Thesis Title

Thesis Subtitle

James Campbell

B.Sc. Final Year Dissertation

Cardiff School of Mathematics


CARDIFF
UNIVERSITY
PRIFYSGOL
CAERDYDD

Acknowledgments

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

Nam dui ligula, fringilla a, euismod sodales, sollicitudin vel, wisi. Morbi auctor lorem non justo. Nam lacus libero, pretium at, lobortis vitae, ultricies et, tellus. Donec aliquet, tortor sed accumsan bibendum, erat ligula aliquet magna, vitae ornare odio metus a mi. Morbi ac orci et nisl hendrerit mollis. Suspendisse ut massa. Cras nec ante. Pellentesque a nulla. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Aliquam tincidunt urna. Nulla ullamcorper vestibulum turpis. Pellentesque cursus luctus mauris.

Nulla malesuada porttitor diam. Donec felis erat, congue non, volutpat at, tincidunt tristique, libero. Vivamus viverra fermentum felis. Donec nonummy pellentesque ante. Phasellus adipiscing semper elit. Proin fermentum massa ac quam. Sed diam turpis, molestie vitae, placerat a, molestie nec, leo. Maecenas lacinia. Nam ipsum ligula, eleifend at, accumsan nec, suscipit a, ipsum. Morbi blandit ligula feugiat magna. Nunc eleifend consequat lorem. Sed lacinia nulla vitae enim. Pellentesque tincidunt purus vel magna. Integer non enim. Praesent euismod nunc eu purus. Donec bibendum quam in tellus. Nullam cursus pulvinar lectus. Donec et mi. Nam vulputate metus eu enim. Vestibulum pellentesque felis eu massa.

Quisque ullamcorper placerat ipsum. Cras nibh. Morbi vel justo vitae lacus tincidunt ultrices. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. In hac habitasse platea dictumst. Integer tempus convallis augue. Etiam facilisis. Nunc elementum fermentum wisi. Aenean placerat. Ut imperdiet, enim sed gravida sollicitudin, felis odio placerat quam, ac pulvinar elit purus eget enim. Nunc vitae tortor. Proin tempus nibh sit amet nisl. Vivamus quis tortor vitae risus porta vehicula.

Contents

List of Figures

Chapter 1

Introduction

The Prisoner's Dilemma (PD) is a classic model in Game Theory. It is a simple two player game where the agents must make a choice of two options without communicating. This is often presented as two suspects who have been arrested and are being interrogated separately. They have the option of whether to Cooperate with each other or Defect and each player receives an individual payoff that depends on the actions that have been taken. This may appear quite abstract, so a more formal definition is given in Section 1.1.

Consider the situation that both prisoners return to their cells each night, to be presented with the same choice the next day. Furthermore, allow this process to continue repeatedly. This is called the Iterated Prisoner's Dilemma (IPD) and has been an object of interest ever since the 1950's but became more popular after Robert Axelrod's work in the 1980's [8, 9]. Every player in IPD represents a strategy. A strategy is a predetermined program that tells the player what actions to take in response to the history of plays so far.

Several computer based IPD tournaments have been held over the years, but the first were held by Axelrod in 1980 [8, 9]. Some much smaller tournaments are the two anniversary tournaments held in 2004 and the Stewart and Plotkin 2012 tournament. The original source code of these is only available for Axelrod's second tournament (in FORTRAN) and it is not well documented, tested or easily re-usable. The general state of reproducible research is discussed further in Section 2.4.

Recently, a group of researchers have been working to improve this situation by producing an open source version of Axelrod's original tournament [33]. Referred to as the Axelrod-Python library, it aims to provide a resource for the design of new strategies and interactions between them, as well as conducting tournaments and ecological simulations for populations of strategies. The version of the library used in this work will be [18]. This version of the library includes 181 strategies and the capability to run evolutionary tournaments and tournaments on different topologies.

1.1 Prisoner's Dilemma

The first formal definition of the PD was presented by Albert W. Tucker during a seminar at Stanford University [21]. However, the idea was first formulated in 1950 [19]. A more detailed explanation of the Prisoner's Dilemma is given in [22].

The PD can be described as follows. Two players simultaneously decide whether to Cooperate (C) or Defect (D), without exchanging information. They receive payoffs as follows:

- They both choose C (mutual cooperation) and receive a payoff R (Reward)
- They both choose D (mutual defection) and receive a payoff P (Punish)
- One player chooses C and the other chooses D. The cooperator receives a payoff S (Sucker) and the defector receives a payoff T (Temptation).

Figure 1.1 shows the payoff matrix.

$$P = \begin{array}{c} \\ C \\ D \end{array} \begin{array}{c} \begin{array}{cc} C & D \end{array} \\ \begin{pmatrix} (R,R) & (T,S) \\ (S,T) & (P,P) \end{pmatrix} \end{array} \tag{1.1}$$

There are also three assumptions that need to be stated. Firstly, both players are rational. Secondly, there is no communication between them. Finally, that the payoffs satisfy the following inequalities:

$$S < D < C < T \tag{1.2}$$

and

$$(S + T) < 2C \tag{1.3}$$

It is then easy to see that regardless of the choice of one player, the other will always obtain a higher payoff by defecting instead of cooperating. Therefore we have a pure Nash Equilibrium where both players defect, despite the fact that both players would do better if they were to cooperate with each other.

Equation 1.2 merely fixes the payoffs in their intuitive order. Equation 1.3 ensures that alternating between cooperating and defecting (players take it in turns to stab each other in the back) performs no better than mutual cooperation. These inequalities allow for many different payoff matrices to be formulated, but values of $(R, S, T, P) = (3, 0, 5, 1)$ are commonly used in literature [10, 8, 8]. We can now explicitly state the payoff matrix, as shown in Figure 1.4

$$P = \begin{array}{c} \\ C \\ D \end{array} \begin{array}{c} \begin{array}{cc} C & D \end{array} \\ \begin{pmatrix} (3,3) & (5,0) \\ (0,5) & (1,1) \end{pmatrix} \end{array} \tag{1.4}$$

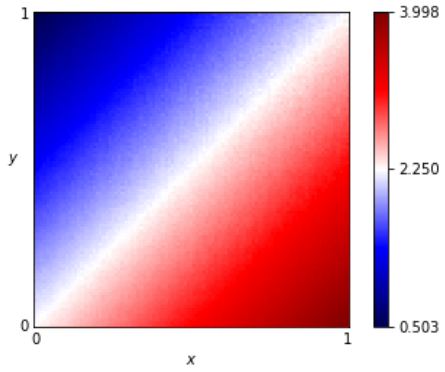


Figure 1.1: A plot of the fingerprint function for Random(0.5) when probed by TitForTat

1.2 Problem Description

As previously mentioned, the version of the Axelrod-Python library used in this report has 181 strategies, significantly more than the 13 and 64 strategies submitted to Axelrod's first and second tournaments [8, 9]. This now raises the issue of duplication, and subsequently, how to differentiate between strategies. Currently the only known approach to this problem in the literature is Fingerprinting which produces a visual representation of a strategy, an example is shown in Figure 1.1.

The issue with this method is that it relies on knowledge of the underlying Markov chain of the strategy which cannot be easily constructed from the source code. Secondly, although the fingerprinting method is described and used at length in [2, 3, 4, 5, 6, 7] there is no source code readily available, it must be requested from the original author. This project will document an implementation of fingerprinting that is at present available in the Axelrod-Python library (Chapter 4) and also consider a statistical approach to identifying similar strategies (Chapter 6).

1.3 Structure

Throughout this report examples and demonstrations of code will be given. Some of this code will be directly copied from the Axelrod-Python source code, at other times it will have been written specifically for this report. In order to help the reader differentiate between each scenario, two different colour schemes have been used. The first, for Axelrod-Python source code is shown in Listing 1 and the second, for any bespoke code, is shown in Listing 2.

This report is organized into several chapters. Continuing from this introduction:

- In Chapter 2 an overview of previous literature regarding the Prisoner's Dilemma is given. Also introduces the Axelrod-Python library and recent work related to Fingerprinting.
- In Chapter 3 several definitions and theorems are presented, ultimately leading to a formal definition of

```
def function_name(parameters):
    """
    multiline comment
    """
    function_code = 4
    return some_output
```

Listing 1: An example of how Axelrod-Python source code will be displayed

```
def function_name(parameters):
    """
    multiline comment
    """
    function_code = 4
    return some_output
```

Listing 2: An example of how demonstrative code will be displayed

Fingerprinting. An example Analytical Fingerprint is then constructed, followed by an original result for extending Fingerprinting.

- In Chapter 4 the development process and main software contribution to Axelrod-Python is described. Some numerical and analytical Fingerprints are compared.
- In Chapter 5 Several Fingerprints are examined in detail, with comparisons being made between different strategies and different probes.
- In Chapter 6 A novel approach to identifying identical strategies using machine learning is outlined. The process of training and assessing the model is described in detail.
- In Chapter 7 An estimate of the models sensitivity is given. The model is then applied to Axelrod-Python with an aim to identify duplicate strategies.
- In Chapter 8 the results of this project are summarised. Some ideas for future research are also presented.

Chapter 2

Literature Review

As described in Chapter 1, the Prisoner's Dilemma is a very popular model in game theory and there have been many papers written about the subject. The game has been applied to many different research areas and is often used to model systems in biology [56], sociology [20], psychology [28], and economics [14]. The start of this chapter will give a brief overview of the literature and particularly relevant work will be highlighted. This is followed by an outline of how Axelrod's work is currently being reproduced by an open-source community. Finally, an introduction to Fingerprinting (a technique used for identifying similar strategies) is given at the end of the chapter.

2.1 Background

The political scientist Robert Axelrod held the first IPD tournament in 1980 [8]. Many well-known game theorists were invited to submit strategies that would compete against each other in a round robin style format. All strategies also competed against a random strategy (that would randomly choose between C and D) and a copy of themselves. All strategies knew that the length of each game was 200 moves, and the whole tournament was repeated 5 times for reliability. Out of the 13 strategies that were entered, TitForTat was announced as the winner and was submitted by Professor Anatol Rapoport from the Department of Psychology of the University of Toronto [46].

TitForTat is a very simple strategy (see Section 2.2.1) and as explained in [9], it won because of three defining characteristics:

- 'Niceness' - A strategy is said to be nice if it is not the first to defect.
- 'Provocability' - Immediately after an opponent defects, the strategy should defect in retaliation.
- 'Forgiveness' - The strategy is willing to continue with mutual cooperation even after some defections.

Axelrod's second tournament [9] saw a dramatic increase in terms of size, with 62 strategies being entered from 6 different countries (the strategy Random was also included). The contestants ranged from a 10-year-

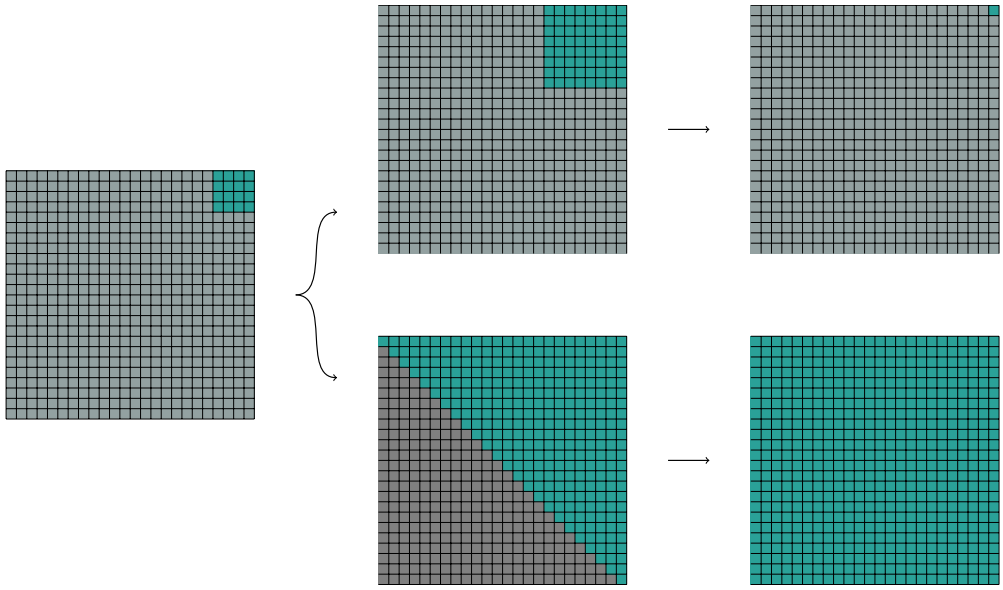


Figure 2.1: When a populations of Grey Strategies
is invaded by a small population of Cyan Strategies.

old computer hobbyist to professors of computer science, economics, psychology, mathematics, sociology, political science and evolutionary biology. The countries represented were the United States, Canada, Great Britain, Norway, Switzerland, and New Zealand. Despite the fact that all contestants had full knowledge of the previous tournament, TitForTat was the overall winner once again. One large difference in the mechanics of the first and second tournament was that the second tournament did not specify how many moves a game would last. Instead, the game ended probabilistically with a 0.00346 chance of finishing on any given move. This parameter was chosen so that the median length of a game would be 200 moves (in line with the first tournament).

Year	Reference	Number of Strategies	Type
1979	[8]	13	Standard
1979	[9]	64	Standard
1984	[11]	64	Evolutionary
1991	[13]	13	Noisy
2005	[15]	223	Varied
2012	[59]	13	Standard

Table 2.1: An overview of published tournaments

Axelrod continued to extend his work by considering an evolutionary version of the tournament [11, 10]. In this case, the proportion of the population playing a certain strategy depends on how the strategy performed on the previous round. A strategy is evolutionarily stable if a population of individuals using that strategy cannot be invaded by a rare mutant adopting a different strategy [11], see Figure 2.1.

In [40, 41, 42], Nowak extends this further by studying Spatial Games. In his variation, the game is played on a 2 dimensional square lattice where the payoff for an individual is the sum over all interactions with its

8 nearest neighbours (see Figure 2.2) and itself. In the next generation, an individual cell is occupied with the strategy that received the highest payoff among all 8 nearest neighbours and itself.

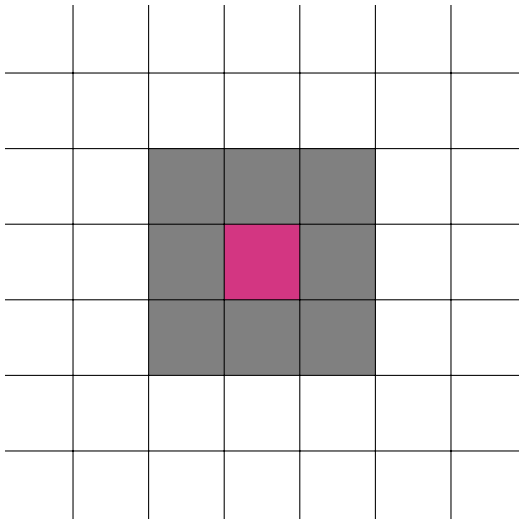


Figure 2.2: The topology of Nowak's spatial variation

To simplify things, Nowak limited each cell to be either Cooperator or Defector. Thus the game is completely deterministic and the outcome depends only on the initial configuration and the payoff matrix. One particularly interesting result is that if a single Defector invades a world of Cooperators, long (but finite) sequences of patterns emerge. Due to the symmetry of the game rules, all the patterns are highly symmetric and have the appearance of kaleidoscopes.

The perfect rules of Prisoner's Dilemma rarely apply in reality. Instead, mistakes are made or information misinterpreted, ie the environment is noisy. The primary purpose of [13] was to discover the effect of noise on TitForTat's performance in an IPD by randomly flipping the plays of players. As had been previously claimed its performance was significantly reduced. The authors of [13] suggest that this is due to the provocability mentioned earlier, which in the presence of noise, can causes it to fall into unintended vendettas with other nice, but provocable strategies. Strategies that outperformed TitForTat were far more forgiving, which enabled them to get back to mutual cooperation much faster after an accidental defection.

2.2 Strategies of Particular Interest

Throughout this report many different strategies will be discussed and used in small examples. This section will provide a description of some of these strategies in order to aid the reader, as a strategy's name is not always descriptive.

2.2.1 TitForTat

TitForTat is the most well known strategy for playing Prisoner's Dilemma due to it winning both of Axelrod's first two tournaments. It starts with a cooperative move and proceeds to play the same as the opponent did on the previous move [9, 23].

2.2.2 Pavlov

In [34], the strategy Pavlov is introduced (sometimes referred to as Win-Stay Lose-Shift) and discussed further in [41]. Pavlov plays by repeating its previous move if it was successful (received payoff T or R), and swapping the move if it was unsuccessful (received payoff P or S). The paper explains how this allows it to take advantage of strategies that cooperate unconditionally and can also correct occasional mistakes.

2.2.3 Gradual

The strategy Gradual is present in [12]. It begins the game by cooperating, then after the first defection of the other player, it defects one time and cooperates twice. After the second defection of the opponent, it defects two times and cooperates twice. After the n^{th} defection it reacts with n consecutive defections and then two cooperations. In [12] it is claimed that Gradual outperforms TitForTat and this has been observed in similar tournaments run through Axelrod-Python [18].

2.2.4 Random

Random plays by choosing randomly between whether to cooperate or defect. The probabilities of choosing cooperate or defect are not necessarily even, instead they are a distribution which can be specified. The strategy that chooses to cooperate with probability p will be denoted as Random(p). For example, the strategy that randomly chooses to cooperate 20% of the time and defect 80% of the time would be Random(0.2).

2.2.5 Cooperator/Defector

Cooperator and Defector are very simple. Cooperator will choose to cooperate at every turn, and similarly Defector will choose to defect every turn.

2.2.6 Cycler

The strategy Cycler(s) will repeat a given sequence of moves s. For example Cycler(CD) would play $CDCDCD\ldots$ and Cycler($CDDDC$) would play $CDDDCCDDDCCDDDC\ldots$

2.2.7 Evolved Looker-Up

The winner of the round robin tournament in the version of Axelrod-Python (see Section 2.4) used throughout this paper is Evolved Looker-Up.

The author, Martin Jones, has produced a thorough explanation of how the strategy was created in [29], and a brief outline will be given here. The strategy uses a lookup table to determine the action it should take. A lookup table is similar to a hash table when programming, where the keys could be the opponents previous two moves and the values the next action the strategy should take, see Table 2.2 for an example of this.

Key	Next action
CC	C
CD	D
DC	C
DD	D

Table 2.2: An example look up table that used the opponents previous 2 moves as the key

The key could then be extended to include the opponents first two moves (or more). It is clear that for a key with l parameters, there are $n = 2^l$ keys, because there are two options for each parameter, C or D. For each key the next action could be Cooperate or Defect, and so there are a total of 2^n possible lookup tables. For example, a Looker-up uses as its keys: the opponents first two moves, the opponents previous two moves, and the strategies own previous two moves. In this case there are $n = 2^6 = 64$ keys, and therefore $2^{64} \simeq 10^{18}$ possible lookup tables.

An evolutionary algorithm is then used to find the best possible table. The operates by initially creating a population of many different lookup tables. Then a new lookup table is created by combining two pre existing ones together. There is a probability (normally low) of some section of the new lookup table mutating. Next, several more random lookup tables are produced. Finally, all lookup tables are scored and poor performers are discarded. Then the whole process is repeated with the new generation of lookup tables.

Evolved Looker-Up was produced after 200 generations, and the final lookup table can be found in the Axelrod-Python source code. Alternatively, the code used to produce the lookup table is given in [37].

2.3 Finite State Machines and Automatons

An Automaton is an abstract model of a machine that can perform operations on an input. A more general (and powerful) abstraction of an automaton is the famous Turing machine, which can perform any computational process carried out by present day computers [63, 32]. In the case where an automaton has a finite set of set of states, it is referred to as a Finite State Machine. If this FSM has outputs it is called a Moore Machine [39]. A more detailed of Finite State Machines is given in Section 3.1.

Finite state machines have become important in game theory for several reasons. In the late fifties the idea

of "bounded rationality" was explored in economics by Simon [57]. This was then extended in [50] where strategies that played the IPD were implemented as Moore Machines with an extra rule that added a cost associated with the complexity of the Moore Machine. Complexity was defined to be the number of states in the machine and the author states that this was a "fairly naive" approach.

In [30, 31], proofs are given that show that every strategy can be represented by an automaton (which has infinite internal states). However, in Section 3.7 it is shown that if the length of a game in IPD is known, every strategy can be represented as a FSM.

In order to continue with IPD research, a mechanism for running tournaments is required, exactly as Axelrod and others did in Table 2.1. The next section introduces a library for implementing IPD strategies on a computer and playing them against each other in tournaments.

2.4 Axelrod-Python Library

The Axelrod-Python library [18] is an open source Python package for carrying out reproducible research into the Prisoner's Dilemma.

The original aim was to recreate Axelrod's tournaments as described in Section 2.1 and verify their results. As the library has grown, so has the overall aim. The goal now, is "to provide a resource, with facilities for the design of new strategies and interactions between them, as well as conducting tournaments and ecological simulations for populations of strategies" [33].

As mentioned in Chapter 1 for many of the tournaments that have been described the original source code is not available, and in the few cases where access was available there were no tests and minimal documentation. The library is partly motivated by a desire to improve this situation, and there is much discussion within academia currently regarding reproducible research [17, 24, 44, 51]. Some key characteristics are that the library is:

- Open: all code is released under an MIT license [49]
- Reproducible and well-tested: at the time of writing there is an excellent level of integrated tests with 99.73% coverage (including property based tests: [35])
- Well-documented: all features of the library are documented for ease of use and modification
- Extensive: 181 strategies are included, with infinitely many available in the case of parametrised strategies
- Extensible: easy to modify to include new strategies and to run new tournaments

Each of these items will be discussed in more detail in Section 4.5.

In listing 3 an example of how to produce a simple tournament is shown. Lines 7 - 10 create two plots. The first is a ranked violin plot of the mean payoff for each player and the second is a matrix plot of pair wise payoffs for each player. These plots can be seen in figures 2.3 and 2.4.

```
>>> import axelrod as axl
>>> axl.seed(0)  # Set a seed
>>> players = [axl.TitForTat, axl.Cooperator, axl.Random, axl.Gradual]  # Create players
>>> tournament = axl.Tournament(players)  # Create a tournament
>>> results = tournament.play()  # Play the tournament
>>> plot = axl.Plot(results)
>>> p = plot.boxplot()
>>> p.show()
>>> q = plot.payoff()
>>> q.show()
```

Listing 3: Example code to produce a simple tournament

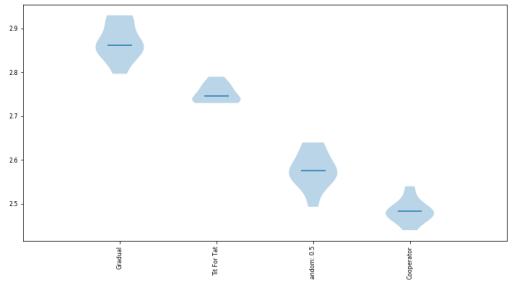


Figure 2.3: Ranked violin plot of the mean payoff for each player

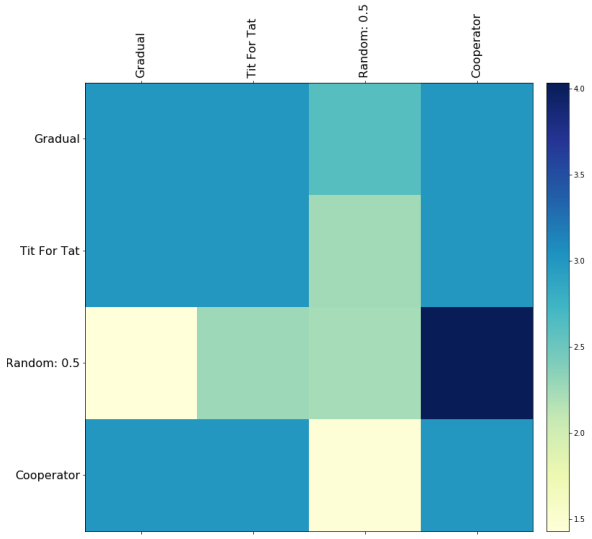


Figure 2.4: Matrix plot of pair wise payoffs for each player

The difference in scores in Figure 2.3 imply that the strategies included are all different from each other. However, as more strategies are included it becomes harder to establish when they might be equivalent.

2.5 Fingerprinting

The method of Fingerprinting is a technique for generating a functional signature for a strategy [2]. Fingerprint functions can then be compared to allow for easier identification of similar strategies. It was first given in [4] and has been extended in [2, 3, 5, 6, 7].

In [4] Ashlock outlines several definitions, theorems and proofs concerning the construction of a Fingerprint. These are then followed by some examples, however they are of low quality and the only probe (see definition 5) used is TitForTat.

A large extension to the work is made in [2]. More examples are presented using 4 different probes instead of solely TitForTat, and a catalogue of several Fingerprint functions is given. Fingerprinting is then used to to assess how three evolutionary algorithms produce different populations. The evolutionary methods are used to train finite-state machines, lookup tables and feed forward neural nets. Fingerprinting demonstrates that all three representations sample the strategy space in a radically different manner.

A recount of Ashlock's work in [2, 3, 4, 5, 6, 7] will now be given in Chapter 3. This culminates in an **original** result allowing his work to be broadened and applied to strategies without an FSM representation.

Chapter 3

Theory

Many definitions will be now be presented, with the overall aim being to have a rigorous definition for a fingerprint. A definition of Finite State Machines will be given, and an explanation of how they relate to fingerprinting. Then definitions for the Dual, Joss-Ann and Fingerprint as presented by Ashlock in [4]. It will then be shown that any strategy can be represented as a Finite State Machine, which allows the work of Ashlock to be extended to include any strategy to be used as a probe. Finally, an example of how to construct an analytical fingerprint will be shown in Section 3.6. In essence this chapter presents a detailed review of the work of [2, 3, 4, 5, 6, 7].

3.1 Finite State Machines (FSM)

A formal definition of a Finite State Machine is given by Definition 1, but first, some motivating key characteristics of a system that can be modelled with a FSM:

- The system must be describable by a finite set of states.
- The system must have a finite set of inputs that can trigger transitions between states.
- The behaviour of the system at a given point in time depends upon the current state and the input that occurs at that time.
- For each state the system may be in, behaviour is defined for each possible input.
- The system has a particular initial state.

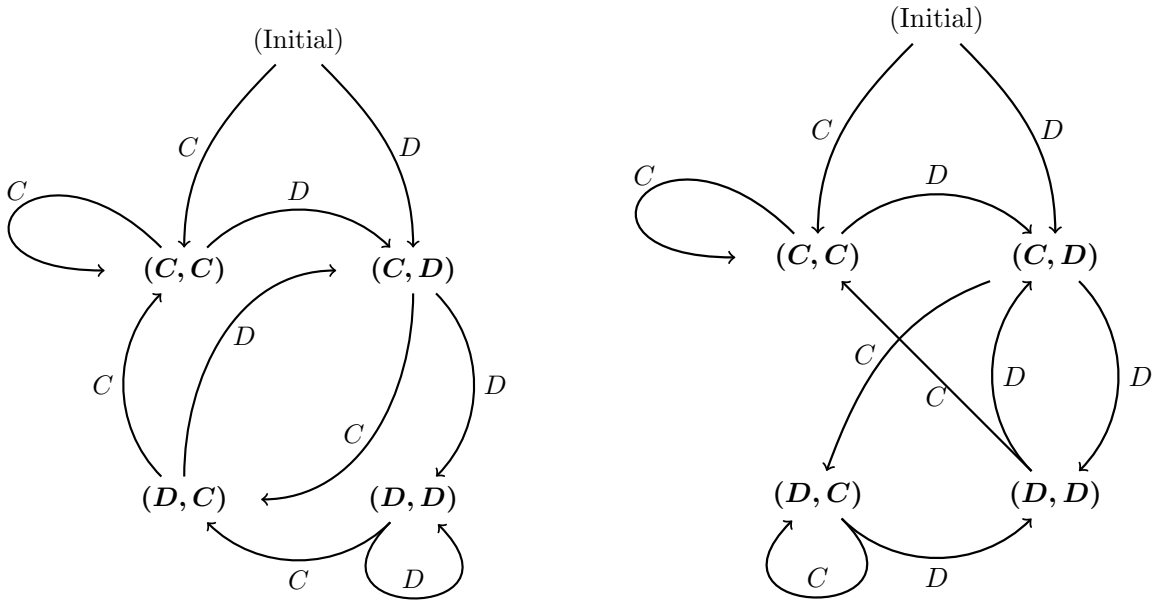
We can make the above bullet points rigorous with the following definition:

Definition 1 *A **Finite State Machine** M is a tuple $(S, \sigma, \delta, s_0, F)$ where*

- *σ is the set of symbols representing the input of M.*
- *S is the set of states of M.*

- $s_0 \in S$ *is the starting state.*
- $F \subseteq S$ *is the set of final states of* M*.*
- $\delta : S \times \sigma \to S$ *is the transition function.*

Figure 3.1a and figure 3.1b show FSM representations for TitForTat and Pavlov respectively (see Sections 2.2.1 and 2.2.2 for an explanation of how these strategies operate). Here nodes represent the previous action taken by the strategy and the opponent, ie. node (D, C) implies that on the preceding turn, the strategy chose to Defect and the opponent chose to Cooperate. Arcs represent the choice made by the opponent at the current turn, and lead us to the state for the next turn.



(a) FSM for TitForTat

(b) FSM for Pavlov

Figure 3.1: Finite State Machine representations for TitForTat and Pavlov

Figures 3.1a and 3.1b are merely pictorial representations of Definition 1. The set of input symbols is given by $\sigma = \{C, D\}$. Nodes are the possible states S, including the initial state s_0. Whilst not given explicitly, the transition function δ can be inferred from the edges of the Figures.

The next few Sections (3.2 to 3.4) outline some transformations that can be applied to strategies. In particular, Section 3.4 relies on the strategy to be represented as a FSM as has just been defined. Once the transformations had been presented, a formal definition of a Fingerprint can be given.

3.2 The Joss-Ann

The Joss-Ann is a basic transformation that can be applied to a strategy [2, 3, 4, 5, 6, 7]. It operates by making a probabilistic choice of Cooperation, Defection or the original move. More formally:

Definition 2 *If A is a strategy for the iterated prisoner's dilemma, then the* ***Joss-Anne of A*** *denoted by $JA(A, x, y)$ is a transformation of that strategy. Instead of the original behaviour, it makes move C with probability x, move D with probability y, and otherwise uses the response appropriate to strategy A (assuming $x + y < 1$).*

The notation JA comes from the initials of the names Joss and Anne. Joss was a strategy submitted to one of Axelrods original tournaments and it would occasionally Defect without provocation in the hopes of a slight improvement in score. Anne is the first name of A. Stanley who suggested the addition of random Cooperation instead of random Defection [2]. When $x + y = 1$, the original strategy is not used, and the resulting behaviour is a random strategy with probabilities (x, y). Note that this corresponds to Random(x) as described in Section 2.2.4. In more general terms, a JA strategy is an alteration of A that causes the strategy to be played with random noise inserted into the responses.

3.3 The Flip

The Flip is a simple transformation that returns the opposite of the strategy. This is subtly different from the Dual as demonstrated in Table 3.1.

Definition 3 *Strategy A' is said to be the* ***Flip*** *of strategy A if, A and A' return different actions for the same game history (all moves made by both players).*

3.4 The Dual

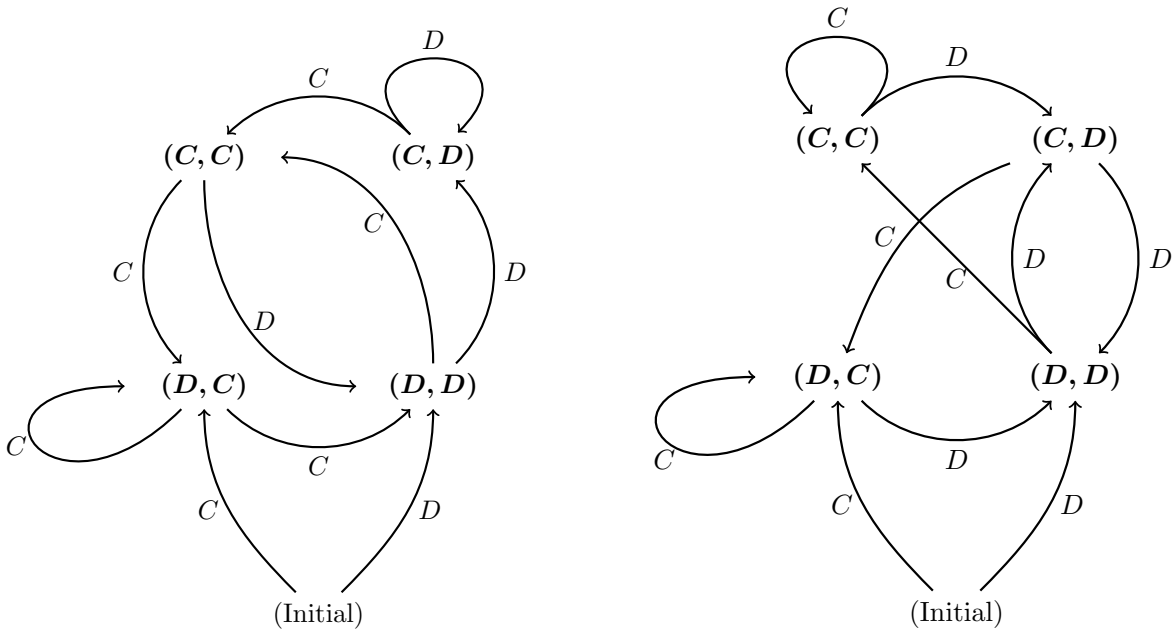
The Dual is another, more complex transformation of a strategy [2, 3, 4, 5, 6, 7] (and these transformations can be applied commutatively). Traditionally, the Dual is only defined on a strategy with an FSM representation.

Definition 4 *Strategy A' is said to be the* ***Dual*** *of strategy A if A and A' can be written as finite-state machines that are identical except that their responses are reversed.*

In Section 3.8 this definition will be extended to allow for the Dual to be calculated for any strategy, even ones without a natural FSM representation. FSM representations for the Dual of TitForTat and Pavlov are given in Figures 3.2a and 3.2b. The FSM representation of the Dual of Pavlov is exactly the same as the original except that the initial move has changed. For TitForTat, the Dual plays in entirely the opposite way and all moves have been changed.

It is important to note in general the Dual is different to taking a strategy and flipping its responses. The Dual relies on knowledge of the underlying state of the original strategy, whereas the flip does not. This is shown in Table 3.1.

The subtle difference between Dual and Flip can be highlighted further by inspecting each row individually.



(a) FSM for the Dual of TitForTat

(b) FSM for the Dual of Pavlov

Figure 3.2: Finite State Machine representations for the Duals of TitForTat and Pavlov

Opponent history	Pavlov	Dual	Flip
C	C	D	D
D	C	D	C
D	D	C	C
C	C	D	C
C	C	D	D
D	C	D	C
C	D	C	C
D	D	C	D
	C	D	D

Table 3.1: The different responses of Pavlov, Pavlov's Dual and Flipped Pavlov

Row 1 - Pavlov always plays C on the first go. Flip will change this to D. Dual knows that Pavlov always plays C regardless of the opponents history and so swaps to D.

Row 2 - In the previous round for Pavlov the strategies played (C, C), and so Pavlov plays C again. For Flip, the preceding interaction was (D, C), in this instance Pavlov would play D again, so this gets flipped to C. The previous turn for Dual was (D, C) so it infers that Pavlov had (C, C). It knows that Pavlov would play C and so plays D.

Row 3 - In the previous round for Pavlov the strategies played (C, D), and so Pavlov would change to play D. For Flip, the preceding interaction was (C, D), in this instance Pavlov would change to D, so this gets flipped to play C again. The previous turn for Dual was (D, D) so it infers that Pavlov had (C, D). It knows that Pavlov would play D in this instance and so plays C.

By following this procedure we can see that the Dual responses and Pavlov's responses are always opposite. This is achieved by the Dual inferring what the original strategy must have played and then reacting appropriately. In the above example, Pavlov's memory is only one turn deep, so the Dual only has to infer the preceding turn, however, for more complex strategies, it may need to infer the entire history of the original strategy in order to proceed correctly.

Using the definitions presented in Sections 3.2 to 3.4, the Fingerprint and Double Fingerprint functions can be defined. This provides the tools to produce a visual representation for a strategy.

3.5 Fingerprint and Double Fingerprint

The Fingerprint is a function that returns the expected score of a strategy when it plays against the Joss-Ann with varying (x, y). The double fingerprint extends this idea to $x + y \geq 1$ [2, 3, 4, 5, 6, 7].

Definition 5 *A **Fingerprint** $F_A(S, x, y)$ with $0 \leq x, y \leq 1$, $x + y \leq 1$ for strategy S and **probe** A, is the function that returns the expected score of strategy S against $JA(A, x, y)$ for each possible (x, y).*

Definition 6 *The **Double Fingerprint** $F_{AB}(S, x, y)$ with $0 \leq x, y \leq 1$ returns the expected score of strategy S against $JA(A, x, y)$ if $x + y \leq 1$, and $JA(B, 1 - y, 1 - x)$ if $x + y \geq 1$.*

A is referred to as the **lower probe** and B is referred to as the **upper probe**. This is shown in Figure 3.3. In [4] it is shown that the Double Fingerprint, with an appropriate choice of lower and upper probe, is equivalent to the Fingerprint function over the unit square. The proof in that paper will now be considered here.

Theorem 1 *If A and A' are dual strategies, then $F_{AA'}(S, x, y)$ is identical to the function $F_A(S, x, y)$ extended over the unit square.*

Proof 1 *The Markov chain for the dual strategy A' will have the same transitions as the Markov chain for the strategy A. However, each entry for x corresponds to the probability that the strategy $JA(A, x, y)$ will randomly choose C when it would not normally do so. For strategy A', this will occur whenever $JA(A', x, y)$ does not randomly respond D, which has probability $1 - y$.*

Similarly, each y corresponds to the probability that the strategy $JA(A, x, y)$ will randomly choose D when it would usually respond C. For strategy A', this will occur whenever $JA(A', x, y)$ does not randomly respond C, which has probability $1 - x$.

Thus the Markov chain for $JA(A', x, y)$ is the Markov chain for $JA(A, x, y)$ with the mapping $(x, y) \to (1 - y, 1 - x)$. Therefore $F_{AA'}(S, x, y)$ extends to the remainder of the unit square the function given by $F_A(S, x, y)$. ■

Theorem 1 allows the Fingerprint function to naturally extend over the unit square and subsequently, the phrases Fingerprint and Double Fingerprint will be used interchangeably. Figure 3.3 shows that in the bottom left region of the plot, the strategy plays against $\text{JA}(A, x, y)$ and in the top right region it plays against $\text{JA}(A', 1 - y, 1 - x)$.

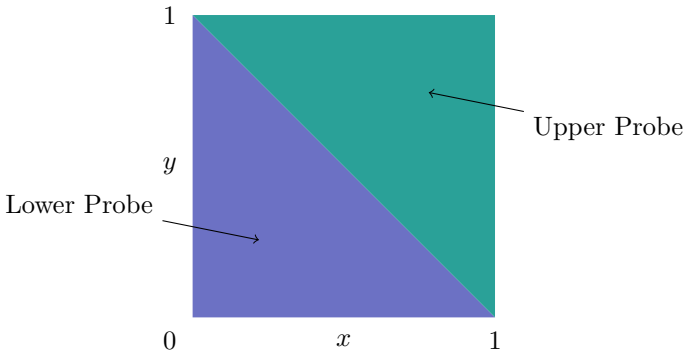


Figure 3.3: Where the Strategy plays against the Joss-Ann of the probe or the Joss-Ann of the Dual of the probe

It is important to note that a Fingerprint is an **analytical** function for the expected score of the strategy. If the probe cannot be represented as an FSM, the Dual of the probe cannot be constructed and the Fingerprint function cannot be computed.

3.6 Analytical Fingerprints

There are several steps to constructing the Fingerprint of a strategy and basic knowledge of Markov Chains is required. An outline of the steps is as follows:

1. Build a Markov chain model of an IPD between the strategy and probe strategy.
2. Construct the corresponding transition matrix.
3. Find the steady state distribution.
4. Calculate the overall expected score by taking the dot product of the steady state distribution with the payoff vector given in Section 1.1 to obtain the fingerprint function.
5. This can then be plotted as a heat map to make it easier to visualize.

As an example, this process will now be applied to obtain a fingerprint for the strategy Win-Stay-Lose-Shift (sometimes referred to as Pavlov) when probed by Tit-For-Tat.

Step 1 - Build the Markov Chain, shown in Figure 3.4.

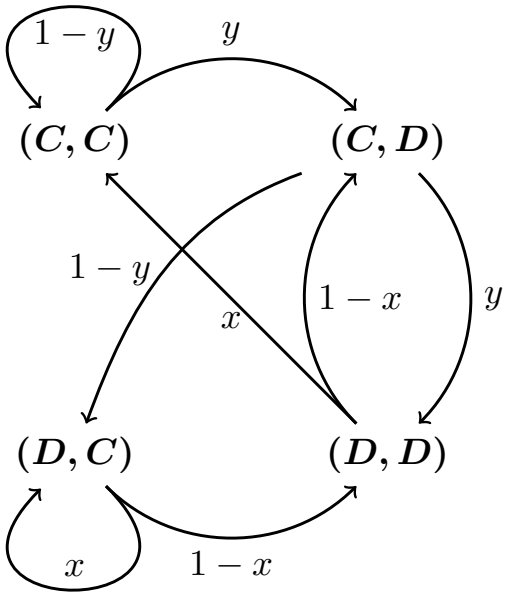


Figure 3.4: Markov chain for Pavlov

Step 2 - Construct the transition matrix.

$$T = \begin{array}{c} \\ (C,C) \\ (C,D) \\ (D,C) \\ (D,D) \end{array} \begin{array}{c} \begin{array}{cccc} (C,C) & (C,D) & (D,C) & (D,D) \end{array} \\ \begin{pmatrix} 1-y & 0 & 0 & x \\ y & 0 & 0 & 1-x \\ 0 & 1-y & x & 0 \\ 0 & y & 1-x & 0 \end{pmatrix} \end{array} \tag{3.1}$$

Step 3 - Find the steady state distribution.

$$\pi = \begin{bmatrix} \frac{x(1-x)}{2y(1-x)+x(1-x)+y(1-y)}, \\ \frac{y(1-x)}{2y(1-x)+x(1-x)+y(1-y)}, \\ \frac{y(1-y)}{2y(1-x)+x(1-x)+y(1-y)}, \\ \frac{y(1-x)}{2y(1-x)+x(1-x)+y(1-y)} \end{bmatrix} \tag{3.2}$$

Step 4 - Calculate the expected score.

$$F = \pi \cdot \begin{bmatrix} 3 \\ 0 \\ 5 \\ 1 \end{bmatrix} = \frac{3x(1-x) + y(1-x) + 5y(1-y)}{2y(1-x) + x(1-x) + y(1-y)} \tag{3.3}$$

Whilst not technically part of the Fingerprinting process, it is now useful to plot the resulting function, shown in Figure 3.5.

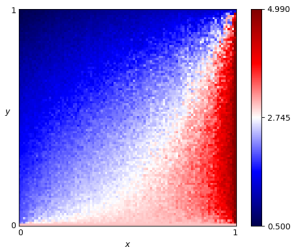


Figure 3.5: Fingerprint example for Pavlov

In the next section, it is shown that any strategy can be represented as a Finite State Machine. This **original** result allows the definition of the Dual to be extended and therefore allow any strategy to be used as a probe.

3.7 Proof of FSM for every strategy

Theorem 2 *Given a deterministic strategy α and 2 histories h_1, h_2, then for all games of length $n \in \mathbb{N}$ there exists an FSM, M such that $\alpha(h_1, h_2)$ can be obtained from it.*

Proof 2 *Let $\sigma = \{C, D\}$ be the set of symbols representing the input of M and*

$$S = \bigcup_{i=0}^{n} \{C, D\}^i \times \{C, D\}^i$$

be the set of states of M.

Then the transition function $\delta : S \times \sigma \to S$ which maps states and actions to a new state can be defined as:

$$\delta((h_1, h_2), a) = (h_1\alpha(h_1, h_2), h_2 a).$$

where a is the opponent's move. Thus the 2 histories h_1, h_2 and an opponents action, a, are mapped to a new state $(h_1\alpha(h_1, h_2), h_2 a)$. Here $h_1\alpha(h_1, h_2)$ is the concatenation of the history, h_1, and the action specified by the strategy, α. Similarly $h_2 a$ is the concatenation of the history, h_2, and the opponents action, a.

Finally the final states of M denoted by F is just $F = \{C, D\}^n \times \{C, D\}^n \subseteq S$.

In essence, if the number of turns in a game is known, any strategy can be represented as an FSM. Thus the FSM of a more complex strategy can be built, for example, Majority. Majority plays in the following way:

- If the opponent has Cooperated the majority of the time, Majority will Cooperate.
- If the opponent has Defected the majority of the time, Majority will Defect
- Note - the strategy shown is technically Soft Majority, if the opponents Cooperations and Defections are equal it will Cooperate. Hard Majority would Defect in this situation.

This implies that the strategy Majority requires knowledge of all previous states and therefore, could not be represented as an FSM. However, by using Theorem 2 this is now possible if the number of turns in a game is known. In Figure 3.6 the FSM for Majority in a game with 4 turns is presented.

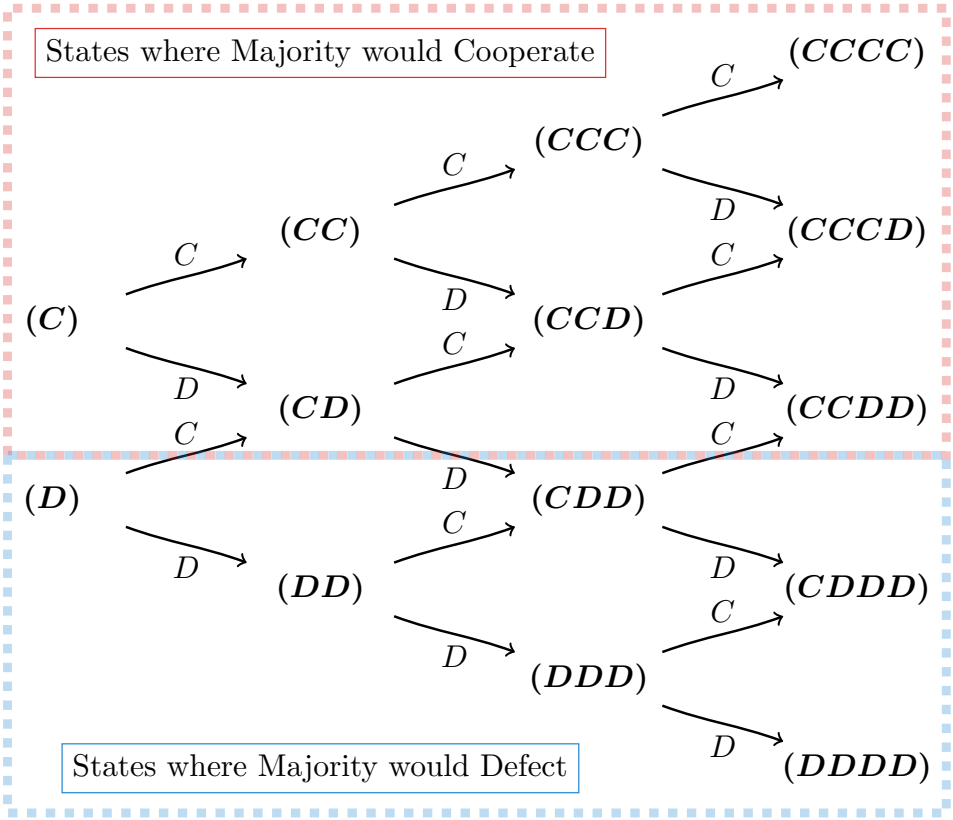


Figure 3.6: FSM for Majority in a game with 4 Turns

Theorem 2 also impacts the Fingerprint process; the result allows the definition of the Dual to be extended to all strategies. Therefore, any strategy can be used as a probe.

3.8 Extended Dual

As described in Section 3.4, traditionally the Dual is only defined for strategies with an FSM representation. This implies that the Dual transformation could not be applied to a strategy without an FSM representation. However, Theorem 3.1 states that every strategy can be represented as an FSM if the number of turns in the game is known.

Definition 7 *Strategy A' is said to be the **Extended Dual** of strategy A if, for a game of finite length $n \in N$, A and A' can be written as finite-state machines that are identical except that their responses are reversed.*

This allows the Dual transformation to be applied to all strategies and therefore, any strategy can be used as the Probe in a Fingerprint (see Definition 5).

The definitions presented throughout this Chapter will now be used to develop code to Fingerprint any strategy with any probe. This is explained in detail in Chapter 4.

Chapter 4

Implementation

This chapter will explain how the method of fingerprinting was implemented within the Axelrod-Python Library. Each of the definitions presented in Chapter 3 directly correspond to functions which are described below.

4.1 The Joss-Ann

A formal definition of the Joss-Ann is given in Section 3.2. Given a probability distribution (x, y) the Joss-Ann Cooperates with probability x, Defects with probability y, and plays the original move with probability $1 - x - y$. This can be implemented very simply as seen in Algorithm 1. First, a random number between 0 and 1 is generated. The thresholds for Cooperation, Defection and original move are then $x, x + y$ and 1 respectively.

```
1  while Game is being played do
2  |  p ← Random number;
3  |  if p ≤ Cooperation Threshold then
4  |  |  Next Move ← C;
5  |  else if p ≤ Defection Threshold then
6  |  |  Next Move ← D;
7  |  else
8  |  |  Next Move ← Original choice of Strategy;
9  |  end
10 end
11 return Next Move
```

Algorithm 1: The Joss-Ann of a Strategy

It is possible to apply a strategy transformation to a strategy that has already been transformed. When fingerprinting, the Joss-Ann must be applied over the Dual.

4.2 The Dual

The Dual of a strategy is defined such that when the original strategy and the Dual are presented with identical histories they will return opposite actions, as outlined in Definition 4. In previous literature this has only applied to strategies with FSM representations [2, 3, 4, 5, 6, 7] and the transformation was applied by changing the FSM. However in Section 3.8 this is extended to include all possible strategies, and an algorithm for achieving this will now be given.

```
while Game is being played do
    if First Turn then
        create copy of original strategy;
    end
    simulate original strategy;
    update original strategy's history/internal state;
end
return Flip of original strategy's move
```

Algorithm 2: The Dual of a Strategy

Due to the way it operates, the Dual relies on knowledge of the internal state of the original strategy. This is not something that can be inferred from source code, but the result can be produced by having the original strategy as an attribute of the Dual. In effect, the Dual 'carries' a copy of the original strategy. Whenever the Dual has to submit a move, it can obtain the state of the original strategy, then compute the next move for the origin strategy, and finally flip that move.

4.3 Implementation of Fingerprinting

As defined in Section 2.5 a Fingerprint is merely a function for the expected score of a strategy when played against a Joss-Ann transformer of a probe with varying parameters (see definition 5). As part of this project, a numerical approximation has now been included in the Axelrod-Python library. This was implemented as a class that will return the numerical data that represents that analytical function on a given finite domain. It begins by taking a sample of the x, y values that may define a Joss-Ann Transformer. The strategy then plays a match against a transformer with each of the sampled values. The average score per turn can be calculated at the end of each match which corresponds to the expected score required by the analytical fingerprint function. The whole process can be repeated for reliability and the resulting scores plotted. The player interactions have been modelled as a spatial tournament within Axelrod-Python, where the strategy plays all of the probes and a probe only plays the strategy. Throughout the rest of this project, this type of spatial tournament will be referred to as an Ashlock Tournament. For an example Ashlock tournament with 9 probes, see Figure 4.1.

How well the numerical fingerprint matches the analytical one relies heavily on the choice of parameters. Specifically the `turns, repetetitons` and `step` variables. The `step` variable determines the number of x, y values taken. Listing 4 shows how a grid of points is constructed over the unit square where the distance

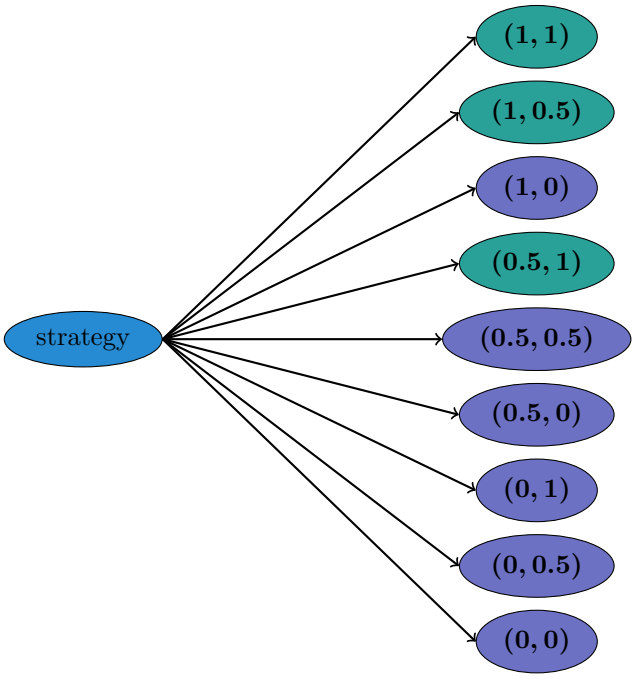


Figure 4.1: An example Ashlock Tournament for a strategy against 9 probes

between each point is taken as `step`. Therefore, a smaller `step` value means more points are created and so greater detail is included in the plot (similar to pixels).

```
def create_points(step):
    """Creates a set of Points over the unit square.
    A Point has coordinates (x, y). This function constructs points that are
    separated by a step equal to `step`. The points are over the unit
    square which implies that the number created will be (1/`step` + 1)^2.
    Parameters
    ----------
    step : float
        The separation between each Point. Smaller steps will produce more
        Points with coordinates that will be closer together.
    Returns
    ----------
    points : list
        of Point objects with coordinates (x, y)
    """
    num = int((1 / step) // 1) + 1
    points = [Point(j, k) for j in np.linspace(0, 1, num)
              for k in np.linspace(0, 1, num)]

    return points
```

Listing 4: Axelrod-Python code to create a sample of x, y points

The `turns` variable determines how many interactions there will be in a match. Enough turns must be selected to ensure that steady long term behaviour is reached otherwise the average score per turn can be wildly inaccurate [48]. However, once this state is reached, extending the number of turns has a minimal effect on the accuracy of the plot. The `repetitions` variable decides how many times the tournament would be repeated. The Axelrod-Python implementation of fingerprinting is a random process (due to the Joss-Ann) and high repetitions helps to reduce the effects of this.

Whilst Axelrod-Python has the ability to use any strategy as a probe, the default is set to TitForTat. This has been done in order to align with previous research [4, 5, 7].

4.4 Comparison of Analytical and Numerical Plots

In figure 4.2, several analytical fingerprints from previous literature are shown [4, 2]. Colourings or shadings are used to make certain features stand out, and an attempt to replicate this behaviour was implemented in Axelrod-Python. The popular plotting library, matplotlib, has many options for different colour maps which are demonstrated in Appendix A.1.

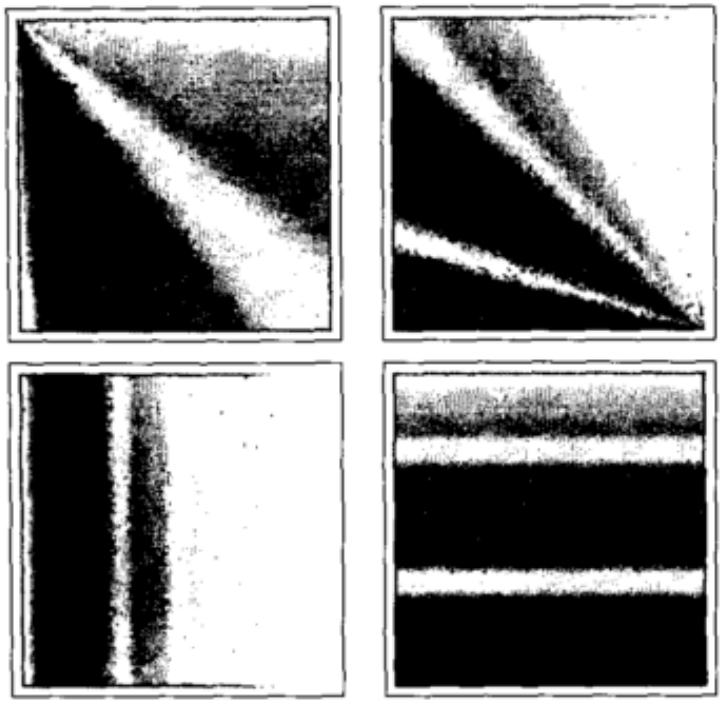
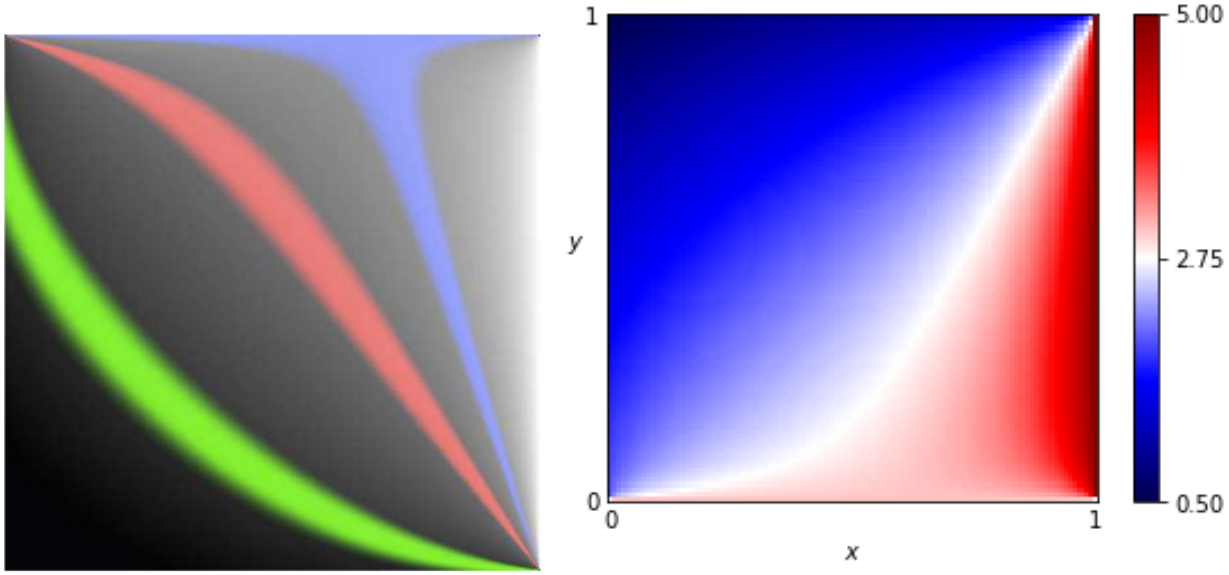


Figure 4.2: Shaded plots of the fingerprint functions for the strategies TitForTat, Psycho, AllD and AllC, in reading order from [4]

Using the analytical fingerprints from previous literature [4, 2], and the fingerprint formulae provided alongside them, the most appropriate colour map was chosen. The colour map Seismic [25] was selected as a default due to its divergent properties (although all colour maps are available within the library). With

divergent colour maps, all extreme values (high or low) are coloured, whilst mid range values are left white [38]. This highlights areas of interest, and in Figure 4.3 it can be seen that this matches previous work well.



(a) WSLS fingerprint from previous literature [2]

(b) Analytical WSLS fingerprint demonstrating Seismic colouring

Figure 4.3: A comparison of a fingerprint plot from previous literature to asses the suitability of the Seismic colour map [2]

With the knowledge that the choice of colour map is appropriate, a comparison can now be made between analytical fingerprints and numerical ones obtained via the Axelrod-Python library. Table 4.1 gives the analytical fingerprint functions of several well known strategies that will then be used to validate the numerical versions.

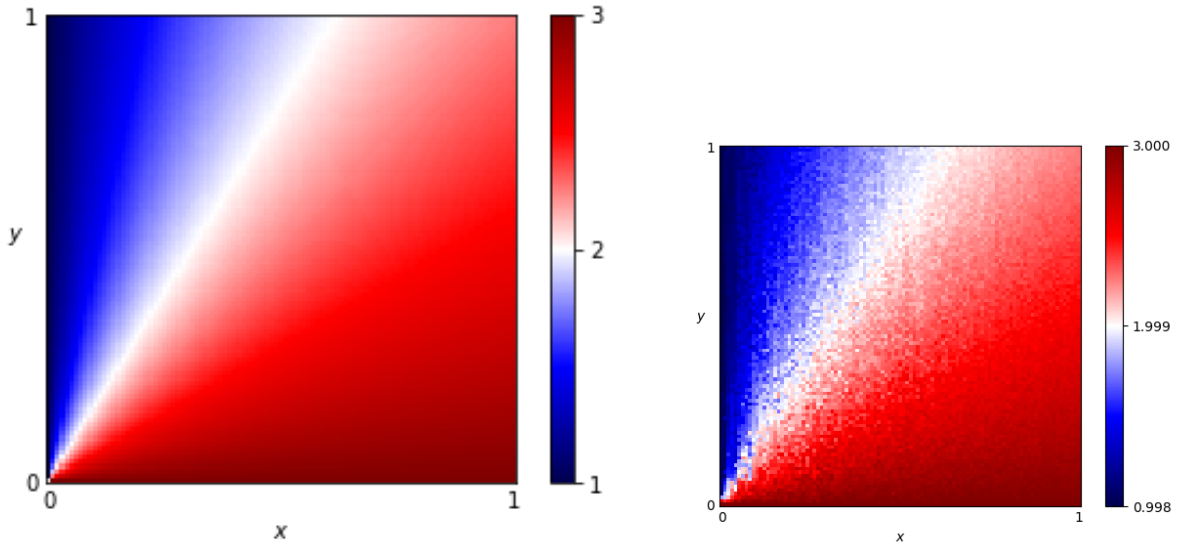
Strategy	Analytical Fingerprint Function
TitForTat	$\frac{y^2+5xy+3x^2}{(x+y)^2}$
Psycho (Anti TitForTat	$\frac{4(y-1)(x-1)+5(y-1)^2}{2(y-1)(x-1)+(x-1)^2+(y-1)^2}$
WinStayLoseShit (Pavlov)	$\frac{(3x+y)(x-1)+5y(y-1)}{(x+2y)(x-1)+y(y-1)}$
AllC (Cooperator)	$3-3y$
AllD (Defector)	$4x+1$

Table 4.1: A selection of analytical fingerprint functions for well known strategies. The probe used is TitForTat.

Figures 4.4 4.5 4.6 4.7 and 4.8 compare plots of known analytical fingerprint functions with numerical approximations obtained with the Axelrod-Python library. The analytical plots were created with the code seen in listing 5. The parameters `turns=500`, `repetitions=200`, `step=0.01` are as described in section 4.3. The parameter `processes=0` ensures that the function will use the maximum number of cores available on the computer.

```
import axelrod as axl
strats = [axl.TitForTat, axl.WinStayLoseShift, axl.AntiTitForTat,
          axl.Cooperator, axl.Defector]
for s in strats:
    probe = axl.TitForTat
    af = axl.AshlockFingerprint(s, probe)
    data = af.fingerprint(turns=500, repetitions=200, step=0.01, processes=0)
    p = af.plot()
    p.savefig('{}-Numerical.pdf'.format(s.name))
```

Listing 5: Code to create the numerical plots for several strategies



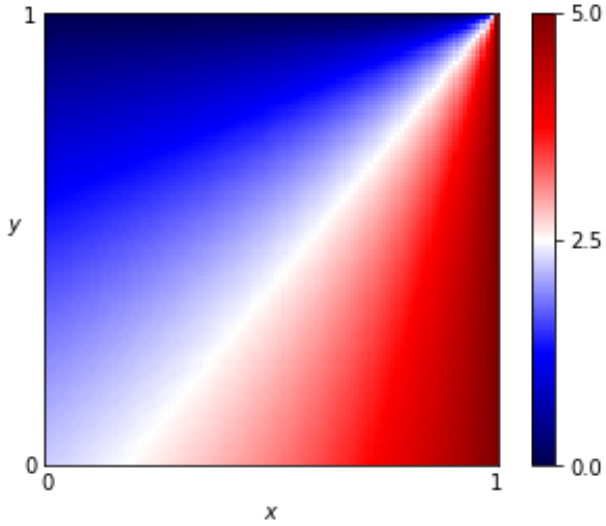
(a) Exact analytical fingerprint

(b) Numerical Fingerprint

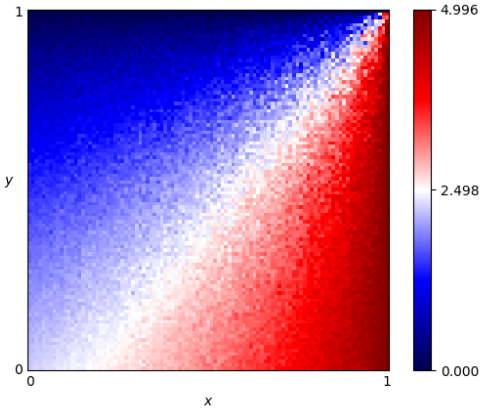
Figure 4.4: A comparison of the analytical fingerprint of TitForTat and the numerical version produced by Axelrod-Python library.

4.5 The Development Process

The Axelrod-Python library aims to follow best practice at all times with regards to development. This section outlines some of these key ideas, and how they were relevant to the implementation of the ability to produce fingerprints within Axelrod-Python.

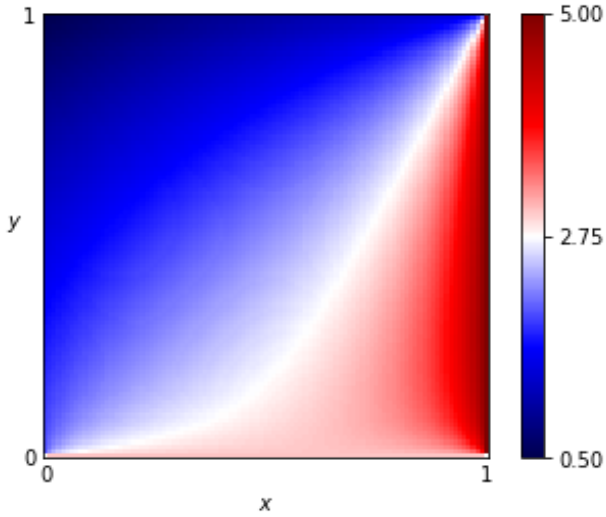


(a) Exact analytical fingerprint

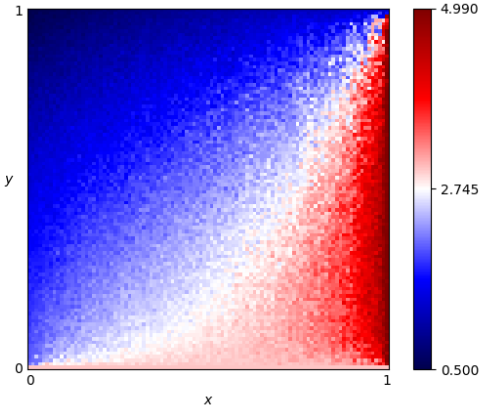


(b) Numerical Fingerprint

Figure 4.5: A comparison of the analytical fingerprint of Psycho and the numerical version produced by Axelrod-Python library.

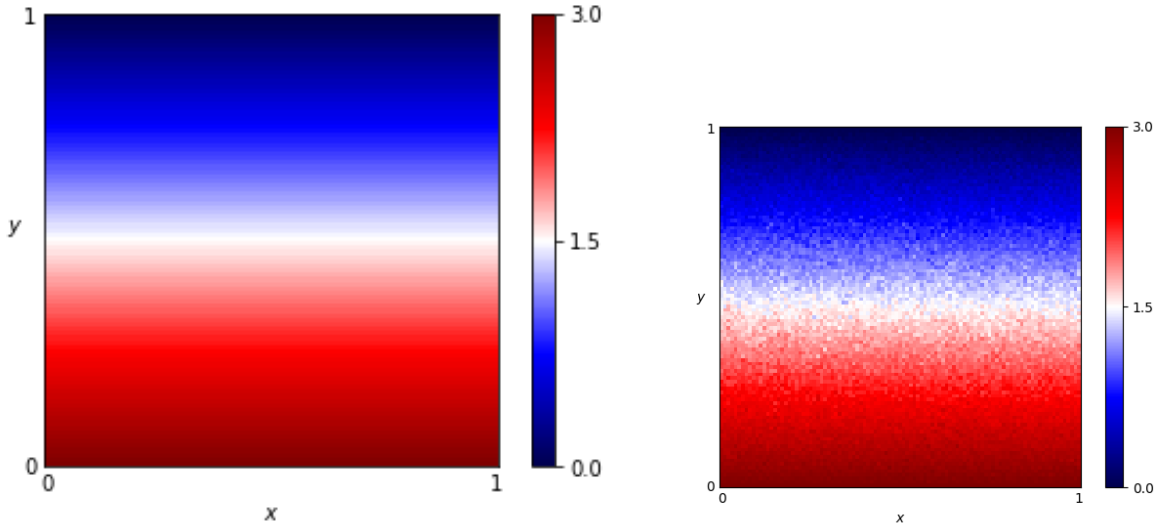


(a) Exact analytical fingerprint



(b) Numerical Fingerprint

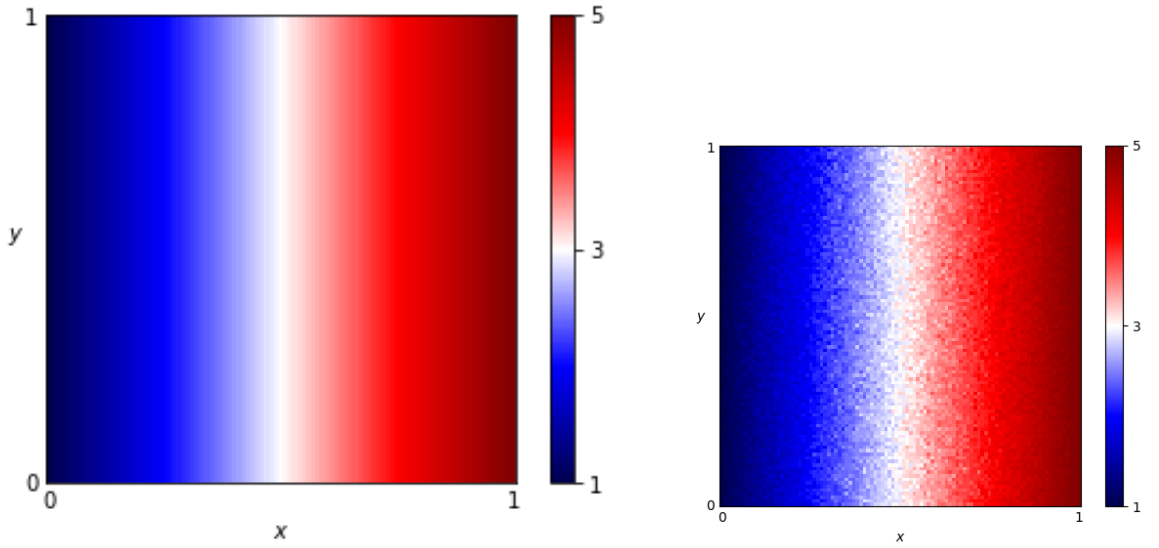
Figure 4.6: A comparison of the analytical fingerprint of WinStayLoseShit and the numerical version produced by Axelrod-Python library.



(a) Exact analytical fingerprint

(b) Numerical Fingerprint

Figure 4.7: A comparison of the analytical fingerprint of Cooperator and the numerical version produced by Axelrod-Python library.



(a) Exact analytical fingerprint

(b) Numerical Fingerprint

Figure 4.8: A comparison of the analytical fingerprint of Defector and the numerical version produced by Axelrod-Python library.

4.5.1 Version Control

Scientists face many issues when producing research in the modern world. One of the main reasons research can be hard to reproduce is due to the lack of access to underlying data and code, which reduces the opportunity for others to verify findings [54, 27]. Also, two of the biggest challenges scientists and other programmers face when working with code and data are keeping track of changes (and being able to revert them if things go wrong), and collaborating on a program or dataset [67, 36]. Both of these issues can be resolved by ensuring that all code is version controlled.

Version control systems give users the ability to save versions of files during development along with informative comments which are referred to as commit messages. Every change and accompanying notes are stored independent of the files. Commits serve as checkpoints where individual files or an entire project can be safely reverted to when necessary [45]. Version control ensures that all changes to a code base are tracked and can be traced back to a particular author. There are many version control systems available and Axelrod-Python uses Git [61] with all code hosted at Github. One of the major benefits is that every repository (including the one on Github) contains the entire history of all changes, including authorship, and can be viewed by anyone.

4.5.2 Review Process

The library also applies a strict review process, where any code submission is analysed by several members of the organisation before it can be included. This normally involves other developers requesting changes to the submission, this ensure that all source code is of the same high standard. Figures 4.9 to 4.12 shows some screen shots of the discussions, requests and suggestions made during the development process of the fingerprint code.

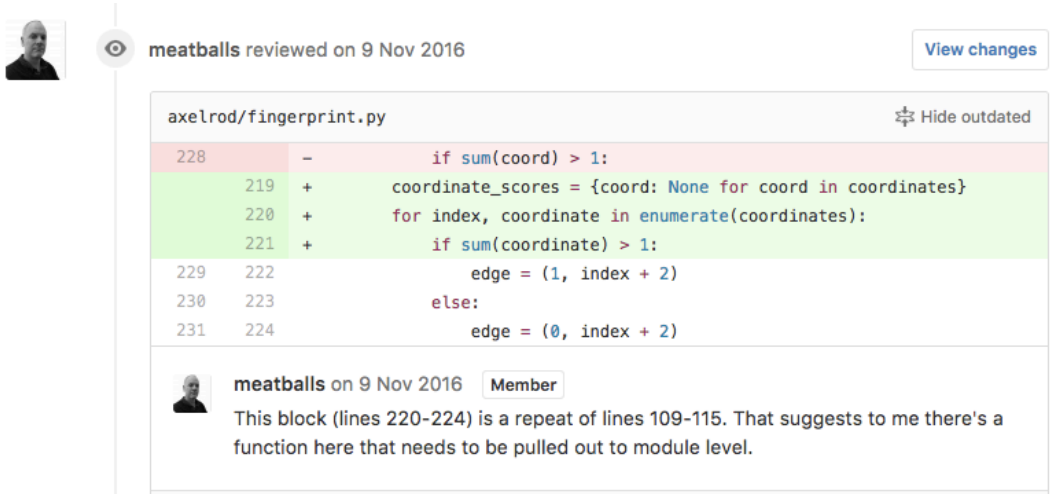


Figure 4.9: A refactoring suggestion to create a new function to avoid repetition

Figures 4.9 and 4.10 show suggestions that have been made. These suggestions do not highlight errors within the code, but are merely small recommendations from experienced developers for ways to improve.

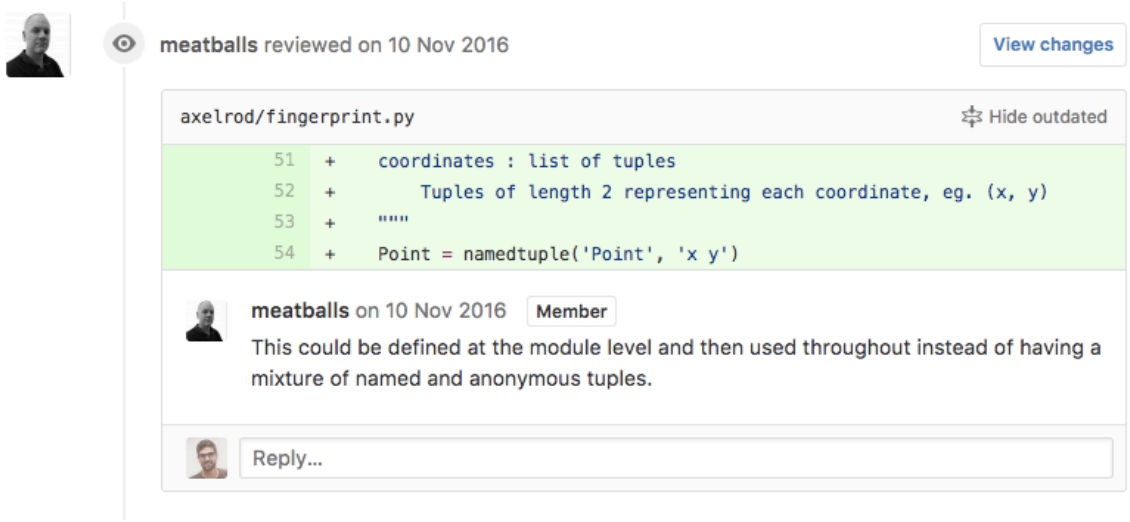


Figure 4.10: Suggestion to move some code to the module level so that it is more accessible

Figure 4.11 is an example of the power of using git. Code that is being developed elsewhere can still be used as part of this project, even though it is not yet available as part of the standard library. This is a common occurrence when developing software; some code may rely on other parts but they are not being written by the same person.

Figure 4.12 is a request to move unnecessary lines of code. Whilst these would not cause the software to break, it is bad coding practice to have unused imports and redundant lines of code. This is an example of something that would have prevented the code written from being included in the production version of the package.

4.5.3 Testing and Documentation

All code in Axelrod-Python must be tested and the most common example of testing is unittesting. A unit test is a function that tests a small pieces of code and not an entire package/module, although it sometimes can. They operate by observing the result for a specific input and comparing it with a known output, then returning whether they are the same. A result of `True` indicates that the code is behaving as intended, and a result of `False` indicates that it is not. Consequently any program relying on that code cannot be trusted to behave as intended [52, 66].

Axelrod-Python uses several external programmes to help with testing. Travis [62] provides continuous integration testing, so all tests are run automatically whenever any changes to code are made. AppVeyor [1] offers a similar service, but specialises in testing on Windows operating systems. Coveralls [16] calculates what proportion of the code is tested, and can highlight areas that are either untested or need improvement.

Figure 4.13 shows that at the point the submission was included into the main code base, all tests where passing on Travis and AppVeyor. Also, Coveralls showed that the overall test coverage had increased slightly.

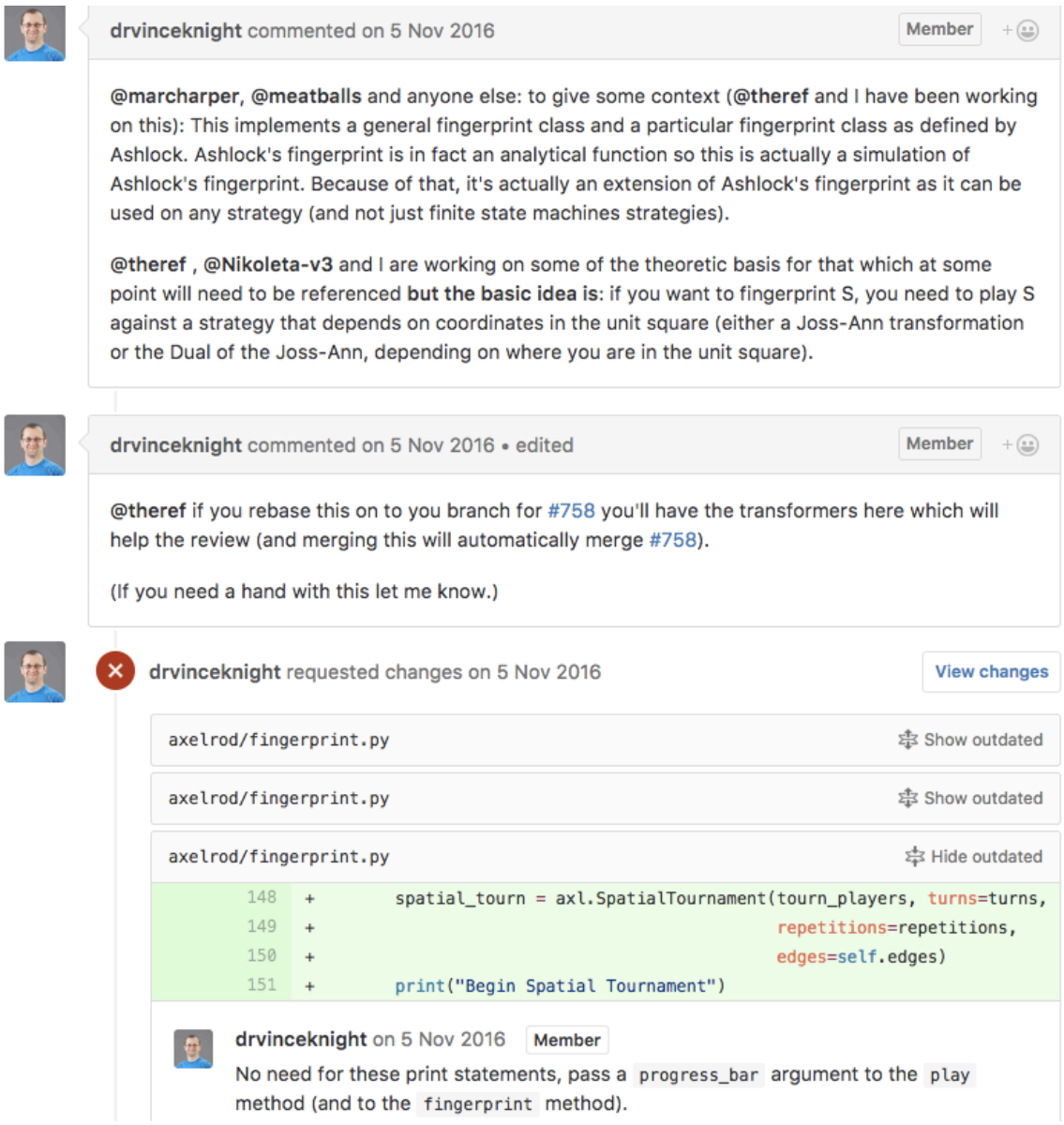


Figure 4.11: Advice to use other code being developed for the library

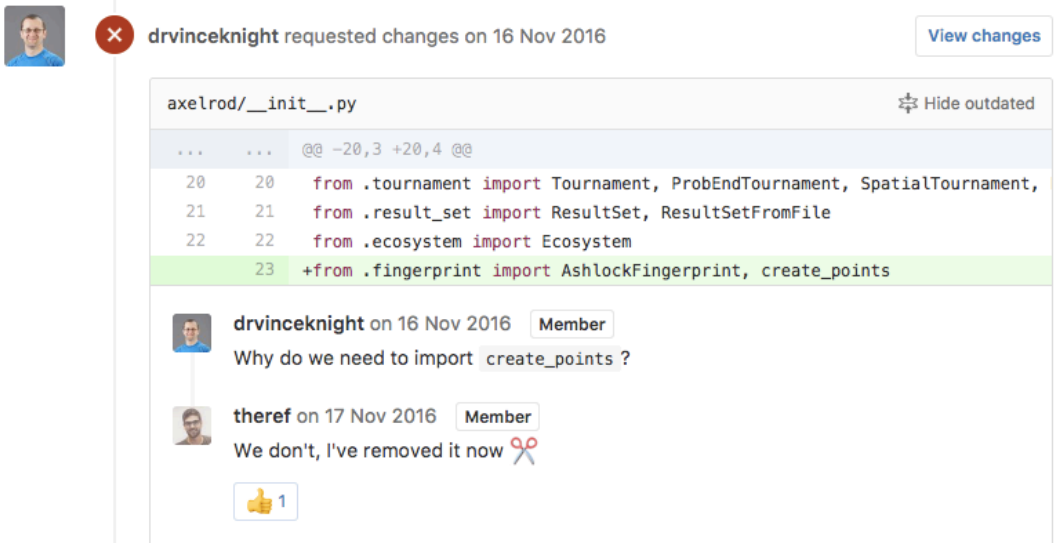


Figure 4.12: Request to remove redundant lines of code

Figure 4.13: A status report for Travis, AppVeyor and Coveralls

Hypothesis [35] is a library for property testing. It works by generating random data matching a specification, and checking that a specified guarantee still holds in that case. If it finds an example where the guarantee does not hold, it takes that example and cuts it down to size, simplifying it until it finds a much smaller example that still causes the problem [26].

Axelrod-Python also has extensive documentation [60] which benefits both developers and end users. For end users, the documentation is often presented as a tutorial to guide them through correct usage. For example, the Fingerprint documentation in Axelrod-Python (see Appendix `http://axelrod.readthedocs.io/en/latest/tutorials/further_topics/fingerprinting.html`) contains short code snippets beside a preview of the expected output. An example is shown in Figure 4.14.

The documentation is written in reStructuredText (RST), a plain text markup system [47]. The original RST code for producing Figure 4.14 is given in Listing 6. These examples also act as tests, referred to as doctests. All lines beginning with `>>>` are executed as Python code and the result is checked against the next line. This ensures that the examples in the documentation always match the source code, otherwise the doctests would fail.

```
Here is how to create a fingerprint of :code:`WinStayLoseShift` using
:code:`TitForTat` as a probe::

    >>> import axelrod as axl
    >>> axl.seed(0)  # Fingerprinting is a random process
    >>> strategy = axl.WinStayLoseShift
    >>> probe = axl.TitForTat
    >>> af = axl.AshlockFingerprint(strategy, probe)
    >>> data = af.fingerprint(turns=10, repetitions=2, step=0.2)
    >>> data
    {...
    >>> data[(0, 0)]
    3.0

The :code:`fingerprint` method returns a dictionary mapping coordinates of the
form :code:`(x, y)` to the mean score for the corresponding interactions.
We can then plot the above to get::

    >>> p = af.plot()
    >>> p.show()

.. image:: _static/fingerprinting/WSLS_small.png
     :width: 100%
     :align: center
```

Listing 6: Documentation for generating a Fingerprint, written in RST

Here is how to create a fingerprint of `WinStayLoseShift` using `TitForTat` as a probe:

```
>>> import axelrod as axl
>>> axl.seed(0)  # Fingerprinting is a random process
>>> strategy = axl.WinStayLoseShift
>>> probe = axl.TitForTat
>>> af = axl.AshlockFingerprint(strategy, probe)
>>> data = af.fingerprint(turns=10, repetitions=2, step=0.2)
>>> data
{...
>>> data[(0, 0)]
3.0
```

The `fingerprint` method returns a dictionary mapping coordinates of the form `(x, y)` to the mean score for the corresponding interactions. We can then plot the above to get:

```
>>> p = af.plot()
>>> p.show()
```

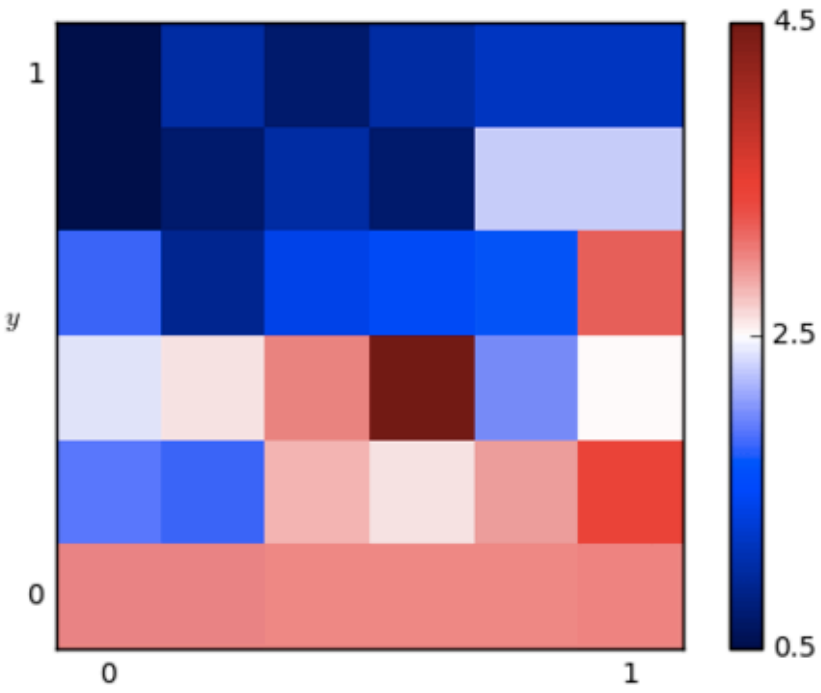
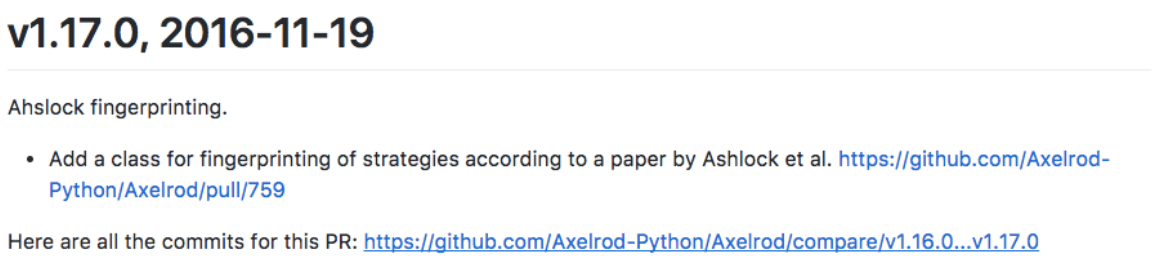


Figure 4.14: An example of how to generate a Fingerprint, taken from the Axelrod-Python documentation

4.6 Conclusion

The version of Axelrod-Python used to produce this report was v2.6.0, however Fingerprinting was first implemented in v1.17.0. Detailed notes are kept for every release of Axelrod-Python, and Figure 4.15 shows those notes for the version where Fingerprinting was first included.



v1.17.0, 2016-11-19

Ahslock fingerprinting.

- Add a class for fingerprinting of strategies according to a paper by Ashlock et al. https://github.com/Axelrod-Python/Axelrod/pull/759

Here are all the commits for this PR: https://github.com/Axelrod-Python/Axelrod/compare/v1.16.0...v1.17.0

Figure 4.15: The release notes for Axelrod-Python v1.17.0

Chapter 5

Results

In this chapter several fingerprints will be examined in detail. In Section 5.1 a breakdown of the fingerprint for TitForTat is given with an explanation of its appearance. Section 5.3 shows several plots for the strategy GoByMajority, with a varying parameter. As the parameter changes, a continuous deformation of the plot of the fingerprint can be observed. Finally, in Section 5.5 we compare the underlying data for all fingerprints of strategies within Axelrod-Python. This data is used to produce a heat plot of strategies similarity.

It may be useful to recall that a fingerprint for a strategy is produced by playing it against varying stochastic transformations of a probe. The parameters varied are x and y, as seen on fingerprints in this section. High x values correspond to high cooperation. Conversely, high y values correspond to high defection. The colour at point $\hat{x}, \hat{y}$ on the plot indicates the expected value of the strategy when played against the transformation of the probe with parameters $\hat{x}, \hat{y}$. This expected score has been approximated as outlined in Chapter 4.

5.1 Interpretation of TFT

TitForTat (see Section 2.2.1 for an explanation of its behaviour) it is one of the most well known strategies for Prisoner's Dilemma. Also, it produces a clean and easy to understand fingerprint, shown in Figure 5.1, making it an ideal place to start.

The plot slowly transitions from a strip of blue in the top left, rotating around to a large block of red in the bottom right. The blue area corresponds to low scores and it can be seen that this occurs for small values of x. However, as x increases higher scores (shown in red) quickly take over. This is exactly as expected, TitForTat plays well in highly cooperative environments. An immediate question is; why does the white stripe not follow the diagonal? The main diagonal is where $x = y$ and due to the random nature of the cooperation and defections, the expected average score for TitForTat would be 2.25, slightly half the maximum possible score. Due to the global minimum and maximum that TitForTat achieves, a score of 2.25 is not midway, and so the white line showing the middle score is off centre.

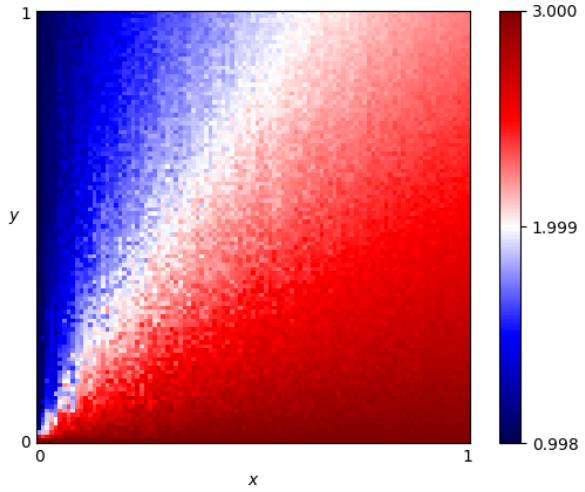


Figure 5.1: Fingerprint for TitForTat, with TitForTat also as the probe

5.2 Varying the parameter for Random

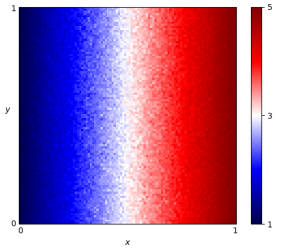
It is possible to observe how changing a parameter for a strategy will affect its behaviour by comparing the fingerprints for each parameter. As described in Section 2.2.4, Random accepts a parameter that determines the probability of cooperation. We can vary this parameter and create a new fingerprint each time. In Figures 5.2b to 5.2j the variable is increased from 0.1 to 0.9 in steps of 0.1. When the parameter equal 0 or 1, Random is equivalent to the strategy Defector or Cooperator respectively. Those fingerprint plots are shown in Figures 5.2a and 5.2k.

5.3 GoByMajority for different parameters

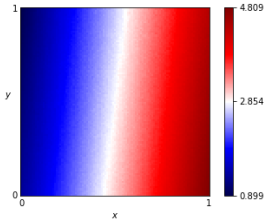
Figures 5.2a to 5.2f show fingerprints for the strategy GoByMajority (see section 3.7) with increasing memory depth when probed by TitForTat Whilst the intricate nature of the fingerprints is hard to interpret, one major observation can be made. As the memory depth increases, a region in the middle of the right hand side of the plot transitions from red to blue. Blue areas correspond to lower scores, and so the strategy actually performs worse in this area when it has a larger memory. This reason for this is not immediately apparent, and would be interesting to investigate in further research.

5.4 Random and Cycler(CDDC) with Different probes

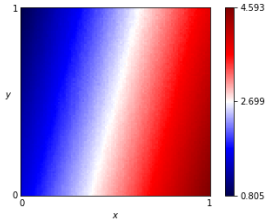
The importance of probe selection will now be demonstrated. Thus far, all fingerprints shown have used TitForTat as a probe, but this is not always enough. For example, Random and Cycler(CDDC) produce



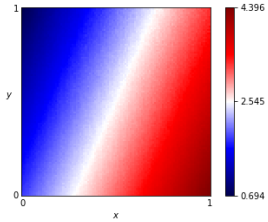
(a) Defector



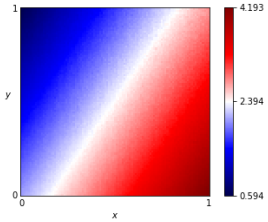
(b) Random(0.1) - will cooperate 10% of the time



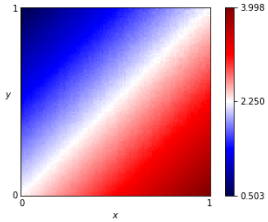
(c) Random(0.2) - will cooperate 20% of the time



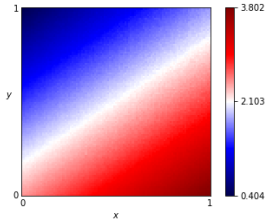
(d) Random(0.3) - will cooperate 30% of the time



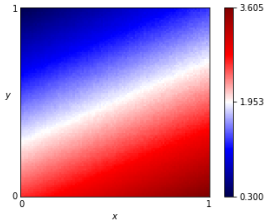
(e) Random(0.4) - will cooperate 40% of the time



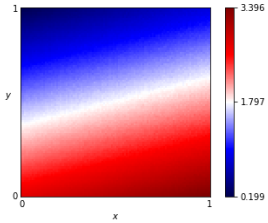
(f) Random(0.5) - will cooperate 50% of the time



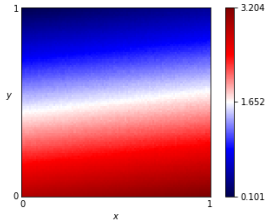
(g) Random(0.6) - will cooperate 60% of the time



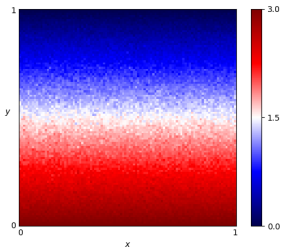
(h) Random(0.7) - will cooperate 70% of the time



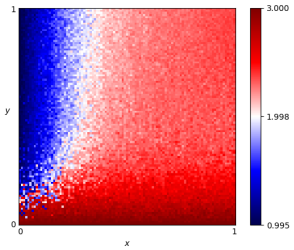
(i) Random(0.8) - will cooperate 80% of the time



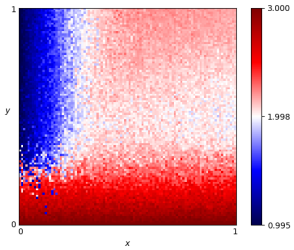
(j) Random(0.8) - will cooperate 90% of the time



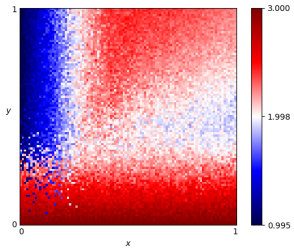
(k) Cooperator



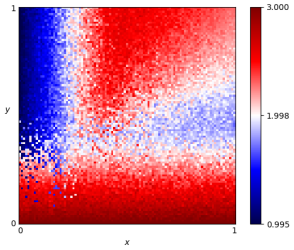
(a) GoByMajority with memory depth 5



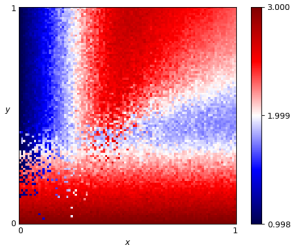
(b) GoByMajority with memory depth 10



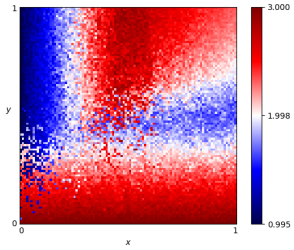
(c) GoByMajority with memory depth 20



(d) GoByMajority with memory depth 40

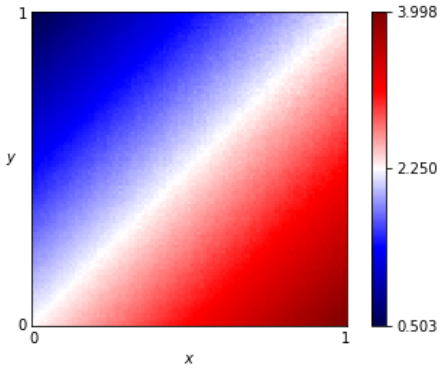


(e) GoByMajority with memory depth 75

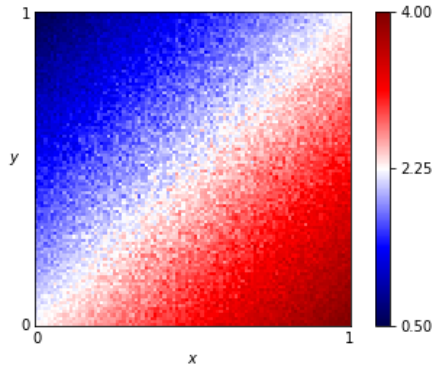


(f) GoByMajority with memory depth infinite

identical fingerprints when probed with TitForTat, as shown in 5.2



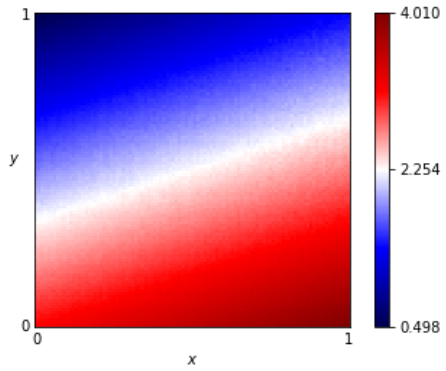
(a) Fingerprint for Random



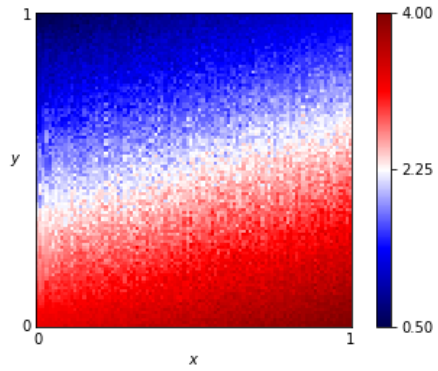
(b) Fingerprint for Cycler(CDDC)

Figure 5.2: Fingerprints for Random and Cylcer(CDDC) when probed by TitForTat

Even when changing the probe to TitForTwoTats or TwoTitsForTat the Fingerprints remain the same, as shown in Figures 5.3 and 5.4.

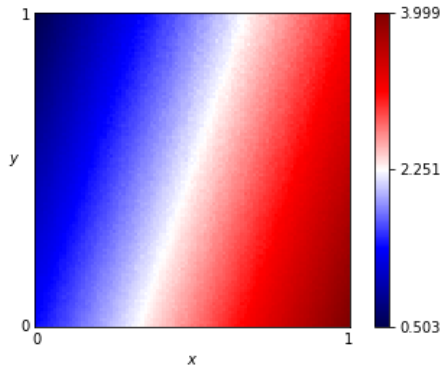


(a) Fingerprint for Random

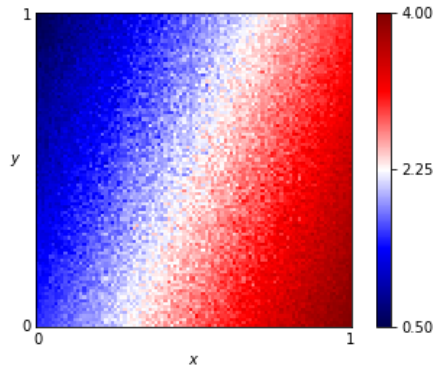


(b) Fingerprint for Cycler(CDDC)

Figure 5.3: Fingerprints for Random and Cylcer(CDDC) when probed by TitForTwoTat



(a) Fingerprint for Random



(b) Fingerprint for Cycler(CDDC)

Figure 5.4: Fingerprints for Random and Cylcer(CDDC) when probed by TwoTitsForTat

5.5 LSE plot and table

As described several times, a fingerprint for a strategy is merely the expected score when it plays against a probe with varying parameters. All the fingerprint plots shown thus far are approximations of this, where the score for specific values of x, y is calculated and then plotted. Within Axelrod-Python all these underlying scores are recorded in a `.csv` file and can therefore be compared directly.

For two strategies A and B where $s_{x,y}^{(A)}$ is the score for strategy A at point x, y and P_x, P_y are the set of points taken by x, y, we can compute a rudimentary value of 'similarity' by calculating the Mean Square Error:

$$\text{MSE}(A, B) = \sum_{x \in P_x} \sum_{y \in P_y} \frac{(s_{x,y}^{(A)} - s_{x,y}^{(B)})^2}{|P_x| \times |P_y|}$$

When $\text{MSE}(A, B) \approx 0$, the two strategies A, B are very similar, and so further investigation should be carried out between them. After calculating $\delta(A, B)$ for all pairs of strategies, a matrix plot can then be created giving an overview of the similarity between the set of strategies. This is shown in Figure 5.5.

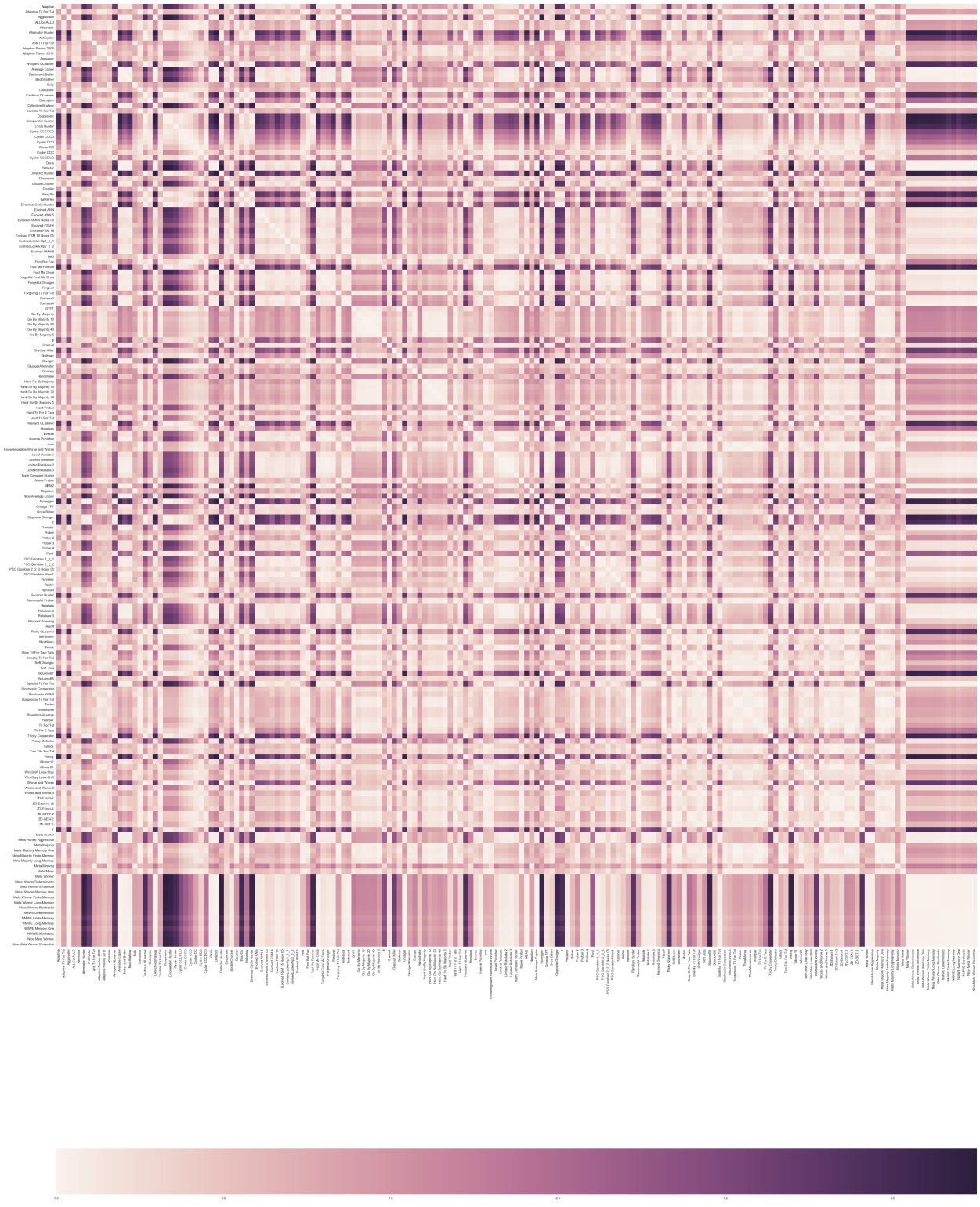


Figure 5.5: Caption here

Chapter 6

Machine Learning

In this chapter a new approach to identifying strategies that are the same will be presented. An Ashlock Tournament will be created for many strategies, and the summary of the results of this tournament will then be used to train a Machine Learning model. The performance of the model will then be assessed.

6.1 Utilising the Ashlock Tournament

As described in Section 4.3, in an Ashlock tournament all strategies entered play against a probe strategy for a range of parameters and do not play each other. In Chapter 4 the average scores from this tournament are used to approximate the analytical Fingerprint function. However, more information is available than just the score. For the purposes of the machine learning approach, the following statistics are collected from the Ashlock tournament (these are automatically calculated within Axelrod-Python):

- Strategy Name
- Rank - Where the strategy ranked overall in the tournament
- Median Score
- Cooperation Rating
- Wins
- Initial C rate
- CC rate
- CD rate
- DC rate
- DD rate
- CC to C rate

- CD to C rate
- DC to C rate
- DD to C rate

An example of the results file for strategies that have been played in an Ashlock Tournament is given in Table 6.1.

6.2 Training the model

At this point a significant assumption must be made. Strategies with different names are assumed to be different, and this is how the model will be trained.

To train the model, all strategies were entered into an Ashlock Tournament and a results file was generated, as described in Section 6.1. Multiple copies of each strategy where included to compensate for the Ashlock Tournament being a stochastic process. Each copy of a strategy was treated separately and thus had an individual line in the results file.

Once the results had been obtained, a new set of data was created that compared all strategies against each other. For two strategies A, B with variables $x_i^{(A)}, x_i^{(B)}$, a ratio $r_i^{(A,B)}$ is calculated for each possible variable x_i where:

$$r_i^{(A,B)} = \frac{\min(x_i^{(A)}, x_i^{(B)})}{\max(x_i^{(A)}, x_i^{(B)})}$$

In the case where both variable equal zero, instead of the ratio being undefined it is set to 1 to demonstrate that the original variables were equal. The resulting data is shown in Table 6.2.

By comparing the names of the two strategies, a new column can be added to the data containing a boolean variable where a value of 1 implies that the two strategies are the same, and 0 if they are different. This comes from the assumption that was stated at the beginning of the Section; that strategies with different names are assumed to be different. Conversely, this also ensures that different copies of the same strategy will still be considered equal.

Figure 6.1 gives an overview of this process with every colour of Purple, Red, Green and Blue representing a different strategy. As mentioned previously several copies of each strategy are used which would produce several nodes of the same colour in Figure 6.1a, but these have been omitted for clarity. Likewise, in Figure 6.1b there should be far more rows for each colour but these would clutter the diagram.

Following the diagram from left to right, it begins by playing an Ashlock Tournament as described in Section 6.1. The results of the the tournament are collected in the next step, and an example of this is given in Table 6.1. In the following stage a new table is created that compares the results of each strategy, again an example is given in Table 6.2. Once this table has been created, a sample is taken from all the strategies included in the initial Ashlock Tournament. Rows from the table containing **only** these strategies are then used to train the model. In Figure 6.1, the Red and Blue strategies were selected, resulting in rows being selected that contained information relating exclusively to the Red and Blue strategies. Finally, all remaining rows

Name	Median_score	Cooperation_rating	Wins	Initial_C_rate	CC_rate	CD_rate	DC_rate	DD_rate	CC_to_C_rate
Defector	0.59536	0.0000	22.0	0.0	0.0000	0.0000	0.4942	0.5058	0.0
Win-Stay Lose-Shift	0.46868	0.5528	7.0	1.0	0.3302	0.2226	0.2264	0.2208	1.0
Bully	0.46388	0.5024	11.0	0.0	0.1606	0.3418	0.3350	0.1626	0.0
Alternator	0.44824	0.5000	13.0	1.0	0.2484	0.2516	0.2490	0.2510	0.0
Tit For Tat	0.43360	0.5144	0.0	1.0	0.3680	0.1464	0.1446	0.3410	1.0

Name	CD_to_C_rate	DC_to_C_rate	DD_to_C_rate
Defector	0.0	0.0	0.0
Win-Stay Lose-Shift	0.0	0.0	1.0
Bully	1.0	0.0	1.0
Alternator	0.0	1.0	1.0
Tit For Tat	0.0	1.0	0.0

Table 6.1: Results of an Ashlock Tournament

Name_A	Name_B	Median_score_r	Cooperation_rating_r	Wins_r	Initial_C_rate_r	CC_rate_r	CD_rate_r	DC_rate_r
Defector	Defector	1.000000	1.0	1.000000	1.0	1.0	1.0	1.000000
Defector	Win-Stay Lose-Shift	0.787221	0.0	0.318182	0.0	0.0	0.0	0.458114
Defector	Bully	0.779159	0.0	0.500000	1.0	0.0	0.0	0.677863
Defector	Alternator	0.752889	0.0	0.590909	0.0	0.0	0.0	0.503845
Defector	Tit For Tat	0.728299	0.0	0.000000	0.0	0.0	0.0	0.292594

Name_A	Name_B	CC_to_C_rate_r	CD_to_C_rate_r	DC_to_C_rate_r	DD_to_C_rate_r
Defector	Defector	1.0	1.0	1.0	1.0
Defector	Win-Stay Lose-Shift	0.0	1.0	1.0	0.0
Defector	Bully	1.0	0.0	1.0	0.0
Defector	Alternator	1.0	1.0	0.0	0.0
Defector	Tit For Tat	0.0	1.0	0.0	1.0

Table 6.2: Comparison of the results of an Ashlock Tournament

that were not involved in training the model are collected. The model is then scored against this unseen data.

6.3 Assessing the Model

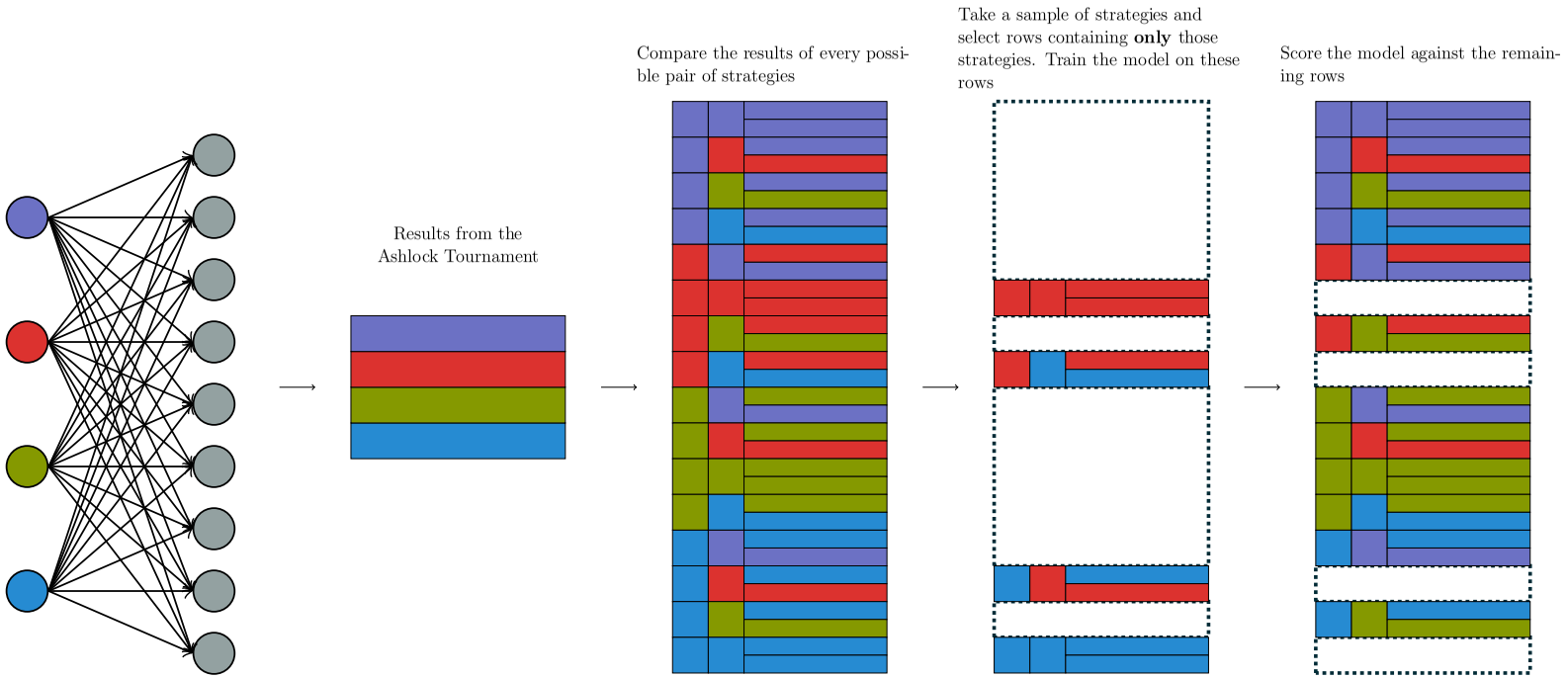
The actual Machine Learning was done using the python library `sk-learn` [43]. A choice of two models were implemented, Logistic Regression [53, 68] and Support Vector Classification [58, 64]. The process for training the models is outlined in Section 6.2 and it is mentioned that a sample must be taken from the strategies used in the initial Ashlock Tournament. Clearly the larger the sample size, the better the model should perform as it has more data to learn from. In order to decide whether to use Logistic Regression or SVC, both models were trained/tested against many samples of varying sizes.

Figure 6.2 shows how both models performed when trained on increasing sample sizes. 50 different samples were taken for each sample size and the models were re-trained and re-scored for each sample. The violin plots in Figure 6.2 show the distributions of those scores.

It can be seen that the violins in Figure 6.2b reach a maximum score faster the those in Figure 6.2a. Those is Figure 6.2b also show far less variance, and so the decision was made to use SVC as the model in any future work. A sample size between 30 and 40 was deemed acceptable, due to a good balance between high score, low variance and computational cost.

The sample was chosen by analysing the results of the Round Robin Tournament in Axelrod-Python. The strategies were ordered by their ranking in the tournament and every 5th one was selected. This gave a balance of strategies that were similar (close ranking) and those that were very different (large difference in ranking). Ordered by ranking, the 38 strategies chosen to train the model where:

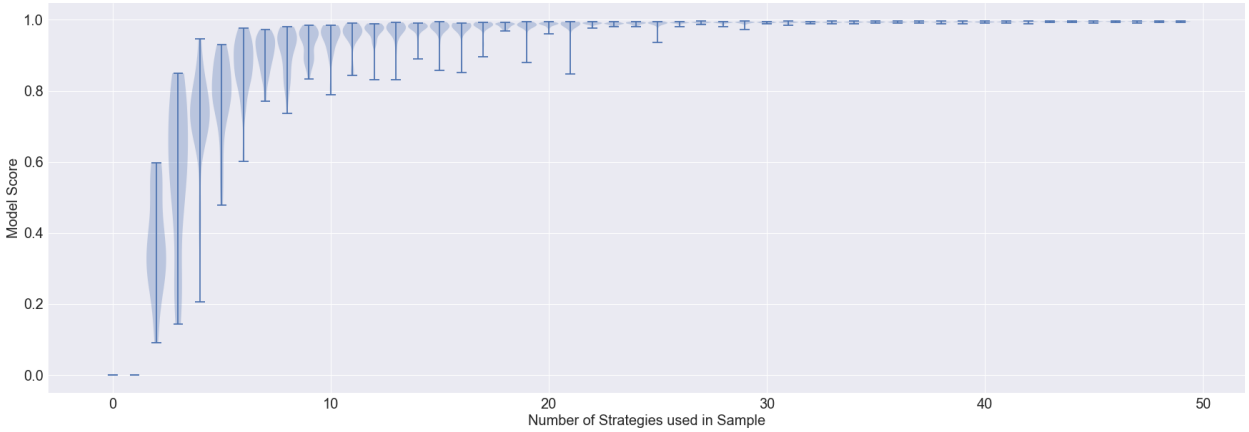
1. Evolved HMM 5
2. Evolved ANN
3. Evolved ANN 5 Noise 05
4. PSO Gambler 2_2_2 Noise 05
5. Nice Meta Winner Ensemble
6. NMWE Long Memory
7. EvolvedLookerUp1_1_1
8. Shubik
9. Retaliate (0.05)
10. Soft Joss: 0.9
11. GTFT: 0.3
12. ShortMem
13. Adaptive Pavlov 2011
14. Punisher
15. Grumpy
16. Hard Tit For Tat
17. Appeaser
18. Meta Majority Finite Memory
19. Average Copier
20. Random Hunter
21. Meta Hunter Aggressive
22. Worse and Worse 2
23. Predator
24. Stochastic Cooperator
25. Knowledgeable Worse and Worse
26. Meta Winner Finite Memory
27. Cautious QLearner
28. Random: 0.1
29. Meta Mixer
30. EasyGo
31. Hard Go By Majority: 40
32. ZD-Extort-2 v2
33. Opposite Grudger
34. Alternator



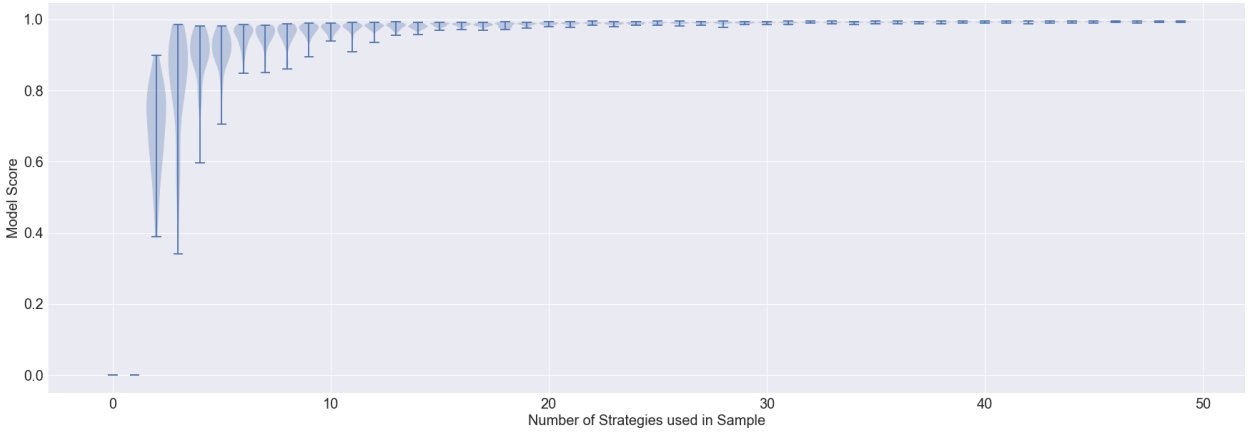
(a) The Ashlock Tournament

(b) Training the Model

Figure 6.1: The process for training the Machine Learning Model



(a) Model performance against sample size for Linear Regression



(b) Model performance against sample size for SVC

Figure 6.2: Violin Plots for both models

35. Tricky Defector
36. Negation
37. Gradual Killer
38. Cycler CCCCCD

In order to properly assess the model's performance, more information is required regarding what it can accurately predict and what mistakes are made. To do this a Confusion Matrix is constructed, as shown in Figure 6.3.

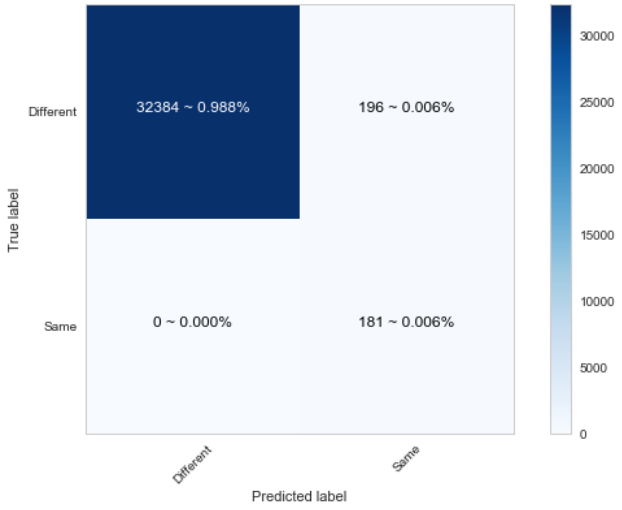


Figure 6.3: Confusion Matrix for SVC Model

Again, it is important to consider the assumption that has been made. For the True Label axis, strategies with the same name are considered equivalent and strategies with different names are assumed to be different. The Confusion Matrix shows that the model predicted the vast majority of labels correctly (values on the main diagonal). No strategies that were assumed to be the same were predicted to be different by the model. This is encouraging, as due to the assumption it is known that all strategies assumed equal are in fact identical. However, the model did predict that some strategies were equivalent, despite being assumed different. Again, this is to be expected as no allowance was made for identical strategies with different names.

Finally, it is useful to obtain a measure of the significance of each of the variables used to train the model. The Python package `sk-learn` provides utilities for obtaining p Values for the features used within the model. In the package documentation, the χ^2 test is recommended for categorical data [55], and this has been used to obtain the p Values shown in Table . This shows that all of the variables used are statistically significant and should be included in model training.

Variable	p Value
CC_rate_r	2.817481e-11
CD_rate_r	9.406413e-43
DC_rate_r	1.090114e-17
DD_rate_r	3.187864e-12
CC_to_C_r	1.237144e-07
CD_to_C_r	6.568244e-46
DC_to_C_r	1.961468e-62
DD_to_C_r	3.518073e-70
Cooperation_rating_r	5.192904e-16
Initial_C_rate_r	2.648249e-08
Median_score_r	6.603733e-03
Wins_r	1.831170e-21

Table 6.3: p Vales for all variables the model was trained on

Chapter 7

Application of the Model

In this Chapter a method for assessing the sensitivity of the model will be outlined. This is done by making several small changes to the parameters of strategies and observing whether the model can discriminate between the new altered vectors. The size of the change made to the strategy will be varied and the models performance recorded for each deviation. It is expected that there is a cut-off where the model changes from performing well to performing badly, and this gives an approximation of it's sensitivity. Finally, model will be applied to the Axelrod-Python library to determine if it contains any duplicate strategies.

7.1 MemoryOne Strategies

A MemoryOne strategy bases its decision entirely on the the last round of play. In Axelrod-Python they are defined by a vector v of length 4 that determines the probability of Cooperation given the previous rounds history, ie:

$$v = (P(C|CC), P(C|CD), P(C|DC), P(C|DD))$$

For example, a MemoryOne strategy defined by the vector $v = (1, 1, 1, 1)$ is equivalent to Cooperator. TitForTat can be implemented as a MemoryOne player with the vector $v = (1, 0, 1, 0)$, Pavlov as the MemoryOne player with $v = (1, 0, 0, 1)$ and Random as the MemoryOne player with $v = (0.5, 0.5, 0.5, 0.5)$. See Section 2.2 for explanations of how these strategies operate.

7.2 Determining the sensitivity of the Model

All possible MemoryOne strategies are defined by the space $S = [0, 1]^4_{\mathbb{R}}$. For a vector $s \in S$ and a deviation $\delta \in (0, 1)$, 16 new deviated vectors are created by taking the sum of s with all possible vectors $(\pm\delta, \pm\delta, \pm\delta, \pm\delta)$, but bounding each element in the range $[0, 1]$. For example, with $s = (0.2, 0, 0.5, 0.95)$ and $\delta = 0.1$, the 16 vectors produced are:

- $(0.3, 0.1, 0.6, 1.0)$
- $(0.3, 0.1, 0.6, 0.85)$
- $(0.3, 0.1, 0.4, 1.0)$
- $(0.3, 0.1, 0.4, 0.85)$
- $(0.3, 0.0, 0.6, 1.0)$
- $(0.3, 0.0, 0.6, 0.85)$
- $(0.3, 0.0, 0.4, 1.0)$
- $(0.3, 0.0, 0.4, 0.85)$
- $(0.1, 0.1, 0.6, 1.0)$
- $(0.1, 0.1, 0.6, 0.85)$
- $(0.1, 0.1, 0.4, 1.0)$
- $(0.1, 0.1, 0.4, 0.85)$
- $(0.1, 0.0, 0.6, 1.0)$
- $(0.1, 0.0, 0.6, 0.85)$
- $(0.1, 0.0, 0.4, 1.0)$
- $(0.1, 0.0, 0.4, 0.85)$

The 16 deviated vectors are then entered into an Ashlock tournament to produce the corresponding results table where all strategies are compared (as described in Section 6.1). The model is then be applied to this set of results, and the proportion of comparisons predicted correctly is recorded. As all the strategies have been defined by unique vectors, it is known that all the strategies are different and so the model should predict that they all different. By repeating this process for different vectors $s \in S$ whilst also varying δ, Figure 7.1 is produced.

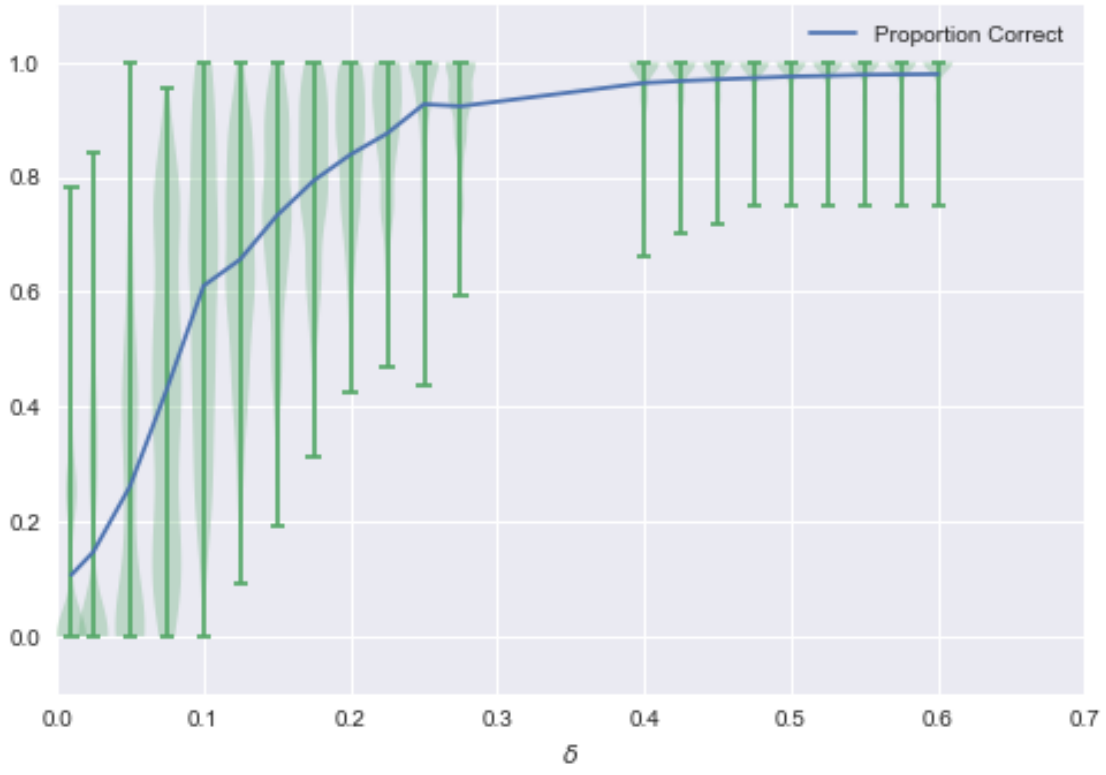


Figure 7.1: Performance of model for increasing δ

The model appears to reach a reasonable level of accuracy when $\delta > 0.3$. However, there is a large amount of variance for all δ used. This suggests that the original vector has a significant effect on how the model performs. For some MemoryOne strategies even very small deviations will be noticed by the model, but for other MemoryOne strategies it takes a much larger change before the model will recognise the difference.

7.3 Applying Model to Axelrod-Python

The final section of this report addresses the problem of determining when strategies in Axelrod-Python are identical. The assumption made in Section 6.2 must now be ignored. It stated: *Strategies with different names are assumed to be different*, but this is not necessarily the case.

In order to determine whether any strategies in Axelrod-Python were identical, all of them were entered into an Ashlock Tournament. The corresponding results table where all strategies are compared (as described in Section 6.1) was then computed, and the model was applied to this. A new table could then be produced containing pairs of strategies that the model predicted were equivalent.

In Figure 7.2, this table is presented as a Matrix Plot. Note that the leading diagonal represents strategies being equivalent to themselves, also the plot is symmetric, because if strategy A is similar to strategy B then strategy B is similar to strategy A.

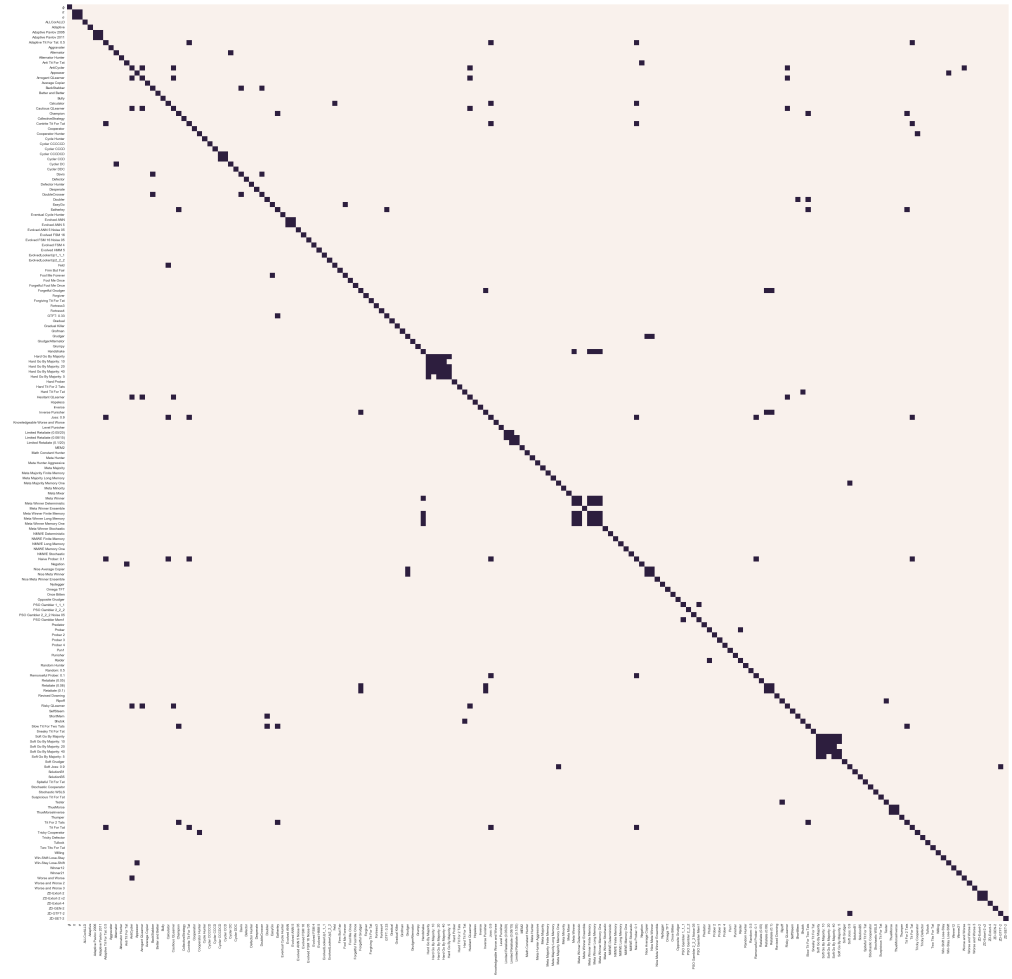


Figure 7.2: Matrix Plot of similarity

However this plot does not provide much insight. Instead it can be presented as Network Graph, as in Figure

7.3. Here nodes represent strategies, and edges link nodes that the model predicted to be equivalent. The graph is undirected for the same reason that Figure 7.2 is symmetric. For clarity, strategies that are only similar to themselves have been omitted.

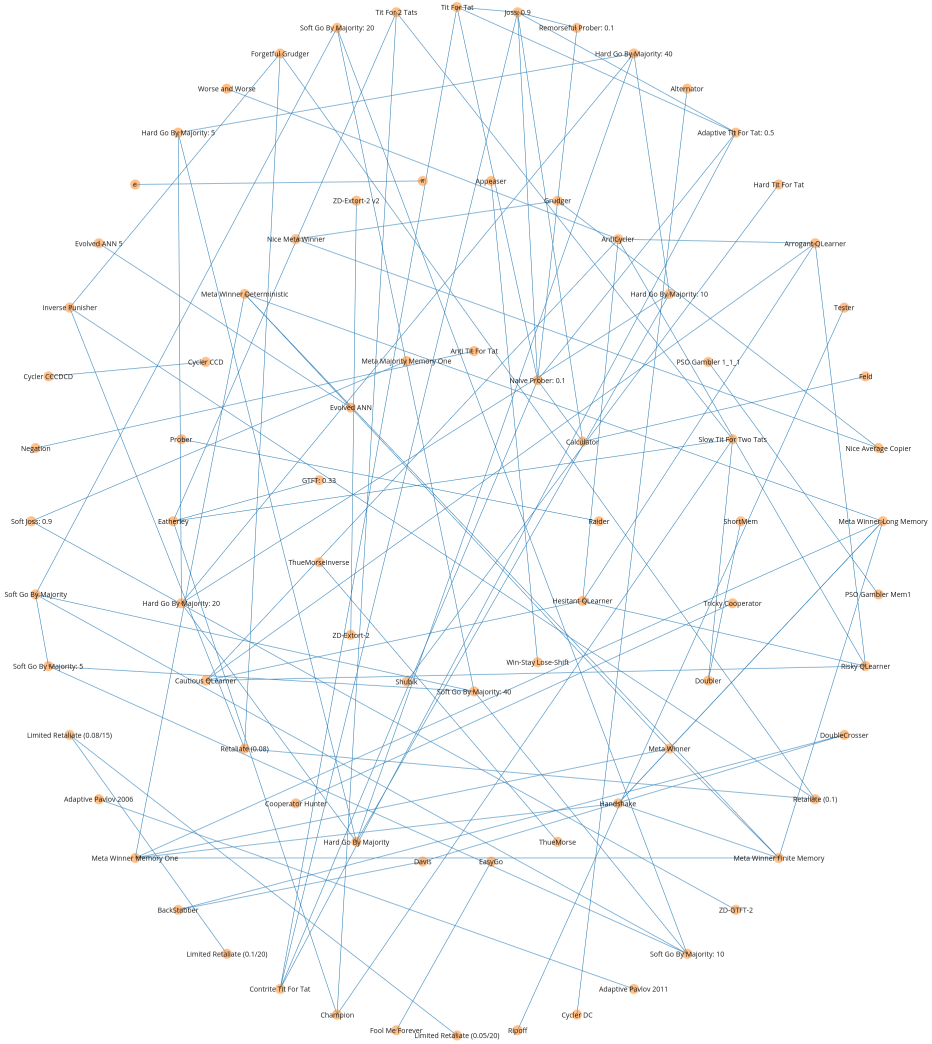
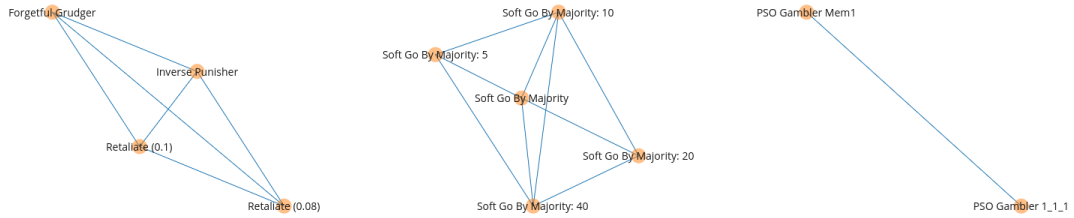


Figure 7.3: Network Graph of strategies that the model had deemed equivalent

In Figure 7.4, all of the disjoint neighbourhoods from Figure 7.3 have been separated out and some of these will now be explored. Figures 7.4b and 7.4v are perfect examples of the model exactly as expected. Figure 7.4b shows that the model predicted several SoftGoByMajority strategies to be equivalent to each other. The only difference between them was the memory depth. Again in Figure 7.4v the only difference between the

various HardGoByMajority strategies was the memory depth. However, the model was able to distinguish between all Hard and Soft version of GoByMajority, hence the two networks are disjoint.

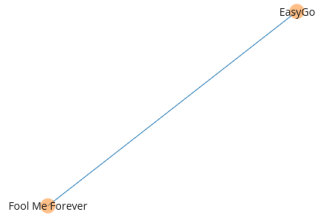
In Figure 7.4r the strategies Alternator and Cycler(DC) are deemed equivalent. Again this is encouraging as the strategies are identical except that one begins with a Cooperation and the other begins with a Defection.



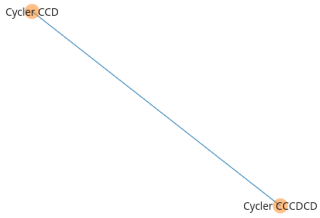
(a)

(b)

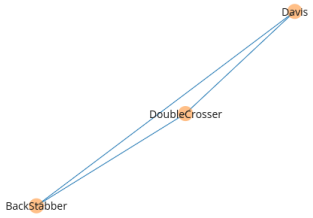
(c)



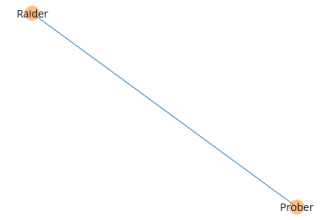
(d)



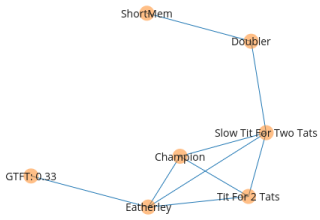
(e)



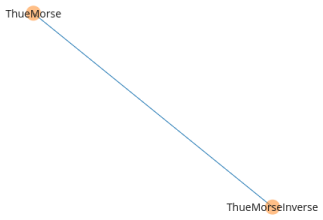
(f)



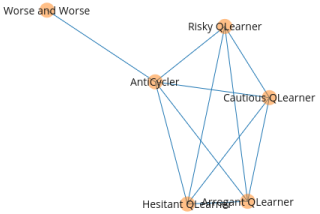
(g)



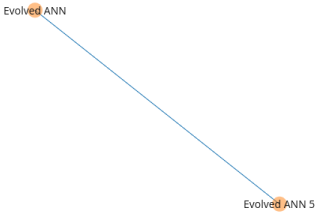
(h)



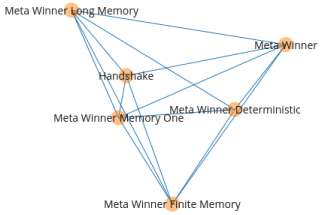
(i)


Worse and Worse
Risky QLearner
AntiCycler
Cautious QLearner
Hesitant QLearner
Arrogant QLearner

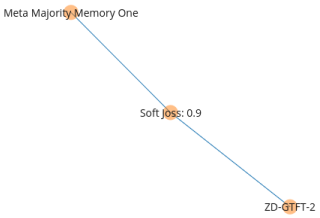
(j)


Evolved ANN
Evolved ANN 5

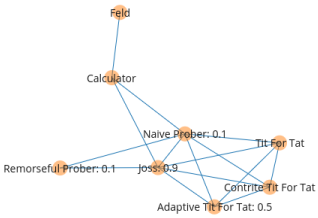
(k)


Meta Winner Long Memory
Meta Winner
Handshake
Meta Winner Memory One
Meta Winner Deterministic
Meta Winner Finite Memory

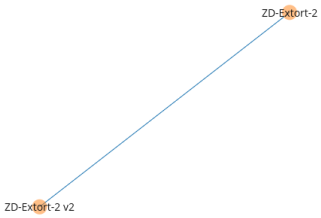
(l)


Meta Majority Memory One
Soft Joss: 0.9
ZD-GTFT-2

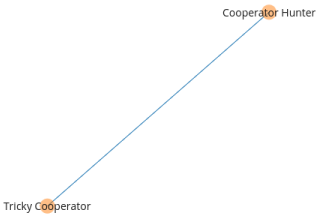
(m)


Feld
Calculator
Naive Prober: 0.1
Tit For Tat
Remorseful Prober: 0.1
Joss: 0.9
Contrite Tit For Tat
Adaptive Tit For Tat: 0.5

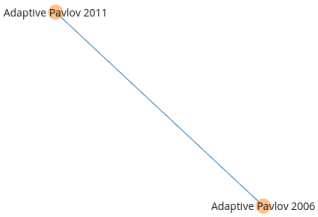
(n)


ZD-Extort-2
ZD-Extort-2 v2

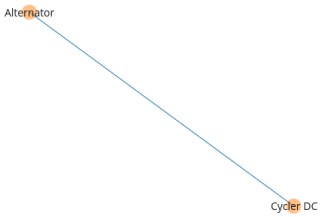
(o)


Cooperator Hunter
Tricky Cooperator

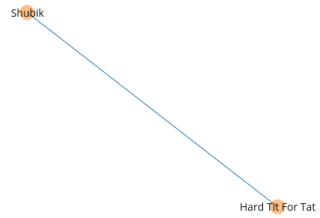
(p)


Adaptive Pavlov 2011
Adaptive Pavlov 2006

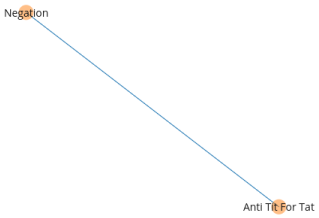
(q)


Alternator
Cycler DC

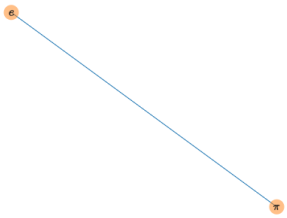
(r)



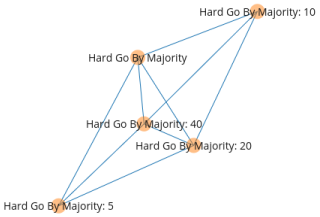
(s)



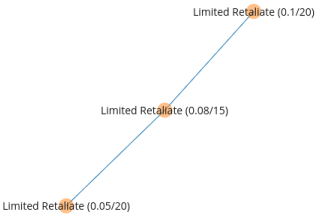
(t)



(u)

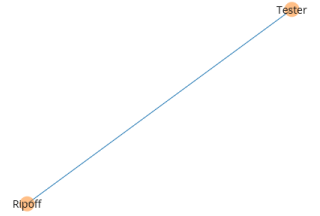


(v)

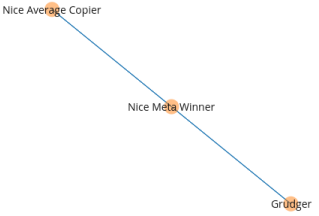


(w)

(x)



(y)



(z)

Figure 7.4: All the disjoint neighbourhoods from Figure 7.3

In Figure 7.4u, the strategies π and e are connect. These operate by attempting to make the ratio between Cooperations and Defections as close as possible to their respective mathematical constant. Interestingly, there is a third strategy of this form implemented in Axelrod-Pythoh, ϕ. The screen shot in Figure 7.5 shows the documentation for these strategies, including an approximation of the ratio they attempt to replicate.

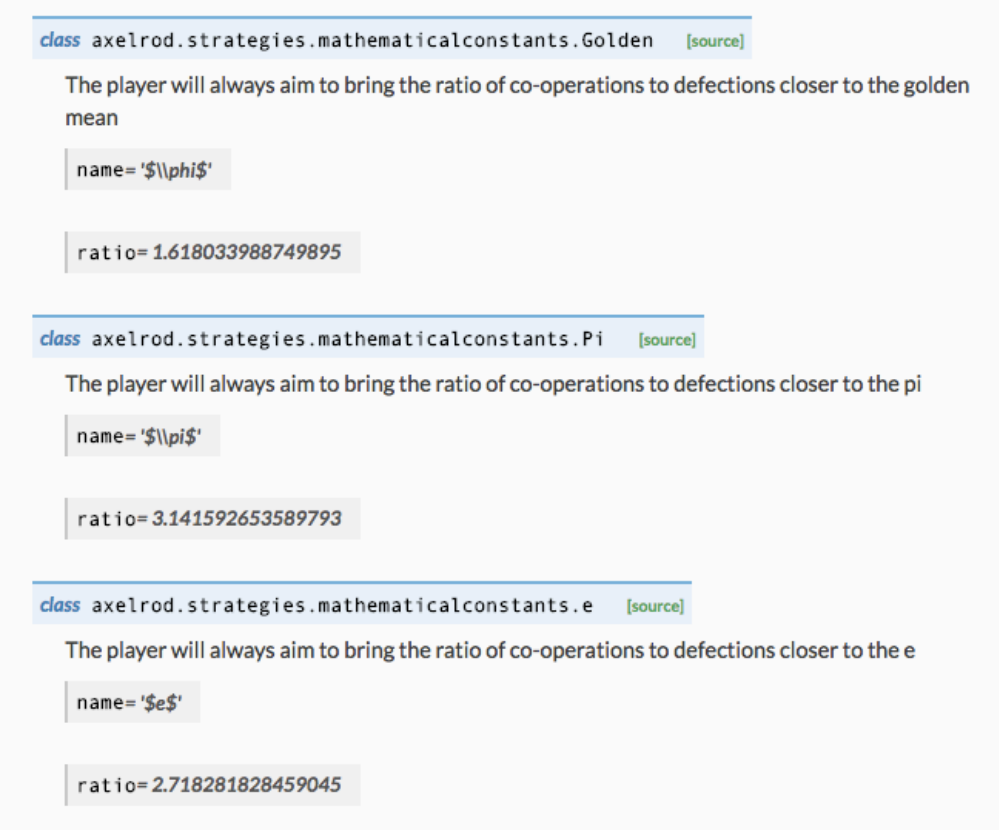


Figure 7.5: Axelrod-Python documentation for ϕ, π and e

Another neighbourhood that requires more inspection is shown in Figure 7.4j. 4 different QLearner strategies and the strategy AntiCycler form a complete graph, ie. the model predicts they are all equivalent to each other. In addition, the strategy WorseAndWorse is deemed similar to AntiCycler, but not to any of the QLearners.

AntiCycler follows a sequence of plays that has no cycles: $CDD\,CD\,CCD\,CCCD\,CCCCD\ldots$, whereas the QLearners use Q-Learning [65] in an attempt to find an optimal solution. It is understandable that different Q-Learners may be predicted equivalent by the model, as it is possible that find the same solution. However, without knowledge of the solution they find it is unclear why they are considered similar to AntiCycler and this would require further work.

Chapter 8

Conclusion

As described in Chapter 1, the Prisoner's Dilemma is a very popular model in game theory and there have been many papers written about the subject. The game has been applied to many different research areas is often used to model systems in biology [56], sociology [20], psychology [28], and economics [14]. The start of this chapter will give a brief overview of the literature and particularly relevant work will be highlighted. This is followed by an outline of how Axelrod's work is currently being reproduced by an open-source community. Finally, an introduction to fingerprinting (a technique used for identify similar strategies) and some necessary definitions and theorems are given at the end of the chapter.

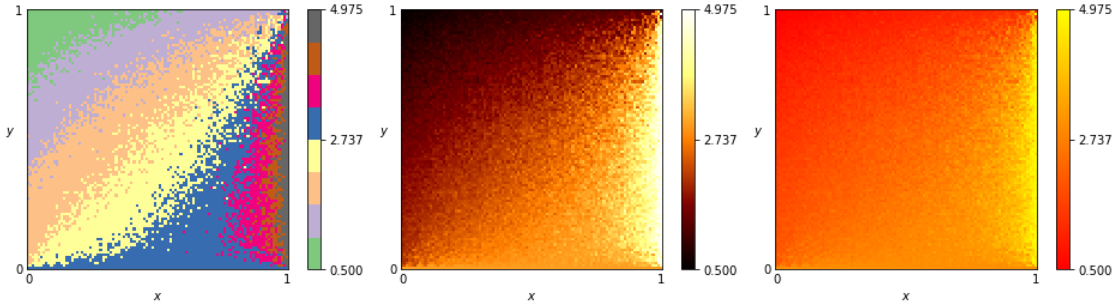
Many definitions will be now be presented, with the overall aim being to have a rigorous definition for a fingerprint. A definition of Finite State Machines will be given, and an explanation of how they relate to fingerprinting. Then definitions for the Dual, Joss-Ann and Fingerprint as presented by Ashlock in [4]. It will then be shown that any strategy can be represented as a Finite State Machine, which allows the work of Ashlock to be extended to include any strategy to be used as a probe. Finally, an example of how to construct an analytical fingerprint will be shown in Section 3.6. In essence this chapter presents a detailed review of the work of [2, 3, 4, 5, 6, 7].

In this chapter several fingerprints will be examined in detail. In Section 5.1 a breakdown of the fingerprint for TitForTat is given with an explanation of its appearance. Section 5.3 shows several plots for the strategy GoByMajority, with a varying parameter. As the parameter changes, a continuous deformation of the plot of the fingerprint can be observed. Finally, in Section 5.5 we compare the underlying data for all fingerprints of strategies within Axelrod-Python. This data is used to produce a heat plot of strategies similarity.

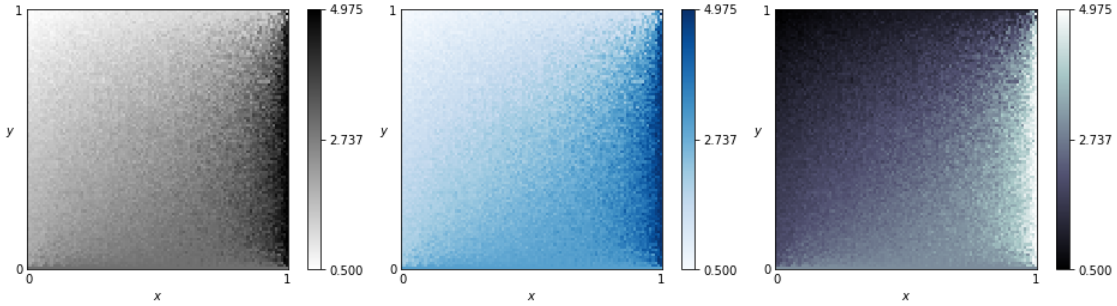
Appendix A

Appendix

A.1 Colour Maps



(a) Accent (b) afmhot (c) autumn



(d) binary (e) Blues (f) bone

A.2 Fingerprinting Documentation

Fingerprinting

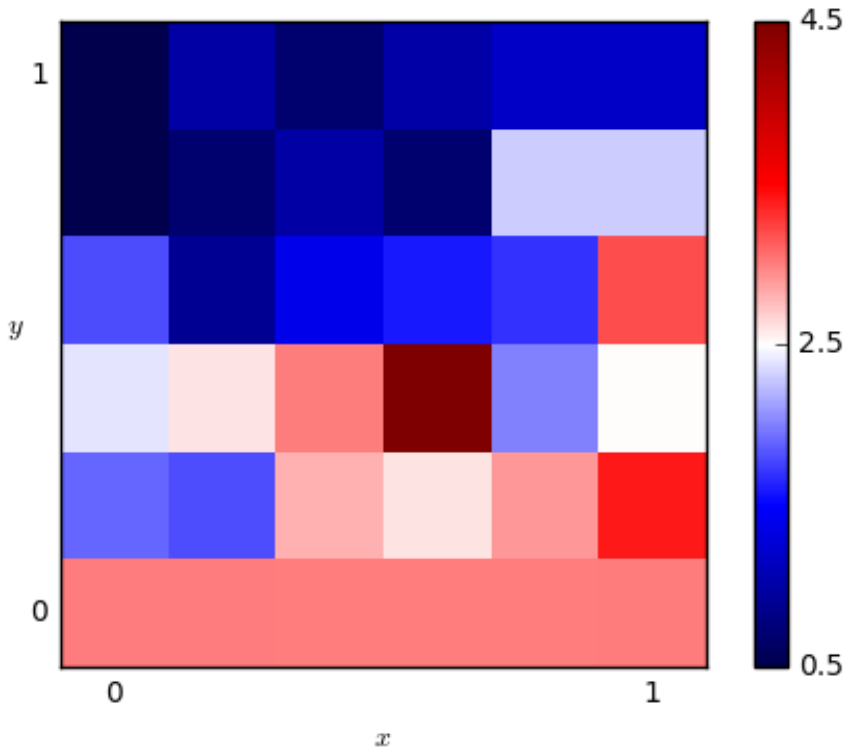
In *[Ashlock2008]*, *[Ashlock2009]* a methodology for obtaining visual representation of a strategy's behaviour is described. The basic method is to play the strategy against a probe strategy with varying noise parameters. These noise parameters are implemented through the `JossAnnTransformer`. The Joss-Ann of a strategy is a new strategy which has a probability `x` of cooperating, a probability `y` of defecting, and otherwise uses the response appropriate to the original strategy. We can then plot the expected score of the strategy against `x` and `y` and obtain a heat plot over the unit square. When `x + y >= 1` the `JossAnn` is created with parameters `(1-y, 1-x)` and plays against the Dual of the probe instead. A full definition and explanation is given in *[Ashlock2008]*, *[Ashlock2009]*.

Here is how to create a fingerprint of `WinStayLoseShift` using `TitForTat` as a probe:

```
>>> import axelrod as axl
>>> axl.seed(0)  # Fingerprinting is a random process
>>> strategy = axl.WinStayLoseShift
>>> probe = axl.TitForTat
>>> af = axl.AshlockFingerprint(strategy, probe)
>>> data = af.fingerprint(turns=10, repetitions=2, step=0.2)
>>> data
{...
>>> data[(0, 0)]
3.0
```

The `fingerprint` method returns a dictionary mapping coordinates of the form `(x, y)` to the mean score for the corresponding interactions. We can then plot the above to get:

```
>>> p = af.plot()
>>> p.show()
```

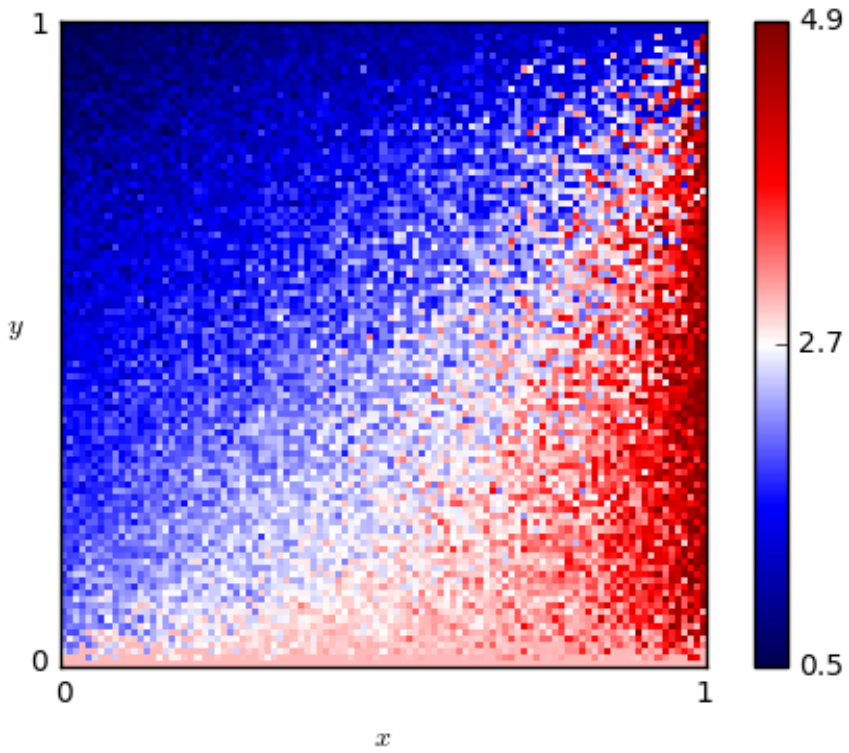


In reality we would need much more detail to make this plot useful.

Running the above with the following parameters:

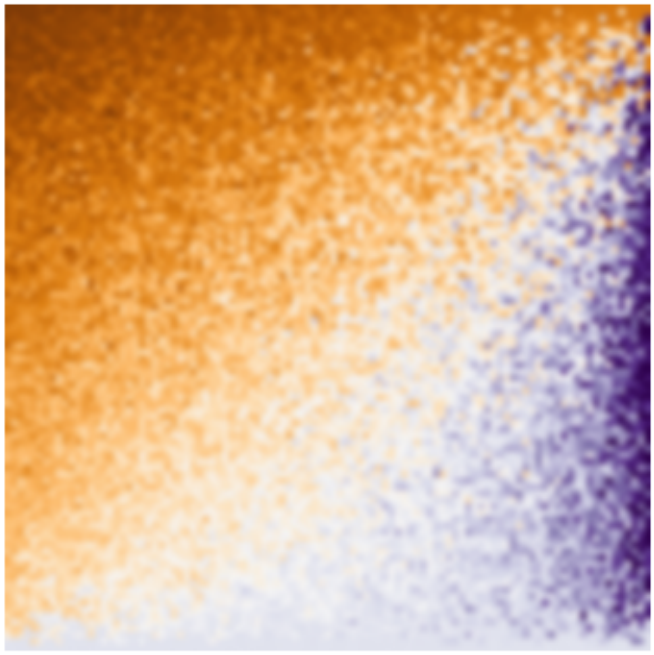
```
>>> af.fingerprint(turns=50, repetitions=2, step=0.01)
```

We get the plot:



We are also able to specify a matplotlib colour map, interpolation and can remove the colorbar and axis labels:

```
>>> p = af.plot(col_map='PuOr', interpolation='bicubic', colorbar=False, labels=False)
>>> p.show()
```



Note that it is also possible to pass a player instance to be fingerprinted and/or as a probe. This allows for the fingerprinting of parametrized strategies:

```
>>> axl.seed(0)
>>> player = axl.Random(.1)
>>> probe = axl.GTFT(.9)
>>> af = axl.AshlockFingerprint(player, probe)
>>> data = af.fingerprint(turns=10, repetitions=2, step=0.2)
>>> data
{...
>>> data[(0, 0)]
4.4...
```

Ashlock's fingerprint is currently the only fingerprint implemented in the library.

Further capabilities in the library

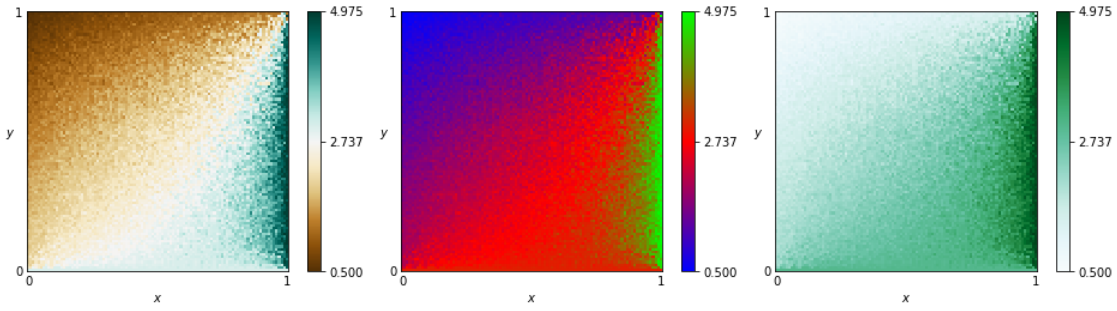
This section shows some of the more intricate capabilities of the library.

Contents:

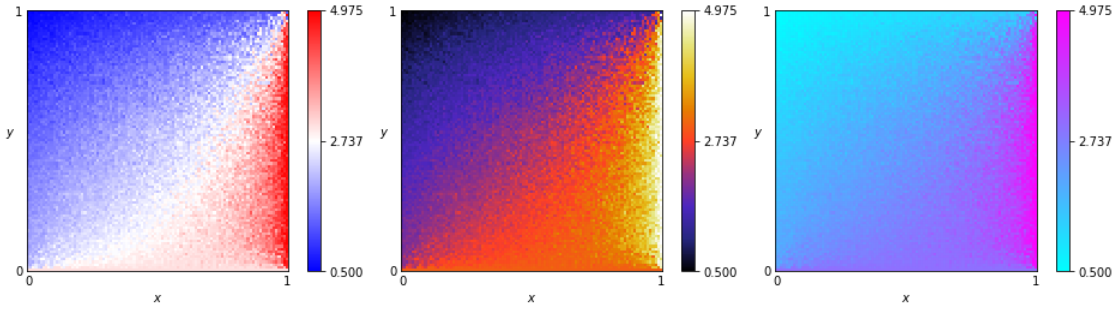
Accessing strategies

All of the strategies are accessible from the main name space of the library. For example:

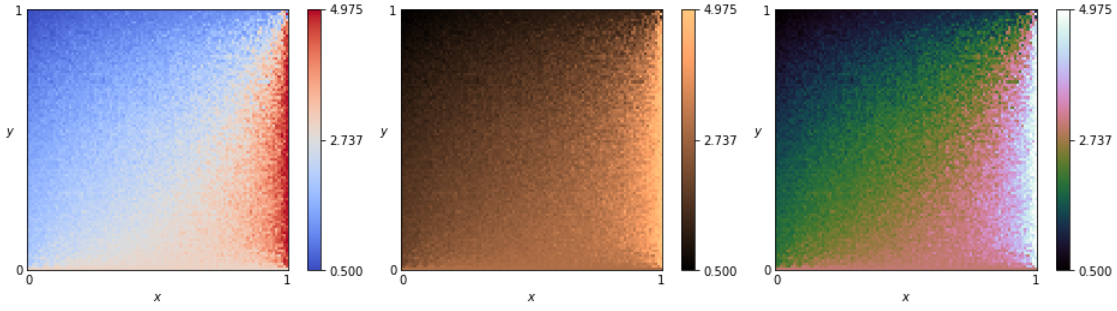
```
>>> import axelrod as axl
>>> axl.TitForTat()
Tit For Tat
```



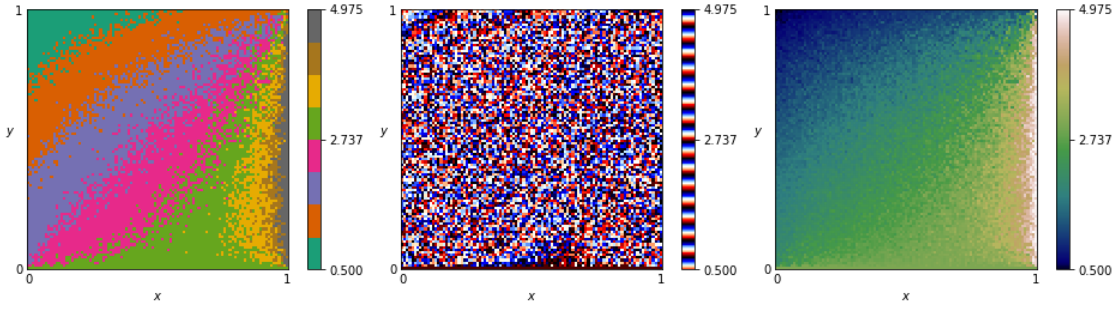
(g) BrBG (h) brg (i) BuGn



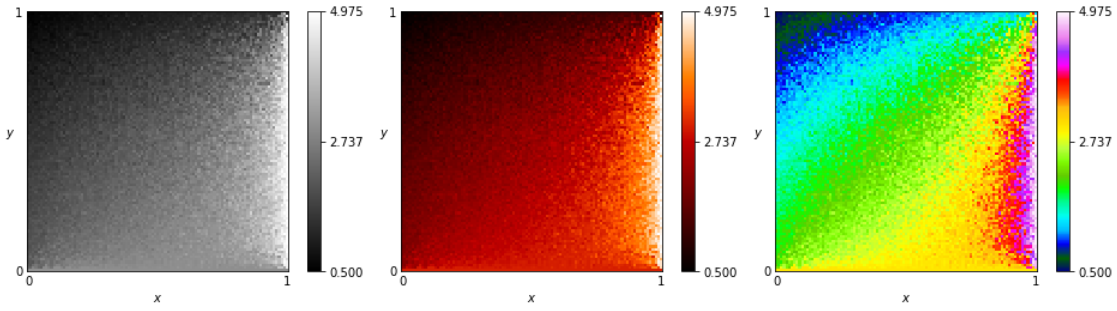
(j) bwr (k) CMRmap (l) cool



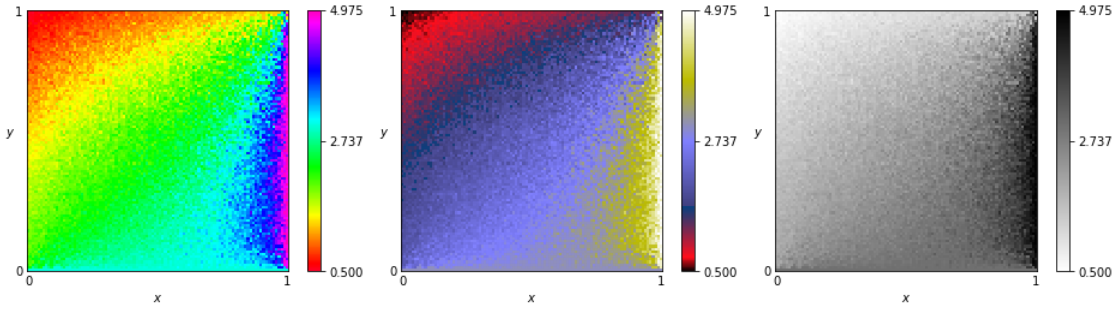
(m) coolwarm (n) copper (o) cubehelix



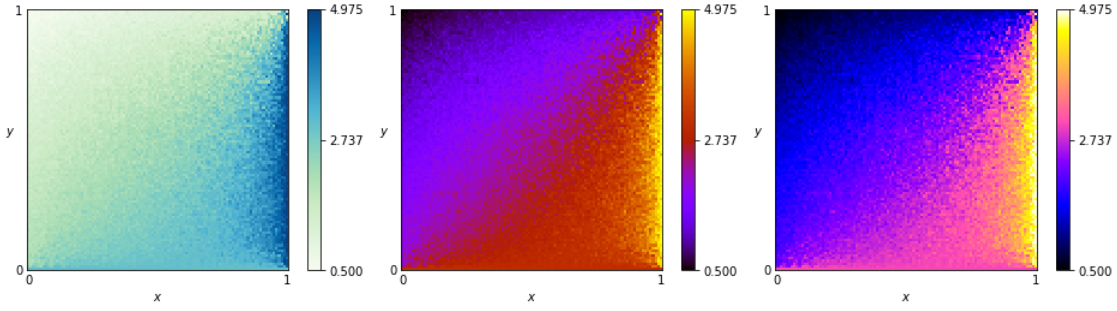
(p) Dark2 (q) flag (r) gist_earth



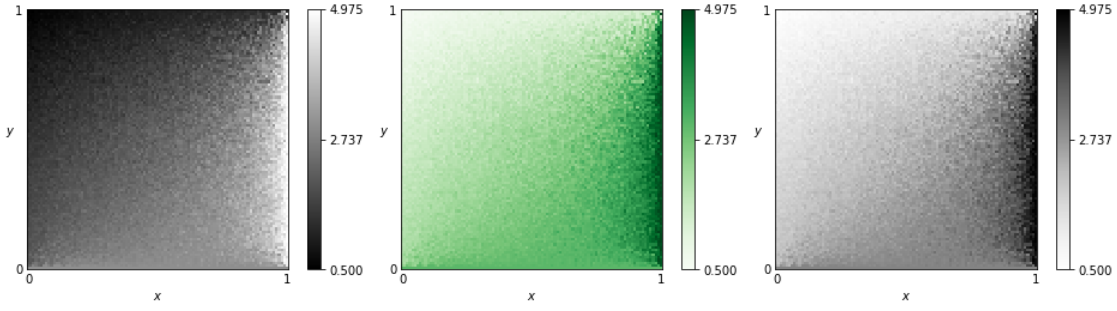
(s) gist_gray (t) gist_heat (u) gist_ncar



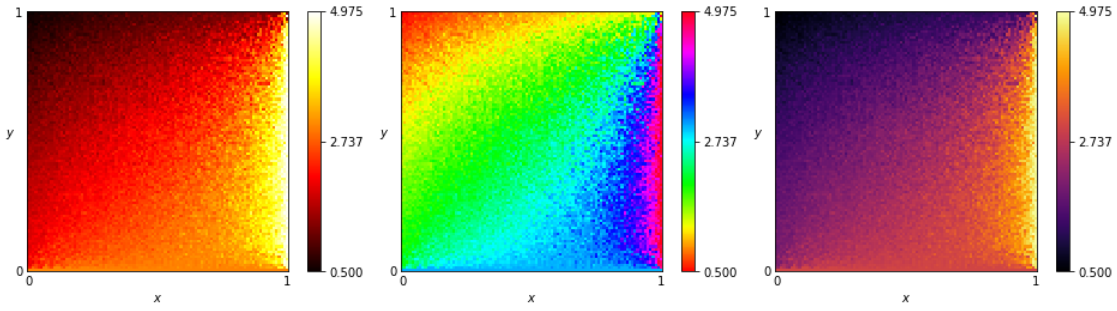
(v) gist_rainbow (w) gist_stern (x) gist_yarg



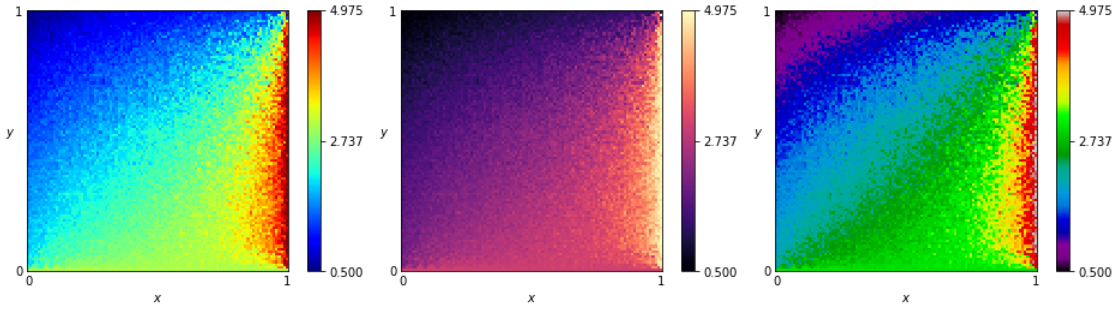
(y) GnBu (z) gnuplot (aa) gnuplot2



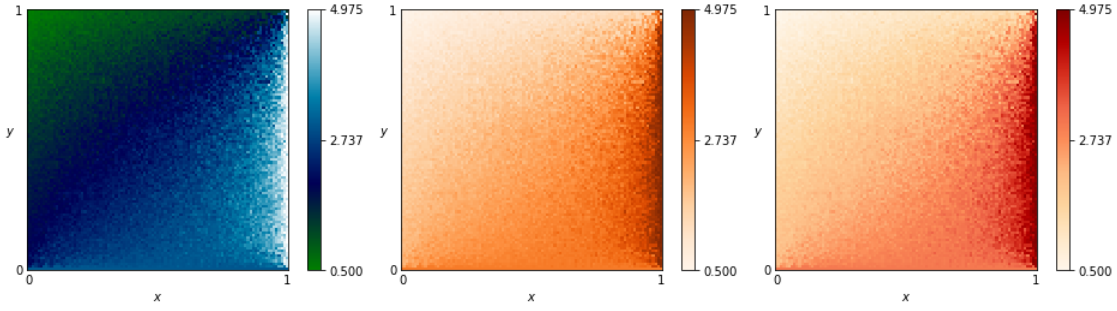
(ab) gray (ac) Greens (ad) Greys



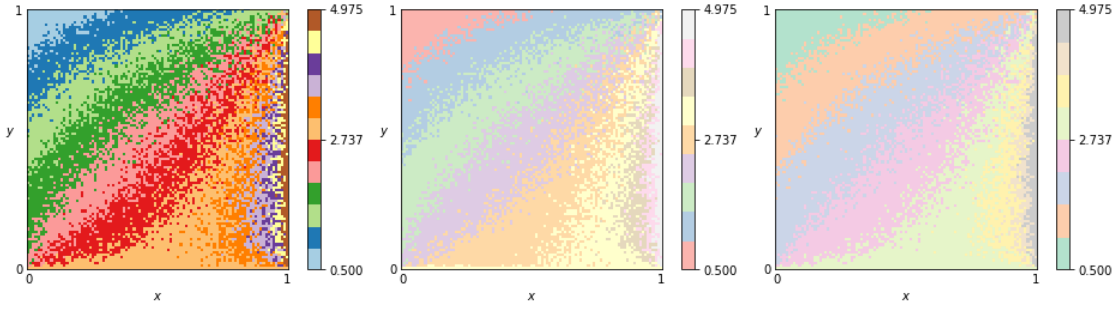
(ae) hot (af) hsv (ag) inferno



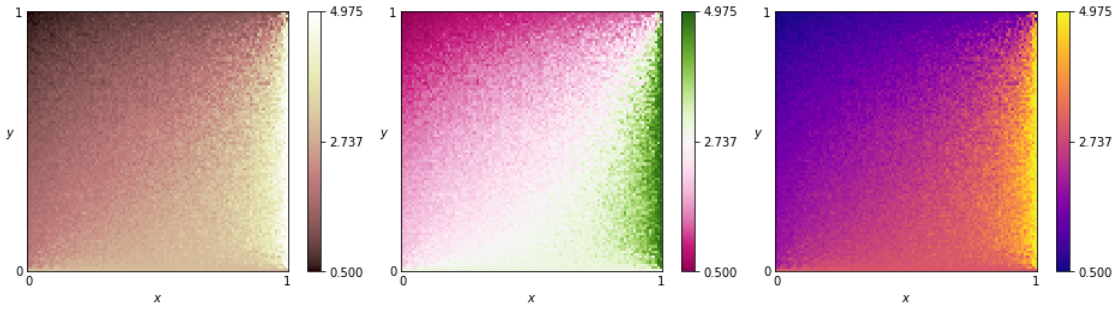
(ah) jet (ai) magma (aj) nipy_spectral



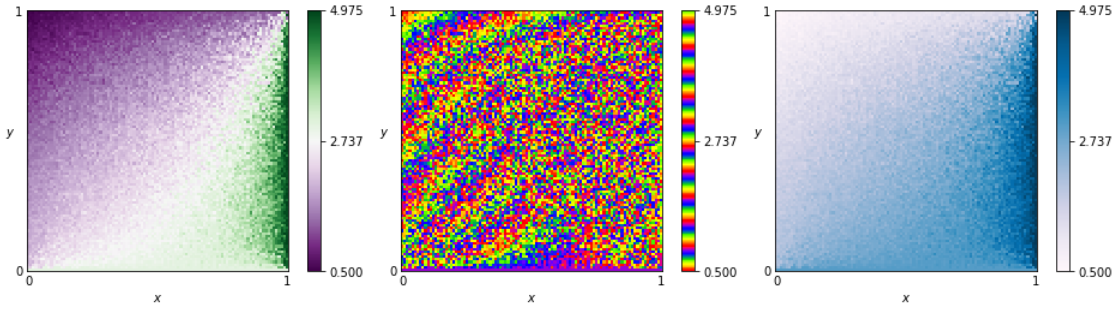
(ak) ocean (al) Oranges (am) OrRd



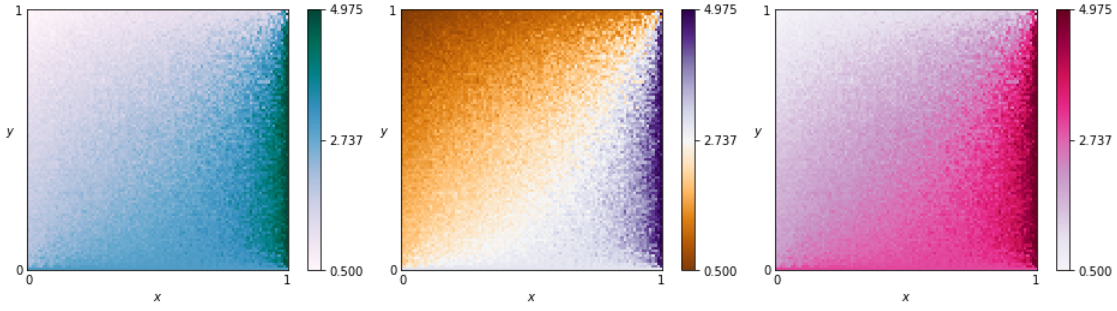
(an) Paired (ao) Pastel1 (ap) Pastel2



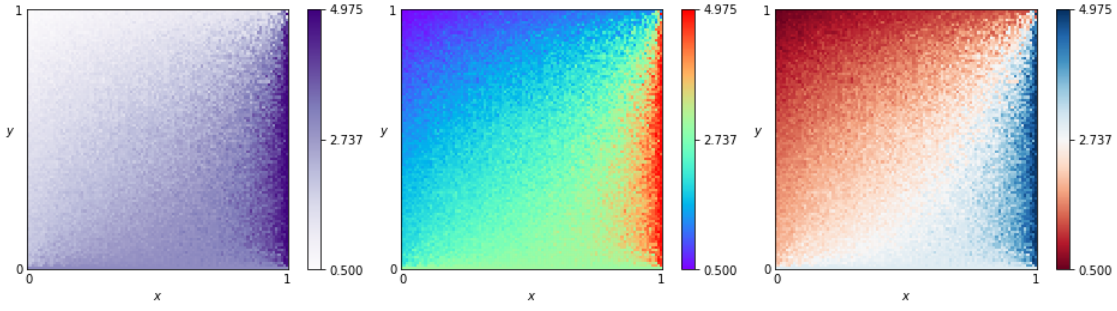
(aq) pink (ar) PiYG (as) plasma



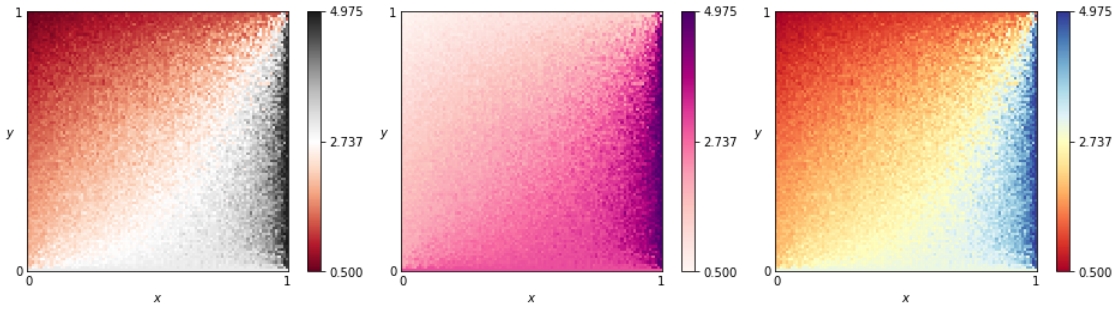
(at) PRGn (au) prism (av) PuBu



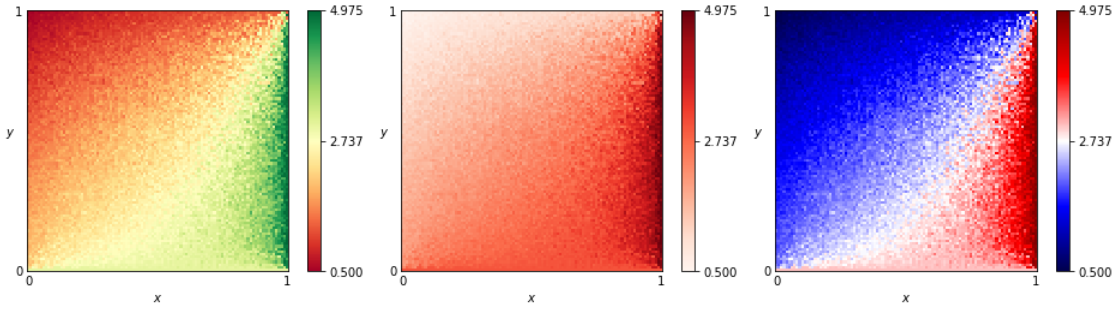
(aw) PuBuGn (ax) PuOr (ay) PuRd



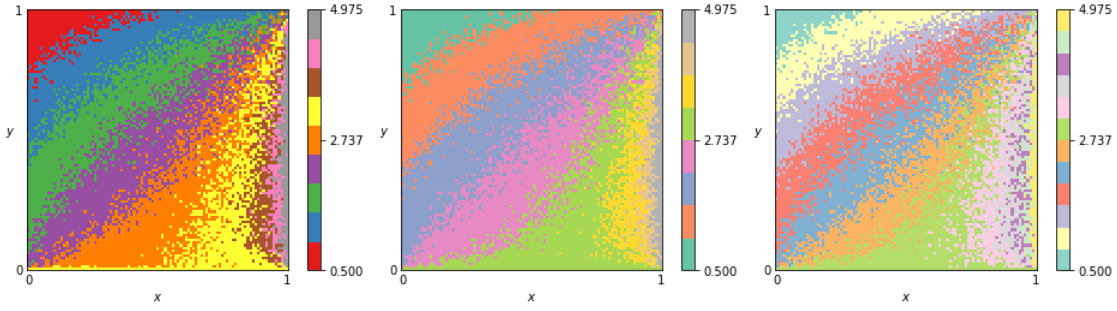
(az) Purples (ba) rainbow (bb) RdBu



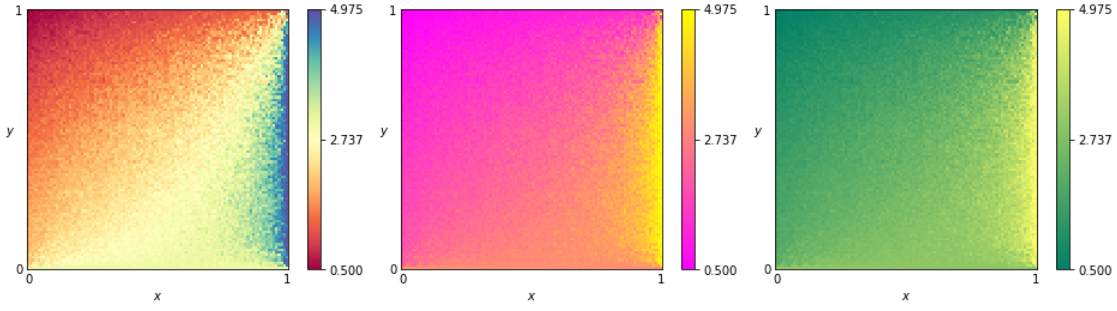
(bc) RdGy (bd) RdPu (be) RdYlBu



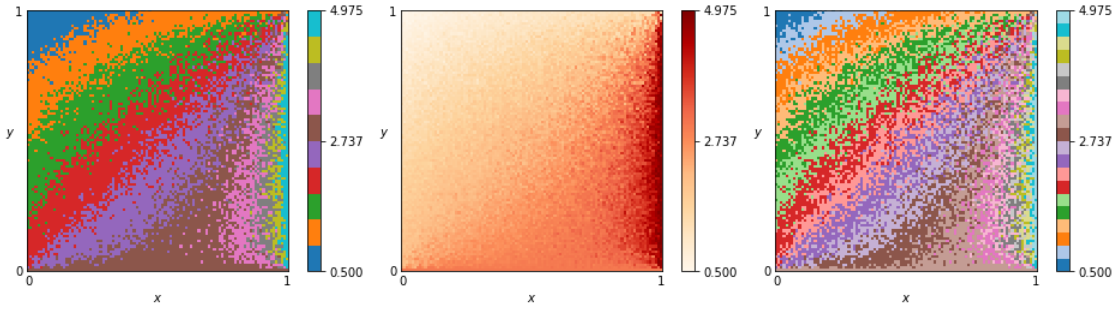
(bf) RdYlGn (bg) Reds (bh) seismic



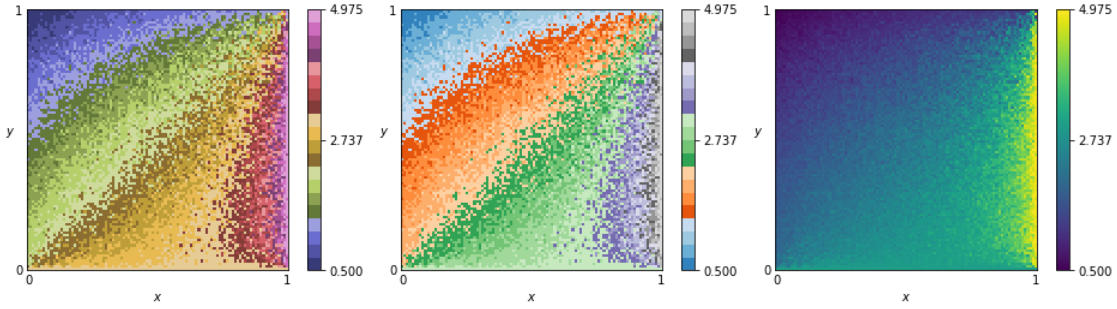
(bi) Set1 (bj) Set2 (bk) Set3



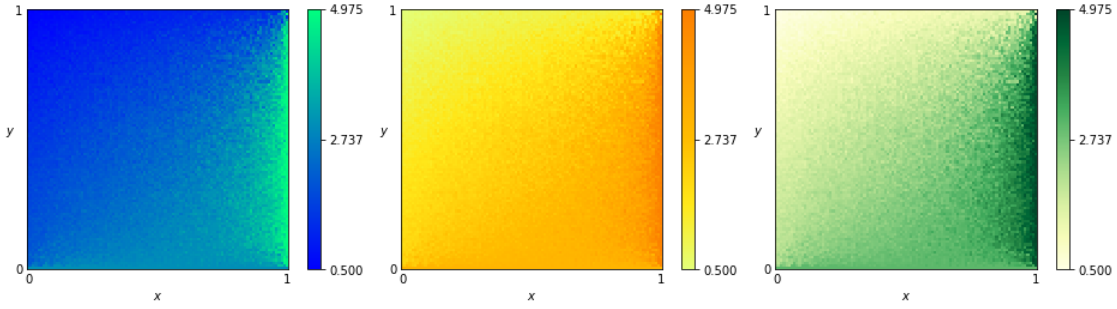
(bl) Spectral (bm) spring (bn) summer



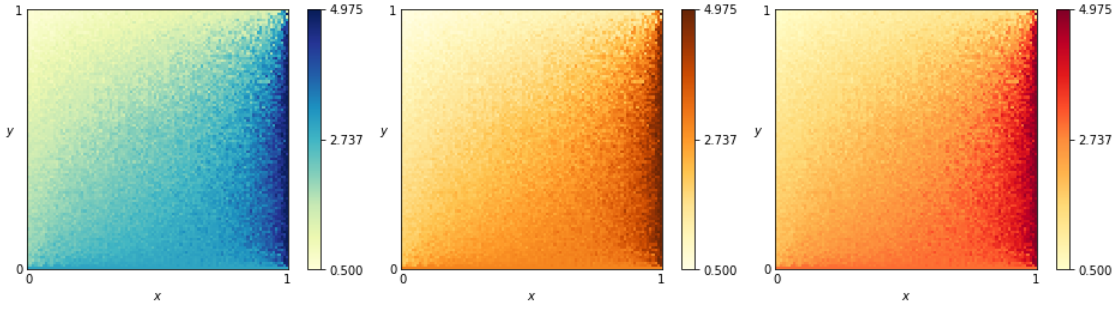
(bo) terrain (bp) Vega10 (bq) Vega20



(br) Vega20b (bs) Vega20c (bt) viridis



(bu) winter (bv) Wistia (bw) YlGn



(bx) YlGnBu (by) YlOrBr (bz) YlOrRd

Figure A.0: All possible colour maps from Matplotlib

A.3 Jupyter Notebooks

Notebook for training and assessing model

- various samples for Liner Regression and SVC
- confusion matrix
- Network graphs

In [ ]:

```
import axelrod as axl
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
import networkx as nx
from MachineLearning import *
import plotly
import plotly.plotly as py
from plotly.graph_objs import *
```

In [ ]:

```
axelrod_strategies = axl.strategies
training_df = pd.read_csv('large_training_data.csv', index_col=0)
compute_sample_scores(3, 3, training_df, axelrod_strategies)
```

...

In [ ]:

```
# this takes a long time
# sample_scores = [compute_sample_scores(i, 50, training_df, axelrod_strategies) fo
```

In [ ]:

```
zipped_scores = [list(zip(*k)) for k in sample_scores[2:]]
LR_sample_scores = [[0], [0]] + [i[0] for i in zipped_scores]
SVC_sample_scores = [[0], [0]] + [i[1] for i in zipped_scores]
```

In [ ]:

```
plt.figure(figsize=(30, 10))
lr_ps = [i for i in range(50)]
sns.set_style("darkgrid", {'axes.grid' : True})
lr_plt = plt.violinplot(LR_sample_scores, positions=lr_ps, widths=0.9)
plt.tick_params(axis='both', which='major', labelsize=20)
# plt.title('Violin Plot of Model Score against Number of Strategies used in Sample
plt.xlabel('Number of Strategies used in Sample', fontsize=20)
plt.ylabel('Model Score', fontsize=20)
plt.savefig('/Users/James/Projects/FinalYearReport-Manuscript/img/ML/score_against_s
plt.show()
plt.clf()
```

In [ ]:

```
plt.figure(figsize=(30, 10))
svc_ps = [i for i in range(50)]
sns.set_style("darkgrid", {'axes.grid' : True})
svc_plt = plt.violinplot(SVC_sample_scores, positions=svc_ps, widths=0.9)
plt.tick_params(axis='both', which='major', labelsize=20)
# plt.title('Violin Plot of Model Score against Number of Strategies used in Sample
plt.xlabel('Number of Strategies used in Sample', fontsize=20)
plt.ylabel('Model Score', fontsize=20)
plt.savefig('/Users/James/Projects/FinalYearReport-Manuscript/img/ML/score_against_s
plt.show()
plt.clf()
```

In [ ]:

```
# create dataframe with no dubplicates, only one row for each strategy pair combina
unique_scoring_df = training_df.groupby(['Name_A', 'Name_B']).first()
unique_scoring_df.head(10)
```

In [ ]:

```
# create models using specified strategies to see how the confusion matrices compar
results_df = pd.read_csv('std_summary.csv')
ordered_strats = results_df.Name.values
model_strats = ordered_strats[::5] # every 5th strategy when order by rank from rou
model_train_df, model_score_df = split_dataframe(model_strats, training_df)
lr_model, svc_model = create_models_for_sample(model_train_df)
```

In [ ]:

```
scoring_equivalent = unique_scoring_df['Equivalent']
scoring_data = unique_scoring_df.copy()
scoring_data.drop('Equivalent', axis=1, inplace=True)
svc_predictions = svc_model.predict(scoring_data)
svc_c_matrix = confusion_matrix(scoring_equivalent, svc_predictions)
np.set_printoptions(precision=2)
class_names = ['Different', 'Same']
sns.set_style("whitegrid", {'axes.grid' : False})
plt.figure()
plot_confusion_matrix(svc_c_matrix, classes=class_names, title='')
plt.savefig('/Users/James/Projects/FinalYearReport-Manuscript/img/ML/confusion_matr
plt.show()
plt.clf()
print(svc_model.score(X=scoring_data, y=scoring_equivalent))
```

In [ ]:

```
from sklearn.feature_selection import chi2
scores, pvalues = chi2(X=scoring_data, y=scoring_equivalent)
p_vals = list(zip(scoring_data.columns, pvalues))
pval_df = pd.DataFrame(data=p_vals, columns=['Variable', 'p Value'])
p = pval_df.to_latex(index=False)
with open("/Users/James/Projects/FinalYearReport-Manuscript/img/p-values.tex", "w")
    text_file.write(p)
```

In [ ]:

```
# collect rows where the model thinks the strategies are the same
actual_model_predictions = svc_predictions
equivalent_df = unique_scoring_df.copy().reset_index()
equivalent_df['Predictions'] = actual_model_predictions
equivalent_df = equivalent_df[equivalent_df['Predictions'] == 1]
equivalent_names = equivalent_df[['Name_A', 'Name_B']]
equivalent_names.head()
```

In [ ]:

```
# create adjacency matrix for strategies that are equivalent
adj_df = pd.crosstab(equivalent_names.Name_A, equivalent_names.Name_B)
idx = adj_df.columns.union(adj_df.index)
adj_df = adj_df.reindex(index = idx, columns=idx, fill_value=0)
adj_df.head()
```

In [ ]:

```
plt.figure(figsize=(50, 50))
sns.heatmap(adj_df, cbar=False)
plt.savefig('/Users/James/Projects/FinalYearReport-Manuscript/img/ML/similarity_hea
plt.show()
```

In [ ]:

```
G=nx.from_pandas_dataframe(equivalent_names, 'Name_A', 'Name_B', True)
UG = G.to_undirected()
sub_graphs = nx.connected_component_subgraphs(UG)
```

In [ ]:

```
plotly.offline.init_notebook_mode()
py.sign_in('theref', '5zi1ecSf80pI5u7I5rP0')
```

In [ ]:

```
def scatter_nodes(pos, labels=None, color=None, size=20, opacity=0.5):
    # pos is the dict of node positions
    # labels is a list  of labels of len(pos), to be displayed when hovering the mo
    # color is the color for nodes. When it is set as None the Plotly default color
    # size is the size of the dots representing the nodes
    #opacity is a value between [0,1] defining the node color opacity
    trace = Scatter(x=[], y=[],  mode='markers', marker=Marker(size=[]))
    for k, v in pos.items():
        trace['x'].append(pos[k][0])
        trace['y'].append(pos[k][1])
    attrib=dict(name='', text=labels , hoverinfo='text', opacity=opacity) # a dict
    trace=dict(trace, **attrib)# concatenate the dict trace and attrib
    trace['marker']['size']=size
    return trace


def scatter_edges(G, pos, line_color=None, line_width=1):
    trace = Scatter(x=[], y=[], mode='lines')
    for edge in G.edges():
        trace['x'] += [pos[edge[0]][0],pos[edge[1]][0], None]
        trace['y'] += [pos[edge[0]][1],pos[edge[1]][1], None]
        trace['hoverinfo']='none'
        trace['line']['width']=line_width
        if line_color is not None: # when it is None a default Plotly color is used
            trace['line']['color']=line_color
    return trace

def make_annotations(pos, font_size=14, font_color='rgb(25,25,25)'):
    annotations = Annotations()
    for k, v in pos.items():
        annotations.append(
            Annotation(
                text=str(k),
                x=pos[k][0], y=pos[k][1],
                xref='x1', yref='y1',
                font=dict(color= font_color, size=font_size),
                showarrow=False)
        )
    return annotations

G=nx.from_pandas_dataframe(equivalent_names, 'Name_A', 'Name_B', True)
sixth = len(G.nodes())//6
inner, middle, outer = G.nodes()[:sixth], G.nodes()[sixth:3*sixth], G.nodes()[3*six
pos=nx.shell_layout(G, nlist=[inner, middle, outer], scale=10)

labels=[str(k) for k in range(len(pos))] # labels are  set as being the nodes indic
trace1=scatter_edges(G, pos)
trace2=scatter_nodes(pos, labels=labels)

width=2000
height=2500
axis=dict(showline=False, # hide axis line, grid, ticklabels and  title
          zeroline=False,
          showgrid=False,
          showticklabels=False,
          title=''
          )
layout=Layout(
    font= Font(),
```

```
    showlegend=False,

    autosize=False,
    width=width,
    height=height,
    xaxis=XAxis(axis),
    yaxis=YAxis(axis),
    margin=Margin(
        l=40,
        r=40,
        b=85,
        t=100,
        pad=0,

    ),
    hovermode='closest',
#     plot_bgcolor='#EFECEA', #set background color
    )


data=Data([trace1, trace2])

fig = Figure(data=data, layout=layout)
fig['layout'].update(annotations=make_annotations(pos))

py.image.ishow(fig)
# py.image.save_as(fig, filename='/Users/James/Projects/FinalYearReport-Manuscript/
```

In [ ]:

```
 G.to_undirected()
graphs = nx.connected_component_subgraphs(UG)

index, sg in enumerate(sub_graphs):
if nx.number_of_nodes(sg) < 2:
    continue # ignore this graph and move onto the next one
pos=nx.spring_layout(sg)
labels=[str(k) for k in range(len(pos))] # labels are  set as being the nodes indice
trace1=scatter_edges(sg, pos)
trace2=scatter_nodes(pos, labels=labels)
width=500
height=500
data=Data([trace1, trace2])
layout=Layout(
    font= Font(),
    showlegend=False,
    autosize=False,
    width=width,
    height=height,
    xaxis=XAxis(axis),
    yaxis=YAxis(axis),
    margin=Margin(
        l=40,
        r=40,
        b=85,
        t=100,
        pad=0,

    ),
hovermode='closest',
  plot_bgcolor='#EFECEA', #set background color
)
fig = Figure(data=data, layout=layout)
fig['layout'].update(annotations=make_annotations(pos))

py.image.ishow(fig)
py.image.save_as(fig, filename='/Users/James/Projects/FinalYearReport-Manuscript/img
```

Notebook for creating several Plots

- Sum of Squares plot
- Many numerical fingerprints
- Example tournament and results

In [ ]:

```
import string
import numpy as np
import axelrod as axl
import pandas as pd
from tqdm import tqdm
import matplotlib.pyplot as plt
import matplotlib
```

In [ ]:

```
def format_filename(s):
    """
    Take a string and return a valid filename constructed from the string.
    Uses a whitelist approach: any characters not present in valid_chars are
    removed. Also spaces are replaced with underscores.
    Note: this method may produce invalid filenames such as ``, `.` or `..`
    When I use this method I prepend a date string like '2009_01_15_19_46_32_'
    and append a file extension like '.txt', so I avoid the potential of using
    an invalid filename.
    Borrowed from https://gist.github.com/seanh/93666
    """
    valid_chars = "-_.() {}{}".format(string.ascii_letters, string.digits)
    filename = ''.join(c for c in s if c in valid_chars)
    filename = filename.replace(' ','_')
    return filename
```

In [ ]:

```
strats = axl.strategies

strategies = [s.name for s in strats]
# need access to the fingerprint csv files from Axelrod-fingerprint repo
path = '/Users/James/Projects/Axelrod-fingerprint/assets/'
filenames = [path + format_filename(s) + '.csv' for s in strategies]
dataframes = [pd.read_csv(f) for f in filenames]


def score_for_df(A, B):
    """
    Compute the sum of squares score for two dataframes, A and B
    """
    result = pd.merge(A, B, on=['x', 'y'], suffixes=('_A', '_B'))
    result['SQ_difference'] = (result['score_A'] - result['score_B'])**2
    result.drop(['score_A', 'score_B'], axis=1, inplace=True)
    return sum(result.SQ_difference)


sum_squares_df = pd.DataFrame(index=strategies, columns=strategies)

for indexA, strategyA in enumerate(tqdm(strategies)):
    A_df = dataframes[indexA]
    for indexB, strategyB in enumerate(strategies):
        B_df = dataframes[indexB]
        similarity_score = score_for_df(A_df, B_df)
        sum_squares_df.set_value(strategyA, strategyB, similarity_score)

sum_squares_df
```

In [ ]:

```
sum_squares_df = sum_squares_df.apply(pd.to_numeric)
size = len(dataframes[0].index)
mean_squares_df = sum_squares_df.divide(size)
mean_squares_df.head()
```

In [ ]:

```
plt.figure(figsize=(50, 75))
sns.heatmap(mean_squares_df, cbar_kws={"orientation": "horizontal"})
cbar = ax.figure.colorbar(ax.collections[0])
cbar.set_ticks([0, 1])
cbar.set_ticklabels(["0%", "100%"])
# cbar.set_ticks([0, .2, .75, 1])
# cbar.set_ticklabels(['low', '20%', '75%', '100%'])
# fig.colorbar(ax, orientation="horizontal",fraction=0.07,anchor=(1.0,0.0))
plt.savefig('/Users/James/Projects/FinalYearReport-Manuscript/img/mean_squares.png'
plt.show()

# f, ax = plt.subplots(figsize=(50, 50))
# sns.heatmap(sum_squares_df, cbar=False)
```

In [ ]:

```

```

In [ ]:

```
axl.seed(0)  # Set a seed
players = [axl.TitForTat(), axl.Cooperator(), axl.Random(), axl.Gradual()]  # Creat
tournament = axl.Tournament(players)  # Create a tournament
results = tournament.play()  # Play the tournament
plot = axl.Plot(results)
p = plot.boxplot()
q = plot.payoff()
```

In [ ]:

```
p.savefig('/Users/James/Projects/FinalYearReport-Manuscript/img/examples/small_viol
p
```

In [ ]:

```
q.savefig('/Users/James/Projects/FinalYearReport-Manuscript/img/examples/small_payo
q
```

Analytical Plots

In [ ]:

```
def TFT(coord):
    x, y = coord
    numerator = y**2 + 5*x*y + 3*x**2
    denominator = (x + y)**2
    return numerator/denominator

def WSLS(coord):
    x, y = coord
    numerator = (3*x + y)*(x - 1) + 5*y*(y - 1)
    denominator = (x + 2*y)*(x - 1) + y*(y - 1)
    return numerator/denominator

def Psycho(coord):
    x, y = coord
    numerator = 4*(y - 1)*(x - 1) + 5*(y - 1)**2
    denominator = 2*(y - 1)*(x - 1) + (x - 1)**2 + (y - 1)**2
    return numerator/denominator

def Coop(coord):
    x, y = coord
    return 3 - 3*y

def Defect(coord):
    x, y = coord
    return 4*x + 1
```

In [ ]:

```
from collections import namedtuple

Point = namedtuple('Point', 'x y')

def reshape_data(data, points, size):
    """Shape the data so that it can be plotted easily.
    Parameters
    ----------
    data : dictionary
        A dictionary where the keys are Points of the form (x, y) and
        the values are the mean score for the corresponding interactions.
    points : list
        of Point objects with coordinates (x, y).
    size : int
        The number of Points in every row/column.
    Returns
    ----------
    plotting_data : list
        2-D numpy array of the scores, correctly shaped to ensure that the
        score corresponding to Point (0, 0) is in the left hand corner ie.
        the standard origin.
    """
    ordered_data = [data[point] for point in points]
    shaped_data = np.reshape(ordered_data, (size, size), order='F')
    plotting_data = np.flipud(shaped_data)
    return plotting_data


def create_points(step, progress_bar=False):
    """Creates a set of Points over the unit square.
    A Point has coordinates (x, y). This function constructs points that are
    separated by a step equal to `step`. The points are over the unit
    square which implies that the number created will be (1/`step` + 1)^2.
    Parameters
    ----------
    step : float
        The separation between each Point. Smaller steps will produce more
        Points with coordinates that will be closer together.
    progress_bar : bool
        Whether or not to create a progress bar which will be updated
    Returns
    ----------
    points : list
        of Point objects with coordinates (x, y)
    """
    num = int((1 / step) // 1) + 1

    if progress_bar:
        p_bar = tqdm(total=num ** 2, desc="Generating points")

    points = []
    for x in np.linspace(0, 1, num):
        for y in np.linspace(0, 1, num):
            points.append(Point(x, y))

            if progress_bar:
                p_bar.update()

    if progress_bar:
```

```
        p_bar.close()

    return points


def plot(plotting_data, col_map='seismic', interpolation='none', title=None,
         colorbar=True, labels=True):
    """Plot the results of the spatial tournament.
    Parameters
    ----------
    col_map : str, optional
        A matplotlib colour map, full list can be found at
        http://matplotlib.org/examples/color/colormaps_reference.html
    interpolation : str, optional
        A matplotlib interpolation, full list can be found at
        http://matplotlib.org/examples/images_contours_and_fields/interpolation_meth
    title : str, optional
        A title for the plot
    colorbar : bool, optional
        Choose whether the colorbar should be included or not
    labels : bool, optional
        Choose whether the axis labels and ticks should be included
    Returns
    ----------
    figure : matplotlib figure
        A heat plot of the results of the spatial tournament
    """
    fig, ax = plt.subplots()
    cax = ax.imshow(
        plotting_data, cmap=col_map, interpolation=interpolation)

    if colorbar:
        max_score = np.nanmax(plotting_data)
        min_score = np.nanmin(plotting_data)
        ticks = [min_score, (max_score + min_score) / 2, max_score]
        fig.colorbar(cax, ticks=ticks)

    plt.xlabel('$x$')
    plt.ylabel('$y$', rotation=0)
    ax.tick_params(axis='both', which='both', length=0)
    plt.xticks([0, len(plotting_data) - 1], ['0', '1'])
    plt.yticks([0, len(plotting_data) - 1], ['1', '0'])

    if not labels:
        plt.axis('off')

    if title is not None:
        plt.title(title)
    return fig
```

In [ ]:

```
step=0.01
size = int((1 / step) // 1) + 1
points = create_points(step)
```

In [ ]:

```
TFT_Data = {p: TFT(p) for p in points}
WSLS_Data = {p: WSLS(p) for p in points}
Psycho_Data = {p: Psycho(p) for p in points}
Coop_Data = {p: Coop(p) for p in points}
Defect_Data = {p: Defect(p) for p in points}
```

In [ ]:

```
TFT_Data = reshape_data(TFT_Data, points, size)
WSLS_Data = reshape_data(WSLS_Data, points, size)
Psycho_Data = reshape_data(Psycho_Data, points, size)
Coop_Data = reshape_data(Coop_Data, points, size)
Defect_Data = reshape_data(Defect_Data, points, size)
```

In [ ]:

```
np.nanmax(TFT_Data)
```

In [ ]:

```
TFT_plot = plot(TFT_Data)
WSLS_plot = plot(WSLS_Data)
Psycho_plot = plot(Psycho_Data)
Coop_plot = plot(Coop_Data)
Defect_plot = plot(Defect_Data)
```

In [ ]:

```
TFT_plot.savefig('/Users/James/Projects/FinalYearReport-Manuscript/img/Analytical/T
WSLS_plot.savefig('/Users/James/Projects/FinalYearReport-Manuscript/img/Analytical/W
Psycho_plot.savefig('/Users/James/Projects/FinalYearReport-Manuscript/img/Analytica
Coop_plot.savefig('/Users/James/Projects/FinalYearReport-Manuscript/img/Analytical/C
Defect_plot.savefig('/Users/James/Projects/FinalYearReport-Manuscript/img/Analytica
```

Notebook for determining the sensitivity of the model using Memory 1 strategies

In [ ]:

```
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from MachineLearning import *
```

In [ ]:

```
training_df = pd.read_csv('large_training_data.csv', index_col=0)
results_df = pd.read_csv('std_summary.csv')
ordered_strats = results_df.Name.values
model_strats = ordered_strats[::5] # every 5th strategy when order by rank from rou
model_train_df, model_score_df = split_dataframe(model_strats, training_df)
lr_model, svc_model = create_models_for_sample(model_train_df)
```

In [ ]:

```
results_file_path = '/Users/James/Desktop/memory1.csv'
memory1_comparison_df = pd.read_csv(results_file_path)
```

In [ ]:

```
memory1_comparison_df = memory1_comparison_df[memory1_comparison_df.Epsilon != 0.02
prediction_df = memory1_comparison_df.copy()
memory1_results_df = memory1_comparison_df.copy()
prediction_df.drop(['Name_A', 'Name_B', 'Epsilon'], axis=1, inplace=True)
predictions = svc_model.predict(X=prediction_df)
memory1_results_df = memory1_comparison_df.copy()
memory1_results_df['Prediction'] = predictions
memory1_results_df = memory1_results_df[['Name_A', 'Name_B', 'Epsilon', 'Prediction

line_results_df = memory1_results_df.groupby('Epsilon')['Prediction'].agg(['sum','c
violin_results_df = memory1_results_df.groupby(['Epsilon', 'Name_A'])['Prediction']
line_results_df['Proportion Correct'] = 1 - line_results_df['sum']/line_results_df[
violin_results_df['Proportion Correct'] = 1 - violin_results_df['sum']/violin_resul
violin_results_df.drop(['sum', 'count', 'Name_A'], axis=1, inplace=True)

violin_groups = violin_results_df.groupby('Epsilon')['Proportion Correct'].apply(lis
pos = violin_results_df['Epsilon'].unique()
proportions = violin_groups.tolist()
```

In [ ]:

```
fig, ax = plt.subplots()
line_results_df.plot(x='Epsilon', y='Proportion Correct', ax=ax)
plt.violinplot(proportions, positions=pos, widths=0.02)
plt.xlabel('$\delta$')
plt.xlim([0, 0.7])
plt.ylim([-0.1, 1.1])
plt.savefig('/Users/James/Projects/FinalYearReport-Manuscript/img/ML/proportion-cor
plt.show()
```

Bibliography

[1] *AppVeyor*. https://www.appveyor.com/ (cit. on p. 37).

[2] Daniel Ashlock and Eon Youn Kim. "Fingerprinting: Visualization and automatic analysis of prisoner's dilemma strategies". In: *IEEE Transactions on Evolutionary Computation* 12.5 (2008), pp. 647–659. DOI: 10.1109/TEVC.2008.920675 (cit. on pp. 8, 17–20, 22, 29, 31, 32, 67).

[3] Daniel Ashlock, Eun Youn Kim, and Wendy Ashlock. "A fingerprint comparison of different Prisoner's Dilemma payoff matrices". In: *Proceedings of the 2010 IEEE Conference on Computational Intelligence and Games, CIG2010* 2009 (2010), pp. 219–226. DOI: 10.1109/ITW.2010.5593352 (cit. on pp. 8, 17–20, 22, 29, 67).

[4] Daniel Ashlock, Eun Youn Kim, and Warren Kurt. "Finite Rationality and Interpersonal Complexity in Repeated Games". In: 56.2 (2004), pp. 397–410 (cit. on pp. 8, 17–20, 22, 29, 31, 67).

[5] Daniel Ashlock and Eun-youn Kim. "Techniques for Analysis of Evolved Prisoner's Dilemma Strategies with Fingerprints." In: (2005), pp. 2613–2620 (cit. on pp. 8, 17–20, 22, 29, 31, 67).

[6] Daniel Ashlock, Eun-youn Kim, and Wendy Ashlock. "Fingerprint Analysis of the Noisy Prisoner's Dilemma Using a Finite-State Representation". In: *IEEE TRANSACTIONS ON COMPUTATIONAL INTELLIGENCE AND AI IN GAMES* 1.2 (2009), pp. 154–167 (cit. on pp. 8, 17–20, 22, 29, 67).

[7] Daniel Ashlock et al. "Understanding Representational Sensitivity in the Iterated Prisoner ' s Dilemma with Fingerprints". In: *IEEE TRANSACTIONS ON SYSTEMS, MAN, AND CYBERNETICS* 36.4 (2006), pp. 464–475 (cit. on pp. 8, 17–20, 22, 29, 31, 67).

[8] Robert Axelrod. "Effective Choice in the Prisoner's Dilemma". In: *The Journal of Conflict Resolution* 65.2 (1980), pp. 217–233 (cit. on pp. 6–8, 10, 11).

[9] Robert Axelrod. "More Effective Choice in the Prisoner' s Dilemma". In: *The Journal of Conflict Resolution* 24.1 (1980), pp. 3–25. DOI: 10.1177/002200278002400101 (cit. on pp. 6, 8, 10, 11, 13).

[10] Robert Axelrod. *The Evolution of Cooperation*. 1984 (cit. on pp. 7, 11).

[11] Robert Axelrod and William D. Hamilton. "The Evolution of Cooperation". In: *Science* 211.4489 (1981), pp. 1390–1396. DOI: 10.1126/science.7466396. arXiv: t8jd4qr3m [13960]. URL: http://www.jstor.org/stable/1685895 (cit. on p. 11).

[12] Bruno Beaufils, Jean-Paul Delahaye, and Philippe Mathieu. "Our meeting with gradual, a good strategy for the iterated prisoner's dilemma". In: *Proceedings of the Fifth International Workshop on the Synthesis and Simulation of Living Systems*. 1997, pp. 202–209. ISBN: 9788578110796. DOI: 10.1017/CBO9781107415324.004. arXiv: arXiv:1011.1669v3 (cit. on p. 13).

[13] J. Bendor, R. M. Kramer, and S. Stout. "When in Doubt ... Cooperation in a Noisy Prisoner's Dilemma". In: *The Journal of Conflict Resolution* 35.4 (1991), pp. 691–719. URL: http://jcr.sagepub.com/content/35/4/691 (cit. on pp. 11, 12).

[14] Siang Y. Chong and Xin Yao. "Behavioral diversity, choices and noise in the iterated prisoner's dilemma". In: *IEEE Transactions on Evolutionary Computation* 9.6 (2005), pp. 540–551. DOI: 10.1109/TEVC.2005.856200 (cit. on pp. 10, 67).

[15] Siang Yew Chong et al. "Chapter 1 The Iterated Prisoner ' s Dilemma : 20 Years On". In: (2004), pp. 1–21 (cit. on p. 11).

[16] *Coveralls*. https://coveralls.io/ (cit. on p. 37).

[17] Tom Crick. "Share and Enjoy: Publishing Useful and Usable Scientific Models". In: (2014). arXiv: arXiv:1409.0367v1 (cit. on p. 15).

[18] The Axelrod project developers. *Axelrod: v2.6.0*. 2016. DOI: 10.5281/322624. URL: http://dx.doi.org/10.5281/zenodo.322624 (cit. on pp. 6, 13, 15).

[19] Merrill M. Flood and Melvin Dresher. *Some Experimental Games*. 1958. DOI: 10.1287/mnsc.5.1.5 (cit. on p. 7).

[20] N Franken and a P Engelbrecht. "Particle swarm optimization approaches to coevolve strategies for the iterated prisoner's dilemma". In: *Evolutionary Computation, IEEE Transactions on* 9.6 (2005), pp. 562–579. DOI: 10.1109/TEVC.2005.856202 (cit. on pp. 10, 67).

[21] Saul I. Gass and Arjang a Assad. *an Annotated Timeline of Operations Research*. 2005, p. 125. ISBN: 140208112X. DOI: 1402081138. URL: http://ebooks.kluweronline.com (cit. on p. 7).

[22] N. M. Gotts, J. G. Polhill, and A. N R Law. "Agent-based simulation in the study of social dilemmas". In: *Artificial Intelligence Review* 19.1 (2003), pp. 3–92. ISSN: 02692821. DOI: 10.1023/A:1022120928602 (cit. on p. 7).

[23] Shaun Hargreaves Heap and Yanis Varoufakis. *Game Theory: A Critical Text*. 2nd ed. 2003. ISBN: 0203199278 (cit. on p. 13).

[24] Neil P Chue Hong et al. "Top Tips to Make Your Research Irreproducible". In: (2015), pp. 5–6. arXiv: arXiv:1504.00062v2 (cit. on p. 15).

[25] J. D. Hunter. "Matplotlib: A 2D graphics environment". In: *Computing In Science & Engineering* 9.3 (2007), pp. 90–95. DOI: 10.1109/MCSE.2007.55. URL: http://matplotlib.org/examples/color/colormaps_reference.html (cit. on p. 31).

[26] *Hypothesis Documentation*. https://hypothesis.readthedocs.io/en/latest/ (cit. on p. 40).

[27] Darrel C Ince, Leslie Hatton, and John Graham-cumming. "The case for open computer programs". In: *Nature* 482.7386 (2012), pp. 485–488. ISSN: 0028-0836. DOI: 10.1038/nature10836. URL: http://dx.doi.org/10.1038/nature10836 (cit. on p. 36).

[28] Hisao Ishibuchi and Naoki Namikawa. "Evolution of iterated prisoner's dilemma game strategies in structured demes under random pairing in game playing". In: *IEEE Transactions on Evolutionary Computation* 9.6 (2005), pp. 552–561. DOI: 10.1109/TEVC.2005.856198 (cit. on pp. 10, 67).

[29] Martin Jones. *Evolving strategies for an Iterated Prisoner's Dilemma tournament*. 2015. URL: http://mojones.net/evolving-strategies-for-an-iterated-prisoners-dilemma-tournament.html (cit. on p. 14).

[30] Ehud Kalai and William Stanford. “Finite Rationality and Interpersonal Complexity in Repeated Games”. In: 679 (1986) (cit. on p. 15).

[31] Ehud Kalai and William Stanford. “Finite Rationality and Interpersonal Complexity in Repeated Games”. In: *The Econometric Society* 56.2 (1988), pp. 397–410. URL: http://www.jstor.org/stable/1911078 (cit. on p. 15).

[32] Shyamalendu Kandar. *Introduction to Automata Theory, Formal Languages and Computation*. 1st ed. 2013 (cit. on p. 14).

[33] Vincent Knight et al. “An Open Framework for the Reproducible Study of the Iterated Prisoner’s Dilemma”. In: *Journal of Open Research Software* 4.1 (2016), e35. DOI: 10.5334/jors.125 (cit. on pp. 6, 15).

[34] David Kraines and Vivian Kraines. “Pavlov and the prisoner’s dilemma”. In: *Theory and Decision* 26 (1989), pp. 47–79. DOI: 10.1007/BF00134056. URL: https://link.springer.com/article/10.1007/BF00134056 (cit. on p. 13).

[35] David R. MacIver. *Hypothesis 3.6.1*. https://github.com/HypothesisWorks/hypothesis-python. 2016 (cit. on pp. 15, 40).

[36] David Matthews, Greg Wilson, and Steve Easterbrook. “Configuration Management for Large-Scale Scientific Computing at the UK Met Office”. In: *Computing in Science and Engineering* () (cit. on p. 36).

[37] Mojones. *axelrod-evolver*. https://github.com/Axelrod-Python/axelrod-dojo. 2016 (cit. on p. 14).

[38] Kenneth Moreland. “Diverging color maps for scientific visualization”. In: 5876 LNCS.PART 2 (2009), pp. 92–103. ISSN: 03029743. DOI: 10.1007/978-3-642-10520-3_9 (cit. on p. 32).

[39] Tertulien Ndjountche. *Digital Electronics, Volume 3*. 1st ed. Wiley, 2016 (cit. on p. 14).

[40] Martin A Nowak and Robert M May. “Evolutionary games and spatial chaos”. In: *Letters to Nature* (1992) (cit. on p. 11).

[41] Martin A Nowak and Robert M May. “The Spatial Dilemmas of Evolution”. In: *International Journal of Bifurcation and Chaos* 3 (1992), pp. 35–78 (cit. on pp. 11, 13).

[42] Martin A Nowak, Robert M May, and Sebastian Bonhoeffer. “Spatial games and the maintenance of cooperation”. In: 91.May (1994), pp. 4877–4881 (cit. on p. 11).

[43] F. Pedregosa et al. “Scikit-learn: Machine Learning in Python”. In: *Journal of Machine Learning Research* 12 (2011), pp. 2825–2830 (cit. on p. 54).

[44] James B Procter and Andreas Prlic. “Ten Simple Rules for the Open Development of Scientific Software”. In: 8.12 (2012), pp. 8–10. DOI: 10.1371/journal.pcbi.1002802 (cit. on p. 15).

[45] Karthik Ram. “Git can facilitate greater reproducibility and increased transparency in science”. In: *Source Code for Biology and Medicine* (2013), pp. 1–8 (cit. on p. 36).

[46] Amnon Rapoport, Darryl A Seale, and Andrew M Colman. “Is Tit-for-Tat the Answer ? On the Conclusions Drawn from Axelrod ’ s Tournaments”. In: *PLOS one* (2015), pp. 1–11. DOI: 10.1371/journal.pone.0134128 (cit. on p. 10).

[47] *reStructuredText*. http://docutils.sourceforge.net/rst.html (cit. on p. 40).

[48] Stewart Robinson. *Simulation : The Practice of Model Development and Use*. John Wiley and Sons, 2004. ISBN: 0470847727 (cit. on p. 31).

[49] Rosen and Lawrence. *Open Source Licensing*. 2004, p. 85. ISBN: 0131487876 (cit. on p. 15).

[50] Ariel Rubinstein. "Finite Automata Play the Repeated Prisoner's Dilemma". In: *Journal of Economic Theory* 96 (1986), pp. 83–96 (cit. on p. 15).

[51] Geir Kjetil Sandve et al. "Ten Simple Rules for Reproducible Computational". In: 9.10 (2013), pp. 1–4. DOI: 10.1371/journal.pcbi.1003285 (cit. on p. 15).

[52] Gopal P Sarma et al. "Unit testing , model validation , and biological simulation". In: (2016), pp. 1–13. URL: https://peerj.com/preprints/1315.pdf (cit. on p. 37).

[53] Mark Schmidt et al. "Minimizing Finite Sums with the Stochastic Average Gradient". In: *HAL* (2016). URL: https://hal.inria.fr/hal-00860051v2 (cit. on p. 54).

[54] Matthias Schwab, Martin Karrenbach, and Jon Claerbout. "Making Scientific Computations Reproducible". In: *Scientific Computations* (2000), pp. 61–67 (cit. on p. 36).

[55] *Scikit-learn Documentation*. https://scikit-learn.org/stable/modules/feature_selection.html (cit. on p. 57).

[56] Karl Sigmund and Martin A Nowak. "Evolutionary game theory". In: *Current Biology* 9.14 (1999), pp. 503–505. DOI: 10.1007/978-1-4614-1800-9_63 (cit. on pp. 10, 67).

[57] Herbert A. Simon. *Theories of Bounded Rationality*. 1972 (cit. on p. 15).

[58] Alex J Smola and Bernhard Scholkopf. "A Tutorial on Support Vector Regression". In: *Statistics and Computing* (2004), pp. 199–222. DOI: 10.1023/B:STCO.0000035301.49549.88 (cit. on p. 54).

[59] a. J. Stewart and J. B. Plotkin. "Extortion and cooperation in the Prisoner's Dilemma". In: *Proceedings of the National Academy of Sciences* 109.26 (2012), pp. 10134–10135. ISSN: 0027-8424. DOI: 10.1073/pnas.1208087109 (cit. on p. 11).

[60] The Axelrod project developers. *Axelrod Documentation*. URL: http://axelrod.readthedocs.io/en/latest/index.html (cit. on p. 40).

[61] Linus Torvald. *Git*. https://github.com/git/git (cit. on p. 36).

[62] *Travis*. https://travis-ci.org/ (cit. on p. 37).

[63] A M Turing. "On Computable Numbers, with an Application to the Entscheidungsproblem". In: *Proceedings of the London Mathematical Society* (1936) (cit. on p. 14).

[64] J Vandewalle and J A K Suykens. "Least Squares Support Vector Machine Classifiers". In: *Neural Processing Letters* (1999), pp. 293–300. DOI: 10.1023/A:1018628609742 (cit. on p. 54).

[65] Christopher J C H Watkins. "Q-Learning". In: *Machine Learning Journal* 292 (1992), pp. 279–292. DOI: 10.1007/BF00992698 (cit. on p. 66).

[66] Laurie Williams, Gunnar Kudrjavets, and Nachiappan Nagappan. "On the Effectiveness of Unit Test Automation at Microsoft". In: (). URL: https://collaboration.csc.ncsu.edu/laurie/Papers/Unit%7B%5C_%7Dtesting%7B%5C_%7DcameraReady.pdf (cit. on p. 37).

[67] Greg Wilson et al. "Best Practices for Scientific Computing". In: (2013), pp. 1–18. arXiv: arXiv:1210.0530v4 (cit. on p. 36).

[68] Hsiang-fu Yu, Chih-jen Lin, and Fang-lan Huang. "Dual coordinate descent methods for logistic regression and maximum entropy models". In: *Machine Learning Journal* (2011), pp. 41–75. DOI: 10.1007/s10994-010-5221-8 (cit. on p. 54).