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 Data Structure Index

2.1 Data Structures

Here are the data structures with brief descriptions:

3 File Index

3.1 File List

Here is a list of all files with brief descriptions:

4 Data Structure Documentation

4.1 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

4.1.1 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.

4.1.2 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

4.2 BtnPos Struct Reference

Stores the position of a button in the game grid.

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

Data Fields

- int pos [2]

4.2.1 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 34 of file main.h.

4.2.2 Field Documentation

pos

```
int pos[2]
```

Array to hold row and column

Definition at line 36 of file main.h.

The documentation for this struct was generated from the following file:

- header/main.h

4.3 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]

4.3.1 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.

4.3.2 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

4.4 PlayerMode Struct Reference

Stores the current game mode and its textual representation.

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

Data Fields

- char txt [2]
- int mode

4.4.1 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 46 of file main.h.

4.4.2 Field Documentation

mode

```
int mode
```

Integer value representing the current game mode

Definition at line 49 of file main.h.

txt

```
char txt[2]
```

Textual representation of the current game mode

Definition at line 48 of file main.h.

The documentation for this struct was generated from the following file:

- header/main.h

4.5 Position Struct Reference

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

`#include <macros.h>`

Data Fields

- int row
- int col

4.5.1 Detailed Description

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

Definition at line 65 of file macros.h.

4.5.2 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

5 File Documentation

5.1 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.

5.1.1 Detailed Description

Provides functionality for measuring elapsed time for profiling purposes.

Author

jacktan-jk

Version

1.0

Date

2024-11-13

Copyright

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.

5.1.2 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 18 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 37 of file elapsedTime.c.

5.2 elapsedTime.h

Go to the documentation of this file.

```
00001
00016 #ifndef ELAPSED_TIME_H
00017 #define ELAPSED_TIME_H
00018
00019 #include <macros.h>
00020 #include <sys/time.h>
00021
00031 void startElapseTime();
00032
00044 void stopElapseTime(char *str);
00045
00046 #endif // ELAPSED_TIME_H
```

5.3 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 file and optionally splits data randomly.
- int splitFile ()

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

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

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

  Retrieves the testing data and its length.

5.3.1 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.

5.3.2 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.

5.3.3 Function Documentation

getRandomNo()

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

Generates an array of unique random numbers within the dataset size.

Parameters

random	Array to store generated random numbers.

Generates an array of unique random numbers within 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 182 of file importData.c.

getTestingData()

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

Retrieves the testing data and its length.

Parameters

out	*d*	Pointer to a dataset pointer that will reference the testing data.

Returns

Number of entries in the testing data.

Retrieves the testing data and 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

out	*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 240 of file importData.c.

getTrainingData()

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

Retrieves the training data and its length.

Parameters

out	*d*	Pointer to a dataset pointer that will reference the training data.

Returns

Number of entries in the training data.

Retrieves the training data and 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

out	*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 219 of file importData.c.

readDataset()

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

Reads a dataset file and optionally splits data randomly.

Parameters

filename	Name of the file to read.
split	Flag indicating whether to split data into training/testing.

Returns

Success or error code.

Reads a dataset file and optionally splits data randomly.

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, ERROR (-1) if the file cannot be opened, or the return value of `splitFile()` if `split` is enabled.

See also

getRandomNo, splitFile

Definition at line 59 of file importData.c.

splitFile()

```
int splitFile ()
```

Splits dataset into 80% training and 20% testing files.

Returns

Success or error code.

Splits dataset into 80% training and 20% testing files.

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 (-1) if either file cannot be opened.

See also

data, trainingFile, testingFile

Definition at line 118 of file importData.c.

5.4 importData.h

Go to the documentation of this file.

```
00001
00011 #ifndef IMPORTDATA_H
00012 #define IMPORTDATA_H
00013
00014 #include <macros.h>
00015
00016 // changed to 100 for testing. make sure to chg back
00017 #define RES_PATH "./resources/"
00018 #define DATA_PATH "tic-tac-toe.data"
00019 #define TRAIN_PATH "training-"
00020 #define TEST_PATH "testing-"
00028 struct Dataset
00029 {
00030     char grid[3][3];
00031     char outcome[9];
00032 };
00033
00041 int readDataset(const char *filename, bool split);
00042
00048 int splitFile();
00049
00055 void getRandomNo(int random[DATA_SIZE]);
00056
00063 int getTrainingData(struct Dataset **d);
00064
00071 int getTestingData(struct Dataset **d);
00072
00073 #endif // IMPORTDATA_H
```

5.5 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(...)

5.5.1 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.

5.5.2 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.

5.6 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
00035 #define MODE_ML 2
00037 // Player identifiers
00038 #define EMPTY 0
00039 #define PLAYER1 1
00040 #define BOT 2
00042 // Data constants
00043 #define DATA_SIZE 958
00044 #define CLASSES 2
00046 // Debugging and configuration options
00047 #define DEBUG 1
00048 #define MINIMAX_GODMODE 0
00049 #define DISABLE_LOOKUP  0
00050 #define DISABLE_ELAPSED 0
00051 #define DISABLE_ASM     0
00053 #if DEBUG
00054 #define PRINT_DEBUG(...) printf(__VA_ARGS__);
00055 #else
00056 #define PRINT_DEBUG(...)
00057 #endif
00058
00065 struct Position
00066 {
00067     int row;
00068     int col;
00069 };
00070
00071 #endif // MACROS_H
```

5.7 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 ()

  Makes the bot perform a move based on the current game mode.
- int chkPlayerWin ()

  Checks if a player has won the game.
- void clearBtn ()

  Clears the game buttons to reset the game grid.
- void updateScoreBtn (gpointer data)

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

  Handles the click event for a button in the game grid.
- void on_btnScore_clicked (GtkWidget ∗widget, gpointer data)

  Handles the click event for the score button.
- void showWin ()

  Displays the winning positions on the game grid.

5.7.1 Detailed Description

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

Author

jacktan-jk

Version

1.0

Date

2024-11-12

Copyright

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.

5.7.2 Function Documentation

chkPlayerWin()

```
int chkPlayerWin ()
```

Checks if a player has won the game.

This function checks all rows, columns, and diagonals for a winning condition. If a player wins, the win positions are marked and `WIN` is returned. If the board is full and no player has won, `TIE` is returned.

Returns

WIN if a player has won, PLAY if the game is ongoing, TIE if it's a draw.

See also

iBoard, iWinPos

Checks if a player has won the game.

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 383 of file main.c.

clearBtn()

```
void clearBtn ()
```

Clears the game buttons to reset the game grid.

This function resets the button labels and grid for a new game.

See also

btnGrid

Clears the game buttons to reset the game grid.

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 91 of file main.c.

doBOTmove()

```
int doBOTmove ()
```

Makes the bot perform a move based on the current game mode.

If in player vs player mode, the bot makes a strategic move, while in ML mode it selects the best move based on available data.

Returns

SUCCESS if the move is made successfully.

See also

iBoard, BOT, findBestMove, getBestPosition

Makes the bot perform a move based on the current game mode.

In MM mode:

- The bot performs a minimax move by default.
- If the minimax move is not chosen, the bot 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 315 of file main.c.

on_btnGrid_clicked()

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

Handles the click event for a button in the game grid.

This function handles user interactions with the game grid and updates the game state based on the player's move.

Parameters

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

See also

iBoard, isPlayer1Turn, updateScoreBtn

Handles the click event for a button in 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 Bot).
- 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 **Bot mode**, the Bot 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 162 of file main.c.

on_btnScore_clicked()

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

Handles the click event for the score button.

This function toggles the game mode and updates the displayed score when the score button is clicked.

Parameters

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

See also

playerMode, isMLAvail, isPlayer1Turn, updateScoreBtn, clearBtn

Handles the click event for the score button.

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, clearBtn

Definition at line 243 of file main.c.

showWin()

```
void showWin ()
```

Displays the winning positions on the game grid.

This function updates the game grid to show the positions of the winning combination, if any, and clears the board for a new game.

See also

iWinPos, iBoard

Displays the winning positions on the game 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 279 of file main.c.

updateScoreBtn()

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

Updates the score display on the score button.

This function updates the score button label based on the current game scores and player mode.

Parameters

data	Additional data passed to the callback.

See also

playerMode, isPlayer1Turn, iPlayer1_score, iPlayer2_score

Updates the score display on the score 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 120 of file main.c.

5.8 main.h

Go to the documentation of this file.

```
00001
00014 #ifndef MAIN_H // Start of include guard
00015 #define MAIN_H
00016 #include <gtk/gtk.h>
00017
00018 #include <macros.h>
00019 #include <minimax.h>
00020 #include <ml-naive-bayes.h>
00021 #include <elapsedTime.h>
00022
00023 /*=============================================================================================
00024 GLOBAL DECLARATION
00025 =============================================================================================*/
00026
00034 typedef struct
00035 {
00036     int pos[2];
00037 } BtnPos;
00038
00046 struct PlayerMode
00047 {
00048     char txt[2];
00049     int mode;
00050 };
00051
```

```
00061 int doBOTmove();
00062
00073 int chkPlayerWin();
00074
00082 void clearBtn();
00083
00093 void updateScoreBtn(gpointer data);
00094
00105 void on_btnGrid_clicked(GtkWidget *widget, gpointer data);
00106
00117 void on_btnScore_clicked(GtkWidget *widget, gpointer data);
00118
00127 void showWin();
00128
00129 #endif // MAIN_H  // End of include guard
```

5.9 header/minimax.h File Reference

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

```
#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)

  Compares two integers and returns the larger of the two.
- int min (int a, int b)

  Compares two integers and returns the smaller of the two.
- struct Position findBestMove (int board[3][3])

  Finds the best move for the bot on the given board.
- int minimax (int board[3][3], int depth, bool isMax)

  Recursively evaluates all possible moves using the Minimax algorithm.
- int evaluate (int b[3][3])

  Evaluates the current state of the Tic-Tac-Toe board.
- bool isMovesLeft (int board[3][3])

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

  Checks and updates the best move for a board state.
- void writeBestMoveToFile (int board[3][3], struct Position bestMove)

  Writes the best move for the current board to a file.
- int loadBoardStates (struct BoardState boardStates[ ])

  Loads the board states from the file.
- void printFileContents ()

  Prints the contents of the best move file.

5.9.1 Detailed Description

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

Author

jacktan-jk

Version

1.0

Date

2024-11-12

Copyright

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.

5.9.2 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.

5.9.3 Function Documentation

checkAndUpdateBestMove()

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

Checks and updates the best move for a board state.

This function checks if the current board state matches any previously stored board states. If a match is found, it updates the best move for that board.

Parameters

board	The current state of the Tic-Tac-Toe board.
bestMove	A pointer to the best move that will be updated.
boardStates	An array of stored board states.
count	The number of board states in memory.

Returns

True if a matching board state is found and the best move is updated, false otherwise.

See also

loadBoardStates

Checks and updates the best move for a board state.

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 591 of file minimax.c.

evaluate()

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

Evaluates the current state of the Tic-Tac-Toe board.

This function checks for a win or draw condition and returns a score based on the result. A win for the bot is +10, a win for the player is -10, and a tie is 0.

Parameters

board	The current state of the Tic-Tac-Toe board.

Returns

A score representing the result of the evaluation: +10 for a bot win, -10 for a player win, and 0 for a tie.

See also

minimax

Evaluates the current state of the Tic-Tac-Toe board.

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 322 of file minimax.c.

findBestMove()

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

Finds the best move for the bot on the given board.

This function uses the Minimax algorithm to evaluate all possible moves and returns the best move for the bot to make based on the current board state.

Parameters

board	The current state of the Tic-Tac-Toe board.

Returns

The best move for the bot.

See also

minimax, evaluate

Finds the best move for the bot on the given board.

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 136 of file minimax.c.

isMovesLeft()

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

Checks if there are any moves left on the Tic-Tac-Toe board.

This function checks if there are any empty cells on the board to make a move.

Parameters

board	The current state of the Tic-Tac-Toe board.

Returns

True if there are moves left (i.e., empty cells), false if the board is full.

See also

minimax

Checks if there are any moves left on the Tic-Tac-Toe 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).
```

Warning

Ensure the platform supports the specified assembly code paths if `DISABLE_ASM` is not defined.

Definition at line 405 of file minimax.c.

loadBoardStates()

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

Loads the board states from the file.

This function loads previously saved board states and their corresponding best moves from the file into memory.

Parameters

boardStates	An array to store the loaded board states.

Returns

The number of board states loaded from the file.

See also

writeBestMoveToFile

Loads the board states from the 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 521 of file minimax.c.

max()

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

Compares two integers and returns the larger of the two.

This function is used by the Minimax algorithm to maximize the score for the maximizer's move (usually the bot).

Parameters

a	The first integer.
b	The second integer.

Returns

The larger of the two values.

Compares two integers and returns the larger of the two.

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.

Warning

Ensure the platform supports the specified assembly code paths if DISABLE_ASM is not defined.

Example usage:

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

Definition at line 30 of file minimax.c.

min()

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

Compares two integers and returns the smaller of the two.

This function is used by the Minimax algorithm to minimize the score for the minimizer's move (usually the player).

Parameters

a	The first integer.
b	The second integer.

Returns

The smaller of the two values.

Compares two integers and returns the smaller of the two.

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.

Warning

Ensure the platform supports the specified assembly code paths if `DISABLE_ASM` is not defined.

Example usage:

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

Definition at line 88 of file minimax.c.

minimax()

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

Recursively evaluates all possible moves using the Minimax algorithm.

This function determines the best score for the current board state based on the depth of the search tree and whether it is the maximizer's or minimizer's turn.

Parameters

board	The current state of the Tic-Tac-Toe board.
depth	The current depth in the Minimax search tree.
isMax	True if it's the maximizer's turn (bot), false if it's the minimizer's turn (player).

Returns

The best score for the current board state.

See also

evaluate

Recursively evaluates all possible moves using the Minimax algorithm.

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 219 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)
```

Writes the best move for the current board to a file.

This function appends the current board state and its corresponding best move to the file for future reference and lookup.

Parameters

board	The current state of the Tic-Tac-Toe board.
bestMove	The best move for the current board state.

See also

checkAndUpdateBestMove

Writes the best move for the current board 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 620 of file minimax.c.

5.10 minimax.h

Go to the documentation of this file.

```
00001
00016 #ifndef MINIMAX_H // Start of include guard
00017 #define MINIMAX_H
00018
00019 #include <macros.h>
00020 #include <elapsedTime.h>
00021
00029 struct BoardState
00030 {
00031     int board[3][3];
00032     struct Position bestMove;
00033 };
00034
00035 #define FILE_BESTMOV "resources/bestmove.txt"
00036 #define MAX_BOARDS 10000
00048 int max(int a, int b);
00049
00060 int min(int a, int b);
00061
00072 struct Position findBestMove(int board[3][3]);
00073
00086 int minimax(int board[3][3], int depth, bool isMax);
00087
00099 int evaluate(int b[3][3]);
00100
00110 bool isMovesLeft(int board[3][3]);
00111
00125 bool checkAndUpdateBestMove(int board[3][3], struct Position *bestMove, struct BoardState
      boardStates[], int count);
00126
00137 void writeBestMoveToFile(int board[3][3], struct Position bestMove);
00138
00149 int loadBoardStates(struct BoardState boardStates[]);
00150
00157 void printFileContents();
00158
00159 #endif // MINIMAX_H // End of include guard
```

5.11 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)

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

  Calculate the class and conditional probabilities using the training dataset.
- void resetTrainingData ()

  Reset all training data and statistics to their initial state.
- int initData ()

  Initialize the training data by reading it from the dataset.
- int predictOutcome (struct Dataset board)

  Predict the outcome (positive/negative) of a given board state.
- void calcTrainErrors ()

  Calculate the training errors by comparing the predicted outcomes with actual outcomes.
- void calcConfusionMatrix ()

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

  Get the best move and position for the bot based on the highest probability.
- int getTruthValue (char ∗str1)

  Get the truth value of the outcome ('positive' or 'negative') from a string.
- void assignCMValue (int actual, int predicted)

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

  Debug function to print the contents of the dataset.

5.11.1 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

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.

5.11.2 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.

5.11.3 Function Documentation

assignCMValue()

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

Assign a value to the confusion matrix based on actual and predicted outcomes.

Parameters

actual	The actual outcome (positive/negative).
predicted	The predicted outcome (positive/negative).

Assign a value to 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 475 of file ml-naive-bayes.c.

assignMoveIndex()

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

Assign an index for each move ("x", "o", or "b").

Parameters

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

Returns

int The index representing the move (BOT, PLAYER1, EMPTY), or -1 if invalid.

Assign an index for 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 76 of file ml-naive-bayes.c.

calcConfusionMatrix()

```
void calcConfusionMatrix ()
```

Calculate the confusion matrix and error probabilities for the testing dataset.

Definition at line 508 of file ml-naive-bayes.c.

calcTrainErrors()

```
void calcTrainErrors ()
```

Calculate the training errors by comparing the predicted outcomes with actual outcomes.

Calculate the training errors by comparing the predicted outcomes with actual outcomes.

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 576 of file ml-naive-bayes.c.

calculateProbabilities()

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

Calculate the class and conditional probabilities using the training dataset.

Parameters

dataset_size	The size of the dataset used to calculate probabilities.

Calculate the class and conditional probabilities using the training dataset.

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 106 of file ml-naive-bayes.c.

debugDataset()

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

Debug function to print the contents of the dataset.

Parameters

data	The dataset to be debugged.
len	The length of the dataset.

Debug function to print the contents of the dataset.

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 615 of file ml-naive-bayes.c.

getBestPosition()

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

Get the best move and position for the bot based on the highest probability.

Parameters

grid	The current Tic-Tac-Toe board grid.
player	The current player ('x' or 'o').

Returns

struct Position The best move position for the bot.

Get the best move and position for the bot 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 258 of file ml-naive-bayes.c.

getTruthValue()

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

Get the truth value of the outcome ('positive' or 'negative') from a string.

Parameters

str1	The outcome string.

Returns

int The truth value (1 for positive, 0 for negative).

Get the truth value of the outcome ('positive' or 'negative') from a string.

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 550 of file ml-naive-bayes.c.

initData()

```
int initData ()
```

Initialize the training data by reading it from the dataset.

Returns

int Success or failure code (SUCCESS or ERROR).

Initialize the training data by reading it from the dataset.

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 395 of file ml-naive-bayes.c.

predictOutcome()

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

Predict the outcome (positive/negative) of a given board state.

Parameters

board	The current Tic-Tac-Toe board.

Returns

int The predicted outcome (1 for positive, 0 for negative, -1 for error).

Predict the outcome (positive/negative) of a given board state.

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 174 of file ml-naive-bayes.c.

resetTrainingData()

```
void resetTrainingData ()
```

Reset all training data and statistics to their initial state.

Reset all training data and statistics to their initial state.

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 358 of file ml-naive-bayes.c.

5.12 ml-naive-bayes.h

Go to the documentation of this file.

```
00001
00018 #ifndef ML_NAIVE_BAYES_H
00019 #define ML_NAIVE_BAYES_H
00020
00021 #include <macros.h>
00022 #include <importData.h>
00023
00024 #define TRAINING_DATA_SIZE 0.8 * DATA_SIZE
00025 #define TESTING_DATA_SIZE 0.2 * DATA_SIZE
00026 #define CLASSES 2
00034 int assignMoveIndex(char move);
00035
00041 void calculateProbabilities(int dataset_size);
00042
00046 void resetTrainingData();
00047
00053 int initData();
00054
00061 int predictOutcome(struct Dataset board);
00062
00066 void calcTrainErrors();
00067
00071 void calcConfusionMatrix();
00072
00080 struct Position getBestPosition(int grid[3][3], char player);
00081
00088 int getTruthValue(char *str1);
00089
00096 void assignCMValue(int actual, int predicted);
00097
00104 void debugDataset(struct Dataset *data, int len);
00105
00106 #endif // ML_NAIVE_BAYES_H
```

5.13 mainpage.md File Reference

5.14 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

5.14.1 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 18 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 37 of file elapsedTime.c.

5.14.2 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.

5.15 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
00018 void startElapseTime()
00019 {
00020 #if !(DISABLE_ELAPSED)
00021     gettimeofday(&gTime, NULL);
00022     gStartTime = gTime.tv_sec + 1.0e-6 * gTime.tv_usec;
00023 #endif
00024 }
00025
00037 void stopElapseTime(char *str)
00038 {
00039 #if !(DISABLE_ELAPSED)
00040     gettimeofday(&gTime, NULL);
00041     gEndTime = gTime.tv_sec + 1.0e-6 * gTime.tv_usec;
00042     PRINT_DEBUG("[ELAPSED] %s -> took %f seconds \n\n", str, (double)(gEndTime - gStartTime));
00043 #endif
00044 }
```

5.16 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.

5.16.1 Function Documentation

getRandomNo()

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

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

Generates an array of unique random numbers within 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 182 of file importData.c.

getTestingData()

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

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

Retrieves the testing data and 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

`out`	*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 240 of file importData.c.

getTrainingData()

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

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

Retrieves the training data and 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

`out`	*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 219 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.

Reads a dataset file and optionally splits data randomly.

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, ERROR (-1) if the file cannot be opened, or the return value of `splitFile()` if `split` is enabled.

See also

getRandomNo, splitFile

Definition at line 59 of file importData.c.

splitFile()

```
int splitFile ()
```

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

Splits dataset into 80% training and 20% testing files.

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 (-1) if either file cannot be opened.

See also

data, trainingFile, testingFile

Definition at line 118 of file importData.c.

5.16.2 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.

5.17 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
00059 int readDataset(const char *filename, bool split)
00060 {
00061     FILE *file = fopen(filename, "r");
00062     if (!file)
00063     {
00064         PRINT_DEBUG("[ERROR] Error opening file.\n");
00065         return ERROR;
00066     }
00067
00068     if (split)
00069     {
00070         // get an array of random int where each position is different
00071         getRandomNo(randomNo);
00072     }
00073
00074     char line[100];
00075     for (int i = 0; i < DATA_SIZE && fgets(line, sizeof(line), file); i++)
00076     {
00077         // Get first token with delimiter being ","
00078         char *token = strtok(line, ",");
00079         for (int row = 0; row < 3; row++)
00080         {
00081             for (int col = 0; col < 3; col++)
00082             {
00083                 if (token != NULL)
00084                 {
00085                     data[split ? randomNo[i] : i].grid[row][col] = token[0];
00086                     token = strtok(NULL, ",");
00087                 }
00088             }
00089         }
00090
00091         if (token != NULL)
00092         {
00093             strncpy(data[split ? randomNo[i] : i].outcome, token, sizeof(data[split ? randomNo[i] :
      i].outcome) - 1);
00094         }
00095     }
00096     fclose(file);
00097
00098     if (split)
00099     {
00100         return splitFile();
00101     }
00102     return SUCCESS;
00103 }
00104
00118 int splitFile()
00119 {
00120     // get 80% and 20% respectively
00121     int eighty = len_train = 0.8 * DATA_SIZE;
00122     len_test = 0.2 * DATA_SIZE;
00123
00124     // write into training dataset
```

```
00125     FILE *trainFile;
00126     trainFile = fopen(trainingFile, "w");
00127     if (!trainFile)
00128     {
00129         PRINT_DEBUG("[ERROR] Error opening file.\n");
00130         return BAD_PARAM;
00131     }
00132 
00133     for (int i = 0; eighty > i; i++)
00134     {
00135         for (int row = 0; 3 > row; row++)
00136         {
00137             for (int col = 0; 3 > col; col++)
00138             {
00139                 fprintf(trainFile, "%c,", data[i].grid[row][col]);
00140             }
00141         }
00142         fprintf(trainFile, "%s\n", data[i].outcome);
00143     }
00144 
00145     fclose(trainFile);
00146 
00147     // write into testing dataset
00148     FILE *testFile;
00149     testFile = fopen(testingFile, "w");
00150     if (!testFile)
00151     {
00152         PRINT_DEBUG("[ERROR] Error opening file.\n");
00153         return BAD_PARAM;
00154     }
00155 
00156     for (int i = eighty; DATA_SIZE > i; i++)
00157     {
00158         for (int row = 0; 3 > row; row++)
00159         {
00160             for (int col = 0; 3 > col; col++)
00161             {
00162                 fprintf(testFile, "%c,", data[i].grid[row][col]);
00163             }
00164         }
00165         fprintf(testFile, "%s\n", data[i].outcome);
00166     }
00167     fclose(testFile);
00168     return SUCCESS;
00169 }
00170 
00182 void getRandomNo(int random[DATA_SIZE])
00183 {
00184     int count = 0;
00185     srand(time(NULL));
00186 
00187     // initialize all to 0 for proper check
00188     int check[DATA_SIZE];
00189     for (int i = 0; DATA_SIZE > i; i++)
00190     {
00191         check[i] = 0;
00192     }
00193 
00194     while (DATA_SIZE > count)
00195     {
00196         int randNo = rand() % DATA_SIZE;
00197         if (check[randNo] == 0)
00198         {
00199             check[randNo] = 1;
00200             random[count] = randNo;
00201             count++;
00202         }
00203     }
00204 }
00205 
00219 int getTrainingData(struct Dataset **d)
00220 {
00221     memset(data, 0, len_train * sizeof(struct Dataset));
00222     readDataset(trainingFile, false);
00223     *d = data;
00224     return len_train;
00225 }
00226 
00240 int getTestingData(struct Dataset **d)
00241 {
00242     memset(data, 0, len_test * sizeof(struct Dataset));
00243     readDataset(testingFile, false);
00244     *d = data;
00245     return len_test;
00246 }
```

5.18 src/main.c File Reference

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

Functions

- void clearBtn ()

  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.

5.18.1 Function Documentation

chkPlayerWin()

```
int chkPlayerWin ()
```

Checks the current game board for a win or tie.

Checks if a player has won the game.

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 383 of file main.c.

clearBtn()

```
void clearBtn ()
```

Clears the game board and resets the player's turn.

Clears the game buttons to reset the game grid.

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 91 of file main.c.

doBOTmove()

```
int doBOTmove ()
```

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

Makes the bot perform a move based on the current game mode.

In MM mode:

- The bot performs a minimax move by default.
- If the minimax move is not chosen, the bot 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 315 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, score_button

Definition at line 451 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.

Handles the click event for a button in 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 Bot).
- 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 **Bot mode**, the Bot 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 162 of file main.c.

on_btnScore_clicked()

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

Handles button click for score.

Handles the click event for the score button.

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, clearBtn

Definition at line 243 of file main.c.

showWin()

```
void showWin ()
```

Clears the winning positions and resets the grid.

Displays the winning positions on the game 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 279 of file main.c.

updateScoreBtn()

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

Updates the score display on the button.

Updates the score display on the score 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 120 of file main.c.

5.18.2 Variable Documentation

btnGrid

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

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

Definition at line 65 of file main.c.

iBoard

```
int iBoard[3][3]
```

Global 2D array representing the Tic-Tac-Toe game board.

Definition at line 57 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 56 of file main.c.

iPlayer1_score

```
int iPlayer1_score = 0
```

Global variable to track Player 1's score.

Definition at line 53 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 54 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 61 of file main.c.

isPlayer1Turn

```
bool isPlayer1Turn = true
```

Global flag indicating if it's Player 1's turn.

Definition at line 60 of file main.c.

iTie_score

```
int iTie_score = 0
```

Global variable to track the number of ties/draws.

Definition at line 55 of file main.c.

iWinPos

```
int iWinPos[3][3]
```

Global 2D array to track winning positions on the board.

Definition at line 58 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 63 of file main.c.

5.19 main.c

Go to the documentation of this file.

```
00001 #include <main.h>
00002
00003 /*============================================================================================
00004 GLOBAL DECLARATION
00005 ============================================================================================*/
00006
00053 int iPlayer1_score = 0;
00054 int iPlayer2_score = 0;
00055 int iTie_score = 0;
00056 int iGameState = PLAY;
00057 int iBoard[3][3];
00058 int iWinPos[3][3];
00059
00060 bool isPlayer1Turn = true;
00061 bool isMLAvail = true;
00062
00063 struct PlayerMode playerMode = {"2P", MODE_2P};
00064
00065 GtkWidget *btnGrid[3][3];
00066
00067 /*============================================================================================
00068 END OF GLOBAL DECLARATION
00069 ============================================================================================*/
00070
00071 /*============================================================================================
00072 GUI FUNCTIONS
00073 ============================================================================================*/
00074
00091 void clearBtn()
00092 {
00093     isPlayer1Turn = true;
00094     for (int i = 0; i < 3; i++)
00095     {
00096         for (int j = 0; j < 3; j++)
00097         {
00098             gtk_button_set_label(GTK_BUTTON(btnGrid[i][j]), ""); // Clear the button labels
00099             iBoard[i][j] = 0;
00100         }
00101     }
00102 }
00103
00120 void updateScoreBtn(gpointer data)
00121 {
00122     // Update the score display
00123     char score_text[100];
00124     if (isPlayer1Turn == true)
00125     {
00126         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);
00127     }
00128     else
00129     {
00130         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);
00131     }
00132     gtk_button_set_label(GTK_BUTTON(data), score_text); // Update the score button label
00133     gtk_label_set_markup(GTK_LABEL(gtk_bin_get_child(GTK_BIN(data))), score_text);
00134 }
00135
00161 // Callback function for button clicks
00162 void on_btnGrid_clicked(GtkWidget *widget, gpointer data)
00163 {
00164     const char *current_label = gtk_button_get_label(GTK_BUTTON(widget));
00165     BtnPos *btnPos = (BtnPos *)g_object_get_data(G_OBJECT(widget), "button-data");
00166
00167     if (iGameState != PLAY)
00168     {
00169         iGameState = PLAY;
00170         clearBtn();
00171         updateScoreBtn(data);
00172         return;
00173     }
00174
00175     if (strcmp(current_label, "") != 0)
00176     {
00177         return;
00178     }
00179
00180     iBoard[btnPos->pos[0]][btnPos->pos[1]] = isPlayer1Turn ? PLAYER1 : BOT; // O (1), X(2), BOT is the
      same as player 2
00181
00182     // Update the button text, for example, with an "O"
00183     gtk_button_set_label(GTK_BUTTON(widget), isPlayer1Turn ? "O" : "X");
```

```
00184
00185     int retVal = chkPlayerWin();
00186
00187     if (retVal == PLAY)
00188     {
00189         isPlayer1Turn = !isPlayer1Turn;
00190         updateScoreBtn(data);
00191
00192         if (playerMode.mode == MODE_2P)
00193         {
00194             return;
00195         }
00196
00197         doBOTmove();
00198         retVal = chkPlayerWin();
00199     }
00200
00201     if (retVal == WIN)
00202     {
00203         showWin();
00204         PRINT_DEBUG("[DEBUG] GAME RESULT -> %s Win\n", isPlayer1Turn ? "Player 1" : playerMode.mode ==
      MODE_2P ? "Player 2"
00205
      : "BOT");
00206         isPlayer1Turn ? iPlayer1_score++ : iPlayer2_score++;
00207         iGameState = WIN;
00208     }
00209
00210     if (retVal == TIE)
00211     {
00212         PRINT_DEBUG("[DEBUG] GAME RESULT -> TIE\n");
00213         iTie_score++;
00214         iGameState = TIE;
00215     }
00216
00217     if (playerMode.mode != MODE_2P)
00218     {
00219         isPlayer1Turn = !isPlayer1Turn;
00220     }
00221
00222     if (isMLAvail && playerMode.mode == MODE_ML)
00223     {
00224         if (retVal == WIN || retVal == TIE)
00225         {
00226             readDataset(RES_PATH "" DATA_PATH, true);
00227             initData();
00228         }
00229     }
00230     updateScoreBtn(data);
00231 }
00232
00243 void on_btnScore_clicked(GtkWidget *widget, gpointer data)
00244 {
00245     playerMode.mode = (playerMode.mode > 1 ? MODE_2P : ++playerMode.mode);
00246     switch (playerMode.mode)
00247     {
00248     case MODE_MM:
00249         strncpy(playerMode.txt, "MM", sizeof(playerMode.txt));
00250         break;
00251
00252     case MODE_ML:
00253         if (isMLAvail)
00254         {
00255             strncpy(playerMode.txt, "ML", sizeof(playerMode.txt));
00256             break;
00257         }
00258
00259     default:
00260         playerMode.mode = MODE_2P;
00261         strncpy(playerMode.txt, "2P", sizeof(playerMode.txt));
00262     }
00263     PRINT_DEBUG("playerMode: %d\n", playerMode.mode);
00264     isPlayer1Turn = true;
00265     iPlayer1_score = iPlayer2_score = iTie_score = 0;
00266
00267     clearBtn();
00268     updateScoreBtn(data);
00269 }
00270
00279 void showWin()
00280 {
00281     for (int i = 0; i < 3; i++)
00282     {
00283         for (int j = 0; j < 3; j++)
00284         {
00285             if (iWinPos[i][j] != WIN)
00286             {
```

```
00287                 gtk_button_set_label(GTK_BUTTON(btnGrid[i][j]), "");
00288             }
00289         }
00290     }
00291     memset(iWinPos, 0, sizeof(iWinPos));
00292 }
00293 /*============================================================================================
00294 END OF GUI FUNCTIONS
00295 ============================================================================================*/
00296
00297 /*============================================================================================
00298 LOGIC FUNCTIONS
00299 ============================================================================================*/
00300
00315 int doBOTmove()
00316 {
00317     struct Position botMove;
00318     if (playerMode.mode == MODE_MM)
00319     {
00320         startElapseTime();
00321 #if !(MINIMAX_GODMODE)
00322         if (rand() % 100 < 80)
00323 #endif
00324         {
00325             botMove = findBestMove(iBoard);
00326         }
00327 #if !(MINIMAX_GODMODE)
00328         else
00329         {
00330             startElapseTime();
00331             int randRow = rand() % 3;
00332             int randCol = rand() % 3;
00333             bool bIsDone = false;
00334
00335             while (!bIsDone)
00336             {
00337                 if (iBoard[randRow][randCol] == EMPTY)
00338                 {
00339                     PRINT_DEBUG("Random Move -> R:%d C:%d\n", randRow, randCol);
00340                     botMove.row = randRow;
00341                     botMove.col = randCol;
00342                     bIsDone = !bIsDone;
00343                 }
00344                 else
00345                 {
00346                     randRow = rand() % 3;
00347                     randCol = rand() % 3;
00348                 }
00349             }
00350             stopElapseTime("Minimax Random Move");
00351         }
00352 #endif
00353         stopElapseTime("Minimax Move");
00354     }
00355     else // ML mode, sets ML as default if for some reason playermode.mode has expected value.
00356     {
00357         if (isMLAvail)
00358         {
00359             botMove = getBestPosition(iBoard, 'x');
00360         }
00361     }
00362
00363     iBoard[botMove.row][botMove.col] = BOT;
00364     gtk_button_set_label(GTK_BUTTON(btnGrid[botMove.row][botMove.col]), "X");
00365     return SUCCESS;
00366 }
00367
00383 int chkPlayerWin()
00384 {
00385     // check both dia
00386     if (iBoard[0][0] == iBoard[1][1] && iBoard[1][1] == iBoard[2][2] && iBoard[0][0] != 0)
00387     {
00388         iWinPos[0][0] = iWinPos[1][1] = iWinPos[2][2] = WIN;
00389         return WIN;
00390     }
00391
00392     if (iBoard[0][2] == iBoard[1][1] && iBoard[1][1] == iBoard[2][0] && iBoard[0][2] != 0)
00393     {
00394         iWinPos[0][2] = iWinPos[1][1] = iWinPos[2][0] = WIN;
00395         return WIN;
00396     }
00397
00398     // check rows and col
00399     for (int i = 0; i < 3; i++)
00400     {
00401         // Check rows
00402         if (iBoard[i][0] == iBoard[i][1] && iBoard[i][1] == iBoard[i][2] && iBoard[i][0] != 0)
```

```
00403         {
00404             iWinPos[i][0] = iWinPos[i][1] = iWinPos[i][2] = WIN;
00405             return WIN;
00406         }
00407         // Check columns
00408         if (iBoard[0][i] == iBoard[1][i] && iBoard[1][i] == iBoard[2][i] && iBoard[0][i] != 0)
00409         {
00410             iWinPos[0][i] = iWinPos[1][i] = iWinPos[2][i] = WIN;
00411             return WIN;
00412         }
00413     }
00414 
00415     // check for unclicked grid, if none left then tie
00416     for (int i = 0; i < 3; i++)
00417     {
00418         for (int j = 0; j < 3; j++)
00419         {
00420             if (iBoard[i][j] == 0)
00421             {
00422                 return PLAY;
00423             }
00424         }
00425     }
00426 
00427     return TIE;
00428 }
00429 
00430 /*============================================================================================
00431 END OF LOGIC FUNCTIONS
00432 ============================================================================================*/
00433 
00434 /*============================================================================================
00435 MAIN
00436 *Init GUI interface and global variable/objects
00437 ============================================================================================*/
00438 
00451 int main(int argc, char *argv[])
00452 {
00453     int retVal = SUCCESS;
00454     srand(time(NULL));
00455 
00456     retVal = initData();
00457     if (retVal != SUCCESS) // disable ML
00458     {
00459         isMLAvail = false;
00460     }
00461 
00462     GtkWidget *window;
00463     GtkWidget *grid;
00464     GtkWidget *score_button;
00465 
00466     // Initialize GTK
00467     gtk_init(&argc, &argv);
00468 
00469     // Create a new window
00470     window = gtk_window_new(GTK_WINDOW_TOPLEVEL);
00471     gtk_window_set_title(GTK_WINDOW(window), "Tic-Tac-Toe");
00472     gtk_window_set_default_size(GTK_WINDOW(window), 250, 950);
00473     g_signal_connect(window, "destroy", G_CALLBACK(gtk_main_quit), NULL);
00474 
00475     // Create a box to hold the grid and score button with padding
00476     GtkWidget *box = gtk_box_new(GTK_ORIENTATION_VERTICAL, 0);
00477     gtk_container_set_border_width(GTK_CONTAINER(box), 50); // Set padding
00478 
00479     // Create a grid to hold the btnGrid
00480     grid = gtk_grid_new();
00481     gtk_box_pack_start(GTK_BOX(box), grid, TRUE, TRUE, 0); // Add grid to the box
00482 
00483     // Set the grid to expand
00484     gtk_grid_set_row_homogeneous(GTK_GRID(grid), TRUE);
00485     gtk_grid_set_column_homogeneous(GTK_GRID(grid), TRUE);
00486 
00487     // Set CSS styles
00488     GtkCssProvider *css_provider = gtk_css_provider_new();
00489     gtk_css_provider_load_from_data(css_provider,
00490                                     "window { background-color: black; }\n"
00491                                     "button { background-color: black; color: white; border: 3px solid
    white; font-size: 24px; background-image: none; }\n"
00492                                     "button:pressed { background-color: darkgray; }\n",
00493                                     -1, NULL);
00494     gtk_style_context_add_provider_for_screen(gdk_screen_get_default(),
00495                                               GTK_STYLE_PROVIDER(css_provider),
00496                                               GTK_STYLE_PROVIDER_PRIORITY_USER);
00497 
00498     // Create a button for the score display
00499     score_button = gtk_button_new_with_label("");
00500     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]  ");
00501     g_signal_connect(score_button, "clicked", G_CALLBACK(on_btnScore_clicked), score_button);
00502     gtk_grid_attach(GTK_GRID(grid), score_button, 0, 3, 3, 1); // Attach score button below the grid
00503
00504     // Create the 9 btnGrid and add them to the grid
00505     for (int i = 0; i < 3; i++)
00506     {
00507         for (int j = 0; j < 3; j++)
00508         {
00509             btnGrid[i][j] = gtk_button_new_with_label("");
00510
00511             BtnPos *data = g_new(BtnPos, 1); // Allocate memory for the structure
00512             data->pos[0] = i;                   // Store row
00513             data->pos[1] = j;                   // Store column
00514
00515             // Set the structure as data on the button
00516             g_object_set_data(G_OBJECT(btnGrid[i][j]), "button-data", data);
00517
00518             g_signal_connect(btnGrid[i][j], "clicked", G_CALLBACK(on_btnGrid_clicked), score_button);
      // Pass score_button as data
00519             gtk_grid_attach(GTK_GRID(grid), btnGrid[i][j], j, i, 1, 1);
      // Attach btnGrid to the grid
00520         }
00521     }
00522
00523     // Make the btnGrid expand to fill the available space
00524     for (int i = 0; i < 3; i++)
00525     {
00526         gtk_widget_set_vexpand(btnGrid[i][0], TRUE);
00527         gtk_widget_set_hexpand(btnGrid[i][0], TRUE);
00528     }
00529     gtk_widget_set_vexpand(score_button, TRUE);
00530     gtk_widget_set_hexpand(score_button, TRUE);
00531
00532     // Add the box to the window
00533     gtk_container_add(GTK_CONTAINER(window), box);
00534
00535     // Show everything
00536     gtk_widget_show_all(window);
00537
00538     // Start the GTK main loop
00539     gtk_main();
00540
00541     return SUCCESS;
00542 }
00543 /*
00544 ============================================================================================
00545 END OF MAIN
00546 ============================================================================================
00547 */
```

5.20 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

5.20.1 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.

Checks and updates the best move for a board state.

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 591 of file minimax.c.

evaluate()

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

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

Evaluates the current state of the Tic-Tac-Toe board.

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 322 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.

Finds the best move for the bot on the given board.

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 136 of file minimax.c.

isMovesLeft()

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

Checks if there are any moves left on the board.

Checks if there are any moves left on the Tic-Tac-Toe 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).
```

Warning

Ensure the platform supports the specified assembly code paths if `DISABLE_ASM` is not defined.

Definition at line 405 of file minimax.c.

loadBoardStates()

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

Loads board states and their best moves from a file.

Loads the board states from the 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 521 of file minimax.c.

max()

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

Returns the maximum of two integers.

Compares two integers and returns the larger of the two.

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.

Warning

Ensure the platform supports the specified assembly code paths if `DISABLE_ASM` is not defined.

Example usage:

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

Definition at line 30 of file minimax.c.

min()

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

Returns the minimum of two integers.

Compares two integers and returns the smaller of the two.

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.

Warning

Ensure the platform supports the specified assembly code paths if `DISABLE_ASM` is not defined.

Example usage:

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

Definition at line 88 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.

Recursively evaluates all possible moves using the Minimax algorithm.

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 219 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.

Writes the best move for the current board 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 620 of file minimax.c.

5.20.2 Variable Documentation

depthCounter

```
int depthCounter = 0
```

Definition at line 3 of file minimax.c.

5.21 minimax.c

Go to the documentation of this file.

```
00001 #include <minimax.h>
00002
00003 int depthCounter = 0;
00004
00030 int max(int a, int b)
00031 {
00032 #if (DISABLE_ASM)
00033     return (a > b) ? a : b;
00034 #else
00035     int result;
00036 #ifdef __aarch64__
00037     __asm__(
00038         "mov %w0, %w1;"       // Move 'a' to result
00039         "cmp %w0, %w2;"       // Compare result (a) and b
00040         "csel %w0, %w0, %w2, ge;" // If a >= b, keep a in result; otherwise, move b to result
00041         : "=&r" (result)      // Output
00042         : "r" (a), "r" (b)    // Inputs
00043         : "cc"                // Clobbered flags (condition codes)
00044     );
00045 #else   //x86-64
00046     __asm__(
00047         "movl %1, %%eax;"     // Move 'a' to eax
00048         "movl %2, %%ebx;"     // Move 'b' to ebx
00049         "cmpl %%ebx, %%eax;" // Compare eax and ebx
00050         "jge 1f;"             // If a >= b, jump to label 1
00051         "movl %%ebx, %%eax;" // Otherwise, move ebx to eax
00052         "1:;"
00053         "movl %%eax, %0;" // Move result back to C variable
00054         : "=r"(result)    // Output
00055         : "r"(a), "r"(b)  // Inputs
00056         : "%eax", "%ebx"  // Clobbered registers
00057     );
00058 #endif
00059     return result;
00060 #endif
00061 }
00062
00088 int min(int a, int b)
00089 {
00090 #if (DISABLE_ASM)
00091     return (a < b) ? a : b;
00092 #else
00093     int result;
00094 #ifdef __aarch64__
00095     __asm__(
00096         "mov %w0, %w1;"       // Move 'a' to result
00097         "cmp %w0, %w2;"       // Compare result (a) and b
00098         "csel %w0, %w0, %w2, le;" // If a <= b, keep a in result; otherwise, move b to result
00099         : "=&r" (result)      // Output
00100         : "r" (a), "r" (b)    // Inputs
00101         : "cc"                // Clobbered flags (condition codes)
00102     );
00103 #else   //x86-64
00104     __asm__(
00105         "movl %1, %%eax;"     // Move 'a' to eax
00106         "movl %2, %%ebx;"     // Move 'b' to ebx
00107         "cmpl %%ebx, %%eax;" // Compare eax and ebx
00108         "jle 1f;"             // If a <= b, jump to label 1
00109         "movl %%ebx, %%eax;" // Otherwise, move ebx to eax
00110         "1:;"
00111         "movl %%eax, %0;" // Move result back to C variable
00112         : "=r"(result)    // Output
00113         : "r"(a), "r"(b)  // Inputs
00114         : "%eax", "%ebx"  // Clobbered registers
00115     );
00116 #endif
00117     return result;
00118 #endif
00119 }
00120
00136 struct Position findBestMove(int board[3][3])
00137 {
00138     int bestVal = -1000;
00139     struct Position bestMove;
00140
00141     struct BoardState boardStates[MAX_BOARDS];
00142
00143 #if !(DISABLE_LOOKUP)
00144     startElapseTime();
00145     int boardCount = loadBoardStates(boardStates);
00146     stopElapseTime("Loading lookup table");
00147 #endif
00148
```

```
00149     bestMove.row = ERROR;
00150     bestMove.col = ERROR;
00151
00152     startElapseTime();
00153 #if !(DISABLE_LOOKUP)
00154     if (checkAndUpdateBestMove(board, &bestMove, boardStates, boardCount))
00155     {
00156         stopElapseTime("Find best move in lookup table");
00157         PRINT_DEBUG("Best move found in memory: Row = %d, Col = %d\n", bestMove.row, bestMove.col);
00158     }
00159     else
00160 #endif
00161     {
00162         startElapseTime();
00163         // Traverse all cells, evaluate minimax function for
00164         // all empty cells. And return the cell with optimal
00165         // value.
00166         for (int i = 0; i < 3; i++)
00167         {
00168             for (int j = 0; j < 3; j++)
00169             {
00170                 // Check if cell is empty
00171                 if (board[i][j] == EMPTY)
00172                 {
00173                     // Make the move
00174                     board[i][j] = BOT;
00175
00176                     // compute evaluation function for this
00177                     // move.
00178                     int moveVal = minimax(board, 0, false);
00179                     PRINT_DEBUG("[DEBUG] Depth exited at -> %d\n", depthCounter);
00180                     // Undo the move
00181                     board[i][j] = EMPTY;
00182
00183                     // If the value of the current move is more than the best value, then update best
    move
00184                     if (moveVal > bestVal)
00185                     {
00186                         bestMove.row = i;
00187                         bestMove.col = j;
00188                         bestVal = moveVal;
00189                     }
00190                 }
00191             }
00192         }
00193         stopElapseTime("Minimax depth search");
00194         writeBestMoveToFile(board, bestMove);
00195     }
00196
00197     depthCounter = 0;
00198     return bestMove;
00199 }
00200
00219 int minimax(int board[3][3], int depth, bool isMax)
00220 {
00221 #if DEBUG
00222     depthCounter++;
00223 #endif
00224     int score = evaluate(board);
00225     // If Maximizer has won the game return his/her
00226     // evaluated score
00227     if (score == 10)
00228         return score;
00229
00230     // If Minimizer has won the game return his/her
00231     // evaluated score
00232     if (score == -10)
00233         return score;
00234
00235     // If there are no more moves and no winner then
00236     // it is a tie
00237     if (isMovesLeft(board) == false)
00238         return 0;
00239
00240 #if !(MINIMAX_GODMODE)
00241     if (depth > 2)
00242         return 0;
00243 #endif
00244
00245     // If this maximizer's move
00246     if (isMax)
00247     {
00248         int best = -1000;
00249
00250         // Traverse all cells
00251         for (int i = 0; i < 3; i++)
00252         {
```

```
00253             for (int j = 0; j < 3; j++)
00254             {
00255                 // Check if cell is empty
00256                 if (board[i][j] == EMPTY)
00257                 {
00258                     // Make the move
00259                     board[i][j] = BOT;
00260 
00261                     // Call minimax recursively and choose
00262                     // the maximum value
00263                     best = max(best, minimax(board, depth + 1, !isMax));
00264 
00265                     // Undo the move
00266                     board[i][j] = EMPTY;
00267                 }
00268             }
00269         }
00270         return best;
00271     }
00272 
00273     // If this minimizer's move
00274     else
00275     {
00276         int best = 1000;
00277 
00278         // Traverse all cells
00279         for (int i = 0; i < 3; i++)
00280         {
00281             for (int j = 0; j < 3; j++)
00282             {
00283                 // Check if cell is empty
00284                 if (board[i][j] == EMPTY)
00285                 {
00286                     // Make the move
00287                     board[i][j] = PLAYER1;
00288 
00289                     // Call minimax recursively and choose
00290                     // the minimum value
00291                     best = min(best,
00292                                minimax(board, depth + 1, !isMax));
00293 
00294                     // Undo the move
00295                     board[i][j] = EMPTY;
00296                 }
00297             }
00298         }
00299         return best;
00300     }
00301 }
00302 
00322 int evaluate(int b[3][3])
00323 {
00324     // Checking for Rows for X or O victory.
00325     for (int row = 0; row < 3; row++)
00326     {
00327         if (b[row][0] == b[row][1] &&
00328             b[row][1] == b[row][2])
00329         {
00330             if (b[row][0] == BOT)
00331                 return +10;
00332             else if (b[row][0] == PLAYER1)
00333                 return -10;
00334         }
00335     }
00336 
00337     // Checking for Columns for X or O victory.
00338     for (int col = 0; col < 3; col++)
00339     {
00340         if (b[0][col] == b[1][col] &&
00341             b[1][col] == b[2][col])
00342         {
00343             if (b[0][col] == BOT)
00344                 return +10;
00345 
00346             else if (b[0][col] == PLAYER1)
00347                 return -10;
00348         }
00349     }
00350 
00351     // Checking for Diagonals for X or O victory.
00352     if (b[0][0] == b[1][1] && b[1][1] == b[2][2])
00353     {
00354         if (b[0][0] == BOT)
00355             return +10;
00356         else if (b[0][0] == PLAYER1)
00357             return -10;
00358     }
```

```
00359
00360     if (b[0][2] == b[1][1] && b[1][1] == b[2][0])
00361     {
00362         if (b[0][2] == BOT)
00363             return +10;
00364         else if (b[0][2] == PLAYER1)
00365             return -10;
00366     }
00367
00368     // Else if none of them have won then return 0
00369     return 0;
00370 }
00371
00372
00405 bool isMovesLeft(int board[3][3])
00406 {
00407 #if (DISABLE_ASM)
00408     for (int i = 0; i<3; i++)
00409         for (int j = 0; j<3; j++)
00410             if (board[i][j] == EMPTY)
00411                 return true;
00412     return false;
00413
00414 #else
00415     int result;
00416 #ifdef __aarch64__
00417 __asm__(
00418         "mov x1, #0;"           // x1 = i = 0
00419         "outer_loop:;"
00420         "cmp x1, #3;"           // if i >= 3, go to return_false
00421         "bge return_false;"
00422
00423         "mov x2, #0;"           // x2 = j = 0
00424         "inner_loop:;"
00425         "cmp x2, #3;"           // if j >= 3, increment i
00426         "bge increment_i;"
00427
00428         // Calculate board[i][j]
00429         "mov x3, x1;"           // Copy i to x3
00430         "lsl x4, x1, #3;"       // x4 = i * 8
00431         "add x3, x4, x1, lsl #2;" // x3 = i * 8 + i * 4 = i * 12
00432         "add x3, %1, x3;"       // x3 = board + (i * 12), points to board[i]
00433         "ldr w0, [x3, x2, lsl #2];" // Load board[i][j] (each int is 4 bytes)
00434
00435         "cbz w0, return_true;"   // If board[i][j] == 0, go to return_true
00436
00437         "add x2, x2, #1;"       // Increment j
00438         "b inner_loop;"
00439
00440         "increment_i:;"
00441         "add x1, x1, #1;"       // Increment i
00442         "b outer_loop;"
00443
00444         "return_false:;"
00445         "mov %w0, #0;"          // Set result to 0 (false)
00446         "b end;"
00447
00448         "return_true:;"
00449         "mov %w0, #1;"          // Set result to 1 (true)
00450
00451         "end:;"
00452         : "=r"(result)           // Output operand
00453         : "r"(board)             // Input operand
00454         : "x0", "x1", "x2", "x3", "x4" // Clobbered registers
00455     );
00456 #else   //x86-64
00457     __asm__(
00458         "xor %%rbx, %%rbx;" // rbx = i = 0
00459         "outer_loop:;"
00460         "cmp $3, %%ebx;" // if i >= 3, return false
00461         "jge return_false;"
00462
00463         "xor %%rcx, %%rcx;" // rcx = j = 0
00464         "inner_loop:;"
00465         "cmp $3, %%ecx;" // if j >= 3, increment i
00466         "jge increment_i;"
00467
00468         // Calculate board[i][j]
00469         "mov %%rbx, %%rdx;"             // Copy i to rdx
00470         "imul $12, %%rdx, %%rdx;"       // rdx = i * 12 (calculate row offset)
00471         "add %1, %%rdx;"                // rdx = board + (i * 12), points to board[i]
00472         "mov (%%rdx, %%rcx, 4), %%eax;" // Load board[i][j]
00473
00474         "test %%eax, %%eax;" // Check if board[i][j] == 0
00475         "jz return_true;"    // If board[i][j] == 0, return true
00476
00477         "inc %%rcx;" // Increment j
```

```
00478         "jmp inner_loop;"
00479
00480         "increment_i:;"
00481         "inc %%rbx;" // Increment i
00482         "jmp outer_loop;"
00483
00484         "return_false:;"
00485         "mov $0, %0;" // Set result to 0 (false)
00486         "jmp end;"
00487
00488         "return_true:;"
00489         "mov $1, %0;" // Set result to 1 (true)
00490
00491         "end:;"
00492         : "=r"(result)               // Output operand
00493         : "r"(board)                 // Input operand
00494         : "rbx", "rcx", "rdx", "rax" // Clobbered registers
00495     );
00496 #endif
00497     return result;
00498 #endif
00499 }
00500
00521 int loadBoardStates(struct BoardState boardStates[])
00522 {
00523     FILE *file = fopen(FILE_BESTMOV, "r");
00524     if (file == NULL)
00525     {
00526         PRINT_DEBUG("%s <- File does not exist. Creating new file.\n", FILE_BESTMOV);
00527         FILE *file = fopen(FILE_BESTMOV, "w");
00528         PRINT_DEBUG("Text file created.\n");
00529         fclose(file);
00530         return 0; // No boards loaded
00531     }
00532     PRINT_DEBUG("File exist. Checking.\n");
00533     int count = 0;
00534     char line[100];
00535     while (fgets(line, sizeof(line), file) != NULL && count < MAX_BOARDS)
00536     {
00537         // Parse the line
00538         char *token = strtok(line, ",");
00539         int index = 0;
00540
00541         // Read the board condition
00542         while (token != NULL && index < 9)
00543         {
00544             if (strcmp(token, "x") == 0)
00545             {
00546                 boardStates[count].board[index / 3][index % 3] = BOT;
00547             }
00548             else if (strcmp(token, "o") == 0)
00549             {
00550                 boardStates[count].board[index / 3][index % 3] = PLAYER1;
00551             }
00552             else
00553             {
00554                 boardStates[count].board[index / 3][index % 3] = EMPTY;
00555             }
00556             token = strtok(NULL, ",");
00557             index++;
00558         }
00559
00560         // Read the best move
00561         if (token != NULL)
00562         {
00563             boardStates[count].bestMove.row = atoi(token);
00564             token = strtok(NULL, ",");
00565             boardStates[count].bestMove.col = atoi(token);
00566         }
00567         count++;
00568     }
00569
00570     fclose(file);
00571     return count; // Return the number of boards loaded
00572 }
00573
00591 bool checkAndUpdateBestMove(int board[3][3], struct Position *bestMove, struct BoardState
    boardStates[], int count)
00592 {
00593     for (int i = 0; i < count; i++)
00594     {
00595         if (memcmp(board, boardStates[i].board, sizeof(boardStates[i].board)) == 0)
00596         {
00597             // Board matches, update the best move
00598             *bestMove = boardStates[i].bestMove;
00599             PRINT_DEBUG("Found position in lookup table\n");
00600             PRINT_DEBUG("Best Move = R:%d C:%d\n", bestMove->row, bestMove->col);
```

```
00601             return bestMove; // Board matches, return the best move
00602         }
00603     }
00604     PRINT_DEBUG("Position not found in lookup table\n");
00605     return false; // No matching board found
00606 }
00607
00620 void writeBestMoveToFile(int board[3][3], struct Position bestMove)
00621 {
00622     FILE *file = fopen(FILE_BESTMOV, "a"); // Open the file for appending
00623     if (file == NULL)
00624     {
00625         PRINT_DEBUG("Error opening file for writing. -> %s\n", FILE_BESTMOV);
00626         return;
00627     }
00628
00629     // Write the board state to the file
00630     for (int j = 0; j < 3; j++)
00631     {
00632         for (int k = 0; k < 3; k++)
00633         {
00634             if (board[j][k] == PLAYER1)
00635             {
00636                 fprintf(file, "o,");
00637                 PRINT_DEBUG("o,");
00638             }
00639             else if (board[j][k] == BOT)
00640             {
00641                 fprintf(file, "x,");
00642                 PRINT_DEBUG("x,");
00643             }
00644             else
00645             {
00646                 fprintf(file, "b,");
00647                 PRINT_DEBUG("b,");
00648             }
00649         }
00650     }
00651     // Write the best move to the file
00652     fprintf(file, "%d,%d\n", bestMove.row, bestMove.col);
00653     if (fprintf(file, "%d,%d\n", bestMove.row, bestMove.col) < 0)
00654     {
00655         PRINT_DEBUG("Error writing best move to file. -> %s\n", FILE_BESTMOV);
00656     }
00657     PRINT_DEBUG("\nAttempting to write best move to file: Row = %d, Col = %d\n", bestMove.row,
      bestMove.col);
00658     PRINT_DEBUG("New best move written to file.\n");
00659     fclose(file);
00660 }
```

5.22 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 ()

  Calculate the confusion matrix and error probabilities for the testing dataset.
- int getTruthValue (char *str1)

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

5.22.1 Function Documentation

assignCMValue()

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

Updates the confusion matrix based on actual and predicted outcomes.

Assign a value to 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 475 of file ml-naive-bayes.c.

assignMoveIndex()

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

Assigns an index to each move ("x", "o", or "b").

Assign an index for 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 76 of file ml-naive-bayes.c.

calcConfusionMatrix()

```
void calcConfusionMatrix ()
```

Calculate the confusion matrix and error probabilities for the testing dataset.

Definition at line 508 of file ml-naive-bayes.c.

calcTrainErrors()

```
void calcTrainErrors ()
```

Calculates the training errors and the probability of error.

Calculate the training errors by comparing the predicted outcomes with actual outcomes.

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 576 of file ml-naive-bayes.c.

calculateProbabilities()

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

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

Calculate the class and conditional probabilities using the training dataset.

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 106 of file ml-naive-bayes.c.

debugDataset()

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

Debug function to display dataset contents.

Debug function to print the contents of the dataset.

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 615 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.

Get the best move and position for the bot 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 258 of file ml-naive-bayes.c.

getTruthValue()

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

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

Get the truth value of the outcome ('positive' or 'negative') from a string.

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 550 of file ml-naive-bayes.c.

initData()

```
int initData ()
```

Initializes the training data and model statistics.

Initialize the training data by reading it from the dataset.

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 395 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.

Predict the outcome (positive/negative) of a given board state.

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 174 of file ml-naive-bayes.c.

resetTrainingData()

```
void resetTrainingData ()
```

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

Reset all training data and statistics to their initial state.

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 358 of file ml-naive-bayes.c.

5.22.2 Variable Documentation

actual

```
int actual
```

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

Definition at line 52 of file ml-naive-bayes.c.

cM

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

Confusion matrix for evaluating model performance.

Definition at line 44 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 42 of file ml-naive-bayes.c.

negativeClassProbability

```
double negativeClassProbability
```

Probability of a negative outcome in the dataset.

Definition at line 55 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 46 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 41 of file ml-naive-bayes.c.

positiveClassProbability

```
double positiveClassProbability
```

Probability of a positive outcome in the dataset.

Definition at line 54 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 45 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 51 of file ml-naive-bayes.c.

probabilityErrors

```
double probabilityErrors
```

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

Definition at line 56 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 48 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 49 of file ml-naive-bayes.c.

5.23 ml-naive-bayes.c

Go to the documentation of this file.

```
00001 #include <ml-naive-bayes.h>
00002 #include <math.h>
00003
00041 int positive_count = 0;
00042 int negative_count = 0;
00043
00044 int cM[4] = {0, 0, 0, 0};
00045 int positiveMoveCount[3][3][3];
00046 int negativeMoveCount[3][3][3];
00047
00048 int test_PredictedErrors = 0;
00049 int train_PredictedErrors = 0;
00050
00051 int predicted;
00052 int actual;
00053
00054 double positiveClassProbability;
00055 double negativeClassProbability;
00056 double probabilityErrors;
00057
00076 int assignMoveIndex(char move) //converts char to int value for easier calculation
00077 {
00078     switch (move)
00079     {
00080     case 'x':
00081         return BOT;
00082     case 'o':
00083         return PLAYER1;
00084     case 'b':
00085         return EMPTY;
00086     default:
00087         return ERROR;
00088     }
00089 }
00090
00106 void calculateProbabilities(int dataset_size)
00107 {
00108     // Calculate class probability
00109     positiveClassProbability = (double)positive_count / dataset_size;
00110     negativeClassProbability = (double)negative_count / dataset_size;
00111     PRINT_DEBUG("Positive Class Probability: %lf\n", positiveClassProbability);
00112     PRINT_DEBUG("Negative Class Probability: %lf\n", negativeClassProbability);
00113
00114     // Calculate conditional probability with laplace smoothing
00115     int laplace_smoothing = 1;
00116     for (int row = 0; row < 3; row++)
00117     {
00118         for (int col = 0; col < 3; col++)
00119         {
00120             for (int moveIndex = 0; moveIndex < 3; moveIndex++)
00121             {
00122                 char move;
00123                 if (moveIndex == 0) //convert
```

```
00124                 {
00125                     move = 'x';
00126                 }
00127                 else if (moveIndex == 1)
00128                 {
00129                     move = 'o';
00130                 }
00131                 else
00132                 {
00133                     move = 'b';
00134                 }
00135 
00136                 double positiveProbability = (double)(positiveMoveCount[row][col][moveIndex] +
      laplace_smoothing) / (positive_count + 3 * laplace_smoothing);
00137                 double negativeProbability = (double)(negativeMoveCount[row][col][moveIndex] +
      laplace_smoothing) / (negative_count + 3 * laplace_smoothing);
00138                 if (positive_count == 0)
00139                 {
00140                     PRINT_DEBUG("Probability of %c (positive) at grid(%d,%d): No positive outcomes\n",
      move, row, col);
00141                     PRINT_DEBUG("Probability of %c (negative) at grid(%d,%d): %lf\n", move, row, col,
      negativeProbability);
00142                 }
00143                 else if (negative_count == 0)
00144                 {
00145                     PRINT_DEBUG("Probability of %c (positive) at grid(%d,%d): %lf\n", move, row, col,
      positiveProbability);
00146                     PRINT_DEBUG("Probability of %c (negative) at grid(%d,%d): No negative outcomes\n",
      move, row, col);
00147                 }
00148                 else
00149                 {
00150                     PRINT_DEBUG("Probability of %c (positive) at grid(%d,%d): %lf\n", move, row, col,
      positiveProbability);
00151                     PRINT_DEBUG("Probability of %c (negative) at grid(%d,%d): %lf\n", move, row, col,
      negativeProbability);
00152                 }
00153             }
00154         }
00155     }
00156 }
00157 
00174 int predictOutcome(struct Dataset board)
00175 {
00176     double positiveProbability = positiveClassProbability;
00177     double negativeProbability = negativeClassProbability;
00178 
00179     // required as 0*anything = 0
00180     if (positiveProbability == 0)
00181     {
00182         positiveProbability = 1;
00183     }
00184     if (negativeProbability == 0)
00185     {
00186         negativeProbability = 1;
00187     }
00188 
00189     //loops through board grid and sums up probability for each grid
00190     for (int row = 0; row < 3; row++)
00191     {
00192         for (int col = 0; col < 3; col++)
00193         {
00194             int moveIndex = assignMoveIndex(board.grid[row][col]);
00195             if (moveIndex != -1)
00196             {
00197                 // PRINT_DEBUG("\nPC_%d, NC_%d, pMC_%d,
      nMC_%d",positive_count,negative_count,positiveMoveCount[row][col][moveIndex],negativeMoveCount[row][col][moveIndex]);
00198                 if (positive_count > 0)
00199                 {
00200                     positiveProbability *= (double)positiveMoveCount[row][col][moveIndex] /
      (double)positive_count;
00201                 }
00202 
00203                 if (negative_count > 0)
00204                 {
00205                     negativeProbability *= (double)negativeMoveCount[row][col][moveIndex] /
      (double)negative_count;
00206                 }
00207             }
00208         }
00209     }
00210 
00211     // guard cases if either negativeProbability is unset
00212     if (positiveProbability == 1)
00213     {
00214         positiveProbability = 0;
00215     }
```

```
00216     if (negativeProbability == 1)
00217     {
00218         negativeProbability = 0;
00219     }
00220 
00221     // Output probabilities for debugging
00222     // PRINT_DEBUG("\nPositive: %lf, Negative: %lf Probability: \n", positiveProbability,
      negativeProbability);
00223 
00224     //returns a value based on condition
00225     if (positiveProbability > negativeProbability)
00226     {
00227         // PRINT_DEBUG("Predicted Outcome: Positive\n");
00228         return 1;
00229     }
00230     else if (positiveProbability == 0 || negativeProbability == 0)
00231     {
00232         // PRINT_DEBUG("Unable to predict outcome based on available data.");
00233         return -1;
00234     }
00235     else
00236     {
00237         // PRINT_DEBUG("Predicted Outcome: Negative\n");
00238         return 0;
00239     }
00240 }
00241 
00242 
00258 struct Position getBestPosition(int grid[3][3], char player)
00259 {
00260     // Determine whether bot is X or O depending on current player
00261     char bot = (player == 'x' ? 'o' : 'x');
00262     char bestMove = 'b';
00263     int bestRow = -1;
00264     int bestCol = -1;
00265     double highestProbability = 0.0;
00266 
00267     int bot_count;
00268     int(*botMoveCount)[3][3];
00269 
00270     // Use positive or negative count for calculating probability depending on whether bot is X or O
00271     // 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
00272     if (bot == 'x')
00273     {
00274         bot_count = positive_count;
00275         botMoveCount = positiveMoveCount;
00276     }
00277     else
00278     {
00279         bot_count = negative_count;
00280         botMoveCount = negativeMoveCount;
00281     }
00282 
00283     for (int row = 0; row < 3; row++)
00284     {
00285         for (int col = 0; col < 3; col++)
00286         {
00287             // If the grid position is empty
00288             if (grid[row][col] != EMPTY)
00289             {
00290                 continue;
00291             }
00292 
00293             // Calculate probability for X or O to determine best move for bot
00294             for (int moveIndex = 0; moveIndex < 2; moveIndex++)
00295             {
00296                 double moveProbability;
00297 
00298                 if (bot == 'x')
00299                 {
00300                     // Calculate probability for move 'x'
00301                     if (bot_count > 0)
00302                     {
00303                         moveProbability = (double)botMoveCount[row][col][0] / bot_count;
00304                     }
00305                     else
00306                     {
00307                         moveProbability = 0.0;
00308                     }
00309                 }
00310                 else
00311                 {
00312                     // Calculate probability for move 'o'
00313                     if (bot_count > 0)
00314                     {
00315                         moveProbability = (double)botMoveCount[row][col][1] / bot_count;
```

```
00316                     }
00317                     else
00318                     {
00319                         moveProbability = 0.0;
00320                     }
00321                 }
00322 
00323                 // Update best move and position for bot if it has higher probability
00324                 if (moveProbability > highestProbability)
00325                 {
00326                     highestProbability = moveProbability;
00327                     bestMove = bot;
00328                     bestRow = row;
00329                     bestCol = col;
00330                 }
00331             }
00332         }
00333     }
00334 
00335     // Return best position
00336     if (bestRow != ERROR && bestCol != ERROR)
00337     {
00338         grid[bestRow][bestCol] = bestMove;
00339         PRINT_DEBUG("Best move: %c at grid (%d, %d) with probability: %lf\n", bestMove, bestRow,
      bestCol, highestProbability);
00340         return (struct Position){bestRow, bestCol};
00341     }
00342     else
00343     {
00344         PRINT_DEBUG("\nNo valid move found.\n");
00345         return (struct Position){ERROR, ERROR}; // Indicate no valid move found
00346     }
00347 }
00348 
00358 void resetTrainingData() {
00359     // Reset outcome counts
00360     positive_count = 0;
00361     negative_count = 0;
00362 
00363     // Reset move count arrays for each grid position
00364     for (int row = 0; row < 3; row++) {
00365         for (int col = 0; col < 3; col++) {
00366             for (int moveIndex = 0; moveIndex < 3; moveIndex++) {
00367                 positiveMoveCount[row][col][moveIndex] = 0;
00368                 negativeMoveCount[row][col][moveIndex] = 0;
00369             }
00370         }
00371     }
00372 
00373     // Reset the confusion matrix counters
00374     cM[0] = 0; // True positive
00375     cM[1] = 0; // False negative
00376     cM[2] = 0; // False positive
00377     cM[3] = 0; // True negative
00378 
00379     // Reset prediction errors
00380     test_PredictedErrors = 0;
00381     train_PredictedErrors = 0;
00382 }
00383 
00395 int initData()
00396 {
00397     resetTrainingData();
00398     int retVal = SUCCESS;
00399 
00400 doGetTrainingData:
00401     static bool doOnce = false;
00402     struct Dataset *trainingData = NULL;      // Initialize pointer
00403     int len = getTrainingData(&trainingData); // Pass address of pointer
00404 
00405     if (len <= 0)
00406     {
00407         retVal = readDataset(RES_PATH "" DATA_PATH, true);
00408         if (retVal != SUCCESS)
00409         {
00410             return retVal;
00411         }
00412 
00413         if (doOnce) //prevents potential loopback/deadlock. Edge case tbh.
00414         {
00415             return BAD_PARAM;
00416         }
00417 
00418         doOnce = true;
00419         goto doGetTrainingData; //loops until training data is set
00420     }
00421 
```

```
00422     //loops through train dataset for ml training
00423     for (int i = 0; i < len; i++)
00424     {
00425         // Get outcome class count for each position
00426         if (strcmp(trainingData[i].outcome, "positive") == 0)
00427         {
00428             positive_count++;
00429             for (int row = 0; row < 3; row++)
00430             {
00431                 for (int col = 0; col < 3; col++)
00432                 {
00433                     int moveIndex = assignMoveIndex(trainingData[i].grid[row][col]);
00434                     if (moveIndex != -1)
00435                     {
00436                         positiveMoveCount[row][col][moveIndex]++;
00437                     }
00438                 }
00439             }
00440         }
00441         else if (strcmp(trainingData[i].outcome, "negative") == 0)
00442         {
00443             negative_count++;
00444             for (int row = 0; row < 3; row++)
00445             {
00446                 for (int col = 0; col < 3; col++)
00447                 {
00448                     int moveIndex = assignMoveIndex(trainingData[i].grid[row][col]);
00449                     if (moveIndex != -1)
00450                     {
00451                         negativeMoveCount[row][col][moveIndex]++;
00452                     }
00453                 }
00454             }
00455         }
00456     }
00457     calcTrainErrors();
00458     calcConfusionMatrix();
00459     return SUCCESS;
00460 }
00461
00475 void assignCMValue(int actual, int predicted)
00476 {
00477     // PRINT_DEBUG("\nactual_%i, predicted_%i\n",actual,predicted);
00478
00479     if (actual == ERROR || predicted == ERROR)
00480     {
00481         PRINT_DEBUG("ERROR either value is -1. actual: %d predicted: %d", actual, predicted);
00482     }
00483
00484     if (actual == 1)
00485     {
00486         if (predicted == 1)
00487         {
00488             cM[0] += 1; // True positive
00489         }
00490         else
00491         {
00492             cM[1] += 1; // False negative
00493         }
00494     }
00495     else
00496     {
00497         if (predicted == 1)
00498         {
00499             cM[2] += 1; // False positive
00500         }
00501         else
00502         {
00503             cM[3] += 1; // True negative
00504         }
00505     }
00506 }
00507
00508 void calcConfusionMatrix()
00509 {
00510     //Tests ml on test dataset and stores result in a confusion matrix
00511
00512     struct Dataset *test = NULL;
00513     int len = getTestingData(&test);
00514     // PRINT_DEBUG("Test_Data length: %d\n", len);
00515     //loops through testing dataset
00516     if (len > 0)
00517     { // Ensure len is valid before accessing test
00518         for (int i = 0; i < len; i++)
00519         {
00520             actual = getTruthValue(test[i].outcome); //converts char* to int for comparison
00521             predicted = predictOutcome(test[i]);
```

```
00522
00523             // checks and updates total errors for test dataset
00524             if (actual != predicted)
00525             {
00526                 test_PredictedErrors += 1;
00527             }
00528
00529             //sets value based on actual vs predicted
00530             assignCMValue(actual, predicted);
00531         }
00532     }
00533
00534     double i = TESTING_DATA_SIZE;                       // assign macro to double as you cant cast
      macros
00535     probabilityErrors = (1 / i) * test_PredictedErrors; // round to 2dp? not in spec though
00536
00537     PRINT_DEBUG("For testing dataset: %d errors, %lf probability of error.\n", test_PredictedErrors,
      probabilityErrors);
00538     PRINT_DEBUG("TP: %d, FN: %d, FP: %d, TN: %d\n", cM[0], cM[1], cM[2], cM[3]);
00539 }
00540
00550 int getTruthValue(char *str1) //returns an integer value based on input
00551 {
00552     if (strcmp(str1, "positive") == 0)
00553     {
00554         return 1;
00555     }
00556     else if (strcmp(str1, "negative") == 0)
00557     {
00558         return 0;
00559     }
00560     else
00561     {
00562         //guard case if inputs are neither "positive" nor "negative"
00563         PRINT_DEBUG("ERROR: Not truth value: %p", str1);
00564         return -1;
00565     }
00566 }
00567
00576 void calcTrainErrors()
00577 {
00578     struct Dataset *train = NULL;      // Initialize pointer
00579     int len = getTrainingData(&train); // Pass address of pointer
00580     // debugDataset(test,len);
00581
00582     if (len > 0)
00583     { // Ensure len is valid before accessing test
00584         for (int i = 0; i < len; i++)
00585         {
00586             predicted = predictOutcome(train[i]);
00587             actual = getTruthValue(train[i].outcome);
00588             // PRINT_DEBUG("Actual dataset outcome: %s, Dataset outcome: %d, Predicted outcome: %d\n",
      test[i].outcome, actual, predicted);
00589             // checks and updates total errors for train dataset
00590             if (actual != predicted)
00591             {
00592                 train_PredictedErrors += 1;
00593             }
00594         }
00595     }
00596
00597     double i = TRAINING_DATA_SIZE;                       // assign macro to double var as macros cant
      be cast
00598     probabilityErrors = (1 / i) * train_PredictedErrors; // round to 2dp? not in spec though
00599
00600     PRINT_DEBUG("\nFor training dataset: %d errors, %lf probability of error.\n",
      train_PredictedErrors, probabilityErrors);
00601 }
00602
00615 void debugDataset(struct Dataset *data, int len)
00616 {
00617     PRINT_DEBUG("%d\n", len);
00618     if (len > 0)
00619     { // Ensure len is valid before accessing test
00620         for (int i = 0; i < len; i++)
00621         {
00622             PRINT_DEBUG("%d ", i);
00623             for (int j = 0; j < 3; j++)
00624             {
00625                 for (int k = 0; k < 3; k++)
00626                 {
00627                     PRINT_DEBUG("%c,", data->grid[j][k]);
00628                 }
00629             }
00630             PRINT_DEBUG("%s\n", data->outcome);
00631         }
00632     }
```

```
00633 }
```

Index