Logic Simulation Test

& ML AI Opponent Development

Drafted by:

Levi Holmstrom

Objective

- To understand how balanced the game logic is for each player.
- Discover variables that can tip the ratio in targeted directions.
- Possibility for logic to be trained into a model for AI opponent/difficulty.

Approach

1. Configure the core logic into a headless client to allow looping of the game mechanics.
2. Construct a defined framework that allows easy manipulation of variables
3. Introduce a new variable each testing phase until desired results are achieved.
4. Capture and plot results.
5. Examine results and determine if expenditure of testing is required.

Environment Setup

- Language:
  - Python
    - Modules:
      - Pygame, Random, Enum, Time
- IDE:
  - PyCharm
- Database:
  - MS Excel
- Learning Resource:
  - GPT3.5/4, Coding Discord, Google Search, YouTube (IBM Data Analyst from Australia)

Core Game Logic

Please refer to blindeye_rules_rev2 for a more in-depth explanation of the game mechanics.

https://github.com/iMlearnDinG/Python-Training/blob/main/blindeye_RULES_rev2.pdf

Variables
(Static)

- Rank
- Suit

```
class Suit(Enum):
    Spades = 4
    Hearts = 3
    Diamonds = 2
    Clubs = 1

def __init__(self):
    self.cards = [self.Card(rank, suit) for rank in range(1,
14) for suit in self.Suit]
    self.discard_pile = []
```

Win Condition

- Player with the shortest count to the dealer rank wins
- If rank proximity is a tie, we use a static suit value to determine the winner.
- If rank proximity and suit value are both tied, a pair of dice is rolled for each player. The highest scoring dice roll determines the winner.

```
rank_diff1 = min(abs(player1_card.rank -
dealer_card.rank), abs(13 - abs(player1_card.rank -
dealer_card.rank)))
    rank_diff2 = min(abs(player2_card.rank -
dealer_card.rank), abs(13 - abs(player2_card.rank -
dealer_card.rank)))

    if rank_diff1 < rank_diff2:
        return "player1"
    elif rank_diff1 > rank_diff2:
        return "player2"
    else:
        if player1_card.suit.value >
player2_card.suit.value:
            return "player1"
        elif player1_card.suit.value <
player2_card.suit.value:
            return "player2"
        else:
```

Tie Determination

- Dice Roll
- Value Comparison (Higher Wins)

```
def roll_dice():
    return random.randint(1, 6)

print("Tie! Rolling dice...")
player1_dice = roll_dice()
player2_dice = roll_dice()
print("Player 1 rolled a", player1_dice)
print("Player 2 rolled a", player2_dice)

if player1_dice > player2_dice:
    score.update_score("player1")
elif player1_dice < player2_dice:
    score.update_score("player2")
```

Development

Main Instance

Class definitions

. Cards .

- *Rank*
- *Suit*

```
class Card:
    def __init__(self, rank, suit):
        self.rank = rank
        self.suit = suit

    def __str__(self):
        return f"{self.rank} {self.suit.name}"
```

. Deck .

- *Shuffle*
- *Deal*
- *Discard*

```
class Deck:
    def __init__(self):
        self.cards = [Card(rank, suit) for rank in
range(1, 14) for suit in Suit]
        self.discard_pile = []

    def shuffle(self):
        random.seed(time.time())
        random.shuffle(self.cards)

    def deal(self):
        if not self.cards:
            self.cards, self.discard_pile =
self.discard_pile, self.cards
            self.shuffle()
        return self.cards.pop()

    def discard(self, card):
        self.discard_pile.append(card)
```

. Player/Dealer .

- *Hand*
- *Score Track*

```
class Player:
    def __init__(self, name):
        self.name = name
        self.score = 0
        self.hand = []

    def add_point(self):
        self.score += 1
```

. Assets .

Implementation @ later date

```
class Assets:
    def __init__(self):
        pass

    def draw(self):
        pass
```

. Score .

- *Score Add*
- *Score Update*
- *Score Display*

```
class Score:
    def __init__(self):
        self.players = {"player1": 0, "player2": 0}

    def update_score(self, player):
        self.players[player] += 1

    def get_score(self, player):
        return self.players[player]
```

Function definitions

. Proximity .

```
def rank_proximity(player_card, dealer_card):
    return abs(player_card.rank - dealer_card.rank) % 13


def suit_proximity(player_card, dealer_card):
    return abs(player_card.suit.value -
dealer_card.suit.value)
```

. Comparison .

```
def compare_cards(player1_card, player2_card,
dealer_card):
    rank_diff1 = min(abs(player1_card.rank -
dealer_card.rank), abs(13 - abs(player1_card.rank -
dealer_card.rank)))
    rank_diff2 = min(abs(player2_card.rank -
dealer_card.rank), abs(13 - abs(player2_card.rank -
dealer_card.rank)))

    if rank_diff1 < rank_diff2:
        return "player1"
    elif rank_diff1 > rank_diff2:
        return "player2"
    else:
        if player1_card.suit.value >
player2_card.suit.value:
            return "player1"
        elif player1_card.suit.value <
player2_card.suit.value:
            return "player2"
        else:
            return "tie"
```

. Print Info .

```
def print_cards(players, dealer):
    print("Player 1 cards:")
    for card in players[0].hand:
        print(card)
    print("\nPlayer 2 cards:")
    for card in players[1].hand:
        print(card)
    print("\nDealer cards:")
    for card in dealer.hand:
        print(card)
```

. Print Info .

```
def roll_dice():
    return random.randint(1, 6)
```

Main Initialization

```
def main():
    pygame.init()
    assets = Assets()
    players = [Player("player1"), Player("player2")]
    dealer = Player("dealer")
    score = Score()

    # Initialize and deal cards for the first game
    deck = Deck()
    deck.shuffle()
    for _ in range(5):
        for player in players:
            player.hand.append(deck.deal())
        dealer.hand.append(deck.deal())

    print_cards(players, dealer)

    playing = True
    while playing:
        assets.draw()

        for i in range(5):
            player1_card = players[0].hand[i]
            player2_card = players[1].hand[i]
            dealer_card = dealer.hand[i]

            winner = compare_cards(player1_card, player2_card, dealer_card)

            if winner == "player1":
                score.update_score("player1")
            elif winner == "player2":
                score.update_score("player2")
            else:
                print("Tie! Rolling dice...")
```

Output

First Output	Output /w Tie
Player 1 cards:	Player 1 cards:
1 Diamonds	10 Spades
10 Clubs	3 Spades
13 Clubs	9 Clubs
11 Spades	4 Hearts
7 Spades	8 Hearts
Player 2 cards:	Player 2 cards:
12 Diamonds	5 Spades
4 Spades	7 Clubs
3 Clubs	12 Hearts
12 Hearts	13 Clubs
9 Diamonds	1 Hearts
Dealer cards:	Dealer cards:
9 Spades	11 Hearts
2 Spades	4 Clubs
9 Clubs	7 Diamonds
7 Clubs	8 Diamonds
10 Diamonds	11 Diamonds
Current scores:	Tie! Rolling dice...
Player 1: 2	Player 1 rolled a 4
Player 2: 3	Player 2 rolled a 6

First output shows promising results. The player scores align with the implemented logic. A complete turn with no errors has been achieved.

Testing

We will now test the logic to retrieve the following statistics:

- Baseline Win Ratio
- Win acceleration
- Score Distribution
- Variable Outcomes
- Outcome Manipulation

Baseline / Balance

Player 1- & 2- Win Condition Totals.

3 Cycle Ratios to determine dataset accuracy.

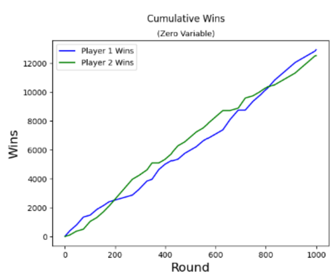
Zero game variables appended.

1 x 10^6 provides a thoroughly accurate baseline for preliminary results, with each player having almost a 50/50 chance of winning.

Also observed is the geometric form the graph lines take; it almost resembles DNA strands… interesting!

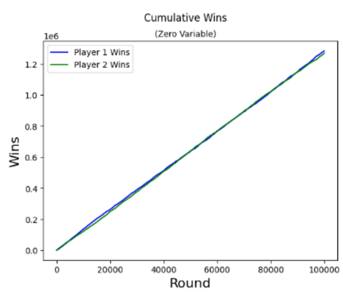
More research into this by-result is warranted.

Cycle Count: 1,000



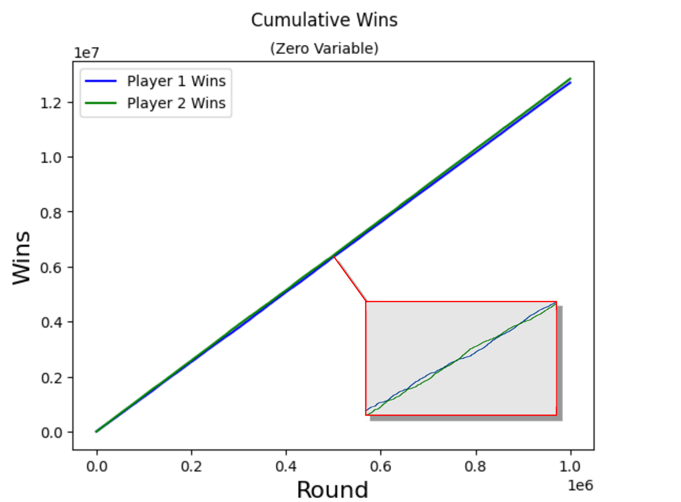
Total scores: Player 1 - 12930 (50.80%), Player 2 - 12519 (49.18%), Ties - 6 (0.02%)

Cycle Count: 100,000



Total scores: Player 1 - 1282819 (50.02%), Player 2 - 1268242 (49.45%), Ties - 13649 (0.53%)

Cycle Count: 1,000,000



Total scores: Player 1 - 12681906 (49.48%), Player 2 - 12823843 (50.03%), Ties - 124036 (0.48%)

Testing

continued...

Win Tracking

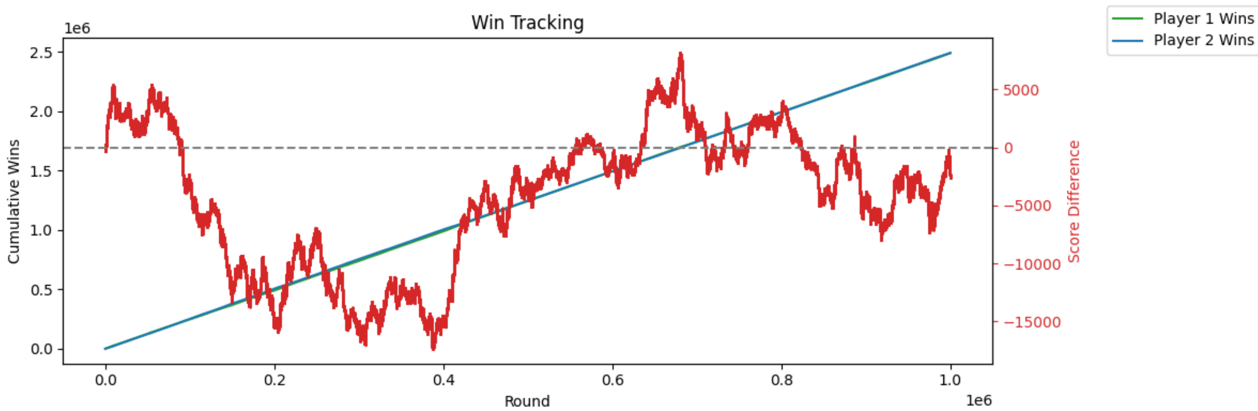
Visualise the number of times players are winning on the timeline.

+0 = Player 1 (*above line*)
-0 = Player 2 (*below line*)

Player 2 *has a significant lead at the start of game and peaks out at around 16k score ahead at 400k games.*

Player 1 *has a significant lead from 400k to 700k peeking out at 7k ahead in score.*

The remaining rounds *play out evenly, with the score difference returning to the zero line at the end of the game, indicating a fair chance for each player*



Score Distribution

Visualise the score value designation for each round.

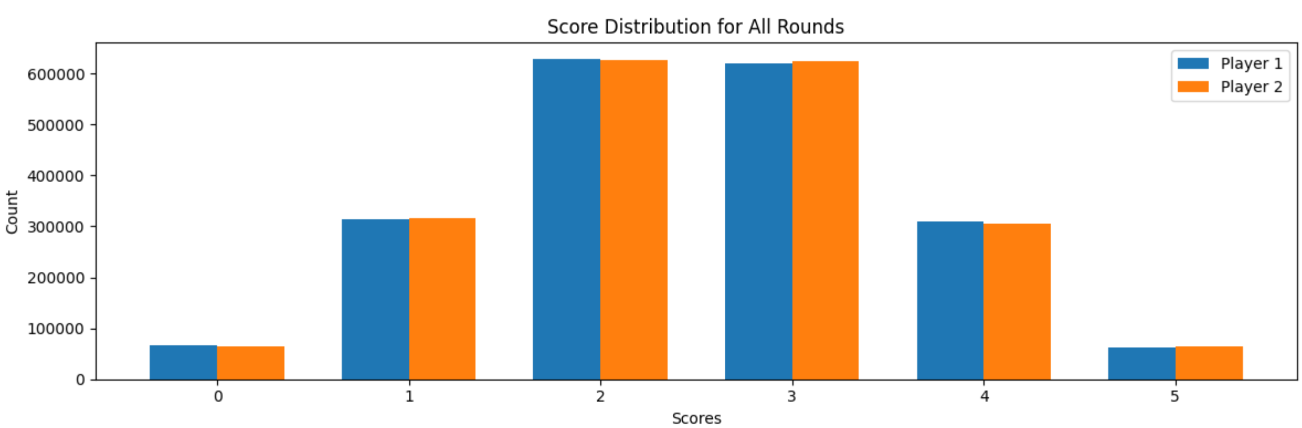
Help determine common vs rare scores

Testing *reports even score distribution for both players. This is exceptionally handy for setting a preliminary baseline betting ratio.*

0 & 5 – 5x

1 & 4 – 2.5x

2 & 3 – 1.5x



Testing

continued…

Outcome Manipulation

Bias penetration prevention and detection threshold.

100,000 Rounds x 8

Artificial Variables:

Player 1 – More odds of receiving higher suit from dealer. (Spades)

Player 2 – More Odds of receiving lower rank suits from dealer. (Clubs)

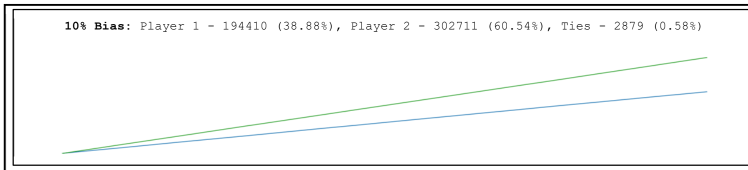
Manipulation Definition

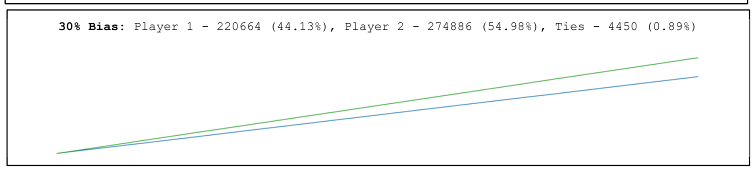
```
def deal_biased_cards(deck,
bias_factor=1):
    high_ranked_cards = [card for card
in deck if card[1] > 6]
    low_ranked_cards = [card for card
in deck if card[1] <= 6]

    num_high_ranked_cards = int(5 *
bias_factor)
    num_low_ranked_cards = 5 -
num_high_ranked_cards

    random.shuffle(high_ranked_cards)
    random.shuffle(low_ranked_cards)

    biased_hand_high =
high_ranked_cards[:num_high_ranked_car
ds]
    biased_hand_low =
low_ranked_cards[:num_low_ranked_cards
]
    random.shuffle(biased_hand_high)
    random.shuffle(biased_hand_low)

    for card in biased_hand_high +
biased_hand_low:
        deck.remove(card)

    return biased_hand_high +
biased_hand_low
```

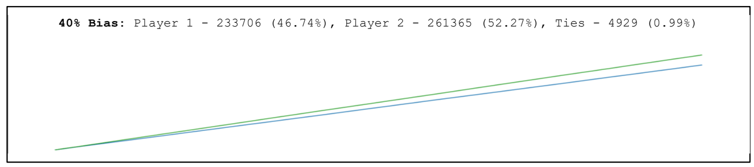
Manipulated Shuffle/Deal

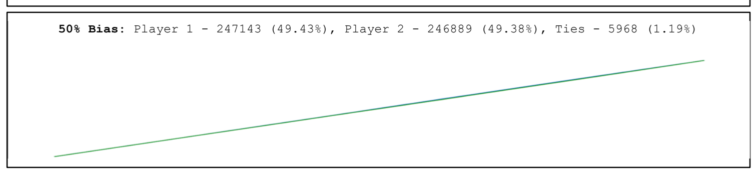
```
# Shuffle deck and deal cards
    deck = [(f"{k} of {s}",
card_values[k], suit_values[s])
for k in card_values for s in
suit_values for _ in
range(int(k))]
    random.shuffle(deck)
    player1_cards =
deal_biased_cards(deck,
bias_factor=0.5)  # You can
adjust the bias_factor to
increase or decrease the bias
    player2_cards =
deal_biased_cards(deck,
bias_factor=1 - 0.5)  # Subtract
the bias_factor from 1 to create
an inverse bias for player 2
    dealer_cards = [deck.pop()
for _ in range(5)]
```

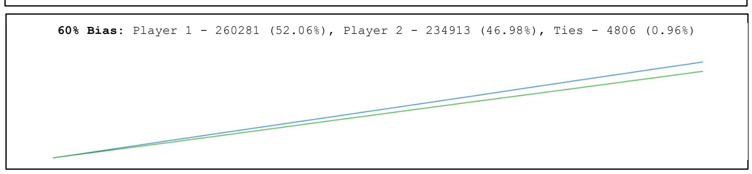
Bias Testing Results

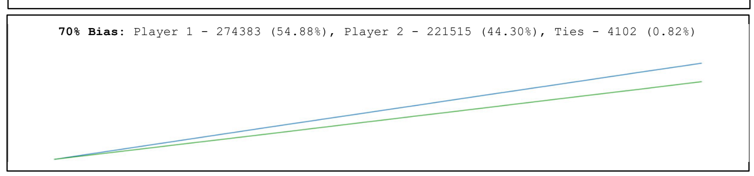


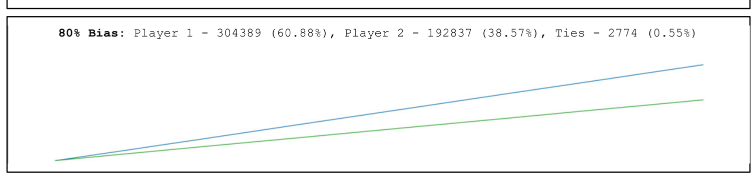


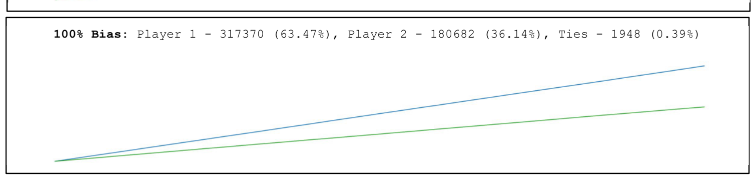












***The results show** that a bias penetration is possible with a code injection.*

This could be due to the simple nature of the Random Algorithm being used.

A more advanced algorithm is warranted for testing to see if a reduction in baseline deviation is achieved.

Testing

continued…

Introduce Variable

Game Mechanic Variable Initiated

5th Card Swapped cycle to simulate Blind Eye

***No noticeable deviation** from the initial game balance results. Reinforces game mechanics are performing as intended.*