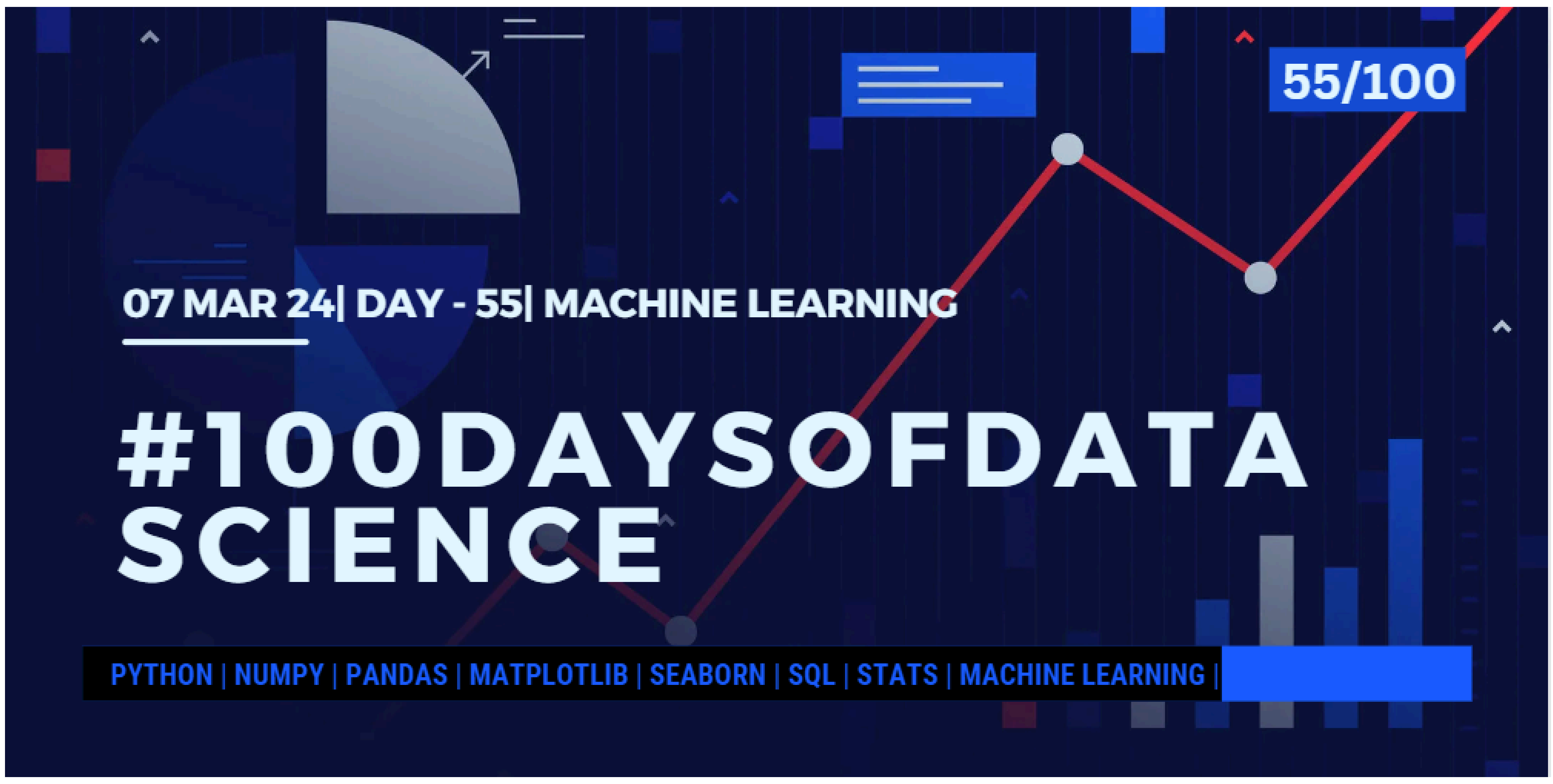
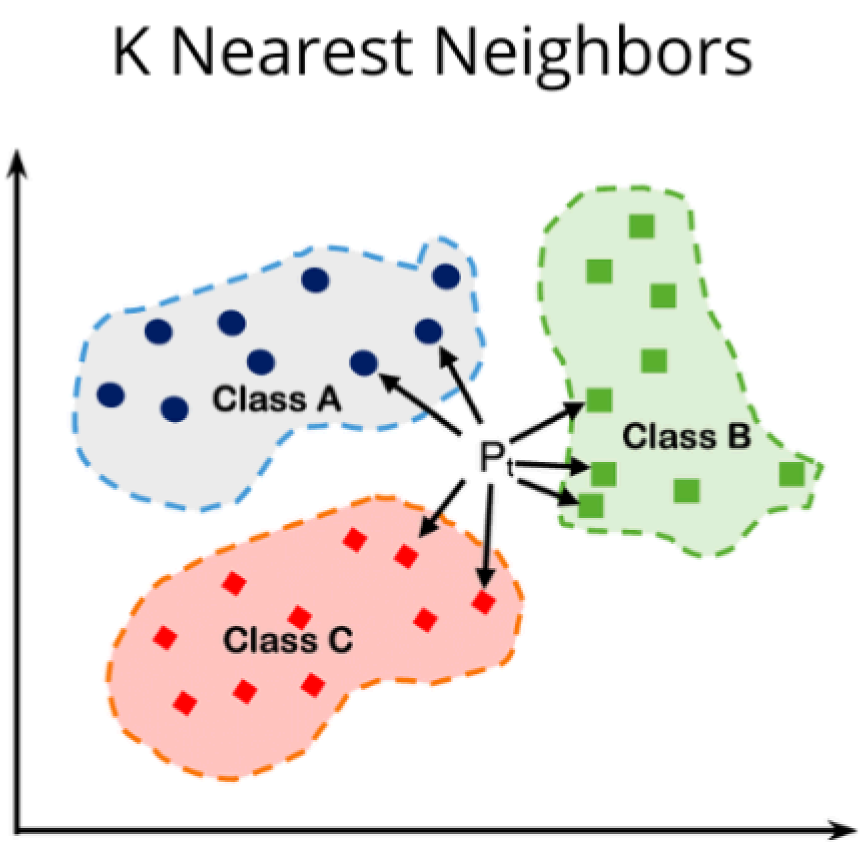


K-nearest neighbor (KNN)



- KNN might not be the most complex algorithm out there, but its simplicity and adaptability make it a workhorse in the machine learning world.
- KNN works based on the idea that similar data points tend to have similar characteristics. This makes it easy to understand and a great starting point for beginners in machine learning.
- Unlike some algorithms that struggle with categorical data, KNN can handle both numerical and categorical features without extensive pre-processing. This makes it applicable to a wide range of datasets.
- KNN uses the concept of distance in the feature space to make predictions. It analyzes the nearest neighbors (closest data points based on features) of a new data point for classification tasks (predicting a category) or regression tasks (predicting a continuous value).
- While not the most powerful for all situations, KNN can be very effective in classification tasks, particularly for datasets with well-defined clusters.
- KNN is a relatively interpretable algorithm. By analyzing the nearest neighbors, you can gain some insights into the factors influencing the predictions.
- Finding nearest neighbors for large datasets can be computationally demanding, which can be a drawback.
- **KNN: Strengths and Applications:**
  - **Simple and Versatile:** KNN is easy to understand and implement, making it a great starting point for beginners.
  - **Effective for Certain Tasks:** KNN excels in classification tasks for datasets with well-defined clusters.
  - **No Feature Scaling Required:** Unlike some algorithms, KNN doesn't require feature scaling.
- **Considerations for KNN:**
  - **Curse of Dimensionality:** KNN's performance can suffer in high-dimensional datasets.
  - **Interpretability:** While the concept is simple, understanding the impact of individual features can be challenging.
  - **Computationally Expensive:** For large datasets, finding the nearest neighbors can be computationally demanding.
- **Distance Matters:** KNN's core principle relies on the concept of distance in the feature space. By analyzing the nearest neighbors (data points closest in terms of features) of a new data point, KNN can make predictions for classification or regression tasks. This approach allows KNN to adapt to the underlying structure of the data without requiring complex feature engineering.

In [1]:
```
### Importing Libraries
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
import plotly.express as px
import plotly.figure_factory as ff
from sklearn.metrics import confusion_matrix, accuracy_score, classification_report
from sklearn.metrics import mean_squared_error, r2_score
import warnings
warnings.filterwarnings('ignore')
```

In [2]:
```
### Import the Dataset
df = pd.read_csv(r'C:\Users\hp\Desktop\100DaysOfDataScience\Day 55\diabetes.csv',header=0)
df.head()
```

Out[2]:

	Pregnancies	Glucose	BloodPressure	SkinThickness	Insulin	BMI	DiabetesPedigreeFunction	Age	Outcome
0	6	148	72	35	0	33.6	0.627	50	1
1	1	85	66	29	0	26.6	0.351	31	0
2	8	183	64	0	0	23.3	0.672	32	1
3	1	89	66	23	94	28.1	0.167	21	0
4	0	137	40	35	168	43.1	2.288	33	1

In [3]:
```
df.shape ### Checking Shape
```

Out[3]:
```
(768, 9)
```

In [4]:
```
df.describe() ### Get information of the Dataset
```

Out[4]:

	Pregnancies	Glucose	BloodPressure	SkinThickness	Insulin	BMI	DiabetesPedigreeFunction	Age	Outcome
count	768.000000	768.000000	768.000000	768.000000	768.000000	768.000000	768.000000	768.000000	768.000000
mean	3.845052	120.894531	69.105469	20.536458	79.799479	31.992578	0.471876	33.240885	0.348958
std	3.369578	31.972618	19.355807	15.952218	115.244002	7.884160	0.331329	11.760232	0.476951
min	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.078000	21.000000	0.000000
25%	1.000000	99.000000	62.000000	0.000000	0.000000	27.300000	0.243750	24.000000	0.000000
50%	3.000000	117.000000	72.000000	23.000000	30.500000	32.000000	0.372500	29.000000	0.000000
75%	6.000000	140.250000	80.000000	32.000000	127.250000	36.600000	0.626250	41.000000	1.000000
max	17.000000	199.000000	122.000000	99.000000	846.000000	67.100000	2.420000	81.000000	1.000000

In [5]:
```
df.columns ### Checking Columns
```

Out[5]:
```
Index(['Pregnancies', 'Glucose', 'BloodPressure', 'SkinThickness', 'Insulin',
       'BMI', 'DiabetesPedigreeFunction', 'Age', 'Outcome'],
      dtype='object')
```

In [6]:
```
df.info() ### Checking Information About a DataFrame
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 768 entries, 0 to 767
Data columns (total 9 columns):
 #   Column                    Non-Null Count  Dtype  
---  ------                    --------------  -----  
 0   Pregnancies               768 non-null    int64  
 1   Glucose                   768 non-null    int64  
 2   BloodPressure             768 non-null    int64  
 3   SkinThickness             768 non-null    int64  
 4   Insulin                   768 non-null    int64  
 5   BMI                       768 non-null    float64
 6   DiabetesPedigreeFunction  768 non-null    float64
 7   Age                       768 non-null    int64  
 8   Outcome                   768 non-null    int64  
dtypes: float64(2), int64(7)
memory usage: 54.1 KB
```

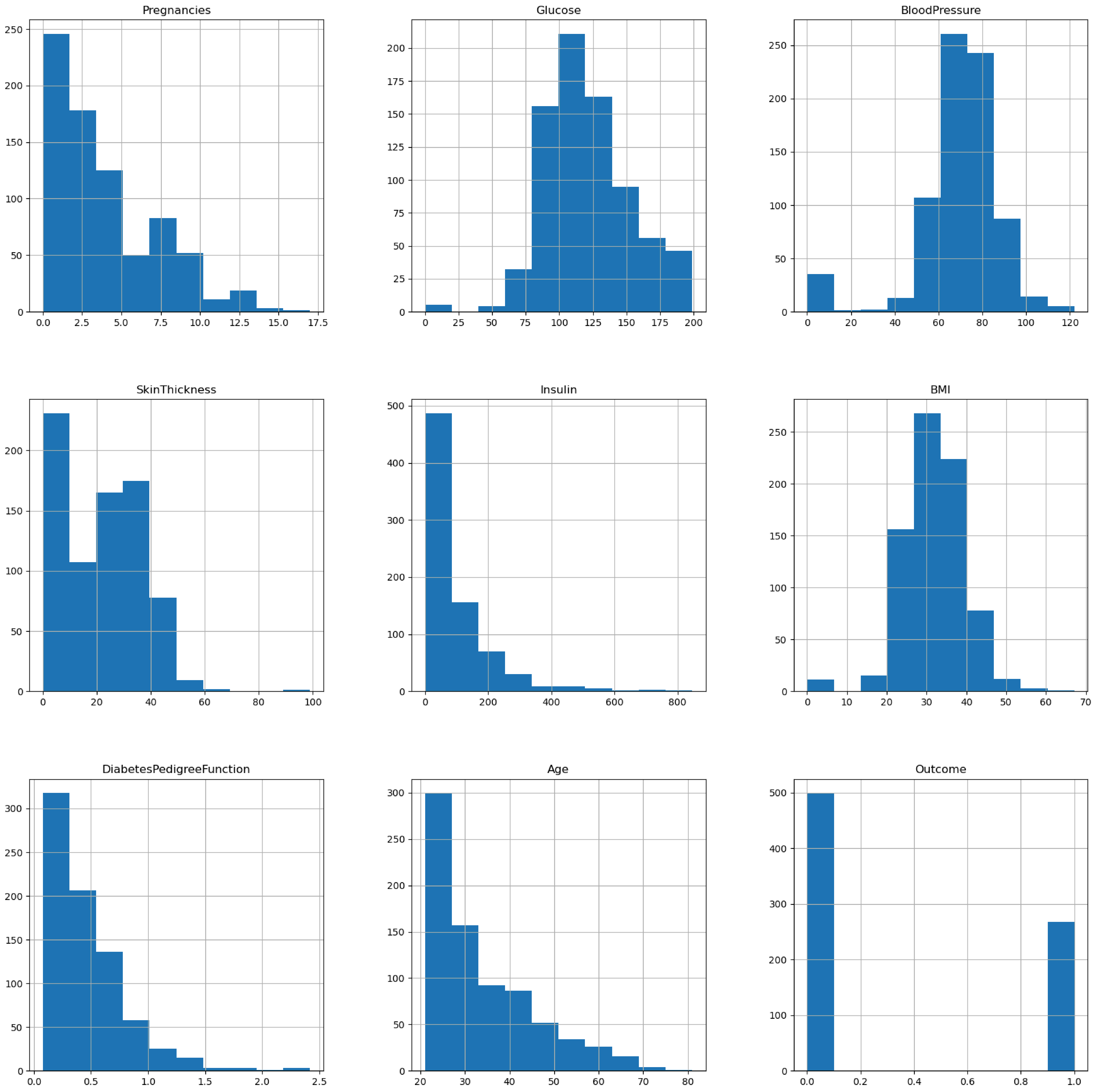
In [7]:
```
df.isnull().sum() ### Checking Null Values in the Data
```

Out[7]:
```
Pregnancies                 0
Glucose                     0
BloodPressure               0
SkinThickness               0
Insulin                     0
BMI                         0
DiabetesPedigreeFunction    0
Age                         0
Outcome                     0
dtype: int64
```

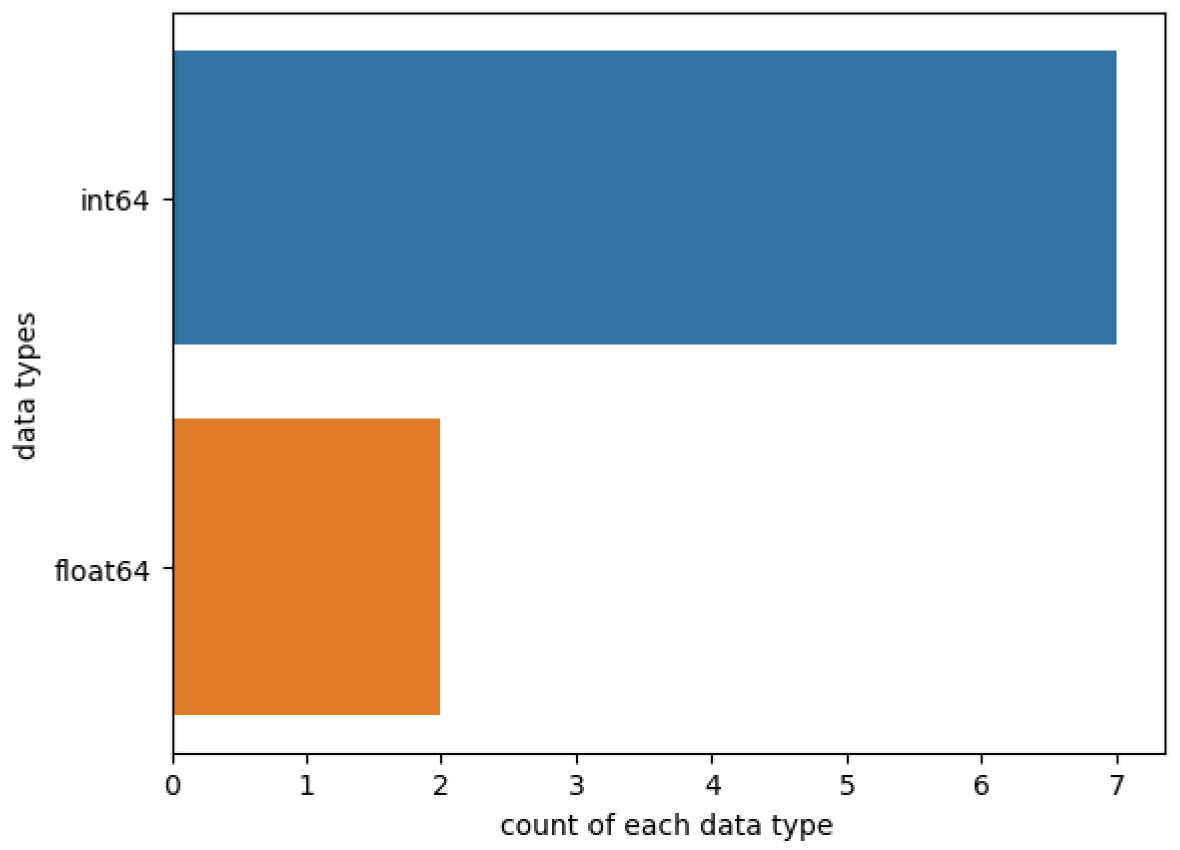
In [8]:
```
df1 = pd.DataFrame.copy(df)
df1.shape
```

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

In [9]:
```
### Visualizing columns
p = df1.hist(figsize = (20,20))
```



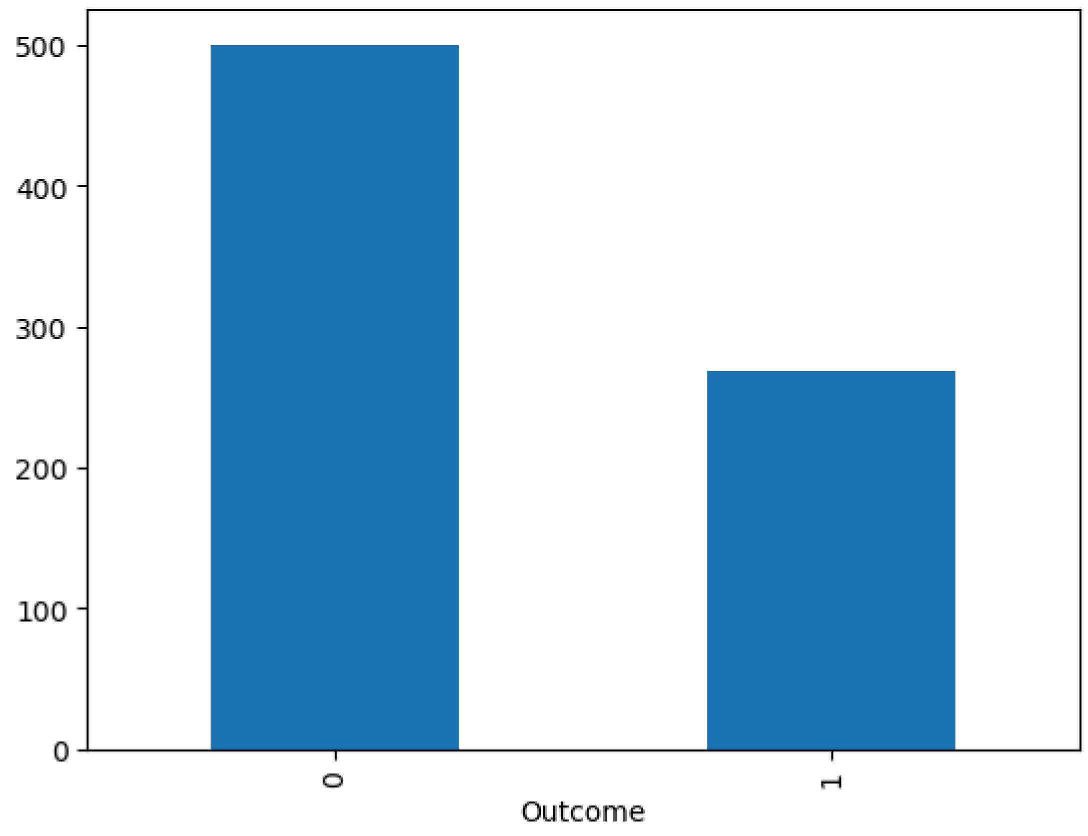
In [10]:
```
### Checking dtypes
sns.countplot(y=df1.dtypes ,data=df1)
plt.xlabel("count of each data type")
plt.ylabel("data types")
plt.show()
```



In [11]:

```
### Checking the balance of the data by plotting the count of outcomes by their value
color_wheel = {1: "#0392cf",
               2: "#7bc043",
               3: "#f44336"}
colors = df1["Outcome"].map(lambda x: color_wheel.get(x + 1))
print(df1.Outcome.value_counts())
p=df1.Outcome.value_counts().plot(kind="bar")
```

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

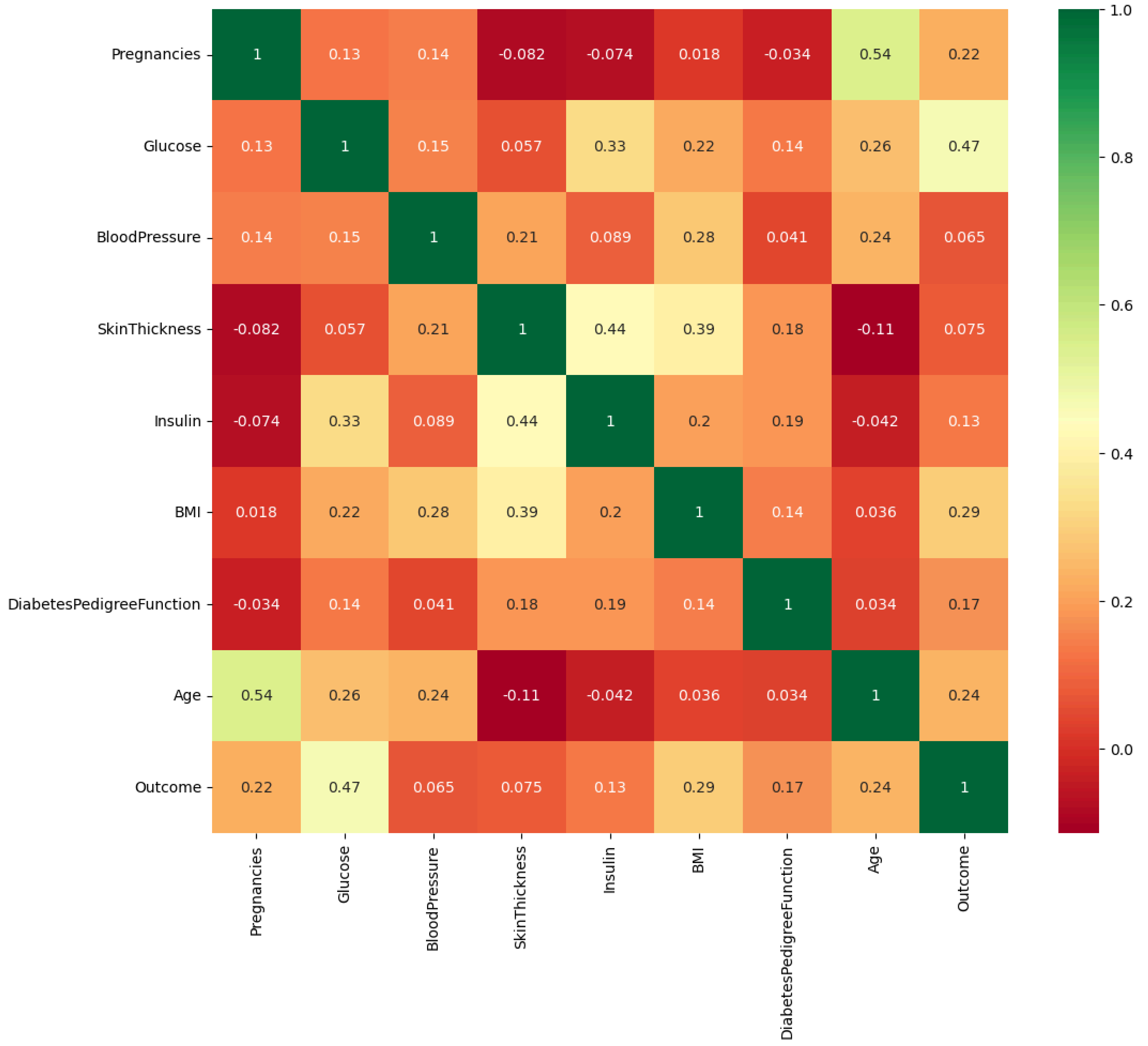


In [12]:
```
for i in df1.columns:
    print({i:df1[i].unique()}) ### Checking Unique values in each columns
```

```
{'Pregnancies': array([ 6,  1,  8,  0,  5,  3, 10,  2,  4,  7,  9, 11, 13, 15, 17, 12, 14],
      dtype=int64)}
{'Glucose': array([148,  85, 183,  89, 137, 116,  78, 115, 197, 125, 110, 168, 139,
       189, 166, 100, 118, 107, 103, 126,  99, 196, 119, 143, 147,  97,
       145, 117, 109, 158,  88,  92, 122, 138, 102,  90, 111, 180, 133,
       106, 171, 159, 146,  71, 105, 101, 176, 150,  73, 187,  84,  44,
       141, 114,  95, 129,  79,   0,  62, 131, 112, 113,  74,  83, 136,
        80, 123,  81, 134, 142, 144,  93, 163, 151,  96, 155,  76, 160,
       124, 162, 132, 120, 173, 170, 128, 108, 154,  57, 156, 153, 188,
       152, 104,  87,  75, 179, 130, 194, 181, 135, 184, 140, 177, 164,
        91, 165,  86, 193, 191, 161, 167,  77, 182, 157, 178,  61,  98,
       127,  82,  72, 172,  94, 175, 195,  68, 186, 198, 121,  67, 174,
       199,  56, 169, 149,  65, 190], dtype=int64)}
{'BloodPressure': array([ 72,  66,  64,  40,  74,  50,   0,  70,  96,  92,  80,  60,  84,
        30,  88,  90,  94,  76,  82,  75,  58,  78,  68, 110,  56,  62,
        85,  86,  48,  44,  65, 108,  55, 122,  54,  52,  98, 104,  95,
        46, 102, 100,  61,  24,  38, 106, 114], dtype=int64)}
{'SkinThickness': array([35, 29,  0, 23, 32, 45, 19, 47, 38, 30, 41, 33, 26, 15, 36, 11, 31,
       37, 42, 25, 18, 24, 39, 27, 21, 34, 10, 60, 13, 20, 22, 28, 54, 40,
       51, 56, 14, 17, 50, 44, 12, 46, 16,  7, 52, 43, 48,  8, 49, 63, 99],
      dtype=int64)}
{'Insulin': array([  0,  94, 168,  88, 543, 846, 175, 230,  83,  96, 235, 146, 115,
       140, 110, 245,  54, 192, 207,  70, 240,  82,  36,  23, 300, 342,
       304, 142, 128,  38, 100,  90, 270,  71, 125, 176,  48,  64, 228,
        76, 220,  40, 152,  18, 135, 495,  37,  51,  99, 145, 225,  49,
        50,  92, 325,  63, 284, 119, 204, 155, 485,  53, 114, 105, 285,
       156,  78, 130,  55,  58, 160, 210, 318,  44, 190, 280,  87, 271,
       129, 120, 478,  56,  32, 744, 370,  45, 194, 680, 402, 258, 375,
       150,  67,  57, 116, 278, 122, 545,  75,  74, 182, 360, 215, 184,
        42, 132, 148, 180, 205,  85, 231,  29,  68,  52, 255, 171,  73,
       108,  43, 167, 249, 293,  66, 465,  89, 158,  84,  72,  59,  81,
       196, 415, 275, 165, 579, 310,  61, 474, 170, 277,  60,  14,  95,
       237, 191, 328, 250, 480, 265, 193,  79,  86, 326, 188, 106,  65,
       166, 274,  77, 126, 330, 600, 185,  25,  41, 272, 321, 144,  15,
       183,  91,  46, 440, 159, 540, 200, 335, 387,  22, 291, 392, 178,
       127, 510,  16, 112], dtype=int64)}
{'BMI': array([33.6, 26.6, 23.3, 28.1, 43.1, 25.6, 31. , 35.3, 30.5,  0. , 37.6,
       38. , 27.1, 30.1, 25.8, 30. , 45.8, 29.6, 43.3, 34.6, 39.3, 35.4,
       39.8, 29. , 36.6, 31.1, 39.4, 23.2, 22.2, 34.1, 36. , 31.6, 24.8,
       19.9, 27.6, 24. , 33.2, 32.9, 38.2, 37.1, 34. , 40.2, 22.7, 45.4,
       27.4, 42. , 29.7, 28. , 39.1, 19.4, 24.2, 24.4, 33.7, 34.7, 23. ,
       37.7, 46.8, 40.5, 41.5, 25. , 25.4, 32.8, 32.5, 42.7, 19.6, 28.9,
       28.6, 43.4, 35.1, 32. , 24.7, 32.6, 43.2, 22.4, 29.3, 24.6, 48.8,
       32.4, 38.5, 26.5, 19.1, 46.7, 23.8, 33.9, 20.4, 28.7, 49.7, 39. ,
       26.1, 22.5, 39.6, 29.5, 34.3, 37.4, 33.3, 31.2, 28.2, 53.2, 34.2,
       26.8, 55. , 42.9, 34.5, 27.9, 38.3, 21.1, 33.8, 30.8, 36.9, 39.5,
       27.3, 21.9, 40.6, 47.9, 50. , 25.2, 40.9, 37.2, 44.2, 29.9, 31.9,
       28.4, 43.5, 32.7, 67.1, 45. , 34.9, 27.7, 35.9, 22.6, 33.1, 30.4,
       52.3, 24.3, 22.9, 34.8, 30.9, 40.1, 23.9, 37.5, 35.5, 42.8, 42.6,
       41.8, 35.8, 37.8, 28.8, 23.6, 35.7, 36.7, 45.2, 44. , 46.2, 35. ,
       43.6, 44.1, 18.4, 29.2, 25.9, 32.1, 36.3, 40. , 25.1, 27.5, 45.6,
       27.8, 24.9, 25.3, 37.9, 27. , 26. , 38.7, 20.8, 36.1, 30.7, 32.3,
       52.9, 21. , 39.7, 25.5, 26.2, 19.3, 38.1, 23.5, 45.5, 23.1, 39.9,
       36.8, 21.8, 41. , 42.2, 34.4, 27.2, 36.5, 29.8, 39.2, 38.4, 36.2,
       48.3, 20. , 22.3, 45.7, 23.7, 22.1, 42.1, 42.4, 18.2, 26.4, 45.3,
       37. , 24.5, 32.2, 59.4, 21.2, 26.7, 30.2, 46.1, 41.3, 38.8, 35.2,
       42.3, 40.7, 46.5, 33.5, 37.3, 30.3, 26.3, 21.7, 36.4, 28.5, 26.9,
       38.6, 31.3, 19.5, 20.1, 40.8, 23.4, 28.3, 38.9, 57.3, 35.6, 49.6,
       44.6, 24.1, 44.5, 41.2, 49.3, 46.3])}
{'DiabetesPedigreeFunction': array([0.627, 0.351, 0.672, 0.167, 2.288, 0.201, 0.248, 0.134, 0.158,
       0.232, 0.191, 0.537, 1.441, 0.398, 0.587, 0.484, 0.551, 0.254,
       0.183, 0.529, 0.704, 0.388, 0.451, 0.263, 0.205, 0.257, 0.487,
       0.245, 0.337, 0.546, 0.851, 0.267, 0.188, 0.512, 0.966, 0.42 ,
       0.665, 0.503, 1.39 , 0.271, 0.696, 0.235, 0.721, 0.294, 1.893,
       0.564, 0.586, 0.344, 0.305, 0.491, 0.526, 0.342, 0.467, 0.718,
       0.962, 1.781, 0.173, 0.304, 0.27 , 0.699, 0.258, 0.203, 0.855,
       0.845, 0.334, 0.189, 0.867, 0.411, 0.583, 0.231, 0.396, 0.14 ,
       0.391, 0.37 , 0.307, 0.102, 0.767, 0.237, 0.227, 0.698, 0.178,
       0.324, 0.153, 0.165, 0.443, 0.261, 0.277, 0.761, 0.255, 0.13 ,
       0.323, 0.356, 0.325, 1.222, 0.179, 0.262, 0.283, 0.93 , 0.801,
       0.207, 0.287, 0.336, 0.247, 0.199, 0.543, 0.192, 0.588, 0.539,
       0.22 , 0.654, 0.223, 0.759, 0.26 , 0.404, 0.186, 0.278, 0.496,
       0.452, 0.403, 0.741, 0.361, 1.114, 0.457, 0.647, 0.088, 0.597,
       0.532, 0.703, 0.159, 0.268, 0.286, 0.318, 0.272, 0.572, 0.096,
       1.4  , 0.218, 0.085, 0.399, 0.432, 1.189, 0.687, 0.137, 0.637,
       0.833, 0.229, 0.817, 0.204, 0.368, 0.743, 0.722, 0.256, 0.709,
       0.471, 0.495, 0.18 , 0.542, 0.773, 0.678, 0.719, 0.382, 0.319,
       0.19 , 0.956, 0.084, 0.725, 0.299, 0.244, 0.745, 0.615, 1.321,
       0.64 , 0.142, 0.374, 0.383, 0.578, 0.136, 0.395, 0.187, 0.905,
       0.15 , 0.874, 0.236, 0.787, 0.407, 0.605, 0.151, 0.289, 0.355,
       0.29 , 0.375, 0.164, 0.431, 0.742, 0.514, 0.464, 1.224, 1.072,
       0.805, 0.209, 0.666, 0.101, 0.198, 0.652, 2.329, 0.089, 0.645,
       0.238, 0.394, 0.293, 0.479, 0.686, 0.831, 0.582, 0.446, 0.402,
       1.318, 0.329, 1.213, 0.427, 0.282, 0.143, 0.38 , 0.284, 0.249,
       0.926, 0.557, 0.092, 0.655, 1.353, 0.612, 0.2  , 0.226, 0.997,
       0.933, 1.101, 0.078, 0.24 , 1.136, 0.128, 0.422, 0.251, 0.677,
       0.296, 0.454, 0.744, 0.881, 0.28 , 0.259, 0.619, 0.808, 0.34 ,
       0.434, 0.757, 0.613, 0.692, 0.52 , 0.412, 0.84 , 0.839, 0.156,
       0.215, 0.326, 1.391, 0.875, 0.313, 0.433, 0.626, 1.127, 0.315,
       0.345, 0.129, 0.527, 0.197, 0.731, 0.148, 0.123, 0.127, 0.122,
       1.476, 0.166, 0.932, 0.343, 0.893, 0.331, 0.472, 0.673, 0.389,
       0.485, 0.349, 0.279, 0.346, 0.252, 0.243, 0.58 , 0.559, 0.302,
       0.569, 0.378, 0.385, 0.499, 0.306, 0.234, 2.137, 1.731, 0.545,
       0.225, 0.816, 0.528, 0.509, 1.021, 0.821, 0.947, 1.268, 0.221,
       0.66 , 0.239, 0.949, 0.444, 0.463, 0.803, 1.6  , 0.944, 0.196,
       0.241, 0.161, 0.135, 0.376, 1.191, 0.702, 0.674, 1.076, 0.534,
       1.095, 0.554, 0.624, 0.219, 0.507, 0.561, 0.421, 0.516, 0.264,
       0.328, 0.233, 0.108, 1.138, 0.147, 0.727, 0.435, 0.497, 0.23 ,
       0.955, 2.42 , 0.658, 0.33 , 0.51 , 0.285, 0.415, 0.381, 0.832,
       0.498, 0.212, 0.364, 1.001, 0.46 , 0.733, 0.416, 0.705, 1.022,
       0.269, 0.6  , 0.571, 0.607, 0.17 , 0.21 , 0.126, 0.711, 0.466,
       0.162, 0.419, 0.63 , 0.365, 0.536, 1.159, 0.629, 0.292, 0.145,
       1.144, 0.174, 0.547, 0.163, 0.738, 0.314, 0.968, 0.409, 0.297,
       0.525, 0.154, 0.771, 0.107, 0.493, 0.717, 0.917, 0.501, 1.251,
       0.735, 0.804, 0.661, 0.549, 0.825, 0.423, 1.034, 0.16 , 0.341,
       0.68 , 0.591, 0.3  , 0.121, 0.502, 0.401, 0.601, 0.748, 0.338,
       0.43 , 0.892, 0.813, 0.693, 0.575, 0.371, 0.206, 0.417, 1.154,
       0.925, 0.175, 1.699, 0.682, 0.194, 0.4  , 0.1  , 1.258, 0.482,
       0.138, 0.593, 0.878, 0.157, 1.282, 0.141, 0.246, 1.698, 1.461,
       0.347, 0.362, 0.393, 0.144, 0.732, 0.115, 0.465, 0.649, 0.871,
       0.149, 0.695, 0.303, 0.61 , 0.73 , 0.447, 0.455, 0.133, 0.155,
       1.162, 1.292, 0.182, 1.394, 0.217, 0.631, 0.88 , 0.614, 0.332,
       0.366, 0.181, 0.828, 0.335, 0.856, 0.886, 0.439, 0.253, 0.598,
       0.904, 0.483, 0.565, 0.118, 0.177, 0.176, 0.295, 0.441, 0.352,
       0.826, 0.97 , 0.595, 0.317, 0.265, 0.646, 0.426, 0.56 , 0.515,
       0.453, 0.785, 0.734, 1.174, 0.488, 0.358, 1.096, 0.408, 1.182,
       0.222, 1.057, 0.766, 0.171])}
{'Age': array([50, 31, 32, 21, 33, 30, 26, 29, 53, 54, 34, 57, 59, 51, 27, 41, 43,
       22, 38, 60, 28, 45, 35, 46, 56, 37, 48, 40, 25, 24, 58, 42, 44, 39,
       36, 23, 61, 69, 62, 55, 65, 47, 52, 66, 49, 63, 67, 72, 81, 64, 70,
       68], dtype=int64)}
{'Outcome': array([1, 0], dtype=int64)}
```

In [13]:
```
p=sns.pairplot(df1, hue = 'Outcome')
```

In [14]:
```
### Visualizaing correlation on Heatmap
plt.figure(figsize=(12,10))
p=sns.heatmap(df1.corr(), annot=True,cmap ='RdYlGn')
```



In [15]:
```
### Finding numerical variables
colname_num = [var for var in df1.columns if df1[var].dtype!='O']
print('There are {} numerical variables\n'.format(len(colname_num)))
print('The numerical variables are :', colname_num)
```

```
There are 9 numerical variables

The numerical variables are : ['Pregnancies', 'Glucose', 'BloodPressure', 'SkinThickness', 'Insulin', 'BMI', 'DiabetesPedigreeFunction', 'Age', 'Outcome']
```

In [16]:
```
df2 = df1.copy()
df2.shape
```

Out[16]:
```
(768, 9)
```

In [17]:
```
for i in colname_num:
    print("Column Names: ", i)
    print("Null Values: ", df2[i].isna().sum())
    print("Mean Values: ",df2[i].mean())
    print("Median Values: ",df2[i].median())
    print("Mode Values: ",df2[i].mode())
    print('-' * 50)
```

```
Column Names:  Pregnancies
Null Values:  0
Mean Values:  3.8450520833333335
Median Values:  3.0
Mode Values:  0    1
Name: Pregnancies, dtype: int64
--------------------------------------------------
Column Names:  Glucose
Null Values:  0
Mean Values:  120.89453125
Median Values:  117.0
Mode Values:  0     99
1    100
Name: Glucose, dtype: int64
--------------------------------------------------
Column Names:  BloodPressure
Null Values:  0
Mean Values:  69.10546875
Median Values:  72.0
Mode Values:  0    70
Name: BloodPressure, dtype: int64
--------------------------------------------------
Column Names:  SkinThickness
Null Values:  0
Mean Values:  20.536458333333332
Median Values:  23.0
Mode Values:  0    0
Name: SkinThickness, dtype: int64
--------------------------------------------------
Column Names:  Insulin
Null Values:  0
Mean Values:  79.79947916666667
Median Values:  30.5
Mode Values:  0    0
Name: Insulin, dtype: int64
--------------------------------------------------
Column Names:  BMI
Null Values:  0
Mean Values:  31.992578124999998
Median Values:  32.0
Mode Values:  0    32.0
Name: BMI, dtype: float64
--------------------------------------------------
Column Names:  DiabetesPedigreeFunction
Null Values:  0
Mean Values:  0.47187630208333325
Median Values:  0.3725
Mode Values:  0    0.254
1    0.258
Name: DiabetesPedigreeFunction, dtype: float64
--------------------------------------------------
Column Names:  Age
Null Values:  0
Mean Values:  33.240885416666664
Median Values:  29.0
Mode Values:  0    22
Name: Age, dtype: int64
--------------------------------------------------
Column Names:  Outcome
Null Values:  0
Mean Values:  0.3489583333333333
Median Values:  0.0
Mode Values:  0    0
Name: Outcome, dtype: int64
--------------------------------------------------
```

In [18]:
```
df2.head(15)
```

Out[18]:

	Pregnancies	Glucose	BloodPressure	SkinThickness	Insulin	BMI	DiabetesPedigreeFunction	Age	Outcome
0	6	148	72	35	0	33.6	0.627	50	1
1	1	85	66	29	0	26.6	0.351	31	0
2	8	183	64	0	0	23.3	0.672	32	1
3	1	89	66	23	94	28.1	0.167	21	0
4	0	137	40	35	168	43.1	2.288	33	1
5	5	116	74	0	0	25.6	0.201	30	0
6	3	78	50	32	88	31.0	0.248	26	1
7	10	115	0	0	0	35.3	0.134	29	0
8	2	197	70	45	543	30.5	0.158	53	1
9	8	125	96	0	0	0.0	0.232	54	1
10	4	110	92	0	0	37.6	0.191	30	0
11	10	168	74	0	0	38.0	0.537	34	1
12	10	139	80	0	0	27.1	1.441	57	0
13	1	189	60	23	846	30.1	0.398	59	1
14	5	166	72	19	175	25.8	0.587	51	1

In [19]:
```
for col in df2.columns:
    print(f"{col} has {df2[col].nunique()} categories\n")
```

```
Pregnancies has 17 categories

Glucose has 136 categories

BloodPressure has 47 categories

SkinThickness has 51 categories

Insulin has 186 categories

BMI has 248 categories

DiabetesPedigreeFunction has 517 categories

Age has 52 categories

Outcome has 2 categories
```

```
In [20]: df3 = df2.copy()
         df3.columns
```

```
Out[20]: Index(['Pregnancies', 'Glucose', 'BloodPressure', 'SkinThickness', 'Insulin',
               'BMI', 'DiabetesPedigreeFunction', 'Age', 'Outcome'],
              dtype='object')
```

```
In [21]: ### Spliting Data into X and y
         X = df3.values[:,:-1]
         y = df3.values[:,-1]
         print('X:',X.shape)
         print('*' * 13)
         print('y:',y.shape)
```

```
X: (768, 8)
*************
y: (768,)
```

```
In [22]: ### Feature Scaling
         from sklearn.preprocessing import StandardScaler
         scaler = StandardScaler()
         scaler.fit(X)
         X = scaler.transform(X)
         #x = scaler.fit_transform(x)
         print(X)
```

```
[[ 0.63994726  0.84832379  0.14964075 ...  0.20401277  0.46849198
   1.4259954 ]
 [-0.84488505 -1.12339636 -0.16054575 ... -0.68442195 -0.36506078
  -0.19067191]
 [ 1.23388019  1.94372388 -0.26394125 ... -1.10325546  0.60439732
  -0.10558415]
 ...
 [ 0.3429808   0.00330087  0.14964075 ... -0.73518964 -0.68519336
  -0.27575966]
 [-0.84488505  0.1597866  -0.47073225 ... -0.24020459 -0.37110101
   1.17073215]
 [-0.84488505 -0.8730192   0.04624525 ... -0.20212881 -0.47378505
  -0.87137393]]
```

```
In [23]: y = y.astype(int) ### convert y in to integer always perform this operation
```

```
In [24]: ### Spliting into Training and Testing Data
         from sklearn.model_selection import train_test_split
         X_train, X_test, y_train, y_test = train_test_split(X,y,test_size=0.2,random_state=42)

         print("X_train: ",X_train.shape)
         print("X_test: ",X_test.shape)
         print("y_train: ",y_train.shape)
         print("y_test: ",y_test.shape)
```

```
X_train:  (614, 8)
X_test:  (154, 8)
y_train:  (614,)
y_test:  (154,)
```

```
In [25]: train_scores = []
         test_scores = []
         #importing model
         from sklearn.neighbors import KNeighborsClassifier
         for i in range(1,30):
             #create a model object
             knn = KNeighborsClassifier(i, metric="manhattan")
             #train the model object
             knn.fit(X_train,y_train)
             #predict score using the model
             train_scores.append(knn.score(X_train,y_train))
             test_scores.append(knn.score(X_test,y_test))
```

```
In [26]: ### Printing train and test scores
         print("Train Scores: ",train_scores)
         print('-' * 138)
         print("Test Scores: ",test_scores)
```

```
Train Scores:  [1.0, 0.8436482084690554, 0.8599348534201955, 0.8078175895765473, 0.8078175895765473, 0.8094462540716613, 0.8110749185667753, 0.7964169381107492, 0.799674267100977
2, 0.7882736156351792, 0.7850162866449512, 0.7736156351791531, 0.7817589576547231, 0.7768729641693811, 0.7768729641693811, 0.7687296416938111, 0.7736156351791531, 0.7719869706840
391, 0.7752442996742671, 0.7719869706840391, 0.7785016286644951, 0.7719869706840391, 0.7866449511400652, 0.7768729641693811, 0.7785016286644951, 0.7801302931596091, 0.77361563517
91531, 0.7752442996742671, 0.7719869706840391]
------------------------------------------------------------------------------------------------------------------------------------------
Test Scores:  [0.6623376623376623, 0.6688311688311688, 0.6688311688311688, 0.6753246753246753, 0.6623376623376623, 0.6818181818181818, 0.6883116883116883, 0.7207792207792207, 0.7
077922077922078, 0.7337662337662337, 0.7142857142857143, 0.7337662337662337, 0.7142857142857143, 0.7402597402597403, 0.7402597402597403, 0.7337662337662337, 0.7467532467532467,
0.7337662337662337, 0.7272727272727273, 0.7467532467532467, 0.7922077922077922, 0.7857142857142857, 0.7922077922077922, 0.7597402597402597, 0.7922077922077922, 0.798701298701298
7, 0.7727272727272727, 0.7662337662337663, 0.7662337662337663]
```

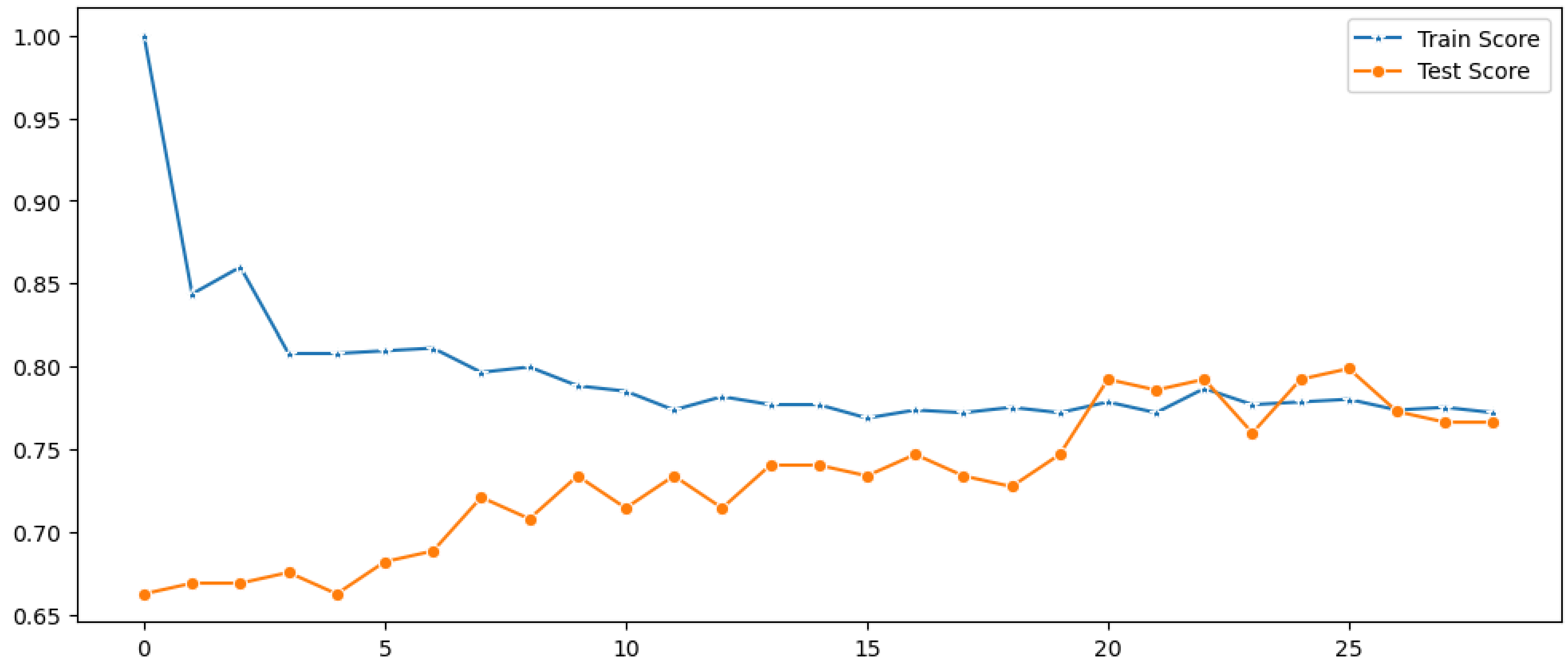
```
In [27]: ### Score that comes from testing on the same datapoints that were used for training
         max_train_score = max(train_scores)
         train_scores_ind = [i for i, v in enumerate(train_scores) if v == max_train_score]
         print('Max train score {} % and k = {}'.format(max_train_score*100,list(map(lambda x: x+1, train_scores_ind))))
```

```
Max train score 100.0 % and k = [1]
```

```
In [28]: ### Score that comes from testing on the datapoints that were split in the beginning to be used for testing solely
         max_test_score = max(test_scores)
         test_scores_ind = [i for i, v in enumerate(test_scores) if v == max_test_score]
         print('Max test score {} % and k = {}'.format(max_test_score*100,list(map(lambda x: x+1, test_scores_ind))))
```

```
Max test score 79.87012987012987 % and k = [26]
```

In [29]:
```python
### Visualizing the scores
plt.figure(figsize=(12,5))
p = sns.lineplot(train_scores,marker='*',label='Train Score')
p = sns.lineplot(test_scores,marker='o',label='Test Score')
```



In [30]:
```python
#importing model
from sklearn.neighbors import KNeighborsClassifier
#create a model object
model_knn = KNeighborsClassifier(26, metric="manhattan")
#train the model object
model_knn.fit(X_train,y_train)
#predict using the model
y_pred = model_knn.predict(X_test)
print(y_pred)
```

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

In [31]:
```python
# Checking confusion matrix for the model
cfm = confusion_matrix(y_test,y_pred)
dff = pd.DataFrame(cfm)
dff.style.set_properties(**{"background-color": "#F3FFFF","color":"black","border": "2px  solid black"})
```

Out[31]:

	0	1
0	93	6
1	25	30

In [32]:
```python
# Checking classification report score for the model
cr = classification_report(y_test,y_pred)
print("Classification report: ")
print(cr)

# Checking accuracy score for the model
acc = accuracy_score(y_test,y_pred)
print("Accuracy of the model: ",acc)
```

```
Classification report: 
              precision    recall  f1-score   support

           0       0.79      0.94      0.86        99
           1       0.83      0.55      0.66        55

    accuracy                           0.80       154
   macro avg       0.81      0.74      0.76       154
weighted avg       0.80      0.80      0.79       154

Accuracy of the model:  0.7987012987012987
```

Made with ❤ by Zahid Salim Shaikh

In [ ]: