import numpy as np import pandas as pd

import matplotlib.pyplot as plt import scipy.stats

import seaborn as sns

# Simulating Continuous Random Variables:

Inverse Transform Sampling for Exponential Distribution

*# consider the lambda parameter for the exponential distribution*

lambda\_param = 5.0

*# Inverse transform sampling*

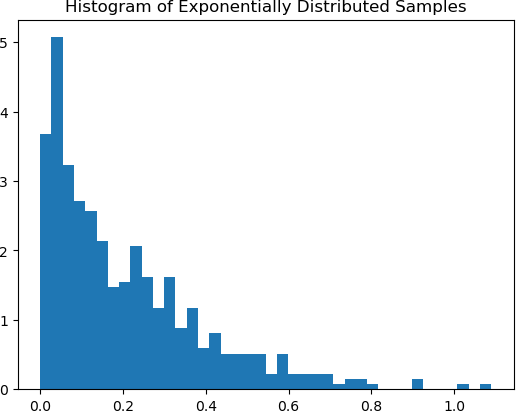
def inverse\_transform\_sampling(lambda\_param, n=500): uniform\_samples = np.random.uniform(0, 1, n)

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)

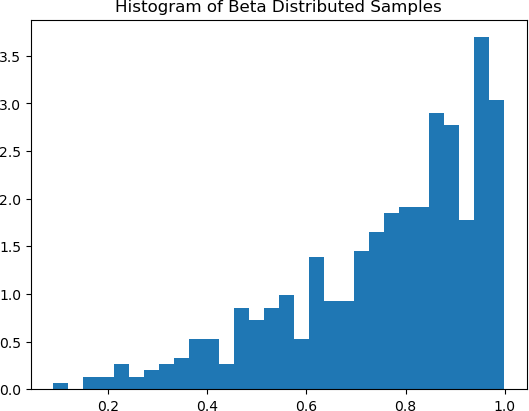
if u <= y\*\*2: *# condition for beta distribution*

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

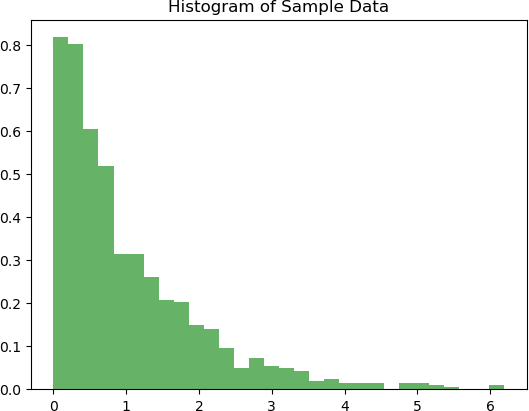
First Quantile: 0.0608708498043777

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')

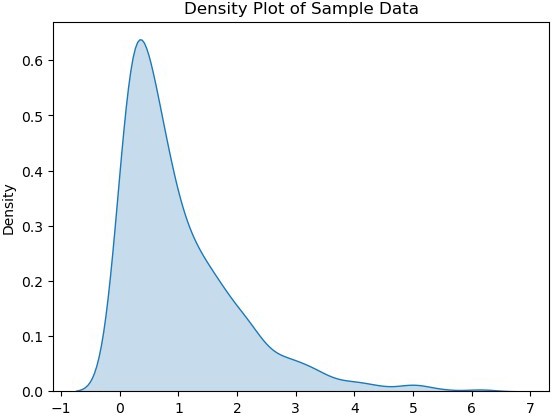
plt.show()

*# 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):

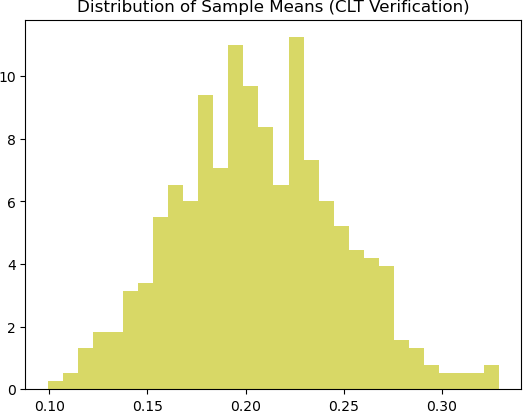
sample = np.random.choice(samples, n) sample\_means.append(np.mean(sample))

*# 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

# Simulating from Discrete Distributions

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

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

def visualize\_data(data, title): plt.figure(figsize=(15, 10)) sns.histplot(data, kde=True) plt.title(title)

plt.show()

Central Limit Theorem Verification

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)

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)

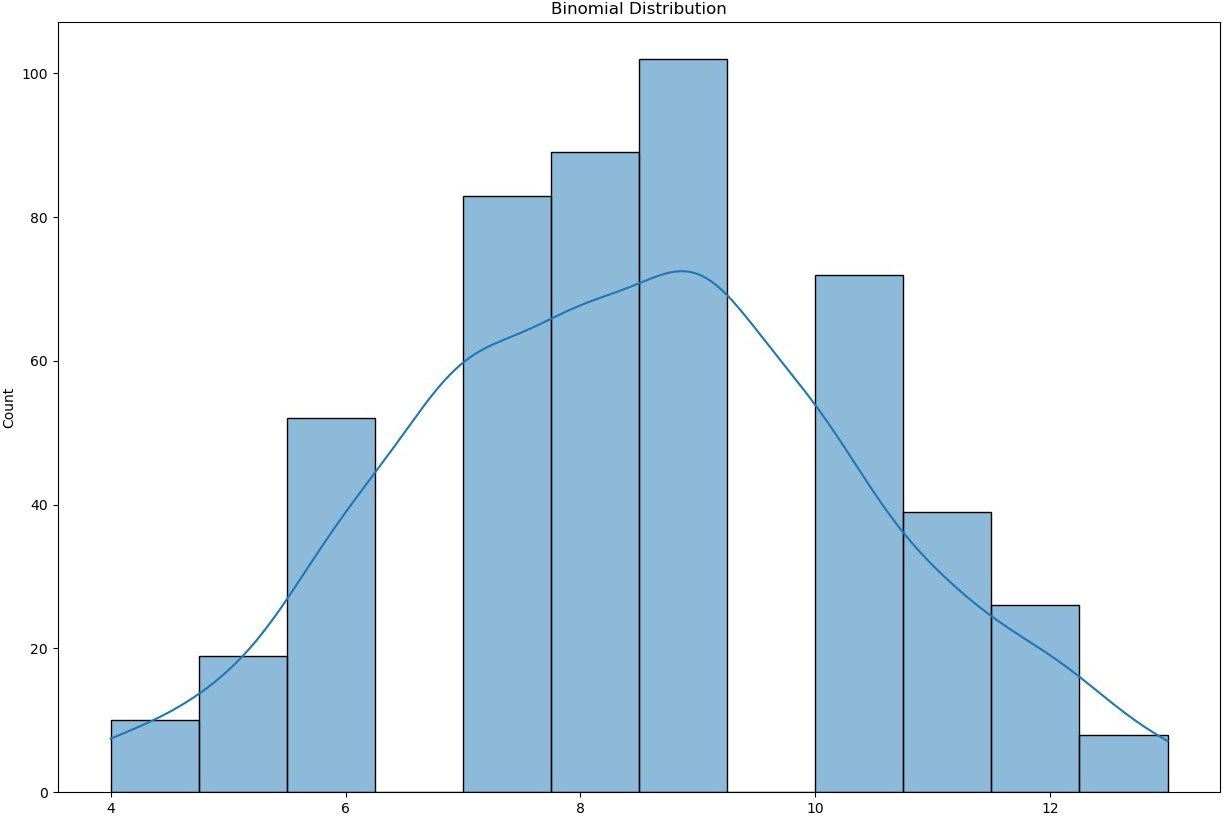
outliers = detect\_outliers(binomial\_data)

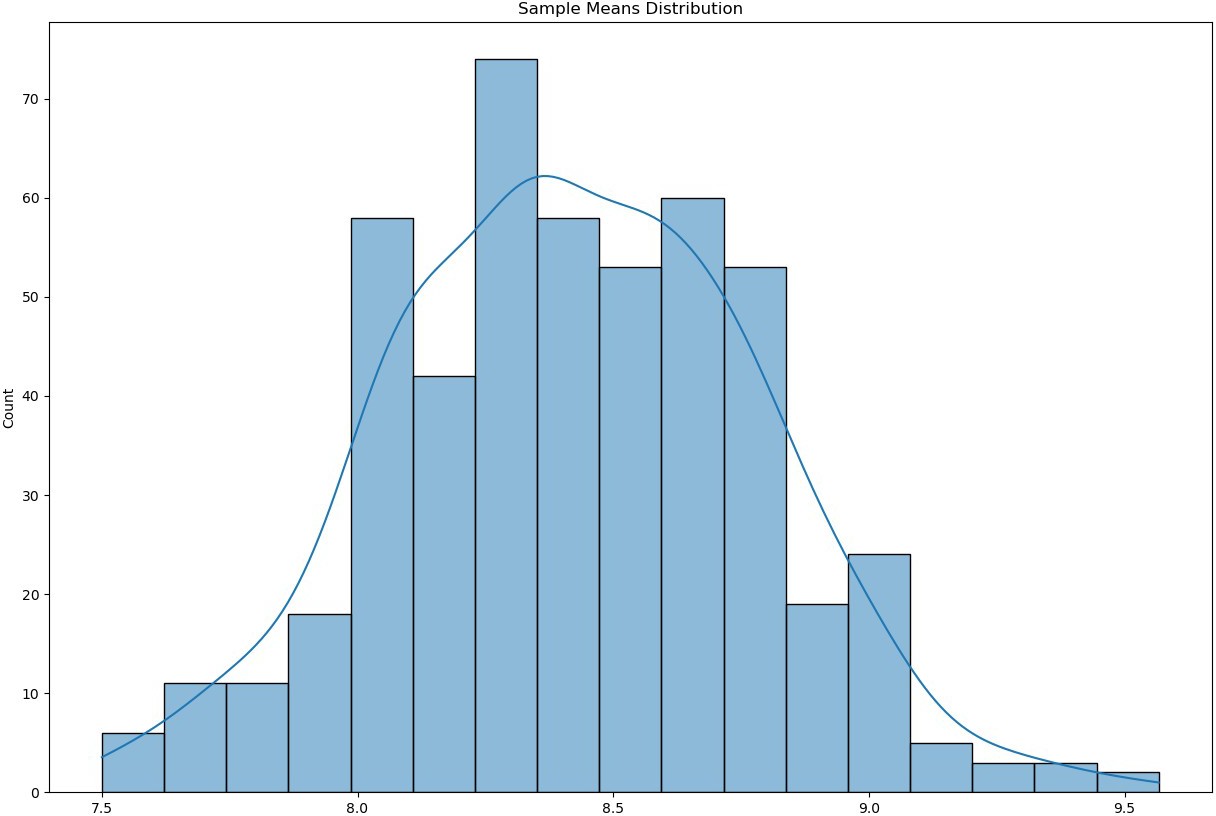
C:\Users\hp\AppData\Local\Temp\ipykernel\_20924\1632294513.py:8: FutureWarning: Unlike other reduction functions (e.g. `skew`,

`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],





# 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*

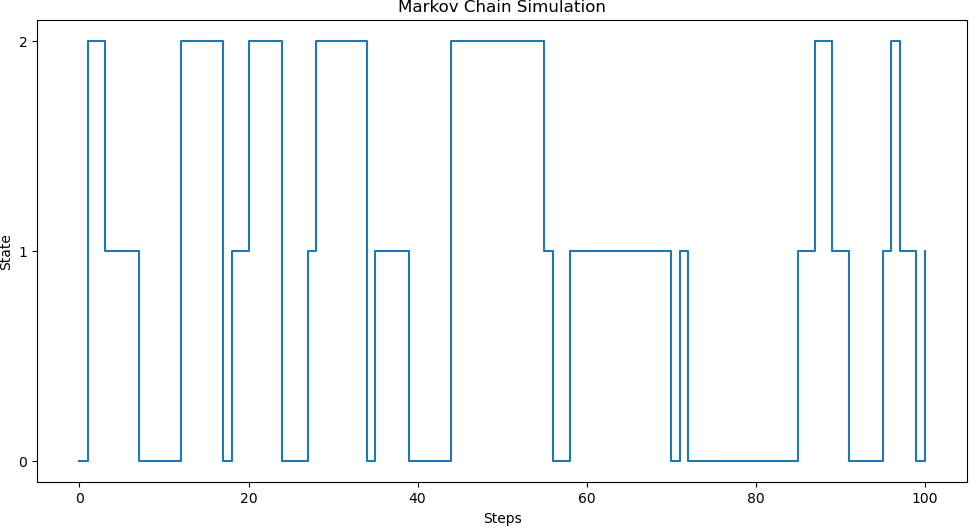
time\_averaged\_behavior = np.mean([states\_visited.count(state) / len(states\_visited) for state in range(len(transition\_matrix))])

*# 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]

# Variance Reduction Techniques:

Basic Monte Carlo Simulation to estimate the mean

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

*# 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')

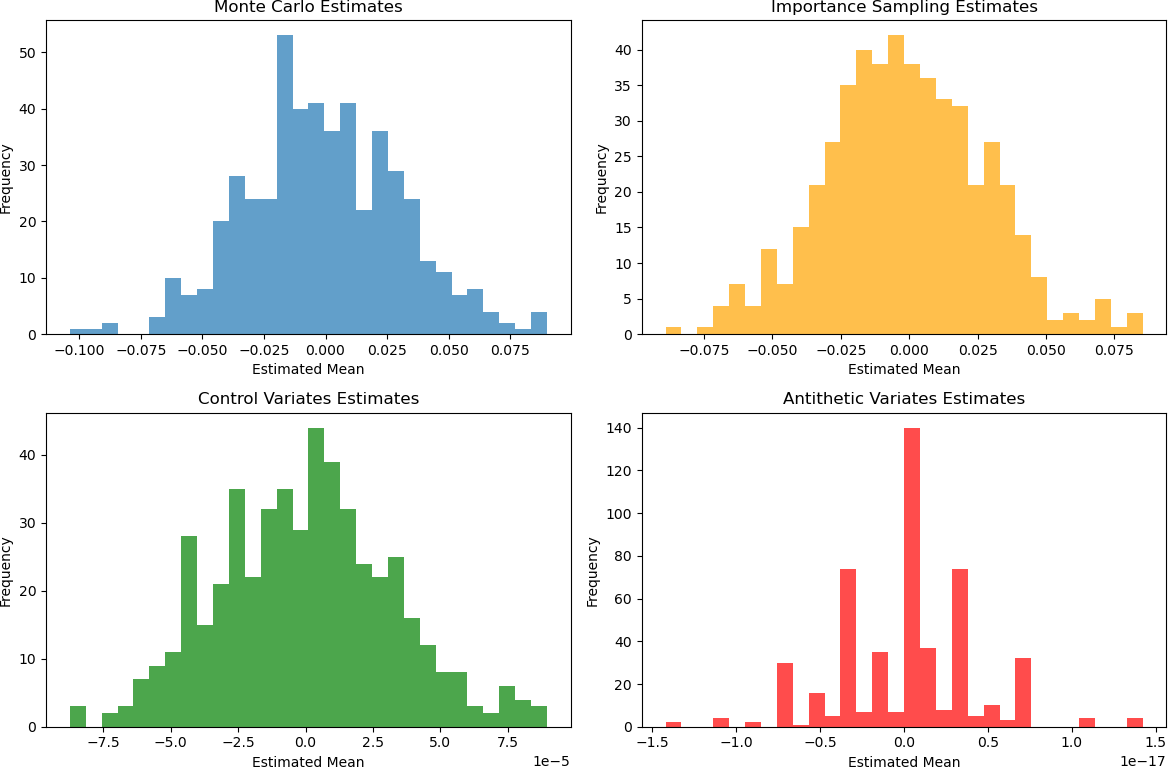
plt.title('Control Variates Estimates') plt.xlabel('Estimated Mean') plt.ylabel('Frequency')

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()



# 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]])

*# 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')

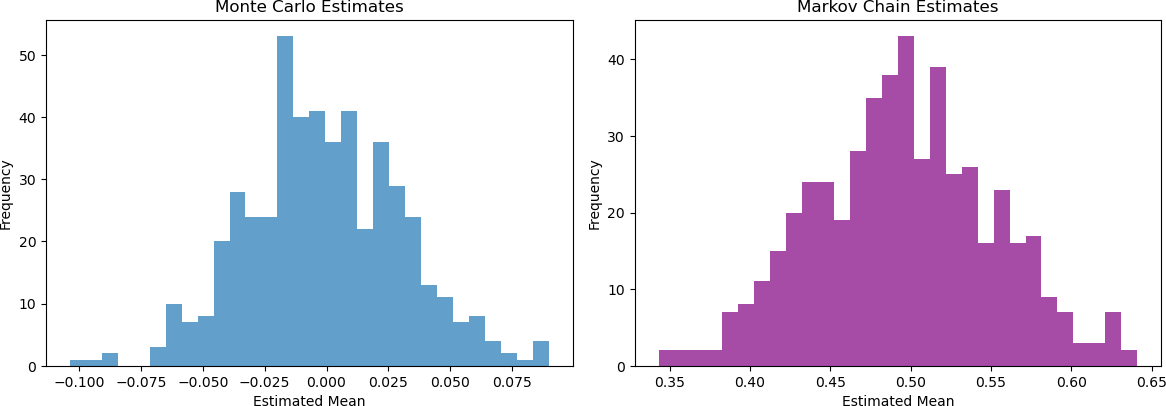
plt.xlabel('Estimated Mean') plt.ylabel('Frequency')

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()



# Simulation for Combinatorial Analysis:

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']

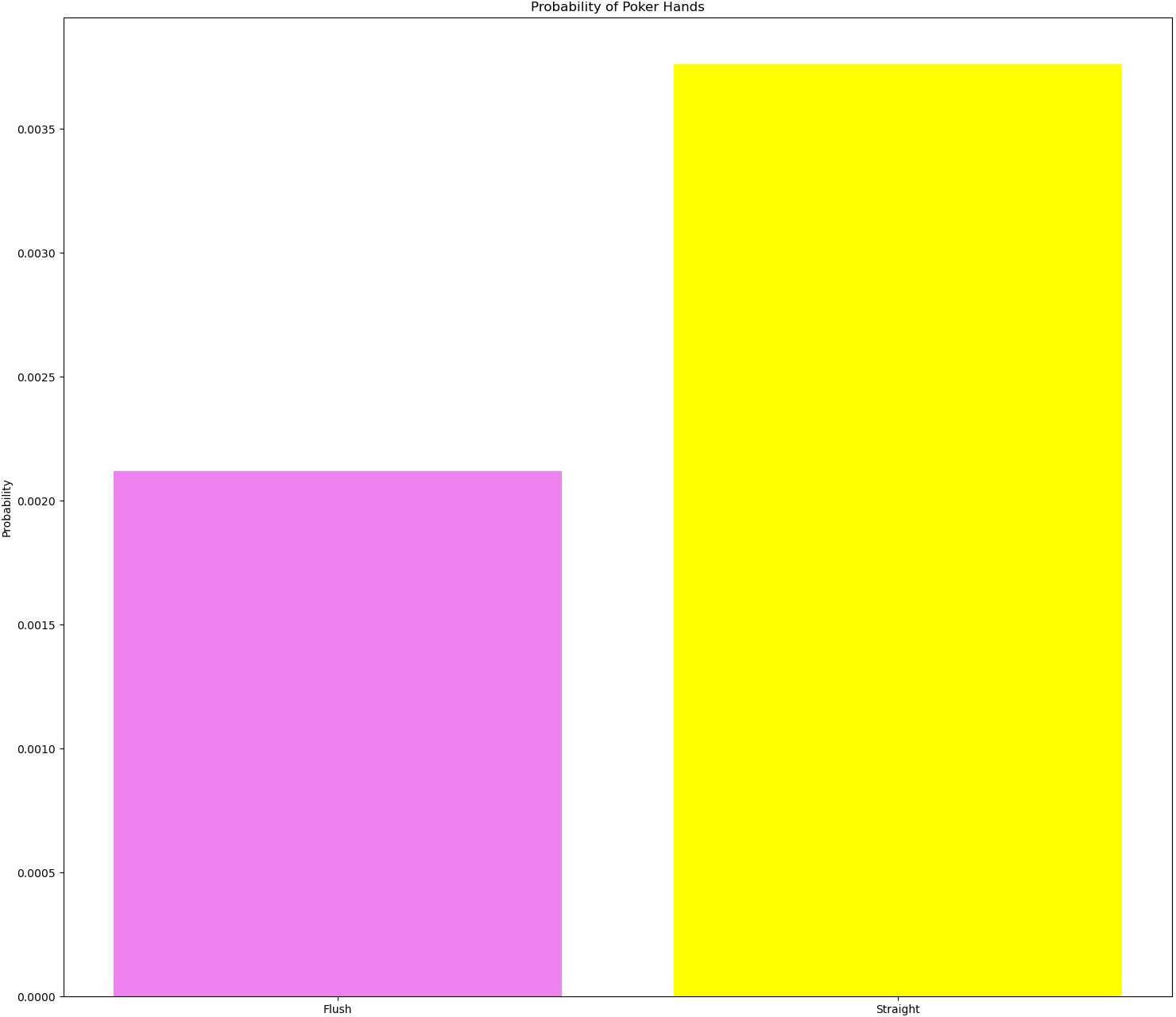
probabilities = [flush\_probability, straight\_probability]

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

# Bayes’ Theorem:

1. Total Generated
2. Total Recycled
3. Year

45 non-null

45 non-null

45 non-null

int64 int64 int64

5 Generated Category 45 non-null

category

dtypes: bool(1), category(1), int64(3), object(1)

*# 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

4 Is Recyclable 45 non-null bool

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

# 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()

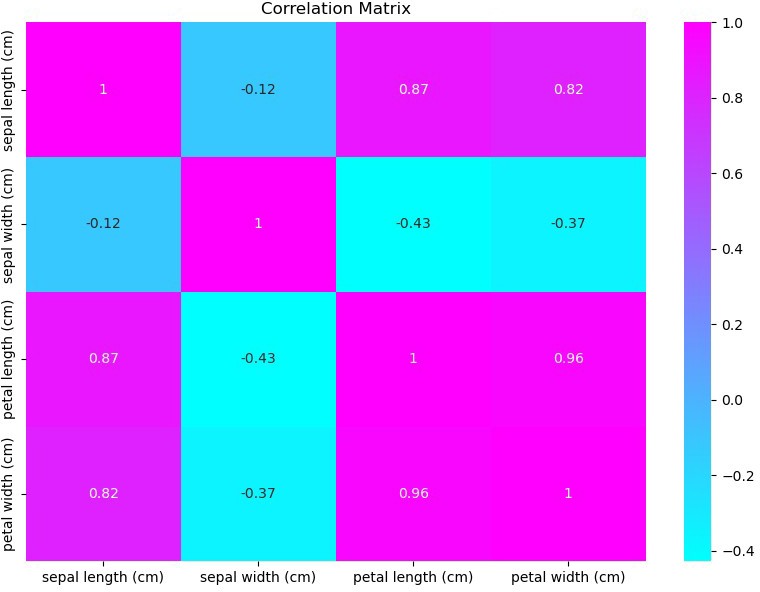
df = pd.DataFrame(iris.data, columns=iris.feature\_names) df['species'] = pd.Categorical.from\_codes(iris.target, iris.target\_names)

*# 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()

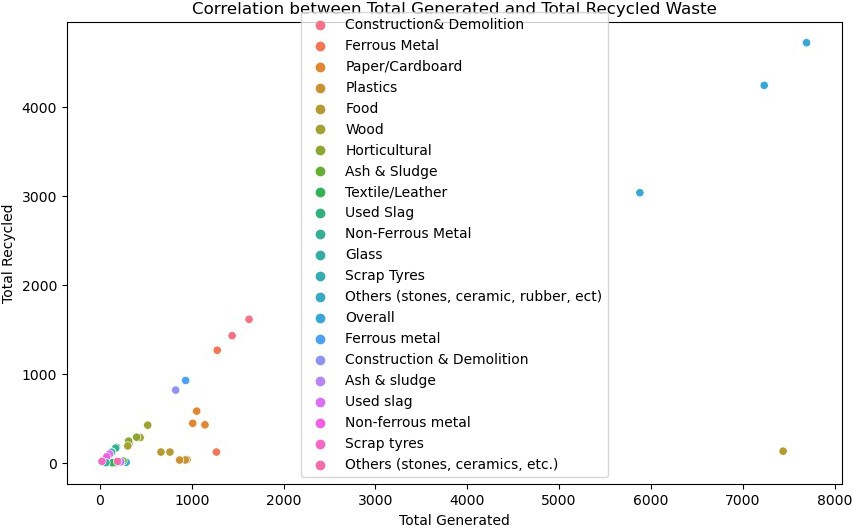
plt.show()

*# 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)

* + 1. 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]

|  |  |  |
| --- | --- | --- |
| princ  0 | ipal component 1 princ 1.460429 | ipal component 2  -1.360479 |
| 1 | -0.350339 | 1.124436 |
| 2 | 0.074758 | -0.143314 |
| 3 | -0.608379 | 1.210054 |
| 4 | -0.597978 | 0.853449 |