Aalto University

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.

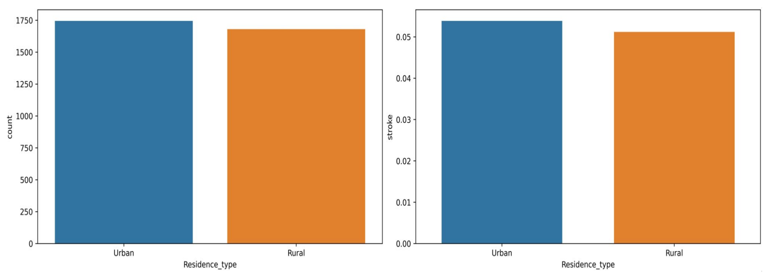


Figure 1: Total count of each residence type (left) and their impact on the probability of having a stroke (right).

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.

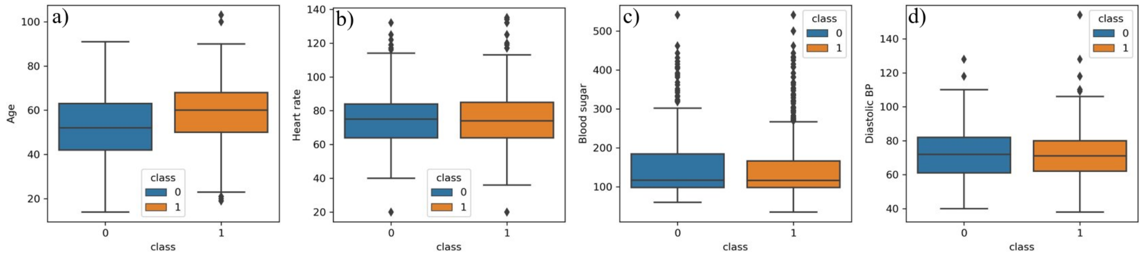


Figure 2: Examples of data visualisation showing the range of a) age, b) heart rate, c) blood sugar, and d) diastolic BP for each class using box plots (class 0 correspond to absence and 1 to presence of heart attack).

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. The outliers were cleaned in the next step using IQR(Interquartile Range) method. After processing the dataset, it now contains 2882 data points.

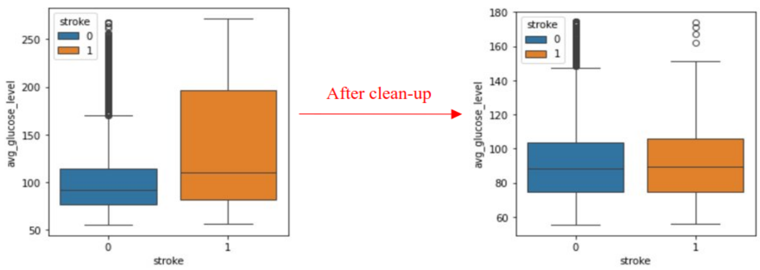


Figure 3: Examples of data visualisation showing the effect of cleaning outliers.

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, the reason behind choosing the splitting method was the same. However, the process itself had one difference. Here, 80% of the data (2305 points) were randomly sampled and allocated to the training set, while the remaining 20% were randomly designated for the validation (288 points) and test (289 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 6 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.

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%

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.

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 KNN, Decision Tree, and Random Forest). Hence, test accuracy is reported for all models, as the variation in their validation errors is negligible. Additionally, the test accuracy of the models remains closely matched (similar for Logistic Regression, KNN, and Decision Tree), indicating that all four models yield comparable outcomes, as evident in their confusion matrix (Figure 4).

Table 4: Accuracy of each model for dataset 2.

Model	Validation accuracy	Test accuracy
Logistic Regression	95.83%	96.89%
KNN	96.18%	96.89%
Decision Trees	96.18%	96.89%
Random Forest	96.18%	96.19%

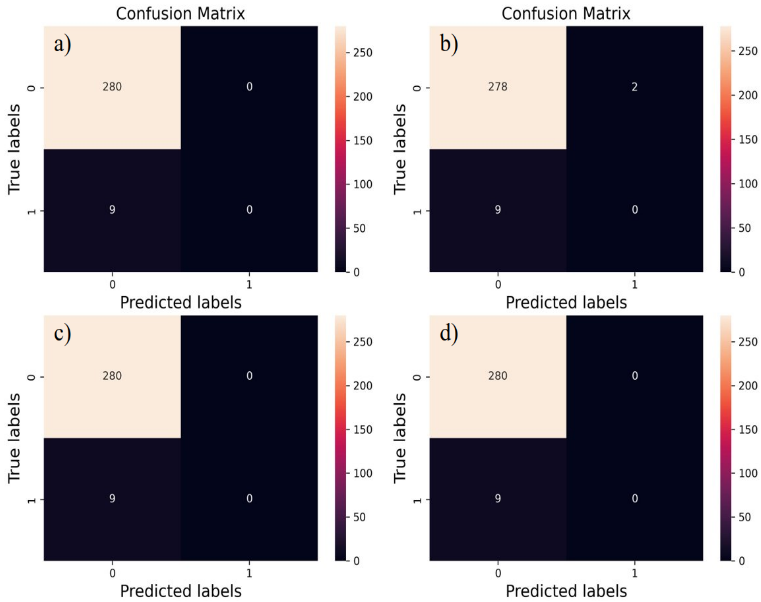


Figure 4: Confusion matrix for a) decision tres, b) random forest, c) KNN, and d) logistic regression.

Therefore, opting for either method will lead to the same, highly accurate result. Nevertheless, it's worth noting that the KNN and Decision Tree methods are more preferable, given their slightly superior validation accuracy (0.35% higher than Logistic Regression) and test accuracy (0.7% higher than Random Forest).

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 and test accuracy, posing a challenge in selecting the best model. Nonetheless, the KNN and Decision Tree methods displayed slightly higher accuracy, rendering them marginally more suitable. Furthermore, the proximity of validation and test accuracy across all methods suggests the absence of overfitting in the models. It's noteworthy that in dataset 1, the Decision Tree and Random Forest methods demonstrated the highest validation accuracy among the four methods.

For future work, collecting more data points could 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 10, 2023

[1]:
```python
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]:
```python
# 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]:
```python
#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  \
1178   66       1       73            125           78    112.0   2.87
64     61       1      102            130           83    201.0   1.24
827    50       1       66            112           74    146.0  10.11
25     72       1       64            106           68    111.0   2.11
198    50       1       69            165          104    194.0   1.50

      troponin  class
1178     0.028      1
64       0.089      1
827      1.400      1
25       1.390      1
198      0.007      0
```

[4]:
```python
# 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
```

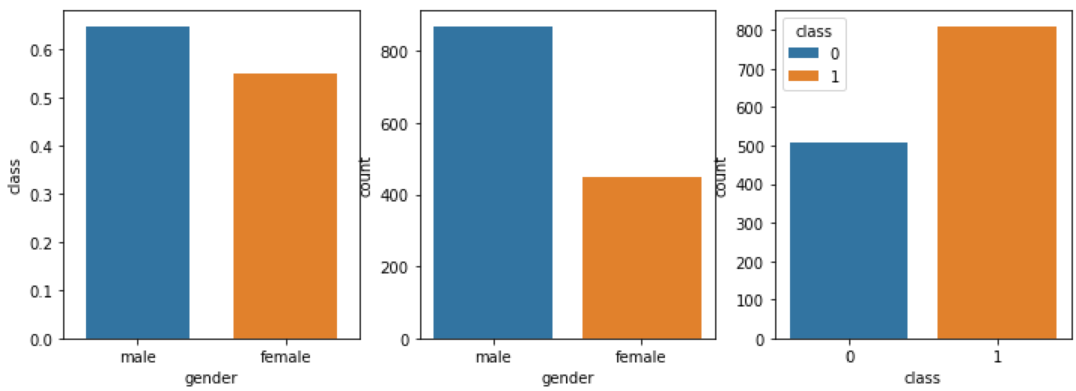
```
1274   70    male      103            126           75     541.0    0.665

      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
```

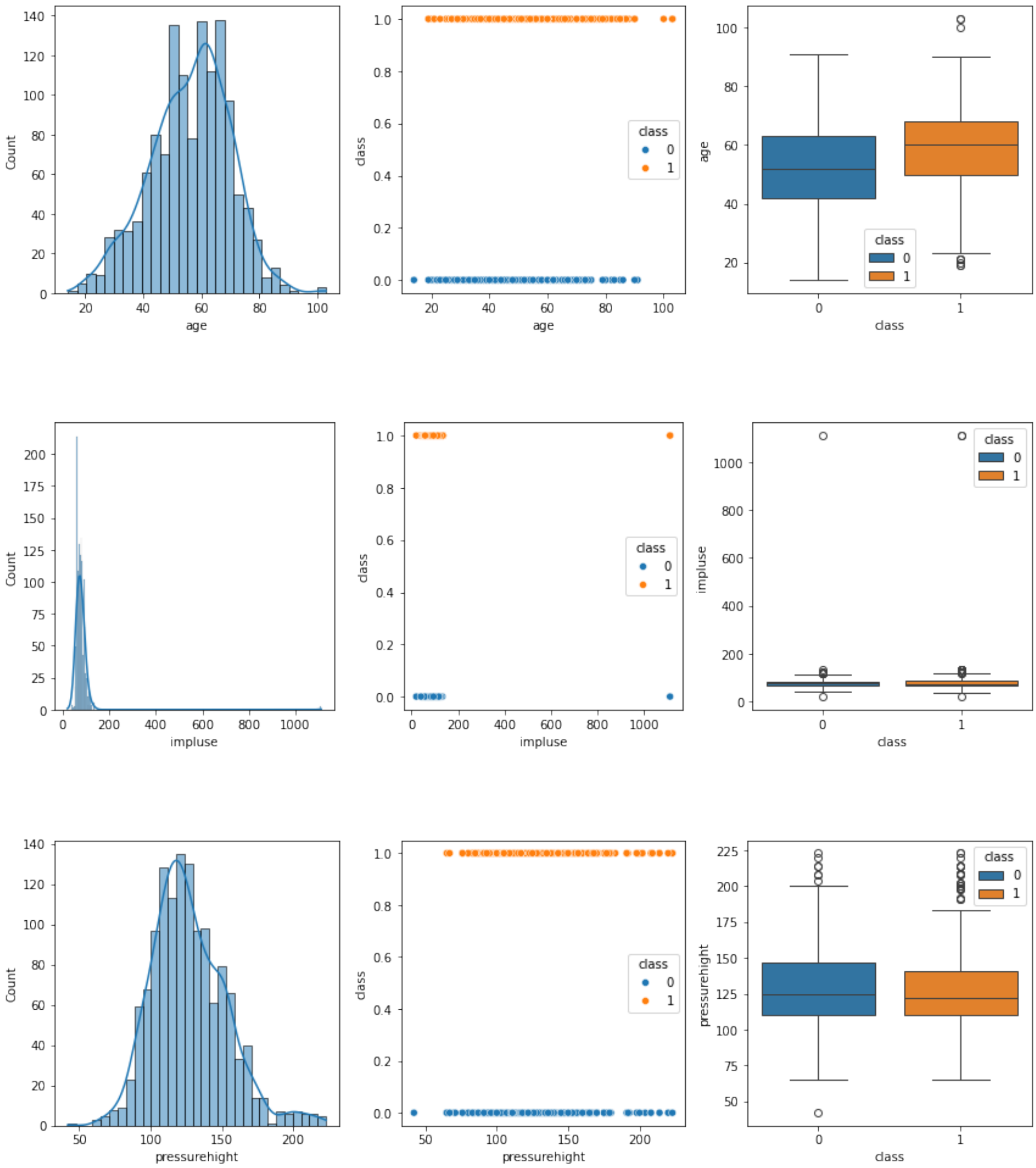
[6]:
```
# preprocessing2: Data visualization and analysis
# Seperate columns into categorical(discrete) and numerical(continuous)
# All features: 'gender', 'age', 'impluse', 'pressurehight', 'pressurelow', 'glucose', 'kcm', 'troponin'
categoric_columns = ["gender"]
numeric_columns = ['age', 'impluse',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

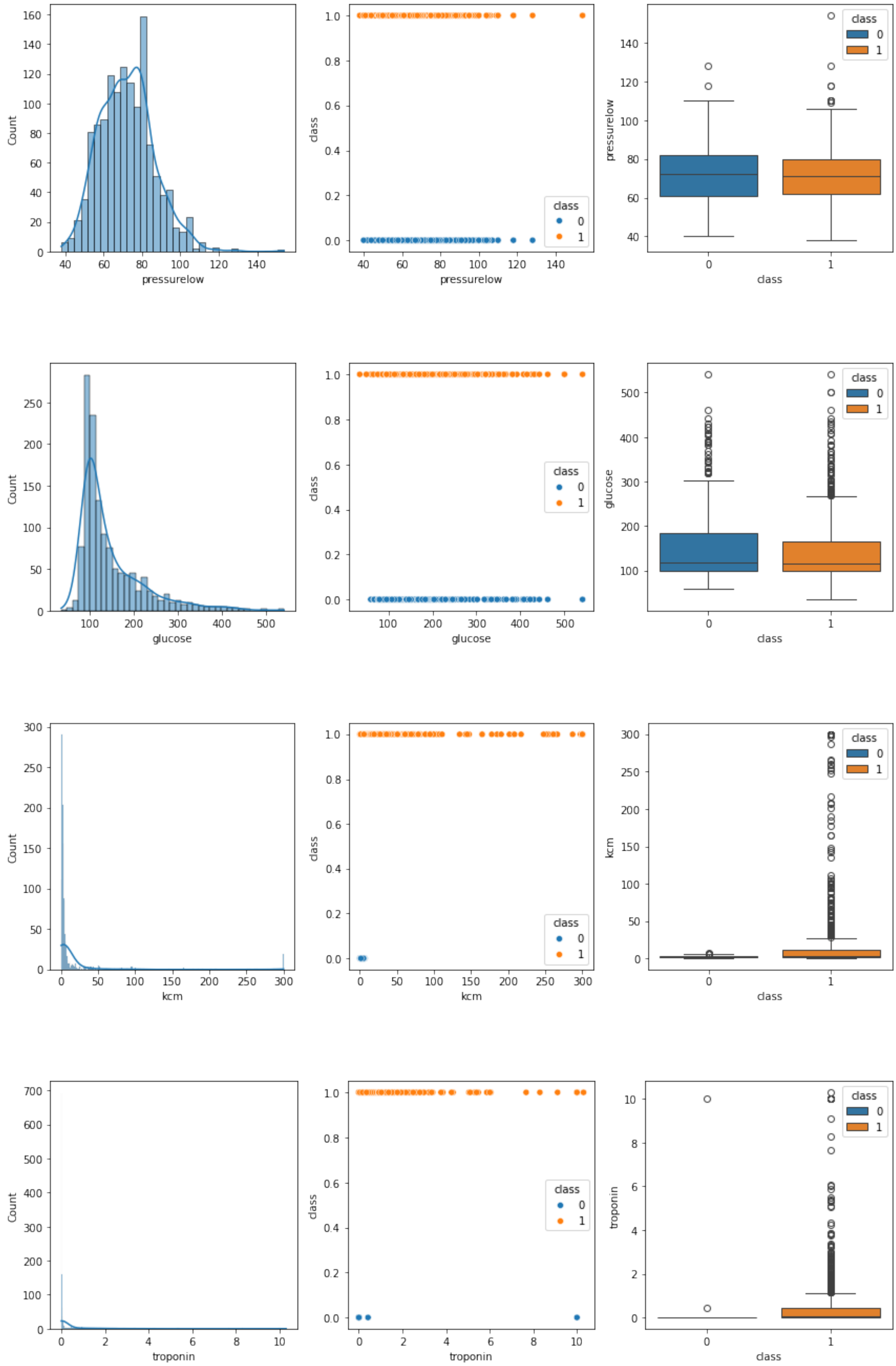
[7]:
```
# Gender(the only categorical feature)
fig, axes = plt.subplots(1, 3, figsize=(12, 4))
# Gender vs Class
sns.barplot(x="gender", y="class", data=visual_data, width=0.7, errorbar=None, hue="gender",␣
↪dodge=False, ax=axes[0])
# Gender count
sns.countplot(data=visual_data, x='gender', ax=axes[1], 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  \
63     45       1     1111            141           95    109.0  1.33
717    70       0     1111            141           95    138.0  3.87
1069   32       0     1111            141           95     82.0  2.66

      troponin  class
63       1.010      1
717      0.028      1
1069     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  \
29     63       1       66            135           55    166.0   0.493
475    58       0       80            107           67    166.0   6.480
753    49       1       75            116           71     98.0  37.690
988    57       1       95            129           77    251.0   4.340
1003   68       1       60            199           99    115.0   2.670
1028   68       1       89            145           68    134.0   0.706
1048   68       1       97            105           80     91.0   1.160
1094   65       1       74            140           85    106.0   4.350
1252   70       0       63            105           64    217.0   1.800
1310   70       0       80            135           75    351.0   2.210

      troponin  class
29       10.00      0
475       9.11      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
```

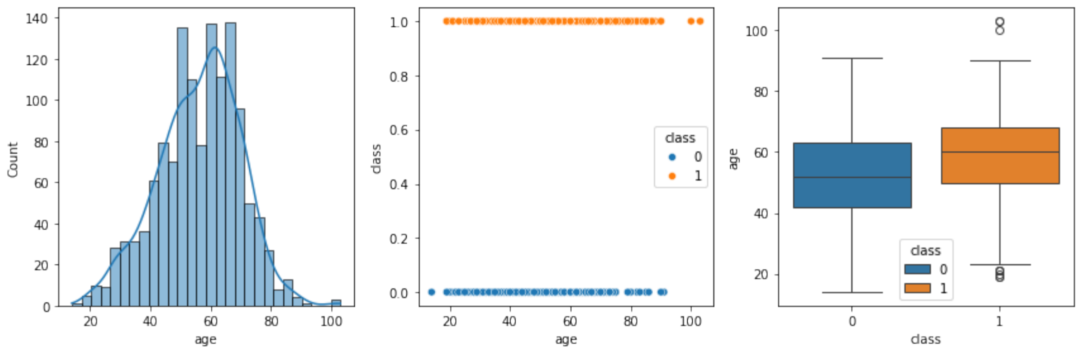
I think row29 is noise and should be deleted

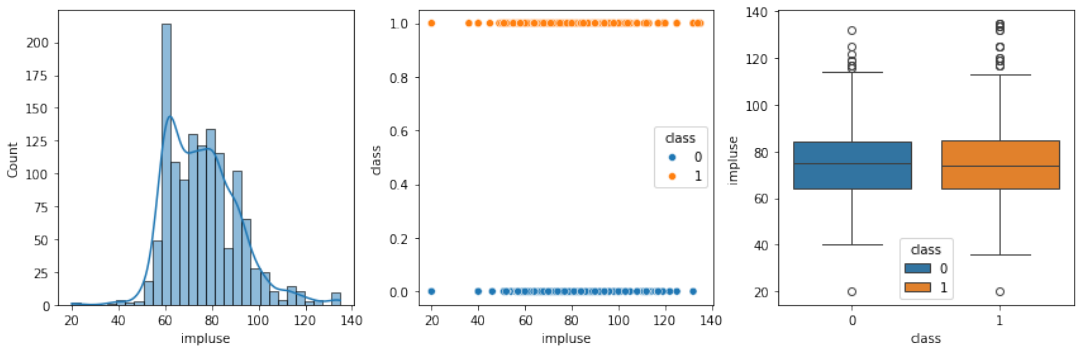
[12]:
```
#Delete abnormal row
model_data = model_data.drop(29)
visual_data = visual_data.drop(29)
model_data[model_data["troponin"] > 9]
```

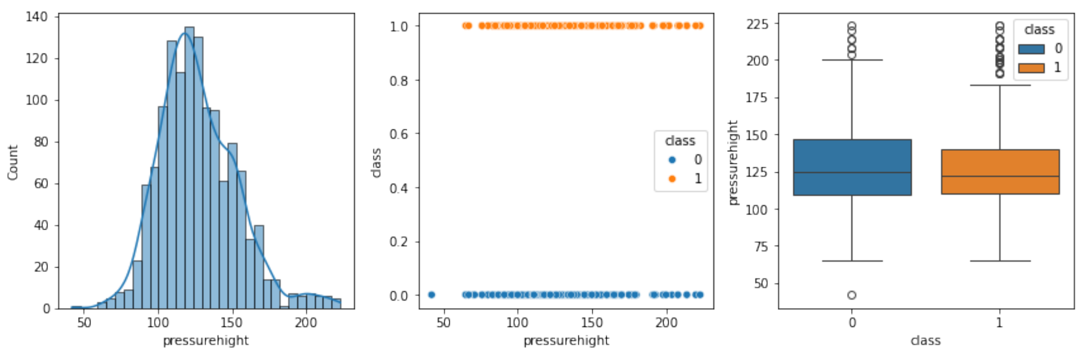
[12]:
```
      age  gender  impluse  pressurehight  pressurelow  glucose     kcm  \
475    58       0       80            107           67    166.0   6.480
753    49       1       75            116           71     98.0  37.690
988    57       1       95            129           77    251.0   4.340
1003   68       1       60            199           99    115.0   2.670
1028   68       1       89            145           68    134.0   0.706
1048   68       1       97            105           80     91.0   1.160
1094   65       1       74            140           85    106.0   4.350
1252   70       0       63            105           64    217.0   1.800
1310   70       0       80            135           75    351.0   2.210

      troponin  class
475       9.11      1
```

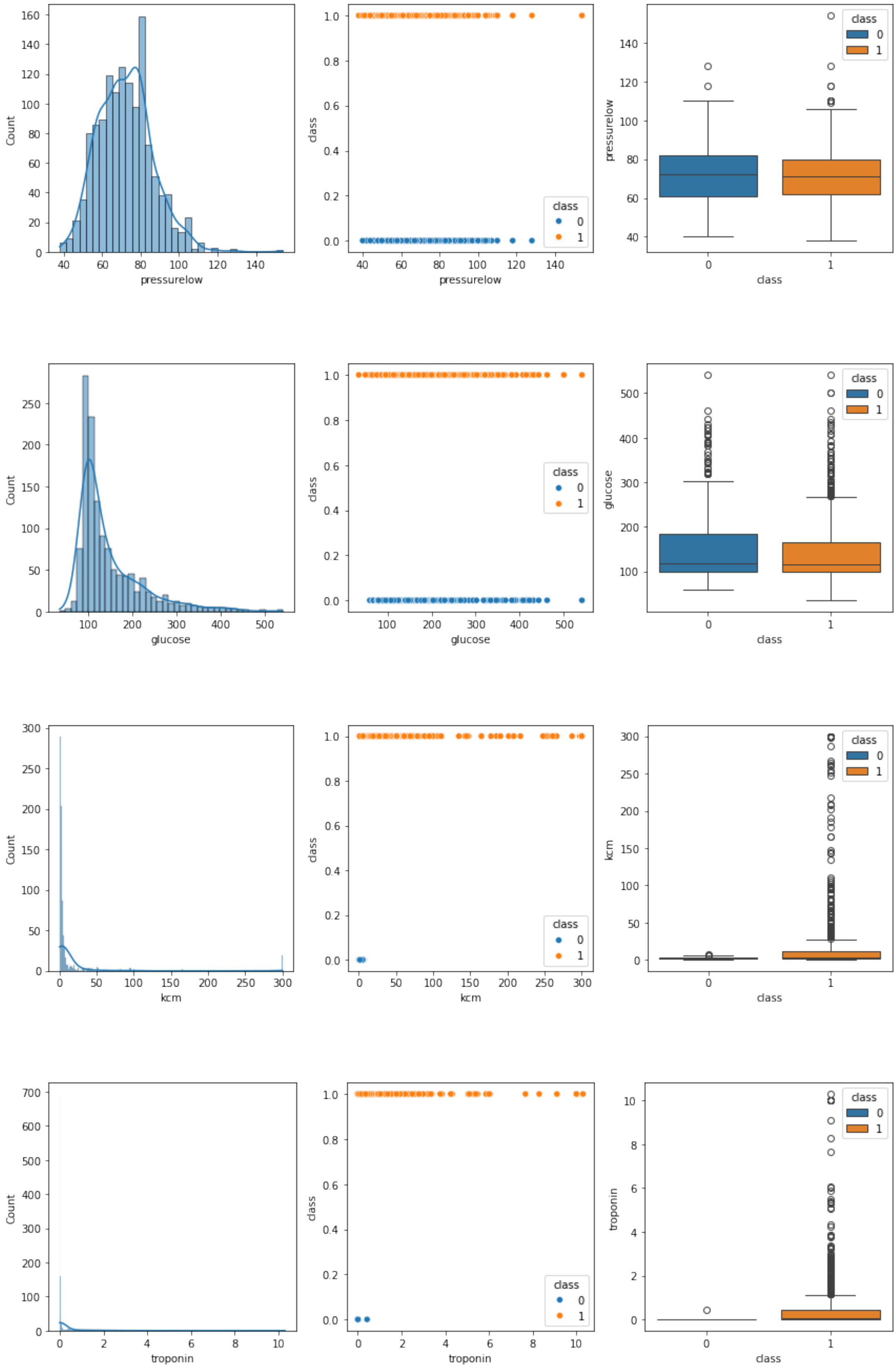
```
753     10.00     1
988     10.30     1
1003    10.00     1
1028    10.00     1
1048    10.00     1
1094    10.00     1
1252    10.00     1
1310    10.00     1
```

[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()
```







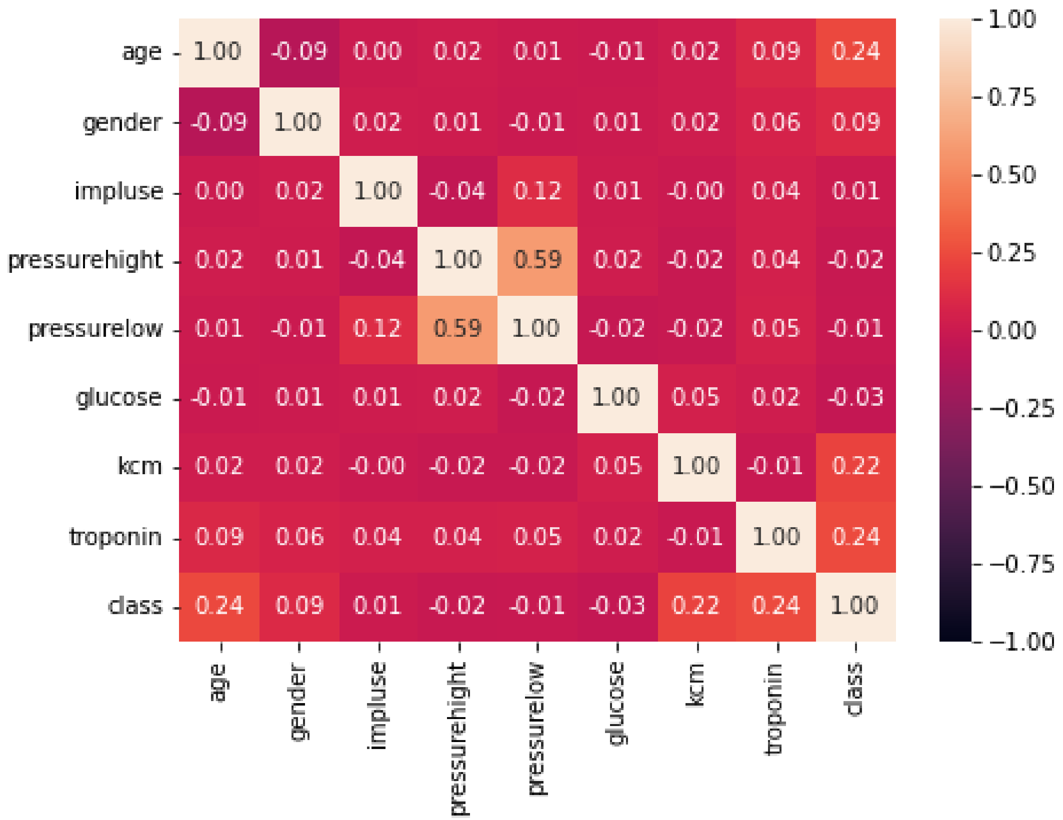


[14]:
```
model_data.describe().style.background_gradient()
```

[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,)
```

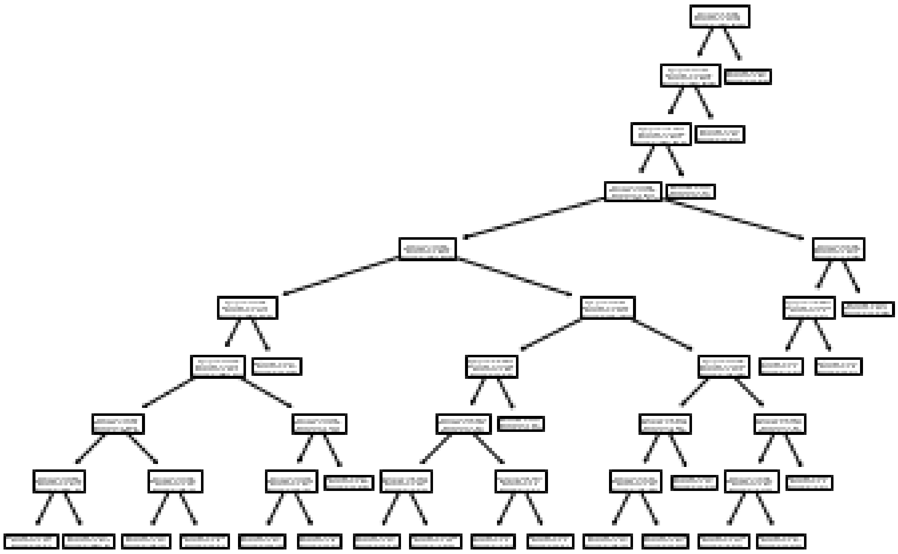
```
[17]: # Find best parameters for DTs

      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]:
```python
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)
```

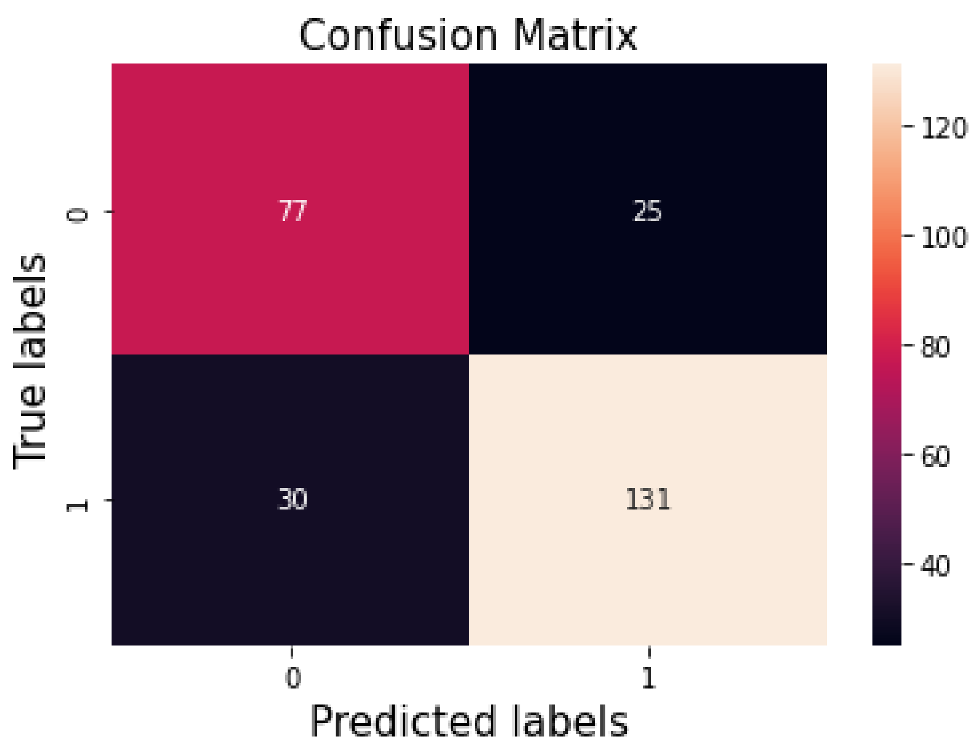
```
Best k : 11
score :  0.6768060836501901
```

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 10, 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.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)
```

[2]:
```
         id  gender   age  hypertension  heart_disease ever_married  \
472    2953  Female  43.0             0              0          Yes
4281  53105  Female  29.0             0              0          Yes
2932  48455  Female  37.0             0              0          Yes
493   66570  Female  23.0             0              0           No
1381  62272  Female  78.0             0              0          Yes
3666  20098  Female  31.0             0              0          Yes
3937  27675  Female   7.0             0              0           No
682   61300    Male  20.0             0              0           No
2357  39308    Male  62.0             0              0          Yes
3204  38348  Female  66.0             0              0          Yes

          work_type Residence_type  avg_glucose_level   bmi smoking_status  \
472         Private          Rural              75.05  22.9         smokes
4281        Private          Urban              63.90  45.4         smokes
2932        Private          Urban              60.05  24.1        Unknown
493         Private          Rural              69.24  51.0   never smoked
1381        Private          Urban             119.03  31.0   never smoked
3666  Self-employed          Rural             108.64  43.3   never smoked
3937       children          Urban             103.11  18.3        Unknown
682         Private          Urban              55.25  20.4   never smoked
2357        Private          Urban             145.37  33.3        Unknown
3204        Private          Urban              80.10  32.0   never smoked

      stroke
472        0
4281       0
2932       0
493        0
1381       0
3666       0
3937       0
682        0
2357       0
3204       0
```

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 excluded 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.

```
#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"]

# Create 2 copies 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)
```

```
rawdata.shape
```

```
(3425, 12)
```

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

```
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):
    print(label)
    print(m)
    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)
```

```
gender
{'Female': 0, 'Male': 1}
ever_married
{'No': 0, 'Yes': 1}
work_type
{'Never_worked': 0, 'Private': 1, 'Govt_job': 2, 'children': 3, 'Self-employed':
4}
Residence_type
{'Rural': 0, 'Urban': 1}
smoking_status
{'never smoked': 0, 'formerly smoked': 1, 'smokes': 2}
```

```
         id  gender   age  hypertension  heart_disease  ever_married  \
4786  30335       1  21.0             0              0             0
930   37290       1  80.0             0              0             1
4787  26305       1  29.0             0              0             0
4321  71143       1  65.0             0              0             1
243   40460       0  68.0             1              1             1

      work_type  Residence_type  avg_glucose_level   bmi  smoking_status  \
4786          1               0              92.86  23.2               0
930           4               0             236.84  26.8               0
4787          4               0              96.77  30.3               1
4321          4               1             179.67  30.7               1
243           1               1             247.51  40.5               1

      stroke
4786       0
930        0
4787       0
4321       0
```

```
243        1
```

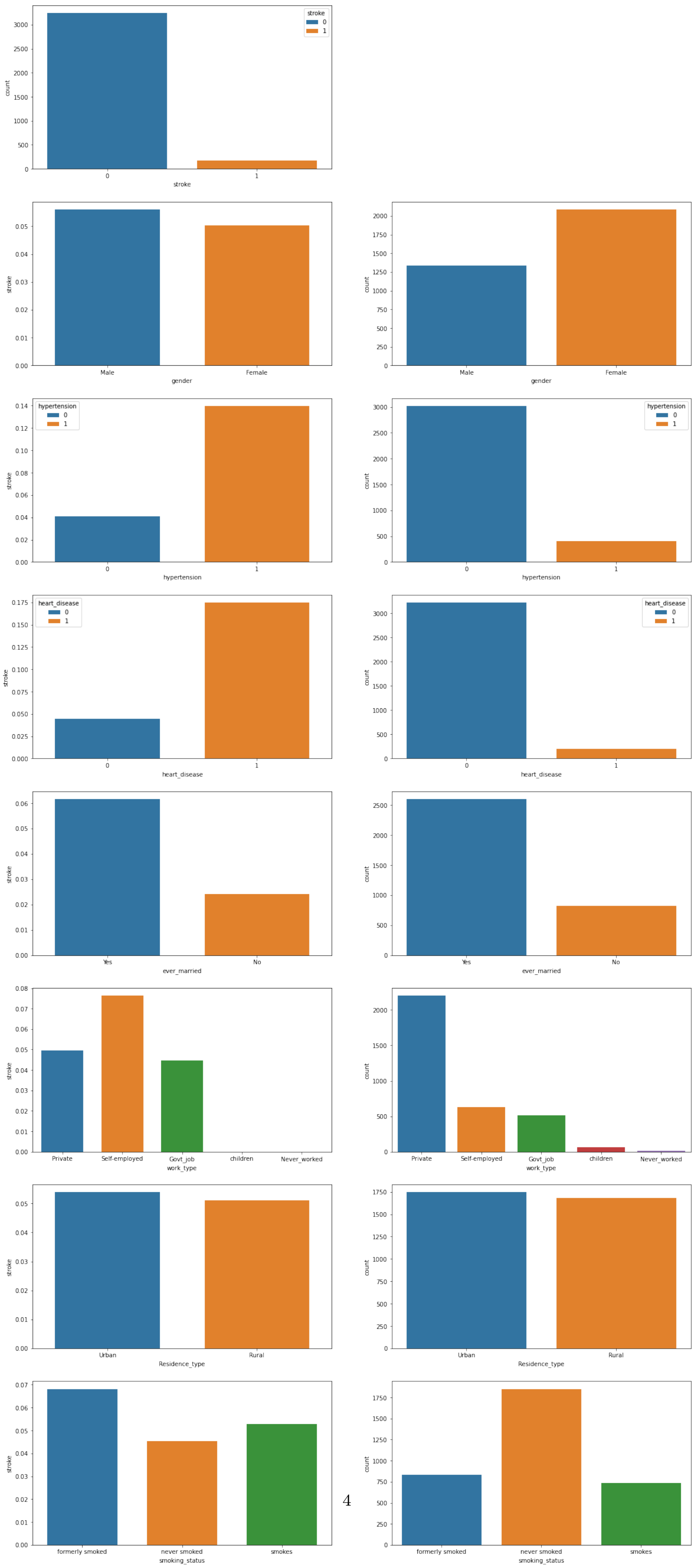
[6]:
```
model_data.describe()
```

[6]:
```
                 id       gender          age  hypertension  heart_disease  \
count   3425.000000  3425.000000  3425.000000   3425.000000    3425.000000
mean   37333.512117     0.390949    48.652555      0.119124       0.060146
std    21050.593185     0.488034    18.850018      0.323982       0.237792
min       84.000000     0.000000    10.000000      0.000000       0.000000
25%    18986.000000     0.000000    34.000000      0.000000       0.000000
50%    38067.000000     0.000000    50.000000      0.000000       0.000000
75%    55459.000000     1.000000    63.000000      0.000000       0.000000
max    72915.000000     1.000000    82.000000      1.000000       1.000000

       ever_married    work_type  Residence_type  avg_glucose_level  \
count   3425.000000  3425.000000     3425.000000        3425.000000
mean       0.758832     1.736642        0.509489         108.311670
std        0.427854     1.159385        0.499983          47.706754
min        0.000000     0.000000        0.000000          55.120000
25%        1.000000     1.000000        0.000000          77.230000
50%        1.000000     1.000000        1.000000          92.350000
75%        1.000000     2.000000        1.000000         116.200000
max        1.000000     4.000000        1.000000         271.740000

               bmi  smoking_status       stroke
count  3425.000000     3425.000000  3425.000000
mean     30.292350        0.674453     0.052555
std       7.295778        0.806301     0.223175
min      11.500000        0.000000     0.000000
25%      25.300000        0.000000     0.000000
50%      29.100000        0.000000     0.000000
75%      34.100000        1.000000     0.000000
max      92.000000        2.000000     1.000000
```

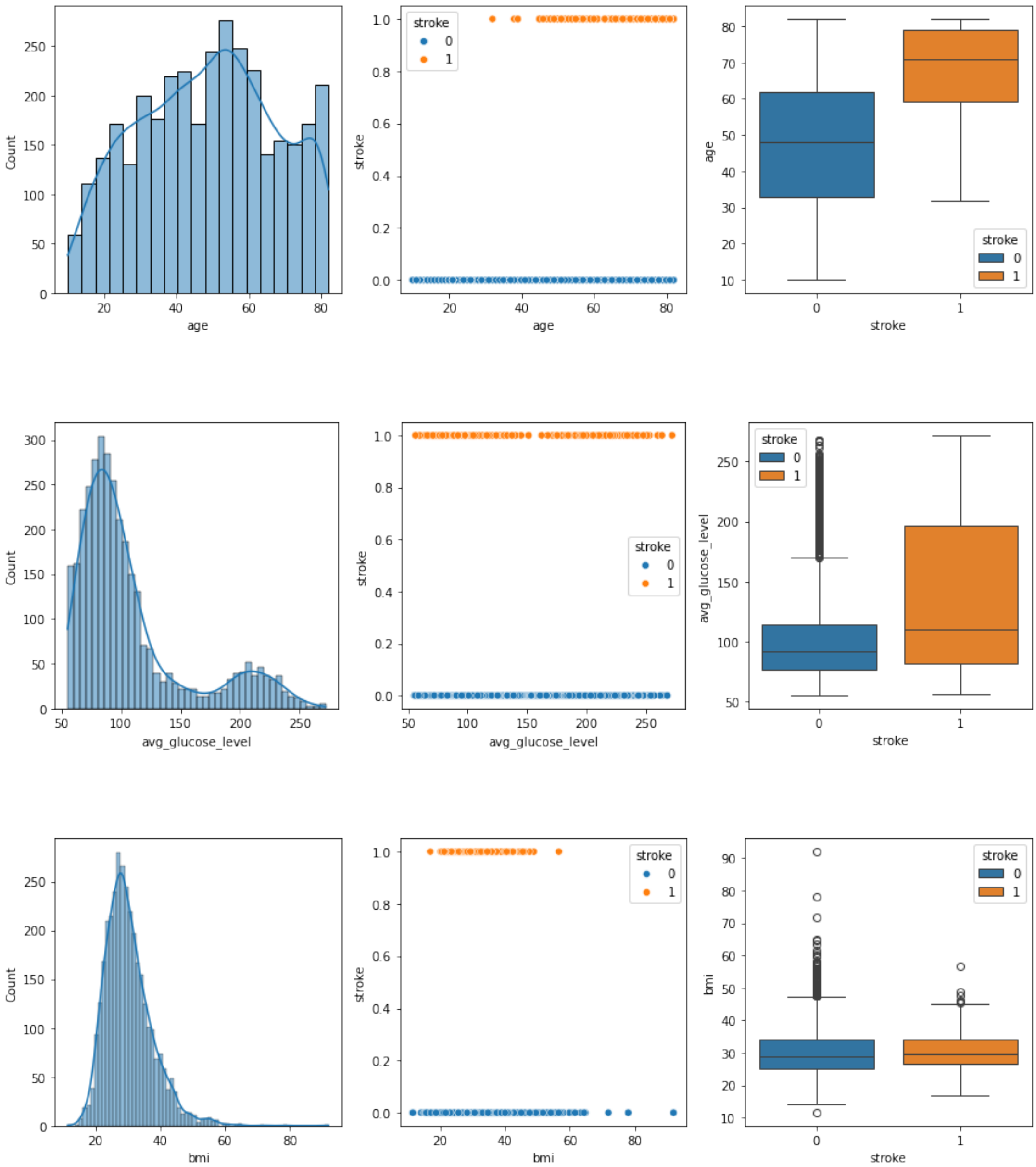
[7]:
```
# preprocessing2: Data visualization and analysis
# Seperate columns into categorical(discrete) and numerical(continuous)
# All features: 'gender', 'age', 'impluse', 'pressurehight', 'pressurelow', 'glucose', 'kcm', 'troponin'

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

[8]:
```
# 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.savefig(f"%c.jpg",dpi=500)


plt.show()
```


stroke
0
1
count
stroke
gender
Male
Female
hypertension
heart_disease
ever_married
Yes
No
work_type
Private
Self-employed
Govt_job
children
Never_worked
Residence_type
Urban
Rural
smoking_status
formerly smoked
never smoked
smokes

```
[9]: # 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

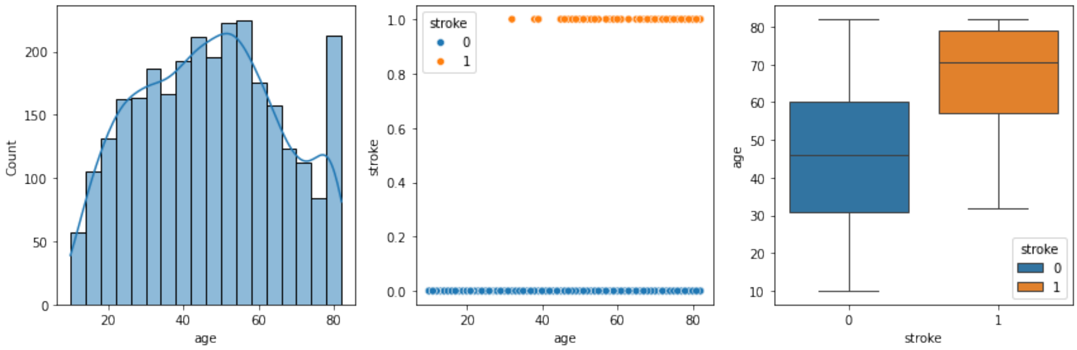
Outlier detection In a relatively small dataset of more than samples, outliers can have a more significant impact on statistical analyses or machine learning models compared to larger datasets.

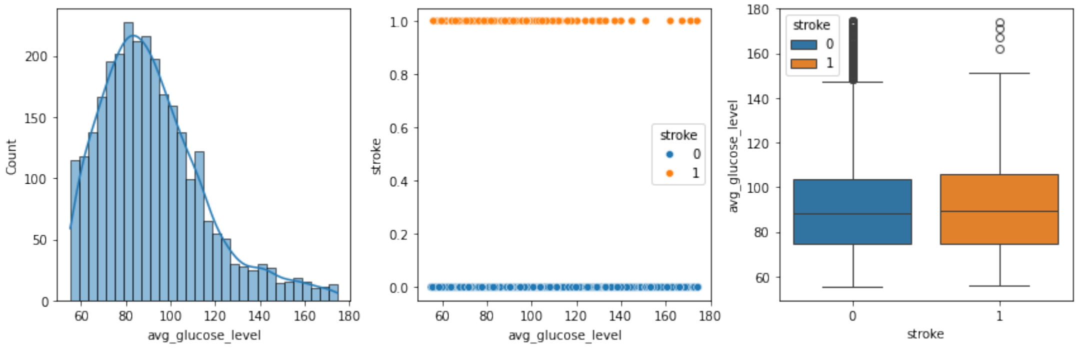
When we examine the boxplots, we can see that there are some outlier values, although not too many. We will clean these in the next step using IQR(Interquartile Range) method

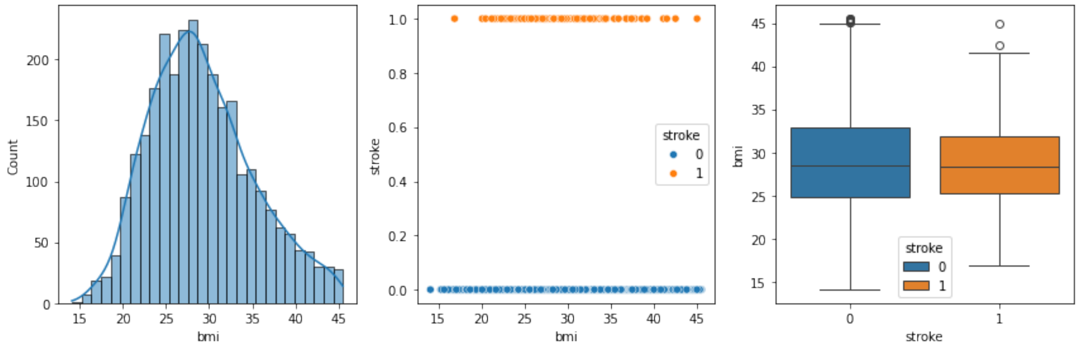
[10]:
```
for i in numeric_columns:
    Q1 = model_data[i].quantile(0.25)
    Q3 = model_data[i].quantile(0.75)
    IQR = Q3 - Q1
    lower_bound = Q1 - 1.5 * IQR
    upper_bound = Q3 + 1.5 * IQR
    print(f'Original shape: {model_data.shape}')
    model_data = model_data[(model_data[i] >= lower_bound) & (model_data[i] <= upper_bound)]
    print(f'Current shape: {model_data.shape}')
    print()
visual_data = model_data.copy(deep=True)
```

```
Original shape: (3425, 12)
Current shape: (3425, 12)

Original shape: (3425, 12)
Current shape: (2960, 12)

Original shape: (2960, 12)
Current shape: (2882, 12)
```

[11]:
```
# 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`

[12]:
```
# 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=42)
#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=42)

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_test set : {}'.format(X_val.shape))
print('Shape of y_test set : {}'.format(y_val.shape))
```

```
Shape of X_Train set : (2305, 9)
Shape of y_Train set : (2305,)
__________________________________________________
Shape of X_test set : (289, 9)
Shape of y_test set : (289,)
__________________________________________________
Shape of X_test set : (288, 9)
Shape of y_test set : (288,)
```

[13]:
```
def generate_confusion_matrix(y_true, y_pred):
    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)
```

[14]:
```
# 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(f"Prediction accuracy: {100*best_acc:.2f}%")
tree.plot_tree(DTs)
```

```
Best criterion :  gini
Best splitter :  best
Best depth :  5
Prediction accuracy: 96.18%
```

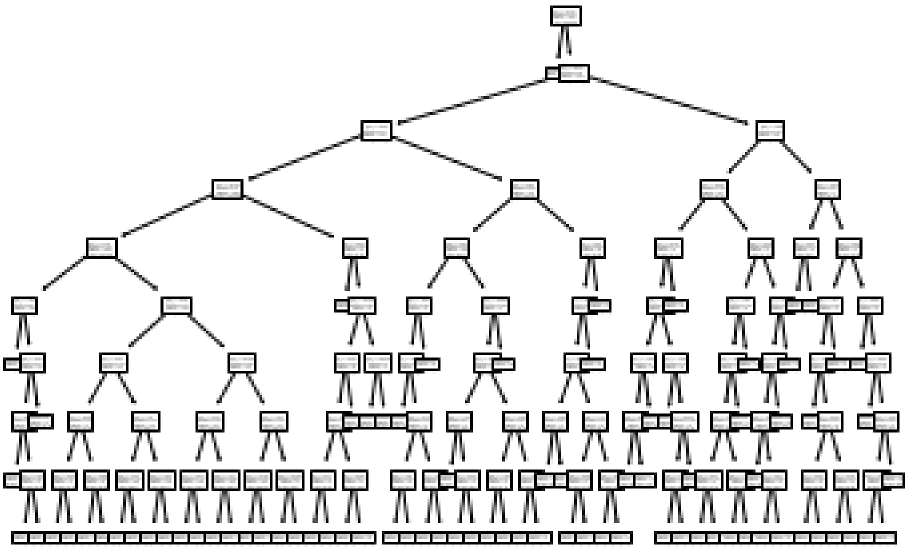
[14]:
```
[Text(208.77871621621622, 206.56799999999998, 'X[1] <= 24.758\nentropy =
0.248\nsamples = 2305\nvalue = [2210, 95]'),
 Text(205.76250000000002, 184.824, 'entropy = 0.0\nsamples = 346\nvalue = [346,
0]'),
 Text(211.79493243243246, 184.824, 'X[1] <= 76.198\nentropy = 0.28\nsamples =
1959\nvalue = [1864, 95]'),
 Text(137.8033783783784, 163.07999999999998, 'X[2] <= 0.644\nentropy =
0.213\nsamples = 1775\nvalue = [1715, 60]'),
 Text(82.1918918918919, 141.336, 'X[1] <= 72.794\nentropy = 0.19\nsamples =
1609\nvalue = [1562, 47]'),
 Text(34.68648648648649, 119.592, 'X[1] <= 37.191\nentropy = 0.179\nsamples =
1553\nvalue = [1511, 42]'),
 Text(6.032432432432433, 97.848, 'X[8] <= 1.483\nentropy = 0.024\nsamples =
430\nvalue = [429, 1]'),
 Text(3.0162162162162165, 76.10399999999998, 'entropy = 0.0\nsamples =
319\nvalue = [319, 0]'),
 Text(9.04864864864865, 76.10399999999998, 'X[0] <= 0.697\nentropy =
0.074\nsamples = 111\nvalue = [110, 1]'),
 Text(6.032432432432433, 54.360000000000014, 'X[1] <= 30.127\nentropy =
0.116\nsamples = 64\nvalue = [63, 1]'),
 Text(3.0162162162162165, 32.615999999999985, 'entropy = 0.0\nsamples =
28\nvalue = [28, 0]'),
 Text(9.04864864864865, 32.615999999999985, 'X[5] <= 1.604\nentropy =
0.183\nsamples = 36\nvalue = [35, 1]'),
 Text(6.032432432432433, 10.872000000000014, 'entropy = 0.222\nsamples =
28\nvalue = [27, 1]'),
 Text(12.064864864864866, 10.872000000000014, 'entropy = 0.0\nsamples = 8\nvalue
= [8, 0]'),
 Text(12.064864864864866, 54.360000000000014, 'entropy = 0.0\nsamples =
47\nvalue = [47, 0]'),
 Text(63.340540540540545, 97.848, 'X[8] <= 0.928\nentropy = 0.226\nsamples =
1123\nvalue = [1082, 41]'),
 Text(39.21081081081081, 76.10399999999998, 'X[4] <= 0.001\nentropy =
0.149\nsamples = 561\nvalue = [549, 12]'),
 Text(27.145945945945947, 54.360000000000014, 'X[1] <= 55.207\nentropy =
0.297\nsamples = 57\nvalue = [54, 3]'),
 Text(21.113513513513517, 32.615999999999985, 'X[0] <= 0.924\nentropy =
0.162\nsamples = 42\nvalue = [41, 1]'),
 Text(18.0972972972973, 10.872000000000014, 'entropy = 0.0\nsamples = 30\nvalue
= [30, 0]'),
 Text(24.129729729729732, 10.872000000000014, 'entropy = 0.414\nsamples =
12\nvalue = [11, 1]'),
 Text(33.17837837837838, 32.615999999999985, 'X[0] <= 0.508\nentropy =
0.567\nsamples = 15\nvalue = [13, 2]'),
 Text(30.162162162162165, 10.872000000000014, 'entropy = 0.918\nsamples =
```

```
6\nvalue = [4, 2]'),
 Text(36.1945945945946, 10.872000000000014, 'entropy = 0.0\nsamples = 9\nvalue =
[9, 0]'),
 Text(51.27567567567568, 54.360000000000014, 'X[0] <= 0.79\nentropy =
0.129\nsamples = 504\nvalue = [495, 9]'),
 Text(45.24324324324325, 32.615999999999985, 'X[5] <= 2.334\nentropy =
0.166\nsamples = 326\nvalue = [318, 8]'),
 Text(42.227027027027034, 10.872000000000014, 'entropy = 0.201\nsamples =
255\nvalue = [247, 8]'),
 Text(48.259459459459464, 10.872000000000014, 'entropy = 0.0\nsamples =
71\nvalue = [71, 0]'),
 Text(57.308108108108115, 32.615999999999985, 'X[5] <= 1.784\nentropy =
0.05\nsamples = 178\nvalue = [177, 1]'),
 Text(54.29189189189189, 10.872000000000014, 'entropy = 0.071\nsamples =
117\nvalue = [116, 1]'),
 Text(60.32432432432433, 10.872000000000014, 'entropy = 0.0\nsamples = 61\nvalue
= [61, 0]'),
 Text(87.47027027027028, 76.10399999999998, 'X[6] <= 101.384\nentropy =
0.293\nsamples = 562\nvalue = [533, 29]'),
 Text(75.40540540540542, 54.360000000000014, 'X[6] <= 84.948\nentropy =
0.218\nsamples = 401\nvalue = [387, 14]'),
 Text(69.37297297297297, 32.615999999999985, 'X[1] <= 64.018\nentropy =
0.142\nsamples = 248\nvalue = [243, 5]'),
 Text(66.35675675675677, 10.872000000000014, 'entropy = 0.079\nsamples =
207\nvalue = [205, 2]'),
 Text(72.3891891891892, 10.872000000000014, 'entropy = 0.378\nsamples =
41\nvalue = [38, 3]'),
 Text(81.43783783783785, 32.615999999999985, 'X[8] <= 1.691\nentropy =
0.323\nsamples = 153\nvalue = [144, 9]'),
 Text(78.42162162162163, 10.872000000000014, 'entropy = 0.213\nsamples =
89\nvalue = [86, 3]'),
 Text(84.45405405405407, 10.872000000000014, 'entropy = 0.449\nsamples =
64\nvalue = [58, 6]'),
 Text(99.53513513513515, 54.360000000000014, 'X[3] <= 0.782\nentropy =
0.447\nsamples = 161\nvalue = [146, 15]'),
 Text(93.5027027027027, 32.615999999999985, 'X[1] <= 42.763\nentropy =
0.402\nsamples = 150\nvalue = [138, 12]'),
 Text(90.4864864864865, 10.872000000000014, 'entropy = 0.0\nsamples = 22\nvalue
= [22, 0]'),
 Text(96.51891891891893, 10.872000000000014, 'entropy = 0.449\nsamples =
128\nvalue = [116, 12]'),
 Text(105.56756756756758, 32.615999999999985, 'X[6] <= 141.122\nentropy =
0.845\nsamples = 11\nvalue = [8, 3]'),
 Text(102.55135135135136, 10.872000000000014, 'entropy = 0.918\nsamples =
9\nvalue = [6, 3]'),
 Text(108.58378378378379, 10.872000000000014, 'entropy = 0.0\nsamples = 2\nvalue
= [2, 0]'),
 Text(129.6972972972973, 119.592, 'X[6] <= 64.894\nentropy = 0.434\nsamples =
56\nvalue = [51, 5]'),
 Text(126.68108108108109, 97.848, 'entropy = 0.0\nsamples = 5\nvalue = [5, 0]'),
 Text(132.71351351351353, 97.848, 'X[6] <= 133.675\nentropy = 0.463\nsamples =
51\nvalue = [46, 5]'),
 Text(126.68108108108109, 76.10399999999998, 'X[6] <= 96.938\nentropy =
0.408\nsamples = 49\nvalue = [45, 4]'),
 Text(123.66486486486488, 54.360000000000014, 'X[7] <= 29.073\nentropy =
0.503\nsamples = 36\nvalue = [32, 4]'),
 Text(117.63243243243244, 32.615999999999985, 'X[5] <= 3.241\nentropy =
0.25\nsamples = 24\nvalue = [23, 1]'),
 Text(114.61621621621623, 10.872000000000014, 'entropy = 0.0\nsamples =
14\nvalue = [14, 0]'),
 Text(120.64864864864866, 10.872000000000014, 'entropy = 0.469\nsamples =
10\nvalue = [9, 1]'),
 Text(129.6972972972973, 32.615999999999985, 'X[6] <= 74.593\nentropy =
0.811\nsamples = 12\nvalue = [9, 3]'),
 Text(126.68108108108109, 10.872000000000014, 'entropy = 1.0\nsamples = 4\nvalue
= [2, 2]'),
 Text(132.71351351351353, 10.872000000000014, 'entropy = 0.544\nsamples =
8\nvalue = [7, 1]'),
 Text(129.6972972972973, 54.360000000000014, 'entropy = 0.0\nsamples = 13\nvalue
= [13, 0]'),
 Text(138.74594594594595, 76.10399999999998, 'X[6] <= 140.342\nentropy =
1.0\nsamples = 2\nvalue = [1, 1]'),
 Text(135.72972972972974, 54.360000000000014, 'entropy = 0.0\nsamples = 1\nvalue
```

```
= [0, 1]'),
 Text(141.7621621621621 8, 54.360000000000014, 'entropy = 0.0\nsamples = 1\nvalue
= [1, 0]'),
 Text(193.41486486486488, 141.336, 'X[5] <= 3.184\nentropy = 0.396\nsamples =
166\nvalue = [153, 13]'),
 Text(168.15405405405406, 119.592, 'X[8] <= 0.561\nentropy = 0.33\nsamples =
132\nvalue = [124, 8]'),
 Text(153.82702702702704, 97.848, 'X[7] <= 28.92\nentropy = 0.194\nsamples =
67\nvalue = [65, 2]'),
 Text(150.81081081081084, 76.10399999999998, 'X[7] <= 25.789\nentropy =
0.414\nsamples = 24\nvalue = [22, 2]'),
 Text(147.7945945945946, 54.360000000000014, 'entropy = 0.0\nsamples = 12\nvalue
= [12, 0]'),
 Text(153.82702702702704, 54.360000000000014, 'X[3] <= 0.612\nentropy =
0.65\nsamples = 12\nvalue = [10, 2]'),
 Text(147.7945945945946, 32.615999999999985, 'X[0] <= 0.12\nentropy =
0.469\nsamples = 10\nvalue = [9, 1]'),
 Text(144.7783783783784, 10.872000000000014, 'entropy = 0.722\nsamples =
5\nvalue = [4, 1]'),
 Text(150.81081081081084, 10.872000000000014, 'entropy = 0.0\nsamples = 5\nvalue
= [5, 0]'),
 Text(159.8594594594595, 32.615999999999985, 'X[7] <= 26.398\nentropy =
1.0\nsamples = 2\nvalue = [1, 1]'),
 Text(156.84324324324325, 10.872000000000014, 'entropy = 0.0\nsamples = 1\nvalue
= [1, 0]'),
 Text(162.8756756756757, 10.872000000000014, 'entropy = 0.0\nsamples = 1\nvalue
= [0, 1]'),
 Text(156.84324324324325, 76.10399999999998, 'entropy = 0.0\nsamples = 43\nvalue
= [43, 0]'),
 Text(182.4810810810811, 97.848, 'X[6] <= 130.726\nentropy = 0.444\nsamples =
65\nvalue = [59, 6]'),
 Text(179.4648648648649, 76.10399999999998, 'X[1] <= 51.692\nentropy =
0.485\nsamples = 57\nvalue = [51, 6]'),
 Text(168.90810810810814, 54.360000000000014, 'X[8] <= 1.645\nentropy =
0.811\nsamples = 12\nvalue = [9, 3]'),
 Text(165.8918918918919, 32.615999999999985, 'entropy = 0.0\nsamples = 2\nvalue
= [0, 2]'),
 Text(171.92432432432435, 32.615999999999985, 'X[6] <= 85.666\nentropy =
0.469\nsamples = 10\nvalue = [9, 1]'),
 Text(168.90810810810814, 10.872000000000014, 'entropy = 0.722\nsamples =
5\nvalue = [4, 1]'),
 Text(174.94054054054055, 10.872000000000014, 'entropy = 0.0\nsamples = 5\nvalue
= [5, 0]'),
 Text(190.02162162162165, 54.360000000000014, 'X[3] <= 0.169\nentropy =
0.353\nsamples = 45\nvalue = [42, 3]'),
 Text(183.9891891891892, 32.615999999999985, 'X[8] <= 1.585\nentropy =
0.292\nsamples = 39\nvalue = [37, 2]'),
 Text(180.972972972973, 10.872000000000014, 'entropy = 0.414\nsamples =
24\nvalue = [22, 2]'),
 Text(187.0054054054054, 10.872000000000014, 'entropy = 0.0\nsamples = 15\nvalue
= [15, 0]'),
 Text(196.05405405405406, 32.615999999999985, 'X[1] <= 64.771\nentropy =
0.65\nsamples = 6\nvalue = [5, 1]'),
 Text(193.03783783783786, 10.872000000000014, 'entropy = 1.0\nsamples = 2\nvalue
= [1, 1]'),
 Text(199.0702702702703, 10.872000000000014, 'entropy = 0.0\nsamples = 4\nvalue
= [4, 0]'),
 Text(185.4972972972973, 76.10399999999998, 'entropy = 0.0\nsamples = 8\nvalue =
[8, 0]'),
 Text(218.6756756756757, 119.592, 'X[3] <= 0.709\nentropy = 0.602\nsamples =
34\nvalue = [29, 5]'),
 Text(215.65945945945947, 97.848, 'X[7] <= 34.802\nentropy = 0.533\nsamples =
33\nvalue = [29, 4]'),
 Text(212.64324324324326, 76.10399999999998, 'X[4] <= 0.882\nentropy =
0.667\nsamples = 23\nvalue = [19, 4]'),
 Text(205.10270270270271, 54.360000000000014, 'X[7] <= 23.896\nentropy =
0.918\nsamples = 3\nvalue = [2, 1]'),
 Text(202.0864864864865, 32.615999999999985, 'entropy = 0.0\nsamples = 2\nvalue
= [2, 0]'),
 Text(208.11891891891895, 32.615999999999985, 'entropy = 0.0\nsamples = 1\nvalue
= [0, 1]'),
 Text(220.1837837837838, 54.360000000000014, 'X[8] <= 1.047\nentropy =
0.61\nsamples = 20\nvalue = [17, 3]'),
```

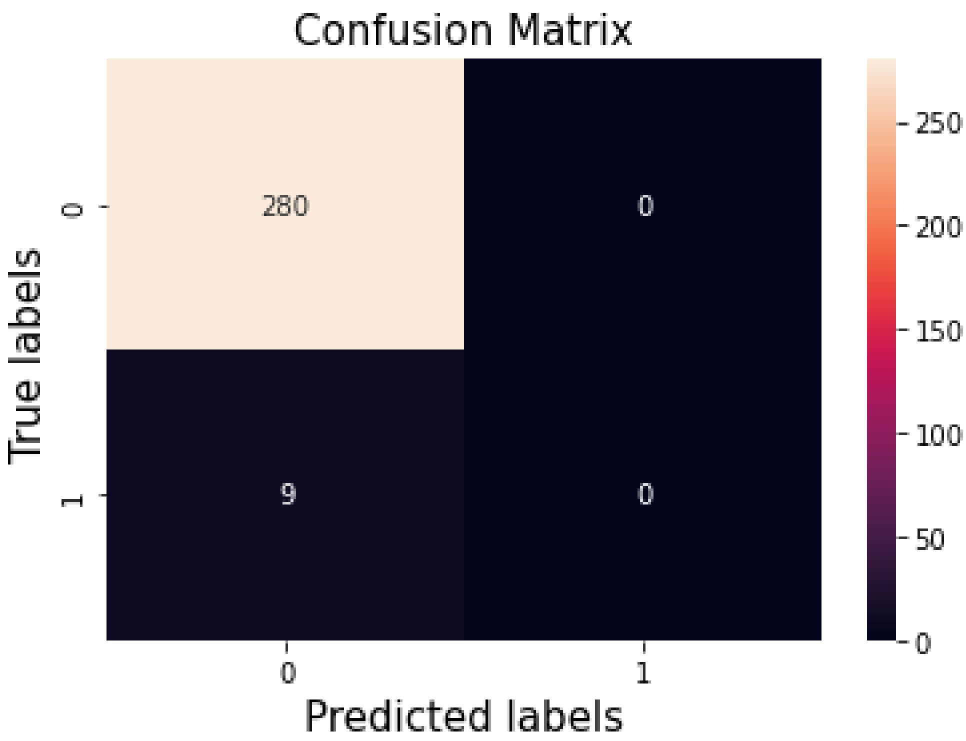
```
 Text(214.15135135135137, 32.615999999999985, 'X[6] <= 99.979\nentropy =
0.323\nsamples = 17\nvalue = [16, 1]'),
 Text(211.13513513513516, 10.872000000000014, 'entropy = 0.0\nsamples =
10\nvalue = [10, 0]'),
 Text(217.16756756756757, 10.872000000000014, 'entropy = 0.592\nsamples =
7\nvalue = [6, 1]'),
 Text(226.21621621621622, 32.615999999999985, 'X[6] <= 79.261\nentropy =
0.918\nsamples = 3\nvalue = [1, 2]'),
 Text(223.20000000000002, 10.872000000000014, 'entropy = 0.0\nsamples = 2\nvalue
= [0, 2]'),
 Text(229.23243243243246, 10.872000000000014, 'entropy = 0.0\nsamples = 1\nvalue
= [1, 0]'),
 Text(218.6756756756757, 76.10399999999998, 'entropy = 0.0\nsamples = 10\nvalue
= [10, 0]'),
 Text(221.6918918918919, 97.848, 'entropy = 0.0\nsamples = 1\nvalue = [0, 1]'),
 Text(285.7864864864865, 163.07999999999998, 'X[8] <= 0.775\nentropy =
0.702\nsamples = 184\nvalue = [149, 35]'),
 Text(264.672972972973, 141.336, 'X[1] <= 79.465\nentropy = 0.784\nsamples =
107\nvalue = [82, 25]'),
 Text(247.32972972972976, 119.592, 'X[7] <= 32.785\nentropy = 0.714\nsamples =
51\nvalue = [41, 10]'),
 Text(244.31351351351353, 97.848, 'X[4] <= 0.665\nentropy = 0.801\nsamples =
41\nvalue = [31, 10]'),
 Text(238.2810810810811, 76.10399999999998, 'X[7] <= 27.098\nentropy =
1.0\nsamples = 6\nvalue = [3, 3]'),
 Text(235.26486486486488, 54.360000000000014, 'X[3] <= 0.656\nentropy =
0.811\nsamples = 4\nvalue = [1, 3]'),
 Text(232.24864864864867, 32.615999999999985, 'entropy = 0.0\nsamples = 3\nvalue
= [0, 3]'),
 Text(238.2810810810811, 32.615999999999985, 'entropy = 0.0\nsamples = 1\nvalue
= [1, 0]'),
 Text(241.29729729729732, 54.360000000000014, 'entropy = 0.0\nsamples = 2\nvalue
= [2, 0]'),
 Text(250.34594594594597, 76.10399999999998, 'X[7] <= 21.694\nentropy =
0.722\nsamples = 35\nvalue = [28, 7]'),
 Text(247.32972972972976, 54.360000000000014, 'entropy = 0.0\nsamples = 3\nvalue
= [3, 0]'),
 Text(253.36216216216218, 54.360000000000014, 'X[6] <= 155.552\nentropy =
0.758\nsamples = 32\nvalue = [25, 7]'),
 Text(250.34594594594597, 32.615999999999985, 'X[0] <= 0.778\nentropy =
0.709\nsamples = 31\nvalue = [25, 6]'),
 Text(247.32972972972976, 10.872000000000014, 'entropy = 0.811\nsamples =
20\nvalue = [15, 5]'),
 Text(253.36216216216218, 10.872000000000014, 'entropy = 0.439\nsamples =
11\nvalue = [10, 1]'),
 Text(256.3783783783784, 32.615999999999985, 'entropy = 0.0\nsamples = 1\nvalue
= [0, 1]'),
 Text(250.34594594594597, 97.848, 'entropy = 0.0\nsamples = 10\nvalue = [10,
0]'),
 Text(282.01621621621626, 119.592, 'X[0] <= 0.249\nentropy = 0.838\nsamples =
56\nvalue = [41, 15]'),
 Text(274.4756756756757, 97.848, 'X[3] <= 0.789\nentropy = 0.769\nsamples =
40\nvalue = [31, 9]'),
 Text(271.4594594594595, 76.10399999999998, 'X[6] <= 98.766\nentropy =
0.822\nsamples = 35\nvalue = [26, 9]'),
 Text(268.4432432432433, 54.360000000000014, 'X[2] <= 0.112\nentropy =
0.906\nsamples = 28\nvalue = [19, 9]'),
 Text(262.41081081081086, 32.615999999999985, 'X[4] <= 0.261\nentropy =
0.742\nsamples = 19\nvalue = [15, 4]'),
 Text(259.3945945945946, 10.872000000000014, 'entropy = 0.0\nsamples = 6\nvalue
= [6, 0]'),
 Text(265.42702702702707, 10.872000000000014, 'entropy = 0.89\nsamples =
13\nvalue = [9, 4]'),
 Text(274.4756756756757, 32.615999999999985, 'X[5] <= 1.028\nentropy =
0.991\nsamples = 9\nvalue = [4, 5]'),
 Text(271.4594594594595, 10.872000000000014, 'entropy = 0.985\nsamples =
7\nvalue = [4, 3]'),
 Text(277.4918918918919, 10.872000000000014, 'entropy = 0.0\nsamples = 2\nvalue
= [0, 2]'),
 Text(274.4756756756757, 54.360000000000014, 'entropy = 0.0\nsamples = 7\nvalue
= [7, 0]'),
 Text(277.4918918918919, 76.10399999999998, 'entropy = 0.0\nsamples = 5\nvalue =
[5, 0]'),
```

```
 Text(289.5567567567568, 97.848, 'X[7] <= 32.191\nentropy = 0.954\nsamples =
16\nvalue = [10, 6]'),
 Text(286.5405405405406, 76.10399999999998, 'X[2] <= 0.141\nentropy =
0.863\nsamples = 14\nvalue = [10, 4]'),
 Text(283.52432432432437, 54.360000000000014, 'X[6] <= 68.085\nentropy =
0.946\nsamples = 11\nvalue = [7, 4]'),
 Text(280.50810810810816, 32.615999999999985, 'entropy = 0.0\nsamples = 2\nvalue
= [2, 0]'),
 Text(286.5405405405406, 32.615999999999985, 'X[7] <= 25.571\nentropy =
0.991\nsamples = 9\nvalue = [5, 4]'),
 Text(283.52432432432437, 10.872000000000014, 'entropy = 0.918\nsamples =
3\nvalue = [1, 2]'),
 Text(289.5567567567568, 10.872000000000014, 'entropy = 0.918\nsamples =
6\nvalue = [4, 2]'),
 Text(289.5567567567568, 54.360000000000014, 'entropy = 0.0\nsamples = 3\nvalue
= [3, 0]'),
 Text(292.572972972973, 76.10399999999998, 'entropy = 0.0\nsamples = 2\nvalue =
[0, 2]'),
 Text(306.90000000000003, 141.336, 'X[4] <= 0.961\nentropy = 0.557\nsamples =
77\nvalue = [67, 10]'),
 Text(298.6054054054054, 119.592, 'X[0] <= 0.964\nentropy = 0.971\nsamples =
5\nvalue = [3, 2]'),
 Text(295.5891891891892, 97.848, 'entropy = 0.0\nsamples = 2\nvalue = [0, 2]'),
 Text(301.62162162162167, 97.848, 'entropy = 0.0\nsamples = 3\nvalue = [3, 0]'),
 Text(315.1945945945946, 119.592, 'X[3] <= 0.184\nentropy = 0.503\nsamples =
72\nvalue = [64, 8]'),
 Text(307.6540540540541, 97.848, 'X[8] <= 1.701\nentropy = 0.451\nsamples =
53\nvalue = [48, 5]'),
 Text(304.6378378378379, 76.10399999999998, 'X[6] <= 69.386\nentropy =
0.552\nsamples = 39\nvalue = [34, 5]'),
 Text(301.62162162162167, 54.360000000000014, 'entropy = 0.0\nsamples = 7\nvalue
= [7, 0]'),
 Text(307.6540540540541, 54.360000000000014, 'X[1] <= 79.674\nentropy =
0.625\nsamples = 32\nvalue = [27, 5]'),
 Text(301.62162162162167, 32.615999999999985, 'X[7] <= 25.937\nentropy =
0.764\nsamples = 18\nvalue = [14, 4]'),
 Text(298.6054054054054, 10.872000000000014, 'entropy = 0.918\nsamples =
9\nvalue = [6, 3]'),
 Text(304.6378378378379, 10.872000000000014, 'entropy = 0.503\nsamples =
9\nvalue = [8, 1]'),
 Text(313.6864864864865, 32.615999999999985, 'X[2] <= 0.918\nentropy =
0.371\nsamples = 14\nvalue = [13, 1]'),
 Text(310.6702702702703, 10.872000000000014, 'entropy = 0.0\nsamples = 11\nvalue
= [11, 0]'),
 Text(316.7027027027027, 10.872000000000014, 'entropy = 0.918\nsamples =
3\nvalue = [2, 1]'),
 Text(310.6702702702703, 76.10399999999998, 'entropy = 0.0\nsamples = 14\nvalue
= [14, 0]'),
 Text(322.7351351351352, 97.848, 'X[1] <= 78.805\nentropy = 0.629\nsamples =
19\nvalue = [16, 3]'),
 Text(319.718918918919, 76.10399999999998, 'entropy = 0.0\nsamples = 5\nvalue =
[5, 0]'),
 Text(325.7513513513514, 76.10399999999998, 'X[1] <= 79.361\nentropy =
0.75\nsamples = 14\nvalue = [11, 3]'),
 Text(322.7351351351352, 54.360000000000014, 'entropy = 0.0\nsamples = 5\nvalue
= [5, 0]'),
 Text(328.7675675675676, 54.360000000000014, 'X[2] <= 0.281\nentropy =
0.918\nsamples = 9\nvalue = [6, 3]'),
 Text(325.7513513513514, 32.615999999999985, 'X[6] <= 106.869\nentropy =
1.0\nsamples = 6\nvalue = [3, 3]'),
 Text(322.7351351351352, 10.872000000000014, 'entropy = 0.971\nsamples =
5\nvalue = [3, 2]'),
 Text(328.7675675675676, 10.872000000000014, 'entropy = 0.0\nsamples = 1\nvalue
= [0, 1]'),
 Text(331.7837837837838, 32.615999999999985, 'entropy = 0.0\nsamples = 3\nvalue
= [3, 0]')]
```



[15]:
```
DTs = tree.DecisionTreeClassifier(criterion=criterions[0], splitter="best", max_depth=5, random_state=0)
DTs.fit(X_train, y_train)
y_pred = DTs.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.savefig('1.jpg', dpi = 300)
plt.show()
```

```
Prediction accuracy: 96.89%
```

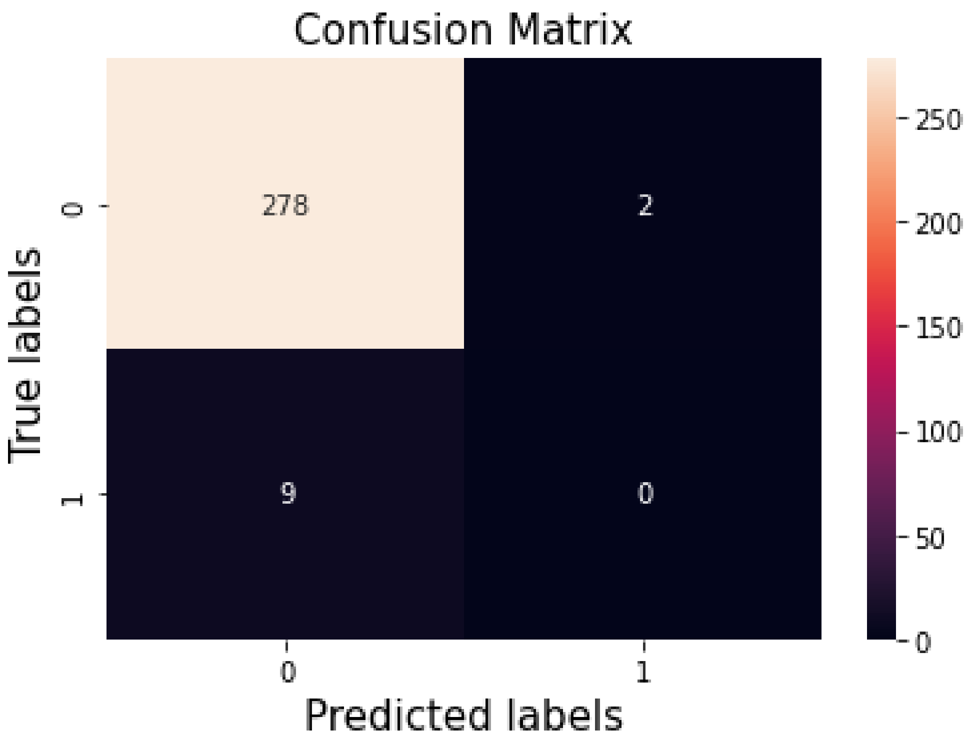


[16]:
```
n_estimators = [10, 50, 100, 250, 500]
criterions = ['gini', 'entropy']
max_depthes = [None, 2,  4, 6, 8]
best_acc = 0

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(f"Prediction accuracy: {100*best_acc:.2f}%")
```

```
Best Criterion :  gini
Best estimator :  10
Best depth :  None
Prediction accuracy: 96.18%
```

[17]:
```
RF = RandomForestClassifier(n_estimators=10, criterion="gini", max_depth=None, random_state=0)
RF.fit(X_train, y_train)
y_pred = RF.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.savefig('2.jpg', dpi = 300)
plt.show()
```

```
Prediction accuracy: 96.19%
```

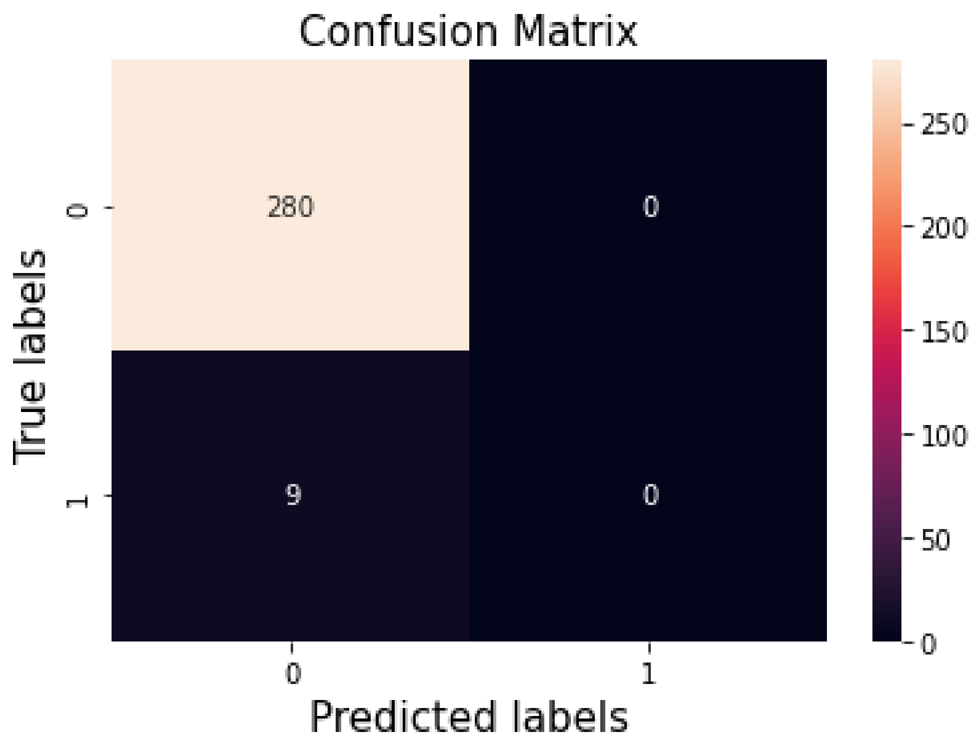


[18]:
```
# Find best parameters for KNN
best_acc = 0

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(f"Prediction accuracy: {100*best_acc:.2f}%")
```

```
Best k : 6
Prediction accuracy: 96.18%
```

[19]:
```
KNN = KNeighborsClassifier(n_neighbors=6, n_jobs=-1)
KNN.fit(X_train, y_train)
y_pred = KNN.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.savefig('3.jpg', dpi = 300)
plt.show()
```

```
Prediction accuracy: 96.89%
```



[20]:
```
y_test[y_test==0].count()
```

[20]: 280

[21]:
```
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: 95.83%
```

[22]:
```
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.savefig('4.jpg', dpi = 300)
plt.show()
```

```
Prediction accuracy: 96.89%
```