HEART ATTACK PREDICTION

GROUP MEMBERS:

YASHOWARDHAN SAMDHANI

ADISH BHAGWAT

ARYA SRIVASTAVA

S. SANJITH

TABLE OF CONTENTS

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

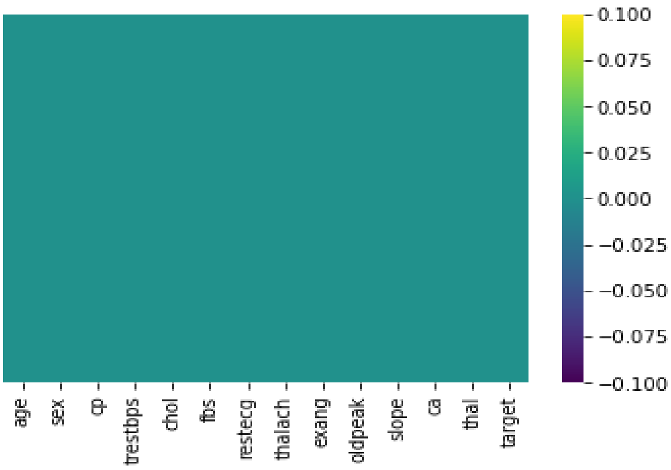
DATA TYPE

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

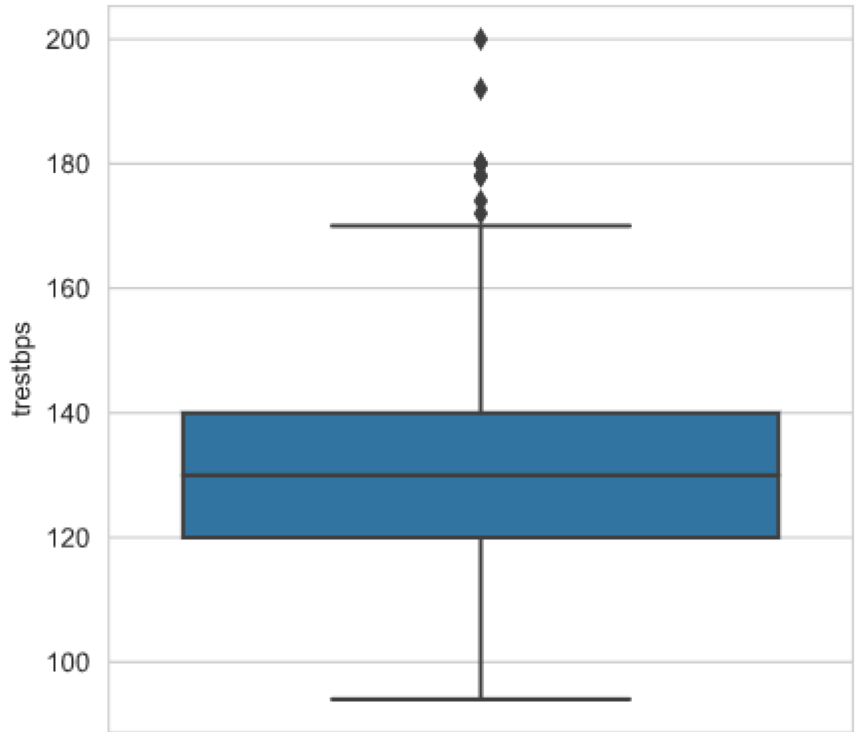
```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 302 entries, 0 to 301
Data columns (total 14 columns):
 #   Column    Non-Null Count  Dtype
---  ------    --------------  -----
 0   age       302 non-null    int64
 1   sex       302 non-null    int64
 2   cp        302 non-null    int64
 3   trestbps  302 non-null    int64
 4   chol      302 non-null    int64
 5   fbs       302 non-null    int64
 6   restecg   302 non-null    int64
 7   thalach   302 non-null    int64
 8   exang     302 non-null    int64
 9   oldpeak   302 non-null    float64
 10  slope     302 non-null    int64
 11  ca        302 non-null    int64
 12  thal      302 non-null    int64
 13  target    302 non-null    int64
dtypes: float64(1), int64(13)
memory usage: 33.2 KB
```

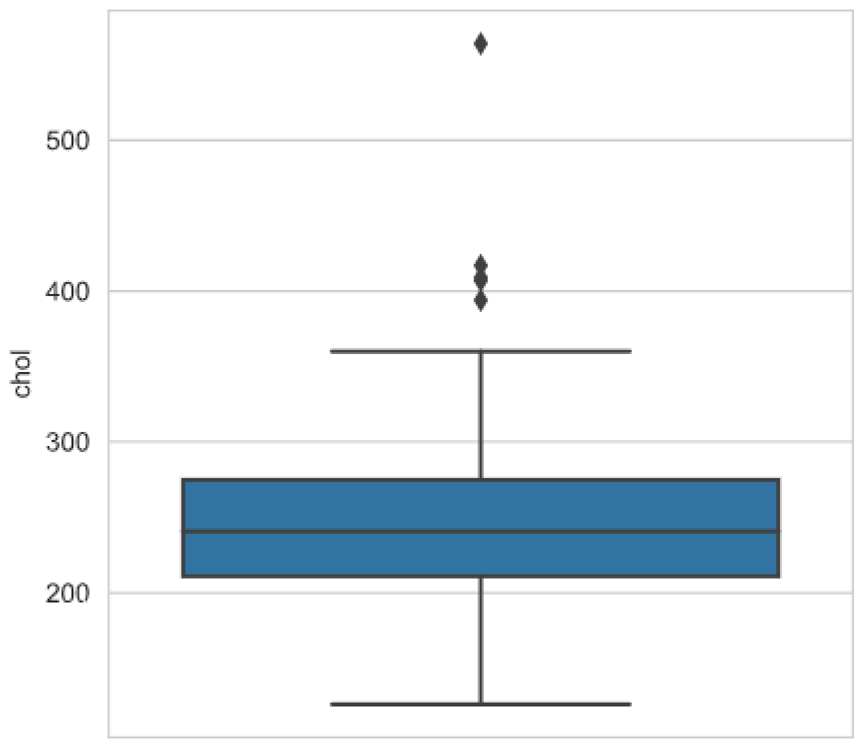
REMOVING OUTLIERS

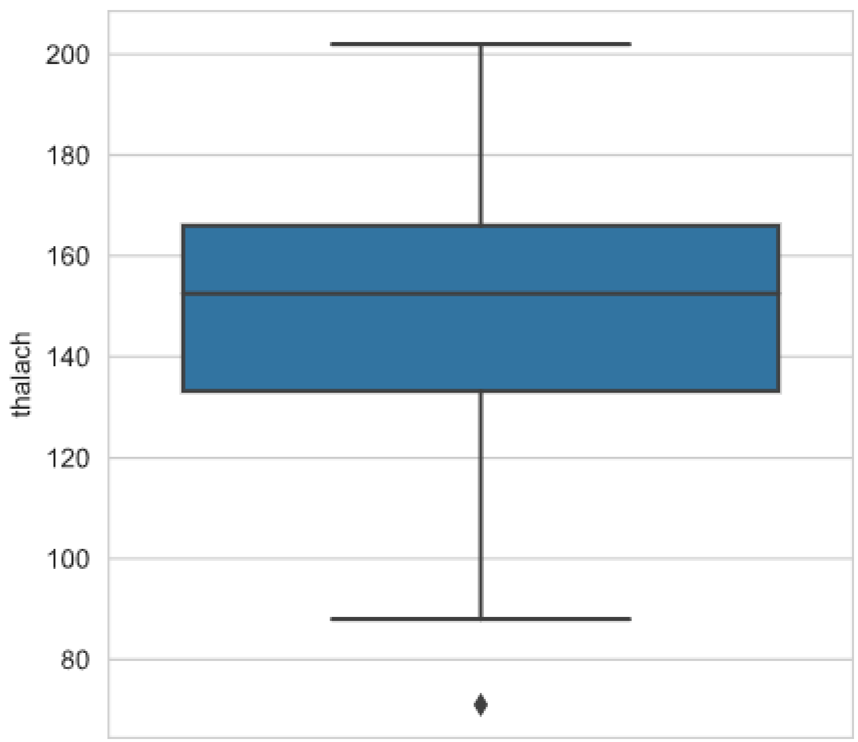
The number of outliers in each feature variable are:

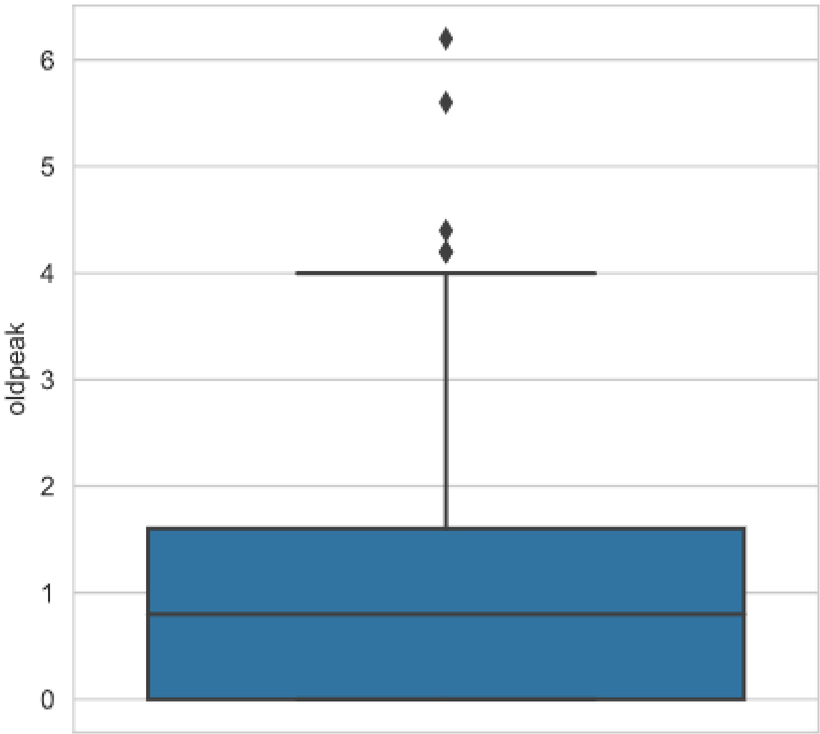
1. age = 0
2. sex = 0
3. cp (4 values) = 0
4. trestbps = 9
5. chol in mg/dl = 14
6. fbs > 120 mg/dl = 0
7. restecg (values 0,1,2) = 0
8. thalach = 45
9. exang = 0
10. oldpeak = 49
11. slope = 0
12. ca = 0
13. thal = 0

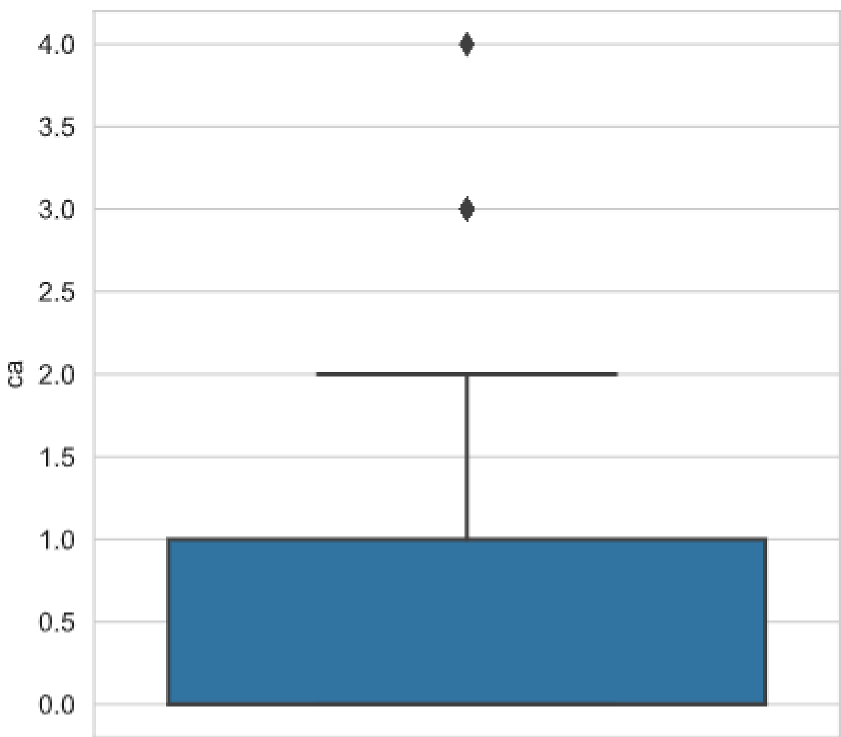
BEFORE

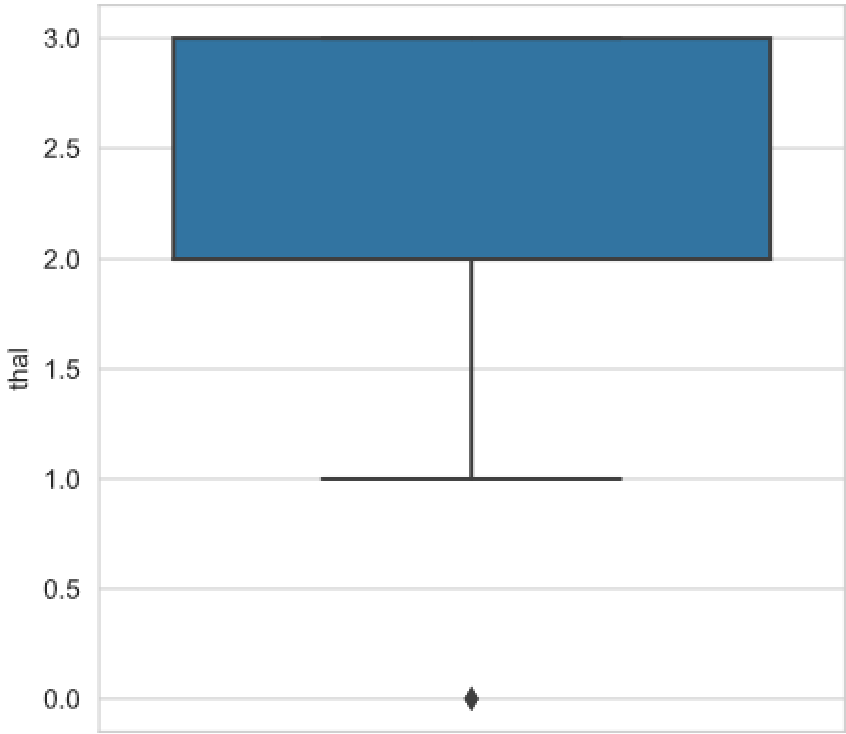



500
400
300
200
chol


200
180
160
140
120
100
80
thalach

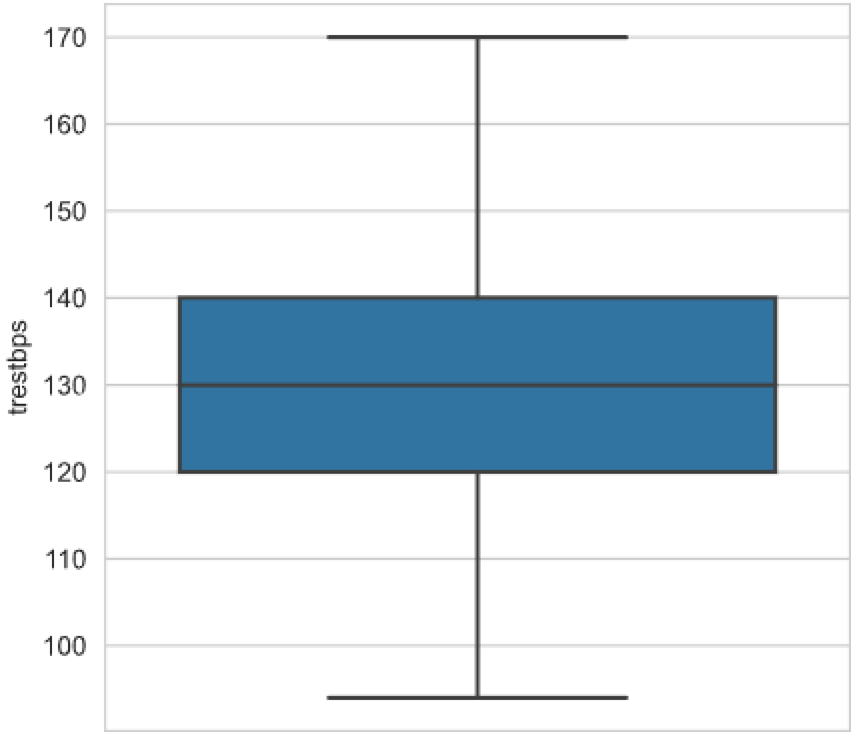

oldpeak
0
1
2
3
4
5
6

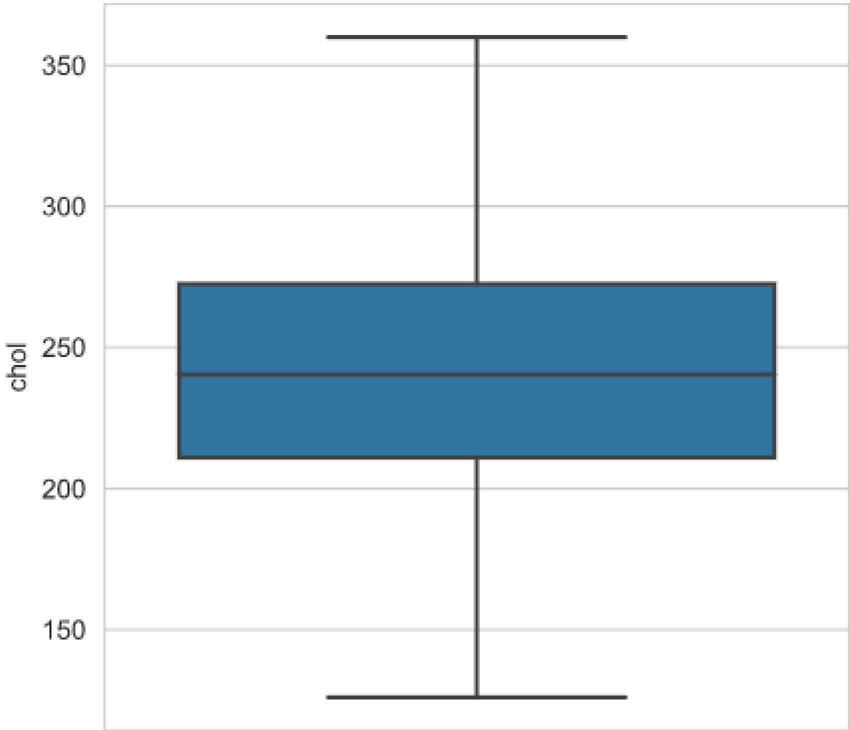

ca
0.0
0.5
1.0
1.5
2.0
2.5
3.0
3.5
4.0

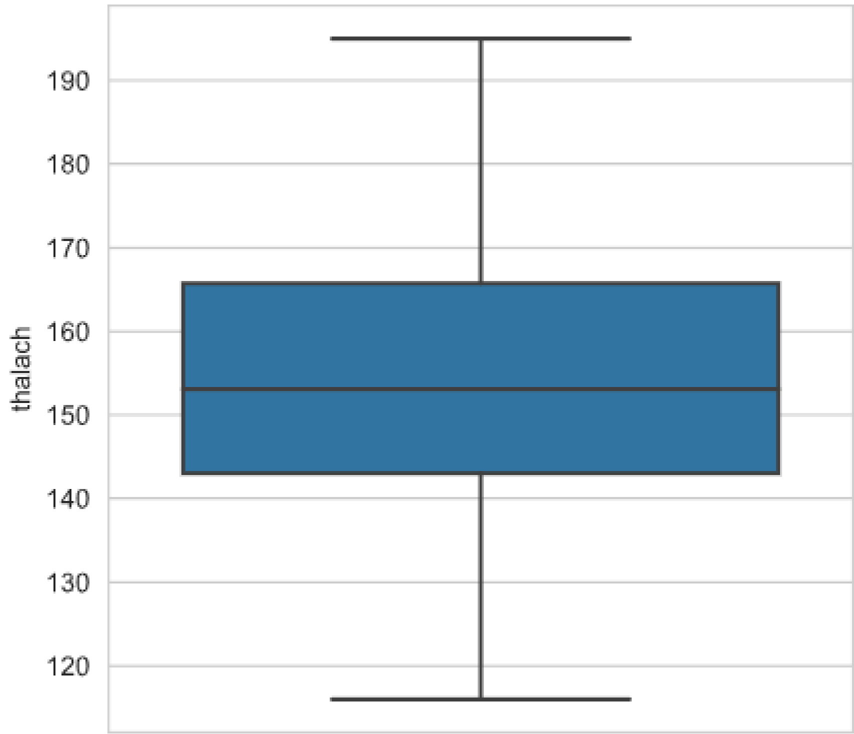

3.0
2.5
2.0
1.5
1.0
0.5
0.0
thal

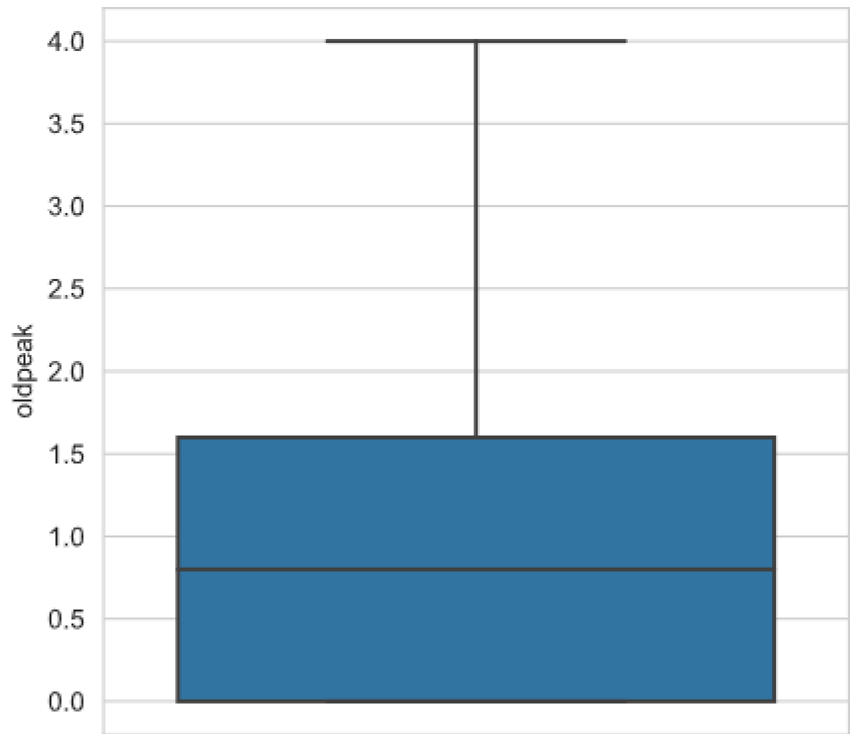
AFTER

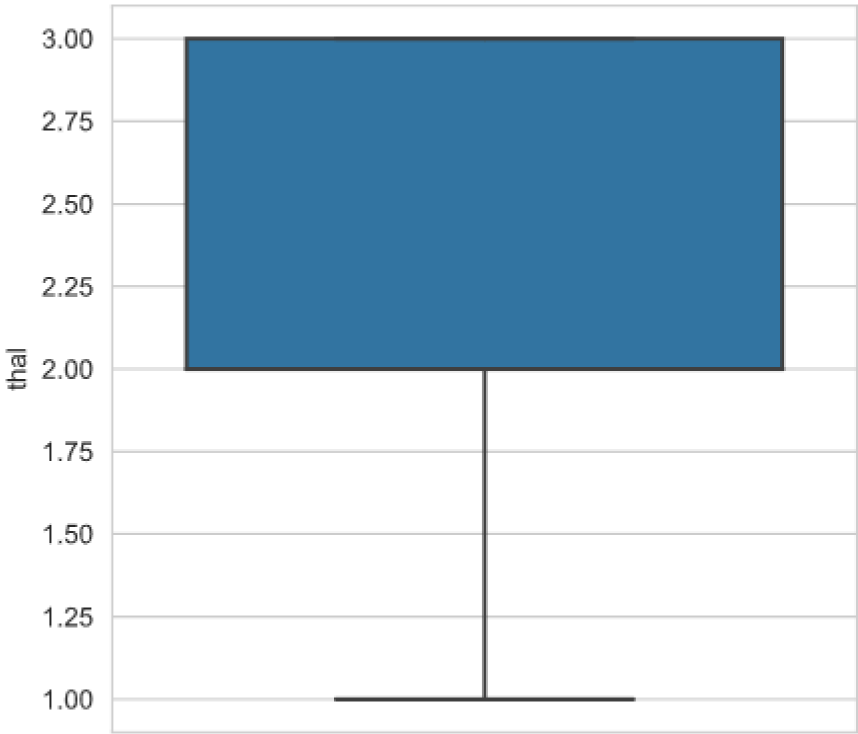
REPLACED BY MEDIAN



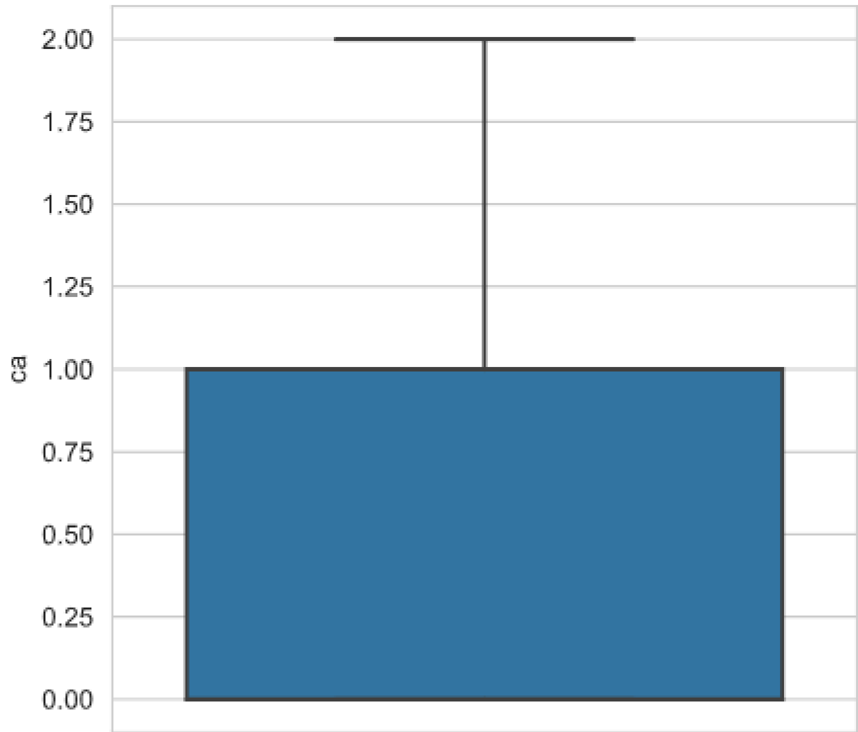



thalach
190
180
170
160
150
140
130
120

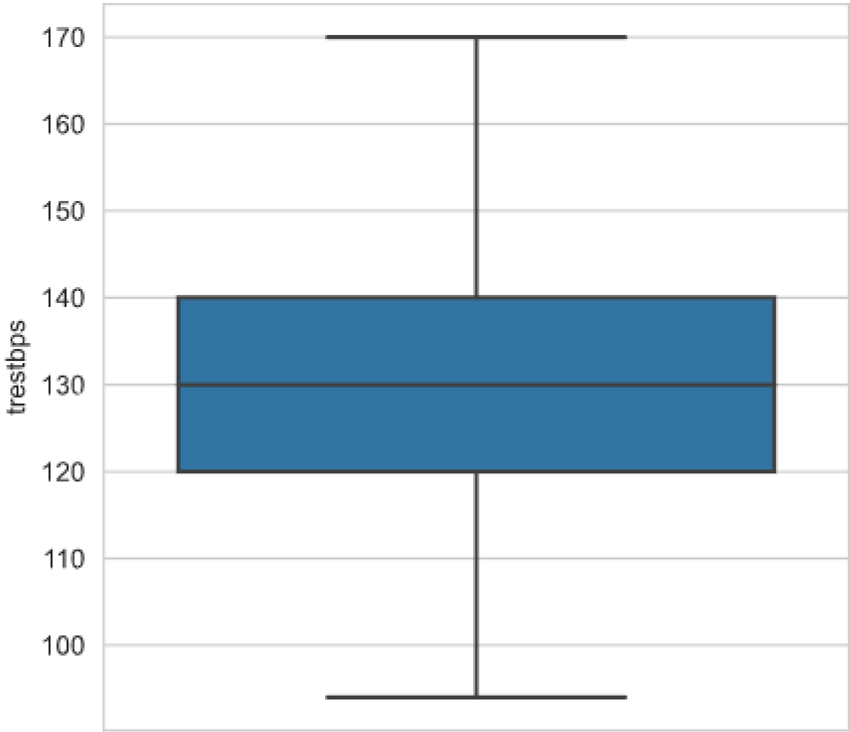

oldpeak
4.0
3.5
3.0
2.5
2.0
1.5
1.0
0.5
0.0

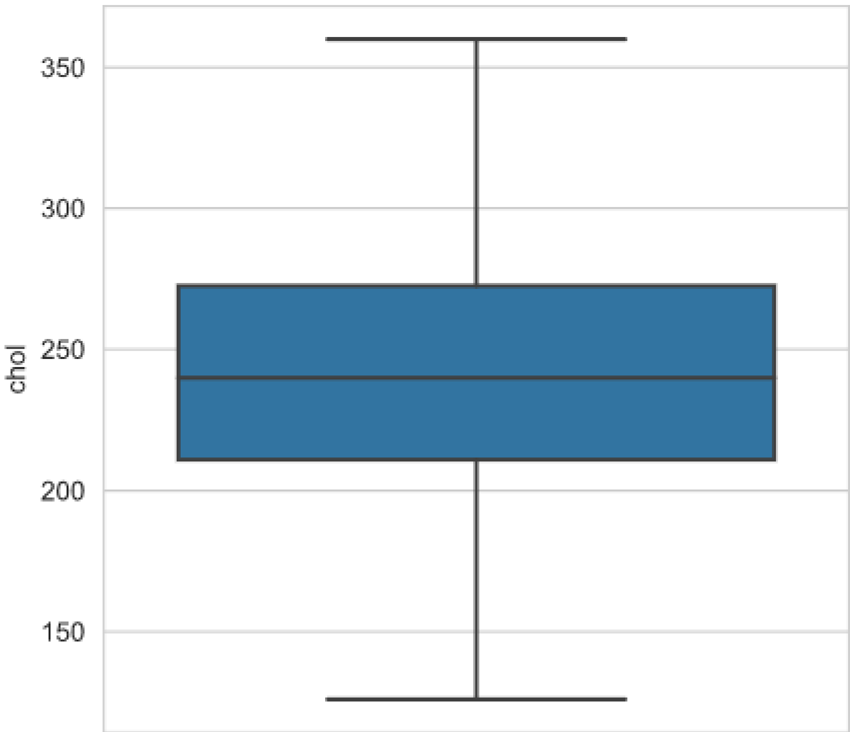


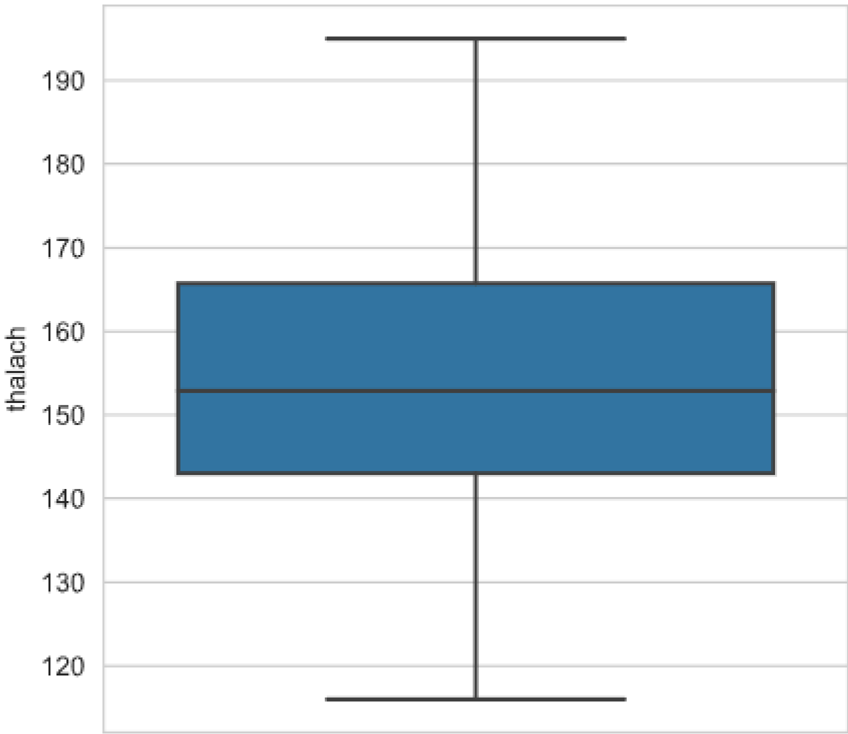
3.00
2.75
2.50
2.25
thal
2.00
1.75
1.50
1.25
1.00

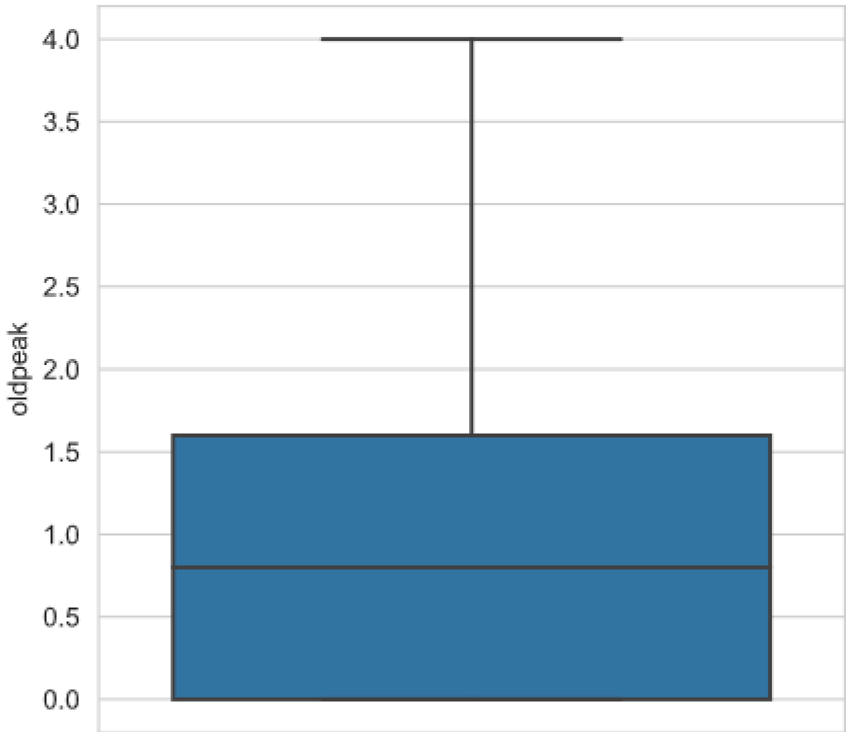


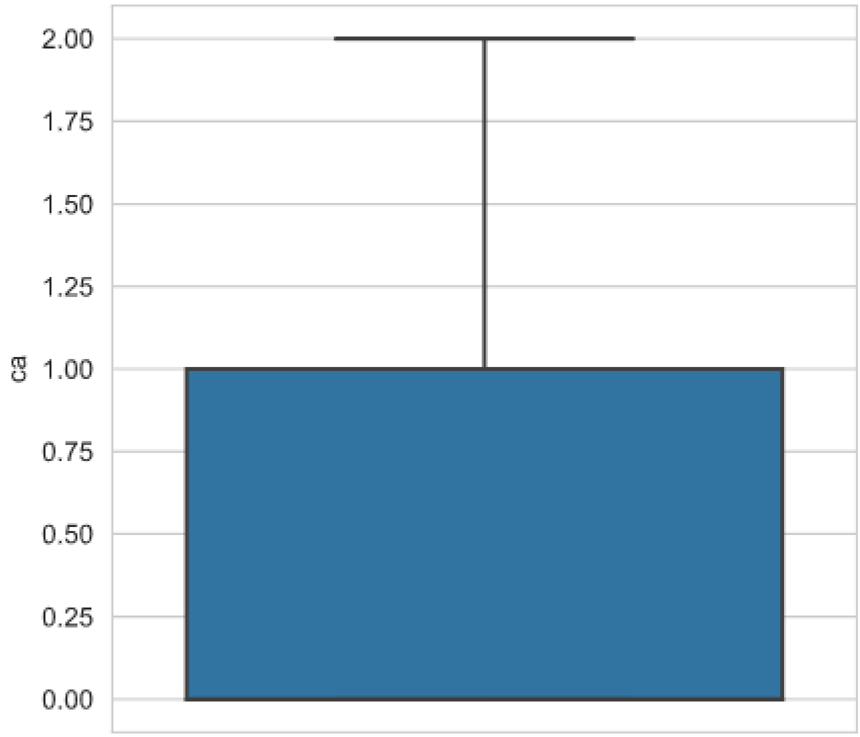
2.00
1.75
1.50
1.25
ca
1.00
0.75
0.50
0.25
0.00

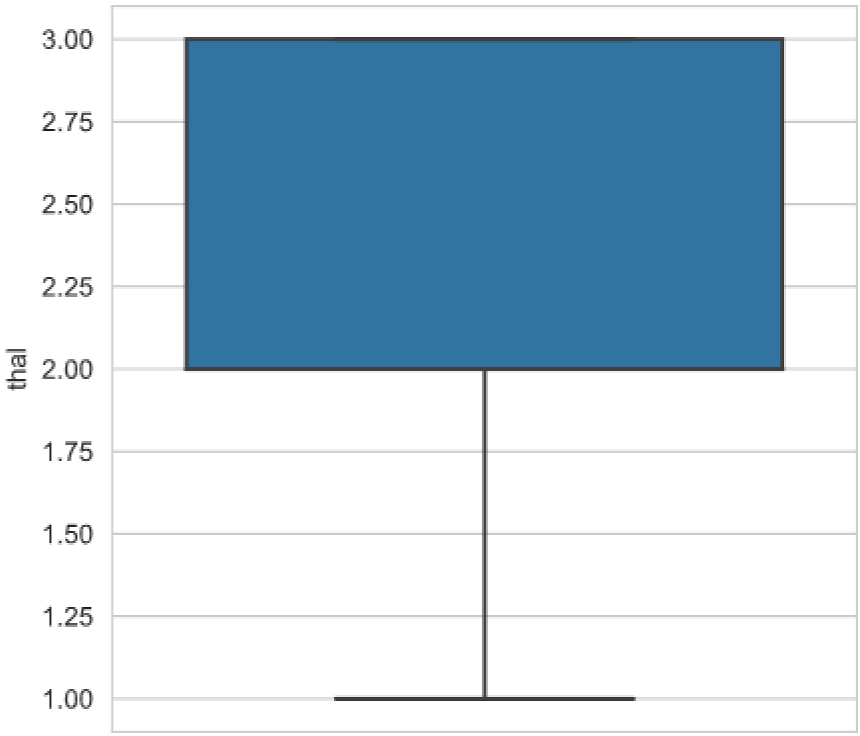

170
160
150
140
130
120
110
100
trestbps


350
300
250
200
150
chol


thalach
190
180
170
160
150
140
130
120

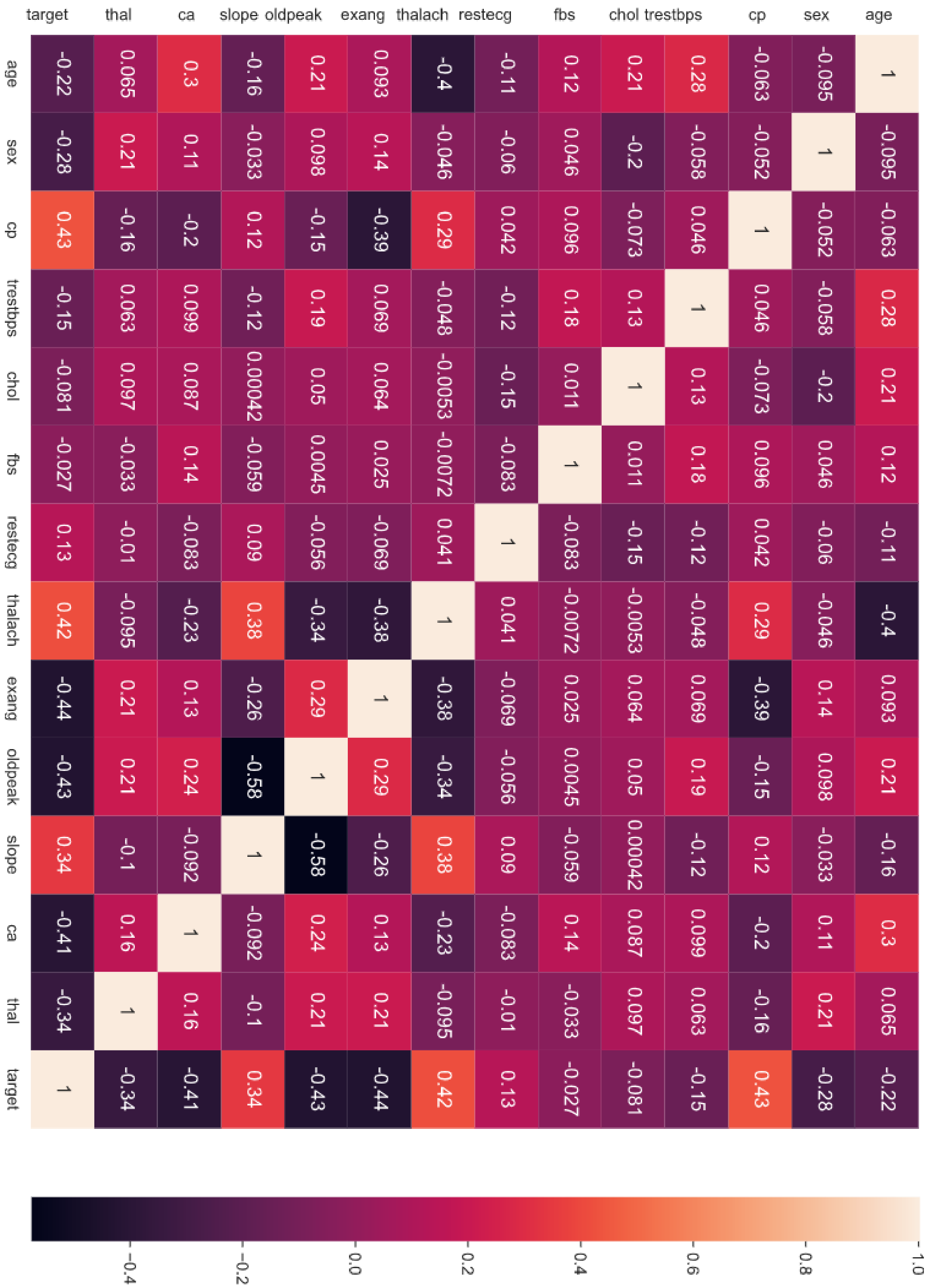

oldpeak
4.0
3.5
3.0
2.5
2.0
1.5
1.0
0.5
0.0


2.00
1.75
1.50
1.25
1.00
0.75
0.50
0.25
0.00
ca

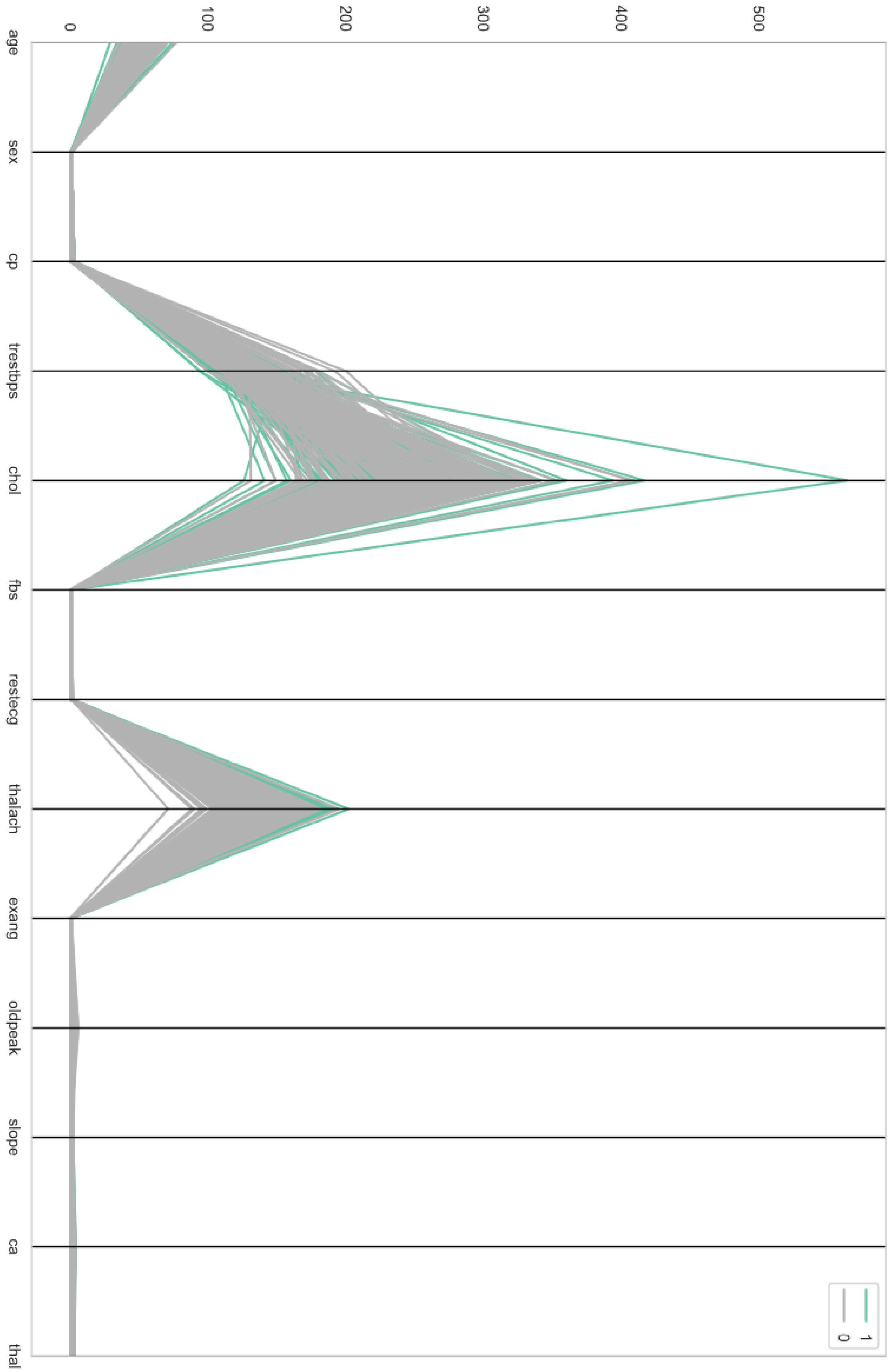

3.00
2.75
2.50
2.25
2.00
1.75
1.50
1.25
1.00
thal

DATA STUDY

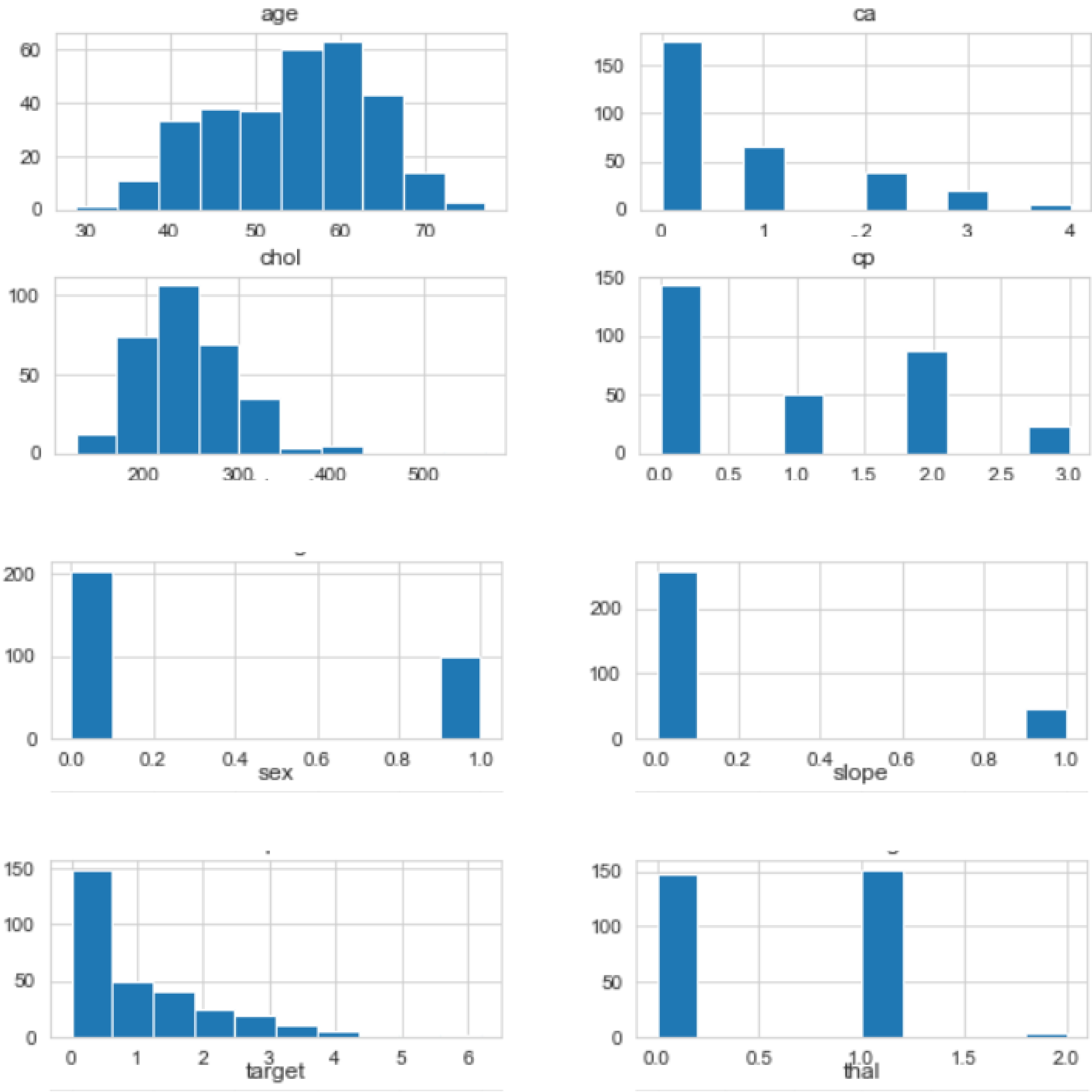
CORRELATION

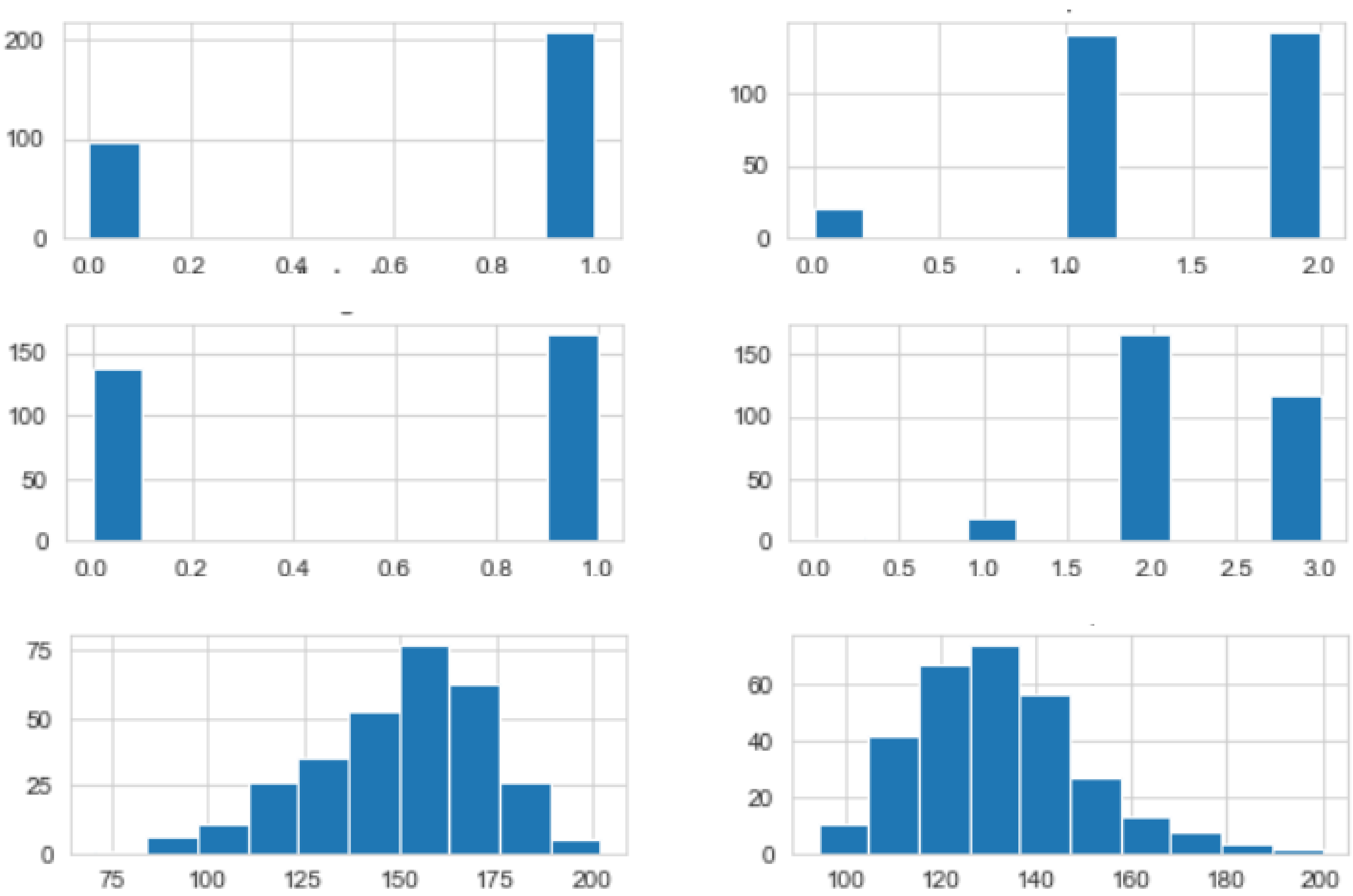


PARALLEL COORDINATES



HISTOGRAM




0
100
200
0.0 0.2 0.4 0.6 0.8 1.0
0
50
100
0.0 0.5 1.0 1.5 2.0
0
50
100
150
0.0 0.2 0.4 0.6 0.8 1.0
0
50
100
150
0.0 0.5 1.0 1.5 2.0 2.5 3.0
0
25
50
75
75 100 125 150 175 200
0
20
40
60
100 120 140 160 180 200

SCALING

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

	age	sex	cp	trestbps	chol	fbs	restecg	thalach	exang	oldpeak	slope	ca	thal	target
0	63	1	3	145	233	1	0	150	0	2.3	0	0	1	1
1	37	1	2	130	250	0	1	187	0	3.5	0	0	2	1
2	41	0	1	130	204	0	0	172	0	1.4	2	0	2	1
3	56	1	1	120	236	0	1	178	0	0.8	2	0	2	1
4	57	0	0	120	354	0	1	163	1	0.6	2	0	2	1
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
297	57	0	0	140	241	0	1	123	1	0.2	1	0	3	0
298	45	1	3	110	264	0	1	132	0	1.2	1	0	3	0
299	68	1	0	144	193	1	1	141	0	3.4	1	2	3	0
300	57	1	0	130	131	0	1	115	1	1.2	1	1	3	0
301	57	0	1	130	236	0	0	174	0	0.0	1	1	2	0

302 rows × 14 columns

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

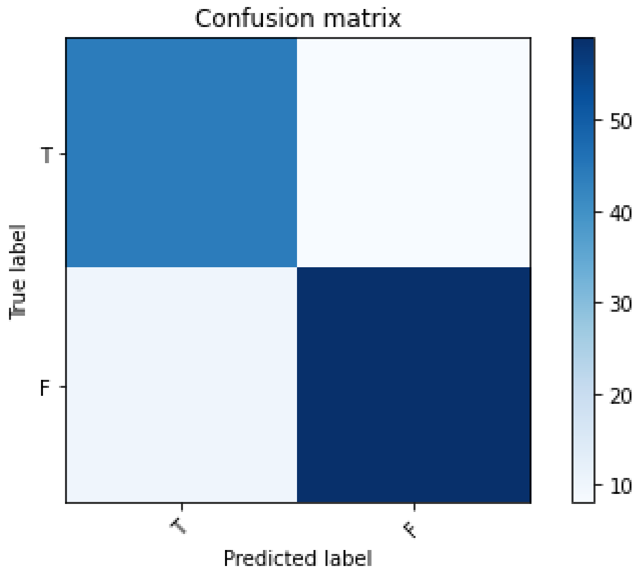
MODEL IMPLEMENTATION

MEDIAN

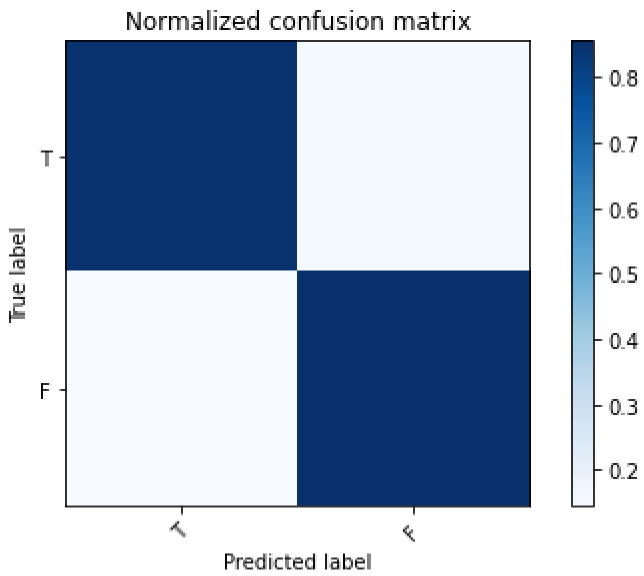
SCORE

Score: 0.8512396694214877

CONFUSION MATRIX



NORMALISED CONFUSION MATRIX



ACCURACY, PRECISION, RECALL AND F1 SCORE

Accuracy: 0.8512396694214877

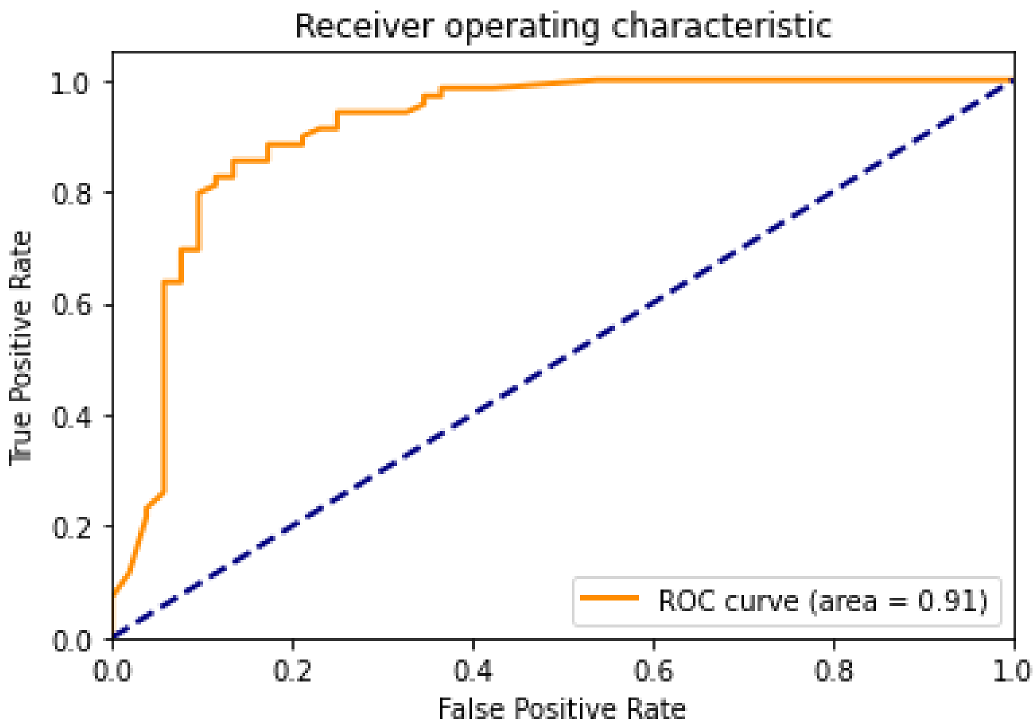
Recall: 0.855072463768116

Precision: 0.8805970149253731

F1: 0.8676470588235295

ROC (RECEIVER OPERATING CHARACTERISTIC) CURVE

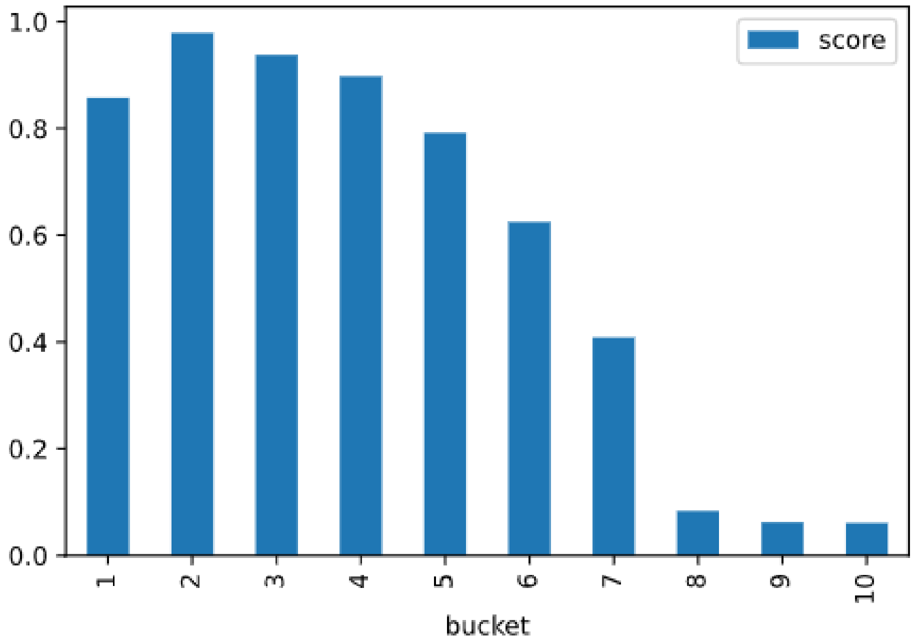
Optimal threshold value: 0.52



AUC

The AUC score of the Model is 0.9109531772575251

DECILE ANALYSIS



LOG LOSS

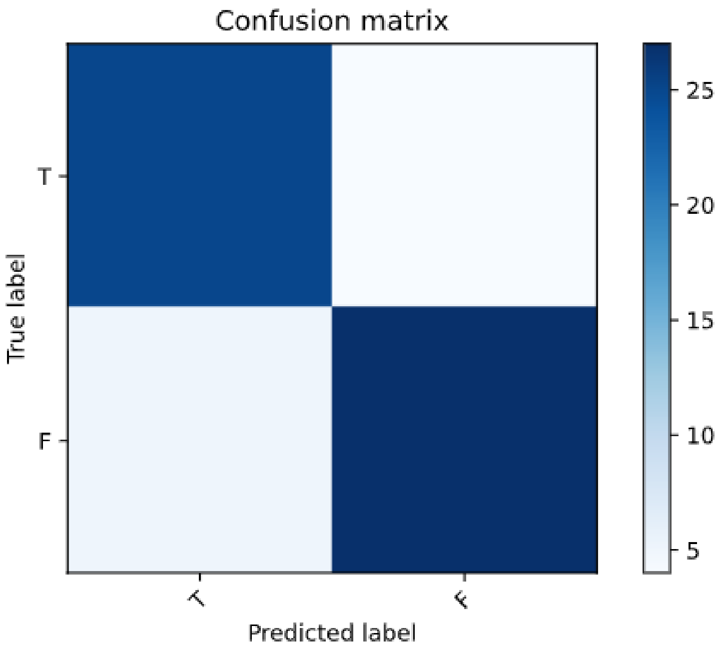
Log loss: 0.379049368470849

MEDIAN

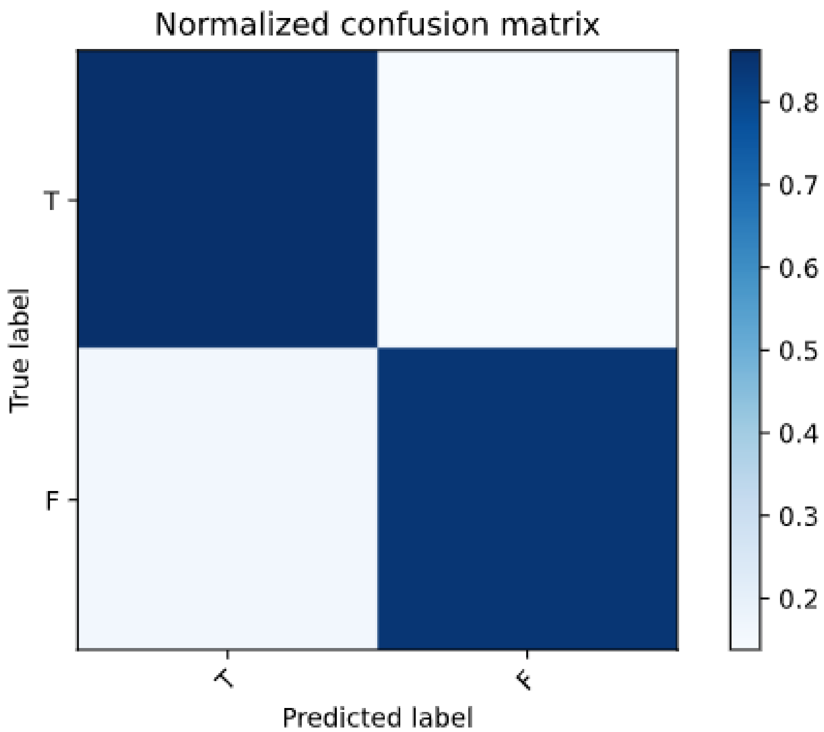
SCORE

Score: 0.8524590163934426

Confusion Matrix



NORMALISED CONFUSION MATRIX



ACCURACY, PRECISION, RECALL AND F1 SCORE

Accuracy: 0.8524590163934426

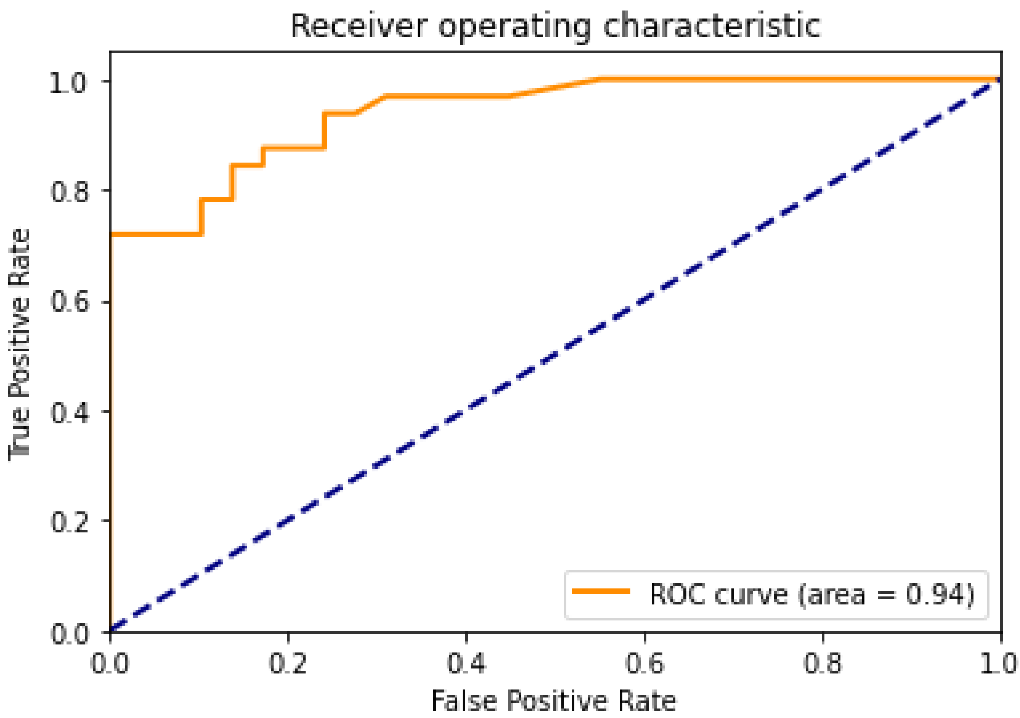
Recall: 0.84375

Precision: 0.8709677419354839

F1: 0.8571428571428571

ROC (RECEIVER OPERATING CHARACTERISTIC) CURVE

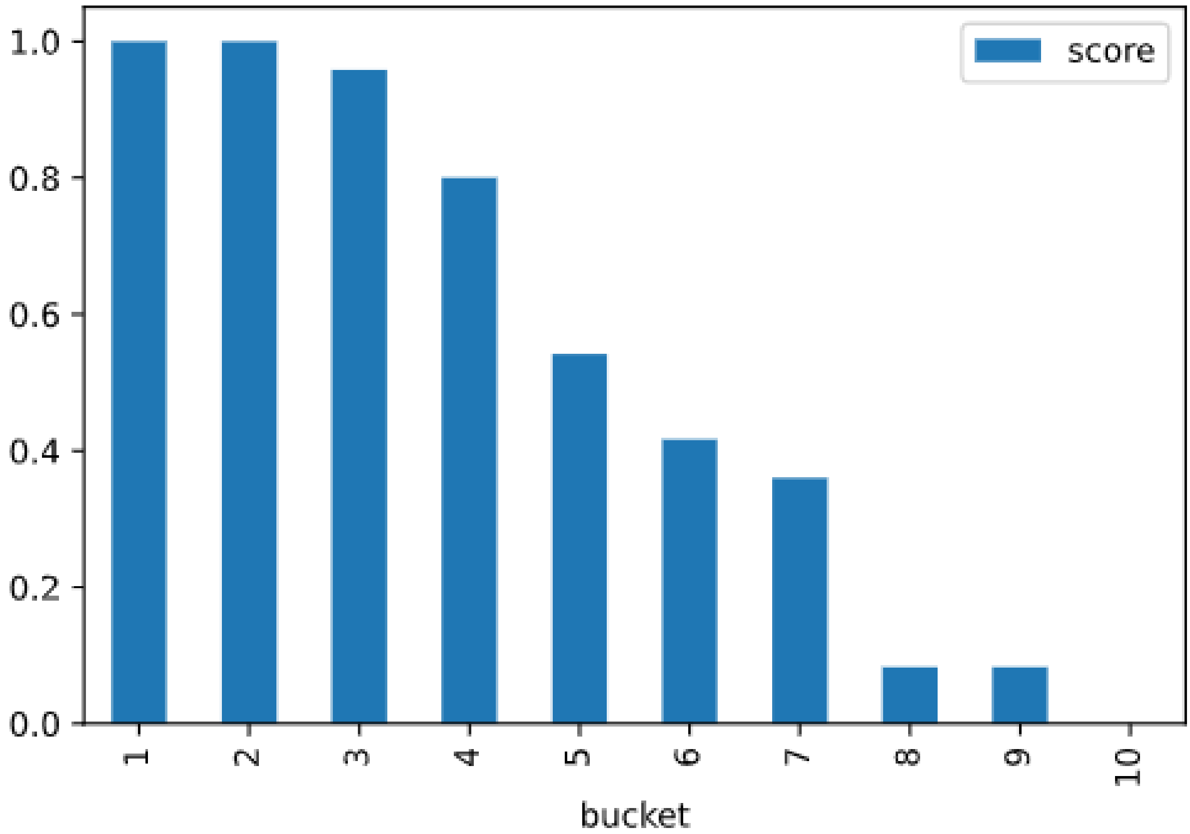
Optimal threshold value: 0.77



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

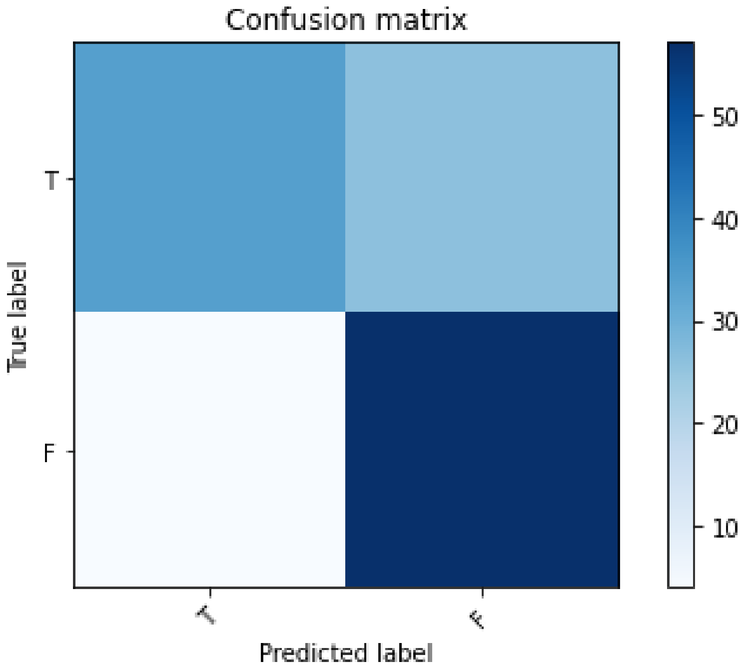
MODEL IMPLEMENTATION

MEDIAN

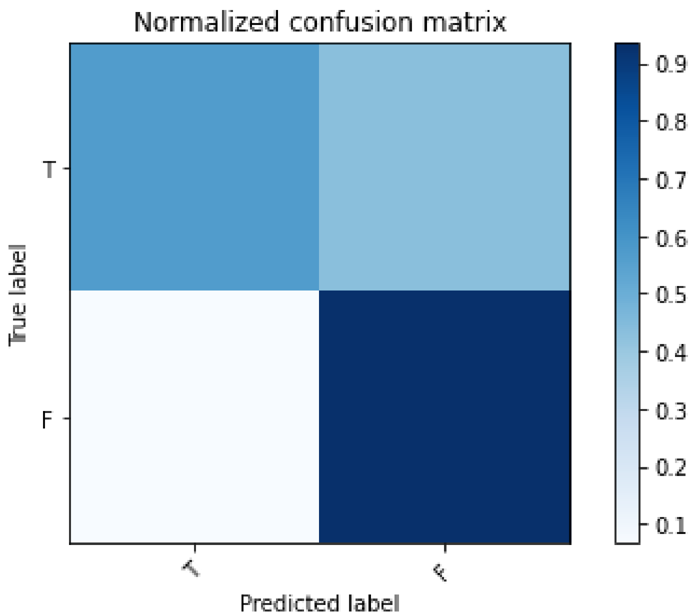
SCORE

Score: 0.743801652892562

CONFUSION MATRIX



NORMALISED CONFUSION MATRIX



ACCURACY, PRECISION, RECALL AND F1 SCORE

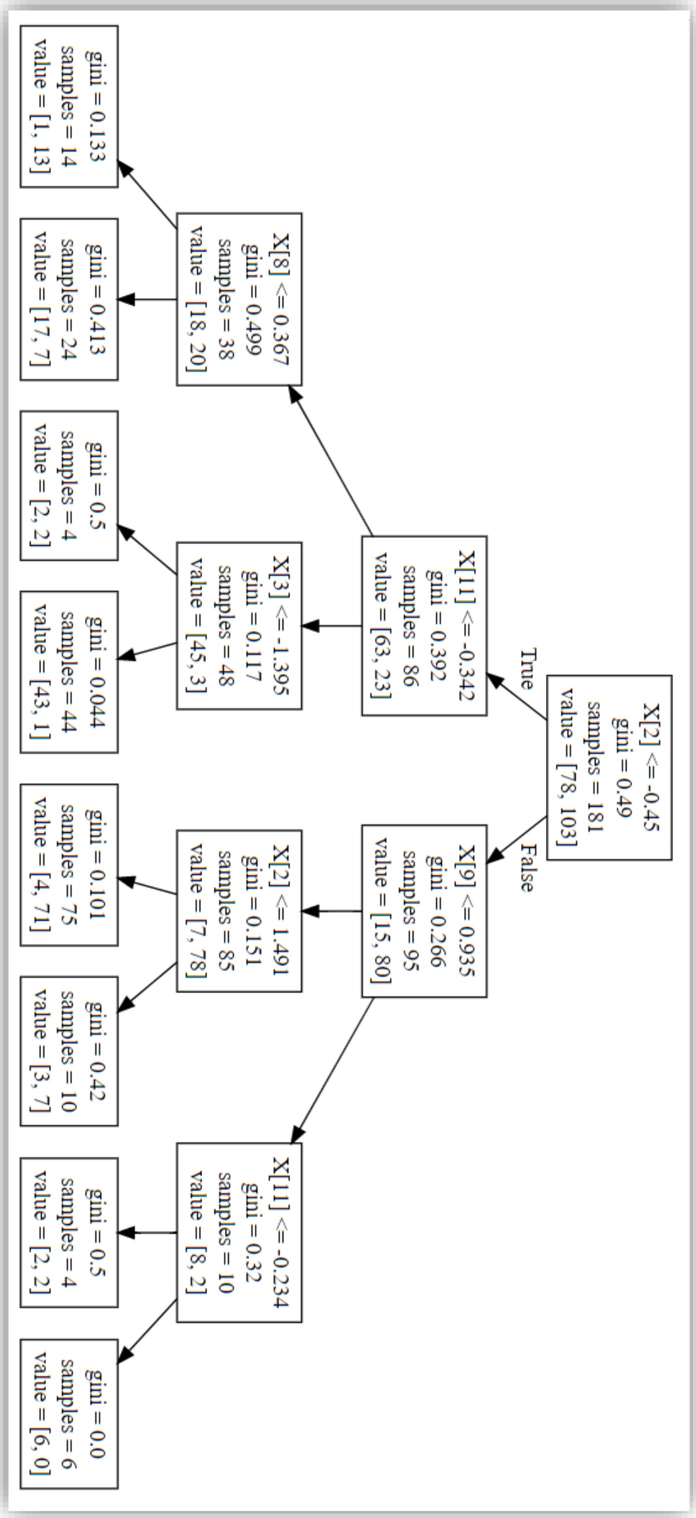
Accuracy: 0.743801652892562

Recall: 0.8852459016393442

Precision: 0.6923076923076923

F1: 0.7769784172661871

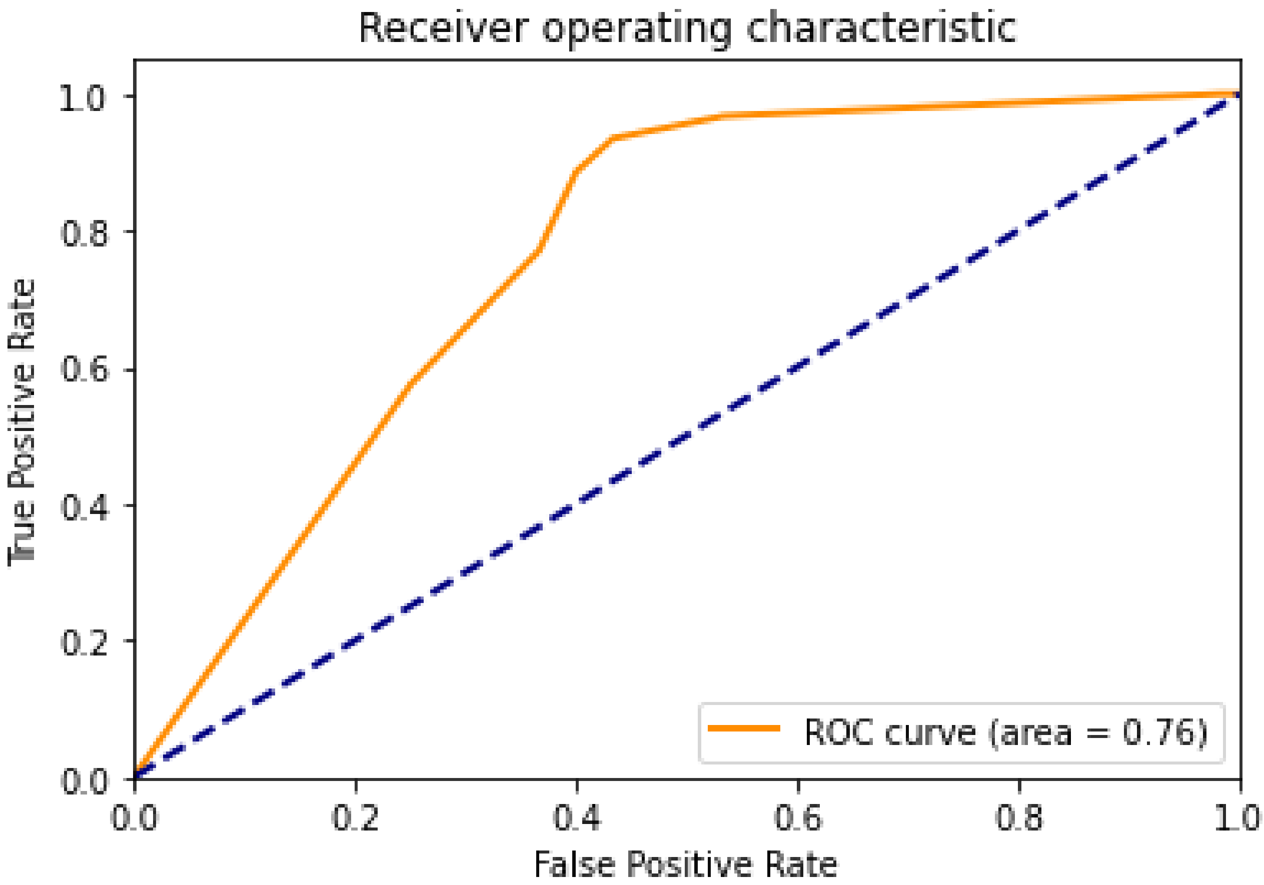
DECISION TREE



ROC (RECEIVER OPERATING CHARACTERISTIC) CURVE

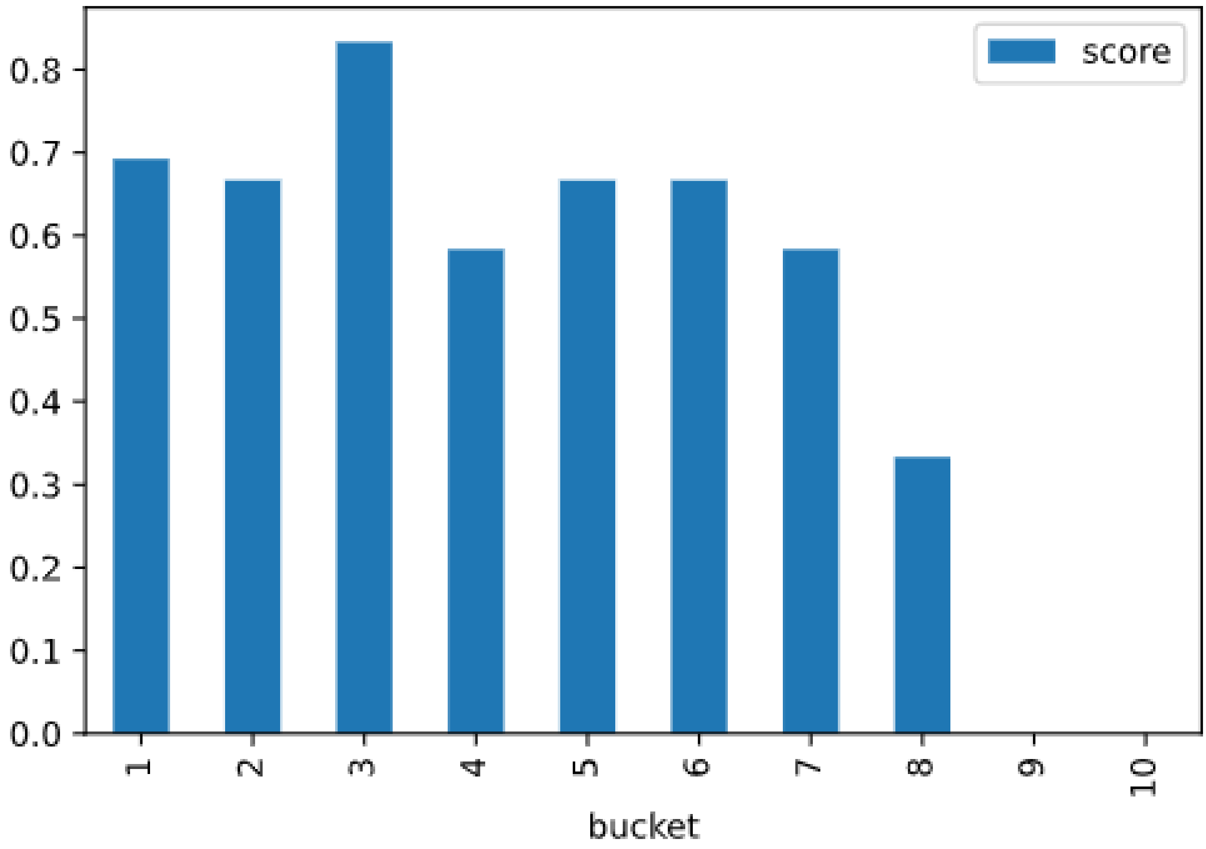
Optimal threshold value: 0.5

AUC



The AUC score of the Model is 0.7624316939890711

DECILE ANALYSIS



LOG LOSS

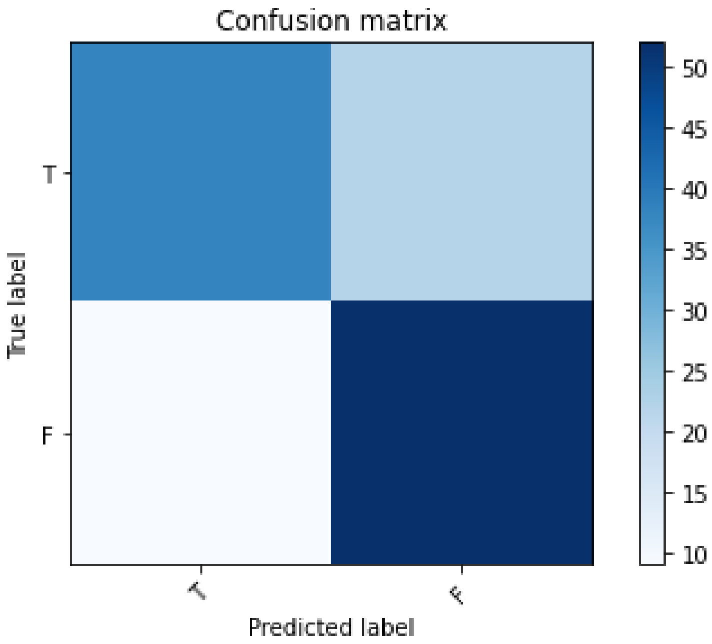
Log loss: 0.7230374691946901

MEDIAN

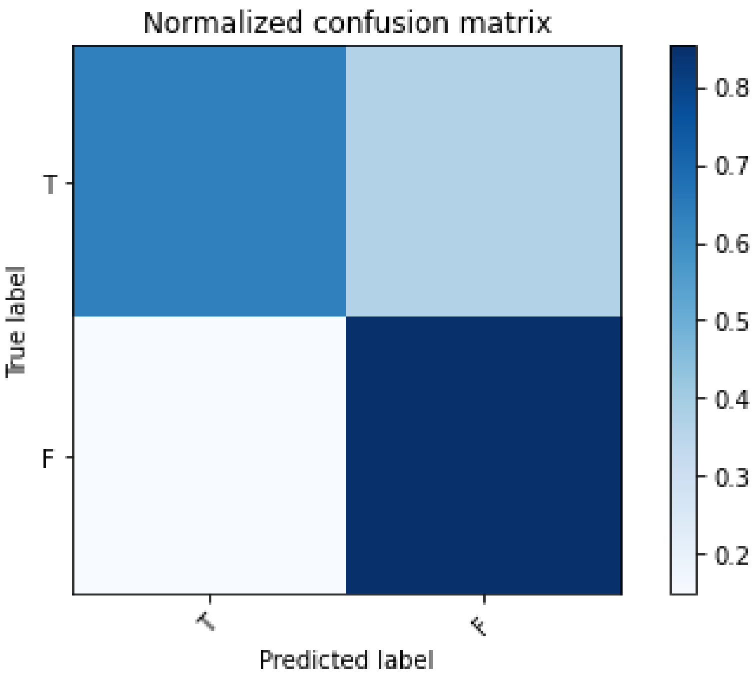
SCORE

Score: 0.743801652892562

CONFUSION MATRIX



NORMALISED CONFUSION MATRIX



ACCURACY, PRECISION, RECALL AND F1 SCORE

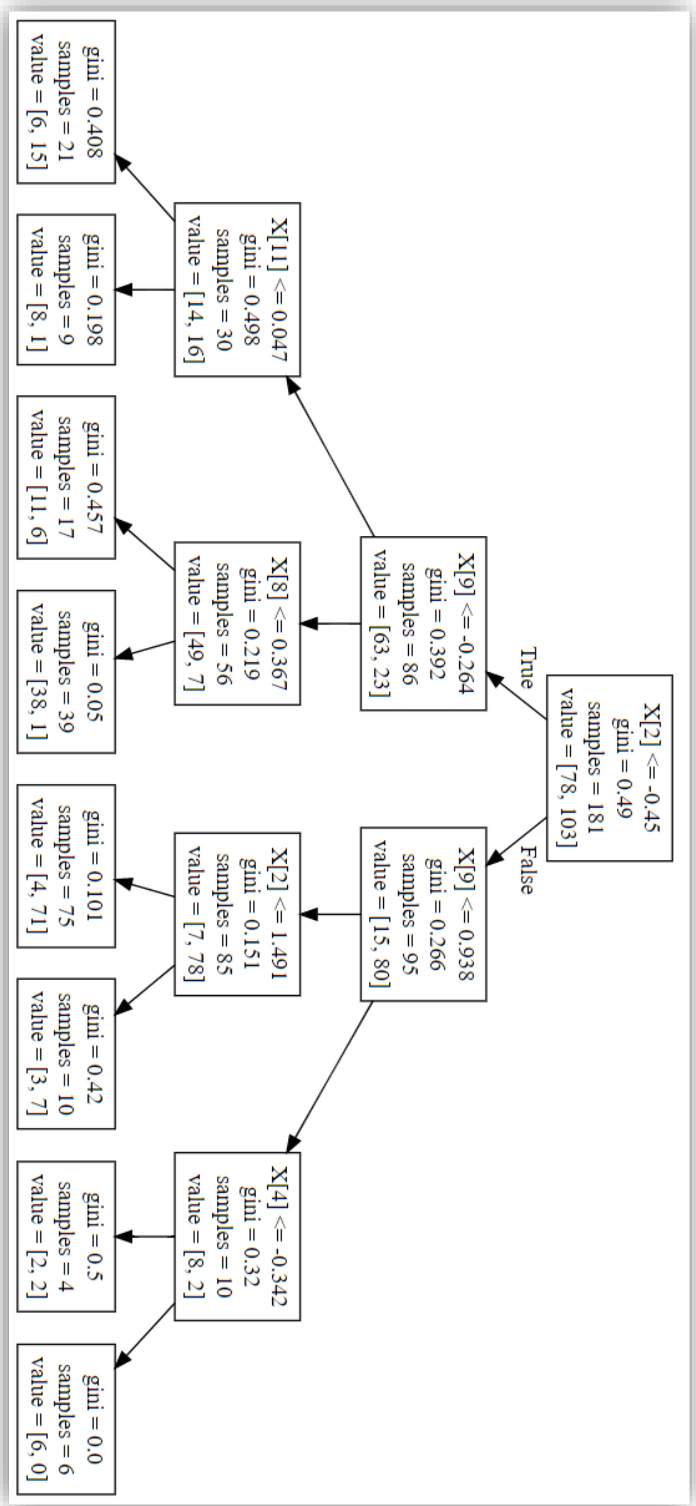
Accuracy: 0.743801652892562

Recall: 0.8524590163934426

Precision: 0.7027027027027027

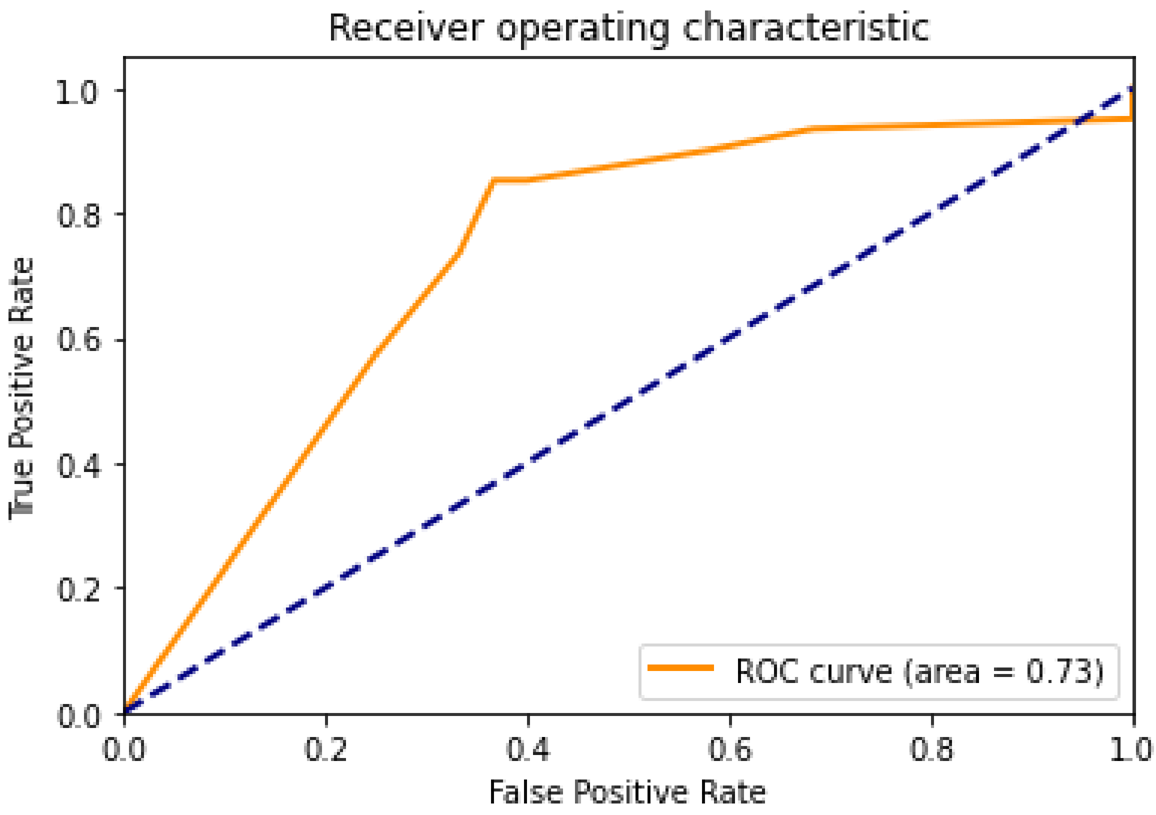
F1: 0.7703703703703704

DECISION TREE



ROC (RECEIVER OPERATING CHARACTERISTIC) CURVE

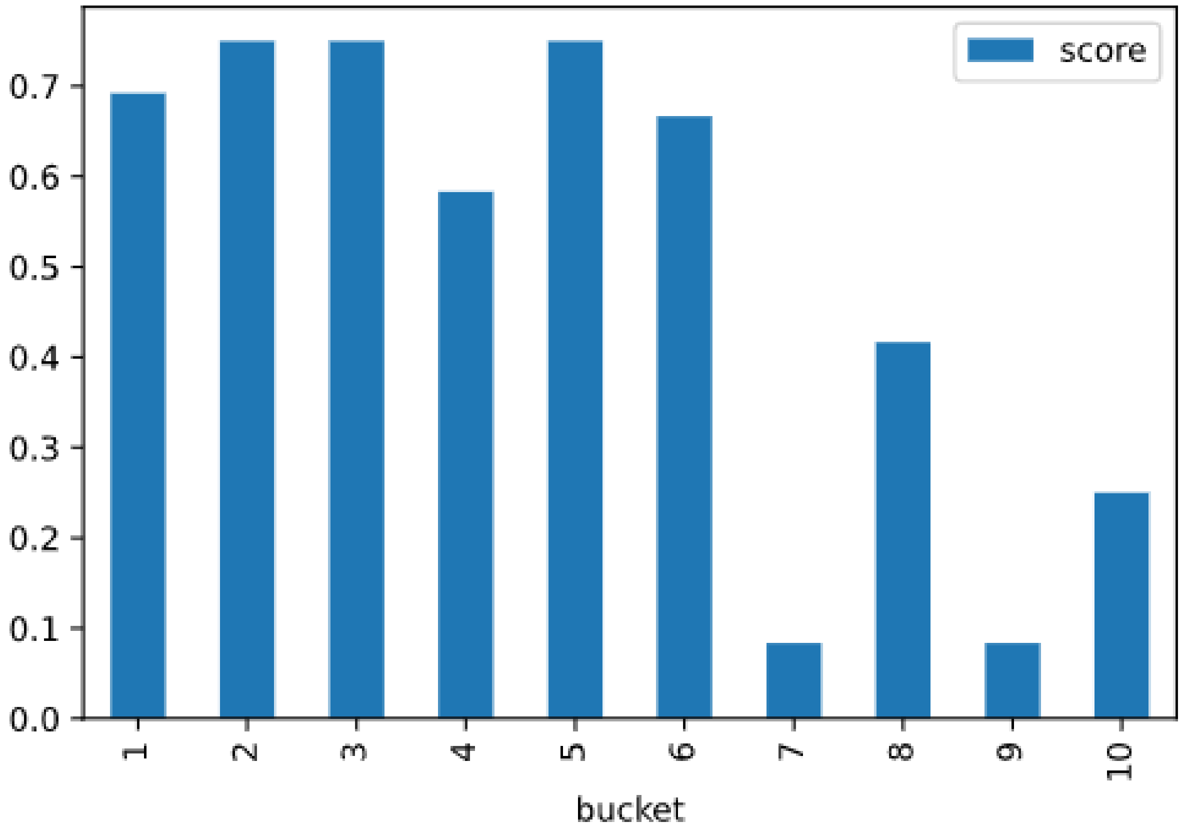
Optimal threshold value: 0.7



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:

$$P(A \mid B) = \frac{P(B \mid A)\, P(A)}{P(B)}$$

where:

- class is a particular class (e.g., male)
- data is an observation's data

$$p(\text{class} \mid \mathbf{data}) = \frac{p(\mathbf{data} \mid \text{class}) * p(\text{class})}{p(\mathbf{data})}$$

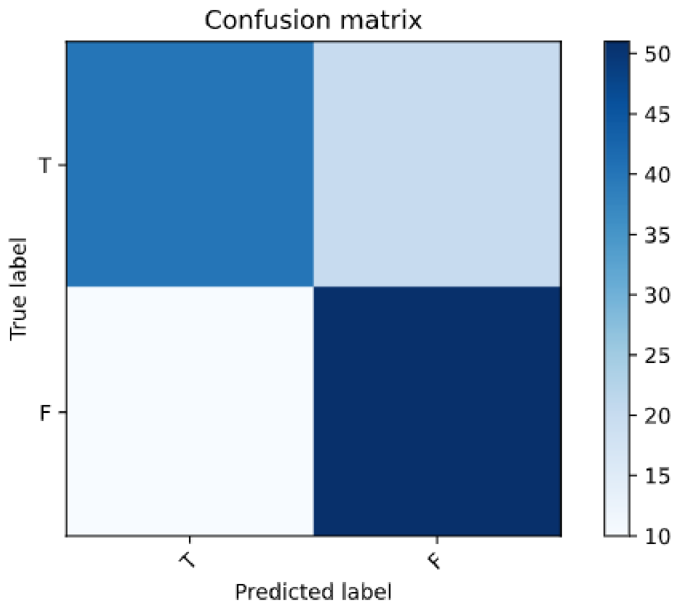
- p(class I data) is called the posterior
- p(data | class) is called the likelihood
- p(class) is called the prior
- p(data) is called the marginal probability

MEDIAN

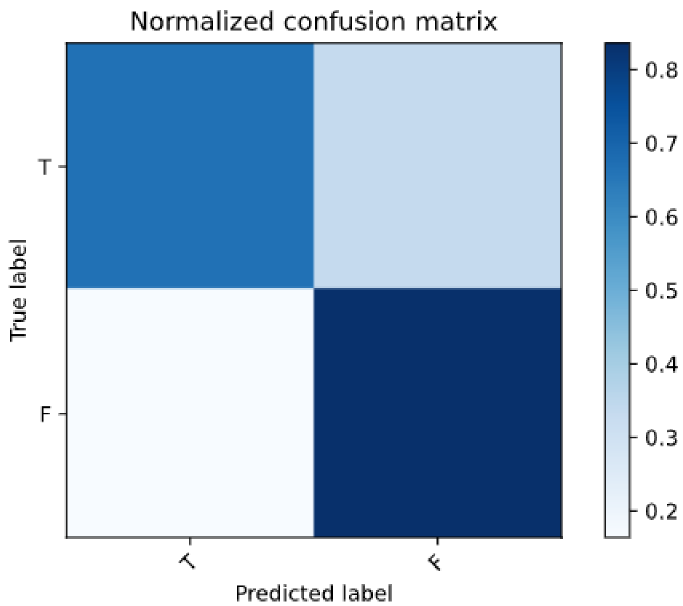
SCORE

Score: 0.7520661157024794

CONFUSION MATRIX



NORMALISED CONFUSION MATRIX



ACCURACY, PRECISION, RECALL AND F1 SCORE

Accuracy: 0.7520661157024794

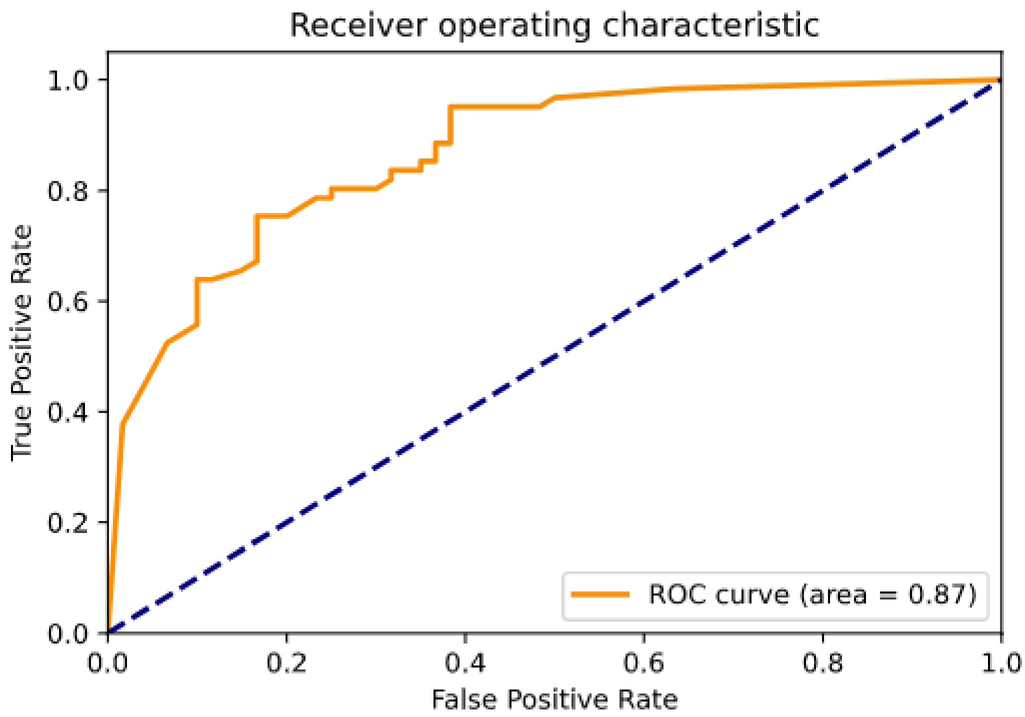
Recall: 0.8360655737704918

Precision: 0.7183098591549296

F1: 0.7727272727272727

ROC (RECEIVER OPERATING CHARACTERISTIC) CURVE

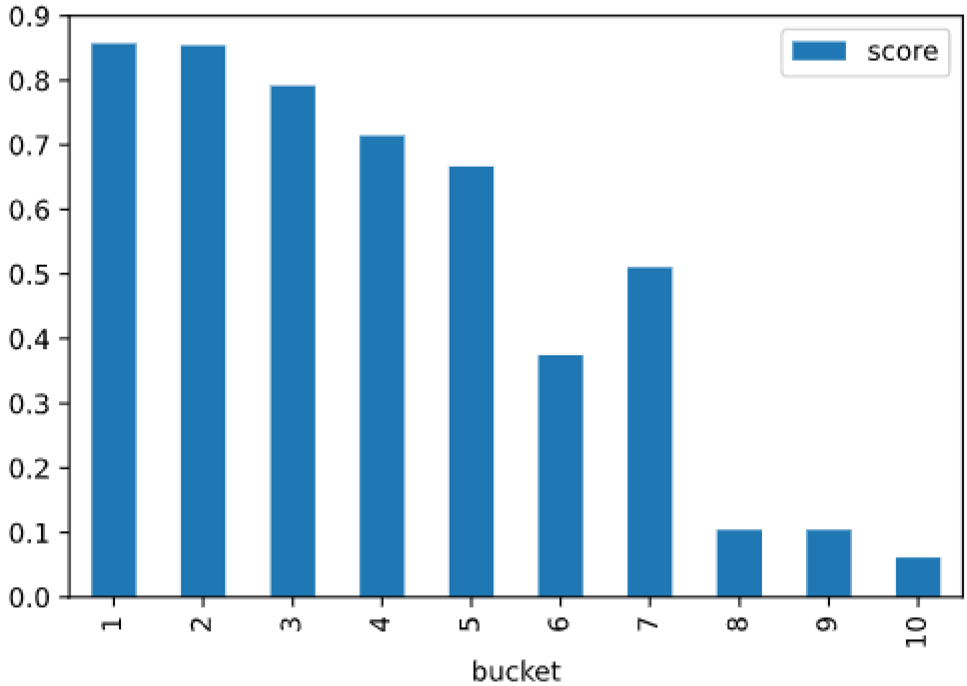
Optimal threshold value: 0.84



AUC

The AUC score of the Model is 0.8662568306010929

DECILE ANALYSIS



LOG LOSS

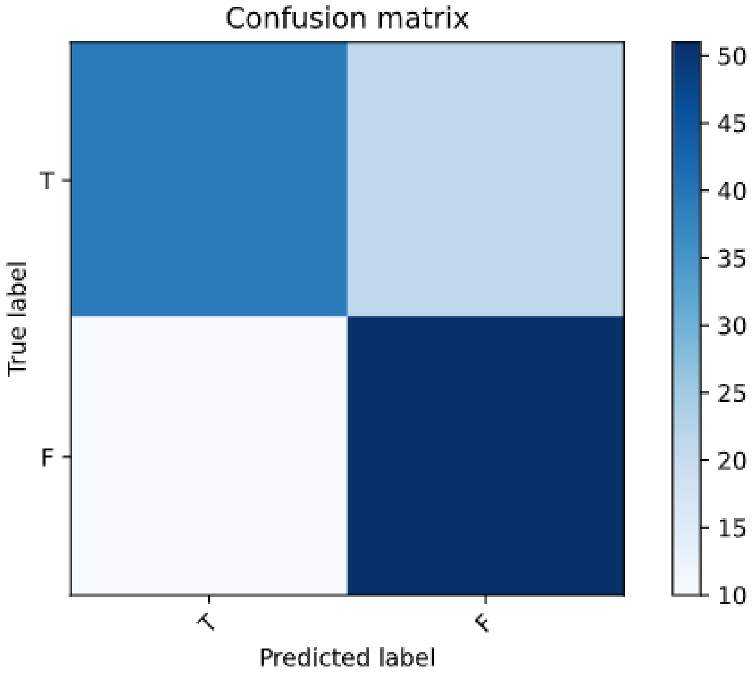
Log loss: 1.1112545612072136

MEDIAN

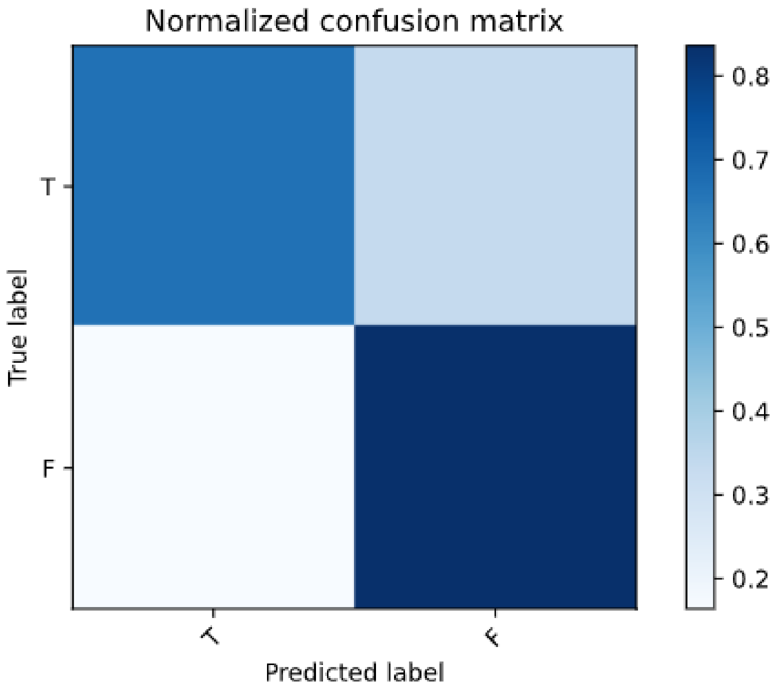
SCORE

Score: 0.743801652892562

CONFUSION MATRIX



NORMALISED CONFUSION MATRIX



ACCURACY, PRECISION, RECALL AND F1 SCORE

Accuracy: 0.743801652892562

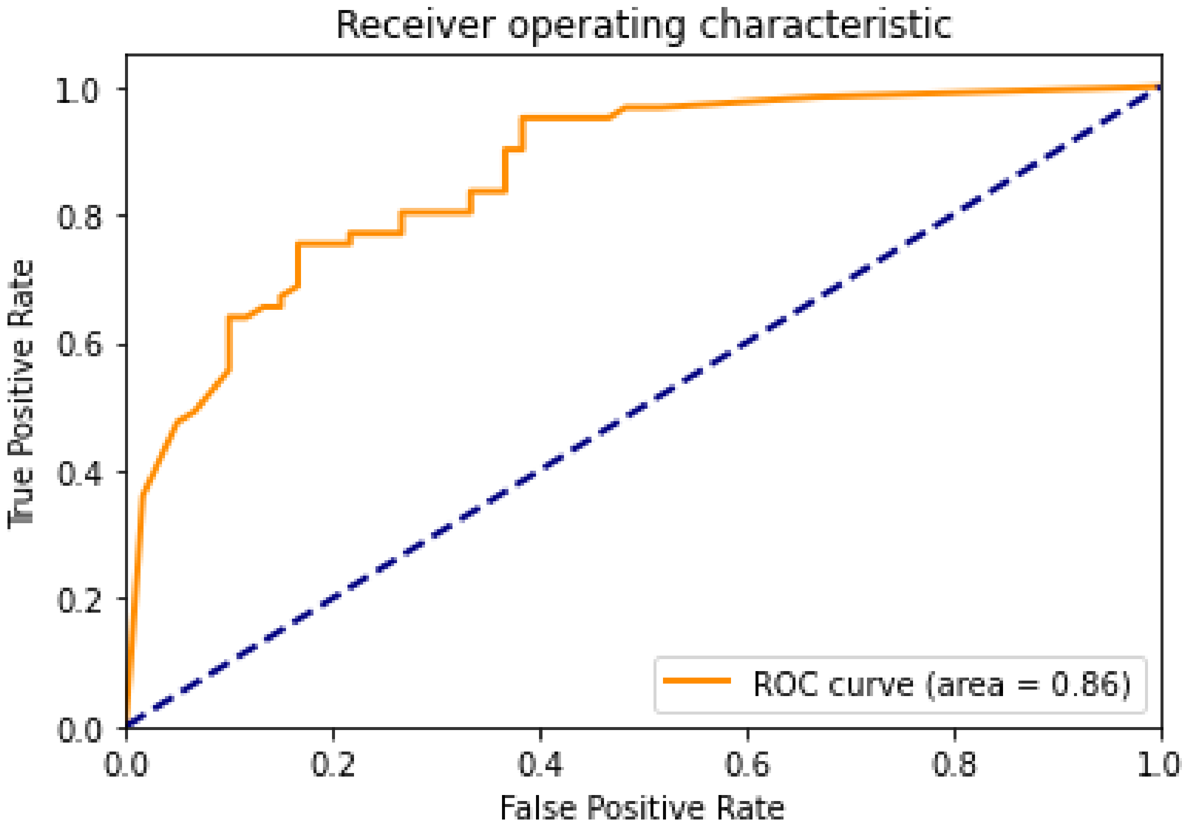
Recall: 0.8360655737704918

Precision: 0.7083333333333334

F1: 0.7669172932330828

ROC (RECEIVER OPERATING CHARACTERISTIC) CURVE

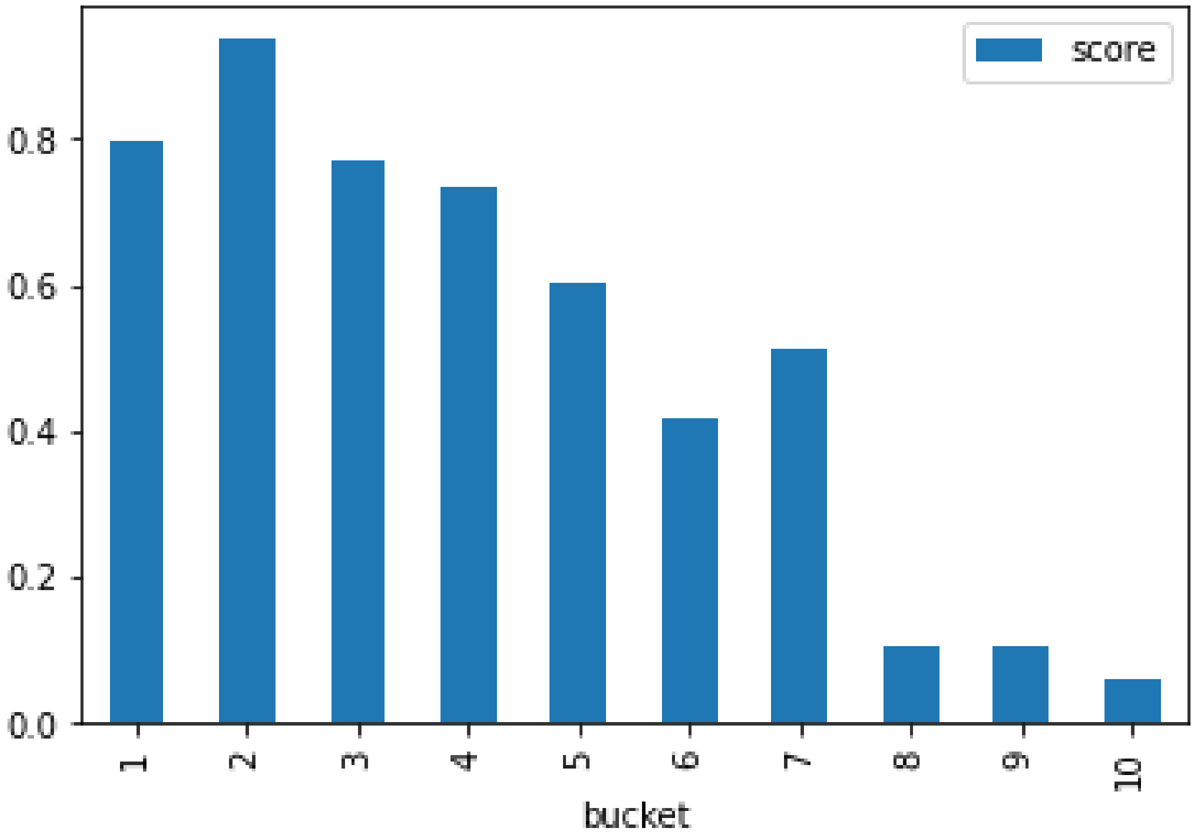
Optimal threshold value: 0.82



AUC

The AUC score of the Model is 0.8632513661202186

DECILE ANALYSIS



LOG LOSS

Log loss: 1.1106203966238264

KNN

In statistics, the *k*-nearest neighbours algorithm (*k*-NN) is a non-parametric method proposed by Thomas Cover used for classification and regression.[1] In both cases, the input consists of the *k* closest training examples in the 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, 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.

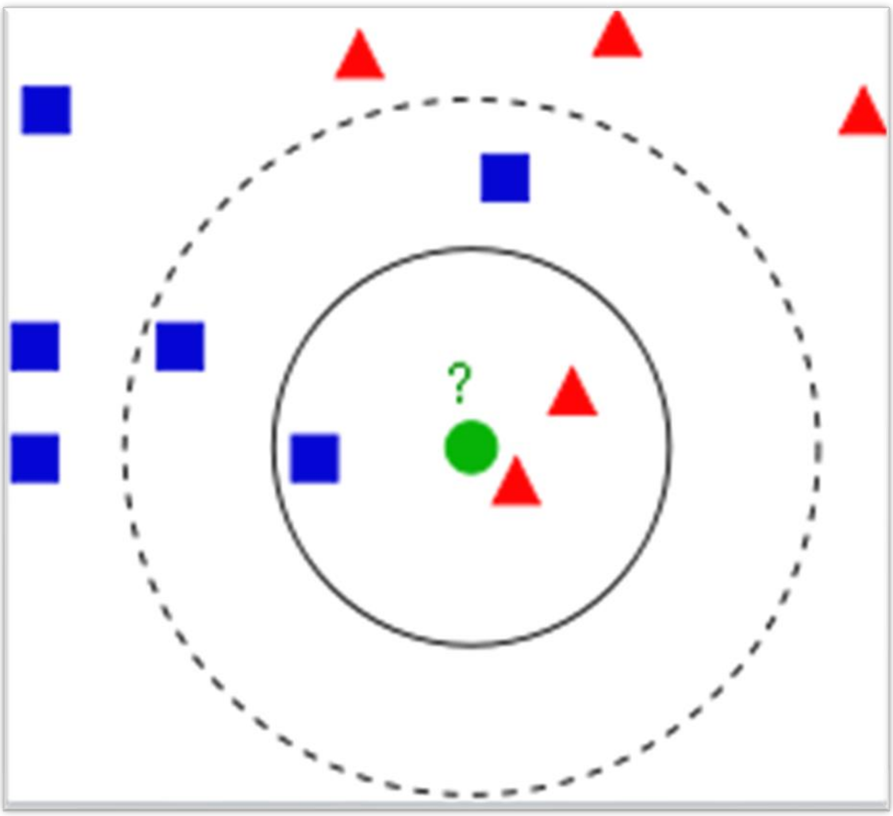
PARAMETER SELECTION AND VALUE OF K

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

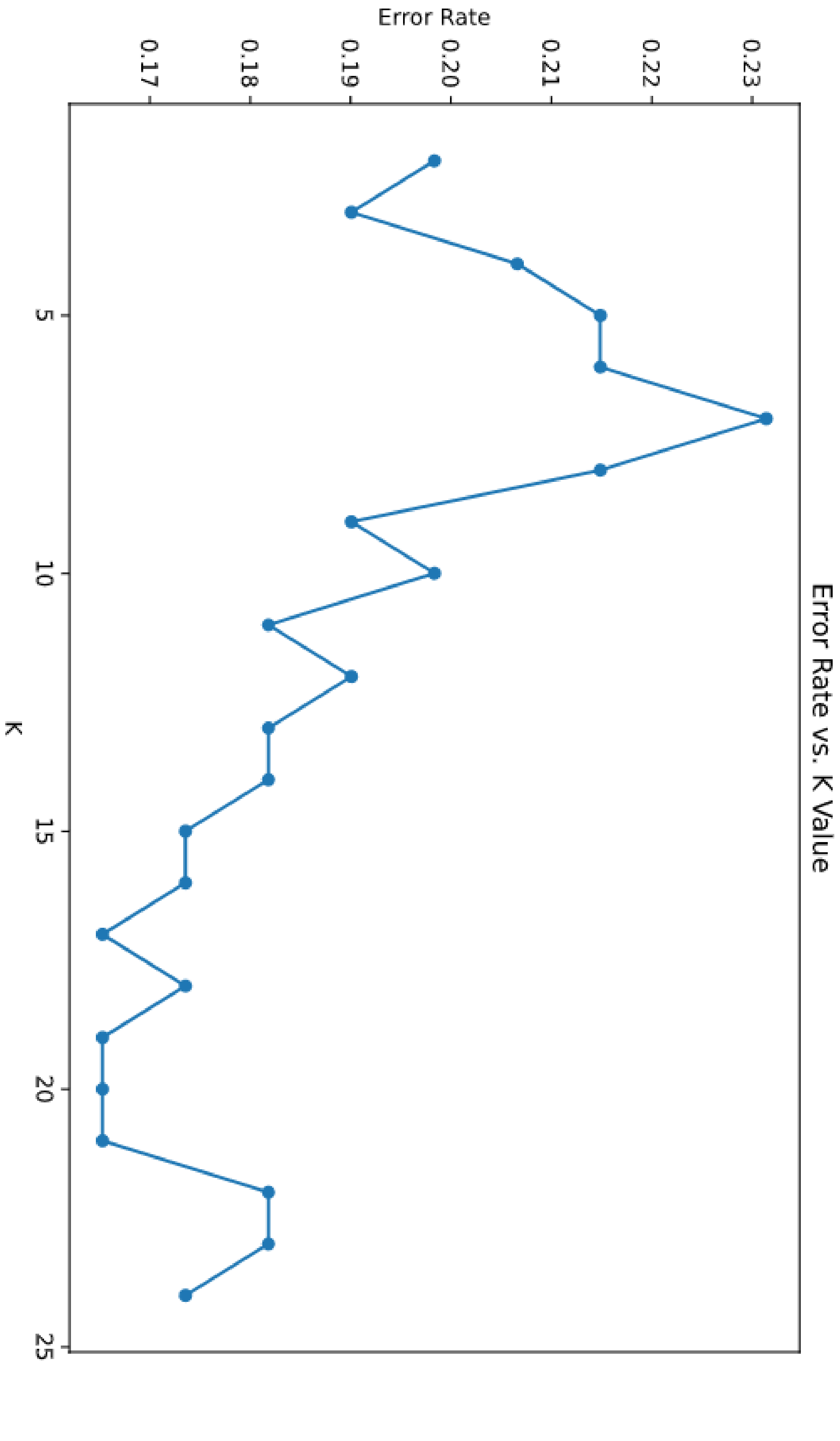
EXAMPLE

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)



MEDIAN

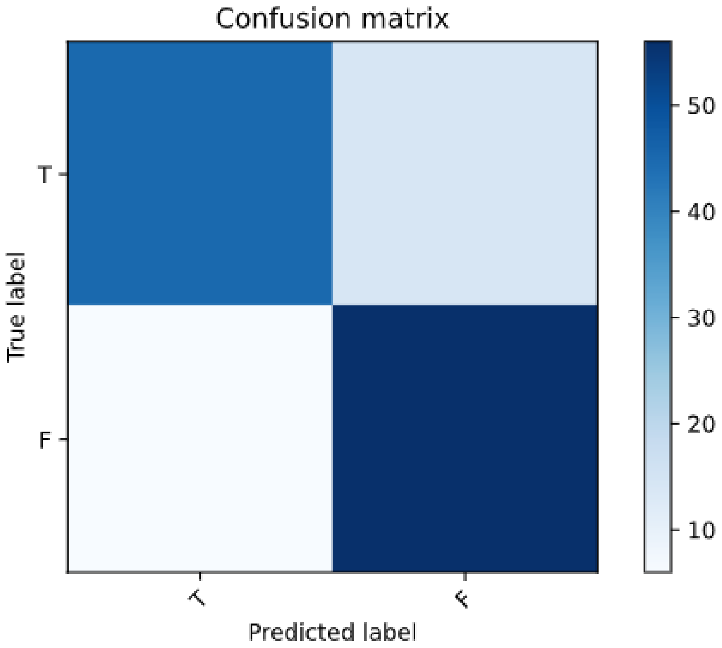
K VALUE



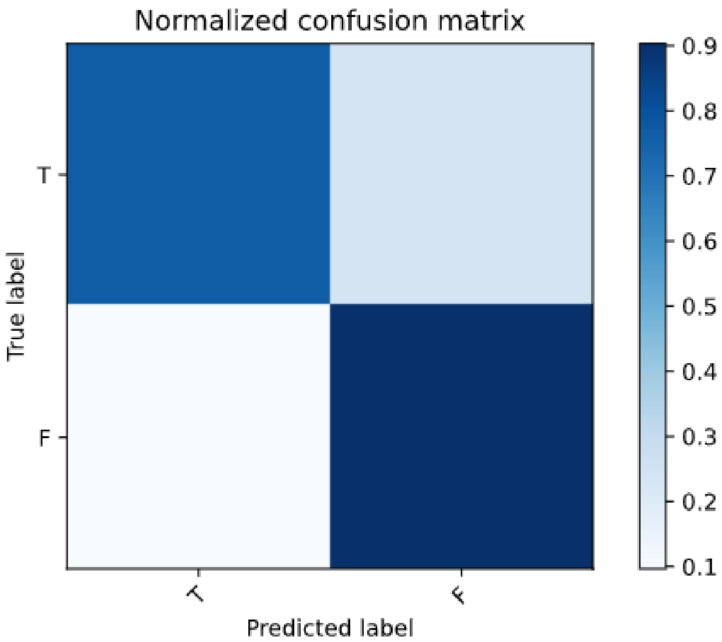
SCORE

Score: 0.8287292817679558

CONFUSION MATRIX



NORMALISED CONFUSION MATRIX



ACCURACY, PRECISION, RECALL AND F1 SCORE

Accuracy: 0.8347107438016529

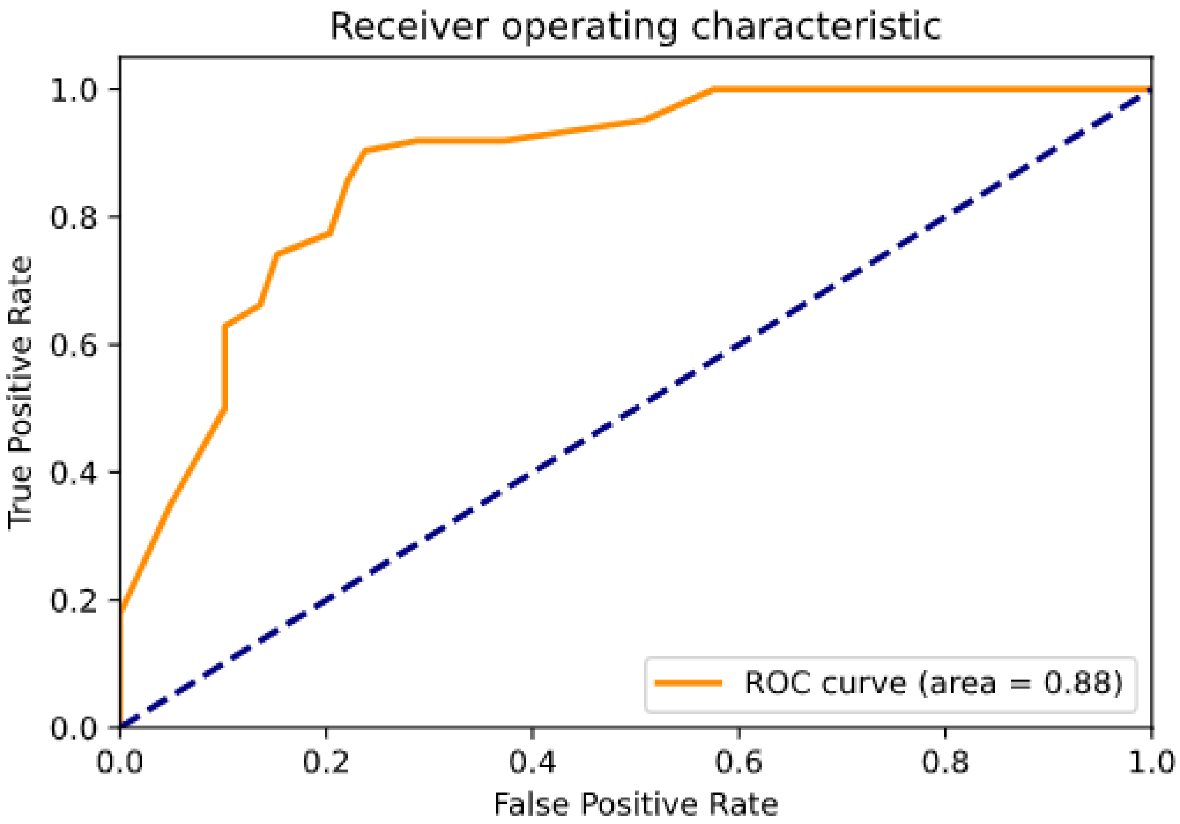
Recall: 0.9032258064516129

Precision: 0.8

F1: 0 .8484848484848486

ROC (RECEIVER OPERATING CHARACTERISTIC) CURVE

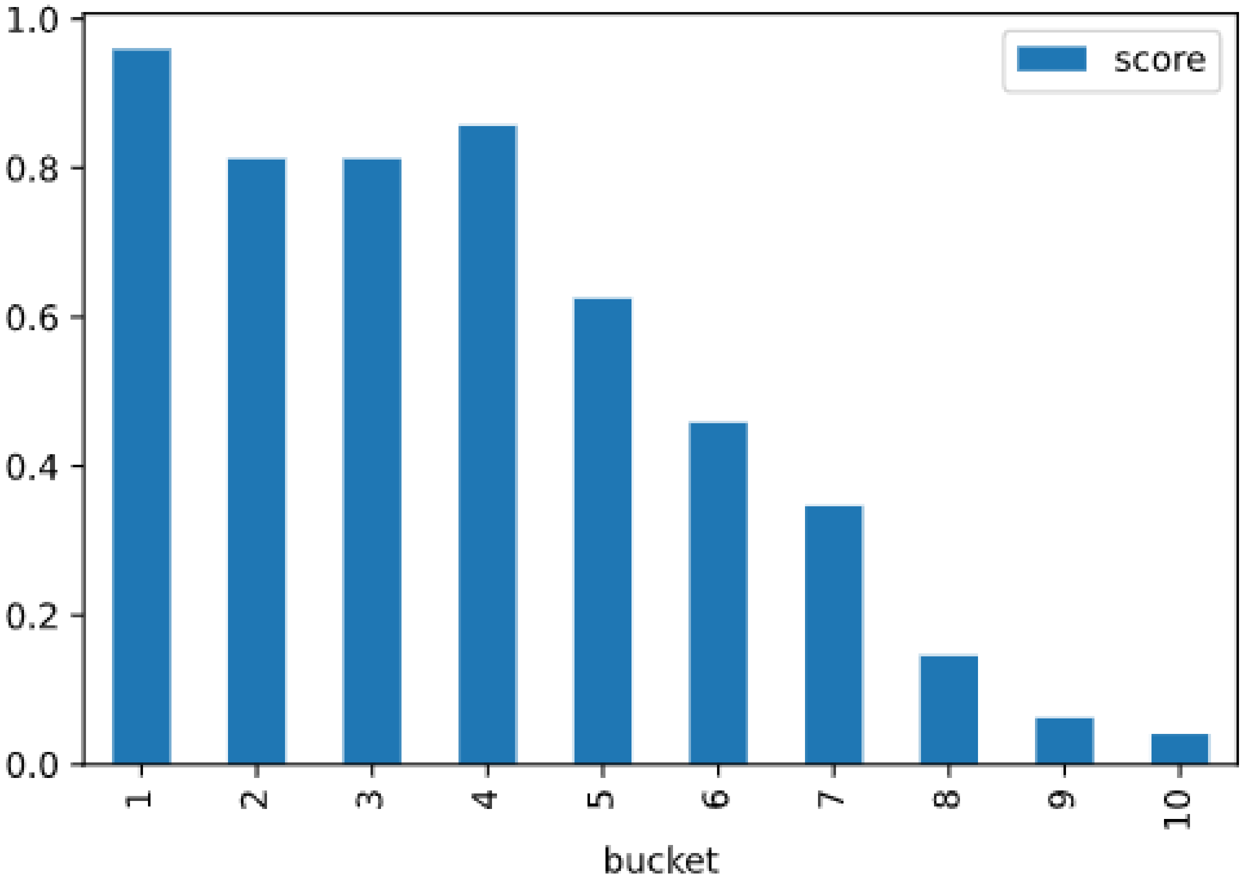
Optimal threshold value: 0.5294117647058824



AUC

The AUC score of the Model is 0.877255330781848

DECILE ANALYSIS

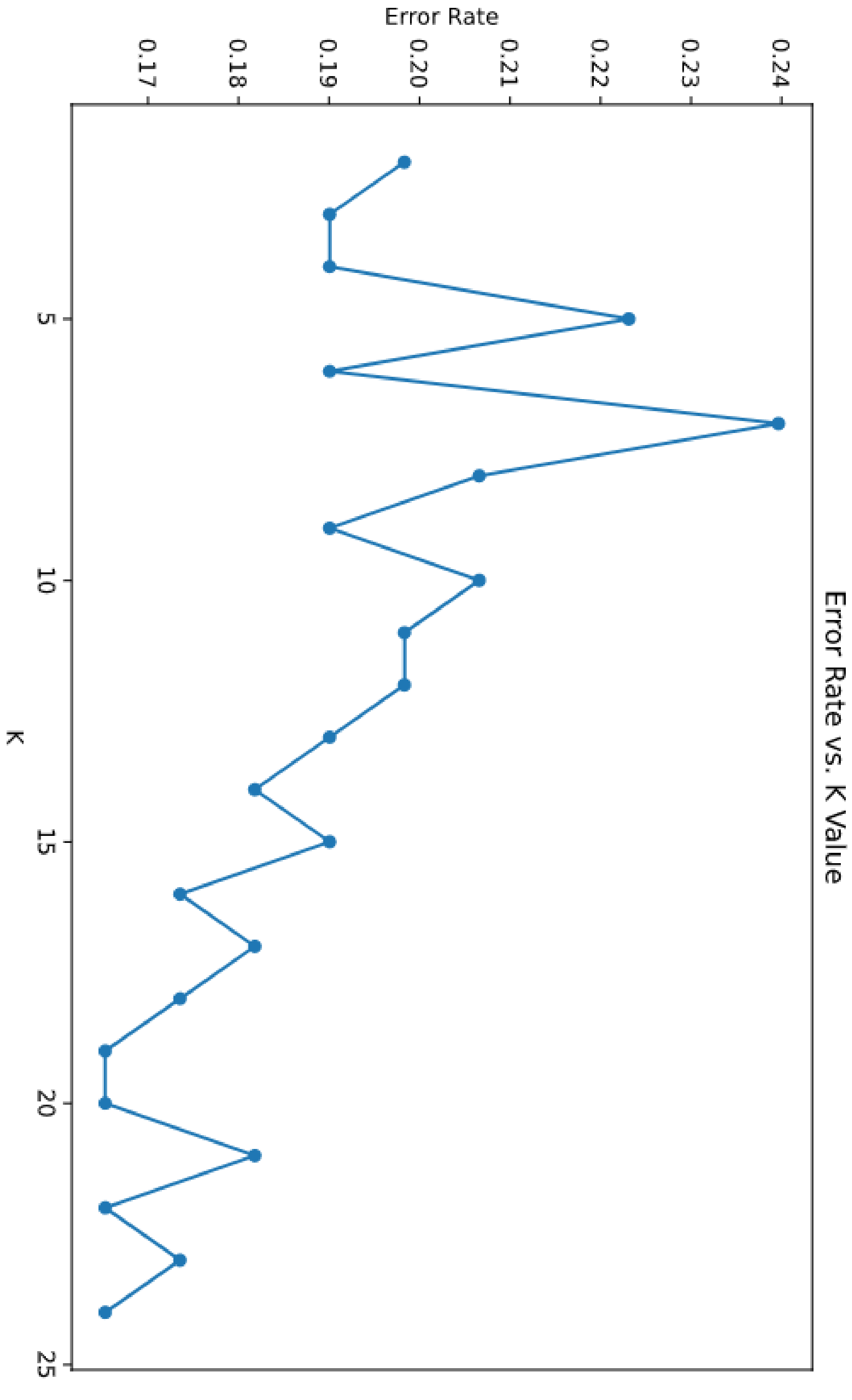


LOG LOSS

Log loss: 0.42653563898605484

MEDIAN

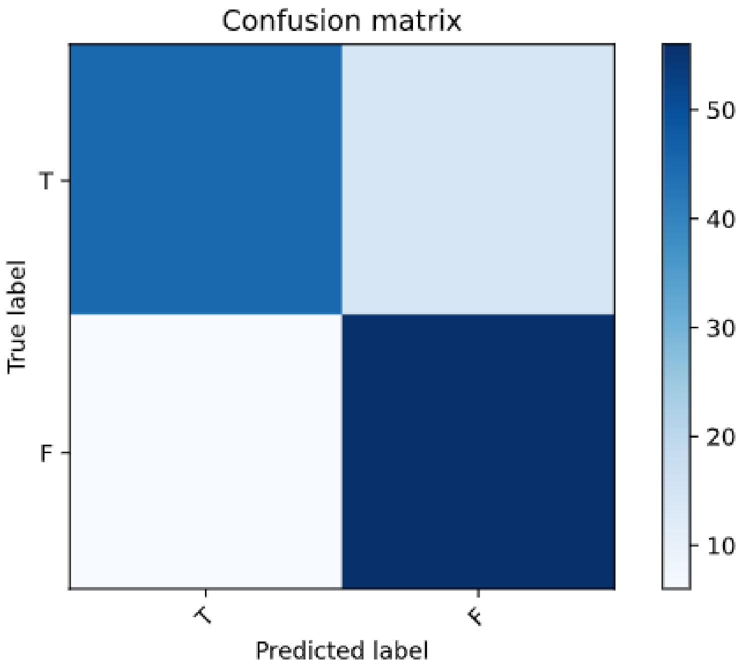
K VALUE



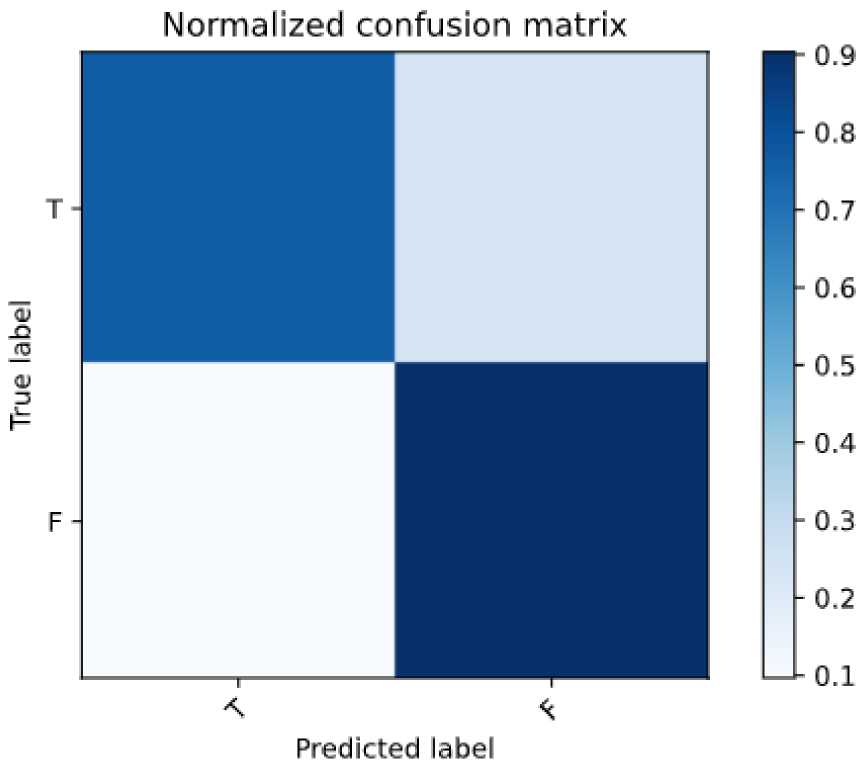
SCORE

Score: 0.8397790055248618

CONFUSION MATRIX



NORMALISED CONFUSION MATRIX



ACCURACY, PRECISION, RECALL AND F1 SCORE

Accuracy: 0.8347107438016529

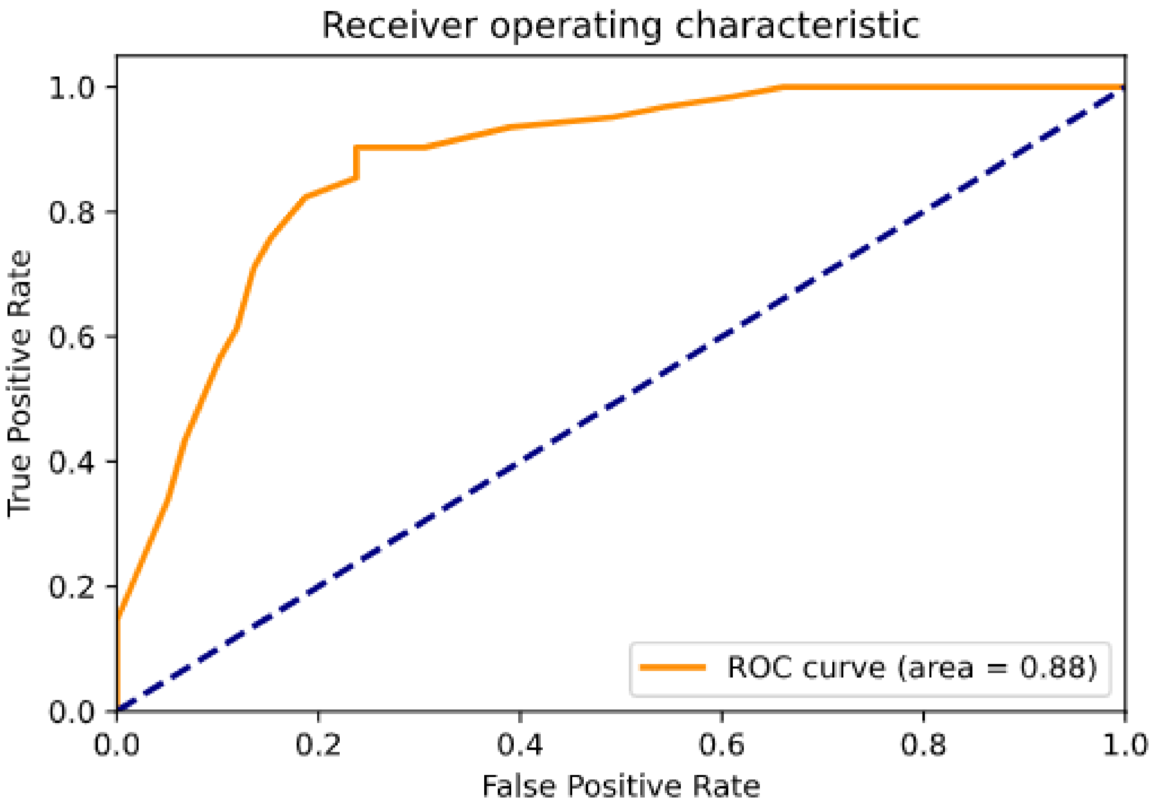
Recall: 0.9032258064516129

Precision: 0.8

F1: 0.8484848484848486

ROC (RECEIVER OPERATING CHARACTERISTIC) CURVE

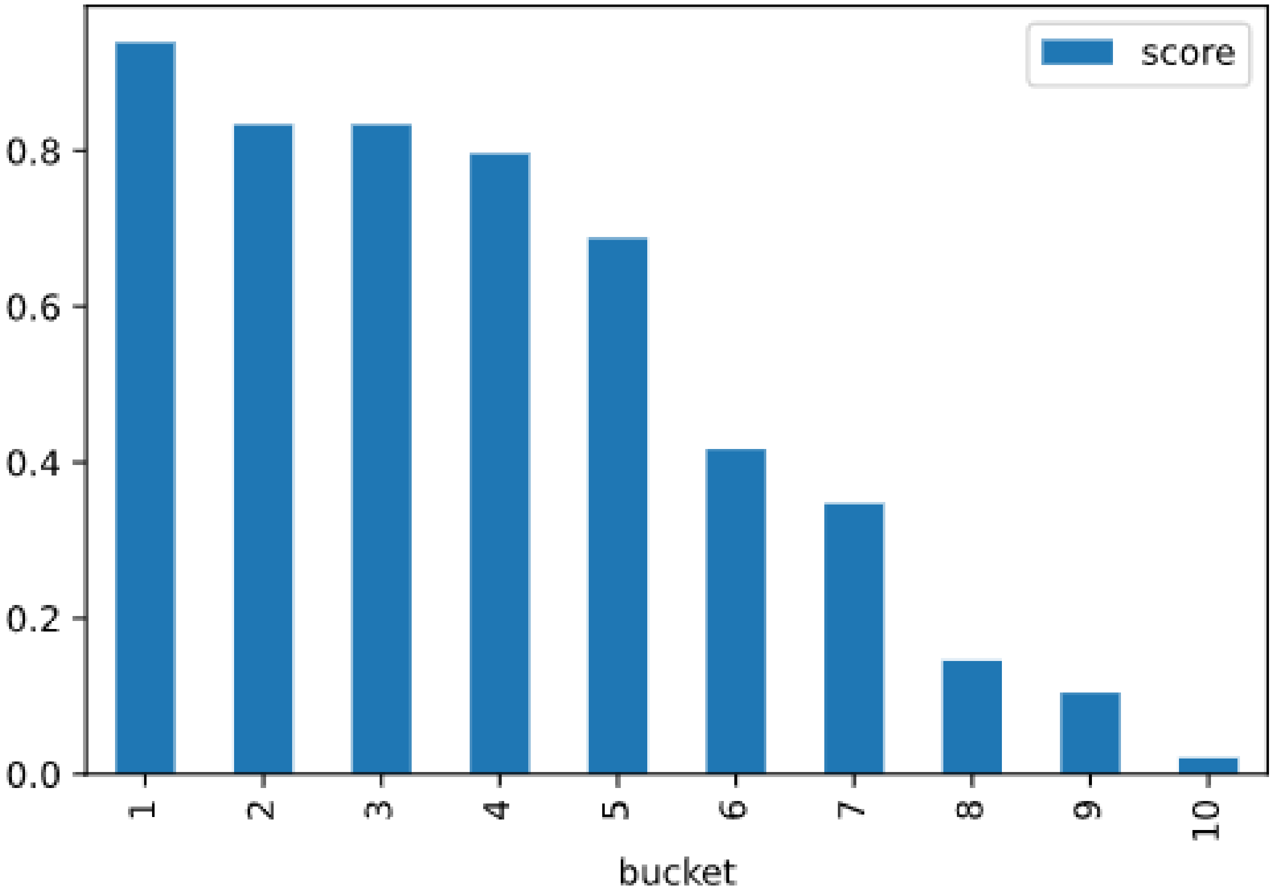
Optimal threshold value: 0.5263157894736842



AUC

The AUC score of the Model is 0.8783488244942592

DECILE ANALYSIS



LOG LOSS

Log loss: 0.4490046250780085

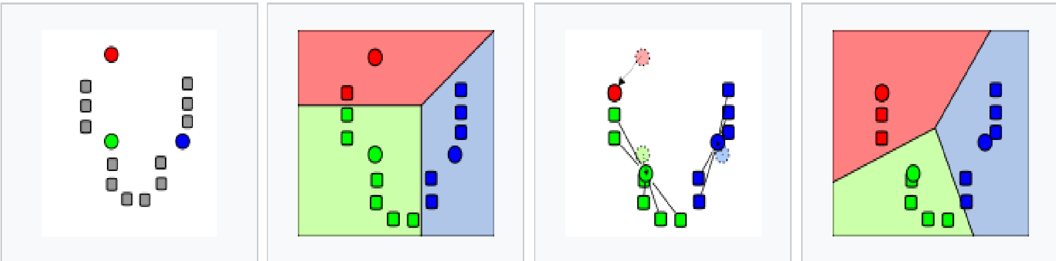
K-MEDIANS

k-Medians clustering is a method of vector quantization, originally from signal processing, that aims to partition *n* observations into *k* clusters in which each observation belongs to the cluster with the nearest Median (cluster centers or cluster centroid), serving as a prototype of the cluster. This results in a partitioning of the data space into voronoi cells. *k*-Medians clustering minimizes within-cluster variances (squared euclidean distances), but not regular euclidean distances, which would be the more difficult weber problem: the Median optimizes squared errors, whereas only the geometric Median minimizes euclidean distances. For instance, better euclidean solutions can be found using k-Medians and 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.

Demonstration of the standard algorithm



1. *k* initial "means" (in this case *k*=3) are randomly generated within the data domain (shown in color).

2. *k* clusters are created by associating every observation with the nearest mean. The partitions here represent the Voronoi diagram generated by the means.

3. The centroid of each of the *k* clusters becomes the new mean.

4. Steps 2 and 3 are repeated until convergence has been reached.

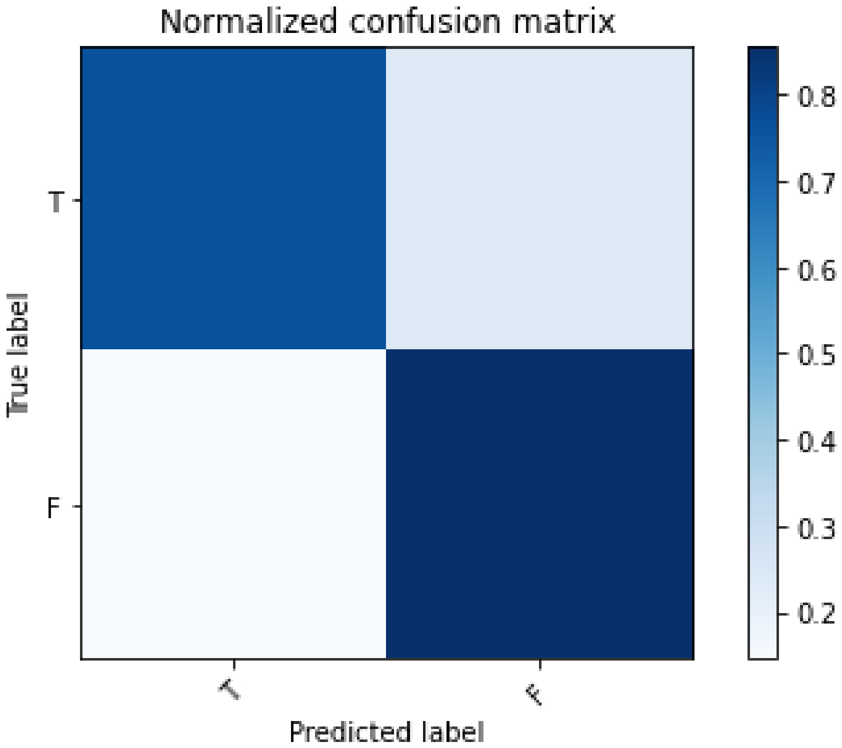
MODEL IMPLEMENTATION

MEDIAN

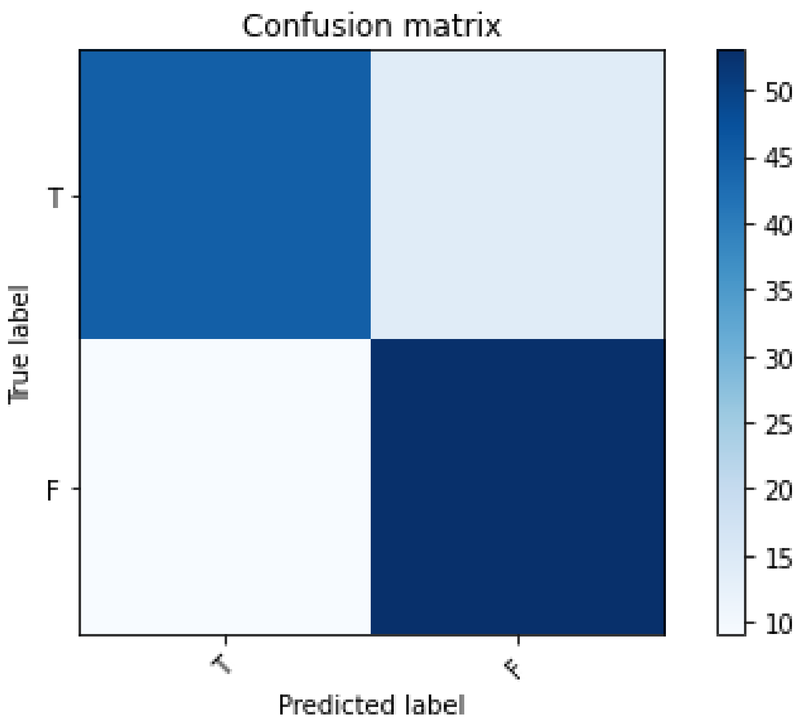
SCORE

Score: -1360.6191878608547

CONFUSION MATRIX



NORMALISED CONFUSION MATRIX



ACCURACY, PRECISION, RECALL AND F1 SCORE

Accuracy: 0.8099173553719008

Recall: 0.8548387096774194

Precision: 0.7910447761194029

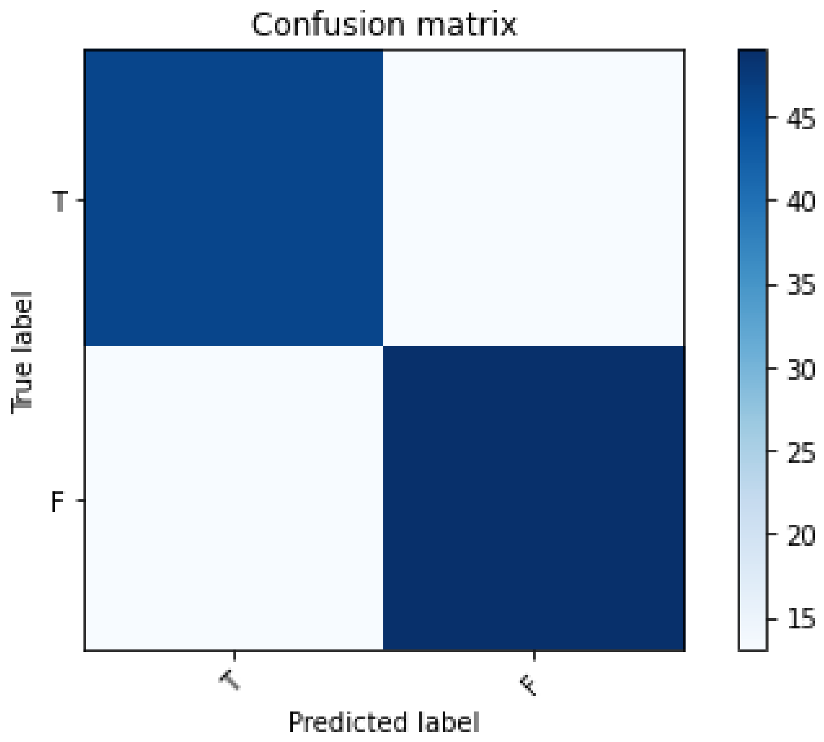
F1: 0.8217054263565892

MEDIAN

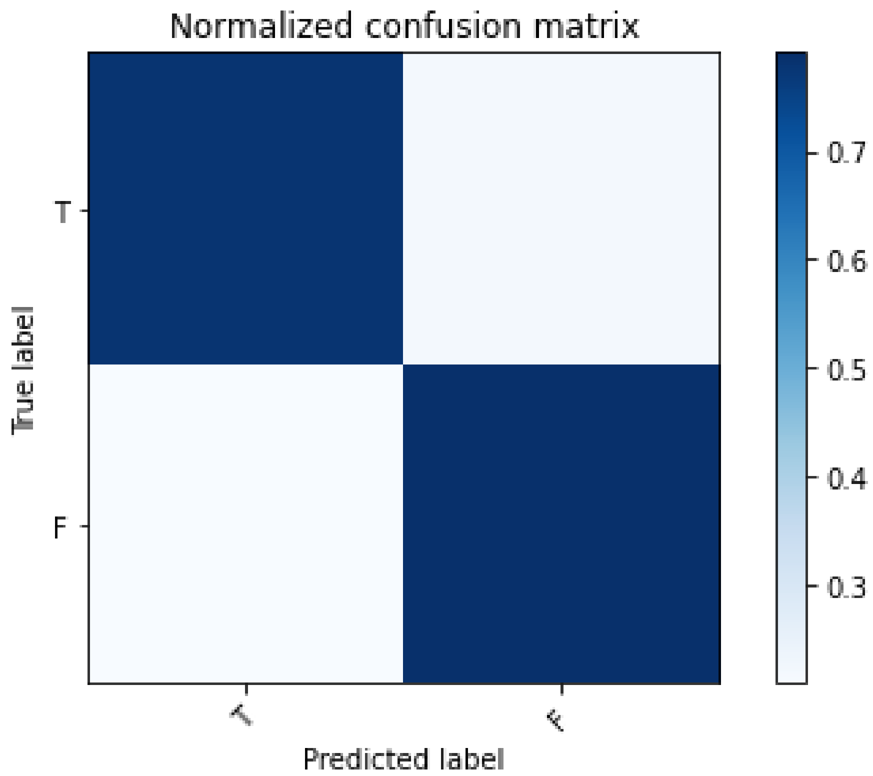
SCORE

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

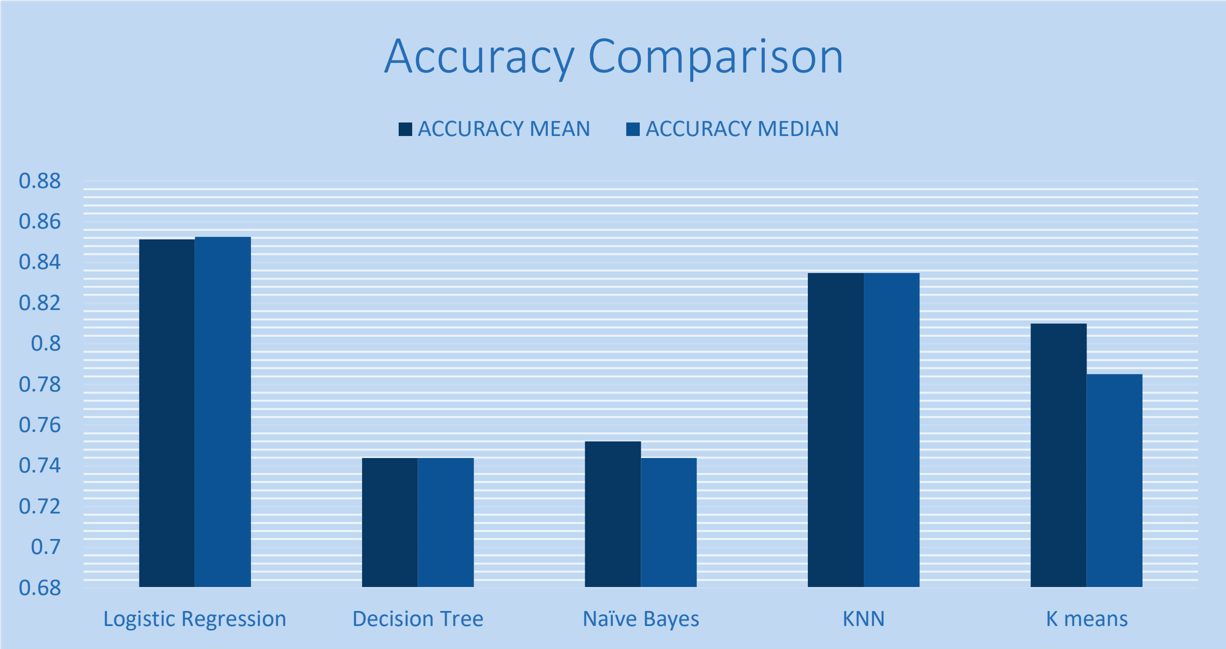
MODEL	MEDIAN	MEDIAN
Logistic Regression	0.8805970149253731	0.8709677419354839
Decision Tree	0.6923076923076923	0.7027027027027027
Naïve Bayes	0.7183098591549296	0.7083333333333334
KNN	0.8	0.8
K Medians	0.7910447761194029	0.7903225806451613738

F1

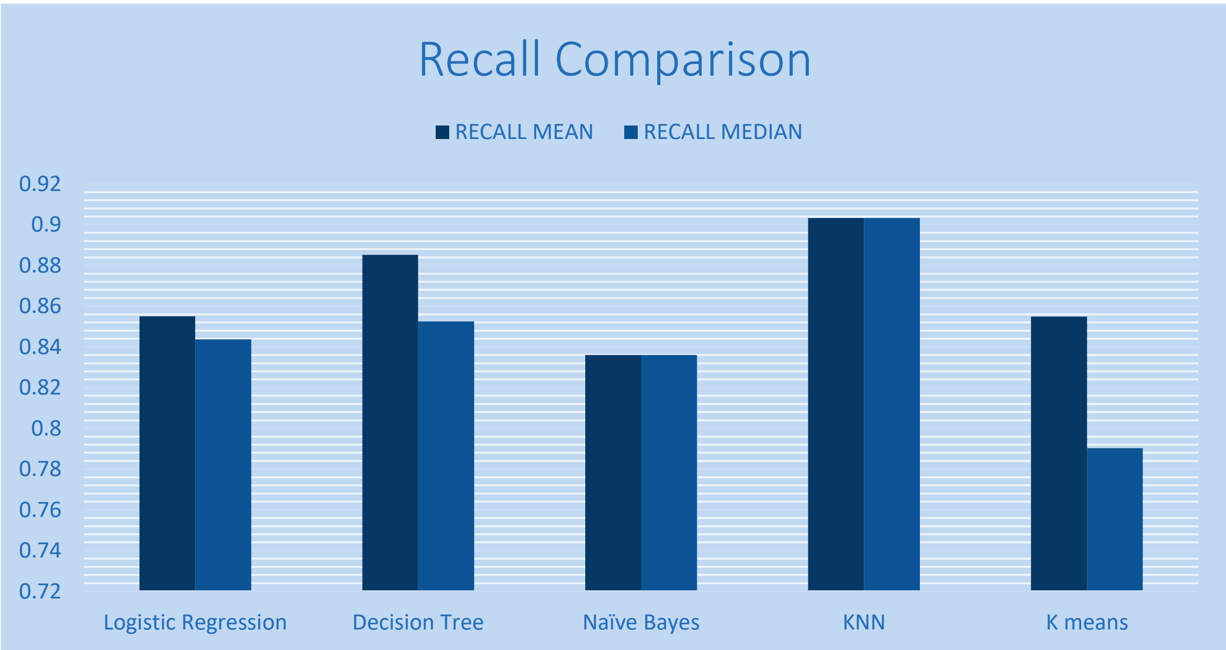
MODEL	MEDIAN	MEDIAN
Logistic Regression	0.8676470588235295	0.8571428571428571
Decision Tree	0.7769784172661871	0.7703703703703704
Naïve Bayes	0.7727272727272727	0.7669172932330828
KNN	0.8484848484848486	0.8484848484848486
K Medians	0.8217054263565892	0.7903225806451614

Log Loss

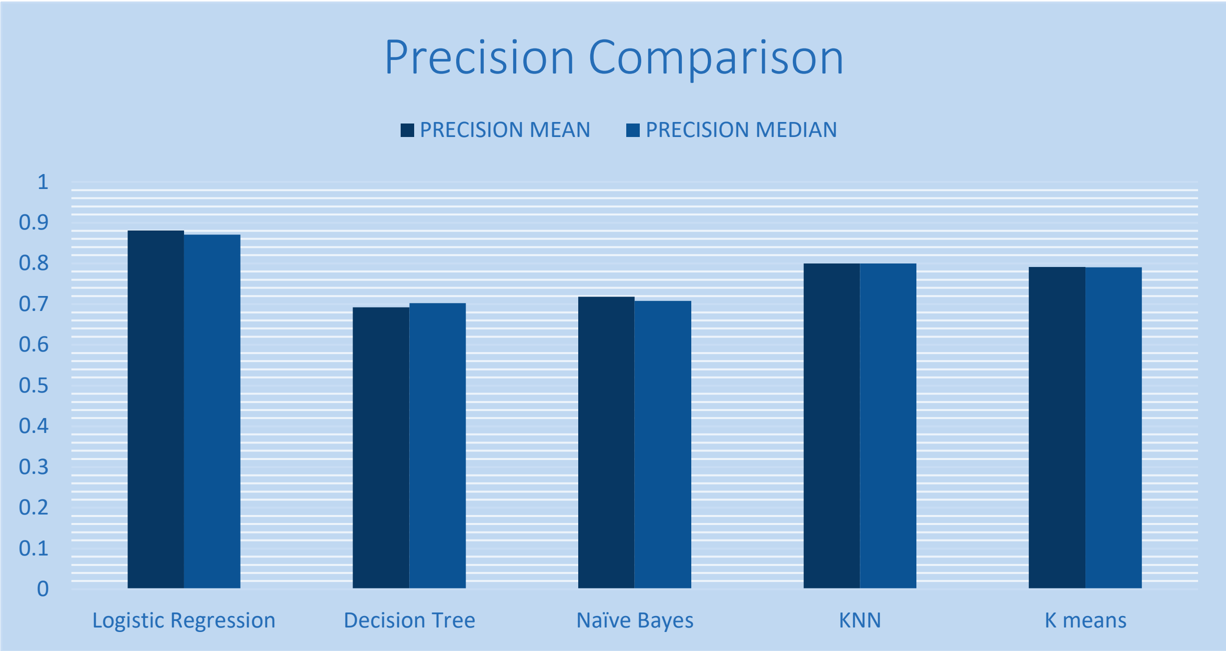
MODEL	MEDIAN	MEDIAN
Logistic Regression	0.379049368470849	0.3365017438911693
Decision Tree	0.7230374691946901	1.5152060500196904
Naïve Bayes	1.1112545612072136	1.1106203966238264
KNN	0.42653563898605484	0.4490046250780085



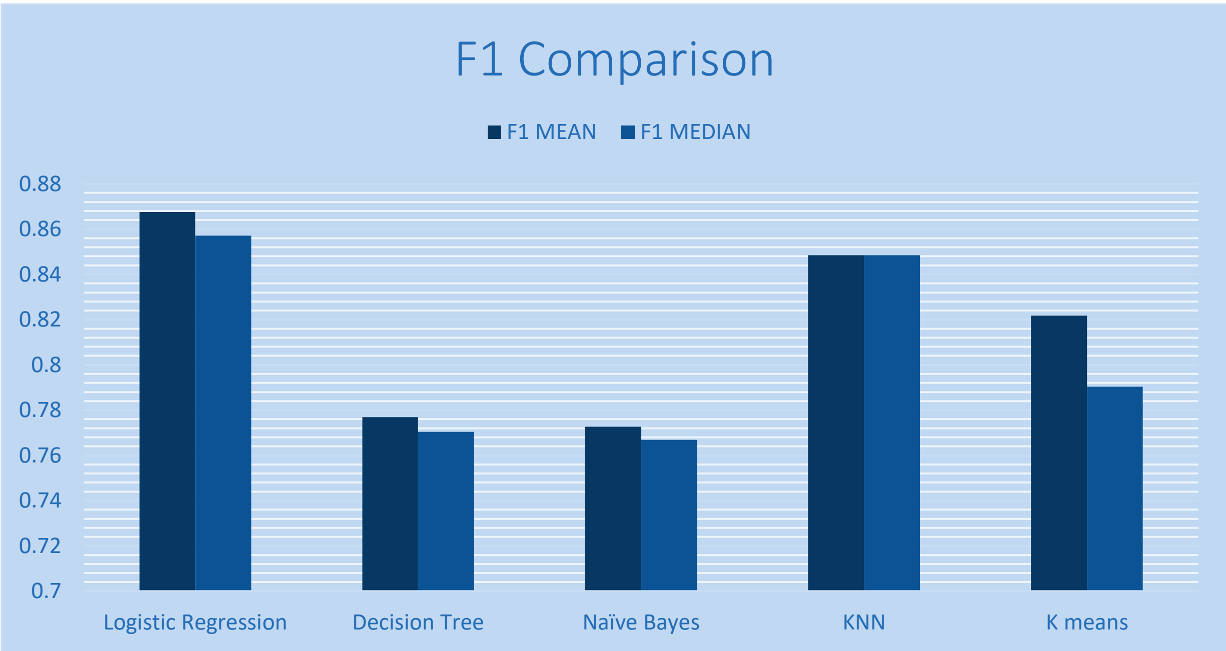
Therefore, Accuracy is highest for Logistic Regression with outliers replaced with Median.



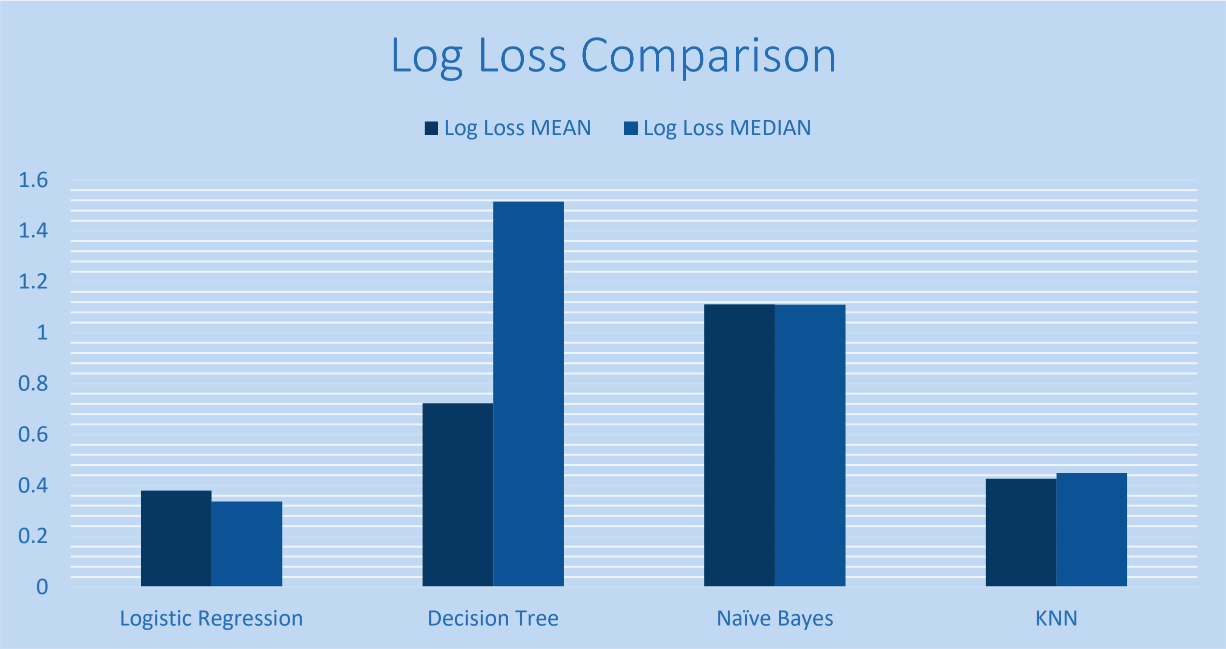
Therefore, Recall is highest for KNN, irrespective of how the outliers are removed.



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 pressu
re')
```

```
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 ac
hieved')
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.medi
an()})


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)), c
olumns = 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.t
arget)
medianheart = pd.DataFrame(preprocessing.scale(medianheart.drop("target", ax
is = 1)), columns = medianheart.drop("target", axis = 1).columns).join(media
nheart.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 300
0 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 tra
ined
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 m
atrix')

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 m
atrix')

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 m
atrix')

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 300
0 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 tra
ined
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 m
atrix')

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 m
atrix')

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 m
atrix')

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_lea
f=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 tra
ined
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 m
atrix')

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 m
atrix')

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 m
atrix')

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_lea
f=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 tra
ined
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 m
atrix')

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 m
atrix')

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 m
atrix')

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 tra
ined
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 m
atrix')

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 m
atrix')

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 m
atrix')

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 tra
ined
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 m
atrix')

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 [ ]:

```python
# 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 m
atrix')

plt.show()
```

In [ ]:

```python
# 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 m
atrix')

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 tra
ined
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 m
atrix')

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.
```

```python
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 m
atrix')

plt.show()
```

In [ ]:

```python
# 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 m
atrix')

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 tra
ined
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 m
atrix')

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 m
atrix')

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 m
atrix')

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 m
atrix')

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 m
atrix')

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