Foundation of Data Engineering

Experiment Record

Sarvesh Adithya J

Registration Number: 2512004

Contents

1 Experiment 1: Activation and Basic Functions

1.1 Activation Functions

1.1.1 Sigmoid Function

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

def sigmoid(x):
    return 1 / (1 + np.exp(-x))

def sigmoid_derivative(x):
    s = sigmoid(x)
    return s * (1 - s)

x = np.linspace(-10, 10, 1000)
y_sigmoid = sigmoid(x)
y_derivative = sigmoid_derivative(x)

plt.figure(figsize=(10, 5))
plt.subplot(1, 2, 1)
plt.plot(x, y_sigmoid, label="Sigmoid", color='green')
plt.title("Sigmoid Function")
plt.xlabel("x")
plt.ylabel("sigma(x)")
plt.grid(True)
plt.legend()

plt.subplot(1, 2, 2)
plt.plot(x, y_derivative, label="Sigmoid Derivative", color='orange')
plt.title("Derivative of Sigmoid")
plt.xlabel("x")
plt.ylabel("d(sigma)/dx")
plt.grid(True)
plt.legend()

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

Listing 1: Sigmoid function and its derivative

Output:

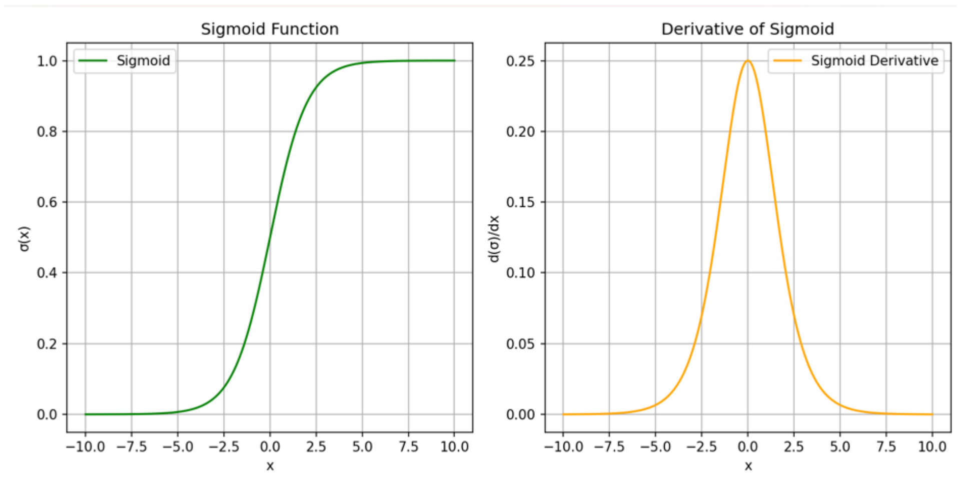


Figure 1: Sigmoid function and its derivative

1.1.2 ReLU Function

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

def relu(x):
    return np.maximum(0, x)

def relu_derivative(x):
    return np.where(x > 0, 1, 0)

x = np.linspace(-10, 10, 1000)
y_relu = relu(x)
y_derivative = relu_derivative(x)

plt.figure(figsize=(10, 5))
plt.subplot(1, 2, 1)
plt.plot(x, y_relu, label="ReLU", color='blue')
plt.title("ReLU Function")
plt.xlabel("x")
plt.ylabel("ReLU(x)")
plt.grid(True)
plt.legend()

plt.subplot(1, 2, 2)
plt.plot(x, y_derivative, label="ReLU Derivative", color='red')
plt.title("Derivative of ReLU")
plt.xlabel("x")
plt.ylabel("d(ReLU)/dx")
plt.grid(True)
plt.legend()

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

Listing 2: ReLU function and its derivative

Output:

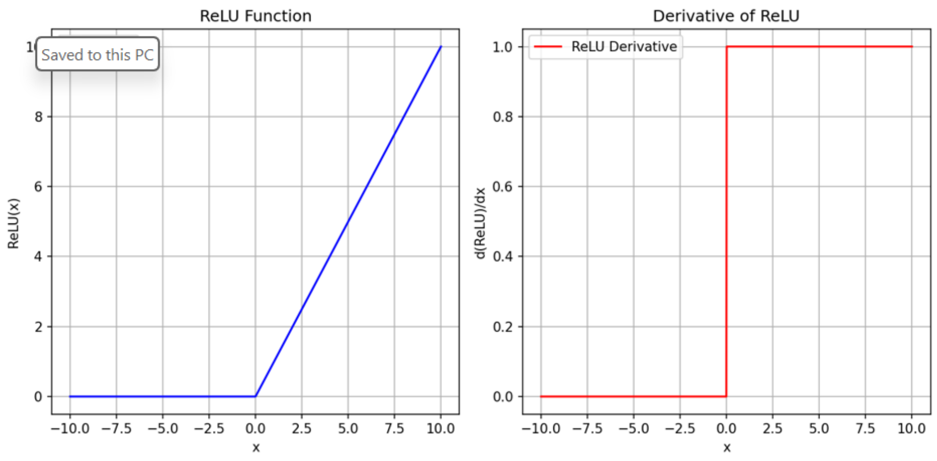


Figure 2: ReLU function and its derivative

1.1.3 Tanh Function

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

def tanh(x):
    return np.tanh(x)

def tanh_derivative(x):
    return 1 - np.tanh(x)**2

x = np.linspace(-10, 10, 1000)
y_tanh = tanh(x)
y_derivative = tanh_derivative(x)

plt.figure(figsize=(10, 5))
plt.subplot(1, 2, 1)
plt.plot(x, y_tanh, label="Tanh", color='purple')
plt.title("Tanh Function")
plt.xlabel("x")
plt.ylabel("tanh(x)")
plt.grid(True)
plt.legend()

plt.subplot(1, 2, 2)
plt.plot(x, y_derivative, label="Tanh Derivative", color='brown')
plt.title("Derivative of Tanh")
plt.xlabel("x")
plt.ylabel("d(tanh)/dx")
plt.grid(True)
plt.legend()

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

Listing 3: Tanh function and its derivative

Output:

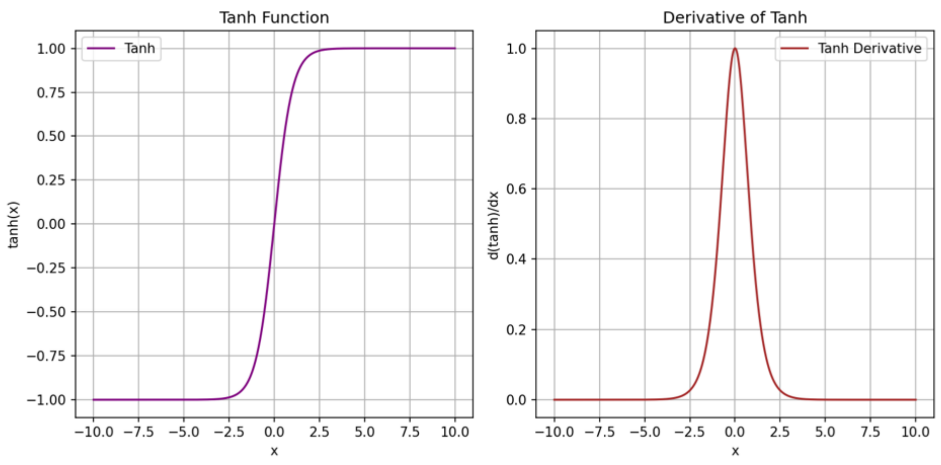


Figure 3: Tanh function and its derivative

1.2 Basic Functions

1.2.1 $f(x) = e^{-x}$

```
import numpy as np
import matplotlib.pyplot as plt

def f(x):
    return np.exp(-x)

def f_derivative(x):
    return -np.exp(-x)

x = np.linspace(-5, 5, 500)
y = f(x)
y_prime = f_derivative(x)

plt.figure(figsize=(10, 5))
plt.plot(x, y, label=r'$f(x) = e^{-x}$', color='blue')
plt.plot(x, y_prime, label=r"$f'(x) = -e^{-x}$", color='red', linestyle
    ='--')
plt.title("Function and Derivative: $f(x) = e^{-x}$")
plt.xlabel("x")
plt.ylabel("y")
plt.axhline(0, color='black', linewidth=0.5)
plt.axvline(0, color='black', linewidth=0.5)
plt.grid(True)
plt.legend()
plt.tight_layout()
plt.show()
```

Listing 4: $f(x) = e^{-x}$ and its derivative

Output:

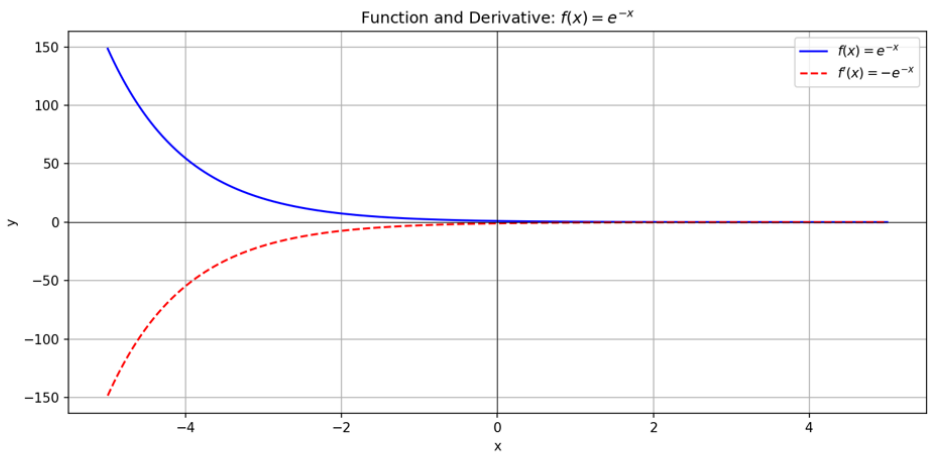


Figure 4: $f(x) = e^{-x}$ function and its derivative

1.2.2 $f(x) = e^{-|x|}$

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

def f(x):
    return np.exp(-np.abs(x))

def f_derivative(x):
    return np.where(x > 0, -np.exp(-x), np.where(x < 0, np.exp(x), 0))

x = np.linspace(-5, 5, 1000)
y = f(x)
y_prime = f_derivative(x)

plt.figure(figsize=(10, 5))
plt.plot(x, y, label=r'$f(x) = e^{-|x|}$', color='blue')
plt.plot(x, y_prime, label=r"$f'(x)$", color='orange', linestyle='--')
plt.title("Function and Derivative: $f(x) = e^{-|x|}$")
plt.xlabel("x")
plt.ylabel("y")
plt.grid(True)
plt.axhline(0, color='black', linewidth=0.5)
plt.axvline(0, color='black', linewidth=0.5)
plt.legend()
plt.tight_layout()
plt.show()
```

Listing 5: $f(x) = e^{-|x|}$ and its derivative

Output:

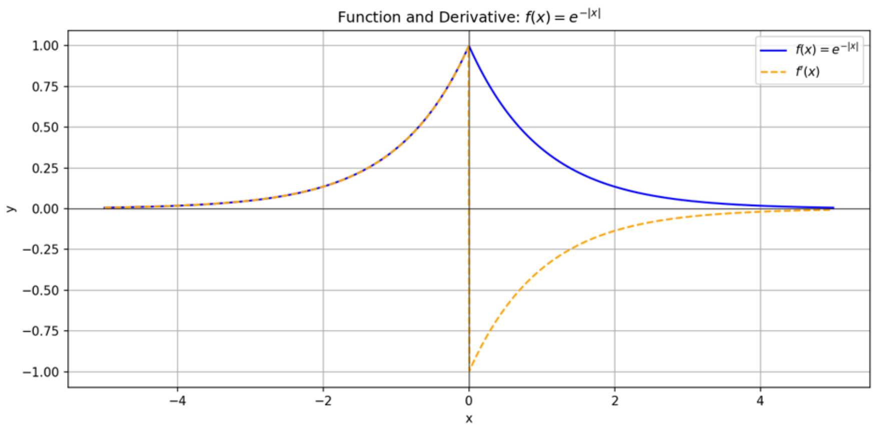


Figure 5: $f(x) = e^{-|x|}$ function and its derivative

1.2.3 $f(x) = e^{-x^2}$

```
import numpy as np
import matplotlib.pyplot as plt

def f(x):
    return np.exp(-x**2)

def f_derivative(x):
    return -2 * x * np.exp(-x**2)

x = np.linspace(-3, 3, 500)
y = f(x)
y_prime = f_derivative(x)

plt.figure(figsize=(10, 5))
plt.plot(x, y, label=r'$f(x) = e^{-x^2}$', color='blue')
plt.plot(x, y_prime, label=r"$f'(x) = -2x e^{-x^2}$", color='red',
    linestyle='--')
plt.title("Function and Derivative: $f(x) = e^{-x^2}$")
plt.xlabel("x")
plt.ylabel("y")
plt.grid(True)
plt.axhline(0, color='black', linewidth=0.5)
plt.axvline(0, color='black', linewidth=0.5)
plt.legend()
plt.tight_layout()
plt.show()
```

Listing 6: $f(x) = e^{-x^2}$ and its derivative

Output:

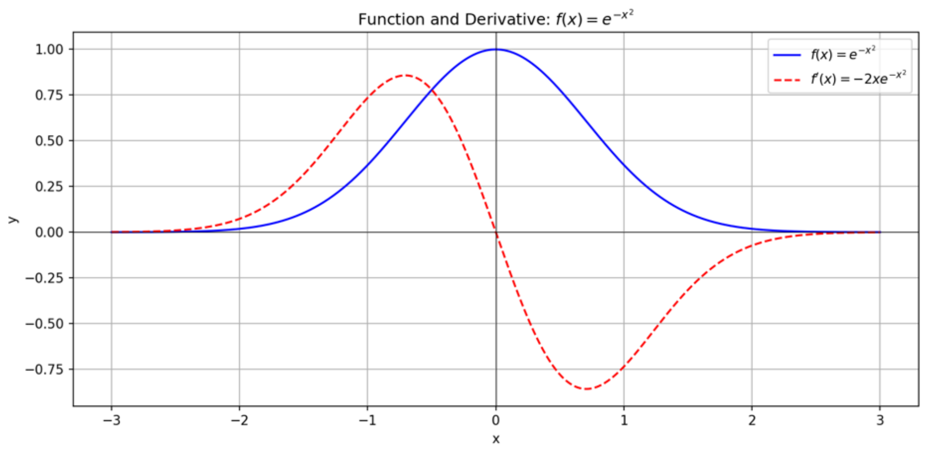


Figure 6: $f(x) = e^{-x^2}$ function and its derivative

2 Experiment 2: Student Data Analysis

Objective

To analyze student performance data using Python by applying normalization, correlation analysis, and extracting meaningful insights.

Tools Used

Python, NumPy, Pandas, Seaborn, Matplotlib

Step 1: Import Libraries

```python
import pandas as pd
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
```

Listing 7: Import Libraries

Step 2: Create Dataset

```python
np.random.seed(42)
data = {
    'Math_Score': np.random.randint(50, 100, 25),
    'English_Score': np.random.randint(45, 95, 25),
    'Science_Score': np.random.randint(40, 100, 25),
    'Attendance (%)': np.random.randint(70, 100, 25),
    'Study_Hours_per_Week': np.random.randint(5, 25, 25)
}
df = pd.DataFrame(data)
print("Original Student Dataset:\n")
df
```

Listing 8: Create Synthetic Student Dataset

Step 3: Data Normalization

```python
normalized_df = df.copy()
for column in normalized_df.columns:
    mean = normalized_df[column].mean()
    std = normalized_df[column].std()
    normalized_df[column] = (normalized_df[column] - mean) / std
print("\nNormalized Student Dataset:\n")
normalized_df
```

Listing 9: Normalize the Data

Step 4: Feature Ranges

```python
for column in df.columns:
    print(f"{column} - Min: {df[column].min()}, Max: {df[column].max()}
   ")
```

Listing 10: Show Feature Ranges

Step 5: Correlation Matrix and Heatmap

```python
correlation_matrix = df.corr()
sns.heatmap(correlation_matrix, annot=True, cmap='coolwarm')
plt.title("Correlation Heatmap")
plt.show()
```

Listing 11: Plot Correlation Heatmap

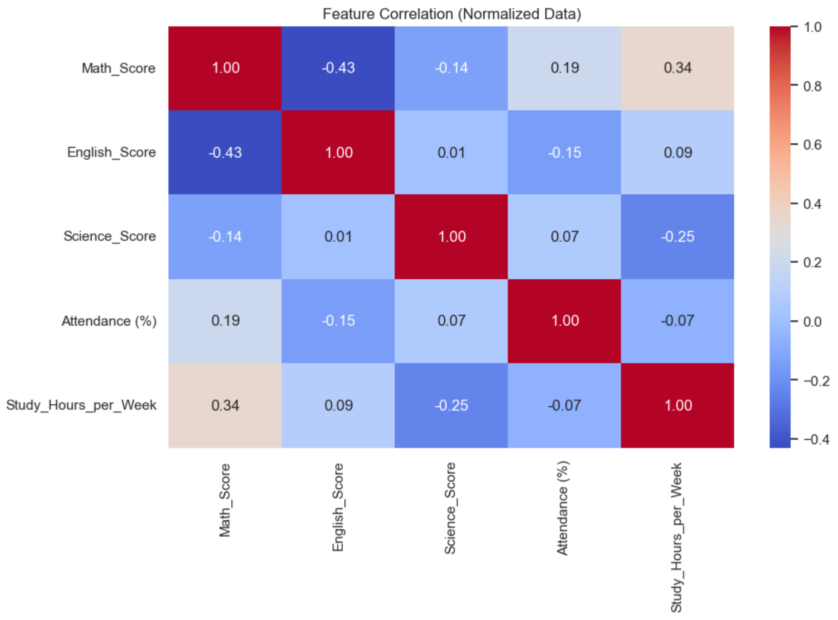


Figure 7: Correlation Heatmap of Student Features

Conclusion

This experiment shows how we can create and normalize synthetic student data, analyze the relationships between features using correlation, and visualize patterns. Such analysis helps in understanding how different factors like attendance and study hours relate to academic performance.

3 Experiment 3: Binomial Distribution – 100 Coin Tosses

3.1 Objective

To compute and visualize:

- The Probability Mass Function (PMF)
- The Cumulative Distribution Function (CDF)

for the Binomial distribution with $n = 100$ trials and $p = 0.5$ probability of success.

3.2 Introduction

The Binomial distribution is a way to understand how likely it is to get a certain number of successes in a set number of tries, where each try has only two possible outcomes — like win or lose. In this experiment, we flip a fair coin 100 times and look at how often we get heads to understand the pattern of results.

3.3 Methodology

We used the Python programming language with the following libraries:

- `numpy`: for array manipulations
- `matplotlib.pyplot`: for plotting PMF and CDF
- `scipy.stats.binom`: for computing PMF and CDF

3.4 Python Code

```python
import numpy as np
import matplotlib.pyplot as plt
from scipy.stats import binom

n = 100
p = 0.5

x = np.arange(0, n + 1)
pmf_values = binom.pmf(x, n, p)
cdf_values = binom.cdf(x, n, p)

plt.figure(figsize=(12, 5))

plt.subplot(1, 2, 1)
plt.stem(x, pmf_values, basefmt=" ")
plt.title('PMF of 100 Coin Tosses')
plt.xlabel('Number of Heads')
plt.ylabel('Probability')

plt.subplot(1, 2, 2)
plt.plot(x, cdf_values, 'b-')
plt.title('CDF of 100 Coin Tosses')
plt.xlabel('Number of Heads')
plt.ylabel('Cumulative Probability')

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

3.5 Results

The resulting plots clearly demonstrate the Binomial distribution's characteristics:

- The PMF peaks around 50 heads, showing the most probable outcome.
- The CDF increases gradually and approaches 1 as the number of heads approaches 100.

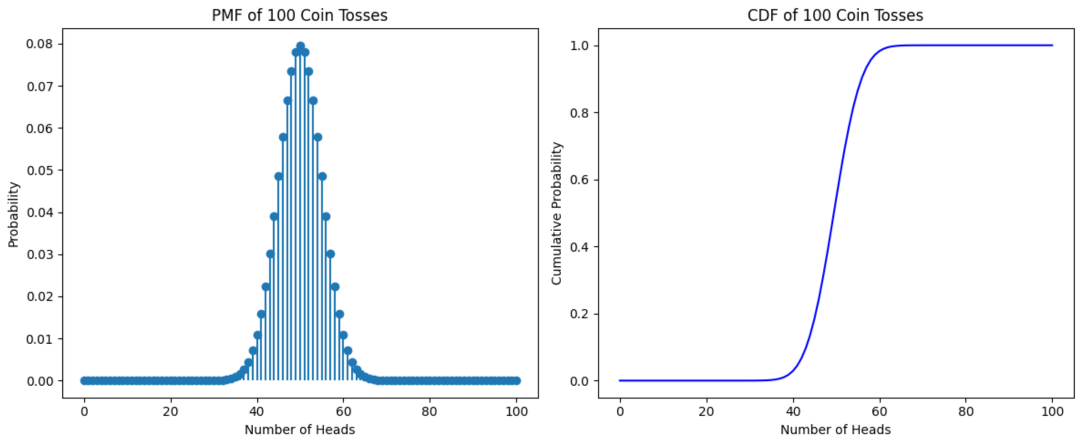


Figure 8: PMF and CDF of 100 Coin Tosses

3.6 Conclusion

This experiment used Python to show how a large Binomial distribution behaves. As expected, the probability distribution (PMF) was balanced and peaked around 50. The cumulative distribution (CDF) also increased quickly near that point. Overall, this confirms what we know in theory about the Binomial distribution and shows that we can simulate it accurately using code.

4 Experiment 4: Distribution Functions in Python

4.1 Objective

To generate and visualize random values from three different statistical distributions using Python:

- Uniform Distribution
- Normal Distribution
- Exponential Distribution

4.2 Tools Used

- Python 3
- NumPy
- Matplotlib
- SciPy (scipy.stats)

4.3 Theory

4.3.1 Uniform Distribution

A uniform distribution has constant probability. Each value within a specific interval is equally likely to occur. In `scipy.stats.uniform`, the parameters are:

- `loc`: The start of the interval.
- `scale`: The width of the interval.

4.3.2 Normal Distribution

Also known as the Gaussian distribution. It is symmetric around the mean. The parameters are:

- `loc`: The mean of the distribution.
- `scale`: The standard deviation.

4.3.3 Exponential Distribution

Models the time between events in a Poisson process. The parameter is:

- `scale`: Equal to $\frac{1}{\lambda}$ where λ is the rate.

4.4 Python Code

Import Libraries and Set Seed

```python
import numpy as np
import matplotlib.pyplot as plt
from scipy.stats import uniform, norm, expon

# Set seed for reproducibility
np.random.seed(42)
```

Uniform Distribution

```python
# Uniform Distribution: loc = start, scale = width
uniform_data = uniform.rvs(loc=0, scale=10, size=1000)

plt.figure(figsize=(10, 4))
plt.hist(uniform_data, bins=30, density=True, alpha=0.6,
         color='skyblue', edgecolor='black')
plt.title('Uniform Distribution (loc=0, scale=10)')
plt.xlabel('Value')
plt.ylabel('Density')
plt.grid(True)
plt.savefig('uniform.png')
plt.show()
```

Normal Distribution

```python
# Normal Distribution: loc = mean, scale = std deviation
normal_data = norm.rvs(loc=0, scale=1, size=1000)

plt.figure(figsize=(10, 4))
plt.hist(normal_data, bins=30, density=True, alpha=0.6,
         color='lightgreen', edgecolor='black')
plt.title('Normal Distribution (mean=0, std=1)')
plt.xlabel('Value')
plt.ylabel('Density')
plt.grid(True)
plt.savefig('normal.png')
plt.show()
```

Exponential Distribution

```python
# Exponential Distribution: scale = 1/lambda
exponential_data = expon.rvs(scale=1.0, size=1000)

plt.figure(figsize=(10, 4))
plt.hist(exponential_data, bins=30, density=True, alpha=0.6,
         color='salmon', edgecolor='black')
```

```python
plt.title('Exponential Distribution (scale=1.0)')
plt.xlabel('Value')
plt.ylabel('Density')
plt.grid(True)
plt.savefig('exponential.png')
plt.show()
```

4.5 Output Plots

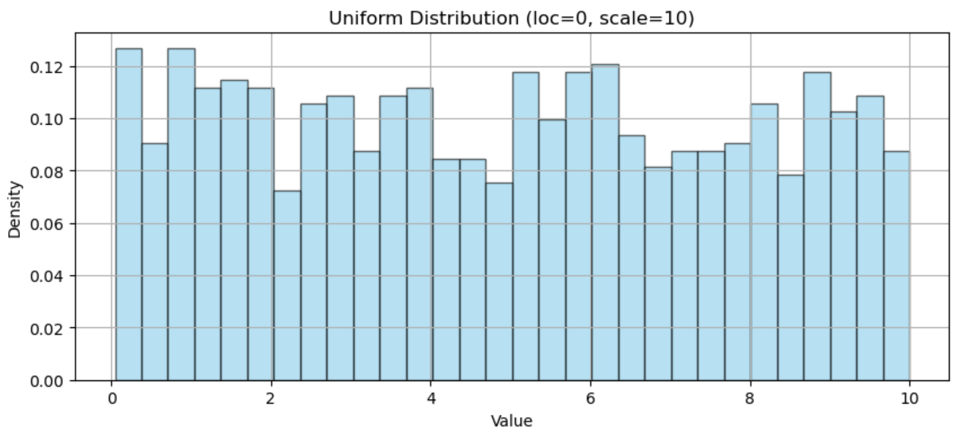


Figure 9: Uniform Distribution (loc=0, scale=10)

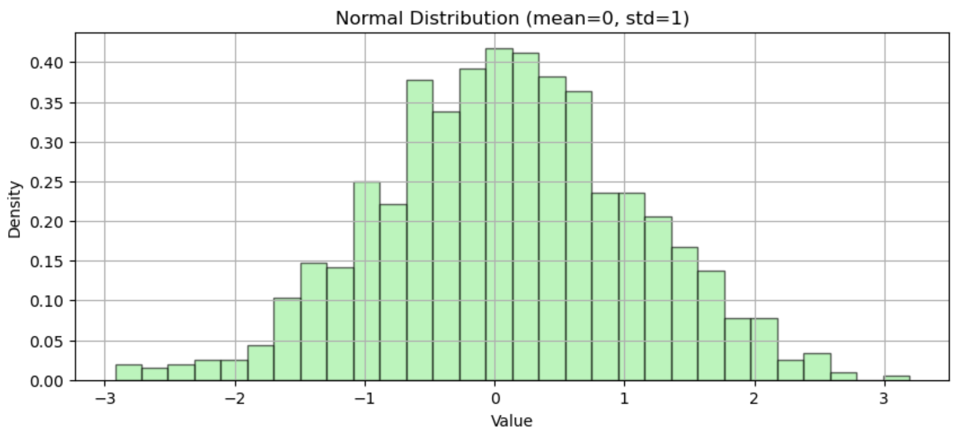


Figure 10: Normal Distribution (mean=0, std=1)

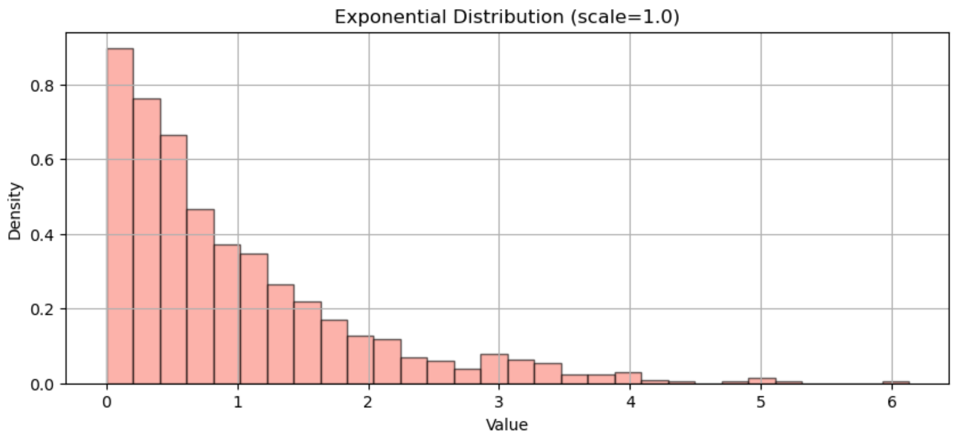


Figure 11: Exponential Distribution (scale=1.0)

4.6 Conclusion

This experiment successfully demonstrated how to generate and visualize data from uniform, normal, and exponential distributions using Python. The `scipy.stats` library offers a convenient interface to handle random variable generation and the associated parameters like `loc` and `scale`.

5 Experiment 5: Distribution of $Z = X_1 + X_2$

5.1 Objective

To study the distribution of a random variable $Z = X_1 + X_2$, where X_1 and X_2 are independently drawn from:

- Uniform distributions
- Normal distributions

and to observe the effect of different combinations of ranges and parameters on the distribution of Z using simulations in Python.

5.2 Problem Statement

Consider a distribution function Z where:

$$Z = X_1 + X_2$$

with X_1 and X_2 being random variables generated using the `rvs` function from the `scipy.stats` library.

Tasks

A) Uniform Distribution

  i) $X_1, X_2 \sim U(0, 1)$

  ii) $X_1 \sim U(0, 1), X_2 \sim U(0, 2)$

B) Normal Distribution

  i) $X_1, X_2 \sim N(0, 1)$

  ii) $X_1 \sim N(0, 1), X_2 \sim N(0, 4)$

5.3 Theory

5.3.1 Sum of Independent Random Variables

If X_1 and X_2 are independent random variables:

- For uniform distributions, the sum Z follows a triangular-like distribution.
- For normal distributions, the sum Z also follows a normal distribution with:

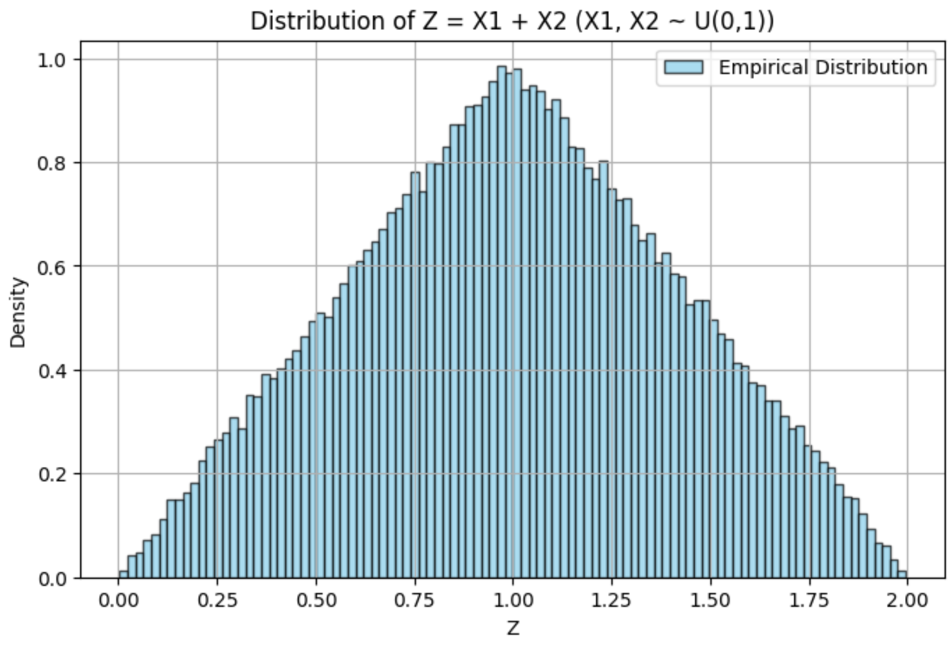
$$\mu_Z = \mu_1 + \mu_2, \quad \sigma_Z = \sqrt{\sigma_1^2 + \sigma_2^2}$$

5.4 Python Code and Results

A) Uniform Distributions

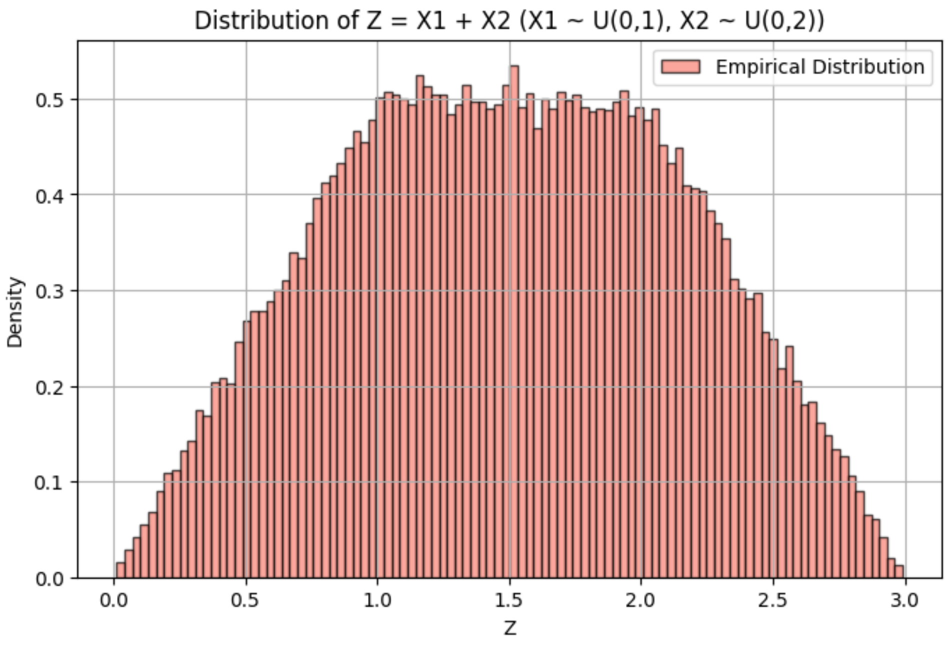
Case i: $X_1, X_2 \sim U(0, 1)$

```python
from scipy.stats import uniform
import matplotlib.pyplot as plt
import numpy as np

n = 100000
X1 = uniform.rvs(loc=0, scale=1, size=n)
X2 = uniform.rvs(loc=0, scale=1, size=n)
Z = X1 + X2

plt.hist(Z, bins=100, density=True, alpha=0.7, color='skyblue',
    ↪ edgecolor='black')
plt.title('Z = X1 + X2, X1,X2 ~ U(0,1)')
plt.xlabel('Z')
plt.ylabel('Density')
plt.grid(True)
plt.show()
```



Case ii: $X_1 \sim U(0, 1), X_2 \sim U(0, 2)$

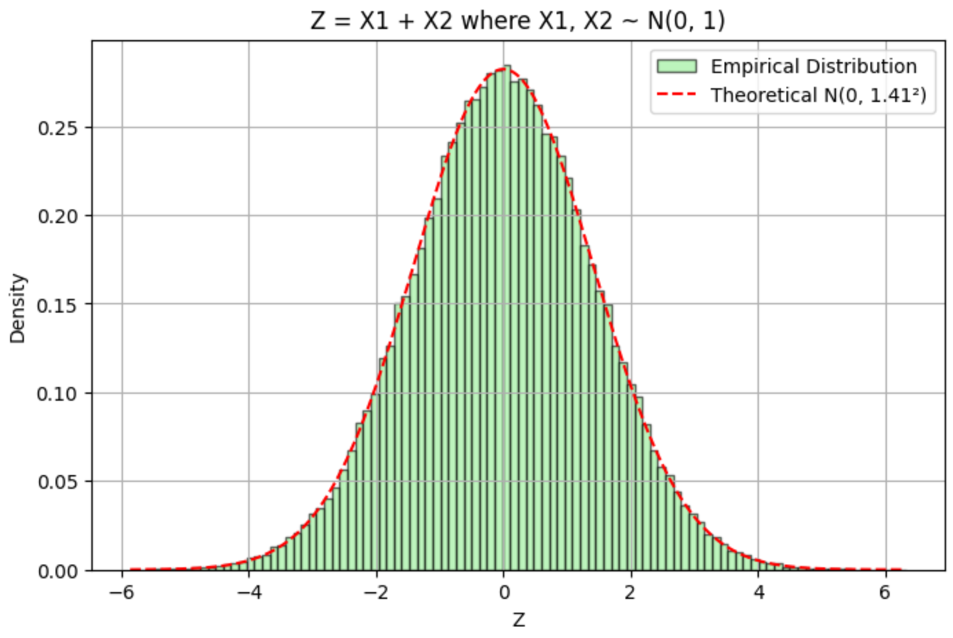
```python
X1 = uniform.rvs(loc=0, scale=1, size=n)
X2 = uniform.rvs(loc=0, scale=2, size=n)
Z = X1 + X2

plt.hist(Z, bins=100, density=True, alpha=0.7, color='salmon',
    ↪ edgecolor='black')
plt.title('Z = X1 + X2, X1 ~ U(0,1), X2 ~ U(0,2)')
plt.xlabel('Z')
plt.ylabel('Density')
plt.grid(True)
plt.show()
```



B) Normal Distributions

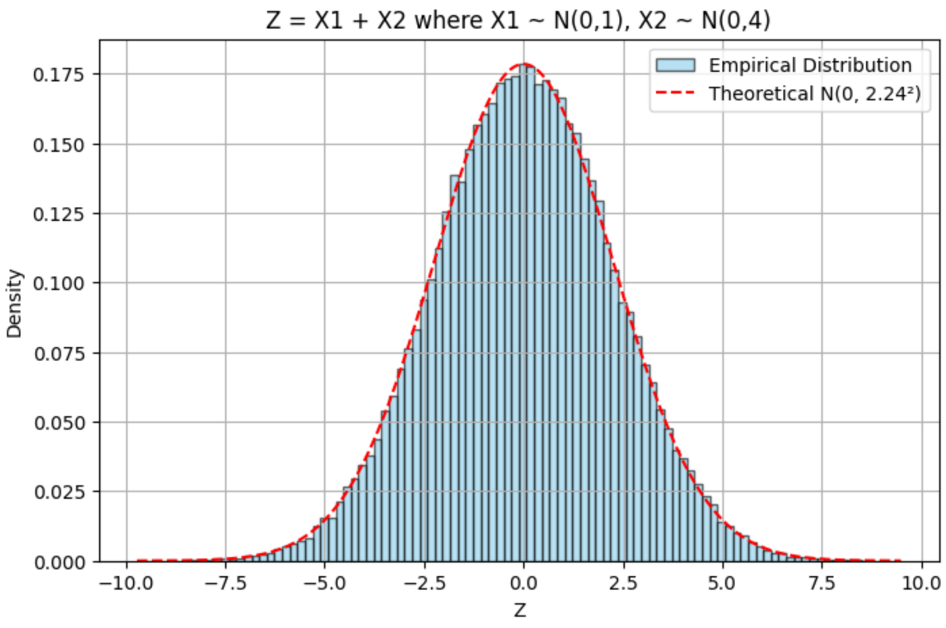
Case i: $X_1, X_2 \sim N(0,1)$

```
from scipy.stats import norm

mu1, sigma1 = 0, 1
mu2, sigma2 = 0, 1

X1 = norm.rvs(loc=mu1, scale=sigma1, size=n)
X2 = norm.rvs(loc=mu2, scale=sigma2, size=n)
Z = X1 + X2

mu_z = mu1 + mu2
sigma_z = np.sqrt(sigma1**2 + sigma2**2)

plt.hist(Z, bins=100, density=True, alpha=0.6, color='lightgreen'
    ↪ , edgecolor='black')
x = np.linspace(min(Z), max(Z), 1000)
pdf = norm.pdf(x, loc=mu_z, scale=sigma_z)
plt.plot(x, pdf, 'r--', label=f'N({mu_z}, {sigma_z:.2f}^2)')
plt.title('Z = X1 + X2, X1,X2 ~ N(0,1)')
plt.xlabel('Z')
plt.ylabel('Density')
plt.legend()
plt.grid(True)
plt.show()
```



Case ii: $X_1 \sim N(0,1), X_2 \sim N(0,4)$

```python
mu1, sigma1 = 0, 1
mu2, sigma2 = 0, 2

X1 = norm.rvs(loc=mu1, scale=sigma1, size=n)
X2 = norm.rvs(loc=mu2, scale=sigma2, size=n)
Z = X1 + X2

mu_z = mu1 + mu2
sigma_z = np.sqrt(sigma1**2 + sigma2**2)

plt.hist(Z, bins=100, density=True, alpha=0.6, color='skyblue',
    ↪ edgecolor='black')
x = np.linspace(min(Z), max(Z), 1000)
pdf = norm.pdf(x, loc=mu_z, scale=sigma_z)
plt.plot(x, pdf, 'r--', label=f'N({mu_z}, {sigma_z:.2f}^2)')
plt.title('Z = X1 + X2, X1 ~ N(0,1), X2 ~ N(0,4)')
plt.xlabel('Z')
plt.ylabel('Density')
plt.legend()
plt.grid(True)
plt.show()
```



5.5 Observations

- The sum of two uniform distributions results in a triangular or trapezoidal-like distribution.
- The sum of two normal distributions results in another normal distribution with mean and variance equal to the sum of the respective means and variances.
- In the case of mixed parameter ranges (e.g., $U(0,1) + U(0,2)$), the shape of Z's distribution becomes more spread and asymmetric.

5.6 Conclusion

This experiment demonstrates how the addition of independent random variables affects the resulting distribution. It provides a hands-on understanding of convolution of probability distributions and the central limit behavior.

6 Experiment 6: Descriptive Statistical Analysis of Diabetes Dataset

Introduction

This report presents a descriptive statistical analysis of the Diabetes dataset. The objective is to explore key statistical measures such as mean, median, and mode for numerical variables. We also visualize the distribution of each variable using histograms with Kernel Density Estimation (KDE) and indicate measures of central tendency on the plots.

Dataset Overview

The dataset contains information about patients and includes features like glucose levels, blood pressure, insulin levels, BMI, and other health-related indicators. Below is the initial snapshot of the dataset:

(First 10 rows of the dataset)

Methodology

The following steps were carried out:

1. Loaded the dataset using Python (pandas).
2. Identified numeric columns for analysis.
3. Calculated:
   - **Mean** – The average value of each column.
   - **Median** – The middle value when data is sorted.
   - **Mode** – The most frequently occurring value.
4. Created a summary table showing mean, median, and all modes.
5. Visualized the distribution of each numeric column using histograms with KDE curves.
6. Added vertical lines for Mean (red), Median (green), and Mode (purple).

Statistical Summary

Table 1 shows the computed mean, median, and mode values for each numeric feature.

Table 1: Summary Statistics of Numeric Columns

Feature	**Mean**	**Median**	**Mode(s)**
Pregnancies	3.845052	3	1
Glucose	120.894531	117	99
BloodPressure	69.105469	72	70

Continued on next page

Feature	Mean	Median	Mode(s)
SkinThickness	20.536458	23	0
Insulin	79.799479	30.5	0.0
BMI	31.992578	32	32
DiabetesPedigreeFunction	0.471876	0.3725	0.254
Age	33.240885	29	22
Outcome	0.348958	0	0

Distribution Plots

The figures below show the distribution of each numeric variable in the dataset. Each histogram includes KDE for smooth visualization. Vertical lines indicate:

- **Red dashed line:** Mean
- **Green dash-dot line:** Median
- **Purple dotted line:** Mode

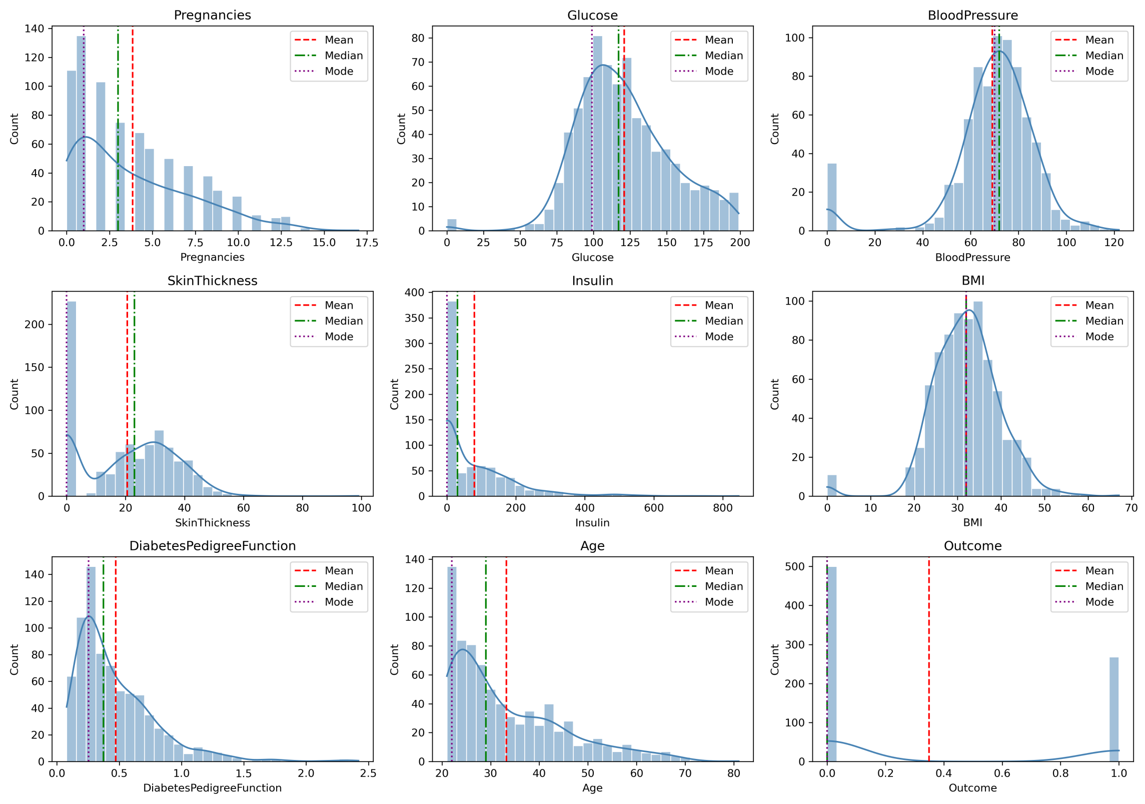


Figure 12: Distribution of Numeric Features with Mean, Median, and Mode Indicators

Observations

- Most features show slight skewness, particularly Insulin and SkinThickness, indicating non-normal distributions.
- Glucose and Age distributions appear closer to normal.

- The mean and median are close for most variables, suggesting mild skewness in some cases.

Conclusion

This analysis provided insights into the central tendency and distribution of each numeric variable in the Diabetes dataset. These findings are useful for further steps such as feature engineering and model building.

7 Experiment 7: Analysis of Variability and Distribution in Diabetes Dataset

Introduction

The purpose of this report is to analyze a diabetes dataset by calculating various measures of variability and understanding the distribution of numerical features. We will compute:

- Variance
- Coefficient of Variation (CV)
- Interquartile Range (IQR)
- Skewness

Additionally, visualizations such as boxplots and distribution plots will be used to interpret the data characteristics.

Dataset

The dataset used in this analysis is `diabetes.csv`. It contains several numerical features related to health indicators such as glucose level, blood pressure, BMI, etc.

Methodology

The following steps were followed:

1. Load the dataset using Pandas.
2. Identify numeric columns.
3. Compute measures of variability:
    - Variance: Measure of spread around the mean.
    - Coefficient of Variation (CV): Relative variability, expressed as a percentage.
    - Interquartile Range (IQR): Difference between 75th and 25th percentiles.
    - Skewness: Indicates the asymmetry of the distribution.
4. Plot:
    - Boxplots for all numeric features.
    - Histogram with KDE for each numeric column along with skewness value.

Measures of Variability

The calculated measures are shown in Table 6.

Table 2: Measures of Variability for Numeric Features

Feature	Variance	CV (%)	IQR	Skewness
Pregnancies	11.354056	87.634133	5	0.899912
Glucose	1022.248314	26.446703	41.2500	0.173414
BloodPressure	374.647271	28.009082	18.00	-1.840005
SkinThickness	254.473245	77.677549	32.00	0.109159
Insulin	13281.180078	144.416986	127.2500	2.267810
BMI	62.159984	24.643717	9.30	-0.428143
DiabetesPedigree	0.109779	70.215138	0.38	1.916159
Age	138.303046	35.378816	17.00	1.127389
Outcome	0.227483	136.678604	1.00	0.633776

Visualizations

Boxplots

The boxplot in Figure 13 shows the spread and outliers for all numeric features.

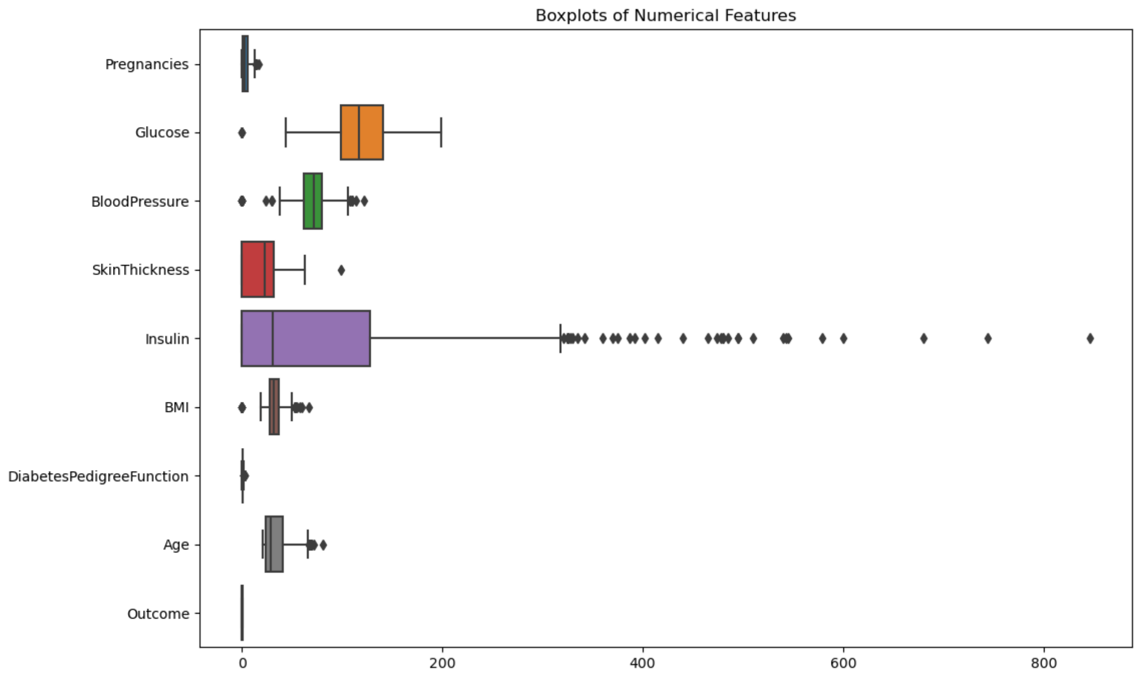
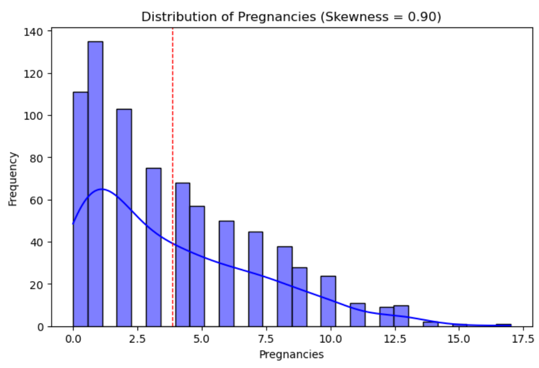


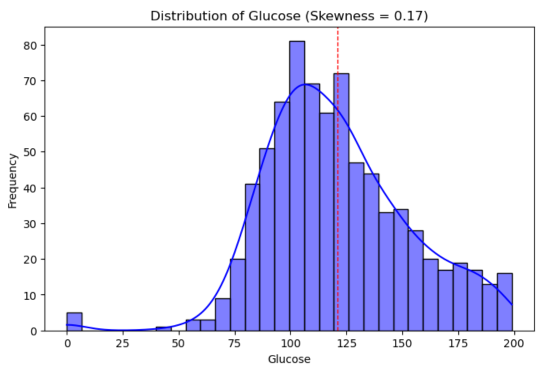
Figure 13: Boxplots of All Numeric Features

Distribution Plots with Skewness

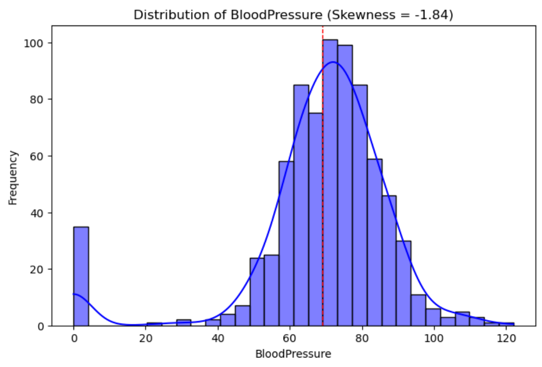
Each numeric feature is visualized using a histogram with KDE. The skewness value is displayed in the title.



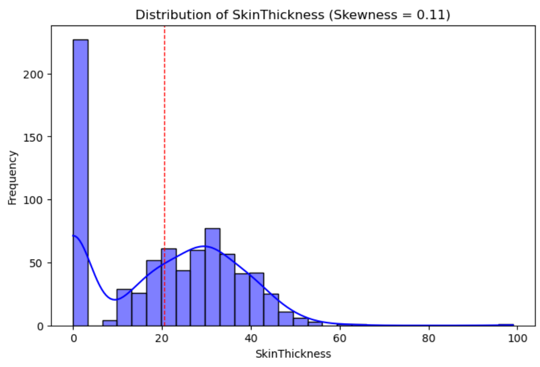
(a) Pregnancies



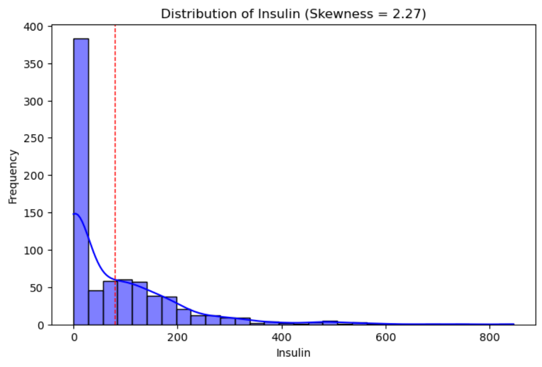
(b) Glucose



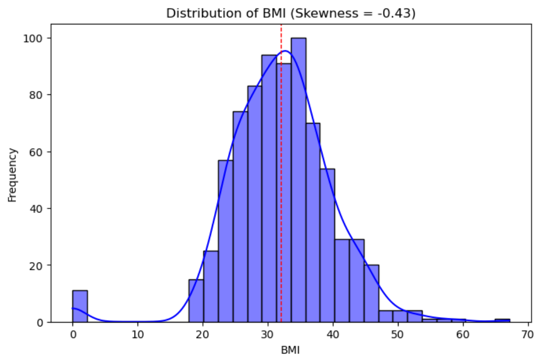
(c) Blood Pressure



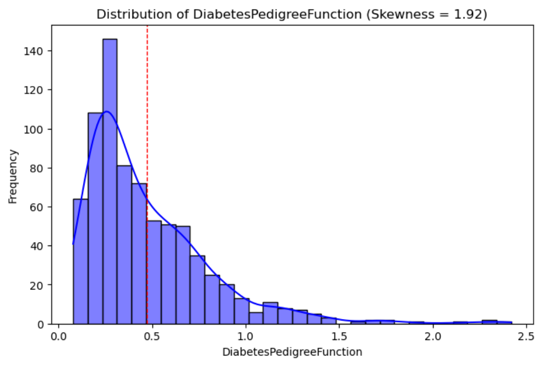
(d) Skin Thickness



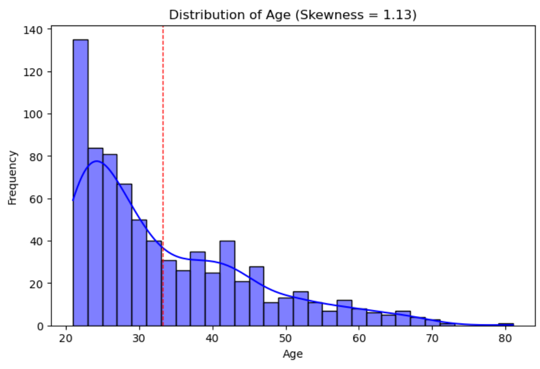
(e) Insulin



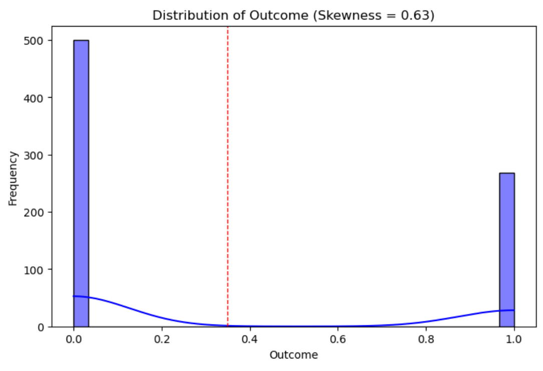
(f) BMI



(g) Diabetes Pedigree



(h) Age



Conclusion

- Variance and IQR show which features have a wider spread (e.g., Insulin has a large variance).
- Features like Insulin and Diabetes Pedigree have high positive skewness, indicating a long tail to the right.
- Some variables, such as Blood Pressure, show negative skewness.
- Outliers are evident in features like Insulin and Skin Thickness.

This analysis helps us understand data variability and shape, which is essential for preprocessing and modeling.

8 Experiment 8: Analysis of Skewness and Box-Cox Transformation on Diabetes Dataset

Introduction

This experiment focuses on analyzing skewness and normalizing the data in the diabetes dataset using the Box-Cox transformation. The main objectives are:

- Calculate Pearson's coefficient of skewness for each numerical column.
- Apply the Box-Cox transformation to normalize the data.
- Visualize the distribution of data before and after the transformation.

Dataset Description

The dataset used for this analysis is `diabetes.csv`, which contains various health-related attributes such as glucose level, blood pressure, BMI, and more. All features used in this analysis are numerical.

Step 1: Loading the Dataset

The dataset was loaded into a pandas DataFrame for analysis:

```python
import pandas as pd
data = pd.read_csv("diabetes.csv")
```

Listing 12: Loading the dataset

Step 2: Pearson's Coefficient of Skewness

Skewness measures the asymmetry of the data distribution. Pearson's coefficient of skewness is calculated using the formula:

$$\text{Skewness} = \frac{3(\text{Mean} - \text{Median})}{\text{Standard Deviation}}$$

A positive skew indicates a distribution with a longer tail on the right, while a negative skew indicates a longer tail on the left.

Table 3: Pearson's Coefficient of Skewness

Feature	**Skewness**
Pregnancies	0.752366
Glucose	0.365425
BloodPressure	-0.448630
SkinThickness	-0.463298
Insulin	1.283350
BMI	-0.002824
DiabetesPedigreeFunction	0.899798
Age	1.081837
Outcome	2.194930

Step 3: Box-Cox Transformation

To reduce skewness and make data closer to a normal distribution, we applied the Box-Cox transformation. The transformation is defined as:

$$y(\lambda) = \begin{cases} \frac{y^\lambda - 1}{\lambda}, & \text{if } \lambda \neq 0 \\ \ln(y), & \text{if } \lambda = 0 \end{cases}$$

Box-Cox requires strictly positive values, so all features were shifted if necessary.

Visualization of Transformation

Figures below show distributions before and after the Box-Cox transformation for different features.

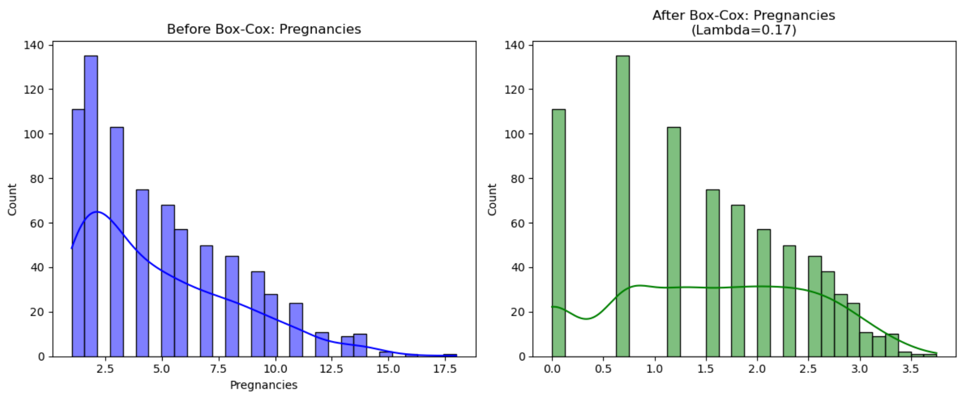


Figure 15: Distribution before and after Box-Cox Transformation for Pregnancies.

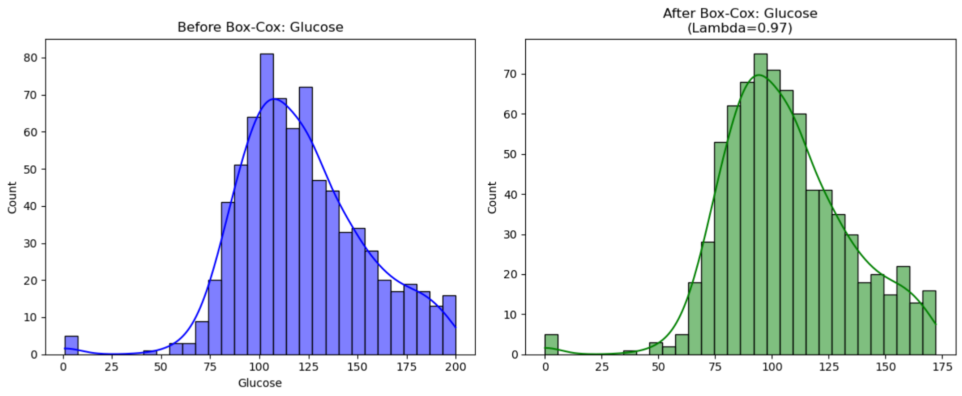


Figure 16: Distribution before and after Box-Cox Transformation for Glucose.

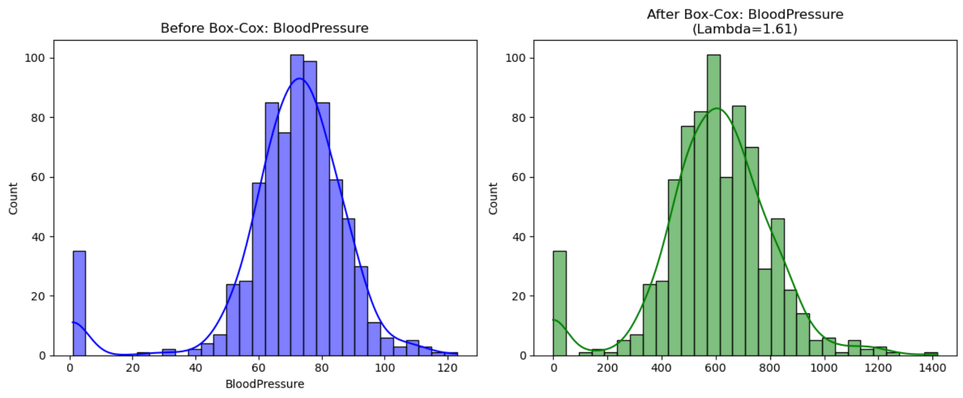


Figure 17: Distribution before and after Box-Cox Transformation for Blood Pressure.

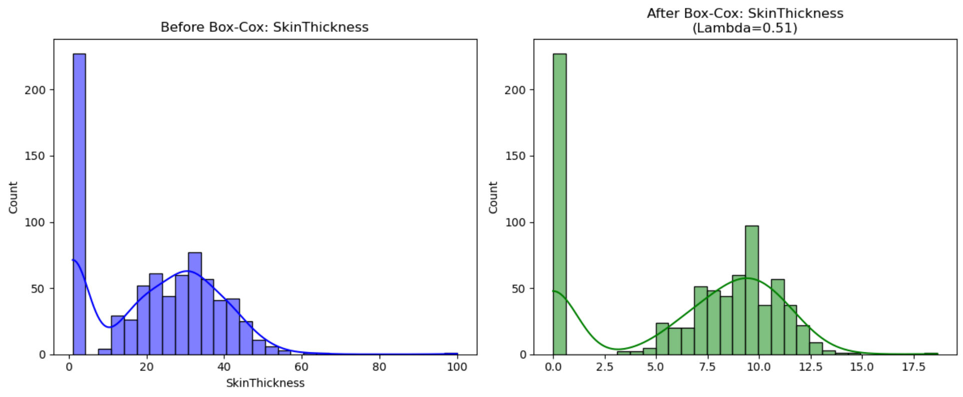


Figure 18: Distribution before and after Box-Cox Transformation for SkinThickness.

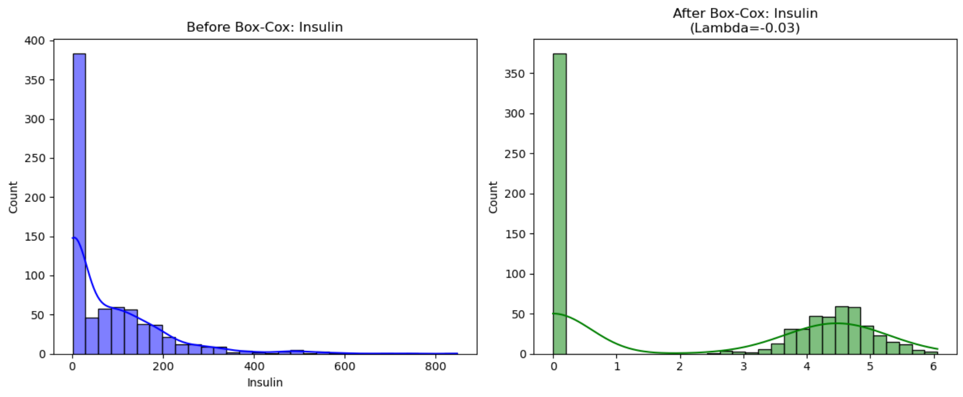


Figure 19: Distribution before and after Box-Cox Transformation for Insulin.

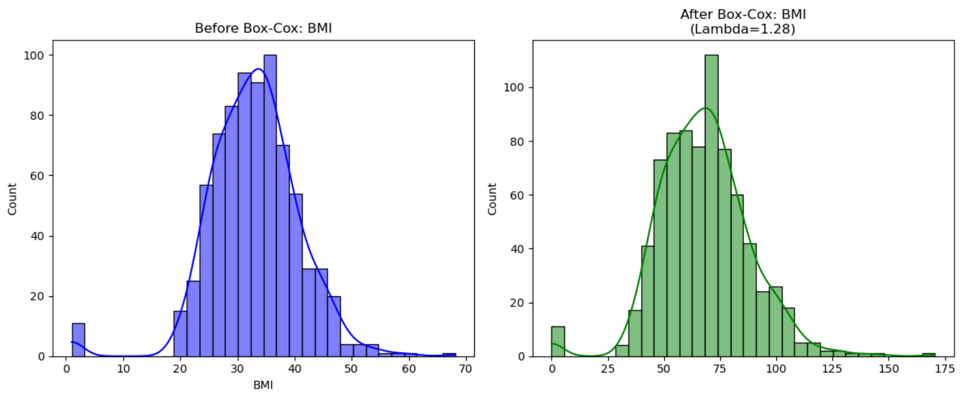


Figure 20: Distribution before and after Box-Cox Transformation for BMI.

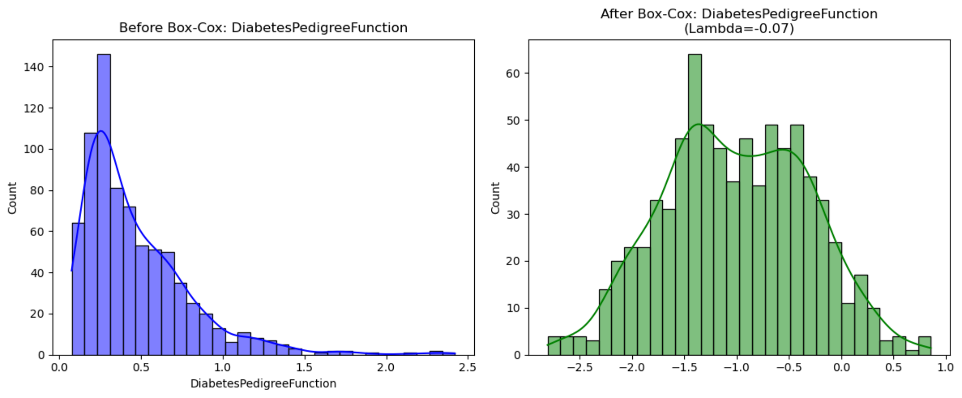


Figure 21: Distribution before and after Box-Cox Transformation for Diabetes Pedigree Function.

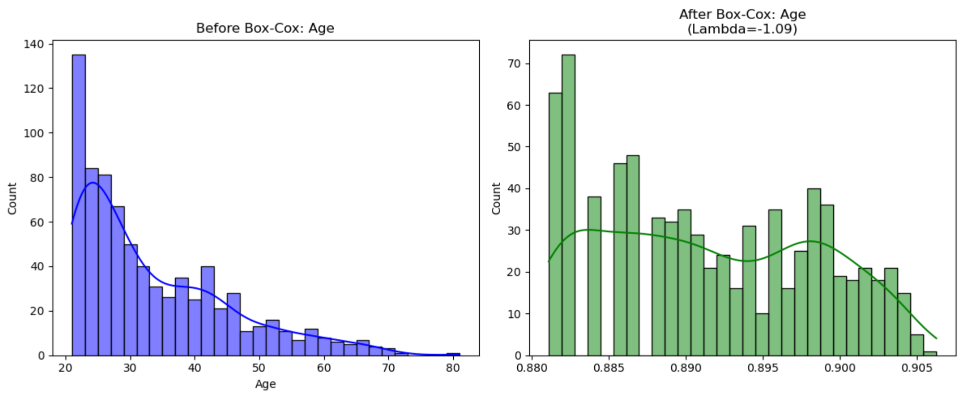


Figure 22: Distribution before and after Box-Cox Transformation for Age.

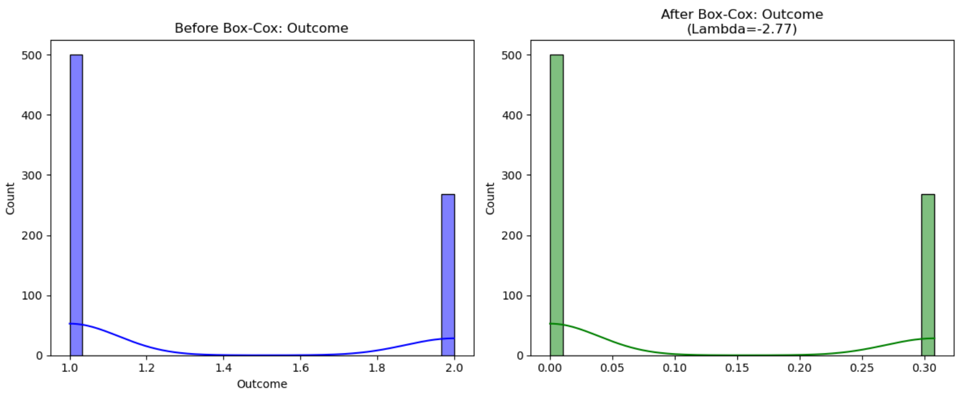


Figure 23: Distribution before and after Box-Cox Transformation for Outcome.

Lambda Values

Table 4: Box-Cox Lambda Values

Feature	**Lambda**
Pregnancies	0.172724
Glucose	0.966405
BloodPressure	1.606631
SkinThickness	0.511566
Insulin	-0.032285
BMI	1.276566
DiabetesPedigreeFunction	-0.073108
Age	-1.094423
Outcome	-2.771665

Conclusion

The analysis revealed that many features in the diabetes dataset exhibited skewness. Applying the Box-Cox transformation helped to normalize the distributions, which is essential for statistical modeling and machine learning algorithms that assume normality.

9 Experiment 9: Descriptive Statistical Analysis Report Diabetes Dataset

22 July, 2025

9.1 Abstract

This report provides a comprehensive descriptive statistical analysis of the `diabetes.csv` dataset. It covers distribution analysis, measures of central tendency, variability, skewness, kurtosis, and visual interpretations using histograms, KDE plots, and boxplots.

9.2 Introduction

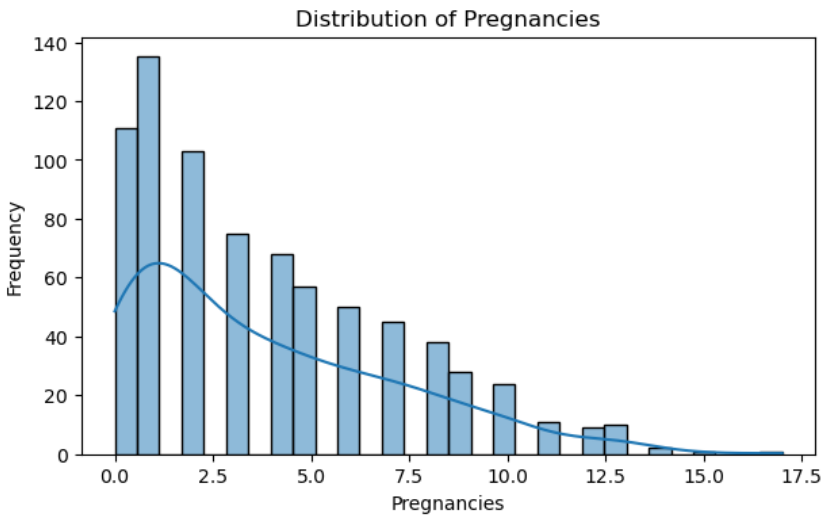
Understanding the distribution and characteristics of data is essential before applying machine learning models. This analysis aims to summarize key properties of each feature in the dataset, detect skewness, outliers, and identify the overall data distribution.

9.3 Distribution Analysis

The following figures show the distribution of each feature along with KDE curves. We performed the Shapiro-Wilk normality test and calculated skewness to determine whether the distribution is normal, left-skewed, or right-skewed.

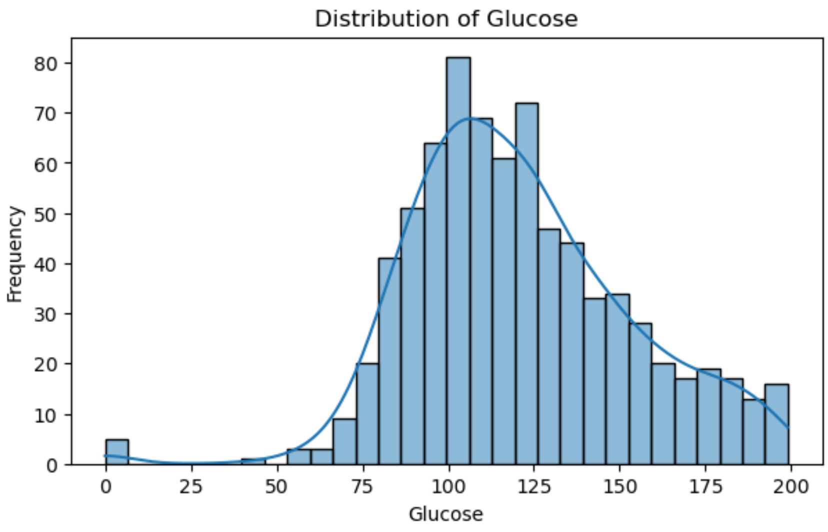
Feature: Pregnancies

Shapiro-Wilk p-value: 0.0000
Skewness: 0.8999
Distribution Type: Right-Skewed



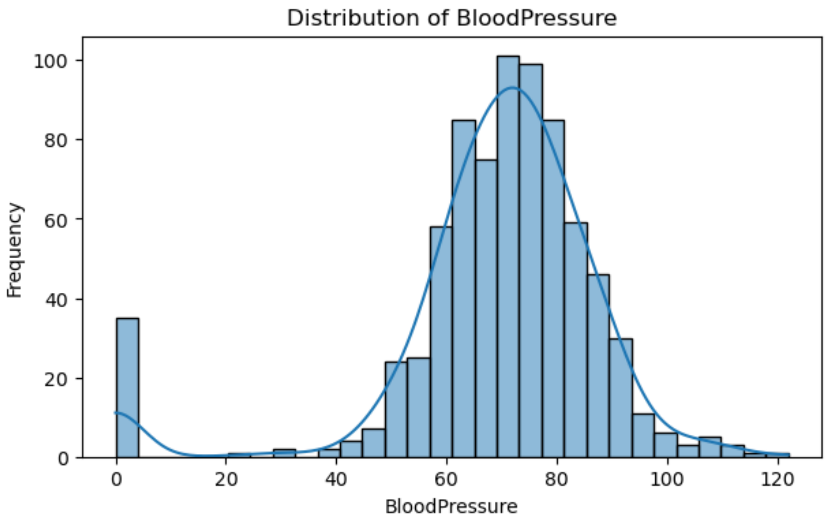
Feature: Glucose

Shapiro-Wilk p-value: 0.0000
Skewness: 0.1734
Distribution Type: Approximately Symmetric (Non-normal)



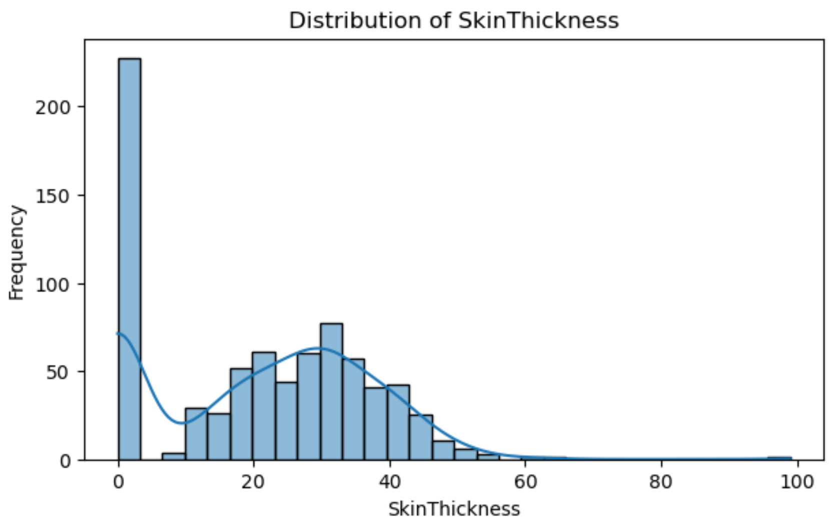
Feature: BloodPressure

Shapiro-Wilk p-value: 0.0000
Skewness: -1.8400
Distribution Type: Left-Skewed



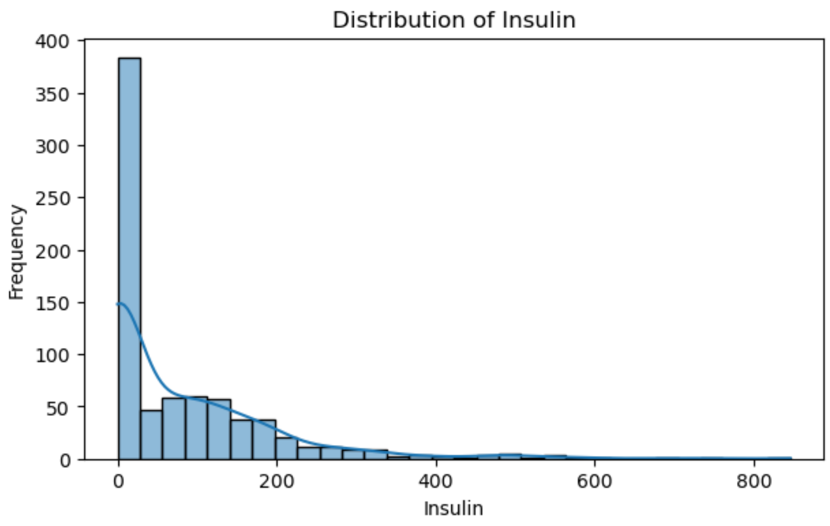
Feature: SkinThickness

Shapiro-Wilk p-value: 0.0000
Skewness: 0.1092
Distribution Type: Approximately Symmetric (Non-normal)



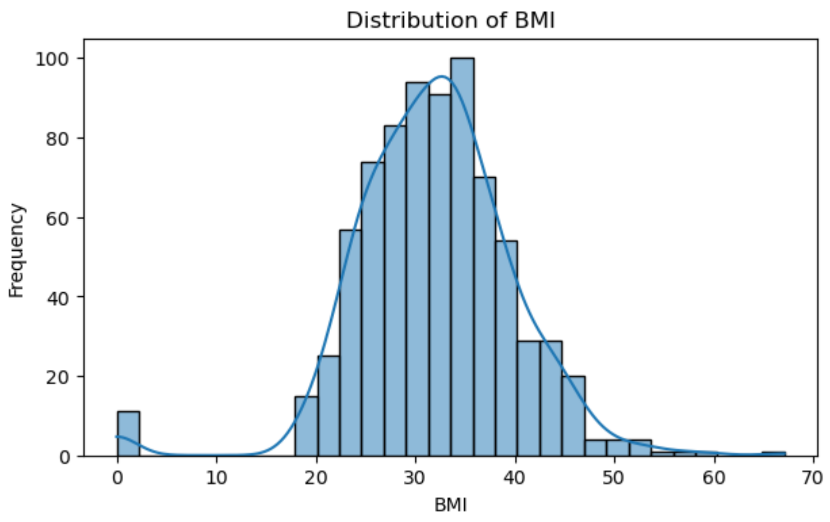
Feature: Insulin

Shapiro-Wilk p-value: 0.0000
Skewness: 2.2678
Distribution Type: Right-Skewed



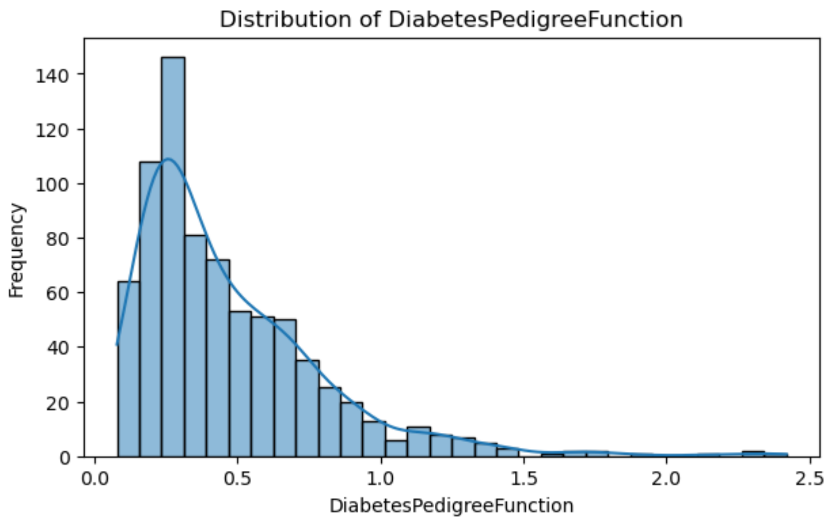
Feature: BMI

Shapiro-Wilk p-value: 0.0000
Skewness: -0.4281
Distribution Type: Approximately Symmetric (Non-normal)



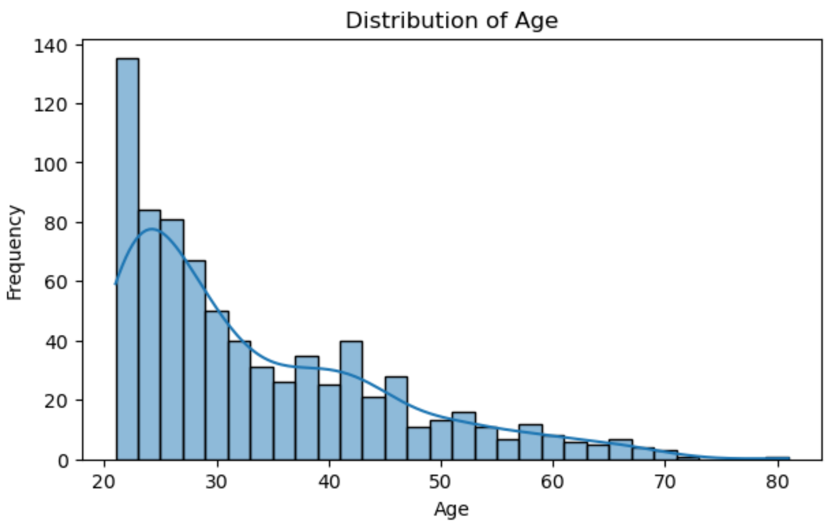
Feature: DiabetesPedigreeFunction

Shapiro-Wilk p-value: 0.0000
Skewness: 1.9162
Distribution Type: Right-Skewed



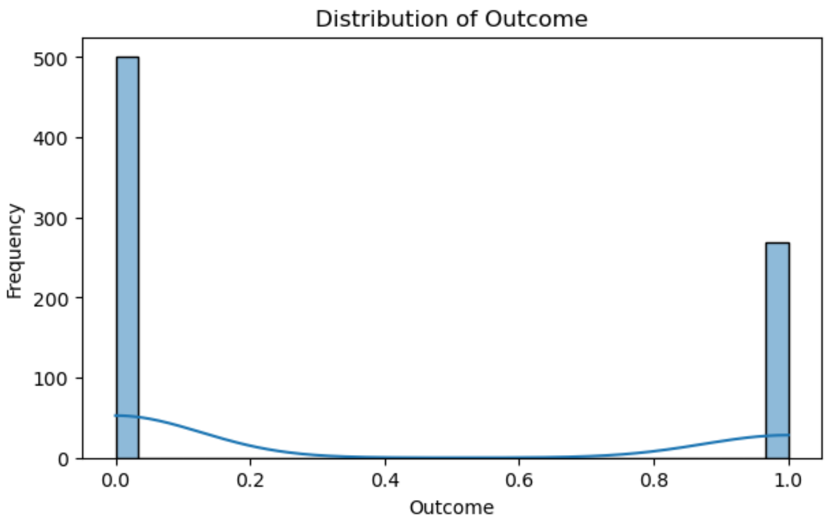
Feature: Age

Shapiro-Wilk p-value: 0.0000
Skewness: 1.1274
Distribution Type: Right-Skewed



Feature: Outcome

Shapiro-Wilk p-value: 0.0000
Skewness: 0.6338
Distribution Type: Right-Skewed



9.4 Central Tendency Measures

Table 5 shows the mean, median, and mode for each feature.

Table 5: Central Tendency Measures

Feature	Mean	Median	Mode
Pregnancies	3.85	3.00	1.00
Glucose	120.89	117.00	99.00
BloodPressure	69.11	72.00	70.00
SkinThickness	20.54	23.00	0.00

Feature	Mean	Median	Mode
Insulin	79.80	30.50	0.00
BMI	31.99	32.00	32.00
DiabetesPedigreeFunction	0.47	0.37	0.25
Age	33.24	29.00	22.00
Outcome	0.35	0.00	0.00

9.5 Measures of Variability

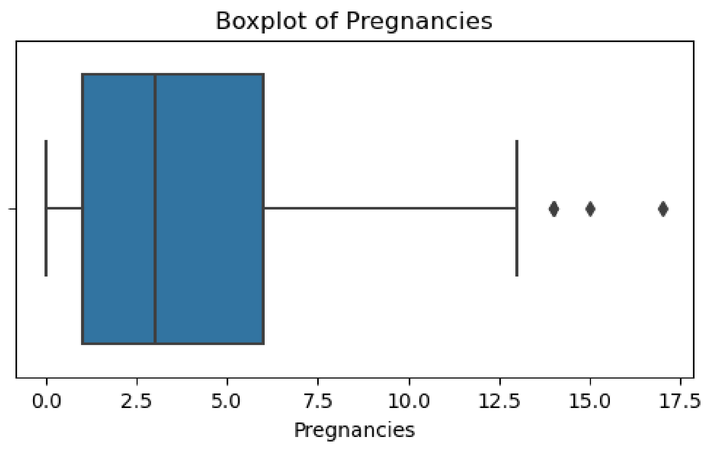
Table 6 summarizes variance, standard deviation, coefficient of variation (CV), interquartile range (IQR), and skewness for each feature.

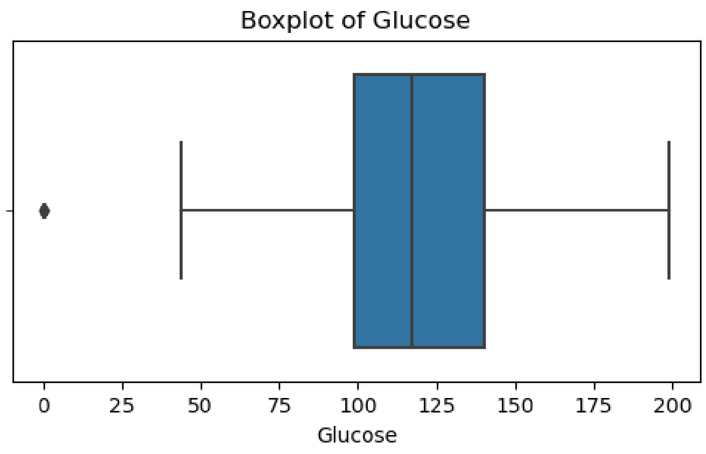
Table 6: Measures of Variability

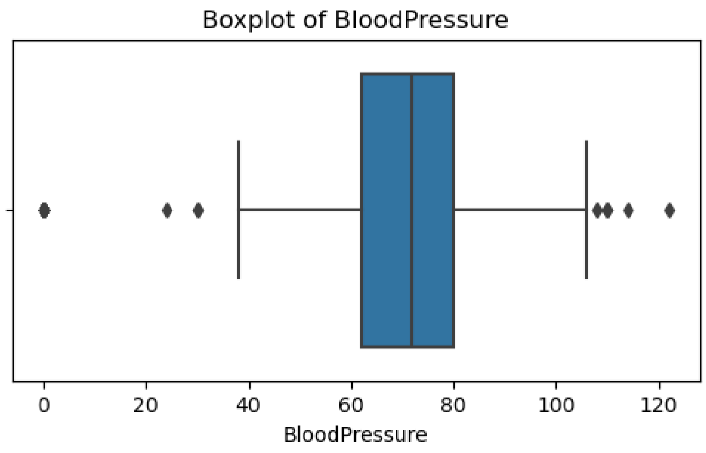
Feature	Variance	Std Dev	CV (%)	IQR	Skewness
Pregnancies	11.35	3.37	87.63	5.00	0.90
Glucose	1022.25	31.97	26.45	41.25	0.17
BloodPressure	374.65	19.36	28.01	18.00	-1.84
SkinThickness	254.47	15.95	77.68	32.00	0.11
Insulin	13281.18	115.24	144.42	127.25	2.27
BMI	62.16	7.88	24.64	9.30	-0.43
DiabetesPedigreeFunction	0.11	0.33	70.22	0.38	1.92
Age	138.30	11.76	35.38	17.00	1.13
Outcome	0.23	0.48	136.68	1.00	0.63

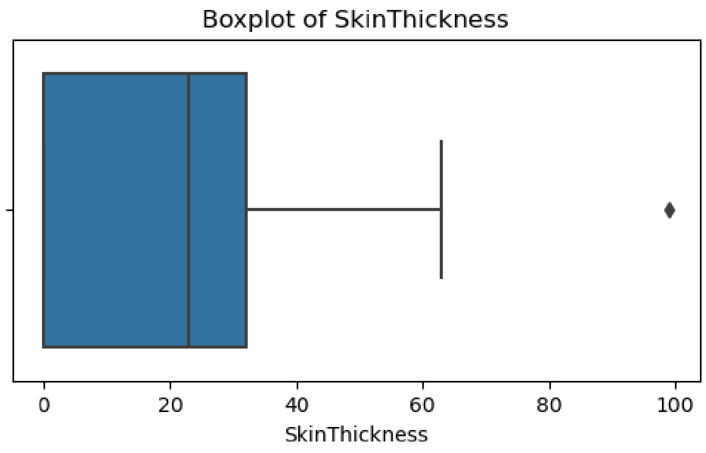
Boxplots for Variability and Outliers

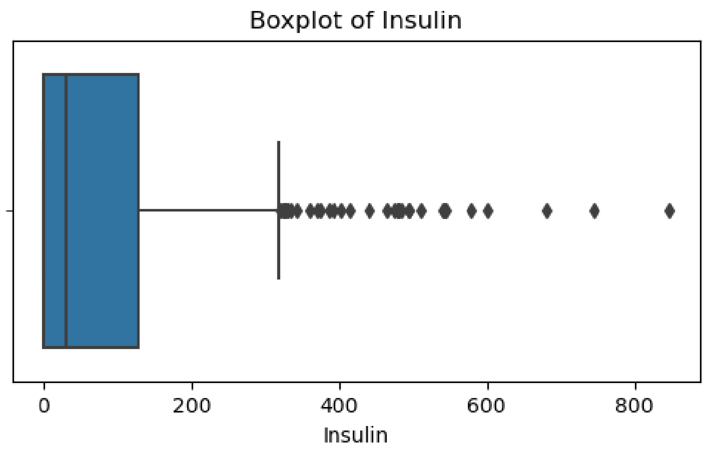
The following figures illustrate the spread and presence of outliers in each feature.

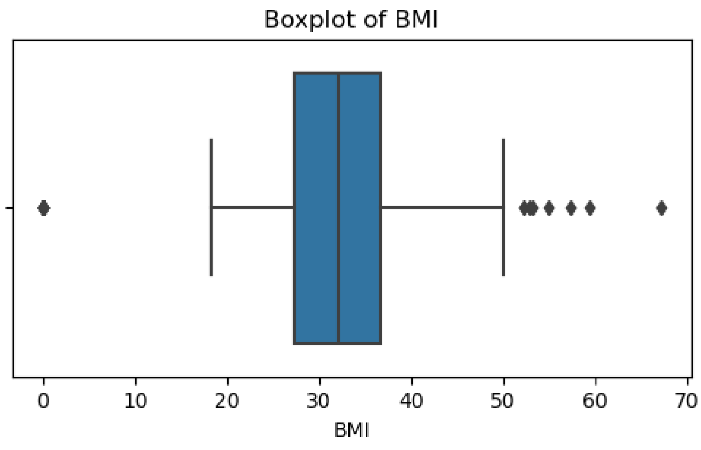


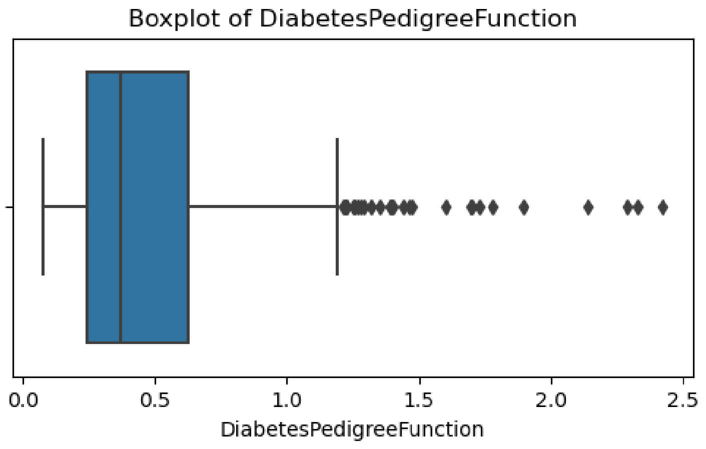

Boxplot of Glucose
0
25
50
75
100
125
150
175
200
Glucose

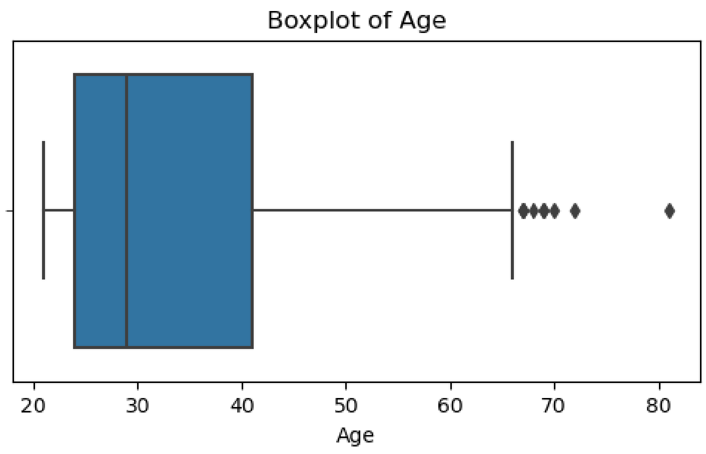

Boxplot of BloodPressure
0
20
40
60
80
100
120
BloodPressure

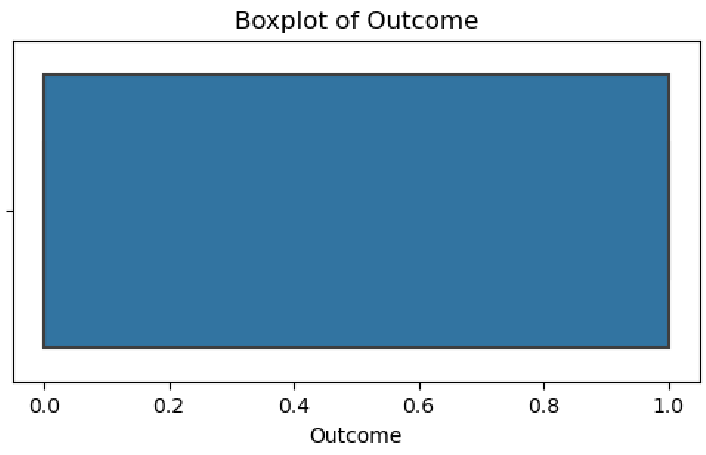

Boxplot of SkinThickness
0
20
40
60
80
100
SkinThickness


Boxplot of Insulin
0
200
400
600
800
Insulin


Boxplot of BMI
0
10
20
30
40
50
60
70
BMI


Boxplot of DiabetesPedigreeFunction
0.0
0.5
1.0
1.5
2.0
2.5
DiabetesPedigreeFunction





9.6 Kurtosis Analysis

Table 7 shows the kurtosis value for each feature and its classification.

Table 7: Kurtosis Analysis

Feature	Kurtosis	Type
Pregnancies	0.15	Leptokurtic
Glucose	0.63	Leptokurtic
BloodPressure	5.14	Leptokurtic
SkinThickness	-0.52	Platykurtic
Insulin	7.16	Leptokurtic
BMI	3.26	Leptokurtic
DiabetesPedigreeFunction	5.55	Leptokurtic
Age	0.63	Leptokurtic
Outcome	-1.60	Platykurtic

9.7 Summary and Recommendations

- Most features are not normally distributed; several exhibit right-skewness.

- High kurtosis in some features indicates heavy tails and potential outliers.
- Features such as Insulin and SkinThickness show large variability.
- Before modeling, apply transformations (e.g., log, Box-Cox) or normalization to reduce skewness.
- Handle outliers using robust methods or trimming.

10 Experiment 10: Gaussian distribution of Height, Weight of students

1 September, 2025

Introduction

In this report, we analyze simulated student data consisting of **height (cm)** and **weight (kg)**. We assume that both variables follow a joint Gaussian distribution with a given mean and covariance. Our goal is to visualize the data distribution using both one-dimensional and three-dimensional plots.

We will cover:

- One-dimensional Gaussian fits for height and weight.
- A 3D Gaussian surface plot.
- A 3D bar plot showing the density distribution of student data.

One-Dimensional Gaussian Distribution

We first fit Gaussian curves to the height and weight distributions separately. This helps us understand how well the data follows a normal distribution.

Height Distribution

We plotted the histogram of student heights and overlaid a Gaussian curve fitted using the calculated mean and standard deviation. The fitted values are:

$$\mu_{height} = 169.46\,\text{cm}, \quad \sigma_{height} = 9.45\,\text{cm}$$

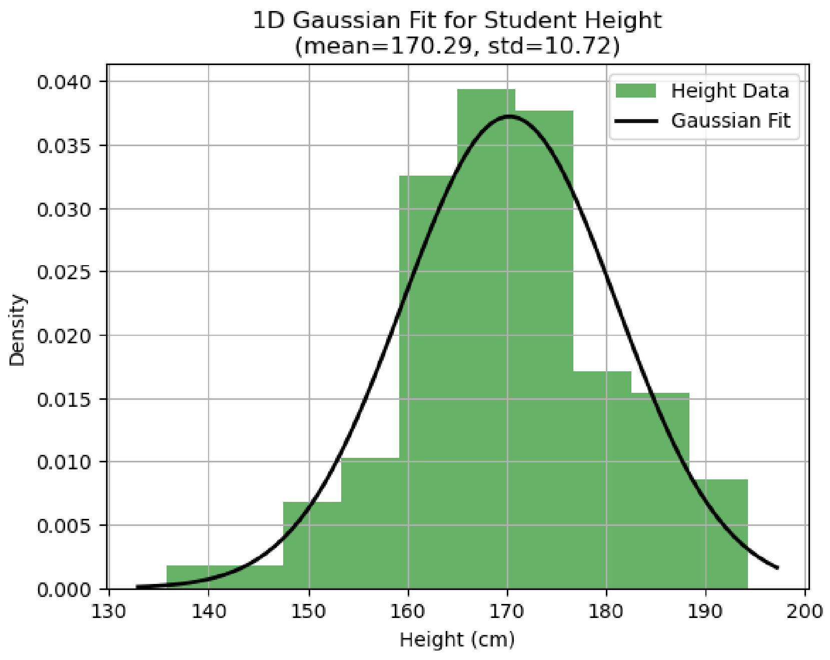


Figure 24: 1D Gaussian Fit for Student Height

Weight Distribution

Similarly, we plotted the histogram of student weights and fitted a Gaussian curve. The fitted values are:

$$\mu_{weight} = 63.20\,\text{kg}, \quad \sigma_{weight} = 12.79\,\text{kg}$$

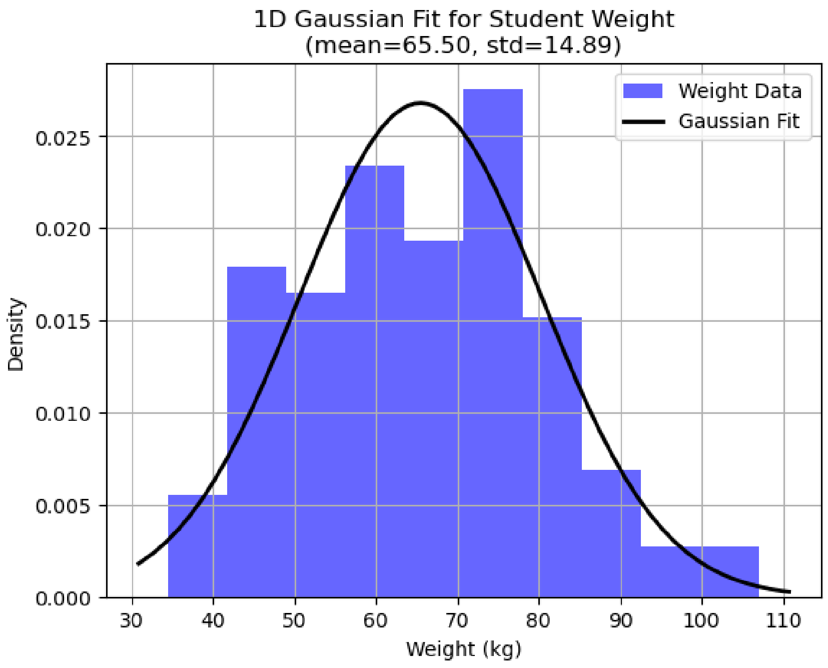


Figure 25: 1D Gaussian Fit for Student Weight

Two-Dimensional Gaussian Distribution

Height and weight are not completely independent. To study their joint distribution, we created a 2D Gaussian representation and visualized it using surface and bar plots.

3D Gaussian Surface Plot

The 3D surface plot shows the probability density of students' height and weight. The peak of the surface indicates the most likely combinations of height and weight.

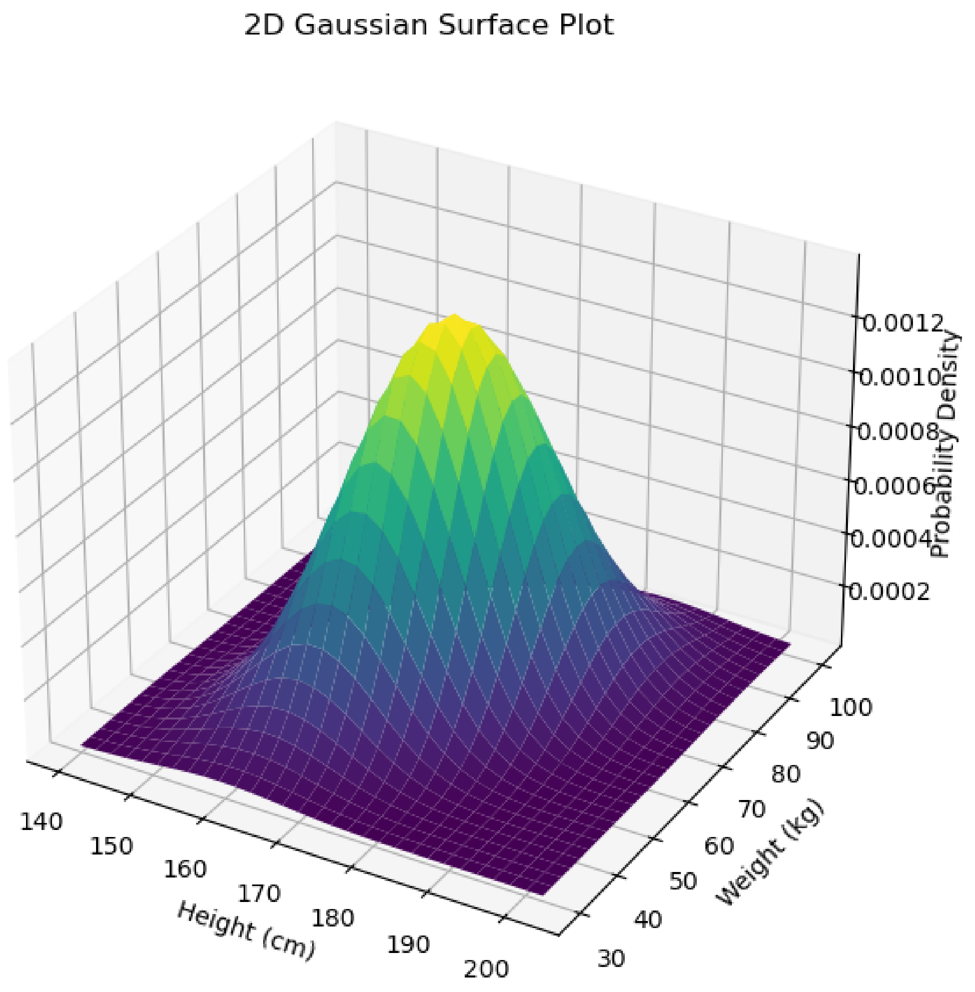


Figure 26: 2D Gaussian Surface Plot for Height and Weight

3D Bar Plot

To complement the surface plot, we also created a 3D bar plot. Here, each bar represents the density of students within a particular height–weight range. This provides a more discrete view of the distribution.

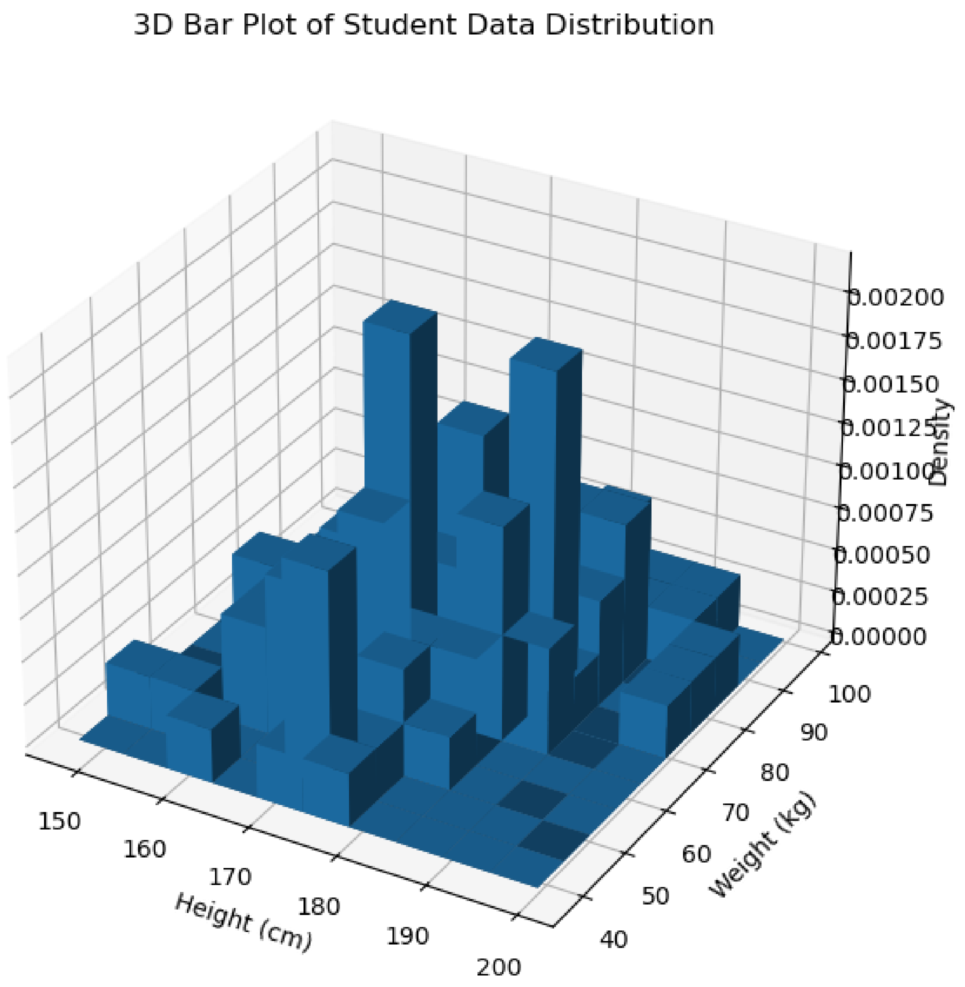


Figure 27: 3D Bar Plot of Student Data Distribution

Conclusion

From our analysis:

- Student height follows a Gaussian distribution with mean 169.46 cm and standard deviation 9.45 cm.
- Student weight follows a Gaussian distribution with mean 63.20 kg and standard deviation 12.79 kg.
- The joint distribution reveals correlations between height and weight.
- The 3D plots give a clear visualization of how students' height and weight are spread across the population.

Overall, Gaussian models provide a good fit for the data and help us better understand the underlying structure.

11 Experiment 11: Covariance Matrix Analysis on the Diabetes Dataset

15 September, 2025

Overview

This experiment computes and explains the **covariance matrix** for the Diabetes dataset. Covariance tells us how two variables change together. A positive value means they move in the same direction; a negative value means they move in opposite directions. We keep the language simple and focus on what the numbers suggest.

Aim

- Load the Diabetes dataset.
- Compute the covariance matrix across all numeric features.
- Explain the main relationships in simple terms.

Dataset

- **File:** `diabetes.csv`
- **Features:** Pregnancies, Glucose, BloodPressure, SkinThickness, Insulin, BMI, DiabetesPedigreeFunction (DPF), Age, Outcome.
- **Outcome:** 1 = diabetic, 0 = non-diabetic.

Tools Used

- Python 3 (Pandas)

Method

1. Read the CSV into a Pandas DataFrame.
2. Use `DataFrame.cov()` to compute covariance.
3. Review and interpret the largest positive/negative values.

Python Code

The code below can be run as-is (paths may need adjustment). It prints the covariance matrix and also saves it to a CSV for reference.

```python
import pandas as pd

# 1) Load the dataset (adjust path if needed)
df_diabetes = pd.read_csv('diabetes.csv')

# 2) Compute the covariance matrix
covariance_matrix = df_diabetes.cov(numeric_only=True)
```

```python

# 3) Print the result
print("Covariance matrix:")
print(covariance_matrix)

# 4) (Optional) Save to CSV
covariance_matrix.to_csv('covariance_matrix.csv', index=True)
```

Listing 13: Compute covariance matrix for the Diabetes dataset

Results

The table below shows the covariance matrix computed for this dataset.

Table 8: Covariance Matrix of Diabetes Dataset

Feature	Pregnancies	Glucose	BloodPressure	SkinThickness	Insulin	BMI
Pregnancies	11.3541	13.9471	9.2145	-4.3900	-28.5552	0.4698
Glucose	13.9471	1022.2483	94.4310	29.2392	1220.9358	55.7270
BloodPressure	9.2145	94.4310	374.6473	64.0294	198.3784	43.0047
SkinThickness	-4.3900	29.2392	64.0294	254.4732	802.9799	49.3739
Insulin	-28.5552	1220.9358	198.3784	802.9799	13281.1801	179.7752
BMI	0.4698	55.7270	43.0047	49.3739	179.7752	62.1600
DPF	-0.0374	1.4549	0.2646	0.9721	7.0667	0.3674
Age	21.5706	99.0828	54.5235	-21.3810	-57.1433	3.3603
Outcome	0.3566	7.1151	0.6007	0.5687	7.1757	1.1006

Interpretation (Simple English)

- **Glucose and Insulin** have a strong positive covariance (≈ 1220.94). When glucose goes up, insulin tends to go up too.
- **SkinThickness and Insulin** also rise together (≈ 802.98).
- **Age and Pregnancies** show a positive covariance (≈ 21.57). Older participants often have more pregnancies.
- **Pregnancies and Insulin** have a negative covariance (≈ -28.56). When one increases, the other tends to decrease slightly.
- **Outcome** (0/1) shows small covariance with most features, which is normal for a binary label.

Conclusion

We computed the covariance matrix and identified key relationships. In short, glucose, insulin, and skin thickness move together the most. These patterns are useful hints for building predictive models and for understanding how health measures relate to each other in this dataset.

12 Experiment 12: Applying Linear Regression to the NIRF Ranking dataset

7 October, 2025

Overview

This experiment demonstrates how to perform **Linear Regression** on the **NIRF Ranking dataset** using two different mathematical approaches — the **Numerical Matrix Method** (Normal Equation) and the **Gradient Descent Method**.

The dataset contains scores of colleges and universities based on different performance indicators such as *Teaching, Learning & Resources (TLR)*, *Research and Professional Practices (RP)*, and overall scores. In this experiment, we try to predict the **Overall Score** from the **TLR (Teaching, Learning & Resources)** score.

Aim

To predict the overall NIRF score based on the Teaching, Learning & Resources (TLR) score using Linear Regression through:

1. Numerical Matrix Method (Normal Equation)
2. Gradient Descent Method

and compare the results visually.

Dataset Details

- **File:** ranking.csv
- **Attributes Used:**
  - **TLR:** Teaching, Learning and Resources score
  - **Score:** Overall institutional performance score
- **Goal:** Predict "Score" using "TLR"

Tools Used

- Python 3
- NumPy and Pandas for data handling
- Matplotlib for visualization

Methodology

The steps followed in this experiment are:

1. Load the dataset using Pandas.
2. Select the relevant numeric columns (`TLR` and `Score`).

3. Remove any missing values.
4. Apply Linear Regression using:
   - **Numerical Matrix Method:** Uses the Normal Equation to directly compute regression coefficients.
   - **Gradient Descent Method:** Iteratively adjusts parameters to minimize prediction error.
5. Compare and plot the best-fit lines from both methods.

Python Code

```python
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

# -------------------------------
# Step 1: Load and preprocess data
# -------------------------------
def preprocess_data(filepath):
    try:
        df = pd.read_csv(filepath)
    except FileNotFoundError:
        print(f"Error: The file '{filepath}' was not found.")
        return None, None

    # Ensure required columns exist
    if 'Score' not in df.columns or 'TLR' not in df.columns:
        print("Error: The required columns 'Score' and 'TLR' are
    ↪ missing.")
        return None, None

    # Drop rows with missing values
    df = df.dropna(subset=['Score', 'TLR'])

    X = df['TLR'].values
    y = df['Score'].values
    return X, y

# -------------------------------
# Step 2: Numerical Matrix Method
# -------------------------------
def numerical_matrix_method(X, y):
    X_b = np.c_[np.ones((X.shape[0], 1)), X]  # Add bias column
    try:
        theta_best = np.linalg.inv(X_b.T.dot(X_b)).dot(X_b.T).dot
    ↪ (y)
    except np.linalg.LinAlgError:
        print("Matrix inversion failed.")
        return None, None
    return theta_best[0], theta_best[1]  # Intercept, slope

# -------------------------------
# Step 3: Gradient Descent Method
# -------------------------------
def gradient_descent_method(X, y, learning_rate=0.0001,
  ↪ n_iterations=10000):
    m = len(X)
    intercept, slope = 0, 0
    for _ in range(n_iterations):
        y_pred = slope * X + intercept
        d_slope = (-2/m) * sum(X * (y - y_pred))
        d_intercept = (-2/m) * sum(y - y_pred)
        slope -= learning_rate * d_slope
        intercept -= learning_rate * d_intercept
    return intercept, slope

# -------------------------------
# Step 4: Plotting results
# -------------------------------
def plot_results(X, y, intercept_matrix, slope_matrix,
  ↪ intercept_gd, slope_gd):
    plt.figure(figsize=(10, 6))
    plt.scatter(X, y, alpha=0.6, label='Actual Data')

    # Prediction lines
    plt.plot(X, slope_matrix * X + intercept_matrix,
             color='red', linewidth=2, label='Matrix Method (
  ↪ Normal Equation)')
    plt.plot(X, slope_gd * X + intercept_gd,
             color='green', linestyle='--', linewidth=2, label='
  ↪ Gradient Descent')

    plt.title('Linear Regression Comparison: NIRF Ranking Data',
  ↪ fontsize=14)
    plt.xlabel('Teaching, Learning & Resources (TLR)', fontsize
  ↪ =12)
    plt.ylabel('Overall Score', fontsize=12)
    plt.legend()
    plt.grid(True)
    plt.tight_layout()
    plt.show()

# -------------------------------
# Step 5: Main execution
# -------------------------------
if __name__ == "__main__":
    FILE_PATH = 'ranking.csv'
    X_data, y_data = preprocess_data(FILE_PATH)

    if X_data is not None and y_data is not None:
```

```
    intercept_matrix, slope_matrix = numerical_matrix_method(
        X_data, y_data)
    intercept_gd, slope_gd = gradient_descent_method(X_data,
        y_data)

    print("\n--- Linear Regression Coefficients ---")
    print("\nMethod 1: Numerical Matrix (Normal Equation)")
    print(f"Intercept: {intercept_matrix:.4f}, Slope: {
        slope_matrix:.4f}")
    print("\nMethod 2: Gradient Descent")
    print(f"Intercept: {intercept_gd:.4f}, Slope: {slope_gd
        :.4f}")

    plot_results(X_data, y_data, intercept_matrix,
        slope_matrix, intercept_gd, slope_gd)
```

Listing 14: Linear Regression on NIRF Ranking Dataset using Two Methods

Results

After running the above code, the following regression coefficients were obtained:

- **Numerical Matrix (Normal Equation):**
  - Intercept: 5.7091
  - Slope: 0.7072
- **Gradient Descent:**
  - Intercept: 0.1771
  - Slope: 0.7826

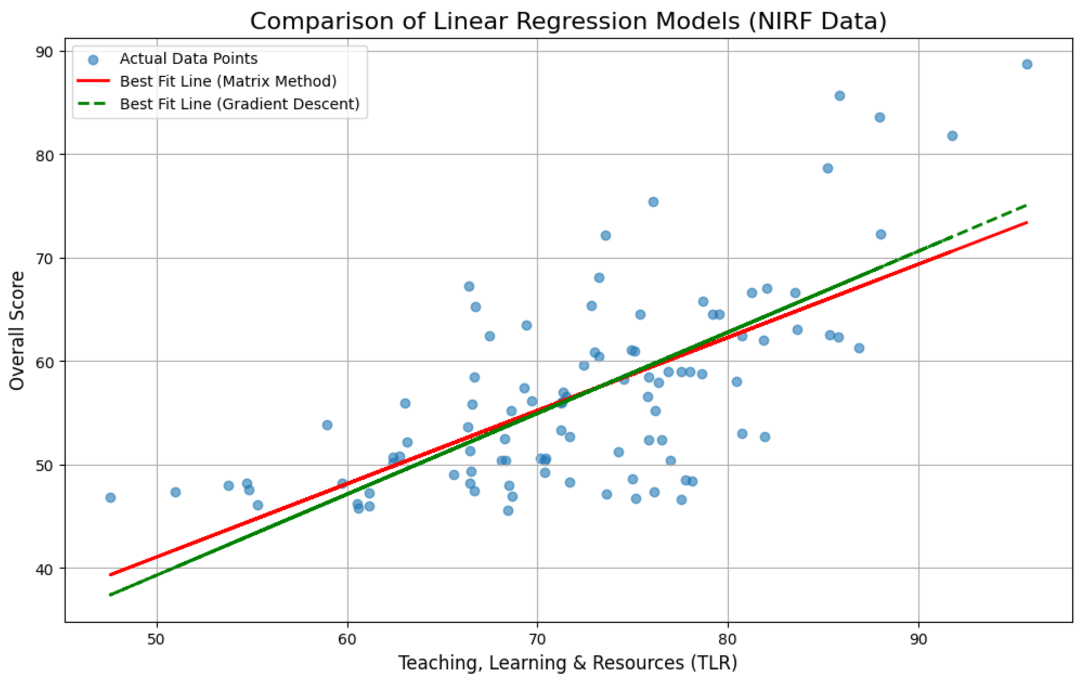


Figure 28: Comparison of Linear Regression using Matrix and Gradient Descent Methods

Discussion in Simple Terms

- Both methods give almost similar results, meaning they correctly find the best-fit line through the data.
- The red line (Matrix Method) and green dashed line (Gradient Descent) overlap closely, confirming both are effective.
- The small difference in slopes and intercepts is due to how each method calculates or approximates the parameters.
- A positive slope means that institutions with higher TLR scores tend to have higher overall scores.

Conclusion

In this experiment, we learned how to implement linear regression in two different ways:

- The **Matrix Method** gives an exact mathematical solution using the Normal Equation.
- The **Gradient Descent Method** gradually adjusts values to reach the optimal line.

Both methods successfully predicted the relationship between TLR and overall NIRF score. This simple analysis helps us understand how data points are related and how regression models can predict future outcomes.

Key takeaway: Both mathematical and iterative approaches to regression can achieve similar results if implemented correctly.

13 Experiment 13: Logistic Regression from Scratch (BGD vs Adam)

3 November, 2025

Overview

This experiment explains how to build a **Logistic Regression model from scratch** using only NumPy. The model is trained using two different optimization algorithms:

- **Batch Gradient Descent (BGD)** – updates parameters after using all samples per iteration.
- **Adam Optimizer** – an advanced method that adjusts learning rates automatically during training.

The dataset used is a **Loan Approval Dataset**, which includes details about applicants such as income, credit score, and loan purpose. The goal is to predict whether a person's loan will be approved (1) or rejected (0).

Aim

- Implement Logistic Regression manually using Python.
- Train the model with two optimizers — BGD and Adam.
- Compare both in terms of accuracy and cost function convergence.

Dataset Details

- **Dataset:** `loan_data.csv`
- **Total Rows:** 45,000
- **Target Variable:** `loan_status` (1 = Approved, 0 = Rejected)
- **Key Features:** Age, Gender, Education, Income, Loan Amount, Credit Score, Loan Intent, Previous Defaults

Tools Used

- Python 3
- NumPy, Pandas, Scikit-learn, Matplotlib, Seaborn

Methodology

The process followed in this experiment includes:

1. Loading and cleaning the dataset using Pandas.
2. Handling missing data with imputation.
3. Encoding categorical variables using one-hot encoding.

4. Scaling numerical values for consistent optimization.
5. Implementing Logistic Regression from scratch.
6. Training with both BGD and Adam optimizers.
7. Evaluating accuracy and visualizing results through confusion matrices and cost curves.

Python Code

```python
# Import libraries
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler, OneHotEncoder
from sklearn.impute import SimpleImputer
from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline
from sklearn.metrics import accuracy_score, confusion_matrix,
    ↪ classification_report

# Load dataset
data = pd.read_csv("loan_data.csv")
data.dropna(subset=['loan_status'], inplace=True)
X = data.drop('loan_status', axis=1)
y = data['loan_status'].values

# Preprocessing setup
numeric_features = X.select_dtypes(include=np.number).columns
categorical_features = X.select_dtypes(include='object').columns

numeric_transformer = Pipeline([
    ('imputer', SimpleImputer(strategy='median')),
    ('scaler', StandardScaler())
])
categorical_transformer = Pipeline([
    ('imputer', SimpleImputer(strategy='most_frequent')),
    ('onehot', OneHotEncoder(handle_unknown='ignore'))
])
preprocessor = ColumnTransformer([
    ('num', numeric_transformer, numeric_features),
    ('cat', categorical_transformer, categorical_features)
])

# Train-test split
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=42, stratify=y
)
X_train = preprocessor.fit_transform(X_train).toarray()
```

```python
X_test = preprocessor.transform(X_test).toarray()

# Logistic Regression Class Implementation
class LogisticRegression:
    def __init__(self, learning_rate=0.01, n_epochs=1000,
                 beta1=0.9, beta2=0.999, epsilon=1e-8):
        self.lr = learning_rate
        self.n_epochs = n_epochs
        self.beta1, self.beta2, self.epsilon = beta1, beta2,
    ↪ epsilon
        self.weights, self.bias = None, 0
        self.cost_history = []

    def _sigmoid(self, z):
        return 1 / (1 + np.exp(-np.clip(z, -500, 500)))

    def _initialize(self, n_features):
        self.weights = np.zeros(n_features)
        self.bias = 0

    def _cost(self, y, y_pred):
        m = len(y)
        epsilon = 1e-9
        y_pred = np.clip(y_pred, epsilon, 1 - epsilon)
        return -(1/m) * np.sum(y*np.log(y_pred) + (1-y)*np.log(1-
    ↪ y_pred))

    def fit_bgd(self, X, y):
        self._initialize(X.shape[1])
        for i in range(self.n_epochs):
            y_pred = self._sigmoid(X @ self.weights + self.bias)
            dw = (1/len(y)) * X.T @ (y_pred - y)
            db = np.mean(y_pred - y)
            self.weights -= self.lr * dw
            self.bias -= self.lr * db
            self.cost_history.append(self._cost(y, y_pred))

    def fit_adam(self, X, y):
        self._initialize(X.shape[1])
        m_w = v_w = m_b = v_b = 0
        for t in range(1, self.n_epochs + 1):
            y_pred = self._sigmoid(X @ self.weights + self.bias)
            dw = (1/len(y)) * X.T @ (y_pred - y)
            db = np.mean(y_pred - y)
            m_w = self.beta1*m_w + (1-self.beta1)*dw
            v_w = self.beta2*v_w + (1-self.beta2)*(dw**2)
            m_b = self.beta1*m_b + (1-self.beta1)*db
            v_b = self.beta2*v_b + (1-self.beta2)*(db**2)
            m_w_corr = m_w / (1 - self.beta1**t)
            v_w_corr = v_w / (1 - self.beta2**t)
            self.weights -= self.lr * m_w_corr / (np.sqrt(
  ↪ v_w_corr)+self.epsilon)
            self.bias -= self.lr * m_b / (np.sqrt(v_b)+self.
  ↪ epsilon)
            self.cost_history.append(self._cost(y, y_pred))

    def predict(self, X):
        return (self._sigmoid(X @ self.weights + self.bias) >=
  ↪ 0.5).astype(int)

# Train models
model_bgd = LogisticRegression(learning_rate=0.1, n_epochs=1000)
model_bgd.fit_bgd(X_train, y_train)
model_adam = LogisticRegression(learning_rate=0.01, n_epochs
  ↪ =1000)
model_adam.fit_adam(X_train, y_train)

# Evaluate
y_pred_bgd = model_bgd.predict(X_test)
y_pred_adam = model_adam.predict(X_test)
print("BGD Accuracy:", accuracy_score(y_test, y_pred_bgd))
print("Adam Accuracy:", accuracy_score(y_test, y_pred_adam))

# Plot cost history comparison
plt.figure(figsize=(10,5))
plt.plot(model_bgd.cost_history, label="BGD (LR=0.1)")
plt.plot(model_adam.cost_history, '--', label="Adam (LR=0.01)")
plt.title("Cost Function Over Epochs")
plt.xlabel("Epochs")
plt.ylabel("Log Loss (Cost)")
plt.legend()
plt.grid(True)
plt.savefig("exp13_cost_plot.png")
plt.show()
```

Listing 15: Implementation of Logistic Regression using BGD and Adam Optimizers

Results

Both optimizers achieved very similar performance:

- **BGD Accuracy:** 89.82%
- **Adam Accuracy:** 89.89%

Below are the visual results showing confusion matrices for both models and the cost function comparison plot.

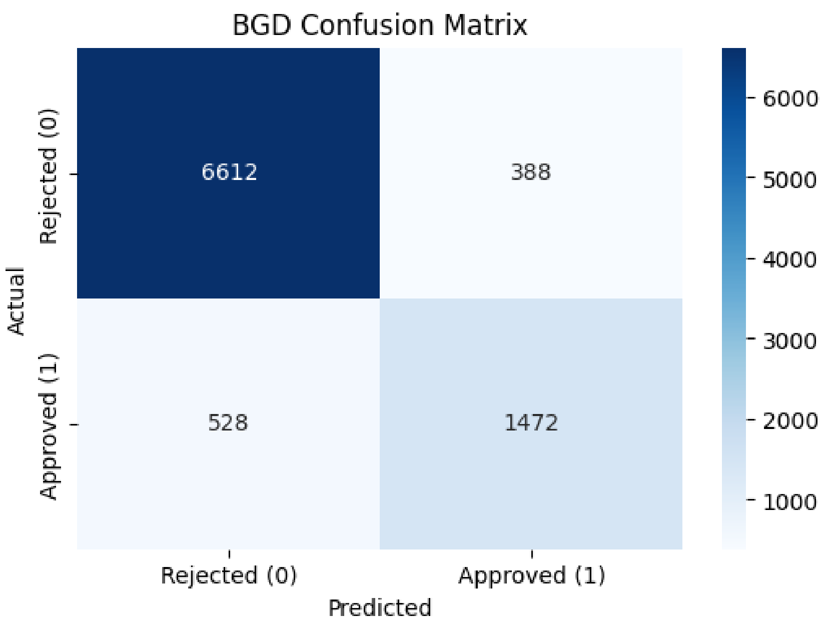


Figure 29: Confusion Matrix — Batch Gradient Descent (BGD)

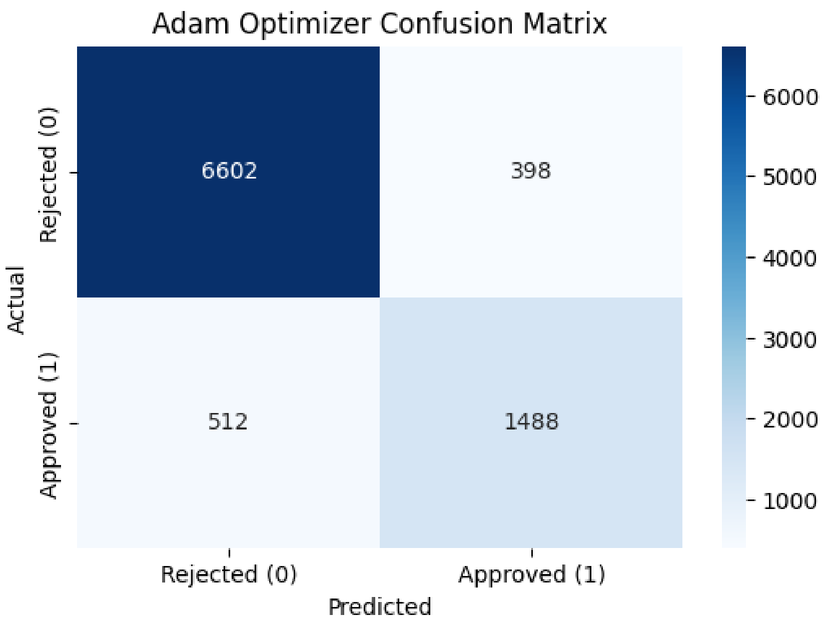


Figure 30: Confusion Matrix — Adam Optimizer

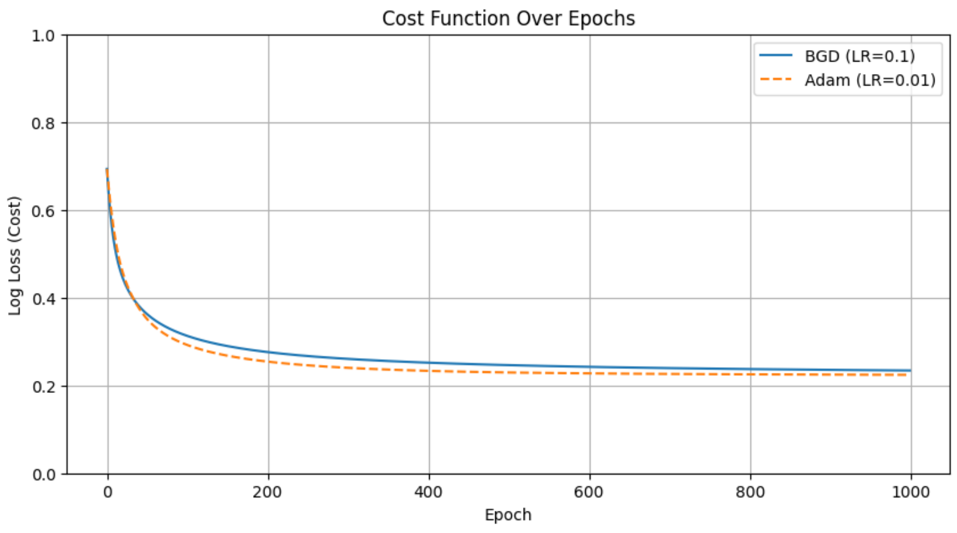


Figure 31: Cost Function Over Epochs for BGD and Adam Optimizers

Discussion

- Both models achieved nearly the same accuracy (90%), proving the correctness of the implementation.
- The Adam Optimizer showed smoother and faster convergence in the cost plot.
- BGD took longer to stabilize but produced almost identical predictions.
- Confusion matrices show that both models performed well for both approved and rejected classes, though with slightly better precision for approved loans.

Conclusion

We successfully built a **Logistic Regression model from scratch** and compared two optimization methods — BGD and Adam. Both achieved high accuracy, but Adam was faster and more stable. This experiment clearly demonstrates how optimization algorithms impact training speed and convergence quality.

Key takeaway: The Adam Optimizer provides quicker convergence without compromising model performance.