Berkay Altıntaş

22002709

EEE 485 – THE FINAL REPORT OF HEART DISEASE PREDICTION PROJECT

INTRODUCTION

According to the report of World Health Organization (WHO), cardiovascular diseases are the leading cause of death globally [1]. Since 2019, approximately 4.2 million people in Europe died from cardiovascular diseases. The situation has no difference with Turkey. Approximately 40% of the death in Turkey are due to cardiovascular disease [2]. Therefore, heart diseases need more effective solutions. It is important to detect heart disease as early as possble [3]. Thus, detection capability of machine learning and deep learning algorithms may offer a early diagnosis opportunity.

DATASET DESCRIPTION

In this project, since the heart disease is number one problem in health topics, it is aimed to detect and classify the heart attack. To be more precise, it is aimed that whether a person has heart disease or not. To do so, 'Heart Disease Cleveland UCI' is chosen as dataset to work on from Kaggle (https://www.kaggle.com/datasets/cherngs/heart-disease-clevelanduci). In this dataset, there are 297 different data wih 13 features. These features are

1- Age : Age in years

2- Sex: 1 for Male, 0 for female

3- Chest Pain Type (cp): There are 4 values. 0 for typical angina, 1 for atypical angina, 2 for non-anginal pain and 3 for asymptomatic.

4- Trestbps: Resting blood pressure in terms of mm Hg.

5- Chol: Serum cholestoral in terms of mg/dl

6- Fbs: Fasting blood sugar. It has 2 values. If it is greater than 120 mg/dl it represented as 1, otherwise 0.

7- Restecg: Resting electrocardiographic results. It get 3 values. 0 for normal, 1 for having ST-T wave abnormality and 2 for showing probable or definite left ventricular hypertrophy.

8- Thalach: Maximum heart rate achieved

9- Exang: Exercise induced angina. It has 2 values. 1 for yes, 0 for no.

10- Oldpeak: ST deprresion induced by exercise relative to rest

11- Slope: The slope of the peak exercise ST segment. It has 3 values. 0 for upsloping, 1 for flat and 2 for downsloping.

12- Ca: Number of major vessels. It gets values in the range of 0 to 3.

13- Thal: Thallium stress results. It get 3 values. 0 for normal, 1 for fixed defect and 2 for reversable defect

Regarding these features, data is labeled as 1 if it has disease and is labeled as 0 if it has no disease.

To understand what the features include, their range, their type and their correlation with each other, some visualization techniques were used for data understanding and data preprocessing stage in Python via Seaborn, Numpy Matplotlib and Pandas libraries. In Figure 1, the correlation matrix which represent the relationships between the features in entire dataset could be seen. In Figure 2, it could be seen that we have both categorical and numerical data together. Moreover, the numerical data for different features varied in different ranges which might affect the performance of the machine learning algorithms. Regarding these, firstly, the categorical data for different features were represented as one-hot encoded form. Moreover, to arrange the wide ranges of numerical data, standartization technique which ensures better optimizaton was used. The formula for standartization was as follows,

$$z_i = \frac{x_i - mean(x)}{std(x)}$$

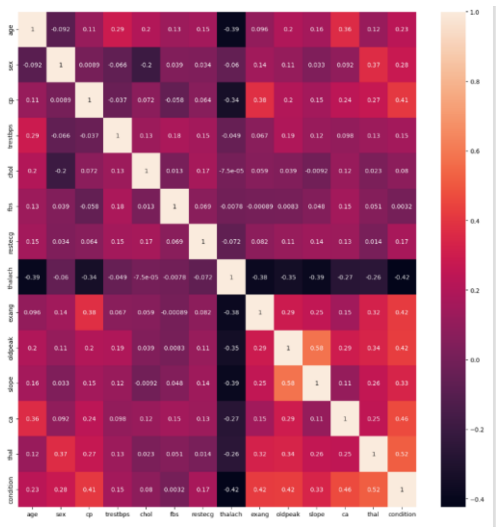
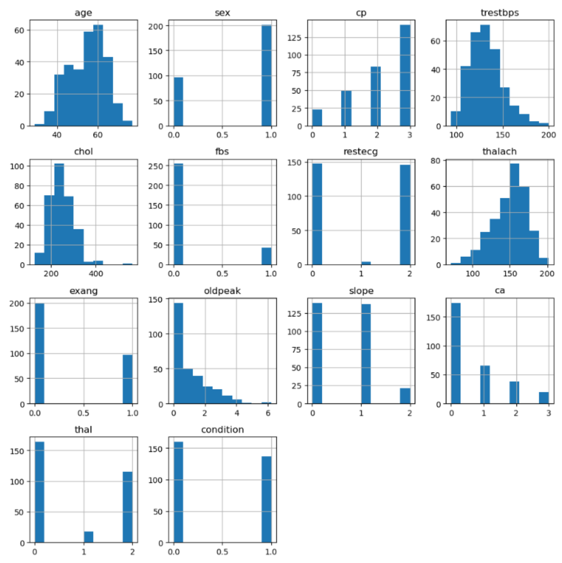


Figure 1: Correlation Matrix of Features



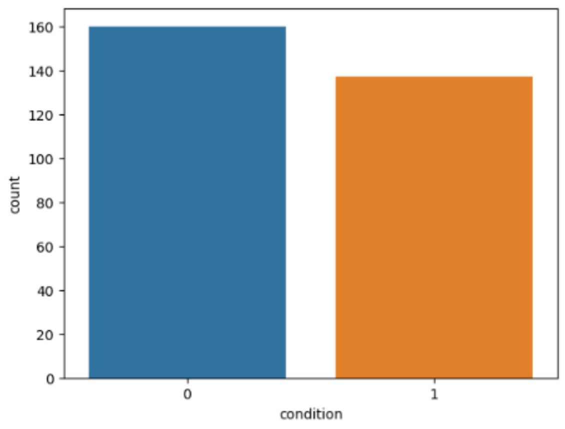


Figure 2: Distribution of Entire Dataset **Figure 3:** Distribution of Diagnosis

After completed the data understanding, visualization and standartization part, I have splitted the data set as %85 training and %15 testing without cross validation cases regarding the sample size I had. For the cases I used cross validation, I have splitted the dataset as %70 training, %15 validation and %15 testing.

REVIEW OF MACHINE LEARNING ALGORITHMS

The main aim of this project is to classify whether the patient has heart disease or not. Therefore, this goal could be done using binary classification approaches and algorithms. Therefore, it was planned to use logistic regression, multi layer perceptron and decision trees which have different approaches and implementation difficulties.

Logistic Regression

The first machine learning algorithm I have chosen was logistic regression due to its capability of binary classification. Since this tasks required to classify whether the patient has heart disease or not, the logistic regression algorithm fitted this job. Logistic regression is not only a simple model but also an efficent model. More specifically, it creates a linear decision boundary which seperated the linearly related data. Therefore, it works well if the classes are linearly seperable and the relationship between the features and log-odds of the outcomes are linear. The idea behind the logistic regression can be seen in Figure 4.

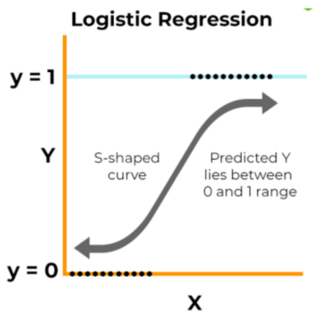


Figure 4: Idea Behind the Logistic Regression [4]

Moreover, the mathematics behind the logistic regression can be seen below. The formulation becomes the idea of the probabilities of being two different classes which their probabilies add up to 1. By using the maximum likelihood estimation, this problem can be converted into minimization problem. After defining a minimization problem, by using gradient descent algorithm, the optimal parameters can be found for this task. All the related math and formulations are as follows.

$$\sigma(z) = \frac{1}{1+e^{-z}}$$

$$z = \boldsymbol{w}^T\boldsymbol{x}$$; **w:** weight vector, **x**: feature vector.

$$P(\,y = 1 \mid \boldsymbol{x}) \; = \frac{1}{1 + e^{-(\boldsymbol{w}^T\boldsymbol{x})}}$$

$$P(\,y = 0 \mid \boldsymbol{x}) = 1 - \; P(\,y = 1 \mid \boldsymbol{x}) = \frac{e^{-(\boldsymbol{w}^T\boldsymbol{x})}}{1 + e^{-(\boldsymbol{w}^T\boldsymbol{x})}}$$

Likelihood function: $$\mathrm{L}(\mathbf{w}) \; = \prod_{y\_t's=1} \frac{1}{1+e^{-(\boldsymbol{w}^T\boldsymbol{x})}} \prod_{y\_t's=0} \frac{e^{-(\boldsymbol{w}^T\boldsymbol{x})}}{1+e^{-(\boldsymbol{w}^T\boldsymbol{x})}}$$

-Loglikelihood: $$-\mathrm{l}(\mathbf{w}) = \sum_{t=1}^{n} \left[\log\left(1 + e^{-(\boldsymbol{w}^T\boldsymbol{x}_i)}\right) - (1 - y_i)\boldsymbol{w}^T\boldsymbol{x_i}\right]$$

$$\frac{\partial(-l(\boldsymbol{w}))}{\partial \boldsymbol{w}} = \sum_{t=1}^{n} \left[\frac{x_i e^{-(\boldsymbol{w}^T\boldsymbol{x}_i)}}{1+e^{-(\boldsymbol{w}^T\boldsymbol{x}_i)}} - (1 - y_i)\boldsymbol{w}^T\boldsymbol{x_i}\right] = 0$$

However, as mentioned before, logistic regression works well for linearly seperable data. If our data is non-linear then this algorithm struggles to find good decision boundary. Since the real-life data like the dataset I have chosen are mostly nonlinear, different algorithm ,which can work well with nonlinear data, such as multi layer perceptron and decision trees can be chosen.

Multi Layer Perceptron (MLP)

A multi layer perceptron (MLP) is a type of neural network which is capable of learning complex relationship from data for both regression and classification tasks. Therefore, it fits the aim of this project. To be more clear about MLP, it a feedwork and fully connected neural network which has at least one hidden layer. In MLP, the neuron in each layer is connected to the all neurons in the next layer. Moreover, the layer is feeding the next layer after the computation of neurons. Its capability of handling the nonlinear data comes from the activation functions which are used in these computations. After these feedforward computations are done, the learning process is done thanks to backpropagation algorithm. The backpropagation algorithm is an iterative method which allows weightst to be adjused in a way that reduces the loss function. While the most common activation functions are sigmoid, ReLU and tanh, the most common optimization functions are gradient descent, stochacstic gradient descent and mini-batch gradient descent. The formula for gradient descent can be seen below.

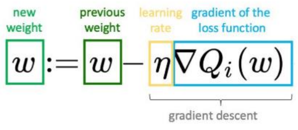


Figure 5: Gradient Descent Algorithm [5]

In Figure 6, the multi layer network with one hidden layer can be seen. Moreover, how the neurons do the calculations and feed the next neuron can be seen clearly.

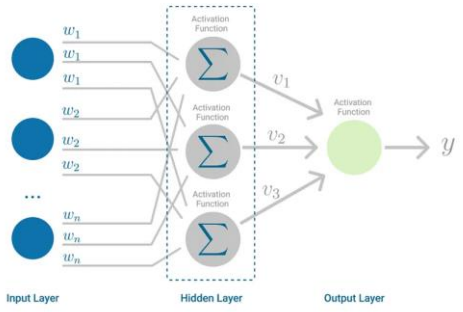


Figure 6: Multi Layer Perceptron with One Hidden Layer [6]

In Figure 7, the workflow of the neural network having ReLU as an activation function can be seen. As mentioned before, in the forward direction the loss is calculated and in the backward direction the weights updated in a way that reduces loss function iteratively.

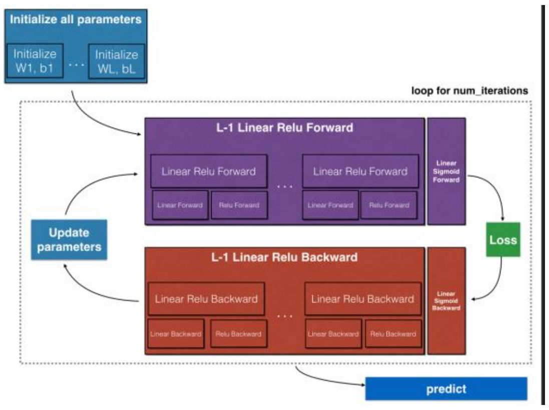


Figure 7: Workflow of Neural Network [7]

Besides these, one has to keep in mind that as the depth of the neural networks or the number of neurons increases, the neural network can learn more complex patterns. However, this may lead us to overfitting and expensive computation problems. Since the dataset used in this project is relatively small, this problems may occur during the project. Although the some regularization techniques like L1 regularization or L2 regularization prevents the overfitting, the designing and implementing a neural network are still an art.

Decision Tree

Decision Tree is a supervised learning algorithm, which recursively splits the data into subset based on specific criteria, for both regression and classification tasks. It works well with nonlinear data and can handle the both categorical and numerical data at the same time. It starts with root node and splits the data into branches regarding specific criteria and creates decision nodes. More specifically, the nodes represent the features in dataset. The edges represent the specific criteria and the leaf nodes represent the predicted class for classification tasks. Therefore, it is a powerful algorithm and it fits to goal of this project.

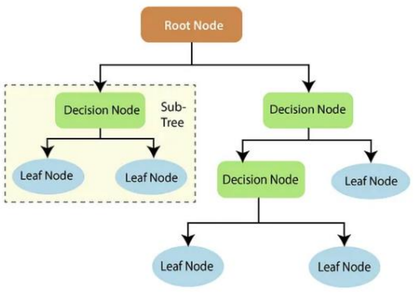


Figure 8: Architecture of Decision Tree [8]

Entropy and information gain help us to split the dataset based on the criteria. While the entropy computes the uncertainty in the dataset, information gain measures the reducton in entropy. Therefore, the best split can be chosen by maximizing the information gain. More specifically, if the root node represented as parent node and decision nodes which is a branched version of the previous node to the right and left can be presented as right and left child nodes respectively. Then the goal is to maximize the information gain by minimizing the entropy of the child nodes as the decision tree goes deeper. Mathematically, the entropy and information gain can be formulated as follows,

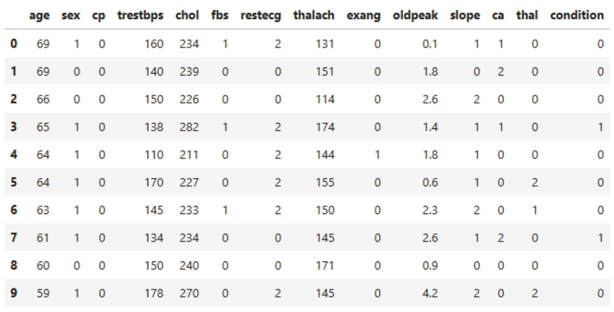
$$\text{Entropy} = -\sum p_i \log_2 p_i \quad ; p_i \text{ is the probability of } i^{th} \text{ class.}$$

$$\text{Information Gain} = \text{Entropy (parent)} - \left(\frac{\text{len(left)}}{\text{len(parent)}}\text{Entropy(left)} + \frac{\text{len(right)}}{\text{len(parent)}}\text{Entropy(right)}\right)$$

One has to keep mind that although the decision tree is a good algorithm to handle nonlinear data, it may suffer from the overfitting. Moreover, they make biased decision if the dataset is not balanced.

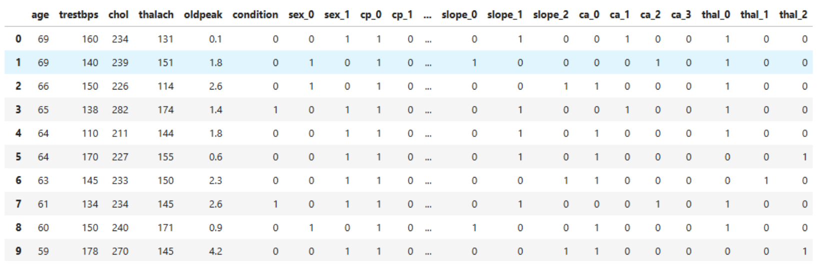
SIMULATION SETUP

I have started the project by understanding, visualizing and preprocessing the data. Thanks to libraries such as NumPy, Pandas, Matplotlib and Seaborn, I have created the plot which tells me the distribution of the both numerical and categorical features. Then I created correlation matrix to understand the relationship between features. After understood what I had, I thought that I need a standartization due to wide range of values of numerical data in different features and one hot encoding for the categorical values. After these steps, I have decided to use %85 of data as training and %15 of the data as testing for training the models without cross validation. For the cross validation cases, I have splitted my dataset %70, %15 and %15 of entire dataset as training validation and testing respectively. How I handle my dataset can be seen in below.



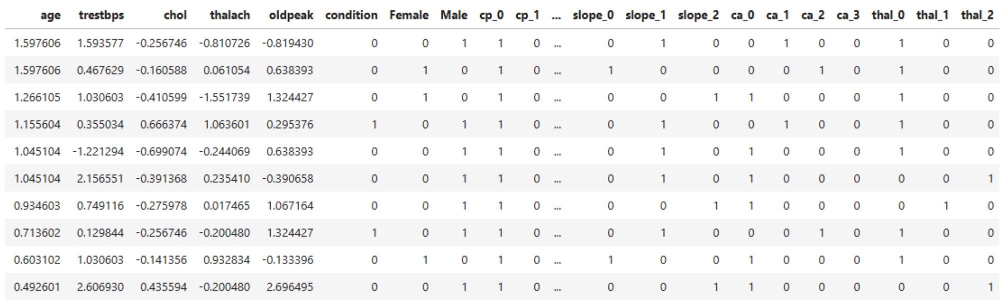
	age	sex	cp	trestbps	chol	fbs	restecg	thalach	exang	oldpeak	slope	ca	thal	condition
0	69	1	0	160	234	1	2	131	0	0.1	1	1	0	0
1	69	0	0	140	239	0	0	151	0	1.8	0	2	0	0
2	66	0	0	150	226	0	0	114	0	2.6	2	0	0	0
3	65	1	0	138	282	1	2	174	0	1.4	1	1	0	1
4	64	1	0	110	211	0	2	144	1	1.8	1	0	0	0
5	64	1	0	170	227	0	2	155	0	0.6	1	0	2	0
6	63	1	0	145	233	1	2	150	0	2.3	2	0	1	0
7	61	1	0	134	234	0	0	145	0	2.6	1	2	0	1
8	60	0	0	150	240	0	0	171	0	0.9	0	0	0	0
9	59	1	0	178	270	0	2	145	0	4.2	2	0	2	0

Figure 9: First 10 Data of Dataset



	age	trestbps	chol	thalach	oldpeak	condition	sex_0	sex_1	cp_0	cp_1	...	slope_0	slope_1	slope_2	ca_0	ca_1	ca_2	ca_3	thal_0	thal_1	thal_2
0	69	160	234	131	0.1	0	0	1	1	0	...	0	1	0	0	1	0	0	1	0	0
1	69	140	239	151	1.8	0	1	0	1	0	...	1	0	0	0	0	1	0	1	0	0
2	66	150	226	114	2.6	0	1	0	1	0	...	0	0	1	1	0	0	0	1	0	0
3	65	138	282	174	1.4	1	0	1	1	0	...	0	1	0	0	1	0	0	1	0	0
4	64	110	211	144	1.8	0	0	1	1	0	...	0	1	0	1	0	0	0	1	0	0
5	64	170	227	155	0.6	0	0	1	1	0	...	0	1	0	1	0	0	0	0	0	1
6	63	145	233	150	2.3	0	0	1	1	0	...	0	0	1	1	0	0	0	0	1	0
7	61	134	234	145	2.6	1	0	1	1	0	...	0	1	0	0	0	1	0	1	0	0
8	60	150	240	171	0.9	0	1	0	1	0	...	1	0	0	1	0	0	0	1	0	0
9	59	178	270	145	4.2	0	0	1	1	0	...	0	0	1	1	0	0	0	0	0	1

Figure 10: First 10 Data of Dataset After One Hot Encoding



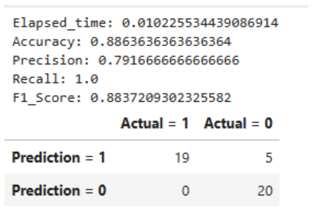
age	trestbps	chol	thalach	oldpeak	condition	Female	Male	cp_0	cp_1	...	slope_0	slope_1	slope_2	ca_0	ca_1	ca_2	ca_3	thal_0	thal_1	thal_2
1.597606	1.593577	-0.256746	-0.810726	-0.819430	0	0	1	1	0	...	0	1	0	0	1	0	0	1	0	0
1.597606	0.467629	-0.160588	0.061054	0.638393	0	1	0	1	0	...	1	0	0	0	0	1	0	1	0	0
1.266105	1.030603	-0.410599	-1.551739	1.324427	0	1	0	1	0	...	0	0	1	1	0	0	0	1	0	0
1.155604	0.355034	0.666374	1.063601	0.295376	1	0	1	1	0	...	0	1	0	0	1	0	0	1	0	0
1.045104	-1.221294	-0.699074	-0.244069	0.638393	0	0	1	1	0	...	0	1	0	1	0	0	0	1	0	0
1.045104	2.156551	-0.391368	0.235410	-0.390658	0	0	1	1	0	...	0	1	0	1	0	0	0	0	0	1
0.934603	0.749116	-0.275978	0.017465	1.067164	0	0	1	1	0	...	0	0	1	1	0	0	0	0	1	0
0.713602	0.129844	-0.256746	-0.200480	1.324427	1	0	1	1	0	...	0	1	0	0	0	1	0	1	0	0
0.603102	1.030603	-0.141356	0.932834	-0.133396	0	1	0	1	0	...	1	0	0	1	0	0	0	1	0	0
0.492601	2.606930	0.435594	-0.200480	2.696495	0	0	1	1	0	...	0	0	1	1	0	0	0	0	0	1

Figure 11: First 10 Data of Dataset After One Hot Encoding and Standartization

SIMULATION RESULTS

Logictic Regression

To train the logistic regression without k-fold cross validation, learning rate and epochs were chosen as 0.001 and 500 respectively and arbitrarily. The obtained results can be seen in Figure 12.

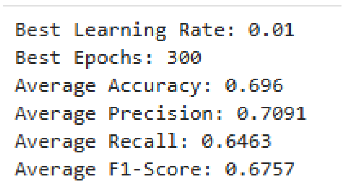


```
Elapsed_time: 0.010225534439086914
Accuracy: 0.8863636363636364
Precision: 0.7916666666666666
Recall: 1.0
F1_Score: 0.8837209302325582
```

	Actual = 1	Actual = 0
Prediction = 1	19	5
Prediction = 0	0	20

Figure 12: Results of Logistic Regression

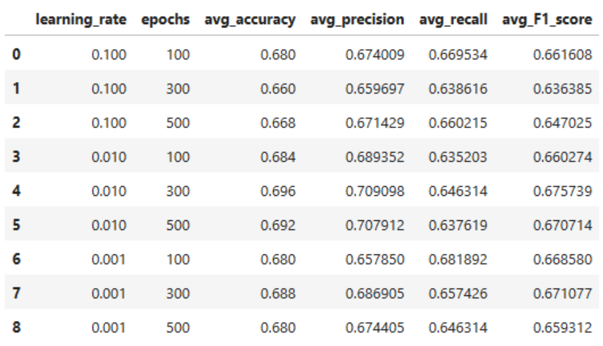
Then, logistic regression was trained by using k-fold cross validation. For cross validation two lists ,which includes 3 values in them, for learning rate and epochs were created. Thus, we had 9 combinations in total and with the help of the cross validation, the best combination was chosen regarding the maximum average accuracy.



```
Best Learning Rate: 0.01
Best Epochs: 300
Average Accuracy: 0.696
Average Precision: 0.7091
Average Recall: 0.6463
Average F1-Score: 0.6757
```

Figure 13: Results of K-Fold Cross Validation for Logistic Regression

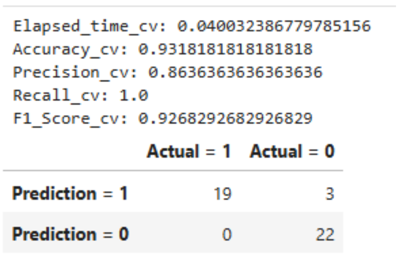
Results of the all combinations can be seen in Figure 14.



	learning_rate	epochs	avg_accuracy	avg_precision	avg_recall	avg_F1_score
0	0.100	100	0.680	0.674009	0.669534	0.661608
1	0.100	300	0.660	0.659697	0.638616	0.636385
2	0.100	500	0.668	0.671429	0.660215	0.647025
3	0.010	100	0.684	0.689352	0.635203	0.660274
4	0.010	300	0.696	0.709098	0.646314	0.675739
5	0.010	500	0.692	0.707912	0.637619	0.670714
6	0.001	100	0.680	0.657850	0.681892	0.668580
7	0.001	300	0.688	0.686905	0.657426	0.671077
8	0.001	500	0.680	0.674405	0.646314	0.659312

Figure 14: Results of All Combinations

After determined the optimal parameters which were 0.01 and 300 for learning rate and epochs respectively, the model was trained in test dataset using optimal parameters. The results can be seen in Figure 15



```
Elapsed_time_cv: 0.040032386779785156
Accuracy_cv: 0.9318181818181818
Precision_cv: 0.8636363636363636
Recall_cv: 1.0
F1_Score_cv: 0.9268292682926829
```

	Actual = 1	Actual = 0
Prediction = 1	19	3
Prediction = 0	0	22

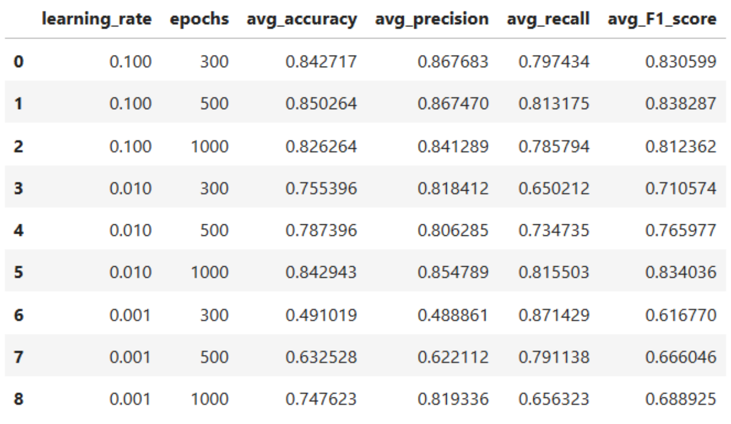
Figure 15: Results of Logistic Regression with Optimal Parameters

It can be seen that using optimal values determined by k-fold cross validation algorithm has increased the performance of the logistic regression model.

Multi Layer Perceptron (MLP)

The multi layer perceptron algorithm is tested using the k-fold cross validation. As it mentioned before, all dataset is splitted into %70 training, %15 validation and % 15 test data by taking my dataset size into consideration. In the k-fold cross validation, the learning rate, the number of epochs and the number of neurons in the hidden layer of the MLP are considered. To be more clear, the MLP algorithm is executed with all combinations where the learning rates are 0.1, 0.01 and 0.001, the number of epochs are 300, 500, 1000 and number of neurons in the hidden layer is 32, 64, 96.

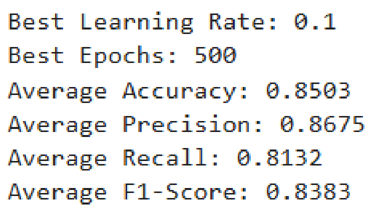
Let's first see the results obtained when the hidden layer is 32.



	learning_rate	epochs	avg_accuracy	avg_precision	avg_recall	avg_F1_score
0	0.100	300	0.842717	0.867683	0.797434	0.830599
1	0.100	500	0.850264	0.867470	0.813175	0.838287
2	0.100	1000	0.826264	0.841289	0.785794	0.812362
3	0.010	300	0.755396	0.818412	0.650212	0.710574
4	0.010	500	0.787396	0.806285	0.734735	0.765977
5	0.010	1000	0.842943	0.854789	0.815503	0.834036
6	0.001	300	0.491019	0.488861	0.871429	0.616770
7	0.001	500	0.632528	0.622112	0.791138	0.666046
8	0.001	1000	0.747623	0.819336	0.656323	0.688925

Figure 16: Results of All Combinations of MLP when number of neuron in hidden layer is 32

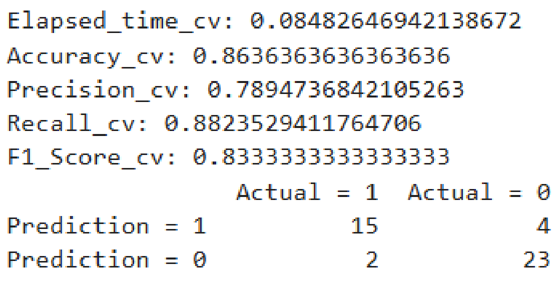
The optimal parameters for this case is chosen regarding the highest accuracy.



```
Best Learning Rate: 0.1
Best Epochs: 500
Average Accuracy: 0.8503
Average Precision: 0.8675
Average Recall: 0.8132
Average F1-Score: 0.8383
```

Figure 17: Results of K-Fold Cross Validation for MLP with hidden size 32

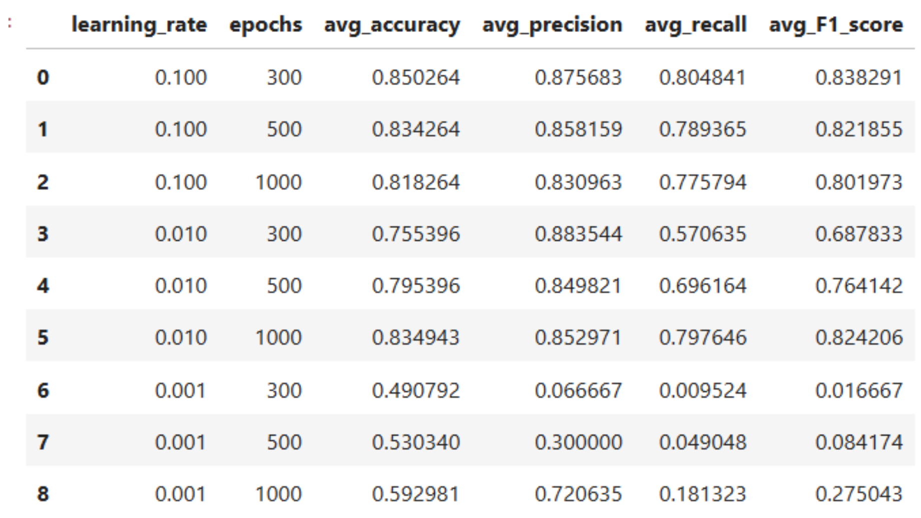
Then the optimal parameters are tested in the test data.



```
Elapsed_time_cv: 0.08482646942138672
Accuracy_cv: 0.8636363636363636
Precision_cv: 0.7894736842105263
Recall_cv: 0.8823529411764706
F1_Score_cv: 0.8333333333333333
                Actual = 1  Actual = 0
Prediction = 1          15           4
Prediction = 0           2          23
```

Figure 18: Final Results of MLP with Optimal Parameters when hidden size is 32

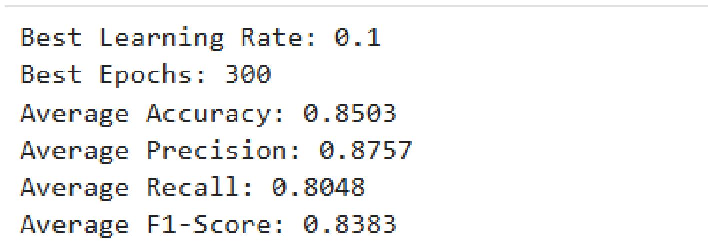
Now, we can see the results obtained when the hidden size is 64.



	learning_rate	epochs	avg_accuracy	avg_precision	avg_recall	avg_F1_score
0	0.100	300	0.850264	0.875683	0.804841	0.838291
1	0.100	500	0.834264	0.858159	0.789365	0.821855
2	0.100	1000	0.818264	0.830963	0.775794	0.801973
3	0.010	300	0.755396	0.883544	0.570635	0.687833
4	0.010	500	0.795396	0.849821	0.696164	0.764142
5	0.010	1000	0.834943	0.852971	0.797646	0.824206
6	0.001	300	0.490792	0.066667	0.009524	0.016667
7	0.001	500	0.530340	0.300000	0.049048	0.084174
8	0.001	1000	0.592981	0.720635	0.181323	0.275043

Figure 19: Results of All Combinations of MLP when number of neuron in hidden layer is 64

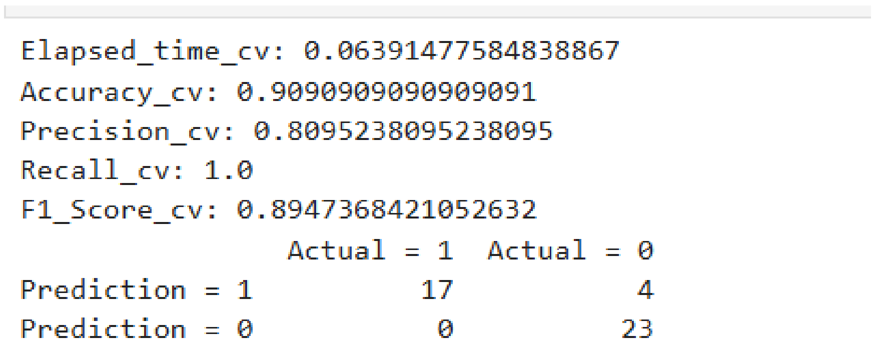
The optimal parameters is chosen regarding the highest average accuracy.



```
Best Learning Rate: 0.1
Best Epochs: 300
Average Accuracy: 0.8503
Average Precision: 0.8757
Average Recall: 0.8048
Average F1-Score: 0.8383
```

Figure 20: Results of K-Fold Cross Validation for MLP with hidden size 64

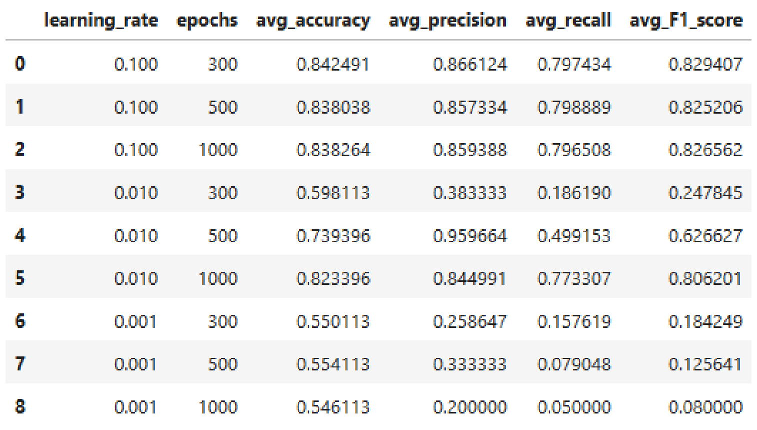
Then the optimal parameters are tested in the test data.



```
Elapsed_time_cv: 0.06391477584838867
Accuracy_cv: 0.9090909090909091
Precision_cv: 0.8095238095238095
Recall_cv: 1.0
F1_Score_cv: 0.8947368421052632
                 Actual = 1  Actual = 0
Prediction = 1           17           4
Prediction = 0            0          23
```

Figure 21: Final Results of MLP with Optimal Parameters when hidden size is 64.

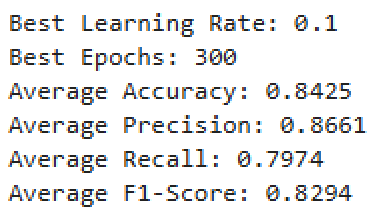
Finally, we can see the results obtained when the hidden size is 96.



	learning_rate	epochs	avg_accuracy	avg_precision	avg_recall	avg_F1_score
0	0.100	300	0.842491	0.866124	0.797434	0.829407
1	0.100	500	0.838038	0.857334	0.798889	0.825206
2	0.100	1000	0.838264	0.859388	0.796508	0.826562
3	0.010	300	0.598113	0.383333	0.186190	0.247845
4	0.010	500	0.739396	0.959664	0.499153	0.626627
5	0.010	1000	0.823396	0.844991	0.773307	0.806201
6	0.001	300	0.550113	0.258647	0.157619	0.184249
7	0.001	500	0.554113	0.333333	0.079048	0.125641
8	0.001	1000	0.546113	0.200000	0.050000	0.080000

Figure 22: Results of All Combinations of MLP when number of neuron in hidden layer is 96

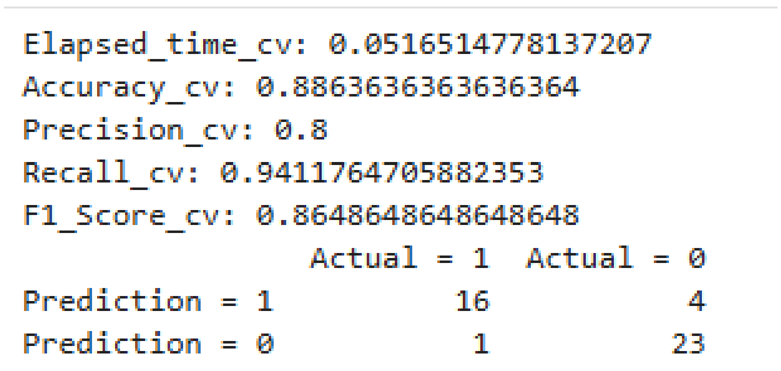
The optimal parameters is chosen regarding the highest average accuracy.



```
Best Learning Rate: 0.1
Best Epochs: 300
Average Accuracy: 0.8425
Average Precision: 0.8661
Average Recall: 0.7974
Average F1-Score: 0.8294
```

Figure 23: Results of K-Fold Cross Validation for MLP with hidden size 96

Then the optimal parameters are tested in the test data.



```
Elapsed_time_cv: 0.0516514778137207
Accuracy_cv: 0.8863636363636364
Precision_cv: 0.8
Recall_cv: 0.9411764705882353
F1_Score_cv: 0.8648648648648648
                Actual = 1  Actual = 0
Prediction = 1          16           4
Prediction = 0           1          23
```

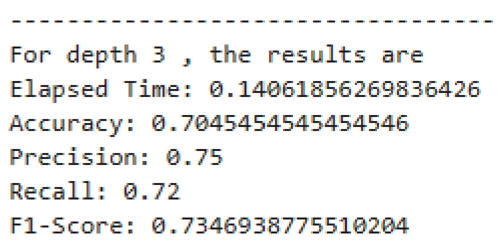
Figure 24: Final Results of MLP with Optimal Parameters when hidden size is 96

When we look at the results obtained in the MLP part, we can see that the highest accuracy is achieved when the number of neurons in the hidden layer is 64, the learning rate is 0.1 and number of epochs is 300. Moreover, the accuracy for this configuration is almost 91%. Furthermore, one of the important results which is crucial for diagnosis analysis is the case when actual situation says there is a disease but the model says there is no disease. It is important because the model says that a person is healthy although he/she is not. We do not want that because this results obtained my model may cause the death of patient. In the case where we got the highest accuracy, it is not the case and it is very good for us. As you can see in the Figure 21, the number when the actual situation is 1 and the prediction is 0, is 0 which means our model does not say that the patient is healty although it is not. Even though these results are satisfactory for our task, it is still open to further development. One have to keep in mind that ,in the MLP, there are lots of learnable parameters and hyperparameters which we define in advance. Thus, the highest accuracy may hide in the combinations we did not test before. Thus, machine learning is still an art.

Decision Trees

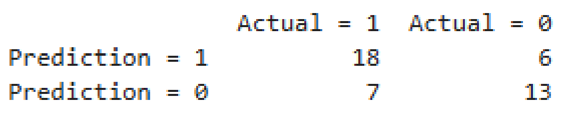
The decision tree algorithm does not the data to be standartized or encoded before. Therefore, in this algorithm, the original data in the dataset is used. Moreover, since our dataset includes only 297 data in it, I choose small depth sizes for the decision tree. Specifically, the depth size is chosen as 3, 4 and 5.

The obtained results when the depth is 3 like as follows,



```
----------------------------------
For depth 3 , the results are
Elapsed Time: 0.14061856269836426
Accuracy: 0.7045454545454546
Precision: 0.75
Recall: 0.72
F1-Score: 0.7346938775510204
```

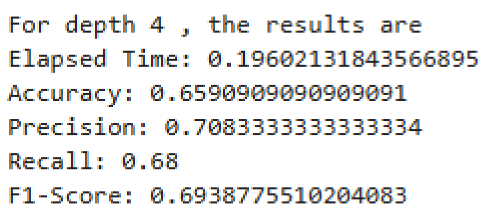
Figure 25: Results for Decision Tree with Depth = 3



```
                Actual = 1  Actual = 0
Prediction = 1          18           6
Prediction = 0           7          13
```

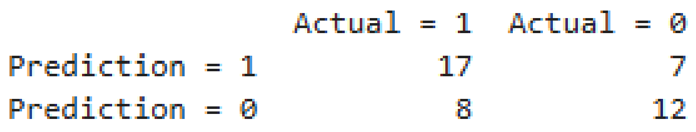
Figure 26: Confusion Matrix for Decision Tree with Depth = 3

The obtained results when the depth is 4 like as follows,



```
For depth 4 , the results are
Elapsed Time: 0.19602131843566895
Accuracy: 0.6590909090909091
Precision: 0.7083333333333334
Recall: 0.68
F1-Score: 0.6938775510204083
```

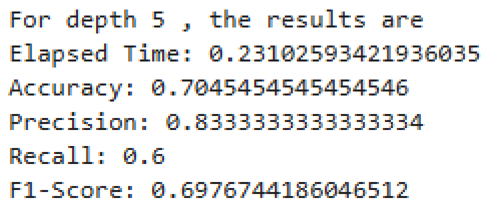
Figure 27: Results for Decision Tree with Depth = 4



```
                 Actual = 1  Actual = 0
Prediction = 1           17           7
Prediction = 0            8          12
```

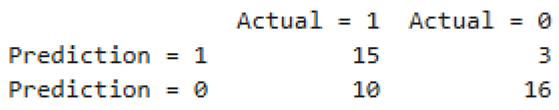
Figure 28: Confusion Matrix for Decision Tree with Depth = 4

The obtained results when the depth is 5 like as follows,



```
For depth 5 , the results are
Elapsed Time: 0.23102593421936035
Accuracy: 0.7045454545454546
Precision: 0.8333333333333334
Recall: 0.6
F1-Score: 0.6976744186046512
```

Figure 29: Results for Decision Tree with Depth = 5



```
                 Actual = 1  Actual = 0
Prediction = 1           15           3
Prediction = 0           10          16
```

Figure 30: Confusion Matrix for Decision Tree with Depth = 5

If we look at the results we obtained in this part, we can see not satisfactory results here unfortunately. Although the depth of the decision tree is small, the results are not good enough. The higher accuracy is obtained when the depth 3 which is 70.4%. Although the model with depth size 5 gets the similar accuracy its other results like recall or F1-score gives less score than the model with depth size 3. We can say that the model with depth size 3 gives balanced results compared to other decision tree algorithms. However, it is still noot good enough. It labels 8 case as healthy although they are not healthy which may causes the death of patients. Besides these, we can say that the model suffers from overfitting when we increase the depth size. This problem was mentioned when the machine learning algorithms was introduced before in this report. Overcoming this problem may require more advanced techniques like ensemble methods or gradient boosting algortihms like XGBoost which may be beyond the scope of this course.

COMPARISON OF THE RESULTS OBTAINED FROM DIFFERENT ALGORITHMS

When we look at the optimal results for each machine learning algorithm, we can observe that the Logistic Regression outperforms the other models in terms of its accuracy and elapsed time. In the logistic regression we get 93% accuracy with 0.04s elapsed time. In the MLP, we get 90.9% accuracy with 0.06s elapsed time. In the decision tree, we get 70.4% accuracy with 0.14s elapsed time.

Since our dataset is small, we might expect that the logistic regression may overperform the other models if the dataset is available for this results. Multi Layer Perceptrons have lots of learnable parameters so they require large datasets. Moreover, if the datasets are not large enough, they may suffer from the overfitting. In our case, although the MLP gives sufficient results for this task regarding the our dataset size, still it may be developed using more detailed parameter tuning. On the other hand, the decision tree algorithm gives less accuracy in the higher elapsed time. Thus, we can say that the decision tree is not suitable for this task. However, we increase the size of the dataset and use more advanced techniques like ensemble learning or XGBoost algorithm, the decision tree algoritm may be developed and give more desirable results.

CHALLENGES and SOLUTIONS

In this project, although the all algorithms like splitting the dataset, evaluation metrics, k-fold cross validation and other machine learning algoritmhs were implemented correctly, I have faced with some problems throughout the project. The most difficult thing in the project was the controlling and handling the shape of the variables. Since we use vectors and matrices, it was important to check the shapes of the variable to make calculations or operations. Thus, I have used .shape() command frequently. Other challenge was the implementing the decision tree algorithm. Since we did not cover this topic in our lectures, it requires lots of search and coding trials. Although it takes too much time compared to other methods, I implemented it succesfully as well. Throughout the project, I have found the solutions for the challenges mentioned above by using the documentations of the libraries and trial and fail method.

CONCLUSION

In this project, it was aimed to solve the diagnosis analysis problem in the healthcare sector. To do so, using the dataset we have, it was tried to develop machine learning algorithms which predict whether the person has a heart disease or not. More specifically, the logistic regression, the multi layer perceptron and the decision tree algorihms were implemented and compared with each other to understand which model works better. Before doing this, the optimal solutions for each algorithm were found. Since the extension was given for the Final Report, I have used the cross validation algorithms in a more detailed way and compare the all combinations for that machine learning algorithm. After that, I have compared the best solutions obtained for each machine learning algorithm. As a result, I concluded that the logistic regression overperforms the other machine learning algorithm with the given dataset and chosen parameters. However, one has to keep in mind that the algorithms may be developed using more advanced techniques and more detailed fine tuning methods. As a last comment, logistic regression and multi layer perceptron algorithms might be used for this heart disease prediction task and give satisfactory results for both patients and doctors.

REFERENCES

[1] World Health Organization, "Cardiovascular diseases," WHO, Oct. 16, 2024. [Online]. Available: https://www.who.int/europe/news-room/fact-sheets/item/cardiovascular-diseases. [Accessed: Oct. 16, 2024]

[2] Düzce Üniversitesi Hastanesi, "Türkiye'de ölümlerin yaklaşık %40'ı kalp ve damar hastalıklarından," Apr. 20, 2023. [Online]. Available: https://hastane.duzce.edu.tr/article/2023- 04-20/turkiyede-olumlerin-yaklasik-yuzde-40i-kalp-ve-damar-hastaliklarindan. [Accessed: Oct. 16, 2024].

[3] Centre Cardiologique Laval, "The benefits of early diagnosis for heart disease," Mar. 6, 2024. [Online]. Available: https://www.centrecardiolaval.com/en/2024/03/06/the-benefitsof039early-diagnosis-for-heart-disease. [Accessed: Oct. 16, 2024].

[4] "What is Logistic Regression?", *Spiceworks*, Oct. 2023. [Online]. Available: https://www.spiceworks.com/tech/artificial-intelligence/articles/what-is-logistic-regression/. [Accessed: 19-Nov-2024].

[5] "Gradient Descent Explanation," *Suboptimal.wiki*. [Online]. Available: https://suboptimal.wiki/explanation/gradient-descent/. [Accessed: 19-Nov-2024].

[6] A. Simran, "Multilayer Perceptron explained with a real-life example and Python code: Sentiment Analysis," *Towards Data Science*, Apr. 21, 2020. [Online]. Available: https://towardsdatascience.com/multilayer-perceptron-explained-with-a-real-life-example-and-python-code-sentiment-analysis-cb408ee93141. [Accessed: 19-Nov-2024].

[7]"EEE-443 Course Page," Bilkent University Moodle, Fall 2024. [Online]. Available: https://moodle.bilkent.edu.tr/2024-2025-fall/course/view.php?id=307. [Accessed: 19-Nov-2024].

[8] F. Philip, "Decision Tree," *Medium*, Jul. 21, 2020. [Online]. Available: https://medium.com/@favourphilic/decision-tree-5c1c7b6db59. [Accessed: 19-Nov-2024].

APPENDIX

```
import numpy as np

import pandas as pd

import seaborn as sns

import matplotlib.pyplot as plt

import time

data = pd.read_csv('heart_cleveland_upload.csv')

data_decision_tree = data

data.shape

data.head(10)

data.info()  # It can be seen that we don't have any missing values. All data is non-null. Moreover, we can see that 13 attributes have integer format. However, some them should be categorical as well.

data.describe().T

data.hist(figsize = (10,10))

plt.tight_layout()

plt.show()

sns.countplot(x = 'condition', data = data)  # Condition represent whether there is a heart disease ( condition = 1) or not (condition = 0)

data.condition.value_counts()         # So we have 160 heart disease case, 137 no heart disease case.

correlation_matrix = data.corr()

plt.figure(figsize=(12, 12))

sns.heatmap(correlation_matrix, annot = True)       # So far we looked at the attributes individually. Now let's look at how they are related with each other using correlation matrix.

plt.tight_layout()

# We have some categorical values. We have to use one-hot encodng to represent them.

# Our categorical features are sex, cp, fbs, restecg, exang, slope, ca, thal, condition

# Noncategorical features are age, trestbps, chol, thalach, old peak
```

```
categorical_features = ['sex', 'cp', 'fbs', 'restecg', 'exang', 'slope', 'ca', 'thal'] # 'condition' is already either 0 or 1 so no need for one-hot encoding.

encoded_data = pd.get_dummies(data, columns = categorical_features)

encoded_data.head(20)

# To increase the readibility of the table, we can change the name of columns

encoded_data.rename(columns = {'fbs_0': 'fbs ≤ 120'}, inplace = True)

encoded_data.rename(columns = {'fbs_1': 'fbs > 120'}, inplace = True)

encoded_data.rename(columns = {'sex_0': 'Female'}, inplace = True)

encoded_data.rename(columns = {'sex_1': 'Male'}, inplace = True)


encoded_data.head(20)

# We completed the one-hot encoding for categorical features. Now, it is time to handle the non-categorical features.

# Non-categorical values are continuous and they have different ranges. Therefore, we have to apply standartization.

# To do so, we will transform feature by subtracting the mean and dividing by standart deviation.

encoded_data.describe()

# Noncategorical features are age, trestbps, chol, thalach, old peak.

age_mean, age_std = encoded_data['age'].mean(), encoded_data['age'].std()

encoded_data['age'] = (encoded_data['age'] - age_mean) / age_std


trestbps_mean, trestbps_std = encoded_data['trestbps'].mean(), encoded_data['trestbps'].std()

encoded_data['trestbps'] = (encoded_data['trestbps'] - trestbps_mean) / trestbps_std


chol_mean, chol_std = encoded_data['chol'].mean(), encoded_data['chol'].std()

encoded_data['chol'] = (encoded_data['chol'] - chol_mean) / chol_std


thalach_mean, thalach_std = encoded_data['thalach'].mean(), encoded_data['thalach'].std()

encoded_data['thalach'] = (encoded_data['thalach'] - thalach_mean) / thalach_std


oldpeak_mean, oldpeak_std = encoded_data['oldpeak'].mean(), encoded_data['oldpeak'].std()

encoded_data['oldpeak'] = (encoded_data['oldpeak'] - oldpeak_mean) / oldpeak_std

encoded_data.head(10)    # Let's see our standartized data.
```

```
def evaluation_metrics(y_pred, y_test):           # Accuracy, confusion_matrix, precision, recall, F1
Score
    tp, fp, fn, tn = 0, 0, 0, 0
    tp = np.sum((y_pred == 1) & (y_test == 1))     # True Positive
    fp = np.sum((y_pred == 1) & (y_test == 0))     # False Positive
    fn = np.sum((y_pred == 0) & (y_test == 1))     # False Negative
    tn = np.sum((y_pred == 0) & (y_test == 0))     # True Negative


    if (tp + fp + fn + tn) != 0:
        accuracy = (tp + tn) / (tp + fp + fn + tn)
    else:
        accuracy = 0

    if (tp + fp) != 0:
        precision = (tp) / (tp + fp)
    else:
        precision = 0

    if (tp + fn) != 0:
        recall = (tp) / (tp + fn)
    else:
        recall = 0

    if precision + recall != 0:
        F1_score = 2 * (precision * recall) / (precision + recall)
    else:
        F1_score = 0



    confusion_matrix_columns = ['Actual = 1', 'Actual = 0']
    confusion_matrix_rows = ['Prediction = 1', 'Prediction = 0']
    values = [[tp, fp], [fn, tn]]
    confusion_matrix = pd.DataFrame(values, index = confusion_matrix_rows, columns = 
confusion_matrix_columns)


    return accuracy, precision, recall, F1_score, confusion_matrix
# So far we have completed data understanding, data preprocessing, standartization and evaluation 
metrics.
# However, before applying any machine algorithm to our dataset, we first split our dataset into 
training, validation and testing parts.


data.info()     # We have 297 data in total.
X = data.drop('condition', axis = 1)
y = data['condition']
X.values.shape, y.values.shape
type(X.values), type(y.values)
data.index
def splitting_dataset(data, y, validation_size, test_size):
    num_total_samples = data.shape[0]
    shuffled_samples = np.random.permutation(data.index)          # First shuffle the samples


    validation_size = int(num_total_samples * validation_size)
    test_size = int(num_total_samples * test_size)
    training_size = num_total_samples - (validation_size + test_size)


    training_samples = shuffled_samples[: training_size]                        # Approximately %70 of total 
samples
    validation_samples = shuffled_samples[training_size: training_size + validation_size]          # 
Approximately %15 of total samples
    test_samples = shuffled_samples[training_size + validation_size: ]                        # 
Approxiamtely %15 of total samples


    # Now, we have to represent our splitted data in different DataFrames.
    training_df = data.iloc[training_samples]
    validation_df = data.iloc[validation_samples]
    test_df = data.iloc[test_samples]
    # Now, seperate the data into features and target.
    X_training = training_df.drop([y], axis = 1)                # Drop the target from column
    y_training = training_df[y]                                 # Save only target

    X_validation = validation_df.drop([y], axis = 1)
    y_validation = validation_df[y]

    X_test = test_df.drop([y], axis = 1)
    y_test = test_df[y]


    return X_training, y_training, X_validation, y_validation, X_test, y_test
# Now, let's try our splitting_dataset function. Remember we have 297 data in total.
X_training, y_training, X_validation, y_validation, X_test, y_test = splitting_dataset(data =
encoded_data, y = 'condition', validation_size = 0.15, test_size = 0.15)
print('Shape of X_training:', X_training.shape)
print('Shape of y_training:', y_training.shape)
print('Shape of X_validation:', X_validation.shape)
print('Shape of y_validation:', y_validation.shape)
print('Shape of X_test:', X_test.shape)
print('Shape of y_test:', y_test.shape)
encoded_data.head(10)

class Logistic_Regression:
    def __init__(self, learning_rate, epochs):
        self.learning_rate = learning_rate
        self.epoch = epochs
        self.w = None

    def sigmoid(self, z):
        sigmoid = 1 / (1 + np.exp(-z))   # score/probability of the positive class
        return sigmoid

    def training_logistic(self, X, y):
        num_of_samples, num_of_features = X.shape
        X_biased = np.hstack((np.ones((num_of_samples, 1)), X))
        self.w = np.zeros(X_biased.shape[1]) # We have a weight for each feature. w is column vector.
        for epoch in range(self.epoch):
            z = np.dot(X_biased, self.w)
            y_score = self.sigmoid(z)    # Related scores/probabilities.
            error = y - y_score
            gradient = np.dot(X_biased.T, error)
            self.w = self.w + self.learning_rate * gradient

    def score_prediction(self, X):
        X_biased = np.hstack((np.ones((X.shape[0], 1)), X))
        z = np.dot(X_biased, self.w)
        score_prediction = self.sigmoid(z)
        return score_prediction

    def prediction(self, X, threshold = 0.5):
        X_biased = np.hstack((np.ones((X.shape[0], 1)), X))
        y_scores = self.score_prediction(X)
        predictions = np.zeros(X.shape[0])              # Number of predictions = number of samples

        for index, y_score in enumerate(y_scores):              # We need corresponding index to update
prediction's terms either as 0 or 1.
            if y_score >= threshold:
                predictions[index] = 1
            else:
                predictions[index] = 0
        return predictions




    # I have splitted this into three parts. However, to see the result without cross validation, I
concatenate the
# training and validation sets again. As a results, I have splitted the entire dataset
# as %85 training and %15 testing.
```

```
X_training = np.concatenate((X_training, X_validation), axis=0)
y_training = np.concatenate((y_training, y_validation), axis=0)


start_time_model_1 = time.time()
model_1 = Logistic_Regression(learning_rate = 0.001, epochs = 500)
model_1.training_logistic(X_training, y_training)
end_time_model_1 = time.time()
elapsed_time = end_time_model_1 - start_time_model_1
print('Elapsed_time:', elapsed_time)
model_1_pred = model_1.prediction(X_test)
accuracy, precision, recall, F1_score, confusion_matrix = evaluation_metrics(y_pred = model_1_pred,
y_test = y_test.values)


print('Accuracy:', accuracy)
print('Precision:', precision)
print('Recall:', recall)
print('F1_Score:', F1_score)
confusion_matrix


def kfold_cross_validation(data, k, learning_rates, epochs, test_size):
  number_of_samples = data.shape [0]
  highest_accuracy = 0
  optimal_accuracies = []
  optimal_precisions = []
  optimal_recalls = []
  optimal_F1_scores = []
  hyperparameter_results = []
  optimal_parameters = {'learning_rate': None, 'epochs': None}
  shuffled_samples = np.random.permutation(data.index)
  test_size = int(number_of_samples * test_size)
  training_and_validation_size = number_of_samples - test_size          # Seperate the test data to
evaluate the performance of the model later.



  test_samples = shuffled_samples[: test_size]
```

```
    training_and_validation_samples = shuffled_samples[test_size: ]
    test_samples_df = data.iloc[test_samples]
    training_and_validation_samples_df = data.iloc[training_and_validation_samples]
    X_test_cv = test_samples_df.drop('condition', axis = 1)
    y_test_cv = test_samples_df['condition']
    X_without_test_cv = training_and_validation_samples_df.drop('condition', axis = 1)
    y_without_test_cv = training_and_validation_samples_df['condition']

    fold_size = training_and_validation_size // k     # To make the size integer.

    for learning_rate in learning_rates:
        for epoch in epochs:
            accuracies, precisions, recalls, F1_scores = [], [], [], []
            for fold in range(k):
                fold_starting = fold_size * fold
                if fold < k - 1:
                    fold_ending = fold_starting + fold_size
                elif fold == k:
                    fold_ending = len(data)

                validation_fold = training_and_validation_samples_df.iloc[fold_starting: fold_ending]
                training_folds = pd.concat([training_and_validation_samples_df.iloc[: fold_starting],
training_and_validation_samples_df.iloc[fold_ending: ]])        # Concatenate intervals outside the
validation_fold
                X_training = training_folds.drop('condition', axis = 1)
                y_training = training_folds['condition']
                X_validation = validation_fold.drop('condition', axis = 1)
                y_validation = validation_fold['condition']

                model_1_with_cv = Logistic_Regression(learning_rate = learning_rate, epochs = epoch)
                model_1_with_cv.training_logistic(X_training, y_training)
                model_1_with_cv_pred = model_1_with_cv.prediction(X_validation)

                accuracy, precision, recall, F1_score, confusion_matrix = evaluation_metrics(y_pred =
model_1_with_cv_pred, y_test = y_validation)
```

```
            accuracies.append(accuracy)
            precisions.append(precision)
            recalls.append(recall)
            F1_scores.append(F1_score)

        average_accuracy = np.mean(accuracies)
        average_precision = np.mean(precisions)
        average_recall = np.mean(recalls)
        average_F1_score = np.mean(F1_scores)

        hyperparameter_results.append({
            'learning_rate': learning_rate,
            'epochs': epoch,
            'avg_accuracy': average_accuracy,
            'avg_precision': average_precision,
            'avg_recall': average_recall,
            'avg_F1_score': average_F1_score
        })

        if average_accuracy > highest_accuracy:
            highest_accuracy = average_accuracy
            optimal_parameters = {'learning_rate': learning_rate, 'epochs': epoch}
            optimal_accuracies = accuracies
            optimal_precisions = precisions
            optimal_recalls = recalls
            optimal_F1_scores = F1_scores


results_df = pd.DataFrame(hyperparameter_results)

print(f"Best Learning Rate: {optimal_parameters['learning_rate']}")
print(f"Best Epochs: {optimal_parameters['epochs']}")
print("Average Accuracy:", round(np.mean(optimal_accuracies), 4))
print("Average Precision:", round(np.mean(optimal_precisions), 4))
```

```
    print("Average Recall:", round(np.mean(optimal_recalls), 4))

    print("Average F1-Score:", round(np.mean(optimal_F1_scores), 4))

    return optimal_parameters, highest_accuracy, results_df, X_test, y_test, X_without_test_cv,
y_without_test_cv

# Example usage

optimal_parameters, best_accuracy, results_cv_df, X_test_cv, y_test_cv, X_without_test_cv,
y_without_test_cv  = kfold_cross_validation(data = encoded_data, k=5, learning_rates=[0.1, 0.01,
0.001], epochs=[100, 300, 500], test_size = 0.15)

results_cv_df

optimal_learning_rate = optimal_parameters['learning_rate']

optimal_epochs = optimal_parameters['epochs']

start_time_model_1_cv = time.time()

model_1_cv = Logistic_Regression(learning_rate = optimal_learning_rate, epochs = optimal_epochs)

model_1_cv.training_logistic(X_without_test_cv, y_without_test_cv)

end_time_model_1_cv = time.time()

elapsed_time_cv = end_time_model_1_cv - start_time_model_1_cv

print('Elapsed_time_cv:', elapsed_time_cv)

model_1_cv_pred = model_1_cv.prediction(X_test_cv)

accuracy, precision, recall, F1_score, confusion_matrix = evaluation_metrics(y_pred =
model_1_cv_pred, y_test = y_test_cv)


print('Accuracy_cv:', accuracy)

print('Precision_cv:', precision)

print('Recall_cv:', recall)

print('F1_Score_cv:', F1_score)

confusion_matrix

X_training.shape

# Let's prepare our data set for multi layer perceptron again.

X_training, y_training, X_validation, y_validation, X_test, y_test = splitting_dataset(data =
encoded_data, y = 'condition', validation_size = 0.15, test_size = 0.15)

print('Shape of X_training:', X_training.shape)

print('Shape of y_training:', y_training.shape)

print('Shape of X_validation:', X_validation.shape)

print('Shape of y_validation:', y_validation.shape)

print('Shape of X_test:', X_test.shape)

print('Shape of y_test:', y_test.shape)
```

```
class Multi_Layer_Perceptron:
    def __init__(self, input_size, hidden_size, lr, epochs):
        np.random.seed(42)
        # After some trial, I realize that multiply with 0.01 provides better convergence.
        self.W1 = np.random.randn(input_size, hidden_size) * 0.01
        # Same for W2.
        self.W2 = np.random.randn(hidden_size, 1) * 0.01
        self.b1 = np.zeros((1, hidden_size))
        self.b2 = np.zeros((1, 1))
        self.lr = lr
        self.epochs = epochs

    def relu(self, z):
        return np.maximum(0, z)

    def relu_derivative(self, z):
        return (z > 0).astype(float)

    def sigmoid(self, z):
        return 1 / (1 + np.exp(-z))

    def sigmoid_derivative(self, z):
        return self.sigmoid(z) * (1 - self.sigmoid(z))

    def forward_pass(self, X):
        self.Z1 = np.dot(X, self.W1) + self.b1          # Induced field for hidden layer
        self.A1 = self.relu(self.Z1)                    # Activation for hidden layer.
        self.Z2 = np.dot(self.A1, self.W2) + self.b2    # Induced field for output
        self.A2 = self.sigmoid(self.Z2)                 # Actıvation for output. Since it is a binary
classification. Sigmoid is preferred.
        return self.A2

    def backpropagation(self, X, y, y_pred):
        N = X.shape[0]
        # Gradients for output layer. I have applied some reshaping procedures.
        dZ2 = y_pred - y.reshape(-1, 1)
        dW2 = (1 / N) * np.dot(self.A1.T, dZ2)
        db2 = (1 / N) * np.sum(dZ2, axis=0, keepdims=True)

        # Gradient for hidden layer. Same procedures hold.
        dA1 = np.dot(dZ2, self.W2.T)
        dZ1 = dA1 * self.relu_derivative(self.Z1)  # Use Z1 here
        dW1 = (1 / N) * np.dot(X.T, dZ1)
        db1 = (1 / N) * np.sum(dZ1, axis=0, keepdims=True)

        # Now, we can update the our learnable parameters.
        self.W1 = self.W1 - (self.lr * dW1)
        self.W2 = self.W2 - (self.lr * dW2)
        self.b1 = self.b1 - (self.lr * db1)
        self.b2 = self.b2 - (self.lr * db2)

    def training_MLP(self, X_train, y_train):
        stabilizer = 1e-8    # I have to define this for better convergence. Otherwise, since I use logloss, it
suffers from some values.
        for epoch in range(self.epochs):
            y_pred = self.forward_pass(X_train)
            self.backpropagation(X_train, y_train, y_pred)


            if epoch % 100 == 0:
                loss = -np.mean(y_train * np.log(y_pred + stabilizer) + (1 - y_train) * np.log(1 - y_pred +
stabilizer))

    def predict(self, X, threshold):
        y_pred = self.forward_pass(X)
        return (y_pred >= threshold).astype(int)
# Lets check shapes of data before applying MLP.
print('Shape of X_training:', X_training.shape)
print('Shape of y_training:', y_training.shape)
print('Shape of X_validation:', X_validation.shape)
print('Shape of y_validation:', y_validation.shape)
print('Shape of X_test:', X_test.shape)
print('Shape of y_test:', y_test.shape)


# I have already coded the k-fold cross validation. However, it needs to be arranged for MLP
specifically.


def kfold_cross_validation(data, k, input_size, hidden_size, learning_rates, epochs, test_size):
    number_of_samples = data.shape[0]
    highest_accuracy = 0
    optimal_accuracies, optimal_precisions, optimal_recalls, optimal_F1_scores = [], [], [], []
    hyperparameter_results = []
    optimal_parameters = {'learning_rate': None, 'epochs': None}
    shuffled_samples = np.random.permutation(data.index)
    test_size = int(number_of_samples * test_size)
    training_and_validation_size = number_of_samples - test_size

    test_samples = shuffled_samples[:test_size]
    training_and_validation_samples = shuffled_samples[test_size:]
    test_samples_df = data.iloc[test_samples]
    training_and_validation_samples_df = data.iloc[training_and_validation_samples]
    X_test_cv = test_samples_df.drop('condition', axis=1)
    y_test_cv = test_samples_df['condition']
    X_without_test_cv = training_and_validation_samples_df.drop('condition', axis=1)
    y_without_test_cv = training_and_validation_samples_df['condition']

    fold_size = training_and_validation_size // k

    for learning_rate in learning_rates:
        for epoch in epochs:
            accuracies, precisions, recalls, F1_scores = [], [], [], []
            for fold in range(k):
                fold_starting = fold_size * fold
                fold_ending = fold_starting + fold_size if fold < k - 1 else
len(training_and_validation_samples_df)
```

```
        validation_fold = training_and_validation_samples_df.iloc[fold_starting:fold_ending]
        training_folds = pd.concat([
            training_and_validation_samples_df.iloc[:fold_starting],
            training_and_validation_samples_df.iloc[fold_ending:]
        ])

        X_training = training_folds.drop('condition', axis=1)
        y_training = training_folds['condition']
        X_validation = validation_fold.drop('condition', axis=1)
        y_validation = validation_fold['condition']

        # Convert to numpy arrays and reshape y
        X_training = X_training.values
        y_training = y_training.values.reshape(-1, 1)
        X_validation = X_validation.values
        y_validation = y_validation.values.reshape(-1, 1)

        # Initialize and train the model
        model_2_with_cv = Multi_Layer_Perceptron(input_size=input_size,
hidden_size=hidden_size,
                                     lr=learning_rate, epochs=epoch)
        model_2_with_cv.training_MLP(X_training, y_training)
        model_2_with_cv_pred = model_2_with_cv.predict(X_validation, threshold=0.5)

        # Evaluate metrics
        accuracy, precision, recall, F1_score, confusion_matrix = evaluation_metrics(
            y_pred=model_2_with_cv_pred.ravel(), y_test=y_validation.ravel())
        accuracies.append(accuracy)
        precisions.append(precision)
        recalls.append(recall)
        F1_scores.append(F1_score)

    average_accuracy = np.mean(accuracies)
```

```
        average_precision = np.mean(precisions)
        average_recall = np.mean(recalls)
        average_F1_score = np.mean(F1_scores)

        hyperparameter_results.append({
            'learning_rate': learning_rate,
            'epochs': epoch,
            'avg_accuracy': average_accuracy,
            'avg_precision': average_precision,
            'avg_recall': average_recall,
            'avg_F1_score': average_F1_score
        })

        if average_accuracy > highest_accuracy:
            highest_accuracy = average_accuracy
            optimal_parameters = {'learning_rate': learning_rate, 'epochs': epoch}
            optimal_accuracies = accuracies
            optimal_precisions = precisions
            optimal_recalls = recalls
            optimal_F1_scores = F1_scores

    results_df = pd.DataFrame(hyperparameter_results)

    print(f"Best Learning Rate: {optimal_parameters['learning_rate']}")
    print(f"Best Epochs: {optimal_parameters['epochs']}")
    print("Average Accuracy:", round(np.mean(optimal_accuracies), 4))
    print("Average Precision:", round(np.mean(optimal_precisions), 4))
    print("Average Recall:", round(np.mean(optimal_recalls), 4))
    print("Average F1-Score:", round(np.mean(optimal_F1_scores), 4))
    return optimal_parameters, highest_accuracy, results_df, X_test_cv, y_test_cv, X_without_test_cv, 
y_without_test_cv
optimal_parameters, best_accuracy, results_cv_df, X_test_cv, y_test_cv, X_without_test_cv, 
y_without_test_cv  = kfold_cross_validation(data = encoded_data, k=5, input_size = 28, hidden_size 
= 96, learning_rates=[0.1, 0.01, 0.001], epochs=[300, 500, 1000], test_size = 0.15)
# Lets see results of all combinations.
```

```
results_cv_df

optimal_learning_rate = optimal_parameters['learning_rate']

optimal_epochs = optimal_parameters['epochs']

print(optimal_learning_rate)

print(optimal_epochs)

y_without_test_cv = y_without_test_cv.to_numpy().reshape(-1, 1)

y_test_cv = y_test_cv.to_numpy().reshape(-1, 1)


start_time_model_2_cv = time.time()

model_2_cv = Multi_Layer_Perceptron(input_size=28, hidden_size=32, lr=optimal_learning_rate, epochs=optimal_epochs)

model_2_cv.training_MLP(X_without_test_cv.values, y_without_test_cv)

end_time_model_2_cv = time.time()

elapsed_time_cv = end_time_model_2_cv - start_time_model_2_cv

print('Elapsed_time_cv:', elapsed_time_cv)



model_2_cv_pred = model_2_cv.predict(X_test_cv.values, threshold=0.5)

accuracy, precision, recall, F1_score, confusion_matrix = evaluation_metrics(y_pred=model_2_cv_pred.ravel(), y_test=y_test_cv.ravel())


print('Accuracy_cv:', accuracy)

print('Precision_cv:', precision)

print('Recall_cv:', recall)

print('F1_Score_cv:', F1_score)

print(confusion_matrix)

# Now, we can move on to our last machine learning algorithm which is Decision Tree.

# Decision Tree algorithm is different from previous models. It doesnt need to use encoded or standartized data to process.

# Thus, lets use our initial raw data.


X_training, y_training, X_validation, y_validation, X_test, y_test = splitting_dataset(

    data=data_decision_tree, y='condition', validation_size=0.15, test_size=0.15

)

# I have converted my data frames into numpy arrays.
```

```
X_training = X_training.to_numpy()
y_training = y_training.to_numpy()
X_validation = X_validation.to_numpy()
y_validation = y_validation.to_numpy()
X_test = X_test.to_numpy()
y_test = y_test.to_numpy()
X_training = np.concatenate((X_training, X_validation), axis=0)
y_training = np.concatenate((y_training, y_validation), axis=0)
#X_training = np.concatenate((X_training, X_validation), axis=0)
#y_training = np.concatenate((y_training, y_validation), axis=0)
#print(X_training.shape)
print(X_training.shape)
print(y_training.shape)
print(X_test.shape)
print(y_test.shape)
# Lets first define our single node.
class Node:
    def __init__(self, feature = None, threshold = None, left_child = None, right_child = None, label = None):
        self.feature = feature
        self.threshold = threshold
        self.right_child = right_child
        self.left_child = left_child
        self.label = label


# Now, lets build our tree.
class DecisionTree:
    def __init__(self, max_depth=None):
        self.root = None
        self.max_depth = max_depth


    def entropy(self, y):
        unique_values, counts = np.unique(y, return_counts = True)
        prob = counts / counts.sum()
        valid_probs = prob[prob > 0]     # To eliminate the problematic values for log.
        entropy = -np.sum(valid_probs * np.log2(valid_probs))         # This formula uses log2 specifically
as I mentioned in report.
        return entropy


    def data_splitting(self, X, y, feature, threshold):
        right_index = X[:, feature] > threshold
        right_y = y[right_index]

        left_index = X[:, feature] <= threshold
        left_y = y[left_index]

        return left_y, right_y



    def information_gain(self, y, left_child, right_child):
        right_entropy = self.entropy(right_child)
        left_entropy = self.entropy(left_child)
        parent_entropy = self.entropy(y)

        w_left = len(left_child) / len(y)
        w_right = len(right_child) / len(y)
        IG =  parent_entropy - ((w_right * right_entropy) + (w_left * left_entropy) )
        return IG

    def best_split(self, X, y):
        best_gain = -np.inf
        best_feature = None
        best_threshold = None
        all_features = X.shape[1]

        for feature in range(all_features):

            thresholds = np.unique(X[:, feature])
            for threshold in thresholds:
                left_child, right_child = self.data_splitting(X, y, feature, threshold)
                if len(left_child) == 0 or len(right_child) == 0:
                    continue

                gain = self.information_gain(y, left_child, right_child)
                if gain > best_gain:
                    best_gain = gain
                    best_feature = feature
                    best_threshold = threshold



        return best_feature, best_threshold

    def build_tree(self, X, y, depth = 0):

        # We have 3 reasons to stop expanding the tree further.
        if (depth == self.max_depth) or (len(np.unique(y)) == 1) or len(y) <= 1:
            label = np.bincount(y).argmax()

            return Node(label = label)

        feature, threshold = self.best_split(X, y)

        if feature is None or feature >= X.shape[1]:   # I have tried to assign majority class, if feature
does not find its location.

            return Node(label = np.bincount(y).argmax())

        right_index = X[:, feature] > threshold
        right_tree = self.build_tree(X[right_index], y[right_index], depth + 1)

        left_index = X[:, feature] <= threshold
        left_tree = self.build_tree(X[left_index], y[left_index], depth + 1)
        return Node(feature = feature, threshold = threshold, left_child = left_tree, right_child =
right_tree)




    def model_fitting(self, X, y):
        self.root = self.build_tree(X, y)


    def single_prediction(self, x, node):
        if node.label is not None:
            return node.label

        if node.feature is None or node.feature >= len(x):  # Check for valid feature

            return 0

        if x[node.feature] <= node.threshold:
            return self.single_prediction(x, node.left_child)

        else:
            return self.single_prediction(x, node.right_child)



    def multiple_prediction(self, X):

        return np.array([self.single_prediction(x, self.root) for x in X])
# Remember our data size is relatively small for other dataset. Specifically, we have 297 data in our
dataset.
# Thus, the numbers for maximum depth is chosen regarding these.
max_depth = [3, 4, 5]

for depth in max_depth:
```

```
start_time_model_3 = time.time()
model_3 = DecisionTree(max_depth = depth)
model_3.model_fitting(X_training, y_training)
end_time_model_3 = time.time()
elapsed_time = end_time_model_3 - start_time_model_3

model_3_pred = model_3.multiple_prediction(X_test)

accuracy, precision, recall, F1_score, confusion_matrix = evaluation_metrics(
    y_pred = model_3_pred, y_test = y_test)



print('----------------------------------')
print('For depth', depth, ', the results are')
print("Elapsed Time:", elapsed_time)
print("Accuracy:", accuracy)
print("Precision:", precision)
print("Recall:", recall)
print("F1-Score:", F1_score)
print()
print(confusion_matrix)
```