Department of Computer Science and Engineering-Data Science

LAB MANUAL

Course code :

Course Name : PREDICTIVE ANALYTICS LAB

Regulation : R22

Credit Structure :

L	T	P	C
0	0	2	1

Year/Semester : IV/I

Academic Year : 2025-26

Specialization : CSE-DS

Prepared By : Mr. Abdul Majeed
Assistant Professor
CSE-DS

VISION :
MISSION :

PROGRAM OUTCOMES (POs):

1. **Engineering Knowledge:** Apply the knowledge of basic sciences and engineering fundamentals to solve engineering problems.
2. **Problem Analysis:** Analyze the complex engineering problems and give solutions related to chemical & allied industries.
3. **Design/ development of solutions:** Identify the chemical engineering problems, design and formulate solutions to solve both industrial & social related problems.
4. **Conduct investigations of complex problems:** Design & conduct experiments, analyze and interpret the resulting data to solve Chemical Engineering problems.
5. **Modern tool usage:** Apply appropriate techniques, resources and modern engineering & IT tools for the design, modeling, simulation and analysis studies.
6. **The engineer and society:** Assess societal, health, safety, legal and cultural issues and their consequent responsibilities relevant to professional engineering practice.
7. **Environment and sustainability:** Understand the relationship between society, environment and work towards sustainable development.
8. **Ethics:** Understand their professional and ethical responsibility and enhance their commitment towards best engineering practices.
9. **Individual and team work:** Function effectively as a member or a leader in diverse teams, and be competent to carry out multidisciplinary tasks.
10. **Communication:** Communicate effectively in both verbal & non-verbal and able to comprehend& write effective reports.
11. **Project management and finance:** Understand the engineering and management principles to manage the multidisciplinary projects in whatsoever position they are employed.
12. **Life-long learning:** Recognize the need of self education and life-long learning process in order to keep abreast with the ongoing developments in the field of engineering.

PROGRAM SPECIFIC OUTCOMES:

At the end of the B. Tech. Program in Computer Science and Engineering (Data Science) the students shall:

PSO 01: [Problem Analysis]: Identify, formulate, research literature, and analyze complex engineering problems related to Data Science principles and practices, Programming and Computing technologies reaching substantiated conclusions using first principles of mathematics, natural sciences, and engineering sciences.

PSO 02: [Design/development of Solutions]: Design solutions for complex engineering problems related to Data Science principles and practices, Programming and Computing technologies and design system components or processes that meet t h e specified needs with appropriate consideration for the public health and safety, and the cultural, societal, and environmental considerations.

PSO 03: [Modern Tool usage]: Create, select, and apply appropriate techniques, resources, and modern engineering and IT tools including prediction and modelling to complex engineering activities related to Data Science principles and practices, Programming and Computing technologies with an understanding of the limitations

PREDICTIVE ANALYTICS LAB

B.Tech IV Year I Semester CSE-DS

L	T	P	C
0	0	2	1

Course outcomes:

1. Understand the processing steps for predictive analytics
2. Construct and deploy prediction models with integrity
3. Explore various techniques (machine learning/data mining, ensemble) for predictive analytics.
4. Apply predictive analytics to real world examples.

PAGE-5
CO-PO MAPPING

LIST OF EXPERIMENTS:

Following experiments to be carried out using Python/SPSS/SAS/R/Power BI

1. Simple Linear regression
2. Multiple Linear regression
3. Logistic Regression
4. CHAID
5. CART
6. ARIMA–stock market data
7. Exponential Smoothing
8. Hierarchical clustering
9. Ward's method of clustering
10. Crowd source predictive analytics- Netflix data

TEXT BOOKS:

1. Eric Siegel, Predictive analytics- the power to predict who will Click, buy, lie, or die, John Wiley & Sons, 2013.
2. Dean Abbott, Applied Predictive Analytics - Principles and Techniques for the Professional Data Analyst, 2014.

REFERENCE BOOKS:

1. Trevor Hastie, Robert Tibshirani, Jerome Friedman, The Elements of Statistical Learning-Data Mining, Inference, and Prediction, Second Edition, Springer Verlag, 2009.
2. G. James, D. Witten, T. Hastie, R. Tibshirani-An introduction to statistical learning with applications in R, Springer, 2013
3. E. Alpaydin, Introduction to Machine Learning, Prentice Hall of India, 2010

1. Simple Linear Regression

Theory: Simple Linear Regression predicts the relationship between two variables assuming a linear relation: **Y = mX + C.**

Steps for Implementing Simple Linear Regression in Python:

- **Import Libraries:**

  Import necessary libraries such as

  numpy for numerical operations,

  pandas for data handling,

  matplotlib.pyplot for visualization,

  sklearn.linear_model.LinearRegression for the regression model.

- **Prepare Data:**

  Load or generate your dataset. The data should consist of pairs of values for the independent variable (X) and the dependent variable (y). Reshape X if necessary to be a 2D array e.g., X.reshape(-1, 1).

- **Split Data (Optional but Recommended):**

  Divide your dataset into training and testing sets using sklearn.model_selection.train_test_split. This allows you to train the model on one portion of the data and evaluate its performance on unseen data.

- **Create and Fit the Model:**

  Initialize a LinearRegression object: model = LinearRegression().

  Fit the model to your training data: model.fit(X_train, y_train).

- **Make Predictions:**
  Use the trained model to predict values for the dependent variable on new or test data: y_pred = model.predict(X_test).

- **Evaluate the Model (Optional):**

  Assess the model's performance using metrics like Mean Squared Error (MSE) or R-squared, available in sklearn.metrics.

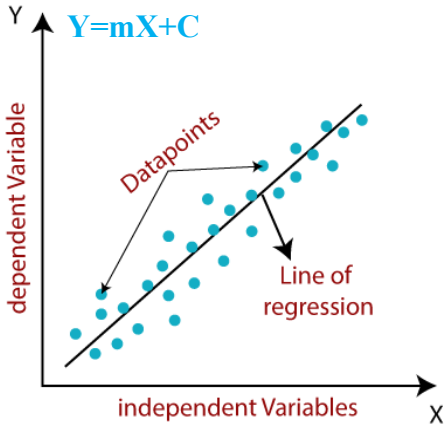
- **Visualize Results (Optional):**
  Plot the original data points and the regression line to visually inspect the fit.

Python Implementation:

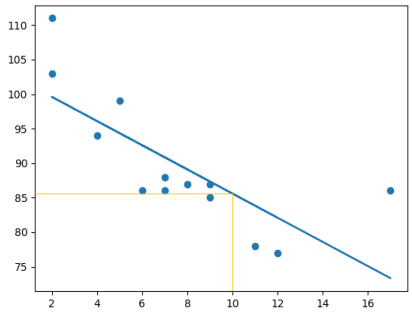
```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.linear_model import LinearRegression
X = np.array([1, 2, 3, 4, 5]).reshape(-1, 1)
Y = np.array([2, 3, 5, 6, 8])
model = LinearRegression()
model.fit(X, Y)
Y_pred = model.predict(X)
plt.scatter(X, Y, color='blue', label='Actual Data')
plt.plot(X, Y_pred, color='red', label='Regression Line')
plt.legend()
plt.show()
```

- **Expected Output:**

  A scatter plot with a regression line.



(Increasing data Value)



(Decreasing data values)

2. Multiple Linear Regression

Theory:

Multiple Linear Regression extends Simple Linear Regression to multiple independent variables.

Steps for Implementing Multiple Linear Regression in Python:

Import necessary Python **libraries.**

```
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error, r2_score
```

- **Prepare your data:** Load your dataset into a Pandas DataFrame. Define your independent variables (features, X) and your dependent variable (target, y).

```
# Assuming 'df' is your DataFrame, 'feature1', 'feature2' are independent, 'target' is dependent
X = df[['feature1', 'feature2', 'feature3']] # Include all relevant independent variables
y = df['target']
```

- **Split data into training and testing sets:**

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
```

- **Create and train the model.**

```
model = LinearRegression()
model.fit(X_train, y_train)
```

- **Make predictions.**

```
y_pred = model.predict(X_test)
```

- **Evaluate the model.**

```
mse = mean_squared_error(y_test, y_pred)
r2 = r2_score(y_test, y_pred)
print(f"Mean Squared Error: {mse}")
print(f"R-squared: {r2}")
```

- **Using statsmodels:** statsmodels provides a more detailed statistical output, including p-values and confidence intervals for coefficients. Import necessary libraries.

```
import statsmodels.api as sm
```

- **Prepare your data (similar to scikit-learn):**

  X = df[['feature1', 'feature2', 'feature3']]

  y = df['target']

  X = sm.add_constant(X) # Add a constant (intercept) term

- **Create and fit the OLS model.**

  model = sm.OLS(y, X).fit()

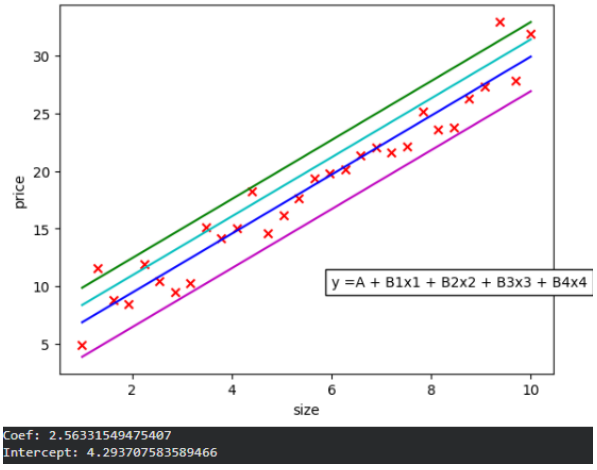
- **Print the model summary.**

  print(model.summary())

Python Implementation:

```
import numpy as np, matplotlib.pyplot as plt
from sklearn.linear_model import LinearRegression
X=np.linspace(1,10,30).reshape(-1,1);Y=2.5*X+np.random.randn(30,1)*2+5
m=LinearRegression().fit(X,Y);
p=m.predict(X)
for s,c in [(0,'b'),(3,'g'),(-3,'m'),(1.5,'c')]:plt.plot(X,p+s,c)
plt.scatter(X,Y,c='r',marker='x');
plt.xlabel('size');plt.ylabel('price')
eq=f"y =A + B1x1 + B2x2 + B3x3 + B4x4"
plt.text(6,10,eq,bbox=dict(facecolor='white',edgecolor='black'))
plt.show();print('Coef:',m.coef_[0][0]);print('Intercept:',m.intercept_[0])
```

Expected Output:



3. Logistic Regression

Theory: Logistic Regression is a supervised machine learning algorithmused for binary classification problems.

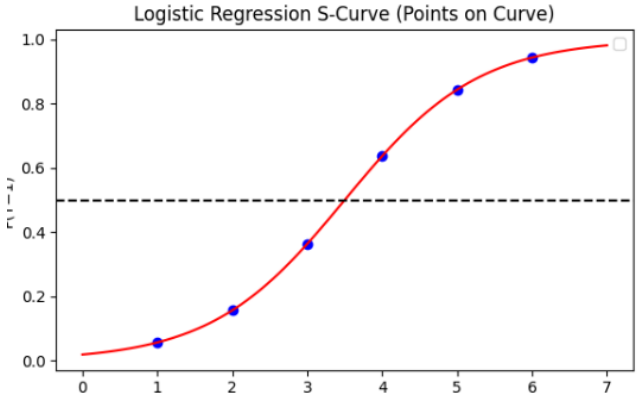
- **Input:** Logistic regression takes a set of independent variables (features) and predicts a categorical dependent variable.
- **Linear combination:** It first calculates a linear combination of the input features and their coefficients, similar to linear regression.
- **Sigmoid function:** This linear combination is then passed through a sigmoid (or logistic) function, which squashes the output into a probability that lies between 0 and 1.
- The formula for the sigmoid function is $s(x) = \frac{1}{1 + e^{-x}}$.
- **Prediction**: A threshold (commonly 0.5) is used to classify the output. If the probability is greater than the threshold, it's classified as one class (e.g., 1); otherwise, it's the other class (e.g., 0).

Python Implementation:

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.linear_model import LogisticRegression
X = np.array([[1],[2],[3],[4],[5],[6]])
Y = np.array([0,0,0,1,1,1])
model = LogisticRegression().fit(X, Y)
x_vals = np.linspace(0, 7, 100).reshape(-1, 1)
y_curve = model.predict_proba(x_vals)[:, 1]
y_points = model.predict_proba(X)[:, 1]
plt.figure(figsize=(6,4))
plt.plot(x_vals, y_curve, 'r',)
plt.scatter(X, y_points, c='b',)
plt.axhline(0.5, ls='--', c='k')
plt.xlabel('X'); plt.ylabel('P(Y=1)')
plt.title('Logistic Regression S-Curve (Points on Curve)')
plt.legend(); plt.tight_layout(); plt.show()
```

Expected Output:

Binary classification outputs (0 or 1).



4. CHAID (Chi-square Automatic Interaction Detector)

Theory: CHAID is used for decision tree analysis.

CHAID is a supervised machine learning algorithm that builds decision trees to find relationships between variables.

Chi-square is statistical measure to find the difference between the child and the parent node.

$$Chi - Square = \sqrt{\frac{(E-O)^2}{E}}$$

It works by recursively partitioning data using chi-square tests to determine the best split at each node, and it is particularly useful for segmentation and prediction with categorical variables.

STEPS:

- **Import necessary libraries:**
  `DecisionTreeClassifier` is imported from `sklearn.tree` for building the decision tree model.
  `numpy` is imported as `np` for creating numerical arrays.

- **Prepare data:**
  `X` is a NumPy array representing the features, where each inner array `[value]` is a single feature for a sample. In this case, it's a 1-dimensional feature set.
  `Y` is a NumPy array representing the target labels corresponding to `X`.

- **Initialize and train the model:**
  `model = DecisionTreeClassifier(criterion='gini')` creates an instance of the Decision Tree Classifier. The `criterion='gini'` specifies that the Gini impurity will be used to measure the quality of a split.
  `model.fit(X, Y)` trains the decision tree model using the provided features (`X`) and target labels (`Y`). The model learns the optimal splits to classify the data.

- **Retrieve and print the tree depth:**
  `model.get_depth()` is a method of the `DecisionTreeClassifier` object that returns the maximum depth of the trained decision tree.
  The result is then printed along with the label "Tree depth:"

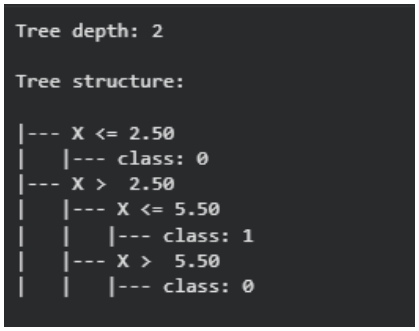
Python Implementation:

```
from sklearn.tree import DecisionTreeClassifier, export_text
import numpy as np
X = np.array([[1],[2],[3],[4],[5],[6]])
Y = np.array([0,0,1,1,1,0])
model = DecisionTreeClassifier(criterion='gini')
model.fit(X, Y)
print('Tree depth:', model.get_depth())
print('\nTree structure:\n')
print(export_text(model, feature_names=['X']))
```

Expected Output:

Tree depth and structure.

```
Tree depth: 2
```



5. CART (Classification And Regression Tree)

It can handle both Classification and Regression tasks.

It is a supervised learning algorithm that learns from labelled data to predict unseen data.

Scikit-Learn uses the Classification And Regression Tree (CART) algorithm to train Decision Trees (“growing” trees).

- **Classification Trees:** The tree is used to determine which "class" the target variable is most likely to fall into when it is continuous.
- **Regression trees:** These are used to predict a continuous variable's value.
- **Tree structure:** CART builds a tree-like structure consisting of nodes and branches. The nodes represent different decision points, and the branches represent the possible outcomes of those decisions. The leaf nodes in the tree contain a predicted class label or value for the target variable.
- **Splitting criteria:** CART uses a greedy approach to split the data at each node. It evaluates all possible splits and selects the one that best reduces the impurity of the resulting subsets. For classification tasks, CART uses Gini impurity as the splitting criterion. The lower the Gini impurity, the more pure the subset is. For regression tasks, CART uses residual reduction as the splitting criterion. The lower the residual reduction, the better the fit of the model to the data.
- **Pruning:** To prevent overfitting of the data, pruning is a technique used to remove the nodes that contribute little to the model accuracy.

 # CART for classification using Iris dataset

```
from sklearn.datasets import load_iris
from sklearn.tree import DecisionTreeClassifier, export_text

X, y = load_iris(return_X_y=True)
clf = DecisionTreeClassifier(max_depth=3, random_state=0)
clf.fit(X, y)
print("=== Classification Tree (Iris) ===")
print(export_text(clf, feature_names=load_iris().feature_names))
```

OUTPUT:

```
=== Classification Tree (Iris) ===
|--- petal width (cm) <= 0.80
|   |--- class: 0
|--- petal width (cm) >  0.80
|   |--- petal width (cm) <= 1.75
|   |   |--- petal length (cm) <= 4.95
|   |   |   |--- class: 1
|   |   |--- petal length (cm) >  4.95
|   |   |   |--- class: 2
|   |--- petal width (cm) >  1.75
|   |   |--- petal length (cm) <= 4.85
|   |   |   |--- class: 2
|   |   |--- petal length (cm) >  4.85
|   |   |   |--- class: 2
```

 # CART for regression using Diabetes dataset

```
from sklearn.datasets import load_diabetes
from sklearn.tree import DecisionTreeRegressor, export_text

X, y = load_diabetes(return_X_y=True)
reg = DecisionTreeRegressor(max_depth=3, random_state=0)
reg.fit(X, y)

print("=== Regression Tree (Diabetes) ===")
print(export_text(reg, feature_names=load_diabetes().feature_names))
```

OUTPUT:

```
=== Regression Tree (Diabetes) ===
|--- s5 <= -0.00
|   |--- bmi <= 0.01
|   |   |--- s3 <= 0.02
|   |   |   |--- value: [108.80]
|   |   |--- s3 >  0.02
|   |   |   |--- value: [83.37]
|   |--- bmi >  0.01
|   |   |--- age <= -0.08
|   |   |   |--- value: [274.00]
|   |   |--- age >  -0.08
|   |   |   |--- value: [154.67]
|--- s5 >  -0.00
|   |--- bmi <= 0.01
|   |   |--- bmi <= -0.02
|   |   |   |--- value: [137.69]
|   |   |--- bmi >  -0.02
|   |   |   |--- value: [176.86]
|   |--- bmi >  0.01
|   |   |--- bmi <= 0.07
|   |   |   |--- value: [208.57]
|   |   |--- bmi >  0.07
|   |   |   |--- value: [268.87]
```

6. ARIMA - Stock Market Prediction

Theory: ARIMA (AutoRegressive Integrated Moving Average) is a powerful statistical model used for **time series forecasting**, including **stock market prediction**. It captures trends, seasonality, and noise in sequential data. The ARIMA model combines three key components:

- **AR (AutoRegressive):** Uses past values (lags) of the variable to predict future values.
- **I (Integrated):** Represents differencing of raw observations to make the series stationary (remove trend).
- **MA (Moving Average):** Uses past forecast errors in a regression-like model to improve accuracy.

The model is represented as **ARIMA(p, d, q)** where:

- **p** → order of the AutoRegressive part (number of lag observations included)
- **d** → degree of differencing (number of times data is differenced to make it stationary)
- **q** → order of the Moving Average part (size of the moving average window)

ARIMA is widely used for forecasting financial and economic time series, where historical data is used to predict future values such as stock prices, demand, or sales.

Steps for Implementing ARIMA in Python:

1. **Import necessary libraries.**

   ```
   from statsmodels.tsa.arima.model import ARIMA
   import pandas as pd
   ```

2. **Load or prepare the time series data.**

   Create a pandas Series containing the stock price or time-dependent data.

   ```
   data = pd.Series([100, 102, 105, 110, 108, 107, 111, 115])
   ```

3. **Define and fit the ARIMA model.**

   Specify the order (p, d, q) based on the data characteristics.

   ```
   model = ARIMA(data, order=(2, 1, 2))
   model_fit = model.fit()
   ```

4. **View the model summary.**

   The summary displays the estimated coefficients, AIC (Akaike Information Criterion), and statistical significance of terms.

   ```
   print(model_fit.summary())
   ```

5. **Make predictions (optional).**

   Once trained, you can forecast future values:

   ```
   forecast = model_fit.forecast(steps=5)
   ```

print(forecast)

Python Implementation:

```
from statsmodels.tsa.arima.model import ARIMA
import pandas as pd
data = pd.Series([100, 102, 105, 110, 108, 107, 111, 115])
model = ARIMA(data, order=(2, 1, 2))
model_fit = model.fit()
print(model_fit.summary())
```

Expected Output:

ARIMA model summary.

```
                               SARIMAX Results
==============================================================================
Dep. Variable:                      y   No. Observations:                    8
Model:                 ARIMA(2, 1, 2)   Log Likelihood                -17.004
Date:                Sun, 02 Nov 2025   AIC                             44.008
Time:                        16:59:02   BIC                             43.737
Sample:                             0   HQIC                            40.665
                                  - 8
Covariance Type:                  opg
==============================================================================
                 coef    std err          z      P>|z|      [0.025      0.975]
------------------------------------------------------------------------------
ar.L1          0.5546      3.245      0.171      0.864      -5.805       6.914
ar.L2          0.4454      4.218      0.106      0.916      -7.822       8.713
ma.L1          0.0040    168.643   2.37e-05      1.000    -330.530     330.538
ma.L2         -0.9960      7.033     -0.142      0.887     -14.780      12.788
sigma2         5.1683     31.255      0.165      0.869     -56.090      66.426
===================================================================================
Ljung-Box (L1) (Q):                   0.14   Jarque-Bera (JB):                 3.19
Prob(Q):                              0.71   Prob(JB):                         0.20
Heteroskedasticity (H):               1.61   Skew:                            -1.56
Prob(H) (two-sided):                  0.77   Kurtosis:                         4.09
===================================================================================
```

7. Exponential Smoothing

Theory:

Exponential Smoothing is a simple forecasting technique for time series data that gives more weight to recent observations while gradually reducing the influence of older data. It helps smooth out random fluctuations and highlight trends. The smoothing factor α (alpha) controls how responsive the model is — higher α makes it react faster to recent changes, while lower α produces smoother results.

Steps for Implementation:

 Import necessary libraries.

```
import numpy as np

import matplotlib.pyplot as plt

from statsmodels.tsa.holtwinters import SimpleExpSmoothing
```

 Prepare the data.

Create a NumPy array or pandas Series containing time-dependent data.

```
data = np.array([10, 12, 13, 12, 14, 16, 18, 20, 21, 23])
```

 Create and fit the model.

Initialize the Simple Exponential Smoothing model and fit it with a chosen smoothing level (α).

```
model = SimpleExpSmoothing(data).fit(smoothing_level=0.3, optimized=False)
```

 Visualize the results.

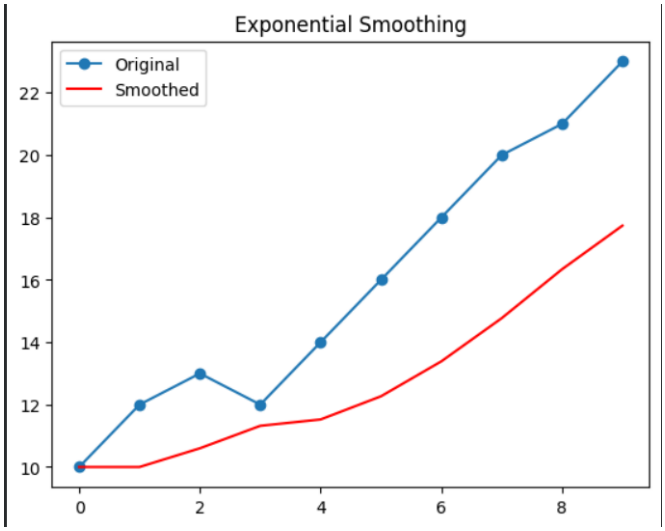
Plot both the original data and the smoothed values for comparison.

```
plt.plot(data, 'o-', label='Original')

plt.plot(model.fittedvalues, 'r-', label='Smoothed')

plt.legend()

plt.title('Exponential Smoothing')

plt.show()
```

Python Implementation:

```
import numpy as np, matplotlib.pyplot as plt

from statsmodels.tsa.holtwinters import SimpleExpSmoothing


data = np.array([10,12,13,12,14,16,18,20,21,23])

model = SimpleExpSmoothing(data).fit(smoothing_level=0.3, optimized=False)

plt.plot(data, 'o-', label='Original')

plt.plot(model.fittedvalues, 'r-', label='Smoothed')

plt.legend(); plt.title('Exponential Smoothing'); plt.show()
```

OUTPUT:



8. Hierarchical Clustering

Theory: Hierarchical Clustering groups similar data points into clusters.

Steps for Implementing Hierarchical Clustering in Python:

1. **Import necessary libraries.**

   ```
   import numpy as np
   import scipy.cluster.hierarchy as sch
   import matplotlib.pyplot as plt
   ```

2. **Prepare the dataset.**

   Create an array of data points to be clustered.

   ```
   X = np.array([[5,3],[10,15],[24,30],[30,40],[85,70],[71,80]])
   ```

3. **Compute linkage matrix.**

   The linkage function calculates the distance between clusters using a method like 'ward', 'single', 'complete', or 'average'.

   ```
   linkage_matrix = sch.linkage(X, method='ward')
   ```

4. **Plot the dendrogram.**

   The dendrogram visually represents how clusters are merged at each step.

   ```
   dendrogram = sch.dendrogram(linkage_matrix)
   plt.title('Hierarchical Clustering Dendrogram')
   plt.xlabel('Data Points')
   plt.ylabel('Euclidean Distance')
   plt.show()
   ```

5. **Decide the number of clusters.**

   By drawing a horizontal cut across the dendrogram, you can determine how many clusters to form based on distance threshold.

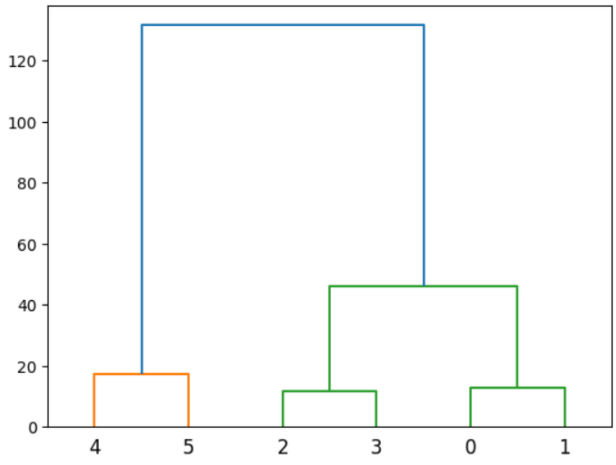
Python Implementation:

```
import numpy as np
import scipy.cluster.hierarchy as sch
import matplotlib.pyplot as plt
X = np.array([[5, 3], [10, 15], [24, 30], [30, 40], [85, 70], [71, 80]])
dendrogram = sch.dendrogram(sch.linkage(X, method='ward'))
```

```
plt.show()
```

Expected Output:

A dendrogram showing hierarchical clustering.



9. Principal Component Analysis (PCA)

Theory: PCA reduces dimensionality while preserving variance.

Steps for Implementing PCA in Python:

1. **Import necessary libraries.**

   from sklearn.decomposition import PCA

   import numpy as np

2. **Prepare the dataset.**

   Define or load the feature matrix containing correlated variables.

   X = np.array([[2,4],[3,6],[4,8],[5,10],[6,12]])

3. **Initialize the PCA model.**

   Choose the number of principal components to keep.

   pca = PCA(n_components=1)

4. **Fit and transform the data.**

   Reduce the dataset to its main component(s).

   X_reduced = pca.fit_transform(X)

   print('Reduced Data:', X_reduced)

Python Implementation:

```
from sklearn.decomposition import PCA
import numpy as np
X = np.array([[2, 4], [3, 6], [4, 8], [5, 10], [6, 12]])
pca = PCA(n_components=1)
X_reduced = pca.fit_transform(X)
print('Reduced Data:', X_reduced)
```

Expected Output:

Reduced dataset with principal components.

```
Reduced Data: [[-4.47213595]
 [-2.23606798]
 [ 0.        ]
 [ 2.23606798]
 [ 4.47213595]]
```

10.Crowdsourced Predictive Analytics - Netflix Data

Theory: Using machine learning for predicting user ratings in Netflix dataset.

Steps for Implementing Rating Prediction in Python:

1. **Import necessary libraries.**

   import pandas as pd

2. **Create or load the dataset.**

   Define user IDs, movie names, and corresponding ratings.

   data = {'User': [1, 2, 3, 4, 5],

   'Movie': ['A', 'B', 'C', 'A', 'B'],

   'Rating': [5, 4, 3, 5, 4]}

   df = pd.DataFrame(data)

3. **Explore the dataset.**

   Display the first few rows to verify structure and content.

   print(df.head())

4. **(Next steps in a full ML model)**
   - Preprocess data (handle missing values, encode categorical features).
   - Create a user–item matrix for collaborative filtering.
   - Apply ML algorithms such as K-Nearest Neighbors, Matrix Factorization, or Neural Networks to predict missing ratings.
   - Evaluate performance using metrics like RMSE or MAE.

Python Implementation:

```
import pandas as pd
data = {'User': [1, 2, 3, 4, 5], 'Movie': ['A', 'B', 'C', 'A', 'B'], 'Rating': [5, 4, 3,5, 4]}
df = pd.DataFrame(data)
print(df.head())
```

Expected Output:

A preview of Netflix-style rating data.

```
   User Movie  Rating
0     1     A       5
1     2     B       4
2     3     C       3
3     4     A       5
4     5     B       4
```