Department of Electrical Engineering

California State University, Long Beach

Probability and Statistics with Applications to Computing

Duc Tran

Binomial Coefficient

Submitted by

Devin Suy

2020 October 15

Introduction

The purpose of this laboratory was to utilize the Python scripting language to perform three different probability simulations. The simulations conducted were: given a total population size, three party affiliations along with their respective sizes, a sample size, the probability was calculated of the random sample being comprised entirely of individuals from the same party, for each party; in the next experiment for a given value *n = {10, 50}* where a class of children is comprised of *4n* children, with *2n* boys and *2n* girls respectively, the probability was calculated that a random sample of size *2n* contains an equal amount of boys and girls for all values of *n*; lastly given a number pool with values *[1, 20]* in which a player select four values at random and four numbers are selected at random as the winning numbers, the probability was calculated for all four of the player's numbers matching the winning numbers in any order. Each experiment is by default is shuffled *500* times on initialization and performed over *100,000* trials.

Methods

In the first experiment we begin by appending the party letter *A* amount of times for party A, then *B* amount of times for party B, and *C* amount of times for party C to our "population" list resulting in a total size of *N*. The population list is then shuffled 500 times using Python's random library to ensure the values are spread out fairly. A data dictionary is created that maps the party letter to the amount of times the sample returned unanimous support for that party for each party *{A, B, C}*, with each mapped value initialized to zero. To simulate this experiment, we simply randomly sample the population for *group* number of elements. The utility function *full_support(sample)* is used to check if the sample is unanimous by returning True *if* set(sample) is of length 1 since sets only maintain distinct elements. Each time a *full_support* returns True, we increment the mapped occurrence value for the party that received unanimous support in the sample. The probability of each party reaching this state is simply calculated by dividing the mapped count by the total number of trials and outputted.

Similar to the first experiment, the following experiment is implemented by generating a population by appending the value True *boy_scalar*N* amount of times and the value False *girl_scalar*N* amount of times where *boy_scalar = girl_scalar = select_scalar = 2* by default representing the *2n* boys, girls, and sample size we want to randomly select for values *n = {10,50}*. We sample from our population *select_scalar*N* amount of elements which are passed to utility function *has_equal(sample)*, recall that boy is represented by the value of True, that maps ‘Boy’ and ‘Girl’ to the amount of times they appear in the sample, returning True only when the occurrences for each are the same. For each sample in our trials that returns a value of True, we simply increment the variable *equal_count*, from which we can calculate the probability of a random sample containing an equal amount of boys and girls by again dividing this value by the total number of trials conducted.

The last experiment is implemented by generating a randomized distinct numbers pool that contains values from *[min_number, max_number]* which take the values *1* and *20* by default. For each trial we then randomly sample the number pool to select *draw_size* elements, *4* by default, for the player, and repeat the process to select the winning numbers. The two selections are then compared as *set()* objects since their ordering is irrelevant here, should the set elements of both the player and winning set match, we simply increment *win_count*. The probability of winning said lottery is then calculated by dividing *win_count* by the total number of trials conducted and outputted along with the other parameters.

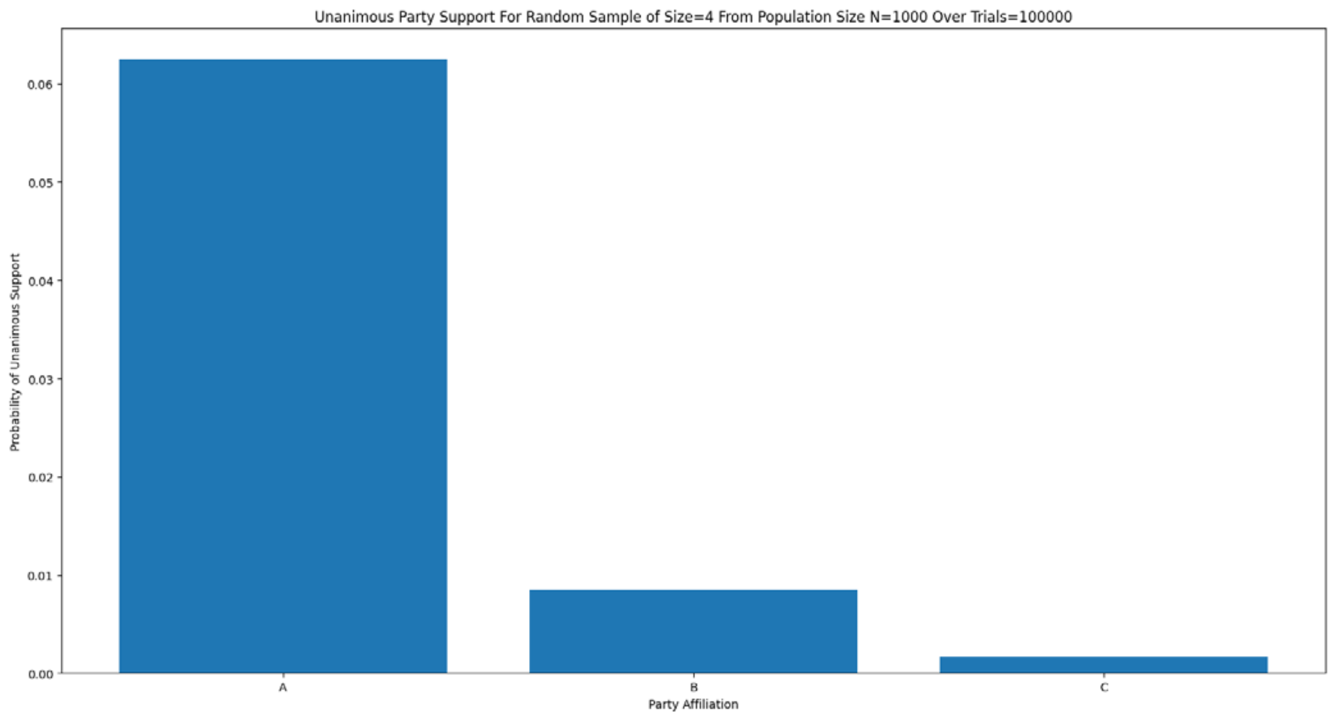
$$\binom{n}{k} = \frac{n!}{k! \cdot (n-k)!}$$

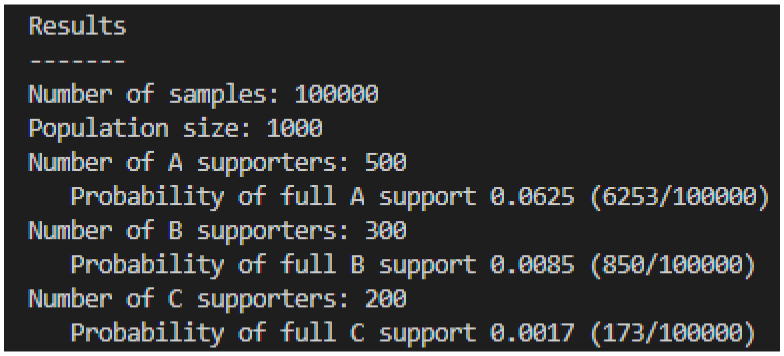
$$\binom{n}{k} = \begin{cases} (-1)^k \binom{-n+k-1}{k} & \text{for } k \geq 0 \\ (-1)^{n-k} \binom{-k-1}{n-k} & \text{for } k \leq n \\ 0 & \text{otherwise} \end{cases}$$

Each of the experiments revolve around selecting *k* unordered outcomes from a total of *n* possible choices, resulting in the binomial coefficient and is also referred to as the combinatorial number. A key distinction from permutations is that our outcomes are unordered and concerned with the selected distinct elements, treating all such subsets as a single combination regardless of order.

Results and Discussion

Experiment One:

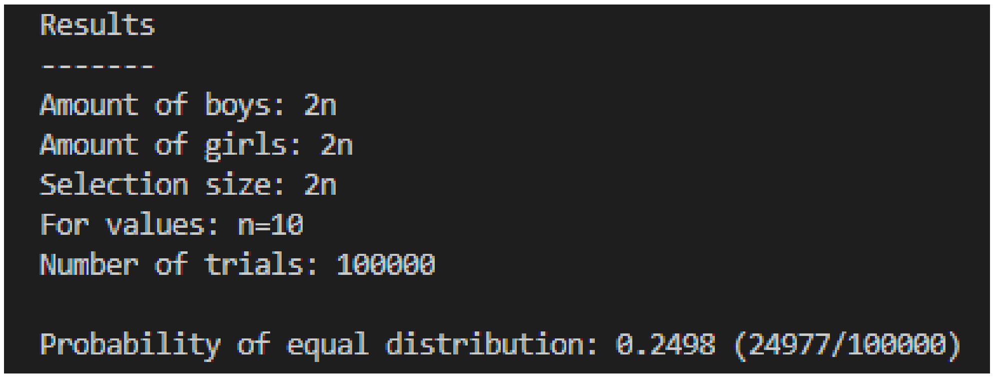




```
Results
-------
Number of samples: 100000
Population size: 1000
Number of A supporters: 500
   Probability of full A support 0.0625 (6253/100000)
Number of B supporters: 300
   Probability of full B support 0.0085 (850/100000)
Number of C supporters: 200
   Probability of full C support 0.0017 (173/100000)
```

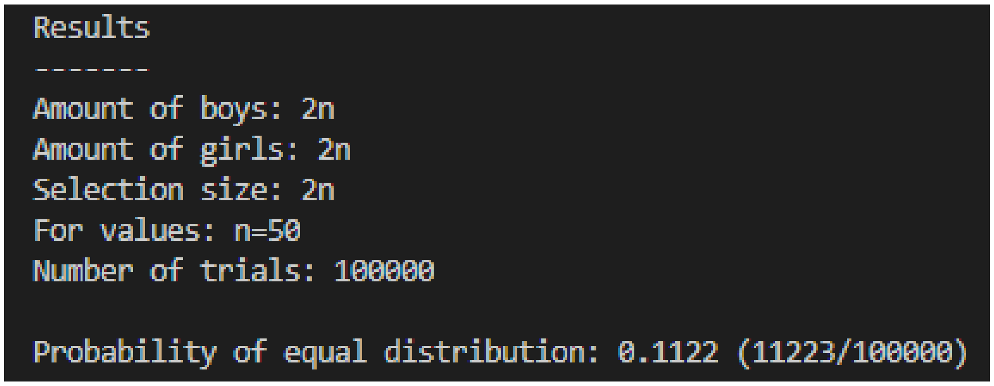
In our total population size of *N=1000*, we see that in descending order *50* percent is comprised of A supporters, *30* percent by B supporters, and *20* percent by C supporters. This is reflected accordingly in the probabilities that a random sample contains unanimous support for a given party. As shown in the graph there is a significant drop off in probability between A and B as well as between B and C since the following parties are a minority and comprise an increasingly smaller amount of the total population.

Experiment Two:



```
Results
-------
Amount of boys: 2n
Amount of girls: 2n
Selection size: 2n
For values: n=10
Number of trials: 100000

Probability of equal distribution: 0.2498 (24977/100000)
```

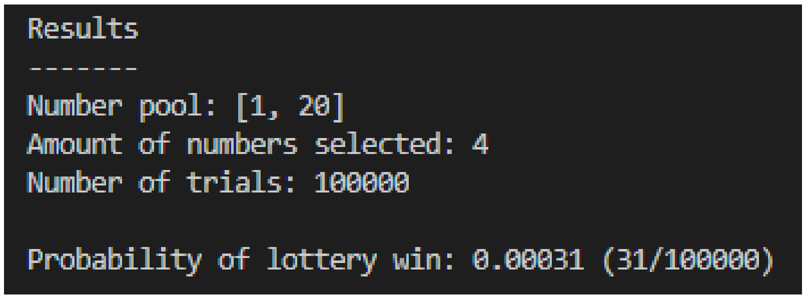


```
Results
-------
Amount of boys: 2n
Amount of girls: 2n
Selection size: 2n
For values: n=50
Number of trials: 100000

Probability of equal distribution: 0.1122 (11223/100000)
```

For the group selection in which we are looking for the probability of selecting a group with an equal distribution of boys and girls we observe a different trend. The same experiment carried out for a value of *n=10* has a higher probability when conducted with the value *n=50* in this case. This was not as expected as initially one might believe that since the scalar values for boys, girls, and selection remain the same, the probability should as well. However, it should instead logically follow that increasing the *n* value in fact increases the size of the sample space, even though distribution ratio remains the same. Therefore, the probability of maintaining said ratio when there are more elements to select from decreases as represented in the comparison here between the two different n values.

Experiment Three:



```
Results
-------
Number pool: [1, 20]
Amount of numbers selected: 4
Number of trials: 100000

Probability of lottery win: 0.00031 (31/100000)
```

In this last experiment we aimed to simulate that for *20* consecutive numbers from *1* to *20* we select four numbers at random from the number set, then another four at random from the same full number set and the probability that the two number sets contain exactly the same numbers, in any order. In other words this is represented by a combination as discussed and in particular there are *C(20, 4) = 4845* possible ways to select *4* numbers from a pool of *20*.

Conclusion

Performing this laboratory was a valuable way in which to apply the core probabilistic concepts of combinatorial analysis with actual simulated experiments. Each of the described experiments may not necessarily be a perfect implementation however they serve to illustrate at least the general magnitude of the phenomena. The implementation logic for each of them is also largely dependent on Python's pseudorandom number generation which on its own is not perfect and is utilized for both randomizing our population as well as randomly sampling it for our elements.

The default amount of trials conducted for each of the experiments is *100,000* however increasing this value should theoretically yield us a more accurate probability value, we may also be able to increase the amount of randomization shuffles performed on the population after initialization, since elements are added in place consecutively and may still partly remain clustered despite multiple shuffling passes. Reshuffling after each insertion for example could be another possible scheme for populating the list fairly. Moreover utilizing Python to run these probalistic simulations is another useful method for sampling the probability.

Appendix

Experiment 1 Implementation:

```
import random
import matplotlib.pyplot as plt

'''
A certain population consists of N=1000 people. 500 of them support party A; 300 of them
support party B; and 200 support party C. A group of 4 people is chosen at random from the
population. What is the probability that all persons in the group support {A, B, C}
'''

# Returns True if the randomly selected supporters all
# hold the same party affiliation, False otherwise
def full_support(sample): return len(set(sample)) == 1

def party_support(N=1000, A=500, B=300, C=200, group=4, trials=100000):
    # Create the population with the given supporter values
    population = []
    for _ in range(A): population.append("A")
    for _ in range(B): population.append("B")
    for _ in range(C): population.append("C")

    # Shuffle the population for randomized distribution
    for _ in range(500): random.shuffle(population)

    # Randomly select group amount from the population, log the amount
    # of times the entire selection contains all A, B, or C supporters
    support_count = {'A' : 0, 'B' : 0, 'C' : 0}
    for _ in range(trials):
        sample = random.sample(population, group)
        # The sample is entirely affiliated, increment its count
        if full_support(sample): support_count[sample[0]] += 1

    # Calculate probabilities
    a_support = round(support_count['A']/trials, 5)
    b_support = round(support_count['B']/trials, 5)
    c_support = round(support_count['C']/trials, 5)

    # Output the sampled probability of each party having unanimous support from our trials
    print("Results\n-------")
    print("Number of samples:", trials)
    print("Population size:", N)
    print("Number of A supporters:", A)
    print("   Probability of full A support", a_support,
        "(" + str(support_count['A']) + "/" + str(trials) + ")")
    print("Number of B supporters:", B)
    print("   Probability of full B support", b_support,
        "("+ str(support_count['B']) + "/" + str(trials) + ")")
    print("Number of C supporters:", C)
    print("   Probability of full C support", c_support,
        "(" + str(support_count['C']) + "/" + str(trials) + ")")

    # Generate plot
    plt.bar(x=['A','B','C'], height=[a_support, b_support, c_support])
    plt.title("Unanimous Party Support For Random Sample of Size="
        + str(group) + " From Population Size N=" + str(N) + " Over Trials=" + st
r(trials))
    plt.xlabel("Party Affiliation")
    plt.ylabel("Probability of Unanimous Support")
    plt.show()

party_support()
```

Experiment 2 Implementation:

```python
'''
A class of 4n children contains 2n boys and 2n girls. A group of 2n children is c
hosen at random.
What is the probability that the group contains an equal number of boys and girls
?
'''
# Returns True if the sample contains an equal number of boys and girls
# RECALL: a "boy" is represented by a boolean value of True
def has_equal(sample):
    counts = {'Boy' : 0, 'Girl': 0}
    for boy in sample:
        if not boy: counts['Girl'] += 1
        else: counts['Boy'] += 1
    return counts['Boy'] == counts['Girl']

def class_select(select_scalar=2, boy_scalar=2, girl_scalar=2, N=10, trials=10000
0):
    # Let True represent a "boy", create population according to the ratios
    population = []
    for _ in range(boy_scalar*N): population.append(True)
    for _ in range(girl_scalar*N): population.append(False)

    # Shuffle the population for randomized distribution
    for _ in range(500): random.shuffle(population)

    # Randomly select select_scalar*N children trials amount of times, log the
    # amount of times the sample contains an equal amount of boys and girls
    equal_count = 0
    for _ in range(trials):
        sample = random.sample(population, (select_scalar*N))
        if has_equal(sample): equal_count += 1
    p_equal = round(equal_count/trials, 5)

    # Output results
    print("Results\n-------")
    print("Amount of boys:", str(boy_scalar) + "n")
    print("Amount of girls:", str(girl_scalar) + "n")
    print("Selection size:", str(select_scalar) + "n")
    print("For values: n=" + str(N))
    print("Number of trials:", trials)
    print("\nProbability of equal distribution:", p_equal,
        "(" + str(equal_count) + "/" + str(trials) + ")")

class_select()
```

Experiment 3 Implementation:

```
'''
In a lottery game, the player picks 4 numbers from a sequence of 1 through 20. At
 lottery
drawing, 4 balls are drawn at random from a box containing 20 balls numbered 1 th
rough 20.
What is the probability that the player will win the lottery (i.e. getting 4 matc
hes in any order)?
'''
def lottery(min_number=1, max_number=20, draw_size=4, trials=100000):
    # Generate randomized number pool
    number_pool = [i for i in range(min_number, max_number+1)]
    for _ in range(500): random.shuffle(number_pool)

    # For each trial, randomly select draw_size numbers for the player, then
    # draw_size numbers representing the the winning numbers
    win_count = 0
    for _ in range(trials):
        player = set(random.sample(number_pool, draw_size))
        winning = set(random.sample(number_pool, draw_size))
        # Sets are unordered, winning numbers can match in any order
        if player == winning: win_count += 1
    p_win = round(win_count/trials, 5)

    # Output results
    print("Results\n-------")
    print("Number pool: [" + str(min_number) + ", " + str(max_number) + "]")
    print("Amount of numbers selected:", draw_size)
    print("Number of trials:", trials)
    print("\nProbability of lottery win:", p_win,
        "(" + str(win_count) + "/" + str(trials) + ")")

lottery()
```