Data analytics (ACSDS0603)

UNIT-3 NOTES

Data preprocessing is a crucial step in data analytics, ensuring that raw data is cleaned, transformed, and structured for effective analysis. The main forms of data preprocessing include:

1. Data Cleaning

- Handling missing values (e.g., imputation, deletion)
- Removing duplicates
- Correcting errors and inconsistencies
- Handling outliers (e.g., using Z-score, IQR)

2. Data Integration

- Combining data from multiple sources
- Handling schema mismatches and entity resolution
- Merging datasets for a unified analysis

3. Data Transformation

- **Normalization:** Scaling values between a fixed range (e.g., Min-Max Scaling, Z-score normalization)
- **Standardization:** Converting data to have a mean of 0 and a standard deviation of 1
- **Encoding categorical data:** One-hot encoding, Label encoding
- **Feature engineering:** Creating new meaningful features from existing ones

4. Data Reduction

- **Dimensionality reduction:** PCA, t-SNE, LDA
- **Feature selection:** Selecting relevant features to reduce complexity
- **Sampling:** Random sampling, Stratified sampling

5. Data Discretization and Binning

- Converting continuous variables into categorical bins (e.g., age groups: young, middle-aged, senior)

- Helps in handling skewed distributions

Data Cleaning Techniques in Data Analytics

Data cleaning is a crucial step in data preprocessing to ensure that the dataset is accurate, consistent, and ready for analysis. Below are the key techniques used for cleaning data:

---

1. Handling Missing Data

Missing values can distort analysis and machine learning models. Common methods to handle them include:

- **Removal:** Delete rows or columns with missing values (only if the missing data is minimal).
- **Imputation:** Fill in missing values using:
    - Mean, median, or mode (for numerical data)
    - Forward fill or backward fill (for time-series data)
    - Predictive imputation using machine learning models (e.g., KNN imputer)

---

2. Handling Duplicate Data

- Identify duplicate records using pandas:

```
df.duplicated().sum()  # Count duplicates

df.drop_duplicates(inplace=True)  # Remove duplicates
```

- Ensure unique identifiers (e.g., IDs) do not have duplicates.

---

3. Handling Outliers

Outliers can skew the results. Methods to handle them:

- **Z-score Method:** Remove values that are too many standard deviations from the mean.
- **IQR Method:** Remove values outside the Interquartile Range (IQR).

```
Q1 = df['column'].quantile(0.25)

Q3 = df['column'].quantile(0.75)
```

```
IQR = Q3 - Q1
df = df[(df['column'] >= (Q1 - 1.5 * IQR)) & (df['column'] <= (Q3 + 1.5 * IQR))]
```

- **Capping/Winsorization:** Replace extreme values with upper/lower threshold values.

---

4. Standardizing Data Formats

Ensure consistency in data formats:

- Convert dates to a uniform format:

```
df['date_column'] = pd.to_datetime(df['date_column'])
```

- Standardize text case (e.g., lowercase all entries):

```
df['column'] = df['column'].str.lower()
```

- Remove special characters in categorical data:

```
df['column'] = df['column'].str.replace('[^A-Za-z0-9 ]+', '', regex=True)
```

---

5. Fixing Data Inconsistencies

- Check for spelling errors or inconsistent labels:

```
df['column'].unique()
```

- Replace incorrect values:

```
df['column'].replace({'wrong_value': 'correct_value'}, inplace=True)
```

---

6. Handling Categorical Data Issues

- Remove leading/trailing spaces in text:

```
df['column'] = df['column'].str.strip()
```

- Encode categorical values (if required for modeling):

```
df = pd.get_dummies(df, columns=['categorical_column'])
```

---

7. Handling Inconsistent Units & Scaling

- Convert units to a common standard (e.g., kg to grams, USD to EUR).
- Normalize or standardize numerical data:

```
from sklearn.preprocessing import MinMaxScaler

scaler = MinMaxScaler()

df[['column']] = scaler.fit_transform(df[['column']])
```

---

8. Handling Structural Errors

- Check column names for typos and standardize them.
- Ensure data types are correct:

```
df['column'] = df['column'].astype('int')  # Convert to integer
```

Data Transformation

1.Data Normalization

Normalization scales numeric data to a fixed range, usually [0,1] or [-1,1], ensuring that all features contribute equally to the model.

Min-Max Scaling (0 to 1)

$X' = X - X_{min} / (X_{max} - X_{min})$

- X is the original value,
- X_{min} is the minimum value in the dataset,
- X_{max} is the maximum value in the dataset,
- X' is the scaled value.

```
from sklearn.preprocessing import MinMaxScaler

import numpy as np


# Sample data

data = np.array([[10], [20], [30], [40], [50]])


# Initialize MinMaxScaler

scaler = MinMaxScaler(feature_range=(0, 1))

scaled_data = scaler.fit_transform(data)
```

print(scaled_data)

[[0.]

[0.25]

[0.5]

[0.75]

[1.]]

Z-score Normalization (Standardization)

$X' = (X - \mu)/\sigma$

Where:

- X = Individual data point
- μ = Mean of the dataset
- σ= Standard deviation of the dataset

```
import numpy as np
import pandas as pd
from scipy.stats import zscore
# Sample dataset
data = {'Values': [50, 55, 53, 58, 54, 52, 110, 49, 56, 51]}
df = pd.DataFrame(data)
 # Calculate Z-score
 df['Z-score'] = zscore(df['Values'])
print(df)
```

```
   Values   Z-score
0      50 -0.49
1      55  0.06
2      53 -0.17
3      58  0.37
4      54 -0.06
5      52 -0.28
```

6   110 3.72 # Outlier

7   49 -0.61

8   56 0.17

9   51 -0.39

110 has a Z-score of 3.72, meaning it is an outlier.

2. Data Discretization (Binning)

Converts continuous variables into discrete bins.

Equal-Width Binning

Divides data into equal-sized bins.

df['binned_column'] = pd.cut(df['column'], bins=5, labels=False)

Equal-Frequency Binning

Each bin has approximately the same number of data points.

df['binned_column'] = pd.qcut(df['column'], q=5, labels=False)

Data Discretization Using Pandas

Let's apply **Equal-Width Binning** and **Equal-Frequency Binning** to a dataset.

Step 1: Create a Sample Dataset

```
import pandas as pd

import numpy as np


# Generate a sample dataset with continuous numerical values

np.random.seed(42)

df = pd.DataFrame({'Age': np.random.randint(18, 80, 15)})

print(df)
```

Sample Output:

```
   Age
0   22
1   34
2   57
3   45
```

4 29

5 76

6 62

7 19

8 43

9 31

10 50

11 70

12 27

13 38

14 65

Step 2: Apply Equal-Width Binning

Create 3 equal-width bins

df['Age_Binned_Equal_Width'] = pd.cut(df['Age'], bins=3, labels=["Young", "Middle-aged", "Senior"])

print(df[['Age', 'Age_Binned_Equal_Width']])

- This method divides the data range into 3 equal-width bins.
- Labels are assigned based on the bin ranges.

Example Output:

	Age	Age_Binned_Equal_Width
0	22	Young
1	34	Young
2	57	Middle-aged
3	45	Middle-aged
4	29	Young
5	76	Senior
6	62	Senior
7	19	Young

8	43	Middle-aged
9	31	Young
10	50	Middle-aged
11	70	Senior
12	27	Young
13	38	Young
14	65	Senior

- Here, **"Young" (18-38)**, **"Middle-aged" (38-58)**, and **"Senior" (58-78)** are the categories.

Step 3: Apply Equal-Frequency Binning

```
# Create 3 bins with approximately equal number of samples
df['Age_Binned_Equal_Freq'] = pd.qcut(df['Age'], q=3, labels=["Young", "Middle-aged", "Senior"])
print(df[['Age', 'Age_Binned_Equal_Freq']])
```

- This method ensures that each bin has (approximately) the same number of data points.

Example Output:

	Age	Age_Binned_Equal_Freq
0	22	Young
1	34	Young
2	57	Middle-aged
3	45	Middle-aged
4	29	Young
5	76	Senior
6	62	Senior
7	19	Young
8	43	Middle-aged
9	31	Young
10	50	Middle-aged
11	70	Senior

12	27	Young
13	38	Middle-aged
14	65	Senior

- The bins here **do not have fixed width** but ensure a nearly equal number of observations.

Data Integration in Data Analytics

Data Integration is the process of combining data from multiple sources into a unified and consistent format to enable efficient analysis and decision-making.

Types of Data Integration Techniques

1. Manual Data Integration

- Involves manually merging datasets from different sources.
- Suitable for small datasets but inefficient for large-scale integration.

2. ETL (Extract, Transform, Load)

- **Extract**: Data is collected from various sources.
- **Transform**: Data is cleaned, formatted, and standardized.
- **Load**: Transformed data is stored in a database or data warehouse.

Example:

```
import pandas as pd

# Load data from different sources
df1 = pd.read_csv("sales_data.csv")
df2 = pd.read_excel("customer_data.xlsx")


# Merge the datasets
df = pd.merge(df1, df2, on="Customer_ID", how="inner")
print(df.head())
```

---

3. Data Warehousing

- Stores data from different sources in a central repository.
- Used for large-scale analytics (e.g., Amazon Redshift, Google BigQuery).

---

4. Data Virtualization

- Provides real-time access to multiple data sources without physically moving data.
- Examples: Denodo, IBM Cloud Pak.

---

5. Schema Integration

- Combines data with different formats or structures into a common schema.
- Used when merging relational databases.

---

6. Record Linkage (Entity Resolution)

- Identifies and merges duplicate records from different datasets.

Example using Python:

```
import recordlinkage

indexer = recordlinkage.Index()
indexer.block('name')  # Blocking on 'name' column

# Compare datasets
compare = recordlinkage.Compare()
compare.string('name', 'name', method='jarowinkler', threshold=0.85)

# Generate matches
matches = compare.compute(indexer.index(df1, df2), df1, df2)
print(matches)
```

---

7. API-Based Integration

- Retrieves real-time data from web services or cloud sources.

Example:

```
import requests

response = requests.get("https://api.example.com/data")
data = response.json()
df = pd.DataFrame(data)
```

Challenges in Data Integration

1. **Schema Mismatch**: Different column names and formats.
2. **Duplicate Data**: Repetitive records across sources.

3. **Data Inconsistency**: Variations in data formats.
4. **Scalability**: Handling large datasets efficiently.

Data Reduction Techniques

(a) Dimensionality Reduction

Reduces the number of features while retaining most of the information.

- **Principal Component Analysis (PCA)**:
  - Transforms features into a new set of uncorrelated variables (principal components).
  - Useful for high-dimensional datasets.
- **Linear Discriminant Analysis (LDA)**:
  - Similar to PCA but optimizes for class separability in classification problems.
- **Feature Selection Methods**:
  - **Filter Methods** (Correlation, Mutual Information)
  - **Wrapper Methods** (Recursive Feature Elimination)

- **Embedded Methods** (LASSO, Decision Trees)

Principal Component Analysis (PCA) is a dimensionality reduction technique used to transform high-dimensional data into a lower-dimensional space while retaining as much variance as possible. It works by finding a set of new orthogonal axes (principal components) that maximize the variance in the data.

Key Concepts of PCA:

1. **Standardization** – Data is normalized (zero mean, unit variance).
2. **Covariance Matrix Computation** – Finds relationships between features.
3. **Eigen Decomposition** – Extracts eigenvalues and eigenvectors from the covariance matrix.
4. **Principal Components Selection** – Top components (based on eigenvalues) are chosen to reduce dimensions.
5. **Projection** – Data is transformed into the new feature space.

Why Use PCA?

- Reduces **computational cost** and memory usage.
- Helps in **removing multicollinearity** in datasets.
- Improves **visualization** of high-dimensional data.
- Enhances **model performance** by focusing on the most relevant information.

Step-by-Step Working of PCA

1. Standardization of Data (Preprocessing)

Before applying PCA, the dataset should be standardized so that features have **zero mean** and **unit variance** to prevent large-value features from dominating. Mathematically, for each feature:

$X_{scaled} = X - \mu / \sigma$

where:

- X is the original feature value.
- μ is the mean of the feature.
- σ is the standard deviation of the feature.

2. Compute Covariance Matrix

- The covariance matrix captures the relationships between different features. It helps to understand how features vary together.
  For two features X_1 and X_2, covariance is given by:

$$\mathrm{Cov}(X_1, X_2) = \frac{1}{n}\sum_{i=1}^{n}(X_{1i} - \bar{X}_1)(X_{2i} - \bar{X}_2)$$

3. Compute Eigenvalues and Eigenvectors

- **Eigenvalues**: Measure the variance explained by each principal component.
- **Eigenvectors**: Represent the new axes in transformed space.

$$Cv = \lambda v$$

where:

- λ are the eigenvalues.
- v are the eigenvectors.

4. Sort Eigenvalues and Select Principal Components

- Eigenvalues are sorted in descending order.
- The corresponding top eigenvectors form the **principal components**.

Example: If we choose **top 2 components**, we retain most of the variance.

5. Project Data onto New Feature Space

The original dataset X is projected onto the selected principal components:

$$X_{\text{new}} = X_{\text{scaled}} \times V_k$$

where V_k contains the top k eigenvectors.

(b) Numerosity Reduction

Stores data in a compressed form instead of raw data.

- **Sampling**:

- Selects a subset of data points (e.g., random sampling, stratified sampling).

- **Regression Models**:
  - Replaces actual data with a mathematical model (e.g., linear regression, polynomial regression).
- **Histograms**:
  - Summarizes data by grouping values into bins.
- **Clustering**:
  - Groups similar data points and stores cluster representatives (e.g., K-Means).

(c) Data Compression

Encodes data in a compact format.

- **Lossless Compression** (e.g., Huffman Coding, Run-Length Encoding)
  - No information loss, reversible.
- **Lossy Compression** (e.g., JPEG, MP3)
  - Reduces storage size by removing less important details.

Use Cases

✅ Speeding up machine learning models
✅ Reducing memory and storage requirements
✅ Improving model interpretability

Data Cube Aggregation

Data cube aggregation is a multi-dimensional summarization technique used in Online Analytical Processing (OLAP). It enables fast query execution by precomputing aggregations at different levels.

(a) Concept of Data Cubes

A **data cube** organizes data along multiple dimensions, allowing aggregated analysis at various levels.

- Example: **Sales Data Cube** with dimensions:

- Time (Year → Quarter → Month → Week)
- Location (Country → State → City → Store)
- Product (Category → Subcategory → Item)

(b) Aggregation Types

- **Roll-Up (Drill-Up)**: Aggregates data to a higher level (e.g., sum monthly sales to get yearly sales).
- **Drill-Down**: Moves from higher-level aggregation to detailed data (e.g., yearly sales → monthly sales).
- **Slice**: Extracts a subset along one dimension (e.g., sales in 2023).
- **Dice**: Extracts a subset along multiple dimensions (e.g., sales of electronics in Q1 2023).
- **Pivot**: Rotates dimensions for different perspectives.

(c) Benefits

✅ Reduces query processing time by precomputing summaries.
✅ Optimizes storage by storing aggregated values instead of raw data.
✅ Enhances data analysis efficiency in business intelligence.

Data cube aggregation, or roll-up, involves summarizing data across multiple dimensions to create a higher-level view, like aggregating daily sales to monthly or yearly totals. For example, you might aggregate sales data by region, product category, and time period to create a cube showing total sales for each region, product, and year.

Here's a more detailed explanation and example:

What is a Data Cube?

- A data cube is a multidimensional structure used in Online Analytical Processing (OLAP) to store and analyze data.
- It organizes data along multiple dimensions (like time, location, product, etc.) and measures (like sales, revenue, etc.).
- Each cell in the cube represents a specific combination of dimensions and holds a corresponding measure value.

What is Data Cube Aggregation (Roll-up)?

- Data cube aggregation (also called roll-up or drill-up) is the process of summarizing data from a lower level of detail to a higher level.

- This is achieved by combining data across multiple dimensions to create aggregated values.
- For example, you can roll up daily sales data to monthly sales data, or monthly sales data to quarterly sales data.

Example:

Imagine you have sales data for a company, with dimensions of:

- **Product Category:** (e.g., "Electronics", "Clothing", "Food")
- **Region:** (e.g., "North", "South", "East", "West")
- **Time Period:** (e.g., "Daily", "Monthly", "Quarterly", "Yearly")
- **Measure:** (e.g., "Total Sales")

Here's how data cube aggregation (roll-up) would work:

1. **Initial Data:**

You start with detailed sales data, like daily sales for each product category and region.

2. **Roll-up by Time:**

You can roll up the daily sales data to monthly sales data by summing the sales for each product category and region within each month.

3. **Roll-up by Region:**

You can roll up the monthly sales data to regional sales data by summing the sales for each product category within each region.

4. **Roll-up by Product Category:**

You can roll up the regional sales data to product category sales data by summing the sales for each region within each product category.

5. **Final Cube:**

The resulting data cube would contain aggregated sales data, showing total sales for each product category, region, and time period.

Benefits of Data Cube Aggregation:

- **Simplified Analysis:** Aggregated data is easier to analyze and understand than raw data.
- **Improved Performance:** Aggregating data can improve query performance, especially for large datasets.
- **Trend Identification:** Aggregating data helps identify trends and patterns in the data.

- **Data Reduction:** Aggregation helps reduce the volume of data by summarizing it along different dimensions, making it easier to analyze and visualize trends, patterns, and relationships.