



Sri

# SAI RAM ENGINEERING COLLEGE

An Autonomous Institution | Affiliated to Anna University & Approved by AICTE, New Delhi

Accredited by NBA and NAAC 'A+' | BIS/EOMS ISO 21001 : 2018 and BVQI 9001 : 2015 Certified and NIRF ranked institution

Sai Leo Nagar, West Tambaram, Chennai - 600 044. [www.sairam.edu.in](http://www.sairam.edu.in)



## **24AMPW401 - MACHINE LEARNING TECHNIQUES WITH LABORATORY (II YEAR / IV SEM)**

**(BATCH: 2024 – 2028)**

**ACADEMIC YEAR: 2025 – 2026**

**EVEN SEMESTER**

**DEPARTMENT OF CSE (ARTIFICIAL INTELLIGENCE & MACHINE LEARNING)**

## **PREFACE**

### **“PRACTICE LEADS TO PERFECTION”**

Practical work encourages and familiarizes students with the latest tools that will be required to become experts. Taking this into consideration, this manual is compiled as a preparatory note for the **MACHINE LEARNING TECHNIQUES WITH LABORATORY** programs. Sufficient details have been included to impart self- learning.

This manual is intended for the IV semester CSE (AI&ML) students under Autonomous Regulations 2024. Introductory information and procedure to perform is detailed in this manual.

It is expected that this will help the students develop a broad understanding and learn quick adaptations to real-time problems.

## **INSTITUTE VISION**

To emerge as a “Centre of excellence” offering Technical Education and Research opportunities of very high standards to students, develop the total personality of the individual and instill high levels of discipline and strive to set global standards, making our students technologically superior and ethically stronger, who in turn shall contribute to the advancement of society and humankind.

## **INSTITUTION MISSION**

We dedicate and commit ourselves to achieve, sustain and foster unmatched excellence in Technical Education. To this end, we will pursue continuous development of infra-structure and enhance state-of-art equipment to provide our students a technologically up-to date and intellectually inspiring environment of learning, research, creativity, innovation and professional activity and inculcate in them ethical and moral values.

## **INSTITUTE POLICY**

We at Sri Sai Ram Engineering College are committed to build a better Nation through Quality Education with team spirit. Our students are enabled to excel in all values of Life and become Good Citizens. We continually improve the System, Infrastructure and Service to satisfy the Students, Parents, Industry and Society.

## **DEPARTMENT VISION**

To emerge as a “Centre of Excellence” in the field of Artificial Intelligence and Machine Learning by providing required skill sets, domain expertise and interactive industry interface for students and shape them to be a socially conscious and responsible citizen.

## **DEPARTMENT MISSION**

Computer Science and Engineering (Artificial Intelligence and Machine Learning), Sri Sairam Engineering College is committed to

**M1:** Nurture students with a sound understanding of fundamentals, theory and practice of Artificial Intelligence and Machine Learning.

**M2:** Develop students with the required skill sets and enable them to take up assignments in the field of AI & ML

**M3:** Facilitate Industry Academia interface to update the recent trends in AI &ML

**M4:** Create an appropriate environment to bring out the latent talents, creativity and innovation among students to contribute to the society

## **Program Educational Objectives(PEOs)**

**To prepare the graduates to:**

1. Graduates imbibe fundamental knowledge in Artificial Intelligence, Programming, Mathematical modelling and Machine Learning.
2. Graduates will be trained to gain domain expertise by applying the theory basics into practical situation through simulation and modelling techniques.
3. Graduates will enhance the capability through skill development and make them industry ready by inculcating leadership and multitasking abilities.
4. Graduates will apply the gained knowledge of AI & ML in Research & Development, Innovation and contribute to the society in making things simpler.

## **Program Outcomes (PO's)**

The graduates in Engineering will:

**PO1: Engineering Knowledge:** Apply knowledge of mathematics, natural science, computing, engineering fundamentals and an engineering specialization as specified in WK1 to WK4 respectively to develop to the solution of complex engineering problems.

**PO2: Problem Analysis:** Identify, formulate, review research literature and analyze complex engineering problems reaching substantiated conclusions with consideration for sustainable development. (WK1 to WK4)

**PO3: Design/Development of Solutions:** Design creative solutions for complex engineering problems and design/develop systems/components/processes to meet identified needs with consideration for the public health and safety, whole-life cost, net zero carbon, culture, society and environment as required. (WK5)

**PO4: Conduct Investigations of Complex Problems:** Conduct investigations of complex engineering problems using research-based knowledge including design of experiments, modelling, analysis & interpretation of data to provide valid conclusions. (WK8).

**PO5: Engineering Tool Usage:** Create, select and apply appropriate techniques, resources and modern engineering & IT tools, including prediction and modelling recognizing their limitations to solve complex engineering problems. (WK2 and WK6)

**PO6: The Engineer and The World:** Analyze and evaluate societal and environmental aspects while solving complex engineering problems for its impact on sustainability with reference to economy, health, safety, legal framework, culture and environment. (WK1, WK5, and WK7).

**PO7: Ethics:** Apply ethical principles and commit to professional ethics, human values, diversity and inclusion; adhere to national & international laws. (WK9)

**PO8: Individual and Collaborative Team work:** Function effectively as an individual, and as a member or leader in diverse/multi-disciplinary teams.

**PO9: Communication:** Communicate effectively and inclusively within the engineering community and society at large, such as being able to comprehend and write effective reports and design documentation, make effective presentations considering cultural, language, and learning differences.

**PO10: Project Management and Finance:** Apply knowledge and understanding of engineering management principles and economic decision-making and apply these to one's own work, as a member and leader in a team, and to manage projects and in multidisciplinary environments.

**PO11: Life-Long Learning:** Recognize the need for, and have the preparation and ability for i) independent and life-long learning ii) adaptability to new and emerging technologies and iii) critical thinking in the broadest context of technological change. (WK8)

### **Program Specific Outcomes (PSOs)**

Computer Science and Engineering (Artificial Intelligence and Machine Learning), graduates will be able to:

1. The graduates will be in a position to design, develop, test and deploy appropriate mathematical and programming algorithms required for practical applications.
2. The graduates will have the required skills and domain expertise to provide solutions in the field of Artificial Intelligence and Machine Learning for the Industry and society at large.

## **COURSE OUTCOMES**

Upon completion of the course, the students will be able to :

<b>CO1</b>	Utilize appropriate machine learning techniques to solve real-world problems across various domains. (K3)
<b>CO2</b>	Apply the dimensionality reduction techniques to reduce data complexity and improve model performance. (K3)
<b>CO3</b>	Make use of data preprocessing and feature engineering techniques to refine datasets for better model performance. (K3)
<b>CO4</b>	Select suitable supervised classification algorithms to effectively address real-world problem scenarios.(K3)
<b>CO5</b>	Examine supervised regression methods and assess their performance for predictive modelling tasks. (K4)
<b>CO6</b>	Analyze clustering algorithms to uncover patterns and groupings in unlabeled data. (K4)

## **24AMPW401- MACHINE LEARNING TECHNIQUES WITH LABORATORY**

### **OBJECTIVES**

- To teach the theoretical foundations of various machine learning techniques.
- To learn the dimensionality reduction techniques for feature engineering.
- To know about the data pre-processing and feature engineering approaches.
- To understand the supervised classification and regression algorithms for solving real-world problems.
- To apply the clustering algorithm techniques for solving real-world problems.
- To explore Unsupervised learning and clustering methods.

### **LIST OF EXPERIMENTS**

- 1) Explore Numpy and Pandas libraries for data pre-processing techniques such as handling missing data and outliers, encoding to prepare data for machine learning models.
- 2) Implement Linear Regression to predict a continuous target variable and explore how Multiple Linear Regression can capture the relationship between multiple predictors and the target. Use GHI dataset to perform model training, validation, and performance evaluation.
- 3) Apply Polynomial Regression to model non-linear relationships between the input variables and the target variable. Investigate how the degree of the polynomial impacts the model's ability to fit GHI data, avoiding overfitting or underfitting.
- 4) Learn how to apply Naïve Bayes and K-Nearest Neighbors (KNN) supervised learning techniques to identify Level of experience.
- 5) Implement a Decision Tree classifier technique to identify whether the cars require service or not and plot the confusion Matrix using seaborn library.
- 6) Implementation of Support Vector Machine technique to identify user behavior classes and plot the confusion Matrix using seaborn library.
- 7) Apply the Random Forest algorithm to identify Classification of users based on driving habits (e.g., Commuter, Long-Distance Traveler) and plot the confusion Matrix using seaborn library.

- 8) Implementation of AdaBoost technique to Classify loan approval and loan not approval and plot the confusion Matrix using seaborn library.
- 9) Implementation of XGBoost and LightGBM technique to Classify Waitlist and admit in Wharton Class of 2025's statistics and plot the confusion Matrix using seaborn library.
- 10) Implementation of K-means and Hierarchical Clustering on customer segmentation dataset and plot the elbow and dendrogram for k-means and Hierarchical clustering respectively.



## INDEX

S. No	Title of the Experiment	Page No.
1	Explore Numpy and Pandas libraries for data pre-processing techniques such as handling missing data and outliers, encoding to prepare data for machine learning models.	
2	Implement Linear Regression to predict a continuous target variable and explore how Multiple Linear Regression can capture the relationship between multiple predictors and the target. Use GHI dataset to perform model training, validation, and performance evaluation.	
3	Apply Polynomial Regression to model non-linear relationships between the input variables and the target variable. Investigate how the degree of the polynomial impacts the model's ability to fit GHI data, avoiding overfitting or underfitting.	
4	Learn how to apply Naïve Bayes and K-Nearest Neighbors (KNN) supervised learning techniques to identify Level of experience.	
5	Implement a Decision Tree classifier technique to identify whether the cars require service or not and plot the confusion Matrix using seaborn library.	
6	Implementation of Support Vector Machine technique to identify user behavior classes and plot the confusion Matrix using seaborn library.	
7	Apply the Random Forest algorithm to identify Classification of users based on driving habits (e.g., Commuter, Long-Distance Traveler) and plot the confusion Matrix using seaborn library.	
8	Implementation of AdaBoost technique to Classify loan approval and loan not approval and plot the confusion Matrix using seaborn library.	
9	Implementation of XGBoost and LightGBM technique to Classify Waitlist and admit in Wharton Class of 2025's statistics and plot the confusion Matrix using seaborn library.	
10	Implementation of K-means and Hierarchical Clustering on customer segmentation dataset and plot the elbow and dendrogram for k-means and Hierarchical clustering respectively.	



<b>Ex.No:1</b>	<b>Explore Numpy and Pandas libraries for data pre-processing techniques such as handling missing data and outliers, encoding to prepare data for machine learning models.</b>
----------------	--

## INTRODUCTION

Data preprocessing is a crucial step in machine learning that involves cleaning, transforming, and organizing raw data to improve the quality and accuracy of models. NumPy provides efficient numerical operations, while Pandas offers powerful tools for handling missing values, outliers, and categorical encoding.

## AIM

To implement data preprocessing using NumPy and Pandas by handling missing data, outliers, and encoding categorical variables.

## ALGORITHM

### A. Handling Missing Data

1. Load the dataset into a Pandas DataFrame.
2. Identify missing values using `isnull()` or `isna()`.
3. Replace missing numerical values using mean or median.
4. Replace missing categorical values using mode.
5. Alternatively, remove rows/columns containing missing data using `dropna()`.

### B. Detecting & Handling Outliers (IQR Method)

1. Select numerical columns with possible outliers.
2. Calculate Q1 (25th percentile) and Q3 (75th percentile).
3. Compute  $IQR = Q3 - Q1$ .
4. Determine lower bound =  $Q1 - 1.5 \times IQR$ .
5. Determine upper bound =  $Q3 + 1.5 \times IQR$ .
6. Cap, replace, or remove values outside these bounds.

### C. Encoding Categorical Data

1. Identify categorical columns in the dataset.
2. Apply **Label Encoding** for ordinal categories.
3. Apply **One-Hot Encoding** for nominal categories using `pd.get_dummies()`.
4. Merge encoded columns with the numerical dataset.

### D. Final Dataset Preparation

1. Verify all missing values and outliers are handled.
2. Combine processed numerical and categorical features.
3. Use the cleaned dataset for machine learning model training.

## Program

```
import numpy as np
import pandas as pd

# Sample dataset
data = {
    "Age": [25, 28, np.nan, 32, 100, 29],
    "Salary": [50000, np.nan, 45000, 52000, 51000, 700000],
    "Department": ["HR", "IT", "IT", np.nan, "Finance", "HR"]
}

df = pd.DataFrame(data)
print("Original Data:")
print(df)

# -----
# Handling Missing Data
# -----
df["Age"].fillna(df["Age"].mean(), inplace=True)
df["Salary"].fillna(df["Salary"].median(), inplace=True)
df["Department"].fillna(df["Department"].mode()[0], inplace=True)

# -----
# Outlier Treatment (IQR method)
# -----
Q1 = df["Salary"].quantile(0.25)
Q3 = df["Salary"].quantile(0.75)
IQR = Q3 - Q1
lower = Q1 - 1.5 * IQR
upper = Q3 + 1.5 * IQR

# Cap outliers
df["Salary"] = np.where(df["Salary"] > upper, upper,
                        np.where(df["Salary"] < lower, lower, df["Salary"]))

# -----
# Encoding Categorical Data
# -----
df_encoded = pd.get_dummies(df, columns=["Department"])
print("\nProcessed Data:")
print(df_encoded)
```

## OUTPUT

Original Data:

	Age	Salary	Department
0	25.0	50000.0	HR
1	28.0	NaN	IT
2	NaN	45000.0	IT
3	32.0	52000.0	NaN
4	100.0	51000.0	Finance
5	29.0	700000.0	HR

Processed Data:

	Age	Salary	Department_Finance	Department_HR	Department_IT
0	25.0	50000.0	False	True	False
1	28.0	51000.0	False	False	True
2	42.8	48000.0	False	False	True
3	32.0	52000.0	False	True	False
4	100.0	51000.0	True	False	False
5	29.0	54000.0	False	True	False

## RESULT

The dataset was successfully pre-processed by handling missing values, treating outliers, encoding categorical variables, and generating a clean dataset ready for machine learning.

<b>Ex.No:2</b>	<b>Implement Linear Regression to predict a continuous target variable and explore how Multiple Linear Regression can capture the relationship between multiple predictors and the target. Use GHI dataset to perform model training, validation, and performance evaluation.</b>
----------------	---

## INTRODUCTION

Linear Regression is a supervised machine learning technique used to predict a continuous target variable by establishing a linear relationship between the input feature(s) and the output. In Simple Linear Regression, the model uses only one predictor variable, and it fits a straight-line equation of the form:

$$y = mX + c$$

Here, the slope ( $m$ ) represents how the target changes with the predictor, while the intercept ( $c$ ) indicates the starting value when the predictor is zero. However, real-world problems often involve more than one feature. Multiple Linear Regression extends this concept by using two or more predictor variables to capture more complex relationships. Its equation is:

$$y = b_0 + b_1X_1 + b_2X_2 + \dots + b_nX_n$$

This allows the model to learn how each independent variable contributes to the target value.

In this experiment, the GHI (Global Hunger Index) dataset is used to train, validate, and evaluate both Simple Linear Regression and Multiple Linear Regression models to predict the GHI Score based on various health and nutrition indicators.

## AIM

To implement Simple Linear Regression and Multiple Linear Regression techniques to predict the Global Hunger Index(GHI) using key hunger and nutrition indicators.

## ALGORITHM

1. Imported the required Python libraries such as NumPy, Pandas, Matplotlib, and Scikit-learn.
2. Generated a synthetic Global Hunger Index dataset containing indicators such as undernourishment, child wasting, child stunting, and child mortality.
3. Computed the Global Hunger Index value using a weighted combination of the hunger indicators with added random noise.
4. Selected undernourishment as the independent variable and Global Hunger Index as the dependent variable for Simple Linear Regression.
5. Split the dataset into training and testing sets using an 80:20 ratio.
6. Trained the Simple Linear Regression model using the training dataset.
7. Predicted Global Hunger Index values for the test dataset and evaluated the model using MAE, MSE, and R<sup>2</sup> score.
8. Selected undernourishment, child wasting, child stunting, and child mortality as independent variables for Multiple Linear Regression.
9. Trained the Multiple Linear Regression model using the training dataset.

10. Predicted Global Hunger Index values for the test dataset and evaluated the model performance.

## PROGRAM

```
# Import libraries
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_absolute_error, mean_squared_error, r2_score

# Step 1: Create Synthetic Global Hunger Index Dataset
np.random.seed(42)
data = pd.DataFrame({
    "Undernourishment": np.random.uniform(5, 40, 200),    # %
    "ChildWasting": np.random.uniform(2, 25, 200),      # %
    "ChildStunting": np.random.uniform(10, 50, 200),     # %
    "ChildMortality": np.random.uniform(1, 15, 200)      # %
})
data["GHI"] = (
    0.25 * data["Undernourishment"]
    + 0.25 * data["ChildWasting"]
    + 0.25 * data["ChildStunting"]
    + 0.25 * data["ChildMortality"]
    + np.random.normal(0, 1.5, 200)
)

# Display dataset
print("Synthetic Global Hunger Index Dataset:")
print(data.head())

# Step 2: Simple Linear Regression
X_simple = data[['Undernourishment']]
y = data['GHI']
X_train_s, X_test_s, y_train_s, y_test_s = train_test_split(
    X_simple, y, test_size=0.2, random_state=42
)

simple_lr = LinearRegression()
simple_lr.fit(X_train_s, y_train_s)
y_pred_s = simple_lr.predict(X_test_s)
```

```

print("\n--- Simple Linear Regression (Undernourishment → GHI) ---")
print("MAE:", mean_absolute_error(y_test_s, y_pred_s))
print("MSE:", mean_squared_error(y_test_s, y_pred_s))
print("R2 Score:", r2_score(y_test_s, y_pred_s))

# Step 3: Multiple Linear Regression
X_multi = data[['Undernourishment', 'ChildWasting', 'ChildStunting', 'ChildMortality']]

X_train_m, X_test_m, y_train_m, y_test_m = train_test_split(
    X_multi, y, test_size=0.2, random_state=42
)
multi_lr = LinearRegression()
multi_lr.fit(X_train_m, y_train_m)
y_pred_m = multi_lr.predict(X_test_m)

print("\n--- Multiple Linear Regression (All Indicators → GHI) ---")
print("MAE:", mean_absolute_error(y_test_m, y_pred_m))
print("MSE:", mean_squared_error(y_test_m, y_pred_m))
print("R2 Score:", r2_score(y_test_m, y_pred_m))

# Step 4: Plot Actual vs Predicted GHI
plt.figure()
plt.scatter(y_test_m, y_pred_m)
plt.xlabel("Actual Global Hunger Index")
plt.ylabel("Predicted Global Hunger Index")
plt.title("Actual vs Predicted Global Hunger Index")
plt.show()

```

## OUTPUT :

Synthetic Global Hunger Index Dataset:

	Undernourishment	ChildWasting	ChildStunting	ChildMortality	GHI
0	18.108904	16.766728	14.124955	3.365091	11.905708
1	38.275001	3.935219	46.102116	4.900265	24.010353
2	30.619788	5.717460	30.210095	3.478147	20.329409
3	25.953047	22.666746	43.058299	2.241835	25.498112
4	10.460652	15.947868	22.801984	2.688902	15.364632

--- Simple Linear Regression (Undernourishment → GHI) ---

MAE: 3.041983782620014

MSE: 14.956599744135037

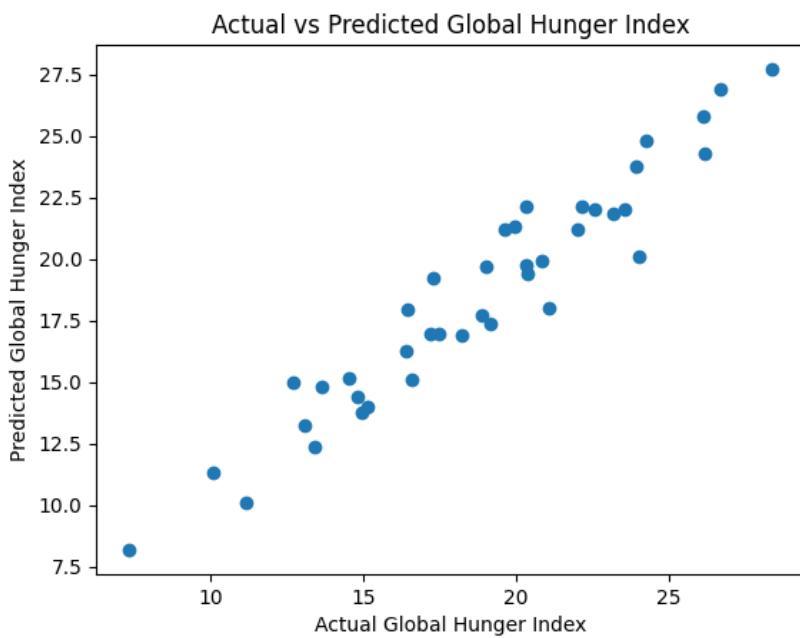
R2 Score: 0.34734049973251757

--- Multiple Linear Regression (All Indicators → GHI) ---

MAE: 1.1059453765970535

MSE: 1.8504677196206836

R2 Score: 0.9192513433661731



## RESULT

Thus, Simple Linear Regression and Multiple Linear Regression models were successfully implemented on the GHI dataset, and Multiple Linear Regression produced better prediction accuracy by considering multiple input variables.

<b>Ex.No:3</b>	<b>Apply Polynomial Regression to model non-linear relationships between the input variables and the target variable. Investigate how the degree of the polynomial impacts the model's ability to fit GHI data, avoiding overfitting or underfitting.</b>
----------------	---

## INTRODUCTION

Polynomial Regression is an extension of Linear Regression that allows modeling of non-linear relationships between the independent variables and the dependent variable. While Linear Regression fits a straight line, Polynomial Regression introduces higher-degree terms of the predictors, enabling the model to capture curves and more complex patterns in the data.

For the Global Hunger Index (GHI), the relationship between hunger indicators such as undernourishment, child wasting, child stunting, and child mortality and the overall index may not always be strictly linear. Polynomial Regression can help model such non-linear relationships and improve prediction accuracy. Choosing the right polynomial degree is important to avoid underfitting (too simple) or overfitting (too complex).

## AIM

To apply Polynomial Regression to model the non-linear relationship between input features and the target variable and to study the effect of polynomial degree on underfitting and overfitting.

## ALGORITHM

1. Import necessary Python libraries: NumPy, Pandas, Matplotlib, and Scikit-learn.
2. Generate a synthetic Global Hunger Index dataset with indicators: undernourishment, child wasting, child stunting, and child mortality.
3. Define the target variable GHI using a non-linear combination of the indicators.
4. Select one independent variable (e.g., Undernourishment) for simplicity in polynomial regression visualization.
5. Split the dataset into training and testing sets.
6. Transform the independent variable into polynomial features of the chosen degree.
7. Train a Linear Regression model on the polynomial features.
8. Predict GHI values for the test dataset.
9. Evaluate the model using Mean Squared Error (MSE), Mean Absolute Error (MAE), and R<sup>2</sup> score.
10. Repeat for different polynomial degrees to study underfitting and overfitting.

## PROGRAM

```
# Import libraries
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from sklearn.linear_model import LinearRegression
from sklearn.preprocessing import PolynomialFeatures
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_absolute_error, mean_squared_error, r2_score
```

```

# Step 1: Create synthetic Global Hunger Index dataset
np.random.seed(42)
data = pd.DataFrame({
    "Undernourishment": np.random.uniform(5, 40, 200),
    "ChildWasting": np.random.uniform(2, 25, 200),
    "ChildStunting": np.random.uniform(10, 50, 200),
    "ChildMortality": np.random.uniform(1, 15, 200)
})

# GHI as target variable (non-linear relationship)
data["GHI"] = (
    0.5 * data["Undernourishment"]**2
    - 0.3 * data["ChildWasting"]
    + 0.2 * data["ChildStunting"]
    + 0.1 * data["ChildMortality"]**2
    + np.random.normal(0, 2, 200)
)

# Step 2: Select feature and target
X = data[['Undernourishment']]
y = data['GHI']

# Step 3: Train-test split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Step 4: Apply Polynomial Regression for different degrees
degrees = [2, 3, 4] # Example degrees
for deg in degrees:
    poly = PolynomialFeatures(degree=deg)
    X_train_poly = poly.fit_transform(X_train)
    X_test_poly = poly.transform(X_test)

    model = LinearRegression()
    model.fit(X_train_poly, y_train)
    y_pred = model.predict(X_test_poly)

    print(f"\nPolynomial Degree: {deg}")
    print("MAE:", mean_absolute_error(y_test, y_pred))
    print("MSE:", mean_squared_error(y_test, y_pred))
    print("R2 Score:", r2_score(y_test, y_pred))

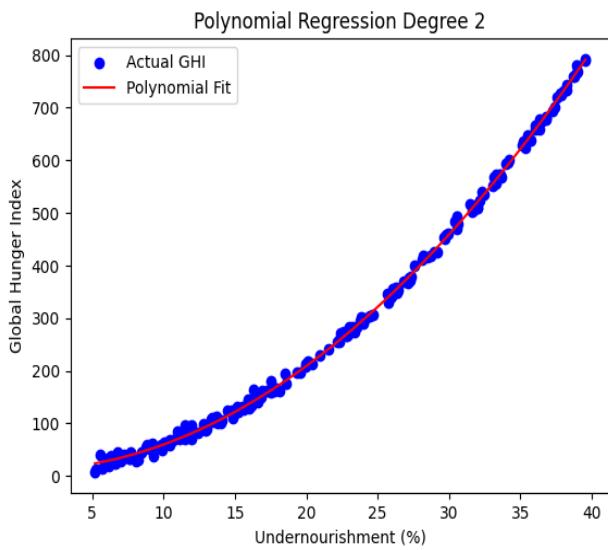
```

```
# Plot
X_plot = np.linspace(X.min(), X.max(), 200).reshape(-1,1)
X_plot_poly = poly.transform(X_plot)
y_plot = model.predict(X_plot_poly)

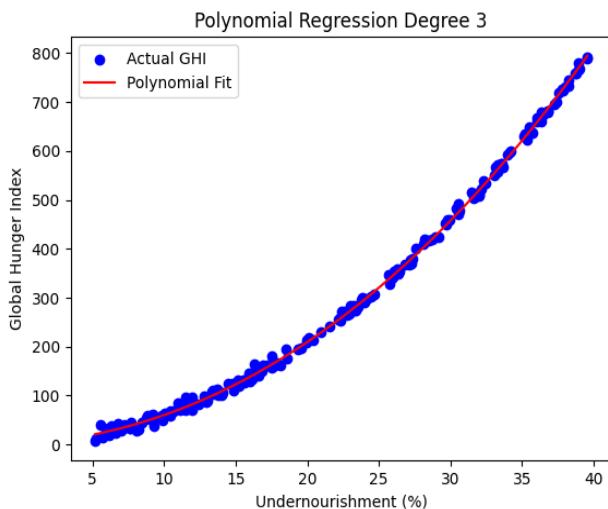
plt.figure()
plt.scatter(X, y, color='blue', label='Actual GHI')
plt.plot(X_plot, y_plot, color='red', label='Polynomial Fit')
plt.xlabel("Undernourishment (%)")
plt.ylabel("Global Hunger Index")
plt.title(f"Polynomial Regression Degree {deg}")
plt.legend()
plt.show()
```

## Output

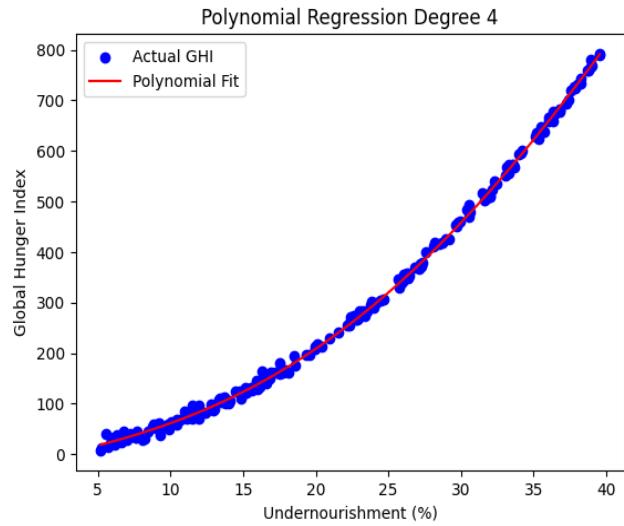
**Polynomial Degree:** 2  
**MAE:** 8.016642169039732  
**MSE:** 86.67263045315755  
**R2 Score:** 0.9984125977139668



**Polynomial Degree:** 3  
**MAE:** 8.006337541687225  
**MSE:** 85.8495575145922  
**R2 Score:** 0.9984276722289253



**Polynomial Degree: 4**  
**MAE: 7.941560255854671**  
**MSE: 85.90949759470602**  
**R2 Score: 0.998426574431158**



## RESULT

Thus Polynomial Regression was successfully modeled the non-linear relationship, and an optimal polynomial degree achieved minimum validation error by balancing underfitting and overfitting.

# **1. Write a Python program to implement a binary search tree with operations for inserting, deleting, and searching for elements.**

## **AIM**

To Write a Python program to implement a binary search tree with operations for inserting, deleting, and searching for elements.

### **Binary Search Tree (BST) – Overview A**

BST is a binary tree where:

- For any node:
  - Values in the left subtree are less than the node's value
  - Values in the right subtree are greater

## **Algorithms**

### **1. Insert(value)**

- If tree is empty, insert at root
- Else:
  - Recursively go left if  $\text{value} < \text{current node}$
  - Go right if  $\text{value} > \text{current node}$
  - Insert at the correct position (no duplicates here)

### **2. Search(value)**

- If node is None, return False
- If  $\text{value} == \text{node.data} \rightarrow \text{found}$
- If  $\text{value} < \text{node.data} \rightarrow \text{search left}$
- Else  $\rightarrow \text{search right}$

### **3. Delete(value)**

- Three cases:
  1. Node has no children (leaf)  $\rightarrow$  Remove directly
  2. Node has one child  $\rightarrow$  Replace with child

3. Node has two children → Replace with in-order successor (smallest in right subtree)

## PROGRAM

```
python CopyEdit
```

```
# Define the node class class
```

```
Node:
```

```
def __init__(self, key):  
    self.key = key  
    self.left = None  
    self.right = None
```

```
# Define the BST class class
```

```
BST:
```

```
def __init__(self): self.root = None
```

```
# Insert a node
```

```
def insert(self, root, key): if  
    root is None:  
        return Node(key) if  
        key < root.key:  
            root.left = self.insert(root.left, key) elif  
            key > root.key:  
                root.right = self.insert(root.right, key) return  
root
```

```
# Search for a node

def search(self, root, key):

    if root is None or root.key == key: return

        root

    if key < root.key:

        return self.search(root.left, key)

    else:

        return self.search(root.right, key)
```

```
# Find minimum value node (used in delete) def

minValueNode(self, node):

    current = node while

        current.left:

            current = current.left return

    current
```

```
# Delete a node

def delete(self, root, key): if

    root is None:

        return root

    if key < root.key:

        root.left = self.delete(root.left, key) elif

        key > root.key:

            root.right = self.delete(root.right, key) else:

                # Node found

                if root.left is None:
```

```
    return root.right elif
root.right is None:
    return root.left
# Node with two children
temp = self.minValueNode(root.right)
root.key = temp.key
root.right = self.delete(root.right, temp.key) return
root
```

```
# In-order traversal (to display the BST) def
inorder(self, root):
    if root:
        self.inorder(root.left)
        print(root.key, end=' ')
        self.inorder(root.right)
```

```
# Example usage
if __name__ == "__main__":
    tree = BST()
    root = None
    keys = [50, 30, 20, 40, 70, 60, 80]
```

```
# Insert nodes for
key in keys:
    root = tree.insert(root, key)

print("In-order traversal of the BST:")
```

```

tree.inorder(root) # Output should be sorted: 20 30 40 50 60 70 80

# Search

print("\n\nSearch 60:", "Found" if tree.search(root, 60) else "Not found")
print("Search 90:", "Found" if tree.search(root, 90) else "Not found")

# Delete a node

print("\n\nDeleting 20 (leaf node)...") root
= tree.delete(root, 20) tree.inorder(root)

print("\n\nDeleting 30 (one child)...")
root = tree.delete(root, 30)
tree.inorder(root)

print("\n\nDeleting 50 (two children)...")
root = tree.delete(root, 50) tree.inorder(root)

```

Output:

sql CopyEdit

In-order traversal of the BST: 20

30 40 50 60 70 80

Search 60: Found Search

90: Not found

Deleting 20 (leaf node)... 30

40 50 60 70 80

Deleting 30 (one child)... 40

50 60 70 80

Deleting 50 (two children)... 40

60 70 80

## **RESULT**

To Write a Python program to implement a binary search tree with operations for inserting, deleting, and searching for elements is verified.

## **2. Implement an AVL tree (a self-balancing binary search tree) in Python, with node insertion and automatic balancing using rotations to maintain the AVL property.**

### **AIM**

To Implement an AVL tree (a self-balancing binary search tree) in Python, with node insertion and automatic balancing using rotations to maintain the AVL property.

### **Algorithm**

Step-by-step:

1. Perform a standard BST insertion
2. Update the height of each node during recursion
3. Compute the balance factor:

ini CopyEdit

balance = height(left subtree) - height(right subtree)

4. If the balance factor is out of range (-1, 0, 1), fix it using rotations:

Four Cases:

- Left Left (LL) → Right Rotate
- Right Right (RR) → Left Rotate
- Left Right (LR) → Left Rotate + Right Rotate
- Right Left (RL) → Right Rotate + Left Rotate

### **Python Code**

python CopyEdit

# Node class class

AVLNode:

```
def __init__(self, key): self.key
```

```
= key
```

```
    self.left = None self.right  
    = None  
    self.height = 1 # New node is initially added at leaf
```

```
# AVL Tree class class
```

```
AVLTree:
```

```
# Get height of a node  
def get_height(self, node): if  
    not node:  
        return 0  
    return node.height
```

```
# Get balance factor  
def get_balance(self, node): if  
    not node:  
        return 0  
    return self.get_height(node.left) - self.get_height(node.right)
```

```
# Right rotation  
def right_rotate(self, z): y =  
    z.left  
    T3 = y.right
```

```
# Perform rotation  
    y.right = z  
    z.left = T3
```

```
# Update heights  
z.height = 1 + max(self.get_height(z.left), self.get_height(z.right))  
y.height = 1 + max(self.get_height(y.left), self.get_height(y.right))  
  
return y
```

```
# Left rotation  
def left_rotate(self, z): y =  
    z.right  
    T2 = y.left  
  
    # Perform rotation y.left  
    = z  
    z.right = T2
```

```
# Update heights  
z.height = 1 + max(self.get_height(z.left), self.get_height(z.right))  
y.height = 1 + max(self.get_height(y.left), self.get_height(y.right))  
  
return y
```

```
# Insert node  
def insert(self, root, key):  
    # 1. Normal BST insertion if  
    not root:  
        return AVLNode(key)
```

```

        elif key < root.key:
            root.left = self.insert(root.left, key)
        elif key > root.key:
            root.right = self.insert(root.right, key)
        else:
            return root # Duplicate keys not allowed

# 2. Update height
root.height = 1 + max(self.get_height(root.left), self.get_height(root.right))

# 3. Check balance factor balance =
self.get_balance(root)

# 4. Balance the tree #
Case 1 - Left Left
if balance > 1 and key < root.left.key:
    return self.right_rotate(root)

# Case 2 - Right Right
if balance < -1 and key > root.right.key:
    return self.left_rotate(root)

# Case 3 - Left Right
if balance > 1 and key > root.left.key:
    root.left = self.left_rotate(root.left)
    return self.right_rotate(root)

```

```
# Case 4 - Right Left

if balance < -1 and key < root.right.key:

    root.right = self.right_rotate(root.right)

    return self.left_rotate(root)

return root
```

```
# Inorder traversal (sorted) def

inorder(self, root):

    if root:

        self.inorder(root.left)

        print(root.key, end=' ')

        self.inorder(root.right)
```

```
# Example usage

if __name__ == "__main__":
    tree = AVLTree()
    root = None
```

```
elements = [10, 20, 30, 40, 50, 25]
```

```
for elem in elements:
    root = tree.insert(root, elem)
```

```
print("Inorder traversal of the AVL tree:")
tree.inorder(root)
```

**Output:** objectivec

CopyEdit

Inorder traversal of the AVL tree: 10

20 25 30 40 50

## RESULT

To Implement an AVL tree (a self-balancing binary search tree) in Python, with node insertion and automatic balancing using rotations to maintain the AVL property is verified

**3. Write a Python program to represent a graph using an adjacency list or matrix, and implement both Breadth-First Search (BFS) and Depth-First Search (DFS) traversal methods.**

### **AIM**

To Write a Python program to represent a graph using an adjacency list or matrix, and implement both Breadth-First Search (BFS) and Depth-First Search (DFS) traversal methods.

### **Algorithms**

#### BFS (Breadth-First Search)

1. Start from a source node
2. Use a queue to explore nodes level by level
3. Mark visited nodes to avoid revisiting
4. Visit all neighbors before going deeper

#### DFS (Depth-First Search)

1. Start from a source node
2. Use recursion (or a stack) to explore as deep as possible
3. Backtrack when no unvisited neighbors remain

### **PROGRAM**

```
python CopyEdit
from collections import deque, defaultdict
class Graph:
    def __init__(self):
        self.adj_list = defaultdict(list)

    # Add edge to the graph (undirected by default)
    def add_edge(self, u, v, directed=False):
        if directed == False:
            self.adj_list[u].append(v)
            self.adj_list[v].append(u)
        else:
            self.adj_list[u].append(v)
```

```
    self.adj_list[u].append(v) if
```

```
        not directed:
```

```
            self.adj_list[v].append(u)
```

```
# Display the graph def
```

```
def display(self):
```

```
    for node in self.adj_list:
```

```
        print(f"\n{node} -> {self.adj_list[node]}")
```

```
# Breadth-First Search def
```

```
def bfs(self, start):
```

```
    visited = set()
```

```
    queue = deque([start])
```

```
    print("\nBFS traversal:")
```

```
    while queue:
```

```
        node = queue.popleft() if
```

```
            node not in visited:
```

```
                print(node, end=' ')
```

```
                visited.add(node)
```

```
                for neighbor in self.adj_list[node]: if
```

```
                    neighbor not in visited:
```

```
                        queue.append(neighbor)
```

```
# Depth-First Search (Recursive) def
```

```
def dfs(self, start):
```

```
    visited = set()
```

```

print("\nDFS traversal:")
self._dfs_recursive(start, visited)

def _dfs_recursive(self, node, visited):
    if node not in visited:
        print(node, end=' ')
        visited.add(node)
        for neighbor in self.adj_list[node]:
            if neighbor not in visited:
                self._dfs_recursive(neighbor, visited)

# Example usage
if __name__ == "__main__":
    g = Graph()
    g.add_edge(0, 1)
    g.add_edge(0, 2)
    g.add_edge(1, 3)
    g.add_edge(1, 4)
    g.add_edge(2, 5)
    g.add_edge(2, 6)

    print("Graph adjacency list:")
    g.display()

g.bfs(0) # Output: 0 1 2 3 4 5 6
g.dfs(0) # Output: 0 1 3 4 2 5 6

```

Output Example:

less

CopyEdit

Graph adjacency list:

0 -> [1, 2]

1 -> [0, 3, 4]

2 -> [0, 5, 6]

3 -> [1]

4 -> [1]

5 -> [2]

6 -> [2]

BFS traversal:

0 1 2 3 4 5 6

DFS traversal:

0 1 3 4 2 5 6

## RESULT

To Write a Python program to represent a graph using an adjacency list or matrix, and implement both Breadth-First Search (BFS) and Depth-First Search (DFS) traversal methods is verified .

#### **4. Implement Prim's algorithm in Python to find the Minimum Spanning Tree of a weighted, undirected graph.**

##### **AIM**

To Implement Prim's algorithm in Python to find the Minimum Spanning Tree of a weighted, undirected graph.

##### **Algorithms**

Prim's Algorithm – Steps

1. Start with an arbitrary node (say node 0)
2. Use a min-heap (priority queue) to always pick the edge with the smallest weight
3. Maintain a set of visited nodes to avoid cycles
4. For each newly visited node:
  - o Add its adjacent edges (that lead to unvisited nodes) into the heap
5. Repeat until all nodes are in the MST

Python Implementation Using Min-Heap

python

CopyEdit

```
import heapq
```

```
from collections import defaultdict
```

```
class Graph:
```

```
    def __init__(self, vertices): self.V =  
        vertices  
        self.graph = defaultdict(list) # {u: [(v, weight), ...]}
```

```
    def add_edge(self, u, v, weight):  
        self.graph[u].append((v, weight))
```

```

self.graph[v].append((u, weight)) # Because the graph is undirected

def prim_mst(self):
    visited = set()
    min_heap = [(0, 0)] # (weight, start_node)
    mst_weight = 0
    mst_edges = []

    while len(visited) < self.V and min_heap:
        weight, u = heapq.heappop(min_heap)
        if u in visited:
            continue

        visited.add(u)
        mst_weight += weight

        # Include edge in result if not starting node if
        # weight != 0:
        mst_edges.append((prev_node[u], u, weight))

    # Add all unvisited neighbors to heap for
    # v, w in self.graph[u]:
    if v not in visited:
        heapq.heappush(min_heap, (w, v))
        prev_node[v] = u # Store the previous node for edge tracing

return mst_weight, mst_edges

```

```
#  
  
# Example usage  
  
if __name__ == "__main__": g =  
  
    Graph(5)  
    g.add_edge(0, 1, 2)  
  
    g.add_edge(0, 3, 6)  
  
    g.add_edge(1, 2, 3)  
  
    g.add_edge(1, 3, 8)  
  
    g.add_edge(1, 4, 5)  
  
    g.add_edge(2, 4, 7)  
  
    g.add_edge(3, 4, 9)  
  
  
    # Store previous node globally for edge tracking  
    prev_node = {}  
  
  
    total_weight, mst = g.prim_mst()  
  
    print("Minimum Spanning Tree edges and weights:")  
    for  
        u, v, w in mst:  
            print(f"\t{u} - {v} (weight: {w})")  
  
  
    print("Total weight of MST:", total_weight)
```

Output Example:

less CopyEdit

Minimum Spanning Tree edges and weights:

0 - 1 (weight: 2)

1 - 2 (weight: 3)

1 - 4 (weight: 5)

0 - 3 (weight: 6)

Total weight of MST: 16

## RESULT

To Implement Prim's algorithm in Python to find the Minimum Spanning Tree of a weighted, undirected graph is verified

## **9.Implement Dijkstra's algorithm to find the shortest path from a starting node to all other nodes in a graph with non-negative edge weights.**

### **AIM**

To Implement Dijkstra's algorithm to find the shortest path from a starting node to all other nodes in a graph with non-negative edge weights.

### **Algorithms**

1. Initialize distances:
  - o Set distance to source node = 0
  - o All other nodes =  $\infty$  (infinity)
2. Use a min-heap (priority queue) to select the node with the smallest current distance
3. For each selected node:
  - o Visit all unvisited neighbors
  - o Update their distances if a shorter path is found
4. Repeat until all nodes are visited (or the heap is empty)

### **PROGRAM**

```
python CopyEdit
```

```
import heapq
```

```
from collections import defaultdict
```

```
class Graph:
```

```
    def __init__(self):
```

```
        self.graph = defaultdict(list) # {node: [(neighbor, weight), ...]}
```

```
    def add_edge(self, u, v, weight):
```

```
        self.graph[u].append((v, weight))
```

```

# For undirected graph, add the reverse edge too: #
self.graph[v].append((u, weight))

def dijkstra(self, start):
    # Step 1: Initialize distances
    distances = {node: float('inf') for node in self.graph}
    distances[start] = 0

    # Step 2: Min-heap: (distance, node) heap
    = [(0, start)]

    while heap:
        current_dist, current_node = heapq.heappop(heap)

        # Step 3: Visit each neighbor
        for neighbor, weight in self.graph[current_node]:
            distance = current_dist + weight

            # Step 4: Update distance if shorter path is found if
            distance < distances[neighbor]:
                distances[neighbor] = distance
                heapq.heappush(heap, (distance, neighbor))

    return distances

```

□ Example Usage

python

CopyEdit

```
if __name__ == "__main__":
    g = Graph()
    g.add_edge('A', 'B', 4)
    g.add_edge('A', 'C', 2)
    g.add_edge('B', 'C', 1)
    g.add_edge('B', 'D', 5)
    g.add_edge('C', 'D', 8)
    g.add_edge('C', 'E', 10)
    g.add_edge('D', 'E', 2)
    g.add_edge('D', 'Z', 6)
    g.add_edge('E', 'Z', 3)
```

```
start_node = 'A'
```

```
distances = g.dijkstra(start_node)
```

```
print(f"Shortest distances from node '{start_node}':")
for node in distances:
```

```
    print(f"{node}: {distances[node]}")
```

**Output** vbnet

CopyEdit

Shortest distances from node 'A': A:

0

B: 3

C: 2

D: 8

E: 10

Z: 13

## **RESULT**

To Implement Dijkstra's algorithm to find the shortest path from a starting node to all other nodes in a graph with non-negative edge weights is verified.

## **10. Write a Python program to implement Bubble sort, Selection sort, Insertion sort Algorithms.**

### **AIM**

To Write a Python program to implement Bubble sort, Selection sort, Insertion sort Algorithms

#### **1. Bubble Sort**

##### **Algorithm**

1. Repeatedly step through the list
2. Compare adjacent items
3. Swap them if they are in the wrong order
4. Continue until the list is sorted

Time Complexity:

- Worst-case:  $O(n^2)$

### **PROGRAM**

python

CopyEdit

```
def bubble_sort(arr): n  
    = len(arr)  
    for i in range(n):  
        # Last i elements are already in place for j  
        in range(0, n - i - 1):  
            if arr[j] > arr[j + 1]: #  
                Swap  
                arr[j], arr[j + 1] = arr[j + 1], arr[j]  
            return arr
```

- o 2. Selection Sort

- Algorithm:

- 1. For each index i from 0 to n-1:

- o Find the minimum element from i to end
    - o Swap it with element at i

- Time Complexity:

- Worst-case:  $O(n^2)$

python

CopyEdit

```
def selection_sort(arr): n  
    = len(arr)  
    for i in range(n):  
        min_index = i  
        # Find the smallest element in the remaining unsorted list for j  
        in range(i + 1, n):  
            if arr[j] < arr[min_index]:  
                min_index = j  
        # Swap the found minimum with the current element arr[i],  
        arr[min_index] = arr[min_index], arr[i]  
    return arr
```

- o 3. Insertion Sort

- Algorithm:

- 1. Start from the second element
- 2. Compare with all previous elements and insert it into the correct position
- 3. Shift elements to make space

□ Time Complexity:

- Worst-case:  $O(n^2)$
- Best-case (already sorted):  $O(n)$

python

CopyEdit

```
def insertion_sort(arr):
```

```
    for i in range(1, len(arr)):
```

```
        key = arr[i]
```

```
        j = i - 1
```

```
        # Shift elements greater than key to the right while j
```

```
>= 0 and arr[j] > key:
```

```
            arr[j + 1] = arr[j]
```

```
            j -= 1
```

```
    arr[j + 1] = key
```

```
return arr
```

□ Example Usage

python

CopyEdit

```
if __name__ == "__main__": data =
```

```
[64, 25, 12, 22, 11]
```

```
print("Original array:", data)
```

```
print("\nBubble Sort:")
```

```
print(bubble_sort(data.copy()))
```

```
print("\nSelection Sort:")
```

```
print(selection_sort(data.copy()))
```

```
print("\nInsertion Sort:")
```

```
print(insertion_sort(data.copy()))
```

- o Sample Output

less

CopyEdit

Original array: [64, 25, 12, 22, 11]

Bubble Sort:

[11, 12, 22, 25, 64]

Selection Sort: [11,

12, 22, 25, 64]

Insertion Sort:

[11, 12, 22, 25, 64]

**11. Write a Python program to implement Linear Search and Binary Search, with input validation (e.g., ensuring sorted input for binary search) and returning the index of the searched element if found.**

### **AIM**

To Write a Python program to implement Linear Search and Binary Search, with input validation (e.g., ensuring sorted input for binary search) and returning the index of the searched element if found.

### **Algorithm**

#### **Linear Search**

5. Start from the first element.
6. Compare each element with the target.
7. If found, return the index.
8. If loop ends without finding, return -1.

#### **o 2. Binary Search**

Prerequisites:

- Input list must be sorted

Algorithm:

1. Set low = 0, high = n - 1
2. While low <= high:
  - o Compute mid = (low + high) // 2
  - o If arr[mid] == target: return mid
  - o If arr[mid] < target: search right half (low = mid + 1)
  - o Else: search left half (high = mid - 1)
3. If not found, return -1

## PROGRAM

```
python CopyEdit

def linear_search(arr, target): """Linear
    Search algorithm"""

for index, value in enumerate(arr): if
    value == target:
        return index

return -1

def binary_search(arr, target):
    """Binary Search algorithm with sorted input validation"""
    if arr != sorted(arr):
        raise ValueError("Input array must be sorted for Binary Search.")

    low = 0
    high = len(arr) - 1

    while low <= high:
        mid = (low + high) // 2 if
            arr[mid] == target:
                return mid

        elif arr[mid] < target:
            low = mid + 1

        else:
            high = mid - 1

    return -1
```

□ Example Usage

```
python
```

```
CopyEdit
```

```
if __name__ == "__main__":
    # Unsorted input for linear search unsorted_data
    = [23, 45, 12, 9, 34, 56]
    target = 34

    print("Linear Search:")
    index = linear_search(unsorted_data, target)
    if index != -1:
        print(f"Element {target} found at index {index}")
    else:
        print(f"Element {target} not found")

# Sorted input for binary search sorted_data =
sorted(unsorted_data)
print("\nBinary Search
(on sorted array):")
try:
    index = binary_search(sorted_data, target)
    if index != -1:
        print(f"Element {target} found at index {index}")
    else:
        print(f"Element {target} not found")
except ValueError as ve:
    print("Error:", ve)
```

o Sample Output sql

CopyEdit Linear

Search:

Element 34 found at index 4

Binary Search (on sorted array): Element

34 found at index 3

## **RESULT**

To Write a Python program to implement Linear Search and Binary Search, with input validation (e.g., ensuring sorted input for binary search) and returning the index of the searched element if found is verified .

## **12. Write a program to implement Linear Probing and Quadratic Probing for inserting and searching elements.**

### **AIM**

To Write a program to implement Linear Probing and Quadratic Probing for inserting and searching elements.

#### **Linear Probing**

- Check next index:  $\text{index} = (\text{hash} + i) \% \text{table\_size}$
- Quadratic Probing
  - Use a quadratic jump:  $\text{index} = (\text{hash} + i^2) \% \text{table\_size}$

#### **Algorithm**

- Insertion
  1. Compute the hash index:  $\text{index} = \text{key \% table\_size}$
  2. If the index is empty, insert the key
  3. If there's a collision, probe using:
    - Linear:  $(\text{index} + i) \% \text{table\_size}$
    - Quadratic:  $(\text{index} + i*i) \% \text{table\_size}$
  4. Repeat until an empty slot is found or the table is full

- Search

1. Use the same probing method as insertion
2. If key is found, return index
3. If an empty slot is found before the key, it's not present

#### **PROGRAM**

python CopyEdit

class HashTable:

```
def __init__(self, size):
```

```
self.size = size  
self.table_linear = [None] * size  
self.table_quadratic = [None] * size
```

```
# Linear Probing Insert  
  
def insert_linear(self, key): for  
    i in range(self.size):  
        index = (key + i) % self.size  
        if self.table_linear[index] is None:  
            self.table_linear[index] = key return  
            index  
        raise Exception("Hash table (linear) is full")
```

```
# Quadratic Probing Insert  
  
def insert_quadratic(self, key): for i  
    in range(self.size):  
        index = (key + i*i) % self.size  
        if self.table_quadratic[index] is None:  
            self.table_quadratic[index] = key return  
            index  
        raise Exception("Hash table (quadratic) is full")
```

```
# Linear Probing Search def  
  
search_linear(self, key):  
    for i in range(self.size):  
        index = (key + i) % self.size  
        if self.table_linear[index] == key:
```

```

        return index

    if self.table_linear[index] is None:
        return -1

    return -1

# Quadratic Probing Search

def search_quadratic(self, key):
    for i in range(self.size):
        index = (key + i*i) % self.size
        if self.table_quadratic[index] == key:
            return index
    if self.table_quadratic[index] is None:
        return -1
    return -1

```

```

def display_tables(self):
    print("Linear
          Probing Table:")
    print(self.table_linear)
    print("Quadratic Probing Table:")
    print(self.table_quadratic)

```

□ Example Usage

```

python
CopyEdit

if __name__ == "__main__":
    ht =
        HashTable(10)
    keys = [27, 18, 29, 28, 39]

```

```

print("Inserting keys using Linear Probing:") for
key in keys:
    ht.insert_linear(key)

print("\nInserting keys using Quadratic Probing:") for
key in keys:
    ht.insert_quadratic(key)

ht.display_tables()

# Search example
key_to_search = 28

print(f"\nSearching for {key_to_search}...") lin_index
= ht.search_linear(key_to_search) quad_index =
ht.search_quadratic(key_to_search) print(f"Linear
Probing Index: {lin_index}") print(f"Quadratic
Probing Index: {quad_index}")

```

### **Sample Output:**

mathematica CopyEdit

Inserting keys using Linear Probing:

Inserting keys using Quadratic Probing:

Linear Probing Table:

[None, None, None, None, 27, 18, 29, 28, 39, None]

Quadratic Probing Table:

[None, None, None, None, 27, 18, 29, 39, 28, None]

Searching for 28... Linear

Probing Index: 7

Quadratic Probing Index: 8

## RESULT

To Write a Python program to implement Linear Search and Binary Search, with input validation (e.g., ensuring sorted input for binary search) and returning the index of the searched element if found.