Report

Appendix (Code)

```
import csv
import numpy as np
import matplotlib.pyplot as plt
from scipy.stats import multivariate_normal
from abc import ABC, abstractmethod
from scipy.stats import beta

class ContinuousDistribution(ABC):
    @abstractmethod
    def import_data(self, filename):
        pass

    @abstractmethod
    def export_data(self, filename):
        pass

    @abstractmethod
    def compute_mean(self):
        pass

    @abstractmethod
    def compute_std_dev(self):
        pass

    @abstractmethod
    def visualize(self):
        pass

    @abstractmethod
    def generate_samples(self, n_samples):
        pass

    @abstractmethod
    def drawing_samples(self):
        pass


class GaussDistribution(ContinuousDistribution):
    def __init__(self, dim=1):
        self.dim = dim
        self.data = None
        self.generated_samples = None
        self.mean = None
        self.std_dev = None

    def import_data(self, filename):
        with open(filename, 'r') as csvfile:
            csvreader = csv.reader(csvfile)
            data = []
            for i, row in enumerate(csvreader):
                if i == 0:
                    continue  # Skip first row with column headers
                x = [float(val) for val in row]
                data.append(x)

        self.data = np.array(data)

    def export_data(self, filename):
        with open(filename, 'w', newline='') as csvfile:
            csvwriter = csv.writer(csvfile)
            csvwriter.writerow(['x', 'y', 'z'])
            for row in self.data:
                csvwriter.writerow(row)

    def compute_mean(self):
        self.mean = np.mean(self.data, axis=0)

    def compute_std_dev(self):
        self.std_dev = np.std(self.data, axis=0)

    def visualize(self):
        self.compute_mean()
        self.compute_std_dev()

        if self.dim == 1:
            # Define parameters
            mu = self.mean  # mean values
            sigma = self.std_dev  # standard deviations

            # Create grid for x
            x = np.linspace(min(self.data), max(self.data), len(self.data))

            # Compute Gaussian function
            y = 1 / np.sqrt(2 * np.pi * sigma**2) * np.exp(-(x - mu)**2 / (2 * sigma**2))

            # Plot the Gaussian distribution
            plt.plot(x, y)
            plt.xlabel('X')
            plt.ylabel('Y')
            plt.title('1D Gaussian Distribution')
            plt.show()

        elif self.dim == 2:
            # Define parameters
            mu_x, mu_y = self.mean[0], self.mean[1]  # mean values
            sigma_x, sigma_y = self.std_dev[0], self.std_dev[1]  # standard deviations

            # Create grid for x and y
            x = np.linspace(min(self.data.T[0]), max(self.data.T[0]), len(self.data.T[0]))
            y = np.linspace(min(self.data.T[1]), max(self.data.T[1]), len(self.data.T[1]))
            X, Y = np.meshgrid(x, y)

            # Compute Gaussian function
            G = np.exp(-((X - mu_x) ** 2 / (2 * sigma_x ** 2) + (Y - mu_y) ** 2 / (2 * sigma_y ** 2))) / (2 * np.pi * sigma_x * sigma_

            # Plot the Gaussian distribution
            fig, ax = plt.subplots()
            ax.contourf(X, Y, G)
            ax.set_xlabel('X')
            ax.set_ylabel('Y')
            ax.set_title('2D Gaussian Function')
            plt.show()

        elif self.dim == 3:
            # Define parameters
            mu = np.array(self.mean)  # mean vector
            sigma = np.cov(self.data.T)  # covariance matrix

            # Create grid for x, y, z
            x = np.linspace(min(self.data.T[0]), max(self.data.T[0]), len(self.data.T[0]))
            y = np.linspace(min(self.data.T[1]), max(self.data.T[1]), len(self.data.T[1]))
            X, Y = np.meshgrid(x, y)

            # Compute Gaussian function
            rv = multivariate_normal(mean=mu[:2], cov=sigma[:2, :2])
            pos = np.empty(X.shape + (2,))
            pos[:, :, 0] = X
            pos[:, :, 1] = Y
            Z = rv.pdf(pos)

            # Visualize the Gaussian function
            fig = plt.figure()
            ax = fig.add_subplot(111, projection='3d')
            ax.plot_surface(X, Y, Z, cmap='viridis', alpha=0.7)
            ax.set_xlabel('X')
            ax.set_ylabel('Y')
            ax.set_zlabel('Z')
            ax.set_title('3D Gaussian Function')
            plt.show()

    def generate_samples(self, n_samples=100):
        self.compute_mean()
        self.compute_std_dev()
        self.generated_samples = np.random.normal(loc=self.mean, scale=self.std_dev, size=(n_samples, self.mean.shape[0]))

    def find_empirical_params(self):
        self.import_data('MGD.csv')
        self.compute_mean()
        self.compute_std_dev()

    def drawing_samples(self):
        fig = plt.figure(figsize=(10, 5))
        ax1 = fig.add_subplot(121, projection='3d')
        ax2 = fig.add_subplot(122, projection='3d')

        self.find_empirical_params()
        self.generate_samples()

        # Plotting the samples from MGD.csv file
        mu = np.array(self.mean)
        sigma = np.cov(self.data.T)
        x = np.linspace(min(self.data.T[0]), max(self.data.T[0]), len(self.data.T[0]))
        y = np.linspace(min(self.data.T[1]), max(self.data.T[1]), len(self.data.T[1]))
        X, Y = np.meshgrid(x, y)
        rv = multivariate_normal(mean=mu[:2], cov=sigma[:2, :2])
        pos = np.empty(X.shape + (2,))
        pos[:, :, 0] = X
        pos[:, :, 1] = Y
        Z = rv.pdf(pos)
        ax1.plot_surface(X, Y, Z, cmap='viridis', alpha=0.7)
        ax1.set_xlabel('X')
        ax1.set_ylabel('Y')
        ax1.set_zlabel('Z')
        ax1.set_title('3D Gaussian Function of MGD.csv')

        # Plotting the samples generated from the learned distribution
        mu = np.array(self.mean)
        sigma = np.cov(self.generated_samples.T)
        x = np.linspace(min(self.generated_samples.T[0]), max(self.generated_samples.T[0]), len(self.generated_samples.T[0]))
        y = np.linspace(min(self.generated_samples.T[1]), max(self.generated_samples.T[1]), len(self.generated_samples.T[1]))
        X, Y = np.meshgrid(x, y)
        rv = multivariate_normal(mean=mu[:2], cov=sigma[:2, :2])
        pos = np.empty(X.shape + (2,))
        pos[:, :, 0] = X
        pos[:, :, 1] = Y
        Z = rv.pdf(pos)
        ax2.plot_surface(X, Y, Z, cmap='viridis', alpha=0.7)
        ax2.set_xlabel('X')
        ax2.set_ylabel('Y')
        ax2.set_zlabel('Z')
        ax2.set_title('3D Gaussian Function of sample from the learned distribution')

        plt.show()

class BetaDistribution(ContinuousDistribution):
    def __init__(self, a, b):
        self.a = a
        self.b = b
        self.data = None
        self.generated_samples = None
        self.mean = None
        self.std_dev = None

    def import_data(self, filename):
        with open(filename, 'r') as csvfile:
            csvreader = csv.reader(csvfile)
            data = []
            for i, row in enumerate(csvreader):
                if i == 0:
                    continue  # Skip first row with column headers
                x = [float(val) for val in row]
                data.append(x)

        self.data = np.array(data)

    def export_data(self, filename):
        with open(filename, 'w', newline='') as csvfile:
            csvwriter = csv.writer(csvfile)
            csvwriter.writerow(['x', 'y', 'z'])
            for row in self.data:
                csvwriter.writerow(row)

    def compute_mean(self):
        self.mean = beta.mean(self.a, self.b)

    def compute_std_dev(self):
        self.std_dev = beta.std(self.a, self.b)

    def visualize(self):
        self.compute_mean()
        self.compute_std_dev()
        x = np.linspace(0, 1, 100)
        y = beta.pdf(x, self.a, self.b) # y = (x ** (self.a - 1)) * ((1 - x) ** (self.b - 1)) / (beta(self.a, self.b))
        plt.plot(x, y)
        plt.axvline(x=self.mean, color='red', linestyle='--', label='Mean')
        plt.axvline(x=self.mean - self.std_dev, color='green', linestyle='--', label='Standard Deviation')
        plt.axvline(x=self.mean + self.std_dev, color='green', linestyle='--')
        plt.title('Visualization of Beta distributions')
        plt.legend()

    def generate_samples(self, n_samples=1000000):
        self.generated_samples = beta.rvs(self.a, self.b, size=n_samples)

    def drawing_samples(self):
        plt.hist(self.generated_samples, bins='auto', alpha=0.7)
        plt.xlabel('X')
        plt.ylabel('Frequency')
        plt.title(f'Samples from Beta Distribution (a={self.a}, b={self.b})')
        plt.show()
```

Introduction
The code implements the abstract base class ContinuousDistribution, and the concrete subclass GaussDistribution and BetaDistribution. GaussDistribution is a continuous probability distribution function that creates Gaussian distribution models with one, two, or three dimensions, visualizes them, generates samples, and computes mean and standard deviation from given data. The beta distribution is a continuous probability distribution with values between 0 and 1, often used to model probabilities and proportions. It is characterized by two parameters, denoted by α and β, which represent the shape of the distribution. Likewise, it visualizes distribution, generates samples, and computes mean and standard deviation.

Mathematical Definitions
Gaussian distribution, also known as the normal distribution, is a continuous probability distribution that is often used to describe real-world phenomena, such as measurements of physical quantities, errors in scientific experiments, and natural variations in data. It is characterized by a bell-shaped curve, which is symmetrical around the mean (average) value, and has a standard deviation that determines the spread of the curve.
In one dimension, the Gaussian function is defined as:

$$f(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2}$$

μ means mean and σ means standard deviation.

In two dimension, the Gaussian function is defined as:

$$G(x,y) = \frac{1}{2\pi\sigma_x\sigma_y} e^{-\frac{(x-\mu_x)^2}{2\sigma_x^2} - \frac{(y-\mu_y)^2}{2\sigma_y^2}}$$

Here, x and y are variables, $(x\mu, yx)$ are mean values, and σy and $\sigma\mu$ are standard deviations of x and y, respectively.

In three dimension, the Gaussian function is defined as:

$$G(x,y,z) = \frac{1}{(2\pi)^{3/2}|\mathbf{\Sigma}|^{1/2}} e^{-\frac{1}{2}(\mathbf{x}-\mu)^{\intercal}\mathbf{\Sigma}^{-1}(\mathbf{x}-\mu)}$$

Here, x, y, and z are variables, μ is the mean vector, and σ is the covariance matrix.

Beta distribution is a continuous probability distribution that is used to model the behavior of random variables that are bounded between 0 and 1, such as proportions, probabilities, and percentages. It has two parameters, *α* and *β*, which determine the shape of the distribution.

$$f(x;\alpha,\beta) = \frac{x^{\alpha-1}(1-x)^{\beta-1}}{B(\alpha,\beta)} \qquad \text{for } 0 \le x \le 1$$

where B(*α*,*β*) is the beta function, which is used as a normalizing constant to ensure that the integral of the PDF over the entire domain is equal to 1.

$$B(\alpha,\beta) = \int_0^1 t^{\alpha-1}(1-t)^{\beta-1}\,\mathrm{d}t$$

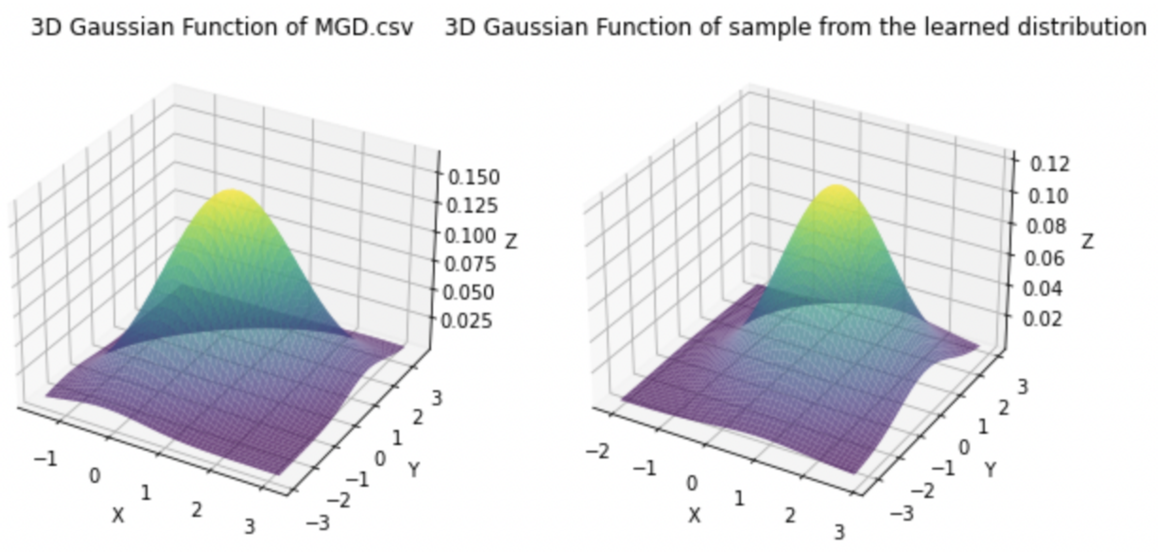
Both Gaussian distribution and beta distribution are widely used in various fields of study, such as statistics, physics, engineering, finance, and biology. They have different properties and applications, but both are useful tools for modeling and analyzing random variables.

Code of GaussDistribution

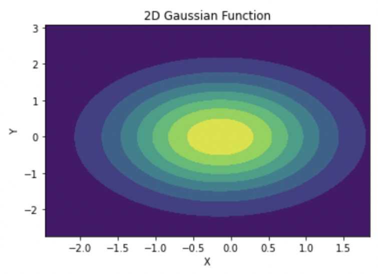
In GaussDistribution class, ‘find_empirical_params’ function is function that import data from ‘MGD.csv’ and compute mean and stadard deviation.

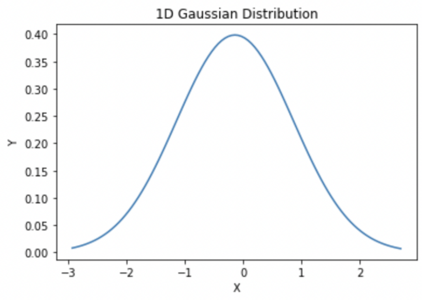
In ‘generate_samples’ function, it generate a set of random numbers from the probability distribution. In other words, it means sampling from the distribution to obtain a set of random data points that follow the same underlying distribution as the original data. These samples can be used to analyze the properties of the distribution or to estimate unknown parameters of the distribution. The number of samples generated may vary depending on the complexity of the problem to be analyzed, but the code used 100 as the default. The reason is that the number of data in the MGD.csv file is 100 so that it is easy to compare. Typically, at least 30 samples can be taken to apply a centralized theorem, and the more complex the distribution looks or the thicker the tail, the more samples may be required. In addition, because the number of samples affects the accuracy of the parameters you want to estimate, calculating the number of samples required for parameter estimation requires an understanding of the distribution and how the parameters are estimated. The code uses the NumPy function np.random.normal to generate random numbers from a normal distribution. This function generates samples with a Gaussian distribution, with the specified mean and standard deviation.

In ‘drawing_sampels’ function, the code both plot the samples from MGD.csv file and samples generated from the learned distribution. So you can compare two subfigures. The result is below.



Below is result of one-demension Gaussian Distribution using code when the mean is 0 and standard deviation is 1 and result of two-demension Gaussian Distribution when the mean is [0, 0] and standard deviation is [1, 1].



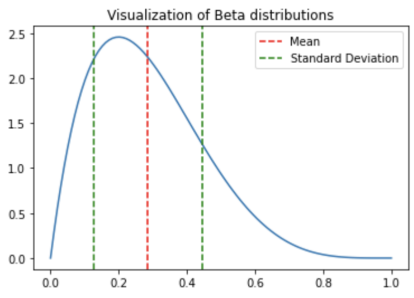


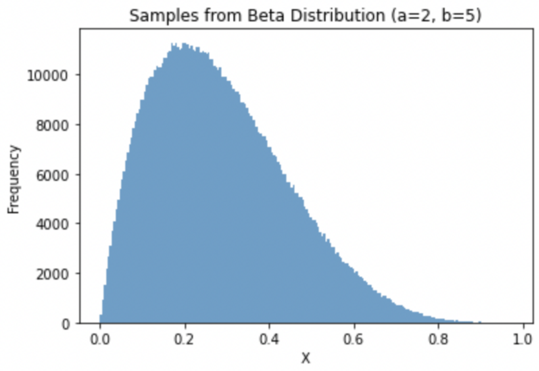
Code of BetaDistribution

In ‘compute_mean’ and ‘compute_std_dev’ function, I use method of the ‘beta’ module from scipy.stats.

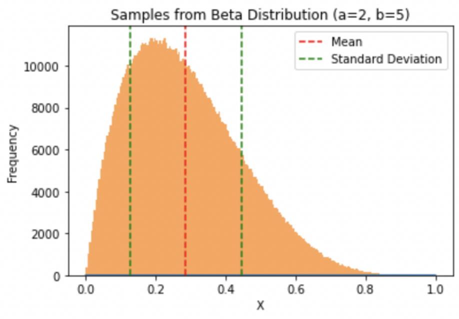
The ‘visualize’ function generates a plot of the beta distribution, including the probability density function, mean, and standard deviation. It uses the pdf method of the beta distribution from scipy.stats to compute the probability density function.

The ‘generate_samples’ function generates n_samples samples from the beta distribution using the rvs method of the beta distribution from scipy.stats. It stores the generated samples in self.generated_samples. As the number of samples in a beta distribution increases, the difference between the theoretical distribution and the distribution estimated using the sample decreases, so estimation is usually performed using a large number of samples. More than 1000 samples are usually recommended. The code used 1000000 pieces to increase accuracy. So when you run ‘drawing_samples’ function, you can see that there is almost no difference between the visualization of beta distribution and the drawing of sampling. Below is the comparison result.





This is result that print both together.



Citation

One publication discussing how to sample from these distributions is the article "A simple method for generating normal random variables," by Marsaglia and Bray, published in the Journal of the American Statistical Association in 1964. They proposed a method for generating normal random variables using the Box-Muller transform, which involves transforming two uniformly distributed random variables into two normally distributed random variables.

Conclusion
The code provides a useful tool for creating, visualizing, and generating samples from Gaussian or normal distribution models. It can be used in various applications such as statistical analysis, machine learning, and data science. The code also highlights the importance of defining the distributions in mathematical terms and understanding the sampling techniques used to generate the data.