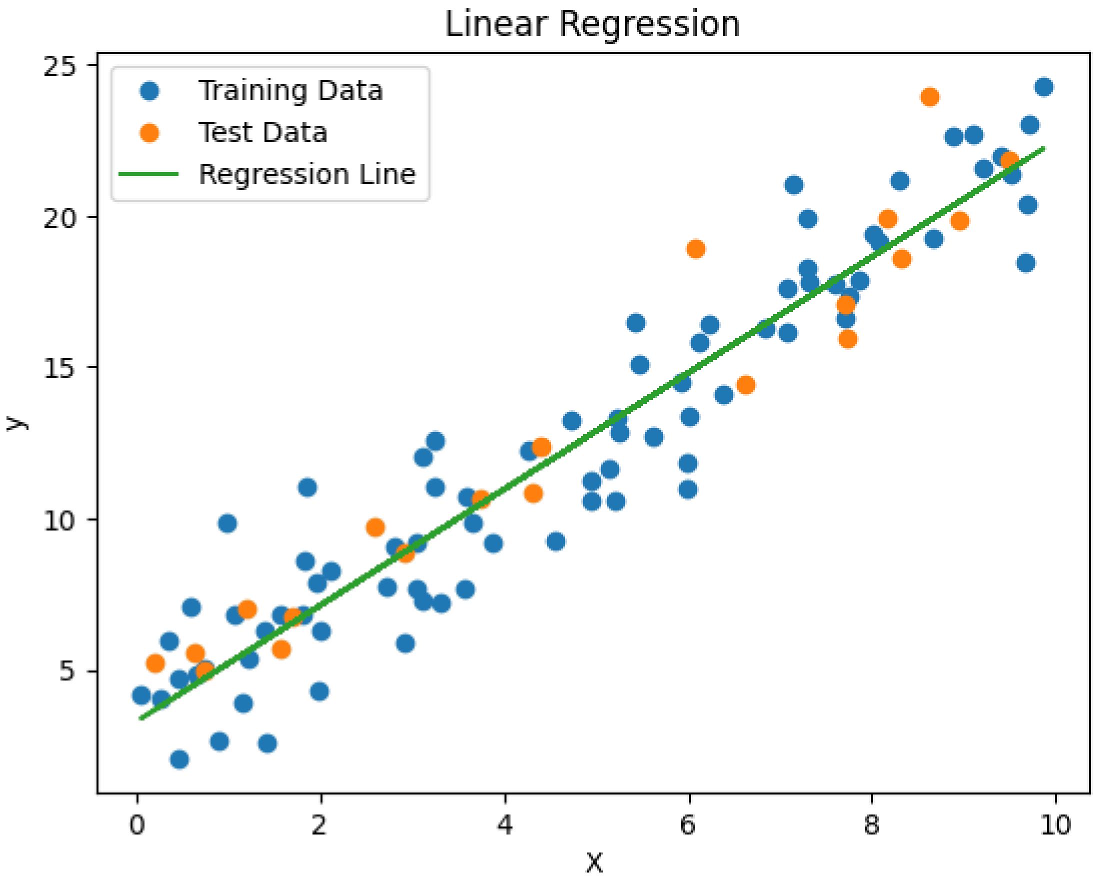
Name:- Kalyani Ravindera Kadam
Roll No.:-23101099
Class:- CSE (B)

Practical No.1 :- Regression Analysis

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt #library used for creating visualizations in
Python, like charts and graphs.

from sklearn.model_selection import train_test_split, KFold
from sklearn.linear_model import LinearRegression

from sklearn.metrics import mean_squared_error

# 1. Generate 2-D Dataset
np.random.seed(42) # For reproducibility (random number initialisation)

N = 100 # Number of data points
X = np.random.rand(N, 1) * 10 # Random X values between 0 and 10 (input
variables)

y = 2 * X + 3 + np.random.randn(N, 1) * 2 # Linear relationship with noise
(Targeted variables)

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

# 3. Linear Regression with Least Squares
reg = LinearRegression().fit(X_train, y_train)

# 4. Calculate and Plot MSE
y_train_pred = reg.predict(X_train)
y_test_pred = reg.predict(X_test)

train_mse = mean_squared_error(y_train, y_train_pred)
test_mse = mean_squared_error(y_test, y_test_pred)

plt.plot(X_train, y_train, 'o', label='Training Data')
plt.plot(X_test, y_test, 'o', label='Test Data')
plt.plot(X, reg.predict(X), label='Regression Line')
plt.legend()
plt.title('Linear Regression')
plt.xlabel('X')
plt.ylabel('y')
plt.show()

print(f"Training MSE: {train_mse:.2f}")
print(f"Test MSE: {test_mse:.2f}")

# ... (Continue with steps for Cross-Validation and Subset Selection)
```

Output:-


Linear Regression
Training Data
Test Data
Regression Line
y
X
25
20
15
10
5
0
2
4
6
8
10

Name:- Kalyani Ravindera Kadam
Roll No.:-23101099
Class:- CSE (B)

Practical No.2 :- Classification using Naïve Bayes Classifier

```
import numpy as np
from sklearn.naive_bayes import GaussianNB
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score

X = np.array([[1, 2], [2, 3], [3, 1], [4, 3], [5, 3], [6, 2]])
y = np.array([0, 0, 1, 1, 1, 0])

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=40)

model = GaussianNB()
model.fit(X_train, y_train)

# GaussianNB
# GaussianNB()

y_pred = model.predict(X_test)
accuracy = accuracy_score(y_test, y_pred)
print("Accuracy:", accuracy)
```

Output:-

Accuracy: 0.0

Name:- Kalyani Ravindera Kadam
Roll No.:-23101099
Class:- CSE (B)

Practical No.3 :- k-Nearest Neighbors (k-NN) Classification

```
import numpy as np
from sklearn.neighbors import KNeighborsClassifier
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score

# Sample data
X = np.array([[3, 5], [4, 7], [6, 3], [7, 4], [8, 2], [9, 6]])
y = np.array([0, 0, 1, 1, 1, 0])

# 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=12)

# Create and fit the model
model = KNeighborsClassifier(n_neighbors=3)  # Set the number of neighbors
model.fit(X_train, y_train)

# Predict the output
y_pred = model.predict(X_test)

# Calculate accuracy
accuracy = accuracy_score(y_test, y_pred)
print("Accuracy:", accuracy)
```

Output:-

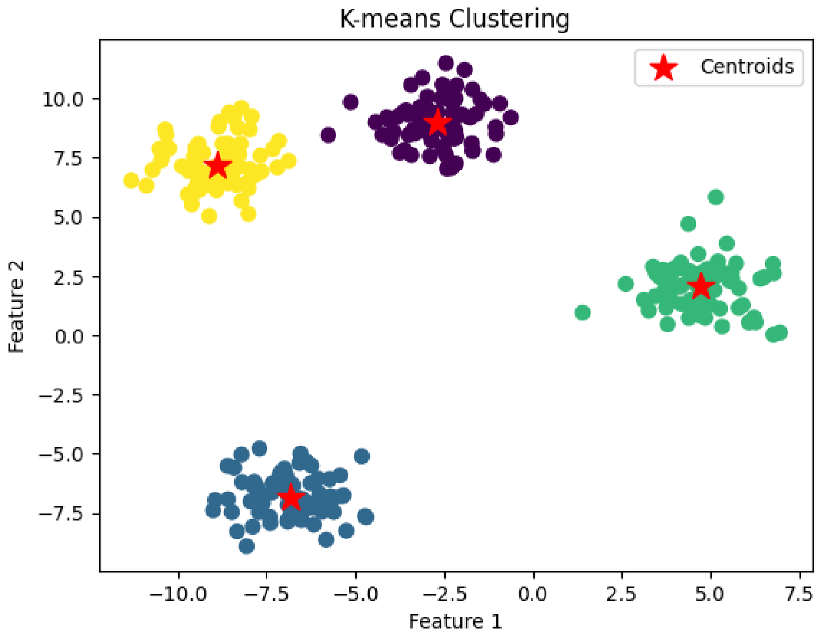
Accuracy: 0.5

Name:- Kalyani Ravindera Kadam
Roll No.:-23101099
Class:- CSE (B)

Practical No.4 :- k-Means Clustering for Classification

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.cluster import KMeans
from sklearn.datasets import make_blobs

# Step 1: Generate sample data using make_blobs (can be replaced with your own data)
X, y = make_blobs(n_samples=300, centers=4, random_state=42)

# Step 2: Perform KMeans clustering
kmeans = KMeans(n_clusters=4, random_state=42)
kmeans.fit(X)

# Step 3: Get cluster centers and labels
cluster_centers = kmeans.cluster_centers_
labels = kmeans.labels_

# Step 4: Visualize the clusters
plt.scatter(X[:, 0], X[:, 1], c=labels, cmap='viridis', s=50)
plt.scatter(cluster_centers[:, 0], cluster_centers[:, 1], s=200, c='red', marker='*',
label='Centroids')
plt.title('K-means Clustering')
plt.xlabel('Feature 1')
plt.ylabel('Feature 2')
plt.legend()
plt.show()
```

Output:-



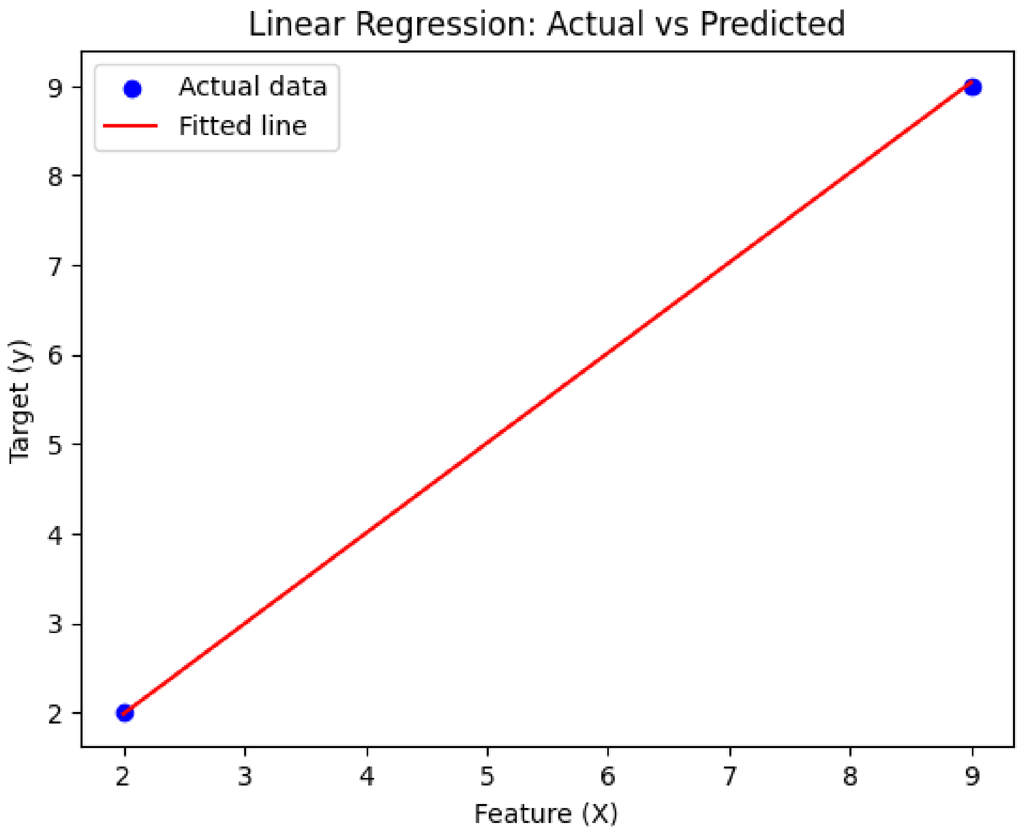
GH raisoni COLLEGE
Engineering and Management
Jalgaon

Name:- Kalyani Ravindera Kadam
Roll No.:-23101099
Class:- CSE (B)

Practical No.5 :- Linear Regression

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.linear_model import LinearRegression
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error, r2_score

# Step 1: Generate synthetic data (example)
# Create a simple dataset for linear regression (1 feature and 1 target)
X = np.array([[1], [2], [3], [4], [5], [6], [7], [8], [9], [10]])  # feature (independent variable)
y = np.array([1.1, 2.0, 2.9, 4.0, 5.0, 5.9, 7.0, 8.1, 9.0, 10.1]) # target (dependent variable)

# Step 2: Split the data into training and testing sets (80% training, 20% testing)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Step 3: Create a Linear Regression model
model = LinearRegression()

# Step 4: Train the model using the training data
model.fit(X_train, y_train)

# Step 5: Make predictions on the test data
y_pred = model.predict(X_test)

# Step 6: Evaluate the model's performance
mse = mean_squared_error(y_test, y_pred)
r2 = r2_score(y_test, y_pred)

# Output model evaluation results
print(f"Mean Squared Error (MSE): {mse}")
print(f"R-squared (R2): {r2}")

# Step 7: Visualize the results
plt.scatter(X_test, y_test, color='blue', label='Actual data')  # Actual data points
plt.plot(X_test, y_pred, color='red', label='Fitted line')    # Predicted values

plt.title('Linear Regression: Actual vs Predicted')
plt.xlabel('Feature (X)')
plt.ylabel('Target (y)')
plt.legend()
plt.show()

# Step 8: Display the learned model parameters (slope and intercept)
print(f"Model Slope (Coefficient): {model.coef_[0]}")
print(f"Model Intercept: {model.intercept_}")
```

Output:-




```
Mean Squared Error (MSE): 0.00089365338882227551
R-squared (R2): 0.9999270487029532
```

Name:- Kalyani Ravindera Kadam
Roll No.:-23101099
Class:- CSE (B)

Practical No.6 :- Classification using Naïve Bayes Theorem

```
import math
from collections import defaultdict

class NaiveBayesClassifier:
  def __init__(self):
    self.class_probs = defaultdict(float)  # Stores log P(class)
    self.word_probs = defaultdict(lambda: defaultdict(float))  # Stores log P(word|class)
    self.class_counts = defaultdict(int)  # Stores number of documents per class
    self.word_counts = defaultdict(lambda: defaultdict(int))  # Stores word frequency per class
    self.total_docs = 0  # Total number of documents
    self.total_words_per_class = defaultdict(int)  # Stores total words in each class

  def train(self, documents, labels):
    """Trains the Naive Bayes model with given documents and labels."""
    self.total_docs = len(documents)

    for doc, label in zip(documents, labels):
      words = doc.lower().split()  # Tokenization
      self.class_counts[label] += 1
      for word in words:
        self.word_counts[label][word] += 1
        self.total_words_per_class[label] += 1  # Count words per class

    # Calculate log P(class)
    for label in self.class_counts:
      self.class_probs[label] = math.log(self.class_counts[label] / self.total_docs)

    # Calculate log P(word|class) with Laplace smoothing
    for label in self.word_counts:
      vocab_size = len(self.word_counts[label])  # Unique words in class
      for word in self.word_counts[label]:
        self.word_probs[label][word] = math.log(
          (self.word_counts[label][word] + 1) /
          (self.total_words_per_class[label] + vocab_size)  # Normalize by class words
        )

  def predict(self, text):
    """Predicts the class of a given text input."""
    words = text.lower().split()
    class_scores = {}

    for label in self.class_counts:
      score = self.class_probs[label]  # Start with log P(class)
      vocab_size = len(self.word_counts[label])  # Get vocabulary size of class

      for word in words:
        if word in self.word_probs[label]:
```

```
                score += self.word_probs[label][word]  # Use known probability
            else:
                # Apply Laplace smoothing for unknown words
                score += math.log(1 / (self.total_words_per_class[label] + vocab_size))

        class_scores[label] = score

    return max(class_scores, key=class_scores.get)

# Example usage
documents = [
    "I love this song",
    "This song is amazing",
    "I hate this movie",
    "This movie is terrible"
]
labels = ["positive", "positive", "negative", "negative"]

nb_classifier = NaiveBayesClassifier()
nb_classifier.train(documents, labels)

test_text = "I hate this movie"
predicted_class = nb_classifier.predict(test_text)
print(f"Predicted class for '{test_text}': {predicted_class}")
```

Output:-

Predicted class for 'I hate this movie': negative

Name:- Kalyani Ravindera Kadam
Roll No.:-23101099
Class:- CSE (B)

Practical No.7 :- Genetic Algorithm

```
import random

# Function to maximize (fitness function)
def fitness(x):
    return x**2  # Maximizing x^2

# Selection process using tournament selection
def selection(population):
    tournament = random.sample(population, 3)  # Select 3 random individuals
    return max(tournament, key=lambda ind: fitness(int("".join(map(str, ind)), 2)))  # Best of 3

# Crossover operation (single-point crossover)
def crossover(parent1, parent2):
    crossover_point = random.randint(1, len(parent1) - 1)
    offspring1 = parent1[:crossover_point] + parent2[crossover_point:]
    offspring2 = parent2[:crossover_point] + parent1[crossover_point:]
    return offspring1, offspring2

# Mutation operation (bit flip mutation)
def mutation(individual, mutation_rate=0.1):
    new_individual = individual[:]  # Copy to avoid modifying the original
    for i in range(len(new_individual)):
        if random.random() < mutation_rate:  # Apply mutation based on probability
            new_individual[i] = 1 - new_individual[i]  # Flip bit
    return new_individual

# Main Genetic Algorithm
def genetic_algorithm(pop_size=10, generations=20, mutation_rate=0.1):
    # Initialize population (8-bit binary representation)
    population = [[random.choice([0, 1]) for _ in range(8)] for _ in range(pop_size)]

    for gen in range(generations):
        # Sort population by fitness (descending order)
        population = sorted(population, key=lambda x: fitness(int("".join(map(str, x)), 2)),
reverse=True)

        best_value = int("".join(map(str, population[0])), 2)
        print(f"Generation {gen}: Best value = {best_value}, Fitness = {fitness(best_value)}")

        # Preserve top 20% of the population (elitism)
        elite_size = max(1, pop_size // 5)
        new_population = population[:elite_size]

        # Create the next generation
        while len(new_population) < pop_size:
            parent1 = selection(population)
            parent2 = selection(population)
```

```
            offspring1, offspring2 = crossover(parent1, parent2)
            offspring1 = mutation(offspring1, mutation_rate)
            offspring2 = mutation(offspring2, mutation_rate)
            new_population.extend([offspring1, offspring2])

        # Ensure population size remains constant
        population = new_population[:pop_size]

    # Return the best individual after all generations
    best_individual = population[0]
    return best_individual

# Run the Genetic Algorithm
best_individual = genetic_algorithm()
best_value = int("".join(map(str, best_individual)), 2)
print(f"Best individual: {best_individual}, Value: {best_value}, Fitness: {fitness(best_value)}")
```

Output:-

Generation 0: Best value = 198, Fitness = 39204
Generation 1: Best value = 227, Fitness = 51529
Generation 2: Best value = 244, Fitness = 59536
...
Generation 18: Best value = 253, Fitness = 64009
Generation 19: Best value = 255, Fitness = 65025
Best individual: [1, 1, 1, 1, 1, 1, 1, 1], Value: 255, Fitness: 65025