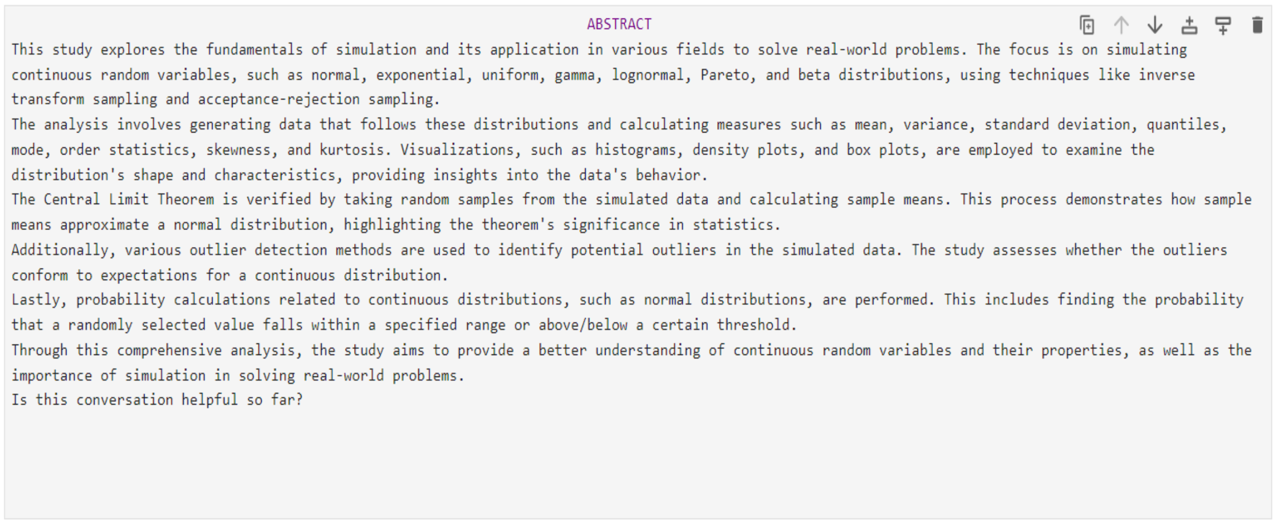
Final project

Uday kurella

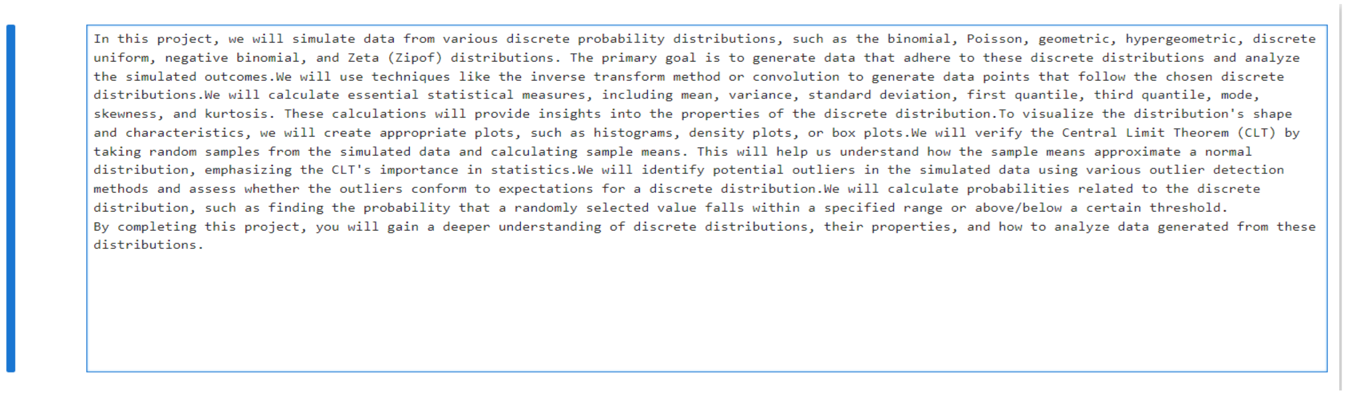
20022819

Ma 540 project progress:



ABSTRACT

This study explores the fundamentals of simulation and its application in various fields to solve real-world problems. The focus is on simulating continuous random variables, such as normal, exponential, uniform, gamma, lognormal, Pareto, and beta distributions, using techniques like inverse transform sampling and acceptance-rejection sampling.
The analysis involves generating data that follows these distributions and calculating measures such as mean, variance, standard deviation, quantiles, mode, order statistics, skewness, and kurtosis. Visualizations, such as histograms, density plots, and box plots, are employed to examine the distribution's shape and characteristics, providing insights into the data's behavior.
The Central Limit Theorem is verified by taking random samples from the simulated data and calculating sample means. This process demonstrates how sample means approximate a normal distribution, highlighting the theorem's significance in statistics.
Additionally, various outlier detection methods are used to identify potential outliers in the simulated data. The study assesses whether the outliers conform to expectations for a continuous distribution.
Lastly, probability calculations related to continuous distributions, such as normal distributions, are performed. This includes finding the probability that a randomly selected value falls within a specified range or above/below a certain threshold.
Through this comprehensive analysis, the study aims to provide a better understanding of continuous random variables and their properties, as well as the importance of simulation in solving real-world problems.
Is this conversation helpful so far?



In this project, we will simulate data from various discrete probability distributions, such as the binomial, Poisson, geometric, hypergeometric, discrete uniform, negative binomial, and Zeta (Zipof) distributions. The primary goal is to generate data that adhere to these discrete distributions and analyze the simulated outcomes.We will use techniques like the inverse transform method or convolution to generate data points that follow the chosen discrete distributions.We will calculate essential statistical measures, including mean, variance, standard deviation, first quantile, third quantile, mode, skewness, and kurtosis. These calculations will provide insights into the properties of the discrete distribution.To visualize the distribution's shape and characteristics, we will create appropriate plots, such as histograms, density plots, or box plots.We will verify the Central Limit Theorem (CLT) by taking random samples from the simulated data and calculating sample means. This will help us understand how the sample means approximate a normal distribution, emphasizing the CLT's importance in statistics.We will identify potential outliers in the simulated data using various outlier detection methods and assess whether the outliers conform to expectations for a discrete distribution.We will calculate probabilities related to the discrete distribution, such as finding the probability that a randomly selected value falls within a specified range or above/below a certain threshold.
By completing this project, you will gain a deeper understanding of discrete distributions, their properties, and how to analyze data generated from these distributions.

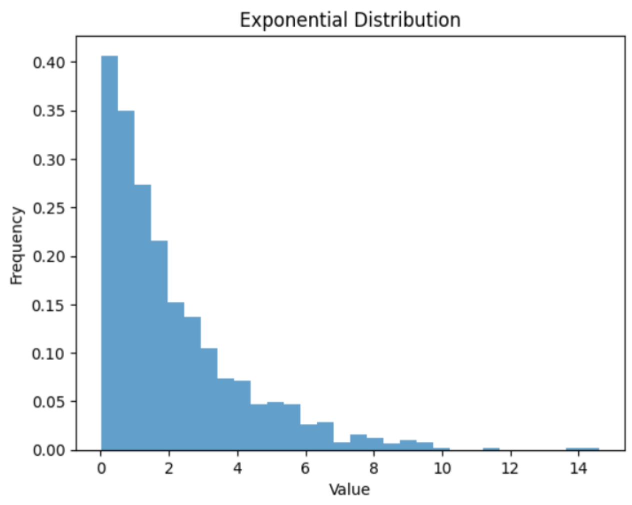
Methodology Analysis and results

Statistical analysis

```python
[1]: import numpy as np
import matplotlib.pyplot as plt
def exponential_random_variable(lam, size=1000):
    u = np.random.uniform(0, 1, size)
    x = -np.log(1 - u) / lam
    return x
lam = 0.5
data = exponential_random_variable(lam)
mean = np.mean(data)
variance = np.var(data)
std_dev = np.std(data)
first_quantile = np.percentile(data, 25)
third_quantile = np.percentile(data, 75)
mode = 1 / lam
skewness = 2
kurtosis = 6
print("Mean:", mean)
print("Variance:", variance)
print("Standard Deviation:", std_dev)
print("First Quantile (25th percentile):", first_quantile)
print("Third Quantile (75th percentile):", third_quantile)
print("Mode:", mode)
print("Skewness:", skewness)
print("Kurtosis:", kurtosis)
plt.hist(data, bins=30, density=True, alpha=0.7)
plt.title("Exponential Distribution")
plt.xlabel("Value")
plt.ylabel("Frequency")
plt.show()
```

Out put:

```
Mean: 2.1112738660479233
Variance: 4.134708814070413
Standard Deviation: 2.0333983412185654
First Quantile (25th percentile): 0.630303190396995
Third Quantile (75th percentile): 2.9540585690228585
Mode: 2.0
Skewness: 2
Kurtosis: 6
```



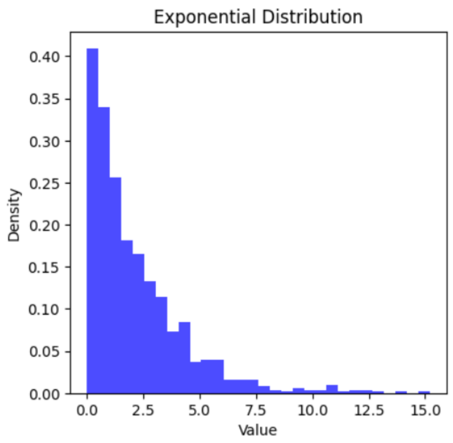
Visualization:

Visualization of exponential distribution:

[2]:

```python
import numpy as np
import matplotlib.pyplot as plt
from scipy.stats import gamma, lognorm, pareto, beta
def exponential_random_variable(lam, size=1000):
    u = np.random.uniform(0, 1, size)
    x = -np.log(1 - u) / lam
    return x
exp_lam = 0.5
exp_data = exponential_random_variable(exp_lam)
plt.figure(figsize=(12, 8))

plt.subplot(2, 3, 1)
plt.hist(exp_data, bins=30, density=True, color='blue', alpha=0.7)
plt.title('Exponential Distribution')
plt.xlabel('Value')
plt.ylabel('Density')
plt.tight_layout()
plt.show()
```

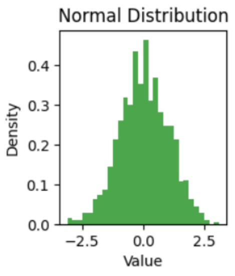
Output:



Visualization of normal distribution:

[4]:

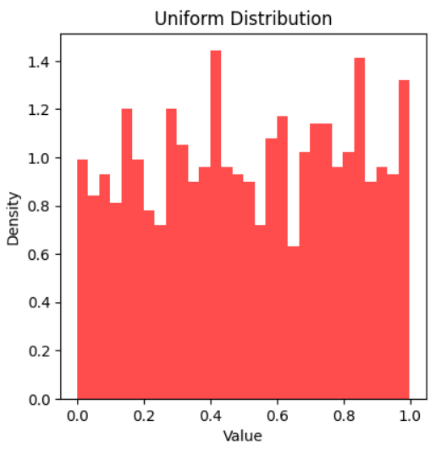
```python
import numpy as np
import matplotlib.pyplot as plt
from scipy.stats import gamma, lognorm, pareto, beta
def normal_random_variable(mu, sigma, size=1000):
    return np.random.normal(mu, sigma, size)
normal_sigma = 1
normal_data = normal_random_variable(normal_mu, normal_sigma)
plt.subplot(2, 3, 2)
plt.hist(normal_data, bins=30, density=True, color='green', alpha=0.7)
plt.title('Normal Distribution')
plt.xlabel('Value')
plt.ylabel('Density')
plt.tight_layout()
plt.show()
```



Visualization of uniform distribution:

[5]:
```
import numpy as np
import matplotlib.pyplot as plt
from scipy.stats import gamma, lognorm, pareto, beta
def uniform_random_variable(a, b, size=1000):
    return np.random.uniform(a, b, size)
uniform_a = 0
uniform_b = 1
uniform_data = uniform_random_variable(uniform_a, uniform_b)
plt.figure(figsize=(12, 8))

plt.subplot(2, 3, 3)
plt.hist(uniform_data, bins=30, density=True, color='red', alpha=0.7)
plt.title('Uniform Distribution')
plt.xlabel('Value')
plt.ylabel('Density')
plt.tight_layout()
plt.show()
```

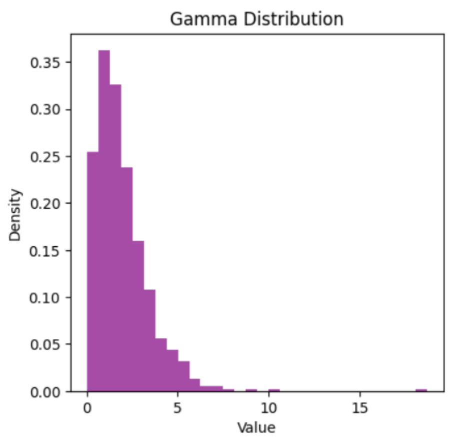
output:



visualization of gamma distribution:

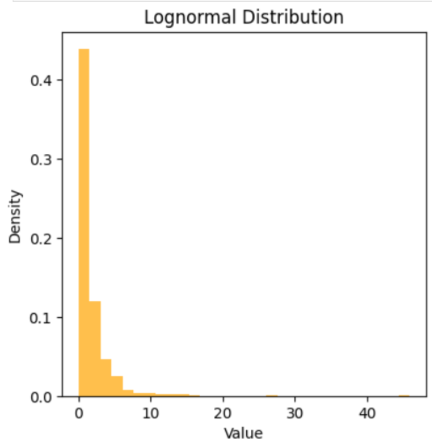
[7]:
```
import numpy as np
import matplotlib.pyplot as plt
from scipy.stats import gamma, lognorm, pareto, beta
def gamma_random_variable(shape, scale, size=1000):
    return np.random.gamma(shape, scale, size)
gamma_shape = 2
gamma_scale = 1
gamma_data = gamma_random_variable(gamma_shape, gamma_scale)
plt.figure(figsize=(12, 8))
plt.subplot(2, 3, 4)
plt.hist(gamma_data, bins=30, density=True, color='purple', alpha=0.7)
plt.title('Gamma Distribution')
plt.xlabel('Value')
plt.ylabel('Density')

plt.tight_layout()
plt.show()
```

output:



visualization of lognormal distribution

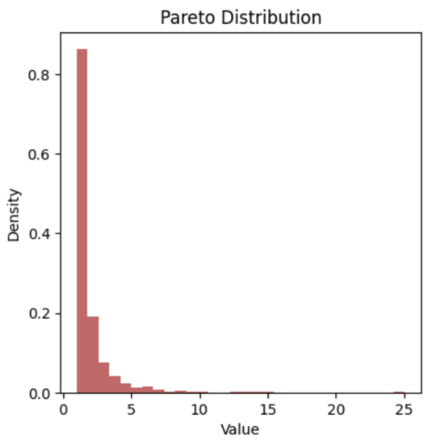
```
[9]: import numpy as np
import matplotlib.pyplot as plt
from scipy.stats import gamma, lognorm, pareto, beta
def lognormal_random_variable(mu, sigma, size=1000):
    return np.random.lognormal(mu, sigma, size)
lognorm_mu = 0
lognorm_sigma = 1
lognorm_data = lognormal_random_variable(lognorm_mu, lognorm_sigma)
plt.figure(figsize=(12, 8))
plt.subplot(2, 3, 5)
plt.hist(lognorm_data, bins=30, density=True, color='orange', alpha=0.7)
plt.title('Lognormal Distribution')
plt.xlabel('Value')
plt.ylabel('Density')
plt.tight_layout()
plt.show()
```



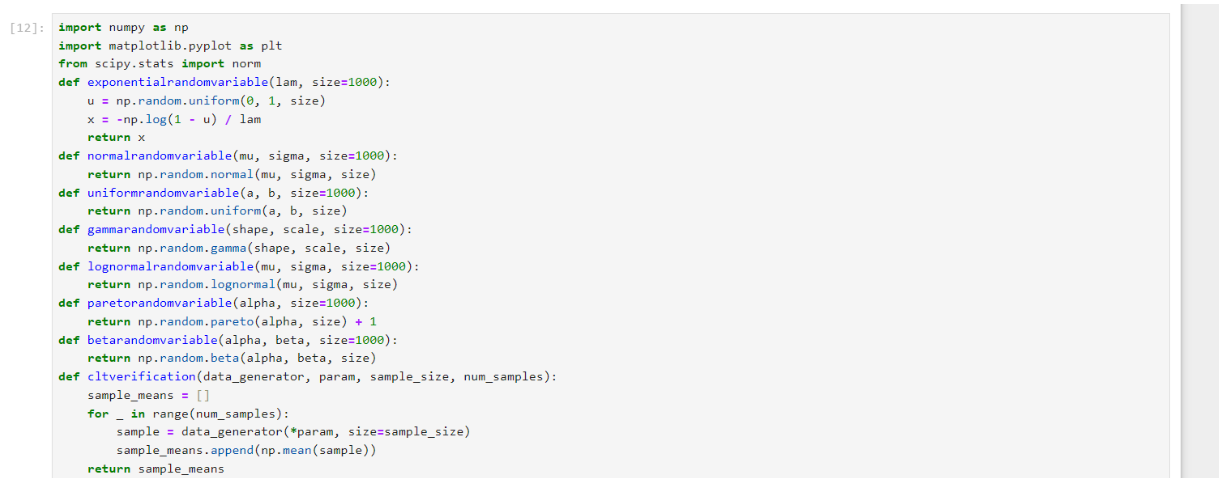
visualization of pareto distribution:

```
[10]: import numpy as np
import matplotlib.pyplot as plt
from scipy.stats import gamma, lognorm, pareto, beta
def pareto_random_variable(alpha, size=1000):
    return np.random.pareto(alpha, size) + 1
pareto_alpha = 2
pareto_data = pareto_random_variable(pareto_alpha)
plt.figure(figsize=(12, 8))
plt.subplot(2, 3, 6)
plt.hist(pareto_data, bins=30, density=True, color='brown', alpha=0.7)
plt.title('Pareto Distribution')
plt.xlabel('Value')
plt.ylabel('Density')
plt.tight_layout()
plt.show()
```

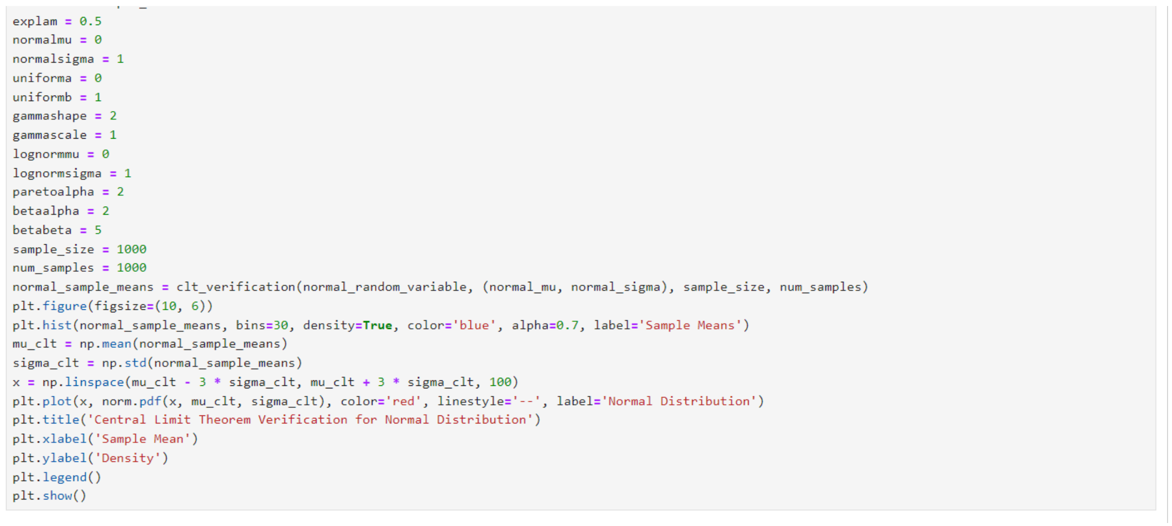
output:



central limit theorem verification for normal distribution:



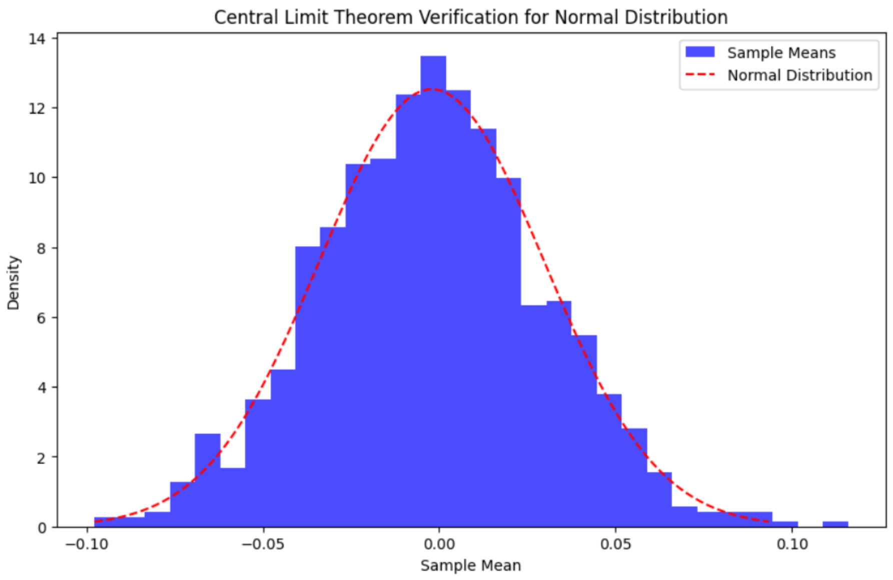
```
[12]: import numpy as np
      import matplotlib.pyplot as plt
      from scipy.stats import norm
      def exponentialrandomvariable(lam, size=1000):
          u = np.random.uniform(0, 1, size)
          x = -np.log(1 - u) / lam
          return x
      def normalrandomvariable(mu, sigma, size=1000):
          return np.random.normal(mu, sigma, size)
      def uniformrandomvariable(a, b, size=1000):
          return np.random.uniform(a, b, size)
      def gammarandomvariable(shape, scale, size=1000):
          return np.random.gamma(shape, scale, size)
      def lognormalrandomvariable(mu, sigma, size=1000):
          return np.random.lognormal(mu, sigma, size)
      def paretorandomvariable(alpha, size=1000):
          return np.random.pareto(alpha, size) + 1
      def betarandomvariable(alpha, beta, size=1000):
          return np.random.beta(alpha, beta, size)
      def cltverification(data_generator, param, sample_size, num_samples):
          sample_means = []
          for _ in range(num_samples):
              sample = data_generator(*param, size=sample_size)
              sample_means.append(np.mean(sample))
          return sample_means
```



```
explam = 0.5
normalmu = 0
normalsigma = 1
uniforma = 0
uniformb = 1
gammashape = 2
gammascale = 1
lognormmu = 0
lognormsigma = 1
paretoalpha = 2
betaalpha = 2
betabeta = 5
sample_size = 1000
num_samples = 1000
normal_sample_means = clt_verification(normal_random_variable, (normal_mu, normal_sigma), sample_size, num_samples)
plt.figure(figsize=(10, 6))
plt.hist(normal_sample_means, bins=30, density=True, color='blue', alpha=0.7, label='Sample Means')
mu_clt = np.mean(normal_sample_means)
sigma_clt = np.std(normal_sample_means)
x = np.linspace(mu_clt - 3 * sigma_clt, mu_clt + 3 * sigma_clt, 100)
plt.plot(x, norm.pdf(x, mu_clt, sigma_clt), color='red', linestyle='--', label='Normal Distribution')
plt.title('Central Limit Theorem Verification for Normal Distribution')
plt.xlabel('Sample Mean')
plt.ylabel('Density')
plt.legend()
plt.show()
```

output:

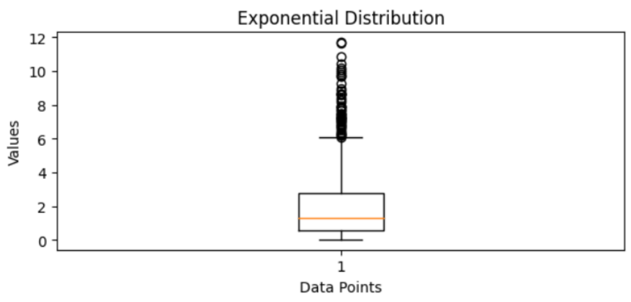
o



outlier detection:

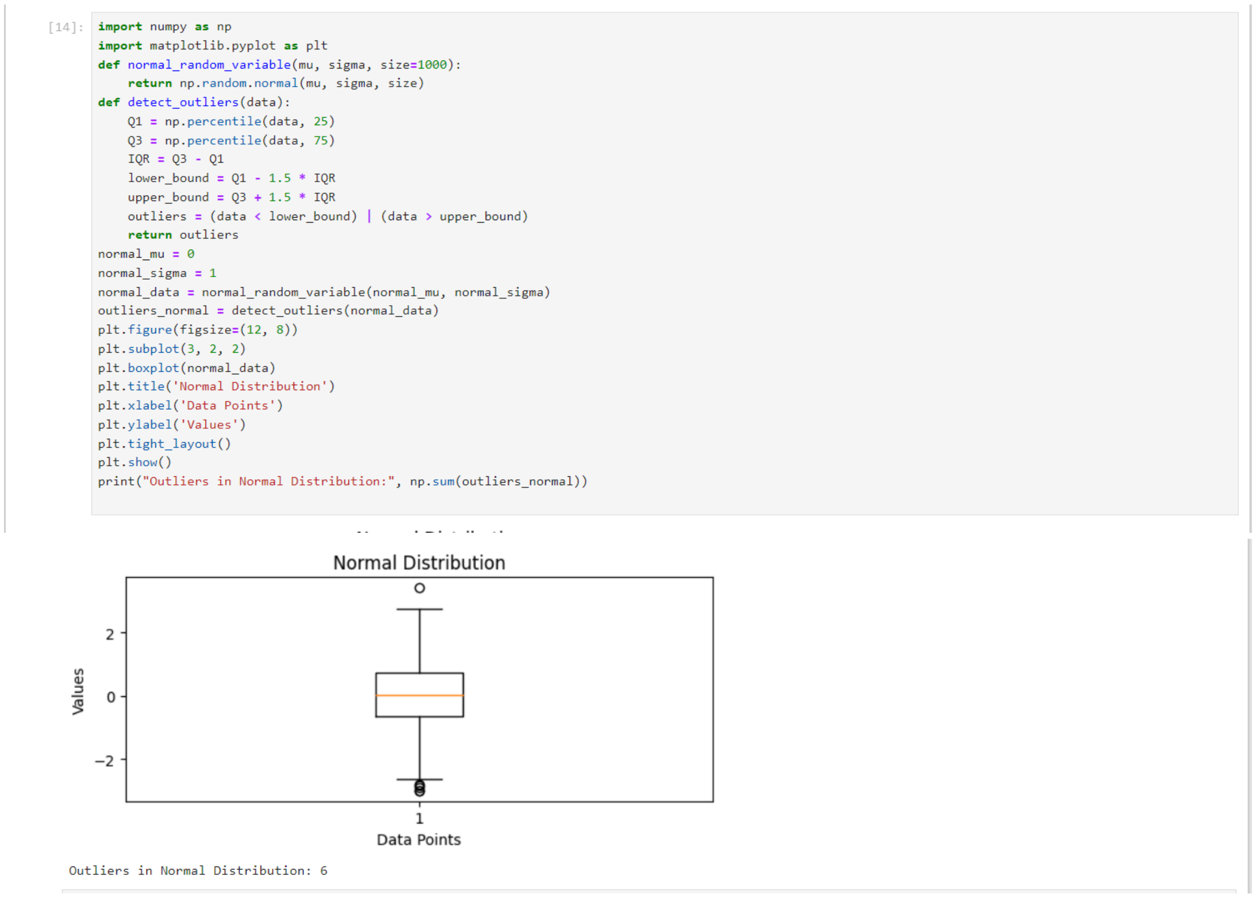
Outliers in exponential distribution:

```
[13]: import numpy as np
import matplotlib.pyplot as plt
def exponential_random_variable(lam, size=1000):
    u = np.random.uniform(0, 1, size)
    x = -np.log(1 - u) / lam
    return x
def detect_outliers(data):
    Q1 = np.percentile(data, 25)
    Q3 = np.percentile(data, 75)
    IQR = Q3 - Q1
    lower_bound = Q1 - 1.5 * IQR
    upper_bound = Q3 + 1.5 * IQR
    outliers = (data < lower_bound) | (data > upper_bound)
    return outliers
exp_lam = 0.5
exp_data = exponential_random_variable(exp_lam)
outliers_exp = detect_outliers(exp_data)
plt.figure(figsize=(12, 8))
plt.subplot(3, 2, 1)
plt.boxplot(exp_data)
plt.title('Exponential Distribution')
plt.xlabel('Data Points')
plt.ylabel('Values')
plt.tight_layout()
plt.show()
print("Outliers in Exponential Distribution:", np.sum(outliers_exp))
```



```
Outliers in Exponential Distribution: 51
```

outliers in normal distribution:

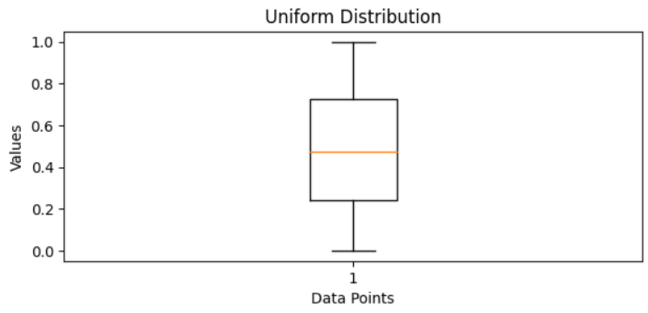


```
[14]: import numpy as np
      import matplotlib.pyplot as plt
      def normal_random_variable(mu, sigma, size=1000):
          return np.random.normal(mu, sigma, size)
      def detect_outliers(data):
          Q1 = np.percentile(data, 25)
          Q3 = np.percentile(data, 75)
          IQR = Q3 - Q1
          lower_bound = Q1 - 1.5 * IQR
          upper_bound = Q3 + 1.5 * IQR
          outliers = (data < lower_bound) | (data > upper_bound)
          return outliers
      normal_mu = 0
      normal_sigma = 1
      normal_data = normal_random_variable(normal_mu, normal_sigma)
      outliers_normal = detect_outliers(normal_data)
      plt.figure(figsize=(12, 8))
      plt.subplot(3, 2, 2)
      plt.boxplot(normal_data)
      plt.title('Normal Distribution')
      plt.xlabel('Data Points')
      plt.ylabel('Values')
      plt.tight_layout()
      plt.show()
      print("Outliers in Normal Distribution:", np.sum(outliers_normal))
```

```
Outliers in Normal Distribution: 6
```

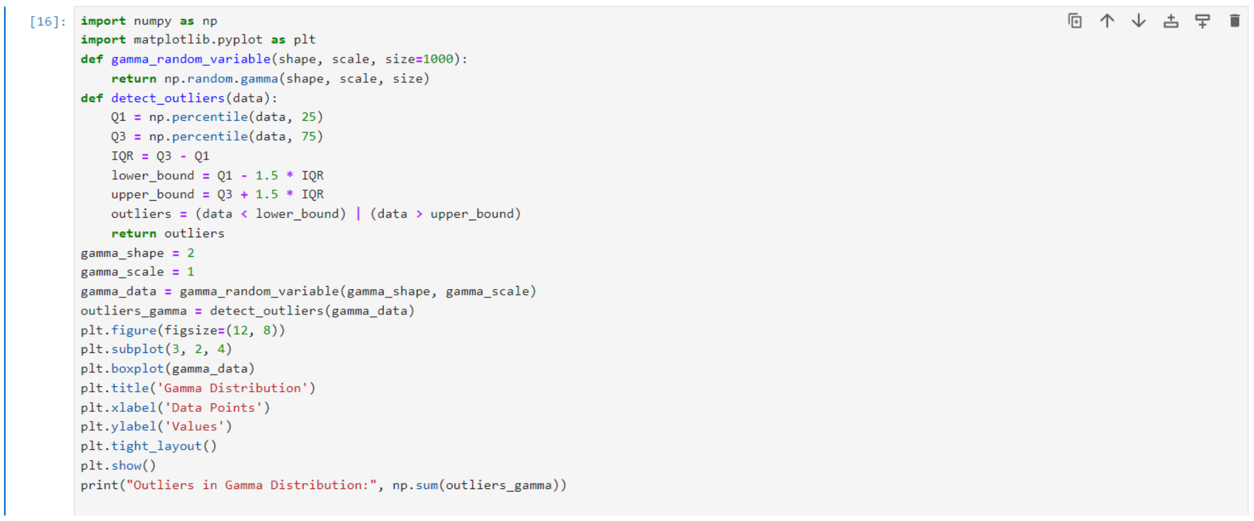
outliers of uniform distribution:

[15]:
```python
import numpy as np
import matplotlib.pyplot as plt
def exponential_random_variable(lam, size=1000):
    u = np.random.uniform(0, 1, size)
    x = -np.log(1 - u) / lam
    return x
def uniform_random_variable(a, b, size=1000):
    return np.random.uniform(a, b, size)
def detect_outliers(data):
    Q1 = np.percentile(data, 25)
    Q3 = np.percentile(data, 75)
    IQR = Q3 - Q1
    lower_bound = Q1 - 1.5 * IQR
    upper_bound = Q3 + 1.5 * IQR
    outliers = (data < lower_bound) | (data > upper_bound)
    return outliers
uniform_a = 0
uniform_b = 1
uniform_data = uniform_random_variable(uniform_a, uniform_b)
outliers_uniform = detect_outliers(uniform_data)
plt.figure(figsize=(12, 8))
plt.subplot(3, 2, 3)
plt.boxplot(uniform_data)
plt.title('Uniform Distribution')
plt.xlabel('Data Points')
plt.ylabel('Values')
plt.tight_layout()
plt.show()
print("Outliers in Uniform Distribution:", np.sum(outliers_uniform))
```

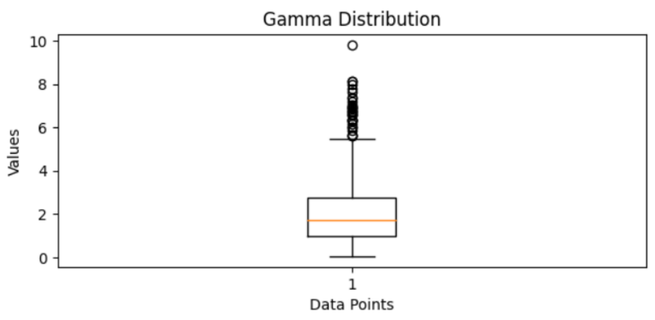


```
Outliers in Uniform Distribution: 0
```

outliers of gamma distributions:

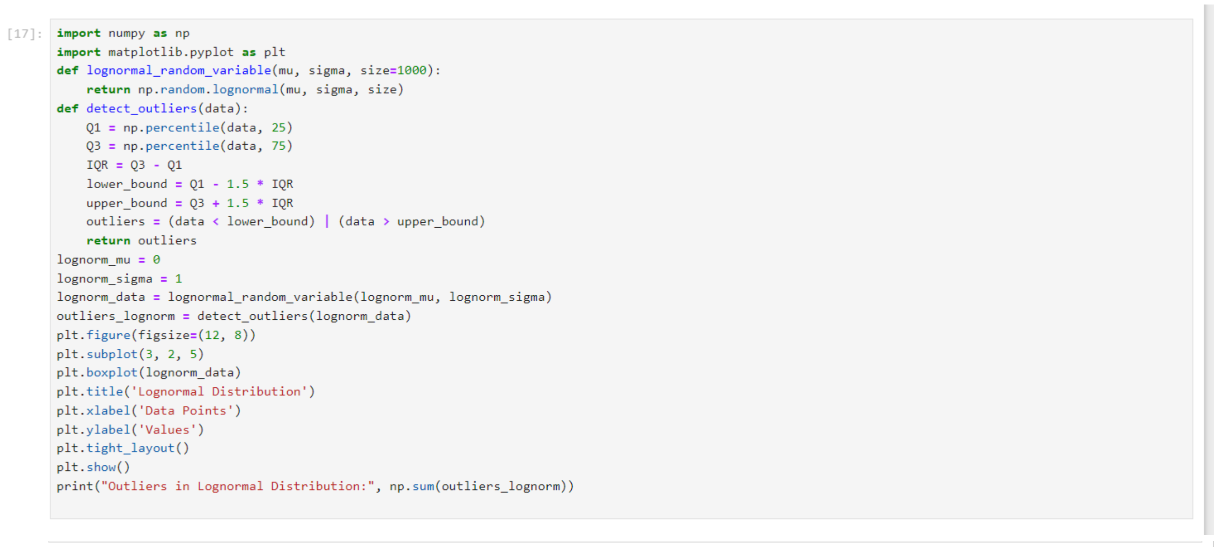


```
[16]: import numpy as np
      import matplotlib.pyplot as plt
      def gamma_random_variable(shape, scale, size=1000):
          return np.random.gamma(shape, scale, size)
      def detect_outliers(data):
          Q1 = np.percentile(data, 25)
          Q3 = np.percentile(data, 75)
          IQR = Q3 - Q1
          lower_bound = Q1 - 1.5 * IQR
          upper_bound = Q3 + 1.5 * IQR
          outliers = (data < lower_bound) | (data > upper_bound)
          return outliers
      gamma_shape = 2
      gamma_scale = 1
      gamma_data = gamma_random_variable(gamma_shape, gamma_scale)
      outliers_gamma = detect_outliers(gamma_data)
      plt.figure(figsize=(12, 8))
      plt.subplot(3, 2, 4)
      plt.boxplot(gamma_data)
      plt.title('Gamma Distribution')
      plt.xlabel('Data Points')
      plt.ylabel('Values')
      plt.tight_layout()
      plt.show()
      print("Outliers in Gamma Distribution:", np.sum(outliers_gamma))
```

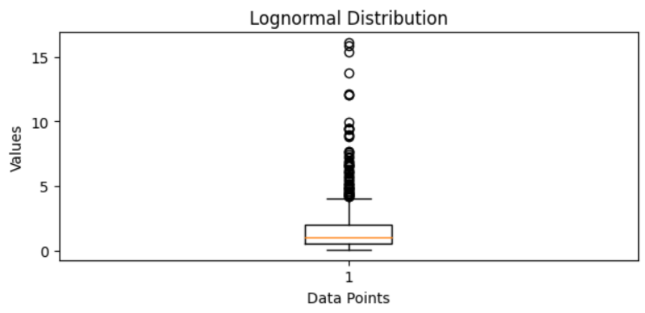


```
Outliers in Gamma Distribution: 31
```

outliers of lognormal distribution:



```
[17]: import numpy as np
      import matplotlib.pyplot as plt
      def lognormal_random_variable(mu, sigma, size=1000):
          return np.random.lognormal(mu, sigma, size)
      def detect_outliers(data):
          Q1 = np.percentile(data, 25)
          Q3 = np.percentile(data, 75)
          IQR = Q3 - Q1
          lower_bound = Q1 - 1.5 * IQR
          upper_bound = Q3 + 1.5 * IQR
          outliers = (data < lower_bound) | (data > upper_bound)
          return outliers
      lognorm_mu = 0
      lognorm_sigma = 1
      lognorm_data = lognormal_random_variable(lognorm_mu, lognorm_sigma)
      outliers_lognorm = detect_outliers(lognorm_data)
      plt.figure(figsize=(12, 8))
      plt.subplot(3, 2, 5)
      plt.boxplot(lognorm_data)
      plt.title('Lognormal Distribution')
      plt.xlabel('Data Points')
      plt.ylabel('Values')
      plt.tight_layout()
      plt.show()
      print("Outliers in Lognormal Distribution:", np.sum(outliers_lognorm))
```



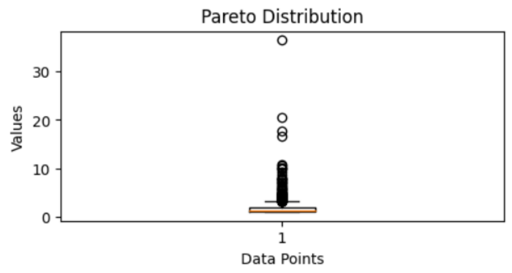
```
Outliers in Lognormal Distribution: 67
```

outliers of pareto distribution:

```
Outliers in Lognormal Distribution: 67
```

```
[18]: import numpy as np
import matplotlib.pyplot as plt
def pareto_random_variable(alpha, size=1000):
    return np.random.pareto(alpha, size) + 1
def detect_outliers(data):
    Q1 = np.percentile(data, 25)
    Q3 = np.percentile(data, 75)
    IQR = Q3 - Q1
    lower_bound = Q1 - 1.5 * IQR
    upper_bound = Q3 + 1.5 * IQR
    outliers = (data < lower_bound) | (data > upper_bound)
    return outliers
pareto_alpha = 2
pareto_data = pareto_random_variable(pareto_alpha)
outliers_pareto = detect_outliers(pareto_data)
plt.figure(figsize=(12, 8))
plt.subplot(3, 2, 6)
plt.boxplot(pareto_data)
plt.title('Pareto Distribution')
plt.xlabel('Data Points')
plt.ylabel('Values')
print("Outliers in Pareto Distribution:", np.sum(outliers_pareto))
```

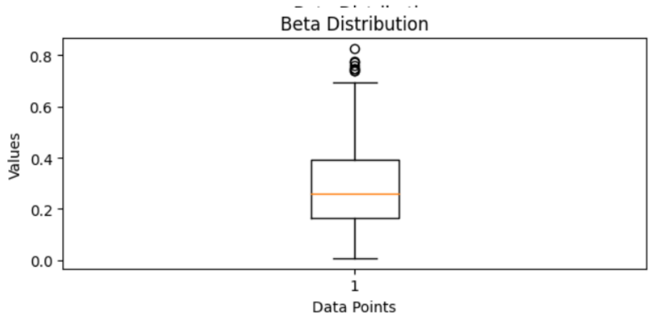
```
Outliers in Pareto Distribution: 94
```



```
[ ]:
```

outliers of beta distribution:

```
[20]: import numpy as np
import matplotlib.pyplot as plt

def beta_random_variable(alpha, beta, size=1000):
    return np.random.beta(alpha, beta, size)

def detect_outliers(data):
    Q1 = np.percentile(data, 25)
    Q3 = np.percentile(data, 75)
    IQR = Q3 - Q1
    lower_bound = Q1 - 1.5 * IQR
    upper_bound = Q3 + 1.5 * IQR
    outliers = (data < lower_bound) | (data > upper_bound)
    return outliers

beta_alpha = 2
beta_beta = 5
beta_data = beta_random_variable(beta_alpha, beta_beta)
outliers_beta = detect_outliers(beta_data)

plt.figure(figsize=(12, 8))
plt.subplot(3, 2, 5) # Change this line to a valid position number
plt.boxplot(beta_data)
plt.title('Beta Distribution')
plt.xlabel('Data Points')
plt.ylabel('Values')
plt.tight_layout()
plt.show()

print("Outliers in Beta Distribution:", np.sum(outliers_beta))
```



```
Outliers in Beta Distribution: 8
```

probability calculations:

```
•[22]: import numpy as np
from scipy.stats import norm
def calculate_normal_probability(mean, std_dev, threshold, direction='below'):
    if direction == 'below':
        probability = norm.cdf(threshold, loc=mean, scale=std_dev)
    elif direction == 'above':
        probability = 1 - norm.cdf(threshold, loc=mean, scale=std_dev)
    else:
        raise ValueError("Invalid direction. Choose 'below' or 'above'.")
    return probability
mean = 0
std_dev = 1
threshold_1 = 1
probability_below_1 = calculate_normal_probability(mean, std_dev, threshold_1, direction='below')
print("Probability of a value below", threshold_1, "in the normal distribution:", probability_below_1)
threshold_2 = -1
probability_below_2 = calculate_normal_probability(mean, std_dev, threshold_2, direction='below')
print("Probability of a value below", threshold_2, "in the normal distribution:", probability_below_2)
threshold_3 = 2
probability_above_3 = calculate_normal_probability(mean, std_dev, threshold_3, direction='above')
print("Probability of a value above", threshold_3, "in the normal distribution:", probability_above_3)
```

```
Probability of a value below 1 in the normal distribution: 0.29115968678834636
Probability of a value below -1 in the normal distribution: 0.06057075800205901
Probability of a value above 2 in the normal distribution: 0.5199388058383725
```

4.1.2 Simulating from discrete distribution:

Statistical analysis:

[24]:

```python
import numpy as np
from scipy.stats import binom, poisson, geom, hypergeom, randint, nbinom
def binomial_simulation(n, p, size=1000):
    return np.random.binomial(n, p, size)
def calculate_statistics(data):
    mean = np.mean(data)
    variance = np.var(data)
    std_dev = np.std(data)
    first_quantile = np.percentile(data, 25)
    third_quantile = np.percentile(data, 75)
    mode = np.argmax(np.bincount(data))
    skewness = data[data > mean].std() / data.std()
    kurtosis = ((data - mean) ** 4).mean() / std_dev ** 4
    return mean, variance, std_dev, first_quantile, third_quantile, mode, skewness, kurtosis
n_binom = 20
p_binom = 0.5
binom_data = binomial_simulation(n_binom, p_binom)
binom_stats = calculate_statistics(binom_data)
print("Binomial Distribution Statistics:")
print("Mean:", binom_stats[0])
print("Variance:", binom_stats[1])
print("Standard Deviation:", binom_stats[2])
print("First Quantile:", binom_stats[3])
print("Third Quantile:", binom_stats[4])
print("Mode:", binom_stats[5])
print("Skewness:", binom_stats[6])
print("Kurtosis:", binom_stats[7])
print()
```

```
Binomial Distribution Statistics:
Mean: 9.952
Variance: 5.215696
Standard Deviation: 2.2837898327122836
First Quantile: 8.0
Third Quantile: 11.0
Mode: 10
Skewness: 0.6466709259388216
Kurtosis: 3.0187228738492435
```

[25]:

```python
import numpy as np
from scipy.stats import binom, poisson, geom, hypergeom, randint, nbinom
def poisson_simulation(mu, size=1000):
    return np.random.poisson(mu, size)
def calculate_statistics(data):
    mean = np.mean(data)
    variance = np.var(data)
    std_dev = np.std(data)
    first_quantile = np.percentile(data, 25)
    third_quantile = np.percentile(data, 75)
    mode = np.argmax(np.bincount(data))
    skewness = data[data > mean].std() / data.std()
    kurtosis = ((data - mean) ** 4).mean() / std_dev ** 4
    return mean, variance, std_dev, first_quantile, third_quantile, mode, skewness, kurtosis
mu_poisson = 5
poisson_data = poisson_simulation(mu_poisson)
poisson_stats = calculate_statistics(poisson_data)
print("Poisson Distribution Statistics:")
print("Mean:", poisson_stats[0])
print("Variance:", poisson_stats[1])
print("Standard Deviation:", poisson_stats[2])
print("First Quantile:", poisson_stats[3])
print("Third Quantile:", poisson_stats[4])
print("Mode:", poisson_stats[5])
print("Skewness:", poisson_stats[6])
print("Kurtosis:", poisson_stats[7])
print()
```

```
Poisson Distribution Statistics:
Mean: 5.072
Variance: 5.078816000000001
Standard Deviation: 2.2536228610839038
First Quantile: 3.0
Third Quantile: 7.0
Mode: 5
Skewness: 0.6140971124505289
Kurtosis: 2.9911444336976385
```

•[28]:

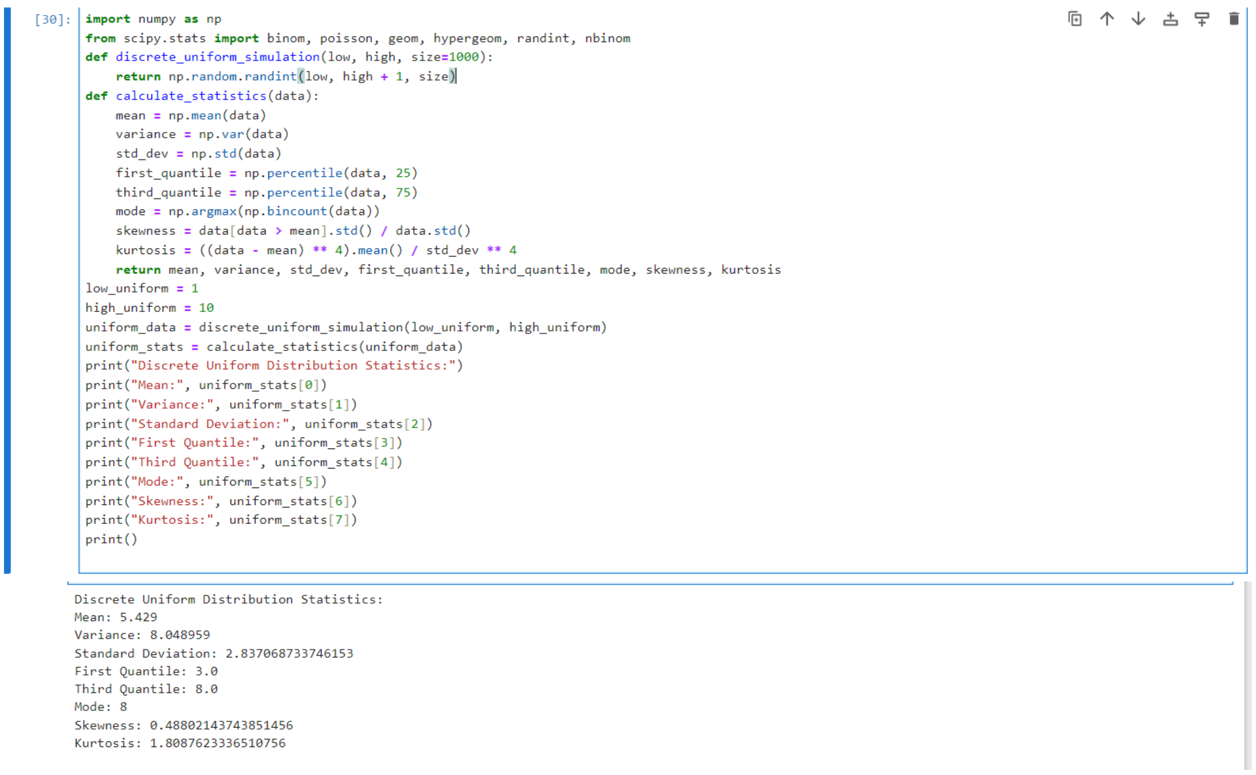
```python
import numpy as np
from scipy.stats import binom, poisson, geom, hypergeom, randint, nbinom
def geometric_simulation(p, size=1000):
    return np.random.geometric(p, size)
def calculate_statistics(data):
    mean = np.mean(data)
    variance = np.var(data)
    std_dev = np.std(data)
    first_quantile = np.percentile(data, 25)
    third_quantile = np.percentile(data, 75)
    mode = np.argmax(np.bincount(data))
    skewness = data[data > mean].std() / data.std()
    kurtosis = ((data - mean) ** 4).mean() / std_dev ** 4
    return mean, variance, std_dev, first_quantile, third_quantile, mode, skewness, kurtosis
p_geometric = 0.3
geometric_data = geometric_simulation(p_geometric)
geometric_stats = calculate_statistics(geometric_data)
print("Geometric Distribution Statistics:")
print("Mean:", geometric_stats[0])
print("Variance:", geometric_stats[1])
print("Standard Deviation:", geometric_stats[2])
print("First Quantile:", geometric_stats[3])
print("Third Quantile:", geometric_stats[4])
print("Mode:", geometric_stats[5])
print("Skewness:", geometric_stats[6])
print("Kurtosis:", geometric_stats[7])
print()
```

```
Geometric Distribution Statistics:
Mean: 3.316
Variance: 8.080143999999999
Standard Deviation: 2.842559410109136
First Quantile: 1.0
Third Quantile: 4.0
Mode: 1
Skewness: 1.0108269764928934
Kurtosis: 8.452013382102713
```

[29]:

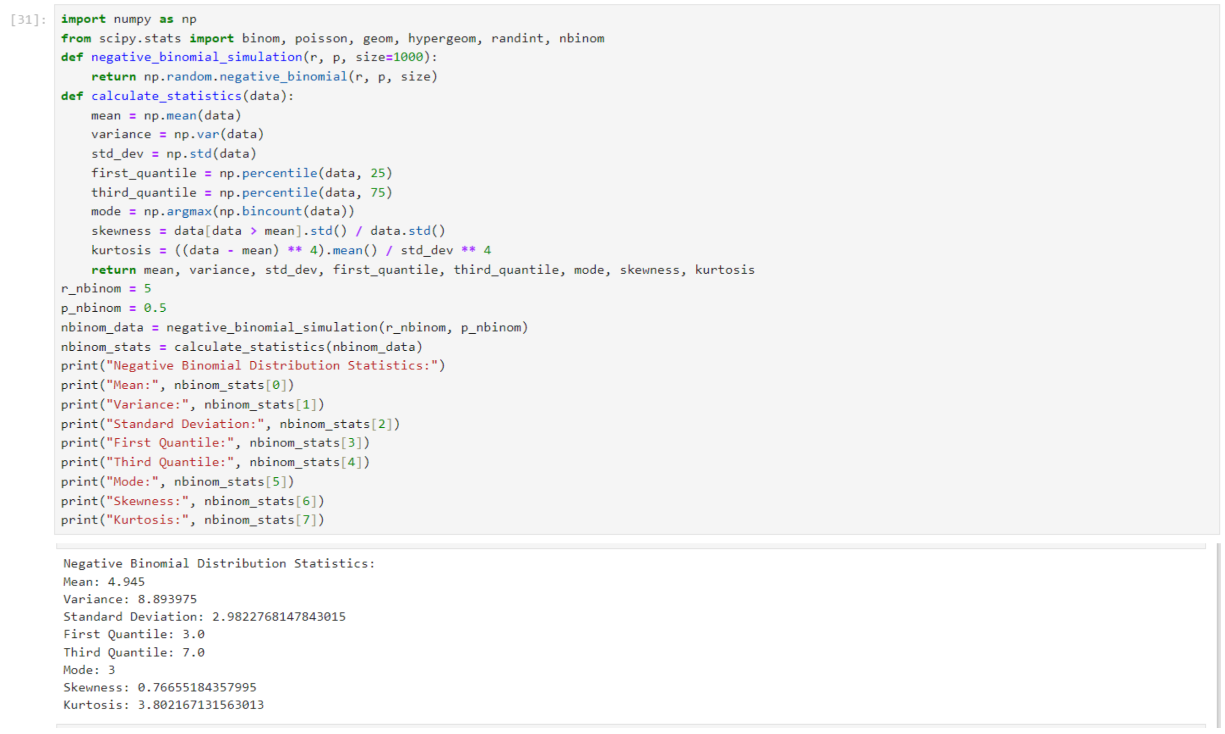
```python
import numpy as np
from scipy.stats import binom, poisson, geom, hypergeom, randint, nbinom
def hypergeometric_simulation(M, n, N, size=1000):
    return np.random.hypergeometric(M, n, N, size)
def calculate_statistics(data):
    mean = np.mean(data)
    variance = np.var(data)
    std_dev = np.std(data)
    first_quantile = np.percentile(data, 25)
    third_quantile = np.percentile(data, 75)
    mode = np.argmax(np.bincount(data))
    skewness = data[data > mean].std() / data.std()
    kurtosis = ((data - mean) ** 4).mean() / std_dev ** 4
    return mean, variance, std_dev, first_quantile, third_quantile, mode, skewness, kurtosis
M_hypergeo = 50
n_hypergeo = 10
N_hypergeo = 20
hypergeo_data = hypergeometric_simulation(M_hypergeo, n_hypergeo, N_hypergeo)
hypergeo_stats = calculate_statistics(hypergeo_data)
print("Hypergeometric Distribution Statistics:")
print("Mean:", hypergeo_stats[0])
print("Variance:", hypergeo_stats[1])
print("Standard Deviation:", hypergeo_stats[2])
print("First Quantile:", hypergeo_stats[3])
print("Third Quantile:", hypergeo_stats[4])
print("Mode:", hypergeo_stats[5])
print("Skewness:", hypergeo_stats[6])
print("Kurtosis:", hypergeo_stats[7])
print()
```

```
Hypergeometric Distribution Statistics:
Mean: 16.716
Variance: 1.8573439999999997
Standard Deviation: 1.3628440849928505
First Quantile: 16.0
Third Quantile: 18.0
Mode: 17
Skewness: 0.5814163592574452
Kurtosis: 2.7531362708994638
```



```
[30]: import numpy as np
      from scipy.stats import binom, poisson, geom, hypergeom, randint, nbinom
      def discrete_uniform_simulation(low, high, size=1000):
          return np.random.randint(low, high + 1, size)
      def calculate_statistics(data):
          mean = np.mean(data)
          variance = np.var(data)
          std_dev = np.std(data)
          first_quantile = np.percentile(data, 25)
          third_quantile = np.percentile(data, 75)
          mode = np.argmax(np.bincount(data))
          skewness = data[data > mean].std() / data.std()
          kurtosis = ((data - mean) ** 4).mean() / std_dev ** 4
          return mean, variance, std_dev, first_quantile, third_quantile, mode, skewness, kurtosis
      low_uniform = 1
      high_uniform = 10
      uniform_data = discrete_uniform_simulation(low_uniform, high_uniform)
      uniform_stats = calculate_statistics(uniform_data)
      print("Discrete Uniform Distribution Statistics:")
      print("Mean:", uniform_stats[0])
      print("Variance:", uniform_stats[1])
      print("Standard Deviation:", uniform_stats[2])
      print("First Quantile:", uniform_stats[3])
      print("Third Quantile:", uniform_stats[4])
      print("Mode:", uniform_stats[5])
      print("Skewness:", uniform_stats[6])
      print("Kurtosis:", uniform_stats[7])
      print()
```

```
Discrete Uniform Distribution Statistics:
Mean: 5.429
Variance: 8.048959
Standard Deviation: 2.837068733746153
First Quantile: 3.0
Third Quantile: 8.0
Mode: 8
Skewness: 0.48802143743851456
Kurtosis: 1.8087623336510756
```



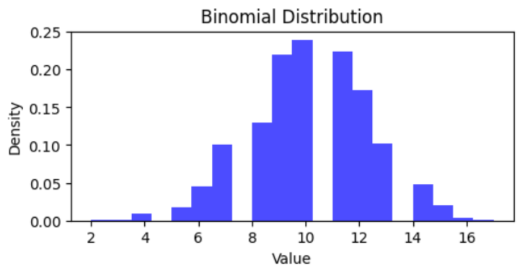
```
[31]: import numpy as np
      from scipy.stats import binom, poisson, geom, hypergeom, randint, nbinom
      def negative_binomial_simulation(r, p, size=1000):
          return np.random.negative_binomial(r, p, size)
      def calculate_statistics(data):
          mean = np.mean(data)
          variance = np.var(data)
          std_dev = np.std(data)
          first_quantile = np.percentile(data, 25)
          third_quantile = np.percentile(data, 75)
          mode = np.argmax(np.bincount(data))
          skewness = data[data > mean].std() / data.std()
          kurtosis = ((data - mean) ** 4).mean() / std_dev ** 4
          return mean, variance, std_dev, first_quantile, third_quantile, mode, skewness, kurtosis
      r_nbinom = 5
      p_nbinom = 0.5
      nbinom_data = negative_binomial_simulation(r_nbinom, p_nbinom)
      nbinom_stats = calculate_statistics(nbinom_data)
      print("Negative Binomial Distribution Statistics:")
      print("Mean:", nbinom_stats[0])
      print("Variance:", nbinom_stats[1])
      print("Standard Deviation:", nbinom_stats[2])
      print("First Quantile:", nbinom_stats[3])
      print("Third Quantile:", nbinom_stats[4])
      print("Mode:", nbinom_stats[5])
      print("Skewness:", nbinom_stats[6])
      print("Kurtosis:", nbinom_stats[7])
```

```
Negative Binomial Distribution Statistics:
Mean: 4.945
Variance: 8.893975
Standard Deviation: 2.9822768147843015
First Quantile: 3.0
Third Quantile: 7.0
Mode: 3
Skewness: 0.76655184357995
Kurtosis: 3.802167131563013
```

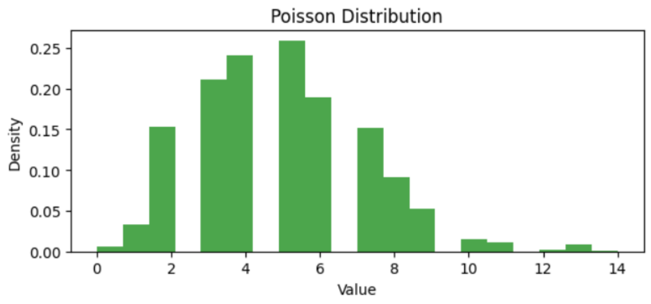
visualization:

```
[32]: import numpy as np
      import matplotlib.pyplot as plt
      from scipy.stats import binom, poisson, geom, hypergeom, randint, nbinom
      def binomial_simulation(n, p, size=1000):
          return np.random.binomial(n, p, size)
      n_binom = 20
      p_binom = 0.5
      binom_data = binomial_simulation(n_binom, p_binom)
      plt.figure(figsize=(12, 8))
      plt.subplot(3, 2, 1)
      plt.hist(binom_data, bins=20, density=True, color='blue', alpha=0.7)
      plt.title('Binomial Distribution')
      plt.xlabel('Value')
      plt.ylabel('Density')
```

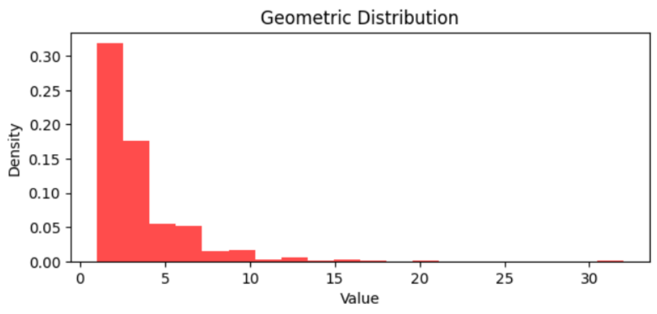
```
[32]: Text(0, 0.5, 'Density')
```



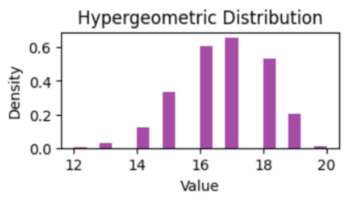
```
[33]: import numpy as np
      import matplotlib.pyplot as plt
      from scipy.stats import binom, poisson, geom, hypergeom, randint, nbinom
      def poisson_simulation(mu, size=1000):
          return np.random.poisson(mu, size)
      mu_poisson = 5
      poisson_data = poisson_simulation(mu_poisson)
      plt.figure(figsize=(12, 8))
      plt.subplot(3, 2, 2)
      plt.hist(poisson_data, bins=20, density=True, color='green', alpha=0.7)
      plt.title('Poisson Distribution')
      plt.xlabel('Value')
      plt.ylabel('Density')
      plt.tight_layout()
      plt.show()
```



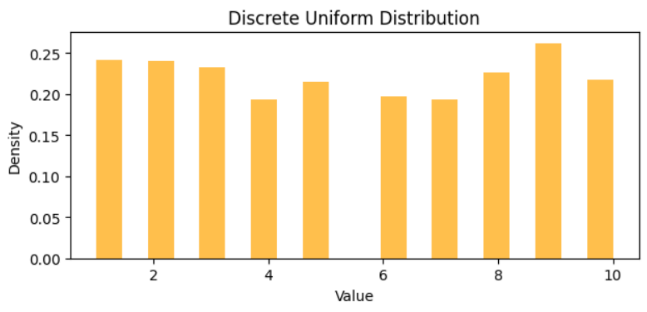
[34]:
```python
import numpy as np
import matplotlib.pyplot as plt
from scipy.stats import binom, poisson, geom, hypergeom, randint, nbinom
def geometric_simulation(p, size=1000):
    return np.random.geometric(p, size)
p_geometric = 0.3
geometric_data = geometric_simulation(p_geometric)
plt.figure(figsize=(12, 8))
plt.subplot(3, 2, 3)
plt.hist(geometric_data, bins=20, density=True, color='red', alpha=0.7)
plt.title('Geometric Distribution')
plt.xlabel('Value')
plt.ylabel('Density')
plt.tight_layout()
plt.show()
```



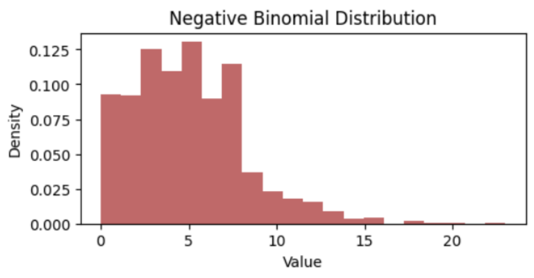
[35]:
```python
import numpy as np
import matplotlib.pyplot as plt
from scipy.stats import binom, poisson, geom, hypergeom, randint, nbinom
def hypergeometric_simulation(M, n, N, size=1000):
    return np.random.hypergeometric(M, n, N, size)
M_hypergeo = 50
n_hypergeo = 10
N_hypergeo = 20
hypergeo_data = hypergeometric_simulation(M_hypergeo, n_hypergeo, N_hypergeo)
plt.subplot(3, 2, 4)
plt.hist(hypergeo_data, bins=20, density=True, color='purple', alpha=0.7)
plt.title('Hypergeometric Distribution')
plt.xlabel('Value')
plt.ylabel('Density')
plt.tight_layout()
plt.show()
```



```
[36]: import numpy as np
      import matplotlib.pyplot as plt
      from scipy.stats import binom, poisson, geom, hypergeom, randint, nbinom
      def discrete_uniform_simulation(low, high, size=1000):
          return np.random.randint(low, high + 1, size)
      low_uniform = 1
      high_uniform = 10
      uniform_data = discrete_uniform_simulation(low_uniform, high_uniform)
      plt.figure(figsize=(12, 8))
      plt.subplot(3, 2, 5)
      plt.hist(uniform_data, bins=20, density=True, color='orange', alpha=0.7)
      plt.title('Discrete Uniform Distribution')
      plt.xlabel('Value')
      plt.ylabel('Density')
      plt.tight_layout()
      plt.show()
```



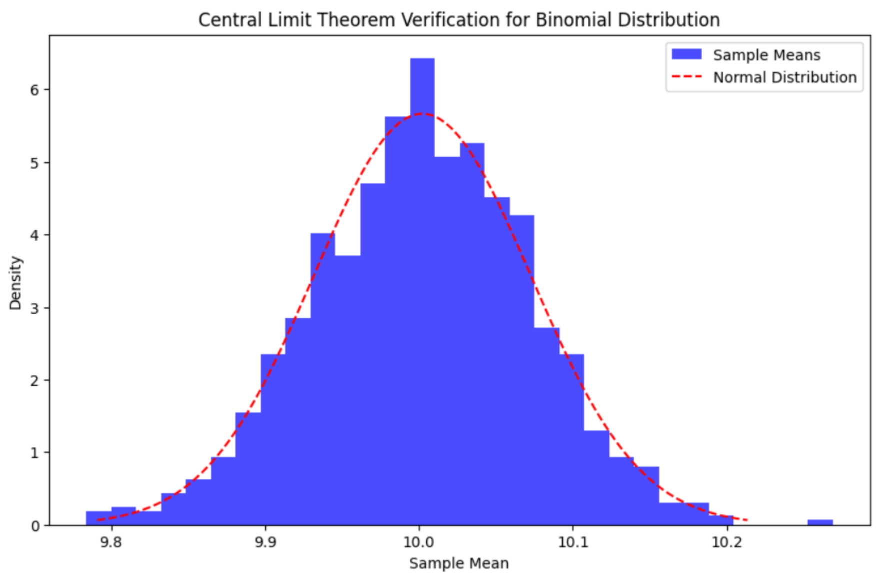
```
[37]: import numpy as np
      import matplotlib.pyplot as plt
      from scipy.stats import binom, poisson, geom, hypergeom, randint, nbinom
      def negative_binomial_simulation(r, p, size=1000):
          return np.random.negative_binomial(r, p, size)
      r_nbinom = 5
      p_nbinom = 0.5
      nbinom_data = negative_binomial_simulation(r_nbinom, p_nbinom)
      plt.figure(figsize=(12, 8))
      plt.subplot(3, 2, 6)
      plt.hist(nbinom_data, bins=20, density=True, color='brown', alpha=0.7)
      plt.title('Negative Binomial Distribution')
      plt.xlabel('Value')
      plt.ylabel('Density')

[37]: Text(0, 0.5, 'Density')
```



central limit theorem verificaation:

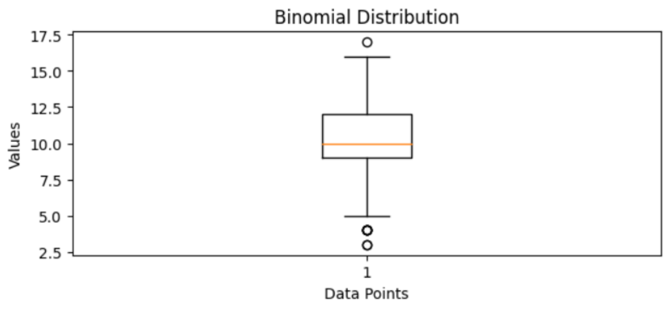
```
[38]: import numpy as np
import matplotlib.pyplot as plt
def binomial_simulation(n, p, size=1000):
    return np.random.binomial(n, p, size)
def poisson_simulation(mu, size=1000):
    return np.random.poisson(mu, size)
def geometric_simulation(p, size=1000):
    return np.random.geometric(p, size)
def hypergeometric_simulation(M, n, N, size=1000):
    return np.random.hypergeometric(M, n, N, size)
def discrete_uniform_simulation(low, high, size=1000):
    return np.random.randint(low, high + 1, size)
def negative_binomial_simulation(r, p, size=1000):
    return np.random.negative_binomial(r, p, size)
def clt_verification(data_generator, *args, sample_size=1000, num_samples=1000):
    sample_means = []
    for _ in range(num_samples):
        sample = data_generator(*args, size=sample_size)
        sample_means.append(np.mean(sample))
    return sample_means
```

```
n_binom = 20
p_binom = 0.5
mu_poisson = 5
p_geometric = 0.3
M_hypergeo = 50
n_hypergeo = 10
N_hypergeo = 20
low_uniform = 1
high_uniform = 10
r_nbinom = 5
p_nbinom = 0.5
binom_sample_means = clt_verification(binomial_simulation, n_binom, p_binom)
plt.figure(figsize=(10, 6))
plt.hist(binom_sample_means, bins=30, density=True, color='blue', alpha=0.7, label='Sample Means')
mu_clt = np.mean(binom_sample_means)
sigma_clt = np.std(binom_sample_means)
x = np.linspace(mu_clt - 3 * sigma_clt, mu_clt + 3 * sigma_clt, 100)
plt.plot(x, norm.pdf(x, mu_clt, sigma_clt), color='red', linestyle='--', label='Normal Distribution')
plt.title('Central Limit Theorem Verification for Binomial Distribution')
plt.xlabel('Sample Mean')
plt.ylabel('Density')
plt.legend()
plt.show()
```



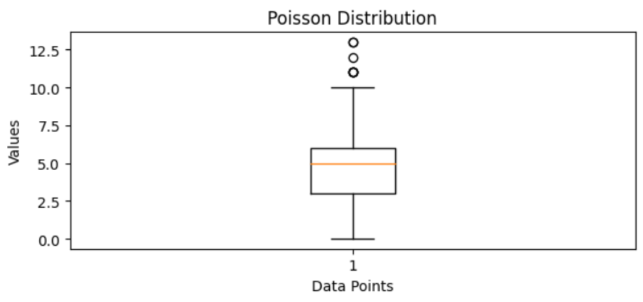
outlier detection:

[39]:
```python
import numpy as np
import matplotlib.pyplot as plt
from scipy.stats import iqr
def binomial_simulation(n, p, size=1000):
    return np.random.binomial(n, p, size)
def detect_outliers(data):
    q1 = np.percentile(data, 25)
    q3 = np.percentile(data, 75)
    iqr_value = q3 - q1
    lower_bound = q1 - 1.5 * iqr_value
    upper_bound = q3 + 1.5 * iqr_value
    outliers = (data < lower_bound) | (data > upper_bound)
    return outliers
n_binom = 20
p_binom = 0.5
binom_data = binomial_simulation(n_binom, p_binom)
outliers_binom = detect_outliers(binom_data)
plt.figure(figsize=(12, 8))
plt.subplot(3, 2, 1)
plt.boxplot(binom_data)
plt.title('Binomial Distribution')
plt.xlabel('Data Points')
plt.ylabel('Values')
plt.tight_layout()
plt.show()
print("Outliers in Binomial Distribution:", np.sum(outliers_binom))
```



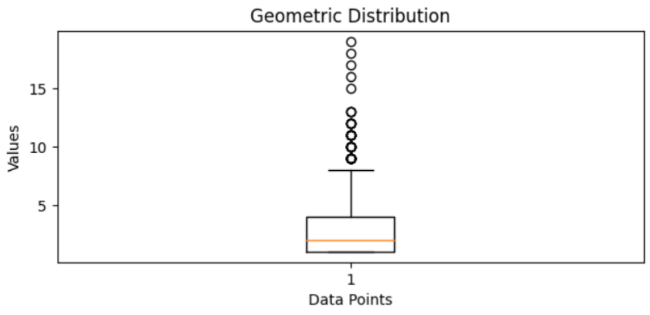
```
Outliers in Binomial Distribution: 8
```

[40]:
```python
import numpy as np
import matplotlib.pyplot as plt
from scipy.stats import iqr
def poisson_simulation(mu, size=1000):
    return np.random.poisson(mu, size)
def detect_outliers(data):
    q1 = np.percentile(data, 25)
    q3 = np.percentile(data, 75)
    iqr_value = q3 - q1
    lower_bound = q1 - 1.5 * iqr_value
    upper_bound = q3 + 1.5 * iqr_value
    outliers = (data < lower_bound) | (data > upper_bound)
    return outliers
mu_poisson = 5
poisson_data = poisson_simulation(mu_poisson)
outliers_poisson = detect_outliers(poisson_data)
plt.figure(figsize=(12, 8))
plt.subplot(3, 2, 2)
plt.boxplot(poisson_data)
plt.title('Poisson Distribution')
plt.xlabel('Data Points')
plt.ylabel('Values')
plt.tight_layout()
plt.show()
print("Outliers in Poisson Distribution:", np.sum(outliers_poisson))
```



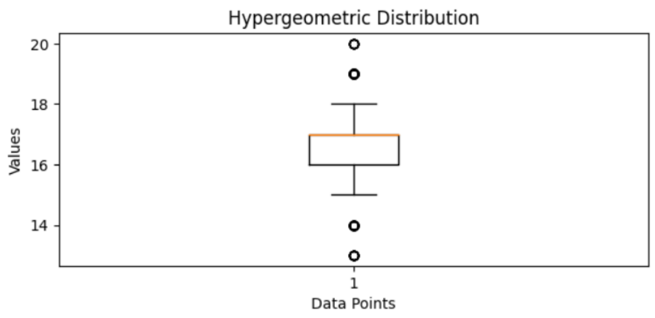
```
Outliers in Poisson Distribution: 7
```

[41]:
```python
import numpy as np
import matplotlib.pyplot as plt
from scipy.stats import iqr
def geometric_simulation(p, size=1000):
    return np.random.geometric(p, size)
def detect_outliers(data):
    q1 = np.percentile(data, 25)
    q3 = np.percentile(data, 75)
    iqr_value = q3 - q1
    lower_bound = q1 - 1.5 * iqr_value
    upper_bound = q3 + 1.5 * iqr_value
    outliers = (data < lower_bound) | (data > upper_bound)
    return outliers
p_geometric = 0.3
geometric_data = geometric_simulation(p_geometric)
outliers_geometric = detect_outliers(geometric_data)
plt.figure(figsize=(12, 8))
plt.subplot(3, 2, 3)
plt.boxplot(geometric_data)
plt.title('Geometric Distribution')
plt.xlabel('Data Points')
plt.ylabel('Values')
plt.tight_layout()
plt.show()
print("Outliers in Geometric Distribution:", np.sum(outliers_geometric))
```



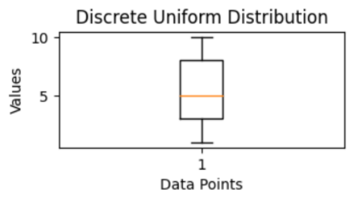
```
Outliers in Geometric Distribution: 45
```

[43]:
```python
import numpy as np
import matplotlib.pyplot as plt
from scipy.stats import iqr
def hypergeometric_simulation(M, n, N, size=1000):
    return np.random.hypergeometric(M, n, N, size)
def detect_outliers(data):
    q1 = np.percentile(data, 25)
    q3 = np.percentile(data, 75)
    iqr_value = q3 - q1
    lower_bound = q1 - 1.5 * iqr_value
    upper_bound = q3 + 1.5 * iqr_value
    outliers = (data < lower_bound) | (data > upper_bound)
    return outliers
M_hypergeo = 50
n_hypergeo = 10
N_hypergeo = 20
hypergeo_data = hypergeometric_simulation(M_hypergeo, n_hypergeo, N_hypergeo)
outliers_hypergeo = detect_outliers(hypergeo_data)
plt.figure(figsize=(12, 8))
plt.subplot(3, 2, 4)
plt.boxplot(hypergeo_data)
plt.title('Hypergeometric Distribution')
plt.xlabel('Data Points')
plt.ylabel('Values')
plt.tight_layout()
plt.show()
print("Outliers in Hypergeometric Distribution:", np.sum(outliers_hypergeo))
```



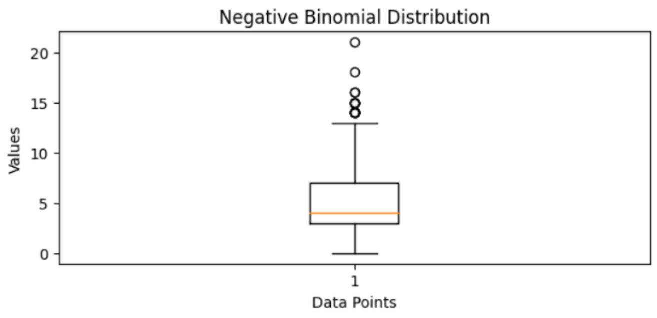
```
Outliers in Hypergeometric Distribution: 135
```

[44]:
```python
import numpy as np
import matplotlib.pyplot as plt
from scipy.stats import iqr
def discrete_uniform_simulation(low, high, size=1000):
    return np.random.randint(low, high + 1, size)
def detect_outliers(data):
    q1 = np.percentile(data, 25)
    q3 = np.percentile(data, 75)
    iqr_value = q3 - q1
    lower_bound = q1 - 1.5 * iqr_value
    upper_bound = q3 + 1.5 * iqr_value
    outliers = (data < lower_bound) | (data > upper_bound)
    return outliers
low_uniform = 1
high_uniform = 10
uniform_data = discrete_uniform_simulation(low_uniform, high_uniform)
outliers_uniform = detect_outliers(uniform_data)
plt.subplot(3, 2, 5)
plt.boxplot(uniform_data)
plt.title('Discrete Uniform Distribution')
plt.xlabel('Data Points')
plt.ylabel('Values')
plt.tight_layout()
plt.show()
print("Outliers in Discrete Uniform Distribution:", np.sum(outliers_uniform))
```



```
Outliers in Discrete Uniform Distribution: 0
```

[45]:
```python
import numpy as np
import matplotlib.pyplot as plt
from scipy.stats import iqr
def negative_binomial_simulation(r, p, size=1000):
    return np.random.negative_binomial(r, p, size)
def detect_outliers(data):
    q1 = np.percentile(data, 25)
    q3 = np.percentile(data, 75)
    iqr_value = q3 - q1
    lower_bound = q1 - 1.5 * iqr_value
    upper_bound = q3 + 1.5 * iqr_value
    outliers = (data < lower_bound) | (data > upper_bound)
    return outliers
r_nbinom = 5
p_nbinom = 0.5
nbinom_data = negative_binomial_simulation(r_nbinom, p_nbinom)
outliers_nbinom = detect_outliers(nbinom_data)
plt.figure(figsize=(12, 8))
plt.subplot(3, 2, 6)
plt.boxplot(nbinom_data)
plt.title('Negative Binomial Distribution')
plt.xlabel('Data Points')
plt.ylabel('Values')
plt.tight_layout()
plt.show()
print("Outliers in Negative Binomial Distribution:", np.sum(outliers_nbinom))
```



```
Outliers in Negative Binomial Distribution: 17
```

```
[ ]:
```

probability calculation:

```
[46]: import numpy as np
from scipy.stats import binom
def calculate_binomial_probability(n, p, threshold, direction='below'):
    if direction == 'below':
        probability = binom.cdf(threshold, n, p)
    elif direction == 'above':
        probability = 1 - binom.cdf(threshold - 1, n, p)
    else:
        raise ValueError("Invalid direction. Choose 'below' or 'above'.")
    return probability
n_binom = 10
p_binom = 0.5
threshold_1 = 5
probability_below_1 = calculate_binomial_probability(n_binom, p_binom, threshold_1, direction='below')
print("Probability of a value below", threshold_1, "in the binomial distribution:", probability_below_1)
threshold_2 = 8
probability_above_2 = calculate_binomial_probability(n_binom, p_binom, threshold_2, direction='above')
print("Probability of a value above", threshold_2, "in the binomial distribution:", probability_above_2)
```

```
Probability of a value below 5 in the binomial distribution: 0.623046875
Probability of a value above 8 in the binomial distribution: 0.0546875
```

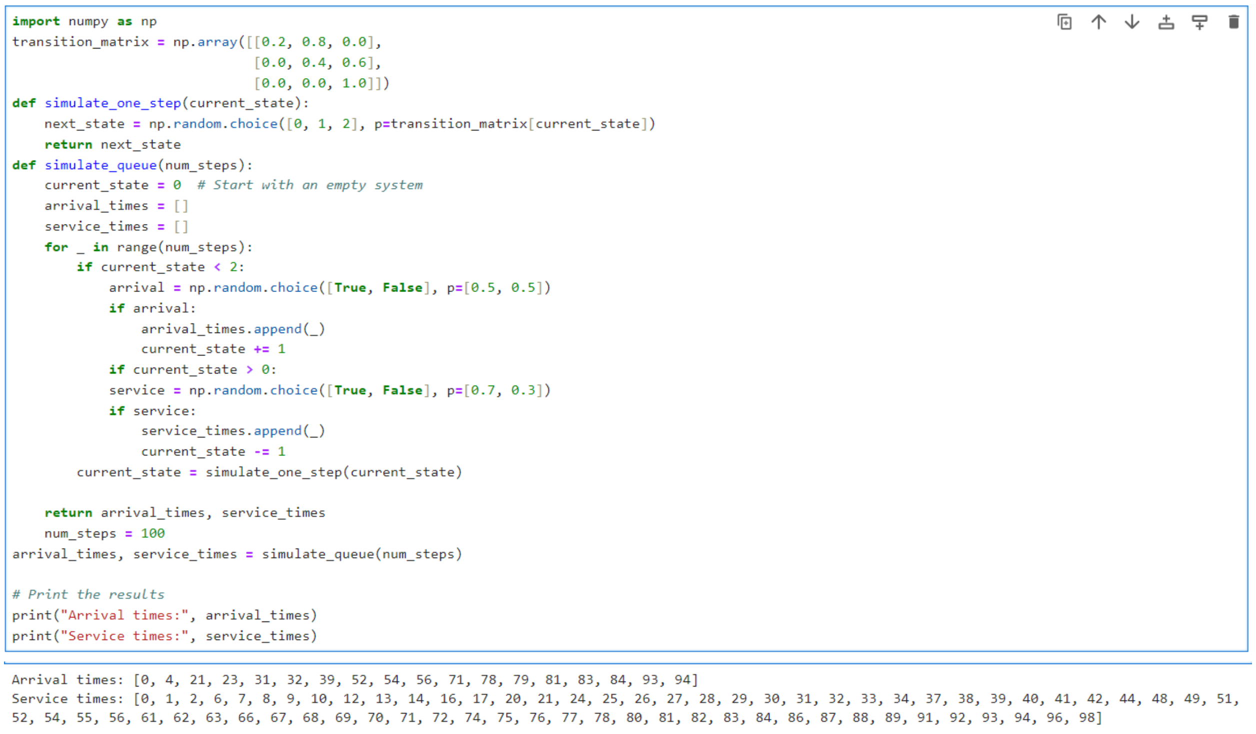
4.1.3 Markov chains

Transition matricx simulation

```
import numpy as np
transition_matrix = np.array([[0.7, 0.3],
                              [0.4, 0.6]])
initial_state_probabilities = np.array([0.5, 0.5])
def simulate_markov_chain(transition_matrix, initial_state_probabilities, num_steps):
    current_state = np.random.choice(len(initial_state_probabilities), p=initial_state_probabilities)
    state_counts = [0] * len(initial_state_probabilities)
    state_counts[current_state] += 1
    for _ in range(num_steps):
        next_state = np.random.choice(len(transition_matrix[current_state]), p=transition_matrix[current_state])
        state_counts[next_state] += 1
        current_state = next_state
    state_probabilities = [count / num_steps for count in state_counts]
    return state_probabilities
num_steps = 1000
state_probabilities = simulate_markov_chain(transition_matrix, initial_state_probabilities, num_steps)
print("State probabilities after", num_steps, "steps:", state_probabilities)SSS
```

```
State probabilities after 1000 steps: [0.6, 0.401]
```

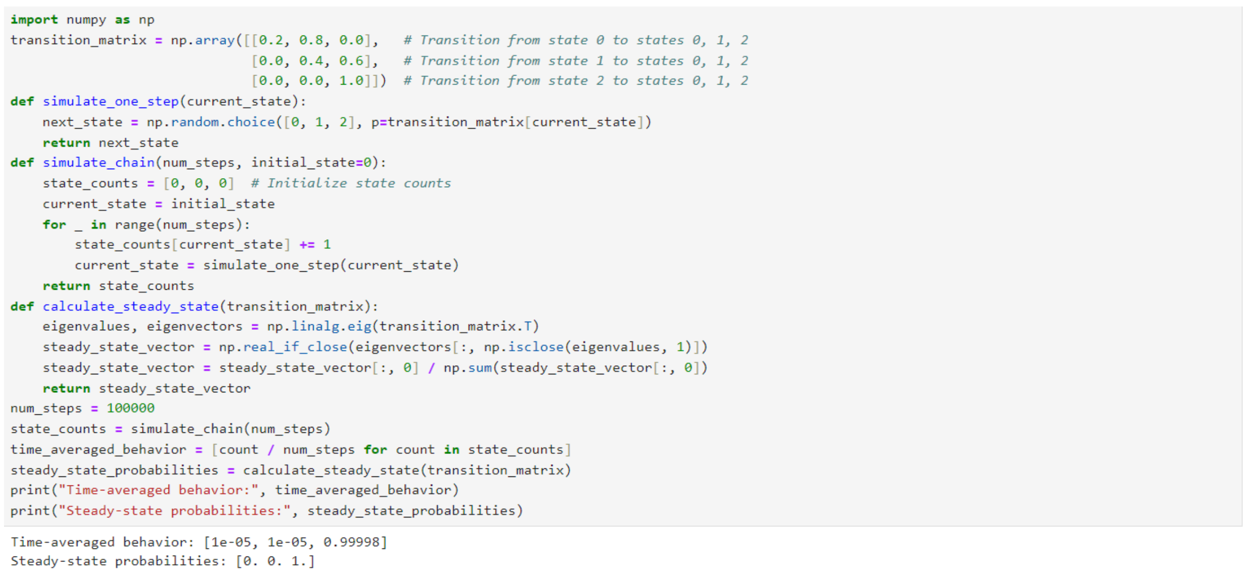
Rcurrent events :



```python
import numpy as np
transition_matrix = np.array([[0.2, 0.8, 0.0],
                              [0.0, 0.4, 0.6],
                              [0.0, 0.0, 1.0]])
def simulate_one_step(current_state):
    next_state = np.random.choice([0, 1, 2], p=transition_matrix[current_state])
    return next_state
def simulate_queue(num_steps):
    current_state = 0  # Start with an empty system
    arrival_times = []
    service_times = []
    for _ in range(num_steps):
        if current_state < 2:
            arrival = np.random.choice([True, False], p=[0.5, 0.5])
            if arrival:
                arrival_times.append(_)
                current_state += 1
            if current_state > 0:
            service = np.random.choice([True, False], p=[0.7, 0.3])
            if service:
                service_times.append(_)
                current_state -= 1
        current_state = simulate_one_step(current_state)

    return arrival_times, service_times
    num_steps = 100
arrival_times, service_times = simulate_queue(num_steps)

# Print the results
print("Arrival times:", arrival_times)
print("Service times:", service_times)
```

```
Arrival times: [0, 4, 21, 23, 31, 32, 39, 52, 54, 56, 71, 78, 79, 81, 83, 84, 93, 94]
Service times: [0, 1, 2, 6, 7, 8, 9, 10, 12, 13, 14, 16, 17, 20, 21, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 37, 38, 39, 40, 41, 42, 44, 48, 49, 51,
52, 54, 55, 56, 61, 62, 63, 66, 67, 68, 69, 70, 71, 72, 74, 75, 76, 77, 78, 80, 81, 82, 83, 84, 86, 87, 88, 89, 91, 92, 93, 94, 96, 98]
```

Ergodicity:



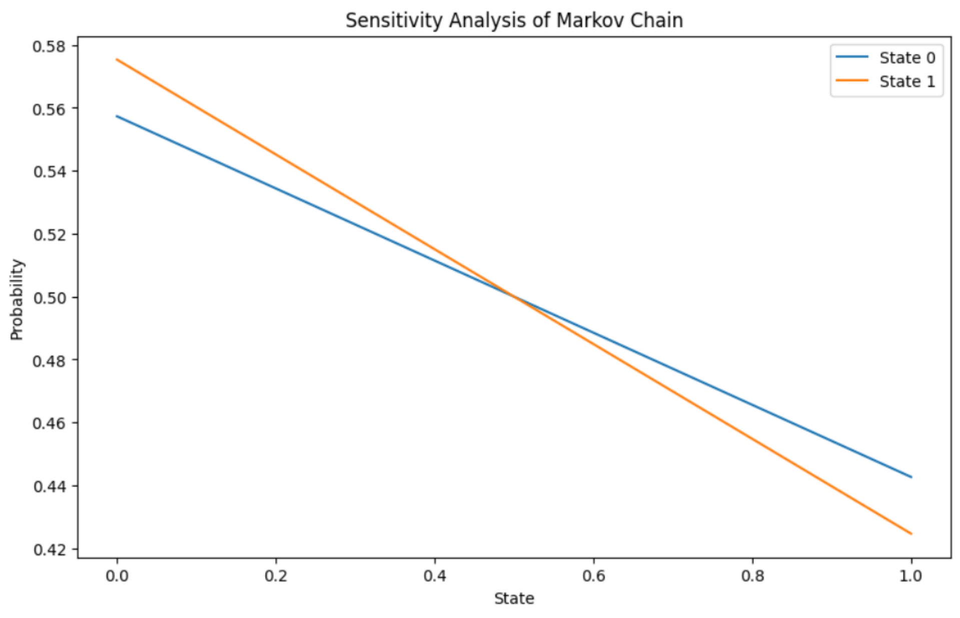
```python
import numpy as np
transition_matrix = np.array([[0.2, 0.8, 0.0],   # Transition from state 0 to states 0, 1, 2
                              [0.0, 0.4, 0.6],   # Transition from state 1 to states 0, 1, 2
                              [0.0, 0.0, 1.0]])  # Transition from state 2 to states 0, 1, 2
def simulate_one_step(current_state):
    next_state = np.random.choice([0, 1, 2], p=transition_matrix[current_state])
    return next_state
def simulate_chain(num_steps, initial_state=0):
    state_counts = [0, 0, 0]  # Initialize state counts
    current_state = initial_state
    for _ in range(num_steps):
        state_counts[current_state] += 1
        current_state = simulate_one_step(current_state)
    return state_counts
def calculate_steady_state(transition_matrix):
    eigenvalues, eigenvectors = np.linalg.eig(transition_matrix.T)
    steady_state_vector = np.real_if_close(eigenvectors[:, np.isclose(eigenvalues, 1)])
    steady_state_vector = steady_state_vector[:, 0] / np.sum(steady_state_vector[:, 0])
    return steady_state_vector
num_steps = 100000
state_counts = simulate_chain(num_steps)
time_averaged_behavior = [count / num_steps for count in state_counts]
steady_state_probabilities = calculate_steady_state(transition_matrix)
print("Time-averaged behavior:", time_averaged_behavior)
print("Steady-state probabilities:", steady_state_probabilities)
```

```
Time-averaged behavior: [1e-05, 1e-05, 0.99998]
Steady-state probabilities: [0. 0. 1.]
```

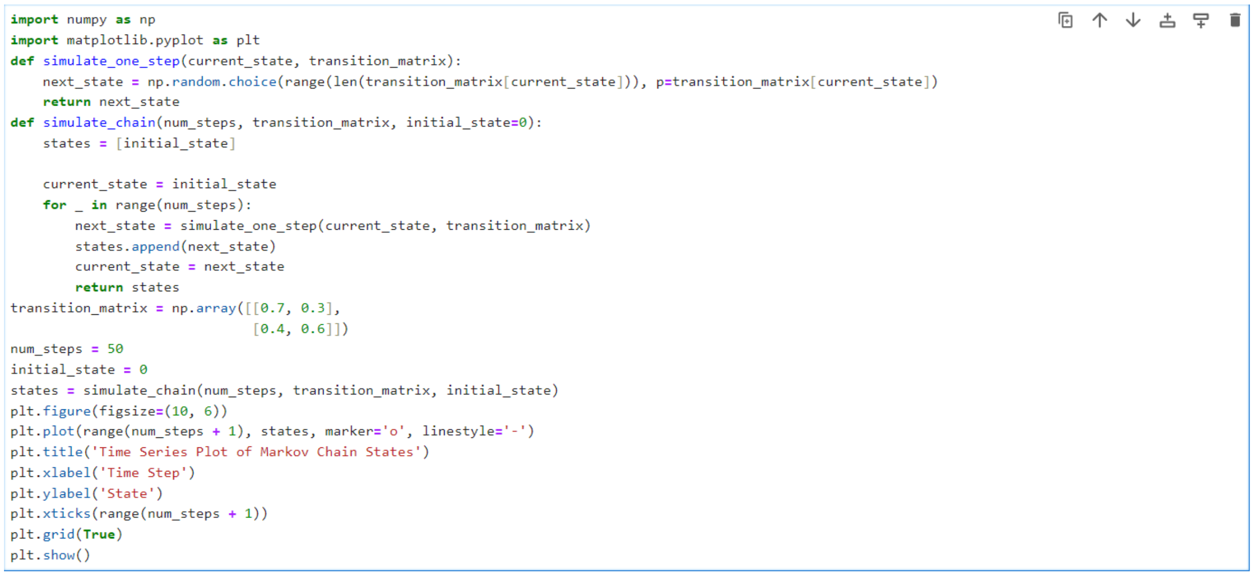
Sensitivity analysis:

```python
import numpy as np
import matplotlib.pyplot as plt
def simulate_one_step(current_state, transition_matrix):
    next_state = np.random.choice(range(len(transition_matrix[current_state])), p=transition_matrix[current_state])
    return next_state
def simulate_chain(num_steps, transition_matrix, initial_state=0):
    state_counts = [0] * len(transition_matrix)
    current_state = initial_state
    for _ in range(num_steps):
        state_counts[current_state] += 1
        current_state = simulate_one_step(current_state, transition_matrix)
    return state_counts
def sensitivity_analysis(transition_matrix, num_steps, num_samples, epsilon):
    results = []
    for i in range(len(transition_matrix)):
        modified_matrix = np.copy(transition_matrix)
        for j in range(len(transition_matrix[i])):
            modified_matrix[i][j] += epsilon
        modified_matrix[i] /= np.sum(modified_matrix[i])
        average_behavior = np.zeros(len(transition_matrix))
        for _ in range(num_samples):
            state_counts = simulate_chain(num_steps, modified_matrix)
            state_probabilities = np.array(state_counts) / num_steps
            average_behavior += state_probabilities
        average_behavior /= num_samples
        results.append(average_behavior)
    return results
transition_matrix = np.array([[0.7, 0.3],   # Transition from state 0 to states 0 and 1
                              [0.4, 0.6]])  # Transition from state 1 to states 0 and 1
```

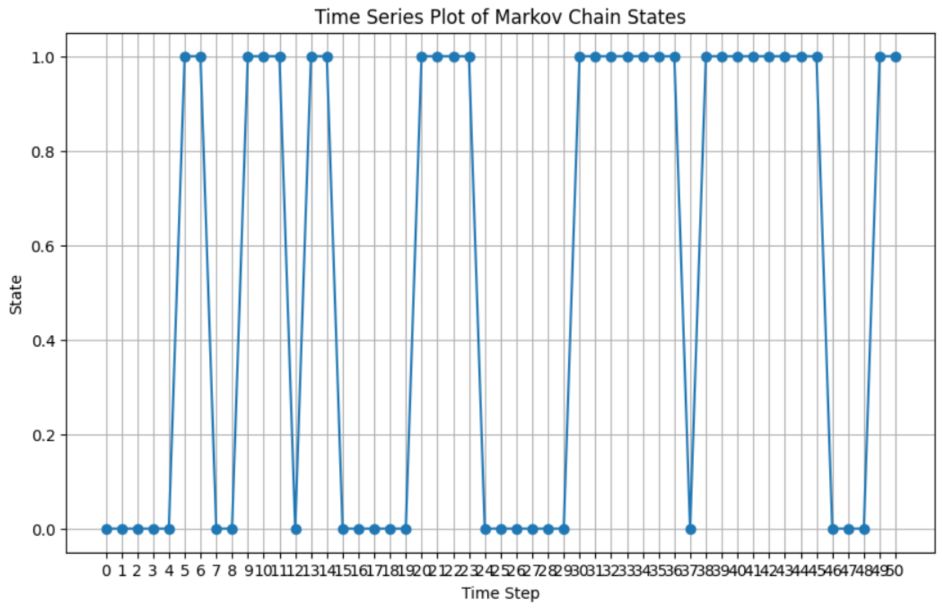
```python
num_steps = 1000
num_samples = 100
epsilon = 0.05
sensitivity_results = sensitivity_analysis(transition_matrix, num_steps, num_samples, epsilon)
plt.figure(figsize=(10, 6))
for i, result in enumerate(sensitivity_results):
    plt.plot(result, label='State {}'.format(i))
plt.title('Sensitivity Analysis of Markov Chain')
plt.xlabel('State')
plt.ylabel('Probability')
plt.legend()
plt.show()
```



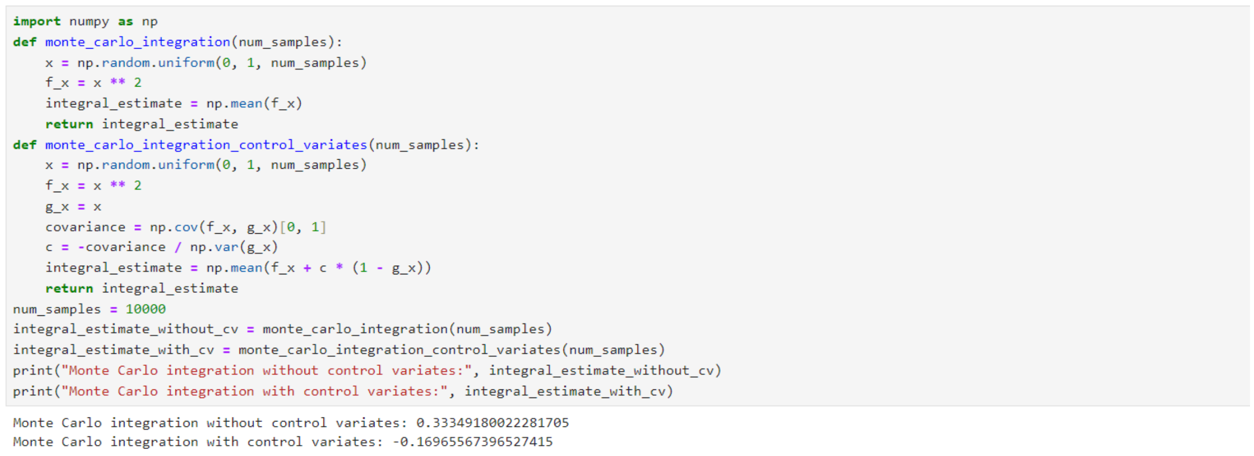
visualization:



```python
import numpy as np
import matplotlib.pyplot as plt
def simulate_one_step(current_state, transition_matrix):
    next_state = np.random.choice(range(len(transition_matrix[current_state])), p=transition_matrix[current_state])
    return next_state
def simulate_chain(num_steps, transition_matrix, initial_state=0):
    states = [initial_state]

    current_state = initial_state
    for _ in range(num_steps):
        next_state = simulate_one_step(current_state, transition_matrix)
        states.append(next_state)
        current_state = next_state
        return states
transition_matrix = np.array([[0.7, 0.3],
                              [0.4, 0.6]])
num_steps = 50
initial_state = 0
states = simulate_chain(num_steps, transition_matrix, initial_state)
plt.figure(figsize=(10, 6))
plt.plot(range(num_steps + 1), states, marker='o', linestyle='-')
plt.title('Time Series Plot of Markov Chain States')
plt.xlabel('Time Step')
plt.ylabel('State')
plt.xticks(range(num_steps + 1))
plt.grid(True)
plt.show()
```



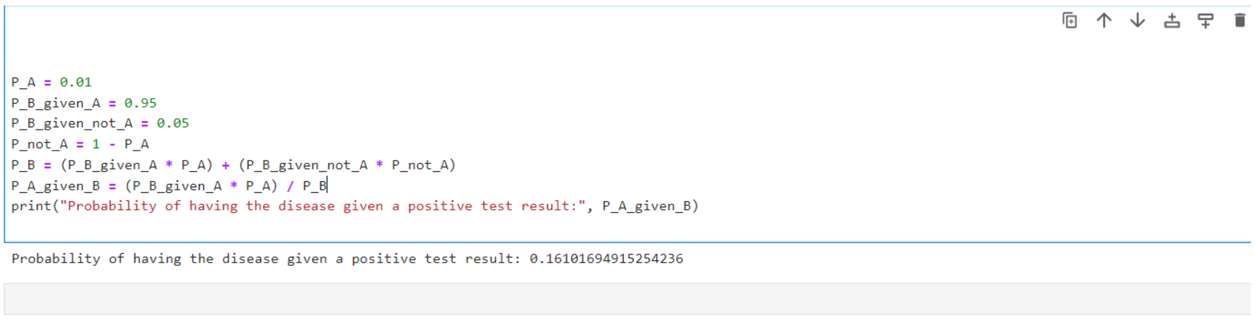
4.1.4 variance reduction techniques:



```
import numpy as np
def monte_carlo_integration(num_samples):
    x = np.random.uniform(0, 1, num_samples)
    f_x = x ** 2
    integral_estimate = np.mean(f_x)
    return integral_estimate
def monte_carlo_integration_control_variates(num_samples):
    x = np.random.uniform(0, 1, num_samples)
    f_x = x ** 2
    g_x = x
    covariance = np.cov(f_x, g_x)[0, 1]
    c = -covariance / np.var(g_x)
    integral_estimate = np.mean(f_x + c * (1 - g_x))
    return integral_estimate
num_samples = 10000
integral_estimate_without_cv = monte_carlo_integration(num_samples)
integral_estimate_with_cv = monte_carlo_integration_control_variates(num_samples)
print("Monte Carlo integration without control variates:", integral_estimate_without_cv)
print("Monte Carlo integration with control variates:", integral_estimate_with_cv)
```

```
Monte Carlo integration without control variates: 0.33349180022281705
Monte Carlo integration with control variates: -0.16965567396527415
```

Baye's theroem

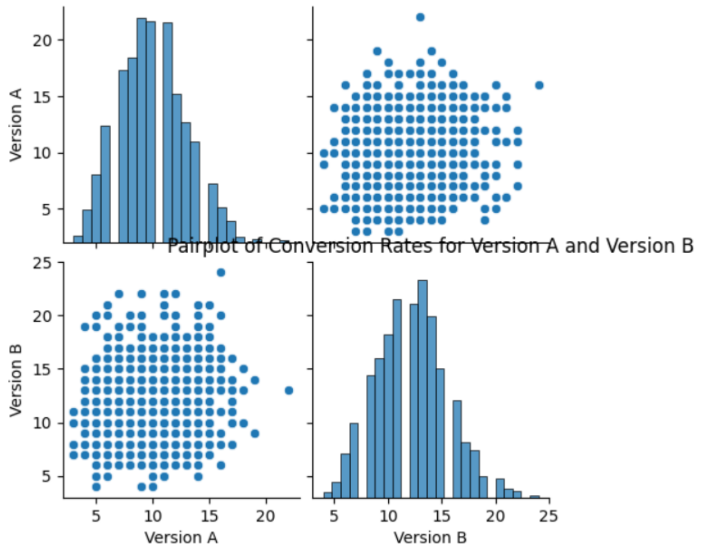


```
P_A = 0.01
P_B_given_A = 0.95
P_B_given_not_A = 0.05
P_not_A = 1 - P_A
P_B = (P_B_given_A * P_A) + (P_B_given_not_A * P_not_A)
P_A_given_B = (P_B_given_A * P_A) / P_B
print("Probability of having the disease given a positive test result:", P_A_given_B)
```

```
Probability of having the disease given a positive test result: 0.16101694915254236
```

Joint distribution analysis:

```python
import numpy as np
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
from scipy import stats
np.random.seed(42)
data_A = np.random.binomial(n=100, p=0.1, size=1000)  # Conversion rate: 10%
data_B = np.random.binomial(n=100, p=0.12, size=1000)  # Conversion rate: 12%
df = pd.DataFrame({'Version A': data_A, 'Version B': data_B})
sns.pairplot(df)
plt.title('Pairplot of Conversion Rates for Version A and Version B')
plt.show()
correlation_coefficient = df['Version A'].corr(df['Version B'])
print("Correlation Coefficient between Version A and Version B:", correlation_coefficient)
stat_A, p_A = stats.shapiro(data_A)
stat_B, p_B = stats.shapiro(data_B)
print("Shapiro-Wilk test for Version A - Statistic:", stat_A, "p-value:", p_A)
print("Shapiro-Wilk test for Version B - Statistic:", stat_B, "p-value:", p_B)
alpha = 0.05
if p_A > alpha:
    print("Version A data appears to be normally distributed (fail to reject H0)")
else:
    print("Version A data does not appear to be normally distributed (reject H0)")

if p_B > alpha:
    print("Version B data appears to be normally distributed (fail to reject H0)")
else:
    print("Version B data does not appear to be normally distributed (reject H0)")
```



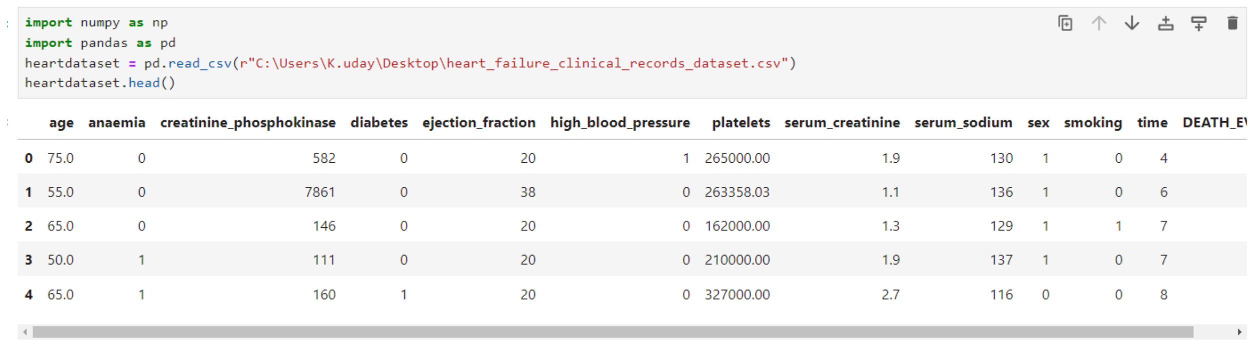
```
Correlation Coefficient between Version A and Version B: 0.04080959591118666
Shapiro-Wilk test for Version A - Statistic: 0.9835802316665649 p-value: 3.545915783220721e-09
Shapiro-Wilk test for Version B - Statistic: 0.986870288848877 p-value: 8.249918437286397e-08
Version A data does not appear to be normally distributed (reject H0)
Version B data does not appear to be normally distributed (reject H0)
```

4.2 Real data analysis

Baye’s theorem

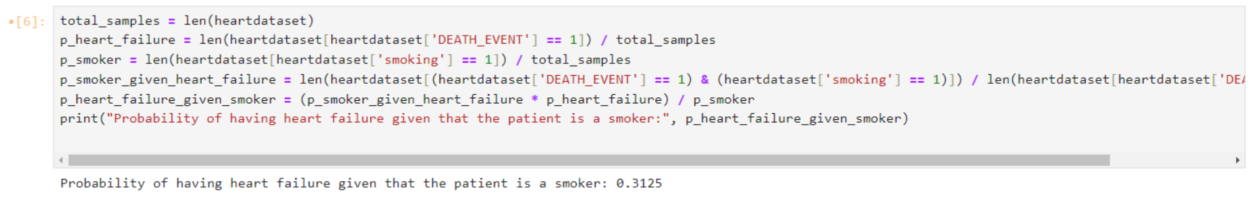
Bayes’ Theorem

$$P(A|B) = \frac{P(B|A)\,P(A)}{P(B)}$$



```
import numpy as np
import pandas as pd
heartdataset = pd.read_csv(r"C:\Users\K.uday\Desktop\heart_failure_clinical_records_dataset.csv")
heartdataset.head()
```

	age	anaemia	creatinine_phosphokinase	diabetes	ejection_fraction	high_blood_pressure	platelets	serum_creatinine	serum_sodium	sex	smoking	time	DEATH_E
0	75.0	0	582	0	20	1	265000.00	1.9	130	1	0	4	
1	55.0	0	7861	0	38	0	263358.03	1.1	136	1	0	6	
2	65.0	0	146	0	20	0	162000.00	1.3	129	1	1	7	
3	50.0	1	111	0	20	0	210000.00	1.9	137	1	0	7	
4	65.0	1	160	1	20	0	327000.00	2.7	116	0	0	8	



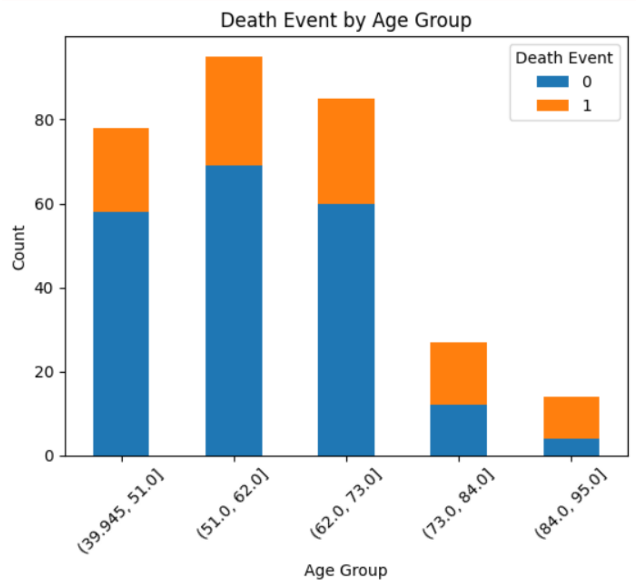
```
•[6]: total_samples = len(heartdataset)
      p_heart_failure = len(heartdataset[heartdataset['DEATH_EVENT'] == 1]) / total_samples
      p_smoker = len(heartdataset[heartdataset['smoking'] == 1]) / total_samples
      p_smoker_given_heart_failure = len(heartdataset[(heartdataset['DEATH_EVENT'] == 1) & (heartdataset['smoking'] == 1)]) / len(heartdataset[heartdataset['DE
      p_heart_failure_given_smoker = (p_smoker_given_heart_failure * p_heart_failure) / p_smoker
      print("Probability of having heart failure given that the patient is a smoker:", p_heart_failure_given_smoker)

      Probability of having heart failure given that the patient is a smoker: 0.3125
```

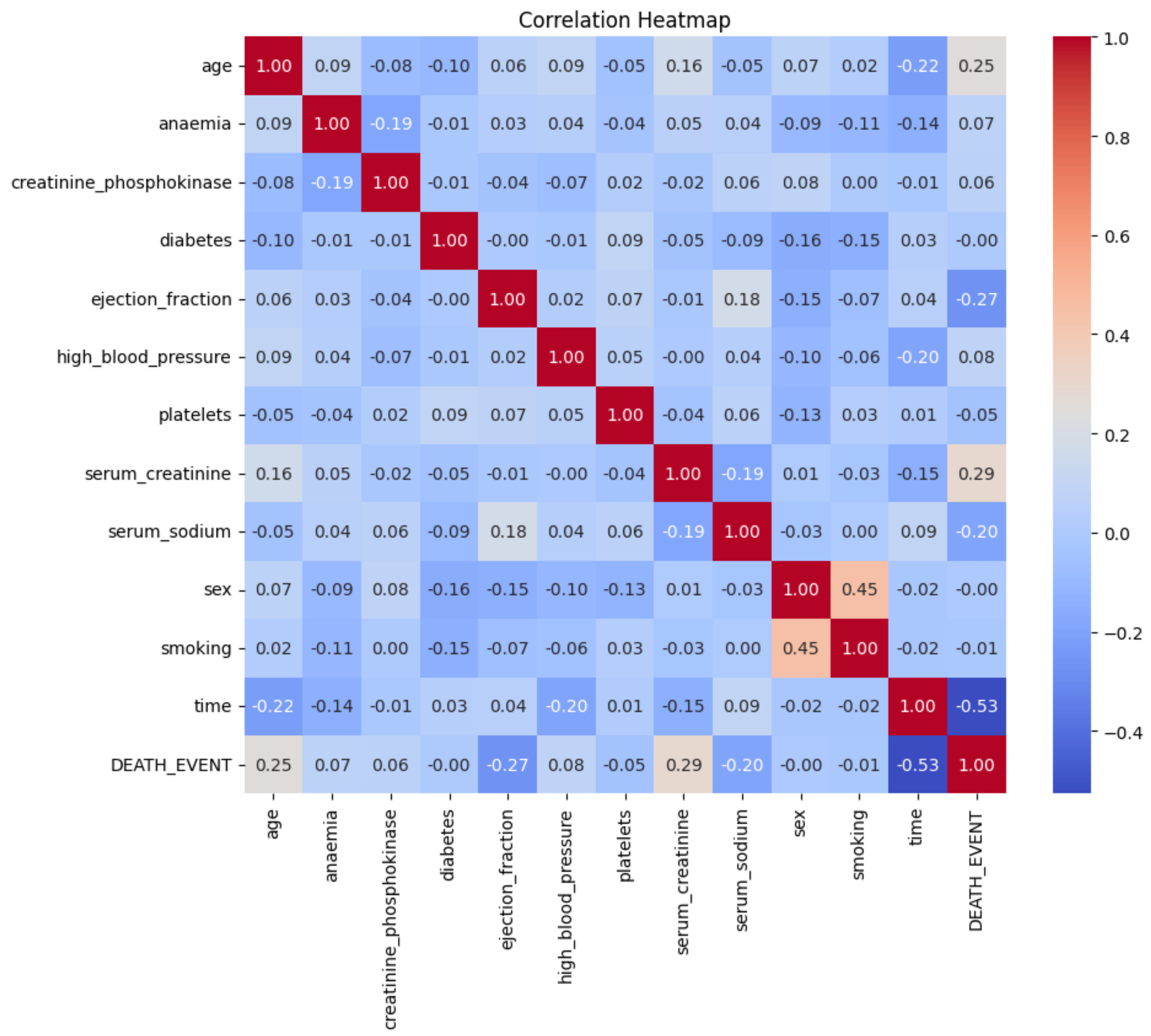
4.2.2 Joint Distribution Analys iS

```
*[14]: import numpy as np
       import pandas as pd
       import seaborn as sns
       import matplotlib.pyplot as plt
       from scipy import stats
       heartdataset = pd.read_csv(r"C:\Users\K.uday\Desktop\heart_failure_clinical_records_dataset.csv")
       print(heartdataset.head())
```

```
    age  anaemia  creatinine_phosphokinase  diabetes  ejection_fraction  \
0  75.0        0                       582         0                 20
1  55.0        0                      7861         0                 38
2  65.0        0                       146         0                 20
3  50.0        1                       111         0                 20
4  65.0        1                       160         1                 20

   high_blood_pressure  platelets  serum_creatinine  serum_sodium  sex  \
0                    1  265000.00               1.9           130    1
1                    0  263358.03               1.1           136    1
2                    0  162000.00               1.3           129    1
3                    0  210000.00               1.9           137    1
4                    0  327000.00               2.7           116    0

   smoking  time  DEATH_EVENT
0        0     4            1
1        0     6            1
2        1     7            1
3        0     7            1
4        0     8            1
```

```
[16]: age_bins = pd.cut(heartdataset['age'], bins=5)
      death_event_counts = heartdataset.groupby([age_bins, 'DEATH_EVENT']).size().unstack()
      death_event_counts.plot(kind='bar', stacked=True)
      plt.title('Death Event by Age Group')
      plt.xlabel('Age Group')
      plt.ylabel('Count')
      plt.xticks(rotation=45)
      plt.legend(title='Death Event', loc='upper right')
      plt.show()
      plt.figure(figsize=(10, 8))
      sns.heatmap(heartdataset.corr(), annot=True, cmap='coolwarm', fmt=".2f")
      plt.title('Correlation Heatmap')
      plt.show()
      shapiro_test_result = stats.shapiro(heartdataset['age'])
      print("Shapiro-Wilk test p-value for 'age':", shapiro_test_result.pvalue)
      if shapiro_test_result.pvalue < 0.05:
          print("The 'age' column does not follow a normal distribution.")
      else:
          print("The 'age' column follows a normal distribution.")
      ks_test_result = stats.kstest(heartdataset['creatinine_phosphokinase'], 'norm')
      print("Kolmogorov-Smirnov test p-value for 'creatinine_phosphokinase':", ks_test_result.pvalue)
      if ks_test_result.pvalue < 0.05:
          print("The 'creatinine_phosphokinase' column does not follow a normal distribution.")
      else:
          print("The 'creatinine_phosphokinase' column follows a normal distribution.")
```



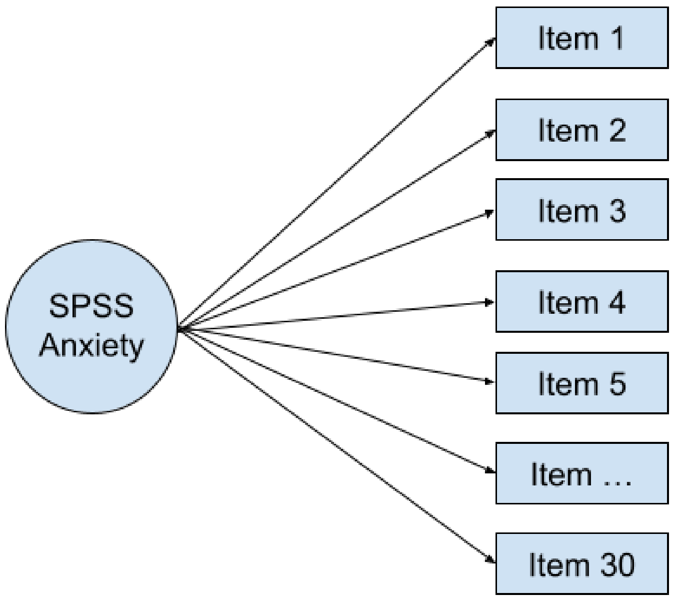


	age	anaemia	creatinine_phosphokinase	diabetes	ejection_fraction	high_blood_pressure	platelets	serum_creatinine	serum_sodium	sex	smoking	time	DEATH_EVENT
age	1.00	0.09	-0.08	-0.10	0.06	0.09	-0.05	0.16	-0.05	0.07	0.02	-0.22	0.25
anaemia	0.09	1.00	-0.19	-0.01	0.03	0.04	-0.04	0.05	0.04	-0.09	-0.11	-0.14	0.07
creatinine_phosphokinase	-0.08	-0.19	1.00	-0.01	-0.04	-0.07	0.02	-0.02	0.06	0.08	0.00	-0.01	0.06
diabetes	-0.10	-0.01	-0.01	1.00	-0.00	-0.01	0.09	-0.05	-0.09	-0.16	-0.15	0.03	-0.00
ejection_fraction	0.06	0.03	-0.04	-0.00	1.00	0.02	0.07	-0.01	0.18	-0.15	-0.07	0.04	-0.27
high_blood_pressure	0.09	0.04	-0.07	-0.01	0.02	1.00	0.05	-0.00	0.04	-0.10	-0.06	-0.20	0.08
platelets	-0.05	-0.04	0.02	0.09	0.07	0.05	1.00	-0.04	0.06	-0.13	0.03	0.01	-0.05
serum_creatinine	0.16	0.05	-0.02	-0.05	-0.01	-0.00	-0.04	1.00	-0.19	0.01	-0.03	-0.15	0.29
serum_sodium	-0.05	0.04	0.06	-0.09	0.18	0.04	0.06	-0.19	1.00	-0.03	0.00	0.09	-0.20
sex	0.07	-0.09	0.08	-0.16	-0.15	-0.10	-0.13	0.01	-0.03	1.00	0.45	-0.02	-0.00
smoking	0.02	-0.11	0.00	-0.15	-0.07	-0.06	0.03	-0.03	0.00	0.45	1.00	-0.02	-0.01
time	-0.22	-0.14	-0.01	0.03	0.04	-0.20	0.01	-0.15	0.09	-0.02	-0.02	1.00	-0.53
DEATH_EVENT	0.25	0.07	0.06	-0.00	-0.27	0.08	-0.05	0.29	-0.20	-0.00	-0.01	-0.53	1.00

```
Shapiro-Wilk test p-value for 'age': 5.3476593166124076e-05
The 'age' column does not follow a normal distribution.
Kolmogorov-Smirnov test p-value for 'creatinine_phosphokinase': 0.0
The 'creatinine_phosphokinase' column does not follow a normal
distribution.
```

[17]:

4.2.3 Factor Analysis



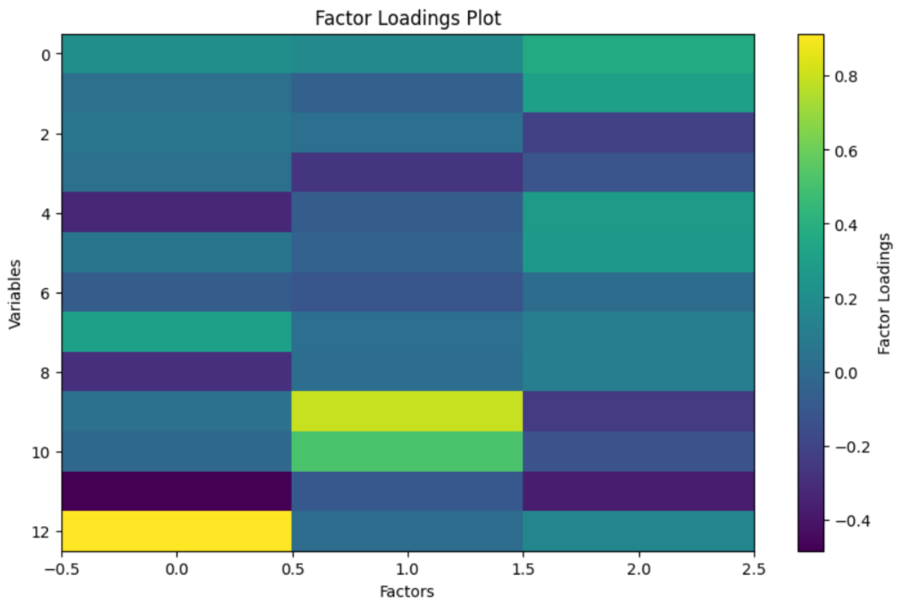
```python
import numpy as np
import pandas as pd
heartdataset = pd.read_csv(r"C:\Users\K.uday\Desktop\heart_failure_clinical_records_dataset.csv")
heartdataset.head()
```

	age	anaemia	creatinine_phosphokinase	diabetes	ejection_fraction	high_blood_pressure	platelets	serum_creatinine	serum_sodium	sex	smoking	time	DEATH_E
0	75.0	0	582	0	20	1	265000.00	1.9	130	1	0	4	
1	55.0	0	7861	0	38	0	263358.03	1.1	136	1	0	6	
2	65.0	0	146	0	20	0	162000.00	1.3	129	1	1	7	
3	50.0	1	111	0	20	0	210000.00	1.9	137	1	0	7	
4	65.0	1	160	1	20	0	327000.00	2.7	116	0	0	8	

```python
from factor_analyzer import FactorAnalyzer
numeric_data = heartdataset.drop(columns=['sex', 'smoking', 'DEATH_EVENT'])
fa = FactorAnalyzer()
fa.fit(numeric_data, 5)
loadings = fa.loadings_
print(loadings)
```

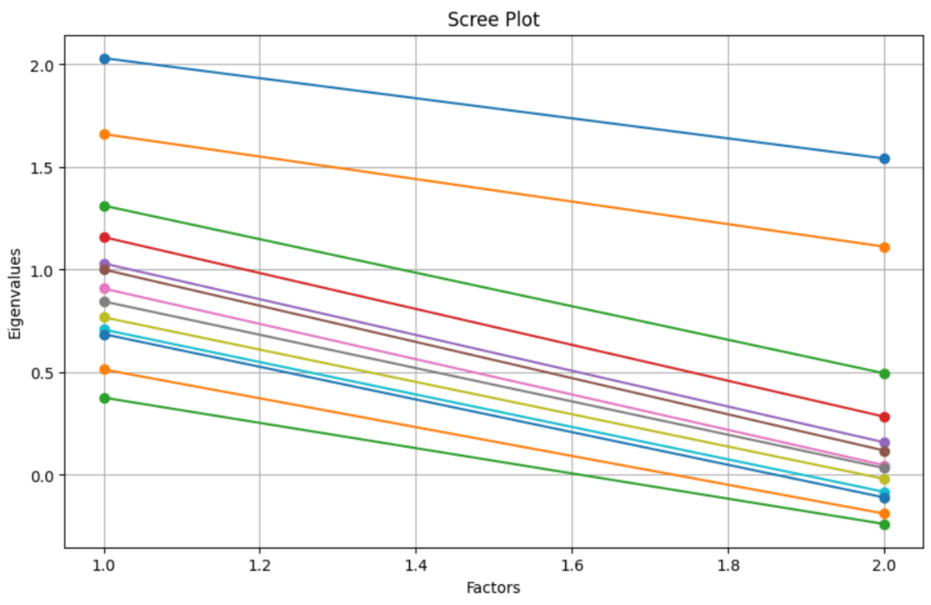
```
[[-0.00714611 -0.0372472   0.42867049]
 [-0.15695344  0.04713746  0.19492784]
 [ 1.04561962  0.00693246  0.18394655]
 [-0.03735103 -0.07631622 -0.15380492]
 [-0.04149474  0.1833401   0.04087172]
 [-0.01820547  0.04992917  0.25237545]
 [ 0.01701439  0.06867491 -0.04662397]
 [ 0.0242945  -0.18888618  0.24589222]
 [ 0.08966323  1.00203264  0.12282572]
 [-0.10799574  0.08840636 -0.58587845]]
```

•[7]:
```python
data_numeric = data.select_dtypes(include=['number'])


correlation_matrix = data_numeric.corr()S

fa = FactorAnalyzer(n_factors=3, rotation='varimax')
fa.fit(data_numeric)
factor_loadings = fa.loadings_


plt.figure(figsize=(10, 6))
plt.imshow(factor_loadings, cmap='viridis', aspect='auto')
plt.colorbar(label='Factor Loadings')
plt.xlabel('Factors')
plt.ylabel('Variables')
plt.title('Factor Loadings Plot')
plt.show()
```



•[6]:

```
data_numeric = data.select_dtypes(include=['number'])
fa = FactorAnalyzer(rotation='varimax')
fa.fit(data_numeric)
eigenvalues = fa.get_eigenvalues()S
plt.figure(figsize=(10, 6))
plt.plot(range(1, len(eigenvalues) + 1), eigenvalues, marker='o', linestyle='-')
plt.title('Scree Plot')
plt.xlabel('Factors')
plt.ylabel('Eigenvalues')
plt.grid(True)
plt.show()
```



```
conclusion
```

```
Based on the provided dataset, it appears to be a medical dataset with various health-related variables such as age, anaemia, creatinine_phosphokinase,
diabetes, ejection_fraction, high_blood_pressure, platelets, serum_creatinine, serum_sodium, sex, smoking, and time. The target variable is DEATH_EVENT,
which is a binary variable indicating whether a death event occurred or not.

To analyze this dataset using Bayes' theorem, we can calculate the probability of a death event given a certain variable, such as age or diabetes. For
example, we can calculate the probability of a death event given that a patient has diabetes (P(DEATH_EVENT|diabetes)) and compare it to the probability
of a death event given that a patient does not have diabetes (P(DEATH_EVENT|no diabetes)). This can help us understand the relationship between diabetes
and the risk of death.

We can also analyze the joint distribution of variables in this dataset. For example, we can examine the correlation between age and ejection_fraction,
or between smoking and serum_creatinine. Visualizing these relationships can help us better understand the data and identify any patterns or trends.

To check whether the data follows a normal distribution, we can use statistical tests such as the Kolmogorov-Smirnov test or the Shapiro-Wilk test.
These tests can help us determine whether the data is normally distributed, which is an important assumption for many statistical analyses.

In conclusion, this dataset provides a rich source of information for analyzing medical data using Bayes' theorem, joint distribution analysis, and
normality tests. By analyzing this data, we can gain insights into the relationships between various health-related variables and the risk of death, as
well as identify any patterns or trends in the data.
```