MACHINE LEARNING LAB PRACTICAL FILE

Faculty name: Dr. Himanshu Khanna

Student name: Dipen Kalsi

Roll No: 01514812721

Semester: 7th

Group: 7CST-AIML3

Maharaja Agrasen Institute of Technology, PSP Area,

Sector – 22, Rohini, New Delhi – 110085

INDEX

Name: Dipen Kalsi

Enrolment Number: 01514812721

Branch: Computer Science and Technology

Group: 7CST/AIML-3A

S.No.	Experiment Name	Date	R1	R2	R3	R4	R5	Total Marks	Signature
1.	Introduction to JUPYTER IDE and its libraries Pandas and NumPy.								
2.	Write a Program to demonstrate Simple Linear Regression.								
3.	Write a Program to demonstrate Logistic Regression.								
4.	Write a Program to demonstrate Decision Tree ID3 Algorithm.								
5.	Write a Program to demonstrate k-Nearest Neighbor flowers (Iris) classification.								

6.	Write a program to demonstraite naïve bayes classifier.								
7.	Write a program to demonstrate PCA and LDA on iris dataset.								
8.	Write a program to demonstrate DBSCAN clustering algorithm.								
9.	Write a Program to demonstrate K-Medoid clustering algorithm.								
10.	Write a Program to demonstrate K-Means clustering algorithm on Handwritten dataset.								

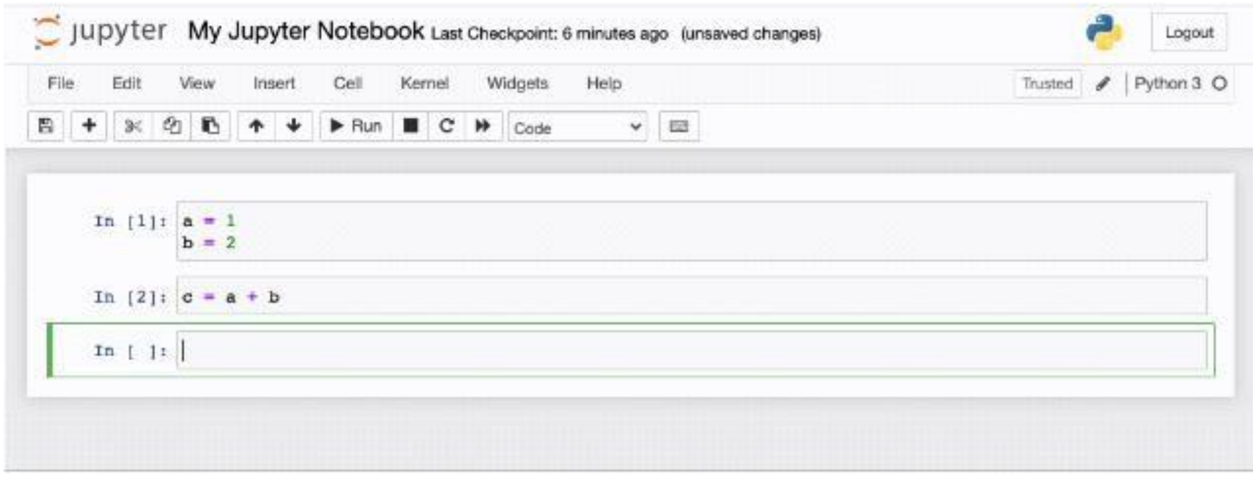
Experiment 1

Aim - Introduction to JUPYTER IDE and its libraries Pandas and NumPy.

Theory

Jupyter Notebook (sometimes called IPython Notebook) is a popular way to write and run Python code, especially for data analysis, data science and machine learning. Jupyter Notebooks are easy-to-use because they let you execute code and review the output quickly. This iterative process is central to data analytics and makes it easy to test hypotheses and record the results (just like a notebook).

For example, let's say you are visualizing a dataset about life expectancy by country. You only want to show some countries, but you are not sure which ones to select. With a Jupyter Notebook, you can try multiple versions and easily compare. Even better, you have a written record of what you've already tried that you can show a teammate (or your future self). This is just one example of the many benefits of working within a notebook-like environment.


jupyter
My Jupyter Notebook Last Checkpoint: 6 minutes ago (unsaved changes)
Logout
File Edit View Insert Cell Kernel Widgets Help
Trusted
Python 3
Run
Code

```
In [1]: a = 1
        b = 2

In [2]: c = a + b

In [ ]:
```

Jupyter Notebook uses a back-end kernel called IPython. The 'I' stands for 'Interactive', which means that a program or script can be broken up into smaller pieces, and those pieces can be run independently from the rest of the program. You do not need to worry about the difference between Python and IPython. The important thing to know is that you can run small pieces of code, which can be helpful when working with data.

Integrated Development Environments (IDEs) - Jupyter Notebook is a type of Integrated Development Environment (IDE). IDEs are places to write code that offer some supportive features. Almost all IDEs provide syntax highlighting, debugging, and code completion. Jupyter

Notebook also offers embedded help documentation and introspection (i.e., you can check each command's parameters) and in-line display of charts and images.

Pandas - Pandas is a very popular library for working with data (its goal is to be the most powerful and flexible open-source tool, and in our opinion, it has reached that goal). DataFrames are at the center of pandas. A DataFrame is structured like a table or spreadsheet. The rows and the columns both have indexes, and you can perform operations on rows or columns separately. A pandas DataFrame can be easily changed and manipulated. Pandas has helpful functions for handling missing data, performing operations on columns and rows, and transforming data. If that wasn't enough, a lot of SQL functions have counterparts in pandas, such as join, merge, filter by, and group by. With all of these powerful tools, it should come as no surprise that pandas is very popular among data scientists.

NumPy - NumPy is an open-source Python library that facilitates efficient numerical operations on large quantities of data. There are a few functions that exist in NumPy that we use on pandas DataFrames. For us, the most important part about NumPy is that pandas is built on top of it. So, NumPy is a dependency of Pandas.

Installation

If you have Anaconda installed, NumPy and pandas may have been auto-installed as well! If they haven't been, or if you want to update to the latest versions, you can open a terminal window and run the following commands:

```
conda install numpy conda
install pandas
```

If you don't have Anaconda installed, you can alternatively install the libraries using pip by running the following commands from your terminal:

```
pip install numpy pip
install pandas
```

Once you’ve installed these libraries, you’re ready to open any Python coding environment (we recommend Jupyter Notebook). Before you can use these libraries, you’ll need to import them using the following lines of code. We’ll use the abbreviations np and pd, respectively, to simplify our function calls in the future.

```
import numpy as np import

pandas as pd
```

NumPy Arrays - NumPy arrays are unique in that they are more flexible than normal Python lists. They are called ndarrays since they can have any number (n) of dimensions (d). They hold a collection of items of any one data type and can be either a vector (one-dimensional) or a matrix (multi-dimensional). NumPy arrays allow for fast element access and efficient data manipulation.

The code below initializes a Python list named list1:

```
list1 = [1,2,3,4]
```

To convert this to a one-dimensional ndarray with one row and four columns, we can use the np.array() function:

```
array1 = np.array(list1)

print(array1)
```

Output:

```
[1 2 3 4]
```

To get a two-dimensional ndarray from a list, we must start with a Python list of lists:

```
list2 = [[1,2,3],[4,5,6]]
```

```
array2 = np.array(list2)

print(array2)
```

Output:

```
[[1 2 3]
 [4 5 6]]
```

In the above output, you may notice that the NumPy array print-out is displayed in a way that clearly demonstrates its multi-dimensional structure: two rows and three columns.
Many operations can be performed on NumPy arrays which makes them very helpful for manipulating data:

Selecting array elements

Slicing arrays

Reshaping arrays

Splitting arrays

Combining arrays

Numerical operations (min, max, mean, etc) - Mathematical operations can be performed on all values in a ndarray at one time rather than having to loop through values, as is necessary with a Python list. This is very helpful in many scenarios. Say you own a toy store and decide to decrease the price of all toys by €2 for a weekend sale. With the toy prices stored in an ndarray, you can easily facilitate this operation.

```
toyPrices = np.array([5,8,3,6])

print(toyPrices - 2)
```

Output:

```
[3 6 1 4]
```

If, however, you had stored your toy prices in a Python list, you would have to manually loop through the entire list to decrease each toy price.

```
toyPrices = [5,8,3,6]

# print(toyPrices - 2) -- Not possible. Causes an error

for i in range(len(toyPrices)):

        toyPrices[i]

        -= 2

        print(toyPric

        es)
```

Output:

```
[3,6,1,4]
```

Pandas Series and Dataframes

Just as the ndarray is the foundation of the NumPy library, the Series is the core object of the pandas library. A pandas Series is very similar to a one-dimensional NumPy array, but it has additional functionality that allows values in the Series to be indexed using labels. A NumPy array does not have the flexibility to do this. This labeling is useful when you are storing pieces of data that have other data associated with them. Say you want to store the ages of students in an online course to eventually figure out the average student age. If stored in a NumPy array, you could only access these ages with the internal ndarray indices 0,1,2. With a Series object, the indices of values are set to 0,1,2... by default, but you can customize the indices to be other values such as student names so an age can be accessed using a name. Customized indices of a Series are established by sending values into the Series constructor, as you will see below.

A Series holds items of any one data type and can be created by sending in a scalar value, Python list, dictionary, or ndarray as a parameter to the pandas Series constructor. If a dictionary is sent in, the keys may be used as the indices.

```
# Create a Series using a NumPy array of ages with the default numerical indices

ages = np.array([13,25,19])

series1 = pd.Series(ages)

print(series1)
```

Output:

```
0  |  13

1  |  25

2  |  19

dtype: int64
```

When printing a Series, the data type of its elements is also printed. To customize the indices of a Series object, use the index argument of the Series constructor.

```
# Create a Series using a NumPy array of ages but customize the indices to be the names that correspond to each age

    ages = np.array([13,25,19])

    series1 = pd.Series(ages,index=['Emma', 'Swetha',

    'Serajh']) print(series1)
```

Output:

```
Emma    |  13

Swetha  |  25

Serajh  |  19
```

```
dtype: int64
```

Series objects provide more information than NumPy arrays do. Printing a NumPy array of ages does not print the indices or allow us to customize them.

```
ages = np.array([13,25,19])

print(ages)
```

Output: [13 25 19]

Another important type of object in the pandas library is the DataFrame. This object is similar in form to a matrix as it consists of rows and columns. Both rows and columns can be indexed with integers or String names. One DataFrame can contain many different types of data types, but within a column, everything has to be the same data type. A column of a DataFrame is essentially a Series. All columns must have the same number of elements (rows).

There are different ways to fill a DataFrame such as with a CSV file, a SQL query, a Python list, or a dictionary. Here we have created a DataFrame using a Python list of lists. Each nested list represents the data in one row of the DataFrame. We use the keyword columns to pass in the list of our custom column names.

```
dataf = pd.DataFrame([

['John Smith','123 Main St',34],

['Jane Doe', '456 Maple Ave',28],

['Joe Schmo', '789 Broadway',51]],

columns=['name','address','age'])
```

This is how the DataFrame is displayed:

```
        name      |  address     |  age
0    | John Smith   | 123 Main St   |   34
1    | Jane Doe     | 456 Maple Ave |   28
2    | Joe Schmo    | 789 Broadway  |   51
```

The default row indices are 0,1,2..., but these can be changed. For example, they can be set to be the elements in one of the columns of the DataFrame. To use the names column as indices instead of the default numerical values, we can run the following command on our DataFrame:

```
dataf.set_index('name')
```

Output:

```
   name      |  address     | age
John Smith  | 123 Main St  |  34
Jane Doe     | 456 Maple Ave |   28
Joe Schmo    | 789 Broadway  |   51
```

DataFrames are useful because they make it much easier to select, manipulate, and summarize data. Their tabular format (a table with rows and columns) also makes it easier to label, simpler to read, and easier to export data to and from a spreadsheet. Understanding the power of these new data structures is the key to unlocking many new avenues for data manipulation, exploration, and analysis!

Conclusion:

Successfully learned and applied basic python data manipulation operations using Jupyter notebook.

Viva Questions

Q.1 What are Pandas?

A.1 Pandas is an open-source Python library that is built on top of the NumPy library. It is made for working with relational or labelled data. It provides various data structures for manipulating, cleaning and analyzing numerical data. It can easily handle missing data as well. Pandas are fast and have high performance and productivity.

Q.2 What are the Different Types of Data Structures in Pandas?

A. 2 The two data structures that are supported by Pandas are Series and DataFrames.

i. Pandas Series is a one-dimensional labelled array that can hold data of any type. It is mostly used to represent a single column or row of data.
ii. Pandas DataFrame is a two-dimensional heterogeneous data structure. It stores data in a tabular form. Its three main components are data, rows, and columns.

Q. 3 List Key Features of Pandas.

A. 3 Pandas are used for efficient data analysis. The key features of Pandas are as follows:

i. Fast and efficient data manipulation and analysis.
ii. Provides time-series functionality.
iii. Easy missing data handling.
iv. Faster data merging and joining.
v. Flexible reshaping and pivoting of data sets.
vi. Powerful group by functionality.
vii. Data from different file objects can be loaded.
viii. Integrates with NumPy.

Q. 4 What is NumPy, and why is it popular in the field of scientific computing?

A.4 NumPy is a powerful Python library for numerical and matrix operations. It provides support for large, multi-dimensional arrays and matrices, along with mathematical functions to operate on these arrays efficiently.

Q. 5 Which is faster - NumPy or Pandas?

A.5 Pandas is more user-friendly, but NumPy is faster. Pandas has a lot more options for handling missing data, but NumPy has better performance on large datasets. Pandas uses Python objects internally, making it easier to work with than NumPy (which uses C arrays).

Experiment 2

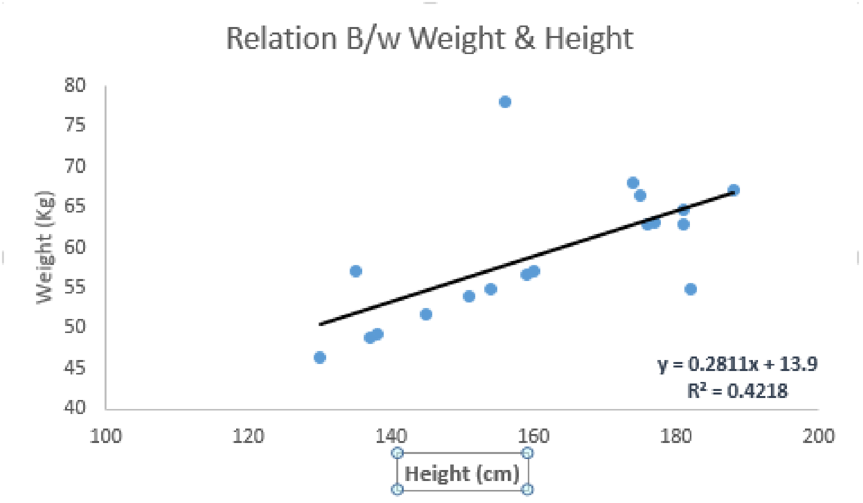
Aim - Write a Program to demonstrate Simple Linear Regression.

Theory

Linear Regression is a machine learning algorithm based on supervised learning. It performs a regression task. Regression models a target prediction value based on independent variables. It is mostly used for finding out the relationship between variables and forecasting. Different regression models differ based on – the kind of relationship between the dependent and independent variables, they are considering and the number of independent variables being used.

It is used to estimate real values (cost of houses, number of calls, total sales etc.) based on continuous variable(s). Here, we establish relationship between independent and dependent variables by fitting a best line. This best fit line is known as regression line and represented by a linear equation Y= a*X + b.

Look at the below example. Here we have identified the best fit line having linear equation **y=0.2811x+13.9**. Now using this equation, we can find the weight, knowing the height of a person.



Linear Regression is of mainly two types: Simple Linear Regression and Multiple Linear Regression

Python Code flow

```
#Import Library

#Import other necessary libraries like pandas, numpy...

from sklearn import linear_model

#Load Train and Test datasets

#Identify feature and response variable(s) and values must be numeric and
numpy arrays

x_train=input_variables_values_training_datasets

y_train=target_variables_values_training_datasets

x_test=input_variables_values_test_datasets

# Create linear regression object

linear = linear_model.LinearRegression()

# Train the model using the training sets and check score

linear.fit(x_train, y_train)

linear.score(x_train, y_train)

#Equation coefficient and Intercept

print('Coefficient: \n', linear.coef_)

print('Intercept: \n', linear.intercept_)

#Predict Output

predicted= linear.predict(x_test)
```

Python Script

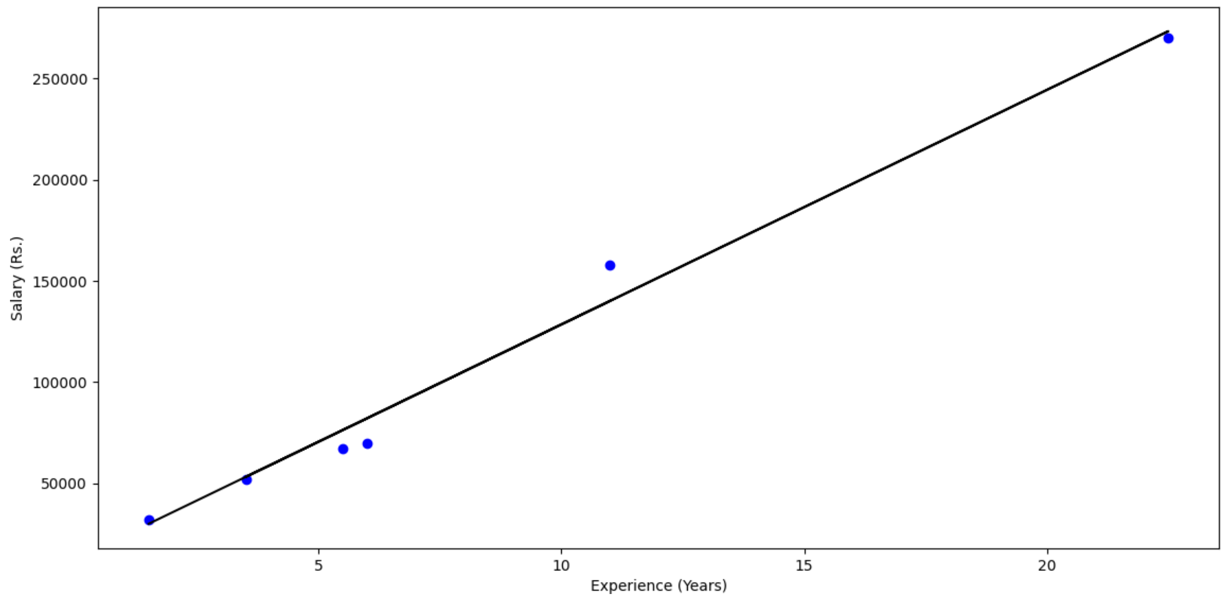
```
# Importing all the required libraries

import numpy as np
import pandas as pd

import matplotlib.pyplot as plt

from sklearn.model_selection import train_test_split

from sklearn.linear_model import LinearRegression

data_set= pd.read_csv('Salary_Data.csv')

df_binary = data_set[['Experience', 'Salary']]



# Taking only the selected two attributes from the dataset

df_binary.columns = ['Salary', 'Experience']

# display the first 5 rows

print(df_binary.head())



# Eliminating NaN or missing input number
```

```
df_binary.fillna(method ='ffill', inplace = True)

# Converting each dataframe into a numpy array, since each dataframe
contains only one column,

# Separating the data into independent and dependent variables

X = np.array(df_binary['Salary']).reshape(-1, 1)

y = np.array(df_binary['Experience']).reshape(-1, 1)

# Dropping any rows with Nan values

df_binary.dropna(inplace=True)

X_train,X_test,y_train,y_test    =train_test_split(X,y, test_size=0.25)

# Splitting the data into training and testing data

regr = LinearRegression()

regr.fit(X_train, y_train)

print(regr.score(X_test, y_test))

y_pred = regr.predict(X_test)

plt.scatter(X_test, y_test, color='b')

plt.plot(X_test, y_pred, color='k')

plt.xlabel('Experience (Years)')

plt.ylabel('Salary (Rs.)')

plt.show()
```

“Salary_Data.csv” file

Index	**Experience**	**Salary**
0	10	100000
1	5	60000
2	3	50000
3	15	190000
4	21	250000
5	2	40000
6	3.5	52000
7	1	30000
8	12	185000
9	11	158000
10	11.5	160000
11	10.5	150000
12	4	55000
13	7	75000
14	6	70000
15	22	260000
16	1.5	32000
17	2.5	45000
18	4.5	58000
19	5.5	67000
20	22.5	270000

Output

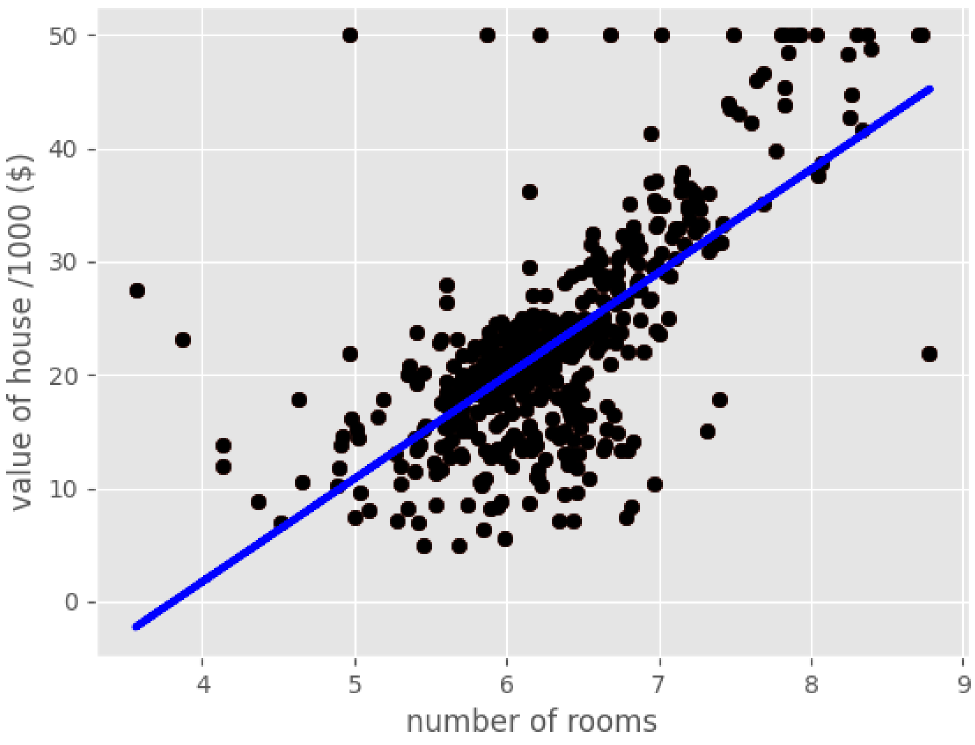


<u>Python Script (Using boston dataset)</u>

```
# Import necessary packages

import matplotlib.pyplot as plt

plt.style.use('ggplot')

from sklearn import datasets

from sklearn import linear_model



# Load data

boston = datasets.load_boston()

yb = boston.target.reshape(-1, 1)

Xb = boston['data'][:,5].reshape(-1, 1)
```

```
# Plot data

plt.scatter(Xb,yb)

plt.ylabel('value of house /1000 ($)')

plt.xlabel('number of rooms')

# Create linear regression object

regr = linear_model.LinearRegression()

# Train the model using the training sets

regr.fit( Xb, yb)



# Plot outputs

plt.scatter(Xb, yb,  color='black')

plt.plot(Xb, regr.predict(Xb), color='blue', linewidth=3)

plt.show()
```

Output



Conclusion : Successfully implemented Linear regression in python.

Viva Questions

Q.1 What are the assumptions of a linear regression model?

A.1 The assumptions of a linear regression model are: The relationship between the independent and dependent variables is linear. The residuals, or errors, are normally distributed with a mean of zero and a constant variance. The independent variables are not correlated with each other (i.e. they are not collinear). The residuals are independent of each other (i.e. they are not autocorrelated). The model includes all the relevant independent variables needed to accurately predict the dependent variable.

Q.2 What is the difference between simple and multiple linear regression?

A. 2 Simple linear regression models the relationship between one independent variable and one dependent variable, while multiple linear regression models the relationship between multiple independent variables and one dependent variable. The goal of both methods is to find a linear model that best fits the data and can be used to make predictions about the dependent variable based on the independent variables.

Q. 3 What is the difference between linear regression and logistic regression?

A. 3 Linear regression is a statistical method used for predicting a numerical outcome, such as the price of a house or the likelihood of a person developing a disease. Logistic regression, on the other hand, is used for predicting a binary outcome, such as whether a person will pass or fail a test, or whether a customer will churn or not.

Q. 4 What are the common techniques used to improve the accuracy of a linear regression model?

A.4

i. Feature selection: selecting the most relevant features for the model to improve its predictive power.

ii. Feature scaling: scaling the features to a similar range to prevent bias towards certain features.

iii. Regularization: adding a penalty term to the model to prevent overfitting and improve generalization.

iv. Cross-validation: dividing the data into multiple partitions and using a different partition for validation in each iteration to avoid overfitting.

v. Ensemble methods: combining multiple models to improve the overall accuracy and reduce variance.

Q. 5 What is the concept of overfitting in linear regression?

A.5 Overfitting in linear regression occurs when a model is trained on a limited amount of data and becomes too complex, resulting in poor performance when making predictions on unseen data. This happens because the model has learned to fit the noise or random fluctuations in the training data, rather than the underlying patterns and trends. As a result, the model is not able to generalize well to new data and may produce inaccurate or unreliable predictions. Overfitting can be avoided by using regularization techniques, such as introducing penalty terms to the objective function or using cross-validation to assess the model's performance.

Experiment 3

Aim - Write a Program to demonstrate Logistic Regression.

Theory

Classification techniques are an essential part of machine learning and data mining applications. Approximately 70% of data science problems are classification problems. There are lots of clas- sification problems available, but logistic regression is common and is a useful regression method for solving the binary classification problem. Another category of classification is Multi- nomial classification, which handles the issues where multiple classes are present in the target variable. For example, the IRIS dataset is a very famous example of multi-class classification. Other examples are classifying article/blog/document categories.

Logistic regression can be used for various classification problems, such as spam detection. Some other examples include: diabetes prediction, whether a given customer will purchase a par- ticular product; whether or not a customer will churn, whether the user will click on a given ad- vertisement link or not, and many more examples.

Logistic Regression is one of the most simple and commonly used Machine Learning algorithms for two-class classification. It is easy to implement and can be used as the baseline for any binary classification problem. Its basic fundamental concepts are also constructive in deep learning. Lo- gistic regression describes and estimates the relationship between one dependent binary variable and independent variables.

What is Logistic Regression?

Logistic regression is a statistical method for predicting binary classes. The outcome or target variable is dichotomous in nature. Dichotomous means there are only two possible classes. For example, it can be used for cancer detection problems. It computes the probability of an event occurrence. It is used to estimate discrete values (Binary values like 0/1, yes/no, true/false) based on given set of independent variable(s). In simple words, it predicts the probability of occurrence of an event by fitting data to a logit function. Hence, it is also known as *logit regression*. Since, it predicts the probability, its output values lies between 0 and 1 (as expected). It is a special case of linear regression where the target variable is categorical in nature. It uses a log of odds as the dependent variable. Logistic Regression predicts the probability of occurrence of a binary event utilizing a logit function.

Linear Regression Equation:

$$y = \beta_0 + \beta_1 X_1 + \beta_2 X_2 + \ldots.. + \beta_n X_n$$

Where y is a dependent variable and x1, x2 ... and Xn are explanatory variables.

Sigmoid Function:

$$p = \frac{1}{1 + e^{-y}}$$

Apply Sigmoid function on linear regression:

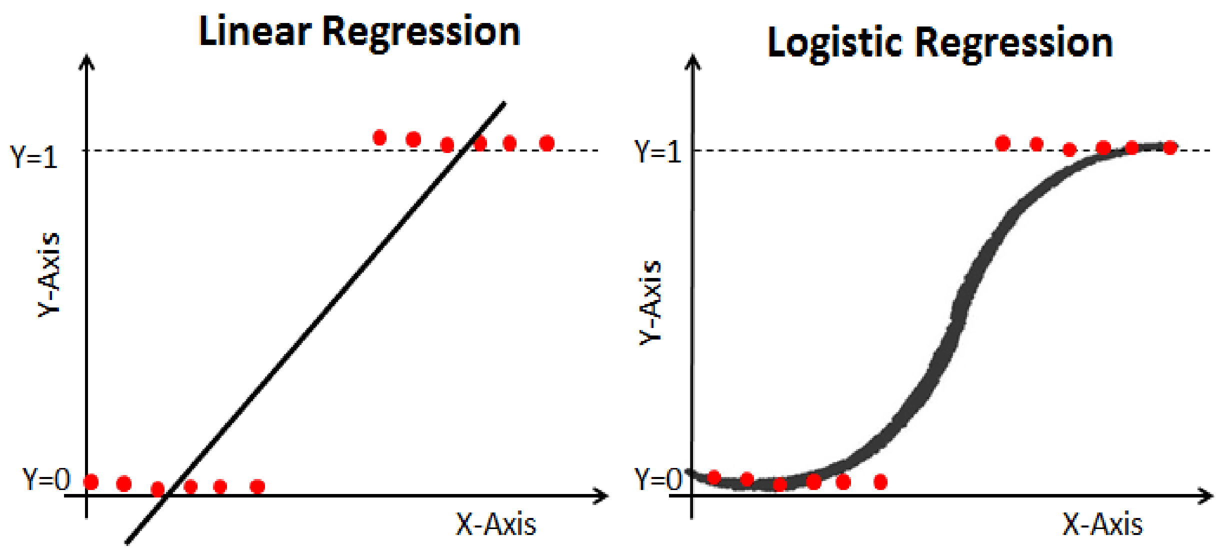
$$p = \frac{1}{1 + e^{-(\beta_0 + \beta_1 X_1 + \beta_2 X_2 + \ldots.. + \beta_n X_n)}}$$

Properties of Logistic Regression:

- The dependent variable in logistic regression follows Bernoulli Distribution.
- Estimation is done through maximum likelihood.
- No R Square, Model fitness is calculated through Concordance, KS-Statistics.

Linear Regression Vs. Logistic Regression

Linear regression gives you a continuous output, but logistic regression provides a constant output. An example of the continuous output is house price and stock price. Examples of the dis- crete output are predicting whether a patient has cancer or not and predicting whether a customer will churn. Logistic regression is estimated using the maximum likelihood estimation (MLE) approach, while linear regression is typically estimated using ordinary least squares (OLS), which can also be considered a special case of MLE when the errors in the model are normally distributed.



Maximum Likelihood Estimation Vs. Least Squares Method

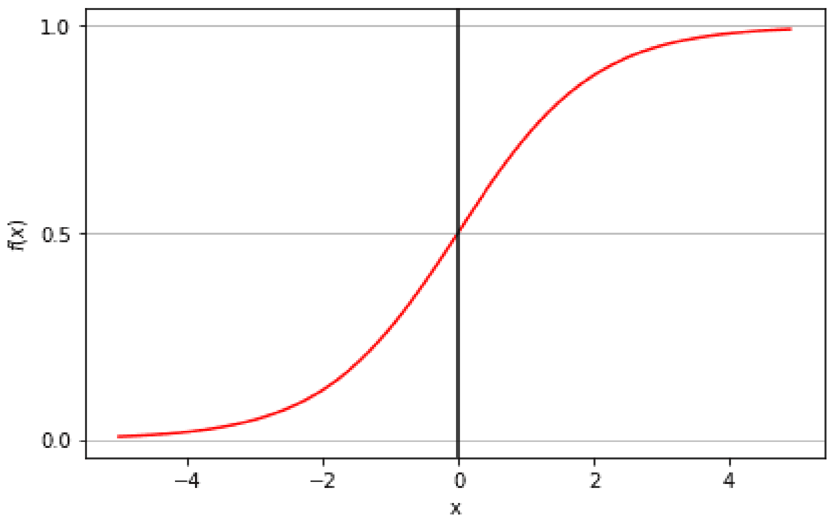
The MLE is a "likelihood" maximization method, while OLS is a distance-minimizing approximation method. Maximizing the likelihood function determines the parameters that are most likely to produce the observed data. From a statistical point of view, MLE sets the mean and variance as parameters in determining the specific parametric values for a given model. This set of parameters can be used for predicting the data needed in a normal distribution.

Ordinary Least squares estimates are computed by fitting a regression line on given data points that has the minimum sum of the squared deviations (least square error). Both are used to estimate the parameters of a linear regression model. MLE assumes a joint probability mass func- tion, while OLS doesn't require any stochastic assumptions for minimizing distance.

Sigmoid function

The sigmoid function, also called logistic function, gives an ‘S’ shaped curve that can take any real-valued number and map it into a value between 0 and 1. If the curve goes to positive infinity, y predicted will become 1, and if the curve goes to negative infinity, y predicted will become 0. If the output of the sigmoid function is more than 0.5, we can classify the outcome as 1 or YES, and if it is less than 0.5, we can classify it as 0 or NO. For example, if the output is 0.75, we can say in terms of the probability that there is a 75 percent chance that a patient will suffer from cancer.

$$f(x) = \frac{1}{1 + e^{-x}}$$



Types of Logistic Regression

Types of Logistic Regression:

- **Binary Logistic Regression**: The target variable has only two possible outcomes such as Spam or Not Spam, Cancer or No Cancer.
- **Multinomial Logistic Regression**: The target variable has three or more nominal categories, such as predicting the type of Wine.
- **Ordinal Logistic Regression**: the target variable has three or more ordinal categories, such as restaurant or product rating from 1 to 5.

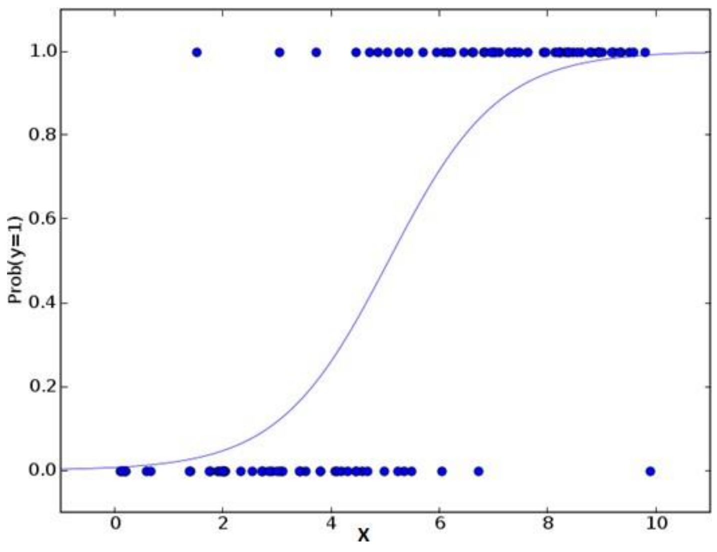
Python Code flow

```
#Import Library

from sklearn.linear_model import LogisticRegression

#Assumed you have, X (predictor) and Y (target) for training data set and
x_test(predictor) of test_dataset

# Create logistic regression object

model = LogisticRegression()

# Train the model using the training sets and check score

model.fit(X, y)
model.score(X, y)


#Equation coefficient and Intercept

print('Coefficient: \n', model.coef_)

print('Intercept: \n', model.intercept_)

#Predict Output

predicted= model.predict(x_test)
```

Python Implementation

```
import numpy as np

from sklearn.linear_model import LogisticRegression

from sklearn.model_selection import train_test_split

from sklearn.metrics import accuracy_score
```

```
# Sample data

X = np.array([[1, 2], [2, 3], [3, 1], [4, 3], [5, 3]])

y = np.array([0, 0, 0, 1, 1])




# 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)


# Create and fit the model

model = LogisticRegression()

model.fit(X_train, y_train)

# Predict the output

y_pred = model.predict(X_test) # Calculate accuracy

accuracy = accuracy_score(y_test, y_pred)

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



Output

```
Accuracy: 1.0
```

Python Implementation (Using Iris Dataset) #

```
Import necessary libraries

import pandas as pd

from sklearn.model_selection import train_test_split

from sklearn.linear_model import LogisticRegression

from sklearn.metrics import accuracy_score
```

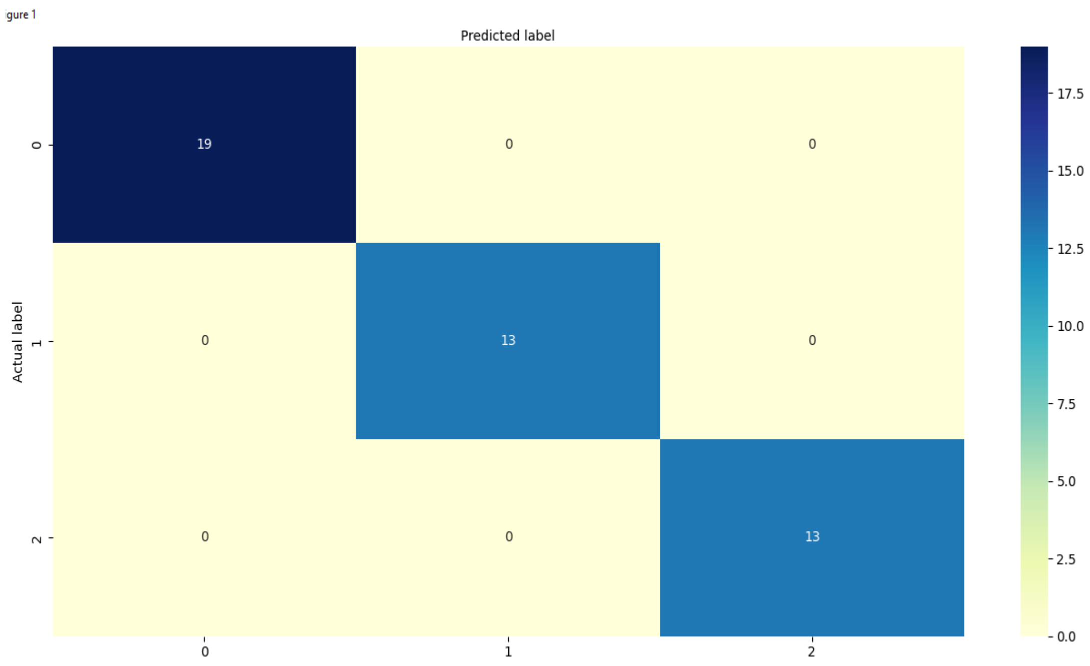
```
# Load the dataset

#col_names    =    ['sepal_length',    'sepal_width',    'petal_length',
'petal_width', 'species']

dataset = pd.read_csv('IRIS.csv')

# The output class is in the categorical form in this data set,

# and we need to convert it into the numeric format. So We will use
Label Encoder.

from sklearn.preprocessing import LabelEncoder

le = LabelEncoder()

dataset['species'] = le.fit_transform(dataset['species'])

print (dataset.head(100)) # Select only first 100 rows

# Split dataset into features and target variable

feature_cols    =    ['sepal_length',    'sepal_width',    'petal_length',
'petal_width']

#X = dataset.iloc[:, :-1]

#y = dataset.iloc[:, -1]

X = dataset[feature_cols]

Y = dataset.species

# Split dataset into training set and test set

X_train,    X_test,    y_train,    y_test    =    train_test_split(X,    Y,
test_size=0.3, random_state=42)



# Initialize the model

model = LogisticRegression()
```

```
# Train the model


model.fit(X_train, y_train)



# Predict the response for test dataset

y_pred = model.predict(X_test)



# Evaluate accuracy

print("Accuracy:", accuracy_score(y_test, y_pred)*100)



#Model Evaluation using Confusion Matrix

#A confusion matrix is a table that is used to evaluate the
performance of a classification model.

# You can also visualize the performance of an algorithm.

# The fundamental part of a confusion matrix is the number of correct
and incorrect predictions summed up class-wise.



# import the metrics class

from sklearn import metrics

cnf_matrix = metrics.confusion_matrix(y_test, y_pred)

print(cnf_matrix)

#Diagonal values represent accurate predictions, while non-diagonal
elements are inaccurate predictions.



#Visualizing confusion matrix using a heatmap
```

```
#Let's visualize the results of the model in the form of a confusion
matrix using matplotlib and seaborn.

#Here, you will visualize the confusion matrix using Heatmap.

# import required modules

import numpy as np

import seaborn as sns

class_names=[0,1] # name  of classes

fig, ax = plt.subplots()

tick_marks = np.arange(len(class_names))

plt.xticks(tick_marks, class_names)

plt.yticks(tick_marks, class_names)

# create heatmap

sns.heatmap(pd.DataFrame(cnf_matrix),      annot=True,     cmap="YlGnBu"
,fmt='g')

ax.xaxis.set_label_position("top")

plt.tight_layout()

plt.title('Confusion matrix', y=1.1)

plt.ylabel('Actual label')

plt.xlabel('Predicted label')

plt.show()
```

Output

```
            sepal_length  sepal_width  petal_length   petal_width  species
0             5.1            3.5       1.4                0.2         0
1             4.9            3.0       1.4                0.2         0
2             4.7            3.2       1.3                0.2         0
3             4.6            3.1       1.5                0.2         0
4             5.0            3.6       1.4                0.2         0
..            ...            ...       ...                ...       ...
95            5.7            3.0       4.2                1.2         1
96            5.7            2.9       4.2                1.3         1
97            6.2            2.9       4.3                1.3         1

98       5.1          2.5           3.0           1.1        1
99       5.7          2.8           4.1           1.3        1

[100 rows x 5 columns]


Accuracy: 100.0

[[19 0  0]
[ 0  13 0]
[ 0  0  13]]
```



```
Process finished with exit code 0
```

Conclusion: Successfully implemented Logistic regression using Python.

Viva Questions

Q.1 Can you use logistic regression for classification between more than two classes?

A.1 Yes, it is possible to use logistic regression for classification between more than two classes, and it is called multinomial logistic regression (e.g. SoftMax). However, this is not possible to implement without modifications to the conventional logistic regression model.

Q. 2 Why can't we use the mean square error cost function used in linear regression for logistic regression?

A. 2 If we use mean square error in logistic regression, the resultant cost function will be non-convex, i.e., a function with many local minima, owing to the presence of the sigmoid function in h(x). As a result, an attempt to find the parameters using gradient descent may fail to optimize cost function properly. It may end up choosing a local minima instead of the actual global minima.

Q. 3 If you observe that the cost function decreases rapidly before increasing or stagnating at a specific high value, what could you infer?

A. 3 A trend pattern of the cost curve exhibiting a rapid decrease before then increasing or stagnating at a specific high value indicates that the learning rate is too high. The gradient descent is bouncing around the global minimum but missing it owing to the larger than necessary step size.

Q. 4 How do you decide the cut-off for the output of logistic regression?

A.4 The cut-off is decided such that the accuracy is maximum. Confusion matrix is used here; true negative (actual = 0 and predicted = 0), false negative (actual = 1 and predicted = 0), false positive (actual = 0 and predicted = 1), true positive (actual = 1 and predicted = 1).

Q. 5 What is the importance of regularisation?

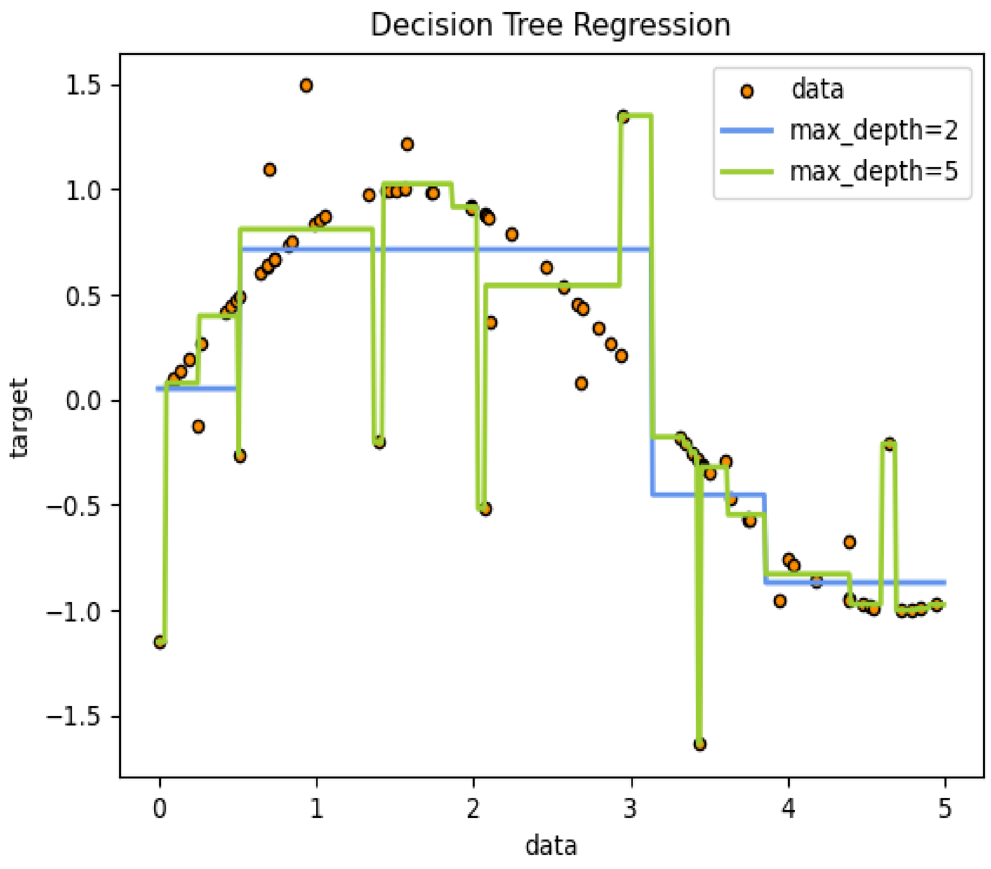
A.5 Regularisation is a technique that can help alleviate the problem of overfitting a model. It is beneficial when a large number of parameters are present, which help predict the target function. In these circumstances, it is difficult to select which features to keep manually.Regularisation essentially involves adding coefficient terms to the cost function so that the terms are penalized and are small in magnitude. This helps, in turn, to preserve the overall trends in the data while not letting the model become too complex. These penalties, in effect, restrict the influence a predictor variable can have over the target by compressing the coefficients, thereby preventing overfitting.

Experiment 4

Aim: Write a Program to demonstrate Decision Tree ID3 Algorithm.

Theory

Decision Trees (DTs) are a non-parametric supervised learning method used for *classification* and *regression*. The goal is to create a model that predicts the value of a target variable by learning simple decision rules inferred from the data features. A tree can be seen as a piecewise constant approximation. For instance, in the example below, decision trees learn from data to approximate a sine curve with a set of if-then-else decision rules. The deeper the tree, the more complex the decision rules and the fitter the model.



Some advantages of decision trees are:

- Simple to understand and to interpret. Trees can be visualized.

- Requires little data preparation. Other techniques often require data normalization, dummy variables need to be created and blank values to be removed. Some tree and algo- rithm combinations support missing values.
- The cost of using the tree (i.e., predicting data) is logarithmic in the number of data points used to train the tree.
- Able to handle both numerical and categorical data. However, the scikit-learn implementation does not support categorical variables for now. Other techniques are usually specialized in analyzing datasets that have only one type of variable. See algorithms for more information.
- Able to handle multi-output problems.
- Uses a white box model. If a given situation is observable in a model, the explanation for the condition is easily explained by boolean logic. By contrast, in a black box model (e.g., in an artificial neural network), results may be more difficult to interpret.
- Possible to validate a model using statistical tests. That makes it possible to account for the reliability of the model.
- Performs well even if its assumptions are somewhat violated by the true model from which the data were generated.

The disadvantages of decision trees include:

- Decision-tree learners can create over-complex trees that do not generalize the data well. This is called overfitting. Mechanisms such as pruning, setting the minimum number of samples required at a leaf node or setting the maximum depth of the tree are necessary to avoid this problem.
- Decision trees can be unstable because small variations in the data might result in a completely different tree being generated. This problem is mitigated by using decision trees within an ensemble.
- Predictions of decision trees are neither smooth nor continuous, but piecewise constant approximations as seen in the above figure. Therefore, they are not good at extrapolation.
- The problem of learning an optimal decision tree is known to be NP-complete under several aspects of optimality and even for simple concepts. Consequently, practical decision-tree learning algorithms are based on heuristic algorithms such as the greedy algorithm where locally optimal decisions are made at each node. Such algorithms cannot guarantee to return the globally optimal decision tree. This can be mitigated by training multiple trees in an ensemble learner, where the features and samples are randomly sampled with replacement.
- There are concepts that are hard to learn because decision trees do not express them easily, such as XOR, parity or multiplexer problems.
- Decision tree learners create biased trees if some classes dominate. It is therefore recommended to balance the dataset prior to fitting with the decision tree.

Classification - DecisionTreeClassifier is a class capable of performing multi-class classification on a dataset.

Iterative Dichotomiser3 Algorithm - ID3 (Iterative Dichotomiser3 Algorithm) is used in machine learning for building decision trees from a given dataset. It was developed in 1986 by Ross Quinlan. It is a greedy algorithm that builds a decision tree by recursively partitioning the data set into smaller and smaller subsets until all data points in each subset belong to the same class. It employs a top-down approach, recursively selecting features to split the dataset based on information gain. The ID3 algorithm is a classic decision tree algorithm used for both classification and regression tasks.ID3 deals primarily with categorical properties, which means that it can efficiently handle objects with a discrete set of values. This property is consistent with its suitability for problems where the input features are categorical rather than continuous. One of the strengths of ID3 is its ability to generate interpretable decision trees. The resulting tree structure is easily understood and visualized, providing insight into the deci- sion-making process. However, ID3 can be sensitive to noisy data and prone to overfitting, capturing details in the training data that may not adequately account for new unseen data.

How ID3 Algorithms work?

The ID3 algorithm works by building a decision tree, which is a hierarchical structure that classifies data points into different categories and splits the dataset into smaller subsets based on the values of the features in the dataset. The ID3 algorithm then selects the feature that provides the most information about the target variable. The decision tree is built top-down, start- ing with the root node, which represents the entire dataset. At each node, the ID3 algorithm selects the attribute that provides the most information gain about the target variable. The at- tribute with the highest information gain is the one that best separates the data points into dif- ferent categories.

ID3 metrices

The ID3 algorithm utilizes metrics related to information theory, particularly entropy and information gain, to make decisions during the tree-building process.

Information Gain and Attribute Selection

The ID3 algorithm uses a measure of impurity, such as entropy or Gini impurity, to calculate the *information gain* of each attribute. *Entropy* is a measure of disorder in a dataset. A dataset with high entropy is a dataset where the data points are evenly distributed across the different categories. A dataset with low entropy is a dataset where the data points are concentrated in one or a few categories.

$H(S)=\Sigma-(Pi*\log2(Pi))$

- where, Pi represents the fraction of the sample within a particular node.
- S – The current dataset.
- i – Set of classes in S

If entropy is low, data is well understood; if high, more information is needed. Preprocessing data before using ID3 can enhance accuracy. In sum, ID3 seeks to reduce uncertainty and make informed decisions by picking attributes that offer the most insight in a dataset.

Information gain assesses how much valuable information an attribute can provide. We select the attribute with the highest information gain, which signifies its potential to contribute the

most to understanding the data. If information gain is high, it implies that the attribute offers a significant insight. ID3 acts like an investigator, making choices that maximize the information gain in each step. This approach aims to minimize uncertainty and make well-informed deci- sions, which can be further enhanced by preprocessing the data.
$IG(A,D)=H(S)-\sum_{v} |Sv|/|S|\times H(Sv)$

- where, |S| is the total number of instances in dataset.
- $|Sv|$ is the number of instances in dataset for which attribute D has values v.
- $H(S)$ is the entropy of dataset.

Steps in ID3 algorithm -
1. Determine entropy for the overall the dataset using class distribution.
2. For each feature.
   - Calculate Entropy for Categorical Values.
   - Assess information gain for each unique categorical value of the feature.
3. Choose the feature that generates highest information gain.
4. Iteratively apply all above steps to build the decision tree structure.

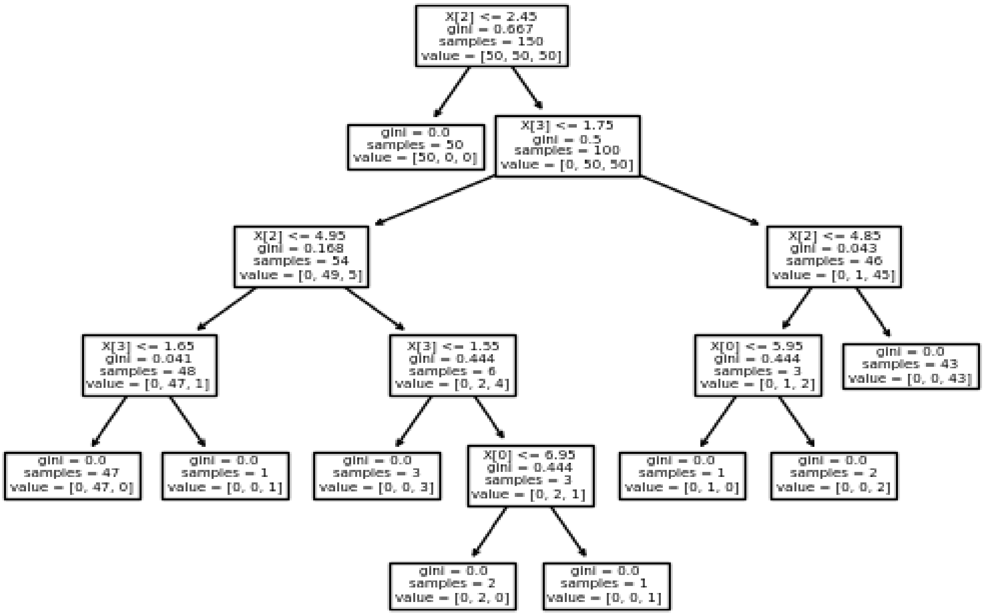
Python Script

```
# As with other classifiers, DecisionTreeClassifier takes as input two
arrays: an array X, sparse or dense, of shape (n_samples, n_features)

# holding the training samples, and an array Y of integer values,
shape (n_samples,), holding the class labels for the training samples:

from sklearn import tree

X = [[0, 0], [1, 1]]

Y = [0, 1]

clf = tree.DecisionTreeClassifier()

clf = clf.fit(X, Y)


#After being fitted, the model can then be used to predict the class
of samples:


print(clf.predict([[2., 2.]]))


#In case that there are multiple classes with the same and highest
probability,
# the classifier will predict the class with the lowest index amongst
those classes.

#As an alternative to outputting a specific class, the probability of
each class can be predicted,

# which is the fraction of training samples of the class in a leaf:



print(clf.predict_proba([[2., 2.]]))



#DecisionTreeClassifier is capable of both binary (where the labels
are [-1, 1]) classification and multiclass

# (where the labels are [0, …, K-1]) classification.

#Using the Iris dataset, we can construct a tree as follows:



from sklearn.datasets import load_iris

from sklearn import tree

import matplotlib.pyplot as plt

iris = load_iris()

X, y = iris.data, iris.target

clf = tree.DecisionTreeClassifier()

clf = clf.fit(X, y)


#Once trained, you can plot the tree with the plot_tree function:

tree.plot_tree(clf)

plt.show()
```

```
#Decision trees can also be applied to regression problems, using the
DecisionTreeRegressor class.

#As in the classification setting, the fit method will take as
argument arrays X and y,


# only that in this case y is expected to have floating point values
instead of integer values:



from sklearn import tree

X = [[0, 0], [2, 2]]

y = [0.5, 2.5]

clf = tree.DecisionTreeRegressor()

clf = clf.fit(X, y)

pred_value = clf.predict([[1, 1]])

print(pred_value)
```

Output

```
[1]

[[0. 1.]

[0.5]
```


X[2] <= 2.45
gini = 0.667
samples = 150
value = [50, 50, 50]
gini = 0.0
samples = 50
value = [50, 0, 0]
X[3] <= 1.75
gini = 0.5
samples = 100
value = [0, 50, 50]
X[2] <= 4.95
gini = 0.168
samples = 54
value = [0, 49, 5]
X[2] <= 4.85
gini = 0.043
samples = 46
value = [0, 1, 45]
X[3] <= 1.65
gini = 0.041
samples = 48
value = [0, 47, 1]
X[3] <= 1.55
gini = 0.444
samples = 6
value = [0, 2, 4]
X[0] <= 5.95
gini = 0.444
samples = 3
value = [0, 1, 2]
gini = 0.0
samples = 43
value = [0, 0, 43]
gini = 0.0
samples = 47
value = [0, 47, 0]
gini = 0.0
samples = 1
value = [0, 0, 1]
gini = 0.0
samples = 3
value = [0, 0, 3]
X[0] <= 6.95
gini = 0.444
samples = 3
value = [0, 2, 1]
gini = 0.0
samples = 1
value = [0, 1, 0]
gini = 0.0
samples = 2
value = [0, 0, 2]
gini = 0.0
samples = 2
value = [0, 2, 0]
gini = 0.0
samples = 1
value = [0, 0, 1]

Conclusion: Successfully implemented Decision Tree ID3 algorithm using Python.

Viva Questions

Q.1 What is the Decision Tree Algorithm?

A.1 It works by recursively partitioning the data into subsets based on the values of the features and the target variable. The goal is to create a tree-like model that represents all possible decisions and their possible consequences, which can be used to predict the target variable for a new instance. Each node in the tree represents a feature, and each branch represents a decision based on the value of that feature. The end points of the branches are the final decisions, represented by leaf nodes. The leaf nodes contain the class label for the instances in the subset of the data represented by that leaf. The tree is built top-down, starting from the root node and splitting the data based on the feature that provides the highest reduction in impurity, as determined by a metric such as information gain or gini impurity.

Q.2 Are Decision Trees affected by the outliers? Explain.

A. 2 Outliers, or extreme values that are far from the majority of the data, can have an impact on Decision Trees. The way outliers affect Decision Trees depends on how the tree is built and how the splits are made. In general, Decision Trees are sensitive to outliers in the sense that outliers can influence the location of the splits in the tree, and thus affect the final predictions. For example, if an outlier is present in a feature, it can dominate the split at the root node of the tree, leading to a biased tree structure that is not representative of the majority of the data.

Q. 3 How can the impact of Outliers be mitigated using Decision trees?

A. 3 The impact of outliers can be mitigated to some extent by using appropriate measures for impurity such as Gini impurity or information gain, which take into account the frequency of the different classes and are less sensitive to extreme values. Additionally, pruning the tree or using an ensemble of Decision Trees can help to reduce the impact of outliers, as the final prediction is based on a combination of multiple trees, rather than just one.

Q. 4 How would you define the Stopping Criteria for decision trees?

A.4 Stopping criteria for decision trees refer to the rules or conditions that determine when to stop building the tree and create the final model. These rules are used to prevent overfitting, which occurs when a decision tree is too complex and fits the training data too closely, resulting in poor generalization to new data. By using appropriate stopping criteria, we can create decision trees that are simple and interpretable, and that generalize well to new data. Following are the stopping criteria:

i. Maximum depth: This limits the depth or the number of levels in the tree. Once the tree reaches the maximum depth, no further splits are made, and the node becomes a leaf node. For e.g., we can set a maximum depth of 3 for a decision tree that is being used to classify whether a person will buy a particular product. If the tree reaches a depth of 3 and there are no further improvements in classification accuracy, we stop building the tree and use the final model.

ii. Minimum number of samples: This sets a threshold for the minimum number of samples required to split a node. If a node has fewer samples than the specified threshold, it is not split, and the node becomes a leaf node. For e.g., we can set a minimum number of samples of 10 for a decision tree that is being used to predict whether a student will pass a particular exam. If a node has fewer than 10 samples, we stop splitting the node and use the majority class as the prediction.

iii. Minimum impurity decrease: This sets a threshold for the minimum amount of impurity decrease required to split a node. If the impurity decrease from splitting a node is below the specified threshold, the node is not split, and it becomes a leaf node. For e.g., we can set a minimum impurity decrease of 0.01 for a decision tree that is being used to classify whether a patient has a particular disease based on their medical history. If a split does not result in an impurity decrease of at least 0.01, we stop splitting the node and use the majority class as the prediction.

iv. Maximum number of leaf nodes: This sets a limit on the maximum number of leaf nodes in the tree. Once this limit is reached, no further splits are made.

Q. 5 How would you deal with an Overfitted Decision Tree?

A.5 An overfitted Decision Tree is a tree that is too complex and fits the training data too closely, but may not generalize well to new, unseen data. Overfitting occurs when the tree is too large and has too many branches, leading to a model that captures noise and random fluctuations in the training data instead of the underlying patterns. To deal with an overfitted Decision Tree, we can use one or more of the following methods:

i. Pruning: Pruning is the process of removing branches of the tree that contribute little to the accuracy of the model, in order to simplify the tree and reduce overfitting. One common pruning method is reduced error pruning, where we remove branches of the tree and evaluate the impact on the validation set.

ii. Limiting tree depth: Limiting the maximum depth of the tree can prevent overfitting by reducing the number of branches and ensuring that the tree is not too complex.

iii. Using ensemble methods: Ensemble methods, such as Random Forest or Gradient Boosting, can reduce overfitting by combining multiple Decision Trees, each of which has a lower chance of overfitting the data.

iv. Early stopping: In the case of decision trees built using a greedy algorithm, such as ID3 or C4.5, you can use early stopping to stop the tree-growing process when the accuracy on a validation set starts to decrease, indicating that the tree is overfitting.

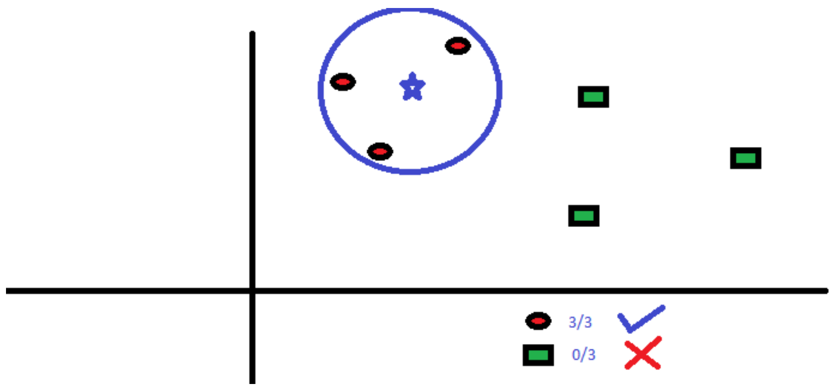
v. Feature selection: Overfitting can also be reduced by removing features that have low importance or high correlation with other features, as this reduces the number of branches and makes the tree simpler.

Experiment 5

Aim: Write a Program to demonstrate k-Nearest Neighbor flowers (Iris) classification.

Theory

K nearest neighbours (KNN) is a simple algorithm that stores all available cases and classifies new cases based on a similarity measure (e.g., distance functions). KNN has been used in statistical estimation and pattern recognition already in the beginning of 1970"s as a non- parametric technique. It can be used for both classification and regression problems. However, it is more widely used in classification problems in the industry. K nearest neighbors is a simple algorithm that stores all available cases and classifies new cases by a majority vote of its k neighbors. The case is assigned to the class is most common amongst its K nearest neighbors measured by a distance function. These distance functions can be Euclidean, Manhattan, Minkowski and Hamming distance. First three functions are used for continuous function and fourth one (Hamming) for categorical variables. If K = 1, then the case is simply assigned to the class of its nearest neighbor. At times, choosing K turns out to be a challenge while performing kNN modeling.



KNN can easily be mapped to our real lives. If you want to learn about a person, of whom you have no information, you might like to find out about his close friends and the circles he moves in and gain access to his/her information!

Things to consider before selecting kNN:

- KNN is computationally expensive
- Variables should be normalized else higher range variables can bias it
- Works on pre-processing stage more before going for KNN like outlier, noise removal

Algorithm: A case is classified by a majority vote of its neighbors, with the case being assigned to the class most common amongst its K nearest neighbors measured by a distance function. If K = 1, then the case is simply assigned to the class of its nearest neighbor.

Distance functions

Euclidean $$\sqrt{\sum_{i=1}^{k}(x_i - y_i)^2}$$

Manhattan $$\sum_{i=1}^{k}|x_i - y_i|$$

Minkowski $$\left(\sum_{i=1}^{k}(|x_i - y_i|)^q\right)^{1/q}$$

It should also be noted that all three distance measures are only valid for continuous variables. In the instance of categorical variables, the Hamming distance must be used. It also brings up the issue of standardization of the numerical variables between 0 and 1 when there is a mixture of numerical and categorical variables in the dataset.

Hamming Distance

$$D_H = \sum_{i=1}^{k}|x_i - y_i|$$

$$x = y \Rightarrow D = 0$$
$$x \neq y \Rightarrow D = 1$$

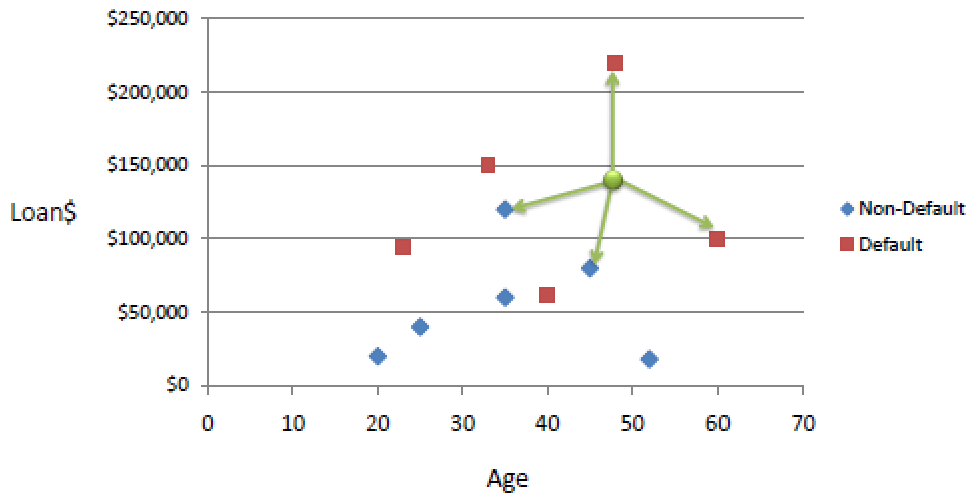
X	Y	Distance
Male	Male	0
Male	Female	1

Choosing the optimal value for K is best done by first inspecting the data. In general, a large K value is more precise as it reduces the overall noise but there is no guarantee. Cross-validation is another way to retrospectively determine a good K value by using an independent dataset to validate the K value.

Historically, the optimal K for most datasets has been between 3-10. That produces much better results than 1NN

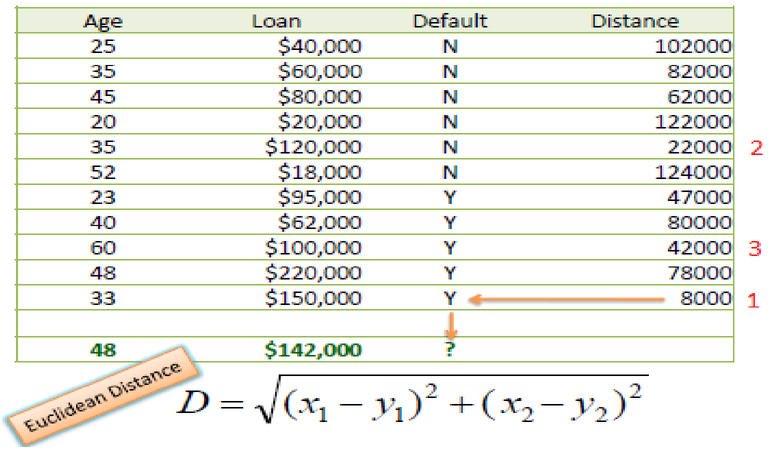
Example:

Consider the following data concerning credit default. Age and Loan are two numerical variables (predictors) and Default is the target.



We can now use the training set to classify an unknown case (Age=48 and Loan=$142,000) using Euclidean distance. If K=1 then the nearest neighbor is the last case in the training set with Default=Y.

D = Sqrt[(48-33)^2 + (142000-150000)^2] = 8000.01 >> Default=Y



Age	Loan	Default	Distance	
25	$40,000	N	102000	
35	$60,000	N	82000	
45	$80,000	N	62000	
20	$20,000	N	122000	
35	$120,000	N	22000	2
52	$18,000	N	124000	
23	$95,000	Y	47000	
40	$62,000	Y	80000	
60	$100,000	Y	42000	3
48	$220,000	Y	78000	
33	$150,000	Y	8000	1
48	**$142,000**	?		

Euclidean Distance

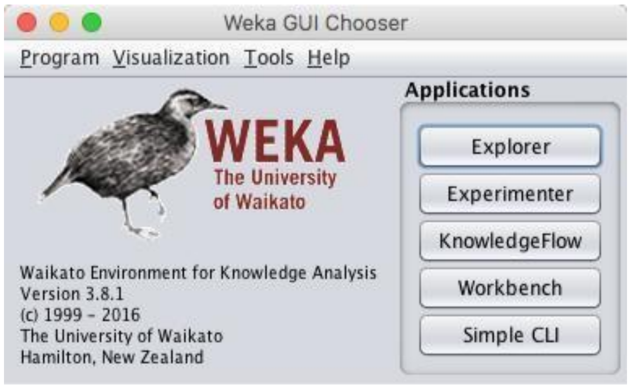
$$D = \sqrt{(x_1 - y_1)^2 + (x_2 - y_2)^2}$$

Fig-1

With K=3, there are two Default=Y and one Default=N out of three closest neighbors. The prediction for the unknown case is again Default=Y.

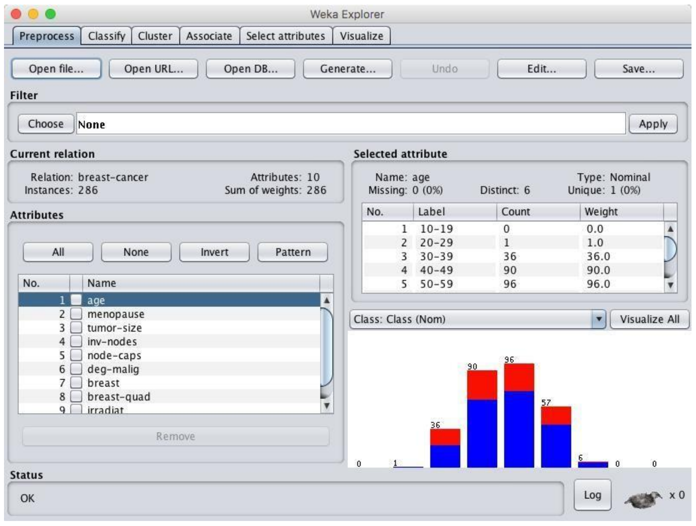
Sample Input:

Step 1: Open WEKA GUI and Select Explorer



Waikato Environment for Knowledge Analysis
Version 3.8.1
(c) 1999 - 2016
The University of Waikato
Hamilton, New Zealand

Step 2: Load the Breast Cancer data set (breast-cancer.arff) given in experiment 4 databases, using "Open File"



Filter: None

Current relation
Relation: breast-cancer   Attributes: 10
Instances: 286   Sum of weights: 286

Selected attribute
Name: age   Type: Nominal
Missing: 0 (0%)   Distinct: 6   Unique: 1 (0%)

No.	Label	Count	Weight
1	10-19	0	0.0
2	20-29	1	1.0
3	30-39	36	36.0
4	40-49	90	90.0
5	50-59	96	96.0

Attributes

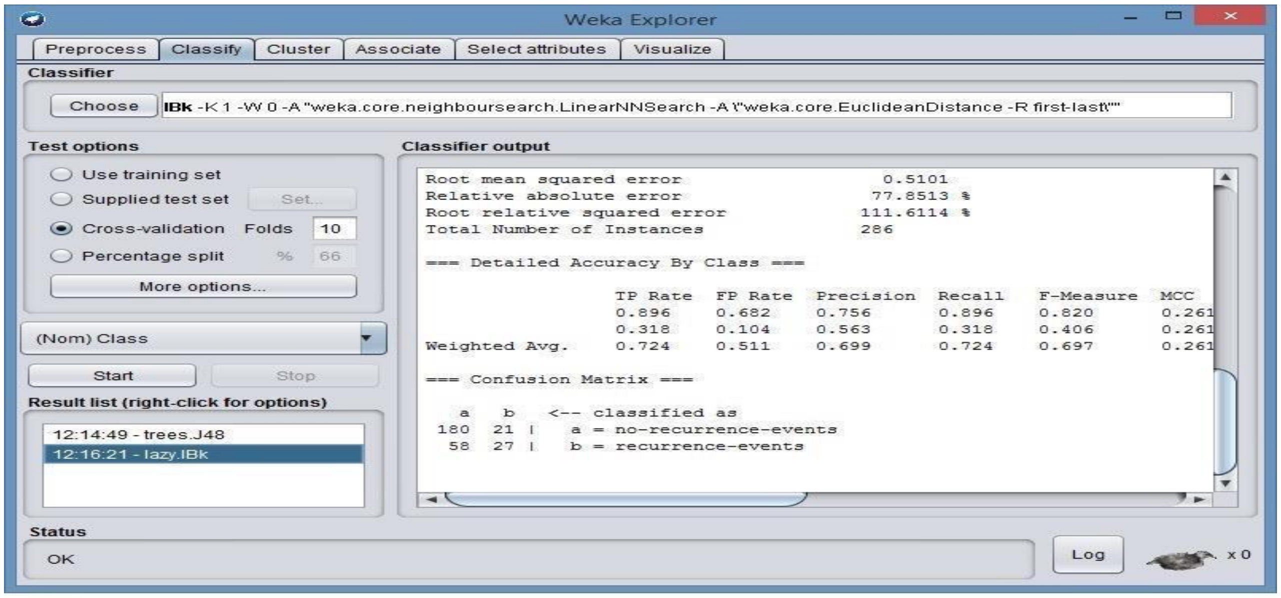
No.	Name
1	age
2	menopause
3	tumor-size
4	inv-nodes
5	node-caps
6	deg-malig
7	breast
8	breast-quad
9	irradiat

Class: Class (Nom)

Status
OK

Step 3: Choose the "Classify" Tab and Choose The Decision Tree Classifier labeled as *J48* in the *trees* folder and choose "Cross-Validation" and enter *5* in the Folds, then press *Start*

Sample Output:



IBk -K 1 -W 0 -A "weka.core.neighboursearch.LinearNNSearch -A \"weka.core.EuclideanDistance -R first-last\""

Cross-validation Folds 10

Classifier output

```
Root mean squared error                  0.5101
Relative absolute error                 77.8513 %
Root relative squared error            111.6114 %
Total Number of Instances              286

=== Detailed Accuracy By Class ===

                 TP Rate  FP Rate  Precision  Recall   F-Measure  MCC
                 0.896    0.682    0.756      0.896    0.820      0.261
                 0.318    0.104    0.563      0.318    0.406      0.261
Weighted Avg.    0.724    0.511    0.699      0.724    0.697      0.261

=== Confusion Matrix ===

   a   b   <-- classified as
 180  21 |   a = no-recurrence-events
  58  27 |   b = recurrence-events
```

<u>PYTHON CODE FLOW</u>

```
#IMPORT LIBRARY

FROM SKLEARN.NEIGHBORS IMPORT KNEIGHBORSCLASSIFIER

#ASSUMED YOU HAVE, X (PREDICTOR) AND Y (TARGET) FOR TRAINING DATA SET AND
X_TEST(PREDICTOR) OF TEST_DATASET

# CREATE KNEIGHBORS CLASSIFIER OBJECT MODEL

KNEIGHBORSCLASSIFIER(N_NEIGHBORS=6) # DEFAULT VALUE FOR N_NEIGHBORS IS 5

# TRAIN THE MODEL USING THE TRAINING SETS AND CHECK SCORE

MODEL.FIT(X, Y)

#PREDICT OUTPUT

PREDICTED= MODEL.PREDICT(X_TEST)
```

Python Implementation

```
import numpy as np

from sklearn.neighbors import KNeighborsClassifier

from sklearn.model_selection import train_test_split

from sklearn.metrics import accuracy_score


# Sample data

X = np.array([[3, 5], [4, 7], [6, 3], [7, 4], [8, 2], [9, 6]])

y = np.array([0, 0, 1, 1, 1, 0])


# Split the data into training and testing sets

X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.25, random_state=0)


# Create and fit the model

model = KNeighborsClassifier(n_neighbors=3) # Set the number of neighbors

model.fit(X_train, y_train)
```

```
# Predict the output
y_pred = model.predict(X_test)


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

Output

```
Accuracy: 0.5
```

Python Implementation (Using Iris dataset)

```
#  Import libraries
import pandas as pd

#from sklearn import datasets
from sklearn.model_selection import train_test_split


# Load data
#dataset = datasets.load_iris()


#col_names = ['sepal_length', 'sepal_width', 'petal_length', 'petal_width',
'species']
dataset = pd.read_csv('IRIS.csv')


# dataset = dataset.drop(columns = ['s.no.']) # drop s.no. column which is not
#required
```

```
# Let’s have the information about the data type of the data set.

print(dataset.info())

print(dataset.describe())


## Let’s now take a look at the number of instances (rows) that belong to each

#class. We can view this as an absolute count.

#print(dataset.groupby('species').size())

print(dataset['species'].value_counts())


# The output class is in the categorical form in this data set,

# and we need to convert it into the numeric format. So We will use Label Encoder.

from sklearn.preprocessing import LabelEncoder

le = LabelEncoder()

dataset['species'] = le.fit_transform(dataset['species'])

print (dataset.head(100)) # Select only first 100 rows


# Split dataset into features and target variable

feature_cols = ['sepal_length', 'sepal_width', 'petal_length', 'petal_width']

#X = dataset.iloc[:, :-1]

#y = dataset.iloc[:, -1]

X = dataset[feature_cols]

Y = dataset.species


# Split dataset into training set and test set

X_train, X_test, y_train, y_test = train_test_split(X, Y, test_size=0.3,
random_state=42)
```

```
#Using KNN for classification
# Making predictions
# Fitting clasifier to the Training set


# Loading libraries
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import confusion_matrix, accuracy_score


# Instantiate learning model (k = 3)
classifier = KNeighborsClassifier(n_neighbors=3)


# Fitting the model
classifier.fit(X_train, y_train)


# Predicting the Test set results
y_pred = classifier.predict(X_test)

####Evaluating predictions######


#Building confusion matrix:
cm = confusion_matrix(y_test, y_pred)
print(cm)

#Calculating model accuracy:
```

```
accuracy = accuracy_score(y_test, y_pred)*100

print('Accuracy of our model is equal ' + str(round(accuracy, 2)) + ' %.')
```

Output

```
<class 'pandas.core.frame.DataFrame'>

RangeIndex: 150 entries, 0 to 149 Data

columns (total 5 columns):

 #   Column         Non-Null Count  Dtype
---  ---------      --------------  -------

0    sepal_length   150  non-null    float64

1    sepal_width    150  non-null    float64

2    petal_length   150  non-null    float64

3    petal_width    150  non-null    float64

4    species        150  non-null    object

dtypes: float64(4), object(1)

memory usage: 6.0+ KB

None

       sepal_length  sepal_width petal_length  petal_width

count    150.000000   150.000000 150.000000    150.000000

mean       5.843333     3.054000 3.758667      1.198667

std        0.828066     0.433594 1.764420      0.763161

min        4.300000     2.000000 1.000000      0.100000

25%        5.100000     2.800000 1.600000      0.300000

50%        5.800000     3.000000 4.350000      1.300000

75%        6.400000     3.300000 5.100000      1.800000

max        7.900000     4.400000 6.900000      2.500000

Iris-setosa        50

Iris-versicolor    50

Iris-virginica     50

Name: species, dtype: int64
```

```
    sepal_length  sepal_width  petal_length  petal_width  species

0              5.1          3.5            1.4          0.2        0

1              4.9          3.0            1.4          0.2        0

2              4.7          3.2            1.3          0.2        0

3              4.6          3.1            1.5          0.2        0

4              5.0          3.6            1.4          0.2        0

..             ...          ...            ...          ...        .
                                                                   .
                                                                   .
95             5.7          3.0            4.2          1.2        1

96             5.7          2.9            4.2          1.3        1

97             6.2          2.9            4.3          1.3        1

98             5.1          2.5            3.0          1.1        1

99             5.7          2.8            4.1          1.3        1


[100 rows x 5 columns]

[[19  0  0]

 [ 0 13  0]

 [ 0  0 13]]

Accuracy of our model is equal 100.0 %.
```

Conclusion: Successfully implemented kNN algorithm on iris dataset.

Viva Questions

Q1: How is the KNN algorithm different from K means Clustering.

A1.

Feature	**KNN**	**K-Means Clustering**
Type	Supervised Learning	Unsupervised Learning
Task	Classification/Regression	Clustering
Model	Lazy learner (no training phase)	Model-based learner (centroids are updated)
Input	Labeled data	Unlabeled data
Output	Predicted label for new data	k clusters
Goal	Predict class/label for new data points	Group similar data points into clusters

Q2: How is the KNN algorithm used in radar target specification?

A2. In radar target specification, the K-Nearest Neighbors (KNN) algorithm is used to classify radar signals based on the characteristics of detected objects. By analyzing features such as signal strength, velocity, and range, KNN identifies the "k" closest known objects (e.g., aircraft, birds, or drones) in the dataset. The radar target is then classified based on the majority label of these nearest neighbors. Since KNN is a non-parametric method, it can adapt to various radar environments without assuming a specific distribution, making it effective in distinguishing between multiple types of targets in real-time applications.

Q3: Does the KNN algorithm have a centroid as *k*-means?

A3. No, the KNN algorithm does not use centroids. KNN is a **lazy learning** algorithm that classifies data points based on the **nearest neighbors** in the training set. It works by calculating the distance between the new point and existing labeled data points, then assigning the majority label of the closest neighbors (k-nearest points).

In contrast, **K-Means** uses centroids, which are the mean points of clusters. K-Means updates these centroids iteratively to minimize intra-cluster distances, grouping similar data points into clusters. Thus, KNN and K-Means use fundamentally different approaches.

Q4: Is there a way to obtain the centroid for the classified data by KNN?

A4. Yes, you can obtain a **centroid** for the classified data in KNN by calculating the mean of the feature values of the data points within the same class. Once KNN classifies a new data point, you can group all the points assigned to that class (including the newly classified point) and compute the centroid as the average of their coordinates in the feature space. However, this centroid isn't part of the KNN algorithm itself, as KNN doesn't involve centroids inherently, but this post-classification method can provide a central reference point for each class.

Q5: What is the difference between Supervised and unsupervised learning?

A5. The main difference between **supervised** and **unsupervised learning** lies in the data used. **Supervised learning** uses **labeled data**, where each input has a corresponding output (e.g., class label). The algorithm learns to predict outputs for new data by mapping inputs to known outputs. Common tasks include classification and regression.

In contrast, **unsupervised learning** uses **unlabeled data** and aims to find hidden patterns or groupings without predefined outputs. It is often used for clustering and dimensionality reduction tasks.

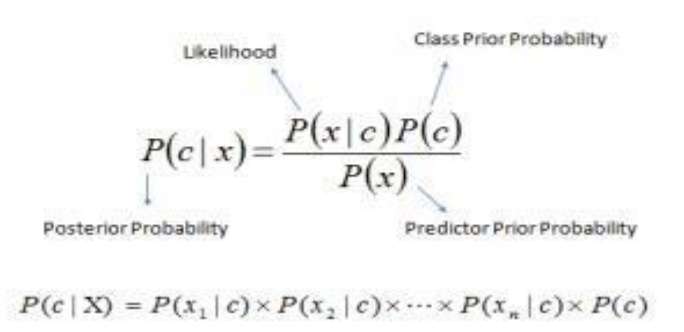
Supervised learning predicts specific outcomes, while unsupervised learning explores data structure and relationships without guidance.

Experiment 6

Aim: Write a Program to demonstrate Naive- Bayes Classifier**.**

Theory

Naive Bayes algorithm - It is a classification technique based on Bayes' Theorem with an as- sumption of independence among predictors. In simple terms, a Naive Bayes classifier assumes that the presence of a particular feature in a class is unrelated to the presence of any other fea- ture. For example, a fruit may be considered to be an apple if it is red, round, and about 3 inches in diameter. Even if these features depend on each other or upon the existence of the other fea- tures, all of these properties independently contribute to the probability that this fruit is an apple and that is why it is known as 'Naive'. Naive Bayes model is easy to build and particularly use- ful for very large data sets. Along with simplicity, Naive Bayes is known to outperform even highly sophisticated classification methods. Bayes theorem provides a way of calculating posterior probability P(c|x) from P(c), P(x) and P(x|c). Look at the equation below:



$$P(c \mid x) = \frac{P(x \mid c)\,P(c)}{P(x)}$$

$$P(c \mid X) = P(x_1 \mid c) \times P(x_2 \mid c) \times \cdots \times P(x_n \mid c) \times P(c)$$

Above,

- $P(c|x)$ is the posterior probability of *class* (c, *target*) given *predictor* (x, *attributes*).
- $P(c)$ is the prior probability of *class*.
- $P(x|c)$ is the likelihood which is the probability of *predictor* given *class*.
- $P(x)$ is the prior probability of *predictor*.

Pros and Cons of Naive Bayes

Pros:

- It is easy and fast to predict the class of the test data set. It also performs well in multi-class prediction
- When the assumption of independence holds, a Naive Bayes classifier performs better compared to other models like logistic regression and you need less training data.
- It performs well in case of categorical input variables compared to a numerical vari- able(s). For a numerical variable, the normal distribution is assumed (bell curve, which is a strong assumption).

Cons:

- If the categorical variable has a category (in the test data set), which was not observed in training data set, then the model will assign a 0 (zero) probability and will be unable to make a prediction. This is often known as “Zero Frequency”. To solve this, we can use the smoothing technique. One of the simplest smoothing techniques is called Laplace es-timation.
- On the other side, naive Bayes is also known as a bad estimator, so the probability out- puts from predict_proba are not to be taken too seriously.
- Another limitation of Naive Bayes is the assumption of independent predictors. In real life, it is almost impossible that we get a set of predictors which are completely inde- pendent.

Applications of Naive Bayes Algorithms

- **Real-time Prediction:** Naive Bayes is an eager learning classifier and it is sure fast. Thus, it could be used for making predictions in real time.
- **Multi-class Prediction:** This algorithm is also well known for multi-class prediction fea-ture. Here we can predict the probability of multiple classes of the target variable.
- **Text classification/ Spam Filtering/ Sentiment Analysis:** Naive Bayes classifiers mostly used in text classification (due to a better result in multi-class problems and inde- pendence rule) have a higher success rate as compared to other algorithms. As a result, it is widely used in Spam filtering (identify spam e-mail) and Sentiment Analysis (in social media analysis, to identify positive and negative customer sentiments)
- **Recommendation System:** Naive Bayes Classifier and Collaborative Filtering tech-niques, together build a Recommendation System that uses machine learning and data-mining techniques to filter unseen information and predict whether a user would like a given resource or not.

Python Implementation for Naive Bayes Classifier

```
import numpy as np

from sklearn.naive_bayes import GaussianNB

from sklearn.model_selection import train_test_split

from sklearn.metrics import accuracy_score


# Sample data

X = np.array([[1, 2], [2, 3], [3, 1], [4, 3], [5, 3], [6, 2]])

y = np.array([0, 0, 1, 1, 1, 0])


# 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=0)


# Create and fit the Naive Bayes classifier

model = GaussianNB()

model.fit(X_train, y_train)


# Predict the output

y_pred = model.predict(X_test)


# Calculate accuracy

accuracy = accuracy_score(y_test, y_pred)

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

Output

```
Accuracy: 0.5
```

<u>Python Implementation for Naive Bayes Classifier using Iris dataset</u>

```
from sklearn.datasets import load_iris

from sklearn.model_selection import train_test_split

from sklearn.naive_bayes import GaussianNB

X, y = load_iris(return_X_y=True)

X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.5, random_state=0)

gnb = GaussianNB()

y_pred = gnb.fit(X_train, y_train).predict(X_test)

print("Number of mislabeled points out of a total %d points : %d" %
(X_test.shape[0], (y_test != y_pred).sum()))
```

Output

```
Number of mislabeled points out of a total 75 points : 4



Process finished with exit code 0
```

Conclusion: Successfully implemented naïve bayes classifier on iris dataset using python.

Viva Questions

Q1: **What is the Naive Bayes classifier?**
A1: Naive Bayes is a probabilistic machine learning algorithm based on Bayes' Theorem. It assumes independence between features, making it "naive." Despite this assumption, it works well for classification tasks, especially with high-dimensional datasets. The algorithm calculates the posterior probability of a class given the input features and selects the class with the highest probability.

Q2: **How does Naive Bayes calculate probabilities?**
A2: Naive Bayes uses Bayes' Theorem:

$P(y|X) = P(X|y) . P(y)/P(X)$
Here, $P(y|X)$ is the posterior probability of class yyy given input features X, $P(X|y)$ is the likelihood, $P(y)$ is the prior probability, and $P(X)$ is the marginal likelihood. It predicts the class with the highest posterior probability.

Q3: **What are the main types of Naive Bayes classifiers?**
A3: There are three main types of Naive Bayes classifiers:
1. **Gaussian Naive Bayes**: Assumes that features follow a normal distribution.
2. **Multinomial Naive Bayes**: Suitable for discrete counts, often used in text classification.
3. **Bernoulli Naive Bayes**: Used when features are binary (e.g., presence or absence of a word in text).

Q4: **What are the key advantages of Naive Bayes?**
A4: Naive Bayes is fast, efficient, and works well with large datasets. It performs well even with less training data. The algorithm is simple to implement and interpretable. It's particularly effective for text classification, spam filtering, and sentiment analysis due to its assumption of feature independence.

Q5: **What are the limitations of Naive Bayes?**
A5: The key limitation is its assumption of feature independence, which rarely holds in real-world data. It can also be sensitive to zero probabilities if a feature does not appear in the training data for a given class, though this can be addressed with techniques like Laplace smoothing. It may not perform well if correlations between features are significant.

Experiment 7

Aim: Write a Program to demonstrate PCA and LDA on Iris dataset.

Theory

Principal Component Analysis (PCA) - As the number of features or dimensions in a dataset increases, the amount of data required to obtain a statistically significant result increases exponentially. This can lead to issues such as overfitting, increased computation time, and reduced accuracy of machine learning models this is known as the curse of dimensionality problems that arise while working with high-dimensional data. Moreover, as the number of dimensions increases, the number of possible combinations of features increases exponentially, which makes it computationally difficult to obtain a representative sample of the data. It becomes expensive to perform tasks such as clustering or classification because the algorithms need to process a much larger feature space, which increases computation time and complexity. Additionally, some machine learning algorithms can be sensitive to the number of dimensions, requiring more data to achieve the same level of accuracy as lower-dimensional data. To address the curse of dimensionality, *feature engineering techniques* are used which include feature selection and feature extraction. Dimensionality reduction is a type of feature extraction technique that aims to reduce the number of input features while retaining as much of the original information as possible. PCA is one such technique that was introduced by the mathematician **Karl Pearson** in 1901**.** It works on the condition that while the data in a higher dimensional space is mapped to data in a lower dimension space, the variance of the data in the lower dimensional space should be maximum.

- It is a statistical procedure that uses an orthogonal transformation that converts a set of correlated variables to a set of uncorrelated variables. PCA is the most widely used tool in exploratory data analysis and in machine learning for predictive models.
- It is an unsupervised learning algorithm technique used to examine the interrelations among a set of variables. It is also known as a general factor analysis where regression determines a line of best fit. It reduces the dimensionality of a data set by finding a new set of vari- ables, smaller than the original set of variables, retaining most of the sample's information, and useful for the regression and classification of data. It identifies a set of orthogonal axes, called *principal components*, that capture the maximum variance in the data. The principal components are linear combinations of the original variables in the dataset and are ordered in decreasing order of importance. The total variance captured by all the principal compo- nents is equal to the total variance in the original dataset. The first principal component captures the most variation in the data, but the second principal component captures the maximum variance that is orthogonal to the first principal component, and so on.

In *scikit-learn*, PCA is implemented as a *transformer* object that learns *n* components in its `fit` method, and can be used on new data to project it on these components. PCA centers but does not scale the input data for each feature before applying the singular-value-decomposition (SVD).

Linear Discriminant Analysis (LDA) - LDA and Quadratic Discriminant Analysis (QDA) are two classic classifiers, with, as their names suggest, a linear and a quadratic decision surface, respectively. These classifiers are attractive because they have closed-form solutions that can be easily computed, are inherently multiclass, have proven to work well in practice, and have no hyperparameters to tune.

Dimensionality reduction using Linear Discriminant Analysis

LDA can be used to perform supervised dimensionality reduction, by projecting the input data to a linear subspace consisting of the directions which maximize the separation between classes. The dimension of the output is necessarily less than the number of classes, so this is in general a rather strong dimensionality reduction, and only makes sense in a multiclass setting. This is implemented in the `transform` method. The desired dimensionality can be set using the `n_components` parameter. This parameter has no influence on the `fit` and `predict` methods.

Both LDA and QDA can be derived from simple probabilistic models which model the class conditional distribution of the data $P(X|y=k)$ for each class k. Predictions can then be obtained by using Bayes' rule, for each training sample $x \in R^d$:

$$P(y=k|x)=P(x|y=k)P(y=k)/P(x)$$

$$=P(x|y=k)P(y=k)/(\sum_l P(x|y=l)\cdot P(y=l)),$$

and we select the class k which maximizes this posterior probability. More specifically, for linear and quadratic discriminant analysis, $P(x|y)$ is modeled as a multivariate Gaussian distribution with density:

$$P(x|y=k)=1/((2\pi)^{d/2}|\Sigma_k|^{1/2}).\ exp(-1/2.(x-\mu_k)^t\Sigma_k^{-1}(x-\mu_k)),$$

where d is the number of features and Σ_k is the covariance matrix.

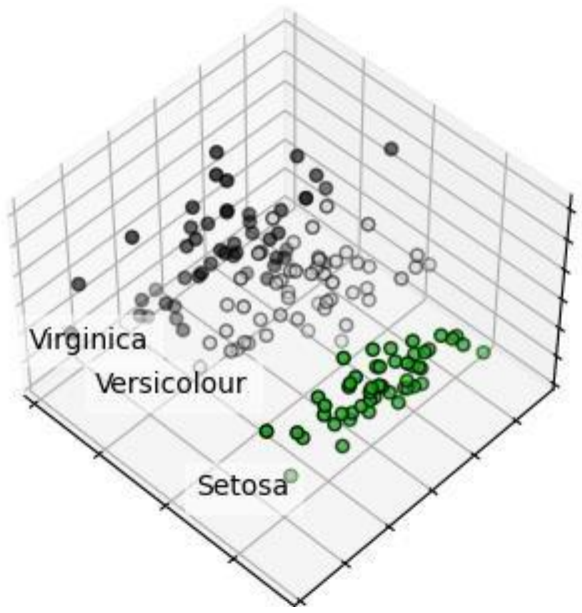
LDA is a special case of QDA, where the Gaussians for each class are assumed to share the same covariance matrix: $\Sigma_k = \Sigma$ for all k. Moreover, if in the QDA model we assume that the covariance matrices are diagonal, then the inputs are assumed to be conditionally independent in each class, and the resulting classifier is equivalent to the *Gaussian Naive Bayes classifier*.

Python Script

```
# PCA on Iris dataset

import matplotlib.pyplot as plt



# unused but required import for doing 3d projections with matplotlib
# < 3.2

import mpl_toolkits.mplot3d

import numpy as np


from sklearn import datasets, decomposition

np.random.seed(5)


iris = datasets.load_iris()

X  =  iris.data

y = iris.target


fig = plt.figure(1, figsize=(4, 3))

plt.clf()


ax = fig.add_subplot(111, projection="3d", elev=48, azim=134)

ax.set_position([0, 0, 0.95, 1])


plt.cla()

pca = decomposition.PCA(n_components=3)

pca.fit(X)

X = pca.transform(X)
```

```
for name, label in [("Setosa", 0), ("Versicolour", 1), ("Virginica",
2)]:

    ax.text3D(

        X[y == label, 0].mean(),

        X[y == label, 1].mean() + 1.5,

        X[y == label, 2].mean(),

        name,

        horizontalalignment="center",

        bbox=dict(alpha=0.5, edgecolor="w", facecolor="w"),

    )

# Reorder the labels to have colors matching the cluster results

y = np.choose(y, [1, 2, 0]).astype(float)

ax.scatter(X[:, 0], X[:, 1], X[:, 2], c=y, cmap=plt.cm.nipy_spectral,
edgecolor="k")



ax.xaxis.set_ticklabels([])

ax.yaxis.set_ticklabels([])

ax.zaxis.set_ticklabels([])

plt.show()
```

Output



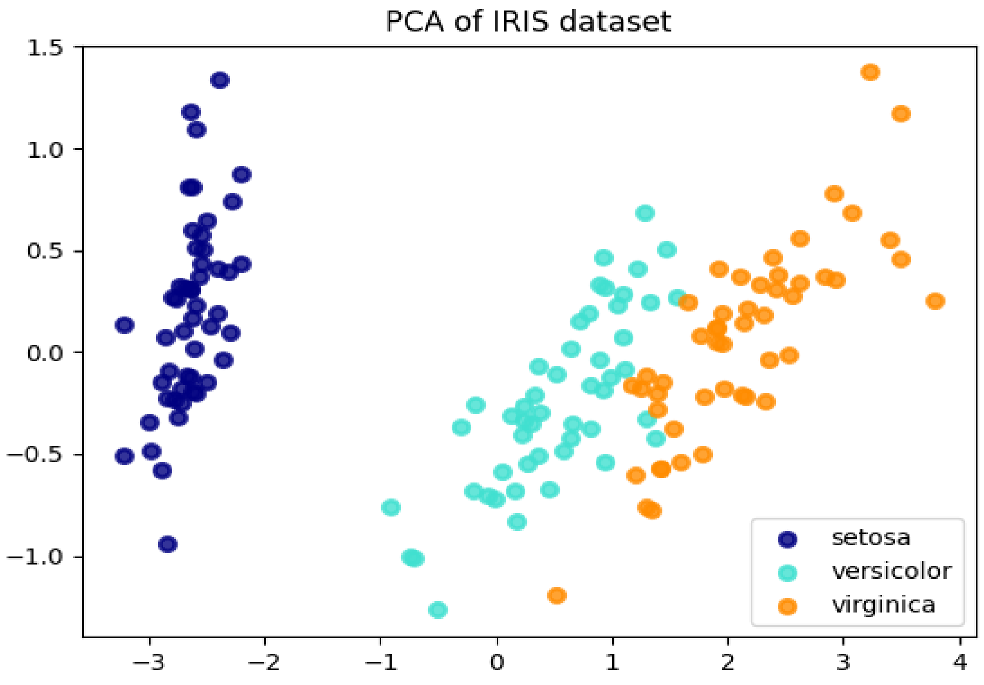
Python Script

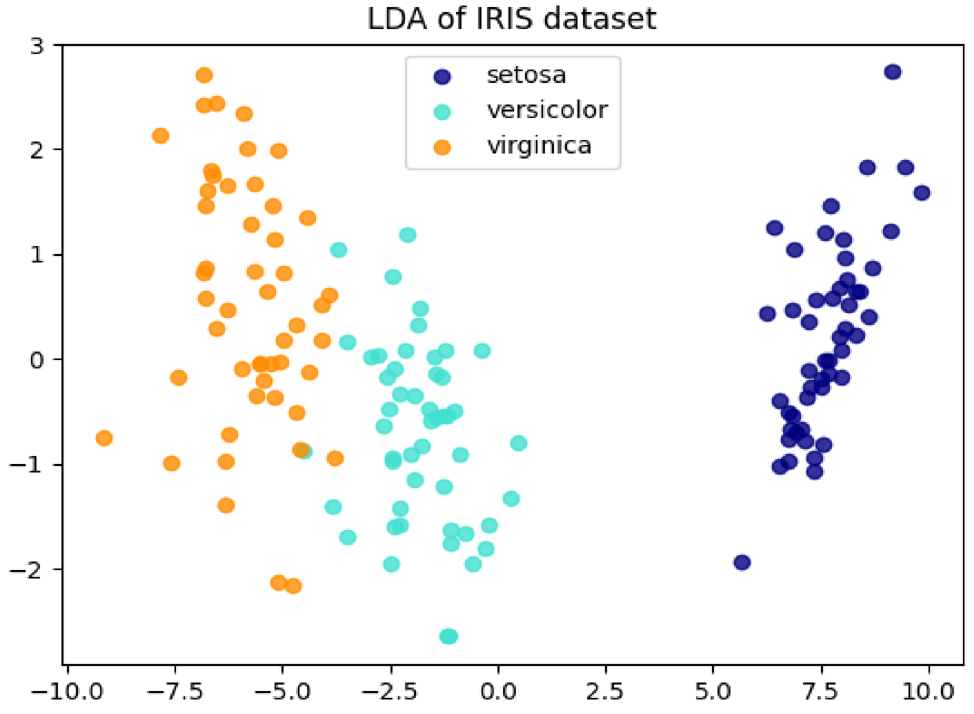
```
import matplotlib.pyplot as plt

from sklearn import datasets

from sklearn.decomposition import PCA

from sklearn.discriminant_analysis import LinearDiscriminantAnalysis

iris = datasets.load_iris()

X  =  iris.data
y = iris.target

target_names = iris.target_names

pca = PCA(n_components=2)
```

```
X_r = pca.fit(X).transform(X)

lda = LinearDiscriminantAnalysis(n_components=2)

X_r2 = lda.fit(X, y).transform(X)

# Percentage of variance explained for each components

print(

    "Explained variance ratio (first two components): %s"

    % str(pca.explained_variance_ratio_)

)



plt.figure()

colors = ["navy", "turquoise", "darkorange"]

lw = 2

for color, i, target_name in zip(colors, [0, 1, 2], target_names):

    plt.scatter(

        X_r[y == i, 0], X_r[y == i, 1], color=color, alpha=0.8, lw=lw,
label=target_name

    )

plt.legend(loc="best", shadow=False, scatterpoints=1)

plt.title("PCA of IRIS dataset")

plt.figure()
```

```
for color, i, target_name in zip(colors, [0, 1, 2], target_names):

    plt.scatter(

        X_r2[y == i, 0], X_r2[y == i, 1], alpha=0.8, color=color, la-
bel=target_name

    )

plt.legend(loc="best", shadow=False, scatterpoints=1)

plt.title("LDA of IRIS dataset")

plt.show()
```

Output


PCA of IRIS dataset
1.5
1.0
0.5
0.0
−0.5
−1.0
−3
−2
−1
0
1
2
3
4
setosa
versicolor
virginica


LDA of IRIS dataset
3
2
1
0
−1
−2
−10.0
−7.5
−5.0
−2.5
0.0
2.5
5.0
7.5
10.0
setosa
versicolor
virginica

```
Explained variance ratio (first two components): [0.92461872
0.05306648]

Process finished with exit code 0
```

Conclusion: Successfully demonstrated Principal Component analysis and Linear discriminant analysis on iris dataset.

Viva Questions

Q1: **What is Principal Component Analysis (PCA) and what is its purpose?**
A1: Principal Component Analysis (PCA) is a dimensionality reduction technique used to simplify large datasets by transforming them into fewer, uncorrelated variables called principal components. It preserves the most important variance in the data while reducing noise and redundancy. PCA is unsupervised and primarily used for data visualization, noise reduction, and preprocessing for machine learning models.

Q2: **What is Linear Discriminant Analysis (LDA) and how is it different from PCA?**
A2: Linear Discriminant Analysis (LDA) is a supervised dimensionality reduction technique that aims to maximize class separability by projecting data onto a lower-dimensional space. Unlike PCA, which focuses on capturing variance, LDA takes class labels into account and attempts to find a linear combination of features that best separates the classes. LDA is useful for classification tasks.

Q3: **How does PCA work mathematically?**
A3: PCA works by calculating the covariance matrix of the data to understand feature relationships. It then computes the eigenvectors (principal components) and eigenvalues, which define the direction and magnitude of the variance. The data is projected onto these eigenvectors, transforming it into a new set of uncorrelated features (principal components) ordered by the amount of variance they capture.

Q4: **When should you prefer LDA over PCA?**
A4: LDA is preferred when the goal is to perform classification and when class labels are available. Since LDA maximizes class separability, it works well when the data follows a Gaussian distribution for each class and class labels play a key role in determining meaningful components. PCA, on the other hand, is more suited for unsupervised tasks or when reducing dimensions for exploratory data analysis.

Q5: **What are the limitations of PCA and LDA?**
A5: PCA may struggle when the most important features for classification are not those with the most variance. It also assumes linear relationships between features. LDA, while more powerful for classification, assumes that classes follow a Gaussian distribution and may perform poorly when this assumption is violated. Both techniques can also lose information when too much dimensionality reduction is applied.

Experiment 8

Aim: Write a Program to demonstrate DBSCAN clustering algorithm.

Theory

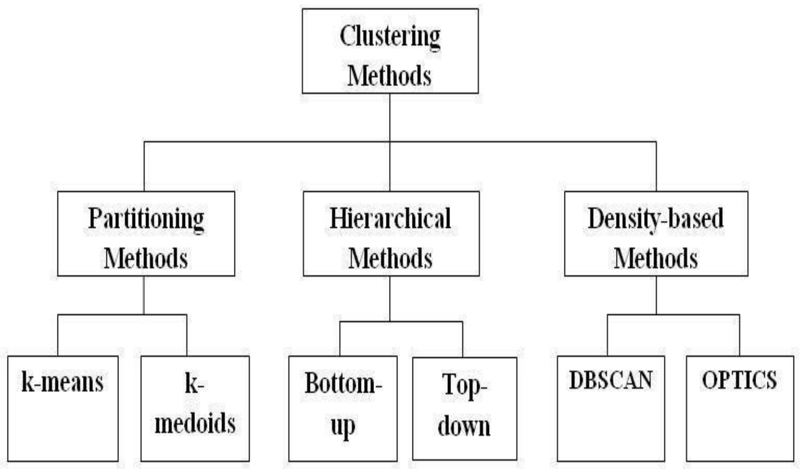


Fig. : Clustering techniques

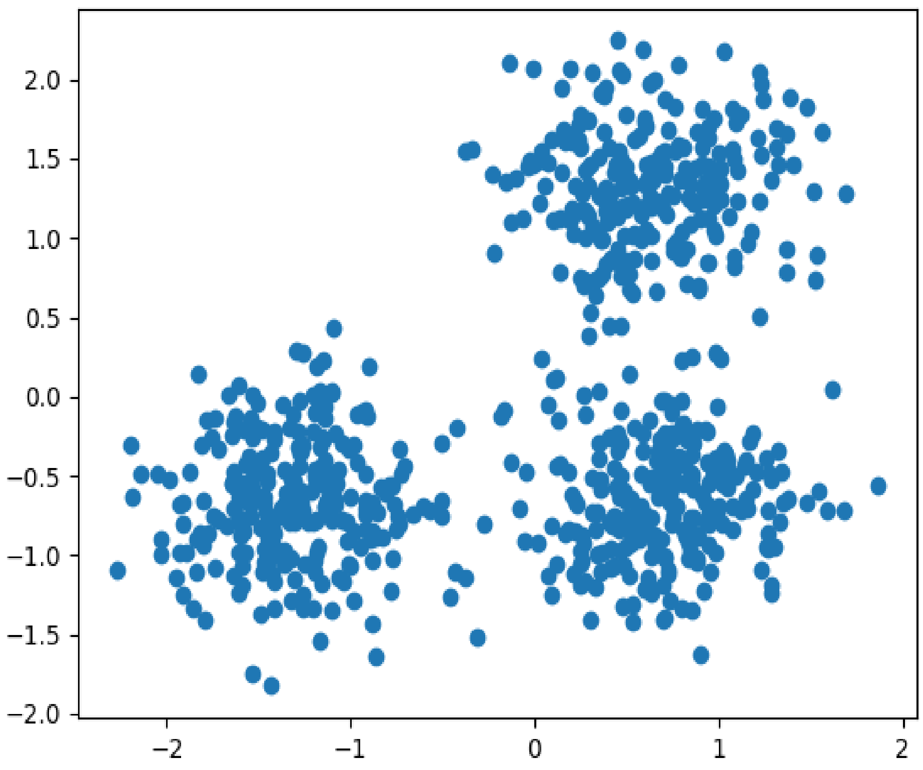
DBSCAN (Density-Based Spatial Clustering of Applications with Noise) finds core samples in regions of high density and expands clusters from them. This algorithm is good for data which contains clusters of similar density.

Clustering algorithms are fundamentally unsupervised learning methods. However, since `make_blobs` gives access to the true labels of the synthetic clusters, it is possible to use evaluation metrics that leverage this "supervised" ground truth information to quantify the quality of the resulting clusters. Examples of such metrics are the homogeneity, completeness, V-measure, Rand-Index, Adjusted Rand-Index and Adjusted Mutual Information (AMI). If the ground truth labels are not known, evaluation can only be performed using the model results itself. In that case, the Silhouette Coefficient comes in handy.

Python Script

```
# Data generation

# We use make_blobs to create 3 synthetic clusters.

from sklearn.datasets import make_blobs

from sklearn.preprocessing import StandardScaler

centers = [[1, 1], [-1, -1], [1, -1]]

X, labels_true = make_blobs(n_samples=750, centers=centers,

cluster_std=0.4, random_state=0)


X = StandardScaler().fit_transform(X)

# We can visualize the resulting data:

import matplotlib.pyplot as plt

plt.scatter(X[:, 0], X[:, 1])

plt.show()



# Compute DBSCAN

# One can access the labels assigned by DBSCAN using the labels_ at-
tribute. Noisy samples are given the label math:-1.



from sklearn import metrics
from sklearn.cluster import DBSCAN


db = DBSCAN(eps=0.3, min_samples=10).fit(X)

labels = db.labels_


# Number of clusters in labels, ignoring noise if present.

n_clusters_ = len(set(labels)) - (1 if -1 in labels else 0)

n_noise_ = list(labels).count(-1)


print("Estimated number of clusters: %d" % n_clusters_)

print("Estimated number of noise points: %d" % n_noise_)


print(f"Homogeneity: {metrics.homogeneity_score(labels_true, la-
bels):.3f}")

print(f"Completeness: {metrics.completeness_score(labels_true, la-
bels):.3f}")

print(f"V-measure: {metrics.v_measure_score(labels_true, la-
bels):.3f}")

print(f"Adjusted Rand Index: {metrics.adjusted_rand_score(labels_true,
labels):.3f}")

print(

    "Adjusted Mutual Information:"

    f" {metrics.adjusted_mutual_info_score(labels_true, labels):.3f}"

)

print(f"Silhouette Coefficient: {metrics.silhouette_score(X, la-
bels):.3f}")
```

Output



```
Estimated number of clusters: 3

Estimated number of noise points: 18


Homogeneity: 0.953


Completeness: 0.883


V-measure: 0.917


Adjusted Rand Index: 0.952

Adjusted Mutual Information: 0.916

Silhouette Coefficient: 0.626



Process finished with exit code 0
```

Conclusion: Successfully demonstrated DBSCAN algorithm in python.

Viva Questions

Q1: **What is DBSCAN, and how does it differ from other clustering algorithms?**
A1: DBSCAN (Density-Based Spatial Clustering of Applications with Noise) is a density-based clustering algorithm that groups together points that are closely packed (high density) and marks points in low-density areas as outliers. Unlike K-means, which requires specifying the number of clusters, DBSCAN can find clusters of arbitrary shapes and automatically detects the number of clusters based on data density, making it robust to noise.

Q2: **What are the key parameters in DBSCAN?**
A2: DBSCAN has two main parameters:

1. **Epsilon (ε)**: The maximum distance between two points for them to be considered part of the same cluster.
2. **MinPts**: The minimum number of points required to form a dense region. A point is classified as a core point if it has at least MinPts neighbors within the ε distance.

Q3: **How does DBSCAN classify data points?**
A3: DBSCAN classifies data points into three categories:

1. **Core points**: Points that have at least MinPts neighbors within the ε radius.
2. **Border points**: Points that are within the ε radius of a core point but do not have enough neighbors to be a core point themselves.
3. **Noise points**: Points that do not belong to any cluster, i.e., they are neither core nor border points.

Q4: **What are the advantages of using DBSCAN?**
A4: DBSCAN has several advantages: it can detect clusters of arbitrary shapes, automatically determines the number of clusters, and is robust to noise and outliers. It does not require the number of clusters to be specified in advance, making it ideal for exploratory data analysis, especially when clusters are non-spherical or unevenly distributed.

Q5: **What are the limitations of DBSCAN?**
A5: DBSCAN's performance is sensitive to the selection of ε and MinPts parameters, which can be difficult to set optimally for complex datasets. It may struggle with clusters of varying density, as a fixed ε may not work well for all clusters. Additionally, DBSCAN does not perform well with high-dimensional data, where the concept of "density" becomes less meaningful due to the curse of dimensionality.

Experiment 9

Aim: Write a Program to demonstrate K-Medoid clustering algorithm.

Theory

K-Medoids is an unsupervised clustering algorithm in which data points called “medoids" act as the cluster's center. A medoid is a point in the cluster whose sum of distances (also called dissimilarity) to all the objects in the cluster is minimal. The distance can be the Euclidean distance, Manhattan distance, or any other suitable distance function. Therefore, the K -medoids algorithm divides the data into K clusters by selecting K medoids from the data sample.

K-Medoids Clustering Algorithm - The K-medoids clustering algorithm can be summarized as follows −

- **Initialize k medoids** − Select k random data points from the dataset as the initial medoids.
- **Assign data points to medoids** − Assign each data point to the nearest medoid.
- **Update medoids** − For each cluster, select the data point that minimizes the sum of distances to all the other data points in the cluster, and set it as the new medoid.
- Repeat steps 2 and 3 until convergence or a maximum number of iterations is reached.

K-Medoids Clustering – Advantages

Here are the advantages of using K-medoids clustering −

i. Robust to outliers and noise − K-medoids clustering is more robust to outliers and noise than K-means clustering because it uses a representative data point, called a medoid, to rep- resent the center of the cluster.

ii. Can handle non-Euclidean distance metrics − K-medoids clustering can be used with any distance metric, including non-Euclidean distance metrics, such as Manhattan distance and cosine similarity.

iii. Computationally efficient − K-medoids clustering has a computational complexity of O(k*n^2), which is lower than the computational complexity of K-means clustering.

K-Medoids Clustering - Disadvantages

The disadvantages of using K-medoids clustering are as follows −

i. Sensitive to the choice of k − The performance of K-medoids clustering can be sensitive to the choice of k, the number of clusters.

ii. Not suitable for high-dimensional data − K-medoids clustering may not perform well on high-dimensional data because the medoid selection process becomes computationally expensive.

Implementation in Python

To implement K-medoids clustering in Python, we can use the scikit-learn library. The scikit-learn library provides the **KMedoids** class, which can be used to perform K-medoids clustering on a dataset.

Firstly, we need to install `scikit-learn-extra` using `pip install scikit-learn-extra`.

Then, we need to import the required libraries −

```
from sklearn_extra.cluster import KMedoids

from sklearn.datasets import make_blobs
```

Next, we generate a sample dataset using the make_blobs() function from scikit-learn −

```
X, y = make_blobs(n_samples=500, centers=3, random_state=42)
```

Here, we generate a dataset with 500 data points and 3 clusters.

Next, we initialize the KMedoids class and fit the data −

```
kmedoids = KMedoids(n_clusters=3, random_state=42)
kmedoids.fit(X)
```

Here, we set the number of clusters to 3 and use the random_state parameter to ensure reproducibility.

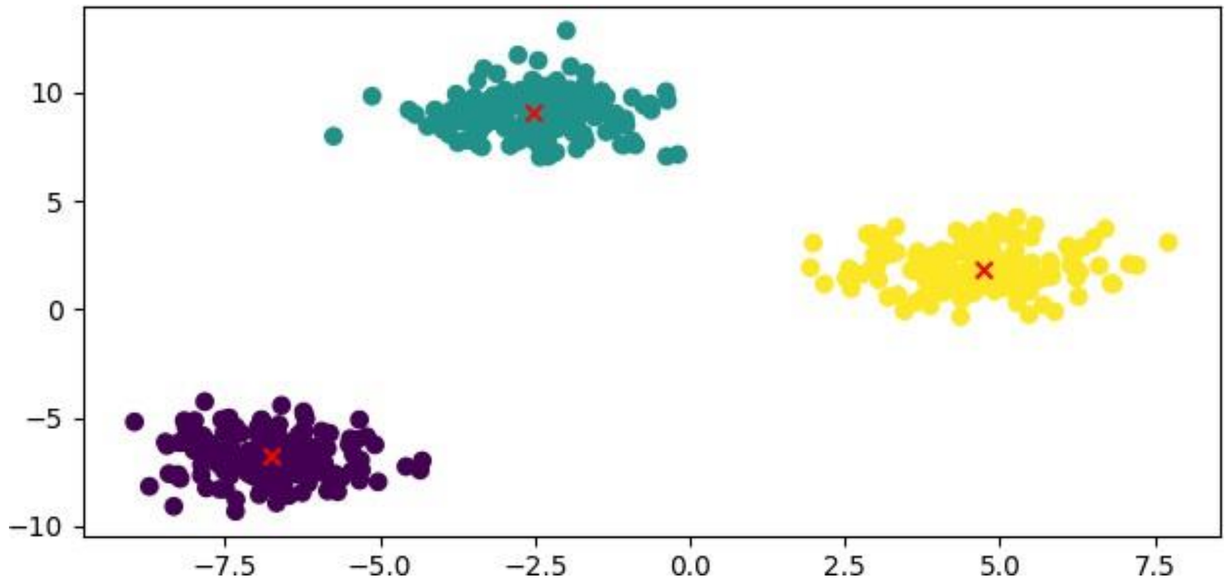
Finally, we can visualize the clustering results using a scatter plot −

```
plt.figure(figsize=(7.5, 3.5))

plt.scatter(X[:, 0], X[:, 1], c=kmedoids.labels_, cmap='viridis')

plt.scatter(kmedoids.cluster_centers_[:, 0],
kmedoids.cluster_centers_[:, 1], marker='x', color='red')
```

Complete Python Script

```
from sklearn_extra.cluster import KMedoids

from sklearn.datasets import make_blobs

import matplotlib.pyplot as plt


# Generate sample data

X, y = make_blobs(n_samples=500, centers=3, random_state=42)


# Cluster the data using KMedoids

kmedoids = KMedoids(n_clusters=3, random_state=42)

kmedoids.fit(X)


# Plot the results

plt.figure(figsize=(7.5, 3.5))

plt.scatter(X[:, 0], X[:, 1], c=kmedoids.labels_, cmap='viridis')

plt.scatter(kmedoids.cluster_centers_[:, 0],

kmedoids.cluster_centers_[:, 1], marker='x', color='red')

plt.show()
```

Output



# Here, the data points are plotted as a scatter plot, and colored based on their cluster labels. Also, the medoids are plotted as red crosses.

Conclusion: Successfully implemented K-mediod algorithm in Python.

Viva Questions

Q1: **What is the K-medoids algorithm and how does it work?**
A1: K-medoids is a partition-based clustering algorithm that groups data into a predefined number of clusters, similar to K-means. However, instead of using the mean to represent the cluster center, K-medoids selects actual data points (medoids) as the cluster centers. It works by iteratively minimizing the sum of dissimilarities between points and their nearest medoid, making it more robust to noise and outliers compared to K-means.

Q2: **How are medoids chosen in the K-medoids algorithm?**
A2: Initially, K-medoids randomly selects K data points as medoids. The algorithm then iteratively improves the selection by swapping medoids with non-medoids if this reduces the total cost, defined as the sum of dissimilarities (e.g., distance) between data points and their nearest medoid. This process continues until no further cost reduction is possible or a maximum number of iterations is reached.

Q3: **What are the advantages of K-medoids over K-means?**
A3: K-medoids is more robust to noise and outliers because it uses actual data points as cluster centers

instead of the mean, which can be skewed by extreme values. It also works better for datasets with non-convex shapes or when the dataset contains categorical or mixed data, where calculating a meaningful mean may be difficult or inappropriate.

Q4: **What are the limitations of K-medoids?**
A4: K-medoids can be computationally expensive compared to K-means, especially for large datasets, because it involves calculating pairwise dissimilarities between all data points. The algorithm's complexity increases with the size of the dataset. Additionally, K-medoids may struggle with very high-dimensional data and can be sensitive to the initial selection of medoids, potentially leading to suboptimal clustering.

Q5: **When would you use K-medoids instead of K-means?**
A5: K-medoids is preferred over K-means when the data contains noise or outliers, as it is less sensitive to extreme values. It is also suitable when dealing with non-Euclidean distances or when working with categorical data, where calculating the mean is not meaningful. Additionally, K-medoids is more effective for datasets with irregular cluster shapes that K-means may fail to capture.

Experiment 10

Aim: Write a Program to demonstrate K-Means clustering algorithm on Handwritten dataset.

Theory

K – means clustering is an unsupervised algorithm that is used in customer segmentation applications. In this algorithm, we try to form clusters within our datasets that are closely related to each other in a high-dimensional space. In other words, K-Means clustering is an Unsupervised Machine Learning algorithm, which groups the unlabeled dataset into different clusters. Unsupervised Machine Learning is the process of teaching a computer to use unlabeled, unclassified data and enabling the algorithm to operate on that data without supervision. Without any pre- vious data training, the machine's job in this case is to organize unsorted data according to parallels, patterns, and variations. K-means clustering, assigns data points to one of the K clus- ters depending on their distance from the center of the clusters. It starts by randomly assigning the clusters centroid in the space. Then each data point assign to one of the cluster based on its distance from centroid of the cluster. After assigning each point to one of the cluster, new clus- ter centroids are assigned. This process runs iteratively until it finds good cluster. In the analy- sis we assume that number of cluster is given in advanced and we have to put points in one of the group. In some cases, K is not clearly defined, and we have to think about the optimal number of K. K-Means clustering performs best data is well separated. When data points over- lapped this clustering is not suitable. K-Means is faster as compare to other clustering tech- nique. It provides strong coupling between the data points. K-Means cluster do not provide clear information regarding the quality of clusters. Different initial assignment of cluster cen- troid may lead to different clusters. Also, K-Means algorithm is sensitive to noise. It may have stuck in local minima.

Objective of K-means clustering- The goal of clustering is to divide the population or set of data points into a number of groups so that the data points within each group are more comparable to one another and different from the data points within the other groups. It is essentially a grouping of things based on how similar and different they are to one another.

How k-means clustering works?

Suppose, we are given a data set of items, with certain features, and values for these features (like a vector). The task is to categorize those items into groups. To achieve this, we will use the K-means algorithm, an unsupervised learning algorithm. 'K' in the name of the algorithm represents the number of groups/clusters we want to classify our items into. The algorithm will categorize the items into k groups or clusters of similarity. To calculate that similarity, we will use the Euclidean distance as a measurement.

The algorithm works as follows:

- First, we randomly initialize k points, called means or cluster centroids.
- We categorize each item to its closest mean, and we update the mean's coordinates, which are the averages of the items categorized in that cluster so far.
- We repeat the process for a given number of iterations and at the end, we have our clus- ters.

The "points" mentioned above are called means because they are the mean values of the items categorized in them. To initialize these means, we have a lot of options. An intuitive method is to initialize the means at random items in the data set. Another method is to initialize the means at random values between the boundaries of the data set (if for a feature x, the items have values in [0,3], we will initialize the means with values for x at [0,3]).

In this experiment, we will see how to use the K-means algorithm to identify the clusters of the digits.

Python Script

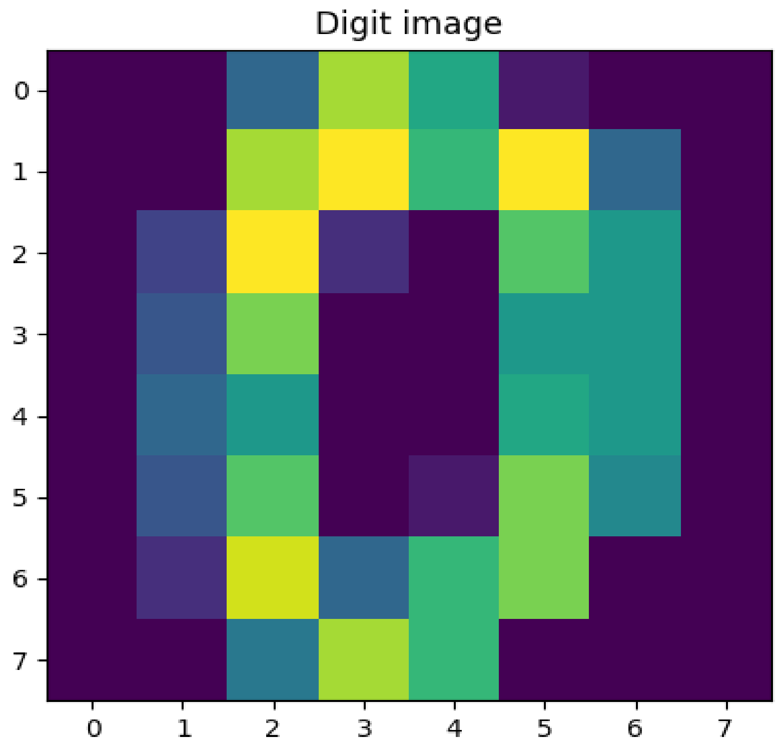
```
# Load the dataset

# We will start by loading the digits dataset.

# In the context of clustering, one would like to group images such
that the handwritten digits on the image are the same.


from sklearn.datasets import load_digits


digits_data = load_digits().data

print(digits_data)

# Each handwritten digit in the data is an array of color values of
pixels of its image.

# For better understanding, let's print how the data of the first
digit looks like and then display its's respective image.


import matplotlib.pyplot as plt

print("First handwritten digit data: ", + digits_data[0])

sample_digit = digits_data[0].reshape(8, 8)
```

```
plt.imshow(sample_digit)

plt.title("Digit image")

plt.show()


# In the next step, we scale the data. Scaling is an optional yet very
helpful technique for the faster processing of the model.

# In our model, we scale the pixel values which are typically between
0 – 255 to -1 – 1,

# easing the computation and avoiding super large numbers.

# Another point to consider is that a train test split is not required
for this model as it is unsupervised learning with no labels to test.

# Then, we define the k value, which is 10 as we have 0-9 digits in
our data.Also setting up the target variable.


from sklearn.preprocessing import scale


scaled_data = scale(digits_data)

print(scaled_data)


Y = load_digits().target

print(Y)


# Defining k-means clustering:

# Now we define the K-means cluster using the KMeans function from the
sklearn module.


#  Method 1: Using a Random initial cluster.

# Setting the initial cluster points as random data points by using
the ‘init‘ argument.

# The argument ‘n_init‘ is the number of iterations the k-means
clustering should run with different initial clusters chosen at random,
# in the end, the clustering with the least total variance is considered'

# The random state is kept to 0 (any number can be given) to fix the same random initial clusters every time the code is run.

from sklearn.cluster import KMeans

k = 10

kmeans_cluster = KMeans(init = "random",

n_clusters = k, n_init = 10,

random_state = 0)


# Method 2: Using k-means++

# It is similar to method-1 however, it is not completely random, and chooses the initial clusters far away from each other.

# Therefore, it should require fewer iterations in finding the clusters when compared to the random initialization.

kmeans_cluster = KMeans(init="k-means++", n_clusters=k, n_init=10, random_state=0)


# Model Evaluation

# We will use scores like silhouette score, time taken to reach optimum position, v_measure and some other important metrics.


from sklearn import metrics

from time import time

def bench_k_means(estimator, name, data):

initial_time = time() estimator.fit(data)

print("Initial-cluster: " + name)

print("Time taken: {0:0.3f}".format(time() - initial_time)) print("Homogeneity:

{0:0.3f}".format(
metrics.homogeneity_score(Y, estimator.labels_))) print("Completeness:

{0:0.3f}".format(

metrics.completeness_score(Y, estimator.labels_))) print("V_measure:

{0:0.3f}".format(

metrics.v_measure_score(Y, estimator.labels_)))

print("Adjusted random: {0:0.3f}".format(

metrics.adjusted_rand_score(Y, estimator.labels_))) print("Adjusted

mutual info: {0:0.3f}".format(

metrics.adjusted_mutual_info_score(Y, estimator.labels_))) print("Silhouette:

{0:0.3f}".format(metrics.silhouette_score(

data,estimator.labels_, metric='euclidean', sample_size=300)))


kmeans_cluster=KMeans(init="random",n_clusters=k,n_init=10,
random_state=0)

bench_k_means(estimator=kmeans_cluster,name="random",
data=digits_data)


kmeans_cluster = KMeans(init="k-means++", n_clusters=k,

n_init=10, random_state=0)

bench_k_means(estimator=kmeans_cluster,name="random",
data=digits_data)


# Visualizing the K-means clustering for handwritten data:

# Plotting the k-means cluster using the scatter function provided by
the matplotlib module.

# Reducing the large dataset by using Principal Component Analysis
(PCA) and fitting it to the previously defined k-means++ model.

# Plotting the clusters with different colors, a centroid was marked
for each cluster.


from sklearn.decomposition import PCA
```

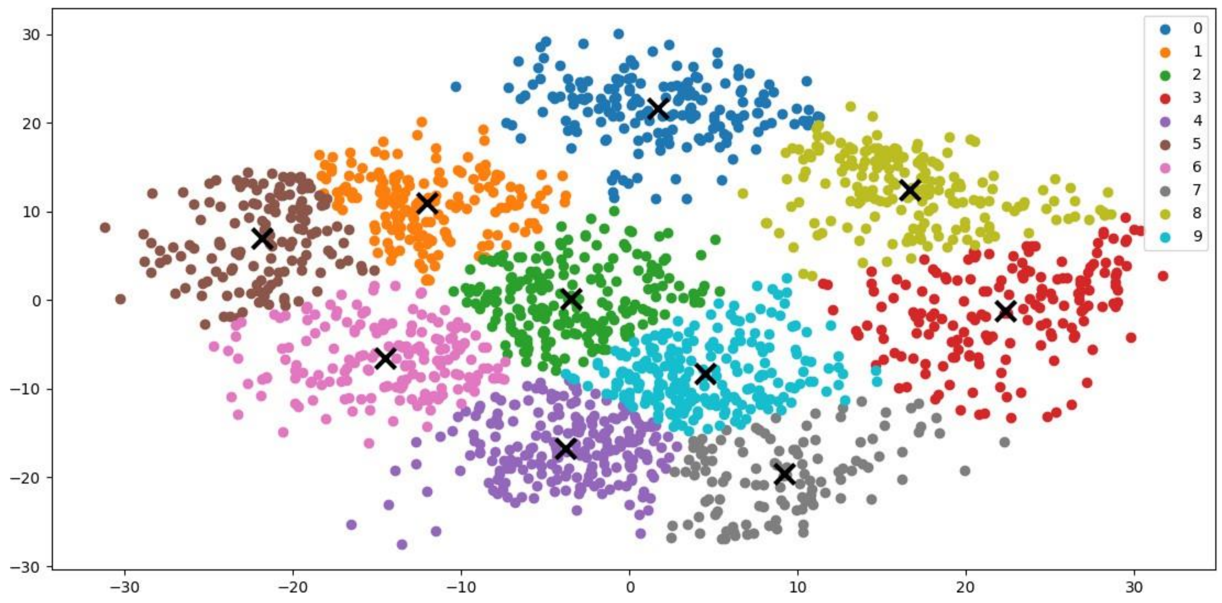
```
import numpy as np


# Reducing the dataset

pca = PCA(2)

reduced_data = pca.fit_transform(digits_data)

kmeans_cluster.fit(reduced_data)


# Calculating the centroids

centroids = kmeans_cluster.cluster_centers_

label = kmeans_cluster.fit_predict(reduced_data)

unique_labels = np.unique(label)


# plotting the clusters:

plt.figure(figsize=(8, 8))

for i in unique_labels:

plt.scatter(reduced_data[label == i, 0],

reduced_data[label == i, 1], label=i)

plt.scatter(centroids[:, 0], centroids[:, 1],

marker='x', s=169, linewidths=3, color='k', zorder=10)

plt.legend()

plt.show()
```

Output

```
[[ 0.  0. 5. ...  0.   0.  0.]
 [ 0.  0. 0. ... 10.   0.  0.]
 [ 0.  0. 0. ... 16.   9.  0.]
...

       [ 0.  0.  1. ...  6.   0.   0.]

       [ 0.  0.  2. ... 12.   0.   0.]

       [ 0.  0. 10. ... 12.  1.   0.]]

First handwritten digit data:  [ 0.   0.   5. 13.   9.   1.   0.   0.   0.   0. 13.
15. 10. 15.   5.   0.   0.   3.

15.   2.   0. 11.   8.   0.   0.   4. 12.   0.   0.   8.   8.   0.   0.   5.   8.   0.

0.   9.   8.   0.   0.   4. 11.   0.   1. 12.   7.   0.   0.   2. 14.   5. 10. 12.

0.   0.   0.   0.   6. 13. 10.   0.   0.   0.]
```



```
[[ 0.                 -0.33501649 -0.04308102 ... -1.14664746 -0.5056698

-0.19600752]

[ 0.                 -0.33501649 -1.09493684 ...  0.54856067 -0.5056698

-0.19600752]

[ 0.                 -0.33501649 -1.09493684 ...  1.56568555  1.6951369

-0.19600752]
...
[ 0.                -0.33501649 -0.88456568 ... -0.12952258 -0.5056698
-0.19600752]
[ 0.                -0.33501649 -0.67419451 ...  0.8876023  -0.5056698
-0.19600752]
[ 0.                -0.33501649  1.00877481 ...  0.8876023  -0.26113572
-0.19600752]]
[0 1 2 ... 8 9 8]
Initial-cluster: random Time
taken: 0.363
Homogeneity: 0.739
Completeness: 0.748
V_measure: 0.744
Adjusted random: 0.666 Adjusted
mutual info: 0.741 Silhouette:
0.178
Initial-cluster: random Time
taken: 0.276
Homogeneity: 0.742
Completeness: 0.751
V_measure: 0.747
Adjusted random: 0.669 Adjusted
mutual info: 0.744 Silhouette:
0.180
```



Conclusion: Successfully implemented k-means clustering algorithm.

Viva Questions

Q1: **When would you use K-means clustering and when would you use hierarchical clustering?**
A1: K-means is typically used for large datasets when the number of clusters is predefined and speed is essential. It is effective for spherical-shaped clusters and handles large data well. Hierarchical clustering is better for smaller datasets where the cluster structure is unknown, as it creates a dendrogram, showing a hierarchy of clusters. Hierarchical clustering is computationally expensive but can capture more complex cluster structures, allowing users to choose different cluster levels.

Q2: **What is the difference between classification and clustering?**
A2: Classification is a supervised learning technique where data is assigned to predefined classes based on labeled training data. The model learns the relationship between input features and output labels. Clustering, on the other hand, is an unsupervised learning method that groups data points into clusters based on similarity, without prior knowledge of class labels. Clustering discovers hidden patterns in the data, while classification predicts class labels for unseen data.

Q3: **What definition of similarity is used by the K-means clustering algorithm?**
A3: K-means clustering typically uses **Euclidean distance** as the similarity measure. The algorithm minimizes the sum of squared distances between each data point and the centroid (mean) of its assigned cluster. Points that are closer to a cluster centroid are considered more similar, and the centroid represents the average position of all points in that cluster. Other distance measures like Manhattan or cosine distance can also be used in specific cases.

Q4: **Describe the optimal K-means optimization problem; is it NP-Hard?**
A4: The K-means optimization problem seeks to minimize the total within-cluster variance, or the sum of squared distances between each point and its cluster centroid. Formally, it minimizes the objective function:
∑i=1k∑x∈Ci∥x−μi∥2\sum_{i=1}^{k} \sum_{x \in C_i} \| x - \mu_i \|^2i=1∑kx∈Ci∑∥x−μi∥2
where CiC_iCi is a cluster and μi\mu_iμi is the centroid. The K-means problem is NP-hard in general, especially in high dimensions or for a large number of clusters. Approximate solutions are found using heuristic methods like the Lloyd's algorithm.

Q5: **How are the initial centroids typically chosen in the K-means algorithm?**
A5: The initial centroids in K-means are often chosen randomly from the dataset. However, random initialization can lead to suboptimal clustering. To improve centroid selection, the **K-means++** initialization method is commonly used. It selects the first centroid randomly, then chooses subsequent centroids with a probability proportional to their distance from the existing centroids, reducing the likelihood of poor clustering and leading to faster convergence and better solutions.