Unit – II

Handling Missing Data and Data Encoding

Syllabus
Impute missing data: - Interpretation of missing data, handling missing data - mean, mode, median, min, max, forward fill, backward fill, remove missing data.
Data Encoding: - Significance of data encoding, Types of encoding techniques - one hot encoding, ordinal encoding, label encoding, mean encoding

2.1. Introduction

- ✓ Handling missing data is a fundamental part of Exploratory Data Analysis (EDA**)** because almost every real-world dataset contains some degree of incomplete information.
- ✓ Whether due to data collection errors, system failures, or human factors, missing data can bias results and reduce the quality of your analysis or model if not handled properly.

2.2. Why Missing Data Matters

2.1.1. Bias in Results

When data is missing in a **non-random way**, it can introduce bias.

- **Example**: Suppose you're analyzing the relationship between income and spending habits, but many high-income individuals didn't report their income. If you ignore or drop these rows, your analysis may **underestimate the true average income** and lead to **false conclusions**.
- **Impact**: Your model might assume, for instance, that most customers have low income, leading to poor segmentation or targeting strategies.

2.2.2. Loss of Valuable Information

- Dropping rows or columns with missing values can lead to a significant **reduction in dataset size**.
- This is especially problematic when:
  - You have a small dataset.
  - The missing data is spread across multiple important columns.
- **Example**: Dropping 20% of rows from a dataset with only 500 records means losing 100 records — which could carry important insights.

2.2.3. Incompatibility with Machine Learning Algorithms

Many ML algorithms **do not support missing values** as inputs:

- **Fails with missing values**:
  - Linear regression
  - Logistic regression
  - SVM
  - KNN (default implementations)

- **Works with missing values** (some tree-based models like XGBoost, LightGBM handle them natively):
  - But even then, **explicit handling is usually better**.

If not addressed, models may crash or silently perform poorly.

2.2.4. Invalid Model Evaluation

- Missing data can affect the way **train-test splits** are made.
- If the missing data is **distributed differently** across train and test sets, it can lead to:
  - Overestimating performance
  - Underestimating error
  - Bad generalization

2.2.5. Misleading Visualizations

Charts and plots (like histograms, scatterplots, boxplots) may **misrepresent the data** if missing values are not accounted for.

- For example, plotting a histogram of `Age` without filling or removing missing values might show a distorted distribution — especially if missing data is not random.

2.3. Interpretation of missing data

Interpreting missing data correctly is crucial before deciding how to handle it. This means not just noticing that data is missing, but understanding *why* **it's** missing**,** *where* it's missing, and *what that implies* about data and your analysis.

2.3.1. Look at the Missingness Quantitatively

```
# Number of missing values per column
df.isnull().sum()


# Percentage of missing values
df.isnull().mean() * 100
```

Interpretation:

- If a column has > 50% missing values, it might be a candidate for dropping — unless it's critical.
- A column with < 5% missing values might be suitable for simple imputation.

2.3.2. Visualize the Missing Data

Use tools to detect patterns or clusters of missingness.

```python
import seaborn as sns
sns.heatmap(df.isnull(), cbar=False)

# Or use missingno library
import missingno as msno
msno.matrix(df)
msno.heatmap(df)
```

Interpretation:

- If missing data is clustered together (e.g., all missing in the same rows or after a certain time), it might indicate a **systematic issue** (e.g., data not collected under certain conditions).
- A missingness heatmap can also reveal whether **two or more columns tend to be missing together**, which may suggest a dependent pattern.

2.3.3. Check Missingness by Groups

Examine whether missing data is more common in certain subgroups of the dataset.

```python
df.groupby(df['Age'].isnull())['Survived'].mean()
```

Interpretation:

- If missing data correlates with the target variable (e.g., survival, default, churn), it could indicate **informative missingness** — you may want to **preserve** this pattern using an indicator column.

2.3.4. Determine the Type of Missingness

Understanding the mechanism of missing data helps choose the right handling strategy:

Type	Definition	Example	Interpretation
MCAR (Missing Completely At Random)	Missing values are unrelated to any other data	Random survey skip	Can safely delete or impute
MAR (Missing At Random)	Missing values depend on other observed variables	Income missing for young people	Impute based on other features
MNAR (Missing Not At Random)	Missing values depend on unobserved data or the value itself	People with high income hide it	Needs domain knowledge; hard to handle

2.4. Handling missing data

- ✓ Handling missing data is a critical step in data analysis because incomplete data can lead to biased results and reduce the accuracy of models.
- ✓ Properly identifying and addressing missing values ensures that analyses are reliable and insights are valid.
- ✓ Various techniques like deletion, imputation, and modeling are used depending on the nature and extent of the missing data.

Consider below dataset on which apply various imputation techniques

#	Age	Gender
0	25.0	M
1	NaN	F
2	30.0	NaN
3	22.0	F
4	NaN	M
5	28.0	NaN

2.4.1. Mean Imputation

Replace missing Age values with the **mean**:

```
mean_age = df['Age'].mean()
df['Age'].fillna(mean_age, inplace=True)
print(df)
```

#	Age	Gender
0	25.0	M
1	***26.25***	F
2	30.0	NaN
3	22.0	F
4	**26.25**	M
5	28.0	NaN

(Mean Age = (25 + 30 + 22 + 28) / 4 = 26.25)

2.4.2. Median Imputation

Replace missing Age with the **median**:

```
df['Age'] = [25, np.nan, 30, 22, np.nan, 28]  # reset data
median_age = df['Age'].median()
df['Age'].fillna(median_age, inplace=True)
print(df)
```

OUTPUT

#	Age	Gender
0	25.0	M
1	***26.5***	F
2	30.0	NaN
3	22.0	F
4	***26.5***	M
5	28.0	NaN

2.4.3. Mode Imputation

Replace missing Gender with the **mode** (most frequent):

```
mode_gender = df['Gender'].mode()[0]
df['Gender'].fillna(mode_gender, inplace=True)
print(df)
```

#	Age	Gender
0	25.0	M
1	26.5	F
2	30.0	F
3	22.0	F
4	26.5	M
5	28.0	F

2.4.4. Min Imputation

Replace missing Age with **minimum** value:

```
df['Age'] = [25, np.nan, 30, 22, np.nan, 28]  # reset data
min_age = df['Age'].min()
df['Age'].fillna(min_age, inplace=True)
print(df)
```

#	Age	Gender
0	25.0	M
1	22.0	F
2	30.0	NaN
3	22.0	F
4	22.0	M
5	28.0	NaN

2.4.5. Forward Fill (ffill)

Fill missing values with the **previous** non-missing value:

```
df = pd.DataFrame(data)  # reset to original
df['Age'].fillna(method='ffill', inplace=True)
df['Gender'].fillna(method='ffill', inplace=True)
print(df)
```

OUTPUT

#	Age	Gender
0	25.0	M
1	25.0	F
2	30.0	F
3	22.0	F
4	22.0	M
5	28.0	M

2.4.6. Backward Fill (bfill)

Fill missing values with the **next** non-missing value:

```
df = pd.DataFrame(data)  # reset to original
df['Age'].fillna(method='bfill', inplace=True)
df['Gender'].fillna(method='bfill', inplace=True)
print(df)
```

#	Age	Gender
0	25.0	M
1	30.0	F
2	30.0	M
3	22.0	F
4	28.0	M
5	28.0	NaN

2.4.7. Remove Missing Data (Drop Rows)

Remove rows with **any missing values**:

```
df = pd.DataFrame(data)  # reset to original
df.dropna(inplace=True)
print(df)
```

#	Age	Gender
0	25.0	M
3	22.0	F

(Only rows with no missing values remain)

2.5. Data Encoding

- ✓ Data Encoding in EDA is the process of converting categorical data into numerical format to facilitate analysis and modeling.
- ✓ Since many analytical methods and machine learning algorithms require numerical input, encoding transforms categories (like labels or names) into numbers, enabling statistical exploration, visualization, and predictive modeling.

2.5.1. Significance of Data Encoding

1. **Enables Numerical Analysis**
   Most statistical methods and machine learning algorithms require numeric input. Encoding converts categorical data into numbers so these techniques can be applied.
2. **Preserves Information**
   Proper encoding methods help maintain the meaning and relationships within categorical variables, such as order or frequency.
3. **Improves Model Performance**
   Well-encoded features can help models learn patterns more effectively, leading to better accuracy and generalization.
4. **Facilitates Visualization and Insights**
   Encoded data allows easier plotting and correlation analysis, helping identify trends and anomalies during exploration.
5. **Handles High Cardinality Variables**
   Special encoding techniques can manage variables with many categories without inflating data size unnecessarily.

2.5.2. One Hot Encoding

One-Hot Encoding is a method of converting categorical variables into a set of binaries (0 or 1) columns, each representing one category. For each observation, the column corresponding to the category is marked as 1, and all others are 0.

2.5.2.1.How It Works

- ✓ Suppose you have a categorical feature **Color** with three categories: `Red`, `Green`, and `Blue`.
- ✓ One-hot encoding will create three new columns: `Color_Red`, `Color_Green`, and `Color_Blue`.
- ✓ If a data point's color is `Green`, the encoded vector would be: `[0, 1, 0]`.

```python
import pandas as pd

# Original dataset
data = {
    'ID': [1, 2, 3, 4, 5],
    'Color': ['Red', 'Green', 'Blue', 'Green', 'Red']
}

df = pd.DataFrame(data)

# One-hot encoding
df_encoded = pd.get_dummies(df, columns=['Color'])

print(df_encoded)
```

OUTPUT:

```
   ID  Color_Blue  Color_Green  Color_Red
0   1           0            0          1
1   2           0            1          0
2   3           1            0          0
3   4           0            1          0
4   5           0            0          1
```

2.5.2.2. Advantages of One-Hot Encoding

1. **No Ordinal Relationship Assumed**
   - Prevents the model from interpreting categorical variables as having a natural order.
2. **Simple and Intuitive**
   - Easy to implement and understand.
3. **Widely Supported**
   - Compatible with many machine learning algorithms and libraries.
4. **Preserves Information**
   - Captures all categories explicitly without losing detail.

2.5.2.3. Disadvantages of One-Hot Encoding

1. **High Dimensionality**
   - For categorical variables with many unique categories (high cardinality), it creates many new columns, increasing memory usage and computation time.
2. **Sparsity**
   - Results in sparse matrices (mostly zeros), which can be inefficient for some algorithms.
3. **Dummy Variable Trap**
   - The encoded columns are linearly dependent (sum to 1), which can cause issues like multicollinearity in linear models unless one column is dropped.
4. **Not Suitable for Ordinal Data**
   - Does not capture order in categories (e.g., “Low”, “Medium”, “High”).

2.5.3. Ordinal Encoding

Ordinal Encoding is a technique that converts categorical variables with a meaningful order (ordinal variables) into numeric values, preserving their order.

2.5.3.1. How It Works

- ✓ Assign integers to categories based on their rank or order.
- ✓ For example, a feature **Size** with categories: `Small < Medium < Large` can be encoded as:
    - Small → 0
    - Medium → 1
    - Large → 2

2.5.3.2. When to Use Ordinal Encoding

- ✓ When the categorical variable has a **natural order** or ranking.
- ✓ Examples: Education level (High School < Bachelor < Master), Customer satisfaction (Low < Medium < High).

```python
import pandas as pd
from sklearn.preprocessing import OrdinalEncoder

# Sample dataset
data = {
    'ID': [1, 2, 3, 4, 5],
    'Education': ['High School', 'Bachelor', 'Master', 'Bachelor', 'High School']
}

df = pd.DataFrame(data)

# Define the order of categories
education_order = ['High School', 'Bachelor', 'Master']

# Create OrdinalEncoder with defined categories
encoder = OrdinalEncoder(categories=[education_order])

# Fit and transform the Education column
df['Education_Encoded'] = encoder.fit_transform(df[['Education']])

print(df)
```

```
    ID   Education  Education_Encoded
0   1  High School                0.0
1   2     Bachelor               1.0
2   3       Master               2.0
3   4     Bachelor               1.0
4   5  High School                0.0
```

2.5.3.3. Advantages

1. Preserves order information.
2. Simple and compact (only one column).
3. Efficient for models that can exploit order relationships.

2.5.3.4. Disadvantages

1. Implies a linear relationship between categories that may not be strictly true.
2. Can mislead some algorithms if the intervals between categories aren’t uniform.
3. Not suitable for nominal categorical variables without order.

2.5.4. Label Encoding:

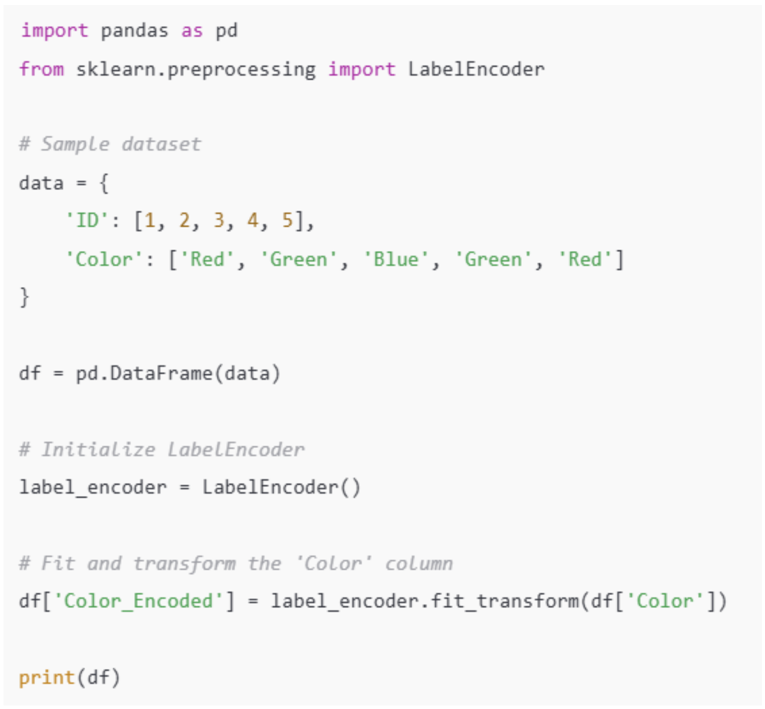
Label Encoding is a technique that converts categorical variables into numeric form by assigning each unique category an integer value. Unlike ordinal encoding, it does **not** assume any specific order between categories—it just gives them distinct numeric labels.

2.5.4.1. How It Works

- ✓ Each unique category is mapped to an integer.
- ✓ For example, a feature **Color** with categories: `Red`, `Green`, `Blue` might be encoded as:
    - Red → 0
    - Green → 1
    - Blue → 2
    -

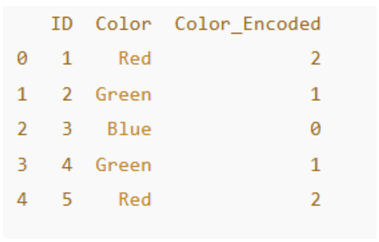
2.5.4.2. When to Use Label Encoding

- ✓ When the categorical variable is **nominal** (no order), but you are using algorithms that can handle or interpret label encoding properly (e.g., tree-based models).
- ✓ Quick and simple encoding when categories don’t have meaningful order.



```python
import pandas as pd
from sklearn.preprocessing import LabelEncoder

# Sample dataset
data = {
    'ID': [1, 2, 3, 4, 5],
    'Color': ['Red', 'Green', 'Blue', 'Green', 'Red']
}

df = pd.DataFrame(data)

# Initialize LabelEncoder
label_encoder = LabelEncoder()

# Fit and transform the 'Color' column
df['Color_Encoded'] = label_encoder.fit_transform(df['Color'])

print(df)
```

OUTPUT



```
   ID  Color  Color_Encoded
0   1    Red              2
1   2  Green              1
2   3   Blue              0
3   4  Green              1
4   5    Red              2
```

2.5.4.3. Advantages of Label Encoding

1. **Simple and Fast**
    - Easy to implement and computationally efficient.
2. **Memory Efficient**

   - Uses a single column with integer values instead of multiple columns like one-hot encoding.
3. **Works Well with Tree-Based Models**
   - Algorithms like decision trees and random forests can handle label-encoded data without issues.
4. **Useful for Ordinal Data**
   - Can be used for ordinal variables if the encoding matches the natural order.

2.5.4.4. Disadvantages of Label Encoding

1. **Imposes Ordinal Relationship on Nominal Data**
   - For nominal categorical variables, the numeric labels imply a rank/order which may mislead algorithms that assume numerical magnitude.
2. **Not Suitable for Many Algorithms**
   - Models like linear regression, logistic regression, and SVM may interpret encoded values as continuous numbers, which can lead to poor performance.
3. **Inconsistent Encoding Across Datasets**
   - If not careful, encoding might differ between training and test sets, causing errors or inconsistencies.

2.5.5. Mean Encoding

Mean Encoding replaces each category of a categorical variable with the mean value of the target variable for that category. It’s a way to encode categories based on their relationship with the target.

2.5.5.1. How It Works

- ✓ For each category in a feature, calculate the average of the target variable for all rows with that category.
- ✓ Replace the category with this average value.
- ✓ For example, if you have a feature **City** and the target is house prices, each city is replaced by the average house price in that city.

2.5.5.2. When to Use Mean Encoding

- ✓ When you want to capture the impact of categorical variables on a numeric target.
- ✓ Especially useful in supervised learning tasks.

Example:

City	Target (Price)	Mean Encoded Value
New York	500,000	450,000 (avg)
Boston	400,000	350,000 (avg)
New York	400,000	450,000

```python
import pandas as pd

# Sample dataset
data = {
    'City': ['NY', 'LA', 'NY', 'LA', 'SF', 'SF'],
    'Price': [500000, 450000, 520000, 470000, 600000, 620000]
}

df = pd.DataFrame(data)

# Calculate mean Price per City
mean_encoding = df.groupby('City')['Price'].mean()

# Map the mean to the City column
df['City_Mean_Encoded'] = df['City'].map(mean_encoding)

print(df)
```

```
  City   Price  City_Mean_Encoded
0   NY  500000           510000.0
1   LA  450000           460000.0
2   NY  520000           510000.0
3   LA  470000           460000.0
4   SF  600000           610000.0
5   SF  620000           610000.0
```

Review Questions:

1. Define missing data. Why does it matter in data analysis?
2. Explain with examples how missing data can lead to bias, loss of information, and misleading visualizations.
3. Explain any four reasons, why considering missing data matters in EDA.
4. Discuss the impact of missing data on machine learning algorithms and model evaluation.
5. Explain the types of missing data: MCAR, MAR, and MNAR with suitable examples.
6. Differentiate between various types of data missingness in terms of definition, example and interpretation.
7. How can missingness be interpreted using visualization, grouping, and quantitative analysis?
8. Illustrate mean and median imputation to handle missing data. Also highlight python implantation for both.
9. Illustrate mode and min imputation to handle missing data. Also highlight python implementation for both.
10. Apply Mean, Median, and Mode imputation on a dataset with missing Age and Gender values.
11. Compare Min Imputation, Forward Fill, and Backward Fill with examples.
12. Explain forward fill and backward fill with example. Also highlight python implementation.
13. What is data encoding? Explain any four significance of data encoding.

14. When is it appropriate to remove missing data instead of imputing? Give reasons.
15. What is data encoding? Explain its significance in Exploratory Data Analysis.
16. Illustrate working of one hot encoding.
17. Describe ordinal encoding with its advantageous and disadvantageous.
18. Differentiate between One-Hot Encoding, Ordinal Encoding, and Label Encoding.
19. Explain the advantages and disadvantages of One-Hot Encoding with examples.
20. Describe Ordinal Encoding with examples. When is it appropriate to use it?
21. Explain Label Encoding. Discuss its strengths and limitations with examples.
22. Explain mean encoding with an example and also highlight python implantation.

****