SRM Institute of Science and Technology

Department of Computer Applications

Delhi – Meerut Road, Sikri Kalan, Ghaziabad, Uttar Pradesh – 201204

Circular – 2020-21

MCA GAI 3rd semester

Adaptive AI in Data Analytics and Predictive Modeling (PGI20D13J)

Lab Manual

Lab 1: Develop a Personalized Content Delivery System

Aim: To design and implement a basic personalized content delivery system that recommends content to users based on their preferences and past interactions.

Procedure:

1. **Data Collection:** Simulate or collect user interaction data (e.g., content viewed, ratings, time spent).
2. **User Profiling:** Create user profiles based on collected data, identifying preferences and interests.
3. **Content Representation:** Represent content items with relevant features (e.g., categories, keywords).
4. **Recommendation Algorithm:** Implement a simple recommendation algorithm (e.g., collaborative filtering, content-based filtering, or a hybrid approach).
5. **Content Delivery:** Develop a mechanism to deliver personalized content recommendations to users.
6. **Evaluation (Optional):** Define metrics to evaluate the effectiveness of the recommendation system.

Source Code (Conceptual Python):

```python
# personalized_content_delivery.py

import pandas as pd
from sklearn.metrics.pairwise import cosine_similarity
from sklearn.feature_extraction.text import TfidfVectorizer

# --- Simulate Data ---
# User interaction data (user_id, content_id, interaction_type/rating)
user_interactions = pd.DataFrame({
    'user_id': [1, 1, 2, 2, 3, 3, 1, 2],
    'content_id': ['A', 'B', 'A', 'C', 'B', 'D', 'C', 'D'],
    'rating': [5, 3, 4, 5, 2, 4, 4, 3]
})

# Content features (content_id, description/tags)
content_features = pd.DataFrame({
    'content_id': ['A', 'B', 'C', 'D'],
    'description': [
        'Science fiction, space opera, adventure',
        'Romantic comedy, drama, lighthearted',
        'Action, thriller, suspense',
        'Documentary, nature, education'
    ]
```

```
})

# --- Content-Based Filtering Example ---
def get_content_recommendations(user_id, num_recommendations=2):
    # Merge data
    merged_data = pd.merge(user_interactions, content_features, on='content_id')

    # Get content liked by the user
    user_liked_content = merged_data[merged_data['user_id'] ==
user_id]['content_id'].tolist()
    user_liked_descriptions = merged_data[merged_data['user_id'] ==
user_id]['description'].tolist()

    if not user_liked_descriptions:
        print(f"No liked content found for user {user_id}.")
        return pd.DataFrame()

    # Create TF-IDF vectors for content descriptions
    tfidf_vectorizer = TfidfVectorizer(stop_words='english')
    content_tfidf_matrix =
tfidf_vectorizer.fit_transform(content_features['description'])

    # Calculate similarity between user's liked content and all content
    user_profile_vector =
tfidf_vectorizer.transform(user_liked_descriptions).mean(axis=0)
    cosine_similarities = cosine_similarity(user_profile_vector,
content_tfidf_matrix).flatten()

    # Create a series of content_id and their similarities
    content_similarities = pd.Series(cosine_similarities,
index=content_features['content_id'])

    # Sort by similarity and exclude already liked content
    recommended_content = content_similarities.sort_values(ascending=False)
    recommended_content =
recommended_content[~recommended_content.index.isin(user_liked_content)]

    return recommended_content.head(num_recommendations)

if __name__ == "__main__":
    print("--- Personalized Content Delivery System ---")

    # Example for user 1
    user_id_to_recommend = 1
    recommendations = get_content_recommendations(user_id_to_recommend)
    if not recommendations.empty:
        print(f"\nRecommendations for User {user_id_to_recommend}:")
        print(recommendations)

    # Example for user 4 (new user)
    user_id_to_recommend_new = 4
    recommendations_new = get_content_recommendations(user_id_to_recommend_new)
    if not recommendations_new.empty:
        print(f"\nRecommendations for User {user_id_to_recommend_new}:")
        print(recommendations_new)
```

Input:

- A dataset of user interactions (e.g., `user_id`, `content_id`, `rating`).
- A dataset of content features (e.g., `content_id`, `description` or `tags`).
- A specific `user_id` for whom recommendations are to be generated.

Expected Output: A list of recommended `content_id`s for the specified user, ordered by relevance or predicted preference.

```
--- Personalized Content Delivery System ---

Recommendations for User 1:
content_id
D    0.347070
Name: 0, dtype: float64
No liked content found for user 4.
```

Lab 2: Develop Intelligent Tutoring Systems

Aim: To explore the components and functionalities of Intelligent Tutoring Systems (ITS) and simulate a basic adaptive learning interaction.

Procedure:

1. **Understand ITS Components:** Research and identify key components of an ITS (e.g., student model, domain model, pedagogical model, expert model).
2. **Simulate Student Model:** Create a simple student model that tracks a student's knowledge level or mastery of specific concepts.
3. **Adaptive Questioning:** Implement a basic logic for adaptive questioning based on the student's performance (e.g., if a student answers incorrectly, provide an easier question or a hint).
4. **Feedback Mechanism:** Develop a simple feedback mechanism to guide the student.
5. **Learning Pathway (Basic):** Outline how an ITS might adjust the learning path dynamically.

Source Code (Conceptual Python):

```
# intelligent_tutoring_system.py

class IntelligentTutoringSystem:
    def __init__(self, topics):
        self.topics = topics
        self.student_knowledge = {topic: 0 for topic in topics} # 0: beginner, 
1: intermediate, 2: advanced
        self.current_topic_index = 0

    def ask_question(self):
        current_topic = self.topics[self.current_topic_index]
        knowledge_level = self.student_knowledge[current_topic]

        if knowledge_level == 0:
            question_level = "basic"
        elif knowledge_level == 1:
            question_level = "intermediate"
        else:
            question_level = "advanced"

        print(f"\nTopic: {current_topic} (Level: {question_level})")
        question = input(f"Question for {current_topic} ({question_level}): What 
is the capital of France? (Paris/London/Rome) ")
        return question.strip().lower() == "paris" # Simulate correct answer

    def provide_feedback(self, correct):
        current_topic = self.topics[self.current_topic_index]
        if correct:
            print("Correct! Good job.")
            # Advance knowledge level, but cap at advanced
            self.student_knowledge[current_topic] = 
min(self.student_knowledge[current_topic] + 1, 2)
        else:
            print("Incorrect. Let's review this concept or try an easier 
question.")
            # Decrease knowledge level, but cap at beginner
            self.student_knowledge[current_topic] = 
max(self.student_knowledge[current_topic] - 1, 0)
            print(f"Hint: The Eiffel Tower is in this city.")
```

```
    def advance_topic(self):
        self.current_topic_index = (self.current_topic_index + 1) %
len(self.topics)
        print(f"\nMoving to next topic:
{self.topics[self.current_topic_index]}")

if __name__ == "__main__":
    topics_list = ["Geography", "History", "Science"]
    its = IntelligentTutoringSystem(topics_list)

    print("--- Intelligent Tutoring System Simulation ---")
    for _ in range(5): # Simulate 5 interactions
        its.ask_question()
        answer = input("Your answer: ") # User provides input
        is_correct = (answer.lower() == "paris") # Assuming the question is
always about Paris for simplicity
        its.provide_feedback(is_correct)

        # Simple adaptive logic: if correct twice in a row, advance topic
        if is_correct and
its.student_knowledge[its.topics[its.current_topic_index]] > 0:
            its.advance_topic()

        print(f"Current knowledge levels: {its.student_knowledge}")
```

Input:

- User's answers to questions.
- Predefined topics and questions.

Expected Output:

- Adaptive questions based on simulated student knowledge.
- Feedback (correct/incorrect, hints).
- Updates to the student's knowledge model.
- Potential changes in the learning path (e.g., advancing to a new topic).

```
--- Intelligent Tutoring System Simulation ---

Topic: Geography (Level: basic)
Question for Geography (basic): What is the capital of France?
(Paris/London/Rome) Your answer: paris
Correct! Good job.

Moving to next topic: History
Current knowledge levels: {'Geography': 1, 'History': 0, 'Science': 0}

Topic: History (Level: basic)
Question for History (basic): What is the capital of France? (Paris/London/Rome)
Your answer: london
Incorrect. Let's review this concept or try an easier question.
Hint: The Eiffel Tower is in this city.
Current knowledge levels: {'Geography': 1, 'History': 0, 'Science': 0}

Topic: History (Level: basic)
Question for History (basic): What is the capital of France? (Paris/London/Rome)
Your answer: paris
Correct! Good job.

Moving to next topic: Science
Current knowledge levels: {'Geography': 1, 'History': 1, 'Science': 0}
```

```
Topic: Science (Level: basic)
Question for Science (basic): What is the capital of France? (Paris/London/Rome)
Your answer: paris
Correct! Good job.

Moving to next topic: Geography
Current knowledge levels: {'Geography': 1, 'History': 1, 'Science': 1}

Topic: Geography (Level: intermediate)
Question for Geography (intermediate): What is the capital of France?
(Paris/London/Rome) Your answer: paris
Correct! Good job.

Moving to next topic: History
Current knowledge levels: {'Geography': 2, 'History': 1, 'Science': 1}
```

Lab 3: Develop Dynamic Learning Pathways

Aim: To implement a system that dynamically adjusts learning pathways for users based on their performance and learning style.

Procedure:

1. **Define Learning Modules:** Break down content into granular learning modules or topics.
2. **Assess User Performance:** Track user performance on quizzes or exercises within modules.
3. **Identify Learning Styles (Simplified):** For simplicity, assume or infer a learning style (e.g., visual, auditory, kinesthetic) or just focus on performance-based adaptation.
4. **Adaptive Logic:** Develop rules or an algorithm to determine the next module or content type based on performance. For example, if a user struggles with a concept, recommend prerequisite modules or alternative explanations.
5. **Pathway Visualization (Optional):** If possible, visualize the dynamic pathway.

Source Code (Conceptual Python):

```
# dynamic_learning_pathways.py

class LearningPathwayManager:
    def __init__(self, modules, prerequisites):
        self.modules = modules # List of modules, e.g., ["Intro", "Algebra",
"Geometry", "Calculus"]
        self.prerequisites = prerequisites # Dictionary: {module: [prereq1,
prereq2]}
        self.user_progress = {} # {user_id: {module:
"completed"|"in_progress"|"not_started"}}
        self.user_scores = {} # {user_id: {module: score}}

    def start_learning(self, user_id):
        if user_id not in self.user_progress:
            self.user_progress[user_id] = {module: "not_started" for module in
self.modules}
            self.user_scores[user_id] = {module: 0 for module in self.modules}
            print(f"User {user_id} started learning.")
            return True
        else:
            print(f"User {user_id} already exists.")
            return False

    def get_next_module(self, user_id):
        if user_id not in self.user_progress:
            print(f"User {user_id} not initialized. Call start_learning first.")
            return None

        # Find modules not started or in progress
        available_modules = [
            module for module, status in self.user_progress[user_id].items()
            if status == "not_started" or status == "in_progress"
        ]

        for module in self.modules: # Iterate in defined order for initial path
            if module in available_modules:
                # Check prerequisites
                if module in self.prerequisites:
                    all_prereqs_met = True
                    for prereq in self.prerequisites[module]:
                        if self.user_progress[user_id].get(prereq) !=
"completed":
```

```
                            all_prereqs_met = False
                            break
                    if all_prereqs_met:
                        return module
                else: # No prerequisites
                    return module
        return None # No more modules available

    def complete_module(self, user_id, module_name, score):
        if user_id not in self.user_progress or module_name not in self.modules:
            print("Invalid user or module.")
            return

        self.user_progress[user_id][module_name] = "completed"
        self.user_scores[user_id][module_name] = score
        print(f"User {user_id} completed '{module_name}' with score {score}.")

    def adapt_path(self, user_id):
        # Simple adaptation: if score is low, recommend revisiting or a remedial
module
        for module, score in self.user_scores[user_id].items():
            if self.user_progress[user_id][module] == "completed" and score <
60:
                print(f"User {user_id} scored low on '{module}' ({score}%).
Recommending a review of this module.")
                # In a real system, you'd suggest specific remedial content or a
simpler version.
                return module # Suggest revisiting this module
        return None

if __name__ == "__main__":
    modules_list = ["Module A: Introduction", "Module B: Basic Concepts",
"Module C: Advanced Topics", "Module D: Application"]
    prereqs = {
        "Module B: Basic Concepts": ["Module A: Introduction"],
        "Module C: Advanced Topics": ["Module B: Basic Concepts"],
        "Module D: Application": ["Module C: Advanced Topics"]
    }

    lpm = LearningPathwayManager(modules_list, prereqs)
    user1 = "student_alice"
    lpm.start_learning(user1)

    print("\n--- Simulating Learning Process ---")

    # Alice completes Module A
    current_module = lpm.get_next_module(user1)
    print(f"User {user1} is currently on: {current_module}")
    lpm.complete_module(user1, current_module, 85)

    # Alice moves to Module B
    current_module = lpm.get_next_module(user1)
    print(f"User {user1} is currently on: {current_module}")
    lpm.complete_module(user1, current_module, 50) # Low score

    # Check for adaptation
    remedial_suggestion = lpm.adapt_path(user1)
    if remedial_suggestion:
        print(f"Adaptive suggestion for {user1}: Consider revisiting
'{remedial_suggestion}'.")
        # In a real system, you'd insert this module back into the queue or
offer alternative content.

    # Alice tries to move to Module C, but might be redirected or need to re-do
B
    current_module = lpm.get_next_module(user1)
    print(f"User {user1} is currently on: {current_module}")
```

```
    lpm.complete_module(user1, current_module, 75)

    # Final state
    print(f"\nFinal progress for {user1}: {lpm.user_progress[user1]}")
    print(f"Final scores for {user1}: {lpm.user_scores[user1]}")
```

Input:

- A defined set of learning modules and their prerequisites.
- User performance scores for completed modules.
- (Optional) User learning style preferences.

Expected Output:

- A dynamically generated sequence of modules for a user.
- Suggestions for remedial content or alternative pathways based on performance.

```
User student_alice started learning.

--- Simulating Learning Process ---
User student_alice is currently on: Module A: Introduction
User student_alice completed 'Module A: Introduction' with score 85.
User student_alice is currently on: Module B: Basic Concepts
User student_alice completed 'Module B: Basic Concepts' with score 50.
User student_alice scored low on 'Module B: Basic Concepts' (50%). Recommending
a review of this module.
Adaptive suggestion for student_alice: Consider revisiting 'Module B: Basic
Concepts'.
User student_alice is currently on: Module C: Advanced Topics
User student_alice completed 'Module C: Advanced Topics' with score 75.

Final progress for student_alice: {'Module A: Introduction': 'completed',
'Module B: Basic Concepts': 'completed', 'Module C: Advanced Topics':
'completed', 'Module D: Application': 'not_started'}
Final scores for student_alice: {'Module A: Introduction': 85, 'Module B: Basic
Concepts': 50, 'Module C: Advanced Topics': 75, 'Module D: Application': 0}
```

Lab 4: Implement Fraud Detection in Banking and Finance

Aim: To implement a basic fraud detection system using machine learning techniques to identify anomalous transactions.

Procedure:

1. **Dataset Acquisition:** Obtain or simulate a transaction dataset containing features like transaction amount, time, location, and a 'fraud' label.
2. **Data Preprocessing:** Clean and preprocess the data, handling missing values, encoding categorical features, and scaling numerical features.
3. **Feature Engineering:** Create new features that might be indicative of fraud (e.g., frequency of transactions, ratio of transaction amount to average).
4. **Model Selection:** Choose an appropriate machine learning model for anomaly detection or classification (e.g., Isolation Forest, One-Class SVM, Logistic Regression, Random Forest).
5. **Model Training:** Train the selected model on the prepared dataset.
6. **Evaluation:** Evaluate the model's performance using metrics relevant to imbalanced datasets (e.g., Precision, Recall, F1-score, AUC-ROC).
7. **Prediction:** Use the trained model to predict fraud on new, unseen transactions.

Source Code (Conceptual Python using Scikit-learn):

```
# fraud_detection.py

import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import classification_report, confusion_matrix,
roc_auc_score
from sklearn.preprocessing import StandardScaler
import numpy as np

# --- Simulate a highly imbalanced dataset ---
def create_simulated_data(num_transactions=10000, fraud_ratio=0.01):
    data = {
        'transaction_amount': np.random.normal(500, 200, num_transactions),
        'transaction_hour': np.random.randint(0, 24, num_transactions),
        'location_risk_score': np.random.rand(num_transactions) * 10,
        'num_transactions_last_24h': np.random.randint(1, 10, num_transactions),
        'is_fraud': np.zeros(num_transactions, dtype=int)
    }
    df = pd.DataFrame(data)

    # Introduce fraud cases
    num_fraud = int(num_transactions * fraud_ratio)
    fraud_indices = np.random.choice(df.index, num_fraud, replace=False)
    df.loc[fraud_indices, 'is_fraud'] = 1

    # Make fraudulent transactions slightly different
    df.loc[fraud_indices, 'transaction_amount'] = np.random.normal(1500, 500,
num_fraud) # Higher amount
    df.loc[fraud_indices, 'transaction_hour'] = np.random.choice([2, 3, 22, 23],
num_fraud) # Odd hours
    df.loc[fraud_indices, 'location_risk_score'] = np.random.rand(num_fraud) * 5
+ 5 # Higher risk locations
    df.loc[fraud_indices, 'num_transactions_last_24h'] = np.random.randint(10,
20, num_fraud) # More frequent

    return df
```

```
if __name__ == "__main__":
    print("--- Fraud Detection System ---")

    # 1. Simulate Dataset
    df = create_simulated_data(num_transactions=10000, fraud_ratio=0.005)
    print(f"Dataset shape: {df.shape}")
    print(f"Fraudulent transactions: {df['is_fraud'].sum()}
({df['is_fraud'].mean()*100:.2f}%)")

    # 2. Data Preprocessing
    X = df.drop('is_fraud', axis=1)
    y = df['is_fraud']

    # Scale numerical features
    scaler = StandardScaler()
    X_scaled = scaler.fit_transform(X)
    X_scaled = pd.DataFrame(X_scaled, columns=X.columns)

    # 3. Data Splitting
    X_train, X_test, y_train, y_test = train_test_split(X_scaled, y,
test_size=0.3, random_state=42, stratify=y)

    print(f"\nTraining set size: {X_train.shape[0]} samples")
    print(f"Testing set size: {X_test.shape[0]} samples")

    # 4. Model Selection and Training (Random Forest Classifier)
    # RandomForest is good for imbalanced data and feature importance
    model = RandomForestClassifier(n_estimators=100, random_state=42,
class_weight='balanced')
    print("\nTraining the RandomForestClassifier model...")
    model.fit(X_train, y_train)
    print("Model training complete.")

    # 5. Prediction and Evaluation
    y_pred = model.predict(X_test)
    y_proba = model.predict_proba(X_test)[:, 1] # Probability of being fraud

    print("\n--- Model Evaluation ---")
    print("Classification Report:")
    print(classification_report(y_test, y_pred))

    print("\nConfusion Matrix:")
    print(confusion_matrix(y_test, y_pred))

    print(f"\nAUC-ROC Score: {roc_auc_score(y_test, y_proba):.4f}")

    # Example of predicting a new transaction
    print("\n--- Predicting a new transaction ---")
    new_transaction_data = pd.DataFrame([[1600, 2, 8.5, 15]], columns=X.columns)
    new_transaction_scaled = scaler.transform(new_transaction_data)

    prediction = model.predict(new_transaction_scaled)
    probability = model.predict_proba(new_transaction_scaled)[:, 1]

    print(f"New transaction: {new_transaction_data.iloc[0].to_dict()}")
    print(f"Predicted Fraud: {'Yes' if prediction[0] == 1 else 'No'}")
    print(f"Probability of Fraud: {probability[0]:.4f}")

    new_transaction_data_legit = pd.DataFrame([[100, 10, 2.0, 3]],
columns=X.columns)
    new_transaction_scaled_legit = scaler.transform(new_transaction_data_legit)

    prediction_legit = model.predict(new_transaction_scaled_legit)
    probability_legit = model.predict_proba(new_transaction_scaled_legit)[:, 1]
```

```
    print(f"\nNew legitimate-like transaction:
{new_transaction_data_legit.iloc[0].to_dict()}")
    print(f"Predicted Fraud: {'Yes' if prediction_legit[0] == 1 else 'No'}")
    print(f"Probability of Fraud: {probability_legit[0]:.4f}")
```

Input:

- A CSV or DataFrame containing transaction data with features such as `transaction_amount`, `transaction_hour`, `location_risk_score`, `num_transactions_last_24h`, and a target variable `is_fraud` (0 for legitimate, 1 for fraud).

Expected Output:

- Model performance metrics (Precision, Recall, F1-score, Confusion Matrix, AUC-ROC).
- Predictions (fraud/legitimate) for new, unseen transactions.

```
--- Fraud Detection System ---
Dataset shape: (10000, 5)
Fraudulent transactions: 50 (0.50%)

Training set size: 7000 samples
Testing set size: 3000 samples

Training the RandomForestClassifier model...
Model training complete.

--- Model Evaluation ---
Classification Report:
              precision    recall  f1-score   support

           0       1.00      1.00      1.00      2985
           1       0.89      0.80      0.84        15

    accuracy                           1.00      3000
   macro avg       0.94      0.90      0.92      3000
weighted avg       1.00      1.00      1.00      3000

Confusion Matrix:
[[2984    1]
 [   3   12]]

AUC-ROC Score: 0.9992

--- Predicting a new transaction ---
New transaction: {'transaction_amount': 1600.0, 'transaction_hour': 2.0,
'location_risk_score': 8.5, 'num_transactions_last_24h': 15.0}
Predicted Fraud: Yes
Probability of Fraud: 0.9900

New legitimate-like transaction: {'transaction_amount': 100.0,
'transaction_hour': 10.0, 'location_risk_score': 2.0,
'num_transactions_last_24h': 3.0}
Predicted Fraud: No
Probability of Fraud: 0.0000
```

Lab 5: Implement adaptive AI algorithms that can analyze student performance data, such as test scores and homework assignments

Aim: To implement an adaptive AI algorithm that analyzes student performance data to identify areas of weakness and suggest personalized interventions.

Procedure:

1. **Data Collection:** Simulate or acquire a dataset of student performance (e.g., student ID, assignment scores, test scores, topic mastery).
2. **Data Preprocessing:** Clean and prepare the data for analysis.
3. **Performance Analysis:** Implement algorithms to analyze performance trends (e.g., identifying declining scores, consistently low scores in specific topics).
4. **Adaptive Intervention Logic:** Develop rules or a simple model to recommend interventions (e.g., "review Module X," "practice problems on Topic Y," "suggest tutoring").
5. **Personalized Feedback:** Generate personalized feedback based on the analysis.

Source Code (Conceptual Python):

```
# student_performance_analysis.py

import pandas as pd
import numpy as np

class StudentPerformanceAnalyzer:
    def __init__(self, students_data):
        self.df = pd.DataFrame(students_data)
        self.df['average_score'] = self.df[['quiz_score', 'homework_score',
'exam_score']].mean(axis=1)

    def analyze_performance(self, student_id):
        student_data = self.df[self.df['student_id'] == student_id].iloc[0]

        analysis = {
            'student_id': student_id,
            'overall_average': student_data['average_score'],
            'quiz_score': student_data['quiz_score'],
            'homework_score': student_data['homework_score'],
            'exam_score': student_data['exam_score'],
            'weak_topics': [],
            'intervention_suggestions': []
        }

        # Identify weak topics (example based on low scores in specific
assignments/exams)
        if student_data['quiz_score'] < 60:
            analysis['weak_topics'].append('Quizzes/Basic Concepts')
        if student_data['homework_score'] < 60:
            analysis['weak_topics'].append('Homework/Application')
        if student_data['exam_score'] < 60:
            analysis['weak_topics'].append('Exams/Comprehensive Understanding')

        # Suggest interventions based on weaknesses
        if analysis['overall_average'] < 70:
            analysis['intervention_suggestions'].append('Overall review of
course material.')
        if 'Quizzes/Basic Concepts' in analysis['weak_topics']:
            analysis['intervention_suggestions'].append('Focus on foundational
concepts and practice quizzes.')
        if 'Homework/Application' in analysis['weak_topics']:
```

```
            analysis['intervention_suggestions'].append('Work through more
practice problems and case studies.')
        if 'Exams/Comprehensive Understanding' in analysis['weak_topics']:
            analysis['intervention_suggestions'].append('Review past exam
questions and consider group study.')

        if not analysis['weak_topics'] and analysis['overall_average'] >= 90:
            analysis['intervention_suggestions'].append('Excellent performance!
Consider advanced topics or enrichment activities.')

        return analysis

    def get_all_student_summaries(self):
        summaries = []
        for student_id in self.df['student_id'].unique():
            summaries.append(self.analyze_performance(student_id))
        return summaries

if __name__ == "__main__":
    # Simulate student data
    students_data = [
        {'student_id': 'S001', 'quiz_score': 85, 'homework_score': 90,
'exam_score': 88},
        {'student_id': 'S002', 'quiz_score': 55, 'homework_score': 65,
'exam_score': 50}, # Needs help
        {'student_id': 'S003', 'quiz_score': 70, 'homework_score': 75,
'exam_score': 68},
        {'student_id': 'S004', 'quiz_score': 95, 'homework_score': 98,
'exam_score': 92}  # High performer
    ]

    analyzer = StudentPerformanceAnalyzer(students_data)

    print("--- Student Performance Analysis ---")

    for summary in analyzer.get_all_student_summaries():
        print(f"\nAnalysis for Student {summary['student_id']}:")
        print(f"  Overall Average Score: {summary['overall_average']:.2f}")
        print(f"  Quiz Score: {summary['quiz_score']}")
        print(f"  Homework Score: {summary['homework_score']}")
        print(f"  Exam Score: {summary['exam_score']}")
        print(f"  Identified Weak Topics: {', '.join(summary['weak_topics']) if
summary['weak_topics'] else 'None'}")
        print(f"  Intervention Suggestions: {',
'.join(summary['intervention_suggestions']) if
summary['intervention_suggestions'] else 'None'}")
```

Input:

- A dataset (e.g., CSV, DataFrame) with student IDs and various performance metrics (e.g., `quiz_score`, `homework_score`, `exam_score`).

Expected Output:

- For each student, an analysis of their performance, including overall scores, identified weak areas, and personalized intervention suggestions.

```
--- Student Performance Analysis ---

Analysis for Student S001:
  Overall Average Score: 87.67
  Quiz Score: 85
```

Homework Score: 90
Exam Score: 88
Identified Weak Topics: None
Intervention Suggestions: None

Analysis for Student S002:
Overall Average Score: 56.67
Quiz Score: 55
Homework Score: 65
Exam Score: 50
Identified Weak Topics: Quizzes/Basic Concepts, Exams/Comprehensive Understanding
Intervention Suggestions: Overall review of course material., Focus on foundational concepts and practice quizzes., Review past exam questions and consider group study.

Analysis for Student S003:
Overall Average Score: 71.00
Quiz Score: 70
Homework Score: 75
Exam Score: 68
Identified Weak Topics: None
Intervention Suggestions: None

Analysis for Student S004:
Overall Average Score: 95.00
Quiz Score: 95
Homework Score: 98
Exam Score: 92
Identified Weak Topics: None
Intervention Suggestions: Excellent performance! Consider advanced topics or enrichment activities.

Lab 6: Implement adaptive AI algorithms that can analyze traffic patterns and adjust traffic lights in real-time to optimize traffic flow.

Aim: To simulate an adaptive traffic light control system that uses AI to optimize traffic flow based on real-time traffic patterns.

Procedure:

1. **Traffic Data Simulation:** Simulate traffic flow data at an intersection (e.g., vehicle counts per lane, waiting times).
2. **Define Traffic Light States:** Define possible states for traffic lights (e.g., green for N-S, green for E-W).
3. **Optimization Objective:** Define an objective function to optimize (e.g., minimize total waiting time, maximize throughput).
4. **Adaptive Control Logic:** Implement a simple adaptive algorithm (e.g., rule-based, Q-learning inspired) that adjusts traffic light timings based on simulated traffic conditions.
5. **Simulation:** Run a short simulation to demonstrate the adaptive behavior.

Source Code (Conceptual Python Simulation):

```
# adaptive_traffic_lights.py

import time
import random

class TrafficIntersection:
    def __init__(self, lanes):
        self.lanes = {lane: {'queue': 0, 'flow_rate': random.randint(1, 5)} for
lane in lanes}
        self.light_states = {
            'NS': {'green': ['North', 'South'], 'red': ['East', 'West']},
            'EW': {'green': ['East', 'West'], 'red': ['North', 'South']}
        }
        self.current_light_phase = 'NS' # Start with North-South green
        self.timer = 0
        self.max_green_time = 30 # Max time for a phase
        self.min_green_time = 10 # Min time for a phase

    def update_traffic(self, time_step=1):
        for lane, data in self.lanes.items():
            # Simulate new cars arriving
            new_arrivals = random.randint(0, data['flow_rate'])
            self.lanes[lane]['queue'] += new_arrivals

            # Cars leaving on green light
            if lane in self.light_states[self.current_light_phase]['green']:
                cars_cleared = min(self.lanes[lane]['queue'], random.randint(5,
10)) # Simulate cars clearing
                self.lanes[lane]['queue'] -= cars_cleared
                self.lanes[lane]['queue'] = max(0, self.lanes[lane]['queue']) #
Queue can't be negative
        self.timer += time_step

    def get_queue_lengths(self):
        return {lane: data['queue'] for lane, data in self.lanes.items()}

    def adapt_light_phase(self):
        # Adaptive logic: Switch if current phase has low queue or other phase
has high queue
```

```
        current_green_lanes =
self.light_states[self.current_light_phase]['green']
        other_phase = 'EW' if self.current_light_phase == 'NS' else 'NS'
        other_green_lanes = self.light_states[other_phase]['green']

        current_queue_sum = sum(self.lanes[lane]['queue'] for lane in
current_green_lanes)
        other_queue_sum = sum(self.lanes[lane]['queue'] for lane in
other_green_lanes)

        # Conditions to switch:
        # 1. Current phase has been green for max time
        # 2. Current phase has very low queue AND other phase has high queue
        # 3. Current phase has been green for min time AND other phase has
significantly higher queue

        switch_condition = False
        if self.timer >= self.max_green_time:
            switch_condition = True
            print(f"  --> Max green time reached for {self.current_light_phase}
phase.")
        elif self.timer >= self.min_green_time and current_queue_sum < 5 and
other_queue_sum > 20:
            switch_condition = True
            print(f"  --> Current phase ({self.current_light_phase}) clear,
other phase ({other_phase}) congested.")
        elif self.timer >= self.min_green_time and other_queue_sum >
current_queue_sum * 2 and other_queue_sum > 15:
            switch_condition = True
            print(f"  --> Other phase ({other_phase}) significantly more
congested.")

        if switch_condition:
            print(f"Switching from {self.current_light_phase} to {other_phase}
phase.")
            self.current_light_phase = other_phase
            self.timer = 0 # Reset timer for new phase

    def run_simulation(self, total_time_steps=100):
        print("--- Adaptive Traffic Light Simulation ---")
        print(f"Initial Light Phase: {self.current_light_phase}")
        print(f"Initial Queues: {self.get_queue_lengths()}")

        for step in range(total_time_steps):
            self.update_traffic()
            self.adapt_light_phase()

            print(f"\nTime Step {step+1}:")
            print(f"  Current Light Phase: {self.current_light_phase} (Timer:
{self.timer}s)")
            print(f"  Queue Lengths: {self.get_queue_lengths()}")
            # time.sleep(0.1) # Uncomment for slower simulation

if __name__ == "__main__":
    lanes_at_intersection = ['North', 'South', 'East', 'West']
    intersection = TrafficIntersection(lanes_at_intersection)
    intersection.run_simulation(total_time_steps=50)
```

Input:

- Simulated traffic data (e.g., vehicle arrival rates, initial queue lengths).
- Defined traffic light phases and timings.

Expected Output:

- Real-time updates on traffic light states.
- Changes in queue lengths at different lanes.
- Demonstration of adaptive switching of traffic light phases based on simulated traffic conditions to minimize congestion.

```
--- Adaptive Traffic Light Simulation ---
Initial Light Phase: NS
Initial Queues: {'North': 0, 'South': 0, 'East': 0, 'West': 0}

Time Step 1:
  Current Light Phase: NS (Timer: 1s)
  Queue Lengths: {'North': 2, 'South': 0, 'East': 4, 'West': 1}

... (intermediate steps) ...

Time Step 9:
  Current Light Phase: NS (Timer: 9s)
  Queue Lengths: {'North': 0, 'South': 0, 'East': 20, 'West': 15}

Time Step 10:
  Current Light Phase: NS (Timer: 10s)
  Queue Lengths: {'North': 0, 'South': 0, 'East': 23, 'West': 18}
  --> Current phase (NS) clear, other phase (EW) congested.
Switching from NS to EW phase.

Time Step 11:
  Current Light Phase: EW (Timer: 1s)
  Queue Lengths: {'North': 3, 'South': 4, 'East': 15, 'West': 10}

... (intermediate steps) ...

Time Step 20:
  Current Light Phase: EW (Timer: 10s)
  Queue Lengths: {'North': 10, 'South': 12, 'East': 0, 'West': 0}
  --> Current phase (EW) clear, other phase (NS) congested.
Switching from EW to NS phase.

Time Step 21:
  Current Light Phase: NS (Timer: 1s)
  Queue Lengths: {'North': 5, 'South': 7, 'East': 2, 'West': 3}
```

Lab 7: Understanding Predictive Models: Identify and discuss examples of predictive, descriptive, and decision models.

Aim: To understand and differentiate between predictive, descriptive, and decision models, providing examples for each.

Procedure:

1. **Research Definitions:** Define predictive, descriptive, and decision models.
2. **Identify Characteristics:** List the key characteristics and purposes of each model type.
3. **Find Real-World Examples:** Research and identify at least three distinct real-world examples for each model type.
4. **Discuss Applications:** Explain how each example functions and what kind of insights or actions it enables.
5. **Comparative Analysis:** Create a brief comparative analysis highlighting the differences and overlaps.

Source Code: (Not applicable for this theoretical lab. This lab focuses on conceptual understanding and discussion.)

Input: (Not applicable. This lab involves research and conceptual understanding.)

Expected Output: A structured discussion document with definitions, characteristics, and examples for each model type.

Example Structure for Output:

1. Predictive Models

- **Definition:** Models that forecast future outcomes or probabilities based on historical data. They answer "What will happen?"
- **Characteristics:** Focus on forecasting, often use supervised learning, output is a prediction (e.g., a value, a class).
- **Examples:**
  - **Stock Price Prediction:** Predicts future stock prices based on historical data, market trends, and economic indicators. Used by investors to make buying/selling decisions.
  - **Customer Churn Prediction:** Identifies customers likely to cancel a service based on usage patterns, demographics, and past interactions. Allows companies to proactively offer incentives to retain customers.
  - **Weather Forecasting:** Predicts future weather conditions (temperature, precipitation) using atmospheric data and climate models. Essential for planning and disaster preparedness.

2. Descriptive Models

- **Definition:** Models that summarize and describe past or current data to gain insights into relationships and patterns. They answer "What happened?" or "What is happening?"
- **Characteristics:** Focus on understanding data, often use unsupervised learning or statistical analysis, output is a summary or pattern.
- **Examples:**

    - **Customer Segmentation:** Groups customers into distinct segments based on their purchasing behavior, demographics, or preferences. Helps businesses tailor marketing strategies.
    - **Market Basket Analysis:** Identifies items frequently purchased together (e.g., "customers who buy bread also buy milk"). Used for product placement and cross-selling.
    - **Fraud Pattern Analysis:** Analyzes past fraudulent transactions to identify common characteristics and patterns of fraudulent activity. Helps in building rules for fraud detection.

3. Decision Models

- **Definition:** Models that recommend actions or strategies to optimize outcomes, often incorporating elements of predictive and descriptive analysis. They answer "What should we do?"
- **Characteristics:** Focus on prescriptive actions, often involve optimization, simulation, or rule-based systems.
- **Examples:**
    - **Dynamic Pricing Models:** Determines the optimal price for a product or service in real-time based on demand, supply, competitor prices, and customer segments. Used in e-commerce and ride-sharing.
    - **Loan Approval Systems:** Decides whether to approve a loan application based on an applicant's credit score, income, and other financial indicators, aiming to minimize risk.
    - **Inventory Optimization:** Recommends optimal stock levels for products to minimize holding costs and avoid stockouts, considering demand forecasts and lead times.

Lab 8: Analytical Techniques Overview: Create a comparative analysis chart highlighting different analytical techniques and their applications.

Aim: To provide a comprehensive overview of various analytical techniques used in data analytics and their respective applications.

Procedure:

1. **Categorize Techniques:** Group analytical techniques into logical categories (e.g., statistical, machine learning, data mining).
2. **Select Key Techniques:** Choose at least 10-15 significant analytical techniques.
3. **Describe Each Technique:** For each selected technique, provide a brief description of its purpose and how it works.
4. **Identify Applications:** List common real-world applications where each technique is effectively used.
5. **Create Comparative Chart:** Organize the information into a clear and concise comparative chart or table.

Source Code: (Not applicable for this theoretical lab.)

Input: (Not applicable. This lab involves research and conceptual understanding.)

Expected Output: A comparative analysis chart (table) summarizing various analytical techniques and their applications.

Example Table Structure for Output:

Analytical Technique	Category	Description	Common Applications
Regression Analysis	Statistical/ML	Models the relationship between a dependent variable and one or more independent variables.	Sales forecasting, predicting housing prices, risk assessment.
Classification	Machine Learning	Assigns data points to predefined categories or classes.	Spam detection, medical diagnosis, customer churn prediction.
Clustering	Machine Learning	Groups similar data points together into clusters without prior labels.	Customer segmentation, anomaly detection, document clustering.
Time Series Analysis	Statistical	Analyzes time-ordered data to identify patterns and forecast future values.	Stock market prediction, demand forecasting, weather forecasting.
Decision Trees	Machine Learning	Uses a tree-like model of decisions and their possible consequences.	Classification, regression, medical diagnosis, credit scoring.
Neural Networks	Machine Learning	Inspired by the human brain, used for complex pattern recognition.	Image recognition, natural language processing, speech recognition, fraud detection.
Association Rule Mining	Data Mining	Discovers relationships between variables in large datasets.	Market basket analysis (e.g., "customers who buy X also buy Y"), recommendation systems.

Principal Component Analysis (PCA)	Dimensionality Reduction	Reduces the number of features in a dataset while retaining most variance.	Image compression, noise reduction, data visualization, pre-processing for other ML models.
Hypothesis Testing	Statistical	Uses statistical methods to test assumptions about a population based on sample data.	A/B testing, clinical trials, quality control.
Sentiment Analysis	Natural Language Processing	Determines the emotional tone behind a piece of text.	Social media monitoring, customer feedback analysis, brand reputation management.
Survival Analysis	Statistical	Models the time until one or more events happen, such as death or failure.	Medical research (patient survival rates), customer churn in subscription services, equipment failure prediction.
Simulation Modeling	Operations Research	Creates a computer model of a real-world system to study its behavior.	Supply chain optimization, queue management, risk analysis, process improvement.

Lab 9: Data Transformation Techniques: Implement data transformations for individual and multiple predictors using Python.

Aim: To implement various data transformation techniques in Python to prepare data for machine learning models, focusing on individual and multiple predictors.

Procedure:

1. **Dataset Loading:** Load a sample dataset (e.g., from scikit-learn or a CSV).
2. **Identify Data Types:** Understand numerical and categorical features.
3. **Individual Predictor Transformations:**
   - **Scaling:** Implement Min-Max Scaling and Standardization (Z-score normalization).
   - **Log Transformation:** Apply logarithmic transformation for skewed numerical data.
   - **Power Transformation:** Implement Box-Cox or Yeo-Johnson transformation.
4. **Multiple Predictors Transformations:**
   - **One-Hot Encoding:** Apply one-hot encoding for categorical features.
   - **Polynomial Features:** Generate polynomial features to capture non-linear relationships.
   - **Interaction Features:** Create interaction terms between relevant features.
5. **Visualize Effects:** (Optional) Visualize the distribution of features before and after transformation.

Source Code (Python using Pandas and Scikit-learn):

```
# data_transformation.py

import pandas as pd
import numpy as np
from sklearn.preprocessing import MinMaxScaler, StandardScaler,
PowerTransformer, OneHotEncoder, PolynomialFeatures
from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline
from sklearn.datasets import load_diabetes # Example dataset

if __name__ == "__main__":
    print("--- Data Transformation Techniques ---")

    # 1. Load a sample dataset
    data = load_diabetes(as_frame=True)
    df = data.frame
    print("Original DataFrame head:")
    print(df.head())
    print("\nOriginal DataFrame info:")
    df.info()

    # Separate features (X) and target (y)
    X = df.drop(columns=['target'])
    y = df['target']

    # Identify numerical and categorical features (diabetes dataset is all
numerical,
    # let's simulate a categorical one for demonstration)
    # For demonstration, let's assume 'sex' is categorical and 'bmi' is skewed
    numerical_features = ['age', 'bmi', 'bp', 's1', 's2', 's3', 's4', 's5',
's6']
    categorical_features = ['sex'] # 'sex' is actually numerical (0/1) but we'll
treat as categorical for encoding example

    # Let's artificially make 'bmi' skewed for log/power transformation example
```

```
    # In a real scenario, you'd check df['bmi'].hist()
    X_transformed = X.copy()
    X_transformed['bmi'] = X_transformed['bmi']**2 # Make it skewed for
demonstration

    print("\n--- Individual Predictor Transformations ---")

    # Min-Max Scaling
    min_max_scaler = MinMaxScaler()
    X_transformed['bmi_minmax_scaled'] =
min_max_scaler.fit_transform(X_transformed[['bmi']])
    print("\nBMI after Min-Max Scaling (first 5 values):")
    print(X_transformed['bmi_minmax_scaled'].head())

    # Standardization (Z-score normalization)
    std_scaler = StandardScaler()
    X_transformed['bmi_std_scaled'] =
std_scaler.fit_transform(X_transformed[['bmi']])
    print("\nBMI after Standardization (first 5 values):")
    print(X_transformed['bmi_std_scaled'].head())

    # Log Transformation (useful for positively skewed data)
    # Add a small constant to avoid log(0) if data contains zeros
    X_transformed['bmi_log_transformed'] = np.log1p(X_transformed['bmi']) #
log1p handles 0 values
    print("\nBMI after Log Transformation (first 5 values):")
    print(X_transformed['bmi_log_transformed'].head())

    # Power Transformation (Yeo-Johnson, handles positive and negative values)
    power_transformer = PowerTransformer(method='yeo-johnson')
    X_transformed['bmi_power_transformed'] =
power_transformer.fit_transform(X_transformed[['bmi']])
    print("\nBMI after Power Transformation (first 5 values):")
    print(X_transformed['bmi_power_transformed'].head())

    print("\n--- Multiple Predictors Transformations ---")

    # One-Hot Encoding for 'sex' (assuming it's categorical)
    # Use ColumnTransformer for robust handling of different column types
    preprocessor = ColumnTransformer(
        transformers=[
            ('num', StandardScaler(), numerical_features),
            ('cat', OneHotEncoder(handle_unknown='ignore'),
categorical_features)
        ])

    # Create a dummy DataFrame with mixed types for demonstration of
ColumnTransformer
    df_mixed = pd.DataFrame({
        'numerical_col1': np.random.rand(10) * 100,
        'numerical_col2': np.random.normal(50, 10, 10),
        'categorical_col': ['A', 'B', 'A', 'C', 'B', 'A', 'C', 'B', 'A', 'C']
    })

    # Redefine features for this mixed dataframe
    numerical_features_mixed = ['numerical_col1', 'numerical_col2']
    categorical_features_mixed = ['categorical_col']

    preprocessor_mixed = ColumnTransformer(
        transformers=[
            ('num', StandardScaler(), numerical_features_mixed),
            ('cat', OneHotEncoder(handle_unknown='ignore'),
categorical_features_mixed)
        ])

    X_processed_mixed = preprocessor_mixed.fit_transform(df_mixed)
```

```
    print("\nDataFrame after One-Hot Encoding and Standardization (Mixed
Data):")
    # Convert back to DataFrame for better readability
    # Get feature names after one-hot encoding
    ohe_feature_names =
preprocessor_mixed.named_transformers_['cat'].get_feature_names_out(categorical_
features_mixed)
    all_feature_names = numerical_features_mixed + list(ohe_feature_names)
    print(pd.DataFrame(X_processed_mixed, columns=all_feature_names).head())


    # Polynomial Features and Interaction Terms
    # Let's use 'bmi' and 'bp' from the original diabetes dataset for this
    poly = PolynomialFeatures(degree=2, include_bias=False)
    poly_features = poly.fit_transform(X[['bmi', 'bp']])

    # Get feature names for polynomial features
    poly_feature_names = poly.get_feature_names_out(['bmi', 'bp'])
    X_poly_df = pd.DataFrame(poly_features, columns=poly_feature_names)
    print("\nPolynomial Features (degree 2) for 'bmi' and 'bp' (first 5 rows):")
    print(X_poly_df.head())

    print("\n--- Summary of Transformations Applied ---")
    print("Individual transformations demonstrated: Min-Max Scaling,
Standardization, Log Transformation, Power Transformation.")
    print("Multiple predictors transformations demonstrated: One-Hot Encoding,
Polynomial Features.")
    print("These techniques are crucial for preparing data for various machine
learning models.")
```

Input:

- A Pandas DataFrame containing numerical and/or categorical features. For demonstration, the `load_diabetes` dataset is used, and a dummy mixed dataset is created.

Expected Output:

- Transformed DataFrames or arrays showing the effect of each applied transformation (e.g., scaled numerical values, one-hot encoded columns, new polynomial features).
- Confirmation of the transformations applied.

```
--- Data Transformation Techniques ---
Original DataFrame head:
        age       sex       bmi        bp        s1        s2        s3
s4        s5        s6  target
0  0.038076  0.050680  0.061696  0.021872 -0.044223 -0.034821 -0.043401 -
0.002592  0.019907 -0.017646   151.0
1 -0.001882 -0.044642 -0.051474 -0.026328 -0.008449 -0.019163  0.074412 -
0.039493 -0.068332 -0.092204    75.0
2  0.085299  0.050680  0.044451 -0.005671 -0.045599 -0.034194 -0.032356 -
0.002592  0.002861 -0.025930   141.0
3 -0.089063 -0.044642 -0.011595 -0.036656  0.012191  0.024991 -0.036038
0.034309  0.022688 -0.009362   206.0
4  0.008449 -0.044642 -0.036385  0.021872  0.003935  0.015596  0.008142 -
0.002592 -0.031988 -0.046641   135.0

Original DataFrame info:
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 442 entries, 0 to 441
Data columns (total 11 columns):
 #   Column      Non-Null Count  Dtype
---  ------      --------------  -----
 0   age         442 non-null    float64
```

```
 1   sex         442 non-null    float64
 2   bmi         442 non-null    float64
 3   bp          442 non-null    float64
 4   s1          442 non-null    float64
 5   s2          442 non-null    float64
 6   s3          442 non-null    float64
 7   s4          442 non-null    float64
 8   s5          442 non-null    float64
 9   s6          442 non-null    float64
 10  target      442 non-null    float64
dtypes: float64(11)
memory usage: 38.1 KB

--- Individual Predictor Transformations ---

BMI after Min-Max Scaling (first 5 values):
0    0.697523
1    0.334057
2    0.627616
3    0.470072
4    0.380486
Name: bmi_minmax_scaled, dtype: float64

BMI after Standardization (first 5 values):
0    1.002956
1   -0.900492
2    0.730302
3   -0.089201
4   -0.569429
Name: bmi_std_scaled, dtype: float64

BMI after Log Transformation (first 5 values):
0    0.060136
1   -0.052844
2    0.043485
3   -0.011663
4   -0.036987
Name: bmi_log_transformed, dtype: float64

BMI after Power Transformation (first 5 values):
0    1.002956
1   -0.900492
2    0.730302
3   -0.089201
4   -0.569429
Name: bmi_power_transformed, dtype: float64

--- Multiple Predictors Transformations ---

DataFrame after One-Hot Encoding and Standardization (Mixed Data):
   numerical_col1  numerical_col2  categorical_col_A  categorical_col_B
categorical_col_C
0        0.478950       -1.229158           1.000000           0.000000
0.000000
1        0.669814        1.479532           0.000000           1.000000
0.000000
2        1.139626       -0.505779           1.000000           0.000000
0.000000
3       -1.218556       -0.087799           0.000000           0.000000
1.000000
4       -0.655866        0.457813           0.000000           1.000000
0.000000

Polynomial Features (degree 2) for 'bmi' and 'bp' (first 5 rows):
        bmi        bp      bmi^2  bmi bp      bp^2
0  0.061696  0.021872   0.003807  0.001349  0.000478
1 -0.051474 -0.026328   0.002649  0.001355  0.000693
```

```
2  0.044451 -0.005671   0.001976 -0.000252  0.000032
3 -0.011595 -0.036656   0.000134  0.000425  0.001344
4 -0.036385  0.021872   0.001324 -0.000796  0.000478

--- Summary of Transformations Applied ---
Individual transformations demonstrated: Min-Max Scaling, Standardization, Log
Transformation, Power Transformation.
Multiple predictors transformations demonstrated: One-Hot Encoding, Polynomial
Features.
These techniques are crucial for preparing data for various machine learning
models.
```

Lab 10: Dealing with Missing Values: Practice techniques for handling missing data such as imputation or removal.

Aim: To practice various techniques for handling missing values in a dataset, including removal and different imputation methods.

Procedure:

1. **Load Dataset with Missing Values:** Load a dataset that contains missing values (e.g., `NaN`). If not available, introduce missing values artificially.
2. **Identify Missing Values:** Detect and quantify missing values in the dataset.
3. **Removal Techniques:**
   - **Row-wise Deletion:** Remove rows containing any missing values.
   - **Column-wise Deletion:** Remove columns with a high percentage of missing values.
4. **Imputation Techniques:**
   - **Mean/Median/Mode Imputation:** Impute missing numerical values with the mean or median, and categorical with the mode.
   - **Constant Value Imputation:** Impute with a specific constant (e.g., 0 or a placeholder).
   - **Forward/Backward Fill:** Impute using the previous or next valid observation (for time series or ordered data).
   - **K-Nearest Neighbors (KNN) Imputation:** Impute missing values based on the values of the k-nearest neighbors.
   - **Regression Imputation:** Predict missing values using a regression model trained on available data.
5. **Compare Effects:** (Optional) Compare the impact of different handling techniques on data distribution or a simple model's performance.

Source Code (Python using Pandas and Scikit-learn):

```
# missing_values_handling.py

import pandas as pd
import numpy as np
from sklearn.impute import SimpleImputer, KNNImputer
from sklearn.experimental import enable_iterative_imputer # Required for
IterativeImputer
from sklearn.impute import IterativeImputer
from sklearn.linear_model import BayesianRidge # Default estimator for
IterativeImputer

if __name__ == "__main__":
    print("--- Dealing with Missing Values ---")

    # 1. Create a sample DataFrame with missing values
    data = {
        'FeatureA': [10, 20, np.nan, 40, 50, 60, np.nan, 80],
        'FeatureB': [1, np.nan, 3, 4, 5, np.nan, 7, 8],
        'FeatureC': ['X', 'Y', 'X', np.nan, 'Z', 'Y', 'X', 'Z'],
        'FeatureD': [100, 200, 300, 400, 500, 600, 700, np.nan]
    }
    df = pd.DataFrame(data)
    print("Original DataFrame with Missing Values:")
    print(df)
    print("\nMissing values count per column:")
    print(df.isnull().sum())
```

```
    print("\n--- Removal Techniques ---")

    # Row-wise Deletion (dropna)
    df_row_dropped = df.dropna()
    print("\nDataFrame after Row-wise Deletion:")
    print(df_row_dropped)
    print(f"Original rows: {len(df)}, Rows after deletion:
{len(df_row_dropped)}")

    # Column-wise Deletion (if > 50% missing, for example)
    df_col_dropped = df.copy()
    for col in df_col_dropped.columns:
        if df_col_dropped[col].isnull().sum() / len(df_col_dropped) > 0.5: #
Example threshold
            df_col_dropped = df_col_dropped.drop(columns=[col])
    print("\nDataFrame after Column-wise Deletion (if > 50% missing):")
    print(df_col_dropped) # In this example, no column meets the 50% threshold

    print("\n--- Imputation Techniques ---")

    # Mean Imputation (Numerical)
    df_mean_imputed = df.copy()
    mean_imputer = SimpleImputer(strategy='mean')
    df_mean_imputed[['FeatureA', 'FeatureB', 'FeatureD']] =
mean_imputer.fit_transform(df_mean_imputed[['FeatureA', 'FeatureB',
'FeatureD']])
    print("\nDataFrame after Mean Imputation (Numerical):")
    print(df_mean_imputed)

    # Median Imputation (Numerical)
    df_median_imputed = df.copy()
    median_imputer = SimpleImputer(strategy='median')
    df_median_imputed[['FeatureA', 'FeatureB', 'FeatureD']] =
median_imputer.fit_transform(df_median_imputed[['FeatureA', 'FeatureB',
'FeatureD']])
    print("\nDataFrame after Median Imputation (Numerical):")
    print(df_median_imputed)

    # Mode Imputation (Categorical)
    df_mode_imputed = df.copy()
    mode_imputer = SimpleImputer(strategy='most_frequent')
    df_mode_imputed[['FeatureC']] =
mode_imputer.fit_transform(df_mode_imputed[['FeatureC']])
    print("\nDataFrame after Mode Imputation (Categorical):")
    print(df_mode_imputed)

    # Constant Value Imputation (e.g., 0 for numerical, 'Missing' for
categorical)
    df_constant_imputed = df.copy()
    df_constant_imputed['FeatureA'] = df_constant_imputed['FeatureA'].fillna(0)
    df_constant_imputed['FeatureC'] =
df_constant_imputed['FeatureC'].fillna('Missing')
    print("\nDataFrame after Constant Imputation (0 for A, 'Missing' for C):")
    print(df_constant_imputed)

    # KNN Imputation (Numerical)
    df_knn_imputed = df.copy()
    knn_imputer = KNNImputer(n_neighbors=2)
    # KNNImputer works best on numerical data, so we'll impute numerical columns
    df_knn_imputed[['FeatureA', 'FeatureB', 'FeatureD']] =
knn_imputer.fit_transform(df_knn_imputed[['FeatureA', 'FeatureB', 'FeatureD']])
    print("\nDataFrame after KNN Imputation (Numerical, k=2):")
    print(df_knn_imputed)

    # Regression Imputation (IterativeImputer)
    # This is more advanced and imputes values based on other features
    df_iterative_imputed = df.copy()
```

```
    # Need to convert categorical to numerical first for IterativeImputer
    df_iterative_imputed['FeatureC_encoded'] =
df_iterative_imputed['FeatureC'].astype('category').cat.codes
    # Replace -1 (for NaN) with actual NaN for IterativeImputer to handle
    df_iterative_imputed['FeatureC_encoded'] =
df_iterative_imputed['FeatureC_encoded'].replace(-1, np.nan)

    iterative_imputer = IterativeImputer(max_iter=10, random_state=0)
    # Impute all numerical columns, including the encoded categorical one
    imputed_data =
iterative_imputer.fit_transform(df_iterative_imputed[['FeatureA', 'FeatureB',
'FeatureD', 'FeatureC_encoded']])
    df_iterative_imputed_result = pd.DataFrame(imputed_data,
columns=['FeatureA', 'FeatureB', 'FeatureD', 'FeatureC_encoded'])
    print("\nDataFrame after Iterative Imputation (Numerical & Encoded
Categorical):")
    print(df_iterative_imputed_result)

    print("\n--- Summary ---")
    print("Various techniques for handling missing values have been
demonstrated.")
    print("The choice of technique depends on the nature of the data and the
extent of missingness.")
```

Input:

- A Pandas DataFrame with explicitly marked missing values (e.g., `np.nan`).

Expected Output:

- DataFrames showing the effect of each missing value handling technique (rows/columns removed, imputed values).
- A summary of missing values before and after transformations.

```
--- Dealing with Missing Values ---
Original DataFrame with Missing Values:
   FeatureA  FeatureB FeatureC  FeatureD
0      10.0       1.0        X     100.0
1      20.0       NaN        Y     200.0
2       NaN       3.0        X     300.0
3      40.0       4.0      NaN     400.0
4      50.0       5.0        Z     500.0
5      60.0       NaN        Y     600.0
6       NaN       7.0        X     700.0
7      80.0       8.0        Z       NaN

Missing values count per column:
FeatureA    2
FeatureB    2
FeatureC    1
FeatureD    1
dtype: int64

--- Removal Techniques ---

DataFrame after Row-wise Deletion:
   FeatureA  FeatureB FeatureC  FeatureD
0      10.0       1.0        X     100.0
4      50.0       5.0        Z     500.0

DataFrame after Column-wise Deletion (if > 50% missing):
   FeatureA  FeatureB FeatureC  FeatureD
0      10.0       1.0        X     100.0
```

```
1      20.0       NaN        Y     200.0
2       NaN       3.0        X     300.0
3      40.0       4.0      NaN     400.0
4      50.0       5.0        Z     500.0
5      60.0       NaN        Y     600.0
6       NaN       7.0        X     700.0
7      80.0       8.0        Z       NaN

--- Imputation Techniques ---

DataFrame after Mean Imputation (Numerical):
   FeatureA   FeatureB FeatureC   FeatureD
0      10.0   1.000000        X  100.00000
1      20.0   4.666667        Y  200.00000
2      43.0   3.000000        X  300.00000
3      40.0   4.000000      NaN  400.00000
4      50.0   5.000000        Z  500.00000
5      60.0   4.666667        Y  600.00000
6      43.0   7.000000        X  700.00000
7      80.0   8.000000        Z  400.00000

DataFrame after Median Imputation (Numerical):
   FeatureA  FeatureB FeatureC  FeatureD
0      10.0       1.0        X     100.0
1      20.0       4.5        Y     200.0
2      45.0       3.0        X     300.0
3      40.0       4.0      NaN     400.0
4      50.0       5.0        Z     500.0
5      60.0       4.5        Y     600.0
6      45.0       7.0        X     700.0
7      80.0       8.0        Z     400.0

DataFrame after Mode Imputation (Categorical):
   FeatureA  FeatureB FeatureC  FeatureD
0      10.0       1.0        X     100.0
1      20.0       NaN        Y     200.0
2       NaN       3.0        X     300.0
3      40.0       4.0        X     400.0
4      50.0       5.0        Z     500.0
5      60.0       NaN        Y     600.0
6       NaN       7.0        X     700.0
7      80.0       8.0        Z       NaN

DataFrame after Constant Imputation (0 for A, 'Missing' for C):
   FeatureA  FeatureB FeatureC  FeatureD
0      10.0       1.0        X     100.0
1      20.0       NaN        Y     200.0
2       0.0       3.0        X     300.0
3      40.0       4.0  Missing     400.0
4      50.0       5.0        Z     500.0
5      60.0       NaN        Y     600.0
6       0.0       7.0        X     700.0
7      80.0       8.0        Z       NaN

DataFrame after KNN Imputation (Numerical, k=2):
   FeatureA  FeatureB FeatureC  FeatureD
0      10.0  1.000000        X     100.0
1      20.0  4.000000        Y     200.0
2      45.0  3.000000        X     300.0
3      40.0  4.000000      NaN     400.0
4      50.0  5.000000        Z     500.0
5      60.0  5.500000        Y     600.0
6      45.0  7.000000        X     700.0
7      80.0  8.000000        Z     550.0

DataFrame after Iterative Imputation (Numerical & Encoded Categorical):
   FeatureA  FeatureB  FeatureD  FeatureC_encoded
```

```
0  10.000000  1.000000  100.000000          2.000000
1  20.000000  4.666667  200.000000          1.000000
2  43.000000  3.000000  300.000000          2.000000
3  40.000000  4.000000  400.000000          1.666667
4  50.000000  5.000000  500.000000          0.000000
5  60.000000  4.666667  600.000000          1.000000
6  43.000000  7.000000  700.000000          2.000000
7  80.000000  8.000000  400.000000          0.000000

--- Summary ---
Various techniques for handling missing values have been demonstrated.
The choice of technique depends on the nature of the data and the extent of
missingness.
```

Lab 11: Model Tuning and Data Splitting: Split datasets into training and testing sets, perform model tuning, and evaluate performance.

Aim: To understand and implement proper data splitting strategies and perform hyperparameter tuning to optimize model performance.

Procedure:

1. **Load Dataset:** Load a suitable dataset for a classification or regression task (e.g., Iris, Breast Cancer, Boston Housing).
2. **Data Splitting:** Split the dataset into training and testing sets using `train_test_split`. Discuss the importance of `random_state` and `stratify` (for classification).
3. **Model Selection:** Choose a machine learning model (e.g., Logistic Regression, Support Vector Machine, Decision Tree).
4. **Hyperparameter Tuning:**
   - Define a range of hyperparameters to tune.
   - Implement **Grid Search** or **Randomized Search** with cross-validation (`GridSearchCV`, `RandomizedSearchCV`).
   - Identify the best hyperparameters.
5. **Model Training:** Train the model with the best hyperparameters on the training data.
6. **Evaluation:** Evaluate the final model's performance on the unseen testing set using appropriate metrics.

Source Code (Python using Scikit-learn):

```
# model_tuning_data_splitting.py

import pandas as pd
from sklearn.model_selection import train_test_split, GridSearchCV,
RandomizedSearchCV
from sklearn.linear_model import LogisticRegression
from sklearn.svm import SVC
from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import accuracy_score, classification_report,
confusion_matrix
from sklearn.datasets import load_breast_cancer # Example classification dataset

if __name__ == "__main__":
    print("--- Model Tuning and Data Splitting ---")

    # 1. Load Dataset (Breast Cancer dataset for classification)
    data = load_breast_cancer(as_frame=True)
    X = data.data
    y = data.target
    print(f"Dataset loaded: {X.shape[0]} samples, {X.shape[1]} features.")
    print(f"Target distribution (0: Malignant, 1: Benign):\n{y.value_counts()}")

    # 2. Data Splitting
    # Stratify is crucial for imbalanced classification datasets to maintain
target distribution
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3,
random_state=42, stratify=y)
    print(f"\nTraining set size: {X_train.shape[0]} samples")
    print(f"Testing set size: {X_test.shape[0]} samples")
    print(f"Training target
distribution:\n{y_train.value_counts(normalize=True)}")
    print(f"Testing target
distribution:\n{y_test.value_counts(normalize=True)}")
```

```
    # 3. Model Selection (Decision Tree Classifier for demonstration)
    model = DecisionTreeClassifier(random_state=42)
    print("\nInitial Decision Tree model created.")

    # 4. Hyperparameter Tuning using GridSearchCV
    print("\n--- Hyperparameter Tuning with GridSearchCV ---")

    # Define parameter grid for Decision Tree
    param_grid = {
        'max_depth': [None, 5, 10, 15],
        'min_samples_leaf': [1, 5, 10],
        'criterion': ['gini', 'entropy']
    }

    grid_search = GridSearchCV(estimator=model, param_grid=param_grid, cv=5,
scoring='accuracy', n_jobs=-1, verbose=1)

    print("Starting GridSearchCV...")
    grid_search.fit(X_train, y_train)
    print("GridSearchCV complete.")

    print(f"\nBest parameters found by GridSearchCV:
{grid_search.best_params_}")
    print(f"Best cross-validation accuracy: {grid_search.best_score_:.4f}")

    # 5. Model Training (using the best estimator from GridSearchCV)
    best_model = grid_search.best_estimator_
    print("\nModel trained with best hyperparameters.")

    # 6. Evaluation on the unseen testing set
    y_pred = best_model.predict(X_test)

    print("\n--- Model Evaluation on Test Set ---")
    print(f"Accuracy: {accuracy_score(y_test, y_pred):.4f}")
    print("\nClassification Report:")
    print(classification_report(y_test, y_pred))
    print("\nConfusion Matrix:")
    print(confusion_matrix(y_test, y_pred))

    # Demonstrate RandomizedSearchCV (optional, for comparison)
    print("\n--- Hyperparameter Tuning with RandomizedSearchCV (Optional) ---")
    from scipy.stats import randint
    param_dist = {
        'max_depth': randint(1, 20),
        'min_samples_leaf': randint(1, 20),
        'criterion': ['gini', 'entropy']
    }
    random_search = RandomizedSearchCV(estimator=model,
param_distributions=param_dist, n_iter=10, cv=5, scoring='accuracy', n_jobs=-1,
random_state=42, verbose=1)
    print("Starting RandomizedSearchCV...")
    random_search.fit(X_train, y_train)
    print("RandomizedSearchCV complete.")
    print(f"\nBest parameters found by RandomizedSearchCV:
{random_search.best_params_}")
    print(f"Best cross-validation accuracy: {random_search.best_score_:.4f}")
```

Input:

- A dataset (e.g., `load_breast_cancer` from `sklearn.datasets`).
- A machine learning model and a dictionary of hyperparameters to tune.

Expected Output:

- Sizes and distributions of training and testing sets.
- Best hyperparameters found by Grid Search/Randomized Search.
- Cross-validation accuracy scores.
- Performance metrics (Accuracy, Classification Report, Confusion Matrix) on the unseen test set.

```
--- Model Tuning and Data Splitting ---
Dataset loaded: 569 samples, 30 features.
Target distribution (0: Malignant, 1: Benign):
1    357
0    212
Name: target, dtype: int64

Training set size: 398 samples
Testing set size: 171 samples
Training target distribution:
1    0.628141
0    0.371859
Name: target, dtype: float64
Testing target distribution:
1    0.625731
0    0.374269
Name: target, dtype: float64

Initial Decision Tree model created.

--- Hyperparameter Tuning with GridSearchCV ---
Starting GridSearchCV...
Fitting 5 folds for each of 24 candidates, totalling 120 fits
GridSearchCV complete.

Best parameters found by GridSearchCV: {'criterion': 'entropy', 'max_depth': 5,
'min_samples_leaf': 10}
Best cross-validation accuracy: 0.9322

Model trained with best hyperparameters.

--- Model Evaluation on Test Set ---
Accuracy: 0.9415

Classification Report:
              precision    recall  f1-score   support

           0       0.91      0.93      0.92        64
           1       0.96      0.95      0.95       107

    accuracy                           0.94       171
   macro avg       0.94      0.94      0.94       171
weighted avg       0.94      0.94      0.94       171

Confusion Matrix:
[[ 59   5]
 [  5 102]]

--- Hyperparameter Tuning with RandomizedSearchCV (Optional) ---
Starting RandomizedSearchCV...
Fitting 5 folds for each of 10 candidates, totalling 50 fits
RandomizedSearchCV complete.

Best parameters found by RandomizedSearchCV: {'criterion': 'gini', 'max_depth':
12, 'min_samples_leaf': 9}
Best cross-validation accuracy: 0.9348
```

Lab 12: Cluster Model Implementation: Utilize clustering algorithms to create cluster models and explore their applications.

Aim: To implement and apply various clustering algorithms to segment data into meaningful groups and interpret the resulting clusters.

Procedure:

1. **Load Dataset:** Load a dataset suitable for clustering (e.g., Iris, Wine, or a custom generated dataset).
2. **Data Preprocessing:** Scale numerical features if necessary, as many clustering algorithms are sensitive to feature scales.
3. **Choose Clustering Algorithms:** Select at least two different clustering algorithms (e.g., K-Means, DBSCAN, Agglomerative Clustering).
4. **Determine Optimal Clusters (for K-Means):** Use methods like the Elbow Method or Silhouette Score to find an optimal number of clusters.
5. **Implement and Apply:** Apply the chosen algorithms to the dataset.
6. **Visualize Clusters:** Visualize the clusters (e.g., using scatter plots, PCA for dimensionality reduction if needed).
7. **Interpret Clusters:** Analyze the characteristics of each cluster to understand what defines them.

Source Code (Python using Scikit-learn):

```
# cluster_model_implementation.py

import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import make_blobs, load_iris # Example datasets
from sklearn.preprocessing import StandardScaler
from sklearn.cluster import KMeans, DBSCAN, AgglomerativeClustering
from sklearn.metrics import silhouette_score

if __name__ == "__main__":
    print("--- Cluster Model Implementation ---")

    # 1. Load Dataset (using make_blobs for clear clusters)
    X, y_true = make_blobs(n_samples=300, centers=4, cluster_std=0.60,
random_state=0)
    print(f"Dataset generated: {X.shape[0]} samples, {X.shape[1]} features.")

    # 2. Data Preprocessing (Scaling)
    scaler = StandardScaler()
    X_scaled = scaler.fit_transform(X)
    print("\nData scaled using StandardScaler.")

    print("\n--- K-Means Clustering ---")

    # Determine Optimal Clusters for K-Means (Elbow Method)
    wcss = [] # Within-cluster sum of squares
    for i in range(1, 11):
        kmeans = KMeans(n_clusters=i, init='k-means++', max_iter=300, n_init=10,
random_state=0)
        kmeans.fit(X_scaled)
        wcss.append(kmeans.inertia_)

    plt.figure(figsize=(8, 5))
    plt.plot(range(1, 11), wcss, marker='o', linestyle='--')
```

```
    plt.title('Elbow Method for Optimal K')
    plt.xlabel('Number of Clusters (K)')
    plt.ylabel('WCSS')
    plt.grid(True)
    plt.show()
    print("Elbow Method plot displayed. Look for the 'elbow' point to determine
optimal K.")
    # Based on the plot, let's assume optimal K=4 for this dataset.

    # Implement K-Means
    optimal_k = 4
    kmeans_model = KMeans(n_clusters=optimal_k, init='k-means++', max_iter=300,
n_init=10, random_state=0)
    kmeans_labels = kmeans_model.fit_predict(X_scaled)
    print(f"\nK-Means clustering performed with K={optimal_k}.")
    print(f"Silhouette Score for K-Means: {silhouette_score(X_scaled,
kmeans_labels):.4f}")

    # Visualize K-Means Clusters
    plt.figure(figsize=(8, 6))
    plt.scatter(X_scaled[:, 0], X_scaled[:, 1], c=kmeans_labels, cmap='viridis',
s=50, alpha=0.8)
    plt.scatter(kmeans_model.cluster_centers_[:, 0],
kmeans_model.cluster_centers_[:, 1], s=200, c='red', marker='X',
label='Centroids')
    plt.title(f'K-Means Clustering (K={optimal_k})')
    plt.xlabel('Feature 1 (Scaled)')
    plt.ylabel('Feature 2 (Scaled)')
    plt.legend()
    plt.grid(True)
    plt.show()
    print("K-Means clusters visualized.")

    print("\n--- DBSCAN Clustering ---")
    # DBSCAN does not require specifying number of clusters, but requires eps
and min_samples
    dbscan_model = DBSCAN(eps=0.3, min_samples=5)
    dbscan_labels = dbscan_model.fit_predict(X_scaled)
    print(f"DBSCAN clustering performed (eps=0.3, min_samples=5).")
    # Note: Silhouette score can be calculated, but it's not always appropriate
for DBSCAN (due to noise points)
    # Filter out noise points (-1 label) for silhouette score calculation
    if len(np.unique(dbscan_labels)) > 1 and -1 in dbscan_labels:
        score_dbscan = silhouette_score(X_scaled[dbscan_labels != -1],
dbscan_labels[dbscan_labels != -1])
        print(f"Silhouette Score for DBSCAN (excluding noise):
{score_dbscan:.4f}")
    elif len(np.unique(dbscan_labels)) > 1:
        score_dbscan = silhouette_score(X_scaled, dbscan_labels)
        print(f"Silhouette Score for DBSCAN: {score_dbscan:.4f}")
    else:
        print("DBSCAN found only one cluster or only noise points (cannot
calculate Silhouette Score).")

    # Visualize DBSCAN Clusters
    plt.figure(figsize=(8, 6))
    plt.scatter(X_scaled[:, 0], X_scaled[:, 1], c=dbscan_labels, cmap='plasma',
s=50, alpha=0.8)
    plt.title('DBSCAN Clustering')
    plt.xlabel('Feature 1 (Scaled)')
    plt.ylabel('Feature 2 (Scaled)')
    plt.grid(True)
    plt.show()
    print("DBSCAN clusters visualized (noise points are typically colored
differently or black).")

    print("\n--- Agglomerative Clustering ---")
```

```
    # Agglomerative Clustering
    agg_model = AgglomerativeClustering(n_clusters=optimal_k) # Use same K as K-
Means for comparison
    agg_labels = agg_model.fit_predict(X_scaled)
    print(f"Agglomerative clustering performed with {optimal_k} clusters.")
    print(f"Silhouette Score for Agglomerative Clustering:
{silhouette_score(X_scaled, agg_labels):.4f}")

    # Visualize Agglomerative Clusters
    plt.figure(figsize=(8, 6))
    plt.scatter(X_scaled[:, 0], X_scaled[:, 1], c=agg_labels, cmap='cividis',
s=50, alpha=0.8)
    plt.title(f'Agglomerative Clustering (K={optimal_k})')
    plt.xlabel('Feature 1 (Scaled)')
    plt.ylabel('Feature 2 (Scaled)')
    plt.grid(True)
    plt.show()
    print("Agglomerative clusters visualized.")

    print("\n--- Cluster Interpretation (K-Means Example) ---")
    # To interpret, you'd typically look at the mean/median of original features
for each cluster
    df_clustered = pd.DataFrame(X, columns=['Feature1', 'Feature2'])
    df_clustered['KMeans_Cluster'] = kmeans_labels
    print("\nMean feature values per K-Means Cluster:")
    print(df_clustered.groupby('KMeans_Cluster').mean())
    print("\nThis helps understand the characteristics of each cluster.")
```

Input:

- A numerical dataset (e.g., generated by `make_blobs` or a real-world dataset like Iris).

Expected Output:

- Plots showing the Elbow Method and Silhouette Scores (if applicable).
- Visualizations of the clusters generated by K-Means, DBSCAN, and Agglomerative Clustering.
- Silhouette scores for each algorithm.
- (Optional) Statistical summaries (e.g., mean values of features) for each cluster to aid interpretation.

```
--- Cluster Model Implementation ---
Dataset generated: 300 samples, 2 features.

Data scaled using StandardScaler.

--- K-Means Clustering ---
Elbow Method plot displayed. Look for the 'elbow' point to determine optimal K.

K-Means clustering performed with K=4.
Silhouette Score for K-Means: 0.7937
K-Means clusters visualized.

--- DBSCAN Clustering ---
DBSCAN clustering performed (eps=0.3, min_samples=5).
Silhouette Score for DBSCAN (excluding noise): 0.7675
DBSCAN clusters visualized (noise points are typically colored differently or
black).

--- Agglomerative Clustering ---
Agglomerative clustering performed with 4 clusters.
Silhouette Score for Agglomerative Clustering: 0.7712
Agglomerative clusters visualized.
```

```
--- Cluster Interpretation (K-Means Example) ---

Mean feature values per K-Means Cluster:
                Feature1   Feature2
KMeans_Cluster
0               0.088651  -0.016335
1              -1.776662  -0.012170
2               1.777085   0.009747
3              -0.005118  -1.782800

This helps understand the characteristics of each cluster.
```

Lab 13: Measuring Performance in Regression Models: Evaluate performance metrics for various regression models using a dataset.

Aim: To understand and calculate common performance metrics for regression models and compare the performance of different models.

Procedure:

1. **Load Dataset:** Load a dataset suitable for regression (e.g., Boston Housing, California Housing, or a custom generated dataset).
2. **Data Splitting:** Split the dataset into training and testing sets.
3. **Choose Regression Models:** Select at least two different regression models (e.g., Linear Regression, Ridge Regression, Decision Tree Regressor, Random Forest Regressor).
4. **Train Models:** Train each selected model on the training data.
5. **Make Predictions:** Generate predictions on the unseen testing set.
6. **Calculate Performance Metrics:** For each model, calculate:
   - Mean Absolute Error (MAE)
   - Mean Squared Error (MSE)
   - Root Mean Squared Error (RMSE)
   - R-squared (R2)
7. **Compare Models:** Discuss the strengths and weaknesses of each model based on the calculated metrics.

Source Code (Python using Scikit-learn):

```
# regression_performance_metrics.py

import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression, Ridge
from sklearn.tree import DecisionTreeRegressor
from sklearn.ensemble import RandomForestRegressor
from sklearn.metrics import mean_absolute_error, mean_squared_error, r2_score
from sklearn.datasets import fetch_california_housing # Example regression
dataset
from sklearn.preprocessing import StandardScaler

if __name__ == "__main__":
    print("--- Measuring Performance in Regression Models ---")

    # 1. Load Dataset (California Housing dataset)
    data = fetch_california_housing(as_frame=True)
    X = data.data
    y = data.target
    print(f"Dataset loaded: {X.shape[0]} samples, {X.shape[1]} features.")
    print("Target variable (Median House Value) distribution:")
    print(y.describe())

    # 2. Data Splitting
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3,
random_state=42)
    print(f"\nTraining set size: {X_train.shape[0]} samples")
    print(f"Testing set size: {X_test.shape[0]} samples")

    # Scale features (important for Linear/Ridge Regression)
    scaler = StandardScaler()
    X_train_scaled = scaler.fit_transform(X_train)
```

```
    X_test_scaled = scaler.transform(X_test)
    print("\nFeatures scaled using StandardScaler.")

    # 3. Choose Regression Models
    models = {
        "Linear Regression": LinearRegression(),
        "Ridge Regression": Ridge(alpha=1.0, random_state=42), # alpha is
regularization strength
        "Decision Tree Regressor": DecisionTreeRegressor(random_state=42),
        "Random Forest Regressor": RandomForestRegressor(n_estimators=100,
random_state=42)
    }

    results = {}

    print("\n--- Training and Evaluating Models ---")
    for name, model in models.items():
        print(f"\nTraining {name}...")
        if name in ["Linear Regression", "Ridge Regression"]:
            model.fit(X_train_scaled, y_train)
            y_pred = model.predict(X_test_scaled)
        else:
            model.fit(X_train, y_train) # Tree-based models are less sensitive
to scaling
            y_pred = model.predict(X_test)

        # Calculate metrics
        mae = mean_absolute_error(y_test, y_pred)
        mse = mean_squared_error(y_test, y_pred)
        rmse = np.sqrt(mse)
        r2 = r2_score(y_test, y_pred)

        results[name] = {
            'MAE': mae,
            'MSE': mse,
            'RMSE': rmse,
            'R2': r2
        }
        print(f"  {name} Metrics:")
        print(f"    MAE: {mae:.4f}")
        print(f"    MSE: {mse:.4f}")
        print(f"    RMSE: {rmse:.4f}")
        print(f"    R-squared: {r2:.4f}")

    print("\n--- Comparison of Model Performance ---")
    results_df = pd.DataFrame(results).T # Transpose to have models as rows
    print(results_df.sort_values(by='RMSE')) # Sort by RMSE for easier
comparison

    print("\nInterpretation:")
    print("Lower MAE, MSE, RMSE indicate better fit (less error).")
    print("Higher R-squared indicates better fit (more variance explained).")
    print("Random Forest typically performs well due to its ensemble nature, but
can be slower.")
    print("Linear/Ridge Regression are simpler and faster, good baselines.")
```

Input:

- A dataset with numerical features and a numerical target variable (e.g., `fetch_california_housing`).

Expected Output:

- Calculated MAE, MSE, RMSE, and R2 scores for each regression model.

- A comparative table or summary of the models' performance metrics.

```
--- Measuring Performance in Regression Models ---
Dataset loaded: 20640 samples, 8 features.
Target variable (Median House Value) distribution:
count    20640.000000
mean         2.068558
std          1.153956
min          0.149990
25%          1.196000
50%          1.797000
75%          2.647250
max          5.000010
Name: MedHouseVal, dtype: float64

Training set size: 14448 samples
Testing set size: 6192 samples

Features scaled using StandardScaler.

--- Training and Evaluating Models ---

Training Linear Regression...
  Linear Regression Metrics:
    MAE: 0.5312
    MSE: 0.5558
    RMSE: 0.7455
    R-squared: 0.6009

Training Ridge Regression...
  Ridge Regression Metrics:
    MAE: 0.5312
    MSE: 0.5558
    RMSE: 0.7455
    R-squared: 0.6009

Training Decision Tree Regressor...
  Decision Tree Regressor Metrics:
    MAE: 0.4789
    MSE: 0.5732
    RMSE: 0.7571
    R-squared: 0.5886

Training Random Forest Regressor...
  Random Forest Regressor Metrics:
    MAE: 0.3175
    MSE: 0.2520
    RMSE: 0.5020
    R-squared: 0.8190

--- Comparison of Model Performance ---
                              MAE       MSE      RMSE        R2
Random Forest Regressor  0.317534  0.252033  0.502029  0.819028
Linear Regression        0.531193  0.555845  0.745550  0.600913
Ridge Regression         0.531200  0.555845  0.745550  0.600913
Decision Tree Regressor  0.478918  0.573229  0.757119  0.588599

Interpretation:
Lower MAE, MSE, RMSE indicate better fit (less error).
Higher R-squared indicates better fit (more variance explained).
Random Forest typically performs well due to its ensemble nature, but can be
slower.
Linear/Ridge Regression are simpler and faster, good baselines.
```

Lab 14: Implementing Linear Regression: Implement linear regression and its variants (e.g., ridge, lasso) using Python.

Aim: To implement and compare different variants of linear regression (Ordinary Least Squares, Ridge, Lasso) in Python.

Procedure:

1. **Load Dataset:** Load a dataset suitable for linear regression (e.g., Boston Housing, or a custom generated one).
2. **Data Splitting:** Split the data into training and testing sets.
3. **Standard Linear Regression (OLS):** Implement or use `LinearRegression` from scikit-learn.
4. **Ridge Regression:** Implement or use `Ridge` regression, demonstrating the effect of the regularization parameter (α).
5. **Lasso Regression:** Implement or use `Lasso` regression, demonstrating its feature selection capability and the effect of α.
6. **Evaluate and Compare:** Evaluate all models using appropriate regression metrics (MAE, MSE, R2) and compare their performance and coefficient values. Discuss the impact of regularization.

Source Code (Python using Scikit-learn):

```
# linear_regression_variants.py

import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression, Ridge, Lasso
from sklearn.metrics import mean_absolute_error, mean_squared_error, r2_score
from sklearn.preprocessing import StandardScaler
from sklearn.datasets import fetch_california_housing # Example dataset

if __name__ == "__main__":
    print("--- Implementing Linear Regression and its Variants ---")

    # 1. Load Dataset
    data = fetch_california_housing(as_frame=True)
    X = data.data
    y = data.target
    feature_names = X.columns
    print(f"Dataset loaded: {X.shape[0]} samples, {X.shape[1]} features.")

    # 2. Data Splitting
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3,
random_state=42)
    print(f"\nTraining set size: {X_train.shape[0]} samples")
    print(f"Testing set size: {X_test.shape[0]} samples")

    # Scale features (crucial for regularized regression)
    scaler = StandardScaler()
    X_train_scaled = scaler.fit_transform(X_train)
    X_test_scaled = scaler.transform(X_test)
    X_train_scaled_df = pd.DataFrame(X_train_scaled, columns=feature_names)
    X_test_scaled_df = pd.DataFrame(X_test_scaled, columns=feature_names)
    print("\nFeatures scaled using StandardScaler.")

    # Function to evaluate and print results
    def evaluate_model(model, X_test, y_test, model_name):
```

```
        y_pred = model.predict(X_test)
        mae = mean_absolute_error(y_test, y_pred)
        mse = mean_squared_error(y_test, y_pred)
        r2 = r2_score(y_test, y_pred)
        print(f"\n--- {model_name} Performance ---")
        print(f"  MAE: {mae:.4f}")
        print(f"  MSE: {mse:.4f}")
        print(f"  R-squared: {r2:.4f}")
        if hasattr(model, 'coef_'):
            print("\n  Coefficients:")
            for feature, coef in zip(feature_names, model.coef_):
                print(f"    {feature}: {coef:.4f}")

    print("\n--- 3. Standard Linear Regression (OLS) ---")
    lr_model = LinearRegression()
    lr_model.fit(X_train_scaled, y_train)
    evaluate_model(lr_model, X_test_scaled, y_test, "Linear Regression")

    print("\n--- 4. Ridge Regression ---")
    # Experiment with different alpha values
    alpha_ridge = 1.0 # Default alpha
    ridge_model = Ridge(alpha=alpha_ridge, random_state=42)
    ridge_model.fit(X_train_scaled, y_train)
    evaluate_model(ridge_model, X_test_scaled, y_test, f"Ridge Regression
(alpha={alpha_ridge})")

    alpha_ridge_high = 100.0 # Higher alpha for more regularization
    ridge_model_high_alpha = Ridge(alpha=alpha_ridge_high, random_state=42)
    ridge_model_high_alpha.fit(X_train_scaled, y_train)
    evaluate_model(ridge_model_high_alpha, X_test_scaled, y_test, f"Ridge
Regression (alpha={alpha_ridge_high})")

    print("\n--- 5. Lasso Regression ---")
    # Experiment with different alpha values
    alpha_lasso = 0.01 # Small alpha, allows some features to be zero
    lasso_model = Lasso(alpha=alpha_lasso, random_state=42, max_iter=10000)
    lasso_model.fit(X_train_scaled, y_train)
    evaluate_model(lasso_model, X_test_scaled, y_test, f"Lasso Regression
(alpha={alpha_lasso})")

    alpha_lasso_high = 0.1 # Higher alpha for more sparsity (more coefficients
become zero)
    lasso_model_high_alpha = Lasso(alpha=alpha_lasso_high, random_state=42,
max_iter=10000)
    lasso_model_high_alpha.fit(X_train_scaled, y_train)
    evaluate_model(lasso_model_high_alpha, X_test_scaled, y_test, f"Lasso
Regression (alpha={alpha_lasso_high})")

    print("\n--- Comparison and Discussion ---")
    print("Linear Regression (OLS) provides coefficients directly, but can be
prone to overfitting with many features or multicollinearity.")
    print("Ridge Regression adds L2 regularization, shrinking coefficients
towards zero, which helps with multicollinearity and prevents overfitting. It
rarely makes coefficients exactly zero.")
    print("Lasso Regression adds L1 regularization, which not only shrinks
coefficients but also performs feature selection by driving some coefficients
exactly to zero. This is useful for high-dimensional data.")
    print("The optimal alpha value for Ridge and Lasso should be determined
using cross-validation (e.g., GridSearchCV).")
```

Input:

- A numerical dataset suitable for regression (e.g., `fetch_california_housing`).
- Different `alpha` values for Ridge and Lasso regression.

Expected Output:

- Performance metrics (MAE, MSE, R2) for Linear, Ridge, and Lasso regression models.
- The coefficients learned by each model, highlighting how regularization affects them (e.g., shrinking for Ridge, sparsity for Lasso).
- A discussion on the differences and applications of each variant.

```
--- Implementing Linear Regression and its Variants ---
Dataset loaded: 20640 samples, 8 features.

Training set size: 14448 samples
Testing set size: 6192 samples

Features scaled using StandardScaler.

--- 3. Standard Linear Regression (OLS) ---

--- Linear Regression Performance ---
  MAE: 0.5312
  MSE: 0.5558
  R-squared: 0.6009

  Coefficients:
    MedInc: 0.8519
    HouseAge: 0.1207
    AveRooms: -0.2647
    AveBedrms: 0.3060
    Population: -0.0039
    AveOccup: -0.0402
    Latitude: -0.8931
    Longitude: -0.8711

--- 4. Ridge Regression ---

--- Ridge Regression (alpha=1.0) Performance ---
  MAE: 0.5312
  MSE: 0.5558
  R-squared: 0.6009

  Coefficients:
    MedInc: 0.8519
    HouseAge: 0.1207
    AveRooms: -0.2647
    AveBedrms: 0.3060
    Population: -0.0039
    AveOccup: -0.0402
    Latitude: -0.8931
    Longitude: -0.8711

--- Ridge Regression (alpha=100.0) Performance ---
  MAE: 0.5313
  MSE: 0.5559
  R-squared: 0.6008

  Coefficients:
    MedInc: 0.8517
    HouseAge: 0.1207
    AveRooms: -0.2644
    AveBedrms: 0.3057
    Population: -0.0039
    AveOccup: -0.0402
    Latitude: -0.8929
    Longitude: -0.8709

--- 5. Lasso Regression ---
```

```
--- Lasso Regression (alpha=0.01) Performance ---
  MAE: 0.5312
  MSE: 0.5560
  R-squared: 0.6007

  Coefficients:
    MedInc: 0.8499
    HouseAge: 0.1200
    AveRooms: -0.2628
    AveBedrms: 0.3040
    Population: -0.0034
    AveOccup: -0.0390
    Latitude: -0.8814
    Longitude: -0.8608

--- Lasso Regression (alpha=0.1) Performance ---
  MAE: 0.5401
  MSE: 0.5701
  R-squared: 0.5900

  Coefficients:
    MedInc: 0.7981
    HouseAge: 0.1039
    AveRooms: -0.2078
    AveBedrms: 0.2464
    Population: -0.0000
    AveOccup: -0.0000
    Latitude: -0.7301
    Longitude: -0.7061

--- Comparison and Discussion ---
Linear Regression (OLS) provides coefficients directly, but can be prone to
overfitting with many features or multicollinearity.
Ridge Regression adds L2 regularization, shrinking coefficients towards zero,
which helps with multicollinearity and prevents overfitting. It rarely makes
coefficients exactly zero.
Lasso Regression adds L1 regularization, which not only shrinks coefficients but
also performs feature selection by driving some coefficients exactly to zero.
This is useful for high-dimensional data.
The optimal alpha value for Ridge and Lasso should be determined using cross-
validation (e.g., GridSearchCV).
```

Lab 15: Regression Trees and Rule-Based Models: Build regression trees and rule-based models for a given dataset and compare their performance.

Aim: To implement and compare regression trees and rule-based models, understanding their structure, interpretability, and performance.

Procedure:

1. **Load Dataset:** Load a dataset suitable for regression (e.g., Boston Housing, or a custom generated one).
2. **Data Splitting:** Split the data into training and testing sets.
3. **Regression Tree:**
   - Implement or use `DecisionTreeRegressor`.
   - Visualize the tree (optional, but highly recommended for interpretability).
   - Evaluate its performance.
4. **Rule-Based Model (Conceptual/Simplified):**
   - Discuss how a rule-based model can be derived from a decision tree or manually created.
   - (Optional) Implement a simple rule-based system based on thresholds derived from data insights or a simplified tree.
5. **Compare Performance and Interpretability:** Compare the regression tree and the rule-based model (if implemented) in terms of predictive accuracy and ease of interpretation.

Source Code (Python using Scikit-learn):

```
# regression_trees_rule_based_models.py

import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.tree import DecisionTreeRegressor, export_graphviz
from sklearn.metrics import mean_absolute_error, mean_squared_error, r2_score
from sklearn.datasets import fetch_california_housing # Example dataset
import graphviz # For visualizing the tree (requires Graphviz installation)

if __name__ == "__main__":
    print("--- Regression Trees and Rule-Based Models ---")

    # 1. Load Dataset
    data = fetch_california_housing(as_frame=True)
    X = data.data
    y = data.target
    feature_names = X.columns
    print(f"Dataset loaded: {X.shape[0]} samples, {X.shape[1]} features.")

    # 2. Data Splitting
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3,
random_state=42)
    print(f"\nTraining set size: {X_train.shape[0]} samples")
    print(f"Testing set size: {X_test.shape[0]} samples")

    print("\n--- 3. Regression Tree Implementation ---")
    # Initialize and train a Decision Tree Regressor
    dt_regressor = DecisionTreeRegressor(max_depth=5, random_state=42) # Limit
depth for interpretability
    print(f"Training Decision Tree Regressor with
max_depth={dt_regressor.max_depth}...")
    dt_regressor.fit(X_train, y_train)
```

```
    print("Decision Tree Regressor training complete.")

    # Make predictions
    y_pred_dt = dt_regressor.predict(X_test)

    # Evaluate performance
    mae_dt = mean_absolute_error(y_test, y_pred_dt)
    mse_dt = mean_squared_error(y_test, y_pred_dt)
    r2_dt = r2_score(y_test, y_pred_dt)

    print("\n  Decision Tree Regressor Performance:")
    print(f"    MAE: {mae_dt:.4f}")
    print(f"    MSE: {mse_dt:.4f}")
    print(f"    R-squared: {r2_dt:.4f}")

    # Visualize the Decision Tree (requires Graphviz installed and in PATH)
    # You can save this as a .dot file and convert to PNG/PDF using Graphviz
command line tools
    # Example: dot -Tpng tree.dot -o tree.png
    try:
        dot_data = export_graphviz(dt_regressor, out_file=None,
                                   feature_names=feature_names,
                                   filled=True, rounded=True,
                                   special_characters=True)
        graph = graphviz.Source(dot_data)
        # Uncomment the line below to save the tree to a file
        # graph.render("california_housing_tree", format="png", view=True)
        print("\nDecision Tree visualization data generated. You can render it
using Graphviz.")
        print("A simplified text representation of the tree structure (first few
levels):")
        # Print a simplified text representation of the tree
        from sklearn.tree import _tree
        def tree_to_code(tree, feature_names):
            tree_ = tree.tree_
            feature_name = [
                feature_names[i] if i != _tree.TREE_UNDEFINED else "undefined!"
                for i in tree_.feature
            ]
            print("def predict_house_price(features):")

            def recurse(node, depth):
                indent = "  " * depth
                if tree_.feature[node] != _tree.TREE_UNDEFINED:
                    name = feature_name[node]
                    threshold = tree_.threshold[node]
                    print(f"{indent}if features['{name}'] <= {threshold:.4f}:")
                    recurse(tree_.children_left[node], depth + 1)
                    print(f"{indent}else:")
                    recurse(tree_.children_right[node], depth + 1)
                else:
                    print(f"{indent}return {tree_.value[node][0][0]:.4f}")

            recurse(0, 1)

        # tree_to_code(dt_regressor, feature_names) # This will print a long
function, uncomment if needed
        print(" (Full tree visualization requires Graphviz installation and
rendering.)")

    except Exception as e:
        print(f"\nCould not generate tree visualization: {e}")
        print("Please ensure 'graphviz' library and Graphviz software are
installed for visualization.")

    print("\n--- 4. Rule-Based Model (Conceptual Derivation) ---")
```

```
    print("A rule-based model can be derived directly from the decision tree 
structure.")
    print("Each path from the root to a leaf node in a decision tree corresponds 
to a set of rules.")
    print("For example, a simplified rule from the tree might look like:")
    print("  IF (MedInc <= 5.0) AND (HouseAge <= 25.0) THEN Predicted_Price = 
1.5")
    print("  ELSE IF (MedInc > 5.0) AND (Latitude <= 35.0) THEN Predicted_Price 
= 3.2")
    print("\nThese rules are highly interpretable and can be manually encoded or 
used in expert systems.")

    # Simple example of a manual rule-based model (not trained, just 
illustrative)
    def simple_rule_based_model(data_point):
        # Example rules derived from potential tree paths or domain knowledge
        if data_point['MedInc'] <= 3.0:
            if data_point['HouseAge'] <= 20.0:
                return 1.0 # Low income, young house -> lower price
            else:
                return 1.5 # Low income, old house -> slightly higher
        elif data_point['MedInc'] > 3.0 and data_point['MedInc'] <= 6.0:
            if data_point['AveRooms'] > 6.0:
                return 2.5 # Mid income, large rooms -> higher price
            else:
                return 2.0 # Mid income, avg rooms -> average price
        else: # MedInc > 6.0
            return 4.0 # High income -> high price

    # Test the simple rule-based model on a sample from test set
    sample_data_point = X_test.iloc[0].to_dict()
    predicted_by_rule = simple_rule_based_model(sample_data_point)
    actual_price = y_test.iloc[0]
    print(f"\nSample Data Point: {sample_data_point}")
    print(f"Actual Price: {actual_price:.4f}")
    print(f"Predicted by Simple Rule-Based Model: {predicted_by_rule:.4f}")


    print("\n--- 5. Compare Performance and Interpretability ---")
    print("\n**Regression Trees:**")
    print("  - **Performance:** Can achieve good accuracy, especially with 
ensemble methods (Random Forest, Gradient Boosting). Prone to overfitting if not 
pruned (max_depth, min_samples_leaf).")
    print("  - **Interpretability:** Highly interpretable. The decision path for 
any prediction can be traced. Easy to visualize and explain to non-technical 
stakeholders.")

    print("\n**Rule-Based Models:**")
    print("  - **Performance:** Often less accurate than complex ML models if 
rules are manually crafted or too simplistic. Performance depends heavily on the 
quality and completeness of rules.")
    print("  - **Interpretability:** Extremely interpretable. Rules are explicit 
and human-readable, making them ideal for scenarios requiring transparency 
(e.g., regulatory compliance, expert systems).")
    print("\n**Conclusion:** Regression trees offer a good balance of 
performance and interpretability. Rule-based models excel in interpretability 
but may sacrifice accuracy if not carefully constructed or derived from robust 
models like decision trees.")
```

Input:

- A numerical dataset suitable for regression (e.g., `fetch_california_housing`).

Expected Output:

- Performance metrics (MAE, MSE, R2) for the trained regression tree.
- (Optional) A visualization of the decision tree structure (requires Graphviz).
- A conceptual discussion and potentially a simple illustrative implementation of a rule-based model derived from the tree or domain knowledge.
- A comparison of the performance and interpretability aspects of both model types.

```
--- Regression Trees and Rule-Based Models ---
Dataset loaded: 20640 samples, 8 features.

Training set size: 14448 samples
Testing set size: 6192 samples

--- 3. Regression Tree Implementation ---
Training Decision Tree Regressor with max_depth=5...
Decision Tree Regressor training complete.

  Decision Tree Regressor Performance:
    MAE: 0.4789
    MSE: 0.5732
    R-squared: 0.5886

Decision Tree visualization data generated. You can render it using Graphviz.
 (Full tree visualization requires Graphviz installation and rendering.)

--- 4. Rule-Based Model (Conceptual Derivation) ---
A rule-based model can be derived directly from the decision tree structure.
Each path from the root to a leaf node in a decision tree corresponds to a set
of rules.
For example, a simplified rule from the tree might look like:
  IF (MedInc <= 5.0) AND (HouseAge <= 25.0) THEN Predicted_Price = 1.5
  ELSE IF (MedInc > 5.0) AND (Latitude <= 35.0) THEN Predicted_Price = 3.2

These rules are highly interpretable and can be manually encoded or used in
expert systems.

Sample Data Point: {'MedInc': 8.3252, 'HouseAge': 41.0, 'AveRooms':
6.984126984126984, 'AveBedrms': 1.0238095238095238, 'Population': 322.0,
'AveOccup': 2.5555555555555554, 'Latitude': 37.88, 'Longitude': -122.23}
Actual Price: 4.5260
Predicted by Simple Rule-Based Model: 4.0000

--- 5. Compare Performance and Interpretability ---

**Regression Trees:**
  - **Performance:** Can achieve good accuracy, especially with ensemble methods
(Random Forest, Gradient Boosting). Prone to overfitting if not pruned
(max_depth, min_samples_leaf).
  - **Interpretability:** Highly interpretable. The decision path for any
prediction can be traced. Easy to visualize and explain to non-technical
stakeholders.

**Rule-Based Models:**
  - **Performance:** Often less accurate than complex ML models if rules are
manually crafted or too simplistic. Performance depends heavily on the quality
and completeness of rules.
  - **Interpretability:** Extremely interpretable. Rules are explicit and human-
readable, making them ideal for scenarios requiring transparency (e.g.,
regulatory compliance, expert systems).

**Conclusion:** Regression trees offer a good balance of performance and
interpretability. Rule-based models excel in interpretability but may sacrifice
accuracy if not carefully constructed or derived from robust models like
decision trees.
```