Practical 9

Aim: Naive Bayes' learning algorithm.

Theory:

Naive Bayes is a family of probabilistic machine learning algorithms based on the Bayes Theorem with an assumption of independence among the features.

The Naive Bayes classifier assumes that the presence of a feature in a class is not related to any other feature.

Naive Bayes is a classification algorithm for binary and multi-class classification problems.

Naive Bayes Theorem

Based on prior knowledge of conditions that may be related to an event, Bayes theorem describes the probability of the event conditional probability can be found this way.

Assume we have a Hypothesis(H) and evidence(E),

According to Bayes theorem, the relationship between the probability of the Hypothesis before getting the evidence represented as P(H) and the probability of the hypothesis after getting the evidence represented as P(H|E) is:

P(H|E) = P(E|H)*P(H)/P(E)

Prior probability = P(H) is the probability before getting the evidence

Posterior probability = P(H|E) is the probability after getting evidence

In general,

P(class|data) = (P(data|class) * P(class)) / P(data)

Code/Output:

Naive Bayes Scratch Implementation using Python

Here we are implementing a Naive Bayes Algorithm using Gaussian distributions. It performs all the necessary steps from data preparation and model training to testing and evaluation.

Importing Libraries

Importing necessary libraries:

1. **math: for mathematical operations**
2. **random: for random number generation**
3. **pandas: for data manipulation**
4. **numpy: for scientific computing**

Code:1.

```
import math
import random
import pandas as pd
import numpy as np
```

2. Encode Class

The encode_class function converts class labels in the dataset into numeric values. It assigns a unique numeric identifier to each class.

```
def encode_class(mydata):
    classes = []
    for i in range(len(mydata)):
        if mydata[i][-1] not in classes:
            classes.append(mydata[i][-1])
    for i in range(len(classes)):
        for j in range(len(mydata)):
            if mydata[j][-1] == classes[i]:
                mydata[j][-1] = i
    return mydata
```

3. Data Splitting

The splitting function is used to split the dataset into training and testing sets based on the given ratio.

```
def splitting(mydata, ratio):
    train_num = int(len(mydata) * ratio)
    train = []
    test = list(mydata)
    while len(train) < train_num:
        index = random.randrange(len(test))
        train.append(test.pop(index))
    return train, test
```

4. Group Data by Class

The groupUnderClass function takes the data and returns a dictionary where each key is a class label and the value is a list of data points belonging to that class.

```
def groupUnderClass(mydata):
    data_dict = {}
    for i in range(len(mydata)):
        if mydata[i][-1] not in data_dict:
            data_dict[mydata[i][-1]] = []
        data_dict[mydata[i][-1]].append(mydata[i])
    return data_dict
```

5. Calculate Mean and Standard Deviation for Class

The **MeanAndStdDev** function takes a list of numbers and calculates the mean and standard deviation.
The **MeanAndStdDevForClass** function takes the data and returns a dictionary where each key is a class label and the value is a list of lists, where each inner list contains the mean and standard deviation for each attribute of the class.

```
def MeanAndStdDev(numbers):
    avg = np.mean(numbers)
    stddev = np.std(numbers)
    return avg, stddev
def MeanAndStdDevForClass(mydata):
    info = {}
    data_dict = groupUnderClass(mydata)
    for classValue, instances in data_dict.items():
        info[classValue] = [MeanAndStdDev(attribute) for attribute in
zip(*instances)]
    return info
```

6. Calculate Gaussian and Class Probabilities

The calculateGaussianProbability function takes a value, mean, and standard deviation and calculates the probability of the value occurring under a Gaussian distribution with that mean and standard deviation.The calculateClassProbabilities function takes the information dictionary and a test data point as arguments. It iterates through each class and calculates the probability of the test data point belonging to that class based on the mean and standard deviation of each attribute for that class.

```
def calculateGaussianProbability(x, mean, stdev):
    epsilon = 1e-10
    expo = math.exp(-(math.pow(x - mean, 2) / (2 * math.pow(stdev +
epsilon, 2))))
    return (1 / (math.sqrt(2 * math.pi) * (stdev + epsilon))) * expo

def calculateClassProbabilities(info, test):
    probabilities = {}
    for classValue, classSummaries in info.items():
        probabilities[classValue] = 1
        for i in range(len(classSummaries)):
            mean, std_dev = classSummaries[i]
            x = test[i]
            probabilities[classValue] *=
calculateGaussianProbability(x, mean, std_dev)
```

```
    return probabilities
```

7. **Prediction for Test Set**

The predict function takes the information dictionary and a test data point as arguments. It calculates the class probabilities and returns the class with the highest probability.
The getPredictions function takes the information dictionary and the test set as arguments. It iterates through each test data point and predicts its class using the predict function.

```
def predict(info, test):
    probabilities = calculateClassProbabilities(info, test)
    bestLabel = max(probabilities, key=probabilities.get)
    return bestLabel

def getPredictions(info, test):
    predictions = [predict(info, instance) for instance in test]
    return predictions
```

8. **Calculate Accuracy**

The accuracy_rate function takes the test set and the predictions as arguments. It compares the predicted classes with the actual classes and calculates the percentage of correctly predicted data points.

```
def accuracy_rate(test, predictions):
    correct = sum(1 for i in range(len(test)) if test[i][-1] ==
predictions[i])
    return (correct / float(len(test))) * 100.0
```

9. **Load and Preprocess Data**

The code then loads the data from a CSV file using pandas and converts it into a list of lists. It then encodes the class labels and converts all attributes to floating-point numbers.

```
# Load data using pandas
filename = '/content/diabetes_data.csv'  # Add the correct file path
df = pd.read_csv(filename)
mydata = df.values.tolist()

# Encode classes and convert attributes to float
mydata = encode_class(mydata)
for i in range(len(mydata)):
    for j in range(len(mydata[i]) - 1):
        mydata[i][j] = float(mydata[i][j])
```

1. **Split Data into Training and Testing Sets**

The code splits the data into training and testing sets using a specified ratio. It then trains the model by calculating the mean and standard deviation for each attribute in each class.

```
# Split the data into training and testing sets
ratio = 0.7
train_data, test_data = splitting(mydata, ratio)

print('Total number of examples:', len(mydata))
print('Training examples:', len(train_data))
print('Test examples:', len(test_data))
```

Output:

```
Total number of examples: 768
Training examples: 537
Test examples: 231
```

11. Train and Test the Model

Calculate mean and standard deviation for each attribute within each class for the training set. Finally, it tests the model on the test set and calculates the accuracy.

```
# Train the model
info = MeanAndStdDevForClass(train_data)

# Test the model
predictions = getPredictions(info, test_data)
accuracy = accuracy_rate(test_data, predictions)
print('Accuracy of the model:', accuracy)
```

Output:

```
Accuracy of the model: 100.0
```