Experiment Title- 2.2

Student Name: Lipakshi
UID: 20BCS5082
Branch: BE-CSE
Section/Group-20BCSWM_607-B
Semester:5
Date of Performance:10/10/2022
Subject Name: Machine Learning Lab
Subject Code: 20CSP-317

1. Aim: To Implement Naïve Bayes algorithm and justify the outcome with relevant Parameters.

2. Source Code and Output:

Naive Bayes experiment

First we will develop each piece of the algorithm, then we will tie all of the elements together into a working implementation applied to a real dataset.

This Naive Bayes tutorial is broken down into 5 parts:

Step 1: Separate By Class.

Department of Computer Science & Engineering Page 34

Step 2: Summarize Dataset.

Step 3: Summarize Data By Class.

Step 4: Gaussian Probability Density Function.

Step 5: Class Probabilities.

These steps will provide the foundation that you need to implement Naive Bayes from scratch

and apply it to your own predictive modeling problems.

Step 1: Separate By Class

We will need to calculate the probability of data by the class they belong to, the so-called base rate.

This means that we will first need to separate our training data by class. A relatively straightforward operation.

We can create a dictionary object where each key is the class value and then add a list of all the records as the value in the dictionary.

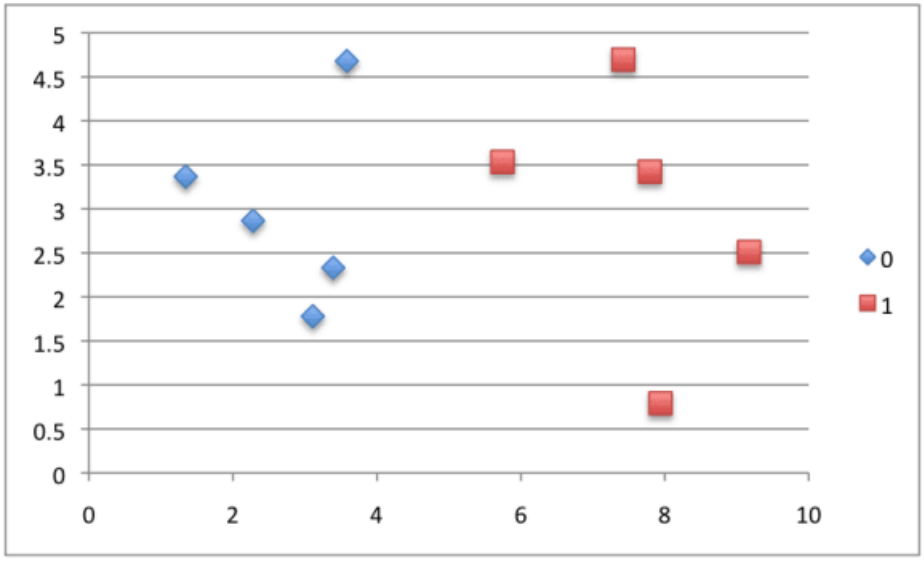
Below is a function named *separate_by_class()* that implements this approach. It assumes that the last column in each row is the class value.

```
# Split the dataset by class values, returns a dictionary
def separate_by_class(dataset):
        separated = dict()
        for i in range(len(dataset)):
                vector = dataset[i]
                class_value = vector[-1]
                if (class_value not in separated):
                        separated[class_value] = list()
                separated[class_value].append(vector)
        return separated
```

We can contrive a small dataset to test out this function.

X1	X2	Y
3.393533211	2.331273381	0
3.110073483	1.781539638	0
1.343808831	3.368360954	0
3.582294042	4.67917911	0
2.280362439	2.866990263	0
7.423436942	4.696522875	1
5.745051997	3.533989803	1
9.172168622	2.511101045	1
7.792783481	3.424088941	1
7.939820817	0.791637231	1

We can plot this dataset and use separate colors for each class.



Scatter Plot of Small Contrived Dataset for Testing the Naive Bayes Algorithm

Putting this all together, we can test our *separate_by_class()* function on the contrived dataset.

```
# Example of separating data by class value

# Split the dataset by class values, returns a dictionary
def separate_by_class(dataset):
        separated = dict()
        for i in range(len(dataset)):
                vector = dataset[i]
                class_value = vector[-1]
                if (class_value not in separated):
                        separated[class_value] = list()
                separated[class_value].append(vector)
        return separated
```

```
# Test separating data by class
dataset = [[3.393533211,2.331273381,0],
        [3.110073483,1.781539638,0],
        [1.343808831,3.368360954,0],
        [3.582294042,4.67917911,0],
        [2.280362439,2.866990263,0],
        [7.423436942,4.696522875,1],
        [5.745051997,3.533989803,1],
        [9.172168622,2.511101045,1],


        [7.792783481,3.424088941,1],
        [7.939820817,0.791637231,1]]
separated = separate_by_class(dataset)
for label in separated:
        print(label)
        for row in separated[label]:
                print(row)
```

Running the example sorts observations in the dataset by their class value, then prints the class value followed by all identified records.

```
0
[3.393533211, 2.331273381, 0]
[3.110073483, 1.781539638, 0]
[1.343808831, 3.368360954, 0]
[3.582294042, 4.67917911, 0]
[2.280362439, 2.866990263, 0]
1
[7.423436942, 4.696522875, 1]
[5.745051997, 3.533989803, 1]
[9.172168622, 2.511101045, 1]
[7.792783481, 3.424088941, 1]
[7.939820817, 0.791637231, 1]
```

Next we can start to develop the functions needed to collect statistics.

Step 2: Summarize Dataset

We need two statistics from a given set of data.

We'll see how these statistics are used in the calculation of probabilities in a few steps. The two statistics we require from a given dataset are the mean and the standard deviation (average deviation from the mean).

The mean is the average value and can be calculated as:

mean = sum(x)/n * count(x)

Where *x* is the list of values or a column we are looking.

Below is a small function named *mean()* that calculates the mean of a list of numbers.

```
1 # Calculate the mean of a list of numbers
2 def mean(numbers):
3         return sum(numbers)/float(len(numbers))
```

The sample standard deviation is calculated as the mean difference from the mean value. This can be calculated as:

standard deviation = sqrt((sum i to N (x_i – mean(x))^2) / N-1)

You can see that we square the difference between the mean and a given value, calculate the average squared difference from the mean, then take the square root to return the units back to their original value.

Below is a small function named *standard_deviation()* that calculates the standard deviation of a list of numbers. You will notice that it calculates the mean. It might be more efficient to calculate the mean of a list of numbers once and pass it to the *standard_deviation()* function as a parameter. You can explore this optimization if you're interested later.

```
1 from math import sqrt
2
3 # Calculate the standard deviation of a list of numbers
4 def stdev(numbers):
5         avg = mean(numbers)
6         variance = sum([(x-avg)**2 for x in numbers]) / float(len(numbers)-1)
7         return sqrt(variance)
```

We require the mean and standard deviation statistics to be calculated for each input attribute or each column of our data.

We can do that by gathering all of the values for each column into a list and calculating the mean and standard deviation on that list. Once calculated, we can gather the statistics together into a list or tuple of statistics. Then, repeat this operation for each column in the dataset and return a list of tuples of statistics.

Below is a function named *summarize_dataset()* that implements this approach. It uses some Python tricks to cut down on the number of lines required.

```
# Calculate the mean, stdev and count for each column in a dataset
def summarize_dataset(dataset):
        summaries = [(mean(column), stdev(column), len(column)) for column in zip(*dataset)]
        del(summaries[-1])
        return summaries
```

The first trick is the use of the zip() function that will aggregate elements from each provided argument. We pass in the dataset to the *zip()* function with the * operator that separates the dataset (that is a list of lists) into separate lists for each row. The *zip()* function then iterates over each element of each row and returns a column from the dataset as a list of numbers. A clever little trick.

We then calculate the mean, standard deviation and count of rows in each column. A tuple is created from these 3 numbers and a list of these tuples is stored. We then remove the statistics for the class variable as we will not need these statistics.

Step 3: Summarize Data By Class

We require statistics from our training dataset organized by class.

Above, we have developed the *separate_by_class()* function to separate a dataset into rows by class. And we have developed *summarize_dataset()* function to calculate summary statistics for each column.

We can put all of this together and summarize the columns in the dataset organized by class values.

Below is a function named *summarize_by_class()* that implements this operation. The dataset is first split by class, then statistics are calculated on each subset. The results in the form of a list of tuples of statistics are then stored in a dictionary by their class value.

```
# Split dataset by class then calculate statistics for each row
def summarize_by_class(dataset):
        separated = separate_by_class(dataset)
        summaries = dict()
        for class_value, rows in separated.items():
                summaries[class_value] = summarize_dataset(rows)
        return summaries
```

Step 4: Gaussian Probability Density Function

Calculating the probability or likelihood of observing a given real-value like X1 is difficult. One way we can do this is to assume that X1 values are drawn from a distribution, such as a bell curve or Gaussian distribution.

A Gaussian distribution can be summarized using only two numbers: the mean and the standard deviation. Therefore, with a little math, we can estimate the probability of a given value. This piece of math is called a Gaussian Probability Distribution Function (or Gaussian PDF) and can be calculated as:

f(x) = (1 / sqrt(2 * PI) * sigma) * exp(-((x-mean)^2 / (2 * sigma^2)))

Where *sigma* is the standard deviation for *x*, *mean* is the mean for *x* and *PI* is the value of pi.

Below is a function that implements this. I tried to split it up to make it more readable.

```
# Calculate the Gaussian probability distribution function for x
def calculate_probability(x, mean, stdev):
    exponent = exp(-((x-mean)**2 / (2 * stdev**2 )))
    return (1 / (sqrt(2 * pi) * stdev)) * exponent
```

Step 5: Class Probabilities

Now it is time to use the statistics calculated from our training data to calculate probabilities for new data.

Probabilities are calculated separately for each class. This means that we first calculate the probability that a new piece of data belongs to the first class, then calculate probabilities that it belongs to the second class, and so on for all the classes.

The probability that a piece of data belongs to a class is calculated as follows:

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

You may note that this is different from the Bayes Theorem described above.

The division has been removed to simplify the calculation.

This means that the result is no longer strictly a probability of the data belonging to a class. The value is still maximized, meaning that the calculation for the class that results in the largest value is taken as the prediction. This is a common implementation simplification as we are often more interested in the class prediction rather than the probability.

The input variables are treated separately, giving the technique it's name "*naive*". For the above example where we have 2 input variables, the calculation of the probability that a row belongs to the first class 0 can be calculated as:

P(class=0|X1,X2) = P(X1|class=0) * P(X2|class=0) * P(class=0)

Now you can see why we need to separate the data by class value. The Gaussian Probability Density function in the previous step is how we calculate the probability of a real value like X1 and the statistics we prepared are used in this calculation.

Learning outcomes (What I have learnt):

1. I learnt what is Bayes Theorem.
2. Practical use of Bayes Theorem.
3. How machine learning helps to analyze datasets.

Evaluation Grid :

Sr. No.	Parameters	Marks Obtained	Maximum Marks
1.	Student Performance (Conduct of experiment) objectives/Outcomes.		12
2.	Viva Voce		10

3.	**Submission of Work Sheet (Record)**		**8**
	Total		**30**