assignment

April 15, 2025

Question 1. Explain the differences and similarities between Python and Excel. Provide real-life scenarios where Python is preferred over Excel and justify why.

1 Python vs Excel: Differences, Similarities, and Use Cases

1.1 Similarities

- **Data Analysis**: Both tools can analyze and manipulate data
- **Visualization**: Both can create charts and graphs
- **Calculations**: Both can perform complex calculations and data transformations
- **Automation**: Both support automation of repetitive tasks

1.2 Key Differences

- **Programming vs Spreadsheet**: Python is a full programming language; Excel is primarily a spreadsheet tool
- **Scalability**: Python can handle much larger datasets (millions+ rows) than Excel (~1M row limit)
- **Reproducibility**: Python code is more reproducible and self-documenting than Excel formulas
- **Flexibility**: Python supports complex algorithms, machine learning, web scraping, etc.
- **Learning Curve**: Excel is generally easier to learn initially; Python requires programming knowledge

1.3 When Python is Preferred (with Justifications)

1. **Big Data Processing**
   - *Scenario*: Analyzing customer behavior data for an e-commerce site (10M+ records)
   - *Justification*: Python libraries like pandas can handle data that would crash Excel, and distributed computing frameworks like Spark can process petabytes
2. **Complex Data Science**
   - *Scenario*: Predicting customer churn using machine learning
   - *Justification*: Python's scikit-learn, TensorFlow, and PyTorch enable sophisticated ML models that Excel cannot support
3. **Automated Reporting**
   - *Scenario*: Daily financial reports that pull from multiple databases
   - *Justification*: Python scripts can be scheduled, access multiple data sources, perform complex transformations, and generate reports automatically
4. **Web Scraping & API Integration**

- *Scenario*: Collecting competitor pricing data from websites
- *Justification*: Python's libraries (requests, BeautifulSoup) can extract web data and interact with APIs, which is impossible in Excel alone

5. **Version Control & Collaboration**
   - *Scenario*: Multiple data analysts working on the same analytical pipeline
   - *Justification*: Python code can be version-controlled with Git, enabling better collaboration and tracking changes

Question 2. Write a Python script that imports data from a CSV file, processes it using Pandas, and saves the filtered data to a new Excel file.

[ ]:
```python
import pandas as pd

# Import data from CSV file
try:
    df = pd.read_csv('data.csv')
except FileNotFoundError:
    print("Error: 'data.csv' not found. Please ensure the file exists in the␣
 ↪correct directory.")

    data = {'Date': pd.to_datetime(['2023-01-01', '2023-01-02', None,␣
 ↪'2023-01-04', '2023-01-05']),
            'Product': ['A', 'B', 'A', 'C', 'B'],
            'Region': ['North', 'South', 'North', 'West', 'East'],
            'Sales': [1200, 1600, 800, 1700, 1400],
            'Units': [50, 70, None, 80, 60]}
    df = pd.DataFrame(data)
    print("\nUsing sample data as 'data.csv' was not found:")


# Display the first few rows to understand the data
print("Original data:")
print(df.head())

# Process the data
# 1. Drop any rows with missing values
df_cleaned = df.dropna()

# 2. Filter data (example: keep only rows where 'Sales' > 1500)
df_filtered = df_cleaned[df_cleaned['Sales'] > 1500].copy()

# 3. Perform calculations (example: create a new column 'Doubled_Value')
df_filtered.loc[:, 'Doubled_Value'] = df_filtered['Sales'] * 2

# Display summary statistics
print("\nSummary of processed data:")
print(df_filtered.describe())
print(f"\nShape of original data: {df.shape}")
```

```python
print(f"Shape of processed data: {df_filtered.shape}")

# Save the processed data to an Excel file
output_file = 'processed_data.xlsx'
df_filtered.to_excel(output_file, index=False)
print(f"\nProcessed data saved to {output_file}")
```

```
Original data:
         Date   Product Region  Sales  Units
0  2023-01-01  Widget C   East   1363     76
1  2023-01-02  Widget D  South   1242     28
2  2023-01-03  Widget A  South   1764     29
3  2023-01-04  Widget C   West   1063     80
4  2023-01-05  Widget C  South   1520     61

Summary of processed data:
             Sales      Units  Doubled_Value
count    46.000000  46.000000      46.000000
mean   1782.086957  49.152174    3564.173913
std     136.433431  22.639167     272.866862
min    1512.000000  12.000000    3024.000000
25%    1685.750000  30.250000    3371.500000
50%    1774.000000  45.500000    3548.000000
75%    1899.250000  62.500000    3798.500000
max    1996.000000  98.000000    3992.000000

Shape of original data: (100, 5)
Shape of processed data: (46, 6)

Processed data saved to processed_data.xlsx
```

Question 3. Describe the process of installing Anaconda and launching Jupyter Notebook. Explain how to create, save, and run Python code using Jupyter Notebookary.

1.4 Installing Anaconda and Launching Jupyter Notebook

Step 1: Download Anaconda
- Visit the official Anaconda website and choose the installer for your operating system.
- Click “Download” and save the installer.

Step 2: Install Anaconda
- Run the installer file.
- On Windows, select “Add Anaconda to my PATH”and “Register Anaconda as my default Python.”
- On macOS and Linux, follow the default settings.

Step 3: Verify Installation
- Open a terminal or command prompt.
- Type: `bash` `conda --version` You should see the Anaconda version number.

Step 4: Launch Jupyter Notebook

- In a terminal or command prompt, type: `bash jupyter notebook` - Your browser will open the Jupyter Notebook dashboard.

Step 5: Create, Save, and Run Python Code
1. In the Jupyter dashboard, click "New" -> "Python 3" to create a new notebook.
2. Type your Python code in a cell, then press Shift + Enter to run it. 3. Save your notebook by clicking the "Save" icon or pressing Ctrl + S (Cmd + S on macOS).
4. Click the notebook name at the top to rename it.

1.4.1 Example: Simple Python Code

```
# Print a custom message
print("Hello, Universe!")

# Perform basic arithmetic
x = 15
y = 25
total = x + y
print("Sum of x and y is:", total)

# Loop example
for i in range(3):
        print("This is loop iteration:", i)

# Function example
def greet(name):
        return f"Greetings, {name}!"

print(greet("Jupyter"))
```

Question 3. Illustrate the differences between various Python data types (int, float, str, list, tuple, dict, etc.) with examples. Discuss the importance of understanding data types while handling large datasets.

Question 4. Define and explain conditional statements in Python with multiple use-case examples. Include examples of nested if-else and practical scenarios where nested conditions are necessary.

[22]:
```
# Illustrating differences between common Python data types

# Integer (int)
a = 10
print(f"Type of a ({a}):", type(a))

# Float (float)
b = 3.14
print(f"Type of b ({b}):", type(b))

# String (str)
```

```python
c = "Hello, World!"
print(f"Type of c ('{c}'):", type(c))

# List (list): Mutable, ordered collection
d = [1, 2, 3, "apple", 4.5]
print(f"Type of d ({d}):", type(d))

# Tuple (tuple): Immutable, ordered collection
e = (1, 2, 3, "banana", 6.7)
print(f"Type of e ({e}):", type(e))

# Dictionary (dict): Key-value pairs, mutable, unordered (insertion order␣
 ↪preserved in Python 3.7+)
f = {"name": "Alice", "age": 25, "city": "Delhi"}
print(f"Type of f ({f}):", type(f))

# Set (set): Unordered, unique elements
g = {1, 2, 3, 2, 1}
print(f"Type of g ({g}):", type(g))

print("\n--- Data Type Differences ---")
print("""
- int: Whole numbers, used for counting, indexing, etc.
- float: Numbers with decimals, used for measurements, calculations.
- str: Textual data, used for names, labels, etc.
- list: Ordered, mutable collection; can hold mixed types.
- tuple: Ordered, immutable collection; often used for fixed data.
- dict: Key-value mapping; fast lookup, flexible structure.
- set: Unordered, unique elements; useful for membership tests.
""")

print("--- Importance of Data Types in Large Datasets ---")
print("""
- Correct data types save memory and improve performance (e.g., using int8␣
 ↪instead of int64 for small numbers).
- Ensures accurate computations (e.g., float division vs integer division).
- Prevents errors (e.g., cannot sum strings and numbers).
- Enables efficient filtering, grouping, and aggregation in pandas DataFrames.
- Data type mismatches can cause bugs, slowdowns, or incorrect results in␣
 ↪analysis and machine learning.
""")

# --- Conditional Statements in Python ---

print("\n--- Conditional Statements in Python ---")

# Basic if-else
```

```python
x = 15
if x > 10:
    print(f"{x} is greater than 10")
else:
    print(f"{x} is not greater than 10")

# if-elif-else
score = 82
if score >= 90:
    grade = 'A'
elif score >= 75:
    grade = 'B'
elif score >= 60:
    grade = 'C'
else:
    grade = 'D'
print(f"Score: {score}, Grade: {grade}")

# Nested if-else (practical scenario: eligibility check)
age = 20
has_id = True
if age >= 18:
    if has_id:
        print("Eligible to vote.")
    else:
        print("Not eligible: ID required.")
else:
    print("Not eligible: Underage.")

# Real-world nested condition: Loan approval
income = 50000
credit_score = 720
if income > 40000:
    if credit_score > 700:
        print("Loan Approved")
    else:
        print("Loan Denied: Low credit score")
else:
    print("Loan Denied: Low income")

print("""
--- Explanation ---
- 'if' checks a condition; 'elif' checks additional conditions if previous ones␣
 ↪are False; 'else' runs if none are True.
- Nested if-else is used when decisions depend on multiple, hierarchical␣
 ↪conditions (e.g., eligibility, multi-step validation).
```

```
- In data analysis, conditional logic is used for filtering, categorizing, and␣
 ↪handling missing or special values.
""")
```

```
Type of a (10): <class 'int'>
Type of b (3.14): <class 'float'>
Type of c ('Hello, World!'): <class 'str'>
Type of d ([1, 2, 3, 'apple', 4.5]): <class 'list'>
Type of e ((1, 2, 3, 'banana', 6.7)): <class 'tuple'>
Type of f ({'name': 'Alice', 'age': 25, 'city': 'Delhi'}): <class 'dict'>
Type of g ({1, 2, 3}): <class 'set'>

--- Data Type Differences ---

- int: Whole numbers, used for counting, indexing, etc.
- float: Numbers with decimals, used for measurements, calculations.
- str: Textual data, used for names, labels, etc.
- list: Ordered, mutable collection; can hold mixed types.
- tuple: Ordered, immutable collection; often used for fixed data.
- dict: Key-value mapping; fast lookup, flexible structure.
- set: Unordered, unique elements; useful for membership tests.

--- Importance of Data Types in Large Datasets ---

- Correct data types save memory and improve performance (e.g., using int8
instead of int64 for small numbers).
- Ensures accurate computations (e.g., float division vs integer division).
- Prevents errors (e.g., cannot sum strings and numbers).
- Enables efficient filtering, grouping, and aggregation in pandas DataFrames.
- Data type mismatches can cause bugs, slowdowns, or incorrect results in
analysis and machine learning.


--- Conditional Statements in Python ---
15 is greater than 10
Score: 82, Grade: B
Eligible to vote.
Loan Approved

--- Explanation ---
- 'if' checks a condition; 'elif' checks additional conditions if previous ones
are False; 'else' runs if none are True.
- Nested if-else is used when decisions depend on multiple, hierarchical
conditions (e.g., eligibility, multi-step validation).
- In data analysis, conditional logic is used for filtering, categorizing, and
handling missing or special values.
```

2 SECTION 2

Question 1. Write a Python script that reads data from an Excel file, performs data cleaning (removes duplicates and fills missing values), and outputs a summary of the dat.

[21]:
```
# Create sample Indian data (including salary) and save it to an Excel file
df_sample = pd.DataFrame({
    'Name': ['Karthik', 'Ananya', 'Meena', 'Meena', 'Karthik'],
    'Age': [30, 24, None, 24, 30],
    'Salary': [50000, 60000, 45000, 45000, 50000],
    'City': ['Bengaluru', 'Mumbai', 'Delhi', 'Delhi', 'Bengaluru']
})
df_sample.to_excel("indian_data.xlsx", index=False)

# Read data from the Excel file
df_read = pd.read_excel("indian_data.xlsx")

# Remove duplicates and create a copy to avoid SettingWithCopyWarning
df_cleaned_data = df_read.drop_duplicates().copy()

# Convert 'Age' to string (object) before filling with non-numeric value
df_cleaned_data['Age'] = df_cleaned_data['Age'].astype(str).replace({'nan':␣
 ↪'Unknown'})

# Output summary
print(df_cleaned_data.info())
print(df_cleaned_data.describe(include='all'))
```

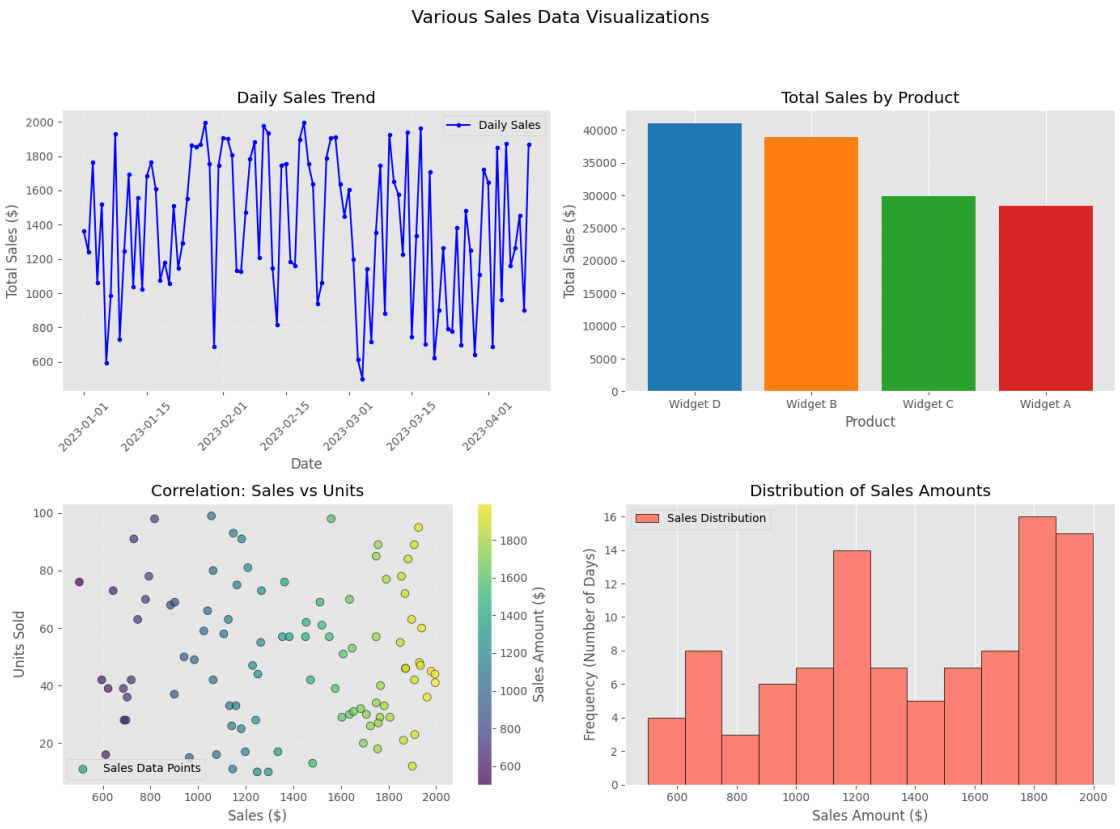
```
<class 'pandas.core.frame.DataFrame'>
Index: 4 entries, 0 to 3
Data columns (total 4 columns):
 #   Column  Non-Null Count  Dtype
---  ------  --------------  -----
 0   Name    4 non-null      object
 1   Age     4 non-null      object
 2   Salary  4 non-null      int64
 3   City    4 non-null      object
dtypes: int64(1), object(3)
memory usage: 160.0+ bytes
None
          Name   Age        Salary   City
count        4     4      4.000000      4
unique       3     3           NaN      3
top     Meena  24.0           NaN  Delhi
freq         2     2           NaN      2
mean       NaN   NaN  50000.000000    NaN
std        NaN   NaN   7071.067812    NaN
min        NaN   NaN  45000.000000    NaN
```

```
25%        NaN    NaN  45000.000000     NaN
50%        NaN    NaN  47500.000000     NaN
75%        NaN    NaN  52500.000000     NaN
max        NaN    NaN  60000.000000     NaN
```

Question 2. Demonstrate how to create various types of visualizations (line plots, bar charts, scatter plots, and histograms) using Matplotlib. Customize the visualizations with labels, legends, and colors .

[23]:
```
import matplotlib.pyplot as plt
import pandas as pd

df_sorted = df.sort_values('Date')

fig, axes = plt.subplots(nrows=2, ncols=2, figsize=(14, 10))
fig.suptitle('Various Sales Data Visualizations', fontsize=16, y=1.02)

# --- Plot 1: Line Plot (Top-Left) ---
# Visualize sales trend over time
axes[0, 0].plot(df_sorted['Date'], df_sorted['Sales'], label='Daily Sales',␣
 ↪color='blue', marker='.', linestyle='-')
axes[0, 0].set_title('Daily Sales Trend')
axes[0, 0].set_xlabel('Date')
axes[0, 0].set_ylabel('Total Sales ($)')
axes[0, 0].tick_params(axis='x', rotation=45)
axes[0, 0].legend()
axes[0, 0].grid(True, linestyle=':', alpha=0.7)

# --- Plot 2: Bar Chart (Top-Right) ---
# Visualize total sales per product
product_sales = df.groupby('Product')['Sales'].sum().
 ↪sort_values(ascending=False)
colors_bar = ['#1f77b4', '#ff7f0e', '#2ca02c', '#d62728'] # Example colors
axes[0, 1].bar(product_sales.index, product_sales.values, color=colors_bar,␣
 ↪label=product_sales.index)
axes[0, 1].set_title('Total Sales by Product')
axes[0, 1].set_xlabel('Product')
axes[0, 1].set_ylabel('Total Sales ($)')
# axes[0, 1].legend(title='Products') # Legend might be redundant if x-axis␣
 ↪labels are clear
axes[0, 1].grid(axis='y', linestyle=':', alpha=0.7)

# --- Plot 3: Scatter Plot (Bottom-Left) ---
# Visualize relationship between Sales and Units sold
scatter = axes[1, 0].scatter(df['Sales'], df['Units'], c=df['Sales'],␣
 ↪cmap='viridis', alpha=0.7, edgecolors='k', s=50, label='Sales Data Points')
axes[1, 0].set_title('Correlation: Sales vs Units')
axes[1, 0].set_xlabel('Sales ($)')
```

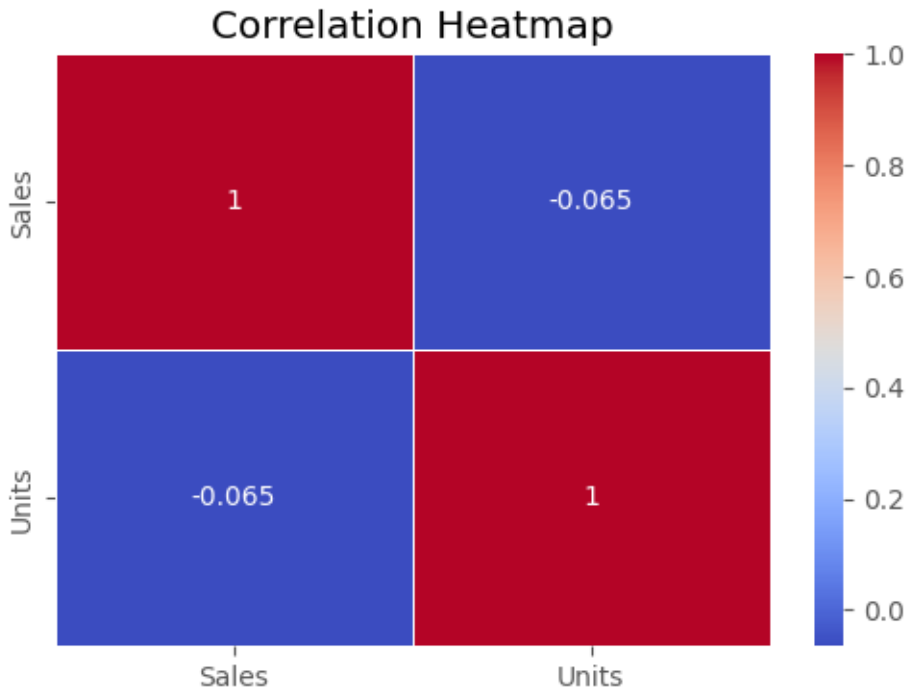
```
axes[1, 0].set_ylabel('Units Sold')
fig.colorbar(scatter, ax=axes[1, 0], label='Sales Amount ($)')
axes[1, 0].grid(True, linestyle=':', alpha=0.7)
axes[1, 0].legend()

# --- Plot 4: Histogram (Bottom-Right) ---
# Visualize the distribution of Sales values
axes[1, 1].hist(df['Sales'], bins=12, color='salmon', edgecolor='black',␣
 ↪label='Sales Distribution')
axes[1, 1].set_title('Distribution of Sales Amounts')
axes[1, 1].set_xlabel('Sales Amount ($)')
axes[1, 1].set_ylabel('Frequency (Number of Days)')
axes[1, 1].legend()
axes[1, 1].grid(axis='y', linestyle=':', alpha=0.7)

plt.tight_layout(rect=[0, 0, 1, 0.97])

# Display the plots
plt.show()
```



Question 3. Write a Python program that generates a heatmap of correlation between different

variables in a given dataset using Seaborn. Explain how to interpret the heatmp..

[26]:
```
import seaborn as sns

import matplotlib.pyplot as plt

corr_matrix = df[['Sales', 'Units']].corr()

# Create a heatmap using Seaborn
plt.figure(figsize=(6, 4))
sns.heatmap(corr_matrix, annot=True, cmap='coolwarm', linewidths=0.5)
plt.title("Correlation Heatmap")
plt.show()

# Interpretation:
# - Each cell shows how strongly two variables are correlated (closer to 1 or␣
 ↪-1 means higher correlation).
# - Positive values (red) indicate that when one variable increases, the other␣
 ↪also tends to increase.
# - Negative values (blue) indicate an inverse relationship.
# - Values near 0 suggest little to no linear correlation between the variables.
```



3 SECTION 3

Question 1. Describe the key steps involved in building a Machine Learning model using Scikit-learn. Explain how to split data into training and testing sets and evaluate the model. t.

from sklearn.model_selection import train_test_split

3.1 Building a Machine Learning Model with Scikit-learn

Building a machine learning model using the Scikit-learn library typically involves the following key steps:

1. **Import Libraries**: Load necessary modules from Scikit-learn (e.g., `model_selection`, specific estimators like `LinearRegression` or `LogisticRegression`, `metrics`) and data manipulation libraries like Pandas.
2. **Load and Prepare Data**:
   - Load the dataset, often into a Pandas DataFrame.
   - Perform **data cleaning**: Handle missing values (imputation or removal), correct errors, and deal with outliers.
   - **Feature Engineering**: Create new features from existing ones if needed.
   - **Feature Encoding**: Convert categorical features into numerical representations (e.g., One-Hot Encoding, Label Encoding).
   - **Feature Scaling**: Standardize or normalize numerical features, especially for algorithms sensitive to feature scales (e.g., SVM, KNN, Linear Regression with regularization).
3. **Select Features and Target**: Define which columns are the input features (X) and which column is the target variable (y) to predict.
4. **Split Data into Training and Testing Sets**:
   - This is crucial to evaluate the model's performance on unseen data.
   - Use `sklearn.model_selection.train_test_split`.
   - Typically, 70-80% of the data is used for training (`X_train`, `y_train`) and the remaining 20-30% for testing (`X_test`, `y_test`).
   - Setting a `random_state` ensures the split is the same each time the code runs, making results reproducible.

```
# Example using train_test_split
# Assuming X contains features and y contains the target variable
# X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42
```

5. **Choose and Instantiate a Model**: Select an appropriate algorithm based on the problem type (regression, classification) and data characteristics. Instantiate the model class.

```
# Example for Linear Regression
# from sklearn.linear_model import LinearRegression
# model = LinearRegression()

# Example for Logistic Regression (Classification)
# from sklearn.linear_model import LogisticRegression
# model = LogisticRegression()
```

6. **Train the Model**: Fit the model to the training data. The model learns the relationship between `X_train` and `y_train`.

   ```
   # model.fit(X_train, y_train)
   ```

7. **Make Predictions**: Use the trained model to make predictions on the test set (`X_test`).

   ```
   # y_pred = model.predict(X_test)
   ```

8. **Evaluate the Model**: Compare the predictions (`y_pred`) with the actual values (`y_test`) using appropriate evaluation metrics from `sklearn.metrics`.

   - **Regression Metrics**: Mean Absolute Error (MAE), Mean Squared Error (MSE), R-squared (R^2).
   - **Classification Metrics**: Accuracy, Precision, Recall, F1-score, Confusion Matrix, ROC AUC score.

   ```
   # Example for Regression Evaluation
   # from sklearn.metrics import mean_squared_error, r2_score
   # mse = mean_squared_error(y_test, y_pred)
   # r2 = r2_score(y_test, y_pred)
   # print(f"MSE: {mse}, R-squared: {r2}")

   # Example for Classification Evaluation
   # from sklearn.metrics import accuracy_score, classification_report
   # accuracy = accuracy_score(y_test, y_pred)
   # report = classification_report(y_test, y_pred)
   # print(f"Accuracy: {accuracy}\nClassification Report:\n{report}")
   ```

9. **Tune Hyperparameters (Optional)**: Optimize model performance by tuning its hyperparameters using techniques like `GridSearchCV` or `RandomizedSearchCV`.

10. **Finalize Model**: Once satisfied with the performance, the model can be saved or deployed.

[31]:
```
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score, classification_report

# Load iris dataset
iris = load_iris()
X, y = iris.data, iris.target

# Split the dataset
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3,␣
 ↪random_state=42)

# Train a logistic regression model
model = LogisticRegression(max_iter=200)
model.fit(X_train, y_train)
```

```
# Evaluate
y_pred = model.predict(X_test)
accuracy = accuracy_score(y_test, y_pred)

report = classification_report(y_test, y_pred)

print(f"Accuracy: {accuracy}")
print("Classification Report:")
print(report)
```

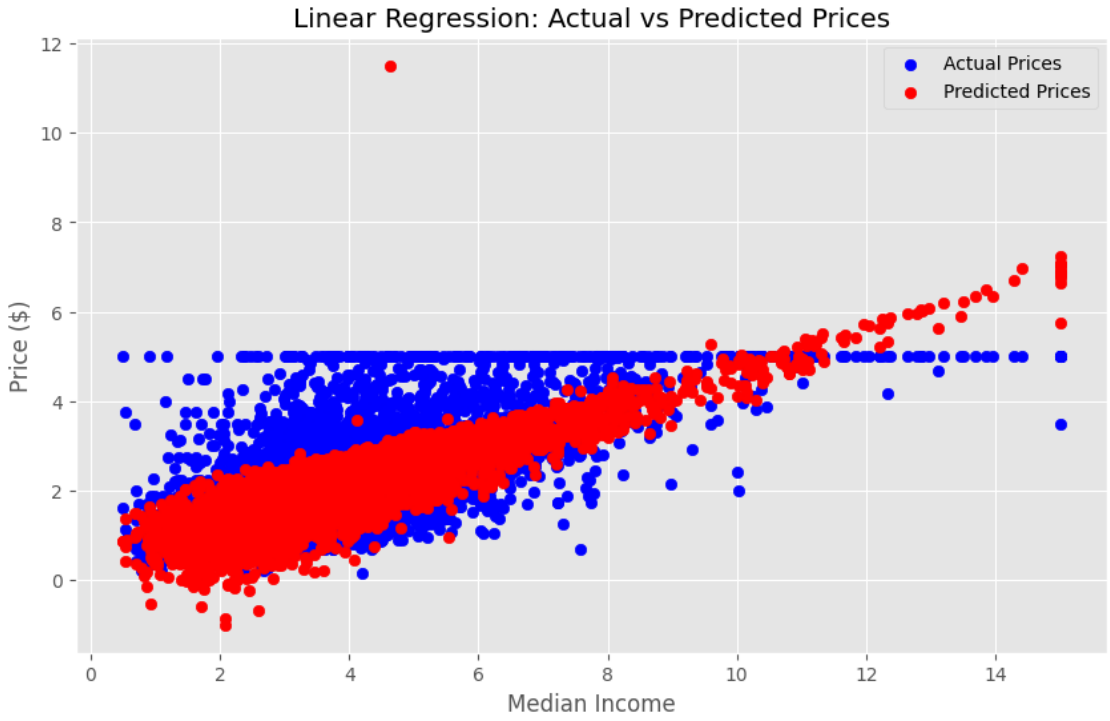
```
Accuracy: 1.0
Classification Report:
              precision    recall  f1-score   support

           0       1.00      1.00      1.00        19
           1       1.00      1.00      1.00        13
           2       1.00      1.00      1.00        13

    accuracy                           1.00        45
   macro avg       1.00      1.00      1.00        45
weighted avg       1.00      1.00      1.00        45
```

Question 2. Demonstrate the implementation of a Linear Regression model using Scikit-learn to predict house prices based on a given dataset. Include evaluation of model performance using appropriate metrics.t.

[39]:
```
from sklearn.datasets import fetch_california_housing
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error, r2_score
import matplotlib.pyplot as plt

# Load the California housing dataset
housing = fetch_california_housing()
X, y = housing.data, housing.target

# Split into training and test sets
X_train, X_test, y_train, y_test = train_test_split(
    X,
    y,
    test_size=0.3,
    random_state=42
)

# Initialize and train the model
lin_reg = LinearRegression()
lin_reg.fit(X_train, y_train)
```

```python
# Predict
y_pred = lin_reg.predict(X_test)

# Evaluate
mse = mean_squared_error(y_test, y_pred)
r2 = r2_score(y_test, y_pred)
print("Mean Squared Error:", mse)
print("R-squared Score:", r2)

# Visualization: Actual vs Predicted (using the first feature of X_test)
plt.figure(figsize=(10, 6))
plt.scatter(X_test[:, 0], y_test, color='blue', label='Actual Prices')
plt.scatter(X_test[:, 0], y_pred, color='red', label='Predicted Prices')
plt.xlabel('Median Income')
plt.ylabel('Price ($)')
plt.title('Linear Regression: Actual vs Predicted Prices')
plt.legend()
plt.show()
```

```
Mean Squared Error: 0.5305677824766757
R-squared Score: 0.595770232606166
```



Question 3. Write a Python script that builds a Logistic Regression model to classify email spam

and evaluates its performance using accuracy, confusion matrix, and classification report..

[51]:
```
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score, confusion_matrix,␣
 ↪classification_report
from sklearn.feature_extraction.text import CountVectorizer


# Sample dataset
data = {
    'Email': [
        'FREE exclusive offer just for you',
        'Project status update',
        'URGENT: Your payment is due',
        'Notes from yesterday\'s meeting',
        'Buy luxury watches at 90% discount',
        'Team lunch on Friday',
        'You are the 1,000,000th visitor!',
        'Please review the proposal',
        'Your Netflix subscription expiring',
        'Request for vacation approval',
        'Hot singles in your area',
        'Weekly team progress report',
        'Make money fast from home',
        'Reminder: Client meeting tomorrow',
        'Enlarge your body parts now',
        'Quarterly financial summary',
        'Click here to claim your inheritance',
        'New security protocols update',
        'Lose 20 pounds in 2 weeks guaranteed',
        'Feedback on your presentation'
    ],
    'Spam': [ 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0]
}

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

# Split the data into features and target
X = df['Email']
y = df['Spam']

# Convert text data to numerical data using CountVectorizer
vectorizer = CountVectorizer()
```

```
X_vectorized = vectorizer.fit_transform(X)

# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X_vectorized, y,␣
 ↪test_size=0.3, random_state=42)

# Create a Logistic Regression model
model = LogisticRegression()

# Train the model
model.fit(X_train, y_train)

# Make predictions
y_pred = model.predict(X_test)

# Evaluate the model
accuracy = accuracy_score(y_test, y_pred)
conf_matrix = confusion_matrix(y_test, y_pred)
class_report = classification_report(y_test, y_pred)

print(f"Accuracy: {accuracy}")
print("Confusion Matrix:")
print(conf_matrix)
print("Classification Report:")
print(class_report)
```

```
Accuracy: 0.5
Confusion Matrix:
[[1 3]
 [0 2]]
Classification Report:
              precision    recall  f1-score   support

           0       1.00      0.25      0.40         4
           1       0.40      1.00      0.57         2

    accuracy                           0.50         6
   macro avg       0.70      0.62      0.49         6
weighted avg       0.80      0.50      0.46         6
```

Question 4. Explain the concept of Machine Learning (ML) and differentiate between supervised and unsupervised learning. Provide real-world examples of each type.

4 Machine Learning: Concepts, Types, and Real-World Applications

4.1 What is Machine Learning?

Machine Learning (ML) is a subset of artificial intelligence that enables systems to learn and improve from experience without being explicitly programmed. Instead of following static program instructions, ML algorithms build models based on sample data, known as "training data," to make predictions or decisions without being explicitly programmed to do so.

The key characteristic of ML is its ability to automatically identify patterns in data and use these patterns to make predictions on new, unseen data.

4.2 Supervised Learning vs. Unsupervised Learning

4.2.1 Supervised Learning

Definition: In supervised learning, the algorithm learns from labeled training data, meaning the input data comes with the correct output (or "answer"). The algorithm's task is to learn a mapping function from inputs to outputs.

Process: 1. The algorithm is trained on a dataset with known outcomes 2. It learns to predict the outcome given a set of inputs 3. The model's performance is evaluated based on how accurately it predicts the correct answers

Key Characteristics: - Requires labeled data - Goal is prediction - Clear feedback on accuracy (model knows when it's wrong) - Examples include classification and regression problems

Real-World Examples: 1. **Email Spam Classification**: Using features like sender information, subject line keywords, and content patterns to predict whether an email is spam or legitimate.

2. **Credit Scoring**: Predicting a customer's creditworthiness based on income, debt, payment history, and other financial attributes.
3. **Medical Diagnosis**: Predicting diseases from symptoms, lab tests, and medical images (e.g., detecting cancer in radiological images).
4. **House Price Prediction**: Estimating property values based on features like location, square footage, number of rooms, etc.
5. **Sentiment Analysis**: Determining whether text reviews or social media posts express positive, negative, or neutral sentiment.

4.2.2 Unsupervised Learning

Definition: In unsupervised learning, the algorithm works with unlabeled data, discovering hidden patterns or intrinsic structures without guidance about correct outputs.

Process: 1. The algorithm is presented with data without predefined labels 2. It finds patterns, similarities, or differences in the data 3. Success is measured by the algorithm's ability to discover meaningful structures

Key Characteristics: - Uses unlabeled data - Goal is discovery of patterns - No explicit right or wrong answers - Examples include clustering, association, and dimensionality reduction

Real-World Examples: 1. **Customer Segmentation**: Grouping customers with similar purchasing behaviors without predefined categories, used in targeted marketing.

2. **Anomaly Detection**: Identifying unusual patterns in credit card transactions to flag potential fraud, without prior examples of what fraud looks like.

3. **Recommendation Systems**: Discovering groups of similar products or content that users might like based on their previous interactions.

4. **Topic Modeling**: Discovering abstract topics in a collection of documents, used in content organization and search optimization.

5. **Market Basket Analysis**: Finding combinations of products that are frequently purchased together, helping with store layout and promotions.

4.3 Key Differences

Aspect	Supervised Learning	Unsupervised Learning
Data	Labeled	Unlabeled
Feedback	Direct	Indirect
Applications	Prediction, classification	Pattern discovery, grouping
Complexity	Generally simpler to understand	Often more complex to interpret
Examples	Regression, classification	Clustering, association rules

Machine learning continues to evolve rapidly, with newer paradigms like semi-supervised learning (using both labeled and unlabeled data) and reinforcement learning (learning through trial and error with rewards) expanding the possibilities beyond the traditional supervised/unsupervised dichotomy.

5 SECTION 4

Question 1. Write a Python script to connect to a MySQL database, create a table, insert data, and retrieve records using SQL commands executed through Python.

[2]:
```python
from mysql.connector import Error
import mysql.connector
import socket
import time

# MySQL Database Connection Parameters
DB_HOST = 'localhost'
DB_USER = 'sudeepwebdev'
DB_PASSWORD = 'password'
DB_NAME = 'your_database'
DB_PORT = 3306  # Default MySQL port

def test_mysql_connection():
    """Test if MySQL server is running and accessible"""
```

```
    try:
        # Try to establish a socket connection to check if MySQL server is␣
↪listening
        sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        sock.settimeout(2)  # 2 second timeout
        result = sock.connect_ex((DB_HOST, DB_PORT))
        sock.close()

        if result == 0:
            print(f"MySQL server is accessible at {DB_HOST}:{DB_PORT}")
            return True
        else:
            print(f"Error: MySQL server not accessible at {DB_HOST}:{DB_PORT}")
            print("Possible reasons:")
            print("1. MySQL server is not running")
            print("2. MySQL server is configured to use a different port")
            print("3. Firewall is blocking the connection")
            return False
    except Exception as e:
        print(f"Error checking MySQL connectivity: {e}")
        return False

def connect_to_mysql_database(retries=3, delay=2):
    """Connect to MySQL database with retry logic and return connection␣
↪object"""

    # First check if the server is accessible
    if not test_mysql_connection():
        print("Skipping connection attempt since server is not accessible")
        return None

    attempt = 0
    while attempt < retries:
        try:
            connection = mysql.connector.connect(
                host=DB_HOST,
                user=DB_USER,
                password=DB_PASSWORD,
                database=DB_NAME,
                port=DB_PORT,
                connection_timeout=10  # Timeout after 10 seconds
            )

            if connection.is_connected():
                db_info = connection.get_server_info()
                print(f"Successfully connected to MySQL server version␣
↪{db_info}")
```

```
                cursor = connection.cursor()
                cursor.execute("SELECT DATABASE();")
                db_name = cursor.fetchone()[0]
                print(f"Connected to database: {db_name}")
                cursor.close()
                return connection

        except Error as e:
            attempt += 1
            if "Unknown database" in str(e):
                print(f"Database '{DB_NAME}' does not exist. Attempting to␣
↪create it...")
                try:
                    # Connect without specifying database
                    temp_conn = mysql.connector.connect(
                        host=DB_HOST,
                        user=DB_USER,
                        password=DB_PASSWORD,
                        port=DB_PORT
                    )
                    temp_cursor = temp_conn.cursor()
                    temp_cursor.execute(f"CREATE DATABASE IF NOT EXISTS␣
↪{DB_NAME}")
                    temp_conn.close()
                    print(f"Created database {DB_NAME}")
                    # Now try again with the database
                    continue
                except Error as create_error:
                    print(f"Failed to create database: {create_error}")

            print(f"Connection attempt {attempt}/{retries} failed: {e}")
            if attempt < retries:
                print(f"Retrying in {delay} seconds...")
                time.sleep(delay)
            else:
                print("Maximum connection attempts reached. Could not connect␣
↪to MySQL.")
                print("\nTroubleshooting tips:")
                print("1. Verify MySQL service is running")
                print("2. Check credentials (username/password)")
                print("3. Ensure the database exists or you have permission to␣
↪create it")
                print("4. Check if MySQL is listening on the expected port")
                print("5. Verify there are no firewall restrictions")

    return None
```

```
# Rest of the functions (create_table, insert_sample_data, retrieve_records)␣
 ↪remain the same
# ...

# Main execution
if __name__ == "__main__":
    # Connect to MySQL database
    connection = connect_to_mysql_database()

    if connection is not None:
        # Create table
        create_table(connection)

        # Insert sample data
        insert_sample_data(connection)

        # Retrieve and display records
        retrieve_records(connection)

        # Close the connection
        if connection.is_connected():
            connection.close()
            print("\nMySQL connection closed")
```

```
Error: MySQL server not accessible at localhost:3306
Possible reasons:
1. MySQL server is not running
2. MySQL server is configured to use a different port
3. Firewall is blocking the connection
Skipping connection attempt since server is not accessible
```

Question 2. Implement Ridge and Lasso Regression on a dataset and compare the impact of L1 and L2 regularization techniques. Explain which model performs better and why. .

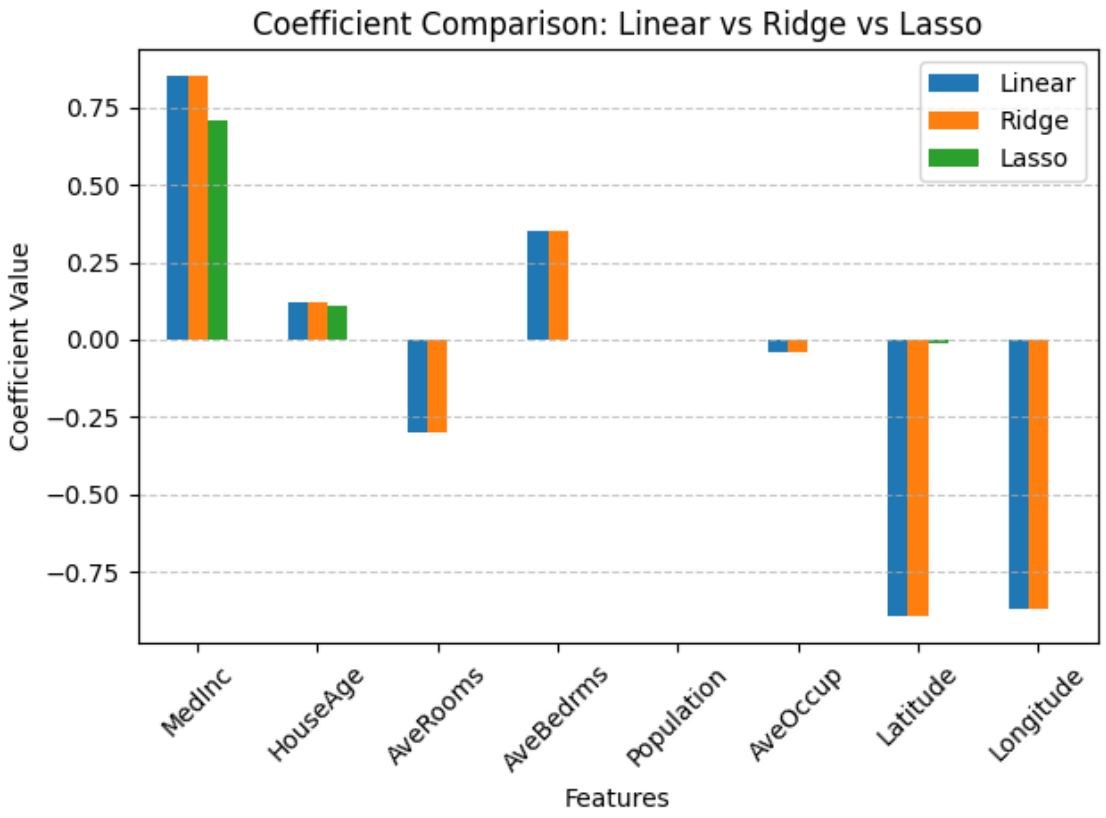
[3]:
```
import numpy as np
import pandas as pd
from sklearn.datasets import fetch_california_housing
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.linear_model import LinearRegression, Ridge, Lasso
from sklearn.metrics import mean_squared_error, r2_score

import matplotlib.pyplot as plt

# Load the California housing dataset
housing = fetch_california_housing()
X, y = housing.data, housing.target
```

```
# Feature names for reference
feature_names = housing.feature_names

# Split into training and test sets
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.3, random_state=42
)

# Standardize features
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)

# 1. Linear Regression (No Regularization)
lr = LinearRegression()
lr.fit(X_train_scaled, y_train)
y_pred_lr = lr.predict(X_test_scaled)
mse_lr = mean_squared_error(y_test, y_pred_lr)
r2_lr = r2_score(y_test, y_pred_lr)

# 2. Ridge Regression (L2 Regularization)
ridge = Ridge(alpha=1.0)  # alpha is the regularization strength
ridge.fit(X_train_scaled, y_train)
y_pred_ridge = ridge.predict(X_test_scaled)
mse_ridge = mean_squared_error(y_test, y_pred_ridge)
r2_ridge = r2_score(y_test, y_pred_ridge)

# 3. Lasso Regression (L1 Regularization)
lasso = Lasso(alpha=0.1)  # alpha is the regularization strength
lasso.fit(X_train_scaled, y_train)
y_pred_lasso = lasso.predict(X_test_scaled)
mse_lasso = mean_squared_error(y_test, y_pred_lasso)
r2_lasso = r2_score(y_test, y_pred_lasso)

# Print results
print("Model Performance Comparison:")
print("-" * 50)
print(f"Linear Regression: MSE = {mse_lr:.4f}, R² = {r2_lr:.4f}")
print(f"Ridge Regression:  MSE = {mse_ridge:.4f}, R² = {r2_ridge:.4f}")
print(f"Lasso Regression:  MSE = {mse_lasso:.4f}, R² = {r2_lasso:.4f}")

# Plot coefficients for comparison
plt.figure(figsize=(12, 6))
coefs = pd.DataFrame({
    'Linear': lr.coef_,
    'Ridge': ridge.coef_,
    'Lasso': lasso.coef_
```

```
}, index=feature_names)

coefs.plot(kind='bar')
plt.title('Coefficient Comparison: Linear vs Ridge vs Lasso')
plt.xlabel('Features')
plt.ylabel('Coefficient Value')
plt.grid(axis='y', linestyle='--', alpha=0.7)
plt.xticks(rotation=45)
plt.tight_layout()
plt.show()

# Explanation of results
print("\nInterpretation of Results:")
print("-" * 50)
print("1. Impact of L1 Regularization (Lasso):")
print("   - Encourages sparsity by setting some coefficients to exactly zero")
print("   - Feature selection capability: Eliminates less important features")
print("   - Better for high-dimensional data with many irrelevant features")
print()
print("2. Impact of L2 Regularization (Ridge):")
print("   - Shrinks all coefficients proportionally but rarely sets them to␣
 ↪exactly zero")
print("   - Better for handling multicollinearity (correlated features)")
print("   - Often provides more stable predictions when features are␣
 ↪correlated")
print()
print("3. Model Comparison:")
print("   - The model with the lower MSE and higher R² performs better")
print("   - Ridge typically outperforms Linear when multicollinearity exists")
print("   - Lasso is better when feature selection is desired")
print()

# Check which features Lasso deemed most important (non-zero coefficients)
important_features = [(feature_names[i], lasso.coef_[i])
                      for i in range(len(feature_names))
                      if abs(lasso.coef_[i]) > 0.001]
print("Features selected by Lasso (non-zero coefficients):")
for feature, coef in important_features:
    print(f"   - {feature}: {coef:.4f}")
```

```
Model Performance Comparison:
--------------------------------------------------
Linear Regression: MSE = 0.5306, R² = 0.5958
Ridge Regression:  MSE = 0.5305, R² = 0.5958
Lasso Regression:  MSE = 0.6648, R² = 0.4935
```

```
<Figure size 1200x600 with 0 Axes>
```




```
Interpretation of Results:
--------------------------------------------------
1. Impact of L1 Regularization (Lasso):
   - Encourages sparsity by setting some coefficients to exactly zero
   - Feature selection capability: Eliminates less important features
   - Better for high-dimensional data with many irrelevant features

2. Impact of L2 Regularization (Ridge):
   - Shrinks all coefficients proportionally but rarely sets them to exactly
zero
   - Better for handling multicollinearity (correlated features)
   - Often provides more stable predictions when features are correlated

3. Model Comparison:
   - The model with the lower MSE and higher R² performs better
   - Ridge typically outperforms Linear when multicollinearity exists
   - Lasso is better when feature selection is desired

Features selected by Lasso (non-zero coefficients):
   - MedInc: 0.7083
```

- HouseAge: 0.1066
- Latitude: -0.0104

Question 3: Explain overfitting and underfitting in machine learning models. Provide practical examples where overfitting occurs and discuss strategies to mitigate it using regularization.

5.1 Overfitting and Underfitting in Machine Learning

5.1.1 Overfitting

Definition:
Overfitting occurs when a machine learning model learns not only the underlying patterns in the training data but also the noise and random fluctuations. As a result, the model performs very well on the training data but poorly on unseen (test) data because it fails to generalize.

Practical Example:
Suppose you train a polynomial regression model with a very high degree on a small housing dataset. The model fits every data point in the training set perfectly, but when predicting prices for new houses, its predictions are inaccurate and erratic.

Symptoms:
- High accuracy on training data, low accuracy on test data. - Model is too complex relative to the amount of data.

5.1.2 Underfitting

Definition:
Underfitting happens when a model is too simple to capture the underlying structure of the data. It performs poorly on both training and test data.

Practical Example:
Using a linear regression model to fit data that has a clear nonlinear relationship (e.g., quadratic or exponential trend). The model cannot capture the complexity, resulting in high errors.

Symptoms:
- Poor performance on both training and test data. - Model is too simple for the problem.

5.2 Overfitting Example in Practice

In the California housing dataset, if you use a very complex model (e.g., a high-degree polynomial regression or a decision tree with no depth limit), the model may fit the training data extremely well but fail to predict new house prices accurately.

5.3 Strategies to Mitigate Overfitting: Regularization

Regularization adds a penalty to the loss function to discourage overly complex models (large coefficients), helping the model generalize better.

- **L1 Regularization (Lasso):**
  Adds a penalty equal to the absolute value of the coefficients. It can shrink some coefficients to zero, effectively performing feature selection.
- **L2 Regularization (Ridge):**
  Adds a penalty equal to the square of the coefficients. It shrinks all coefficients but does not set them exactly to zero.

Example from this notebook:
When comparing Linear Regression, Ridge, and Lasso on the California housing data: - **Linear Regression** may overfit if features are highly correlated or if there are many irrelevant features. - **Ridge Regression** (L2) reduces overfitting by shrinking coefficients, especially useful when features are correlated. - **Lasso Regression** (L1) can eliminate irrelevant features by setting their coefficients to zero, further reducing overfitting.

Other Strategies: - Use simpler models. - Gather more training data. - Use cross-validation to tune model complexity. - Prune decision trees or limit model parameters.

Summary:
- Overfitting: Model too complex, fits noise, poor generalization. - Underfitting: Model too simple, misses patterns, poor performance. - Regularization (L1/L2) helps control model complexity and reduce overfitting.

6 Section E

Question 1: Write a Python script to identify and remove duplicate rows from a large dataset using Pandas. Provide a detailed explanation of how Pandas handles duplicates.

6.1 Identifying and Removing Duplicate Rows in Pandas

To efficiently identify and remove duplicate rows from a large dataset using Pandas, you can use the `duplicated()` and `drop_duplicates()` methods.

6.1.1 Example Script

```
import pandas as pd

# Simulate a large dataset with duplicates
data = {
    'ID': [1, 2, 3, 4, 2, 5, 3, 6],
    'Name': ['Alice', 'Bob', 'Charlie', 'David', 'Bob', 'Eve', 'Charlie', 'Frank'],
    'Score': [85, 90, 95, 88, 90, 92, 95, 87]
}
df = pd.DataFrame(data)

print("Original DataFrame:")
print(df)
```

```
# Identify duplicate rows (all columns)
duplicates = df.duplicated()
print("\nDuplicate rows (True = duplicate):")
print(duplicates)

# Remove duplicate rows (keep first occurrence)
df_no_duplicates = df.drop_duplicates()
print("\nDataFrame after removing duplicates:")
print(df_no_duplicates)
```

6.1.2 How Pandas Handles Duplicates

- **`duplicated()`**: Returns a Boolean Series indicating whether each row is a duplicate of a previous row. By default, it considers all columns.
  - `df.duplicated()` marks `True` for every row that is a duplicate of a previous row.
  - You can specify columns: `df.duplicated(subset=['Name', 'Score'])`
  - The `keep` parameter controls which duplicates to mark (`'first'`, `'last'`, or `False` for all).
- **`drop_duplicates()`**: Removes duplicate rows, keeping the first occurrence by default.
  - `df.drop_duplicates()` removes all but the first occurrence of each duplicate row.
  - You can specify columns and the `keep` parameter as above.

Notes:

- Both methods are efficient for large datasets.
- By default, the index is not reset after removing duplicates.
- You can remove duplicates in-place with `inplace=True`.

Summary:
Pandas makes it easy to detect and remove duplicates, ensuring data integrity and preventing issues in downstream analysis.

[4]:
```
import pandas as pd

# Simulate a large dataset with duplicates
data = {
    'ID': [1, 2, 3, 4, 2, 5, 3, 6],
    'Name': ['Alice', 'Bob', 'Charlie', 'David', 'Bob', 'Eve', 'Charlie',␣
 ↪'Frank'],
    'Score': [85, 90, 95, 88, 90, 92, 95, 87]
}
df = pd.DataFrame(data)

print("Original DataFrame:")
print(df)

# Identify duplicate rows (all columns)
duplicates = df.duplicated()
print("\nDuplicate rows (True = duplicate):")
```

```
print(duplicates)

# Remove duplicate rows (keep first occurrence)
df_no_duplicates = df.drop_duplicates()
print("\nDataFrame after removing duplicates:")
print(df_no_duplicates)
```

```
Original DataFrame:
   ID     Name  Score
0   1    Alice     85
1   2      Bob     90
2   3  Charlie     95
3   4    David     88
4   2      Bob     90
5   5      Eve     92
6   3  Charlie     95
7   6    Frank     87

Duplicate rows (True = duplicate):
0    False
1    False
2    False
3    False
4     True
5    False
6     True
7    False
dtype: bool

DataFrame after removing duplicates:
   ID     Name  Score
0   1    Alice     85
1   2      Bob     90
2   3  Charlie     95
3   4    David     88
5   5      Eve     92
7   6    Frank     87
```

Question 2: Explain the importance of handling missing values in a dataset. Write Python code that demonstrates different methods to handle missing values (drop, fill, or interpolate).

[5]:
```
import pandas as pd
import numpy as np

# Create a sample DataFrame with missing values
data_missing = {
    'ID': [1, 2, 3, 4, 5],
    'Name': ['Alice', 'Bob', np.nan, 'David', 'Eve'],
```

```
    'Score': [85, np.nan, 95, 88, 92]
}
df_missing = pd.DataFrame(data_missing)
print("Original DataFrame with missing values:")
print(df_missing)

# 1. Drop rows with any missing values
df_drop = df_missing.dropna()
print("\nAfter dropping rows with missing values:")
print(df_drop)

# 2. Fill missing values with a constant (e.g., 0 or 'Unknown')
df_fill = df_missing.fillna({'Name': 'Unknown', 'Score': 0})
print("\nAfter filling missing values with constants:")
print(df_fill)

# 3. Interpolate missing values (numeric columns only)
df_interp = df_missing.copy()
df_interp['Score'] = df_interp['Score'].interpolate()
print("\nAfter interpolating missing numeric values:")
print(df_interp)
```

```
Original DataFrame with missing values:
   ID   Name  Score
0   1  Alice   85.0
1   2    Bob    NaN
2   3    NaN   95.0
3   4  David   88.0
4   5    Eve   92.0

After dropping rows with missing values:
   ID   Name  Score
0   1  Alice   85.0
3   4  David   88.0
4   5    Eve   92.0

After filling missing values with constants:
   ID     Name  Score
0   1    Alice   85.0
1   2      Bob    0.0
2   3  Unknown   95.0
3   4    David   88.0
4   5      Eve   92.0

After interpolating missing numeric values:
   ID   Name  Score
0   1  Alice   85.0
```

```
1   2    Bob   90.0
2   3    NaN   95.0
3   4  David   88.0
4   5    Eve   92.0
```

6.2 Importance of Handling Missing Values in a Dataset

Handling missing values is a crucial step in data preprocessing for the following reasons:

- **Data Integrity**: Missing values can distort statistical analyses and lead to incorrect conclusions. Many algorithms cannot handle missing data and may fail or produce unreliable results.
- **Bias Prevention**: If missing values are not handled properly, they can introduce bias. For example, simply dropping rows with missing values may disproportionately remove certain groups, skewing the dataset.
- **Model Performance**: Machine learning models often require complete data. Missing values can reduce the amount of usable data, decrease model accuracy, or cause errors during training and prediction.
- **Accurate Insights**: Properly handling missing data ensures that analyses and models reflect the true patterns in the data, leading to more reliable insights and decisions.

Common strategies to handle missing values: - Removing rows or columns with missing data (if the proportion is small) - Filling (imputing) missing values with statistical measures (mean, median, mode) - Using advanced imputation techniques (e.g., KNN, regression) - Interpolating values for time series data

Summary:
Addressing missing values is essential to maintain data quality, prevent bias, and ensure robust, accurate analysis and modeling.

Question 3: Illustrate the concept of indexing and slicing in Pandas DataFrame. Provide examples of selecting specific rows, columns, and sub-sections of the DataFrame.

[8]:
```python
# Illustrating indexing and slicing in Pandas DataFrame

# Display the original DataFrame
print("Original DataFrame:")
print(df)

# 1. Selecting a specific column (as a Series)
print("\nSelect the 'Name' column:")
print(df['Name'])

# 2. Selecting multiple columns (as a DataFrame)
print("\nSelect 'Name' and 'Score' columns:")
print(df[['Name', 'Score']])

# 3. Selecting specific rows by index
print("\nSelect rows with index 2 and 4:")
print(df.loc[[2, 4]])
```

```
# 4. Slicing rows by position (using iloc)
print("\nSelect first three rows:")
print(df.iloc[:3])

# 5. Selecting a sub-section (rows 1 to 4, columns 'Name' and 'Score')
print("\nSelect rows 1 to 4 and columns 'Name' and 'Score':")
print(df.loc[1:4, ['Name', 'Score']])

# 6. Conditional selection (rows where Score > 90)
print("\nRows where Score > 90:")
print(df[df['Score'] > 90])

# --- Explanation ---
print("""
Explanation:
- Selecting columns: Use df['col'] for a single column (Series), or df[['col1',␣
  ↪'col2']] for multiple columns (DataFrame).
- Selecting rows: Use df.loc[] for label-based indexing, and df.iloc[] for␣
  ↪position-based indexing.
- Slicing: df.iloc[start:end] selects rows by integer position.
- Sub-sections: Combine row and column selection, e.g., df.loc[rows, columns].
- Conditional selection: Use boolean indexing, e.g., df[df['Score'] > 90], to␣
  ↪filter rows based on conditions.
""")
```

```
Original DataFrame:
   ID     Name  Score
0   1    Alice     85
1   2      Bob     90
2   3  Charlie     95
3   4    David     88
4   2      Bob     90
5   5      Eve     92
6   3  Charlie     95
7   6    Frank     87

Select the 'Name' column:
0      Alice
1        Bob
2    Charlie
3      David
4        Bob
5        Eve
6    Charlie
7      Frank
Name: Name, dtype: object
```

```
Select 'Name' and 'Score' columns:
      Name  Score
0    Alice     85
1      Bob     90
2  Charlie     95
3    David     88
4      Bob     90
5      Eve     92
6  Charlie     95
7    Frank     87

Select rows with index 2 and 4:
   ID     Name  Score
2   3  Charlie     95
4   2      Bob     90

Select first three rows:
   ID     Name  Score
0   1    Alice     85
1   2      Bob     90
2   3  Charlie     95

Select rows 1 to 4 and columns 'Name' and 'Score':
      Name  Score
1      Bob     90
2  Charlie     95
3    David     88
4      Bob     90

Rows where Score > 90:
   ID     Name  Score
2   3  Charlie     95
5   5      Eve     92
6   3  Charlie     95

Explanation:
- Selecting columns: Use df['col'] for a single column (Series), or df[['col1',
'col2']] for multiple columns (DataFrame).
- Selecting rows: Use df.loc[] for label-based indexing, and df.iloc[] for
position-based indexing.
- Slicing: df.iloc[start:end] selects rows by integer position.
- Sub-sections: Combine row and column selection, e.g., df.loc[rows, columns].
- Conditional selection: Use boolean indexing, e.g., df[df['Score'] > 90], to
filter rows based on conditions.
```

7 Section 6

Question 1. Demonstrate how to create and customize interactive plots using Plotly. Explain how it differs from Matplotlib and Seaborn for data visualization.

7.1 Creating and Customizing Interactive Plots with Plotly

Plotly is a powerful Python library for creating interactive, publication-quality visualizations. Unlike Matplotlib and Seaborn, which primarily generate static images, Plotly produces interactive plots that users can zoom, pan, hover, and export directly from the browser.

7.1.1 How Plotly Differs from Matplotlib and Seaborn

Feature	Plotly	Matplotlib/Seaborn
Interactivity	Fully interactive by default	Mostly static (limited interactivity with mpld3 or widgets)
Ease of Use	High-level API (Plotly Express)	Matplotlib: low-level, Seaborn: high-level for stats
Web Integration	Native HTML/JS output	Static images (PNG, SVG, PDF)
Customization	Extensive, via JSON-like syntax	Extensive, but sometimes verbose
3D/Maps	Built-in support	Limited (Matplotlib 3D is basic)
Performance	Slower for very large datasets	Faster for static, large datasets

7.1.2 Example Use Cases

- **Dashboards**: Plotly is ideal for building interactive dashboards (e.g., with Dash).
- **Data Exploration**: Hover tooltips and zooming make it easy to explore data visually.
- **Presentations**: Interactive plots engage audiences and allow dynamic exploration.

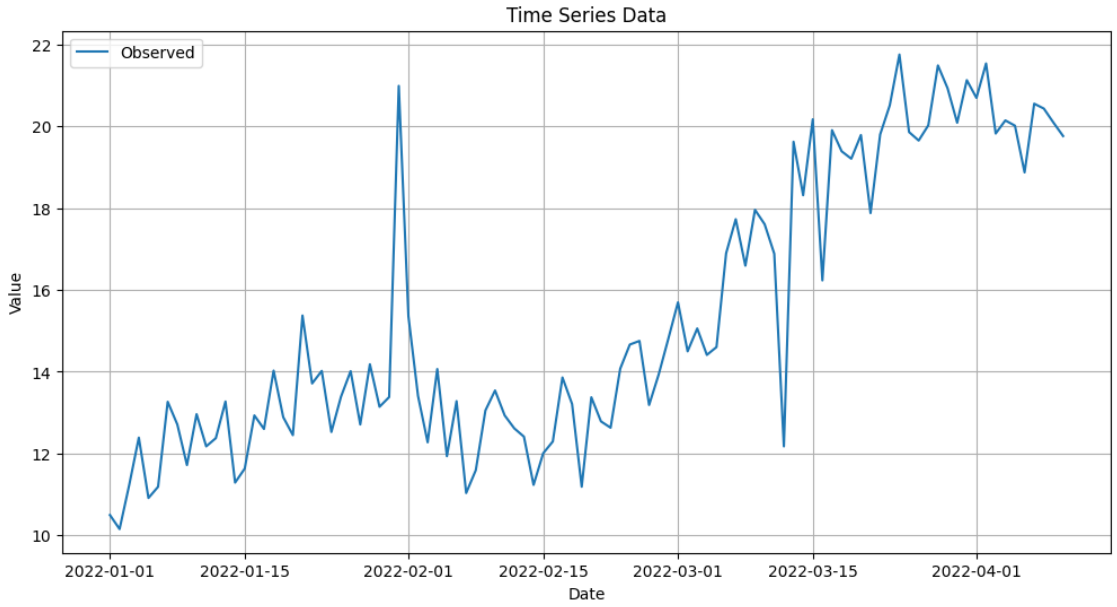
[14]:
```python
import plotly.express as px
import pandas as pd

# Sample DataFrame
df = pd.DataFrame({
    'ID': [1, 2, 3, 4, 5, 6],
    'Name': ['Alice', 'Bob', 'Charlie', 'David', 'Eve', 'Frank'],
    'Score': [85, 90, 95, 88, 92, 87]
})

# Create an interactive scatter plot
fig = px.scatter(
    df,
    x='ID',
    y='Score',
    color='Name',
    title='Interactive Score Plot',
```

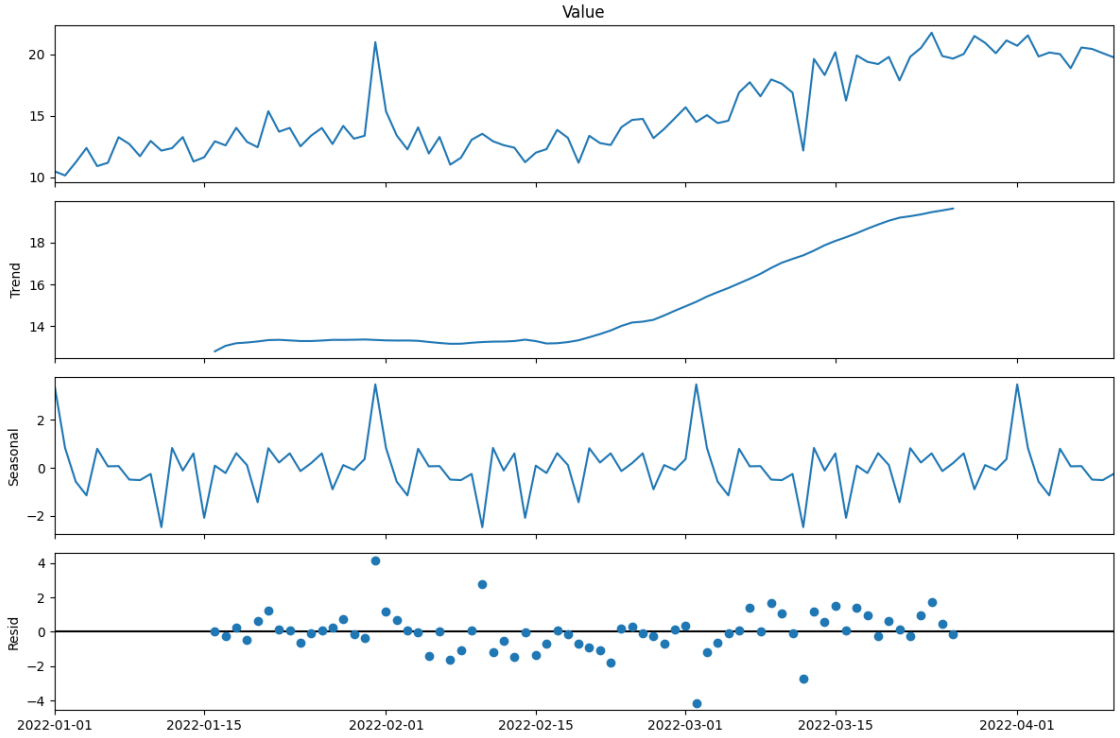
```
    labels={'Score': 'Test Score', 'ID': 'Student ID'},
    hover_data=['Name']
)

# Customize layout
fig.update_layout(
    xaxis=dict(showgrid=True),
    yaxis=dict(showgrid=True),
    legend_title_text='Student Name'
)

fig.show()
```

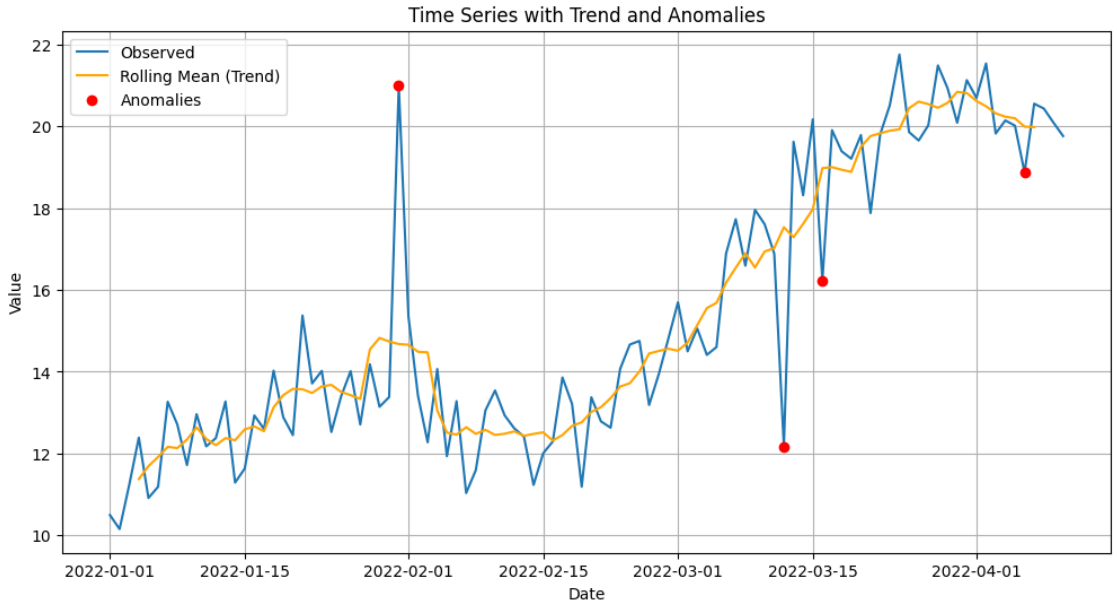
Question 2. Write a Python script to perform time series analysis and visualization on a dataset. Explain how to identify trends, seasonality, and anomalies.

[18]:
```
import pandas as pd
import numpy as np
from statsmodels.tsa.seasonal import seasonal_decompose

import matplotlib.pyplot as plt

# Create a sample time series dataset
np.random.seed(42)
date_range = pd.date_range(start='2022-01-01', periods=100, freq='D')
trend = np.linspace(10, 20, 100)  # upward trend
seasonality = 2 * np.sin(np.linspace(0, 3 * np.pi, 100))  # seasonal pattern
noise = np.random.normal(0, 1, 100)  # random noise
anomaly = np.zeros(100)
anomaly[30] = 8  # inject an anomaly
anomaly[70] = -6

values = trend + seasonality + noise + anomaly

df_ts = pd.DataFrame({'Date': date_range, 'Value': values})
df_ts.set_index('Date', inplace=True)

# Plot the time series
plt.figure(figsize=(12, 6))
plt.plot(df_ts.index, df_ts['Value'], label='Observed')
plt.title('Time Series Data')
plt.xlabel('Date')
plt.ylabel('Value')
plt.legend()
plt.grid(True)
plt.show()
```

```
# Decompose the time series (trend, seasonality, residuals)

decomposition = seasonal_decompose(df_ts['Value'], model='additive', period=30)
fig = decomposition.plot()
fig.set_size_inches(12, 8)
plt.tight_layout()
plt.show()

# Identify anomalies (simple method: points far from rolling mean)
rolling_mean = df_ts['Value'].rolling(window=7, center=True).mean()
rolling_std = df_ts['Value'].rolling(window=7, center=True).std()
anomalies = df_ts[np.abs(df_ts['Value'] - rolling_mean) > 2 * rolling_std]

plt.figure(figsize=(12, 6))
plt.plot(df_ts.index, df_ts['Value'], label='Observed')
plt.plot(df_ts.index, rolling_mean, color='orange', label='Rolling Mean␣
 ↪(Trend)')
plt.scatter(anomalies.index, anomalies['Value'], color='red',␣
 ↪label='Anomalies', zorder=5)
plt.title('Time Series with Trend and Anomalies')
plt.xlabel('Date')
plt.ylabel('Value')
plt.legend()
plt.grid(True)
plt.show()

# --- Explanation ---
print("""
How to identify:
- Trend: Look for long-term upward or downward movement (see rolling mean or␣
 ↪decomposition trend).
- Seasonality: Look for repeating patterns at regular intervals (see␣
 ↪decomposition seasonal component).
- Anomalies: Points that deviate significantly from the trend/seasonality␣
 ↪(highlighted in red).
""")
```


Time Series Data
Observed
Value
Date
2022-01-01
2022-01-15
2022-02-01
2022-02-15
2022-03-01
2022-03-15
2022-04-01


Value
Trend
Seasonal
Resid
2022-01-01
2022-01-15
2022-02-01
2022-02-15
2022-03-01
2022-03-15
2022-04-01




```
How to identify:
- Trend: Look for long-term upward or downward movement (see rolling mean or
decomposition trend).
- Seasonality: Look for repeating patterns at regular intervals (see
decomposition seasonal component).
- Anomalies: Points that deviate significantly from the trend/seasonality
(highlighted in red).
```

Question 3. Explain the importance of correlation in data analysis. Create a correlation matrix using Pandas and visualize it with a heatmap using Seaborn.

[20]:
```python
import seaborn as sns

import matplotlib.pyplot as plt

# Calculate the correlation matrix for the numeric columns of DataFrame 'df'
corr_matrix = df.select_dtypes(include='number').corr()

print("Correlation Matrix:")
print(corr_matrix)

# Visualize the correlation matrix with a heatmap
plt.figure(figsize=(6, 4))
sns.heatmap(corr_matrix, annot=True, cmap='coolwarm', linewidths=0.5)
plt.title("Correlation Heatmap")
plt.show()
```

```
# Explanation:
# Correlation measures the strength and direction of the linear relationship␣
 ↪between variables.
# In data analysis, understanding correlations helps identify which variables␣
 ↪are related,
# detect multicollinearity, and select features for modeling.
# The heatmap visually highlights strong positive (closer to 1) and negative␣
 ↪(closer to -1) correlations.
```

```
Correlation Matrix:
              ID     Score
ID      1.000000  0.132915
Score   0.132915  1.000000
```