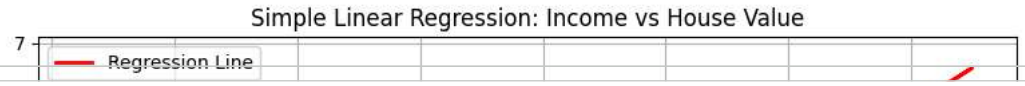
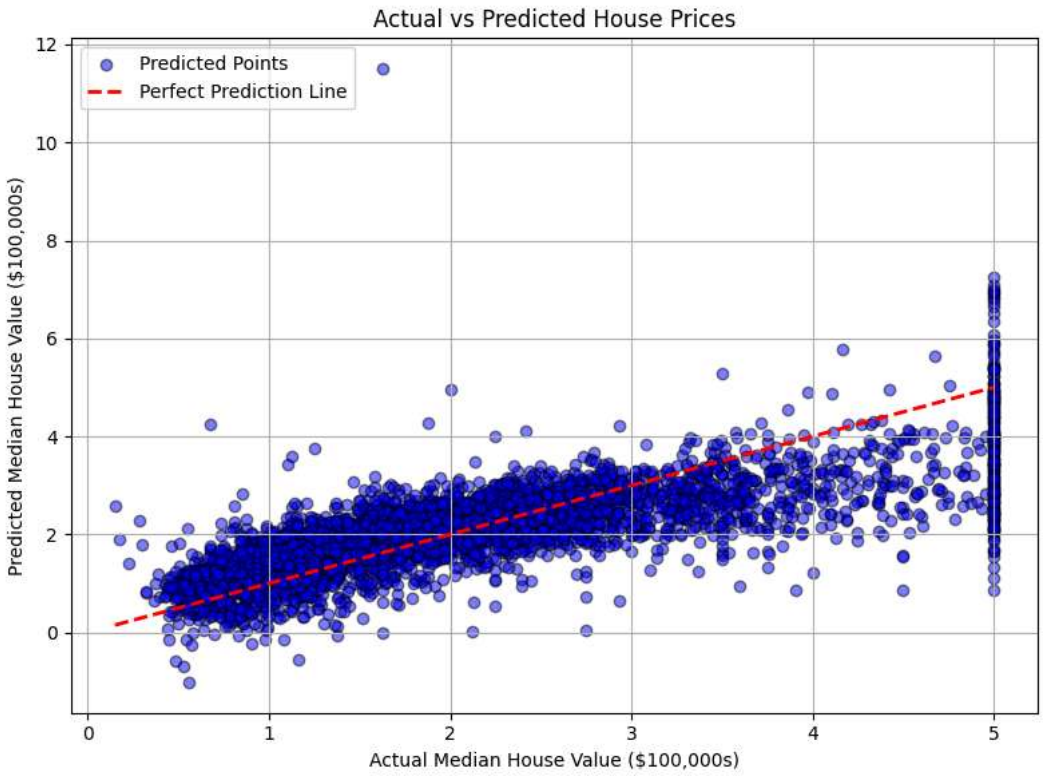
```
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.datasets import fetch_california_housing
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error, r2_score

# Load dataset
data = fetch_california_housing(as_frame=True)
df = data.frame
print(df.head(10))

# Use only one feature: Median Income
X = df[['MedInc']]  # Independent variable
y = df['MedHouseVal']  # Dependent variable (target)

# Split data
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Create and train model
model = LinearRegression()
model.fit(X_train, y_train)

# Make predictions
y_pred = model.predict(X_test)

# Evaluate model
mse = mean_squared_error(y_test, y_pred)
r2 = r2_score(y_test, y_pred)

# Print results
print("Simple Linear Regression - California Housing Dataset")
print(f"Mean Squared Error (MSE): {mse:.4f}")
print(f"R² Score: {r2:.4f}")

# Plot results
plt.figure(figsize=(8, 6))
plt.scatter(X_test, y_test)
plt.plot(X_test, y_pred, color='red', linewidth=2, label='Regression Line')
plt.xlabel('Median Income (in $10,000s)')
plt.ylabel('Median House Value (in $100,000s)')
plt.title('Simple Linear Regression: Income vs House Value')
plt.legend()
plt.grid(True)
plt.tight_layout()
plt.show()
```

```
   MedInc  HouseAge  AveRooms  AveBedrms  Population  AveOccup  Latitude  \
0  8.3252      41.0  6.984127   1.023810       322.0  2.555556     37.88
1  8.3014      21.0  6.238137   0.971880      2401.0  2.109842     37.86
2  7.2574      52.0  8.288136   1.073446       496.0  2.802260     37.85
3  5.6431      52.0  5.817352   1.073059       558.0  2.547945     37.85
4  3.8462      52.0  6.281853   1.081081       565.0  2.181467     37.85
5  4.0368      52.0  4.761658   1.103627       413.0  2.139896     37.85
6  3.6591      52.0  4.931907   0.951362      1094.0  2.128405     37.84
7  3.1200      52.0  4.797527   1.061824      1157.0  1.788253     37.84
8  2.0804      42.0  4.294118   1.117647      1206.0  2.026891     37.84
9  3.6912      52.0  4.970588   0.990196      1551.0  2.172269     37.84

   Longitude  MedHouseVal
0    -122.23        4.526
1    -122.22        3.585
2    -122.24        3.521
3    -122.25        3.413
4    -122.25        3.422
5    -122.25        2.697
6    -122.25        2.992
7    -122.25        2.414
8    -122.26        2.267
9    -122.25        2.611
Simple Linear Regression - California Housing Dataset
Mean Squared Error (MSE): 0.7091
R² Score: 0.4589
```



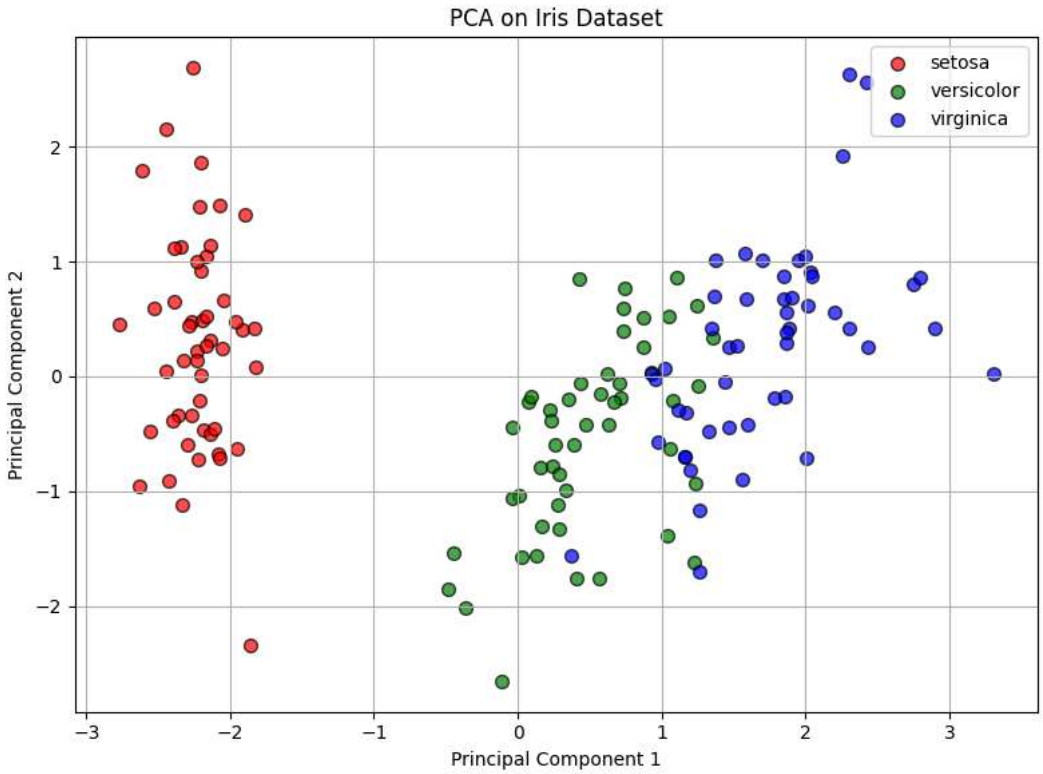

```
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.datasets import fetch_california_housing
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error, r2_score

data = fetch_california_housing(as_frame=True)
df = data.frame

X = df.drop(columns='MedHouseVal')
y = df['MedHouseVal']  # Median house value (in $100,000s)

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

mlr = LinearRegression()
mlr.fit(X_train, y_train)

y_pred = mlr.predict(X_test)

mse = mean_squared_error(y_test, y_pred)
r2 = r2_score(y_test, y_pred)

print("Multiple Linear Regression - California Housing Dataset")
print(f"Mean Squared Error (MSE): {mse:.4f}")
print(f"R² Score: {r2:.4f}")

plt.figure(figsize=(8, 6))
plt.scatter(y_test, y_pred, color='blue', edgecolors='black', alpha=0.5, label='Predicted Points')
plt.plot([y.min(), y.max()], [y.min(), y.max()], 'r--', lw=2, label='Perfect Prediction Line')
plt.xlabel('Actual Median House Value ($100,000s)')
plt.ylabel('Predicted Median House Value ($100,000s)')
plt.title('Actual vs Predicted House Prices')
plt.legend(loc='upper left')
plt.grid(True)
plt.tight_layout()
plt.show()
```

```
Multiple Linear Regression - California Housing Dataset
Mean Squared Error (MSE): 0.5559
R² Score: 0.5758
```



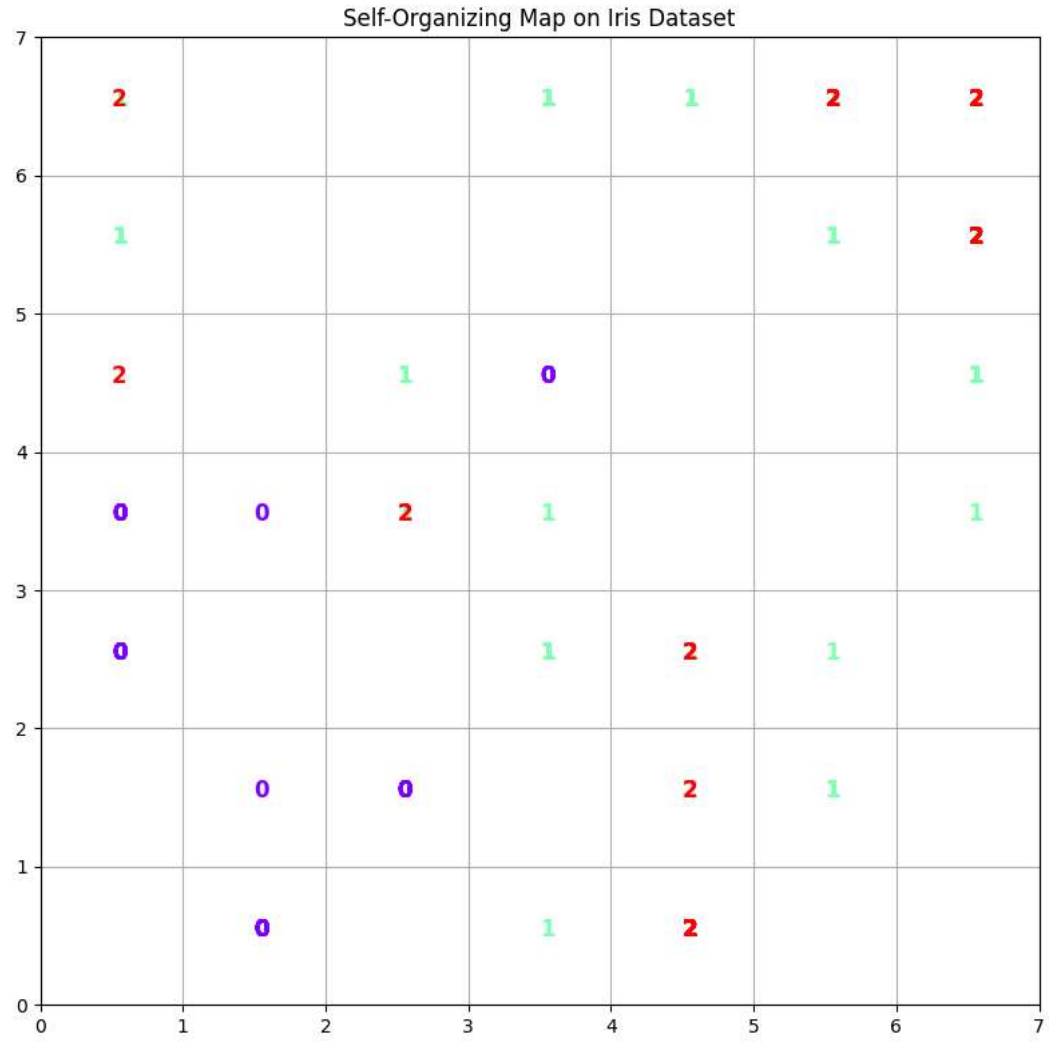

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import load_iris
from sklearn.decomposition import PCA
from sklearn.preprocessing import StandardScaler

iris = load_iris()
X = iris.data  # Features (4D)
y = iris.target  # Labels (0, 1, 2)
target_names = iris.target_names

# Standardize features
X_std = StandardScaler().fit_transform(X)

# Apply PCA to reduce from 4D to 2D
pca = PCA(n_components=2)
X_pca = pca.fit_transform(X_std)

# Print explained variance
print("Explained variance ratio:", pca.explained_variance_ratio_)
print(f"Total variance explained: {sum(pca.explained_variance_ratio_):.2%}")

# Visualize PCA results
plt.figure(figsize=(8, 6))
colors = ['red', 'green', 'blue']

for color, i, target_name in zip(colors, [0, 1, 2], target_names):
    plt.scatter(X_pca[y == i, 0], X_pca[y == i, 1],
                color=color, edgecolors='black', s=50,
                alpha=0.7, label=target_name)

plt.xlabel('Principal Component 1')
plt.ylabel('Principal Component 2')
plt.title('PCA on Iris Dataset')
plt.legend()
plt.grid(True)
plt.tight_layout()
plt.show()
```

```
Explained variance ratio: [0.72962445 0.22850762]
Total variance explained: 95.81%
```




```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import load_iris
from sklearn.preprocessing import MinMaxScaler
from minisom import MiniSom

# Load Iris dataset
iris = load_iris()
X = iris.data  # 4 features
y = iris.target  # Labels (0, 1, 2)

# Normalize features to [0, 1] range
scaler = MinMaxScaler()
X_scaled = scaler.fit_transform(X)

# Create SOM: 7x7 grid
som = MiniSom(x=7, y=7, input_len=X_scaled.shape[1],
              sigma=1.0, learning_rate=0.5)
som.random_weights_init(X_scaled)

# Train SOM
print("Training SOM...")
som.train_random(X_scaled, num_iteration=100)
print("Training completed.")

# Visualize SOM
plt.figure(figsize=(8, 8))

for i, x in enumerate(X_scaled):
    w = som.winner(x)  # Best Matching Unit (BMU)
    plt.text(w[0] + 0.5, w[1] + 0.5, str(y[i]),
             color=plt.cm.rainbow(y[i] / 2),
             fontdict={'size': 12, 'weight': 'bold'})

plt.xlim([0, som.get_weights().shape[0]])
plt.ylim([0, som.get_weights().shape[1]])
plt.title("Self-Organizing Map on Iris Dataset")
plt.grid(True)
plt.tight_layout()
plt.show()
```

```
Training SOM...
Training completed.
```




```
!pip install MiniSom
```

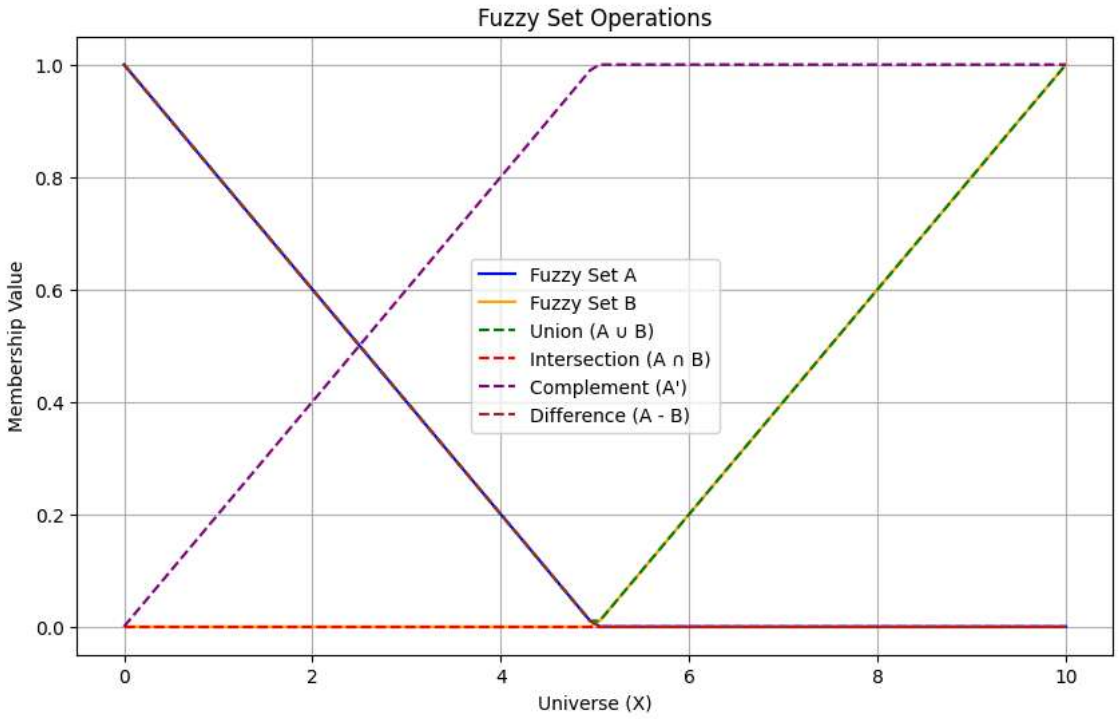
```
Collecting MiniSom
  Downloading minisom-2.3.5.tar.gz (12 kB)
  Preparing metadata (setup.py) ... done
Building wheels for collected packages: MiniSom
  Building wheel for MiniSom (setup.py) ... done
  Created wheel for MiniSom: filename=MiniSom-2.3.5-py3-none-any.whl size=12031 sha256=17bced7512ea6aee247b361b456d07d9ebd86ca3c
  Stored in directory: /root/.cache/pip/wheels/0f/8c/a4/5b7aa56fa6ef11d536d45da775bcc5a2a1c163ff0f8f11990b
Successfully built MiniSom
Installing collected packages: MiniSom
Successfully installed MiniSom-2.3.5
```

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn import datasets
from sklearn.preprocessing import StandardScaler, OneHotEncoder
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score

# Supervised Self-Organizing Map
class SSOM:
    def __init__(self, m, n, input_dim, n_classes, lr=0.5, sigma=None):
        self.m, self.n = m, n  # Grid dimensions
        self.grid_size = m * n  # Total neurons
        self.input_dim, self.n_classes = input_dim, n_classes
        self.lr0, self.sigma0 = lr, sigma if sigma else max(m, n) / 2

        # Initialize weights for features and labels
        self.weights = np.random.randn(self.grid_size, input_dim)
        self.label_weights = np.random.randn(self.grid_size, n_classes) * 0.01
```

```
        self.coords = np.array([(i, j) for i in range(m) for j in range(n)])

    def _decay(self, val, epoch, max_epoch):
        """Exponential decay for learning rate and sigma"""
        return val * np.exp(-epoch / max_epoch)

    def _bmu(self, x):
        """Find Best Matching Unit"""
        return np.argmin(np.linalg.norm(self.weights - x, axis=1))

    def _neigh(self, bmu, sigma):
        """Calculate neighborhood function"""
        d2 = np.sum((self.coords - self.coords[bmu])**2, axis=1)
        return np.exp(-d2 / (2 * sigma**2))

    def fit(self, X, y, epochs=50):
        """Train SSOM"""
        self.hist = []
        for e in range(epochs):
            lr = self._decay(self.lr0, e, epochs)
            sigma = self._decay(self.sigma0, e, epochs)

            for x, yv in zip(X, y):
                bmu = self._bmu(x)
                h = self._neigh(bmu, sigma)[:, None]

                # Update feature weights
                self.weights += lr * h * (x - self.weights)
                # Update label weights (supervised part)
                self.label_weights += lr * h * (yv - self.label_weights)

            self.hist.append(accuracy_score(np.argmax(y, 1), self.predict(X)))

    def predict(self, X):
        """Predict class labels"""
        return [np.argmax(self.label_weights[self._bmu(x)]) for x in X]


# Load and preprocess data
iris = datasets.load_iris()
X, y = iris.data, iris.target

# Standardize features
X = StandardScaler().fit_transform(X)

# One-hot encode labels
oh = OneHotEncoder(sparse_output=False)
y_oh = oh.fit_transform(y.reshape(-1, 1))

# Split data
Xtr, Xte, ytr, yte = train_test_split(X, y_oh, test_size=0.3,
                                      random_state=42, stratify=y)

# Train SSOM
som = SSOM(m=6, n=6, input_dim=X.shape[1], n_classes=3, lr=0.5)
som.fit(Xtr, ytr, epochs=60)

# Predict and evaluate
ypred = som.predict(Xte)
print("Test accuracy:", accuracy_score(np.argmax(yte, 1), ypred))

# Plot training progress
plt.figure(figsize=(8, 5))
plt.plot(som.hist, linewidth=2)
plt.xlabel("Epoch")
plt.ylabel("Train Accuracy")
plt.title("SSOM Training Accuracy")
plt.grid(True)
plt.tight_layout()
plt.show()
```

```
Test accuracy: 0.9111111111111111
```


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

# Universe of Discourse
X = np.linspace(0, 10, 100)

# Fuzzy Set A and B (Triangular Membership Functions)
A = np.maximum(0, np.minimum((5 - X) / 5, 1))
B = np.maximum(0, np.minimum((X - 5) / 5, 1))

# Fuzzy Operations
fuzzy_union = np.maximum(A, B)
fuzzy_intersection = np.minimum(A, B)
fuzzy_complement_A = 1 - A
fuzzy_difference = np.minimum(A, 1 - B)

# Plotting
plt.figure(figsize=(10, 6))
plt.plot(X, A, label='Fuzzy Set A', color='blue')
plt.plot(X, B, label='Fuzzy Set B', color='orange')
plt.plot(X, fuzzy_union, '--', label='Union (A ∪ B)', color='green')
plt.plot(X, fuzzy_intersection, '--', label='Intersection (A ∩ B)', color='red')
plt.plot(X, fuzzy_complement_A, '--', label="Complement (A')", color='purple')
plt.plot(X, fuzzy_difference, '--', label='Difference (A - B)', color='brown')

plt.title('Fuzzy Set Operations')
plt.xlabel('Universe (X)')
plt.ylabel('Membership Value')
plt.legend()
plt.grid(True)
plt.show()

# Function to print fuzzy set values at intervals
def fuzzy_repr(name, X, M):
    pairs = [f"({x:.1f}, {m:.2f})" for x, m in zip(X[::10], M[::10])]
    return f"{name} = {{ " + ", ".join(pairs) + " }}"


print("\n===== FUZZY SET REPRESENTATIONS =====\n")
print(fuzzy_repr("A", X, A))
print(fuzzy_repr("B", X, B))
print(fuzzy_repr("Union (A ∪ B)", X, fuzzy_union))
print(fuzzy_repr("Intersection (A ∩ B)", X, fuzzy_intersection))
print(fuzzy_repr("Complement (A')", X, fuzzy_complement_A))
print(fuzzy_repr("Set Difference (A - B)", X, fuzzy_difference))
```




```
===== FUZZY SET REPRESENTATIONS =====

A = { (0.0, 1.00), (1.0, 0.80), (2.0, 0.60), (3.0, 0.39), (4.0, 0.19), (5.1, 0.00), (6.1, 0.00), (7.1, 0.00), (8.1, 0.00), (9.1,
B = { (0.0, 0.00), (1.0, 0.00), (2.0, 0.00), (3.0, 0.00), (4.0, 0.00), (5.1, 0.01), (6.1, 0.21), (7.1, 0.41), (8.1, 0.62), (9.1,
Union (A ∪ B) = { (0.0, 1.00), (1.0, 0.80), (2.0, 0.60), (3.0, 0.39), (4.0, 0.19), (5.1, 0.01), (6.1, 0.21), (7.1, 0.41), (8.1,
Intersection (A ∩ B) = { (0.0, 0.00), (1.0, 0.00), (2.0, 0.00), (3.0, 0.00), (4.0, 0.00), (5.1, 0.00), (6.1, 0.00), (7.1, 0.00),
Complement (A') = { (0.0, 0.00), (1.0, 0.20), (2.0, 0.40), (3.0, 0.61), (4.0, 0.81), (5.1, 1.00), (6.1, 1.00), (7.1, 1.00), (8.1
```

```
import numpy as np

# Define the function to optimize (maximize)
def f(x):
    return x * np.sin(10 * np.pi * x) + 1.0  # Example function

# GA Parameters
population_size = 20
generations = 50
crossover_prob = 0.8
mutation_prob = 0.1
x_bounds = [0, 1]  # variable bounds

# Initialize population randomly within bounds
population = np.random.uniform(x_bounds[0], x_bounds[1], population_size)

# Fitness function
def fitness(x):
    return f(x)

# Selection: Roulette Wheel
def select(pop, fit):
    fit_sum = np.sum(fit)
    probs = fit / fit_sum
    indices = np.random.choice(len(pop), size=len(pop), p=probs)
    return pop[indices]

# Crossover: Single-point (blend-based)
def crossover(parent1, parent2):
    if np.random.rand() < crossover_prob:
        alpha = np.random.rand()
        child1 = alpha * parent1 + (1 - alpha) * parent2
        child2 = alpha * parent2 + (1 - alpha) * parent1
        return child1, child2
    else:
        return parent1, parent2

# Mutation
def mutate(x):
    if np.random.rand() < mutation_prob:
```

```
            x += np.random.uniform(-0.1, 0.1)
            x = np.clip(x, x_bounds[0], x_bounds[1])
        return x

    # Main GA loop
    best_history = []
    for gen in range(generations):
        fit_values = fitness(population)
        best_idx = np.argmax(fit_values)
        best_history.append(fit_values[best_idx])

        # Selection
        population = select(population, fit_values)

        # Crossover
        new_population = []
        for i in range(0, population_size, 2):
            parent1, parent2 = population[i], population[i+1]
            child1, child2 = crossover(parent1, parent2)
            new_population.extend([child1, child2])

        # Mutation
        population = np.array([mutate(x) for x in new_population])

    # Final best solution
    fit_values = fitness(population)
    best_idx = np.argmax(fit_values)
    best_x = population[best_idx]
    best_y = fit_values[best_idx]

    print("\n===== GENETIC ALGORITHM OPTIMIZATION =====")
    print(f"Best solution x: {best_x:.6f}")
    print(f"Maximum value f(x): {best_y:.6f}")
    print("==========================================\n")

    # Plot fitness over generations
    import matplotlib.pyplot as plt

    plt.plot(best_history, 'r-o')
    plt.title('Best Fitness Over Generations')
    plt.xlabel('Generation')
    plt.ylabel('Fitness')
    plt.grid(True)
    plt.show()
```

```
===== GENETIC ALGORITHM OPTIMIZATION =====
Best solution x: 0.637560
Maximum value f(x): 1.589484
==========================================
```