Repeated games with mistakes

Nikolas M. Skoufis
Supervisor: Julian Garcia

October 20th, 2015

Abstract

This project studies efficient methods to compute the expected payoff of an iterated prisoner's dilemma between two opponents. In order to efficiently compute these payoffs, the mathematics underpinning the iterated prisoners dilemma and the computation of expected payoff was analyzed with a view to improving in speed of computation over existing methods. A Monte-Carlo method of computing expected payoff was used as a benchmark against which two fast methods for computing payoff were compared. The results of these fast methods were then used to generate three-population simplex plots of evolutionary dynamics as a further benchmark.

1 Introduction

1.1 Prisoner's Dilemma

The prisoner's dilemma is an archetypal problem in game theory, particularly in the study of the evolution of cooperation. The prisoner's dilemma involves two prisoners, each of which has two moves available to them. The prisoner's are told of the two options available to them, and they are told that they have 24 hours to make a decision or they will be punished. Each player can either inform on the other player (known as *defection*) or stay silent (known as *cooperation*). If both prisoners defect, will both spend 10 years in prison. If both prisoners cooperate, they will each spend only 5 years in prison; a better outcome for both. However if one prisoner cooperates and the other defects, the prisoner who defected goes free, and the prisoner that cooperated will spend 20 years in prison; longer than any other option.

Given these options and outcomes, we can see that a natural dilemma arises. If a player cooperates, they may receive a better outcome than if they had defected, depending on the other player's action. However if they defect, they have the opportunity for best possible outcome (going free), but they also risk punishment if the other player defects. This problem is the prisoner's dilemma.

An important concept in game theory is that of the Nash equilibrium. The Nash equilibrium is a set of strategies where each player cannot improve their own outcome by changing their strategy while the opponent's strategy remains the same. This means that in effect, a Nash equilibrium is the optimal result for all players. It is a well known result in game theory that the only Nash equilibrium in the prisoner's dilemma is for both players to defect. However this result is only valid for the single-shot prisoner's dilemma ie. the prisoner's dilemma where the players only play the game once. If players player multiple times, more complex strategies can emerge and there emerges an incentive for mutual cooperation [5]. This problem is known as the 'iterated prisoner's dilemma'.

One issue with the iterated prisoner's dilemma is that if players know ahead of time how many rounds of the prisoner's dilemma they will play, the cooperation breaks down. Since there are no repercussions for defecting in the final round (because there are no more rounds in which the opponent can retaliate), it is always best to defect. However if it is optimal for both players to defect in the final round, then logically there are no repercussions for defecting in the second-to-last round either. By this logic, the optimal strategy is simply to always defect. In order to overcome this predictable and uninteresting pattern, we introduce the idea that the game length is determined by a repeated random event. After playing the first and subsequent rounds, the game continues for another round with probability $0 < \delta < 1$. This means that there is a always a chance that there will be another round of the game, and thus there is always the possibility that defection will carry consequences.

1.2 Expected Payoff

The different moves and their respective outcomes of the prisoner's dilemma are generally abbreviated, and we will use these abbreviations from here onwards. Cooperation and defection are usually abbreviated to C and D respectively. The payoff for mutual cooperation is abbreviated to R for reward. The payoff for mutual defection is abbreviated to P for punishment. If one player defects and the other cooperates, the defector is said to receive the T (for temptation) payoff and the cooperator receives the S (for sucker) payoff.

When playing the iterated prisoner's dilemma, many different strategies are possible. The most basic of strategies is to unconditionally perform the same action: unconditionally defect or unconditionally cooperate. These strategies are known as Always Defect (AllD) and Always Cooperate (AllC) respectively. More complicated strategies include:

- Tit For Tat (TFT): If the opponent cooperate last round, you cooperate. If the opponent defected last round, you defect. In the first round, the player cooperates.
- Win Stay, Lose Shift (WSLS): If the outcome is a win for the player (R payoff or T payoff) then the previous move is repeated. If the outcome is a lose for the player (P payoff or S payoff) then the opposite move is chosen in the next round.
- Grim: Cooperates until the opponent defects, and then unconditionally defects from then on.

It is useful to be able to compare the effectiveness of each of these strategies when paired against another strategy. For simple cases such as AllC against AllD, AllD is trivially the superior strategy. However in more complicated cases, it becomes necessary to use a metric to evaluate the performance of a strategy. The most commonly used metric is the expected payoff.

The expected payoff is defined as the expected value of the payoff of one strategy when pitted against another strategy. For deterministic strategies, this can easily be computed with an infinite sum [4]:

$$\sum_{i=0}^{\infty} \delta^i \pi_i \tag{1}$$

where δ is the continuation probability and π_i is the payoff of the strategy in round i.

This sum holds for any pair of deterministic strategies. This sum is straight-forward to compute programatically, however when we move to non-deterministic strategies it quickly becomes difficult. Any strategy that employs non-deterministic play styles (eg. defect 75% of the time, cooperate the other 25%) introduces additional options in the outcome space which must be accounted for. We must now multiply the continuation probability by the probability of that particular outcome, and we have to continue this 'branch' of the outcome tree forever. For more complex strategies, the space of outcomes quickly becomes very large, and consequently, the expected payoff becomes very difficult to compute.

1.3 Mistakes and fault tolerant strategies

Another possible complication of the outcome space is the addition of mistakes. Mistakes occur when a player intends to play a particular move, but instead they play a different move. In the context of the iterated prisoner's dilemma, this occurs when a player intends to cooperate, but instead they defect (or vice-versa). Since mistakes are probabilistic and can occur at any round, they serve to worsen the problem of exploding outcome space and associated difficulty in calculating expected payoffs. This project's primary goal was to determine a method of computing the expected payoff of an iterated prisoner's dilemma with mistakes in an efficient manner.

In the presence of mistakes, some strategies fare better than others. In particular, it is a well known result [1] that the simple TFT strategy outperforms all other strategies in the absence of mistakes. However in the presence of mistakes, the TFT strategy displays poor 'fault tolerance' because it loses the 'robustness' property. For a rigorous definition of fault tolerance and robustness, refer to [3] and [2]. Briefly, robustness is a measure of how a strategy performs against another strategy in a population consisting only of those two strategies. Informally, a robust strategy is said to be sometimes better and never worse than other strategies. A fault tolerant strategy is a strategy that is able to retain this robust quality, even in the face of mistakes.

Since TFT is not particularly fault tolerant, modern game theorists have attempted to study strategies that are able to deal with faults while still remaining competitive. And in order to evaluate these strategies, it becomes necessary to efficiently and easily calculate their expected payoffs against other strategies.

2 Software tools and techniques

In order to study the expected payoff, it was necessary to first design a software framework that allows for the construction, simulation and computation of scenarios in game theory.

The code used in this project is open sourced under the GNU GPL v3 license and is available at https://github.com/computationalevolutionarydynamics/repeatedmistakes.

Python was chosen as the language for development of this framework, due to both the author's familiarity with the language coupled with high quality libraries for numerical calculations, visualization and testing. In particular, this project made use of the Hypothesis property-based testing library [6] and the Nose testing framework [7]. I also used the Travis CI continuous integration service and the Coveralls test coverage service.

2.1 Hypothesis and Nose

The Hypothesis library is inspired by the Haskell library 'Quickcheck', and is a property-based form of testing. The chief difference is that instead of defining in terms of specific inputs and outputs, you define tests in terms of classes of input and their expected output. Hypothesis uses in intelligent search strategy to attempt to produce an input which falsifies the property. Hypothesis then reduces this input to the minimum falsifying example and returns this.

As an example, here is how we might test a `reverse` method with conventional testing and with Hypothesis:

- Conventional: when called with the input `foobar`, the method should return `raboof`.
- Hypothesis: when called with any string, the method should return the reverse of that string.

Clearly the Hypothesis version is much more powerful and expressive. For example, if the `reverse` method didn't correctly handle empty strings, then Hypothesis would catch this and return it to the user.

Hypothesis proved to be an invaluable tool in this project because it allowed me to define the general properties that should be obeyed for any strategy and any history, and then evaluate my implementations of the strategies against these properties.

One issue I ran into with using Hypothesis was running tests can be slow when compared to standard unit testing. For example, running a test to ensure that strategies always return the correct length of history over 8 different strategies can take on the order of seconds. Although this is not a huge concern, this did cause some issues with Travis as discussed below.

Another issue with Hypothesis is that occasionally it would produce an error because it was unable to produce enough test data satisfied the requirements given. This is most likely due to inexperience with Hypothesis, and could likely be remedied by optimizing the functions that construct test data.

Finally, since Hypothesis's testing is fundamentally probabilistic in nature, occasionally tests would fail even though the code that changed did not affect them. This occurs because Hypothesis uses some randomized test data to find edge cases, and sometimes that test data doesn't hit upon a falsifying example. This can be remedied by increasing the number of examples to test for each property, however there is always the possibility that a falsifying example will be skipped over.

In addition to Hypothesis, I made use of the Nose testing framework. Nose aims to be a replacement for Python's default `unittest` library, with an emphasis on simpler and more readable test code. A key reason behind my use of Nose is that Nose allows for what it calls 'test generators'. Test generators allow for generation of multiple tests over a variety of input data. This feature was also useful when evaluating the correctness of the strategies that were implemented.

2.2 Travis and Coveralls

In order to easily keep track of the status of the unit tests, I used the Travis CI continuous integration service. Travis CI can be configured to automatically build your project, and then to run any tests you have implemented. Travis builds and runs the tests upon every Github commit, so it is a valuable tool in ensuring that new code does not cause tests to fail.

As mentioned in the previous section, some issues occurred when using Travis with Hypothesis. Since some tests took a very long time to execute, an additional option had to be set in Travis to allow for long periods of time without output. Even with this setting, builds sometimes failed because Travis has an upper limit on how long it will wait before timing out.

To assess the test coverage of the tests I had written, I used the Coveralls service. This service integrates with Travis to automatically generate test coverage reports.

Both of these services offer badges (dynamically updated images) that can be displayed on Github so that it's easy to see at a glance whether your build is passing and how much of the code base is covered by your tests.

3 Software framework

In order to design and test alternate methods of computing expected payoff, we need a framework that supports creation and simulation of strategies and payoffs. To this end I implemented a `Strategy` class which implements basic functionality of a strategy such as tracking the player's history and computing the next move for the player. This `Strategy` class is never instantiated directly, but is instead subclassed by concrete implementations of strategies eg. `AllC`, `AllD`, `TitForTat` etc. This implementation should allow for any strategy to be implemented (including non-deterministic strategies) since the method that computes the next move for the strategy knows about both the player's own history as well as being passed the opponent's history.

We also implement a class for defining a payoff structure known as a payoff matrix. The payoff matrix encodes the payoffs for any combination of player actions, and so can be used to compute the payoff of a sequence of game results.

4 Calculation strategies

4.1 Closed form solutions

Closed form solutions for the expected payoff are known for combinations of simple strategies. Results from [4] were used to evaluate the simple strategies such as AllC, AllD, TFT and others that have the property that their behaviour is only dependent on at most the last move the opponent made. These results were used to assess the accuracy of the different calculation methods during initial trials.

4.2 Monte Carlo methods

The most basic calculation strategy is to simply simulate a large number of random games and to average the results. This class of methods are known as Monte Carlo methods. These methods are relatively easy to implement, but because they are by their nature randomized, we must conduct a large number of trials in order to arrive at an accurate result.

One method of Monte Carlo simulation is to simply run a large, fixed number of trials. Although this is easy to implement, it doesn't provide the user with any estimate of accuracy. Accuracy of the result must simply be evaluated against known, closed form solutions where they are available. Then the number of trials used can be assumed to be accurate for other situations for which a closed form solution is not available.

A more complex method of estimation is to compute the standard deviation of the estimator, and to stop conducting trials when this estimator standard deviation is within a certain threshold. This method and its derivation is explain in [8]. By computing the standard deviation of existing results, additional trials can be generated until eventually a guarantee of the accuracy of the answer is given. This result gives you a guarantee that the difference between the true value and the estimate you have produced is distributed with a given standard deviation.

Both methods of Monte Carlo simulation where implemented, though in practice the former method was used because it is less computationally expensive and is also more widely used and understood. Both methods were also reimplemented to make use of multiple processors using the Python `multiprocessing` library. Since individual trials of the Monte Carlo simulation are independent, it is efficiently to compute the results of trials in parallel and to aggregate the results once trials have been completed.

4.3 Brute force approach

One approach to calculation is to simply compute all of the possible outcomes and their associated likelihoods, and to combine these as a weighted sum. However as discussed above, in the presence of mistakes this quickly becomes prohibitively expensive. At each round in a repeated game, there are four possible mistake related outcomes. Both players can make no mistakes, either player can make a mistake, or both players can make a mistake. Therefore, the number of possible paths via which a repeated game can evolve grows like n^4 where n is the round number.

In order to attempt to control this explosion of outcomes, we can attempt to 'prune' the outcome tree so that it is more manageable. We can choose some cutoff term size ϵ, so that when a particular term in the infinite sum (payoff multiplied by the probability of reaching that state) drops below the threshold ϵ, we truncate that branch of the outcome tree.

In order to efficiently perform this calculation with pruning, we use a queue based algorithm. After each term is computed, the associated history and the probability of reaching that state is stored on a queue, but only if the term that is produced is above the ϵ threshold. This way, unlikely outcomes are pruned until all outcomes become sufficiently unlikely and the algorithm terminates.

The pseudocode for this algorithm can be found in listing 1.

4.4 Simplified approach

Another approach to computing expected payoff is to take a greatly simplified approach which reduces the number of outcomes that must be explored. This approach assumes that all games last a fixed length. In order to achieve the most accurate result, this fixed game length is chosen to be the expected number of rounds given the continuation probability. Since the length of games follows a geometric distribution, this value is $\frac{1}{\delta}$ where δ is the continuation probability. With this added limitation we can use the similar algorithm to the one described in listing 1 but only for games of the given length.

5 Results

6 Conclusion

References

[1] Robert Axelrod, *Effective choice in the Prisoner's Dilemma*, The Journal of Conflict Resolution, Vol. 24, No. 1, 1980.

[2] Andrzej Pelc, Krzysztof J. Pelc, *Same Game, New Tricks: What Makes a Good Strategy in the Prisoner's Dilemma?*, The Journal of Conflict Resolution, Vol. 35, No 5, 2009.

[3] Andrzej Pelc, *Fault-tolerant strategies in the Iterated Prisoner's Dilemma*, Information Processing Letters, Vol. 110, No. 10, 2010.

[4] Julian Garcia, Arne Traulsen, *The Structure of Mutations and the Evolution of Cooperation*, PLoS ONE, Vol. 7, No. 4, 2012.

[5] Robert Trivers, *The Evolution of Reciprocal Altruism*, The Quarterly Review of Biology, Vol. 46, No. 1, 1971.

[6] David R. MacIver, *Hypothesis*, https://github.com/DRMacIver/hypothesis, 2015.

Listing 1: An algorithm for efficiently estimating expected payoff

```
# Set up a variable for the expected payoff
expected_payoff = 0

# Set up a queue to hold the partial histories
q = Queue()

# Initialize the queue with an empty history, with probability 1
q.put((1, '', ''))

while not q.empty():

    # Get an item from the front of the queue
    item = q.get()

    # Set up Strategy objects with the given histories
    player_one = Strategy(item.history1)
    player_two = Strategy(item.history2)

    # Compute the moves that the strategies produce with the
       given histories, passing the opponent's history as well
    move_one = player_one.next_move(player_two.history)
    move_two = player_two.next_move(player_one.history)

    # Compute the probability of no mistakes occurring
    probability = item.probability * no_mistake_probability *
       continuation_probability

    # If this maximum possible term size is larger than the
       threshold
    if probability * max_payoff > epsilon:
        # Multiply this by the payoff from the outcome of a no-
           mistake round to find the term
        term = probability * payoff(move_one, move_two)
        # Add the term to the expected payoff
        expected_payoff += term
        # Add the probability along with the histories (
           including the new moves) back onto the queue
        q.put(probability,
              item.history1 + move_one,
              item.history2 + move_two)

    else:
        # The probability was too small, so don't add it back to
            the queue

    # Repeat this for each of the two one mistake cases and the
       two mistake case
```

[7] *Nose*, https://nose.readthedocs.org/en/latest/index.html, Version 1.3.7.

[8] Sheldon M. Ross, *Simulation*, Fourth edition, Elsevier Academic Press, 2006.