**EX:6** **Data Fitting by Regression**

**AIM:**

To perform data fitting using simple linear regression and polynomial regression in Python to model and analyze relationships between variables**.**

**ALGORITHM:**

1) Import necessary libraries for numerical operations, plotting, and regression modeling.

2) Define input features (X) and target values (y) as NumPy arrays.

3) Visualize the original data using a scatter plot to understand the data distribution.

4) Train a linear regression model on the input data and plot its predictions.

5) Transform the features for polynomial regression, train the model, and plot its predictions.

6) Make predictions with both models for a new input and print the results.

**IMPLEMENTATION:**

import numpy as np

import matplotlib.pyplot as plt

from sklearn.linear\_model import LinearRegression

from sklearn.preprocessing import PolynomialFeatures

# Prepare the data

X = np.array([1, 2, 3, 4, 5, 6, 7, 8, 9]).reshape(-1, 1)

y = np.array([2.5, 3.7, 4.1, 6.0, 7.2, 9.0, 10.5, 13.0, 15.6])

# Plot the original data

plt.scatter(X, y, color='blue', label='Original Data')

# Simple Linear Regression

linear\_model = LinearRegression()

linear\_model.fit(X, y)

y\_linear\_pred = linear\_model.predict(X)

plt.plot(X, y\_linear\_pred, color='green', label='Linear Fit')

# Polynomial Regression (degree = 2)

poly = PolynomialFeatures(degree=2)

X\_poly = poly.fit\_transform(X)

poly\_model = LinearRegression()

poly\_model.fit(X\_poly, y)

y\_poly\_pred = poly\_model.predict(X\_poly)

plt.plot(X, y\_poly\_pred, color='red', label='Polynomial Fit')

# Show plot

plt.xlabel('X')

plt.ylabel('y')

plt.title('Data Fitting using Regression')

plt.legend()

plt.grid(True)

plt.show()

# Prediction example

x\_new = np.array([[10]])

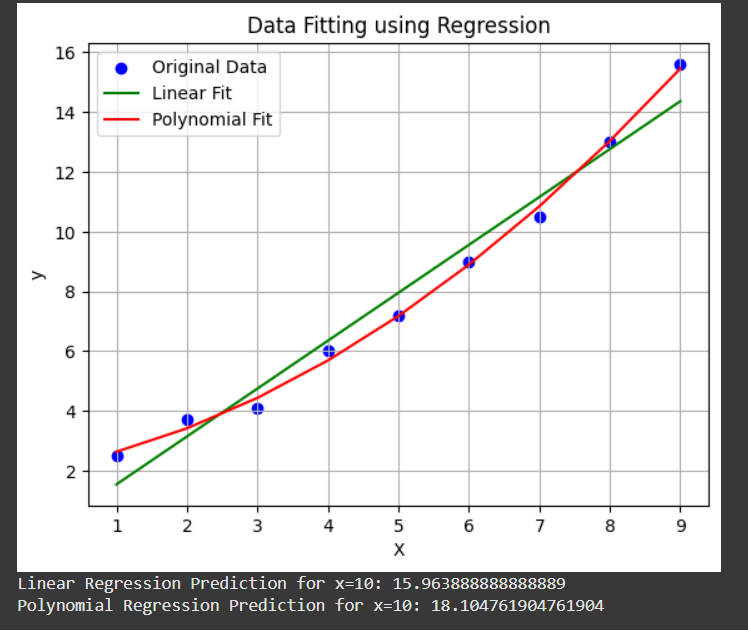
y\_linear\_new = linear\_model.predict(x\_new)

y\_poly\_new = poly\_model.predict(poly.transform(x\_new))

print("Linear Regression Prediction for x=10:", y\_linear\_new[0])

print("Polynomial Regression Prediction for x=10:", y\_poly\_new[0])

**OUTPUT:**



Linear Regression Prediction for x=10: 15.963888888888889

Polynomial Regression Prediction for x=10: 18.104761904761904

**RESULT:**

EX:7 IMPLEMENTATION OF CRISP MODEL

**AIM:**

To implement the CRISP-DM process using Python for a simple data analysis and modeling task.

**ALGORITHM:**

1) Define the problem of classifying iris flowers into species based on their measurements.

2) Load and explore the dataset to understand the features and target classes.

3) Split the dataset into input features (X) and target labels (y), and prepare training and testing data.

4) Train a Decision Tree Classifier using the training data.

5) Predict and evaluate the model on the test data using accuracy as the performance metric.

6) Use the trained model to predict the class of a new, unseen flower sample.

**IMPLEMENTATION:**

# Import necessary libraries

from sklearn.datasets import load\_iris

import pandas as pd

from sklearn.model\_selection import train\_test\_split

from sklearn.tree import DecisionTreeClassifier

from sklearn.metrics import accuracy\_score

# Business Understanding

# Classify iris flowers based on petal/sepal size

# Data Understanding

iris = load\_iris()

df = pd.DataFrame(iris.data, columns=iris.feature\_names)

df['target'] = iris.target

print("First 5 rows of data:\n", df.head())

# Data Preparation

X = df.drop('target', axis=1)

y = df['target']

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=42)

# Modeling

model = DecisionTreeClassifier()

model.fit(X\_train, y\_train)

# Evaluation

y\_pred = model.predict(X\_test)

accuracy = accuracy\_score(y\_test, y\_pred)

print(f"Model Accuracy: {accuracy:.2f}")

# Deployment (predict for a new flower)

sample = [[5.1, 3.5, 1.4, 0.2]]

prediction = model.predict(sample)

print("Predicted Class:", iris.target\_names[prediction][0])

**OUTPUT:**

**First 5 rows of data:**

**sepal length (cm) sepal width (cm) petal length (cm) petal width (cm)**

**0 5.1 3.5 1.4 0.2**

**1 4.9 3.0 1.4 0.2**

**2 4.7 3.2 1.3 0.2**

**3 4.6 3.1 1.5 0.2**

**4 5.0 3.6 1.4 0.2**

**target**

**0 0**

**1 0**

**2 0**

**3 0**

**4 0**

**Model Accuracy: 1.00**

**Predicted Class: setosa**

**RESULT:**

EX:8 IMPLEMENTATION OF LOGIC GATES

**AIM:**

**ALGORITHM:**

**IMPLEMENTATION:**

# Define logic gate functions

def AND(a, b):

return a & b

def OR(a, b):

return a | b

def NOT(a):

return ~a & 1

# To ensure output is 0 or 1 only

def NAND(a, b):

return ~(a & b) & 1

def NOR(a, b):

return ~(a | b) & 1

def XOR(a, b):

return a ^ b

def XNOR(a, b):

return ~(a ^ b) & 1

# Test inputs

inputs = [(0, 0), (0, 1), (1, 0), (1, 1)]

# Display output

print("A B | AND OR NAND NOR XOR XNOR")

print("-------------------------------")

for a, b in inputs:

print(f"{a} {b} | {AND(a, b)} {OR(a, b)} {NAND(a, b)} {NOR(a, b)} {XOR(a, b)} {XNOR(a, b)}")

**OUTPUT:**

A B | AND OR NAND NOR XOR XNOR

----------------------------------------------------------

0 0 | 0 0 1 1 0 1

0 1 | 0 1 1 0 1 0

1 0 | 0 1 1 0 1 0

1 1 | 1 1 0 0 0 1

**​**

**RESULT:**

EX:9 Implementation of Genetic Algorithm

**AIM:**

**ALGORITHM:**

**IMPLEMENTATION:**

import random

# Define the problem

def fitness\_function(x):

return x \*\* 2 # The function we want to maximize

# Create an initial population

def create\_population(size, lower\_bound, upper\_bound):

return [random.randint(lower\_bound, upper\_bound) for \_ in range(size)]

# Select parents based on fitness

def select\_parents(population):

return random.choices(population, weights=[fitness\_function(x) for x in population], k=2)

# Perform crossover

def crossover(parent1, parent2):

return (parent1 + parent2) // 2 # Simple average crossover

# Perform mutation

def mutate(individual, lower\_bound, upper\_bound, mutation\_rate=0.1):

if random.random() < mutation\_rate:

return random.randint(lower\_bound, upper\_bound)

return individual

# Genetic Algorithm

def genetic\_algorithm(generations, population\_size, lower\_bound, upper\_bound):

population = create\_population(population\_size, lower\_bound, upper\_bound)

for generation in range(generations):

new\_population = []

for \_ in range(population\_size):

parent1, parent2 = select\_parents(population)

child = crossover(parent1, parent2)

child = mutate(child, lower\_bound, upper\_bound)

new\_population.append(child)

population = new\_population

best\_individual = max(population, key=fitness\_function)

print(f"Generation {generation + 1}: Best = {best\_individual}, Fitness = {fitness\_function(best\_individual)}")

return max(population, key=fitness\_function)

# Run the Genetic Algorithm

best\_solution = genetic\_algorithm(generations=10, population\_size=10, lower\_bound=0, upper\_bound=100)

print(f"Best solution found: {best\_solution}, Fitness: {fitness\_function(best\_solution)}")

**OUTPUT:**

Generation 1: Best = 76, Fitness = 5776

Generation 2: Best = 93, Fitness = 8649

Generation 3: Best = 84, Fitness = 7056

Generation 4: Best = 95, Fitness = 9025

Generation 5: Best = 95, Fitness = 9025

Generation 6: Best = 82, Fitness = 6724

Generation 7: Best = 82, Fitness = 6724

Generation 8: Best = 78, Fitness = 6084

Generation 9: Best = 77, Fitness = 5929

Generation 10: Best = 77, Fitness = 5929

Best solution found: 77, Fitness: 5929

**RESULT:**

EX:10 IMPLEMENTATION OF CLASSIFICATION ALGORITHM

**AIM:**

**ALGORITHM**:

**IMPLEMENTATION:**

import numpy as np

# Prepare the data

X = np.array([[0, 0], [0, 1], [1, 0], [1, 1]]) # Inputs

y = np.array([0, 0, 0, 1]) # AND gate labels

# Define the sigmoid function

def sigmoid(z):

return 1 / (1 + np.exp(-z))

# Initialize parameters

weights = np.zeros(X.shape[1])

bias = 0

learning\_rate = 0.1

epochs = 1000

# Train using gradient descent

for i in range(epochs):

model = np.dot(X, weights) + bias

prediction = sigmoid(model)

error = prediction - y

dw = np.dot(X.T, error) / len(X)

db = np.sum(error) / len(X)

weights -= learning\_rate \* dw

bias -= learning\_rate \* db

# Predict function

def predict(X):

return sigmoid(np.dot(X, weights) + bias) >= 0.5

# Evaluate

predictions = predict(X).astype(int)

print("Predictions:", predictions)

print("Actual :", y)

accuracy = np.mean(predictions == y)

print("Accuracy :", accuracy)

**OUTPUT:**

Predictions: [0 0 0 1]

Actual : [0 0 0 1]

Accuracy : 1.0

**RESULT:**