In [1]:
```
# Diabetes Prediction Using Machine Learning
```

In [3]:
```
# importing the necessary libraries
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
sns.set()
import warnings
warnings.filterwarnings('ignore')
%matplotlib inline
```

Basic Data Science and ML Pipeline

In [4]:
```
#Loading the dataset
diabetes_data = pd.read_csv('diabetes.csv')

#Print the first 5 rows of the dataframe.
diabetes_data.head()
```

Out[4]:

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

Basic EDA and statistical analysis

In [7]:
```
diabetes_data.info(verbose=True)
```

```
<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
```

DataFrame.describe() method generates descriptive statistics that summarize the central tendency, dispersion and shape of a dataset's distribution, excluding NaN values. This method tells us a lot of things about a dataset. One important thing is that the describe()

method deals only with numeric values. It doesn't work with any categorical values. So if there are any categorical values in a column the describe() method will ignore it and display summary for the other columns unless parameter include="all" is passed.

Now, let's understand the statistics that are generated by the describe() method:

- count tells us the number of NoN-empty rows in a feature.
- mean tells us the mean value of that feature.
- std tells us the Standard Deviation Value of that feature.
- min tells us the minimum value of that feature.
- 25%, 50%, and 75% are the percentile/quartile of each features. This quartile information helps us to detect Outliers.
- max tells us the maximum value of that feature.

In [8]:
```
diabetes_data.describe()
```

Out[8]:

	Pregnancies	Glucose	BloodPressure	SkinThickness	Insulin	BMI	DiabetesPe
count	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	
std	3.369578	31.972618	19.355807	15.952218	115.244002	7.884160	
min	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	
25%	1.000000	99.000000	62.000000	0.000000	0.000000	27.300000	
50%	3.000000	117.000000	72.000000	23.000000	30.500000	32.000000	
75%	6.000000	140.250000	80.000000	32.000000	127.250000	36.600000	
max	17.000000	199.000000	122.000000	99.000000	846.000000	67.100000	

In [9]:
```
diabetes_data.describe().T  # creating the trsnspose of the description of the Data
```

Out[9]:

	count	mean	std	min	25%	50%	75%	
Pregnancies	768.0	3.845052	3.369578	0.000	1.00000	3.0000	6.00000	
Glucose	768.0	120.894531	31.972618	0.000	99.00000	117.0000	140.25000	19
BloodPressure	768.0	69.105469	19.355807	0.000	62.00000	72.0000	80.00000	12
SkinThickness	768.0	20.536458	15.952218	0.000	0.00000	23.0000	32.00000	9
Insulin	768.0	79.799479	115.244002	0.000	0.00000	30.5000	127.25000	84
BMI	768.0	31.992578	7.884160	0.000	27.30000	32.0000	36.60000	6
DiabetesPedigreeFunction	768.0	0.471876	0.331329	0.078	0.24375	0.3725	0.62625	
Age	768.0	33.240885	11.760232	21.000	24.00000	29.0000	41.00000	8
Outcome	768.0	0.348958	0.476951	0.000	0.00000	0.0000	1.00000	

The Question creeping out of this summary

Can minimum value of below listed columns be zero (0)?

On these columns, a value of zero does not make sense and thus indicates missing value.

Following columns or variables have an invalid zero value:

1. Glucose
2. BloodPressure
3. SkinThickness
4. Insulin
5. BMI

It is better to replace zeros with nan since after that counting them would be easier and zeros need to be replaced with suitable values

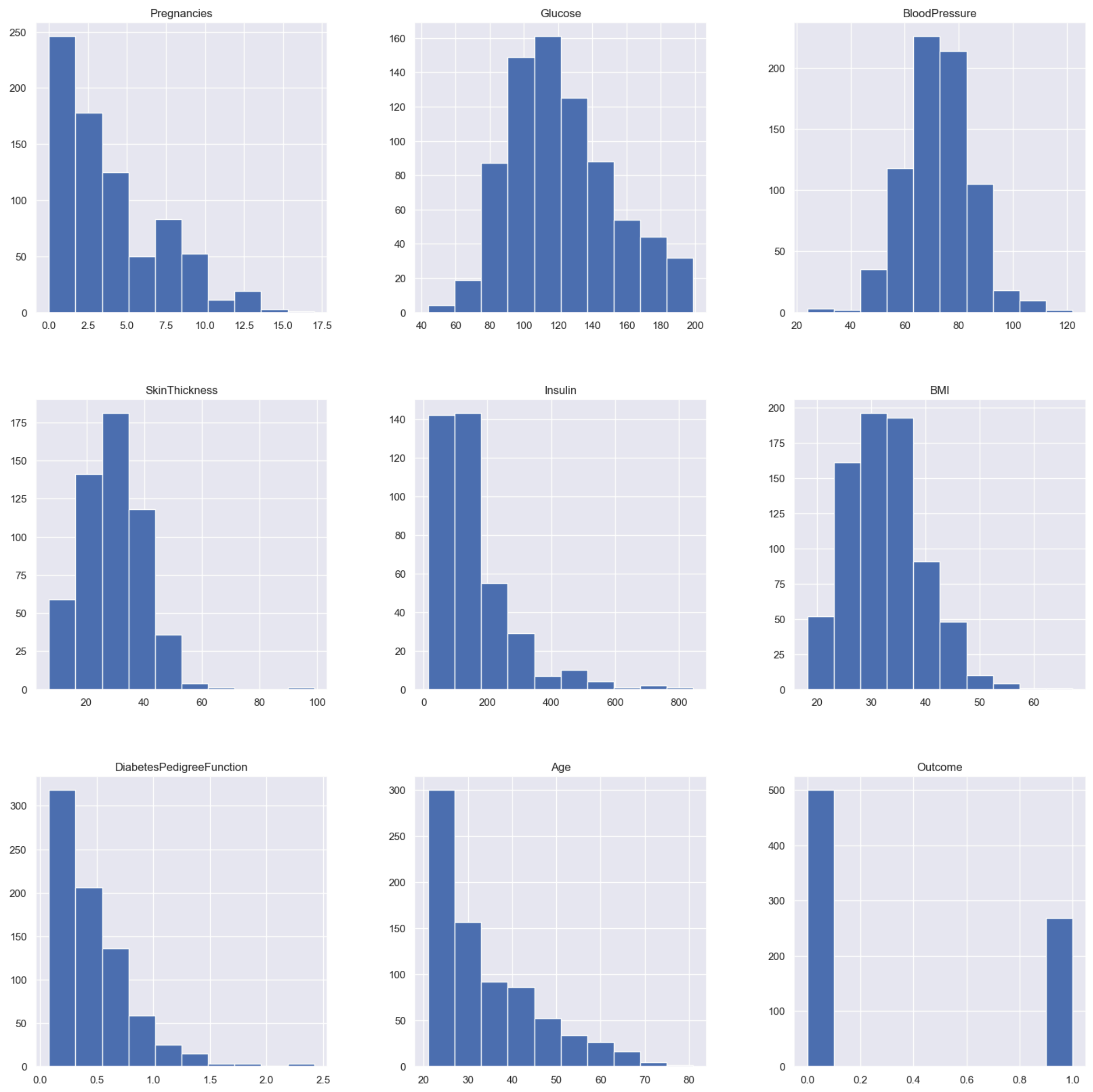
In [10]:
```
diabetes_data_copy = diabetes_data.copy(deep = True) # creating the copy of the dat
# replacing the 0 values with Nan
diabetes_data_copy[['Glucose','BloodPressure','SkinThickness','Insulin','BMI']] = d

## showing the count of Nans
print(diabetes_data_copy.isnull().sum())
```

```
Pregnancies                   0
Glucose                       5
BloodPressure                35
SkinThickness               227
Insulin                     374
BMI                          11
DiabetesPedigreeFunction      0
Age                           0
Outcome                       0
dtype: int64
```

In [ ]:

To fill these Nan values the data distribution needs to be understood

In [11]:
```
p = diabetes_data_copy.hist(figsize = (20,20))
```



Aiming to impute nan values for the columns in accordance with their distribution

```
In [12]: diabetes_data_copy['Glucose'].fillna(diabetes_data_copy['Glucose'].mean(), inplace
         diabetes_data_copy.isna().sum()
```

```
Out[12]: Pregnancies                   0
         Glucose                       0
         BloodPressure                35
         SkinThickness               227
         Insulin                     374
         BMI                          11
         DiabetesPedigreeFunction      0
         Age                           0
         Outcome                       0
         dtype: int64
```

```
In [13]: diabetes_data_copy['BloodPressure'].fillna(diabetes_data_copy['BloodPressure'].mear
         diabetes_data_copy.isna().sum()
```

Out[13]:
```
Pregnancies                   0
Glucose                       0
BloodPressure                 0
SkinThickness               227
Insulin                     374
BMI                          11
DiabetesPedigreeFunction      0
Age                           0
Outcome                       0
dtype: int64
```

In [14]:
```
diabetes_data_copy['SkinThickness'].fillna(diabetes_data_copy['SkinThickness'].medi
diabetes_data_copy.isna().sum()
```

Out[14]:
```
Pregnancies                   0
Glucose                       0
BloodPressure                 0
SkinThickness                 0
Insulin                     374
BMI                          11
DiabetesPedigreeFunction      0
Age                           0
Outcome                       0
dtype: int64
```

In [15]:
```
diabetes_data_copy['Insulin'].fillna(diabetes_data_copy['Insulin'].median(), inplac
diabetes_data_copy.isna().sum()
```

Out[15]:
```
Pregnancies                  0
Glucose                      0
BloodPressure                0
SkinThickness                0
Insulin                      0
BMI                         11
DiabetesPedigreeFunction     0
Age                          0
Outcome                      0
dtype: int64
```

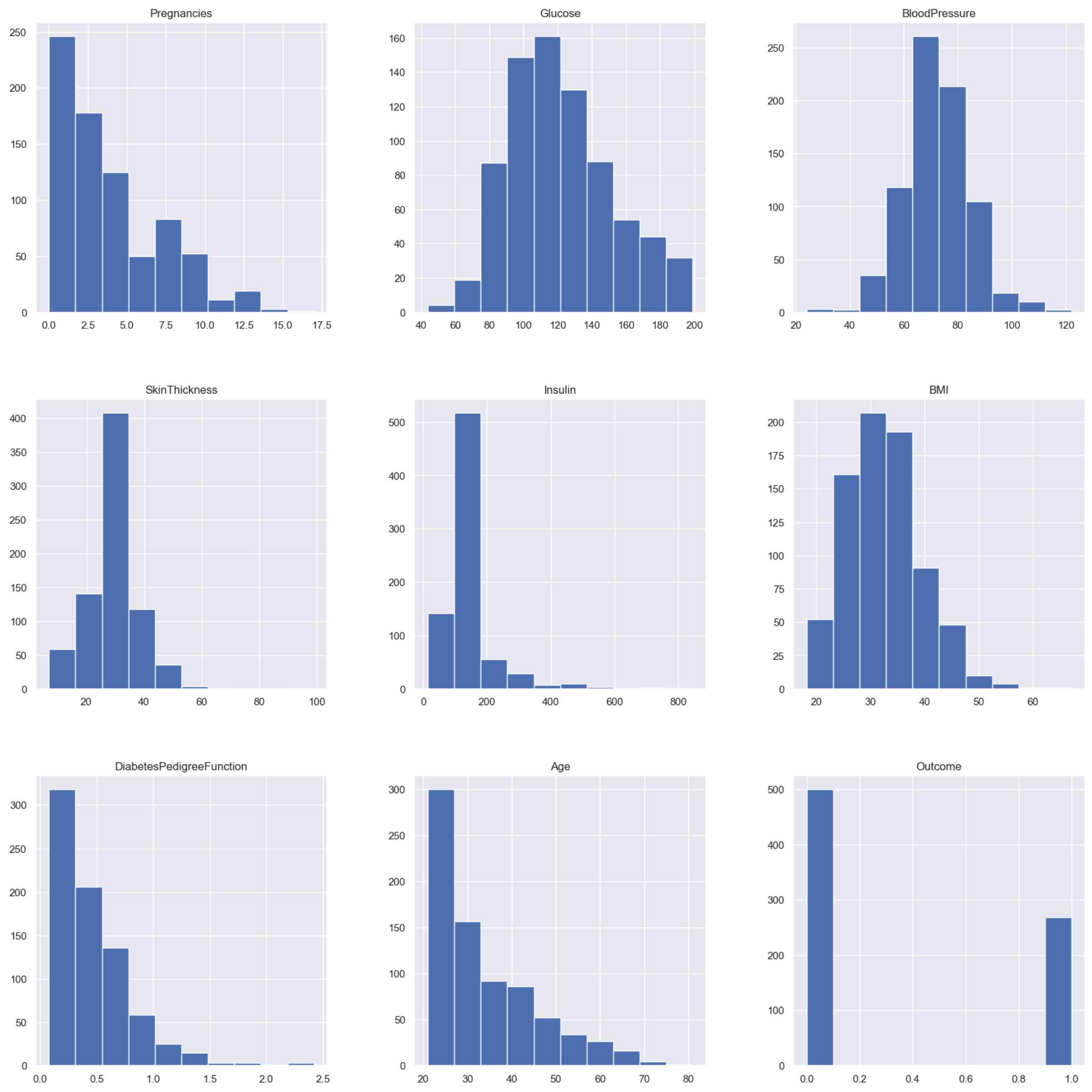
In [16]:
```
diabetes_data_copy['BMI'].fillna(diabetes_data_copy['BMI'].median(), inplace = True
diabetes_data_copy.isna().sum()
```

Out[16]:
```
Pregnancies                 0
Glucose                     0
BloodPressure               0
SkinThickness               0
Insulin                     0
BMI                         0
DiabetesPedigreeFunction    0
Age                         0
Outcome                     0
dtype: int64
```

Finally we have imputated all the missing values

Plotting after Nan removal

In [17]:
```
p = diabetes_data_copy.hist(figsize = (20,20))
```



Skewness

A ***left-skewed distribution*** has a long left tail. Left-skewed distributions are also called negatively-skewed distributions. That’s because there is a long tail in the negative direction on the number line. The mean is also to the left of the peak.

A ***right-skewed distribution*** has a long right tail. Right-skewed distributions are also called positive-skew distributions. That’s because there is a long tail in the positive direction on the number line. The mean is also to the right of the peak.

to learn more about skewness

https://www.statisticshowto.datasciencecentral.com/probability-and-statistics/skewed-distribution/

In [18]:
```
## observing the shape of the data
diabetes_data.shape
```

Out[18]:
```
(768, 9)
```

In [19]:
```
diabetes_data.info()
```

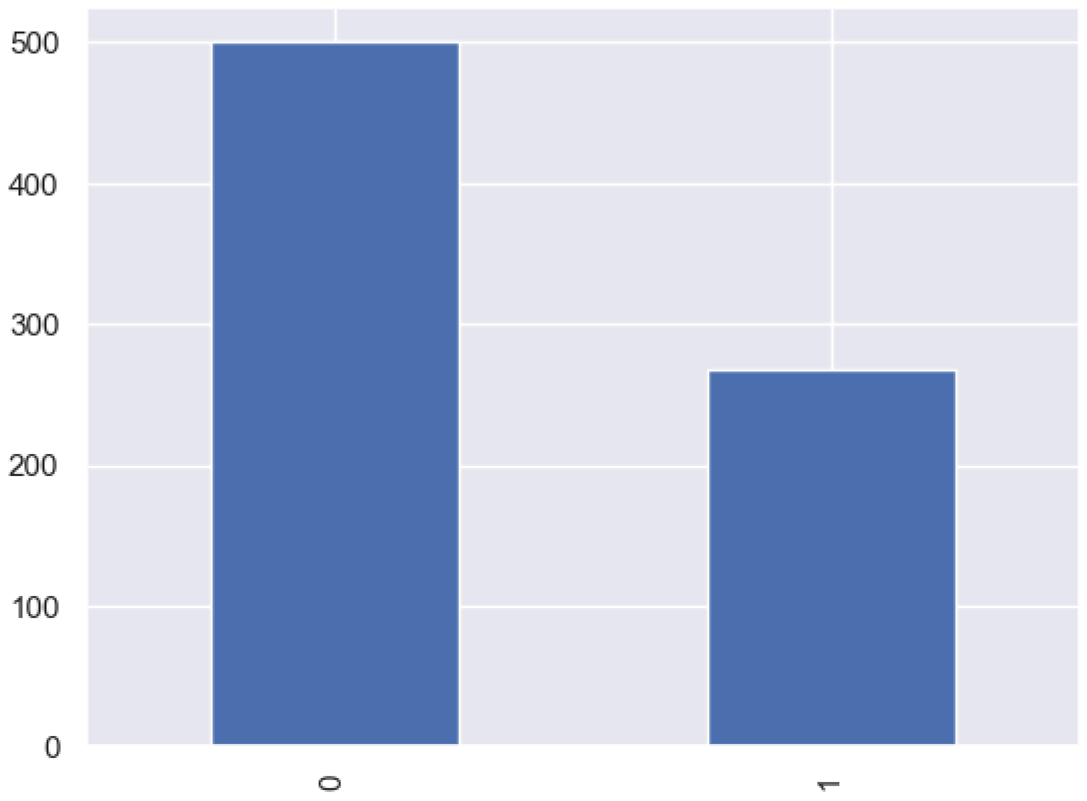
```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 768 entries, 0 to 767
Data columns (total 9 columns):
 #   Column                    Non-Null Count  Dtype  
---  ------                    --------------  -----  
 0   Pregnancies               768 non-null    int64  
 1   Glucose                   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 [20]:
```
diabetes_data.dtypes
```

Out[20]:
```
Pregnancies                   int64
Glucose                       int64
BloodPressure                 int64
SkinThickness                 int64
Insulin                       int64
BMI                         float64
DiabetesPedigreeFunction    float64
Age                           int64
Outcome                       int64
dtype: object
```

In [22]:
```
## checking the balance of the data by plotting the count of outcomes by their valu
color_wheel = {1: "#0392cf",
               2: "#7bc043"}
colors = diabetes_data["Outcome"].map(lambda x: color_wheel.get(x + 1))
print(diabetes_data.Outcome.value_counts())
p=diabetes_data.Outcome.value_counts().plot(kind="bar")
```

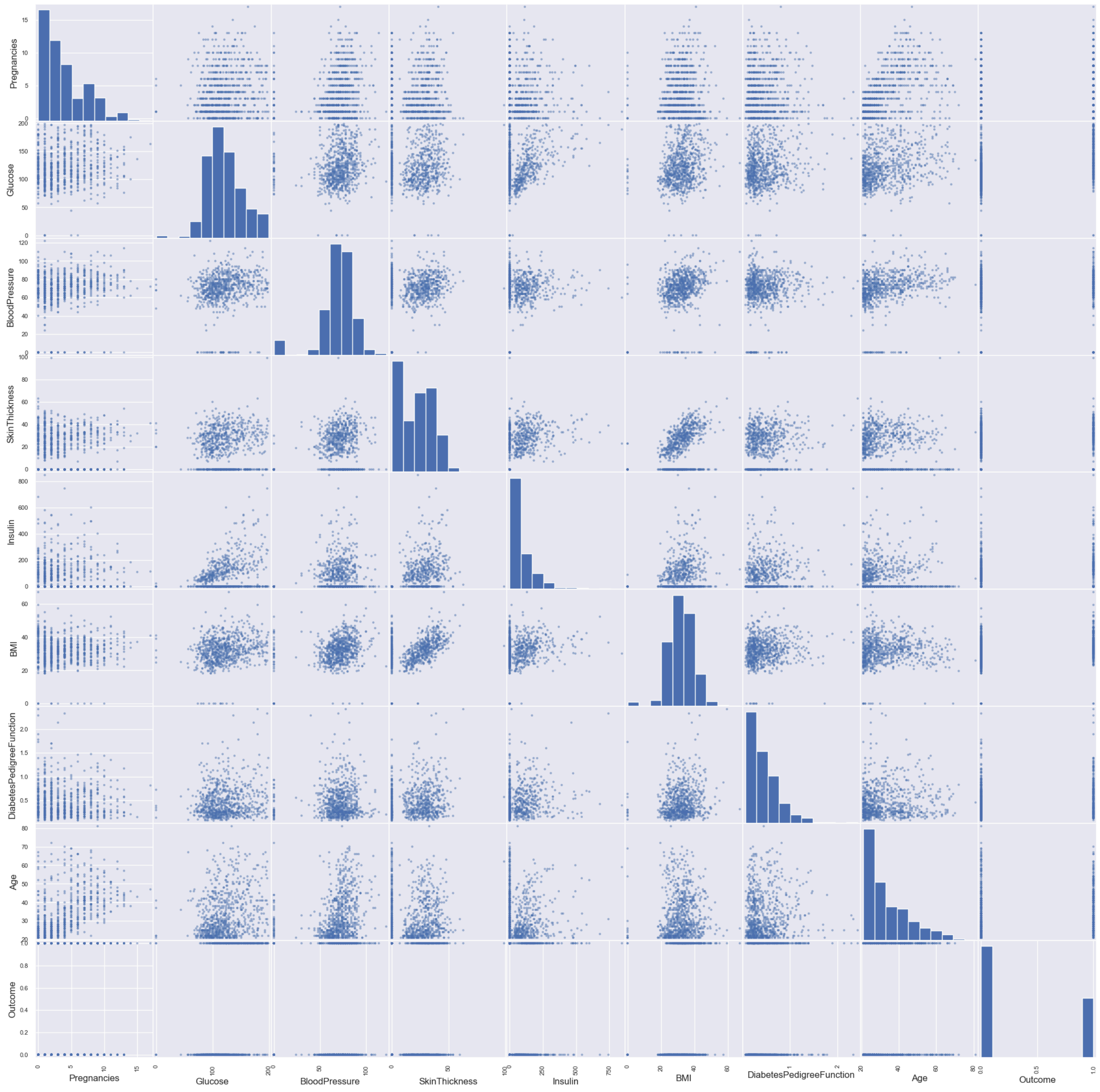
```
0    500
1    268
Name: Outcome, dtype: int64
```



The above graph shows that the data is biased towards datapoints having outcome value as 0 where it means that diabetes was not present actually. The number of non-diabetics is almost twice the number of diabetic patients

Scatter matrix of uncleaned data

In [23]:
```
from pandas.plotting import scatter_matrix
p=scatter_matrix(diabetes_data,figsize=(25, 25))
```

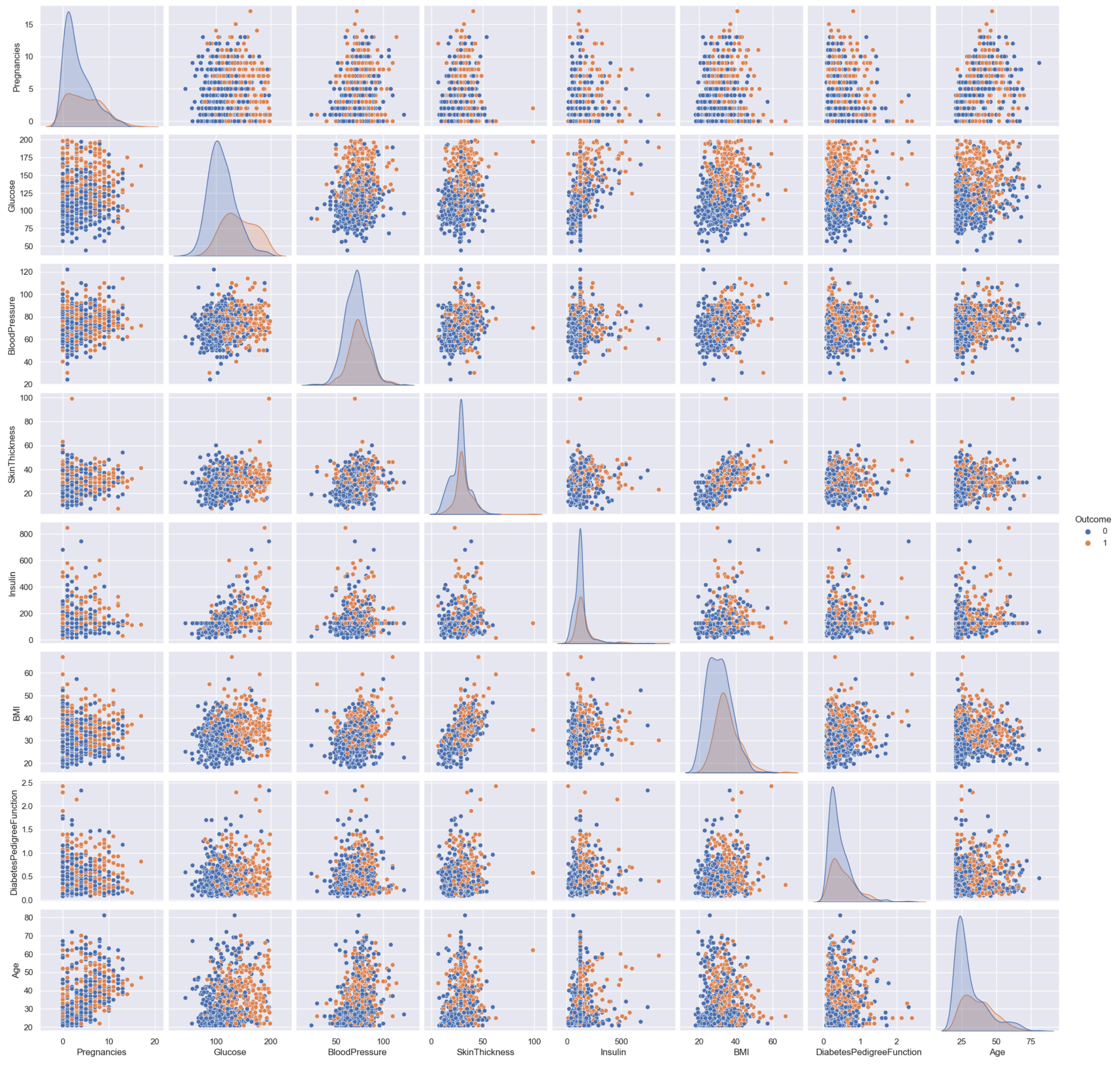


The pairs plot builds on two basic figures, the histogram and the scatter plot. The histogram on the diagonal allows us to see the distribution of a single variable while the scatter plots on the upper and lower triangles show the relationship (or lack thereof) between two variables.

For Reference: https://towardsdatascience.com/visualizing-data-with-pair-plots-in-python-f228cf529166

Pair plot for clean data

In [24]:
```
p=sns.pairplot(diabetes_data_copy, hue = 'Outcome')
```

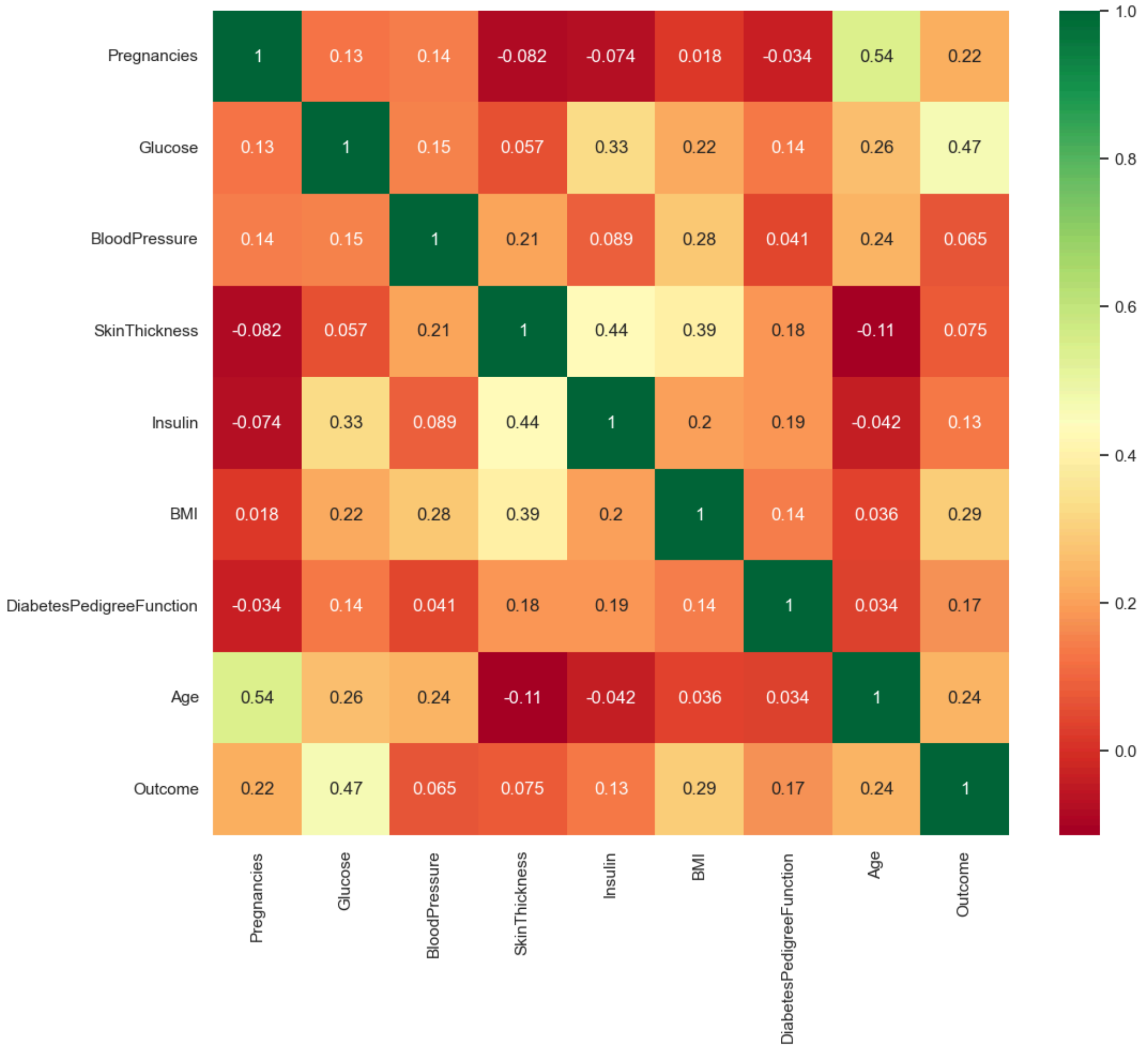


Pearson's Correlation Coefficient: helps you find out the relationship between two quantities. It gives you the measure of the strength of association between two variables. The value of Pearson's Correlation Coefficient can be between -1 to +1. 1 means that they are highly correlated and 0 means no correlation.

A heat map is a two-dimensional representation of information with the help of colors. Heat maps can help the user visualize simple or complex information.

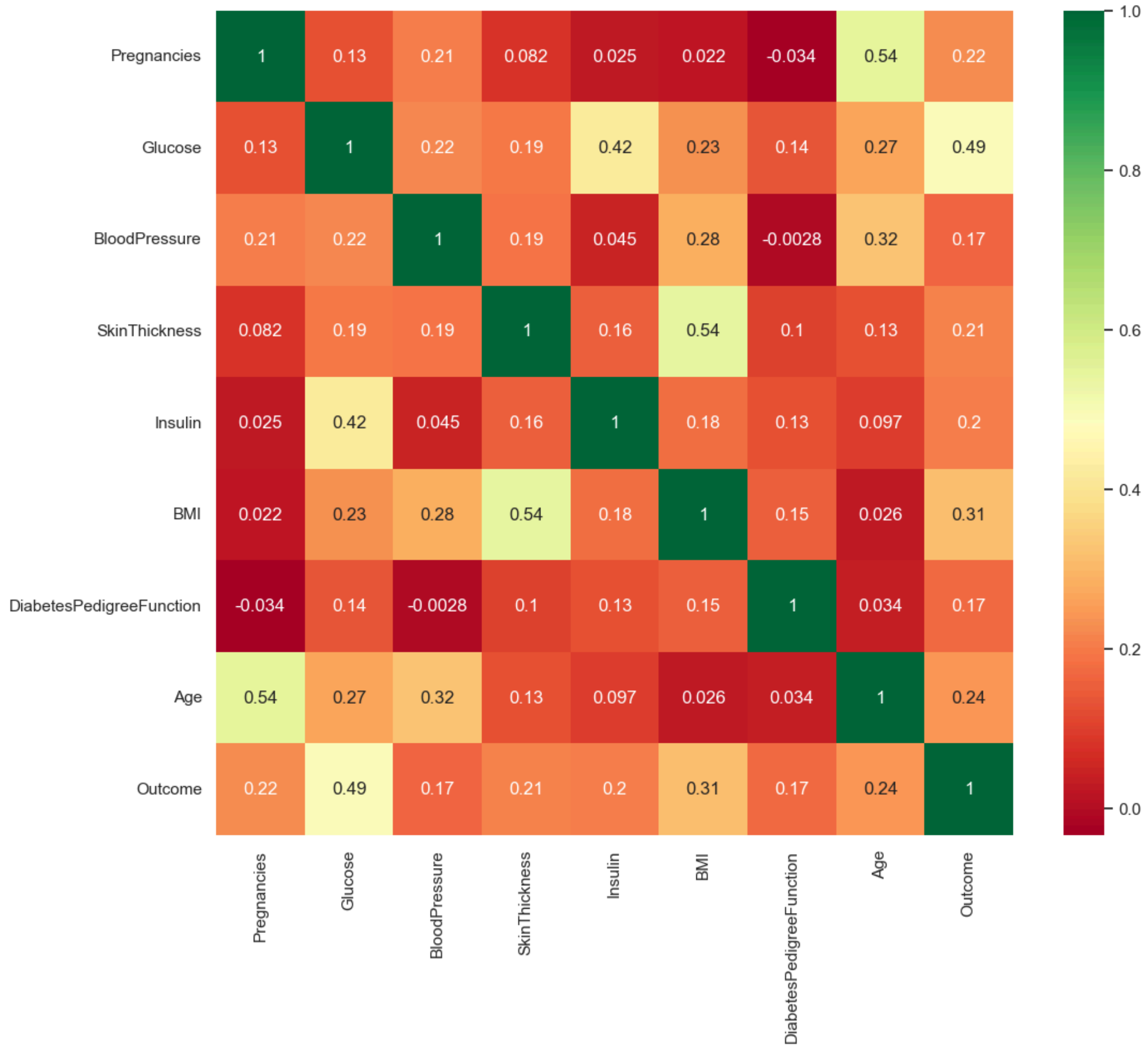
Heatmap for unclean data

In [25]:
```
plt.figure(figsize=(12,10))  # on this line I just set the size of figure to 12 by
p=sns.heatmap(diabetes_data.corr(), annot=True,cmap ='RdYlGn')  # seaborn has very
```



Heatmap for clean data

In [26]:
```
plt.figure(figsize=(12,10))  # on this line I just set the size of figure to 12 by
p=sns.heatmap(diabetes_data_copy.corr(), annot=True,cmap ='RdYlGn')  # seaborn has
```



Scaling the data

data Z is rescaled such that μ = 0 and **σ** = 1, and is done through this formula:

$$z = \frac{x_i - \mu}{\sigma}$$

to learn more about scaling techniques

https://medium.com/@rrfd/standardize-or-normalize-examples-in-python-e3f174b65dfc
https://machinelearningmastery.com/rescaling-data-for-machine-learning-in-python-with-

scikit-learn/

In [27]:
```
# dataframe before transformation
diabetes_data_copy.head()
```

Out[27]:

	Pregnancies	Glucose	BloodPressure	SkinThickness	Insulin	BMI	DiabetesPedigreeFunction	A
0	6	148.0	72.0	35.0	125.0	33.6	0.627	
1	1	85.0	66.0	29.0	125.0	26.6	0.351	
2	8	183.0	64.0	29.0	125.0	23.3	0.672	
3	1	89.0	66.0	23.0	94.0	28.1	0.167	
4	0	137.0	40.0	35.0	168.0	43.1	2.288	

In [28]:
```
# scaling the data
from sklearn.preprocessing import StandardScaler
sc_X = StandardScaler()
X =  pd.DataFrame(sc_X.fit_transform(diabetes_data_copy.drop(["Outcome"],axis = 1),
        columns=['Pregnancies', 'Glucose', 'BloodPressure', 'SkinThickness', 'Insul
       'BMI', 'DiabetesPedigreeFunction', 'Age'])
```

In [29]:
```
X.head()  # looking at the transformed data
```

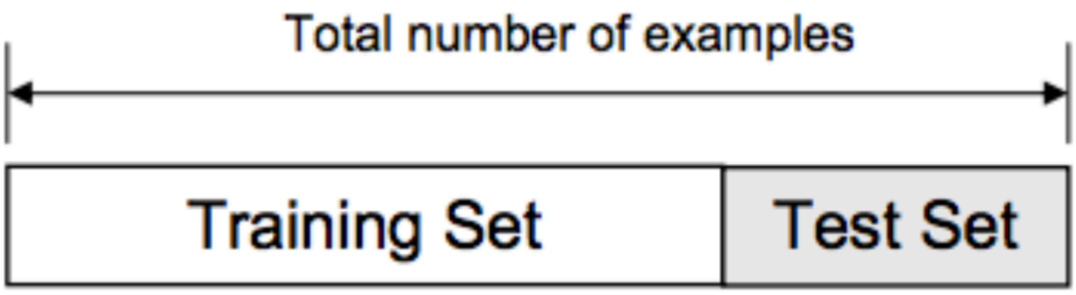
Out[29]:

	Pregnancies	Glucose	BloodPressure	SkinThickness	Insulin	BMI	DiabetesPedigreeFu
0	0.639947	0.865108	-0.033518	0.670643	-0.181541	0.166619	0.4
1	-0.844885	-1.206162	-0.529859	-0.012301	-0.181541	-0.852200	-0.3
2	1.233880	2.015813	-0.695306	-0.012301	-0.181541	-1.332500	0.6
3	-0.844885	-1.074652	-0.529859	-0.695245	-0.540642	-0.633881	-0.9
4	-1.141852	0.503458	-2.680669	0.670643	0.316566	1.549303	5.4

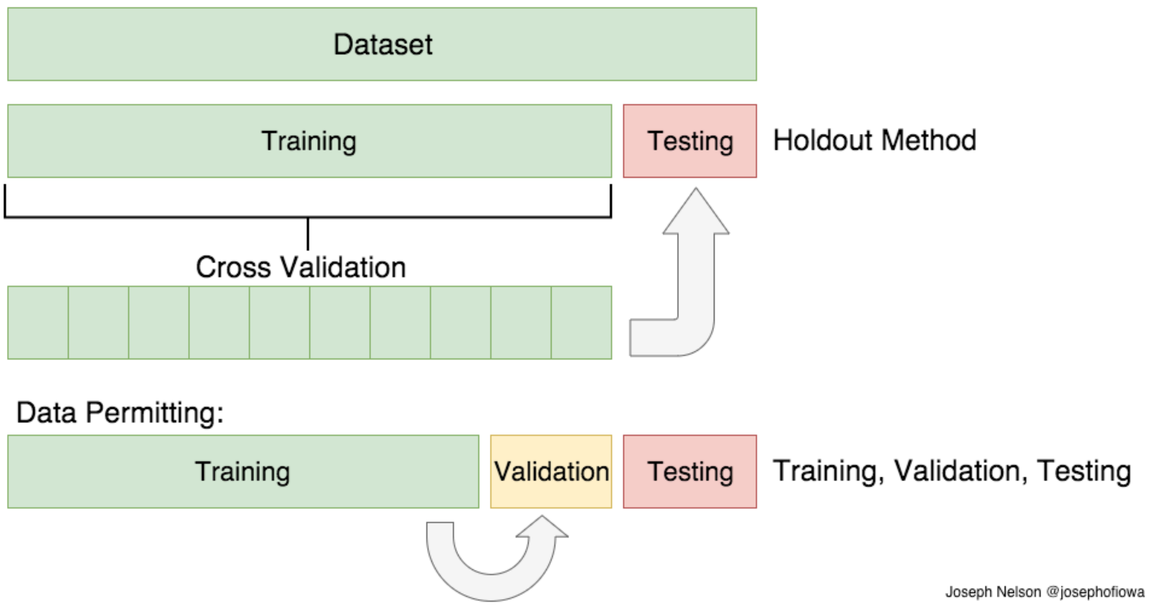
In [30]:
```
#X = diabetes_data.drop("Outcome",axis = 1)
y = diabetes_data_copy.Outcome  # assigning the label column
```

Test Train Split and Cross Validation methods

Train Test Split : To have unknown datapoints to test the data rather than testing with the same points with which the model was trained. This helps capture the model performance much better.



Cross Validation: When model is split into training and testing it can be possible that specific type of data point may go entirely into either training or testing portion. This would lead the model to perform poorly. Hence over-fitting and underfitting problems can be well avoided with cross validation techniques



About Stratify : Stratify parameter makes a split so that the proportion of values in the sample produced will be the same as the proportion of values provided to parameter stratify.

For example, if variable y is a binary categorical variable with values 0 and 1 and there are 25% of zeros and 75% of ones, stratify=y will make sure that your random split has 25% of 0's and 75% of 1's.

For Reference : https://towardsdatascience.com/train-test-split-and-cross-validation-in-python-80b61beca4b6

In [31]:
```
#importing train_test_split
from sklearn.model_selection import train_test_split
X_train,X_test,y_train,y_test = train_test_split(X,y,test_size=1/3,random_state=42,
```

In [32]:
```
from sklearn.neighbors import KNeighborsClassifier


test_scores = []
train_scores = []

for i in range(1,15):
    knn = KNeighborsClassifier(i)
    knn.fit(X_train,y_train)

    train_scores.append(knn.score(X_train,y_train))
    test_scores.append(knn.score(X_test,y_test))
```

In [33]:
```
print(train_scores)
print(test_scores)
```

```
[1.0, 0.84375, 0.8671875, 0.8359375, 0.828125, 0.8046875, 0.814453125, 0.80273437
5, 0.798828125, 0.802734375, 0.798828125, 0.79296875, 0.794921875, 0.796875]
[0.73046875, 0.73046875, 0.74609375, 0.7421875, 0.7421875, 0.72265625, 0.74609375,
0.74609375, 0.74609375, 0.73046875, 0.765625, 0.734375, 0.75, 0.734375]
```

In [34]:
```
## 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
```

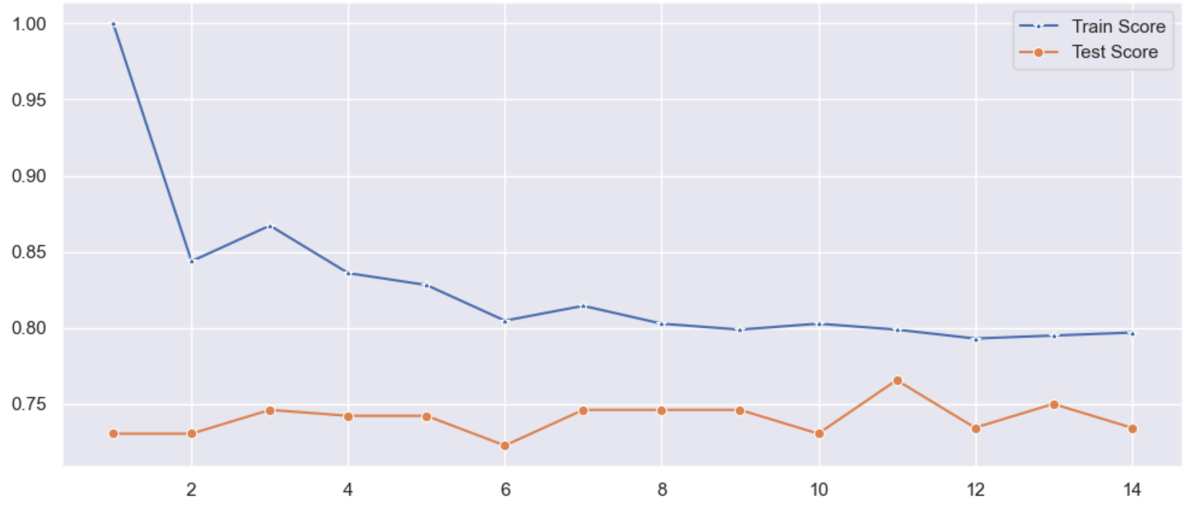
```
Max train score 100.0 % and k = [1]
```

In [35]:
```
## score that comes from testing on the datapoints that were split in the beginning
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:
```

```
Max test score 76.5625 % and k = [11]
```

Result Visualisation

In [36]:
```
plt.figure(figsize=(12,5))
p = sns.lineplot(range(1,15),train_scores,marker='*',label='Train Score')
p = sns.lineplot(range(1,15),test_scores,marker='o',label='Test Score')
```



The best result is captured at k = 11 hence 11 is used for the final model

In [37]:
```
#Setup a knn classifier with k neighbors
knn = KNeighborsClassifier(11)

knn.fit(X_train,y_train)
knn.score(X_test,y_test)
```

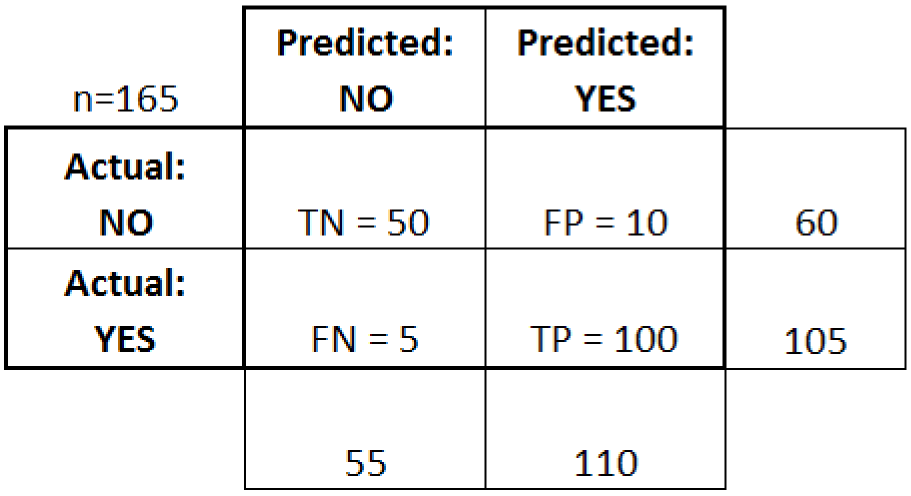
Out[37]:
```
0.765625
```

In [38]:
```
# trying to plot decision boundary
```

Model Performance Analysis

1. Confusion Matrix

The confusion matrix is a technique used for summarizing the performance of a classification algorithm i.e. it has binary outputs.



n=165	Predicted: NO	Predicted: YES	
Actual: NO	TN = 50	FP = 10	60
Actual: YES	FN = 5	TP = 100	105
	55	110	

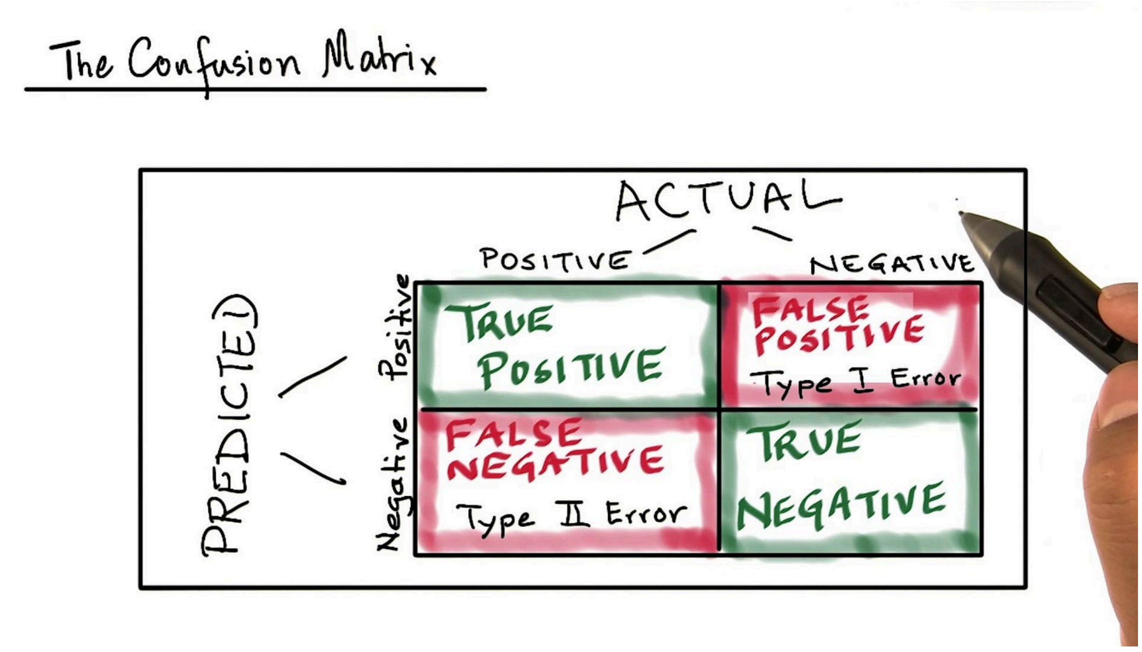
In the famous cancer example:

Cases in which the doctor predicted YES (they have the disease), and they do have the disease will be termed as TRUE POSITIVES (TP). The doctor has correctly predicted that the patient has the disease.

Cases in which the doctor predicted NO (they do not have the disease), and they don't have the disease will be termed as TRUE NEGATIVES (TN). The doctor has correctly predicted that the patient does not have the disease.

Cases in which the doctor predicted YES, and they do not have the disease will be termed as FALSE POSITIVES (FP). Also known as "Type I error".

Cases in which the doctor predicted NO, and they have the disease will be termed as FALSE NEGATIVES (FN). Also known as "Type II error".



For Reference: https://medium.com/@djocz/confusion-matrix-aint-that-confusing-d29e18403327

In [40]:
```
#import confusion_matrix
from sklearn.metrics import confusion_matrix
#let us get the predictions using the classifier we had fit above. Creating the cor
y_pred = knn.predict(X_test)
y_pred
```

Out[40]:
```
array([0, 1, 0, 0, 0, 0, 1, 1, 0, 0, 1, 0, 0, 1, 1, 1, 1, 0, 0, 1, 0, 0,
       0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       1, 0, 0, 1, 1, 0, 0, 0, 0, 0, 1, 1, 0, 1, 0, 1, 0, 1, 0, 0, 0, 0,
       1, 0, 0, 1, 0, 0, 1, 1, 0, 1, 0, 0, 1, 0, 1, 0, 0, 1, 0, 0, 1, 0,
       0, 0, 0, 0, 1, 0, 1, 1, 1, 0, 1, 0, 1, 0, 1, 1, 1, 0, 0, 0, 0, 0,
       0, 1, 0, 0, 0, 0, 1, 0, 0, 1, 1, 1, 1, 0, 0, 0, 1, 0, 0, 0, 1, 0,
       1, 1, 0, 1, 0, 0, 0, 1, 0, 1, 0, 1, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0,
       0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0,
       0, 1, 0, 0, 0, 0, 0, 0, 1, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 1, 0, 0, 0, 0, 1, 0, 1, 1, 1, 1,
       0, 1, 0, 0, 0, 1, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 1, 1, 0,
       0, 0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 1, 0, 0], dtype=int64)
```

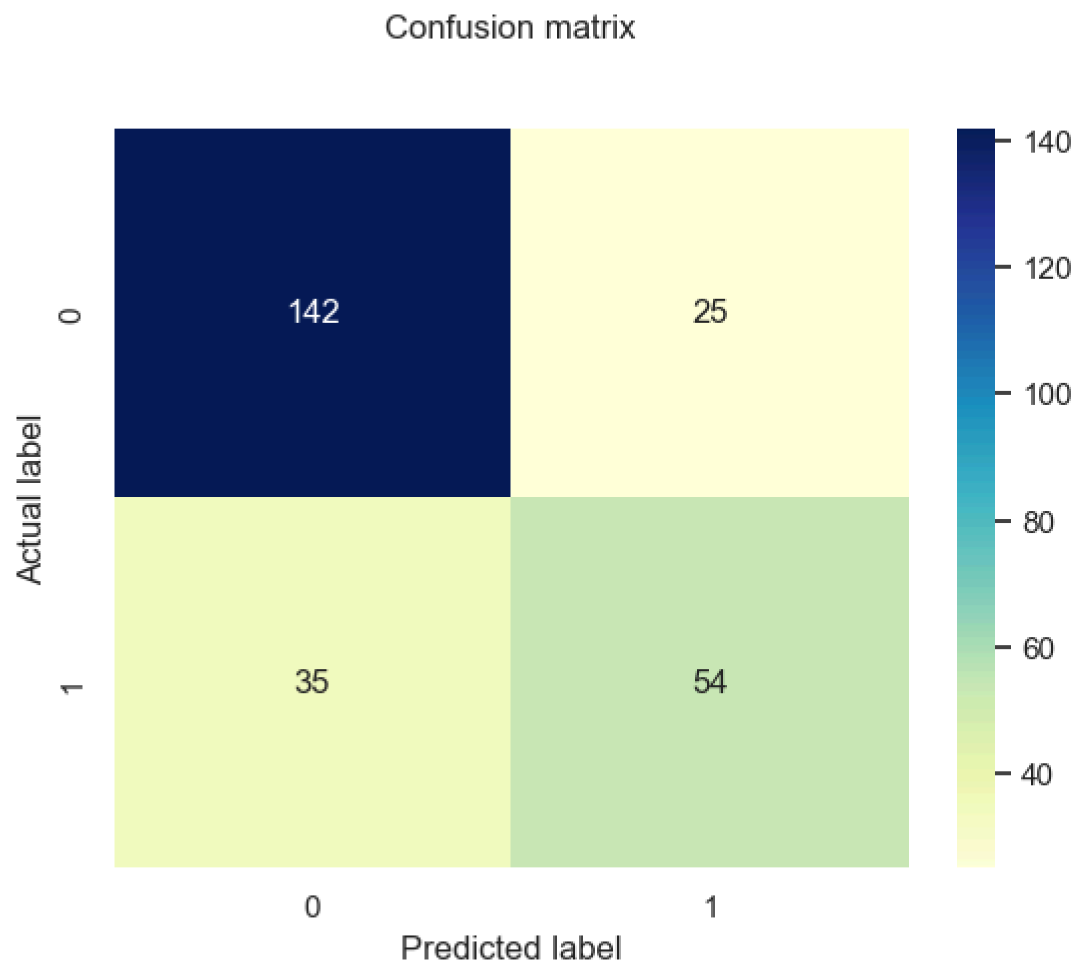
In [41]:
```
confusion_matrix(y_test,y_pred)
pd.crosstab(y_test, y_pred, rownames=['True'], colnames=['Predicted'], margins=True
```

Out[41]:

Predicted	**0**	**1**	**All**
True			
0	142	25	167
1	35	54	89
All	177	79	256

In [42]:
```
# Creating a Heatmap for the confusion matrix.
y_pred = knn.predict(X_test)
from sklearn import metrics
cnf_matrix = metrics.confusion_matrix(y_test, y_pred)
p = sns.heatmap(pd.DataFrame(cnf_matrix), annot=True, cmap="YlGnBu" ,fmt='g')
plt.title('Confusion matrix', y=1.1)
plt.ylabel('Actual label')
plt.xlabel('Predicted label')
```

Out[42]:
```
Text(0.5, 20.049999999999997, 'Predicted label')
```



2. Classification Report

Report which includes Precision, Recall and F1-Score.

Precision Score

```
    TP - True Positives
    FP - False Positives

    Precision - Accuracy of positive predictions.
    Precision = TP/(TP + FP)
```

Recall Score

```
    FN - False Negatives

    Recall(sensitivity or true positive rate): Fraction of
positives that were correctly identified.
    Recall = TP/(TP+FN)
```

F1 Score

```
    F1 Score (aka F-Score or F-Measure) - A helpful metric for
comparing two classifiers.
```

```
    F1 Score takes into account precision and the recall.
    It is created by finding the the harmonic mean of precision and
recall.

    F1 = 2 x (precision x recall)/(precision + recall)
```

> > ***Precision*** - Precision is the ratio of correctly predicted positive observations to the total predicted positive observations. The question that this metric answer is of all passengers that labeled as survived, how many actually survived? High precision relates to the low false positive rate. We have got 0.788 precision which is pretty good.
> >
> > Precision = TP/TP+FP
> >
> > ***Recall (Sensitivity)*** - Recall is the ratio of correctly predicted positive observations to the all observations in actual class - yes. The question recall answers is: Of all the passengers that truly survived, how many did we label? A recall greater than 0.5 is good.
> >
> > Recall = TP/TP+FN
> >
> > ***F1 score*** - F1 Score is the weighted average of Precision and Recall. Therefore, this score takes both false positives and false negatives into account. Intuitively it is not as easy to understand as accuracy, but F1 is usually more useful than accuracy, especially if you have an uneven class distribution. Accuracy works best if false positives and false negatives have similar cost. If the cost of false positives and false negatives are very different, it's better to look at both Precision and Recall.
> >
> > F1 Score = 2(*Recall* Precision) / (Recall + Precision)

For Reference: http://joshlawman.com/metrics-classification-report-breakdown-precision-recall-f1/ : https://blog.exsilio.com/all/accuracy-precision-recall-f1-score-interpretation-of-performance-measures/

In [43]:
```
#import classification_report
from sklearn.metrics import classification_report
print(classification_report(y_test,y_pred))
```

```
              precision    recall  f1-score   support

           0       0.80      0.85      0.83       167
           1       0.68      0.61      0.64        89

    accuracy                           0.77       256
   macro avg       0.74      0.73      0.73       256
weighted avg       0.76      0.77      0.76       256
```

3. ROC - AUC

ROC (Receiver Operating Characteristic) Curve tells us about how good the model can distinguish between two things (e.g If a patient has a disease or no). Better models can accurately distinguish between the two. Whereas, a poor model will have difficulties in distinguishing between the two

Well Explained in this video: https://www.youtube.com/watch?v=OAl6eAyP-yo

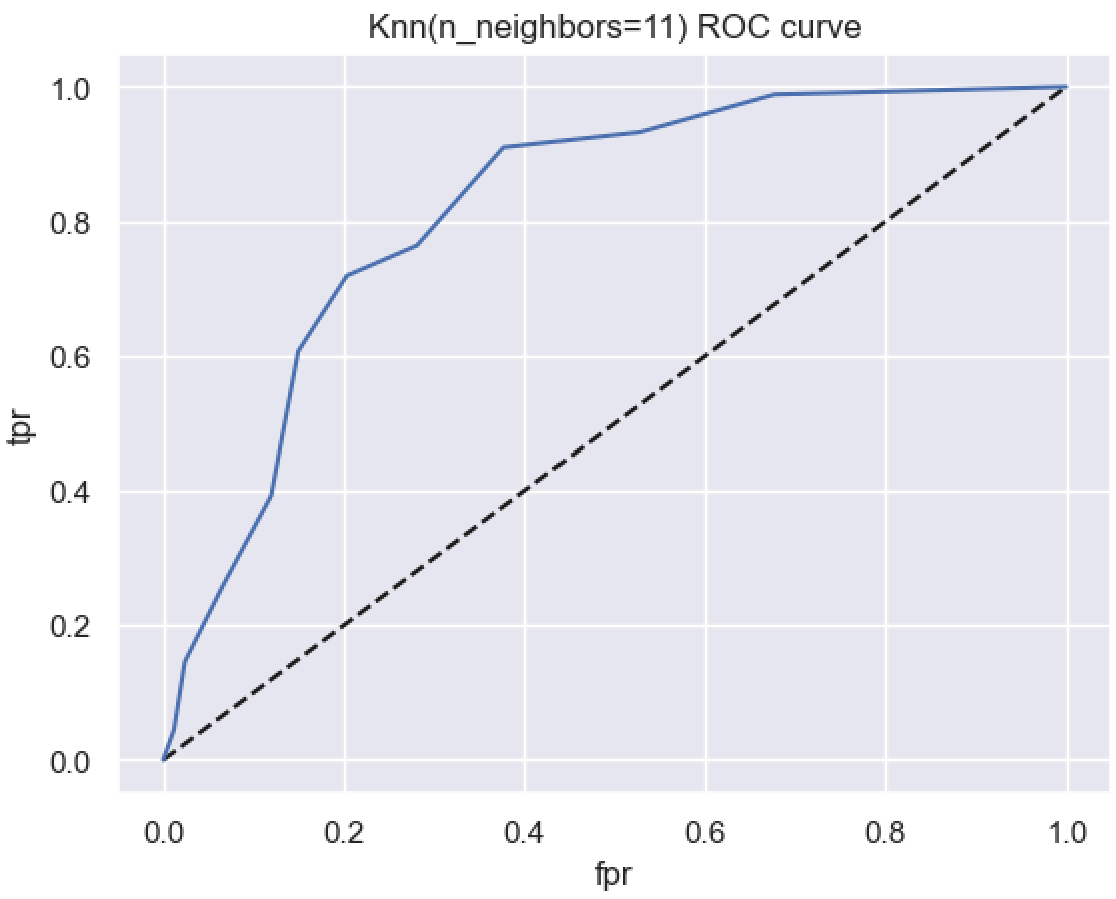
In [44]:
```
from sklearn.metrics import roc_curve
y_pred_proba = knn.predict_proba(X_test)[:,1]
y_pred_proba
```

Out[44]:
```
array([0.        , 0.72727273, 0.36363636, 0.09090909, 0.45454545,
       0.27272727, 0.72727273, 0.90909091, 0.        , 0.18181818,
       0.54545455, 0.45454545, 0.45454545, 0.90909091, 0.63636364,
       0.72727273, 0.54545455, 0.        , 0.27272727, 0.72727273,
       0.09090909, 0.09090909, 0.18181818, 0.36363636, 0.09090909,
       0.27272727, 0.63636364, 0.27272727, 0.        , 0.09090909,
       0.45454545, 0.        , 0.27272727, 0.        , 0.        ,
       0.36363636, 0.18181818, 0.        , 0.        , 0.        ,
       0.45454545, 0.18181818, 0.27272727, 0.        , 0.90909091,
       0.18181818, 0.27272727, 0.63636364, 0.63636364, 0.        ,
       0.45454545, 0.        , 0.09090909, 0.        , 0.63636364,
       0.63636364, 0.        , 0.72727273, 0.36363636, 0.63636364,
       0.09090909, 0.81818182, 0.09090909, 0.09090909, 0.        ,
       0.        , 0.54545455, 0.45454545, 0.45454545, 0.63636364,
       0.27272727, 0.27272727, 0.54545455, 0.90909091, 0.18181818,
       0.54545455, 0.36363636, 0.27272727, 0.54545455, 0.09090909,
       0.54545455, 0.27272727, 0.18181818, 0.72727273, 0.27272727,
       0.27272727, 0.54545455, 0.45454545, 0.09090909, 0.18181818,
       0.18181818, 0.18181818, 0.72727273, 0.09090909, 0.54545455,
       0.54545455, 0.54545455, 0.        , 0.81818182, 0.        ,
       0.54545455, 0.09090909, 0.54545455, 0.81818182, 0.63636364,
       0.45454545, 0.09090909, 0.        , 0.        , 0.18181818,
       0.45454545, 0.54545455, 0.        , 0.        , 0.        ,
       0.27272727, 0.72727273, 0.        , 0.27272727, 0.54545455,
       0.63636364, 0.81818182, 0.81818182, 0.45454545, 0.27272727,
       0.27272727, 0.81818182, 0.09090909, 0.36363636, 0.36363636,
       0.72727273, 0.        , 0.54545455, 0.81818182, 0.18181818,
       0.54545455, 0.45454545, 0.        , 0.27272727, 0.90909091,
       0.        , 0.63636364, 0.        , 0.54545455, 0.09090909,
       0.81818182, 0.        , 0.        , 0.09090909, 0.72727273,
       0.        , 0.18181818, 0.09090909, 0.18181818, 0.        ,
       0.27272727, 0.27272727, 0.54545455, 0.09090909, 0.36363636,
       0.09090909, 0.63636364, 0.18181818, 0.18181818, 0.45454545,
       0.63636364, 0.        , 0.18181818, 0.18181818, 0.09090909,
       0.27272727, 0.        , 0.        , 0.72727273, 0.90909091,
       0.09090909, 0.18181818, 0.63636364, 0.        , 0.09090909,
       0.        , 0.27272727, 0.        , 0.36363636, 0.63636364,
       0.        , 0.63636364, 0.72727273, 0.        , 0.09090909,
       0.27272727, 0.36363636, 0.        , 0.        , 0.        ,
       0.09090909, 0.09090909, 0.36363636, 0.18181818, 0.45454545,
       0.18181818, 0.27272727, 0.45454545, 0.36363636, 0.63636364,
       0.18181818, 0.        , 0.54545455, 0.36363636, 0.72727273,
       0.        , 0.09090909, 0.        , 0.18181818, 0.54545455,
       0.45454545, 0.72727273, 0.81818182, 0.81818182, 0.54545455,
       0.27272727, 0.63636364, 0.36363636, 0.        , 0.36363636,
       0.63636364, 0.81818182, 0.        , 0.18181818, 0.27272727,
       0.45454545, 0.63636364, 0.09090909, 0.        , 0.        ,
       0.27272727, 0.63636364, 0.18181818, 0.45454545, 0.72727273,
       0.72727273, 0.09090909, 0.09090909, 0.36363636, 0.        ,
       0.27272727, 0.27272727, 0.18181818, 0.54545455, 0.        ,
       0.72727273, 0.18181818, 0.36363636, 0.54545455, 0.        ,
       0.        ])
```

In [45]:
```python
fpr, tpr, thresholds = roc_curve(y_test, y_pred_proba)
print('FPR')
print(fpr)
print('TPR')
print(tpr)
print('Thresholds')
print(thresholds)
```

```
FPR
[0.         0.01197605 0.0239521  0.06586826 0.11976048 0.1497006
 0.20359281 0.28143713 0.37724551 0.52694611 0.67664671 1.        ]
TPR
[0.         0.04494382 0.14606742 0.25842697 0.39325843 0.60674157
 0.71910112 0.76404494 0.91011236 0.93258427 0.98876404 1.        ]
Thresholds
[1.90909091 0.90909091 0.81818182 0.72727273 0.63636364 0.54545455
 0.45454545 0.36363636 0.27272727 0.18181818 0.09090909 0.        ]
```

In [46]:
```python
# Plotting the ROC Curve
plt.plot([0,1],[0,1],'k--')
plt.plot(fpr,tpr, label='Knn')
plt.xlabel('fpr')
plt.ylabel('tpr')
plt.title('Knn(n_neighbors=11) ROC curve')
plt.show()
```



In [47]:
```python
#Area under ROC curve
from sklearn.metrics import roc_auc_score
roc_auc_score(y_test,y_pred_proba)
```

Out[47]:
```
0.8193500639171096
```

Hyper Parameter optimization

Grid search is an approach to hyperparameter tuning that will methodically build and evaluate a model for each combination of algorithm parameters specified in a grid.

Let's consider the following example:

Suppose, a machine learning model X takes hyperparameters a1, a2 and a3. In grid searching, you first define the range of values for each of the hyperparameters a1, a2 and a3. You can think of this as an array of values for each of the hyperparameters. Now the grid search technique will construct many versions of X with all the possible combinations of hyperparameter (a1, a2 and a3) values that you defined in the first place. This range of hyperparameter values is referred to as the grid.

Suppose, you defined the grid as: a1 = [0,1,2,3,4,5] a2 = [10,20,30,40,5,60] a3 = [105,105,110,115,120,125]

Note that, the array of values of that you are defining for the hyperparameters has to be legitimate in a sense that you cannot supply Floating type values to the array if the hyperparameter only takes Integer values.

Now, grid search will begin its process of constructing several versions of X with the grid that you just defined.

It will start with the combination of [0,10,105], and it will end with [5,60,125]. It will go through all the intermediate combinations between these two which makes grid search computationally very expensive.

In [48]:
```
#import GridSearchCV
from sklearn.model_selection import GridSearchCV
#In case of classifier like knn the parameter to be tuned is n_neighbors
param_grid = {'n_neighbors':np.arange(1,50)}
knn = KNeighborsClassifier()
knn_cv= GridSearchCV(knn,param_grid,cv=5)
knn_cv.fit(X,y)

print("Best Score:" + str(knn_cv.best_score_))
print("Best Parameters: " + str(knn_cv.best_params_))
```

```
Best Score:0.7721840251252015
Best Parameters: {'n_neighbors': 25}
```