RUTUJA DOIPHODE

#PROJECT NAME: Stroke Prediction Visualization and Model Making Email: rutujadoiphode2000@gmail.com (mailto:rutujadoiphode2000@gmail.com) Linkedin: https://www.linkedin.com/in/rutujadoiphode/ (https://www.linkedin.com/in/rutujadoiphode/)

In [1]:
```
import pandas as pd
import numpy as np
import seaborn as sns
import tensorflow as tf
import matplotlib.pyplot as plt
import warnings
warnings.filterwarnings("ignore")

# Load the dataset
dataset = pd.read_csv("healthcare-dataset-stroke-data.csv")

dataset
```

Out[1]:

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

5110 rows × 12 columns

In [2]:
```
dataset.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 5110 entries, 0 to 5109
Data columns (total 12 columns):
 #   Column             Non-Null Count  Dtype  
---  ------             --------------  -----  
 0   id                 5110 non-null   int64  
 1   gender             5110 non-null   object 
 2   age                5110 non-null   float64
 3   hypertension       5110 non-null   int64  
 4   heart_disease      5110 non-null   int64  
 5   ever_married       5110 non-null   object 
 6   work_type          5110 non-null   object 
 7   Residence_type     5110 non-null   object 
 8   avg_glucose_level  5110 non-null   float64
 9   bmi                4909 non-null   float64
 10  smoking_status     5110 non-null   object 
 11  stroke             5110 non-null   int64  
dtypes: float64(3), int64(4), object(5)
memory usage: 479.2+ KB
```

In [3]:
```
dataset.isnull().sum()
```

Out[3]:
```
id                     0
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 [4]:
```
dataset.bmi.replace(to_replace=np.nan, value=dataset.bmi.mean(), inplace=True)
```

In [5]:
```
dataset.isnull().sum()
```

Out[5]:
```
id                   0
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
```

In [6]:
```
dataset.describe()
```

Out[6]:

	id	age	hypertension	heart_disease	avg_glucose_level	bmi	stroke
count	5110.000000	5110.000000	5110.000000	5110.000000	5110.000000	5110.000000	5110.000000
mean	36517.829354	43.226614	0.097456	0.054012	106.147677	28.893237	0.048728
std	21161.721625	22.612647	0.296607	0.226063	45.283560	7.698018	0.215320
min	67.000000	0.080000	0.000000	0.000000	55.120000	10.300000	0.000000
25%	17741.250000	25.000000	0.000000	0.000000	77.245000	23.800000	0.000000
50%	36932.000000	45.000000	0.000000	0.000000	91.885000	28.400000	0.000000
75%	54682.000000	61.000000	0.000000	0.000000	114.090000	32.800000	0.000000
max	72940.000000	82.000000	1.000000	1.000000	271.740000	97.600000	1.000000

In [7]:
```
dataset.corr()
```

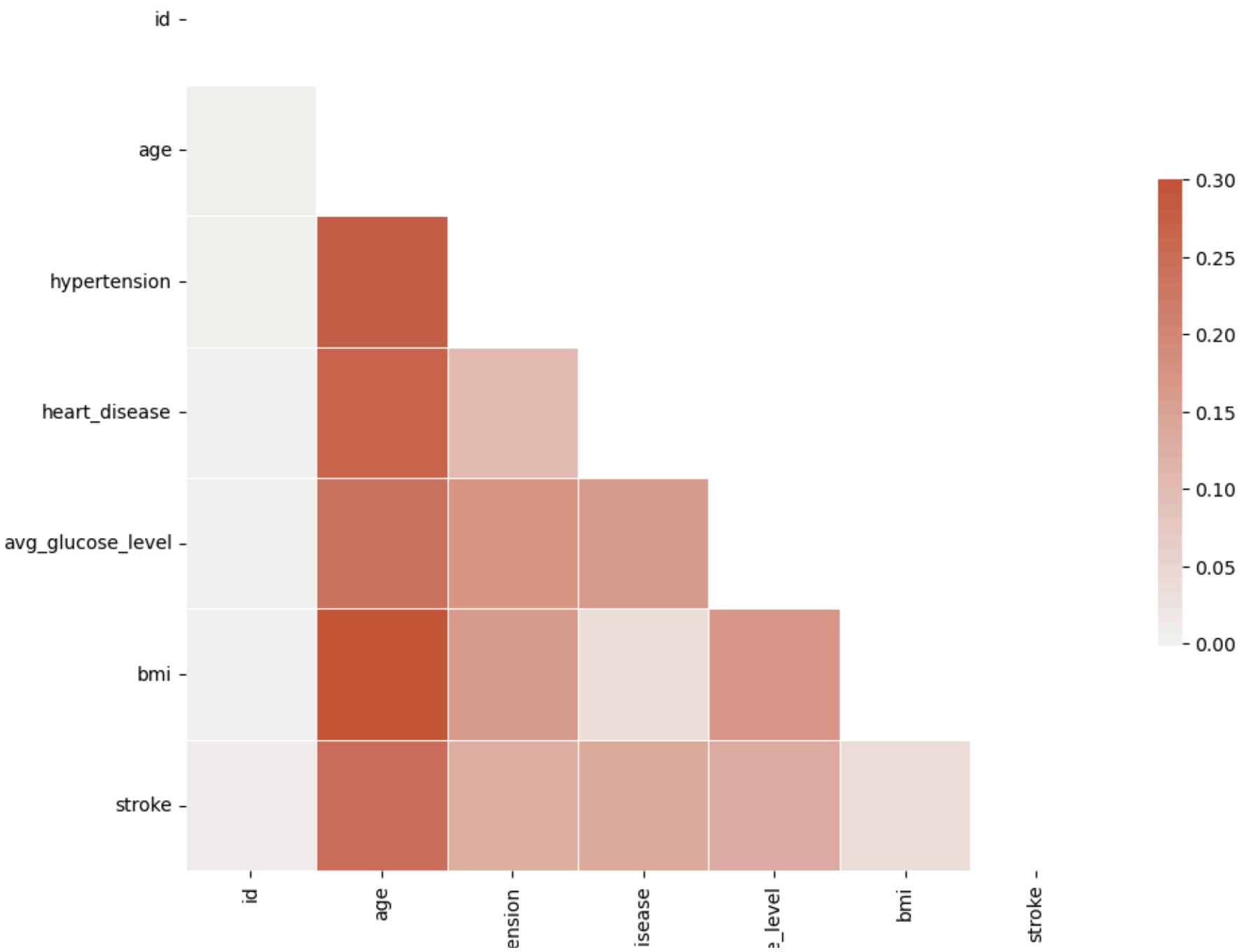
Out[7]:

	id	age	hypertension	heart_disease	avg_glucose_level	bmi	stroke
id	1.000000	0.003538	0.003550	-0.001296	0.001092	0.002999	0.006388
age	0.003538	1.000000	0.276398	0.263796	0.238171	0.325942	0.245257
hypertension	0.003550	0.276398	1.000000	0.108306	0.174474	0.160189	0.127904
heart_disease	-0.001296	0.263796	0.108306	1.000000	0.161857	0.038899	0.134914
avg_glucose_level	0.001092	0.238171	0.174474	0.161857	1.000000	0.168751	0.131945
bmi	0.002999	0.325942	0.160189	0.038899	0.168751	1.000000	0.038947
stroke	0.006388	0.245257	0.127904	0.134914	0.131945	0.038947	1.000000

Data Visualization

In [8]:
```
# Compute the correlation matrix
corr = dataset.corr()

# Generate a mask for the upper triangle
mask = np.triu(np.ones_like(corr, dtype=bool))

# Set up the matplotlib figure
f, ax = plt.subplots(figsize=(11, 9))

# Generate a custom diverging colormap
cmap = sns.diverging_palette(230, 20, as_cmap=True)

# Draw the heatmap with the mask and correct aspect ratio
sns.heatmap(corr, mask=mask, cmap=cmap, vmax=.3, center=0, square=True, linewidths=.5, cbar_kws={"shrink": .5})
```

Out[8]:
```
<Axes: >
```


id
age
hypertension
heart_disease
avg_glucose_level
bmi
stroke
0.30
0.25
0.20
0.15
0.10
0.05
0.00

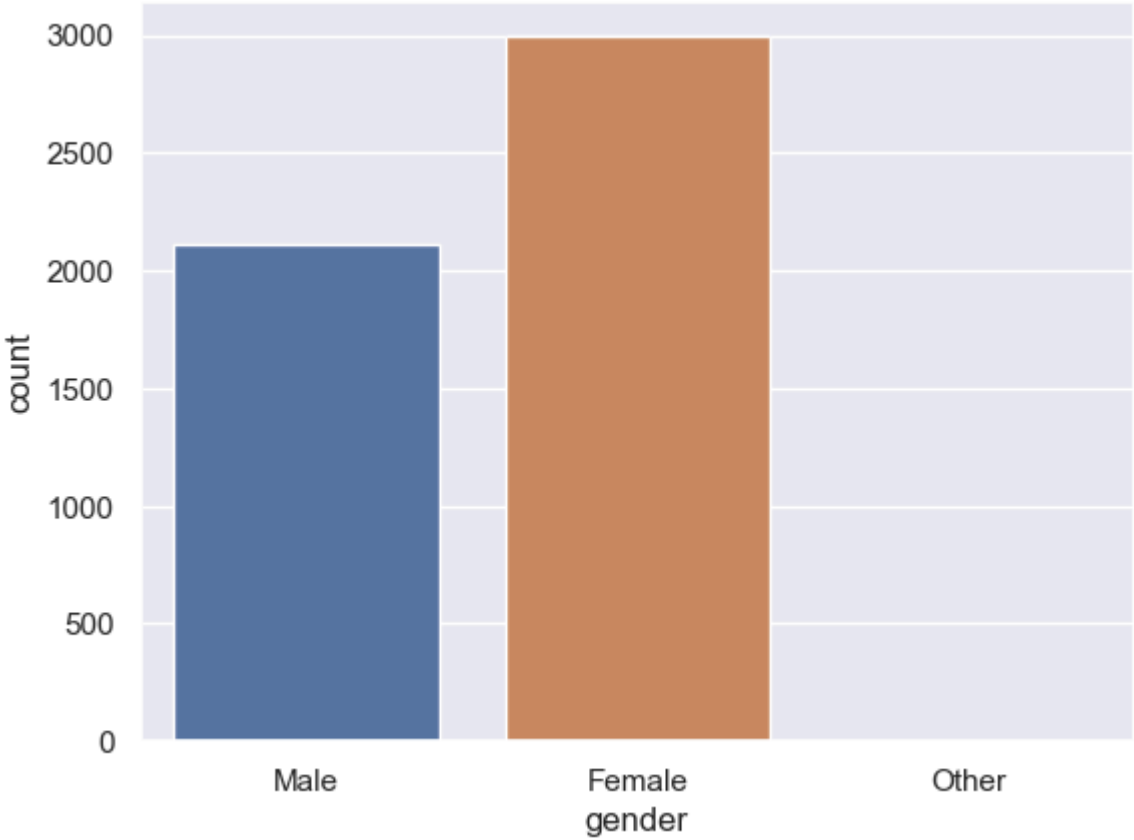
hypert

heart_d

avg_glucose

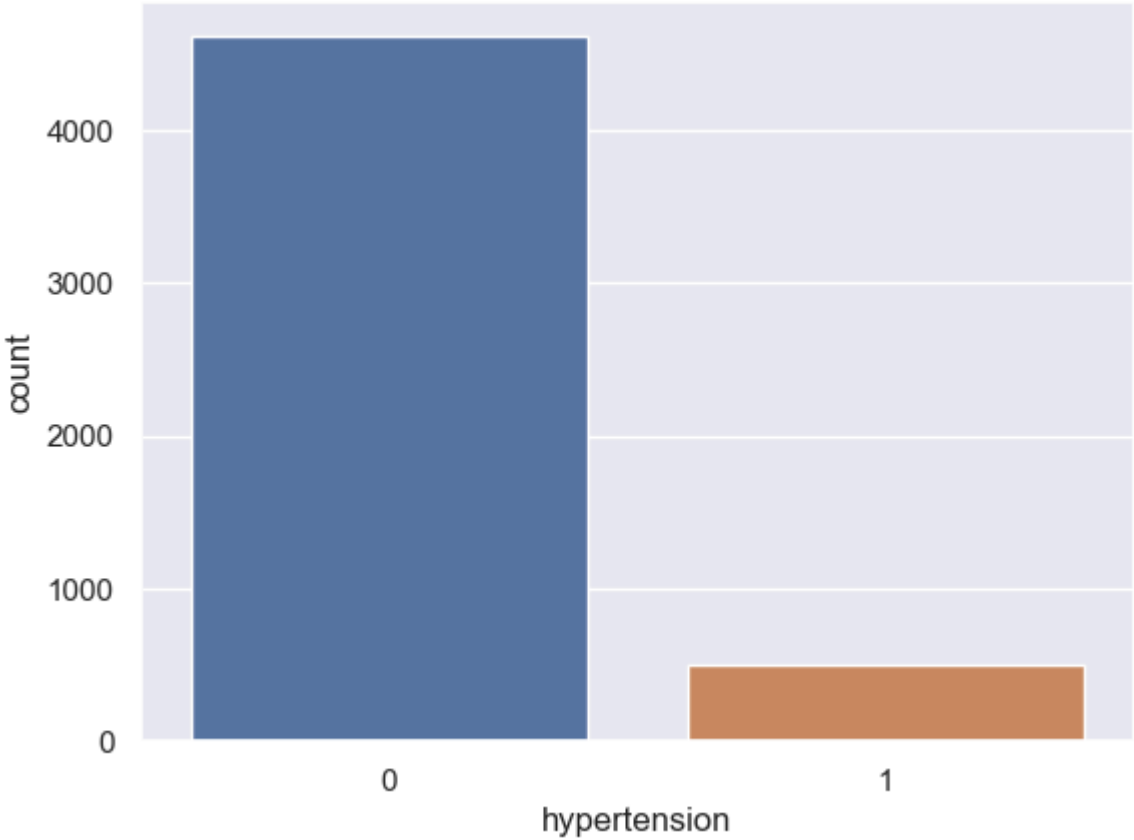
In [9]:
```
#GENDER

print(dataset.gender.value_counts())
sns.set_theme(style="darkgrid")
ax = sns.countplot(data=dataset, x="gender")
plt.show()
```

```
Female    2994
Male      2115
Other        1
Name: gender, dtype: int64
```



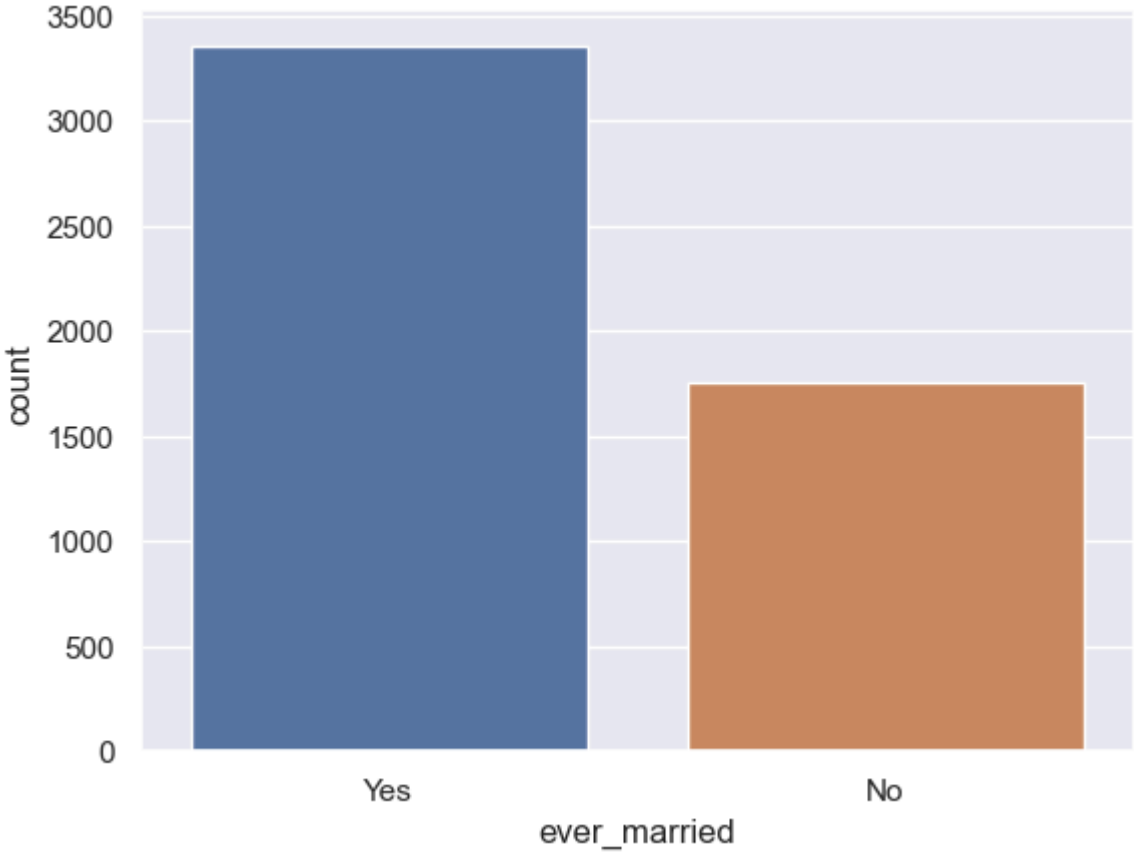
In [10]:
```
#HypterTension

print(dataset.hypertension.value_counts())
sns.set_theme(style="darkgrid")
ax = sns.countplot(data=dataset, x="hypertension")
plt.show()
```

```
0    4612
1     498
Name: hypertension, dtype: int64
```



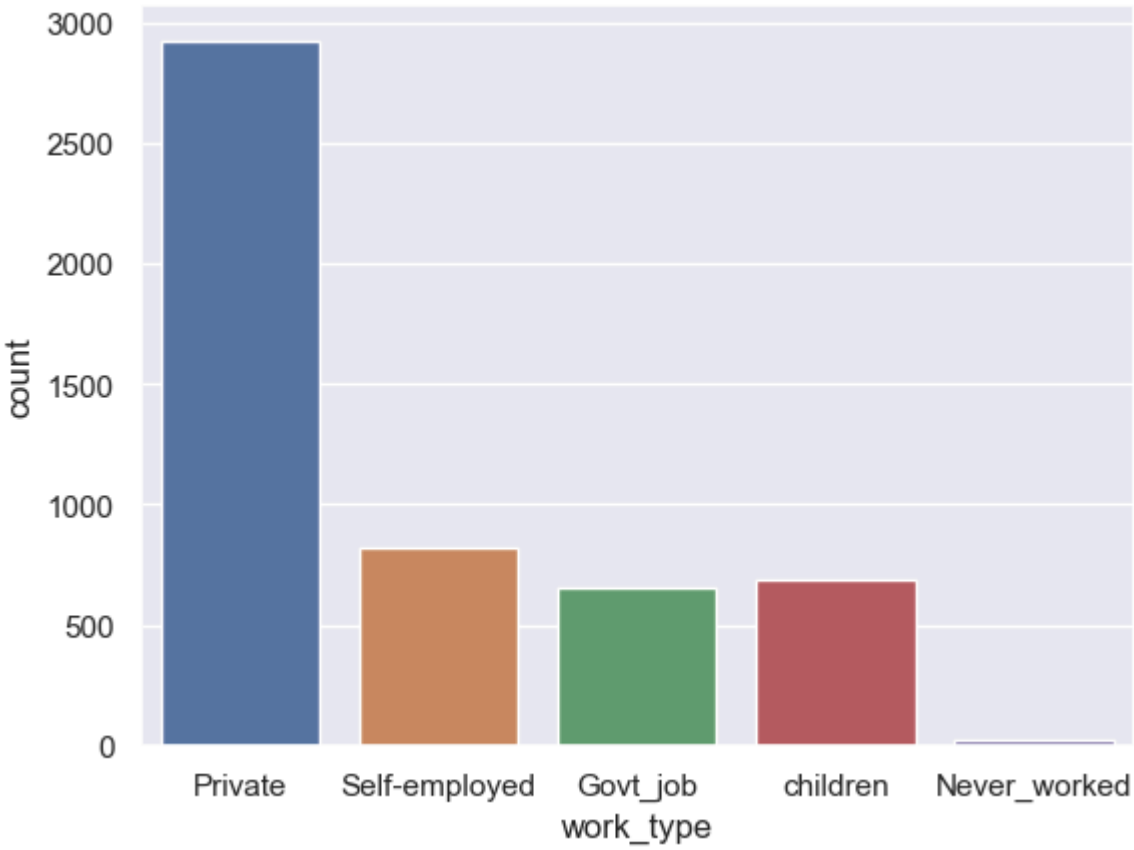
In [11]:
```
#Marraige Status

print(dataset.ever_married.value_counts())
sns.set_theme(style="darkgrid")
ax = sns.countplot(data=dataset, x="ever_married")
plt.show()
```

```
Yes    3353
No     1757
Name: ever_married, dtype: int64
```



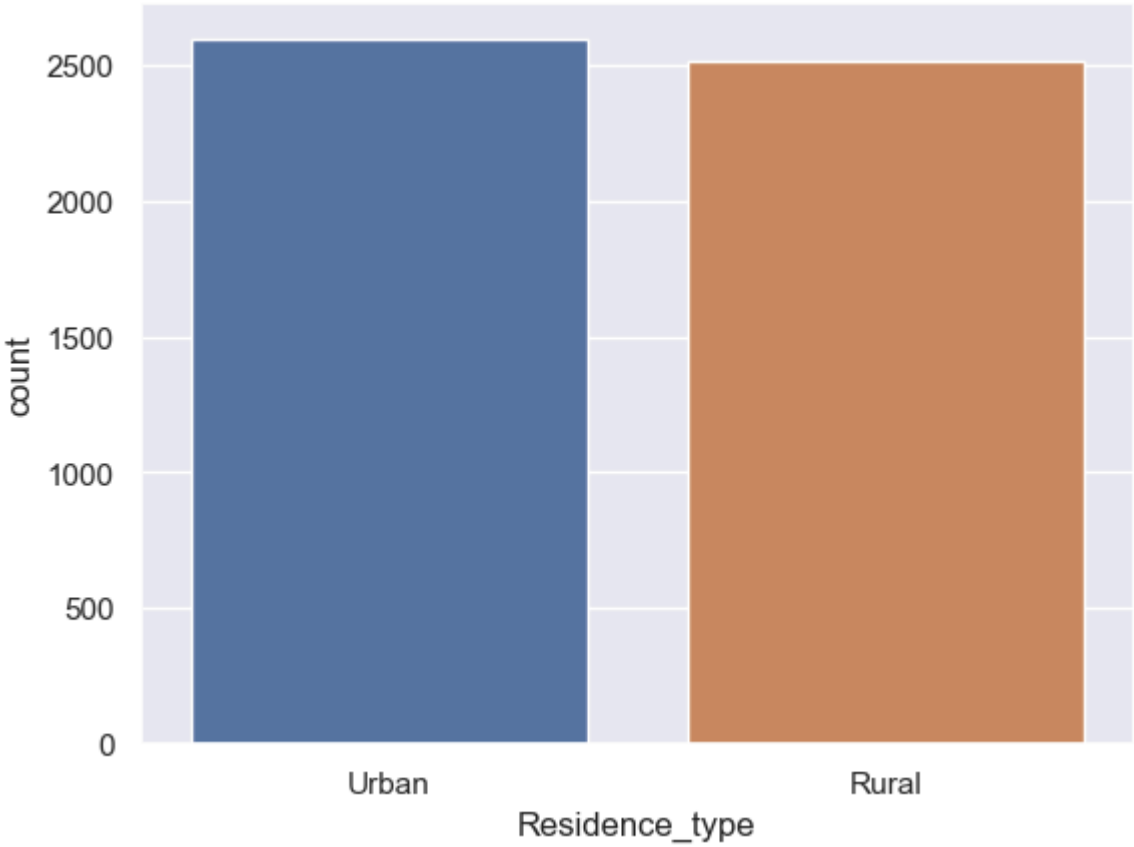
In [12]:
```
#Work Type

print(dataset.work_type.value_counts())
sns.set_theme(style="darkgrid")
ax = sns.countplot(data=dataset, x="work_type")
plt.show()
```

```
Private          2925
Self-employed     819
children          687
Govt_job          657
Never_worked       22
Name: work_type, dtype: int64
```



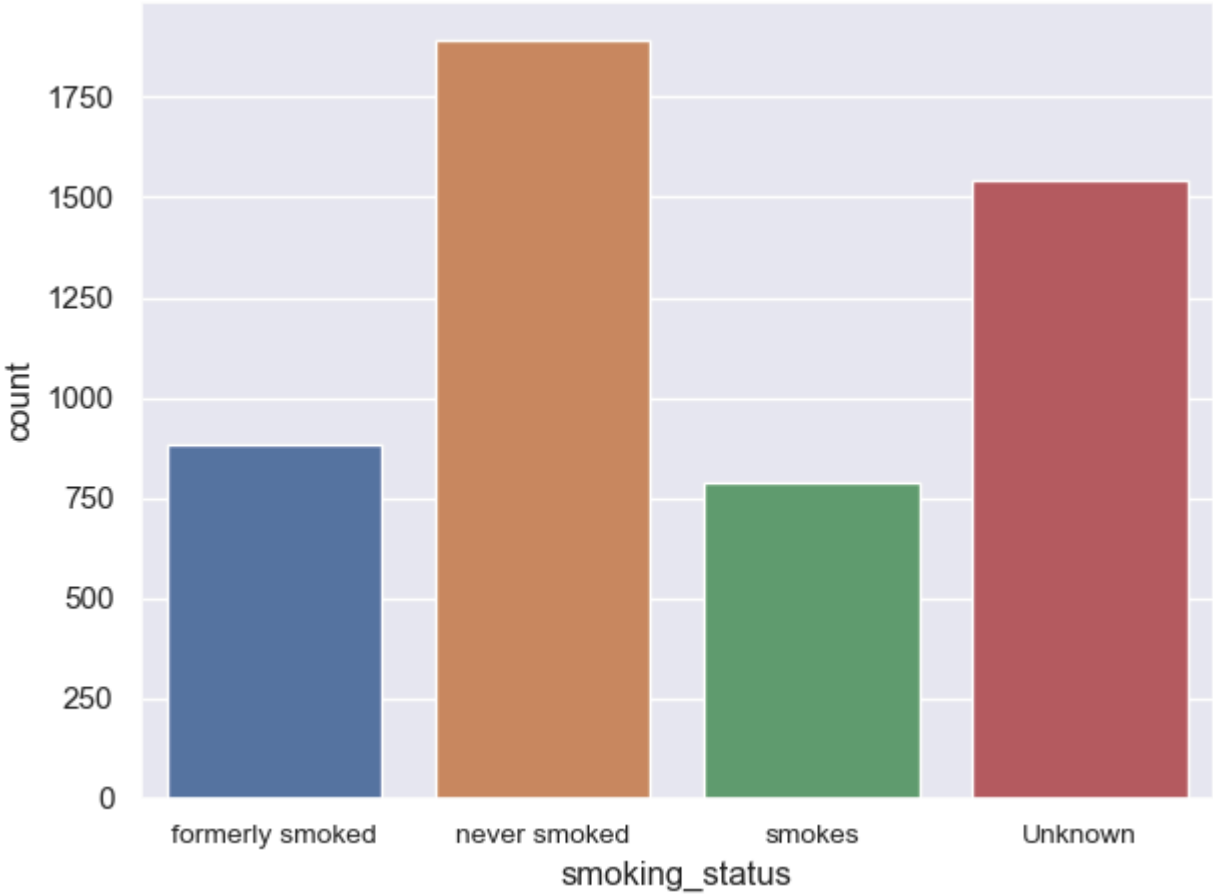
In [13]:
```
#Residence Type

print(dataset.Residence_type.value_counts())
sns.set_theme(style="darkgrid")
ax = sns.countplot(data=dataset, x="Residence_type")
plt.show()
```

```
Urban    2596
Rural    2514
Name: Residence_type, dtype: int64
```



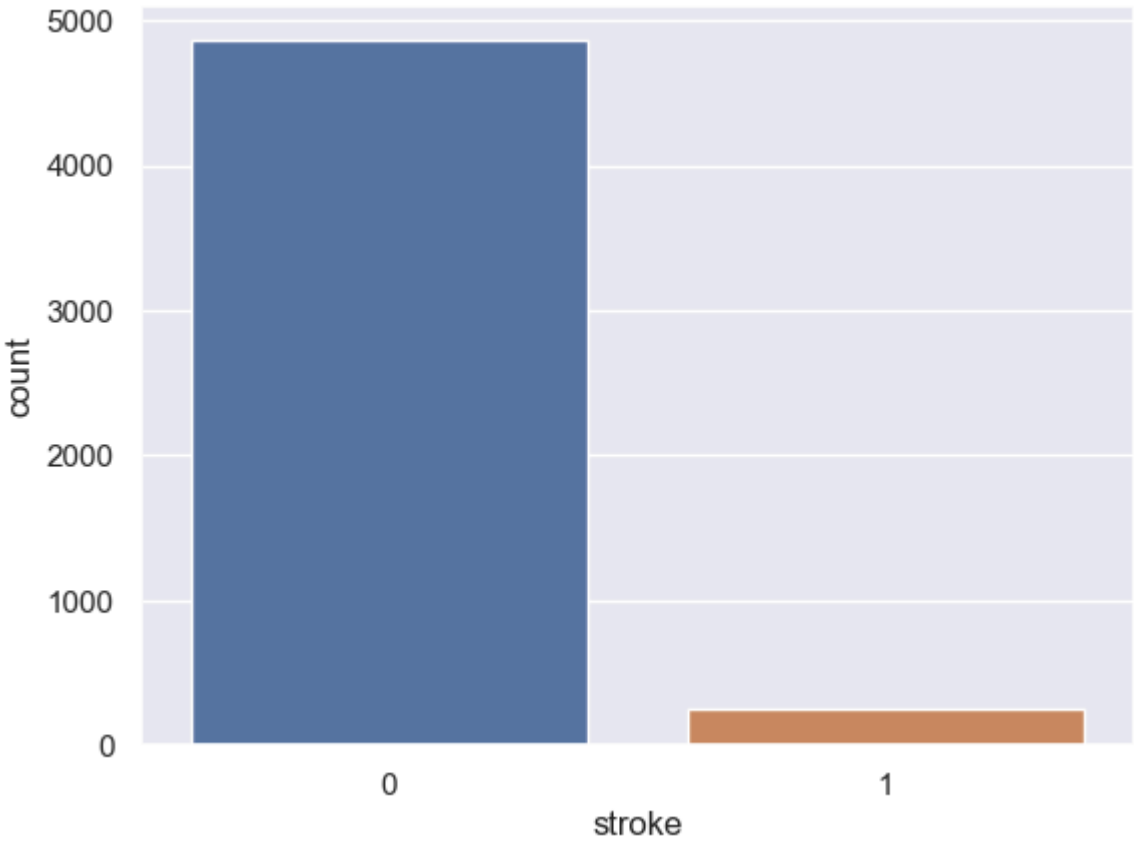
In [14]:
```
#Smoking Status

print(dataset.smoking_status.value_counts())
sns.set_theme(style="darkgrid")
ax = sns.countplot(data=dataset, x="smoking_status")
ax.set_xticklabels(ax.get_xticklabels(), fontsize=10)
plt.tight_layout()
plt.show()
```

```
never smoked       1892
Unknown            1544
formerly smoked     885
smokes              789
Name: smoking_status, dtype: int64
```


1750
1500
1250
1000
750
500
250
0
count
formerly smoked
never smoked
smokes
Unknown
smoking_status

In [15]:
```
#Stroke

print(dataset.stroke.value_counts())
sns.set_theme(style="darkgrid")
ax = sns.countplot(data=dataset, x="stroke")
plt.show()
```

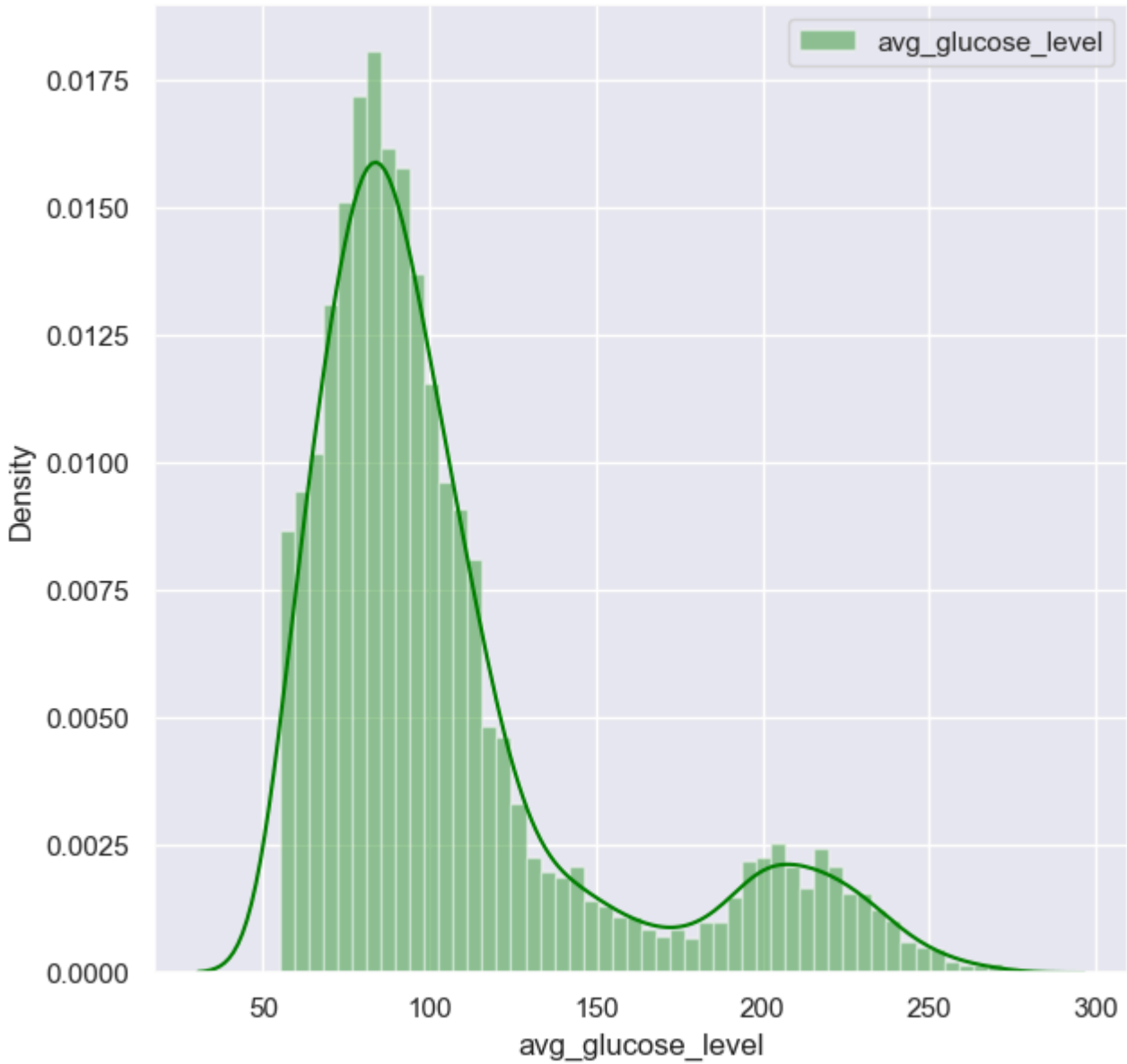
```
0    4861
1     249
Name: stroke, dtype: int64
```



Data Visualization - Distribution Plot

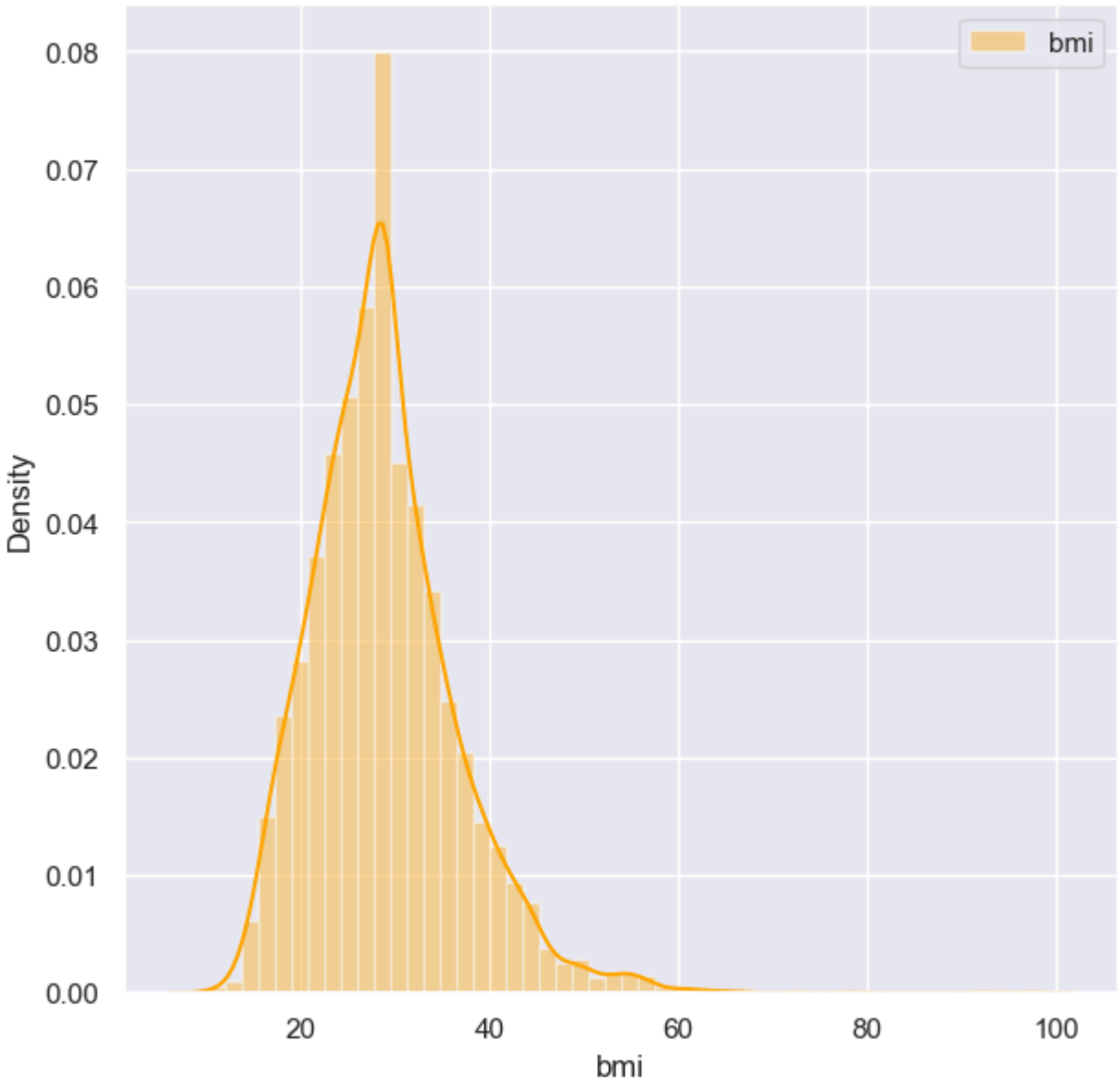
In [16]:
```
#AVG Glucose Level

fig = plt.figure(figsize=(7,7))
sns.distplot(dataset.avg_glucose_level, color="green", label="avg_glucose_level", kde= True)
plt.legend()
```

Out[16]:
```
<matplotlib.legend.Legend at 0x2395dad8710>
```

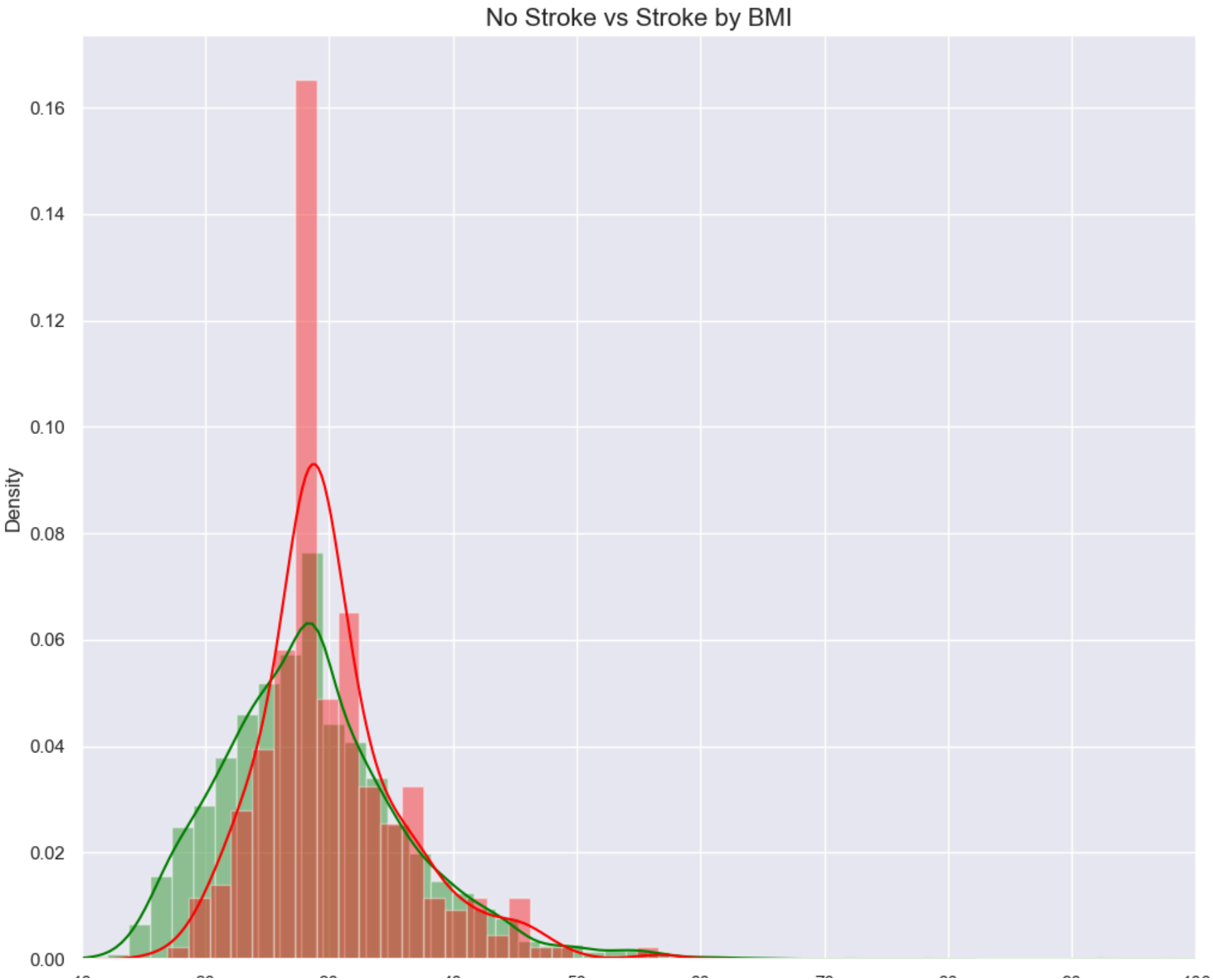

avg_glucose_level
0.0175
0.0150
0.0125
0.0100
0.0075
0.0050
0.0025
0.0000
Density
50
100
150
200
250
300
avg_glucose_level

In [17]:
```
#BMI Disitribution

fig = plt.figure(figsize=(7,7))
sns.distplot(dataset.bmi, color="orange", label="bmi", kde= True)
plt.legend()
```

Out[17]:
```
<matplotlib.legend.Legend at 0x2395db2ca50>
```

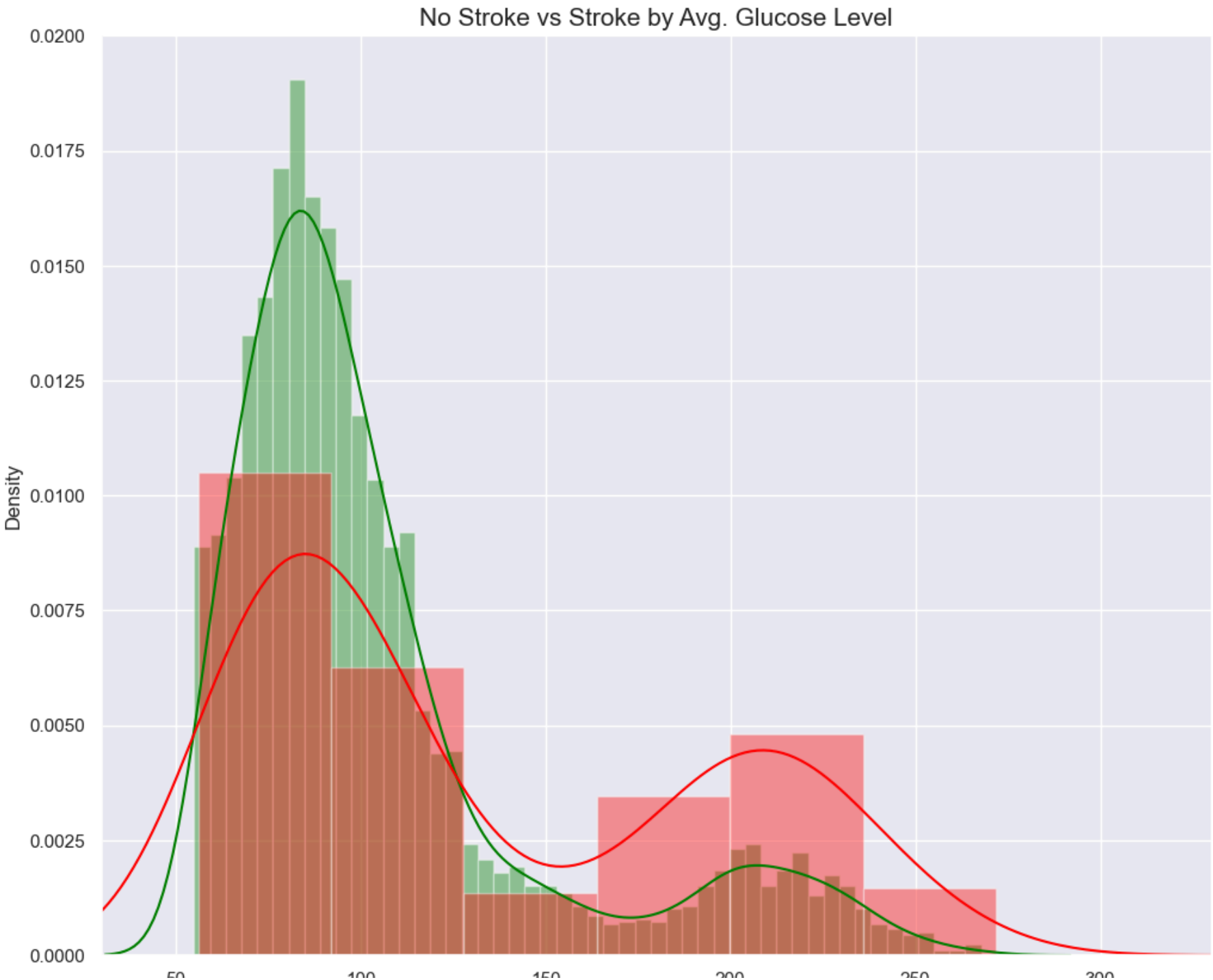

bmi
0.08
0.07
0.06
0.05
0.04
0.03
0.02
0.01
0.00
Density
20
40
60
80
100
bmi

In [18]:
```
# Stroke vs No Stroke by BMI

plt.figure(figsize=(12,10))

sns.distplot(dataset[dataset['stroke'] == 0]["bmi"], color='green') # No Stroke - green
sns.distplot(dataset[dataset['stroke'] == 1]["bmi"], color='red') # Stroke - Red

plt.title('No Stroke vs Stroke by BMI', fontsize=15)
plt.xlim([10,100])
plt.show()
```


No Stroke vs Stroke by BMI
Density
0.16
0.14
0.12
0.10
0.08
0.06
0.04
0.02
0.00

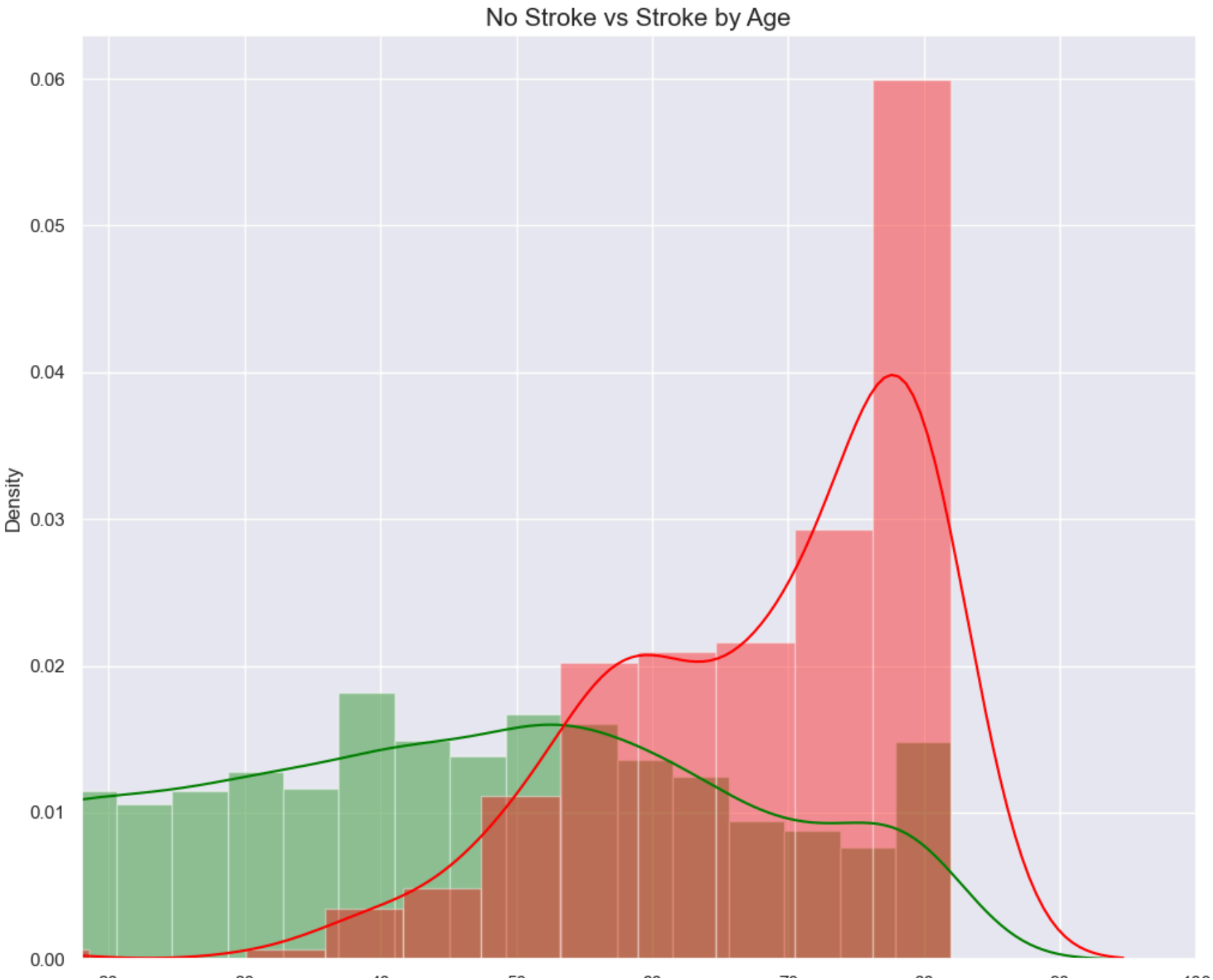
The graph reveals that the density of overweight people who suffered a stroke is more.

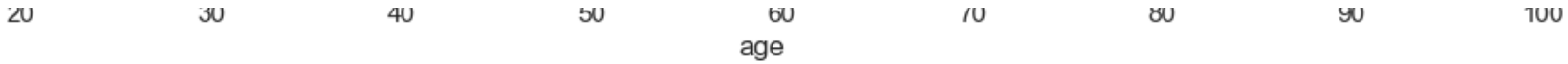
In [19]:
```
# Stroke vs No Stroke by AVG Glucose Level

plt.figure(figsize=(12,10))

sns.distplot(dataset[dataset['stroke'] == 0]["avg_glucose_level"], color='green') # No Stroke - green
sns.distplot(dataset[dataset['stroke'] == 1]["avg_glucose_level"], color='red') # Stroke - Red

plt.title('No Stroke vs Stroke by Avg. Glucose Level', fontsize=15)
plt.xlim([30,330])
plt.show()
```


No Stroke vs Stroke by Avg. Glucose Level
Density
0.0200
0.0175
0.0150
0.0125
0.0100
0.0075
0.0050
0.0025
0.0000

The graph reveals there is a higher occurrence of strokes among individuals with glucose levels below 100.

In [20]:
```
#Stroke vs No Stroke by Age

plt.figure(figsize=(12,10))

sns.distplot(dataset[dataset['stroke'] == 0]["age"], color='green') # No Stroke - green
sns.distplot(dataset[dataset['stroke'] == 1]["age"], color='red') # Stroke - Red

plt.title('No Stroke vs Stroke by Age', fontsize=15)
plt.xlim([18,100])
plt.show()
```

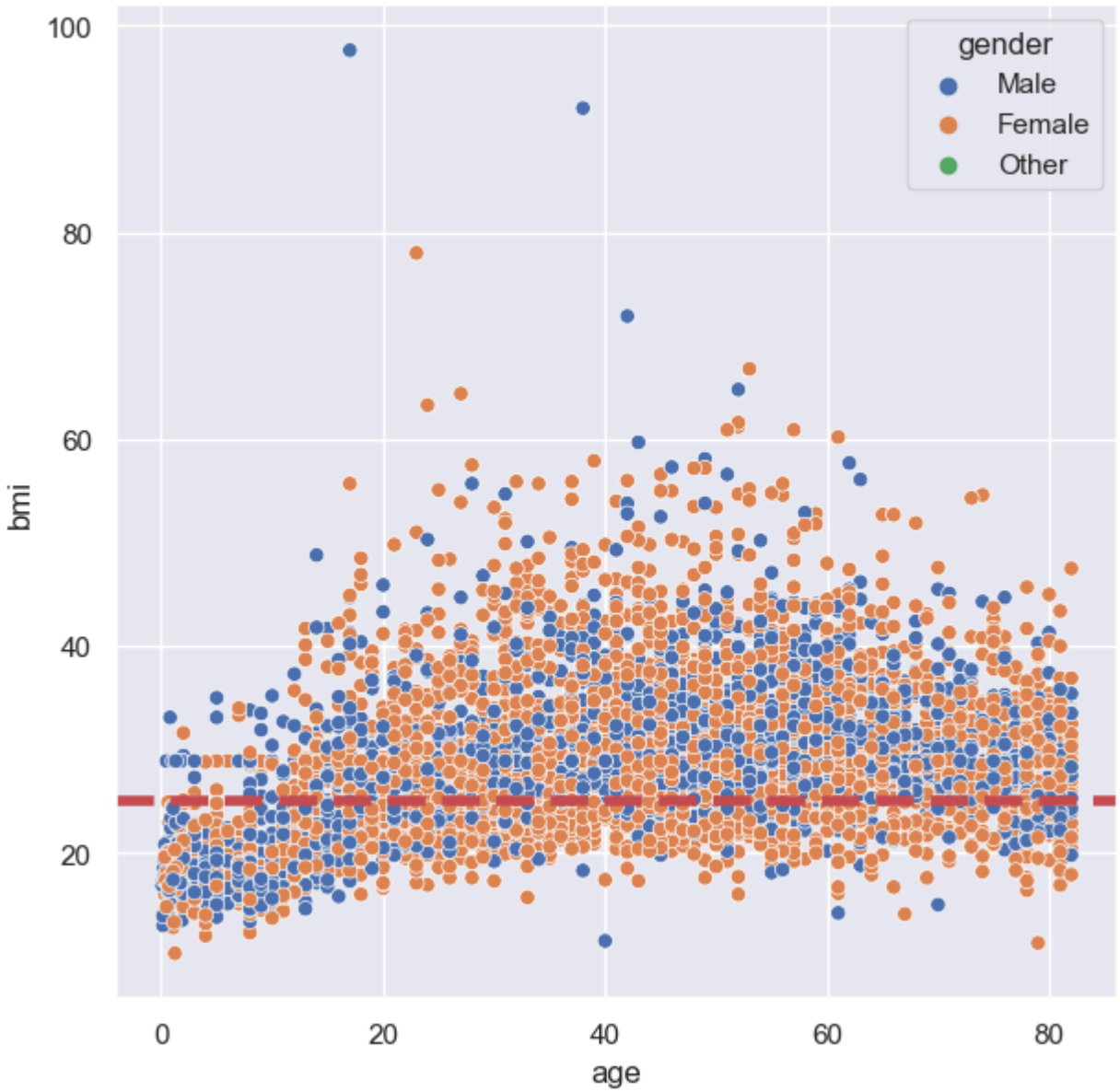

No Stroke vs Stroke by Age
Density
0.06
0.05
0.04
0.03
0.02
0.01
0.00



The graph reveals that individuals over the age of 50 experienced a higher incidence of strokes

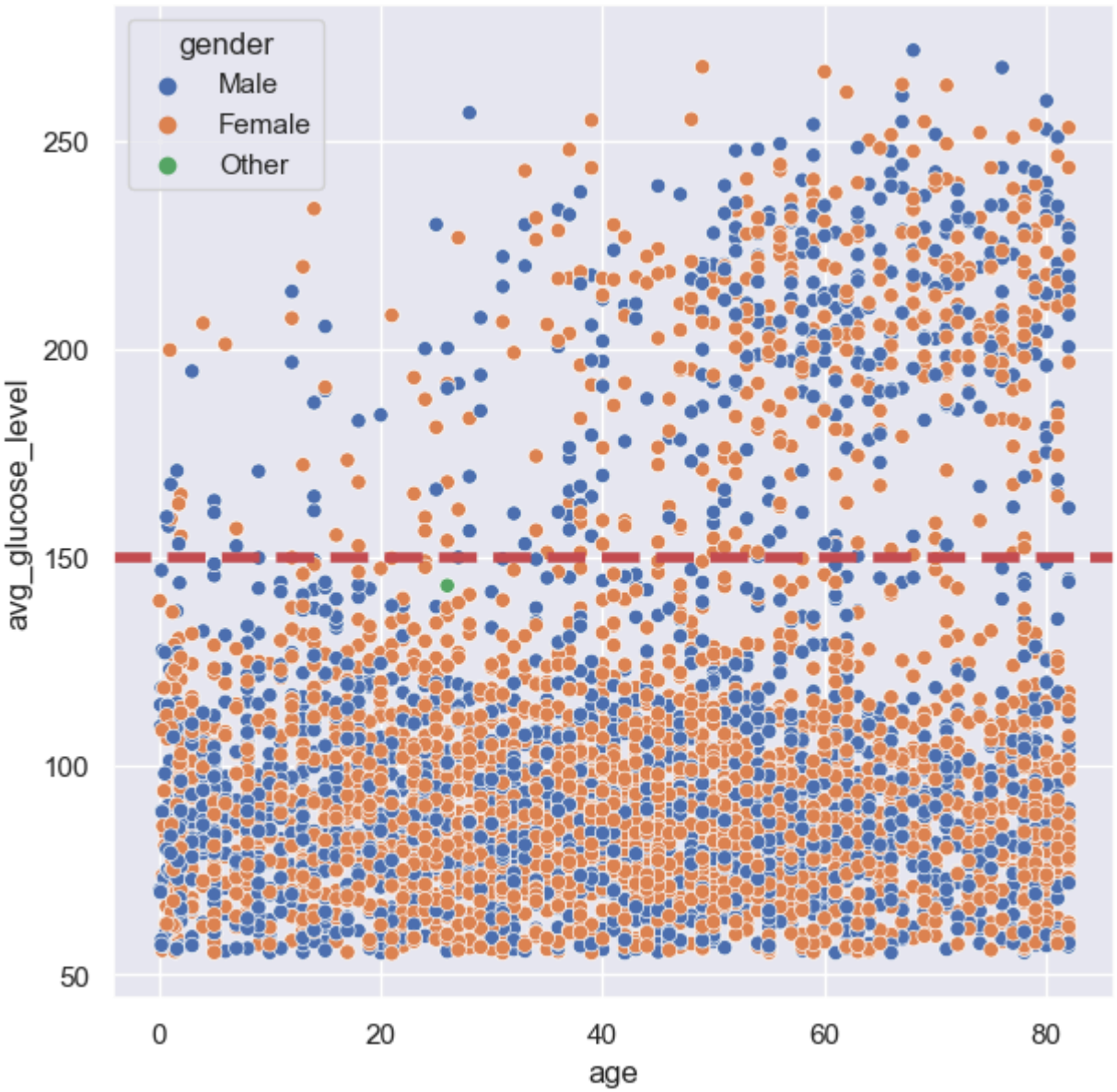
Scatter Plot

In [21]:
```
#BMI vs Age

fig = plt.figure(figsize=(7,7))
graph = sns.scatterplot(data=dataset, x="age", y="bmi", hue='gender')
graph.axhline(y= 25, linewidth=4, color='r', linestyle= '--')
plt.show()
```



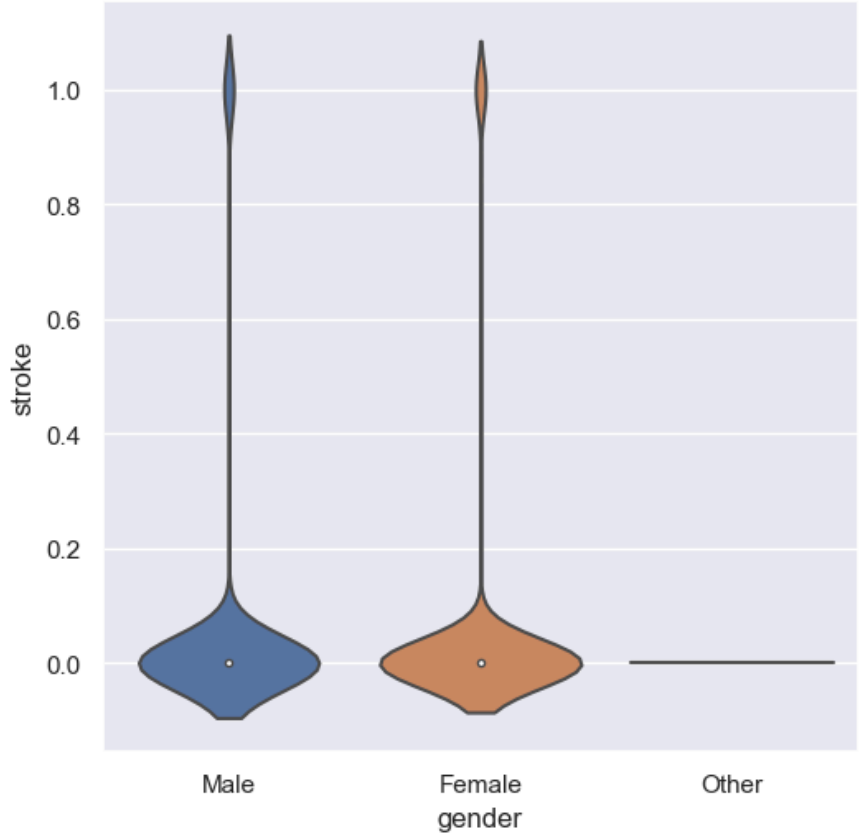
The graph reveals that a significant number of individuals with a BMI exceeding 25 fall into the categories of overweight and obese.

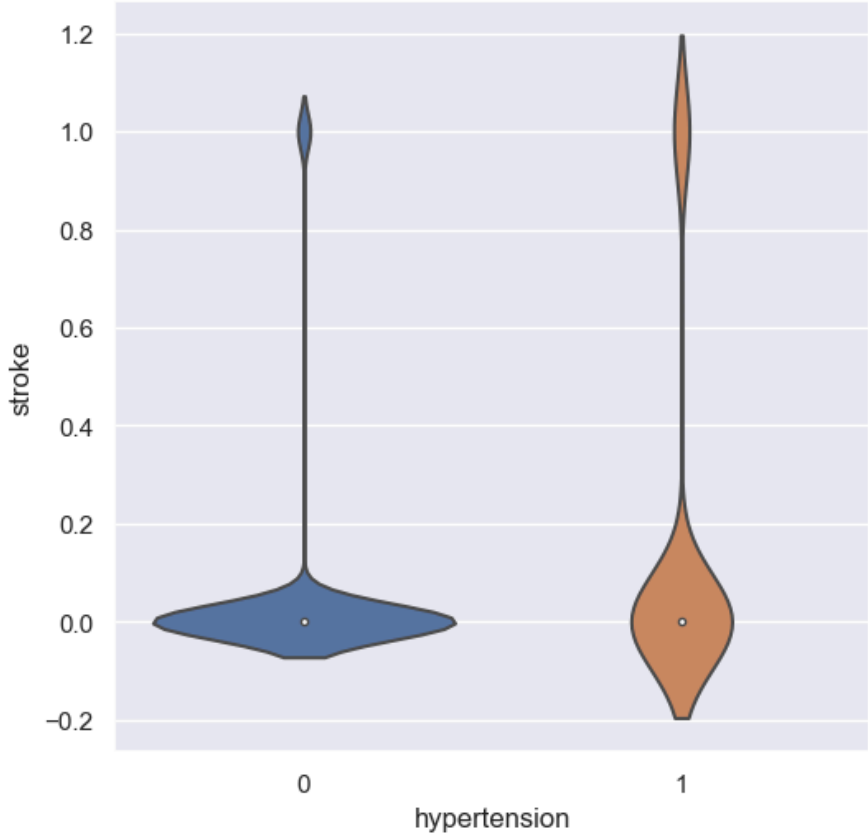
In [22]:
```
# Avg Glucose vs Age

fig = plt.figure(figsize=(7,7))
graph = sns.scatterplot(data=dataset, x="age", y="avg_glucose_level", hue='gender')
graph.axhline(y= 150, linewidth=4, color='r', linestyle= '--')
plt.show()
```

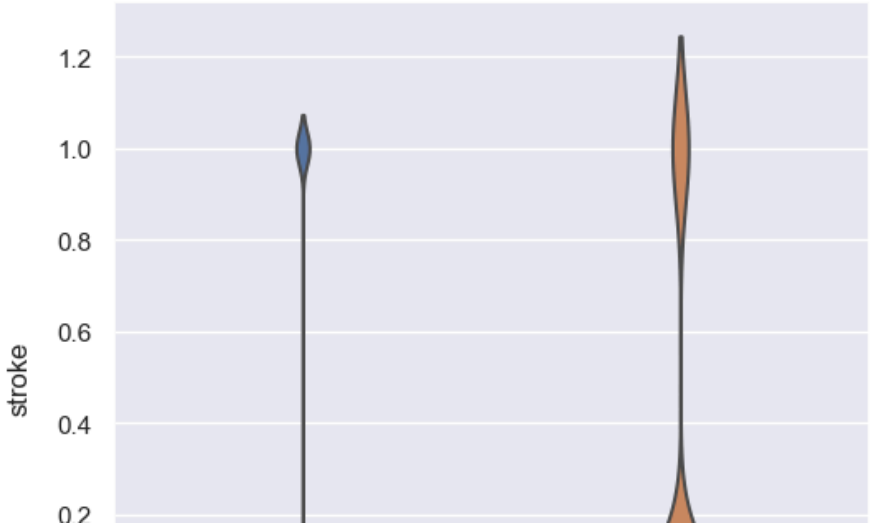


The graph reveals apparently that there are fewer individuals with glucose levels above 150 compared to those below it. Consequently, it could be inferred that individuals above 150 might be more likely to have diabetes.

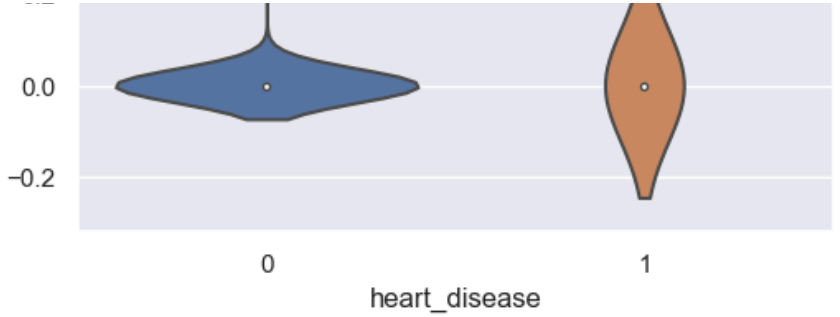
In [23]:
```
#Violin Plot

import matplotlib.pyplot as plt
import seaborn as sns

plt.figure(figsize=(13,13))
sns.set_theme(style="darkgrid")

plt.subplot(2, 2, 1)
sns.violinplot(x='gender', y='stroke', data=dataset)

plt.subplot(2, 2, 2)
sns.violinplot(x='hypertension', y='stroke', data=dataset)

plt.subplot(2, 2, 3)
sns.violinplot(x='heart_disease', y='stroke', data=dataset)

plt.subplot(2, 2, 4)
sns.violinplot(x='ever_married', y='stroke', data=dataset)

plt.xticks(fontsize=9, rotation=45)
plt.show()

plt.figure(figsize=(13,13))
sns.set_theme(style="darkgrid")

plt.subplot(2, 2, 1)
sns.violinplot(x='work_type', y='stroke', data=dataset)

plt.subplot(2, 2, 2)
sns.violinplot(x='Residence_type', y='stroke', data=dataset)

plt.show()
```

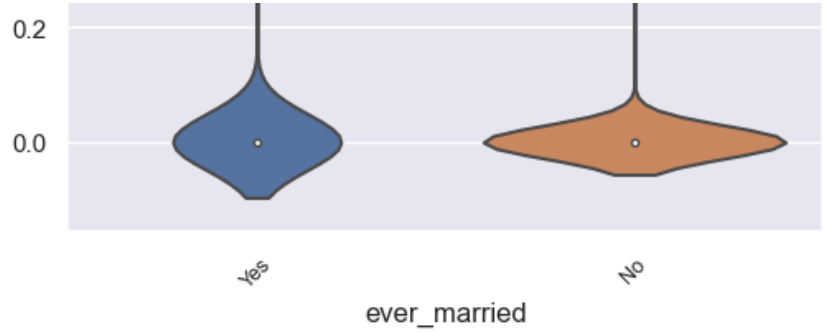

1.0
0.8
0.6
0.4
0.2
0.0
stroke
Male
Female
Other
gender

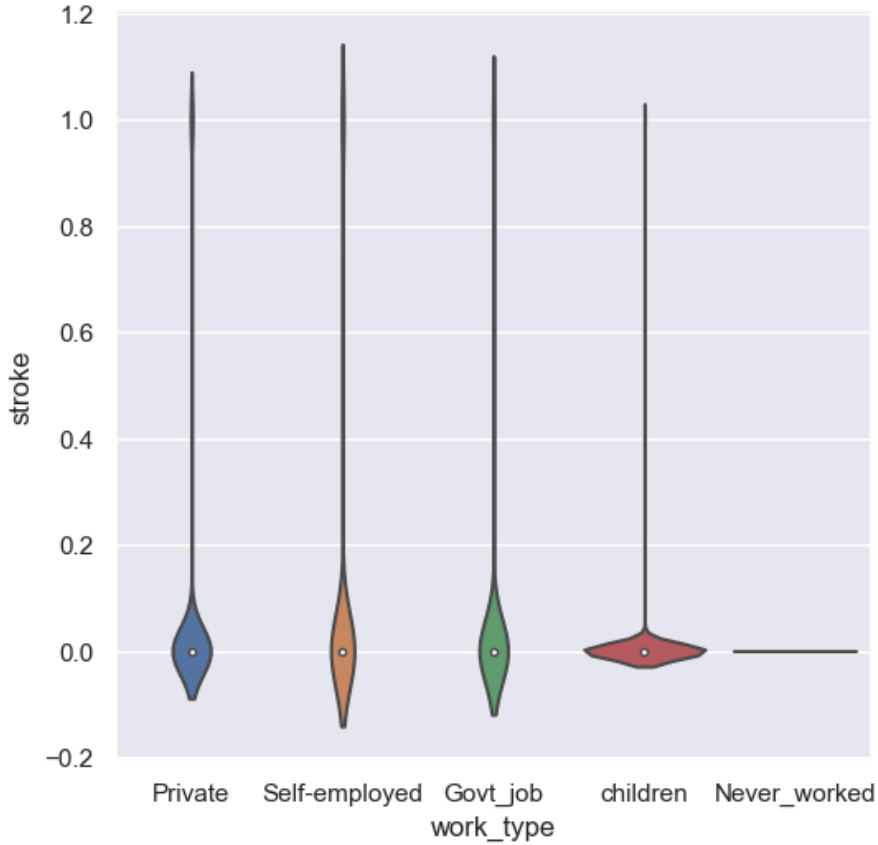

1.2
1.0
0.8
0.6
0.4
0.2
0.0
−0.2
stroke
0
1
hypertension

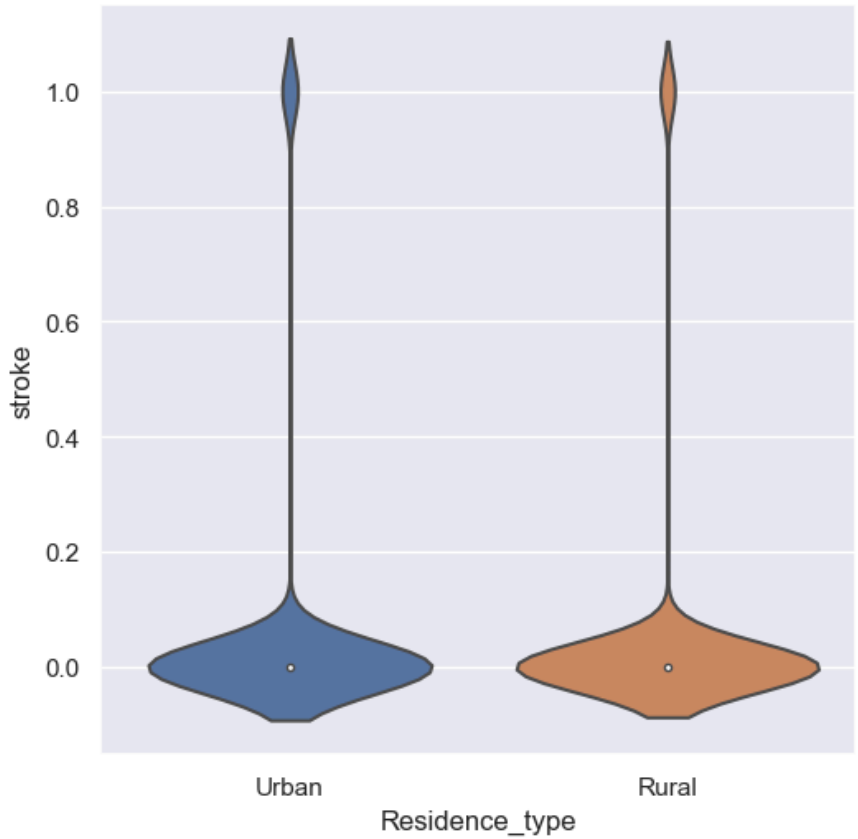

1.2
1.0
0.8
0.6
0.4
0.2
stroke


1.0
0.8
0.6
0.4
stroke


0.0
−0.2
0
1
heart_disease


0.2
0.0
Yes
No
ever_married


1.2
1.0
0.8
0.6
0.4
0.2
0.0
−0.2
stroke
Private
Self-employed
Govt_job
children
Never_worked
work_type


1.0
0.8
0.6
0.4
0.2
0.0
stroke
Urban
Rural
Residence_type

Data Processing

In [24]:
```
#Extraction & Sepereation

x = dataset.iloc[:, 1:-1].values
y = dataset.iloc[:, -1].values
```

In [25]:
```
x
```

Out[25]:
```
array([['Male', 67.0, 0, ..., 228.69, 36.6, 'formerly smoked'],
       ['Female', 61.0, 0, ..., 202.21, 28.893236911794666,
        'never smoked'],
       ['Male', 80.0, 0, ..., 105.92, 32.5, 'never smoked'],
       ...,
       ['Female', 35.0, 0, ..., 82.99, 30.6, 'never smoked'],
       ['Male', 51.0, 0, ..., 166.29, 25.6, 'formerly smoked'],
       ['Female', 44.0, 0, ..., 85.28, 26.2, 'Unknown']], dtype=object)
```

In [26]:
```
y
```

Out[26]:
```
array([1, 1, 1, ..., 0, 0, 0], dtype=int64)
```

In [27]:
```
#Encoding - OnehotEncoder

from sklearn.compose import ColumnTransformer
from sklearn.preprocessing import OneHotEncoder
ct = ColumnTransformer(transformers= [('encoder', OneHotEncoder(), [0,5,9])], remainder= 'passthrough')
x = np.array(ct.fit_transform(x))
```

In [28]:
```
x
```

Out[28]:
```
array([[0.0, 1.0, 0.0, ..., 'Urban', 228.69, 36.6],
       [1.0, 0.0, 0.0, ..., 'Rural', 202.21, 28.893236911794666],
       [0.0, 1.0, 0.0, ..., 'Rural', 105.92, 32.5],
       ...,
       [1.0, 0.0, 0.0, ..., 'Rural', 82.99, 30.6],
       [0.0, 1.0, 0.0, ..., 'Rural', 166.29, 25.6],
       [1.0, 0.0, 0.0, ..., 'Urban', 85.28, 26.2]], dtype=object)
```

Label Encoding

LabelEncoder(): 'ever_married' and 'residence_type'

In [29]:
```
from sklearn.preprocessing import LabelEncoder
le = LabelEncoder()
x[:, 15] = le.fit_transform(x[:, 15])
x[:, 16] = le.fit_transform(x[:, 16])
```

In [30]:
```
x
```

Out[30]:
```
array([[0.0, 1.0, 0.0, ..., 1, 228.69, 36.6],
       [1.0, 0.0, 0.0, ..., 0, 202.21, 28.893236911794666],
       [0.0, 1.0, 0.0, ..., 0, 105.92, 32.5],
       ...,
       [1.0, 0.0, 0.0, ..., 0, 82.99, 30.6],
       [0.0, 1.0, 0.0, ..., 0, 166.29, 25.6],
       [1.0, 0.0, 0.0, ..., 1, 85.28, 26.2]], dtype=object)
```

In [31]:
```
print('Shape of X: ', x.shape)
print('Shape of Y: ', y.shape)
```

```
Shape of X:  (5110, 19)
Shape of Y:  (5110,)
```

Splitting Dataset

In [32]:
```
from sklearn.model_selection import train_test_split
x_train, x_test, y_train, y_test = train_test_split(x, y, test_size= 0.2, random_state= 0)
```

In [33]:
```
print("Number transactions x_train dataset: ", x_train.shape)
print("Number transactions y_train dataset: ", y_train.shape)
print("Number transactions x_test dataset: ", x_test.shape)
print("Number transactions y_test dataset: ", y_test.shape)
```

```
Number transactions x_train dataset:  (4088, 19)
Number transactions y_train dataset:  (4088,)
Number transactions x_test dataset:  (1022, 19)
Number transactions y_test dataset:  (1022,)
```

Feature Scaling

StandardScaler is a method used for standardizing features within a dataset. It achieves this by subtracting the mean from each feature and then dividing by the standard deviation, effectively scaling the features to have a mean of 0 and a standard deviation of 1. This process ensures that the features have a uniform scale, which can be beneficial for certain machine learning algorithms, particularly those sensitive to the scale of features

In [34]:
```
from sklearn.preprocessing import StandardScaler
sc = StandardScaler()
x_train = sc.fit_transform(x_train)
x_test = sc.transform(x_test)
```

The number of people suffering stroke is 19:1, so it is very less. But, this also means that our dataset is imbalanced. So we use Synthetic Minority Oversampling Technique(SMOTE)

In [35]:
```
from imblearn.over_sampling import SMOTE
```

In [36]:
```
print("Before OverSampling, counts of label '1': {}".format(sum(y_train==1)))
print("Before OverSampling, counts of label '0': {} \n".format(sum(y_train==0)))

sm = SMOTE(random_state=2)
x_train_res, y_train_res = sm.fit_resample(x_train, y_train.ravel())

print('After OverSampling, the shape of train_X: {}'.format(x_train_res.shape))
print('After OverSampling, the shape of train_y: {} \n'.format(y_train_res.shape))

print("After OverSampling, counts of label '1': {}".format(sum(y_train_res==1)))
print("After OverSampling, counts of label '0': {}".format(sum(y_train_res==0)))
```

```
Before OverSampling, counts of label '1': 195
Before OverSampling, counts of label '0': 3893

After OverSampling, the shape of train_X: (7786, 19)
After OverSampling, the shape of train_y: (7786,)

After OverSampling, counts of label '1': 3893
After OverSampling, counts of label '0': 3893
```

Model Selection

In [37]:
```
from sklearn.linear_model import LogisticRegression
from sklearn.neighbors import KNeighborsClassifier
from sklearn.naive_bayes import GaussianNB
from sklearn.naive_bayes import BernoulliNB
from sklearn.svm import SVC
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import RandomForestClassifier
from xgboost import XGBClassifier
```

In [38]:
```
from sklearn.metrics import accuracy_score, confusion_matrix, roc_auc_score, ConfusionMatrixDisplay, precision_score,
from sklearn.model_selection import cross_val_score
```

In [39]:
```
models = []
models.append(['Logistic Regreesion', LogisticRegression(random_state=0)])
models.append(['SVM', SVC(random_state=0)])
models.append(['KNeighbors', KNeighborsClassifier()])
models.append(['GaussianNB', GaussianNB()])
models.append(['BernoulliNB', BernoulliNB()])
models.append(['Decision Tree', DecisionTreeClassifier(random_state=0)])
models.append(['Random Forest', RandomForestClassifier(random_state=0)])
models.append(['XGBoost', XGBClassifier(eval_metric= 'error')])

lst_1= []

for m in range(len(models)):
    lst_2= []
    model = models[m][1]
    model.fit(x_train_res, y_train_res)
    y_pred = model.predict(x_test)
    cm = confusion_matrix(y_test, y_pred)  #Confusion Matrix
    accuracies = cross_val_score(estimator = model, X = x_train_res, y = y_train_res, cv = 10)   #K-Fold Validation
    roc = roc_auc_score(y_test, y_pred)  #ROC AUC Score
    precision = precision_score(y_test, y_pred)  #Precision Score
    recall = recall_score(y_test, y_pred)  #Recall Score
    f1 = f1_score(y_test, y_pred)  #F1 Score
    print(models[m][0],':')
    print(cm)
    print('Accuracy Score: ',accuracy_score(y_test, y_pred))
    print('')
    print("K-Fold Validation Mean Accuracy: {:.2f} %".format(accuracies.mean()*100))
    print('')
    print("Standard Deviation: {:.2f} %".format(accuracies.std()*100))
    print('')
    print('ROC AUC Score: {:.2f}'.format(roc))
    print('')
    print('Precision: {:.2f}'.format(precision))
    print('')
    print('Recall: {:.2f}'.format(recall))
    print('')
    print('F1: {:.2f}'.format(f1))
    print('-----------------------------------')
    print('')
    lst_2.append(models[m][0])
```

```
    lst_2.append((accuracy_score(y_test, y_pred))*100)
    lst_2.append(accuracies.mean()*100)
    lst_2.append(accuracies.std()*100)
    lst_2.append(roc)
    lst_2.append(precision)
    lst_2.append(recall)
    lst_2.append(f1)
    lst_1.append(lst_2)
```

```
Logistic Regreesion :
[[750 218]
 [ 15  39]]
Accuracy Score:  0.7720156555772995

K-Fold Validation Mean Accuracy: 79.46 %

Standard Deviation: 1.39 %

ROC AUC Score: 0.75

Precision: 0.15

Recall: 0.72

F1: 0.25
-----------------------------------

SVM :
```

In [40]:
```
df = pd.DataFrame(lst_1, columns= ['Model', 'Accuracy', 'K-Fold Mean Accuracy', 'Std. Deviation', 'ROC AUC', 'Precisic
```

In [41]:
```
df.sort_values(by= ['Accuracy', 'K-Fold Mean Accuracy'], inplace= True, ascending= False)
```

In [42]:
```
df
```

Out[42]:

	Model	Accuracy	K-Fold Mean Accuracy	Std. Deviation	ROC AUC	Precision	Recall	F1
7	XGBoost	91.976517	95.107316	3.770300	0.537994	0.150000	0.111111	0.127660
6	Random Forest	90.606654	95.967343	1.377559	0.522019	0.096154	0.092593	0.094340
5	Decision Tree	86.007828	90.393013	2.936513	0.515228	0.067961	0.129630	0.089172
2	KNeighbors	83.659491	90.765631	0.866181	0.599001	0.120805	0.333333	0.177340
1	SVM	79.843444	88.363831	1.536875	0.613828	0.112245	0.407407	0.176000
0	Logistic Regreesion	77.201566	79.462926	1.391926	0.748508	0.151751	0.722222	0.250804
4	BernoulliNB	60.567515	72.938676	1.593421	0.713154	0.102506	0.833333	0.182556
3	GaussianNB	19.275930	57.333590	0.861249	0.556378	0.059429	0.962963	0.111948

The output presents performance metrics for multiple machine learning models. Notably, Random Forest and XGBoost exhibit strong accuracy scores around 90.61% and 91.98%, respectively. K-Fold Validation Mean Accuracy further confirms their robustness, averaging approximately 95.97% and 95.11%, respectively. Standard deviation indicates consistent performance, particularly for GaussianNB and KNeighbors. Logistic Regression achieves a moderate ROC AUC score of about 0.75, reflecting its ability to distinguish between classes. However, models like Random Forest show lower precision, recall, and F1-score, suggesting limitations in correctly identifying positive instances. Overall, Random Forest and XGBoost emerge as top-performing models, demonstrating reliability across various evaluation metrics.

Model Tuning

In [43]:
```
from sklearn.model_selection import GridSearchCV
```

The GridSearchCV function is a utility provided by the scikit-learn library's model_selection module. It facilitates an automated search through a predefined set of hyperparameters, allowing for the fitting of your estimator (model) on a training set with different parameter combinations. Ultimately, GridSearchCV aids in identifying the optimal parameters from the specified hyperparameter grid.

In [44]:
```
grid_models = [(LogisticRegression(),[{'C':[0.25,0.5,0.75,1],'random_state':[0]}]),
               (KNeighborsClassifier(),[{'n_neighbors':[5,7,8,10], 'metric': ['euclidean', 'manhattan', 'chebyshev', '
               (SVC(),[{'C':[0.25,0.5,0.75,1],'kernel':['linear', 'rbf'],'random_state':[0]}]),
               (GaussianNB(),[{'var_smoothing': [1e-09]}]),
               (BernoulliNB(), [{'alpha': [0.25, 0.5, 1]}]),
               (DecisionTreeClassifier(),[{'criterion':['gini','entropy'],'random_state':[0]}]),
               (RandomForestClassifier(),[{'n_estimators':[100,150,200],'criterion':['gini','entropy'],'random_state':
              (XGBClassifier(), [{'learning_rate': [0.01, 0.05, 0.1], 'eval_metric': ['error']}])]
```

In [45]:
```
for i,j in grid_models:
    grid = GridSearchCV(estimator=i,param_grid = j, scoring = 'accuracy',cv = 10)
    grid.fit(x_train_res, y_train_res)
    best_accuracy = grid.best_score_
    best_param = grid.best_params_
    print('{}:\nBest Accuracy : {:.2f}%'.format(i,best_accuracy*100))
    print('Best Parameters : ',best_param)
    print('')
    print('----------------')
    print('')
```

```
LogisticRegression():
Best Accuracy : 79.46%
Best Parameters :  {'C': 1, 'random_state': 0}

----------------

KNeighborsClassifier():
Best Accuracy : 91.92%
Best Parameters :  {'metric': 'manhattan', 'n_neighbors': 5}

----------------

SVC():
Best Accuracy : 88.36%
Best Parameters :  {'C': 1, 'kernel': 'rbf', 'random_state': 0}

----------------

GaussianNB():
Best Accuracy : 57.33%
Best Parameters :  {'var_smoothing': 1e-09}

----------------

BernoulliNB():
Best Accuracy : 72.94%
Best Parameters :  {'alpha': 0.25}

----------------

DecisionTreeClassifier():
Best Accuracy : 91.10%
Best Parameters :  {'criterion': 'entropy', 'random_state': 0}

----------------

RandomForestClassifier():
Best Accuracy : 95.97%
Best Parameters :  {'criterion': 'gini', 'n_estimators': 100, 'random_state': 0}

----------------
```

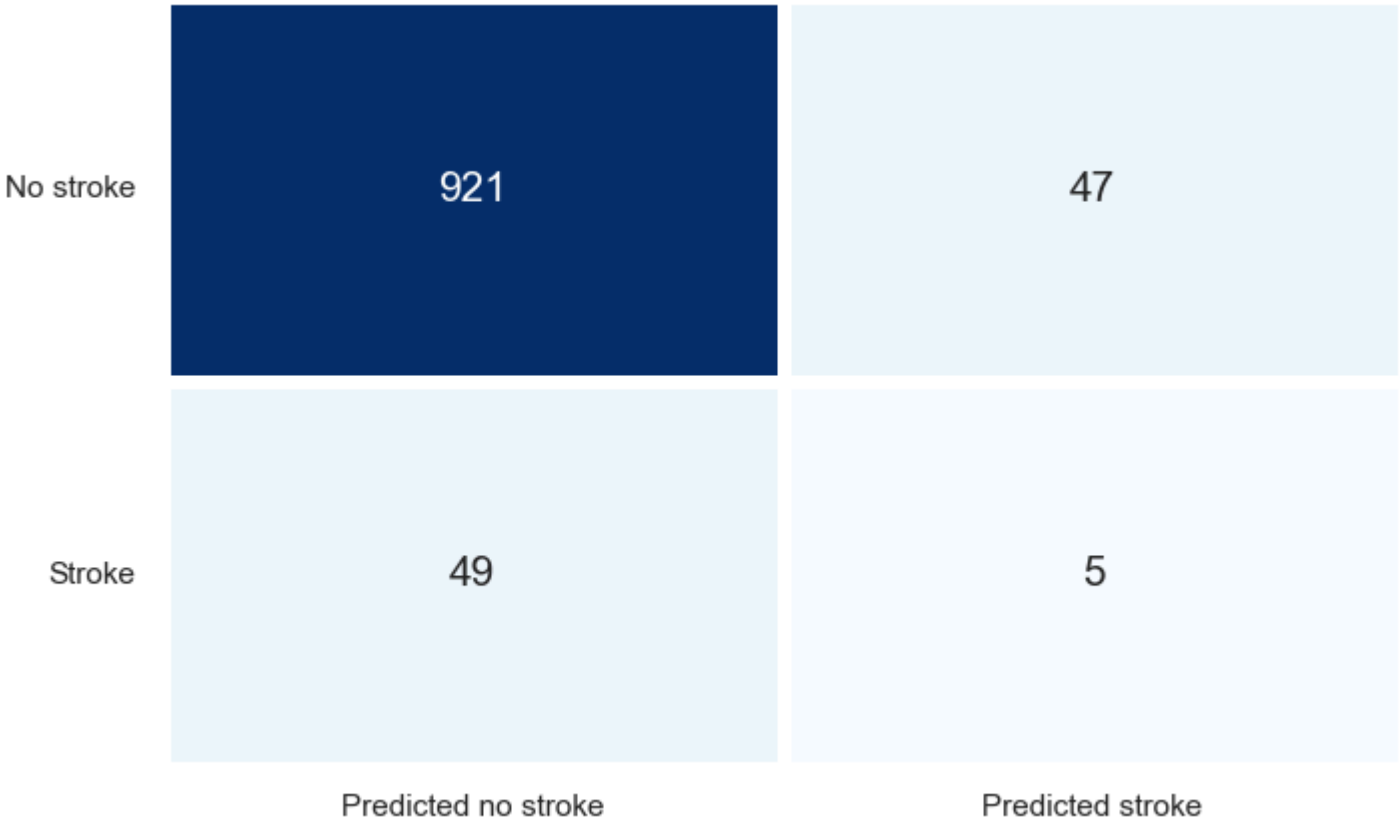
```
XGBClassifier(base_score=None, booster=None, callbacks=None,
              colsample_bylevel=None, colsample_bynode=None,
              colsample_bytree=None, device=None, early_stopping_rounds=None,
              enable_categorical=False, eval_metric=None, feature_types=None,
              gamma=None, grow_policy=None, importance_type=None,
              interaction_constraints=None, learning_rate=None, max_bin=None,
              max_cat_threshold=None, max_cat_to_onehot=None,
              max_delta_step=None, max_depth=None, max_leaves=None,
              min_child_weight=None, missing=nan, monotone_constraints=None,
              multi_strategy=None, n_estimators=None, n_jobs=None,
              num_parallel_tree=None, random_state=None, ...):
Best Accuracy : 92.98%
Best Parameters :  {'eval_metric': 'error', 'learning_rate': 0.1}

----------------
```

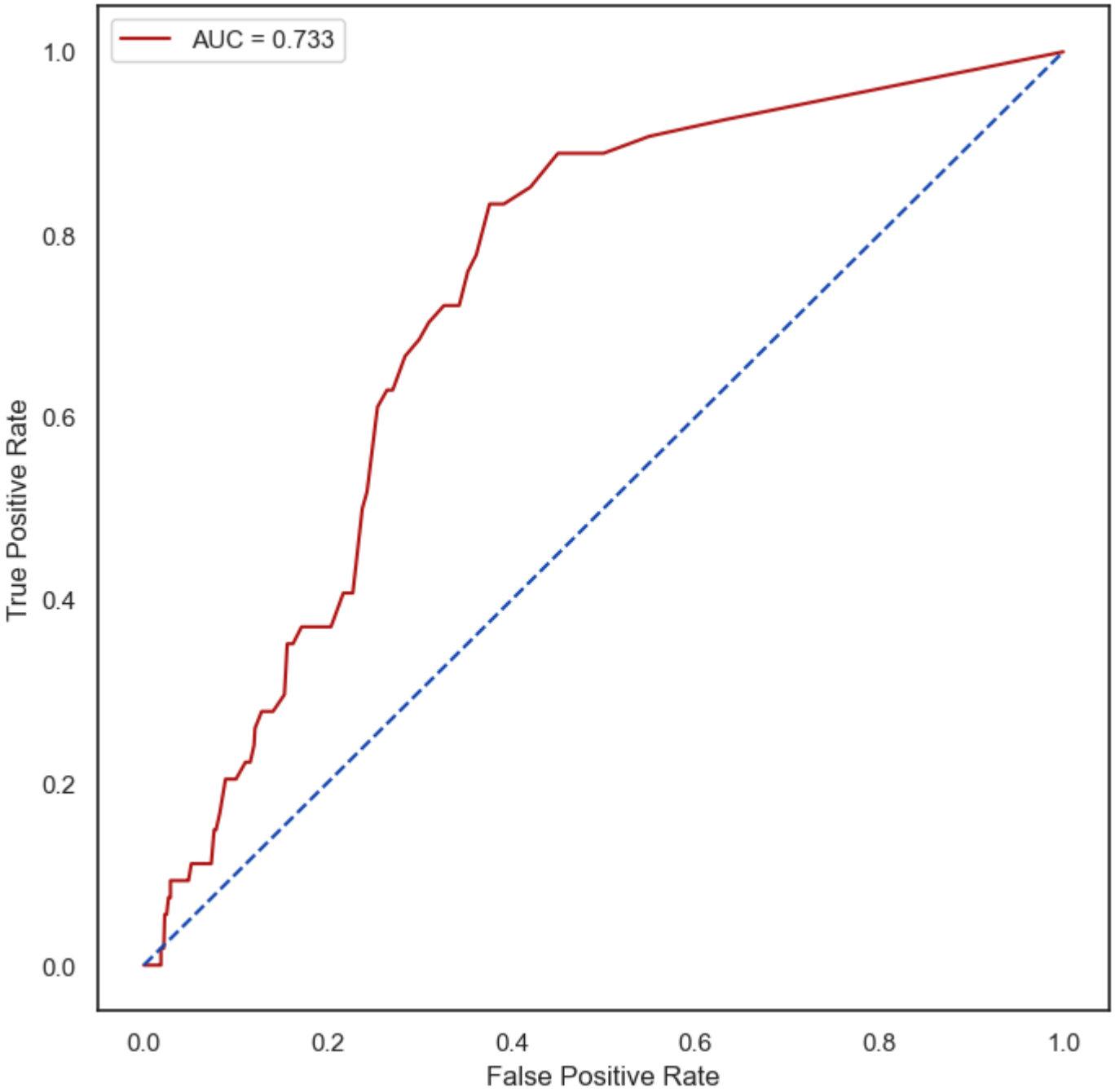
After examining the results of the GridSearch, it's clear that RandomForest and XGBoost are the most suitable models for our dataset.

Hyperparameter Tuning

In [46]:
```
#RandomForest

#Fitting
classifier = RandomForestClassifier(criterion= 'gini', n_estimators= 100, random_state= 0)
classifier.fit(x_train_res, y_train_res)
y_pred = classifier.predict(x_test)
y_prob = classifier.predict_proba(x_test)[:,1]
cm = confusion_matrix(y_test, y_pred)

print(classification_report(y_test, y_pred))
print(f'ROC AUC score: {roc_auc_score(y_test, y_prob)}')
print('Accuracy Score: ',accuracy_score(y_test, y_pred))

# Visualizing Confusion Matrix
plt.figure(figsize = (8, 5))
sns.heatmap(cm, cmap = 'Blues', annot = True, fmt = 'd', linewidths = 5, cbar = False, annot_kws = {'fontsize': 15},
            yticklabels = ['No stroke', 'Stroke'], xticklabels = ['Predicted no stroke', 'Predicted stroke'])
plt.yticks(rotation = 0)
plt.show()

# Roc AUC Curve
false_positive_rate, true_positive_rate, thresholds = roc_curve(y_test, y_prob)
roc_auc = auc(false_positive_rate, true_positive_rate)

sns.set_theme(style = 'white')
plt.figure(figsize = (8, 8))
plt.plot(false_positive_rate,true_positive_rate, color = '#b01717', label = 'AUC = %0.3f' % roc_auc)
plt.legend(loc = 'lower right')
plt.plot([0, 1], [0, 1], linestyle = '--', color = '#174ab0')
plt.axis('tight')
plt.ylabel('True Positive Rate')
plt.xlabel('False Positive Rate')
plt.legend()
plt.show()
```

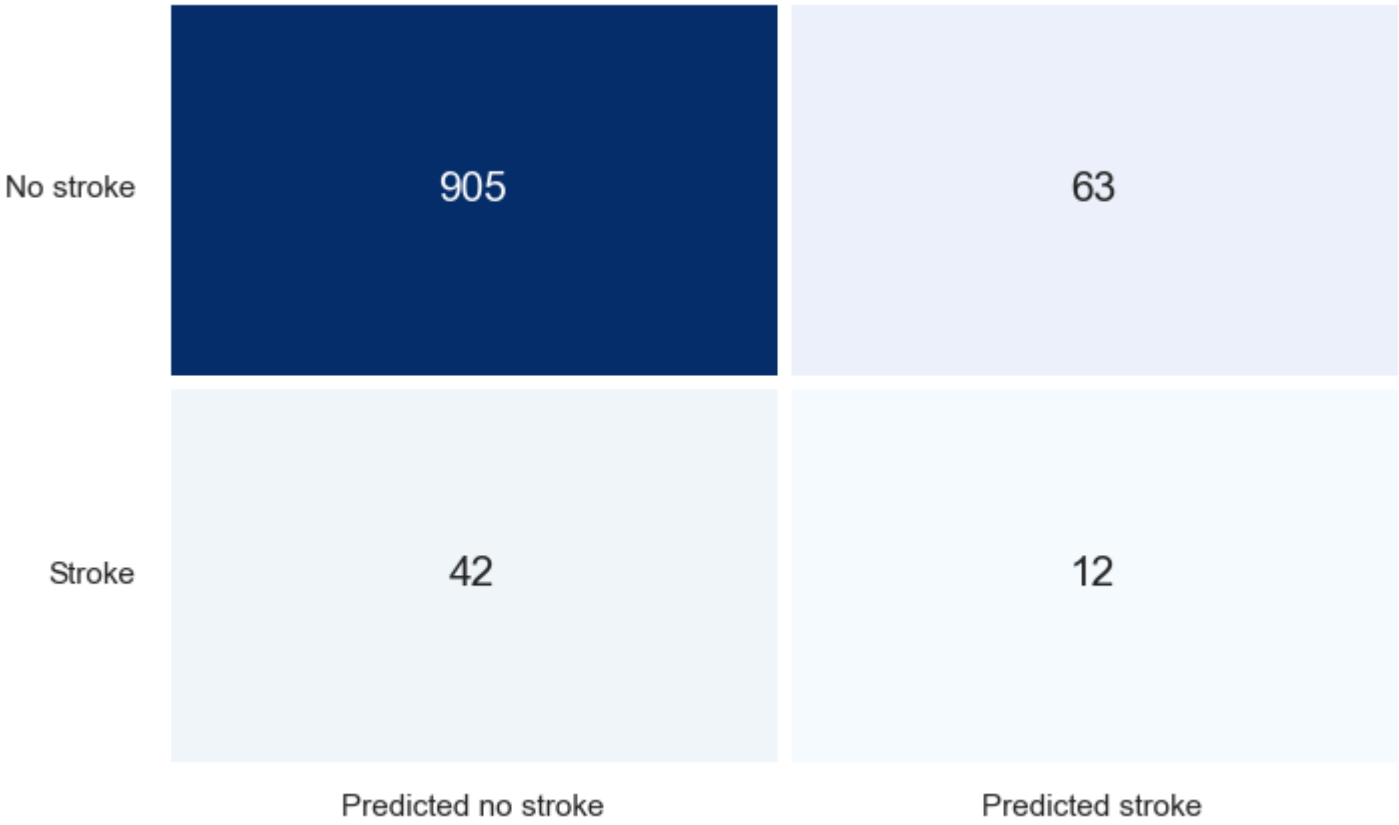
```
              precision    recall  f1-score   support

           0       0.95      0.95      0.95       968
           1       0.10      0.09      0.09        54

    accuracy                           0.91      1022
   macro avg       0.52      0.52      0.52      1022
weighted avg       0.90      0.91      0.91      1022

ROC AUC score: 0.7328397612488522
Accuracy Score:  0.9060665362035225
```



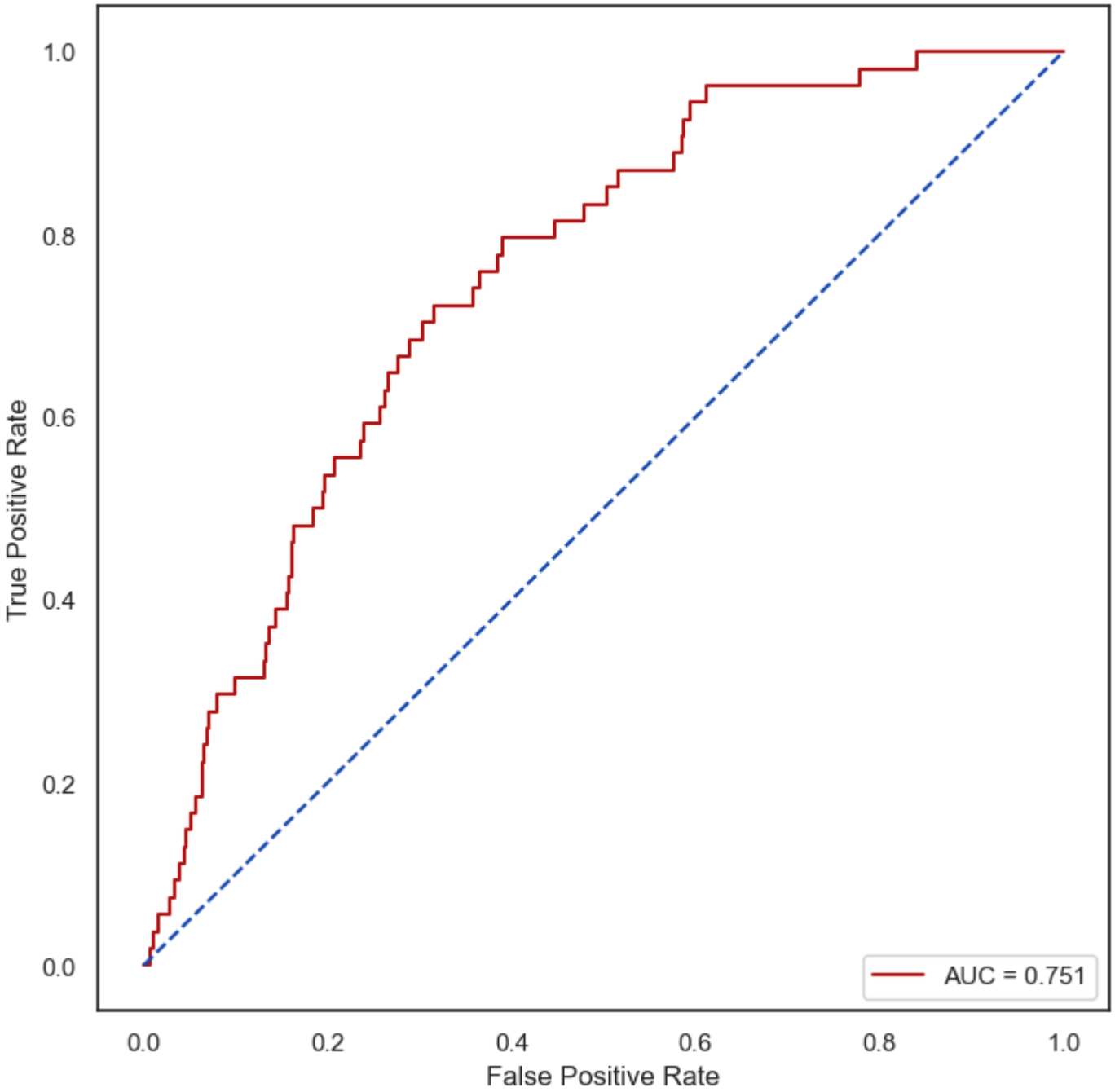

AUC = 0.733
1.0
0.8
0.6
0.4
0.2
0.0
True Positive Rate
0.0
0.2
0.4
0.6
0.8
1.0
False Positive Rate

In [47]:
```
#XGBoost
#Fitting

classifier = XGBClassifier(eval_metric= 'error', learning_rate= 0.1)
classifier.fit(x_train_res, y_train_res)
y_pred = classifier.predict(x_test)
y_prob = classifier.predict_proba(x_test)[:,1]
cm = confusion_matrix(y_test, y_pred)

print(classification_report(y_test, y_pred))
print(f'ROC AUC score: {roc_auc_score(y_test, y_prob)}')
print('Accuracy Score: ',accuracy_score(y_test, y_pred))

# Visualizing Confusion Matrix
plt.figure(figsize = (8, 5))
sns.heatmap(cm, cmap = 'Blues', annot = True, fmt = 'd', linewidths = 5, cbar = False, annot_kws = {'fontsize': 15},
            yticklabels = ['No stroke', 'Stroke'], xticklabels = ['Predicted no stroke', 'Predicted stroke'])
plt.yticks(rotation = 0)
plt.show()

# Roc Curve
false_positive_rate, true_positive_rate, thresholds = roc_curve(y_test, y_prob)
roc_auc = auc(false_positive_rate, true_positive_rate)

sns.set_theme(style = 'white')
plt.figure(figsize = (8, 8))
plt.plot(false_positive_rate,true_positive_rate, color = '#b01717', label = 'AUC = %0.3f' % roc_auc)
plt.legend(loc = 'lower right')
plt.plot([0, 1], [0, 1], linestyle = '--', color = '#174ab0')
plt.axis('tight')
plt.ylabel('True Positive Rate')
plt.xlabel('False Positive Rate')
```

```
              precision    recall  f1-score   support

           0       0.96      0.93      0.95       968
           1       0.16      0.22      0.19        54

    accuracy                           0.90      1022
   macro avg       0.56      0.58      0.57      1022
weighted avg       0.91      0.90      0.91      1022

ROC AUC score: 0.7513965411692685
Accuracy Score:  0.8972602739726028
```



Out[47]: Text(0.5, 0, 'False Positive Rate')


1.0
0.8
0.6
0.4
0.2
0.0
True Positive Rate
0.0
0.2
0.4
0.6
0.8
1.0
False Positive Rate
AUC = 0.751

Based on our evaluation metrics, it seems like the XGBoost model may be slightly better for our classification task compared to the Random Forest model. Here's why:

Firstly, XGBoost shows a higher recall for class 1, which means it's better at identifying individuals who have had a stroke. This is crucial because we want our model to catch as many stroke cases as possible.

Secondly, the F1-score for class 1 is slightly higher in XGBoost, indicating a better balance between precision and recall. This suggests that XGBoost is more accurate in identifying stroke cases while minimizing false positives.

Lastly, XGBoost has a slightly higher ROC AUC score, which means it's better at distinguishing between positive and negative instances.

Overall, while both models perform similarly in terms of accuracy, XGBoost appears to have a slight edge in correctly identifying stroke cases. However, further analysis and model tuning may be needed to improve the performance of both models.

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]: