

[articles](#) [Q&A](#) [forums](#) [stuff](#) [lounge](#) [?](#)

Search for articles, questions, tips



Follow



# How to train your Pokerbot



CatchExAs, 11 Jul 2014



4.83 (26 votes)

Rate:

Using evolutionary techniques to deduce an optimal poker strategy

[Download PokerEvo.1.0.zip - 736.6 KB](#)

## Introduction

Back in 2008 when the global financial crisis was raging and the industry I worked in (Financial Services) was cutting software development costs (i.e. laying people off), I asked myself the question "What can one do to insulate themselves from the loss of their one single source of income?".

The only answers I could come up with were:

1. Become so good at what I do that it would be inconceivable even in a poor job market for companies to not want to hire me.
2. Build something that could work unattended and generate a second source of income.

So in an attempt to satisfy both these requirements I undertook the challenge 'build a software system that will play and win at on line poker unattended by the owner' aka PokerBot or as some might say 'a fools errand'.



*Two pair, but mix with alcohol and it'll seem like a fullhouse!*

Online poker seemed to me like a good arena because:

- All transactions are online and could be fully automated, meaning I don't have to take delivery of some physical commodity and worry about the related logistics.
- The participants in the market are overwhelmingly human and as we all know humans are fallable.
- The size of the market is large with many tens of thousands of players across numerous casinos.

I am not the first [person to investigate](#) or [open source the code](#) and will likely not be the last to attempt such a feat. But at the time, all the others seemed to be using C, C++ or a mixture of these with VB6. I wanted to start from scratch to help satisfy requirement **1**, using C# so that if I didn't achieve requirement **2** then I could at least maintain it or reuse some or all of the code elsewhere.

Whilst I've still not achieved either **1** (it's a never ending process) or **2** (I now make more money consulting than I would do from poker), over several years I did manage to create a winning **play-money** bot that can execute user configurable strategies.

There have been a number of articles posted recently (Paulo Zemek [1](#), [2](#) Peter Loew) concerning evolutionary and genetic algorithms which have rekindled my interest in a similar problem I confronted back then. As far as I know, it is one of the few applications of such an approach that you will find on codeproject that doesn't come straight from a book.

# Characterising the problem

A game of Texas Hold'em has 4 distinct phases:

1. **Preflop** - The dealer deals 2 cards to each player. The players then perform a round of betting and all survivors proceed to the next phase.
2. **Flop** - The dealer deals 3 communal cards and again the players perform a round of betting
3. **Turn** - 1 card is dealt communally and another round of betting occurs
4. **River** - 1 more card is dealt communally and the final betting round occurs

Cards are dealt either privately (aka the hole) or publically as part of a communal pool (aka the board) and the position of the dealer increments after each game (aka hand).

A betting round is defined as (in order from the dealer), each player sequentially given the options:

1. **Fold** - Exit the game and lose all the money they have staked so far
2. **Check** - If the stake required (aka amount to Call) is zero then pass to the next player in the sequence
3. **Call** - Bet the required amount to continue to the next phase
4. **Raise** - Increase the stake required for all players to continue to the next phase by X amount

The betting round will continue until each non-folding player has staked an equal amount of money in that particular phase.

Once the betting round completes, all staked money is added to a communal Pot and the game continues to the next phase.

We can summarize that the action each player performs should (at the very least) depend on:

1. The total money they have aka (their bank roll or stack)
2. The amount they have staked so far
3. The amount they need to bet in order to proceed to the next phase
4. And most importantly their view of how strong their hand is.

So if we are to devise an automated 'Strategy' then it must be able to:

- Respond to events during the lifetime of the game.
- Respond differently depending on the current state of the game
- Be capable of making different strategic decisions for different game phases
- Be able to re-evaluate a decision as a game phase evolves.

This implies that we might devise a strategy that would depend on 2 degrees of freedom, the number of phases in the game (preflop, flop, turn, river) and the possible actions that a player might use (check/fold, call, raise).

From this we could create a generic strategy that involves  $4 \times 3 = 12$  variables whose optimal values we don't yet know.

For example, imagine that the current game phase is pre-flop and we've been dealt hole cards whose relative strength we can evaluate in some way. It naturally follows that the action that our strategy should recommend should follow the pseudo code:

Hide Copy Code

```
Case P(win) < Y0

    Then Check if we can, otherwise Fold

Case Y1 > P(win) >= Y0
    Then Call if our bankroll allows, otherwise Fold
Case P(win) >= Y1
    Then Raise if our bankroll allows otherwise evaluate if we can call, check or fold
```

Where,  $P(\text{win})$  is the probability we will win and  $Y_n$  are some threshold values corresponding the actions available.

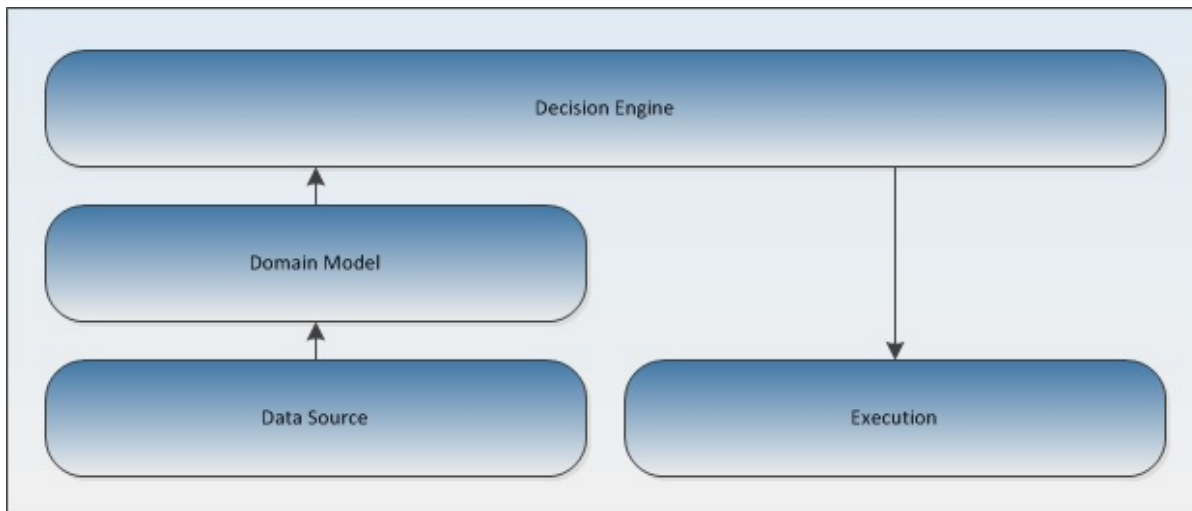
Whilst this is simplistic, this is reasonable starting point for understanding what the attached code is trying to achieve.

*NB. In reality  $P(\text{win})$  will be conditional (remember the Mont Hall problem?) on the current number of players and cards dealt, the amount we Raise should also vary and should be limited by the Kelly Criterion.*

## Anatomy of a Robot

Any real-time automated decisioning system will rely on 4 components:

1. **Datasource** which continuously drives and updates...
2. A representation of the current state aka **Domain Model**
3. The **Decision Engine** which knows about the current state and contains a set of rules which when satisfied will then trigger...
4. Actions that can be **Executed** either to the origin of the source or some other subsystem



*The general architecture of an automated decisioning system*

Discussing datasourcing and execution is beyond the scope of this article, but we will see later how the simulation mocks both.

## The Domain Model

Naturally, the domain model represents the current state of the poker game. It is designed to be driven by one or more driving threads that will provide text updates to the system. As such, most of the elements of the game are based on a **VisualEntity** class that raises events when it's text changes. These events bubble up through the model so that an observer may register it's interest in a particular occurrence. In our case we are interested in knowing when it's our player's turn to interact with the game.

Importantly Game raises the **OnInteractionRequired** event, using a Memory stream to take a complete value-copy of it's current state. If we were in multithreaded-land it would be very important to lock access to the model and then take a snapshot of state. We don't want to be in a position where the state we are currently making decisions on is subject to race conditions from any driving threads.

## The Decision Engine

### 1. Orchestrator

The observer of Game events is the **Orchestrator** and just like a conductor in an orchestra will decide and co-ordinate the various instruments and their musicians, our Orchestrator decides which strategies to use and what subsequent actions to apply.

### 2. Strategies

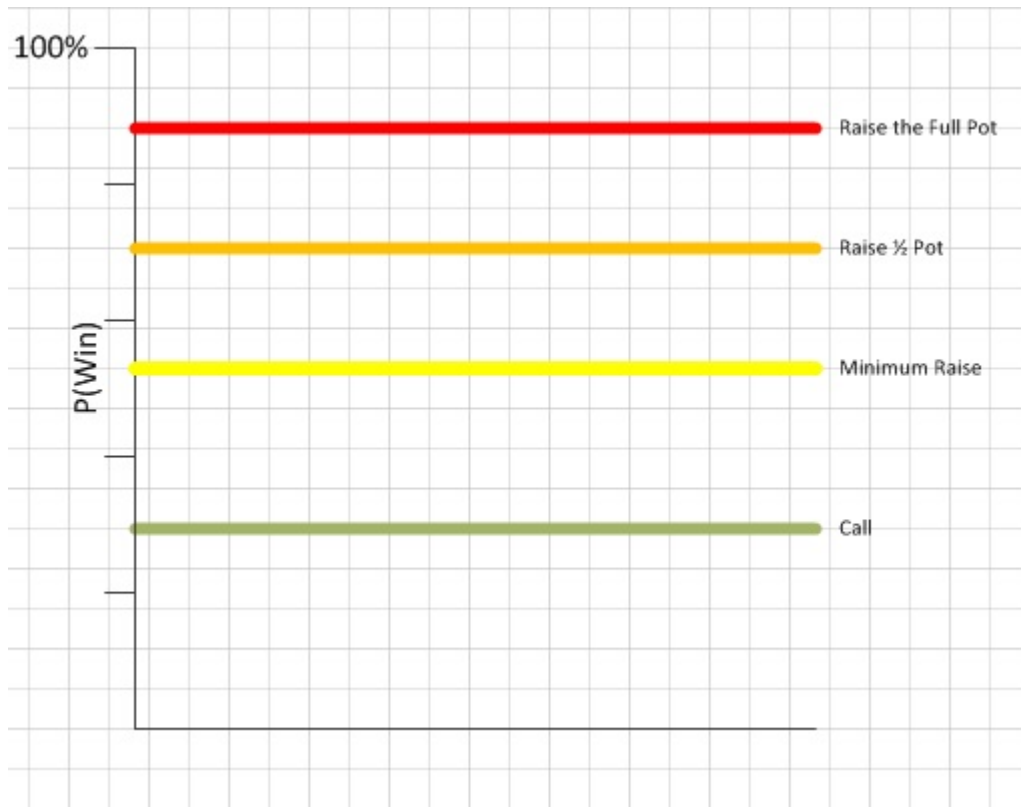
Once the Orchestrator confirms that an event has occurred that requires it to perform an action, it then asks the current strategy what action it recommends to perform.

In this case, we have defined a base class called **Strategy** with an abstract method to be implemented for each particular phase of the game life cycle. Our derived **MstStrategy** does a number of things:

- Determines what options have been displayed to the player.
- Uses an evaluator (detailed below) to calculate it's current win probability
- Based on the win probability climbs the ladder of thresholds defined by the MeanStratifiedThreshold in order and picks out the most aggressive action it can perform.

**Mean**, implies that we are using the mean player expectation as a base (e.g. if there are 10 players in the game then the mean chance of winning should be 10%), for deciding if a hand is worth spending money on i.e. Callable.

A **Threshold**, in this case implies that in order to perform an action the expectation of winning must be above a certain value.



*How our thresholds map to our probability of winning*

For example with 5 players in total and the Callable threshold is 1.2 x the Mean

$$\Rightarrow 1.2 \times 20\% = 24\%$$

So if our win expectation is greater than 24% then the strategy will Call.

The remaining probability (i.e 100% - Mean%), is then divided equally between all other distinct options hence being **Stratified**. Again each threshold is a multiple of the mean +/- modifier.

In the MstStrategy we have split the raise option into Minimum raise, Half pot raise and Full raise. For the first two if they are greater than the player's stack then the Strategy should recommend a Check or Fold, but for the Full raise it recommends the player go All-In.

### 3. Evaluators

**AllInEquity** is a class that wraps a C++ implementation of an All-In Equity calculator. Derived from poker-source and the [poker-eval library](#) and originally written by [James Devlin](#) my wrapper allows the .NET user to specify their hand, the board and the number of other players in the game to calculate their expected win percentage should all players go All-In.

The **BadBeat** class is provided in order to check whether either the current player or an opponent has a high chance of having a straight or flush. When running in real world conditions, I often found that opponents would stay in a low staked game and achieve

either of those hands on the river. To protect against this scenario BadBeat protection downgrades the suggested action to Check or Fold.

## Natural Selection

We have a generic strategy which 'plays by the numbers'. It depends on the current known state of the game but is also dependent on a number of thresholds whose optimal values we don't yet know.

How can we solve this problem?

We could probably use calculus to analytically solve what probably amounts to a boundary value problem. However, we could also use a more numerical 'brute force' approach to make an approximation.

I chose the 2nd approach because my math ability is not sufficiently advanced enough to even state the equations let alone know in advance if they can be solved! Maybe the [University of Alberta, Computer Poker Research Group](#) have?

## Simulating Texas Holdem

If we could simulate a poker game, fill it with robotic players (each with variants of our generic strategy), run it as fast as possible and let the robots fight it out to be the winner.

If we could also record which strategy variants won, the details of their strategy and replace losing robots with new ones (with different thresholds) then perhaps

If we ran it long enough patterns in the data might begin to emerge.

Assuming all of the above holds true then this presents us with the opportunity to calibrate the strategy and find the optimal values we would need to make the strongest pokerbot we can.

As the attached code was developed on and off between jobs and in my spare time it's state is what you might call 'the opposite of SOLID' and could certainly do with some more work to make it understandable to others. But lets assume that we only want to run it once and don't really care too much about re-use so here are the main points of interest...

The **Program** class and it's supporting methods simulate an ongoing game of Hold'em.

The main loop implements a never ending poker table, loading a representation of the visual state of a casino window and initialising the domain model.

It then generates 6 players, to fill the table's seats and selects a distinct strategy for each.

To make the algorithm more efficient we would like to weight the selection of new strategies such that thresholds that won before (or ones that are close to), are more likely to be used. That way useless threshold values will be discarded earlier and our simulation might more quickly 'zero in' on the optimal strategy.

I achieve this in the **GetPartialPivot()** function by selecting upto 3 variants who've lost the least and filling the remaining seats with some of the 'least used'.

[Hide](#) [Copy Code](#)

```
var ranked = dc.MeanStratifiedThresholds
    .Where(x => x.Used > 0 & x.Won > 0)
    .OrderBy(x => x.Used - x.Won).ToArray();

MeanStratifiedThreshold[] top = null;

if (ranked.Count() >= 3)
    top = ranked.Take(3).ToArray();
else if (ranked.Count() == 2)
```

```

    top = ranked.Take(2).ToArray();
else if (ranked.Count() == 1)
    top = ranked.Take(1).ToArray();

```

Each player is initialised with an observing Orchestrator and their Strategy variant, while the simulation itself observes the Orchestrator and updates the model.

The next loop in the nest simulates a poker hand until all but 1 player runs out of money. The player is declared the winner and the database is updated accordingly.

For each hand, the deck is shuffled randomly

[Hide](#) [Copy Code](#)

```

public void Shuffle()
{
    current.Clear();
    Random r = new Random((int)DateTime.Now.Ticks);

    while (current.Count < 52)
    {
        int index = r.Next(52);
        var cardToInsert = cards[index];
        if (!current.Contains(cardToInsert))
            current.Enqueue(cardToInsert);
    }
}

```

Cards are dequeued from the deck and dealt to players and the 2 players next to the dealer bet the Blinds.

The betting round loops until either, only one player is left in, all players are all-in or the amount left to call is zero. Each player is shown his options and a decision is triggered by setting his name to a count down (just like in a real on-line casino), before checking whether the round should complete.

[Hide](#) [Copy Code](#)

```

var currentSeat = GetSeat(seatIndex);

if (currentSeat.HasHoleCards & 0.0M < currentSeat.Player.PLayerDetail.Stack)
{
    noofPlayersNotAllIn = 0;
    amountLeftToCall = FindCallValue(theGame, currentSeat);

    foreach (var player in theGame.PlayersInGame)
        if (player.Stack != 0.0M)
            noofPlayersNotAllIn++;

    if (noofPlayersNotAllIn <= 1 & amountLeftToCall <= 0.0M)
        break;

    ShowOptions(theGame, currentSeat);
    string playerName = currentSeat.Player.Name.Text;

    //trigger an action from the player's associated orchestrator:
    Console.Write(string.Format("{0}:\\t", playerName));
    currentSeat.Player.Name.Text = string.Format("Time {0} secs", timeTrigger);
    currentSeat.Player.Name.Text = playerName;
}

```



Through, Preflop, Flop, Turn and River each phase is completed if enough players choose to stake money. Then at the end of the hand, for the players left in, the simulation determines the winner:

[Hide](#) [Copy Code](#)

```
private static Seat EvaluateWinner()
{
    PokerEquity.Evaluator ev = new PokerEquity.Evaluator();
    string board = theGame.Hand.TableCards;
    var res = new Dictionary<Seat, ulong>();
    foreach(Seat s in theGame.Table.Seats.Where (x =>x.HasHoleCards))
    {
        ulong val = ev.Evaluate(s.HoleCards.Text, board);
        res.Add(s, val);
    }
    var ordered = res.OrderBy( x => x.Value);

    var winner = ordered.Last();

    return winner.Key;
}
```

Before repeating the process again and again and again with the weak players dying off and the strong continuing to play...

## Running the code

Once you've expanded the zip file, PokerEvol.sln should load and compile in Visual Studio 2012 Express and greater.

Things to note:

1. There is a post build event that copies Poker.Equity.dll into the run directory
2. Poker.Equity.dll is a C++ library built in debug mode, so please only build your solution in the same mode.

You will also need to create a database in order to persist individual game results and store the strategy variants. Download SQL Express if you need and create a Strategy database then use the script StrategyDb\_Create.sql I provide to create the tables.

Next, prime the database with initial data by running PokerEvo.exe -setup

Execute the program with no arguments and the experiment will run the simulator until you hit CTRL+C.



```

Player.5987 is the dealer
Dealer:
Player.9640:Qd 6d
Player.9642:Td Kd
Player.9648:As Qh
Player.19671:Jh Th
Player.5987:3d 5s
Player.6322:Js 7d

Player.6322 bets 5.0 => stake = 5.0
Player.9640 bets 10.0 => stake = 10.0

Phase=Deal Board='Preflop'
Player.9642: Hand = TdKd, All-In Equity = 22.6690476190476
Player.9642 - Type=Fold Value=0.0
Player.9648: Hand = AsQh, All-In Equity = 26.4125
Player.9648 bets 10.0 => stake = 10.0
Player.9648 - Type=Call Value=10.0
Player.19671: Hand = JhTh, All-In Equity = 24.7241666666667
Player.19671 - Type=Fold Value=0.0
Player.5987: Hand = 3d5s, All-In Equity = 13.8443333333334
Player.5987 - Type=Fold Value=0.0
Player.6322: Hand = Js7d, All-In Equity = 23.3066666666667
Player.6322 bets 5.0 => stake = 10.0
Player.6322 - Type=Call Value=5.0
POT=30.0

Phase=Flop Board='3c 4s Qc'
Player.9648: Hand = AsQh, All-In Equity = 58.1516666666667
Player.9648 bets 10.0 => stake = 20.0
Player.9648 - Type=RaiseBy Value=10.0
Player.6322: Hand = Js7d, All-In Equity = 9.32333333333334
Player.6322 - Type=Fold Value=0.0
Player.9640: Hand = Qd6d, All-In Equity = 61.64
Player.9640 bets 30.0 => stake = 40.0
Player.9640 - Type=RaiseBy Value=20.0
Player.9648: Hand = AsQh, All-In Equity = 68.615
Player.9648 bets 50.0 => stake = 70.0
Player.9648 - Type=RaiseBy Value=30.0
Player.9640: Hand = Qd6d, All-In Equity = 62.0366666666667
Player.9640 bets 70.0 => stake = 110.0
Player.9640 - Type=RaiseBy Value=40.0
Player.9648: Hand = AsQh, All-In Equity = 69.0866666666667
Player.9648 bets 90.0 => stake = 160.0
Player.9648 - Type=RaiseBy Value=50.0
Player.9640: Hand = Qd6d, All-In Equity = 61.89
Player.9640 bets 110.0 => stake = 220.0
Player.9640 - Type=RaiseBy Value=60.0
Player.9648: Hand = AsQh, All-In Equity = 68.415
Player.9648 bets 130.0 => stake = 290.0
Player.9648 - Type=RaiseBy Value=70.0
Player.9640: Hand = Qd6d, All-In Equity = 61.2966666666667
Player.9640 bets 150.0 => stake = 370.0
Player.9640 - Type=RaiseBy Value=80.0
Player.9648: Hand = AsQh, All-In Equity = 69.0583333333333
Player.9648 bets 170.0 => stake = 460.0
Player.9648 - Type=RaiseBy Value=90.0
Player.9640: Hand = Qd6d, All-In Equity = 62.1683333333333
Player.9640 bets 190.0 => stake = 560.0
Player.9640 - Type=RaiseBy Value=100.0
Player.9648: Hand = AsQh, All-In Equity = 68.9633333333333
Player.9648 bets 210.0 => stake = 670.0
Player.9648 - Type=RaiseBy Value=110.0
Player.9640: Hand = Qd6d, All-In Equity = 61.1533333333333
Player.9640 bets 400.600 => stake = 960.600
Player.9640 - Type=AllIn Value=400.600
Player.9648: Hand = AsQh, All-In Equity = 68.76
Player.9648 bets 361.800 => stake = 1031.800
Player.9648 - Type=AllIn Value=361.800
POT=2002.400

Phase=Turn Board='3c 4s Qc 3h'

Phase=River Board='3c 4s Qc 3h Jc'
*****
Winner Player.9648 wins 1922.30400 with 'As Qh' & '3c 4s Qc 3h Jc'
'Player.9640' stack '0.00'
'Player.9642' stack '841.48'
'Player.9648' stack '1922.30'
'Player.19671' stack '995.40'
'Player.5987' stack '1287.10'
'Player.6322' stack '794.98'
Money at table = 5841.25400 vs 6000, Pot = 2002.400, Raked =158.74600

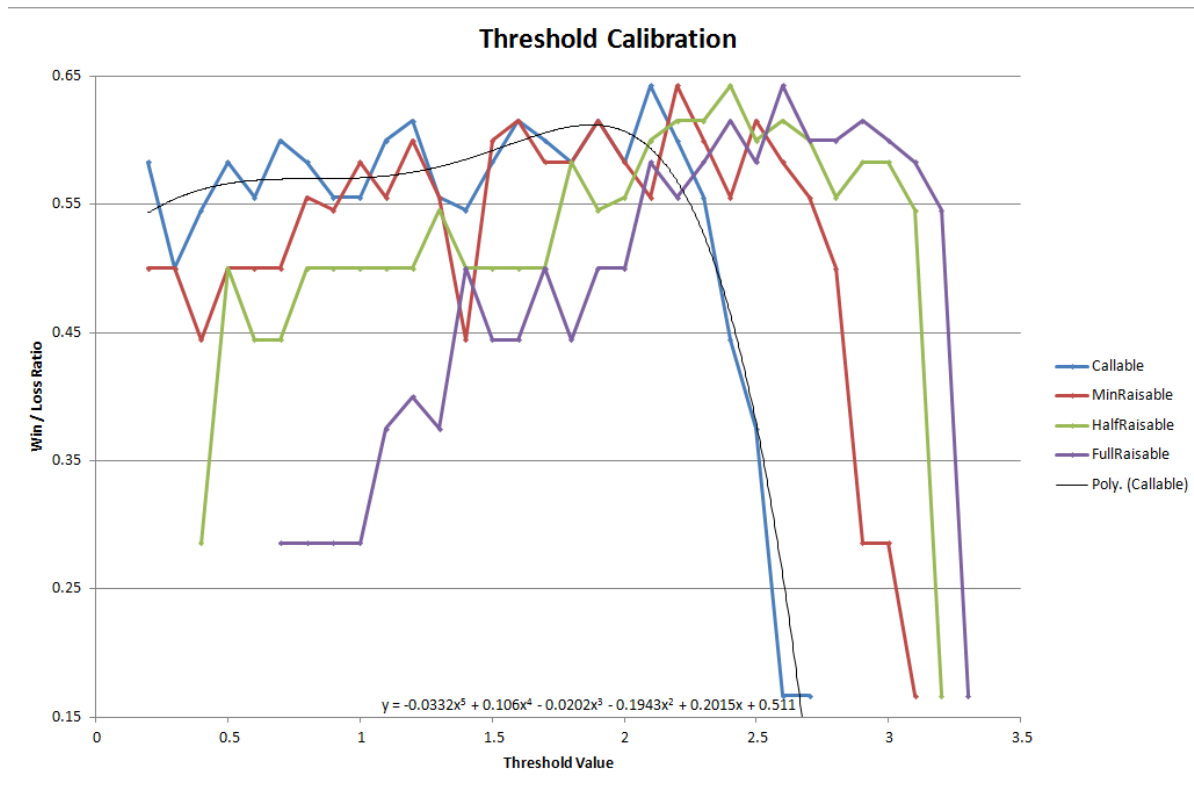
```

Here, 3 players reached the Flop but 2 felt their hands were strong enough to go all-in. Seems Player.9640 was wrong :(

## Fast forward 1 week...

Over the next few hours the database will be populated with data as the simulation returns results after every game is completed. Patterns in the data will begin to emerge over time and you can stop or pause the simulation without causing a fatal issue with the experiment.

Originally I ran this for a week however if you don't want to wait a long time doing it yourself, here are my results.



There is one marginal winner, representing the peaks of each of our plots:

Id	Callable	MinRaisable	HalfRaisable	FullRaisable	Win/Loss Ratio
40837	2.1	2.2	2.4	2.6	0.642857142857143

As we might have expected, as we increase each threshold value then our Win/Loss ratio gently rises to a peak before dropping off sharply. Hence, the tighter your strategy the more likely your pokerbot is to be the last man standing. However play too tightly and you'll lose your money in the ante.

Here's how I generated the graph above:

Add a View to the database with the following query:

Hide Copy Code

```
SELECT TOP (100) PERCENT Id, Callable, FullRaisable, HalfRaisable, MinRaisable, Used, Won,
CAST(Won AS float) / CAST(Used AS float) AS WinLossRatio
FROM dbo.MeanStratifiedThreshold AS m
WHERE (Used > 0) AND (Won > 0)
```

Then for each threshold add another View of the form:

Hide Copy Code

```
SELECT Callable, Max(WinLossRatio)
FROM dbo.MstWinLossRatio
```

Then simply import the results into Excel and generate the plots.

## Conclusions

Given a generic 'by the numbers' poker strategy, we were able to simulate a poker table long and often enough for an evolutionary process to take place. As 'less fit' strategies lost hands and money to the 'fitter' strategies they tended to die off earlier to be replaced with new variants that would also battle against the survivors.

We chose to vary our threshold values by one decimal point and within a constrained range, meaning we were able to pre-populate a table with each strategy variant. So our algorithm doesn't seem Genetic in nature, however we were able to use natural selection and weighting to mimic an evolutionary process.

As poker is a game of winner 'takes all' as well as chance then we should note that even an optimal strategy will lose some games. Consequently the simulation needs to run long enough for us to select the ones that have the highest win/loss percentage, but also ensure that we've given each a fair chance.

Other sources of error currently include:

- Split pots, which are not currently catered for
- Rounding differences as chips tend to be integers rather than decimals

Finally, (whilst not the aim of this article), what would happen if we used the results against real players?

Probably bad things so do **NOT** attempt to use! MST is just one type of strategy that could be improved upon in many ways. It does not cater for behavioral strategies encountered when playing against real players:- bluffing in particular.

## Addendum

Did you run the simulation? Why not post your results below.

Did you like/hate the article? I like reading feedback and am happy to make changes so please vote and comment.

I've also been nominated for 'Best Database' article for June, so feel free to head that way and [vote](#).

Cheers!

## History

#1 Basic article and code for publication

#2 Added diagrams, pictures and reformatted code examples, modified instructions as per feedback

#3 Added results and analysis of 1 week of computations including instructions

## License

This article, along with any associated source code and files, is licensed under [The Code Project Open License \(CPOL\)](#)


## Share

## About the Author



### CatchExAs



Technical Lead Alpha Integralis Limited  
United Kingdom 

Follow  
this Member

CatchExAs aka Nick Wilton runs Alpha Integralis, a software development consultancy to companies in the City of London.

Main interests include exploring Algo-Trading, Quant Development and Physics using C# and Java.

[www.nickwilton.info/Blog](http://www.nickwilton.info/Blog)

## You may also be interested in...

[Design pattern FAQ Part 1 \(Training\)](#)

[Dynamic Treeview with Drag and Drop by Kendo](#)

[GameTrainer: A Tool to Train Games](#)

[AI: Neural Network for Beginners \(Part 3 of 3\)](#)

[Solution Build Timer for VS  
2005/2013/2015/2017/2019](#)

[More Neural Network mathematics and code \(C#\)](#)

## Comments and Discussions





[First](#) [Prev](#) [Next](#)

**"poker-eval.lib" code**

**emotian** 21-Sep-17 19:24

**[My vote of 1] Sry but doesnt suit**

**Alexey KK** 3-Oct-15 7:16

Re: [My vote of 1] Sry but doesnt suit

**CatchExAs** 5-Oct-15 5:55

Re: [My vote of 1] Sry but doesnt suit

**Alexey KK** 5-Oct-15 6:13

Re: [My vote of 1] Sry but doesnt suit

**CatchExAs** 6-Oct-15 4:59

Re: [My vote of 1] Sry but doesnt suit

**Alexey KK** 6-Oct-15 5:51

Re: [My vote of 1] Sry but doesnt suit

**CatchExAs** 6-Oct-15 6:45

Re: [My vote of 1] Sry but doesnt suit

**Alexey KK** 6-Oct-15 7:15

Re: [My vote of 1] Sry but doesnt suit

**CatchExAs** 7-Oct-15 1:23

Re: [My vote of 1] Sry but doesnt suit

**Alexey KK** 7-Oct-15 2:23

Re: [My vote of 1] Sry but doesnt suit

**CatchExAs** 7-Oct-15 4:45

Re: [My vote of 1] Sry but doesnt suit

**Alexey KK** 7-Oct-15 5:19

Re: [My vote of 1] Sry but doesnt suit

**CatchExAs** 9-Oct-15 9:23

**My vote of 5**

**Member 11694860** 16-May-15 1:08

**But how to read the visible cards and interact with the poker game?**

**totty90** 15-Jan-15 2:31

Re: But how to read the visible cards and interact with the poker game?

**CatchExAs** 22-Jan-15 4:17

**Great**

**Sacha Barber** 17-Jul-14 0:06

[Re: Great](#) **CatchExAs** 17-Jul-14 13:28[My vote of 5](#) **Kel\_** 13-Jul-14 3:38[Re: My vote of 5](#) **CatchExAs** 17-Jul-14 13:29[My vote of 5](#) **Paulo Zemek** 12-Jul-14 9:30[Re: My vote of 5](#) **CatchExAs** 13-Jul-14 3:29[Good one..](#) **John Underhill** 9-Jul-14 14:05[Re: Good one..](#) **CatchExAs** 9-Jul-14 22:48[Nicely written - useful for the layman](#) **DaveCline** 9-Jul-14 9:38[Refresh](#)[1](#) [2](#) [Next »](#)[General](#) [News](#) [Suggestion](#) [Question](#) [Bug](#) [Answer](#) [Joke](#) [Praise](#) [Rant](#) [Admin](#)

Use Ctrl+Left/Right to switch messages, Ctrl+Up/Down to switch threads, Ctrl+Shift+Left/Right to switch pages.

[Permalink](#) | [Advertise](#) | [Privacy](#) | [Cookies](#) | [Terms of Use](#) | [Mobile](#)  
Web01 | 2.8.190404.1 | Last Updated 11 Jul 2014

언어 선택 | ▼

Layout: [fixed](#) | [fluid](#)Article Copyright 2014 by CatchExAs  
Everything else Copyright © [CodeProject](#), 1999-2019