Assignment III: Gauss and Beta Distribution

Exercises in Machine Learning (190.013), SS2022
Stefan Nehl[1]

[1]*stefan-christopher.nehl@stud.unileoben.ac.at, MNr: 00935188*, Montanuniversität Leoben, Austria

March 20, 2022

In the third assignment, I had to create the abstract class ContinuousDistribution. This class contains functions for importing and exporting the data to CSV files, compute the mean and standard deviation, generating samples and visualize the given data. Furthermore, I had to implement two additional classes, GaussDistribution and BetaDistribution which inheritances the class ContinuousDistribution and implement gauss and beta distribution. Finally, I plotted the result of this distributions.

1 Introduction

The gauss and beta distributions are often used in machine learning to generate random variables. The target of this assignment was to create a module *inference* which implements this two different distributions.

2 Implementation

For the implementation I created a module with the name *inference*. In this module i implemented the abstract class *ContinuousDistribution* with it's abstract methods.

3 Abstract class *ContinuousDistribution*

First the class was *ContinuousDistribution* created. This class is not completely abstract because some methods I reused in the other classes. For this I implemented first a constructor with the default properties. These properties are *dataSet, normalizedDataSet, mean, median, variance* and *standardDeviation*. This properties are reused in the *GaussDistribution* and *BetaDistribution*. The functions, *importCSV, exportCSV, calculateMean, calculateVariance, calculateStandardDeviation* and *normalizeDataSet*. Only the method *generateSamples* and *plotData* is abstract, because every class needs there own implementation of this functions.

4 Gauss Distribution

The class *GaussDistribution* has a constructor with the parameters *dimension*, sets the dimension of this gauss distribution and the optional parameters *fileName*, for importing a *CSV* file, *numberOfSamplesToGenerate*, number of samples generated by the class, *mean* and *variance*. Important to mention here is, that the importing of a file and the generating of samples is excluding each other. Only one parameter can be set otherwise the class throws an exception. The construction also sets the data and calculate the needed values like mean and standard deviation and generates the gauss distribution.

4.1 Generating Samples

As already mentioned, the generating of samples needs to be implemented for each class separately. For the generation I used the *random()* function from the *numpy* library with the values of the mean and the variance for the generation and the dimension with the number of samples for the amount of data.

```

        self.
            calculateStandardDeviation
            ()
        self.gaussenDistribution = []
        self.generateGaussen()

    def generateSampels(self):
        if len(self.dataSet) != 0:
```

4.2 Calculation

For the calculation I implemented two different methods. One for the one dimensional calculation, *generate-*

Gaussen1D, and for the two dimensional calculation, *calculateGaussen2D*. For the one dimensional implementation I used the following formula.

$$N(x|\mu, \sigma) = \frac{1}{(2\pi\sigma^2)^{1/2}}\, exp^{-\frac{1}{2\sigma^2}(x-\mu)^2}$$

Where μ stands for the mean and σ for the standard deviation. For the two dimensional implementation I used the following formula.

$$N(x|\mu, \sigma) = \frac{1}{(2\pi)^{D/2}|\Sigma|^{1/2}}\, exp^{-\frac{1}{2}(x-\mu)^{T}\Sigma^{-1}(x-\mu)}$$

Where D is the dimension, Σ the covariance and $|\Sigma|$ the determinant of the covariance.For the covariance I used created a vector with the mean and zeros. $\begin{pmatrix} \sigma & 0 \\ 0 & \sigma \end{pmatrix}$ Both formulas are from the book An Introduction to Probabilistic Machine Learning. (Rueckert, 2022)

4.3 Plotting

The implementation for the one dimensional plot was straightforward. I plotted a histogram of the generated data and the gauss distribution as a line above the histogram. Additional, I add the raw the of the generated samples. With the two dimensional plots I had some issues. I was able to create a 3d model of the raw data and the 3d bar chart of the distribution. I wanted to plot the surface of the two dimensional gauss distribution following the paper (Roelants, 2018), but unfortunately I failed to create the surface.

5 Beta Distribution

The class *BetaDistribution* has analogue to the class *GaussDistribution* also a constructor for handling the initialization for the parameters. Also the limitation for file name for *CSV* or generation of samples is the same. The difference is, that the beta distribution needs the parameter *a* and *b* and not the dimension for the distribution.

5.1 Generating Samples

The generation for the samples was created again with the *random()* function from the *numpy* library. However, I changed the distribution to the beta distribution.

```
        self.calculateMean()
        self.calculateVariance()
        self.calculateStandardDeviation()
        self.generateBetaDistribution()

    def generateSampels(self):
        if len(self.dataSet) != 0:
```

5.2 Calculation

For the calculation I used the following formula.

$$Beta(x|a, b) = B(a, b)x^{a-1}(1-x)^{b-1}$$

where a,b are in the constructor given parameters as a scalar and *B(a,b)* the Beta function.

$$B(a, b) = \frac{\Gamma(a+b)}{\Gamma(a)\Gamma(b)}$$

where Γ is the gamma function. The formulas are again from book An Introduction to Probabilistic Machine Learning. (Rueckert, 2022)

5.3 Plotting

For plotting the results I used a histogram with the distribution and the raw data. In addition I created a plot with the beta distribution with different values for the parameters *alpha* and *beta*. The values are displayed in Table 1. For the plotting of the parameters with different values I added a method named *plotDataWithDifferentAlphasAndBetas* with a list of the different settings as a parameter. These function sets the values and generates the plots.

Table 1: *Values for alpha and beta*

alpha	beta
0.5	0.5
5	2
2	5

6 Results

Figure 1 shows the gauss distribution for one dimension. It displays the values distribution and the frequencies of those values. The orange line displays the gauss distribution itself. Figure 2 shows the gauss distribution for two dimensions in a three dimensional plot. The beta distribution is displayed in figure 3. This plot uses an alpha with 0.5 and a beta with 0.5. The result displays a higher frequency of the values around 0 and 1. This is also displayed in the scatter chart with the raw values of this distribution. Figure 4 displays the beta distribution with alpha 2 and beta 5. The frequency is now higher at the start of the plot with a maximum at around 0.2. The beta distribution in figure 5 displays the frequency with the value for alpha 5 and beta 2. Therefore, the frequency is higher at the end of the plot near 0.8. The last figure, Figure 5, displays the three settings for the beta distribution.

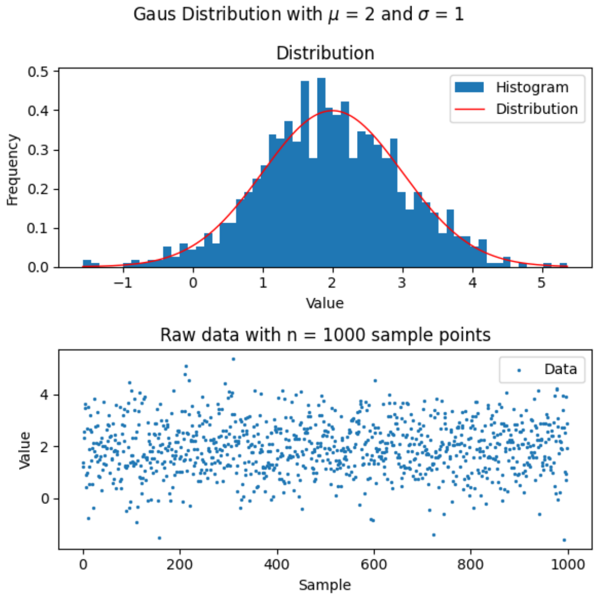

Figure 1: *Gauss Distribution for one dimension with the raw data*

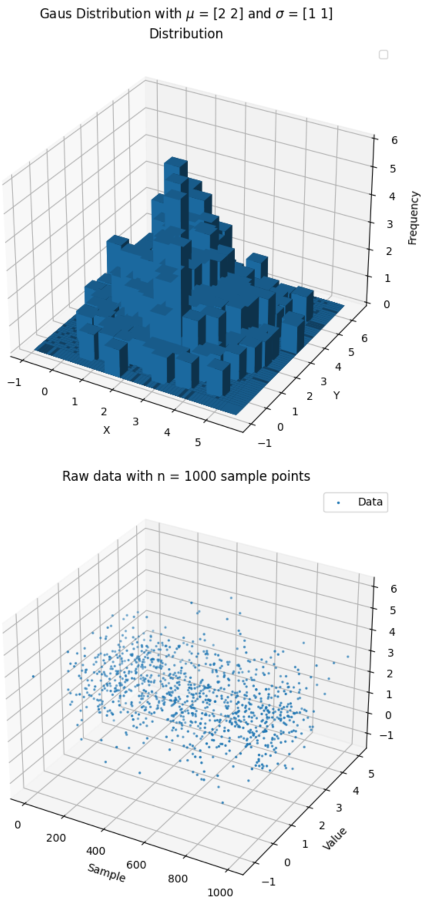

Figure 2: *Gauss Distribution for two dimensions with the raw data*

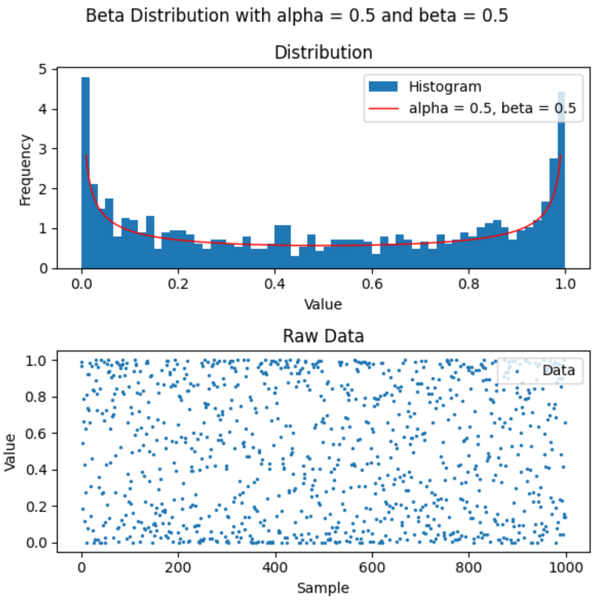

Figure 3: *Beta Distribution with alpha = 0.5 and beta = 0.5*

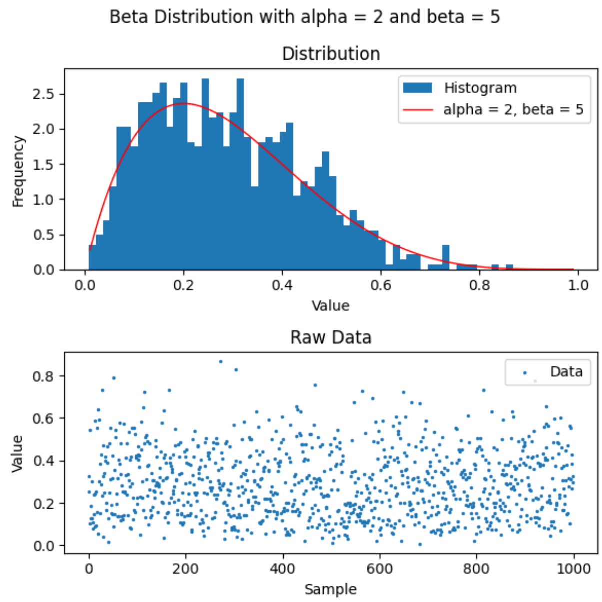

Figure 4: *Beta Distribution with alpha = 2 and beta = 5*

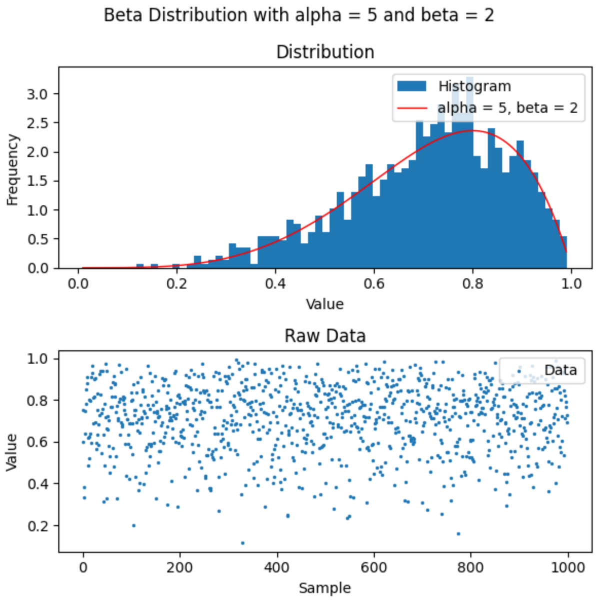

Figure 5: *Beta Distribution with alpha = 5 and beta = 2*

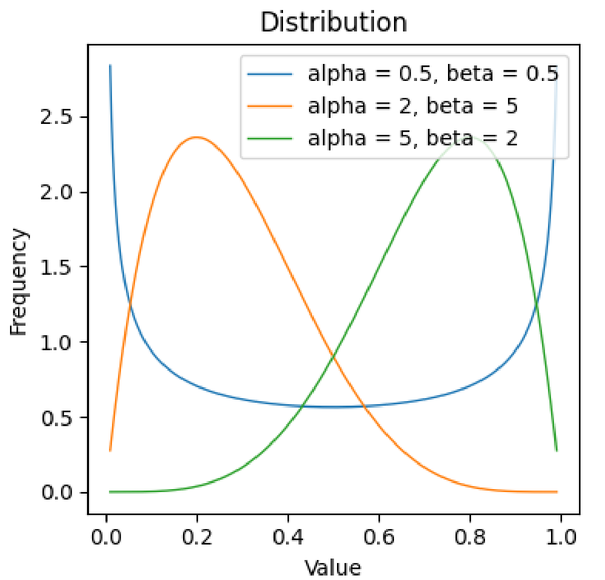

Figure 6: *Beta Distribution with the the different values for alpha and beta*

7 Conclusion

The implementation of the abstract class was straightforward. For the other classes I made some changes. I moved the classes to separate files to handle them better. Unfortunately, I was not able to create a satisfying plot for the gauss distribution for two dimensions. The whole code is in the appendix of this paper.

APPENDIX

```python
"""
@author: Nehl Stefan
"""

import sys
sys.path.insert(0, '../modules')
import inference
from BetaDistribution import BetaDistribution
from GaussDistribution import GaussDistribution

gaussDistr = GaussDistribution(1, numberOfSamplesToGenerate = 1000, mean = 2,
    variance = 1)
gaussDistr.plotData()

gaussDistr = GaussDistribution(2, numberOfSamplesToGenerate = 1000, mean = (2,
    2), variance = (1, 1))
gaussDistr.plotData()

betaDistr = BetaDistribution(0.5,0.5, numberOfSamplesToGenerate = 1000)
betaDistr.plotData()

betaDistr = BetaDistribution(2,5, numberOfSamplesToGenerate = 1000)
betaDistr.plotData()

betaDistr = BetaDistribution(5,2, numberOfSamplesToGenerate = 1000)
betaDistr.plotData()

settings = [(0.5, 0.5), (2, 5), (5, 2)]
betaDistr.plotDataWithDifferentAlphasAndBetas(settings)
```

```python
import csv
import numpy as np
import matplotlib.pyplot as plt
import math

from abc import ABC, abstractmethod

class ContiniousDustribution():

    def __init__(self):
        self.dataSet = []
        self.normalizedDataSet = []
        self.mean = None
        self.median = None
        self.variance = None
        self.standardDeviation = None

    def importCsv(self, filename):
        if len(self.dataSet) != 0:
            raise Exception("Data already added")

        with open(filename, mode="r") as file:
            csvFile = csv.reader(file)

            for row in csvFile:
                self.dataSet.append(row)

    def exportCsv(self, filename):
        if len(self.dataSet) == 0:
            raise Exception("No Data added")

        with open(filename, mode="w") as file:
            csvWriter = csv.writer(file, delimiter = ";")
            csvWriter.writerows(self.dataSet)

    def calculateMean(self):
        self.mean = np.mean(self.dataSet)

    def calculateVariance(self):
        length = len(self.dataSet)
        mean = self.mean

        squareDeviations = [(x - mean) ** 2 for x in self.dataSet]

        # Bessel's correction (n-1) instead of n for better results
        self.variance = sum(squareDeviations) / (length - 1)
        return self.variance

    def calculateStandardDeviation(self):
        self.standardDeviation = math.sqrt(self.variance)
        return self.standardDeviation

    def normalizeDataSet(self):
        self.dataSet = [((x - self.mean)/self.standardDeviation) for x in self.
            dataSet]

    @abstractmethod
    def generateSampels(self):
        pass
```

```python

    @abstractmethod
    def plotData(self):
        pass
```

```
"""
@author: Nehl Stefan
"""

import numpy as np
import matplotlib.pyplot as plt
import math
from inference import ContiniousDustribution

class GaussDistribution(ContiniousDustribution):

    def __init__(self, dimension, fileName = None, numberOfSamplesToGenerate =
        None, mean = None, variance = None):
        if((fileName is not None) & (numberOfSamplesToGenerate is not None)):
            raise Exception("Can't load data and generate samples")

        ContiniousDustribution.__init__(self)
        self.dimension = dimension

        if(fileName is not None):
            self.importCsv(fileName)
            self.numberOfSamples = len(self.dataSet)
            self.calculateMean()
            self.calculateVariance()

        if(numberOfSamplesToGenerate is not None) & \
                (mean is not None) & \
                (variance is not None):
            self.numberOfSamples = numberOfSamplesToGenerate
            self.mean = np.array(mean)
            self.variance = np.array(variance)
            self.generateSampels()

        if(len(self.dataSet) == 0):
            raise Exception("Could not generate data, verify parameters")

        self.calculateStandardDeviation()
        self.gaussenDistribution = []
        self.generateGaussen()

    def generateSampels(self):
        if len(self.dataSet) != 0:
            raise Exception("Data already added")

        self.dataSet =  np.random.default_rng().normal(self.mean, self.variance,
             size=(self.numberOfSamples, self.dimension))

    def generateGaussen(self):
        if len(self.dataSet) == 0:
            raise Exception("No Data added")

        if self.dimension == 1:
            return self.generateGaussen1D()

        return self.generateGaussen2D()

    def calculateMean(self):
        self.mean = np.mean(self.dataSet)

```

```
    def calculateStandardDeviation(self):
        if self.dimension == 1:
            self.standardDeviation = np.std(self.dataSet)
        else:
            self.standardDeviation = np.array((math.sqrt(self.mean[0]), math.
                sqrt(self.mean[1])))

    def calculateGaussen1D(self, x):
        exponentialTerm = (-(1 / (2 * self.variance ** 2)) * (x - self.mean) **
            2)
        denominator = (2 * math.pi * self.variance ** 2) ** (0.5)
        return (1 / denominator) * math.e ** (exponentialTerm)

    def generateGaussen1D(self):
        vectorArray = np.array(self.dataSet)
        self.gaussen = []

        for x in vectorArray:
            self.gaussen.append(self.calculateGaussen1D(x))

    def calculateGaussen2D(self, vector):
        xs = [self.mean[0], 0]
        ys = [0, self.mean[1]]
        covariance = [xs, ys]
        inverseCovariance = np.linalg.inv(covariance)
        determinantCovariance = np.linalg.det(covariance)

        # exponentialTerm = (-0.5 * np.transpose(vector - self.mean)) *
            inverseCovariance * (vector - self.mean)
        exponentialTerm = -(np.linalg.solve(covariance, (vector - self.mean)).T.
            dot((vector - self.mean))) / 2
        denominator = ((2 * math.pi) ** (self.dimension / 2)) *
            determinantCovariance ** (0.5)
        result = (1 / denominator) * math.exp(exponentialTerm)
        return result

    def generateGaussen2D(self):
        vectorArray = np.array(self.dataSet)

        self.gaussenDistribution = []

        for vector in vectorArray:
            self.gaussenDistribution.append(self.calculateGaussen2D(vector))

    def plotData(self):
        if len(self.dataSet) == 0:
            raise Exception("No Data added")

        if self.dimension == 1:
            self.plotData1D()
        else:
            self.plotData2D()

    def plotData1D(self):
        plotRange = range(len(self.dataSet))
        x = np.linspace(min(self.dataSet), max(self.dataSet), self.
            numberOfSamples)
        y = self.calculateGaussen1D(x)

```

```python
        plt.figure(figsize=(6, 6))

        plt.subplot(2, 1, 1)
        plt.hist(self.dataSet, bins=60, density=True, label="Histogram")
        plt.plot(x, y, "r-", linewidth=1, label="Distribution")

        plt.title("Distribution")
        plt.xlabel("Value")
        plt.ylabel("Frequency")
        plt.legend(loc="upper right")

        plt.subplot(2, 1, 2)
        plt.scatter(plotRange, self.dataSet, label="Data", s=2)

        plt.title(f"Raw data with n = {self.numberOfSamples} sample points")
        plt.xlabel("Sample")
        plt.ylabel("Value")
        plt.legend(loc="best")

        plt.suptitle(f"Gaus Distribution with $\mu$ = {self.mean} and $\sigma$ = {self.variance}")

        plt.tight_layout()
        plt.show()

    def plotData2D(self):
        plt.figure(figsize=(8, 12))

        hist, xedges, yedges = np.histogram2d(self.dataSet[:,0], self.dataSet[:,1], bins=60)

        # Construct arrays for the anchor positions of the 16 bars.
        xpos, ypos = np.meshgrid(xedges[:-1] + 0.25, yedges[:-1] + 0.25, indexing="ij")
        xpos = xpos.ravel()
        ypos = ypos.ravel()
        zpos = 0

        # Construct arrays with the dimensions for the 16 bars.
        dx = dy = 0.5 * np.ones_like(zpos)
        dz = hist.ravel()

        ax = plt.subplot(2, 1, 1, projection="3d")
        ax.bar3d(xpos, ypos, zpos, dx, dy, dz)
        ax.set_zlabel("Frequency")

        xy = np.linspace([min(self.dataSet[0]),min(self.dataSet[1])], [max(self.dataSet[0]), max(self.dataSet[1])], self.numberOfSamples)
        # y = np.linspace(min(self.dataSet[1]), max(self.dataSet[1]), self.numberOfSamples)
        z = np.array([self.calculateGaussen2D(v) for v in xy])
        # does not work
        # ax.plot_surface(xy[0], xy[1], z, )

        plt.title("Distribution")
        plt.xlabel("X")
        plt.ylabel("Y")
        plt.legend(loc="best")

```

```
        ax = plt.subplot(2, 1, 2, projection="3d")
        plotRange = range(self.numberOfSamples)
        ax.scatter3D(plotRange, self.dataSet[:, 0], self.dataSet[:, 1], label="Data", s=2)

        plt.title(f"Raw data with n = {self.numberOfSamples} sample points")
        plt.xlabel("Sample")
        plt.ylabel("Value")
        plt.legend(loc="best")

        plt.suptitle(f"Gaus Distribution with $\mu$ = {self.mean} and $\sigma$ = {self.variance}")

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

```python
"""
@author: Nehl Stefan
"""

import numpy as np
import matplotlib.pyplot as plt
import math
from inference import ContiniousDustribution

class BetaDistribution(ContiniousDustribution):

    def __init__(self, a, b, fileName = None, numberOfSamplesToGenerate = None):
        if((fileName is not None) & (numberOfSamplesToGenerate is not None)):
            raise Exception("Can't load data and generate samples")

        if((fileName is None) & (numberOfSamplesToGenerate is None)):
            raise Exception("No parameters for data")

        ContiniousDustribution.__init__(self)
        self.a = a
        self.b = b
        self.calculateAndSetBFromAAndB()
        self.generatedBetaDistribution = []

        if(numberOfSamplesToGenerate is not None):
            self.numberOfSamples = numberOfSamplesToGenerate
            self.generateSampels()

        if (fileName is not None):
            self.importCsv(fileName)
            self.numberOfSamples = len(self.dataSet)

        self.calculateMean()
        self.calculateVariance()
        self.calculateStandardDeviation()
        self.generateBetaDistribution()

    def generateSampels(self):
        if len(self.dataSet) != 0:
            raise Exception("Data already added")

        self.dataSet = np.random.default_rng().beta(self.a, self.b, size=self.
            numberOfSamples)

    def calculateAndSetBFromAAndB(self):
        self.bFromAAndB = (math.gamma(self.a + self.b) / (math.gamma(self.a) +
            math.gamma(self.b)))

    def calculateBeta(self, x):
        return self.bFromAAndB * pow(x, (self.a - 1)) * pow((1 - x), (self.b -
            1))

    def generateBetaDistribution(self):
        self.generatedBetaDistribution = []

        for x in self.dataSet:
            result = self.calculateBeta(x)
            self.generatedBetaDistribution.append(result)

```

```python
    def plotData(self):
        plotRange = range(len(self.dataSet))
        x = np.linspace(0.01, 0.99, self.numberOfSamples)
        y = self.calculateBeta(x)

        plt.figure(figsize=(6, 6))

        plt.subplot(2, 1, 1)
        plt.hist(self.dataSet, bins=60, density=True, label="Histogram")
        plt.plot(x, y, "r-", linewidth=1, label=f"alpha = {self.a}, beta = {self.b}")

        plt.title("Distribution")
        plt.xlabel("Value")
        plt.ylabel("Frequency")
        plt.legend(loc="upper right")

        plt.subplot(2, 1, 2)
        plt.scatter(plotRange, self.dataSet, label="Data", s=2)

        plt.title("Raw Data")
        plt.xlabel("Sample")
        plt.ylabel("Value")
        plt.legend(loc="upper right")

        plt.suptitle(f"Beta Distribution with alpha = {self.a} and beta = {self.b}")

        plt.tight_layout()
        plt.show()

    def plotDataWithDifferentAlphasAndBetas(self, alphaAndBetas):
        if len(alphaAndBetas) == 0:
            return

        plt.figure(figsize=(4, 4))
        plt.title("Distribution")
        plt.xlabel("Value")
        plt.ylabel("Frequency")

        for alphaAndBeta in alphaAndBetas:
            self.a = alphaAndBeta[0]
            self.b = alphaAndBeta[1]
            self.calculateAndSetBFromAAndB()

            x = np.linspace(0.01, 0.99, self.numberOfSamples)
            y = self.calculateBeta(x)
            plt.plot(x, y, linewidth=1, label=f"alpha = {self.a}, beta = {self.b}")

        plt.legend(loc = "best")

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

Bibliography

Roelants, Peter (2018). *Multivariate normal distribution*. URL: `https://peterroelants.github.io/posts/multivariate-normal-primer/`.

Rueckert, Elmar (2022). *An Introduction to Probabilistic Machine Learning*. Elmar Rueckert.