J. Hayes O'Brien

Registration number 100246359

2022

Investigation and Simulation of Playing Optimisation and Card Counting in Blackjack

Supervised by Dr. Elena Kulinskaya

University of East Anglia
Faculty of Science
School of Computing Sciences

Abstract

In this project report, I document my investigation of the effectiveness of card counting techniques for cultivating a betting advantage against the casino in blackjack. I begin by providing an overview of how blackjack is played and discuss how card counting methods were initially developed. I then discuss my use of simulatory methods to explore the effectiveness of different methods for optimising the betting advantage of the player and the outcomes of these experiments.

Contents

List of Figures

1. Blackjack

1.1. Game Overview

Across the world there are several forms of card game deriving from the game of Twenty-one. Pontoon is a widely known variation of the game in Britian, and in the United States, there is the casino game of Blackjack. This game has been a popular method of gambling for centuries, and is likely to be the most widley known variation of it's base game Twenty-one in the world. With the 51 liscensed casinos in the las vegas strip area bringing in $3.1B of revenue each year, nearly 50% of the casino tables there are Blackjack tables (Research).

At it's core, Blackjack is quite a simple game. The goal of the player is to gain a hand of cards that make a value closer to 21 than the cards of the dealer, without their hand going "bust" by exceeding 21. Blackjack tables in Casinos have multiple seats open for many players to play against the dealer at once. Each player who wishes to take part in that round places a wager of their choice between the casino's minimum and maximum allowed betting amounts before the round begins. After all bets are placed, the round begins.

At the beginning of each round, the dealer deals two cards to each player and themself. Unlike all other cards on the table, the first card of the dealer's hand is placed down with it's face visable. Players can then chose to stand, hit, double down, or split to play out their hand.

By standing, a player choses to settle with their hand for the round. By notioning to "hit", a player opts to be dealt another card to add to their hand. By "Doubling Down", a player choses to double their wager for the hand and be dealt one card more only. By chosing to "Split", a player choses to separate their 2 cards into two different hands that now have two different wagers equal to that of the original hand wager, and a second card is dealt to each. A split can only be performed on a pair of two cards with the same face. Most casinos allow numerous splits to take place from one hand, in an act called "resplitting"; If a hand is the result of a split pair of aces, it cannot be split again. Once the player is finished with the play of their hand, only now does the dealer plays their's. Thus, the player can only win as a result of the dealer going bust if the player's hand did not happen to go bust first. If the dealer's hand makes a total value of 17 or more, they stand. If not, they hit.

Two	Three	Four	Five	Six	Seven	Eight	Nine	Ten/J/Q/K	Ace
2	3	4	5	6	7	8	9	10	11/1

Figure 1: Card Values In Blackjack

Payouts

Once a round of Blackjack has been played out, the outcome of the round determines the size of payout awarded to winning players, and losing players forfeit their bets to the house. In the event of a pay-out, (in cases other than insurance) the player keeps their original wager also.

- When a player's hand that has equal value to that of the dealer's hand, this is called a "push". No payout is awarded, but the player is able to take back their hand wager.
- When a player's hand has a value closer to 21 than the dealer's hand value and has not bust, a 1:1 payout is awarded, meaning that the payout is of equal size to the wager placed on the hand.
- If a player has a blackjack - a hand made of a Ten-value card and an Ace - and the dealer does not, a 3:2 payout is awarded, meaning that the pay-out is 1.5 times the value of the player's wager for that hand.

Insurance and Even Money

If a dealer's up card is an Ace after the initial dealing of cards for a round has taken place, there is the possibility for the dealer's hand to be a blackjack. In this eventuality, side bets are offered to the players pertaining to the outcome of this situation (maximum bets are half the player's existing wager). If a player, upon inspection of their cards possesses a blackjack themselves, they are allowed to make an "Even Money" bet. This allows the player to immediately receive a 1:1 pay-out and be finished with the round; meaning that if the dealer does have a blackjack, they are paid 1:1 where they wouldn't have been otherwise, but they could receive only two-thirds of what they could have won if the dealer did not. If the player does not possess a blackjack however, they are offered insurance. An insurance bet could award a player with a pay-out up to the size of the player's bet if the dealer did have a blackjack, and a players existing bet would be lost

to the house, resulting in no net loss or gain if the player bet half of their original wager for insurance. If the dealer did not possess a blackjack however, the player's insurance bet is lost, and the round continues Werthamer (8).

Hard Hands and Soft Hands

Hard is a term that applies to a hand in Blackjack that's composite value cannot be reduced or adjusted. As we have just noted, ace cards can be valued at either 1 or 11. So, if a hand features an ace holding a value of 11 at the time, it's composite value could be reduced by ten if necessary, but until then we use the larger of its possible totals for its value. Hands that exist in such a state of uncertainty as this are referred to as Soft until they need to be made Hard.

1.2. Betting Outcomes In Blackjack: House Advantage

Casinos make profit by ensuring that the games they provide to players result in the net loss of cash for players over time. They wish to profit from the tendancy for people to forgo a general trend towards loss in pursuit of short term wins. Over a high enough number of bets, for every x pounds wagered by the player, a fraction will be lost to the Casino. This figure, as a percentage, is called *House advantage*.

In many casino games, players wager their money upon completely random events that have no influence from contextual factors. The house edge rating of such games like Roulette, Baccarat and others remain constant and range from as low as 1.5% up to 5% (Chatter). This leads to these games being referred to as independent trials processes. What makes Blackjack stand out from these other casino games, is that this is not true for games of blackjack. Factors that influence the probability of player win for a round of blackjack are:

- the casino rule-set
- the cards dealt thoughout the round
- the decisions (strategy) of the player through out the round

In blackjack games where imperfect choices are made by casual players, the house advantage fall around the mark of 2%*. If a user were to make the best possible de-

* under normal rulesets where blackjack is paid 3:2

cisions in every scenario throughtout the game however, this house advantage rating would drop to approximately 0.5% (Young).

1.3. Blackjack Strategies

Casual players often try to integrate some simple approaches into their playing when attempting to win more games of blackjack rather than playing by personal instict alone. The player may mirror the playing choices of the dealer with a strategy called "mimic the dealer" in an attempt to attain the winning rate experienced by the dealer. Alterantively, they may try to improve their game results by utilising the "never Bust" strategy which completely negates the possibility of reaching a hand value above 21 by never choosing to hit with a hand value above 11.

The study of blackjack in academia first began in the 1950's when a group of academics from the American Journal Of Statistical Association worked on a paper titled "The Optimum Strategy in Blackjack" that would be released in 1956 (Baldwin). A professor of Mathematics from U.C Irvine called Edward O. Thorpe, who was also a member of the American Journal Of Statistical Association, was intrigued by the work of his colleagues and began investigating how a player could increase their chances of success in blackjack games. Using simulation methods he determined the effect of each possible action in all possible in-game scenarios on the probability of a player win. With the results he extracted from this, he constructed what is today known today as "blackjack basic strategy" as an arrangement of the best moves to be taken by players to minimise the advantage of the house.

As mentioned previously, the house advantage for blackjack games is reduced significantly when the player adopts the use of basic strategy rather than making in game decisions on instinct alone. So, as part of this investigation I wished to simulate the use of basic strategy and other playing strategies to understand how they operate and the effects they have on the probability of player win.

2. Initial System Build: Blackjack Card Game Simulation

To begin the build of this project, considerations of project goals had to take place. This project aims to investigate the ways in which a player can optimise their betting outcomes in blackjack. Thus, the most vital components of this system will be the accurate

simulation of blackjack games featuring these methods, and the extraction of results that can indicate the effect of such in-game variables on the games' betting outcomes. Requirement analysis documentation can be found in Appendix A of this report. As this system requirement analysis includes consideration of all system elements over the course of this project's development, only item 1 through 6 are relevant to to this use case of blackjack simulation featuring varying player strategy. After consideration of these initial requirements, it was decided that the Python programming language would be used in the development of this system. The simplicity offered in carrying out object oriented programming in Python aswell as it's suitablity and favourability for projects relating to data science make it perfect for this data gathering endevour. There are several python libraries built to equip developers with useful data analysis and data visualisation tools.

2.1. Designing A Simulation Controller

To model a real life casino session, a playing session class was created. A playing session would be started with a new player object and execute the desired number of blackjack rounds. Two lists are kept in every playing session object called roundBetsTotals and roundBetsOutcomes. For each round that is executed, the integer sum of all bets made by the player would be counted and added as an element in the roundBetsTotals list. Then, the sum of the outcomes to these bets would be counted up and stored as an element in the roundBetsOutcomes list.

Figure 2: Pseudocode: CollectBetTotals

```
1:  function COLLECTBETTOTALS(this)
2:      for it in range(this.numRounds) do
3:          round = BlackjackRound(casinoTable)
4:          betsThisRound ← 0
5:          betOutcomesThisRound ← 0                    ▷ variables are initialised
6:          if round.playerInsuranceWager != 0 then
7:              betsThisRound+ = round.playerInsuranceWager
8:              betOutcomesThisRound+ = round.playerInsuranceOutcome
9:          end if
10:         betsThisRound+ = round.playerWager
11:         betOutcomesThisRound+ = round.playersPayout
12:         this.roundBetsTotals.append(betsThisRound)
13:         this.roundBetsOutcomes.append(betOutcomesThisRound)
14:     end for
15: end function
```

Outside of the Playing Session class, playing session objects can be iteratively instantiated to build a large data set of blackjack simulations. For every playing session executed, the sum of its roundBetTotals and roundbetOutcomes lists can be stored in variables to store the total sum of all player bets over the course of all simulations, and sum of all player profits/losses. A simple percentage calculation can be done with these figures to find the player advantage over the course of these simulations.

2.2. Designing A Blackjack Round

Card Game Foundations - Card Objects, Card Collections

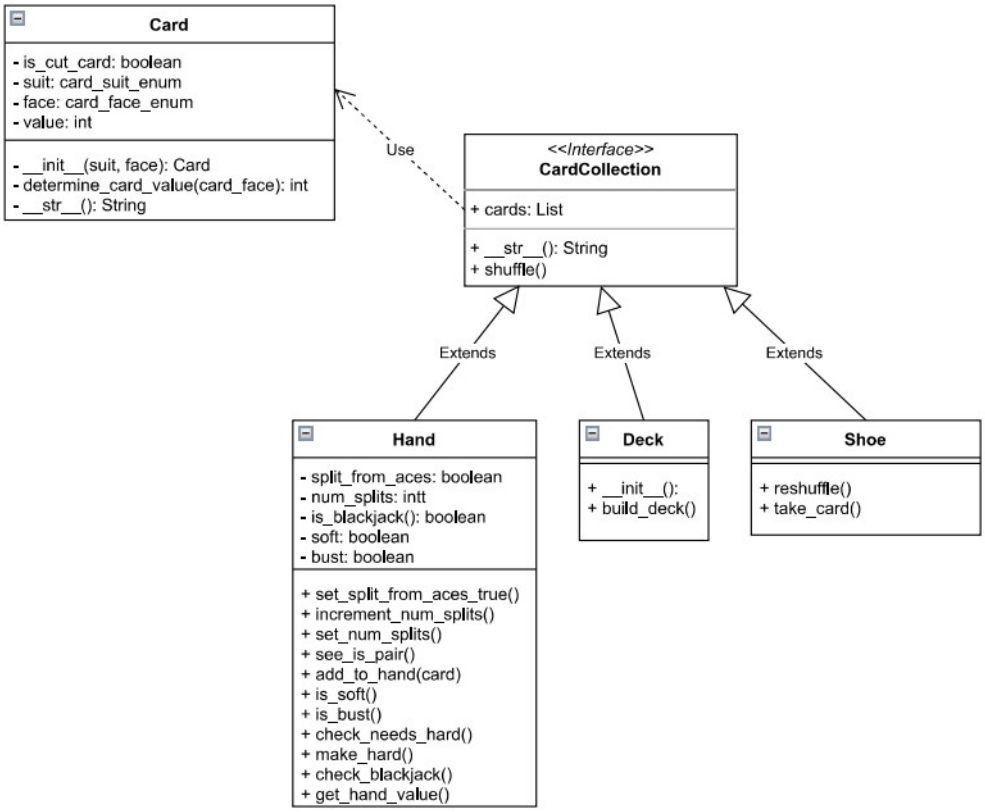
To carry out reliable and accurate simulations of Blackjack, a foundation of elements to simulate card games in general had to be designed. This began with the Card object class. Enum data types were used to represent the card attributes "Suit" and "CardFace". The attribute "cut card" was added to store a boolean data type that could flag a card object as the cut card that signals the dealer to reshuffle the shoe.

Classes for groupings of card objects then had to be designed. In simulations, players will hold their cards in a *hand*, and *decks* of cards would need to be instantiated to added

to the dealers' *shoe*. To model these in-game entities, the class interface Card Collection would be made that subclasses Hand, Deck and Shoe can inherit from.

The process of creating class diagrams would feature in the design of the simulations in this project to allow close inspection into details of entities and the reationships between them through visualisation. The act of building pseudocode was done to design how complex processes in the system can be executed and consider their effect on other entities in the system.

Figure 3: UML Diagram For Card and Card Collection Objects



A Unified Modeling Language (UML) diagram is a sequence diagram which shows the stream of communications between interacting objects. A sequence diagram consists of a set of objects represented by lifelines that exchange messages during the period of contact (IBM).

The key methods Deck buildDeck(), Shoe construct() and card collection shuffle() were planned in pseudocode before implementation.

Figure 4: Pseudocode: Deck.buildDeck

```
1: function BUILDDECK(this)  ▷ Non-static method instantiates a deck arrangement of Card objects
2:     for x in["Spades", "Clubs", "Diamonds", "Hearts"] do
3:         for y in["2", "3", "4", "5", "6", "7", "8", "9", "10", "Jack", "Queen", "King", "Ace"] do
4:             this.append(new Card(i, j))
5:         end for
6:     end for
7: end function
```

Figure 5: Pseudocode: Shoe.construct

```
1: function CONSTRUCT(this) ▷ Non-static method clears any existing cards and fills a Shoe object with a new set of cards
2:     this.cards ← []
3:     for x in range(6) do
4:         this.cards.extend(newDeck)
5:     end for
6:     this.shuffle()
7:     this.cards.insert(this.cutCardPoint, cutCard)  ▷ insert method used to place empty card object at position this.cutCardPoint in the cards list
8: end function
```

In blackjack, the dealer's shoe is reshuffled regularly. To do this, a dealer will insert a blank plastic card, called a cut card into each shoe before it's use. This ensures the regular reshuffling of a casino shoe. The depth in the shoe at which the cut card is inserted varies across casinos, but is generally between 50% and 75%. Thus, when a drawn card shows to be the cut card of a shoe, a function call to reshuffle the shoe occurs.

Initially, it was considered that when reshuffling a shoe, the cut card would be removed and a number of cards would be appended in order to bring the set of cards back to the desired length. However, in cases where the number of cards being inserted into the shoe is not an exact multiple of 52, an incomplete deck would need to be added into

the card list. With this slight change to the set of cards existing in the shoe, the house advantage of the game would consequencially be affected. While the affect on the make up of the shoe would be minor upon the first reshuffle, the entopy incurred would increase with every reshuffle of the shoe. To prevent this complication, simply all of the cards in the shoe would be replaced each shuffle.

Figure 6: Shuffle Pseudocode

```
1:  procedure PSEUDOCODE: CARD COLLECTION.SHUFFLE(this)   ▷ Non-static
    method shuffles card list order in lrgCardCollection subclass object
2:      shuffledCards ← []
3:      usedIndices ← []
4:      for i=0 to len(this.cards) do
5:          indexValid ← False
6:          complete ← False              ▷ Initialise variable holding end-condition
7:          index ← 0                                  ▷ Initialise index variable
8:          while indexValid == False do      ▷ iteratively searches for suitable index
9:              index ← random.randint(0, len(this.cards))
10:             for j = 0 to len(usedIndices) do
11:                 if usedIndices[j] == index then
12:                     indexValid ← False
13:                 else
14:                     index ← j
15:                     indexValid ← True
16:                 end if
17:             end for
18:         end while
19:         shuffledCards.append(this.cards[index])
20:         usedIndices.append(index)
21:     end for
22:     this.cards ← shuffledCards        ▷ cards attribute updated with shuffled list
23: end procedure
```

Blackjack Game Elements

In building a blackjack round, a number of object classes were designed to carry out

elements of the simulation. These consist of:

- A blackjack table class
- A blackjack round class
- A dealer object class holding a hand of cards as an attribute.
- A player object with class attributes for the current hand, bank roll and the size of the minimum casino bet; aswell as methods that use conditional logic to return decisions when needed during blackjack rounds.

2.2.1. Table

The table object holds the player, dealer, and casino shoe objects as attributes. Any action that requires access to two or more of these class attributes is performed using table class methods, such as: dealCardToPlayer or dealCardToDealer. At first it seemed logical for the dealer class objects to hold the shoe as an attribute. What was later realised however, is that this makes incurs difficulty if a player entity needs to be notified of the cards being dealt from the shoe in real time. To prevent this design issue from incurring difficulty in development, the simple approach to hold the shoe in the same local scope as the player and dealer objects was taken.

2.2.2. Dealer

A dealer object class simply holds a set of cards in a hand object as it's sole attribute. Rather than having different attributes for the intial up card and hole card, the first card is accessed in blackjack round and passed to player decision methods, and the second card is not. One class method contains the simple conditional logic for the playing out of a dealers hand. The dealer's hand is evaluated and returns a string indicating the desired response.

2.2.3. Player

The class interface player is defined as a super class for all forms of player entity to inherit methods and attributes from. It holds one hand and one splitHands object to hold the player's hand data to be acted on over the course of games. Numerous methods are defined to handle the operation of player betting.

Make-Choice Functions

A method called *makeChoice* is outlined in this class to act as each player's decision making engine. Information about the dealer's upcard is passed in a function parameter, and using knowledge of the player's own hand or split hand data, conditional logic is applied to return a string value illustrating whether the player should *stand*, *hit*, *double down*, or *split*. Responses from makeChoice will be called as many times as is necessary for the player to finish the playing out of their hand.

In the player strategies Mimic The Dealer and Never Bust, aswell as the dealer's "strategy" of play; the conditional logic applied is very simple. Below a certain value a response to "hit" is given, and a response of "stand" is given for all values above this. In Mimic The Dealer and Dealer Strategy, "stand" is returned for all hand values 17 and above (soft 17 as noted later in section 2.5). For Never Bust, "stand" is returned for all hand values 12 and above.

Figure 7: Pseudocode: straightforward makeChoice() functions

```
1: function MAKECHOICE(this, dealersUpCard)
2:     if this.hand.getHandValue >= setValue then
3:         return "std"
4:     else
5:         return "h"
6:     end if
7: end function
```

The conditional logic to perform Basic Strategy is more complex. The distribution of the responses "hit", "stand", "double down" and "split" across the array of possible card combinations cannot be summed up with a simple line of logic unlike the previous two strategies. A resource provided by the site blackjack apprenticeship shown in figure 9 sums up the actions that should be taken by the player in the case of each possible combination of cards for soft hands, hard hands and pairs. To program this logic in a system, this approach of dividing hands catagorically must be taken before the correct return value can be computed. To outline in the system the correct return values associated with each possible hand of each catagory, dictionary-based look-up tables would be used.

A lookup table is a data structure where upon beingg queried with a "key" element,

returns a "value" element that is associated with this key. To access the response lookup tables for data structures a function will be created for pair splitting, hard hands and soft hands. The two functions for hard hands and soft hands will perform a query of their respective lookup tables and respond with the correct in-game response. The function regarding the splitting of pairs will simply respond with a boolean indicating whether or not the pair should be split.

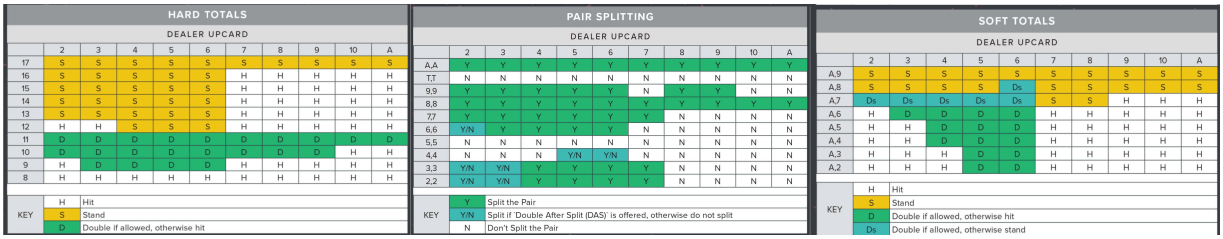


Figure 8: Resource For Basic Strategy Instructions

Figure 9: Pseudocode: BasicStrategyPlayer.makeChoice()

```
1:  function MAKECHOICE(this, dealersUpCard)
2:      if this.hand.isPair == True then
3:          if pairSplitting(this.hand.cards[0.face], dealersUpCard) == True then
4:              return "split"
5:          else
6:              hardHands(this.hand, dealersUpCard)
7:          end if
8:      else if this.hand.soft == True then
9:          return softHands(this.hand, dealersUpCard)
10:     else
11:         hardHands(this.hand, dealersUpCard)
```

The List Tree Data Structure

During the development of initial designs, it became clear that the initially proposed use of a single-dimension list would not be suitable to store the multiple hand objects of a player who split their hand. Before a split occurs, a player simply holds one hand object in the attribute *hand*. After splitting, mulitple hand objects need to be stored in the splitHands attribute of the player. As addressed later in this report (seen in figure

XX), the execution of a player object's returned decisions when playing out a hand is done recursively. When decisions are enacted for one of a player's split hands, an index needs to be passed to identify the relevant hand in the function call. When an initial split of the player hand is performed, the indexes of both of these hand objects are passed into recursive function calls (as seen in line x-x of figure y).

If the split hands attribute took the form of a unidimensional list, and the first recursive call performed another split on the child hand before standing on both child hands: the split hands index passed to the second recursive call would now point to the wrong element. For this use case, a data structure whose elements would not be deindexed upon the addition of elements before it would be needed.

To do this, I intergrated the use of a Trie-like data structure with the functionality to add multiple dimensions to the splitHands list into the project solution. Unlike a Trie, the root index [0] does not exist in this custom data structure. Upon it's instantiation, the listTrie simply holds an empty list. When the first elements are appended after a player's initial split, the split hands are stored at indexes [0] and [1] of the list. If the hand at position [0] needed to be split once again, it's two child-hands would be stored in a new list and this new list would be placed at position [0] - such that these new split hands could be accessed at child nodes [0, 1] and [0, 2].

In line with good program design practices, this data structure is designed to be generic such that it can be used in the storage and manipulation of various data types in this project and others. Python's dynamically typed variables allow class methods to be applied in the use of various data types, as opposed to only being able to be applied to elements of the static data type that was specified in a statically typed language such as Java. This is another reason why the Python programming language was chosen as a technology to utilise in this project.

A function to facilitate the addition of elements to the listTree is called ***addChildrenToTree()***. It will take 3 arguements being: the first being the index of the element in the data structure that 2 children elements are desired to overwrite, and the second and third being the two objects to be stored in the left and right branches.

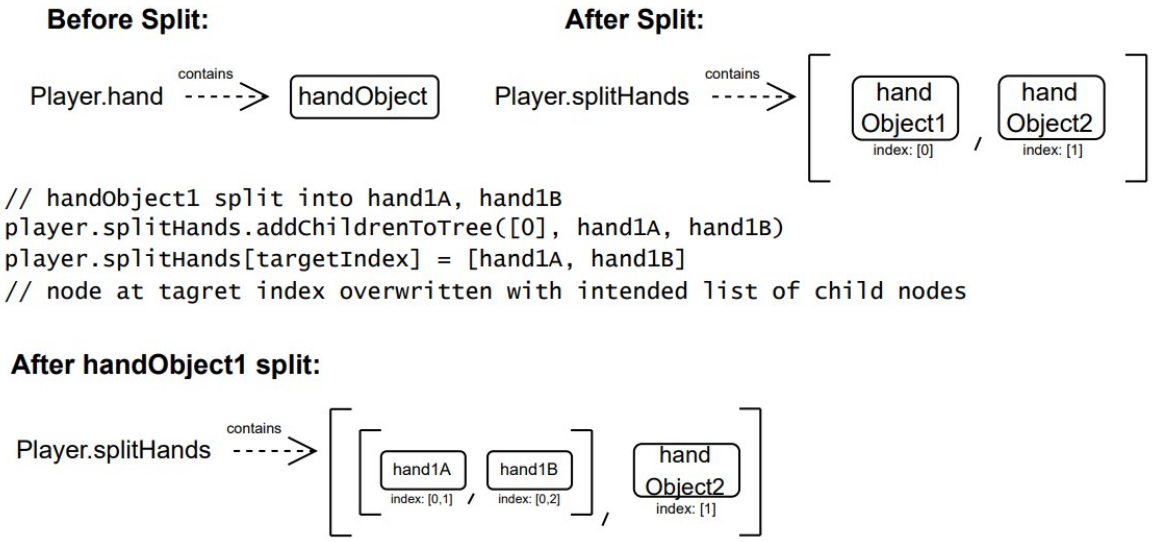


Figure 10: List Structure Visualisation

In order for the contents of listTree objects to feature as a functional part of project programming, the contents of listTree objects will need to be able to be addressed and accessed through specialised accessor methods. The standard syntax for accessing elements of a list in Python take an integer positional index to retrieve an item from the given index in the list. For listTree objects, the existing list class' __getitem__ and __setitem__ operator functions will need to be overridden. With the orgasnisation of list objects to form a structure of multiple dimensions, the positional indexes of listTree elements taken by the object's accessor methods will need to be provided as a list with a variable number of dimensions for the accessor method to be able to identify the correct list node.

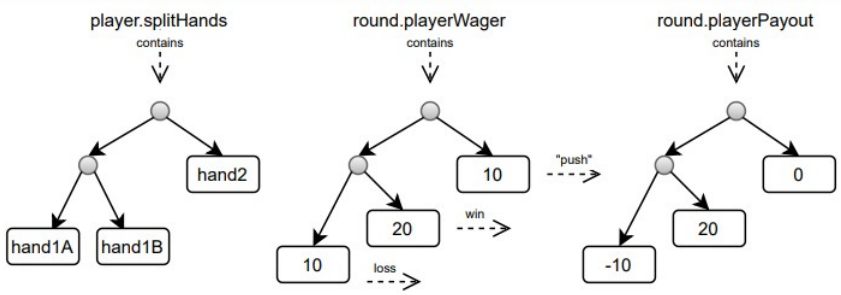
Figure 11: Pseudocode: ListTree.__getitem__

```
1: function __GETITEM__(this, key)
2:     if len(key) == 1 then
3:         return this.objects[key[0]]
4:     else if len(key) == 2 then
5:         return this.objects[key[0]][key[1]]
6:     else if len(key) == 3 then
7:         return this.objects[key[0]][key[1]][key[2]]
8:     end if
9: end function=0
```

Figure 12: Pseudocode: ListTree.__setitem__

```
1: function __SETITEM__(this, key, value)
2:     if len(key) == 1 then
3:         this.objects[key[0]] ← value
4:     else if len(key) == 2 then
5:         this.objects[key[0]][key[1]] ← value
6:     else if len(key) == 3 then
7:         this.objects[key[0]][key[1]][key[2]] ← value
8:     end if
9: end function
```

For a player object where the splitting of a hand has been opted for, the bets placed upon each hand need to be stored in a listTree. The changes to the player wager data can be performed alongside the changes to player splitHands via the addChildrenToTree() method, but when the conclusion of a round is reached, the outcomes of the bets for each player hand need to then be calculated and stored in a way where the split hand that each betting outcome is associated with is clear. It was decided that the betting outcomes will be stored in a listTree that mirrors the structure of the existing player splitHands data. As shown in the graphic below, the data structure held in round.playerPayout is a listTree that would have been instantiated to mirror existing listTrees held in player.splitHands and round.playerWager.



A significant amount of effort was put in to design algorithms to facilitate this. The first item of functionality that would be needed to do this is a function that can return a sorted list of terminal node indexes for a given trie that needs to be replicated. To do this a depth first traversal of the listTree object will be performed before the resultant list of indexes is sorted using a bubble sort. Pseudocode for this proposed implementation of depth first traversal can be seen in the figure below - while pseudocode for this integration of bubble sort was not deemed important enough to include in this report.

```
1:  function TRIETRAVERSAL(this, args)
2:      if len(args) == 0 then                                  ▷ start traversal
3:          nodesToVisit = [[0], [1]]
4:          return this.trieTraversal([0], nodesToVisit)
5:      else
6:          if len(args) == 3 then
7:              terminalNodeIndexes = args[2]
8:          else
9:              terminalNodeIndexes = []
10:         end if
11:         args[1].remove(args[0])                          ▷ node marked as visited
12:         if type(this[args[0]]) == list then
13:             for x in [0, 1] do
14:                 args[1].append(args[0] + [x])
15:             end for
16:             return self.trieTraversal(args[1][0], args[1], terminalNodeIndexes)
17:         else
18:             terminalNodeIndexes.append(args[0])
19:             if len(args[1]) == 0 then
20:                 return terminalNodeIndexes       ▷ return list - no nodes left to visit
21:             else
22:                 return self.trieTraversal(args[1][0], args[1], terminalNodeIndexes)
```

Figure 13: Psuedocode: TrieTraversal()

To facilitate this, an alternate constructor will be implemented. If no arguments are passed to the constructor, a standard empty listTree will be instantiated. Alternatively, arguments can be passed to the constructor to build a listTree that stores data in a predefined structure. In order to describe the structure that the listTree follow, a list of indexes where the terminal nodes are located would need to be passed as the first argument. The second argument would then need to hold a list containing the data that needs to be held at each node.

```
function __INIT__(this, args)
    if args == 0 then
        this.objects = []                                   ▷ standard instantiation path
    else
        this.objects = []
        for i in range(len(args[0])) do
            subscript = []
            for n in range(len(args[0][i])) do
                subscript.append(args[0][i][n])
                Try:
                    if type(this[subscript[:n + 1]]) is list or int: then
                        pass
                    else
                        raise indexError
                Except indexError:
                    if subscript = [0] then
                        if args[0][i] == 0 and args[0][-1] != 1 then
                            this.objects.extend([args[1][i], []])
                        else if args[0][i] == 0 and args[0][-1] == 1 then
                            this.objects.extend([args[1][i], args[1][-1]])
                        else if args[0][i] != 0 and args[0][-1] == 1 then
                            this.objects.extend([[], args[1][-1]])
                        else
                            this.objects.extend([[], []])
                    else
                        ammendedSubscript = subscript[:-1]
                        ammendedSubscript.append(1)
                        if n = len(args[0][i]) - 1 then
                            if subscript[n] == 0 then
                                if args[0][i+1] == ammendedSubscript then
                                    this[subcript[:n]] = [args[1][i], args[1][i + 1]]
                                else
                                    this[subcript[:n]] = [args[1][i], []
                            else
                                if len(args[0][i+2]) == len(args[0]) - 1 then
                                    this[subscript[:n]] = [[], args[1][i + 2]]
                                else
                                    this[subscript[:n]] = [[], []]
```

Figure 14: Psuedocode: listTree.__init__()

The algorithm (as displayed above) will start in the first else branch after it is ruled

out that a standard listTree initialisation is not required. After an empty list is assigned to the object, an iteration will take place over a range equal to the number of terminal node indexes passed in a list into args[0]. For each of the terminal node indexes, starting with the left-most node, a subscript begins to be built. With each terminal node index being provided as a list of integers, these integers are added one by one to the variable "subscript". With each addition, a try/except block will be used to see if a node exists at the index referenced by the current subscript. If nodes do not exist, they will be added to the data structure one by one until a terminal node index is reached. When a terminal node index is reached, the object from the corresponding index in the second list argument is stored at this index.

2.2.4. Round: The Playing Out Of A Round

The blackjack round class holds a blackjack table entity as an attribute to be accessed in order to orchestrate the playing out of a blackjack round. Other class attributes contain key information regarding the events taking place in this game round, such as bets made by the player and the outcomes of these bets.

The following class methods will be written in the round class to conduct the playing out of a round:

- collectWagers()
- dealCards()
- insurance()
- playerBlackjack()
- playerPlayHand()
- playerTakeAction()
- dealerPlayHand()
- dealerTakeAction()

Recursion is used in this design to conduct the playing of of the players, and the dealers hands. A "takeAction()" method will be called and passed the value of a player or dealers' decision. Based on the action being enacted, this function may perform this

process recurively until hands are fully played out. Aswell as being passed the player's decision that needs to be excecute, in case where the player has split their hand, this function will also take an additional argument of a listTree index pointing to the hand of a player for the action to be excecuted on.

Figure 15: Pseudocode: playerPlayHand

```
1:  function PLAYERPLAYHAND(this, response, args)
2:      if args == 0 then
3:          if response == "stand" then
4:              exit
5:          else if response == "hit" then
6:              this.table.dealCardToPlayer()
7:              if not this.table.player.hand.isBust() then
8:                  choice ← this.table.player.makeChoice(DealerUpCard)
9:                  this.playerTakeAction(choice)
10:             end if
11:         else if response == "double down" then
12:             this.playerWager ← this.playerWager ∗ 2
13:             this.table.dealCardToPlayer()
14:         else if response == "split" then
15:             hand1 ← newHand(this.table.player.hand.cards[0])
16:             hand2 ← newHand(this.table.player.hand.cards[1])
17:             this.table.player.splitHands.addChildrenToTree([], hand1, hand2)
18:             this.table.player.dealCardToPlayer([0])
19:             this.table.player.dealCardToPlayer([1])
20:             if this.table.player.hand.cards[0].face == CardFace.ACE then
21:                 this.table.player.splitHands[[0]].setSplitFromAcesTrue()
22:                 this.table.player.splitHands[[1]].setSplitFromAcesTrue()
23:             end if
24:             self.table.player.splitHands[[0]].incrementNumSplits()
25:             self.table.player.splitHands[[1]].incrementNumSplits()
26:             wager = this.playerWager
27:             this.playerWager ← ListTree()
28:             this.playerWager.addChildrenToTree([], wager, wager)          ▷ player
    wagers stored in corresponding listTree
29:             this.playerTakeAction(this.table.player.makeChoice(this.table.dealer.hand.cards[0], [0])
30:             this.playerTakeAction(this.table.player.makeChoice(this.table.dealer.hand.cards[0], [1])
31:         end if
32:     else    ▷ function applied in same way now with addressing of split hand index
```

Final function calls in "split" branch of code pass indexes [0] and [1] for each of the players splitHands. The "else" branch of the function is not shown in pseudocode due to excessive psuedocode length. The same branches will be followed with player hands being addressed with listTree indexes.

2.2.5. Round: Round Conclusion

When the players and the dealers hands have been fully played out, the round outcomes need to be determined with the calculation of wager outcomes. These wager outcomes will be stored in the round class attributes playerInsuranceOutcome, playerEvenMoney-Outcome and playerPayout.

When a round has concluded, the roundComplete boolean will be set to True, and it's attributes can be accessed to attain necessary information about the results of the round simulation.

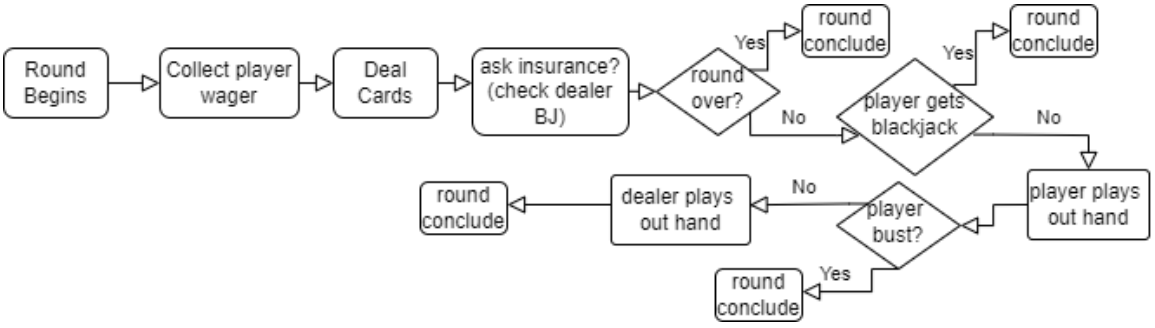
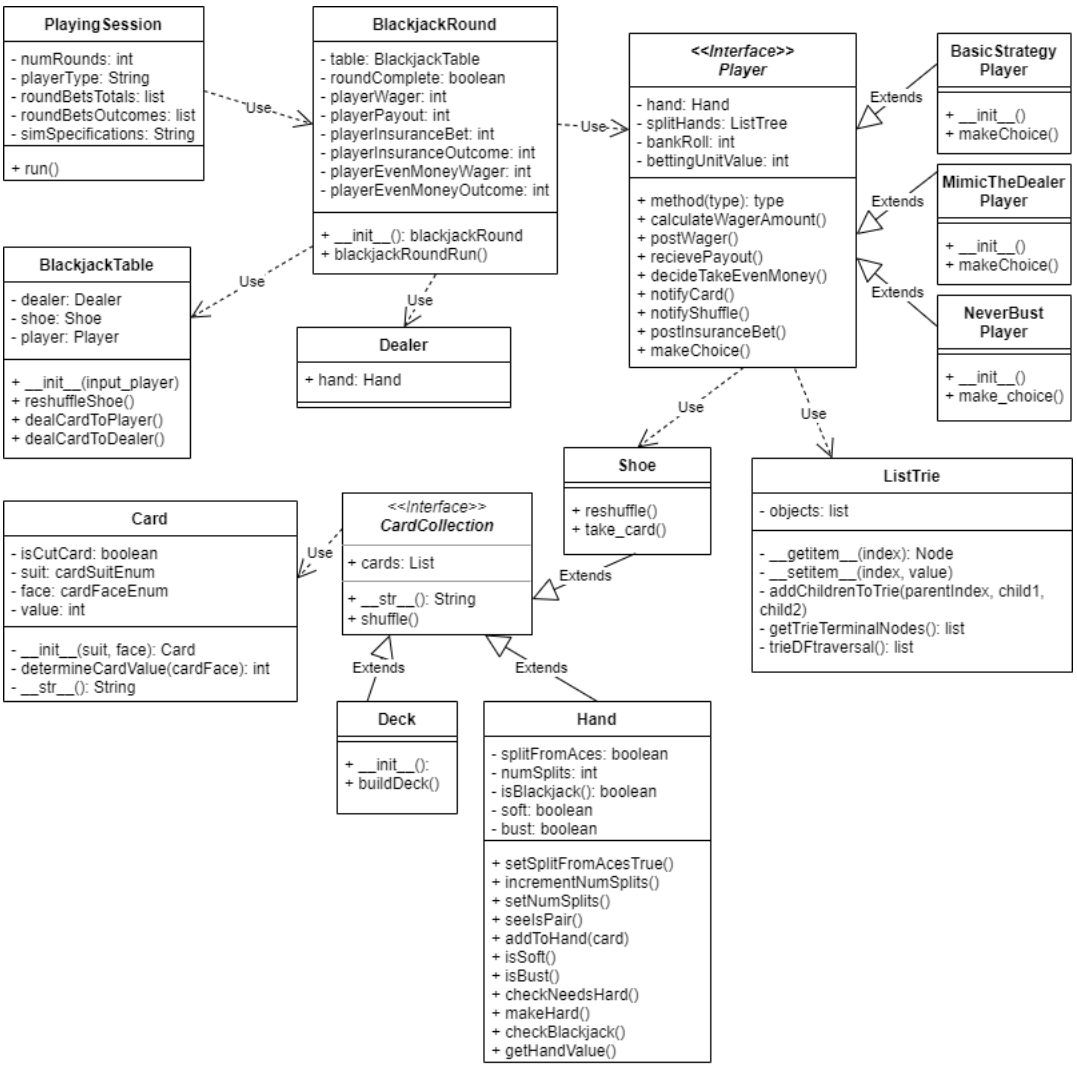


Figure 16: Blackjack Round Program Flow

2.3. System UML Visualisation



2.4. Simulation Conditions

With the house adantage of blackjack varying greatly based on the specific rulesets which are applied, the rules applied is am important set ofvariables to keep consistent across simulations to maintain accurate results.

2.4.1. Variance across Blackjack Rules

Pay-outs

With different casinos offering blackjack games featuring different sets of rules, many of these deviations from the standard ruleset of blackjack increase the games' house advantage. While the printed words "blackjack pays 3:2" are a classic identifying feature of most blackjack tables, some casinos actually implement a lower payout of 6:5 for winning by blackjack. While the house advantage of single-deck blackjack games where perfect stategy is applied is said to be 0.20%, the reduction of blackjack payout size increases this number to 1.6% (Zender).

Deck Counts

Nowadays, the dealer in Blackjack deals cards from a Shoe. This wasn't always the case however, and for much of Blackjacks' history a single deck of cards was used. When knowledge of strategy optimisation and card counting became widespread in 1960, casinos added more decks of cards to the dealers' shoe to make it more difficult for players to take advantage and increasing theirs bet when the favourability of the deck was high (Woodshut). Now it is common for 4, 6, 8 or even 12 decks of cards to be shuffled into dealer's shoe. The greater the number of decks used to form a shoe, the less certainty of what cards are likely to be dealt next can be gained. This unfortunate fact decreases the profitiability of card counting as the number of decks used increases (3).

Another factor that effects house advantage and the profitability of card counting is deck penetration. As the number of cards in the dealer's shoe decreases, a card counter is more able to accurately say how likely it is for certain cards to be dealt next. To prevent card counters from attaining high levels of certainty in their predictions of the deck and claiming great profits, a cut card is inserted somewhere between 50% and 80% of the way through the deck. When the dealer reaches this plastic card when dealing, the shoe is reshuffled.

Variation In Blackjack Rules

While it is common for a player to be able split their hand up to a limit of 4 hands, this limit varies much across different casinos (8). It is also common for casinos to allow a player to double down on a hand which has been split, but this is not always the case.

2.4.2. Simulation Ruleset

In this simulation the set of rules has been standardised to emulate a casino with rules that are reasonably kind to customers. Care was taken to ensure that the results seen in these simulations are not unrealistic or likely to overstate the merrits to methods of playing optimisation. Meanwhile these test conditions were selected in the program design to also not completely impede the visibility of positive results from player strategy optimisation.

- Shoes are constructed of 6 decks
- Blackjack pays 3:2
- Deck penetration is 75%
- Double after split allowed
- Double of split aces not allowed
- Hands are allowed to be split up to 3 times from the root hand
- Dealer hits on soft 17

2.5. Testing and Evaluation Of Initial Simulation

When programming of this initial system build reaches it's conclusion, testing must be done to ensure correct functionality. When the build of this initial system was complete, program testing had to be done to ensure it's functionality. Branch testing was used to validate simulation results. It was evaluated that the most important component of the system to be tested was the execution of blackjack basic strategy via the *makeChoice()* in BasicStrategyPlayer.py.

The test harness written to validate the performance of this module evaluated the correctness of responses returned by the method with different player hand compositions and up cards held by the dealer. In order to build correct testing conditions, elements used by *makeChoice()* in BasicStrategyPlayer.py such as Hand.get_hand_value() or Hand.is_soft() could not be used to evaluate the correct responses for hands, as then any programatic errors in these functions would cause the same errors in the test condition as well as the original code. Hence, a completely new, isolated comprehension of

basic strategy had to be implemented. Aswell as a method for testing the comprehension of player hand and up card combinations, a set of all possible player hand and up card combinations had to be instantiated. A heuristic decision was made in the design of this test to only instantiate hands of up to 3 cards in length, as it was felt that this would provide8 program testing that was comprehensive enough without incurring unsiotable complexity. With 13 different card faces, the number of combinations that can be made from 2 elements (without repetitions) is 91. For combinations of 3 elements this number is 364. However, many of these possible 3 card combinations are hands that would exceed the value of 21, and hence, are removed from this set. Each of these card combinations are stored as string combinations of the characters '2', '3', '4', '5', '6', '7', '8', '9', 't', and 'a'. For each of these ten up card values, a card string is stored with it in a tuple pair. This tuple of player hand and dealer up card would act as keys for each possible test case of the makeChoice() method. An "alternate basic strategy" function was written to take keys such as this and return a correct basic strategy response. With 275 hand strings, each being paired in keys with 10 different dealer up card values; these 2750 keys were stored with their corresponding correct basic strategy response in the *key, value* pairs of a dictionary. This allowed the response from *makeChoice()* to be compared with the correct return value corresponding to each hand, and any offending responses of *makeChoice()* could be flagged.

This test branch allowed for a strenuous debugging process to take place where errors in the *makeChoice()* were highlighted and dealt with accordingly. An error highlighted in this process was the incorrect catagorisation of soft and hard hands. When two or more aces featured in a soft hand, this hand was incorrectly catagorised as soft, and hence incorrect basic strategy responses of 'stand' were given in at least 33 cases where 'hit' was the correct reponse. As such the system components involved in the catagorisation of a hand as soft were reconfigured and all of these errors were remediated successfully. As a result of this it can be definitively said that this system has fulfilled its requirement to implement blackjack basic strategy aswell as the other integrated player strategies. Through examination of program variables printed to console during development, this program is completely capable of reliably and accurately simulating blackjack round, aswell as determining and storing their correct outcome. A specific "simulation controller" class has not been implemented as the functionality that is aimed to be achieved through the implementation of a controller class has been produced through a "simulate" function in the the file main.py. This "must-have" listed

element of the system criteria has been fulfilled with the functionality of the program. The system also succeeds in producing graph plots of results that are both visually stimulating and highly informative. While a few elements of the code have been altered from their intended design and added in in order to create a working solution, the code is generally well structured with adequate use of comments and technical documentation.

3. Results of Blackjack Strategy Optimisation

Using the project system, results were able to be gathered that show the effectiveness of basic strategy in comparison to other strategies of play. Two forms of graph plots have been produced. The first of which shows the avaerage of the cumulative returns each round across the median 10% of playing sessions. While this form of graphic is useful for illustrating the difference in performance across different playing systems, it does not offer any information about the total currency bet by the player that produces such returns. With this is mind, the second graph plot used displays the change to level of betting advantage experienced by the player round-by-round in an average of the median 10% of playing sessions.

In the figures below, graph plots of simulation results can be seen for the strategies: mimic the dealer, never bust, and basic strategy. In this average case, where basic strategy is followed over the course of 200 rounds of blackjack, the final sum of betting outcomes is -£10.85. Across the median subset of playing sessions, £2352665 was placed in wagers and led to the total betting outcome of -£10845. From this, we can say that the house advantage experienced by the basic strategy player was a mere 0.461% over this dataset. This result is very much in line with what would be expected from such a combination of player strategy and blackjack rules. In the other two entities shown in the graph, the results seen are far less pleasant. The final sum of betting outcomes in the average case after 200 rounds for the Mimic The Dealer and Never Bust strategies are -£115.79 and -£166.38 respectively. A player implementing the Mimic The Dealer strategy suffered a house advantage of 5.58%, and the implementation of Never Bust led to a house advantage of 8.01%.

From these findings, it is clear how important the adherance to Basic Strategy is for anyone wishing to optimise their experienced betting outcomes in Blackjack. Interestingly, these results also indicate that the house advantage suffered by the average

blackjack player playing on instinct alone may be multiple percentage points higher than the 2% indicated in my online source.

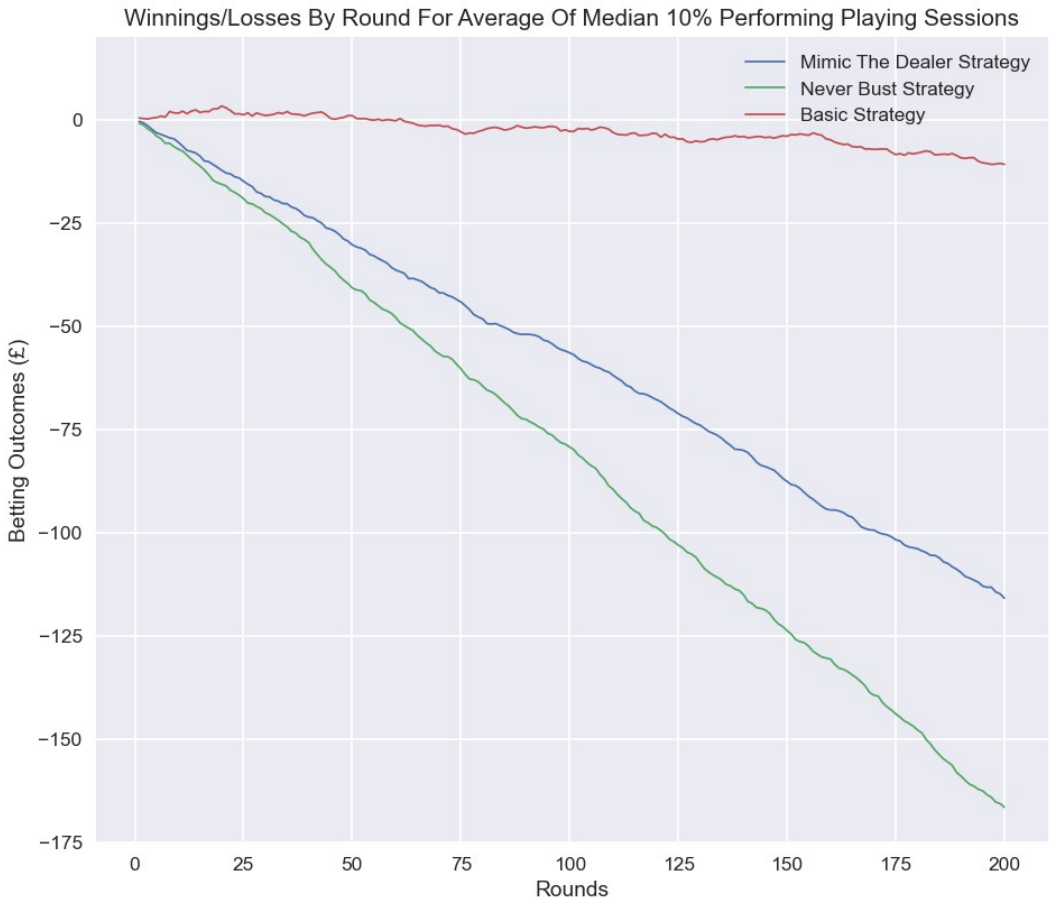


Figure 17: Betting Returns Across Strategies

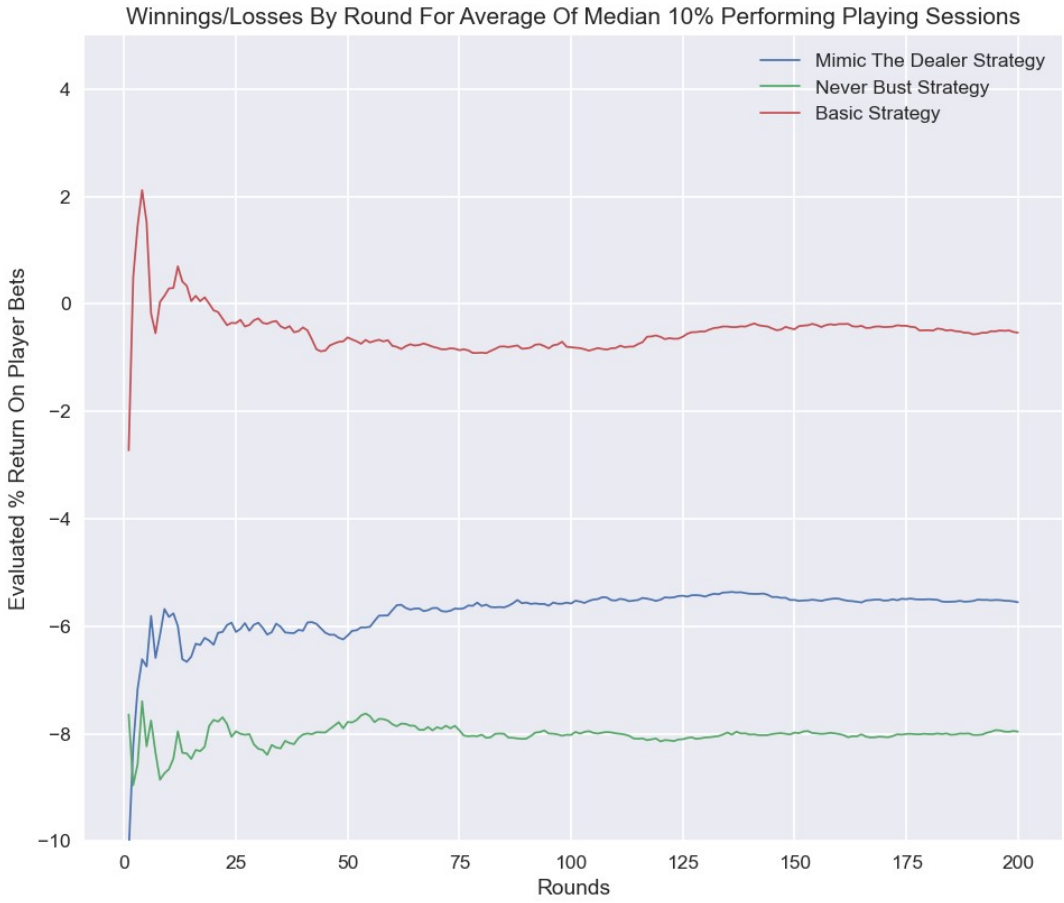


Figure 18: Evaluated % Return On Player Bets Across Strategies

4. Card Counting In Blackjack

Earlier in this report, it has been discussed how in-game factors influence the outcome probabilities of a blackjack round - with the set of cards dealt by the dealer over the course of a round being one of these factors. While the effects of each different card dealt in a blackjack round have different impacts on the house advantage experienced by the player that round for various different reasons; one simple generalisation that can be made for all cards is that high-value cards increase the experienced betting odds of the player for that rounds, and low cards decrease them. Card counts are systems used by players to track metrics of the shoe's favourability over rounds of blackjack.

True Counts and Running Counts

To metrify the favourability of a dealer's shoe, card count systems attribute values to each card - such as negative 1 for cards with face of Ten, Jack, Queen, King or Ace; zero for cards with a face of Seven, Eight or Nine; and positive one for cards with faces Two, Three, Four, Five, Six. A card counting player will keep track of a *running count*, starting from zero (generally) and adjusting this metric in accordance with the assigned card values for every card seen dealt from the deck. A positive running count number, such as 4, indicates that cards that negatively impact the likelyhood of success for the player have been dispensed from the shoe four times as frequently as cards that are helpful to the player over the course of counting. When calculating the density of helpful cards to the player in the deck however, we must consider the size of the shoe as well as the frequency of helpful cards. This is done in a process called true count conversion, where the running count is divided by the number of decks remaining in the shoe to determine a true count. While most modern card counting systems use a true count to inform bets, some systems forgo this step.

At a true count rating of 0, the average 0.5% house advantage of blackjack is experienced by the player. As the true count rating of the dealers shoe increases from negative values to positive values, the betting advantage experienced by the player increases linearly. Only when the true count rating of the dealers shoe increase to a value of one does the betting advantage experienced by the player reach 0% (even odds). Only at true count values of 2 or larger does the player experience a positive betting advantage over the casino.

In order to illustrate this phenomenon, simulations of basic strategy in blackjack were performed with the implementation of altered shoe objects that emulated casino shoe's of different true counts. The betting outcomes across a reasonable sample size of data were represented in graph form (shown in figure x). A short discussion of design for this system implementation is held in the appendicies of this report.

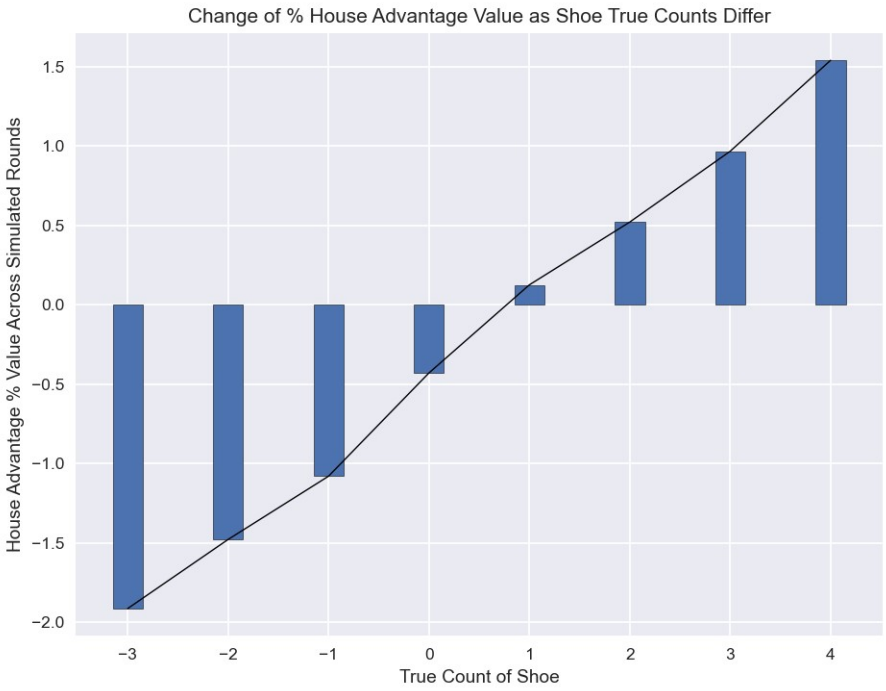


Figure 19: Change To House Advantage Value With CHanging True Count

This knowledge may evoke a very hopeful view of the effectiveness of card counting. However, the number of instances in blackjack games where positive betting odds are present are few. Later in this project, the existing simulation framework is added upon to feature the simulation of different card counting methods. Modifications to this simulation system were able to be made in order to track the number of instances of different true count ratings across a large sample size of blackjack rounds. For each true count value a percentage value for it's number of instances from all true count instances was calculated.

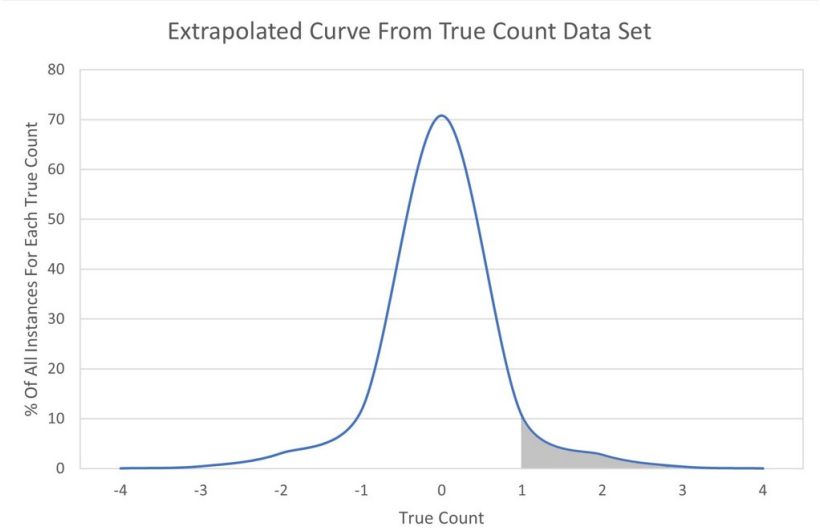


Figure 20: Extrapolated Curve From TC Distribution Dataset

True Count	Number Of Instances	% Of Instances
-7	1	0.00005
-6	6	0.00030
-5	88	0.00440
-4	1530	0.07650
-3	9650	0.48250
-2	62347	3.11735
-1	234848	11.74240
0	1416116	70.80580
1	210188	10.50940
2	55647	2.78235
3	8285	0.41425
4	1230	0.06150
5	59	0.00295
6	5	0.00025

Figure 21: Tc Distribution Tabular DataFrame

In the figure at the top of this page, from the discrete data points for % values of the frequency of instances of each true count from all instances, line smoothing has been applied to form a gaussian-style distribution from discrete points of data. As can be seen in the shape of the curve, the frequency is significantly higher at the mean true count value of 0, and steeply declines as points deviate away from 0 along the x axis. As a result, the combined percentage of true count instances in this simulation sample that occur at a value of 1 or higher is a very small subsection of all instances. This area has been highlighted under the curve. The size of this subset where the player experiences even or positive betting odds is a mere 13.77%. While it is possible for card counters

to take a short intermission from play when the most unfavourable betting cases arise, unfortunately it is not possible to simply opt out of play any time that the true count of the deck reaches a certain negative value without attracting suspicion from casino representatives. Therefore, with the information gathered about the betting odds of each round through the use of a card count system, a counting player will adjust the size of their bets made before each round in line with a predetermined '*bet spread* in order to maximise their probable profits while avoiding incurring too low or too high a level of risk.

Origins Of Card Counting - The Work of Edward O. Thorp

In the 1960's, a professor of Mathematics from U.C Irvine called Edward O. Thorpe began investigating how players could increase their betting returns in blackjack games. He was intrigued by the work of his multiple colleagues from the American Journal of Statistical Association in "The Optimum Strategy in Blackjack". He used simulation methods to investigate the benefit to the player resultant of each possible move for every possible scenario in blackjack. With the results he extracted from this, he produced Basic strategy as an arrangement of perfect moves for minimising the advantage from the house.

He then modified his simulations to investigate what effect changing the composition of the deck would have on game outcomes. With a clear increase to the player advantage as the porportion of ten-value cards in the deck increased, Thorpe created a counting system where the count would turn positive when the deck is rich enough in tens to favour the betting odds of the player. This was called the Ten-Count system, and was the first card count created in the world. The success of the Ten-count led casinos to add more decks to the dealer's shoe to reduce the effectiveness of this relatively simple system.

4.1. Balanced Card Counts

4.1.1. The Hi-Low Count

Since the addition of more decks to blackjack games, the HiLo count has became the most widely used and effective card counting methods. It was established by Harvey Dubner and, after presenting his strategy to a panel discussion "Computers Applied to Games of Skill and Chance" at the 1963 Fall Joint Computer Conference, the strategy was published in the 1966 revised edition of Beat the Dealer. In the Hilo count, a

running count is adjusted as the player sees cards dispensed from the deck according to the card values shown below. A true count conversion is then performed to provide a value that informs the size of the player's bet in line with their chosen bet spread.

Two	Three	Four	Five	Six	Seven	Eight	Nine	Ten	Ace
+1	+1	+1	+1	+1	0	0	0	-1	-1

4.1.2. The Ace/Five Count

The Ace/Five count is a system proposed to serve as a simplified version of the HiLow count. This count system is purposed to benefit amateur card counters that do not possess the level of experience required to use some of the more accurate count systems, but still wish to improve their betting odds (Shackleford). While it requires alot less maintenence in running count adjustments, a true count conversion is generally still applied in the use of the Ace/Five count.

Two	Three	Four	Five	Six	Seven	Eight	Nine	Ten	Ace
0	0	0	+1	0	0	0	0	0	-1

4.1.3. The WongHalves Count

The Wong Halves count is one that works similarly to the common HiLo system with the incorporation of a more complicated array of values assigned to cards. It was introduced in the book "Professional Blackjack" (-CITEP-). Rather than adhering to a relatively simple system such as HiLo where 1 is added to the running count for cards two through 6, and 1 is subtracted from the running count for cards 10 and above; the wong halves count takes into account the change to Ev associated with each card more precisely. As a result, the count should have a slight edge over HiLo in maximising a player's betting odds. After adjusting the running count in accordance with the values below, a player then performs a true count conversion to inform their betting spread.
note in comment–

Two	Three	Four	Five	Six	Seven	Eight	Nine	Ten	Ace
+0.5	+1	+1	+1.5	+1	0.5	0	-0.5	-1	-1

4.2. Bet Spreads For Balanced Counts

By successfully using a card counting system to determine the current count, a player has made themself more informed of the current Ev rating for the current round. In order for a player to start imrpoving their betting outcomes, bets must be precisely adjusted in size as their card count changes to avoid incurring to little or too much risk with regard to the level of house advantage for the current round.

A bet spread is essentially a function used by the player to determine the size of the bet they wish to make based on the value of their current count. The size of the bets to be made is given in a number of betting units. The two bet spreads that will be considered in this investigation are the 1-8 and 1-12 bet spreads. In both systems, the player will bet 1 betting unit while the true count is equal to 1 or lower. Then over 3 increments where the true count is equal to 2, 3, and then 4; the size of the bets to be made increases to their maximum betting amounts of 8 and 12 units.

The 1-8 bet spread is thought to operate at a lower level of risk than the 1-12, as the amount of money put at risk of being lost per round is greater. By comparing the performance of all three balanced counts examined in this in project with either betting system will allow us to make an informed determination as to whether supposed "high risk" of the 1-12 bet spread is worth incurring.

True Count	1-8 Betting Units	1-12 Betting Units
<= 1	1	1
2	2	3
3	4	6
>=4	8	12

4.3. Unbalanced Card Counting: The KO Count

The KO, or Knock Out count was introcuced in the book "KnockOut Blackjack". The KO count works differently to those I have previously discussed in that it is what is called an *unbalanced count*. This means that the number of cards in the deck assigned a positive value is not equal to the number of cards valued negative, meaning that the running count will not total zero for a deck of cards. The KO count is different to many count systems in that it forgoes the conversion of the running count to a true count, and the starting count index is below zero and dependent on the number of decks in the shoe. bet spread takenfrom writings in book

Number Of Decks	Starting Count
One	-2
Two	-4
Four	-8
Six	-12
Eight	-16

It was proposed as a simple, but equally effective system for players to use who wish to go without the complications of the true count conversions associated with many card counting systems.

Two	Three	Four	Five	Six	Seven	Eight	Nine	Ten	Ace
+1	+1	+1	+1	+1	+1	0	0	-1	-1

4.4. Deviating From Basic Strategy

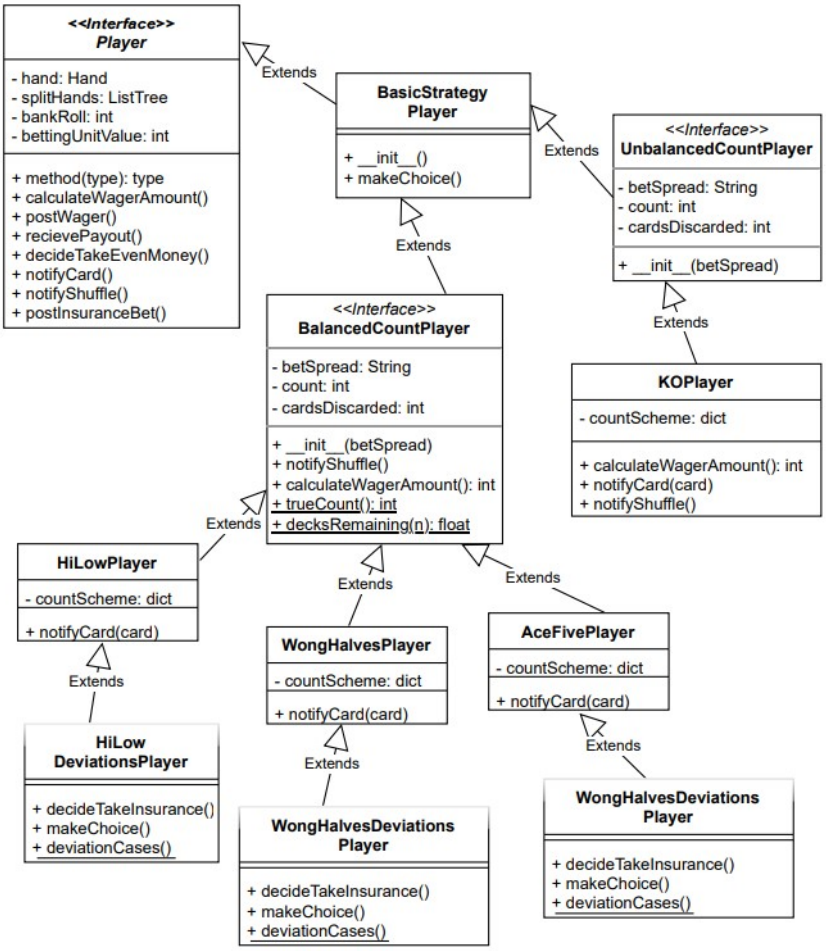
Basic strategy was defined to be a set of the best moves for the player to follow in a context where the player has no knowledge about the composition of the dealer's shoe. When a player uses card counting methods to gain insight into the expected levels of house advantage likely hood of different cards being dealt, some moves from basic strategy can actually be modified with consideration to the current true count to improve betting outcomes. The set of playing deviations that will be those introduced in the book "Blackjack Attack" by Don Schlesinger (6).

5. Build of Card Counting Simulation

In order to extend the system to integrate card counting methods aswell as well as perfect player strategy, some modules have to be added to the program. Each form of card counting player entity will be modelled in this simulation with different player subclasses. Each individual card counting player object will have their own specified "countScheme" attribute. This attribute will store a dictionary which can be queried to return the count value associated with each card.

Card counting player objects will also store the current running count and number of cards discarded from the shoe since last shuffle as integer values in variables called "count" and "cardsDiscarded". These attributes will be adjusted over the course of

rounds with the functions "notifyCard()" and "notifyShuffle()". notifyCard() will be called and passed in the cardface of each card that is dealt from the dealer. The value of this cardFace will then be passed into the count scheme of this player object, returning a value that will be added or subtracted from the player's count attribute. notifyShuffle() will be called each time the dealer shuffles the shoe of cards of this blackjack table, resetting the player's "cardsDiscarded" attribute to the integer value of 0. All counting player classes will implement a bet spread through selection statements overriding the calculateWagerAmount() method held by all Player objects. For an unbalanced count method, the player running count held in "count" will be addressed in this method to determine the size of bet returned. However, for balanced count players, a true count conversion needs to first be applied. Player objects that use card counting deviations from basic strategy will do so through modifications to their makeChoice() functions. Selection statements will be used in the program to follow specified branches in code if the true count of the player is within a given value range.



6. Results Of Card Counting In Blackjack And Conclusions

The inclusion of multiple graph plots for the lower 10%, median, and highest 10% of result values was suggested to me by my supervisor for this project. Unfortunately due to concerns of this report's length this was not able to be done. The incorporation of error bars into result graphs was considered, but it became clear that this inclusion would make the figures overly chaotic and difficult to interpret. Ultimately, the original approach of displaying an average plot from the median 10% of all values was taken.

Results from the simulation system were printed in String form to a local text file and

results were represented in graph plot form using pythons matplotlib library.

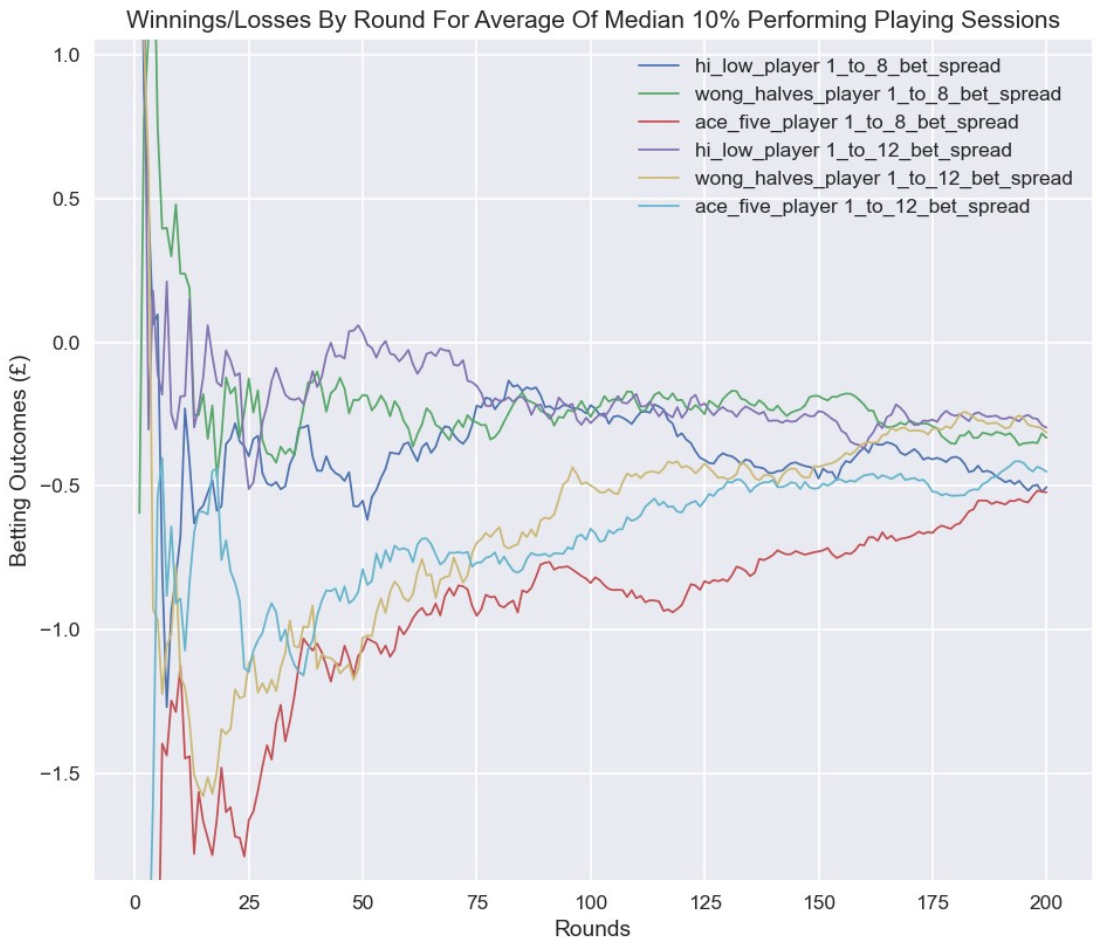


Figure 22: Evaluated % Return On Player Bets Across Card Counting Systems

Playing Strategy	Improvement To Betting Odds With 1-12 over 1-8
HiLow	+0.2071%
WongHalves	+0.0357%
Ace5	+0.0718%

As is shown in the data above, an improvement is made from the use of the 1-12 bet spread as opposed to the 1-8 system. While there was a degree of deviation across simulation results, an average improvement of 0.1% proved to be a repeatable phenomena. Thus, we can say that over numerous rounds of blackjack the overall risk of the 1-12 betting spread is actually lower than that of the 1-8 system.

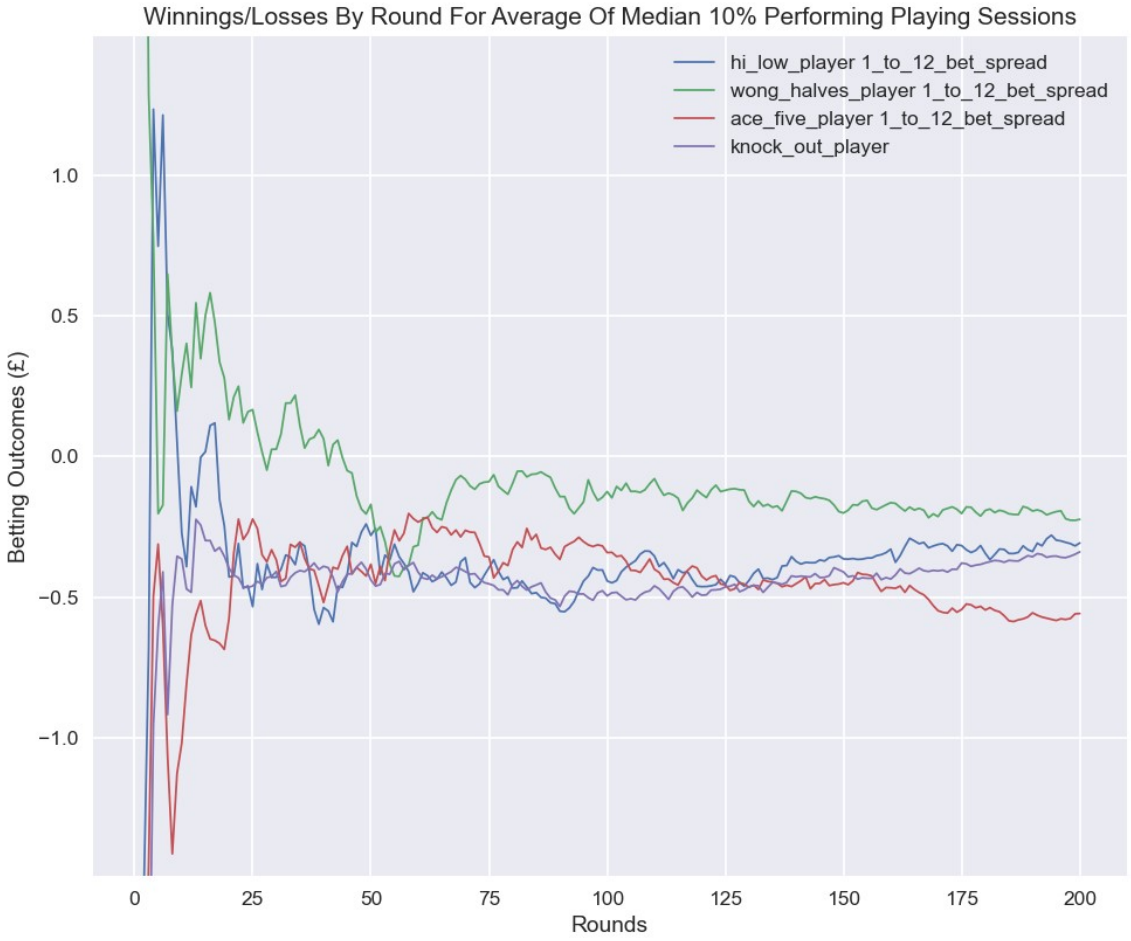


Figure 23: Evaluated % Return On Player Bets Across Card Counting Systems

Playing Strategy + Bet Spread Experienced	Total Bets Made	Total Returns	House Advantage
HiLow 1-12	5051155	-15505	-0.3070%
WongHalves 1-12	5096425	-11365	-0.2230%
Ace5 1-12	4714110	-26275	-0.5574%
Knock Out	6005940	-20330	-0.3385%

The Ace5 count is said to be a slightly effective counting method to be used by players who are not skilled enough to use the HiLow count, but still want to improve their odds. However, from this data we can say that the Ace5 count has a largely insignificant effect

on player betting outcomes. This leads me to conclude that the Ace5 offers nothing to players but a tool for learning more effective methods, and can't be relied on to improve the betting outcomes of players.

The Hilow count is known as reliable and effective counting system. From the data we can see that the HiLow system can offer a 0.2% improvement in betting odds when compared to basic strategy alone. With this we can safely conclude that the reputation of the HiLow method is well-earned and offers definite value to players wishing to improve their betting odds.

The Wong Halves is of course proposed as a more effective extension of the HiLow method with it's greater mathematical accuracy across the values featured in it's count scheme. In the data we can see an improvement of 0.28% in betting odds when compared to the use of basic strategy. From this we can definitely say that the Wong Halves count system is one of above average performance. An important consideration however with these results, aswell as the results from this project as a whole, is the impact of human error. While the perfect employment of this system offers a dramatic improvement in results compared to the HiLow system, it would take a rather experienced and confident card counter to be able to perform this without a margin of error diminishing this significantly.

The KnockOut count, while being proposed as a simpler alternative method of card counting than the more commonly used HiLow count, succeeds in achieving a significant improvement to betting odds with this lower complexity. With a cultivated house advantage of -0.34%, it's level of success is not quite as significant as that of the HiLow count. It is a system with an interesting method of execution that should be held in high regard as a generally reliable and effect method.

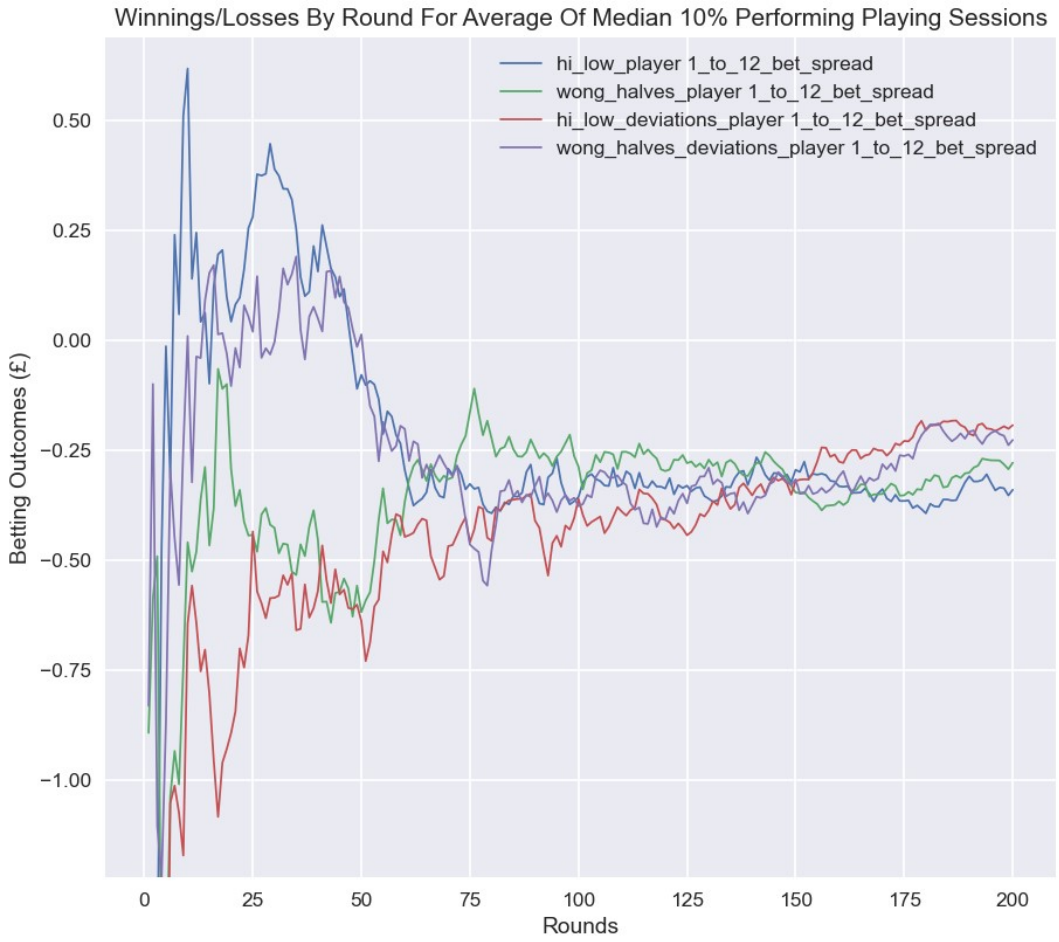


Figure 24: Evaluated % Return On Player Bets Across Card Counting Systems

Playing Strategy	Improvement To Betting Odds With Deviations
HiLow	+0.1469%
WongHalves	+0.0516%

A clear additional improvement upon the betting odds of a player can be made through the inclusion of playing deviations into a card counting system by around 0.1%. This does come across as slightly underwhelming when compared to the claims of playing deviations increasing a player's betting advantage of as much as 40% in a game, but these claims are clearly made in the context of their individual scenarios of use.

Unfortunately, these card counting methods will not make blackjack into a wildly profitable game for any gambler. What they can do is decrease the overall losses experienced a player and give them a slight edge the game of luck that is blackjack.

A. Appendix A: Full system MoSCow System Requirement Analysis

"Must-have" features:

The project system must reliably and accurately simulate rounds of blackjack. This involves many things including:

- the instantiation of valid decks of cards aswell as proper shuffling.
- a module to return decisions in response to in-game events in line with the "mimic the dealer", "no bust" and "basic strategy" playing strategies in blackjack.
- the correct excecution of responses returned by the aforementioned "in-game-decisions" module.
- the correct determinations made for the outcomes of completed blackjack rounds.
- the correct adjustment of card counting player running counts in line with their specific card count-value scheme.*
- the correct upkeep to the value of the attribute "cardsDiscarded" in card-counting type player objects.*
- the correct determination of "true count" using the values of card-counting player attributes "cardsDiscarded" and "runningCount".*
- bets returned in line with the bet spread specified for each card counting player type across different true count and running count values - aswell as the return of appropriate betting amounts for non-card-counting player object types.*

A **Simulation Controller** must also be built as part of this project system that can handle the execution of numerous blackjack "playing sessions" comprimising of a given number of rounds; aswell as storing 'results-data' gathered from the playing sessions that have been simulated.

The results gathered over the course of smiulations in the program must be returned in graph plots, text printed to file, and data-frame tables where appropriate.

"Should-have" features:

The programming of this system should be produced in a well-structured, clear, modular form with clear adherance to sound programming practises. Technical system documentation should also be produced once system development is complete to aid the understanding of the system during the evaluation or future development of the system.

"Could-have" features:

* points unmarked with asterisk pertain to the initial system build before the integration of card counting methods. Points marked with an asterisk are relevant to the subsequent full system implementation.

This project system could be extended to feature the integration of shoe objects that model a set of cards with a true count of a certain value. This would be done by adjusting the ratio of high-value cards to low-value cards upon the instantiation of each shoe.

The project system could also be extended to integrate the tracking of the number of instances in which the true count value of the dealers shoe is equal to different values over the course of many rounds. This would be done to measure the frequencies that different true counts occur over the course of blackjack rounds.

In order to attempt increasing the speed at which this system could perform it's many thousands of blackjack round simulations, the integatration of parallel processing could be made in either the execution of "split" hands during rounds, or the execution of rounds themselves.

"Won't-have" features:

This project will not incorporate the use of machine learning elements to build optimal playing procedures such as a "maximally optimal bet spread", ect as this task of great programming complexity is not in line with the investigation of current methods of blackjack betting optimisation.

A feature that might harvest interesting results but will not be implemented in this project system is a variable card counting "inaccuracy" factor that simulates the influence of human error in the use of different card counts.

References

[Baldwin] Baldwin, R. R., C. W. E. M. H. . M. J. P. The optimum strategy in blackjack.

[Chatter] Chatter, V. Know the house advantage for popular casino games. Available at *https://www.businessinsider.com/house-advantage-for-casino-games-2013-5?r=US&IR=T*. Accessed: 2022-07-04.

[3] Golden, L. M. (2016). An analysis of the disadvantage to players of multiple decks in the game of 21. pages 7–8. International Gaming Institute.

[IBM] IBM. Class diagrams. Available at *https://www.ibm.com/docs/en/rational-soft-arch/9.7.0?topic=diagrams-class*.

[Research] Research, C. F. G. The evolution of casino games, an examination of shifting unit and revenue trends. Available at *https://gaming.unlv.edu/reports/strip_game_mix.pdf*.

[6] Schlesinger, D. (1997). *Blackjack Attack, Playing the Pros' Way*. RGE Publications.

[Shackleford] Shackleford, M. The ace/five count. Available at *https://wizardofodds.com/games/blackjack/ace-five-count/*. Accessed: 2020-12-02.

[8] Werthamer, N. (2009). *Risk and Reward The Science of Casino Blackjack*. Springer Science+Business Media.

[Woodshut] Woodshut, E. A brief history of blackjack the best place to play online. Available at *https://www.blackjackreview.com/wp/2020/02/04/a-brief-history-of-blackjack-and-the-best-place-to-play-online/*.

[Young] Young, S. The expected value of an advantage blackjack player. Available at *https://edge.twinspires.com/casino-news/the-house-edge-in-blackjack-everything-you-need-to-know/#:~:text=The%20house%20edge%20is%20the,%2C%20you'll%20lose%20%242*. Accessed: 2021-01-09.

[Zender] Zender, B. 6 to 5 blackjack payout. Available at *https://bj21.com/category/blackjack/pages/6to5blackjackpayout*. Accessed: 2020-12-02.