The Iterated Prisoner's Dilemma

COMP3710 Artificial Intelligence

April 10, 2019

Benjamin Davidson

Samar El-Houssami

Suraiya Khandakar

Mio Tanaka

Abstract

In this paper, we apply a Genetic Algorithm to find the best finite machine state against a collection of other strategies from the Axelrod collection—ALLD, ALLC, TFT, Random, and Grudger. This project uses Jupyter Notebooks, Python 3.7, the Axelrod-Python and the Axelrod-Dojo library to run, analyze, and visualize an Iterated Prisoner's Dilemma Tournament and introduce machine learning strategies with finite state machines. We managed to outperform most of the strategies we trained with and tested on. We found that Cooperator is at a big disadvantage to Defector. Defector does very well against Cooperator and Random and is the only strategy with all positive scores. Also, FSM Player is very similar to Tit-for-tat; however, it performs better against the Random strategy.

Table of Contents

Introduction

Since its inception in the 1950's, the Prisoner's Dilemma has been borrowed and adapted from Game Theory and applied to many different fields of study including political science, psychology, economics and more recently, computer science. In this game, two players, Ann and Bob, are criminals who have been caught, and are forced to decide about whether to confess or not without getting a chance to consult with their partner in crime. There are three possible outcomes for the two prisoners:

1) If only one prisoner defects—confesses and accuses the other—the confessor is pardoned while the accused is sentenced to twenty years in prison,
2) If both prisoners defect—both confess and accuse the other—they each get ten years in prison, and lastly,
3) If both prisoners cooperate as in neither confesses, they each receive a lessened sentence of only one year in prison.

Whereas a rational analysis will lead both Ann and Bob to defect, this "dominant" strategy yields suboptimal result—ten years in prison versus one year if they both choose irrationally to cooperate. Hence, acting unilaterally in one's own best interest is not the best option in this case. If the players had the possibility of interacting in future games, however, it would make more sense for the players to cooperate unilaterally. The Iterated Prisoner's Dilemma (IPD), which involves multiple iterations of this game, creates such a scenario.

In computer simulations, the IPD is basically an infinitely iterated game because the players are unaware of how many iterations are left in the game. Being aware of the endpoint of the game can affect a player's cooperation. For instance, if the player knows it's the last round,

he is most likely going to defect. Similarly, the player is also more likely to defect in the second last round of the game and might be more likely to defect in all rounds. Thus, making the endpoint unknown to the players allows computer scientists to study cooperation behaviour without the end point affecting the player's decisions. In IPD, each player can make future decisions based on the memory of the past three or more iterations, and strategies are deterministic, where the players will always make the same move for a given history 'h'. This history includes both the player's own moves and the partner's moves. With four possible outcomes in each move in three rounds of history, the total number of moves in each strategy totals to 64 moves. What strategies work best in IPD given a pattern of moves in memory has been one of the main pursuits of interest in Artificial Intelligence (AI) research.

One of the most important contributions to this field was made by David Axelrod, who arranged two tournaments in the early 1980's, where human designed strategies which competed against each other. While his tournaments declared victory for Tit-for-Tat (TFT), there has been many new developments since then, and new analyses suggest that TFT is not necessarily the best strategy for every situation (Rapaport, Seale, & Coleman, 2015). In fact, in order to win a tournament, a player must figure out the opponent's strategy and come up with a strategy that is best suited for the situation. This can be studied using a Genetic Algorithm. Genetic algorithm tries to mimic evolutionary processes in order to optimize a particular function on some space of candidates of solutions.

Motivation and Contribution

The IPD has been studied extensively in AI since the 1990's (Woodbridge, 2012). The multiple but indefinite number of iterations allows for a more in-depth exploration of the

evolution of the decision-making process. This is because the agent now gets a chance to make future decisions based on the opponent's past decisions in memory. Thus, the IPD allows computer scientists to better understand multi-agent decision making, where an agent can take into account another agent's behaviour (Woodbridge, 2012).

Although data from Axelrod's first tournaments suggested that TFT was the best 'general purpose' strategy, Rapoport and colleagues (2015) reanalyzed data from Axelrod's first tournament along with more newly available data and found that the original conclusion does not hold up. They concluded instead that the efficacy of TFT depends on the design of the tournament, the success criterion, and the particular values that were chosen for the Prisoner's Dilemma Payoff Matrix (Rapoport et. al, 2015). In this case study, we apply genetic algorithm to find the best Finite State Machine against a collection of other strategies from the Axelrod library—Always Defect (ALLD), Always Cooperate (ALLC), TFT, Random, and Grudger. These strategies were purposefully selected in order to cover a whole range of cooperation and defection behaviours. The goal is to allow the finite state machine to make use of the best of a whole range of strategies.

Relevant Literature Review

In Axelrod's (1984) first experiment, fourteen computer programs played against each other for 200 rounds of IPD. Some of the strategies that participated in this tournament were Tit-for-Tat (TFT), Always Cooperate (ALLC), Always Defect (ALLD), and Random. The winner was Anatol Rapaport's Tit-for-Tat (TFT). In Axelrod's second experiment, sixty-two programs competed, and the winner again was TFT. In TFT, a player begins with cooperation in the first move, and always chooses the partner's last move in all subsequent rounds. There are

several features that make TFT a strong mechanism. It can be said that it is a nice strategy because it begins with cooperation, but at the same time, it is also a regulatory mechanism, as it punishes defection with defection. Furthermore, it is forgiving as it continues to cooperate after opponent cooperates. Also, this is a very predictable strategy, as the opponent can easily guess the next move, which makes it easier to attain mutual benefit. One problem, however, is that TFT operates on the assumption that the opponent is trying to maximize his own score, which is not true in the case of a Random strategy. As a result, when playing against Random, TFT can sink to its opponent's level.

Some variations of TFT arose such as Tit-for-Two-Tat (TF2T) and Suspicious Tit-for-Tat (STFT), Generous TFT, Hard TFT, Hard TF2T. To discuss a few of the variations, TF2T is the same as TFT except it begins with cooperation on the first two moves, and on subsequent moves, it requires the opponent to defect at least twice before replying with defection. TF2T outperforms TFT when the opponent's first move is defection. Also, cooperating even after opponent's first defection causes him to cooperate as well. This results in both the players earning more points. STFT, on the other hand, is a strategy that always defects on the first move, and replicates opponent's last move on all subsequent moves. Thus, STFT is similar to TFT in that both strategies follow a course of action that copies the opponent's last move. The only difference is that STFT is not "nice" as it begins with defection. STFT is prone to getting stuck in infinite defection loops. As a result, STFT generally performs worse than TFT, though it does outperform TFT if the opponent's first move is defection. Thus, if the opponent's first move is defection, STFT outperforms TFT, and TF2T outperforms TFT. Despite the emergence of the many variations, TFT was regarded to be the best 'general purpose' strategy based on the results from Axelrod's first two tournaments (Rapoport, Seale, & Coleman, 2015). This notion has since evolved.

In contrast, strategies like Free Rider (ALLD) and Always Cooperate (ALLC) do not fare as well. In ALLD, a player always chooses to defect no matter what the opponent's last move was. This is a dominant strategy against an opponent who tends to cooperate more. In ALLC, a player always chooses to cooperate no matter what the opponent's last turn was. This strategy can not only be abused by a Free Rider strategy, but also by a strategy that leans towards cooperation. On the other hand, Grudger, also known as Grim (Jurisi, Kermek, & Konecki, 2012) cooperates until the first defection, after which it continues to defect. The reason for including Grudger is that it scores well against random strategies and nice strategies but performs poorly against defecting strategies because there is no way to re-establish cooperation when playing against defecting strategies (Jurisi et. al, 2012). Thus, the strategies selected for our training algorithm in our study covers a good spectrum of the cooperation and defection ratios.

Case Study

In this case study, we apply genetic algorithm to find the best Finite State Machine against a collection of other strategies from the Axelrod library—Always Defect (ALLD), Always Cooperate (ALLC), TFT, Random, and Grudger. These strategies were intentionally selected in order to cover a whole range of cooperation behaviours ranging from ALLC, which cooperates 100% of the time to Random, which cooperates 50% of the time to ALLD, which cooperates 0% of the time, while Random and Grudger fall in between in their cooperation frequencies.

Experimental setup and methodology

This project uses Jupyter Notebooks, Python 3.7, the Alexrod-Python and the Axelrod-Dojo library to run, analyze and visualize an Iterated Prisoners Dilemma Tournament and

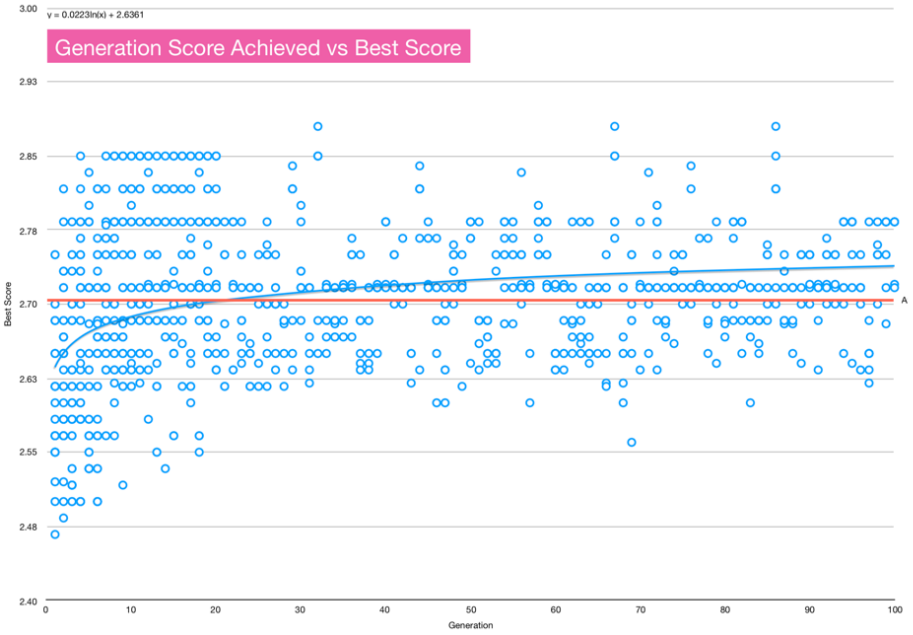
introduce machine learning strategies with finite state machines. We made some minor modifications to the dojo library to improve the reporting and output. This is most reflected in the `training_output.csv` which now records detailed information about the players used in the simulation, mutation rate, bottleneck, size of state machine and the date/time to aid in reproducing the results. We also made minor modifications to the main Axelrod library (`player.py`) to keep strategy name short so they would display correctly in charts and graphs. For further details, please see the Appendix.

Discussion

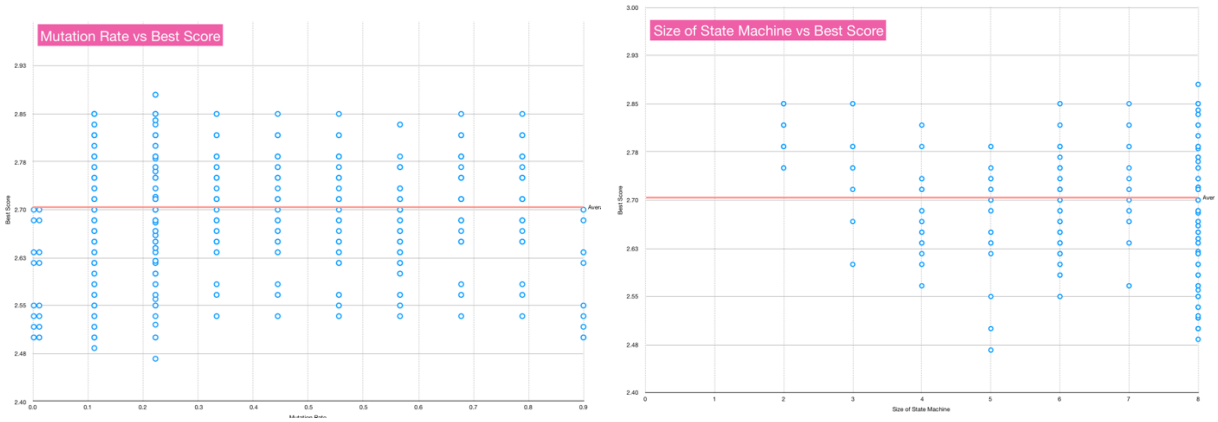
Using the `axelrod-dojo` library, we will able to use Genetic Algorithms to do evolutionary training on selected strategies. By adjusting the numbers of states in our finite state machine, the size of our population, the bottleneck and mutation probability we are able to run 49 tests with over 1600 distinct evolutions. Each evolution was recorded in `training_output.csv` allowing further analysis.

```
Scoring Generation 10
     → Mean score: 2.44, Root variance: 0.155
     Generation  10 |  Best Score:  2.783333 State: 0:C:0_C_0_C:0_D_1_D:1_C_1_D:1_D_1_D
     Generation  10 | Worst Score:  1.800000 State: 0:C:0_C_1_D:0_D_1_C:1_C_0_C:1_D_1_C
```

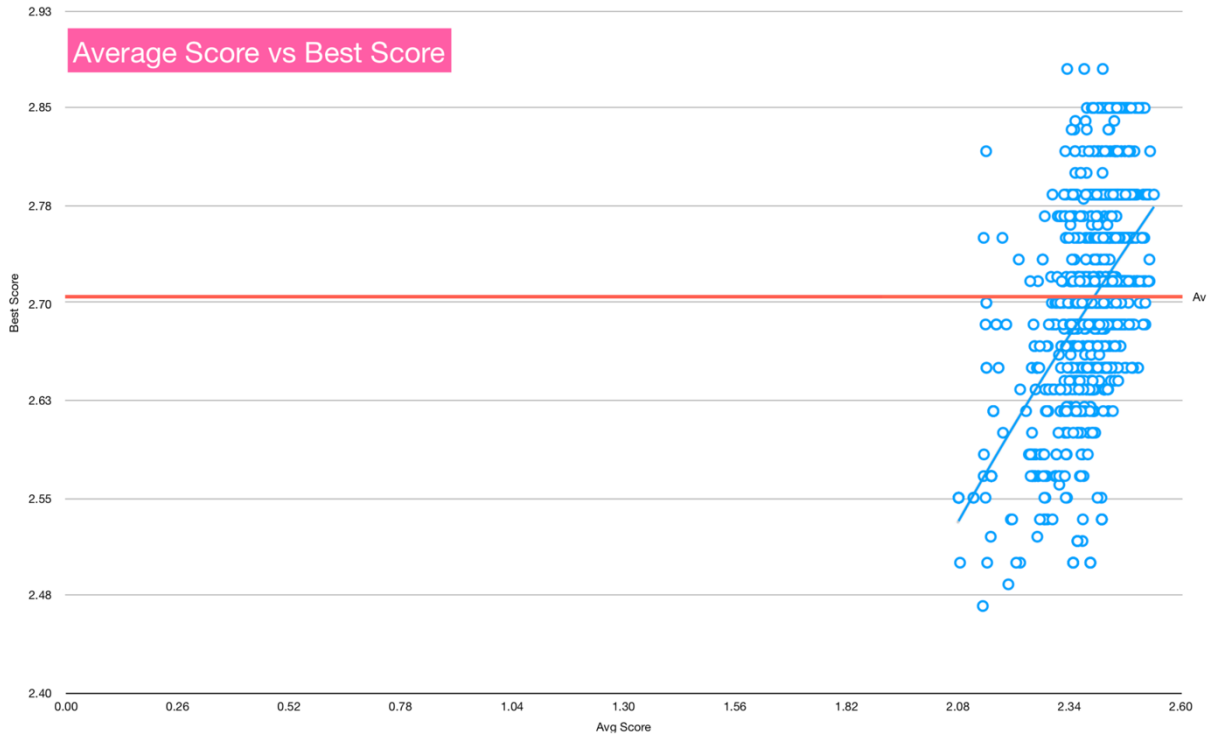
We found that optimal results could occur at any point during evolution but became more like as the number of generations increased. After approximately twenty generations we observed little improvement.



Except at very low mutation rates (<0.05) we observed little improvement from adjusting the mutation rate. The size of the state machine also seemed to have little effect.



There appears to be a strong connection between the average score of the population in a generation and the best score the population achieves.



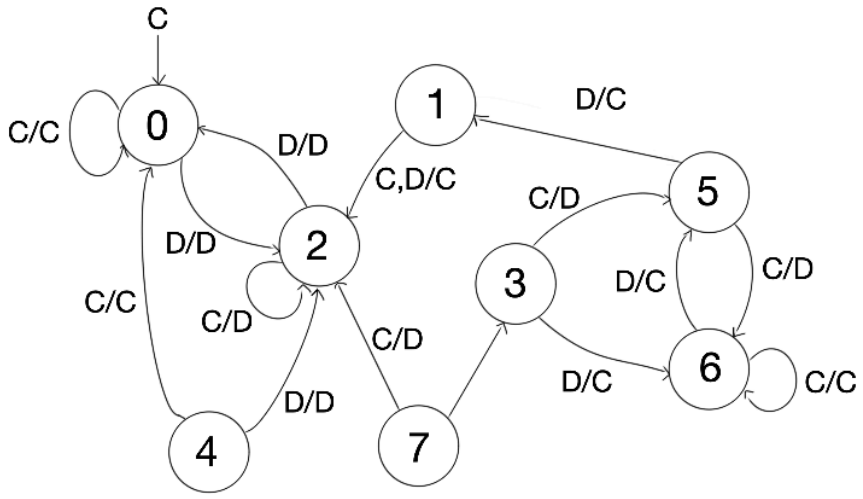
Our top ten results all game from state machines with eight states, mutation rates between 0.1-0.2. Bottlenecks were typically set to 10% of the population size but this varied depending on the test.

G	Mean	pstdev	Best	#State	Size	Mutation	Bottleneck
32	2.37	0.185	2.88	8	50	0.2	5
67	2.33	0.213	2.88	8	50	0.2	5
86	2.42	0.175	2.88	8	50	0.2	5
11	2.43	0.182	2.85	8	400	0.1	10
7	2.38	0.191	2.85	8	1000	0.1	100
9	2.40	0.190	2.85	8	1000	0.1	100
11	2.41	0.200	2.85	8	1000	0.1	100
16	2.41	0.203	2.85	8	1000	0.1	100
17	2.41	0.205	2.85	8	1000	0.1	100

We were able to take our most successful finite state machine and place it in a tournament with other prototypical strategies.

Our Most Successful Finite State Machine

```
0:C:0_C_0_C:0_D_2_D:1_C_2_C:1_D_2_C:2_C_2_D:2_D_0_D:3_C_5_D:3_D_6_C
:4_C_0_C:4_D_2_D:5_C_6_D:5_D_1_C:6_C_6_C:6_D_5_C:7_C_2_D:7_D_3_C
```

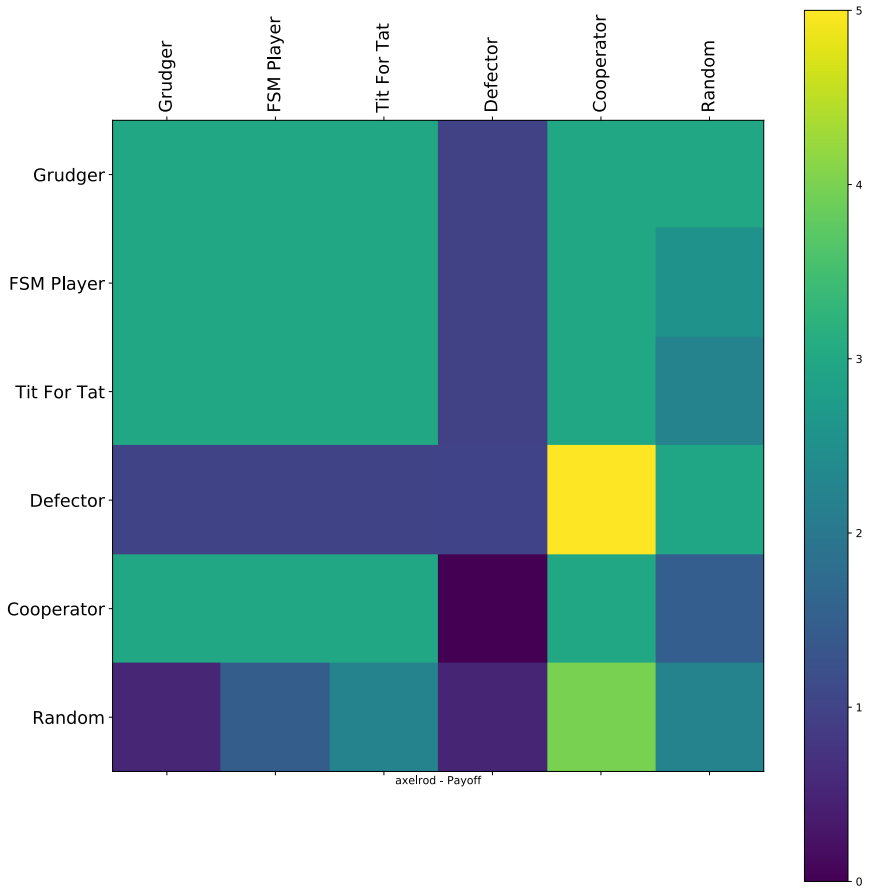


Our most successful finite state machine

We managed to outperform five of the six strategies we trained with and tested on. We found that Cooperator is at a big disadvantage to Defector, Defector does very well against Cooperator, and Random is the only strategy with all positive scores.

The derived FSM Player (Our FSM) is somewhat similar to Tit-For-Tat. However, unlike TFT, the strategy does not go back to cooperating after the opponent has defected. After a single defect by the opponent, the FSM will defect even when the opponent cooperates. This ends only after the opponent defects once more, which 'restarts' the strategy, i.e. makes it return

to the initial state and cooperate when opponent cooperates. This strategy performs better than TFT against the Random player. For more details, please see Appendix.



Tournament Payoff Matrix

Conclusion

AI is fundamentally all about decision-making, where an agent is deemed intelligent if it can reach an optimal decision in any given situation after taking into account all possible outcomes in its external environment (Woodbridge, 2012). For this reason, the IPD has proven to be a very important and relevant topic of study because it mimics real world decision-making, which usually requires the consideration of others' choices that may be unknown to an agent. The Iterated Prisoner's Dilemma, thus, provides a very interesting platform to study the evolution of decision making in order to identify the best strategy for an agent to assume. In this project, we successfully trained and tested a FSM to play against some of the most common strategies known to the IPD. After 100 generations, we ended up with a strategy similar to Tit-For-Tat, but yields a higher median score of 2.85-2.88.

Future Work

We only trained and tested a selected few strategies while there are many more that could be trained and tested. Due to time limitations, we did not go past 100 generations and a population size of 10000. In future work, we could expand on the number of generations and the population size. Again, due to time limitation, population sizes between 2500-10000 were untested. Another question that we could explore is why we could not get past a score of 2.85-2.88, and to find out if there is a hard limit. It must be possible to do better because other algorithms in our test beat ours.

References

W. Ashlock and D. Ashlock, "Changes in Prisoner's Dilemma Strategies Over Evolutionary Time With Different Population Sizes," 2006 IEEE International Conference on Evolutionary Computation, Vancouver, BC, 2006, pp. 297-304.
doi: 10.1109/CEC.2006.1688322
URL: http://ieeexplore.ieee.org.ledproxy2.uwindsor.ca/stamp/stamp.jsp?tp=&arnumber=1688322&isnumber=35623

Harper, Marc et al. "Reinforcement Learning Produces Dominant Strategies for the Iterated Prisoner's Dilemma." Ed. Yong Deng. PLOS ONE 12.12 (2017): e0188046. Crossref. Web.

Jurisi, M., Kermek, D., & Konecki, M. (2012). A review of iterated prisoner's dilemma strategies. *Proceedings of the 35th International Convention.*

Rapoport, A., Seale, D. A., Colman, A. M. (2015). Is Tit-for-Tat the Answer? On the Conclusions Drawn from Axelrod's Tournaments. *PLoS One, 10*(7), e0134128.

Wooldridge, M. (2012). The triumph of rationality. *IEEE Intelligent Systems, 27*(1), 60-64.

Appendix

IteratedPrisonersDilemma - GitHub

https://github.com/tukanuk/IteratedPrisonersDilemma

This project, source code, reports and source material is available at IteratedPrisonersDilemma.

Prerequisites

- Python 3
- pip

Anyone can get started by going to Python.org and downloading the latest version.

If you are on a Mac, Homebrew is the best way to get started.

In Terminal;

```
/usr/bin/ruby -e "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/master/install)"
brew install python
```

Installation

Depending on your setup, you use either `python` and `pip` **OR** `python3` and `pip3` for the installation. On my system , that allows me to distinguish between using Python 2.x or Python 3.x. Your setup may be different.

To determine your setup, type `python --version`. If the response is `3.x.x` you can probably use `python` and `pip`.

First clone the repository into the directory of your choice:

```
cd your/directory/of/choice
git clone https://github.com/tukanuk/IteratedPrisonersDilemma.git
```

A quick `ls` of the repository and you should see something like this:

```
...
-rw-r--r--   1 username  admin    1129  6 Apr 15:43 requirements.txt
...
```

`requirements.txt` is a file containing all the project dependencies.

```
pip3 install -r requirements.txt
```

The important dependencies for this project are:

```
Axelrod
Juptyer
```

This will install all the requirements onto your system. Depending on what you already have installed, this may take several minutes.

If that all went according to plan you should have a working copy of Jupyter Notebook and the Axelrod library.

Try it out

Launch Jupyter Notebook from the project directory

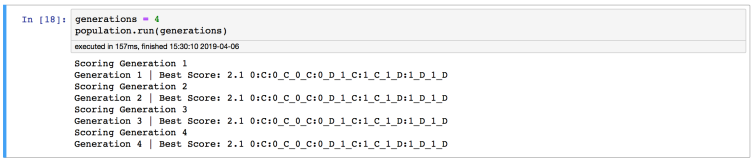
```
jupyter notebook
```

Your web browser should launch and you should be greeted with a browser to allow you to open one of the `.ipynb` files.

`Iterated_Prisoners_Dilemma_Project.ipynb` is the basic notebook that has our working tournament.

The `Iterated_Prisoners_Dilemma_Project_GA.ipynb` notebook is the work-in-progress genetic algorithm.

Yeah Genetic Algorithms!



IPD Evolutionary Training

Group Project for COMP 3710

By: Mio Tanaka, Suraiya Khanda, Samar Houssami, Ben Davidson

This project uses Jupyter Notebooks, Python 3.7, the Alexrod-Python and the Axelrod-Dojo library to run, analyize and visualise an Iterated Prisioners Dilemma Tournament and introduce machine learning strategies with finite state machines.

We made some minor modifications to the dojo library to improve the reporting and output. This is most reflected in the `training_output.csv` which now records detailed information about the players used in the simulation, mutation rate, bottleneck, size of state machine and the date/time to aid in reproducing the results.

We also made minor modifications to the main Axelrod library (`player.py`) to keep strategy name short so they would display correctly in charts and graphs

```
# import IPython
# from IPython.core.display import display, HTML
# display(HTML("<style>.container { width:100% !important; }</style>"))
```

If you are doing serious testing uncomment this block and widen your browser to see the full output and retain clean line breaks

Import the axelrod library

```
import axelrod as axl
%matplotlib inline

from datetime import datetime
print("Run at: " + datetime.now().strftime('%Y-%m-%d %H:%M:%S'))
```

```
Run at: 2019-04-10 19:24:48
```

The parameters that we are working with

Finite State Machine Evolver

```
Usage:
    fsm_evolve.py [-h] [--generations GENERATIONS] [--population POPULATION]
```

```
    [--mu MUTATION_RATE] [--bottleneck BOTTLENECK] [--processes PROCESSORS]
    [--output OUTPUT_FILE] [--objective OBJECTIVE] [--repetitions REPETITIONS]
    [--turns TURNS] [--noise NOISE] [--nmoran NMORAN]
    [--states NUM_STATES]

Options:
    -h --help                   Show help
    --generations GENERATIONS   Generations to run the EA [default: 500]
    --population POPULATION     Population size  [default: 40]
    --mu MUTATION_RATE          Mutation rate [default: 0.1]
    --bottleneck BOTTLENECK     Number of individuals to keep from each generation [default: 10]
    --processes PROCESSES       Number of processes to use [default: 1]
    --output OUTPUT_FILE        File to write data to [default: fsm_params.csv]
    --objective OBJECTIVE       Objective function [default: score]
    --repetitions REPETITIONS   Repetitions in objective [default: 100]
    --turns TURNS               Turns in each match [default: 200]
    --noise NOISE               Match noise [default: 0.00]
    --nmoran NMORAN             Moran Population Size, if Moran objective [default: 4]
    --states NUM_STATES         Number of FSM states [default: 8]
```

Import dojo

```
import axelrod_dojo as dojo
objective = dojo.prepare_objective(name="score", turns=10, repetitions=1)

params_class = dojo.FSMParams
# params_class = dojo.HMMParams
params_kwargs = {"num_states": 2}
```

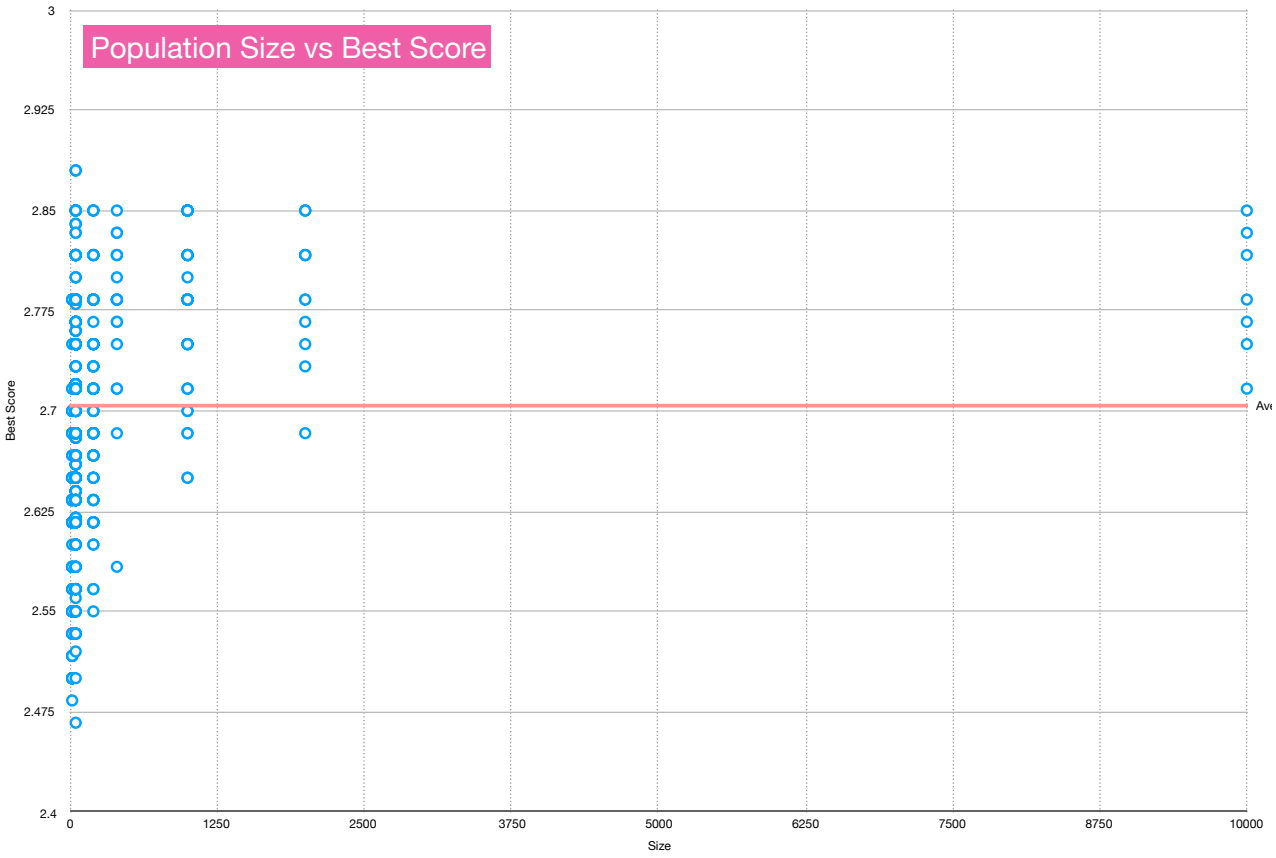
In this example we use a small number of states (2). This allows the output to fit nicely onscreen. The output to `training_output.csv` is unaffected.

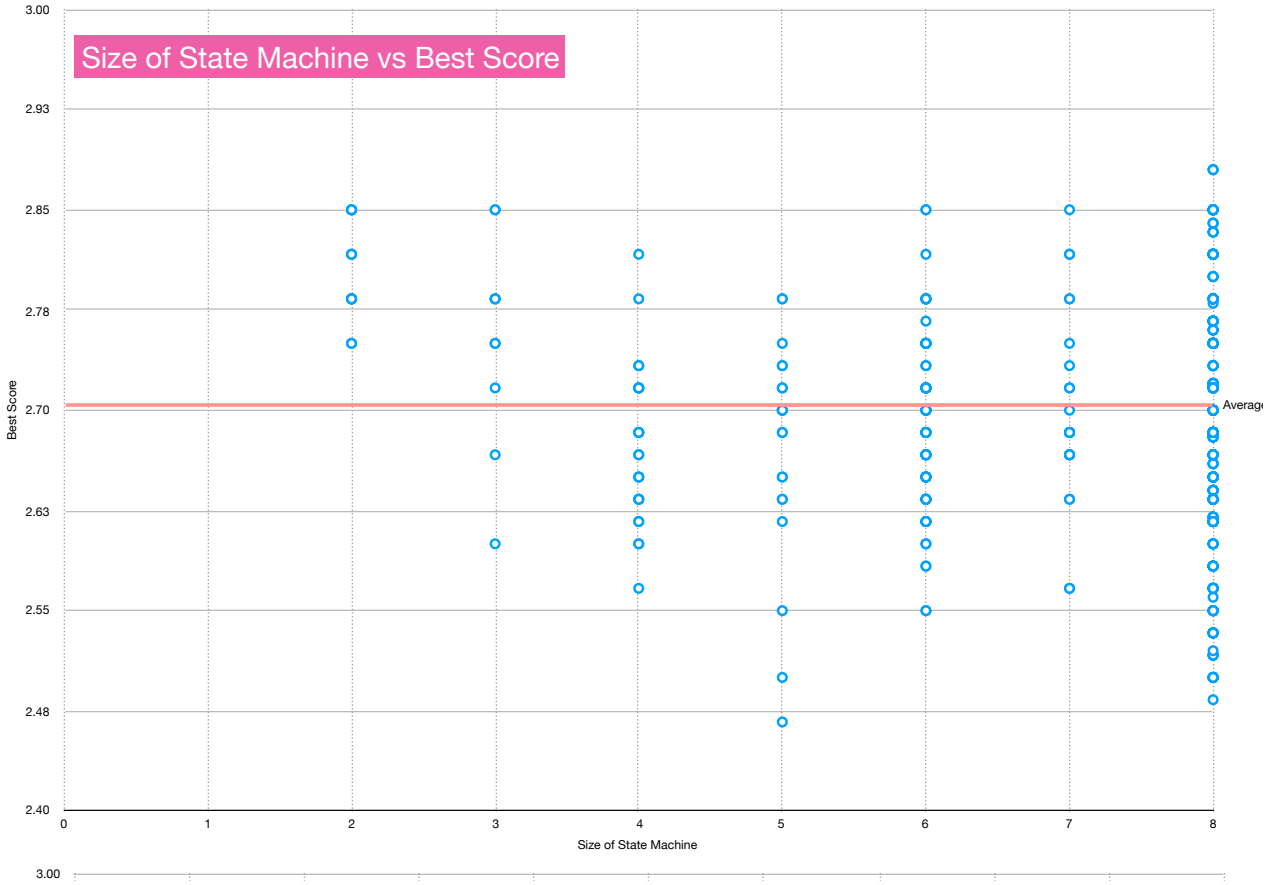
Prepare the tournament

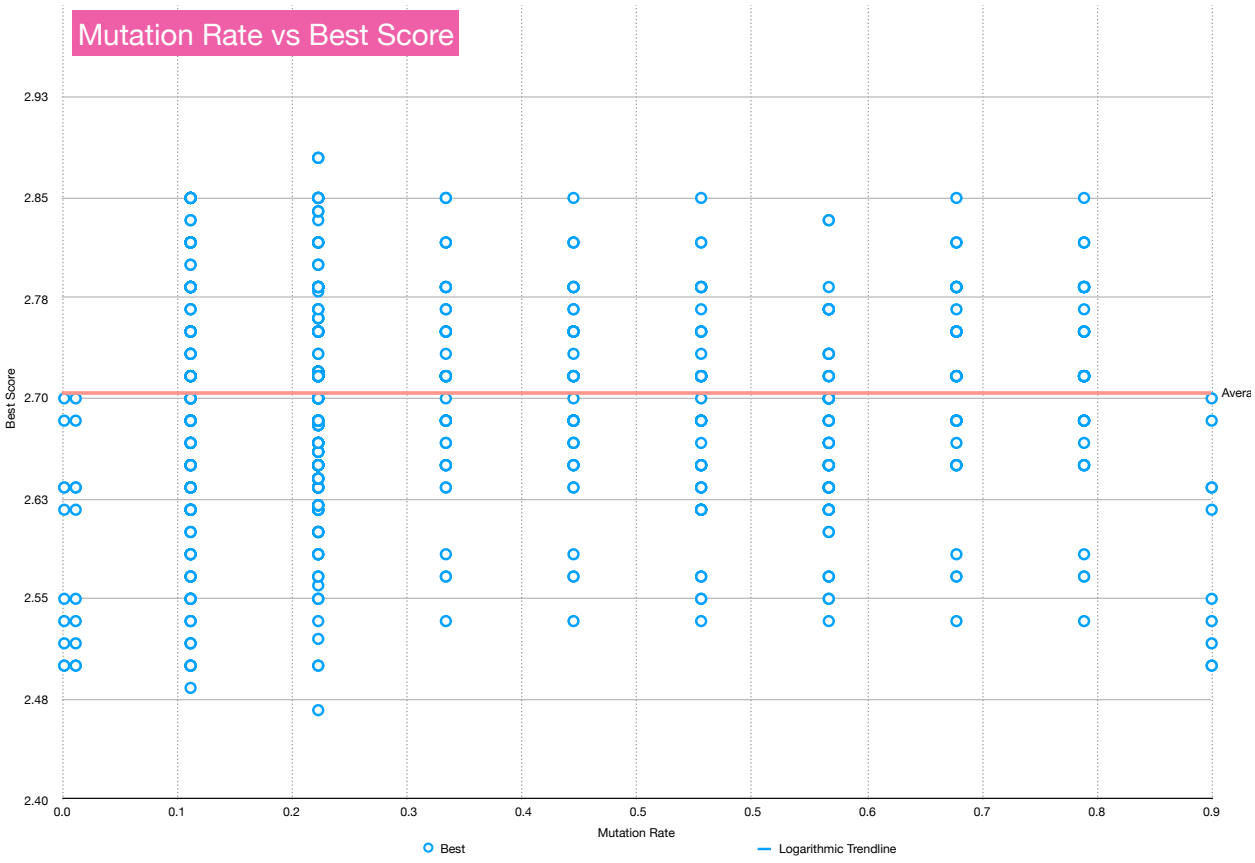
```
axl.seed(1)

# players = [s() for s in axl.demo_strategies]
# players = [axl.Alternator(), axl.Defector(),
#            axl.TitForTat()]
players = [axl.Cooperator(), axl.Defector(),
           axl.TitForTat(), axl.Grudger(),
           axl.Random(), axl.Alternator()]
# players = [axl.TitForTat()]



population = dojo.Population (params_class=params_class,
                              params_kwargs=params_kwargs,
                              size = 100, #20
                              objective= objective,
                              output_filename= "training_output.csv",
                              opponents= players,
                              bottleneck= 5, #2
                              mutation_probability= 0.1, #0.1
                              print_output= False)
```

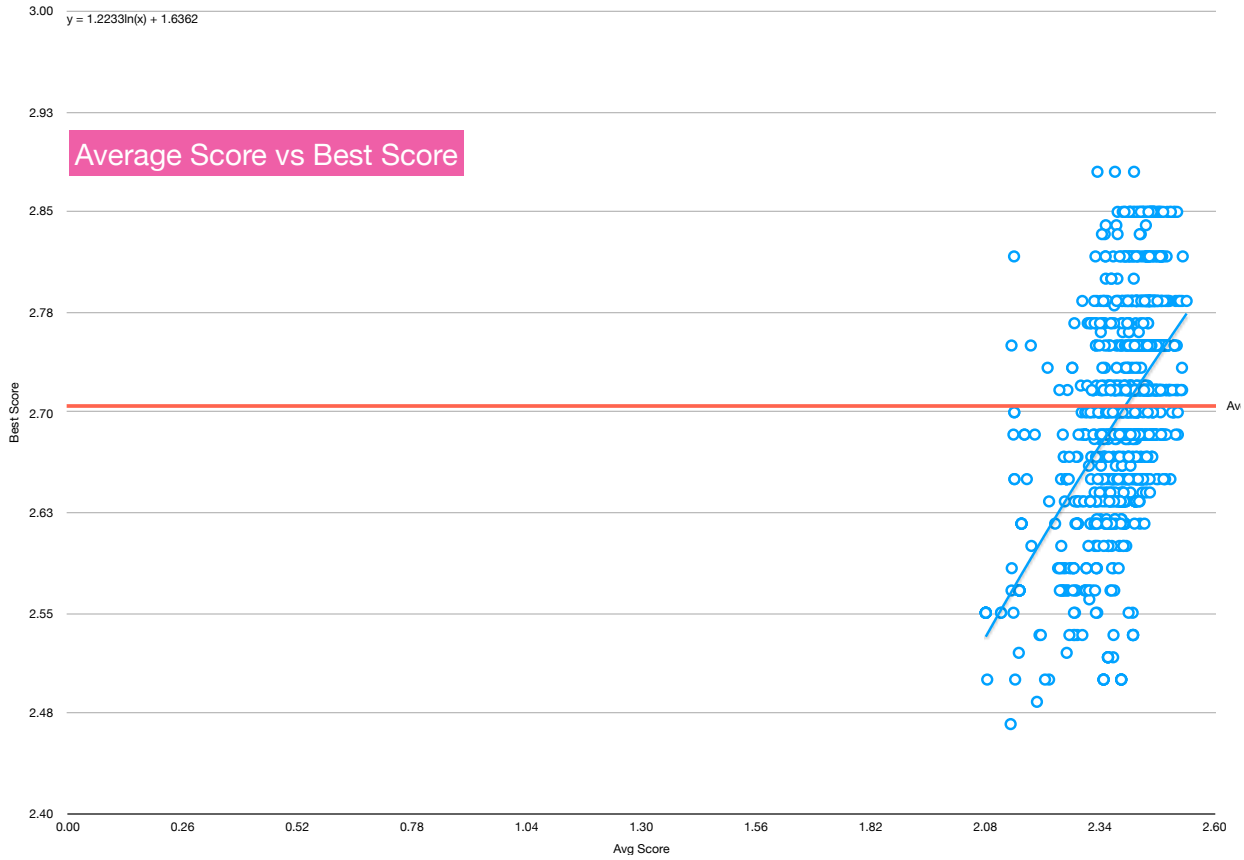
```
generations = 10 #10
results = population.run(generations)
```

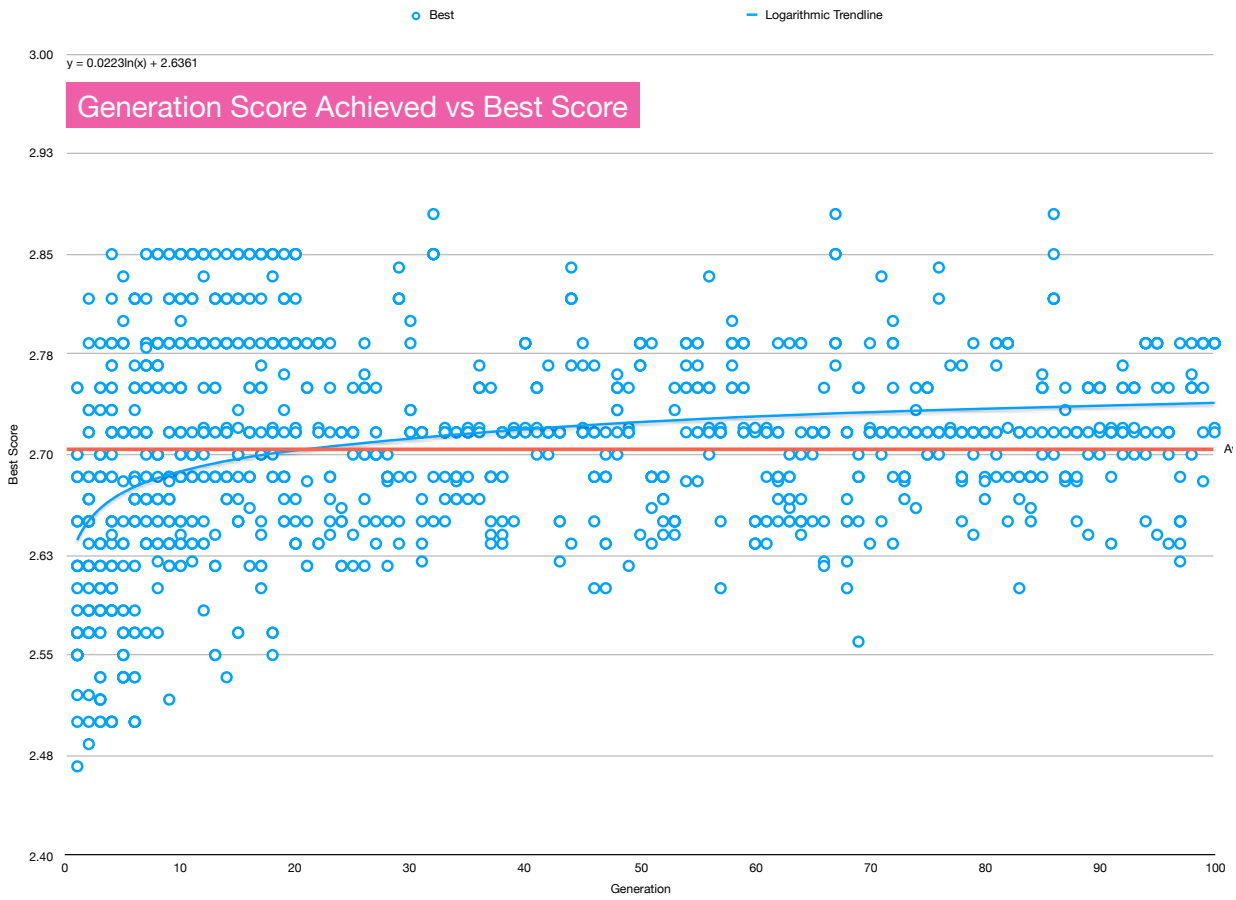
```
Scoring Generation 1
     → Mean score: 2.21, Root variance: 0.19
     Generation  1 |  Best Score:  2.600000 State: 0:C:0_C_0_D:0_D_1_D:1_C_0_D:1_D_0_C
     Generation  1 | Worst Score:  1.650000 State: 0:C:0_C_1_C:0_D_0_C:1_C_0_D:1_D_0_C
Scoring Generation 2
     → Mean score: 2.36, Root variance: 0.161
     Generation  2 |  Best Score:  2.633333 State: 0:C:0_C_0_C:0_D_1_D:1_C_0_D:1_D_1_D
     Generation  2 | Worst Score:  1.916667 State: 0:C:0_C_1_C:0_D_1_D:1_C_0_D:1_D_0_C
Scoring Generation 3
     → Mean score: 2.39, Root variance: 0.148
     Generation  3 |  Best Score:  2.683333 State: 0:C:0_C_0_D:0_D_0_D:1_C_0_D:1_D_0_C
     Generation  3 | Worst Score:  1.850000 State: 0:C:0_C_0_D:0_D_0_C:1_C_1_C:1_D_1_D
Scoring Generation 4
     → Mean score: 2.4, Root variance: 0.133
     Generation  4 |  Best Score:  2.716667 State: 0:C:0_C_0_C:0_D_1_D:1_C_1_D:1_D_1_D
     Generation  4 | Worst Score:  1.983333 State: 0:C:0_C_0_D:0_D_0_C:1_C_0_D:1_D_1_D
Scoring Generation 5
     → Mean score: 2.46, Root variance: 0.192
     Generation  5 |  Best Score:  2.850000 State: 0:C:0_C_0_C:0_D_1_D:1_C_1_D:1_D_1_D
     Generation  5 | Worst Score:  1.900000 State: 0:C:0_C_0_C:0_D_0_C:1_C_1_D:1_D_0_D
Scoring Generation 6
     → Mean score: 2.44, Root variance: 0.153
     Generation  6 |  Best Score:  2.716667 State: 0:C:0_C_0_C:0_D_1_D:1_C_1_D:1_D_1_D
     Generation  6 | Worst Score:  2.033333 State: 0:C:0_C_0_C:0_D_1_D:1_C_0_C:1_D_1_C
Scoring Generation 7
     → Mean score: 2.43, Root variance: 0.12
     Generation  7 |  Best Score:  2.716667 State: 0:C:0_C_0_C:0_D_1_D:1_C_1_D:1_D_1_D
     Generation  7 | Worst Score:  2.133333 State: 0:C:0_C_0_D:0_D_1_C:1_C_0_D:1_D_1_D
Scoring Generation 8
     → Mean score: 2.43, Root variance: 0.158
     Generation  8 |  Best Score:  2.783333 State: 0:C:0_C_0_C:0_D_1_D:1_C_1_D:1_D_1_D
     Generation  8 | Worst Score:  2.000000 State: 0:C:0_C_1_D:0_D_0_C:1_C_0_D:1_D_0_D
Scoring Generation 9
     → Mean score: 2.45, Root variance: 0.15
     Generation  9 |  Best Score:  2.783333 State: 0:C:0_C_0_C:0_D_1_D:1_C_1_D:1_D_1_D
     Generation  9 | Worst Score:  1.983333 State: 0:C:0_C_1_D:0_D_1_D:1_C_1_C:1_D_0_C
Scoring Generation 10
     → Mean score: 2.44, Root variance: 0.155
     Generation  10 |  Best Score:  2.783333 State: 0:C:0_C_0_C:0_D_1_D:1_C_1_D:1_D_1_D
     Generation  10 | Worst Score:  1.800000 State: 0:C:0_C_1_D:0_D_1_C:1_C_0_C:1_D_1_C
```


Population Size vs Best Score
Best Score
Size
3
2.925
2.85
2.775
2.7
2.625
2.55
2.475
2.4
0
1250
2500
3750
5000
6250
7500
8750
10000


Size of State Machine vs Best Score
Best Score
Size of State Machine
3.00
2.93
2.85
2.78
2.70
2.63
2.55
2.48
2.40
0
1
2
3
4
5
6
7
8


Mutation Rate vs Best Score
Best Score
2.93
2.85
2.78
2.70
2.63
2.55
2.48
2.40
0.0
0.1
0.2
0.3
0.4
0.5
0.5
0.6
0.7
0.8
0.9
Mutation Rate
Best
Logarithmic Trendline
Avera


3.00
y = 1.2233ln(x) + 1.6362
Average Score vs Best Score
Best Score
2.93
2.85
2.78
2.70
2.63
2.55
2.48
2.40
0.00
0.26
0.52
0.78
1.04
1.30
1.56
1.82
2.08
2.34
2.60
Avg Score
Av


Best
Logarithmic Trendline
y = 0.0223ln(x) + 2.6361
Generation Score Achieved vs Best Score
3.00
2.93
2.85
2.78
2.70
2.63
2.55
2.48
2.40
Best Score
0
10
20
30
40
50
60
70
80
90
100
Generation

Iterated Prisioners Dilemma Tournament

Group Project for COMP 3710

By: Mio Tanaka, Suraiya Khanda, Samar Houssami, Ben Davidson

This project uses Jupyter Notebooks, Python 3.7, the Alexrod-Python and the Axelrod-Dojo library to run, analyize and visualise an Iterated Prisioners Dilemma Tournament and introduce machine learning strategies with finite state machines.

We made some minor modifications to the dojo library to improve the reporting and output. This is most reflected in the `training_output.csv` which now records detailed information about the players used in the simulation, mutation rate, bottleneck, size of state machine and the date/time to aid in reproducing the results.

We also made minor modifications to the main Axelrod library (`player.py`) to keep strategy name short so they would display correctly in charts and graphs

Import the axelrod library

```
import axelrod as axl
%matplotlib inline

from datetime import datetime
datetime.now().strftime('%Y-%m-%d %H:%M:%S')
```

```
'2019-04-10 21:00:07'
```

Prepare our Finite State Machine created with GA

```
0:C:0_C_0_C:0_D_2_D:1_C_2_C:1_D_2_C:2_C_2_D:2_D_0_D:3_C_5_D:3_D_6_C:
4_C_0_C:4_D_2_D:5_C_6_D:5_D_1_C:6_C_6_C:6_D_5_C:7_C_2_D:7_D_3_C
```

Was the best FSM that was could create with IPD Evolutionary Training. Using an Axelrod Finate State Machine Meta-Strategy we are able to import this into our tournament.

Appears in charts and graphs as **FSM Player**

```
from axelrod import Action
C, D = Action.C, Action.D
MMSB_GA_Strat_transitions = (
    (0, C, 0, C),
    (0, D, 2, D),
    (1, C, 2, C),
    (1, D, 2, C),
    (2, C, 2, D),
    (2, D, 0, D),
    (3, C, 5, D),
    (3, D, 6, C),
    (4, C, 0, C),
    (4, D, 2, D),
    (5, C, 6, D),
    (5, D, 1, C),
    (6, C, 6, C),
    (6, D, 5, C),
    (7, C, 2, D),
    (7, D, 3, C)
)

from axelrod.strategies.finite_state_machines import FSMPlayer

MMSB_GA_Strat = FSMPlayer(transitions=MMSB_GA_Strat_transitions,
    initial_state=0, initial_action=C)
```

Prepare the tournament

```
axl.seed(0)

# players = [s() for s in axl.demo_strategies]
players = [axl.Cooperator(), axl.Defector(),
           axl.TitForTat(), axl.Grudger(),
           axl.Random(), MMSB_GA_Strat]
tournament = axl.Tournament(players)
tournament.name = 'IPDP for COMP 3710'      # set the experiment name
tournament.turns = 200                      # set the number of turns
tournament.repetitions = 10                 # number of times to repeat
results = tournament.play()
```

```
Playing matches: 100%|██████████| 21/21 [00:00<00:00, 21.93it/s]
Analysing: 100%|██████████| 25/25 [00:00<00:00, 142.96it/s]
```

Tournament

{{tournament.name}}
Ran for {{tournament.turns}} turns and repeated {{tournament.repetitions}} times

```
print("The players were: ")
print(results.players)


print("\nTheir final ranking was: ")
print(results.ranked_names)
```

```
The players were:
['Cooperator', 'Defector', 'Tit For Tat', 'Grudger', 'Random', 'FSM Player']

Their final ranking was:
['Grudger', 'FSM Player', 'Tit For Tat', 'Defector', 'Cooperator', 'Random']
```

Detailed Results

For each round, how many times did each strategy win?

```
i = 0
for e in results.wins:
    print( "{0:>15}".format(results.players[i]) + " " + str(e) )
    i += 1
```

```
      Cooperator [0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
        Defector [5, 5, 5, 5, 5, 5, 5, 5, 5, 5]
     Tit For Tat [0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
         Grudger [1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
          Random [1, 1, 2, 2, 2, 1, 1, 1, 1, 1]
      FSM Player [1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
```

For each round, what were the scores?

```
i = 0
for s in results.scores:
    print("{0:>15} ".format(results.players[i]) + " " + str(s) )
    i += 1
```

```
      Cooperator  [2079, 2100, 2121, 2133, 2091, 2082, 2082, 2118, 2082, 2109]
        Defector  [2196, 2188, 2204, 2220, 2188, 2200, 2200, 2184, 2164, 2244]
     Tit For Tat  [2461, 2442, 2448, 2447, 2431, 2441, 2446, 2449, 2424, 2474]
         Grudger  [2590, 2622, 2558, 2556, 2584, 2630, 2594, 2552, 2642, 2606]
          Random  [1835, 1751, 1746, 1746, 1712, 1772, 1724, 1772, 1752, 1729]
      FSM Player  [2478, 2508, 2499, 2514, 2562, 2483, 2516, 2528, 2534, 2495]
```

Okay, pretty good, now let's normalize those scores...

```
i = 0

for r in results.normalised_scores:
    e = []
    for c in r:
        e += [ float('{0:.2f}'.format(c ))]
    print("{0:>15} {1:} ".format(results.players[i], e ) )
    i += 1

# results.normalised_scores
```

```
     Cooperator [2.08, 2.1, 2.12, 2.13, 2.09, 2.08, 2.08, 2.12, 2.08, 2.11]
       Defector [2.2, 2.19, 2.2, 2.22, 2.19, 2.2, 2.2, 2.18, 2.16, 2.24]
    Tit For Tat [2.46, 2.44, 2.45, 2.45, 2.43, 2.44, 2.45, 2.45, 2.42, 2.47]
        Grudger [2.59, 2.62, 2.56, 2.56, 2.58, 2.63, 2.59, 2.55, 2.64, 2.61]
         Random [1.84, 1.75, 1.75, 1.75, 1.71, 1.77, 1.72, 1.77, 1.75, 1.73]
     FSM Player [2.48, 2.51, 2.5, 2.51, 2.56, 2.48, 2.52, 2.53, 2.53, 2.5]
```

This means that **{{results.players[0]}}** got, on average, a score of **{{results.normalised_scores[0][0]}}** per turn, per opponent

So who did well against who?

```
player = 0
opponent = 0

for r in results.score_diffs:
    print("{:_<88}".format(results.players[player] + " vs "))
    opponent = 0
    for opp in r:
        e = []
        for c in opp:
            e += [ float('{0:2.2f}'.format(c ))]
        print("{0:>15} {1:} ".format(results.players[opponent], e ) )
        opponent += 1
    player += 1
    print()

# results.score_diffs
```

```
Cooperator vs ____________________________________________________________________________
     Cooperator [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
       Defector [-5.0, -5.0, -5.0, -5.0, -5.0, -5.0, -5.0, -5.0, -5.0, -5.0]
    Tit For Tat [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
        Grudger [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
         Random [-2.67, -2.5, -2.33, -2.23, -2.58, -2.65, -2.65, -2.35, -2.65, -2.42]
     FSM Player [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]

Defector vs ______________________________________________________________________________
     Cooperator [5.0, 5.0, 5.0, 5.0, 5.0, 5.0, 5.0, 5.0, 5.0, 5.0]
       Defector [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
    Tit For Tat [0.03, 0.03, 0.03, 0.03, 0.03, 0.03, 0.03, 0.03, 0.03, 0.03]
        Grudger [0.03, 0.03, 0.03, 0.03, 0.03, 0.03, 0.03, 0.03, 0.03, 0.03]
         Random [2.4, 2.35, 2.45, 2.55, 2.35, 2.42, 2.42, 2.33, 2.2, 2.7]
     FSM Player [0.03, 0.03, 0.03, 0.03, 0.03, 0.03, 0.03, 0.03, 0.03, 0.03]

Tit For Tat vs ___________________________________________________________________________
     Cooperator [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
       Defector [-0.03, -0.03, -0.03, -0.03, -0.03, -0.03, -0.03, -0.03, -0.03, -0.03]
    Tit For Tat [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
        Grudger [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
         Random [0.0, 0.0, -0.03, -0.03, -0.03, 0.0, 0.0, 0.0, 0.0, 0.0]
     FSM Player [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]

Grudger vs _______________________________________________________________________________
     Cooperator [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
       Defector [-0.03, -0.03, -0.03, -0.03, -0.03, -0.03, -0.03, -0.03, -0.03, -0.03]
    Tit For Tat [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
        Grudger [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
         Random [2.4, 2.6, 2.23, 2.2, 2.38, 2.67, 2.45, 2.17, 2.7, 2.52]
     FSM Player [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]

Random vs ________________________________________________________________________________
     Cooperator [2.67, 2.5, 2.33, 2.23, 2.58, 2.65, 2.65, 2.35, 2.65, 2.42]
       Defector [-2.4, -2.35, -2.45, -2.55, -2.35, -2.42, -2.42, -2.33, -2.2, -2.7]
    Tit For Tat [0.0, 0.0, 0.03, 0.03, 0.03, 0.0, 0.0, 0.0, 0.0, 0.0]
        Grudger [-2.4, -2.6, -2.23, -2.2, -2.38, -2.67, -2.45, -2.17, -2.7, -2.52]
         Random [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
     FSM Player [-0.68, -1.05, -1.05, -1.07, -1.55, -0.82, -1.3, -1.1, -1.18, -1.15]

FSM Player vs ____________________________________________________________________________
     Cooperator [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
       Defector [-0.03, -0.03, -0.03, -0.03, -0.03, -0.03, -0.03, -0.03, -0.03, -0.03]
    Tit For Tat [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
        Grudger [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
         Random [0.68, 1.05, 1.05, 1.07, 1.55, 0.82, 1.3, 1.1, 1.18, 1.15]
     FSM Player [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
```

Take aways

- We can see that **Cooperator** is at a big disadvantage to **Defector**.

- **Defector** does very well against **Cooperator** and **Random** and is the only strategy will all positive scores.
- **FSM Player** (Our FSM) is very similar to **Tit-for-tat**. It performs better again random though.

When do different strategies cooperate?

```
i = 0

# Headers for the columns
print("{:17}".format(''), end ='')
for player in results.players:
    print("{:.3s} {:1}".format(player, ''), end='')
print()

# Rows
for r in results.normalised_cooperation:
    e = []
    for c in r:
        e += [ float('{0:.2f}'.format(c ))]
    print("{0:>15} {1:} ".format(results.players[i], e ) )
    i += 1

# results.normalised_cooperation
```

```
                Coo  Def  Tit  Gru  Ran  FSM
     Cooperator [1.0, 1.0, 1.0, 1.0, 1.0, 1.0]
       Defector [0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
    Tit For Tat [1.0, 0.01, 1.0, 1.0, 0.5, 1.0]
        Grudger [1.0, 0.01, 1.0, 1.0, 0.01, 1.0]
         Random [0.5, 0.48, 0.5, 0.5, 0.5, 0.5]
     FSM Player [1.0, 0.01, 1.0, 1.0, 0.28, 1.0]
```

- As expected, **Cooperator** *always* cooperates.
- **Random: 0.5** cooperates… **50%** of the time
- **Tit for Tat** cooperates when the other player does.
- **FSM Player** (our FSM) cooperates like Tit-for-Tat except in learns it's lesson against Random and doesn't cooperate as oftern.

Ok, taken care of, let's send the main data to a .csv file

```
results.write_summary('results/summary.csv')
# import csv
# with open('summary.csv', 'r') as outfile:
#     csvreader = csv.reader(outfile)
#     for row in csvreader:
#         print(row)
```

Now let’s visualize the results

```
# some extra graph options
# import seaborn as sns
import matplotlib.pyplot as plt
```

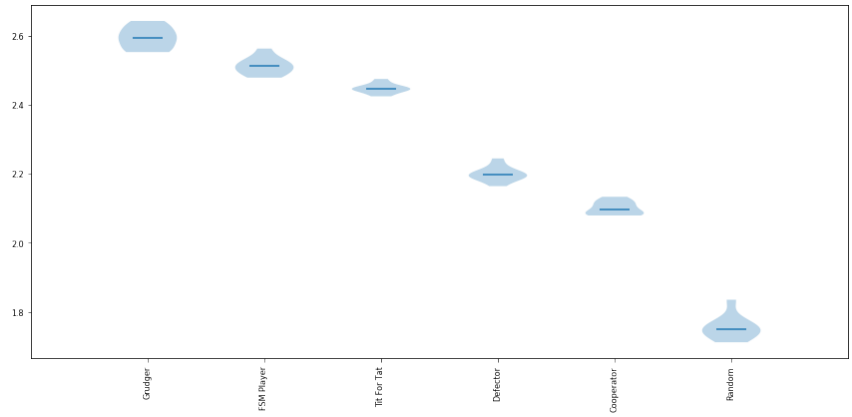
Violin plot Graph

The violin plot graph visualizes average scores in a way that helps us see stochastic effects. If all plyaers behave predictabilly horizontal lines will be shown. If a player behave stochastically, the random effects will be able to be visualized. The addition of a single stochastic strategy introduces stochastic effects into all results.

Violin plots are similar to boxplots but they are able to show the depth of data. The curve of the shape is a kernel density estimation, that shows the distribution of of the results.

Source: Violin Plots in Seaborn, Violin Plots 101: Visualizing Distribution and Probability Density

```
plot = axl.Plot(results)
b = plot.boxplot()
```

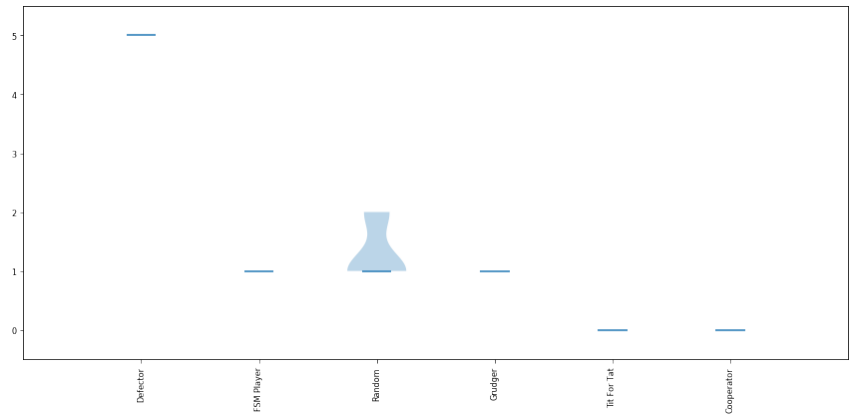


png

Win plot Graph

The win plot graph is…

```
p = plot.winplot()
```

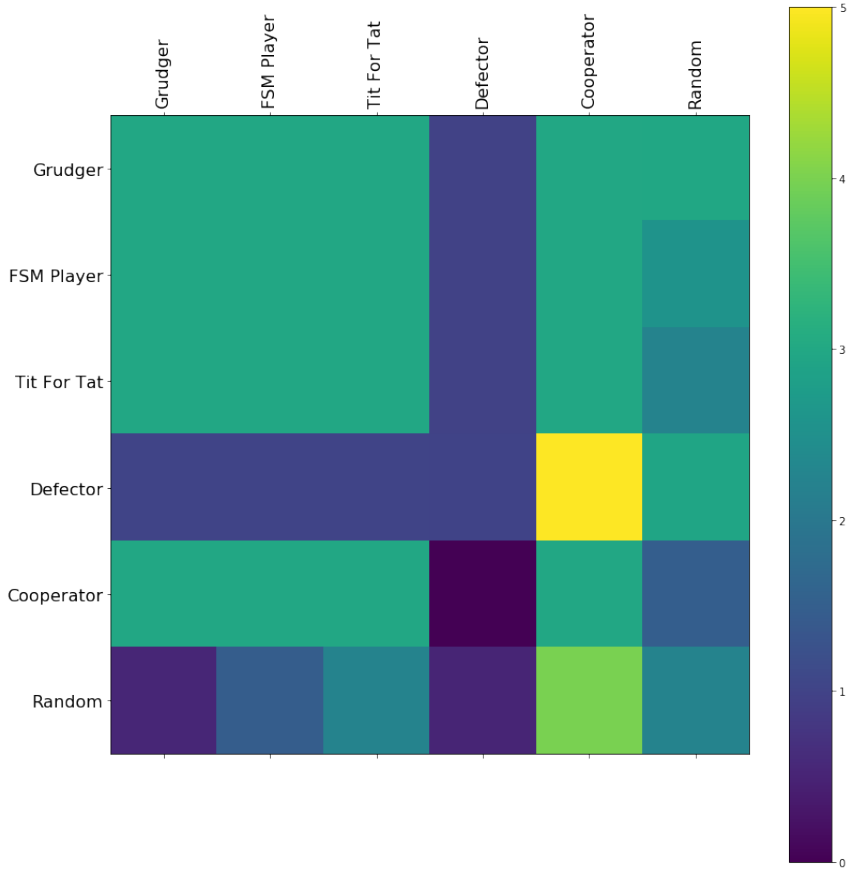


png

Payoff Graph

The payoff graph visualizes performance against other competitors. Brighter colours represent more success. For example, bright yellow indicates that **Defector** does very well against **Cooperator**

```
pay = plot.payoff()
```



png

Finally, let's save a copy of our graphs

```
axl.Plot.save_all_plots(plot)
```

```
Obtaining plots: 100%|██████████| 6/6 [00:00<00:00,  6.87it/s]
```

output_32_0.png (png) [798x821]