Report

"Allows professionals to help their companies, but the nature of their work also enables them to advance in their career."

Diagnosis of Induction Motor Using Machine Learning Algorithms.

By Mr. Nadeem

Supervision: Professor Usman Ali
From Data Science

Academic year 2023

"Converting Knowledge into Practical Experience"

This Report related to Data Science Students

It’s completely Machine Learning based contents.

“Interesting in predictive analytics?”

Abstract

In Which First we pre-process our data then implement Machine Learning Models From Scratch we Implement These Models From Scratch(KNN, Logistic Regression for Binary class as well as Multi class From Scratch, Logistic Regression for Binary class as well as Multi class Use Built In Libraries, SVM for Binary class as well as Multi Class From Scratch, SVM for Binary class as well as Multi Class use Built In Libraries, Naïve Baye’s For Binary Class as well as Multi class From Scratch, Naïve Baye’s For Binary Class as well as Multi class Use Built In Libraries, and Finally compute Training loss, Testing loss and Validation loss For Naïve Baye’s Classifier). One Technique that we implement for Dimension Reduction These Technique are called as Principle Component Analysis(PCA).

Contents

1 Introduction:

This thesis is completely based on Machine Learning. It is relevant to data science students. In which we covered many Machine Learning models from scratch implementation. This is very useful for those students who start machine learning from beginner level and the basic goal of this Machine learning Course.. It is very easy and many explanation of some Machine Learning Algorithms. Here is we learn Data Pre-Processing in Automated way as well as another way that is not automated but it's easy to implement on small dataset. I can explain those data pre-processing that are give me to arrange data like this. (Data pre-processing: In the dataset, the total number of samples in each file are more than 100,000 and any block of 1000 samples can be used to label the corresponding healthy and unhealthy output. Create a new csv file for training and testing your model in which each row containing 1000 samples with assignment of the output label healthy/unhealthy).

2 Problem Statement:

Problem statement is we have dataset given in a folder in which we have says that first pre-processed the data and then implement Machine Learning Models in which(KNN, Logistic Regression for Binary class as well as Multi class, Naïve bayes for Binary class as well as Multiclass, Support Vector Machine(SVM) for Binary class as well as Multi class, and finally compute Validation loss, Testing loss and Training loss for Naïve Bayes Classifier.)

3 Data Pre-Processing:

In Which we give dataset. Dataset is completely based on Induction Motor. In dataset we have 8 Folders. First 6 Folders, files related to bearing fault. Other one folder related to rotor. And Another one is related to healthy motor. Previous 6 files related to Unhealthy motor and rotor are also unhealthy. In each folder we have many files but we collect just inner-100watt and outer -100watt files from each folder but in rotor we just collect 100-watt file and in healthy folder we collect just healthy file. We need just 14 files because throughout the thesis we play with just 14 class classification problem. So, collect 12 files from first 6 folders collect those files who have just named as inner-100watt and outer100-watt, next collect one file from rotor folder that are 100watt, Next collect from healthy folder in which we have 1 file named as healthy. Finally we collect 14 files and that files are csv files. So, implement 14 class classification problem on this dataset:

Here is two way to implement that task Efficiently First is Automated Way, and the Second one is Manual way.

a. Automated Way:

Here is code of Data-Pre-Processing:

We need just “Current-A” column.

```
if __name__ == '__main__':

    path = 'input/unhealthy_0.7_inner_bearing.csv'
    all_data_Frames = []
    file_number = 0.7

    for i in range(1):
        for j in range(6):
            file1_path, file2_path, output_file_path = update_paths(path, file_number)
            file_1, file_2 = read_files(file1_path, file2_path)
            do_the_work(file_1, file_2, file1_path, file2_path, output_file_path)
            file_number = round(file_number + .2, 1)

        file_1, file_2 = read_files()
        do_the_work(file_1, file_2)
        merge_and_write(all_data_Frames, 'Final_Data-Frame')
```

The first line checks if the script is run as the main program. If it is, then code will be executed.

Next few lines define some variables that will be used later on. The “path” variable holds the path to the input file, and the all_data_Frames list will be used to store the data frames generated from each input file. The file_number variable is set to 0.7, which is used later on to update the path to the next input file.

Then enters a loop that iterates 6 times. Each iteration processes one of the input files. The loop also updates the path to the next input file by incrementing file_number by 0.2 and calling the update_paths() function.

Here is update_paths() function:

```
def update_paths(folder_path, folder_number):
    folder_number = str(folder_number)
    location = folder_path.replace(folder_path[16 : 16+3], folder_number)
    return location, location.replace('inner', 'outer'), extract_label(location).replace('_inner', '')
```

This function takes in two arguments folder_path: a string representing the path of the folder where the files are located. It is basically the file path. folder_number: a float representing the current folder number.

Here are the steps performed by the function:

Convert folder_number to a string and store it in a new variable with the same name Update the location string by replacing the substring between the 16th and 18th characters (which represents the current folder number) with the updated folder_number(Here we Perform slicing of path).

Create a new string by replacing 'inner' in the location string with 'outer'

Call the extract_label function (That is shown below) on the location string to extract the label from the filename, and replace the '_inner' substring in the label with an empty string

Return a tuple with the updated paths and label.

```
def extract_label(file_path):
    return file_path[6:len(file_path) - 4]
```

The function takes a file path as input. It uses string slicing to extract the label of the file. Specifically, it starts at the 6th character (which assumes that the file path starts with "./input/"), and ends at the 4th character from the end (which assumes that the file extension is ".csv"). The extracted label is returned as a string.

Then call read_file() function:

```
def read_files(path_1 = 'input/healthy_BRB.csv', path_2 = 'input/unhealthy_BRB_rotor.csv'):
    return pd.DataFrame(pd.read_csv(path_1)[' Current-A']).values[:100000], pd.DataFrame(pd.read_csv(path_2)[' Current-A']).value
```

In which it take initially two paths and read just Current-A and First 1 Lac rows.

Here is rest of the code we get first 12 files Now Next:

Then call do_the_work() function:

```
def do_the_work(frame_1, frame_2, path_1 = 'input/healthy_BRB.csv', path_2 = 'input/unhealthy_BRB_rotor.csv', output_file = 'Heal
    reshaped_frame_1, reshaped_frame_2 = reshape(frame_1, frame_2)
    reshaped_frame_1, reshaped_frame_2 = do_labeling(reshaped_frame_1, reshaped_frame_2, path_1, path_2)
    all_data_Frames.append(reshaped_frame_1)
    all_data_Frames.append(reshaped_frame_2)
    merge_and_write([reshaped_frame_1, reshaped_frame_2], output_file)
```

After this the first six input files, the script reads in the seventh input file and calls the do_the_work() function on it.

In which call do_labeling() function:

```
def do_labeling(df_1, df_2, p_1, p_2):
    label_file_1 = extract_label(p_1)
    label_file_2 = extract_label(p_2)
    label_it(df_1, label_file_1)
    label_it(df_2, label_file_2)
    return df_1, df_2
```

Basically It takes Two DataFrames and two paths, On the basis of this it is assign label and return df_1 and df_2.

In do_the_work() function also call reshape():

Reshape() used for rearranging row into columns that is actually a task of pre-processing:

```
def reshape(df1, df2):

    col = range(1000)
    reshaped_df1 = pd.DataFrame(columns=col)
    reshaped_df2 = pd.DataFrame(columns=col)

    row_index = 0
    for x in range(0, len(df1), 1000):
        reshaped_df1 = pd.concat([reshaped_df1, pd.DataFrame(df1[x:x+1000].T, index=[row_index])])
        row_index = row_index + 1

    row_index = 0
    for x in range(0, len(df2), 1000):
        reshaped_df2 = pd.concat([reshaped_df2, pd.DataFrame(df2[x:x+1000].T, index=[row_index])])
        row_index = row_index + 1

    return reshaped_df1, reshaped_df2
```

It takes two Dataframe and Rearrange it and return these two dataframe after Reshaped.

Here is rest of code we get 2 files after rearranging.

And then update another these two files by add 2.

It iterate up to 6 time So, we get 12 files with Completely Arranged and Label Assigned.

Then, finally call read_file() and do_the_work() and merge_and_write() for another two files. It will give me two files with Complete Arranged and Label Assigned.

So, We have 14 files here. But in merge_and_write function we write all files into one file with respected labels.

Another way is:

Manual way:

Here is code:

```
import pandas as pd
import numpy as np
data = pd.read_csv('./input/0.7inner-100watt.csv').drop(['Time Stamp', ' Current-B', ' Current-C'], axis = 1)[:100000]
data.head()
data.shape
tranformed_df = pd.DataFrame(columns=range(1000))
for i in range(0, 100000, 1000):
    tranformed_df = tranformed_df.append(pd.DataFrame(np.array(data.iloc[i:i+1000].T), columns = range(1000)))
tranformed_df
tranformed_df.to_csv('./Output/0.7inner.csv')
```

First we read file and drop those columns who don't need then Create an empty Pandas data frame tranformed_df with columns ranging from 0 to 999 using the pd.DataFrame() constructor and passing columns=range(1000) as an argument.

Use a for loop to iterate over every 1000 rows of data using the range() function.

Within the loop, slice data to get the 1000 rows, transpose it using the T attribute, and convert it to a Pandas dataframe using the pd.DataFrame() constructor.

Append the resulting dataframe to tranformed_df using the append() method.

Reassign tranformed_df to the same variable name to overwrite the empty dataframe created earlier.

Print tranformed_df to check the data

After it convert it to File.

We have 14 Files this process apply on these 14 files 14 times and then concatenate it. By using this code:

```
import pandas as pd
import numpy as np
import os

# set the path to the directory containing the data files
data_path = './Output/'

# create an empty list to store the data frames
dfs = []

# loop through the data files and load them into data frames
for filename in os.listdir(data_path):
    if filename.endswith('.csv'):
        df = pd.read_csv(os.path.join(data_path, filename), header=None)
        dfs.append(df)

# concatenate the data frames into a single data frame
df = pd.concat(dfs)
df.to_csv('./Output/FinalDataFrame.csv')
```

And save into csv file …

For Binary class Classification we have all files unhealthy except healthy.csv file. So apply Label Unhealthy as well as Healthy. But

For Multiclass use Labels According to data.

That's a Pre-Processing Task.

4 Algorithms:

Machine Learning Algorithms Start From Here:

4.1 KNN:

Here is code:

```
import math
def euclidean_distance(x1, x2):
    distance = 0
    for j in range(len(x1)):
        distance += (x1[j] - x2[j])**2
    return math.sqrt(distance)
```

The next block of code defines a function called euclidean_distance that takes two vectors x1 and x2 as input, calculates the Euclidean distance between them, and returns the result. The Euclidean distance is the straight-line distance between two points in a multidimensional space. The function uses a loop to iterate over the elements of the two input vectors and calculates the square of the difference between each element. These squared differences are then summed up and square rooted to obtain the final Euclidean distance.

```
def predict(input_train,output_train , test_point, k_):
    distances = []
    for index in range(len(input_train)):
        dist = euclidean_distance(input_train.iloc[index,:], test_point)
        distances.append((output_train.iloc[index], dist))

    distances = sorted(distances, key=lambda x: x[1])
    neighbors = distances[:k_]

    classes = {}
    for neighbor in neighbors:
        label = neighbor[0]
        if label in classes:
            classes[label] += 1
        else:
            classes[label] = 1

    return max(classes, key=classes.get)
un_transformed_data = pd.read_csv('Output/FinalDataFrame.csv')
X, y = un_transformed_data.drop('label', axis = 1), un_transformed_data['label']
x_train, x_test, y_train, y_test = train_test_split(X, y, train_size=.8)
x_train.head()
y_train.head()
x_test.head()
y_test.head()
k = 3
prediction = [predict(x_train, y_train, x_test.iloc[i], k) for i in range(len(x_test))]
prediction
print('Model is ',accuracy_score(y_true = y_test, y_pred = prediction) * 100,'% accurate')
```

Here are the steps that the function takes:

It first initializes an empty list distances to store the distances between the test point and each training data point.

It then loops through each training data point using the range() function and calculates the Euclidean distance between the test point and that training data point using a helper function euclidean_distance().

The euclidean_distance() helper function calculates the Euclidean distance between two data points. In this case, it takes in two arguments: a Pandas Series representing one data point and another Pandas Series representing another data point. It calculates the Euclidean distance between these two points using the NumPy linalg.norm() function.

The calculated distances and their corresponding labels are then appended to the distances list.

The distances list is then sorted in ascending order based on the distances.

The neighbors list is initialized with the first k_ elements of the distances list, which correspond to the k-nearest neighbors to the test point.

The classes dictionary is initialized as an empty dictionary to store the counts of each class label among the k-nearest neighbors.

The function then loops through each neighbor in the neighbors list and updates the count of its class label in the classes dictionary.

Finally, the predicted class label is the class label with the highest count in the classes dictionary. This is returned as the output of the function.

4.1.1 Classification Measures From Scratch:

Here is code

```
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import confusion_matrix
from sklearn.model_selection import train_test_split

df = pd.read_csv('output/Principle_Components_labeled.csv')
# Create a KNN classifier with k=3
knn = KNeighborsClassifier(n_neighbors=3)
X = df.iloc[:, :-1].values
y = df.iloc[:, -1].values
x_train, x_test, y_train, y_test = train_test_split(X, y, train_size=.8)

# Train the classifier on your training data
knn.fit(x_train, y_train)

# Predict the labels of your test data
y_pred = knn.predict(x_test)

# Compute the confusion matrix
cm = confusion_matrix(y_test, y_pred)

# Extract the values from the confusion matrix
TP = cm[1, 1]
TN = cm[0, 0]
FP = cm[0, 1]
FN = cm[1, 0]

# Compute the classification measures
accuracy = (TP + TN) / (TP + TN + FP + FN)
recall = TP / (TP + FN)
precision = TP / (TP + FP)
specificity = TN / (TN + FP)
f1_score = 2 * (precision * recall) / (precision + recall)

# Print the classification measures
print("Accuracy:", accuracy)
print("Recall:", recall)
print("Precision:", precision)
print("Specificity:", specificity)
print("F1 Score:", f1_score)
```

Import necessary libraries: Import KNeighborsClassifier from sklearn.neighbors, confusion_matrix from sklearn.metrics, and train_test_split from sklearn.model_selection.

Load the dataset: Load the dataset from the CSV file using the read_csv() function from pandas library.

Split the dataset: Separate the features and target variable from the dataset using iloc and then split the data into training and testing sets using the train_test_split() function.

Create a KNN classifier: Create a KNN classifier with n_neighbors set to 3 using the KNeighborsClassifier() function.

Train the classifier: Train the classifier on the training data using the fit() method of the classifier.

Predict the labels of the test data: Use the predict() method of the classifier to predict the labels of the test data.

Compute the confusion matrix: Compute the confusion matrix using the confusion_matrix() function from sklearn.metrics.

Extract values from the confusion matrix: Extract the true positive, true negative, false positive, and false negative values from the confusion matrix.

Compute the classification measures: Compute the accuracy, recall, precision, specificity, and F1 score using the formulas for each measure. Formula write in Code.

Print the classification measures: Print the accuracy, recall, precision, specificity, and F1 score values.

4.2 PCA:

Here is code:

```
from sklearn.decomposition import PCA
pca = PCA()
df = pd.read_csv('./Output/FinalDataFrame.csv')
scaler_1 = StandardScaler()
scaled_df = pd.DataFrame(scaler_1.fit_transform(df.drop('label', axis=1)))

pca.fit(scaled_df)
per_var = np.round(pca.explained_variance_ratio_ * 100, decimals=0)
per_var = per_var[per_var > 0]
per_var

principle_components = pca.transform(scaled_df)
principle_components

input_features = df.drop('label', axis=1)
input_features.head()
scaler = StandardScaler()
scaler.fit(input_features)
scaled_input_features = pd.DataFrame(scaler.transform(input_features))
scaled_input_features.head()
cov_matrix = scaled_input_features.cov()
cov_matrix
eigen_vector_values = np.linalg.eig(cov_matrix)
eigen_vector_values
eigen_vector_values[0]
eigen_vector_values[1]
order_of_index = np.argsort(eigen_vector_values[0])[::-1]
order_of_index
# Ordering eigen vectors in descending order based on eigen values
principle_components_manually = pd.DataFrame(eigen_vector_values[1][:, order_of_index])
principle_components_manually.head()
total_variables = len(principle_components_manually)
total_variables
var_per_vect = np.array([round(x / total_variables * 100) for x in eigen_vector_values[0]])
var_per_vect = var_per_vect[var_per_vect > 0]
var_per_vect
labels = ['PC'+str(x) for x in range(1, len(var_per_vect) + 1)]
labels
plt.style.use('ggplot')
plt.bar(labels, per_var)
plt.xlabel('Principle Components')
plt.ylabel('Explained Variance')
plt.show()
top_principle_components = pd.DataFrame(principle_components_manually.iloc[:, :len(var_per_vect)])
top_principle_components.head()
low_dimensional_data = pd.DataFrame(np.dot(scaled_input_features, top_principle_components))
low_dimensional_data.head()
low_dimensional_data.columns.values[:len(labels)] = labels
low_dimensional_data['label'] = df['label']
low_dimensional_data.head()
```

The necessary libraries are imported: StandardScaler and PCA from sklearn.preprocessing and sklearn.decomposition, respectively.

PCA() is initialized.

The dataset is read into a Pandas DataFrame from the CSV file.

StandardScaler() is initialized.

The dataset is scaled using scaler_1.fit_transform(), which subtracts the mean and divides by the standard deviation of each column in the dataset.

pca.fit() is called on the scaled dataset to fit the PCA model.

The explained_variance_ratio_ attribute of the fitted PCA model is used to calculate the percentage of variance explained by each principal component. The results are rounded to 0 decimal places and stored in the per_var variable.

Principal component scores are obtained using pca.transform(), which applies the PCA transformation to the scaled dataset.

The input features (all columns except the target variable) are stored in the input_features variable.

StandardScaler() is initialized again.

The input features are scaled using scaler.fit_transform().

The covariance matrix of the scaled input features is calculated using scaled_input_features.cov().

numpy.linalg.eig() is called on the covariance matrix to obtain the eigenvalues and eigenvectors.

The eigenvalues are stored in eigen_vector_values[0].

The eigenvectors are stored in eigen_vector_values[1].

The eigenvalues are sorted in descending order using numpy.argsort() and stored in the order_of_index variable.

The eigenvectors are ordered based on the sorted eigenvalues using pd.DataFrame() and stored in the principle_components_manually variable.

The variance explained by each principal component is calculated by dividing the eigenvalues by the total number of variables and multiplying by 100. The results are rounded to 0 decimal places and stored in the var_per_vect variable.

Labels for the principal components are generated using a list comprehension and stored in the labels variable.

A bar chart is plotted using plt.bar() to show the percentage of variance explained by each principal component.

The top principal components are stored in top_principle_components using principle_components_manually.iloc[].

The low-dimensional data is obtained by taking the dot product of the scaled input features and the top principal components using np.dot() and stored in low_dimensional_data.

The column names of low_dimensional_data are updated to the principal component labels using low_dimensional_data.columns.values[].

The target variable is added to low_dimensional_data using df['label'].

The final output is displayed using low_dimensional_data.head().

4.3 Logistic Regression:

Logistic Regression is a statistical algorithm used for binary classification problems, where the goal is to predict a binary outcome (e.g., yes/no, true/false) based on one or more input features. It models the probability of the binary outcome using a logistic function and learns the optimal parameters of the model using maximum likelihood estimation. In practice, logistic regression can be extended to handle multi-class classification problems using techniques such as one-vs-all or softmax regression.

4.3.1 Logistic for Binary Class From Scratch:

Here is code:

```
import numpy as np

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

class LogisticRegression():

    def __init__(self, lr=0.001, n_iters=1000):
        self.lr = lr
        self.n_iters = n_iters
        self.weights = None
        self.bias = None
```

The first step is to import the NumPy library and define a sigmoid function. The sigmoid function is used to squash any input value to a range between 0 and 1, which is useful in logistic regression for calculating probabilities.

Next, LogisticRegression class is defined with a constructor that initializes various parameters. lr and n_iters are the learning rate and the number of iterations used in gradient descent. weights and bias are the model parameters that will be learned during training.

```
    def fit(self, X, y):
        n_samples, n_features = X.shape
        self.weights = np.zeros(n_features)
        self.bias = 0

        for _ in range(self.n_iters):
            linear_pred = np.dot(X, self.weights) + self.bias
            predictions = sigmoid(linear_pred)
            y = y.astype('int')

            dw = (1/n_samples) * np.dot(X.T, (predictions - y))
            db = (1/n_samples) * np.sum(predictions-y)

            self.weights = self.weights - self.lr*dw
            self.bias = self.bias - self.lr*db
```

The fit method is used to train the logistic regression model. It takes two arguments X and y, where X is the input data (a NumPy array of shape (n_samples, n_features)), and y is the corresponding labels (a NumPy array of shape (n_samples, 1)). n_samples and n_features are the number of samples and features in the input data, respectively. The weights and bias are initialized to zero.

The fit method then trains the model using gradient descent. In each iteration, the linear predictions (linear_pred) are computed by taking the dot product of the input data X with the weights and adding the bias. Then, the sigmoid function is applied to the linear predictions to obtain the final predictions (predictions).

The cost function used in logistic regression is the binary cross-entropy loss, which can be minimized using gradient descent. The gradients of the cost function with respect to the weights and bias are computed and used to update the model parameters (self.weights and self.bias) using the learning rate (self.lr).

```
    def predict(self, X):
        linear_pred = np.dot(X, self.weights) + self.bias
        y_pred = sigmoid(linear_pred)
        class_pred = [0 if y<=0.5 else 1 for y in y_pred]
        return class_pred
```

Finally, the predict method is used to predict the labels for new input data. The linear predictions (linear_pred) are computed using the learned weights and bias. The sigmoid function is applied to the linear predictions to obtain the predicted probabilities (y_pred). Then, the predicted class is assigned based on a threshold of 0.5. If the predicted probability is less than or equal to 0.5, the predicted class is 0, otherwise, it is 1.

Now test our class:

```
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import LabelEncoder
from sklearn.metrics import accuracy_score

# Load data from CSV file
df = pd.read_csv('./Output/FinalDataFrameBinaryClass.csv')

# Split data into features and target variable
X = df.drop(columns=['label'])
y = df['label']

# Convert target variable to numeric values
le = LabelEncoder()
y = le.fit_transform(y)

# Split data into training and test sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Create and fit logistic regression model
clf = LogisticRegression(lr=0.01)
clf.fit(X_train, y_train)

# Predict test set labels and evaluate model accuracy
y_pred = clf.predict(X_test)
acc = accuracy_score(y_pred, y_test)
print(acc)
```

```
0.9357142857142857
```

Import necessary libraries:

a. pandas: data manipulation and analysis

b. train_test_split: function for splitting dataset into training and test sets

c. LabelEncoder: encoder to convert categorical variables to numeric values

d. accuracy_score: function to calculate the accuracy score of the model

Load data from a CSV file using pandas read_csv function.

Split data into features and target variable. The features are assigned to X and the target variable is assigned to y.

Convert target variable to numeric values using LabelEncoder.

Split data into training and test sets using the train_test_split function. The test set size is set to 0.2, meaning that 20% of the data will be used for testing, while the remaining 80% will be used for training. The random_state parameter is set to 42 to ensure reproducibility.

Create an instance of the LogisticRegression class with a learning rate (lr) of 0.01.

Fit the logistic regression model to the training data using the fit method of the LogisticRegression class.

Predict the labels of the test set using the predict method of the LogisticRegression class.

Calculate the accuracy score of the model by comparing the predicted labels with the true labels using the accuracy_score function.Print the accuracy score of the model.

4.3.2 Logistic For Binary Class Using Built in Libraries:

Here is code :

```python
# Import necessary libraries
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score, recall_score, precision_score, f1_score, confusion_matrix


# Load dataset and split into features and labels
X = df.drop('label', axis=1)
y = df['label']

# Split dataset into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
y_train = y_train.replace({"healthy": 0, "unhealthy": 1})
y_test = y_test.replace({"healthy": 0, "unhealthy": 1})
# Create logistic regression model and train on training set
model = LogisticRegression()
model.fit(X_train, y_train)

# Make predictions on testing set
y_pred = model.predict(X_test)

# Compute classification measures
accuracy = accuracy_score(y_test, y_pred)
recall = recall_score(y_test, y_pred)
precision = precision_score(y_test, y_pred)
f1 = f1_score(y_test, y_pred)
confusion = confusion_matrix(y_test, y_pred)

# Print results
print(f"Accuracy: {accuracy}")
print(f"Recall: {recall}")
print(f"Precision: {precision}")
print(f"F1 Score: {f1}")
print(f"Confusion Matrix: \n{confusion}")
```

Import necessary libraries:

a. LogisticRegression: a machine learning model used for binary classification

b. train_test_split: function for splitting dataset into training and test sets

c. accuracy_score: function to calculate the accuracy score of the model

d. recall_score: function to calculate the recall score of the model

e. precision_score: function to calculate the precision score of the model

f. f1_score: function to calculate the F1 score of the model

g. confusion_matrix: function to generate the confusion matrix of the model

Load the dataset and split it into features and labels. The features are assigned to X and the labels are assigned to y.

Split the dataset into training and testing sets using the train_test_split function. The test set size is set to 0.2, meaning that 20% of the data will be used for testing, while the remaining 80% will be used for training. The random_state parameter is set to 42 to ensure reproducibility.

Replace the string labels 'healthy' and 'unhealthy' with numerical labels 0 and 1 respectively for both training and testing labels.

Create an instance of the LogisticRegression class.

Train the logistic regression model on the training set using the fit method of the LogisticRegression class.

Predict the labels of the test set using the predict method of the LogisticRegression class.

Compute classification measures including accuracy, recall, precision, F1 score, and confusion matrix using their respective functions.

Print the accuracy, recall, precision, F1 score, and confusion matrix of the model.

4.3.3 Logistic For Multi class From Scratch:

Here is code:

```
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split


def softmax(z):

    e_z = np.exp(z - np.max(z, axis=0, keepdims=False))
    return e_z / np.sum(e_z, axis=0, keepdims=False)


class LogisticRegression():

    def __init__(self, lr=0.001, n_iters=1000):
        self.lr = lr
        self.n_iters = n_iters
        self.weights = None
        self.bias = None
```

softmax function takes in an array z and returns the softmax function applied to z, which is the exponential of each element of z divided by the sum of exponentials of all elements of z.

LogisticRegression class is defined which has two methods, __init__, fit, and predict.

__init__ method initializes the instance variables lr (learning rate), n_iters (number of iterations), weights, and bias to None or zero.

```
def fit(self, X, y):
    n_samples, n_features = X.shape
    self.weights = np.zeros((n_features, len(np.unique(y))))
    self.bias = np.zeros(len(np.unique(y)))

    for i, c in enumerate(np.unique(y)):
        y_one_vs_all = np.where(y == c, 1, 0)
        for _ in range(self.n_iters):
            linear_pred = np.dot(X, self.weights[:,i]) + self.bias[i]
            predictions = softmax(linear_pred)

            dw = (1/n_samples) * np.dot(X.T, (predictions - y_one_vs_all))
            db = (1/n_samples) * np.sum(predictions - y_one_vs_all)

            self.weights[:,i] = self.weights[:,i] - self.lr * dw
            self.bias[i] = self.bias[i] - self.lr * db
```

fit method takes in training data X and target variable y, and updates weights and bias iteratively using gradient descent for each class. The method first calculates the number of samples and number of features in X, initializes weights and bias for each class with zero, and then loops through each unique class in y. Within each class loop, the method calculates the one-vs-all target variable, which assigns a value of 1 to the samples that belong to the current class and 0 to the samples that do not belong to the current class. The method then updates weights and bias for the current class using the gradient of the loss function with respect to weights and bias. The gradient is calculated as the dot product of the transpose of X and the difference between the predicted probabilities and the one-vs-all target variable, divided by the number of samples. Finally, the method updates weights and bias for the current class using the learning rate and the gradient.

```
def predict(self, X):
    linear_pred = np.dot(X, self.weights) + self.bias
    y_pred = softmax(linear_pred)
    class_pred = np.argmax(y_pred, axis=1)
    return class_pred

def accuracy(y_pred, y_test):
    return np.sum(y_pred==y_test)/len(y_test)
```

Predict() method takes in test data X and returns the predicted class labels for each sample in X. The method first calculates the linear prediction for each class using weights and bias, and then applies the softmax function to the linear prediction to obtain the predicted probabilities for each class. The method then returns the class with the highest probability as the predicted class label for each sample.

Accuracy() function takes in predicted labels y_pred and true labels y_test, and returns the accuracy score of the predicted labels. The function first calculates the number of correctly predicted labels by comparing y_pred and y_test, and then divides the number of correctly predicted labels by the total number of samples in y_test to obtain the accuracy score.

Now Test Our Class:

```
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import LabelEncoder
from sklearn.metrics import accuracy_score

# Load data from CSV file
df = pd.read_csv('./Output/FinalDataFrame.csv')

# Split data into features and target variable
X = df.drop(columns=['label'])
y = df['label']

# Convert target variable to numeric values
le = LabelEncoder()
y = le.fit_transform(y)

# Split data into training and test sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Create and fit logistic regression model
clf = LogisticRegression(lr=0.01)
clf.fit(X_train, y_train)

# Predict test set labels and evaluate model accuracy
y_pred = clf.predict(X_test)
acc = accuracy_score(y_pred, y_test)
print(acc)
```

First, we import the necessary libraries: pandas for data manipulation, train_test_split for splitting the data into training and testing sets, LabelEncoder for converting the target variable to numeric values, and accuracy_score for evaluating the model's performance.

We load the data from a CSV file using the read_csv() function from pandas and store it in the variable df.

We split the data into features (independent variables) and target variable (dependent variable) and store them in the variables X and y, respectively. We drop the 'label' column from X as it is the target variable.

We use LabelEncoder to convert the target variable y to numeric values. This is done because most machine learning algorithms cannot work with categorical data directly.

We split the data into training and test sets using the train_test_split() function. We use 20% of the data for testing and set the random_state to 42 to ensure reproducibility.

We create an instance of the LogisticRegression class and set the learning rate to 0.01.

We fit the logistic regression model to the training data using the fit() method of the LogisticRegression class.

We use the predict() method of the LogisticRegression class to predict the labels for the test set.

We use the accuracy_score() function to compute the accuracy of the model on the test set by comparing the predicted labels to the actual labels.

We print the accuracy score.

4.3.4 Logistic For Multic Class Use Built In Libraries:

Here is code

```
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score, confusion_matrix

# Load the dataset
data = pd.read_csv('./Output/FinalDataFrame.csv')
X = df.drop("label", axis=1)
y = df["label"]
# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Train the logistic regression model
model = LogisticRegression(multi_class='multinomial', solver='lbfgs', max_iter=1000)
model.fit(X_train, y_train)

# Make predictions on the test set
y_pred = model.predict(X_test)

# Compute evaluation metrics
accuracy = accuracy_score(y_test, y_pred)
precision = precision_score(y_test, y_pred, average='weighted')
recall = recall_score(y_test, y_pred, average='weighted')
f1 = f1_score(y_test, y_pred, average='weighted')
conf_matrix = confusion_matrix(y_test, y_pred)

# Print the evaluation metrics
print("Accuracy:", accuracy)
print("Precision:", precision)
print("Recall:", recall)
print("F1 score:", f1)
print("Confusion matrix:\n", conf_matrix)
```

First, necessary libraries are imported, including NumPy, scikit-learn's train_test_split, LogisticRegression, and evaluation metrics such as accuracy_score, precision_score, recall_score, f1_score, and confusion_matrix.

The dataset is loaded from a CSV file using pandas' read_csv method, and features and target variables are separated.

The target variable is encoded using scikit-learn's LabelEncoder to convert string labels into numeric values.

The data is split into training and testing sets using train_test_split with a test size of 20% and a random state of 42.

A logistic regression model is initialized with the multi_class parameter set to "multinomial" for multiclass classification, solver set to "lbfgs" for optimization, and max_iter set to 1000.

The model is trained on the training set using the fit method of the logistic regression object.

Predictions are made on the test set using the predict method of the logistic regression object.

Evaluation metrics are computed using the corresponding scikit-learn functions. accuracy_score calculates the proportion of correct predictions, precision_score calculates the proportion of true positives among all positive predictions, recall_score calculates the proportion of true positives among all actual positives, and f1_score calculates the harmonic mean of precision and recall. confusion_matrix returns a matrix of true/false positive/negative predictions for each class.

The evaluation metrics are printed to the console using print.

4.4 Naïve Bayes:

Naive Bayes is a classification algorithm that is based on Bayes' theorem, which describes the probability of an event based on prior knowledge of conditions that might be related to the event. Naive Bayes assumes that all features in a dataset are independent of each other, and

calculates the conditional probability of each class given the values of the features. It works well with high-dimensional data and can be trained quickly on large datasets. Naive Bayes is commonly used in text classification, spam filtering, and recommendation systems.

4.4.1 Naive Baye's Binary Class From Scratch:

First, the required libraries are imported:

"NumPy " and "Pandas" are used for data manipulation and calculations, while train_test_split from sklearn.model_selection is used for splitting the data into training and test sets.

```python
class NaiveBayes:

    def fit(self, X, y):
        n_samples, n_features = X.shape
        self._classes = np.unique(y)
        n_classes = len(self._classes)

        # calculate mean, var, and prior for each class
        self._mean = np.zeros((n_classes, n_features), dtype=np.float64)
        self._var = np.zeros((n_classes, n_features), dtype=np.float64)
        self._priors = np.zeros(n_classes, dtype=np.float64)

        for idx, c in enumerate(self._classes):
            X_c = X[y == c]
            self._mean[idx, :] = X_c.mean(axis=0)
            self._var[idx, :] = X_c.var(axis=0)
            self._priors[idx] = X_c.shape[0] / float(n_samples)


    def predict(self, X):
        y_pred = [self._predict(x) for x in X]
        return np.array(y_pred)

    def _predict(self, x):
        posteriors = []

        # calculate posterior probability for each class
        for idx, c in enumerate(self._classes):
            prior = np.log(self._priors[idx])
            posterior = np.sum(np.log(self._pdf(idx, x)))
            posterior = posterior + prior
            posteriors.append(posterior)

        # return class with the highest posterior
        return self._classes[np.argmax(posteriors)]

    def _pdf(self, class_idx, x):
        mean = self._mean[class_idx]
        var = self._var[class_idx]
        numerator = np.exp(-((x - mean) ** 2) / (2 * var))
        denominator = np.sqrt(2 * np.pi * var)
        return numerator / denominator
```

The NaiveBayes class has three methods:

Fit(): This method is used to train the Naive Bayes model. It takes in two arguments: X and y. X is a numpy array containing the features of the dataset, and y is a numpy array containing the corresponding labels.

a. First, the method determines the number of samples and features in the dataset by using the shape attribute of the numpy array X.

b. Next, the unique classes in the dataset are determined using the unique() method of numpy, and the number of unique classes is stored in a variable called n_classes.

c. Then, the mean, variance, and prior probability of each class are calculated and stored in self._mean, self._var, and self._priors respectively. These values are calculated by iterating over each class, selecting only the samples with that class, and calculating the mean, variance, and prior probability.

d. The self._mean, self._var, and self._priors are instance variables of the NaiveBayes class and are used to make predictions in the predict() method.

Predict(): This method is used to predict the labels of the test data. It takes in a single argument X, which is a numpy array containing the features of the test dataset.

a. First, the method calls the _predict() method for each sample in the test dataset and stores the predicted label in a list.

b. Finally, the method returns the list of predicted labels as a numpy array.

_predict(): This method is used by the predict() method to predict the label of a single sample. It takes in two arguments: class_idx and x. class_idx is the index of the class being considered, and x is a numpy array containing the features of the sample.

a. First, the method calculates the prior probability of the class by taking the logarithm of the prior probability stored in self._priors.

b. Next, the method calculates the likelihood of the features given the class by taking the logarithm of the probability density function (_pdf()) of the features for the given class.

c. Finally, the method returns the sum of the logarithm of the prior probability and the logarithm of the likelihood, which gives the posterior probability of the class..

_pdf():This method is used by the _predict() method to calculate the probability density function of the features for the given class. It takes in two arguments: class_idx and x. class_idx is the index of the class being considered, and x is a numpy array containing the features of the sample.

a. First, the method extracts the mean and variance of the features for the given class from the instance variables self._mean and self._var.

b. Next, the method calculates the numerator and denominator of the probability density function.

c. Finally, the method returns the ratio of the numerator and denominator, which gives the probability density function.

```
def accuracy(y_true, y_pred):
    accuracy = np.sum(y_true == y_pred) / len(y_true)
    return accuracy
```

This function is used to calculate the accuracy of the model. It takes in two arguments: y_true and y_pred. y_true is a NumPy array containing the true labels of the test dataset, and y_pred is a NumPy array containing the predicted labels.

```
# Testing
if __name__ == "__main__":



    # Load CSV file
    df = pd.read_csv('./Output/FinalDataFrameBinaryClass.csv')

    # Split into features and target variable
    X = df.iloc[:, :-1].values
    y = df.iloc[:, -1].values

    # Split into train and test sets
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=123)
```

Load the dataset:

The dataset is loaded from a CSV file using the pandas library. The dataset is split into features (X) and target variable (y).

Split the dataset into training and testing sets

The dataset is split into training and testing sets using the train_test_split method from the sklearn library. 20% of the dataset is used for testing.

```
nb = NaiveBayes()
nb.fit(X_train, y_train)
predictions = nb.predict(X_test)

print("Naive Bayes classification accuracy", accuracy(y_test, predictions))
```

```
Naive Bayes classification accuracy 0.9107142857142857
```

Fit the Naive Bayes model. The NaiveBayes model is fit to the training data using the fit method.

Make predictions on the test set. The NaiveBayes model is used to make predictions on the test set using the predict method.

Calculate the accuracy of the model. The accuracy of the NaiveBayes model is calculated using the accuracy method.

Finally Print Accuracy.

4.4.2 Naïve Baye's Use Built in Libraries:

First, the required libraries are imported:

Naive Bayes For Binary Class use Built in library

```
In [2]: import pandas as pd
        from sklearn.naive_bayes import GaussianNB
        from sklearn.metrics import accuracy_score
```

"Pandas" for reading the dataset, "sklearn.naive_bayes" for the Gaussian Naive Bayes classifier, and "sklearn.metrics" for calculating the accuracy score.

```
df = pd.read_csv('./Output/FinalDataFrameBinaryClass.csv')
df.head()

X_train, X_test, y_train, y_test = train_test_split(df.drop('label', axis=1), df['label'], test_size=0.2, random_state=42)
```

The dataset is then read using pandas.read_csv() method and stored in a DataFrame named df. df.head() is used to display the first few rows of the DataFrame.

The data is split into training and testing sets using the train_test_split() method from the sklearn.model_selection module. The drop() method is used to drop the target variable (label) from the feature set.

```
clf = GaussianNB()
clf.fit(X_train, y_train)
y_pred = clf.predict(X_test)
```

An instance of the Gaussian Naive Bayes classifier is created using GaussianNB().
The fit() method is called on the classifier instance using the training feature set and target variable to train the model.
The predict() method is called on the classifier instance using the test feature set to make predictions on the test data.

```
accuracy = accuracy_score(y_test, y_pred)
accuracy
```

```
Out[2]: 0.8785714285714286
```

The accuracy_score() method is used to compare the predicted labels with the actual labels of the test data and compute the accuracy score. The accuracy score is printed as the final output.

4.4.3 Naïve Baye's for Multi Class From Scratch:

Required libraries are imported:

Naive baye's From Scratch For multi classification

```
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score
```

"Pandas" for reading the dataset, "NumPy" for data manipulation, "sklearn.model_selection" for splitting the data into training and testing sets, and "sklearn.metrics" for calculating the accuracy score.

```
class NaiveBayes:

    def fit(self, X, y):
        n_samples, n_features = X.shape
        self._classes = np.unique(y)
        n_classes = len(self._classes)

        # calculate mean, var, and prior for each class
        self._mean = np.zeros((n_classes, n_features), dtype=np.float64)
        self._var = np.zeros((n_classes, n_features), dtype=np.float64)
        self._priors = np.zeros(n_classes, dtype=np.float64)

        for idx, c in enumerate(self._classes):
            X_c = X[y == c]
            self._mean[idx, :] = X_c.mean(axis=0)
            self._var[idx, :] = X_c.var(axis=0)
            self._priors[idx] = X_c.shape[0] / float(n_samples)


    def predict(self, X):
        y_pred = [self._predict(x) for x in X]
        return np.array(y_pred)
```

fit(self, X, y): This method takes in the training data X and target variable y and trains the Naive Bayes classifier. It calculates the mean, variance, and prior probability for each class in the target variable. Here is how it works:

```
n_samples, n_features = X.shape
self._classes = np.unique(y)
n_classes = len(self._classes)
```

First, it calculates the number of samples and number of features in the training data.It then finds the unique classes in the target variable y. Next, it calculates the number of classes in the target variable.

```
# calculate mean, var, and prior for each class
self._mean = np.zeros((n_classes, n_features), dtype=np.float64)
self._var = np.zeros((n_classes, n_features), dtype=np.float64)
self._priors = np.zeros(n_classes, dtype=np.float64)
```

The method initializes arrays to store the mean, variance, and prior probability for each class. For each class in the target variable, the method calculates the mean, variance, and prior probability.

```
for idx, c in enumerate(self._classes):
    X_c = X[y == c]
    self._mean[idx, :] = X_c.mean(axis=0)
    self._var[idx, :] = X_c.var(axis=0)
    self._priors[idx] = X_c.shape[0] / float(n_samples)
```

The variable X_c contains all the rows in X where y equals the current class c.

self._mean[idx, :] is assigned the mean of X_c along each feature.

self._var[idx, :] is assigned the variance of X_c along each feature.

self._priors[idx] is assigned the prior probability of class c, which is the proportion of samples in X that belong to class c.

```
y_pred = [self._predict(x) for x in X]
return np.array(y_pred)
```

It creates an empty list y_pred to store the predicted labels. For each sample in X, it calls the _predict method with the sample as input and appends the predicted class label to y_pred.

```
def _predict(self, x):
    posteriors = []

    # calculate posterior probability for each class
    for idx, c in enumerate(self._classes):
        prior = np.log(self._priors[idx])
        posterior = np.sum(np.log(self._pdf(idx, x)))
        posterior = posterior + prior
        posteriors.append(posterior)

    # return class with the highest posterior
    return self._classes[np.argmax(posteriors)]

def _pdf(self, class_idx, x):
    mean = self._mean[class_idx]
    var = self._var[class_idx]
    numerator = np.exp(-((x - mean) ** 2) / (2 * var))
    denominator = np.sqrt(2 * np.pi * var)
    return numerator / denominator
```

_predict(self, x): This method takes in a single sample x and returns the predicted class label. It calculates the posterior probability of the sample belonging to each class and returns the class with the highest probability. Here is how it works: It initializes an empty list posteriors to store the posterior probability of the sample belonging to each class. For each class in the target variable, it calculates the prior probability of the class, the likelihood of the sample given the class, and the posterior probability of the sample belonging to the class. posteriors = []: Create an empty list to store the posterior probability of each class.

for idx, c in enumerate(self._classes): Loop through each class and its corresponding index.

prior = np.log(self._priors[idx]): Calculate the logarithm of the prior probability for the current class.

posterior = np.sum(np.log(self._pdf(idx, x))): Calculate the sum of the logarithms of the conditional probabilities for each feature, given the current class. The _pdf method calculates the conditional probability density function (PDF) for a given class and feature vector.

posterior = posterior + prior: Add the logarithm of the prior probability to the sum of the logarithms of the conditional probabilities to get the logarithm of the posterior probability.

posteriors.append(posterior): Append the logarithm of the posterior probability to the posteriors list.

return self._classes[np.argmax(posteriors)]: Return the class label with the highest posterior probability. This is calculated by finding the index of the highest value in the posteriors list using the np.argmax() function, and then using that index to access the corresponding class label from the _classes attribute.

_pdf() method of the NaiveBayes class, which calculates the probability density function (PDF) for a given class and feature vector.

mean = self._mean[class_idx]: Get the mean of the feature values for the current class from the _mean attribute.

var = self._var[class_idx]: Get the variance of the feature values for the current class from the _var attribute.

numerator = np.exp(-((x - mean) ** 2) / (2 * var)): Calculate the numerator of the PDF using the Gaussian probability density function formula. This formula calculates the probability density of a given feature value x given the mean and variance of the feature values for the current class.

denominator = np.sqrt(2 * np.pi * var): Calculate the denominator of the PDF using the Gaussian probability density function formula. This formula calculates the standard deviation of the feature values for the current class.

return numerator / denominator: Return the value of the PDF for the given feature vector and current class. This is calculated by dividing the numerator by the denominator.

Next Test This All Methods.

```
if __name__ == "__main__":
    # Load data
    data = pd.read_csv('./Output/FinalDataFrame.csv')

# Split features and target variable
X = data.drop('label', axis=1)
y = data['label']

# Convert data to NumPy arrays
X = np.array(X)
y = np.array(y)

# Split data into training and test sets

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=123)
```

Load data: the code reads in a CSV file called 'FinalDataFrame.csv' using Pandas and assigns it to a variable called 'data'.

split features and target variable: the feature columns are separated from the target column by dropping the 'label' column from the dataframe and assigning it to the variable 'X', and by assigning the 'label' column to the variable 'y'.

Convert data to NumPy arrays: the Pandas dataframe 'X' and 'y' are converted to NumPy arrays using the 'np.array()' function.

Split data into training and test sets: the 'train_test_split()' function from the scikit-learn library is used to split the data into training and test sets with a test size of 20% and a random state of 123. The features and target arrays are split into 'X_train', 'X_test', 'y_train', and 'y_test'.

```
# Train and evaluate Naive Bayes model
nb = NaiveBayes()
nb.fit(X_train, y_train)
predictions = nb.predict(X_test)


print("Naive Bayes classification accuracy:", accuracy_score(y_test, predictions))
```

```
Naive Bayes classification accuracy: 0.1392857142857143
```

Train and evaluate Naive Bayes model: an instance of the NaiveBayes class is created and fit to the training data using the 'fit()' method. Then, the 'predict()' method is used to generate

predictions for the test data. Finally, the accuracy of the predictions is calculated using the 'accuracy_score()' function from scikit-learn and printed it.

4.4.4 Naïve Baye's From Multi class Use Built In Libraries:

Here is Complete code:

Naive bayes For MultiClass use Built in Library

In [5]:
```
import pandas as pd
from sklearn.naive_bayes import MultinomialNB
from sklearn.metrics import accuracy_score
from sklearn.model_selection import train_test_split

data = pd.read_csv('./Output/FinalDataFrame.csv')

X = data.iloc[:, :-1]  # features
y = data.iloc[:, -1]   # target variable
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2)

clf = MultinomialNB()
clf.fit(X_train, y_train)
y_pred = clf.predict(X_test)

accuracy = accuracy_score(y_test, y_pred)
print('Accuracy:', accuracy)
```

```
Accuracy: 0.06428571428571428
```

Load data from a CSV file using pandas library.

Split the data into features (X) and target variable (y) using the iloc function.

Split the data into training and testing sets using the train_test_split function from scikit-learn.

Create an instance of the Multinomial Naive Bayes classifier.

Train the classifier using the training data using the fit method.

Use the trained classifier to predict the class labels for the test data using the predict method.

Calculate the accuracy of the classifier by comparing the predicted labels with the true labels using the accuracy_score function from scikit-learn.

Print the accuracy of the classifier.

4.4.5 Training Loss For Binary Class Of Naïve Baye's:

Here is Complete code:

Training loss for Binary class

In [4]:
```
import pandas as pd
from sklearn.naive_bayes import GaussianNB
from sklearn.model_selection import cross_val_score


# Load dataset
# Load CSV file
df = pd.read_csv('./Output/FinalDataFrameBinaryClass.csv')

# Split into features and target variable
X = df.iloc[:, :-1].values
y = df.iloc[:, -1].values

# Initialize Naive Bayes model
gnb = GaussianNB()

# Compute cross-validation scores
scores = cross_val_score(gnb, X, y, cv=5)

# Print average training loss
print("Average training loss: %.2f" % (1 - scores.mean()))
```

```
Average training loss: 0.11
```

The code imports necessary modules from the sklearn library: GaussianNB and cross_val_score. Then loads a dataset from a CSV file using pandas.

Then splits the dataset into features and target variables using the iloc method.

Then initializes a GaussianNB object called gnb.

Then calls cross_val_score method to compute the cross-validation scores of the Naive Bayes model. cv=5 specifies that 5-fold cross-validation is used. The X and y variables are passed as the features and target variable, respectively.

Then prints the average training loss, which is computed as 1 - mean of the cross-validation scores. The average training loss is a measure of how well the model fits the data.

4.4.6 Training Loss for Multi Class of Naïve Baye's:

Here is Complete code:

Training loss For Multiclass

In [5]:
```
import pandas as pd
from sklearn.naive_bayes import GaussianNB
from sklearn.model_selection import cross_val_score


# Load dataset
# Load CSV file
df = pd.read_csv('./Output/FinalDataFrame.csv')

# Split into features and target variable
X = df.iloc[:, :-1].values
y = df.iloc[:, -1].values

# Initialize Naive Bayes model
gnb = GaussianNB()

# Compute cross-validation scores
scores = cross_val_score(gnb, X, y, cv=5)

# Print average training loss
print("Average training loss: %.2f" % (1 - scores.mean()))
```

```
Average training loss: 0.88
```

Import necessary libraries: pandas, GaussianNB from sklearn.naive_bayes, cross_val_score from sklearn.model_selection. Then Load the dataset from a CSV file using pandas read_csv() function. Split the data into features and target variable.

Initialize a Gaussian Naive Bayes model.

Compute cross-validation scores using the cross_val_score() function. The function takes the classifier, the feature and target data, and the number of cross-validation folds as inputs. In this example, the number of folds is set to 5.

Compute the average training loss by subtracting the mean of the cross-validation scores from 1. The training loss is a measure of how well the model fits the training data. The lower the training loss, the better the model's fit to the training data.

.

4.4.7 Validation Loss for Binary Of Naïve Baye's:

Here is Complete code:

Validation loss for Binary class

In [8]:
```
from sklearn.naive_bayes import GaussianNB
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score

# Load CSV file
df = pd.read_csv('./Output/FinalDataFrameBinaryClass.csv')

# Split into features and target variable
X = df.iloc[:, :-1].values
y = df.iloc[:, -1].values

# Split data into training and validation sets
X_train, X_val, y_train, y_val = train_test_split(X, y, test_size=0.2, random_state=42)

# Initialize Naive Bayes model
gnb = GaussianNB()

# Train model on training data
gnb.fit(X_train, y_train)

# Make predictions on validation data
y_pred = gnb.predict(X_val)

# Compute validation loss (1 - accuracy)
val_loss = 1 - accuracy_score(y_val, y_pred)

# Print validation loss
print("Validation loss: %.2f" % val_loss)
```

```
Validation loss: 0.12
```

Import the required modules: GaussianNB for implementing the Gaussian Naive Bayes algorithm, train_test_split for splitting the data into training and validation sets, and accuracy_score for computing the accuracy of the predictions.

Load the dataset by reading a CSV file using pd.read_csv.

Split the data into the features and target variable using iloc.

Split the data into training and validation sets using train_test_split with a test size of 0.2 and a random state of 42.

Initialize a GaussianNB object to represent the Naive Bayes model.

Train the model on the training data using fit.

Make predictions on the validation data using predict.

Compute the validation loss by subtracting the accuracy of the predictions from 1.

Finally Print the validation loss.

4.4.8 Validation loss For Multi Class OF Naïve Baye's:

Here is complete code:

Validation loss for multi class

In [9]:
```
from sklearn.naive_bayes import GaussianNB
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score

# Load CSV file
df = pd.read_csv('./Output/FinalDataFrame.csv')

# Split into features and target variable
X = df.iloc[:, :-1].values
y = df.iloc[:, -1].values

# Split data into training and validation sets
X_train, X_val, y_train, y_val = train_test_split(X, y, test_size=0.2, random_state=42)

# Initialize Naive Bayes model
gnb = GaussianNB()

# Train model on training data
gnb.fit(X_train, y_train)

# Make predictions on validation data
y_pred = gnb.predict(X_val)

# Compute validation loss (1 - accuracy)
val_loss = 1 - accuracy_score(y_val, y_pred)

# Print validation loss
print("Validation loss: %.2f" % val_loss)
```

```
Validation loss: 0.90
```

Import necessary libraries: We import Gaussian Naive Bayes model from sklearn.naive_bayes and train_test_split and accuracy_score from sklearn.model_selection.metrics. Load the CSV file: The dataset is loaded from the CSV file using pandas read_csv function.

Split into features and target variable: The independent variables are separated from the target variable using the iloc method. The values attribute is used to convert the data frames into numpy arrays.

Split data into training and validation sets: The dataset is split into training and validation sets using the train_test_split function. The test_size argument specifies the percentage of data to be used for validation and the random_state argument is used to ensure reproducibility of the results.

Initialize Naive Bayes model: We create an instance of the Gaussian Naive Bayes model.

Train model on training data: We fit the model to the training data using the fit method.

Make predictions on validation data: We predict the class labels of the validation data using the predict method.

Compute validation loss (1 - accuracy): We compute the validation loss by subtracting the accuracy score from 1.

Print validation loss: We print the validation loss using the print function.

4.4.9 Testing Loss Of Binary Class Of Naïve Baye's:

Here is Complete Code:

Testing Loss for Binary class

In [12]:
```
from sklearn.naive_bayes import GaussianNB
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score

# Load CSV file
df = pd.read_csv('./Output/FinalDataFrameBinaryClass.csv')

# Split into features and target variable
X = df.iloc[:, :-1].values
y = df.iloc[:, -1].values

# Split data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)

# Initialize Naive Bayes model
gnb = GaussianNB()

# Train model on training data
gnb.fit(X_train, y_train)

# Make predictions on testing data
y_pred = gnb.predict(X_test)

# Compute testing loss (1 - accuracy)
test_loss = 1 - accuracy_score(y_test, y_pred)

# Print testing loss
print("Testing loss: %.2f" % test_loss)
```

```
Testing loss: 0.11
```

Import necessary modules: The code imports the GaussianNB class from the naive_bayes module, the train_test_split function from the model_selection module, and the accuracy_score function from the metrics module. Load CSV file: The code reads a CSV file into a pandas DataFrame.

Split into features and target variable: The code uses iloc to split the DataFrame into a feature matrix X and a target vector y.

Split data into training and testing sets: The code uses train_test_split to split the data into two sets: training and testing. The test_size parameter is set to 0.3, which means that 30% of the data is used for testing and 70% for training.

Initialize Naive Bayes model: The code creates an instance of the GaussianNB class.

Train model on training data: The code trains the model on the training data by calling the fit method of the GaussianNB object, passing in the training data.

Make predictions on testing data: The code uses the predict method of the GaussianNB object to generate predictions on the testing data.

Compute testing loss (1 - accuracy): The code computes the testing loss by subtracting the accuracy from 1. The accuracy is computed using the accuracy_score function from the metrics module, which compares the predicted values to the true values and returns the fraction of correct predictions.

Print testing loss: The code prints the testing loss to the console using a formatted string. The result shows the fraction of incorrect predictions on the testing data.

4.4.10 Testing Loss For Multi Class Of Naïve Baye's:

Here is complete code:

Testing loss for Multiclass

In [11]:
```
from sklearn.naive_bayes import GaussianNB
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score

# Load CSV file
df = pd.read_csv('./Output/FinalDataFrame.csv')

# Split into features and target variable
X = df.iloc[:, :-1].values
y = df.iloc[:, -1].values

# Split data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)

# Initialize Naive Bayes model
gnb = GaussianNB()

# Train model on training data
gnb.fit(X_train, y_train)

# Make predictions on testing data
y_pred = gnb.predict(X_test)

# Compute testing loss (1 - accuracy)
test_loss = 1 - accuracy_score(y_test, y_pred)

# Print testing loss
print("Testing loss: %.2f" % test_loss)
```

```
Testing loss: 0.88
```

Import necessary libraries. Then load CSV file.

Split into features and target variable:

Split data into training and testing sets:

Initialize Naive Bayes model by creating an instance of the GaussianNB class.

Train model on training data by calling the fit method on the model object and passing in the training data (X_train, y_train) as arguments.

Make predictions on testing data by calling the predict method on the model object and passing in the testing data (X_test) as an argument.

Compute testing loss (1 - accuracy) by subtracting the accuracy score from 1. The accuracy score is computed using the accuracy_score function, which takes the true labels (y_test) and predicted labels (y_pred) as arguments.

Print testing loss which is a measure of how well the model performs on new, unseen data. A lower testing loss indicates better performance.

4.5 SVM

SVM (Support Vector Machines) is a supervised machine learning algorithm used for classification or regression tasks that finds the best separating hyperplane to maximize the margin between classes. It can handle high-dimensional data and is effective in dealing with noisy data.

4.5.1 SVM For Binary Class From Scratch:

Here is code:

```
import numpy as np
import cvxopt

class BinarySVM:

    def _init_(self):
        """
        Parameters:
        C : float
            Regularization parameter
        """

        self.coef_ = None
        self.intercept_ = None
```

This is an implementation of a binary SVM (Support Vector Machine) classifier using the CVXOPT library to solve the dual SVM problem. This is the definition of the BinarySVM class, which has two attributes coef_ and intercept_ initialized as None.

```
def fit(self, X, y):
    """
    Fits SVM model to given training dataset
    """
    C = 1.0
    n_samples, n_features = X.shape

    # Compute the Gram matrix of size n_samples * n_samples
    K = np.dot(X, X.T)

    # Apply the Lagrange optimization to solve the dual SVM problem
    P = cvxopt.matrix(np.outer(y, y) * K)
    q = cvxopt.matrix(-1 * np.ones(n_samples))
    G = cvxopt.matrix(np.vstack((-np.identity(n_samples), np.identity(n_samples))))
    h = cvxopt.matrix(np.hstack((np.zeros(n_samples), C * np.ones(n_samples))))
    solution = cvxopt.solvers.qp(P, q, G, h)
    alphas = np.ravel(solution['x'])

    # Compute the support vectors and corresponding coefficients
    sv_indices = alphas > 1e-5
    self.support_vectors_ = X[sv_indices]
    self.support_vectors_labels_ = y[sv_indices]
    self.coef_ = np.dot(alphas * y, X)
    self.intercept_ = np.mean(self.support_vectors_labels_ - np.dot(self.support_vectors_, self.coef_))
```

This is the fit method, which takes as input a training dataset X of shape (n_samples, n_features) and corresponding labels y of shape (n_samples,). C is the regularization parameter, set to a default value of 1.0. The Gram matrix is computed as the dot product of the feature matrix X with its transpose X.T. This is used later in the optimization problem. The dual SVM problem is solved using CVXOPT's Quadratic Programming (QP) solver. where P is the Gram matrix multiplied by the outer product of y, q is an array of -1, G and h define inequality constraints on the optimization problem.

The solution to this optimization problem is a set of coefficients alphas that define the hyperplane in feature space that separates the two classes.

The support vectors and their corresponding coefficients are computed from the alphas. These are the points closest to the hyperplane that are used to define the margin. The support_vectors_ attribute is set to the subset of X that correspond to these support vectors, and the support_vectors_labels_ attribute is set to their corresponding labels.

The coefficients of the hyperplane are computed as the dot product of alphas with y and X, respectively. The intercept is computed as the mean of the support vector labels minus the dot product of the support vectors and the coefficients.

```
def predict(self, X):
    """
    Predicts class labels for given test dataset
    """
    decision_function = np.dot(X, self.coef_) + self.intercept_
    pred = np.sign(decision_function)
    return pred
```

This is the predict method of the BinarySVM class, which takes as input a test dataset X of shape (n_samples, n_features). The decision function is computed as the dot product of the test data X with the coefficients of the hyperplane self.coef_, plus the intercept self.intercept_.The predicted class labels are computed by taking the sign of the decision function. If the decision function is positive, the sample is classified as the positive class (+1), and if it's negative, the sample is classified as the negative class (-1). This is done using the np.sign function. The predicted class labels are then returned.

Now Test our code:

```
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import LabelEncoder

df=pd.read_csv('./Output/FinalDataFrameBinaryClass.csv')




# Split into features and target variable
X = df.iloc[:, :-1].values
y = df.iloc[:, -1].values

# Encode labels as numeric values
le = LabelEncoder()
y = le.fit_transform(y)

# Split data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)



# Create an instance of BinarySVM
svm = BinarySVM()

# Fit the SVM to the training data
svm.fit(X_train, y_train)

# Make predictions on the test data
y_pred = svm.predict(X_test)
accuracy = accuracy_score(y_test, y_pred)
print('Accuracy:', accuracy)
```

The features are extracted from the DataFrame and stored in X, while the target variable is stored in y. The labels in y are converted from their original string representation to numeric values using the LabelEncoder class from scikit-learn. The data is split into training and testing sets using the train_test_split function from scikit-learn. The training set will be used to train the SVM, while the testing set will be used to evaluate its performance. An instance of the BinarySVM class, which was defined previously, is created. The SVM is trained on the training data using the fit method of the BinarySVM class. The predict method of the BinarySVM class is used to make predictions on the testing data, and the accuracy of the classifier is calculated using the accuracy_score function from scikit-learn. The accuracy is then printed it.

4.5.2 SVM For Binary Cass Use built In Libraries:

Here is code:

```
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score, confusion_matrix
from sklearn.svm import SVC

# Load data from CSV file
df = pd.read_csv("./Output/FinalDataFrameBinaryClass.csv")
X = df.iloc[:, :-1].values
y = df.iloc[:, -1].values

# Split data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=123)

# Train the SVM on the training data
svm = SVC(kernel='linear', C=1.0, gamma='scale')
svm.fit(X_train, y_train)

# Use the trained SVM to predict the labels for the test data
y_pred = svm.predict(X_test)

# Calculate accuracy
accuracy = accuracy_score(y_test, y_pred)
print("Accuracy:", accuracy)

# Calculate precision, recall, and f1 score
precision = precision_score(y_test, y_pred, average='weighted')
recall = recall_score(y_test, y_pred, average='weighted')
f1 = f1_score(y_test, y_pred, average='weighted')
print("Precision:", precision)
print("Recall:", recall)
print("F1 Score:", f1)

# Calculate sensitivity and specificity
tn, fp, fn, tp = confusion_matrix(y_test, y_pred).ravel()
sensitivity = tp / (tp + fn)
specificity = tn / (tn + fp)
print("Sensitivity:", sensitivity)
print("Specificity:", specificity)
```

Import required libraries: Pandas, train_test_split from model_selection module, accuracy_score, precision_score, recall_score, f1_score and confusion_matrix from metrics module, and SVM from svm module.

Load data from the CSV file using pandas read_csv function and save the features in X and the target variable in y.

Split the data into training and testing sets using the train_test_split function from sklearn, with a test size of 20% and a random state of 123.

Create an SVM object with a linear kernel, C=1.0, and gamma='scale'.

Fit the SVM object to the training data using the fit method.

Predict the labels of the test data using the predict method of the SVM object.

Calculate the accuracy score by comparing the predicted labels with the actual labels of the test data using the accuracy_score function.

Calculate the precision, recall, and F1 score by comparing the predicted labels with the actual labels of the test data using precision_score, recall_score, and f1_score functions respectively.

Calculate the sensitivity and specificity by computing the confusion matrix using confusion_matrix function and extracting the values for true negative, false positive, false negative, and true positive.

Print out the accuracy, precision, recall, F1 score, sensitivity, and specificity.

4.5.3 SVM For Multi Class Use Built In Libraries:

Here is Code:

```
# Import required libraries
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.svm import SVC
from sklearn.metrics import accuracy_score

# Load dataset using pandas
data = pd.read_csv('./Output/FinalDataFrame.csv')

# Split data into features and target variable
X = data.drop('label', axis=1)
y = data['label']

# Split data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)

# Create SVM classifier object
svm_classifier = SVC(kernel='linear', C=1, decision_function_shape='ovo')

# Train SVM classifier on training data
svm_classifier.fit(X_train, y_train)

# Make predictions on test data
y_pred = svm_classifier.predict(X_test)

# Evaluate model performance using accuracy score
accuracy = accuracy_score(y_test, y_pred)

# Print model accuracy
print("Accuracy:", accuracy)
```

In this step, we import the necessary libraries to implement SVM using built-in libraries for multiclass classification file read through Pandas.

Load dataset using pandas:
Split data into features and target variable:
Split data into training and testing sets:
Create SVM classifier object:
Train or fit SVM classifier on training data:
Make predictions on test data:
Evaluate model performance using accuracy score:
Finally print model accuracy: