**Project Title: AI-Based Fighting Bot for Street Fighter II Turbo**

**Member 1: Umar Farooq 22i1978  
Member 2: Anas Hamid 22i1951  
Member 3: Muhammad Suleman 22i1931  
Member 4: Muhammad Umer 22i1891**

**Overview**

This project involves developing an AI-powered bot for the classic game *Street Fighter II Turbo*. The bot is trained using a machine learning model that learns from labeled gameplay data to predict the next optimal action based on the in-game state. This report outlines the training script, model architecture, data preprocessing, and integration into the bot's fighting logic.

**Part 1: Model Training Script - train\_model.py**

**Objective:**

Train a machine learning model (Random Forest Classifier) to predict the best next move based on in-game features such as player coordinates and button states.

**Code Breakdown:**

**1. Importing Libraries**

import pandas as pd

from sklearn.ensemble import RandomForestClassifier

from sklearn.model\_selection import train\_test\_split

from sklearn.preprocessing import LabelEncoder

from sklearn.metrics import classification\_report, accuracy\_score

import joblib

* Imports the necessary libraries for data processing, model training, evaluation, and saving.

**2. Loading the Dataset**

df = pd.read\_csv("gameplay\_data.csv")

* Loads the gameplay data saved in a CSV file into a DataFrame.

**3. Data Cleaning**

df = df.dropna(subset=['Command'])

df = df[df['Command'].apply(lambda x: isinstance(x, str))]

* Removes rows with missing or invalid command labels.

**4. Feature and Target Selection**

X = df[['X', 'Y', 'Left', 'Right', 'Up', 'Down', 'A', 'B', 'Y\_btn', 'L', 'R']]

y = df['Command']

* Defines input features (X) and the target variable (y).

**5. Label Encoding**

label\_encoder = LabelEncoder()

y\_encoded = label\_encoder.fit\_transform(y)

* Encodes command labels as integers for model compatibility.

**6. Train/Test Split**

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y\_encoded, test\_size=0.2, random\_state=42)

* Splits data into training (80%) and testing (20%) sets.

**7. Model Training**

model = RandomForestClassifier(n\_estimators=100, random\_state=42)

model.fit(X\_train, y\_train)

* Trains a Random Forest classifier with 100 decision trees.

**8. Evaluation**

y\_pred = model.predict(X\_test)

print("Accuracy:", accuracy\_score(y\_test, y\_pred))

print(classification\_report(y\_test, y\_pred, target\_names=label\_encoder.classes\_.astype(str)))

* Evaluates model performance and prints metrics including accuracy, precision, recall, and F1 score.

**9. Saving the Model**

joblib.dump(model, "trained\_model.pkl")

joblib.dump(label\_encoder, "label\_encoder.pkl")

* Saves the trained model and label encoder to files for later use in the bot.

**Part 2: Model Integration in the Bot - fight() Function**

**Objective:**

To use the trained model during live gameplay and predict actions based on the current state of the game.

**Code:**

def fight(self, current\_game\_state, player):

self.buttn = Buttons()

if player == "1":

if self.exe\_code != 0:

self.run\_command([], current\_game\_state.player1)

player1 = current\_game\_state.player1

features = np.array([[player1.x\_coord, player1.y\_coord,

int(player1.player\_buttons.left), int(player1.player\_buttons.right),

int(player1.player\_buttons.up), int(player1.player\_buttons.down),

int(player1.player\_buttons.A), int(player1.player\_buttons.B),

int(player1.player\_buttons.Y), int(player1.player\_buttons.L),

int(player1.player\_buttons.R)]])

predicted\_label = model.predict(features)

command\_str = label\_encoder.inverse\_transform(predicted\_label)[0]

for action in command\_str.split('+'):

action = action.strip()

if hasattr(self.buttn, action):

setattr(self.buttn, action, True)

self.my\_command.player\_buttons = self.buttn

elif player == "2":

if self.exe\_code != 0:

self.run\_command([], current\_game\_state.player2)

player2 = current\_game\_state.player2

features = np.array([[player2.x\_coord, player2.y\_coord,

int(player2.player\_buttons.left), int(player2.player\_buttons.right),

int(player2.player\_buttons.up), int(player2.player\_buttons.down),

int(player2.player\_buttons.A), int(player2.player\_buttons.B),

int(player2.player\_buttons.Y), int(player2.player\_buttons.L),

int(player2.player\_buttons.R)]])

predicted\_label = model.predict(features)

command\_str = label\_encoder.inverse\_transform(predicted\_label)[0]

for action in command\_str.split('+'):

action = action.strip()

if hasattr(self.buttn, action):

setattr(self.buttn, action, True)

self.my\_command.player2\_buttons = self.buttn

return self.my\_command

**Explanation:**

* **Input:** Current game state and player identifier ("1" or "2").
* **Process:**
  + Extracts features from the selected player's game state.
  + Predicts the next action using the trained model.
  + Converts the action string into button presses.
* **Output:** Returns the constructed command object for execution.

**More Explanation:**

The provided code consists of two main parts:

1. **Training a Random Forest Classifier**: This script trains a machine learning model to predict game commands based on gameplay data.
2. **Bot Implementation**: This script defines a Bot class that uses the trained model to make decisions in a game environment, alongside a rule-based command execution system.

Below is a detailed analysis of the code, its logic, strengths, weaknesses, and recommendations for improvement.

**1. Training Script Analysis**

**Purpose**

The training script builds a Random Forest Classifier to predict game commands (e.g., button combinations like v+<, >+Y) based on player state features (position and button states).

**Logic Breakdown**

1. **Data Loading**:
   * Loads gameplay\_data.csv, which contains gameplay data with columns like X, Y (coordinates), Command (target), and button states (Left, Right, etc.).
   * Drops rows with missing or invalid Command values to ensure data quality.
2. **Feature and Target Selection**:
   * Features (X): Player coordinates (X, Y) and button states (Left, Right, Up, Down, A, B, Y\_btn, L, R).
   * Target (y): The Command column, representing the action to predict.
3. **Label Encoding**:
   * Uses LabelEncoder to convert string commands (e.g., v+<) into numerical labels for model training.
4. **Data Splitting**:
   * Splits data into 80% training and 20% testing sets using train\_test\_split with a fixed random\_state for reproducibility.
5. **Model Training**:
   * Trains a Random Forest Classifier with 100 trees (n\_estimators=100) and a fixed random\_state.
6. **Evaluation**:
   * Evaluates the model on the test set, printing accuracy and a classification report (precision, recall, F1-score per command).
7. **Model Saving**:
   * Saves the trained model and label encoder to trained\_model.pkl and label\_encoder.pkl using joblib for later use.

**Strengths**

* **Robust Model Choice**: Random Forest is well-suited for classification tasks with potentially noisy or imbalanced data, as it handles non-linear relationships and feature interactions effectively.
* **Data Cleaning**: Dropping invalid rows ensures the model trains on clean data.
* **Evaluation Metrics**: The classification report provides detailed insights into model performance across different commands.
* **Reusability**: Saving the model and encoder allows the bot to use them in real-time gameplay.

**Weaknesses**

* **Limited Feature Engineering**: The features are raw (coordinates and button states) without derived features (e.g., distance to opponent, velocity) that could improve predictions.
* **No Hyperparameter Tuning**: The Random Forest uses default parameters (n\_estimators=100). Grid search or random search could optimize performance.
* **No Cross-Validation**: The script uses a single train-test split, which may not robustly estimate model performance across different data subsets.
* **Assumption of Data Quality**: The script assumes gameplay\_data.csv is well-structured and representative, which may not always be true.

**Recommendations**

1. **Feature Engineering**:
   * Add features like the distance between players, player velocity, or opponent button states to capture more game context.
   * Normalize or scale coordinates (X, Y) if their ranges vary significantly.
2. **Cross-Validation**:
   * Implement k-fold cross-validation to ensure the model generalizes well.
3. **Hyperparameter Tuning**:
   * Use GridSearchCV to tune parameters like n\_estimators, max\_depth, or min\_samples\_split.
4. **Handle Imbalanced Data**:
   * Check if certain commands are underrepresented in the dataset and use techniques like class weighting or oversampling (e.g., SMOTE).
5. **Error Handling**:
   * Add checks for file existence or data format issues when loading gameplay\_data.csv.

**2. Bot Implementation Analysis**

**Purpose**

The Bot class implements a game-playing agent that uses the trained Random Forest model to predict commands and a rule-based system to execute predefined command sequences.

**Logic Breakdown**

1. **Initialization**:
   * Defines a fire\_code sequence (e.g., ["<","!<","v+<",...]), likely a combo or attack sequence.
   * Tracks execution state (exe\_code, remaining\_code) and initializes a Command and Buttons object.
2. **Fight Method**:
   * Takes current\_game\_state (game state) and player (1 or 2) as inputs.
   * For each player:
     + If a command sequence is active (exe\_code != 0), it continues executing it via run\_command.
     + Otherwise, it extracts features (player coordinates and button states), predicts a command using the Random Forest model, and translates the predicted command into button presses.
     + Assigns the button states to my\_command.player\_buttons (Player 1) or my\_command.player2\_buttons (Player 2).
3. **Run Command Method**:
   * Executes a sequence of commands (either from fire\_code or provided as input).
   * Maps command strings (e.g., v+<, >+Y) to button states (e.g., down=True, left=True).
   * Supports complex commands (e.g., >+^+B) and their "release" versions (e.g., !>+!^+!B).
   * Logs executed commands and button states to gameplay\_data.csv for potential retraining.
   * Advances the command sequence (remaining\_code) and resets when complete.

**Strengths**

* **Hybrid Approach**: Combines machine learning (Random Forest predictions) with rule-based logic (predefined sequences), allowing flexibility between learned and hardcoded strategies.
* **Dynamic Command Execution**: The run\_command method supports a wide range of command formats, including combinations and release actions.
* **Data Logging**: Appending gameplay data to gameplay\_data.csv enables continuous data collection for model retraining.
* **Player-Specific Logic**: Handles both Player 1 and Player 2 with mirrored logic, ensuring fairness in a two-player game.

**Weaknesses**

* **Commented-Out Logic**: The commented fight method suggests an older rule-based approach, which could cause confusion or indicate incomplete refactoring.
* **Limited Model Integration**: The model is used only when no command sequence is active, limiting its impact if sequences dominate gameplay.
* **Hardcoded Sequences**: The fire\_code sequence is static and may not adapt to dynamic game situations (e.g., opponent actions).
* **Error Handling**: The code assumes valid inputs (e.g., current\_game\_state, player\_buttons) and lacks checks for edge cases (e.g., missing attributes).
* **Command Parsing**: The run\_command method uses a long series of elif statements, which is hard to maintain and error-prone for new commands.

**Recommendations**

1. **Refactor Command Parsing**:
   * Use a dictionary or mapping to translate commands to button states, reducing elif chains and improving maintainability.
   * Example:

python

Copy

command\_map = {

"v+<": {"down": True, "left": True},

"!v+!<": {"down": False, "left": False},

*# Add other commands*

}

1. **Dynamic Sequence Selection**:
   * Instead of a static fire\_code, select sequences based on game state (e.g., opponent distance, health) or model predictions.
2. **Error Handling**:
   * Add checks for valid current\_game\_state attributes and handle missing model files.
   * Validate command strings before processing.
3. **Balance Model and Rule-Based Logic**:
   * Allow the model to influence sequence selection or interrupt hardcoded sequences based on confidence scores.
   * Example: If model prediction confidence is high, override the current sequence.
4. **Logging Improvements**:
   * Log additional context (e.g., opponent state, prediction confidence) to gameplay\_data.csv for richer training data.
5. **Clean Up Commented Code**:
   * Remove or document the commented fight method to avoid confusion and clarify the active logic.

**Overall Logic Evaluation**

**Strengths**

* **Effective Combination**: The hybrid approach (machine learning + rule-based) balances adaptability (model predictions) with reliability (hardcoded sequences).
* **Scalability**: The data logging mechanism supports iterative improvement by collecting new gameplay data.
* **Clear Structure**: The training and bot scripts are well-separated, with distinct responsibilities (model training vs. gameplay logic).

**Weaknesses**

* **Complexity in Command Execution**: The run\_command method is verbose and hard to extend for new commands.
* **Limited Contextual Awareness**: The model and bot logic rely heavily on player state without considering opponent actions or game context (e.g., health, stage).
* **Incomplete Error Handling**: Both scripts lack robust checks for edge cases, which could cause runtime errors in real-world scenarios.

**Recommendations for Improvement**

1. **Enhance Model Features**:
   * Incorporate opponent state (position, buttons) and game context (health, time) as features to improve prediction relevance.
2. **Streamline Command Logic**:
   * Refactor run\_command to use a modular, data-driven approach for command-to-button mapping.
3. **Add Robustness**:
   * Implement try-catch blocks and input validation to handle unexpected game states or file issues.
4. **Iterative Training**:
   * Periodically retrain the model using new data from gameplay\_data.csv to adapt to evolving gameplay patterns.
5. **Testing Framework**:
   * Develop unit tests for the Bot class and training pipeline to ensure reliability and catch bugs early

**Conclusion**

This setup allows for a machine-learning-driven gameplay bot capable of making context-aware decisions in real time. With enough training data and careful feature engineering, the bot can learn complex action patterns, leading to improved performance and adaptability during matches.

Future improvements may include:

* Using deep learning models for higher accuracy.
* Incorporating temporal/game history into predictions.
* Simulating human-like decision-making using recurrent networks.
* The provided code demonstrates a solid foundation for a game-playing bot that combines machine learning and rule-based strategies. The Random Forest Classifier effectively predicts commands based on player state, while the Bot class executes both predicted and predefined actions in a game environment. However, improvements in feature engineering, command parsing, error handling, and dynamic decision-making could enhance its performance and maintainability. By addressing the identified weaknesses and implementing the recommended changes, the bot could become more adaptive, robust, and effective in competitive gameplay scenarios.