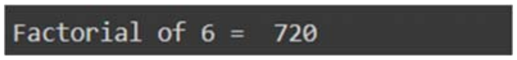
# Experiment 1

**Aim:** Write a program in Python to compute the factorial of any given number.

## Code:

```
fact = lambda x: x*fact(x-1) if x!=1 else 1
print("Factorial of 6 = ", fact(6))
```

## Output:



```
Factorial of 6 =  720
```

# Experiment 2

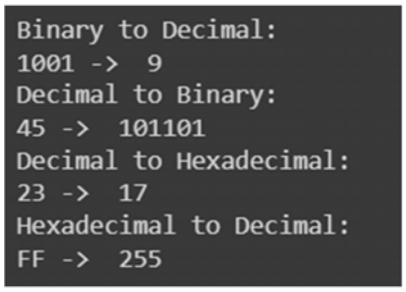
**Aim:** Write a program in Python to convert:

a. Binary to Decimal
b. Decimal to Binary
c. Decimal to Hexadecimal
d. Hexadecimal to Decimal

## Code:

```
b2d = lambda x: int(x, 2)
print("Binary to Decimal:")
print("1001 -> ", b2d("1001"))
d2b = lambda x: bin(x).replace("0b", "")
print("Decimal to Binary:")
print("45 -> ", d2b(45))
d2h = lambda x: hex(x).replace("0x", "")
print("Decimal to Hexadecimal:")
print("23 -> ", d2h(23))
h2d = lambda x: int(x, 16)
print("Hexadecimal to Decimal:")
print("FF -> ", h2d("FF"))
```

## Output:



```
Binary to Decimal:
1001 ->  9
Decimal to Binary:
45 ->  101101
Decimal to Hexadecimal:
23 ->  17
Hexadecimal to Decimal:
FF ->  255
```

# Experiment 3

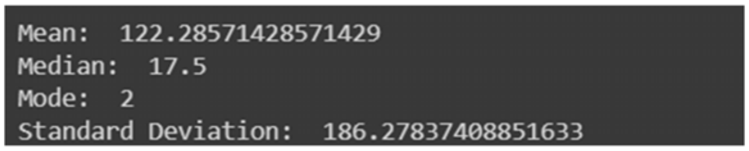
**Aim:** Write a program to compute the:

a. Mean
b. Mode
c. Median
d. Standard Deviation

## Code:

```
import numpy as np
from scipy import stats
mean = lambda x: np.mean(x)
median = lambda x: np.median(x)
mode = lambda x: stats.mode(x).mode
std = lambda x: np.std(x)

data = [2,12,43,5,46,6,8,4,241,353,635,323,23,11]
print("Mean: ",mean(data))
print("Median: ", median(data))
print("Mode: ", mode(data))
print("Standard Deviation: ", std(data))
```

## Output:



```
Mean:  122.28571428571429
Median:  17.5
Mode:  2
Standard Deviation:  186.27837408851633
```

## Experiment 4

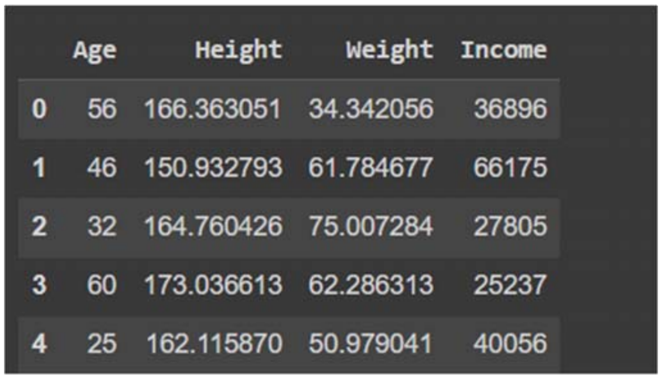
**Aim:** Write a program to represent a data set and its single-variate/multi-variate plots.

### Code:

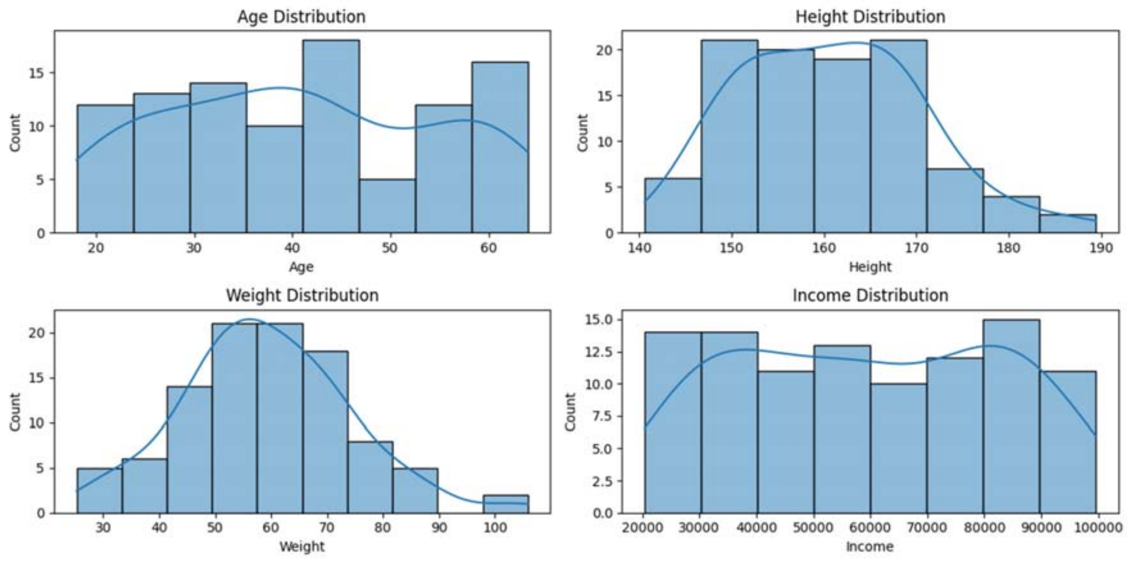
```
import numpy as np
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt

# Generate a sample data set
np.random.seed(42)
num_samples = 100
data = {
    'Age': np.random.randint(18, 65, num_samples),
    'Height': np.random.normal(160, 10, num_samples),
    'Weight': np.random.normal(60, 15, num_samples),
    'Income': np.random.randint(20000, 100000, num_samples)
}

# Create a pandas DataFrame
df = pd.DataFrame(data)
display(df.head())

# Single variate plots
plt.figure(figsize=(12, 6))

plt.subplot(2, 2, 1)
sns.histplot(df['Age'], kde=True)
plt.title('Age Distribution')

plt.subplot(2, 2, 2)
sns.histplot(df['Height'], kde=True)
plt.title('Height Distribution')

plt.subplot(2, 2, 3)
sns.histplot(df['Weight'], kde=True)
plt.title('Weight Distribution')

plt.subplot(2, 2, 4)
sns.histplot(df['Income'], kde=True)
plt.title('Income Distribution')

plt.tight_layout()
plt.show()

# Multi-variate plots
plt.figure(figsize=(10, 6))
```

```
sns.pairplot(df)
plt.suptitle('Pairwise Relationships', y=1.02)
plt.show()
```

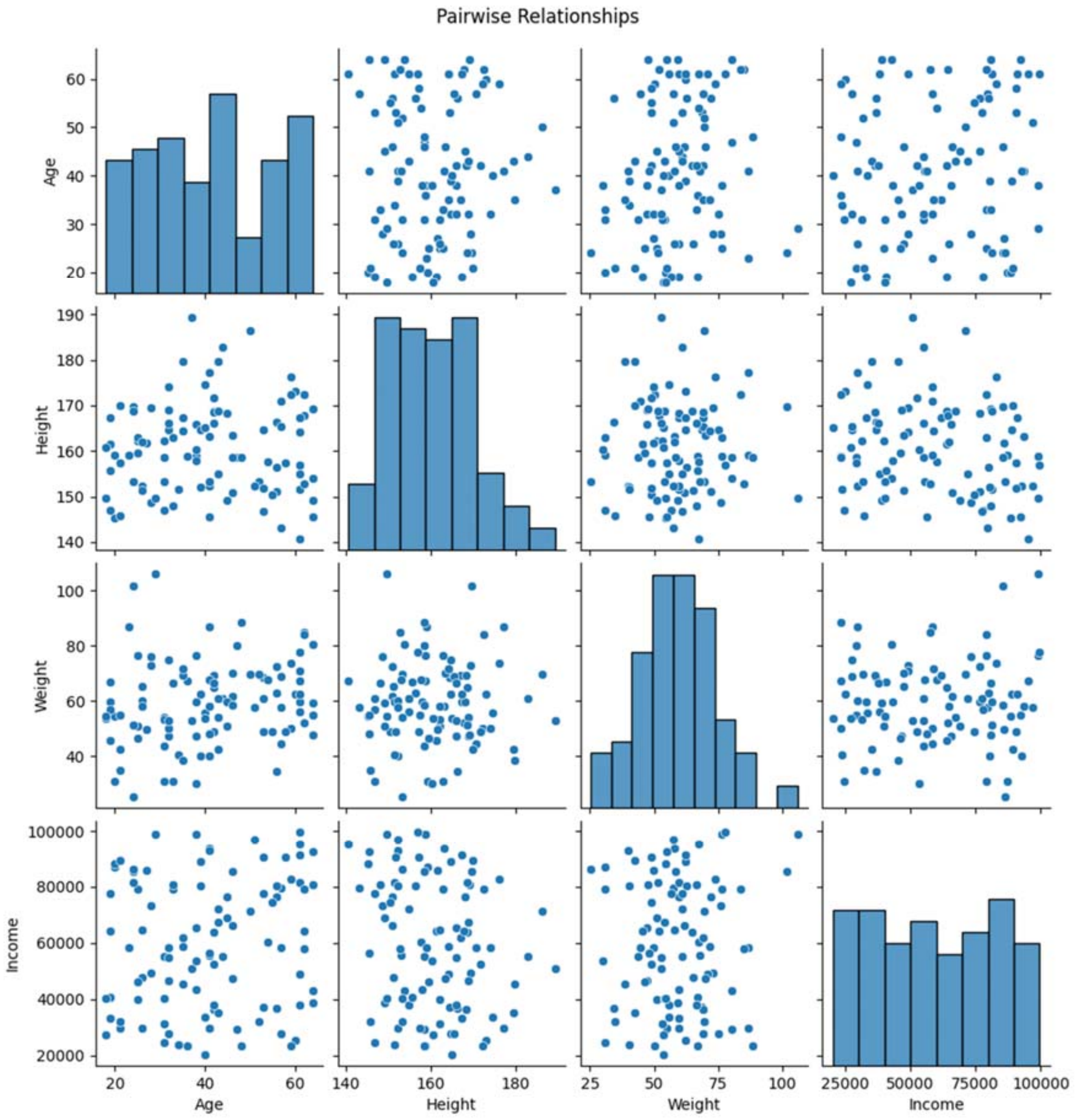
**Output:**



|   | Age | Height | Weight | Income |
|---|---|---|---|---|
| 0 | 56 | 166.363051 | 34.342056 | 36896 |
| 1 | 46 | 150.932793 | 61.784677 | 66175 |
| 2 | 32 | 164.760426 | 75.007284 | 27805 |
| 3 | 60 | 173.036613 | 62.286313 | 25237 |
| 4 | 25 | 162.115870 | 50.979041 | 40056 |



```
<Figure size 1000x600 with 0 Axes>
```


Pairwise Relationships
Age
Height
Weight
Income
Age
Height
Weight
Income

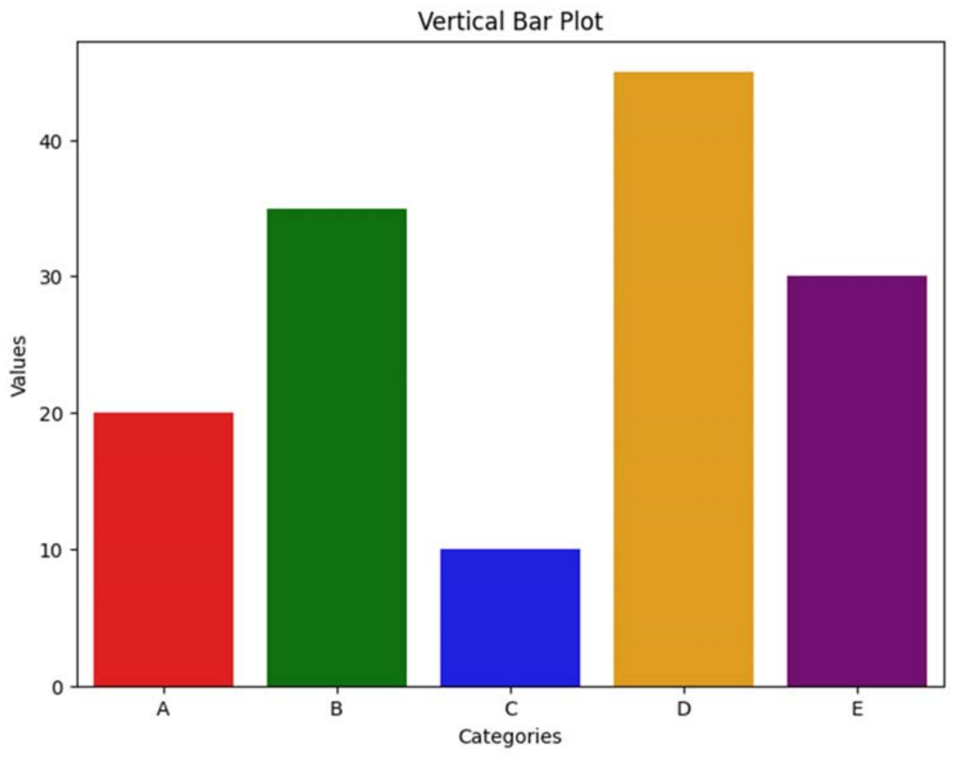
## Experiment 5

**Aim:** Write a program to plot a horizontal and vertical bar plot using Python. Change the colour of each bar also.

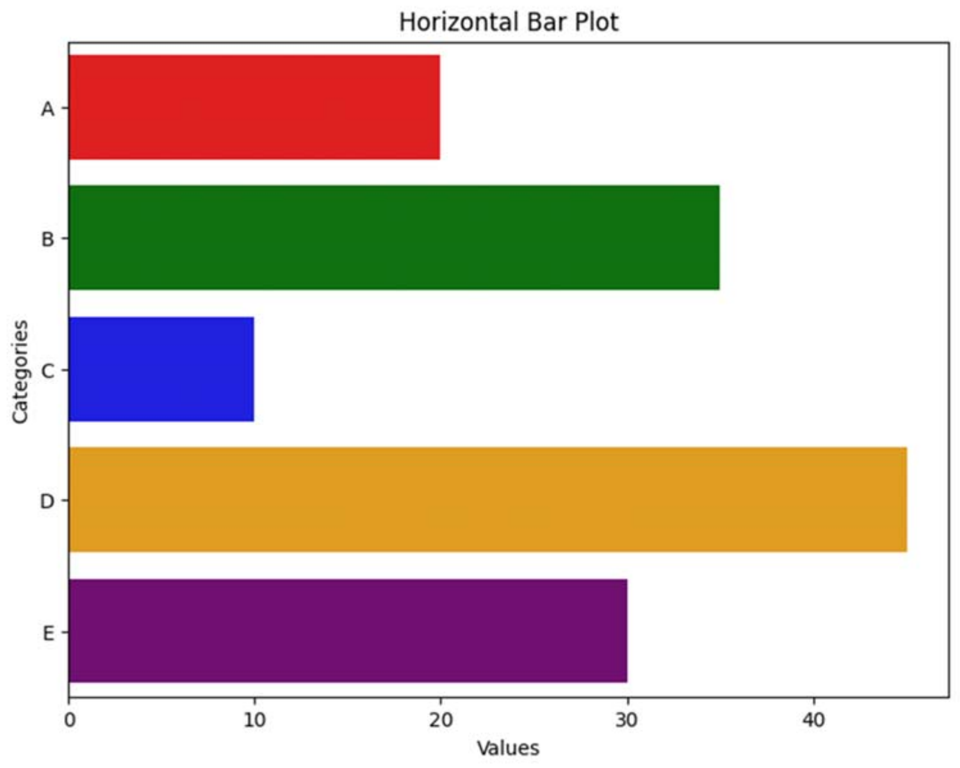
### Code:

```
import numpy as np
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt

# Sample data
data = {
    'Categories': ['A', 'B', 'C', 'D', 'E'],
    'Values': [20, 35, 10, 45, 30],
    'Colors': ['red', 'green', 'blue', 'orange', 'purple']
}

# Create a pandas DataFrame
df = pd.DataFrame(data)

# Vertical bar plot
plt.figure(figsize=(8, 6))
sns.barplot(x='Categories', y='Values', data=df,
palette=df['Colors'])
plt.title('Vertical Bar Plot')
plt.xlabel('Categories')
plt.ylabel('Values')
plt.show()

# Horizontal bar plot
plt.figure(figsize=(8, 6))
sns.barplot(x='Values', y='Categories', data=df,
palette=df['Colors'])
plt.title('Horizontal Bar Plot')
plt.xlabel('Values')
plt.ylabel('Categories')
plt.show()
```

**Output:**



Vertical Bar Plot
Values
Categories
A
B
C
D
E
0
10
20
30
40



Horizontal Bar Plot
Categories
Values
A
B
C
D
E
0
10
20
30
40

## Experiment 6

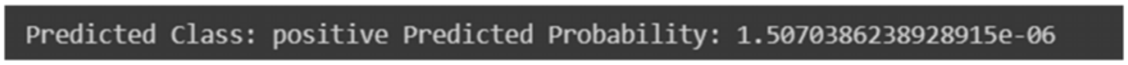
**Aim:** Implement Naïve Bayes theorem to classify the English text.

### Code:

```
import numpy as np
import pandas as pd

# Sample training data
data = {
    'text': [
        "I love this product",
        "This is great",
        "Not good at all",
        "Awful experience",
        "Excellent quality"
    ],
    'label': ['positive', 'positive', 'negative', 'negative',
'positive']
}

# Create a pandas DataFrame
df = pd.DataFrame(data)

# Separate data by class labels
positive_data = df[df['label'] == 'positive']
negative_data = df[df['label'] == 'negative']

# Combine all text data for each class
positive_text = ' '.join(positive_data['text']).split()
negative_text = ' '.join(negative_data['text']).split()

# Calculate prior probabilities
total_documents = len(df)
prior_positive = len(positive_data) / total_documents
prior_negative = len(negative_data) / total_documents

# Create vocabulary
vocabulary = list(set(positive_text + negative_text))

# Calculate word frequencies in each class
positive_word_freq = {word: positive_text.count(word) for word
in vocabulary}
negative_word_freq = {word: negative_text.count(word) for word
in vocabulary}

# Classify new text
new_text = "This is a good product"
new_text_words = new_text.split()
```

```
# Calculate likelihoods using Laplace smoothing
smoothing_factor = 1

likelihood_positive = 1
for word in new_text_words:
    likelihood_positive *= (positive_word_freq.get(word, 0) +
smoothing_factor) / (len(positive_text) + smoothing_factor *
len(vocabulary))

likelihood_negative = 1
for word in new_text_words:
    likelihood_negative *= (negative_word_freq.get(word, 0) +
smoothing_factor) / (len(negative_text) + smoothing_factor *
len(vocabulary))

# Apply Naive Bayes formula
posterior_positive = prior_positive * likelihood_positive
posterior_negative = prior_negative * likelihood_negative

# Classify based on the higher posterior probability
predicted_class = 'positive' if posterior_positive >
posterior_negative else 'negative'

print("Predicted Class:", predicted_class, "Predicted
Probability:", (posterior_positive)/(posterior_positive +
prior_negative))
```

**Output:**



```
Predicted Class: positive Predicted Probability: 1.5070386238928915e-06
```

# Experiment 7

**Aim:** Write a program to implement the KNN Algorithm.

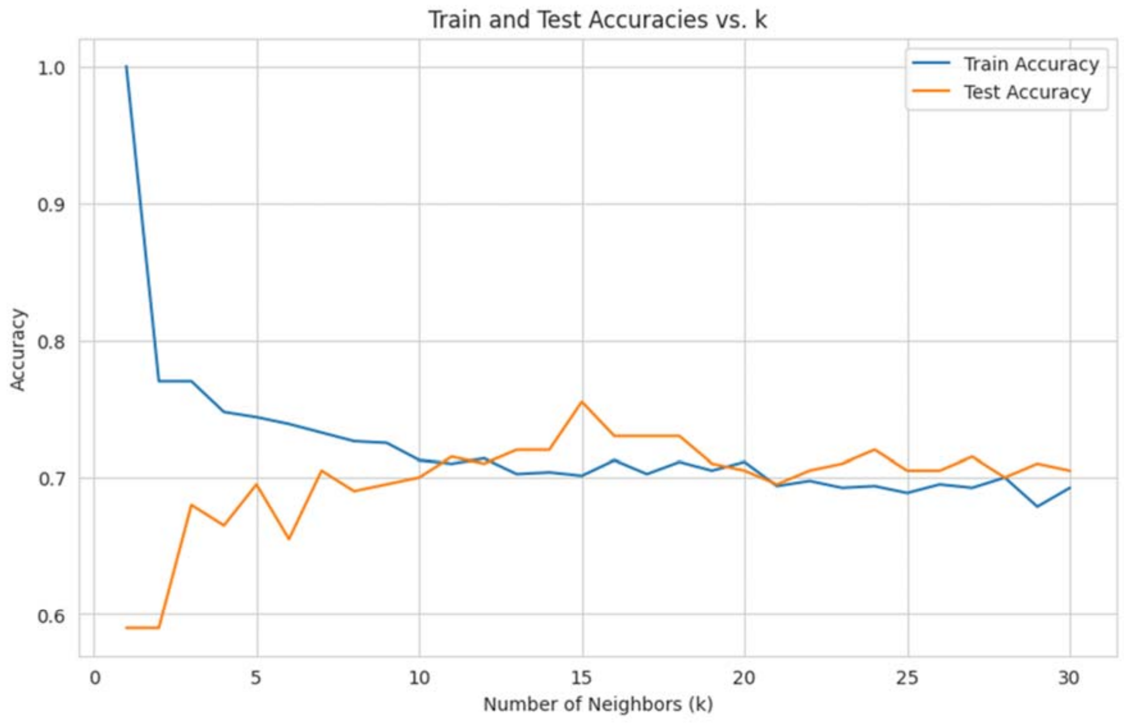
## Code:

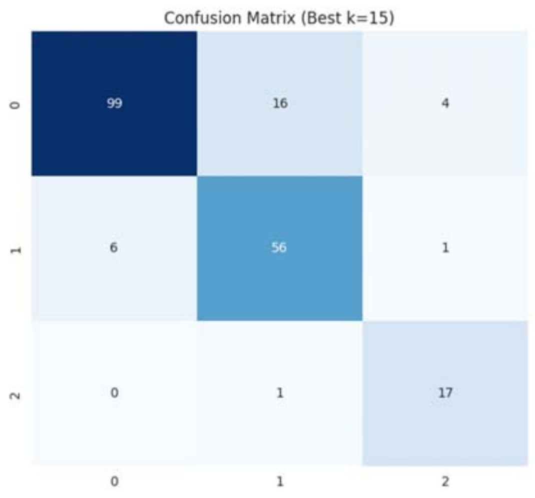
```
import numpy as np
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
from sklearn.datasets import make_multilabel_classification
from sklearn.model_selection import train_test_split
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import accuracy_score, confusion_matrix
from sklearn.cluster import KMeans
from tqdm.auto import tqdm
# Generate synthetic multilabel classification data
X, y = make_multilabel_classification(n_samples=1000,
n_classes=3, n_labels=2, random_state=42)

# Split the data into a training and testing set
X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.2, random_state=42)

# Initialize lists to store train and test accuracies
train_accuracies = []
test_accuracies = []

# Initialize variables to store best and worst k
best_k = None
worst_k = None
best_accuracy = 0
worst_accuracy = 1

# Train KNN classifiers for different values of k
for k in tqdm(range(1, 31)):
    knn = KNeighborsClassifier(n_neighbors=k)
    knn.fit(X_train, y_train)

    # Predict on the training and test sets
    y_train_pred = knn.predict(X_train)
    y_test_pred = knn.predict(X_test)

    # Calculate accuracies
    train_accuracy = accuracy_score(y_train, y_train_pred)
    test_accuracy = accuracy_score(y_test, y_test_pred)

    # Store accuracies
    train_accuracies.append(train_accuracy)
    test_accuracies.append(test_accuracy)
    # Update best and worst k values
    if test_accuracy > best_accuracy:
        best_accuracy = test_accuracy
        best_k = k
    if test_accuracy < worst_accuracy:
        worst_accuracy = test_accuracy
        worst_k = k
```

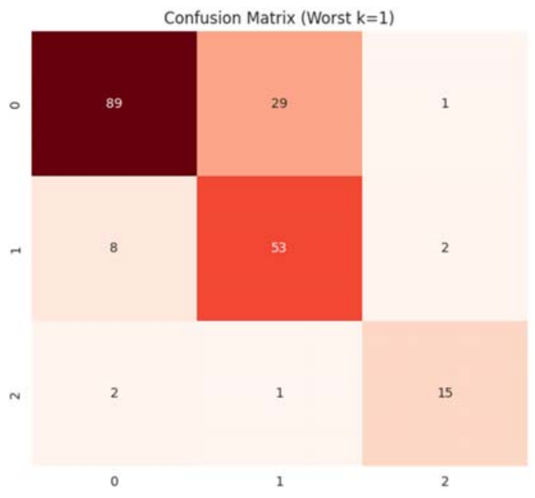
```
100% 30/30 [00:06<00:00, 5.41it/s]
```

```
# Plot train and test accuracies vs. k
plt.figure(figsize=(10, 6))
sns.set_style("whitegrid")
plt.plot(range(1, 31), train_accuracies, label="Train
Accuracy")
plt.plot(range(1, 31), test_accuracies, label="Test Accuracy")
plt.xlabel("Number of Neighbors (k)")
plt.ylabel("Accuracy")
plt.title("Train and Test Accuracies vs. k")
plt.legend()
plt.show()

# Train KNN with the best k and worst k values
best_knn = KNeighborsClassifier(n_neighbors=best_k)
worst_knn = KNeighborsClassifier(n_neighbors=worst_k)
best_knn.fit(X_train, y_train)
worst_knn.fit(X_train, y_train)

# Calculate predictions for best and worst k
y_best_pred = best_knn.predict(X_test)
y_worst_pred = worst_knn.predict(X_test)

# Create confusion matrices for best and worst k
best_cm = confusion_matrix(y_test.argmax(axis=1),
y_best_pred.argmax(axis=1))
worst_cm = confusion_matrix(y_test.argmax(axis=1),
y_worst_pred.argmax(axis=1))

# Plot confusion matrices using seaborn
plt.figure(figsize=(15, 6))
plt.subplot(1, 2, 1)
sns.heatmap(best_cm, annot=True, fmt="d", cmap="Blues",
cbar=False)
plt.title(f"Confusion Matrix (Best k={best_k})")

plt.subplot(1, 2, 2)
```

```
sns.heatmap(worst_cm, annot=True, fmt="d", cmap="Reds",
cbar=False)
plt.title(f"Confusion Matrix (Worst k={worst_k})")

plt.show()
```







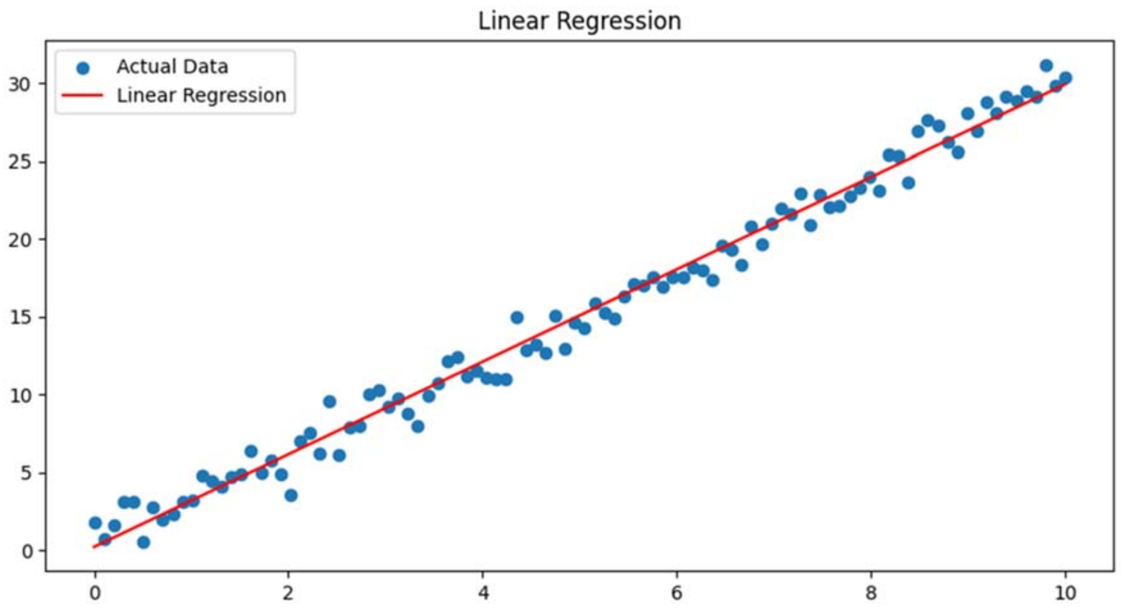
# Experiment 8

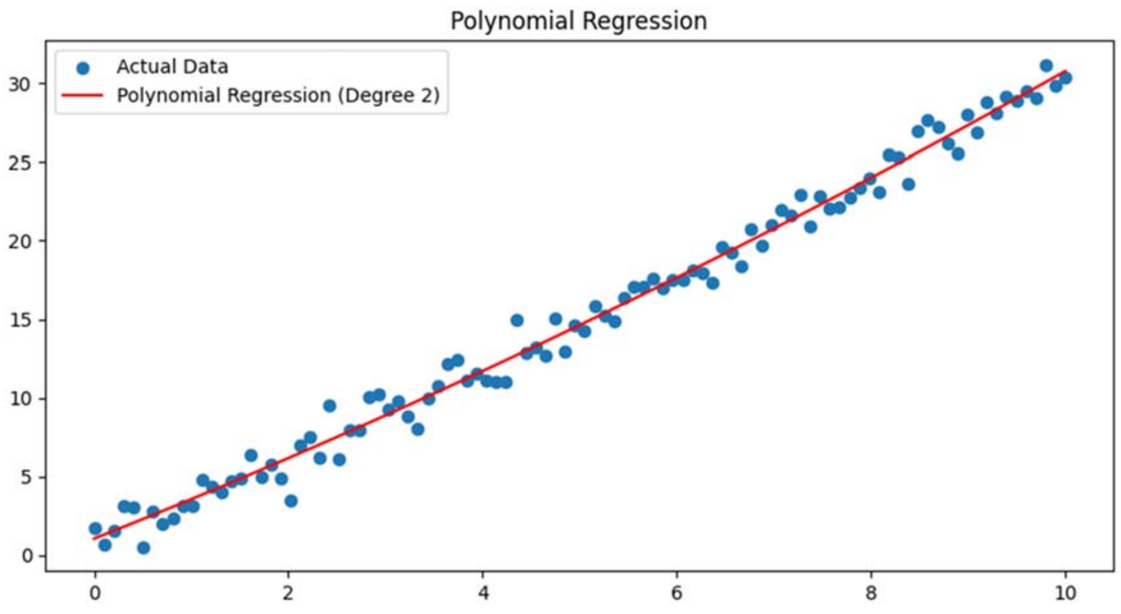
**Aim:** Write a program to implement Linear and Non-Linear regression.

## Code:

```
# Import necessary libraries
import numpy as np
import matplotlib.pyplot as plt
from sklearn.linear_model import LinearRegression
from sklearn.preprocessing import PolynomialFeatures

# Generate random dataset
np.random.seed(0)
X = np.linspace(0, 10, 100)
y = 3*X + np.random.normal(0, 1, 100)  # Linear relationship
with some noise

# Reshape X for sklearn compatibility
X = X.reshape(-1, 1)

# Linear Regression
lr = LinearRegression()
lr.fit(X, y)

# Generate predictions
y_pred = lr.predict(X)

# Plot the results
plt.figure(figsize=(10, 5))
plt.scatter(X, y, label='Actual Data')
plt.plot(X, y_pred, color='red', label='Linear Regression')
plt.title('Linear Regression')
plt.legend()
plt.show()

# Non-linear Regression (Polynomial)
poly_features = PolynomialFeatures(degree=2)  # You can change
degree for non-linearity
X_poly = poly_features.fit_transform(X)

poly_reg = LinearRegression()
poly_reg.fit(X_poly, y)

y_poly_pred = poly_reg.predict(X_poly)

# Plot the results
plt.figure(figsize=(10, 5))
plt.scatter(X, y, label='Actual Data')
plt.plot(X, y_poly_pred, color='red', label='Polynomial
Regression (Degree 2)')
plt.title('Polynomial Regression')
plt.legend()
plt.show()
```

**Output:**



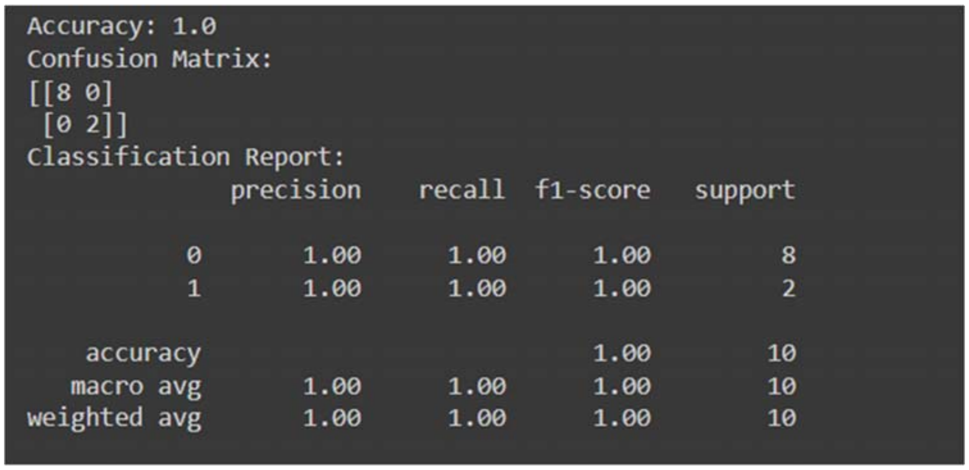


## Experiment 9

**Aim:** Write a program to implement Logistic Regression.

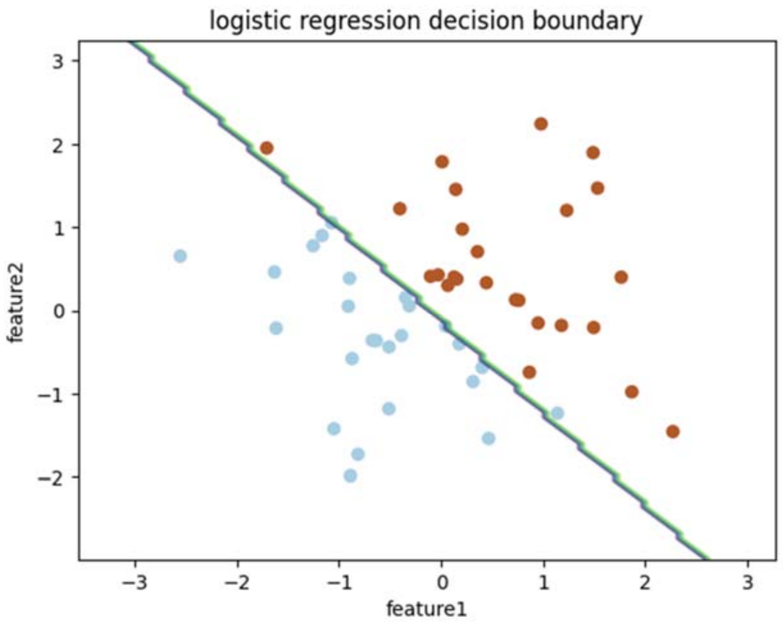
### Code:

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score, confusion_matrix,
classification_report

# Generate some sample data (replace this with your own
dataset)
np.random.seed(0)
X = np.random.randn(50,2)
y = (X[:, 0] + X[:, 1] > 0).astype(int)  # Binary
classification task

# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.2, random_state=42)

# Create a logistic regression model
model = LogisticRegression()

# Fit the model on the training data
model.fit(X_train, y_train)

# Make predictions on the test data
y_pred = model.predict(X_test)

# Evaluate the model
accuracy = accuracy_score(y_test, y_pred)
confusion = confusion_matrix(y_test, y_pred)
classification_rep = classification_report(y_test, y_pred)

print(f"Accuracy: {accuracy}")
print(f"Confusion Matrix:\n{confusion}")
print(f"Classification Report:\n{classification_rep}")
```



```
Accuracy: 1.0
Confusion Matrix:
[[8 0]
 [0 2]]
Classification Report:
              precision    recall  f1-score   support

           0       1.00      1.00      1.00         8
           1       1.00      1.00      1.00         2

    accuracy                           1.00        10
   macro avg       1.00      1.00      1.00        10
weighted avg       1.00      1.00      1.00        10
```

```
xx , yy = np.meshgrid(np.linspace(X[:,0].min() - 1 ,
X[:,0].max() + 1 , 100) ,
                      np.linspace(X[:,1].min() - 1 ,
X[:,1].max() + 1 , 100)
                      )

Z = model.predict(np.c_[xx.ravel(),yy.ravel()])
Z = Z.reshape(xx.shape)

plt.contour(xx,yy,Z,alpha = 0.4)
plt.scatter(X[:,0] , X[:,1], c = y , cmap = plt.cm.Paired)
plt.xlabel("feature1")
plt.ylabel("feature2")
plt.title("logistic regression decision boundary")
plt.show()
```



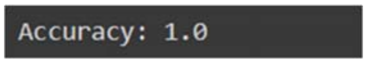
## Experiment 10

**Aim:** Write a program to implement a Decision Tree.

### Code:

```
from sklearn.tree import DecisionTreeClassifier
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score

# Load the Iris dataset (replace this with your dataset)
iris = load_iris()
X = iris.data
y = iris.target

# Split the dataset into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.2, random_state=42)

# Create a Decision Tree Classifier
clf = DecisionTreeClassifier()

# Fit the classifier to the training data
clf.fit(X_train, y_train)

# Make predictions on the test data
y_pred = clf.predict(X_test)

# Calculate accuracy
accuracy = accuracy_score(y_test, y_pred)

print("Accuracy:", accuracy)
```

### Output:



```
Accuracy: 1.0
```

## Experiment 11

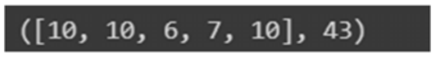
**Aim:** Implement an algorithm to demonstrate the significance of the genetic algorithm.

### Code:

```
import random

# Function to calculate the fitness score
def fitness_score(individual):
# The fitness score is the sum of the individual's elements
return sum(individual)

# Function to select individuals for breeding
def select(population, fitness_scores):
# Select two individuals with probability proportional to
fitness score
return random.choices(population, weights=fitness_scores, k=2)

# Function for crossover
def crossover(ind1, ind2):
# Single-point crossover
point = random.randint(1, len(ind1) - 1)
return ind1[:point] + ind2[point:], ind2[:point] +
ind1[point:]

# Function for mutation
def mutate(individual):
# Randomly change one element in the individual
idx = random.randint(0, len(individual) - 1)
individual[idx] = random.randint(min(individual),
max(individual))

# Genetic algorithm implementation
def genetic_algorithm(population_size, individual_size,
generations):
# Initialize a random population
population = [[random.randint(0, 10) for _ in
range(individual_size)] for _ in range(population_size)]

for _ in range(generations):
# Calculate fitness score for each individual
fitness_scores = list(map(fitness_score, population))

# Create a new population
new_population = []
for _ in range(len(population) // 2):
# Selection
parent1, parent2 = select(population, fitness_scores)
```

```
# Crossover
child1, child2 = crossover(parent1, parent2)
# Mutation
mutate(child1)
mutate(child2)
# Add the new individuals to the new population
new_population.extend([child1, child2])

# Replace the old population with the new population
population = new_population

# Calculate fitness score for each individual in the final
population
final_fitness_scores = list(map(fitness_score, population))
# Find the individual with the highest fitness score
fittest_individual =
population[final_fitness_scores.index(max(final_fitness_scores
))]
return fittest_individual, max(final_fitness_scores)

# Run the genetic algorithm
fittest_individual, max_fitness =
genetic_algorithm(population_size=10, individual_size=5,
generations=10)
fittest_individual, max_fitness
```

**Output:**



```
([10, 10, 6, 7, 10], 43)
```

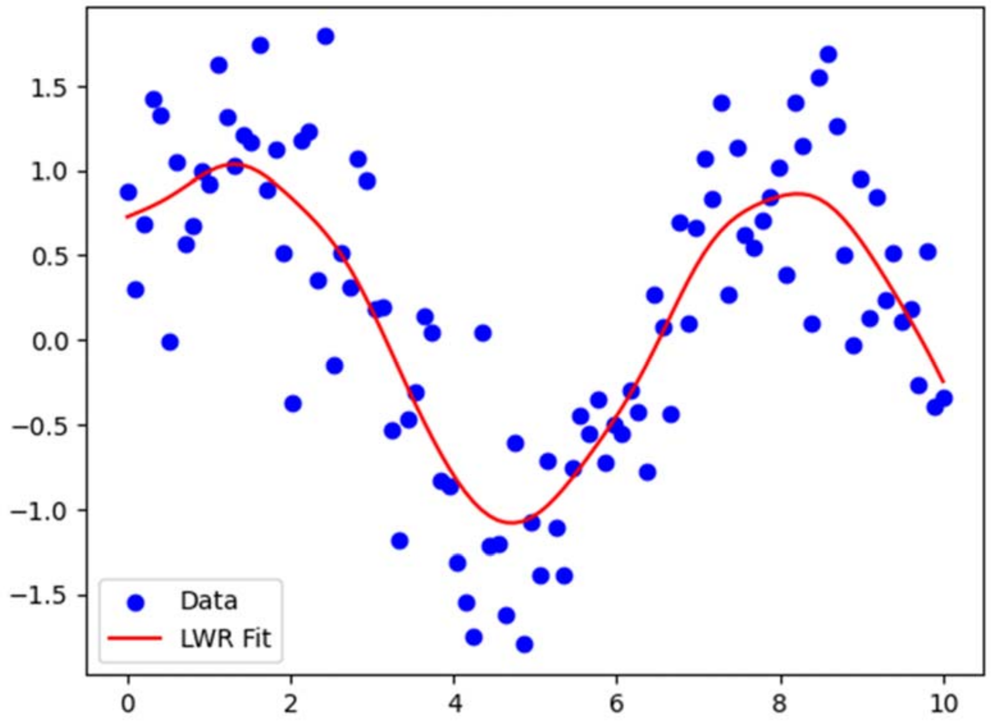
## Experiment 12

**Aim:** Implement the non-parametric Locally Weighted Regression algorithm in order to fit data points. Select the appropriate data set for your experiment and draw graphs.

### Code:

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

# Let's generate some non-linear data
np.random.seed(0)
X = np.linspace(0, 10, 100)
y = np.sin(X) + np.random.normal(0, 0.5, X.shape)

# Let's define the Locally Weighted Regression function
def locally_weighted_regression(x0, X, y, tau):
    # Add a column of ones for the intercept
    X_ = np.hstack((np.ones((len(X), 1)), X.reshape(-1, 1)))
    x0_ = np.array([1, x0])

    # Calculate weights for the data points
    weights = np.exp(-np.sum((X_ - x0_) ** 2, axis=1) / (2 *
tau ** 2))

    # Calculate the weighted least squares solution
    XTX = X_.T.dot(weights[:, None] * X_)
    XTy = X_.T.dot(weights * y)
    theta = np.linalg.solve(XTX, XTy)

    return x0_.dot(theta)

# Let's apply LWR for each point in X
tau = 0.5  # Bandwidth parameter
y_pred = [locally_weighted_regression(x0, X, y, tau) for x0 in
X]

# Plotting the data and the LWR fit
plt.scatter(X, y, color='blue', label='Data')
plt.plot(X, y_pred, color='red', label='LWR Fit')
plt.legend()
plt.show()
```

**Output:**

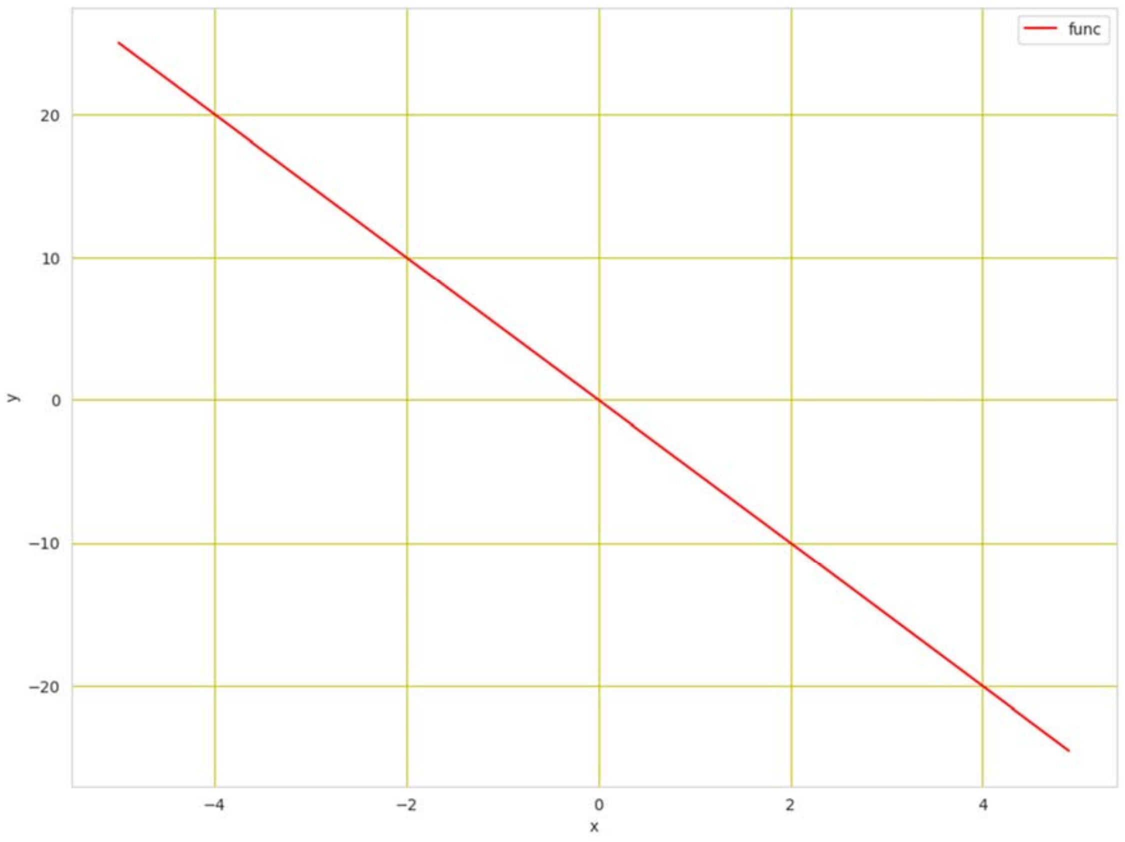


## Experiment 13

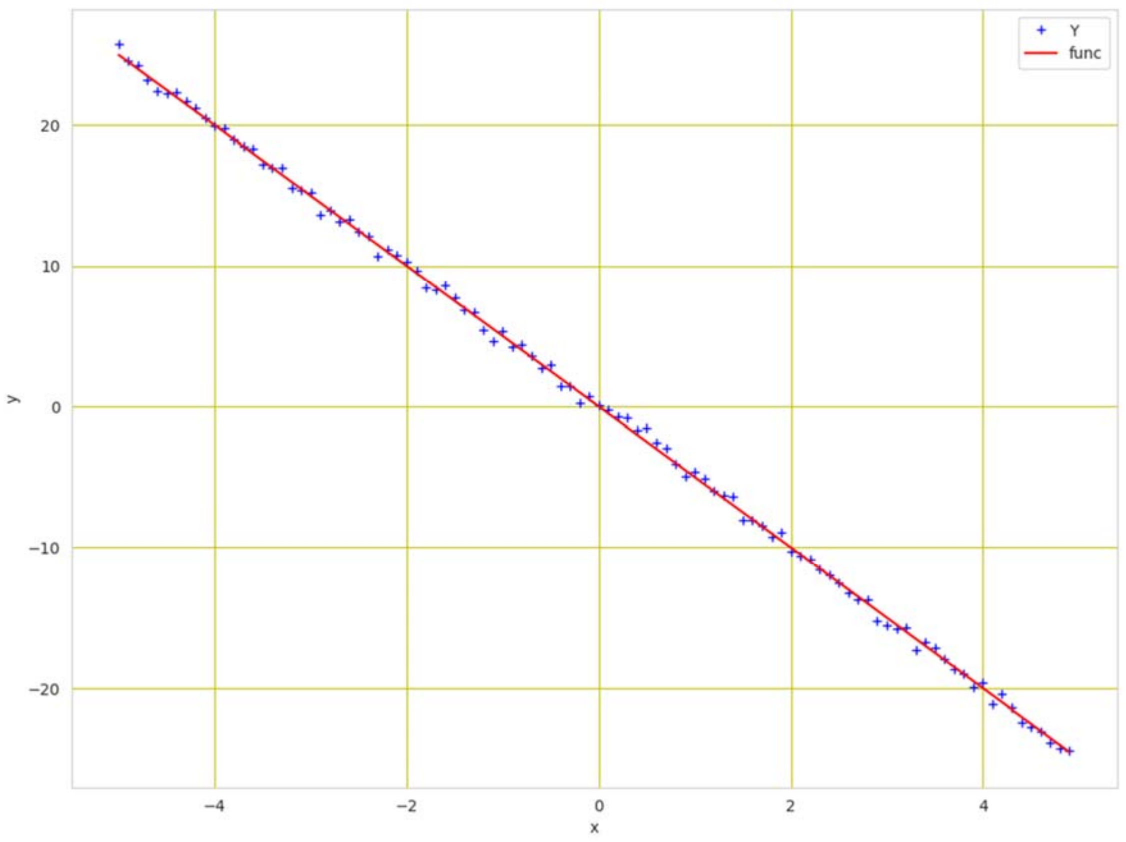
**Aim:** Implement Linear Regression using PyTorch.

**Code:**

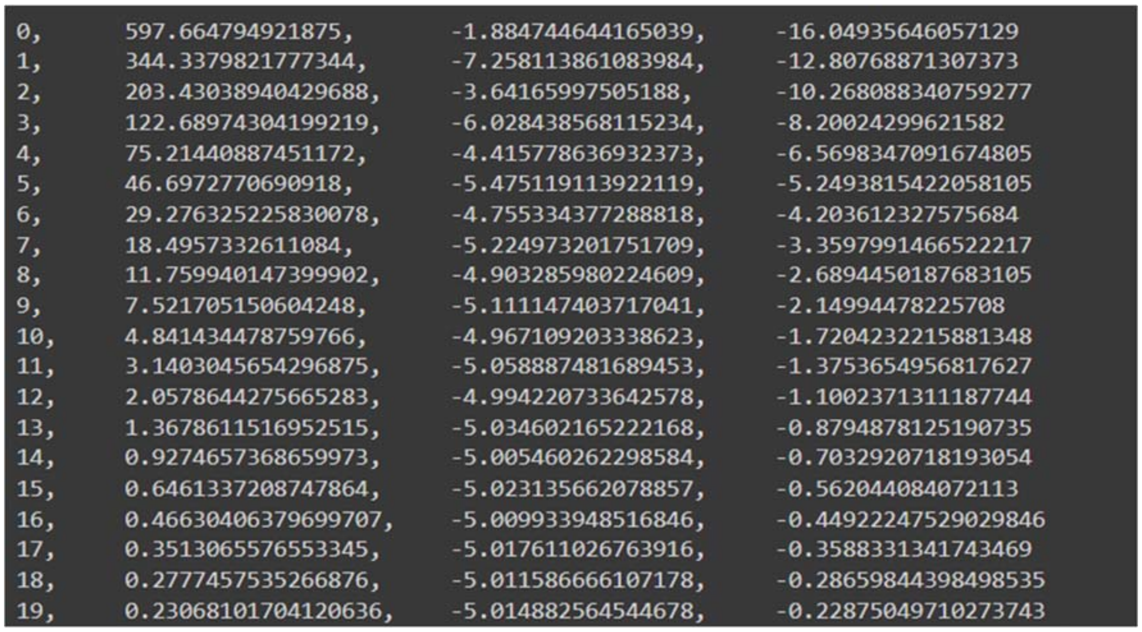
```
import torch
import numpy as np
import matplotlib.pyplot as plt

X = torch.arange(-5, 5, 0.1).view(-1, 1)
func = -5 * X
# Plot the line in red with grids
plt.plot(X.numpy(), func.numpy(), 'r', label='func')
plt.xlabel('x')
plt.ylabel('y')
plt.legend()
plt.grid('True', color='y')
plt.show()
```



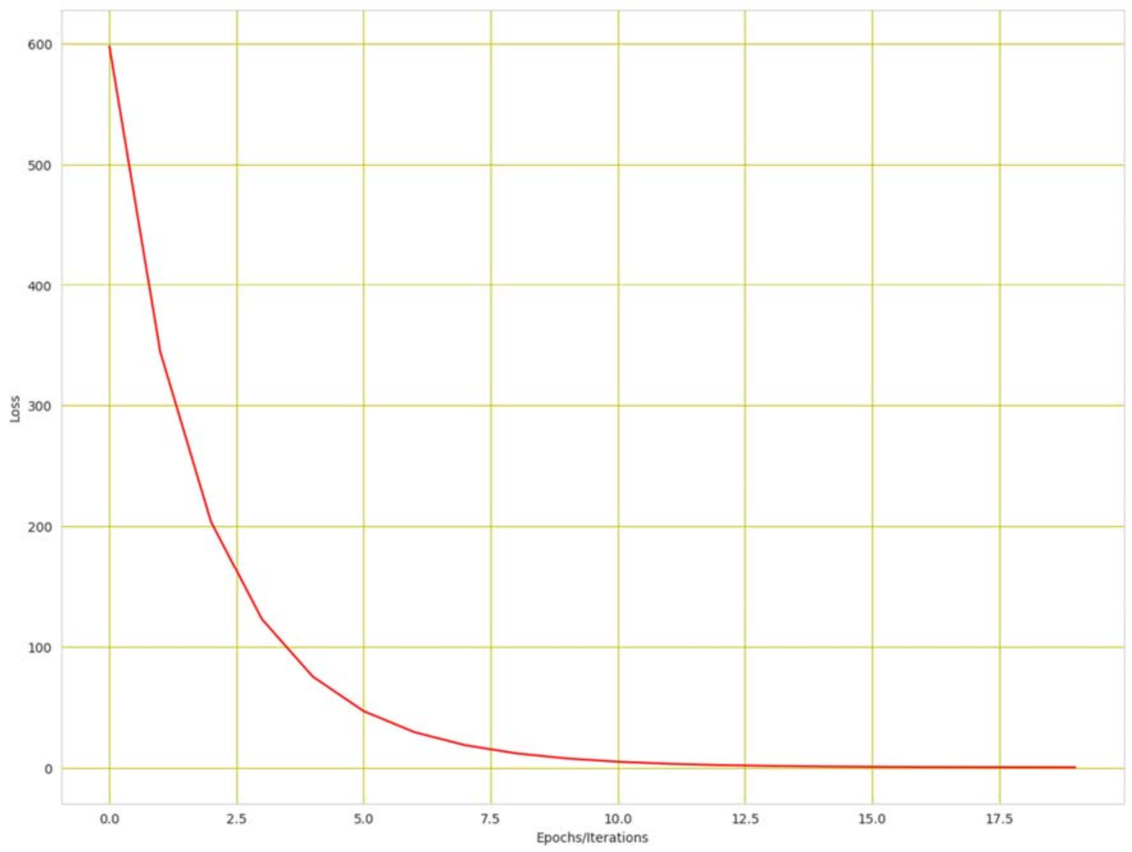
```
Y = func + 0.4 * torch.randn(X.size())

# Plot and visualizing the data points in blue
plt.plot(X.numpy(), Y.numpy(), 'b+', label='Y')
plt.plot(X.numpy(), func.numpy(), 'r', label='func')
plt.xlabel('x')
plt.ylabel('y')
plt.legend()
plt.grid('True', color='y')
plt.show()
```



```
# defining the function for forward pass for prediction
def forward(x):
    return w * x + b

# evaluating data points with Mean Square Error.
def criterion(y_pred, y):
    return torch.mean((y_pred - y) ** 2)

w = torch.tensor(-10.0, requires_grad=True)
b = torch.tensor(-20.0, requires_grad=True)

step_size = 0.1
loss_list = []
iter = 20

for i in range (iter):
    # making predictions with forward pass
    Y_pred = forward(X)
    # calculating the loss between original and predicted data
points
    loss = criterion(Y_pred, Y)
    # storing the calculated loss in a list
    loss_list.append(loss.item())
    # backward pass for computing the gradients of the loss
w.r.t to learnable parameters
    loss.backward()
    # updating the parameters after each iteration
    w.data = w.data - step_size * w.grad.data
    b.data = b.data - step_size * b.grad.data
    # zeroing gradients after each iteration
    w.grad.data.zero_()
    b.grad.data.zero_()
    # priting the values for understanding
    print('{}, \t{}, \t{}, \t{}'.format(i, loss.item(),
w.item(), b.item()))

# Plotting the loss after each iteration
plt.plot(loss_list, 'r')
plt.tight_layout()
plt.grid('True', color='y')
plt.xlabel("Epochs/Iterations")
plt.ylabel("Loss")
plt.show()
```



```
0,      597.664794921875,       -1.884744644165039,     -16.04935646057129
1,      344.3379821777344,      -7.258113861083984,     -12.80768871307373
2,      203.43038940429688,     -3.64165997505188,      -10.268088340759277
3,      122.68974304199219,     -6.028438568115234,     -8.20024299621582
4,      75.21440887451172,      -4.415778636932373,     -6.5698347091674805
5,      46.6972770690918,       -5.475119113922119,     -5.2493815422058105
6,      29.276325225830078,     -4.755334377288818,     -4.203612327575684
7,      18.4957332611084,       -5.224973201751709,     -3.3597991466522217
8,      11.759940147399902,     -4.903285980224609,     -2.6894450187683105
9,      7.521705150604248,      -5.111147403717041,     -2.14994478225708
10,     4.841434478759766,      -4.967109203338623,     -1.7204232215881348
11,     3.1403045654296875,     -5.058887481689453,     -1.3753654956817627
12,     2.0578644275665283,     -4.994220733642578,     -1.1002371311187744
13,     1.3678611516952515,     -5.034602165222168,     -0.8794878125190735
14,     0.9274657368659973,     -5.005460262298584,     -0.7032920718193054
15,     0.6461337208747864,     -5.023135662078857,     -0.562044084072113
16,     0.46630406379699707,    -5.009933948516846,     -0.44922247529029846
17,     0.3513065576553345,     -5.017611026763916,     -0.3588331341743469
18,     0.2777457535266876,     -5.011586666107178,     -0.28659844398498535
19,     0.23068101704120636,    -5.014882564544678,     -0.22875049710273743
```



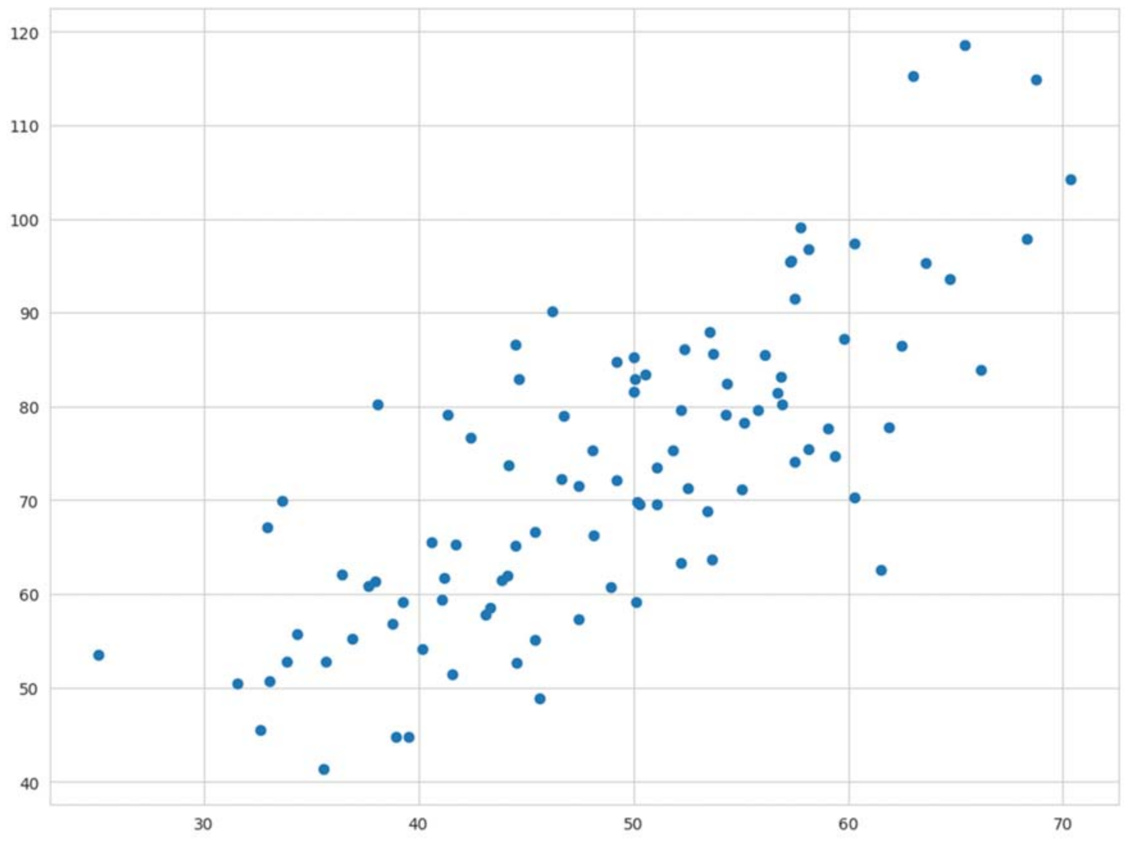
600
500
400
300
200
100
0
Loss
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
Epochs/Iterations

# Experiment 14

**Aim:** Write a program to implement gradient descent in linear regression.

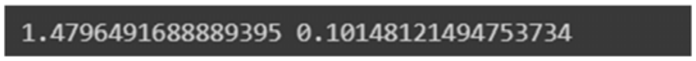
## Code:

```
# Making the imports
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
plt.rcParams['figure.figsize'] = (12.0, 9.0)

# Preprocessing Input data
data =
pd.read_csv('/content/sample_data/linear_reg_pytorch_data.csv'
)
X = data.iloc[:, 0]
Y = data.iloc[:, 1]
plt.scatter(X, Y)
plt.show()
```



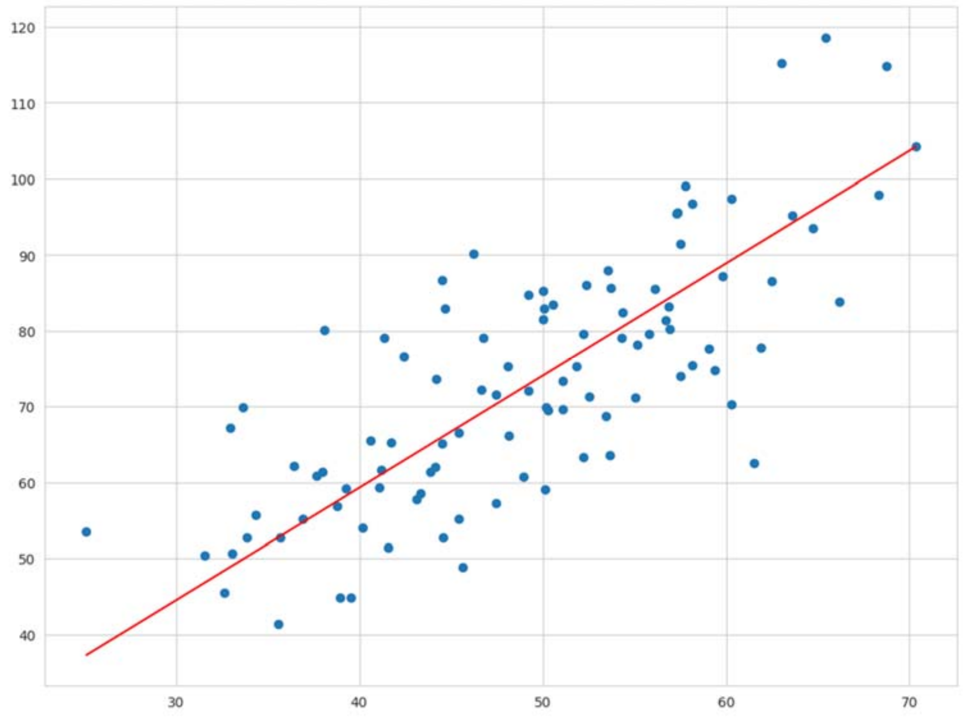
```
# Building the model
m = 0
c = 0
```

```
L = 0.0001  # The learning Rate
epochs = 1000  # The number of iterations to perform gradient
descent

n = float(len(X)) # Number of elements in X

# Performing Gradient Descent
for i in range(epochs):
    Y_pred = m*X + c  # The current predicted value of Y
    D_m = (-2/n) * sum(X * (Y - Y_pred))  # Derivative wrt m
    D_c = (-2/n) * sum(Y - Y_pred)  # Derivative wrt c
    m = m - L * D_m  # Update m
    c = c - L * D_c  # Update c

print (m, c)
```



```
1.4796491688889395 0.10148121494753734
```

```
# Making predictions
Y_pred = m*X + c

plt.scatter(X, Y)
plt.plot([min(X), max(X)], [min(Y_pred), max(Y_pred)],
color='red')  # regression line
plt.show()
```



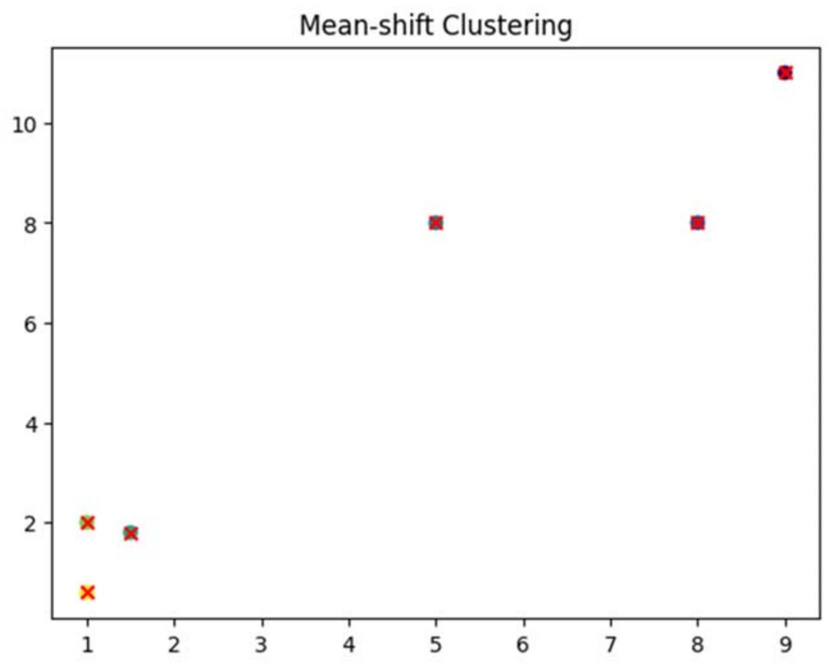
## Experiment 15

**Aim:** Write a program to implement Mean – Shift Clustering.

### Code:

```
from sklearn.cluster import MeanShift

# Sample data
X_meanshift = np.array([[1, 2], [1.5, 1.8], [5, 8], [8, 8],
[1, 0.6], [9, 11]])

# Mean shift clustering
ms = MeanShift()
ms.fit(X_meanshift)

# Cluster centers and labels
cluster_centers = ms.cluster_centers_
labels = ms.labels_

# Plotting the clusters and centers
plt.scatter(X_meanshift[:, 0], X_meanshift[:, 1], c=labels,
cmap='viridis', marker='o')
plt.scatter(cluster_centers[:, 0], cluster_centers[:, 1],
marker='x', color='red')
plt.title('Mean-shift Clustering')
plt.show()
```

### Output:



## Experiment 16

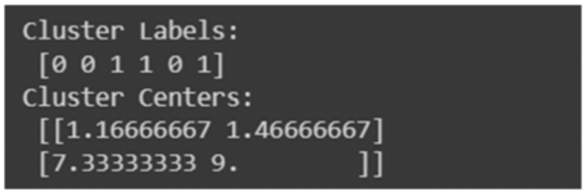
**Aim:** Write a program a program to perform KMeans and KMedoids in Python.

### KMeans

### Code:

```
import numpy as np

def kmeans(data, k, max_iterations=100):
    # Initialize centroids randomly
    centroids = data[np.random.choice(len(data), k,
replace=False)]

    for _ in range(max_iterations):
        # Assign each data point to the nearest centroid
        distances = np.linalg.norm(data[:, np.newaxis] -
centroids, axis=2)
        labels = np.argmin(distances, axis=1)

        # Update centroids
        new_centroids = np.array([data[labels ==
i].mean(axis=0) for i in range(k)])

        # Check for convergence
        if np.all(new_centroids == centroids):
            break

        centroids = new_centroids

    return labels, centroids

# Create a sample dataset (replace this with your data)
data = np.array([[1, 2],
                 [1.5, 1.8],
                 [5, 8],
                 [8, 8],
                 [1, 0.6],
                 [9, 11]])

# Perform K-Means clustering with 2 clusters
k = 2
cluster_labels, cluster_centers = kmeans(data, k)
print("Cluster Labels:\n", cluster_labels)
print("Cluster Centers:\n", cluster_centers)
```

**Output:**



```
Cluster Labels:
 [0 0 1 1 0 1]
Cluster Centers:
 [[1.16666667 1.46666667]
 [7.33333333 9.        ]]
```

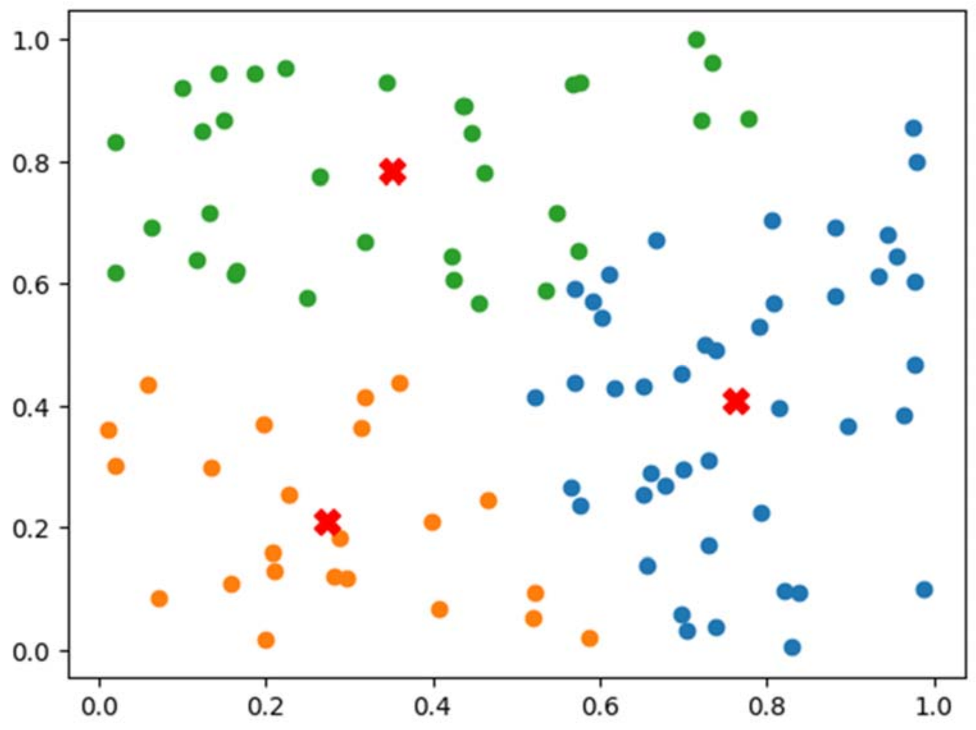
**KMedoids**

**Code:**

```
import numpy as np
from scipy.spatial.distance import cdist
from scipy.spatial import distance_matrix
from scipy.cluster.vq import kmeans, vq
from matplotlib import pyplot

def kmedoids(X, k, max_iters=100):
    # Initialize medoids
    medoids = X[np.random.choice(X.shape[0], k,
replace=False)]
    for _ in range(max_iters):
        # Calculate distance matrix
        D = distance_matrix(X, medoids)
        # Assign each point to the closest medoid
        labels = np.argmin(D, axis=1)
        # Update medoids
        new_medoids = np.array([X[labels == i].mean(axis=0)
for i in range(k)])
        if np.all(new_medoids == medoids):
            break
        medoids = new_medoids
    return labels, medoids

# Generate random data
np.random.seed(0)
X = np.random.rand(100, 2)

# Perform K-medoids clustering
k = 3
labels, medoids = kmedoids(X, k)

# Create scatter plot for samples from each cluster
for i in range(k):
    row_ix = np.where(labels == i)
    pyplot.scatter(X[row_ix, 0], X[row_ix, 1])
```

```
# Show the plot
pyplot.scatter(medoids[:, 0], medoids[:, 1], c='red',
marker='X', s=100)  # Mark medoids with red 'X'
pyplot.show()
```

**Output:**



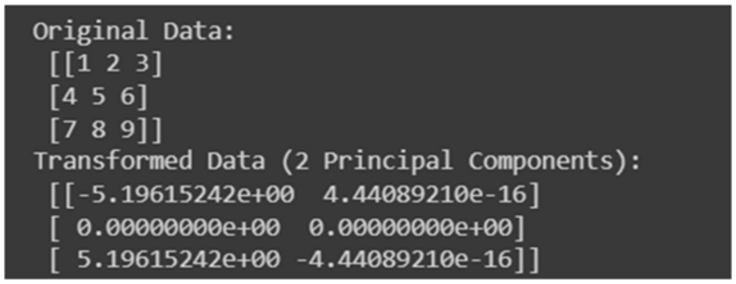
## Experiment 17

**Aim:** Write a program a program to perform PCA in Python.

### Code:

```
import numpy as np

def center_data(data):
    """Center the data by subtracting the mean of each
feature."""
    mean = np.mean(data, axis=0)
    centered_data = data - mean
    return centered_data, mean

def calculate_covariance_matrix(data):
    """Calculate the covariance matrix of the centered
data."""
    num_samples = data.shape[0]
    covariance_matrix = np.dot(data.T, data) / (num_samples -
1)
    return covariance_matrix

def perform_pca(data, num_components=2):
    """Perform Principal Component Analysis (PCA) on the
data."""
    centered_data, mean = center_data(data)
    covariance_matrix =
calculate_covariance_matrix(centered_data)
    eigenvalues, eigenvectors =
np.linalg.eig(covariance_matrix)

    # Sort eigenvalues and eigenvectors
    sorted_indices = np.argsort(eigenvalues)[::-1]
    eigenvalues = eigenvalues[sorted_indices]
    eigenvectors = eigenvectors[:, sorted_indices]

    # Choose the top 'num_components' eigenvectors
    top_eigenvectors = eigenvectors[:, :num_components]

    # Project the data onto the top 'num_components'
eigenvectors
    transformed_data = np.dot(centered_data, top_eigenvectors)

    return transformed_data, eigenvalues, top_eigenvectors,
mean

# Example usage:
data = np.array([[1, 2, 3],
                 [4, 5, 6],
                 [7, 8, 9]])

transformed_data, eigenvalues, top_eigenvectors, mean =
perform_pca(data, num_components=2)
print("Original Data:\n", data)
print("Transformed Data (2 Principal Components):\n",
transformed_data)
```

**Output:**



```
Original Data:
 [[1 2 3]
 [4 5 6]
 [7 8 9]]
Transformed Data (2 Principal Components):
 [[-5.19615242e+00  4.44089210e-16]
 [ 0.00000000e+00  0.00000000e+00]
 [ 5.19615242e+00 -4.44089210e-16]]
```

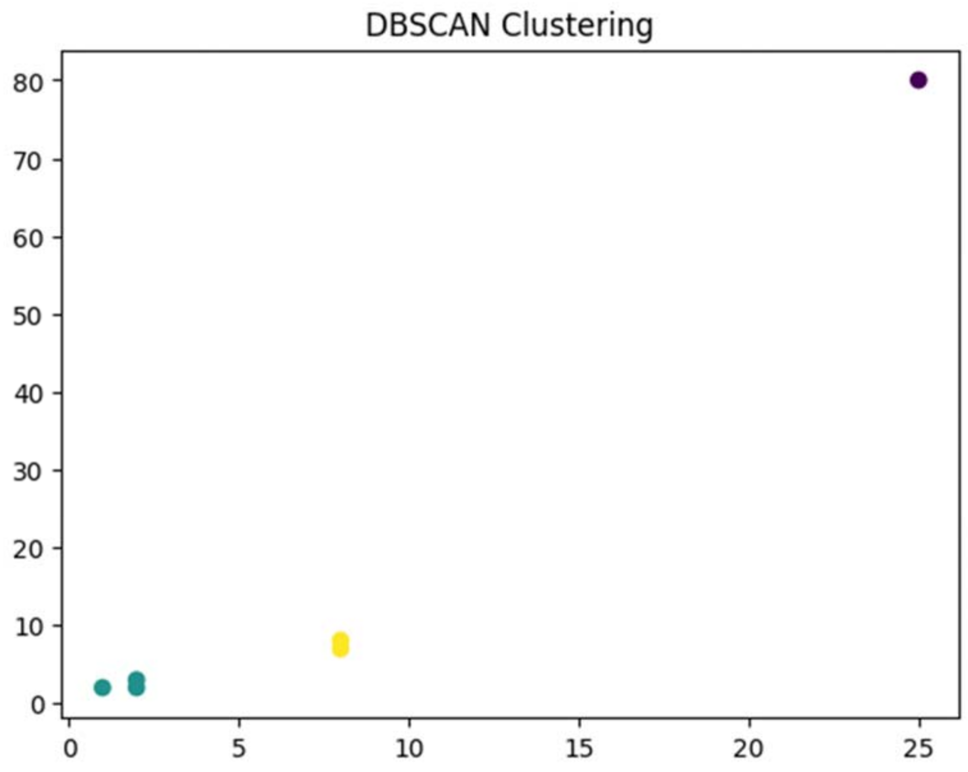
## Experiment 18

**Aim:** Implement the DBSCAN clustering algorithm using Python.

### Code:

```
from sklearn.cluster import DBSCAN

# Sample data
X_dbscan = np.array([[1, 2], [2, 2], [2, 3], [8, 7], [8, 8],
[25, 80]])

# DBSCAN clustering
dbscan = DBSCAN(eps=3, min_samples=2)
dbscan_labels = dbscan.fit_predict(X_dbscan)

# Plotting the cluster assignments
plt.scatter(X_dbscan[:, 0], X_dbscan[:, 1], c=dbscan_labels,
cmap='viridis', marker='o')
plt.title('DBSCAN Clustering')
plt.show()
```

### Output:



## Experiment 19

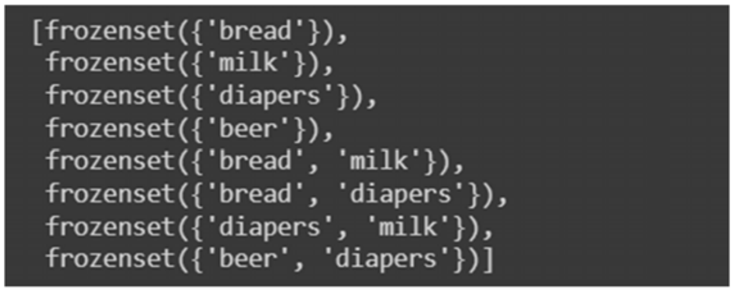
**Aim:** Implement the Apriori algorithm in Python.

### Code:

```
from itertools import combinations

# Function to calculate support
def calculate_support(transactions, itemset):
    return sum(1 for transaction in transactions if
itemset.issubset(transaction)) / len(transactions)

# Function to run apriori algorithm
def apriori(transactions, min_support):
    # Generate initial itemsets of size one
    itemsets = [{item} for transaction in transactions for
item in transaction]
    itemsets = [itemset for itemset in set(frozenset(item) for
item in itemsets) if calculate_support(transactions, itemset)
>= min_support]

    # Generate higher order itemsets until no more are found
    k = 2
    while True:
        new_itemsets = []
        for combo in combinations(itemsets, 2):
            itemset = combo[0].union(combo[1])
            if len(itemset) == k and itemset not in itemsets
and itemset not in new_itemsets:
                if calculate_support(transactions, itemset) >=
min_support:
                    new_itemsets.append(itemset)
        if not new_itemsets:
            break
        itemsets.extend(new_itemsets)
        k += 1

    return itemsets

# Sample transactions data
transactions_apriori = [
    {'bread', 'milk'},
    {'bread', 'diapers', 'beer', 'eggs'},
    {'milk', 'diapers', 'beer', 'cola'},
    {'bread', 'milk', 'diapers', 'beer'},
    {'bread', 'milk', 'diapers', 'cola'},
]

# Run Apriori algorithm
```

```
min_support = 0.6
itemsets = apriori(transactions_apriori, min_support)

itemsets  # Return the frequent itemsets
```

**Output:**



```
[frozenset({'bread'}),
 frozenset({'milk'}),
 frozenset({'diapers'}),
 frozenset({'beer'}),
 frozenset({'bread', 'milk'}),
 frozenset({'bread', 'diapers'}),
 frozenset({'diapers', 'milk'}),
 frozenset({'beer', 'diapers'})]
```