move 'a' to result
00014         "cmp %w0, %w2;"       // Compare result (a) and b
00015         "csel %w0, %w0, %w2, ge;" // If a >= b, keep a in result; otherwise, move b to result
00016         : "=&r" (result)      // Output
00017         : "r" (a), "r" (b)    // Inputs
00018         : "cc"                // Clobbered flags (condition codes)
00019     );
00020 #else   //x86-64
00021     __asm__(
00022         "movl %1, %%eax;"    // Move 'a' to eax
00023         "movl %2, %%ebx;"    // Move 'b' to ebx
00024         "cmpl %%ebx, %%eax;" // Compare eax and ebx
00025         "jge 1f;"            // If a >= b, jump to label 1
00026         "movl %%ebx, %%eax;" // Otherwise, move ebx to eax
00027         "1:;"
00028         "movl %%eax, %0;" // Move result back to C variable
00029         : "=r"(result)    // Output
00030         : "r"(a), "r"(b)  // Inputs
00031         : "%eax", "%ebx"  // Clobbered registers
00032     );
00033 #endif
00034     return result;
00035 #endif
00036 }
00037 
00038 int min(int a, int b)
00039 {
00040 #if (DISABLE_ASM)
00041     return (a < b) ? a : b;
00042 #else
00043     int result;
00044 #ifdef __aarch64__
00045     __asm__(
00046         "mov %w0, %w1;"       // Move 'a' to result
00047         "cmp %w0, %w2;"       // Compare result (a) and b
00048         "csel %w0, %w0, %w2, le;" // If a <= b, keep a in result; otherwise, move b to result
00049         : "=&r" (result)      // Output
00050         : "r" (a), "r" (b)    // Inputs
00051         : "cc"                // Clobbered flags (condition codes)
00052     );
00053 #else   //x86-64
00054     __asm__(
00055         "movl %1, %%eax;"    // Move 'a' to eax
00056         "movl %2, %%ebx;"    // Move 'b' to ebx
00057         "cmpl %%ebx, %%eax;" // Compare eax and ebx
00058         "jle 1f;"            // If a <= b, jump to label 1
00059         "movl %%ebx, %%eax;" // Otherwise, move ebx to eax
00060         "1:;"
00061         "movl %%eax, %0;" // Move result back to C variable
00062         : "=r"(result)    // Output
00063         : "r"(a), "r"(b)  // Inputs
00064         : "%eax", "%ebx"  // Clobbered registers
00065     );
00066 #endif
00067     return result;
00068 #endif
00069 }
00070 
00071 struct Position findBestMove(int board[3][3])
00072 {
00073     int bestVal = -1000;
00074     struct Position bestMove;
```

```

    struct BoardState boardStates[MAX_BOARDS];

#if !(DISABLE_LOOKUP)
    startElapseTime();
    int boardCount = loadBoardStates(boardStates);
    stopElapseTime("Loading lookup table");
#endif

    bestMove.row = ERROR;
    bestMove.col = ERROR;

    startElapseTime();
#if !(DISABLE_LOOKUP)
    if (checkAndUpdateBestMove(board, &bestMove, boardStates, boardCount))
    {
        stopElapseTime("Find best move in lookup table");
        PRINT_DEBUG("Best move found in memory: Row = %d, Col = %d\n", bestMove.row, bestMove.col);
    }
    else
#endif
    {
        startElapseTime();
        // Traverse all cells, evaluate minimax function for
        // all empty cells. And return the cell with optimal
        // value.
        for (int i = 0; i < 3; i++)
        {
            for (int j = 0; j < 3; j++)
            {
                // Check if cell is empty
                if (board[i][j] == EMPTY)
                {
                    // Make the move
                    board[i][j] = BOT;

                    // compute evaluation function for this
                    // move.
                    int moveVal = minimax(board, 0, false);
                    PRINT_DEBUG("[DEBUG] Depth exited at -> %d\n", depthCounter);
                    // Undo the move
                    board[i][j] = EMPTY;

                    // If the value of the current move is more than the best value, then update best move
                    if (moveVal > bestVal)
                    {
                        bestMove.row = i;
                        bestMove.col = j;
                        bestVal = moveVal;
                    }
                }
            }
        }
        stopElapseTime("Minimax depth search");
        writeBestMoveToFile(board, bestMove);
    }

    depthCounter = 0;
    return bestMove;
}

int minimax(int board[3][3], int depth, bool isMax)
{
#if DEBUG
    depthCounter++;
#endif
    int score = evaluate(board);
    // If Maximizer has won the game return his/her
    // evaluated score
    if (score == 10)
        return score;

    // If Minimizer has won the game return his/her
    // evaluated score
    if (score == -10)
        return score;

    // If there are no more moves and no winner then
    // it is a tie
    if (isMovesLeft(board) == false)
        return 0;

#if !(MINIMAX_GODMODE)
    if (depth > 2)
        return 0;
#endif
```

```

    // If this maximizer's move
    if (isMax)
    {
        int best = -1000;

        // Traverse all cells
        for (int i = 0; i < 3; i++)
        {
            for (int j = 0; j < 3; j++)
            {
                // Check if cell is empty
                if (board[i][j] == EMPTY)
                {
                    // Make the move
                    board[i][j] = BOT;

                    // Call minimax recursively and choose
                    // the maximum value
                    best = max(best, minimax(board, depth + 1, !isMax));

                    // Undo the move
                    board[i][j] = EMPTY;
                }
            }
        }
        return best;
    }

    // If this minimizer's move
    else
    {
        int best = 1000;

        // Traverse all cells
        for (int i = 0; i < 3; i++)
        {
            for (int j = 0; j < 3; j++)
            {
                // Check if cell is empty
                if (board[i][j] == EMPTY)
                {
                    // Make the move
                    board[i][j] = PLAYER1;

                    // Call minimax recursively and choose
                    // the minimum value
                    best = min(best,
                               minimax(board, depth + 1, !isMax));

                    // Undo the move
                    board[i][j] = EMPTY;
                }
            }
        }
        return best;
    }
}

int evaluate(int b[3][3])
{
    // Checking for Rows for X or O victory.
    for (int row = 0; row < 3; row++)
    {
        if (b[row][0] == b[row][1] &&
            b[row][1] == b[row][2])
        {
            if (b[row][0] == BOT)
                return +10;
            else if (b[row][0] == PLAYER1)
                return -10;
        }
    }

    // Checking for Columns for X or O victory.
    for (int col = 0; col < 3; col++)
    {
        if (b[0][col] == b[1][col] &&
            b[1][col] == b[2][col])
        {
            if (b[0][col] == BOT)
                return +10;

            else if (b[0][col] == PLAYER1)
                return -10;
        }
    }

    // Checking for Diagonals for X or O victory.
    if (b[0][0] == b[1][1] && b[1][1] == b[2][2])
    {
        if (b[0][0] == BOT)
            return +10;
        else if (b[0][0] == PLAYER1)
            return -10;
    }

    if (b[0][2] == b[1][1] && b[1][1] == b[2][0])
    {
        if (b[0][2] == BOT)
            return +10;
        else if (b[0][2] == PLAYER1)
            return -10;
    }

    // Else if none of them have won then return 0
    return 0;
}

bool isMovesLeft(int board[3][3])
{
#if (DISABLE_ASM)
    for (int i = 0; i<3; i++)
        for (int j = 0; j<3; j++)
            if (board[i][j] == EMPTY)
                return true;
    return false;

#else
    int result;
#ifdef __aarch64__
__asm__(
        "mov x1, #0;"           // x1 = i = 0
        "outer_loop:;"
        "cmp x1, #3;"           // if i >= 3, go to return_false
        "bge return_false;"

        "mov x2, #0;"           // x2 = j = 0
        "inner_loop:;"
        "cmp x2, #3;"           // if j >= 3, increment i
        "bge increment_i;"

        // Calculate board[i][j]
        "mov x3, x1;"           // Copy i to x3
        "lsl x4, x1, #3;"       // x4 = i * 8
        "add x3, x4, x1, lsl #2;" // x3 = i * 8 + i * 4 = i * 12
        "add x3, %1, x3;"       // x3 = board + (i * 12), points to board[i]
        "ldr w0, [x3, x2, lsl #2];" // Load board[i][j] (each int is 4 bytes)

        "cbz w0, return_true;"   // If board[i][j] == 0, go to return_true

        "add x2, x2, #1;"       // Increment j
        "b inner_loop;"

        "increment_i:;"
        "add x1, x1, #1;"       // Increment i
        "b outer_loop;"

        "return_false:;"
        "mov %w0, #0;"          // Set result to 0 (false)
        "b end;"

        "return_true:;"
        "mov %w0, #1;"          // Set result to 1 (true)

        "end:;"
        : "=r"(result)           // Output operand
        : "r"(board)             // Input operand
        : "x0", "x1", "x2", "x3", "x4" // Clobbered registers
    );
#else   //x86-64
    __asm__(
        "xor %%rbx, %%rbx;" // rbx = i = 0
        "outer_loop:;"
        "cmp $3, %%ebx;" // if i >= 3, return false
        "jge return_false;"

        "xor %%rcx, %%rcx;" // rcx = j = 0
        "inner_loop:;"
        "cmp $3, %%ecx;" // if j >= 3, increment i
        "jge increment_i;"

        // Calculate board[i][j]
        "mov %%rbx, %%rdx;"             // Copy i to rdx
        "imul $12, %%rdx, %%rdx;"       // rdx = i * 12 (calculate row offset)
        "add %1, %%rdx;"                // rdx = board + (i * 12), points to board[i]
        "mov (%%rdx, %%rcx, 4), %%eax;" // Load board[i][j]

        "test %%eax, %%eax;" // Check if board[i][j] == 0
        "jz return_true;"    // If board[i][j] == 0, return true

        "inc %%rcx;" // Increment j
        "jmp inner_loop;"

        "increment_i:;"
        "inc %%rbx;" // Increment i
        "jmp outer_loop;"

        "return_false:;"
        "mov $0, %0;" // Set result to 0 (false)
        "jmp end;"

        "return_true:;"
        "mov $1, %0;" // Set result to 1 (true)

        "end:;"
        : "=r"(result)               // Output operand
        : "r"(board)                 // Input operand
        : "rbx", "rcx", "rdx", "rax" // Clobbered registers
    );
#endif
    return result;
#endif
}

int loadBoardStates(struct BoardState boardStates[])
{
    FILE *file = fopen(FILE_BESTMOV, "r");
    if (file == NULL)
    {
        PRINT_DEBUG("%s <- File does not exist. Creating new file.\n", FILE_BESTMOV);
        FILE *file = fopen(FILE_BESTMOV, "w");
        PRINT_DEBUG("Text file created.\n");
        fclose(file);
        return 0; // No boards loaded
    }
    PRINT_DEBUG("File exist. Checking.\n");
    int count = 0;
    char line[100];
    while (fgets(line, sizeof(line), file) != NULL && count < MAX_BOARDS)
    {
        // Parse the line
        char *token = strtok(line, ",");
        int index = 0;

        // Read the board condition
        while (token != NULL && index < 9)
        {
            if (strcmp(token, "x") == 0)
            {
                boardStates[count].board[index / 3][index % 3] = BOT;
            }
            else if (strcmp(token, "o") == 0)
            {
                boardStates[count].board[index / 3][index % 3] = PLAYER1;
            }
            else
            {
                boardStates[count].board[index / 3][index % 3] = EMPTY;
            }
            token = strtok(NULL, ",");
            index++;
        }

        // Read the best move
        if (token != NULL)
        {
            boardStates[count].bestMove.row = atoi(token);
            token = strtok(NULL, ",");
            boardStates[count].bestMove.col = atoi(token);
        }
        count++;
    }

    fclose(file);
    return count; // Return the number of boards loaded
}

bool checkAndUpdateBestMove(int board[3][3], struct Position *bestMove, struct BoardState
boardStates[], int count)
{
    for (int i = 0; i < count; i++)
    {
        if (memcmp(board, boardStates[i].board, sizeof(boardStates[i].board)) == 0)
        {
            // Board matches, update the best move
            *bestMove = boardStates[i].bestMove;
            PRINT_DEBUG("Found position in lookup table\n");
            PRINT_DEBUG("Best Move = R:%d C:%d\n", bestMove->row, bestMove->col);
            return bestMove; // Board matches, return the best move
        }
    }
    PRINT_DEBUG("Position not found in lookup table\n");
    return false; // No matching board found
}

void writeBestMoveToFile(int board[3][3], struct Position bestMove)
{
    FILE *file = fopen(FILE_BESTMOV, "a"); // Open the file for appending
    if (file == NULL)
    {
        PRINT_DEBUG("Error opening file for writing. -> %s\n", FILE_BESTMOV);
        return;
    }

    // Write the board state to the file
    for (int j = 0; j < 3; j++)
    {
        for (int k = 0; k < 3; k++)
        {
            if (board[j][k] == PLAYER1)
            {
                fprintf(file, "o,");
                PRINT_DEBUG("o,");
            }
            else if (board[j][k] == BOT)
            {
                fprintf(file, "x,");
                PRINT_DEBUG("x,");
            }
            else
            {
                fprintf(file, "b,");
                PRINT_DEBUG("b,");
            }
        }
    }
    // Write the best move to the file
    fprintf(file, "%d,%d\n", bestMove.row, bestMove.col);
    if (fprintf(file, "%d,%d\n", bestMove.row, bestMove.col) < 0)
    {
        PRINT_DEBUG("Error writing best move to file. -> %s\n", FILE_BESTMOV);
    }
    PRINT_DEBUG("\nAttempting to write best move to file: Row = %d, Col = %d\n", bestMove.row,
bestMove.col);
    PRINT_DEBUG("New best move written to file.\n");
    fclose(file);
}
```

6.23 src/ml-naive-bayes.c File Reference

```
#include <ml-naive-bayes.h>
#include <math.h>
```

Functions

- int assignMoveIndex (char move)

  Assigns an index to each move ("x", "o", or "b").
- void calculateProbabilities (int dataset_size)

  Calculates the probabilities for each class and conditional probabilities with Laplace smoothing.
- int predictOutcome (struct Dataset board)

  Predicts the outcome of a given Tic Tac Toe board based on previously calculated probabilities.
- struct Position getBestPosition (int grid[3][3], char player)

  Determines the best position for the bot to make a move based on the highest probability.
- void resetTrainingData ()

  Resets the training data and associated statistics for a fresh training cycle.

- int initData ()

    Initializes the training data and model statistics.
- void assignCMValue (int actual, int predicted)

    Updates the confusion matrix based on actual and predicted outcomes.
- void calcConfusionMatrix ()

    Calculates the confusion matrix and error probability for the testing dataset.
- int getTruthValue (char ∗str1)

    Returns an integer value representing a truth value based on input.
- void calcTrainErrors ()

    Calculates the training errors and the probability of error.
- void debugDataset (struct Dataset ∗data, int len)

    Debug function to display dataset contents.

Variables

- int positive_count = 0

    Counter for the number of positive outcomes in the training dataset.
- int negative_count = 0

    Counter for the number of negative outcomes in the training dataset.
- int cM [4] = {0, 0, 0, 0}

    Confusion matrix for evaluating model performance.
- int positiveMoveCount [3][3][3]

    3D array to count occurrences of each move for positive outcomes.
- int negativeMoveCount [3][3][3]

    3D array to count occurrences of each move for negative outcomes.
- int test_PredictedErrors = 0

    Counter for the number of errors in the testing dataset predictions.
- int train_PredictedErrors = 0

    Counter for the number of errors in the training dataset predictions.
- int predicted

    The predicted outcome for the current dataset (1 for positive, 0 for negative).
- int actual

    The actual outcome for the current dataset (1 for positive, 0 for negative).
- double positiveClassProbability

    Probability of a positive outcome in the dataset.
- double negativeClassProbability

    Probability of a negative outcome in the dataset.
- double probabilityErrors

    Probability of error in the predictions, calculated from the testing dataset.

6.23.1 Function Documentation

assignCMValue()

```
void assignCMValue (
             int actual,
             int predicted)
```

Updates the confusion matrix based on actual and predicted outcomes.

This function updates the confusion matrix counters for true positives, false negatives, false positives, and true negatives. It checks the actual and predicted outcomes and increments the appropriate counter in the confusion matrix.

If either the actual or predicted value is ERROR, an error is logged.

Parameters

actual	The actual outcome value (1 for positive, 0 for negative).
predicted	The predicted outcome value (1 for positive, 0 for negative).

See also

cM, ERROR

Definition at line 340 of file ml-naive-bayes.c.

assignMoveIndex()

```
int assignMoveIndex (
            char move)
```

Assigns an index to each move ("x", "o", or "b").
This function maps the board move characters to their corresponding integer values:

- 'x' is mapped to the BOT.
- 'o' is mapped to PLAYER1.
- 'b' is mapped to EMPTY. If the character does not match any of the valid moves, -1 is returned.

Parameters

move	The character representing the move ('x', 'o', or 'b').

Returns

int The integer corresponding to the move:

- BOT for 'x',
- PLAYER1 for 'o',
- EMPTY for 'b',
- ERROR for invalid input.

See also

BOT, PLAYER1, EMPTY, ERROR

Definition at line 21 of file ml-naive-bayes.c.

calcConfusionMatrix()

```
void calcConfusionMatrix ()
```

Calculates the confusion matrix and error probability for the testing dataset.
This function evaluates the model's performance by calculating the confusion matrix based on actual and predicted outcomes. It iterates through the testing data, compares actual outcomes with predicted ones, and updates the confusion matrix values. The number of prediction errors and the probability of error are also computed.

See also

cM, test_PredictedErrors, probabilityErrors, getTruthValue, predictOutcome

Definition at line 373 of file ml-naive-bayes.c.

calcTrainErrors()

```
void calcTrainErrors ()
```

Calculates the training errors and the probability of error.

This function evaluates the model's performance on the training dataset by comparing predicted outcomes with actual ones. It updates the count of prediction errors and computes the probability of error based on the number of errors and the size of the training dataset.

See also

train_PredictedErrors, probabilityErrors, getTruthValue, predictOutcome

Definition at line 424 of file ml-naive-bayes.c.

calculateProbabilities()

```
void calculateProbabilities (
            int dataset_size)
```

Calculates the probabilities for each class and conditional probabilities with Laplace smoothing.

This function calculates:

- The class probabilities for positive and negative outcomes.
- The conditional probabilities for each move ('x', 'o', 'b') at each position on the board, given the class (positive or negative) with Laplace smoothing applied.

The Laplace smoothing is used to prevent zero probabilities for moves that may not have been observed in the training data. The resulting probabilities are printed for debugging purposes.

Parameters

dataset_size	The total number of samples in the dataset used for probability calculation.

See also

positive_count, negative_count, positiveMoveCount, negativeMoveCount

Definition at line 36 of file ml-naive-bayes.c.

debugDataset()

```
void debugDataset (
            struct Dataset * data,
            int len)
```

Debug function to display dataset contents.

This function prints the details of the provided dataset, including the grid values and the corresponding outcomes. It is primarily used for debugging purposes and is not currently in use within the code.

Parameters

data	Pointer to the dataset to be printed.
len	The length of the dataset (number of entries).

See also

PRINT_DEBUG

Definition at line 451 of file ml-naive-bayes.c.

getBestPosition()

```
struct Position getBestPosition (
            int grid[3][3],
            char player)
```

Determines the best position for the bot to make a move based on the highest probability.

This function evaluates all empty positions on the Tic Tac Toe grid and calculates the probability of the bot winning (either as 'x' or 'o') using the pre-calculated move probabilities from the training data. The bot chooses the position with the highest probability of winning, where the move is either 'x' or 'o' depending on the current player. It returns the best position for the bot to make its move.

Parameters

grid	The current state of the Tic Tac Toe game board.
player	The current player, either 'x' or 'o'.

Returns

A struct `Position` representing the row and column of the best move for the bot. If no valid move is found, it returns an error indicator.

See also

positive_count, negative_count, positiveMoveCount, negativeMoveCount

Definition at line 156 of file ml-naive-bayes.c.

getTruthValue()

```
int getTruthValue (
            char * str1)
```

Returns an integer value representing a truth value based on input.

This function converts a string input into a corresponding integer truth value. The input string is compared against "positive" and "negative" to determine the output:

- Returns `1` for "positive".
- Returns `0` for "negative".
- Returns `-1` for any other input, and logs an error message for invalid cases.

Parameters

str1	A pointer to the input string to be evaluated.

Returns

int The corresponding truth value:

- `1` for "positive".
- `0` for "negative".
- `-1` for invalid inputs.

See also

PRINT_DEBUG

Definition at line 406 of file ml-naive-bayes.c.

initData()

```
int initData ()
```

Initializes the training data and model statistics.

This function resets the training data, then retrieves the training dataset for model training. It processes the dataset to count occurrences of positive and negative outcomes and updates the move counts for each grid position based on the data. Afterward, it calculates training errors and updates the confusion matrix.

If the initial dataset is empty, it attempts to load the data again.

See also

resetTrainingData, getTrainingData, calcTrainErrors, calcConfusionMatrix

Definition at line 273 of file ml-naive-bayes.c.

predictOutcome()

```
int predictOutcome (
             struct Dataset board)
```

Predicts the outcome of a given Tic Tac Toe board based on previously calculated probabilities.

This function calculates the probabilities of a positive (Player 1 wins) or negative (Bot wins) outcome for a given board state by multiplying the conditional probabilities of each move in the grid with the class probabilities. The prediction is made based on which outcome (positive or negative) has the higher probability.

If the calculated probabilities are zero, indicating that the outcome cannot be predicted with the available data, the function returns -1.

Parameters

board	The current Tic Tac Toe board whose outcome needs to be predicted.

Returns

1 if the predicted outcome is positive (Player 1 wins), 0 if negative (Bot wins), and -1 if the outcome cannot be predicted.

See also

positiveClassProbability, negativeClassProbability, positiveMoveCount, negativeMoveCount, assignMoveIndex

Definition at line 88 of file ml-naive-bayes.c.

resetTrainingData()

```
void resetTrainingData ()
```

Resets the training data and associated statistics for a fresh training cycle.

This function resets all relevant variables used in the machine learning model's training process. It clears the outcome counts, resets the move count arrays for each grid position, and reinitializes the confusion matrix. Additionally, it clears the prediction error counters, ensuring that the model starts with a clean state.

See also

positive_count, negative_count, positiveMoveCount, negativeMoveCount, cM, test_PredictedErrors, train_PredictedErrors

Definition at line 247 of file ml-naive-bayes.c.

6.23.2 Variable Documentation

actual

```
int actual
```

The actual outcome for the current dataset (1 for positive, 0 for negative).

Definition at line 15 of file ml-naive-bayes.c.

cM

```
int cM[4] = {0, 0, 0, 0}
```

Confusion matrix for evaluating model performance.
Definition at line 7 of file ml-naive-bayes.c.

negative_count

```
int negative_count = 0
```

Counter for the number of negative outcomes in the training dataset.
Definition at line 5 of file ml-naive-bayes.c.

negativeClassProbability

```
double negativeClassProbability
```

Probability of a negative outcome in the dataset.
Definition at line 18 of file ml-naive-bayes.c.

negativeMoveCount

```
int negativeMoveCount[3][3][3]
```

3D array to count occurrences of each move for negative outcomes.
Definition at line 9 of file ml-naive-bayes.c.

positive_count

```
int positive_count = 0
```

Counter for the number of positive outcomes in the training dataset.
Definition at line 4 of file ml-naive-bayes.c.

positiveClassProbability

```
double positiveClassProbability
```

Probability of a positive outcome in the dataset.
Definition at line 17 of file ml-naive-bayes.c.

positiveMoveCount

```
int positiveMoveCount[3][3][3]
```

3D array to count occurrences of each move for positive outcomes.
Definition at line 8 of file ml-naive-bayes.c.

predicted

```
int predicted
```

The predicted outcome for the current dataset (1 for positive, 0 for negative).
Definition at line 14 of file ml-naive-bayes.c.

probabilityErrors

```
double probabilityErrors
```

Probability of error in the predictions, calculated from the testing dataset.
Definition at line 19 of file ml-naive-bayes.c.

test_PredictedErrors

```
int test_PredictedErrors = 0
```

Counter for the number of errors in the testing dataset predictions.
Definition at line 11 of file ml-naive-bayes.c.

train_PredictedErrors

`int train_PredictedErrors = 0`

Counter for the number of errors in the training dataset predictions.

Definition at line 12 of file ml-naive-bayes.c.

6.24 ml-naive-bayes.c

Go to the documentation of this file.

```
00001 #include <ml-naive-bayes.h>
00002 #include <math.h>
00003 
00004 int positive_count = 0;
00005 int negative_count = 0;
00006 
00007 int cM[4] = {0, 0, 0, 0};
00008 int positiveMoveCount[3][3][3];
00009 int negativeMoveCount[3][3][3];
00010 
00011 int test_PredictedErrors = 0;
00012 int train_PredictedErrors = 0;
00013 
00014 int predicted;
00015 int actual;
00016 
00017 double positiveClassProbability;
00018 double negativeClassProbability;
00019 double probabilityErrors;
00020 
00021 int assignMoveIndex(char move) //converts char to int value for easier calculation
00022 {
00023     switch (move)
00024     {
00025     case 'x':
00026         return BOT;
00027     case 'o':
00028         return PLAYER1;
00029     case 'b':
00030         return EMPTY;
00031     default:
00032         return ERROR;
00033     }
00034 }
00035 
00036 void calculateProbabilities(int dataset_size)
00037 {
00038     // Calculate class probability
00039     positiveClassProbability = (double)positive_count / dataset_size;
00040     negativeClassProbability = (double)negative_count / dataset_size;
00041     PRINT_DEBUG("Positive Class Probability: %lf\n", positiveClassProbability);
00042     PRINT_DEBUG("Negative Class Probability: %lf\n", negativeClassProbability);
00043 
00044     // Calculate conditional probability with laplace smoothing
00045     int laplace_smoothing = 1;
00046     for (int row = 0; row < 3; row++)
00047     {
00048         for (int col = 0; col < 3; col++)
00049         {
00050             for (int moveIndex = 0; moveIndex < 3; moveIndex++)
00051             {
00052                 char move;
00053                 if (moveIndex == 0) //convert
00054                 {
00055                     move = 'x';
00056                 }
00057                 else if (moveIndex == 1)
00058                 {
00059                     move = 'o';
00060                 }
00061                 else
00062                 {
00063                     move = 'b';
00064                 }
00065 
00066                 double positiveProbability = (double)(positiveMoveCount[row][col][moveIndex] +
      laplace_smoothing) / (positive_count + 3 * laplace_smoothing);
00067                 double negativeProbability = (double)(negativeMoveCount[row][col][moveIndex] +
      laplace_smoothing) / (negative_count + 3 * laplace_smoothing);
00068                 if (positive_count == 0)
00069                 {
00070                     PRINT_DEBUG("Probability of %c (positive) at grid(%d,%d): No positive outcomes\n",
      move, row, col);
00071                     PRINT_DEBUG("Probability of %c (negative) at grid(%d,%d): %lf\n", move, row, col,
      negativeProbability);
```

```
                }
                else if (negative_count == 0)
                {
                    PRINT_DEBUG("Probability of %c (positive) at grid(%d,%d): %lf\n", move, row, col, positiveProbability);
                    PRINT_DEBUG("Probability of %c (negative) at grid(%d,%d): No negative outcomes\n", move, row, col);
                }
                else
                {
                    PRINT_DEBUG("Probability of %c (positive) at grid(%d,%d): %lf\n", move, row, col, positiveProbability);
                    PRINT_DEBUG("Probability of %c (negative) at grid(%d,%d): %lf\n", move, row, col, negativeProbability);
                }
            }
        }
    }
}

int predictOutcome(struct Dataset board)
{
    double positiveProbability = positiveClassProbability;
    double negativeProbability = negativeClassProbability;

    // required as 0*anything = 0
    if (positiveProbability == 0)
    {
        positiveProbability = 1;
    }
    if (negativeProbability == 0)
    {
        negativeProbability = 1;
    }

    //loops through board grid and sums up probability for each grid
    for (int row = 0; row < 3; row++)
    {
        for (int col = 0; col < 3; col++)
        {
            int moveIndex = assignMoveIndex(board.grid[row][col]);
            if (moveIndex != -1)
            {
                // PRINT_DEBUG("\nPC_%d, NC_%d, pMC_%d, nMC_%d",positive_count,negative_count,positiveMoveCount[row][col][moveIndex],negativeMoveCount[row][col][moveIndex]);
                if (positive_count > 0)
                {
                    positiveProbability *= (double)positiveMoveCount[row][col][moveIndex] / (double)positive_count;
                }

                if (negative_count > 0)
                {
                    negativeProbability *= (double)negativeMoveCount[row][col][moveIndex] / (double)negative_count;
                }
            }
        }
    }

    // guard cases if either negativeProbability is unset
    if (positiveProbability == 1)
    {
        positiveProbability = 0;
    }
    if (negativeProbability == 1)
    {
        negativeProbability = 0;
    }

    // Output probabilities for debugging
    // PRINT_DEBUG("\nPositive: %lf, Negative: %lf Probability: \n", positiveProbability, negativeProbability);

    //returns a value based on condition
    if (positiveProbability > negativeProbability)
    {
        // PRINT_DEBUG("Predicted Outcome: Positive\n");
        return 1;
    }
    else if (positiveProbability == 0 || negativeProbability == 0)
    {
        // PRINT_DEBUG("Unable to predict outcome based on available data.");
        return -1;
    }
    else
    {
        // PRINT_DEBUG("Predicted Outcome: Negative\n");
        return 0;
    }
}

struct Position getBestPosition(int grid[3][3], char player)
{
    // Determine whether bot is X or O depending on current player
    char bot = (player == ’x’ ? ’o’ : ’x’);
    char bestMove = ’b’;
    int bestRow = -1;
    int bestCol = -1;
    double highestProbability = 0.0;

    int bot_count;
    int(*botMoveCount)[3][3];

    // Use positive or negative count for calculating probability depending on whether bot is X or O
    // Note that for the dataset, negative outcome is for X, meaning the position of O in negative outcomes are good for the bot playing as O
    if (bot == ’x’)
    {
        bot_count = positive_count;
        botMoveCount = positiveMoveCount;
    }
    else
    {
        bot_count = negative_count;
        botMoveCount = negativeMoveCount;
    }

    for (int row = 0; row < 3; row++)
    {
        for (int col = 0; col < 3; col++)
        {
            // If the grid position is empty
            if (grid[row][col] != EMPTY)
            {
                continue;
            }

            // Calculate probability for X or O to determine best move for bot
            for (int moveIndex = 0; moveIndex < 2; moveIndex++)
            {
                double moveProbability;

                if (bot == ’x’)
                {
                    // Calculate probability for move ’x’
                    if (bot_count > 0)
                    {
                        moveProbability = (double)botMoveCount[row][col][0] / bot_count;
                    }
                    else
                    {
                        moveProbability = 0.0;
                    }
                }
                else
                {
                    // Calculate probability for move ’o’
                    if (bot_count > 0)
                    {
                        moveProbability = (double)botMoveCount[row][col][1] / bot_count;
                    }
                    else
                    {
                        moveProbability = 0.0;
                    }
                }

                // Update best move and position for bot if it has higher probability
                if (moveProbability > highestProbability)
                {
                    highestProbability = moveProbability;
                    bestMove = bot;
                    bestRow = row;
                    bestCol = col;
                }
            }
        }
    }

    // Return best position
    if (bestRow != ERROR && bestCol != ERROR)
    {
        grid[bestRow][bestCol] = bestMove;
```

```c
        PRINT_DEBUG("Best move: %c at grid (%d, %d) with probability: %lf\n", bestMove, bestRow, bestCol, highestProbability);
        return (struct Position){bestRow, bestCol};
    }
    else
    {
        PRINT_DEBUG("\nNo valid move found.\n");
        return (struct Position){ERROR, ERROR}; // Indicate no valid move found
    }
}

void resetTrainingData() {
    // Reset outcome counts
    positive_count = 0;
    negative_count = 0;

    // Reset move count arrays for each grid position
    for (int row = 0; row < 3; row++) {
        for (int col = 0; col < 3; col++) {
            for (int moveIndex = 0; moveIndex < 3; moveIndex++) {
                positiveMoveCount[row][col][moveIndex] = 0;
                negativeMoveCount[row][col][moveIndex] = 0;
            }
        }
    }

    // Reset the confusion matrix counters
    cM[0] = 0; // True positive
    cM[1] = 0; // False negative
    cM[2] = 0; // False positive
    cM[3] = 0; // True negative

    // Reset prediction errors
    test_PredictedErrors = 0;
    train_PredictedErrors = 0;
}

int initData()
{
    resetTrainingData();
    int retVal = SUCCESS;

doGetTrainingData:
    static bool doOnce = false;
    struct Dataset *trainingData = NULL;      // Initialize pointer
    int len = getTrainingData(&trainingData); // Pass address of pointer

    if (len <= 0)
    {
        retVal = readDataset(RES_PATH "" DATA_PATH, true);
        if (retVal != SUCCESS)
        {
            return retVal;
        }

        if (doOnce) //prevents potential loopback/deadlock. Edge case tbh.
        {
            return BAD_PARAM;
        }

        doOnce = true;
        goto doGetTrainingData; //loops until training data is set
    }

    //loops through train dataset for ml training
    for (int i = 0; i < len; i++)
    {
        // Get outcome class count for each position
        if (strcmp(trainingData[i].outcome, "positive") == 0)
        {
            positive_count++;
            for (int row = 0; row < 3; row++)
            {
                for (int col = 0; col < 3; col++)
                {
                    int moveIndex = assignMoveIndex(trainingData[i].grid[row][col]);
                    if (moveIndex != -1)
                    {
                        positiveMoveCount[row][col][moveIndex]++;
                    }
                }
            }
        }
        else if (strcmp(trainingData[i].outcome, "negative") == 0)
        {
            negative_count++;
            for (int row = 0; row < 3; row++)
            {
                for (int col = 0; col < 3; col++)
                {
                    int moveIndex = assignMoveIndex(trainingData[i].grid[row][col]);
                    if (moveIndex != -1)
                    {
                        negativeMoveCount[row][col][moveIndex]++;
                    }
                }
            }
        }
    }
    calcTrainErrors();
    calcConfusionMatrix();
    return SUCCESS;
}

void assignCMValue(int actual, int predicted)
{
    // PRINT_DEBUG("\nactual_%i, predicted_%i\n",actual,predicted);

    if (actual == ERROR || predicted == ERROR)
    {
        PRINT_DEBUG("ERROR either value is -1. actual: %d predicted: %d", actual, predicted);
    }

    if (actual == 1)
    {
        if (predicted == 1)
        {
            cM[0] += 1; // True positive
        }
        else
        {
            cM[1] += 1; // False negative
        }
    }
    else
    {
        if (predicted == 1)
        {
            cM[2] += 1; // False positive
        }
        else
        {
            cM[3] += 1; // True negative
        }
    }
}

void calcConfusionMatrix()
{
    //Tests ml on test dataset and stores result in a confusion matrix

    struct Dataset *test = NULL;
    int len = getTestingData(&test);
    // PRINT_DEBUG("Test_Data length: %d\n", len);
    //loops through testing dataset
    if (len > 0)
    { // Ensure len is valid before accessing test
        for (int i = 0; i < len; i++)
        {
            actual = getTruthValue(test[i].outcome); //converts char* to int for comparison
            predicted = predictOutcome(test[i]);

            // checks and updates total errors for test dataset
            if (actual != predicted)
            {
                test_PredictedErrors += 1;
            }

            //sets value based on actual vs predicted
            assignCMValue(actual, predicted);
        }
    }

    double i = TESTING_DATA_SIZE;                       // assign macro to double as you cant cast macros
    probabilityErrors = (1 / i) * test_PredictedErrors; // round to 2dp? not in spec though

    PRINT_DEBUG("For testing dataset: %d errors, %lf probability of error.\n", test_PredictedErrors, probabilityErrors);
    PRINT_DEBUG("TP: %d, FN: %d, FP: %d, TN: %d\n", cM[0], cM[1], cM[2], cM[3]);
}

int getTruthValue(char *str1) //returns an integer value based on input
{
    if (strcmp(str1, "positive") == 0)
    {
        return 1;
    }
    else if (strcmp(str1, "negative") == 0)
    {
        return 0;
    }
    else
    {
        //guard case if inputs are neither "positive" nor "negative"
        PRINT_DEBUG("ERROR: Not truth value: %p", str1);
        return -1;
    }
}

void calcTrainErrors()
{
    struct Dataset *train = NULL;      // Initialize pointer
    int len = getTrainingData(&train); // Pass address of pointer
    // debugDataset(test,len);

    if (len > 0)
    { // Ensure len is valid before accessing test
        for (int i = 0; i < len; i++)
        {
            predicted = predictOutcome(train[i]);
            actual = getTruthValue(train[i].outcome);
            // PRINT_DEBUG("Actual dataset outcome: %s, Dataset outcome: %d, Predicted outcome: %d\n", test[i].outcome, actual, predicted);
            // checks and updates total errors for train dataset
            if (actual != predicted)
            {
                train_PredictedErrors += 1;
            }
        }
    }

    double i = TRAINING_DATA_SIZE;                       // assign macro to double var as macros cant be cast
    probabilityErrors = (1 / i) * train_PredictedErrors; // round to 2dp? not in spec though

    PRINT_DEBUG("\nFor training dataset: %d errors, %lf probability of error.\n", train_PredictedErrors, probabilityErrors);
}

void debugDataset(struct Dataset *data, int len)
{
    PRINT_DEBUG("%d\n", len);
    if (len > 0)
    { // Ensure len is valid before accessing test
        for (int i = 0; i < len; i++)
        {
            PRINT_DEBUG("%d ", i);
            for (int j = 0; j < 3; j++)
            {
                for (int k = 0; k < 3; k++)
                {
                    PRINT_DEBUG("%c,", data->grid[j][k]);
                }
            }
            PRINT_DEBUG("%s\n", data->outcome);
        }
    }
}
```

Index