AI Investigation: Using "Rock Paper Scissors" like game to experiment with using Confidence Intervals on results that should be conclusive

**with double mean CI instead of single mean CI*

AI Investigation Part 4+

Goal: test effectiveness of Confidence Intervals on games that should have dominant strategy: 2+ options and non-binary game results. Also to validate the use of sample mean as statistic.

Also amended previous problem of using single mean to construct CI

Creator: `Lang (Ron) Chen 2022`

In [1]:

```
import random
import numpy as np
import statistics as s
import scipy.stats
```

Define Game

Rock Paper Scissors Like game:

A: 0.25 +1; 0.25 0; 0.25 -1; 0.25 -2; `E(X) = -0.5; V(X) = 1.5`

B: 0.25 +1; 0.25 0; 0.4 -1; 0.1 -2; `E(X) = -0.35; V(X) = 1.05`

C: 0.25 +1; 0.25 0; 0.1 -1; 0.4 -2; `E(X) = -0.65; V(X) = 1.95`

There are also varied versions for B:

1. B: 0.25 +1; 0.25 0; 0.3 -1; 0.2 -2; `E(X) = -0.45; V(X) = 1.35`
2. B: 0.25 +1; 0.25 0; 0.275 -1; 0.225 -2; `E(X) = -0.575; V(X) = 1.425`
3. B: 0.25 +1; 0.25 0; 0.26 -1; 0.24 -2; `E(X) = -0.49; V(X) = 1.47`

In [2]:

```
CHOICE = ['A', 'B', 'C']
```

In [3]:

```
# # Case 1
# def winloss(player1, player2):
#     if player1 == 'A':
#         if player2 < 0.25:
#             return 1
#         elif player2 > 0.25 and player2 < 0.50:
#             return 0
#         elif player2 > 0.50 and player2 < 0.75:
#             return -1
#         elif player2 > 0.75:
```

```
#               return -2

#     elif player1 == 'B':
#         if player2 < 0.25:
#             return 1
#         elif player2 > 0.25 and player2 < 0.50:
#             return 0
#         elif player2 > 0.50 and player2 < 0.90:
#             return -1
#         elif player2 > 0.90:
#             return -2

#     else:
#         if player2 < 0.25:
#             return 1
#         elif player2 > 0.25 and player2 < 0.50:
#             return 0
#         elif player2 > 0.50 and player2 < 0.60:
#             return -1
#         elif player2 > 0.60:
#             return -2
```

In [4]:

```
# # Case 2
# def winloss(player1, player2):
#     if player1 == 'A':
#         if player2 < 0.25:
#             return 1
#         elif player2 > 0.25 and player2 < 0.50:
#             return 0
#         elif player2 > 0.50 and player2 < 0.75:
#             return -1
#         elif player2 > 0.75:
#             return -2

#     elif player1 == 'B':
#         if player2 < 0.25:
#             return 1
#         elif player2 > 0.25 and player2 < 0.50:
#             return 0
#         elif player2 > 0.50 and player2 < 0.8:
#             return -1
#         elif player2 > 0.8:
#             return -2

#     else:
#         if player2 < 0.25:
#             return 1
#         elif player2 > 0.25 and player2 < 0.50:
#             return 0
#         elif player2 > 0.50 and player2 < 0.60:
#             return -1
#         elif player2 > 0.60:
#             return -2
```

In [5]:

```
# # Case 3
# def winloss(player1, player2):
#     if player1 == 'A':
#         if player2 < 0.25:
#             return 1
#         elif player2 > 0.25 and player2 < 0.50:
#             return 0
#         elif player2 > 0.50 and player2 < 0.75:
#             return -1
```

```python
#         elif player2 > 0.75:
#             return -2

#     elif player1 == 'B':
#         if player2 < 0.25:
#             return 1
#         elif player2 > 0.25 and player2 < 0.50:
#             return 0
#         elif player2 > 0.50 and player2 < 0.775:
#             return -1
#         elif player2 > 0.775:
#             return -2

#     else:
#         if player2 < 0.25:
#             return 1
#         elif player2 > 0.25 and player2 < 0.50:
#             return 0
#         elif player2 > 0.50 and player2 < 0.60:
#             return -1
#         elif player2 > 0.60:
#             return -2
```

In [6]:

```python
# Case 4
def winloss(player1, player2):
    if player1 == 'A':
        if player2 < 0.25:
            return 1
        elif player2 > 0.25 and player2 < 0.50:
            return 0
        elif player2 > 0.50 and player2 < 0.75:
            return -1
        elif player2 > 0.75:
            return -2

    elif player1 == 'B':
        if player2 < 0.25:
            return 1
        elif player2 > 0.25 and player2 < 0.50:
            return 0
        elif player2 > 0.50 and player2 < 0.76:
            return -1
        elif player2 > 0.76:
            return -2

    else:
        if player2 < 0.25:
            return 1
        elif player2 > 0.25 and player2 < 0.50:
            return 0
        elif player2 > 0.50 and player2 < 0.60:
            return -1
        elif player2 > 0.60:
            return -2
```

In [7]:

```python
def validation(choice, CHOICE):
    if "Inconclusive" in choice:
        return 'Inconclusive'
    if choice not in CHOICE:
        return False
    return True
```

Simulation

In [8]:
```
RUNS = 1000000

data = {'A': list(), 'B': list(), 'C': list()}

sample = list()
victory = list()
for i in range(RUNS):
    obs1 = random.sample(CHOICE, 1)[0]
    obs2 = random.uniform(0, 1)

    data[obs1].append(winloss(obs1, obs2))
```

Algorithm for final choice

First: Manipulate data so that it is in a dictionary and the dictionary value is [mean, stdev, length]

In [9]:
```
stats = dict()
for choice in CHOICE:
    tmp = list()
    tmp.append(s.mean(data[choice]))
    tmp.append(s.stdev(data[choice]))
    tmp.append(len(data[choice]))

    stats[choice] = tmp

stats
```

Out[9]:
```
{'A': [-0.49940753140083577, 1.117561833519635, 333351],
 'B': [-0.4886360569411666, 1.108015539167295, 333467],
 'C': [-0.6490986908056258, 1.235547578636235, 333182]}
```

In [10]:
```
def final_choice(stats):

    stats.sort(key = lambda x:x[1][0], reverse = True)

    # Then tests whether the second, third and so forth values contain 0 within their joi
    inconclusive_list = [stats[0][0]]
    inconclusive = False
    for i in range(1, len(stats)):
        if in_range(construct_CI(stats[0], stats[i])):
            inconclusive_list.append(stats[i][0])
            inconclusive = True
        else: # Because all values are sorted, if the current choice's joint Confidence In
            break

    if inconclusive: # If inconclusive, return statement with the list of 'drawed' choices
        return f'Inconclusive: the following came to a draw {inconclusive_list}'

    return stats[0][0] # Else, return the dominant strategy
```

In [11]:
```
def construct_CI(stat1, stat2):
    """ Uses Welch's approximation to construct a joint CI of two means, unknown populatio

    xbar1 = stat1[1][0]
    xbar2 = stat2[1][0]
    s1 = stat1[1][1]
    s2 = stat2[1][1]
    n = stat1[1][2]
    m = stat2[1][2]
```

```python
    r = (s1**2 /n + s2**2 /m)**2 /(s1**4 /(n**2 * (n-1)) + s2**4 /(m**2 * (m-1)))

    q = scipy.stats.t.ppf(0.95, df = r)

    poolsd = np.sqrt(s1**2 /n + s2**2 /m)

    CI = (xbar1 - xbar2 - q * poolsd, xbar1 - xbar2 + q * poolsd)

    print(f'{stat1[0]} {stat2[0]}: {CI}')
    print('\n')

    return CI
```

In [12]:

```python
def in_range(CI):
    """ Helper function to check whether 0 is within the Confidence Interval """

    if CI[0] <= 0 and CI[1] >= 0:
        return True
    return False
```

In [13]:

```python
stats
```

Out[13]:

```
{'A': [-0.49940753140083577, 1.117561833519635, 333351],
 'B': [-0.4886360569411666, 1.108015539167295, 333467],
 'C': [-0.6490986908056258, 1.235547578636235, 333182]}
```

In [14]:

```python
result = final_choice(list(stats.items()))
result
```

```
B A: (0.0062884487871592825, 0.015254500132179107)
```

Out[14]:

```
'B'
```

The algorithm successfully returned the dominant strategy: B

Validation

In [15]:

```python
validation(result, CHOICE)
```

Out[15]:

```
True
```

Emperical Testing

In [16]:

```python
victory = list()
for i in range(RUNS):
    obs2 = random.uniform(0, 1)
    victory.append(winloss(result, obs2))
s.mean(victory)
```

Out[16]:

```
-0.490721
```

A few words on experimental results:

Using the two-mean CI, the experiment is returning much better results even for case 4 (51% vs 50%). However this does not rule out the urgent need for a more comprehensive experiment to determine what

Confidence % to use.