Stroke Healthcare Dataset

In this project, stroke disease dataset was loaded and some basic steps were performed for getting familiar with the dataset. We further inspected and vsiualised the dataset in detail to achieve a better understanding of it. This step also involved data-preprocessing to prepare the data for model building. Finally, we built predictive models and applied grid search validation to tune hyper-parameters in each model.

> Stroke dataset is provided by Kaggle: https://www.kaggle.com/datasets/fedesoriano/stroke-prediction-dataset (https://www.kaggle.com/datasets/fedesoriano/stroke-prediction-dataset)

In [1]:

```python
#  Loading some of the libraries that will be use in this project

import pandas as pd
import numpy as np

import matplotlib.pyplot as plt
%matplotlib inline
import seaborn as sns
from matplotlib import pyplot

import warnings
warnings.filterwarnings('ignore')
import copy

from sklearn.preprocessing import LabelEncoder
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split
from sklearn.model_selection import GridSearchCV
from sklearn.metrics import roc_curve, roc_auc_score
from sklearn import metrics
from sklearn.metrics import precision_score, recall_score, f1_score, accuracy_score,confusion_matrix
from sklearn.model_selection import cross_val_score
from sklearn.metrics import auc
from sklearn.metrics import precision_recall_curve
from imblearn.over_sampling import RandomOverSampler
from sklearn.linear_model import LogisticRegression
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import RandomForestClassifier
from xgboost.sklearn import XGBClassifier
import xgboost as xgb
```

Introduction

Return Contents

According to the World Health Organization (WHO) stroke is the 2nd leading cause of death globally, responsible for approximately 11% of total deaths.

This dataset is used to predict whether a patient is likely to get stroke based on the input parameters like gender, age, various diseases, and smoking status. Each row in the data provides relavant information about the patient.

Attribute Information

1. id: unique identifier
2. gender: "Male", "Female" or "Other"
3. age: age of the patient
4. hypertension: 0 if the patient doesn't have hypertension, 1 if the patient has hypertension
5. heart_disease: 0 if the patient doesn't have any heart diseases, 1 if the patient has a heart disease
6. ever_married: "No" or "Yes"
7. work_type: "children", "Govt_jov", "Never_worked", "Private" or "Self-employed"
8. Residence_type: "Rural" or "Urban"
9. avg_glucose_level: average glucose level in blood
10. bmi: body mass index
11. smoking_status: "formerly smoked", "never smoked", "smokes" or "Unknown"*
12. stroke: 1 if the patient had a stroke or 0 if not

*Note: "Unknown" in smoking_status means that the information is unavailable for this patient

Data loading

Return Contents

Loading CSV file for dataset

In [2]:
```
data = pd.read_csv('healthcare-dataset-stroke-data.csv')
data
```

Out[2]:

	id	gender	age	hypertension	heart_disease	ever_married	work_type	Residence_type	avg_glucose_level	bmi	smoking_status	stroke
0	9046	Male	67.0	0	1	Yes	Private	Urban	228.69	36.6	formerly smoked	1
1	51676	Female	61.0	0	0	Yes	Self-employed	Rural	202.21	NaN	never smoked	1
2	31112	Male	80.0	0	1	Yes	Private	Rural	105.92	32.5	never smoked	1
3	60182	Female	49.0	0	0	Yes	Private	Urban	171.23	34.4	smokes	1
4	1665	Female	79.0	1	0	Yes	Self-employed	Rural	174.12	24.0	never smoked	1
...	...	...	...	...	...	...	...	...	...	...	...	...
5105	18234	Female	80.0	1	0	Yes	Private	Urban	83.75	NaN	never smoked	0
5106	44873	Female	81.0	0	0	Yes	Self-employed	Urban	125.20	40.0	never smoked	0
5107	19723	Female	35.0	0	0	Yes	Self-employed	Rural	82.99	30.6	never smoked	0
5108	37544	Male	51.0	0	0	Yes	Private	Rural	166.29	25.6	formerly smoked	0
5109	44679	Female	44.0	0	0	Yes	Govt_job	Urban	85.28	26.2	Unknown	0

5110 rows × 12 columns

In [3]:
```
data.drop(['id'], axis =1, inplace = True)
```

Note:

- There are 5110 rows (instances) and 12 columns (features) in the given dataset
- Column "id" should be dropped as it could lead to overfitting because classifer might use that column to fit perfectly on the training set, ignoring all the other columns

Exploratory data analysis

Return Contents

In [4]:
```
# Datatypes of columns

data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 5110 entries, 0 to 5109
Data columns (total 11 columns):
 #   Column             Non-Null Count  Dtype  
---  ------             --------------  -----  
 0   gender             5110 non-null   object 
 1   age                5110 non-null   float64
 2   hypertension       5110 non-null   int64  
 3   heart_disease      5110 non-null   int64  
 4   ever_married       5110 non-null   object 
 5   work_type          5110 non-null   object 
 6   Residence_type     5110 non-null   object 
 7   avg_glucose_level  5110 non-null   float64
 8   bmi                4909 non-null   float64
 9   smoking_status     5110 non-null   object 
 10  stroke             5110 non-null   int64  
dtypes: float64(3), int64(3), object(5)
memory usage: 439.3+ KB
```

In [5]:
```
# Descriptive statsitics of continuous features

data.describe()
```

Out[5]:

	age	hypertension	heart_disease	avg_glucose_level	bmi	stroke
count	5110.000000	5110.000000	5110.000000	5110.000000	4909.000000	5110.000000
mean	43.226614	0.097456	0.054012	106.147677	28.893237	0.048728
std	22.612647	0.296607	0.226063	45.283560	7.854067	0.215320
min	0.080000	0.000000	0.000000	55.120000	10.300000	0.000000
25%	25.000000	0.000000	0.000000	77.245000	23.500000	0.000000
50%	45.000000	0.000000	0.000000	91.885000	28.100000	0.000000
75%	61.000000	0.000000	0.000000	114.090000	33.100000	0.000000
max	82.000000	1.000000	1.000000	271.740000	97.600000	1.000000

Note:

- Individuals are aged between less than a month old to 82 years
- BMI of indviduals ranges between 10.3 to 97.6 kg/m2

In [6]:
```
# Calculating number of diseased and non diseased individuals

total = len(data)
diseased = len(data[data.stroke==1])
non_diseased = len(data[data.stroke==0])

print ("Total number of individuals : ", total)
print ("Number of individuals with stroke :", diseased)
print ("Number of individuals without stroke : ", non_diseased)
```

```
Total number of individuals :  5110
Number of individuals with stroke : 249
Number of individuals without stroke :  4861
```

In [7]:
```python
# Calculating number of individuals in each gender type

print(data.gender.unique())

print("Other individuals : ", (data['gender']=='Other').sum())
print("Number of males : ", (data['gender']=='Male').sum())
print("Number of females : ", (data['gender']=='Female').sum())
```

```
['Male' 'Female' 'Other']
Other individuals :  1
Number of males :  2115
Number of females :  2994
```

In [8]:
```python
# Dropping individual with gender type "Other"

data.drop(data.index[data['gender'] == 'Other'], inplace = True)
```

In [9]:
```python
# Calculating number of individuals by smoking status

print(data.smoking_status.unique())

print("Unknown status : ", data.smoking_status.value_counts()['Unknown'])
print("Formerly smoked : ",data.smoking_status.value_counts()['formerly smoked'])
print("Never smoked : ", data.smoking_status.value_counts()['never smoked'])
print("Smokes : ", data.smoking_status.value_counts()['smokes'])
```

```
['formerly smoked' 'never smoked' 'smokes' 'Unknown']
Unknown status :  1544
Formerly smoked :  884
Never smoked :  1892
Smokes :  789
```

Note:

- There are a total of 5110 indiviuals out of which only 249 have stroke. This suggests that dataset is highly imbalanced
- There are more females than males in the dataset
- There is only 1 individual with gender category as "Other". Hence we can remove it and conduct analyses only on males and females
- There are more individuals in never smoked category as compared to those in formerly smoked and currently smokes catgeory

In [10]:
```python
# Encode Categorical Columns

categorical_columns = ['gender','ever_married','work_type', 'Residence_type', 'smoking_status']
le = LabelEncoder()
data[categorical_columns] = data[categorical_columns].apply(le.fit_transform)
data.head()
```

Out[10]:

	gender	age	hypertension	heart_disease	ever_married	work_type	Residence_type	avg_glucose_level	bmi	smoking_status	stroke
0	1	67.0	0	1	1	2	1	228.69	36.6	1	1
1	0	61.0	0	0	1	3	0	202.21	NaN	2	1
2	1	80.0	0	1	1	2	0	105.92	32.5	2	1
3	0	49.0	0	0	1	2	1	171.23	34.4	3	1
4	0	79.0	1	0	1	3	0	174.12	24.0	2	1

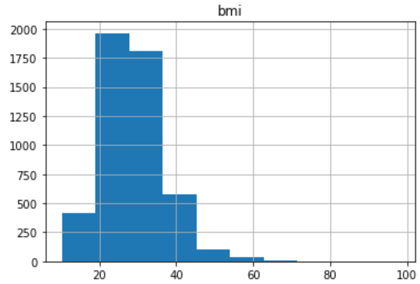
In [11]:
```python
# Checking data for missing values

print('Missing data: ')
print(data.isnull().sum())
```

```
Missing data:
gender                 0
age                    0
hypertension           0
heart_disease          0
ever_married           0
work_type              0
Residence_type         0
avg_glucose_level      0
bmi                  201
smoking_status         0
stroke                 0
dtype: int64
```

In [12]:

```python
# Checking distribution and skewness of BMI column

data.hist(column = 'bmi')
print(data.skew(axis=0))
```

```
gender               0.349410
age                 -0.137430
hypertension         2.715026
heart_disease        3.946786
ever_married        -0.658345
work_type           -0.308679
Residence_type      -0.032506
avg_glucose_level    1.572815
bmi                  1.055063
smoking_status      -0.039430
stroke               4.192807
dtype: float64
```



In [13]:

```python
# Replacing missing valeus with median

data['bmi'].fillna(data['bmi'].median(), inplace=True)
```

In [14]:

```python
data.isnull().sum()
```

Out[14]:

```
gender               0
age                  0
hypertension         0
heart_disease        0
ever_married         0
work_type            0
Residence_type       0
avg_glucose_level    0
bmi                  0
smoking_status       0
stroke               0
dtype: int64
```

Note:

- BMI column has 201 missing values. Instead of dropping the individuals with missing data, we can impute missing BMI values
- The distribution of BMI is positively skewed with value for skewness equals 1.05. Hence, we can impute missing values with median of BMI column.

Data visualisation

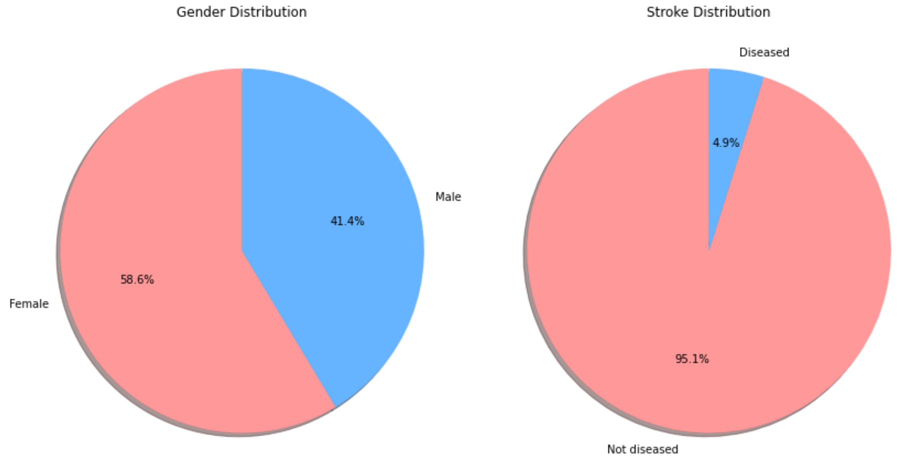
Return Contents

In [15]:

```python
labels_gender = ['Female', 'Male']
labels_disease = ['Not diseased', 'Diseased']
labels_smoking = ['Formerly smoked', 'Never smoked', 'Smokes', 'Unknown']
labels_hypertension = ['No', 'Yes']
labels_heart = ['No', 'Yes']
```

In [16]:

```python
# Displaying gender and stroke distribution in the dataset

fig = plt.subplots(figsize=(12, 8))

colors = ['#ff9999','#66b3ff']

ax1 = plt.subplot2grid((1,2),(0,0))
ax1.pie(data['gender'].value_counts(), labels=labels_gender, autopct='%.1f%%',  colors = colors, shadow = True, startangle=90)
ax1.set_title('Gender Distribution')
plt.tight_layout()

ax2 = plt.subplot2grid((1,2),(0,1))
ax2.pie(data['stroke'].value_counts(), labels=labels_disease, autopct='%.1f%%', colors = colors,  shadow = True, startangle=90)
ax2.set_title('Stroke Distribution')
plt.tight_layout()
```

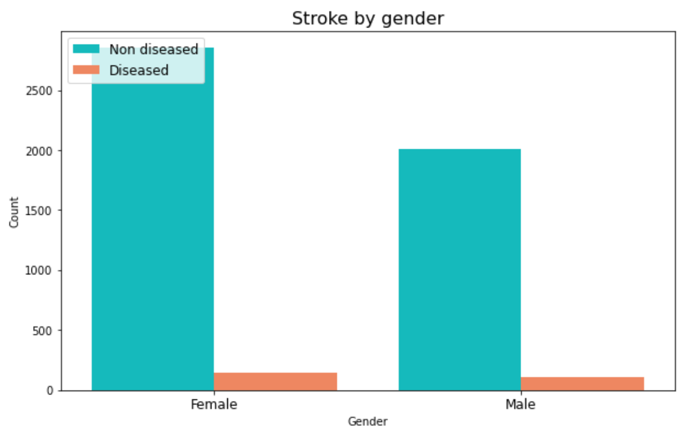


Important findings from pie chart

- Proportion of females is 58.6% and males is 41.4%
- Class label is highly imbalanced with 5% diseased individuals and 95% non diseased individuals

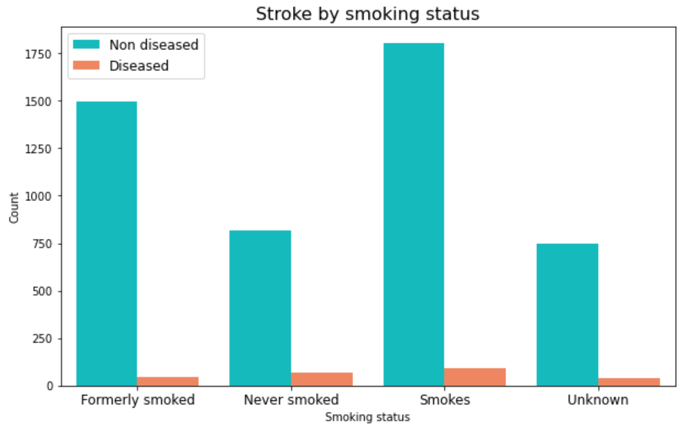
In [17]:

```python
# Displaying distribution of diseased and non-diseased individuals by gender

plt.figure(figsize=(10,6))
ax3 = sns.countplot(data['gender'], hue=data['stroke'], palette=['#00CED1', '#FF7F50'], saturation=0.8)
ax3.set_xticklabels(labels_gender, fontsize=12)
plt.xlabel('Gender')
plt.ylabel('Count')
plt.title('Stroke by gender', fontsize=16)
plt.legend(loc='upper left', fontsize=12, labels=['Non diseased', 'Diseased'])
plt.show()
```



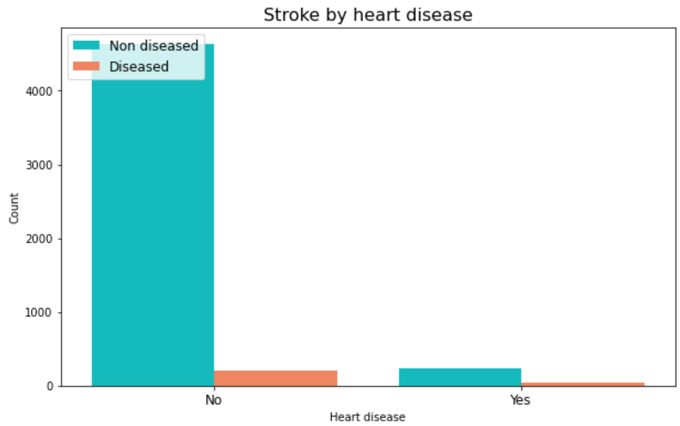
In [18]:

```python
# Displaying distribution of diseased and non-diseased individuals by smoking status

plt.figure(figsize=(10,6))
ax4 = sns.countplot(data['smoking_status'], hue=data['stroke'], palette=['#00CED1', '#FF7F50'], saturation=0.8)
ax4.set_xticklabels(labels_smoking, fontsize=12)
plt.xlabel('Smoking status')
plt.ylabel('Count')
plt.title('Stroke by smoking status', fontsize=16)
plt.legend(loc='upper left', fontsize=12, labels=['Non diseased', 'Diseased'])
plt.show()
```



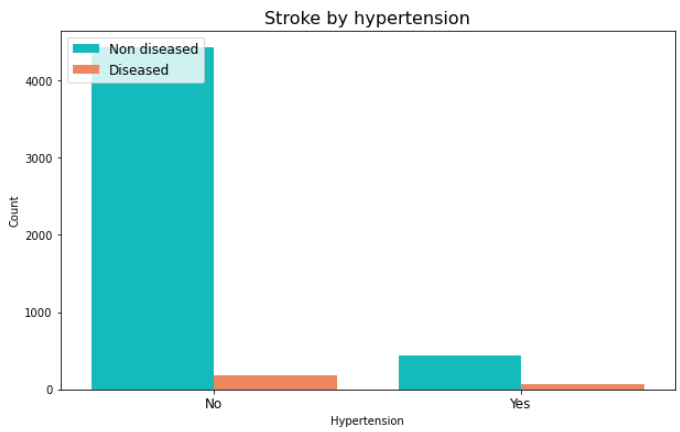
In [19]:

```python
# Displaying distribution of diseased and non-diseased individuals by heart disease

plt.figure(figsize=(10,6))
ax5 = sns.countplot(data['heart_disease'], hue=data['stroke'], palette=['#00CED1', '#FF7F50'], saturation=0.8)
ax5.set_xticklabels(labels_heart, fontsize=12)
plt.xlabel('Heart disease')
plt.ylabel('Count')
plt.title('Stroke by heart disease', fontsize=16)
plt.legend(loc='upper left', fontsize=12, labels=['Non diseased', 'Diseased'])
plt.show()
```



In [20]:

```python
# Displaying distribution of diseased and non-diseased individuals by hypertension

plt.figure(figsize=(10,6))
ax6 = sns.countplot(data['hypertension'], hue=data['stroke'], palette=['#00CED1', '#FF7F50'], saturation=0.8)
ax6.set_xticklabels(labels_hypertension, fontsize=12)
plt.xlabel('Hypertension')
plt.ylabel('Count')
plt.title('Stroke by hypertension', fontsize=16)
plt.legend(loc='upper left', fontsize=12, labels=['Non diseased', 'Diseased'])
plt.show()
```



Important findings from count plots

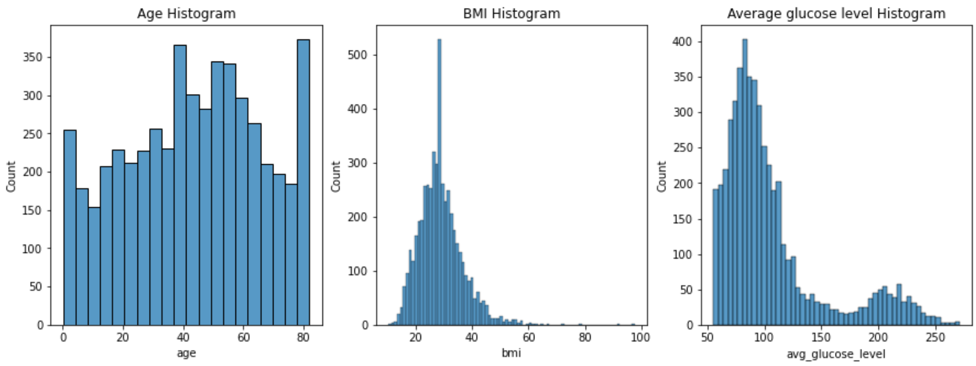
- Count of males and females with stroke is approximately similar
- The number of individuals with stroke are slightly more in currently smokes category as compared to other smoking categories
- There are more individuals without hypertension and heart disease who suffer from stroke as compared to those with hypertension and stroke

In [21]:

```python
# Displaying histogram for Age, BMI and Average glucose level

fig, axes = plt.subplots(1, 3, figsize=(15, 5))

# Histogram Plot for Age

ax8 = sns.histplot(x='age', data=data, ax = axes[0])
ax8.set_title("Age Histogram")

# Histogram Plot for BMI

ax9 = sns.histplot(x='bmi', data=data, ax =axes[1])
ax9.set_title("BMI Histogram")

# Histogram Plot for Average glucose level

ax10 = sns.histplot(x='avg_glucose_level', data=data, ax =axes[2])
ax10.set_title("Average glucose level Histogram")

```

Out[21]:

```
Text(0.5, 1.0, 'Average glucose level Histogram')
```



Important findings from histogram

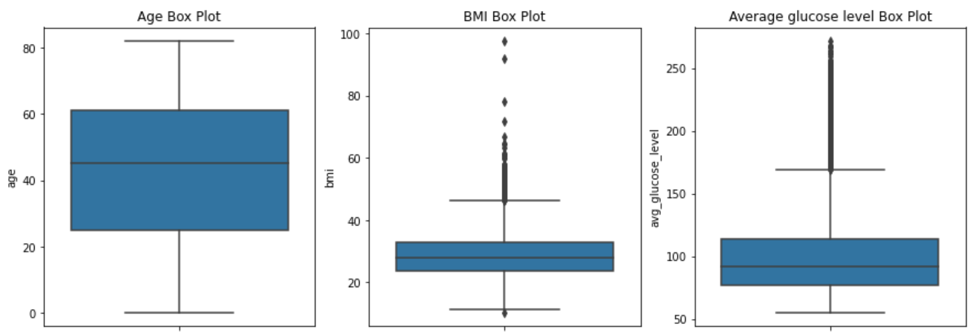
- Distribution of BMI and average glucose level is highly positively skewed
- Distribution of age is approximately normal

In [22]:

```python
# Displaying histogram for Age, BMI and Average glucose level

# Box Plot for Age
fig, axes = plt.subplots(1, 3, figsize=(15, 5))

ax14 = sns.boxplot(y= 'age', data=data, ax = axes[0])
ax14.set_title("Age Box Plot")

# Box Plot for BMI

ax15 = sns.boxplot(y= 'bmi', data=data, ax =axes[1])
ax15.set_title("BMI Box Plot")

# Box Plot for Average glucose level

ax16 = sns.boxplot(y='avg_glucose_level', data=data, ax =axes[2])
ax16.set_title("Average glucose level Box Plot")
```

Out[22]:

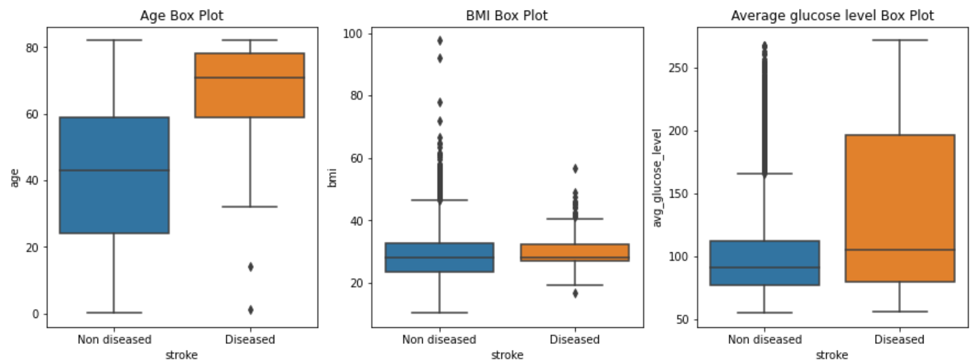
```
Text(0.5, 1.0, 'Average glucose level Box Plot')
```



In [23]:

```python
# Displaying boxplot for Age, BMI and Average glucose level by disease status


# Box Plot for Age
fig, axes = plt.subplots(1, 3, figsize=(15, 5))

ax11 = sns.boxplot(x='stroke', y= 'age', data=data, ax = axes[0])
ax11.set_title("Age Box Plot")
ax11.set_xticklabels(['Non diseased', 'Diseased'])

# Box Plot for BMI

ax12 = sns.boxplot(x='stroke', y= 'bmi', data=data, ax =axes[1])
ax12.set_title("BMI Box Plot")
ax12.set_xticklabels(['Non diseased', 'Diseased'])

# Box Plot for Average glucose level

ax13 = sns.boxplot(x='stroke', y= 'avg_glucose_level', data=data, ax =axes[2])
ax13.set_title("Average glucose level Box Plot")
ax13.set_xticklabels(['Non diseased', 'Diseased'])
```

Out[23]: [Text(0, 0, 'Non diseased'), Text(1, 0, 'Diseased')]

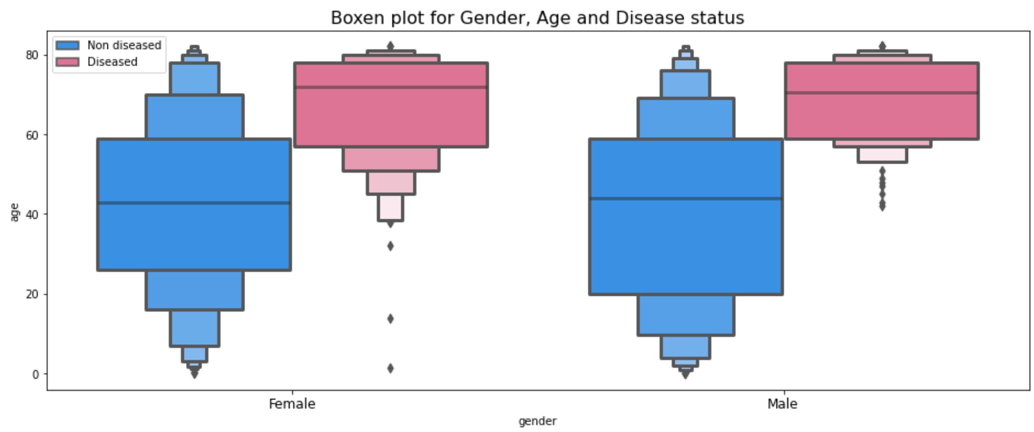


Important findings from box plots

- Distribution of all 3 features age, bmi and average glucose level is highly dispersed with high range score
- There is a greater variability in non diseased individuals for bmi and average glucose level as well as larger outliers.

In [24]:

```python
# Displaying distribution of diseased and non-diseased individuals by gender and age

plt.figure(figsize=(16, 6))
palette=['#1E90FF', '#ee638f']
labels_gender = ['Female', 'Male']
ax7=sns.boxenplot(x=data.gender, y=data.age, hue=data.stroke, palette=palette, linewidth=3)
ax7.set_xticklabels(labels_gender, fontsize=12)
handles = ax7.get_legend_handles_labels()[0]
ax7.legend(handles, ['Non diseased', 'Diseased'], loc = 'upper left')
ax7.set_title("Boxen plot for Gender, Age and Disease status",fontsize=16)
plt.show()
```

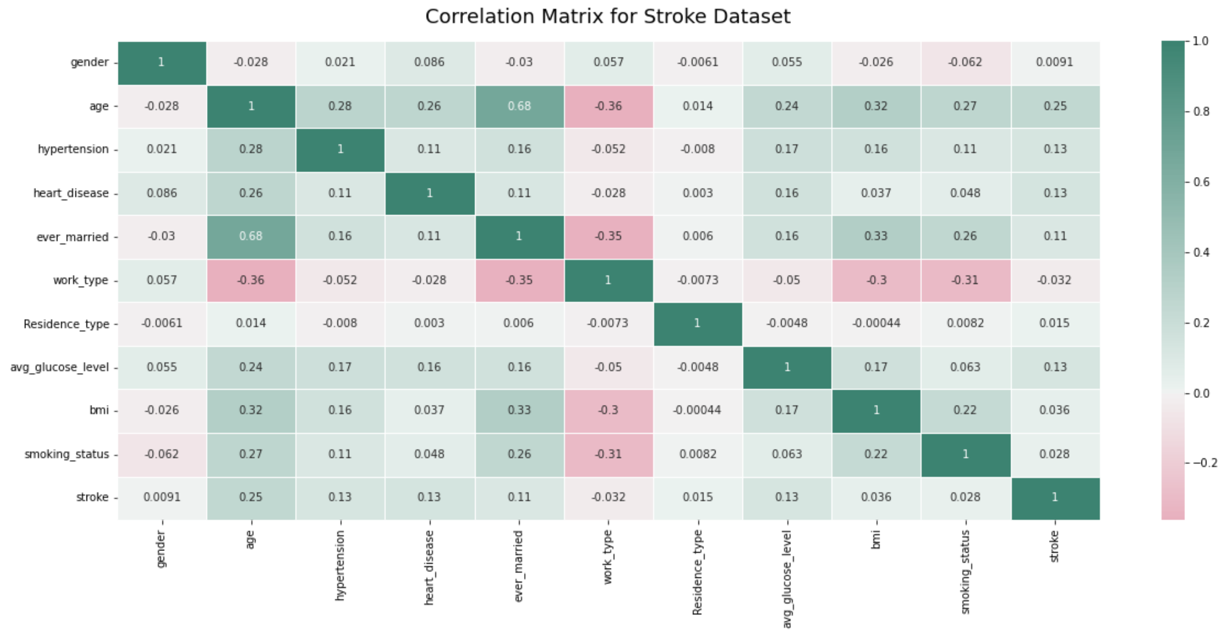


Important findings from boxen plot

- Older individuals suffer from stroke more as compared to younger individuals. However, there are some outliers in stroke group with childhood ages. Although, this is very rare but paediatric stroke is still possible.
- Men suffer from stroke at a slightly older age as compared to women

In [25]:

```python
# Correlation matrix

plt.figure(figsize=(20, 8))
cmap = sns.diverging_palette(2, 165, as_cmap=True)
heatmap =sns.heatmap(data.corr(),  linewidths=1, cmap=cmap, annot = True, center=0)
heatmap.set_title('Correlation Matrix for Stroke Dataset', fontdict={'fontsize':18}, pad=16);
```



Important findings from correlation matrix

- There are no strong correlations observed between features as well as between features and class label

Model building

Return Contents

In [26]:

```python
# Dividing data into input features and output class
X, y = data.iloc[:, :-1], data.iloc[:, -1:]
```

In [27]:

```python
# Input data

X.head()
```

Out[27]:

	gender	age	hypertension	heart_disease	ever_married	work_type	Residence_type	avg_glucose_level	bmi	smoking_status
0	1	67.0	0	1	1	2	1	228.69	36.6	1
1	0	61.0	0	0	1	3	0	202.21	28.1	2
2	1	80.0	0	1	1	2	0	105.92	32.5	2
3	0	49.0	0	0	1	2	1	171.23	34.4	3
4	0	79.0	1	0	1	3	0	174.12	24.0	2

In [28]:

```python
# Class label

y.head()
```

Out[28]:

	stroke
0	1
1	1
2	1
3	1
4	1

In [29]:

```python
print(X.shape)
print(y.shape)
```

```
(5109, 10)
(5109, 1)
```

In [30]:
```
#  Splitting data into train and test

X_train_tune, X_test_df, y_train_tune, y_test_df = train_test_split(X, y, test_size=0.2, random_state=1)

X_train_df, X_val_df, y_train_df, y_val_df = train_test_split(X_train_tune, y_train_tune, test_size=0.15, random_state=1) # 0.25 x 0.8 = 0

# X_train_df, X_test_df, y_train_df, y_test_df = train_test_split(X, y, test_size=0.3, random_state= 0)
```

In [31]:
```
print("Train Shape: {}".format(X_train_df.shape))
print("Validation Shape: {}".format(X_val_df.shape))
print("Test Shape: {}".format(X_test_df.shape))
```

```
Train Shape: (3473, 10)
Validation Shape: (614, 10)
Test Shape: (1022, 10)
```

In [32]:
```
# Performing oversampling

OS = RandomOverSampler(random_state=0)
osx, osy = OS.fit_resample(X_train_df, y_train_df)
print("X_train shape after oversampling: ", osx.shape)
print("y_train shape after oversampling: ", osy.shape)
```

```
X_train shape after oversampling:  (6606, 10)
y_train shape after oversampling:  (6606, 1)
```

In [33]:
```
# Performing standardization

sc = StandardScaler()
X_train = sc.fit_transform(osx)
X_val = sc.fit_transform(X_val_df)
X_test = sc.fit_transform(X_test_df)
```

In [34]:
```
X_train
```

Out[34]:
```
array([[ 1.20617142,  0.31939861, -0.46826179, ..., -0.61591723,
         1.01154191, -0.42030218],
       [-0.82906955, -0.71710541, -0.46826179, ..., -0.71883621,
        -0.98632002,  0.5467881 ],
       [ 1.20617142, -2.06906718, -0.46826179, ...,  0.04870741,
        -1.57392647, -1.38739245],
       ...,
       [ 1.20617142, -0.04112452, -0.46826179, ..., -0.5664509 ,
         0.24765353,  1.51387837],
       [-0.82906955,  0.86018332, -0.46826179, ..., -0.80001174,
        -0.41340373,  0.5467881 ],
       [-0.82906955,  1.04044489,  2.13555755, ...,  1.53577786,
         2.39241707,  0.5467881 ]])
```

In [35]:
```
X_val
```

Out[35]:
```
array([[-0.85130435,  0.38147098, -0.3105295 , ...,  0.03177528,
         0.25078602,  0.62127172],
       [-0.85130435, -1.42357819, -0.3105295 , ..., -0.79802466,
        -1.88840489, -1.2578457 ],
       [-0.85130435,  0.33634475, -0.3105295 , ..., -0.3935435 ,
         0.42035603,  0.62127172],
       ...,
       [-0.85130435, -0.65643229, -0.3105295 , ..., -0.71652645,
        -0.3100994 ,  1.56083043],
       [ 1.17466802, -0.61130606,  3.22030594, ..., -0.61581134,
        -0.10139785, -0.31828699],
       [-0.85130435, -0.61130606, -0.3105295 , ..., -0.74685675,
        -0.16661709,  0.62127172]])
```

In [36]:
```
X_test
```

Out[36]:
```
array([[ 1.21926946, -1.55443213, -0.3401772 , ..., -1.05401379,
        -1.26468711, -1.35160688],
       [ 1.21926946, -1.81639089, -0.3401772 , ..., -0.34378401,
        -0.73833399, -1.35160688],
       [-0.82016325, -1.77273109, -0.3401772 , ..., -0.36412003,
        -1.72837916, -1.35160688],
       ...,
       [ 1.21926946, -1.51077234, -0.3401772 , ..., -0.24997595,
        -0.73833399, -1.35160688],
       [-0.82016325,  0.10463996, -0.3401772 , ..., -0.320168  ,
        -0.53781851,  0.5419397 ],
       [ 1.21926946, -0.20097858, -0.3401772 , ...,  0.16964564,
         1.39214295, -0.40483359]])
```

In [37]:
```
# Converting dataframe to numpy array

y_train = osy.to_numpy().ravel()
y_train
```

Out[37]:
```
array([0, 0, 0, ..., 1, 1, 1])
```

In [38]:
```python
y_val = y_val_df.to_numpy().ravel()
y_val
```

Out[38]:
```
array([0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 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, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0])
```

In [39]:
```python
y_test = y_test_df.to_numpy().ravel()
y_test
```

Out[39]:
```
array([0, 0, 0, ..., 0, 0, 0])
```

Logistic Regression

In [40]:
```python
# Logistic regression

lor = LogisticRegression(max_iter = 20000)
lor.fit(X_train, y_train)
y_val_pred_lor = lor.predict(X_val)
```

In [41]:
```python
# Printing scores for accuracy, precision, recall and F1 with threshold 0.5

print("Accuracy : ", metrics.accuracy_score(y_val, y_val_pred_lor))
print("Precision : ", metrics.precision_score(y_val, y_val_pred_lor))
print("Recall : ", metrics.recall_score(y_val, y_val_pred_lor))
print("F1 score : ", metrics.f1_score(y_val, y_val_pred_lor))
```

```
Accuracy :  0.5553745928338762
Precision :  0.059027777777777776
Recall :  0.8947368421052632
F1 score :  0.11074918566775245
```

In [42]:
```python
val_lor_proba = lor.predict_proba(X_val)
y_val_prob_lor = val_lor_proba[:, 1]
y_val_prob_lor
```

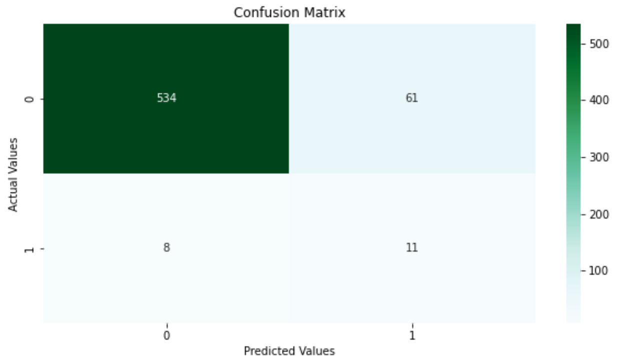
Out[42]:
```
array([0.5901919 , 0.07348657, 0.55158436, 0.8692085 , 0.81749494,
       0.03563532, 0.93889825, 0.04000919, 0.13656255, 0.21132713,
       0.13315282, 0.14863129, 0.8012939 , 0.20443922, 0.50997239,
       0.09294311, 0.07758153, 0.43499084, 0.5294772 , 0.34350958,
       0.17778339, 0.84533251, 0.12316908, 0.0367498 , 0.13534342,
       0.28062785, 0.10598128, 0.97459354, 0.16509053, 0.96784532,
       0.59833853, 0.76919572, 0.80635983, 0.46907555, 0.94735302,
       0.6283407 , 0.70578857, 0.92609312, 0.57164407, 0.63196893,
       0.07222791, 0.40361193, 0.74390412, 0.42652412, 0.25841966,
       0.60342324, 0.61226367, 0.79849076, 0.92239013, 0.05378858,
       0.69976722, 0.75610241, 0.58323382, 0.04803207, 0.06110172,
       0.91880072, 0.38966455, 0.90732755, 0.1102547 , 0.26207897,
       0.76322798, 0.2811851 , 0.11135666, 0.38189743, 0.76340337,
       0.11210117, 0.81468001, 0.62315942, 0.84396673, 0.60669488,
       0.3587874 , 0.18129746, 0.29918093, 0.54519342, 0.61435371,
       0.95067833, 0.75847122, 0.87901685, 0.15169192, 0.91614148,
       0.38326964, 0.29508097, 0.18540214, 0.13517798, 0.02784977,
       0.52395234, 0.96421653, 0.24765595, 0.52412349, 0.39798421,
       0.24177847, 0.0629371 , 0.16683823, 0.95039168, 0.54809554,
```

In [43]:
```python
def lor_bestThreshold(y_true, y_pred):

# Calculating best threshold for maximum F1 score
    P, R, T = precision_recall_curve(y_true, y_pred.round(3))
    F1index, = np.where( (2*(P*R)/(P+R)) == max((2*(P*R)/(P+R))))
    global lor_f1index
    lor_f1index = T[F1index][0]
    print("Best Threshold for maximum F1 score: ", lor_f1index)
```

In [44]:
```python
def lor_performance(y_true, y_pred):

# Predicting class label based on best threshold

    y_pred_new = np.where(y_pred >= lor_f1index, 1, 0)

# Calculating performace metrices

    pf1 = metrics.precision_score(y_true, y_pred_new)
    rf1 = metrics.recall_score(y_true,y_pred_new)
    f1_1 = (2 * (pf1*rf1) / ( pf1 + rf1))

# Printing accuracy, precision, recall and F1 score:

    print('Accuracy: %.3f' % metrics.accuracy_score(y_true, y_pred_new.round(2)))
    print("Precision at Best F1 :", pf1)
    print("Recall at Best F1 :", rf1)
    print("Best F1 Score :", f1_1)

# Printing confusion matrix

    lor_cm = confusion_matrix(y_true, y_pred_new)
    lor_df = pd.DataFrame(lor_cm,
                     index = [0, 1],
                     columns = [0, 1])
    plt.figure(figsize=(10,5))
    sns.heatmap(lor_df, fmt="d", cmap = 'BuGn', annot=True)
    plt.title('Confusion Matrix')
    plt.ylabel('Actual Values')
    plt.xlabel('Predicted Values')
    plt.show()
```

In [45]:
```python
# Best threshold for F1 score and performance on validaton set

lor_bestThreshold(y_val, y_val_prob_lor)
lor_performance(y_val, y_val_prob_lor)
```

```
Best Threshold for maximum F1 score:  0.894
Accuracy: 0.888
Precision at Best F1 : 0.15277777777777778
Recall at Best F1 : 0.5789473684210527
Best F1 Score : 0.2417582417582418
```

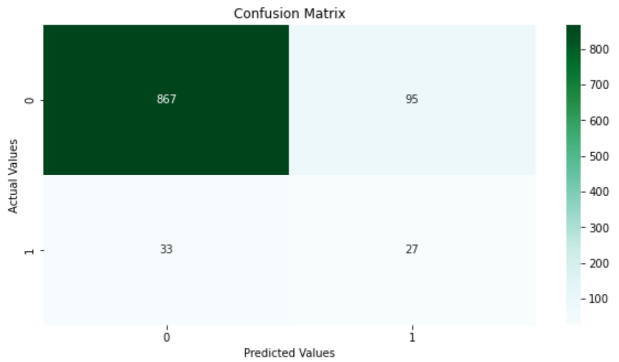


In [46]:
```python
test_lor_proba = lor.predict_proba(X_test)
y_test_prob_lor = test_lor_proba[:, 1]
y_test_prob_lor
```

Out[46]:
```
array([0.04513708, 0.03643784, 0.04723962, ..., 0.06081127, 0.47787425,
       0.31931657])
```

In [47]:
```python
# Test set peformance: Best threshold  F1

lor_performance(y_test, y_test_prob_lor)
```

```
Accuracy: 0.875
Precision at Best F1 : 0.22131147540983606
Recall at Best F1 : 0.45
Best F1 Score : 0.2967032967032967
```



Decision Tree

In [48]:
```python
#  Decision Tree

dtc = DecisionTreeClassifier(criterion='gini', max_depth=3, min_samples_split=3, min_samples_leaf=1)
dtc.fit(X_train, y_train)
y_val_pred_dtc = dtc.predict(X_val)
```

In [49]:

```
# Printing scores for accuracy, precision, recall and F1 with threshold 0.5

print("Accuracy : ", metrics.accuracy_score(y_val, y_val_pred_dtc))
print("Precision : ", metrics.precision_score(y_val, y_val_pred_dtc))
print("Recall : ", metrics.recall_score(y_val, y_val_pred_dtc))
print("F1 score : ", metrics.f1_score(y_val, y_val_pred_dtc))
```

```
Accuracy :  0.49185667752442996
Precision :  0.04923076923076923
Recall :  0.8421052631578947
F1 score :  0.0930232558139535
```

In [50]:

```
val_dtc_proba = dtc.predict_proba(X_val)
y_val_prob_dtc = val_dtc_proba[:, 1]
y_val_prob_dtc
```

Out[50]:

```
array([0.62885906, 0.01351351, 0.62885906, 0.7522604 , 0.62885906,
       0.01351351, 0.85365854, 0.01351351, 0.01351351, 0.21116139,
       0.01351351, 0.01351351, 0.62885906, 0.21116139, 0.32404181,
       0.01351351, 0.01351351, 0.62885906, 0.62885906, 0.21116139,
       0.01351351, 0.7522604 , 0.01351351, 0.01351351, 0.01351351,
       0.21116139, 0.01351351, 0.85365854, 0.01351351, 0.85365854,
       0.62885906, 0.85365854, 0.62885906, 0.62885906, 0.85365854,
       0.7522604 , 0.85365854, 0.7522604 , 0.62885906, 0.62885906,
       0.01351351, 0.32404181, 0.62885906, 0.62885906, 0.21116139,
       0.62885906, 0.7522604 , 0.85365854, 0.85365854, 0.01351351,
       0.62885906, 0.85365854, 0.62885906, 0.01351351, 0.01351351,
       0.85365854, 0.32404181, 0.7522604 , 0.01351351, 0.21116139,
       0.85365854, 0.21116139, 0.01351351, 0.32404181, 0.7522604 ,
       0.01351351, 0.7522604 , 0.62885906, 0.7522604 , 0.62885906,
       0.62885906, 0.21116139, 0.21116139, 0.62885906, 0.32404181,
       0.7522604 , 0.7522604 , 0.62885906, 0.01351351, 0.7522604 ,
       0.21116139, 0.21116139, 0.01351351, 0.01351351, 0.01351351,
       0.62885906, 0.85365854, 0.21116139, 0.32404181, 0.62885906,
       0.21116139, 0.01351351, 0.01351351, 0.85365854, 0.62885906,
```

In [51]:

```
def dtc_bestThreshold(y_true, y_pred):

# Calculating best threshold for maximum F1 score
    P, R, T = precision_recall_curve(y_true, y_pred.round(3))
    F1index, = np.where( (2*(P*R)/(P+R)) == max((2*(P*R)/(P+R))))
    global dtc_f1index
    dtc_f1index = T[F1index][0]
    print("Best Threshold for maximum F1 score: ", dtc_f1index)
```

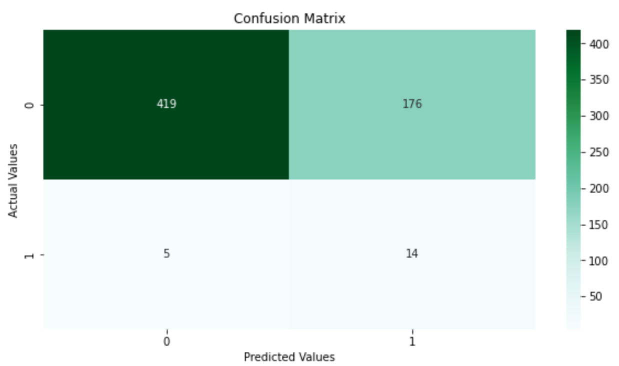
In [52]:

```
def dtc_performance(y_true, y_pred):

# Predicting class label based on best threshold

    y_pred_new = np.where(y_pred >= dtc_f1index, 1, 0)

# Calculating performace metrices

    pf1 = metrics.precision_score(y_true, y_pred_new)
    rf1 = metrics.recall_score(y_true,y_pred_new)
    f1_1 = (2 * (pf1*rf1) / ( pf1 + rf1))

# Printing accuracy, precision, recall and F1 score:

    print('Accuracy: %.3f' % metrics.accuracy_score(y_true, y_pred_new.round(2)))
    print("Precision at Best F1 :", pf1)
    print("Recall at Best F1 :", rf1)
    print("Best F1 Score :", f1_1)

# Printing confusion matrix

    dtc_cm = confusion_matrix(y_true, y_pred_new)
    dtc_df = pd.DataFrame(dtc_cm,
                     index = [0, 1],
                     columns = [0, 1])
    plt.figure(figsize=(10,5))
    sns.heatmap(dtc_df, fmt="d", cmap = 'BuGn', annot=True)
    plt.title('Confusion Matrix')
    plt.ylabel('Actual Values')
    plt.xlabel('Predicted Values')
    plt.show()
```

In [53]:

```
# Best threshold for F1 score and performance on validaton set

dtc_bestThreshold(y_val, y_val_prob_dtc)
dtc_performance(y_val, y_val_prob_dtc)
```

```
Best Threshold for maximum F1 score:  0.752
Accuracy: 0.705
Precision at Best F1 : 0.07368421052631578
Recall at Best F1 : 0.7368421052631579
Best F1 Score : 0.13397129186602869
```

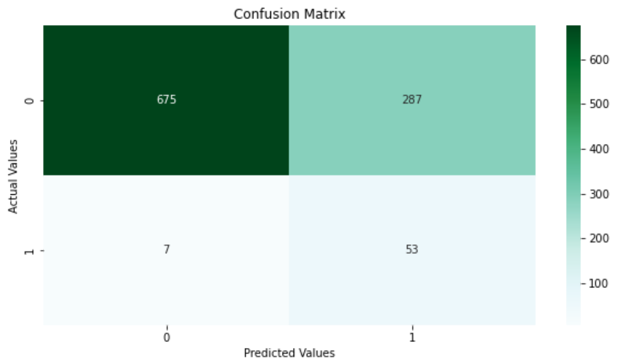


In [54]:
```python
test_dtc_proba = dtc.predict_proba(X_test)
y_test_prob_dtc = test_dtc_proba[:, 1]
y_test_prob_dtc
```

Out[54]:
```
array([0.01351351, 0.01351351, 0.01351351, ..., 0.01351351, 0.62885906,
       0.62885906])
```

In [55]:
```python
# Test set peformance: Best threshold  F1
dtc_performance(y_test, y_test_prob_dtc)
```

```
Accuracy: 0.712
Precision at Best F1 : 0.15588235294117647
Recall at Best F1 : 0.8833333333333333
Best F1 Score : 0.265
```



Random Forest

In [56]:
```python
# Random Forest

rfc =  RandomForestClassifier(n_estimators=50, max_depth=3, min_samples_split=4, min_samples_leaf = 1)
rfc.fit(X_train, y_train)
y_val_pred_rfc = rfc.predict(X_val)
```

In [57]:
```python
# Printing scores for accuracy, precision, recall and F1 with threshold 0.5
print("Accuracy : ", metrics.accuracy_score(y_val, y_val_pred_rfc))
print("Precision : ", metrics.precision_score(y_val, y_val_pred_rfc))
print("Recall : ", metrics.recall_score(y_val, y_val_pred_rfc))
print("F1 score : ", metrics.f1_score(y_val, y_val_pred_rfc))
```

```
Accuracy :  0.511400651465798
Precision :  0.056782334384858045
Recall :  0.9473684210526315
F1 score :  0.10714285714285715
```

In [58]:
```python
val_rfc_proba = rfc.predict_proba(X_val)
y_val_prob_rfc = val_rfc_proba[:, 1]
y_val_prob_rfc
```

Out[58]:
```
array([0.59297157, 0.04094033, 0.57607762, 0.66224072, 0.74110192,
       0.03505334, 0.73664409, 0.0334643 , 0.1919148 , 0.25479207,
       0.20719544, 0.23448558, 0.62123817, 0.26580574, 0.5529246 ,
       0.04051241, 0.06928641, 0.512068  , 0.55490021, 0.27702061,
       0.21483538, 0.66255897, 0.03763732, 0.03482201, 0.20229804,
       0.27561875, 0.21055901, 0.75089014, 0.19342663, 0.7350246 ,
       0.59200824, 0.61872996, 0.63648861, 0.46726754, 0.67850677,
       0.5970046 , 0.58022316, 0.60471499, 0.59033394, 0.69389336,
       0.19971047, 0.48370525, 0.63375816, 0.45178967, 0.24991452,
       0.5956311 , 0.60950397, 0.659911  , 0.63458148, 0.07824234,
       0.6166636 , 0.67612425, 0.61981484, 0.03839601, 0.03839601,
       0.74959841, 0.43308421, 0.62148192, 0.20026379, 0.27190205,
       0.63194384, 0.29664841, 0.192634  , 0.43031146, 0.60082409,
       0.21271897, 0.62046825, 0.591809  , 0.66735782, 0.59548848,
       0.47048845, 0.26869351, 0.27827135, 0.60220737, 0.56976963,
       0.68153672, 0.6031155 , 0.64177233, 0.21733885, 0.61990574,
       0.26574834, 0.30214481, 0.19012038, 0.21658015, 0.0885531 ,
       0.5777183 , 0.70575606, 0.27993626, 0.55400065, 0.4896141 ,
       0.27629352, 0.06928641, 0.21259379, 0.6440106 , 0.59189552,
```

In [59]:
```python
def rfc_bestThreshold(y_true, y_pred):

# Calculating best threshold for maximum F1 score
    P, R, T = precision_recall_curve(y_true, y_pred.round(3))
    F1index, = np.where( (2*(P*R)/(P+R)) == max((2*(P*R)/(P+R))))
    global rfc_f1index
    rfc_f1index = T[F1index][0]
    print("Best Threshold for maximum F1 score: ", rfc_f1index)
```

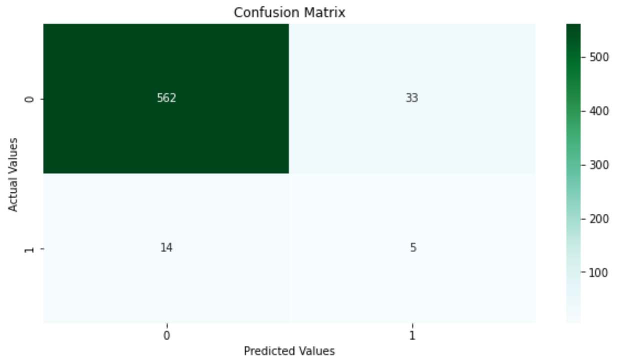
In [60]:

```python
def rfc_performance(y_true, y_pred):

# Predicting class label based on best threshold

    y_pred_new = np.where(y_pred >= rfc_f1index, 1, 0)

# Calculating performace metrices

    pf1 = metrics.precision_score(y_true, y_pred_new)
    rf1 = metrics.recall_score(y_true,y_pred_new)
    f1_1 = (2 * (pf1*rf1) / ( pf1 + rf1))

# Printing accuracy, precision, recall and F1 score:

    print('Accuracy: %.3f' % metrics.accuracy_score(y_true, y_pred_new.round(2)))
    print("Precision at Best F1 :", pf1)
    print("Recall at Best F1 :", rf1)
    print("Best F1 Score :", f1_1)

# Printing confusion matrix

    rfc_cm = confusion_matrix(y_true, y_pred_new)
    rfc_df = pd.DataFrame(rfc_cm,
                      index = [0, 1],
                      columns = [0, 1])
    plt.figure(figsize=(10,5))
    sns.heatmap(rfc_df, fmt="d", cmap = 'BuGn', annot=True)
    plt.title('Confusion Matrix')
    plt.ylabel('Actual Values')
    plt.xlabel('Predicted Values')
    plt.show()
```

In [61]:

```python
# Best threshold for F1 score and performance on validaton set

rfc_bestThreshold(y_val, y_val_prob_rfc)
rfc_performance(y_val, y_val_prob_rfc)
```

```
Best Threshold for maximum F1 score:  0.726
Accuracy: 0.923
Precision at Best F1 : 0.13157894736842105
Recall at Best F1 : 0.2631578947368421
Best F1 Score : 0.17543859649122803
```



In [62]:

```python
test_rfc_proba = rfc.predict_proba(X_test)
y_test_prob_rfc = test_rfc_proba[:, 1]
y_test_prob_rfc
```

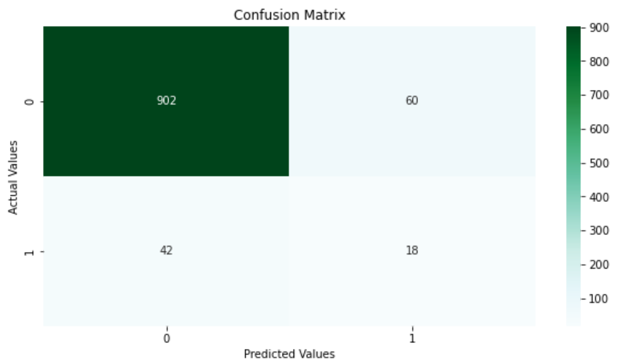
Out[62]:

```
array([0.04400927, 0.07936583, 0.03848498, ..., 0.07936583, 0.53176916,
       0.45801838])
```

In [63]:

```python
# Test set peformance: Best threshold  F1

rfc_performance(y_test, y_test_prob_rfc)
```

```
Accuracy: 0.900
Precision at Best F1 : 0.23076923076923078
Recall at Best F1 : 0.3
Best F1 Score : 0.2608695652173913
```



XGBoost

In [64]:

```python
# XGBoost

xgb = XGBClassifier(booster = 'gbtree', learning_rate = 0.2,  max_depth = 3, min_child_weight = 4, n_estimators  = 100)
xgb.fit(X_train, y_train)
y_val_pred_xgb = xgb.predict(X_val)
```

In [65]:

```python
# Printing scores for accuracy, precision, recall and F1 with threshold 0.5
print("Accuracy : ", metrics.accuracy_score(y_val, y_val_pred_xgb))
print("Precision : ", metrics.precision_score(y_val, y_val_pred_xgb))
print("Recall : ", metrics.recall_score(y_val, y_val_pred_xgb))
print("F1 score : ", metrics.f1_score(y_val, y_val_pred_xgb))
```

```
Accuracy :  0.737785016286645
Precision :  0.05625
Recall :  0.47368421052631576
F1 score :  0.1005586592178771
```

In [66]:

```python
val_xgb_proba = xgb.predict_proba(X_val)
y_val_prob_xgb = val_xgb_proba[:, 1]
y_val_prob_xgb
```

Out[66]:

```
array([6.78817153e-01, 2.98074214e-03, 3.70944083e-01, 7.87746072e-01,
       4.77646738e-01, 1.98925799e-03, 5.62546790e-01, 4.13316506e-04,
       9.00347997e-03, 3.64418253e-02, 1.21001888e-03, 2.25282721e-02,
       4.02316898e-02, 4.70763594e-02, 3.32985967e-01, 5.70106134e-03,
       2.30093766e-03, 6.05080366e-01, 6.83031827e-02, 4.31910791e-02,
       1.33705884e-02, 1.68648228e-01, 3.02909000e-04, 1.49806775e-03,
       2.99105770e-04, 8.07479247e-02, 4.54469398e-02, 5.08826733e-01,
       2.56600732e-04, 7.44657040e-01, 9.39358249e-02, 5.69034040e-01,
       3.43420744e-01, 4.61541861e-02, 4.77598786e-01, 2.09523112e-01,
       5.61590254e-01, 3.15018177e-01, 5.36622286e-01, 7.00215101e-01,
       9.56199598e-04, 5.67753434e-01, 1.17981724e-01, 8.37576240e-02,
       3.31274047e-02, 5.41981578e-01, 4.91242260e-01, 5.63326955e-01,
       6.90959632e-01, 3.37825762e-03, 5.37757754e-01, 6.28371656e-01,
       2.46840447e-01, 6.62178500e-04, 3.83408013e-04, 8.05537879e-01,
       4.15352322e-02, 5.15885472e-01, 2.82293232e-03, 9.95177496e-03,
       4.18184280e-01, 2.12080926e-01, 1.79896061e-03, 4.76122871e-02,
       7.49996185e-01, 2.89719389e-03, 8.67162943e-01, 5.49740732e-01,
       2.25281745e-01, 5.26018560e-01, 4.78884697e-01, 2.71404423e-02,
       1.09763034e-01, 1.38265774e-01, 1.88607216e-01, 6.98699415e-01,
```

In [67]:

```python
def xgb_bestThreshold(y_true, y_pred):

# Calculating best threshold for maximum F1 score
    P, R, T = precision_recall_curve(y_true, y_pred.round(3))
    F1index, = np.where( (2*(P*R)/(P+R)) == max((2*(P*R)/(P+R))))
    global xgb_f1index
    xgb_f1index = T[F1index][0]
    print("Best Threshold for maximum F1 score: ", xgb_f1index)
```

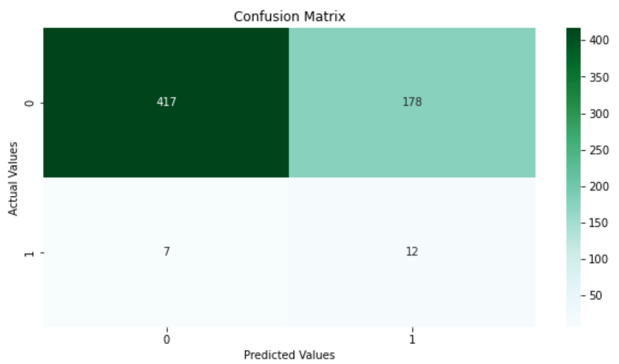
In [68]:

```python
def xgb_performance(y_true, y_pred):

# Predicting class label based on best threshold

    y_pred_new = np.where(y_pred >= xgb_f1index, 1, 0)

# Calculating performace metrices

    pf1 = metrics.precision_score(y_true, y_pred_new)
    rf1 = metrics.recall_score(y_true,y_pred_new)
    f1_1 = (2 * (pf1*rf1) / ( pf1 + rf1))

# Printing accuracy, precision, recall and F1 score:

    print('Accuracy: %.3f' % metrics.accuracy_score(y_true, y_pred_new.round(2)))
    print("Precision at Best F1 :", pf1)
    print("Recall at Best F1 :", rf1)
    print("Best F1 Score :", f1_1)

# Printing confusion matrix

    xgb_cm = confusion_matrix(y_true, y_pred_new)
    xgb_df = pd.DataFrame(xgb_cm,
                     index = [0, 1],
                     columns = [0, 1])
    plt.figure(figsize=(10,5))
    sns.heatmap(xgb_df, fmt="d", cmap = 'BuGn', annot=True)
    plt.title('Confusion Matrix')
    plt.ylabel('Actual Values')
    plt.xlabel('Predicted Values')
    plt.show()
```

In [69]:

```python
# Best threshold for F1 score and performance on validaton set

xgb_bestThreshold(y_val, y_val_prob_xgb)
xgb_performance(y_val, y_val_prob_xgb)
```

```
Best Threshold for maximum F1 score:  0.439
Accuracy: 0.699
Precision at Best F1 : 0.06315789473684211
Recall at Best F1 : 0.631578947368421
Best F1 Score : 0.11483253588516747
```



In [70]:

```
test_xgb_proba = xgb.predict_proba(X_test)
y_test_prob_xgb = test_xgb_proba[:, 1]
y_test_prob_xgb
```

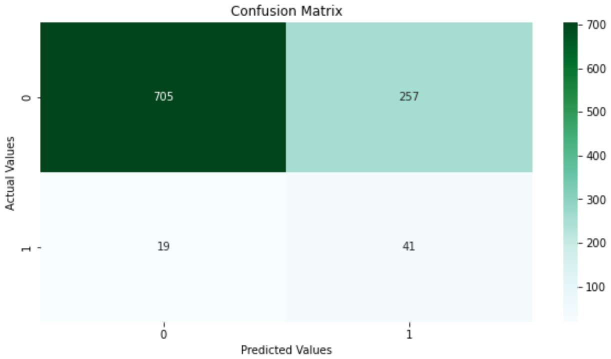
Out[70]:

```
array([0.00295475, 0.00448489, 0.00062596, ..., 0.0042313 , 0.32490686,
       0.09692448], dtype=float32)
```

In [71]:

```
# Test set peformance: Best threshold  F1

xgb_performance(y_test, y_test_prob_xgb)
```

```
Accuracy: 0.730
Precision at Best F1 : 0.13758389261744966
Recall at Best F1 : 0.6833333333333333
Best F1 Score : 0.22905027932960892
```



In [75]:

```
fpr_lor , tpr_lor, threshold_lor = roc_curve(y_test, y_test_prob_lor)

fpr_dtc , tpr_dtc, threshold_dtc = roc_curve(y_test, y_test_prob_dtc)

fpr_rfc , tpr_rfc, threshold_rfc = roc_curve(y_test, y_test_prob_rfc)

fpr_xgb , tpr_xgb, threshold_xgb = roc_curve(y_test, y_test_prob_xgb)

lor_AUC = roc_auc_score(y_test, y_test_prob_lor)
dtc_AUC = roc_auc_score(y_test, y_test_prob_dtc)
rfc_AUC = roc_auc_score(y_test, y_test_prob_rfc)
xgb_AUC = roc_auc_score(y_test, y_test_prob_xgb)
```

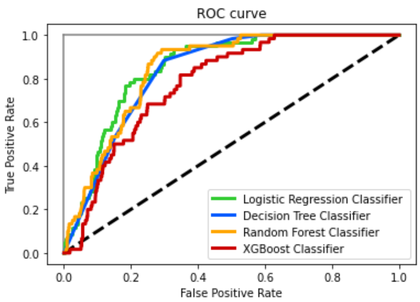
In [76]:

```
print('AUC for Logistic Regression: ' , lor_AUC)
print('AUC for Decision Tree: ' , dtc_AUC)
print('AUC for Random Forest: ' , rfc_AUC)
print('AUC for XGBoost: ' , xgb_AUC)
```

```
AUC for Logistic Regression:  0.8507796257796258
AUC for Decision Tree:  0.8311850311850312
AUC for Random Forest:  0.8430959805959806
AUC for XGBoost:  0.7785343035343035
```

In [78]:

```
plt.plot([0,1], ls = '--', linewidth=3, color = 'black')
plt.plot([0,0],[1,0], c='.5')
plt.plot([1,1],c='.5')
plt.plot(fpr_lor, tpr_lor, linewidth=3, color = '#33CC33', label= "Logistic Regression Classifier ")
plt.plot(fpr_dtc, tpr_dtc, linewidth=3, color = '#005aff', label= "Decision Tree Classifier")
plt.plot(fpr_rfc, tpr_rfc, linewidth=3, color = '#ffa500', label= "Random Forest Classifier")
plt.plot(fpr_xgb, tpr_xgb, linewidth=3, color = '#cc0000', label= "XGBoost Classifier")
plt.legend()
plt.xlabel("False Positive Rate")
plt.ylabel("True Positive Rate")
plt.title('ROC curve')
plt.show()
```



Conclusion

Return Contents

- For classification problems that have a severe class imbalance, the default threshold can result in poor performance. As such, a simple spproach to improving the performance of a classifier that predicts probabilities on an imbalanced classification problem is to tune the threshold used to map probabilities to class labels