PokerSimulation2

February 12, 2019

1 Analysis of 5-Card Poker Hands

1.1 Author: Gregory Betman

In [2]:
```
import numpy as np
import pandas as pd
import time
```

In [3]:
```
values = np.array(\
["2", "3", "4", "5", "6", "7", "8", "9", "10", "Jack", "Queen", "King", "Ace"])

suits = np.array(["Spades", "Clubs", "Hearts", "Diamonds"])

results = np.array(\
["High Card", "Pair", "Two Pair", "Three of a Kind", "Straight",\
 "Flush", "Full House", "Four of a Kind", "Straight Flush", "Royal Flush"])

results_prob = np.array(\
[50.1177, 42.2569, 4.7539, 2.1128, .3925, .1965, .1441, .0240, .00139, .000154])
```

In [4]:
```
# Builds a standard 52 card deck, each card with indices (i,j) where i represents value,
# and j represents suit

def build_deck():
    deck = np.empty(104).reshape(52,2)
    for j in range(4):
        for i in range(13):
            deck[i+13*j,:] = [i+2, j+1]
    return deck
```

In [5]:
```
# Prints hand in an interpretable format

def print_hand(hand):
    for i in range(hand.shape[0]):
        print(values[int(hand[i,0]-2)] + " of " + suits[int(hand[i,1]-1)])
```

In [6]:
```
# Builds a full deck

deck = build_deck()
print_hand(deck)
```

```
2 of Spades
3 of Spades
4 of Spades
5 of Spades
6 of Spades
7 of Spades
8 of Spades
9 of Spades
10 of Spades
Jack of Spades
Queen of Spades
King of Spades
Ace of Spades
2 of Clubs
3 of Clubs
4 of Clubs
5 of Clubs
6 of Clubs
7 of Clubs
8 of Clubs
9 of Clubs
10 of Clubs
Jack of Clubs
Queen of Clubs
King of Clubs
Ace of Clubs
2 of Hearts
3 of Hearts
4 of Hearts
5 of Hearts
6 of Hearts
7 of Hearts
8 of Hearts
9 of Hearts
10 of Hearts
Jack of Hearts
Queen of Hearts
King of Hearts
Ace of Hearts
2 of Diamonds
3 of Diamonds
4 of Diamonds
5 of Diamonds
6 of Diamonds
7 of Diamonds
8 of Diamonds
9 of Diamonds
10 of Diamonds
Jack of Diamonds
Queen of Diamonds
```

```
King of Diamonds
Ace of Diamonds
```

In [7]:
```
# Evaluates the hand value of a standard 5 card hand

    # Returns two integers (a, b) where a is the ranking representing each of
    # the 10 possible outcomes,
    # b is the value of the tie-breaking card (if there is one)

    def hand_value(hand):
        if royal_flush(hand)[0]:     return 10, royal_flush(hand)[1]
        if straight_flush(hand)[0]:  return 9, straight_flush(hand)[1]
        if four_of_a_kind(hand)[0]:  return 8, four_of_a_kind(hand)[1]
        if full_house(hand)[0]:      return 7, full_house(hand)[1]
        if flush(hand)[0]:           return 6, flush(hand)[1]
        if straight(hand)[0]:        return 5, straight(hand)[1]
        if three_of_a_kind(hand)[0]: return 4, three_of_a_kind(hand)[1]
        if two_pair(hand)[0]:        return 3, two_pair(hand)[1]
        if pair(hand)[0]:            return 2, pair(hand)[1]
        return 1, high_card(hand)
```

In [8]:
```
# Prints the result of the output from hand_value(hand) in an interpretable format

    def print_result(a, b):
        print("-------> " + results[int(a-1)] + " with " + values[int(b-2)])
```

In [9]:
```
# Returns the highest card value in hand

    def high_card(hand):
        return np.amax(hand[:,0])
```

In [10]:
```
# Evaluates if hand contains a royal flush
     # Returns 1 if true, 0 if false

     def royal_flush(hand):
         if straight_flush(hand) == (1, 14):
             return straight_flush(hand)
         return 0, 0
```

In [11]:
```
# Evaluates if hand contains a straight flush
     # Returns 1 if true, 0 if false

     def straight_flush(hand):
         if flush(hand)[0] and straight(hand)[0]:
             return straight(hand)
         return 0, 0
```

In [12]:
```
# Evaluates if hand contains a four of a kind
     # Returns two values (a, b) where a=1 if true, a=0 if false,
     # b is the value of the tie-breaking card
```

```
        def four_of_a_kind(hand):
            uniques = np.unique(hand[:,0])
            if hand[hand[:,0]==uniques[0]].shape[0]==4:
                return 1, uniques[0]
            if hand[hand[:,0]==uniques[1]].shape[0]==4:
                return 1, uniques[1]
            return 0, 0
```

In [13]:
```
        # Evaluates if hand contains a full house
        # Returns two values (a, b) where a=1 if true, a=0 if false,
        # b is the value of the tie-breaking card

        def full_house(hand):
            uniques = np.unique(hand[:,0])
            if hand[hand[:,0]==uniques[0]].shape[0]==3\
            and hand[hand[:,0]==uniques[1]].shape[0]==2:
                return 1, uniques[0]
            if hand[hand[:,0]==uniques[0]].shape[0]==2\
            and hand[hand[:,0]==uniques[1]].shape[0]==3:
                return 1, uniques[1]
            return 0, 0
```

In [14]:
```
        # Evaluates if hand contains a flush
        # Returns two values (a, b) where a=1 if true, a=0 if false,
        # b is the value of the tie-breaking card

        def flush(hand):
            if hand[0,1] == hand[1,1] == hand[2,1] == hand[3,1] == hand[4,1]:
                return 1, high_card(hand)
            return 0, 0
```

In [15]:
```
        # Evaluates if hand contains a straight
        # Returns two values (a, b) where a=1 if true, a=0 if false,
        # b is the value of the tie-breaking card

        def straight(hand):
            if high_card(hand)==14: # Ace is treated as both a high and low card
                sorted_acehigh = np.sort(hand[:,0], axis=0)
                acelow = hand.copy()
                acelow[np.where(acelow == 14)] = 1
                sorted_acelow = np.sort(acelow[:,0], axis=0)
                if np.all(np.diff(sorted_acehigh) == 1):
                    return 1, 14
                if np.all(np.diff(sorted_acelow) == 1):
                    return 1, 5
            sorted_values = np.sort(hand[:,0], axis=0)
            if np.all(np.diff(sorted_values) == 1):
                return 1, high_card(hand)
            return 0, 0
```

In [16]:
```
        # Evaluates if hand contains a three of a kind
        # Returns two values (a, b) where a=1 if true, a=0 if false,
```

```
        # b is the value of the tie-breaking card

        def three_of_a_kind(hand):
            uniques = np.unique(hand[:,0])
            if hand[hand[:,0]==uniques[0]].shape[0]==3:
                return 1, uniques[0]
            if hand[hand[:,0]==uniques[1]].shape[0]==3:
                return 1, uniques[1]
            if hand[hand[:,0]==uniques[2]].shape[0]==3:
                return 1, uniques[2]
            return 0, 0
```

In [17]:
```
        # Evaluates if hand contains a two pair
        # Returns two values (a, b) where a=1 if true, a=0 if false,
        # b is the value of the tie-breaking card

        def two_pair(hand):
            pair1 = 0
            pair2 = 0
            pair3 = 0
            uniques = np.unique(hand[:,0])
            if uniques.shape[0]==3:
                if hand[hand[:,0]==uniques[0]].shape[0]==2:
                    pair1 = uniques[0]
                if hand[hand[:,0]==uniques[1]].shape[0]==2:
                    pair2 = uniques[1]
                if hand[hand[:,0]==uniques[2]].shape[0]==2:
                    pair3 = uniques[2]
                return 1, np.amax([pair1, pair2, pair3])
            return 0, 0
```

In [18]:
```
        # Evaluates if hand contains a pair
        # Returns two values (a, b) where a=1 if true, a=0 if false,
        # b is the value of the tie-breaking card

        def pair(hand):
            uniques = np.unique(hand[:,0])
            if uniques.shape[0]==4:
                if hand[hand[:,0]==uniques[0]].shape[0]==2:
                    return 1, uniques[0]
                if hand[hand[:,0]==uniques[1]].shape[0]==2:
                    return 1, uniques[1]
                if hand[hand[:,0]==uniques[2]].shape[0]==2:
                    return 1, uniques[2]
                if hand[hand[:,0]==uniques[3]].shape[0]==2:
                    return 1, uniques[3]
            return 0, 0
```

In [19]:
```
        # Example of the code running with cards, manually inputted

        deck = build_deck()
```

```
        hand = np.empty(10).reshape(5,2)
        hand[0,:] = deck[17,:]
        hand[1,:] = deck[14,:]
        hand[2,:] = deck[15,:]
        hand[3,:] = deck[16,:]
        hand[4,:] = deck[13,:]

        print_hand(hand)

        a, b = hand_value(hand)
        print_result(a, b)
```

```
6 of Clubs
3 of Clubs
4 of Clubs
5 of Clubs
2 of Clubs
-------> Straight Flush with 6
```

```
In [20]: # Generates a permutation of random cards in a 52 card deck,
         # where n = number of cards generated

         def random_cards(n=10):
             deck = build_deck()
             return np.random.permutation(deck)[:n,:]
```

```
In [21]: # Test random hand generator

         x = random_cards(n=5)
         print_hand(x)
         a, b = hand_value(x)
         print_result(a, b)
```

```
Jack of Hearts
Ace of Hearts
Queen of Clubs
Jack of Clubs
5 of Hearts
-------> Pair with Jack
```

```
In [22]: # Function to simulate rounds of two random, computer-generated poker players

         # 5 random cards are given to player 1 (as hand1), and 5 random cards from
         # that same deck are given to player 2 (as hand2). The players compare hands
         # and the winner of the round is recorded. Then a new shuffled set of 10 cards
         # is generated and the simulation cycles again a number of times equal to the
         # value of hands (default = 1000000).
         # A progress message is printed for every 50,000 games simulated.
```

```
        # Returns data, with values (a, b, c) where a is hand value of player 1,
        # b is tie-breaking card, c is a number representing game status.
        # c = 1 if player 1 wins, c = -1 if player 1 loses,
        # c = 0 if tie (even after tie-breaking card)

        def simulate(hands = 1000000, display=False):
            data = np.zeros([hands, 3])
            for i in range(hands):
                if i%50000==0:
                    print("Games simulated: "+str(i))
                cards = random_cards()
                hand1 = cards[:5,:]
                hand2 = cards[5:,:]
                a1, b1 = hand_value(hand1)
                a2, b2 = hand_value(hand2)
                if a1 > a2:
                    data[i,:] = [a1, b1, 1]
                elif a1 == a2:
                    if b1 > b2:
                        data[i,:] = [a1, b1, 1]
                    elif b1 == b2:
                        data[i,:] = [a1, b1, 0]
                    else:
                        data[i,:] = [a1, b1, -1]
                else:
                    data[i,:] = [a1, b1, -1]
                if display:
                    print("Game "+str(i+1)+":\nHand 1:")
                    print_hand(hand1)
                    print_result(a1, b1)
                    print("\nHand 2:")
                    print_hand(hand2)
                    print_result(a2, b2)
                    print("\nStatus: "+str(int(data[i,2])))
                    print("---------------------------------")
            print("Games simulated: "+str(hands)+" ---> Done!")
            return data
```

In [23]:
```
# Example of small (2 game) simulation where display is enabled

        x = simulate(hands = 2, display=True)
```

```
Games simulated: 0
Game 1:
Hand 1:
9 of Spades
Ace of Clubs
3 of Hearts
Queen of Clubs
King of Hearts
-------> High Card with Ace
```

```
Hand 2:
Jack of Spades
6 of Hearts
8 of Spades
5 of Diamonds
6 of Spades
-------> Pair with 6

Status: -1
----------------------------------
Game 2:
Hand 1:
Jack of Diamonds
7 of Diamonds
2 of Clubs
2 of Spades
3 of Clubs
-------> Pair with 2

Hand 2:
8 of Hearts
3 of Spades
4 of Hearts
5 of Hearts
4 of Diamonds
-------> Pair with 4

Status: -1
----------------------------------
Games simulated: 2 ---> Done!
```

```
In [24]: # Simulation with 1,000,000 games (building a larger dataset for analysis)

         start = time.time() # Track time to completion
         x = simulate()
         end = time.time()
         print("Time elapsed: " + str(round(end - start, 2)) + " seconds")
```

```
Games simulated: 0
Games simulated: 50000
Games simulated: 100000
Games simulated: 150000
Games simulated: 200000
Games simulated: 250000
Games simulated: 300000
Games simulated: 350000
Games simulated: 400000
Games simulated: 450000
Games simulated: 500000
```

```
Games simulated: 550000
Games simulated: 600000
Games simulated: 650000
Games simulated: 700000
Games simulated: 750000
Games simulated: 800000
Games simulated: 850000
Games simulated: 900000
Games simulated: 950000
Games simulated: 1000000 ---> Done!
Time elapsed: 344.76 seconds
```

In [25]:
```
# Distribution of hand values for player 1

count = np.zeros(10)
for i in range(10):
    count[i] = x[x[:,0]==i+1].shape[0]

df1 = pd.DataFrame([results, count.astype(int), count/x.shape[0]*100, results_prob],\
                   index = ['Result', 'Hand Count', 'Percentage', 'True Probability']\
                  ).transpose()
df1
```

Out[25]:

	Result	Hand Count	Percentage	True Probability
0	High Card	500563	50.0563	50.1177
1	Pair	422762	42.2762	42.2569
2	Two Pair	47658	4.7658	4.7539
3	Three of a Kind	21358	2.1358	2.1128
4	Straight	3928	0.3928	0.3925
5	Flush	1965	0.1965	0.1965
6	Full House	1490	0.149	0.1441
7	Four of a Kind	267	0.0267	0.024
8	Straight Flush	8	0.0008	0.00139
9	Royal Flush	1	0.0001	0.000154

In [26]:
```
print('Win Rate:  ' + str(100*x[x[:,2]==1].shape[0]/x.shape[0]) + '%')
print('Tie Rate:   ' + str(100*x[x[:,2]==0].shape[0]/x.shape[0]) + '%')
print('Loss Rate: ' + str(100*x[x[:,2]==-1].shape[0]/x.shape[0]) + '%')
```

```
Win Rate:  47.0984%
Tie Rate:   5.8867%
Loss Rate: 47.0149%
```

In [27]:
```
# Export simulation data to CSV file as a pandas dataframe

df = pd.DataFrame(x)
df.to_csv("poker_data.csv", sep=',')
```

In [ ]: