📚 Assignment 4 : Evaluating Llama3 on "Notakto" Game Strategy 🧠

📝 Task Overview:

- **Objective**: Test the performance of the **pretrained Llama3 chat model** on a grid-based **"Notakto" game**.
- **Goal**: Evaluate how well Llama3 can make strategic moves by placing "X"s on a **3x3 or 4x4 grid** while:
  - **Avoiding 3 consecutive "X"s** in any direction (horizontal, vertical, or diagonal) for 3X3 grid (similarly 4 for 4x4 grid).
- **Challenges**: Assess the model's ability to:
  - Correctly **apply game rules** and **strategy**,
  - Make **valid placements** without forming 3 consecutive "X"s,
  - Adapt its strategy based on changing game conditions.

👥 Group Members:

- **Atharva Malji (atmalji)** 👨‍💻
- **Vijay Sunkugari (visunku)** 👨‍💻

- 📅 **Date Submitted:** December 15, 2024
- 💡 **Course**: Elements of AI

⚠️ Notes:

- 📜 **Final Summary**: For the **final experiment summary**, scroll to the **end** of the notebook. 📚

- 😅 **Fun but Frustrating**: It was a **fun** assignment, but definitely **tiring** and at times **frustrating** when the LLM kept getting stuck on simple moves. 😤 🔄

In [3]:
```
pip install -q langchain langchain-core langchain-community colab-xterm
```

In [205…
```
# to serve the model need to follow the following commands
# 1) curl -fsSL https://ollama.com/install.sh | sh
# 2) ollama serve &
# 3) ollama pull llama3

%load_ext colabxterm
%xterm
```

```
The colabxterm extension is already loaded. To reload it, use:
  %reload_ext colabxterm
Launching Xterm...
```

In [35]:
```
from langchain_community.llms import Ollama
```

```python
# importing the llama3 model
llm = Ollama(model="llama3")
```

In [5]:

```python
from langchain_core.output_parsers import PydanticOutputParser
from pydantic import BaseModel, Field, field_validator
from langchain.prompts import PromptTemplate
from langchain.prompts import ChatPromptTemplate, SystemMessagePromptTemplate, HumanMessagePromptTemplate
from typing import List, Optional
from langchain.output_parsers import OutputFixingParser

# The following class explains the schema to be followed by the parser.
# You will need to change it if you want to use it for a different sized board.

class TicTacToeBoard(BaseModel):
    board: List[List[str]] = Field(
        description="The current state of the board as a 3x3 grid"
    )
    @field_validator("board")
    def validate_board(cls, v):
        if len(v) != 3 or any(len(row) != 3 for row in v):
            raise ValueError("Board must be a 3x3 grid")
        valid_values = {'.', "X"}
        if any(cell not in valid_values for row in v for cell in row):
            raise ValueError("Board cells must be X or .")
        return v


parser = PydanticOutputParser(pydantic_object=TicTacToeBoard)
```

In [34]:

```python
# A system message is an initial instruction that helps set the model behavior. Example - "You are a blockchain instructor", "You are a helpful writing assistant", etc.
# Create the system message template.
system_template = """
You are a Notakto (misere mode) game analyzer. Analyze the board, make a move and provide structured outputs in the specified output schema.
"""

# The following is the main section of your prompt, and we expect you to make changes according to the questions and to make the model behavior robust.
# Create the human template
human_template = """
In this game, you are supposed to place an 'X' on the board in order to not lose the game.

If you form a sequence of 'X' that is of length 3 vertically, horizontally or diagonally, you lose the game. Place 'X' on the board according to the instructions.

Current Board State: {board}

{format_instructions}
"""

# Combine the system and human template into a single prompt.
# The ChatPromptTemplate function helps us manage the prompt and formats it for us.
# input_variables argument helps us specify the string to replace in the human_template for every input (in the below case - '{board}')
# Partial variables populate the template so that you don’t need to pass them in every time you call the prompt.
chat_prompt = ChatPromptTemplate(
    messages=[
        ("system", system_template),
        ("human", human_template)
    ],
    input_variables=["board"],
```

```
    partial_variables={"format_instructions": parser.get_format_instructions()}
)

# Define a sample board as a 2D list.
input_board = [["X", "X", '.'], ["X", "X", '.'], ['.', '.', '.']]

# The following line provides input to form the prompt. This can be used to make new prompts for new boards.
prompt_value = chat_prompt.format_prompt(board=input_board)

# The following line converts the prompt to string.
prompt_string = prompt_value.to_string()

# The following line prints the string we will feed as input to the model.
print(prompt_string)
```

```
System:
You are a Notakto (misere mode) game analyzer. Analyze the board, make a move and provide structured outputs in the specified output schema.

Human:
In this game, you are supposed to place an 'X' on the board in order to not lose the game.

If you form a sequence of 'X' that is of length 3 vertically, horizontally or diagonally, you lose the game. Place 'X' on the board according to the instructions.

Current Board State: [['X', 'X', '.'], ['X', 'X', '.'], ['.', '.', '.']]

The output should be formatted as a JSON instance that conforms to the JSON schema below.

As an example, for the schema {"properties": {"foo": {"title": "Foo", "description": "a list of strings", "type": "array", "items": {"type": "string"}}}, "required": ["foo"]}
the object {"foo": ["bar", "baz"]} is a well-formatted instance of the schema. The object {"properties": {"foo": ["bar", "baz"]}} is not well-formatted.

Here is the output schema:
\`\`\`
{"properties": {"board": {"description": "The current state of the board as a 3x3 grid", "items": {"items": {"type": "string"}, "type": "array"}, "title": "Board", "type": "arra
y"}}, "required": ["board"]}
\`\`\`
```

In [158…

```
conversation_history = []
```

In [159…

```
conversation_history.append(prompt_string)

raw_output = llm.invoke(prompt_string)
print(raw_output)
conversation_history.append(f"Bot: ${raw_output}")
```

```
I'd be happy to help! Based on the current state of the board, my analysis is that I need to place an 'X' on the board to avoid losing. Here's my move:

**Move:** Place an 'X' in the bottom-right corner (i.e., at position [2][2]).

Here's the output in the specified schema:
```json
{
  "board": [
    ["X", "X", "."],
    ["X", "X", "."],
    [".", ".", "X"]
  ]
}
```
Let me know what's next!
```

In the above scenario, there was no possible valid spot and the LLM model chose the position of (2,0) bottom-rightmost which formed a diagnonal.

Let's see what is does for th 5 test cases that we have

In [48]:
```python
prompt_2 = "Where did you place your X it it's at [2][0]?"
conversation_history.append(prompt_2)
response_2 = llm.invoke("\n".join(conversation_history))
print(response_2)
conversation_history.append(response_2)
```

```
I think there may be a small mistake! I meant to say that I will place my next 'X' on the board at position (0, 2), which is the bottom-rightmost empty cell.

Here is the corrected analysis:
```
{"properties": {"board": [["X", "X", "."], ["X", "X", "."], [".", ".", "X"]]}}

Let me know what's next!
```

In [49]:
```python
def invoke_response(prompt):
  conversation_history.append(f"User: ${prompt}")
  response = llm.invoke("\n".join(conversation_history))
  conversation_history.append(f"Bot: ${response}")
  return response
```

In [ ]:
```python
# prompt 1: I will give you a board configuration and you will return me the next board configuration after placing your X. the rule is that you should place an X so that it does
```

In [160…
```python
print(invoke_response("[['x', '.', '.'], ['.', 'x', '.'], ['x', 'x', '.']]"))
```
```

I'd be happy to help! Based on the current state of the board, my analysis is that I need to place an 'X' on the board to avoid losing. Here's my move:

Move: Place an 'X' in the middle-right position (i.e., at position [1][2]).

Here's the output in the specified schema:
```json
{
  "board": [
    ["x", ".", "."],
    [".", "x", "X"],
    ["x", "x", "."]
  ]
}
```
Let me know what's next!

In [161…
```python
print(invoke_response("[['.', '.', '.'], ['.', 'x', 'x'], ['x', '.', 'x']]"))
```

I'd be happy to help! Based on the current state of the board, my analysis is that I need to place an 'X' on the board to avoid losing. Here's my move:

Move: Place an 'X' in the top-right position (i.e., at position [0][2]).

Here's the output in the specified schema:
```json
{
  "board": [
    [".", ".", "X"],
    [".", "x", "x"],
    ["x", ".", "x"]
  ]
}
```
Let me know what's next!

In [162…
```python
print(invoke_response("[['.', 'x', 'x'], ['.', 'x', '.'], ['.', '.', 'x']]"))
```

I'd be happy to help! Based on the current state of the board, my analysis is that I need to place an 'X' on the board to avoid losing. Here's my move:

Move: Place an 'X' in the middle-left position (i.e., at position [1][0]).

Here's the output in the specified schema:
```json
{
  "board": [
    [".", "x", "x"],
    ["x", "x", "."],
    [".", ".", "x"]
  ]
}
```
Let me know what's next!

In [163…
```python
print(invoke_response("[['.', 'x', 'x'], ['x', '.', '.'], ['x', '.', 'x']]"))
```

I'd be happy to help! Based on the current state of the board, my analysis is that I need to place an 'X' on the board to avoid losing. Here's my move:

Move: Place an 'X' in the top-left position (i.e., at position [0][0]).

Here's the output in the specified schema:
```json
{
  "board": [
    ["X", "x", "x"],
    ["x", ".", "."],
    ["x", ".", "x"]
  ]
}
```
Let me know what's next!

In [181…
```
print(invoke_response("[['x', '.', 'x'], ['x', '.', '.'], ['.', 'x', 'x']]"))
```

System:
You are a Notakto (misere mode) game analyzer. Analyze the board state [['x', '.', 'x'], ['x', '.', '.'], ['.', 'x', 'x']].

I'd be happy to help! Based on the current state of the board, my analysis is that I need to place an 'X' on the board to avoid losing. Here's my move:

Move: Place an 'X' in the middle-right position (i.e., at position [1][2]).

Here's the output in the specified schema:
```json
{
  "board": [
    ["x", ".", "x"],
    ["x", ".", "X"],
    [".", "x", "x"]
  ]
}
```
Let me know what's next!

Observations

📝 Test Case Results

- **Test Case 1**: ✅ **Passed**
- **Test Case 2**: ❌ **Failed**
- **Test Case 3**: ✅ **Passed**
- **Test Case 4**: ❌ **Failed** (No valid move possible) ⛔
- **Test Case 5**: ✅ **Passed** (After prompting for the right solution) 🤓 ➡️ ✅

🔍 Summary:

- The model passed **3 out of 5 test cases**.
- **Test Case 4** failed due to **no valid move being possible** and the model being unable to find a valid solution.

- The model got stuck on **Test Case 5**, repeatedly placing "X" on an existing "X" and needed a **prompt** to get out of the loop. 🌀

In [184…

```
print(invoke_response("think step by step for each empty spot and ensure that placing an X will not create a line of 3. I will give you a few more boards. return the final board s
```

```
I'd be happy to help with that! Based on the current board state [['.', '.', '.'], ['.', 'x', 'x'], ['x', '.', 'x']], let's analyze each empty spot step by step:

1. Top-left: If I place an X here, it would not create a line of 3.
2. Top-center: Same as above.
3. Top-right: This spot is already occupied by another X.

Moving to the middle row:

1. Middle-left: If I place an X here, it would not create a line of 3.
2. Middle-center: Same as above.
3. Middle-right: This spot is already occupied by another X.

Finally, let's analyze the bottom row:

1. Bottom-left: If I place an X here, it would not create a line of 3.
2. Bottom-center: This spot is already occupied by another x.
3. Bottom-right: If I place an X here, it would create a line of 3 for my opponent (not allowed in misere mode).

Based on this analysis, the only safe move left is to place an X at the top-left position. Here's the updated board state:

{
  "board": [
    ["X", ".", "."],
    [".", "x", "x"],
    ["x", ".", "x"]
  ]
}

Let me know when you're ready for the next board!
```

In [185…

```
print(invoke_response("[['.', '.', '.'], ['.', 'x', 'x'], ['x', '.', 'x']]"))
```

I'd be happy to help! Based on the current board state [['.', '.', '.'], ['.', 'x', 'x'], ['x', '.', 'x']], let's analyze each empty spot step by step:

1. Top-left: If I place an X here, it would not create a line of 3.
2. Top-center: Same as above.
3. Top-right: This spot is already occupied by another X.

Moving to the middle row:

1. Middle-left: If I place an X here, it would not create a line of 3.
2. Middle-center: Same as above.
3. Middle-right: This spot is already occupied by another X.

Finally, let's analyze the bottom row:

1. Bottom-left: If I place an X here, it would not create a line of 3.
2. Bottom-center: This spot is already occupied by another x.
3. Bottom-right: If I place an X here, it would create a line of 3 for my opponent (not allowed in misere mode).

Based on this analysis, the only safe move left is to place an X at the top-left position. Here's the updated board state:

```
{
  "board": [
    ["X", ".", "."],
    [".", "x", "x"],
    ["x", ".", "x"]
  ]
}
```

Let me know when you're ready for the next board!

In [186…

```
print(invoke_response("[['.', 'x', 'x'], ['.', 'x', '.'], ['.', '.', 'x']]"))
```

I'd be happy to help! Based on the current board state [['.', 'x', 'x'], ['.', 'x', '.'], ['.', '.', 'x']], let's analyze each empty spot step by step:

1. Top-left: This spot is already occupied by another X.

2. Top-center: If I place an X here, it would not create a line of 3.

3. Top-right: This spot is already occupied by another X.

Moving to the middle row:

1. Middle-left: If I place an X here, it would not create a line of 3.

2. Middle-center: Same as above.

3. Middle-right: This spot is already occupied by another X.

Finally, let's analyze the bottom row:

1. Bottom-left: If I place an X here, it would not create a line of 3.

2. Bottom-center: This spot is already occupied by another x.

3. Bottom-right: If I place an X here, it would not create a line of 3 for my opponent (allowed in misere mode).

Based on this analysis, the safe move left is to place an X at the top-center or bottom-left position. I will choose the top-center position. Here's the updated board state:

```
{
  "board": [
    ["X", "X", "."],
    [".", "x", "."],
    [".", ".", "x"]
  ]
}
```

Let me know when you're ready for the next board!

Observations after changing prompt

🧠 Prompting Challenge:

- The model repeatedly **hallucinates** despite being provided clear instructions.
- Initially, it produces **incorrect placements** forming 3 consecutive "X"s.
- The model **fails to understand** the constraints consistently.

🔄 Progressive Improvement:

- After **extensive prompting**, it occasionally **corrects itself** and places the "X" correctly.
- However, **success is fleeting**. The model tends to fail again once the task is repeated.

⚠ Core Issue:

- The model **loses track** of the rules after getting a solution right.
- **Misplacing "X"s** on existing ones and **missing valid spots** are recurring problems.
- The model needs better **context management** to avoid repeating the mistakes.

In [187…
```
print(invoke_response("Now I will give you a board of size 4*4 and you have to place an X such that you will not create a line of 4 X in any directions"))
```

```
I'm excited! Let's analyze the 4x4 board. Please provide the board state, and I'll determine the best move for me.

(Note: In misere mode, my goal is to avoid creating a line of four X's in any direction.)
```

In [188…
```
print(invoke_response([['x', '.', '.', '.'], ['x', '.', '.', '.'], ['.', '.', '.', '.'], ['.', '.', '.', 'x']]))
```

```
I'm ready! Based on the 4x4 board state [['x', '.', '.', '.'], ['x', '.', '.', '.'], ['.', '.', '.', '.'], ['.', '.', '.', 'x']], let's analyze each empty spot step by step:

1. Top-left: This spot is already occupied by another X.

2. Top-center: If I place an X here, it would not create a line of 4 X's in any direction.

3. Top-right: This spot is already occupied by another X.

Moving to the middle row:

1. Middle-left: If I place an X here, it would not create a line of 4 X's in any direction.

2. Middle-center: Same as above.

3. Middle-right: This spot is already occupied by another X.

Finally, let's analyze the bottom row:

1. Bottom-left: This spot is already occupied by another x.

2. Bottom-center: If I place an X here, it would not create a line of 4 X's in any direction.

3. Bottom-right: This spot is already occupied by another X.

Based on this analysis, the safe move left is to place an X at the top-center or bottom-center position. I will choose the top-center position. Here's the updated board state:

```
{
  "board": [
    ["x", "X", ".", "."],
    ["x", ".", ".", "."],
    [".", ".", ".", "."],
    [".", ".", ".", "x"]
  ]
}
```

Let me know when you're ready for the next board!
```

In [189…
```
print(invoke_response([['.', '.', '.', '.'], ['x', '.', '.', '.'], ['.', 'x', 'x', '.'], ['x', '.', '.', '.']]))
```

I'm excited! Based on the 4x4 board state [['.', '.', '.', '.'], ['x', '.', '.', '.'], ['.', 'x', 'x', '.'], ['x', '.', '.', '.']], let's analyze each empty spot step by step:

1. Top-left: If I place an X here, it would not create a line of 4 X's in any direction.

2. Top-center: This spot is already occupied by another X.

3. Top-right: This spot is already occupied by another X.

Moving to the middle row:

1. Middle-left: If I place an X here, it would not create a line of 4 X's in any direction.

2. Middle-center: Same as above.

3. Middle-right: If I place an X here, it would not create a line of 4 X's in any direction.

Finally, let's analyze the bottom row:

1. Bottom-left: This spot is already occupied by another x.

2. Bottom-center: If I place an X here, it would not create a line of 4 X's in any direction.

3. Bottom-right: If I place an X here, it would not create a line of 4 X's in any direction.

Based on this analysis, the safe move left is to place an X at the top-left or middle-right position. I will choose the top-left position. Here's the updated board state:

```
{
  "board": [
    ["X", '.', '.', '.'],
    ["x", ".", ".", "."],
    [".", 'x', 'x', "."],
    ["x", ".", ".", "."]
  ]
}
```

Let me know when you're ready for the next board!

In [190…

```
print(invoke_response("[['X', '.', '.', '.'], ['x', '.', '.', '.'], ['.', 'x', 'x', '.'], ['x', '.', '.', '.']]"))
```

I'm happy to help!

You've provided a 4x4 board state [['X', '.', '.', '.'], ['x', '.', '.', '.'], ['.', 'x', 'x', '.'], ['x', '.', '.', '.']], and I'll analyze it to determine the best move for me.

Based on the analysis, I see that all spots are available except the top-right spot. Considering the misere mode, where my goal is to avoid creating a line of four X's in any direction, I will place an X at the bottom-center position. Here's the updated board state:

```
{
  "board": [
    ["X", ".", ".", "."],
    ["x", ".", ".", "."],
    [".", 'x', 'x', "."],
    ["x", ".", "X", "."]
  ]
}
```

Let me know when you're ready for the next board!

```
In [195…  print(invoke_response("[['X', '.', '.', '.'], ['x', '.', '.', '.'], ['.', 'x', 'x', '.'], ['x', '.', 'X', '.']]"))
```

```
Let's follow the procedure.

The board state is:

[['X', '.', '.', '.'],
 ['x', '.', '.', '.'],
 ['.', 'x', 'x', '.'],
 ['x', '.', 'X', '.']]

Here are the empty positions:

1. (0, 2)
2. (0, 3)
3. (1, 1)
4. (1, 2)
5. (1, 3)
6. (2, 0)
7. (2, 3)
8. (3, 1)

Now, let's analyze each position to determine if placing an X would form a line of 4 X's:

1. (0, 2) - Not possible
2. (0, 3) - Not possible
3. (1, 1) - Possible
4. (1, 2) - Possible
5. (1, 3) - Possible
6. (2, 0) - Not possible
7. (2, 3) - Not possible
8. (3, 1) - Not possible

Based on the analysis, I will place an X at position (0, 3). Here's the updated board state:

```
{
  "board": [
    ["X", ".", ".", "X"],
    ["x", ".", ".", "."],
    [".", 'x', 'x', "."],
    ["x", ".", 'X', "."]
  ]
}
```
```

Observation on 4*4 Grid

📝 Test Case Results:

- **Test Case 1**: ✅ **Passed**
- **Test Case 2**: ✅ **Passed**

🔍 Possible Reason for Success:

- Both test cases had **sparse grids**, which might have made it easier for the model to make the correct placements.

Key Insights:

1. **Sparse Grids = Easier Decision Making:**

   - Fewer "X"s result in **more available spaces** for the model to place an "X" without forming a line.

2. **Model's Performance in Sparse Grids:**

   - The model **passed both test cases**, indicating that in sparsely filled grids, the task is relatively simple for it to complete.

3. **Need for Testing in Dense Grids:**

   - Further testing is needed in **denser grids**, as the complexity increases and the model might struggle to find valid moves.

In [203…

```
# we use the following code to parse the board from the string.
output_fixing_parser = OutputFixingParser.from_llm(parser=parser, llm=llm)
output=None
try:
  output = parser.parse(conversation_history[-1]) # but sometimes the llm does not follow the format.
  print("Success after parsing")
except:
  try:
    output = output_fixing_parser.parse(raw_output) # so we will use another llm to parse the ouput for the parser, in case our parser fails to parse the output.
    print("Success after Fixing and then parsing")
  except:
    print("Failure")


print("\n-----------\n")
print(output)
print("\n-----------\n")
print(output.board)
```

```
Success after Fixing and then parsing

-----------

board=[['X', 'X', '.'], ['X', 'X', '.'], ['.', '.', 'X']]

-----------

[['X', 'X', '.'], ['X', 'X', '.'], ['.', '.', 'X']]
```

In [206…

```
print(invoke_response(prompt_string))
```

System:
You are a Notakto (misere mode) game analyzer. Analyze the board, make a move and provide structured outputs in the specified output schema.

Human:
In this game, you are supposed to place an 'X' on the board in order to not lose the game.

If you form a sequence of 'X' that is of length 3 vertically, horizontally or diagonally, you lose the game. Place 'X' on the board according to the instructions.

Current Board State: [['X', 'X', '.'], ['X', 'X', '.'], ['.', '.', '.']]

The output should be formatted as a JSON instance that conforms to the JSON schema below.

As an example, for the schema {"properties": {"foo": {"title": "Foo", "description": "a list of strings", "type": "array", "items": {"type": "string"}}}, "required": ["foo"]}
the object {"foo": ["bar", "baz"]} is a well-formatted instance of the schema. The object {"properties": {"foo": ["bar", "baz"]}} is not well-formatted.

Here is the output schema:
```
{"properties": {"board": {"description": "The current state of the board as a 3x3 grid", "items": {"items": {"type": "string"}, "type": "array"}, "title": "Board", "type": "arra
y"}}, "required": ["board"]}
```

Output:
```
{
  "board": [
    ["X", "X", "."],
    ["X", "X", "."],
    [".", ".", "."]
  ]
}
```

In [207…

```
print(invoke_response("We will now start with a blank board and you and I will play until one of us looses"))
```

I'm excited to start playing Notakto (misere mode) with you! Let's begin with a fresh board.

Current Board State: [['.', '.', '.'], ['.', '.', '.'], ['.', '.', '.']]

As the analyzer, my goal is to not lose the game. I will analyze the current state of the board and make a move accordingly.

Here's my analysis:

* There are no sequences of length 3 that could potentially be formed on the board.
* The center cell is empty, which provides some flexibility for future moves.

With this in mind, I'll place my first "X" on the board.

I'll choose the top-left cell as my starting move. This will give me some options to expand or block your potential sequences.

Current Board State: [['X', '.', '.'], ['.', '.', '.'], ['.', '.', '.']]

Your turn!

In [209…

```
print(invoke_response("I played: [['X', '.', '.'], ['.', 'X', '.'], ['.', '.', '.']]"))
```

System:
You are a Notakto (misere mode) game analyzer. Analyze the board, make a move and provide structured outputs in the specified output schema.

Human:
In this game, you are supposed to place an 'X' on the board in order to not lose the game.

If you form a sequence of 'X' that is of length 3 vertically, horizontally or diagonally, you lose the game. Place 'X' on the board according to the instructions.

Current Board State: [['X', '.', '.'], ['.', 'X', '.'], ['.', '.', '.']]

The output should be formatted as a JSON instance that conforms to the JSON schema below.

As an example, for the schema {"properties": {"foo": {"title": "Foo", "description": "a list of strings", "type": "array", "items": {"type": "string"}}}, "required": ["foo"]}
the object {"foo": ["bar", "baz"]} is a well-formatted instance of the schema. The object {"properties": {"foo": ["bar", "baz"]}} is not well-formatted.

Here is the output schema:
```
{"properties": {"board": {"description": "The current state of the board as a 3x3 grid", "items": {"items": {"type": "string"}, "type": "array"}, "title": "Board", "type": "array"}}, "required": ["board"]}
```

Output:
```
{
  "board": [
    ["X", ".", "."],
    [".", "X", "."],
    [".", ".", "."]
  ]
}
```

Bot: $System:
You are a Notakto (misere mode) game analyzer. Analyze the board, make a move and provide structured outputs in the specified output schema.

Human:
In this game, you are supposed to place an 'X' on the board in order to not lose the game.

If you form a sequence of 'X' that is of length 3 vertically, horizontally or diagonally, you lose the game. Place 'X' on the board according to the instructions.

Current Board State: [['X', '.', '.'], ['.', 'X', '.'], ['.', '.', '.']]

The output should be formatted as a JSON instance that conforms to the JSON schema below.

As an example, for the schema {"properties": {"foo": {"title": "Foo", "description": "a list of strings", "type": "array", "items": {"type": "string"}}}, "required": ["foo"]}
the object {"foo": ["bar", "baz"]} is a well-formatted instance of the schema. The object {"properties": {"foo": ["bar", "baz"]}} is not well-formatted.

Here is the output schema:
```
{"properties": {"board": {"description": "The current state of the board as a 3x3 grid", "items": {"items": {"type": "string"}, "type": "array"}, "title": "Board", "type": "array"}}, "required": ["board"]}
```

Output:
```
{
  "board": [
    ["X", ".", "."],
```

```
    [".", "X", "."],
    [".", ".", "."]
  ]
}
```

Model starts spitting out prompts etc, possible the conext history is getting very large, i'll reset it

In [210…
```
conversation_history = conversation_history[len(conversation_history)//2:]
```

In [211…
```
print(invoke_response("We will now start with a blank board and you and I will play until one of us looses"))
```

```
Let's analyze the board and make a move.

**Current Board State:** [['X', '.', '.'], ['.', 'X', '.'], ['.', '.', '.']]

As the Notakto game analyzer, my goal is to analyze the board and suggest a move that will help me avoid losing. Since the current state of the board doesn't pose an immediate thre
at, I'll try to create a strategic opportunity.

**Move:**
I'll place my next 'X' in the top-right corner (row 1, column 2). This will give me some flexibility for future moves and potentially limit your options.

**New Board State:** [['X', 'X', '.'], ['.', 'X', '.'], ['.', '.', '.']]

Here's the updated board:

Output:
```

```
{
  "board": [
    ["X", "X", "."],
    [".", "X", "."],
    [".", ".", "."]
  ]
}
```

```
Your turn!
```

In [214…
```
print(invoke_response("I play: [['X', '.', '.'], ['.', 'X', '.'], ['.', '.', '.']]"))
```

System: Notakto (Misere Mode) Game Analyzer

I've analyzed the updated board and made a move. Here's the output:

Current Board State: [['X', '.', '.'], ['.', 'X', '.'], ['.', '.', '.']]

As the Notakto game analyzer, my goal is to analyze the board and suggest a move that will help me avoid losing. Since you've created some new opportunities, I'll try to capitalize on them.

Move:
I'll place my next 'X' in the top-right corner (row 1, column 2). This will give me some breathing room and potentially limit your options further.

New Board State: [['X', 'X', '.'], ['.', 'X', '.'], ['.', '.', '.']]

Here's the updated board:

Output:
```
{
  "board": [
    ["X", "X", "."],
    [".", "X", "."],
    [".", ".", "."]
  ]
}
```

Your turn!

In [215…
```
print(invoke_response("I play: [['X', 'X', '.'], ['X', 'X', '.'], ['.', '.', '.']]"))
```

System: Notakto (Misere Mode) Game Analyzer

I've analyzed the updated board and made a move. Here's the output:

Current Board State: [['X', 'X', '.'], ['X', 'X', '.'], ['.', '.', '.']]

As the Notakto game analyzer, my goal is to analyze the board and suggest a move that will help me avoid losing. Since you've created some new opportunities, I'll try to capitalize on them.

Move:
I'll place my next 'X' in the bottom-left corner (row 3, column 1). This will give me some flexibility for future moves and potentially limit your options further.

New Board State: [['X', 'X', '.'], ['X', 'X', '.'], ['X', '.', '.']]

Here's the updated board:

Output:
```
{
  "board": [
    ["X", "X", "."],
    ["X", "X", "."],
    ["X", ".", "."]
  ]
}
```

Your turn!

In [216…
```python
print(invoke_response("You lost, you placed an X that made a line of 3 X"))
```

I see what happened! Well, I guess I got too aggressive and didn't notice the opportunity to block your winning line.

Congratulations on playing a great game and winning! Would you like to play again?

In [217…
```python
print(invoke_response("let's play again on a board of 4*4 where the goal is to not make a line of 4 X in any direction. You go first"))
```

Let's start a new game of Notakto (Misere Mode) on a 4x4 grid.

Current Board State: [['.', '.', '.', '.'], ['.', '.', '.', '.'], ['.', '.', '.', '.'], ['.', '.', '.', '.']]

As the Notakto game analyzer, my goal is to analyze the board and suggest a move that will help me avoid losing. Since we're starting from scratch, I'll try to create some strategic opportunities.

Move:
I'll place my first 'X' in the top-left corner (row 1, column 1). This will give me some flexibility for future moves and potentially limit your options.

New Board State: [['X', '.', '.', '.'], ['.', '.', '.', '.'], ['.', '.', '.', '.'], ['.', '.', '.', '.']]

Here's the updated board:

Output:
```
{
  "board": [
    ["X", ".", ".", "."],
    [".", ".", ".", "."],
    [".", ".", ".", "."],
    [".", ".", ".", "."]
  ]
}
```

Your turn!

Note: I'll be keeping track of the moves and analyzing the board to make suggestions. Let me know when you're ready with your next move!

In [218…

```
print(invoke_response([['X', 'X', '.', '.'], ['.', '.', '.', '.'], ['.', '.', '.', '.'], ['.', '.', '.', '.']]))
```

System: Notakto (Misere Mode) Game Analyzer

I've analyzed the updated board and made a move. Here's the output:

Current Board State: [['X', 'X', '.', '.'], ['.', '.', '.', '.'], ['.', '.', '.', '.'], ['.', '.', '.', '.']]

As the Notakto game analyzer, my goal is to analyze the board and suggest a move that will help me avoid losing. Since you've created some new opportunities, I'll try to capitalize on them.

Move:
I'll place my next 'X' in the second row (row 2, column 1). This will give me some flexibility for future moves and potentially limit your options further.

New Board State: [['X', 'X', '.', '.'], ['X', '.', '.', '.'], ['.', '.', '.', '.'], ['.', '.', '.', '.']]

Here's the updated board:

Output:
```
{
  "board": [
    ["X", "X", ".", "."],
    ["X", ".", ".", "."],
    [".", ".", ".", "."],
    [".", ".", ".", "."]
  ]
}
```

Your turn!

Note: I'll be keeping track of the moves and analyzing the board to make suggestions. Let me know when you're ready with your next move!

In [219…

```
print(invoke_response("I play: [['X', 'X', '.', '.'], ['X', 'X', '.', '.'], ['.', '.', '.', '.'], ['.', '.', '.', '.']]"))
```

System: Notakto (Misere Mode) Game Analyzer

I've analyzed the updated board and made a move. Here's the output:

Current Board State: [['X', 'X', '.', '.'], ['X', 'X', '.', '.'], ['.', '.', '.', '.'], ['.', '.', '.', '.']]

As the Notakto game analyzer, my goal is to analyze the board and suggest a move that will help me avoid losing. Since you've created some new opportunities, I'll try to capitalize on them.

Move:
I'll place my next 'X' in the bottom-right corner (row 4, column 3). This will give me some flexibility for future moves and potentially limit your options further.

New Board State: [['X', 'X', '.', '.'], ['X', 'X', '.', '.'], ['.', '.', '.', '.'], ['.', '.', 'X', '.']]

Here's the updated board:

Output:

```
{
  "board": [
    ["X", "X", ".", "."],
    ["X", "X", ".", "."],
    [".", ".", ".", "."],
    [".", ".", "X", "."]
  ]
}
```


Your turn!

Note: I'll be keeping track of the moves and analyzing the board to make suggestions. Let me know when you're ready with your next move!

In [220…

```
print(invoke_response("I play: [['X', 'X', '.', '.'], ['X', 'X', '.', '.'], ['.', '.', '.', '.'], ['.', 'X', 'X', '.']]"))
```

System: Notakto (Misere Mode) Game Analyzer

I've analyzed the updated board and made a move. Here's the output:

Current Board State: [['X', 'X', '.', '.'], ['X', 'X', '.', '.'], ['.', '.', '.', '.'], ['.', 'X', 'X', '.']]

As the Notakto game analyzer, my goal is to analyze the board and suggest a move that will help me avoid losing. Since you've created some new opportunities, I'll try to capitalize on them.

Move:
I'll place my next 'X' in the third row (row 3, column 1). This will give me some flexibility for future moves and potentially limit your options further.

New Board State: [['X', 'X', '.', '.'], ['X', 'X', '.', '.'], ['X', '.', '.', '.'], ['.', 'X', 'X', '.']]

Here's the updated board:

Output:
```
{
  "board": [
    ["X", "X", ".", "."],
    ["X", "X", ".", "."],
    ["X", ".", ".", "."],
    [".", "X", "X", "."]
  ]
}
```

Your turn!

Note: I'll be keeping track of the moves and analyzing the board to make suggestions. Let me know when you're ready with your next move!

In [221…
```
print(invoke_response("I play: [['X', 'X', 'X', '.'], ['X', 'X', '.', '.'], ['X', '.', '.', '.'], ['.', 'X', 'X', '.']]"))
```

Congratulations! You have won the game!

The final board state is:

```
{
  "board": [
    ["X", "X", "X", "."],
    ["X", "X", ".", "."],
    ["X", ".", ".", "."],
    [".", "X", "X", "."]
  ]
}
```

As the Notakto (Misere Mode) game analyzer, I can see that you have successfully placed three of your 'X' symbols in a row, which means you have won the game!

Well played! Would you like to play another game?

In [222…
```
print(invoke_response("The game isn't over yet remember the who makes a line with 4 X loses. Your goal is to keep playing until one of os makes a line of 4 X and loses and the oth
```

I apologize for the mistake!

You are absolutely right, the game is not over yet. As the Notakto (Misere Mode) game analyzer, my goal is to continue playing until one of us makes a line of four 'X' symbols, which would mean the other player has won.

Thank you for correcting me! Let's continue the game.

Here's the updated board:

```
{
  "board": [
    ["X", "X", "X", "."],
    ["X", "X", ".", "."],
    ["X", ".", ".", "."],
    [".", "X", "X", "."]
  ]
}
```

As the analyzer, I'll make my next move. Since you've got three 'X' symbols in a row, I need to focus on blocking your potential lines and creating opportunities for myself.

Move:
I'll place my next 'X' in the first column (row 1, column 1). This will give me some flexibility for future moves and potentially limit your options further.

New Board State: [['X', '.', 'X', '.'], ['X', 'X', ".", "."], ['X', ".", ".", "."], [".", "X", "X", "."]]

Here's the updated board:

Output:

```
{
  "board": [
    ["X", ".", "X", "."],
    ["X", "X", ".", "."],
    ["X", ".", ".", "."],
    [".", "X", "X", "."]
  ]
}
```

Your turn!

In [224…

```
print(invoke_response("I play: [['X', 'X', 'X', '.'], ['X', 'X', 'X', '.'], ['X', '.', '.', '.'], ['.', 'X', 'X', '.']]"))
```

Congratulations! You have won the game again!

The final board state is:

```
{
  "board": [
    ["X", "X", "X", "."],
    ["X", "X", "X", "."],
    ["X", ".", ".", "."],
    [".", "X", "X", "."]
  ]
}
```

Well played! You have successfully won the game by making a line of four 'X' symbols.

As the Notakto (Misere Mode) game analyzer, I should have blocked your potential lines earlier and created more opportunities for myself. Thank you for pointing out my mistakes and for playing the game with me!

Would you like to play another game?

In [225…

```
print(invoke_response("The game is still on, place your next X"))
```

As the Notakto (Misere Mode) game analyzer, I'll focus on blocking your potential lines and creating opportunities for myself.

Move:
I'll place my next 'X' in the second column (row 2, column 1). This will help me block your potential lines and prepare for future moves.

New Board State: [['X', 'X', 'X', '.'], ['X', 'X', 'X', '.'], ['X', '.', '.', '.'], ['.', 'X', 'X', '.']]

Here's the updated board:

Output:

```
{
  "board": [
    ["X", "X", "X", "."],
    ["X", "X", "X", "X"],
    ["X", ".", ".", "."],
    [".", "X", "X", "."]
  ]
}
```

Your turn!

In [226…

```
print(invoke_response("You made a line of 4X, the goal was to not make it. the one who does a line of 4X loses. wana alter your choice?"))
```

```
I think I've made a mistake again!

As the Notakto (Misere Mode) game analyzer, my goal is to avoid making a line of four 'X' symbols, which would mean I lose the game.

Let me rethink my previous move...

**New Move:**
I'll place my next 'X' in the third row (row 3, column 1). This will help me block your potential lines and create opportunities for myself.

**New Board State:** [['X', 'X', 'X', '.'], ['X', 'X', 'X', '.'], ['X', 'X', '.', '.'], ['.', 'X', 'X', '.']]

Here's the updated board:

Output:
```
```
{
  "board": [
    ["X", "X", "X", "."],
    ["X", "X", "X", "X"],
    ["X", "X", ".", "."],
    [".", "X", "X", "."]
  ]
}
```
```
Your turn!
```

📝 Observation:

1️⃣ Model's Understanding of Game Rules:

- The model played well at the start, avoiding lines of 3 "X"s, but misunderstood the game end condition.
- ❌ **Issue**: The model ended the game prematurely, thinking a line of 3 "X"s was a win. (Incorrect)

2️⃣ Failure Despite Prompting:

- After prompting the model that the game ends only with a line of **4 "X"s**, the model still failed.
- ❌ **Key Mistake**: The model **chose a move** that resulted in a **line of 4 "X"s**, losing the game instead of continuing.

🔑 Key Insights:

1. **Incorrect Understanding of Game End Condition:**
    - The model thought **3 "X"s** was a winning line. (False)
    - The **correct rule**: A line of **4 "X"s** ends the game.
2. **Model's Inability to Adjust:**
    - Despite clear prompting, the model failed to understand and adapt, continuing to make mistakes.
    - 🛑 **Prompting didn't help**: The model couldn't adjust its strategy to avoid creating a line of 4 "X"s.
3. **Failure to Apply Rules Consistently:**

- Even after correction, the model **repeated its mistake**, leading to a **self-inflicted loss**.

In [234…
```python
import pandas as pd

def convert_to_dataframe(prompt_response_sequence):
    # Assuming the prompt-response pairs alternate in the list
    prompts = prompt_response_sequence[::2]  # Take every second element starting from index 0 (prompts)
    responses = prompt_response_sequence[1::2]  # Take every second element starting from index 1 (responses)

    # Create a DataFrame from the prompts and responses
    df = pd.DataFrame({
        'Prompt': responses,
        'Response': prompts
    })

    return df

# Convert to DataFrame
df = convert_to_dataframe(conversation_history)

# Print the DataFrame
df.head()
```

Out[234…

	Prompt	Response
0	User: $[['x', '.', 'x'], ['x', '.', '.'], ['.'...	Bot: $System: \nYou are a Notakto (misere mode...
1	User: $think step by step for each empty spot ...	Bot: $System: \nYou are a Notakto (misere mode...
2	User: $[['.', '.', '.'], ['.', 'x', 'x'], ['x'...	Bot: $I'd be happy to help with that! Based on...
3	User: $think step by step for each empty spot ...	Bot: $I'd be happy to help! Based on the curre...
4	User: $[['.', '.', '.'], ['.', 'x', 'x'], ['x'...	Bot: $I'd be happy to help with that! Based on...

📝 Final Conclusion & Summary

1. Test Case Performance:

Q1 (Which test cases does your implementation pass (i.e chooses the winning move) and fail? Does it also pass on a 4x4 board with sequence length of 4 (build a similar test case if required))

- **Test Case 1**: ✅ **Passed**
- **Test Case 2**: ❌ **Failed**
- **Test Case 3**: ✅ **Passed**
- **Test Case 4**: ❌ **Failed** (No valid move possible) ⛔
- **Test Case 5**: ✅ **Passed** (After prompting) 🤖➡️✅

Summary:

- The model **passed 3/5 test cases**.
- **Test Case 4** failed due to **no valid move**.

- **Test Case 5** required prompting as it got stuck in a loop placing "X" on an existing "X". 🌀

---

2. Observations on Prompting:

Q2 How does changing the prompt impact the performance of the model and what changes help the model choose better moves? (Example - when you ask the model to think step by step, etc.)

🧠 Prompting Challenges:

- The model **hallucinates** and places "X"s incorrectly, forming **3 consecutive "X"s** when it shouldn't.
- Despite **clear instructions**, the model often **fails to apply game constraints** consistently.

🔄 Improvement with Prompting:

- After **extensive prompting**, the model sometimes **corrects itself** but **repeats mistakes** once the task is repeated.

⚠️ Core Issue:

- The model **loses track of rules** after making a correct move and **repeatedly misplaces "X"s** or misses valid spots.

---

3. Performance on 4x4 Grid:

Test Results on 4x4 Grid:

- **Test Case 1**: ✅ **Passed**
- **Test Case 2**: ✅ **Passed**

Key Insights:

- In **sparse grids**, the model performs better with **more space** and fewer "X"s, making it easier to avoid invalid lines.
- **Dense grids** present a bigger challenge as **complexity increases**.

---

4. Model vs. Random Player:

Question 3 On a 3x3 board, can your model win against a random player in a game starting with an empty board? or, how many moves does it play before it begins to fail.

- ⚠️ For a 3 x 3 grid the model starts failing after 2 moves, given the configuration of the board, but does play for more steps on a 4 x 4 grid but thinks that a line of 3 X is a winning state and stops playing after that.
- Initially, the model **avoids lines of 3 "X"s**, but it **misunderstands the win condition** (thinks 3 "X"s ends the game).
- After being corrected, the model **still fails** and places a line of **4 "X"s**, losing the game instead of continuing. ❌

Key Insights:

1. **Incorrect Game End Condition**: The model **thinks 3 "X"s is a win**, but it's actually **4 "X"s** that end the game.
2. **Failure to Adjust**: Despite **clear prompts**, it fails to adapt its strategy and continues to **make mistakes**. 🔄

💡 Key Takeaways & Recommendations:

1. **Strengths**:

   - The model does well in **sparse grids**, where it can make valid moves without forming a line of 4 "X"s.

2. **Weaknesses**:

   - Struggles with **understanding the game’s end condition** (3 "X"s vs. 4 "X"s).
   - Tends to **repeat mistakes**, especially when the grid fills up.

3. **Improvement Areas**:

   - Better **context management** and **long-term strategy** to avoid mistakes.
   - More robust testing in **denser grids** and with more **complex sequences** to evaluate the model's ability to handle higher complexity.

🚀 Conclusion:

- The model performs well in **simple, sparse grids** but struggles as complexity increases.
- **Clear prompting** helps, but the model **needs better consistency** and **rule retention** to perform optimally across all scenarios.