EDA & DATA TRANSFORMATION

CELERATES CAMP 2026

EXPLORATORY DATA ANALYSIS (EDA)

01

EXPLORATORY DATA ANALYSIS (EDA)

What is Exploratory Data Analysis (EDA)?

Exploratory Data Analysis (EDA) is a crucial step in the data science workflow that involves understanding, summarizing, and visualizing data before applying machine learning models or making business decisions. EDA helps uncover patterns, detect anomalies, test assumptions, and identify relationships within the data.

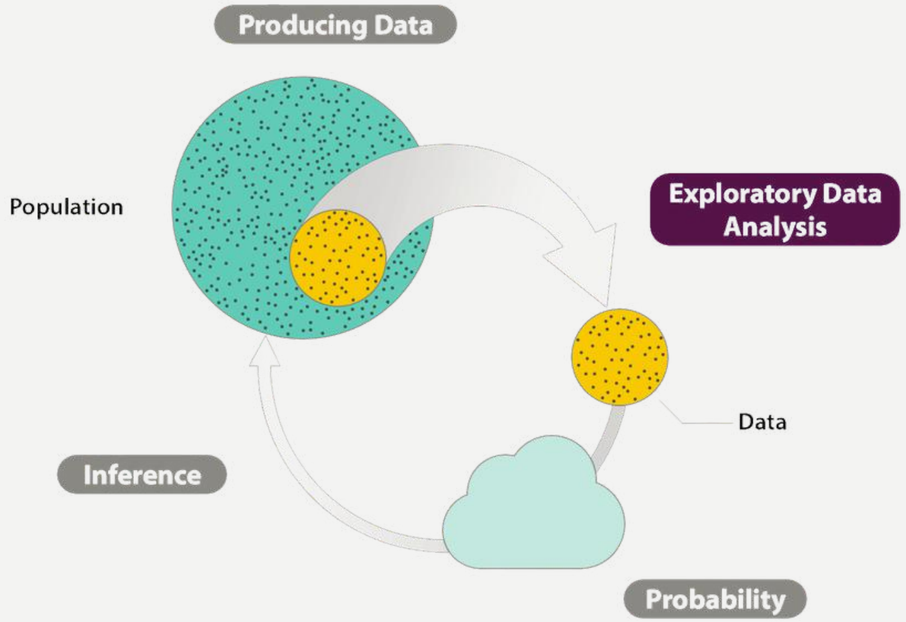
Why is EDA Important?

EDA is essential for the following reasons:

- Data Quality Assessment: Ensures data is clean, complete, and relevant.
- Insight Generation: Helps in understanding trends, patterns, and relationships.
- Model Preparation: Guides feature selection and engineering before applying machine learning models.
- Anomaly Detection: Identifies missing values, outliers, and inconsistencies.
- Hypothesis Testing: Validates assumptions before formal modeling.

EDA IN THE DATA SCIENCE PROCESS

EDA is typically performed after data collection and cleaning but before feature engineering and model building. It acts as a bridge between raw data and actionable insights, ensuring that models are built on a strong foundation of well-understood data.



Key Goals of EDA

- Understand the Structure of Data: Identify the type of data, dimensions, and basic statistics.
- Identify Missing and Erroneous Data: Detect and handle missing values and inconsistencies.
- Summarize Data Distributions: Use descriptive statistics to get an overview of the dataset.
- Visualize Data for Insights: Use charts and graphs to explore patterns and trends.
- Examine Relationships Between Variables: Perform correlation analysis and bivariate exploration.

Common Techniques in EDA

- Descriptive Statistics: Mean, median, mode, standard deviation, skewness, and kurtosis.
- Data Visualization: Histograms, box plots, scatter plots, heatmaps, and bar charts.
- Feature Engineering: Creating new variables based on existing data.
- Outlier Detection: Identifying anomalies using statistical methods.

UNDERSTANDING THE DATASET

Data Collection & Sources

Before starting EDA, it is essential to gather data from reliable sources. Common data sources include:

- CSV & Excel Files: Common formats for structured datasets.
- Databases (SQL & NoSQL): Data stored in relational and non-relational databases.
- APIs & Web Scraping: Extracting data from web sources.
- Public Datasets: Open-source data from platforms like Kaggle, UCI, and government portals.

Loading the Dataset

Once the data is collected, it must be loaded into a suitable environment for analysis. Common tools include:

- Pandas (Python): pd.read_csv(), pd.read_json(), pd.read_sql()
- R: read.csv(), read.table()
- SQL Queries: Extracting data directly from a database.

Overview of the Dataset

To understand the dataset, we perform an initial inspection:

- Check Data Structure: df.shape, df.info() to see rows, columns, and data types.
- Preview Data: df.head(), df.tail() to inspect sample records.
- Summary Statistics: df.describe() provides an overview of numerical features.
- Check for Unique Values: df.nunique() helps identify categorical distributions.

DATA CLEANING

Handling Missing Values

Missing data can affect analysis and model performance. Common techniques to handle missing values include:

- Removing Missing Data: Use df.dropna() to remove rows or columns with missing values.
- Imputing Missing Values: Fill missing values with mean, median, mode, or interpolation (df.fillna()).
- Using Predictive Imputation: Machine learning models can predict missing values based on other features.

Identifying & Removing Duplicates

Duplicate records can distort data analysis. To handle duplicates:

- Detect Duplicates: Use df.duplicated().sum() to count duplicate rows.
- Remove Duplicates: Use df.drop_duplicates() to clean redundant data.

Dealing with Outliers

Outliers can significantly impact statistical analysis. Methods to detect and handle outliers include:

- Boxplots & Z-Scores: Use sns.boxplot() or Z-score methods to identify extreme values.
- Capping or Trimming: Replace outliers with upper/lower quantiles or remove extreme values.
- Transformation Techniques: Use log transformations or scaling to reduce outlier impact.

Fixing Incorrect or Inconsistent Data

Data inconsistencies can arise due to manual entry errors, differing formats, or incorrect values. Techniques to fix these include:

- Standardizing Data Formats: Ensure date formats, categorical labels, and numerical precision are uniform.
- Correcting Data Entries: Replace incorrect values using mapping or logical rules.
- Validating Data Integrity: Cross-check data consistency using domain knowledge and external references.

DATA EXPLORATION & DESCRIPTIVE STATISTICS

Summary Statistics

Summary statistics help in understanding the distribution and characteristics of numerical data. Key metrics include:

- Mean, Median, Mode: Measures of central tendency.
- Standard Deviation & Variance: Measures of data spread.
- Minimum & Maximum Values: Helps in detecting potential outliers.
- Interquartile Range (IQR): Measures dispersion between quartiles.

Visualizing Data Distributions

Graphical methods help in identifying trends and anomalies in data. Common visualization techniques include:

- Histograms: plt.hist(df['column']) – Shows frequency distribution.
- Boxplots: sns.boxplot(x=df['column']) – Highlights outliers and spread.
- Violin Plots: Combines boxplot and kernel density estimation.

Analyzing Categorical Variables

For categorical data, summarization techniques include:

- Value Counts: df['category_column'].value_counts() to see frequency distribution.
- Bar Plots: sns.countplot(x='category_column', data=df) for visualization.
- Cross-tabulation: pd.crosstab(df['col1'], df['col2']) to analyze relationships.
- Correlation Matrix: df.corr() to measure relationships between numerical variables.
- Scatter Plots: sns.scatterplot(x='feature1', y='feature2', data=df) to visualize relationships.
- Pair Plots: sns.pairplot(df) for multi-variable visualization.

FEATURE ENGINEERING

What is Feature Engineering?

Feature engineering is the process of creating new features or modifying existing ones to improve model performance. It helps in extracting meaningful information from raw data.

Importance of Feature Engineering in EDA

Feature engineering plays a crucial role in improving model accuracy by enhancing the quality of input data. It helps in reducing noise and making patterns more detectable by algorithms.

Techniques for Feature Engineering

- Creating New Features: Deriving new columns based on existing data.
- Feature Transformation: Applying logarithmic, square root, or polynomial transformations.
- Encoding Categorical Variables: Using one-hot encoding, label encoding, or target encoding.
- Feature Scaling: Normalization (MinMaxScaler) and standardization (StandardScaler).
- Feature Selection: Removing irrelevant or redundant features using statistical tests.
- Handling Date & Time Features: Extracting components like year, month, day, or creating time-based aggregations.

02

DATA TRANSFORMATION

ABOUT

DATA TRANSFORMATION

Data transformation in Data Science refers to the process of converting or mapping data from one format or structure to another, in order to make it suitable for analysis or to facilitate the application of machine learning algorithms. This can include tasks such as normalization, aggregration, and feature engineering. It is an important step in the data preprocessing phase, which is often required before data can be effectively analyzed or modeled.

Example :

- **Predictive Modeling**: Preparing features (e.g., normalizing data) so that your model can make accurate predictions.
- **ETL (Extract, Transform, Load) Processes**: The “T” in ETL stands for Transform, and that’s where you restructure your raw data into formats suitable for downstream analytics.
- **Feature Engineering**: Creating new features by combining, encoding, or scaling existing ones to unlock deeper insights.

SCALING

This involves scaling the data so that it falls within a specific range, such as 0 to 1, or -1 to 1. This can help to ensure that the data is on a similar scale and prevent certain features from dominating the analysis.

```
>>> from sklearn.preprocessing import StandardScaler
>>> data = [[0, 0], [0, 0], [1, 1], [1, 1]]
>>> scaler = StandardScaler()
>>> print(scaler.fit(data))
StandardScaler()
>>> print(scaler.mean_)
[0.5 0.5]
>>> print(scaler.transform(data))
[[-1. -1.]
 [-1. -1.]
 [ 1.  1.]
 [ 1.  1.]]
>>> print(scaler.transform([[2, 2]]))
[[3. 3.]]
```

```
>>> from sklearn.preprocessing import MinMaxScaler
>>> data = [[-1, 2], [-0.5, 6], [0, 10], [1, 18]]
>>> scaler = MinMaxScaler()
>>> print(scaler.fit(data))
MinMaxScaler()
>>> print(scaler.data_max_)
[ 1. 18.]
>>> print(scaler.transform(data))
[[0.   0.  ]
 [0.25 0.25]
 [0.5  0.5 ]
 [1.   1.  ]]
>>> print(scaler.transform([[2, 2]]))
[[1.5 0. ]]
```

SCALING

What is Feature Scaling?

Feature scaling is a data preprocessing technique used in machine learning to transform the features of a dataset onto a common scale. It’s nothing but, we scale the widely spread data into a range(0 ->1 range or -1 -> 1).

Why feature scaling is important?

- **Handling Extreme Values**: Some data has very big or small numbers, which can confuse models. Scaling puts everything on a similar range, making patterns easier to find.
- **Faster Training**: Models learn faster when features are scaled because the optimization process runs more smoothly.
- **Better for Distance-Based Models**: Algorithms like KNN and SVM rely on distances. Scaling ensures all features are treated fairly.
- **Helps Linear Models**: Scaling prevents some features from dominating others, so the model learns properly.
- **Fair Model Comparison**: When testing different models, scaling makes sure they all use the same type of data for a fair comparison.
- **Works Better in High Dimensions**: In big datasets, scaling keeps distances meaningful, helping models perform better.

SCALING

Types of Feature Scaling

The main aim of Feature scaling is to bring data into a common scale (-1,1) or (0,1) based on the scale factor, this scaling is divided into 2 types.

- Standardization
- Normalization

Impact on the machine learning models

- Fixing Non-Normal Data: Some models work best with normally distributed data. Transformations like log or Box-Cox help adjust the data for better performance.
- Handling Mixed Data: When a dataset has numbers and categories, transformations like encoding or vectorizing help models understand the data correctly.
- Reducing Outlier Impact: Techniques like robust scaling prevent extreme values from affecting the model too much, making predictions more reliable.
- Improving Date-Time Features: Extracting details like the day of the week or time of day helps models detect useful patterns in time-based data.
- Simplifying Complex Patterns: Transforming features, like grouping continuous values into categories, helps models understand non-linear relationships more easily.

SCALING

- **MinMaxScaler**: This method scales the data so that it falls within a specific range, such as 0 to 1, or -1 to 1. It is useful for data that has a non-normal distribution.
- **StandardScaler**: This method scales the data so that it has a mean of 0 and a standard deviation of 1. It is useful for data that has a normal distribution.
- **RobustScaler**: This method scales the data based on the median and quartiles, rather than the mean and standard deviation. It is useful for data that has outliers.
- **MaxAbsScaler**: This method scales the data so that the maximum absolute value of each feature is 1.0.
- **Normalizer**: This method normalizes the data, by scaling each sample to have unit norm. It is useful for data that needs to be normalized.
- **QuantileTransformer**(with normal or uniform output distribution): This method transforms the features to follow a uniform or a normal distribution. It is useful for data with non-normal distribution.

ENCODING

What is Data Encoding?

Data encoding converts categorical or text data into numbers so that algorithms can process them efficiently.

But which categorical data needs encoding? There are two types:

- Nominal Data – Categories with no specific order (e.g., colors, gender, animal types).
- Ordinal Data – Categories with a meaningful order, but the gaps between them are not necessarily equal (e.g., education levels, customer ratings).

Why Does This Matter?

Consider a food delivery app recommending restaurants. If users rate restaurants from “Poor” to “Excellent,” this is ordinal data because the ratings have a clear order. The system should prioritize highly rated restaurants, even if the exact difference between ratings isn’t measurable.

Understanding these types helps in choosing the right encoding method for machine learning models.

ENCODING

Label Encoding

Label Encoding is the simplest form of encoding categorical variables. It assigns a unique integer value to each category. This technique works well when the categories have an inherent ordinal relationship (e.g., Low, Medium, High), but it can introduce unintended order in purely nominal categories (e.g., colors).

```
from sklearn.preprocessing import LabelEncoder

# Label encoding with Scikit-learn
le = LabelEncoder()
df['Fruit_LabelEncoded'] = le.fit_transform(df['Fruit'])
print(df)
```

```
    Fruit  Fruit_LabelEncoded
0   Apple                   0
1  Banana                   1
2  Orange                   2
3   Apple                   0
4  Banana                   1
```

ENCODING

One-Hot Encoding

One-Hot Encoding converts each unique category into a new column (binary vector), with each row marked as 1 in the column corresponding to the category it belongs to and 0 in all other columns. This is suitable for nominal categorical variables.

```
from sklearn.preprocessing import OneHotEncoder

# One-Hot Encoding with Scikit-learn
ohe = OneHotEncoder(sparse=False)
encoded = ohe.fit_transform(df[['Fruit']])

# Convert back to DataFrame for better visualization
df_one_hot = pd.DataFrame(encoded, columns=ohe.get_feature_names_out(['Fruit']))
print(df_one_hot)
```

	Fruit_Apple	Fruit_Banana	Fruit_Orange
0	1.0	0.0	0.0
1	0.0	1.0	0.0
2	0.0	0.0	1.0
3	1.0	0.0	0.0
4	0.0	1.0	0.0

ENCODING

Ordinal Encoding

Ordinal Encoding is used when the categories have an inherent order or ranking (e.g., Low < Medium < High). It assigns integer values based on the rank of each category.

```
from sklearn.preprocessing import OrdinalEncoder

# Sample data
df = pd.DataFrame({'Size': ['Small', 'Medium', 'Large', 'Small', 'Large']})

# Ordinal Encoding
oe = OrdinalEncoder(categories=[['Small', 'Medium', 'Large']])
df['Size_OrdinalEncoded'] = oe.fit_transform(df[['Size']])
print(df)
```

	Size	Size_OrdinalEncoded
0	Small	0.0
1	Medium	1.0
2	Large	2.0
3	Small	0.0
4	Large	2.0

SCIKIT-LEARN'S COLUMN TRANSFORMER

What is SciKit-Learn's Column Transformer?

Scikit-Learn's Column Transformer is a powerful tool for applying different transformations to different columns of a dataset. It allows you to define a set of transformers and specify which columns each transformer should be applied to. This is particularly useful when dealing with datasets that have heterogeneous data types or when you want to apply different preprocessing steps to different subsets of features.

Why do we use a column transformer?

- Simplifies your code by applying different transformations to different subsets of columns in a single step.
- It helps organize your preprocessing steps efficiently by allowing you to define transformations for different feature types.
- Prevents unintentional transformations on unrelated features, avoiding data leakage.
- Can be used to apply different imputation strategies to different subsets of features, addressing missing data in a customized way.

SCIKIT-LEARN'S COLUMN TRANSFORMER

How do we use this column transformer?

1. Import necessary libraries:

```
from sklearn.compose import ColumnTransformer
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import StandardScaler, OneHotEncoder
```

2. Specify the features(categorical and numerical):

```
# Specify Features and Transformers
scaling_features = ['scaling_column1', 'scaling_column2']
imputer_features = ['imputer_column1', 'imputer_column2']
categorical_features = ['categorical_column1', 'categorical_column2']
```

3. Specify Transformer:

```
# Combine Transformers using ColumnTransformer
preprocessor = ColumnTransformer(
    transformers=[
        ('SI', SimpleImputer(), imputer_features),
        ('SS', StandardScaler(), scaling_features),
        ('OHE', OneHotEncoder(sparse=False), categorical_features],
        remainder='passthrough'
)

# In the transformer,
# SI is a given name for that transformation
# SimpleImputer is a sklearn library for transformation
# imputer_features are the columns/features
```

THANK YOU

CELERATES CAMP 2026