Breast Cancer Diagnosis Using Artificial Neural Networks (ANN)

I. Abstract:

Breast Cancer is the second cause of death among women. Earlier detection followed by the treatment can reduce the risk. There will be some cases where even the medical professionals can make mistakes in identifying the disease. So, with the help of the technology in Machine Learning and Artificial Intelligence can substantially improve the diagnosis accuracy. Artificial Neural Networks (ANN) has been widely in healthcare domain so as in detecting the cancer in earlier stage. Here, I am proposing a **Gradient based Back Propagation Artificial Neural Networks** (GBP- ANN). The development of this technique is promising as intelligent component in medical decision support systems.

II. Introduction:

Artificial Neural Networks (ANN) is one of the best Artificial Intelligence techniques for common Data Mining tasks, such as classification and regression problems (Supervised Learning). A lot of research already showed that ANN delivered good accuracy in breast cancer diagnosis. In this method, first it involves in building the network for the model, parameters to be tuned in the beginning of the training process such as number of input nodes, hidden nodes, output nodes and initial weights, learning rates, and activation function. We know that it takes long time for training process due to complex architecture and parameter update process in each iteration and that has more computation cost. So, to overcome this I am using the Gradient based Back Propagation techniques and to speed up the computation I am leveraging the **Deep Learning platform** (**Cloud GPU**) called **FloydHub.**

In this project, I am going to implement the Artificial Neural Network from scratch for efficiently detecting the Breast Cancer. Here to calculate the performance of the model I am using various performance metrics like Accuracy, Specificity, Sensitivity, Recall, Precision and visualizing it by using Confusion matrix.

III. Description:

The main goal of this project is to detect or predict whether a tumor found in breast cancer is a **Malignant** or **Benign.**

Dataset:

The Breast Cancer Wisconsin (Diagnostic) dataset was taken from a Kaggle competition and can be found at UCI Machine Learning Repository. The dataset has a dimension of 569 x 32, it has 30 real attributes and one numeric attribute (id field), and one categorical attribute, which is a class label. Since this is a two-class classification problem which is also called as Binary Classification, so this dataset has two class values for diagnosis they are M (Malignant) and B (Benign).

Ten real-valued features are computed for each cell nucleus:

a) radius (mean of distances from center to points on the perimeter) b) texture (standard deviation of gray-scale values) c) perimeter d) area e) smoothness (local variation in radius lengths)f) compactness (perimeter^2 / area - 1.0) g) concavity (severity of concave portions of the contour) h) concave points (number of concave portions of the contour) i) symmetry j) fractal dimension ("coastline approximation" - 1)

The mean, standard error and "worst" or largest (mean of the three largest values) of these features were computed for each image, resulting in 30 features. For instance, field 3 is Mean Radius, field 13 is Radius SE, field 23 is Worst Radius.

All feature values are recorded with four significant digits.

This dataset doesn't contain any missing attribute values. The class distribution is 357 benign and 212 malignant.

Features are computed from a digitized image of a fine needle aspirate (FNA) of a breast mass. They describe characteristics of the cell nuclei present in the image.

IV. Implementation:

There are mainly two steps involved in this project

1. Data Preparation
2. Building the Model

1. Data Preparation:

This is an important step in a cleaning and processing a data that is unstructured with inconsistencies. The step is crucial for the model building, it is worth noting that good data is more important that a good model.

The first step is to import all libraries that we are going to use in the project like numpy, pandas, seaborn, etc., and then load in our dataset.

```
# importing the libraries needed in this project

import numpy as np
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.metrics import confusion_matrix

from sklearn.preprocessing import LabelEncoder
from sklearn.preprocessing import StandardScaler

# Data Preparation Stage

# Loading the data into a dataframe
data = pd.read_csv('C:/Users/sbhavaniprasad/Desktop/Data Mining/breast-cancer-wisconsin-data/data.csv')
print('Dimension of the dataset : ', data.shape)
print(data.head())
```

```
C:\Users\ebhavaniprasad\PycharmProjects\KyleProject\venv\Scripts\python.exe C:/Users/ebhavaniprasad/PycharmProjects/KyleProject/Newton
C:\Users\ebhavaniprasad\PycharmProjects\KyleProject\venv\lib\site-packages\sklearn\externals\joblib\externals\cloudpickle\cloudpickle.py:47:
  import imp
Dimension of the dataset :  (569, 33)
         id diagnosis     ...       fractal_dimension_worst  Unnamed: 32
0    842302         M     ...                       0.11890          NaN
1    842517         M     ...                       0.08902          NaN
2  84300903         M     ...                       0.08758          NaN
3  84348301         M     ...                       0.17300          NaN
4  84358402         M     ...                       0.07678          NaN
```

Here it is clear that dataset contains 569 samples with 33 features. The last column unnamed: 32 is an empty column and it is of no use, so first we are going to remove this column.

```
# Removing the empty column from the dataset
del data['Unnamed: 32']
```

Next, we are going to separate our feature data and label data (target). The feature data is that we are going to build and learn from our model and the target data is a diagnosis, which says whether the tumor is malignant or benign.

```
# Separating the feature variables and class variable(target variable)

X = data.iloc[:, 2:].values       # Feature variable
Y = data.iloc[:, 1].values        # Actual class label
print(type(Y))
print("\n Actual Class Labels : ", Y)
```

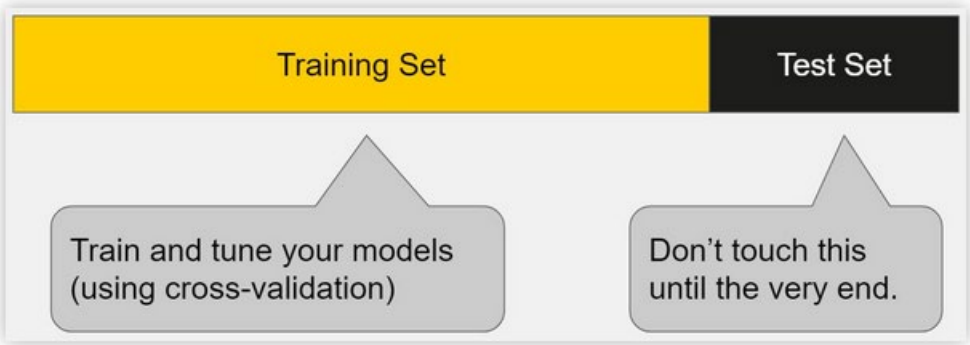
From the diagnosis column labelled as M or B. We humans can interpret this, but machine can’t so in order to make easy for the model here we are going to decode the class M and B to 1 and 0 respectively.

```
# Class Label encoding M & B to 1 & 0
encoder = LabelEncoder()
Y = encoder.fit_transform(Y)
print('After Encoding : ', Y)
```

Next, we are going to split the data into two sets i.e., training and testing sets. Here we will randomly sample from our data and split it into 70:30

ratio, the reason why we are randomly selecting the data from the whole data is that we don't want to select the data that are highly correlated that might affect our results.

The reason behind the splitting the data into training and test sets is that we can't train the model on the entire data because it will lead to ***overfitting***. That is, model will perform well on the training data and fails on the new unseen data (i.e., training error is low and testing error is high).



Train-Test Split

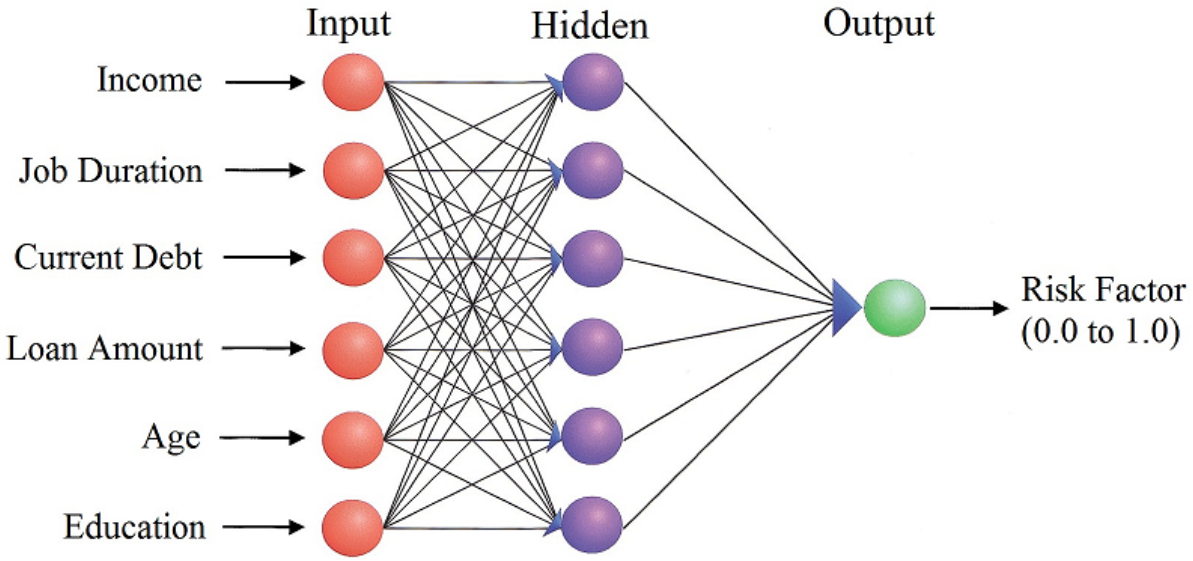
```
# Splitting data into test and training sets and randomly selecting in order to bias
# (sometimes they are highly correlated data)

X_train, X_test, Y_train, Y_test = train_test_split(X, Y, test_size=0.3, random_state=0)
```

The final task in this data preparation phase is ***feature scaling,*** here we are going to convert of data so that all values are scaled to be within the same range. It makes easier for our model to predict and to perform faster.

```
# Scaling our training data (feature scaling)
# Each feature in our dataset now will have a mean = 0 and standard deviation = 1

sc = StandardScaler()
X_train = sc.fit_transform(X_train)
X_test = sc.fit_transform(X_test)
```

The StandardScaler function will transform each variable of our data to have a mean =0 and standard deviation =1.

3. Building the Model:



Artificial Neural Network (ANN):

The word Neural Network suggest that this are designed from the architecture of our human brain. Electrical signals are sent through layers of neurons in our brain that are connected and eventually forms an output and brain will act as per the output. Like our brains, neural networks have several layers made up of perceptron nodes(neurons).

Each of this node in network simply hold a number. This node then has their number multiplied by the weight connected to them by next layer. The data is inputted to the first layer of a network, there is a node for each feature in our data (In our case, there are 30 features, so we will have 30 input nodes). These inputs are then passed to each layers of the network with layer weights applied to them. Once the inputs make it through the entire network we are left with our output (where we will have a single output node that will be either 0 or 1 (Benign or Malignant). This process is known as the **Forward Propagation**. The next step is a **Backward Propagation**, this is where actual learning occurs and here, we will check how well our model predicted the outcome. We then move backward through the

layers and updates weights of the layers in order to improve our prediction. Nodes that are more important will have higher weight value.

Building Blocks:

Till now we have seen a big picture of what the ANN is. So, here I am going to implement a neural network by dividing into subparts or several logical steps. They are as follows:

1. Network Initialization.
2. Forward Propagation
3. Back Propagation
4. Update Weights
5. Testing and Evaluation

1. Network Initialization:

In this step, we are going to initialize our network and we will define parameter like number of nodes (Input, Hidden, Output), learning rate, number of epochs, initial random weights for each of layer.

Input nodes:

The number of features in our data is equal to the number of the input nodes (In our project, there are 30 feature and 30 input nodes).

Hidden Nodes:

This is a number of nodes in our hidden layer. Here I am considering 12 hidden nodes.

Output Nodes:

This indicates the number of nodes that output layer will have. Since we are solving a Binary Classification problem, where there are 2 possible outputs 0 or 1. So, here I am using 1 node for output layer.

Learning Rate:

The Learning Rate will determine how much we adjust our weights during the back-propagation phase. A high learning rate will make our network to progress quickly and converge very quickly but it will adjust weights too much and in cause for high error rate instead of decreasing it. Whereas, low learning rate will make network to slow in learning and it generalizes better and converges very slowly and guarantees a low error rate (better result). So, here I choose 0.1 as a learning rate.

Epochs:

It defines number of times we train our model on our dataset.

```
def __init__(self, X, Y, X_test, Y_test, hidden_nodes=12, learninig_rate=0.1, epochs=5000):
    # data

    self.X = X
    self.Y = Y[:, None]
    self.X_test = X_test
    self.Y_test = Y_test

    # defining parameters

    np.random.seed(4)
    self.input_nodes = len(X[0])     # number of features in the training data
    self.hidden_nodes = hidden_nodes
    self.output_npdes = self.Y.shape[1]
    self.learning_rate = learninig_rate

    # initializing the weights for our network

    self.w1 = 2 * np.random.random((self.input_nodes, self.hidden_nodes)) - 1
    self.w2 = 2 * np.random.random((self.hidden_nodes, self.output_npdes)) - 1

    self.train(epochs)  # Since we have to train our model for many times we here pass epochs count
    self.test()
```

2. Forward Propagation:

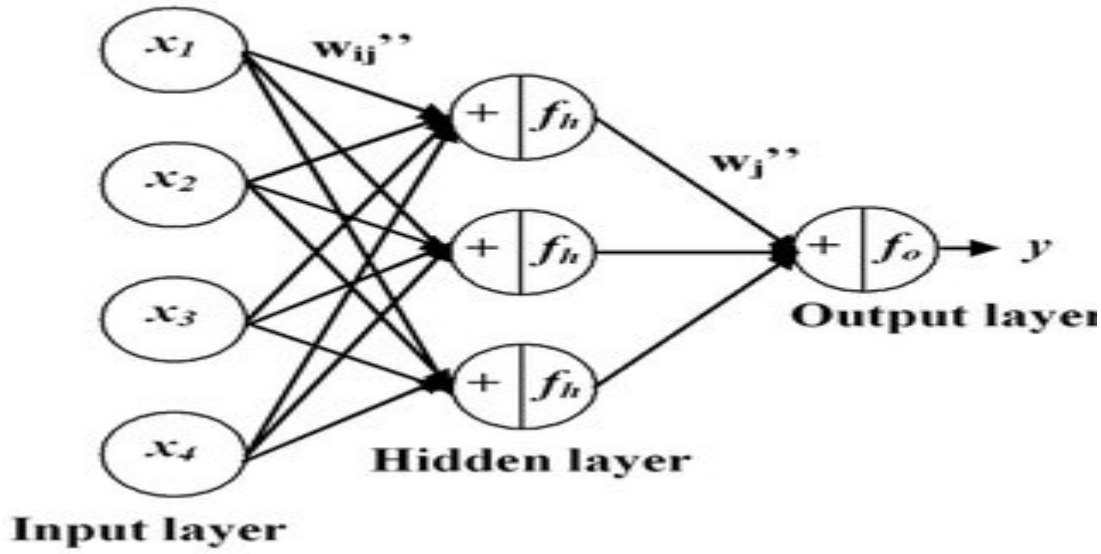


fig: **Forward pass in Neural Network**

The steps in the forward pass is as follows:

a) Getting the input:

In the input for the layer is the output of the previous layer or input for the input layer is training data.

In the figure above the input for the hidden layer is the output of the input layer, i.e., the $\mathbf{f(x_1*w_{ij} + x_2*w_{ij} + x_3*w_{ij} + x_4*w_{ij})}$ is the input for the first node in the hidden layer. Similarly, the input for the input layer is the training data ($\mathbf{x_1, x_2, x_3, x_4}$).

b) Applying the Weights:

In this step, we will apply the layer weights for the corresponding input nodes. It is multiplication of the input and the layer weight.

In the figure above, the weights applied for the input node1 is

$(x_1*w_{11} + x_1*w_{12} + x_1*w_{13})$

c) Activation Function:

Once we apply the weights to the inputs then we have to apply the activation function to it and here I am using the most popular **'Sigmoid function'** and when we apply this sigmoid function to $(x_1*w_{11} + x_1*w_{12} + x_1*w_{13})$ the function will return a value between 0 and 1. This function is since we are dealing with the Binary Classification that has a possible outcomes either 0 or 1. If we have to deal with the Multi-class classification problem then we will use other activation function called **'Softmax function'.**

```
def train(self, epochs):

    for e in range(epochs):
        # FORWARDPROPAGATION
        l1 = self.sigmoid(np.dot(self.X, self.w1))
        # in between hidden and output
        l2 = self.sigmoid(np.dot(l1, self.w2))
```

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

***fig*: Activation function: Sigmoid function**

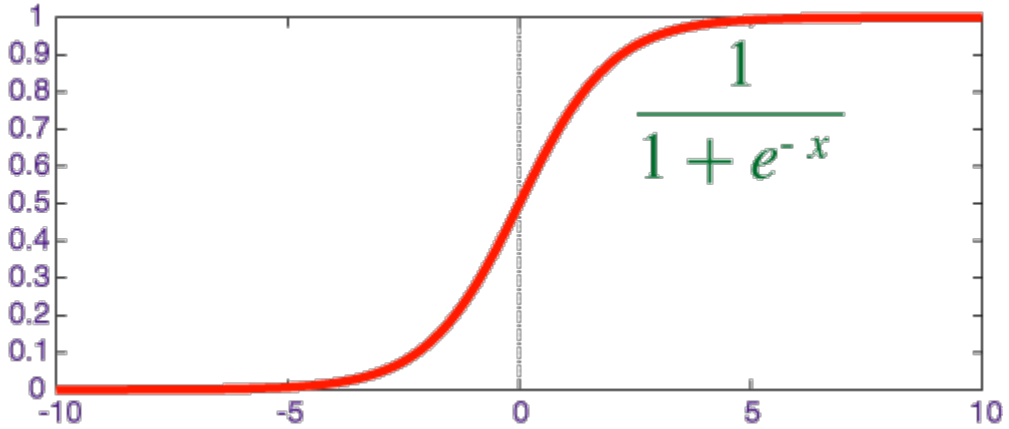


***fig*: Sigmoid function return values [0, 1]**

3. **Back Propagation:**

In this step, the network will actual learning the data and it predicts the result. We train the model using the gradient descent. Here out network will first check the how far our predicted results are from the true value. Then we will move in backward direction in our network and the find the error in each layer then according to that we will adjust weights in order to reduce the error (to improve the prediction).

The two major steps involved in this phase are:

a) Finding the error of the network:

We will calculate the difference between the true value and the predicted value.

```
# BACKPROPAGATION
# Network error (True value - Predicted value)

error = self.Y - l2
```

b) Error of each layer:

We know the error of the network and now we have to find the each for each layer in our network. So, here we find the error in layer using derivative of the current layer multiplied by the error of the previous layer.

```
# error for each of the layers

l2_delta = error * self.sigmoid_prime(l2)
l1_delta = l2_delta.dot(self.w2.T) * self.sigmoid_prime(l1)
```

error in the layer2 (hidden layer) = network error x derivative of the layer2

similarly,

error in the layer1 (input layer) = error of layer2 x weight of layer2 x derivative of the layer1

Here, the derivative of the current layer is calculated by using the sigmoidPrime function.

```
def sigmoid_prime(self, X):
    return X * (1 - X)
```

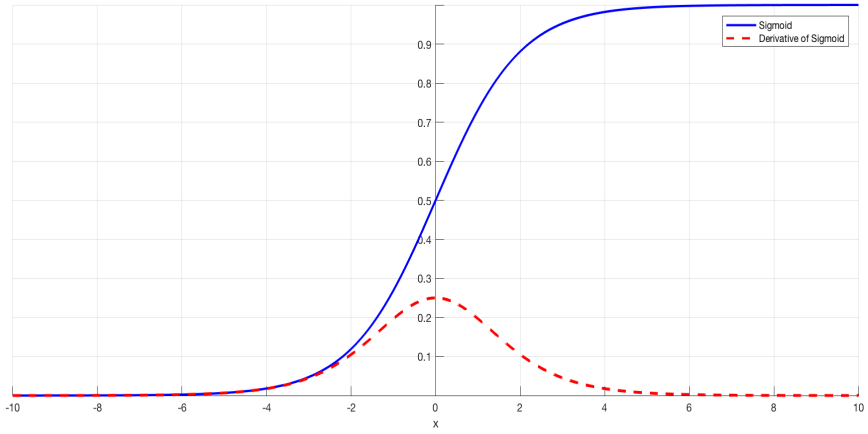


fig: **Sigmoid vs SigmoidPrime**

4. Update Weights:

In this we need to find the weight that need to be added to the layer weight to improve the prediction.

a) **Find weight that need to be added:**

To find the adjustment of weight that we need to add can be calculated by taking the dot product of previous and the error of the current layer and multiplied by learning rate.

b) Add the weights to the layer:

Add the adjustment weights to each layer weights.

```
self.w2 = np.add(self.w2, l1.T.dot(l2_delta) * self.learning_rate)
self.w1 = np.add(self.w1, self.X.T.dot(l1_delta) * self.learning_rate)
```

This conclude the training part of the neural network.

5. Testing & Evaluation:

Till now we spent most of the time in building a model and training the network to better predict the outcome. So, here we will test the model performance by running over the different sample that is not part of the training data also called a **testing set.**

This testing set is used to test check the model performance in generalizing the unseen data and the number of correct predictions made by comparing with the actual class labels.

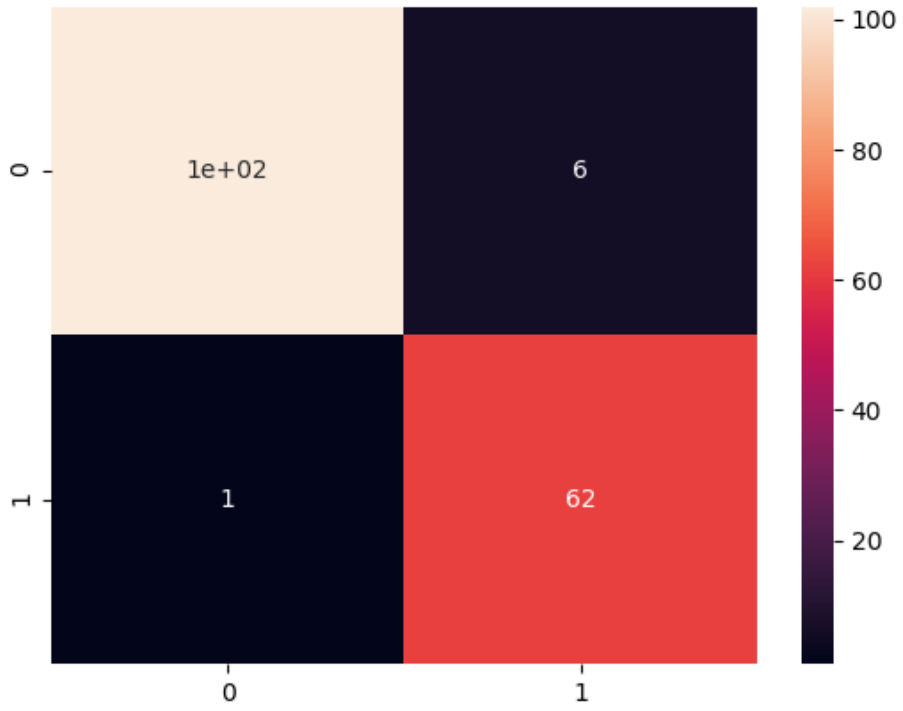
```
    def test(self):
        correct = 0
        pred_list = []
        l1 = self.sigmoid(np.dot(self.X_test, self.w1))
        l2 = self.sigmoid(np.dot(l1, self.w2))

        for i in range(len(l2)):
            if l2[i] >= 0.5:
                pred = 1
            else:
                pred = 0

            if pred == self.Y_test[i]:
                correct += 1

            pred_list.append(pred)

        print('Test Accuracy : ', ((correct / len(Y_test)) * 100), '%')

        cm = confusion_matrix(Y_test, pred_list)
        sns.heatmap(cm, annot=True)
        plt.savefig('h.png')
        plt.show()


nn = NeuralNetwork(X_train, Y_train, X_test, Y_test)
```

Results:

Performance Metrics:

```
Error :  0.0038812214102433634
Test Accuracy :  95.90643274853801 %
True Nagative 102
False Positive 6
False Negative 1
True Positive 62
Test Accuracy :  0.9590643274853801
Missclassification Rate :  0.04093567251461988
precision :  [0.99029126 0.91176471]
recall :  [0.94444444 0.98412698]
FScore :  [0.96682464 0.94656489]
Support :  [108  63]
Sensitivity or TPR :  0.9841269841269841
Specificity or TNR :  0.9444444444444444
False Positive Rate or Fallout :  0.05555555555555555
False Negative Rate :  0.015873015873015872
False Discovery Rate :  0.08823529411764706
```

Confusion Matrix:



Reference:

https://archive.ics.uci.edu/ml/datasets/Breast+Cancer+Wisconsin+%28Diagnostic%29

https://www.kaggle.com/uciml/breast-cancer-wisconsin-data/home

http://ftp.cs.wisc.edu/math-prog/cpo-dataset/machine-learn/cancer/WDBC/