

TicTacToe

v1.0

Generated by Doxygen 1.12.0

| | |
|--|----------|
| 1 TicTacToe | 1 |
| 1.1 TIC TAC TOE - CSC1103 & CSC1104 Project | 1 |
| 1.1.1 Installation Instructions (Linux) | 1 |
| 1.1.2 Installation Instructions (MacOS) | 2 |
| 1.1.3 Installation Instructions (Windows) | 3 |
| 1.1.4 BASIC REQUIREMENTS (BOTH) | 4 |
| 1.1.5 [^1]PM-CSC1103 REQUIRMENTS | 4 |
| 1.1.6 [^2]COA-CSC1104 REQUIRMENTS | 5 |
| 2 Doxygen Documentation Generation and Viewing Instructions | 5 |
| 2.1 Steps to Generate and View the Documentation: | 5 |
| 2.1.1 1. Run the Doxygen Configuration File: | 5 |
| 2.1.2 2. View the HTML Documentation: | 5 |
| 2.1.3 3. Generate and View LaTeX Documentation: | 5 |
| 2.2 Troubleshooting: | 6 |
| 3 Data Structure Index | 6 |
| 3.1 Data Structures | 6 |
| 4 File Index | 6 |
| 4.1 File List | 6 |
| 5 Data Structure Documentation | 7 |
| 5.1 BoardState Struct Reference | 7 |
| 5.1.1 Detailed Description | 7 |
| 5.1.2 Field Documentation | 7 |
| 5.2 BtnPos Struct Reference | 7 |
| 5.2.1 Detailed Description | 7 |
| 5.2.2 Field Documentation | 8 |
| 5.3 Dataset Struct Reference | 8 |
| 5.3.1 Detailed Description | 8 |
| 5.3.2 Field Documentation | 8 |
| 5.4 PlayerMode Struct Reference | 8 |
| 5.4.1 Detailed Description | 8 |
| 5.4.2 Field Documentation | 9 |
| 5.5 Position Struct Reference | 9 |
| 5.5.1 Detailed Description | 9 |
| 5.5.2 Field Documentation | 9 |
| 6 File Documentation | 9 |
| 6.1 Documentation/README.md File Reference | 9 |
| 6.2 header/elapsedTime.h File Reference | 9 |
| 6.2.1 Detailed Description | 10 |
| 6.2.2 Function Documentation | 10 |

| | | |
|--------|--|----|
| 6.3 | elapsedTime.h | 11 |
| 6.4 | header/importData.h File Reference | 11 |
| 6.4.1 | Detailed Description | 11 |
| 6.4.2 | Macro Definition Documentation | 12 |
| 6.4.3 | Function Documentation | 12 |
| 6.5 | importData.h | 14 |
| 6.6 | header/macros.h File Reference | 14 |
| 6.6.1 | Detailed Description | 15 |
| 6.6.2 | Macro Definition Documentation | 15 |
| 6.7 | macros.h | 17 |
| 6.8 | header/main.h File Reference | 18 |
| 6.8.1 | Detailed Description | 19 |
| 6.8.2 | Function Documentation | 19 |
| 6.9 | main.h | 22 |
| 6.10 | header/minimax.h File Reference | 22 |
| 6.10.1 | Detailed Description | 23 |
| 6.10.2 | Macro Definition Documentation | 23 |
| 6.10.3 | Function Documentation | 23 |
| 6.11 | minimax.h | 28 |
| 6.12 | header/ml-naive-bayes.h File Reference | 28 |
| 6.12.1 | Detailed Description | 29 |
| 6.12.2 | Macro Definition Documentation | 30 |
| 6.12.3 | Function Documentation | 30 |
| 6.13 | ml-naive-bayes.h | 34 |
| 6.14 | mainpage.md File Reference | 34 |
| 6.15 | src/elapsedTime.c File Reference | 34 |
| 6.15.1 | Function Documentation | 35 |
| 6.15.2 | Variable Documentation | 35 |
| 6.16 | elapsedTime.c | 35 |
| 6.17 | src/importData.c File Reference | 36 |
| 6.17.1 | Function Documentation | 36 |
| 6.17.2 | Variable Documentation | 38 |
| 6.18 | importData.c | 39 |
| 6.19 | src/main.c File Reference | 40 |
| 6.19.1 | Function Documentation | 41 |
| 6.19.2 | Variable Documentation | 44 |
| 6.20 | main.c | 46 |
| 6.21 | src/minimax.c File Reference | 50 |
| 6.21.1 | Function Documentation | 51 |
| 6.21.2 | Variable Documentation | 56 |
| 6.22 | minimax.c | 56 |
| 6.23 | src/ml-naive-bayes.c File Reference | 61 |

| | |
|-------------------------------|-----------|
| 6.23.1 Function Documentation | 62 |
| 6.23.2 Variable Documentation | 66 |
| 6.24 ml-naive-bayes.c | 68 |
| Index | 75 |

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 verify 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. xquartz settings

- setting -> security -> allow connections from network clients
- Restart xquartz

4. Terminal command

```
xhost + 127.0.0.1
```

5. Build and Run Docker

```
docker-compose up --build -d
```

1.1.3 Installation Instructions (Windows)

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

1. Docker Desktop

- `Install Docker Desktop`

2. WSL2 (Windows Subsystem for Linux)

- Follow `Microsoft's guide to install WSL2`
- Install Ubuntu with the following command:

```
wsl --install -d Ubuntu
```
- Set Ubuntu as your default WSL distribution:

```
wsl --set-default Ubuntu
```
- Check default: `wsl -l -v`

1.1.3.1 Building the Project via Docker (Windows)

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

1.1.3.1.1 [OPTIONAL] Loading Docker Image

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

1.1.3.1.2 Build and Run TicTacToe Application on Docker (Windows)

```
./run_docker.sh
```

Warning

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

```
dos2unix SCRIPT_NAME.sh
```

Remarks

Saving Docker Image

```
./run_script.sh -s
```

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

Attention

Not recommended, unless you know what you're doing and install the right packages. (refer Dockerfile)

```
./compile.sh
```

1.1.4 BASIC REQUIREMENTS (BOTH)

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

1.1.5 [^1]PM-CSC1103 REQUIRMENTS

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

[TIP] Linear regression, Navie bayes, Neural network and Reinforcement learning

- ☒ Plot the confusion matrix for the training and testing accuracy
- ☐ Calculate the number of times the computer wins as a gauge of difficulty level.

1.1.6 [^2]COA-CSC1104 REQUIRMENTS

- ☒ Replace one function with assembly

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

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

2 Doxygen Documentation Generation and Viewing Instructions

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

2.1 Steps to Generate and View the Documentation:

2.1.1 1. Run the Doxygen Configuration File:

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

2.1.2 2. View the HTML Documentation:

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

2.1.2.1 Recommended Browsers:

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

2.1.3 3. Generate and View LaTeX Documentation:

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

2.2 Troubleshooting:

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

3 Data Structure Index

3.1 Data Structures

Here are the data structures with brief descriptions:

| | | |
|-------------------|--|---|
| BoardState | Stores the current state of the Tic-Tac-Toe board along with the best move | 7 |
| BtnPos | Stores the position of a button in the game grid | 7 |
| Dataset | Structure to hold a Tic-Tac-Toe board state and its outcome | 8 |
| PlayerMode | Stores the current game mode and its textual representation | 8 |
| Position | Represents a position on the Tic-Tac-Toe grid | 9 |

4 File Index

4.1 File List

Here is a list of all files with brief descriptions:

| | | |
|---------------------------------|--|----|
| header/ elapsedTime.h | Provides functionality for measuring elapsed time for profiling purposes | 9 |
| header/ importData.h | Header file for handling dataset import and manipulation for Tic-Tac-Toe game data | 11 |
| header/ macros.h | Header file containing macros, constants, and structure definitions for the Tic-Tac-Toe game | 14 |
| header/ main.h | Header file for the Tic-Tac-Toe game logic | 18 |
| header/ minimax.h | Header file for the Tic-Tac-Toe Minimax algorithm | 22 |
| header/ ml-naive-bayes.h | Header file for Naive Bayes classifier functions for Tic-Tac-Toe outcome prediction | 28 |
| src/ elapsedTime.c | | 34 |

| | |
|--------------------------------------|----|
| src/importData.c | 36 |
| src/main.c | 40 |
| src/minimax.c | 50 |
| src/ml-naive-bayes.c | 61 |

5 Data Structure Documentation

5.1 BoardState Struct Reference

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

```
#include <minimax.h>
```

Data Fields

- int [board](#) [3][3]
- struct [Position](#) [bestMove](#)

5.1.1 Detailed Description

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

< Include macro definitions

This structure holds a 3x3 board array and the best move associated with that board state. It is used for storing and comparing previous board states and their corresponding optimal moves.

Definition at line 29 of file [minimax.h](#).

5.1.2 Field Documentation

bestMove

```
struct Position bestMove
```

The best move for the bot

Definition at line 32 of file [minimax.h](#).

board

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

The Tic-Tac-Toe board

Definition at line 31 of file [minimax.h](#).

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

- header/[minimax.h](#)

5.2 BtnPos Struct Reference

Stores the position of a button in the game grid.

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

Data Fields

- int [pos](#) [2]

5.2.1 Detailed Description

Stores the position of a button in the game grid.

This structure contains an array `pos[2]` that holds the row and column of the button in the grid.

Definition at line 80 of file [main.h](#).

5.2.2 Field Documentation

pos

```
int pos[2]
```

Array to hold row and column

Definition at line 82 of file [main.h](#).

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

- header/[main.h](#)

5.3 Dataset Struct Reference

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

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

Data Fields

- char [grid](#) [3][3]
- char [outcome](#) [9]

5.3.1 Detailed Description

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

This structure represents the board state as a 3x3 grid and the outcome as a string.

Definition at line 28 of file [importData.h](#).

5.3.2 Field Documentation

grid

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

3x3 grid representing the board state

Definition at line 30 of file [importData.h](#).

outcome

```
char outcome[9]
```

Outcome of the board state

Definition at line 31 of file [importData.h](#).

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

- header/[importData.h](#)

5.4 PlayerMode Struct Reference

Stores the current game mode and its textual representation.

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

Data Fields

- char [txt](#) [2]
- int [mode](#)

5.4.1 Detailed Description

Stores the current game mode and its textual representation.

This structure holds the mode of the game (e.g., player vs player, player vs bot) and a textual representation of the mode for display.

Definition at line 92 of file [main.h](#).

5.4.2 Field Documentation

mode

```
int mode
```

Integer value representing the current game mode

Definition at line 95 of file [main.h](#).

txt

```
char txt[2]
```

Textual representation of the current game mode

Definition at line 94 of file [main.h](#).

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

- header/[main.h](#)

5.5 Position Struct Reference

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

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

Data Fields

- int [row](#)
- int [col](#)

5.5.1 Detailed Description

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

Definition at line 65 of file [macros.h](#).

5.5.2 Field Documentation

col

```
int col
```

Column index (0-2)

Definition at line 68 of file [macros.h](#).

row

```
int row
```

Row index (0-2)

Definition at line 67 of file [macros.h](#).

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

- header/[macros.h](#)

6 File Documentation

6.1 Documentation/README.md File Reference

6.2 header/elapsedTime.h File Reference

Provides functionality for measuring elapsed time for profiling purposes.

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

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

Functions

- void [startElapsedTime](#) ()
Starts the elapsed time tracking.
- void [stopElapsedTime](#) (char *str)
Stops the elapsed time tracking and outputs the result.

6.2.1 Detailed Description

Provides functionality for measuring elapsed time for profiling purposes.

Author

jacktan-jk

Version

1.0

Date

2024-11-13

Copyright

Copyright (c) 2024

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

Definition in file [elapsedTime.h](#).

6.2.2 Function Documentation

startElapsedTime()

```
void startElapsedTime ()
```

Starts the elapsed time tracking.

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

Definition at line 9 of file [elapsedTime.c](#).

stopElapsedTime()

```
void stopElapsedTime (  
    char * str)
```

Stops the elapsed time tracking and outputs the result.

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

Parameters

| | |
|------------|--|
| <i>str</i> | Label describing the operation or section being timed. |
|------------|--|

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

Definition at line 17 of file [elapsedTime.c](#).

6.3 elapsedTime.h

[Go to the documentation of this file.](#)

```
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
```

6.4 header/importData.h File Reference

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

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

Data Structures

- struct [Dataset](#)
Structure to hold a Tic-Tac-Toe board state and its outcome.

Macros

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

Functions

- int [readDataset](#) (const char *filename, bool split)
Reads a dataset from a file and optionally randomizes entries for training and testing.
- int [splitFile](#) ()
Splits the dataset into training and testing files with an 80-20 ratio.
- void [getRandomNo](#) (int random[[DATA_SIZE](#)])
Generates an array of unique random integers within the range of the dataset size.
- int [getTrainingData](#) (struct [Dataset](#) **d)
Retrieves the training data from a file and returns its length.
- int [getTestingData](#) (struct [Dataset](#) **d)
Retrieves the testing data from a file and returns its length.

6.4.1 Detailed Description

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

Author

jacktan-jk

Version

1.0

Date

2024-11-12

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

Definition in file [importData.h](#).

6.4.2 Macro Definition Documentation

DATA_PATH

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

Name of the primary dataset file

Definition at line 18 of file [importData.h](#).

RES_PATH

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

Path to resources directory

Definition at line 17 of file [importData.h](#).

TEST_PATH

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

Prefix for testing data file

Definition at line 20 of file [importData.h](#).

TRAIN_PATH

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

Prefix for training data file

Definition at line 19 of file [importData.h](#).

6.4.3 Function Documentation

getRandomNo()

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

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

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

Parameters

| | |
|---------------|--|
| <i>random</i> | Array to store the generated unique random integers. |
|---------------|--|

See also

[DATA_SIZE](#)

Definition at line 143 of file [importData.c](#).

getTestingData()

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

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

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

Parameters

| | |
|----------|---|
| <i>d</i> | Pointer to a dataset pointer that will reference the loaded testing data array. |
|----------|---|

Returns

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

See also

[readDataset](#), [testingFile](#)

Definition at line 175 of file [importData.c](#).

getTrainingData()

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

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

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

Parameters

| | |
|----------|--|
| <i>d</i> | Pointer to a dataset pointer that will reference the loaded training data array. |
|----------|--|

Returns

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

See also

[readDataset](#), [trainingFile](#)

Definition at line 167 of file [importData.c](#).

readDataset()

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

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

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

Parameters

| | |
|-----------------|--|
| <i>filename</i> | The name of the dataset file to read. |
| <i>split</i> | Boolean indicating whether to randomize entries for dataset splitting. |

Returns

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

See also

[getRandomNo](#), [splitFile](#)

Definition at line 44 of file [importData.c](#).

splitFile()

```
int splitFile ()
```

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

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

Returns

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

See also

[data](#), [trainingFile](#), [testingFile](#)

Definition at line 90 of file [importData.c](#).

6.5 importData.h

[Go to the documentation of this file.](#)

```
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
00049 int readDataset(const char *filename, bool split);
00050
00064 int splitFile();
00065
00077 void getRandomNo(int random[DATA_SIZE]);
00078
00092 int getTrainingData(struct Dataset **d);
00093
00107 int getTestingData(struct Dataset **d);
00108
00109 #endif // IMPORTDATA_H
```

6.6 header/macros.h File Reference

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

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

Data Structures

- struct [Position](#)

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

Macros

- #define [PLAY](#) 0x69
- #define [TIE](#) 0xFF

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

6.6.1 Detailed Description

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

Author

jacktan-jk

Version

1.0

Date

2024-11-13

Copyright

Copyright (c) 2024

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

Definition in file [macros.h](#).

6.6.2 Macro Definition Documentation

BAD_PARAM

```
#define BAD_PARAM -5
```

Bad parameter error

Definition at line 30 of file [macros.h](#).

BOT

```
#define BOT 2
```

Bot (Player 2)

Definition at line 40 of file [macros.h](#).

CLASSES

```
#define CLASSES 2
```

Number of outcome classes

Definition at line 44 of file [macros.h](#).

DATA_SIZE

```
#define DATA_SIZE 958
```

Dataset size

Definition at line 43 of file [macros.h](#).

DEBUG

```
#define DEBUG 1
```

Enable debug messages

Definition at line 47 of file [macros.h](#).

DISABLE_ASM

```
#define DISABLE_ASM
```

Disable ASM functions

Definition at line 51 of file [macros.h](#).

DISABLE_ELAPSED

```
#define DISABLE_ELAPSED
```

Disable Elapsed time function

Definition at line 50 of file [macros.h](#).

DISABLE_LOOKUP

```
#define DISABLE_LOOKUP
```

Disable Minimax lookup table

Definition at line 49 of file [macros.h](#).

EMPTY

```
#define EMPTY 0
```

Empty cell

Definition at line 38 of file [macros.h](#).

ERROR

```
#define ERROR -1
```

Error

Definition at line 29 of file [macros.h](#).

MINIMAX_GODMODE

```
#define MINIMAX_GODMODE
```

Minimax god mode toggle

Definition at line 48 of file [macros.h](#).

MODE_2P

```
#define MODE_2P 0
```

Two-player mode

Definition at line 33 of file [macros.h](#).

MODE_ML

```
#define MODE_ML 2
```

Machine Learning mode

Definition at line 35 of file [macros.h](#).

MODE_MM

```
#define MODE_MM 1
```

Minimax mode
Definition at line 34 of file [macros.h](#).

PLAY

```
#define PLAY 0x69
```

Player move
Definition at line 25 of file [macros.h](#).

PLAYER1

```
#define PLAYER1 1
```

Player 1
Definition at line 39 of file [macros.h](#).

PRINT_DEBUG

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

Value:
`printf(__VA_ARGS__);`
Definition at line 54 of file [macros.h](#).

SUCCESS

```
#define SUCCESS 0
```

Success
Definition at line 28 of file [macros.h](#).

TIE

```
#define TIE 0xFF
```

Tie state
Definition at line 26 of file [macros.h](#).

WIN

```
#define WIN 0xAA
```

Winning state
Definition at line 27 of file [macros.h](#).

6.7 macros.h

[Go to the documentation of this file.](#)

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

```

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

```

6.8 header/main.h File Reference

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

```

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

```

Data Structures

- struct [BtnPos](#)
Stores the position of a button in the game grid.
- struct [PlayerMode](#)
Stores the current game mode and its textual representation.

Functions

- int [doBOTmove](#) ()
Executes the bot's move based on the current game mode.
- int [chkPlayerWin](#) ()
Checks the current game board for a win or tie.
- void [clearGrid](#) ()
Clears the game board and resets the player's turn.
- void [updateScoreBtn](#) (gpointer [data](#))
Updates the score display on the button.
- void [on_btnGrid_clicked](#) (GtkWidget *widget, gpointer [data](#))
Callback function for handling button clicks on the game grid.
- void [on_btnScore_clicked](#) (GtkWidget *widget, gpointer [data](#))
Handles button click for score.
- void [showWin](#) ()
Clears the winning positions and resets the grid.

6.8.1 Detailed Description

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

Author

jacktan-jk

Version

1.0

Date

2024-11-12

Copyright

Copyright (c) 2024

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

Definition in file [main.h](#).

6.8.2 Function Documentation

chkPlayerWin()

```
int chkPlayerWin ()
```

Checks the current game board for a win or tie.

This function checks all possible win conditions:

- Diagonals
- Rows
- Columns

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

Returns

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

See also

[iBoard](#), [iWinPos](#)

Definition at line [234](#) of file [main.c](#).

clearGrid()

```
void clearGrid ()
```

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

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

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

See also

[iBoard](#)

[btnGrid](#)

[isPlayer1Turn](#)

Definition at line [29](#) of file [main.c](#).

doBOTmove()

```
int doBOTmove ()
```

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

In MM mode:

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

In ML mode, the bot uses machine learning to determine the best position.

The function also measures and logs the time taken for the minimax move.

Returns

SUCCESS if the bot's move was made successfully.

See also

[playerMode](#), [isMLAvail](#), [iBoard](#), [findBestMove](#), [getBestPosition](#), [btnGrid](#)

Definition at line 181 of file [main.c](#).

on_btnGrid_clicked()

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

Callback function for handling button clicks on the game grid.

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

Parameters

| | |
|---------------|--|
| <i>widget</i> | The GtkWidget that was clicked (the button in the grid). |
| <i>data</i> | Additional data passed to the callback (usually the score display data). |

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

See also

[iBoard](#), [isPlayer1Turn](#), [iPlayer1_score](#), [iPlayer2_score](#), [iTie_score](#)
[playerMode](#), [updateScoreBtn](#), [chkPlayerWin](#), [doBOTmove](#), [showWin](#)
[PLAY](#), [TIE](#), [WIN](#)

Definition at line 60 of file [main.c](#).

on_btnScore_clicked()

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

Handles button click for score.

Toggles the player mode and updates the displayed score.

Parameters

| | |
|---------------|---|
| <i>widget</i> | The widget that triggered the event. |
| <i>data</i> | Additional data passed to the callback. |

See also

[playerMode](#), [isMLAvail](#), [isPlayer1Turn](#), [updateScoreBtn](#), [clearGrid](#)

Definition at line 131 of file [main.c](#).

showWin()

```
void showWin ()
```

Clears the winning positions and resets the grid.

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

See also

[iWinPos](#), [btnGrid](#)

Definition at line 159 of file [main.c](#).

updateScoreBtn()

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

Updates the score display on the button.

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

Parameters

| | |
|-------------|--|
| <i>data</i> | A gpointer (usually a button widget) that is used to update the label. |
|-------------|--|

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

See also

[iPlayer1_score](#), [iTie_score](#), [iPlayer2_score](#), [playerMode](#)

Definition at line 42 of file [main.c](#).

6.9 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 =====*/
00080 typedef struct
00081 {
00082     int pos[2];
00083 } BtnPos;
00084
00092 struct PlayerMode
00093 {
00094     char txt[2];
00095     int mode;
00096 };
00097
00112 int doBOTmove();
00113
00129 int chkPlayerWin();
00130
00147 void clearGrid();
00148
00165 void updateScoreBtn(gpointer data);
00166
00192 void on_btnGrid_clicked(GtkWidget *widget, gpointer data);
00193
00204 void on_btnScore_clicked(GtkWidget *widget, gpointer data);
00205
00214 void showWin();
00215
00216 #endif // MAIN_H // End of include guard

```

6.10 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)
Returns the maximum of two integers.
- int [min](#) (int a, int b)
Returns the minimum of two integers.
- struct [Position](#) [findBestMove](#) (int board[3][3])
Finds the best move for the bot in the Tic-Tac-Toe game.
- int [minimax](#) (int board[3][3], int depth, bool isMax)
Implements the Minimax algorithm to evaluate the best move for the bot.
- int [evaluate](#) (int b[3][3])

- Evaluates the current board state to determine if there is a winner.*
- bool [isMovesLeft](#) (int board[3][3])
Checks if there are any moves left on the board.
- bool [checkAndUpdateBestMove](#) (int board[3][3], struct [Position](#) *bestMove, struct [BoardState](#) boardStates[], int count)
Checks if the current board configuration exists in the lookup table and updates the best move.
- void [writeBestMoveToFile](#) (int board[3][3], struct [Position](#) bestMove)
Appends the current board state and the best move to a file.
- int [loadBoardStates](#) (struct [BoardState](#) boardStates[])
Loads board states and their best moves from a file.
- void [printFileContents](#) ()
Prints the contents of the best move file.

6.10.1 Detailed Description

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

Author

jacktan-jk

Version

1.0

Date

2024-11-12

Copyright

Copyright (c) 2024

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

Definition in file [minimax.h](#).

6.10.2 Macro Definition Documentation

FILE_BESTMOV

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

Path to the file storing best moves

Definition at line 35 of file [minimax.h](#).

MAX_BOARDS

```
#define MAX_BOARDS 10000
```

Maximum number of boards to store in memory

Definition at line 36 of file [minimax.h](#).

6.10.3 Function Documentation

checkAndUpdateBestMove()

```
bool checkAndUpdateBestMove (
    int board[3][3],
    struct Position * bestMove,
```

```
struct BoardState boardStates[],
int count)
```

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

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

Parameters

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

Returns

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

See also

[BoardState](#), [Position](#)

Definition at line 419 of file [minimax.c](#).

evaluate()

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

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

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

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

Parameters

| | |
|----------|---|
| <i>b</i> | A 3x3 array representing the Tic-Tac-Toe board. |
|----------|---|

Returns

The evaluation score:

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

See also

[BOT](#), [PLAYER1](#)

Definition at line 220 of file [minimax.c](#).

findBestMove()

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

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

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

Parameters

| | |
|--------------|---|
| <i>board</i> | A 3x3 array representing the current Tic-Tac-Toe board. |
|--------------|---|

Returns

The best move for the bot as a struct [Position](#) containing the row and column.

See also

[minimax](#), [loadBoardStates](#), [checkAndUpdateBestMove](#), [writeBestMoveToFile](#)

Definition at line 71 of file [minimax.c](#).

isMovesLeft()

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

Checks if there are any moves left on the board.

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

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

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

Parameters

| | |
|--------------|---|
| <i>board</i> | A 3x3 array representing the board state. Each cell should contain: <ul style="list-style-type: none"> • <code>EMPTY</code> (typically 0) if the cell is empty. • Any non-zero value if the cell is occupied. |
|--------------|---|

Returns

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

Note

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

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

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

Example usage:

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

Definition at line 270 of file [minimax.c](#).

loadBoardStates()

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

Loads board states and their best moves from a file.

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

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

Parameters

| | |
|--------------------------|--|
| <code>boardStates</code> | An array of <code>BoardState</code> structures to store the loaded board states. |
|--------------------------|--|

Returns

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

See also

[BoardState](#), [FILE_BESTMOV](#)

Definition at line 366 of file [minimax.c](#).

max()

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

Returns the maximum of two integers.

This function computes the maximum of two integers using either:

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

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

Parameters

| | |
|----------------|--------------------------------|
| <code>a</code> | The first integer to compare. |
| <code>b</code> | The second integer to compare. |

Returns

The greater of the two integers.

Note

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

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

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

Example usage:

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

Definition at line 5 of file [minimax.c](#).

min()

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

Returns the minimum of two integers.

This function computes the minimum of two integers using either:

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

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

Parameters

| | |
|----------|--------------------------------|
| <i>a</i> | The first integer to compare. |
| <i>b</i> | The second integer to compare. |

Returns

The smaller of the two integers.

Note

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

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

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

Example usage:

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

Definition at line 38 of file [minimax.c](#).

minimax()

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

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

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

Parameters

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

Returns

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

See also

[evaluate](#), [isMovesLeft](#), [max](#), [min](#)

Definition at line 136 of file [minimax.c](#).

printFileContents()

```
void printFileContents ()
```

Prints the contents of the best move file.

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

writeBestMoveToFile()

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

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

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

Parameters

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

See also

[Position](#), [BoardState](#)

Definition at line 436 of file [minimax.c](#).

6.11 minimax.h

[Go to the documentation of this file.](#)

```
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
00061 int max(int a, int b);
00062
00086 int min(int a, int b);
00087
00103 struct Position findBestMove(int board[3][3]);
00104
00123 int minimax(int board[3][3], int depth, bool isMax);
00124
00144 int evaluate(int b[3][3]);
00145
00177 bool isMovesLeft(int board[3][3]);
00178
00196 bool checkAndUpdateBestMove(int board[3][3], struct Position *bestMove, struct BoardState
    boardStates[], int count);
00197
00210 void writeBestMoveToFile(int board[3][3], struct Position bestMove);
00211
00232 int loadBoardStates(struct BoardState boardStates[]);
00233
00240 void printFileContents();
00241
00242 #endif // MINIMAX_H // End of include guard
```

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

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

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

Macros

- #define [TRAINING_DATA_SIZE](#) 0.8 * [DATA_SIZE](#)
- #define [TESTING_DATA_SIZE](#) 0.2 * [DATA_SIZE](#)
- #define [CLASSES](#) 2

Functions

- int [assignMoveIndex](#) (char move)
Assigns an index to each move ("x", "o", or "b").
- void [calculateProbabilities](#) (int dataset_size)
Calculates the probabilities for each class and conditional probabilities with Laplace smoothing.
- void [resetTrainingData](#) ()
Resets the training data and associated statistics for a fresh training cycle.
- int [initData](#) ()
Initializes the training data and model statistics.
- int [predictOutcome](#) (struct [Dataset](#) board)
Predicts the outcome of a given Tic Tac Toe board based on previously calculated probabilities.
- void [calcTrainErrors](#) ()
Calculates the training errors and the probability of error.
- void [calcConfusionMatrix](#) ()
Calculates the confusion matrix and error probability for the testing dataset.
- struct [Position](#) [getBestPosition](#) (int grid[3][3], char player)
Determines the best position for the bot to make a move based on the highest probability.
- int [getTruthValue](#) (char *str1)
Returns an integer value representing a truth value based on input.
- void [assignCMValue](#) (int [actual](#), int [predicted](#))
Updates the confusion matrix based on actual and predicted outcomes.
- void [debugDataset](#) (struct [Dataset](#) *data, int len)
Debug function to display dataset contents.

6.12.1 Detailed Description

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

Author

jacktan-jk

Version

1.0

Date

2024-11-13

Copyright

Copyright (c) 2024

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

Definition in file [ml-naive-bayes.h](#).

6.12.2 Macro Definition Documentation

CLASSES

```
#define CLASSES 2
```

Number of possible outcome classes (positive/negative)

Definition at line 26 of file [ml-naive-bayes.h](#).

TESTING_DATA_SIZE

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

Size of the testing dataset (20%)

Definition at line 25 of file [ml-naive-bayes.h](#).

TRAINING_DATA_SIZE

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

Size of the training dataset (80%)

Definition at line 24 of file [ml-naive-bayes.h](#).

6.12.3 Function Documentation

assignCMValue()

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

Updates the confusion matrix based on actual and predicted outcomes.

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

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

Parameters

| | |
|------------------|---|
| <i>actual</i> | The actual outcome value (1 for positive, 0 for negative). |
| <i>predicted</i> | The predicted outcome value (1 for positive, 0 for negative). |

See also

[cM](#), [ERROR](#)

Definition at line 340 of file [ml-naive-bayes.c](#).

assignMoveIndex()

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

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

This function maps the board move characters to their corresponding integer values:

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

Parameters

| | |
|-------------|---|
| <i>move</i> | The character representing the move ('x', 'o', or 'b'). |
|-------------|---|

Returns

int The integer corresponding to the move:

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

See also

[BOT](#), [PLAYER1](#), [EMPTY](#), [ERROR](#)

Definition at line 21 of file [ml-naive-bayes.c](#).

calcConfusionMatrix()

```
void calcConfusionMatrix ()
```

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

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

See also

[cM](#), [test_PredictedErrors](#), [probabilityErrors](#), [getTruthValue](#), [predictOutcome](#)

Definition at line 373 of file [ml-naive-bayes.c](#).

calcTrainErrors()

```
void calcTrainErrors ()
```

Calculates the training errors and the probability of error.

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

See also

[train_PredictedErrors](#), [probabilityErrors](#), [getTruthValue](#), [predictOutcome](#)

Definition at line 424 of file [ml-naive-bayes.c](#).

calculateProbabilities()

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

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

This function calculates:

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

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

Parameters

| | |
|---------------------|--|
| <i>dataset_size</i> | The total number of samples in the dataset used for probability calculation. |
|---------------------|--|

See also

[positive_count](#), [negative_count](#), [positiveMoveCount](#), [negativeMoveCount](#)

Definition at line 36 of file [ml-naive-bayes.c](#).

debugDataset()

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

Debug function to display dataset contents.

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

Parameters

| | |
|-------------|--|
| <i>data</i> | Pointer to the dataset to be printed. |
| <i>len</i> | The length of the dataset (number of entries). |

See also

[PRINT_DEBUG](#)

Definition at line 451 of file [ml-naive-bayes.c](#).

getBestPosition()

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

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

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

Parameters

| | |
|---------------|--|
| <i>grid</i> | The current state of the Tic Tac Toe game board. |
| <i>player</i> | The current player, either 'x' or 'o'. |

Returns

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

See also

[positive_count](#), [negative_count](#), [positiveMoveCount](#), [negativeMoveCount](#)

Definition at line 156 of file [ml-naive-bayes.c](#).

getTruthValue()

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

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

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

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

Parameters

| | |
|-------------|--|
| <i>str1</i> | A pointer to the input string to be evaluated. |
|-------------|--|

Returns

int The corresponding truth value:

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

See also

[PRINT_DEBUG](#)

Definition at line 406 of file [ml-naive-bayes.c](#).

initData()

```
int initData ()
```

Initializes the training data and model statistics.

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

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

See also

[resetTrainingData](#), [getTrainingData](#), [calcTrainErrors](#), [calcConfusionMatrix](#)

Definition at line 273 of file [ml-naive-bayes.c](#).

predictOutcome()

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

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

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

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

Parameters

| | |
|--------------|--|
| <i>board</i> | The current Tic Tac Toe board whose outcome needs to be predicted. |
|--------------|--|

Returns

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

See also

[positiveClassProbability](#), [negativeClassProbability](#), [positiveMoveCount](#), [negativeMoveCount](#), [assignMoveIndex](#)

Definition at line 88 of file [ml-naive-bayes.c](#).

resetTrainingData()

```
void resetTrainingData ()
```

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

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

See also

[positive_count](#), [negative_count](#), [positiveMoveCount](#), [negativeMoveCount](#), [cM](#), [test_PredictedErrors](#), [train_PredictedErrors](#)

Definition at line 247 of file [ml-naive-bayes.c](#).

6.13 ml-naive-bayes.h

[Go to the documentation of this file.](#)

```
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
00084 int assignMoveIndex(char move);
00085
00101 void calculateProbabilities(int dataset_size);
00102
00112 void resetTrainingData();
00113
00125 int initData();
00126
00143 int predictOutcome(struct Dataset board);
00144
00153 void calcTrainErrors();
00154
00164 void calcConfusionMatrix();
00165
00181 struct Position getBestPosition(int grid[3][3], char player);
00182
00201 int getTruthValue(char *str1);
00202
00216 void assignCMValue(int actual, int predicted);
00217
00229 void debugDataset(struct Dataset *data, int len);
00230
00231 #endif // ML_NAIVE_BAYES_H
```

6.14 mainpage.md File Reference

6.15 src/elapsedTime.c File Reference

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

Functions

- void [startElapseTime](#) ()
Starts the elapsed time tracking.
- void [stopElapseTime](#) (char *str)
Stops the elapsed time tracking and outputs the result.

Variables

- struct timeval [gTime](#)
- double [gStartTime](#)
- double [gEndTime](#)

6.15.1 Function Documentation

startElapsedTime()

```
void startElapsedTime ()
```

Starts the elapsed time tracking.

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

Definition at line 9 of file [elapsedTime.c](#).

stopElapsedTime()

```
void stopElapsedTime (
    char * str)
```

Stops the elapsed time tracking and outputs the result.

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

Parameters

| | |
|------------|--|
| <i>str</i> | Label describing the operation or section being timed. |
|------------|--|

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

Definition at line 17 of file [elapsedTime.c](#).

6.15.2 Variable Documentation

gEndTime

```
double gEndTime
```

Holds the end time in seconds for the timed section.

Definition at line 6 of file [elapsedTime.c](#).

gStartTime

```
double gStartTime
```

Holds the start time in seconds for the timed section.

Definition at line 5 of file [elapsedTime.c](#).

gTime

```
struct timeval gTime
```

Stores the current time values for elapsed time calculation.

Definition at line 4 of file [elapsedTime.c](#).

6.16 elapsedTime.c

[Go to the documentation of this file.](#)

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

```

00019 #if !(DISABLE_ELAPSED)
00020     gettimeofday(&gTime, NULL);
00021     gEndTime = gTime.tv_sec + 1.0e-6 * gTime.tv_usec;
00022     PRINT_DEBUG("[ELAPSED] %s -> took %f seconds \n\n", str, (double)(gEndTime - gStartTime));
00023 #endif
00024 }

```

6.17 src/importData.c File Reference

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

Functions

- int [readDataset](#) (const char *filename, bool split)
Reads a dataset from a file and optionally randomizes entries for training and testing.
- int [splitFile](#) ()
Splits the dataset into training and testing files with an 80-20 ratio.
- void [getRandomNo](#) (int random[DATA_SIZE])
Generates an array of unique random integers within the range of the dataset size.
- int [getTrainingData](#) (struct [Dataset](#) **d)
Retrieves the training data from a file and returns its length.
- int [getTestingData](#) (struct [Dataset](#) **d)
Retrieves the testing data from a file and returns its length.

Variables

- int [len_train](#) = 0
Global variable to store the number of training dataset entries.
- int [len_test](#) = 0
Global variable to store the number of testing dataset entries.
- int [randomNo](#) [DATA_SIZE]
Global array to store unique random indices for dataset splitting.
- struct [Dataset](#) [data](#) [DATA_SIZE]
Global array to store the dataset.
- const char * [trainingFile](#) = RES_PATH "" TRAIN_PATH "" DATA_PATH
Global variable to store the path for the training dataset file.
- const char * [testingFile](#) = RES_PATH "" TEST_PATH "" DATA_PATH
Global variable to store the path for the testing dataset file.

6.17.1 Function Documentation

[getRandomNo\(\)](#)

```

void getRandomNo (
    int random[DATA_SIZE])

```

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

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

Parameters

| | |
|---------------|--|
| <i>random</i> | Array to store the generated unique random integers. |
|---------------|--|

See also

[DATA_SIZE](#)

Definition at line 143 of file [importData.c](#).

getTestingData()

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

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

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

Parameters

| | |
|----------|---|
| <i>d</i> | Pointer to a dataset pointer that will reference the loaded testing data array. |
|----------|---|

Returns

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

See also

[readDataset](#), [testingFile](#)

Definition at line 175 of file [importData.c](#).

getTrainingData()

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

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

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

Parameters

| | |
|----------|--|
| <i>d</i> | Pointer to a dataset pointer that will reference the loaded training data array. |
|----------|--|

Returns

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

See also

[readDataset](#), [trainingFile](#)

Definition at line 167 of file [importData.c](#).

readDataset()

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

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

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

Parameters

| | |
|-----------------|--|
| <i>filename</i> | The name of the dataset file to read. |
| <i>split</i> | Boolean indicating whether to randomize entries for dataset splitting. |

Returns

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

See also

[getRandomNo](#), [splitFile](#)

Definition at line 44 of file [importData.c](#).

splitFile()

```
int splitFile ()
```

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

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

Returns

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

See also

[data](#), [trainingFile](#), [testingFile](#)

Definition at line 90 of file [importData.c](#).

6.17.2 Variable Documentation**data**

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

Global array to store the dataset.

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

Definition at line 38 of file [importData.c](#).

len_test

```
int len_test = 0
```

Global variable to store the number of testing dataset entries.

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

Definition at line 36 of file [importData.c](#).

len_train

```
int len_train = 0
```

Global variable to store the number of training dataset entries.

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

Definition at line 35 of file [importData.c](#).

randomNo

```
int randomNo[DATA_SIZE]
```

Global array to store unique random indices for dataset splitting.

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

Definition at line 37 of file [importData.c](#).

testingFile

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

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

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

Definition at line 42 of file [importData.c](#).

trainingFile

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

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

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

Definition at line 41 of file [importData.c](#).

6.18 importData.c

[Go to the documentation of this file.](#)

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

```

00105     for (int i = 0; eighty > i; i++)
00106     {
00107         for (int row = 0; 3 > row; row++)
00108         {
00109             for (int col = 0; 3 > col; col++)
00110             {
00111                 fprintf(trainFile, "%c,", data[i].grid[row][col]);
00112             }
00113             fprintf(trainFile, "%s\n", data[i].outcome);
00114         }
00115     }
00116     fclose(trainFile);
00117
00118     // write into testing dataset
00119     FILE *testFile;
00120     testFile = fopen(testingFile, "w");
00121     if (!testFile)
00122     {
00123         PRINT_DEBUG("[ERROR] Error opening file.\n");
00124         return BAD_PARAM;
00125     }
00126
00127     for (int i = eighty; DATA_SIZE > i; i++)
00128     {
00129         for (int row = 0; 3 > row; row++)
00130         {
00131             for (int col = 0; 3 > col; col++)
00132             {
00133                 fprintf(testFile, "%c,", data[i].grid[row][col]);
00134             }
00135             fprintf(testFile, "%s\n", data[i].outcome);
00136         }
00137     }
00138     fclose(testFile);
00139     return SUCCESS;
00140 }
00141
00142 void getRandomNo(int random[DATA_SIZE])
00143 {
00144     int count = 0;
00145     srand(time(NULL));
00146
00147     // initialize all to 0 for proper check
00148     int check[DATA_SIZE];
00149     for (int i = 0; DATA_SIZE > i; i++)
00150     {
00151         check[i] = 0;
00152     }
00153
00154     while (DATA_SIZE > count)
00155     {
00156         int randNo = rand() % DATA_SIZE;
00157         if (check[randNo] == 0)
00158         {
00159             check[randNo] = 1;
00160             random[count] = randNo;
00161             count++;
00162         }
00163     }
00164 }
00165
00166 int getTrainingData(struct Dataset **d)
00167 {
00168     memset(data, 0, len_train * sizeof(struct Dataset));
00169     readDataset(trainingFile, false);
00170     *d = data;
00171     return len_train;
00172 }
00173
00174 int getTestingData(struct Dataset **d)
00175 {
00176     memset(data, 0, len_test * sizeof(struct Dataset));
00177     readDataset(testingFile, false);
00178     *d = data;
00179     return len_test;
00180 }
00181

```

6.19 src/main.c File Reference

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

Functions

- void `clearGrid` ()
Clears the game board and resets the player's turn.
- void `updateScoreBtn` (gpointer `data`)
Updates the score display on the button.
- void `on_btnGrid_clicked` (GtkWidget *widget, gpointer `data`)
Callback function for handling button clicks on the game grid.
- void `on_btnScore_clicked` (GtkWidget *widget, gpointer `data`)
Handles button click for score.
- void `showWin` ()
Clears the winning positions and resets the grid.
- int `doBOTmove` ()
Executes the bot's move based on the current game mode.
- int `chkPlayerWin` ()
Checks the current game board for a win or tie.
- int `main` (int argc, char *argv[])
Initializes and runs the Tic-Tac-Toe GTK application.

Variables

- int `iPlayer1_score` = 0
Global variable to track Player 1's score.
- int `iPlayer2_score` = 0
Global variable to track Player 2's or Bot's (Minimax/ML) score.
- int `iTie_score` = 0
Global variable to track the number of ties/draws.
- int `iGameState` = `PLAY`
Global variable to track the current game state.
- int `iBoard` [3][3]
Global 2D array representing the Tic-Tac-Toe game board.
- int `iWinPos` [3][3]
Global 2D array to track winning positions on the board.
- bool `isPlayer1Turn` = true
Global flag indicating if it's Player 1's turn.
- bool `isMLAvail` = true
Global flag indicating if Machine Learning mode is available. This is set to false if the ML data file is missing, disabling the ML game mode.
- struct `PlayerMode playerMode` = {"2P", `MODE_2P`}
Global structure to track the current game mode.
- GtkWidget * `btnGrid` [3][3]
Global 2D array of buttons corresponding to the game grid.

6.19.1 Function Documentation

`chkPlayerWin()`

```
int chkPlayerWin ()
```

Checks the current game board for a win or tie.

This function checks all possible win conditions:

- Diagonals
- Rows

- Columns

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

Returns

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

See also

[iBoard](#), [iWinPos](#)

Definition at line 234 of file [main.c](#).

clearGrid()

```
void clearGrid ()
```

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

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

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

See also

[iBoard](#)

[btnGrid](#)

[isPlayer1Turn](#)

Definition at line 29 of file [main.c](#).

doBOTmove()

```
int doBOTmove ()
```

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

In MM mode:

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

In ML mode, the bot uses machine learning to determine the best position.

The function also measures and logs the time taken for the minimax move.

Returns

SUCCESS if the bot's move was made successfully.

See also

[playerMode](#), [isMLAvail](#), [iBoard](#), [findBestMove](#), [getBestPosition](#), [btnGrid](#)

Definition at line 181 of file [main.c](#).

main()

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

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

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

Parameters

| | |
|-------------|--|
| <i>argc</i> | The number of arguments passed to the program. |
| <i>argv</i> | The list of arguments passed to the program. |

Returns

SUCCESS if the program runs successfully.

See also

[initData](#), [on_btnScore_clicked](#), [on_btnGrid_clicked](#), [btnGrid](#)

Definition at line 302 of file [main.c](#).

on_btnGrid_clicked()

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

Callback function for handling button clicks on the game grid.

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

Parameters

| | |
|---------------|--|
| <i>widget</i> | The GtkWidget that was clicked (the button in the grid). |
| <i>data</i> | Additional data passed to the callback (usually the score display data). |

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

See also

[iBoard](#), [isPlayer1Turn](#), [iPlayer1_score](#), [iPlayer2_score](#), [iTie_score](#)
[playerMode](#), [updateScoreBtn](#), [chkPlayerWin](#), [doBOTmove](#), [showWin](#)
[PLAY](#), [TIE](#), [WIN](#)

Definition at line 60 of file [main.c](#).

on_btnScore_clicked()

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

Handles button click for score.

Toggles the player mode and updates the displayed score.

Parameters

| | |
|---------------|---|
| <i>widget</i> | The widget that triggered the event. |
| <i>data</i> | Additional data passed to the callback. |

See also

[playerMode](#), [isMLAvail](#), [isPlayer1Turn](#), [updateScoreBtn](#), [clearGrid](#)

Definition at line 131 of file [main.c](#).

showWin()

```
void showWin ()
```

Clears the winning positions and resets the grid.

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

See also

[iWinPos](#), [btnGrid](#)

Definition at line 159 of file [main.c](#).

updateScoreBtn()

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

Updates the score display on the button.

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

Parameters

| | |
|-------------|--|
| <i>data</i> | A gpointer (usually a button widget) that is used to update the label. |
|-------------|--|

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

See also

[iPlayer1_score](#), [iTie_score](#), [iPlayer2_score](#), [playerMode](#)

Definition at line 42 of file [main.c](#).

6.19.2 Variable Documentation**btnGrid**

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

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

Definition at line 19 of file [main.c](#).

iBoard

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

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

Definition at line 11 of file [main.c](#).

iGameState

```
int iGameState = PLAY
```

Global variable to track the current game state.

Game states:

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

Definition at line 10 of file [main.c](#).

iPlayer1_score

```
int iPlayer1_score = 0
```

Global variable to track Player 1's score.

Definition at line 7 of file [main.c](#).

iPlayer2_score

```
int iPlayer2_score = 0
```

Global variable to track Player 2's or Bot's (Minimax/ML) score.

Definition at line 8 of file [main.c](#).

isMLAvail

```
bool isMLAvail = true
```

Global flag indicating if Machine Learning mode is available. This is set to false if the ML data file is missing, disabling the ML game mode.

Definition at line 15 of file [main.c](#).

isPlayer1Turn

```
bool isPlayer1Turn = true
```

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

Definition at line 14 of file [main.c](#).

iTie_score

```
int iTie_score = 0
```

Global variable to track the number of ties/draws.

Definition at line 9 of file [main.c](#).

iWinPos

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

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

Definition at line 12 of file [main.c](#).

playerMode

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

Global structure to track the current game mode.

Fields:

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

Player modes:

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

Definition at line 17 of file [main.c](#).

6.20 main.c

[Go to the documentation of this file.](#)

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

```

00057
00058
00059 // Callback function for button clicks
00060 void on_btnGrid_clicked(GtkWidget *widget, gpointer data)
00061 {
00062     const char *current_label = gtk_button_get_label(GTK_BUTTON(widget));
00063     BtnPos *btnPos = (BtnPos *)g_object_get_data(G_OBJECT(widget), "button-data");
00064
00065     if (iGameState != PLAY)
00066     {
00067         iGameState = PLAY;
00068         clearGrid();
00069         updateScoreBtn(data);
00070         return;
00071     }
00072
00073     if (strcmp(current_label, "") != 0)
00074     {
00075         return;
00076     }
00077
00078     iBoard[btnPos->pos[0]][btnPos->pos[1]] = isPlayer1Turn ? PLAYER1 : BOT; // O (1), X(2), BOT is the
same as player 2
00079
00080     // Update the button text, for example, with an "O"
00081     gtk_button_set_label(GTK_BUTTON(widget), isPlayer1Turn ? "O" : "X");
00082
00083     int retVal = chkPlayerWin();
00084
00085     if (retVal == PLAY)
00086     {
00087         isPlayer1Turn = !isPlayer1Turn;
00088         updateScoreBtn(data);
00089
00090         if (playerMode.mode == MODE_2P)
00091         {
00092             return;
00093         }
00094
00095         doBOTmove();
00096         retVal = chkPlayerWin();
00097     }
00098
00099     if (retVal == WIN)
00100     {
00101         showWin();
00102         PRINT_DEBUG("[DEBUG] GAME RESULT -> %s Win\n", isPlayer1Turn ? "Player 1" : playerMode.mode ==
MODE_2P ? "Player 2"
00103 : "BOT");
00104         isPlayer1Turn ? iPlayer1_score++ : iPlayer2_score++;
00105         iGameState = WIN;
00106     }
00107
00108     if (retVal == TIE)
00109     {
00110         PRINT_DEBUG("[DEBUG] GAME RESULT -> TIE\n");
00111         iTie_score++;
00112         iGameState = TIE;
00113     }
00114
00115     if (playerMode.mode != MODE_2P)
00116     {
00117         isPlayer1Turn = !isPlayer1Turn;
00118     }
00119
00120     if (isMLAvail && playerMode.mode == MODE_ML)
00121     {
00122         if (retVal == WIN || retVal == TIE)
00123         {
00124             readDataset(RES_PATH "" DATA_PATH, true);
00125             initData();
00126         }
00127     }
00128     updateScoreBtn(data);
00129 }
00130
00131 void on_btnScore_clicked(GtkWidget *widget, gpointer data)
00132 {
00133     playerMode.mode = (playerMode.mode > 1 ? MODE_2P : ++playerMode.mode);
00134     switch (playerMode.mode)
00135     {
00136     case MODE_MM:
00137         strncpy(playerMode.txt, "MM", sizeof(playerMode.txt));
00138         break;
00139     case MODE_ML:

```

```

00141         if (isMLAvail)
00142         {
00143             strncpy(playerMode.txt, "ML", sizeof(playerMode.txt));
00144             break;
00145         }
00146
00147     default:
00148         playerMode.mode = MODE_2P;
00149         strncpy(playerMode.txt, "2P", sizeof(playerMode.txt));
00150     }
00151     PRINT_DEBUG("playerMode: %d\n", playerMode.mode);
00152     isPlayer1Turn = true;
00153     iPlayer1_score = iPlayer2_score = iTie_score = 0;
00154
00155     clearGrid();
00156     updateScoreBtn(data);
00157 }
00158
00159 void showWin()
00160 {
00161     for (int i = 0; i < 3; i++)
00162     {
00163         for (int j = 0; j < 3; j++)
00164         {
00165             if (iWinPos[i][j] != WIN)
00166             {
00167                 gtk_button_set_label(GTK_BUTTON(btnGrid[i][j]), "");
00168             }
00169         }
00170     }
00171     memset(iWinPos, 0, sizeof(iWinPos));
00172 }
00173 /*=====
00174 END OF GUI FUNCTIONS
00175 =====*/
00176
00177 /*=====
00178 LOGIC FUNCTIONS
00179 =====*/
00180
00181 int doBOTmove()
00182 {
00183     struct Position botMove;
00184     if (playerMode.mode == MODE_MM)
00185     {
00186         startElapseTime();
00187         #if !(MINIMAX_GODMODE)
00188             if (rand() % 100 < 80)
00189             #endif
00190             {
00191                 botMove = findBestMove(iBoard);
00192             }
00193         #if !(MINIMAX_GODMODE)
00194             else
00195             {
00196                 startElapseTime();
00197                 int randRow = rand() % 3;
00198                 int randCol = rand() % 3;
00199                 bool bIsDone = false;
00200
00201                 while (!bIsDone)
00202                 {
00203                     if (iBoard[randRow][randCol] == EMPTY)
00204                     {
00205                         PRINT_DEBUG("Random Move -> R:%d C:%d\n", randRow, randCol);
00206                         botMove.row = randRow;
00207                         botMove.col = randCol;
00208                         bIsDone = !bIsDone;
00209                     }
00210                     else
00211                     {
00212                         randRow = rand() % 3;
00213                         randCol = rand() % 3;
00214                     }
00215                 }
00216                 stopElapseTime("Minimax Random Move");
00217             }
00218         #endif
00219         stopElapseTime("Minimax Move");
00220     }
00221     else // ML mode, sets ML as default if for some reason playermode.mode has expected value.
00222     {
00223         if (isMLAvail)
00224         {
00225             botMove = getBestPosition(iBoard, 'x');
00226         }
00227     }

```

```

00228
00229     iBoard[botMove.row][botMove.col] = BOT;
00230     gtk_button_set_label(GTK_BUTTON(btnGrid[botMove.row][botMove.col]), "X");
00231     return SUCCESS;
00232 }
00233
00234 int chkPlayerWin()
00235 {
00236     // check both dia
00237     if (iBoard[0][0] == iBoard[1][1] && iBoard[1][1] == iBoard[2][2] && iBoard[0][0] != 0)
00238     {
00239         iWinPos[0][0] = iWinPos[1][1] = iWinPos[2][2] = WIN;
00240         return WIN;
00241     }
00242
00243     if (iBoard[0][2] == iBoard[1][1] && iBoard[1][1] == iBoard[2][0] && iBoard[0][2] != 0)
00244     {
00245         iWinPos[0][2] = iWinPos[1][1] = iWinPos[2][0] = WIN;
00246         return WIN;
00247     }
00248
00249     // check rows and col
00250     for (int i = 0; i < 3; i++)
00251     {
00252         // Check rows
00253         if (iBoard[i][0] == iBoard[i][1] && iBoard[i][1] == iBoard[i][2] && iBoard[i][0] != 0)
00254         {
00255             iWinPos[i][0] = iWinPos[i][1] = iWinPos[i][2] = WIN;
00256             return WIN;
00257         }
00258         // Check columns
00259         if (iBoard[0][i] == iBoard[1][i] && iBoard[1][i] == iBoard[2][i] && iBoard[0][i] != 0)
00260         {
00261             iWinPos[0][i] = iWinPos[1][i] = iWinPos[2][i] = WIN;
00262             return WIN;
00263         }
00264     }
00265
00266     // check for unclicked grid, if none left then tie
00267     for (int i = 0; i < 3; i++)
00268     {
00269         for (int j = 0; j < 3; j++)
00270         {
00271             if (iBoard[i][j] == 0)
00272             {
00273                 return PLAY;
00274             }
00275         }
00276     }
00277
00278     return TIE;
00279 }
00280
00281 /*=====
00282 END OF LOGIC FUNCTIONS
00283 =====*/
00284
00285 /*=====
00286 MAIN
00287 *Init GUI interface and global variable/objects
00288 =====*/
00289
00302 int main(int argc, char *argv[])
00303 {
00304     int retVal = SUCCESS;
00305     srand(time(NULL));
00306
00307     retVal = initData();
00308     if (retVal != SUCCESS) // disable ML
00309     {
00310         isMLAvail = false;
00311     }
00312
00313     GtkWidget *window;
00314     GtkWidget *grid;
00315     GtkWidget *score_button;
00316
00317     // Initialize GTK
00318     gtk_init(&argc, &argv);
00319
00320     // Create a new window
00321     window = gtk_window_new(GTK_WINDOW_TOPLEVEL);
00322     gtk_window_set_title(GTK_WINDOW(window), "Tic-Tac-Toe");
00323     gtk_window_set_default_size(GTK_WINDOW(window), 250, 950);
00324     g_signal_connect(window, "destroy", G_CALLBACK(gtk_main_quit), NULL);
00325
00326     // Create a box to hold the grid and score button with padding

```

```

00327 GtkWidget *box = gtk_box_new(GTK_ORIENTATION_VERTICAL, 0);
00328 gtk_container_set_border_width(GTK_CONTAINER(box), 50); // Set padding
00329
00330 // Create a grid to hold the btnGrid
00331 grid = gtk_grid_new();
00332 gtk_box_pack_start(GTK_BOX(box), grid, TRUE, TRUE, 0); // Add grid to the box
00333
00334 // Set the grid to expand
00335 gtk_grid_set_row_homogeneous(GTK_GRID(grid), TRUE);
00336 gtk_grid_set_column_homogeneous(GTK_GRID(grid), TRUE);
00337
00338 // Set CSS styles
00339 GtkCssProvider *css_provider = gtk_css_provider_new();
00340 gtk_css_provider_load_from_data(css_provider,
00341     "window { background-color: black; }\n"
00342     "button { background-color: black; color: white; border: 3px solid
00343     white; font-size: 24px; background-image: none; }\n"
00344     "button:pressed { background-color: darkgray; }\n",
00345     -1, NULL);
00346 gtk_style_context_add_provider_for_screen(gdk_screen_get_default(),
00347     GTK_STYLE_PROVIDER(css_provider),
00348     GTK_STYLE_PROVIDER_PRIORITY_USER);
00349
00350 // Create a button for the score display
00351 score_button = gtk_button_new_with_label("");
00352 gtk_label_set_markup(GTK_LABEL(gtk_bin_get_child(GTK_BIN(score_button))), "<b>Player 1 (O): 0</b>
00353 | TIE: 0 | Player 2 (X): 0 | [2P] ");
00354 g_signal_connect(score_button, "clicked", G_CALLBACK(on_btnScore_clicked), score_button);
00355 gtk_grid_attach(GTK_GRID(grid), score_button, 0, 3, 3, 1); // Attach score button below the grid
00356
00357 // Create the 9 btnGrid and add them to the grid
00358 for (int i = 0; i < 3; i++)
00359 {
00360     for (int j = 0; j < 3; j++)
00361     {
00362         btnGrid[i][j] = gtk_button_new_with_label("");
00363
00364         BtnPos *data = g_new(BtnPos, 1); // Allocate memory for the structure
00365         data->pos[0] = i; // Store row
00366         data->pos[1] = j; // Store column
00367
00368         // Set the structure as data on the button
00369         g_object_set_data(G_OBJECT(btnGrid[i][j]), "button-data", data);
00370         g_signal_connect(btnGrid[i][j], "clicked", G_CALLBACK(on_btnGrid_clicked), score_button);
00371 // Pass score_button as data
00372         gtk_grid_attach(GTK_GRID(grid), btnGrid[i][j], j, i, 1, 1);
00373 // Attach btnGrid to the grid
00374     }
00375 }
00376 // Make the btnGrid expand to fill the available space
00377 for (int i = 0; i < 3; i++)
00378 {
00379     gtk_widget_set_vexpand(btnGrid[i][0], TRUE);
00380     gtk_widget_set_hexpand(btnGrid[i][0], TRUE);
00381 }
00382 gtk_widget_set_vexpand(score_button, TRUE);
00383 gtk_widget_set_hexpand(score_button, TRUE);
00384
00385 // Add the box to the window
00386 gtk_container_add(GTK_CONTAINER(window), box);
00387
00388 // Show everything
00389 gtk_widget_show_all(window);
00390
00391 // Start the GTK main loop
00392 gtk_main();
00393
00394 return SUCCESS;
00395 }
00396 /*
00397 =====
00398 END OF MAIN
00399 =====
00400 */

```

6.21 src/minimax.c File Reference

```
#include <minimax.h>
```

Functions

- int `max` (int a, int b)
Returns the maximum of two integers.
- int `min` (int a, int b)
Returns the minimum of two integers.
- struct `Position` `findBestMove` (int board[3][3])
Finds the best move for the bot in the Tic-Tac-Toe game.
- int `minimax` (int board[3][3], int depth, bool isMax)
Implements the Minimax algorithm to evaluate the best move for the bot.
- int `evaluate` (int b[3][3])
Evaluates the current board state to determine if there is a winner.
- bool `isMovesLeft` (int board[3][3])
Checks if there are any moves left on the board.
- int `loadBoardStates` (struct `BoardState` boardStates[])
Loads board states and their best moves from a file.
- bool `checkAndUpdateBestMove` (int board[3][3], struct `Position` *bestMove, struct `BoardState` boardStates[], int count)
Checks if the current board configuration exists in the lookup table and updates the best move.
- void `writeBestMoveToFile` (int board[3][3], struct `Position` bestMove)
Appends the current board state and the best move to a file.

Variables

- int `depthCounter` = 0

6.21.1 Function Documentation

`checkAndUpdateBestMove()`

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

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

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

Parameters

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

Returns

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

See also

[BoardState](#), [Position](#)

Definition at line 419 of file `minimax.c`.

evaluate()

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

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

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

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

Parameters

| | |
|----------|---|
| <i>b</i> | A 3x3 array representing the Tic-Tac-Toe board. |
|----------|---|

Returns

The evaluation score:

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

See also

[BOT](#), [PLAYER1](#)

Definition at line [220](#) of file [minimax.c](#).

findBestMove()

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

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

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

Parameters

| | |
|--------------|---|
| <i>board</i> | A 3x3 array representing the current Tic-Tac-Toe board. |
|--------------|---|

Returns

The best move for the bot as a struct [Position](#) containing the row and column.

See also

[minimax](#), [loadBoardStates](#), [checkAndUpdateBestMove](#), [writeBestMoveToFile](#)

Definition at line [71](#) of file [minimax.c](#).

isMovesLeft()

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

Checks if there are any moves left on the board.

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

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

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

Parameters

| | |
|--------------|--|
| <i>board</i> | <p>A 3x3 array representing the board state. Each cell should contain:</p> <ul style="list-style-type: none"> • <code>EMPTY</code> (typically 0) if the cell is empty. • Any non-zero value if the cell is occupied. |
|--------------|--|

Returns

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

Note

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

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

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

Example usage:

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

Definition at line 270 of file [minimax.c](#).

loadBoardStates()

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

Loads board states and their best moves from a file.

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

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

Parameters

| | |
|--------------------|---|
| <i>boardStates</i> | An array of BoardState structures to store the loaded board states. |
|--------------------|---|

Returns

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

See also

[BoardState](#), [FILE_BESTMOV](#)

Definition at line 366 of file [minimax.c](#).

max()

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

Returns the maximum of two integers.

This function computes the maximum of two integers using either:

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

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

Parameters

| | |
|----------|--------------------------------|
| <i>a</i> | The first integer to compare. |
| <i>b</i> | The second integer to compare. |

Returns

The greater of the two integers.

Note

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

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

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

Example usage:

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

Definition at line 5 of file [minimax.c](#).

min()

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

Returns the minimum of two integers.

This function computes the minimum of two integers using either:

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

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

Parameters

| | |
|----------|--------------------------------|
| <i>a</i> | The first integer to compare. |
| <i>b</i> | The second integer to compare. |

Returns

The smaller of the two integers.

Note

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

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

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

Example usage:

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

Definition at line 38 of file [minimax.c](#).

minimax()

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

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

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

Parameters

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

Returns

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

See also

[evaluate](#), [isMovesLeft](#), [max](#), [min](#)

Definition at line 136 of file [minimax.c](#).

writeBestMoveToFile()

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

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

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

Parameters

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

See also

[Position](#), [BoardState](#)

Definition at line 436 of file [minimax.c](#).

6.21.2 Variable Documentation

depthCounter

int depthCounter = 0

Definition at line 3 of file [minimax.c](#).

6.22 minimax.c

[Go to the documentation of this file.](#)

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

```

00075
00076     struct BoardState boardStates[MAX_BOARDS];
00077
00078 #if !(DISABLE_LOOKUP)
00079     startElapsedTime();
00080     int boardCount = loadBoardStates(boardStates);
00081     stopElapsedTime("Loading lookup table");
00082 #endif
00083
00084     bestMove.row = ERROR;
00085     bestMove.col = ERROR;
00086
00087     startElapsedTime();
00088 #if !(DISABLE_LOOKUP)
00089     if (checkAndUpdateBestMove(board, &bestMove, boardStates, boardCount))
00090     {
00091         stopElapsedTime("Find best move in lookup table");
00092         PRINT_DEBUG("Best move found in memory: Row = %d, Col = %d\n", bestMove.row, bestMove.col);
00093     }
00094     else
00095 #endif
00096     {
00097         startElapsedTime();
00098         // Traverse all cells, evaluate minimax function for
00099         // all empty cells. And return the cell with optimal
00100         // value.
00101         for (int i = 0; i < 3; i++)
00102         {
00103             for (int j = 0; j < 3; j++)
00104             {
00105                 // Check if cell is empty
00106                 if (board[i][j] == EMPTY)
00107                 {
00108                     // Make the move
00109                     board[i][j] = BOT;
00110
00111                     // compute evaluation function for this
00112                     // move.
00113                     int moveVal = minimax(board, 0, false);
00114                     PRINT_DEBUG("[DEBUG] Depth exited at -> %d\n", depthCounter);
00115                     // Undo the move
00116                     board[i][j] = EMPTY;
00117
00118                     // If the value of the current move is more than the best value, then update best
00119                     move
00120                     if (moveVal > bestVal)
00121                     {
00122                         bestMove.row = i;
00123                         bestMove.col = j;
00124                         bestVal = moveVal;
00125                     }
00126                 }
00127             }
00128             stopElapsedTime("Minimax depth search");
00129             writeBestMoveToFile(board, bestMove);
00130         }
00131
00132         depthCounter = 0;
00133         return bestMove;
00134     }
00135
00136 int minimax(int board[3][3], int depth, bool isMax)
00137 {
00138     #if DEBUG
00139         depthCounter++;
00140     #endif
00141     int score = evaluate(board);
00142     // If Maximizer has won the game return his/her
00143     // evaluated score
00144     if (score == 10)
00145         return score;
00146
00147     // If Minimizer has won the game return his/her
00148     // evaluated score
00149     if (score == -10)
00150         return score;
00151
00152     // If there are no more moves and no winner then
00153     // it is a tie
00154     if (isMovesLeft(board) == false)
00155         return 0;
00156
00157 #if !(MINIMAX_GODMODE)
00158     if (depth > 2)
00159         return 0;
00160 #endif

```

```

00161
00162 // If this maximizer's move
00163 if (isMax)
00164 {
00165     int best = -1000;
00166
00167     // Traverse all cells
00168     for (int i = 0; i < 3; i++)
00169     {
00170         for (int j = 0; j < 3; j++)
00171         {
00172             // Check if cell is empty
00173             if (board[i][j] == EMPTY)
00174             {
00175                 // Make the move
00176                 board[i][j] = BOT;
00177
00178                 // Call minimax recursively and choose
00179                 // the maximum value
00180                 best = max(best, minimax(board, depth + 1, !isMax));
00181
00182                 // Undo the move
00183                 board[i][j] = EMPTY;
00184             }
00185         }
00186     }
00187     return best;
00188 }
00189
00190 // If this minimizer's move
00191 else
00192 {
00193     int best = 1000;
00194
00195     // Traverse all cells
00196     for (int i = 0; i < 3; i++)
00197     {
00198         for (int j = 0; j < 3; j++)
00199         {
00200             // Check if cell is empty
00201             if (board[i][j] == EMPTY)
00202             {
00203                 // Make the move
00204                 board[i][j] = PLAYER1;
00205
00206                 // Call minimax recursively and choose
00207                 // the minimum value
00208                 best = min(best,
00209                     minimax(board, depth + 1, !isMax));
00210
00211                 // Undo the move
00212                 board[i][j] = EMPTY;
00213             }
00214         }
00215     }
00216     return best;
00217 }
00218 }
00219
00220 int evaluate(int b[3][3])
00221 {
00222     // Checking for Rows for X or O victory.
00223     for (int row = 0; row < 3; row++)
00224     {
00225         if (b[row][0] == b[row][1] &&
00226             b[row][1] == b[row][2])
00227         {
00228             if (b[row][0] == BOT)
00229                 return +10;
00230             else if (b[row][0] == PLAYER1)
00231                 return -10;
00232         }
00233     }
00234
00235     // Checking for Columns for X or O victory.
00236     for (int col = 0; col < 3; col++)
00237     {
00238         if (b[0][col] == b[1][col] &&
00239             b[1][col] == b[2][col])
00240         {
00241             if (b[0][col] == BOT)
00242                 return +10;
00243
00244             else if (b[0][col] == PLAYER1)
00245                 return -10;
00246         }
00247     }

```

```

00248
00249 // Checking for Diagonals for X or O victory.
00250 if (b[0][0] == b[1][1] && b[1][1] == b[2][2])
00251 {
00252     if (b[0][0] == BOT)
00253         return +10;
00254     else if (b[0][0] == PLAYER1)
00255         return -10;
00256 }
00257
00258 if (b[0][2] == b[1][1] && b[1][1] == b[2][0])
00259 {
00260     if (b[0][2] == BOT)
00261         return +10;
00262     else if (b[0][2] == PLAYER1)
00263         return -10;
00264 }
00265
00266 // Else if none of them have won then return 0
00267 return 0;
00268 }
00269
00270 bool isMovesLeft(int board[3][3])
00271 {
00272     #if (DISABLE_ASM)
00273         for (int i = 0; i<3; i++)
00274             for (int j = 0; j<3; j++)
00275                 if (board[i][j] == EMPTY)
00276                     return true;
00277         return false;
00278     #else
00279         int result;
00280         #ifdef __aarch64__
00281             __asm__(
00282                 "mov x1, #0;"           // x1 = i = 0
00283                 "outer_loop:"
00284                 "cmp x1, #3;"           // if i >= 3, go to return_false
00285                 "bge return_false;"
00286
00287                 "mov x2, #0;"           // x2 = j = 0
00288                 "inner_loop:"
00289                 "cmp x2, #3;"           // if j >= 3, increment i
00290                 "bge increment_i;"
00291
00292                 // Calculate board[i][j]
00293                 "mov x3, x1;"           // Copy i to x3
00294                 "lsl x4, x1, #3;"       // x4 = i * 8
00295                 "add x3, x4, x1, lsl #2;" // x3 = i * 8 + i * 4 = i * 12
00296                 "add x3, %1, x3;"       // x3 = board + (i * 12), points to board[i]
00297                 "ldr w0, [x3, x2, lsl #2];" // Load board[i][j] (each int is 4 bytes)
00298
00299                 "cbz w0, return_true;"  // If board[i][j] == 0, go to return_true
00300
00301                 "add x2, x2, #1;"       // Increment j
00302                 "b inner_loop;"
00303
00304                 "increment_i:"
00305                 "add x1, x1, #1;"       // Increment i
00306                 "b outer_loop;"
00307
00308                 "return_false;"
00309                 "mov %w0, #0;"         // Set result to 0 (false)
00310                 "b end;"
00311
00312                 "return_true:"
00313                 "mov %w0, #1;"         // Set result to 1 (true)
00314
00315                 "end;"
00316                 : "=r"(result)         // Output operand
00317                 : "r"(board)           // Input operand
00318                 : "x0", "x1", "x2", "x3", "x4" // Clobbered registers
00319             );
00320         #else //x86-64
00321             __asm__(
00322                 "xor %%rbx, %%rbx;"     // rbx = i = 0
00323                 "outer_loop:"
00324                 "cmp $3, %%ebx;"       // if i >= 3, return false
00325                 "jge return_false;"
00326
00327                 "xor %%rcx, %%rcx;"     // rcx = j = 0
00328                 "inner_loop:"
00329                 "cmp $3, %%ecx;"       // if j >= 3, increment i
00330                 "jge increment_i;"
00331
00332                 // Calculate board[i][j]
00333                 "mov %%rbx, %%rdx;"     // Copy i to rdx

```

```

00335     "imul $12, %%rdx, %%rdx;"           // rdx = i * 12 (calculate row offset)
00336     "add %1, %%rdx;"                   // rdx = board + (i * 12), points to board[i]
00337     "mov (%%rdx, %%rcx, 4), %%eax;"    // Load board[i][j]
00338
00339     "test %%eax, %%eax;" // Check if board[i][j] == 0
00340     "jz return_true;" // If board[i][j] == 0, return true
00341
00342     "inc %%rcx;" // Increment j
00343     "jmp inner_loop;"
00344
00345     "increment_i:;"
00346     "inc %%rbx;" // Increment i
00347     "jmp outer_loop;"
00348
00349     "return_false:;"
00350     "mov $0, %0;" // Set result to 0 (false)
00351     "jmp end;"
00352
00353     "return_true:;"
00354     "mov $1, %0;" // Set result to 1 (true)
00355
00356     "end:;"
00357     : "=r"(result)           // Output operand
00358     : "r"(board)             // Input operand
00359     : "rbx", "rcx", "rdx", "rax" // Clobbered registers
00360 );
00361 #endif
00362     return result;
00363 #endif
00364 }
00365
00366 int loadBoardStates(struct BoardState boardStates[])
00367 {
00368     FILE *file = fopen(FILE_BESTMOV, "r");
00369     if (file == NULL)
00370     {
00371         PRINT_DEBUG("%s <- File does not exist. Creating new file.\n", FILE_BESTMOV);
00372         FILE *file = fopen(FILE_BESTMOV, "w");
00373         PRINT_DEBUG("Text file created.\n");
00374         fclose(file);
00375         return 0; // No boards loaded
00376     }
00377     PRINT_DEBUG("File exist. Checking.\n");
00378     int count = 0;
00379     char line[100];
00380     while (fgets(line, sizeof(line), file) != NULL && count < MAX_BOARDS)
00381     {
00382         // Parse the line
00383         char *token = strtok(line, ",");
00384         int index = 0;
00385
00386         // Read the board condition
00387         while (token != NULL && index < 9)
00388         {
00389             if (strcmp(token, "x") == 0)
00390             {
00391                 boardStates[count].board[index / 3][index % 3] = BOT;
00392             }
00393             else if (strcmp(token, "o") == 0)
00394             {
00395                 boardStates[count].board[index / 3][index % 3] = PLAYER1;
00396             }
00397             else
00398             {
00399                 boardStates[count].board[index / 3][index % 3] = EMPTY;
00400             }
00401             token = strtok(NULL, ",");
00402             index++;
00403         }
00404
00405         // Read the best move
00406         if (token != NULL)
00407         {
00408             boardStates[count].bestMove.row = atoi(token);
00409             token = strtok(NULL, ",");
00410             boardStates[count].bestMove.col = atoi(token);
00411         }
00412         count++;
00413     }
00414
00415     fclose(file);
00416     return count; // Return the number of boards loaded
00417 }
00418
00419 bool checkAndUpdateBestMove(int board[3][3], struct Position *bestMove, struct BoardState
boardStates[], int count)
00420 {

```

```

00421     for (int i = 0; i < count; i++)
00422     {
00423         if (memcmp(board, boardStates[i].board, sizeof(boardStates[i].board)) == 0)
00424         {
00425             // Board matches, update the best move
00426             *bestMove = boardStates[i].bestMove;
00427             PRINT_DEBUG("Found position in lookup table\n");
00428             PRINT_DEBUG("Best Move = R:%d C:%d\n", bestMove->row, bestMove->col);
00429             return bestMove; // Board matches, return the best move
00430         }
00431     }
00432     PRINT_DEBUG("Position not found in lookup table\n");
00433     return false; // No matching board found
00434 }
00435
00436 void writeBestMoveToFile(int board[3][3], struct Position bestMove)
00437 {
00438     FILE *file = fopen(FILE_BESTMOV, "a"); // Open the file for appending
00439     if (file == NULL)
00440     {
00441         PRINT_DEBUG("Error opening file for writing. -> %s\n", FILE_BESTMOV);
00442         return;
00443     }
00444
00445     // Write the board state to the file
00446     for (int j = 0; j < 3; j++)
00447     {
00448         for (int k = 0; k < 3; k++)
00449         {
00450             if (board[j][k] == PLAYER1)
00451             {
00452                 fprintf(file, "o,");
00453                 PRINT_DEBUG("o,");
00454             }
00455             else if (board[j][k] == BOT)
00456             {
00457                 fprintf(file, "x,");
00458                 PRINT_DEBUG("x,");
00459             }
00460             else
00461             {
00462                 fprintf(file, "b,");
00463                 PRINT_DEBUG("b,");
00464             }
00465         }
00466     }
00467     // Write the best move to the file
00468     fprintf(file, "%d,%d\n", bestMove.row, bestMove.col);
00469     if (fprintf(file, "%d,%d\n", bestMove.row, bestMove.col) < 0)
00470     {
00471         PRINT_DEBUG("Error writing best move to file. -> %s\n", FILE_BESTMOV);
00472     }
00473     PRINT_DEBUG("\nAttempting to write best move to file: Row = %d, Col = %d\n", bestMove.row,
bestMove.col);
00474     PRINT_DEBUG("New best move written to file.\n");
00475     fclose(file);
00476 }

```

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

```

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

```

Functions

- int [assignMoveIndex](#) (char move)
Assigns an index to each move ("x", "o", or "b").
- void [calculateProbabilities](#) (int dataset_size)
Calculates the probabilities for each class and conditional probabilities with Laplace smoothing.
- int [predictOutcome](#) (struct [Dataset](#) board)
Predicts the outcome of a given Tic Tac Toe board based on previously calculated probabilities.
- struct [Position](#) [getBestPosition](#) (int grid[3][3], char player)
Determines the best position for the bot to make a move based on the highest probability.
- void [resetTrainingData](#) ()
Resets the training data and associated statistics for a fresh training cycle.

- int `initData` ()
Initializes the training data and model statistics.
- void `assignCMValue` (int `actual`, int `predicted`)
Updates the confusion matrix based on actual and predicted outcomes.
- void `calcConfusionMatrix` ()
Calculates the confusion matrix and error probability for the testing dataset.
- int `getTruthValue` (char *`str1`)
Returns an integer value representing a truth value based on input.
- void `calcTrainErrors` ()
Calculates the training errors and the probability of error.
- void `debugDataset` (struct `Dataset` *`data`, int `len`)
Debug function to display dataset contents.

Variables

- int `positive_count` = 0
Counter for the number of positive outcomes in the training dataset.
- int `negative_count` = 0
Counter for the number of negative outcomes in the training dataset.
- int `cM` [4] = {0, 0, 0, 0}
Confusion matrix for evaluating model performance.
- int `positiveMoveCount` [3][3][3]
3D array to count occurrences of each move for positive outcomes.
- int `negativeMoveCount` [3][3][3]
3D array to count occurrences of each move for negative outcomes.
- int `test_PredictedErrors` = 0
Counter for the number of errors in the testing dataset predictions.
- int `train_PredictedErrors` = 0
Counter for the number of errors in the training dataset predictions.
- int `predicted`
The predicted outcome for the current dataset (1 for positive, 0 for negative).
- int `actual`
The actual outcome for the current dataset (1 for positive, 0 for negative).
- double `positiveClassProbability`
Probability of a positive outcome in the dataset.
- double `negativeClassProbability`
Probability of a negative outcome in the dataset.
- double `probabilityErrors`
Probability of error in the predictions, calculated from the testing dataset.

6.23.1 Function Documentation

`assignCMValue()`

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

Updates the confusion matrix based on actual and predicted outcomes.

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

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

Parameters

| | |
|------------------|---|
| <i>actual</i> | The actual outcome value (1 for positive, 0 for negative). |
| <i>predicted</i> | The predicted outcome value (1 for positive, 0 for negative). |

See also

[cM](#), [ERROR](#)Definition at line 340 of file [ml-naive-bayes.c](#).**assignMoveIndex()**

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

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

This function maps the board move characters to their corresponding integer values:

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

Parameters

| | |
|-------------|---|
| <i>move</i> | The character representing the move ('x', 'o', or 'b'). |
|-------------|---|

Returns

int The integer corresponding to the move:

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

See also

[BOT](#), [PLAYER1](#), [EMPTY](#), [ERROR](#)Definition at line 21 of file [ml-naive-bayes.c](#).**calcConfusionMatrix()**

```
void calcConfusionMatrix ()
```

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

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

See also

[cM](#), [test_PredictedErrors](#), [probabilityErrors](#), [getTruthValue](#), [predictOutcome](#)Definition at line 373 of file [ml-naive-bayes.c](#).

calcTrainErrors()

```
void calcTrainErrors ()
```

Calculates the training errors and the probability of error.

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

See also

[train_PredictedErrors](#), [probabilityErrors](#), [getTruthValue](#), [predictOutcome](#)

Definition at line 424 of file [ml-naive-bayes.c](#).

calculateProbabilities()

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

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

This function calculates:

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

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

Parameters

| | |
|---------------------|--|
| <i>dataset_size</i> | The total number of samples in the dataset used for probability calculation. |
|---------------------|--|

See also

[positive_count](#), [negative_count](#), [positiveMoveCount](#), [negativeMoveCount](#)

Definition at line 36 of file [ml-naive-bayes.c](#).

debugDataset()

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

Debug function to display dataset contents.

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

Parameters

| | |
|-------------|--|
| <i>data</i> | Pointer to the dataset to be printed. |
| <i>len</i> | The length of the dataset (number of entries). |

See also

[PRINT_DEBUG](#)

Definition at line 451 of file [ml-naive-bayes.c](#).

getBestPosition()

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

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

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

Parameters

| | |
|---------------|--|
| <i>grid</i> | The current state of the Tic Tac Toe game board. |
| <i>player</i> | The current player, either 'x' or 'o'. |

Returns

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

See also

[positive_count](#), [negative_count](#), [positiveMoveCount](#), [negativeMoveCount](#)

Definition at line 156 of file [ml-naive-bayes.c](#).

getTruthValue()

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

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

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

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

Parameters

| | |
|-------------|--|
| <i>str1</i> | A pointer to the input string to be evaluated. |
|-------------|--|

Returns

int The corresponding truth value:

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

See also

[PRINT_DEBUG](#)

Definition at line 406 of file [ml-naive-bayes.c](#).

initData()

```
int initData ()
```

Initializes the training data and model statistics.

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

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

See also

[resetTrainingData](#), [getTrainingData](#), [calcTrainErrors](#), [calcConfusionMatrix](#)

Definition at line 273 of file [ml-naive-bayes.c](#).

predictOutcome()

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

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

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

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

Parameters

| | |
|--------------|--|
| <i>board</i> | The current Tic Tac Toe board whose outcome needs to be predicted. |
|--------------|--|

Returns

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

See also

[positiveClassProbability](#), [negativeClassProbability](#), [positiveMoveCount](#), [negativeMoveCount](#), [assignMoveIndex](#)

Definition at line 88 of file [ml-naive-bayes.c](#).

resetTrainingData()

```
void resetTrainingData ()
```

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

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

See also

[positive_count](#), [negative_count](#), [positiveMoveCount](#), [negativeMoveCount](#), [cM](#), [test_PredictedErrors](#), [train_PredictedErrors](#)

Definition at line 247 of file [ml-naive-bayes.c](#).

6.23.2 Variable Documentation**actual**

```
int actual
```

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

Definition at line 15 of file [ml-naive-bayes.c](#).

cM

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

Confusion matrix for evaluating model performance.

Definition at line 7 of file [ml-naive-bayes.c](#).

negative_count

```
int negative_count = 0
```

Counter for the number of negative outcomes in the training dataset.

Definition at line 5 of file [ml-naive-bayes.c](#).

negativeClassProbability

```
double negativeClassProbability
```

Probability of a negative outcome in the dataset.

Definition at line 18 of file [ml-naive-bayes.c](#).

negativeMoveCount

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

3D array to count occurrences of each move for negative outcomes.

Definition at line 9 of file [ml-naive-bayes.c](#).

positive_count

```
int positive_count = 0
```

Counter for the number of positive outcomes in the training dataset.

Definition at line 4 of file [ml-naive-bayes.c](#).

positiveClassProbability

```
double positiveClassProbability
```

Probability of a positive outcome in the dataset.

Definition at line 17 of file [ml-naive-bayes.c](#).

positiveMoveCount

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

3D array to count occurrences of each move for positive outcomes.

Definition at line 8 of file [ml-naive-bayes.c](#).

predicted

```
int predicted
```

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

Definition at line 14 of file [ml-naive-bayes.c](#).

probabilityErrors

```
double probabilityErrors
```

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

Definition at line 19 of file [ml-naive-bayes.c](#).

test_PredictedErrors

```
int test_PredictedErrors = 0
```

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

Definition at line 11 of file [ml-naive-bayes.c](#).

train_PredictedErrors

```
int train_PredictedErrors = 0
```

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

Definition at line 12 of file [ml-naive-bayes.c](#).

6.24 ml-naive-bayes.c

[Go to the documentation of this file.](#)

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

```

```

00072         }
00073         else if (negative_count == 0)
00074         {
00075             PRINT_DEBUG("Probability of %c (positive) at grid(%d,%d): %lf\n", move, row, col,
positiveProbability);
00076             PRINT_DEBUG("Probability of %c (negative) at grid(%d,%d): No negative outcomes\n",
move, row, col);
00077         }
00078         else
00079         {
00080             PRINT_DEBUG("Probability of %c (positive) at grid(%d,%d): %lf\n", move, row, col,
positiveProbability);
00081             PRINT_DEBUG("Probability of %c (negative) at grid(%d,%d): %lf\n", move, row, col,
negativeProbability);
00082         }
00083     }
00084 }
00085 }
00086 }
00087
00088 int predictOutcome(struct Dataset board)
00089 {
00090     double positiveProbability = positiveClassProbability;
00091     double negativeProbability = negativeClassProbability;
00092
00093     // required as 0*anything = 0
00094     if (positiveProbability == 0)
00095     {
00096         positiveProbability = 1;
00097     }
00098     if (negativeProbability == 0)
00099     {
00100         negativeProbability = 1;
00101     }
00102
00103     //loops through board grid and sums up probability for each grid
00104     for (int row = 0; row < 3; row++)
00105     {
00106         for (int col = 0; col < 3; col++)
00107         {
00108             int moveIndex = assignMoveIndex(board.grid[row][col]);
00109             if (moveIndex != -1)
00110             {
00111                 // PRINT_DEBUG("\nPC_%d, NC_%d, pMC_%d,
nMC_%d",positive_count,negative_count,positiveMoveCount[row][col][moveIndex],negativeMoveCount[row][col][moveIndex]);
00112                 if (positive_count > 0)
00113                 {
00114                     positiveProbability *= (double)positiveMoveCount[row][col][moveIndex] /
(double)positive_count;
00115                 }
00116                 if (negative_count > 0)
00117                 {
00118                     negativeProbability *= (double)negativeMoveCount[row][col][moveIndex] /
(double)negative_count;
00119                 }
00120             }
00121         }
00122     }
00123 }
00124
00125 // guard cases if either negativeProbability is unset
00126 if (positiveProbability == 1)
00127 {
00128     positiveProbability = 0;
00129 }
00130 if (negativeProbability == 1)
00131 {
00132     negativeProbability = 0;
00133 }
00134
00135 // Output probabilities for debugging
00136 // PRINT_DEBUG("\nPositive: %lf, Negative: %lf Probability: \n", positiveProbability,
negativeProbability);
00137
00138 //returns a value based on condition
00139 if (positiveProbability > negativeProbability)
00140 {
00141     // PRINT_DEBUG("Predicted Outcome: Positive\n");
00142     return 1;
00143 }
00144 else if (positiveProbability == 0 || negativeProbability == 0)
00145 {
00146     // PRINT_DEBUG("Unable to predict outcome based on available data.");
00147     return -1;
00148 }
00149 else
00150 {

```



```

00151         // PRINT_DEBUG("Predicted Outcome: Negative\n");
00152         return 0;
00153     }
00154 }
00155
00156 struct Position getBestPosition(int grid[3][3], char player)
00157 {
00158     // Determine whether bot is X or O depending on current player
00159     char bot = (player == 'x' ? 'o' : 'x');
00160     char bestMove = 'b';
00161     int bestRow = -1;
00162     int bestCol = -1;
00163     double highestProbability = 0.0;
00164
00165     int bot_count;
00166     int (*botMoveCount)[3][3];
00167
00168     // Use positive or negative count for calculating probability depending on whether bot is X or O
00169     // 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
00170     if (bot == 'x')
00171     {
00172         bot_count = positive_count;
00173         botMoveCount = positiveMoveCount;
00174     }
00175     else
00176     {
00177         bot_count = negative_count;
00178         botMoveCount = negativeMoveCount;
00179     }
00180
00181     for (int row = 0; row < 3; row++)
00182     {
00183         for (int col = 0; col < 3; col++)
00184         {
00185             // If the grid position is empty
00186             if (grid[row][col] != EMPTY)
00187             {
00188                 continue;
00189             }
00190
00191             // Calculate probability for X or O to determine best move for bot
00192             for (int moveIndex = 0; moveIndex < 2; moveIndex++)
00193             {
00194                 double moveProbability;
00195
00196                 if (bot == 'x')
00197                 {
00198                     // Calculate probability for move 'x'
00199                     if (bot_count > 0)
00200                     {
00201                         moveProbability = (double)botMoveCount[row][col][0] / bot_count;
00202                     }
00203                     else
00204                     {
00205                         moveProbability = 0.0;
00206                     }
00207                 }
00208                 else
00209                 {
00210                     // Calculate probability for move 'o'
00211                     if (bot_count > 0)
00212                     {
00213                         moveProbability = (double)botMoveCount[row][col][1] / bot_count;
00214                     }
00215                     else
00216                     {
00217                         moveProbability = 0.0;
00218                     }
00219                 }
00220
00221                 // Update best move and position for bot if it has higher probability
00222                 if (moveProbability > highestProbability)
00223                 {
00224                     highestProbability = moveProbability;
00225                     bestMove = bot;
00226                     bestRow = row;
00227                     bestCol = col;
00228                 }
00229             }
00230         }
00231     }
00232
00233     // Return best position
00234     if (bestRow != ERROR && bestCol != ERROR)
00235     {
00236         grid[bestRow][bestCol] = bestMove;

```

```

00237     PRINT_DEBUG("Best move: %c at grid (%d, %d) with probability: %lf\n", bestMove, bestRow,
bestCol, highestProbability);
00238     return (struct Position){bestRow, bestCol};
00239 }
00240 else
00241 {
00242     PRINT_DEBUG("\nNo valid move found.\n");
00243     return (struct Position){ERROR, ERROR}; // Indicate no valid move found
00244 }
00245 }
00246
00247 void resetTrainingData() {
00248     // Reset outcome counts
00249     positive_count = 0;
00250     negative_count = 0;
00251
00252     // Reset move count arrays for each grid position
00253     for (int row = 0; row < 3; row++) {
00254         for (int col = 0; col < 3; col++) {
00255             for (int moveIndex = 0; moveIndex < 3; moveIndex++) {
00256                 positiveMoveCount[row][col][moveIndex] = 0;
00257                 negativeMoveCount[row][col][moveIndex] = 0;
00258             }
00259         }
00260     }
00261
00262     // Reset the confusion matrix counters
00263     cM[0] = 0; // True positive
00264     cM[1] = 0; // False negative
00265     cM[2] = 0; // False positive
00266     cM[3] = 0; // True negative
00267
00268     // Reset prediction errors
00269     test_PredictedErrors = 0;
00270     train_PredictedErrors = 0;
00271 }
00272
00273 int initData()
00274 {
00275     resetTrainingData();
00276     int retVal = SUCCESS;
00277
00278     doGetTrainingData:
00279     static bool doOnce = false;
00280     struct Dataset *trainingData = NULL; // Initialize pointer
00281     int len = getTrainingData(&trainingData); // Pass address of pointer
00282
00283     if (len <= 0)
00284     {
00285         retVal = readDataset(RES_PATH " " DATA_PATH, true);
00286         if (retVal != SUCCESS)
00287         {
00288             return retVal;
00289         }
00290
00291         if (doOnce) //prevents potential loopback/deadlock. Edge case tbh.
00292         {
00293             return BAD_PARAM;
00294         }
00295
00296         doOnce = true;
00297         goto doGetTrainingData; //loops until training data is set
00298     }
00299
00300     //loops through train dataset for ml training
00301     for (int i = 0; i < len; i++)
00302     {
00303         // Get outcome class count for each position
00304         if (strcmp(trainingData[i].outcome, "positive") == 0)
00305         {
00306             positive_count++;
00307             for (int row = 0; row < 3; row++)
00308             {
00309                 for (int col = 0; col < 3; col++)
00310                 {
00311                     int moveIndex = assignMoveIndex(trainingData[i].grid[row][col]);
00312                     if (moveIndex != -1)
00313                     {
00314                         positiveMoveCount[row][col][moveIndex]++;
00315                     }
00316                 }
00317             }
00318         }
00319         else if (strcmp(trainingData[i].outcome, "negative") == 0)
00320         {
00321             negative_count++;
00322             for (int row = 0; row < 3; row++)

```

```

00323         {
00324             for (int col = 0; col < 3; col++)
00325             {
00326                 int moveIndex = assignMoveIndex(trainingData[i].grid[row][col]);
00327                 if (moveIndex != -1)
00328                 {
00329                     negativeMoveCount[row][col][moveIndex]++;
00330                 }
00331             }
00332         }
00333     }
00334 }
00335 calcTrainErrors();
00336 calcConfusionMatrix();
00337 return SUCCESS;
00338 }
00339
00340 void assignCMValue(int actual, int predicted)
00341 {
00342     // PRINT_DEBUG("\nactual_%i, predicted_%i\n",actual,predicted);
00343
00344     if (actual == ERROR || predicted == ERROR)
00345     {
00346         PRINT_DEBUG("ERROR either value is -1. actual: %d predicted: %d", actual, predicted);
00347     }
00348
00349     if (actual == 1)
00350     {
00351         if (predicted == 1)
00352         {
00353             cM[0] += 1; // True positive
00354         }
00355         else
00356         {
00357             cM[1] += 1; // False negative
00358         }
00359     }
00360     else
00361     {
00362         if (predicted == 1)
00363         {
00364             cM[2] += 1; // False positive
00365         }
00366         else
00367         {
00368             cM[3] += 1; // True negative
00369         }
00370     }
00371 }
00372
00373 void calcConfusionMatrix()
00374 {
00375     //Tests ml on test dataset and stores result in a confusion matrix
00376
00377     struct Dataset *test = NULL;
00378     int len = getTestingData(&test);
00379     // PRINT_DEBUG("Test_Data length: %d\n", len);
00380     //loops through testing dataset
00381     if (len > 0)
00382     { // Ensure len is valid before accessing test
00383         for (int i = 0; i < len; i++)
00384         {
00385             actual = getTruthValue(test[i].outcome); //converts char* to int for comparison
00386             predicted = predictOutcome(test[i]);
00387
00388             // checks and updates total errors for test dataset
00389             if (actual != predicted)
00390             {
00391                 test_PredictedErrors += 1;
00392             }
00393
00394             //sets value based on actual vs predicted
00395             assignCMValue(actual, predicted);
00396         }
00397     }
00398
00399     double i = TESTING_DATA_SIZE; // assign macro to double as you cant cast
00400     macros
00401     probabilityErrors = (1 / i) * test_PredictedErrors; // round to 2dp? not in spec though
00402     PRINT_DEBUG("For testing dataset: %d errors, %lf probability of error.\n", test_PredictedErrors,
00403     probabilityErrors);
00404     PRINT_DEBUG("TP: %d, FN: %d, FP: %d, TN: %d\n", cM[0], cM[1], cM[2], cM[3]);
00405 }
00406 int getTruthValue(char *str1) //returns an integer value based on input
00407 {

```

```

00408     if (strcmp(str1, "positive") == 0)
00409     {
00410         return 1;
00411     }
00412     else if (strcmp(str1, "negative") == 0)
00413     {
00414         return 0;
00415     }
00416     else
00417     {
00418         //guard case if inputs are neither "positive" nor "negative"
00419         PRINT_DEBUG("ERROR: Not truth value: %p", str1);
00420         return -1;
00421     }
00422 }
00423
00424 void calcTrainErrors()
00425 {
00426     struct Dataset *train = NULL;        // Initialize pointer
00427     int len = getTrainingData(&train); // Pass address of pointer
00428     // debugDataset(test, len);
00429
00430     if (len > 0)
00431     { // Ensure len is valid before accessing test
00432         for (int i = 0; i < len; i++)
00433         {
00434             predicted = predictOutcome(train[i]);
00435             actual = getTruthValue(train[i].outcome);
00436             // PRINT_DEBUG("Actual dataset outcome: %s, Dataset outcome: %d, Predicted outcome: %d\n",
test[i].outcome, actual, predicted);
00437             // checks and updates total errors for train dataset
00438             if (actual != predicted)
00439             {
00440                 train_PredictedErrors += 1;
00441             }
00442         }
00443     }
00444
00445     double i = TRAINING_DATA_SIZE;          // assign macro to double var as macros cant
be cast
00446     probabilityErrors = (1 / i) * train_PredictedErrors; // round to 2dp? not in spec though
00447
00448     PRINT_DEBUG("\nFor training dataset: %d errors, %lf probability of error.\n",
train_PredictedErrors, probabilityErrors);
00449 }
00450
00451 void debugDataset(struct Dataset *data, int len)
00452 {
00453     PRINT_DEBUG("%d\n", len);
00454     if (len > 0)
00455     { // Ensure len is valid before accessing test
00456         for (int i = 0; i < len; i++)
00457         {
00458             PRINT_DEBUG("%d ", i);
00459             for (int j = 0; j < 3; j++)
00460             {
00461                 for (int k = 0; k < 3; k++)
00462                 {
00463                     PRINT_DEBUG("%c, ", data->grid[j][k]);
00464                 }
00465             }
00466             PRINT_DEBUG("%s\n", data->outcome);
00467         }
00468     }
00469 }

```


Index

- actual
 - ml-naive-bayes.c, [66](#)
- assignCMValue
 - ml-naive-bayes.c, [62](#)
 - ml-naive-bayes.h, [30](#)
- assignMoveIndex
 - ml-naive-bayes.c, [63](#)
 - ml-naive-bayes.h, [30](#)
- BAD_PARAM
 - macros.h, [15](#)
- bestMove
 - BoardState, [7](#)
- board
 - BoardState, [7](#)
- BoardState, [7](#)
 - bestMove, [7](#)
 - board, [7](#)
- BOT
 - macros.h, [15](#)
- btnGrid
 - main.c, [44](#)
- BtnPos, [7](#)
 - pos, [8](#)
- calcConfusionMatrix
 - ml-naive-bayes.c, [63](#)
 - ml-naive-bayes.h, [31](#)
- calcTrainErrors
 - ml-naive-bayes.c, [63](#)
 - ml-naive-bayes.h, [31](#)
- calculateProbabilities
 - ml-naive-bayes.c, [64](#)
 - ml-naive-bayes.h, [31](#)
- checkAndUpdateBestMove
 - minimax.c, [51](#)
 - minimax.h, [23](#)
- chkPlayerWin
 - main.c, [41](#)
 - main.h, [19](#)
- CLASSES
 - macros.h, [15](#)
 - ml-naive-bayes.h, [30](#)
- clearGrid
 - main.c, [42](#)
 - main.h, [19](#)
- cM
 - ml-naive-bayes.c, [66](#)
- col
 - Position, [9](#)
- data
 - importData.c, [38](#)
- DATA_PATH
 - importData.h, [12](#)
- DATA_SIZE
 - macros.h, [15](#)
- Dataset, [8](#)
 - grid, [8](#)
 - outcome, [8](#)
- DEBUG
 - macros.h, [16](#)
- debugDataset
 - ml-naive-bayes.c, [64](#)
 - ml-naive-bayes.h, [31](#)
- depthCounter
 - minimax.c, [56](#)
- DISABLE_ASM
 - macros.h, [16](#)
- DISABLE_ELAPSED
 - macros.h, [16](#)
- DISABLE_LOOKUP
 - macros.h, [16](#)
- doBOTmove
 - main.c, [42](#)
 - main.h, [19](#)
- Documentation/README.md, [9](#)
- Doxygen Documentation Generation and Viewing Instructions, [5](#)
- elapsedTime.c
 - gEndTime, [35](#)
 - gStartTime, [35](#)
 - gTime, [35](#)
 - startElapseTime, [35](#)
 - stopElapseTime, [35](#)
- elapsedTime.h
 - startElapseTime, [10](#)
 - stopElapseTime, [10](#)
- EMPTY
 - macros.h, [16](#)
- ERROR
 - macros.h, [16](#)
- evaluate
 - minimax.c, [51](#)
 - minimax.h, [24](#)
- FILE_BESTMOV
 - minimax.h, [23](#)
- findBestMove
 - minimax.c, [52](#)
 - minimax.h, [24](#)
- gEndTime
 - elapsedTime.c, [35](#)
- getBestPosition
 - ml-naive-bayes.c, [64](#)
 - ml-naive-bayes.h, [32](#)
- getRandomNo
 - importData.c, [36](#)
 - importData.h, [12](#)
- getTestingData

- importData.c, 36
- importData.h, 12
- getTrainingData
 - importData.c, 37
 - importData.h, 13
- getTruthValue
 - ml-naive-bayes.c, 65
 - ml-naive-bayes.h, 32
- grid
 - Dataset, 8
- gStartTime
 - elapsedTime.c, 35
- gTime
 - elapsedTime.c, 35
- header/elapsedTime.h, 9, 11
- header/importData.h, 11, 14
- header/macros.h, 14, 17
- header/main.h, 18, 22
- header/minimax.h, 22, 28
- header/ml-naive-bayes.h, 28, 34
- iBoard
 - main.c, 44
- iGameState
 - main.c, 45
- importData.c
 - data, 38
 - getRandomNo, 36
 - getTestingData, 36
 - getTrainingData, 37
 - len_test, 38
 - len_train, 38
 - randomNo, 38
 - readDataset, 37
 - splitFile, 38
 - testingFile, 38
 - trainingFile, 38
- importData.h
 - DATA_PATH, 12
 - getRandomNo, 12
 - getTestingData, 12
 - getTrainingData, 13
 - readDataset, 13
 - RES_PATH, 12
 - splitFile, 13
 - TEST_PATH, 12
 - TRAIN_PATH, 12
- initData
 - ml-naive-bayes.c, 65
 - ml-naive-bayes.h, 33
- iPlayer1_score
 - main.c, 45
- iPlayer2_score
 - main.c, 45
- isMLAvail
 - main.c, 45
- isMovesLeft
 - minimax.c, 52
- minimax.h, 25
- isPlayer1Turn
 - main.c, 45
- iTie_score
 - main.c, 45
- iWinPos
 - main.c, 45
- len_test
 - importData.c, 38
- len_train
 - importData.c, 38
- loadBoardStates
 - minimax.c, 53
 - minimax.h, 25
- macros.h
 - BAD_PARAM, 15
 - BOT, 15
 - CLASSES, 15
 - DATA_SIZE, 15
 - DEBUG, 16
 - DISABLE_ASM, 16
 - DISABLE_ELAPSED, 16
 - DISABLE_LOOKUP, 16
 - EMPTY, 16
 - ERROR, 16
 - MINIMAX_GODMODE, 16
 - MODE_2P, 16
 - MODE_ML, 16
 - MODE_MM, 16
 - PLAY, 17
 - PLAYER1, 17
 - PRINT_DEBUG, 17
 - SUCCESS, 17
 - TIE, 17
 - WIN, 17
- main
 - main.c, 42
- main.c
 - btnGrid, 44
 - chkPlayerWin, 41
 - clearGrid, 42
 - doBOTmove, 42
 - iBoard, 44
 - iGameState, 45
 - iPlayer1_score, 45
 - iPlayer2_score, 45
 - isMLAvail, 45
 - isPlayer1Turn, 45
 - ITie_score, 45
 - iWinPos, 45
 - main, 42
 - on_btnGrid_clicked, 43
 - on_btnScore_clicked, 43
 - playerMode, 45
 - showWin, 44
 - updateScoreBtn, 44
- main.h

- chkPlayerWin, 19
- clearGrid, 19
- doBOTmove, 19
- on_btnGrid_clicked, 20
- on_btnScore_clicked, 20
- showWin, 21
- updateScoreBtn, 21
- mainpage.md, 34
- max
 - minimax.c, 54
 - minimax.h, 26
- MAX_BOARDS
 - minimax.h, 23
- min
 - minimax.c, 54
 - minimax.h, 26
- minimax
 - minimax.c, 55
 - minimax.h, 27
- minimax.c
 - checkAndUpdateBestMove, 51
 - depthCounter, 56
 - evaluate, 51
 - findBestMove, 52
 - isMovesLeft, 52
 - loadBoardStates, 53
 - max, 54
 - min, 54
 - minimax, 55
 - writeBestMoveToFile, 55
- minimax.h
 - checkAndUpdateBestMove, 23
 - evaluate, 24
 - FILE_BESTMOV, 23
 - findBestMove, 24
 - isMovesLeft, 25
 - loadBoardStates, 25
 - max, 26
 - MAX_BOARDS, 23
 - min, 26
 - minimax, 27
 - printFileContents, 27
 - writeBestMoveToFile, 28
- MINIMAX_GODMODE
 - macros.h, 16
- ml-naive-bayes.c
 - actual, 66
 - assignCMValue, 62
 - assignMoveIndex, 63
 - calcConfusionMatrix, 63
 - calcTrainErrors, 63
 - calculateProbabilities, 64
 - cM, 66
 - debugDataset, 64
 - getBestPosition, 64
 - getTruthValue, 65
 - initData, 65
 - negative_count, 67
 - negativeClassProbability, 67
 - negativeMoveCount, 67
 - positive_count, 67
 - positiveClassProbability, 67
 - positiveMoveCount, 67
 - predicted, 67
 - predictOutcome, 66
 - probabilityErrors, 67
 - resetTrainingData, 66
 - test_PredictedErrors, 67
 - train_PredictedErrors, 67
- ml-naive-bayes.h
 - assignCMValue, 30
 - assignMoveIndex, 30
 - calcConfusionMatrix, 31
 - calcTrainErrors, 31
 - calculateProbabilities, 31
 - CLASSES, 30
 - debugDataset, 31
 - getBestPosition, 32
 - getTruthValue, 32
 - initData, 33
 - predictOutcome, 33
 - resetTrainingData, 33
 - TESTING_DATA_SIZE, 30
 - TRAINING_DATA_SIZE, 30
- mode
 - PlayerMode, 9
- MODE_2P
 - macros.h, 16
- MODE_ML
 - macros.h, 16
- MODE_MM
 - macros.h, 16
- negative_count
 - ml-naive-bayes.c, 67
- negativeClassProbability
 - ml-naive-bayes.c, 67
- negativeMoveCount
 - ml-naive-bayes.c, 67
- on_btnGrid_clicked
 - main.c, 43
 - main.h, 20
- on_btnScore_clicked
 - main.c, 43
 - main.h, 20
- outcome
 - Dataset, 8
- PLAY
 - macros.h, 17
- PLAYER1
 - macros.h, 17
- PlayerMode, 8
 - mode, 9
 - txt, 9
- playerMode

- main.c, [45](#)
- pos
 - BtnPos, [8](#)
- Position, [9](#)
 - col, [9](#)
 - row, [9](#)
- positive_count
 - ml-naive-bayes.c, [67](#)
- positiveClassProbability
 - ml-naive-bayes.c, [67](#)
- positiveMoveCount
 - ml-naive-bayes.c, [67](#)
- predicted
 - ml-naive-bayes.c, [67](#)
- predictOutcome
 - ml-naive-bayes.c, [66](#)
 - ml-naive-bayes.h, [33](#)
- PRINT_DEBUG
 - macros.h, [17](#)
- printFileContents
 - minimax.h, [27](#)
- probabilityErrors
 - ml-naive-bayes.c, [67](#)
- randomNo
 - importData.c, [38](#)
- readDataset
 - importData.c, [37](#)
 - importData.h, [13](#)
- RES_PATH
 - importData.h, [12](#)
- resetTrainingData
 - ml-naive-bayes.c, [66](#)
 - ml-naive-bayes.h, [33](#)
- row
 - Position, [9](#)
- showWin
 - main.c, [44](#)
 - main.h, [21](#)
- splitFile
 - importData.c, [38](#)
 - importData.h, [13](#)
- src/elapsedTime.c, [34](#), [35](#)
- src/importData.c, [36](#), [39](#)
- src/main.c, [40](#), [46](#)
- src/minimax.c, [50](#), [56](#)
- src/ml-naive-bayes.c, [61](#), [68](#)
- startElapseTime
 - elapsedTime.c, [35](#)
 - elapsedTime.h, [10](#)
- stopElapseTime
 - elapsedTime.c, [35](#)
 - elapsedTime.h, [10](#)
- SUCCESS
 - macros.h, [17](#)
- TEST_PATH
 - importData.h, [12](#)
- test_PredictedErrors
 - ml-naive-bayes.c, [67](#)
- TESTING_DATA_SIZE
 - ml-naive-bayes.h, [30](#)
- testingFile
 - importData.c, [38](#)
- TicTacToe, [1](#)
- TIE
 - macros.h, [17](#)
- TRAIN_PATH
 - importData.h, [12](#)
- train_PredictedErrors
 - ml-naive-bayes.c, [67](#)
- TRAINING_DATA_SIZE
 - ml-naive-bayes.h, [30](#)
- trainingFile
 - importData.c, [38](#)
- txt
 - PlayerMode, [9](#)
- updateScoreBtn
 - main.c, [44](#)
 - main.h, [21](#)
- WIN
 - macros.h, [17](#)
- writeBestMoveToFile
 - minimax.c, [55](#)
 - minimax.h, [28](#)