Is it possible to make an AI that will play Texas Hold ‘em and beat a human player?

LAURENCE BROWN

Table of Contents

ANALYSIS:

RESEARCH:

Source	Link	Access Date	Summary
Article	https://en.wikipedia.org/wiki/Poker_probability_(Texas_hold_%27em)	17/09/16	Lots of formulas and probability calculations that can be used in the AI for making decisions about what card to play next, how much to bet, what cards other players have.
Web page	http://pokerpredictor.com/headsup	17/09/16	Reads your two inputted cards and calculates various probabilities such as win rates, and what it is strongest and weakest against. Also has other Texas Hold ‘em tools.
Article	http://www.codeproject.com/Articles/19091/More-Texas-Holdem-Analysis-in-C-Part	17/09/16	An article discussing a program that will calculate and analyse certain aspects of the game using several algorithms.
Article	https://en.wikipedia.org/wiki/Monte_Carlo_method	17/09/16	A method of calculating probabilities using randomness to solve problems. This method is used lots in poker to analyse and determine probabilities. Also see the Monte Carlo Algorithm. This algorithm is always fast, probably correct.
Article	https://en.wikipedia.org/wiki/Las_Vegas_algorithm	17/09/16	Another randomised algorithm that calculate various probabilities within poker, this algorithm is probably fast, always correct.
Article	http://www.codeproject.com/Articles/19092/More-Texas-Holdem-Analysis-in-C-Part	18/09/16	The second half of More Texas Hold ‘em Analysis in C# contains algorithms like the Monte Carlo algorithm and the code and analysis of them.
Article	http://pokercoder.blogspot.co.uk/2006/07/towards-meaningful-ordering-of-hands.html	18/09/16	An explanation of a poker AI going in depth with certain methods and such.
Rules set	https://en.wikipedia.org/wiki/Texas_hold_%27em#Rules	18/09/16	The rules for Texas Hold ‘em as explanations for them.

Article	https://www.partypoker.com/how-to-play/texas-holdem.html	18/09/16	A basic and comprehensive tutorial on how to play Texas Hold ‘em.
Article	https://www.partypoker.com/how-to-play/hand-rankings.html	18/09/16	A list of the hands in any game of poker ordered by their ranks.
Glossary	https://www.partypoker.com/how-to-play/school/basics/glossary.html	18/09/16	A list of all poker terms and explanations of them.
Article	https://en.wikipedia.org/wiki/Poker_Effective_Hand_Strength_(EHS)_algorithm	18/09/16	An algorithm that calculates the strength of a poker hand compared to all other hands.
Existing Solution	https://code.google.com/archive/p/openholdembot/	18/09/19	An existing open source Texas Hold ‘em AI that can be used to get ideas and inspiration from.
Existing Solution	http://poker.srv.ualberta.ca/	18/09/16	Another Hold ‘em AI coded by students of the University of Alberta, there are several programs on the webpage which shows the AI’s strategy. These will come in useful when looking for strategies for my AI to see which one is the best.
Existing Solution	https://code.google.com/archive/p/specialkpokereval/	18/09/16	A lightweight Hold ‘em hand evaluator AI.
Article	https://en.wikipedia.org/wiki/Artificial_neural_network	18/09/16	An article all about neural networks.
Article	http://www.codeproject.com/Articles/1028339/Basis-of-Neural-Networks-in-Visual-Basic-NET	18/09/16	An article which talks about neural networks, how they work and how to implement them into visual basic.
Book	https://www.amazon.com/dp/1880685000/?tag=stackoverfl08-20	18/09/16	Chapters discuss the value of deception, bluffing, raising, the slow-play, the value of position, psychology, heads-up play, game theory, implied odds, the free card, and semibluffing. These are all tactics that the AI could employ.
Journal	https://www.doc.ic.ac.uk/~nd/surprise_	18/06/16	Different types of neural networks are explained, demonstrated and applications are

	96/journal/vol4/cs11/report.html		given.
Video	https://youtu.be/h3l4qz76JhQ	18/06/16	A short video explaining how to create a simple neural network.

SELF-DISSCUSSION:

The aim of this project is to see if it possible to make an AI that will play Texas Hold ‘em and beat a player. I will achieve this by first of all making the game itself, which is an easy task, I will then implement an AI to play against a player.

Texas Hold ‘em is a variation of poker where all players are dealt two random cards, “Hole Cards” by the “Dealer”. The two players to the left of the dealer are the “Small Blind” and the “Big Blind” respectively. The “Small Blind” is required to bet a fixed starting bet and the “Big Blind” is required to bet double the “Small Blind”. Then the player to the left of the “Big Blind” will then start the first round of betting. The first player will assess their hand and if they think it is good enough for them to win they will “Call”, which means they will match the previous player's bet or they will “Raise”, which means they will bet more than the minimum bet to raise the minimum bet for other players. The player can also choose to go “All In” if they are certain they will win this round, which means they will bet all their money. If the player decides their cards are not worth playing with and they will probably lose, they will “Fold”, which means not betting and returning your cards to the dealer and skipping the round. Then each player will repeat this until one round of betting is over (no one else “Raises” or goes “All In” for a round). This is known as the “Pre-Flop”. Once all the betting has finished the three shared cards are dealt so that everyone can see (face up). This is called the “Flop”. Then another round of betting occurs and a fourth shared card, “The Turn”, is dealt. Another round of betting occurs and then the fifth and final shared card, “The River”, is dealt. A final round of betting then occurs. The hand can then end in one of two ways; the players turn over their “Hole Cards” and whoever has the best hand wins or someone will bet enough that all the other players fold and they win. The ultimate end goal is to turn a profit and to do that you don’t necessarily need to win every hand.

There are several ways in which I can make the game of poker for the AI to play on. I can use a GUI application or a console application. Since the program is not meant for an end user but is for investigation purposes I can use either method, a console application will be easier and less time consuming to develop yet the GUI application will be much easier to use and test the AI on. As it is not that hard to switch between the two I can always choose which one I want to use later down the road.

The game will be made in Python as I already have prior knowledge of the program and it has several abilities that will be useful to making an AI as well as math libraries like NumPy which will help me manage arrays in more powerful ways. Also if I do decide to make it into a GUI application it is easy to do so, with libraries like tkinter, which make it easy to make a GUI application. I will also use libraries like random which has several randomisation options. I can use this to select a random card from an array. This ensures the game is always fair.

There is also several ways in which I can get the AI to interact with the game. I could build

the AI directly into the game so they both operate in the same program but this could make it harder to troubleshoot when the program is not working to see if it is the AI or the game itself that is causing the program not to work. I could also make the game and the AI different programs and get them to communicate via text file or database, this however would be very time consuming and would not contribute much towards the investigation on the AI. If I were to store information about the game in a database this is what it would look like:

Flop 1	Flop 2	Flop 3	The Turn	The River	AI 1	AI 2
JS	KD	8S	8C	3H	7C	6D

This database can then be shared between the game and the AI allowing the AI to access its cards and the shared cards as they are revealed.

There are two main methods that I could use for the AI:

- I could make a general algorithm which will calculate the odds of my hand winning compared to all the other possible hands. This would be fast and efficient but predictable, meaning that the human player could easily figure out the AI's strategy and easily combat it. It would also only win if it had been repeatedly dealt good hands as if it is dealt a bad hand it would not be able to choose when or when not to bluff (out bet the other players so they all fold and you win).
- I could make a neural network that would learn the best ways to play over trial and error, each time becoming more and more effective at playing the game. A general algorithm would then control the weighting of the neural network according to their success. This method will not be as fast as I will have to run the AI with thousands of AI players, who do not know how to play the game. I will then use the general algorithm to determine which AI player was the best and then weight the ability of the players. A new generation of players would be made with the determined weighting and knowledge of the previous player that they were selected to be. This process happens over and over again through lots of hands and generations until the best individual AI players play really well. This process can be endless as there will always be small improvements but there will be diminishing return up until the point where the improvements are unnoticeable and don't make a difference. Once trained the best player is selected and put up against a human player. This method can be very good at playing Texas Hold 'em as there are no detectable patterns in the AIs logic and if trained correctly, with the right kind of dataset, it can bluff and win hands which the first algorithm would just fold on to save money. However this methods bluffing ability also has the chance to lose money if the bluff is unsuccessful.

The best method is clearly the neural network as it is the method used in most successful AI created for Texas Hold 'em. But if I need to change method later on, due to the complexity of neural networks, it is possible to do so.

Finally to test if the AI is better than a human player I will play the AI myself for a number of hands and judge who made a profit, NOT who won the most games as you can still lose the majority of games but make lots of profit on a few, thus winning overall as you made money and the other player(s) lost money (or gained less than you did). If the AI is

successful then I know that it is possible to create an AI that is better than a human if not then I can try to improve the AI until it wins or cannot be improved anymore, in that case I will know that it is not possible to create an AI with the resources I have that is better than a human but know that other AIs have been created that are far more successful than a human player.

TIMELINE:

Activity	Finish Date
Research	22/09/16
Analysis	06/10/16
Requirements	20/10/16
Design	17/11/16
Write Pseudocode for the Hold ‘em game	
Make Decision on whether it’s a console program or forms program.	
Technical Solution	26/01/17
Make the Hold ‘em game.	
Make a basic AI using a general algorithm to test the game.	
Plan the neural network and make a basic version of it.	
Test the basic neural network and make improvements.	
Make final neural network.	
Testing	23/02/17
Start the neural network playing so it can learn the game to an advanced level.	
Make improvements to neural network and game where needs be.	
Evaluation and Final Progress Check	16/03/17
Final Hand-in	30/03/17

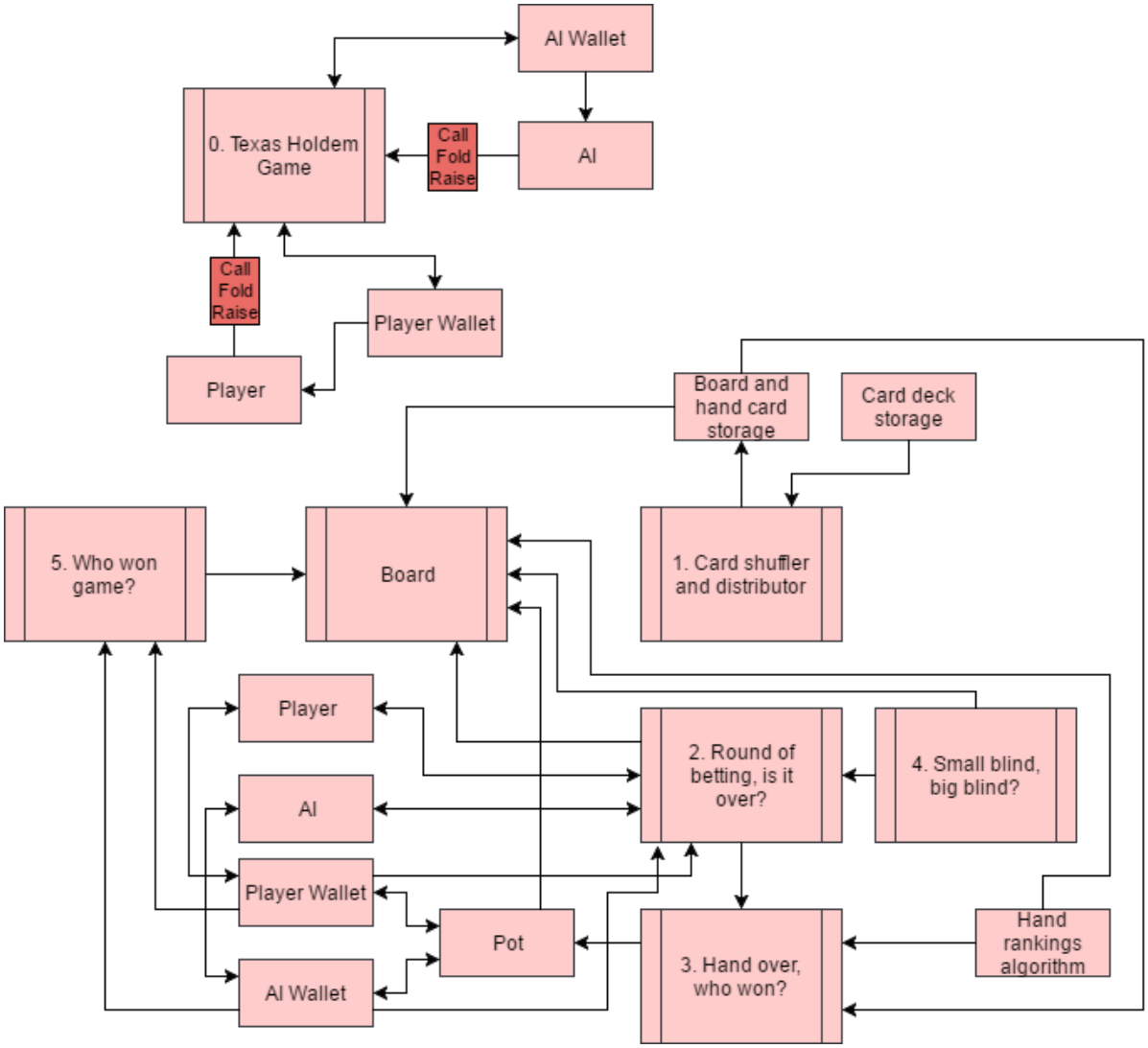
THE GAME:

I will make a poker game that will create two players (the human player and the AI) it will then assign two random cards (card1 and card2) from a deck of cards to each player. It will then also assign five more random cards from the deck to the table cards (flop1, flop2, flop3, turn, and river). I could assign each player big blind and small blind but since this is only a two player game of poker big blind and small blind are not relevant so I shall miss this out. The game will then play through a game of poker, deducting betting money as it goes along and displaying the visible table cards after every betting round when they are meant to be shown. Once the game of poker is finished it will then determine the winner by first checking if anyone has folded and if not it will use a card ranking algorithm to decide the winner. Once the winner is decided the game will then distribute money to the correct player and start a new game of poker, all the while keeping track of scores. To help me truly understand how the game is going to work I created an IPSO, dataflow diagram and flowchart for the game.

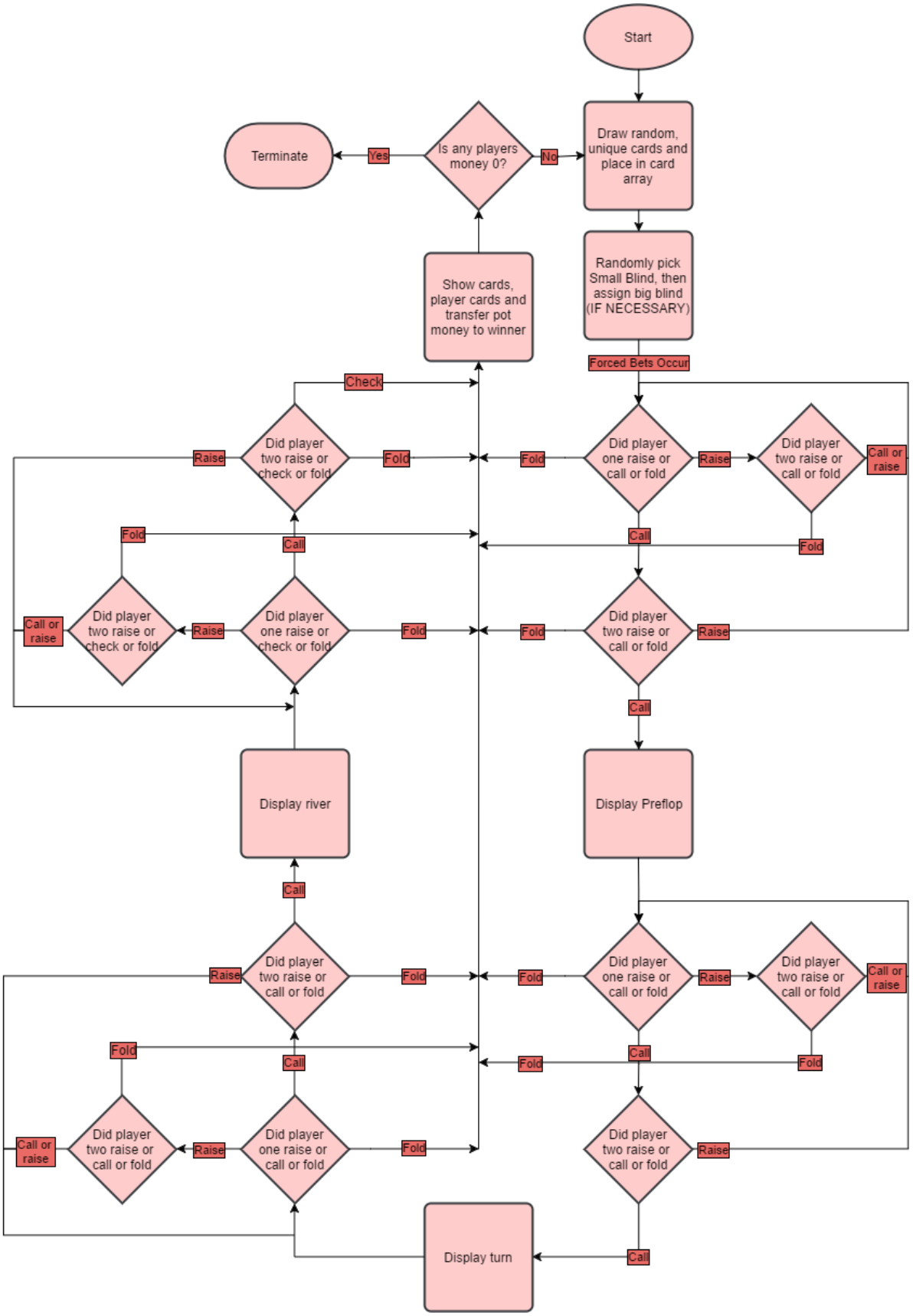
The following is the IPSO diagram for the game:

IN	PROCESS
Call Raise Bet amount (and All in) Fold	Compare hand with rankings to determine who wins. Money processing and transactions. Randomisation of cards. Determine whether the round of betting is over and whether the next card needs to be dealt. Determine who won the game when one player runs out of money.
STORE	**OUT**
Pot Players Money AIs Money Flop1 Flop2 Flop3 Turn River AIs Card 1 AIs Card 2 Players Card 1 Players Card 2 Big Blind Small Blind Hand Rankings Round	A representation of the cards on the game board. The player's current amount of money. The users hand. A representation of who won and with what hand in the hand rankings.

Dataflow diagram for the game:



Flowchart for the game:



This data dictionary that declares the classes and variables that could be used in the game:

Data Dictionary		
Name	Data Type	Regex
Pot	Decimal	(?:\d*\.)?\d+
Player Money	Decimal	(?:\d*\.)?\d+
AI Money	Decimal	(?:\d*\.)?\d+
Card	Class : String	[A+C+S+H]+([A+J+K+Q]|\d{1,10})
Betting stage	Integer	\d{0,3}
Turn	String	[A]+[i]|[P]+[l]
Winner	String	[A]+[i]|[P]+[l]

From this analysis I have formed a good idea of how a Texas Hold 'em game works and how I can make one. Since the main focus of the project is on the AI there are no notable changes or features of this game from the original other than the fact there is not big or small blind.

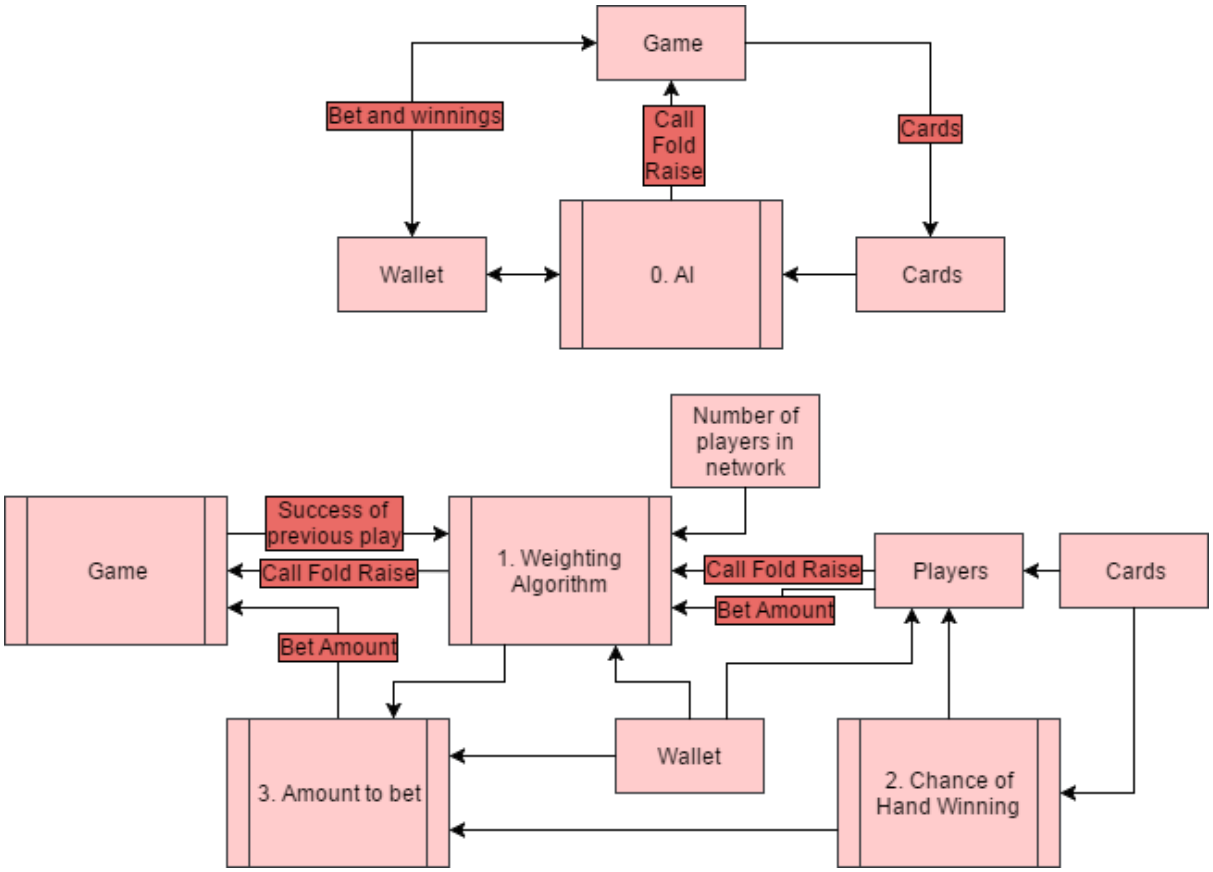
THE AI:

I plan to create a neural network for the AI. The neural network will train itself off a pre-existing database and then play the game, so that it has the highest chance of turning a profit as my research suggests.

The following is the IPSO diagram for the AI:

IN	PROCESS
AI Card 1 AI Card 2 Flop 1 Flop 2 Flop 3 Turn River	Money processing and transactions. Train network on dataset and adjust weights accordingly. Run current cards through weights to determine move.
STORE	**OUT**
Weighting of neural network Number of players in neural network Move	Call Raise Bet amount (and All in) Fold

Dataflow diagram for the AI:



The data dictionary that declares the classes and variables that could be used in the AI:

Data Dictionary		
Name	Data Type	Regex
Pot	Decimal	(?:\d*\.)?\d+
AI Money	Decimal	(?:\d*\.)?\d+
Weightings	Decimal	\d[\d|\.]+%
Player in network	Integer	\d

REQUIREMENTS:

1. The game interface easily readable and useable.
   1.1. The player should be able to easily read their cards and choose what action to make.
   1.2. The player must easily be able to see how much money is in the pot, their wallet and the AI's wallet.
2. The game must be able to store the card deck.
   2.1. The game program must be able to access it easily
   2.2. The game will have to be able to draw random cards from it and ensure the same ones are not drawn twice.
3. The game must be able to send and receive data with the AI.
   3.1. The AI's hand and the shared cards must be given to the AI program.
   3.2. The AI must be able to tell the game what action it wants to make.
   3.3. The game must be able to send and receive money with the AI
4. The game must be able to correctly handle all money and distribute it.
   4.1. The game must receive betting money and place it into a pot.
   4.2. The game must give all winnings to the winner.
5. The game must be able to determine the winner.
   5.1. It must be able to determine the winner by rating each player's hands then comparing them with each other.
6. The game must be able to determine whether a round of betting is over.
7. The AI must be able to receive various data inputs for processing.
   7.1. The AI needs to be able to receive all data from the game that is required.
      7.1.1. AI cards.
      7.1.2. Shared cards.
   7.2. The AI must be able to receive data from its wallet.
8. The AI must be able to use the data it has received and create a decision on what action it should make.
   8.1. The AI must be able to determine whether it should Fold, Call or Raise.
      8.1.1. It will need a dataset to train itself on.
      8.1.2. It will need to judge its move based on what the neural network returns when the AI current cards are put in.
   8.2. The AI must determine how much it wants to bet.

DESIGN:

FILES, DATA STRUCTURES, METHODS OF ACCESS:

What files I will need:

- A text file for moves to train the neural network against
- A text file for correct outcomes of moves needed to train the network.

Methods of access:

- I will use the dependency csv to manipulate my dataset.

PROCESSES:

The game:

There are multiple algorithms involved in the functioning of the game, below are some pseudo code examples of them.

Card assignment algorithm:

```
class CardPile:
deck = (array of all card combinations)
usedCards  = [ ]
def getCard():
    while True:
        randCard = deck[randint(0,51)]
        if randCard not in usedCards:
            usedCards.append(randCard)
            return randCard
            break
```

This algorithm seems to be the most efficient out of all the ways I found to do it. It would also be possible to compare to the randomly drawn cards to the other cards assigned to the plays and tables variables not a used cards variable. This way would be slower however and mean that the program is less secure.

Hand comparison algorithm:

```
class Compare:
    Hcards = (list of human and table cards)
    Acards = (list of AI and table cards)
    def cardCompare(hc,ac,tblc):
        if evaluateCard(Hcards) > evaluateCard(Acards):
            return 'human'
        elif evaluateCard(Acards) > evaluateCard(Hcards):
            return 'ai'
        else:
            return "no-one"

    def evaluateCard(hand):
```

```
        groups = group(['--23456789TJQKA'.index(r) for r, s in hand])
        counts, ranks = zip(*groups)
        print groups
        if ranks == (14, 5, 4, 3, 2):
            ranks = (5, 4, 3, 2, 1)
        straight = len(ranks) == 5 and max(ranks)-min(ranks) == 4
        flush = len(set([s for r, s in hand])) == 1
        return (
            9 if (5, ) == counts else
            8 if straight and flush else
            7 if (4, 1) == counts else
            6 if (3, 2) == counts else
            5 if flush else
            4 if straight else
            3 if (3, 1, 1) == counts else
            2 if (2, 2, 1) == counts else
            1 if (2, 1, 1, 1) == counts else
            0), ranks

    def group(self,items):
        groups = [(items.count(x), x) for x in set(items)]
        return sorted(groups, reverse = True)
```

Since there are few ways to make this algorithm and my investigation is on the AI I decided to use the card sorting algorithm from Udacity's "Design of Computer Programs" course. This way I can focus my time on making the neural network work instead of using it to create a card sorting algorithm.

Winner Calculator Algorithm:

```
    def CalcWinner
        If Showdown = True
            Return CompareCards
        If AI.LastTurn = Fold
            Return Player
        If Player.LastTurn = Fold
            Return AI
        If AI.Bank = 0 and Player.Bank > 0
            Return Player
        If Player.Bank = 0 and AI.Bank > 0
            Return AI
```

This algorithm finds who won the current game by using a series of simple if statements.

The AI:
There are multiple algorithms involved in the functioning of the AI, below are some pseudo code examples of them.
Neural Network:

```
x =handstrained.csv
y = correctpredtrained.csv
w = 2* random((3,1)) - 1
for t in xrange(100000):
        l0 = x
        l1 = 1/(1+exponential(-dotproduct(l0, w)))
        l1_error = y - l1
        l1_change = l1_error* l1*(1-l1)
        w += dotproduct(l0.T,l1_change)
```

This method would allow the AI to be unpredictable as well as play more human like, given the right dataset.

I could also make the AI by using simple probabilities found online. I could then use these probabilities and combine them with the AIs current cards to reach an outcome.

USER INTERFACE DESIGN:

Since this is an investigation on whether AI is able to beat a human at poker a UI element is not necessary. In order to use the program I shall use a command line interface and as the program runs it will print out necessary information along the way.

PACKAGES AND FRAMEWORK:

I will plan on using a scientific computing package such as NumPy (http://www.numpy.org/) to help with sigmoid function and matrices. I will also need to use the built in csv library in order to import and manipulate the datasets to train the neural network with. On top of this I will need the random (https://docs.python.org/2/library/random.html) dependency so I can generate random numbers where necessary.

DESIGN OF TESTING:

I will test several aspects of my program:

1. I will test the basic functionality of the game, to make sure it correctly carried out betting rounds and handles the end of the game correctly.
   1.1. To do this I will use test data to test each section of the game. My project is not focused on the game however so I will not test this in depth. The following is some example test data:

Nº	Purpose	Test Data	Expected result
1	To see if the game launches and prints the welcome screen.	Run the program.	Welcome to texas hold'em, please enter your name:
2	To see if a new player object is created when requested	Enter a name in the "Please enter your name" field.	You are now on round 1 of the game.
3	To see if the game is able to process a move given by the player.	Type "fold" in the "Please enter move" field.	You folded, AI wins. The score is AI - 1 Human - 0.
4	To see if the game is able to correctly determine the winner of the game using the card comparison algorithm.	Play the game until both you and the AI check during the final round of betting to enter the showdown.	[someone] wins with the cards [C1,C2]

   1.1. At the end of each test I will screenshot the outcome and assess if it passes or not.

2. I will test that the AI is able to correctly interface with the game.
   2.1. To do this I will play through several games each time playing in different styles and I will assess how the AI copes with this. Here is an example of some testing data:

Nº	Purpose	Test Data	Expected result
1	To see if the AI can handle the human player folding.	Run the program and fold on the first turn. Then see if the AI functions next game.	Human folds, AI wins. You are now on round 2 of the game. AI plays as normal.

   2.2. I will screen shot the outcome of the test and assess if it passes or not.

3. I will test that the AI correctly reads and trains itself off the dataset.
   3.1. To do this I will isolate the AI and see if it performs the correct move according to the dataset I give it.
   3.2. Here is some example testing data:

Nº	Purpose	Test Data	Expected result
1	To see if the AI makes the correct move according to the dataset given.	Give the AI cards that you would expect it to bet on e.g. AH and AC.	AI raises.

4. I will test that the ability of the AI and see if it is able to gain more money than a human over a period of 10 games.
   4.1. To do this I will play normally against the AI over 10 games with the aim for me to win. This gives the AI the toughest environment possible.
   4.2. I will keep a log of all turns I make, the outcome of each game and the cards each player had, along with the table cards, and finally how much money each player has at the end of each round.

TECHNICAL SOLUTION:

I started off my technical solution by exploring the different types of neural networks and decide what I would use to create my AI.
I first created a simple neural network that could play the game higher or lower by learning off a small dataset (the complete code is found at appendix A):

```
#define datasets
x = np.array(([[0],[12],[4],[7]]), dtype=float)
y = np.array([[1],[0],[1],[0]])

#normalise
x = x/12

#weights
w0 = 2*np.random.random((1,4))-1
w1 = 2*np.random.random((4,4))-1
w2 = 2*np.random.random((4,4))-1
w3 = 2*np.random.random((4,1))-1

#train
for t in xrange(100000):

    #forward propagation
    l0 = x
    l1 = sigma(np.dot(l0, w0))
    l2 = sigma(np.dot(l1, w1))
    l3 = sigma(np.dot(l2, w2))
    l4 = sigma(np.dot(l3, w3))

    #error + change calc
    l4_error = y - l4
    l4_change = l4_error*sigma_deriv(l4)
    l3_error = l4_change.dot(w3.T)
    l3_change = l3_error * sigma_deriv(l3)
    l2_error = l3_change.dot(w2.T)
    l2_change = l2_error * sigma_deriv(l2)
    l1_error= l2_change.dot(w1.T)
    l1_change = l1_error * sigma_deriv(l1)

    #update weights
    w3 += np.dot(l3.T, l4_change)
    w2 += np.dot(l2.T, l3_change)
    w1 += np.dot(l1.T, l2_change)
    w0 += np.dot(l0.T, l1_change)
```

It used a neural network that used gradient descent to learn. Gradient descent is where each weight in the network is assigned a random weight. The learning dataset (x) is then put through the weights and the outcome is recorded. The difference between this and the actual outcome store in dataset (y) is then calculated, this is the error. To find how much it needs to adjust the weights by, the change, it multiplies the error by the gradient of the point on the sigma function on which each layer lies. It then adjusts the weights by the dot product of the layer and the layer change needed. This whole process is repeated 100,000 times to achieve maximum accuracy. The user may then enter their own card, this then gets put through the weights and returns whether the next card is likely to be higher or lower.

This method worked well but only for small datasets. The dataset I will be using will be significantly bigger as it will store the move that should be made for each possible combination of the players hand cards.

I then found a neural network type that is able to handle large data sets. Using a method called “Stochastic gradient descent” the neural network is able to be trained quickly on a large dataset.

I then set about creating an AI that would learn what move to make in poker when given its two hand cards. First I had to create a dataset for the AI to learn off. I did this by writing a program that compared every possible hand combination with every possible hand combination it could go up against multiple times, each time with different table cards. I then assigned each card combination points based on how many times it won. From this I created a dataset. There are two files that make up the dataset, correctpredtrained.csv and handstrained.csv. The headers of each dataset look like the following:

correctpredtrained.csv:

Card1 float value (rank/14)	Card2 float value (rank/14)	Suited (1= suited, 0 = not suited)

handstrained.csv:

Correct prediction (1 = raise, 0 = fold)

The program took roughly eight hours to create the dataset which I am going to use for my AI. The code for this program can be found in appendix B. The dataset can be found in appendix C. After I had created this dataset I then moved on to create the Stochastic Gradient Descent neural network that would play AI:

```
8.      #define datasets
9.      __x = np.genfromtxt('handstrained.csv', delimiter=',')
10.     __y = np.genfromtxt('correctpredtrained.csv', delimiter=',')[np.newaxis]
11.     __y = __y.T
12.     #seed
13.     np.random.seed(1)
14.     #weights
15.     __w0 = 2*np.random.random((3,4))-1
16.     __w1 = 2*np.random.random((4,4))-1
17.     __w2 = 2*np.random.random((4,4))-1
18.     __w3 = 2*np.random.random((4,1))-1

36.         for i in xrange(10):
37.             self.epoch()

50.         def epoch(self):
51.         __z=0
52.         #train
53.         for t in xrange(len(self.__y)/4):
54.             #forward propagation
55.             __l0 = self.__x[__z:(__z+5)]
56.             __l1 = self.sigma(np.dot(__l0, self.__w0))
57.             __l2 = self.sigma(np.dot(__l1, self.__w1))
58.             __l3 = self.sigma(np.dot(__l2, self.__w2))
59.             __l4 = self.sigma(np.dot(__l3, self.__w3))
60.             #error + change calc
61.             __l4_error = self.__y[__z:(__z+5)] - __l4
62.             __l4_change = __l4_error*self.sigmaDeriv(__l4)
63.             __l3_error = __l4_change.dot(self.__w3.T)
64.             __l3_change = __l3_error * self.sigmaDeriv(__l3)
65.             __l2_error = __l3_change.dot(self.__w2.T)
66.             __l2_change = __l2_error * self.sigmaDeriv(__l2)
67.             __l1_error= __l2_change.dot(self.__w1.T)
```

```
            __l1_change = __l1_error * self.sigmaDeriv(__l1)
            #update weights
            self.__w3 += np.dot(__l3.T, __l4_change)
            self.__w2 += np.dot(__l2.T, __l3_change)
            self.__w1 += np.dot(__l1.T, __l2_change)
            self.__w0 += np.dot(__l0.T, __l1_change)
            __z += 5
```

(The full code can be found in appendix B)
Unlike the standard gradient descent neural network this method trains itself off chunks of the dataset at a time.

After I had solved the AI element I then needed to create a poker game for the AI to play against someone on. The following are some of the major algorithms in the game.

The playGame() function calls betting rounds in the right order, reveals table cards as they are needed, calls on the card comparison to determine the winner and returns the winner once run through.

```
    def playGame(self):
        self.__human.setMoney(-self.__table.getEntryMoney())
        self.__ai.setMoney(-self.__table.getEntryMoney())
        self.__table.setPot(2*self.__table.getEntryMoney())
        self.__table.setNextPlayer('r')
```

This is the preflop betting round.

```
        if self.bettingRound() == False:
            print "The table's cards are: " + str(self.__table.getCards(0)+",
"+self.__table.getCards(1)+", "+self.__table.getCards(2))
            self.__human.setAction('-')
            self.__ai.setAction('-')
```

This is the flop betting round.

```
            if self.bettingRound() == False:
                print "The table's cards are: " + str(self.__table.getCards(0)+",
"+self.__table.getCards(1)+", "+self.__table.getCards(2)+", "+self.__table.getCards(3))
                self.__human.setAction('-')
                self.__ai.setAction('-')
```

This is the turn betting round.

```
                if self.bettingRound() == False:
                    print "The table's cards are: " + str(self.__table.getCards(0)+",
"+self.__table.getCards(1)+", "+self.__table.getCards(2)+", "+self.__table.getCards(3)+",
"+self.__table.getCards(4))
                    self.__human.setAction('-')
                    self.__ai.setAction('-')
```

This is the river betting round.

```
                    if self.bettingRound() == False:
```

If neither player folds nor runs out of money during any of the betting rounds then the card comparison function is called to determine the winner.

```
                        self.__table.setWinner(Compare().cardCompare(self.__human.getCard(),
self.__ai.getCard(),[self.__table.getCards(0),self.__table.getCards(1),self.__table.getCards(
2),self.__table.getCards(3),self.__table.getCards(4)]))
                        if self.__table.getWinner() == 'Human':
                            self.__human.setMoney(self.__table.getPot())
                            self.__table.setPot(0)
                            self.__human.setScore(1)
                            self.__table.setWinnersCards(str(self.__human.getCard()))
                            return str(self.__human.getName())
                        elif self.__table.getWinner() == 'AI':
                            self.__ai.setMoney(self.__table.getPot())
                            self.__table.setPot(0)
                            self.__ai.setScore(1)
                            self.__table.setWinnersCards(str(self.__ai.getCard()))
                            return 'AI'
```

```
                        else:
                            self.__human.setMoney(self.__table.getPot()/2)
                            self.__ai.setMoney(self.__table.getPot()/2)
                            self.__table.setPot(0)
                            self.__table.setWinnersCards('')
                            return 'no-one'
```

If a player runs out of money or folds a winner will be chosen in this section.

```
        else:
            if self.__gameOver == 'ai':
                return str(self.__human.getName())
            if self.__gameOver == 'human':
                return 'AI'
            elif self.__table.getWinner() == 'Human':
                self.__table.setWinnersCards(str(self.__human.getCard()))
                return str(self.__human.getName())
            elif self.__table.getWinner() == 'AI':
                self.__table.setWinnersCards(str(self.__ai.getCard()))
                return 'AI'
            else:
                self.__table.setWinnersCards('')
                return 'no-one'
```

This function checks that both players still have money to play with else it will end the game.

```
    def checkMoney(self,flag):
        if int(self.__human.getMoney()) < 0 and flag == False:
                    self.__gameOver = 'human'
                    self.__table.setWinner('AI')
                    self.__table.setWon('t')
                    self.__ai.setScore(1)
                    return True
        elif int(self.__ai.getMoney()) < 0 and flag == False:
                    self.__gameOver = 'ai'
                    self.__table.setWinner('Human')
                    self.__table.setWon('t')
                    self.__human.setScore(1)
                    return True
        elif flag == True: return True
        else: return False
```

This function is the base level of the betting rounds, it displays each players money, checks that no one has folded and calls the checkMoney() algorithm as well as call each players individual betting round in the correct order.

```
    def bettingRound(self):
        self.__table.setWon('f')
        __flag = False
        if self.__table.getNextPlayer() == 1:
            while __flag == False:
                __flag = self.checkMoney(__flag)
                print "Your money: " + str(self.__human.getMoney())
                print "AI money: " + str(self.__ai.getMoney())
                print "Pot: " + str(self.__table.getPot())
                if __flag == False:
                    if self.bettingRoundHuman()== True and __flag == False:
                        self.__table.setNextPlayer('a')
                        __flag = True
                __flag = self.checkMoney(__flag)
                print "Your money: " + str(self.__human.getMoney())
                print "AI money: " + str(self.__ai.getMoney())
                print "Pot: " + str(self.__table.getPot())
                if __flag == False:
                    if self.bettingRoundAi() == True and __flag == False:
                        self.__table.setNextPlayer('h')
                        __flag = True
        elif self.__table.getNextPlayer() == 2:
```

```
            while __flag == False:
                __flag = self.checkMoney(__flag)
                print "Your money: " + str(self.__human.getMoney())
                print "AI money: " + str(self.__ai.getMoney())
                print "Pot: " + str(self.__table.getPot())
                if __flag == False:
                    if self.bettingRoundAi()== True and __flag == False:
                        self.__table.setNextPlayer('h')
                        __flag = True
                __flag = self.checkMoney(__flag)
                print "Your money: " + str(self.__human.getMoney())
                print "AI money: " + str(self.__ai.getMoney())
                print "Pot: " + str(self.__table.getPot())
                if __flag == False:
                    if self.bettingRoundHuman() == True and __flag == False:
                        self.__table.setNextPlayer('a')
                        __flag = True
        if self.__table.getWon() == True:
            return True
        else:
            return False
```

This is the AI betting round, it requests moves from the AI object and deals with the movement of money between players.

```
    def bettingRoundAi(self):
        self.__ai.setAction('x')
        if self.__ai.getAction() == 'f':
            print "AI folds with the cards: " + str(self.__ai.getCard())
            self.__human.setMoney(self.__table.getPot())
            self.__table.setPot(0)
            self.__human.setScore(1)
            self.__table.setWinner('Human')
            self.__table.setWon('t')
            return True
        elif self.__ai.getAction() == 'r':
            self.__ai.setRaiseAmount()
            self.__ai.setMoney(-self.__ai.getRaiseAmount())
            self.__table.setPot(self.__ai.getRaiseAmount())
            print "AI raises by " + str(self.__ai.getRaiseAmount())
            if self.__human.getAction() == 'r':
                self.__ai.setMoney(-self.__human.getRaiseAmount())
                self.__table.setPot(self.__human.getRaiseAmount())
                self.__table.setWon('f')
                return False
            elif self.__human.getAction() == 'c':
                self.__table.setWon('f')
                return False
            else:
                self.__table.setWon('f')
                return False
        elif self.__ai.getAction() == 'c':
            print "AI calls"
            if self.__human.getAction() ==  'r':
                self.__ai.setMoney(-self.__human.getRaiseAmount())
                self.__table.setPot(self.__human.getRaiseAmount())
                self.__table.setWon('f')
                return False
            elif self.__human.getAction() == 'c':
                self.__table.setWon('f')
                return True
            else:
                self.__table.setWon('f')
                return False
```

This is the human betting round, it requests moves from the user and deals with the movement of money between players.

```
    def bettingRoundHuman(self):
```

```
        print str(self.__human.getName())+", your cards are: " + str(self.__human.getCard())
        self.__human.setAction(raw_input("What is your move? (r/c/f): "))
        if self.__human.getAction() == 'f':
            self.__ai.setMoney(self.__table.getPot())
            self.__table.setPot(0)
            self.__ai.setScore(1)
            self.__table.setWinner('AI')
            self.__table.setWon('t')
            return True
        elif self.__human.getAction() == 'r':
            self.__human.setRaiseAmount(input("Raise by: "))
            self.__human.setMoney(-self.__human.getRaiseAmount())
            self.__table.setPot(self.__human.getRaiseAmount())
            if self.__ai.getAction() == 'r':
                self.__human.setMoney(-self.__ai.getRaiseAmount())
                self.__table.setPot(self.__ai.getRaiseAmount())
                        self.__table.setWon('f')
                        return False
                    elif self.__ai.getAction() == 'c':
                        self.__table.setWon('f')
                        return False
                    else:
                        self.__table.setWon('f')
                        return False
                elif self.__human.getAction() == 'c':
                    if self.__ai.getAction() == 'c':
                        self.__table.setWon('f')
                        return True
                    elif self.__ai.getAction() ==  'r':
                        self.__human.setMoney(-self.__ai.getRaiseAmount())
                        self.__table.setPot(self.__ai.getRaiseAmount())
                        self.__table.setWon('f')
                        return False
                    else:
                        self.__table.setWon('f')
                        return False
```

This is the card comparison function that decides which players hand was stronger.

```
class Compare:
    __allHcards = []
    __allAcards = []
```

First it creates two arrays; one for the humans and table cards and one for the AI and table cards.

```
    def cardCompare(self,hc,ac,tblc):
        self.__allHcards = [hc[0], hc[1], tblc[0], tblc[1], tblc[2], tblc[3], tblc[4]]
        self.__allAcards = [ac[0], ac[1], tblc[0], tblc[1], tblc[2], tblc[3], tblc[4]]
```

Here it decides which player had the highest rated hand by comparing tuples.

```
        if self.evaluateCard(self.__allHcards) > self.evaluateCard(self.__allAcards): return 'Human'
        elif self.evaluateCard(self.__allAcards) > self.evaluateCard(self.__allHcards): return 'AI'
        else: return "no-one"

```

This function creates a tuple that represents the players hand strength.

```
    def evaluateCard(self,hand):
        __groups = self.group(['--23456789TJQKA'.index(r) for r, s in hand])
        __counts, __ranks = zip(*__groups)
        if __ranks == (14, 5, 4, 3, 2):
            __ranks = (5, 4, 3, 2, 1)
        __straight = len(__ranks) == 5 and max(__ranks)-min(__ranks) == 4
        __flush = len(set([s for r, s in hand])) == 1
        return (
            9 if (5, ) == __counts else
```

```
            8 if __straight and __flush else
            7 if (4, 1) == __counts else
            6 if (3, 2) == __counts else
            5 if __flush else
            4 if __straight else
            3 if (3, 1, 1) == __counts else
            2 if (2, 2, 1) == __counts else
            1 if (2, 1, 1, 1) == __counts else
            0), __ranks

    def group(self,items):
        __groups = [(items.count(x), x) for x in set(items)]
        return sorted(__groups, reverse = True)
```

Since the card comparison algorithm is a difficult one to code and there are only a few certain ways to perform it I made use of a card comparison algorithm found on Udacity's "Design of Computer Programs" course.

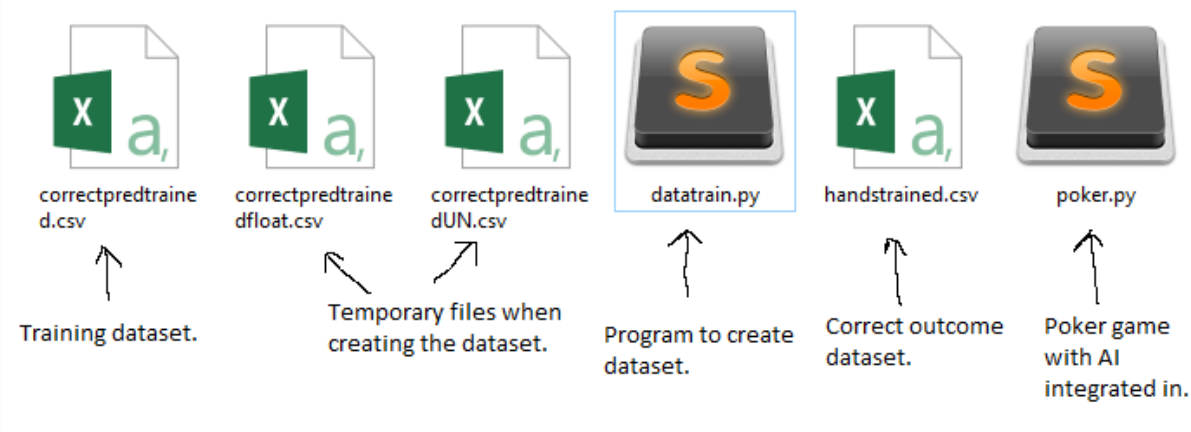
This is the card pile class that contains the deck array and distributes cards to players and the table.

```
class CardPile:
    __deck = [r+s for r in '23456789TJQKA' for s in 'SHDC']
    __usedCards  = []
    def getCard(self):
        while True:
            __randCard = self.__deck[randint(0,51)]
            if __randCard not in self.__usedCards:
                self.__usedCards.append(__randCard)
                return __randCard
                break
    def reset(self):
        self.__usedCards = []

game = Game('One')
game.menu()
```

After I had finished coding each separate algorithm I combined them all into one python file, added linking pieces of code and converted it into OOP style programming. The final result can be found in appendix E.

Here is an overview guide of the program:



TESTING:

I then moved onto the testing where I could verify that all the algorithms were working correctly and where I could start to investigate if it is possible for a poker AI to beat a human player. I started with testing the actual functionality of the game, to ensure that it was correctly dealing with inputs and playing through the game of poker correctly.

Game Functionality testing:

Nº	Purpose	Test Data	Expected result	Actual Result (evidence appendix F)
1.1	The game launches without errors.	N/A.	Games launches and prints welcome text.	Pass. (1.1)
1.2	The game successfully creates a new player object.	Enter “Player name” in the name field.	Game starts the first round of betting.	Pass. (1.2)
1.3a 1.3b 1.3c	The game is able to interpret a player’s move.	Enter: “f” “r” “c” In the move field.	a. Game recognises you wanted to fold and ends the round, declaring the AI as the winner. b. Game asks how much you want to raise by. c. Game proceeds with the betting round	a. Pass. (1.3a) b. Pass. (1.3b) c. Pass. (1.3c)
1.4	To see if the game correctly handles money.	Enter “1000” in the raise field.	The game deducts 1000 from the player’s wallet when raising.	Pass. (1.4)
1.5	To see if the card comparison algorithm works correctly.	Play until the end of a round.	The player who had the better set of cards wins.	Pass, the human won with two pairs. (1.5)
1.6	To see if the game is able to reset itself.	Enter “y” in the reset field.	The game resets each player’s money, score and game round.	Pass. (1.6)
1.7	To see if the game quits successfully.	Enter “y” in the quit field.	The game exits.	Pass. (1.7)
1.8	To see if the game displays relevant information about the game.	N/A	a. It displays information about the money. b. It displays information about the game score and round. c. The game displays information about player’s cards and the table cards. d. The game shows information about the winner.	a. Pass. (1.8a) b. Pass. (1.8b) c. Pass. (1.8c) d. Pass. (1.8d)

1.9	The game is able to correctly carry out an entire betting round	Play normally until the end of a betting round.	The game displays relevant information and doesn't run into any errors as I play.	Pass. (1.9)

Then I moved on to test the integration of the AI within the poker game to ensure that both algorithms were able to communicate successfully and effectively.

AI-Game communication testing:

Nº	Purpose	Test Data	Expected result	Actual Result (evidence appendix F)
2.1	To see if the AI can make a move within the game.	N/A	AI calls / folds / raises.	Pass. (2.1)
2.2	To see if the AI can handle the human player folding.	Run the program and fold on the first turn. Then see if the AI functions next game.	Human folds, AI wins. You are now on round 2 of the game. AI plays as normal.	Pass. (2.2)
2.3	To see if the AI is able to play after multiple resets.	Fold several times over and reset the game multiple times.	AI plays as normal.	Pass. (2.3)

Afterwards I then tested that the dataset was being read correctly by the AI and to see if the AI was learning off of it.

Tests to see if the AI is reading the dataset correctly.

Nº	Purpose	Test Data	Expected result	Actual Result (evidence appendix F)
3.1a 3.1b	To see if the AI makes the correct move according to the dataset given.	Give the AI cards that you would expect it to bet on: a. 8D, QH b. 3H, 2H	a. AI raises. b. AI folds.	a. Pass. (3.1a) b. Pass. (3.1b)

Finally I tested whether it is possible for a poker AI to beat a human player in order to get a conclusion for my investigation.

Tests to see if the AI can outperform a human:

Round Nº	Cards	Method of winning	Players money	Game results (evidence appendix F)
1	AI: QH, 2D Human: 2H, 2C Table: ?	Fold.	AI: 1,000,100 Human: 999,900	AI wins. (4.1)
2	AI: JD, 5D Human: JS, TS Table: KH, 2S, KD, 5H, 5C	Showdown.	AI: 995,168 Human: 994,958	Human wins. (4.2)
3	AI: 8S, KC Human: 9S, JH Table: 6H, 3C, TH, QC, 4D	Showdown.	AI: 1,000,131 Human: 999,869	AI wins. (4.3)
4	AI: JC, QD Human: QS, 8D Table: 4S, 9C, TC, 7C, 8C	Showdown.	AI: 1,050,523 Human: 944,807	AI wins. (4.4)
5	AI: 9D, 6D Human: 3S, KS Table: TD, 9H, 7D, 6S, 7H	Showdown.	AI: 1,058,863 Human: 940,037	AI wins. (4.5)
6	AI: JS, 2D Human: KH, 7C Table: 4C, 4H, 3D, QH, 3S	Showdown.	AI: 1,039,868 Human: 953,821	Human wins. (4.6)
7	AI: 4C, 7H Human: 9H, 8S Table: 6H, AD, QC, 5S, 8H	Showdown.	AI: 1,055,066 Human: 938,623	AI wins. (4.7)
8	AI: 2S, JH Human: 3D, 7D Table: ?	Fold.	AI: 1,055,166 Human: 938,523	AI wins. (4.8)
9	AI: 9S, 6S Human: KS, JC Table: 2C, TS, QS, 7C, TD	Showdown.	AI: 1,054,066 Human: 939,623	Human wins. (4.9)
10	AI: 5H, 6C Human: 7S, 5D Table: KH, KC, 5C, AC, 2D	Showdown.	AI: 1,048,775 Human: 951,225	Human wins. (4.10)

EVALUATION:

To evaluate and assess how well I carried out this investigation I must see if my solution meets the requirements.

I feel that my solution meets most, if not all, requirements I set out to complete. The testing shows that the following requirements were met.

Requirement	Met?	Explanation
1	✓	As evidenced in the screenshot the user was clearly able to see their cards and the table cards as well as input there desired action.
2	✓	The program was successfully able to deal cards to all positions and not draw duplicates.
3	✓	The second stage of testing showed that the AI was successfully able to communicate with the game.
4	✓	As it was demonstrated all throughout my testing that the game was able to constantly transfer money between the players and the pot, rewarding the winner with all the money acquired in the pot at the end of a game.
5	✓	The game was successfully able to determine a winner, as shown in the final stage of testing, if a player folded, if a player's money ran out or if it came to the showdown, a winner was picked.
6	✓	It was shown that in the last stage of testing as the game was clearly able to determine when a round of betting was over.
7	✓	The testing in stage 2 and 4 shows that the AI could make moves within the game and receive its cards and money.
8	✓	In the 3rd stage of testing where I tested if the AI was able to access a dataset and make a correct move when asked.

Since all of my requirements were met this provides a fair platform to put my investigation to test. However if I were to revisit this investigation I would have introduced more validation into the poker game to insure no foul input could be made. I would also create my own hand evaluation algorithm that would be more tailored to my game of poker. In addition to this I would have created a better dataset that does not just work off probabilities. Finally if I had extra time I would make the AI self-learning, so that after each game it played it would update the dataset appropriately according to what happened during that previous game.

To conclude, my investigation was to find out if it is possible to produce an AI that will earn more money from playing Texas hold'em than a human in 10 games. To test this I played my best against the AI for the duration of 10 games of poker and measured the amount of money each player had at the end of it. The final testing stage showed that AI is capable of playing poker better than an average human poker player as it beat me by making a profit of 48,775 over the duration of 10 games.

APPENDICIES:

A: Higher lower standard gradient descent neural network.

```
1.  import numpy as np
2.
3.  #make sigma
4.  def sigma(x):
5.      return 1/(1+np.exp(-x))
6.
7.  #sigma gradient
8.  def sigma_deriv(x):
9.      return x*(1-x)
10.
11. #define datasets
12. x = np.array(([[0],[12],[4],[7]]), dtype=float)
13. y = np.array([[1],[0],[1],[0]])
14.
15. #normalise
16. x = x/12
17.
18. #weights
19. w0 = 2*np.random.random((1,4))-1
20. w1 = 2*np.random.random((4,4))-1
21. w2 = 2*np.random.random((4,4))-1
22. w3 = 2*np.random.random((4,1))-1
23.
24. #train
25. for t in xrange(100000):
26.
27.     #forward propagation
28.     l0 = x
29.     l1 = sigma(np.dot(l0, w0))
30.     l2 = sigma(np.dot(l1, w1))
31.     l3 = sigma(np.dot(l2, w2))
32.     l4 = sigma(np.dot(l3, w3))
33.
34.     #error + change calc
35.     l4_error = y - l4
36.     l4_change = l4_error*sigma_deriv(l4)
37.     l3_error = l4_change.dot(w3.T)
38.     l3_change = l3_error * sigma_deriv(l3)
39.     l2_error = l3_change.dot(w2.T)
40.     l2_change = l2_error * sigma_deriv(l2)
41.     l1_error= l2_change.dot(w1.T)
42.     l1_change = l1_error * sigma_deriv(l1)
43.
44.     #update weights
45.     w3 += np.dot(l3.T, l4_change)
46.     w2 += np.dot(l2.T, l3_change)
47.     w1 += np.dot(l1.T, l2_change)
48.     w0 += np.dot(l0.T, l1_change)
49.
50. print "Output after training"
51. print l4
52.
53. #user entry
54. c = raw_input("Card: ")
55. C = np.array(([[c]]), dtype=float)
56.
57. #normalise
58. C = C/12
59.
60. print sigma(np.dot(sigma(np.dot(sigma(np.dot(sigma(np.dot(C, w0)), w1)), w2)), w3))
```

B: datatrain.py – dataset creation.

```
1.  import itertools, random, sys, math, time, csv
2.  from random import randint
3.  from time import sleep
4.  import numpy as np
5.
6.  def main():
7.      setup()
8.      train()
9.      DataSet().csvList()
10.
11. def setup():
12.     CardPile().reset()
13.     CardPile().setList()
14.
15. def train():
16.     for x in xrange(0,52):
17.         for y in xrange(0,52):
18.             if x!=y:
19.                 tp = TrainPlayer((x,y))
20.                 tp.setCards((CardPile().getSpecCard(tp.getCardNo()[0]),CardPile().getSpecCard
    (tp.getCardNo()[1])))
21.                 for i in xrange(0,52):
22.                     for j in xrange(0,52):
23.                         if i!=j and i!=x and i!=y and j!=x and j!=y and x!=y:
24.                             op = OppPlayer((i,j))
25.                             op.setCards((CardPile().getSpecCard(op.getCardNo()[0]),CardPile()
    .getSpecCard(op.getCardNo()[1])))
26.                             for epoch in xrange(0,100):
27.                                 tbl = Table([CardPile().getRandCard(),CardPile().getRandCard(
    ),CardPile().getRandCard(),CardPile().getRandCard(),CardPile().getRandCard()])
28.                                 tp.setPoints(int(Compare().cardCompare(tp.getCards(),op.getCa
    rds(),tbl.getCards())))
29.                                 CardPile().reset()
30.                                 CardPile().usedCardAdd(tp.getCards()[0])
31.                                 CardPile().usedCardAdd(tp.getCards()[1])
32.                                 CardPile().usedCardAdd(op.getCards()[0])
33.                                 CardPile().usedCardAdd(op.getCards()[1])
34.                             del op
35.                 DataSet().setPoint(tp.getCards()[0],tp.getCards()[1],tp.getPoints())
36.                 print tp.getCards()
37.                 del tp
38.
39.
40. class Compare:
41.     __allTcards = []
42.     __all0cards = []
43.     def cardCompare(self,tpc,opc,tblc):
44.         self.__allTcards = [tpc[0], tpc[1], tblc[0], tblc[1], tblc[2], tblc[3], tblc[4]]
45.         self.__allOcards = [opc[0], opc[1], tblc[0], tblc[1], tblc[2], tblc[3], tblc[4]]
46.         if self.evaluateCard(self.__allTcards) > self.evaluateCard(self.__allOcards):
47.             return 1
48.         elif self.evaluateCard(self.__allOcards) > self.evaluateCard(self.__allTcards):
49.             return 0
50.         else:
51.             return 0
52.
53.     def evaluateCard(self,hand):
54.         groups = self.group(['--23456789TJQKA'.index(r) for r, s in hand])
55.         counts, ranks = zip(*groups)
```

```
56.         if ranks == (14, 5, 4, 3, 2):
57.             ranks = (5, 4, 3, 2, 1)
58.         straight = len(ranks) == 5 and max(ranks)-min(ranks) == 4
59.         flush = len(set([s for r, s in hand])) == 1
60.         return (
61.             9 if (5, ) == counts else
62.             8 if straight and flush else
63.             7 if (4, 1) == counts else
64.             6 if (3, 2) == counts else
65.             5 if flush else
66.             4 if straight else
67.             3 if (3, 1, 1) == counts else
68.             2 if (2, 2, 1) == counts else
69.             1 if (2, 1, 1, 1) == counts else
70.             0), ranks
71.
72.     def group(self,items):
73.         groups = [(items.count(x), x) for x in set(items)]
74.         return sorted(groups, reverse = True)
75.
76. class Table:
77.     __flop1 = []
78.     __flop2 = []
79.     __flop3 = []
80.     __turn = []
81.     __river = []
82.     def __init__(self,cards):
83.         self.__flop1 = cards[0]
84.         self.__flop2 = cards[1]
85.         self.__flop3 = cards[2]
86.         self.__turn = cards[3]
87.         self.__river = cards[4]
88.     def getCards(self):
89.         return [self.__flop1, self.__flop2, self.__flop3, self.__turn, self.__river]
90.
91. class Player:
92.     __cardNo = []
93.     __cards = []
94.     def __init__(self, cardNo):
95.         self.__cardNo = cardNo
96.     def getCardNo(self):
97.         return self.__cardNo
98.     def setCards(self,cards):
99.         self.__cards = cards
100.            def getCards(self):
101.                return self.__cards
102.
103.         class TrainPlayer(Player):
104.             __points = 0
105.             def __init__(self, cardNo):
106.                 Player.__init__(self, cardNo)
107.             def getCardNo(self):
108.                 return Player.getCardNo(self)
109.             def setCards(self,cards):
110.                 Player.setCards(self,cards)
111.             def getCards(self):
112.                 return Player.getCards(self)
113.             def setPoints(self, points):
114.                 self.__points += points
115.             def getPoints(self):
116.                 return self.__points
117.
118.         class OppPlayer(Player):
119.             def __init__(self, cardNo):
120.                 Player.__init__(self, cardNo)
121.             def getCardNo(self):
```

```
122.                 return Player.getCardNo(self)
123.             def setCards(self,cards):
124.                 Player.setCards(self, cards)
125.             def getCards(self):
126.                 return Player.getCards(self)
127.
128.         class CardPile:
129.             __deck = [r+s for r in '23456789TJQKA' for s in 'SHDC']
130.             __usedCards  = []
131.             __list = []
132.             def getSpecCard(self,cardneeded):
133.                 __specCard = self.__deck[cardneeded]
134.                 self.__usedCards.append(__specCard)
135.                 return __specCard
136.             def getRandCard(self):
137.                 while True:
138.                     __randCard = self.__deck[randint(0,51)]
139.                     if __randCard not in self.__usedCards:
140.                         self.__usedCards.append(__randCard)
141.                         return __randCard
142.                         break
143.             def setList(self):
144.                 for x in xrange(0,52):
145.                     for y in xrange(0,52):
146.                         if y != x:
147.                             self.__list.append((x,y))
148.             def getList(self):
149.                 return self.__list
150.             def reset(self):
151.                 del self.__usedCards[:]
152.             def usedCardAdd(self, card):
153.                 self.__usedCards.append(card)
154.
155.         class DataSet:
156.             __cardList = []
157.             __pointList = []
158.             __floatList = []
159.             __predList = []
160.             def setPoint(self,card1,card2,point):
161.                 self.__cardList.append(self.processCards(card1,card2))
162.                 self.__pointList.append([point])
163.             def getPoint(self,pos):
164.                 return (self.__cardList[pos], self.__pointList[pos])
165.             def processCards(self,card1,card2):
166.                 if card1[:1] == 'T': c1 = 10
167.                 elif card1[:1] == 'J': c1 = 11
168.                 elif card1[:1] == 'Q': c1 = 12
169.                 elif card1[:1] == 'K': c1 = 13
170.                 elif card1[:1] == 'A': c1 = 14
171.                 else: c1 = int(card1[:1])
172.                 if card2[:1] == 'T': c2 = 10
173.                 elif card2[:1] == 'J': c2 = 11
174.                 elif card2[:1] == 'Q': c2 = 12
175.                 elif card2[:1] == 'K': c2 = 13
176.                 elif card2[:1] == 'A': c2 = 14
177.                 else: c2 = int(card2[:1])
178.                 if card1[-1:]==card2[-1:]: s = 1
179.                 else: s = 0
180.                 c1 = float(c1)/14
181.                 c2 = float(c2)/14
182.                 return [c1,c2,s]
183.             def pointFloat(self):
184.                 maxPoint = max(self.__pointList)[0]
185.                 self.__floatList = [[float(j)/maxPoint for j in i] for i in self.__pointLis
    t]
186.             def makePred(self, x):
```

```
187.                 return(
188.                     1 if x >= 0.6 else
189.                     0)
190.             def makePredList(self):
191.                 self.__predList = [[self.makePred(x) for x in z] for z in self.__floatList]
192.             def csvList(self):
193.                 with open('handstrained.csv', 'wb') as myfile:
194.                     wr = csv.writer(myfile, delimiter=',')
195.                     for z in xrange(0,len(self.__cardList)):
196.                         wr.writerow(self.__cardList[z])
197.                 with open('correctpredtrainedUN.csv', 'wb') as myfile:
198.                     wr = csv.writer(myfile, delimiter=',')
199.                     for z in xrange(0,len(self.__pointList)):
200.                         wr.writerow(self.__pointList[z])
201.                 self.pointFloat()
202.                 self.makePredList()
203.                 with open('correctpredtrainedfloat.csv', 'wb') as myfile:
204.                     wr = csv.writer(myfile, delimiter=',')
205.                     for z in xrange(0,len(self.__floatList)):
206.                         wr.writerow(self.__floatList[z])
207.                 with open('correctpredtrained.csv', 'wb') as myfile:
208.                     wr = csv.writer(myfile, delimiter=',')
209.                     for z in xrange(0,len(self.__predList)):
210.                         wr.writerow(self.__predList[z])
211.
212.         main()
```

C: Dataset

correctpredtrained.csv:

Card1 float value (rank/14)	Card2 float value (rank/14)	Suited (1= suited, 0 = not suited)
0.1428571429	0.1428571429	0
0.1428571429	0.1428571429	0
0.1428571429	0.1428571429	0
0.1428571429	0.2142857143	1
0.1428571429	0.2142857143	0
0.1428571429	0.2142857143	0

(Continued)

handstrained.csv:

Correct prediction (1 = raise, 0 = fold)
0
0
0
0
0
0

(Continued)

D: Higher lower stochastic gradient descent neural network.

```
1.  import numpy as np
2.  import csv, random, itertools
3.  from random import randint
4.
5.  class sdg_nn:
6.      __action = ''
7.
8.          #define datasets
9.      __x = np.genfromtxt('handstrained.csv', delimiter=',')
10.     __y = np.genfromtxt('correctpredtrained.csv', delimiter=',')[np.newaxis]
11.     __y = __y.T
12.     #seed
13.     np.random.seed(1)
14.     #weights
15.     __w0 = 2*np.random.random((3,4))-1
16.     __w1 = 2*np.random.random((4,4))-1
17.     __w2 = 2*np.random.random((4,4))-1
18.     __w3 = 2*np.random.random((4,1))-1
19.     #raise check
20.     __allReadyRaise = False
21.
22.     __move = 0
23.
24.     def setAction(self,card1,card2):
25.         self.__action = self.predict(card1,card2)
26.     def getAction(self):
27.         return self.__action
28.
29.     def sigma(self,x):
30.         return 1/(1+np.exp(-x))
31.     #sigma gradient
32.     def sigmaDeriv(self,x):
33.         return x*(1-x)
34.     def predict(self,c01,c02):
35.         __carray = self.processCards(c01,c02)
36.         for i in xrange(10):
37.             self.epoch()
38.         #predict
39.         __c1 = __carray[0]
40.         __c2 = __carray[1]
41.         __s = __carray[2]
42.         __C = np.array([[__c1,__c2,__s]])
43.         self.__move = self.sigma(np.dot(self.sigma(np.dot(self.sigma(np.dot(self.sigma(np.dot
    (__C, self.__w0)), self.__w1)), self.__w2)),self.__w3))
44.
45.         if self.__move >= 0.7 and self.__allReadyRaise == False:
46.             self.__allReadyRaise = True
47.             return 'r'
48.         elif self.__move >=0.5: return 'c'
49.         else: return 'f'
50.     def epoch(self):
51.         __z=0
52.         #train
53.         for t in xrange(len(self.__y)/4):
54.             #forward propagation
55.             __l0 = self.__x[__z:(__z+5)]
56.             __l1 = self.sigma(np.dot(__l0, self.__w0))
57.             __l2 = self.sigma(np.dot(__l1, self.__w1))
58.             __l3 = self.sigma(np.dot(__l2, self.__w2))
59.             __l4 = self.sigma(np.dot(__l3, self.__w3))
60.             #error + change calc
61.             __l4_error = self.__y[__z:(__z+5)] - __l4
62.             __l4_change = __l4_error*self.sigmaDeriv(__l4)
63.             __l3_error = __l4_change.dot(self.__w3.T)
```

```
64.             __l3_change = __l3_error * self.sigmaDeriv(__l3)
65.             __l2_error = __l3_change.dot(self.__w2.T)
66.             __l2_change = __l2_error * self.sigmaDeriv(__l2)
67.             __l1_error= __l2_change.dot(self.__w1.T)
68.             __l1_change = __l1_error * self.sigmaDeriv(__l1)
69.             #update weights
70.             self.__w3 += np.dot(__l3.T, __l4_change)
71.             self.__w2 += np.dot(__l2.T, __l3_change)
72.             self.__w1 += np.dot(__l1.T, __l2_change)
73.             self.__w0 += np.dot(__l0.T, __l1_change)
74.             __z += 5
75.     def processCards(self,card01,card02):
76.         if card01[:1] == 'T': __c1 = 10
77.         elif card01[:1] == 'J': __c1 = 11
78.         elif card01[:1] == 'Q': __c1 = 12
79.         elif card01[:1] == 'K': __c1 = 13
80.         elif card01[:1] == 'A': __c1 = 14
81.         else: __c1 = int(card01[:1])
82.         if card02[:1] == 'T': __c2 = 10
83.         elif card02[:1] == 'J': __c2 = 11
84.         elif card02[:1] == 'Q': __c2 = 12
85.         elif card02[:1] == 'K': __c2 = 13
86.         elif card02[:1] == 'A': __c2 = 14
87.         else: __c2 = int(card02[:1])
88.         if card01[-1:]==card02[-1:]: __s = 1
89.         else: __s = 0
90.         __c1 = float(__c1)/14
91.         __c2 = float(__c2)/14
92.         return [__c1,__c2,__s]
93.
94. card1 = raw_input("Card 1: ")
95. card2 = raw_input("Card 2: ")
96. sdg_nn().setAction(card1,card2)
97. print str(sdg_nn().getAction())
```

E: poker.py – The final solution.

```
1.  #Import dependancies
2.  import sys
3.  from random import randint
4.  import numpy as np
5.
6.  class Game:
7.      __gRound = 1 #game round
8.      __quit = False
9.      __reset = False
10.     __gName = '' #game name
11.     __ai = None
12.     __human = None
13.     __gameOver = ''
14.     __table = None
15.
16.     def __init__(self, gName):
17.         self.__gName = gName
18.         print "Welcome to texas holdem!"
19.     def setupGame(self):
20.         self.__human = Player(raw_input("Please enter your name: "))
21.         self.__ai = AI()
22.         self.__table = Table()
23.         self.__human.setMoney(1000000)
24.         self.__ai.setMoney(1000000)
25.     def menu(self):
26.         while self.__quit == False:
27.             if self.__gRound == 1 or self.__reset == True:
28.                 self.setupGame()
29.                 self.__gRound = 1
30.             print "You are on round: " + str(self.__gRound)
31.             CardPile().reset()
32.             self.setupCards()
33.             print "Congratulations " + str(self.playGame()) + ", you won!"
34.             print "The AI's cards were: " + str(self.__ai.getCard())
35.             print str(self.__human.getName()) + ", your cards were:
    " + str(self.__human.getCard())
36.             print "The score is: AI: " + str(self.__ai.getScore()) + " |
    " + str(self.__human.getScore()) + " :" + str(self.__human.getName()).upper()
37.             if self.__gameOver == 'ai' or self.__gameOver == 'human':
38.                 print "Since " + str(self.__gameOver) + " ran out of money the game is over.
    The game will now be reset."
39.                 self.__reset = True
40.                 if raw_input("Would you like to quit the game?
    (y/n):") == 'y': self.__quit = True
41.                 else: self.__quit = False
42.             else:
43.                 self.__gRound = self.__gRound + 1
44.                 if raw_input("Would you like to reset the game?
    (y/n):") == 'y': self.__reset = True
45.                 else: self.__reset = False
46.                 if raw_input("Would you like to quit the game?
    (y/n):") == 'y': self.__quit = True
47.                 else: self.__quit = False
48.         sys.exit()
49.     def setupCards(self):
50.         self.__table.setCards()
51.         self.__human.setCard()
52.         self.__ai.setCard()
53.         self.__ai.reset()
54.     def playGame(self):
55.         self.__human.setMoney(-self.__table.getEntryMoney())
56.         self.__ai.setMoney(-self.__table.getEntryMoney())
57.         self.__table.setPot(2*self.__table.getEntryMoney())
58.         self.__table.setNextPlayer('r')
```

```
59.         if self.bettingRound() == False:
60.             print "The table's cards are: " + str(self.__table.getCards(0)+",
    "+self.__table.getCards(1)+", "+self.__table.getCards(2))
61.             self.__human.setAction('-')
62.             self.__ai.setAction('-')
63.             if self.bettingRound() == False:
64.                 print "The table's cards are: " + str(self.__table.getCards(0)+",
    "+self.__table.getCards(1)+", "+self.__table.getCards(2)+", "+self.__table.getCards(3))
65.                 self.__human.setAction('-')
66.                 self.__ai.setAction('-')
67.                 if self.bettingRound() == False:
68.                     print "The table's cards are: " + str(self.__table.getCards(0)+",
    "+self.__table.getCards(1)+", "+self.__table.getCards(2)+", "+self.__table.getCards(3)+",
    "+self.__table.getCards(4))
69.                     self.__human.setAction('-')
70.                     self.__ai.setAction('-')
71.                     if self.bettingRound() == False:
72.                         self.__table.setWinner(Compare().cardCompare(self.__human.getCard(),
    self.__ai.getCard(),[self.__table.getCards(0),self.__table.getCards(1),self.__table.getCards(
    2),self.__table.getCards(3),self.__table.getCards(4)]))
73.                         if self.__table.getWinner() == 'Human':
74.                             self.__human.setMoney(self.__table.getPot())
75.                             self.__table.setPot(0)
76.                             self.__human.setScore(1)
77.                             self.__table.setWinnersCards(str(self.__human.getCard()))
78.                             return str(self.__human.getName())
79.                         elif self.__table.getWinner() == 'AI':
80.                             self.__ai.setMoney(self.__table.getPot())
81.                             self.__table.setPot(0)
82.                             self.__ai.setScore(1)
83.                             self.__table.setWinnersCards(str(self.__ai.getCard()))
84.                             return 'AI'
85.                         else:
86.                             self.__human.setMoney(self.__table.getPot()/2)
87.                             self.__ai.setMoney(self.__table.getPot()/2)
88.                             self.__table.setPot(0)
89.                             self.__table.setWinnersCards('')
90.                             return 'no-one'
91.         else:
92.             if self.__gameOver == 'ai':
93.                 return str(self.__human.getName())
94.             if self.__gameOver == 'human':
95.                 return 'AI'
96.             elif self.__table.getWinner() == 'Human':
97.                 self.__table.setWinnersCards(str(self.__human.getCard()))
98.                 return str(self.__human.getName())
99.             elif self.__table.getWinner() == 'AI':
100.                     self.__table.setWinnersCards(str(self.__ai.getCard()))
101.                     return 'AI'
102.                 else:
103.                     self.__table.setWinnersCards('')
104.                     return 'no-one'
105.         def checkMoney(self,flag):
106.             if int(self.__human.getMoney()) < 0 and flag == False:
107.                     self.__gameOver = 'human'
108.                     self.__table.setWinner('AI')
109.                     self.__table.setWon('t')
110.                     self.__ai.setScore(1)
111.                     return True
112.             elif int(self.__ai.getMoney()) < 0 and flag == False:
113.                     self.__gameOver = 'ai'
114.                     self.__table.setWinner('Human')
115.                     self.__table.setWon('t')
116.                     self.__human.setScore(1)
117.                     return True
118.             elif flag == True: return True
```

```
119.                 else: return False
120.             def bettingRound(self):
121.                 self.__table.setWon('f')
122.                 __flag = False
123.                 if self.__table.getNextPlayer() == 1:
124.                     while __flag == False:
125.                         __flag = self.checkMoney(__flag)
126.                         print "Your money: " + str(self.__human.getMoney())
127.                         print "AI money: " + str(self.__ai.getMoney())
128.                         print "Pot: " + str(self.__table.getPot())
129.                         if __flag == False:
130.                             if self.bettingRoundHuman()== True and __flag == False:
131.                                 self.__table.setNextPlayer('a')
132.                                 __flag = True
133.                         __flag = self.checkMoney(__flag)
134.                         print "Your money: " + str(self.__human.getMoney())
135.                         print "AI money: " + str(self.__ai.getMoney())
136.                         print "Pot: " + str(self.__table.getPot())
137.                         if __flag == False:
138.                             if self.bettingRoundAi() == True and __flag == False:
139.                                 self.__table.setNextPlayer('h')
140.                                 __flag = True
141.                 elif self.__table.getNextPlayer() == 2:
142.                     while __flag == False:
143.                         __flag = self.checkMoney(__flag)
144.                         print "Your money: " + str(self.__human.getMoney())
145.                         print "AI money: " + str(self.__ai.getMoney())
146.                         print "Pot: " + str(self.__table.getPot())
147.                         if __flag == False:
148.                             if self.bettingRoundAi()== True and __flag == False:
149.                                 self.__table.setNextPlayer('h')
150.                                 __flag = True
151.                         __flag = self.checkMoney(__flag)
152.                         print "Your money: " + str(self.__human.getMoney())
153.                         print "AI money: " + str(self.__ai.getMoney())
154.                         print "Pot: " + str(self.__table.getPot())
155.                         if __flag == False:
156.                             if self.bettingRoundHuman() == True and __flag == False:
157.                                 self.__table.setNextPlayer('a')
158.                                 __flag = True
159.                 if self.__table.getWon() == True:
160.                     return True
161.                 else:
162.                     return False
163.             def bettingRoundAi(self):
164.                 self.__ai.setAction('x')
165.                 if self.__ai.getAction() == 'f':
166.                     print "AI folds with the cards: " + str(self.__ai.getCard())
167.                     self.__human.setMoney(self.__table.getPot())
168.                     self.__table.setPot(0)
169.                     self.__human.setScore(1)
170.                     self.__table.setWinner('Human')
171.                     self.__table.setWon('t')
172.                     return True
173.                 elif self.__ai.getAction() == 'r':
174.                     self.__ai.setRaiseAmount()
175.                     self.__ai.setMoney(-self.__ai.getRaiseAmount())
176.                     self.__table.setPot(self.__ai.getRaiseAmount())
177.                     print "AI raises by " + str(self.__ai.getRaiseAmount())
178.                     if self.__human.getAction() == 'r':
179.                         self.__ai.setMoney(-self.__human.getRaiseAmount())
180.                         self.__table.setPot(self.__human.getRaiseAmount())
181.                         self.__table.setWon('f')
182.                         return False
183.                     elif self.__human.getAction() == 'c':
184.                         self.__table.setWon('f')
```

```
185.                         return False
186.                     else:
187.                         self.__table.setWon('f')
188.                         return False
189.                 elif self.__ai.getAction() == 'c':
190.                     print "AI calls"
191.                     if self.__human.getAction() ==  'r':
192.                         self.__ai.setMoney(-self.__human.getRaiseAmount())
193.                         self.__table.setPot(self.__human.getRaiseAmount())
194.                         self.__table.setWon('f')
195.                         return False
196.                     elif self.__human.getAction() == 'c':
197.                         self.__table.setWon('f')
198.                         return True
199.                     else:
200.                         self.__table.setWon('f')
201.                         return False
202.             def bettingRoundHuman(self):
203.                 print str(self.__human.getName())+", your cards are:
    " + str(self.__human.getCard())
204.                 self.__human.setAction(raw_input("What is your move? (r/c/f): "))
205.                 if self.__human.getAction() == 'f':
206.                     self.__ai.setMoney(self.__table.getPot())
207.                     self.__table.setPot(0)
208.                     self.__ai.setScore(1)
209.                     self.__table.setWinner('AI')
210.                     self.__table.setWon('t')
211.                     return True
212.                 elif self.__human.getAction() == 'r':
213.                     self.__human.setRaiseAmount(input("Raise by: "))
214.                     self.__human.setMoney(-self.__human.getRaiseAmount())
215.                     self.__table.setPot(self.__human.getRaiseAmount())
216.                     if self.__ai.getAction() == 'r':
217.                         self.__human.setMoney(-self.__ai.getRaiseAmount())
218.                         self.__table.setPot(self.__ai.getRaiseAmount())
219.                         self.__table.setWon('f')
220.                         return False
221.                     elif self.__ai.getAction() == 'c':
222.                         self.__table.setWon('f')
223.                         return False
224.                     else:
225.                         self.__table.setWon('f')
226.                         return False
227.                 elif self.__human.getAction() == 'c':
228.                     if self.__ai.getAction() == 'c':
229.                         self.__table.setWon('f')
230.                         return True
231.                     elif self.__ai.getAction() ==  'r':
232.                         self.__human.setMoney(-self.__ai.getRaiseAmount())
233.                         self.__table.setPot(self.__ai.getRaiseAmount())
234.                         self.__table.setWon('f')
235.                         return False
236.                     else:
237.                         self.__table.setWon('f')
238.                         return False
239.         class Table:
240.             __flop1 = []
241.             __flop2 = []
242.             __flop3 = []
243.             __turn = []
244.             __river = []
245.             __entryMoney = 100
246.             __pot = 0
247.             __won = None
248.             __winner = ''
249.             __nextPlayer = None
```

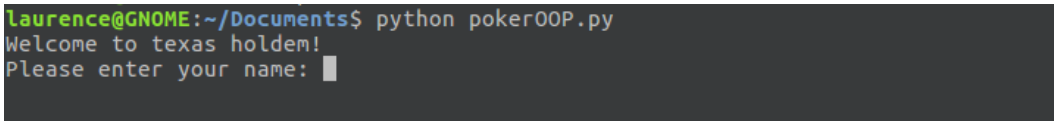
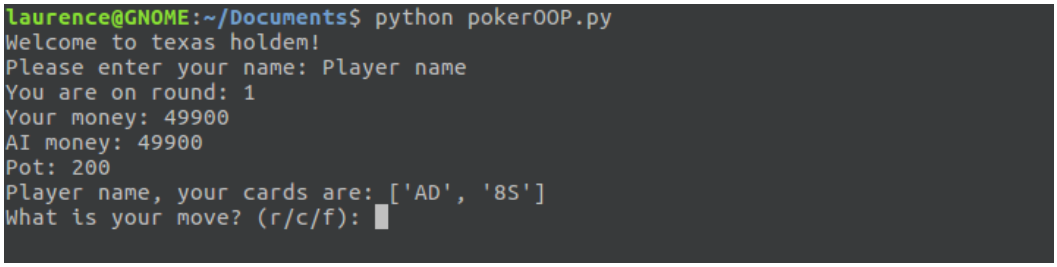
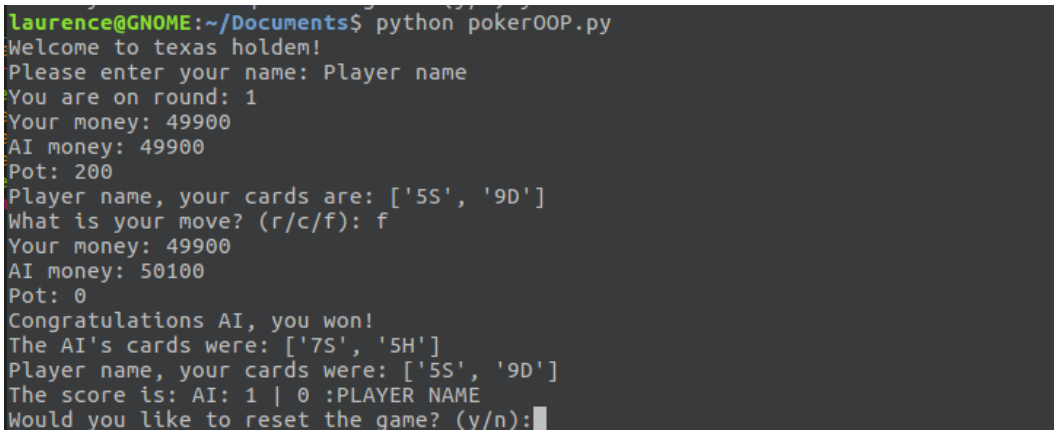
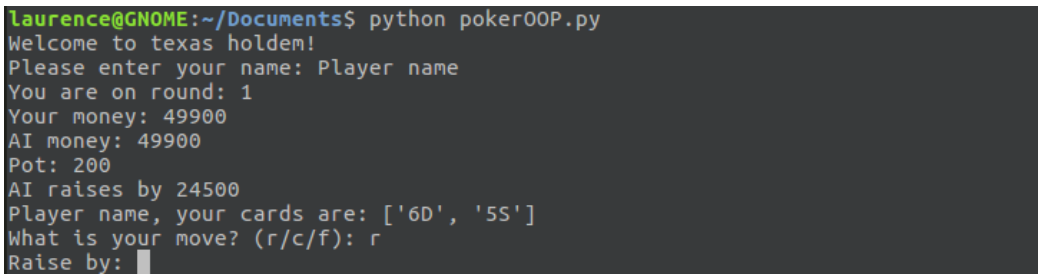
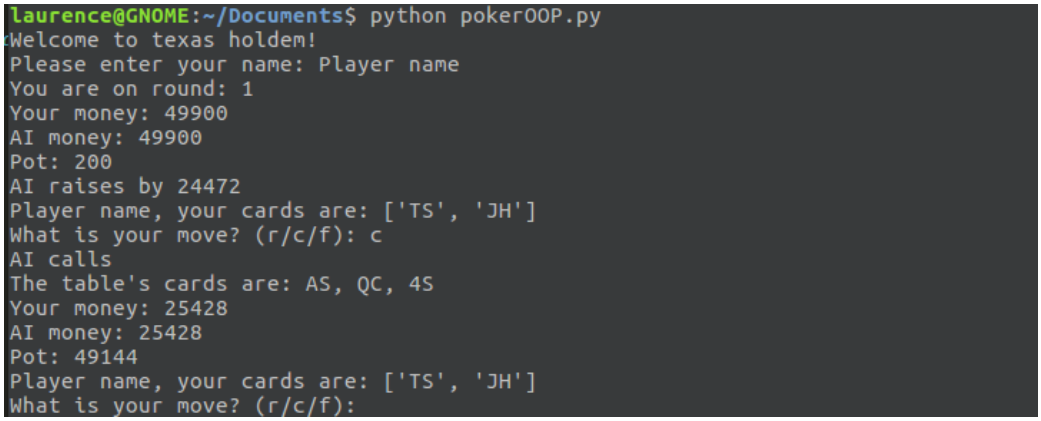
```
250.             __winnersCards = ''
251.
252.             def setCards(self):
253.                 self.__flop1 = CardPile().getCard()
254.                 self.__flop2 = CardPile().getCard()
255.                 self.__flop3 = CardPile().getCard()
256.                 self.__turn = CardPile().getCard()
257.                 self.__river = CardPile().getCard()
258.             def getCards(self, amount):
259.                 return [self.__flop1, self.__flop2, self.__flop3, self.__turn, self.__river
    ][amount]
260.             def getEntryMoney(self):
261.                 return self.__entryMoney
262.             def setPot(self, amount):
263.                 if amount == 0: self.__pot = amount
264.                 else: self.__pot += amount
265.             def getPot(self):
266.                 return self.__pot
267.             def setWinner(self,winner):
268.                 self.__winner = str(winner)
269.             def getWinner(self):
270.                 return self.__winner
271.             def setWon(self,tf):
272.                 if tf == 't': self.__won = True
273.                 elif tf == 'f': self.__won = False
274.             def getWon(self):
275.                 return self.__won
276.             def setNextPlayer(self, x):
277.                 if x == 'r':
278.                     if randint(0,100) >= 50:
279.                         self.__nextPlayer = 1
280.                     else:
281.                         self.__nextPlayer = 2
282.                 elif x == 'h':
283.                     self.__nextPlayer = 1
284.                 else:
285.                     self.__nextPlayer = 2
286.             def getNextPlayer(self):
287.                 return self.__nextPlayer
288.             def setWinnersCards(self,x):
289.                 self.__winnersCards = str(x)
290.             def getWinnersCards(self):
291.                 return self.__winnersCards
292.
293.         class Player:
294.             __card1 = []
295.             __card2 = []
296.             __money = 0
297.             __score = 0
298.             __action = ''
299.             __raiseAmount = 0
300.             __name = ''
301.
302.             def __init__(self, name):
303.                 self.__name = name
304.             def setCard(self):
305.                 self.__card1 = CardPile().getCard()
306.                 self.__card2 = CardPile().getCard()
307.             def getCard(self):
308.                 return [self.__card1, self.__card2]
309.             def setMoney(self, amount):
310.                 self.__money += amount
311.             def getMoney(self):
312.                 return self.__money
313.             def setScore(self, amount):
314.                 self.__score += amount
```

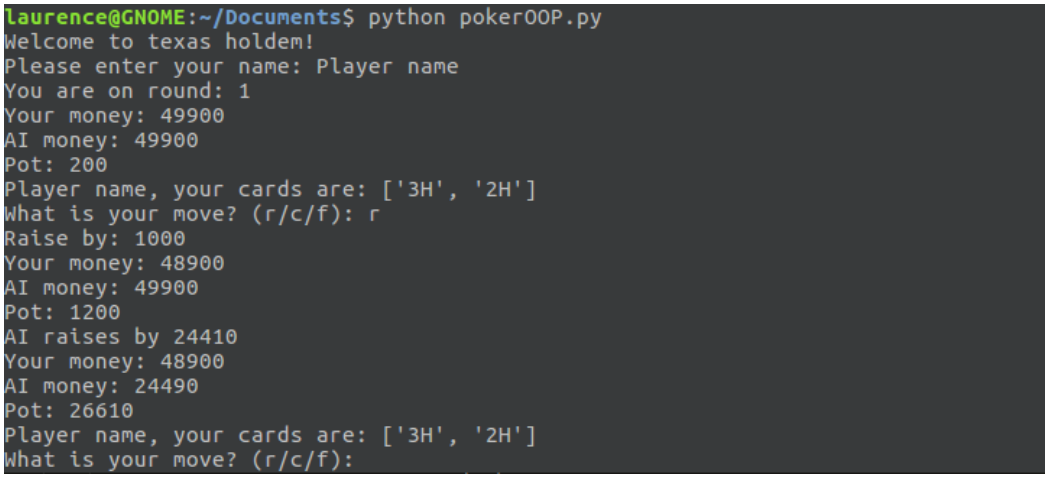
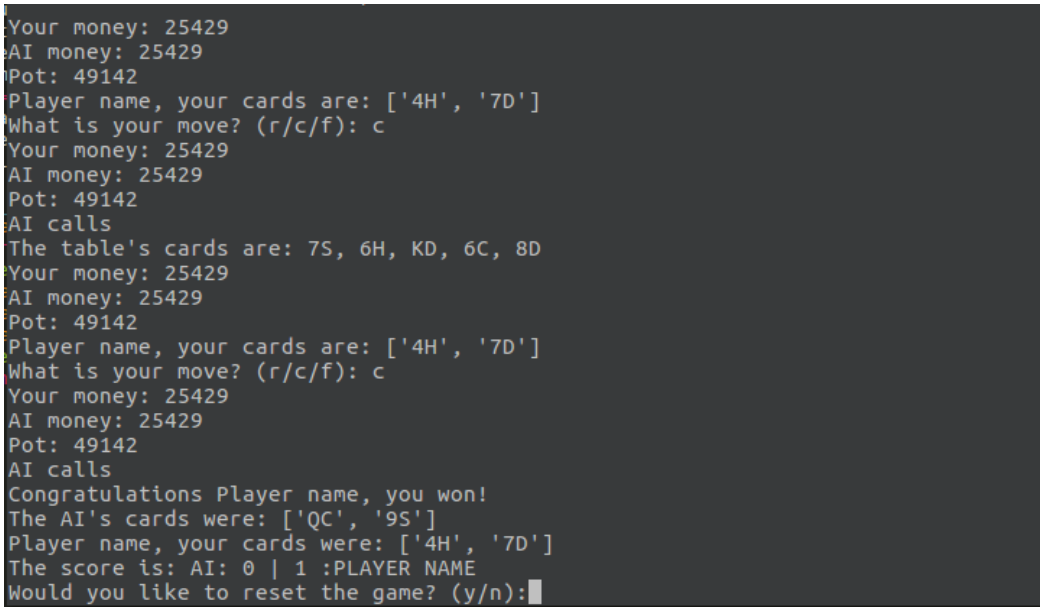
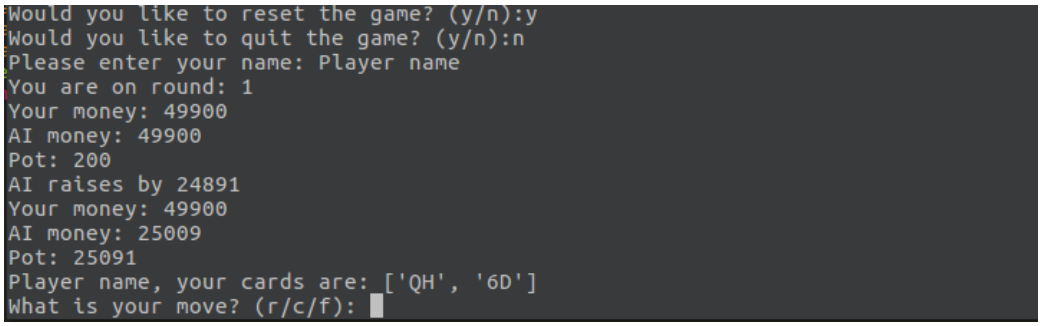
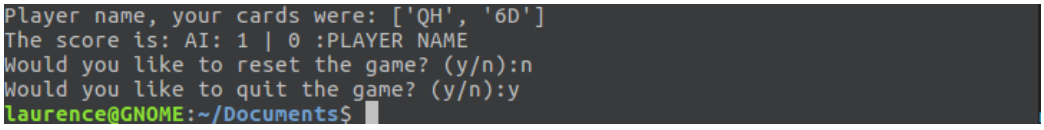
```
315.             def getScore(self):
316.                 return self.__score
317.             def setAction(self, action):
318.                 self.__action = action
319.             def getAction(self):
320.                 return self.__action
321.             def setRaiseAmount(self, amount):
322.                 self.__raiseAmount = amount
323.             def getRaiseAmount(self):
324.                 return self.__raiseAmount
325.             def setName(self, name):
326.                 self.__name = name
327.             def getName(self):
328.                 return self.__name
329.
330.         class AI(Player):
331.             #define datasets
332.             __x = np.genfromtxt('handstrained.csv', delimiter=',')
333.             __y = np.genfromtxt('correctpredtrained.csv', delimiter=',')[np.newaxis]
334.             __y = __y.T
335.             #seed
336.             np.random.seed(1)
337.             #weights
338.             __w0 = 2*np.random.random((3,4))-1
339.             __w1 = 2*np.random.random((4,4))-1
340.             __w2 = 2*np.random.random((4,4))-1
341.             __w3 = 2*np.random.random((4,1))-1
342.             #raise check
343.             __allReadyRaise = False
344.
345.             __move = 0
346.
347.             def __init__(self):
348.                 Player.__init__(self, 'AI')
349.             def setCard(self):
350.                 Player.setCard(self)
351.             def getCard(self):
352.                 return Player.getCard(self)
353.             def setMoney(self, amount):
354.                 Player.setMoney(self, amount)
355.             def getMoney(self):
356.                 return Player.getMoney(self)
357.             def setScore(self, amount):
358.                 Player.setScore(self, amount)
359.             def getScore(self):
360.                 return Player.getScore(self)
361.             def setAction(self,x):
362.                 if x == '-':
363.                    Player.setAction(self, x)
364.                 else:
365.                     Player.setAction(self, self.predict(Player.getCard(self)[0],Player.getC
    ard(self)[1]))
366.             def getAction(self):
367.                 return Player.getAction(self)
368.             def setRaiseAmount(self):
369.                     Player.setRaiseAmount(self, int(*(self.__move)*(Player.getMoney(self)/2
    ))/100)
370.             def getRaiseAmount(self):
371.                 return Player.getRaiseAmount(self)
372.             def reset(self):
373.                 self.__allReadyRaise = False
374.             #make sigma
375.             def sigma(self,x):
376.                 return 1/(1+np.exp(-x))
377.             #sigma gradient
378.             def sigmaDeriv(self,x):
```

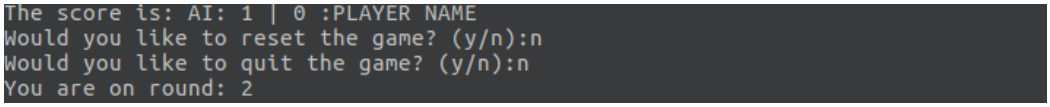
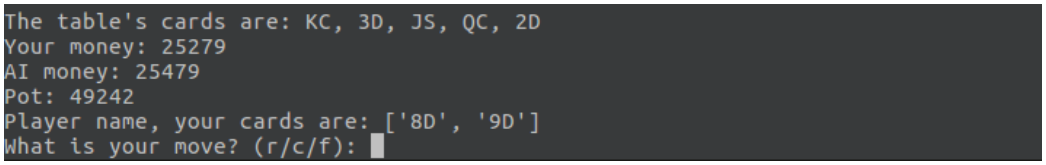
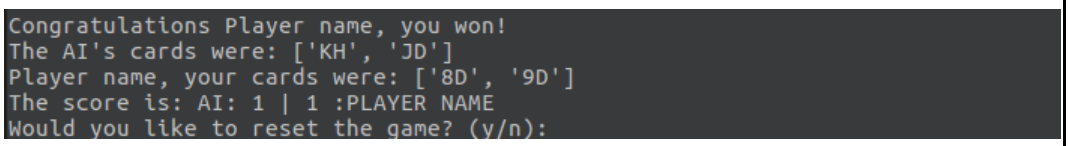
```
379.                 return x*(1-x)
380.             def predict(self,card1,card2):
381.                 __carray = self.processCards(card1,card2)
382.                 for i in xrange(10):
383.                     self.epoch()
384.                 #predict
385.                 __c1 = __carray[0]
386.                 __c2 = __carray[1]
387.                 __s = __carray[2]
388.                 __C = np.array([[__c1,__c2,__s]])
389.                 self.__move = self.sigma(np.dot(self.sigma(np.dot(self.sigma(np.dot(self.si
    gma(np.dot(__C, self.__w0)), self.__w1)), self.__w2)),self.__w3))
390. 
391.                 if self.__move >= 0.7 and self.__allReadyRaise == False:
392.                     self.__allReadyRaise = True
393.                     return 'r'
394.                 elif self.__move >=0.5: return 'c'
395.                 else: return 'f'
396.             def epoch(self):
397.                 __z=0
398.                 #train
399.                 for t in xrange(len(self.__y)/4):
400.                     #forward propagation
401.                     __l0 = self.__x[__z:(__z+5)]
402.                     __l1 = self.sigma(np.dot(__l0, self.__w0))
403.                     __l2 = self.sigma(np.dot(__l1, self.__w1))
404.                     __l3 = self.sigma(np.dot(__l2, self.__w2))
405.                     __l4 = self.sigma(np.dot(__l3, self.__w3))
406.                     #error + change calc
407.                     __l4_error = self.__y[__z:(__z+5)] - __l4
408.                     __l4_change = __l4_error*self.sigmaDeriv(__l4)
409.                     __l3_error = __l4_change.dot(self.__w3.T)
410.                     __l3_change = __l3_error * self.sigmaDeriv(__l3)
411.                     __l2_error = __l3_change.dot(self.__w2.T)
412.                     __l2_change = __l2_error * self.sigmaDeriv(__l2)
413.                     __l1_error= __l2_change.dot(self.__w1.T)
414.                     __l1_change = __l1_error * self.sigmaDeriv(__l1)
415.                     #update weights
416.                     self.__w3 += np.dot(__l3.T, __l4_change)
417.                     self.__w2 += np.dot(__l2.T, __l3_change)
418.                     self.__w1 += np.dot(__l1.T, __l2_change)
419.                     self.__w0 += np.dot(__l0.T, __l1_change)
420.                     __z += 5
421.             def processCards(self,card1,card2):
422.                 if card1[:1] == 'T': __c1 = 10
423.                 elif card1[:1] == 'J': __c1 = 11
424.                 elif card1[:1] == 'Q': __c1 = 12
425.                 elif card1[:1] == 'K': __c1 = 13
426.                 elif card1[:1] == 'A': __c1 = 14
427.                 else: __c1 = int(card1[:1])
428.                 if card2[:1] == 'T': __c2 = 10
429.                 elif card2[:1] == 'J': __c2 = 11
430.                 elif card2[:1] == 'Q': __c2 = 12
431.                 elif card2[:1] == 'K': __c2 = 13
432.                 elif card2[:1] == 'A': __c2 = 14
433.                 else: __c2 = int(card2[:1])
434.                 if card1[-1:]==card2[-1:]: __s = 1
435.                 else: __s = 0
436.                 __c1 = float(__c1)/14
437.                 __c2 = float(__c2)/14
438.                 return [__c1,__c2,__s]
439. 
440.         class Compare:
441.             __allHcards = []
442.             __allAcards = []
443.             def cardCompare(self,hc,ac,tblc):
```

```
444.                 self.__allHcards = [hc[0], hc[1], tblc[0], tblc[1], tblc[2], tblc[3], tblc[
    4]]
445.                 self.__allAcards = [ac[0], ac[1], tblc[0], tblc[1], tblc[2], tblc[3], tblc[
    4]]
446.                 if self.evaluateCard(self.__allHcards) > self.evaluateCard(self.__allAcards
    ): return 'Human'
447.                 elif self.evaluateCard(self.__allAcards) > self.evaluateCard(self.__allHcar
    ds): return 'AI'
448.                 else: return "no-one"
449.
450.             def evaluateCard(self,hand):
451.                 __groups = self.group(['--23456789TJQKA'.index(r) for r, s in hand])
452.                 __counts, __ranks = zip(*__groups)
453.                 if __ranks == (14, 5, 4, 3, 2):
454.                     __ranks = (5, 4, 3, 2, 1)
455.                 __straight = len(__ranks) == 5 and max(__ranks)-min(__ranks) == 4
456.                 __flush = len(set([s for r, s in hand])) == 1
457.                 return (
458.                     9 if (5, ) == __counts else
459.                     8 if __straight and __flush else
460.                     7 if (4, 1) == __counts else
461.                     6 if (3, 2) == __counts else
462.                     5 if __flush else
463.                     4 if __straight else
464.                     3 if (3, 1, 1) == __counts else
465.                     2 if (2, 2, 1) == __counts else
466.                     1 if (2, 1, 1, 1) == __counts else
467.                     0), __ranks
468.
469.             def group(self,items):
470.                 __groups = [(items.count(x), x) for x in set(items)]
471.                 return sorted(__groups, reverse = True)
472.
473.         class CardPile:
474.             __deck = [r+s for r in '23456789TJQKA' for s in 'SHDC']
475.             __usedCards  = []
476.             def getCard(self):
477.                 while True:
478.                     __randCard = self.__deck[randint(0,51)]
479.                     if __randCard not in self.__usedCards:
480.                         self.__usedCards.append(__randCard)
481.                         return __randCard
482.                         break
483.             def reset(self):
484.                 self.__usedCards = []
485.
486.         game = Game('One')
487.         game.menu()
```

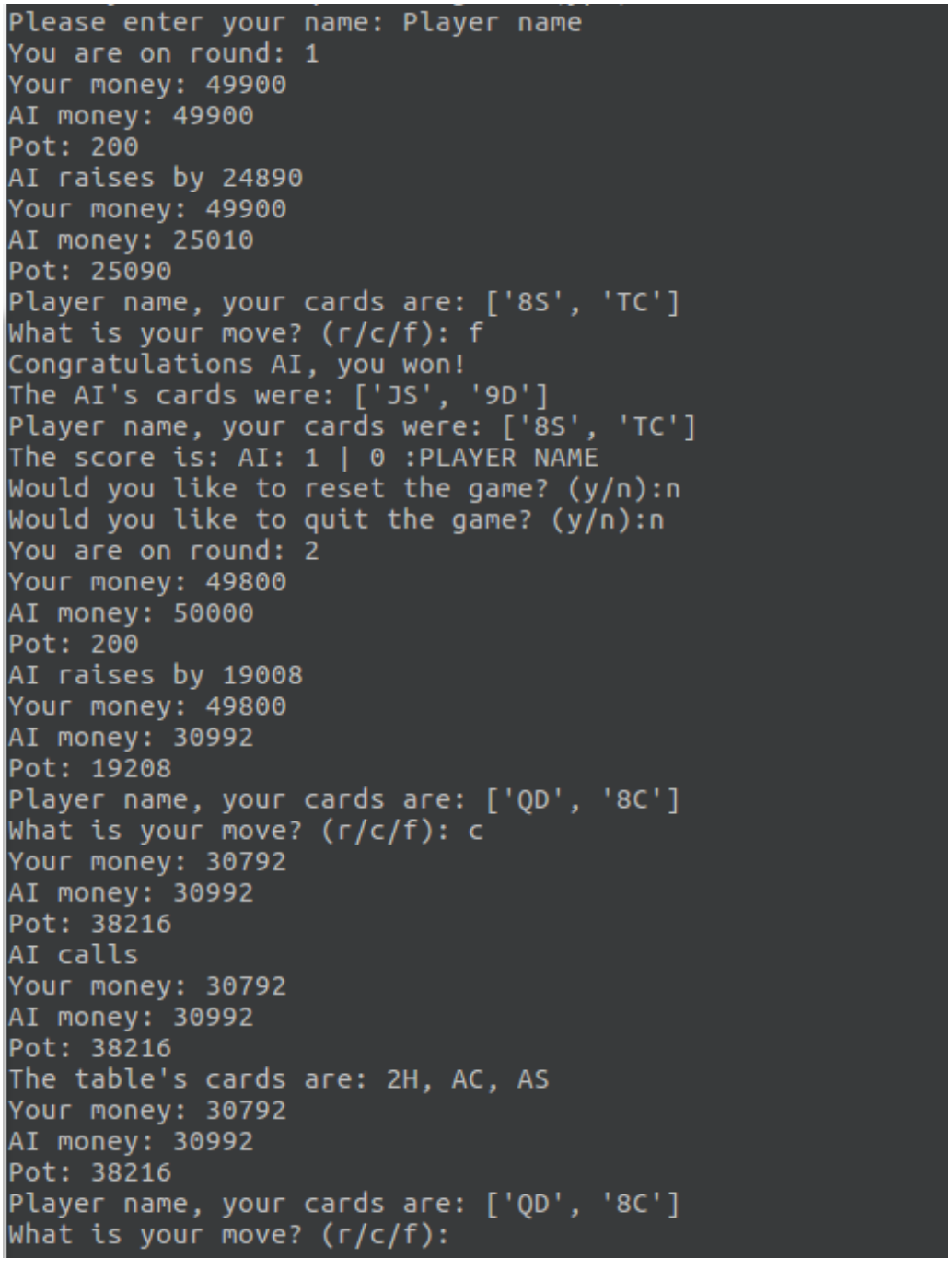
F: Testing evidence.

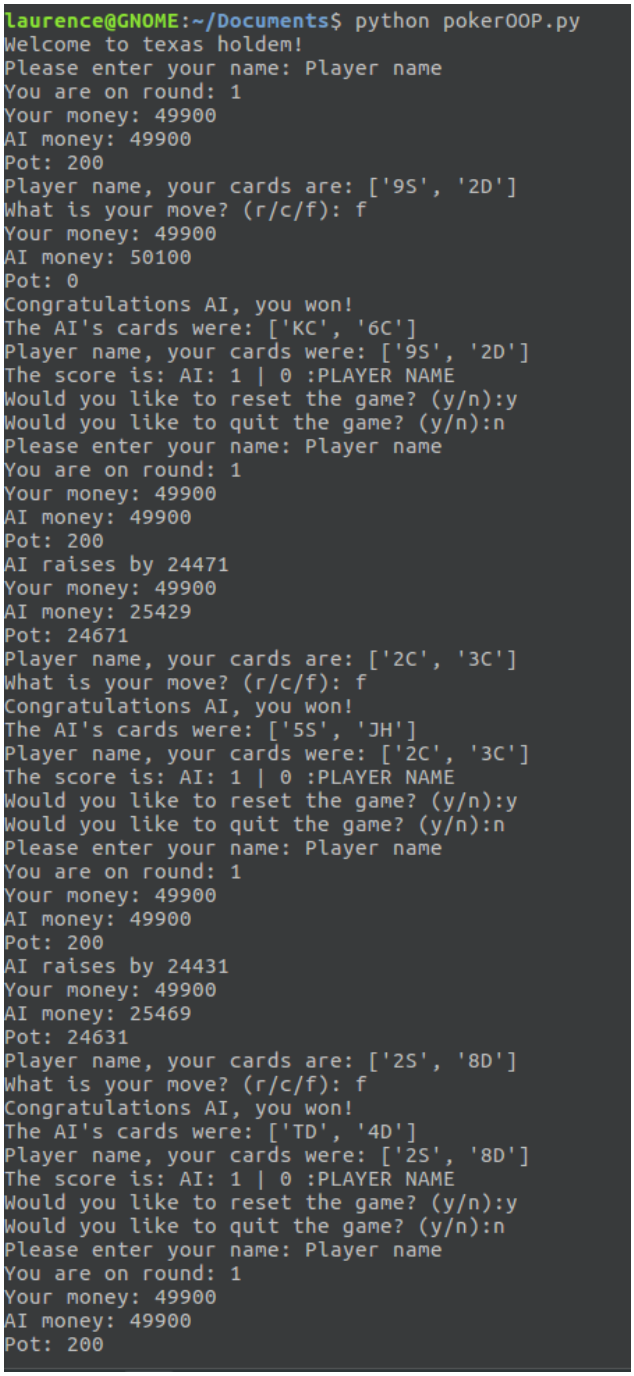
Nº	Purpose	Screenshot
1.1	The game launches without errors.	
1.2	The game successfully creates a new player object.	
1.3a	The game is able to interpret a player's move.	
1.3b	The game is able to interpret a player's move.	
1.3c	The game is able to interpret a player's move.	

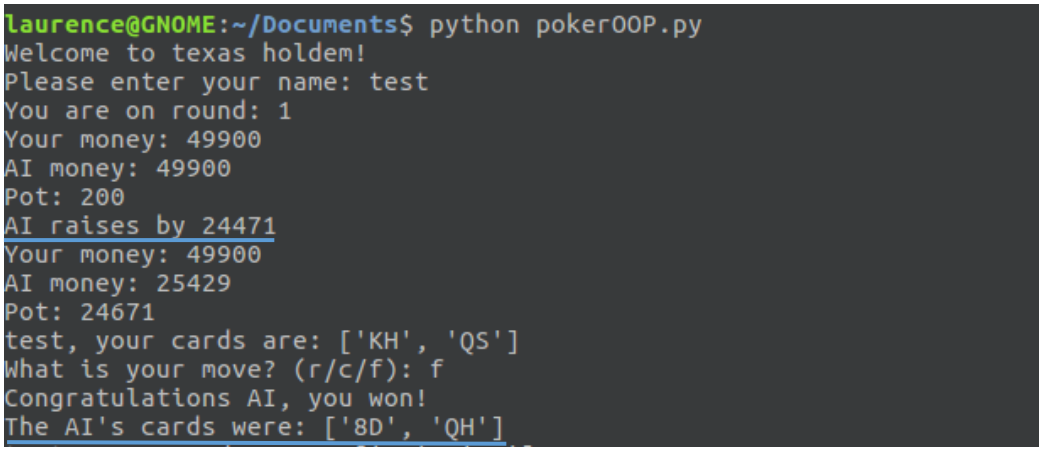
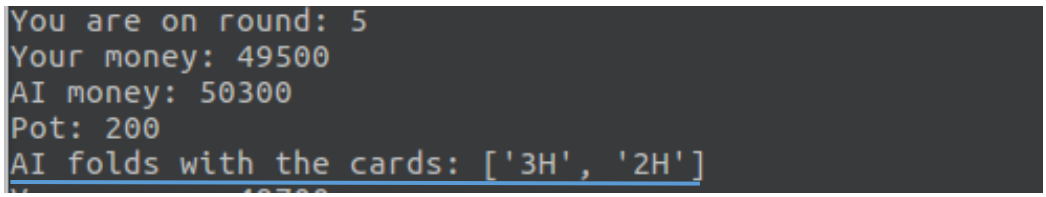
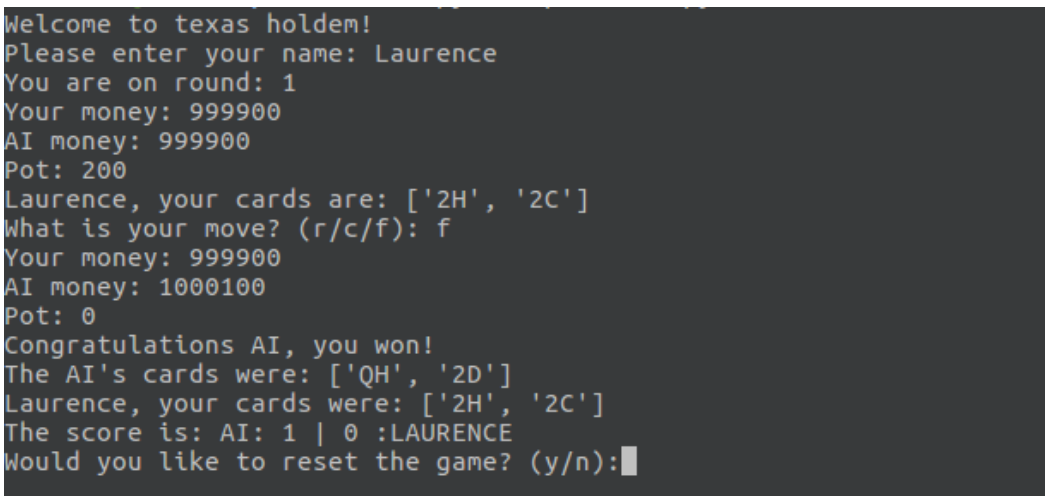
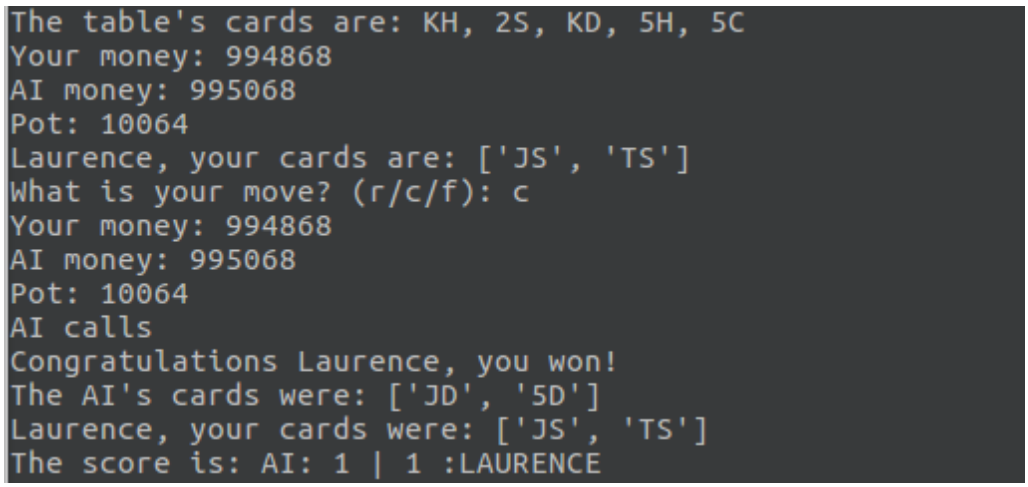
1.4	To see if the game correctly handles money.	
1.5	To see if the card comparison algorithm works correctly.	
1.6	To see if the game is able to reset itself.	
1.7	To see if the game quits successfully.	
1.8a	To see if the game displays relevant information about the game.	

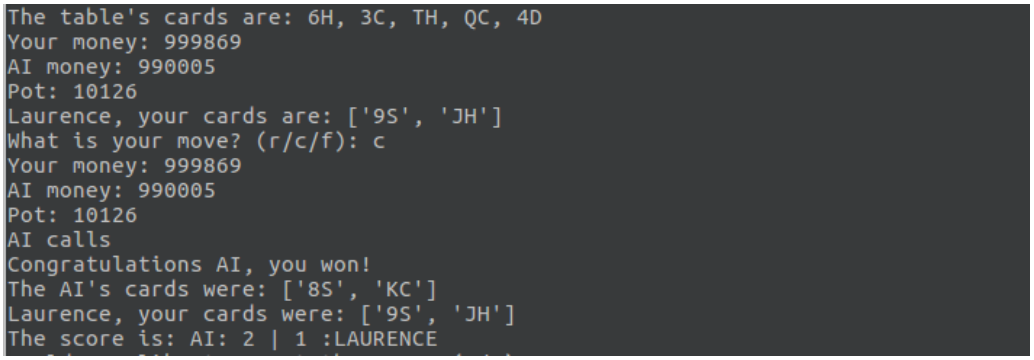
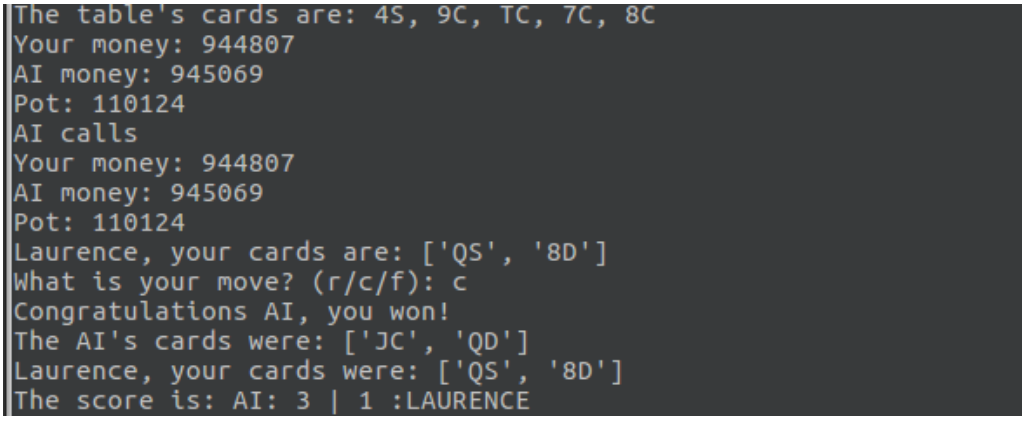
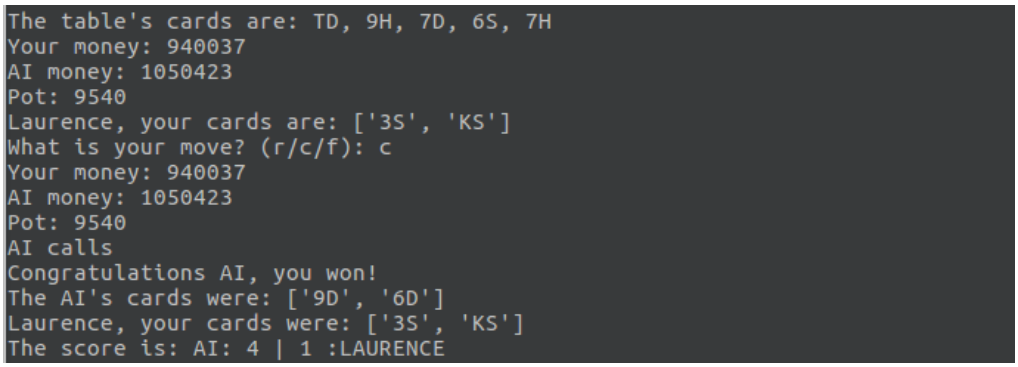
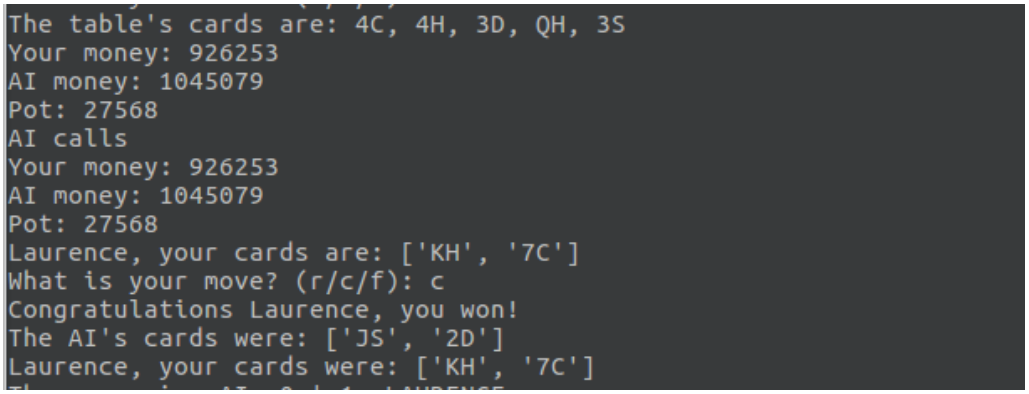
1.8b	To see if the game displays relevant information about the game.	
1.8c	To see if the game displays relevant information about the game.	
1.8d	To see if the game displays relevant information about the game.	

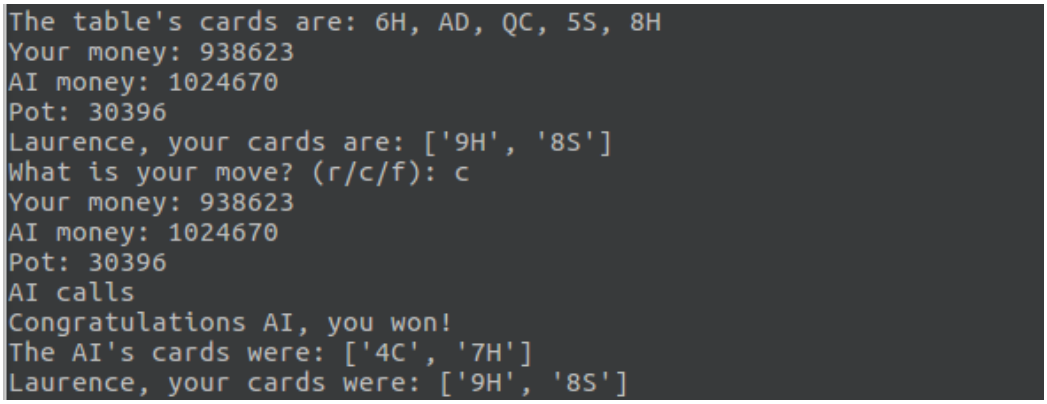
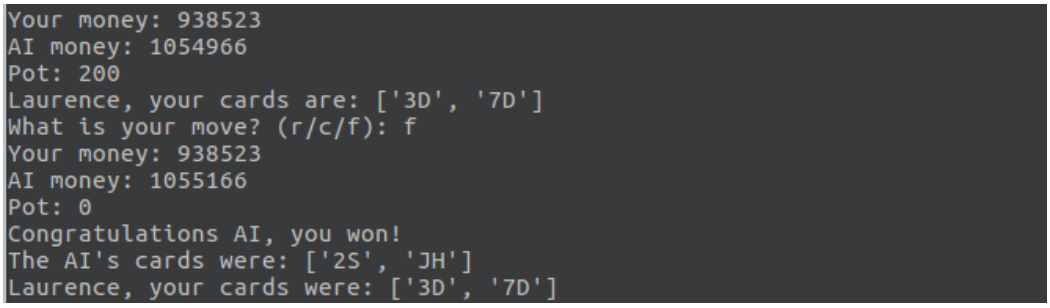
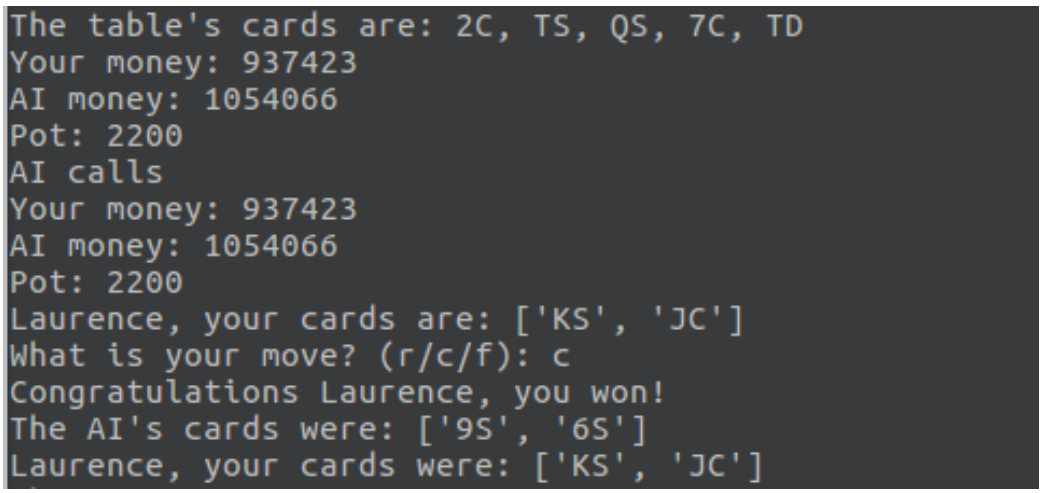
1.9	The game is able to correctly carry out an entire betting round	laurence@GNOME:~/Documents$ python pokerOOP.py Welcome to texas holdem! Please enter your name: Player name You are on round: 1 Your money: 49900 AI money: 49900 Pot: 200 AI raises by 24467 Your money: 49900 AI money: 25433 Pot: 24667 Player name, your cards are: ['2S', '9H'] What is your move? (r/c/f): c Your money: 25433 AI money: 25433 Pot: 49134 AI calls Your money: 25433 AI money: 25433 Pot: 49134 The table's cards are: JD, 2D, 7D Your money: 25433 AI money: 25433 Pot: 49134 Player name, your cards are: ['2S', '9H'] What is your move? (r/c/f): c Your money: 25433 AI money: 25433 Pot: 49134 AI calls The table's cards are: JD, 2D, 7D, 9C Your money: 25433 AI money: 25433 Pot: 49134 Player name, your cards are: ['2S', '9H'] What is your move? (r/c/f): c Your money: 25433 AI money: 25433 Pot: 49134 AI calls The table's cards are: JD, 2D, 7D, 9C, QS Your money: 25433 AI money: 25433 Pot: 49134 Player name, your cards are: ['2S', '9H'] What is your move? (r/c/f): c Your money: 25433 AI money: 25433 Pot: 49134 AI calls Congratulations AI, you won! The AI's cards were: ['TD', '5C'] Player name, your cards were: ['2S', '9H'] The score is: AI: 1 | 0 :PLAYER NAME Would you like to reset the game? (y/n):
2.1	To see if the AI can make a move within the game.	Please enter your name: Player name You are on round: 1 Your money: 49900 AI money: 49900 Pot: 200 AI raises by 24890

2.2	To see if the AI can handle the human player folding.	

2.3	To see if the AI is able to play after multiple resets.	

3.1a	To see if the AI makes the correct move according to the dataset given.	
3.1b	To see if the AI makes the correct move according to the dataset given.	
4.1	Game v AI	
4.2	Game v AI	

4.3	Game v AI	
4.4	Game v AI	
4.5	Game v AI	
4.6	Game v AI	

4.7	Game v AI	
4.8	Game v AI	
4.9	Game v AI	
4.10	Game v AI