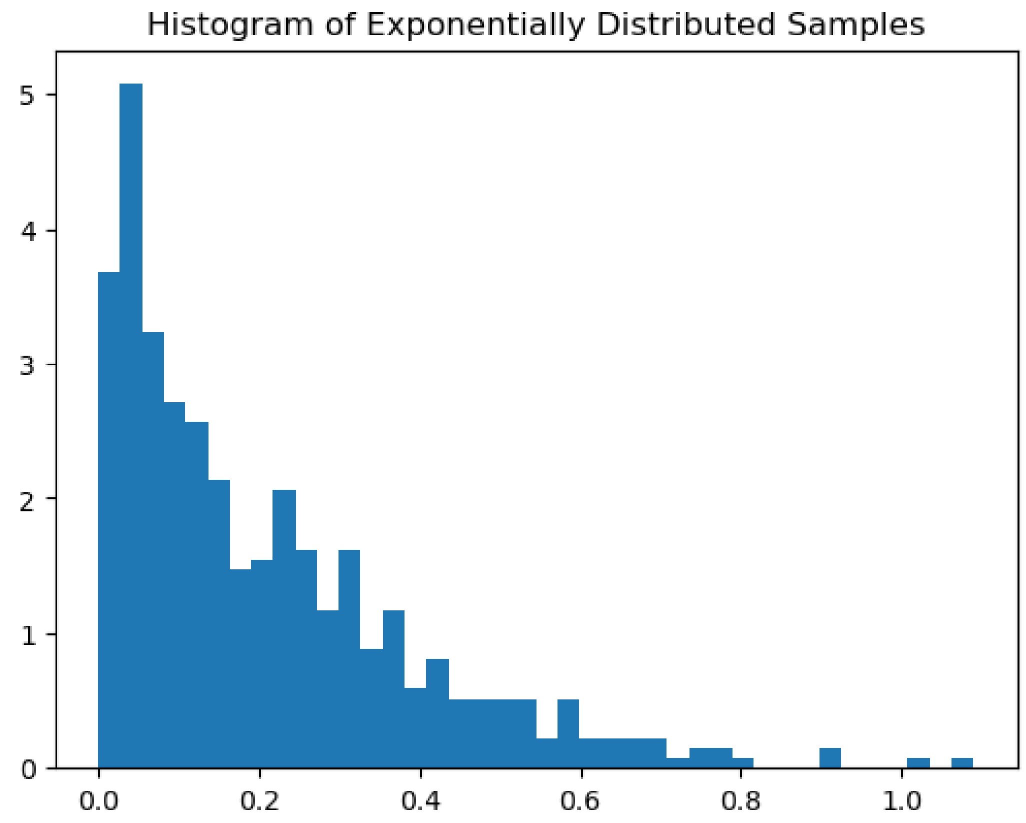
```python
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import scipy.stats
import seaborn as sns
```

4.1.1 Simulating Continuous Random Variables:

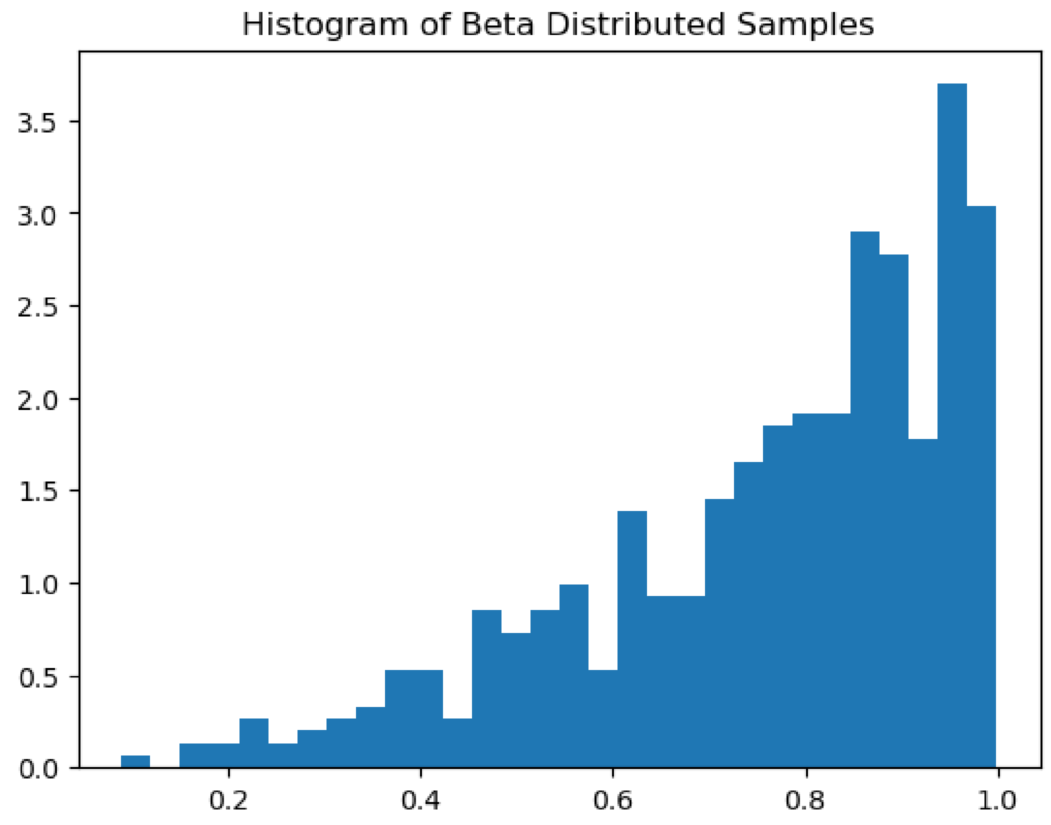
Inverse Transform Sampling for Exponential Distribution

```python
# consider the lambda parameter for the exponential distribution
lambda_param = 5.0

# Inverse transform sampling
def inverse_transform_sampling(lambda_param, n=500):
    uniform_samples = np.random.uniform(0, 1, n)
    exponential_samples = -np.log(1 - uniform_samples) / lambda_param
    return exponential_samples

# Generating the samples
samples = inverse_transform_sampling(lambda_param)

# Plotting the graph of exponential distributed sample
plt.hist(samples, bins=40, density=True)
plt.title('Histogram of Exponentially Distributed Samples')
plt.show()
```



Acceptance-Rejection Sampling for Beta Distribution

```
def acceptance_rejection_sampling(n=500):
    samples = []
    while len(samples) < n:
        y = np.random.uniform(0, 1)
        u = np.random.uniform(0, 1)
        if u <= y**2:  # condition for beta distribution
            samples.append(y)
    return np.array(samples)

# Generating samples
beta_samples = acceptance_rejection_sampling()

# Plotting the graph of Beta Distributed samples
plt.hist(beta_samples, bins=30, density=True)
plt.title('Histogram of Beta Distributed Samples')
plt.show()
```



Statistical Analysis of Simulated Data

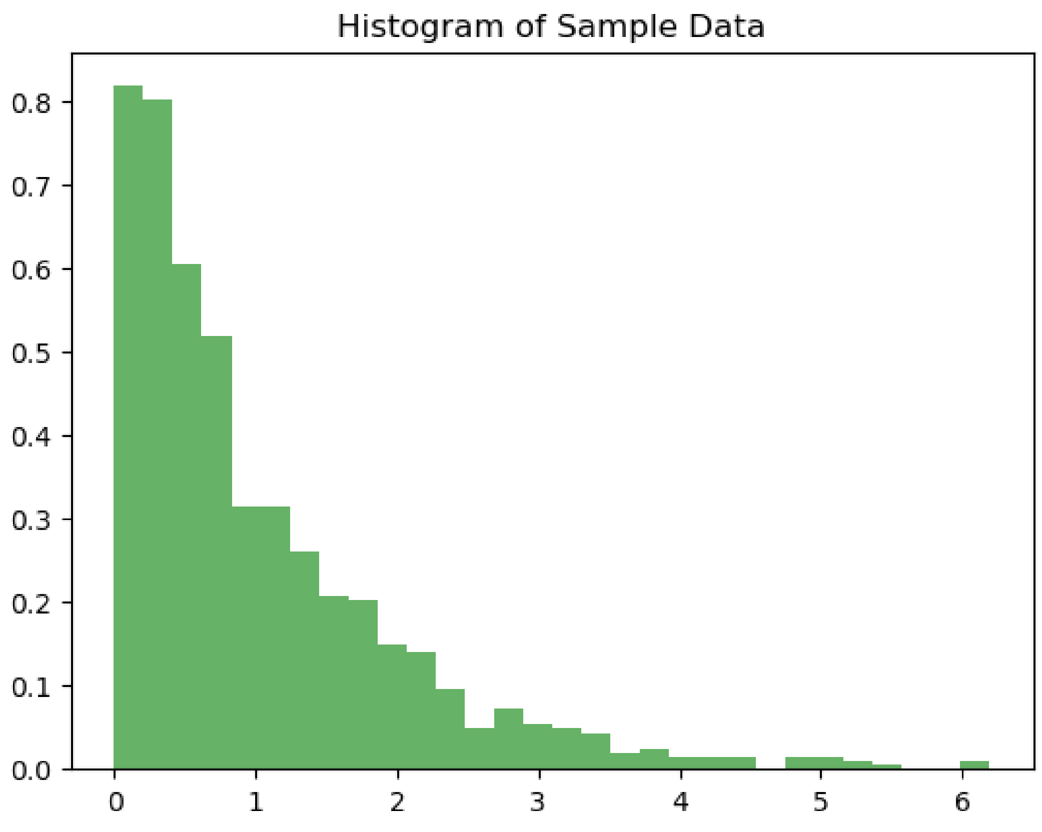
```
def statistical_analysis(samples):
    mean = np.mean(samples)
    variance = np.var(samples)
    std_dev = np.std(samples)
    first_quantile = np.percentile(samples, 25)
    third_quantile = np.percentile(samples, 75)
    skewness = scipy.stats.skew(samples)
    kurtosis = scipy.stats.kurtosis(samples)

    print(f"Mean: {mean}")
    print(f"Variance: {variance}")
    print(f"Standard Deviation: {std_dev}")
    print(f"First Quantile: {first_quantile}")
    print(f"Third Quantile: {third_quantile}")
    print(f"Skewness: {skewness}")
    print(f"Kurtosis: {kurtosis}")

# Calculate statistics for exponential samples
statistical_analysis(samples)
```

```
Mean: 0.20720577704918502
Variance: 0.035260434016763015
Standard Deviation: 0.18777761851925542
```

```
First Quantile: 0.06087084980437777
Third Quantile: 0.30603635432541215
Skewness: 1.387321432652219
Kurtosis: 2.127627376911736
```

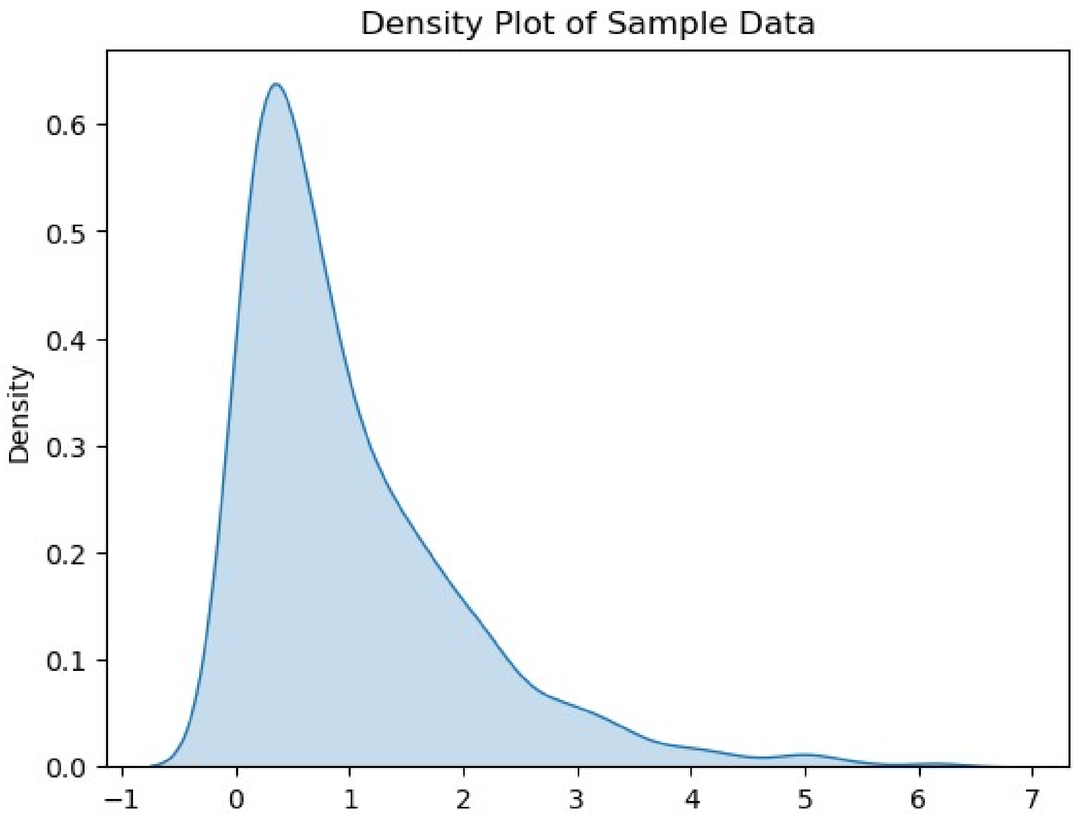
Visualization of Sample Data

```
#Histogram of sample data
plt.hist(samples, bins=30, density=True, alpha=0.6, color='g')
plt.title('Histogram of Sample Data')
plt.show()

# Density Plot of sample data
sns.kdeplot(samples, shade=True)
plt.title('Density Plot of Sample Data')
plt.show()
```



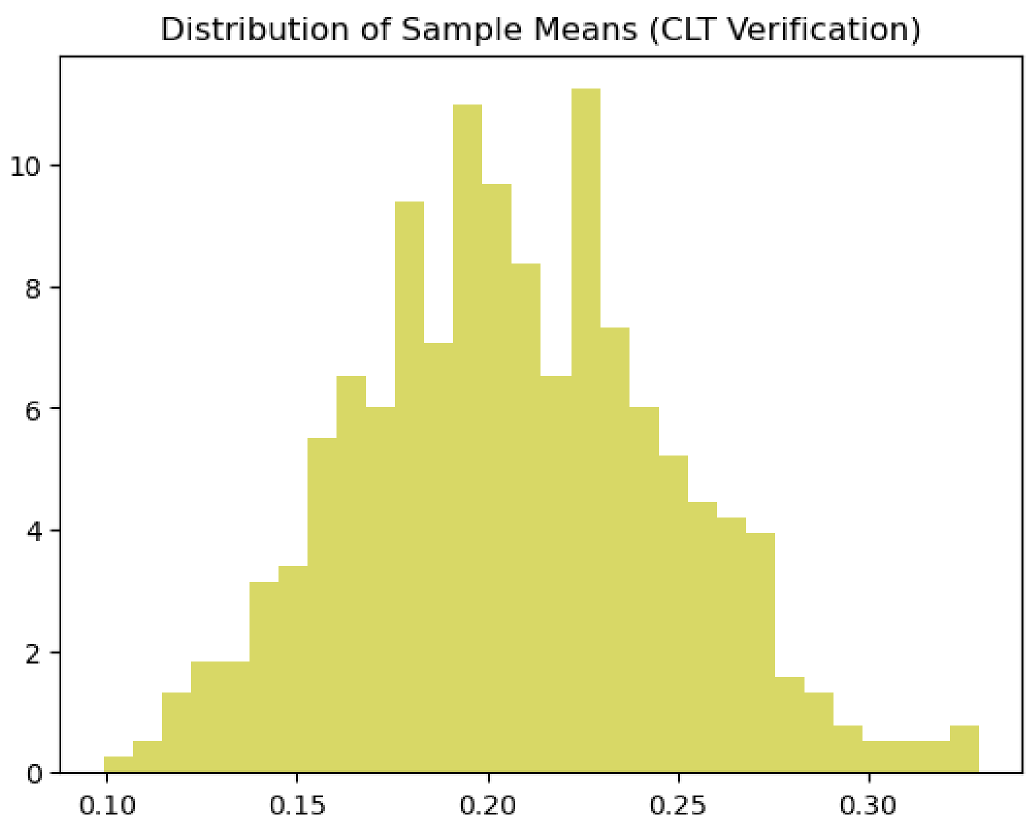
```
C:\Users\hp\AppData\Local\Temp\ipykernel_20924\4137198264.py:7:
FutureWarning:

`shade` is now deprecated in favor of `fill`; setting `fill=True`.
This will become an error in seaborn v0.14.0; please update your code.
```

```
sns.kdeplot(samples, shade=True)
```



Verifying Central Limit Theorem

```
def verify_clt(samples, n=20, num_samples=500):
    sample_means = []
    for _ in range(num_samples):
        sample = np.random.choice(samples, n)
        sample_means.append(np.mean(sample))

    # Plotting the sample means
    plt.hist(sample_means, bins=30, density=True, alpha=0.6,
color='y')
    plt.title('Distribution of Sample Means (CLT Verification)')
    plt.show()

verify_clt(samples)
```



Outlier Detection

```
def detect_outliers(data, m=4):
    mean = np.mean(data)
    std = np.std(data)
    outliers = [x for x in data if abs(x - mean) > m * std]
    return outliers

outliers = detect_outliers(samples)
print(f"Number of Outliers: {len(outliers)}")
```

```
Number of Outliers: 2
```

Probability Calculations

```
def probability_calculation(samples, lower_bound, upper_bound):
    within_range = [x for x in samples if lower_bound <= x <=
upper_bound]
    probability = len(within_range) / len(samples)
    return probability

# Probability of a value falling between two bounds
prob = probability_calculation(samples, 0.5, 1.5)
print(f"Probability: {prob}")
```

```
Probability: 0.086
```

4.1.2 Simulating from Discrete Distributions

```python
from scipy import stats

# Simulating data from various discrete distributions
#Taking the random data
binomial_data = np.random.binomial(n=17, p=0.5, size=500)
poisson_data = np.random.poisson(lam=3.0, size=500)
geometric_data = np.random.geometric(p=0.2, size=500)
```

Statistical Analysis

```python
def analyze_distribution(data):
    analysis_results = {
        'Mean': np.mean(data),
        'Variance': np.var(data, ddof=1),
        'Standard Deviation': np.std(data, ddof=1),
        'First Quantile (25%)': np.quantile(data, 0.25),
        'Third Quantile (75%)': np.quantile(data, 0.75),
        'Mode': stats.mode(data)[0][0],
        'Skewness': stats.skew(data),
        'Kurtosis': stats.kurtosis(data)
    }
    return analysis_results

# considering the binomial data
binomial_data = np.random.binomial(n=17, p=0.5, size=500)
analysis_binomial = analyze_distribution(binomial_data)
print("Statistical Analysis of Binomial Distribution:")
for key, value in analysis_binomial.items():
    print(f"{key}: {value:.2f}")
```

```
Statistical Analysis of Binomial Distribution:
Mean: 8.45
Variance: 3.77
Standard Deviation: 1.94
First Quantile (25%): 7.00
Third Quantile (75%): 10.00
Mode: 9.00
Skewness: 0.05
Kurtosis: -0.38

C:\Users\hp\AppData\Local\Temp\ipykernel_20924\1632294513.py:8:
FutureWarning: Unlike other reduction functions (e.g. `skew`,
`kurtosis`), the default behavior of `mode` typically preserves the
axis it acts along. In SciPy 1.11.0, this behavior will change: the
```

```
default value of `keepdims` will become False, the `axis` over which
the statistic is taken will be eliminated, and the value None will no
longer be accepted. Set `keepdims` to True or False to avoid this
warning.
  'Mode': stats.mode(data)[0][0],
```

Visualization

```python
def visualize_data(data, title):
    plt.figure(figsize=(15, 10))
    sns.histplot(data, kde=True)
    plt.title(title)
    plt.show()
```

Central Limit Theorem Verification

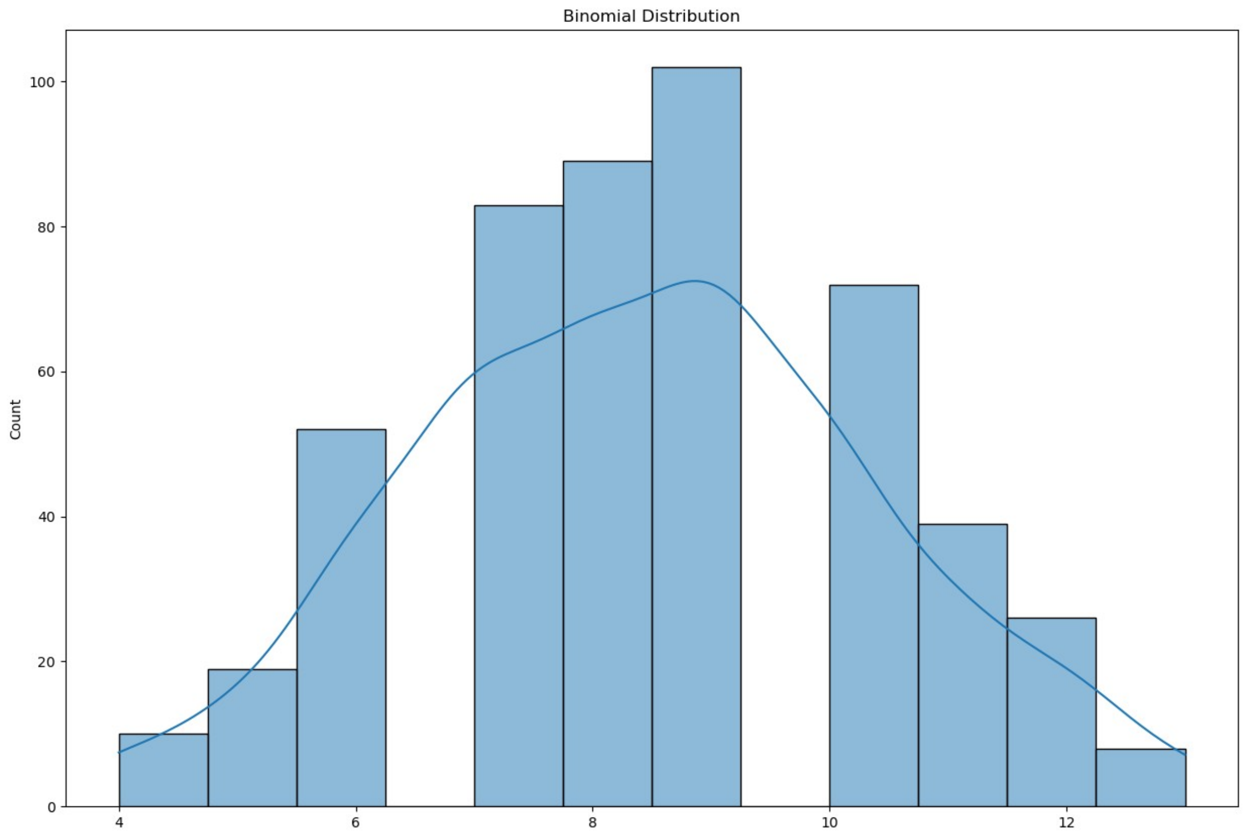
```python
def clt_verification(data, sample_size=30, num_samples=500):
    sample_means = []
    for _ in range(num_samples):
        sample = np.random.choice(data, size=sample_size,
replace=True)
        sample_means.append(np.mean(sample))
    visualize_data(sample_means, "Sample Means Distribution")
```

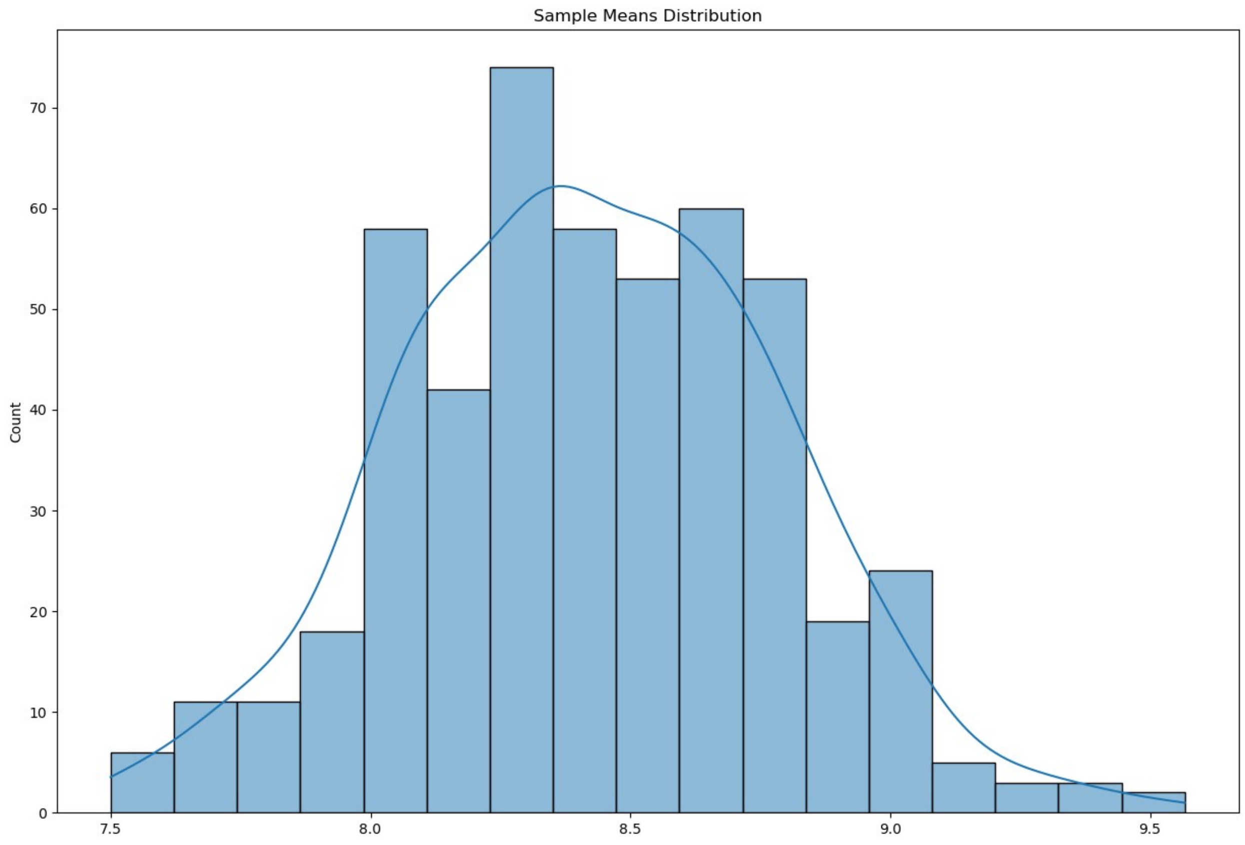
Outlier Detection (Simple Z-score method)

```python
def detect_outliers(data):
    threshold = 3
    mean = np.mean(data)
    std = np.std(data)
    outliers = []
    for i in data:
        z_score = (i - mean) / std
        if np.abs(z_score) > threshold:
            outliers.append(i)
    return outliers

analyze_binomial = analyze_distribution(binomial_data)
visualize_data(binomial_data, "Binomial Distribution")
clt_verification(binomial_data)
outliers = detect_outliers(binomial_data)
```

```
C:\Users\hp\AppData\Local\Temp\ipykernel_20924\1632294513.py:8:
FutureWarning: Unlike other reduction functions (e.g. `skew`,
`kurtosis`), the default behavior of `mode` typically preserves the
axis it acts along. In SciPy 1.11.0, this behavior will change: the
default value of `keepdims` will become False, the `axis` over which
the statistic is taken will be eliminated, and the value None will no
longer be accepted. Set `keepdims` to True or False to avoid this
```

```
warning.
  'Mode': stats.mode(data)[0][0],
```

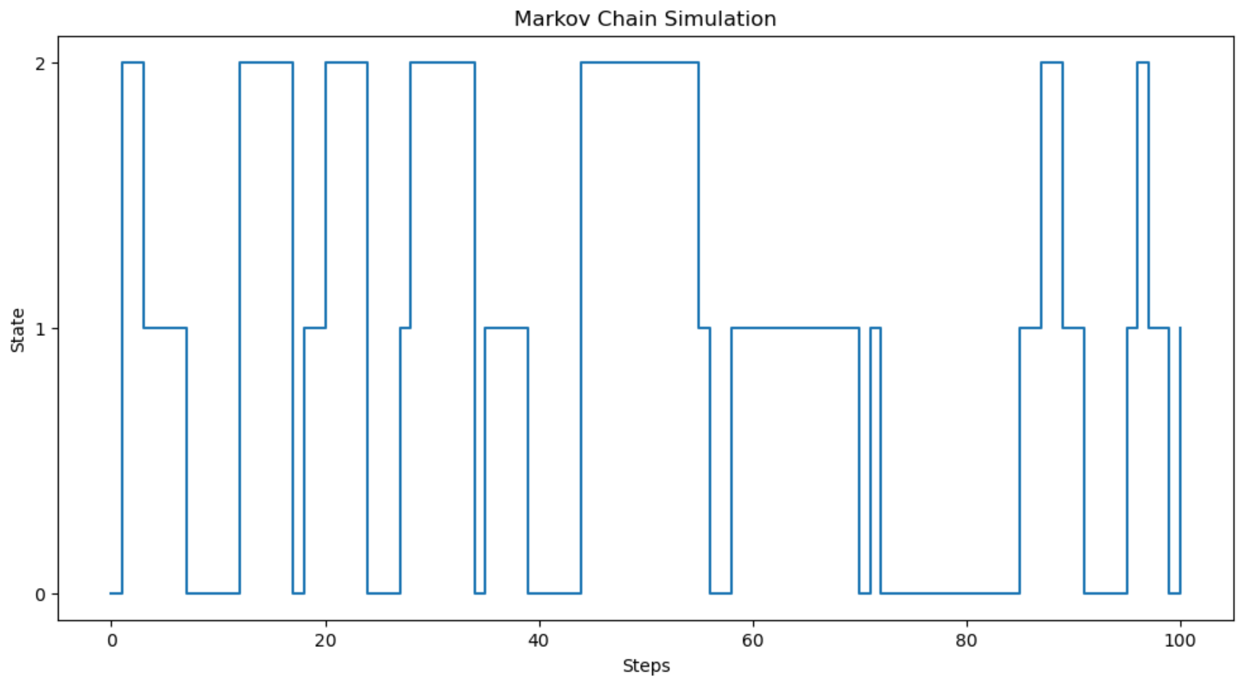




4.1.3 Markov Chains:

```
# Take any simple transition matrix for the Markov chain

transition_matrix = np.array([[0.7, 0.2, 0.1],
                              [0.3, 0.5, 0.2],
                              [0.1, 0.1, 0.8]])

# Function to simulate Markov chain
def simulate_markov_chain(transition_matrix, initial_state, steps):
    state = initial_state
    states_visited = [state]

    for _ in range(steps):
        state = np.random.choice(range(len(transition_matrix)),
p=transition_matrix[state])
        states_visited.append(state)

    return states_visited

# Simulating the Markov chain
initial_state = 0 # This is the starting stage
steps = 100
```

```
states_visited = simulate_markov_chain(transition_matrix,
initial_state, steps)

# Visualization
plt.figure(figsize=(12, 6))
plt.plot(states_visited, drawstyle='steps-post')
plt.title('Markov Chain Simulation')
plt.xlabel('Steps')
plt.ylabel('State')
plt.yticks(range(len(transition_matrix)))
plt.show()

# Ergodicity analysis: Comparing time-averaged behavior with steady-
state probabilities
time_averaged_behavior = np.mean([states_visited.count(state) /
len(states_visited) for state in range(len(transition_matrix))])

# Steady-state probabilities (assuming ergodicity)
steady_state_probabilities = np.linalg.matrix_power(transition_matrix,
1000)[0]

# Sensitivity Analysis: Varying transition probabilities or initial
conditions (not shown in this code snippet)

# Output results
print("Time-averaged behavior of states:", time_averaged_behavior)
print("Steady-state probabilities:", steady_state_probabilities)
```



```
Time-averaged behavior of states: 0.3333333333333333
Steady-state probabilities: [0.36363636 0.22727273 0.40909091]
```

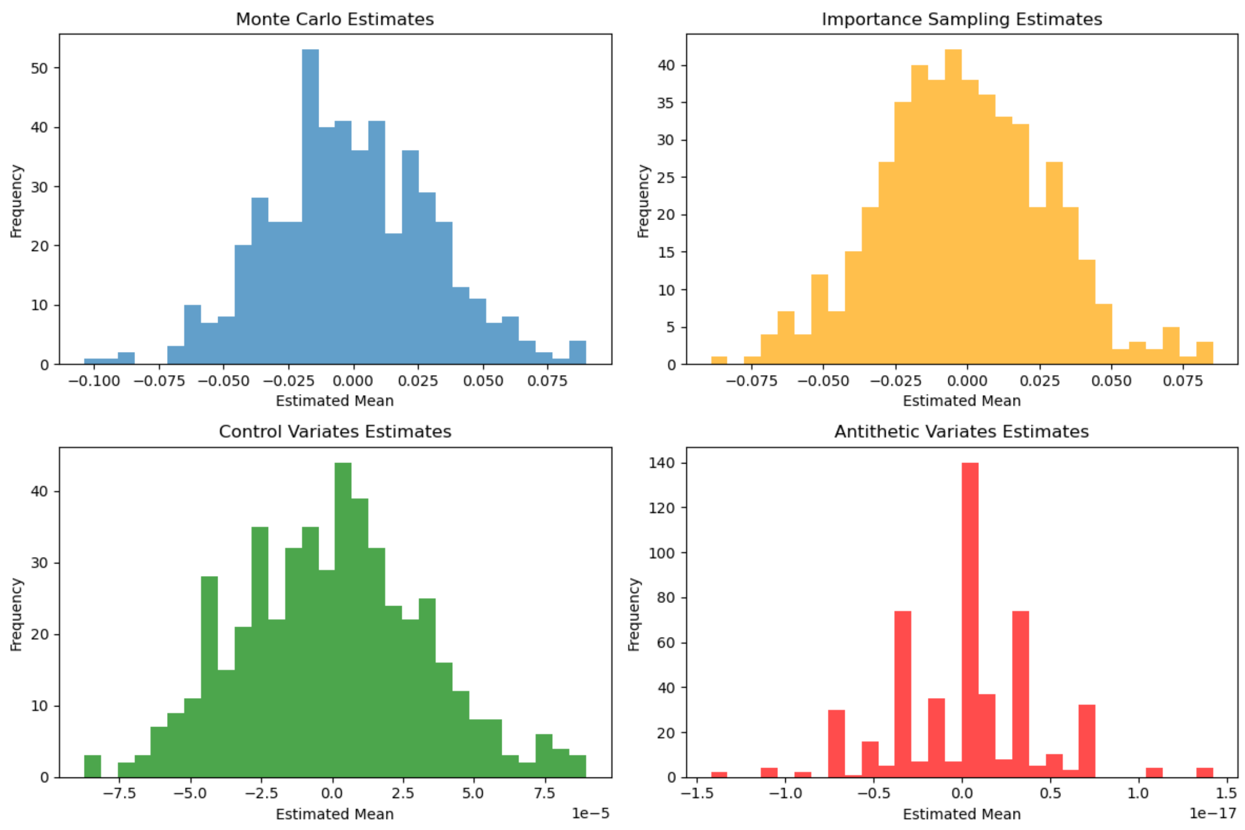
4.1.4 Variance Reduction Techniques:

Basic Monte Carlo Simulation to estimate the mean

```python
def monte_carlo_mean(samples):
    return np.mean(samples)

# Importance Sampling
def importance_sampling(target_mean, target_std, sampling_std,
num_samples):

    # Generate samples from a different distribution
    samples = np.random.normal(target_mean, sampling_std, num_samples)

    # Reweight the samples
    weights = (stats.norm.pdf(samples, target_mean, target_std) /
               stats.norm.pdf(samples, target_mean, sampling_std))
    weighted_mean = np.sum(weights * samples) / np.sum(weights)
    return weighted_mean

# Control Variates
def control_variates(target_mean, num_samples):

    # Generate samples
    samples = np.random.normal(target_mean, 1, num_samples)

    # Control variable with  target mean
    control_mean = target_mean

    # Calculating covariance and variance
    covariance = np.cov(samples, samples)
    variance = np.var(samples)

    # Compute the control variate coefficient (b)
    b = covariance[0, 1] / variance

    # Adjusted mean using control variate
    adjusted_mean = np.mean(samples - b * (samples - control_mean))
    return adjusted_mean

# Antithetic Variates
def antithetic_variates(target_mean, num_samples):

    # Generate samples
    samples = np.random.normal(target_mean, 1, num_samples // 2)

    # Generate antithetic samples
    antithetic_samples = target_mean - (samples - target_mean)

    # Combine and calculate mean
    combined_samples = np.concatenate((samples, antithetic_samples))
    antithetic_mean = np.mean(combined_samples)
    return antithetic_mean

# Parameters
target_mean = 0
target_std = 1
sampling_std = 2
num_samples = 1000

# Running the simulations
mc_mean = monte_carlo_mean(np.random.normal(target_mean, target_std,
num_samples))
is_mean = importance_sampling(target_mean, target_std, sampling_std,
num_samples)
cv_mean = control_variates(target_mean, num_samples)
av_mean = antithetic_variates(target_mean, num_samples)

# Output results
print("Monte Carlo Mean:", mc_mean)
print("Importance Sampling Mean:", is_mean)
print("Control Variates Mean:", cv_mean)
print("Antithetic Variates Mean:", av_mean)
```

```
Monte Carlo Mean: 0.03777138155393885
Importance Sampling Mean: 0.047125861260050084
Control Variates Mean: 5.021286532227588e-05
Antithetic Variates Mean: 0.0
```

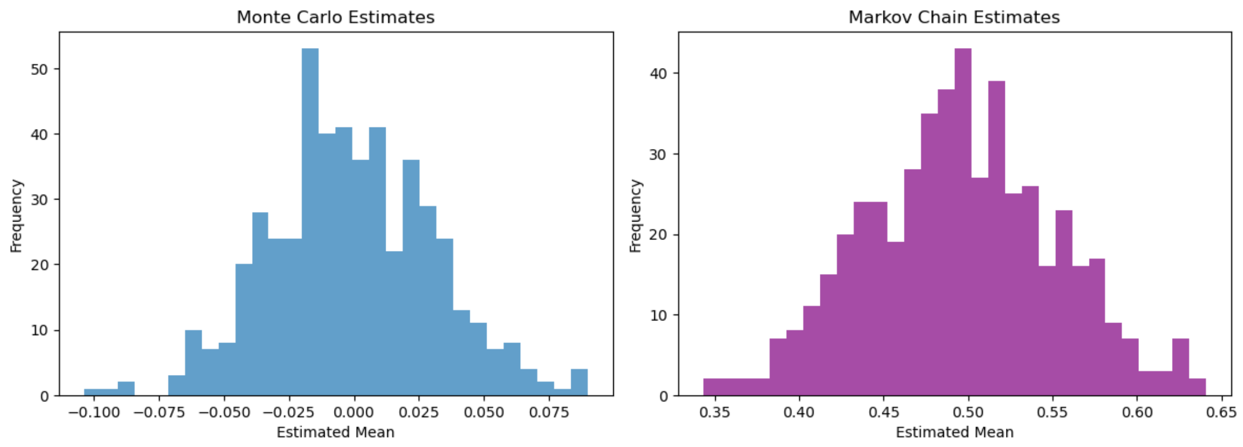
```python
# Number of repetitions for each simulation to generate distributions
num_repetitions = 500

# Running the simulations multiple times
mc_estimates = [monte_carlo_mean(np.random.normal(target_mean,
target_std, num_samples)) for _ in range(num_repetitions)]
is_estimates = [importance_sampling(target_mean, target_std,
sampling_std, num_samples) for _ in range(num_repetitions)]
cv_estimates = [control_variates(target_mean, num_samples) for _ in
range(num_repetitions)]
av_estimates = [antithetic_variates(target_mean, num_samples) for _ in
range(num_repetitions)]

# Plotting the subplots of distributions of estimates
plt.figure(figsize=(12, 8))
plt.subplot(2, 2, 1)
plt.hist(mc_estimates, bins=30, alpha=0.7, label='Monte Carlo')
plt.title('Monte Carlo Estimates')
plt.xlabel('Estimated Mean')
plt.ylabel('Frequency')

plt.subplot(2, 2, 2)
plt.hist(is_estimates, bins=30, alpha=0.7, color='orange',
label='Importance Sampling')
plt.title('Importance Sampling Estimates')
plt.xlabel('Estimated Mean')
plt.ylabel('Frequency')

plt.subplot(2, 2, 3)
plt.hist(cv_estimates, bins=30, alpha=0.7, color='green',
label='Control Variates')
plt.title('Control Variates Estimates')
plt.xlabel('Estimated Mean')
plt.ylabel('Frequency')

plt.subplot(2, 2, 4)
plt.hist(av_estimates, bins=30, alpha=0.7, color='red',
label='Antithetic Variates')
plt.title('Antithetic Variates Estimates')
plt.xlabel('Estimated Mean')
plt.ylabel('Frequency')

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



4.1.5 Comparison of Different Simulation Methods:

```
# Markov Chain Simulation for estimating the mean

def markov_chain_mean(transition_matrix, initial_state, target_mean,
steps):
    state = initial_state
    values = [np.random.normal(target_mean, 1)]

    for _ in range(steps):
        state = np.random.choice(range(len(transition_matrix)),
p=transition_matrix[state])
        values.append(np.random.normal(target_mean + state, 1))  #
Shift the mean based on the state

    return np.mean(values)

# Define the transition matrix for the Markov Chain
transition_matrix = np.array([[0.7, 0.3],
                              [0.3, 0.7]])
```

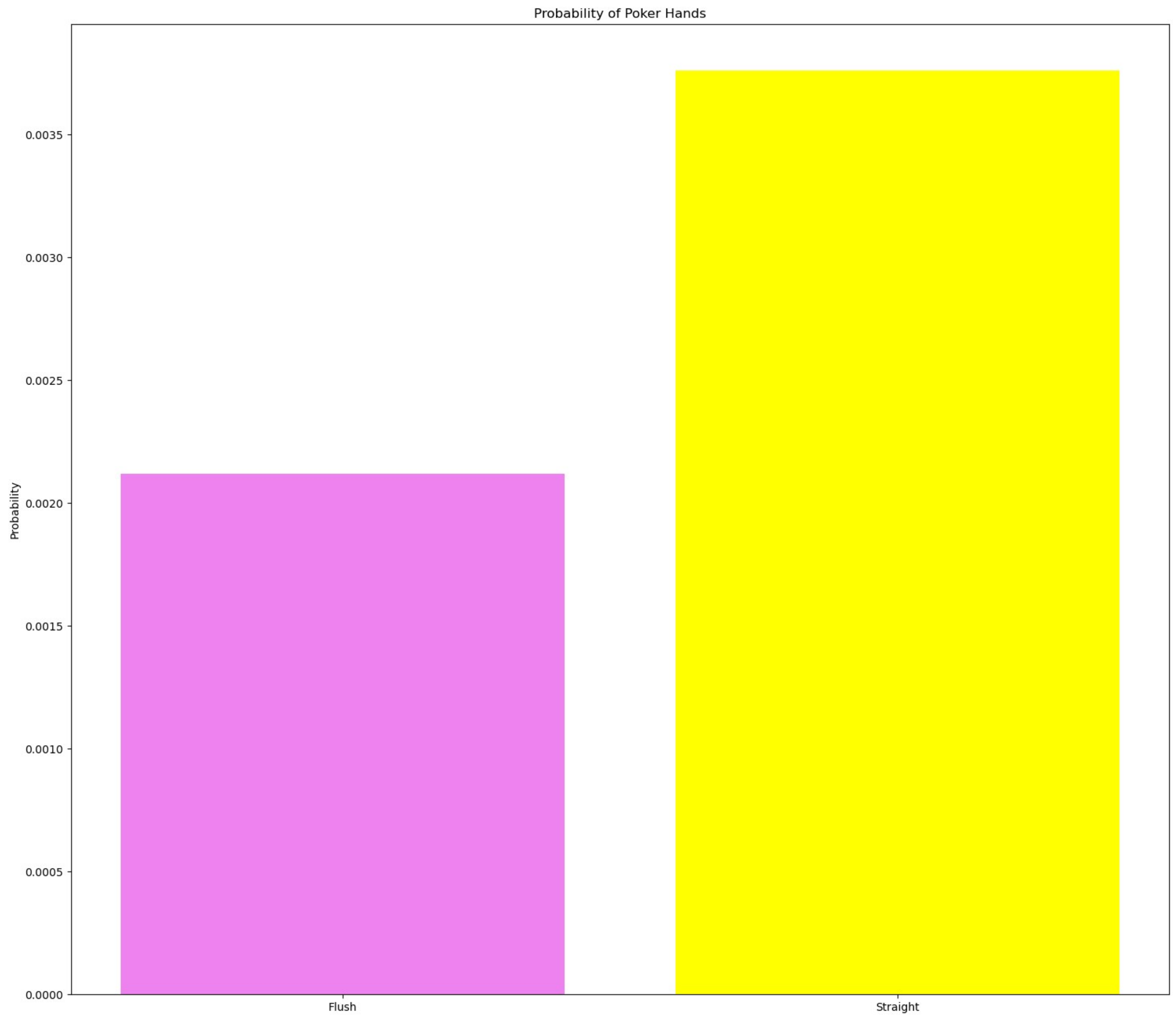
```python
# Parameters for the simulations
target_mean = 0
num_samples = 500
num_repetitions = 500

# Running the Markov Chain simulation multiple times
mc_chain_estimates = [markov_chain_mean(transition_matrix, 0,
target_mean, num_samples) for _ in range(num_repetitions)]

# Plotting the distributions of estimates
plt.figure(figsize=(12, 8))

plt.subplot(2, 2, 1)
plt.hist(mc_estimates, bins=30, alpha=0.7, label='Monte Carlo')
plt.title('Monte Carlo Estimates')
plt.xlabel('Estimated Mean')
plt.ylabel('Frequency')

plt.subplot(2, 2, 2)
plt.hist(mc_chain_estimates, bins=30, alpha=0.7, color='purple',
label='Markov Chain')
plt.title('Markov Chain Estimates')
plt.xlabel('Estimated Mean')
plt.ylabel('Frequency')

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



4.1.6 Simulation for Combinatorial Analysis:

```python
import itertools

# Function to create a standard deck of cards
def create_deck():
    suits = ['Hearts', 'Diamonds', 'Clubs', 'Spades']
    ranks = ['2', '3', '4', '5', '6', '7', '8', '9', '10', 'J', 'Q',
'K', 'A']
    return list(itertools.product(ranks, suits))

# Function to check if a hand is a flush i.e all cards of the same
suit
def is_flush(hand):
    suits = [card[1] for card in hand]
    return len(set(suits)) == 1

# Function to check if a hand is a straight
def is_straight(hand):
    rank_values = {'2': 2, '3': 3, '4': 4, '5': 5, '6': 6, '7': 7,
'8': 8, '9': 9, '10': 10, 'J': 11, 'Q': 12, 'K': 13, 'A': 14}
    ranks = sorted([rank_values[card[0]] for card in hand])
    if ranks == list(range(ranks[0], ranks[0] + 5)):
        return True
    # Check for Ace-low straight
    if ranks == [2, 3, 4, 5, 14]:
        return True
    return False

# Simulation of drawing poker hands
def simulate_poker_hands(num_simulations):
    deck = create_deck()
    flush_count = 0
    straight_count = 0

    for _ in range(num_simulations):
        np.random.shuffle(deck)
        hand = deck[:5]
        if is_flush(hand):
            flush_count += 1
        if is_straight(hand):
            straight_count += 1

    return flush_count / num_simulations, straight_count /
num_simulations

# Number of simulations
num_simulations = 50000

# Running the simulation
flush_probability, straight_probability =
simulate_poker_hands(num_simulations)

# Plotting the results
hands = ['Flush', 'Straight']
probabilities = [flush_probability, straight_probability]
```

```
plt.figure(figsize=(18, 16))
plt.bar(hands, probabilities, color=['violet', 'yellow'])
plt.title('Probability of Poker Hands')
plt.ylabel('Probability')
plt.show()

print("Estimated Probability of Drawing a Flush:", flush_probability)
print("Estimated Probability of Drawing a Straight:",
straight_probability)
```



```
Estimated Probability of Drawing a Flush: 0.00212
Estimated Probability of Drawing a Straight: 0.00376
```

4.2.1 Bayes' Theorem:

```
# Load the dataset
waste_data =
pd.read_csv('/Users/hp/Downloads/archive/2018_2020_waste.csv')

waste_data['Is Recyclable'] = waste_data['Total Recycled'] >
waste_data['Total Generated '] * 0.5
waste_data['Generated Category'] = pd.qcut(waste_data['Total Generated
'], 3, labels=["Low", "Medium", "High"])

waste_data.info()
waste_data.columns

<class 'pandas.core.frame.DataFrame'>
RangeIndex: 45 entries, 0 to 44
Data columns (total 6 columns):
 #   Column              Non-Null Count  Dtype
---  ------              --------------  -----
 0   Waste Type          45 non-null     object
 1   Total Generated     45 non-null     int64
 2   Total Recycled      45 non-null     int64
 3   Year                45 non-null     int64
 4   Is Recyclable       45 non-null     bool
 5   Generated Category  45 non-null     category
dtypes: bool(1), category(1), int64(3), object(1)
memory usage: 1.8+ KB

Index(['Waste Type', 'Total Generated ', 'Total Recycled', 'Year',
       'Is Recyclable', 'Generated Category'],
      dtype='object')

def bayes_theorem(p_a, p_b_given_a, p_b_given_not_a):
    """
    Apply Bayes' Theorem.

    :param p_a: Probability of A (prior probability)
    :param p_b_given_a: Probability of B given A (likelihood)
    :param p_b_given_not_a: Probability of B given not A
    :return: P(A|B) - Probability of A given B (posterior probability)
    """
    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
    return p_a_given_b

# Determine if waste type is recyclable
waste_data['Is Recyclable'] = waste_data['Total Recycled'] >
waste_data['Total Generated '] * 0.5
```

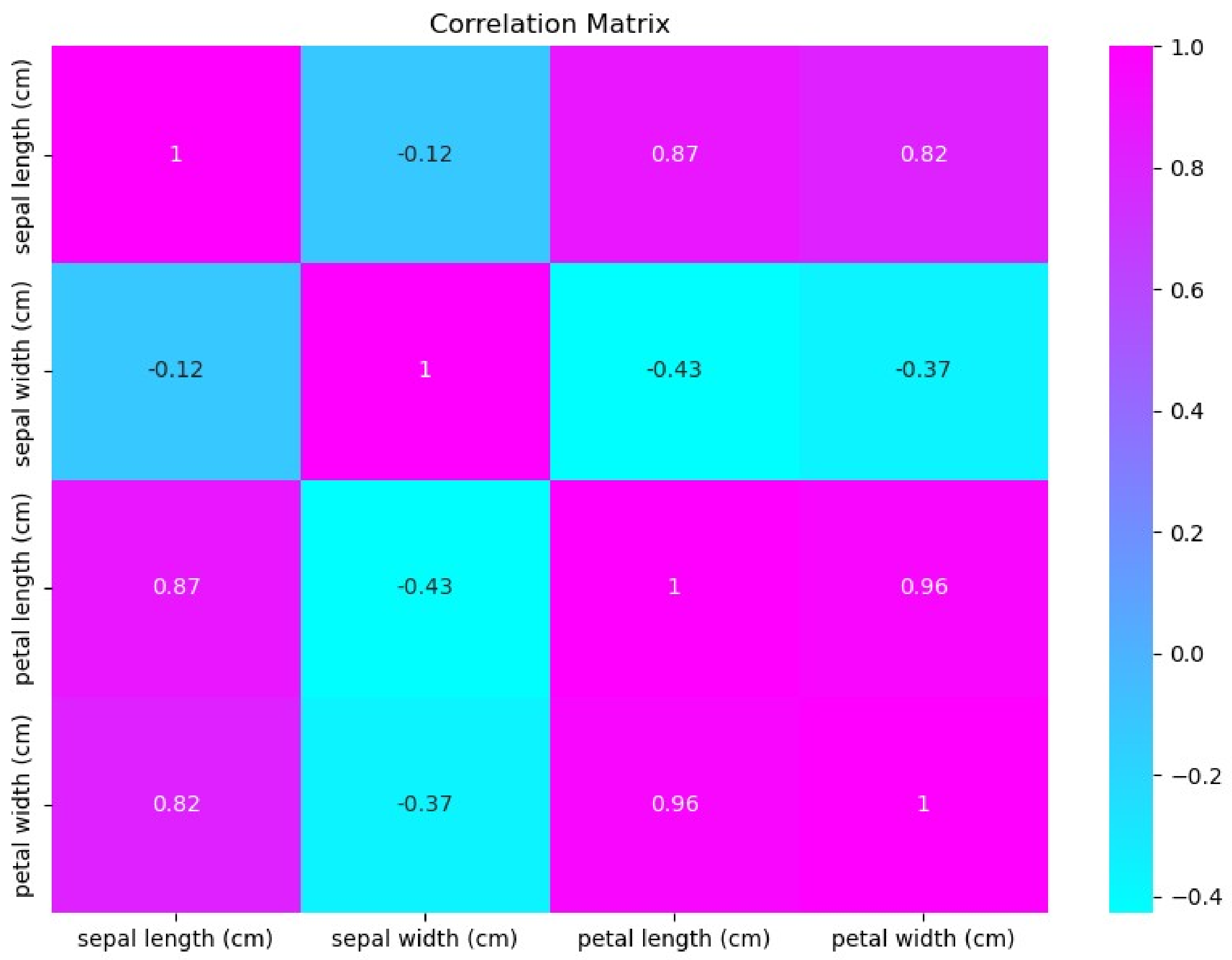
```
# Define categories for total generated waste
waste_data['Generated Category'] = pd.qcut(waste_data['Total Generated
'], 3, labels=["Low", "Medium", "High"])

# Calculate probabilities
p_a = waste_data['Is Recyclable'].mean()  # Probability of being
recyclable
p_b_given_a = waste_data[waste_data['Is Recyclable']]['Generated
Category'].value_counts(normalize=True)  # P(B|A)
p_b_given_not_a = waste_data[~waste_data['Is Recyclable']]['Generated
Category'].value_counts(normalize=True)  # P(B|not A)

# Example: Calculate the probability of being recyclable given a
"High" generated category
p_a_given_b_high = bayes_theorem(p_a, p_b_given_a['High'],
p_b_given_not_a['High'])

print("Probability of being recyclable given a high generated waste
category:", p_a_given_b_high)
```

```
Probability of being recyclable given a high generated waste category:
0.5333333333333333
```

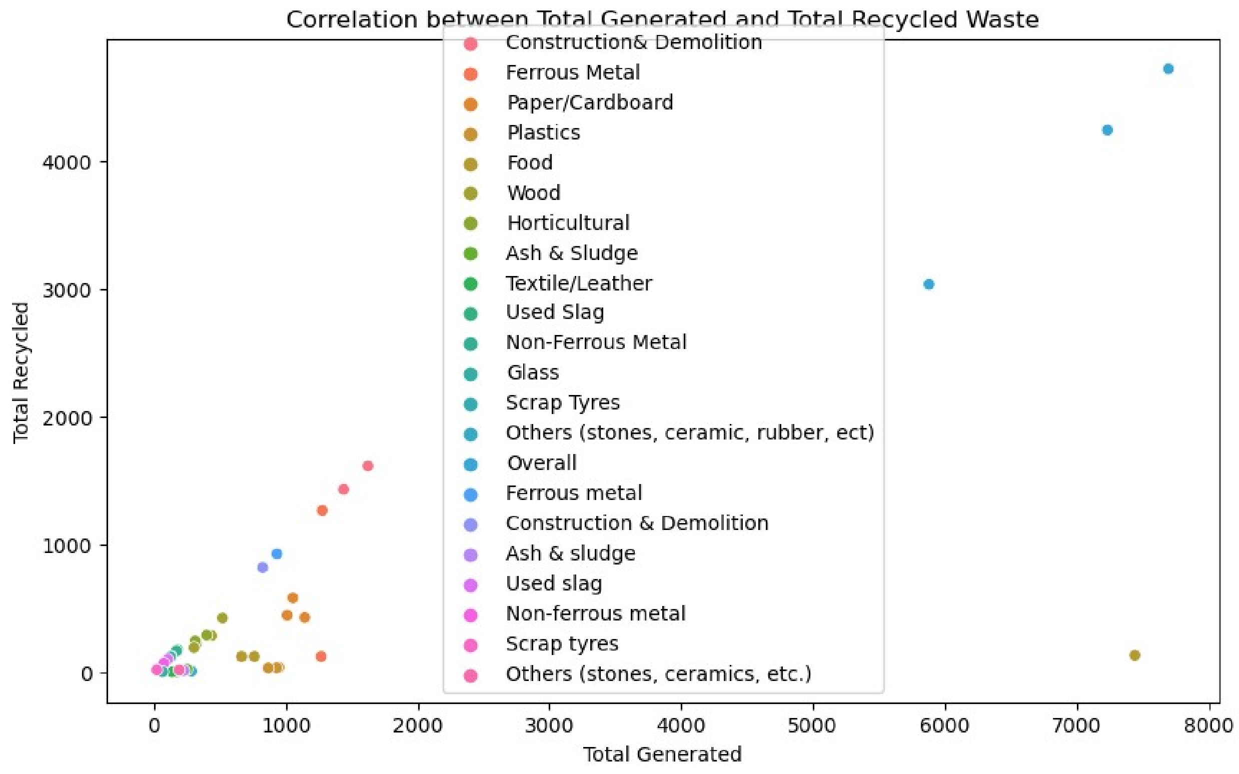
4.2.2 Joint Distribution Analysis

```
from sklearn.datasets import load_iris
from scipy.stats import shapiro, ks_2samp, norm

# I have taken the Iris datset
# Load the Iris dataset

iris = load_iris()
df = pd.DataFrame(iris.data, columns=iris.feature_names)
df['species'] = pd.Categorical.from_codes(iris.target,
iris.target_names)

# Correlation matrix visualization
plt.figure(figsize=(10, 7))
sns.heatmap(df.corr(), annot=True, cmap='cool')
plt.title('Correlation Matrix')
plt.show()
```

```
C:\Users\hp\AppData\Local\Temp\ipykernel_20924\630334875.py:3:
FutureWarning: The default value of numeric_only in DataFrame.corr is
deprecated. In a future version, it will default to False. Select only
valid columns or specify the value of numeric_only to silence this
warning.
  sns.heatmap(df.corr(), annot=True, cmap='cool')
```



```
# Shapiro-Wilk test
stat, p = shapiro(df['petal length (cm)'])
print('Shapiro-Wilk Test: Statistics=%.3f, p=%.3f' % (stat, p))

# Kolmogorov-Smirnov test
ks_stat, ks_p = ks_2samp(df['petal length (cm)'],
norm.rvs(size=len(df)))
print('Kolmogorov-Smirnov Test: Statistics=%.3f, p=%.3f' % (ks_stat,
ks_p))

Shapiro-Wilk Test: Statistics=0.876, p=0.000
Kolmogorov-Smirnov Test: Statistics=0.893, p=0.000

# Visualizing the correlation between 'Total Generated' and 'Total
Recycled'
plt.figure(figsize=(10, 6))
sns.scatterplot(x='Total Generated ', y='Total Recycled', hue='Waste
Type', data=waste_data)
plt.title('Correlation between Total Generated and Total Recycled
Waste')
plt.xlabel('Total Generated')
plt.ylabel('Total Recycled')
plt.legend()
plt.show()

# Conducting normality tests
# Shapiro-Wilk test
shapiro_generated = stats.shapiro(waste_data['Total Generated '])
shapiro_recycled = stats.shapiro(waste_data['Total Recycled'])

print("Shapiro-Wilk Test on Total Generated Waste:",
shapiro_generated)
print("Shapiro-Wilk Test on Total Recycled Waste:", shapiro_recycled)
```



```
Shapiro-Wilk Test on Total Generated Waste:
ShapiroResult(statistic=0.5176205039024353, pvalue=5.928293966839249e-
11)
Shapiro-Wilk Test on Total Recycled Waste:
ShapiroResult(statistic=0.5245072245597839, pvalue=7.243495386832777e-
11)
```

4.2.3 Factor Analysis

```
from sklearn.decomposition import PCA
from sklearn.preprocessing import StandardScaler
```

```
# Since factor analysis works better with more features, creating some
additional features for demonstration purposes

waste_data['Generated_per_Capita'] = waste_data['Total Generated '] /
1000
waste_data['Recycled_per_Capita'] = waste_data['Total Recycled'] /
1000
waste_data['Recycling_Rate'] = waste_data['Total Recycled'] /
waste_data['Total Generated ']

# Standardizing the data
scaler = StandardScaler()
waste_scaled = scaler.fit_transform(waste_data[['Total Generated ',
'Total Recycled', 'Generated_per_Capita', 'Recycled_per_Capita',
'Recycling_Rate']])

# Performing PCA for factor extraction
pca = PCA(n_components=2)  # Reduce the data to 2 components
principalComponents = pca.fit_transform(waste_scaled)

# Creating a DataFrame with the principal components
principalDf = pd.DataFrame(data = principalComponents, columns =
['principal component 1', 'principal component 2'])

# Print the variance explained by each component
print("Variance explained by each component:",
pca.explained_variance_ratio_)

# Displaying the principal components
print(principalDf.head())
```

```
Variance explained by each component: [0.72695767 0.21962308]
   principal component 1  principal component 2
0               1.460429              -1.360479
1              -0.350339               1.124436
2               0.074758              -0.143314
3              -0.608379               1.210054
4              -0.597978               0.853449
```