K-Nearest Neighbors (KNN)

K-Nearest Neighbors (KNN) is a simple, instance-based, non-parametric machine learning algorithm used for both classification and regression tasks. The basic idea is to predict the label or value of a new data point based on the labels or values of the closest data points (neighbors) in the feature space.

How KNN Works

KNN works by following these steps:

1. **Select the number of neighbors (K):** Choose a value of K, which is the number of nearest neighbors to consider for making a prediction.
2. **Compute distances:** Calculate the distance between the new data point and all other points in the dataset using a distance metric (typically Euclidean distance, though other metrics such as Manhattan, Minkowski, etc., can be used).
3. **Identify neighbors:** Find the K points that are closest to the new data point based on the computed distances.
4. **Make a prediction:**
   - **Classification:** Assign the most common class (mode) among the K nearest neighbors to the new data point.
   - **Regression:** Assign the average (mean) value of the K nearest neighbors to the new data point.

KNN for Classification: Example

Let's consider an example where we classify whether a flower is a **setosa** or **versicolor** based on its petal width and length.

- **Dataset:**
  - **Features:** Petal width and length.
  - **Labels:** `Setosa` and `Versicolor`.
- **New Data Point:** Suppose we have a new flower with petal width = 1.5 and length = 4.5, and we want to classify whether it is `Setosa` or `Versicolor`.

Steps:

1. **Choose K:** Let's select (K = 3).
2. **Calculate Distances:** Calculate the Euclidean distance between the new flower and all the other flowers in the dataset:
   `d(x, x_i) = sqrt(sum((x_j - x_{i,j})^2 for j in range(n)))`
3. **Find the 3 Nearest Neighbors:** Based on the calculated distances, identify the 3 nearest flowers to the new data point.
4. **Classify:** Take a majority vote among the 3 nearest neighbors. If 2 out of the 3 neighbors are classified as `Setosa`, and 1 is classified as `Versicolor`, the

new flower would be classified as `Setosa`.

KNN for Regression: Example

Now, let's consider a regression problem where we want to predict the house price based on features like the number of rooms, lot size, and proximity to a city center.

- **Dataset:**
  - **Features:** Number of rooms, lot size, distance from city center.
  - **Labels:** House prices.
- **New Data Point:** Suppose we have a new house with 3 rooms, a lot size of 500 sq meters, and it is 10 km from the city center. We want to predict its price.

Steps:

1. **Choose K:** Let's select (K = 3).
2. **Calculate Distances:** Compute the Euclidean distance between the new house and all the other houses in the dataset based on their features.
3. **Find the 3 Nearest Neighbors:** Identify the 3 houses that are closest to the new house.
4. **Predict the Price:** Take the average of the prices of the 3 nearest houses. Suppose the prices of the nearest houses are $200,000$,220,000, and $210,000. Then the predicted price would be:
   ```
   Predicted Price = (200000 + 220000 + 210000) / 3 = 210000
   ```

Key Points to Consider

1. **Distance Metric:** The choice of distance metric impacts KNN performance. Euclidean distance is the most common, but for categorical data, Hamming distance might be used, and for Manhattan distance, L1 norm can be used.

2. **Value of K:**
   - Small values of K (e.g., (K = 1)) make the model sensitive to noise in the dataset but provide more flexible decision boundaries.
   - Large values of K smooth out predictions but may blur the distinctions between different classes.
3. **Scaling/Normalization:** Since KNN relies on distance calculations, it's important to scale or normalize the feature values to avoid one feature dominating the distance calculation. For instance, features with larger ranges can distort the distances unless the data is scaled.

4. **Lazy Learner:** KNN is a **lazy learning algorithm**, meaning it doesn't learn a model during the training phase. All computation is deferred until prediction, which can lead to high computational cost when the dataset is large.

Advantages of KNN

- **Simplicity:** Easy to implement and understand.
- **No Training Phase:** No explicit model training is required, making KNN suitable for dynamic datasets where data frequently changes.
- **Versatility:** Can be used for both classification and regression tasks.

Disadvantages of KNN

- **Computationally Expensive:** During prediction, KNN requires calculating the distance between the query point and all other points in the dataset, which can be slow for large datasets.
- **Memory Intensive:** KNN stores all training data, which can consume a lot of memory.
- **Sensitive to Irrelevant Features:** If irrelevant features are included, they can distort the distance metric and lead to poor performance. Therefore, feature selection is crucial.
- **Imbalanced Datasets:** KNN struggles with imbalanced datasets where one class dominates, as the majority class will be more likely to be chosen.

Conclusion

KNN is a versatile algorithm that can be used for both classification and regression. Its performance heavily depends on the choice of K, distance metric, and scaling of data. It is best suited for smaller datasets, as it requires a lot of computational resources for large datasets. However, its simplicity and effectiveness make it a popular choice for many applications.

Distance Metrics in KNN

The distance metric is a crucial aspect of KNN, as it determines how "closeness" between data points is calculated.

1. Euclidean Distance

This is the most commonly used distance metric in KNN, which calculates the straight-line distance between two points in a multidimensional space.

- **Formula:**
  `d(p, q) = sqrt(sum((p_i - q_i)^2 for i in range(n)))`
  where (p) and (q) are the points (or vectors) with (n) features.

2. Manhattan Distance

Manhattan distance, also called L1 distance or city block distance, calculates the distance by summing the absolute differences of their coordinates.

- **Formula:**
  `d(p, q) = sum(|p_i - q_i| for i in range(n))`

3. Hamming Distance

Hamming distance is used for categorical variables and counts the number of positions where the corresponding elements differ.

- **Formula:**
  `d(p, q) = sum(1 if p_i != q_i else 0 for i in range(n))`

4. Minkowski Distance

Minkowski distance is a generalization of both Euclidean and Manhattan distances.

- **Formula:**
  `d(p, q) = (sum(|p_i - q_i|^r for i in range(n)))^(1/r)`

For Euclidean distance, (r = 2), and for Manhattan distance, (r = 1).

Hyperparameters in KNN

1. K (Number of Neighbors)

- **Role:** Determines how many neighbors are considered when making predictions.
- **Choosing K:** The optimal value of (K) can be determined using cross-validation. Often, a good starting point is (K = \sqrt{n}), where (n) is the number of data points.

2. Distance Metric

- **Role:** Defines how the distance between points is measured. Common choices include Euclidean, Manhattan, and Hamming distances.

3. Weights

- **Role:** Assigns weights to the neighbors. You can use uniform weights, where each neighbor is weighted equally, or distance-based weights, where closer neighbors have more influence.

Choosing the Value of K

1. **Cross-Validation:** Cross-validation is often used to select the best value of (K).
2. **Empirical Rule:** A commonly used rule of thumb is to set (K) to the square root of the number of samples:

   ```
   K = sqrt(n)
   ```

3. **Error Rates:** Larger values of (K) lead to smoother decision boundaries but increase bias (underfitting), while smaller values decrease bias but increase variance (overfitting).

Conclusion

KNN's performance is influenced by the choice of distance metric, the number of neighbors (K), and how the neighbors are weighted. Cross-validation is a robust way to select the optimal value for these hyperparameters, ensuring that the model performs well on unseen data.

KNN: Pros and Cons

Pros of KNN

1. **No Strong Assumptions:**
    - **Explanation:** Unlike other algorithms like Linear Regression or SVM, KNN doesn't make strong assumptions about the data. It is a non-parametric algorithm, meaning it doesn't assume an underlying probability distribution or model structure.
    - **Benefit:** This makes KNN highly flexible and able to work well with data that doesn't fit into predefined assumptions, such as nonlinear data or complex decision boundaries.
2. **Simple to Implement and Understand:**
    - **Explanation:** KNN is conceptually simple and easy to understand. The idea of finding the closest points to make a decision based on the majority is intuitive.
    - **Benefit:** Easy implementation in applications, even with minimal data preparation or preprocessing.
3. **Adaptable to Classification and Regression:**
    - **Explanation:** KNN can be used for both classification and regression tasks, making it versatile.
    - **Benefit:** Useful in various domains and applications like pattern recognition and data mining.
4. **Low Training Cost:**
    - **Explanation:** As a lazy learner, KNN doesn’t involve any model training phase. All the computation happens at prediction time.
    - **Benefit:** No expensive computations are required to build a model beforehand. You only need to store the data points.

Cons of KNN

1. **Lazy Learner:**
    - **Explanation:** KNN is considered a lazy learner because it doesn't learn a model during the training phase. It simply memorizes the training data and

makes predictions during the testing phase by calculating distances to the training data.
- **Drawback:** The prediction phase can be very slow, especially with large datasets, because KNN has to compute distances between a test point and every single training point. This leads to high computational cost during prediction, which makes it inefficient for real-time applications.

2. **Choice of K (Optimal Value):**
   - **Explanation:** The accuracy of KNN heavily depends on the choice of (K) (number of neighbors).
   - **Drawback:** Too small a (K) leads to high variance (overfitting), while too large a (K) leads to high bias (underfitting). It is often necessary to use techniques like cross-validation to find the optimal value of (K).
3. **High Memory Requirement:**
   - **Explanation:** Since KNN needs to store all the training data, it requires a significant amount of memory, especially with large datasets.
   - **Drawback:** This can make KNN impractical for memory-constrained systems or for handling very large datasets.
4. **Sensitive to Outliers and Noisy Data:**
   - **Explanation:** KNN is sensitive to the presence of noisy data points or outliers because they can skew the classification or regression results.
   - **Drawback:** Preprocessing steps like outlier detection, noise removal, and scaling become crucial to improving KNN's performance.
5. **Scaling and Normalization Required:**
   - **Explanation:** KNN relies on distance measures, which means features with larger magnitudes will dominate the results. Therefore, data needs to be scaled or normalized before applying KNN.
   - **Drawback:** Preprocessing can become an additional step in the workflow.

Applications of KNN

1. **Recommendation Systems:**
   - KNN is used in recommendation systems to find users or items that are similar to a given user or item. By calculating the distance between a user and others based on preferences, KNN can recommend items that similar users liked.
2. **Image Recognition:**
   - KNN is applied in image recognition tasks to classify images based on features extracted from the images. For example, handwritten digit recognition using KNN on image pixel values.
3. **Medical Diagnosis:**

- In medical diagnosis, KNN can be used to classify patients based on the similarity of their symptoms or test results to other known cases, aiding in diagnosing diseases.

4. **Anomaly Detection:**
    - KNN is effective for anomaly detection, where the algorithm can detect outliers or unusual data points based on their distance from the majority of the data.
5. **Pattern Recognition:**
    - KNN is commonly used in pattern recognition tasks, such as text recognition and speech recognition, where it helps in categorizing text or audio data into pre-defined categories.

SMOTE (Synthetic Minority Over-sampling Technique)

SMOTE is a popular technique used to address imbalanced datasets, where one class significantly outnumbers the other(s). In such cases, KNN's performance can suffer because the majority class dominates the decision boundaries. SMOTE helps by synthetically generating new data points for the minority class.

How SMOTE Works:

1. **Identify Minority Class Samples:**
    - SMOTE identifies data points from the minority class that are under-represented.
2. **KNN on Minority Class:**
    - For each minority class data point, SMOTE identifies its K-nearest neighbors from the same minority class using the KNN algorithm.
3. **Create Synthetic Data:**
    - SMOTE creates synthetic data points by interpolating between the original minority class data point and its neighbors. This is done by selecting a random point along the line connecting the two points.
    - **Formula for Generating Synthetic Data:**

      ```
      new_sample = original_sample + (neighbor_sample - original_sample) * random(0, 1)
      ```
4. **Add Synthetic Samples to Dataset:**
    - The newly generated synthetic data points are then added to the dataset, increasing the representation of the minority class.

Advantages of SMOTE:

- Reduces bias in favor of the majority class by creating a balanced dataset.
- Helps improve the performance of classifiers on imbalanced data.

Disadvantages of SMOTE:

- It can introduce noise by generating synthetic data points that might not follow the true distribution of the minority class.
- If applied without careful tuning, it might lead to overfitting, as synthetic samples might be very similar to the original ones.

Summary

- **KNN Pros:** Works without strong assumptions, simple implementation, adaptable to both classification and regression, and has a low training cost.
- **KNN Cons:** Slow during prediction, sensitive to the choice of (K), requires high memory, sensitive to noise, and requires data scaling.
- **Applications of KNN:** Widely used in recommendation systems, image recognition, medical diagnosis, anomaly detection, and pattern recognition.
- **SMOTE:** Balances imbalanced datasets by generating synthetic minority class samples through interpolation between minority class neighbors.

In K-Nearest Neighbors (KNN), finding the nearest neighbors can be computationally expensive, especially with large datasets. To speed up the search, two specialized data structures, **KD-Tree** and **Ball Tree**, are commonly used to efficiently partition the data space.

1. KD-Tree (K-Dimensional Tree)

Overview:

- A **KD-Tree** is a binary tree used for organizing points in a k-dimensional space.
- It is particularly efficient when the number of dimensions (k) is low, typically less than 20.

How KD-Tree Works:

1. **Recursive Partitioning:**
    - The space is recursively split along one of the data dimensions (axes) at each level of the tree.
    - At each node, the dataset is split by a hyperplane that is perpendicular to one of the coordinate axes (i.e., along one of the features).
    - The splitting dimension alternates between the k dimensions as you go down each level of the tree.
2. **Building the Tree:**
    - The first level splits based on the first dimension (e.g., (x_1)), the second level splits based on the second dimension (e.g., (x_2)), and so on, cycling through all dimensions.

- For example, if the data is 2D, the first split might be along the x-axis, and the second split might be along the y-axis.

3. **Querying the KD-Tree:**

   - To find the nearest neighbors, the KD-Tree algorithm prunes large portions of the search space that are unlikely to contain closer points than the current best-known candidate.
   - The algorithm traverses the tree and checks only the regions of the space that could contain a closer point to the query.

Advantages of KD-Tree:

- **Efficient for low-dimensional data:** Can significantly reduce the computational cost of finding the nearest neighbors in low dimensions.
- **Pruning:** Reduces the search space for KNN, making queries faster than brute-force search.

Disadvantages of KD-Tree:

- **Not effective for high-dimensional data:** As the number of dimensions increases, the efficiency of KD-Tree diminishes due to the curse of dimensionality, making it less effective for datasets with more than ~20 dimensions.

---

2. Ball Tree

Overview:

- A **Ball Tree** is another type of binary tree used for nearest neighbor searches, designed to handle higher-dimensional data more effectively than KD-Trees.

How Ball Tree Works:

1. **Spherical Partitioning:**

   - Unlike KD-Trees, which split the data space using axis-aligned hyperplanes, Ball Trees partition the space using hyperspheres (balls).
   - Each node in the tree contains a ball, which is a region of the space that encloses a subset of the data points.

2. **Building the Tree:**

   - At each node, the data points are divided into two smaller balls (spheres), where each ball encloses a portion of the points based on their proximity to a centroid or median point.
   - The goal is to minimize the overlap between balls while maximizing the separation of points between different balls.

3. **Querying the Ball Tree:**

- During the nearest neighbor search, the algorithm examines only those balls that could potentially contain points closer than the current best candidate.
- The search is conducted recursively, pruning out entire balls that cannot contain closer neighbors.

Advantages of Ball Tree:

- **Handles high-dimensional data better:** Ball Trees are generally more efficient for higher-dimensional data than KD-Trees, as they are not restricted to splitting along the axes.
- **More flexibility in partitioning:** Because it uses spherical partitions, Ball Trees are better suited for datasets with irregular distributions.

Disadvantages of Ball Tree:

- **More complex:** Ball Trees can be more complex to implement and maintain than KD-Trees, especially with a high-dimensional dataset.
- **Slower for very low dimensions:** In low-dimensional spaces, KD-Trees are typically faster due to their simplicity.

Choosing Between KD-Tree and Ball Tree

- **Low Dimensions (<20 dimensions):** KD-Trees are generally preferred as they are simpler and faster for small to moderate numbers of dimensions.
- **High Dimensions (>20 dimensions):** Ball Trees often perform better due to their ability to handle the curse of dimensionality more effectively.

KNN 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 [3]:
```
dataframe = pd.read_csv("./diabetes.csv")
dataframe.head()
```

Out[3]:

	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 [4]:
```
dataframe.columns
```

Out[4]:
```
Index(['Pregnancies', 'Glucose', 'BloodPressure', 'SkinThickness', 'Insu
lin',
       'BMI', 'DiabetesPedigreeFunction', 'Age', 'Outcome'],
      dtype='object')
```

In [5]:
```
dataframe.dtypes
```

Out[5]:
```
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 [6]:
```
dataframe.isnull().sum()
```

Out[6]:
```
Pregnancies                 0
Glucose                     0
BloodPressure               0
SkinThickness               0
Insulin                     0
BMI                         0
DiabetesPedigreeFunction    0
Age                         0
Outcome                     0
dtype: int64
```

In [7]:
```
dataframe.head(20)
```

Out[7]:

	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	

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

In [8]:
```
dataframe.shape
```

Out[8]:
```
(768, 9)
```

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

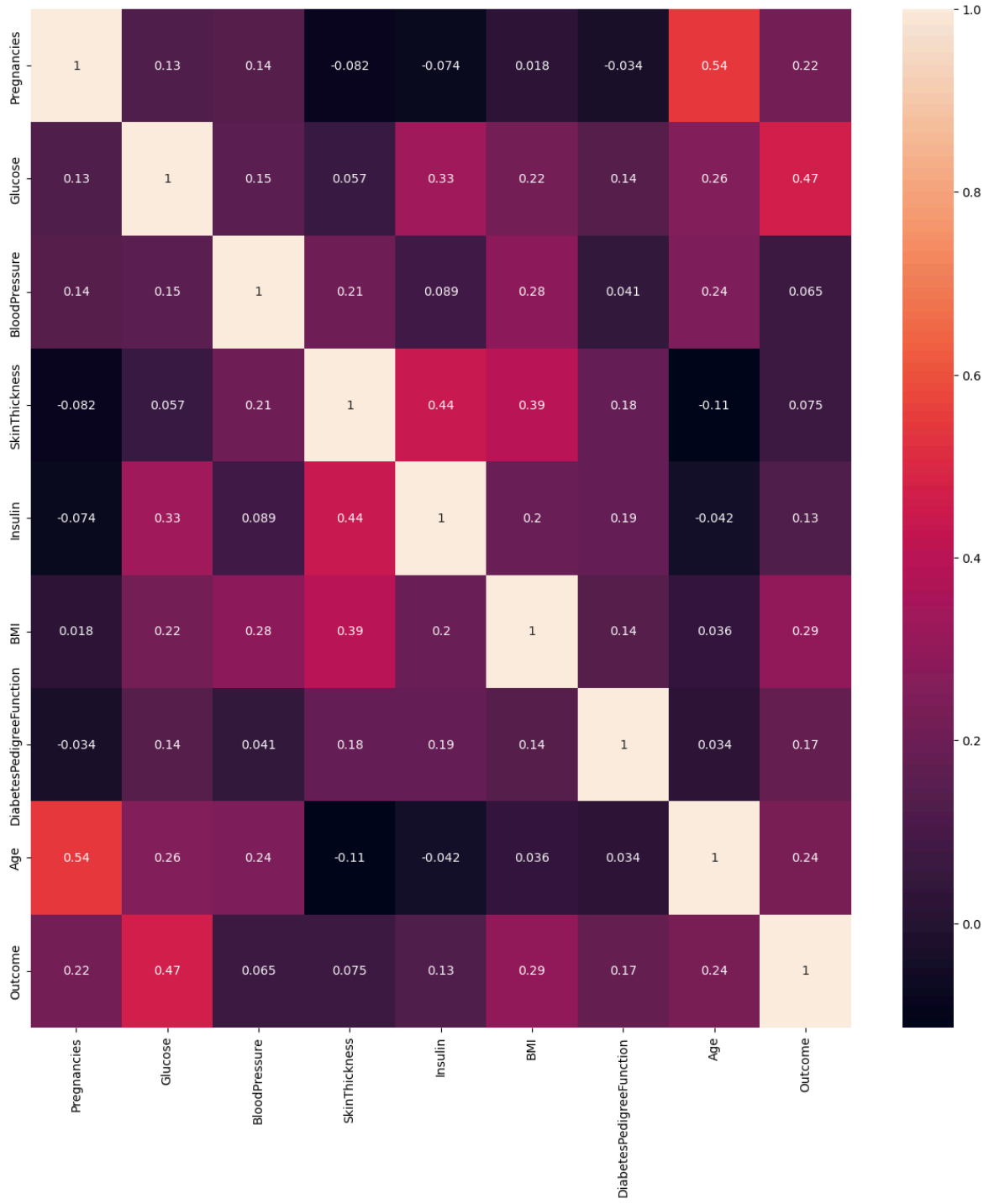
Correlation Coefficient

In [9]:
```
dataframe.corr()
```

Out[9]:

	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 [10]:
```
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 [11]:
```
dataframe.describe()
```

Out[11]:

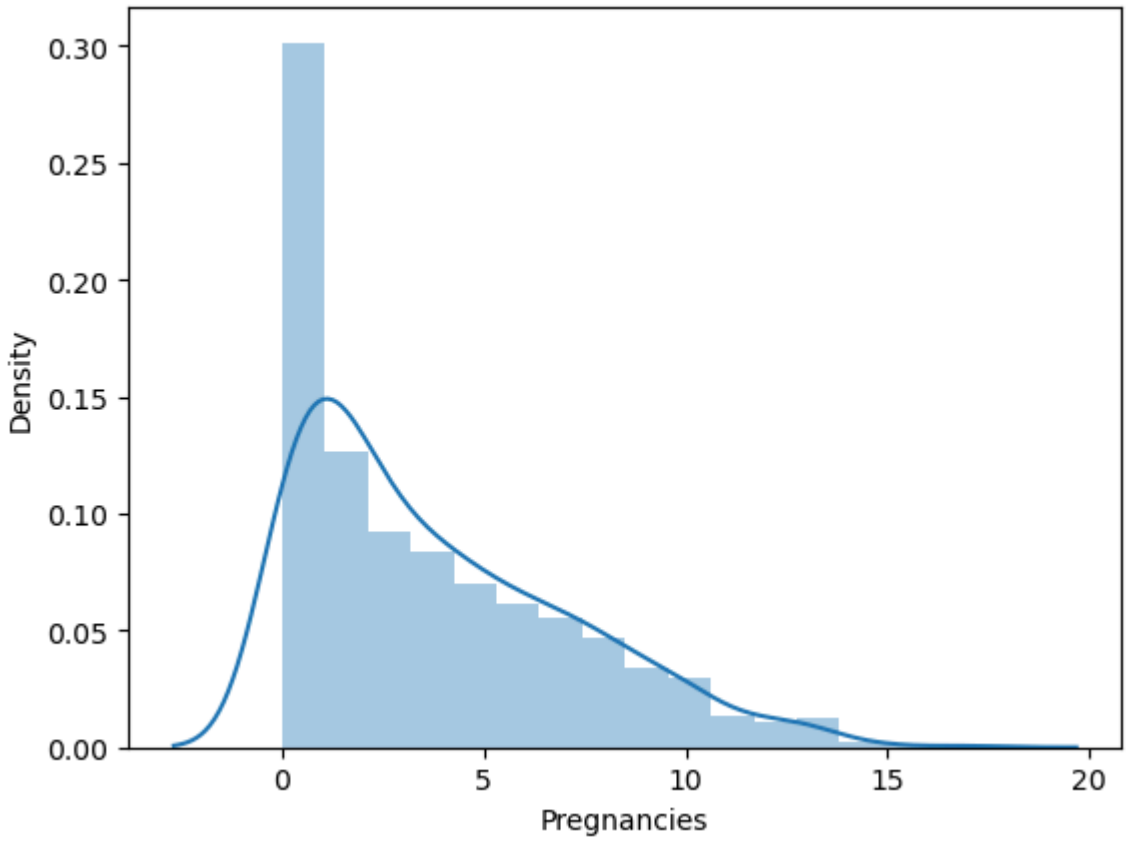
	Pregnancies	Glucose	BloodPressure	SkinThickness	Insulin	
count	768.000000	768.000000	768.000000	768.000000	768.000000	768.0
mean	3.845052	120.894531	69.105469	20.536458	79.799479	31.9
std	3.369578	31.972618	19.355807	15.952218	115.244002	7.8
min	0.000000	0.000000	0.000000	0.000000	0.000000	0.0
25%	1.000000	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

In [12]:
```
# Pregnancies -> Median
sns.distplot(dataframe.Pregnancies)
```

```
/var/folders/8h/zprf7hjs319_78816p34b90c0000gn/T/ipykernel_35649/325720374
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)
```

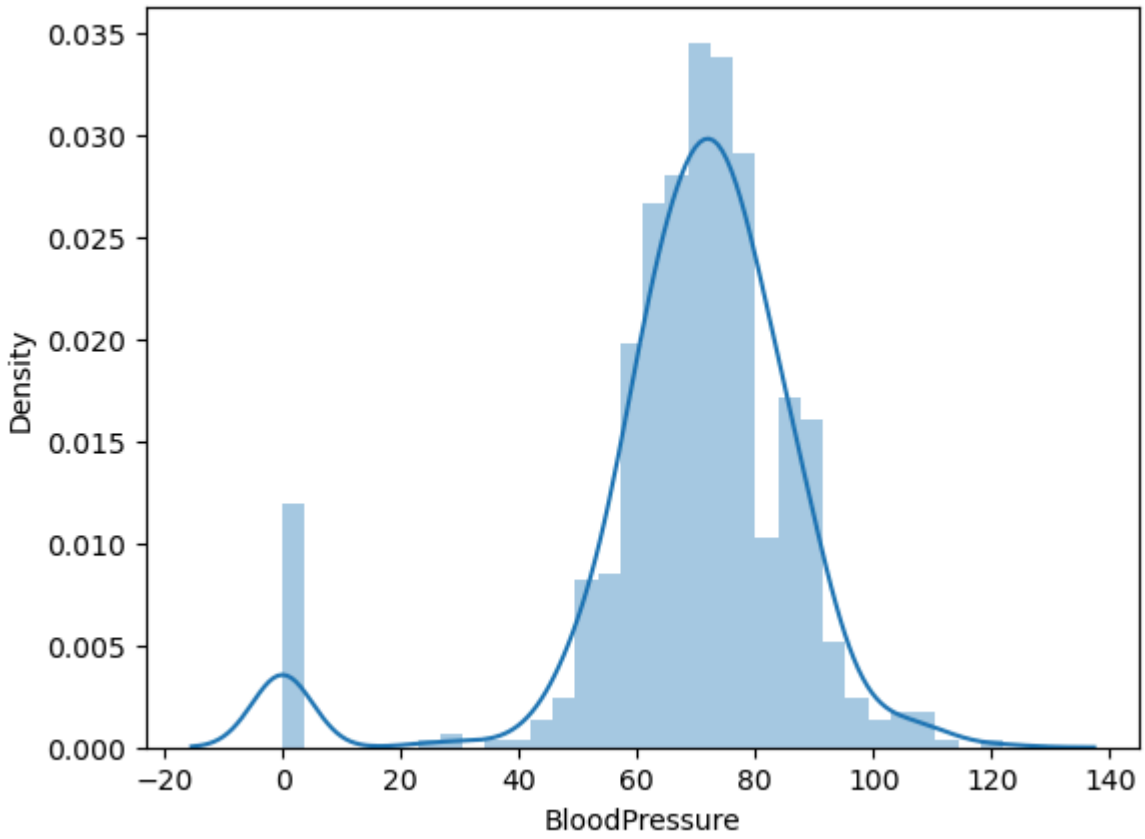
Out[12]:
```
<Axes: xlabel='Pregnancies', ylabel='Density'>
```



In [13]:
```
## BP -> Mean
sns.distplot(dataframe.BloodPressure)
```

```
/var/folders/8h/zprf7hjs319_78816p34b90c0000gn/T/ipykernel_35649/891648068.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 with
similar flexibility) or `histplot` (an axes-level function for histograms).

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

  sns.distplot(dataframe.BloodPressure)
```

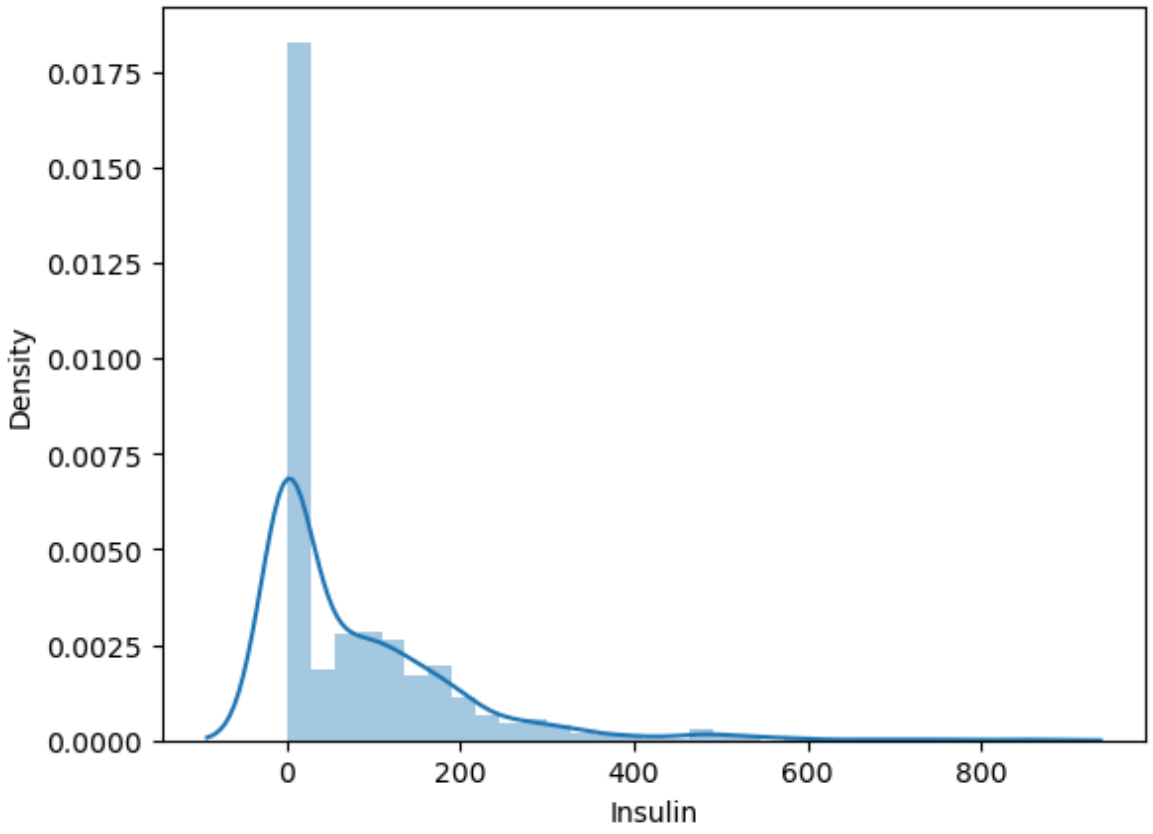
Out[13]: <Axes: xlabel='BloodPressure', ylabel='Density'>



In [14]:
```
# Insulin -> Median
sns.distplot(dataframe.Insulin)
```

```
/var/folders/8h/zprf7hjs319_78816p34b90c0000gn/T/ipykernel_35649/257615224
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[14]: <Axes: xlabel='Insulin', ylabel='Density'>



In [15]:
```
dataframe.columns
```

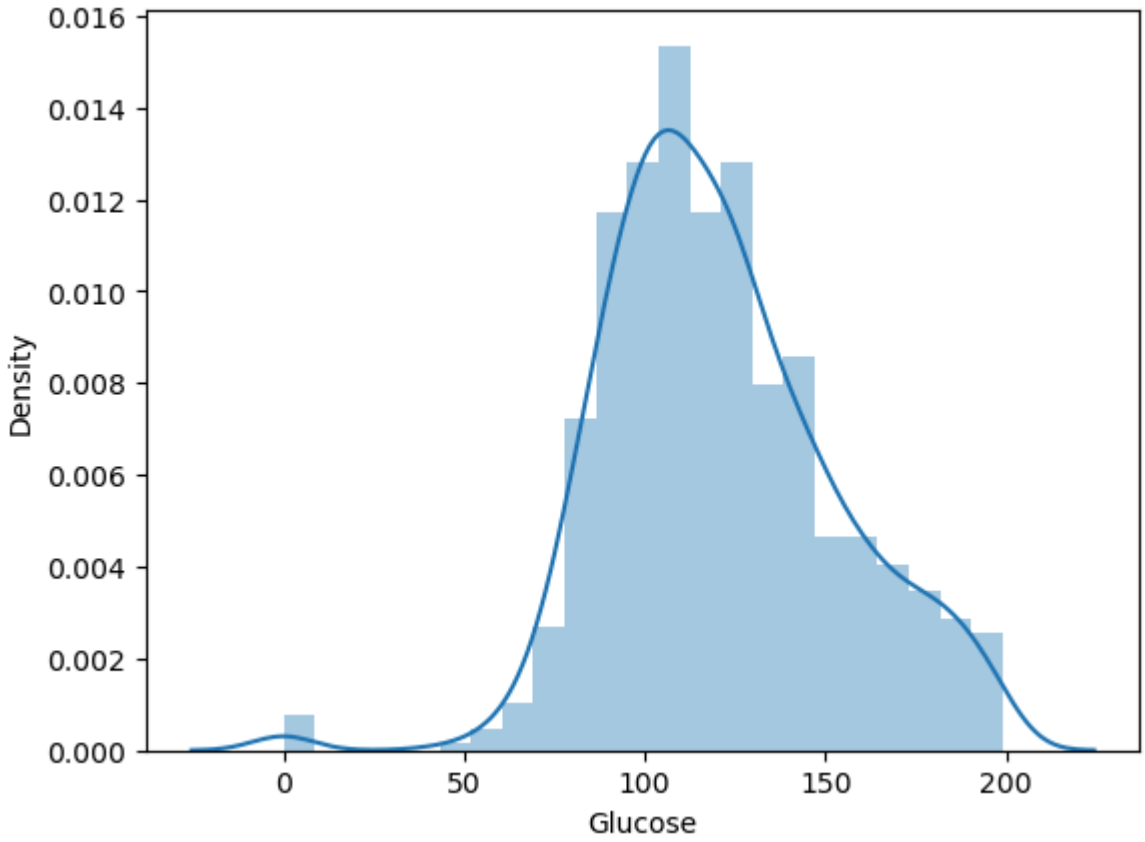
Out[15]:
```
Index(['Pregnancies', 'Glucose', 'BloodPressure', 'SkinThickness', 'Insu
lin',
       'BMI', 'DiabetesPedigreeFunction', 'Age', 'Outcome'],
      dtype='object')
```

In [16]:
```
## Insuline -> Right skewed distribution
dataframe['Insulin'] = dataframe['Insulin'].replace(0, dataframe['Insulin
```

In [17]:
```
sns.distplot(dataframe.Glucose)
```

```
/var/folders/8h/zprf7hjs319_78816p34b90c0000gn/T/ipykernel_35649/223043267
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)
```

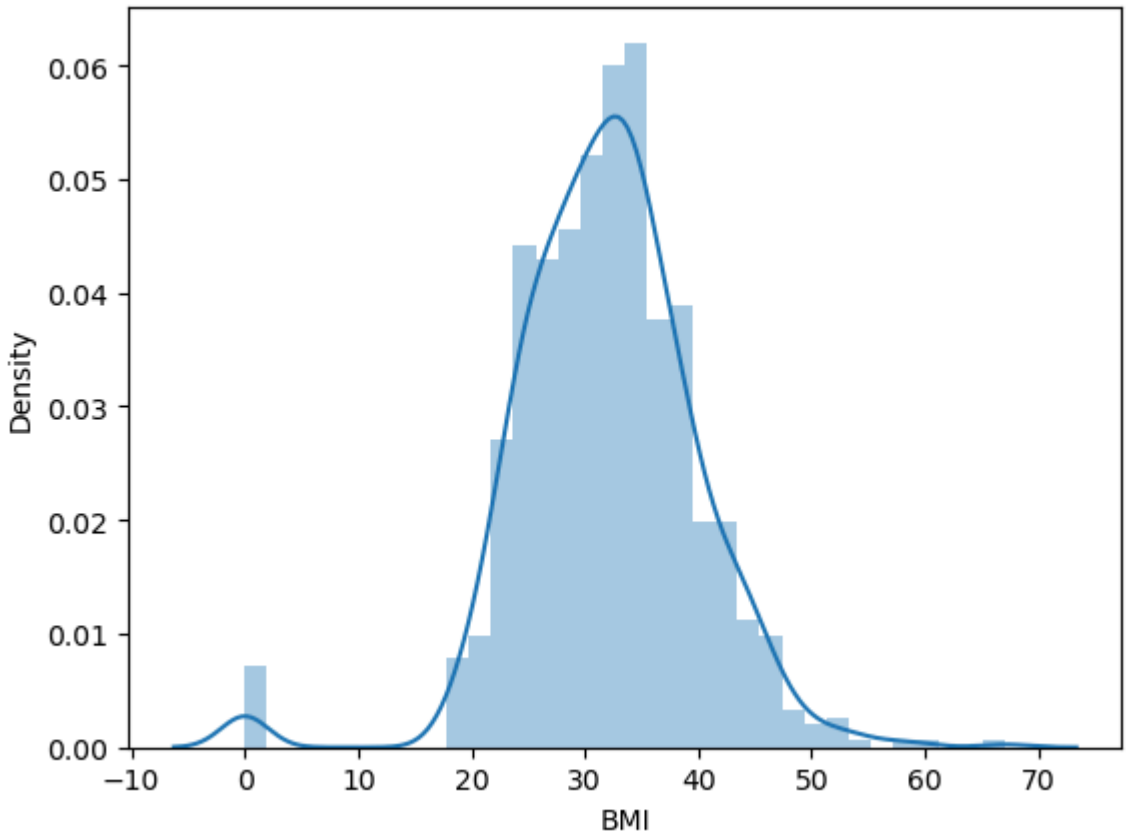
Out[17]:
```
<Axes: xlabel='Glucose', ylabel='Density'>
```



In [18]:
```
sns.distplot(dataframe.BMI)
```

```
/var/folders/8h/zprf7hjs319_78816p34b90c0000gn/T/ipykernel_35649/252098079
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)
```

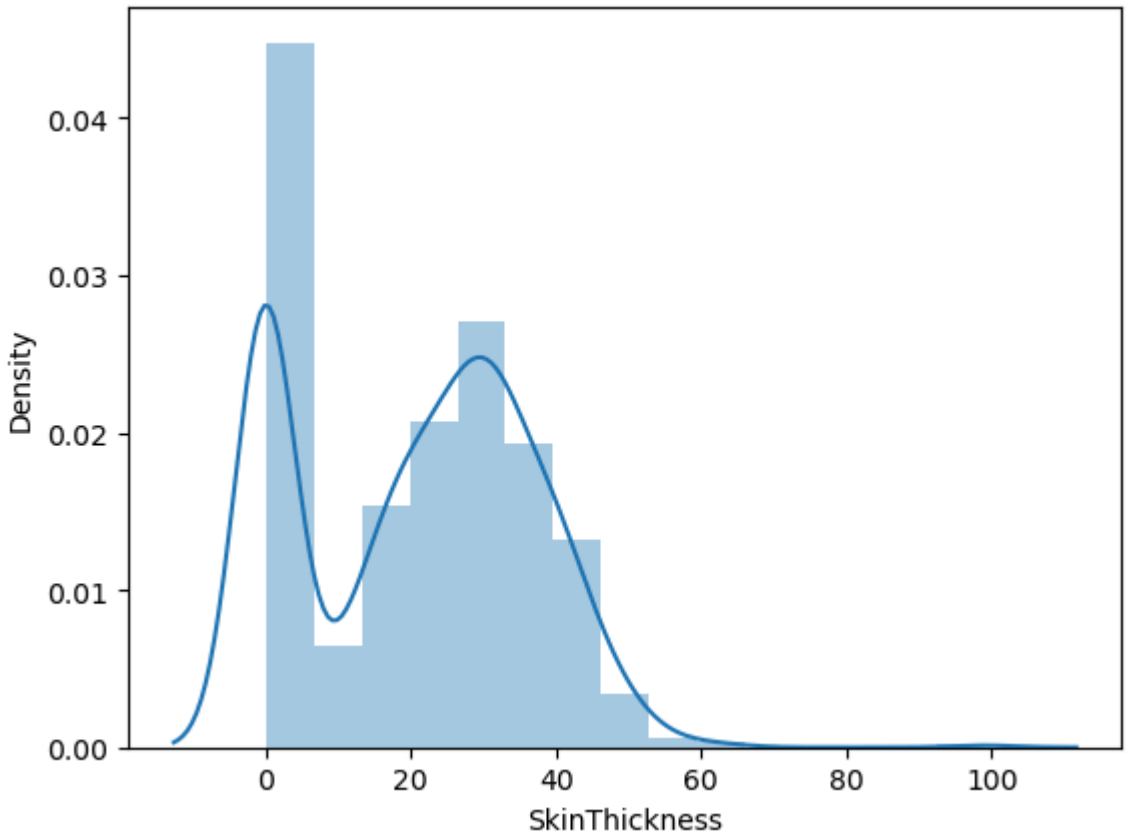
Out[18]: <Axes: xlabel='BMI', ylabel='Density'>



In [19]:
```
sns.distplot(dataframe.SkinThickness)
```

```
/var/folders/8h/zprf7hjs319_78816p34b90c0000gn/T/ipykernel_35649/3861253045.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 with
similar flexibility) or `histplot` (an axes-level function for histograms).

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

  sns.distplot(dataframe.SkinThickness)
```

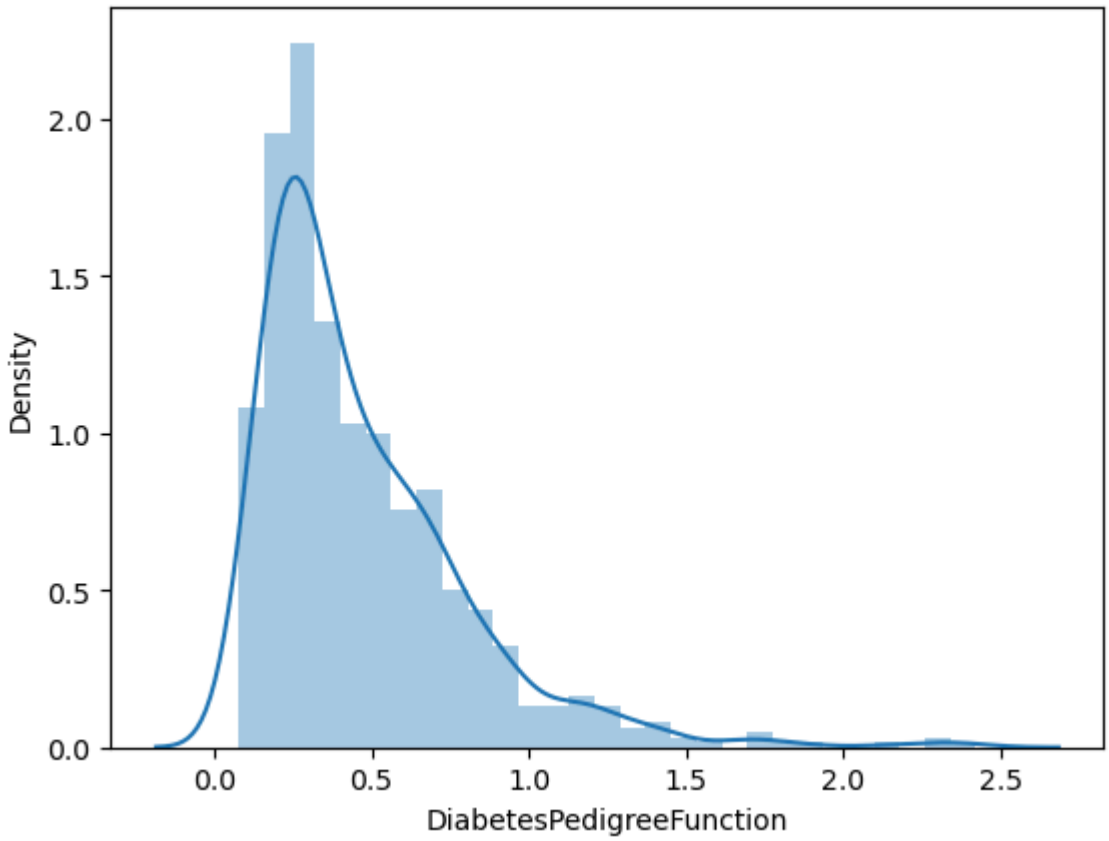
Out[19]: <Axes: xlabel='SkinThickness', ylabel='Density'>



In [20]:
```
sns.distplot(dataframe.DiabetesPedigreeFunction)
```

```
/var/folders/8h/zprf7hjs319_78816p34b90c0000gn/T/ipykernel_35649/264275873
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)
```

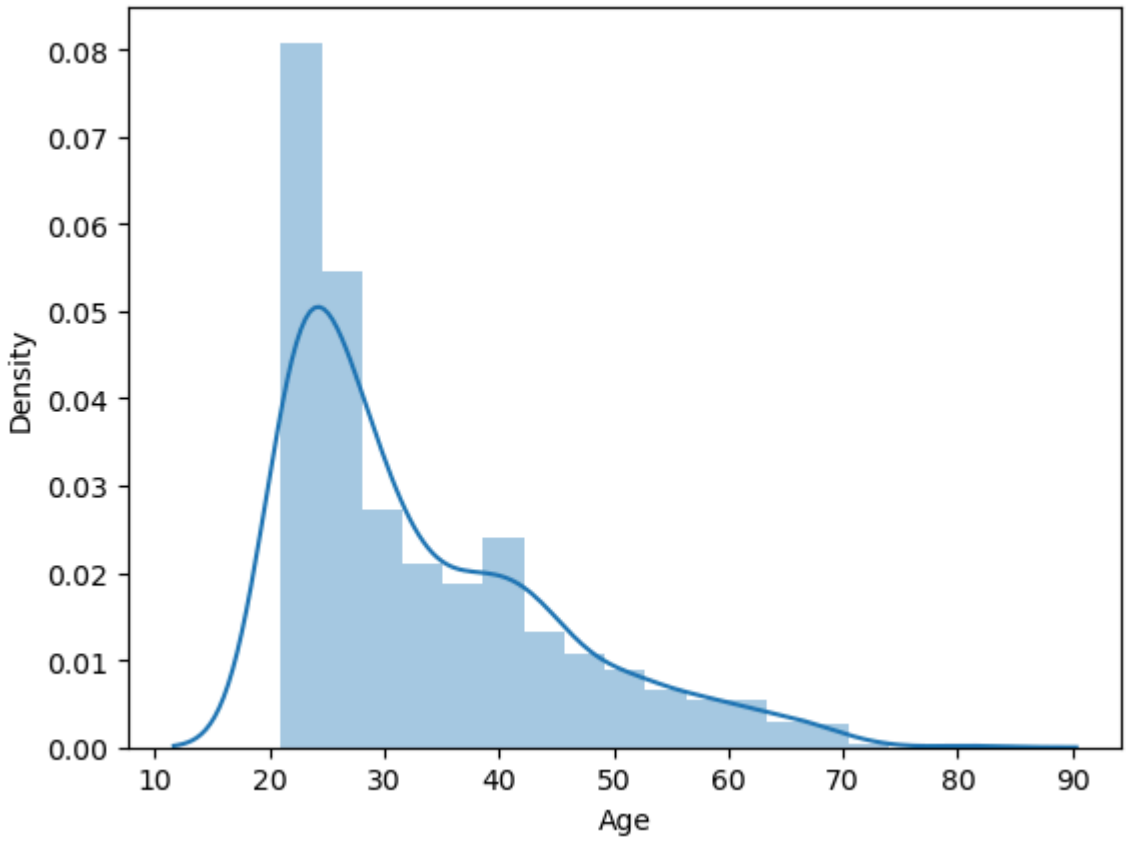
Out[20]: <Axes: xlabel='DiabetesPedigreeFunction', ylabel='Density'>



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

```
/var/folders/8h/zprf7hjs319_78816p34b90c0000gn/T/ipykernel_35649/2691430987.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 with
similar flexibility) or `histplot` (an axes-level function for histograms).

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[21]: <Axes: xlabel='Age', ylabel='Density'>



In [22]:
```
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 [23]:
```
dataframe.head(20)
```

Out[23]:

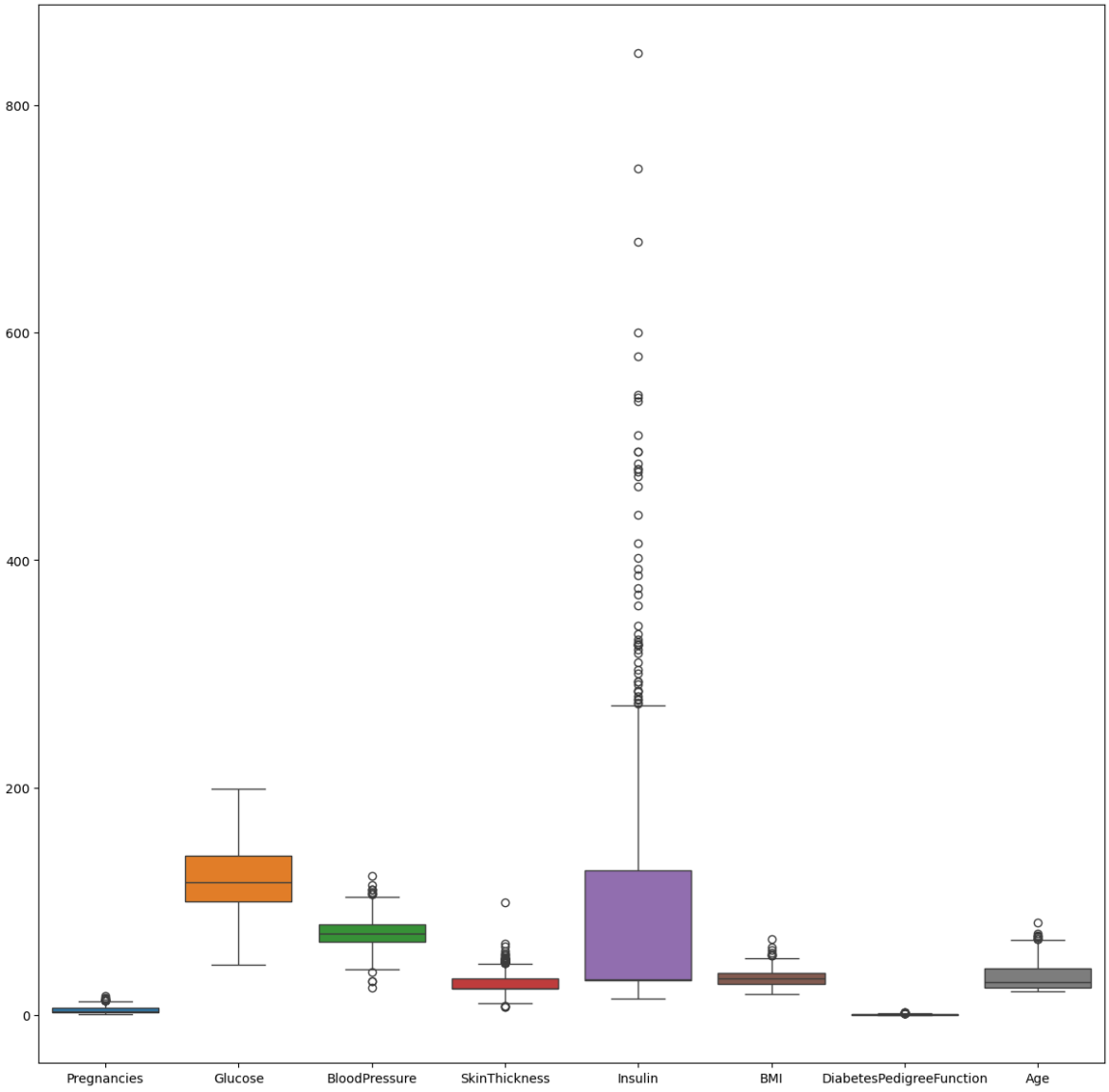
	Pregnancies	Glucose	BloodPressure	SkinThickness	Insulin	BMI	Diabe
0	6	148.0	72.000000	35	30.5	33.600000	
1	1	85.0	66.000000	29	30.5	26.600000	
2	8	183.0	64.000000	23	30.5	23.300000	
3	1	89.0	66.000000	23	94.0	28.100000	
4	3	137.0	40.000000	35	168.0	43.100000	
5	5	116.0	74.000000	23	30.5	25.600000	
6	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 robuts to outliers and why
- Data Imputation via Mean and Median(Numeric Data) => Symmetric -> Mean and Skewed -> Median, Categorical Data => Mode

In [24]:
```
## X -> input features y -> target value
X = dataframe.drop(columns='Outcome', axis=1)
y = dataframe['Outcome']
```

Outlier Detection -> Box Plot

In [25]:
```
fig, ax = plt.subplots(figsize = (15, 15))
sns.boxplot(data = X, ax=ax)
plt.savefig('boxPlot.jpg')
```



In [26]:
```
X.shape
```

Out[26]: (768, 8)

In [27]:
```
y.shape
```

Out[27]: (768,)

In [28]:
```
cols = ['Pregnancies', 'Glucose', 'BloodPressure', 'SkinThickness', 'Insu
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)
```

In [29]:
```
X_outlier_detection = X[mask]
y_outlier_detection = y[mask]
```

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

Out[30]: (759, 8)

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

Out[31]:
```
(759,)
```

Standardization

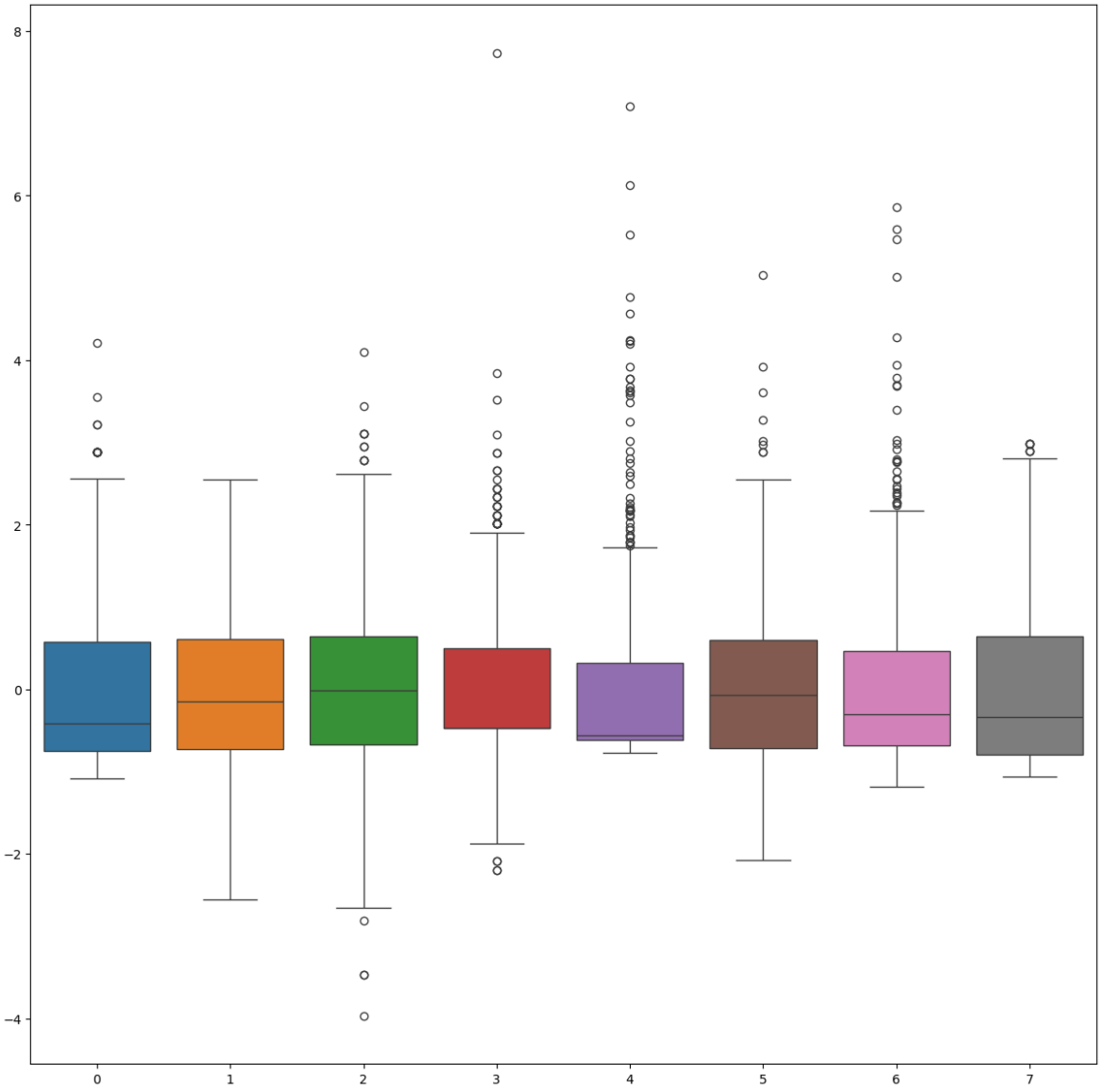
Standard Normal Form -> Mean = 0 and standard deviation = 1

In [32]:
```
from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X_outlier_detection)
```

In [33]:
```
X_scaled
```

Out[33]:
```
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 [34]:
```
fig, ax = plt.subplots(figsize = (15, 15))
sns.boxplot(data = X_scaled, ax=ax)
plt.savefig('boxPlot.jpg')
```



In [35]:
```
dataframe.columns
```

Out[35]:
```
Index(['Pregnancies', 'Glucose', 'BloodPressure', 'SkinThickness', 'Insu
lin',
       'BMI', 'DiabetesPedigreeFunction', 'Age', 'Outcome'],
      dtype='object')
```

In [36]:
```
cols = ['Pregnancies', 'Glucose', 'BloodPressure', 'SkinThickness', 'Insu
```

In [37]:
```
type(X_scaled)
```

Out[37]:
```
numpy.ndarray
```

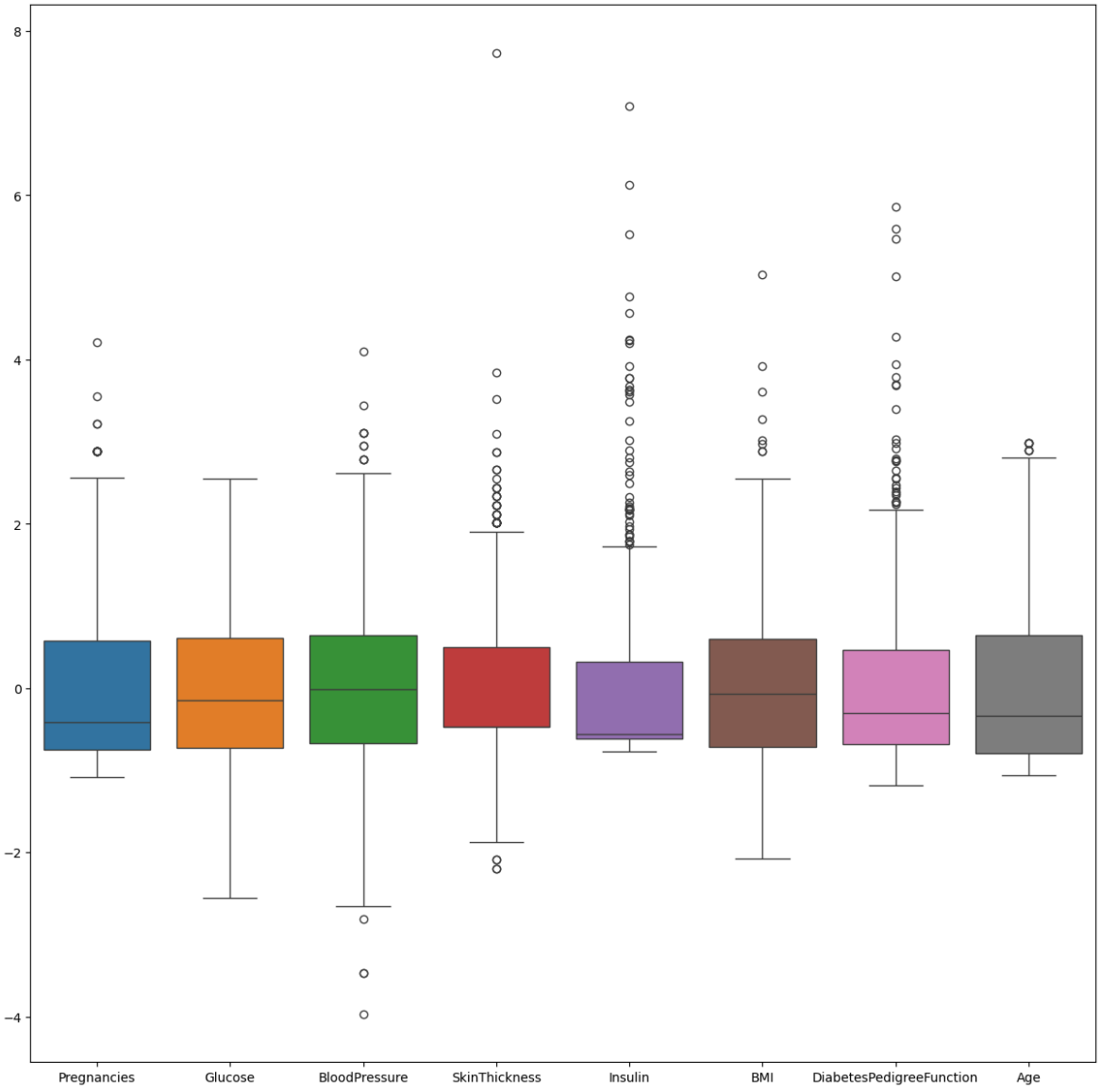
In [38]:
```
X_scaled = pd.DataFrame(X_scaled, columns=cols)
X_scaled.describe()
```

Out[38]:

	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

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

In [39]:
```
fig, ax = plt.subplots(figsize = (15, 15))
sns.boxplot(data = X_scaled, ax=ax)
plt.savefig('boxPlot.jpg')
```



In [40]:
```
y_outlier_detection.shape
```

Out[40]:
```
(759,)
```

In [41]:
```
y_outlier_detection.value_counts()
```

Out[41]:
```
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 [42]:
```
X_scaled.reset_index(drop=True, inplace=True)
y_outlier_detection.reset_index(drop=True, inplace=True)
```

In [43]:
```
q = X_scaled['Insulin'].quantile(.95)
mask = X_scaled['Insulin'] < q
dataNew = X_scaled[mask]
y_outlier_detection = y_outlier_detection[mask]
```

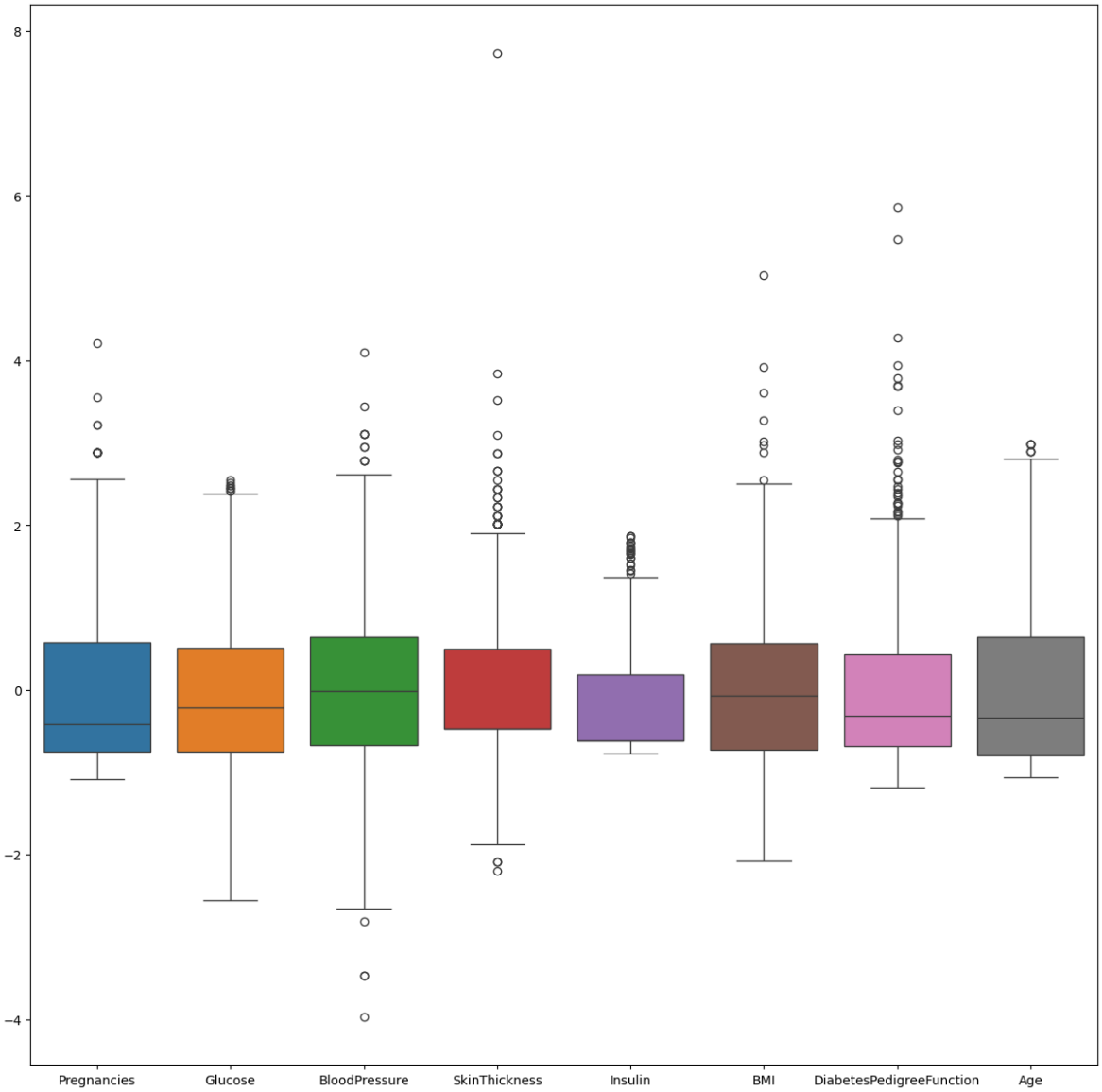
In [44]:
```
dataNew.shape
```

Out[44]: (721, 8)

In [45]:
```
y_outlier_detection.shape
```

Out[45]: (721,)

In [46]:
```
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 [47]:
```
from sklearn.model_selection import train_test_split
```

```
X_train, X_test, y_train, y_test = train_test_split(dataNew, y_outlier_de
```

In [48]:
```
X_train.shape
```

Out[48]:
```
(483, 8)
```

In [49]:
```
X_test.shape
```

Out[49]:
```
(238, 8)
```

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

In [50]:
```
y_train.value_counts()
```

Out[50]:
```
Outcome
0    318
1    165
Name: count, dtype: int64
```

SMOTE Technique

In [51]:
```
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
```

In [52]:
```
from sklearn.linear_model import LogisticRegression
classification = LogisticRegression()
classification.fit(X_train_resampled, y_train_resampled)
```

Out[52]:
```
LogisticRegression()
```

Model Predictions

In [53]:
```
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 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 [54]:
```
from sklearn.metrics import accuracy_score
accuracy_score(y_test, y_predictions)
```

Out[54]:
```
0.7478991596638656
```

Healthcare: Recall is very important metric

In [55]:
```
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
```

In [56]:
```
import pickle
pickle.dump(classification, open("classification_model.pkl", "wb"))
```

In [57]:
```
classification_model = pickle.load(open("classification_model.pkl", "rb")
classification_model.predict(X_test)
```

Out[57]:
```
array([0, 1, 0, 0, 1, 1, 0, 1, 1, 0, 0, 1, 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 [58]:
```
from sklearn.neighbors import KNeighborsClassifier, KNeighborsRegressor
from sklearn.metrics import classification_report, confusion_matrix
knn = KNeighborsClassifier()
```

In [59]:
```
knn.fit(X_train_resampled, y_train_resampled)
```

Out[59]:
```
KNeighborsClassifier()
```

Model Prediction

In [60]:
```
y_prediction_knn = knn.predict(X_test)
y_prediction_knn
```

Out[60]:
```
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 [61]:
```
print("Confusion Matrix")
print(confusion_matrix(y_test, y_prediction_knn))
```

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

In [62]:
```
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
```