ABCM Computer lab 1: Social coordination & Theory of mind

In this computer lab, we will use the tomsup package created by Waade et al. (2022) to simulate various Game Theory games between agents that have varying levels of theory of mind (or other strategies).

If you are completely new to Jupyter notebooks, start with this introduction: https://realpython.com/jupyter-notebook-introduction/

All exercises are indicated with an **Exercise *N*** header. The notebook also contains some explanation, which is also interleaved with small coding exercises (of the form *"In the code cell below, do X"*) which help you understand how the code works.

First, install the `tomsup` package by running the code cell below:

```
In [ ]: #!pip3 install tomsup
```

Now, let's do the necessary imports by running the code cell below:

```
In [ ]: import tomsup as ts
        import numpy as np
        import matplotlib.pyplot as plt
```

Here are a few things to keep in mind throughout the computer lab:

- When plotting results, pay attention to the scale of the y-axis.
- When running simulations, make sure that you run:
    1. Enough rounds such that the agents' behaviour and/or estimates of each other's parameters is no longer changing (i.e., that you run the simulation until convergence).
    2. Enough independent simulation runs to get a good sense of the stochastic variation between runs

Exploring the games in tomsup

Exercise 1:

Use the command `help(ts.PayoffMatrix)` (see page 11 of Waade et al., 2022) to explore what Game Theory games are pre-specified in the tomsup package. Print and investigate each of these pay-off matrices. For each one: Write down whether they are competitive or cooperative in nature. Also explain why.

Note that there's a typo in one of the game names in the documentation of `tomsup`. It should be `'penny_competitive'`, *not* `'penny_competive'`.

```
In [ ]: # help(ts.PayoffMatrix)

        game_names = ['staghunt', 'penny_competitive', 'penny_cooperative', 'party', 's

        for game_name in game_names:
            print('===============[' + game_name + ']===============')
            print(ts.PayoffMatrix(name = game_name))
            print()
```

```
===============[staghunt]===============
<Class PayoffMatrix, Name = staghunt>
The payoff matrix of agent 0
       |  Choice agent 1
       |   | 0 | 1 |
       | --------- |
Choice | 0 | 3 | 3 |
agent 0| 1 | 0 | 5 |

The payoff matrix of agent 1
       |  Choice agent 1
       |   | 0 | 1 |
       | --------- |
Choice | 0 | 3 | 0 |
agent 0| 1 | 3 | 5 |


===============[penny_competitive]===============
<Class PayoffMatrix, Name = penny_competitive>
The payoff matrix of agent 0
       |  Choice agent 1
       |    |  0 |  1 |
       | ------------ |
Choice |  0 | -1 |  1 |
agent 0|  1 |  1 | -1 |

The payoff matrix of agent 1
       |  Choice agent 1
       |    |  0 |  1 |
       | ------------ |
Choice |  0 |  1 | -1 |
agent 0|  1 | -1 |  1 |


===============[penny_cooperative]===============
<Class PayoffMatrix, Name = penny_cooperative>
The payoff matrix of agent 0
       |  Choice agent 1
       |    |  0 |  1 |
       | ------------ |
Choice |  0 |  1 | -1 |
agent 0|  1 | -1 |  1 |

The payoff matrix of agent 1
       |  Choice agent 1
       |    |  0 |  1 |
       | ------------ |
Choice |  0 |  1 | -1 |
agent 0|  1 | -1 |  1 |


===============[party]===============
<Class PayoffMatrix, Name = party>
The payoff matrix of agent 0
       |  Choice agent 1
       |    |  0 |  1 |
       | ------------ |
Choice |  0 |  5 |  0 |
agent 0|  1 |  0 | 10 |

The payoff matrix of agent 1
       |  Choice agent 1
       |    |  0 |  1 |
       | ------------ |
Choice |  0 |  5 |  0 |
agent 0|  1 |  0 | 10 |


===============[sexes]===============
<Class PayoffMatrix, Name = sexes>
The payoff matrix of agent 0
       |  Choice agent 1
       |    |  0 |  1 |
       | ------------ |
Choice |  0 | 10 |  0 |
agent 0|  1 |  0 |  5 |

The payoff matrix of agent 1
       |  Choice agent 1
       |    |  0 |  1 |
       | ------------ |
Choice |  0 |  5 |  0 |
agent 0|  1 |  0 | 10 |


===============[chicken]===============
<Class PayoffMatrix, Name = chicken>
The payoff matrix of agent 0
       |  Choice agent 1
       |       |     0 |     1 |
       | --------------------- |
Choice |     0 | -1000 |    -1 |
agent 0|     1 |     1 |     0 |

The payoff matrix of agent 1
       |  Choice agent 1
       |       |     0 |     1 |
       | --------------------- |
Choice |     0 | -1000 |     1 |
agent 0|     1 |    -1 |     0 |


===============[deadlock]===============
<Class PayoffMatrix, Name = deadlock>
The payoff matrix of agent 0
       |  Choice agent 1
       |   | 0 | 1 |
       | --------- |
Choice | 0 | 1 | 0 |
agent 0| 1 | 3 | 2 |

The payoff matrix of agent 1
       |  Choice agent 1
       |   | 0 | 1 |
       | --------- |
Choice | 0 | 1 | 3 |
agent 0| 1 | 0 | 2 |


===============[prisoners_dilemma]===============
<Class PayoffMatrix, Name = prisoners_dilemma>
The payoff matrix of agent 0
       |  Choice agent 1
       |   | 0 | 1 |
       | --------- |
Choice | 0 | 1 | 5 |
agent 0| 1 | 0 | 3 |

The payoff matrix of agent 1
       |  Choice agent 1
       |   | 0 | 1 |
       | --------- |
Choice | 0 | 1 | 0 |
agent 0| 1 | 5 | 3 |
```

- stughunt: Cooperative, because both agents get the biggest reward when they both choose Choice 1
- penny_competitive: Competitive, because whenver agent 0 wins, agent 1 loses and vice versa
- penny_cooperative: Cooperative, because (the name says so) when they both make the same choice (e.g., 1 & 1), they get rewards, and when they make different choices (e.g., 0 & 1), they both lose
- party: Cooperative, because when they both make the same choice (e.g., 1 & 1), they get rewards, and when they make different choices (e.g., 0 & 1), they both lose
- sexes: Cooperative, because when agent 0 gets reward, agent 1 always gets reward too.
- chicken: Competitive, as there is no scinario where both agents get rewards.
- deadlock: Cooperative, as when both agents choose 1, they both get rewards.
- prisoners_dilemma: Competitive, because when agent 0 gets reward (i.e., agent0 betray [1] and agent1 remains silent [0]), agent 1 loses.

Exercise 2:

`penny_competitive` is an example of a zero-sum game. The definition of a zero-sum game is as follows: "*games in which choices by players can neither increase nor decrease the available resources. In zero-sum games, the total benefit that goes to all players in a game, for every combination of strategies, always adds to zero (more informally, a player benefits only at the equal expense of others)*" Can you find any other example of a zero-sum game among the predefined games in the tomsup package?

I beleive "chicken" is another example of a zero-sum game as whenver either agent gets reward (i.e., 1), the other agent loses the same amount.

Exercise 3:

`prisoners_dilemma` is an example of a game that has a Nash equilibrium that is suboptimal for both agents. That is, when both agents decide to betray each other (i.e, both choose action 0), they are worse off than if they both remain silent (i,e., both choose action 1). However, if they are in a state where they both choose action 0, neither agent can improve their own pay-off by changing strategy, making this state a Nash equilibrium. Can you find any other games among the predefined games that have such a Nash equilibrium that is suboptimal for both agents? If so, explain why.

I believe deadlock is another example of a game that has a Nash equilibrium. When both agents decide to choose Choice 1, changing their strategies (i.e., switching to Choice 0) will only make them lose two points.

Running interactions between agents

Creating a game:

A game can be created using the `PayoffMatrix` class, as follows:

In [ ]:
```
penny = ts.PayoffMatrix(name='penny_competitive')

print(penny)
```

```
<Class PayoffMatrix, Name = penny_competitive>
The payoff matrix of agent 0
       |  Choice agent 1
       |    |  0 |  1 |
       | ------------ |
Choice |  0 | -1 |  1 |
agent 0|  1 |  1 | -1 |

The payoff matrix of agent 1
       |  Choice agent 1
       |    |  0 |  1 |
       | ------------ |
Choice |  0 |  1 | -1 |
agent 0|  1 | -1 |  1 |
```

Try this in the code cell below by creating a staghunt game and printing it:

In [ ]:
```
staghunt = ts.PayoffMatrix(name = 'staghunt')
print(staghunt)
```

```
<Class PayoffMatrix, Name = staghunt>
The payoff matrix of agent 0
       |  Choice agent 1
       |   | 0 | 1 |
       | --------- |
Choice | 0 | 3 | 3 |
agent 0| 1 | 0 | 5 |

The payoff matrix of agent 1
       |  Choice agent 1
       |   | 0 | 1 |
       | --------- |
Choice | 0 | 3 | 0 |
agent 0| 1 | 3 | 5 |
```

Creating a group of agents

A group of agents can be created quickly, using the `create_agents()` function, which returns an object of the `AgentGroup` class. This function takes two input arguments:

1. `agents` : specifies the agent types in the group. Possible agent types are:
    - 'RB'
    - 'QL'
    - 'WSLS'
    - '1-TOM'
    - '2-TOM'
2. `start_params` : specifies the starting parameters for each agent. An empty dictionary {}, denoted by `{}` gives default values

In [ ]:
```python
starting_parameters = [{'bias':0.7}, {'learning_rate':0.5}, {}, {}, {}]
```

In the code cell below, use the `ts.create_agents()` function to create an object of the `AgentGroup` class, which you assign to a variable called `group`. Use the following input arguments:

1. Create a list called `agent_types` which contains all possible agent types names as listed above. Pass that as the `agents` argument
2. Create a list called `starting_parameters` which contains:
    - `{'bias':0.7}` for the 'RB' agent
    - `{'learning_rate':0.5}` for the 'QL' agent
    - the default parameters (i.e., empty dictionary) for all other agent types

Once you've created your `group` object, print it and inspect it, using `print(group)`

In [ ]:
```python
agent_types = ['RB', 'QL', 'WSLS', '1-TOM', '2-TOM']
starting_parameters = [{'bias':0.7}, {'learning_rate':0.5}, {}, {}, {}]

group = ts.create_agents(agent_types, starting_parameters)
print(group)
```

```
<Class AgentGroup, envinment = None>

RB        |      {'bias': 0.7}
QL        |      {'learning_rate': 0.5}
WSLS      |      {}
1-TOM     |      {}
2-TOM     |      {}
```

You can inspect the further functionality of the `AgentGroup` class using the following command:

```
In [ ]: help(ts.AgentGroup)
```

```
Help on class AgentGroup in module tomsup.agent:

class AgentGroup(builtins.object)
 |  AgentGroup(agents: List[str], start_params: Optional[List[dict]] = None)
 |
 |  An agent group is a group of agents. It is a utility class to allow for
 |  easily setting up tournaments.
 |
 |  Examples:
 |      >>> round_table = AgentGroup(agents=['RB']*2,             start_params
=[{'bias': 1}]*2)
 |      >>> round_table.agent_names
 |      ['RB_0', 'RB_1']
 |      >>> RB_0 = round_table.get_agent('RB_0') # extract an agent
 |      >>> RB_0.bias == 1 # should naturally be 1, as we specified it
 |      True
 |      >>> round_table.set_env('round_robin')
 |      >>> result = round_table.compete(p_matrix="penny_competitive",
n_rounds=100, n_sim=10)
 |      Currently the pair, ('RB_0', 'RB_1'), is competing for 10 simulations,
each containg 100 rounds.
 |          Running simulation 1 out of 10
 |          Running simulation 2 out of 10
 |          Running simulation 3 out of 10
 |          Running simulation 4 out of 10
 |          Running simulation 5 out of 10
 |          Running simulation 6 out of 10
 |          Running simulation 7 out of 10
 |          Running simulation 8 out of 10
 |          Running simulation 9 out of 10
 |          Running simulation 10 out of 10
 |      Simulation complete
 |      >>> result.shape[0] == 10*100 # As there is 10 simulations each contai
ning                                    100 round
 |      True
 |      >>> result['payoff_agent0'].mean() == 1  # Given that both agents have
always choose 1, it is clear that agent0 always win, when playing the
competitive pennygame
 |      True
 |
 |  Methods defined here:
 |
 |  __init__(self, agents: List[str], start_params: Optional[List[dict]] = Non
e)
 |      Args:
 |          agents (List[str]): A list of agents
 |          start_params (Optional[List[dict]], optional): The starting parame
ters of the agents specified
 |              as a dictionary pr. agent. Defaults to None, indicating defaul
t for all agent. Use empty to
 |              use default of an agent.
 |
 |  __str__(self) -> str
 |      Return str(self).
 |
 |  compete(self, p_matrix: tomsup.payoffmatrix.PayoffMatrix, n_rounds: int =
10, n_sim: int = 1, reset_agent: bool = True, env: Optional[str] = None, save_
history: bool = False, verbose: bool = True, n_jobs: Optional[int] = None) ->
pandas.core.frame.DataFrame
 |      for each pair competes using the specified parameters
 |
 |      Args:
 |          p_matrix (PayoffMatrix): The payoffmatrix in which the agents comp
ete
 |          n_rounds (int, optional): Number of rounds the agent should play i
n each simulation.
 |              Defaults to 10.
 |          n_sim (int, optional): The number of simulations. Defaults to 1.
 |          reset_agent (bool, optional): Should the agent be reset ? Defaults
to True.
 |          env (Optional[str], optional): The environment in which the agent
should compete.
 |              Defaults to None, indicating the already set environment.
 |          save_history (bool, optional): Should the history of agent be save
d.
 |              Defaults to False, as this is memory intensive.
 |          verbose (bool, optional): Toggles the verbosity of the function. D
efaults to True.
 |          n_jobs (Optional[int], optional): Number of parallel jobs. Default
s to None, indicating no parallelization.
 |              -1 indicate as many jobs as there is cores on your unit.
 |
 |      Returns:
 |          pd.DataFrame: A pandas dataframe of the results.
 |
 |  get_agent(self, agent: str) -> tomsup.agent.Agent
 |
 |  get_environment(self)
 |      Returns:
 |          the pairing resulted from the set environment
 |
 |  get_environment_name(self) -> str
 |      Returns:
 |          str: The name of the set environment
 |
 |  get_names(self) -> List[str]
 |      Returns:
 |          List[str]: the names of the agents
 |
 |  get_results(self) -> pandas.core.frame.DataFrame
 |      Returns:
 |          pd.DataFrame: The results
 |
 |  plot_choice(self, agent0: str, agent1: str, agent: int = 0, sim: Optional
[int] = None, plot_individual_sim: bool = False, show: bool = True)
 |      plots the choice of an agent in a defined agent pair
 |
 |      Args:
 |          agent0 (str): The name of agent0
 |          agent1 (str): The name of agent1
 |          agent (int, optional): An int denoting which of agent 0 or 1 you s
hould plot. Defaults to 0.
 |          plot_individual_sim (bool, optional): Should you plot each individ
ual simulation. Defaults to False.
```

```
 |          show (bool, optional): Should plt.show be run at the end. Defaults
to True.
 |
 |  plot_heatmap(self, aggregate_col: str = 'payoff_agent', aggregate_fun: Cal
lable = <function mean at 0x10765c670>, certainty_fun: Union[Callable, str] =
'mean_ci_95', cmap: str = 'Blues', na_color: str = 'xkcd:white', xlab: str =
'Agent', ylab: str = 'Opponent', cbarlabel: str = 'Average score of the agen
t', show: bool = True)
 |      plot a heatmap of the results.
 |
 |      Args:
 |          aggregate_col (str, optional): The column to aggregate on. Default
s to "payoff_agent".
 |          aggregate_fun (Callable, optional): The function to aggregate by.
Defaults to np.mean.
 |          certainty_fun (Union[Callable, str], optional): The certainty func
tion specified as a string on
 |              the form "mean_ci_X" where X denote the confidence interval, o
r a function.
 |              Defaults to "mean_ci_95".
 |          cmap (str, optional): The color map. Defaults to "Blues".
 |          na_color (str, optional): The color of NAs. Defaults to "xkcd:whit
e", e.g. white.
 |          xlab (str, optional): The name on the x-axis. Defaults to "Agent".
 |          ylab (str, optional): The name of the y-axis. Defaults to "Opponen
t".
 |          show (bool, optional): Should plt.show be run at the end. Defaults
to True.
 |
 |  plot_history(self, agent0: int, agent1: int, state: str, agent: int = 0, f
un: Callable = <function AgentGroup.<lambda> at 0x13cf7ac10>, ylab: str = '',
xlab: str = 'Round', show: bool = True)
 |      Plots the history of an agent in a defined agent pair
 |
 |      Args:
 |          agent0 (str): The name of agent0
 |          agent1 (str): The name of agent1
 |          agent (int, optional): An int denoting which of agent 0 or 1 you s
hould plot. Defaults to 0.
 |          state (str):  The state of the agent you wish to plot.
 |          fun (Callable, optional): A function for extracting the state. Def
aults to lambdax:x[state].
 |          xlab (str, optional): The name on the x-axis. Defaults to "Agent".
 |          ylab (str, optional): The name of the y-axis. Defaults to "Opponen
t".
 |          show (bool, optional): Should plt.show be run at the end. Defaults
to True.
 |
 |  plot_op_states(self, agent0: str, agent1: str, state: str, level: int = 0,
agent: int = 0, show: bool = True)
 |      plots the p_self of a k-ToM agent in a defined agent pair
 |
 |      Args:
 |          agent0 (str): The name of agent0
 |          agent1 (str): The name of agent1
 |          agent (int, optional): An int denoting which of agent 0 or 1 you s
hould plot. Defaults to 0.
```

```
 |          state (str): a state of the simulated opponent you wish to plot.
 |          level (str): level of the similated opponent you wish to plot.
 |          show (bool, optional): Should plt.show be run at the end. Defaults
to True.
 |
 |  plot_p_k(self, agent0: str, agent1: str, level: int, agent: int = 0, show:
bool = True)
 |      plots the p_k of a k-ToM agent in a defined agent pair
 |
 |      Args:
 |          agent0 (str): The name of agent0
 |          agent1 (str): The name of agent1
 |          agent (int, optional): An int denoting which of agent 0 or 1 you s
hould plot. Defaults to 0.
 |          show (bool, optional): Should plt.show be run at the end. Defaults
to True.
 |
 |  plot_p_op_1(self, agent0: str, agent1: str, agent: int = 0, show: bool = T
rue) -> None
 |      plots the p_op_1 of a k-ToM agent in a defined agent pair
 |
 |      Args:
 |          agent0 (str): The name of agent0
 |          agent1 (str): The name of agent1
 |          agent (int, optional): An int denoting which of agent 0 or 1 you s
hould plot. Defaults to 0.
 |          show (bool, optional): Should plt.show be run at the end. Defaults
to True.
 |
 |  plot_p_self(self, agent0: str, agent1: str, agent: int = 0, show: bool = T
rue)
 |      plots the p_self of a k-ToM agent in a defined agent pair
 |
 |      Args:
 |          agent0 (str): The name of agent0
 |          agent1 (str): The name of agent1
 |          agent (int, optional): An int denoting which of agent 0 or 1 you s
hould plot. Defaults to 0.
 |          show (bool, optional): Should plt.show be run at the end. Defaults
to True.
 |
 |  plot_score(self, agent0: str, agent1: str, agent: int = 0, show: bool = Tr
ue)
 |      plots the score of an agent in a defined agent pair
 |
 |      Args:
 |          agent0 (str): The name of agent0
 |          agent1 (str): The name of agent1
 |          agent (int, optional): An int denoting which of agent 0 or 1 you s
hould plot. Defaults to 0.
 |          show (bool, optional): Should plt.show be run at the end. Defaults
to True.
 |
 |  plot_tom_op_estimate(self, agent0: int, agent1: int, level: int, estimate:
str, agent: int = 0, plot: str = 'mean', transformation: Optional[bool] = Non
e, show: bool = True)
 |      plot a k-ToM's estimates the opponent in a given pair
 |
 |      Args:
 |          agent0 (str): The name of agent0
 |          agent1 (str): The name of agent1
 |          agent (int, optional): An int denoting which of agent 0 or 1 you s
hould plot. Defaults to 0.
 |          estimate (str): The desired estimate to plot options include:
 |             "volatility",
 |             "behav_temp" (Behavoural Temperature),
 |             "bias",
 |             "dilution".
 |          level (str): Sophistication level of the similated opponent you wi
sh to plot.
 |          plot (str, optional): Toggle between plotting mean ("mean") or var
iance ("var"). Default to "mean".
 |          show (bool, optional): Should plt.show be run at the end. Defaults
to True.
 |
 |  set_env(self, env: str) -> None
 |      Set environment of the agent group.
 |
 |      Args:
 |          env (str): The string for the environment you wish to set.
 |              Valid environment strings include:
 |              'round_robin': Matches all participant against all others
 |              'random_pairs': Combines the agent in random pairs (the number
of
 |              agent must be even)
 |
 |  ----------------------------------------------------------------------
 |  Data descriptors defined here:
 |
 |  __dict__
 |      dictionary for instance variables (if defined)
 |
 |  __weakref__
 |      list of weak references to the object (if defined)
```

Setting the type of interaction

The `.set_env()` method of the AgentGroup class allows you to set the type of 'tournament' that the agents will interact in. The possible strings that can be passed to the `env` input argument are:

- 'round_robin': Matches all agents against all others
- 'random_pairs': Combines the agents in random pairs (the number of agents must be even)

Assuming you have now created an `AgentGroup` object called `group`, the code cell below shows you how to set the environment to `'round_robin'`:

In [ ]:
```python
group.set_env(env='round_robin')
```

Running a tournament

The `.compete()` method of the AgentGroup class allows you to run a competition between the agents of the type that you've specified using the `.set_env()` method.

Assuming you have now created an `AgentGroup` object called `group`, the code cell below shows you how to run a tournament of the 'penny_competitive' game, where the group cometes for 50 simulations of 50 rounds (note that this takes a little while to run). The `.compete()` method returns a Pandas dataframe containing various results of the tournament. Below, this dataframe is saved in a variable called `results`.

In [ ]:
```
import warnings
warnings.simplefilter(action='ignore', category=FutureWarning) #Suppress future

results = group.compete(p_matrix='penny_competitive', n_rounds=50, n_sim=50, sa
```

Assuming the tournament has finished running, we can have a look at the structure of the `results` dataframe using the `.head` attribute of the Pandas dataframe, which gets the first 5 and last 5 rows of the dataframe:

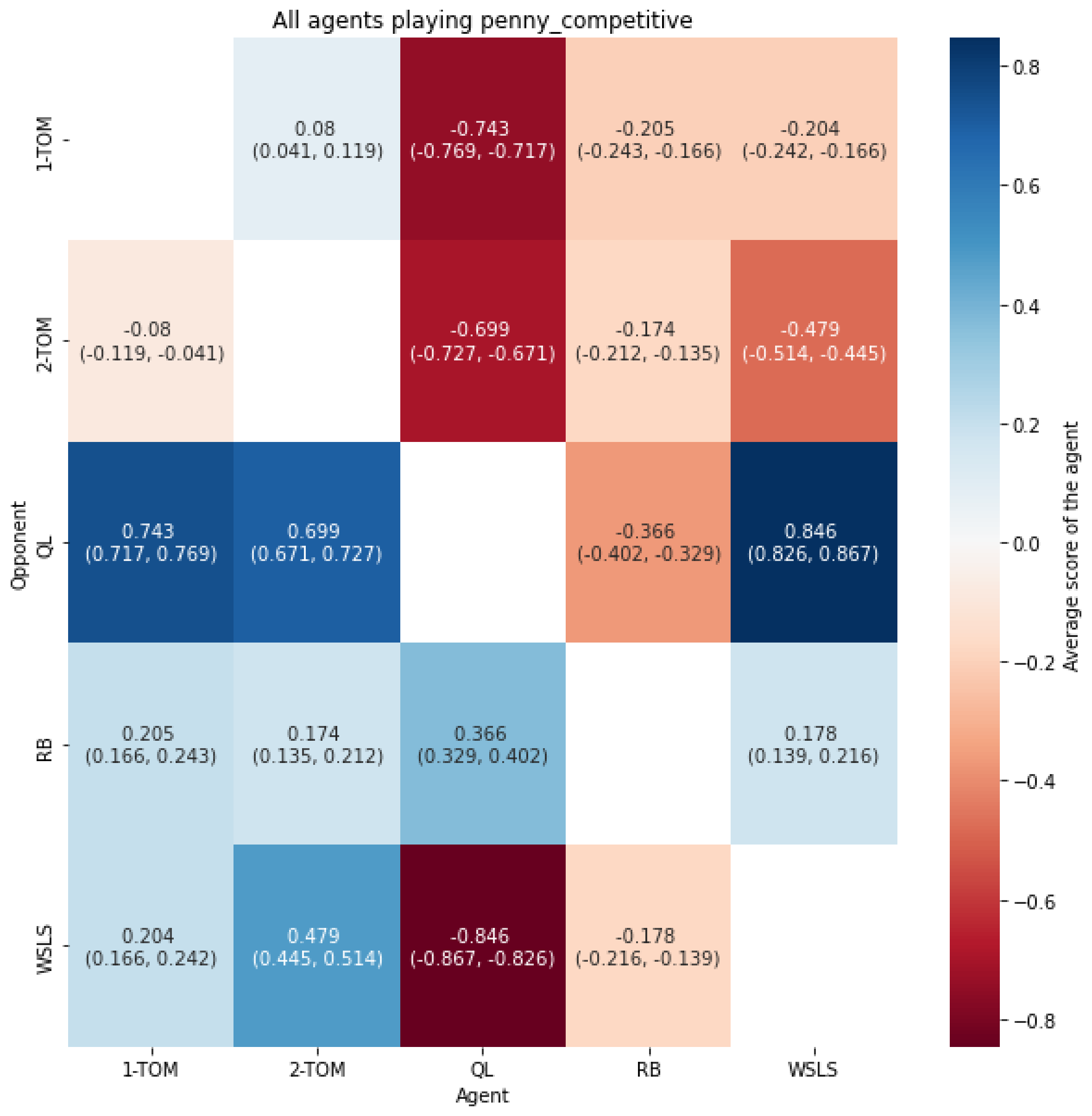
In [ ]:
```
print(results.head) # print the first row
```

```
<bound method NDFrame.head of        n_sim  round  choice_agent0  choice_agent1
payoff_agent0  \
0          0      0              0              0             -1
1          0      1              1              0              1
2          0      2              0              0             -1
3          0      3              1              0              1
4          0      4              0              0             -1
...      ...    ...            ...            ...            ...
2495      49     45              0              1              1
2496      49     46              1              1             -1
2497      49     47              1              0              1
2498      49     48              0              1              1
2499      49     49              1              0              1

      payoff_agent1                                    history_agent0  \
0                 1                                   {'choice': 0}
1                -1                                   {'choice': 1}
2                 1                                   {'choice': 0}
3                -1                                   {'choice': 1}
4                 1                                   {'choice': 0}
...             ...                                              ...
2495             -1  {'index': nan, 'choice': 0, 'internal_states':...
2496              1  {'index': nan, 'choice': 1, 'internal_states':...
2497             -1  {'index': nan, 'choice': 1, 'internal_states':...
2498             -1  {'index': nan, 'choice': 0, 'internal_states':...
2499             -1  {'index': nan, 'choice': 1, 'internal_states':...

                                         history_agent1 agent0 agent1
0     {'choice': 0, 'expected_value0': 0.5, 'expecte...     RB     QL
1     {'choice': 0.0, 'expected_value0': 0.5, 'expec...     RB     QL
2     {'choice': 0.0, 'expected_value0': 0.5, 'expec...     RB     QL
3     {'choice': 0.0, 'expected_value0': 0.5, 'expec...     RB     QL
4     {'choice': 0.0, 'expected_value0': 0.5, 'expec...     RB     QL
...                                                 ...    ...    ...
2495  {'index': nan, 'choice': 1, 'internal_states':...  1-TOM  2-TOM
2496  {'index': nan, 'choice': 1, 'internal_states':...  1-TOM  2-TOM
2497  {'index': nan, 'choice': 0, 'internal_states':...  1-TOM  2-TOM
2498  {'index': nan, 'choice': 1, 'internal_states':...  1-TOM  2-TOM
2499  {'index': nan, 'choice': 0, 'internal_states':...  1-TOM  2-TOM

[25000 rows x 10 columns]>
```

Plotting heatmap of tournament results

The `AgentGroup` class also comes with a number of plotting methods (use `help(ts.AgentGroup)` to inspect). The `.plot_heatmap()` method creates a heatmap of the rewards of all agents in the tournament, similar to Figure 3 (p. 17) in Waade et al. (2022). The code cell below demonstrates how to using this method (assuming the tournament has finished running):

In [ ]:
```
# Set background color of the outer
# area of the plt
plt.figure(facecolor='white')

plt.rcParams["figure.figsize"] = [10, 10] # Set figure size
plt.title("All agents playing penny_competitive") # Set figure title
group.plot_heatmap(cmap="RdBu")
```



This heatmap displays the average score across simulations for each competing pair. The score denotes the score of the agent (x-axis) when playing against the opponent (y-axis). The score in the parenthesis denotes the 95% confidence interval.

Exercise 4:

In the `'penny_competitive'` game, the `'2-TOM'` agent usually has a slight advantage over the `'1-TOM'` agent when they play against each other (see top-left corner of the heatmap). Write some code that focuses on interactions between `'1-TOM'` and `'2-TOM'` agents, in order to find out whether there are games among the predefined games in `tomsup` for which this is the other way around. That is, where the `'1-TOM'` agent outperforms the `'2-TOM'` agent when they interact with each other.

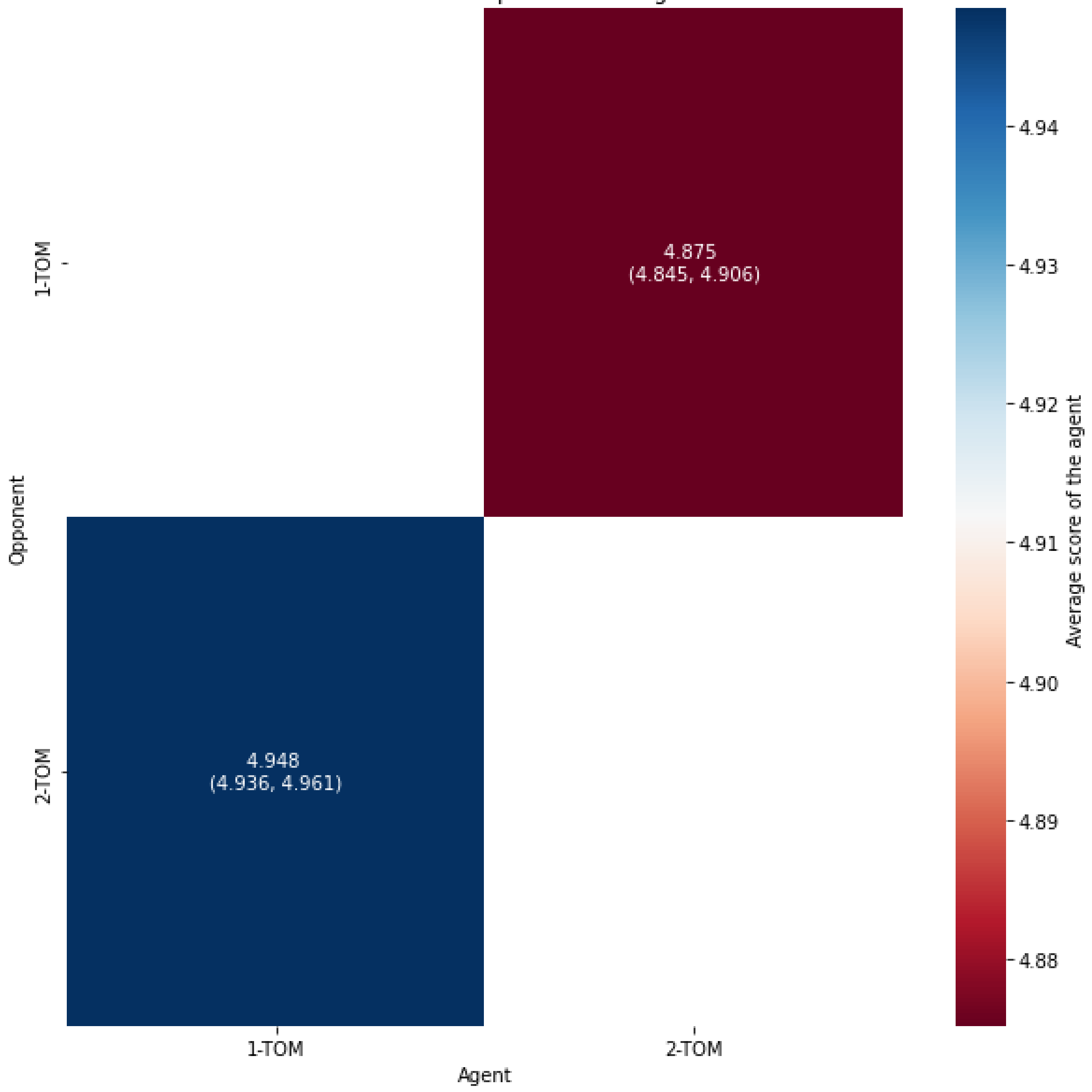
Tip 1: The `.compete()` method does a lot of printing as it's running. If this is getting in your way, you can switch off the printing by setting the `verbose` input argument of the

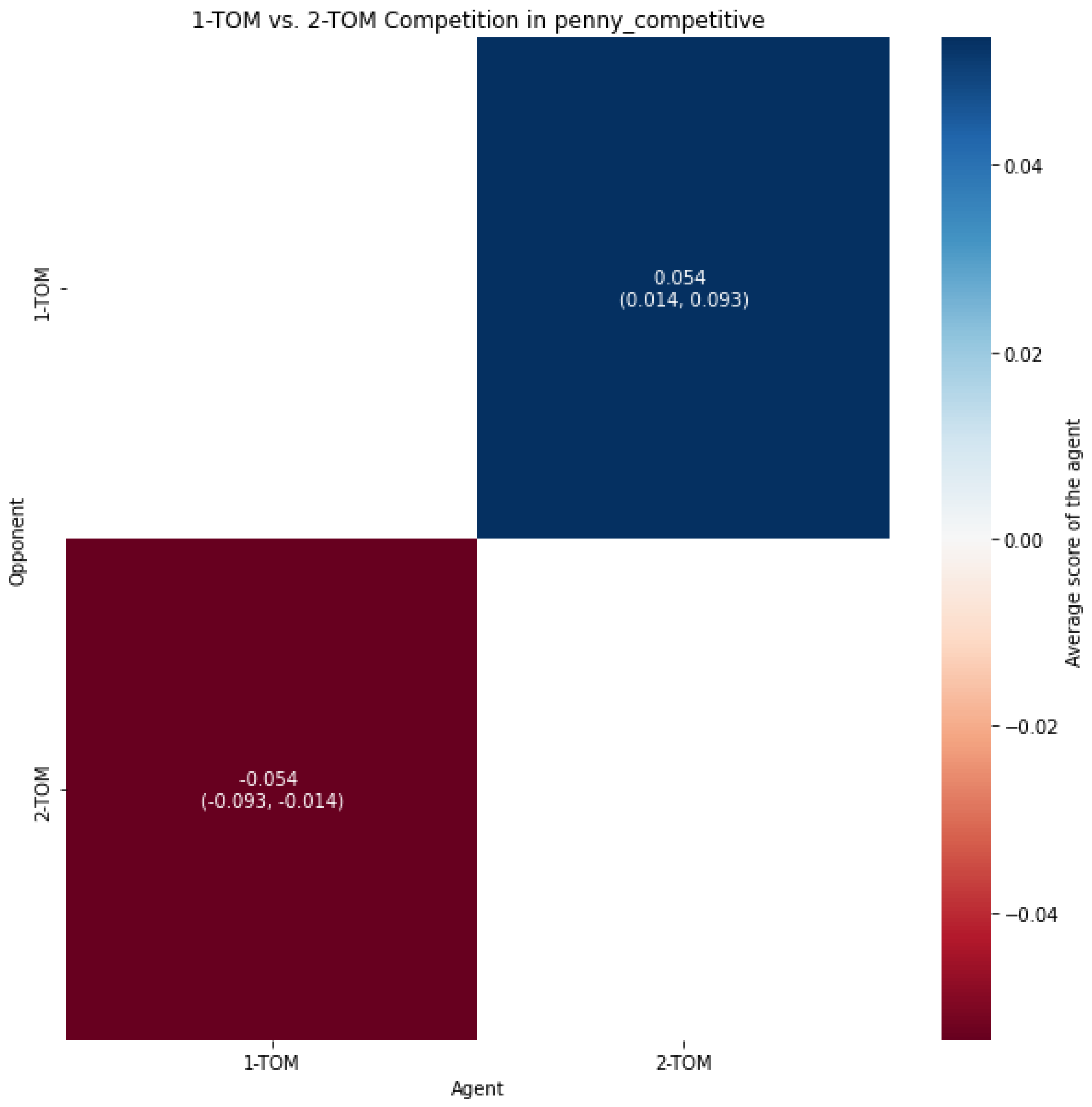
`.compete()` method to `False`.

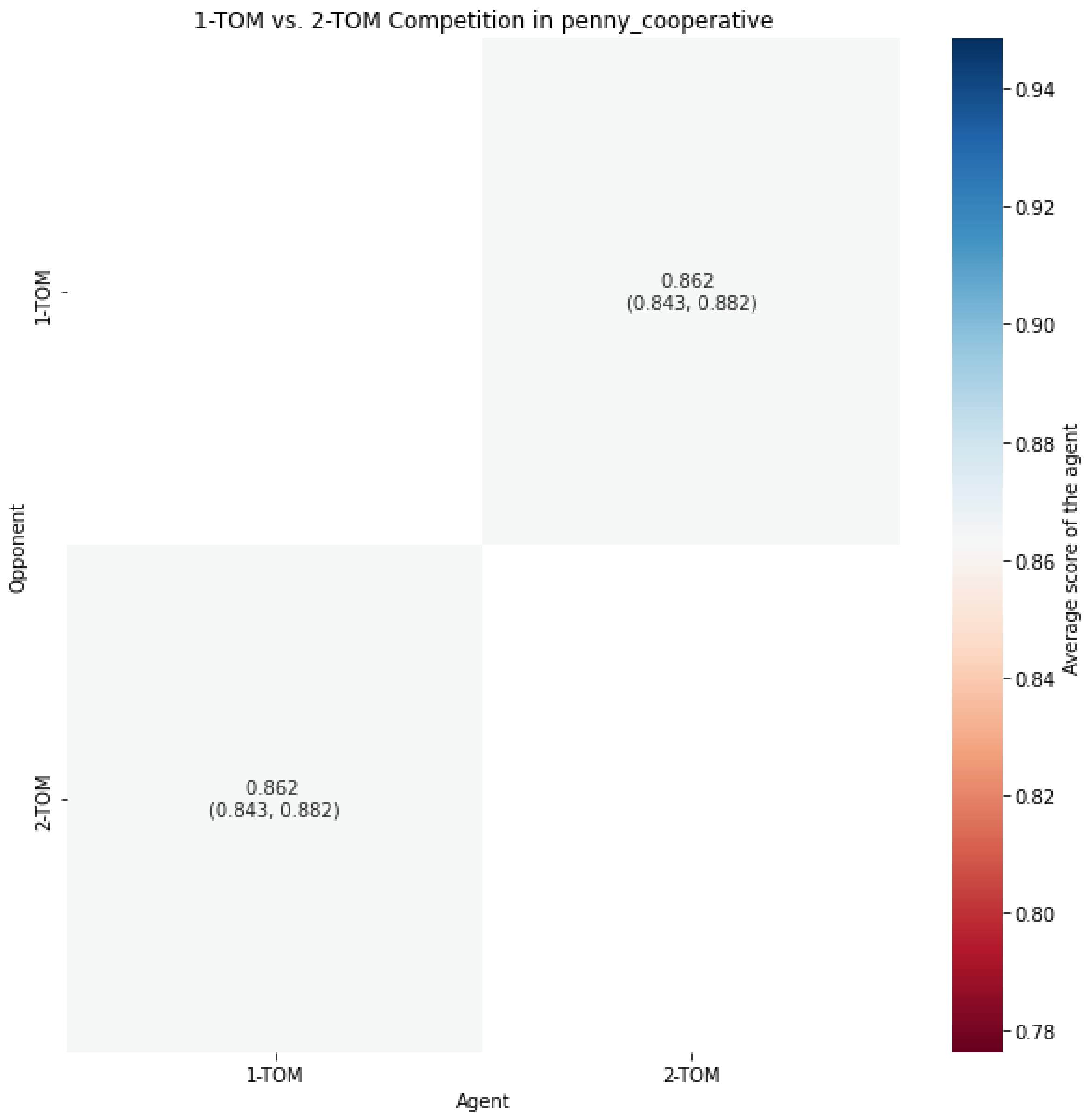
Tip 2: You can give a title to a figure using `plt.title("my_title_string")`.

```
In [ ]: #Suppress future warnings as they make the figures difficult to compare
        import warnings
        warnings.simplefilter(action='ignore', category=FutureWarning)

        #Create group
        agent_tom = ['1-TOM', '2-TOM']
        starting_parameters_tom = [{}, {}]
        group_tom = ts.create_agents(agent_tom, starting_parameters_tom)

        #Set up tornamanet environment for the group
        group_tom.set_env(env='round_robin')

        #Simulate the competition (25 sims & 25 rounds) and plot the scores for all gam
        game_names = ['staghunt', 'penny_competitive', 'penny_cooperative', 'party', 's
        for game_name in game_names:
            result_tom = group_tom.compete(p_matrix=game_name, n_rounds=50, n_sim=50, s

            plt.figure(facecolor='white')
            plt.rcParams["figure.figsize"] = [10, 10] # Set figure size
            plt.title(f"1-TOM vs. 2-TOM Competition in {game_name}") # Set figure title
            group_tom.plot_heatmap(cmap="RdBu")
```

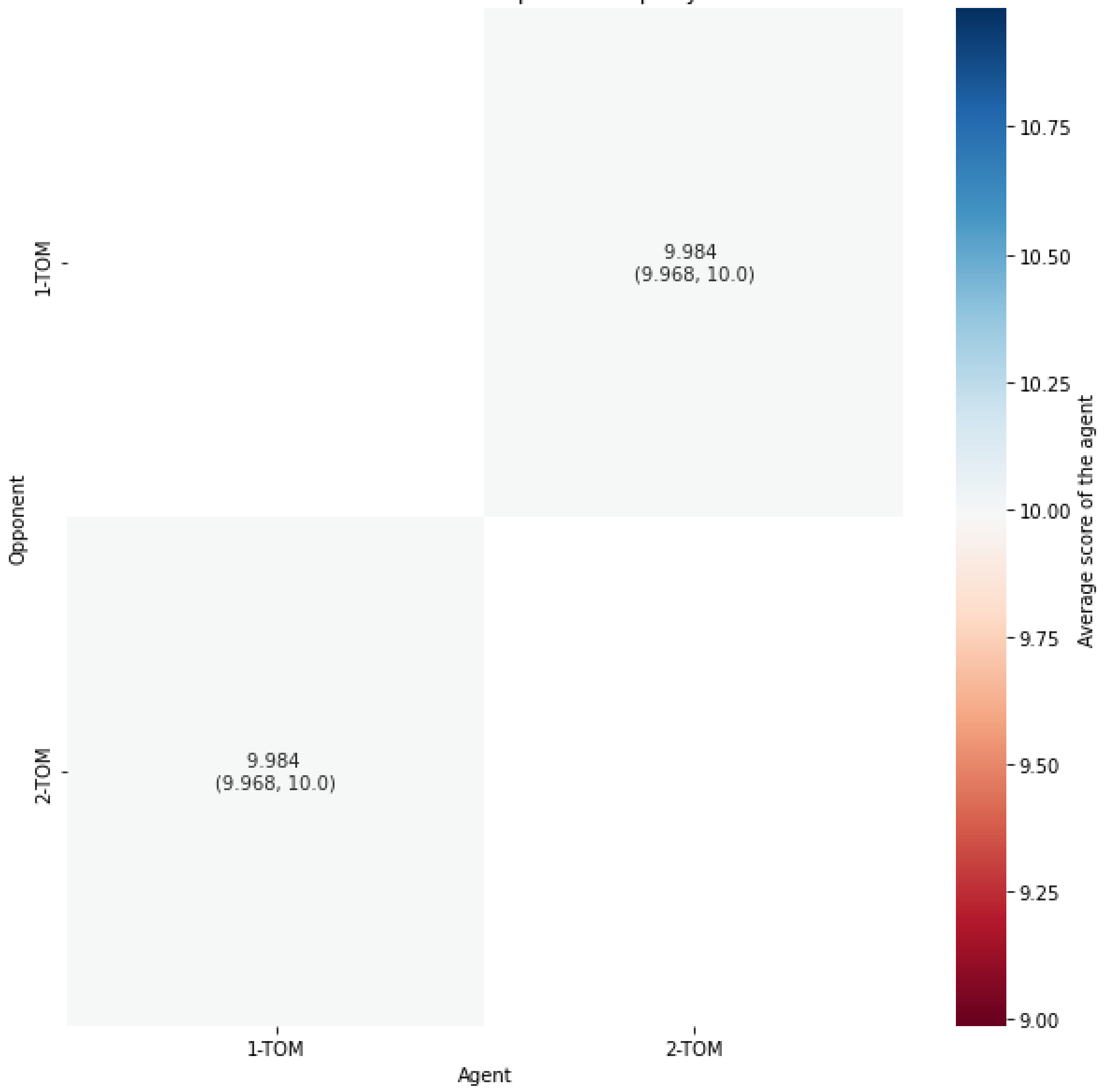
1-TOM vs. 2-TOM Competition in staghunt

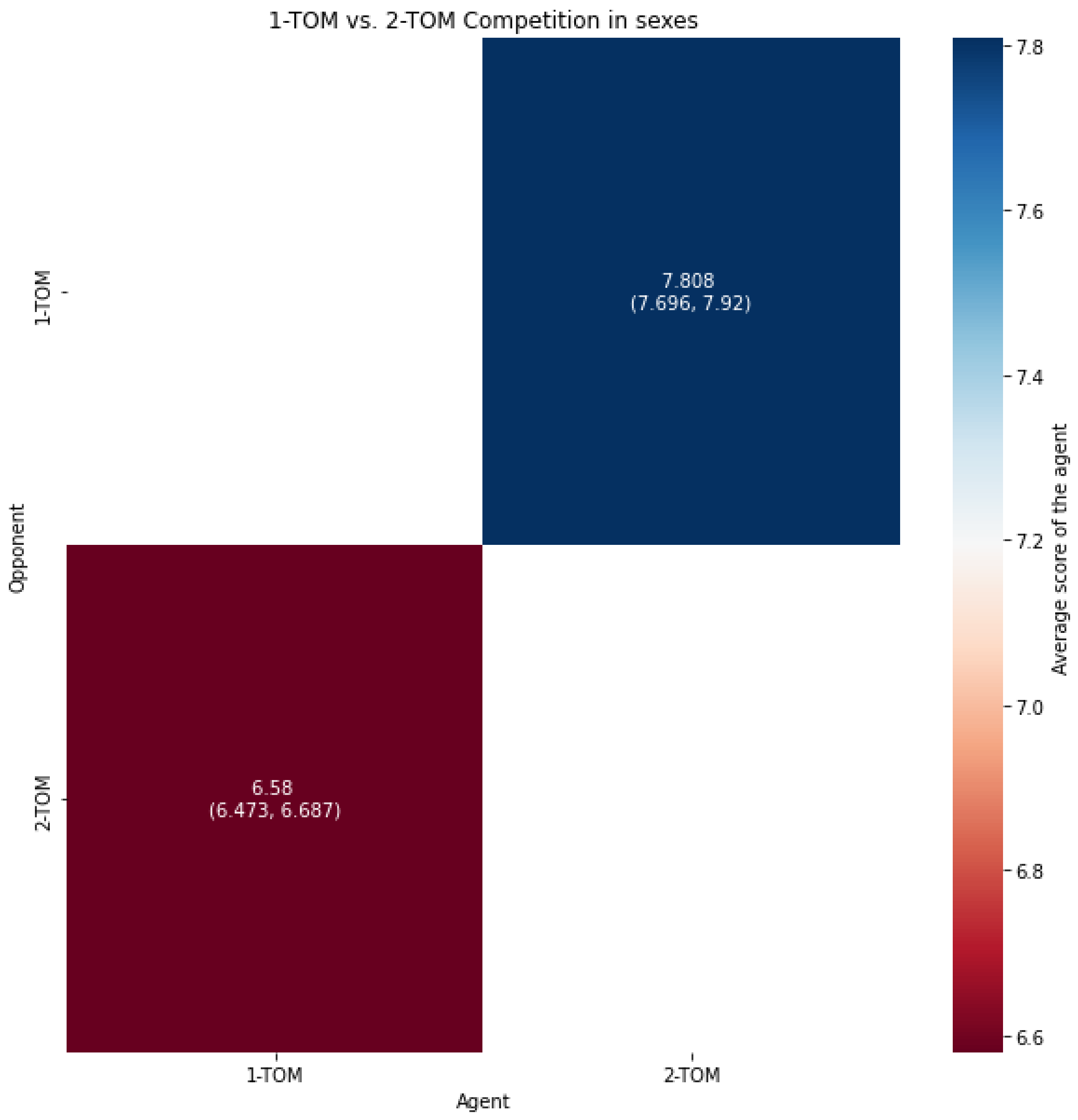



1-TOM vs. 2-TOM Competition in penny_competitive
0.054
(0.014, 0.093)
-0.054
(-0.093, -0.014)
1-TOM
2-TOM
Opponent
1-TOM
2-TOM
Agent
0.04
0.02
0.00
−0.02
−0.04
Average score of the agent

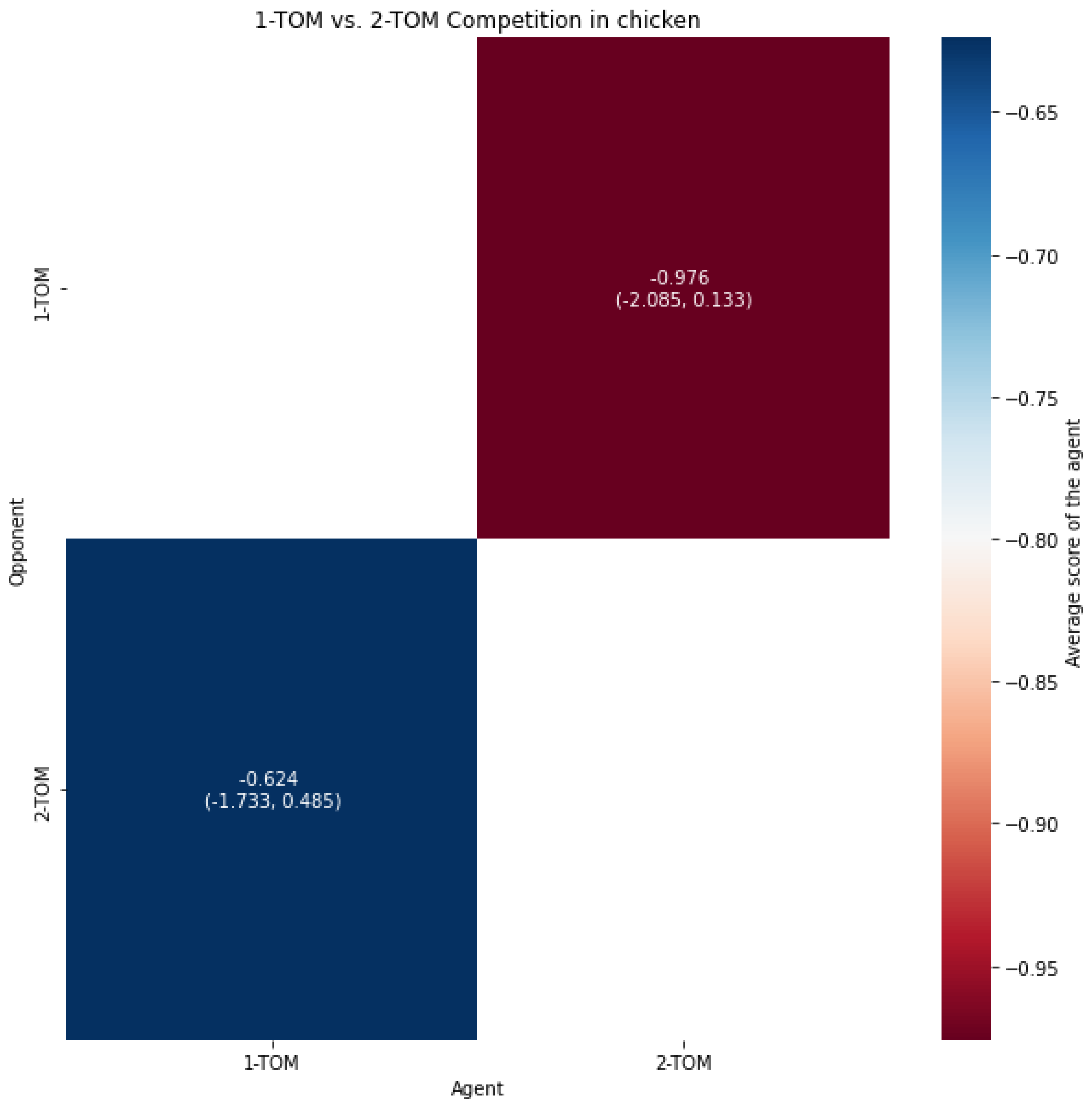

1-TOM vs. 2-TOM Competition in penny_cooperative
Opponent
1-TOM
2-TOM
Agent
1-TOM
2-TOM
0.862
(0.843, 0.882)
0.862
(0.843, 0.882)
Average score of the agent
0.94
0.92
0.90
0.88
0.86
0.84
0.82
0.80
0.78

1-TOM vs. 2-TOM Competition in party

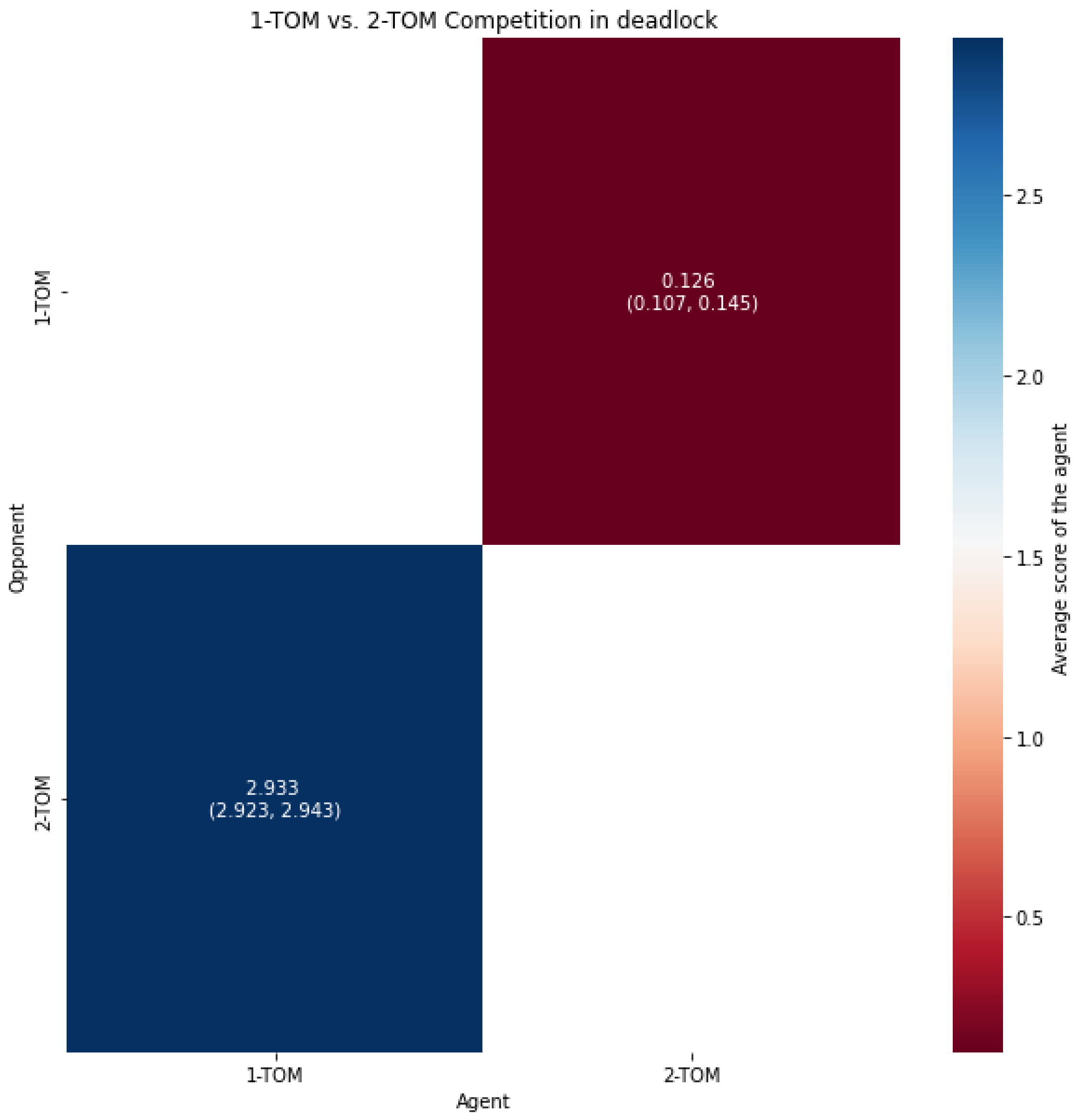


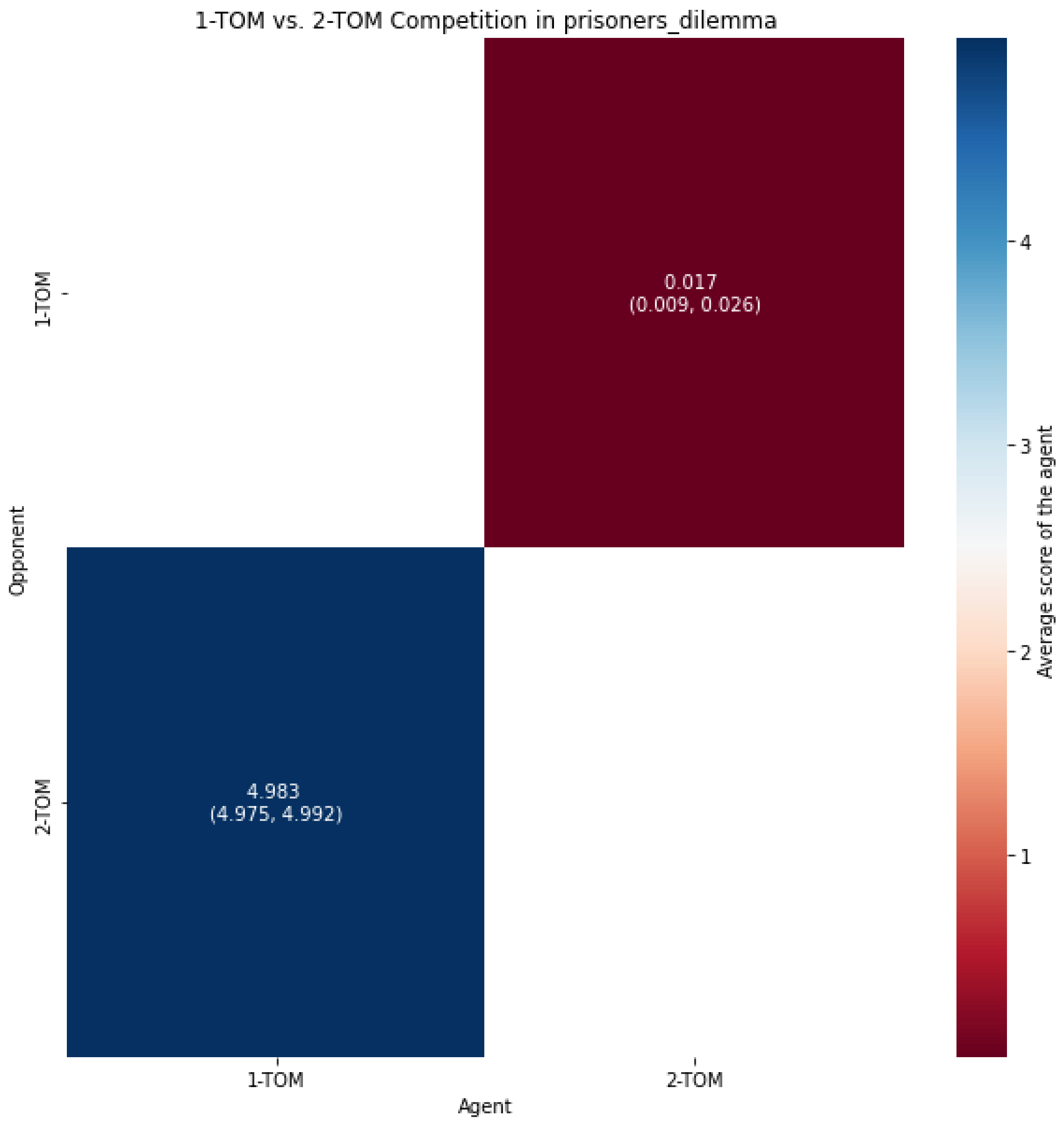


1-TOM vs. 2-TOM Competition in sexes
7.808
(7.696, 7.92)
6.58
(6.473, 6.687)
Opponent
1-TOM
2-TOM
Agent
1-TOM
2-TOM
Average score of the agent
7.8
7.6
7.4
7.2
7.0
6.8
6.6



1-TOM vs. 2-TOM Competition in chicken
-0.976
(-2.085, 0.133)
-0.624
(-1.733, 0.485)
1-TOM
2-TOM
Opponent
1-TOM
2-TOM
Agent
−0.65
−0.70
−0.75
−0.80
−0.85
−0.90
−0.95
Average score of the agent


1-TOM vs. 2-TOM Competition in deadlock
0.126
(0.107, 0.145)
2.933
(2.923, 2.943)
1-TOM
2-TOM
Opponent
1-TOM
2-TOM
Agent
2.5
2.0
1.5
1.0
0.5
Average score of the agent



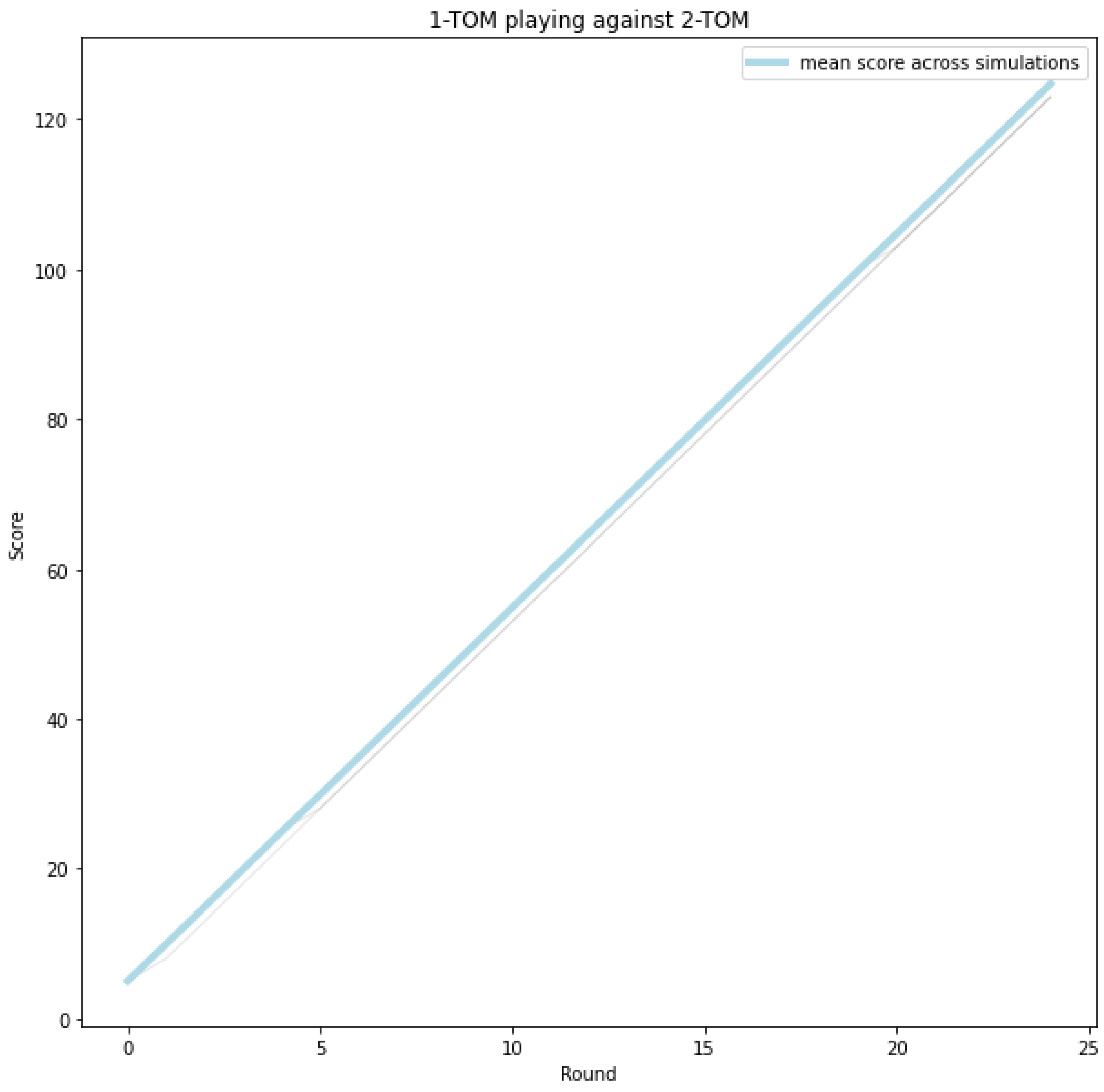
1-TOM agent outperforms 2-TOM agent in staguhunt, chicken, and deadlock and prisoners_dilemma

Plotting agents' scores over rounds:

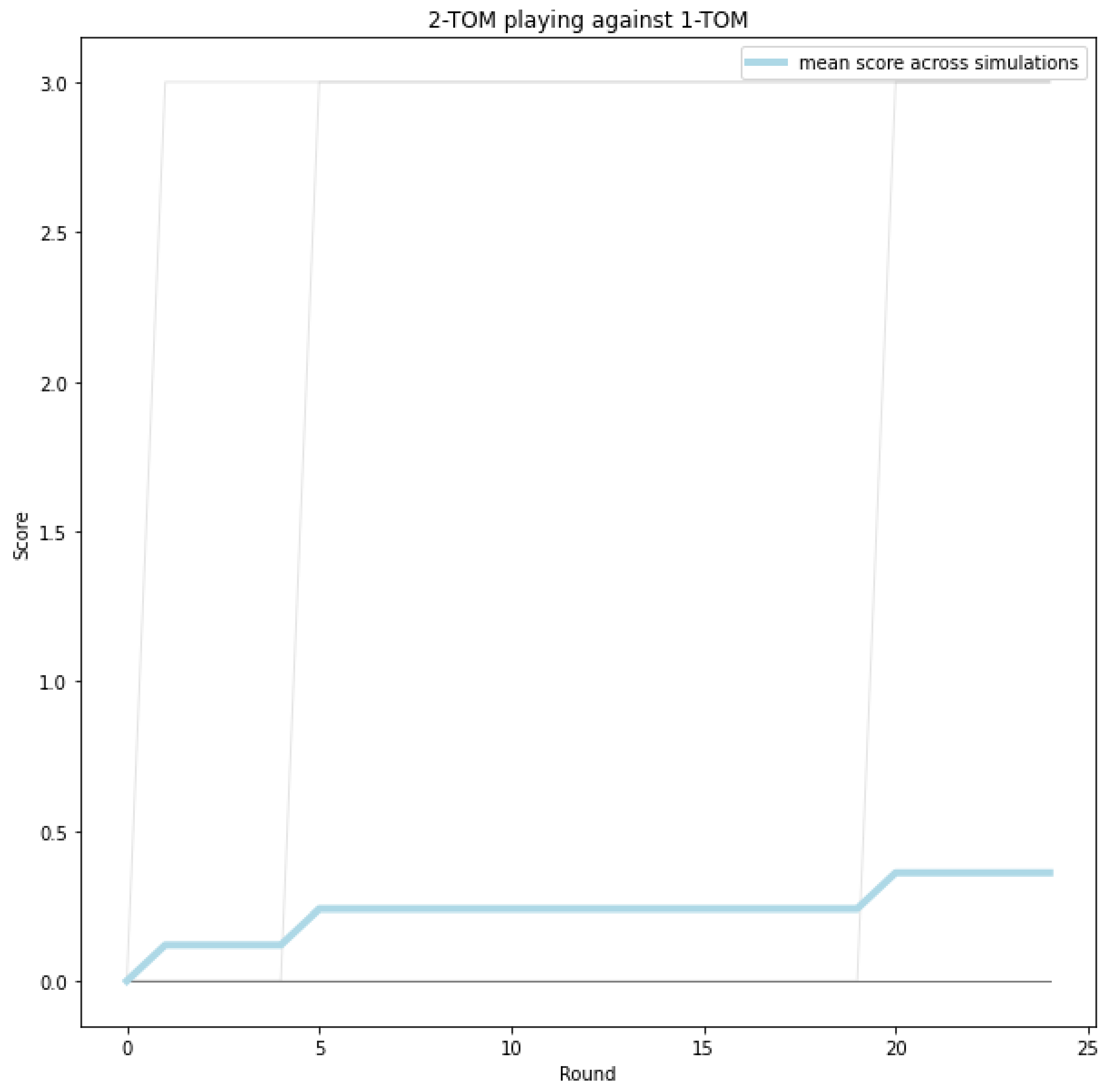
The `AgentGroup` class comes with a method `.plot_score()` which allows you to plot how the scores of the agents change over rounds (as they're learning about each other). Below is an example for how to use this method to plot the scores over time of the `'1-TOM'` agent when playing against the `'2-TOM'` agent.

```
In [ ]: group_tom.plot_score(agent0="1-TOM", agent1="2-TOM", agent=0)
        group_tom.plot_score(agent0="1-TOM", agent1="2-TOM", agent=1)
```

```
<Figure size 720x720 with 0 Axes>
```


1-TOM playing against 2-TOM
mean score across simulations
Score
Round
0
20
40
60
80
100
120
0
5
10
15
20
25

```
<Figure size 720x720 with 0 Axes>
```



As you can see, the `1-TOM` agent's mean score doesn't change much over time, but the individual simulations do start differing more from each other over rounds. In the text box below, write down your thoughts about what could be the cause of this.

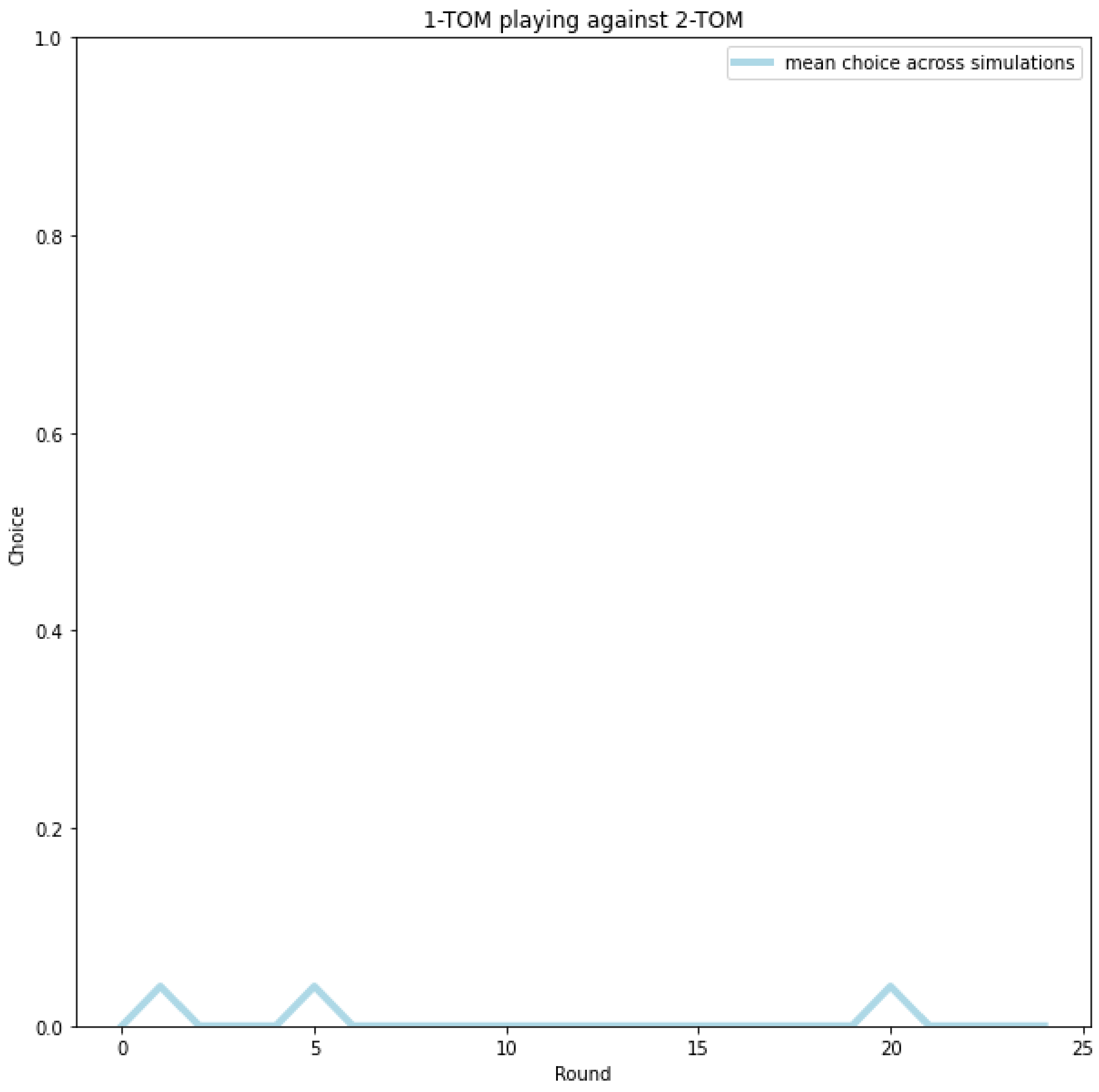
I believe the 1-TOM agent's mean score doesn't change much over rounds as they have equall chance to successufully estimate the 2-TOM agent's choice and fail to do so, and the chance of correctly estimating the oppenent's choice remains over time.

Plotting agent choices over rounds:

The `AgentGroup` class also comes with a method `.plot_choice()` which allows you to plot the choices of the agents across rounds. Below is an example for how to use this method to plot the choices of the `'1-TOM'` agent against those of the `'2-TOM'` agent.

```
In [ ]: group_tom.plot_choice(agent0="1-TOM", agent1="2-TOM", agent=0)
```

```
<Figure size 720x720 with 0 Axes>
```

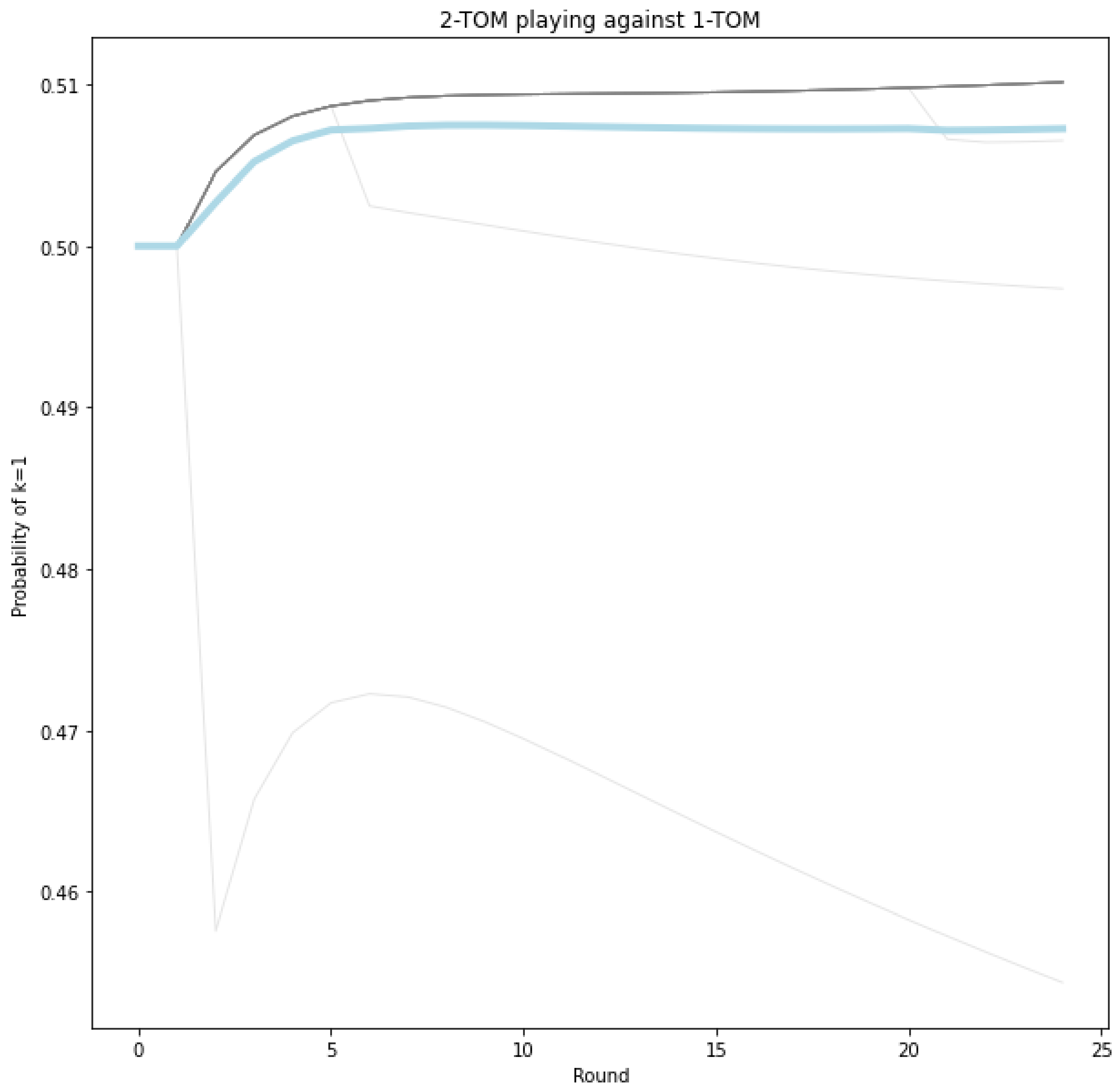


Plotting *k*-ToM agent's estimate of other agent's sophistication level (*k*):

The `AgentGroup` class also comes with a method `.plot_p_k()` which allows you to plot a given *k*-ToM agent's estimate of what the other agent's level of *k* is, over rounds. Below is an example for how to use this method to plot the probability that the `'2-TOM'` agent assigns to the possibility that the `'1-TOM'` agent that they are playing against has sophistication level of *k*=1, over rounds.

```
In [ ]: # The agent input argument specifies which agent's estimate should be shown (ag
        # The level input argument specifies for which level of k the probability shoul

        group_tom.plot_p_k(agent0="1-TOM", agent1="2-TOM", agent=1, level=1)
```

```
<Figure size 720x720 with 0 Axes>
```



Exercise 5:

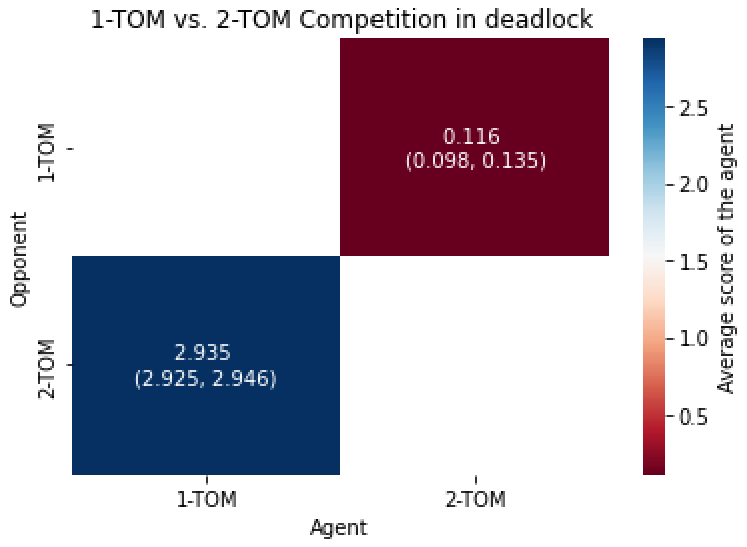
Choose one of the games that came out of Exercise 4 as a very clear example of a game where the `1-TOM` agent has a significant advantage over the `2-TOM` agent. For this particular game, try to figure out why this might be the case. Good first steps towards figuring this out are:

1. Inspect the pay-off matrix of the game in question
2. Plot the scores of the two agents over rounds **when playing the game in question**
3. Plot the choices that the two agents make **when playing the game in question**
4. Plot the agents' estimates of each other's sophistication levels (*k*) **when playing the game in question**

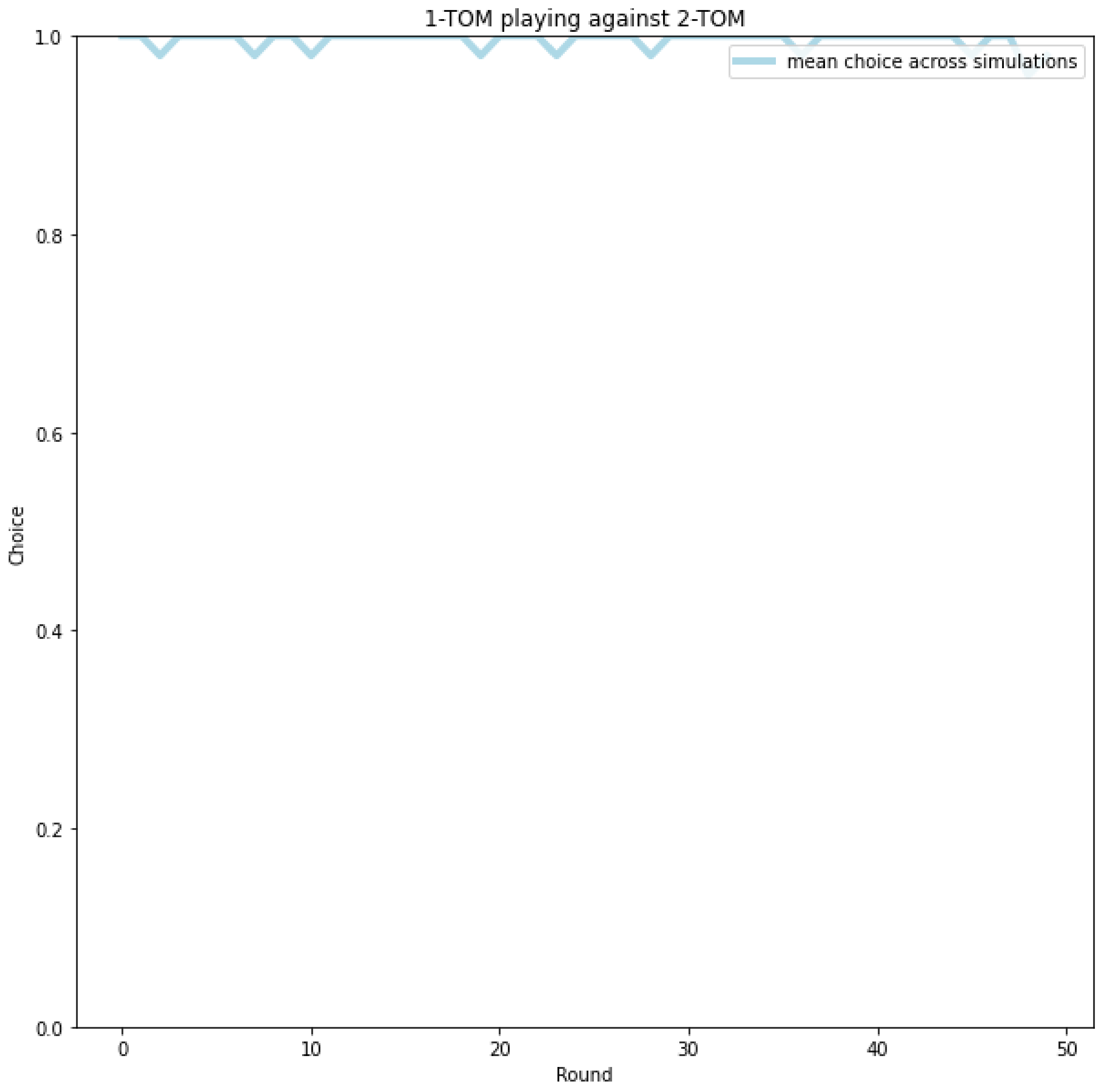
```
In [ ]: #Looking at the plots of 1-TOM agent vs. 2-TOM agent competition,
        #deadlock seems to be the game where 1-TOM agent outperforms 2-TOM agent the mo
        #*I thought until today's lecture that scores in prisoners dilemma corresponded
        #That is why I didn't choose prisoners dilemma.
        game_name = 'deadlock'
```

```
###[Inspect the pay-off matrix of the game in question]
print(ts.PayoffMatrix(name = game_name))

#The payoff matrix suggests that choosing Choice 1 is mutually beneficial for b
#although the agent gets the highest reward when the opponent chooses Choice 0


###[Plot the scores of the two agents over rounds when playing the game in ques
import warnings
warnings.simplefilter(action='ignore', category=FutureWarning)

agent = ['1-TOM', '2-TOM']
starting_parameters = [{}, {}]
group_ex5 = ts.create_agents(agent, starting_parameters)
group_ex5.set_env(env='round_robin')

results = group_ex5.compete(p_matrix=game_name, n_rounds=50, n_sim=50, save_his
plt.figure(facecolor='white')
plt.rcParams["figure.figsize"] = [10, 10] # Set figure size
plt.title(f"1-TOM vs. 2-TOM Competition in {game_name}") # Set figure title
group_ex5.plot_heatmap(cmap="RdBu")

#Again, 1-TOM agent outperforms 2-TOM agent


###[Plot the choices that the two agents make when playing the game in questior
group_ex5.plot_choice(agent0="1-TOM", agent1="2-TOM", agent=0)
group_ex5.plot_choice(agent0="1-TOM", agent1="2-TOM", agent=1)

#1-TOM agent tends to choose Choice 1, while 2-TOM agent tends to choose Choice


###[Plot the agents' estimates of each other's sophistication levels (_k_) wher
group_ex5.plot_p_k(agent0="1-TOM", agent1="2-TOM", agent=1, level=0)
group_ex5.plot_p_k(agent0="1-TOM", agent1="2-TOM", agent=1, level=1)
group_ex5.plot_p_k(agent0="1-TOM", agent1="2-TOM", agent=0, level=0)
```

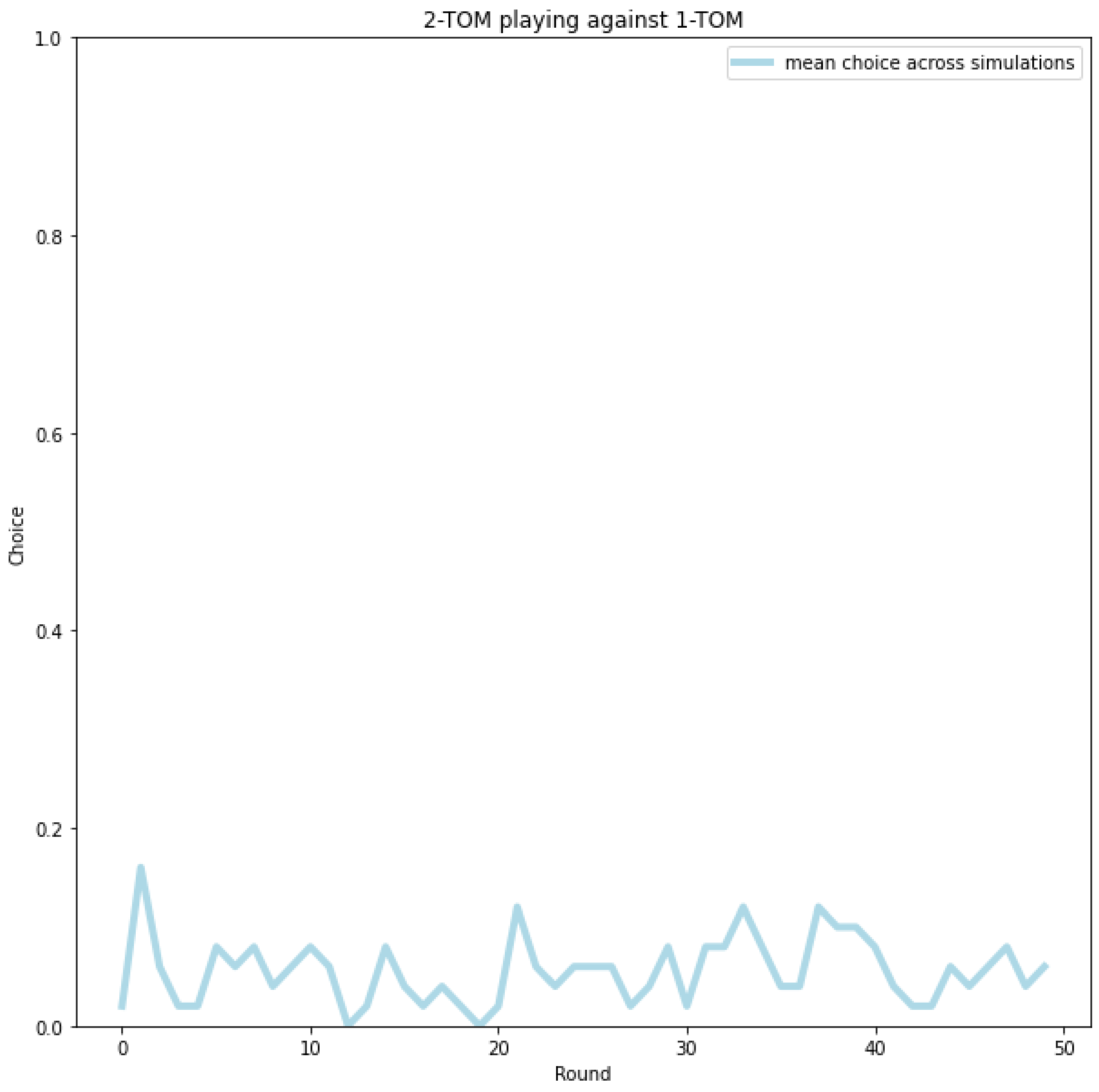
```
<Class PayoffMatrix, Name = deadlock>
The payoff matrix of agent 0
       |  Choice agent 1
       |   | 0 | 1 |
       | --------- |
Choice | 0 | 1 | 0 |
agent 0| 1 | 3 | 2 |

The payoff matrix of agent 1
       |  Choice agent 1
       |   | 0 | 1 |
       | --------- |
Choice | 0 | 1 | 3 |
agent 0| 1 | 0 | 2 |
```


1-TOM vs. 2-TOM Competition in deadlock
0.116
(0.098, 0.135)
2.935
(2.925, 2.946)
Opponent
1-TOM
2-TOM
Agent
1-TOM
2-TOM
Average score of the agent
2.5
2.0
1.5
1.0
0.5

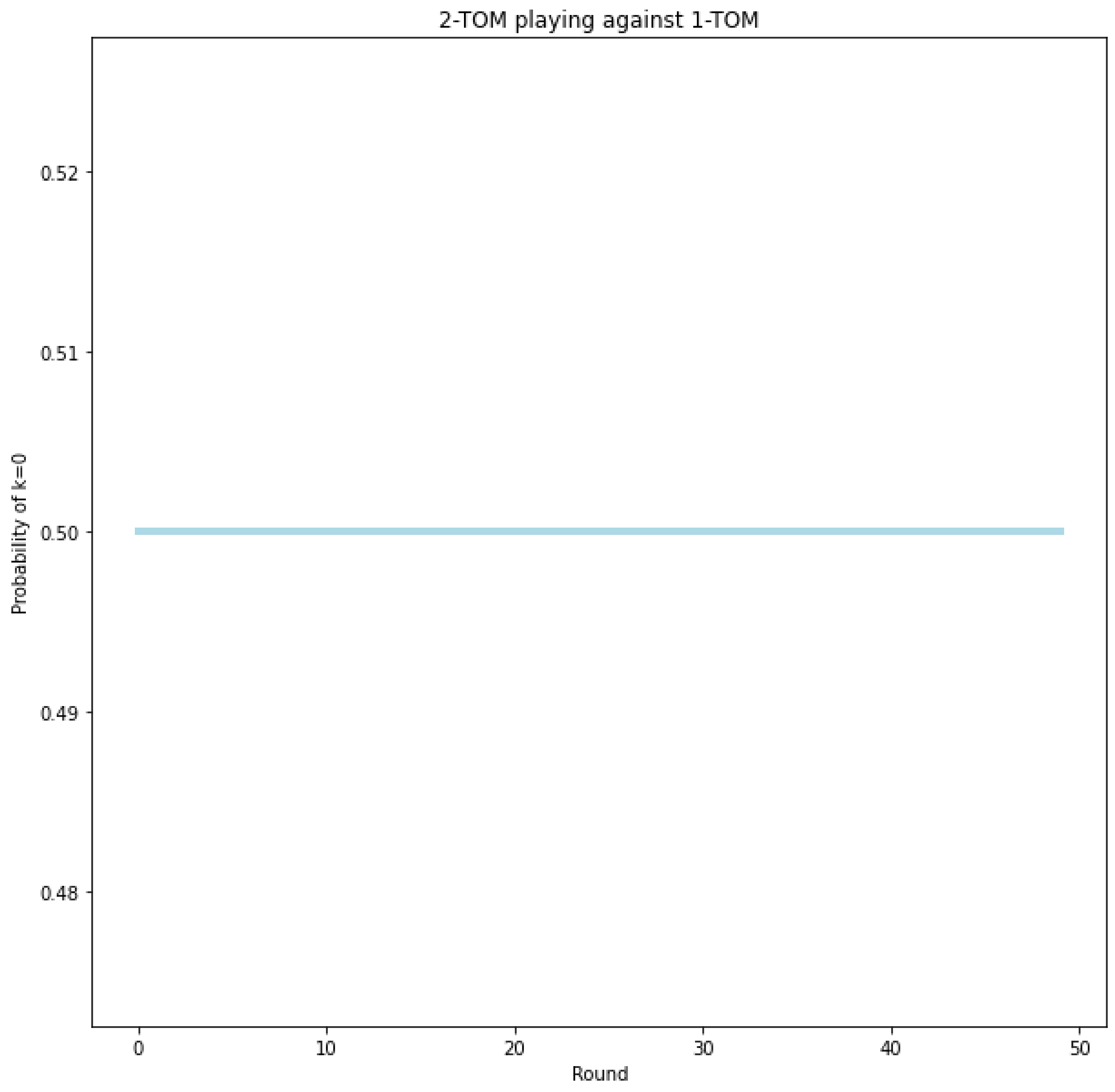
```
<Figure size 720x720 with 0 Axes>
```


1-TOM playing against 2-TOM
mean choice across simulations
Choice
Round
1.0
0.8
0.6
0.4
0.2
0.0
0
10
20
30
40
50

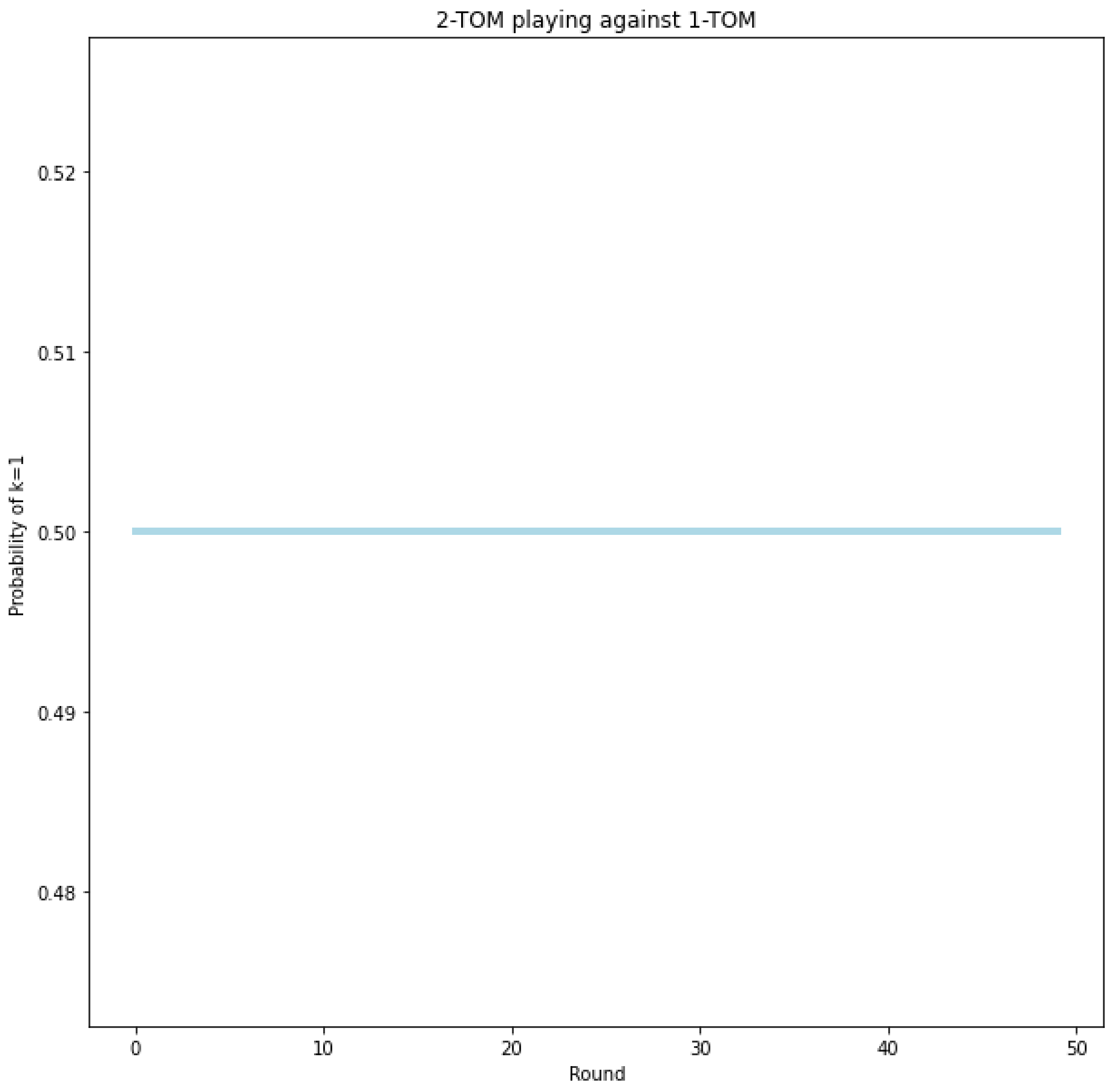
```
<Figure size 720x720 with 0 Axes>
```


2-TOM playing against 1-TOM
mean choice across simulations
Choice
Round
1.0
0.8
0.6
0.4
0.2
0.0
0
10
20
30
40
50

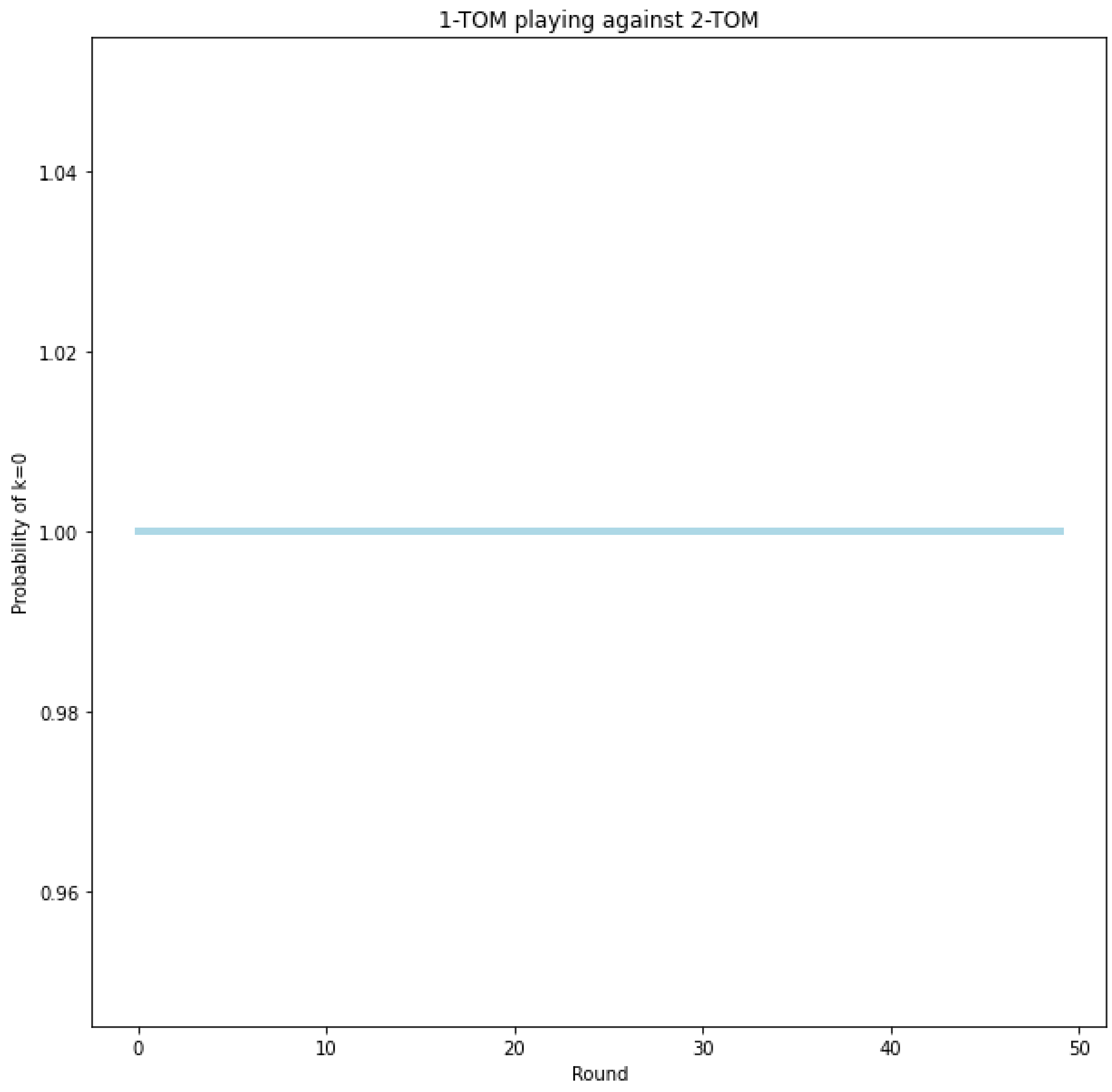
```
<Figure size 720x720 with 0 Axes>
```


2-TOM playing against 1-TOM
Probability of k=0
0.52
0.51
0.50
0.49
0.48
0
10
20
30
40
50
Round

```
<Figure size 720x720 with 0 Axes>
```


2-TOM playing against 1-TOM
Probability of k=1
0.52
0.51
0.50
0.49
0.48
0
10
20
30
40
50
Round

```
<Figure size 720x720 with 0 Axes>
```



I am unsure why 1-TOM agent outperforms 2-TOM agent in this game, but one thing I can say is that 1-TOM agent gets higher points than 2-TOM agent because they almost always choose Choice 1, which results in either 2 or 3 points depending on the oppent's choice. However, 2-TOM agent is more likely to choose Choice 0, which result in either no point or 1 point. This is puzzling because choosing Choice 1 will lead to higher points regardless of the opponent's choice, but somehow 2-TOM agent tends to do so. This could be because 1-TOM agent prioritizes in earning higher points, but 2-TOM agents prioritizes in estimating the oppoent's choice over earing more points.

I am not sure as to why the estimation of k is not changing over time.

BONUS Exercise 6 (only if you have time left):

Continuing with the same game you inspecting for Exercise 5, have a look at what happens when the `1-TOM` and `2-TOM` agent interact with the 'random bias' agent ('RB').

Use the `.plot_tom_op_estimate()` of the `AgentGroup` class to inspect how the `2-TOM` agent estimates the `RB` agent's bias over time. Does the `2-TOM` agent reach the accurate inference eventually. You may want to run more rounds to make sure that the model has converged (i.e., that the `2-TOM` agent's estimate of the bias is no longer changing).

```
In [ ]:
```