In the name of God

E-Sports teams statistics simulation

Researcher: Mobin Nesari 99222107
Professor: Phd. Dehghan

Fall semester 1400-1401

Introduction:

E-Sports leagues and competition are one of the most famous competition on the planet specially among young people. In these competitions, some famous teams and groups which their network values are even more than one million dollar send their best players for playing in this competitions among other teams. These competitions have great prize pools for winners. Before pandemic and now after vaccination, these matches have more than one hundred million viewers around the world. According to all these facts, simulation and even prediction for this matches and their results convinced scientists to analyses these data more precisely. In this article, I want to calculate and evaluate expected value of how much time does a team need to win a competition.

Problem:

As it mentioned, we want to calculate and evaluate some values, run some simulations and then conclude how much time this team needs to get a win against another team in a match. Now let's see problems information:

We know that probability density function (PDF) is $f(x) = \frac{c}{\sqrt{x}}$. We need to compute and evaluate cumulative distribution function (CDF) and then with some algorithms, simulate many games between these two teams and then conclude that what is expected value for winning rounds for the first team.

Solution:

First of all, we need to calculate c in PDF. We know that for every PDF, we have $\int_{-\infty}^{\infty} f(x)dx = 1$.

According this fact:

$$\int_{-\infty}^{\infty} \frac{c}{\sqrt{x}}dx = 1,(0 < x \leq 15) \implies \int_{0}^{15} \frac{c}{\sqrt{x}}dx = 1 \implies 2c\sqrt{x} = 1,(0 < x \leq 15) \implies 2c\sqrt{15} = 1 \implies c = \frac{1}{2\sqrt{15}}$$

Now PDF will be: $f(x) = \frac{1}{2\sqrt{15x}}$. Now we can calculate CDF.

$$F(x) = \int f(x)dx \implies F(x) = \int \frac{1}{2\sqrt{15x}}dx = \sqrt{\frac{x}{15}}.$$

Now we need inverse of CDF: $y = \sqrt{\frac{x}{15}} \implies y^2 = \frac{x}{15} \implies 15y^2 = x \implies F^{-1}(x) = 15x^2.$

We know that range of CDF is (0,1] and its domain is (0,15]. According to this fact domain of F^{-1} is (0,1] and range of it will be (0,15]. If we generate random uniform distribution numbers in range (0,1] and input to F^{-1}, then it will give us a number between (0,15], we can use this number as a simulated data and calculate or evaluate some valuable information like expected value and variance from it. I implemented a program which we will see later that generate random uniform distribution number between 0 and 1 and then calculate expected value and variance from it. Let's see how generate function works.

Generate function:

For generating random uniform function, I wrote 4 version for it which I will explain why I decided to use another and reprogram it.

Mark 1:

```python
def random_number_generator(counter: int) -> list:  # This function generate random numbers in interval [0,1)
    numbers = []
    for i in range(counter):
        numbers.append((random.random() + 0.0000001) % 1)
    return numbers
```

As you can see, this function gets an integer as counter, initialize an array and generating random numbers between 0 and 1 and at the end return all of them. This function has an obvious problem, numbers are not uniformed. This problem causes not accurate compute and evaluation for further processes.

This problem leads us to next generation of random number generator function.

Mark 2:

```
def random_number_generator_mk2(counter: int):
    numbers = np.linspace(0.0, 1.0, counter, endpoint=True)
    return numbers
```

This function uses a third party library which is the most famous library among data scientists, Numpy. This function makes numbers between 0 and 1 and it works great. But the problem is I need my numbers to be a normal array not a numpy array. Because of this problem and some other issues I changed the function to another one.

Mark 3:

```
def random_number_generator_mk3(counter: int):
    numbers = np.random.uniform(0, 1, counter)
    return numbers
```

This function like last one, uses library numpy. np.random.uniform(0, 1, counter) is a function which generate random uniform numbers between 0 and 1. But after testing its output, I figured out that they are not well uniformed and this function has same problem as last one. These problems and hours of debugging ends into this idea that "I need to create this function myself.". This quote leads us to last and main function, mark 4.

Mark 4:

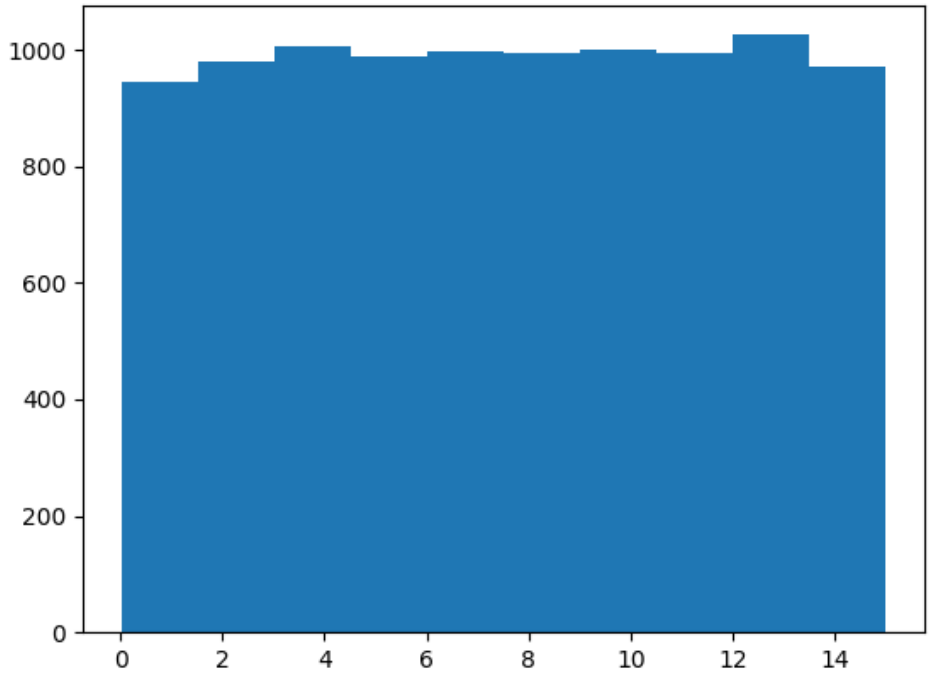
```
def random_number_generator_mk4(counter: int):
    zone = counter // 15
    numbers = []
    for i in range(0, 15):
        for j in range(zone):
            k = random.uniform(math.sqrt(i/15), math.sqrt((i+1)/15))
            if probability_density_function(inverse_cumulative_function(k)) <= 1:
                numbers.append(k)
            else:
                j -= 1

    return numbers
```

This function is main one. I designed and developed this function myself, just to make sure that all random numbers evenly distributed between intervals. My algorithm for this function is to divide all numbers into different zones. Then generate numbers between $[\sqrt{\frac{i}{15}}, \sqrt{\frac{i+1}{15}}]$ for $i \in [0,15)$. After testing this function for more than 1000 cases, I saw in almost 100% of cases, numbers are completely uniformed.

```
/Users/mobinnesari/PycharmProjects/StaticSimulation/venv/bin/python /Users/mobinnesari/PycharmProjects/StaticSimulation/main.py
defaultdict(None, {0: 572, 1: 666, 2: 666, 3: 666, 4: 666, 5: 666, 6: 666, 7: 666, 8: 666, 9: 666, 10: 666, 11: 666, 12: 666, 13: 666, 14: 666, 15: 0})
```

random_number_generator_mk4 distribution test



Histogram of random data between (0,15]

Mathematic functions:

In this program, I implemented PDF and CDF and it's inverse for making code easier to understand and observance of clean code principles. Let’s take a quick look at them:

```python
def probability_density_function(x: float) -> float:  # Probability density function  (0,15] -> [0,1]
    return (1 / (2 * math.sqrt(15))) * (1 / math.sqrt(x))
```

Probability Density Function

```python
def cumulative_function(x: float) -> float:  # Cumulative function: [0,15] -> [0,1]
    return math.sqrt(x / 15)
```

Cumulative Distribution Function

```python
def inverse_cumulative_function(x: float) -> float:  # Inverse cumulative function: [0,1] -> [0,15]
    return 15 * x ** 2
```

Inverse of Cumulative Distribution Function

```python
def expected_value_computer(density: list, data: list) -> float:
    answer = 0
    for i in range(len(density)):
        answer += density[i] * data[i]
    answer /= len(density) / 15
    return answer
```

Expected Value Computation function

```python
def second_order_expected_value(density: list, data: list) -> float:
    answer = 0
    for i in range(len(density)):
        answer += density[i] * data[i] ** 2
    answer /= len(density) / 15
    return answer
```

Second Order Expected Value Computation function

```python
def variance_computer(density: list, data: list) -> float:
    answer = second_order_expected_value(density, data) - expected_value_computer(density, data) ** 2
    return answer
```

Variance Computation function

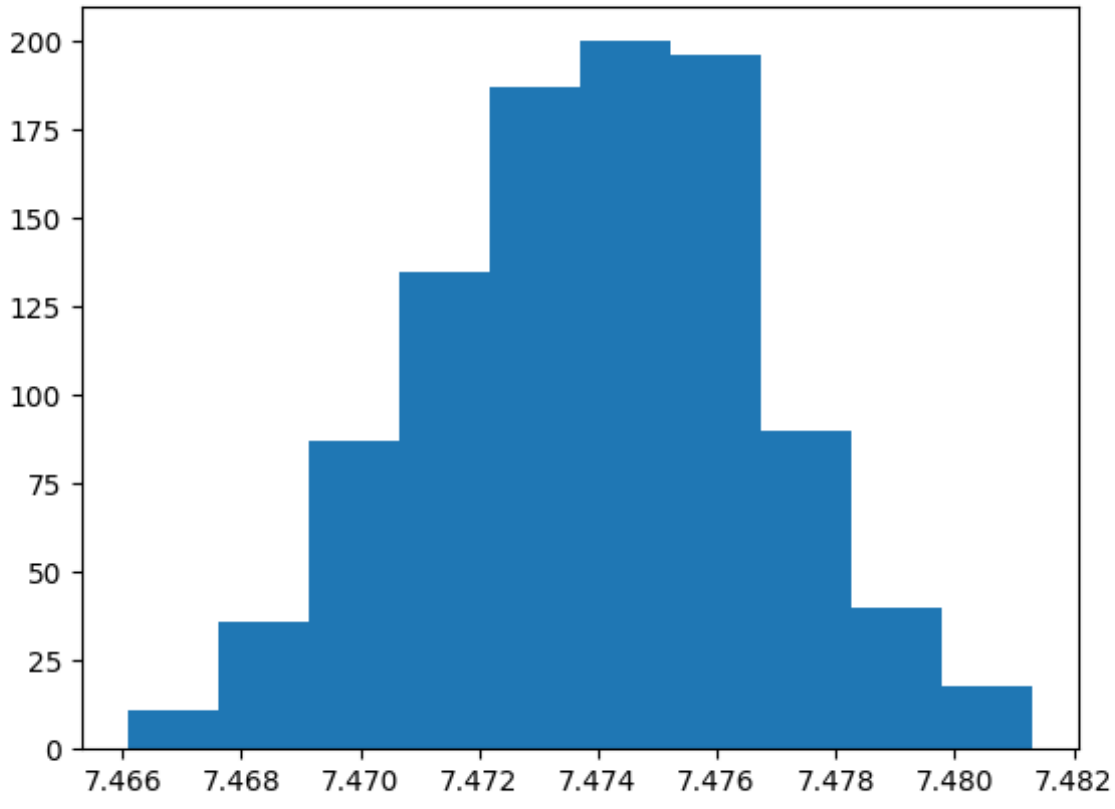
```python
def mean_data_generator(repeat_time: int, data_length: int) -> list:
    answer = []
    for i in range(repeat_time):
        arr = generate_artificial_data(data_length)
        answer.append(sum(arr)/data_length)
    return answer


def means_expected_value_computer(means: list) -> float:
    answer = 0
    for i in means:
        answer += i * means.count(i)/len(means)
    return answer


def means_second_order_expected_value(means: list) -> float:
    answer = 0
    for i in means:
        answer += (i ** 2) * (means.count(i)/len(means))
    return answer


def means_variance(means: list) -> float:
    answer = means_second_order_expected_value(means) - means_expected_value_computer(means) ** 2
    return answer
```

Same function for means of artificial data



Histogram of 1000 set of artificial data mean

Theoretical and Experimental results:

In this section, I want to show you the difference between experimental and theoretical results for this problem. First lets see theoretical results which we will expect it from experimental in the future:

Theoretical:

We want to see expected value, second order expected value and variance of a set of artificial data.

For continuous data, we knew $E(x) = \int_{-\infty}^{\infty} x f(x) dx$. According to this fact:

$$E(x) = \int_{-\infty}^{\infty} x \times \frac{1}{2\sqrt{15x}} dx \implies E(x) = \int_{-\infty}^{\infty} \frac{\sqrt{x}}{2\sqrt{15}} \implies E(x) = \frac{\sqrt{x^3}}{3\sqrt{15}}, x \in (0,15] \implies E(x) = 5 \blacksquare$$

Now for second order expected value:

$$E(x^2) = \int_{-\infty}^{\infty} x^2 \times \frac{1}{2\sqrt{15x}} dx \implies E(x^2) = \int_{-\infty}^{\infty} \frac{\sqrt{x^3}}{2\sqrt{15}} \implies E(x^2) = \frac{\sqrt{x^5}}{3\sqrt{15}}, x \in (0,15] \implies E(x^2) = 45 \blacksquare$$

We know $Var(x) = E(x^2) - E(x)^2$. According to this fact: $Var(x) = E(x^2) - E(x)^2 \implies Var(x) = 45 - 25 = 20 \blacksquare$

For means of number of artificial data, we have: $E(\bar{x}) = E(x) = 5, Var(\bar{x}) = \frac{Var(x)}{n} = \frac{20}{1000} = 0.02 \blacksquare$

Experimental:

Now let's take a look at numbers which have been computed with some artificial data:

```
Expected value:  5.017622442167758
Second order expected value:  45.320259115286085
Variance:  20.143724143140552
Means expected value:  7.4738792429153165
Means second order expected value:  55.85887944493789
Means variance:  8.507257469148044e-06
```

Experimental Data

Conclusion:

According to experimental results, we can say that implemented code and algorithm works precisely on just one set of artificial data. But when we want to compute expected value over means of 1000 artificial data, then we can see that resulted numbers are pretty inaccurate. In my opinion, the main reason of this inaccuracy is small error percentages for every data in a set of artificial data which when we sum all these errors, this may cause this inaccuracy. On the other hand, I think it's mean is pretty accurate, because when we generate some uniform distribution among numbers 0 to 15, then mean of it need to be equal to 7.5. This fact leads us to paradox which I think is a great example of "Sometimes real world doesn't observance theoretical rules.".