Comparison of Different Machine Learning Algorithms in the Prediction of Heart Disease

1. Introduction

Cardiovascular diseases (CVDs) are a global health concern, contributing significantly to mortality rates. CVDs encompass a range of conditions, such as cerebrovascular disease, coronary heart disease, and various other heart and vascular disorders. It's alarming to note that heart attacks and strokes contribute to more than 80% of CVD-related fatalities, and a significant portion of these unfortunate deaths transpires before the age of 70 [1].

Early detection and prediction of heart diseases are vital for timely intervention and prevention. In this study, we compare the performance of five machine learning algorithms, namely, K-Nearest Neighbors (KNN), Logistic Regression, Decision Tree, and Random Forest in predicting the presence or absence of heart disease.

Section 2 of this work delves into problem formulation, providing insights into the data sets, explaining the meaning and types of each data point, and detailing the process of feature selection. Following this, Section 3 sheds light on data pre-processing, the creation of train, validation, and test sets, and introduces the methods used alongside their corresponding loss functions. Moving to Section 4, the report presents and compares results obtained from each model. The concluding Section 5 summarizes the report and interprets the results, providing a comprehensive overview of the entire study.

2. Problem Formulation

The objective of this work is to predict the presence or absence of heart disease using machine learning. This binary classification task aims to classify individuals into two categories: "negative" (no heart failure) and "positive" (heart failure).

To achieve our objectives, two distinct datasets were employed (a reason for this decision will be explained in a later stage of the report). The first dataset contains a total of 1319 data points, while the second one has 5110 data points. Each point in both data sets corresponds to an individual patient and comprising the properties listed in Tables 1 and 2.

Table 1: Characteristics of the first dataset.

Property	Range	type
Age (year)	14 - 103	Discrete
Gender	0 (F) or 1 (M)	Binary
Heart rate	20 - 1111	Continuous
Systolic BP	42 - 223	Continuous
Diastolic BP	38 - 154	Continuous
Blood sugar	35.0 - 541.0	Continuous
CK-MB enzyme	0.321 - 300.0	Continuous
Troponin	0.001 - 10.3	Continuous
class	Negative or Positive	Binary

Table 2: Characteristics of the second dataset.

Property	Range	Type
Age (year)	0.08 - 82	Discrete
Gender	0 (F) or 1 (M)	Binary
Hypertension	0 or 1	Binary
Heart Disease	0 or 1	Binary
Ever Married	Yes or No	Binary
Work Type	Private, Self-employed, Other	Discrete
Residence Type	Urban or Rural	Binary
Glucose	55.1 - 272.0	Continuous
BMI	11.5 - 92.0	Continuous
Smoking	Never, Formerly, smokes	Discrete
Stroke	Negative or Positive	Binary

In the context of our study, the primary aim is to predict heart failure using supervised learning techniques. To achieve this goal, we carefully curated our feature set by selecting properties found in the blue rows of Tables 1 and 2. These features are considered essential factors contributing to heart diseases.

Upon visualizing the second dataset, the residence type was excluded as a feature. This decision was made based on the observation that its impact on the probability of having a stroke appeared to be weak, as illustrated in Figure 1.

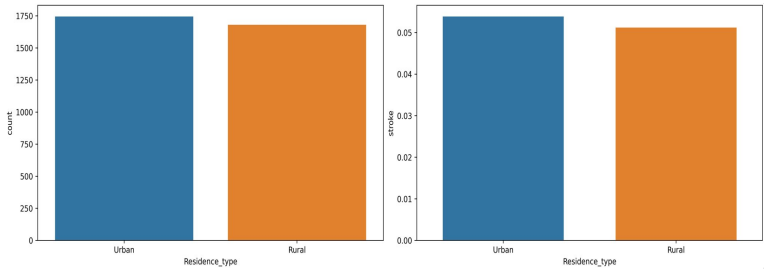


Figure 1: Total count of each residence type (left) and their impact on the probability of having a stroke (right).

Additionally, we identified the "class" and "Stroke" properties (for dataset 1 and 2 respectively), highlighted in the red row of Tables 1 and 2, as our designated label variables. These variables serves as the indicator for the presence or absence of a heart attack, which is central to our predictive modeling.

The datasets were meticulously constructed to collect characteristics and risk factors associated with heart attacks and were sourced from the Kaggle website [2, 3].

3. Methods

3.1. Data Pre-processing

3.1.1 First Dataset

Extensive pre-processing of the dataset was unnecessary, and there were no null values in the set. However, the string values within the "class" property were seamlessly converted to integers, assigning "negative" to 0 and "positive" to 1.

Subsequently, a preliminary data visualization was performed to assess data points, resulting in the identification and removal of 4 noisy data points. The dataset was then re-visualized from various angles, some of which are showcased in Figure 2.

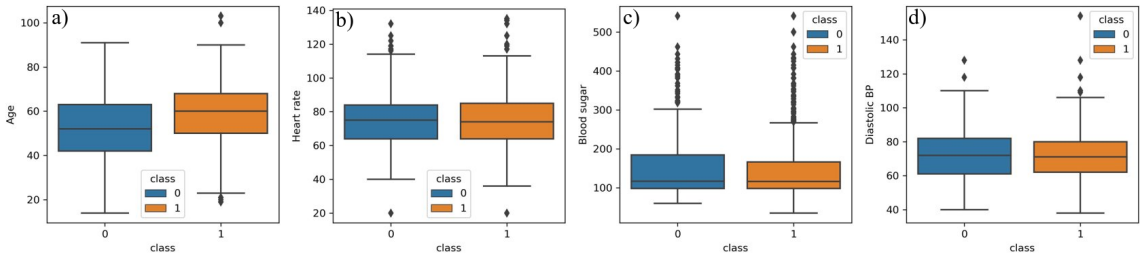


Figure 2: Examples of data visualisation showing the range of a) age, b) heart rate, c) blood sugar, and d) diastolic BP for each class using box plots (class 0 correspond to absence and 1 to presence of heart attack).

3.1.2 Second Dataset

In the second dataset, some data points have the value "other" as gender. Initially, these points were excluded from the set. Similarly, data points with "unknown" value for smoking status were also excluded, to make the dataset suitable fore training. Furthermore, When we examine the boxplots (see the example in Figure 3), we can see that there are some outlier values, although not too many. After removing the outliers, the dataset contains 2882 data points.

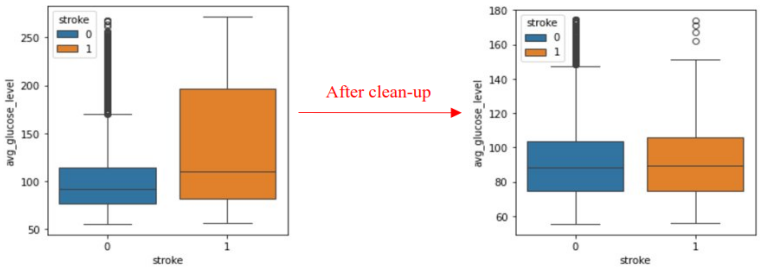


Figure 3: Examples of data visualisation showing the effect of cleaning outliers.

Furthermore, in pursuit of a balanced dataset with an equal representation of positive and negative stroke cases, a subset of the dataset was meticulously chosen and employed, resulting in a total of 350 data points. This decision was driven by the fact that, following pre-processing, only 175 data points remained in the dataset with a positive stroke case. This approach was crucial to mitigate bias in the prediction process and ensure a more equitable and representative model.

3.2. Data Preparation

3.2.1 First Dataset

The first dataset was directly divided into training and validation sets using the train_test_split() function from the sklearn library. To achieve this, 80% of the data (1052 points) were randomly sampled and allocated to the training set, while the remaining 20% (263 points) were designated for the validation set.

The choice of this splitting method was primarily driven by the sizable number of data points available, which makes it a reliable approach.

3.2.2 Second Dataset

For the second dataset, he splitting method was the same. However, the process itself had one difference. Here, 80% of the data (280 points) were randomly sampled and allocated to the training set, while the remaining 20% were randomly designated for the validation (35 points) and test (35 points) sets.

3.3. Logistic Regression

Logistic Regression predicts the probability of a data point belonging to one of the two defined categories by modeling the relationship between input features and the labels. Thus, logistic regression is a binary classification method, which makes it a suitable fit for our problem.

Moreover, logistic regression employs the logistic loss function to evaluate the performance of the linear hypothesis. It's worth noting that the Scikit-learn library already incorporates the logistic loss function, simplifying its integration and usage [4].

3.4. K-Nearest Neighbors (KNN)

K-Nearest Neighbors (KNN) makes predictions by examining the majority class or calculating the average value of its nearest data points in the training dataset. Key parameters of KNN include "K," which represents the number of neighboring data points to consider, and it utilizes distance metrics like the Euclidean distance to assess the similarity between data points. In this work, we experimented with different values of K, ranging from 3 to 15, to find the optimal K value for our dataset. The optimal K values for our cases were determined to be 11 and 12 for the first and second datasets, respectively.

KNN stands out as a non-parametric and instance-based machine learning algorithm. Unlike many other algorithms that rely on traditional loss functions, KNN doesn't follow this approach. Instead, it determines predictions based on the most frequently occurring class among the k-nearest neighbors to a specific data point.

3.5. Decision Trees

Decision Tree create a tree-like structure where each internal node represents a decision based on a feature, and each leaf node represents the outcome or prediction. Decision Tree makes decisions by recursively splitting the dataset into subsets based on the most informative features, aiming to maximize class purity or minimize prediction error. The choice of the loss function for this method will be discussed in the following section.

3.6. Random Forest

Random Forest extends the concept of Decision Tree by constructing multiple trees, each trained on a different subset of the data and utilizing a random subset of features. The primary goal is to mitigate overfitting while enhancing predictive accuracy through the combination of predictions from these diverse trees.

In both Decision Tree and Random Forest, we employed three distinct criteria called, "gini", "entropy", and "log loss". Gini impurity measures the probability of misclassifying a randomly chosen element in a dataset, with values ranging from 0 (complete purity, where all elements belong to a single class) to 0.5 (full impurity, where elements are evenly distributed across all classes). In contrast, entropy measures the amount of disorder or uncertainty in a dataset, with values also ranging from 0 (complete purity) to 1 (maximum impurity) [5]. On the other hand, log loss is a loss function primarily used for classification problems. Unlike Gini impurity and entropy, log loss measures the dissimilarity between predicted class probabilities and true class labels.

In the first dataset, the Decision Tree method demonstrated the most favorable results with the entropy criterion, while the Random Forest achieved its highest accuracy using the gini criterion. Conversely, in the second dataset, the gini criterion proved optimal for both methods, resulting in the highest accuracy.

4. Results and Discussion

While working with the first dataset, it was noticed that the prediction accuracy highly depends on troponin feature. Table 3 shows the accuracy of each model for dataset 1, in three different conditions: 1) using all features, 2) using all features except troponin, and 3) using just troponin to train the model.

It is evident that training the model solely on troponin yields a notably high accuracy. This observation motivated the selection of a second dataset, where each feature contributes more evenly to the overall accuracy of the model.

Table 3: Validation accuracy of each model for dataset 1.

Model	All features	Excluding troponin	Only troponin
Logistic Regression	79.09%	70.34%	71.48%
KNN	67.68%	67.68%	87.07%
Decision Trees	97.71%	71.10%	87.07%
Random Forest	97.71%	71.10%	87.07%

For the second dataset, the same four methods were employed. Table 4 presents the validation and test accuracy of each model. Notably, the validation accuracy for all models is consistently high and closely aligned (similar for Logistic Regression and Random Forest). Hence, test accuracy and recall is reported for all models, as the variation in their validation errors is negligible. Additionally, the confusion matrix of all 4 methods are depicted in Figure 4.

Table 4: Accuracy of each model for dataset 2.

Model	Validation accuracy	Test accuracy	Test recall
Logistic Regression	88.57%	80.00%	78.94%
KNN	85.71%	71.42%	78.94%
Decision Trees	82.85%	68.57%	68.42%
Random Forest	88.57%	77.14%	78.94%

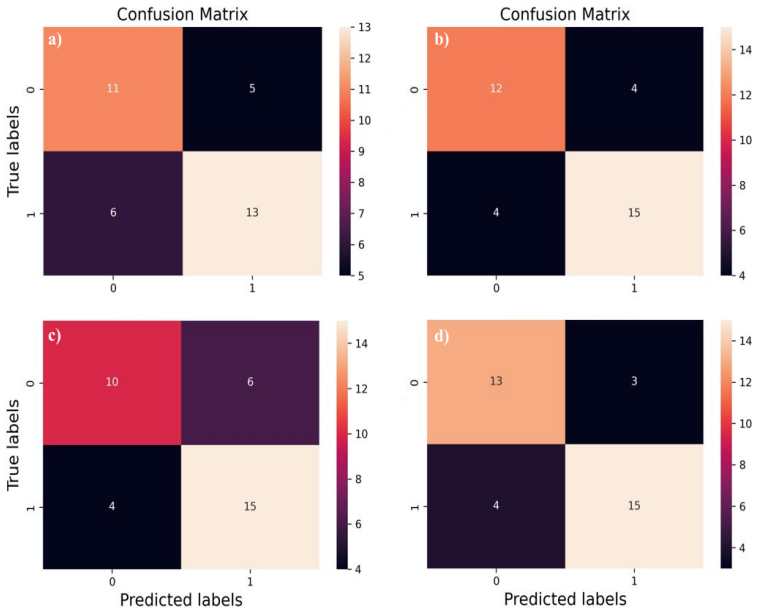


Figure 4: Confusion matrix for a) decision tres, b) random forest, c) KNN, and d) logistic regression.

Drawing insights from the presented results, it is evident that Logistic Regression and Random Forest showcase the most favorable combination of high validation accuracy and test recall. Notably, Logistic Regression stands out with a superior test accuracy, making it the optimal method for this dataset.

5. Conclusion

In this study, we applied four distinct machine learning models to two sets of data for predicting the presence or absence of heart disease or stroke. The decision to use two datasets stemmed from the initial dataset's significant dependency on a single feature, namely troponin.

The results highlight that in the second dataset, all models exhibit similar and high validation accuracy. However, upon comparing the test accuracy and test recall of the 4 methods, it is evident that Logistic Regression is the best method, resulting in a higher accuracy and recall. Moreover, the proximity of validation and test accuracy in this method suggests the absence of overfitting.

For future work, collecting more data points will further enhance the predictive capabilities of the models. Fine-tuning hyperparameters and conducting a more in-depth analysis of the data distribution may further optimize model performance. Additionally, investigating the interpretability of the models and their implications in a clinical setting could provide valuable insights. Moreover, assessing the robustness of the models across diverse populations and datasets would contribute to their generalizability.

References

[1] World Health Organization. *Cardiovascular Diseases (CVDs): A Global Health Concern*. 2021. URL: https://www.who.int/health-topics/cardiovascular-diseases#tab=tab_1.

[2] Kaggle, Inc. *Kaggle: Your Machine Learning and Data Science Community*. URL: https://www.kaggle.com/datasets/bharath011/heart-disease-classification-dataset.

[3] Kaggle, Inc. *Kaggle: Your Machine Learning and Data Science Community*. URL: https://www.kaggle.com/datasets/fedesoriano/stroke-prediction-dataset.

[4] Fabian Pedregosa et al. *Scikit-learn: Machine Learning in Python*. 2011. URL: http://scikit-learn.sourceforge.net..

[5] Laura Elena Raileanu and Kilian Stoffel. *Theoretical comparison between the Gini Index and Information Gain criteria **. 2004. URL: https://www.unine.ch/files/live/sites/imi/files/shared/documents/papers/Gini_index_fulltext.pdf.

Appendix 1

October 11, 2023

[1]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns

%config Completer.use_jedi = False  # enable code auto-completion

from sklearn.model_selection import train_test_split
from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import accuracy_score, confusion_matrix
from sklearn.metrics import recall_score
from sklearn.ensemble import RandomForestClassifier
from sklearn.svm import SVC
from sklearn.neighbors import KNeighborsClassifier
from sklearn.linear_model import LogisticRegression
from sklearn import tree

import sklearn
```

[2]:
```
# Read data
rawdata = pd.read_csv("Heart Attack.csv")
# Show data examples
rawdata.sample(10)
# Create 2 copied of dataset
# visual_data: used for visualization
# model_data: used for training and testing
visual_data = rawdata.copy(deep=True)
model_data = rawdata.copy(deep=True)
```

[3]:
```
#preprocessing1.1: Use 1 to represent "positive", 0 to represent "negative" -> model_data
diago = model_data["class"].copy(deep=True)
diago = diago.map({"negative": 0, "positive": 1}).copy(deep=True)
model_data["class"] = diago.copy(deep=True)
model_data.sample(5)
```

[3]:

	age	gender	impluse	pressurehight	pressurelow	glucose	kcm	troponin	class
1178	66	1	73	125	78	112.0	2.87	0.028	1
64	61	1	102	130	83	201.0	1.24	0.089	1
827	50	1	66	112	74	146.0	10.11	1.400	1
25	72	1	64	106	68	111.0	2.11	1.390	1
198	50	1	69	165	104	194.0	1.50	0.007	0

[4]:
```
# preprocessing1.2: Use 1 to represent "male", 0 to represent "female" -> model_data
# (The mapping is according to the data documentation)
visual_data["class"] = diago.copy(deep=True)
sex = model_data["gender"].copy(deep=True)
sex = sex.map({1: "male", 0: "female"}).copy(deep=True)
visual_data["gender"] = sex.copy(deep=True)
visual_data.sample(5)
```

[4]:

	age	gender	impluse	pressurehight	pressurelow	glucose	kcm
1025	40	male	95	101	76	167.0	3.570
561	50	male	52	171	80	210.0	1.630
519	52	male	100	119	66	127.0	11.730
63	45	male	1111	141	95	109.0	1.330

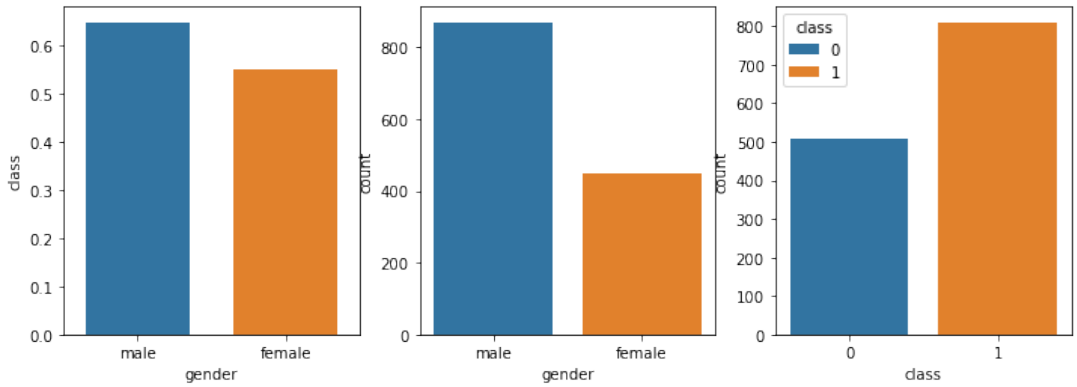
```
1274   70   male      103            126            75     541.0   0.665

      troponin  class
1025     0.029      1
561      0.662      1
519      0.018      1
63       1.010      1
1274     0.014      0
```

[5]:
```
rawdata.describe()
```

[5]:
```
                age       gender      impluse  pressurehight  pressurelow  \
count  1319.000000  1319.000000  1319.000000    1319.000000  1319.000000
mean     56.191812     0.659591    78.336619     127.170584    72.269143
std      13.647315     0.474027    51.630270      26.122720    14.033924
min      14.000000     0.000000    20.000000      42.000000    38.000000
25%      47.000000     0.000000    64.000000     110.000000    62.000000
50%      58.000000     1.000000    74.000000     124.000000    72.000000
75%      65.000000     1.000000    85.000000     143.000000    81.000000
max     103.000000     1.000000  1111.000000     223.000000   154.000000

           glucose          kcm     troponin
count  1319.000000  1319.000000  1319.000000
mean    146.634344    15.274306     0.360942
std      74.923045    46.327083     1.154568
min      35.000000     0.321000     0.001000
25%      98.000000     1.655000     0.006000
50%     116.000000     2.850000     0.014000
75%     169.500000     5.805000     0.085500
max     541.000000   300.000000    10.300000
```

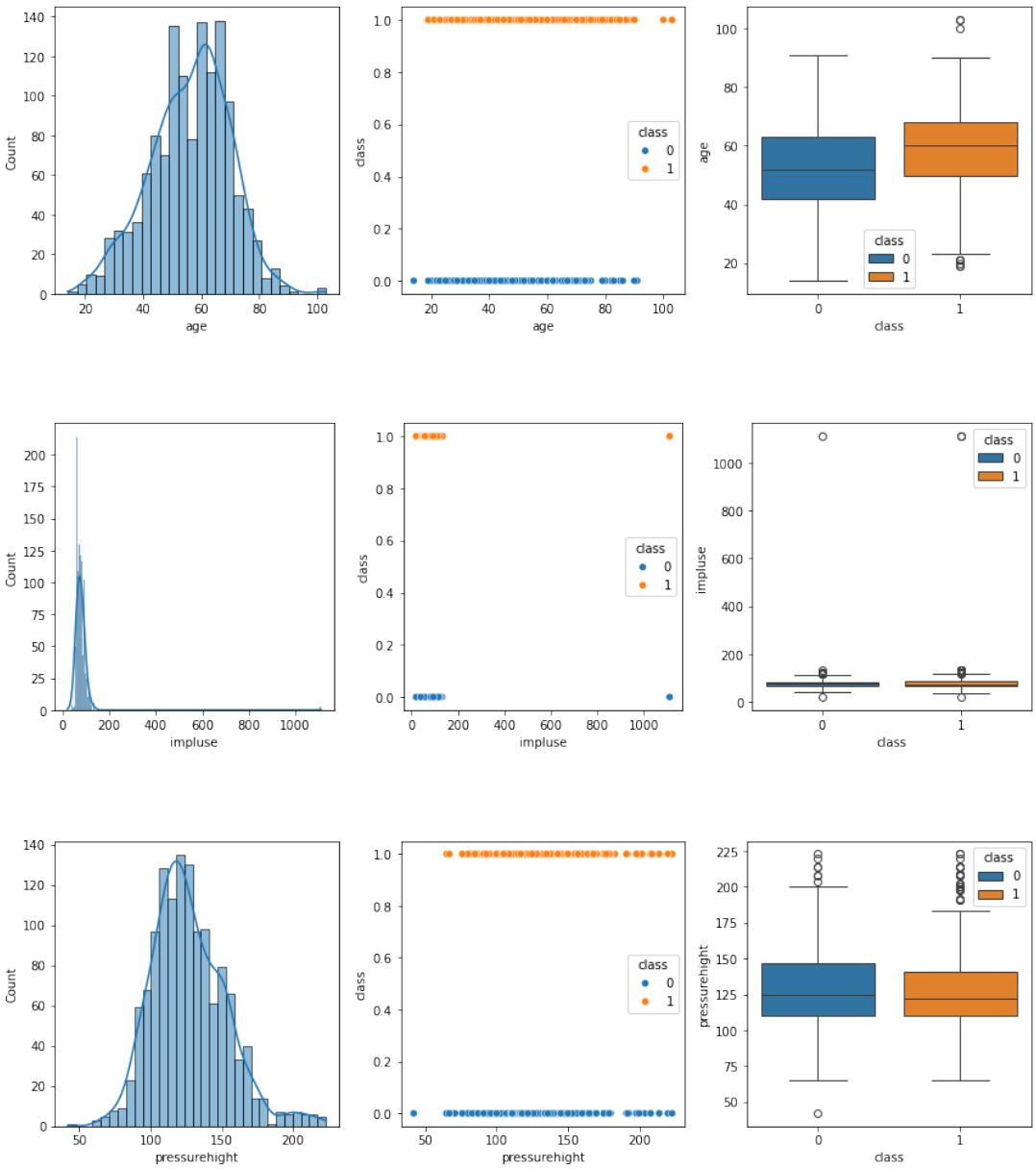
[6]:
```
# preprocessing2: Data visualization and analysis
# Seperate columns into categorical(discrete) and numerical(continuous)
# All features: 'gender', 'age', 'impluse', 'pressurehight', 'pressurelow', 'glucose', 'kcm', 'troponin'
categoric_columns = ["gender"]
numeric_columns = ['age', 'impluse', 'pressurehight', 'pressurelow', 'glucose', 'kcm', 'troponin']
```

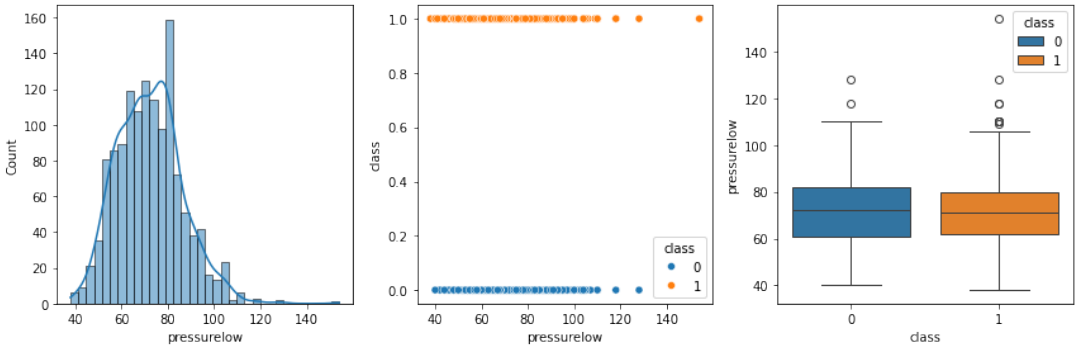
[7]:
```
# Gender(the only categorical feature)
fig, axes = plt.subplots(1, 3, figsize=(12, 4))
# Gender vs Class
sns.barplot(x="gender", y="class", data=visual_data, width=0.7, errorbar=None, hue="gender",␣
↪dodge=False, ax=axes[0])
# Gender count
sns.countplot(data=visual_data, x='gender', ax=axes[1], hue="gender", dodge=False)
# Class count
sns.countplot(data=visual_data, x='class', ax=axes[2], hue="class", dodge=False)

plt.show()
```

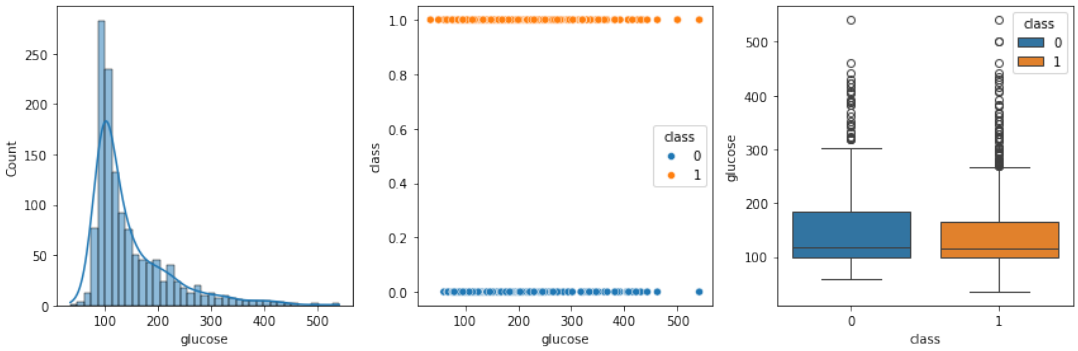


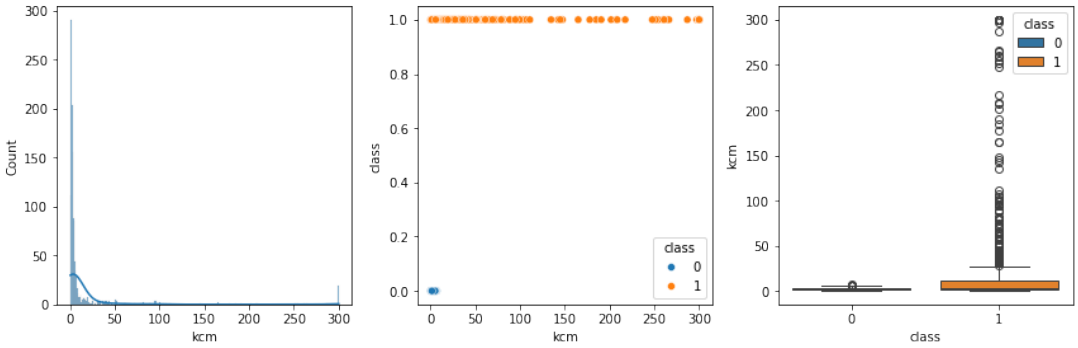
[8]:
```
# numeric data visualization
for index, label in enumerate(numeric_columns):
    fig, axes = plt.subplots(1, 3, figsize=(12,4))
    # distribution
```

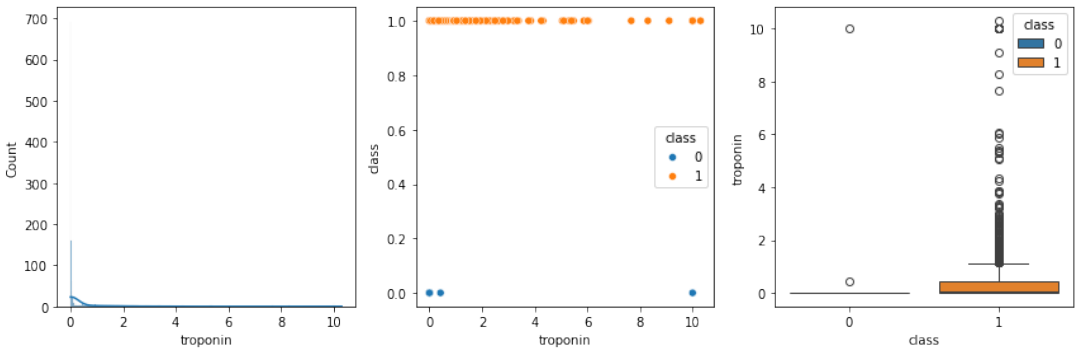
```
    sns.histplot(data=visual_data, x=label, ax=axes[0], kde=True)
    # "label" VS class
    sns.scatterplot(data=visual_data, x=label, y="class", ax=axes[1], hue="class")
    # boxplot
    sns.boxplot(data=visual_data, x='class', y=label, ax=axes[2], hue="class", dodge=False)
    # Adjusting the layout for better visualization
    plt.tight_layout()
    plt.show()
```











Conclusion - "Impluse" $>$ 1000 should be checked - "pressurehight" column is OK - "pressurelow" and "glucose" levels are distributed fairly evenly among patients with both higher and lower risks of heart disease. - kcm $>$ 10 then Heart Attack is Posible - "troponin" $>$ 1 rows are more likely to be positive. So the "troponin" $>$ 9 && "class" == negative should be checked

Noise can lead to unexpected outcome when being trained, so it should be removed from the dataset

[9]:
```
# Abnormal impluse
model_data[model_data["impluse"] > 1000]
```

[9]:

	age	gender	impluse	pressurehight	pressurelow	glucose	kcm	troponin	class
63	45	1	1111	141	95	109.0	1.33	1.010	1
717	70	0	1111	141	95	138.0	3.87	0.028	1
1069	32	0	1111	141	95	82.0	2.66	0.008	0

[10]:
```
# Delete abnormal rows in model_data
model_data = model_data.drop(model_data[model_data["impluse"] > 1000].index)
visual_data = visual_data.drop(visual_data[visual_data["impluse"] > 1000].index)
model_data[model_data["impluse"] > 1000]
```

[10]:
```
Empty DataFrame
Columns: [age, gender, impluse, pressurehight, pressurelow, glucose, kcm,
troponin, class]
Index: []
```

[11]:
```
# Abnormal troponin
model_data[(model_data["troponin"] > 9)]
```

[11]:

	age	gender	impluse	pressurehight	pressurelow	glucose	kcm	troponin	class
29	63	1	66	135	55	166.0	0.493	10.00	0
475	58	0	80	107	67	166.0	6.480	9.11	1
753	49	1	75	116	71	98.0	37.690	10.00	1
988	57	1	95	129	77	251.0	4.340	10.30	1
1003	68	1	60	199	99	115.0	2.670	10.00	1
1028	68	1	89	145	68	134.0	0.706	10.00	1
1048	68	1	97	105	80	91.0	1.160	10.00	1
1094	65	1	74	140	85	106.0	4.350	10.00	1
1252	70	0	63	105	64	217.0	1.800	10.00	1
1310	70	0	80	135	75	351.0	2.210	10.00	1

I think row29 is noise and should be deleted

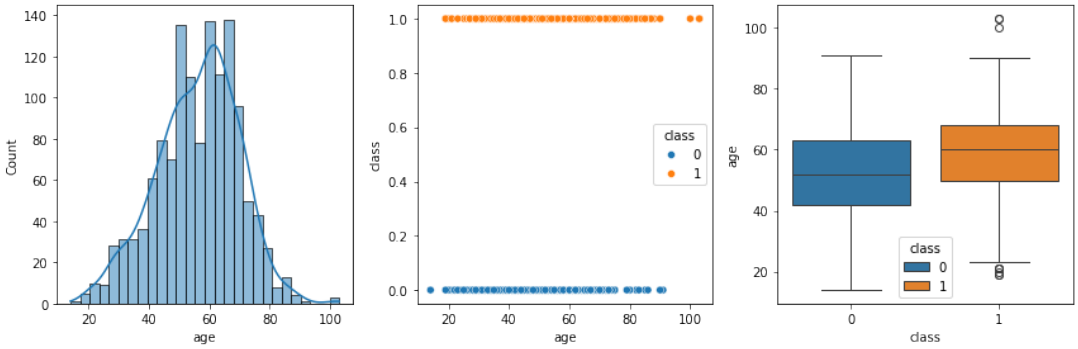
[12]:
```
#Delete abnormal row
model_data = model_data.drop(29)
visual_data = visual_data.drop(29)
model_data[model_data["troponin"] > 9]
```

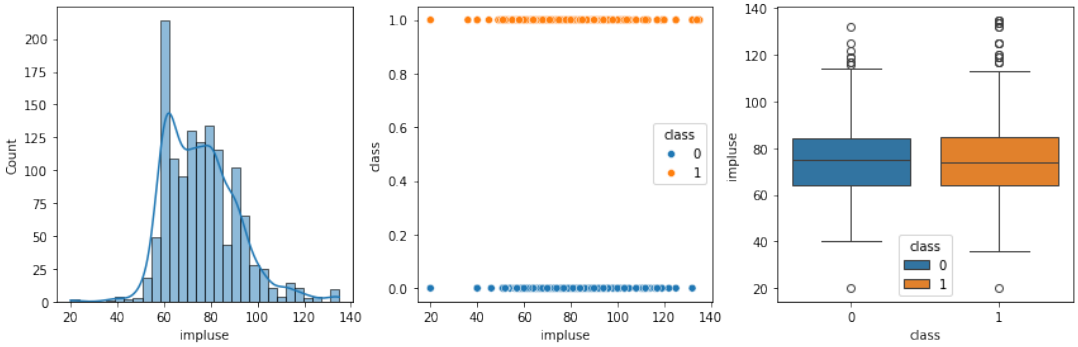
[12]:

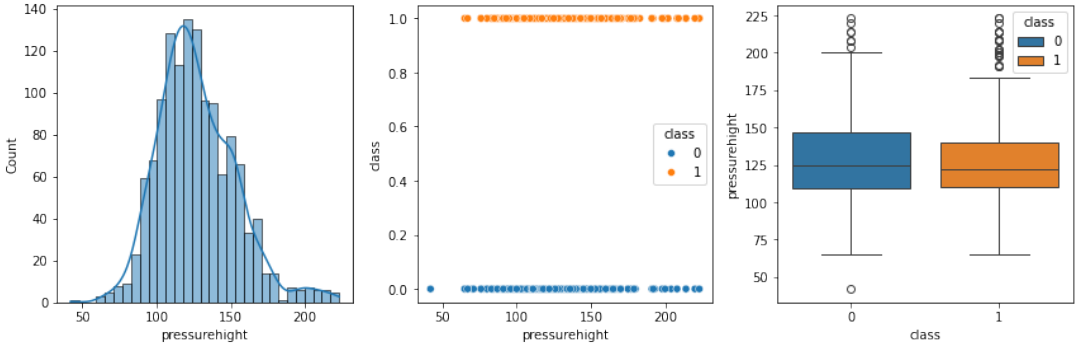
	age	gender	impluse	pressurehight	pressurelow	glucose	kcm	troponin	class
475	58	0	80	107	67	166.0	6.480	9.11	1
753	49	1	75	116	71	98.0	37.690	10.00	1
988	57	1	95	129	77	251.0	4.340	10.30	1
1003	68	1	60	199	99	115.0	2.670	10.00	1
1028	68	1	89	145	68	134.0	0.706	10.00	1
1048	68	1	97	105	80	91.0	1.160	10.00	1
1094	65	1	74	140	85	106.0	4.350	10.00	1
1252	70	0	63	105	64	217.0	1.800	10.00	1
1310	70	0	80	135	75	351.0	2.210	10.00	1

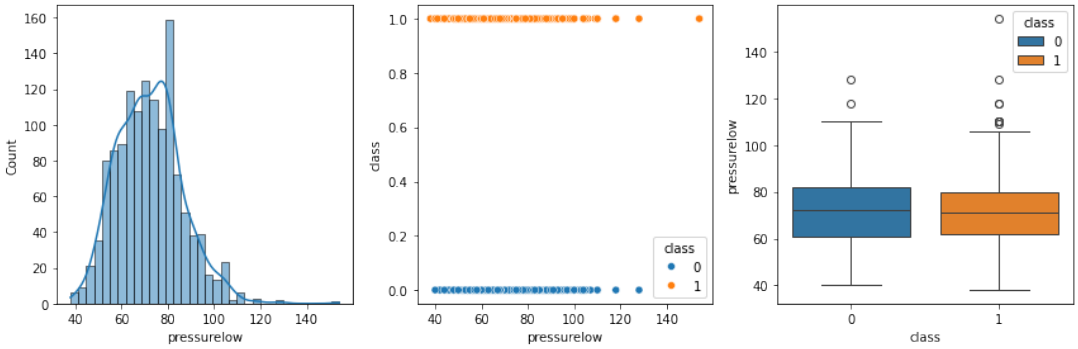
```
753     10.00     1
988     10.30     1
1003    10.00     1
1028    10.00     1
1048    10.00     1
1094    10.00     1
1252    10.00     1
1310    10.00     1
```

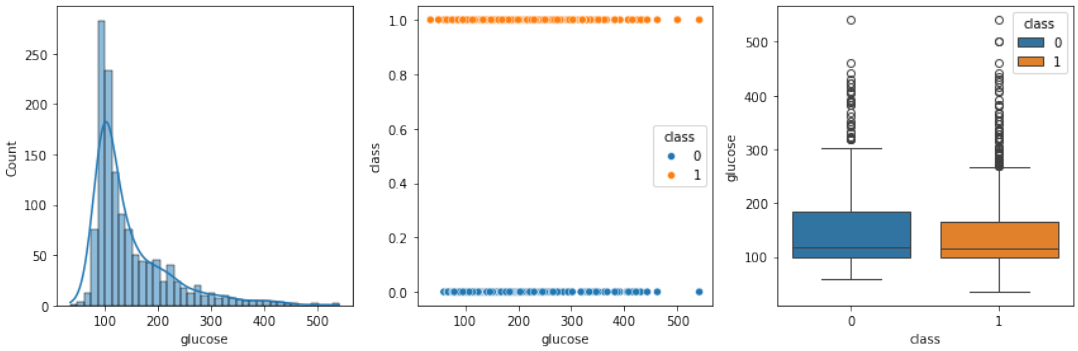
[13]:
```python
# Re-visualization
for index, label in enumerate(numeric_columns):
    fig, axes = plt.subplots(1, 3, figsize=(12,4))
    # distribution
    sns.histplot(data=visual_data, x=label, ax=axes[0], kde=True)
    # "label" VS class
    sns.scatterplot(data=visual_data, x=label, y="class", ax=axes[1], hue="class")
    # boxplot
    sns.boxplot(data=visual_data, x='class', y=label, ax=axes[2], hue="class", dodge=False)
    # Adjusting the layout for better visualization
    plt.tight_layout()
    plt.show()
```

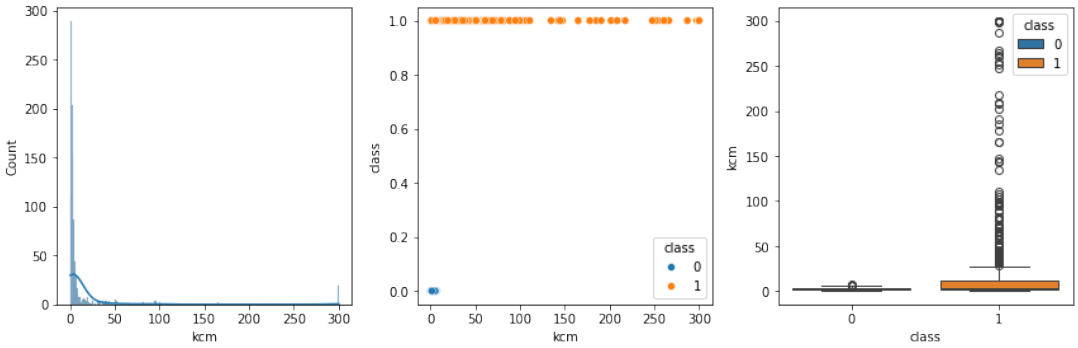


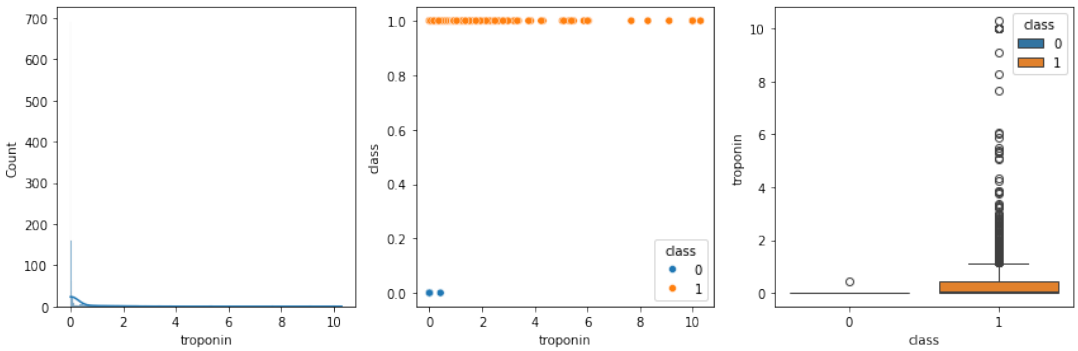










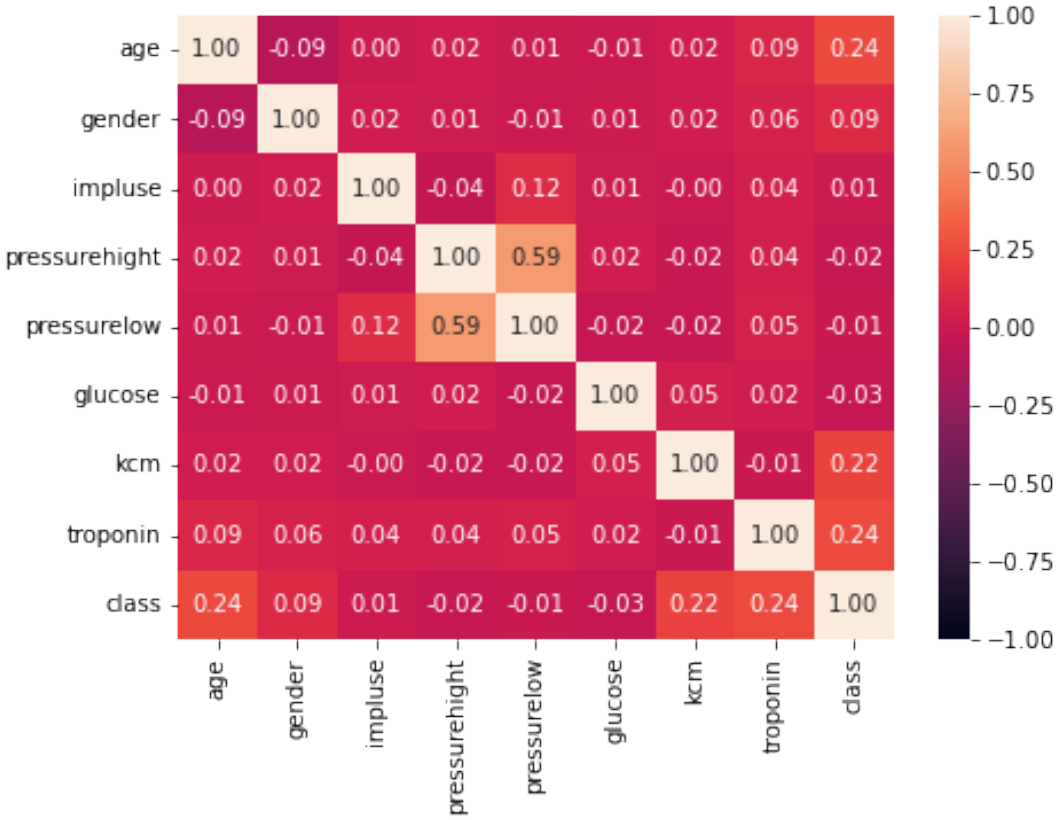


[14]:
```
model_data.describe().style.background_gradient()
```

[14]: <pandas.io.formats.style.Styler at 0x7f5afacddf00>

[15]:
```
plt.figure(figsize=(7,5))
sns.heatmap(model_data.corr(), annot=True, vmin=-1, vmax=1,fmt=".2f")
```

[15]: <AxesSubplot:>



Feature selection: 'gender', 'age', 'impluse', 'pressurehight', 'pressurelow', 'glucose', 'kcm', 'troponin'

[16]:
```
# If you need to drop any other columns, just add it in the [] below
X = model_data.drop(["class"], axis = 1)

y = model_data["class"]

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
#X_train, X_validation, y_train, y_validation = train_test_split(X_train, y_train, test_size=0.1,␣
↪random_state=30)
print('Shape of X_Train set : {}'.format(X_train.shape))
print('Shape of y_Train set : {}'.format(y_train.shape))
print('_'*50)
print('Shape of X_test set : {}'.format(X_test.shape))
print('Shape of y_test set : {}'.format(y_test.shape))
```

```
Shape of X_Train set : (1052, 8)
Shape of y_Train set : (1052,)
--------------------------------------------------
Shape of X_test set : (263, 8)
Shape of y_test set : (263,)
```

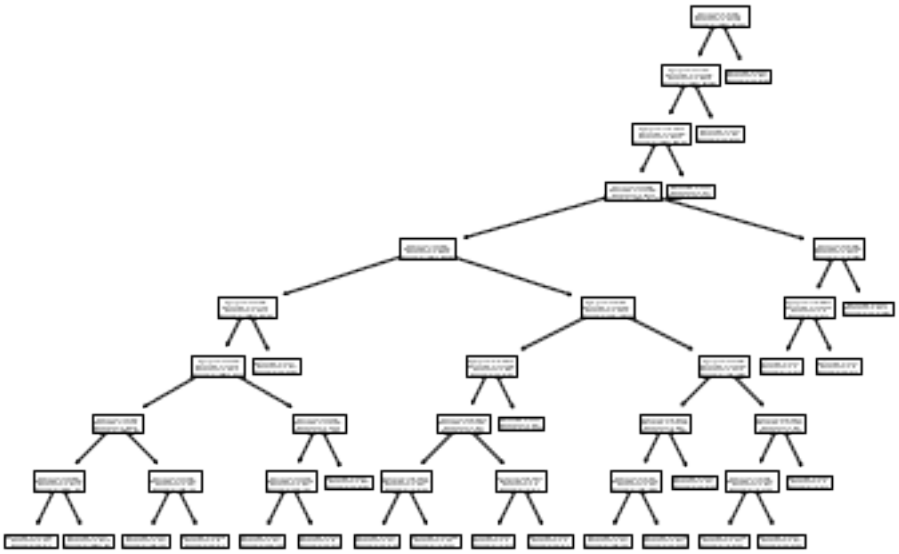
[17]:
```
# Find best parameters for DTs

criterions = ['gini', 'entropy']
best_criterion = str()
splitters = ['best', 'random']
best_splitter = str()
max_depthes = [None, 3, 4, 5, 6, 7, 8, 9]
best_depth = int()
best_acc = 0
```

```
best_recall = 0

for criterion in criterions:
    for splitter in splitters:
        for depth in max_depthes:
            # Modeling
            DTs = tree.DecisionTreeClassifier(criterion=criterion, splitter=splitter, max_depth=depth,␣
↪random_state=0)
            DTs.fit(X_train, y_train)
            y_pred = DTs.predict(X_test)
            # Score
            score = accuracy_score(y_test, y_pred)
            # Recall
            recall = recall_score(y_test, y_pred)
            if (recall > best_recall) and (recall < 0.98):
                best_recall = recall
            # Condition to find best parameters
            if (score > best_acc) and (score < 0.98):
                best_acc = score
                best_criterion = criterion
                best_splitter = splitter
                best_depth = depth
            else:
                continue

print('Best criterion : ', best_criterion)
print('Best splitter : ', best_splitter)
print('Best depth : ', best_depth)
print('Accuracy Score : ', best_acc)
print('Recall Score : ', best_recall)
tree.plot_tree(DTs)
```

```
Best criterion :  entropy
Best splitter :  random
Best depth :  None
Accuracy Score :  0.9771863117870723
Recall Score :  0.9751552795031055
```

[17]:
```
[Text(268.65000000000003, 206.56799999999998, 'X[7] <= 1.509\nentropy =
0.961\nsamples = 1052\nvalue = [405, 647]'),
 Text(257.85, 184.824, 'X[7] <= 0.747\nentropy = 0.978\nsamples = 983\nvalue =
[405, 578]'),
 Text(247.05, 163.07999999999998, 'X[6] <= 72.148\nentropy = 0.989\nsamples =
922\nvalue = [405, 517]'),
 Text(236.25000000000003, 141.336, 'X[7] <= 0.035\nentropy = 0.996\nsamples =
875\nvalue = [405, 470]'),
 Text(159.3, 119.592, 'X[6] <= 4.993\nentropy = 0.963\nsamples = 659\nvalue =
[404, 255]'),
 Text(91.80000000000001, 97.848, 'X[7] <= 0.025\nentropy = 0.771\nsamples =
495\nvalue = [383, 112]'),
 Text(81.0, 76.10399999999998, 'X[7] <= 0.013\nentropy = 0.66\nsamples =
462\nvalue = [383, 79]'),
 Text(43.2, 54.360000000000014, 'X[6] <= 4.632\nentropy = 0.05\nsamples =
358\nvalue = [356, 2]'),
 Text(21.6, 32.615999999999985, 'X[6] <= 0.683\nentropy = 0.029\nsamples =
342\nvalue = [341, 1]'),
 Text(10.8, 10.872000000000014, 'entropy = 0.469\nsamples = 10\nvalue = [9,
1]'),
 Text(32.400000000000006, 10.872000000000014, 'entropy = 0.0\nsamples =
332\nvalue = [332, 0]'),
 Text(64.80000000000001, 32.615999999999985, 'X[7] <= 0.006\nentropy =
0.337\nsamples = 16\nvalue = [15, 1]'),
 Text(54.0, 10.872000000000014, 'entropy = 0.0\nsamples = 10\nvalue = [10, 0]'),
 Text(75.60000000000001, 10.872000000000014, 'entropy = 0.65\nsamples = 6\nvalue
= [5, 1]'),
 Text(118.80000000000001, 54.360000000000014, 'X[7] <= 0.016\nentropy =
0.826\nsamples = 104\nvalue = [27, 77]'),
 Text(108.0, 32.615999999999985, 'X[7] <= 0.014\nentropy = 0.811\nsamples =
36\nvalue = [27, 9]'),
 Text(97.2, 10.872000000000014, 'entropy = 0.0\nsamples = 27\nvalue = [27, 0]'),
 Text(118.80000000000001, 10.872000000000014, 'entropy = 0.0\nsamples = 9\nvalue
= [0, 9]'),
 Text(129.60000000000002, 32.615999999999985, 'entropy = 0.0\nsamples =
```

```
68\nvalue = [0, 68]'),
 Text(102.60000000000001, 76.10399999999998, 'entropy = 0.0\nsamples = 33\nvalue
= [0, 33]'),
 Text(226.8, 97.848, 'X[1] <= 0.531\nentropy = 0.552\nsamples = 164\nvalue =
[21, 143]'),
 Text(183.60000000000002, 76.10399999999998, 'X[6] <= 12.298\nentropy =
0.225\nsamples = 55\nvalue = [2, 53]'),
 Text(172.8, 54.360000000000014, 'X[0] <= 83.939\nentropy = 0.353\nsamples =
30\nvalue = [2, 28]'),
 Text(151.20000000000002, 32.615999999999985, 'X[2] <= 78.128\nentropy =
0.222\nsamples = 28\nvalue = [1, 27]'),
 Text(140.4, 10.872000000000014, 'entropy = 0.0\nsamples = 17\nvalue = [0,
17]'),
 Text(162.0, 10.872000000000014, 'entropy = 0.439\nsamples = 11\nvalue = [1,
10]'),
 Text(194.4, 32.615999999999985, 'X[4] <= 68.445\nentropy = 1.0\nsamples =
2\nvalue = [1, 1]'),
 Text(183.60000000000002, 10.872000000000014, 'entropy = 0.0\nsamples = 1\nvalue
= [1, 0]'),
 Text(205.20000000000002, 10.872000000000014, 'entropy = 0.0\nsamples = 1\nvalue
= [0, 1]'),
 Text(194.4, 54.360000000000014, 'entropy = 0.0\nsamples = 25\nvalue = [0,
25]'),
 Text(270.0, 76.10399999999998, 'X[7] <= 0.008\nentropy = 0.667\nsamples =
109\nvalue = [19, 90]'),
 Text(248.4, 54.360000000000014, 'X[6] <= 23.379\nentropy = 0.825\nsamples =
58\nvalue = [15, 43]'),
 Text(237.60000000000002, 32.615999999999985, 'X[6] <= 6.157\nentropy =
0.911\nsamples = 46\nvalue = [15, 31]'),
 Text(226.8, 10.872000000000014, 'entropy = 0.0\nsamples = 13\nvalue = [13,
0]'),
 Text(248.4, 10.872000000000014, 'entropy = 0.33\nsamples = 33\nvalue = [2,
31]'),
 Text(259.20000000000005, 32.615999999999985, 'entropy = 0.0\nsamples =
12\nvalue = [0, 12]'),
 Text(291.6, 54.360000000000014, 'X[0] <= 89.266\nentropy = 0.397\nsamples =
51\nvalue = [4, 47]'),
 Text(280.8, 32.615999999999985, 'X[6] <= 7.605\nentropy = 0.327\nsamples =
50\nvalue = [3, 47]'),
 Text(270.0, 10.872000000000014, 'entropy = 0.779\nsamples = 13\nvalue = [3,
10]'),
 Text(291.6, 10.872000000000014, 'entropy = 0.0\nsamples = 37\nvalue = [0,
37]'),
 Text(302.40000000000003, 32.615999999999985, 'entropy = 0.0\nsamples = 1\nvalue
= [1, 0]'),
 Text(313.20000000000005, 119.592, 'X[3] <= 82.68\nentropy = 0.043\nsamples =
216\nvalue = [1, 215]'),
 Text(302.40000000000003, 97.848, 'X[3] <= 74.962\nentropy = 0.811\nsamples =
4\nvalue = [1, 3]'),
 Text(291.6, 76.10399999999998, 'entropy = 0.0\nsamples = 1\nvalue = [1, 0]'),
 Text(313.20000000000005, 76.10399999999998, 'entropy = 0.0\nsamples = 3\nvalue
= [0, 3]'),
 Text(324.0, 97.848, 'entropy = 0.0\nsamples = 212\nvalue = [0, 212]'),
 Text(257.85, 141.336, 'entropy = 0.0\nsamples = 47\nvalue = [0, 47]'),
 Text(268.65000000000003, 163.07999999999998, 'entropy = 0.0\nsamples =
61\nvalue = [0, 61]'),
 Text(279.45000000000005, 184.824, 'entropy = 0.0\nsamples = 69\nvalue = [0,
69]')]
```



[18]:
```
n_estimators = [10, 50, 100, 250, 500]
criterions = ['gini', 'entropy']
max_depthes = [None, 2,  4, 6, 8]
best_acc = 0

for estimator in n_estimators:
    for criterion in criterions:
        for depth in max_depthes:
            # Modeling
            RF = RandomForestClassifier(n_estimators=estimator, criterion=criterion,
                                        max_depth=depth, n_jobs=-1)
            RF.fit(X_train, y_train)
            y_pred = RF.predict(X_test)
            # Score
            score = accuracy_score(y_test, y_pred)
            # Condition to find best parameters
            if (score > best_acc) and (score < 0.98): # Condition to avoide overfitting
                best_acc = score
                best_estimator = estimator
                best_criterion = criterion
                best_depth = depth

print('Best Criterion : ', best_criterion)
print('Best estimator : ', best_estimator)
print('Best depth : ', best_depth)
print('Accuracy Score : ', best_acc)
```

```
Best Criterion :  gini
Best estimator :  10
Best depth :  4
Accuracy Score :  0.9771863117870723
```

Criterion - Gini Impurity: Gini impurity measures the probability of misclassifying a randomly chosen element in a dataset. It ranges from 0 (perfectly pure, all elements belong to one class) to 0.5 (completely impure, elements are equally distributed across all classes).

- Entropy (Information Gain): Entropy measures the amount of disorder or uncertainty in a dataset. It ranges from 0 (perfectly pure, all elements belong to one class) to 1 (completely impure, elements are evenly distributed across all classes).

Selection method 1. Entropy might be a little slower to compute (because it makes use of the logarithm); It only matters in 2% of the cases whether you use gini impurity or entropy. source: Theoretical comparison between the gini index and information gain criteria

2. Try both Gini impurity and entropy as splitting criteria and evaluate the performance of tree using cross-validation or other suitable evaluation metrics. Select the one that gives the better results on your specific dataset.

[19]:
```
# Find best parameters for KNN
best_acc = 0

for k in range(3, 15, 1) :
    knn = KNeighborsClassifier(n_neighbors=k, n_jobs=-1).fit(X_train, y_train)
    y_pred = knn.predict(X_test)
    score = accuracy_score(y_test, y_pred)
    if score > best_acc :
        best_acc = score
        best_k = k
print('Best k :', best_k)
print('score : ', best_acc)
```

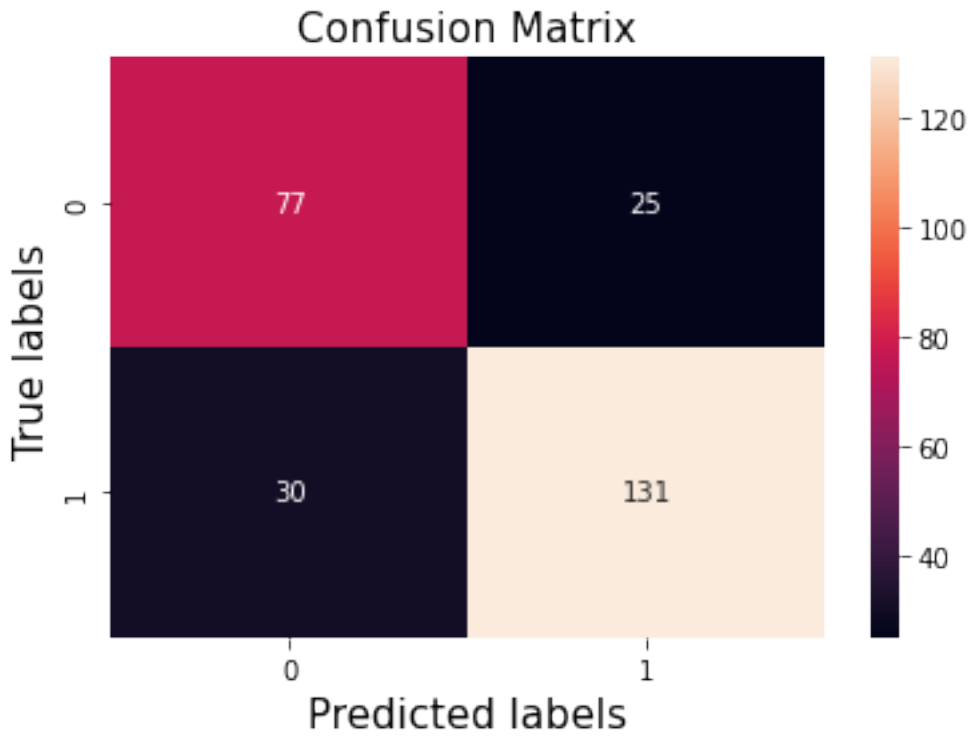
```
Best k : 11
score :  0.6768060836501901
```

K-Nearest Neighbors (KNN) is a non-parametric, instance-based machine learning algorithm that doesn't use a traditional loss function like many other algorithms. Instead, KNN makes predictions based on the majority class of the k-nearest neighbors to a given data point.

[20]:
```
def generate_confusion_matrix(y_true, y_pred):
    # visualize the confusion matrix
    ax = plt.subplot()
    c_mat = confusion_matrix(y_true, y_pred)
    sns.heatmap(c_mat, annot=True, fmt='g', ax=ax)

    ax.set_xlabel('Predicted labels', fontsize=15)
    ax.set_ylabel('True labels', fontsize=15)
    ax.set_title('Confusion Matrix', fontsize=15)
```

[21]:
```
clf_2 = LogisticRegression(solver='liblinear', max_iter=200)
clf_2.fit(X_train,y_train)
y_pred = clf_2.predict(X_test)
accuracy = accuracy_score(y_test, y_pred)

print(f"Prediction accuracy: {100*accuracy:.2f}%")
generate_confusion_matrix(y_test, y_pred)
plt.show()
```

```
Prediction accuracy: 79.09%
```



Appendix 2

October 11, 2023

[1]:
```
import numpy as np                    # import numpy package under shorthand "np"
import pandas as pd                   # import pandas package under shorthand "pd"
import matplotlib.pyplot as plt
import seaborn as sns
# Regression import

%config Completer.use_jedi = False  # enable code auto-completion

from sklearn.model_selection import train_test_split
from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import accuracy_score
from sklearn.metrics import recall_score, confusion_matrix
from sklearn.ensemble import RandomForestClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.svm import SVC
from sklearn.neighbors import KNeighborsClassifier
from sklearn import tree

import sklearn
```

[2]:
```
# Read data
rawdata = pd.read_csv("stroke.csv")
# Show data examples
rawdata.sample(10)
len(rawdata[rawdata["stroke"]==1])
```

[2]: 249

[3]:
```
#Delete NaN data
rawdata=rawdata.dropna(axis=0)
#Delete rows where "smoking_status" is Unknown
rawdata=rawdata[rawdata["smoking_status"] != "Unknown"]
rawdata=rawdata[rawdata["gender"] != "Other"]
```

[4]:
```
condition = (rawdata["stroke"]==0)
num_samples = len(rawdata[rawdata["stroke"]==1])  # Change to the desired number of random samples
rng = np.random.default_rng(seed=44)
random_indices = rng.choice(rawdata[condition].index, size = num_samples)
```

[5]:
```
#random_indices = np.random.choice(rawdata[condition].index, num_samples, replace=False)
random_indices
```

[5]:
```
array([3490,  821, 4248, 1479, 3485, 2206,  666, 4952, 2839, 1005, 5069,
       4438,  338, 1015,  724, 1878, 2195, 3542, 3822, 3239, 2385, 4887,
       2064, 2233, 2189, 4810, 5036, 4752, 3085, 3725, 2234,  345, 3118,
       4144,  759, 4638, 2885, 2780, 4423, 1176, 4950, 1092, 3860, 4376,
       3426,  661, 3603, 2901, 5057, 1204, 1294, 1689, 4343, 4991, 4947,
       1571, 3716, 3568, 1770, 3997, 3444, 1630, 1905,  929, 2181, 4017,
       3088, 2444, 4326, 1628, 1129, 1247, 2362, 5019, 2931, 3394, 4358,
       2777, 3017,  784, 1657, 4691, 3856, 1853, 3159, 4114,  505, 4661,
       1553, 1216, 4648, 4996, 1030, 4608, 2910, 2990, 2534, 3771, 3123,
       1575,  855, 3767, 2005, 2328, 4942, 1813,  675, 3631, 4199, 2102,
       4480, 3668, 1334, 4412, 2723, 1542, 3524, 1221, 1898, 1786, 1323,
       2292, 5036, 3791, 3456, 2360, 3388, 3612, 1088, 5080,  969, 1354,
       2098, 3954,  972, 2559, 4739, 2265, 4556, 3172, 4190,  800, 2160,
       1760, 2251, 1557, 2130, 3210, 2520, 1320, 1435,  471,  257,  436,
       1543, 4029, 4932, 3569, 1177, 4400,  952, 3540,  719,  500, 1882,
       3650, 3190, 2904,  972, 4919, 4441, 1339, 4192,  622, 1181, 4317,
       3984, 2287, 1916, 2526])
```

[6]:
```
#random_indices = np.random.choice(rawdata[condition].index, num_samples, replace=False, random_indices)
selected_rows_0 = rawdata.loc[random_indices]
selected_rows_1 = rawdata[rawdata["stroke"]==1]

print(len(selected_rows_0))
print(len(selected_rows_1))
```

```
180
180
```

[7]:
```
len(rawdata[rawdata["stroke"]==1])
rawdata = pd.concat([selected_rows_0, selected_rows_1], ignore_index=True)
rawdata
```

[7]:

	id	gender	age	hypertension	heart_disease	ever_married	work_type	Residence_type	avg_glucose_level	bmi	smoking_status	stroke
0	17745	Male	79.0	1	0	Yes	Self-employed	Urban	84.88	28.7	formerly smoked	0
1	25458	Female	70.0	1	0	Yes	Govt_job	Rural	88.66	36.7	formerly smoked	0
2	15422	Male	31.0	0	0	No	Govt_job	Rural	80.57	28.2	formerly smoked	0
3	43059	Female	71.0	0	0	Yes	Self-employed	Rural	151.30	26.3	never smoked	0
4	33960	Male	39.0	1	0	Yes	Self-employed	Urban	71.66	28.7	never smoked	0
..	...	...	...	...	...	...	...	...	...	...	...	...
355	10548	Male	66.0	0	0	Yes	Private	Rural	76.46	21.2	formerly smoked	1
356	52282	Male	57.0	0	0	Yes	Private	Rural	197.28	34.5	formerly smoked	1
357	45535	Male	68.0	0	0	Yes	Private	Rural	233.94	42.4	never smoked	1
358	40460	Female	68.0	1	1	Yes	Private	Urban	247.51	40.5	formerly smoked	1
359	27153	Female	75.0	0	0	Yes	Self-employed	Rural	78.80	29.3	formerly smoked	1

360 rows x 12 columns

After examining the dataset, we think some preliminary processing should be done 1. In this report, only man and woman(physically) will be discussed, so the rows contain "Other" value will be excludes 2. In addition, there is "Unknown" category in smoking status which is not suitable for trainning. Rows containing "Unknown" smoking status will be excluded as well.

[8]:
```
# Create 2 copied of dataset
# visual_data: used for visualization
# model_data: used for training and testing
visual_data = rawdata.copy(deep=True)
model_data = rawdata.copy(deep=True)
```

[9]:
```
rawdata.shape
```

[9]:
```
(360, 12)
```

This block turns all the categorical features into numeric values instead of texts (only work for `model_data`)

[10]:
```
map_columns = ["gender", "ever_married", "work_type", "Residence_type", "smoking_status"]
gender_map = {"Female": 0, "Male": 1}
married_map = {"No": 0, "Yes": 1}
work_map = {"Never_worked": 0, "Private": 1, "Govt_job" : 2, "children" : 3, "Self-employed" : 4}
residence_map = {"Rural": 0, "Urban": 1}
smoking_map = {"never smoked": 0, "formerly smoked": 1, "smokes" : 2}

maps = [gender_map, married_map, work_map, residence_map, smoking_map]


for label, m in zip(map_columns, maps):
    diago = model_data[label].copy(deep=True)
    diago = diago.map(m).copy(deep=True)
    model_data[label] = diago.copy(deep=True)

model_data.sample(5)
```

[10]:

	id	gender	age	hypertension	heart_disease	ever_married	work_type	Residence_type	avg_glucose_level	bmi	smoking_status	stroke
178	66490	1	42.0	1	0	1	2	1	118.82	41.0	2	0
50	59157	1	73.0	1	0	1	1	1	88.34	27.5	0	0
52	35210	0	48.0	0	0	1	1	1	112.96	25.4	0	0
121	36589	0	61.0	0	0	1	4	1	180.80	20.3	0	0
61	3154	0	81.0	0	0	1	4	0	114.88	18.3	1	0

[11]:
```
model_data.describe()
```

[11]:

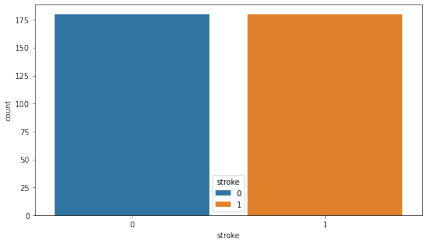
	id	gender	age	hypertension	heart_disease	ever_married	work_type	Residence_type	avg_glucose_level	bmi	smoking_status	stroke
count	360.000000	360.000000	360.000000	360.000000	360.000000	360.000000	360.000000	360.000000	360.000000	360.000000	360.000000	360.000000
mean	37204.183333	0.422222	57.455556	0.213889	0.133333	0.808333	1.950000	0.527778	120.478194	30.235278	0.747222	0.500000
std	21466.126112	0.494601	19.172817	0.410620	0.340408	0.394160	1.298274	0.499923	56.441852	6.986254	0.810786	0.500696
min	156.000000	0.000000	11.000000	0.000000	0.000000	0.000000	1.000000	0.000000	56.110000	16.400000	0.000000	0.000000
25%	19083.750000	0.000000	45.000000	0.000000	0.000000	1.000000	1.000000	0.000000	78.890000	25.375000	0.000000	0.000000
50%	36729.500000	0.000000	60.000000	0.000000	0.000000	1.000000	1.000000	1.000000	97.555000	29.000000	1.000000	0.500000
75%	56576.750000	1.000000	74.000000	0.000000	0.000000	1.000000	4.000000	1.000000	157.497500	33.700000	1.000000	1.000000
max	72594.000000	1.000000	82.000000	1.000000	1.000000	1.000000	4.000000	1.000000	271.740000	57.700000	2.000000	1.000000

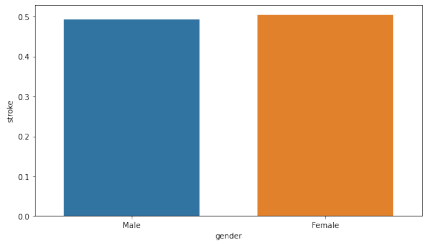
[12]:
```
# preprocessing2: Data visualization and analysis
# Seperate columns into categorical(discrete) and numerical(continuous)
```

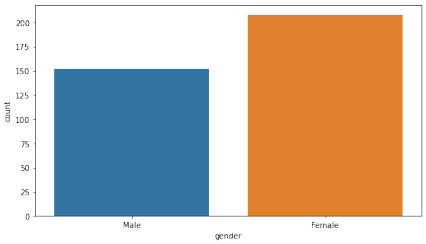
```
# All features: 'gender', 'age', 'impluse', 'pressurehight', 'pressurelow', 'glucose', 'kcm', 'troponin'
categoric_columns = ["gender", "hypertension", "heart_disease", "ever_married", "work_type",␣
↪"Residence_type", "smoking_status"]
numeric_columns = ['age', 'avg_glucose_level', 'bmi']
```

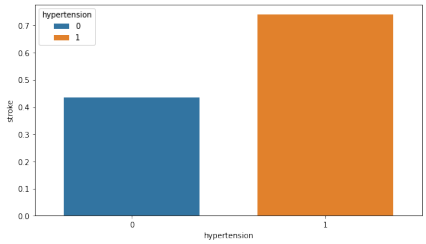
This block shows 1. the relationship between stroke and all categorical features respectively 2. the distribution of all categorical features 3. the count plot of stroke

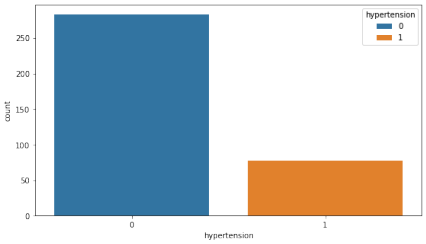
[13]:
```
# Gender(the only categorical feature)
fig, axes = plt.subplots(8, 2, figsize=(20, 48))
# Stroke count
sns.countplot(data=visual_data, x='stroke', ax=axes[0][0], hue="stroke", dodge=False)
axes[0][1].axis('off')

for i, c in enumerate(categoric_columns):
    # Gender vs Class
    sns.barplot(x=c, y="stroke", data=visual_data, width=0.7, errorbar=None, hue=c, dodge=False,␣
↪ax=axes[i+1][0])
    # Gender count
    sns.countplot(data=visual_data, x=c, ax=axes[i+1][1], hue=c, dodge=False)


plt.show()
```

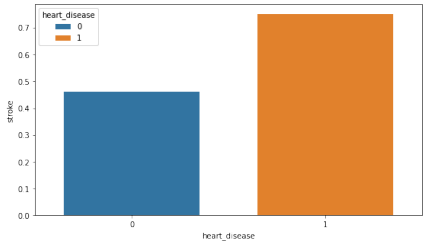

stroke
0
1
count
stroke

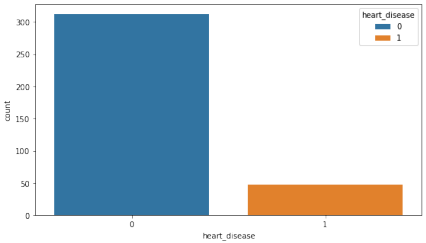

stroke
Male
Female
gender

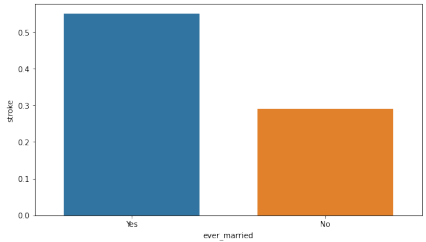

count
Male
Female
gender

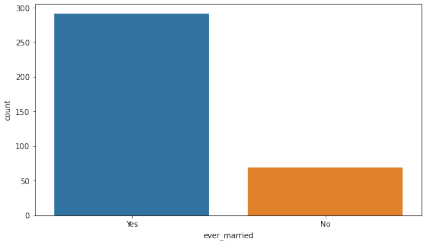

hypertension
0
1
stroke
hypertension

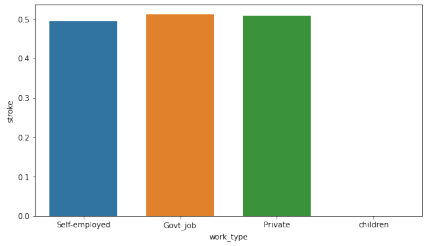

hypertension
0
1
count
hypertension

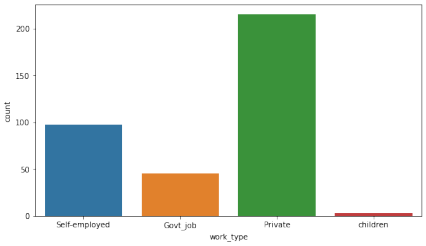

heart_disease
0
1
stroke
heart_disease

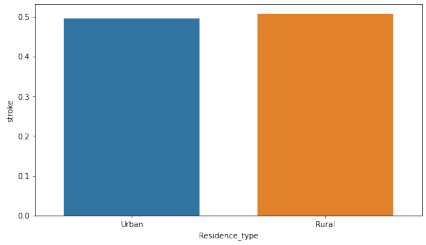

heart_disease
0
1
count
heart_disease

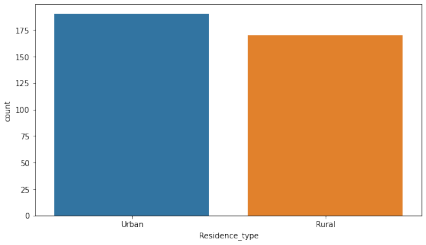

stroke
Yes
No
ever_married

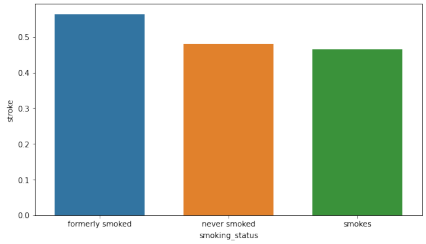

count
Yes
No
ever_married

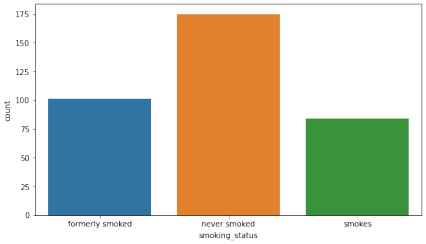

stroke
Self-employed
Govt_job
Private
children
work_type


count
Self-employed
Govt_job
Private
children
work_type

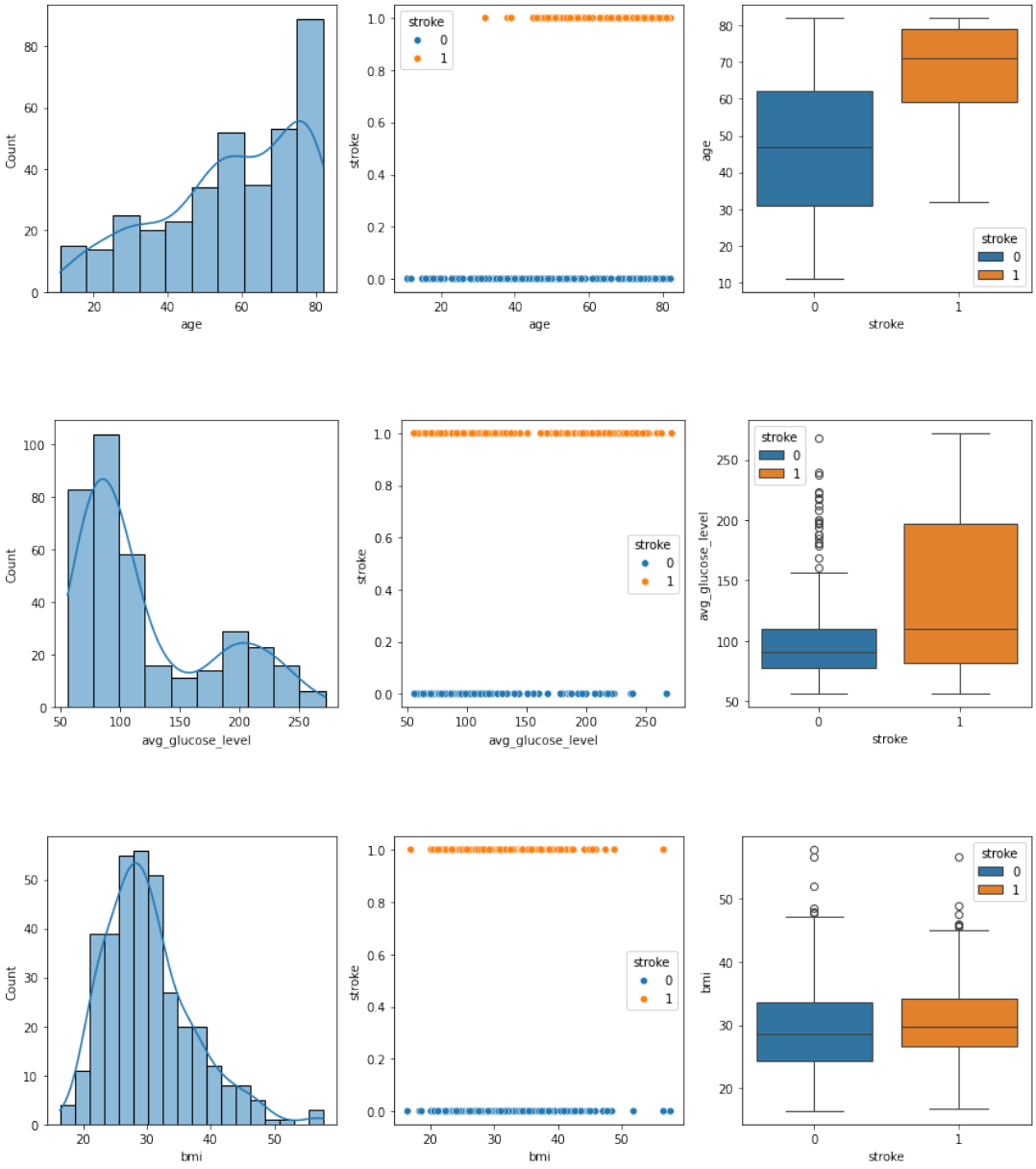

stroke
Urban
Rural
Residence_type


count
Urban
Rural
Residence_type


stroke
formerly smoked
never smoked
smokes
smoking_status


count
formerly smoked
never smoked
smokes
smoking_status

[14]:
```
# numeric data visualization
for index, label in enumerate(numeric_columns):
    fig, axes = plt.subplots(1, 3, figsize=(12,4))
    # distribution
    sns.histplot(data=visual_data, x=label, ax=axes[0], kde=True)
    # "label" VS class
    sns.scatterplot(data=visual_data, x=label, y="stroke", ax=axes[1], hue="stroke")
    # boxplot
    sns.boxplot(data=visual_data, x='stroke', y=label, ax=axes[2], hue="stroke", dodge=False)
    # Adjusting the layout for better visualization
    plt.tight_layout()
    plt.show()
```



According to the plotsthere is not extreme data in sight but some peripheral element. They will be ruled out by the code block below

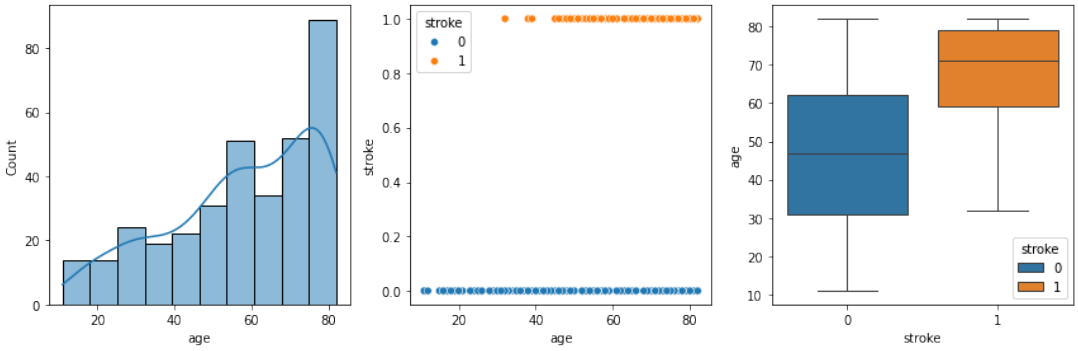
Outlier detection In a relatively small dataset of more than samples, outliers can have a more significant impact on statistical analyses or machine learning models compared to larger datasets.

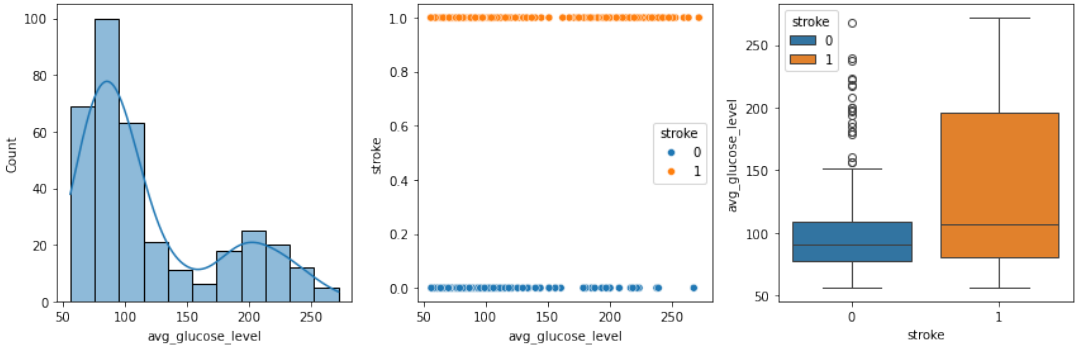
When we examine the boxplots, we can see that there are some outlier values, although not too many. We will clean these in the next step using IQR(Interquartile Range) method

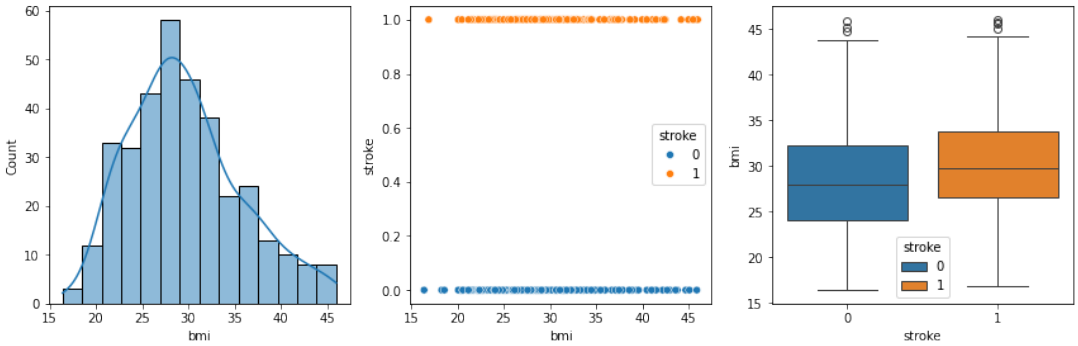
[15]:
```
for i in numeric_columns:
    Q1 = model_data[i].quantile(0.25)
    Q3 = model_data[i].quantile(0.75)
    IQR = Q3 - Q1
    lower_bound = Q1 - 1.5 * IQR
    upper_bound = Q3 + 1.5 * IQR
    print(f'Original shape: {model_data.shape}')
    model_data = model_data[(model_data[i] >= lower_bound) & (model_data[i] <= upper_bound)]
    print(f'Current shape: {model_data.shape}')
    print()
visual_data = model_data.copy(deep=True)
```

```
Original shape: (360, 12)
Current shape: (360, 12)

Original shape: (360, 12)
Current shape: (360, 12)

Original shape: (360, 12)
Current shape: (350, 12)
```

[16]:
```
# numeric data visualization
for index, label in enumerate(numeric_columns):
    fig, axes = plt.subplots(1, 3, figsize=(12,4))
    # distribution
    sns.histplot(data=visual_data, x=label, ax=axes[0], kde=True)
    # "label" VS class
    sns.scatterplot(data=visual_data, x=label, y="stroke", ax=axes[1], hue="stroke")
    # boxplot
    sns.boxplot(data=visual_data, x='stroke', y=label, ax=axes[2], hue="stroke", dodge=False)
    # Adjusting the layout for better visualization
    plt.tight_layout()
    plt.show()
```







Feature selection: According to the plots, `residence_type` does not contribute much to the occurrence of stroke, so it will be excluded from the dataset **Final features**: `gender, age, hypertension, heart_disease, ever_married, work_type, avg_glucose_level, bmi, smoking_status`

[17]:
```
# If you need to drop any other columns, just add it in the [] below
X = model_data.drop(["stroke", "Residence_type", "id"], axis = 1)

y = model_data["stroke"]

X_train, X_temp, y_train, y_temp = train_test_split(X, y, test_size=0.2, random_state=46)
#80% train, 10% testing, 10%validation
X_val, X_test, y_val, y_test = train_test_split(X_temp, y_temp, test_size=0.5, random_state=46)

print('Shape of X_Train set : {}'.format(X_train.shape))
print('Shape of y_Train set : {}'.format(y_train.shape))
print('_'*50)
print('Shape of X_test set : {}'.format(X_test.shape))
print('Shape of y_test set : {}'.format(y_test.shape))
print('_'*50)
print('Shape of X_val set : {}'.format(X_val.shape))
print('Shape of y_val set : {}'.format(y_val.shape))
```

```
Shape of X_Train set : (280, 9)
Shape of y_Train set : (280,)
--------------------------------------------------
Shape of X_test set : (35, 9)
Shape of y_test set : (35,)
--------------------------------------------------
Shape of X_val set : (35, 9)
Shape of y_val set : (35,)
```

[18]:
```
def generate_confusion_matrix(y_true, y_pred):
    # visualize the confusion matrix
    ax = plt.subplot()
    c_mat = confusion_matrix(y_true, y_pred)
    sns.heatmap(c_mat, annot=True, fmt='g', ax=ax)

    ax.set_xlabel('Predicted labels', fontsize=15)
    ax.set_ylabel('True labels', fontsize=15)
    ax.set_title('Confusion Matrix', fontsize=15)
```

[19]:
```
# Find best parameters for DTs

criterions = ['gini', 'entropy']
best_criterion = str()
splitters = ['best', 'random']
best_splitter = str()
max_depthes = [None, 3, 4, 5, 6, 7, 8, 9]
best_depth = int()
best_acc = 0
best_recall = 0

for criterion in criterions:
    for splitter in splitters:
        for depth in max_depthes:
            # Modeling
```

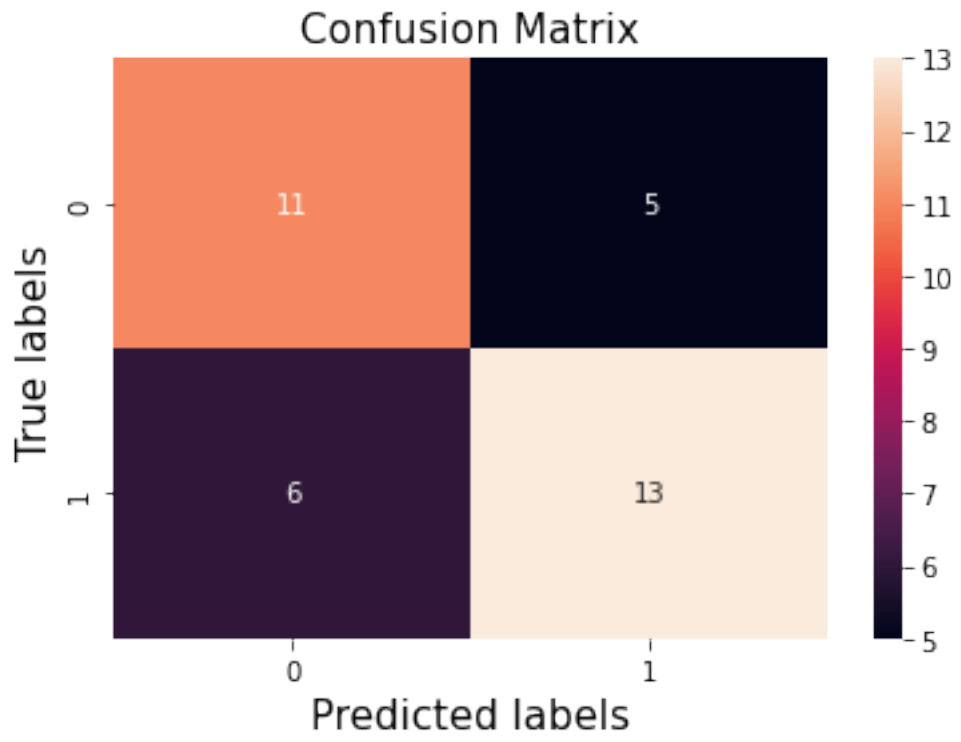
```
            DTs = tree.DecisionTreeClassifier(criterion=criterion, splitter=splitter, max_depth=depth,␣
↪random_state=0)
            DTs.fit(X_train, y_train)
            y_pred = DTs.predict(X_val)
            # Score
            score = accuracy_score(y_val, y_pred)
            # Recall
            recall = recall_score(y_val, y_pred)
            if (recall > best_recall):
                best_recall = recall
            # Condition to find best parameters
            if (score > best_acc) and (score < 0.98):
                best_acc = score
                best_criterion = criterion
                best_splitter = splitter
                best_depth = depth
            else:
                continue

print('Best criterion : ', best_criterion)
print('Best splitter : ', best_splitter)
print('Best depth : ', best_depth)
print('Accuracy Score : ', best_acc)
print("Recall: ", best_recall)
#tree.plot_tree(DTs)
```

```
Best criterion :  gini
Best splitter :  best
Best depth :  None
Accuracy Score :  0.8285714285714286
Recall:  0.9285714285714286
```

[20]:
```
print(f"Using criterion: {best_criterion}, spliter: {best_splitter}, depth: {best_depth}.")
DTs = tree.DecisionTreeClassifier(criterion=best_criterion, splitter=best_splitter,␣
↪max_depth=best_depth, random_state=0)
DTs.fit(X_train, y_train)
y_pred = DTs.predict(X_test)
accuracy_score(y_test, y_pred)

print(f"recall: {recall_score(y_test, y_pred)}")
print(f"accuracy: {accuracy_score(y_test, y_pred)}")
```

```
Using criterion: gini, spliter: best, depth: None.
recall: 0.6842105263157895
accuracy: 0.6857142857142857
```

[21]:
```
generate_confusion_matrix(y_test, y_pred)
plt.savefig('1.jpg', dpi = 300)
```



[22]:
```
n_estimators = [10, 50, 100, 250, 500]
criterions = ['gini', 'entropy']
max_depthes = [None, 2,  4, 6, 8]
best_acc = 0

for estimator in n_estimators:
    for criterion in criterions:
        for depth in max_depthes:
            # Modeling
            RF = RandomForestClassifier(n_estimators=estimator, criterion=criterion,
                                        max_depth=depth, n_jobs=-1)
            RF.fit(X_train, y_train)
            y_pred = RF.predict(X_val)
            # Score
            score = accuracy_score(y_val, y_pred)
            # Condition to find best parameters
            if (score > best_acc) and (score < 0.98): # Condition to avoide overfitting
                best_acc = score
                best_estimator = estimator
                best_criterion = criterion
                best_depth = depth

print('Best Criterion : ', best_criterion)
print('Best estimator : ', best_estimator)
print('Best depth : ', best_depth)
print('Accuracy Score : ', best_acc)
```

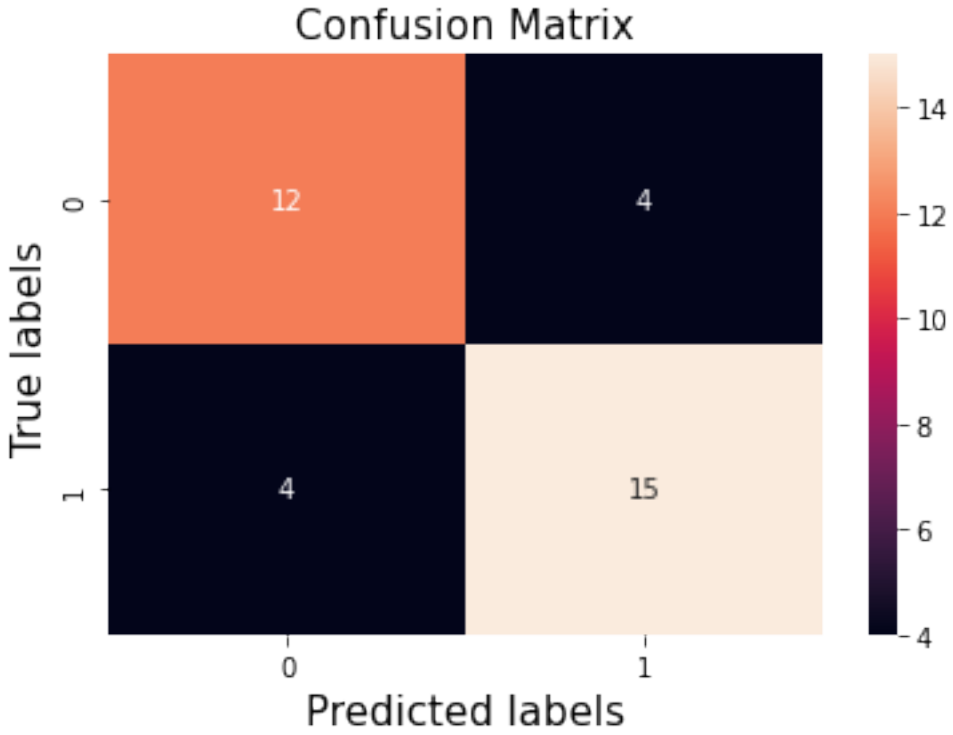
```
Best Criterion :  gini
Best estimator :  50
Best depth :  None
Accuracy Score :  0.8857142857142857
```

[23]:
```
print(f"Using criterion: {best_criterion}, estimator: {best_estimator}, depth: {best_depth}.")
RF = RandomForestClassifier(n_estimators=best_estimator, criterion=best_criterion,␣
↪max_depth=best_depth, random_state=0)
RF.fit(X_train, y_train)
y_pred = RF.predict(X_test)
print(f"recall: {recall_score(y_test, y_pred)}")
print(f"accuracy: {accuracy_score(y_test, y_pred)}")
```

```
#recall_score(y_test, y_pred)
```

```
Using criterion: gini, estimator: 50, depth: None.
recall: 0.7894736842105263
accuracy: 0.7714285714285715
```

[24]:
```
generate_confusion_matrix(y_test, y_pred)
plt.savefig('2.jpg', dpi = 300)
```



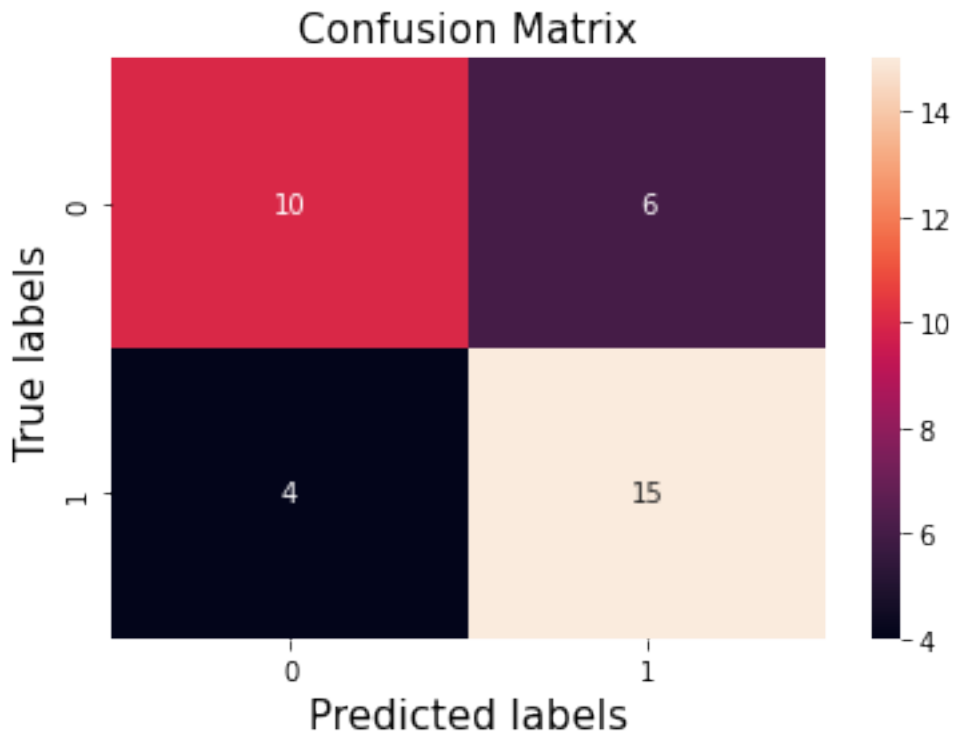
[25]:
```
# Find best parameters for KNN
best_acc = 0

for k in range(3, 15, 1) :
    knn = KNeighborsClassifier(n_neighbors=k, n_jobs=-1).fit(X_train, y_train)
    y_pred = knn.predict(X_val)
    score = accuracy_score(y_val, y_pred)
    if score > best_acc :
        best_acc = score
        best_k = k
print('Best k :', best_k)
print('score : ', best_acc)
```

```
Best k : 12
score :  0.8571428571428571
```

[26]:
```
print(f"Using k: {best_k}")
KNN = KNeighborsClassifier(n_neighbors=best_k, n_jobs=-1)
KNN.fit(X_train, y_train)
y_pred = KNN.predict(X_test)
print(f"recall: {recall_score(y_test, y_pred)}")
print(f"accuracy: {accuracy_score(y_test, y_pred)}")
```

```
Using k: 12
recall: 0.7894736842105263
accuracy: 0.7142857142857143
```

[27]:
```
generate_confusion_matrix(y_test, y_pred)
plt.savefig('3.jpg', dpi = 300)
```

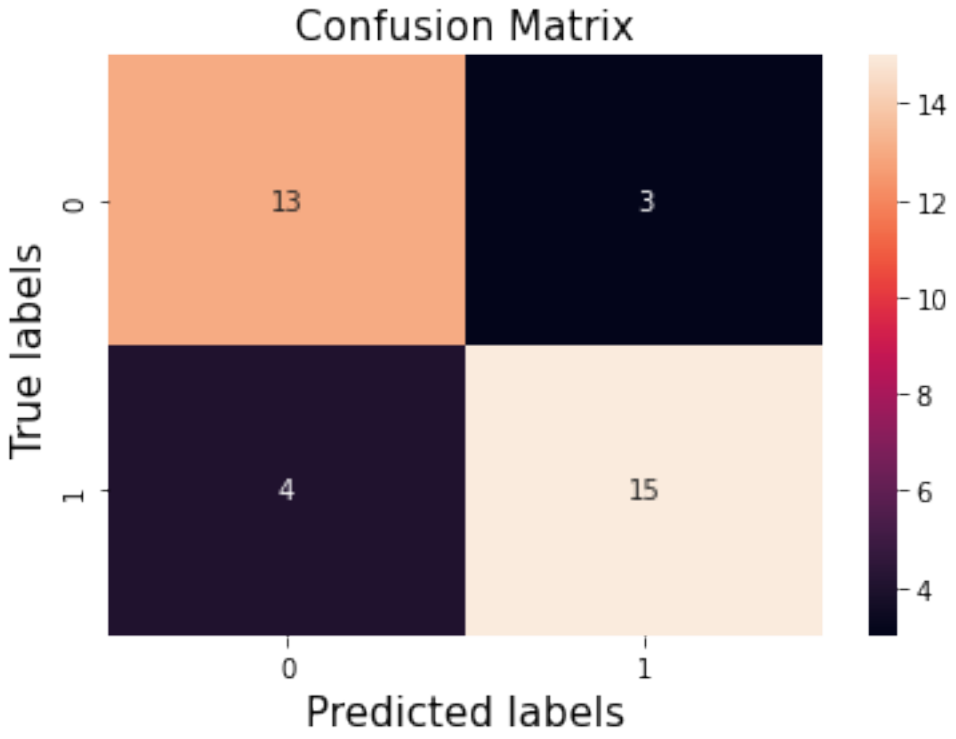


[28]:
```
clf_2 = LogisticRegression(solver='liblinear', max_iter=200)
clf_2.fit(X_train,y_train)
y_pred = clf_2.predict(X_val)
accuracy = accuracy_score(y_val, y_pred)
print(f"Prediction accuracy: {100*accuracy:.2f}%")
```

```
Prediction accuracy: 88.57%
```

[29]:
```
y_pred = clf_2.predict(X_test)
accuracy = accuracy_score(y_test, y_pred)
print(f"recall: {recall_score(y_test, y_pred)}")
print(f"Prediction accuracy: {100*accuracy:.2f}%")
generate_confusion_matrix(y_test, y_pred)
plt.savefig('4.jpg', dpi = 300)
plt.show()
```

```
recall: 0.7894736842105263
Prediction accuracy: 80.00%
```


Confusion Matrix
13
3
4
15
0
1
True labels
0
1
Predicted labels
14
12
10
8
6
4