Logistic Regression Implementation

Import of all the required libraries

In [1]:

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

Create a dataframe

In [2]:

```
dataframe = pd.read_csv("./diabetes.csv")
dataframe.head()
```

Out[2]:

	Pregnancies	Glucose	BloodPressure	SkinThickness	Insulin	BMI	DiabetesPed
0	6	148	72	35	0	33.6	
1	1	85	66	29	0	26.6	
2	8	183	64	0	0	23.3	
3	1	89	66	23	94	28.1	
4	0	137	40	35	168	43.1	

In [3]:

```
dataframe.columns
```

Out[3]:

```
Index(['Pregnancies', 'Glucose', 'BloodPressure', 'SkinThickness', 'Insu
lin',
       'BMI', 'DiabetesPedigreeFunction', 'Age', 'Outcome'],
      dtype='object')
```

In [4]:

```
dataframe.dtypes
```

Out[4]:

```
Pregnancies                   int64
Glucose                       int64
BloodPressure                 int64
SkinThickness                 int64
Insulin                       int64
BMI                         float64
DiabetesPedigreeFunction    float64
Age                           int64
Outcome                       int64
dtype: object
```

Missing values in a given dataset

In [5]:

```
dataframe.isnull().sum()
```

Out[5]:

```
Pregnancies                 0
Glucose                     0
BloodPressure               0
SkinThickness               0
Insulin                     0
BMI                         0
DiabetesPedigreeFunction    0
Age                         0
Outcome                     0
dtype: int64
```

In [6]:

```
dataframe.head(20)
```

Out[6]:

	Pregnancies	Glucose	BloodPressure	SkinThickness	Insulin	BMI	DiabetesPe
0	6	148	72	35	0	33.6	
1	1	85	66	29	0	26.6	
2	8	183	64	0	0	23.3	
3	1	89	66	23	94	28.1	
4	0	137	40	35	168	43.1	
5	5	116	74	0	0	25.6	
6	3	78	50	32	88	31.0	
7	10	115	0	0	0	35.3	
8	2	197	70	45	543	30.5	
9	8	125	96	0	0	0.0	
10	4	110	92	0	0	37.6	
11	10	168	74	0	0	38.0	
12	10	139	80	0	0	27.1	
13	1	189	60	23	846	30.1	
14	5	166	72	19	175	25.8	
15	7	100	0	0	0	30.0	
16	0	118	84	47	230	45.8	
17	7	107	74	0	0	29.6	
18	1	103	30	38	83	43.3	
19	1	115	70	30	96	34.6	

As we Can observe 0 for BloodPressure, SkinThickness that does not seems right hence we need to do data imputation

- Data Imputation of 0's in every feature
- Size of the data

In [7]:

```
dataframe.shape
```

Out[7]:

```
(768, 9)
```

In [8]:

```
dataframe.Outcome.value_counts()
```

Out[8]:

```
Outcome
0    500
1    268
Name: count, dtype: int64
```

In [9]:

```
print(dataframe.Outcome.unique())
print(dataframe["Outcome"].unique())
```

```
[1 0]
[1 0]
```

Observations

- Target Column: Outcome[0,1] (Binary Classification Task)
- As the target column is available in the dataset, supervised machine learning algorithm.
- Records: 768

EDA

Correlation Coefficient

In [10]:

```
# Find featuers whos correlation coeficient is greater than 0.6
dataframe.corr()['Pregnancies']
```

Out[10]:

```
Pregnancies                 1.000000
Glucose                     0.129459
BloodPressure               0.141282
SkinThickness              -0.081672
Insulin                    -0.073535
BMI                         0.017683
DiabetesPedigreeFunction   -0.033523
Age                         0.544341
Outcome                     0.221898
Name: Pregnancies, dtype: float64
```

In [11]:

```
# Find featuers whos correlation coeficient is greater than 0.6
dataframe.corr()['Glucose']
```

Out[11]:

```
Pregnancies                 0.129459
Glucose                     1.000000
BloodPressure               0.152590
SkinThickness               0.057328
Insulin                     0.331357
BMI                         0.221071
DiabetesPedigreeFunction    0.137337
Age                         0.263514
Outcome                     0.466581
Name: Glucose, dtype: float64
```

In [12]:

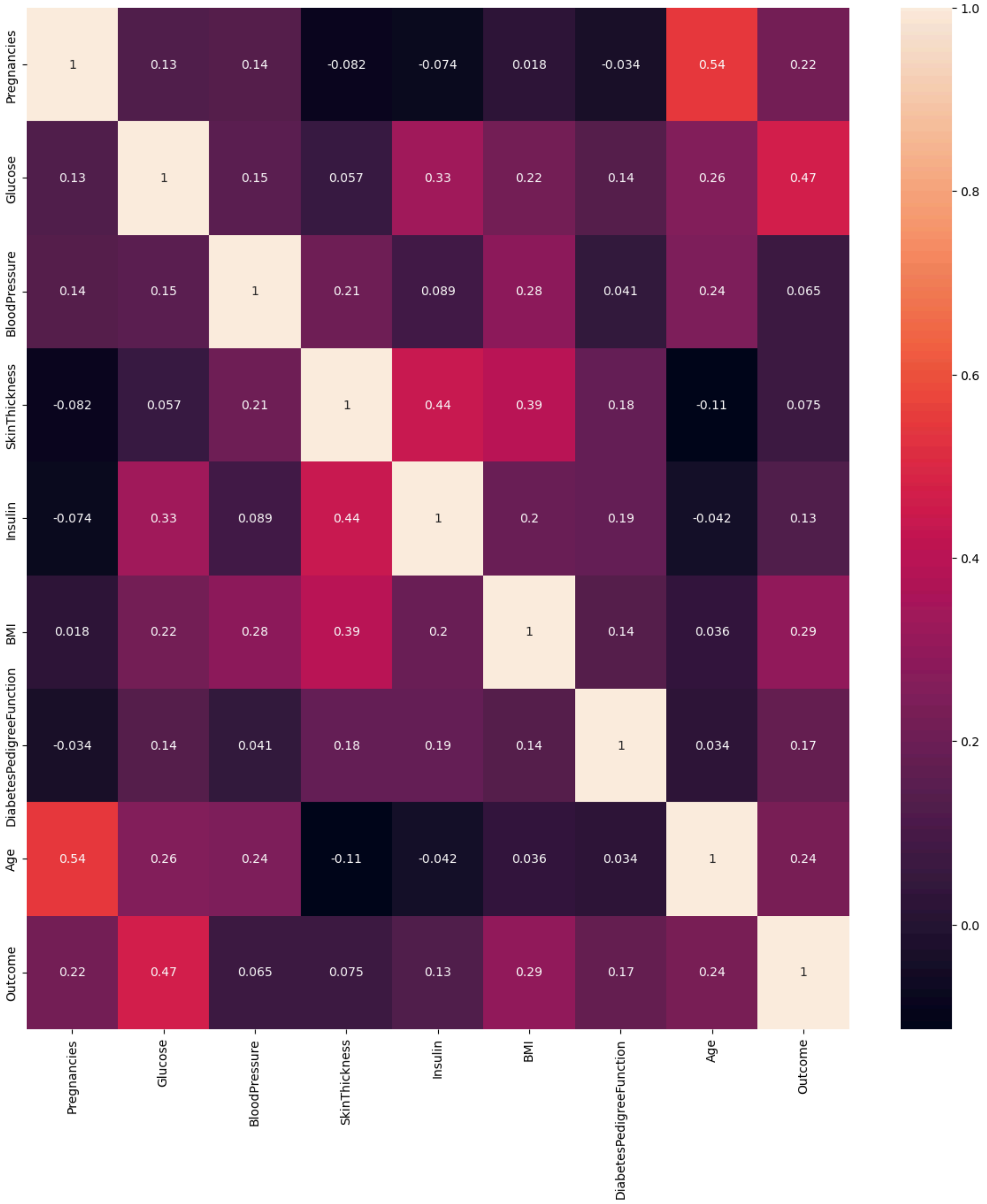
```
dataframe.corr()
```

Out[12]:

	Pregnancies	Glucose	BloodPressure	SkinThickness
Pregnancies	1.000000	0.129459	0.141282	-0.081672
Glucose	0.129459	1.000000	0.152590	0.057328
BloodPressure	0.141282	0.152590	1.000000	0.207371
SkinThickness	-0.081672	0.057328	0.207371	1.000000
Insulin	-0.073535	0.331357	0.088933	0.436783
BMI	0.017683	0.221071	0.281805	0.392573
DiabetesPedigreeFunction	-0.033523	0.137337	0.041265	0.183928
Age	0.544341	0.263514	0.239528	-0.113970
Outcome	0.221898	0.466581	0.065068	0.074752

In [13]:

```python
plt.figure(figsize=(15,15))
ax = sns.heatmap(dataframe.corr(), annot=True)
plt.savefig('correlation-coefficient.jpg')
plt.show()
```



Descriptive Statistics of the given data

In [14]:

```
dataframe.describe()
```

Out[14]:

	Pregnancies	Glucose	BloodPressure	SkinThickness	Insulin	
count	768.000000	768.000000	768.000000	768.000000	768.000000	768.0
mean	3.845052	120.894531	69.105469	20.536458	79.799479	31.9
std	3.369578	31.972618	19.355807	15.952218	115.244002	7.8
min	0.000000	0.000000	0.000000	0.000000	0.000000	0.0
25%	1.000000	99.000000	62.000000	0.000000	0.000000	27.3
50%	3.000000	117.000000	72.000000	23.000000	30.500000	32.0
75%	6.000000	140.250000	80.000000	32.000000	127.250000	36.6
max	17.000000	199.000000	122.000000	99.000000	846.000000	67.1

Data Imputation

Here for data imputation one need to have knowledge of

- Symmetric distribution
- Non-Symmetric distribution
- Left skewed
- Right skewed

So that appropriate mean mode median can be selected for imputation

Data Imputation: When to Use **Mean**, **Mode**, and **Median**

Data imputation is the process of replacing missing values in a dataset with substituted values to maintain data integrity and enable meaningful analysis. The choice of imputation method—whether **mean**, **median**, or **mode**—depends on several factors, including the **type of data**, **distribution of the data**, and the **nature of missing values**.

Here's a detailed explanation of when to use each imputation method:

---

1. **Mean Imputation**

- **Definition**: Mean imputation replaces missing values with the **average** (mean) of the available data for that feature.

- **When to Use**:

  - **Continuous Data**: Mean imputation is suitable for **continuous numerical data** (e.g., age, income, height, weight).
  - **Normally Distributed Data**: It works well when the data follows a **normal distribution** (i.e., the data is symmetrically distributed with no extreme

outliers).

  - **Low Percentage of Missing Data**: Use mean imputation when the percentage of missing values is low (typically <5%).

- **When Not to Use**:

  - **Skewed Data**: For data that is skewed or has extreme outliers, the mean can be significantly affected, leading to distorted imputation.
  - **Categorical Data**: Mean imputation is not suitable for **categorical data** because mean calculation is only valid for numerical data.

Example Use Case:

- In a dataset of **employees' salaries**, you can impute missing salary values using the mean of the available salary data if the distribution is normal.

```
mean_value = df['salary'].mean()
df['salary'].fillna(mean_value, inplace=True)
```

2. Median Imputation

- **Definition**: Median imputation replaces missing values with the **median** (the middle value) of the available data for that feature.

- **When to Use**:

  - **Continuous Data**: Like mean imputation, median imputation is suitable for **continuous numerical data**.
  - **Skewed Data**: Median imputation is more appropriate when the data is **skewed** or contains **outliers**, as the median is robust and not affected by extreme values.
  - **Non-Normal Distributions**: Use median imputation when the distribution of the data is **non-normal** (e.g., skewed left or right).

- **When Not to Use**:

  - **Categorical Data**: Median imputation is not suitable for **categorical data** for the same reasons as mean imputation.
  - **Small Datasets**: If the dataset is small, the median might not adequately capture the central tendency of the data.

Example Use Case:

- For a dataset of **housing prices** where a few extremely high prices skew the distribution, median imputation is a better choice to replace missing housing prices.

```
median_value = df['house_price'].median()
df['house_price'].fillna(median_value, inplace=True)
```

3. Mode Imputation

- **Definition**: Mode imputation replaces missing values with the **mode** (the most frequent value) of the available data for that feature.

- **When to Use**:

    - **Categorical Data**: Mode imputation is suitable for **categorical data** (e.g., gender, occupation, product type) because the mode represents the most common category or class.
    - **Ordinal Data**: For **ordinal data** (e.g., education level, satisfaction rating), mode imputation is also appropriate because it maintains the most frequent category.
    - **Nominal Data**: Mode imputation works well with **nominal data** (categories without a specific order, like color or product type).
    - **High Frequency Categories**: When a category appears frequently in the dataset, using the mode helps maintain consistency with the existing distribution.
- **When Not to Use**:

    - **Continuous Data**: Mode imputation is not suitable for continuous numerical data, as there is no clear "most frequent value" in continuous data.
    - **Multiple Modes**: If there are multiple modes (bimodal or multimodal data), choosing the right mode for imputation can be tricky and might not accurately reflect the true distribution.

Example Use Case:

- In a dataset of **customer survey responses** where the question asks for the most frequent category (e.g., "satisfied", "neutral", "dissatisfied"), mode imputation is appropriate for missing responses.

```
mode_value = df['customer_satisfaction'].mode()[0]
df['customer_satisfaction'].fillna(mode_value, inplace=True)
```

Summary of When to Use Each Method:

Imputation Method	Best for Data Type	When to Use	When to Avoid
Mean Imputation	Continuous (Numerical)	- Normal distribution - Low percentage of missing values	- Skewed data - Data with outliers - Categorical data
Median Imputation	Continuous (Numerical)	- Skewed data - Data with outliers - Non-normal distribution	- Small datasets - Categorical data
Mode Imputation	Categorical & Ordinal	- Categorical data - Ordinal data - High frequency categories	- Continuous data - Multiple modes present

Considerations for Choosing the Right Imputation Method:

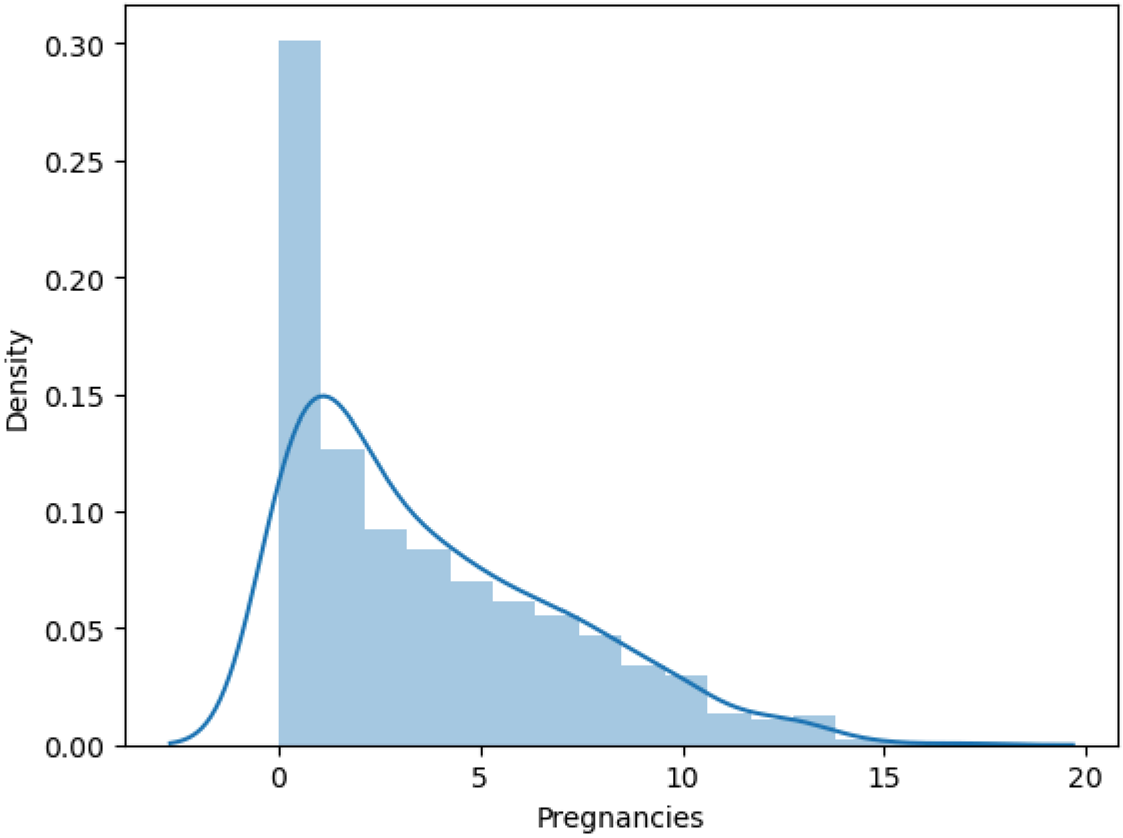
1. **Type of Data**:
   - **Numerical Data**: Use **mean** or **median** depending on the distribution.
   - **Categorical Data**: Use **mode**.
2. **Distribution**:
   - **Normal Distribution**: Mean imputation is effective.
   - **Skewed Distribution/Outliers**: Median imputation is better due to robustness.
3. **Percentage of Missing Data**:
   - **Low Percentage**: Mean or median imputation can work well.
   - **High Percentage**: Use caution; too much imputation can distort the dataset.
4. **Dataset Size**:
   - **Large Datasets**: Median and mean imputation work well.
   - **Small Datasets**: Mode may be more effective for categorical data, as other methods might distort the limited information.

Choosing the right imputation method requires a thorough understanding of your dataset and the nature of the missing data. Each method has its strengths and limitations depending on the context.

In [15]:
```
# Pregnancies -> Median
sns.distplot(dataframe.Pregnancies)
```

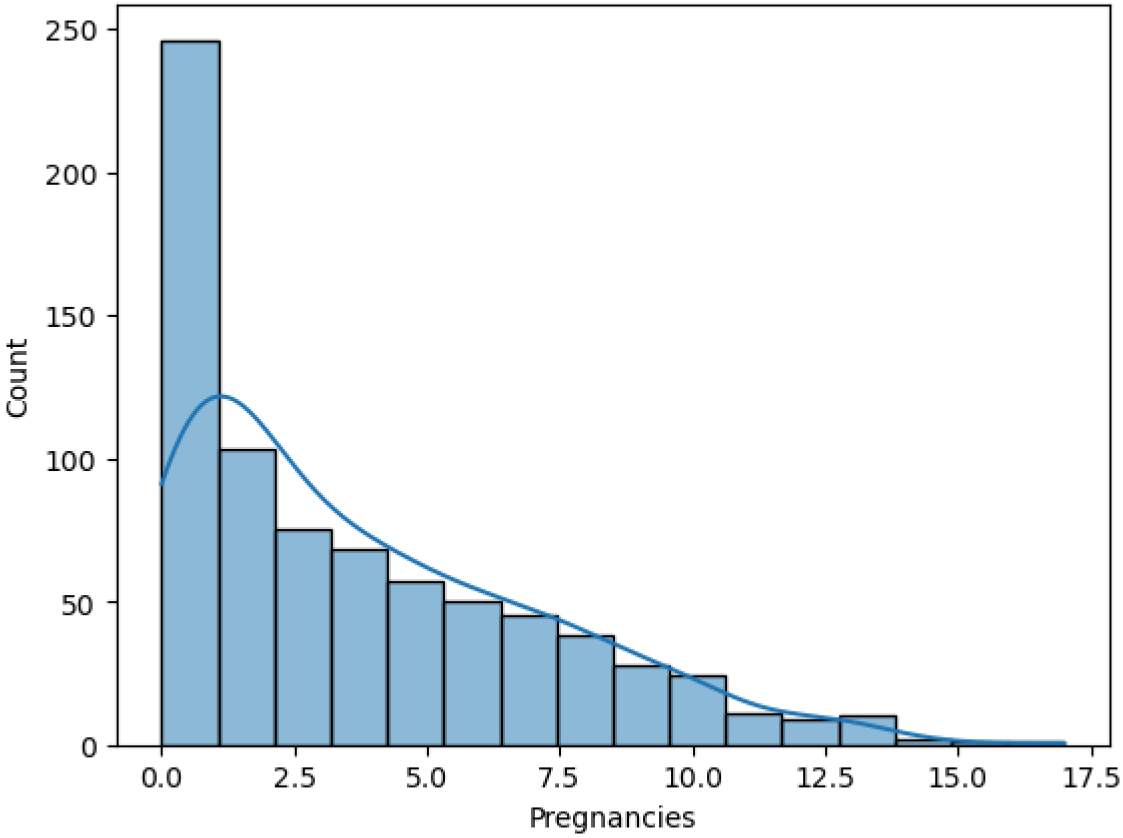
```
/var/folders/8h/zprf7hjs319_78816p34b90c0000gn/T/ipykernel_91019/325720374
0.py:2: UserWarning:

`distplot` is a deprecated function and will be removed in seaborn v0.14.
0.

Please adapt your code to use either `displot` (a figure-level function wi
th
similar flexibility) or `histplot` (an axes-level function for histogram
s).

For a guide to updating your code to use the new functions, please see
https://gist.github.com/mwaskom/de44147ed2974457ad6372750bbe5751

  sns.distplot(dataframe.Pregnancies)
```

Out[15]:
```
<Axes: xlabel='Pregnancies', ylabel='Density'>
```



In [16]:
```
sns.histplot(dataframe.Pregnancies, kde=True)
```

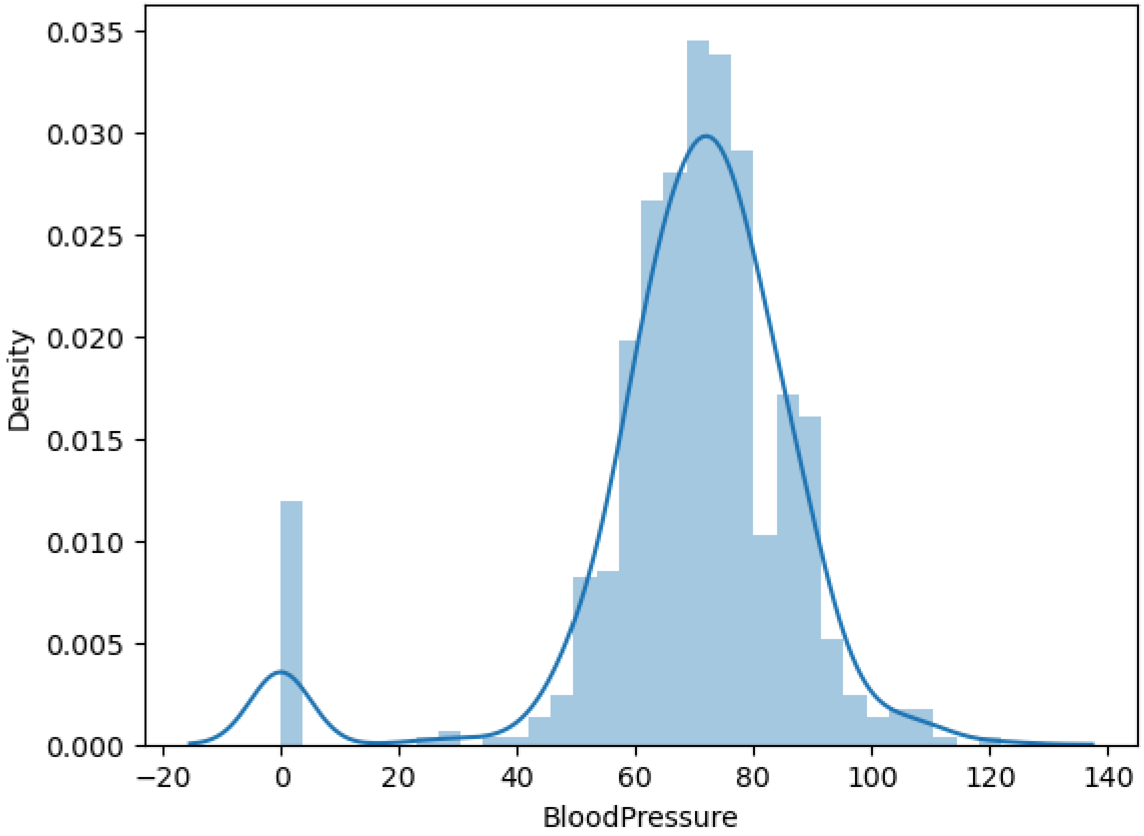
Out[16]:
```
<Axes: xlabel='Pregnancies', ylabel='Count'>
```



In [17]:
```
## BP -> Mean
sns.distplot(dataframe.BloodPressure)
```

```
/var/folders/8h/zprf7hjs319_78816p34b90c0000gn/T/ipykernel_91019/89164806
8.py:2: UserWarning:

`distplot` is a deprecated function and will be removed in seaborn v0.14.
0.

Please adapt your code to use either `displot` (a figure-level function wi
th
similar flexibility) or `histplot` (an axes-level function for histogram
s).

For a guide to updating your code to use the new functions, please see
https://gist.github.com/mwaskom/de44147ed2974457ad6372750bbe5751

  sns.distplot(dataframe.BloodPressure)
```

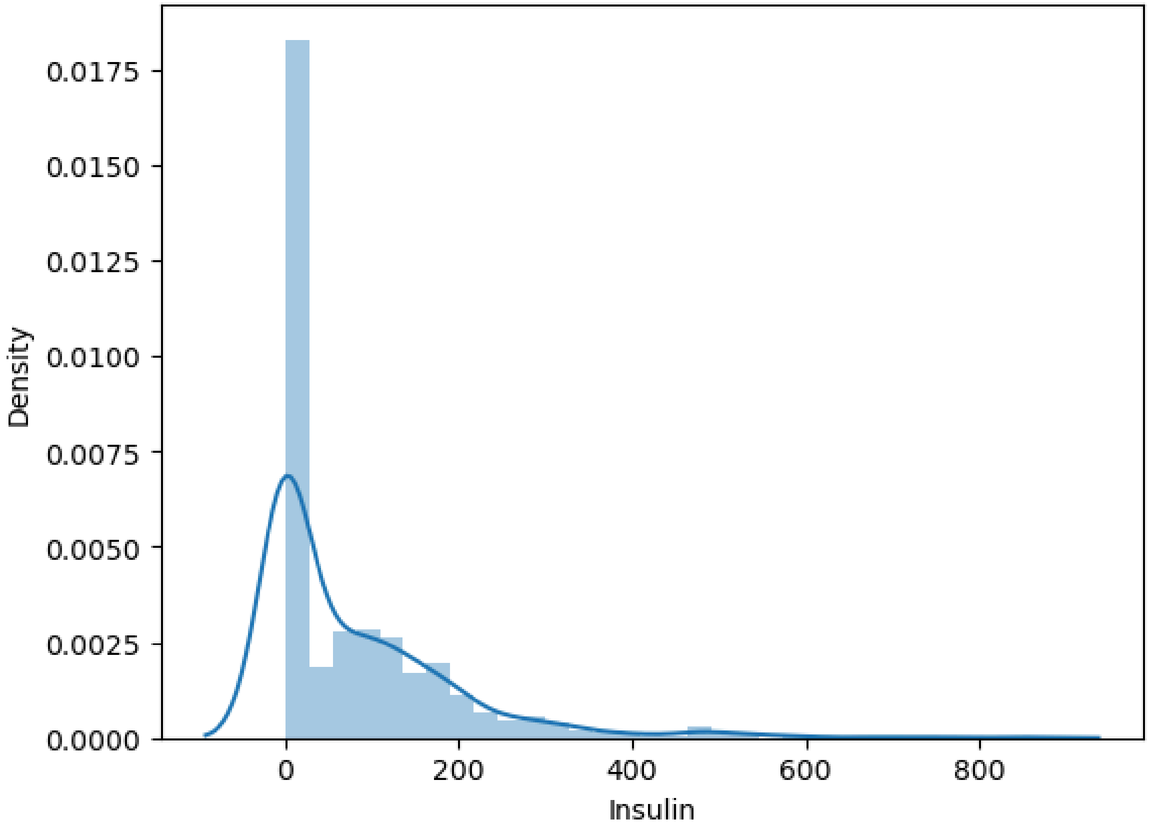
```
Out[17]: <Axes: xlabel='BloodPressure', ylabel='Density'>
```



```
In [18]: # Insulin -> Median
         sns.distplot(dataframe.Insulin)
```

```
/var/folders/8h/zprf7hjs319_78816p34b90c0000gn/T/ipykernel_91019/257615224
7.py:2: UserWarning:

`distplot` is a deprecated function and will be removed in seaborn v0.14.
0.

Please adapt your code to use either `displot` (a figure-level function wi
th
similar flexibility) or `histplot` (an axes-level function for histogram
s).

For a guide to updating your code to use the new functions, please see
https://gist.github.com/mwaskom/de44147ed2974457ad6372750bbe5751

  sns.distplot(dataframe.Insulin)
```

```
Out[18]: <Axes: xlabel='Insulin', ylabel='Density'>
```



```
In [19]: dataframe.columns
```

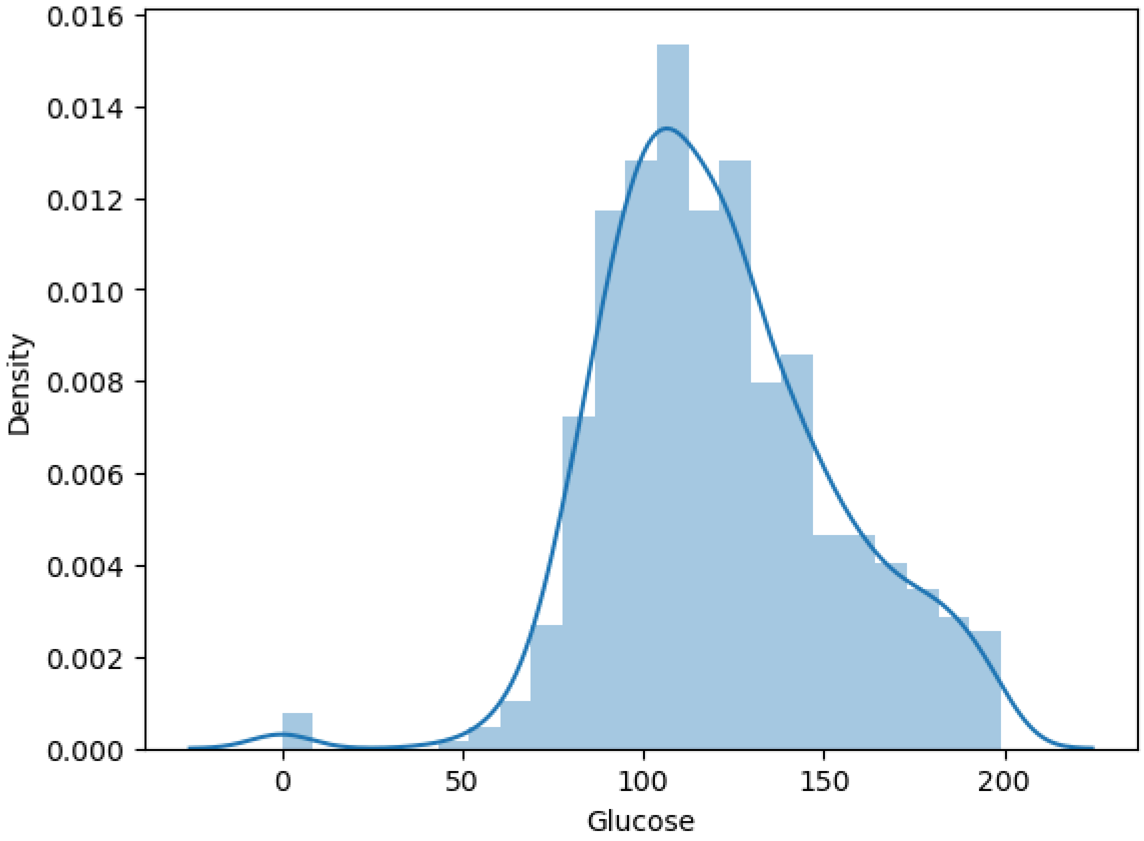
```
Out[19]: Index(['Pregnancies', 'Glucose', 'BloodPressure', 'SkinThickness', 'Insu
lin',
       'BMI', 'DiabetesPedigreeFunction', 'Age', 'Outcome'],
      dtype='object')
```

```
In [20]: ## Insuline -> Right skewed distribution
dataframe['Insulin'] = dataframe['Insulin'].replace(0, dataframe['Insulin
```

```
In [21]: sns.distplot(dataframe.Glucose)
```

```
/var/folders/8h/zprf7hjs319_78816p34b90c0000gn/T/ipykernel_91019/223043267
7.py:1: UserWarning:

`distplot` is a deprecated function and will be removed in seaborn v0.14.
0.

Please adapt your code to use either `displot` (a figure-level function wi
th
similar flexibility) or `histplot` (an axes-level function for histogram
s).

For a guide to updating your code to use the new functions, please see
https://gist.github.com/mwaskom/de44147ed2974457ad6372750bbe5751

  sns.distplot(dataframe.Glucose)
```

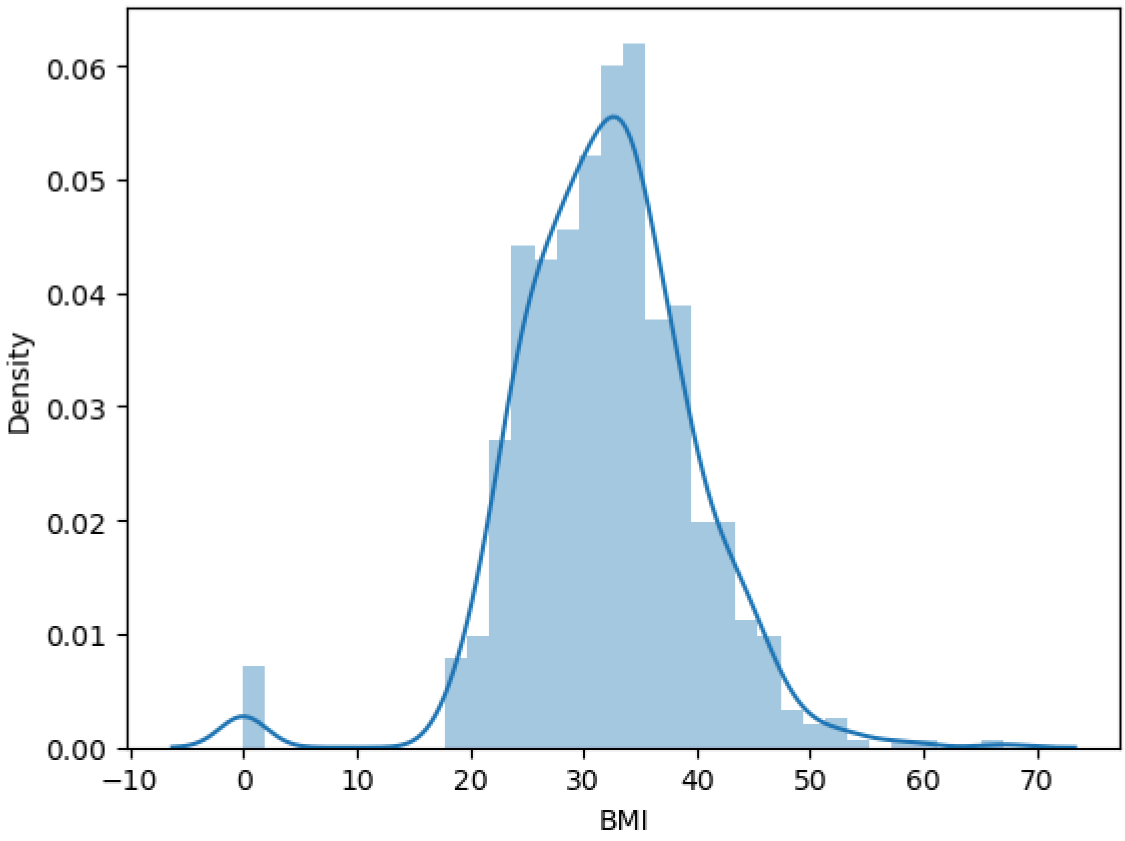
```
Out[21]: <Axes: xlabel='Glucose', ylabel='Density'>
```



```
In [22]: sns.distplot(dataframe.BMI)
```

```
/var/folders/8h/zprf7hjs319_78816p34b90c0000gn/T/ipykernel_91019/252098079
3.py:1: UserWarning: 

`distplot` is a deprecated function and will be removed in seaborn v0.14.
0.

Please adapt your code to use either `displot` (a figure-level function wi
th
similar flexibility) or `histplot` (an axes-level function for histogram
s).

For a guide to updating your code to use the new functions, please see
https://gist.github.com/mwaskom/de44147ed2974457ad6372750bbe5751

  sns.distplot(dataframe.BMI)
```

```
Out[22]:  <Axes: xlabel='BMI', ylabel='Density'>
```

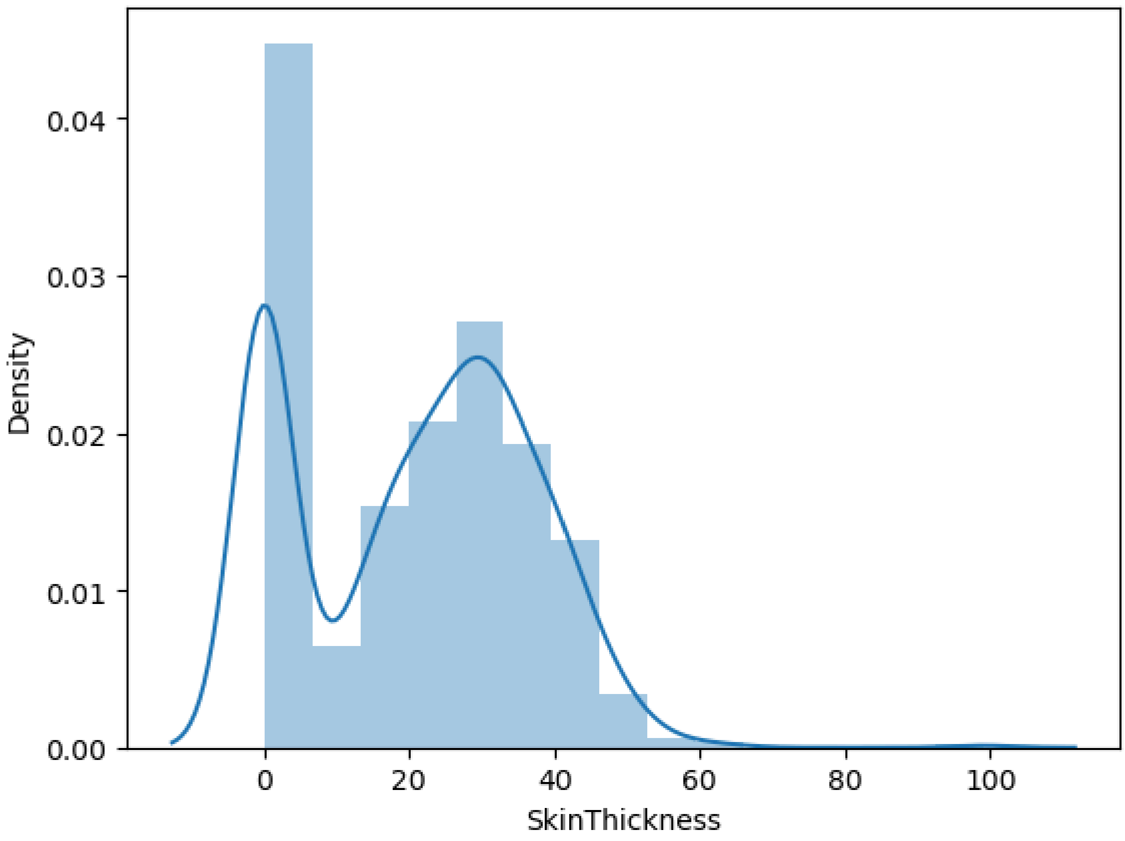


```
In [23]:  sns.distplot(dataframe.SkinThickness)
```

```
/var/folders/8h/zprf7hjs319_78816p34b90c0000gn/T/ipykernel_91019/386125304
5.py:1: UserWarning: 

`distplot` is a deprecated function and will be removed in seaborn v0.14.
0.

Please adapt your code to use either `displot` (a figure-level function wi
th
similar flexibility) or `histplot` (an axes-level function for histogram
s).

For a guide to updating your code to use the new functions, please see
https://gist.github.com/mwaskom/de44147ed2974457ad6372750bbe5751

  sns.distplot(dataframe.SkinThickness)
```

Out[23]:

```
<Axes: xlabel='SkinThickness', ylabel='Density'>
```

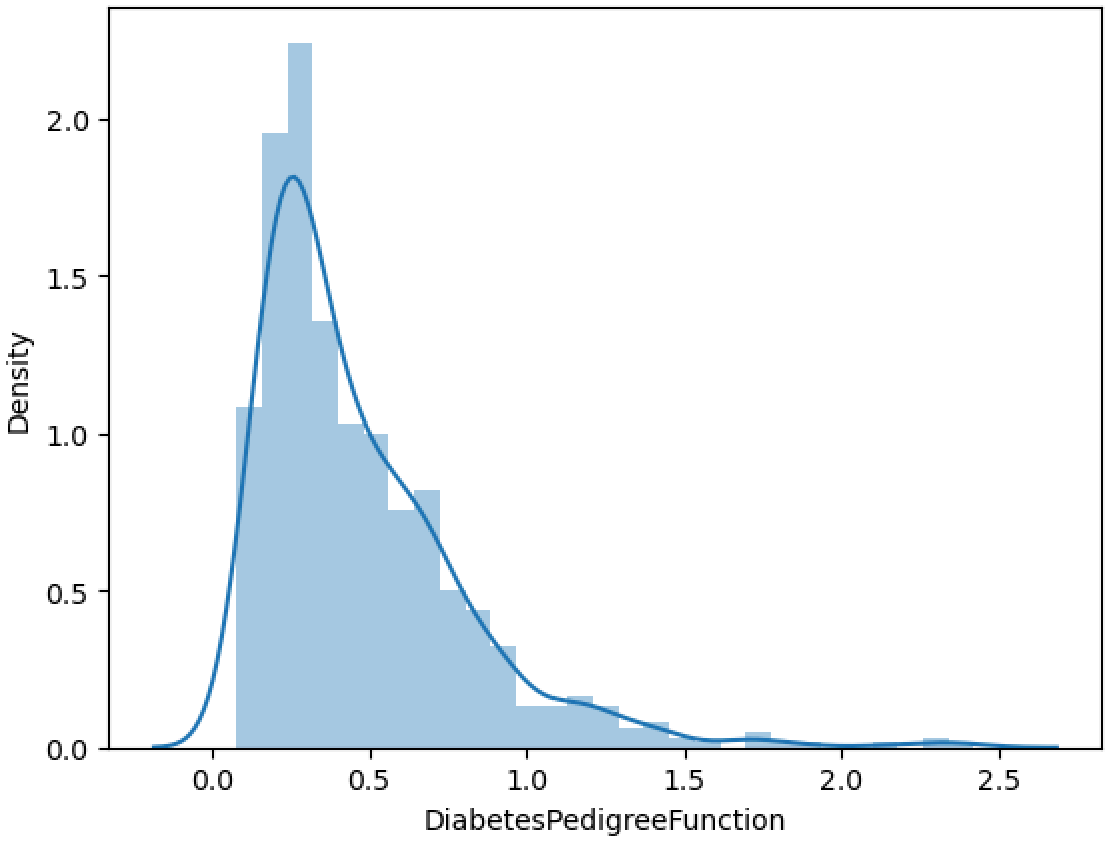


In [24]:

```
sns.distplot(dataframe.DiabetesPedigreeFunction)
```

```
/var/folders/8h/zprf7hjs319_78816p34b90c0000gn/T/ipykernel_91019/264275873
4.py:1: UserWarning:

`distplot` is a deprecated function and will be removed in seaborn v0.14.
0.

Please adapt your code to use either `displot` (a figure-level function wi
th
similar flexibility) or `histplot` (an axes-level function for histogram
s).

For a guide to updating your code to use the new functions, please see
https://gist.github.com/mwaskom/de44147ed2974457ad6372750bbe5751

  sns.distplot(dataframe.DiabetesPedigreeFunction)
```

```
Out[24]: <Axes: xlabel='DiabetesPedigreeFunction', ylabel='Density'>
```

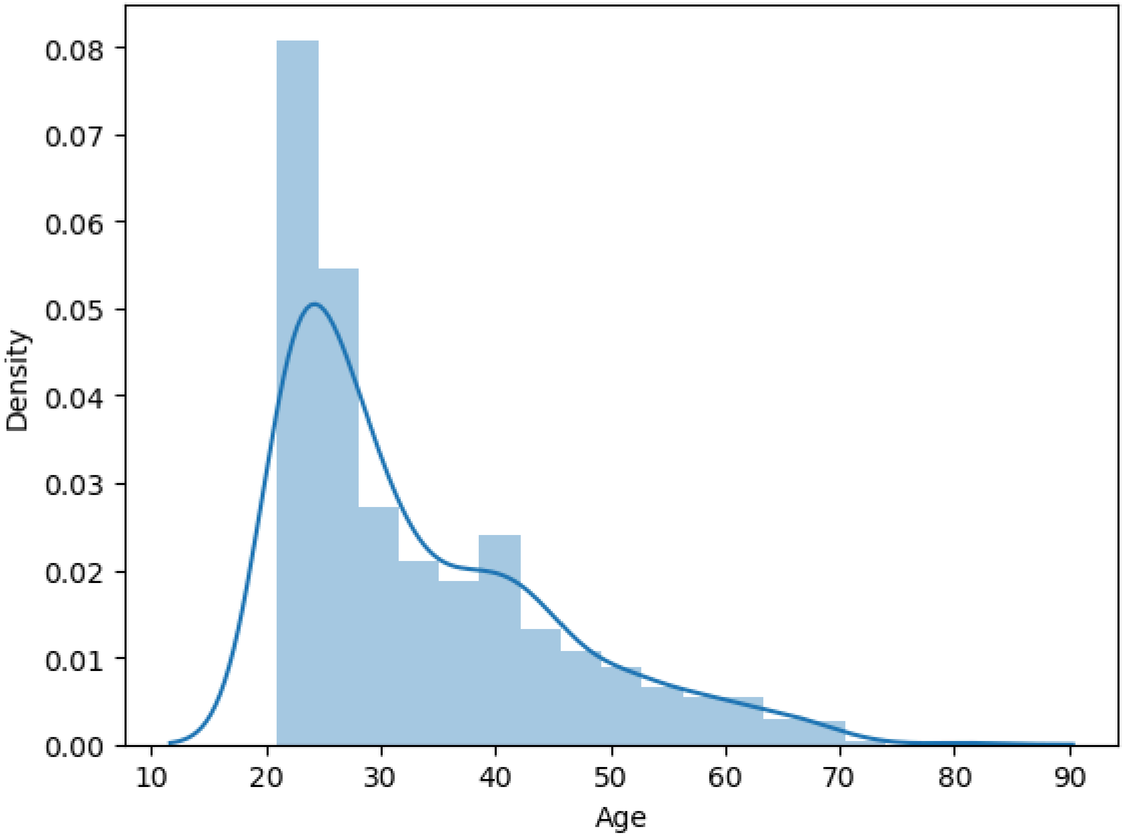


```
In [25]: sns.distplot(dataframe.Age)
```

```
/var/folders/8h/zprf7hjs319_78816p34b90c0000gn/T/ipykernel_91019/269143098
7.py:1: UserWarning:

`distplot` is a deprecated function and will be removed in seaborn v0.14.
0.

Please adapt your code to use either `displot` (a figure-level function wi
th
similar flexibility) or `histplot` (an axes-level function for histogram
s).

For a guide to updating your code to use the new functions, please see
https://gist.github.com/mwaskom/de44147ed2974457ad6372750bbe5751

  sns.distplot(dataframe.Age)
```

Out[25]:

```
<Axes: xlabel='Age', ylabel='Density'>
```



In [26]:

```
dataframe['Pregnancies'] = dataframe['Pregnancies'].replace(0, dataframe[
dataframe['Glucose'] = dataframe['Glucose'].replace(0, dataframe['Glucose
dataframe['BloodPressure'] = dataframe['BloodPressure'].replace(0, datafr
dataframe['SkinThickness'] = dataframe['SkinThickness'].replace(0, datafr
dataframe['BMI'] = dataframe['BMI'].replace(0, dataframe['BMI'].mean())
dataframe['DiabetesPedigreeFunction'] = dataframe['DiabetesPedigreeFuncti
dataframe['Age'] = dataframe['Age'].replace(0, dataframe['Age'].median())
```

In [27]:

```
dataframe.head(20)
```

Out[27]:

	Pregnancies	Glucose	BloodPressure	SkinThickness	Insulin	BMI	Diab
0	6	148.0	72.000000	35	30.5	33.600000	
1	1	85.0	66.000000	29	30.5	26.600000	
2	8	183.0	64.000000	23	30.5	23.300000	
3	1	89.0	66.000000	23	94.0	28.100000	
4	3	137.0	40.000000	35	168.0	43.100000	
5	5	116.0	74.000000	23	30.5	25.600000	
6	3	78.0	50.000000	32	88.0	31.000000	
7	10	115.0	69.105469	23	30.5	35.300000	
8	2	197.0	70.000000	45	543.0	30.500000	
9	8	125.0	96.000000	23	30.5	31.992578	
10	4	110.0	92.000000	23	30.5	37.600000	
11	10	168.0	74.000000	23	30.5	38.000000	
12	10	139.0	80.000000	23	30.5	27.100000	
13	1	189.0	60.000000	23	846.0	30.100000	
14	5	166.0	72.000000	19	175.0	25.800000	
15	7	100.0	69.105469	23	30.5	30.000000	
16	3	118.0	84.000000	47	230.0	45.800000	
17	7	107.0	74.000000	23	30.5	29.600000	
18	1	103.0	30.000000	38	83.0	43.300000	
19	1	115.0	70.000000	30	96.0	34.600000	

- Descriptive Statistics and it's significance
- Correlation Coefficient and it's significance
- Types of Distribution and it's significance
- Median is more robust to outliers and why
- Data Imputation via Mean and Median(Numeric Data) => Symmetric -> Mean and Skewed -> Median, Categorical Data => Mode

Outlier Detection

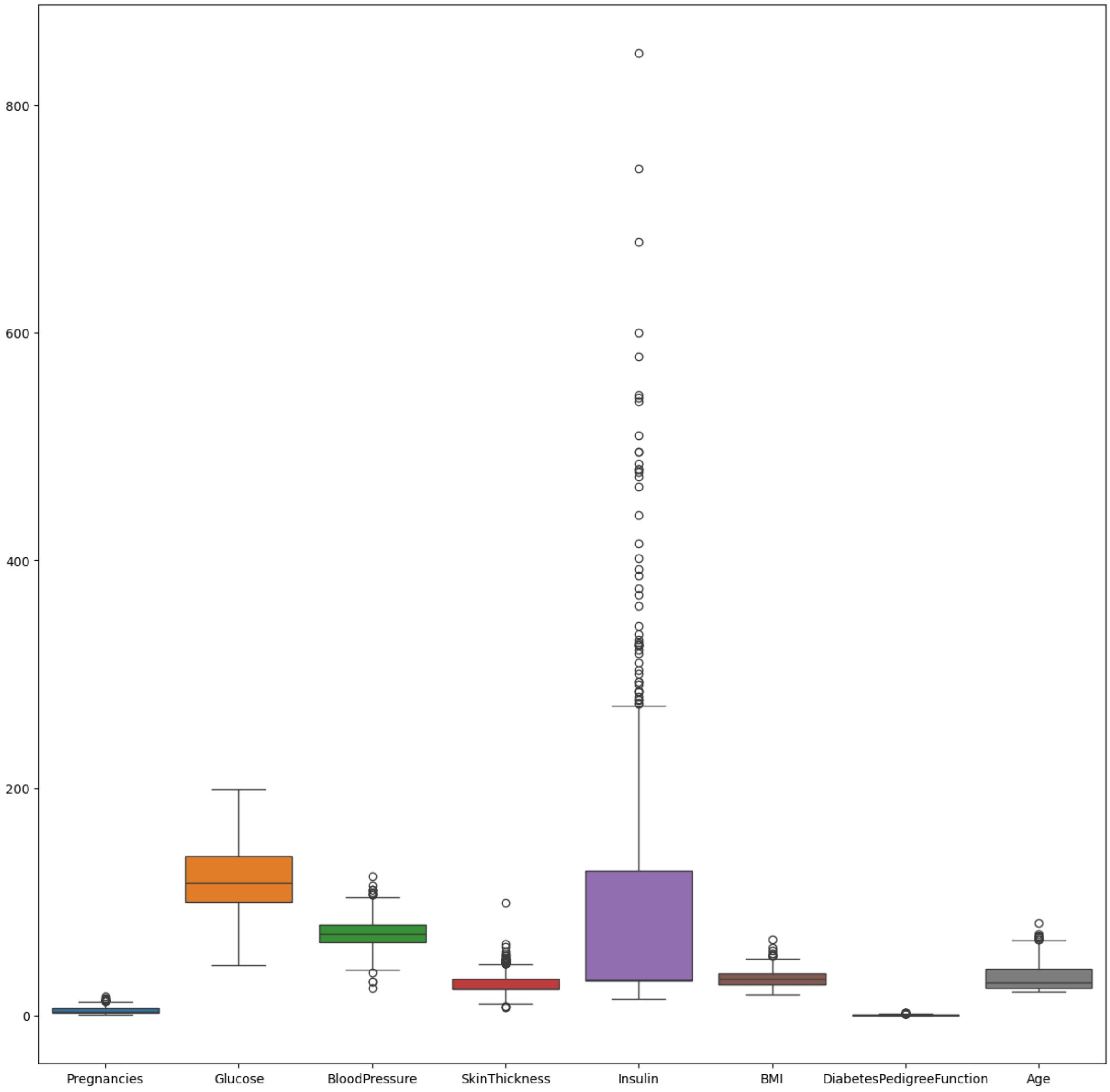
In [28]:

```
## X -> input features y -> target value
X = dataframe.drop(columns='Outcome', axis=1)
y = dataframe['Outcome']
```

Outlier Detection -> Box Plot

In [29]:

```
fig, ax = plt.subplots(figsize = (15, 15))
sns.boxplot(data = X, ax=ax)
plt.savefig('boxPlot.jpg')
```



In [30]:
```
X.shape
```

Out[30]:
```
(768, 8)
```

In [31]:
```
y.shape
```

Out[31]:
```
(768,)
```

Below code remove outliers

In [32]:
```
# Removing outliers for all required columns
cols = ['Pregnancies', 'Glucose', 'BloodPressure', 'SkinThickness', 'Insu
# X_outlier_detection=X
# y_outlier_detection=y
for col in cols:
    Q1 = X[col].quantile(0.25)
    Q3 = X[col].quantile(0.75)
    IQR = Q3 - Q1
    lower_bound = Q1 - 1.5 * IQR
    upper_bound = Q3 + 1.5 * IQR
    mask = (X[col] >= lower_bound) & (X[col] <= upper_bound)
    X_outlier_detection = X[mask]
    y_outlier_detection = y[mask]
    # X_outlier_detection = X_outlier_detection_total[mask]
    # y_outlier_detection = y_outlier_detection_total[mask]
    print(mask)
    for idx, name in enumerate(mask.value_counts().index.tolist()):
     print(f"for {col},{name}->{mask.value_counts()[idx]}")
```

```
0      True
1      True
2      True
3      True
4      True
       ... 
763    True
764    True
765    True
766    True
767    True
Name: Pregnancies, Length: 768, dtype: bool
for Pregnancies,True->754
for Pregnancies,False->14
0      True
1      True
2      True
3      True
4      True
       ... 
763    True
764    True
765    True
766    True
767    True
Name: Glucose, Length: 768, dtype: bool
for Glucose,True->768
0      True
1      True
2      True
3      True
4      True
       ... 
763    True
764    True
765    True
766    True
767    True
Name: BloodPressure, Length: 768, dtype: bool
for BloodPressure,True->754
for BloodPressure,False->14
0       True
1       True
2       True
3       True
4       True
       ...  
763    False
764     True
765     True
766     True
767     True
Name: SkinThickness, Length: 768, dtype: bool
for SkinThickness,True->733
for SkinThickness,False->35
0      True
1      True
2      True
3      True
4      True
```

```
       ...
763    True
764    True
765    True
766    True
767    True
Name: Insulin, Length: 768, dtype: bool
for Insulin,True->719
for Insulin,False->49
0      True
1      True
2      True
3      True
4      True
       ...
763    True
764    True
765    True
766    True
767    True
Name: BMI, Length: 768, dtype: bool
for BMI,True->760
for BMI,False->8
0       True
1       True
2       True
3       True
4      False
       ...
763     True
764     True
765     True
766     True
767     True
Name: DiabetesPedigreeFunction, Length: 768, dtype: bool
for DiabetesPedigreeFunction,True->739
for DiabetesPedigreeFunction,False->29
0      True
1      True
2      True
3      True
4      True
       ...
763    True
764    True
765    True
766    True
767    True
Name: Age, Length: 768, dtype: bool
for Age,True->759
for Age,False->9
```

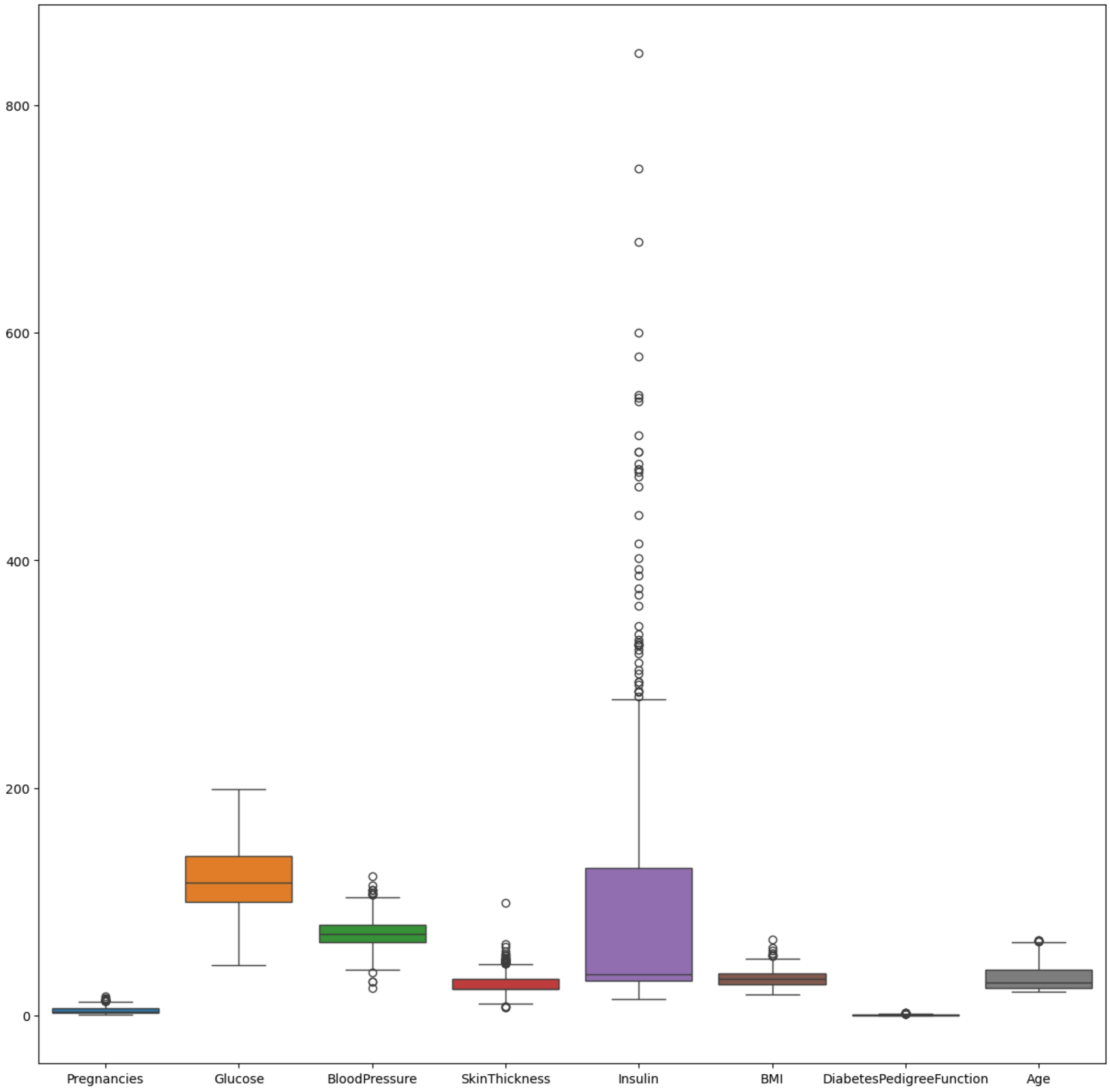
```
/var/folders/8h/zprf7hjs319_78816p34b90c0000gn/T/ipykernel_91019/178433420
6.py:18: FutureWarning: Series.__getitem__ treating keys as positions is d
eprecated. In a future version, integer keys will always be treated as lab
els (consistent with DataFrame behavior). To access a value by position, u
se `ser.iloc[pos]`
  print(f"for {col},{name}->{mask.value_counts()[idx]}")
/var/folders/8h/zprf7hjs319_78816p34b90c0000gn/T/ipykernel_91019/178433420
6.py:18: FutureWarning: Series.__getitem__ treating keys as positions is d
eprecated. In a future version, integer keys will always be treated as lab
els (consistent with DataFrame behavior). To access a value by position, u
se `ser.iloc[pos]`
  print(f"for {col},{name}->{mask.value_counts()[idx]}")
/var/folders/8h/zprf7hjs319_78816p34b90c0000gn/T/ipykernel_91019/178433420
6.py:18: FutureWarning: Series.__getitem__ treating keys as positions is d
eprecated. In a future version, integer keys will always be treated as lab
els (consistent with DataFrame behavior). To access a value by position, u
se `ser.iloc[pos]`
  print(f"for {col},{name}->{mask.value_counts()[idx]}")
/var/folders/8h/zprf7hjs319_78816p34b90c0000gn/T/ipykernel_91019/178433420
6.py:18: FutureWarning: Series.__getitem__ treating keys as positions is d
eprecated. In a future version, integer keys will always be treated as lab
els (consistent with DataFrame behavior). To access a value by position, u
se `ser.iloc[pos]`
  print(f"for {col},{name}->{mask.value_counts()[idx]}")
/var/folders/8h/zprf7hjs319_78816p34b90c0000gn/T/ipykernel_91019/178433420
6.py:18: FutureWarning: Series.__getitem__ treating keys as positions is d
eprecated. In a future version, integer keys will always be treated as lab
els (consistent with DataFrame behavior). To access a value by position, u
se `ser.iloc[pos]`
  print(f"for {col},{name}->{mask.value_counts()[idx]}")
/var/folders/8h/zprf7hjs319_78816p34b90c0000gn/T/ipykernel_91019/178433420
6.py:18: FutureWarning: Series.__getitem__ treating keys as positions is d
eprecated. In a future version, integer keys will always be treated as lab
els (consistent with DataFrame behavior). To access a value by position, u
se `ser.iloc[pos]`
  print(f"for {col},{name}->{mask.value_counts()[idx]}")
/var/folders/8h/zprf7hjs319_78816p34b90c0000gn/T/ipykernel_91019/178433420
6.py:18: FutureWarning: Series.__getitem__ treating keys as positions is d
eprecated. In a future version, integer keys will always be treated as lab
els (consistent with DataFrame behavior). To access a value by position, u
se `ser.iloc[pos]`
  print(f"for {col},{name}->{mask.value_counts()[idx]}")
/var/folders/8h/zprf7hjs319_78816p34b90c0000gn/T/ipykernel_91019/178433420
6.py:18: FutureWarning: Series.__getitem__ treating keys as positions is d
eprecated. In a future version, integer keys will always be treated as lab
els (consistent with DataFrame behavior). To access a value by position, u
se `ser.iloc[pos]`
  print(f"for {col},{name}->{mask.value_counts()[idx]}")
```

In [33]:

```python
fig, ax = plt.subplots(figsize = (15, 15))
sns.boxplot(data = X_outlier_detection, ax=ax)
# plt.savefig('boxPlotPostOutlierRemoval.jpg')
```

Out[33]:

```
<Axes: >
```



Standardization

Standardization and Normalization: Data Preprocessing Techniques

Standardization and normalization are common data preprocessing techniques used in machine learning to ensure that features are on a similar scale. This is important because many machine learning algorithms are sensitive to the scale of the data.

Standardization

Standardization transforms data to have a mean of 0 and a standard deviation of 1. It's often used when the data is normally distributed or when you want to remove the influence of outliers.

Formula:

```
z = (x - μ) / σ
```

Where:

- `z` is the standardized value
- `x` is the original value
- `μ` is the mean of the data
- `σ` is the standard deviation of the data

Min-Max Scaling

Min-max scaling transforms data to a specific range, typically between 0 and 1. It's useful when you want to preserve the relative differences between values.

Formula:

```
x_scaled = (x - min(x)) / (max(x) - min(x))
```

Where:

- `x_scaled` is the scaled value
- `x` is the original value
- `min(x)` is the minimum value of the data
- `max(x)` is the maximum value of the data

When to Use Which:

- **Standardization:** Use when the data is normally distributed or when you want to remove the influence of outliers.
- **Min-Max Scaling:** Use when you want to preserve the relative differences between values, such as in image processing or when dealing with data that has a natural range (e.g., 0 to 100).

Example in Python using scikit-learn:

```python
from sklearn.preprocessing import StandardScaler, MinMaxScaler

# Standardization
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)

# Min-Max Scaling
scaler = MinMaxScaler()
X_scaled = scaler.fit_transform(X)
```

By understanding and applying standardization and normalization, you can improve the performance of your machine learning models and ensure that your data is in a suitable format for training.

Summary

- **Standardization** is used to **center** the data (mean = 0) and **scale** it to unit variance (std = 1).
- `StandardScaler` standardizes data based on the mean and standard deviation, which is best for algorithms requiring normally distributed features.
- `MinMaxScaler` scales data to a **fixed range**, which is ideal when the data doesn't follow a normal distribution and when you need inputs in a specific range (e.g., for neural networks).

Choosing between **StandardScaler** and **MinMaxScaler** depends on the characteristics of your dataset and the requirements of the machine learning algorithm.

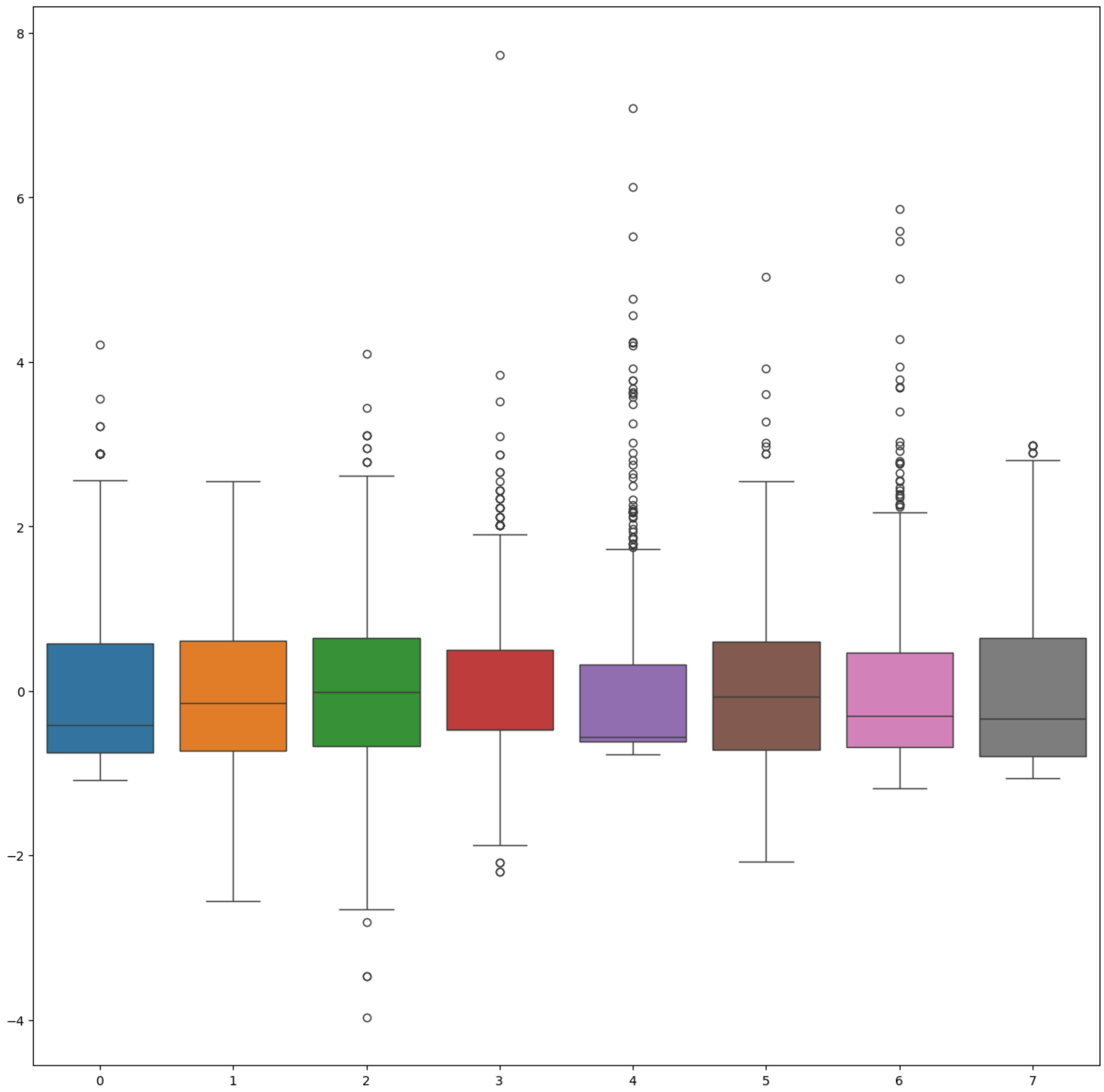
Standard Normal Form -> Mean = 0 and standard deviation = 1

In [34]:
```
from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X_outlier_detection)
```

In [35]:
```
X_scaled
```

Out[35]:
```
array([[ 0.57322173,  0.87008298, -0.01698412, ...,  0.16090077,
         0.46879263,  1.54828125],
       [-1.0797999 , -1.20656984, -0.51093456, ..., -0.85816238,
        -0.36177415, -0.16252742],
       [ 1.23443039,  2.02377899, -0.6755847 , ..., -1.33857787,
         0.60421113, -0.07248486],
       ...,
       [ 0.2426174 , -0.01991109, -0.01698412, ..., -0.91639456,
        -0.68075995, -0.25256998],
       [-1.0797999 ,  0.14490263, -1.00488499, ..., -0.3486308 ,
        -0.36779275,  1.27815356],
       [-1.0797999 , -0.9428679 , -0.18163427, ..., -0.30495667,
        -0.47010895, -0.88286791]])
```

In [36]:
```
fig, ax = plt.subplots(figsize = (15, 15))
sns.boxplot(data = X_scaled, ax=ax)
plt.savefig('boxPlot.jpg')
```



In [37]:
```
dataframe.columns
```

Out[37]:
```
Index(['Pregnancies', 'Glucose', 'BloodPressure', 'SkinThickness', 'Insu
lin',
       'BMI', 'DiabetesPedigreeFunction', 'Age', 'Outcome'],
      dtype='object')
```

In [38]:
```
cols = ['Pregnancies', 'Glucose', 'BloodPressure', 'SkinThickness', 'Insu
```

In [39]:
```
type(X_scaled)
```

Out[39]:
```
numpy.ndarray
```

In [40]:
```
X_scaled = pd.DataFrame(X_scaled, columns=cols)
X_scaled.describe()
```

Out[40]:

	Pregnancies	Glucose	BloodPressure	SkinThickness	Insuli
count	7.590000e+02	7.590000e+02	7.590000e+02	7.590000e+02	7.590000e+0
mean	1.029772e-16	-3.978665e-17	-3.042508e-17	-1.509552e-16	-4.329724e 1
std	1.000659e+00	1.000659e+00	1.000659e+00	1.000659e+00	1.000659e+0
min	-1.079800e+00	-2.558042e+00	-3.968588e+00	-2.200901e+00	-7.684941e 0
25%	-7.491956e-01	-7.286101e-01	-6.755847e-01	-4.729631e-01	-6.126688e 0
50%	-4.185912e-01	-1.517621e-01	-1.698412e-02	-4.729631e-01	-5.607270e 0
75%	5.732217e-01	6.063810e-01	6.416165e-01	4.990017e-01	3.222827e-0
max	4.209869e+00	2.551183e+00	4.099270e+00	7.734740e+00	7.088876e+0

Outlier Handling Technique

Quantile Approach for Removing Outliers

The **quantile approach** is a method used to remove outliers by leveraging statistical percentiles (also known as quantiles). Outliers are data points that lie far away from other observations in the dataset and can negatively affect the performance of machine learning models. The quantile approach removes these outliers by identifying the extreme lower and upper bounds of data based on quantiles.

What are Quantiles?

- **Quantiles** divide a dataset into intervals with equal probabilities.
- For instance:
  - The **25th percentile (Q1)** is the value below which 25% of the data falls.
  - The **75th percentile (Q3)** is the value below which 75% of the data falls.

Common Quantile-Based Metrics:

- **Interquartile Range (IQR)**: Measures the spread of the middle 50% of the data and is calculated as:
  `IQR = Q3 - Q1`
- **Outlier Bounds**:
  - **Lower Bound**: Typically, any data point below `Q1 - 1.5 * IQR` is considered an outlier.
  - **Upper Bound**: Similarly, any data point above `Q3 + 1.5 * IQR` is considered an outlier.

Steps for Removing Outliers Using Quantiles:

1. **Calculate the Quartiles**:

   - **Q1 (25th percentile)**: The value below which 25% of the data falls.
   - **Q3 (75th percentile)**: The value below which 75% of the data falls.
2. **Compute the Interquartile Range (IQR)**:
   - IQR is the difference between the 75th percentile and the 25th percentile (Q3 - Q1).
3. **Define Outlier Boundaries**:
   - **Lower Bound**: `Q1 − 1.5 ∗ IQR`
   - **Upper Bound**: `Q3 + 1.5 ∗ IQR`
4. **Remove Outliers**:
   - Any data point below the lower bound or above the upper bound is considered an outlier and can be removed.

Example in Python:

```
import numpy as np
import pandas as pd

# Sample data
data = {'values': [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 50, 100]}
df = pd.DataFrame(data)

# Calculate Q1 (25th percentile) and Q3 (75th percentile)
Q1 = df['values'].quantile(0.25)
Q3 = df['values'].quantile(0.75)

# Calculate IQR
IQR = Q3 - Q1

# Define the lower and upper bounds
lower_bound = Q1 - 1.5 * IQR
upper_bound = Q3 + 1.5 * IQR

# Filter out the outliers
filtered_df = df[(df['values'] >= lower_bound) & (df['values'] <=
upper_bound)]

print("Data after removing outliers:")
print(filtered_df)
```

Why Use the Quantile Approach?

- **Non-parametric**: The quantile approach does not make assumptions about the underlying distribution of the data (e.g., normality). It can be applied to a variety of datasets.
- **Robust to Outliers**: The IQR method is robust to outliers since it focuses on the middle 50% of the data.
- **Simple and Intuitive**: The method is easy to understand and implement, providing a straightforward way to handle extreme values.

When to Use the Quantile Approach:

- **Data with Skewness**: Suitable for datasets with skewed distributions because it does not assume normality.
- **Small Number of Outliers**: Useful when the number of outliers is small and removing them does not significantly reduce the size of the dataset.
- **Non-Gaussian Data**: Works well with non-Gaussian (non-normal) distributions where traditional z-score or standard deviation-based methods may fail.

When Not to Use:

- **Large Proportion of Outliers**: If the dataset has a large number of outliers, this method could remove too much data, leading to loss of valuable information.
- **Multimodal Data**: It may not work as well for multimodal data (data with multiple peaks) where the outliers might actually represent different subgroups rather than noise.

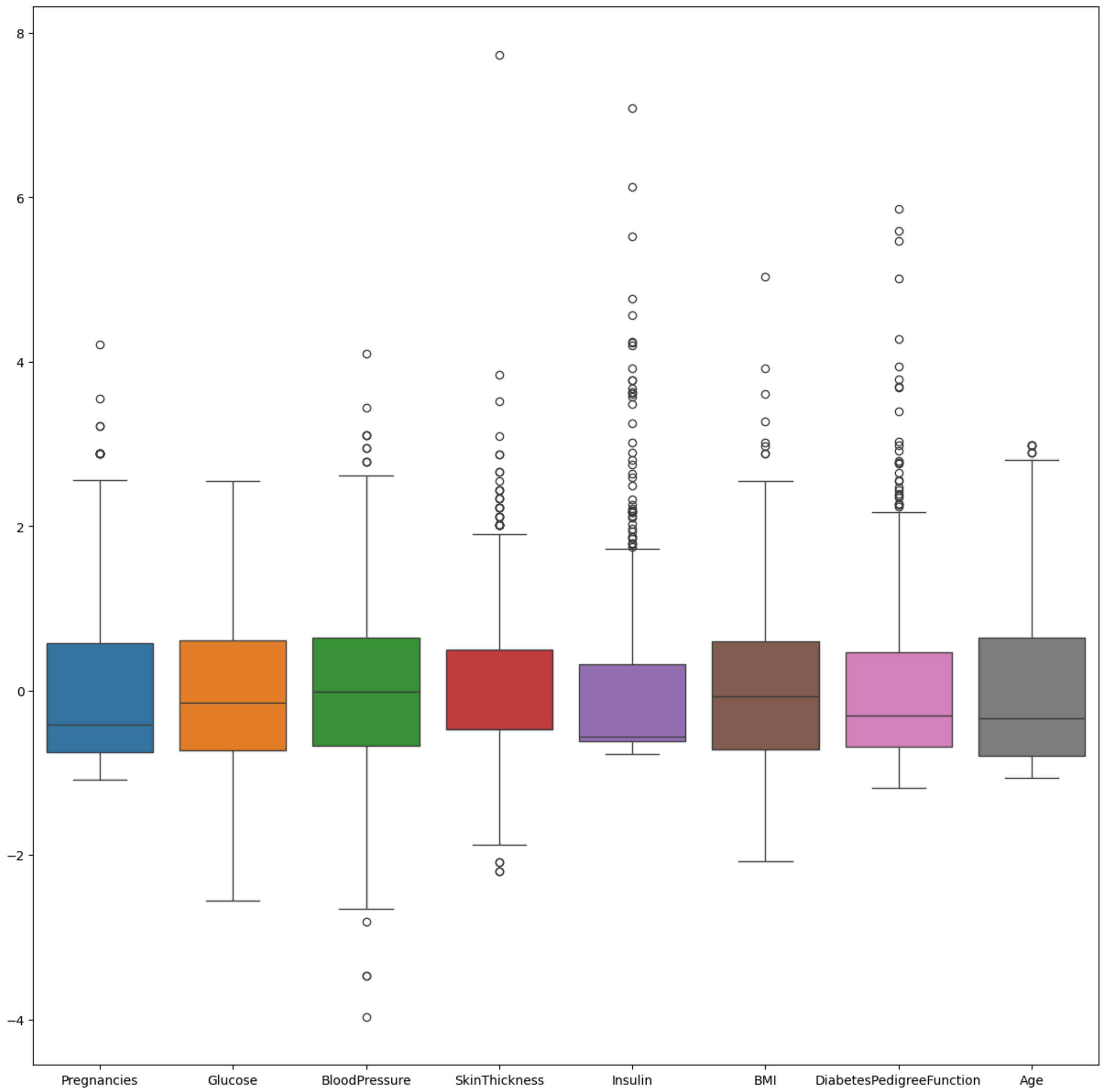
Summary

The quantile approach to outlier removal is an effective method that relies on the **interquartile range (IQR)** to identify and filter out extreme data points. It is particularly robust for skewed or non-normal distributions and offers a simple yet powerful way to handle outliers.

- Approach 2 of quantiles to remove the outliers
- Handling of imbalanced data

In [41]:

```
fig, ax = plt.subplots(figsize = (15, 15))
sns.boxplot(data = X_scaled, ax=ax)
plt.savefig('boxPlot.jpg')
```



In [42]:
```
y_outlier_detection.shape
```

Out[42]:
```
(759,)
```

In [43]:
```
y_outlier_detection.value_counts()
```

Out[43]:
```
Outcome
0    493
1    266
Name: count, dtype: int64
```

Concluding:

- Detection of the outliers
- Normalization via StandardScaler Form & Why it is important(reduce the biasness in the model)

Approach 2: Quantiles

In [44]:
```
X_scaled.reset_index(drop=True, inplace=True)
y_outlier_detection.reset_index(drop=True, inplace=True)
```

In [45]:

```
q = X_scaled['Insulin'].quantile(.95)
mask = X_scaled['Insulin'] < q
print(mask.value_counts())
dataNew = X_scaled[mask]
y_outlier_detection = y_outlier_detection[mask]
```

```
Insulin
True     721
False     38
Name: count, dtype: int64
```

In [46]:

```
dataNew.shape
```

Out[46]:

```
(721, 8)
```

In [47]:

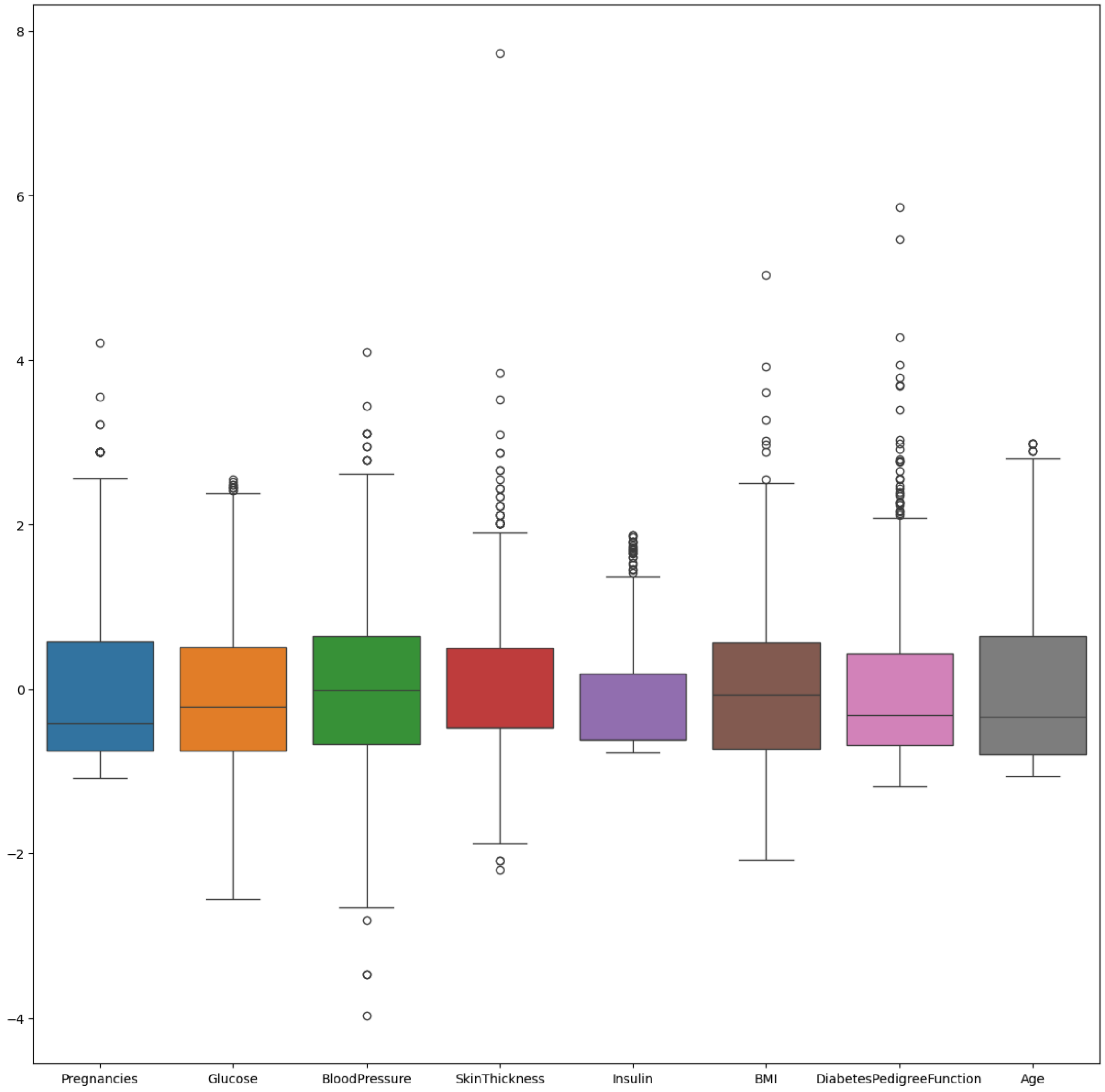
```
y_outlier_detection.shape
```

Out[47]:

```
(721,)
```

In [48]:

```
fig, ax = plt.subplots(figsize = (15, 15))
sns.boxplot(data = dataNew, ax=ax)
plt.savefig('boxPlot.jpg')
```



Model Training

Splitting of data into training and testing

```
In [49]: from sklearn.model_selection import train_test_split
         X_train, X_test, y_train, y_test = train_test_split(dataNew, y_outlier_de
```

```
In [50]: X_train.shape
```

```
Out[50]: (483, 8)
```

```
In [51]: X_test.shape
```

```
Out[51]: (238, 8)
```

Managing Imbalance Data

Data Imbalancing

- Oversampling : Minority Class and increase that number to the majority class
- Undersampling : Majority class and decrease that number to the minority class
- SMOTE : Synthetic data and increase the number of samples to the majority class

Imbalanced Data

Imbalanced data refers to a situation where the classes in a dataset are not represented equally. In other words, one class (often called the **minority class**) has significantly fewer observations than another class (often called the **majority class**). This issue is common in classification problems like fraud detection, spam detection, or rare disease prediction, where the number of positive examples is much lower than the number of negative examples.

Challenges of Imbalanced Data:

1. **Biased Model Performance**: Algorithms tend to be biased towards the majority class, leading to poor performance in detecting the minority class.
2. **Skewed Evaluation Metrics**: Common metrics like accuracy can be misleading because a model can achieve high accuracy simply by predicting the majority class correctly while ignoring the minority class.

Techniques to Handle Imbalanced Data

There are various techniques to handle imbalanced data, both at the **data level** and the **algorithmic level**. Let's explore the key techniques with examples:

1. Resampling Techniques (Data Level)

Resampling involves modifying the dataset to balance the number of observations in each class. There are two main approaches: **undersampling** and **oversampling**.

a) Oversampling

In oversampling, you increase the number of samples in the minority class to make the dataset more balanced. This can be done by duplicating samples or generating synthetic samples.

- **Random Oversampling**: This method randomly duplicates examples from the minority class until the class sizes are balanced.
- **SMOTE (Synthetic Minority Over-sampling Technique)**: SMOTE generates synthetic samples by interpolating between existing examples in the minority class. Instead of simply duplicating minority class instances, SMOTE creates new instances by drawing random points between existing data points.

Example in Python using SMOTE:

```
from imblearn.over_sampling import SMOTE
from sklearn.model_selection import train_test_split
from sklearn.datasets import make_classification

# Generate imbalanced data
X, y = make_classification(n_samples=1000, n_features=20,
n_informative=2,
                           n_redundant=10,
n_clusters_per_class=1, weights=[0.9],
                           flip_y=0, random_state=42)

# Apply SMOTE to balance the dataset
smote = SMOTE(random_state=42)
X_resampled, y_resampled = smote.fit_resample(X, y)

print(f"Original class distribution: {dict(zip(*np.unique(y,
return_counts=True)))}")
print(f"Resampled class distribution:
{dict(zip(*np.unique(y_resampled, return_counts=True)))}")
```

b) Undersampling

Undersampling involves reducing the number of samples in the majority class to balance the dataset. This approach is useful when you have a lot of redundant data in the majority class.

- **Random Undersampling**: Randomly removes samples from the majority class to balance the classes.
- **Tomek Links**: A more sophisticated method that removes pairs of examples (one from the majority class and one from the minority class) that are very close together but belong to different classes.

Example in Python using Random Undersampling:

```
from imblearn.under_sampling import RandomUnderSampler

# Apply Random Undersampling
undersample = RandomUnderSampler(random_state=42)
X_resampled, y_resampled = undersample.fit_resample(X, y)

print(f"Original class distribution: {dict(zip(*np.unique(y,
return_counts=True)))}")
print(f"Resampled class distribution:
{dict(zip(*np.unique(y_resampled, return_counts=True)))}")
```

c) Combined Approach (SMOTE + ENN)

Combines **SMOTE** with **Edited Nearest Neighbors (ENN)**, a method that removes noisy or misclassified examples after applying oversampling.

---

2. Algorithm-Level Techniques

These techniques modify algorithms to account for class imbalance during training.

a) Class Weighting

Many machine learning algorithms, such as Logistic Regression, SVM, and Decision Trees, allow you to assign higher weights to the minority class. This tells the algorithm to give more importance to the minority class when building the model.

Example in Python using Logistic Regression:

```
from sklearn.linear_model import LogisticRegression

# Use class_weight='balanced' to handle imbalanced data
model = LogisticRegression(class_weight='balanced',
random_state=42)
model.fit(X_train, y_train)
```

b) Balanced Random Forest

Random Forests can be adapted to handle imbalanced data by assigning different weights to each class or by undersampling the majority class during the bootstrap sampling of each tree.

---

3. Evaluation Metrics for Imbalanced Data

Using appropriate evaluation metrics is crucial for dealing with imbalanced data. Accuracy is not a good measure for imbalanced datasets. Some of the better metrics include:

- **Precision**: Measures the accuracy of positive predictions (minority class).
- **Recall**: Measures how many of the actual positives were predicted correctly.
- **F1-Score**: Harmonic mean of precision and recall, providing a balance between the two.
- **ROC-AUC (Area Under the ROC Curve)**: Measures the trade-off between true positives and false positives across different threshold levels.

- **Precision-Recall AUC**: Focuses on the performance with respect to the minority class.

Example in Python using classification metrics:

```
from sklearn.metrics import classification_report

# Evaluate model performance
y_pred = model.predict(X_test)
print(classification_report(y_test, y_pred))
```

Example Use Case

Imagine you are building a model for **fraud detection**. Fraudulent transactions (minority class) might represent 1% of the total transactions, while legitimate transactions (majority class) represent 99%. A simple classifier might predict "legitimate" for every transaction and achieve 99% accuracy, but it would fail to detect any fraud. Here's how to handle this using techniques:

1. **Resample the data** using SMOTE to generate synthetic fraud samples.
2. **Apply class weighting** to give more importance to fraud detection during model training.
3. **Use precision, recall, and F1-score** for evaluation to ensure the minority class (fraudulent transactions) is being correctly identified.

Conclusion

Handling imbalanced data requires careful consideration of both data-level and algorithm-level techniques. Resampling methods (oversampling and undersampling), adjusting class weights, and using appropriate evaluation metrics help mitigate the negative effects of imbalanced datasets, ensuring that minority classes are properly represented in the model's predictions.

```
In [52]: y_train.value_counts()
```

```
Out[52]: Outcome
         0    318
         1    165
         Name: count, dtype: int64
```

SMOTE Technique

```
In [54]: from imblearn.over_sampling import SMOTE
         smote = SMOTE(random_state=42)
         X_train_resampled, y_train_resampled = smote.fit_resample(X_train, y_trai

         # Check resampled class distribution
         print("\nResampled class distribution:")
         print(pd.Series(y_train_resampled).value_counts())
```

```
Resampled class distribution:
Outcome
0    318
1    318
Name: count, dtype: int64
```

Implement Logistic Regression

In [55]:
```
from sklearn.linear_model import LogisticRegression
classification = LogisticRegression()
classification.fit(X_train_resampled, y_train_resampled)
```

Out[55]:
```
▾ LogisticRegression
LogisticRegression()
```

Model Predictions

In [56]:
```
y_predictions = classification.predict(X_test)
print(y_predictions)
```

```
[0 1 0 0 1 1 0 1 1 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 1 0 1 0 1 1 0 0 1 1 0 0 0
 0 1 0 1 0 1 1 0 0 1 0 0 0 1 0 0 0 0 0 0 0 1 1 0 0 1 0 0 1 0 1 0 0 1 0 0 0
 0 0 1 1 1 1 1 0 1 1 0 1 0 0 1 0 0 1 0 1 0 0 0 0 1 0 0 1 0 1 0 0 0 1 1 1 1
 0 0 1 1 0 1 0 0 0 1 1 0 0 0 0 0 0 0 0 1 0 1 0 0 1 0 1 1 0 0 1 1 0 0 1 0 1
 1 0 0 0 0 1 1 1 1 0 1 0 0 1 1 1 1 1 1 0 0 0 1 0 0 0 0 0 1 0 1 1 0 0 0 0 0
 1 1 1 1 0 0 1 0 0 0 0 1 0 0 1 0 1 0 1 0 0 0 1 0 0 1 0 0 1 0 0 0 0 1 1 0 0
 0 0 1 0 1 1 0 0 1 0 0 1 1 1 1 1]
```

Model Evaluation

In [57]:
```
from sklearn.metrics import accuracy_score
accuracy_score(y_test, y_predictions)
```

Out[57]:
```
0.7478991596638656
```

Healthcare: Recall is very important metric

In [58]:
```
from sklearn.metrics import classification_report
target_names = ['Non-Diabetic', 'Diabetic']
print(classification_report(y_test, y_predictions, target_names=target_na
```

```
              precision    recall  f1-score   support

Non-Diabetic       0.85      0.76      0.80       159
    Diabetic       0.60      0.72      0.66        79

    accuracy                           0.75       238
   macro avg       0.72      0.74      0.73       238
weighted avg       0.76      0.75      0.75       238
```

Save Model

In [59]:
```
import pickle
pickle.dump(classification, open("classification_model.pkl", "wb"))
```

In [60]:
```
classification_model = pickle.load(open("classification_model.pkl", "rb"))
classification_model.predict(X_test)
```

Out[60]:
```
array([0, 1, 0, 0, 1, 1, 0, 1, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 1, 0, 1, 0, 1, 1, 0, 0, 1, 1, 0, 0, 0, 0, 1, 0, 1, 0, 1, 1,
       0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 1, 0, 0, 1,
       0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 0, 1, 1, 0, 1, 0, 0,
       1, 0, 0, 1, 0, 1, 0, 0, 0, 0, 1, 0, 0, 1, 0, 1, 0, 0, 0, 1, 1, 1,
       1, 0, 0, 1, 1, 0, 1, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0,
       1, 0, 0, 1, 0, 1, 1, 0, 0, 1, 1, 0, 0, 1, 0, 1, 1, 0, 0, 0, 0, 1,
       1, 1, 1, 0, 1, 0, 0, 1, 1, 1, 1, 1, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0,
       1, 0, 1, 1, 0, 0, 0, 0, 0, 1, 1, 1, 1, 0, 0, 1, 0, 0, 0, 0, 1, 0,
       0, 1, 0, 1, 0, 1, 0, 0, 0, 1, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 1, 1,
       0, 0, 0, 0, 1, 0, 1, 1, 0, 0, 1, 0, 0, 1, 1, 1, 1, 1])
```

Model Training: KNNClassifier Model

In [61]:
```
from sklearn.neighbors import KNeighborsClassifier, KNeighborsRegressor
from sklearn.metrics import classification_report, confusion_matrix
knn = KNeighborsClassifier()
```

In [62]:
```
knn.fit(X_train_resampled, y_train_resampled)
```

Out[62]:
```
▾ KNeighborsClassifier
KNeighborsClassifier()
```

Model Prediction

In [63]:
```
y_prediction_knn = knn.predict(X_test)
y_prediction_knn
```

Out[63]:
```
array([0, 1, 0, 0, 0, 1, 0, 1, 1, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0,
       0, 0, 1, 0, 1, 0, 0, 1, 0, 1, 1, 1, 0, 1, 0, 0, 1, 0, 1, 0, 1, 1,
       0, 0, 1, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 1, 1, 0, 0, 1,
       1, 1, 1, 1, 1, 0, 1, 0, 1, 0, 1, 1, 0, 1, 1, 0, 0, 1, 0, 0, 1, 0,
       1, 0, 0, 1, 0, 1, 0, 0, 1, 0, 1, 0, 0, 0, 0, 1, 0, 0, 1, 1, 0, 1,
       1, 0, 0, 1, 1, 0, 1, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0,
       1, 1, 0, 1, 0, 1, 1, 0, 0, 1, 1, 0, 0, 1, 0, 1, 1, 0, 1, 0, 0, 1,
       0, 1, 1, 1, 1, 0, 0, 1, 1, 1, 1, 1, 0, 1, 0, 0, 1, 0, 0, 0, 0, 1,
       1, 1, 1, 1, 1, 0, 0, 1, 0, 1, 0, 0, 1, 0, 0, 1, 1, 1, 0, 1, 1, 1,
       0, 1, 0, 1, 0, 1, 1, 0, 0, 1, 0, 0, 1, 1, 1, 1, 0, 1, 1, 0, 1, 1,
       0, 1, 0, 0, 1, 0, 1, 1, 0, 0, 1, 1, 0, 0, 1, 1, 1, 1])
```

In [64]:
```
print("Confusion Matrix")
print(confusion_matrix(y_test, y_prediction_knn))
```

```
Confusion Matrix
[[103  56]
 [ 19  60]]
```

In [65]:
```
print("Classification Report")
print(classification_report(y_test, y_prediction_knn))
```

```
Classification Report
              precision    recall  f1-score   support

           0       0.84      0.65      0.73       159
           1       0.52      0.76      0.62        79

    accuracy                           0.68       238
   macro avg       0.68      0.70      0.67       238
weighted avg       0.74      0.68      0.69       238
```

Data Modeling: Implementation of Naive Bayes Classifier

In [66]:

```
from sklearn.naive_bayes import GaussianNB
model_gaussian_naive_bayes = GaussianNB()
model_gaussian_naive_bayes.fit(X_train_resampled, y_train_resampled)
```

Out[66]:

```
▾ GaussianNB

GaussianNB()
```

In [67]:

```
y_predict_gaussian_naive_bayes = model_gaussian_naive_bayes.predict(X_tes
print(y_predict_gaussian_naive_bayes)
```

```
[0 1 0 0 1 1 0 1 1 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 1 0 1 0 1 1 0 0 1 1 1 0 0
 0 1 0 1 1 1 1 0 0 1 0 0 0 0 0 0 0 0 0 0 1 1 1 0 0 0 0 0 1 0 1 0 0 1 0 0 0
 0 0 1 1 1 1 1 0 1 1 0 1 0 0 0 0 0 1 0 1 0 0 0 0 1 0 0 1 0 1 0 0 0 1 1 1 1
 0 0 1 1 0 1 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 1 0 0 1 0 0 1 0 0 1 1 0 0 1 0 1
 1 0 0 0 0 1 1 1 1 0 1 0 0 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 1 0 1 1 0 0 0 0 0
 1 1 0 1 0 0 1 0 1 0 1 1 0 0 1 0 1 1 0 0 0 0 1 0 0 0 1 0 1 0 0 0 0 1 1 0 1
 0 0 1 0 1 1 0 0 0 1 0 1 1 1 1 0]
```

In [68]:

```
print("Confusion Matrix")
print(confusion_matrix(y_test, y_predict_gaussian_naive_bayes))
```

```
Confusion Matrix
[[119  40]
 [ 27  52]]
```

In [69]:

```
print("Classification Report")
print(classification_report(y_test, y_predict_gaussian_naive_bayes))
```

```
Classification Report
              precision    recall  f1-score   support

           0       0.82      0.75      0.78       159
           1       0.57      0.66      0.61        79

    accuracy                           0.72       238
   macro avg       0.69      0.70      0.69       238
weighted avg       0.73      0.72      0.72       238
```

In [70]:

```
accuracy_score(y_test, y_predict_gaussian_naive_bayes)
```

Out[70]: 0.7184873949579832