Register Number															

SRM Institute of Science and Technology
College of Engineering and Technology
School of Computing

SRM Nagar, Kattankulathur – 603203, Chengalpattu District, Tamil Nadu

Academic Year: 2024-25 (EVEN)

Set -

Test: FT4 | Date: 29-04-2025
Course Code & Title: 21CSS303T-Data Science | Duration: Two periods
Year& Sem: III Year /VI Sem | Max.Marks:50

Course Articulation Matrix:

Course Outcome	PO1	PO2	PO3	PO4	PO5	PO6	PO7	PO8	PO9	PO10	PO11	PO12
CO3	-	-	-	-	1	-	-	-	-	-	-	-
CO4	-	-	-	-	1	-	-	-	-	-	-	-
CO5	-	-	-	-	1	-	-	-	-	-	-	-

Note: CO3 – To identify data manipulation and cleaning techniques using pandas
CO4 – To constructs the Graphs and plots to represent the data using python packages
CO5 – To apply the principles of the data science techniques to predict and forecast the outcome of real-world problem

Part – A (10 x 1 = 10 Marks)

Instructions:
1) Answer **ALL** questions.
2) The duration for answering Part A is **15 minutes** (this sheet will be collected after 15 minutes).
3) **Encircle the correct answer**.

S.No	Question	Marks	BL	CO	PO	PI Code
1	Which of the following methods is used to remove duplicate rows from a Data Frame in pandas? a) drop() b) drop_duplicates() c) unique() d) remove_duplicates()	1	1	3	5	
2	What function is used to fill missing values in a pandas Data Frame? a) fillna() b) replace_null() c) na_fill() d) fill()	1	1	3	5	
3	Which of the following is NOT a method for handling missing data? a) Deletion b) Imputation c) Forward/Backward fill d) Duplicating	1	1	3	5	
4	When preparing data for modeling, why is scaling important? a) To hide patterns b) To reduce memory c) To ensure equal importance of features d) To convert text to numbers	1	2	3	5	
5	Which of these is NOT a standard data cleaning step? a) Handling missing values b) Removing duplicates	1	2	3	5	

	c) Building machine learning models d) Correcting data types					
6	Which function is used to set the x-axis label in matplotlib? a) plt.labelx() b) plt.xlabel() c) plt.xaxis() d) plt.set_xlabel()	1	1	4	5	
7	Which method is used to add a legend to the plot? a) plt.add_legend() b) plt.show_legend() c) plt.legend() d) plt.make_legend()	1	1	4	5	
8	Which function is used to display multiple plots in one figure? a) plt.split() b) plt.multi_plot() c) plt.subplot() d) plt.div()	1	1	4	5	
9	What function is used to add custom text annotations to a plot? a) plt.comment() b) plt.annotate() c) plt.tag() d) plt.label()	1	2	5	5	
10	Which function sets the size of the overall figure in matplotlib? a) plt.resize() b) plt.figure(figsize=(w, h)) c) plt.set_size() d) plt.figsize()	1	2	5	5	

Register Number													

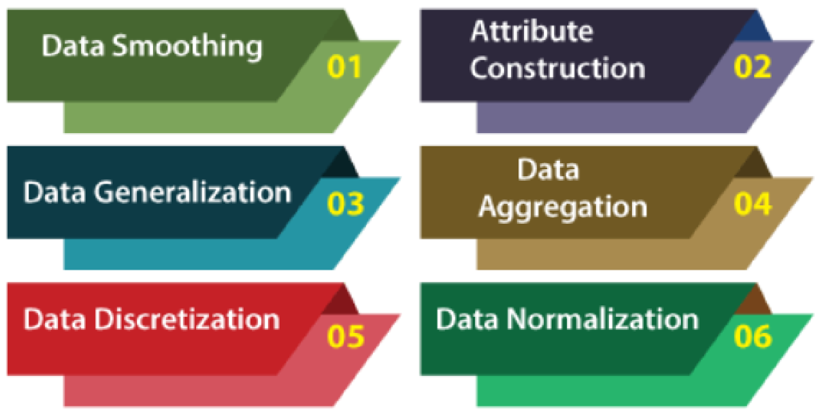
Set -

SRM Institute of Science and Technology
College of Engineering and Technology
School of Computing
SRM Nagar, Kattankulathur – 603203, Chengalpattu District, Tamil Nadu
Academic Year: 2024-25 (EVEN SEM)

Test: FT4 — Date:29-04-2025
Course Code & Title: 21CSS303T-Data Science — Duration: Two periods
Year& Sem: III Year /VI Sem — Max.Marks:50

Part – B (4 x 5 = 20 Marks)
Instructions: Answer **ANY FOUR** Questions

Q. No	Question	Marks	BL	CO	PO	PI Code
11	Discuss the various methods of handling missing data in the dataset. Listwise Deletion: Remove entire rows or columns containing missing values. This method is simple but can result in a significant loss of data, especially if there are many missing values. Pairwise Deletion: Remove pairs of observations where at least one value is missing. This is less wasteful than listwise deletion but can introduce bias if missingness is not random. Imputation: Mean/Median/Mode Imputation: Replace missing values with the mean, median, or mode of the respective column. This is a simple approach but can introduce bias if the distribution is skewed. K-Nearest Neighbors (KNN) Imputation: Impute missing values using the average values of the k nearest neighbors. This method can be effective for numerical data. Regression Imputation: Use regression models to predict missing values based on other features. This is suitable for numerical data with strong relationships between features. Multiple Imputation: Create multiple imputed datasets by filling in missing values with different plausible values. This method can help to account for uncertainty in the imputation process. 3. Using a "Missing" Category (For Categorical Data) In cases of categorical variables, instead of filling in missing values with a mode or using imputation, you can create a new category or label indicating that the data is missing. For example, for a column like Color, if there are missing values, you can replace them with "Unknown" or "Missing". Pandas method: df['Color'].fillna('Unknown'). Preserves information about the missingness. This method can potentially introduce noise, as the new category may not represent an actual value. 4. Using Algorithms That Handle Missing Data	5	2	3	5	

	Some machine learning algorithms, such as XGBoost, Random Forests, and CatBoost, can handle missing data internally during training without requiring explicit imputation. These models can work directly with missing values by learning from the patterns of the data. Not all algorithms can handle missing data natively. Results may vary depending on the implementation and how the algorithm handles the missing values. 5. Multiple Imputation This technique involves creating multiple datasets with different imputed values and then combining the results to account for uncertainty in the imputation process. Typically used when data are missing in a random or non-random fashion. Methods like Multiple Imputation by Chained Equations (MICE) are available in libraries like statsmodels and fancyimpute. 6. Predictive Modeling (Advanced Imputation) Use machine learning algorithms (e.g., regression, decision trees) to predict missing values based on other features. A model is trained using the non-missing data and then used to predict missing values. 7. Leave Missing Values As-Is (For Some Models) In some cases, particularly when using deep learning models, it may be acceptable to leave missing values as they are and let the model learn how to handle them during training. Models like neural networks can handle missing data if they are explicitly designed to do so. May lead to poor model performance if the model does not handle missing values well.					
12	Explain various data transformation techniques used in data preprocessing.  Data Smoothing 01 Attribute Construction 02 Data Generalization 03 Data Aggregation 04 Data Discretization 05 Data Normalization 06 **Data smoothing** is a process that is used to remove noise from the	5	3	3	5	

	dataset using some algorithms.
	It allows for highlighting important features present in the dataset.
	It helps in predicting the patterns.
	When collecting data, it can be manipulated to eliminate or reduce any variance or any other noise form.
	Attribute Construction:
	In the attribute construction method, the new attributes consult the existing attributes to construct a new data set that eases data mining. New attributes are created and applied to assist the mining process from the given attributes. This simplifies the original data and makes the mining more efficient.
	
	Data Generalization:
	Data generalization is the process of converting detailed data into a more abstract, higher-level representation while retaining essential information.
	It is commonly used in data mining, privacy preservation, and machine learning to reduce complexity and improve model generalization.
	Types
	Attribute Generalization
	Hierarchical Generalization
	Numeric Generalization
	Text Generalization
	
	Data collection or aggregation is the method of storing and presenting data in a summary format.
	The data may be obtained from multiple data sources to integrate these data sources into a data analysis description. This is a crucial step since the accuracy of data analysis insights is highly dependent on the quantity and quality of the data used.
	
	Data discretization is the process of converting continuous numerical data into discrete categories (bins).
	It is commonly used in machine learning, data mining, and feature engineering to simplify models and improve Interpretability.
	# Sample dataset
	data = {'Age': [22, 25, 30, 35, 40, 45, 50, 55, 60]}
	df = pd.DataFrame(data)
	 # Equal-width binning into 3 categories df['Age_Binned'] = pd.cut(df['Age'], bins=3, labels=['Young', 'Middle-aged', 'Old’])
	print(df)
	
	Data normalization is a preprocessing technique used to scale numerical data into a specific range, usually [0,1] or [-1,1].
	It ensures that features contribute equally to a model, preventing bias due to different scales.
	Why Normalize Data?
	✅ Improves Machine Learning Performance – Many algorithms (e.g., KNN, SVM, Neural Networks) perform better with normalized data.
	✅ Speeds Up Convergence – Gradient descent optimizes faster when features are scaled.
	✅ Prevents Dominance of Large-Scale Features – Avoids a situation where one feature overpowers others.					

13	Write a Python program that accepts a sentence from the user and performs the following string operations: 1. Display the total number of words in the sentence. 2. Convert the entire sentence to title case (first letter capitalized). 3. Find and display the number of times the word 'the' appears	5	2	3	5	

(case insensitive).
4. Replace all occurrences of the word 'and' with '&'.

```
# Accept sentence from the user
sentence = input("Enter a sentence: ")

# 1. Display the total number of words in the sentence
words = sentence.split()
num_words = len(words)
print(f"Total number of words: {num_words}")

# 2. Convert the entire sentence to title case
title_case_sentence = sentence.title()
print(f"Sentence in title case: {title_case_sentence}")

# 3. Find and display the number of times the word 'the' appears (case
insensitive)
word_count_the = sentence.lower().split().count('the')
print(f"Number of times the word 'the' appears: {word_count_the}")

# 4. Replace all occurrences of the word 'and' with '&'
replaced_sentence = sentence.replace('and', '&').replace('AND', '&')
print(f"Sentence with 'and' replaced by '&': {replaced_sentence}")
```

14	Explain the concept of subplots in Matplotlib with suitable examples.	5	3	4	5	

In **Matplotlib**, **subplots** are a way of organizing multiple plots in a single figure. This is useful when you want to display more than one plot side by side or in a grid, making it easier to compare data and results visually. The concept of subplots allows you to create a grid layout of multiple axes (individual plots) within a single figure.

### \`plt.subplot()\` vs \`plt.subplots()\`

Matplotlib provides two main functions for creating subplots:
- **\`plt.subplot()\`**: This function creates a single subplot in a specific position within a grid.
- **\`plt.subplots()\`**: This function creates multiple subplots at once, returning both the figure and axes objects.

#### 1. **\`plt.subplot()\`**
The \`subplot()\` function divides the figure into a grid and places a subplot in a specific position within that grid.

Syntax:

~~~
  ```python
  plt.subplot(nrows, ncols, index)
  ```
~~~

- \`nrows\`: Number of rows in the grid.
- \`ncols\`: Number of columns in the grid.
- \`index\`: Index of the subplot to create (counting starts from 1).

Example:

~~~
  ```python
  import matplotlib.pyplot as plt
~~~

    # Create a 2x2 grid of subplots and plot different graphs in each.
    plt.subplot(2, 2, 1) # Row 1, Column 1
    plt.plot([1, 2, 3], [1, 4, 9])
    plt.title('Plot 1')

    plt.subplot(2, 2, 2) # Row 1, Column 2
    plt.plot([1, 2, 3], [9, 4, 1])
    plt.title('Plot 2')

    plt.subplot(2, 2, 3) # Row 2, Column 1
    plt.plot([1, 2, 3], [1, 2, 3])
    plt.title('Plot 3')

    plt.subplot(2, 2, 4) # Row 2, Column 2
    plt.plot([1, 2, 3], [3, 2, 1])
    plt.title('Plot 4')

    plt.tight_layout() # Adjusts layout to avoid overlap
    plt.show()

Output: A 2x2 grid with four different plots.

#### 2. **`plt.subplots()`**
The `subplots()` function creates a grid of subplots and returns both the **figure** and **axes** objects. This is a more flexible and modern approach compared to `plt.subplot()`, especially when working with multiple subplots.

Syntax:
```python
fig, axes = plt.subplots(nrows, ncols)
```

- `nrows`: Number of rows in the grid.
- `ncols`: Number of columns in the grid.
- `fig`: The figure object.
- `axes`: A 2D array of axes objects (or a 1D array if there's only one row or column).

Example:
```python
import matplotlib.pyplot as plt

# Create a 2x2 grid of subplots
fig, axes = plt.subplots(2, 2)

# Plot on the first subplot
axes[0, 0].plot([1, 2, 3], [1, 4, 9])
axes[0, 0].set_title('Plot 1')

# Plot on the second subplot
axes[0, 1].plot([1, 2, 3], [9, 4, 1])
axes[0, 1].set_title('Plot 2')

# Plot on the third subplot
axes[1, 0].plot([1, 2, 3], [1, 2, 3])
axes[1, 0].set_title('Plot 3')

# Plot on the fourth subplot
```

	axes[1, 1].plot([1, 2, 3], [3, 2, 1])
	axes[1, 1].set_title('Plot 4')
	
	plt.tight_layout() # Adjusts layout to avoid overlap
	plt.show()
	```
	**Output**: A 2x2 grid with four different plots.

	### Benefits of `plt.subplots()` over `plt.subplot()`
	- **Better Organization**: With `plt.subplots()`, the axes are returned as a 2D array, making it easy to access each subplot programmatically.
	- **Flexibility**: You can create more complex subplot layouts, such as grids of different sizes.
	- **Cleaner Code**: `plt.subplots()` automatically handles figure creation and axes layout, reducing the need to call `plt.figure()` and `plt.subplot()` repeatedly.

	### 3. **Advanced Customizations**

	You can further customize the appearance of subplots using:
	- **`figsize`**: Set the size of the figure (width, height) when calling `plt.subplots()`.
	- **`tight_layout()`**: Adjusts spacing between subplots to avoid overlap.
	- **Sharing Axes**: You can share axes between subplots using the `sharex` and `sharey` parameters in `plt.subplots()`.

	#### Example with `figsize`, `tight_layout()`, and `sharex`/`sharey`:
	```
	python
	import matplotlib.pyplot as plt
	
	# Create a 2x2 grid of subplots with shared x and y axes
	fig, axes = plt.subplots(2, 2, figsize=(10, 6), sharex=True, sharey=True)
	
	# Plot on the first subplot
	axes[0, 0].plot([1, 2, 3], [1, 4, 9])
	axes[0, 0].set_title('Plot 1')
	
	# Plot on the second subplot
	axes[0, 1].plot([1, 2, 3], [9, 4, 1])
	axes[0, 1].set_title('Plot 2')
	
	# Plot on the third subplot
	axes[1, 0].plot([1, 2, 3], [1, 2, 3])
	axes[1, 0].set_title('Plot 3')
	
	# Plot on the fourth subplot
	axes[1, 1].plot([1, 2, 3], [3, 2, 1])
	axes[1, 1].set_title('Plot 4')
	
	plt.tight_layout() # Adjust layout to avoid overlap
	plt.show()
	``					

15	Define annotations in the context of data visualization using Matplotlib and briefly explain the types of annotations used.	5	3	5	5	

In the context of **data visualization**, **annotations** in Matplotlib are used to add explanatory text, labels, arrows, or other elements to the plot to provide additional context or emphasize important points. Annotations help to make the plot more informative, guiding the audience's attention to specific details or key data points.
Annotations can be used to:

- Explain the meaning of data points
- Highlight specific trends or patterns
- Add contextual information (e.g., labels, titles, or descriptions)
- Provide insight into outliers or unusual data points

Key Types of Annotations in Matplotlib

1. **Text Annotations**: Text annotations are used to place text at specific locations within the plot to describe points, trends, or any other important aspect.

   Syntax:
   ```
   plt.text(x, y, 'Text', fontsize=12, color='red',
   ha='center', va='center')
   ```

   - `x`, `y`: Coordinates where the text will be placed.
   - `'Text'`: The actual text to display.
   - `fontsize`: Font size of the text.
   - `color`: Color of the text.
   - `ha`, `va`: Horizontal and vertical alignment of the text.

   Example:
   ```
   import matplotlib.pyplot as plt

   x = [1, 2, 3, 4, 5]
   y = [1, 4, 9, 16, 25]
   plt.plot(x, y)
   plt.text(3, 20, 'This is a point', fontsize=12,
   color='blue', ha='left')
   plt.show()
   ```

2. **Arrow Annotations**: Arrow annotations help direct attention to a specific point or region of the plot. These are useful when you want to point out a specific feature in the graph.

   Syntax:
   ```
   plt.annotate('Text', xy=(x, y), xytext=(x_offset,
   y_offset), arrowprops=dict(facecolor='blue',
   arrowstyle='->'))
   ```

   - `xy`: The coordinates of the point to annotate.
   - `xytext`: The coordinates of the text.
   - `arrowprops`: A dictionary that specifies properties of the arrow (e.g., color, style).

Example:
```
import matplotlib.pyplot as plt

x = [1, 2, 3, 4, 5]
y = [1, 4, 9, 16, 25]
plt.plot(x, y)
plt.annotate('Point (3, 9)', xy=(3, 9),
xytext=(4, 10),
             arrowprops=dict(facecolor='red',
arrowstyle='->'))
plt.show()
```

3. **Bounding Box Annotations**: A bounding box is a box drawn around the annotation text to highlight it. This is helpful to make sure the text stands out clearly against the background.

   Syntax:
   ```
   plt.text(x, y, 'Text',
   bbox=dict(facecolor='yellow', alpha=0.5))
   ```

   - `bbox`: A dictionary specifying the box properties (e.g., facecolor, alpha for transparency).

   Example:
   ```
   import matplotlib.pyplot as plt

   x = [1, 2, 3, 4, 5]
   y = [1, 4, 9, 16, 25]
   plt.plot(x, y)
   plt.text(3, 20, 'This is a point', fontsize=12,
   color='blue', bbox=dict(facecolor='yellow',
   alpha=0.5))
   plt.show()
   ```

4. **Multiple Annotations with `plt.annotate()`**: The `annotate()` function is versatile and can also be used to annotate multiple points on the same plot. You can specify the text and coordinates dynamically, creating a more detailed and interactive plot.

   Example:
   ```
   import matplotlib.pyplot as plt

   x = [1, 2, 3, 4, 5]
   y = [1, 4, 9, 16, 25]
   plt.plot(x, y)

   # Annotating multiple points
   for i in range(len(x)):
       plt.annotate(f'({x[i]}, {y[i]})', xy=(x[i],
   y[i]), xytext=(x[i]+0.1, y[i]+1),

   arrowprops=dict(facecolor='green',
   arrowstyle='->'))

   plt.show()
   ```

5. **Highlighting Specific Points**: Annotations can also be used to highlight specific points with different markers or styles (e.g., circles, squares, etc.).

Example:
```
import matplotlib.pyplot as plt

x = [1, 2, 3, 4, 5]
y = [1, 4, 9, 16, 25]
plt.plot(x, y)
plt.scatter([3], [9], color='red', s=100)  #
Highlight a specific point
plt.text(3, 9, 'Highlighted Point', fontsize=12,
color='black', ha='center', va='center')
plt.show()
```

Part – C (2 x 10 = 20 Marks)
Instructions: Answer ALL questions.

Q. No	Question	Marks	BL	CO	PO	PI Code
16 a	Discuss the major challenges encountered while working with large datasets and how these challenges impact data preprocessing, storage, and analysis.	10	2	3	5	

Working with **large datasets** presents several challenges that can significantly impact the stages of **data preprocessing**, **storage**, and **analysis**. These challenges can arise from the sheer volume of data, the variety of data types, and the complexity of processing that data. Below are the major challenges encountered when working with large datasets and their impact on the different stages of data handling:

1. Data Preprocessing Challenges

a) Memory Limitations

- **Challenge**: Large datasets can easily exceed the available memory (RAM) on a typical machine. Loading the entire dataset into memory may lead to crashes or significant slowdowns in the system.
- **Impact**: This directly affects preprocessing tasks such as cleaning, transforming, and normalizing data. For example, operations like removing duplicates, filling missing values, and encoding categorical features may become difficult or impossible to execute on the full dataset.
- **Solution**:
  - o **Chunking**: Process data in smaller chunks that fit into memory.
  - o **Dask or Vaex**: Use out-of-core libraries designed to handle large datasets that don't fit in memory.

    - **Efficient Data Types**: Reduce memory consumption by using more compact data types (e.g., `float32` instead of `float64`).

b) Data Quality and Inconsistencies

- **Challenge**: Large datasets often contain missing values, outliers, duplicate entries, or inconsistent formats.
- **Impact**: Cleaning and handling missing or inconsistent data can be very time-consuming and complex in large datasets. Inconsistent data may affect data preprocessing tasks such as imputation, encoding, and scaling.
- **Solution**:
    - **Automated Data Cleaning**: Use automated scripts or libraries (e.g., `pandas`, `numpy`) to handle missing data, duplicates, and outliers in bulk.
    - **Distributed Processing**: Utilize distributed frameworks (e.g., **Apache Spark**, **Dask**) to clean data across multiple nodes.

c) Data Transformation Complexity

- **Challenge**: Transforming large datasets—such as scaling, normalizing, encoding, or feature engineering—can be resource-intensive and slow.
- **Impact**: Time-consuming transformations on large datasets can delay analysis and modeling processes. Complex transformations may also require more computational power.
- **Solution**:
    - **Parallel Processing**: Use libraries that support parallel processing (e.g., **joblib**, **Dask**).
    - **Incremental Learning**: Use algorithms that support incremental learning or mini-batch processing (e.g., **Stochastic Gradient Descent (SGD)**, **Naive Bayes**).

2. Storage Challenges

a) Storage Capacity

- **Challenge**: Large datasets can take up significant storage space, which might exceed local storage capacity.
- **Impact**: Storing large amounts of data can be costly, particularly if data needs to be stored in high-performance formats for quick access. It may also slow down read/write operations.
- **Solution**:
    - **Data Compression**: Compress data using formats like **Parquet**, **ORC**, or **HDF5**, which reduce the storage size without losing data.
    - **Distributed Storage**: Use cloud-based storage systems (e.g., **Amazon S3**, **Google Cloud Storage**) or distributed file systems like **HDFS**

    - (Hadoop Distributed File System) for large-scale storage.
    - **Efficient Data Formats**: Use binary file formats like **Parquet** and **Feather** for efficient storage and fast access.

b) Data Integration and Formats

- **Challenge**: Large datasets often come from multiple sources and in various formats, such as CSV, JSON, XML, or databases.
- **Impact**: Merging or integrating data from heterogeneous sources can introduce additional complexity, and working with multiple formats may require additional preprocessing steps like parsing, converting, or standardizing formats.
- **Solution**:
    - **Data Lakes**: Use **data lakes** to store large volumes of raw, unstructured data and then process it as needed.
    - **ETL (Extract, Transform, Load)**: Implement ETL processes to transform data into a consistent format for analysis.

3. Analysis Challenges

a) Slow Computation and Processing Time

- **Challenge**: Analyzing large datasets (e.g., performing complex calculations, aggregations, or machine learning model training) requires considerable computational power.
- **Impact**: Data analysis can take a long time and may result in bottlenecks, especially if the data cannot be processed in parallel or distributed across multiple nodes.
- **Solution**:
    - **Distributed Computing**: Use distributed frameworks like **Apache Spark**, **Dask**, or **Hadoop** that can distribute tasks across multiple nodes and process the data in parallel.
    - **Sampling**: If full analysis isn't feasible, use sampling techniques to work with a subset of the data.
    - **GPU Acceleration**: Use GPUs to speed up computation for large-scale machine learning and deep learning tasks.

b) Modeling and Scalability

- **Challenge**: Machine learning models might struggle with very large datasets in terms of both training time and the ability to scale.
- **Impact**: The time to train machine learning models on large datasets can become prohibitive. Furthermore, not all machine learning algorithms are optimized for

large-scale data, and some may require adjustments to handle them.

- **Solution**:
  - **Mini-Batch Processing**: Use algorithms that support mini-batch training (e.g., **Stochastic Gradient Descent**, **Neural Networks**).
  - **Distributed Machine Learning**: Use frameworks like **Apache Spark MLlib**, **TensorFlow**, or **PyTorch** with distributed training capabilities.

c) Data Shuffling and Random Access

- **Challenge**: For machine learning, random access to large datasets and the ability to shuffle data efficiently are important for model training (to avoid overfitting).
- **Impact**: Large datasets are difficult to shuffle efficiently in memory, and loading random samples may require specialized techniques.
- **Solution**:
  - **Data Generators**: Use data generators that allow you to load data in batches (e.g., **Keras** data generators).
  - **Indexing and Preprocessing**: Preprocess and index data in a way that enables fast access during training.

4. Security and Privacy Concerns

a) Data Security

- **Challenge**: Large datasets may contain sensitive personal or business data. Managing access and ensuring secure storage is vital.
- **Impact**: Handling sensitive data in large quantities increases the risk of breaches, which can have legal and ethical consequences.
- **Solution**:
  - **Encryption**: Encrypt sensitive data both at rest and in transit.
  - **Access Control**: Implement strict access control policies, including role-based access and auditing.

b) Privacy Issues

- **Challenge**: Large datasets often involve personal or confidential data, making it difficult to ensure compliance with privacy regulations (e.g., GDPR, HIPAA).
- **Impact**: Processing and storing data in compliance with privacy laws can be complex, especially when working with large datasets.
- **Solution**:
  - **Data Anonymization**: Anonymize or pseudonymize sensitive data before processing or analysis.

	o **Compliance Frameworks**: Implement frameworks and policies to ensure compliance with data protection laws					
(OR)						
16 b	Explain the concept of data wrangling and discuss the key steps involved in the data wrangling process and the importance of each step. **Data wrangling** (also known as **data munging**) is the process of transforming and mapping raw data into a more useful and accessible format for analysis. It involves cleaning, restructuring, and enriching raw data from various sources to make it suitable for analysis and decision-making. Data wrangling is often considered one of the most time-consuming and important tasks in the data analysis pipeline. **Key Steps in the Data Wrangling Process** 1. **Data Collection** o **Description**: Gathering data from various sources (e.g., databases, flat files like CSV, JSON, XML, APIs, web scraping, or sensor data). o **Importance**: This is the foundational step where the raw data is gathered. It sets the stage for all subsequent steps in the wrangling process. o **Challenges**: Data could be incomplete, in inconsistent formats, or in a form that is difficult to analyze. o **Tools**: APIs, web scraping tools (e.g., BeautifulSoup), SQL queries, data import functions in libraries (e.g., `pandas.read_csv()`). 2. **Data Inspection/Exploration** o **Description**: Inspecting the dataset to understand its structure, content, and identify any problems such as missing values, duplicates, or incorrect formats. o **Importance**: This step helps to get a feel for the data and ensures that any issues or anomalies are identified before any transformations are done. o **Challenges**: Data might be large, unstructured, or might contain inconsistencies that are hard to detect manually. o **Tools**: `pandas` (e.g., `df.info()`, `df.describe()`, `df.head()`), `matplotlib`, `seaborn` (for visualization), or any other exploratory data analysis (EDA) tool. 3. **Data Cleaning** o **Description**: Removing or correcting any errors in the data, such as missing values, duplicates, inconsistent data types, or outliers. o **Importance**: Cleaning ensures the accuracy and quality of the data. Poor-quality data can lead to misleading results in analysis or modeling.	10	3	3	5	

   - **Challenges**: Dealing with missing values, correcting inconsistent data entries, handling noisy data.
   - **Tools**: `pandas` (`fillna()`, `dropna()`, `drop_duplicates()`, `astype()`), `numpy` (e.g., `np.nan` for missing values).
4. **Data Transformation**
   - **Description**: Transforming the data into a more suitable format for analysis. This may involve normalizing or scaling numerical values, converting categorical variables to numerical ones, or reshaping the data.
   - **Importance**: Transformations help prepare the data for various types of analysis or modeling. Some algorithms require data to be in a specific format (e.g., scaling for neural networks).
   - **Challenges**: Applying the right transformations can be complex, especially with heterogeneous data types (e.g., mixing categorical and numerical data).
   - **Tools**: `pandas` (e.g., `pd.get_dummies()` for one-hot encoding, `StandardScaler` from `sklearn` for scaling), `numpy` for mathematical transformations.
5. **Data Integration**
   - **Description**: Combining data from multiple sources or datasets, ensuring that the combined data is consistent and compatible.
   - **Importance**: Many datasets are spread across different sources. Integration allows data from these sources to be merged into a single dataset for analysis.
   - **Challenges**: Merging datasets may introduce discrepancies (e.g., mismatched keys, inconsistent formats) that need to be resolved.
   - **Tools**: `pandas` (e.g., `merge()`, `concat()`), SQL join operations, or using ETL tools for larger datasets.
6. **Data Enrichment**
   - **Description**: Enhancing the dataset with additional information, such as external data sources or creating new features.
   - **Importance**: Enriching the data helps improve the quality and comprehensiveness of the dataset, allowing for more insightful analysis.
   - **Challenges**: Adding external data can introduce its own inconsistencies or issues like missing values.
   - **Tools**: APIs, web scraping, and additional datasets from open data repositories.
7. **Data Formatting**
   - **Description**: Converting data into the required format, such as ensuring that numerical columns are numeric and categorical columns are properly labeled.

- **Importance**: Correct formatting is essential for the subsequent steps in the analysis or modeling pipeline.
- **Challenges**: Ensuring all columns are consistently formatted, especially when dealing with large datasets with diverse data types.
- **Tools**: `pandas` for type casting (e.g., `df['column'].astype(int)`), `str` functions for string manipulation.

8. **Data Sampling/Resampling (if needed)**
   - **Description**: Reducing the dataset size by sampling a subset of data (if the dataset is too large) or balancing the dataset (e.g., in classification problems with imbalanced classes).
   - **Importance**: Sampling can reduce the computational complexity and speed up the analysis, while resampling ensures that models are not biased due to class imbalances.
   - **Challenges**: Ensuring that the sample is representative of the full dataset and that resampling does not distort the underlying patterns.
   - **Tools**: `pandas` (e.g., `df.sample()`), `imblearn` for oversampling/undersampling.
9. **Data Validation**
   - **Description**: Ensuring that the cleaned, transformed, and integrated data meets the requirements of the analysis or machine learning models.
   - **Importance**: Validation ensures that the dataset is accurate, complete, and ready for use in the next stage of analysis or modeling.
   - **Challenges**: Performing robust validation, especially with large datasets, can be difficult and time-consuming.
   - **Tools**: Manual checks, statistical methods, or automated validation scripts.

The Importance of Each Step in Data Wrangling

1. **Data Collection**: Ensures the right data is obtained for analysis.
2. **Data Inspection/Exploration**: Helps uncover patterns and potential issues with the data early on.
3. **Data Cleaning**: Improves data quality, preventing errors from affecting the analysis.
4. **Data Transformation**: Prepares the data in a format suitable for modeling and analysis.
5. **Data Integration**: Combines disparate data sources, enabling a more comprehensive analysis.
6. **Data Enrichment**: Enhances the dataset with additional useful information, improving the depth of the analysis.
7. **Data Formatting**: Ensures the data is in a consistent format, which is essential for correct processing and analysis.

8. **Data Sampling/Resampling**: Helps manage large datasets and deal with class imbalances for more accurate and efficient analysis.
9. **Data Validation**: Ensures the data meets quality and consistency requirements before further processing.

Challenges in Data Wrangling

- **Data Volume**: Large datasets can make it difficult to perform operations like cleaning, transformation, and validation efficiently.
- **Data Quality**: Handling missing values, duplicates, and inconsistent data can be time-consuming.
- **Data Compatibility**: Different datasets may have different formats or schemas, requiring substantial work to merge or integrate them.

17 a	i) Write a Python program to Create a pie chart using Matplotlib showing the percentage distribution of students enrolled in different courses (e.g., Python, Java, C++, AI).	10	2	4	5	

```
import matplotlib.pyplot as plt

# Data for the pie chart: Course names and the number
of students
courses = ['Python', 'Java', 'C++', 'AI']
students = [150, 120, 90, 60]

# Create a pie chart
plt.figure(figsize=(7,7))
plt.pie(students, labels=courses, autopct='%1.1f%%',
startangle=140,
colors=['#ff9999','#66b3ff','#99ff99','#ffcc99'])

# Title of the pie chart
plt.title('Percentage Distribution of Students
Enrolled in Different Courses')

# Display the pie chart
plt.show()
```

Explanation:

- **Data**: The `students` list represents the number of students enrolled in each course (Python, Java, C++, AI).
- **Pie Chart**: The `plt.pie()` function is used to create the pie chart. The `autopct='%1.1f%%'` argument displays the percentage values on the chart with one decimal point.
- **Colors**: The `colors` argument is used to customize the colors of each segment of the pie chart.
- **Title**: `plt.title()` sets the title of the chart.

Output:

This will display a pie chart showing the percentage distribution of students in the Python, Java, C++, and AI courses.
Let me know if you'd like any adjustments or further explanation on any part!

ii) Write a Python program to draw a simple line graph using Matplotlib to represent the number of visitors to a website over 7 days.

Here's a Python program that draws a simple **line graph** using **Matplotlib** to represent the number of visitors to a website over 7 days:

```
import matplotlib.pyplot as plt

# Data for the line graph: Days of the week and number
of visitors
days = ['Monday', 'Tuesday', 'Wednesday', 'Thursday',
'Friday', 'Saturday', 'Sunday']
visitors = [120, 150, 170, 140, 160, 190, 200]

# Create a line graph
plt.plot(days, visitors, marker='o', linestyle='-',
color='b')

# Title and labels
plt.title('Website Visitors Over 7 Days')
plt.xlabel('Days of the Week')
plt.ylabel('Number of Visitors')

# Display the line graph
plt.grid(True)
plt.show()
```

Explanation:

- **Data**: The `days` list represents the days of the week, and the `visitors` list represents the number of visitors to the website for each corresponding day.
- **Line Graph**: The `plt.plot()` function is used to plot the line graph.
  - o `marker='o'` adds a marker at each data point (a circle in this case).
  - o `linestyle='-'` ensures that the points are connected with a line.
  - o `color='b'` sets the line color to blue.
- **Title and Labels**: `plt.title()`, `plt.xlabel()`, and `plt.ylabel()` are used to set the title and axis labels.
- **Grid**: `plt.grid(True)` adds a grid to the graph to make it easier to read the values.

Output:

This will display a line graph representing the number of visitors to a website over the span of 7 days (Monday to Sunday).
Let me know if you need any further adjustments or explanations!

(OR)

17 b	i) Define Seaborn? How does it differ from Matplotlib? Write a Python program to draw a scatter plot using	10	3	5	5	

Seaborn showing the relationship between height and weight of individuals.

What is Seaborn?

Seaborn is a Python visualization library built on top of Matplotlib. It provides a high-level interface for drawing attractive and informative statistical graphics. Seaborn comes with several built-in themes and color palettes that make it easy to generate aesthetically pleasing plots with minimal code.

Differences between Seaborn and Matplotlib

1. **Ease of Use**:
   - **Matplotlib**: While powerful and highly customizable, Matplotlib requires more lines of code to generate common statistical plots. It is great for creating basic and complex plots but can be verbose.
   - **Seaborn**: It is built to simplify the process of creating complex visualizations, especially for statistical data. It provides high-level functions that automatically handle many details, such as axes labels, legends, color schemes, etc.
2. **Style and Aesthetics**:
   - **Matplotlib**: While Matplotlib can generate a wide range of plots, the default style is relatively basic. Customizing the appearance (e.g., changing colors, themes) requires extra work.
   - **Seaborn**: It comes with built-in themes, color palettes, and automatic formatting, making it much easier to generate more visually appealing plots with minimal customization.
3. **Statistical Plotting**:
   - **Matplotlib**: It is primarily focused on general plotting but does not offer built-in support for statistical visualizations (e.g., heatmaps, regression plots).
   - **Seaborn**: It includes specialized functions for creating statistical plots like regression plots, box plots, violin plots, and heatmaps, making it ideal for exploratory data analysis.
4. **Integration with Pandas**:
   - **Matplotlib**: While it can work with Pandas DataFrames, it doesn't provide direct support for DataFrame operations.
   - **Seaborn**: It works seamlessly with Pandas DataFrames and provides functions that directly accept DataFrame columns as input.

Python Program: Scatter Plot using Seaborn

Here’s a Python program to draw a scatter plot using Seaborn to show the relationship between the height and weight of individuals:

```
import seaborn as sns
import matplotlib.pyplot as plt

# Sample data for height and weight
data = {
    'Height': [5.5, 6.1, 5.8, 5.9, 6.0, 5.4, 5.7, 6.2,
5.6, 5.8],
    'Weight': [150, 180, 165, 170, 175, 160, 155, 185,
168, 162]
}

# Create a DataFrame from the data
import pandas as pd
df = pd.DataFrame(data)

# Create a scatter plot using Seaborn
sns.scatterplot(x='Height', y='Weight', data=df)

# Title and labels
plt.title('Scatter Plot: Height vs Weight')
plt.xlabel('Height (in feet)')
plt.ylabel('Weight (in lbs)')

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

Explanation:

1. **Data**: The `data` dictionary contains two lists: `Height` and `Weight`, which represent the height (in feet) and weight (in pounds) of individuals.
2. **DataFrame**: The `pd.DataFrame(data)` converts the dictionary into a Pandas DataFrame, making it easy to work with Seaborn.
3. **Scatter Plot**: The `sns.scatterplot()` function is used to create a scatter plot. The `x` and `y` arguments specify which columns to plot on the x and y axes, and the `data` argument specifies the DataFrame to use.

ii) Write a Python program using Seaborn to create a histogram that displays the distribution of students' exam scores. Customize the bin size and add color.

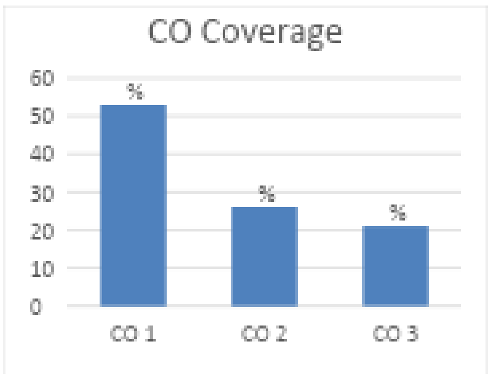
Here's a Python program using **Seaborn** to create a **histogram** that displays the distribution of students' exam scores, with customized bin size and color:

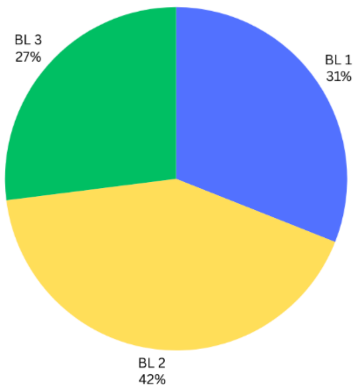
```
import seaborn as sns
import matplotlib.pyplot as plt

# Sample data: Exam scores of students
exam_scores = [95, 88, 76, 85, 92, 99, 78, 85, 93, 89,
84, 91, 77, 80, 94, 79, 87, 82, 83, 90]

# Create a histogram using Seaborn
sns.histplot(exam_scores, bins=8, kde=False,
color='skyblue', edgecolor='black')

# Title and labels
plt.title('Distribution of Students\' Exam Scores')
plt.xlabel('Exam Scores')
plt.ylabel('Frequency')

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

Explanation:

1. **Data**: The list `exam_scores` contains the exam scores of 20 students.
2. **`sns.histplot()`**:
   - `bins=8`: This customizes the number of bins in the histogram to 8.
   - `kde=False`: Disables the Kernel Density Estimate (KDE) plot, which would otherwise show a smooth curve over the histogram.
   - `color='skyblue'`: Sets the color of the bars to a light blue.
   - `edgecolor='black'`: Adds a black border to the histogram bars for better visibility.
3. **Title and Labels**: The `plt.title()`, `plt.xlabel()`, and `plt.ylabel()` functions are used to set the title and axis labels for the plot.
4. **Display**: `plt.show()` displays the histogram.

Course Outcome (CO) and Bloom's level (BL) Coverage in Questions:


CO Coverage
CO 1
CO 2
CO 3


BL 3
BL 1
BL 2