Assignment III: Probability Theory

Introduction to Machine Learning Lab (190.013), SS2023
Björn Ellensohn[1]

[1]*m01435615*, *bjoern.ellensohn@stud.unileoben.ac.at*, *Montanuniversität Leoben, Austria*
April 27, 2023

This document guides through the process of solving Assignment 3.

1 Introduction

The Task of 3rd assignment was coming up with an abstract class called `ContinuousDistribution` that provides the outline for two subclasses `GaussDistribution` and `BetaDistribution`. Again, provided was a .csv file with a dataset. The assignment is divided into three main parts and the bonus part.

2 Part I - Abstract Class

2.1 Preparation

When searching online for information on Python's abstract classes, you may come across the module known as `ABC`. This module includes a specific decorator, `@abstractmethod`, that can be used to define abstract methods. Essentially, this allows you to create a class that serves as a template or blueprint for other classes, without actually implementing the methods. To create a functional child class, you must define and implement the necessary methods there.

2.2 Defining the Abstract Class

The class `ContinuousDistribution` is created that provides the basic outline for the main classes of this exercise. The following functions should be inlcuded:

- Data Import and Export using csv files.
- Computation of the mean based on the samples from the csv.
- Computation of the standard deviation based on the samples from the csv.
- Visualization of the distribution, the raw data or the generated samples.
- Generating/Drawing Samples from the distribution.

So in the end, the abstract class `ContinuousDistribution` should look like this:

```python
import abc

class
↪  ContinuousDistribution(metaclass=abc.ABCMeta):
@abc.abstractmethod
def import_data(self, file_path):
    pass

@abc.abstractmethod
def export_data(self, data, file_path):
    pass

@abc.abstractmethod
def compute_mean(self, data):
    pass

@abc.abstractmethod
def compute_standard_deviation(self, data):
    pass

@abc.abstractmethod
def visualize(self, data=None):
    pass

@abc.abstractmethod
def generate_samples(self, n_samples):
    pass
```

3 Part II - Plot and Sample Gaussian Distributions

In the second part of this exercise, it is required to initiate a child class which is responsible for dealing with Gaussian distributions.

Explicitly, the following should be implemented:

- Implement the functions defined in “ContinuousDistribution”.
- Implement a constructor that optionally takes the dimension of the multivariate distribution.
- Implement a visualization for Multivariate Gaussians up to 3 dimensions.
- Find the empirical parameters of the distribution that created the samples in the ‘MGD.csv’ file.
- Plot the samples of the ‘MGD.csv’ file and the sample from the learned distribution in two subfigures.

So the main goal of this class is to be able to compute the statistical parameters from a provided dataset and being able to sample a new dataset from this distribution. In the end, the datapoints should be plotted.

At first, let us declare how the Gaussian dsitribution is defined. For all normal distributions, 68.2% of the observations will appear within plus or minus one standard deviation of the mean; 95.4% of the observations will fall within +/- two standard deviations; and 99.7% within +/- three standard deviations. This fact is sometimes referred to as the "empirical rule," a heuristic that describes where most of the data in a normal distribution will appear. This means that data falling outside of three standard deviations ("3-sigma") would signify rare occurrences.

In mathematical terms this is explained in Equation 1.

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

For sampling from the Gaussian Distribution, a Box-Muller Transform could be used (Keng, 2015). But for simplicity, I will use a module called `multivariate_normal` from the `scipy` package.

The constructor of `GaussianDistribution` then looks like this:

```
class GaussDistribution(ContinuousDistribution):
    def __init__(self, dim=1):
        self.dim = dim
        self.mean = np.zeros(dim)
        self.covariance = np.eye(dim)
        self.data = pd.DataFrame()
        self.samples = None
```

Figure 1 to Figure 3 are showing the end results. In the appendix you will find the full code.

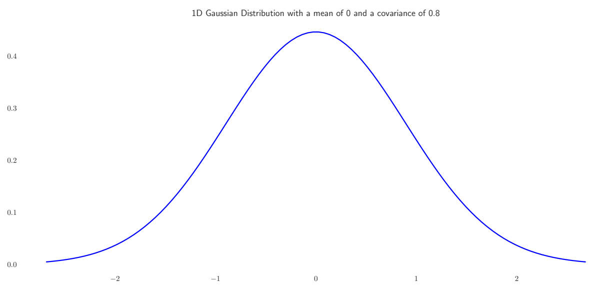

Figure 1: *Plot of one dimensional Gaussian distribution.*

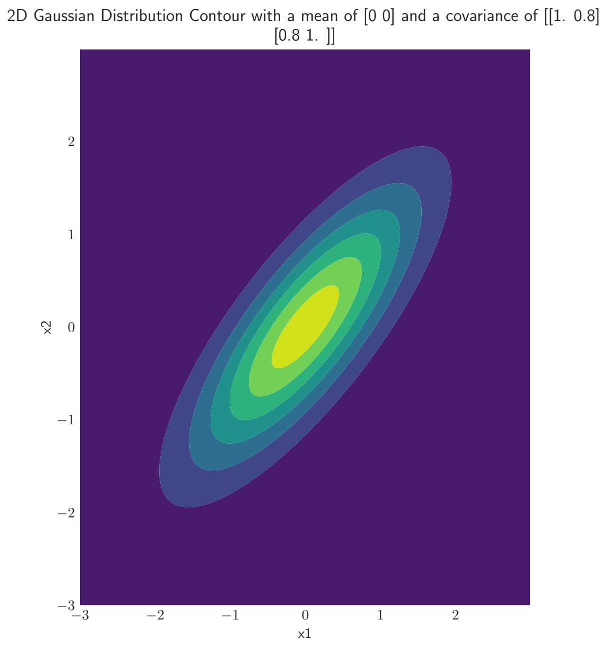

Figure 2: *Contour plot of two dimensional Gaussian distribution.*

4 Part III - Plot and Sample Beta Distributions

Next, a class `BetaDistribution` shouldbe inherited from the meta class. It should implement:

- Generate beta distributed samples and plot the distribution giving the parameters a and b.

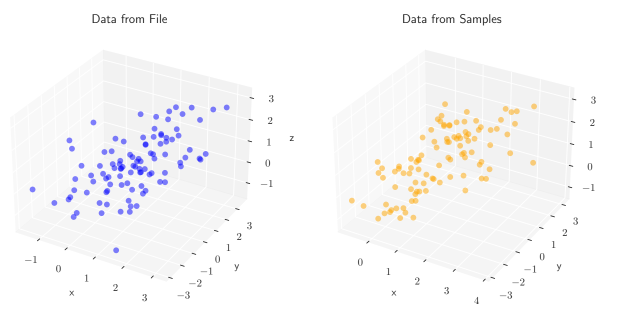

Figure 3: *Plots of three dimensional Gaussian distribution.*

- The constructor should take the parameters a and b as arguments.
- A visualization for Beta distributions, including the mean and the standard deviation lines.

Again, there is a module in the `scipy` package that does the hard work here `beta`. The constructor of `BetaDistribution` then looks like this:

```
class
↪  BetaDistribution(ContinuousDistribution):
def __init__(self, a, b):
    self.a = a
    self.b = b
    self.data = None
```

Provided is a plot showing the end results in Figure 4.

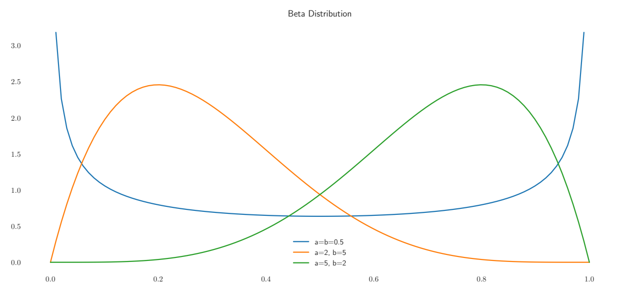

Figure 4: *Plot of three different Beta Distributions.*

5 Part IV - Bonus Exercise

We were provided with a fictional card game with the aim to better understand the Bayes' Rule. A deck of four known cards is shuffled, a card is chosen until the deck is empty. To solve this, missing functions need to be filled. Those functions are:

- `update_probabilities`
- `get_posterior`
- `get_likelihood`
- `get_marginal_likelihood`

I tried to solve the exercise but in the end I ran out of time. Still, I think I was close. The code will be attached in the appendix. Here I will try to explain my functions. The function `update_probabilities` is just there to call the `get_posterior` function. The function `get_posterior` returns the posterior, based on the likelyhood, the prior and the marginal likelyhood. Likewise, `get_likelihood` returns the likelyhood of drawing a certain card (red or black) given the color of the drawn card. Last but not least `get_marginal_likelihood` returns the marginal likelyhood of drawing a certain card (red or black) given the color of the drawn card.

APPENDIX

This section houses the code.

Code for Part I

```python
import abc

class ContinuousDistribution(metaclass=abc.ABCMeta):
    @abc.abstractmethod
    def import_data(self, file_path):
        pass

    @abc.abstractmethod
    def export_data(self, data, file_path):
        pass

    @abc.abstractmethod
    def compute_mean(self, data):
        pass

    @abc.abstractmethod
    def compute_standard_deviation(self, data):
        pass

    @abc.abstractmethod
    def visualize(self, data=None):
        pass

    @abc.abstractmethod
    def generate_samples(self, n_samples):
        pass
```

Code for Part II

```python
import numpy as np
import matplotlib
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
from scipy.stats import multivariate_normal
import pandas as pd


class GaussDistribution(ContinuousDistribution):
    def __init__(self, dim=1):
        self.dim = dim
        self.mean = np.zeros(dim)
        self.covariance = np.eye(dim)
        self.data = pd.DataFrame()
        self.samples = None

    def import_data(self, file_path):
        # implementation to import data from file
        self.data = pd.read_csv(file_path)

    def export_data(self, data, file_path):
        # implementation to export data to file
        df = pd.DataFrame(data)
        df.to_csv(file_path)

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

    def compute_standard_deviation(self, data):
        self.covariance = np.cov(data, rowvar=False)

    def visualize(self, data=None):
        if data is None:
            data = multivariate_normal.rvs(mean=self.mean, cov=self.covariance, size=1000)

        if self.dim == 1:
            mean = 0
            covariance = 0.8
            x = np.linspace(mean - 3*np.sqrt(covariance), mean + 3*np.sqrt(covariance), 100)
            plt.plot(x, multivariate_normal.pdf(x, mean=mean, cov=covariance), color = 'blue')
            plt.title(f'1D Gaussian Distribution with a mean of {mean} and a covariance of {covariance}')

            plt.savefig('gaussian1D.pdf', bbox_inches='tight', transparent=True)
            plt.show()

        elif self.dim == 2:
            covariance = np.array([[1, 0.8],
                                     [0.8, 1]])
            mean = np.array([0, 0])
            x, y = np.mgrid[mean[0]-3*np.sqrt(covariance[0,0]):mean[0]+3*np.sqrt(covariance[0,0]):.01,
                            mean[1]-3*np.sqrt(covariance[1,1]):mean[1]+3*np.sqrt(covariance[1,1]):.01]
            pos = np.empty(x.shape + (2,))
            pos[:, :, 0] = x
            pos[:, :, 1] = y
            rv = multivariate_normal(mean, covariance)

            # Generating the density function
            # for each point in the meshgrid
            pdf = np.zeros(x.shape)
            for i in range(x.shape[0]):
                for j in range(x.shape[1]):
                    pdf[i,j] = rv.pdf([x[i,j], y[i,j]])

            pdf_list = []
            fig = plt.figure()

            # Plotting the density function values
            bx = fig.add_subplot(131, projection = '3d')
            bx.plot_surface(x, y, pdf, cmap = 'viridis')
            plt.xlabel("x1")
            plt.ylabel("x2")
            plt.title(f'2D Gaussian Distribution with a mean of {mean} and a covariance of {covariance}')
            pdf_list.append(pdf)
            bx.axes.zaxis.set_ticks([])

            plt.tight_layout()

            plt.savefig('gaussian2D_surface.pdf', bbox_inches='tight', transparent=True)
            plt.show()

            # Plotting contour plots
            for idx, val in enumerate(pdf_list):
                plt.subplot(1,3,idx+1)
                plt.contourf(x, y, val, cmap='viridis')
                plt.xlabel("x1")
                plt.ylabel("x2")
            plt.tight_layout()
            plt.title(f'2D Gaussian Distribution Contour with a mean of {mean} and a covariance
            ↪ of {covariance}')

            plt.savefig('gaussian2D_contour.pdf', bbox_inches='tight', transparent=True)
            plt.show()

        elif self.dim == 3:
            fig = plt.figure(figsize=(10, 5))
            ax1 = fig.add_subplot(1, 2, 1, projection='3d')
            x, y, z = np.mgrid[self.mean[0]-3*np.sqrt(self.covariance[0,0]):self.mean[0]+3*np.sqrt ⌋
            ↪ (self.covariance[0,0]):.1,
                               self.mean[1]-3*np.sqrt(self.covariance[1,1]):self.mean[1]+3*np.sqrt ⌋
                               ↪ (self.covariance[1,1]):.1,
                               self.mean[2]-3*np.sqrt(self.covariance[2,2]):self.mean[2]+3*np.sqrt ⌋
                               ↪ (self.covariance[2,2]):.1]
            #Plot the samples from the file

            ax1.scatter(data.iloc[:, 0], data.iloc[:, 1], data.iloc[:, 2], c='blue', alpha=0.5)
            ax1.set_xlabel('x')
            ax1.set_ylabel('y')
            ax1.set_zlabel('z')
            ax1.set_title('Data from File')

            ax2 = fig.add_subplot(1, 2, 2, projection='3d')
            ax2.scatter(self.samples[:, 0], self.samples[:, 1], self.samples[:, 2], c='orange',
            ↪ alpha=0.5)
            ax2.set_xlabel('x')
            ax2.set_ylabel('y')
            ax2.set_zlabel('z')
            ax2.set_title('Data from Samples')

            plt.savefig('gaussian3D.pdf', bbox_inches='tight', transparent=True)
            plt.show()

    def generate_samples(self, n_samples):
        self.samples = multivariate_normal.rvs(mean=self.mean, cov=self.covariance, size=n_samples)
```

Code for Part III

```
from scipy.stats import beta

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

    def import_data(self, file_path):
        self.data = pd.read_csv(file_path)

    def export_data(self, data, file_path):
        pass

    def compute_mean(self, data):
        pass

    def compute_standard_deviation(self, data):
        pass

    def visualize(self, data=None):
        # create a range of x values
        x = np.linspace(0, 1, 100)

        # calculate the beta PDF for the given parameters a and b
        y = beta.pdf(x, self.a, self.b)

        # plot the beta PDF
        plt.plot(x, y, label='Beta PDF')

        # plot the mean and standard deviation lines
        mean = beta.mean(self.a, self.b)
        std = beta.std(self.a, self.b)
        plt.axvline(mean, color='red', label=f'Mean={mean:.2f}')
        plt.axvline(mean - std, linestyle='--', color='green', label=f'Std Dev={std:.2f}')
        plt.axvline(mean + std, linestyle='--', color='green')

        # set the plot title and legend
        plt.title(f'Beta Distribution (a={self.a}, b={self.b})')
        plt.legend()

        # show the plot
        plt.savefig('beta.pdf', bbox_inches='tight', transparent=True)

        plt.show()

    def visualize_book(self, data=None):
        # create a range of x values
        x = np.linspace(0, 1, 100)

        # calculate the beta PDF for the given parameters a and b
        y1 = beta.pdf(x, 0.5, 0.5)
        y2 = beta.pdf(x, 2, 5)
        y3 = beta.pdf(x, 5, 2)

        # plot the beta PDF
        plt.plot(x, y1, label='a=b=0.5')
        plt.plot(x, y2, label='a=2, b=5')
        plt.plot(x, y3, label='a=5, b=2')

        # set the plot title and legend
        plt.title(f'Beta Distribution')
        plt.legend()

        # show the plot
        plt.savefig('beta.pdf' , bbox_inches='tight', transparent=True)
        plt.show()

    def generate_samples(self, n_samples):
        # generate beta distributed samples using the given parameters a and b
        return beta.rvs(self.a, self.b, size=n_samples)
```

Code for Part IV

```
import random

class CardDeck:
    def __init__(self):
        # Initialize the deck of cards
        self.cards = ['R', 'R', 'B', 'B']
        # Shuffle the deck of cards
        random.shuffle(self.cards)
        # Initialize the prior probability
        self.prior_red = 0.5
```

```python
        self.likelihood = 0.5


    def draw_card(self, index):
        """
        Draw a card from the deck
        :param index: the index of the card to be drawn
        :return: the card and its color
        """
        card = self.cards.pop(index)
        color = 'Red' if card == 'R' else 'Black'
        return (card, color)

    def get_card_color(self, index):
        """
        Get the color of a card
        :return: the color of the card
        """
        card = self.cards[index]
        color = 'Red' if card == 'R' else 'Black'
        return color

    def get_remainder(self):
        """
        Get the remainder deck
        :return: the remainder deck
        """
        return self.cards

    """
    Update the prior probability based on Bayes' rule
    function name: update_probabilities
    """
    def update_probabilities(self, card_color):
        """
        Update the prior probability based on Bayes' rule
        :param card_color: the color of the card drawn
        """
        if card_color == 'Red':
            self.prior_red = self.get_posterior('Red')
        elif card_color == 'Black':
            self.prior_red = self.get_posterior('Black')

    """
    Calculate the posterior probability based on the prior probability, likelihood, and marginal
↪   likelihood
    function name: get_posterior
    """
    def get_posterior(self, card_color):
        """
        Calculate the posterior probability based on the prior probability, likelihood, and
↪   marginal likelihood
        :param card_color: the color of the card drawn
        :return: the posterior probability of drawing a red card or a black card
        """
        likelihood = self.get_likelihood(card_color)
        marginal_likelihood = self.get_marginal_likelihood(card_color)
        posterior = (self.prior_red * likelihood) / marginal_likelihood
        return posterior

    def get_likelihood(self, card_color):
        """
        Calculate the likelihood of drawing a certain card (red or black) given the color of the
↪   card already drawn
```

```python
        :param card_color: the color of the card drawn
        :return: the likelihood of drawing a certain card (red or black) given the color of the card already drawn
        """
        if card_color == 'Red':
            num_red_cards = self.cards.count('R')
            num_cards = len(self.cards)
            return num_red_cards / num_cards
        elif card_color == 'Black':
            num_black_cards = self.cards.count('B')
            num_cards = len(self.cards)
            return num_black_cards / num_cards

    def get_marginal_likelihood(self, card_color):
        """
        Calculate the marginal likelihood of drawing a certain card (red or black) regardless of the color of the card already drawn
        :param card_color: the color of the card drawn
        :return: the marginal likelihood of drawing a certain card (red or black) regardless of the color of the card already drawn
        """
        if card_color == 'Red':
            num_red_cards = self.cards.count('R')
            num_black_cards = self.cards.count('B')
            return (self.prior_red * num_red_cards / len(self.cards)) + ((1 - self.prior_red) * num_black_cards / len(self.cards))
        elif card_color == 'Black':
            num_red_cards = self.cards.count('R')
            num_black_cards = self.cards.count('B')
            return ((1 - self.prior_red) * num_red_cards / len(self.cards)) + (self.prior_red * num_black_cards / len(self.cards))


    def play_game(self):
        while self.cards:
            for i, card in enumerate(self.cards):
                print(f'{i}: Unknown')
             # only one card left
            if len(self.cards) == 1:
                # code missing here
                self.cards.pop()
            else:
                index = int(input(f"Remainder deck: {self.get_remainder()}\nEnter the index of the card you want to draw:"))
                card, color = self.draw_card(index-1)
                print(f'You drew a {color} card with value {card}')

                # Calculate the likelihood and marginal likelihood based on the color of the drawn card
                self.get_likelihood(color)
                self.get_marginal_likelihood(color)

            # Update the prior probability based on Bayes' rule
            self.update_probabilities(color)
            posterior_red = self.get_posterior('Red')  # you must calculate this value
            posterior_black = self.get_posterior('Black') # you must calculate this value
            # Print the posterior probabilities
            print(f"After drawing a {card} card:")
            print(f"The probability that the other card is red: {posterior_red:.2f}")
            print(f"The probability that the other card is black: {posterior_black:.2f}")
        print("The deck is empty.")
```

Bibliography

Keng, Brian. “Sampling from a Normal Distribution | Bounded Rationality.” Sampling from a Normal Distribution, November 28, 2015. https://bjlkeng.github.io/posts/sampling-from-a-normal-distribution/.