Ömer Tuğrul – 22002723
Selin Ataş- 21902858

EEE 485 – Breast Cancer Classification Project – Final Report

Contents

Ömer Tuğrul – 22002723
Selin Ataş- 21902858

Introduction

Breast cancer, one of the most prevalent cancers globally, affects millions of people each year. This condition arises from the uncontrolled growth of abnormal cells in the breast, forming tumors. Without an appropriate treatment, it can spread throughout the body and become fatal. Consequently, early and precise detection is vital for effective treatment and enhancing patient survival chances.

In 2020, about 2.3 million women were diagnosed with breast cancer, leading to 685,000 deaths worldwide. By the end of that year, approximately 7.8 million women who had been diagnosed with breast cancer in past 5 years were alive. Even though breast cancer is not exclusive to women, 0.5-1% of cases occur in men, women are the primary risk group in this disease.

The symptoms of breast cancer can vary, especially in more advanced stages. Common symptoms include a breast lump or thickening (often painless), changes in the breast's size, shape, or appearance, skin alterations like dimpling, redness, or pitting, changes in the nipple or surrounding area (areola), and abnormal or bloody nipple discharge.

Given that breast cancer can observed with different symptoms in individuals and may not show any symptoms in its early stages, distinguishing between benign (non-cancerous) and malignant (cancerous) breast cells is challenging.

This project aims to develop a machine learning model capable of classifying breast cancer as benign or malignant. It utilizes a dataset includes 569 patient cases and 30 features related to breast cancer which are used to analyze and build predictive models.

Dataset Description

The dataset for this project, sourced from Kaggle, contains 569 instances, each representing a breast cancer case. Each instance includes 32 features, encompassing the ID, diagnosis, and various attributes like radius, texture, perimeter, area, and smoothness. When we exclude ID and diagnosis, it gives us 30 features to analysis and build models. These features include various measurements related to the cell nuclei characteristics from the breast mass images, such as radius, texture, perimeter, area, smoothness, compactness, concavity, and several others. These measurements are provided as mean values, standard errors, and 'worst' or largest values taken from each image.

The dataset does not contain any missing values in any of its columns. There are 357 benign (B) cases and 212 malignant (M) cases. This distribution is slightly unbalanced but not an extreme case. Some features show stronger correlation indicating their potential for diagnosis and there are some high correlations among features suggesting redundancy.

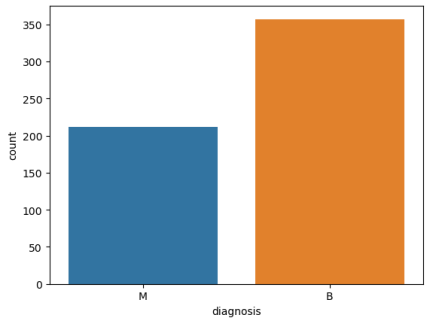

Figure 1: Distribution of Diagnosis of 569 Patients

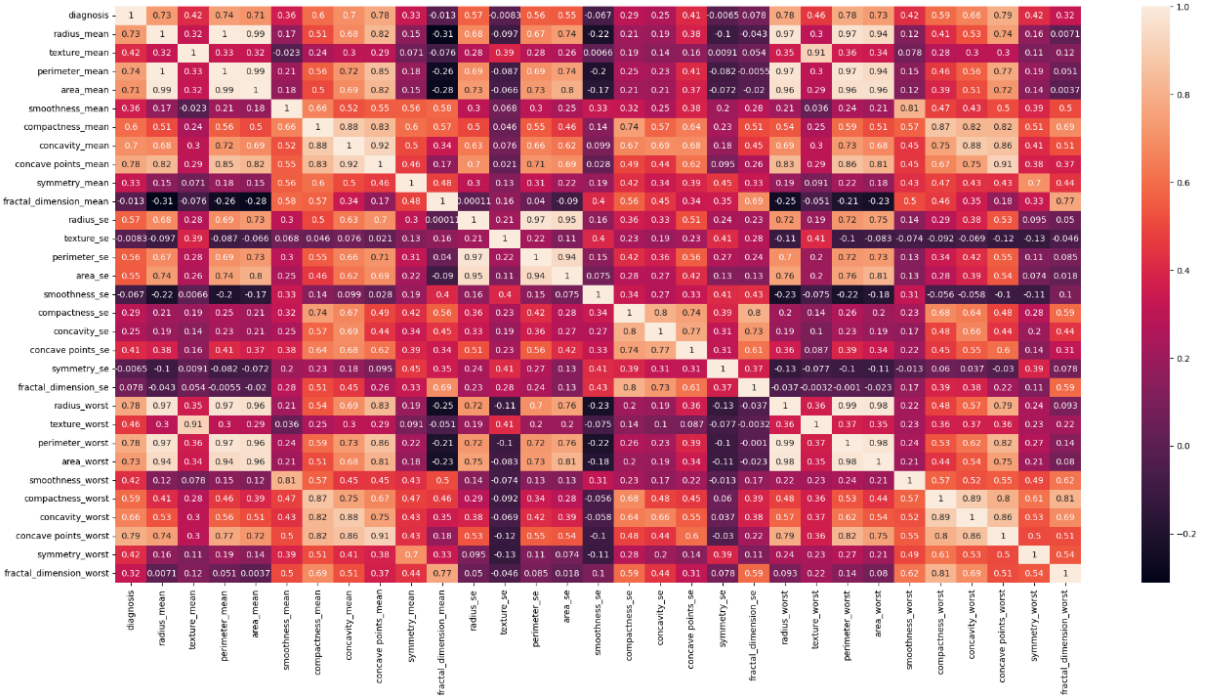

Figure 2: Correlation Heatmap of Breast Cancer Features

After we complete the data visualization, we preprocess our data before we train our machine learning models. First, we eliminate the id column since it does not give information to determine the 'diagnosis'. Then we split the dataset into training and test sets. We used 80% of our data for training and 20% of our data for test. After that, we standardize our dataset by subtraction mean and dividing by standard deviation for each feature. This ensures that all features contribute equally to predictions.

$$z = \frac{x_i - \mu}{\sigma} \qquad \text{(eq.1)}$$

Review of Machine Learning Methods

We plan to implement and compare three different machine learning algorithms since it is beneficial to try different types of algorithms considering it is not apparent which one will perform the best. There are different factors and demands affecting the choice of the model such as characteristics of the dataset, performance, interpretability, and computational efficiency.

Logistic Regression:

Logistic regression is a classification algorithm that is used to predict a binary outcome based on a set of independent variables. Since we have two possible outcomes in breast cancer classification, which are benign and malignant, it is a preferable method. If it works adequately, it may not be necessary to use more complex models. Logistic regression is a simpler and more efficient model relatively and it works well when there is a linear relationship between input features and the log odds of the target outcome.

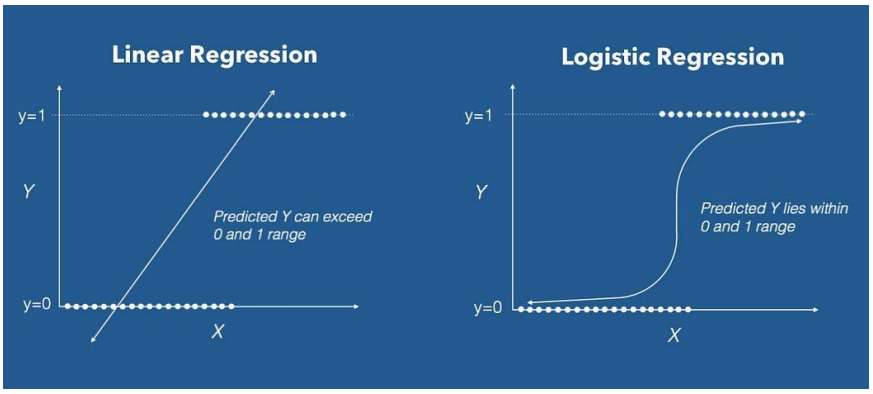

Figure 3: Linear Regression vs Logistic Regression

$$\sigma(z) = \frac{1}{1+e^{-z}} \text{ (eq.2)}$$

$$z = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \ldots + \beta_n x_n \text{ (eq.3)}$$

$$P(y = 1 \mid x) = \frac{1}{1+e^{-(\beta_0+\beta_1 x_1+\beta_2 x_2+\ldots+\beta_n x_n)}} \text{ (eq.4)}$$

$$P(y = 0 \mid x) = \frac{e^{-(\beta_0+\beta_1 x_1+\beta_2 x_2+\ldots+\beta_n x_n)}}{1+e^{-(\beta_0+\beta_1 x_1+\beta_2 x_2+\ldots+\beta_n x_n)}} \text{ (eq.6)}$$

$$\text{Likelihood: } L(w) = \prod_{i:y_i=1} P(y = 1 \mid x) \prod_{i:y_i=0} P(y = 0 \mid x) \text{ (eq.7)}$$

$$\text{-loglikelihood: } -l(w) = \sum_{i=1}^{n} [\log(1 + e^{w^T x_i}) - (1 - y_i) w^T x_i] \text{ (eq.8)}$$

$$\frac{\partial(-l(w))}{\partial w} = \sum_{i=1}^{n} \left[\frac{x_i e^{w^T x_i}}{1+e^{w^T x_i}} - (1 - y_i) x_i \right] = 0 \text{ (eq.9)}$$

To learn the parameters, we need to use gradient descent optimization.

However, logistic regression is not a good solution when using non-linear data. It also behaves as if all features are independent, but it is not the case most of the time in real-world situations like medical datasets. Another problem in terms of logistic regression may be overfitting in cases with a large number of features unless regularization techniques are applied.

Multi-layer Perceptron (Neural Network):

A multilayer perceptron is formed of input and output layers and one or more hidden layers with many neurons stacked together. Multilayer perceptron is a feedforward algorithm since inputs are combined with initial weights in a weighted sum and subjected to the activation function. Each layer is feeding the next one with result of their computation and this goes though until the output layer. However, what makes this a learning algorithm is the process of backpropagation. Backpropagation allows multilayer perceptron iteratively to adjust the weights in the network to minimize the cost function. Gradient descent is typically the optimization function used in multilayer perceptron.

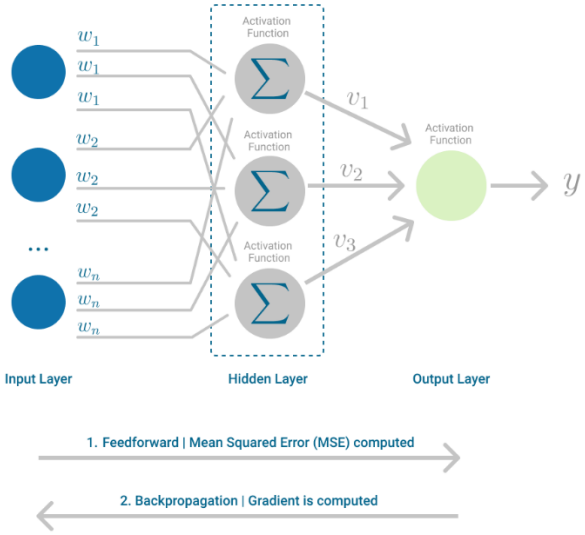

Figure 4: Visualization of Neural Network Algorithm

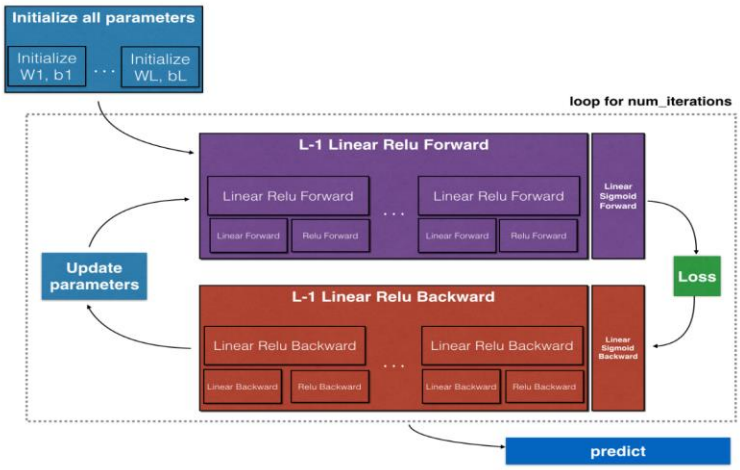

Figure 5: Neural Network Algorithm chart

$$\underbrace{\Delta_w(t)}_{\text{Gradient Current Iteration}} = \underbrace{-\varepsilon}_{\text{Bias}} \frac{\overbrace{dE}^{\text{Error}}}{\underbrace{dw_{(t)}}_{\text{Weight vector}}} + \underbrace{\alpha}_{\text{Learning Rate}} \underbrace{\Delta_{w(t-1)}}_{\text{Gradient Previous Iteration}}$$

Figure 6: Gradient Descent Formula

The reason of a neural network is selected for its ability to model non-linear relationships between features, its flexibility towards complex relationships thanks to its architecture and its adaptability to handle various data types and structures. However, the trade-off is it is hard to interpret, meaning that understanding how decisions are made may be challenging since they considered as ‘black boxes’.

Also, MLP may need more computational resources as the network size grows and it may need large dataset to work optimally.

Decision Tree-Based Algorithms (Decision Tree + Random Forest):

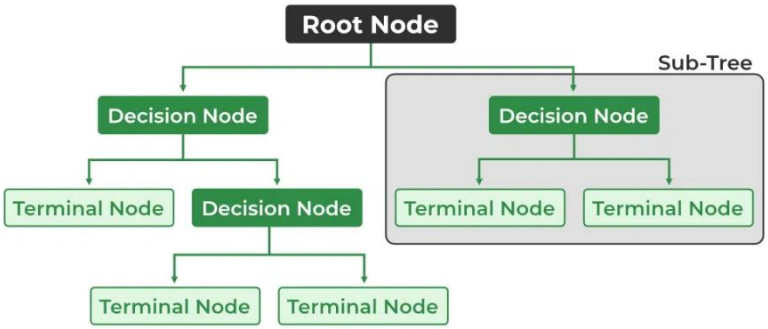

Figure 7: Decision Tree

Decision tree is a supervised learning algorithm used for both classification and regression purposes. A decision tree is built of a flow-chart like tree structure where features are denoted as internal nodes, outputs are denoted as leaf nodes and in between there are branches that denote the rules. Each branch in the decision tree represents a choice between a number of alternatives, and each leaf node represents a classification or decision. This method is particularly useful in decision analysis, allowing for a balanced representation of various outcomes. The algorithm splits the data into subsets, which are then further split into even smaller subsets, in a recursive manner. This process is known as recursive partitioning. In our project we used "Entropy" as an impurity measure. Then we calculated information gain according to equation 11 given below. After the algorithm splits dataset according to the feature that gives the highest information gain, it creates the left and right branches of the tree. It keeps splitting until if a node is 100% one class or maximum depth is reached.

$$Entropy\text{: } H(p_1) = -p_1 \log_2 p_1 - (1-p_1)\log_2(1-p_1) \text{(eq.10)}$$

$$Information\ Gain: \ IG = H(p_1^{root}) - (w^{left}H(p_1^{left}) + w^{right}H(p_1^{right})) \text{ (eq.11)}$$

Random Forest model is made up of multiple decision trees. There are three hyperparameters need to be decided before the training process which are the number of trees and the number of features sampled. Also we need to decide the hyperparameter of the decision tree, which is maximum depth. In a Random Forest, each tree operates independently, and the final decision is made based on the majority voting principle .This means the output class is the one that is the majority class of individual trees. The Random Forest algorithm improves the predictive accuracy and controls over-fitting by averaging or combining the results of different decision trees. The number of trees in the forest is a critical parameter, as a larger number of trees increases the performance and makes the model more robust, but also slows down the computation. Similarly, the node size represents the minimum size of the sample split, and the number of features sampled determines how the decision trees in the forest will be built.

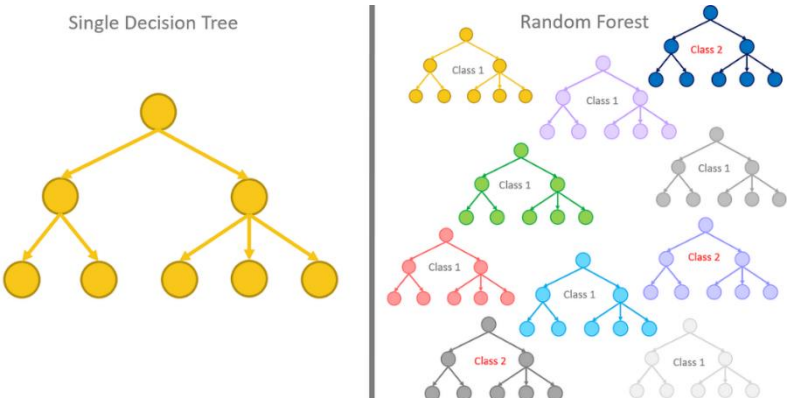

Figure 8: Single Decision Tree vs Random Forest

The reason for choosing the Random Forest is it may robust against overfitting and may provide more insight into feature importance. Moreover, like neural networks, decision trees are also good at capturing non-linearity and there is no need to scale features. However, they may have some weaknesses in terms of overfitting, they may have too complex with large data and trees may be biased through the dominant class in unbalanced data.

Simulation Setup

In this project, we used Python utilizing Jupyter Notebook and Google Colab environments for coding and execution. We imported libraries like NumPy for numerical computations, Matplotlib and Seaborn for visualizations, and Pandas for data manipulation. Then, we explored the data by reading the dataset into a Pandas DataFrame. We examined its structure and whether it has missing values using methods ‘info()’ and ‘df.isna.sum()’ methods. Then, utilizing describe() method, we acquired the statistical information about dataset such as mean, standard deviation, and other descriptive statistics of the features. Then, to prepare the data for model building ‘id’ column of the dataset is removed since it gives no information for target variable and the target variable ‘diagnosis’ converted to integer format such that ‘malignant’ corresponds to 1 and ‘benign’ corresponds to 0.

To visualize data, we generated a count plot for the ‘diagnosis’ variable to see the distribution of benign and malignant cases so that we saw dataset is a bit skewed towards bening cases. Further, to observe the correlations between features a heatmap is generated using ‘sns.heatmap()’ method. After analyzing the heatmap, we identified the features having high correlation with the diagnosis, above 0.7, and generated a pairplot using ‘sns.pairplot’ method for these highly correlated features as given

in Figure 9.

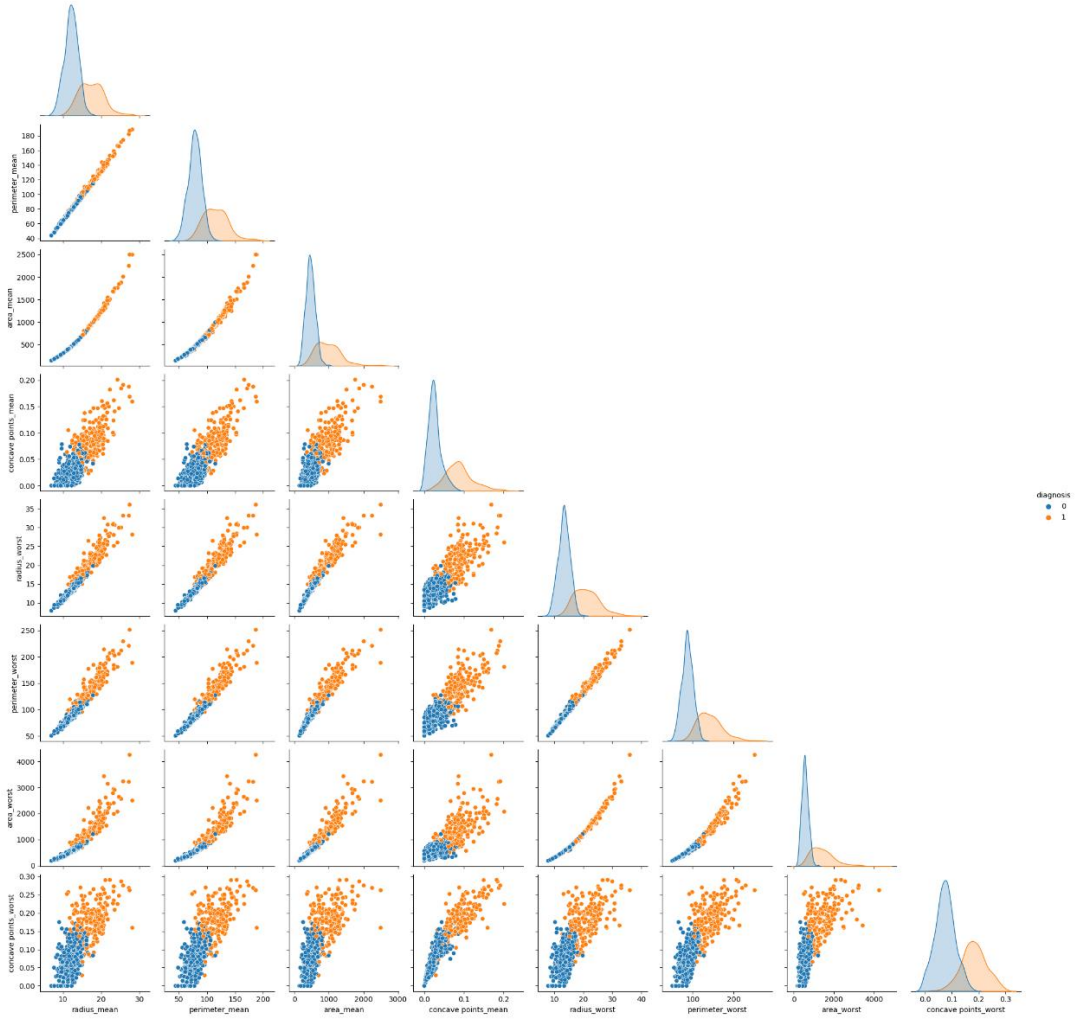

Figure 9: The Pairplot of highly correlated features

Afterwards, we generated custom functions for splitting data as 75% for training, 10% for validation, and 15% for testing as it is given feedback at the first project presentation; and standardizing the data such that it has 0 mean and standard deviation of 1.

The generated models are trained on the training set and evaluated on the validation and test sets. K-fold cross-validation is applied for the logistic regression model to give more robust results. Also, a custom 'metrics' function has been generated to calculate performance metrics such as accuracy, precision, recall and f1-score so that we can measure the effectiveness of the models in classifying breast cancer.

Simulation Results:

Logistic Regression:

The first model implemented is Logistic Regression. Utilizing our custom specific function for Logistic Regression and our other predefined functions, the model is trained and evaluated and performance metrics according to the results are calculated. To assess the performance of the model better, a K-Fold Cross Validation function is implemented and average performance metrics across all

folds are printed. Then, logistic regression model is evaluated again without using cross validation and the results are compared as given in Table 1.

	Learning rate = 0.0001	**Learning rate = 0.001**	**Learning rate = 0.01**	**Learning rate = 0.02**
Average Accuracy	0.9393	0.9536	0.9607	0.9571
Average Precision	0.9282	0.9375	0.9462	0.9379
Average Recall	0.9182	0.9455	0.9545	0.9545
Average F1-Score	0.9226	0.9412	0.9503	0.9461

Table 1: Performance Metrics of Logistic Regression with K-Fold Cross-Validation by fixing iterations but changing Learning Rate

	Iterations = 10	**Iterations = 50**	**Iterations = 100**	**Iterations = 1000**
Average Accuracy	0.9464	0.9607	0.9679	0.9821
Average Precision	0.913	0.9462	0.9727	0.9913
Average Recall	0.9545	0.9545	0.9455	0.9636
Average F1-Score	0.9333	0.9503	0.9586	0.9767

Table 2: Performance Metrics of Logistic Regression with K-Fold Cross-Validation by fixing learning rate but changing number of iterations.

According to the tables above, we chose our hyperparameters as learning rate = 0.01 and number of iterations = 100. We chose the number of iterations as 100, because of computational efficiency. Number of iterations = 1000 gave better results but is not computationally efficient and it is prone to overfitting.

Test Results:

```
Accuracy: 0.9647
Precision: 0.9583
Recall: 0.92
F1-Score: 0.9388
```

Figure 10: Results obtained by logistic regression model with test set and the given hyperparameters above.

Confusion Matrix:

	true=1	true=0
pred=1	23	1
pred=0	2	59

Figure 11: Confusion matrix of logistic regression model using the test set.

Multi-Layer Perceptron:

We implemented the multi-layer perceptron that inputs 30 standardized features of our dataset. There is one hidden layer, we changed the number of neurons in the hidden layer to observe how the results changed. We used sigmoid as the activation function. Also, we changed the learning rate and number of epochs to observe changes in the metrics.

	# of Neurons in Hidden Layer = 32	**# of Neurons in Hidden Layer=64**	**# of Neurons in Hidden Layer = 96**	**# of Neurons in Hidden Layer = 128**
Accuracy	0.9643	0.9643	0.9821	0.9286
Precision	0.9545	0.9545	1.0	0.9091
Recall	0.9545	0.9545	0.9545	0.9091
F1-Score	0.9545	0.9545	0.9767	0.9091

Table 3: Performance Metrics of Multi-layer Perceptron with different number of neurons in hidden layer, epoch is fixed at 100 and learning rate is fixed at 0.01.

	Epoch = 25	**Epoch = 100**	**Epoch = 500**	**Epoch = 1000**
Accuracy	0.9286	0.9643	0.9643	0.9464
Precision	0.9091	0.9545	0.9545	0.913
Recall	0.9091	0.9545	0.9545	0.9545
F1-Score	0.9091	0.9545	0.9545	0.9333

Table 4: Performance Metrics of Multi-layer Perceptron with 96 number of neurons in hidden layer, learning rate is fixed at 0.01 and observed for different epochs.

	Learning rate = 0.0001	**Learning rate = 0.001**	**Learning rate = 0.01**	**Learning rate = 0.05**
Accuracy	0.7321	0.8929	0.9643	0.9464
Precision	0.7692	0.9	0.9545	0.9524
Recall	0.4545	0.8182	0.9545	0.9091
F1-Score	0.5714	0.8571	0.9545	0.9302

Table 5: Performance Metrics of Multi-layer Perceptron with 96 number of neurons in hidden layer, number of epoch is 100 and observed for different learning rates.

According to the tables above, we chose our hyperparameters as number of neurons in the hidden layer = 96, number of epochs = 100 and learning rate = 0.01. Then we used test sets to obtain results.

Test Results:

```
Accuracy: 0.9882
Precision: 1.0
Recall: 0.96
F1-Score: 0.9796
```

Figure 12: Results obtained by multi-layer perceptron model with test set and the given hyperparameters above.

Confusion Matrix:

	true=1	true=0
pred=1	24	0
pred=0	1	60

Figure 11: Confusion matrix of multi-layer perceptron model using the test set.

Decision Tree:

	Maximum depth = 3	Maximum depth = 4	Maximum depth = 6	Maximum depth = 8
Accuracy	0.9107	0.9643	0.9464	0.9464
Precision	1.0	1.0	0.9524	0.9524
Recall	0.7727	0.9091	0.9091	0.9091
F1-Score	0.8718	0.9524	0.9302	0.9302

Table 6: Performance Metrics of Decision Tree model with different maximum depths.

According to the table above, we chose the maximum depth for our decision tree as 4. Then we used test sets to obtain results.

Test Results:

```
Accuracy: 0.9294
Precision: 0.9524
Recall: 0.8
F1-Score: 0.8696
```

Figure 12: Results obtained by decision tree model with test set and for maximum depth = 4.

Confusion Matrix:

	true=1	true=0
pred=1	20	1
pred=0	5	59

Figure 13: Confusion matrix of decision tree model using the test set.

Random Forest:

	Maximum features = 4	Maximum features = 6	Maximum features= 8	Maximum features = 10
Accuracy	0.9107	0.9464	0.9286	0.9286
Precision	1.0	1.0	1.0	1.0
Recall	0.7727	0.8636	0.8182	0.8182
F1-Score	0.8718	0.9268	0.9	0.9

Table 7: Performance Metrics of Random Forest model with different maximum features and the other hyperparameters are fixed.

	# of Tree = 4	# of Tree = 5	# of Tree = 20	# of Tree = 80	# of Tree = 320
Accuracy	0.8929	0.9464	0.8929	0.8929	0.9286
Precision	1.0	1.0	1.0	1.0	1.0
Recall	0.7273	0.8636	0.7273	0.7273	0.8182
F1-Score	0.8421	0.9268	0.8421	0.8421	0.9

Table 8: Performance Metrics of Random Forest model with different number of trees and the other hyperparameters are fixed.

According to the tables above, we chose our hyperparameters as maximum features= 6, number of trees = 5 and maximum depth obtained from decision tree result as 4. Then we used test sets to obtain results.

Test Results:

```
Accuracy: 0.9412
Precision: 1.0
Recall: 0.8
F1-Score: 0.8889
```

Figure 14: Results obtained by random forest model with test set and the given hyperparameters above.

Confusion Matrix:

	true=1	true=0
pred=1	20	0
pred=0	5	60

Figure 15: Confusion matrix of random forest model using the test set.

	Logistic Regression	MLP	Decision Tree	Random Forest
Accuracy	0.9647	0.9882	0.9294	0.9412
Precision	0.9583	1.0	0.9524	1.0
Recall	0.92	0.96	0.8	0.8
F1-Score	0.9388	0.9796	0.8696	0.8889

Run time of algorithms:

Logistic Regression = 0.320241 seconds

MLP = 0.546323 seconds

Decision Tree = 11.371583 seconds

Random Forest = 6.183208 seconds

According to the results multi-layer perceptron model appears to be the best model for our project in terms of performance metrics and fast execution time.

Challenges and Solutions

We faced several challenges during the project. The challenge that we faced was implementing algorithms from scratch. At the first we write the algorithms mostly using for loops but then it appears to be computationally inefficient. Then we changed our functions and implemented them in vectorized way. We used validation sets to select the hyperparameters and tested the algorithms using test sets. There are some differences between results of the validation and test sets, it can be because of overfitting.

Contribution of Each Group Member

WEEKS	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*	*13*	*14*
Dataset Research	Ö	Ö												
Methodology Research		S	S											
Literature Review			S	ÖS	ÖS									
Project Proposal						ÖS								
Data Preprocessing							Ö							
Feature Selection							S							
Performance Validation								S						
Logistic Regression								Ö	Ö					
First Progress Report									ÖS	ÖS				
Multilayer Perceptron									S	S	Ö			
Random Forest												Ö	ÖS	

Table 10: Gant Chart for Work Allocation, Ö: Ömer, S: Selin.

Conclusion

This project aims to contribute to the field of medical diagnostics by developing a reliable and efficient tool for breast cancer classification. By comparing different machine learning methods, we hope to identify the most effective approach for this crucial task. We did data preprocessing to the breast cancer dataset, conducting exploratory data analysis on it, feature selection based on the correlations of features and visualizing the relationship between them. We implemented four different machine learning algorithms and found the best hyperparameters for each algorithm. After obtaining the results, the multi-layer perceptron model appears to be the best model for this crucial task.

References

1. Y. H., "Breast Cancer Dataset," Kaggle. [Online]. Available: https://www.kaggle.com/datasets/yasserh/breast-cancer-dataset/data. [Accessed: Nov. 20, 2023].
2. World Health Organization, "Breast Cancer," WHO. [Online]. Available: https://www.who.int/news-room/fact-sheets/detail/breast-cancer. [Accessed: Nov. 20, 2023].
3. [Pant, A.], "Introduction to Logistic Regression," Towards Data Science, [Online]. Available: https://towardsdatascience.com/introduction-to-logistic-regression-66248243c148. [Accessed: Nov. 20, 2023].
4. [Bento, C.], "Multilayer Perceptron Explained with a Real-Life Example and Python Code (Sentiment Analysis)," Towards Data Science, [Online]. Available: https://towardsdatascience.com/multilayer-perceptron-explained-with-a-real-life-example-and-python-code-sentiment-analysis-cb408ee93141. [Accessed: Nov. 20, 2023].
5. [GeeksforGeeks], "Decision Tree," GeeksforGeeks. [Online]. Available: https://www.geeksforgeeks.org/decision-tree/. [Accessed: Nov. 20, 2023].

Appendix

We write the code in jupyter notebook, so we will provide code as notebook inputs.

```
# -*- coding: utf-8 -*-
"""Data_Project_v9.ipynb

Automatically generated by Colaboratory.

Original file is located at
    https://colab.research.google.com/drive/1bsQtQj8h5iXRn6Emund2VY2hRTaqrSsm

# EE485 Term Project - Breast Cancer Classification
                        Ömer Tuğrul - Selin Ataş
"""

from google.colab import files
from google.colab import drive
drive.mount("/content/gdrive")

"""# Data Visualization and Preprocessing

**Import Libraries**
"""

import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
import datetime

np.random.seed(38)
```

```
df = pd.read_csv("breast-cancer.csv")

df

df.info()

df.shape

df.isna().sum()

df.describe().transpose()

#We need to split the dataset into train and test set
#We dont need id column in our model, so I will delete that column
df.drop('id', axis=1, inplace=True)

df

df.shape

sns.countplot(x='diagnosis',data=df);

df['diagnosis'] = (df['diagnosis'] == 'M').astype(int)#I want to change the target value integer
df

plt.figure(figsize=(25,12))
sns.heatmap(df.corr(),annot=True)

df['diagnosis'].value_counts() #Malign(hastalıklı) means 1 and beign means(0)

corr_matrix = df.corr()
```

"""In cell 13, we found that some features are highly correlated or discorrelated with the diagnosis, so we will extract the scatter plot of these variables."""

```
high_corr_matrix = corr_matrix[abs(corr_matrix['diagnosis'])>= 0.7]
high_corr_matrix
```

```
high_corr_list = high_corr_matrix.index.to_list()
high_corr_list
```

```
sns.pairplot(df[high_corr_list],hue='diagnosis',corner=True)
```

```
df.head(5)
```

"""# Now continue with splitting the dataset"""

```
X= df.drop('diagnosis',axis = 1)
y = df['diagnosis']
```

```
X.values.shape,y.values.shape
```

```
type(X.values),type(y.values)
```

"""According to the feedback given in the first presentation we changed the split to train-75% validation-10% and test-15&"""

```
def train_val_test_split(df, y_column, test_size=0.15, val_size=0.10):
    shuffled_indices = np.random.permutation(df.index)

    test_set_size = int(df.shape[0] * test_size)
    val_set_size = int(df.shape[0] * val_size)

    test_indices = shuffled_indices[:test_set_size]
    val_indices = shuffled_indices[test_set_size:test_set_size + val_set_size]
```

```
    train_indices = shuffled_indices[test_set_size + val_set_size:]

    train_df = df.iloc[train_indices]
    val_df = df.iloc[val_indices]
    test_df = df.iloc[test_indices]

    X_train = train_df.drop(columns=[y_column])
    y_train = train_df[y_column]
    X_val = val_df.drop(columns=[y_column])
    y_val = val_df[y_column]
    X_test = test_df.drop(columns=[y_column])
    y_test = test_df[y_column]

    return X_train, X_val, X_test, y_train, y_val, y_test

X_train,X_val,X_test,y_train,y_val,y_test = train_val_test_split(df,y_column ='diagnosis')

X_train.shape,X_val.shape,X_test.shape,y_train.shape,y_val.shape,y_test.shape

def standardize_data(X_train, X_val, X_test):
    mu = X_train.mean()
    sigma = X_train.std()

    X_train_s = (X_train - mu) / sigma
    X_val_s = (X_val - mu) / sigma
    X_test_s = (X_test - mu) / sigma

    return X_train_s, X_val_s, X_test_s

def metrics(y_pred,y_test):
    tp, fp, tn, fn = 0, 0, 0, 0
    for i in range(len(y_test)):
```

```
    true, pred = y_test.iloc[i], y_pred[i]
    if true == 1 and pred == 1:
        tp += 1
    elif true == 0 and pred == 1:
        fp += 1
    elif true == 0 and pred == 0:
        tn += 1
    elif true == 1 and pred == 0:
        fn += 1
if (tp + fp + tn + fn) != 0:
    accuracy = (tp + tn) / (tp + fp + tn + fn)
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 = 2 * (precision * recall) / (precision + recall)
else:
    f1 = 0

conf_dict = {'true=1':[tp,fn],'true=0':[fp,tn]}
conf_matrix = pd.DataFrame(data=conf_dict)
conf_matrix.index = ['pred=1','pred=0']
```

```
    return accuracy,precision,recall,f1,conf_matrix

"""# MODEL1: LOGISTIC REGRESSION"""

class LogisticRegression:
    def __init__(self,l_rate = 0.01, iterations = 100):
        self.l_rate = l_rate
        self.iterations = iterations
        self.w = None

    def logistic_function(self,x):
        # P(Y=1|X=Xk)
        return 1/(1+np.exp(-x))

    def fit_logistic(self,X,y):
        m,n = X.shape #m=row, n=column/feature
        self.w = np.zeros(n)

        #Do gradient descent algortihm to update weights
        for i in range(self.iterations):
            z = np.dot(X,self.w)
            pred = self.logistic_function(z)

            error = y - pred
            #use cross entropy loss to calculate gradient.
            gradient = np.dot(X.T,error)  #Sum of errors weighted by its features
            self.w = self.w + self.l_rate*gradient

    def predict_prob(self,X):
        z = np.dot(X,self.w)
        return self.logistic_function(z)
```

```
    def predict(self,X,threshold = 0.5):
        probs = self.predict_prob(X)
        predictions = []
        for prob in probs:
            if prob >= threshold:
                predictions.append(1)
            else:
                predictions.append(0)

        return np.array(predictions)

def kfold_cv(df, k=5, l_rate=0.01, iterations=100):
    fold_size = len(df) // k
    accuracies = []
    precisions = []
    recalls = []
    f1_scores = []

    for fold in range(k):
        start = fold * fold_size
        if fold < k - 1:
            end = start + fold_size
        else:
            end = len(df)
        test_df = df.iloc[start:end]
        train_df = pd.concat([df.iloc[:start], df.iloc[end:]])

        X_train = train_df.drop('diagnosis', axis=1)
        y_train = train_df['diagnosis']
        X_test = test_df.drop('diagnosis', axis=1)
        y_test = test_df['diagnosis']
```

```
        X_train_s,X_val_s, X_test_s = standardize_data(X_train,X_val, X_test)

        logistic_model = LogisticRegression(l_rate=l_rate, iterations=iterations)
        logistic_model.fit_logistic(X_train_s, y_train)

        y_pred = logistic_model.predict(X_val_s)

        accuracy,precision,recall,f1,conf_matrix = metrics(y_pred,y_val)

        accuracies.append(accuracy)
        precisions.append(precision)
        recalls.append(recall)
        f1_scores.append(f1)

    print("Average Accuracy:", round(np.mean(accuracies),4))
    print("Average Precision:", round(np.mean(precisions),4))
    print("Average Recall:", round(np.mean(recalls),4))
    print("Average F1-Score:", round(np.mean(f1_scores),4))

kfold_cv(df)

"""Now test the same logistic regression model without cross validation"""

X_train_s,X_val_s,X_test_s = standardize_data(X_train,X_val,X_test)

start_time_log = datetime.datetime.now()
logistic_model_2 = LogisticRegression(l_rate=0.01,iterations=100)
logistic_model_2.fit_logistic(X_train_s,y_train)
end_time_log = datetime.datetime.now()
elapsed_time_log = end_time_log - start_time_log
print(f"Elapsed time to run logistic regression algorithm is: {elapsed_time_log.total_seconds()}")
```

```
y_pred_2 = logistic_model_2.predict(X_test_s)

accuracy,precision,recall,f1,conf_matrix = metrics(y_pred_2,y_test)
print("Accuracy:", round(accuracy,4))
print("Precision:", round(precision,4))
print("Recall:", round(recall,4))
print("F1-Score:", round(f1,4))
conf_matrix

"""# MODEL 2: MLP"""

start_time_mlp = datetime.datetime.now()

class MLP:
    def __init__(self, input_size=30, hidden_size=32):
        np.random.seed(38)
        self.w1 = np.random.randn(input_size, hidden_size)
        self.b1 = np.zeros(hidden_size)
        self.w2 = np.random.randn(hidden_size, 1)
        self.b2 = np.zeros(1)

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

    def sigmoid_derivative(self, x):
        return x * (1 - x)

    def forward(self, x):
        self.z1 = np.dot(x, self.w1) + self.b1
        self.a1 = self.sigmoid(self.z1)
        self.z2 = np.dot(self.a1, self.w2) + self.b2
        self.a2 = self.sigmoid(self.z2)
```

```
    return self.a2.squeeze()

def backward(self, x, y, y_pred, learning_rate):

    y = np.array(y).reshape(-1, 1)
    y_pred = y_pred.reshape(-1, 1)

    error1 = y_pred - y
    sig_deriv = self.sigmoid_derivative(y_pred)

    d_w2 = np.dot(self.a1.T, error1 * sig_deriv)
    d_b2 = np.sum(error1 * sig_deriv, axis=0)

    error2 = np.dot(error1 * sig_deriv, self.w2.T) * self.sigmoid_derivative(self.a1)
    d_w1 = np.dot(x.T, error2)
    d_b1 = np.sum(error2, axis=0)

    self.w1 -= learning_rate * d_w1
    self.b1 -= learning_rate * d_b1
    self.w2 -= learning_rate * d_w2
    self.b2 -= learning_rate * d_b2

def train(self, X_train, y_train, epochs, learning_rate):
    for epoch in range(epochs):
        y_pred = self.forward(X_train)
        self.backward(X_train, y_train, y_pred, learning_rate)

def evaluate(self, X_val, y_val, threshold):
    y_pred = self.forward(X_val) > threshold
    accuracy, precision, recall, f1, conf_matrix = metrics(y_pred, y_val)
    return accuracy, precision, recall, f1, conf_matrix
```

```
model_mlp = MLP(input_size = 30, hidden_size=96)
model_mlp.train(X_train_s, y_train, epochs=100, learning_rate=0.01)
end_time_log = datetime.datetime.now()
elapsed_time_log = end_time_log - start_time_log
print(f"Elapsed time to run MLP algorithm is: {elapsed_time_log.total_seconds()}")
accuracy,precision,recall,f1,conf_matrix = model_mlp.evaluate(X_val_s, y_val,0.5)
print("Accuracy:", round(accuracy,4))
print("Precision:", round(precision,4))
print("Recall:", round(recall,4))
print("F1-Score:", round(f1,4))
conf_matrix

end_time_mlp = datetime.datetime.now()
elapsed_time_mlp = end_time_mlp - start_time_mlp
print(f"Elapsed time to run multi-layer perceptron algorithm is: {elapsed_time_mlp.total_seconds()}")

"""Now hyperparameters are fixed so test the model with input size 30 and hidden size 64, 100
epoch"""

start_time_mlp = datetime.datetime.now()
model_mlp = MLP(input_size = 30, hidden_size=96)
model_mlp.train(X_train_s, y_train, epochs=100, learning_rate=0.01)
end_time_mlp = datetime.datetime.now()
elapsed_time_mlp = end_time_mlp - start_time_mlp
print(f"Elapsed time to run multi-layer perceptron algorithm is: {elapsed_time_mlp.total_seconds()}")
accuracy,precision,recall,f1,conf_matrix = model_mlp.evaluate(X_test_s, y_test,0.5)
print("Accuracy:", round(accuracy,4))
print("Precision:", round(precision,4))
print("Recall:", round(recall,4))
print("F1-Score:", round(f1,4))
conf_matrix

"""# Model 3- Decision Tree"""
```

Ömer Tuğrul – 22002723
Selin Ataş- 21902858

```
class Node:
    def __init__(self, feature=None, threshold=None, left=None, right=None, value=None):
        self.feature = feature
        self.threshold = threshold
        self.left = left
        self.right = right
        self.value = value


class DecisionTree:
    def __init__(self, max_depth=None):
        self.max_depth = max_depth
        self.root = None

    def entropy(self, y):
        _, counts = np.unique(y, return_counts=True)
        p = counts / counts.sum()
        entropy_sum = 0
        for prob in p:
            if prob > 0:
                entropy_sum -= prob * np.log2(prob)
        return entropy_sum

    def split(self, X, y, feature, threshold):
        left_indices = []
        right_indices = []

        for i in range(len(X)):
            if X[i, feature] <= threshold:
                left_indices.append(i)
            else:
                right_indices.append(i)
```

```
    left_y = y[left_indices]
    right_y = y[right_indices]

    return left_y, right_y



def information_gain(self, y, left, right):
    ent_parent = self.entropy(y)
    ent_left = self.entropy(left)
    ent_right = self.entropy(right)
    w_left = len(left) / len(y)
    w_right = len(right) / len(y)
    ig = ent_parent - w_left * ent_left - w_right * ent_right
    return ig

def best_split(self, X, y):
    best_feature_index = None
    best_threshold = None
    best_information_gain = -1

    for feature_index in range(X.shape[1]):
        unique_values = np.unique(X[:, feature_index])

        for value in unique_values:
            left_split, right_split = self.split(X, y, feature_index, value)

            if len(left_split) == 0 or len(right_split) == 0:
                continue

            information_gain = self.information_gain(y, left_split, right_split)

            if information_gain > best_information_gain:
```

```
                best_information_gain = information_gain
                best_feature_index = feature_index
                best_threshold = value

        return best_feature_index, best_threshold

    def build_tree(self, X, y, depth=0):
        if len(np.unique(y)) == 1 or depth == self.max_depth:
            return Node(value=np.unique(y)[0])

        best_feature, best_threshold = self.best_split(X, y)

        if best_feature is None:
            return Node(value=np.bincount(y).argmax())

        left_indices = []
        right_indices = []
        for i in range(len(X)):
            if X[i, best_feature] <= best_threshold:
                left_indices.append(i)
            else:
                right_indices.append(i)

        left_subtree = self.build_tree(X[left_indices], y[left_indices], depth + 1)
        right_subtree = self.build_tree(X[right_indices], y[right_indices], depth + 1)
        return Node(feature=best_feature, threshold=best_threshold, left=left_subtree,
right=right_subtree)


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

    def predict_sample(self, x, node):
```

```
        if node.value is not None:
            return node.value
        if x[node.feature] <= node.threshold:
            return self.predict_sample(x, node.left)
        else:
            return self.predict_sample(x, node.right)

    def predict(self, X):
        predictions = []
        for sample in X:
            predicted_class = self.predict_sample(sample, self.root)
            predictions.append(predicted_class)
        return np.array(predictions)

X_train_np = X_train.to_numpy()
y_train_np = y_train.to_numpy()
X_test_np = X_test.to_numpy()
X_val_np = X_val.to_numpy()

s
model_dt = DecisionTree(max_depth=8)
model_dt.fit(X_train_np, y_train_np)

y_pred_dt = model_dt.predict(X_val_np)
accuracy, precision, recall, f1, conf_matrix = metrics(y_pred_dt, y_val)
print("Accuracy:", round(accuracy,4))
print("Precision:", round(precision,4))
print("Recall:", round(recall,4))
print("F1-Score:", round(f1,4))
conf_matrix

"""Testing the model with chosen hyperparams
```

```
"""

np.random.seed(38)

start_time_log = datetime.datetime.now()
model_dt_final = DecisionTree(max_depth=4)
model_dt_final.fit(X_train_np, y_train_np)
end_time_log = datetime.datetime.now()
elapsed_time_log = end_time_log - start_time_log
print(f"Elapsed time to run Decision Tree algorithm is: {elapsed_time_log.total_seconds()}")
y_pred_dt_final = model_dt_final.predict(X_test_np)
accuracy, precision, recall, f1, conf_matrix = metrics(y_pred_dt_final, y_test)
print("Accuracy:", round(accuracy,4))
print("Precision:", round(precision,4))
print("Recall:", round(recall,4))
print("F1-Score:", round(f1,4))
conf_matrix

"""# Model 4 - Random Forest"""

class RandomForest:
    def __init__(self, n_trees=100, max_features=None, max_depth=None):
        np.random.seed(38)
        self.n_trees = n_trees
        self.max_features = max_features
        self.max_depth = max_depth
        self.trees = []

    def bootstrap_sample(self, X, y):
        indices = np.random.randint(0, len(X), len(X))
```

```
    X_sample = X[indices]
    y_sample = y[indices]

    return X_sample, y_sample

def feature_subset(self, X):
    if self.max_features is not None and self.max_features < X.shape[1]:
        selected_features = np.random.choice(X.shape[1], self.max_features, replace=False)
        X_subset = X[:, selected_features]
        return X_subset, selected_features

    else:
        selected_features = None
        X_subset = X
        return X_subset, selected_features

def fit(self, X, y):
    self.trees = []
    for _ in range(self.n_trees):
        X_sample, y_sample = self.bootstrap_sample(X, y)
        X_sample, features = self.feature_subset(X_sample)
        tree = DecisionTree(max_depth=self.max_depth)
        tree.fit(X_sample, y_sample)
        self.trees.append((tree, features))

def majority_vote(self, predictions):
    majority_votes = np.round(predictions / self.n_trees).astype(int)
    return majority_votes

def predict(self, X):
    predictions = np.zeros(len(X))
    for tree, features in self.trees:
```

```
            if features is not None:
                # Subset the features using the selected feature indices
                preds = tree.predict(X[:, features])
            else:
                preds = tree.predict(X)
            predictions += preds
        return self.majority_vote(predictions)

X_train_np = X_train.to_numpy()
y_train_np = y_train.to_numpy()
X_test_np = X_test.to_numpy()
start_time_log = datetime.datetime.now()
model_rf = RandomForest(n_trees = 4,max_features = 6,max_depth = 4)
model_rf.fit(X_train_np, y_train_np)
end_time_log = datetime.datetime.now()
elapsed_time_log = end_time_log - start_time_log
print(f"Elapsed time to run Random Forest algorithm is: {elapsed_time_log.total_seconds()}")
y_pred_rf = model_rf.predict(X_val_np)
accuracy, precision, recall, f1, conf_matrix = metrics(y_pred_rf, y_val)
print("Accuracy:", round(accuracy,4))
print("Precision:", round(precision,4))
print("Recall:", round(recall,4))
print("F1-Score:", round(f1,4))
conf_matrix

"""Now test the model with the found hyperparameters above"""

model_rf_final = RandomForest(n_trees = 5 ,max_features = 6,max_depth = 4)
model_rf_final.fit(X_train_np, y_train_np)

y_pred_rf_final = model_rf_final.predict(X_test_np)
accuracy, precision, recall, f1, conf_matrix = metrics(y_pred_rf_final, y_test)
```

```
print("Accuracy:", round(accuracy,4))

print("Precision:", round(precision,4))

print("Recall:", round(recall,4))

print("F1-Score:", round(f1,4))

conf_matrix
```