Heart Attack Prediction

Group Members:

Yashowardhan Samdhani

Adish Bhagwat

Arya Srivastava

S. Sanjith

TABLE OF CONTENTS

[Acknowledgement 6](#_Toc59725316)

[Project Objective 7](#_Toc59725317)

[Project Scope 8](#_Toc59725318)

[Data Description 9](#_Toc59725319)

[Attribute Information 9](#_Toc59725320)

[Data Type 9](#_Toc59725321)

[Null Values 10](#_Toc59725322)

[Heat map 10](#_Toc59725323)

[Info 10](#_Toc59725324)

[Data Description 10](#_Toc59725325)

[Removing Outliers 11](#_Toc59725326)

[Before 11](#_Toc59725327)

[After 15](#_Toc59725328)

[Replaced by Median 15](#_Toc59725329)

[Replaced by Median 18](#_Toc59725330)

[Data Study 21](#_Toc59725331)

[Correlation 21](#_Toc59725332)

[Parallel Coordinates 22](#_Toc59725333)

[Histogram 23](#_Toc59725334)

[Scaling 25](#_Toc59725335)

[Model Building 26](#_Toc59725336)

[Logistic Regression 26](#_Toc59725337)

[Coefficients and Intercept 26](#_Toc59725338)

[Model Implementation 27](#_Toc59725339)

[Median 27](#_Toc59725340)

[Score 27](#_Toc59725341)

[Confusion Matrix 27](#_Toc59725342)

[Normalised Confusion Matrix 27](#_Toc59725343)

[Accuracy, Precision, Recall and F1 Score 28](#_Toc59725344)

[Roc (Receiver Operating Characteristic) Curve 28](#_Toc59725345)

[AUC 28](#_Toc59725346)

[Decile Analysis 29](#_Toc59725347)

[Log Loss 29](#_Toc59725348)

[MEDIAN 30](#_Toc59725349)

[Score 30](#_Toc59725350)

[Confusion Matrix 30](#_Toc59725351)

[Normalised Confusion Matrix 30](#_Toc59725352)

[Accuracy, Precision, Recall and F1 Score 31](#_Toc59725353)

[Roc (Receiver Operating Characteristic) Curve 31](#_Toc59725354)

[AUC 31](#_Toc59725355)

[Decile Analysis 32](#_Toc59725356)

[Log Loss 32](#_Toc59725357)

[Decision Tree 33](#_Toc59725358)

[Decision Tree visualisation 33](#_Toc59725359)

[Model Implementation 34](#_Toc59725360)

[Median 34](#_Toc59725361)

[Score 34](#_Toc59725362)

[Confusion Matrix 34](#_Toc59725363)

[Normalised Confusion Matrix 34](#_Toc59725364)

[Accuracy, Precision, Recall and F1 Score 35](#_Toc59725365)

[Decision Tree 35](#_Toc59725366)

[Roc (Receiver Operating Characteristic) Curve 36](#_Toc59725367)

[AUC 36](#_Toc59725368)

[Decile Analysis 37](#_Toc59725369)

[Log Loss 37](#_Toc59725370)

[Median 38](#_Toc59725371)

[Score 38](#_Toc59725372)

[Confusion Matrix 38](#_Toc59725373)

[Normalised Confusion Matrix 38](#_Toc59725374)

[Accuracy, Precision, Recall and F1 Score 39](#_Toc59725375)

[Decision Tree 39](#_Toc59725376)

[Roc (Receiver Operating Characteristic) Curve 40](#_Toc59725377)

[AUC 40](#_Toc59725378)

[Decile Analysis 41](#_Toc59725379)

[Log Loss 41](#_Toc59725380)

[Naïve Bayes 42](#_Toc59725381)

[Types of Naïve Bayes 42](#_Toc59725382)

[Bayes Theorem 42](#_Toc59725383)

[Model Implementation 43](#_Toc59725384)

[Median 43](#_Toc59725385)

[Score 43](#_Toc59725386)

[Confusion Matrix 43](#_Toc59725387)

[Normalised Confusion Matrix 43](#_Toc59725388)

[Accuracy, Precision, Recall and F1 Score 44](#_Toc59725389)

[Roc (Receiver Operating Characteristic) Curve 44](#_Toc59725390)

[AUC 44](#_Toc59725391)

[Decile Analysis 45](#_Toc59725392)

[Log Loss 45](#_Toc59725393)

[Median 46](#_Toc59725394)

[Score 46](#_Toc59725395)

[Confusion Matrix 46](#_Toc59725396)

[Normalised Confusion Matrix 46](#_Toc59725397)

[Accuracy, Precision, Recall and F1 Score 48](#_Toc59725398)

[Roc (Receiver Operating Characteristic) Curve 48](#_Toc59725399)

[AUC 48](#_Toc59725400)

[Decile Analysis 49](#_Toc59725401)

[Log Loss 49](#_Toc59725402)

[KNN 50](#_Toc59725403)

[Parameter Selection And value of K 50](#_Toc59725404)

[Example 50](#_Toc59725405)

[Model Implementation 51](#_Toc59725406)

[Median 51](#_Toc59725407)

[K Value 51](#_Toc59725408)

[Score 52](#_Toc59725409)

[Confusion Matrix 52](#_Toc59725410)

[Normalised Confusion Matrix 52](#_Toc59725411)

[Accuracy, Precision, Recall and F1 Score 53](#_Toc59725412)

[Roc (Receiver Operating Characteristic) Curve 53](#_Toc59725413)

[AUC 53](#_Toc59725414)

[Decile Analysis 54](#_Toc59725415)

[Log Loss 54](#_Toc59725416)

[Median 55](#_Toc59725417)

[K Value 55](#_Toc59725418)

[Score 56](#_Toc59725419)

[Confusion Matrix 56](#_Toc59725420)

[Normalised Confusion Matrix 56](#_Toc59725421)

[Accuracy, Precision, Recall and F1 Score 57](#_Toc59725422)

[Roc (Receiver Operating Characteristic) Curve 57](#_Toc59725423)

[AUC 57](#_Toc59725424)

[Decile Analysis 58](#_Toc59725425)

[Log Loss 58](#_Toc59725426)

[K-Medians 59](#_Toc59725427)

[Example 59](#_Toc59725428)

[Model Implementation 60](#_Toc59725429)

[Median 60](#_Toc59725430)

[Score 60](#_Toc59725431)

[Confusion Matrix 60](#_Toc59725432)

[Normalised Confusion Matrix 60](#_Toc59725433)

[Accuracy, Precision, Recall and F1 Score 61](#_Toc59725434)

[Median 62](#_Toc59725435)

[Score 62](#_Toc59725436)

[Confusion Matrix 62](#_Toc59725437)

[Normalised Confusion Matrix 62](#_Toc59725438)

[Accuracy, Precision, Recall and F1 Score 63](#_Toc59725439)

[Model Comparison 64](#_Toc59725440)

[Code 68](#_Toc59725441)

[**Finding and Removing Outliers** 69](#_Toc59725442)

[**Scaling** 72](#_Toc59725443)

[**Models** 73](#_Toc59725444)

[**LOGISTIC REGRESSION** 73](#_Toc59725445)

[**Mean** 73](#_Toc59725446)

[**ROC AUC** 74](#_Toc59725447)

[**Decile Analysis** 78](#_Toc59725448)

[**Log Loss** 79](#_Toc59725449)

[**Median** 79](#_Toc59725450)

[**ROC AUC** 80](#_Toc59725451)

[Decile Analysis 85](#_Toc59725452)

[**Log Loss** 85](#_Toc59725453)

[**DECISION TREE** 85](#_Toc59725454)

[**Mean** 85](#_Toc59725455)

[**ROC AUC** 87](#_Toc59725456)

[**Decile Analysis** 91](#_Toc59725457)

[**Log Loss** 92](#_Toc59725458)

[**Median** 92](#_Toc59725459)

[**ROC AUC** 94](#_Toc59725460)

[**Decile Analysis** 98](#_Toc59725461)

[**Log Loss** 98](#_Toc59725462)

[**NAIVE BAYES** 99](#_Toc59725463)

[**Mean** 99](#_Toc59725464)

[**ROC AUC** 100](#_Toc59725465)

[**Decile Analysis** 104](#_Toc59725466)

[**Log Loss** 105](#_Toc59725467)

[**Median** 105](#_Toc59725468)

[**ROC AUC** 107](#_Toc59725469)

[**Decile Analysis** 111](#_Toc59725470)

[**Log Loss** 111](#_Toc59725471)

[**KNN** 112](#_Toc59725472)

[**Mean** 112](#_Toc59725473)

[**ROC AUC** 114](#_Toc59725474)

[**Decile Analysis** 118](#_Toc59725475)

[**Log Loss** 118](#_Toc59725476)

[**Median** 119](#_Toc59725477)

[**ROC AUC** 120](#_Toc59725478)

[**Decile Analysis** 125](#_Toc59725479)

[**Log Loss** 125](#_Toc59725480)

[**KMEANS** 125](#_Toc59725481)

[**Mean** 125](#_Toc59725482)

[**Median** 128](#_Toc59725483)

[Future Scope Of Improvements 131](#_Toc59725484)

[Project Certificate 132](#_Toc59725485)

[Project Certificate 133](#_Toc59725486)

[Project Certificate 134](#_Toc59725487)

[Project Certificate 135](#_Toc59725488)

# Acknowledgement

I take this opportunity to express my gratitude and deep regards to my faculty Prof. Arnab Chakraborty for his exemplary guidance, monitoring and constant encouragement throughout the course of the project. The blessing, help and guidance given by him from time to time shall carry me a long way in the journey of life on which I am about to embark.

I am obliged to my project team members for the valuable information provided by them in their respective fields. I am grateful for their cooperation during the period of my assignment

Yashowardhan Samdhani

Adish Bhagwat

Arya Srivastav

S. Sanjith

# Project Objective

Problem - A heart attack occurs when the flow of blood to the heart is blocked. The blockage is most often a build-up of fat, cholesterol, and other substances, which form a plaque in the arteries that feed the heart (coronary arteries). Data of the patient is always given to doctors through several reports. However, analysing every nook and corner of these reports of multiple patients is a tired and slow process making the possibility of human error become much higher.

Objective - The probability of the heart attack occurring or not occurring can be predicted before-hand using A.I, this in turn gives doctors indicators to take precaution or make the patience go through necessary treatments to prevent it from happening. The A.I can effectively predict this for multiple patients without taking much time, with only the expense of entering and feeding the program data.

Solution - Our project uses the Cleveland dataset to learn, analyse and predict the probability of a heart attack through various Models of machine learning. Different Models are compared with each other in order to find out a Model which gives the best prediction accuracy for said dataset.

# Project Scope

The Broad Scope of the Heart Attack Predictor A.I includes:

* less error prone prediction of the occurrence of a heart attack
* Cuts down the time taken for going through report data
* Helps and informs Doctors whether a patient needs much attention or can be attended to without haste.

# Data Description

We have taken the Health Care: Dataset on Heart attack possibility

## Attribute Information

1. age
2. sex: 0 = female; 1 = male
3. chest pain type (4 values)
4. resting blood pressure
5. serum cholesterol in mg/dl
6. fasting blood sugar > 120 mg/dl
7. resting electrocardiographic results (values 0,1,2)
8. maximum heart rate achieved
9. exercise induced angina
10. oldpeak = ST depression induced by exercise relative to rest
11. the slope of the peak exercise ST segment
12. number of major vessels (0-3) coloured by fluoroscopy
13. thal: 0 = normal; 1 = fixed defect; 2 = reversable defect; 3 = irreversible defect
14. target: 0= less chance of heart attack 1= more chance of heart attack

## Data Type

1. age = Continuous
2. sex = Categorical
3. cp (4 values) = Categorical
4. trestbps = Continuous
5. chol in mg/dl = Continuous
6. fbs > 120 mg/dl = Categorical
7. restecg (values 0,1,2) = Categorical
8. thalach = Continuous
9. exang = Categorical
10. oldpeak = Continuous
11. slope = Categorical
12. ca = Categorical
13. thal = Categorical
14. target = Categorical

## Null Values

The data set has no null values

### Heat map

## Info

### Data Description

# Removing Outliers

The number of outliers in each feature variable are:

1. age = 0
2. sex = 0
3. cp (4 values) = 0
4. trestbps = 9
5. chol in mg/dl = 14
6. fbs > 120 mg/dl = 0
7. restecg (values 0,1,2) = 0
8. thalach = 45
9. exang = 0
10. oldpeak = 49
11. slope = 0
12. ca = 0
13. thal = 0

### Before

Chart, bar chart, box and whisker chart

Description automatically generated

Chart, box and whisker chart

Description automatically generatedChart, box and whisker chart

Description automatically generated

Chart, bar chart, box and whisker chart

Description automatically generatedChart, bar chart

Description automatically generated

Chart, box and whisker chart

Description automatically generated

### After

#### Chart, bar chart Description automatically generatedReplaced by Median

Chart, bar chart, box and whisker chart

Description automatically generated

Chart, bar chart

Description automatically generated

Chart, bar chart, box and whisker chart

Description automatically generated

Chart

Description automatically generated

Chart, bar chart

Description automatically generated

#### Chart, bar chart Description automatically generatedReplaced by Median

Chart, bar chart, box and whisker chart

Description automatically generated

Chart, bar chart, box and whisker chart

Description automatically generatedChart, bar chart

Description automatically generated

Chart, bar chart

Description automatically generatedChart

Description automatically generated

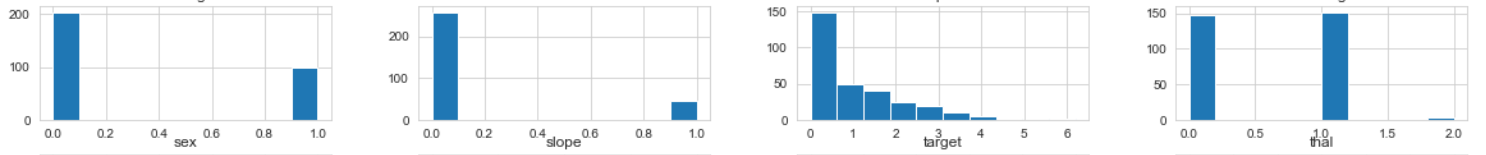
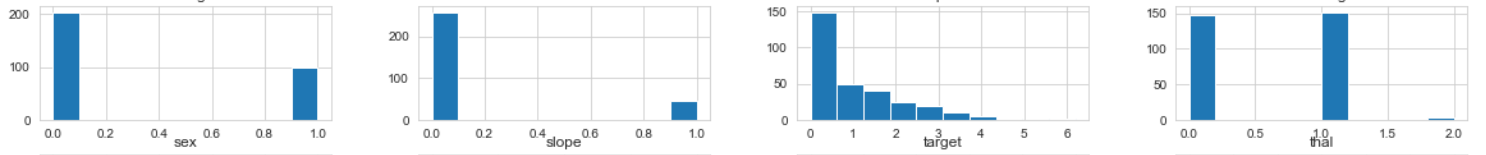
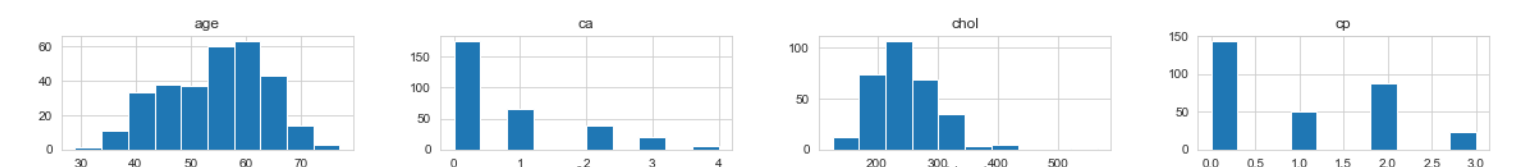
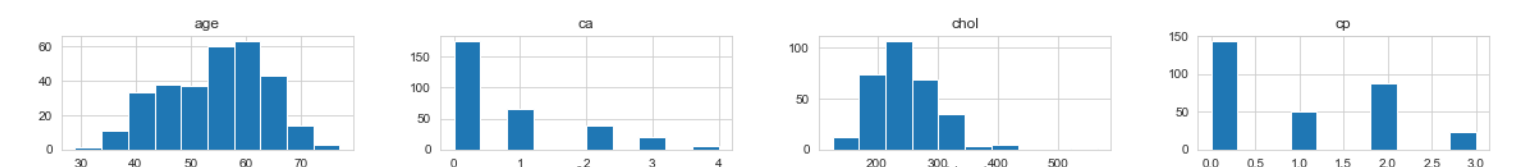
# Data Study

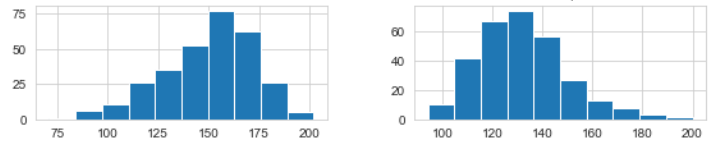
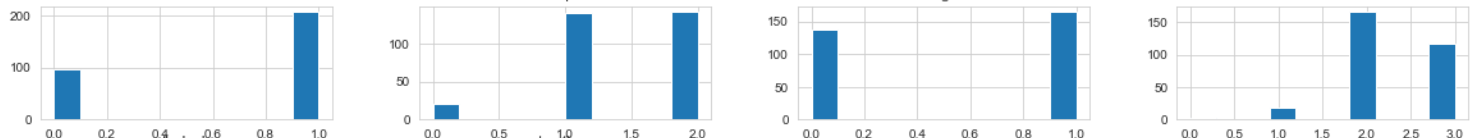
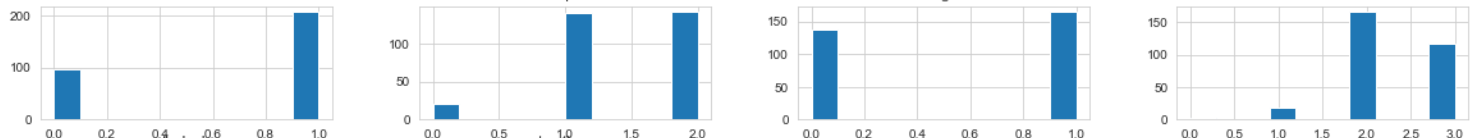
## Correlation



## Parallel Coordinates

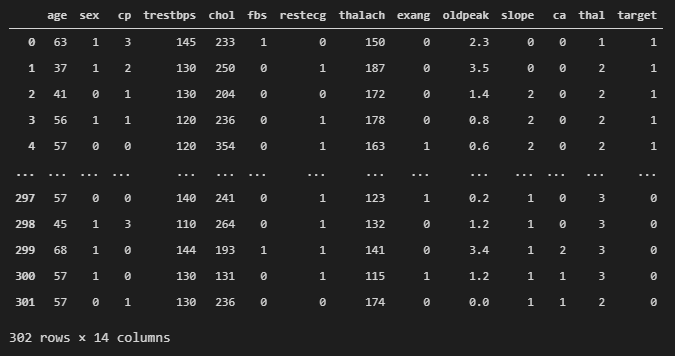
## Histogram





# Scaling

After the study of the data, the data is scaled. The following is the tabular representation of the scaled data:



# Model Building

## Logistic Regression

Logistic regression is a statistical Model that in its basic form uses a logistic function to Model a binary dependent/target variable, although many more complex extensions exist. In regression analysis, logistic regression is estimating the parameters of a logistic Model. Logistic regression is a method for analysing a dataset in which there are one or more independent variables that determine an outcome. The outcome is measured with a dichotomous variable (in which there are only two possible outcomes.

### Coefficients and Intercept

|  |  |
| --- | --- |
| Attribute | Coefficients |
| Age | -0.07 |
| Sex | -0.60 |
| Cp | 1.12 |
| Trestbps | -0.03 |
| Chol | -0.10 |
| Fbs | 0.01 |
| Restecg | 0.32 |
| Thalach | -0.001 |
| Exang | -0.59 |
| Oldpeak | -0.65 |
| Slope | 0.62 |
| Ca | -1.44 |
| Thal | 0.83 |

The Intercept of the Model is -0.18444558

### Model Implementation

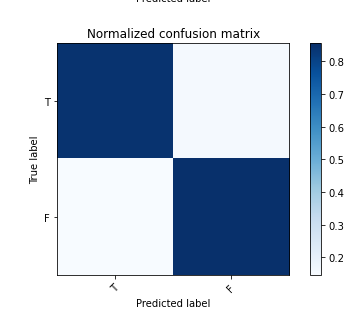
#### Median

##### Score

Score: 0.8512396694214877

##### Confusion Matrix

##### Normalised Confusion Matrix



##### Accuracy, Precision, Recall and F1 Score

Accuracy: 0.8512396694214877

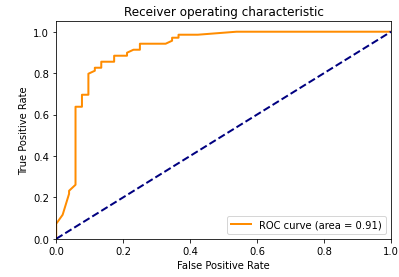
Recall: 0.855072463768116

Precision: 0.8805970149253731

F1: 0.8676470588235295

##### Roc (Receiver Operating Characteristic) Curve

Optimal threshold value: 0.52



##### AUC

The AUC score of the Model is 0.9109531772575251

##### Decile Analysis

##### Log Loss

Log loss: 0.379049368470849

#### MEDIAN

##### Score

Score: 0.8524590163934426

##### Confusion Matrix

##### Normalised Confusion Matrix



##### Accuracy, Precision, Recall and F1 Score

Accuracy: 0.8524590163934426

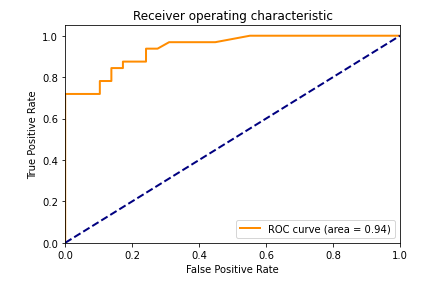
Recall: 0.84375

Precision: 0.8709677419354839

F1: 0.8571428571428571

##### Roc (Receiver Operating Characteristic) Curve

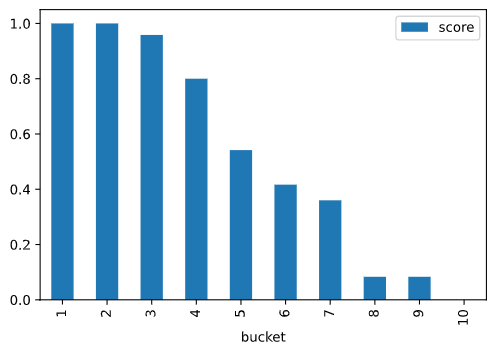
Optimal threshold value: 0.77



##### AUC

The AUC score of the Model is 0.939655172413793

##### Decile Analysis



##### Log Loss

Log loss: 0.3365017438911693

## Decision Tree

A decision tree is a flowchart-like structure in which each internal node represents a "test" on an attribute (e.g whether a coin flip comes up heads or tails), each branch represents the outcome of the test, and each leaf node represents a class label (decision taken after computing all attributes). A tree can be “learned” by splitting the source set into subsets based on an attribute value test. This process is repeated on each derived subset in a recursive manner called recursive partitioning. The recursion is completed when the subset at a node all has the same value of the target variable, or when splitting no longer adds value to the predictions. The construction of decision tree classifier does not require any domain knowledge or parameter setting, and therefore is appropriate for exploratory knowledge discovery. Decision trees can handle high dimensional data. In general decision tree classifier has good accuracy. Decision tree induction is a typical inductive approach to learn knowledge on classification.

### Decision Tree visualisation

### Model Implementation

#### Median

##### Score

Score: 0.743801652892562

##### Confusion Matrix

##### Normalised Confusion Matrix

##### Accuracy, Precision, Recall and F1 Score

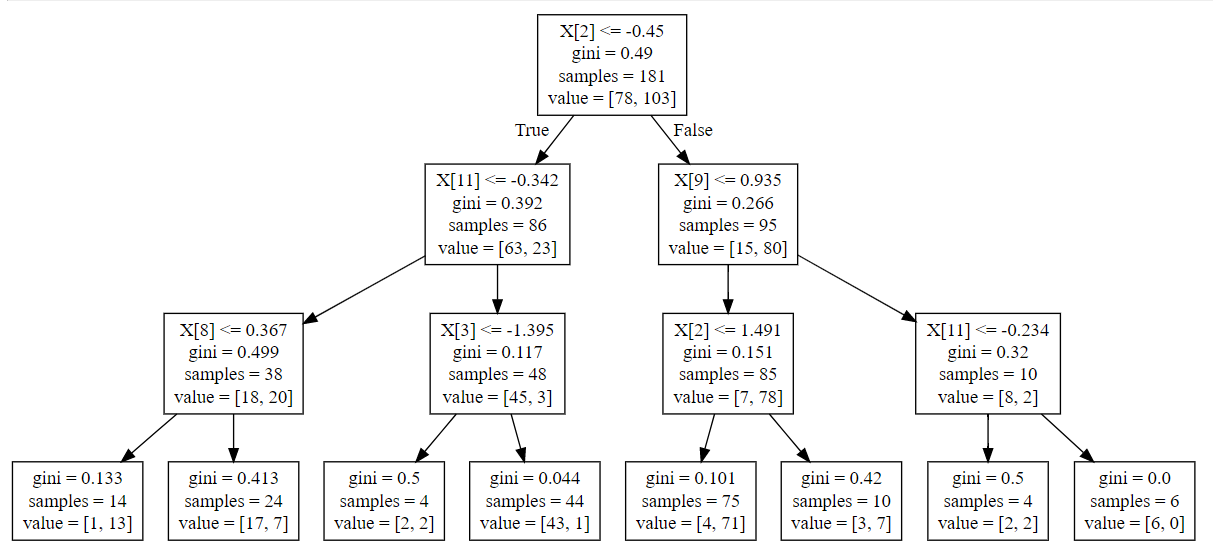
Accuracy: 0.743801652892562

Recall: 0.8852459016393442

Precision: 0.6923076923076923

F1: 0.7769784172661871

##### Decision Tree



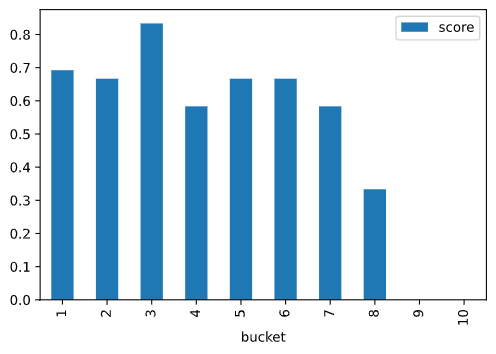
##### Roc (Receiver Operating Characteristic) Curve

Optimal threshold value: 0.5

##### AUC

The AUC score of the Model is 0.7624316939890711

##### Decile Analysis



##### Log Loss

Log loss: 0.7230374691946901

#### Median

##### Score

Score: 0.743801652892562

##### Confusion Matrix

##### Normalised Confusion Matrix

##### Accuracy, Precision, Recall and F1 Score

Accuracy: 0.743801652892562

Recall: 0.8524590163934426

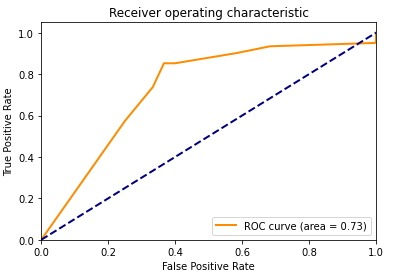
Precision: 0.7027027027027027

F1: 0.7703703703703704

##### Decision Tree

##### Roc (Receiver Operating Characteristic) Curve

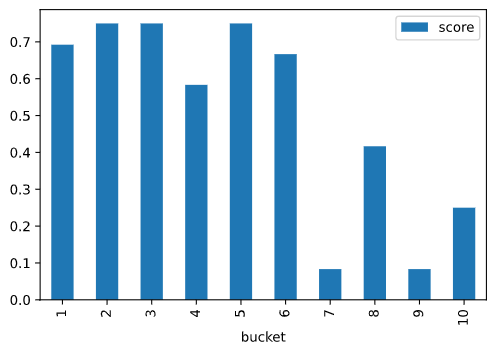
Optimal threshold value: 0.7



##### AUC

The AUC score of the Model is 0. 0.7323770491803279

##### Decile Analysis



##### Log Loss

Log loss: 1.5152060500196904

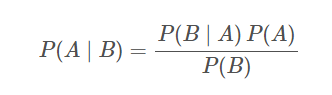
## Naïve Bayes

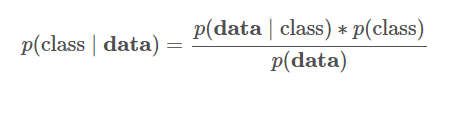
In statistics, Naive Bayes classifiers are a family of simple "probabilistic classifiers" based on applying Bayes' theorem with strong independence assumptions between the features.

### Types of Naïve Bayes

1. Multinomial Naive Bayes: Feature vectors represent the frequencies with which certain events have been generated by a multinomial distribution. This is the event Model typically used for document classification.
2. Bernoulli Naive Bayes: In the multivariate Bernoulli event Model, features are independent Booleans (binary variables) describing inputs. Like the multinomial Model, this Model is popular for document classification.
3. Gaussian Naïve Bayes: This extension of naive Bayes used here is called Gaussian Naive Bayes. Other functions can be used to estimate the distribution of the data, but the Gaussian (or Normal distribution) is the easiest to work with because you only need to estimate the Median and the standard deviation from your training data.

### Bayes Theorem

Bayes theorem is a famous equation that allows us to make predictions based on data. Here is the classic version of the Bayes theorem:

This might be too abstract, so let us replace some of the variables to make it more concrete. In a bayes classifier, we are interested in finding out the class (e.g. male or female, spam or ham) of an observation given the data:

where:

* class is a particular class (e.g., male)
* data is an observation's data
* p(class I data) is called the posterior
* p(data | class) is called the likelihood
* p(class) is called the prior
* p(data) is called the marginal probability

### Model Implementation

#### Median

##### Score

Score: 0.7520661157024794

##### Confusion Matrix

##### Normalised Confusion Matrix

##### Accuracy, Precision, Recall and F1 Score

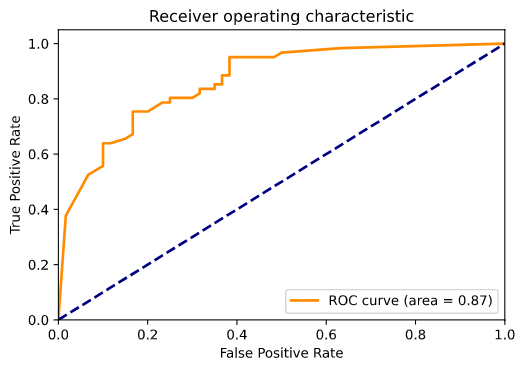
Accuracy: 0.7520661157024794

Recall: 0.8360655737704918

Precision: 0.7183098591549296

F1: 0.7727272727272727

##### Roc (Receiver Operating Characteristic) Curve

Optimal threshold value: 0.84

##### AUC

The AUC score of the Model is 0.8662568306010929

##### Decile Analysis

##### Chart Description automatically generatedLog Loss

Log loss: 1.1112545612072136

#### Median

##### Score

Score: 0.743801652892562

##### Confusion Matrix

##### Normalised Confusion Matrix



##### Accuracy, Precision, Recall and F1 Score

Accuracy: 0.743801652892562

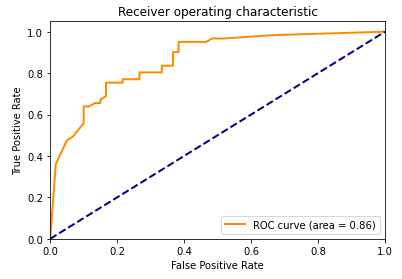
Recall: 0.8360655737704918

Precision: 0.7083333333333334

F1: 0.7669172932330828

##### Roc (Receiver Operating Characteristic) Curve

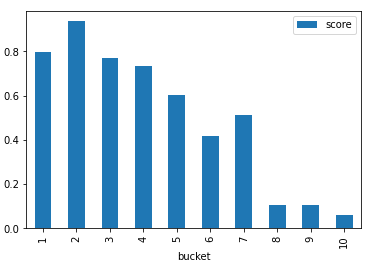
Optimal threshold value: 0.82



##### AUC

The AUC score of the Model is 0.8632513661202186

##### Decile Analysis



##### Log Loss

Log loss: 1.1106203966238264

## KNN

In [statistics](https://en.wikipedia.org/wiki/Statistics), the *k*-nearest neighbours algorithm (*k*-NN) is a [non-parametric](https://en.wikipedia.org/wiki/Non-parametric_statistics) method proposed by [Thomas Cover](https://en.wikipedia.org/wiki/Thomas_M._Cover) used for [classification](https://en.wikipedia.org/wiki/Statistical_classification) and [regression](https://en.wikipedia.org/wiki/Regression_analysis).[[1]](https://en.wikipedia.org/wiki/K-nearest_neighbors_algorithm#cite_note-1) In both cases, the input consists of the *k* closest training examples in the [feature space](https://en.wikipedia.org/wiki/Feature_space). The output depends on whether *k*-NN is used for classification or regression:

* In *k-NN classification*, the output is a class membership. An object is classified by a plurality vote of its neighbours, with the object being assigned to the class most common among its *k* nearest neighbours (*k* is a positive [integer](https://en.wikipedia.org/wiki/Integer), typically small). If *k* = 1, then the object is simply assigned to the class of that single nearest neighbour.
* In *k-NN regression*, the output is the property value for the object. This value is the average of the values of *k* nearest neighbours.

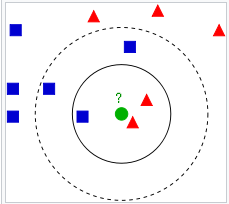
### Parameter Selection And value of K

The best choice of *k* depends upon the data; generally, larger values of *k* reduces effect of the noise on the classification but make boundaries between classes less distinct. A good *k* can be selected by various techniques. The special case where the class is predicted to be the class of the closest training sample (i.e. when *k* = 1) is called the nearest neighbour algorithm.

In binary (two class) classification problems, it is helpful to choose *k* to be an odd number as this avoids tied votes. One popular way of choosing the empirically optimal *k* in this setting is via bootstrap method

### Example

Example of *k*-NN classification. The test sample (green dot) should be classified either to blue squares or to red triangles. If *k = 3* (solid line circle) it is assigned to the red triangles because there are 2 triangles and only 1 square inside the inner circle. If *k = 5* (dashed line circle) it is assigned to the blue squares (3 squares vs. 2 triangles inside the outer circle)



### Model Implementation

#### Median

##### K Value

Chart, line chart

Description automatically generated

##### Score

Score: 0.8287292817679558

##### Confusion Matrix

##### Normalised Confusion Matrix



##### Accuracy, Precision, Recall and F1 Score

Accuracy: 0.8347107438016529

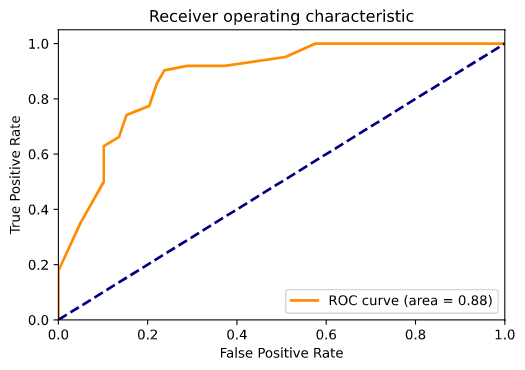
Recall: 0.9032258064516129

Precision: 0.8

F1: 0 .8484848484848486

##### Roc (Receiver Operating Characteristic) Curve

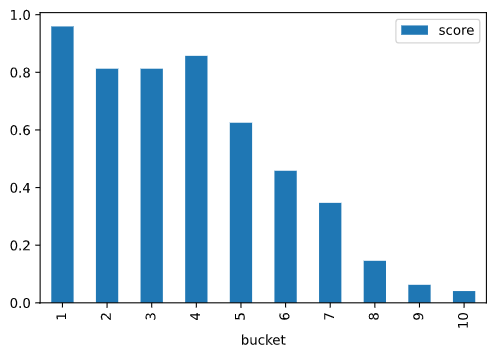
Optimal threshold value: 0.5294117647058824



##### AUC

The AUC score of the Model is 0.877255330781848

##### Decile Analysis

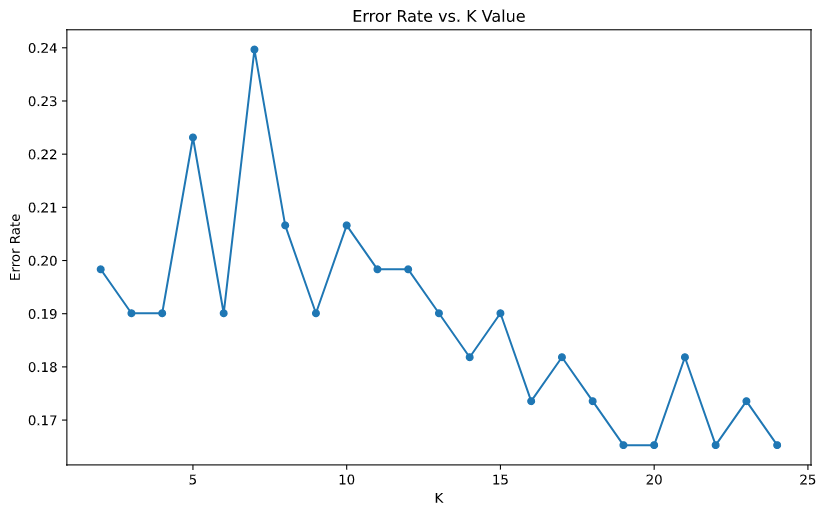


##### Log Loss

Log loss: 0.42653563898605484

#### Median

##### K Value



##### Score

Score: 0.8397790055248618

##### Confusion Matrix

##### Normalised Confusion Matrix



##### Accuracy, Precision, Recall and F1 Score

Accuracy: 0.8347107438016529

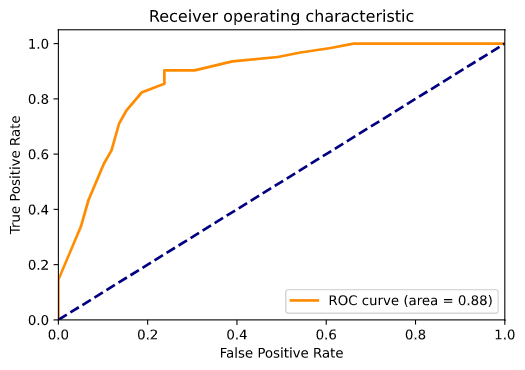
Recall: 0.9032258064516129

Precision: 0.8

F1: 0.8484848484848486

##### Roc (Receiver Operating Characteristic) Curve

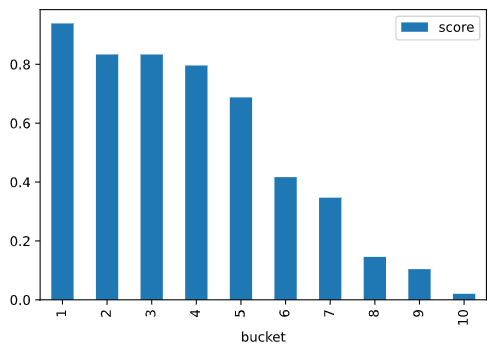
Optimal threshold value: 0.5263157894736842



##### AUC

The AUC score of the Model is 0.8783488244942592

##### Decile Analysis



##### Log Loss

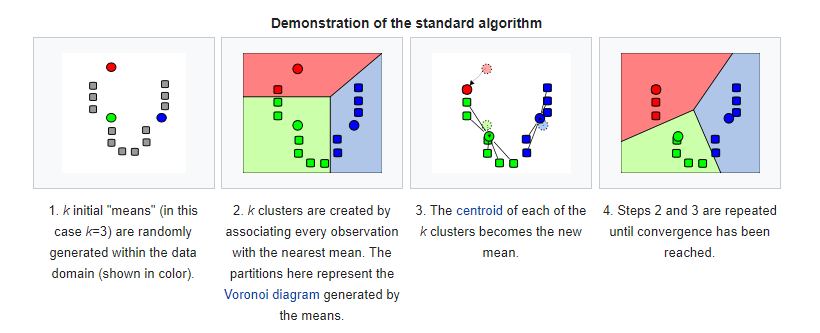
Log loss: 0.4490046250780085

## K-Medians

*k*-Medians clustering is a method of [vector quantization](https://en.wikipedia.org/wiki/Vector_quantization), originally from [signal processing](https://en.wikipedia.org/wiki/Signal_processing), that aims to [partition](https://en.wikipedia.org/wiki/Partition_of_a_set) *n* observations into *k* clusters in which each observation belongs to the [cluster](https://en.wikipedia.org/wiki/Cluster_(statistics)) with the nearest [Median](https://en.wikipedia.org/wiki/Mean) (cluster centers or cluster [centroid](https://en.wikipedia.org/wiki/Centroid)), serving as a prototype of the cluster. This results in a partitioning of the data space into [voronoi cells](https://en.wikipedia.org/wiki/Voronoi_cell). *k*-Medians clustering minimizes within-cluster variances ([squared euclidean distances](https://en.wikipedia.org/wiki/Squared_Euclidean_distance)), but not regular euclidean distances, which would be the more difficult [weber problem](https://en.wikipedia.org/wiki/Weber_problem): the Median optimizes squared errors, whereas only the [geometric Median](https://en.wikipedia.org/wiki/Geometric_median) minimizes euclidean distances. For instance, better euclidean solutions can be found using [k-Medians](https://en.wikipedia.org/wiki/K-medians_clustering) and [k-medoids](https://en.wikipedia.org/wiki/K-medoids).

### Example

Commonly used initialization methods are Forgy and Random Partition. The Forgy method randomly chooses *k* observations from the dataset and uses these as the initial Medians. The Random Partition method first randomly assigns a cluster to each observation and then proceeds to the update step, thus computing the initial Median to be the centroid of the cluster's randomly assigned points. The Forgy method tends to spread the initial Medians out, while Random Partition places all of them close to the center of the data set. For expectation maximization and standard *k*-Medians algorithms, the Forgy method of initialization is preferable.

* 

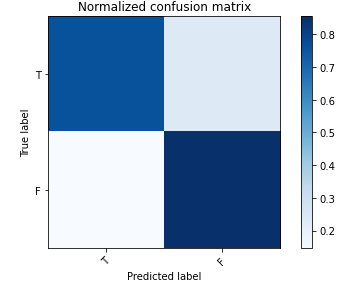
### Model Implementation

#### Median

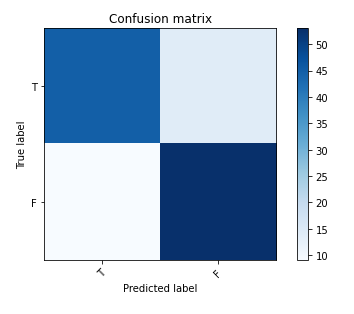
##### Score

Score: -1360.6191878608547

##### Confusion Matrix



##### Normalised Confusion Matrix



##### Accuracy, Precision, Recall and F1 Score

Accuracy: 0.8099173553719008

Recall: 0.8548387096774194

Precision: 0.7910447761194029

F1: 0.8217054263565892

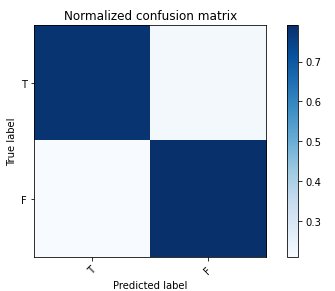
#### Median

##### Score

Score: -1371.377612651223

##### Confusion Matrix

##### Normalised Confusion Matrix



##### Accuracy, Precision, Recall and F1 Score

Accuracy: 0.7851239669421488

Recall: 0.7903225806451613

Precision: 0.7903225806451613

F1: 0.7903225806451614

## Model Comparison

|  |  |  |
| --- | --- | --- |
| ACCURACY | | |
| MODEL | **MEDIAN** | **MEDIAN** |
| Logistic Regression | 0.8512396694214877 | 0.8524590163934426 |
| Decision Tree | 0.743801652892562 | 0.743801652892562 |
| Naïve Bayes | 0.7520661157024794 | 0.743801652892562 |
| KNN | 0.8347107438016529 | 0.8347107438016529 |
| K Medians | 0.8099173553719008 | 0.7851239669421488 |
| RECALL | | |
| MODEL | **MEDIAN** | **MEDIAN** |
| Logistic Regression | 0.855072463768116 | 0.84375 |
| Decision Tree | 0.8852459016393442 | 0.8524590163934426 |
| Naïve Bayes | 0.8360655737704918 | 0.8360655737704918 |
| KNN | 0.9032258064516129 | 0.9032258064516129 |
| K Medians | 0.8548387096774194 | 0.7903225806451613 |
| PRECISION | | |
| MODEL | **MEDIAN** | **MEDIAN** |
| Logistic Regression | 0.8805970149253731 | 0.8709677419354839 |
| Decision Tree | 0.6923076923076923 | 0.7027027027027027 |
| Naïve Bayes | 0.7183098591549296 | 0.7083333333333334 |
| KNN | 0.8 | 0.8 |
| K Medians | 0.7910447761194029 | 0.7903225806451613738 |
| F1 | | |
| MODEL | **MEDIAN** | **MEDIAN** |
| Logistic Regression | 0.8676470588235295 | 0.8571428571428571 |
| Decision Tree | 0.7769784172661871 | 0.7703703703703704 |
| Naïve Bayes | 0.7727272727272727 | 0.7669172932330828 |
| KNN | 0.8484848484848486 | 0.8484848484848486 |
| K Medians | 0.8217054263565892 | 0.7903225806451614 |
| Log Loss | | |
| MODEL | **MEDIAN** | **MEDIAN** |
| Logistic Regression | 0.379049368470849 | 0.3365017438911693 |
| Decision Tree | 0.7230374691946901 | 1.5152060500196904 |
| Naïve Bayes | 1.1112545612072136 | 1.1106203966238264 |
| KNN | 0.42653563898605484 | 0.4490046250780085 |

Therefore, Accuracy is highest for Logistic Regression with outliers replaced with Median.

Therefore, Recall is highest for KNN, irrespective of how the outliers are removed.

Therefore, precision is highest for Logistic Regression, with the outliers replaced with the Median.

Therefore, F1 score is highest for Logistic Regression, with the outliers replaced with the Median.

Therefore, Log Loss is least for Logistic Regression, where it is lowest when outliers are replaced with Median, and second lowest when they are replaced with Median.

Thus, **Logistic regression with outliers replaced by Median** of the column performs best for our Dataset.

# Code

**import** **numpy** **as** **np**

**import** **pandas** **as** **pd**

**import** **matplotlib.pyplot** **as** **plt**

%**matplotlib** inline

**import** **seaborn** **as** **sns**

**from** **pandas.plotting** **import** parallel\_coordinates

**from** **sklearn** **import** neighbors

**from** **sklearn** **import** preprocessing

**from** **sklearn** **import** metrics

**from** **sklearn** **import** tree

**from** **sklearn.metrics** **import** roc\_curve, auc

**from** **sklearn.metrics** **import** confusion\_matrix

**from** **sklearn.metrics** **import** accuracy\_score

**from** **sklearn.metrics** **import** classification\_report

**from** **sklearn.model\_selection** **import** train\_test\_split

**from** **sklearn.linear\_model** **import** LogisticRegression

**from** **sklearn.tree** **import** DecisionTreeClassifier

**from** **sklearn.tree** **import** export\_graphviz

**from** **sklearn.naive\_bayes** **import** GaussianNB

**from** **sklearn.cluster** **import** KMeans

df = pd.read\_csv("E:/heart.csv")

heart = pd.read\_csv("E:/heart.csv")

In [ ]:

df.head()

In [ ]:

df.info()

In [ ]:

df.describe()

In [ ]:

plt.figure(figsize = (15,10))

sns.heatmap(df.corr(), annot = **True**, annot\_kws = {"size":12})

In [ ]:

plt.figure(figsize = (15,10))

sns.countplot(x = "target", data = df, palette = "RdBu\_r")

In [ ]:

plt.figure(figsize = (15,10))

parallel\_coordinates(df, 'target', colormap=plt.get\_cmap("Set2"))

plt.show()

In [ ]:

df.hist()

In [ ]:

fig, ax=plt.subplots(5,3,figsize=(20,28))

sns.distplot(df['age'],bins=10,ax=ax[0,0],axlabel='Age Distribution')

sns.countplot(x="sex", data=df,ax=ax[0,1])

sns.countplot(x="cp", data=df,ax=ax[0,2])

sns.distplot(df['trestbps'],bins=10,ax=ax[1,0],axlabel='resting blood pressure')

sns.distplot(df['chol'],bins=10,ax=ax[1,1],axlabel='serum cholestoral in mg/dl')

sns.countplot(x="fbs", data=df,ax=ax[1,2])

sns.countplot(x="restecg", data=df,ax=ax[2,0])

sns.distplot(df['thalach'],bins=10,ax=ax[2,1],axlabel='maximum heart rate achieved')

sns.countplot(x="exang", data=df,ax=ax[2,2])

sns.distplot(df['oldpeak'],bins=10,ax=ax[3,0],axlabel='ST depression induced by exercise relative to rest')

sns.countplot(x='slope',data=df,ax=ax[3,1])

sns.countplot(x='ca',data=df,ax=ax[3,2])

sns.countplot(x='thal',data=df,ax=ax[4,0])

sns.countplot(x='target',data=df,ax=ax[4,1])

sns.countplot(x='target',hue='sex',data=heart,palette='rainbow')

ax[4,2].set\_title('Sex: Female v Male')

ax[4,1].set\_title('target')

ax[4,0].set\_title('thal')

ax[3,2].set\_title('number of major vessels (0-3) colored by flourosopy')

ax[3,1].set\_title('the slope of the peak exercise ST segment')

ax[2,2].set\_title('exercise induced angina')

ax[1,2].set\_title("fasting blood sugar > 120 mg/dl")

ax[0,2].set\_title("chest pain type")

ax[2,0].set\_title('resting electrocardiographic results')

## **Finding and Removing Outliers**

In [ ]:

**for** column **in** heart.drop("target", axis = 1).columns:

plt.figure(figsize = (5,5))

sns.boxplot(y = heart[column])

In [ ]:

Q1 = np.percentile(heart.age, 25)

Q3 = np.percentile(heart.age, 75)

IQR = Q3 - Q1

low\_lim = Q1 - 1.5 \* IQR

up\_lim = Q3 + 1.5 \* IQR

**for** x **in** heart.age:

**if** (x<low\_lim) **or** (x>up\_lim):

heart['age'] = heart['age'].replace({x:heart.age.mean()})

Q1 = np.percentile(heart.trestbps, 25)

Q3 = np.percentile(heart.trestbps, 75)

IQR = Q3 - Q1

low\_lim = Q1 - 1.5 \* IQR

up\_lim = Q3 + 1.5 \* IQR

**for** x **in** heart.trestbps:

**if** (x<low\_lim) **or** (x>up\_lim):

heart['trestbps'] = heart['trestbps'].replace({x:heart.trestbps.mean()})

Q1 = np.percentile(heart.chol, 25)

Q3 = np.percentile(heart.chol, 75)

IQR = Q3 - Q1

low\_lim = Q1 - 1.5 \* IQR

up\_lim = Q3 + 1.5 \* IQR

**for** x **in** heart.chol:

**if** (x<low\_lim) **or** (x>up\_lim):

heart['chol'] = heart['chol'].replace({x:heart.chol.mean()})

Q1 = np.percentile(heart.thalach, 40)

Q3 = np.percentile(heart.thalach, 75)

IQR = Q3 - Q1

low\_lim = Q1 - 1.5 \* IQR

up\_lim = Q3 + 1.5 \* IQR

**for** x **in** heart.thalach:

**if** (x<low\_lim) **or** (x>up\_lim):

heart['thalach'] = heart['thalach'].replace({x:heart.thalach.mean()})

Q1 = np.percentile(heart.oldpeak, 25)

Q3 = np.percentile(heart.oldpeak, 75)

IQR = Q3 - Q1

low\_lim = Q1 - 1.5 \* IQR

up\_lim = Q3 + 1.5 \* IQR

**for** x **in** heart.oldpeak:

**if** (x<low\_lim) **or** (x>up\_lim):

heart['oldpeak'] = heart['oldpeak'].replace({x:heart.oldpeak.mean()})

Q1 = np.percentile(heart.ca, 25)

Q3 = np.percentile(heart.ca, 75)

IQR = Q3 - Q1

low\_lim = Q1 - 1.5 \* IQR

up\_lim = Q3 + 1.5 \* IQR

**for** x **in** heart.ca:

**if** (x<low\_lim) **or** (x>up\_lim):

heart['ca'] = heart['ca'].replace({x:heart.ca.mean()})

Q1 = np.percentile(heart.thal, 25)

Q3 = np.percentile(heart.thal, 75)

IQR = Q3 - Q1

low\_lim = Q1 - 1.5 \* IQR

up\_lim = Q3 + 1.5 \* IQR

**for** x **in** heart.thal:

**if** (x<low\_lim) **or** (x>up\_lim):

heart['thal'] = heart['thal'].replace({x:heart.thal.mean()})

In [ ]:

meanheart = pd.DataFrame(heart)

**for** column **in** meanheart.drop("target", axis = 1).columns:

plt.figure(figsize = (5,5))

sns.boxplot(y = meanheart[column])

In [ ]:

heart = pd.read\_csv("E:/heart.csv")

In [ ]:

Q1 = np.percentile(heart.age, 25)

Q3 = np.percentile(heart.age, 75)

IQR = Q3 - Q1

low\_lim = Q1 - 1.5 \* IQR

up\_lim = Q3 + 1.5 \* IQR

**for** x **in** heart.age:

**if** (x<low\_lim) **or** (x>up\_lim):

heart['age'] = heart['age'].replace({x:heart.age.median()})

Q1 = np.percentile(heart.trestbps, 25)

Q3 = np.percentile(heart.trestbps, 75)

IQR = Q3 - Q1

low\_lim = Q1 - 1.5 \* IQR

up\_lim = Q3 + 1.5 \* IQR

**for** x **in** heart.trestbps:

**if** (x<low\_lim) **or** (x>up\_lim):

heart['trestbps'] = heart['trestbps'].replace({x:heart.trestbps.median()})

Q1 = np.percentile(heart.chol, 25)

Q3 = np.percentile(heart.chol, 75)

IQR = Q3 - Q1

low\_lim = Q1 - 1.5 \* IQR

up\_lim = Q3 + 1.5 \* IQR

**for** x **in** heart.chol:

**if** (x<low\_lim) **or** (x>up\_lim):

heart['chol'] = heart['chol'].replace({x:heart.chol.median()})

Q1 = np.percentile(heart.thalach, 40)

Q3 = np.percentile(heart.thalach, 75)

IQR = Q3 - Q1

low\_lim = Q1 - 1.5 \* IQR

up\_lim = Q3 + 1.5 \* IQR

**for** x **in** heart.thalach:

**if** (x<low\_lim) **or** (x>up\_lim):

heart['thalach'] = heart['thalach'].replace({x:heart.thalach.median()})

Q1 = np.percentile(heart.oldpeak, 25)

Q3 = np.percentile(heart.oldpeak, 75)

IQR = Q3 - Q1

low\_lim = Q1 - 1.5 \* IQR

up\_lim = Q3 + 1.5 \* IQR

**for** x **in** heart.oldpeak:

**if** (x<low\_lim) **or** (x>up\_lim):

heart['oldpeak'] = heart['oldpeak'].replace({x:heart.oldpeak.median()})

Q1 = np.percentile(heart.ca, 25)

Q3 = np.percentile(heart.ca, 75)

IQR = Q3 - Q1

low\_lim = Q1 - 1.5 \* IQR

up\_lim = Q3 + 1.5 \* IQR

**for** x **in** heart.ca:

**if** (x<low\_lim) **or** (x>up\_lim):

heart['ca'] = heart['ca'].replace({x:heart.ca.median()})

Q1 = np.percentile(heart.thal, 25)

Q3 = np.percentile(heart.thal, 75)

IQR = Q3 - Q1

low\_lim = Q1 - 1.5 \* IQR

up\_lim = Q3 + 1.5 \* IQR

**for** x **in** heart.thal:

**if** (x<low\_lim) **or** (x>up\_lim):

heart['thal'] = heart['thal'].replace({x:heart.thal.median()})

In [ ]:

medianheart = pd.DataFrame(heart)

**for** column **in** medianheart.drop("target", axis = 1).columns:

plt.figure(figsize = (5,5))

sns.boxplot(y = medianheart[column])

## **Scaling**

In [ ]:

df\_scaled = pd.DataFrame(preprocessing.scale(df.drop("target", axis = 1)), columns = df.drop("target", axis = 1).columns).join(df.target)

meanheart = pd.DataFrame(preprocessing.scale(meanheart.drop("target", axis = 1)), columns = meanheart.drop("target", axis = 1).columns).join(meanheart.target)

medianheart = pd.DataFrame(preprocessing.scale(medianheart.drop("target", axis = 1)), columns = medianheart.drop("target", axis = 1).columns).join(medianheart.target)

In [ ]:

df

In [ ]:

meanheart

In [ ]:

Medianheart

## **Models**

### **LOGISTIC REGRESSION**

#### **Mean**

In [ ]:

X\_train, X\_test, y\_train, y\_test = train\_test\_split(meanheart.drop("target", axis = 1), meanheart.target, test\_size=0.4,random\_state=42)

logreg = LogisticRegression(max\_iter=3000) *# set the max iteration to be 3000 otherwise the process can't be finished*

logreg.fit(X\_train,y\_train)

print ("Trained Model:", logreg, "**\n**")

y\_pred = logreg.predict(X\_test)

*# view the model's score, which will indicate how good my model has been trained*

print("Score : ", accuracy\_score(y\_test, y\_pred, normalize = **True**))

In [ ]:

*# we can even look at the probabilities the learner assigned to each class*

y\_pred\_proba = logreg.predict\_proba(X\_test).round(2)

print(y\_pred\_proba, "**\n**")

In [ ]:

*# Plot a confusion matrix.*

*# cm is the confusion matrix, names are the names of the classes.*

**def** plot\_confusion\_matrix(cm, names, title = 'Confusion matrix', cmap = plt.cm.Blues):

*# plt.imshow displays the image on the axes, but if you need to*

*# display multiple images you use show() to finish the figure.*

*# interpolation = 'none': works well when a big image is scaled down*

*# interpolation = 'nearest': works well when a small image is scaled up*

*# cmap: The registered colormap name used to map scalar data to colors.*

plt.imshow(cm, interpolation='nearest', cmap = cmap)

plt.title(title)

plt.colorbar()

tick\_marks = np.arange(len(names))

plt.xticks(tick\_marks, names, rotation = 45)

plt.yticks(tick\_marks, names)

*# Automatically adjust subplot parameters to give specified padding.*

plt.tight\_layout()

plt.ylabel('True label')

plt.xlabel('Predicted label')

labels = ['T', 'F']

*# Compute confusion matrix*

cm = confusion\_matrix(y\_test, y\_pred)

np.set\_printoptions(precision = 2)

print('Confusion matrix, without normalization')

print(cm)

plt.figure()

plot\_confusion\_matrix(cm, labels)

*# Normalize the confusion matrix by row*

*# (i.e. by the number of samples in each class)*

cm\_normalized = cm.astype('float') / cm.sum(axis = 1)[:, np.newaxis]

print('Normalized confusion matrix')

print(cm\_normalized)

plt.figure()

plot\_confusion\_matrix(cm\_normalized, labels, title = 'Normalized confusion matrix')

plt.show()

In [ ]:

*# Accuracy, Precision, Recall and F1 Score*

ac = metrics.accuracy\_score(y\_test, y\_pred)

precision = metrics.precision\_score(y\_test, y\_pred)

recall = metrics.recall\_score(y\_test, y\_pred)

f1 = metrics.f1\_score(y\_test, y\_pred)

print(f"Accuracy: **{**ac**}**")

print(f"recall: **{**recall**}**")

print(f"precision: **{**precision**}**")

print(f"f1: **{**f1**}**")

##### **ROC AUC**

In [ ]:

classification = pd.DataFrame({'y': y\_test, 'yhat': y\_pred\_proba[:,1]})

THRESHOLD = 0.5 *#Random Threshold Value*

y = np.array(y\_test)

y\_hat = np.array([(1 **if** item >= THRESHOLD **else** 0) **for** item **in** y\_pred\_proba[:,1]])

print(f'y\_test: **{**y\_test**}**')

print(f'y\_pred\_proba: **{**y\_pred\_proba**}**')

print(f'y: **{**y**}**')

print(f'yhat: **{**y\_hat**}**')

In [ ]:

count\_pos = sum(y==1)

count\_neg = sum(y==0)

count = len(y)

print(f'Positive count: **{**count\_pos**}**')

print(f'Negative count: **{**count\_neg**}**')

tp = sum(np.logical\_and(y==1, y\_hat==1))

tp\_rate = float(tp)/count\_pos

tn = sum(np.logical\_and(y==0, y\_hat==0))

tn\_rate = float(tn)/count\_neg

fp = sum(np.logical\_and(y==0, y\_hat==1))

fp\_rate = float(fp)/count\_neg

fn = sum(np.logical\_and(y==1, y\_hat==0))

fn\_rate = float(fn)/count\_pos

print(f'Count: **{**count**}**')

print(f'True Positive (TP, sensativity): **{**tp**}** (**{**int(tp\_rate\*100)**}**%)')

print(f'True Negative (TN, specificity): **{**tn**}** (**{**int(tn\_rate\*100)**}**%)')

print(f'False Positive (FP): **{**fp**}** (**{**int(fp\_rate\*100)**}**%)')

print(f'False Negative (FN): **{**fn**}** (**{**int(fn\_rate\*100)**}**%)')

In [ ]:

ac = metrics.accuracy\_score(y, y\_hat)

precision = metrics.precision\_score(y, y\_hat)

recall = metrics.recall\_score(y, y\_hat)

f1 = metrics.f1\_score(y, y\_hat)

print(f"Accuracy: **{**ac**}**")

print(f"recall: **{**recall**}**")

print(f"precision: **{**precision**}**")

print(f"f1: **{**f1**}**")

In [ ]:

*# Plot a confusion matrix.*

*# cm is the confusion matrix, names are the names of the classes.*

**def** plot\_confusion\_matrix(cm, names, title = 'Confusion matrix', cmap = plt.cm.Blues):

*# plt.imshow displays the image on the axes, but if you need to*

*# display multiple images you use show() to finish the figure.*

*# interpolation = 'none': works well when a big image is scaled down*

*# interpolation = 'nearest': works well when a small image is scaled up*

*# cmap: The registered colormap name used to map scalar data to colors.*

plt.imshow(cm, interpolation='nearest', cmap = cmap)

plt.title(title)

plt.colorbar()

tick\_marks = np.arange(len(names))

plt.xticks(tick\_marks, names, rotation = 45)

plt.yticks(tick\_marks, names)

*# Automatically adjust subplot parameters to give specified padding.*

plt.tight\_layout()

plt.ylabel('True label')

plt.xlabel('Predicted label')

labels = ['T', 'F']

*# Compute confusion matrix*

cm = confusion\_matrix(y, y\_hat)

np.set\_printoptions(precision = 2)

print('Confusion matrix, without normalization')

print(cm)

plt.figure()

plot\_confusion\_matrix(cm, labels)

*# Normalize the confusion matrix by row*

*# (i.e. by the number of samples in each class)*

cm\_normalized = cm.astype('float') / cm.sum(axis = 1)[:, np.newaxis]

print('Normalized confusion matrix')

print(cm\_normalized)

plt.figure()

plot\_confusion\_matrix(cm\_normalized, labels, title = 'Normalized confusion matrix')

plt.show()

In [ ]:

*# Compute ROC curve and ROC area for each class*

*# tp\_rate = float(tp)/count\_pos*

*# fp\_rate = float(fp)/count\_neg*

fpr, tpr, thresholds = roc\_curve(y, y\_pred\_proba[:,1])

*# Compute Area Under the Curve (AUC) using the trapezoidal rule*

roc\_auc = auc(fpr, tpr)

print(f"Y: **{**y**}**")

print(f"Y\_HAT: **{**y\_hat**}**")

print(f"FPR: **{**fpr**}**")

print(f"TPR: **{**tpr**}**")

print(f"thresholds: **{**thresholds**}**")

print (F"Optimal threshold index: **{**np.argmax(tpr - fpr)**}**")

print (F"Optimal threshold value: **{**thresholds[np.argmax(tpr - fpr)]**}**")

print(f"AUC: **{**roc\_auc**}**")

In [ ]:

plt.figure()

lw = 2

plt.plot(fpr, tpr, color = 'darkorange',

lw = lw, label = 'ROC curve (area = **%0.2f**)' % roc\_auc)

plt.plot([0, 1], [0, 1], color = 'navy', lw = lw, linestyle = '--')

plt.xlim([0.0, 1.0])

plt.ylim([0.0, 1.05])

plt.xlabel('False Positive Rate')

plt.ylabel('True Positive Rate')

plt.title('Receiver operating characteristic')

plt.legend(loc = "lower right")

plt.show()

In [ ]:

print(f"FPR: **{**fpr**}**")

print(f"TPR: **{**tpr**}**")

print(f"thresholds: **{**thresholds**}**")

In [ ]:

classification = pd.DataFrame({'y': y\_test, 'yhat': y\_pred\_proba[:,1]})

THRESHOLD = 0.52 *#Optimal Threshold Value*

y = np.array(y\_test)

y\_hat = np.array([(1 **if** item >= THRESHOLD **else** 0) **for** item **in** y\_pred\_proba[:,1]])

print(f'y\_test: **{**y\_test**}**')

print(f'y\_pred\_proba: **{**y\_pred\_proba**}**')

print(f'y: **{**y**}**')

print(f'yhat: **{**y\_hat**}**')

In [ ]:

count\_pos = sum(y==1)

count\_neg = sum(y==0)

count = len(y)

print(f'Positive count: **{**count\_pos**}**')

print(f'Negative count: **{**count\_neg**}**')

tp = sum(np.logical\_and(y==1, y\_hat==1))

tp\_rate = float(tp)/count\_pos

tn = sum(np.logical\_and(y==0, y\_hat==0))

tn\_rate = float(tn)/count\_neg

fp = sum(np.logical\_and(y==0, y\_hat==1))

fp\_rate = float(fp)/count\_neg

fn = sum(np.logical\_and(y==1, y\_hat==0))

fn\_rate = float(fn)/count\_pos

print(f'Count: **{**count**}**')

print(f'True Positive (TP, sensativity): **{**tp**}** (**{**int(tp\_rate\*100)**}**%)')

print(f'True Negative (TN, specificity): **{**tn**}** (**{**int(tn\_rate\*100)**}**%)')

print(f'False Positive (FP): **{**fp**}** (**{**int(fp\_rate\*100)**}**%)')

print(f'False Negative (FN): **{**fn**}** (**{**int(fn\_rate\*100)**}**%)')

In [ ]:

ac = metrics.accuracy\_score(y, y\_hat)

precision = metrics.precision\_score(y, y\_hat)

recall = metrics.recall\_score(y, y\_hat)

f1 = metrics.f1\_score(y, y\_hat)

print(f"Accuracy: **{**ac**}**")

print(f"recall: **{**recall**}**")

print(f"precision: **{**precision**}**")

print(f"f1: **{**f1**}**")

In [ ]:

*# Plot a confusion matrix.*

*# cm is the confusion matrix, names are the names of the classes.*

**def** plot\_confusion\_matrix(cm, names, title = 'Confusion matrix', cmap = plt.cm.Blues):

*# plt.imshow displays the image on the axes, but if you need to*

*# display multiple images you use show() to finish the figure.*

*# interpolation = 'none': works well when a big image is scaled down*

*# interpolation = 'nearest': works well when a small image is scaled up*

*# cmap: The registered colormap name used to map scalar data to colors.*

plt.imshow(cm, interpolation='nearest', cmap = cmap)

plt.title(title)

plt.colorbar()

tick\_marks = np.arange(len(names))

plt.xticks(tick\_marks, names, rotation = 45)

plt.yticks(tick\_marks, names)

*# Automatically adjust subplot parameters to give specified padding.*

plt.tight\_layout()

plt.ylabel('True label')

plt.xlabel('Predicted label')

labels = ['T', 'F']

*# Compute confusion matrix*

cm = confusion\_matrix(y, y\_hat)

np.set\_printoptions(precision = 2)

print('Confusion matrix, without normalization')

print(cm)

plt.figure()

plot\_confusion\_matrix(cm, labels)

*# Normalize the confusion matrix by row*

*# (i.e. by the number of samples in each class)*

cm\_normalized = cm.astype('float') / cm.sum(axis = 1)[:, np.newaxis]

print('Normalized confusion matrix')

print(cm\_normalized)

plt.figure()

plot\_confusion\_matrix(cm\_normalized, labels, title = 'Normalized confusion matrix')

plt.show()

##### **Decile Analysis**

In [ ]:

y = np.array(y\_test)

y\_hat = np.array(y\_pred\_proba[:,1])

In [ ]:

*# Increase size and add a little noise*

np.random.seed(42)

y = np.concatenate([y, y, y, y])

y\_hat = np.concatenate([y\_hat, y\_hat, y\_hat, y\_hat])

y\_hat = y\_hat + np.random.normal(size = len(y\_hat)) / 10

y\_hat = np.clip(y\_hat,0.01,0.99)

print(y\_hat, len(y\_hat))

In [ ]:

data = pd.DataFrame({'y':y,'y\_hat':y\_hat})

data.sort\_values(by='y\_hat',ascending = **False**, inplace = **True**)

data['bucket'] = pd.qcut(range(len(data)), 10, labels = **False**) + 1

data

In [ ]:

data.drop('y\_hat', 1, inplace=**True**)

data['count'] = np.ones(len(data))

data = data.groupby(by='bucket').sum()

data

In [ ]:

data['score'] = data['y'].values / data['count'].values

data.columns = ['tp','count','score']

data

In [ ]:

data.drop('count', 1, inplace=**True**)

data.drop('tp', 1, inplace=**True**)

data.plot(kind = "bar")

##### **Log Loss**

In [ ]:

y = np.array(y\_test)

y\_hat = np.array(y\_pred\_proba[:,1])

In [ ]:

llos = metrics.log\_loss(y, y\_hat)

print(f"Log loss: **{**llos**}**")

#### **Median**

In [ ]:

X\_train, X\_test, y\_train, y\_test = train\_test\_split(meanheart.drop("target", axis = 1), meanheart.target, test\_size=0.2,random\_state=42)

logreg = LogisticRegression(max\_iter=3000) *# set the max iteration to be 3000 otherwise the process can't be finished*

logreg.fit(X\_train,y\_train)

print ("Trained Model:", logreg, "**\n**")

y\_pred = logreg.predict(X\_test)

*# view the model's score, which will indicate how good my model has been trained*

print("Score : ", accuracy\_score(y\_test, y\_pred, normalize = **True**))

In [ ]:

*# we can even look at the probabilities the learner assigned to each class*

y\_pred\_proba = logreg.predict\_proba(X\_test).round(2)

print(y\_pred\_proba, "**\n**")

In [ ]:

*# Plot a confusion matrix.*

*# cm is the confusion matrix, names are the names of the classes.*

**def** plot\_confusion\_matrix(cm, names, title = 'Confusion matrix', cmap = plt.cm.Blues):

*# plt.imshow displays the image on the axes, but if you need to*

*# display multiple images you use show() to finish the figure.*

*# interpolation = 'none': works well when a big image is scaled down*

*# interpolation = 'nearest': works well when a small image is scaled up*

*# cmap: The registered colormap name used to map scalar data to colors.*

plt.imshow(cm, interpolation='nearest', cmap = cmap)

plt.title(title)

plt.colorbar()

tick\_marks = np.arange(len(names))

plt.xticks(tick\_marks, names, rotation = 45)

plt.yticks(tick\_marks, names)

*# Automatically adjust subplot parameters to give specified padding.*

plt.tight\_layout()

plt.ylabel('True label')

plt.xlabel('Predicted label')

labels = ['T', 'F']

*# Compute confusion matrix*

cm = confusion\_matrix(y\_test, y\_pred)

np.set\_printoptions(precision = 2)

print('Confusion matrix, without normalization')

print(cm)

plt.figure()

plot\_confusion\_matrix(cm, labels)

*# Normalize the confusion matrix by row*

*# (i.e. by the number of samples in each class)*

cm\_normalized = cm.astype('float') / cm.sum(axis = 1)[:, np.newaxis]

print('Normalized confusion matrix')

print(cm\_normalized)

plt.figure()

plot\_confusion\_matrix(cm\_normalized, labels, title = 'Normalized confusion matrix')

plt.show()

In [ ]:

*# Accuracy, Precision, Recall and F1 Score*

ac = metrics.accuracy\_score(y\_test, y\_pred)

precision = metrics.precision\_score(y\_test, y\_pred)

recall = metrics.recall\_score(y\_test, y\_pred)

f1 = metrics.f1\_score(y\_test, y\_pred)

print(f"Accuracy: **{**ac**}**")

print(f"recall: **{**recall**}**")

print(f"precision: **{**precision**}**")

print(f"f1: **{**f1**}**")

##### **ROC AUC**

In [ ]:

classification = pd.DataFrame({'y': y\_test, 'yhat': y\_pred\_proba[:,1]})

THRESHOLD = 0.5 *#Random Threshold Value*

y = np.array(y\_test)

y\_hat = np.array([(1 **if** item >= THRESHOLD **else** 0) **for** item **in** y\_pred\_proba[:,1]])

print(f'y\_test: **{**y\_test**}**')

print(f'y\_pred\_proba: **{**y\_pred\_proba**}**')

print(f'y: **{**y**}**')

print(f'yhat: **{**y\_hat**}**')

In [ ]:

count\_pos = sum(y==1)

count\_neg = sum(y==0)

count = len(y)

print(f'Positive count: **{**count\_pos**}**')

print(f'Negative count: **{**count\_neg**}**')

tp = sum(np.logical\_and(y==1, y\_hat==1))

tp\_rate = float(tp)/count\_pos

tn = sum(np.logical\_and(y==0, y\_hat==0))

tn\_rate = float(tn)/count\_neg

fp = sum(np.logical\_and(y==0, y\_hat==1))

fp\_rate = float(fp)/count\_neg

fn = sum(np.logical\_and(y==1, y\_hat==0))

fn\_rate = float(fn)/count\_pos

print(f'Count: **{**count**}**')

print(f'True Positive (TP, sensativity): **{**tp**}** (**{**int(tp\_rate\*100)**}**%)')

print(f'True Negative (TN, specificity): **{**tn**}** (**{**int(tn\_rate\*100)**}**%)')

print(f'False Positive (FP): **{**fp**}** (**{**int(fp\_rate\*100)**}**%)')

print(f'False Negative (FN): **{**fn**}** (**{**int(fn\_rate\*100)**}**%)')

In [ ]:

ac = metrics.accuracy\_score(y, y\_hat)

precision = metrics.precision\_score(y, y\_hat)

recall = metrics.recall\_score(y, y\_hat)

f1 = metrics.f1\_score(y, y\_hat)

print(f"Accuracy: **{**ac**}**")

print(f"recall: **{**recall**}**")

print(f"precision: **{**precision**}**")

print(f"f1: **{**f1**}**")

In [ ]:

*# Plot a confusion matrix.*

*# cm is the confusion matrix, names are the names of the classes.*

**def** plot\_confusion\_matrix(cm, names, title = 'Confusion matrix', cmap = plt.cm.Blues):

*# plt.imshow displays the image on the axes, but if you need to*

*# display multiple images you use show() to finish the figure.*

*# interpolation = 'none': works well when a big image is scaled down*

*# interpolation = 'nearest': works well when a small image is scaled up*

*# cmap: The registered colormap name used to map scalar data to colors.*

plt.imshow(cm, interpolation='nearest', cmap = cmap)

plt.title(title)

plt.colorbar()

tick\_marks = np.arange(len(names))

plt.xticks(tick\_marks, names, rotation = 45)

plt.yticks(tick\_marks, names)

*# Automatically adjust subplot parameters to give specified padding.*

plt.tight\_layout()

plt.ylabel('True label')

plt.xlabel('Predicted label')

labels = ['T', 'F']

*# Compute confusion matrix*

cm = confusion\_matrix(y, y\_hat)

np.set\_printoptions(precision = 2)

print('Confusion matrix, without normalization')

print(cm)

plt.figure()

plot\_confusion\_matrix(cm, labels)

*# Normalize the confusion matrix by row*

*# (i.e. by the number of samples in each class)*

cm\_normalized = cm.astype('float') / cm.sum(axis = 1)[:, np.newaxis]

print('Normalized confusion matrix')

print(cm\_normalized)

plt.figure()

plot\_confusion\_matrix(cm\_normalized, labels, title = 'Normalized confusion matrix')

plt.show()

In [ ]:

*# Compute ROC curve and ROC area for each class*

*# tp\_rate = float(tp)/count\_pos*

*# fp\_rate = float(fp)/count\_neg*

fpr, tpr, thresholds = roc\_curve(y, y\_pred\_proba[:,1])

*# Compute Area Under the Curve (AUC) using the trapezoidal rule*

roc\_auc = auc(fpr, tpr)

print(f"Y: **{**y**}**")

print(f"Y\_HAT: **{**y\_hat**}**")

print(f"FPR: **{**fpr**}**")

print(f"TPR: **{**tpr**}**")

print(f"thresholds: **{**thresholds**}**")

print (F"Optimal threshold index: **{**np.argmax(tpr - fpr)**}**")

print (F"Optimal threshold value: **{**thresholds[np.argmax(tpr - fpr)]**}**")

print(f"AUC: **{**roc\_auc**}**")

In [ ]:

plt.figure()

lw = 2

plt.plot(fpr, tpr, color = 'darkorange',

lw = lw, label = 'ROC curve (area = **%0.2f**)' % roc\_auc)

plt.plot([0, 1], [0, 1], color = 'navy', lw = lw, linestyle = '--')

plt.xlim([0.0, 1.0])

plt.ylim([0.0, 1.05])

plt.xlabel('False Positive Rate')

plt.ylabel('True Positive Rate')

plt.title('Receiver operating characteristic')

plt.legend(loc = "lower right")

plt.show()

In [ ]:

print(f"FPR: **{**fpr**}**")

print(f"TPR: **{**tpr**}**")

print(f"thresholds: **{**thresholds**}**")

In [ ]:

classification = pd.DataFrame({'y': y\_test, 'yhat': y\_pred\_proba[:,1]})

THRESHOLD = 0.77 *#Optimal Threshold Value*

y = np.array(y\_test)

y\_hat = np.array([(1 **if** item >= THRESHOLD **else** 0) **for** item **in** y\_pred\_proba[:,1]])

print(f'y\_test: **{**y\_test**}**')

print(f'y\_pred\_proba: **{**y\_pred\_proba**}**')

print(f'y: **{**y**}**')

print(f'yhat: **{**y\_hat**}**')

In [ ]:

count\_pos = sum(y==1)

count\_neg = sum(y==0)

count = len(y)

print(f'Positive count: **{**count\_pos**}**')

print(f'Negative count: **{**count\_neg**}**')

tp = sum(np.logical\_and(y==1, y\_hat==1))

tp\_rate = float(tp)/count\_pos

tn = sum(np.logical\_and(y==0, y\_hat==0))

tn\_rate = float(tn)/count\_neg

fp = sum(np.logical\_and(y==0, y\_hat==1))

fp\_rate = float(fp)/count\_neg

fn = sum(np.logical\_and(y==1, y\_hat==0))

fn\_rate = float(fn)/count\_pos

print(f'Count: **{**count**}**')

print(f'True Positive (TP, sensativity): **{**tp**}** (**{**int(tp\_rate\*100)**}**%)')

print(f'True Negative (TN, specificity): **{**tn**}** (**{**int(tn\_rate\*100)**}**%)')

print(f'False Positive (FP): **{**fp**}** (**{**int(fp\_rate\*100)**}**%)')

print(f'False Negative (FN): **{**fn**}** (**{**int(fn\_rate\*100)**}**%)')

In [ ]:

ac = metrics.accuracy\_score(y, y\_hat)

precision = metrics.precision\_score(y, y\_hat)

recall = metrics.recall\_score(y, y\_hat)

f1 = metrics.f1\_score(y, y\_hat)

print(f"Accuracy: **{**ac**}**")

print(f"recall: **{**recall**}**")

print(f"precision: **{**precision**}**")

print(f"f1: **{**f1**}**")

In [ ]:

*# Plot a confusion matrix.*

*# cm is the confusion matrix, names are the names of the classes.*

**def** plot\_confusion\_matrix(cm, names, title = 'Confusion matrix', cmap = plt.cm.Blues):

*# plt.imshow displays the image on the axes, but if you need to*

*# display multiple images you use show() to finish the figure.*

*# interpolation = 'none': works well when a big image is scaled down*

*# interpolation = 'nearest': works well when a small image is scaled up*

*# cmap: The registered colormap name used to map scalar data to colors.*

plt.imshow(cm, interpolation='nearest', cmap = cmap)

plt.title(title)

plt.colorbar()

tick\_marks = np.arange(len(names))

plt.xticks(tick\_marks, names, rotation = 45)

plt.yticks(tick\_marks, names)

*# Automatically adjust subplot parameters to give specified padding.*

plt.tight\_layout()

plt.ylabel('True label')

plt.xlabel('Predicted label')

labels = ['T', 'F']

*# Compute confusion matrix*

cm = confusion\_matrix(y, y\_hat)

np.set\_printoptions(precision = 2)

print('Confusion matrix, without normalization')

print(cm)

plt.figure()

plot\_confusion\_matrix(cm, labels)

*# Normalize the confusion matrix by row*

*# (i.e. by the number of samples in each class)*

cm\_normalized = cm.astype('float') / cm.sum(axis = 1)[:, np.newaxis]

print('Normalized confusion matrix')

print(cm\_normalized)

plt.figure()

plot\_confusion\_matrix(cm\_normalized, labels, title = 'Normalized confusion matrix')

plt.show()

##### Decile Analysis

In [ ]:

y = np.array(y\_test)

y\_hat = np.array(y\_pred\_proba[:,1])

In [ ]:

*# Increase size and add a little noise*

np.random.seed(42)

y = np.concatenate([y, y, y, y])

y\_hat = np.concatenate([y\_hat, y\_hat, y\_hat, y\_hat])

y\_hat = y\_hat + np.random.normal(size = len(y\_hat)) / 10

y\_hat = np.clip(y\_hat,0.01,0.99)

print(y\_hat, len(y\_hat))

In [ ]:

data = pd.DataFrame({'y':y,'y\_hat':y\_hat})

data.sort\_values(by='y\_hat',ascending = **False**, inplace = **True**)

data['bucket'] = pd.qcut(range(len(data)), 10, labels = **False**) + 1

data

In [ ]:

data.drop('y\_hat', 1, inplace=**True**)

data['count'] = np.ones(len(data))

data = data.groupby(by='bucket').sum()

data

In [ ]:

data['score'] = data['y'].values / data['count'].values

data.columns = ['tp','count','score']

data

In [ ]:

data.drop('count', 1, inplace=**True**)

data.drop('tp', 1, inplace=**True**)

data.plot(kind = "bar")

##### **Log Loss**

In [ ]:

y = np.array(y\_test)

y\_hat = np.array(y\_pred\_proba[:,1])

In [ ]:

llos = metrics.log\_loss(y, y\_hat)

print(f"Log loss: **{**llos**}**")

### **DECISION TREE**

#### **Mean**

In [ ]:

X\_train, X\_test, y\_train, y\_test = train\_test\_split(meanheart.drop("target", axis = 1), meanheart.target, test\_size = 0.4, random\_state = 10)

dtree = DecisionTreeClassifier(random\_state=17, max\_depth=3, min\_samples\_leaf=2)

dtree.fit(X=X\_train, y=y\_train)

print("trained Model: ", dtree, "**\n**")

*# Apply the learner to the new, unclassified observation.*

y\_pred = dtree.predict(X\_test)

print(y\_pred, "**\n**")

*# view the model's score, which will indicate how good my model has been trained*

print("Score: ", dtree.score(X\_test, y\_test))

In [ ]:

*# we can even look at the probabilities the learner assigned to each class*

y\_pred\_proba = dtree.predict\_proba(X\_test).round(2)

print(y\_pred\_proba, "**\n**")

In [ ]:

*# Plot a confusion matrix.*

*# cm is the confusion matrix, names are the names of the classes.*

**def** plot\_confusion\_matrix(cm, names, title = 'Confusion matrix', cmap = plt.cm.Blues):

*# plt.imshow displays the image on the axes, but if you need to*

*# display multiple images you use show() to finish the figure.*

*# interpolation = 'none': works well when a big image is scaled down*

*# interpolation = 'nearest': works well when a small image is scaled up*

*# cmap: The registered colormap name used to map scalar data to colors.*

plt.imshow(cm, interpolation='nearest', cmap = cmap)

plt.title(title)

plt.colorbar()

tick\_marks = np.arange(len(names))

plt.xticks(tick\_marks, names, rotation = 45)

plt.yticks(tick\_marks, names)

*# Automatically adjust subplot parameters to give specified padding.*

plt.tight\_layout()

plt.ylabel('True label')

plt.xlabel('Predicted label')

labels = ['T', 'F']

*# Compute confusion matrix*

cm = confusion\_matrix(y\_test, y\_pred)

np.set\_printoptions(precision = 2)

print('Confusion matrix, without normalization')

print(cm)

plt.figure()

plot\_confusion\_matrix(cm, labels)

*# Normalize the confusion matrix by row*

*# (i.e. by the number of samples in each class)*

cm\_normalized = cm.astype('float') / cm.sum(axis = 1)[:, np.newaxis]

print('Normalized confusion matrix')

print(cm\_normalized)

plt.figure()

plot\_confusion\_matrix(cm\_normalized, labels, title = 'Normalized confusion matrix')

plt.show()

In [ ]:

*# Accuracy, Precision, Recall and F1 Score*

ac = metrics.accuracy\_score(y\_test, y\_pred)

precision = metrics.precision\_score(y\_test, y\_pred)

recall = metrics.recall\_score(y\_test, y\_pred)

f1 = metrics.f1\_score(y\_test, y\_pred)

print(f"Accuracy: **{**ac**}**")

print(f"recall: **{**recall**}**")

print(f"precision: **{**precision**}**")

print(f"f1: **{**f1**}**")

In [ ]:

**with** open("Dtree\_Mean.txt", "w") **as** f:

f = tree.export\_graphviz(dtree, out\_file=f)

##### **ROC AUC**

In [ ]:

classification = pd.DataFrame({'y': y\_test, 'yhat': y\_pred\_proba[:,1]})

THRESHOLD = 0.5 *#Random Threshold Value*

y = np.array(y\_test)

y\_hat = np.array([(1 **if** item >= THRESHOLD **else** 0) **for** item **in** y\_pred\_proba[:,1]])

print(f'y\_test: **{**y\_test**}**')

print(f'y\_pred\_proba: **{**y\_pred\_proba**}**')

print(f'y: **{**y**}**')

print(f'yhat: **{**y\_hat**}**')

In [ ]:

count\_pos = sum(y==1)

count\_neg = sum(y==0)

count = len(y)

print(f'Positive count: **{**count\_pos**}**')

print(f'Negative count: **{**count\_neg**}**')

tp = sum(np.logical\_and(y==1, y\_hat==1))

tp\_rate = float(tp)/count\_pos

tn = sum(np.logical\_and(y==0, y\_hat==0))

tn\_rate = float(tn)/count\_neg

fp = sum(np.logical\_and(y==0, y\_hat==1))

fp\_rate = float(fp)/count\_neg

fn = sum(np.logical\_and(y==1, y\_hat==0))

fn\_rate = float(fn)/count\_pos

print(f'Count: **{**count**}**')

print(f'True Positive (TP, sensativity): **{**tp**}** (**{**int(tp\_rate\*100)**}**%)')

print(f'True Negative (TN, specificity): **{**tn**}** (**{**int(tn\_rate\*100)**}**%)')

print(f'False Positive (FP): **{**fp**}** (**{**int(fp\_rate\*100)**}**%)')

print(f'False Negative (FN): **{**fn**}** (**{**int(fn\_rate\*100)**}**%)')

In [ ]:

ac = metrics.accuracy\_score(y, y\_hat)

precision = metrics.precision\_score(y, y\_hat)

recall = metrics.recall\_score(y, y\_hat)

f1 = metrics.f1\_score(y, y\_hat)

print(f"Accuracy: **{**ac**}**")

print(f"recall: **{**recall**}**")

print(f"precision: **{**precision**}**")

print(f"f1: **{**f1**}**")

In [ ]:

*# Plot a confusion matrix.*

*# cm is the confusion matrix, names are the names of the classes.*

**def** plot\_confusion\_matrix(cm, names, title = 'Confusion matrix', cmap = plt.cm.Blues):

*# plt.imshow displays the image on the axes, but if you need to*

*# display multiple images you use show() to finish the figure.*

*# interpolation = 'none': works well when a big image is scaled down*

*# interpolation = 'nearest': works well when a small image is scaled up*

*# cmap: The registered colormap name used to map scalar data to colors.*

plt.imshow(cm, interpolation='nearest', cmap = cmap)

plt.title(title)

plt.colorbar()

tick\_marks = np.arange(len(names))

plt.xticks(tick\_marks, names, rotation = 45)

plt.yticks(tick\_marks, names)

*# Automatically adjust subplot parameters to give specified padding.*

plt.tight\_layout()

plt.ylabel('True label')

plt.xlabel('Predicted label')

labels = ['T', 'F']

*# Compute confusion matrix*

cm = confusion\_matrix(y, y\_hat)

np.set\_printoptions(precision = 2)

print('Confusion matrix, without normalization')

print(cm)

plt.figure()

plot\_confusion\_matrix(cm, labels)

*# Normalize the confusion matrix by row*

*# (i.e. by the number of samples in each class)*

cm\_normalized = cm.astype('float') / cm.sum(axis = 1)[:, np.newaxis]

print('Normalized confusion matrix')

print(cm\_normalized)

plt.figure()

plot\_confusion\_matrix(cm\_normalized, labels, title = 'Normalized confusion matrix')

plt.show()

In [ ]:

*# Compute ROC curve and ROC area for each class*

*# tp\_rate = float(tp)/count\_pos*

*# fp\_rate = float(fp)/count\_neg*

fpr, tpr, thresholds = roc\_curve(y, y\_pred\_proba[:,1])

*# Compute Area Under the Curve (AUC) using the trapezoidal rule*

roc\_auc = auc(fpr, tpr)

print(f"Y: **{**y**}**")

print(f"Y\_HAT: **{**y\_hat**}**")

print(f"FPR: **{**fpr**}**")

print(f"TPR: **{**tpr**}**")

print(f"thresholds: **{**thresholds**}**")

print (F"Optimal threshold index: **{**np.argmax(tpr - fpr)**}**")

print (F"Optimal threshold value: **{**thresholds[np.argmax(tpr - fpr)]**}**")

print(f"AUC: **{**roc\_auc**}**")

In [ ]:

plt.figure()

lw = 2

plt.plot(fpr, tpr, color = 'darkorange',

lw = lw, label = 'ROC curve (area = **%0.2f**)' % roc\_auc)

plt.plot([0, 1], [0, 1], color = 'navy', lw = lw, linestyle = '--')

plt.xlim([0.0, 1.0])

plt.ylim([0.0, 1.05])

plt.xlabel('False Positive Rate')

plt.ylabel('True Positive Rate')

plt.title('Receiver operating characteristic')

plt.legend(loc = "lower right")

plt.show()

In [ ]:

print(f"FPR: **{**fpr**}**")

print(f"TPR: **{**tpr**}**")

print(f"thresholds: **{**thresholds**}**")

In [ ]:

classification = pd.DataFrame({'y': y\_test, 'yhat': y\_pred\_proba[:,1]})

THRESHOLD = 0.5 *#Optimal Threshold Value*

y = np.array(y\_test)

y\_hat = np.array([(1 **if** item >= THRESHOLD **else** 0) **for** item **in** y\_pred\_proba[:,1]])

print(f'y\_test: **{**y\_test**}**')

print(f'y\_pred\_proba: **{**y\_pred\_proba**}**')

print(f'y: **{**y**}**')

print(f'yhat: **{**y\_hat**}**')

In [ ]:

count\_pos = sum(y==1)

count\_neg = sum(y==0)

count = len(y)

print(f'Positive count: **{**count\_pos**}**')

print(f'Negative count: **{**count\_neg**}**')

tp = sum(np.logical\_and(y==1, y\_hat==1))

tp\_rate = float(tp)/count\_pos

tn = sum(np.logical\_and(y==0, y\_hat==0))

tn\_rate = float(tn)/count\_neg

fp = sum(np.logical\_and(y==0, y\_hat==1))

fp\_rate = float(fp)/count\_neg

fn = sum(np.logical\_and(y==1, y\_hat==0))

fn\_rate = float(fn)/count\_pos

print(f'Count: **{**count**}**')

print(f'True Positive (TP, sensativity): **{**tp**}** (**{**int(tp\_rate\*100)**}**%)')

print(f'True Negative (TN, specificity): **{**tn**}** (**{**int(tn\_rate\*100)**}**%)')

print(f'False Positive (FP): **{**fp**}** (**{**int(fp\_rate\*100)**}**%)')

print(f'False Negative (FN): **{**fn**}** (**{**int(fn\_rate\*100)**}**%)')

In [ ]:

ac = metrics.accuracy\_score(y, y\_hat)

precision = metrics.precision\_score(y, y\_hat)

recall = metrics.recall\_score(y, y\_hat)

f1 = metrics.f1\_score(y, y\_hat)

print(f"Accuracy: **{**ac**}**")

print(f"recall: **{**recall**}**")

print(f"precision: **{**precision**}**")

print(f"f1: **{**f1**}**")

In [ ]:

*# Plot a confusion matrix.*

*# cm is the confusion matrix, names are the names of the classes.*

**def** plot\_confusion\_matrix(cm, names, title = 'Confusion matrix', cmap = plt.cm.Blues):

*# plt.imshow displays the image on the axes, but if you need to*

*# display multiple images you use show() to finish the figure.*

*# interpolation = 'none': works well when a big image is scaled down*

*# interpolation = 'nearest': works well when a small image is scaled up*

*# cmap: The registered colormap name used to map scalar data to colors.*

plt.imshow(cm, interpolation='nearest', cmap = cmap)

plt.title(title)

plt.colorbar()

tick\_marks = np.arange(len(names))

plt.xticks(tick\_marks, names, rotation = 45)

plt.yticks(tick\_marks, names)

*# Automatically adjust subplot parameters to give specified padding.*

plt.tight\_layout()

plt.ylabel('True label')

plt.xlabel('Predicted label')

labels = ['T', 'F']

*# Compute confusion matrix*

cm = confusion\_matrix(y, y\_hat)

np.set\_printoptions(precision = 2)

print('Confusion matrix, without normalization')

print(cm)

plt.figure()

plot\_confusion\_matrix(cm, labels)

*# Normalize the confusion matrix by row*

*# (i.e. by the number of samples in each class)*

cm\_normalized = cm.astype('float') / cm.sum(axis = 1)[:, np.newaxis]

print('Normalized confusion matrix')

print(cm\_normalized)

plt.figure()

plot\_confusion\_matrix(cm\_normalized, labels, title = 'Normalized confusion matrix')

plt.show()

##### **Decile Analysis**

In [ ]:

y = np.array(y\_test)

y\_hat = np.array(y\_pred\_proba[:,1])

In [ ]:

*# Increase size and add a little noise*

np.random.seed(42)

y = np.concatenate([y, y, y, y])

y\_hat = np.concatenate([y\_hat, y\_hat, y\_hat, y\_hat])

y\_hat = y\_hat + np.random.normal(size = len(y\_hat)) / 10

y\_hat = np.clip(y\_hat,0.01,0.99)

print(y\_hat, len(y\_hat))

In [ ]:

data = pd.DataFrame({'y':y,'y\_hat':y\_hat})

data.sort\_values(by='y\_hat',ascending = **False**, inplace = **True**)

data['bucket'] = pd.qcut(range(len(data)), 10, labels = **False**) + 1

data

In [ ]:

data.drop('y\_hat', 1, inplace=**True**)

data['count'] = np.ones(len(data))

data = data.groupby(by='bucket').sum()

data

In [ ]:

data['score'] = data['y'].values / data['count'].values

data.columns = ['tp','count','score']

data

In [ ]:

data.drop('count', 1, inplace=**True**)

data.drop('tp', 1, inplace=**True**)

data.plot(kind = "bar")

##### **Log Loss**

In [ ]:

y = np.array(y\_test)

y\_hat = np.array(y\_pred\_proba[:,1])

In [ ]:

llos = metrics.log\_loss(y, y\_hat)

print(f"Log loss: **{**llos**}**")

#### **Median**

In [ ]:

X\_train, X\_test, y\_train, y\_test = train\_test\_split(medianheart.drop("target", axis = 1), medianheart.target, test\_size = 0.4, random\_state = 10)

dtree = DecisionTreeClassifier(random\_state=17, max\_depth=3, min\_samples\_leaf=2)

dtree.fit(X=X\_train, y=y\_train)

print("trained Model: ", dtree, "**\n**")

*# Apply the learner to the new, unclassified observation.*

y\_pred = dtree.predict(X\_test)

print(y\_pred, "**\n**")

*# view the model's score, which will indicate how good my model has been trained*

print("Score: ", dtree.score(X\_test, y\_test))

In [ ]:

*# we can even look at the probabilities the learner assigned to each class*

y\_pred\_proba = dtree.predict\_proba(X\_test).round(2)

print(y\_pred\_proba, "**\n**")

In [ ]:

*# Plot a confusion matrix.*

*# cm is the confusion matrix, names are the names of the classes.*

**def** plot\_confusion\_matrix(cm, names, title = 'Confusion matrix', cmap = plt.cm.Blues):

*# plt.imshow displays the image on the axes, but if you need to*

*# display multiple images you use show() to finish the figure.*

*# interpolation = 'none': works well when a big image is scaled down*

*# interpolation = 'nearest': works well when a small image is scaled up*

*# cmap: The registered colormap name used to map scalar data to colors.*

plt.imshow(cm, interpolation='nearest', cmap = cmap)

plt.title(title)

plt.colorbar()

tick\_marks = np.arange(len(names))

plt.xticks(tick\_marks, names, rotation = 45)

plt.yticks(tick\_marks, names)

*# Automatically adjust subplot parameters to give specified padding.*

plt.tight\_layout()

plt.ylabel('True label')

plt.xlabel('Predicted label')

labels = ['T', 'F']

*# Compute confusion matrix*

cm = confusion\_matrix(y\_test, y\_pred)

np.set\_printoptions(precision = 2)

print('Confusion matrix, without normalization')

print(cm)

plt.figure()

plot\_confusion\_matrix(cm, labels)

*# Normalize the confusion matrix by row*

*# (i.e. by the number of samples in each class)*

cm\_normalized = cm.astype('float') / cm.sum(axis = 1)[:, np.newaxis]

print('Normalized confusion matrix')

print(cm\_normalized)

plt.figure()

plot\_confusion\_matrix(cm\_normalized, labels, title = 'Normalized confusion matrix')

plt.show()

In [ ]:

*# Accuracy, Precision, Recall and F1 Score*

ac = metrics.accuracy\_score(y\_test, y\_pred)

precision = metrics.precision\_score(y\_test, y\_pred)

recall = metrics.recall\_score(y\_test, y\_pred)

f1 = metrics.f1\_score(y\_test, y\_pred)

print(f"Accuarcy: **{**ac**}**")

print(f"recall: **{**recall**}**")

print(f"precision: **{**precision**}**")

print(f"f1: **{**f1**}**")

In [ ]:

**with** open("Dtree\_Median.txt", "w") **as** f:

f = tree.export\_graphviz(dtree, out\_file=f)

##### **ROC AUC**

In [ ]:

classification = pd.DataFrame({'y': y\_test, 'yhat': y\_pred\_proba[:,1]})

THRESHOLD = 0.5 *#Random Threshold Value*

y = np.array(y\_test)

y\_hat = np.array([(1 **if** item >= THRESHOLD **else** 0) **for** item **in** y\_pred\_proba[:,1]])

print(f'y\_test: **{**y\_test**}**')

print(f'y\_pred\_proba: **{**y\_pred\_proba**}**')

print(f'y: **{**y**}**')

print(f'yhat: **{**y\_hat**}**')

In [ ]:

count\_pos = sum(y==1)

count\_neg = sum(y==0)

count = len(y)

print(f'Positive count: **{**count\_pos**}**')

print(f'Negative count: **{**count\_neg**}**')

tp = sum(np.logical\_and(y==1, y\_hat==1))

tp\_rate = float(tp)/count\_pos

tn = sum(np.logical\_and(y==0, y\_hat==0))

tn\_rate = float(tn)/count\_neg

fp = sum(np.logical\_and(y==0, y\_hat==1))

fp\_rate = float(fp)/count\_neg

fn = sum(np.logical\_and(y==1, y\_hat==0))

fn\_rate = float(fn)/count\_pos

print(f'Count: **{**count**}**')

print(f'True Positive (TP, sensativity): **{**tp**}** (**{**int(tp\_rate\*100)**}**%)')

print(f'True Negative (TN, specificity): **{**tn**}** (**{**int(tn\_rate\*100)**}**%)')

print(f'False Positive (FP): **{**fp**}** (**{**int(fp\_rate\*100)**}**%)')

print(f'False Negative (FN): **{**fn**}** (**{**int(fn\_rate\*100)**}**%)')

In [ ]:

ac = metrics.accuracy\_score(y, y\_hat)

precision = metrics.precision\_score(y, y\_hat)

recall = metrics.recall\_score(y, y\_hat)

f1 = metrics.f1\_score(y, y\_hat)

print(f"Accuracy: **{**ac**}**")

print(f"recall: **{**recall**}**")

print(f"precision: **{**precision**}**")

print(f"f1: **{**f1**}**")

In [ ]:

*# Plot a confusion matrix.*

*# cm is the confusion matrix, names are the names of the classes.*

**def** plot\_confusion\_matrix(cm, names, title = 'Confusion matrix', cmap = plt.cm.Blues):

*# plt.imshow displays the image on the axes, but if you need to*

*# display multiple images you use show() to finish the figure.*

*# interpolation = 'none': works well when a big image is scaled down*

*# interpolation = 'nearest': works well when a small image is scaled up*

*# cmap: The registered colormap name used to map scalar data to colors.*

plt.imshow(cm, interpolation='nearest', cmap = cmap)

plt.title(title)

plt.colorbar()

tick\_marks = np.arange(len(names))

plt.xticks(tick\_marks, names, rotation = 45)

plt.yticks(tick\_marks, names)

*# Automatically adjust subplot parameters to give specified padding.*

plt.tight\_layout()

plt.ylabel('True label')

plt.xlabel('Predicted label')

labels = ['T', 'F']

*# Compute confusion matrix*

cm = confusion\_matrix(y, y\_hat)

np.set\_printoptions(precision = 2)

print('Confusion matrix, without normalization')

print(cm)

plt.figure()

plot\_confusion\_matrix(cm, labels)

*# Normalize the confusion matrix by row*

*# (i.e. by the number of samples in each class)*

cm\_normalized = cm.astype('float') / cm.sum(axis = 1)[:, np.newaxis]

print('Normalized confusion matrix')

print(cm\_normalized)

plt.figure()

plot\_confusion\_matrix(cm\_normalized, labels, title = 'Normalized confusion matrix')

plt.show()

In [ ]:

*# Compute ROC curve and ROC area for each class*

*# tp\_rate = float(tp)/count\_pos*

*# fp\_rate = float(fp)/count\_neg*

fpr, tpr, thresholds = roc\_curve(y, y\_pred\_proba[:,1])

*# Compute Area Under the Curve (AUC) using the trapezoidal rule*

roc\_auc = auc(fpr, tpr)

print(f"Y: **{**y**}**")

print(f"Y\_HAT: **{**y\_hat**}**")

print(f"FPR: **{**fpr**}**")

print(f"TPR: **{**tpr**}**")

print(f"thresholds: **{**thresholds**}**")

print (F"Optimal threshold index: **{**np.argmax(tpr - fpr)**}**")

print (F"Optimal threshold value: **{**thresholds[np.argmax(tpr - fpr)]**}**")

print(f"AUC: **{**roc\_auc**}**")

In [ ]:

plt.figure()

lw = 2

plt.plot(fpr, tpr, color = 'darkorange',

lw = lw, label = 'ROC curve (area = **%0.2f**)' % roc\_auc)

plt.plot([0, 1], [0, 1], color = 'navy', lw = lw, linestyle = '--')

plt.xlim([0.0, 1.0])

plt.ylim([0.0, 1.05])

plt.xlabel('False Positive Rate')

plt.ylabel('True Positive Rate')

plt.title('Receiver operating characteristic')

plt.legend(loc = "lower right")

plt.show()

In [ ]:

print(f"FPR: **{**fpr**}**")

print(f"TPR: **{**tpr**}**")

print(f"thresholds: **{**thresholds**}**")

In [ ]:

classification = pd.DataFrame({'y': y\_test, 'yhat': y\_pred\_proba[:,1]})

THRESHOLD = 0.7 *#Optimal Threshold Value*

y = np.array(y\_test)

y\_hat = np.array([(1 **if** item >= THRESHOLD **else** 0) **for** item **in** y\_pred\_proba[:,1]])

print(f'y\_test: **{**y\_test**}**')

print(f'y\_pred\_proba: **{**y\_pred\_proba**}**')

print(f'y: **{**y**}**')

print(f'yhat: **{**y\_hat**}**')

In [ ]:

count\_pos = sum(y==1)

count\_neg = sum(y==0)

count = len(y)

print(f'Positive count: **{**count\_pos**}**')

print(f'Negative count: **{**count\_neg**}**')

tp = sum(np.logical\_and(y==1, y\_hat==1))

tp\_rate = float(tp)/count\_pos

tn = sum(np.logical\_and(y==0, y\_hat==0))

tn\_rate = float(tn)/count\_neg

fp = sum(np.logical\_and(y==0, y\_hat==1))

fp\_rate = float(fp)/count\_neg

fn = sum(np.logical\_and(y==1, y\_hat==0))

fn\_rate = float(fn)/count\_pos

print(f'Count: **{**count**}**')

print(f'True Positive (TP, sensativity): **{**tp**}** (**{**int(tp\_rate\*100)**}**%)')

print(f'True Negative (TN, specificity): **{**tn**}** (**{**int(tn\_rate\*100)**}**%)')

print(f'False Positive (FP): **{**fp**}** (**{**int(fp\_rate\*100)**}**%)')

print(f'False Negative (FN): **{**fn**}** (**{**int(fn\_rate\*100)**}**%)')

In [ ]:

ac = metrics.accuracy\_score(y, y\_hat)

precision = metrics.precision\_score(y, y\_hat)

recall = metrics.recall\_score(y, y\_hat)

f1 = metrics.f1\_score(y, y\_hat)

print(f"Accuracy: **{**ac**}**")

print(f"recall: **{**recall**}**")

print(f"precision: **{**precision**}**")

print(f"f1: **{**f1**}**")

In [ ]:

*# Plot a confusion matrix.*

*# cm is the confusion matrix, names are the names of the classes.*

**def** plot\_confusion\_matrix(cm, names, title = 'Confusion matrix', cmap = plt.cm.Blues):

*# plt.imshow displays the image on the axes, but if you need to*

*# display multiple images you use show() to finish the figure.*

*# interpolation = 'none': works well when a big image is scaled down*

*# interpolation = 'nearest': works well when a small image is scaled up*

*# cmap: The registered colormap name used to map scalar data to colors.*

plt.imshow(cm, interpolation='nearest', cmap = cmap)

plt.title(title)

plt.colorbar()

tick\_marks = np.arange(len(names))

plt.xticks(tick\_marks, names, rotation = 45)

plt.yticks(tick\_marks, names)

*# Automatically adjust subplot parameters to give specified padding.*

plt.tight\_layout()

plt.ylabel('True label')

plt.xlabel('Predicted label')

labels = ['T', 'F']

*# Compute confusion matrix*

cm = confusion\_matrix(y, y\_hat)

np.set\_printoptions(precision = 2)

print('Confusion matrix, without normalization')

print(cm)

plt.figure()

plot\_confusion\_matrix(cm, labels)

*# Normalize the confusion matrix by row*

*# (i.e. by the number of samples in each class)*

cm\_normalized = cm.astype('float') / cm.sum(axis = 1)[:, np.newaxis]

print('Normalized confusion matrix')

print(cm\_normalized)

plt.figure()

plot\_confusion\_matrix(cm\_normalized, labels, title = 'Normalized confusion matrix')

plt.show()

##### **Decile Analysis**

In [ ]:

y = np.array(y\_test)

y\_hat = np.array(y\_pred\_proba[:,1])

In [ ]:

*# Increase size and add a little noise*

np.random.seed(42)

y = np.concatenate([y, y, y, y])

y\_hat = np.concatenate([y\_hat, y\_hat, y\_hat, y\_hat])

y\_hat = y\_hat + np.random.normal(size = len(y\_hat)) / 10

y\_hat = np.clip(y\_hat,0.01,0.99)

print(y\_hat, len(y\_hat))

In [ ]:

data = pd.DataFrame({'y':y,'y\_hat':y\_hat})

data.sort\_values(by='y\_hat',ascending = **False**, inplace = **True**)

data['bucket'] = pd.qcut(range(len(data)), 10, labels = **False**) + 1

data

In [ ]:

data.drop('y\_hat', 1, inplace=**True**)

data['count'] = np.ones(len(data))

data = data.groupby(by='bucket').sum()

data

In [ ]:

data['score'] = data['y'].values / data['count'].values

data.columns = ['tp','count','score']

data

In [ ]:

data.drop('count', 1, inplace=**True**)

data.drop('tp', 1, inplace=**True**)

data.plot(kind = "bar")

##### **Log Loss**

In [ ]:

y = np.array(y\_test)

y\_hat = np.array(y\_pred\_proba[:,1])

In [ ]:

llos = metrics.log\_loss(y, y\_hat)

print(f"Log loss: **{**llos**}**")

### **NAIVE BAYES**

#### **Mean**

In [ ]:

X\_train, X\_test, y\_train, y\_test = train\_test\_split(meanheart.drop("target", axis = 1), meanheart.target, test\_size = 0.4, random\_state = 10)

clf = GaussianNB()

trained = clf.fit(X\_train, y\_train)

print ("Trained Model:", trained, "**\n**")

*# Apply the learner to the new, unclassified observation.*

y\_pred = clf.predict(X\_test)

print(y\_pred, "**\n**")

*# view the model's score, which will indicate how good my model has been trained*

print("Score : ", accuracy\_score(y\_test, y\_pred, normalize = **True**))

In [ ]:

*# we can even look at the probabilities the learner assigned to each class*

y\_pred\_proba = trained.predict\_proba(X\_test).round(2)

print(y\_pred\_proba, "**\n**")

In [ ]:

*# Plot a confusion matrix.*

*# cm is the confusion matrix, names are the names of the classes.*

**def** plot\_confusion\_matrix(cm, names, title = 'Confusion matrix', cmap = plt.cm.Blues):

*# plt.imshow displays the image on the axes, but if you need to*

*# display multiple images you use show() to finish the figure.*

*# interpolation = 'none': works well when a big image is scaled down*

*# interpolation = 'nearest': works well when a small image is scaled up*

*# cmap: The registered colormap name used to map scalar data to colors.*

plt.imshow(cm, interpolation='nearest', cmap = cmap)

plt.title(title)

plt.colorbar()

tick\_marks = np.arange(len(names))

plt.xticks(tick\_marks, names, rotation = 45)

plt.yticks(tick\_marks, names)

*# Automatically adjust subplot parameters to give specified padding.*

plt.tight\_layout()

plt.ylabel('True label')

plt.xlabel('Predicted label')

labels = ['T', 'F']

*# Compute confusion matrix*

cm = confusion\_matrix(y\_test, y\_pred)

np.set\_printoptions(precision = 2)

print('Confusion matrix, without normalization')

print(cm)

plt.figure()

plot\_confusion\_matrix(cm, labels)

*# Normalize the confusion matrix by row*

*# (i.e. by the number of samples in each class)*

cm\_normalized = cm.astype('float') / cm.sum(axis = 1)[:, np.newaxis]

print('Normalized confusion matrix')

print(cm\_normalized)

plt.figure()

plot\_confusion\_matrix(cm\_normalized, labels, title = 'Normalized confusion matrix')

plt.show()

In [ ]:

*# Accuracy, Precision, Recall and F1 Score*

ac = metrics.accuracy\_score(y\_test, y\_pred)

precision = metrics.precision\_score(y\_test, y\_pred)

recall = metrics.recall\_score(y\_test, y\_pred)

f1 = metrics.f1\_score(y\_test, y\_pred)

print(f"Accuracy: **{**ac**}**")

print(f"recall: **{**recall**}**")

print(f"precision: **{**precision**}**")

print(f"f1: **{**f1**}**")

##### **ROC AUC**

In [ ]:

classification = pd.DataFrame({'y': y\_test, 'yhat': y\_pred\_proba[:,1]})

THRESHOLD = 0.5 *#Random Threshold Value*

y = np.array(y\_test)

y\_hat = np.array([(1 **if** item >= THRESHOLD **else** 0) **for** item **in** y\_pred\_proba[:,1]])

print(f'y\_test: **{**y\_test**}**')

print(f'y\_pred\_proba: **{**y\_pred\_proba**}**')

print(f'y: **{**y**}**')

print(f'yhat: **{**y\_hat**}**')

In [ ]:

count\_pos = sum(y==1)

count\_neg = sum(y==0)

count = len(y)

print(f'Positive count: **{**count\_pos**}**')

print(f'Negative count: **{**count\_neg**}**')

tp = sum(np.logical\_and(y==1, y\_hat==1))

tp\_rate = float(tp)/count\_pos

tn = sum(np.logical\_and(y==0, y\_hat==0))

tn\_rate = float(tn)/count\_neg

fp = sum(np.logical\_and(y==0, y\_hat==1))

fp\_rate = float(fp)/count\_neg

fn = sum(np.logical\_and(y==1, y\_hat==0))

fn\_rate = float(fn)/count\_pos

print(f'Count: **{**count**}**')

print(f'True Positive (TP, sensativity): **{**tp**}** (**{**int(tp\_rate\*100)**}**%)')

print(f'True Negative (TN, specificity): **{**tn**}** (**{**int(tn\_rate\*100)**}**%)')

print(f'False Positive (FP): **{**fp**}** (**{**int(fp\_rate\*100)**}**%)')

print(f'False Negative (FN): **{**fn**}** (**{**int(fn\_rate\*100)**}**%)')

In [ ]:

ac = metrics.accuracy\_score(y, y\_hat)

precision = metrics.precision\_score(y, y\_hat)

recall = metrics.recall\_score(y, y\_hat)

f1 = metrics.f1\_score(y, y\_hat)

print(f"Accuracy: **{**ac**}**")

print(f"recall: **{**recall**}**")

print(f"precision: **{**precision**}**")

print(f"f1: **{**f1**}**")

In [ ]:

*# Plot a confusion matrix.*

*# cm is the confusion matrix, names are the names of the classes.*

**def** plot\_confusion\_matrix(cm, names, title = 'Confusion matrix', cmap = plt.cm.Blues):

*# plt.imshow displays the image on the axes, but if you need to*

*# display multiple images you use show() to finish the figure.*

*# interpolation = 'none': works well when a big image is scaled down*

*# interpolation = 'nearest': works well when a small image is scaled up*

*# cmap: The registered colormap name used to map scalar data to colors.*

plt.imshow(cm, interpolation='nearest', cmap = cmap)

plt.title(title)

plt.colorbar()

tick\_marks = np.arange(len(names))

plt.xticks(tick\_marks, names, rotation = 45)

plt.yticks(tick\_marks, names)

*# Automatically adjust subplot parameters to give specified padding.*

plt.tight\_layout()

plt.ylabel('True label')

plt.xlabel('Predicted label')

labels = ['T', 'F']

*# Compute confusion matrix*

cm = confusion\_matrix(y, y\_hat)

np.set\_printoptions(precision = 2)

print('Confusion matrix, without normalization')

print(cm)

plt.figure()

plot\_confusion\_matrix(cm, labels)

*# Normalize the confusion matrix by row*

*# (i.e. by the number of samples in each class)*

cm\_normalized = cm.astype('float') / cm.sum(axis = 1)[:, np.newaxis]

print('Normalized confusion matrix')

print(cm\_normalized)

plt.figure()

plot\_confusion\_matrix(cm\_normalized, labels, title = 'Normalized confusion matrix')

plt.show()

In [ ]:

*# Compute ROC curve and ROC area for each class*

*# tp\_rate = float(tp)/count\_pos*

*# fp\_rate = float(fp)/count\_neg*

fpr, tpr, thresholds = roc\_curve(y, y\_pred\_proba[:,1])

*# Compute Area Under the Curve (AUC) using the trapezoidal rule*

roc\_auc = auc(fpr, tpr)

print(f"Y: **{**y**}**")

print(f"Y\_HAT: **{**y\_hat**}**")

print(f"FPR: **{**fpr**}**")

print(f"TPR: **{**tpr**}**")

print(f"thresholds: **{**thresholds**}**")

print (F"Optimal threshold index: **{**np.argmax(tpr - fpr)**}**")

print (F"Optimal threshold value: **{**thresholds[np.argmax(tpr - fpr)]**}**")

print(f"AUC: **{**roc\_auc**}**")

In [ ]:

plt.figure()

lw = 2

plt.plot(fpr, tpr, color = 'darkorange',

lw = lw, label = 'ROC curve (area = **%0.2f**)' % roc\_auc)

plt.plot([0, 1], [0, 1], color = 'navy', lw = lw, linestyle = '--')

plt.xlim([0.0, 1.0])

plt.ylim([0.0, 1.05])

plt.xlabel('False Positive Rate')

plt.ylabel('True Positive Rate')

plt.title('Receiver operating characteristic')

plt.legend(loc = "lower right")

plt.show()

In [ ]:

print(f"FPR: **{**fpr**}**")

print(f"TPR: **{**tpr**}**")

print(f"thresholds: **{**thresholds**}**")

In [ ]:

print(y\_pred\_proba)

In [ ]:

classification = pd.DataFrame({'y': y\_test, 'yhat': y\_pred\_proba[:,1]})

THRESHOLD = 0.84 *#Optimal Threshold Value*

y = np.array(y\_test)

y\_hat = np.array([(1 **if** item >= THRESHOLD **else** 0) **for** item **in** y\_pred\_proba[:,1]])

print(f'y\_test: **{**y\_test**}**')

print(f'y\_pred\_proba: **{**y\_pred\_proba**}**')

print(f'y: **{**y**}**')

print(f'yhat: **{**y\_hat**}**')

In [ ]:

count\_pos = sum(y==1)

count\_neg = sum(y==0)

count = len(y)

print(f'Positive count: **{**count\_pos**}**')

print(f'Negative count: **{**count\_neg**}**')

tp = sum(np.logical\_and(y==1, y\_hat==1))

tp\_rate = float(tp)/count\_pos

tn = sum(np.logical\_and(y==0, y\_hat==0))

tn\_rate = float(tn)/count\_neg

fp = sum(np.logical\_and(y==0, y\_hat==1))

fp\_rate = float(fp)/count\_neg

fn = sum(np.logical\_and(y==1, y\_hat==0))

fn\_rate = float(fn)/count\_pos

print(f'Count: **{**count**}**')

print(f'True Positive (TP, sensativity): **{**tp**}** (**{**int(tp\_rate\*100)**}**%)')

print(f'True Negative (TN, specificity): **{**tn**}** (**{**int(tn\_rate\*100)**}**%)')

print(f'False Positive (FP): **{**fp**}** (**{**int(fp\_rate\*100)**}**%)')

print(f'False Negative (FN): **{**fn**}** (**{**int(fn\_rate\*100)**}**%)')

In [ ]:

ac = metrics.accuracy\_score(y, y\_hat)

precision = metrics.precision\_score(y, y\_hat)

recall = metrics.recall\_score(y, y\_hat)

f1 = metrics.f1\_score(y, y\_hat)

print(f"Accuracy: **{**ac**}**")

print(f"recall: **{**recall**}**")

print(f"precision: **{**precision**}**")

print(f"f1: **{**f1**}**")

In [ ]:

*# Plot a confusion matrix.*

*# cm is the confusion matrix, names are the names of the classes.*

**def** plot\_confusion\_matrix(cm, names, title = 'Confusion matrix', cmap = plt.cm.Blues):

*# plt.imshow displays the image on the axes, but if you need to*

*# display multiple images you use show() to finish the figure.*

*# interpolation = 'none': works well when a big image is scaled down*

*# interpolation = 'nearest': works well when a small image is scaled up*

*# cmap: The registered colormap name used to map scalar data to colors.*

plt.imshow(cm, interpolation='nearest', cmap = cmap)

plt.title(title)

plt.colorbar()

tick\_marks = np.arange(len(names))

plt.xticks(tick\_marks, names, rotation = 45)

plt.yticks(tick\_marks, names)

*# Automatically adjust subplot parameters to give specified padding.*

plt.tight\_layout()

plt.ylabel('True label')

plt.xlabel('Predicted label')

labels = ['T', 'F']

*# Compute confusion matrix*

cm = confusion\_matrix(y, y\_hat)

np.set\_printoptions(precision = 2)

print('Confusion matrix, without normalization')

print(cm)

plt.figure()

plot\_confusion\_matrix(cm, labels)

*# Normalize the confusion matrix by row*

*# (i.e. by the number of samples in each class)*

cm\_normalized = cm.astype('float') / cm.sum(axis = 1)[:, np.newaxis]

print('Normalized confusion matrix')

print(cm\_normalized)

plt.figure()

plot\_confusion\_matrix(cm\_normalized, labels, title = 'Normalized confusion matrix')

plt.show()

##### **Decile Analysis**

In [ ]:

y = np.array(y\_test)

y\_hat = np.array(y\_pred\_proba[:,1])

In [ ]:

*# Increase size and add a little noise*

np.random.seed(42)

y = np.concatenate([y, y, y, y])

y\_hat = np.concatenate([y\_hat, y\_hat, y\_hat, y\_hat])

y\_hat = y\_hat + np.random.normal(size = len(y\_hat)) / 10

y\_hat = np.clip(y\_hat,0.01,0.99)

print(y\_hat, len(y\_hat))

In [ ]:

data = pd.DataFrame({'y':y,'y\_hat':y\_hat})

data.sort\_values(by='y\_hat',ascending = **False**, inplace = **True**)

data['bucket'] = pd.qcut(range(len(data)), 10, labels = **False**) + 1

data

In [ ]:

data.drop('y\_hat', 1, inplace=**True**)

data['count'] = np.ones(len(data))

data = data.groupby(by='bucket').sum()

data

In [ ]:

data['score'] = data['y'].values / data['count'].values

data.columns = ['tp','count','score']

data

In [ ]:

data.drop('count', 1, inplace=**True**)

data.drop('tp', 1, inplace=**True**)

data.plot(kind = "bar")

##### **Log Loss**

In [ ]:

y = np.array(y\_test)

y\_hat = np.array(y\_pred\_proba[:,1])

In [ ]:

llos = metrics.log\_loss(y, y\_hat)

print(f"Log loss: **{**llos**}**")

#### **Median**

In [ ]:

X\_train, X\_test, y\_train, y\_test = train\_test\_split(medianheart.drop("target", axis = 1), medianheart.target, test\_size = 0.4, random\_state = 10)

clf = GaussianNB()

trained = clf.fit(X\_train, y\_train)

print ("Trained Model:", trained, "**\n**")

*# Apply the learner to the new, unclassified observation.*

y\_pred = clf.predict(X\_test)

print(y\_pred, "**\n**")

*# view the model's score, which will indicate how good my model has been trained*

print("Score : ", accuracy\_score(y\_test, y\_pred, normalize = **True**))

In [ ]:

*# we can even look at the probabilities the learner assigned to each class*

y\_pred\_proba = trained.predict\_proba(X\_test).round(2)

print(y\_pred\_proba, "**\n**")

In [ ]:

*# Plot a confusion matrix.*

*# cm is the confusion matrix, names are the names of the classes.*

**def** plot\_confusion\_matrix(cm, names, title = 'Confusion matrix', cmap = plt.cm.Blues):

*# plt.imshow displays the image on the axes, but if you need to*

*# display multiple images you use show() to finish the figure.*

*# interpolation = 'none': works well when a big image is scaled down*

*# interpolation = 'nearest': works well when a small image is scaled up*

*# cmap: The registered colormap name used to map scalar data to colors.*

plt.imshow(cm, interpolation='nearest', cmap = cmap)

plt.title(title)

plt.colorbar()

tick\_marks = np.arange(len(names))

plt.xticks(tick\_marks, names, rotation = 45)

plt.yticks(tick\_marks, names)

*# Automatically adjust subplot parameters to give specified padding.*

plt.tight\_layout()

plt.ylabel('True label')

plt.xlabel('Predicted label')

labels = ['T', 'F']

*# Compute confusion matrix*

cm = confusion\_matrix(y\_test, y\_pred)

np.set\_printoptions(precision = 2)

print('Confusion matrix, without normalization')

print(cm)

plt.figure()

plot\_confusion\_matrix(cm, labels)

*# Normalize the confusion matrix by row*

*# (i.e. by the number of samples in each class)*

cm\_normalized = cm.astype('float') / cm.sum(axis = 1)[:, np.newaxis]

print('Normalized confusion matrix')

print(cm\_normalized)

plt.figure()

plot\_confusion\_matrix(cm\_normalized, labels, title = 'Normalized confusion matrix')

plt.show()

In [ ]:

*# Accuracy, Precision, Recall and F1 Score*

ac = metrics.accuracy\_score(y\_test, y\_pred)

precision = metrics.precision\_score(y\_test, y\_pred)

recall = metrics.recall\_score(y\_test, y\_pred)

f1 = metrics.f1\_score(y\_test, y\_pred)

print(f"Accuracy: **{**ac**}**")

print(f"recall: **{**recall**}**")

print(f"precision: **{**precision**}**")

print(f"f1: **{**f1**}**")

##### **ROC AUC**

In [ ]:

classification = pd.DataFrame({'y': y\_test, 'yhat': y\_pred\_proba[:,1]})

THRESHOLD = 0.5 *#Random Threshold Value*

y = np.array(y\_test)

y\_hat = np.array([(1 **if** item >= THRESHOLD **else** 0) **for** item **in** y\_pred\_proba[:,1]])

print(f'y\_test: **{**y\_test**}**')

print(f'y\_pred\_proba: **{**y\_pred\_proba**}**')

print(f'y: **{**y**}**')

print(f'yhat: **{**y\_hat**}**')

In [ ]:

count\_pos = sum(y==1)

count\_neg = sum(y==0)

count = len(y)

print(f'Positive count: **{**count\_pos**}**')

print(f'Negative count: **{**count\_neg**}**')

tp = sum(np.logical\_and(y==1, y\_hat==1))

tp\_rate = float(tp)/count\_pos

tn = sum(np.logical\_and(y==0, y\_hat==0))

tn\_rate = float(tn)/count\_neg

fp = sum(np.logical\_and(y==0, y\_hat==1))

fp\_rate = float(fp)/count\_neg

fn = sum(np.logical\_and(y==1, y\_hat==0))

fn\_rate = float(fn)/count\_pos

print(f'Count: **{**count**}**')

print(f'True Positive (TP, sensativity): **{**tp**}** (**{**int(tp\_rate\*100)**}**%)')

print(f'True Negative (TN, specificity): **{**tn**}** (**{**int(tn\_rate\*100)**}**%)')

print(f'False Positive (FP): **{**fp**}** (**{**int(fp\_rate\*100)**}**%)')

print(f'False Negative (FN): **{**fn**}** (**{**int(fn\_rate\*100)**}**%)')

In [ ]:

ac = metrics.accuracy\_score(y, y\_hat)

precision = metrics.precision\_score(y, y\_hat)

recall = metrics.recall\_score(y, y\_hat)

f1 = metrics.f1\_score(y, y\_hat)

print(f"Accuracy: **{**ac**}**")

print(f"recall: **{**recall**}**")

print(f"precision: **{**precision**}**")

print(f"f1: **{**f1**}**")

In [ ]:

*# Plot a confusion matrix.*

*# cm is the confusion matrix, names are the names of the classes.*

**def** plot\_confusion\_matrix(cm, names, title = 'Confusion matrix', cmap = plt.cm.Blues):

*# plt.imshow displays the image on the axes, but if you need to*

*# display multiple images you use show() to finish the figure.*

*# interpolation = 'none': works well when a big image is scaled down*

*# interpolation = 'nearest': works well when a small image is scaled up*

*# cmap: The registered colormap name used to map scalar data to colors.*

plt.imshow(cm, interpolation='nearest', cmap = cmap)

plt.title(title)

plt.colorbar()

tick\_marks = np.arange(len(names))

plt.xticks(tick\_marks, names, rotation = 45)

plt.yticks(tick\_marks, names)

*# Automatically adjust subplot parameters to give specified padding.*

plt.tight\_layout()

plt.ylabel('True label')

plt.xlabel('Predicted label')

labels = ['T', 'F']

*# Compute confusion matrix*

cm = confusion\_matrix(y, y\_hat)

np.set\_printoptions(precision = 2)

print('Confusion matrix, without normalization')

print(cm)

plt.figure()

plot\_confusion\_matrix(cm, labels)

*# Normalize the confusion matrix by row*

*# (i.e. by the number of samples in each class)*

cm\_normalized = cm.astype('float') / cm.sum(axis = 1)[:, np.newaxis]

print('Normalized confusion matrix')

print(cm\_normalized)

plt.figure()

plot\_confusion\_matrix(cm\_normalized, labels, title = 'Normalized confusion matrix')

plt.show()

In [ ]:

*# Compute ROC curve and ROC area for each class*

*# tp\_rate = float(tp)/count\_pos*

*# fp\_rate = float(fp)/count\_neg*

fpr, tpr, thresholds = roc\_curve(y, y\_pred\_proba[:,1])

*# Compute Area Under the Curve (AUC) using the trapezoidal rule*

roc\_auc = auc(fpr, tpr)

print(f"Y: **{**y**}**")

print(f"Y\_HAT: **{**y\_hat**}**")

print(f"FPR: **{**fpr**}**")

print(f"TPR: **{**tpr**}**")

print(f"thresholds: **{**thresholds**}**")

print (F"Optimal threshold index: **{**np.argmax(tpr - fpr)**}**")

print (F"Optimal threshold value: **{**thresholds[np.argmax(tpr - fpr)]**}**")

print(f"AUC: **{**roc\_auc**}**")

In [ ]:

plt.figure()

lw = 2

plt.plot(fpr, tpr, color = 'darkorange',

lw = lw, label = 'ROC curve (area = **%0.2f**)' % roc\_auc)

plt.plot([0, 1], [0, 1], color = 'navy', lw = lw, linestyle = '--')

plt.xlim([0.0, 1.0])

plt.ylim([0.0, 1.05])

plt.xlabel('False Positive Rate')

plt.ylabel('True Positive Rate')

plt.title('Receiver operating characteristic')

plt.legend(loc = "lower right")

plt.show()

In [ ]:

print(f"FPR: **{**fpr**}**")

print(f"TPR: **{**tpr**}**")

print(f"thresholds: **{**thresholds**}**")

In [ ]:

classification = pd.DataFrame({'y': y\_test, 'yhat': y\_pred\_proba[:,1]})

THRESHOLD = 0.82 *#Optimal Threshold Value*

y = np.array(y\_test)

y\_hat = np.array([(1 **if** item >= THRESHOLD **else** 0) **for** item **in** y\_pred\_proba[:,1]])

print(f'y\_test: **{**y\_test**}**')

print(f'y\_pred\_proba: **{**y\_pred\_proba**}**')

print(f'y: **{**y**}**')

print(f'yhat: **{**y\_hat**}**')

In [ ]:

count\_pos = sum(y==1)

count\_neg = sum(y==0)

count = len(y)

print(f'Positive count: **{**count\_pos**}**')

print(f'Negative count: **{**count\_neg**}**')

tp = sum(np.logical\_and(y==1, y\_hat==1))

tp\_rate = float(tp)/count\_pos

tn = sum(np.logical\_and(y==0, y\_hat==0))

tn\_rate = float(tn)/count\_neg

fp = sum(np.logical\_and(y==0, y\_hat==1))

fp\_rate = float(fp)/count\_neg

fn = sum(np.logical\_and(y==1, y\_hat==0))

fn\_rate = float(fn)/count\_pos

print(f'Count: **{**count**}**')

print(f'True Positive (TP, sensativity): **{**tp**}** (**{**int(tp\_rate\*100)**}**%)')

print(f'True Negative (TN, specificity): **{**tn**}** (**{**int(tn\_rate\*100)**}**%)')

print(f'False Positive (FP): **{**fp**}** (**{**int(fp\_rate\*100)**}**%)')

print(f'False Negative (FN): **{**fn**}** (**{**int(fn\_rate\*100)**}**%)')

In [ ]:

ac = metrics.accuracy\_score(y, y\_hat)

precision = metrics.precision\_score(y, y\_hat)

recall = metrics.recall\_score(y, y\_hat)

f1 = metrics.f1\_score(y, y\_hat)

print(f"Accuracy: **{**ac**}**")

print(f"recall: **{**recall**}**")

print(f"precision: **{**precision**}**")

print(f"f1: **{**f1**}**")

In [ ]:

*# Plot a confusion matrix.*

*# cm is the confusion matrix, names are the names of the classes.*

**def** plot\_confusion\_matrix(cm, names, title = 'Confusion matrix', cmap = plt.cm.Blues):

*# plt.imshow displays the image on the axes, but if you need to*

*# display multiple images you use show() to finish the figure.*

*# interpolation = 'none': works well when a big image is scaled down*

*# interpolation = 'nearest': works well when a small image is scaled up*

*# cmap: The registered colormap name used to map scalar data to colors.*

plt.imshow(cm, interpolation='nearest', cmap = cmap)

plt.title(title)

plt.colorbar()

tick\_marks = np.arange(len(names))

plt.xticks(tick\_marks, names, rotation = 45)

plt.yticks(tick\_marks, names)

*# Automatically adjust subplot parameters to give specified padding.*

plt.tight\_layout()

plt.ylabel('True label')

plt.xlabel('Predicted label')

labels = ['T', 'F']

*# Compute confusion matrix*

cm = confusion\_matrix(y, y\_hat)

np.set\_printoptions(precision = 2)

print('Confusion matrix, without normalization')

print(cm)

plt.figure()

plot\_confusion\_matrix(cm, labels)

*# Normalize the confusion matrix by row*

*# (i.e. by the number of samples in each class)*

cm\_normalized = cm.astype('float') / cm.sum(axis = 1)[:, np.newaxis]

print('Normalized confusion matrix')

print(cm\_normalized)

plt.figure()

plot\_confusion\_matrix(cm\_normalized, labels, title = 'Normalized confusion matrix')

plt.show()

##### **Decile Analysis**

In [ ]:

y = np.array(y\_test)

y\_hat = np.array(y\_pred\_proba[:,1])

In [ ]:

*# Increase size and add a little noise*

np.random.seed(42)

y = np.concatenate([y, y, y, y])

y\_hat = np.concatenate([y\_hat, y\_hat, y\_hat, y\_hat])

y\_hat = y\_hat + np.random.normal(size = len(y\_hat)) / 10

y\_hat = np.clip(y\_hat,0.01,0.99)

print(y\_hat, len(y\_hat))

In [ ]:

data = pd.DataFrame({'y':y,'y\_hat':y\_hat})

data.sort\_values(by='y\_hat',ascending = **False**, inplace = **True**)

data['bucket'] = pd.qcut(range(len(data)), 10, labels = **False**) + 1

data

In [ ]:

data.drop('y\_hat', 1, inplace=**True**)

data['count'] = np.ones(len(data))

data = data.groupby(by='bucket').sum()

data

In [ ]:

data['score'] = data['y'].values / data['count'].values

data.columns = ['tp','count','score']

data

In [ ]:

data.drop('count', 1, inplace=**True**)

data.drop('tp', 1, inplace=**True**)

data.plot(kind = "bar")

##### **Log Loss**

In [ ]:

y = np.array(y\_test)

y\_hat = np.array(y\_pred\_proba[:,1])

In [ ]:

llos = metrics.log\_loss(y, y\_hat)

print(f"Log loss: **{**llos**}**")

### **KNN**

#### **Mean**

In [ ]:

X\_train, X\_test, y\_train, y\_test = train\_test\_split(meanheart.drop("target", axis = 1), meanheart.target, test\_size=0.4, random\_state = 495)

*# Convert DataFrame data into np.arrays*

*# The scikit-learn library requires the data be formatted as a numpy array.*

*# Here are doing the reformatting*

X = np.array(X\_train)

print(X, X.shape, "**\n**")

y = np.array(y\_train)

print(y, y.shape, "**\n**")

In [ ]:

error\_rate = []

**for** i **in** range(2,25):

knn = neighbors.KNeighborsClassifier(n\_neighbors=i, weights="uniform")

knn.fit(X\_train,y\_train)

pred\_i = knn.predict(X\_test)

error\_rate.append(np.mean(pred\_i != y\_test))

plt.figure(figsize=(10,6))

plt.plot(range(2,25),error\_rate, marker='o', markersize=5)

plt.title('Error Rate vs. K Value')

plt.xlabel('K')

plt.ylabel('Error Rate')

print("Minimum error: ",min(error\_rate),"at K =", error\_rate.index(min(error\_rate))+2)

In [ ]:

clf = neighbors.KNeighborsClassifier(17, weights='uniform')

trained\_model = clf.fit(X, y)

print ("Trained Model:", trained\_model, "**\n**")

*# view the model's score, which will indicate how good my model has been trained*

print ("Score = ", trained\_model.score(X, y), "**\n**")

*# Apply the learner to the new, unclassified observation.*

y\_pred = trained\_model.predict(X\_test)

print(y\_pred, "**\n**")

*# we can even look at the probabilities the learner assigned to each class*

y\_pred\_proba = trained\_model.predict\_proba(X\_test)

print(y\_pred\_proba, "**\n**")

In [ ]:

*# Plot a confusion matrix.*

*# cm is the confusion matrix, names are the names of the classes.*

**def** plot\_confusion\_matrix(cm, names, title = 'Confusion matrix', cmap = plt.cm.Blues):

*# plt.imshow displays the image on the axes, but if you need to*

*# display multiple images you use show() to finish the figure.*

*# interpolation = 'none': works well when a big image is scaled down*

*# interpolation = 'nearest': works well when a small image is scaled up*

*# cmap: The registered colormap name used to map scalar data to colors.*

plt.imshow(cm, interpolation='nearest', cmap = cmap)

plt.title(title)

plt.colorbar()

tick\_marks = np.arange(len(names))

plt.xticks(tick\_marks, names, rotation = 45)

plt.yticks(tick\_marks, names)

*# Automatically adjust subplot parameters to give specified padding.*

plt.tight\_layout()

plt.ylabel('True label')

plt.xlabel('Predicted label')

labels = ['T', 'F']

*# Compute confusion matrix*

cm = confusion\_matrix(y\_test, y\_pred)

np.set\_printoptions(precision = 2)

print('Confusion matrix, without normalization')

print(cm)

plt.figure()

plot\_confusion\_matrix(cm, labels)

*# Normalize the confusion matrix by row*

*# (i.e. by the number of samples in each class)*

cm\_normalized = cm.astype('float') / cm.sum(axis = 1)[:, np.newaxis]

print('Normalized confusion matrix')

print(cm\_normalized)

plt.figure()

plot\_confusion\_matrix(cm\_normalized, labels, title = 'Normalized confusion matrix')

plt.show()

In [ ]:

*# Accuracy, Precision, Recall and F1 Score*

ac = metrics.accuracy\_score(y\_test, y\_pred)

precision = metrics.precision\_score(y\_test, y\_pred)

recall = metrics.recall\_score(y\_test, y\_pred)

f1 = metrics.f1\_score(y\_test, y\_pred)

print(f"Accuracy: **{**ac**}**")

print(f"recall: **{**recall**}**")

print(f"precision: **{**precision**}**")

print(f"f1: **{**f1**}**")

##### **ROC AUC**

In [ ]:

classification = pd.DataFrame({'y': y\_test, 'yhat': y\_pred\_proba[:,1]})

THRESHOLD = 0.5 *#Random Threshold Value*

y = np.array(y\_test)

y\_hat = np.array([(1 **if** item >= THRESHOLD **else** 0) **for** item **in** y\_pred\_proba[:,1]])

print(f'y\_test: **{**y\_test**}**')

print(f'y\_pred\_proba: **{**y\_pred\_proba**}**')

print(f'y: **{**y**}**')

print(f'yhat: **{**y\_hat**}**')

In [ ]:

count\_pos = sum(y==1)

count\_neg = sum(y==0)

count = len(y)

print(f'Positive count: **{**count\_pos**}**')

print(f'Negative count: **{**count\_neg**}**')

tp = sum(np.logical\_and(y==1, y\_hat==1))

tp\_rate = float(tp)/count\_pos

tn = sum(np.logical\_and(y==0, y\_hat==0))

tn\_rate = float(tn)/count\_neg

fp = sum(np.logical\_and(y==0, y\_hat==1))

fp\_rate = float(fp)/count\_neg

fn = sum(np.logical\_and(y==1, y\_hat==0))

fn\_rate = float(fn)/count\_pos

print(f'Count: **{**count**}**')

print(f'True Positive (TP, sensativity): **{**tp**}** (**{**int(tp\_rate\*100)**}**%)')

print(f'True Negative (TN, specificity): **{**tn**}** (**{**int(tn\_rate\*100)**}**%)')

print(f'False Positive (FP): **{**fp**}** (**{**int(fp\_rate\*100)**}**%)')

print(f'False Negative (FN): **{**fn**}** (**{**int(fn\_rate\*100)**}**%)')

In [ ]:

ac = metrics.accuracy\_score(y, y\_hat)

precision = metrics.precision\_score(y, y\_hat)

recall = metrics.recall\_score(y, y\_hat)

f1 = metrics.f1\_score(y, y\_hat)

print(f"Accuracy: **{**ac**}**")

print(f"recall: **{**recall**}**")

print(f"precision: **{**precision**}**")

print(f"f1: **{**f1**}**")

In [ ]:

*# Plot a confusion matrix.*

*# cm is the confusion matrix, names are the names of the classes.*

**def** plot\_confusion\_matrix(cm, names, title = 'Confusion matrix', cmap = plt.cm.Blues):

*# plt.imshow displays the image on the axes, but if you need to*

*# display multiple images you use show() to finish the figure.*

*# interpolation = 'none': works well when a big image is scaled down*

*# interpolation = 'nearest': works well when a small image is scaled up*

*# cmap: The registered colormap name used to map scalar data to colors.*

plt.imshow(cm, interpolation='nearest', cmap = cmap)

plt.title(title)

plt.colorbar()

tick\_marks = np.arange(len(names))

plt.xticks(tick\_marks, names, rotation = 45)

plt.yticks(tick\_marks, names)

*# Automatically adjust subplot parameters to give specified padding.*

plt.tight\_layout()

plt.ylabel('True label')

plt.xlabel('Predicted label')

labels = ['T', 'F']

*# Compute confusion matrix*

cm = confusion\_matrix(y, y\_hat)

np.set\_printoptions(precision = 2)

print('Confusion matrix, without normalization')

print(cm)

plt.figure()

plot\_confusion\_matrix(cm, labels)

*# Normalize the confusion matrix by row*

*# (i.e. by the number of samples in each class)*

cm\_normalized = cm.astype('float') / cm.sum(axis = 1)[:, np.newaxis]

print('Normalized confusion matrix')

print(cm\_normalized)

plt.figure()

plot\_confusion\_matrix(cm\_normalized, labels, title = 'Normalized confusion matrix')

plt.show()

In [ ]:

*# Compute ROC curve and ROC area for each class*

*# tp\_rate = float(tp)/count\_pos*

*# fp\_rate = float(fp)/count\_neg*

fpr, tpr, thresholds = roc\_curve(y, y\_pred\_proba[:,1])

*# Compute Area Under the Curve (AUC) using the trapezoidal rule*

roc\_auc = auc(fpr, tpr)

print(f"Y: **{**y**}**")

print(f"Y\_HAT: **{**y\_hat**}**")

print(f"FPR: **{**fpr**}**")

print(f"TPR: **{**tpr**}**")

print(f"thresholds: **{**thresholds**}**")

print (F"Optimal threshold index: **{**np.argmax(tpr - fpr)**}**")

print (F"Optimal threshold value: **{**thresholds[np.argmax(tpr - fpr)]**}**")

print(f"AUC: **{**roc\_auc**}**")

In [ ]:

plt.figure()

lw = 2

plt.plot(fpr, tpr, color = 'darkorange',

lw = lw, label = 'ROC curve (area = **%0.2f**)' % roc\_auc)

plt.plot([0, 1], [0, 1], color = 'navy', lw = lw, linestyle = '--')

plt.xlim([0.0, 1.0])

plt.ylim([0.0, 1.05])

plt.xlabel('False Positive Rate')

plt.ylabel('True Positive Rate')

plt.title('Receiver operating characteristic')

plt.legend(loc = "lower right")

plt.show()

In [ ]:

print(f"FPR: **{**fpr**}**")

print(f"TPR: **{**tpr**}**")

print(f"thresholds: **{**thresholds**}**")

In [ ]:

classification = pd.DataFrame({'y': y\_test, 'yhat': y\_pred\_proba[:,1]})

THRESHOLD = 0.5294117647058824 *#Optimal Threshold Value*

y = np.array(y\_test)

y\_hat = np.array([(1 **if** item >= THRESHOLD **else** 0) **for** item **in** y\_pred\_proba[:,1]])

print(f'y\_test: **{**y\_test**}**')

print(f'y\_pred\_proba: **{**y\_pred\_proba**}**')

print(f'y: **{**y**}**')

print(f'yhat: **{**y\_hat**}**')

In [ ]:

count\_pos = sum(y==1)

count\_neg = sum(y==0)

count = len(y)

print(f'Positive count: **{**count\_pos**}**')

print(f'Negatice count: **{**count\_neg**}**')

tp = sum(np.logical\_and(y==1, y\_hat==1))

tp\_rate = float(tp)/count\_pos

tn = sum(np.logical\_and(y==0, y\_hat==0))

tn\_rate = float(tn)/count\_neg

fp = sum(np.logical\_and(y==0, y\_hat==1))

fp\_rate = float(fp)/count\_neg

fn = sum(np.logical\_and(y==1, y\_hat==0))

fn\_rate = float(fn)/count\_pos

print(f'Count: **{**count**}**')

print(f'True Positive (TP, sensativity): **{**tp**}** (**{**int(tp\_rate\*100)**}**%)')

print(f'True Negative (TN, specificity): **{**tn**}** (**{**int(tn\_rate\*100)**}**%)')

print(f'False Positive (FP): **{**fp**}** (**{**int(fp\_rate\*100)**}**%)')

print(f'False Negative (FN): **{**fn**}** (**{**int(fn\_rate\*100)**}**%)')

In [ ]:

ac = metrics.accuracy\_score(y, y\_hat)

precision = metrics.precision\_score(y, y\_hat)

recall = metrics.recall\_score(y, y\_hat)

f1 = metrics.f1\_score(y, y\_hat)

print(f"Accuracy: **{**ac**}**")

print(f"recall: **{**recall**}**")

print(f"precision: **{**precision**}**")

print(f"f1: **{**f1**}**")

In [ ]:

*# Plot a confusion matrix.*

*# cm is the confusion matrix, names are the names of the classes.*

**def** plot\_confusion\_matrix(cm, names, title = 'Confusion matrix', cmap = plt.cm.Blues):

*# plt.imshow displays the image on the axes, but if you need to display multiple images you use show() to finish the figure.*

*# interpolation = 'none': works well when a big image is scaled down*

*# interpolation = 'nearest': works well when a small image is scaled up*

*# cmap: The registered colormap name used to map scalar data to colors.*

plt.imshow(cm, interpolation='nearest', cmap = cmap)

plt.title(title)

plt.colorbar()

tick\_marks = np.arange(len(names))

plt.xticks(tick\_marks, names, rotation = 45)

plt.yticks(tick\_marks, names)

*# Automatically adjust subplot parameters to give specified padding.*

plt.tight\_layout()

plt.ylabel('True label')

plt.xlabel('Predicted label')

labels = ['T', 'F']

*# Compute confusion matrix*

cm = confusion\_matrix(y, y\_hat)

np.set\_printoptions(precision = 2)

print('Confusion matrix, without normalization')

print(cm)

plt.figure()

plot\_confusion\_matrix(cm, labels)

*# Normalize the confusion matrix by row*

*# (i.e. by the number of samples in each class)*

cm\_normalized = cm.astype('float') / cm.sum(axis = 1)[:, np.newaxis]

print('Normalized confusion matrix')

print(cm\_normalized)

plt.figure()

plot\_confusion\_matrix(cm\_normalized, labels, title = 'Normalized confusion matrix')

plt.show()

##### **Decile Analysis**

In [ ]:

y = np.array(y\_test)

y\_hat = np.array(y\_pred\_proba[:,1])

In [ ]:

*# Increase size and add a little noise*

np.random.seed(42)

y = np.concatenate([y, y, y, y])

y\_hat = np.concatenate([y\_hat, y\_hat, y\_hat, y\_hat])

y\_hat = y\_hat + np.random.normal(size = len(y\_hat)) / 10

y\_hat = np.clip(y\_hat,0.01,0.99)

print(y\_hat, len(y\_hat))

In [ ]:

data = pd.DataFrame({'y':y,'y\_hat':y\_hat})

data.sort\_values(by='y\_hat',ascending = **False**, inplace = **True**)

data['bucket'] = pd.qcut(range(len(data)), 10, labels = **False**) + 1

data

In [ ]:

data.drop('y\_hat', 1, inplace=**True**)

data['count'] = np.ones(len(data))

data = data.groupby(by='bucket').sum()

data

In [ ]:

data['score'] = data['y'].values / data['count'].values

data.columns = ['tp','count','score']

data

In [ ]:

data.drop('count', 1, inplace=**True**)

data.drop('tp', 1, inplace=**True**)

data.plot(kind = "bar")

##### **Log Loss**

In [ ]:

y = np.array(y\_test)

y\_hat = np.array(y\_pred\_proba[:,1])

In [ ]:

llos = metrics.log\_loss(y, y\_hat)

print(f"Log loss: **{**llos**}**")

#### **Median**

In [ ]:

X\_train, X\_test, y\_train, y\_test = train\_test\_split(medianheart.drop("target", axis = 1), medianheart.target, test\_size=0.4, random\_state = 495)

*# Convert DataFrame data into np.arrays*

*# The scikit-learn library requires the data be formatted as a numpy array.*

*# Here are doing the reformatting*

X = np.array(X\_train)

print(X, X.shape, "**\n**")

y = np.array(y\_train)

print(y, y.shape, "**\n**")

In [ ]:

error\_rate = []

**for** i **in** range(2,25):

knn = neighbors.KNeighborsClassifier(n\_neighbors=i, weights="uniform")

knn.fit(X\_train,y\_train)

pred\_i = knn.predict(X\_test)

error\_rate.append(np.mean(pred\_i != y\_test))

plt.figure(figsize=(10,6))

plt.plot(range(2,25),error\_rate, marker='o', markersize=5)

plt.title('Error Rate vs. K Value')

plt.xlabel('K')

plt.ylabel('Error Rate')

print("Minimum error: ",min(error\_rate),"at K =", error\_rate.index(min(error\_rate))+2)

In [ ]:

clf = neighbors.KNeighborsClassifier(19, weights='uniform')

trained\_model = clf.fit(X, y)

print ("Trained Model:", trained\_model, "**\n**")

*# view the model's score, which will indicate how good my model has been trained*

print ("Score = ", trained\_model.score(X, y), "**\n**")

*# Apply the learner to the new, unclassified observation.*

y\_pred = trained\_model.predict(X\_test)

print(y\_pred, "**\n**")

*# we can even look at the probabilities the learner assigned to each class*

y\_pred\_proba = trained\_model.predict\_proba(X\_test)

print(y\_pred\_proba, "**\n**")

In [ ]:

*# Plot a confusion matrix.*

*# cm is the confusion matrix, names are the names of the classes.*

**def** plot\_confusion\_matrix(cm, names, title = 'Confusion matrix', cmap = plt.cm.Blues):

*# plt.imshow displays the image on the axes, but if you need to*

*# display multiple images you use show() to finish the figure.*

*# interpolation = 'none': works well when a big image is scaled down*

*# interpolation = 'nearest': works well when a small image is scaled up*

*# cmap: The registered colormap name used to map scalar data to colors.*

plt.imshow(cm, interpolation='nearest', cmap = cmap)

plt.title(title)

plt.colorbar()

tick\_marks = np.arange(len(names))

plt.xticks(tick\_marks, names, rotation = 45)

plt.yticks(tick\_marks, names)

*# Automatically adjust subplot parameters to give specified padding.*

plt.tight\_layout()

plt.ylabel('True label')

plt.xlabel('Predicted label')

labels = ['T', 'F']

*# Compute confusion matrix*

cm = confusion\_matrix(y\_test, y\_pred)

np.set\_printoptions(precision = 2)

print('Confusion matrix, without normalization')

print(cm)

plt.figure()

plot\_confusion\_matrix(cm, labels)

*# Normalize the confusion matrix by row*

*# (i.e. by the number of samples in each class)*

cm\_normalized = cm.astype('float') / cm.sum(axis = 1)[:, np.newaxis]

print('Normalized confusion matrix')

print(cm\_normalized)

plt.figure()

plot\_confusion\_matrix(cm\_normalized, labels, title = 'Normalized confusion matrix')

plt.show()

In [ ]:

*# Accuracy, Precision, Recall and F1 Score*

ac = metrics.accuracy\_score(y\_test, y\_pred)

precision = metrics.precision\_score(y\_test, y\_pred)

recall = metrics.recall\_score(y\_test, y\_pred)

f1 = metrics.f1\_score(y\_test, y\_pred)

print(f"Accuracy: **{**ac**}**")

print(f"recall: **{**recall**}**")

print(f"precision: **{**precision**}**")

print(f"f1: **{**f1**}**")

##### **ROC AUC**

In [ ]:

classification = pd.DataFrame({'y': y\_test, 'yhat': y\_pred\_proba[:,1]})

THRESHOLD = 0.5 *#Random Threshold Value*

y = np.array(y\_test)

y\_hat = np.array([(1 **if** item >= THRESHOLD **else** 0) **for** item **in** y\_pred\_proba[:,1]])

print(f'y\_test: **{**y\_test**}**')

print(f'y\_pred\_proba: **{**y\_pred\_proba**}**')

print(f'y: **{**y**}**')

print(f'yhat: **{**y\_hat**}**')

In [ ]:

count\_pos = sum(y==1)

count\_neg = sum(y==0)

count = len(y)

print(f'Positive count: **{**count\_pos**}**')

print(f'Negative count: **{**count\_neg**}**')

tp = sum(np.logical\_and(y==1, y\_hat==1))

tp\_rate = float(tp)/count\_pos

tn = sum(np.logical\_and(y==0, y\_hat==0))

tn\_rate = float(tn)/count\_neg

fp = sum(np.logical\_and(y==0, y\_hat==1))

fp\_rate = float(fp)/count\_neg

fn = sum(np.logical\_and(y==1, y\_hat==0))

fn\_rate = float(fn)/count\_pos

print(f'Count: **{**count**}**')

print(f'True Positive (TP, sensativity): **{**tp**}** (**{**int(tp\_rate\*100)**}**%)')

print(f'True Negative (TN, specificity): **{**tn**}** (**{**int(tn\_rate\*100)**}**%)')

print(f'False Positive (FP): **{**fp**}** (**{**int(fp\_rate\*100)**}**%)')

print(f'False Negative (FN): **{**fn**}** (**{**int(fn\_rate\*100)**}**%)')

In [ ]:

ac = metrics.accuracy\_score(y, y\_hat)

precision = metrics.precision\_score(y, y\_hat)

recall = metrics.recall\_score(y, y\_hat)

f1 = metrics.f1\_score(y, y\_hat)

print(f"Accuracy: **{**ac**}**")

print(f"recall: **{**recall**}**")

print(f"precision: **{**precision**}**")

print(f"f1: **{**f1**}**")

In [ ]:

*# Plot a confusion matrix.*

*# cm is the confusion matrix, names are the names of the classes.*

**def** plot\_confusion\_matrix(cm, names, title = 'Confusion matrix', cmap = plt.cm.Blues):

*# plt.imshow displays the image on the axes, but if you need to*

*# display multiple images you use show() to finish the figure.*

*# interpolation = 'none': works well when a big image is scaled down*

*# interpolation = 'nearest': works well when a small image is scaled up*

*# cmap: The registered colormap name used to map scalar data to colors.*

plt.imshow(cm, interpolation='nearest', cmap = cmap)

plt.title(title)

plt.colorbar()

tick\_marks = np.arange(len(names))

plt.xticks(tick\_marks, names, rotation = 45)

plt.yticks(tick\_marks, names)

*# Automatically adjust subplot parameters to give specified padding.*

plt.tight\_layout()

plt.ylabel('True label')

plt.xlabel('Predicted label')

labels = ['T', 'F']

*# Compute confusion matrix*

cm = confusion\_matrix(y, y\_hat)

np.set\_printoptions(precision = 2)

print('Confusion matrix, without normalization')

print(cm)

plt.figure()

plot\_confusion\_matrix(cm, labels)

*# Normalize the confusion matrix by row*

*# (i.e. by the number of samples in each class)*

cm\_normalized = cm.astype('float') / cm.sum(axis = 1)[:, np.newaxis]

print('Normalized confusion matrix')

print(cm\_normalized)

plt.figure()

plot\_confusion\_matrix(cm\_normalized, labels, title = 'Normalized confusion matrix')

plt.show()

In [ ]:

*# Compute ROC curve and ROC area for each class*

*# tp\_rate = float(tp)/count\_pos*

*# fp\_rate = float(fp)/count\_neg*

fpr, tpr, thresholds = roc\_curve(y, y\_pred\_proba[:,1])

*# Compute Area Under the Curve (AUC) using the trapezoidal rule*

roc\_auc = auc(fpr, tpr)

print(f"Y: **{**y**}**")

print(f"Y\_HAT: **{**y\_hat**}**")

print(f"FPR: **{**fpr**}**")

print(f"TPR: **{**tpr**}**")

print(f"thresholds: **{**thresholds**}**")

print (F"Optimal threshold index: **{**np.argmax(tpr - fpr)**}**")

print (F"Optimal threshold value: **{**thresholds[np.argmax(tpr - fpr)]**}**")

print(f"AUC: **{**roc\_auc**}**")

In [ ]:

plt.figure()

lw = 2

plt.plot(fpr, tpr, color = 'darkorange',

lw = lw, label = 'ROC curve (area = **%0.2f**)' % roc\_auc)

plt.plot([0, 1], [0, 1], color = 'navy', lw = lw, linestyle = '--')

plt.xlim([0.0, 1.0])

plt.ylim([0.0, 1.05])

plt.xlabel('False Positive Rate')

plt.ylabel('True Positive Rate')

plt.title('Receiver operating characteristic')

plt.legend(loc = "lower right")

plt.show()

In [ ]:

print(f"FPR: **{**fpr**}**")

print(f"TPR: **{**tpr**}**")

print(f"thresholds: **{**thresholds**}**")

In [ ]:

classification = pd.DataFrame({'y': y\_test, 'yhat': y\_pred\_proba[:,1]})

THRESHOLD = 0.5263157894736842 *#Optimal Threshold Value*

y = np.array(y\_test)

y\_hat = np.array([(1 **if** item >= THRESHOLD **else** 0) **for** item **in** y\_pred\_proba[:,1]])

print(f'y\_test: **{**y\_test**}**')

print(f'y\_pred\_proba: **{**y\_pred\_proba**}**')

print(f'y: **{**y**}**')

print(f'yhat: **{**y\_hat**}**')

In [ ]:

count\_pos = sum(y==1)

count\_neg = sum(y==0)

count = len(y)

print(f'Positive count: **{**count\_pos**}**')

print(f'Negative count: **{**count\_neg**}**')

tp = sum(np.logical\_and(y==1, y\_hat==1))

tp\_rate = float(tp)/count\_pos

tn = sum(np.logical\_and(y==0, y\_hat==0))

tn\_rate = float(tn)/count\_neg

fp = sum(np.logical\_and(y==0, y\_hat==1))

fp\_rate = float(fp)/count\_neg

fn = sum(np.logical\_and(y==1, y\_hat==0))

fn\_rate = float(fn)/count\_pos

print(f'Count: **{**count**}**')

print(f'True Positive (TP, sensativity): **{**tp**}** (**{**int(tp\_rate\*100)**}**%)')

print(f'True Negative (TN, specificity): **{**tn**}** (**{**int(tn\_rate\*100)**}**%)')

print(f'False Positive (FP): **{**fp**}** (**{**int(fp\_rate\*100)**}**%)')

print(f'False Negative (FN): **{**fn**}** (**{**int(fn\_rate\*100)**}**%)')

In [ ]:

ac = metrics.accuracy\_score(y, y\_hat)

precision = metrics.precision\_score(y, y\_hat)

recall = metrics.recall\_score(y, y\_hat)

f1 = metrics.f1\_score(y, y\_hat)

print(f"Accuracy: **{**ac**}**")

print(f"recall: **{**recall**}**")

print(f"precision: **{**precision**}**")

print(f"f1: **{**f1**}**")

In [ ]:

*# Plot a confusion matrix.*

*# cm is the confusion matrix, names are the names of the classes.*

**def** plot\_confusion\_matrix(cm, names, title = 'Confusion matrix', cmap = plt.cm.Blues):

*# plt.imshow displays the image on the axes, but if you need to*

*# display multiple images you use show() to finish the figure.*

*# interpolation = 'none': works well when a big image is scaled down*

*# interpolation = 'nearest': works well when a small image is scaled up*

*# cmap: The registered colormap name used to map scalar data to colors.*

plt.imshow(cm, interpolation='nearest', cmap = cmap)

plt.title(title)

plt.colorbar()

tick\_marks = np.arange(len(names))

plt.xticks(tick\_marks, names, rotation = 45)

plt.yticks(tick\_marks, names)

*# Automatically adjust subplot parameters to give specified padding.*

plt.tight\_layout()

plt.ylabel('True label')

plt.xlabel('Predicted label')

labels = ['T', 'F']

*# Compute confusion matrix*

cm = confusion\_matrix(y, y\_hat)

np.set\_printoptions(precision = 2)

print('Confusion matrix, without normalization')

print(cm)

plt.figure()

plot\_confusion\_matrix(cm, labels)

*# Normalize the confusion matrix by row*

*# (i.e. by the number of samples in each class)*

cm\_normalized = cm.astype('float') / cm.sum(axis = 1)[:, np.newaxis]

print('Normalized confusion matrix')

print(cm\_normalized)

plt.figure()

plot\_confusion\_matrix(cm\_normalized, labels, title = 'Normalized confusion matrix')

plt.show()

##### **Decile Analysis**

In [ ]:

y = np.array(y\_test)

y\_hat = np.array(y\_pred\_proba[:,1])

In [ ]:

*# Increase size and add a little noise*

np.random.seed(42)

y = np.concatenate([y, y, y, y])

y\_hat = np.concatenate([y\_hat, y\_hat, y\_hat, y\_hat])

y\_hat = y\_hat + np.random.normal(size = len(y\_hat)) / 10

y\_hat = np.clip(y\_hat,0.01,0.99)

print(y\_hat, len(y\_hat))

In [ ]:

data = pd.DataFrame({'y':y,'y\_hat':y\_hat})

data.sort\_values(by='y\_hat',ascending = **False**, inplace = **True**)

data['bucket'] = pd.qcut(range(len(data)), 10, labels = **False**) + 1

data

In [ ]:

data.drop('y\_hat', 1, inplace=**True**)

data['count'] = np.ones(len(data))

data = data.groupby(by='bucket').sum()

data

In [ ]:

data['score'] = data['y'].values / data['count'].values

data.columns = ['tp','count','score']

data

In [ ]:

data.drop('count', 1, inplace=**True**)

data.drop('tp', 1, inplace=**True**)

data.plot(kind = "bar")

##### **Log Loss**

In [ ]:

y = np.array(y\_test)

y\_hat = np.array(y\_pred\_proba[:,1])

In [ ]:

llos = metrics.log\_loss(y, y\_hat)

print(f"Log loss: **{**llos**}**")

### **KMEANS**

#### **Mean**

In [ ]:

X\_train, X\_test, y\_train, y\_test = train\_test\_split(meanheart.drop("target", axis = 1), meanheart.target, test\_size=0.4, random\_state = 495)

*# initializing K-Means*

model = KMeans(n\_clusters=2)

*# Fitting with the traning data inputs*

kmeans\_model = model.fit(X\_train, y\_train)

*# predicting the clusters*

y\_pred = kmeans\_model.predict(X\_test)

print ("Predictions**\n**", y\_pred)

In [ ]:

*# Getting the cluster centers*

C = kmeans\_model.cluster\_centers\_

print (pd.DataFrame(C, columns = X\_test.columns))

In [ ]:

*#Score*

print("Score :", kmeans\_model.score(X\_test, y\_test), "**\n**")

*# Accuracy, Precision, Recall and F1 Score*

ac = metrics.accuracy\_score(y\_test, y\_pred)

precision = metrics.precision\_score(y\_test, y\_pred)

recall = metrics.recall\_score(y\_test, y\_pred)

f1 = metrics.f1\_score(y\_test, y\_pred)

print(f"Accuracy: **{**ac**}**")

print(f"recall: **{**recall**}**")

print(f"precision: **{**precision**}**")

print(f"f1: **{**f1**}**")

In [ ]:

classification = pd.DataFrame({'y': y\_test, 'yhat': y\_pred})

y = np.array(y\_test)

y\_hat = np.array(y\_pred)

print(f'y\_test: **{**y\_test**}**')

print(f'y\_pred: **{**y\_pred**}**')

print(f'y: **{**y**}**')

print(f'yhat: **{**y\_hat**}**')

In [ ]:

count\_pos = sum(y==1)

count\_neg = sum(y==0)

count = len(y)

print(f'Positive count: **{**count\_pos**}**')

print(f'Negative count: **{**count\_neg**}**')

tp = sum(np.logical\_and(y==1, y\_hat==1))

tp\_rate = float(tp)/count\_pos

tn = sum(np.logical\_and(y==0, y\_hat==0))

tn\_rate = float(tn)/count\_neg

fp = sum(np.logical\_and(y==0, y\_hat==1))

fp\_rate = float(fp)/count\_neg

fn = sum(np.logical\_and(y==1, y\_hat==0))

fn\_rate = float(fn)/count\_pos

print(f'Count: **{**count**}**')

print(f'True Positive (TP, sensativity): **{**tp**}** (**{**int(tp\_rate\*100)**}**%)')

print(f'True Negative (TN, specificity): **{**tn**}** (**{**int(tn\_rate\*100)**}**%)')

print(f'False Positive (FP): **{**fp**}** (**{**int(fp\_rate\*100)**}**%)')

print(f'False Negative (FN): **{**fn**}** (**{**int(fn\_rate\*100)**}**%)')

In [ ]:

*# Plot a confusion matrix.*

*# cm is the confusion matrix, names are the names of the classes.*

**def** plot\_confusion\_matrix(cm, names, title = 'Confusion matrix', cmap = plt.cm.Blues):

*# plt.imshow displays the image on the axes, but if you need to*

*# display multiple images you use show() to finish the figure.*

*# interpolation = 'none': works well when a big image is scaled down*

*# interpolation = 'nearest': works well when a small image is scaled up*

*# cmap: The registered colormap name used to map scalar data to colors.*

plt.imshow(cm, interpolation='nearest', cmap = cmap)

plt.title(title)

plt.colorbar()

tick\_marks = np.arange(len(names))

plt.xticks(tick\_marks, names, rotation = 45)

plt.yticks(tick\_marks, names)

*# Automatically adjust subplot parameters to give specified padding.*

plt.tight\_layout()

plt.ylabel('True label')

plt.xlabel('Predicted label')

labels = ['T', 'F']

*# Compute confusion matrix*

cm = confusion\_matrix(y, y\_hat)

np.set\_printoptions(precision = 2)

print('Confusion matrix, without normalization')

print(cm)

plt.figure()

plot\_confusion\_matrix(cm, labels)

*# Normalize the confusion matrix by row*

*# (i.e. by the number of samples in each class)*

cm\_normalized = cm.astype('float') / cm.sum(axis = 1)[:, np.newaxis]

print('Normalized confusion matrix')

print(cm\_normalized)

plt.figure()

plot\_confusion\_matrix(cm\_normalized, labels, title = 'Normalized confusion matrix')

plt.show()

#### **Median**

In [ ]:

X\_train, X\_test, y\_train, y\_test = train\_test\_split(medianheart.drop("target", axis = 1), medianheart.target, test\_size=0.4, random\_state = 495)

*# initializing K-Means*

model = KMeans(n\_clusters=2)

*# Fitting with the traning data inputs*

kmeans\_model = model.fit(X\_train, y\_train)

*# predicting the clusters*

y\_pred = kmeans\_model.predict(X\_test)

print ("Predictions**\n**", y\_pred)

In [ ]:

*# Getting the cluster centers*

C = kmeans\_model.cluster\_centers\_

print (pd.DataFrame(C, columns = X\_test.columns))

In [ ]:

*#Score*

print("Score :", kmeans\_model.score(X\_test, y\_test), "**\n**")

*# Accuracy, Precision, Recall and F1 Score*

ac = metrics.accuracy\_score(y\_test, y\_pred)

precision = metrics.precision\_score(y\_test, y\_pred)

recall = metrics.recall\_score(y\_test, y\_pred)

f1 = metrics.f1\_score(y\_test, y\_pred)

print(f"Accuracy: **{**ac**}**")

print(f"recall: **{**recall**}**")

print(f"precision: **{**precision**}**")

print(f"f1: **{**f1**}**")

In [ ]:

classification = pd.DataFrame({'y': y\_test, 'yhat': y\_pred})

y = np.array(y\_test)

y\_hat = np.array(y\_pred)

print(f'y\_test: **{**y\_test**}**')

print(f'y\_pred: **{**y\_pred**}**')

print(f'y: **{**y**}**')

print(f'yhat: **{**y\_hat**}**')

In [ ]:

count\_pos = sum(y==1)

count\_neg = sum(y==0)

count = len(y)

print(f'Positive count: **{**count\_pos**}**')

print(f'Negative count: **{**count\_neg**}**')

tp = sum(np.logical\_and(y==1, y\_hat==1))

tp\_rate = float(tp)/count\_pos

tn = sum(np.logical\_and(y==0, y\_hat==0))

tn\_rate = float(tn)/count\_neg

fp = sum(np.logical\_and(y==0, y\_hat==1))

fp\_rate = float(fp)/count\_neg

fn = sum(np.logical\_and(y==1, y\_hat==0))

fn\_rate = float(fn)/count\_pos

print(f'Count: **{**count**}**')

print(f'True Positive (TP, sensativity): **{**tp**}** (**{**int(tp\_rate\*100)**}**%)')

print(f'True Negative (TN, specificity): **{**tn**}** (**{**int(tn\_rate\*100)**}**%)')

print(f'False Positive (FP): **{**fp**}** (**{**int(fp\_rate\*100)**}**%)')

print(f'False Negative (FN): **{**fn**}** (**{**int(fn\_rate\*100)**}**%)')

In [ ]:

*# Plot a confusion matrix.*

*# cm is the confusion matrix, names are the names of the classes.*

**def** plot\_confusion\_matrix(cm, names, title = 'Confusion matrix', cmap = plt.cm.Blues):

*# plt.imshow displays the image on the axes, but if you need to*

*# display multiple images you use show() to finish the figure.*

*# interpolation = 'none': works well when a big image is scaled down*

*# interpolation = 'nearest': works well when a small image is scaled up*

*# cmap: The registered colormap name used to map scalar data to colors.*

plt.imshow(cm, interpolation='nearest', cmap = cmap)

plt.title(title)

plt.colorbar()

tick\_marks = np.arange(len(names))

plt.xticks(tick\_marks, names, rotation = 45)

plt.yticks(tick\_marks, names)

*# Automatically adjust subplot parameters to give specified padding.*

plt.tight\_layout()

plt.ylabel('True label')

plt.xlabel('Predicted label')

labels = ['T', 'F']

*# Compute confusion matrix*

cm = confusion\_matrix(y, y\_hat)

np.set\_printoptions(precision = 2)

print('Confusion matrix, without normalization')

print(cm)

plt.figure()

plot\_confusion\_matrix(cm, labels)

*# Normalize the confusion matrix by row*

*# (i.e. by the number of samples in each class)*

cm\_normalized = cm.astype('float') / cm.sum(axis = 1)[:, np.newaxis]

print('Normalized confusion matrix')

print(cm\_normalized)

plt.figure()

plot\_confusion\_matrix(cm\_normalized, labels, title = 'Normalized confusion matrix')

plt.show()

# Future Scope Of Improvements

The project includes only a small sample set and works through only 5 Models. Keeping that in mind:

* The project can be bettered by the collection and implementation of more data with more accurate values.
* The dataset can also be passes through more Models to check their compatibility to check which Model suits the dataset the best.

# Project Certificate

This is to certify that Mr/Ms Yashowardhan Samdhani of Don Bosco School, Park Circus, Registration No: 2015252, has successfully completed a project on Heart Attack Prediction using Python Machine Learning under the guidance of Prof. Arnab Chakraborty.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Don Bosco School, Park Circus

# Project Certificate

This is to certify that Mr Adish Bhagwat of Sri kumaran's public school, Mallasandra, Registration No: , has successfully completed a project on Heart Attack Prediction using Python Machine Learning under the guidance of Prof. Arnab Chakraborty.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Sri Kumarans public school, Mallasandra

# Project Certificate

This is to certify that Mr Arya Srivastava of Garodia International Centre for Learning Mumbai, Registration No: , has successfully completed a project on Heart Attack Prediction using Python Machine Learning under the guidance of Prof. Arnab Chakraborty.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Garodia International Centre for Learning Mumbai

# Project Certificate

This is to certify that Mr S Sanjith of St. Joseph Boys' High School Registration No: SJBHS3542, has successfully completed a project on Heart Attack Prediction using Python Machine Learning under the guidance of Prof. Arnab Chakraborty.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

St. Joseph's Boys' High School, Bangalore