**1. Implement graph traversal BFS and DFS using adjacency matrix and list both.**

#include <iostream>

#include <queue>

#include <vector>

using namespace std;

const int MAX\_SIZE = 100;

class GraphMatrix {

private:

int V; // Number of vertices

int adj[MAX\_SIZE][MAX\_SIZE]; // Adjacency matrix

public:

GraphMatrix(int vertices) {

V = vertices;

// Initialize all elements of adjacency matrix to 0

for (int i = 0; i < V; ++i) {

for (int j = 0; j < V; ++j) {

adj[i][j] = 0;

}

}

}

void addEdge(int u, int v) {

adj[u][v] = 1;

adj[v][u] = 1; // Uncomment this for undirected graph

}

void bfs(int start) {

vector<bool> visited(V, false);

queue<int> q;

visited[start] = true;

q.push(start);

while (!q.empty()) {

int curr = q.front();

q.pop();

cout << curr << " ";

for (int i = 0; i < V; ++i) {

if (adj[curr][i] == 1 && !visited[i]) {

visited[i] = true;

q.push(i);

}

}

}

}

void dfsUtil(int v, vector<bool>& visited) {

visited[v] = true;

cout << v << " ";

for (int i = 0; i < V; ++i) {

if (adj[v][i] == 1 && !visited[i]) {

dfsUtil(i, visited);

}

}

}

void dfs(int start) {

vector<bool> visited(V, false);

dfsUtil(start, visited);

}

};

int main() {

GraphMatrix graphMatrix(6);

graphMatrix.addEdge(0, 1);

graphMatrix.addEdge(0, 2);

graphMatrix.addEdge(1, 3);

graphMatrix.addEdge(2, 4);

graphMatrix.addEdge(3, 5);

cout << "BFS traversal starting from node 0: ";

graphMatrix.bfs(0);

cout << endl;

cout << "DFS traversal starting from node 0: ";

graphMatrix.dfs(0);

cout << endl;

// Example usage of GraphList

GraphList graphList(6);

graphList.addEdge(0, 1);

graphList.addEdge(0, 2);

graphList.addEdge(1, 3);

graphList.addEdge(2, 4);

graphList.addEdge(3, 5);

cout << "BFS traversal starting from node 0: ";

graphList.bfs(0);

cout << endl;

cout << "DFS traversal starting from node 0: ";

graphList.dfs(0);

cout << endl;

return 0;

}

**2. Find Eigen value and Eigen vectors using Gaussian elimination.**

import numpy as np

def find\_eigenvalues\_eigenvectors(A):

"""

Finds eigenvalues and eigenvectors of a square matrix using Gaussian elimination.

Args:

A: A numpy array representing the square matrix.

Returns:

A tuple containing:

eigenvalues: A numpy array containing the eigenvalues.

eigenvectors: A numpy array containing the eigenvectors (one column per eigenvector).

"""

n = A.shape[0]

# Find eigenvalues using the characteristic equation

eigenvalues = []

for i in range(n):

# Create a matrix with lambda on the diagonal and subtract A

lambda\_matrix = np.diag(np.full(n, np.nan))

lambda\_matrix[i, i] = 0

lambda\_matrix -= A

# Convert to upper triangular form using Gaussian elimination

for j in range(n - 1):

# Find pivot element

pivot = lambda\_matrix[j, j]

if abs(pivot) < 1e-10:

raise ValueError("Matrix is singular or nearly singular")

for k in range(j + 1, n):

factor = lambda\_matrix[k, j] / pivot

lambda\_matrix[k, j:n] -= factor \* lambda\_matrix[j, j:n]

# Get the characteristic polynomial coefficient from the diagonal element

eigenvalues.append(lambda\_matrix[n-1, n-1])

# Find eigenvectors for each eigenvalue

eigenvectors = np.zeros((n, n))

for i, eigenvalue in enumerate(eigenvalues):

# Create a matrix with lambda on the diagonal and subtract A

lambda\_matrix = np.diag(np.full(n, eigenvalue)) - A

# Convert to reduced row echelon form

for j in range(n - 1):

# Find pivot element

pivot = lambda\_matrix[j, j]

if abs(pivot) < 1e-10:

raise ValueError("Matrix is singular or nearly singular")

for k in range(j + 1, n):

factor = lambda\_matrix[k, j] / pivot

lambda\_matrix[k, :] -= factor \* lambda\_matrix[j, :]

# Solve the linear system for the eigenvector

eigenvector = np.zeros(n)

for j in range(n - 1, -1, -1):

if abs(lambda\_matrix[j, j]) < 1e-10:

eigenvector[j] = 1 # Arbitrary value for nonzero coefficient

else:

eigenvector[j] = (lambda\_matrix[j, n] - np.dot(lambda\_matrix[j, j:n], eigenvector[j:n])) / lambda\_matrix[j, j]

eigenvectors[:, i] = eigenvector

return eigenvalues, eigenvectors

# Example usage

A = np.array([[2, 1], [1, 2]])

eigenvalues, eigenvectors = find\_eigenvalues\_eigenvectors(A)

print("Eigenvalues:", eigenvalues)

print("Eigenvectors:", eigenvectors)

**3. Standardize and normalization of numeric columns of given dataset.**

import pandas as pd

df = pd.read\_csv('adult.csv')

numeric\_columns = df.select\_dtypes(include=['int', 'float']).columns

# Step 3: Implement Standardization

for col in numeric\_columns:

mean = df[col].mean()

std = df[col].std()

df[col + '\_standardized'] = (df[col] - mean) / std

# Step 4: Implement Normalization

for col in numeric\_columns:

min\_val = df[col].min()

max\_val = df[col].max()

df[col + '\_normalized'] = (df[col] - min\_val) / (max\_val - min\_val)

print(df.head())

**4. Final covariance between numeric features of given data set.**

import pandas as pd

def calculate\_covariance(df):

# Select only numeric columns

numeric\_columns = df.select\_dtypes(include=['int', 'float']).columns

# Calculate mean of each numeric column

means = df[numeric\_columns].mean()

# Number of data points

n = len(df)

# Initialize covariance matrix

covariance\_matrix = pd.DataFrame(index=numeric\_columns, columns=numeric\_columns)

for i in range(len(numeric\_columns)):

for j in range(len(numeric\_columns)):

col1 = numeric\_columns[i]

col2 = numeric\_columns[j]

# Calculate covariance between col1 and col2

covariance = 0

for k in range(n):

covariance += (df.at[k, col1] - means[col1]) \* (df.at[k, col2] - means[col2])

covariance /= (n - 1)

# Store covariance in the covariance matrix

covariance\_matrix.at[col1, col2] = covariance

return covariance\_matrix

# Load the dataset (example: 'adult.csv')

df = pd.read\_csv('adult.csv')

# Calculate covariance matrix for numeric features

covariance\_matrix = calculate\_covariance(df)

print("Covariance Matrix:")

print(covariance\_matrix)

**5. Implement PCA to get important features in a dataset.**

import numpy as np

# Sample dataset

data = np.array([

[5, 2, 3],

[11, 21, 6],

[4, 8, 1]

])

# Step 1: Standardize the dataset

mean = np.mean(data, axis=0)

std\_dev = np.std(data, axis=0)

standardized\_data = (data - mean) / std\_dev

# Step 2: Compute the covariance matrix

covariance\_matrix = np.cov(standardized\_data.T)

# Step 3: Compute eigenvalues and eigenvectors

eigenvalues, eigenvectors = np.linalg.eig(covariance\_matrix)

# Step 4: Sort eigenvalues and select top k eigenvectors

k = 2 # Number of principal components

sorted\_indices = np.argsort(eigenvalues)[::-1][:k]

top\_eigenvectors = eigenvectors[:, sorted\_indices]

# Step 5: Project original dataset onto top eigenvectors

principal\_components = np.dot(standardized\_data, top\_eigenvectors)

print("Principal Components:")

print(principal\_components)

**6. Implement HMM model to find probability of observed sequence.**

# Define HMM parameters

states = ['Sunny', 'Rainy']

observations = ['Dry', 'Damp', 'Soggy']

initial\_prob = {'Sunny': 0.5, 'Rainy': 0.5}

transition\_prob = {

'Sunny': {'Sunny': 0.8, 'Rainy': 0.2},

'Rainy': {'Sunny': 0.4, 'Rainy': 0.6}

}

emission\_prob = {

'Sunny': {'Dry': 0.6, 'Damp': 0.3, 'Soggy': 0.1},

'Rainy': {'Dry': 0.1, 'Damp': 0.4, 'Soggy': 0.5}

}

def forward\_algorithm(observations, states, initial\_prob, transition\_prob, emission\_prob):

T = len(observations)

alpha = [{}]

# Initialize base case (t=0)

for state in states:

alpha[0][state] = initial\_prob[state] \* emission\_prob[state][observations[0]]

# Fill in the rest of the probabilities

for t in range(1, T):

alpha.append({})

for curr\_state in states:

alpha[t][curr\_state] = sum(alpha[t-1][prev\_state] \* transition\_prob[prev\_state][curr\_state] \* emission\_prob[curr\_state][observations[t]] for prev\_state in states)

# Calculate the probability of the sequence

prob = sum(alpha[T-1][state] for state in states)

return prob

# Example usage:

observed\_sequence = ['Dry', 'Damp', 'Soggy']

prob\_observed\_sequence = forward\_algorithm(observed\_sequence, states, initial\_prob, transition\_prob, emission\_prob)

print("Probability of observed sequence:", prob\_observed\_sequence)

**7. Implement Viterbi algorithm to find best path of observed events.**

# Define HMM parameters (same as before)

states = ['Sunny', 'Rainy']

observations = ['Dry', 'Damp', 'Soggy']

initial\_prob = {'Sunny': 0.5, 'Rainy': 0.5}

transition\_prob = {

'Sunny': {'Sunny': 0.8, 'Rainy': 0.2},

'Rainy': {'Sunny': 0.4, 'Rainy': 0.6}

}

emission\_prob = {

'Sunny': {'Dry': 0.6, 'Damp': 0.3, 'Soggy': 0.1},

'Rainy': {'Dry': 0.1, 'Damp': 0.4, 'Soggy': 0.5}

}

def viterbi\_algorithm(observations, states, initial\_prob, transition\_prob, emission\_prob):

T = len(observations)

V = [{}]

path = {}

# Initialize base case (t=0)

for state in states:

V[0][state] = initial\_prob[state] \* emission\_prob[state][observations[0]]

path[state] = [state]

# Fill in the rest of the probabilities and paths

for t in range(1, T):

V.append({})

new\_path = {}

for curr\_state in states:

(max\_prob, prev\_state) = max((V[t-1][prev\_state] \* transition\_prob[prev\_state][curr\_state] \* emission\_prob[curr\_state][observations[t]], prev\_state) for prev\_state in states)

V[t][curr\_state] = max\_prob

new\_path[curr\_state] = path[prev\_state] + [curr\_state]

# Update path with the new most likely path

path = new\_path

# Find the most likely final state

(max\_prob, best\_path\_end\_state) = max((V[T-1][state], state) for state in states)

# Return the best path (sequence of states) and its probability

return path[best\_path\_end\_state], max\_prob

# Example usage:

observed\_sequence = ['Dry', 'Damp', 'Soggy']

best\_path, probability = viterbi\_algorithm(observed\_sequence, states, initial\_prob, transition\_prob, emission\_prob)

print("Best Path (Sequence of States):", best\_path)

print("Probability of Best Path:", probability)

**8. Implement SVD to find matrix decomposition set.**

import numpy as np

def svd(A):

# Step 1: Compute A^T A and A A^T

ATA = A.T.dot(A)

AAT = A.dot(A.T)

# Step 2: Compute eigenvalues and eigenvectors of A^T A and A A^T

eigenvalues\_ATA, U = np.linalg.eigh(ATA)

eigenvalues\_AAT, V = np.linalg.eigh(AAT)

# Step 3: Sort eigenvalues in descending order

idx\_ATA = np.argsort(eigenvalues\_ATA)[::-1]

idx\_AAT = np.argsort(eigenvalues\_AAT)[::-1]

eigenvalues\_ATA = eigenvalues\_ATA[idx\_ATA]

U = U[:, idx\_ATA]

eigenvalues\_AAT = eigenvalues\_AAT[idx\_AAT]

V = V[:, idx\_AAT]

# Step 4: Compute singular values and form Sigma matrix

sigma = np.sqrt(eigenvalues\_ATA)

Sigma = np.diag(sigma)

# Step 5: Compute V^T

V = V.T

return U, Sigma, V

# Example matrix

A = np.array([[1, 2],

[3, 4],

[5, 6]])

# Perform SVD

U, Sigma, Vt = svd(A)

# Reconstruct original matrix A

A\_reconstructed = U.dot(Sigma).dot(Vt)

print("U:")

print(U)

print("Sigma:")

print(Sigma)

print("V^T:")

print(Vt)

print("Reconstructed A:")

print(A\_reconstructed)

**9. Find correlation between two matrices.**

import numpy as np

def matrix\_correlation(A, B):

# Calculate the mean of each matrix

mean\_A = np.mean(A)

mean\_B = np.mean(B)

# Calculate the covariance between corresponding elements

covariance = np.mean((A - mean\_A) \* (B - mean\_B))

# Calculate the standard deviations

std\_A = np.std(A)

std\_B = np.std(B)

# Calculate the correlation coefficient

correlation\_coefficient = covariance / (std\_A \* std\_B)

return correlation\_coefficient

# Example matrices

A = np.array([[1, 2, 3],

[4, 5, 6],

[7, 8, 9]])

B = np.array([[2, 4, 6],

[8, 10, 12],

[14, 16, 18]])

correlation = matrix\_correlation(A, B)

print("Correlation between matrices A and B:", correlation)

**10. Implement LDA for classification.**

import numpy as np

def lda(X, y):

"""

Performs Linear Discriminant Analysis (LDA) for classification.

Args:

X: A numpy array of shape (n\_samples, n\_features) containing the data.

y: A numpy array of shape (n\_samples,) containing the class labels.

Returns:

A tuple containing:

w: The weight vector used for projection.

mean\_vectors: A numpy array of shape (n\_classes, n\_features) containing the mean vector for each class.

"""

# Get class labels and number of samples per class

classes, counts = np.unique(y, return\_counts=True)

n\_classes = len(classes)

# Calculate the mean vector for each class

mean\_vectors = np.zeros((n\_classes, X.shape[1]))

for i, class\_label in enumerate(classes):

class\_indices = np.where(y == class\_label)[0]

mean\_vectors[i] = np.mean(X[class\_indices], axis=0)

# Calculate the within-class scatter matrix (Sw)

Sw = np.zeros((X.shape[1], X.shape[1]))

for i, class\_label in enumerate(classes):

class\_indices = np.where(y == class\_label)[0]

class\_data = X[class\_indices]

centered\_data = class\_data - mean\_vectors[i]

Sw += np.cov(centered\_data.T) \* counts[i]

# Calculate the between-class scatter matrix (Sb)

Sb = np.zeros((X.shape[1], X.shape[1]))

total\_mean = np.mean(X, axis=0)

for i in range(n\_classes):

centered\_mean = mean\_vectors[i] - total\_mean

Sb += counts[i] \* np.outer(centered\_mean, centered\_mean)

# Solve the generalized eigenvalue problem (Sw^-1 \* Sb \* w = lambda \* w)

eigenvalues, eigenvectors = np.linalg.eig(np.linalg.inv(Sw) @ Sb)

# Sort eigenvectors by decreasing eigenvalues

sorted\_index = np.argsort(eigenvalues)[::-1]

eigenvalues = eigenvalues[sorted\_index]

eigenvectors = eigenvectors[:, sorted\_index]

# Select the top d eigenvectors corresponding to the largest eigenvalues

# (d is typically chosen to be the number of classes minus 1)

d = n\_classes - 1

w = eigenvectors[:, :d]

return w, mean\_vectors

def predict(X\_new, w, mean\_vectors):

"""

Projects new data points onto the LDA subspace and predicts class labels.

Args:

X\_new: A numpy array of shape (n\_samples, n\_features) containing the new data points.

w: The weight vector obtained from the LDA training.

mean\_vectors: The mean vectors for each class obtained from the LDA training.

Returns:

A numpy array of shape (n\_samples,) containing the predicted class labels.

"""

# Project new data points onto the LDA subspace

projected\_data = X\_new @ w

# Calculate distances to each class mean in the projected space

distances = np.linalg.norm(projected\_data[:, np.newaxis] - mean\_vectors, axis=2)

# Predict the class label based on the minimum distance

predicted\_labels = classes[np.argmin(distances, axis=1)]

return predicted\_labels

# Example usage

X = np.array([[1, 2], [1.5, 1.8], [5, 8], [8, 8], [1, 0.6], [9, 11]])

y = np.array([0, 0, 1, 1, 2, 2])

w, mean\_vectors = lda(X, y)

predicted\_labels = predict(np.array([[3, 5]]), w, mean\_vectors)

print("Predicted label:", predicted\_labels[0])

**11. Implement interpolation using langrage’s method.**

import numpy as np

def lagrange\_interpolation(x\_values, y\_values):

n = len(x\_values)

assert n == len(y\_values), "Number of x\_values must be equal to number of y\_values"

# Define Lagrange basis polynomials

def lagrange\_basis(x, k):

basis = 1

for j in range(n):

if j != k:

basis \*= (x - x\_values[j]) / (x\_values[k] - x\_values[j])

return basis

# Construct the Lagrange interpolation polynomial

def lagrange\_polynomial(x):

return sum(y\_values[k] \* lagrange\_basis(x, k) for k in range(n))

# Return the polynomial as a numpy poly1d object

return np.poly1d(lagrange\_polynomial)

# Example usage:

# Given data points

x\_data = np.array([0, 1, 2, 3])

y\_data = np.array([1, 3, 5, 4])

# Perform Lagrange interpolation

interpolated\_poly = lagrange\_interpolation(x\_data, y\_data)

# Print the interpolated polynomial

print("Interpolated Polynomial:")

print(interpolated\_poly)

# Evaluate the interpolated polynomial at specific points

x\_eval = np.linspace(0, 3, 100) # Evaluate from x = 0 to x = 3

y\_eval = interpolated\_poly(x\_eval)

# Plot the original data points and the interpolated polynomial

import matplotlib.pyplot as plt

plt.figure(figsize=(8, 6))

plt.plot(x\_data, y\_data, 'ro', label='Original Data')

plt.plot(x\_eval, y\_eval, label='Interpolated Polynomial')

plt.xlabel('x')

plt.ylabel('y')

plt.title('Lagrange Interpolation')

plt.legend()

plt.grid(True)

plt.show()

**12. Implement interpolation using Newton divided difference method.**

import numpy as np

def newton\_divided\_difference(x\_values, y\_values):

n = len(x\_values)

assert n == len(y\_values), "Number of x\_values must be equal to number of y\_values"

# Initialize the divided difference table

divided\_diff\_table = np.zeros((n, n))

divided\_diff\_table[:, 0] = y\_values

# Populate the divided difference table

for j in range(1, n):

for i in range(n - j):

divided\_diff\_table[i, j] = (divided\_diff\_table[i + 1, j - 1] - divided\_diff\_table[i, j - 1]) / (x\_values[i + j] - x\_values[i])

# Construct the interpolation polynomial using the coefficients from the table

def newton\_interpolation\_polynomial(x):

result = divided\_diff\_table[0, 0]

for j in range(1, n):

term = divided\_diff\_table[0, j]

for i in range(j):

term \*= (x - x\_values[i])

result += term

return result

# Return the polynomial as a numpy poly1d object

return np.poly1d(newton\_interpolation\_polynomial)

# Example usage:

# Given data points

x\_data = np.array([0, 1, 2, 3])

y\_data = np.array([1, 3, 5, 4])

# Perform Newton divided difference interpolation

interpolated\_poly = newton\_divided\_difference(x\_data, y\_data)

# Print the interpolated polynomial

print("Interpolated Polynomial:")

print(interpolated\_poly)

# Evaluate the interpolated polynomial at specific points

x\_eval = np.linspace(0, 3, 100) # Evaluate from x = 0 to x = 3

y\_eval = interpolated\_poly(x\_eval)

# Plot the original data points and the interpolated polynomial

import matplotlib.pyplot as plt

plt.figure(figsize=(8, 6))

plt.plot(x\_data, y\_data, 'ro', label='Original Data')

plt.plot(x\_eval, y\_eval, label='Interpolated Polynomial')

plt.xlabel('x')

plt.ylabel('y')

plt.title('Newton Divided Difference Interpolation')

plt.legend()

plt.grid(True)

plt.show()

**13. Implement regression using Hermite and least square method.**

import numpy as np

def polynomial\_regression\_least\_squares(x\_values, y\_values, degree):

n = len(x\_values)

# Create the Vandermonde matrix

X = np.vander(x\_values, degree + 1, increasing=True)

# Compute the least squares solution

coefficients = np.linalg.lstsq(X, y\_values, rcond=None)[0]

return coefficients

# Example usage:

# Generate synthetic data points

np.random.seed(0)

x\_data = np.linspace(0, 1, 20)

y\_data = np.sin(2 \* np.pi \* x\_data) + np.random.normal(0, 0.2, size=x\_data.shape)

# Degree of the polynomial

degree = 3

# Perform polynomial regression using least squares method

coefficients = polynomial\_regression\_least\_squares(x\_data, y\_data, degree)

# Print the coefficients of the fitted polynomial

print("Coefficients of the fitted polynomial (from highest degree to lowest):")

print(coefficients)

# Evaluate the fitted polynomial

def evaluate\_polynomial(coefficients, x):

"""Evaluate the polynomial defined by coefficients at the point x."""

return np.polyval(coefficients[::-1], x)

# Plot the original data points and the fitted polynomial

import matplotlib.pyplot as plt

# Generate points for plotting the fitted polynomial

x\_plot = np.linspace(0, 1, 100)

y\_plot = evaluate\_polynomial(coefficients, x\_plot)

plt.figure(figsize=(8, 6))

plt.plot(x\_data, y\_data, 'bo', label='Original Data')

plt.plot(x\_plot, y\_plot, 'r-', label='Fitted Polynomial (Degree {})'.format(degree))

plt.xlabel('x')

plt.ylabel('y')

plt.title('Polynomial Regression Using Least Squares Method')

plt.legend()

plt.grid(True)

plt.show()

**11. Implement interpolation using langrage’s method.**

import numpy as np

def lagrange\_interpolation(x\_values, y\_values):

n = len(x\_values)

assert n == len(y\_values), "Number of x\_values must be equal to number of y\_values"

# Define Lagrange basis polynomials

def lagrange\_basis(x, k):

basis = 1

for j in range(n):

if j != k:

basis \*= (x - x\_values[j]) / (x\_values[k] - x\_values[j])

return basis

# Construct the Lagrange interpolation polynomial

def lagrange\_polynomial(x):

return sum(y\_values[k] \* lagrange\_basis(x, k) for k in range(n))

# Return the polynomial as a numpy poly1d object

return np.poly1d(lagrange\_polynomial)

# Example usage:

# Given data points

x\_data = np.array([0, 1, 2, 3])

y\_data = np.array([1, 3, 5, 4])

# Perform Lagrange interpolation

interpolated\_poly = lagrange\_interpolation(x\_data, y\_data)

# Print the interpolated polynomial

print("Interpolated Polynomial:")

print(interpolated\_poly)

# Evaluate the interpolated polynomial at specific points

x\_eval = np.linspace(0, 3, 100) # Evaluate from x = 0 to x = 3

y\_eval = interpolated\_poly(x\_eval)

# Plot the original data points and the interpolated polynomial

import matplotlib.pyplot as plt

plt.figure(figsize=(8, 6))

plt.plot(x\_data, y\_data, 'ro', label='Original Data')

plt.plot(x\_eval, y\_eval, label='Interpolated Polynomial')

plt.xlabel('x')

plt.ylabel('y')

plt.title('Lagrange Interpolation')

plt.legend()

plt.grid(True)

plt.show()

**15. Implement gradient decent method for parameter optimization of linear regression.**

import numpy as np

class LinearRegressionGradientDescent:

def \_\_init\_\_(self, learning\_rate=0.01, num\_iterations=1000):

self.learning\_rate = learning\_rate

self.num\_iterations = num\_iterations

self.intercept = None

self.slope = None

def fit(self, X, y):

# Initialize parameters

self.intercept = 0

self.slope = 0

n = len(X)

# Gradient descent optimization

for \_ in range(self.num\_iterations):

# Compute predictions

y\_pred = self.intercept + self.slope \* X

# Compute gradients

grad\_intercept = (-2/n) \* np.sum(y - y\_pred)

grad\_slope = (-2/n) \* np.sum(X \* (y - y\_pred))

# Update parameters

self.intercept -= self.learning\_rate \* grad\_intercept

self.slope -= self.learning\_rate \* grad\_slope

def predict(self, X):

return self.intercept + self.slope \* X

# Example usage:

# Generate synthetic data

np.random.seed(0)

X = 2 \* np.random.rand(100, 1)

y = 4 + 3 \* X + np.random.randn(100, 1)

# Instantiate and fit the model

model = LinearRegressionGradientDescent(learning\_rate=0.1, num\_iterations=1000)

model.fit(X, y)

# Print the learned parameters

print("Intercept:", model.intercept)

print("Slope:", model.slope)

# Make predictions

X\_new = np.array([[0], [2]])

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

print("Predictions:")

print(y\_pred)