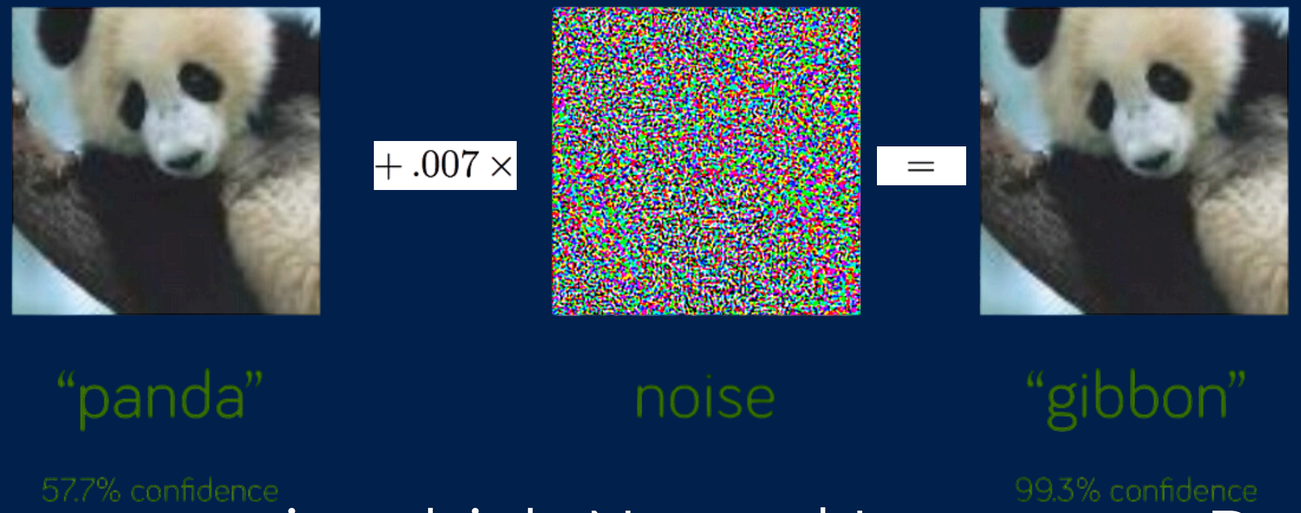
Optimizing Game Logic in Large Language Models

Federica D'Alvano Kirakidis, Lily Gao, Aaron George, Alex Huang, Niv Levy

Mentors: Prof. Benjamin Von Roy, Yifan Zhu, Henry Widjaja

Background

$$adv\_x = x + \epsilon * \text{sign}(\nabla_x J(\theta, x, y))$$



-There are many ways in which Natural Language Processing models are optimized (with adversarial attacks being one of them

-The equation above represents the Fast Gradient Sign Method (FGSM)

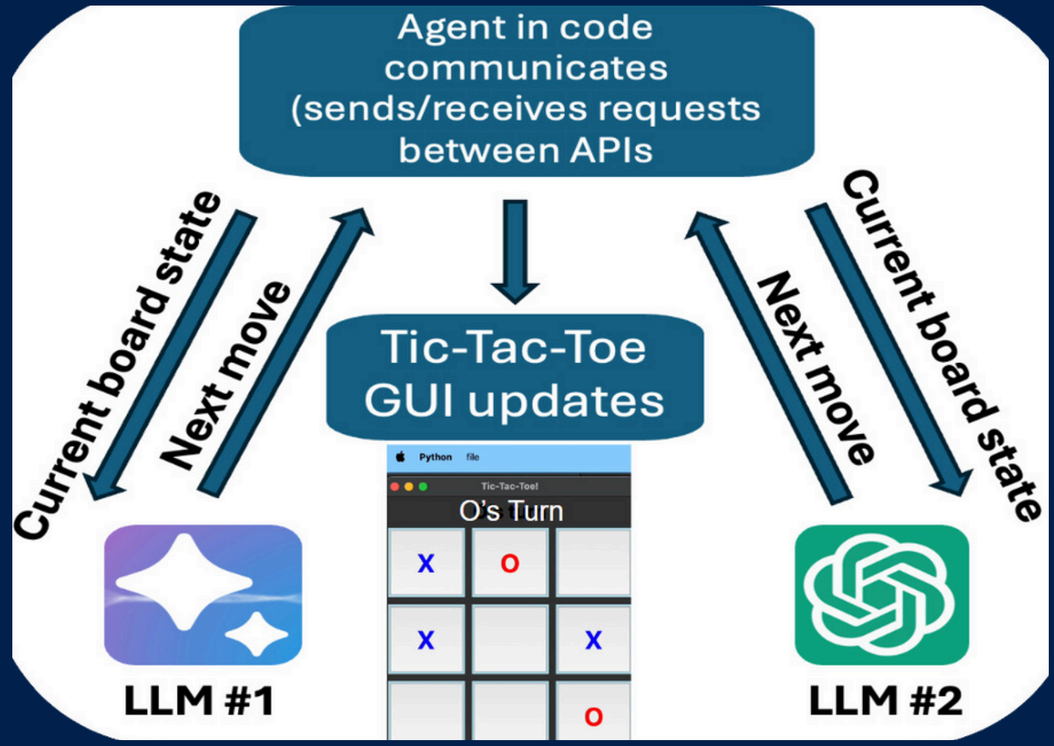
-Natural Language Processing models are able to be improved as they are fed adversarial samples so they can recognize the disparities between adversarial and original inputs

Our Approach?

-**Large Language Models can primarily only take in text inputs**, eliminating other conventional methods for optimization of NLP models

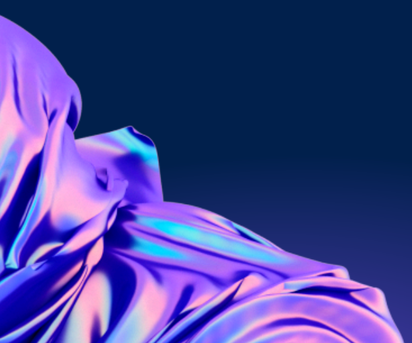
-Through prompt engineering, we **optimized an LLM model for game logic** by playing tic-tac-toe using 3 different methods

-**Tic-tac-toe is “simple” enough** to see immediate optimized results



Our Team's Goals

1. Outline Specific Game Rules
    a. 3x3 Grid, Concept of Turns, Winning Conditions
2. Assist the LLM in generating permissible & optimal moves in the framework of the game.
    a. Simulating a realistic opponent
3. Ensure the LLM is suitable for a variety of game scenarios and modify its approach as necessary.



Methods

- Coding an optimal agent that forces ties 100% of the time.
- Create a randomized near optimal agent that forced ties 57% of the time.
- Run these agents against the LLM to test prompts

```python
def minimax(self, state, player, alpha, beta):
    max_player = self.bot  # Bot is maximizing; human is minimizing
    other_player = 'O' if player == 'X' else 'X'

    # Base cases
    if self.check_winner(other_player):
        return (1, None) if other_player == max_player else (-1, None)
    elif ' ' not in state:
        return (0, None)

    if player == max_player:
        best = [-float('inf'), None]
        for possible_move in self.available_moves():
            state[possible_move] = player
            sim_score = self.minimax(state, other_player, alpha, beta)[0]
            state[possible_move] = ' '
            if sim_score > best[0]:
                best = [sim_score, possible_move]
            alpha = max(alpha, sim_score)
            if beta <= alpha:
                break
    else:
        best = [float('inf'), None]
        for possible_move in self.available_moves():
            state[possible_move] = player
            sim_score = self.minimax(state, max_player, alpha, beta)[0]
            state[possible_move] = ' '
            if sim_score < best[0]:
                best = [sim_score, possible_move]
            beta = min(beta, sim_score)
            if beta <= alpha:
                break
    return best
```

```python
def get_all_optimal_moves(self, state, player, alpha, beta):
    max_player = self.bot
    other_player = self.player if player == self.bot else self.bot
    optimal_moves = []

    if player == max_player:
        max_eval = -float('inf')
        for move in self.available_moves():
            state[move] = player
            sim_eval = self.minimax(state, other_player, alpha, beta)[0]
            state[move] = ' '
            if sim_eval > max_eval:
                max_eval = sim_eval
                optimal_moves = [move]
            elif sim_eval == max_eval:
                optimal_moves.append(move)
            alpha = max(alpha, sim_eval)
            if beta <= alpha:
                break
        return optimal_moves
    else:
        min_eval = float('inf')
        for move in self.available_moves():
            state[move] = player
            sim_eval = self.minimax(state, other_player, alpha, beta)[0]
            state[move] = ' '
            if sim_eval < min_eval:
                min_eval = sim_eval
                optimal_moves = [move]
            elif sim_eval == min_eval:
                optimal_moves.append(move)
            beta = min(beta, sim_eval)
            if beta <= alpha:
                break
        return optimal_moves
```

Initial LLM prompt

```
tic_tac_toe_explanation = '''Tic-Tac-Toe is a two-player game played on a 3 by 3 grid. The first player uses X symbols, and the
second player uses O symbols. Players take turns placing their symbols in an empty cell on the grid. The objective is to align
three of your symbols either horizontally, vertically, or diagonally. The player who first aligns three of their symbols wins the
game. Strategic placement is crucial; besides aiming to align their symbols, players must also block their opponent's potential
alignments to avoid defeat.'''
tic_tac_toe_format = '''The current state of the game is recorded in a specific format: each occupied location is delineated by a
comma (','), and for each occupied location, the row number is listed first, followed by the column number, separated by a period
('.'). If no locations are occupied by a player, 'None' is noted. Both the row and column numbers start from 0, with the top left
corner of the grid indicated by 0.0'''
tic_tac_toe_loss = '''You will lose if the second player gets a combination of one of the following: [0.0, 0.1, 0.2], [1.0, 1.1,
1.2], [2.0, 2.1, 2.2], [0.0, 1.0, 2.0], [0.1, 1.1, 2.1], [0.2, 1.2, 2.2], [0.0, 1.1, 2.2], [2.0, 1.1, 0.2]. If your opponent
could play a move to complete one of these combinations in their next turn, you must play it before them to block them.'''
tic_tac_toe_role = '''You are an adept strategic player, aiming to win the game in the fewest moves possible. You are the first
(second) player. What would be your next move?'''
tic_tac_toe_json = '''Suggest your next move in the following JSON format: {"row": RowNumber, "column": ColumnNumber}. Do not
include any additional commentary in your response. Replace RowNumber and ColumnNumber with the appropriate numbers for your
move. Both RowNumber and ColumnNumber start at 0 (top left corner is {"row": 0, "column": 0}}'). The maximum value for RowNumber
and ColumnNumber is 2, as the grid is 3 by 3.'''
tic_tac_toe_invalid = '''Please note that your move will be considered invalid if your response does not follow the specified
format, or if you provide a RowNumber or ColumnNumber that is out of the allowed range, or already occupied by a previous move.
Making more than 3 invalid moves will result in disqualification.'''

instructions = '''You are a bot playing a Tic-Tac-Toe game to the best of your ability.
    I will give you the available moves in a standard 3x3 Tic-Tac-Toe game.
    You play as X and you go first. Try to get three in a row. You can only play an available move.
    The grids are numbered 0 to 8, like a normal 3x3 tic-tac-toe game, where 0 is the top left corner and 8 is the
    bottom right corner. You must do your best to block your opponent.
    Respond with: the integer number (move) which you want to play, and then reason (walk me through) why that is the
    best move. Once you have played a move, you cannot play it again (you cannot play a number that is NOT on the
    list of available moves I send you.)
    responses must look like this sample: (number) because it is ....'''
```

Prompt Engineering Methods

5 trials per method, every trial = 10 plays

Method 1: Complexity

“Try to get three in a row **diagonally, horizontally, or vertically. Block the other player from getting three in a row horizontally, diagonally, or vertically**. **When choosing a move, you must consider either offensive strategies (trying to win) or defensive strategies (trying to block the other player). “**

Method 2: Game Data List

Game 1:
1. X played grid 4
2. O played grid 2
3. X played grid 7
4. O played grid 1 blocking player X
5. X played grid 3 failing to block O
6. O played grid 0 and won with [0.0, 0.1, 0.2]

Prompt Engineering Methods

Method 3: Complexity

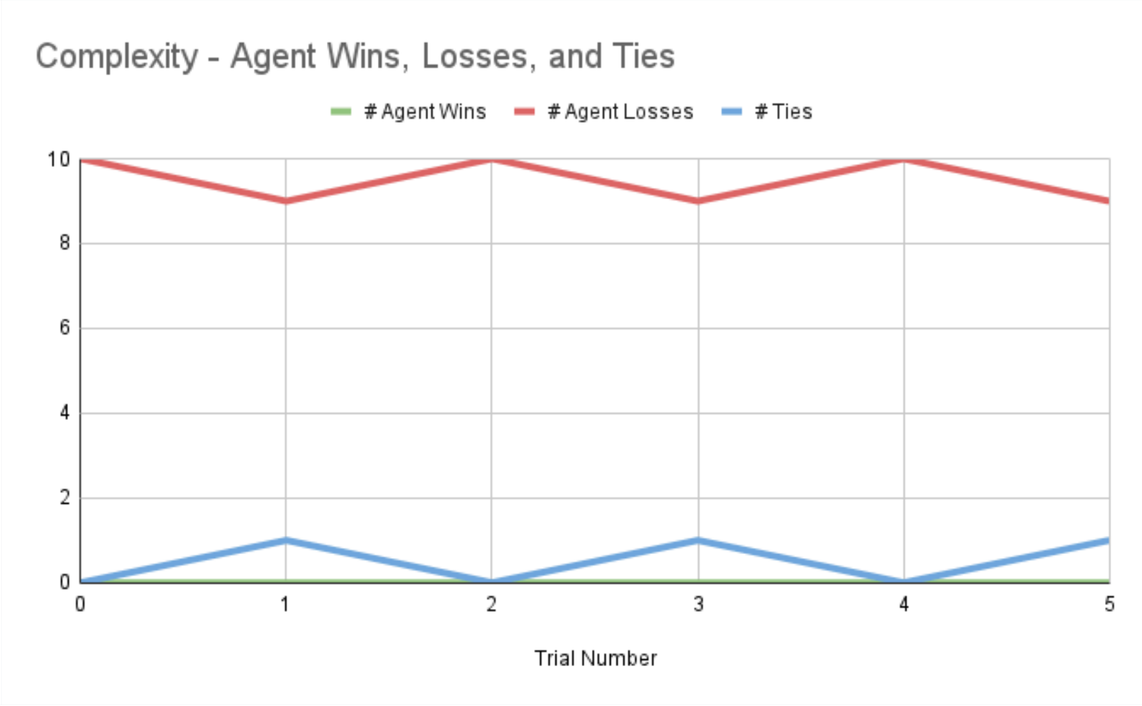
“ Game 2: You lost the game. First, you played grid 4. Your opponent played grid 0. You played grid 3. Opponent played grid 5, blocking your potential three-in-a-row. You played grid 7. Opponent played grid 1, blocking your potential three-in-a-row column. You played grid 6, opening up a line for a diagonal but failing to block your opponent from winning three-in-a-row. Your opponent played grid 2, winning a row with combination [0.0, 0.1, 0.2].”

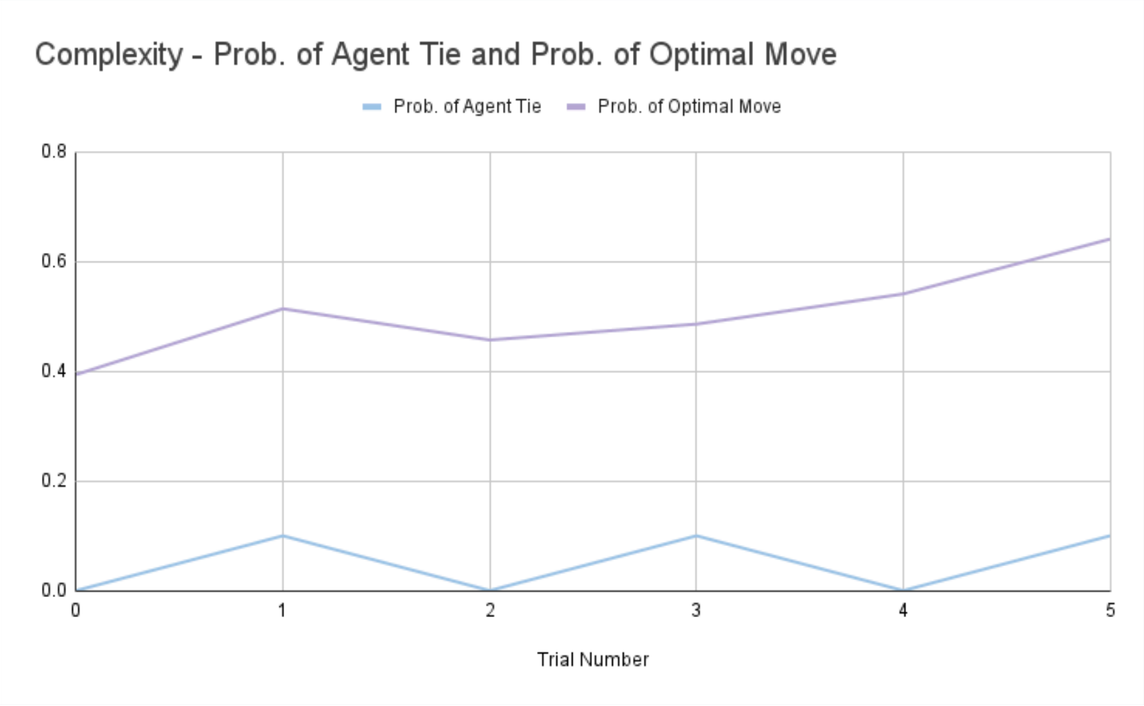
Additional testing: Game Data List vs Random near-optimal

Testing the efficiency of prompt engineering against a less predictable opponent.

Results

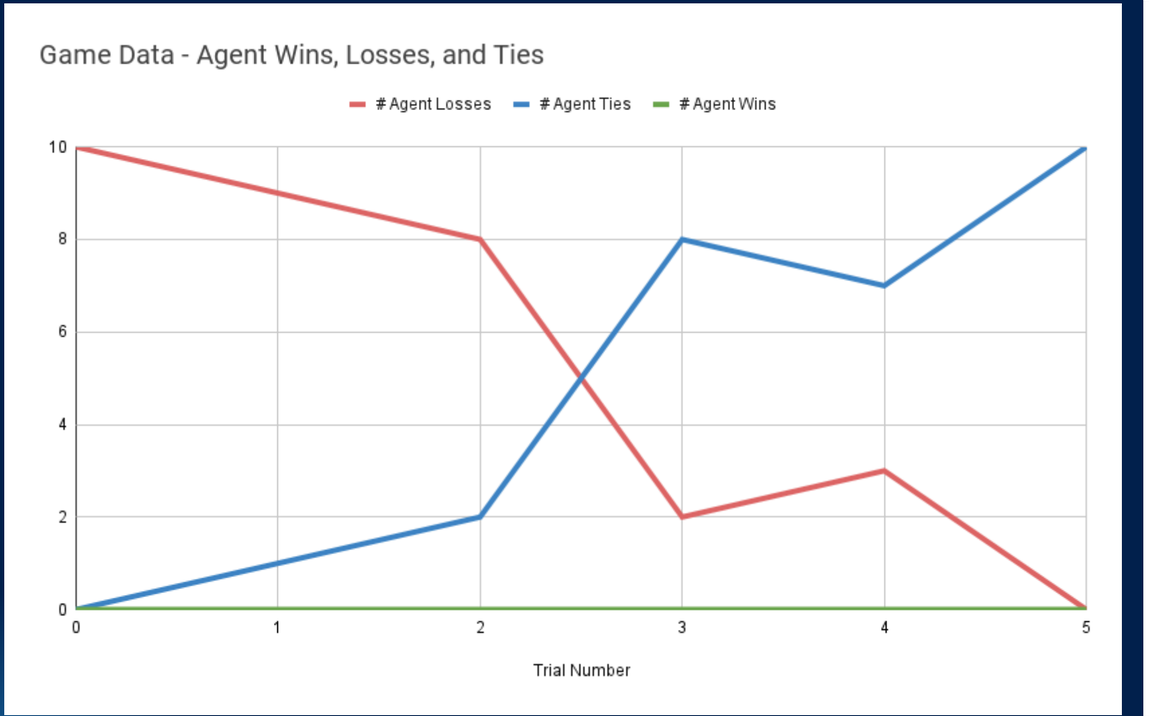
Method 1 - Added Prompt Complexity

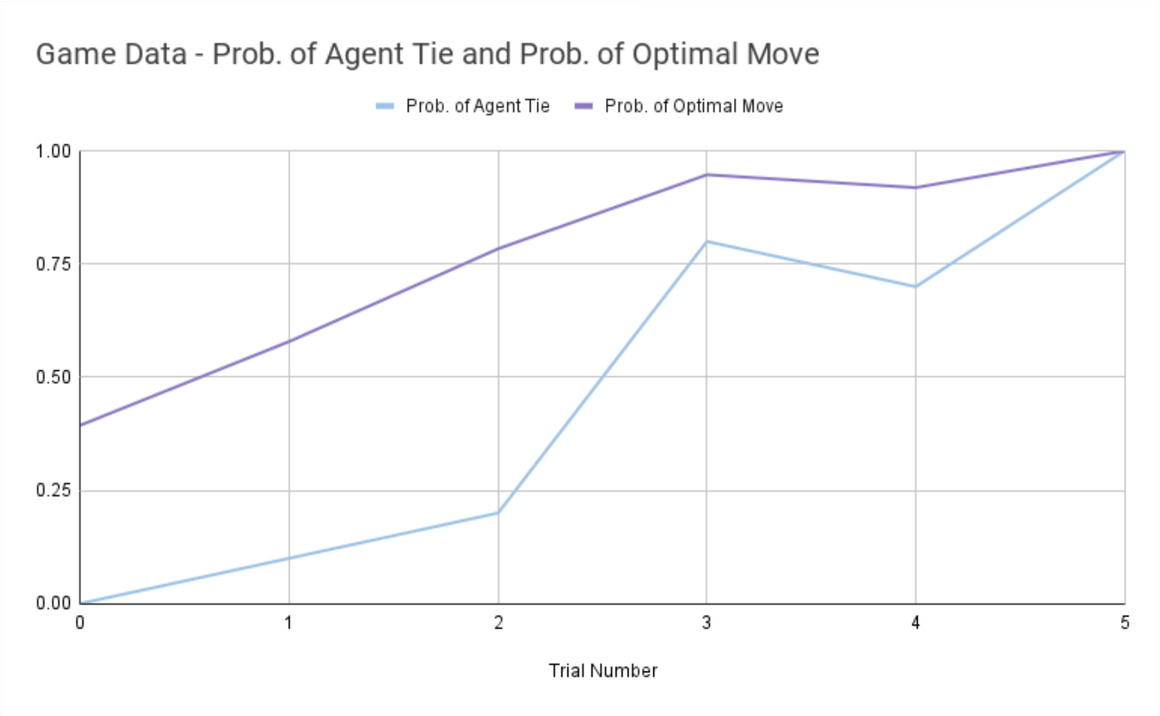




Results

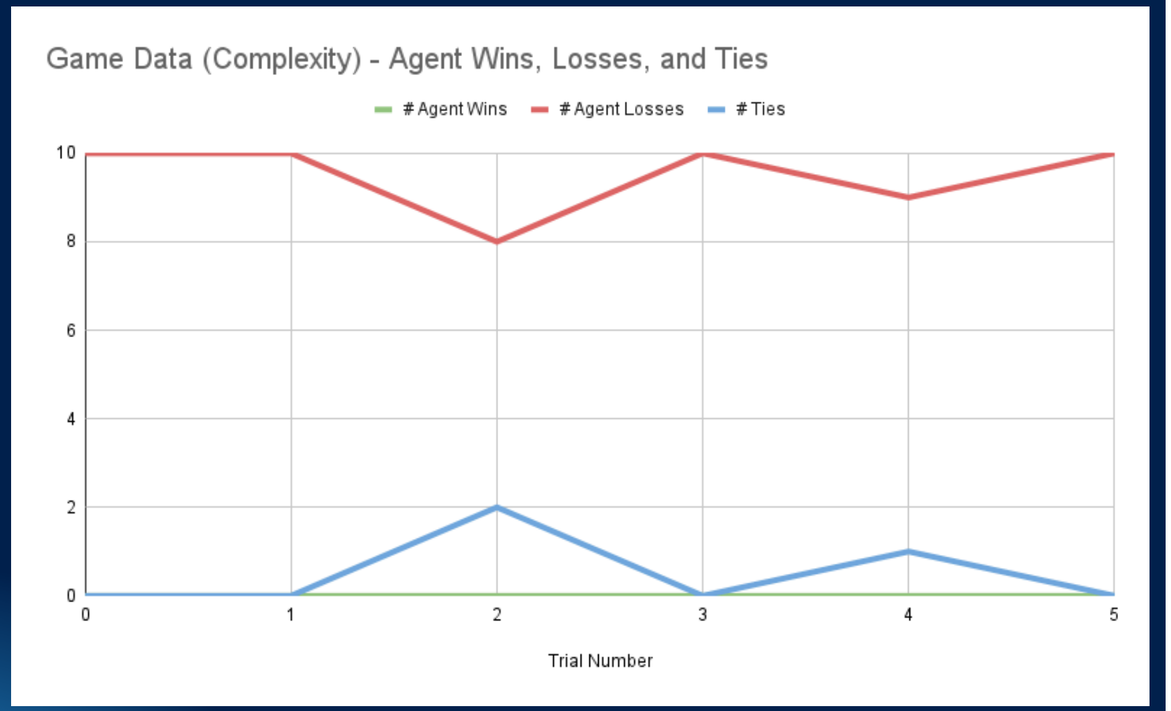
Method 2 - Game Data List

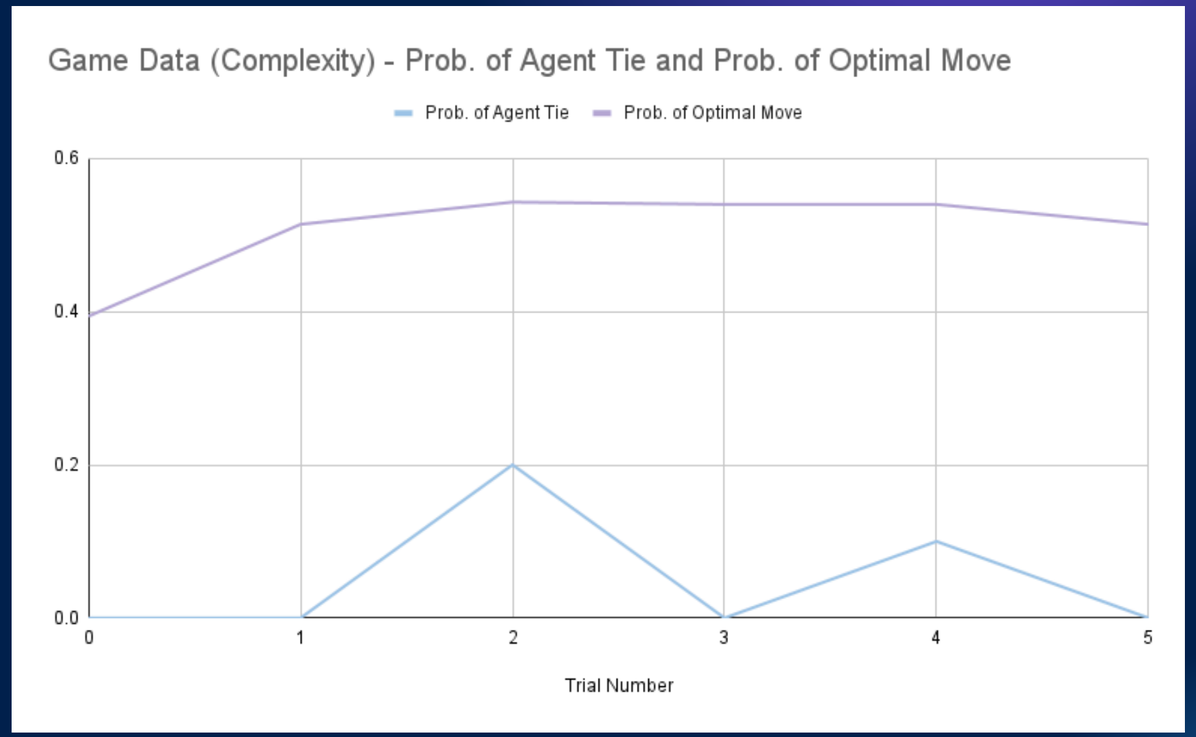




Results

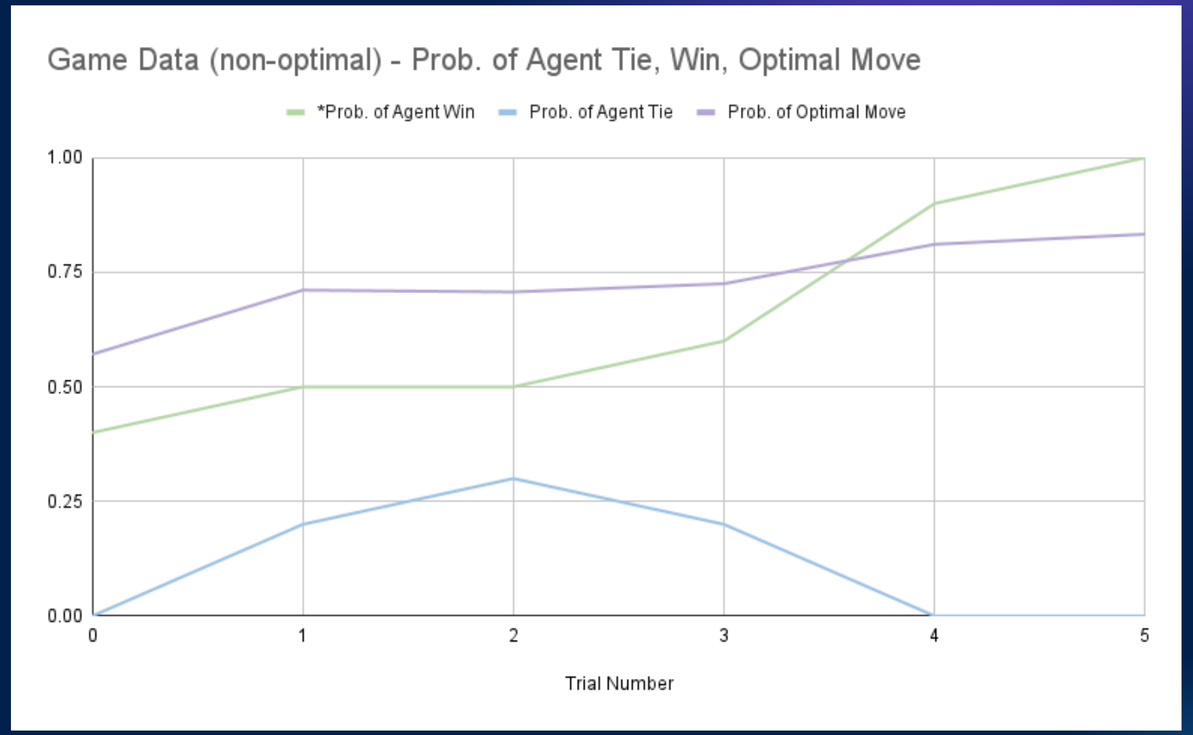
Method 3 - Game Data Complexity





Results

Additional - Game Data List (LLM vs. Near-Optimal)



Results

OPTIMAL GAMEPLAY	INC. OPTIMAL MOVES	INC. TIE RATE	INC. WIN RATE
——	METHOD 1	——	——
METHOD 2	METHOD 2	METHOD 2	——
——	METHOD 3	——	——
——	METHOD 4	——	——
ADD. TESTING	ADD. TESTING	——	ADD. TESTING

Method 2 (Game Data List) and **Additional Testing (Non-optimal)** produced the best results.

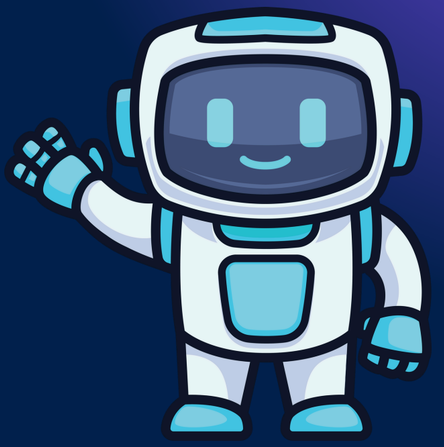
Adding sentence complexity to prompts **does not** correlate strongly with improved gameplay.

Conclusion / Applications

- **Strategic prompt engineering enhances LLM performance**
  - Initial untrained LLM = inaccurate & lacks logic (cannot think)
  - Prompted, trained LLM = improved game logic
- **As prompt engineering depth increased, so did LLM Adaptability**

- **Complex game calculations**
- **Educational tools for logic/critical thinking**
- **Assistive, real humanlike agents**
- **Simulation training**

Future Research



- **Generalization to more complex tasks:**
  - Other board games
  - Decision-making
- **Emotion / Sentiment Integration**
  - LLMs will vary their decision-making logic based on emotion & sentiment analysis
- **Assistive / ethical decision-making agents for those that need it**
  - The elderly
  - Physical/mental disabilities & illness

AI has unlimited potential; but first, we must learn to harness this power – for good.

Thank you!