Probability and statistics

Project 1

Description

Mohammad Mahdi Daghighi

9926163

Numerical

```python
import numpy as np
from scipy.stats import multivariate_normal
from scipy.integrate import dblquad

# Define the mean and covariance for the Gaussian distribution
mean = [2, 3]
cov = [[1, 0], [0, 1]]  # Independent variables, diagonal covariance


# Define the density function (pdf) for X and Y
1 usage
def joint_density(x, y):
    return multivariate_normal.pdf([x, y], mean=mean, cov=cov)


# Integral limits for the circle with radius 1
1 usage
def circle_upper(x):
    return np.sqrt(1 - x ** 2)


1 usage
def circle_lower(x):
    return -np.sqrt(1 - x ** 2)


# Numerical calculation of the double integral over the circle with radius 1
numerical_integral, error = dblquad(joint_density, -1, b: 1, circle_lower, circle_upper)
print(numerical_integral, error)
```

This code snippet is performing a numerical integration to calculate the probability that a bivariate (two-variable) normal distribution falls within a unit circle centered at the origin. Here's a step-by-step explanation:

1. Importing Libraries:

- `numpy`: A fundamental package for scientific computing in Python.
- `scipy.stats`: A module in SciPy that provides tools for statistical analysis.
- `scipy.integrate`: A module in SciPy for integration techniques.

2. Defining Mean and Covariance:

- `mean = [2, 3]`: The mean of the bivariate normal distribution is set as a vector with values 2 and 3 for the two variables (X and Y, respectively).
- `cov = [[1, 0], [0, 1]]`: The covariance matrix is defined as a 2x2 matrix. Diagonal elements are 1, indicating that the variance of each variable is 1. Off-diagonal elements are 0, indicating that the variables X and Y are independent.

3. Defining the Joint Density Function:

- `joint_density(x, y)`: This is a function that computes the probability density function (pdf) of the bivariate normal distribution at a given point `(x, y)`. The `multivariate_normal.pdf` function from SciPy is used for this calculation.

4. Defining the Integration Limits:

- `circle_upper(x)` and `circle_lower(x)`: These functions define the upper and lower limits of the integration for the Y variable. Since the region of

integration is a circle with radius 1, for any x within the range -1 to 1, the corresponding y values range from `-sqrt(1 - x^2)` to `sqrt(1 - x^2)` (the equation of a circle).

5. Performing Numerical Integration:

- `dblquad(...)`: This is a function from SciPy used to compute a double integral. The first argument is the function to integrate (`joint_density` in this case). The next two arguments are the limits for the x variable (-1 to 1, since it's a unit circle). The last two arguments are the functions defining the limits of the y variable (`circle_lower` and `circle_upper`).

- The result of `dblquad` is a tuple where the first element is the value of the integral, and the second is an estimate of the absolute error in the integral's calculation.

6. Printing the Result:

- `print(numerical_integral, error)`: This prints out the value of the integral and the estimated error.

In summary, the code calculates the probability that a random point `(X, Y)` drawn from a bivariate normal distribution with a specified mean and covariance falls within a unit circle centered at the origin. This type of calculation is useful in probability and statistics for understanding how a bivariate distribution behaves over a specific region.

Sampling

```
1 usage
def estimate_probability(mean, cov, desired_accuracy, initial_sample_size=100000):
    # Function to generate samples and estimate the probability
    num_samples = initial_sample_size
    while True:
        # Generate samples from Gaussian distributions
        x_samples = np.random.normal(mean[0], cov[0][0], num_samples)
        y_samples = np.random.normal(mean[1], cov[1][1], num_samples)

        # Check if the samples are within the unit circle
        inside_circle = (x_samples ** 2 + y_samples ** 2) < 1

        # Estimate the probability
        probability_estimate = inside_circle.mean()
        # Calculate the standard error
        standard_error = inside_circle.std() / np.sqrt(num_samples)

        # Check if the standard error is within the desired accuracy
        if standard_error <= desired_accuracy * probability_estimate:
            break

        # If not, increase the sample size
        num_samples *= 2

    return probability_estimate, standard_error, num_samples
```

This code defines and uses a function to estimate the probability of a point from a bivariate Gaussian (normal) distribution falling inside a unit circle, using Monte Carlo simulation. The function iteratively increases the number of samples until the estimated probability meets a specified accuracy level. Here's a detailed explanation:

1. Function Definition `estimate_probability(mean, cov, desired_accuracy, initial_sample_size=100000)`:

- `mean`: The mean of the bivariate Gaussian distribution.

- `cov`: The covariance matrix for the distribution.

- `desired_accuracy`: The relative accuracy desired for the estimated probability.

- `initial_sample_size`: The initial number of samples to start the simulation. Default is 100,000.

2. Monte Carlo Simulation:

- Monte Carlo methods involve using random sampling to obtain numerical results, often for estimating probabilities.

3. Simulation Process:

- The function starts with an initial number of samples and enters a `while` loop, which will keep running until the desired accuracy is achieved.

- Within the loop:

  - It generates `num_samples` random samples for `x` and `y` from the Gaussian distribution with the specified mean and covariance.

  - It checks if each pair of `(x, y)` samples lies inside a unit circle (`(x_samples ** 2 + y_samples ** 2) < 1`).

  - The mean of the boolean array `inside_circle` gives the estimated probability of a point falling inside the circle.

- The standard error of the estimate is calculated as the standard deviation of `inside_circle` divided by the square root of `num_samples`.

- If the standard error is less than or equal to `desired_accuracy * probability_estimate`, it breaks the loop, indicating that the desired accuracy is achieved.

- If not, the number of samples is doubled and the process is repeated.

4. Output of the Function:

- The function returns three values:

- `probability_estimate`: The estimated probability of a point from the distribution falling within the unit circle.

- `standard_error`: The standard error of the estimate.

- `num_samples`: The number of samples used to achieve the desired accuracy.

5. Using the Function:

- The function is called with the mean, covariance, and desired accuracy.

- It prints out the estimated probability, the standard error of this estimate, and the number of samples used.

6. Purpose and Application:

- This code is an example of using Monte Carlo methods for probabilistic estimation. It's particularly useful when analytical solutions are difficult or impossible to obtain.

- The specific task here is to estimate the probability that a point drawn from a specified bivariate Gaussian distribution falls within a unit circle. This kind of simulation is common in fields like statistics, finance, and physics for making probabilistic estimates based on random sampling.