A Project/Dissertation Report

on

HEART DISEASE PREDICTION SYSTEM

Submitted in partial fulfillment of the requirement for the award of the degree of

Bachelor of Technology in Computer Science Engineering

Under The Supervision of
Dr. Ganga Sharma
Associate Professor

Submitted By

Abhinav Kulshreshth
18021011889
Mahima Yadav
18021011527

SCHOOL OF COMPUTING SCIENCE AND ENGINEERING DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING GALGOTIAS UNIVERSITY, GREATER NOIDA INDIA OCTOBER'21

SCHOOL OF COMPUTING SCIENCE AND ENGINEERING
GALGOTIAS UNIVERSITY, GREATER NOIDA

CANDIDATE'S DECLARATION

We hereby certify that the work which is being presented in the project, entitled **"Heart Disease Prediction using Machine Learning"**in partial fulfillment of the requirements for the award of the Bachelors of Technology submitted in the School of Computing Science and Engineering of Galgotias University, Greater Noida, is an original work carried out during the period of July 2021 to December 2021, under the supervision of Dr. Ganga Sharma, Associate Professor, Department of Computer Science and Engineering, of School of Computing Science and Engineering , Galgotias University, Greater Noida.

The matter presented in the project has not been submitted by us for the award of any other degree of this or any other places.

Mahima Yadav
18SCSE1010291

Abhinav Kulshreshth
18SCSE1010665

This is to certify that the above statement made by the candidates is correct to the best of my knowledge.

Dr. Ganga Sharma

Associate Professor

CERTIFICATE

The Final Thesis/Project/ Dissertation Viva-Voce examination of Mahima Yadav(18SCSE1010291) and Abhinav Kulshreshtha(18SCSE1010665) has been held on _____ and his/her work is recommended for the award of Bachelors of Technology.

Signature of Examiner(s) **Signature of Supervisor(s)**

Signature of Project Coordinator **Signature of Dean**

Date: 16th December, 2021

Place: Greater Noida

Table of Contents

Abstract

According to World Health Organization (WHO), cardiovascular diseases are world's leading cause of deaths with approximately 32% (17.9 million) of all deaths every year. Early detection of heart related illness may prove beneficial in decrease the deaths worldwide.

Currently, most of the detection is being done by a doctor based on the reports and lab results, but this can be unreliable but it is the most trusted method right now. Patients check with multiple just to decrease the human error coefficient which costs them very much and takes much time. Many researches have also been conducted to integrate Machine Learning into the complete detection methodology.

This paper proposes a new Machine Learning based disease detector which can be used to check whether a patient has some kind of heart disease or not. The model uses some information from patient's test reports such as chest pain type, cholesterol, condition of thalassemia etc., and classifies the patient as a probable heart disease patient or non-probable.
The data has been through multiple feature transformations based on H2O.ai auto feature transformations. It has also been through multiple feature selection algorithms such as Linear SVC, Lasso, and Logistic Regression etc., to select the best features from the data.

Keywords—Machine Learning, Heart Disease Detection, Healthcare, H2O, Classification

Literature Survey

Researchers Shadab Adam Pattekari and Asma Parveen[1] developed a system using Naïve Bayes data mining modelling technique. Their system obtains hidden information from a heart disease database. It can even be used to find out answers for more complex questions and include other data mining techniques as well.

Arabasadi, Zeinab et al.[2] proposed a hybrid method made up of Neural Network and Genetic Algorithm. Genetic Algorithm was used to select better weights for the Neural Network. This increased the accuracy of their initial model by 10%. The accuracy of their model was 93.85%, while their sensitivity and specificity were 97% and 93% respectively.

Shadman Nashif et al.[3] proposed an SVM based model which was validated using 10-fold cross validation technique and tested on 2 different datasets. They also created a cloud-based monitoring system using Arduino, which was capable of sensing some real-time patient information. The accuracy was found to be 97.53%, the sensitivity was at 97.5% and specificity was at 94.94%.

Researchers Rahma Atallah and Amjed Al-Mousa[4] proposed a majority voting ensemble method, which ran multiple models such as Stochastic Gradient Boosting Classifier, KNN Classifier, Random Forest Classifier, and Logistic Regression Classifier and predicted results based on the majority voting from all the models. This ensemble method produced an accuracy of 90%.

Researchers Ricardo Buettner and Marc Schunter[5] proposed a Random Forest Classifier over the Cleveland Dataset. They ran the classifier over 10-fold cross validation which produced their best results with an accuracy of 84.448%. Without the cross validation, they got an accuracy of 82.895% from the Random Forest Classifier.

Amin Ul Haq et al.[6] compares six different machine learning model for classification along with a Back Propagation Neural Network (BPNN). They found SVM to be the best on model evaluation with 86% model accuracy. While testing with the ensemble methods, SVM obtained an accuracy of 92.3% while BPNN scored 93% in accuracy. They concluded that BPNN was the most effective algorithm for heart disease detection.

Arabasadi Zeinab et al.[7] proposed a hybrid method with neural network using genetic algorithms for enhancing the initial weights to the neural network. This method resulted in a 10% improvement in the neural network. Using this method, they achieved an accuracy of 93.85%, sensitivity of 97% and precision of 92% on the Z-Alizadeh Sani dataset.

Rajkumar Asha and Reena G. Sophia[8] proposed usage of two algorithms Bayes classifier, and k-nearest neighbor and usage of a data mining software known as Tanagra. This software was used to classify the data using 10-fold cross validation.

Problem Formulation

Our problem can be best formulated as the binary classification on whether the patient can have a heart disease or not. The dataset that we used for the problem can be found on the UCI Machine Learning Repository. The database used for the training and validation of the model was the Cleveland UCI Heart Disease Dataset. The dataset consists of 303 records of patients.

There are a total of 13 features columns and 1 target column. Out of the 14 total columns 5 columns were numerical, 8 columns were categorical, and 1 column was Boolean. If the patient has/may have a heart disease then the model will result in 1 or “May have a heart disease”, otherwise 0 or “May not have a heart disease”.

Required Tools

Tools required to solve this ML problem include the computer language Python which will be used for creating the model. Anaconda will be used for the environment as it includes most of the packages needed by us for solving the problem. Tools used are as follows –

- Jupyter Notebook
- Anaconda
- Python
- TensorFlow
- Scikit-learn
- H2O
- Google Colab

Feasibility Analysis

The data for the project can be acquired easily on the Internet. Since the problem is of binary classification, the labeling for the data is inexpensive. The data which has been previously used by researchers consisted of the 303 records that we have used with 13 feature columns and 1 target column.

We want the system to predict whether the patient has/may have a heart disease or may not have a heart disease. And since this model may be used in healthcare sector, so the model created must be as accurate as possible and must have a very now false negative percentage.
There have been previous models based on different models and ensemble methods and data mining methods.

Complete Work Plan

Data collection and feature extraction – We have found the data required by us to create a dataset. We have also build the data-ingestion pipelines which includes feature transformation and validate the quality of our data.
We have also cleaned the redundant data and extract the important features from the data.

Model exploration – Establish baselines for model performance, creating a simple model using initial data pipelines and train it. Try parallel ideas and find a suitable model for our problem and changing the baselines.

Model refinement – Perform model-specific optimizations and perform error analysis while debugging the model due to the added complexity.

Testing and evaluation – Evaluating the model on test distributions, ensuring that the model evaluation metric drives desirable downstream user behavior. We also need to ensure that the false negative rate is at their lowest as this model may or may not be used in healthcare sector.

Workflow

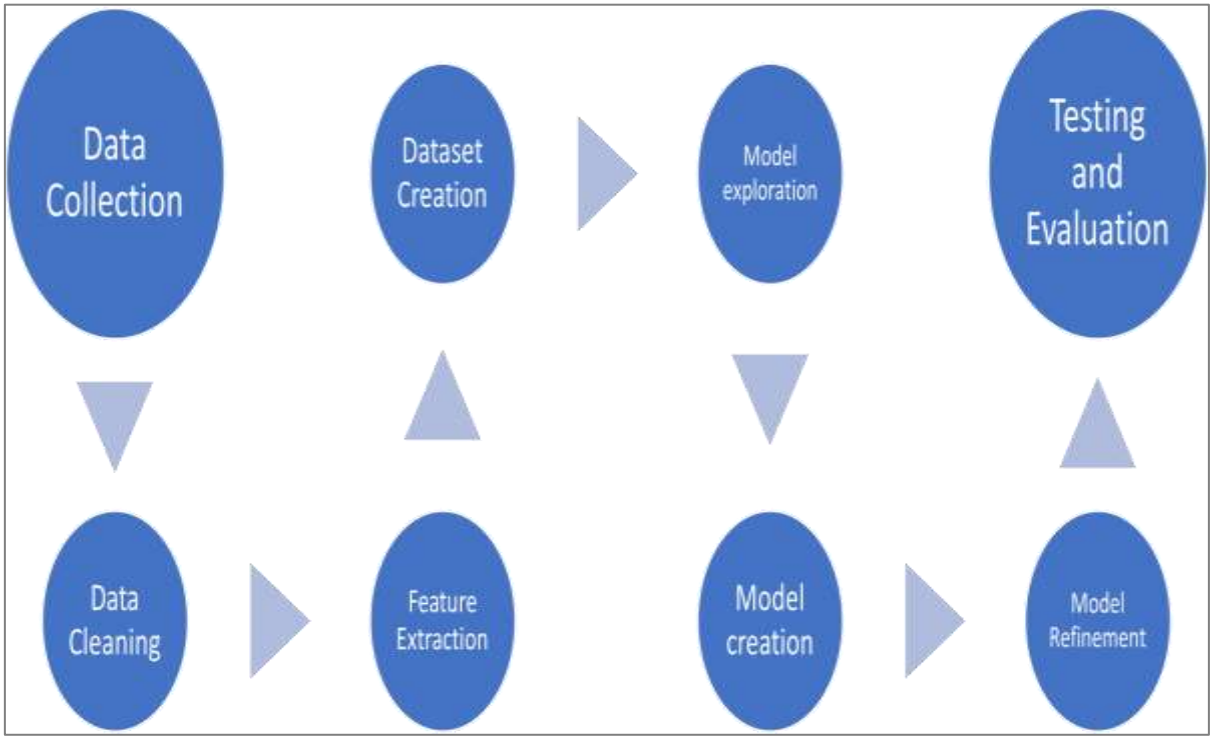


Fig 1: Workflow for the project

Feature Transformation and Selection

Feature Transformation

Weight of Evidence (WoE) measures the "**strength"** of a grouping technique to separate good and bad. This method was developed primarily to build a predictive model to evaluate the risk of loan default in the credit and financial industry. Weight of evidence (WOE) measures how much the **evidence supports or undermines a hypothesis**.

It is computed as below:

$$WoE = \left[ln\left(\frac{Distr\ Goods}{Distr\ Bads} \right) \right] * 100$$

WoE will be 0 if the P(Goods) / P(Bads) = 1. That is, if the outcome is random for that group. If P(Bads) > P(Goods) the odds ratio will be < 1 and the WoE will be < 0; if, on the other hand, P(Goods) > P(Bads) in a group, then WoE > 0.

WoE is well suited for Logistic Regression because the Logit transformation is simply the log of the odds, i.e., ln(P(Goods)/P(Bads)). Therefore, by using WoE-coded predictors in Logistic Regression, the predictors are prepared and coded to the same scale. The parameters in the linear logistic regression equation can be directly compared.

The WoE transformation has (at least) three advantage:

1) It can transform an independent variable to establish a monotonic relationship to the dependent variable. It does more than this — to secure a monotonic relationship it would be enough to "recode" it to any ordered measure (for example 1,2,3,4…), but the WoE transformation orders the categories on a "logistic" scale which is natural for Logistic Regression

2) For variables with too many (sparsely populated) discrete values, these can be grouped into categories (densely populated), and the WoE can be used to express information for the whole category

3) The (univariate) effect of each category on the dependent variable can be compared across categories and variables because WoE is a standardized value (for example, you can compare WoE of married people to WoE of manual workers)

Usage of WOE

Weight of Evidence (WOE) helps to transform a continuous independent variable into a set of groups or bins based on similarity of dependent variable distribution i.e. number of events and non-events.

For continuous independent variables : First, create bins (categories / groups) for a continuous independent variable and then combine categories with similar WOE values and replace categories with WOE values. Use WOE values rather than input values in your model.

For categorical independent variables: Combine categories with similar WOE and then create new categories of an independent variable with continuous WOE values. In other words, use WOE values rather than raw categories in your model. The transformed variable will be a continuous variable with WOE values. It is same as any continuous variable.

K Fold Target Encoding

The basic idea of the k-fold target encoding originates from the mean-target encoding. In the mean-target encoding, the categorical variables are replaced by the mean of the target corresponding to them. It is seen from fig.2 that the mean of the target when "Feature" is A = 0.6 and B=0.3.

Therefore, A and B will be replaced by 0.6 and 0.3 respectively. This new feature might be more correlated to the target. However, this approach might have a tendency to the overfitting when the distribution of the categorical variables in Feature of the train and the test dataset are considerably different.

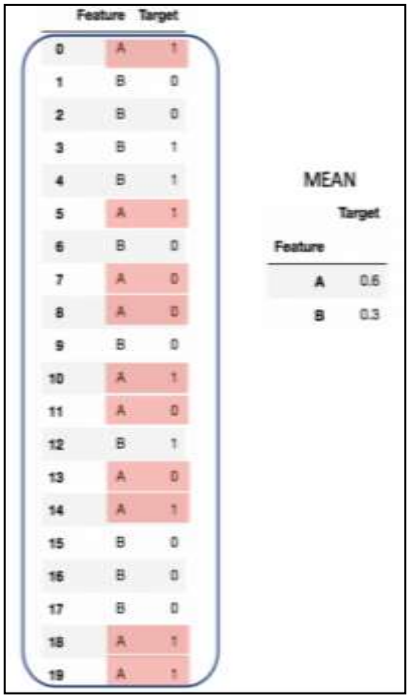


Fig 2: Target or mean encoding.

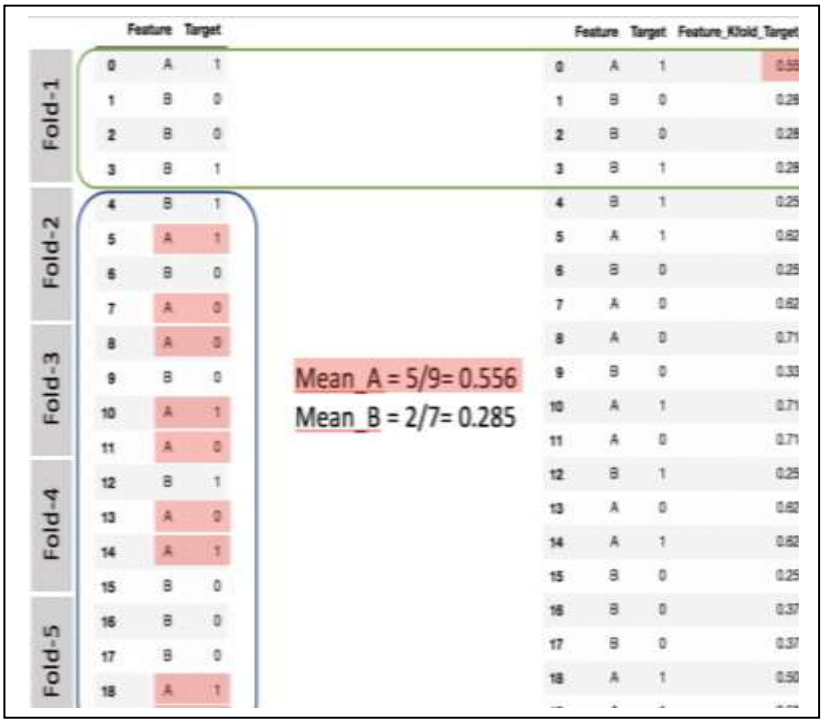
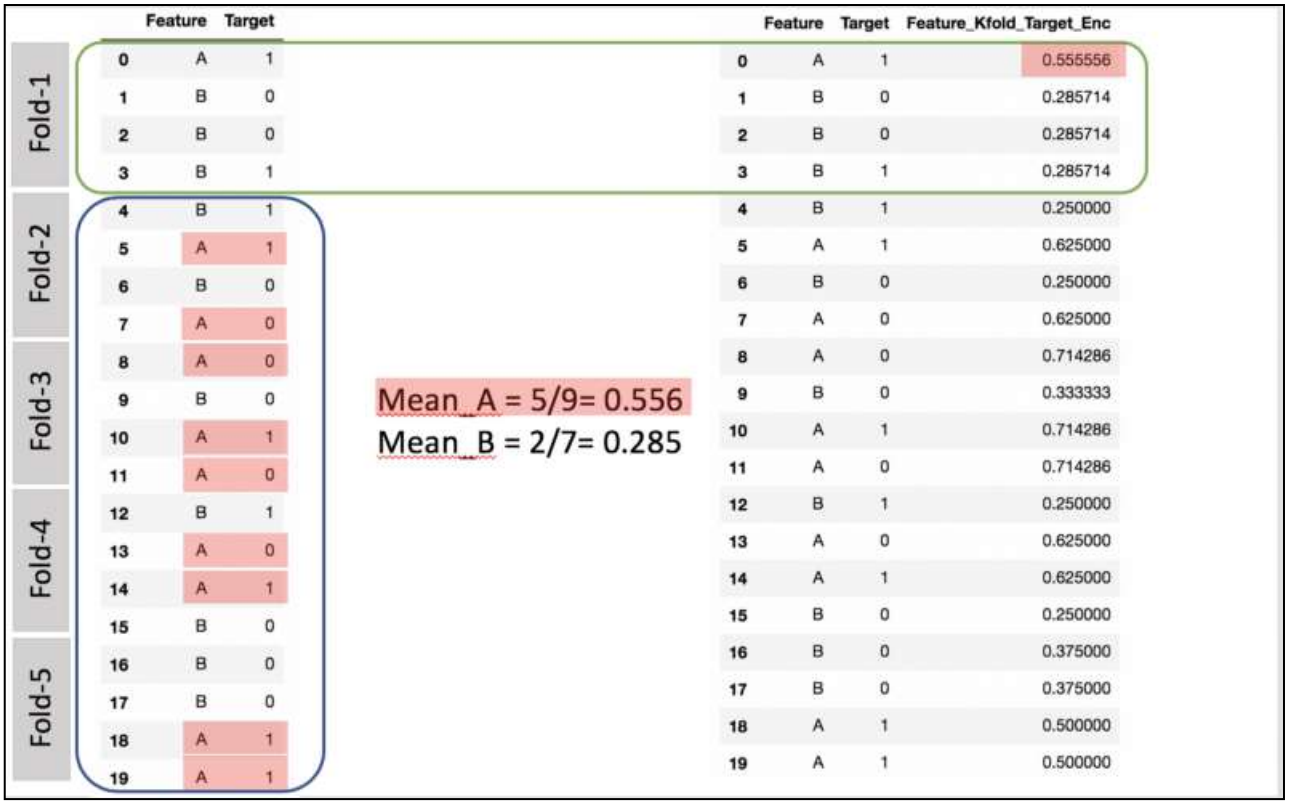


Fig 3: 5-fold target encoding. We use fold 2,3,4,5 to estimate first-fold.

Therefore, k-fold target encoding can be applied to reduce the overfitting. In this method, we divide the dataset into the k-folds, here we consider 5 folds. Fig.3 shows the first round of the 5 fold cross-validation. We calculate mean-target for fold 2, 3, 4 and 5 and we use the calculated values, mean_A = 0.556 and mean_B = 0.285 to estimate mean encoding for the fold-1.

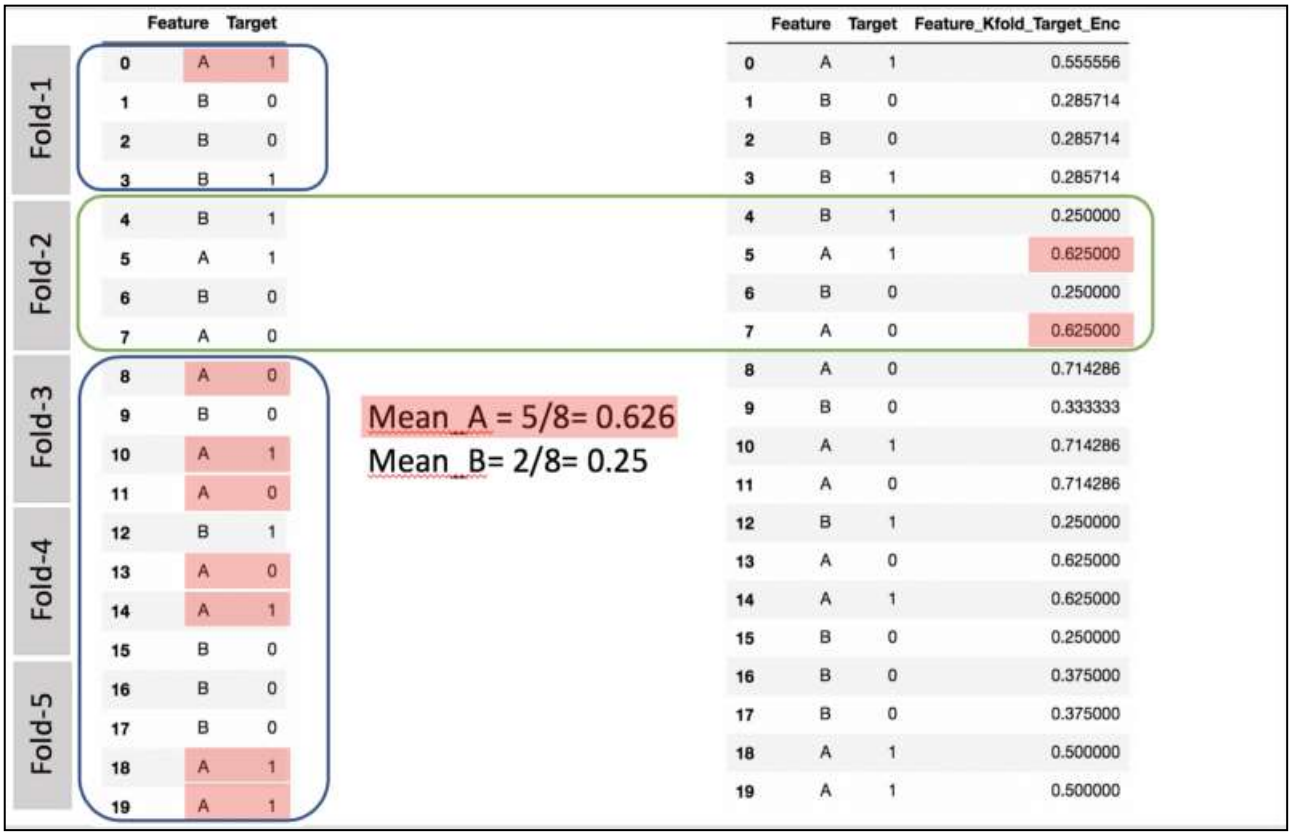


	Feature	Target
0	A	1
1	B	0
2	B	0
3	B	1
4	B	1
5	A	1
6	B	0
7	A	0
8	A	0
9	B	0
10	A	1
11	A	0
12	B	1
13	A	0
14	A	1
15	B	0
16	B	0
17	B	0
18	A	1
19	A	1

	Feature	Target	Feature_Kfold_Target_Enc
0	A	1	0.555556
1	B	0	0.285714
2	B	0	0.285714
3	B	1	0.285714
4	B	1	0.250000
5	A	1	0.625000
6	B	0	0.250000
7	A	0	0.625000
8	A	0	0.714286
9	B	0	0.333333
10	A	1	0.714286
11	A	0	0.714286
12	B	1	0.250000
13	A	0	0.625000
14	A	1	0.625000
15	B	0	0.250000
16	B	0	0.375000
17	B	0	0.375000
18	A	1	0.500000
19	A	1	0.500000

Fig 4: 5-fold target encoding. We use fold 2,3,4,5 to estimate first-fold.

After that, we can calculate for the second fold as it is shown in Fig.4



	Feature	Target
0	A	1
1	B	0
2	B	0
3	B	1
4	B	1
5	A	1
6	B	0
7	A	0
8	A	0
9	B	0
10	A	1
11	A	0
12	B	1
13	A	0
14	A	1
15	B	0
16	B	0
17	B	0
18	A	1
19	A	1

	Feature	Target	Feature_Kfold_Target_Enc
0	A	1	0.555556
1	B	0	0.285714
2	B	0	0.285714
3	B	1	0.285714
4	B	1	0.250000
5	A	1	0.625000
6	B	0	0.250000
7	A	0	0.625000
8	A	0	0.714286
9	B	0	0.333333
10	A	1	0.714286
11	A	0	0.714286
12	B	1	0.250000
13	A	0	0.625000
14	A	1	0.625000
15	B	0	0.250000
16	B	0	0.375000
17	B	0	0.375000
18	A	1	0.500000
19	A	1	0.500000

Fig 5: 5-fold target encoding. We use fold 1,3,4,5 to estimate second-fold.

Now the remaining part is creating “Feature_Kfold_Target_Enc” column in the test dataset. This column values can be obtained from getting mean of “Feature_Kfold_mean_Enc” train column for the categorical variables “A” and “B”.

	Feature	Target	Feature_Kfold_Target_Enc
0	A	1	0.555556
1	B	0	0.285714
2	B	0	0.285714
3	B	1	0.285714
4	B	1	0.250000
5	A	1	0.625000
6	B	0	0.250000
7	A	0	0.625000
8	A	0	0.714286
9	B	0	0.333333
10	A	1	0.714286
11	A	0	0.714286
12	B	1	0.250000
13	A	0	0.625000
14	A	1	0.625000
15	B	0	0.250000
16	B	0	0.375000
17	B	0	0.375000
18	A	1	0.500000
19	A	1	0.500000

Mean_A = (.5556+.625+.625+0.714286+0.714286+0.714286+0.625+0.625+0.5+0.5)/10
Mean_A = 0.61981

	Feature
0	B
1	B
2	B
3	A
4	A

	Feature	Feature_Kfold_Target_Enc
0	B	0.294048
1	B	0.294048
2	B	0.294048
3	A	0.619841
4	A	0.619841

Fig 6: We estimate the Feature_Kfold_target_Enc for the test from the train.

Feature Selection

After the features were transformed using WoE encoders and CVTE, all the features were sent through a number of feature selection algorithms. Finally, 15 features which had featured maximum number of times in the output of all the feature selection algorithms were used as the new features in the dataset. The new features included the newly transformed thalassemia feature by WoE Encoder. They also included Sex, Slope and Thalassemia features transformed by CVTE.

Pearson Correlation

Pearson’s correlation (also called Pearson’s *R*) is a correlation coefficient commonly used in linear regression. If you’re starting out in statistics, you’ll probably learn about Pearson’s *R* first. In fact, when anyone refers to the correlation coefficient, they are usually talking about Pearson’s.

Real Life Example

Pearson correlation is used in thousands of real life situations. For example, scientists in China wanted to know if there was a relationship between how weedy rice populations are different genetically. The goal was to find out the evolutionary potential of the rice. Pearson's correlation between the two groups was analyzed. It showed a positive Pearson Product Moment correlation of between 0.783 and 0.895 for weedy rice populations. This figure is quite high, which suggested a fairly strong relationship.

The Pearson correlation for two objects, with paired attributes, sums the product of their differences from their object means, and divides the sum by the product of the squared differences from the object means.

$$\frac{\sum(x_i-\overline{x})(y_i-\overline{y})}{\sqrt{\sum(x_i-\overline{x})^2}\sqrt{\sum(y_i-\overline{y})^2}}$$

Linear SVC

The Linear Support Vector Classifier (SVC) method applies a linear kernel function to perform classification and it performs well with a large number of samples. If we compare it with the SVC model, the Linear SVC has additional parameters such as penalty normalization which applies 'L1' or 'L2' and loss function. The kernel method cannot be changed in linear SVC, because it is based on the kernel linear method.

Preparing the data

First, we'll generate random classification dataset with make_classification() function. The dataset contains 3 classes with 10 features and the number of samples is 5000.

```
x, y = make_classification(n_samples=5000, n_features=10,
                           n_classes=3,
                           n_clusters_per_class=1)
```

Then, we'll split the data into train and test parts. Here, we'll extract 15 percent of it as test data.

```
xtrain, xtest, ytrain, ytest = train_test_split(x, y, test_size=0.15)
```

Training the model

Next, we'll define the classifier by using the LinearSVC class. We can use the default parameters of the class. The parameters can be changed according to classification data content.

```
lsvc = LinearSVC(verbose=0)
print(lsvc)

LinearSVC(C=1.0, class_weight=None, dual=True, fit_intercept=True,
     intercept_scaling=1, loss='squared_hinge', max_iter=1000,
     multi_class='ovr', penalty='l2', random_state=None, tol=0.0001,
     verbose=0)
```

Then, we'll fit the model on train data and check the model accuracy score.

```
lsvc.fit(xtrain, ytrain)
score = lsvc.score(xtrain, ytrain)
print("Score: ", score)

Score:  0.8602352941176471
```

We can also apply a cross-validation training method to the model and check the training score.

```
cv_scores = cross_val_score(lsvc, xtrain, ytrain, cv=10)
print("CV average score: %.2f" % cv_scores.mean())
CV average score: 0.86
```

Predicting and accuracy check

Now, we can predict the test data by using the trained model. After the prediction, we'll check the accuracy level by using the confusion matrix function.

```
ypred = lsvc.predict(xtest)

cm = confusion_matrix(ytest, ypred)
print(cm)

[[196  46  30]
 [  5 213  10]
 [ 26   7 217]]
```

We can also create a classification report by using classification_report() function on predicted data to check the other accuracy metrics.

```
cr = classification_report(ytest, ypred)
print(cr)
```

	precision	recall	f1-score	support
0	0.86	0.72	0.79	272
1	0.80	0.93	0.86	228
2	0.84	0.87	0.86	250
accuracy			0.83	750
macro avg	0.84	0.84	0.83	750
weighted avg	0.84	0.83	0.83	750

Lasso regression

The word "LASSO" stands for Least Absolute Shrinkage and Selection Operator. It is a statistical formula for the regularisation of data models and feature selection.

What is Lasso Regression?

Lasso regression is a regularization technique. It is used over regression methods for a more accurate prediction. This model uses shrinkage. Shrinkage is where data values are shrunk towards a central point as the mean. The lasso procedure encourages simple, sparse models (i.e. models with fewer parameters). This particular type of regression is well-suited for models showing high levels of multicollinearity or when you want to automate certain parts of model selection, like variable selection/parameter elimination.

Lasso Regression uses L1 regularization technique (will be discussed later in this article). It is used when we have more number of features because it automatically performs feature selection.

L1 Regularization

If a regression model uses the L1 Regularization technique, then it is called Lasso Regression. If it used the L2 regularization technique, it's called Ridge Regression. We will study more about these in the later sections.

L1 regularization adds a penalty that is equal to the absolute value of the magnitude of the coefficient. This regularization type can result in sparse models with few

coefficients. Some coefficients might become zero and get eliminated from the model. Larger penalties result in coefficient values that are closer to zero (ideal for producing simpler models). On the other hand, L2 regularization does not result in any elimination of sparse models or coefficients. Thus, Lasso Regression is easier to interpret as compared to the Ridge.

Mathematical equation of Lasso Regression

Residual Sum of Squares + λ * (Sum of the absolute value of the magnitude of coefficients)

$$\sum_{i=1}^{n}(y_i - \sum_{j} x_{ij}\beta_j)^2 + \lambda \sum_{j=1}^{p} |\beta_j|$$

Where,

- λ denotes the amount of shrinkage.
- λ = 0 implies all features are considered and it is equivalent to the linear regression where only the residual sum of squares is considered to build a predictive model
- λ = ∞ implies no feature is considered i.e, as λ closes to infinity it eliminates more and more features
- The bias increases with increase in λ
- variance increases with decrease in λ

Lasso regression example

```
import numpy as np
```

Creating a New Train and Validation Datasets

```
from sklearn.model_selection import train_test_split
data_train, data_val = train_test_split(new_data_train, test_size = 0.2, random_state = 2)
```

Classifying Predictors and Target

```
#Classifying Independent and Dependent Features
#_____________________________________________
#Dependent Variable
Y_train = data_train.iloc[:, -1].values
#Independent Variables
X_train = data_train.iloc[:,0 : -1].values
#Independent Variables for Test Set
X_test = data_val.iloc[:,0 : -1].values
```

Evaluating The Model With RMLSE

```
def score(y_pred, y_true):
error = np.square(np.log10(y_pred +1) - np.log10(y_true +1)).mean() ** 0.5
score = 1 – error
```

```
return score
actual_cost = list(data_val['COST'])
actual_cost = np.asarray(actual_cost)
```

Building the Lasso Regressor

```
#Lasso Regression

from sklearn.linear_model import Lasso
#Initializing the Lasso Regressor with Normalization Factor as True
lasso_reg = Lasso(normalize=True)
#Fitting the Training data to the Lasso regressor
lasso_reg.fit(X_train,Y_train)
#Predicting for X_test
y_pred_lass =lasso_reg.predict(X_test)
#Printing the Score with RMLSE
print("\n\nLasso SCORE : ", score(y_pred_lass, actual_cost))
```

Output

0.7335508027883148

The Lasso Regression attained an accuracy of 73% with the given Dataset.

Variance Threshold

The variance threshold is a simple baseline approach to feature selection. It removes all features which variance doesn't meet some threshold. By default, it removes all zero-variance features, i.e., features that have the same value in all samples.

```
from sklearn.feature_selection import VarianceThreshold

# Resetting the value of X to make it non-categorical
X = array[:,0:8]

v_threshold = VarianceThreshold(threshold=0)
v_threshold.fit(X)  # fit finds the features with zero variance
v_threshold.get_support()
```

```
array([ True,  True,  True,  True,  True,  True,  True,  True])
```

We assume that features with a higher variance may contain more useful information, but note that we are not taking the relationship between feature variables or feature and target variables into account, which is one of the drawbacks of filter methods.
Default Value of Threshold is 0

- If Variance Threshold = 0 (Remove Constant Features)
- If Variance Threshold > 0 (Remove Quasi-Constant Features)

Regressive feature elimination using Logistic Regression

Recursive Feature Elimination, or RFE for short, is a popular feature selection algorithm.
RFE is popular because it is easy to configure and use and because it is effective at selecting those features (columns) in a training dataset that are more or most relevant in predicting the target variable.

There are two important configuration options when using RFE: the choice in the number of features to select and the choice of the algorithm used to help choose features.
Both of these hyperparameters can be explored, although the performance of the method is not strongly dependent on these hyperparameters being configured well.

- RFE is a wrapper-type feature selection algorithm. This means that a different machine learning algorithm is given and used in the core of the method, is wrapped by RFE, and used to help select features. This is in contrast to filter-based feature selections that score each feature and select those features with the largest (or smallest) score.
- Technically, RFE is a wrapper-style feature selection algorithm that also uses filter-based feature selection internally.
- RFE works by searching for a subset of features by starting with all features in the training dataset and successfully removing features until the desired number remains.
- This is achieved by fitting the given machine learning algorithm used in the core of the model, ranking features by importance, discarding the least important features, and re-fitting the model. This process is repeated until a specified number of features remains.

Following table shows the Top 15 features with frequency and transformations:

Features	Transformed (Yes/No)	Frequency
Age	No	5
Resting Blood Pressure	No	5
Cholesterol	No	5
Max heart rate achieved	No	5
Depression	No	4
Sex	Yes, CVTE	4
Thalassemia	Yes, CVTE	4
Number of major vessels	No	4
Sex	No	4
Slope	No	4
Exercise induced angina	No	4
Resting ECG	No	4
Chest Pain Type	No	4
Slope	Yes, CVTE	3
Thalassemia	Yes, WoE	3

Model Exploration

Stacked ensemble

Stacking or Stacked Generalization is an ensemble machine learning algorithm.
It uses a meta-learning algorithm to learn how to best combine the predictions from two or more base machine learning algorithms.

The benefit of stacking is that it can harness the capabilities of a range of well-performing models on a classification or regression task and make predictions that have better performance than any single model in the ensemble.
Stacking is appropriate when multiple different machine learning models have skill on a dataset, but have skill in different ways. Another way to say this is that the predictions made by the models or the errors in predictions made by the models are uncorrelated or have a low correlation.

Generalized linear models

Generalized Linear Model (GLiM, or GLM) is an advanced statistical modelling technique formulated by John Nelder and Robert Wedderburn in 1972.
It is an umbrella term that encompasses many other models, which allows the response variable y to have an error distribution other than a normal distribution. The models include Linear Regression, Logistic Regression, and Poisson Regression.

GLM models allow us to build a linear relationship between the response and predictors, even though their underlying relationship is not linear. This is made possible by using a link function, which links the response variable to a linear model. Unlike Linear Regression models, the error distribution of the response variable need not be normally distributed. The errors in the response variable are assumed to follow an exponential family of distribution.

XGBoost

XGBoost is an implementation of gradient boosted decision trees designed for speed and performance.

It is an implementation of gradient boosting machines created by Tianqi Chen, now with contributions from many developers. It belongs to a broader collection of tools under the umbrella of the Distributed Machine Learning Community or DMLC who are also the creators of the popular mxnet deep learning library.

XGBoost is a software library that you can download and install on your machine, then access from a variety of interfaces. Specifically, XGBoost supports the following main interfaces:

- Command Line Interface (CLI).
- C++ (the language in which the library is written).
- Python interface as well as a model in scikit-learn.
- R interface as well as a model in the caret package.
- Julia.
- Java and JVM languages like Scala and platforms like Hadoop.

Machine Learning Model

Gradient Boosting Machine:

Gradient boosting machines are a family of powerful machine-learning techniques that have shown considerable success in a wide range of practical applications. They are highly customizable to the particular needs of the application, like being learned with respect to different loss functions. This article gives a tutorial introduction into the methodology of gradient boosting methods with a strong focus on machine learning aspects of modeling. A theoretical information is complemented with descriptive examples and illustrations which cover all the stages of the gradient boosting model design.

Gradient boosting involves three elements:

- A loss function to be optimized.
- A weak learner to make predictions.
- An additive model to add weak learners to minimize the loss function.

1. Loss Function

The loss function used depends on the type of problem being solved.

It must be differentiable, but many standard loss functions are supported and you can define your own.

For example, regression may use a squared error and classification may use logarithmic loss.

A benefit of the gradient boosting framework is that a new boosting algorithm does not have to be derived for each loss function that may want to be used, instead, it is a generic enough framework that any differentiable loss function can be used.

2. Weak Learner

Decision trees are used as the weak learner in gradient boosting.

Specifically regression trees are used that output real values for splits and whose output can be added together, allowing subsequent models outputs to be added and "correct" the residuals in the predictions.

Trees are constructed in a greedy manner, choosing the best split points based on purity scores like Gini or to minimize the loss.

Initially, such as in the case of AdaBoost, very short decision trees were used that only had a single split, called a decision stump. Larger trees can be used generally

with 4-to-8 levels.

It is common to constrain the weak learners in specific ways, such as a maximum number of layers, nodes, splits or leaf nodes.This is to ensure that the learners remain weak, but can still be constructed in a greedy manner.

3. Additive Model

Trees are added one at a time, and existing trees in the model are not changed.

A gradient descent procedure is used to minimize the loss when adding trees.

Traditionally, gradient descent is used to minimize a set of parameters, such as the coefficients in a regression equation or weights in a neural network. After calculating error or loss, the weights are updated to minimize that error.

Instead of parameters, we have weak learner sub-models or more specifically decision trees. After calculating the loss, to perform the gradient descent procedure, we must add a tree to the model that reduces the loss (i.e. follow the gradient). We do this by parameterizing the tree, then modify the parameters of the tree and move in the right direction by (reducing the residual loss.

Generally this approach is called functional gradient descent or gradient descent with functions.

Improvements to Basic Gradient Boosting

Gradient boosting is a greedy algorithm and can overfit a training dataset quickly.

It can benefit from regularization methods that penalize various parts of the algorithm and generally improve the performance of the algorithm by reducing overfitting.

4 enhancements to basic gradient boosting:

1. Tree Constraints
2. Shrinkage
3. Random sampling
4. Penalized Learning

1. Tree Constraints

There are a number of ways that the trees can be constrained.A good general heuristic is that the more constrained tree creation is, the more trees you will need

in the model, and the reverse, where less constrained individual trees, the fewer trees that will be required.

Below are some constraints that can be imposed on the construction of decision trees:

- Number of trees**,** generally adding more trees to the model can be very slow to overfit. The advice is to keep adding trees until no further improvement is observed.
- Tree depth, deeper trees are more complex trees and shorter trees are preferred. Generally, better results are seen with 4-8 levels.
- Number of nodes or number of leaves, like depth, this can constrain the size of the tree, but is not constrained to a symmetrical structure if other constraints are used.
- Number of observations per split imposes a minimum constraint on the amount of training data at a training node before a split can be considered
- Minimim improvement to loss is a constraint on the improvement of any split added to a tree.

2. Weighted Updates

The predictions of each tree are added together sequentially.

The contribution of each tree to this sum can be weighted to slow down the learning by the algorithm. This weighting is called shrinkage or a learning rate.

3. Stochastic Gradient Boosting

A big insight into bagging ensembles and random forest was allowing trees to be greedily created from subsamples of the training dataset.

This same benefit can be used to reduce the correlation between the trees in the sequence in gradient boosting models.

This variation of boosting is called stochastic gradient boosting.A few variants of stochastic boosting that can be used:

- Subsample rows before creating each tree.
- Subsample columns before creating each tree
- Subsample columns before considering each split.

4. Penalized Gradient Boosting

Additional constraints can be imposed on the parameterized trees in addition to their structure.

Classical decision trees like CART are not used as weak learners, instead a

modified form called a regression tree is used that has numeric values in the leaf nodes (also called terminal nodes). The values in the leaves of the trees can be called weights in some literature.

As such, the leaf weight values of the trees can be regularized using popular regularization functions, such as:

- L1 regularization of weights.
- L2 regularization of weights.

```
MSE: 0.031838318165083576
RMSE: 0.17843295145539564
LogLoss: 0.11560433166077712
Mean Per-Class Error: 0.03350640359986157
AUC: 0.9932156455520942
AUCPR: 0.9944999851424547
Gini: 0.9864312911041884
```

Fig 7:Metrics for GBM Model

Source Code

"""Heart Detection Feature Engineering

Automatically generated by Colaboratory.

Original file is located at
https://colab.research.google.com/drive/1hpkWZ9Izbegov0yqKt2FxZvswTntCv8w

Dataset

1. age: age in years
2. sex: sex (1 = male; 0 = female)
3. cp: chest pain type
   1. Value 0: typical angina
   2. Value 1: atypical angina
   3. Value 2: non-anginal pain
   4. Value 3: asymptomatic
4. trestbps: resting blood pressure (in mm Hg on admission to the hospital)
5. chol: serum cholestoral in mg/dl
6. fbs: (fasting blood sugar > 120 mg/dl) (1 = true; 0 = false)
7. restecg: resting electrocardiographic results
   1. Value 0: normal
   2. Value 1: having ST-T wave abnormality (T wave inversions and/or ST elevation or depression of > 0.05 mV)

  3. Value 2: showing probable or definite left ventricular hypertrophy by Estes' criteria
8. thalach: maximum heart rate achieved
9. exang: exercise induced angina (1 = yes; 0 = no)
10. oldpeak = ST depression induced by exercise relative to rest
11. slope: the slope of the peak exercise ST segment
  1. Value 0: upsloping
  2. Value 1: flat
  3. Value 2: downsloping
12. ca: number of major vessels (0-3) colored by flourosopy
13. thal: 0 = normal; 1 = fixed defect; 2 = reversable defect
and the label
14. condition: 0 = no disease, 1 = disease

Install Python Packages
"""

```
!pip install category_encoders
!pip install requests
!pip install tabulate
!pip install "colorama>=0.3.8"
!pip install future
!pip install -f http://h2o-release.s3.amazonaws.com/h2o/latest_stable_Py.html h2o
!pip install xgboost
!pip install scikit-plot
!pip install pycaret

"""# Import libraries"""

import h2o
import numpy as np
import pandas as pd
import pickle
from sklearn import base
import scikitplot as skplt
import matplotlib.pyplot as plt
from h2o.automl import H2OAutoML
from xgboost import XGBClassifier
from sklearn.svm import SVC, LinearSVC
from category_encoders import WOEEncoder
from sklearn.preprocessing import LabelEncoder
from h2o.estimators import H2OXGBoostEstimator
from scikitplot.helpers import binary_ks_curve
from h2o.grid.grid_search import H2OGridSearch
from sklearn.feature_selection import VarianceThreshold
from sklearn.ensemble import GradientBoostingClassifier
from h2o.estimators.gbm import H2OGradientBoostingEstimator
from sklearn.linear_model import LassoCV, LogisticRegression
from h2o.estimators.stackedensemble import H2OStackedEnsembleEstimator
from scikitplot.metrics import plot_ks_statistic, plot_precision_recall
from sklearn.feature_selection import SelectFromModel, SelectKBest, RFE, chi2
```

```
from sklearn.metrics import
accuracy_score,recall_score,f1_score,precision_score,roc_auc_score
from sklearn.model_selection import KFold, GridSearchCV, train_test_split,
RepeatedStratifiedKFold
from sklearn.metrics import roc_curve, log_loss, plot_roc_curve, cohen_kappa_score,
classification_report, plot_confusion_matrix

"""# Code

### Import dataset
"""

dataset = pd.read_csv("/content/drive/MyDrive/Heart Dataset/heart.csv")

"""### Evaluation function"""

def evaluation(y_test,y_pred):
  acc=accuracy_score(y_test,y_pred)
  rcl=recall_score(y_test,y_pred)
  f1=f1_score(y_test,y_pred)
  ps = precision_score(y_test,y_pred)
  roc = roc_auc_score(y_test,y_pred)

  metric_dict={'Accuracy': round(acc,3),
               'Recall': round(rcl,3),
               'F1 Score': round(f1,3),
               'Precision': round(ps,3),
               'ROC-AUC': round(roc,3)
              }

  return print(metric_dict)

"""### Feature Engineering

#### WOE encoder for thal
"""

woe = WOEEncoder(cols=['thal'], random_state=42, regularization=0)

X = dataset['thal']
y = dataset.target
encoded_df = woe.fit_transform(X, y)

encoded_df

"""#### Cross Validation Target Encoding"""

class KFoldTargetEncoderTrain(base.BaseEstimator, base.TransformerMixin):

    def __init__(self,
colnames,targetName,n_fold=5,verbosity=True,discardOriginal_col=False):
```

```
        self.colnames = colnames
        self.targetName = targetName
        self.n_fold = n_fold
        self.verbosity = verbosity
        self.discardOriginal_col = discardOriginal_col

    def fit(self, X, y=None):
        return self


    def transform(self,X):

        #assert(type(self.targetName) == str)
        #assert(type(self.colnames) == str)
        #assert(self.colnames in X.columns)
        #assert(self.targetName in X.columns)

        mean_of_target = X[self.targetName].mean()
        kf = KFold(n_splits = self.n_fold, shuffle = False, random_state=2019)



        col_mean_name = self.colnames + '_' + 'Kfold_Target_Enc'
        X[col_mean_name] = np.nan

        for tr_ind, val_ind in kf.split(X):
            X_tr, X_val = X.iloc[tr_ind], X.iloc[val_ind]
#             print(tr_ind,val_ind)
            X.loc[X.index[val_ind], col_mean_name] =
X_val[self.colnames].map(X_tr.groupby(self.colnames)[self.targetName].mean())

        X[col_mean_name].fillna(mean_of_target, inplace = True)

        if self.verbosity:

            encoded_feature = X[col_mean_name].values
            print('Correlation between the new feature, {} and, {} is
{}.'.format(col_mean_name,
                                                        self.targetName,

np.corrcoef(X[self.targetName].values, encoded_feature)[0][1]))
        if self.discardOriginal_col:
            X = X.drop(self.targetName, axis=1)



        return X

targetc = KFoldTargetEncoderTrain('thal','target',n_fold=5)
new_dataset = targetc.fit_transform(dataset)
```

```
targetc = KFoldTargetEncoderTrain('cp','target',n_fold=5)
new_dataset = targetc.fit_transform(new_dataset)

targetc = KFoldTargetEncoderTrain('ca','target',n_fold=5)
new_dataset = targetc.fit_transform(new_dataset)

targetc = KFoldTargetEncoderTrain('slope','target',n_fold=5)
new_dataset = targetc.fit_transform(new_dataset)

targetc = KFoldTargetEncoderTrain('sex','target',n_fold=5)
new_dataset = targetc.fit_transform(new_dataset)

"""#### Creating new Dataset (After Feature Engineering)"""

new_dataset['new_thal'] = encoded_df['thal']

new_dataset.to_csv("/content/drive/MyDrive/Heart Dataset/new_dataset.csv")

new_dataset

"""### Feature Selection"""

train = new_dataset.copy()
target = train.pop('target')

num_features_opt = 15   # the number of features that we need to choose as a result
num_features_max = 20   # the somewhat excessive number of features, which we
will choose at each stage
features_best = []

"""#### Pearson Correlation"""

threshold = 0.9

def highlight(value):
    if value > threshold:
        style = 'background-color: blue'
    else:
        style = 'background-color: black'
    return style

# Absolute value correlation matrix
corr_matrix = new_dataset.corr().abs().round(2)
upper = corr_matrix.where(np.triu(np.ones(corr_matrix.shape),
k=1).astype(np.bool))
upper.style.format("{:.2f}").applymap(highlight)

collinear_features = [column for column in upper.columns if any(upper[column] >
threshold)]
features_filtered = new_dataset.drop(columns = collinear_features)
print('The number of features that passed the collinearity threshold: ',
```

```
features_filtered.shape[1])
features_best.append(features_filtered.columns.tolist())

"""#### Linear SVC"""

lsvc = LinearSVC(C=0.1, penalty="l1", dual=False).fit(train, target)
model = SelectFromModel(lsvc, prefit=True)
X_new = model.transform(train)
X_selected_df = pd.DataFrame(X_new, columns=[train.columns[i] for i in
range(len(train.columns)) if model.get_support()[i]])
features_best.append(X_selected_df.columns.tolist())

"""#### Lasso"""

lasso = LassoCV(cv=3).fit(train, target)
model = SelectFromModel(lasso, prefit=True)
X_new = model.transform(train)
X_selected_df = pd.DataFrame(X_new, columns=[train.columns[i] for i in
range(len(train.columns)) if model.get_support()[i]])
features_best.append(X_selected_df.columns.tolist())

"""#### Regressive Feature Elimination using Logistic Regression"""

rfe_selector = RFE(estimator=LogisticRegression(),
n_features_to_select=num_features_max, step=10, verbose=5)
rfe_selector.fit(train, target)
rfe_support = rfe_selector.get_support()
rfe_feature = train.loc[:,rfe_support].columns.tolist()
features_best.append(rfe_feature)

"""#### Variance Threshold"""

selector = VarianceThreshold(threshold=10)
np.shape(selector.fit_transform(new_dataset))
features_best.append(list(np.array(new_dataset.columns)[selector.get_support(indic
es=False)]))

"""#### Selecting the best features"""

features_best

main_cols = []
main_cols_opt = {feature_name : 0 for feature_name in
new_dataset.columns.tolist()}
for i in range(len(features_best)):
    for feature_name in features_best[i]:
        main_cols_opt[feature_name] += 1
df_main_cols_opt = pd.DataFrame.from_dict(main_cols_opt, orient='index',
columns=['Num'])
df_main_cols_opt.sort_values(by=['Num'],
ascending=False).head(num_features_opt)
```

```
main_cols = df_main_cols_opt.nlargest(num_features_opt, 'Num').index.tolist()
if not 'target' in main_cols:
    main_cols.append('target')
main_cols

"""#### Creating new Dataset (After Feature Selection)"""

data = new_dataset[main_cols]

data.to_csv("/content/drive/MyDrive/Heart Dataset/data.csv")

#test = data.sample(frac = 0.2)

#train = data.drop(test.index)

#test.to_csv("/content/drive/MyDrive/Heart Dataset/test.csv")
#train.to_csv("/content/drive/MyDrive/Heart Dataset/train.csv")

"""## Model Selection using H2O AutoML"""

h2o.init()

train = h2o.import_file("/content/drive/MyDrive/Heart Dataset/train.csv")
test = h2o.import_file("/content/drive/MyDrive/Heart Dataset/test.csv")

# Identify predictors and response
x = train.columns
y = "target"
x.remove(y)
x.remove("C1")

# For binary classification, response should be a factor
train[y] = train[y].asfactor()
test[y] = test[y].asfactor()

# Run AutoML for 20 base models (limited to 1 hour max runtime by default)
aml = H2OAutoML(max_models=20, seed=1)
aml.train(x=x, y=y, training_frame=train)

# View the AutoML Leaderboard
lb = aml.leaderboard
lb.head(rows=lb.nrows)  # Print all rows instead of default (10 rows)

aml.leader

"""## Model (XGBoost)

### Train Test Split
"""
```

```
data = pd.read_csv("/content/drive/MyDrive/Heart Dataset/data.csv")

# Delete the Unnamed :0 etc columns
data.drop(columns=data.columns[[0,1]],
        axis=1,
        inplace=True)

X = data.iloc[:,:-1]
Y = data.target

seed = 7
test_size = 0.33
X_train, X_test, y_train, y_test = train_test_split(X, Y, test_size=test_size,
random_state=seed)

"""### Training model

#### Parameter Tuning
"""

param_grid = {
    "max_depth": [3, 4, 5, 7],
    "learning_rate": [0.1, 0.01, 0.05],
    "gamma": [0, 0.25, 1],
    "reg_lambda": [0, 1, 10],
    "scale_pos_weight": [1, 3, 5],
    "subsample": [0.8],
    "colsample_bytree": [0.5],
}

model_xgb = XGBClassifier()
grid_cv = GridSearchCV(model_xgb, param_grid, n_jobs=-1, cv=3,
scoring="roc_auc")
_ = grid_cv.fit(X_train, y_train)

print(grid_cv.best_score_)
print(grid_cv.best_params_)

"""#### Model Training"""

final_model_xgb = XGBClassifier(
    **grid_cv.best_params_,
    objective="binary:logistic"
)

"""##### Learning Curve"""

skplt.estimators.plot_learning_curve(final_model_xgb, X_train, y_train)
plt.show()

"""##### Model Fitting"""
```

```
final_model_xgb.fit(X_train, y_train)
print(final_model_xgb)

"""##### Feature Importance Plot"""

skplt.estimators.plot_feature_importances(final_model_xgb,
feature_names=X_train.columns, x_tick_rotation=80)
plt.show()

"""### Model Prediction"""

# make predictions for test data
y_pred_xgb = final_model_xgb.predict(X_test)
y_pred_proba_xgb = final_model_xgb.predict_proba(X_test)
predictions_xgb = [round(value) for value in y_pred_xgb]

"""### Performance Evaluation"""

evaluation(y_test,y_pred_xgb)

"""#### Accuracy"""

accuracy_xgb = accuracy_score(y_test, predictions_xgb)
print("Accuracy: %.2f%%" % (accuracy_xgb * 100.0))

"""#### Kappa Score"""

cks_xgb = cohen_kappa_score(y_test, y_pred_xgb)
print("Cohen Kappa Score: ",cks_xgb)

"""#### Log Loss"""

logloss_xgb = log_loss(y_test, y_pred_proba_xgb)
print("Log Loss: ",logloss_xgb)

"""#### KS Statistics"""

res_xgb = binary_ks_curve(y_test, y_pred_xgb)
ks_stat_xgb = res_xgb[3]
print("KS Statistic: ", ks_stat_xgb)

"""#### Classification Report (Precision, Recall, F1- Score)"""

print(classification_report(y_test, predictions_xgb))

"""#### Confusion Matrix"""

plot_confusion_matrix(final_model_xgb, X_test, y_test)
plt.show()
```

```
"""#### ROC Curve"""

plot_roc_curve(final_model_xgb, X_test, y_test)
plt.show()

"""#### Kolmogorov-Smirnov plot"""

fig, ax = plt.subplots()
plot_ks_statistic(y_test, y_pred_proba_xgb, ax=ax)

"""#### Precision-Recall Curve"""

fig, ax = plt.subplots()
plot_precision_recall(y_test, y_pred_proba_xgb, ax=ax)

"""## Model (GBM)

### Train Test Split
"""

data = pd.read_csv("/content/drive/MyDrive/Heart Dataset/data.csv")

# Delete the Unnamed :0 etc columns
data.drop(columns=data.columns[[0,1]],
        axis=1,
        inplace=True)

X = data.iloc[:,:-1]
Y = data.target

seed = 7
test_size = 0.33
X_train, X_test, y_train, y_test = train_test_split(X, Y, test_size=test_size,
random_state=seed)

"""### Training model

#### Parameter Tuning
"""

grid = dict()
grid['n_estimators'] = [10, 50, 100, 500]
grid['learning_rate'] = [0.0001, 0.001, 0.01, 0.1, 1.0]
grid['subsample'] = [0.5, 0.7, 1.0]
grid['max_depth'] = [3, 7, 9]

model_gbm = GradientBoostingClassifier()

grid_cv_gbm = GridSearchCV(model_gbm, grid, n_jobs=-1, cv=3, scoring="roc_auc")
_ = grid_cv_gbm.fit(X_train, y_train)
```

```
print(grid_cv_gbm.best_score_)
print(grid_cv_gbm.best_params_)

"""#### Model Training"""

final_model_gbm = GradientBoostingClassifier(**grid_cv_gbm.best_params_)

"""##### Learning Curve"""

skplt.estimators.plot_learning_curve(final_model_gbm, X_train, y_train)
plt.show()

"""##### Model Fitting"""

final_model_gbm.fit(X_train, y_train)
print(final_model_gbm)

"""##### Feature Importance Plot"""

skplt.estimators.plot_feature_importances(final_model_gbm,
feature_names=X_train.columns, x_tick_rotation=80)
plt.show()

"""### Model Prediction"""

# make predictions for test data
y_pred_gbm = final_model_gbm.predict(X_test)
y_pred_proba_gbm = final_model_gbm.predict_proba(X_test)
predictions_gbm = [round(value) for value in y_pred_gbm]

"""### Performance Evaluation"""

evaluation(y_test,y_pred_gbm)

"""#### Accuracy"""

accuracy_gbm = accuracy_score(y_test, y_pred_gbm)
print("Accuracy: ", (accuracy_gbm * 100.0))

"""#### Kappa Score"""

cks_gbm = cohen_kappa_score(y_test, y_pred_gbm)
print("Cohen Kappa Score: ",cks_gbm)

"""#### Log Loss"""

logloss_gbm = log_loss(y_test, y_pred_proba_gbm)
print("Log Loss: ",logloss_gbm)

"""#### KS Statistics"""
```

```
res_gbm = binary_ks_curve(y_test, y_pred_gbm)
ks_stat_gbm = res_gbm[3]
print("KS Statistic: ", ks_stat_gbm)

"""#### Classification Report (Precision, Recall, F1- Score)"""

print(classification_report(y_test, predictions_gbm))

"""#### Confusion Matrix"""

plot_confusion_matrix(final_model_gbm, X_test, y_test)
plt.show()

"""#### ROC Curve"""

plot_roc_curve(final_model_gbm, X_test, y_test)
plt.show()

"""#### Kolmogorov-Smirnov plot"""

fig, ax = plt.subplots()
plot_ks_statistic(y_test, y_pred_proba_gbm, ax=ax)

"""#### Precision-Recall Curve"""

fig, ax = plt.subplots()
plot_precision_recall(y_test, y_pred_proba_gbm, ax=ax)
```

Testing and Evaluation

Mean Squared Error (MSE)

The Mean Squared Error (MSE) is perhaps the simplest and most common loss function, often taught in introductory Machine Learning courses. To calculate the MSE, you take the difference between your model's predictions and the ground truth, square it, and average it out across the whole dataset.

The MSE will never be negative, since we are always squaring the errors. The MSE is formally defined by the following equation:

$$\text{MSE} = \frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2$$

Where *N* is the number of samples we are testing against. The code is simple enough, we can write it in plain numpy and plot it using matplotlib:

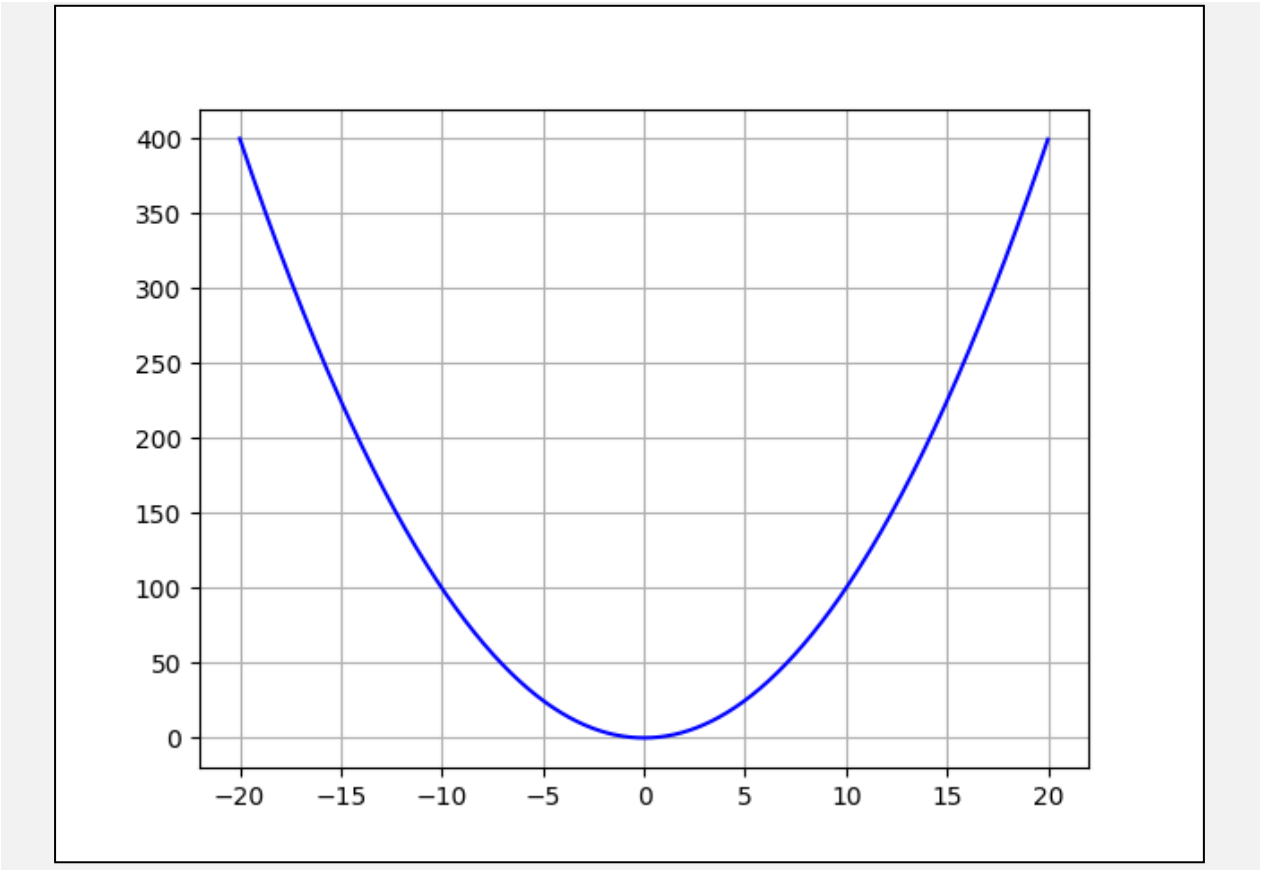


Fig 8:MSE Loss Function

Advantage: The MSE is great for ensuring that our trained model has no outlier predictions with huge errors, since the MSE puts larger weight on theses errors due to the squaring part of the function.

Disadvantage: If our model makes a single very bad prediction, the squaring part of the function magnifies the error. Yet in many practical cases we don't care much about these outliers and are aiming for more of a well-rounded model that performs good enough on the majority.

Root Mean Square Error

Root Mean Square Error (RMSE) is a standard way to measure the error of a model in predicting quantitative data. Formally it is defined as follows:

$$RMSE = \sqrt{\sum_{i=1}^{n} \frac{(\hat{y}_i - y_i)^2}{n}}$$

$\hat{y}_1, \hat{y}_2, \ldots, \hat{y}_n$ are predicted values

$y_1, y_2, \ldots, y_n$ are observed values

n is the number of observations

Let's try to explore why this measure of error makes sense from a mathematical perspective. Ignoring the division by n under the square root, the first thing we can notice is a resemblance to the formula for the Euclidean distance between two vectors in $\mathbb{R}^n$:

$$distance(\mathbf{x}, \mathbf{y}) = \sqrt{\sum_{i=1}^{n}(x_i - y_i)^2}$$

This tells us heuristically that RMSE can be thought of as some kind of (normalized) distance between the vector of predicted values and the vector of observed values.

But why are we dividing by n under the square root here? If we keep n (the number of observations) fixed, all it does is rescale the Euclidean distance by a factor of √(1/n). It's a bit tricky to see why this is the right thing to do, so let's delve in a bit deeper.

Imagine that our observed values are determined by adding random "errors" to each of the predicted values, as follows:

$$y_i = \hat{y}_i + \epsilon_i \text{ for } i = 1, \ldots, n$$

$\epsilon_1, \ldots, \epsilon_n$ independent, identically distributed errors

These errors, thought of as random variables, might have Gaussian distribution with mean μ and standard deviation σ, but any other distribution with a square-integrable PDF (*probability density function*) would also work.

We want to think of $\hat{y}_i$ as an underlying physical quantity, such as the exact distance from Mars to the Sun at a particular point in time. Our observed quantity y_i would then be the distance from Mars to the Sun *as we measure it,* with some errors coming from mis-calibration of our telescopes and measurement noise from atmospheric interference.The mean μ of the distribution of our errors would

correspond to a persistent bias coming from mis-calibration, while the standard deviation σ would correspond to the amount of measurement noise. Imagine now that we know the mean μ of the distribution for our errors exactly and would like to estimate the standard deviation σ. We can see through a bit of calculation that:

$$\begin{aligned}
&\mathbb{E}\left[\frac{\sum_{i=1}^{n}(\hat{y}_i - y_i)^2}{n}\right] \\
&= \mathbb{E}\left[\frac{\sum_{i=1}^{n}\epsilon_i^2}{n}\right] \\
&= \frac{1}{n}\sum_{i=1}^{n}\mathbb{E}[\epsilon_i^2] \\
&= \mathbb{E}[\epsilon^2] \\
&= Var(\epsilon) + \mathbb{E}[\epsilon]^2 \\
&= \sigma^2 + \mu^2
\end{aligned}$$

Here **E**[…] is the expectation, and Var(…) is the variance. We can replace the average of the expectations **E[**ε_i^2] on the third line with the **E**[ε^2] on the fourth line where ε is a variable with the same distribution as each of the ε_i, because the errors ε_i are identically distributed, and thus their squares all have the same expectation.

Remember that we assumed we already knew μ exactly. That is, the persistent bias in our instruments is a known bias, rather than an unknown bias. So we might as well correct for this bias right off the bat by subtracting μ from all our raw observations. That is, we might as well suppose our errors are already distributed with mean μ = 0. Plugging this into the equation above and taking the square root of both sides then yields:

$$\sqrt{\mathbb{E}\left[\frac{\sum_{i=1}^{n}(\hat{y}_i - y_i)^2}{n}\right]} = \sqrt{\sigma^2 + 0^2} = \sigma$$

To sum up our discussion, RMSE is a good measure to use if we want to estimate the standard deviation σ of a typical observed value from our model's prediction, assuming that our observed data can be decomposed as:

observed value = predicted value + predictably distributed random noise with mean zero

The random noise here could be anything that our model does not capture (e.g., unknown variables that might influence the observed values). If the noise is small, as estimated by RMSE, this generally means our model is good at predicting our

observed data, and if RMSE is large, this generally means our model is failing to account for important features underlying our data.

Log Loss

Log Loss is the most important classification metric based on probabilities.

It's hard to interpret raw log-loss values, but log-loss is still a good metric for comparing models. For any given problem, a lower log-loss value means better predictions. Log Loss is a slight twist on something called the **Likelihood Function**. In fact, Log Loss is -1 * the log of the likelihood function. So, we will start by understanding the likelihood function.

The likelihood function answers the question "How likely did the model think the actually observed set of outcomes was." If that sounds confusing, an example should help.

Example

A model predicts probabilities of [0.8, 0.4, 0.1] for three houses. The first two houses were sold, and the last one was not sold. So the actual outcomes could be represented numeically as [1, 1, 0].

Let's step through these predictions one at a time to iteratively calculate the likelihood function.

- The first house sold, and the model said that was 80% likely. So, the likelihood function after looking at one prediction is 0.8.
- The second house sold, and the model said that was 40% likely. There is a rule of probability that the probability of multiple independent events is the product of their individual probabilities. So, we get the combined likelihood from the first two predictions by multiplying their associated probabilities. That is 0.8 * 0.4, which happens to be 0.32.
- Now we get to our third prediction. That home did not sell. The model said it was 10% likely to sell. That means it was 90% likely to not sell. So, the observed outcome of *not selling* was 90% likely according to the model. So, we multiply the previous result of 0.32 by 0.9.
- We could step through all of our predictions. Each time we'd find the probability associated with the outcome that actually occurred, and we'd multiply that by the previous result. That's the likelihood.

From Likelihood to Log Loss

Each prediction is between 0 and 1. If you multiply enough numbers in this range,

the result gets so small that computers can't keep track of it. So, as a clever computational trick, we instead keep track of the log of the Likelihood. This is in a range that's easy to keep track of. We multiply this by negative 1 to maintain a common convention that lower loss scores are better.

ROC-AUC

AUC - ROC curve is a performance measurement for the classification problems at various threshold settings. ROC is a probability curve and AUC represents the degree or measure of separability. It tells how much the model is capable of distinguishing between classes. Higher the AUC, the better the model is at predicting 0 classes as 0 and 1 classes as 1. By analogy, the Higher the AUC, the better the model is at distinguishing between patients with the disease and no disease.

The ROC curve is plotted with TPR against the FPR where TPR is on the y-axis and FPR is on the x-axis.

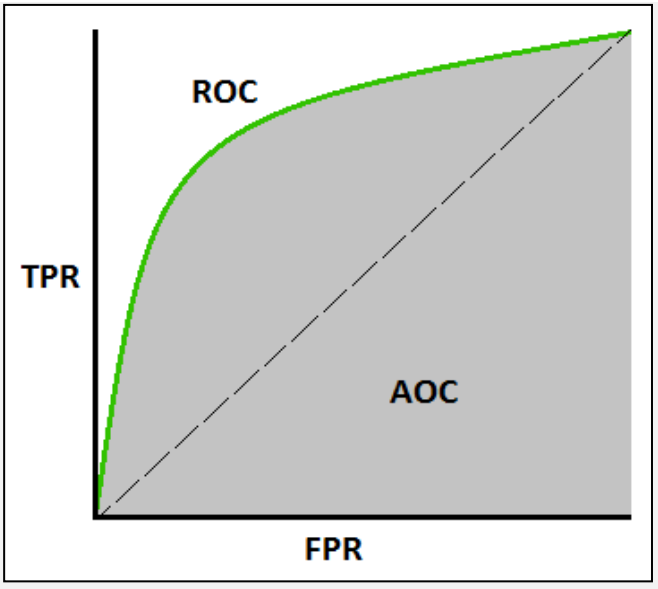


Fig 9: AUC - ROC Curve

Defining terms used in AUC and ROC Curve.

TPR (True Positive Rate) / Recall /Sensitivity

$$\text{TPR /Recall / Sensitivity} = \frac{TP}{TP + FN}$$

Specificity

$$\text{Specificity} = \frac{TN}{TN + FP}$$

FPR

$$FPR = 1 - \text{Specificity} = \frac{FP}{TN + FP}$$

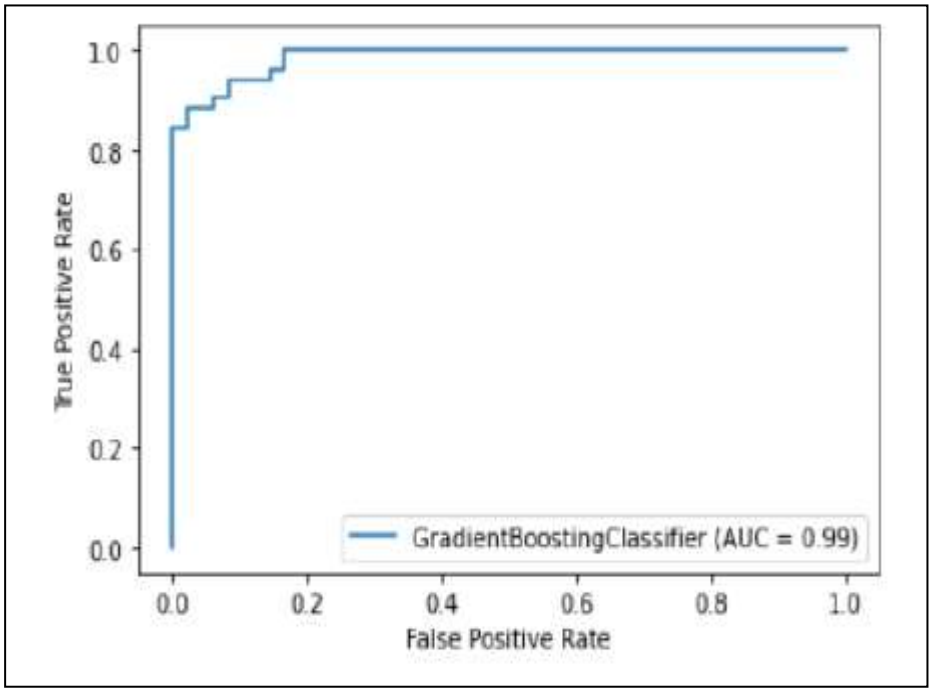


Fig 10:ROC Curve on Test Data

AUCPR

The area under the precision-recall curve (AUPRC) is a useful performance metric for imbalanced data in a problem setting where you care a lot about finding the positive examples. For example, perhaps you are building a classifier to detect pneumothorax in chest x-rays, and you want to ensure that you find all the pneumothoraces without incorrectly marking healthy lungs as positive for pneumothorax.

If your model achieves a perfect AUPRC, it means your model found all of the positive examples/pneumothorax patients (perfect recall) without accidentally marking any negative examples/healthy patients as positive (perfect precision). The

“average precision” is one particular method for calculating the AUPRC.

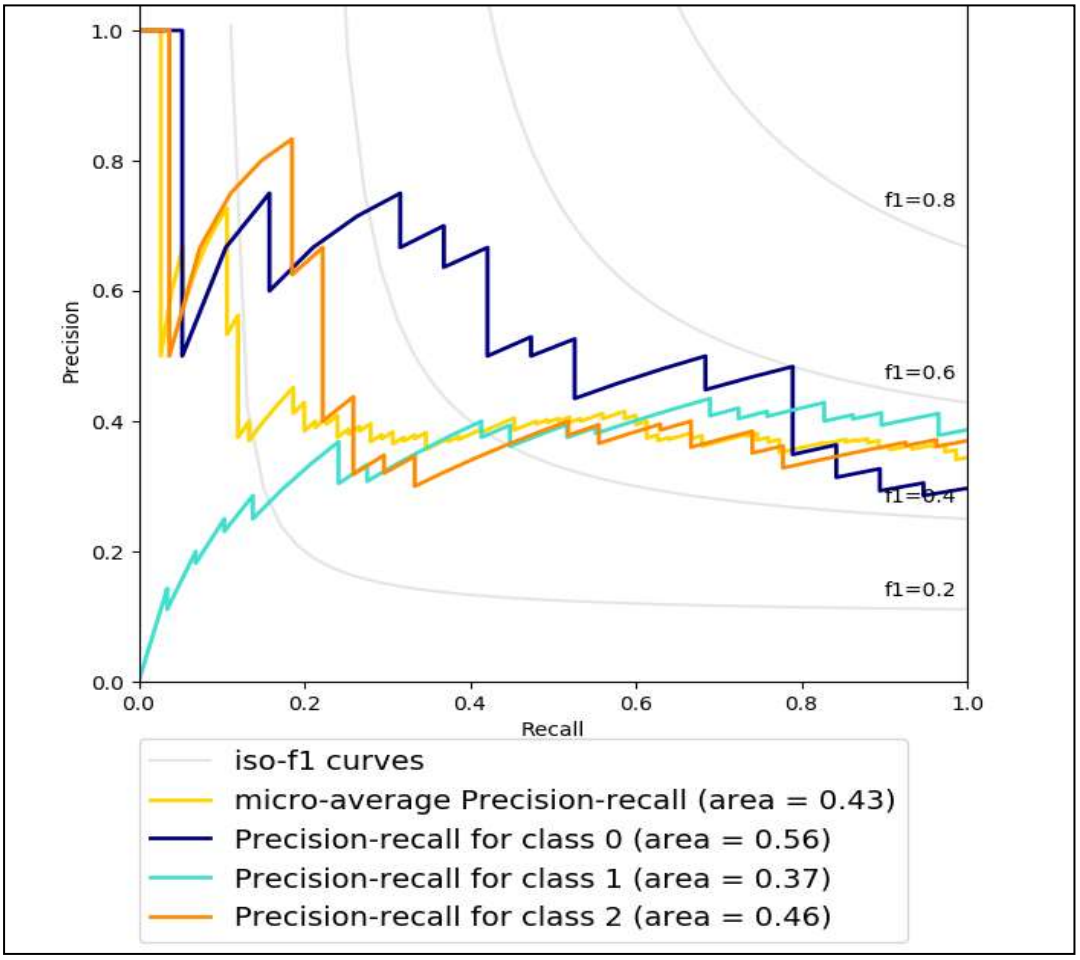


Fig 11: Extension of Precison-Recall curve to multi class

Gini

Gini Index, also known as Gini impurity, **calculates the amount of probability of a specific feature that is classified incorrectly when selected randomly.** If all the elements are linked with a single class then it can be called pure.

Let’s perceive the criterion of the Gini Index, like the properties of entropy, the **Gini index varies between values 0 and 1**, where 0 expresses the purity of classification, i.e. All the elements belong to a specified class or only one class exists there. And 1 indicates the random distribution of elements across various classes. The value of 0.5 of the Gini Index shows an equal distribution of elements over some classes.

While designing the decision tree, the features possessing the least value of the Gini Index would get preferred. You can learn another tree-based algorithm.The Gini Index is determined by deducting the sum of squared of probabilities of each class from one, mathematically, Gini Index can be expressed as:

$$\text{Gini Index} = 1 - \sum_{i=1}^{n} (P_i)^2$$

where P_i denotes the probability of an element being classified for a distinct class.

Classification and Regression Tree (CART) algorithm deploys the method of the Gini Index to originate binary splits.

In addition, decision tree algorithms exploit Information Gain to divide a node and Gini Index or Entropy is the passageway to weigh the Information Gain. Gini index and Information Gain are used for the analysis of the real-time scenario, and data is real that is captured from the real-time analysis. In numerous definitions, it has also been mentioned as **"impurity of data" or "how data is distributed**. So we can calculate which data is taking less or more part in decision making.

Accuracy

Accuracy is one metric for evaluating classification models.
Informally, **accuracy** is the fraction of predictions our model got right. Formally, accuracy has the following definition:

Accuracy=Number of correct predictions/Total number of predictions

For binary classification, accuracy can also be calculated in terms of positives and negatives as follows:

Accuracy=TP+TNTP+TN+FP+FN

Where *TP* = True Positives, *TN* = True Negatives, *FP* = False Positives, and *FN* = False Negatives.

Let's try calculating accuracy for the following model that classified 100 tumors as malignant (the positive class) or benign (the negative class):

True Positive (TP):	False Positive (FP):
Reality: Malignant	Reality: Benign
ML model predicted: Malignant	ML model predicted: Malignant
Number of TP results: 1	Number of FP results: 1

False Negative (FN): Reality: Malignant ML model predicted: Benign Number of FN results: 8	True Negative (TN): Reality: Benign ML model predicted: Benign Number of TN results: 90

Accuracy=TP+TNTP+TN+FP+FN=1+901+90+1+8=0.91

Accuracy comes out to 0.91, or 91% (91 correct predictions out of 100 total examples). That means our tumor classifier is doing a great job of identifying malignancies, right?

Actually, let's do a closer analysis of positives and negatives to gain more insight into our model's performance.

Of the 100 tumor examples, 91 are benign (90 TNs and 1 FP) and 9 are malignant (1 TP and 8 FNs). Of the 91 benign tumors, the model correctly identifies 90 as benign. That's good. However, of the 9 malignant tumors, the model only correctly identifies 1 as malignant—a terrible outcome, as 8 out of 9 malignancies go undiagnosed!

While 91% accuracy may seem good at first glance, another tumor-classifier model that always predicts benign would achieve the exact same accuracy (91/100 correct predictions) on our examples. In other words, our model is no better than one that has zero predictive ability to distinguish malignant tumors from benign tumors.

Precision

In the simplest terms, Precision is the ratio between the True Positives and all the Positives. For our problem statement, that would be the measure of patients that we correctly identify having a heart disease out of all the patients actually having it. Mathematically:

$$Precision = \frac{True\ Positive(TP)}{True\ Positive(TP) + False\ Positive(FP)}$$

What is the Precision for our model? Yes, it is 0.843 or, when it predicts that a patient has heart disease, it is correct around 84% of the time.

Precision also gives us a measure of the relevant data points. It is important that we don’t start treating a patient who actually doesn’t have a heart ailment, but our model predicted as having it.

Recall

The recall is the measure of our model correctly identifying True Positives. Thus, for all the patients who actually have heart disease, recall tells us how many we correctly identified as having a heart disease. Mathematically:

$$Recall = \frac{True\ Positive(TP)}{True\ Positive(TP) + False\ Negative(FN)}$$

For our model, Recall = 0.86. Recall also gives a measure of how accurately our model is able to identify the relevant data. We refer to it as Sensitivity or True Positive Rate. What if a patient has heart disease, but there is no treatment given to him/her because our model predicted so? That is a situation we would like to avoid!

Confusion Matrix

For imbalanced classification problems, the majority class is typically referred to as the negative outcome (e.g. such as "*no change*" or "*negative test result*"), and the minority class is typically referred to as the positive outcome (e.g. "change" or "positive test result").

The confusion matrix provides more insight into not only the performance of a predictive model, but also which classes are being predicted correctly, which incorrectly, and what type of errors are being made.
The simplest confusion matrix is for a two-class classification problem, with negative (class 0) and positive (class 1) classes.
In this type of confusion matrix, each cell in the table has a specific and well-understood name, summarized as follows:

1 | Positive Prediction | Negative Prediction
2 Positive Class | True Positive (TP) | False Negative (FN)
3 Negative Class | False Positive (FP) | True Negative (TN)

The precision and recall metrics are defined in terms of the cells in the confusion matrix, specifically terms like true positives and false negatives.

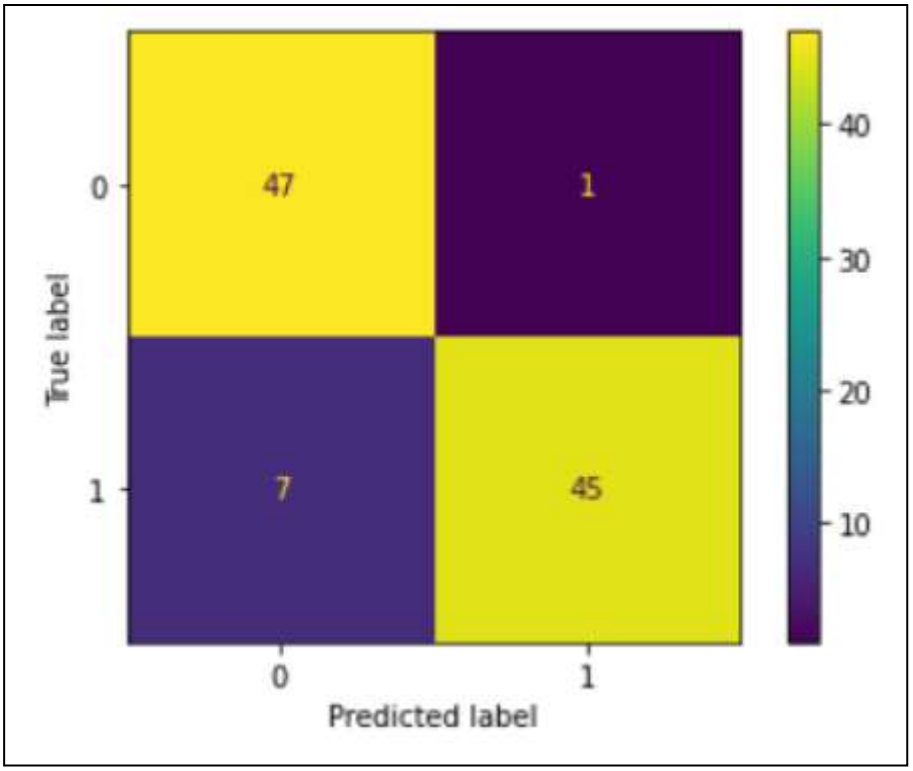


Fig 12: Confusion Matrix on Test Data

Evaluation

```
accuracy_gbm = accuracy_score(y_test, y_pred_gbm)
print("Accuracy: ", (accuracy_gbm * 100.0))
```

```
Accuracy:  95.0
```

Fig. 13 Accuracy

```
cks_gbm = cohen_kappa_score(y_test, y_pred_gbm)
print("Cohen Kappa Score: ",cks_gbm)
```

```
Cohen Kappa Score:  0.899919935948759
```

Fig. 2 Cohen Kappa Score

```
logloss_gbm = log_loss(y_test, y_pred_proba_gbm)
print("Log Loss: ",logloss_gbm)
```

```
Log Loss:  1.1988248664263135
```

Fig. 3 Log Loss

```
res_gbm = binary_ks_curve(y_test, y_pred_gbm)
ks_stat_gbm = res_gbm[3]
print("KS Statistic: ", ks_stat_gbm)
```

```
KS Statistic:  0.9006410256410257
```

Fig. 4 KS Statistics

```
print(classification_report(y_test, predictions_gbm))
```

	precision	recall	f1-score	support
0	0.94	0.96	0.95	48
1	0.96	0.94	0.95	52
accuracy			0.95	100
macro avg	0.95	0.95	0.95	100
weighted avg	0.95	0.95	0.95	100

Fig. 5 Classification Report

```
plot_confusion_matrix(final_model_gbm, X_test, y_test)
plt.show()
```

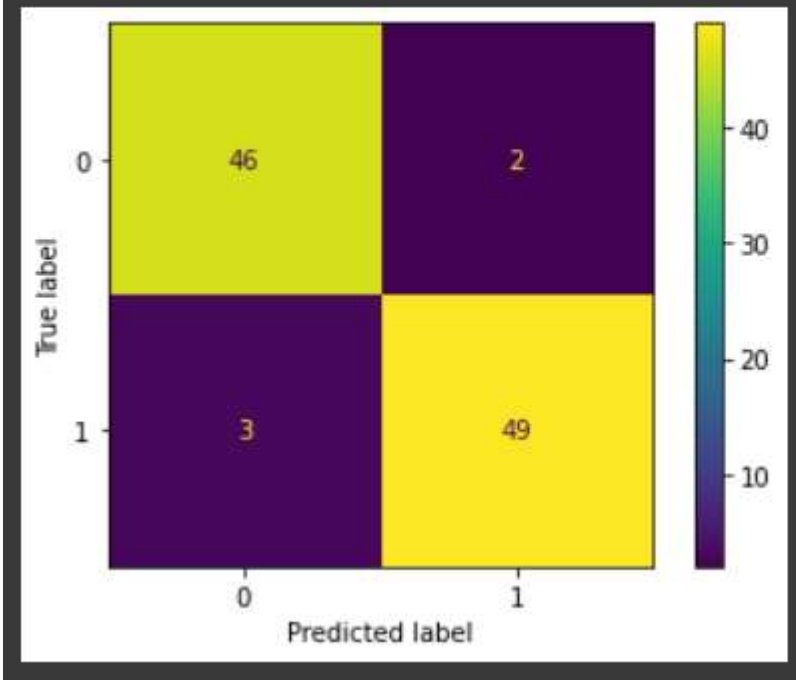


Fig. 6 Confusion Matrix

```
plot_roc_curve(final_model_gbm, X_test, y_test)
plt.show()
```

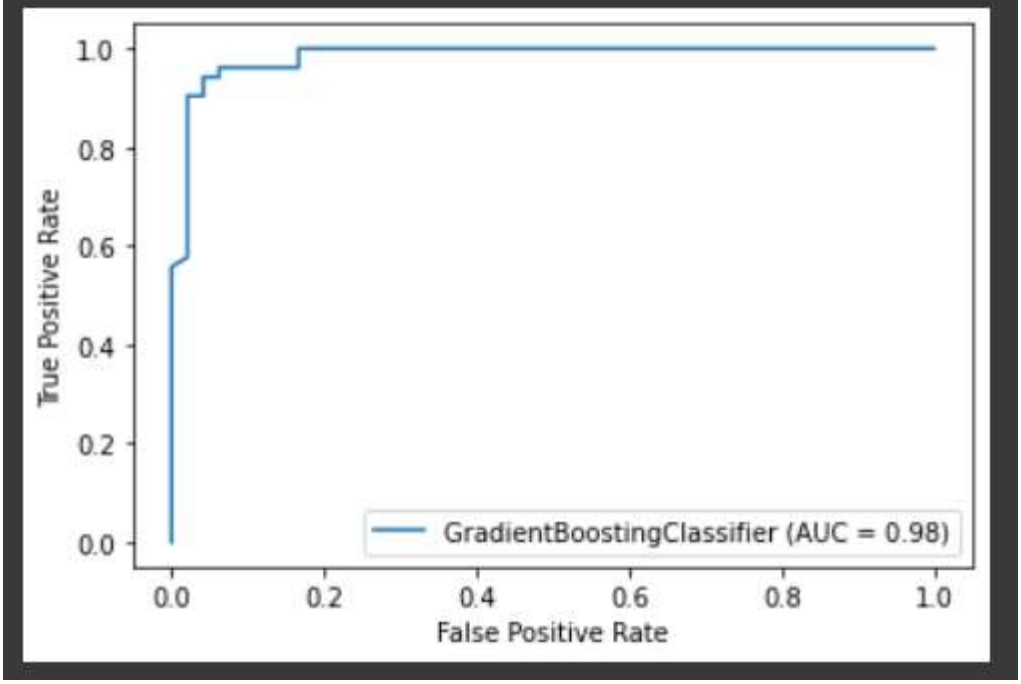


Fig. 7 ROC Curve

```
fig, ax = plt.subplots()
plot_ks_statistic(y_test, y_pred_proba_gbm, ax=ax)
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x7ff81ea63890>
```

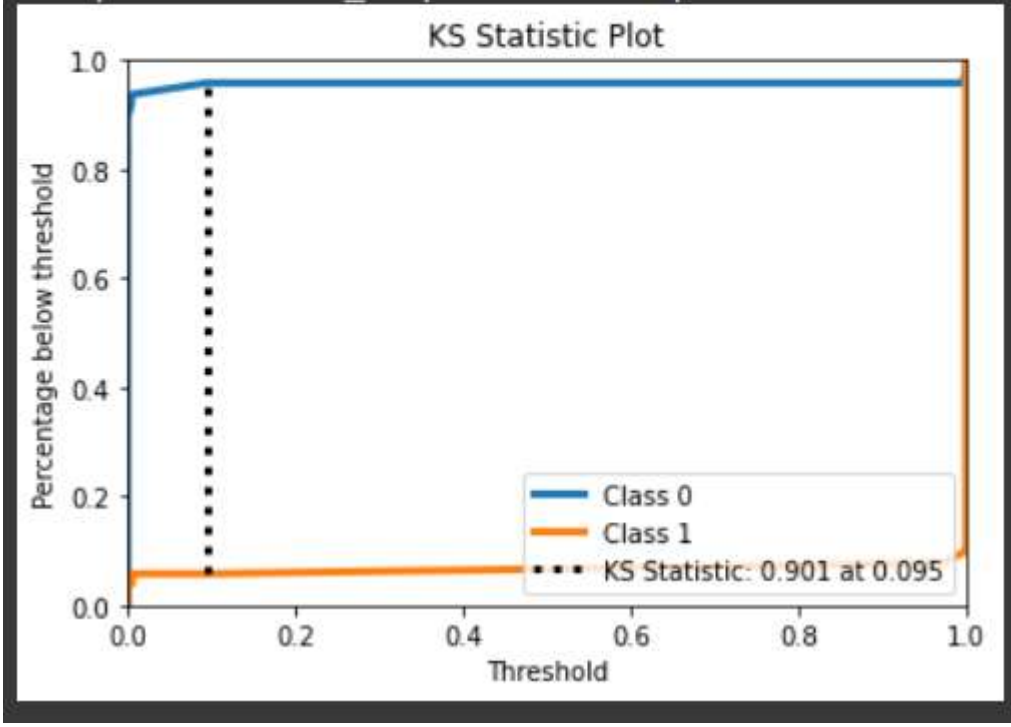


Fig. 8 KS Statistics Plot

```
fig, ax = plt.subplots()
plot_precision_recall(y_test, y_pred_proba_gbm, ax=ax)
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x7ff81e9dced0>
```

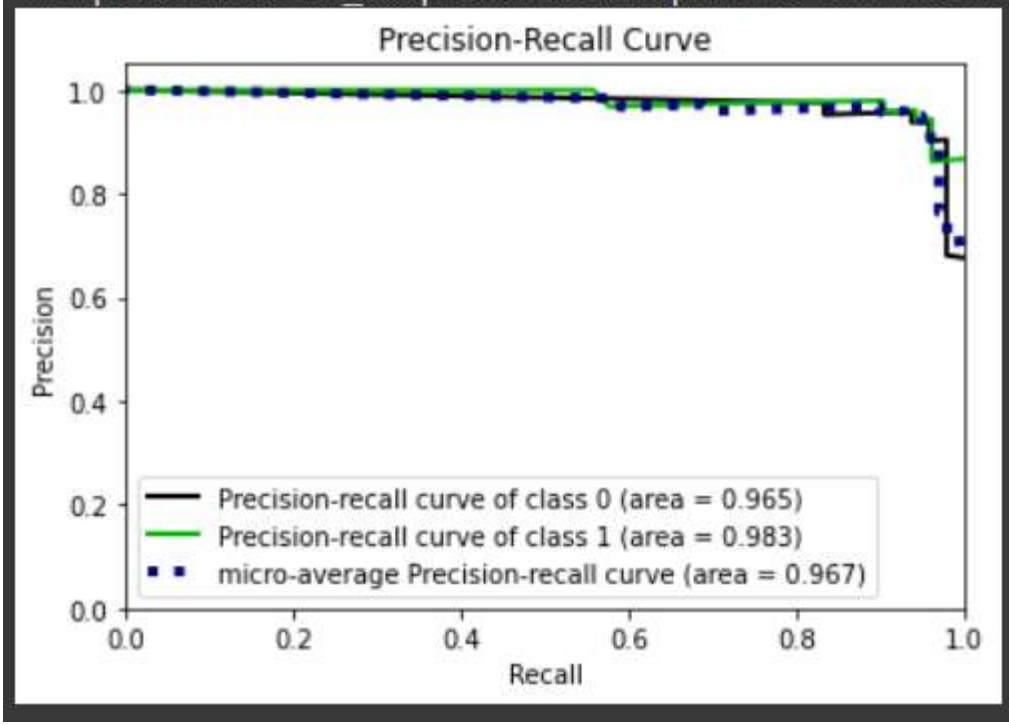


Fig. 9 Precision Recall Plot

Conclusion

The aim of this project was to create a system that was able to detect whether a patient has a heart disease or not. Our first concern was to transform the dataset with less correlations. This led us to transform some of our features using Weight of Evidence and k-fold Target encoders. Using multiple feature selection algorithms such as the Logistic Regression and Pearson correlation, top 15 features were extracted from all the new and old features. H2O AutoML was used for the automatic model selection process which gave the gradient boosting machine as the best classifier for the problem.

After fitting the model on training dataset, the model was tested on a previously unseen testing data. The accuracy on the testing set was found to be 93% while the other metrics such as precision score, f1 score, recall score and kappa score was found to be 92.5%, 93.3%, 94.2% and 84.05% respectively.

References

[1] A. Pattekari, Shadab Adam Parveen, "PREDICTION SYSTEM FOR HEART DISEASE USING NAIVE BAYES," *Int. J. Adv. Comput. Math. Sci.*, vol. 3, no. 3, pp. 290–294, 2012, doi: 10.1.1.1089.1654.

[2] Z. Arabasadi, R. Alizadehsani, M. Roshanzamir, H. Moosaei, and A. A. Yarifard, "Computer aided decision making for heart disease detection using hybrid neural network-Genetic algorithm," *Comput. Methods Programs Biomed.*, vol. 141, pp. 19– 26, 2017, doi: https://doi.org/10.1016/j.cmpb.2017.01.004.

[3] M. H. I. Shadman Nashif, Md. Rakib Raihan, Md. Rasedul Islam, "Heart Disease Detection by Using Machine Learning Algorithms and a Real-Time Cardiovascular Health Monitoring System," *World J. Eng. Technol.*, vol. 6, pp. 854–873, 2018, doi: 10.4236/wjet.2018.64057.

[4] R. Atallah and A. Al-Mousa, "Heart Disease Detection Using Machine Learning Majority Voting Ensemble Method," in *2019 2nd International Conference on new Trends in Computing Sciences (ICTCS)*, 2019, pp. 1–6, doi: 10.1109/ICTCS.2019.8923053.

[5] R. Buettner and M. Schunter, "Efficient machine learning based detection of heart disease," in *2019 IEEE International Conference on E-health Networking, Application Services (HealthCom)*, 2019, pp. 1–6, doi: 10.1109/HealthCom46333.2019.9009429.

[6] A. U. L. HAQ *et al.*, "Identifying the Predictive Capability of Machine Learning
Classifiers for Designing Heart Disease Detection System," in *2019 16th International Computer Conference on Wavelet Active Media Technology and Information Processing*, 2019, pp. 130–138, doi: 10.1109/ICCWAMTIP47768.2019.9067519.

[7] Z. Arabasdi *et al.,* "Computer aided decision making for heart disease detection using hybrid neural network-Genetic algorithm," in "*2017 Computer Methods and Programs in Biomedicine*", 2017, vol. 141, pp. 19-26, doi: 10.1016/j.cmpb.2017.01.004.

[8] A. Rajkumar and G. Sophia Reena, "Diagnosis Of Heart Disease Using Data mining Algorithm," in *2010 Global Journal of Computer Science and Technology,* 2010, vol. 10, pp. 38-43, url: https://computerresearch.org/index.php/computer/article/view/102