

# Overview

#we will be predicting that whether the patient has diabetes or not on the basis of the features #we will provide to our machine learning model, and for that, we will be using the famous Pima Indians Diabetes Database.

## Diabetes Prediction Using Machine Learning

Data analysis:

Here one will get to know about how the data analysis part is done in a data science life cycle.

Exploratory data analysis:

EDA is one of the most important steps in the data science project life cycle and here one will need to know that how to make inferences from the visualizations and data analysis

Model building:

Here we will be using 4 ML models and then we will choose the best performing model.

Saving model:

Saving the best model using pickle to make the prediction from real data.

```
In [2]: # Importing Libraries

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

sns.set()

from mlxtend.plotting import plot_decision_regions
import missingno as msno
from pandas.plotting import scatter_matrix
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split
from sklearn.neighbors import KNeighborsClassifier

from sklearn.metrics import confusion_matrix
from sklearn import metrics
from sklearn.metrics import classification_report
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score
import warnings
warnings.filterwarnings('ignore')
%matplotlib inline
```

```
In [3]: #Here we will be reading the dataset which is in the CSV format
```

```
diabetes_df = pd.read_csv('diabetes.csv')
diabetes_df.head()
```

Out[3]:

	Pregnancies	Glucose	BloodPressure	SkinThickness	Insulin	BMI	DiabetesPedigreeFunction	Outcome
0	6	148	72	35	0	33.6	0.625	1
1	1	85	66	29	0	26.6	0.349	0
2	8	183	64	0	0	23.3	0.672	1
3	1	89	66	23	94	28.1	0.167	0
4	0	137	40	35	168	43.1	2.278	1

## Exploratory Data Analysis (EDA)

```
In [4]: diabetes_df.columns
```

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

In [5]: *#Information about the dataset*

```
diabetes_df.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 [6]: *#To know more about the dataset*

```
diabetes_df.describe()
```

Out[6]:

	Pregnancies	Glucose	BloodPressure	SkinThickness	Insulin	BMI	Diabetes
count	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	0.332422
std	3.369578	31.972618	19.355807	15.952218	115.244002	7.884160	0.471498
min	0.000000	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	0.000000
50%	3.000000	117.000000	72.000000	23.000000	30.500000	32.000000	0.000000
75%	6.000000	140.250000	80.000000	32.000000	127.250000	36.600000	0.000000
max	17.000000	199.000000	122.000000	99.000000	846.000000	67.100000	1.000000

```
In [7]: #To know more about the dataset with transpose - here T is for the transpose
diabetes_df.describe().T
```

```
Out[7]:
```

	count	mean	std	min	25%	50%	75%
<b>Pregnancies</b>	768.0	3.845052	3.369578	0.000	1.00000	3.0000	6.0000
<b>Glucose</b>	768.0	120.894531	31.972618	0.000	99.00000	117.0000	140.2500
<b>BloodPressure</b>	768.0	69.105469	19.355807	0.000	62.00000	72.0000	80.0000
<b>SkinThickness</b>	768.0	20.536458	15.952218	0.000	0.00000	23.0000	32.0000
<b>Insulin</b>	768.0	79.799479	115.244002	0.000	0.00000	30.5000	127.2500
<b>BMI</b>	768.0	31.992578	7.884160	0.000	27.30000	32.0000	36.6000
<b>DiabetesPedigreeFunction</b>	768.0	0.471876	0.331329	0.078	0.24375	0.3725	0.6262
<b>Age</b>	768.0	33.240885	11.760232	21.000	24.00000	29.0000	41.0000
<b>Outcome</b>	768.0	0.348958	0.476951	0.000	0.00000	0.0000	1.0000

```
In [8]: #Now Let's check that if our dataset have null values or not
diabetes_df.isnull().head(10)
```

```
Out[8]:
```

	Pregnancies	Glucose	BloodPressure	SkinThickness	Insulin	BMI	DiabetesPedigreeFunc
0	False	False	False	False	False	False	F
1	False	False	False	False	False	False	F
2	False	False	False	False	False	False	F
3	False	False	False	False	False	False	F
4	False	False	False	False	False	False	F
5	False	False	False	False	False	False	F
6	False	False	False	False	False	False	F
7	False	False	False	False	False	False	F
8	False	False	False	False	False	False	F
9	False	False	False	False	False	False	F

```
In [9]: #Now Let's check the number of null values our dataset has.
diabetes_df.isnull().sum()
```

```
Out[9]: Pregnancies      0
Glucose      0
BloodPressure  0
SkinThickness  0
Insulin      0
BMI          0
DiabetesPedigreeFunction  0
Age          0
Outcome      0
dtype: int64
```

#Here from the above code we first checked that is there any null values from the `IsNull()` function then we are going to take the sum of all those missing values from the `sum()` function and the inference we now get is that there are no missing values but that is actually not a true story as in this particular dataset all the missing values were given the 0 as a value which is not good for the authenticity of the dataset. Hence we will first replace the 0 value with the NAN value then start the imputation process.

```
In [10]: diabetes_df_copy = diabetes_df.copy(deep = True)
diabetes_df_copy[['Glucose', 'BloodPressure', 'SkinThickness', 'Insulin', 'BMI']]
```

```
In [11]: # Showing the Count of NANs
print(diabetes_df_copy.isnull().sum())
```

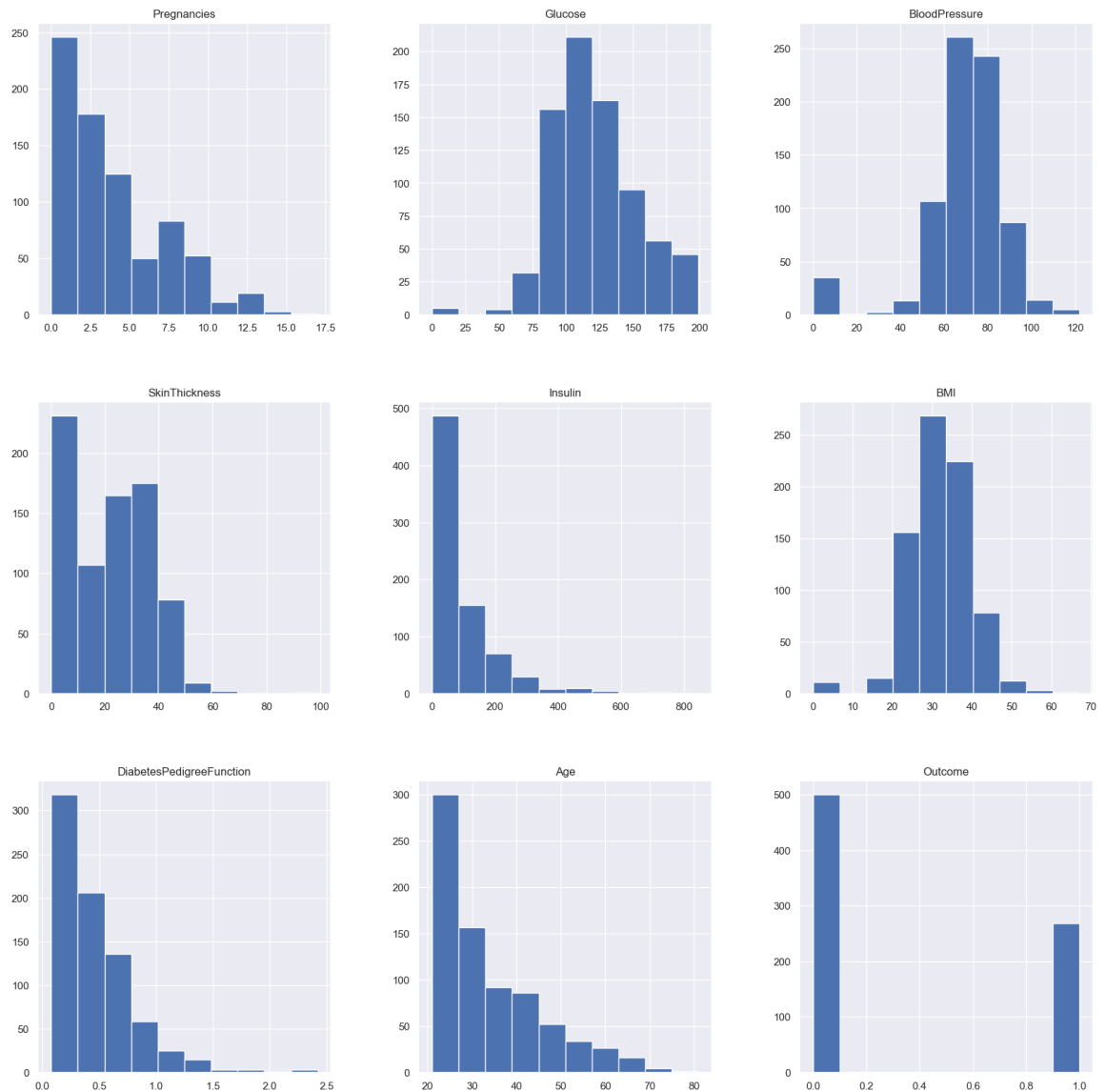
Pregnancies	0
Glucose	5
BloodPressure	35
SkinThickness	227
Insulin	374
BMI	11
DiabetesPedigreeFunction	0
Age	0
Outcome	0
dtype: int64	

#As mentioned above that now we will be replacing the zeros with the NAN values so that we can impute it later to maintain the authenticity of the dataset as well as trying to have a better Imputation approach i.e to apply mean values of each column to the null values of the respective columns.

## Data Visualization

In [12]: *#Plotting the data distribution plots before removing null values*

```
p = diabetes_df.hist(figsize=(20,20))
```



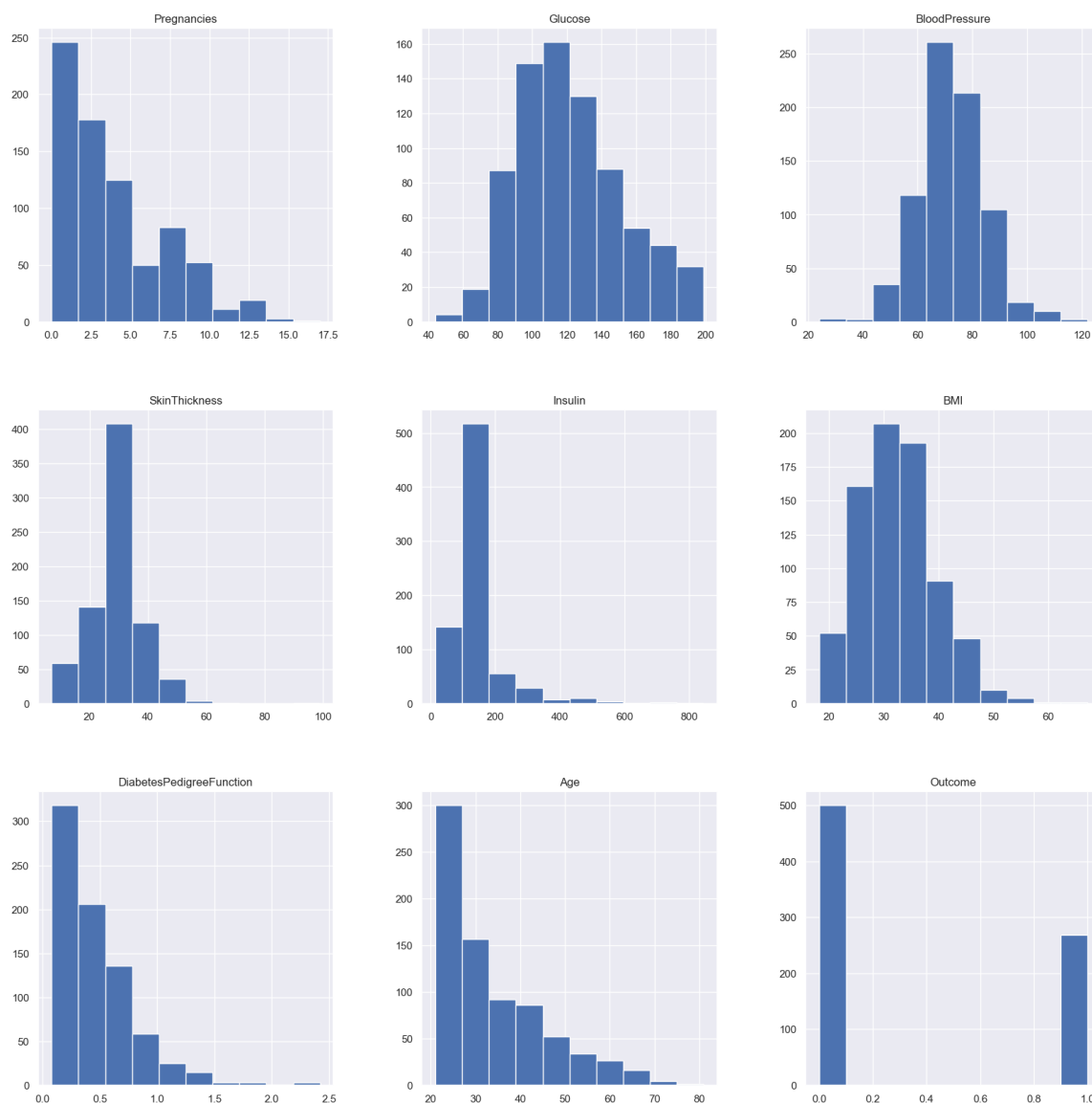
**#Inference:** So here we have seen the distribution of each features whether it is dependent data or independent data and one thing which could always strike that why do we need to see the distribution of data? So the answer is simple it is the best way to start the analysis of the dataset as it shows the occurrence of every kind of value in the graphical structure which in turn lets us know the range of the data.

In [13]: *#Now we will be imputing the mean value of the column to each missing value*

```
diabetes_df_copy['Glucose'].fillna(diabetes_df_copy['Glucose'].mean(), inplace=True)
diabetes_df_copy['BloodPressure'].fillna(diabetes_df_copy['BloodPressure'].mean(), inplace=True)
diabetes_df_copy['SkinThickness'].fillna(diabetes_df_copy['SkinThickness'].median(), inplace=True)
diabetes_df_copy['Insulin'].fillna(diabetes_df_copy['Insulin'].median(), inplace=True)
diabetes_df_copy['BMI'].fillna(diabetes_df_copy['BMI'].median(), inplace=True)
```

In [14]: *#Plotting the distributions after removing the NAN values.*

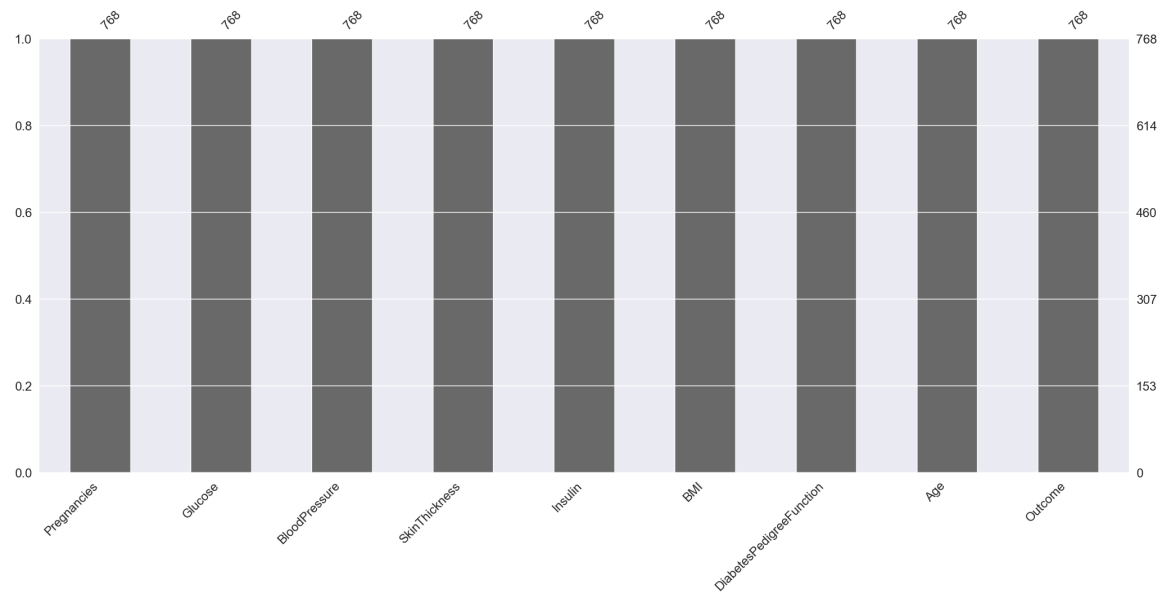
```
p = diabetes_df_copy.hist(figsize = (20,20))
```



Inference: Here we are again using the hist plot to see the distribution of the dataset but this time we are using this visualization to see the changes that we can see after those null values are removed from the dataset and we can clearly see the difference for example – In age column after removal of the null values, we can see that there is a spike at the range of 50 to 100 which is quite logical as well.

In [15]: *#Plotting Null Count Analysis Plot*

```
p = msno.bar(diabetes_df)
```



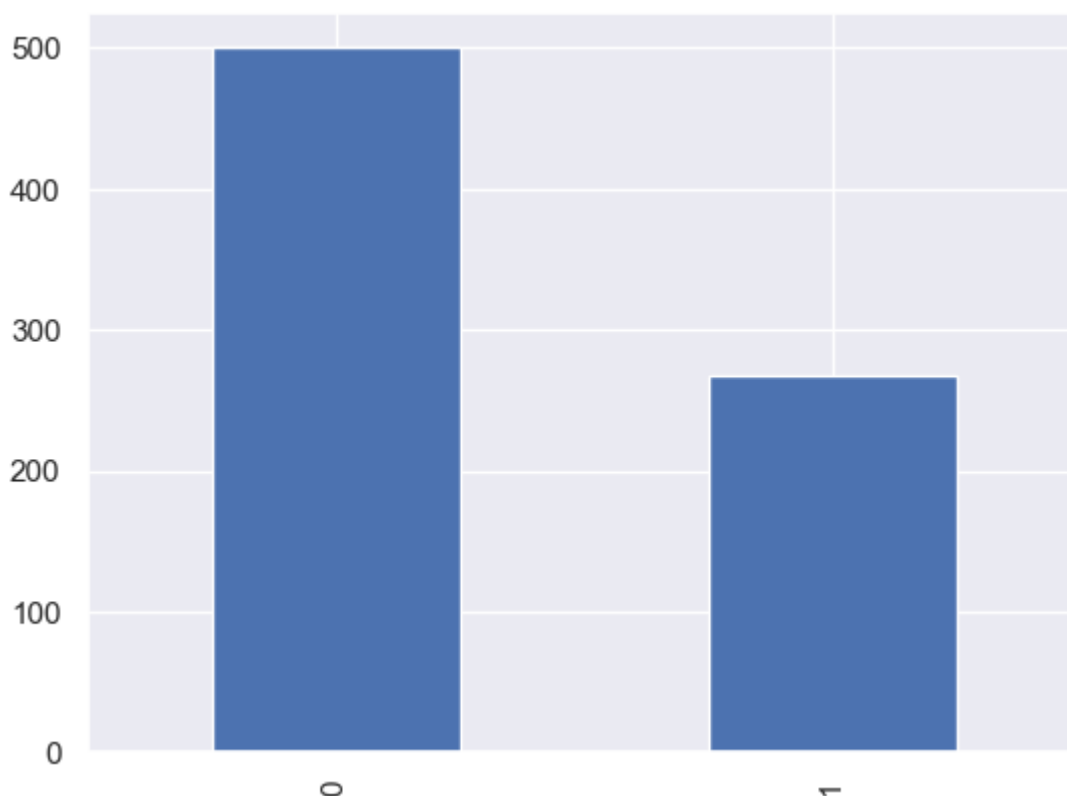
**#Inference:** Now in the above graph also we can clearly see that there are no null values in the dataset.



In [16]: *#Now, let's check that how well our outcome column is balanced*

```
color_wheel = {1: "#0392cf", 2: "#7bc043"}
colors = diabetes_df["Outcome"].map(lambda x: color_wheel.get(x+1))
print(diabetes_df.Outcome.value_counts())
p = diabetes_df.Outcome.value_counts().plot(kind="bar")
```

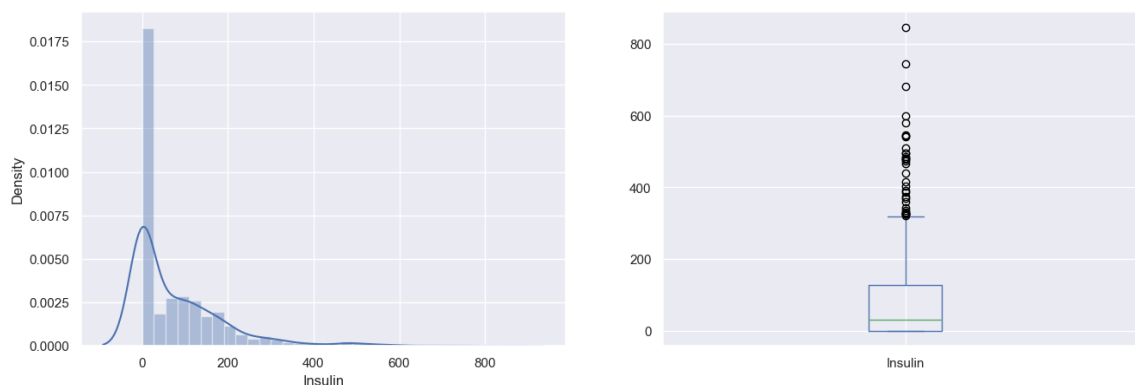
```
0    500
1    268
Name: Outcome, dtype: int64
```



#Here from the above visualization it is clearly visible that our dataset is completely imbalanced in fact the number of patients who are diabetic is half of the patients who are non-diabetic.

In [17]: 

```
plt.subplot(121), sns.distplot(diabetes_df['Insulin'])
plt.subplot(122), diabetes_df['Insulin'].plot.box(figsize=(16,5))
plt.show()
```

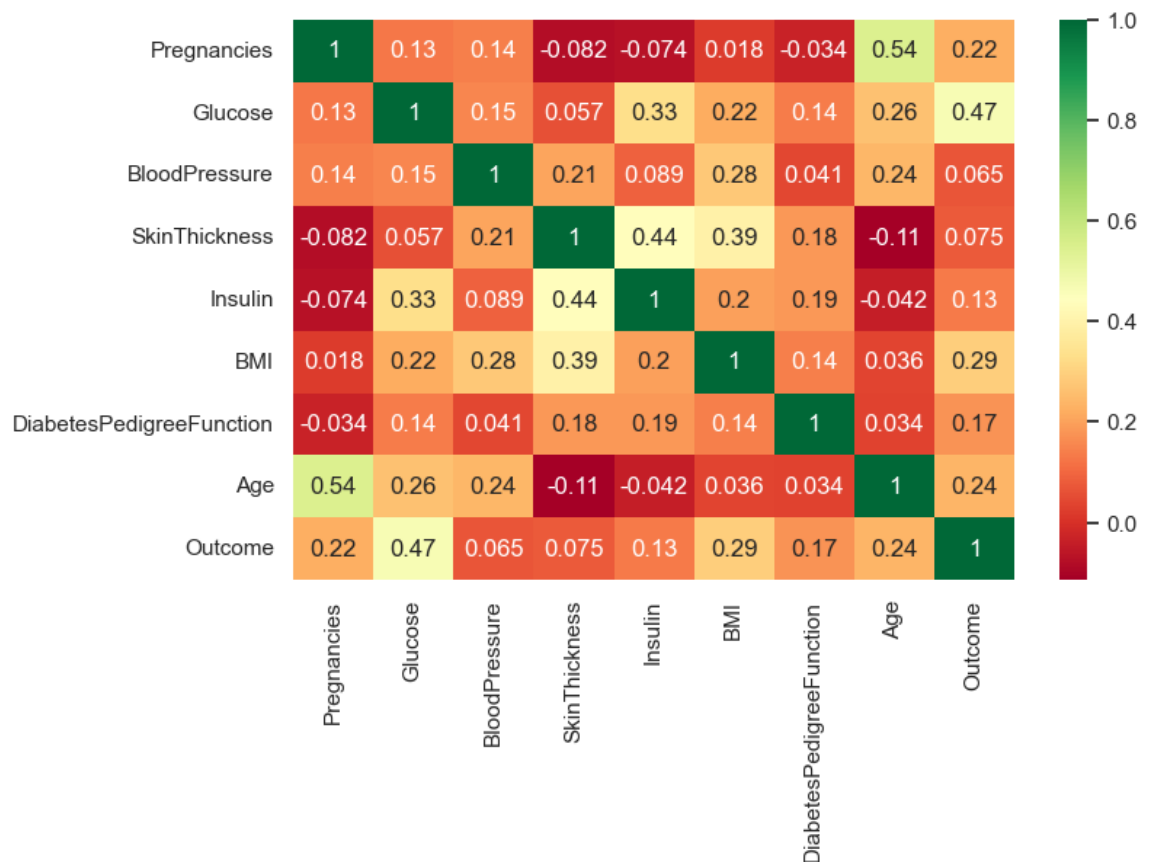


#That's how Distplot can be helpful where one will able to see the distribution of the data as well as with the help of boxplot one can see the outliers in that column and other information too which can be derived by the box and whiskers plot.

## Correlation between all the features

In [18]: *#Correlation between all the features before cleaning*

```
plt.figure(figsize=(8,5))  
# seaborn has an easy method to showcase heatmap  
p = sns.heatmap(diabetes_df.corr(), annot=True, cmap = 'RdYlGn')
```



## Scaling the Data

In [19]: *#Before scaling down the data let's have a look into it*

```
diabetes_df_copy.head()
```

Out[19]:

	Pregnancies	Glucose	BloodPressure	SkinThickness	Insulin	BMI	DiabetesPedigreeFunction
0	6	148.0	72.0	35.0	125.0	33.6	0.6
1	1	85.0	66.0	29.0	125.0	26.6	0.3
2	8	183.0	64.0	29.0	125.0	23.3	0.6
3	1	89.0	66.0	23.0	94.0	28.1	0.1
4	0	137.0	40.0	35.0	168.0	43.1	2.2

In [20]: *#After Standard scaling*

```
sc_X = StandardScaler()  
X = pd.DataFrame(sc_X.fit_transform(diabetes_df_copy.drop(["Outcome"],axis  
'Glucose', 'BloodPressure', 'SkinThickness', 'Insulin', 'BMI', 'DiabetesPedi  
X.head()
```

Out[20]:

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

#That's how our dataset will be looking like when it is scaled down or we can see every value now is on the same scale which will help our ML model to give a better result.

## Model Building

In [21]: *#Splitting the dataset*

```
X = diabetes_df.drop('Outcome', axis = 1)  
y = diabetes_df['Outcome']
```

In [22]: *#Now we will split the data into training and testing data using the train\_t*

```
from sklearn.model_selection import train_test_split  
X_train, X_test, y_train, y_test = train_test_split(X,y,test_size=0.33,randc
```

## Random Forest

In [23]: *#Building the model using RandomForest*

```
from sklearn.ensemble import RandomForestClassifier

rfc = RandomForestClassifier(n_estimators=200)
rfc.fit(X_train,y_train)
```

Out[23]:

```
▼      RandomForestClassifier
RandomForestClassifier(n_estimators=200)
```

In [24]: *#Now after building the model let's check the accuracy of the model on the t*

```
rfc_train = rfc.predict(X_train)
from sklearn import metrics

print("Accuracy_Score = ", format(metrics.accuracy_score(y_train, rfc_train)
```

Accuracy\_Score = 1.0

#So here we can see that on the training dataset our model is overfitted. Getting the accuracy score for Random Forest

In [25]: **from** sklearn **import** metrics

```
predictions = rfc.predict(X_test)
print("Acciuracy_Score = ", format(metrics.accuracy_score(y_test,predictions
```

Acciuracy\_Score = 0.7795275590551181

## Classification Report and Confusion matrix of random forest model

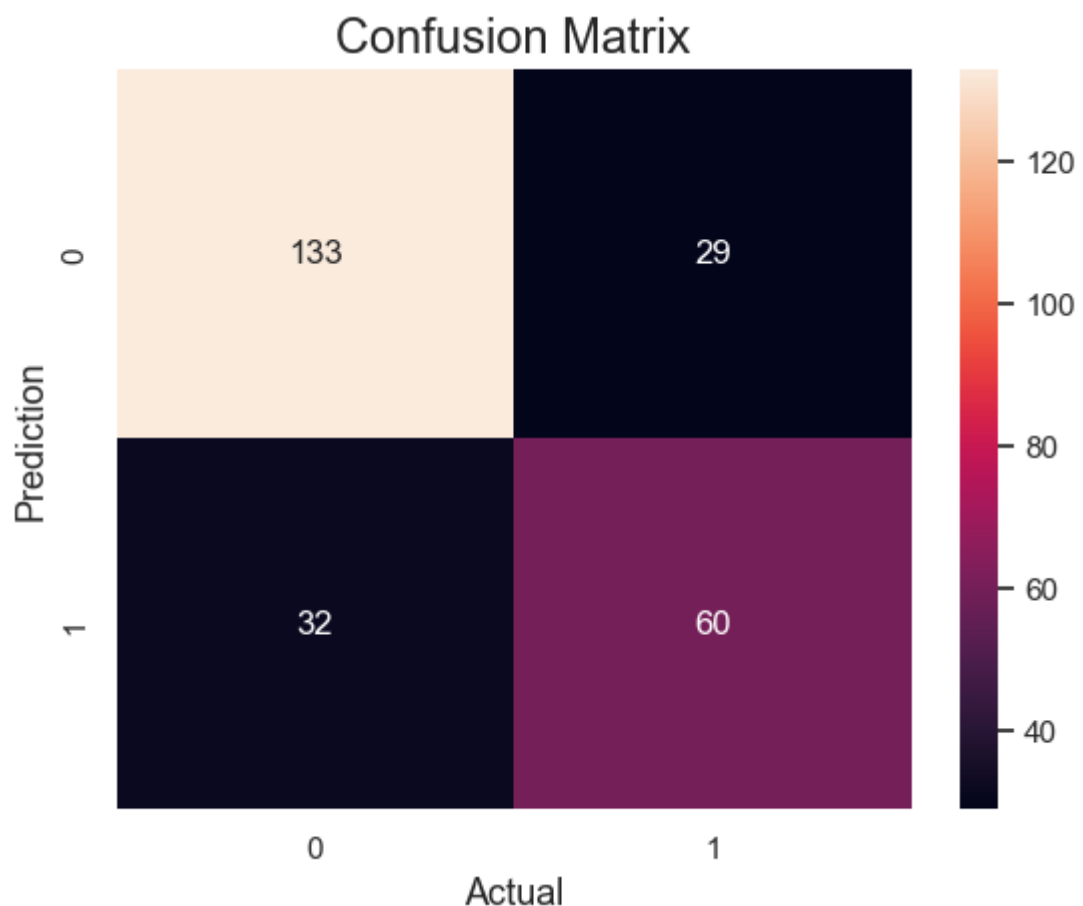
```
In [26]: # Train the model
clf = RandomForestClassifier(random_state=23)
clf.fit(X_train, y_train)

# prediction
y_pred = clf.predict(X_test)

#compute the confusion matrix
cm = confusion_matrix(y_test, y_pred)

#Plot the confusion matrix
sns.heatmap(cm,
            annot = True,
            fmt = 'g')
plt.ylabel('Prediction', fontsize = 13)
plt.xlabel('Actual', fontsize = 13)
plt.title('Confusion Matrix', fontsize=17)
plt.show()

# Finding precision and recall
accuracy = accuracy_score(y_test, y_pred)
print("Accuracy :", accuracy)
```



Accuracy : 0.7598425196850394

```
In [27]: # Finding precision and recall
accuracy = accuracy_score(y_test, y_pred)
print("Accuracy :", accuracy)
precision = precision_score(y_test, y_pred)
print("Precision :", precision)
recall = recall_score(y_test, y_pred)
print("Recall :", recall)
F1_score = f1_score(y_test, y_pred)
print("F1-score :", F1_score)
```

```
Accuracy : 0.7598425196850394
Precision : 0.6741573033707865
Recall : 0.6521739130434783
F1-score : 0.6629834254143646
```

## Decision Tree

```
In [28]: #Building the model using DecisionTree
from sklearn.tree import DecisionTreeClassifier

dtree = DecisionTreeClassifier()
dtree.fit(X_train, y_train)

#Now we will be making the predictions on the testing data directly as it is
```

```
Out[28]: ▾ DecisionTreeClassifier
DecisionTreeClassifier()
```

```
In [29]: #Getting the accuracy score for Decision Tree

from sklearn import metrics

predictions = dtree.predict(X_test)
print("Accuracy Score =", format(metrics.accuracy_score(y_test, predictions)))

Accuracy Score = 0.7086614173228346
```

```
In [30]: #Classification report and confusion matrix of the decision tree model

from sklearn.metrics import classification_report, confusion_matrix

print(confusion_matrix(y_test, predictions))
print(classification_report(y_test, predictions))
```

```
[[127  35]
 [ 39  53]]
```

	precision	recall	f1-score	support
0	0.77	0.78	0.77	162
1	0.60	0.58	0.59	92
accuracy			0.71	254
macro avg	0.68	0.68	0.68	254
weighted avg	0.71	0.71	0.71	254

# XgBoost classifier

In [31]: *#Building model using XGBoost*

```
from xgboost import XGBClassifier

xgb_model = XGBClassifier(gamma=0)
xgb_model.fit(X_train, y_train)
```

Out[31]:

▼ XGBClassifier

XGBClassifier(base\_score=None, booster=None, callbacks=None,  
colsample\_bylevel=None, colsample\_bynode=None,  
colsample\_bytree=None, device=None, early\_stopping\_rounds=  
None,  
enable\_categorical=False, eval\_metric=None, feature\_types=  
None,  
gamma=0, grow\_policy=None, importance\_type=None,  
interaction\_constraints=None, learning\_rate=None, max\_bin=  
None,  
max\_cat\_threshold=None, max\_cat\_to\_onehot=None,

In [32]: *#Now we will be making the predictions on the testing data directly as it is*

In [33]: *#Getting the accuracy score for the XgBoost classifier*

```
from sklearn import metrics

xgb_pred = xgb_model.predict(X_test)
print("Accuracy Score =", format(metrics.accuracy_score(y_test, xgb_pred)))

Accuracy Score = 0.7283464566929134
```

In [34]: *#Classification report and confusion matrix of the XgBoost classifier*

```
In [35]: #Support Vector Machine (SVM)

#Building the model using Support Vector Machine (SVM)

from sklearn.svm import SVC

svc_model = SVC()
svc_model.fit(X_train, y_train)

#Prediction from support vector machine model on the testing data

svc_pred = svc_model.predict(X_test)

#Accuracy score for SVM

from sklearn import metrics

print("Accuracy Score =", format(metrics.accuracy_score(y_test, svc_pred)))
```

Accuracy Score = 0.7480314960629921

```
In [36]: # Classification report and confusion matrix of the SVM classifier

from sklearn.metrics import classification_report, confusion_matrix

print(confusion_matrix(y_test,svc_pred))
print(classification_report(y_test,svc_pred))
```

```
[[145  17]
 [ 47  45]]
```

		precision	recall	f1-score	support
	0	0.76	0.90	0.82	162
	1	0.73	0.49	0.58	92
	accuracy			0.75	254
	macro avg	0.74	0.69	0.70	254
	weighted avg	0.74	0.75	0.73	254

## The Conclusion from Model Building

Therefore Random forest is the best model for this prediction since it has an accuracy\_score of 0.76.

## Feature Importance

Knowing about the feature importance is quite necessary as it shows that how much weightage each feature provides in the model building phase.



```
In [37]: #Getting feature importances
```

```
rfc.feature_importances_
```

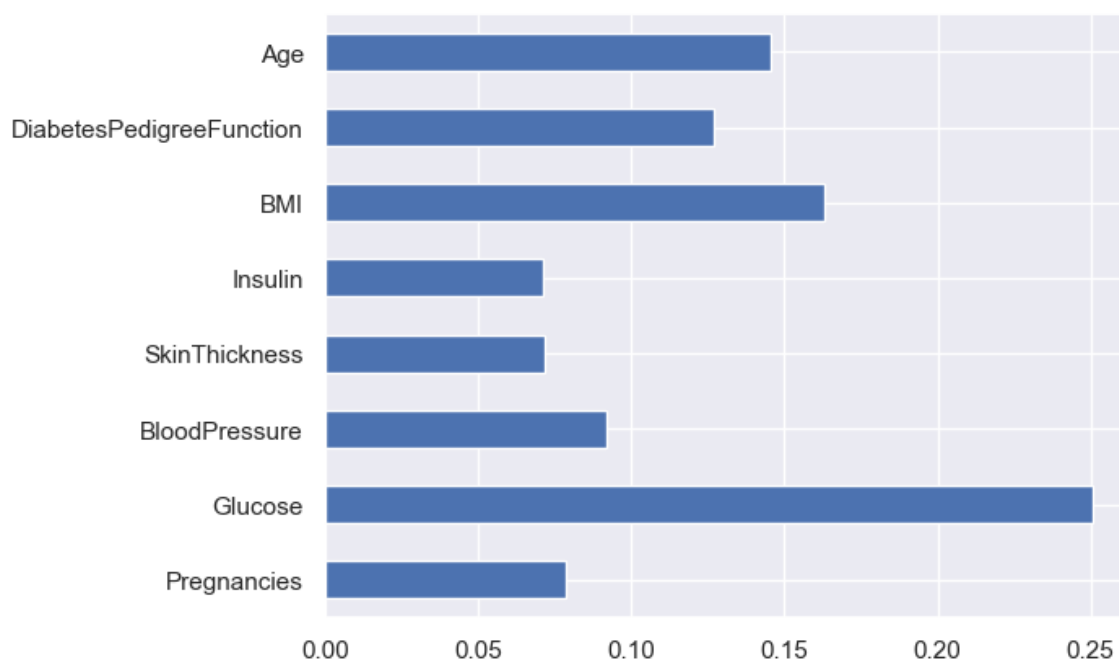
```
Out[37]: array([0.07878467, 0.25095031, 0.09179752, 0.07155992, 0.07123608,  
        0.16308592, 0.12719035, 0.14539523])
```

#From the above output, it is not much clear that which feature is important for that reason we will now make a visualization of the same.

```
In [38]: #Plotting feature importances
```

```
(pd.Series(rfc.feature_importances_, index=X.columns).plot(kind='barh'))
```

```
Out[38]: <Axes: >
```



#Here from the above graph, it is clearly visible that Glucose as a feature is the most important in this dataset.

## Saving Model – Random Forest

```
In [40]: import pickle

# Firstly we will be using the dump() function to save the model using pickle
saved_model = pickle.dumps(rfc)

# Then we will be loading that saved model
rfc_from_pickle = pickle.loads(saved_model)

# Lastly, after loading that model we will use this to make predictions
rfc_from_pickle.predict(X_test)
```

```
# Then we will be loading that saved model
rfc_from_pickle = pickle.loads(saved_model)
```

```
# Lastly, after loading that model we will use this to make predictions
rfc from pickle.predict(X test)
```

#Now for the last time, I'll be looking at the head and tail of the dataset so that we can take any random set of features from both the head and tail of the data to test that if our model is good enough to give the right prediction.

```
In [41]: diabetes_df.head()
```

	Pregnancies	Glucose	BloodPressure	SkinThickness	Insulin	BMI	DiabetesPedigreeFunction
0	6	148	72	35	0	33.6	0.625
1	1	85	66	29	0	26.6	0.349
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	2.278

```
In [42]: diabetes_df.tail()
```

	Pregnancies	Glucose	BloodPressure	SkinThickness	Insulin	BMI	DiabetesPedigreeFun
763	10	101	76	48	180	32.9	
764	2	122	70	27	0	36.8	
765	5	121	72	23	112	26.2	
766	1	126	60	0	0	30.1	
767	1	93	70	31	0	30.4	

```
In [43]: #Putting data points in the model will either return 0 or 1 i.e. person suff
```

```
rfc.predict([[0,137,40,35,168,43.1,2.228,33]]) #4th patient
```

```
Out[43]: array([1], dtype=int64)
```

```
In [44]: #Another one
```

```
rfc.predict([[10,101,76,48,180,32.9,0.171,63]]) # 763 th patient
```

```
Out[44]: array([0], dtype=int64)
```

## Conclusion

After using all these patient records, we are able to build a machine learning model (random forest – best one) to accurately

predict whether or not the patients in the dataset have diabetes or not along with that we were able to draw some insights from

the data via data analysis and visualization.

```
In [ ]:
```