Data Preprocessing

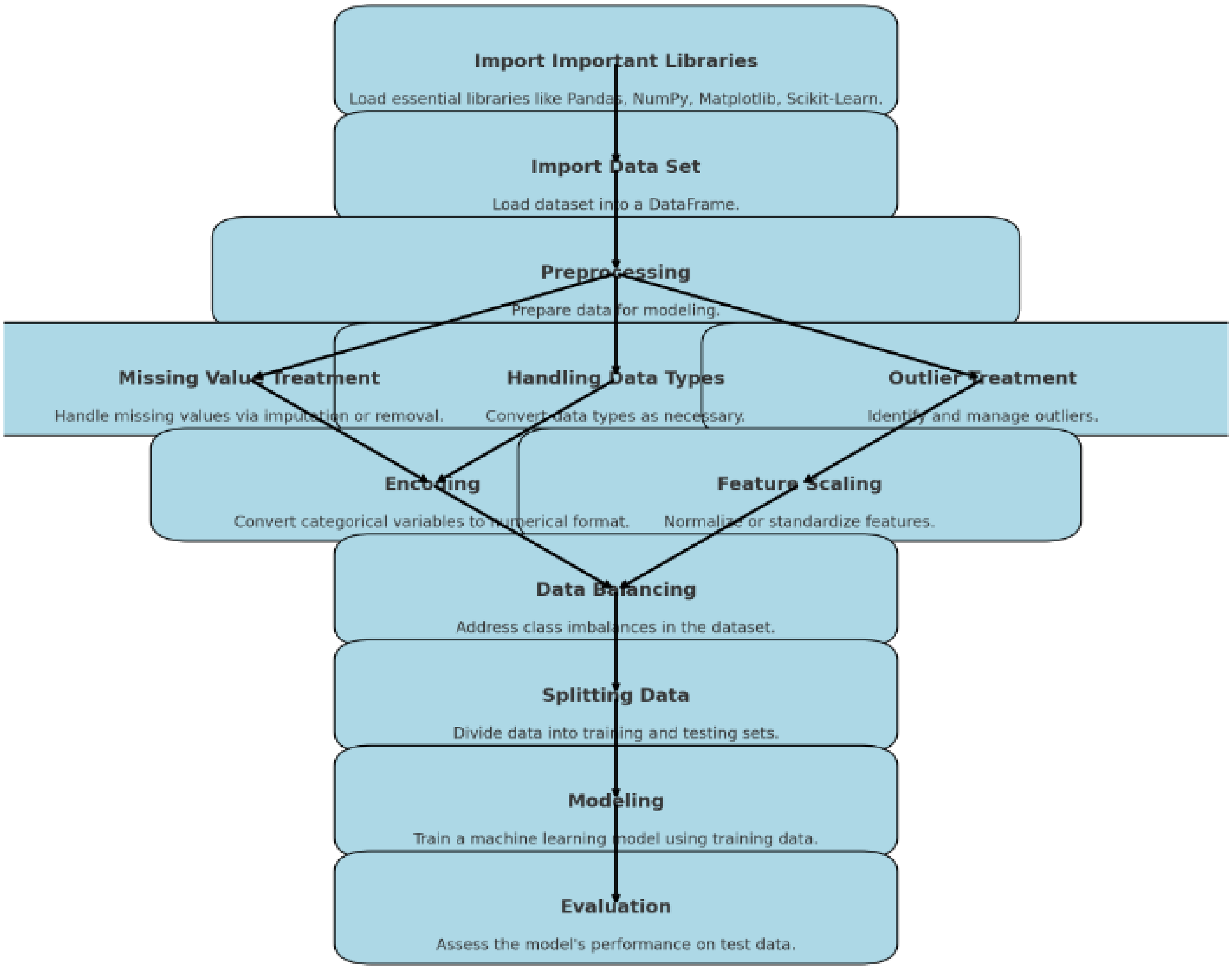
Essential Steps for Preparing Data Before Modeling

Introduction

- **Data preprocessing** is a crucial step in the machine learning and statistical analysis pipeline.
- It involves transforming raw data into a clean and usable format, ensuring that the data is consistent, accurate, and relevant for the analysis.
- Here are the key reasons why data preprocessing is essential:
  - Improving Data Quality
  - **Enhancing Model Performance**
  - Improving Interpretability
  - Ensuring Consistency

Basic Steps in Data Preprocessing

- Step 1 :Import important libraries
- Step 2: Import dataset
- Step 3: Preprocessing:
  - Find duplicates
  - Missing value treatment
  - Encoding
  - Handling data types
  - Outlier treatment
  - Feature scaling
  - Data balancing


Import Important Libraries
Load essential libraries like Pandas, NumPy, Matplotlib, Scikit-Learn.
Import Data Set
Load dataset into a DataFrame.
Preprocessing
Prepare data for modeling.
Missing Value Treatment
Handle missing values via imputation or removal.
Handling Data Types
Convert data types as necessary.
Outlier Treatment
Identify and manage outliers.
Encoding
Convert categorical variables to numerical format.
Feature Scaling
Normalize or standardize features.
Data Balancing
Address class imbalances in the dataset.
Splitting Data
Divide data into training and testing sets.
Modeling
Train a machine learning model using training data.
Evaluation
Assess the model's performance on test data.

Import Important Libraries

```
import os, sys
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
%matplotlib inline
import seaborn as sns
sns.set()

import warnings
warnings.filterwarnings('ignore')

pd.set_option('display.max_rows',500)
pd.set_option('display.max_columns',50)
pd.set_option('display.width',1000)
```

Purpose of Libraries

- **os:** Functions to interact with the operating system.
  - **Example Usage**: **os.listdir()** lists files and directories in the specified path.
- **numpy:** Support for arrays, matrices, and mathematical functions.
- **pandas:** Data manipulation and analysis.
- **matplotlib & seaborn:** Data visualization.
- **warnings:** Manage warning messages in code.
- **sns.set():** Automatically sets the seaborn plot aesthetics to a default theme.
- **%matplotlib inline:** A magic command used in Jupyter notebooks to display matplotlib plots inline within the notebook.

Importing Data

- Data = pd.read_csv(r"C:\Desktop\DataScience\data.csv“)
- Data = pd.read_csv(r"C://Desktop//DataScience//data.csv“)
- Data.head()
- Data.tail()

Finding and Handling Duplicate

- If there is any kind of repetition of data in dataset than it is required to remove them for healthy analysis and prediction.

```
[86]: titanic = pd.read_csv('titanic_dataset.csv')
      titanic.duplicated().sum()
```

```
[39]: titanic.shape
```

```
[39]: (915, 11)
```

```
[88]: titanic.drop_duplicates(keep='first',inplace=True)
```

```
[90]: titanic.shape
```

```
[90]: (891, 11)
```

Handling Duplicates

```
[125]: df.duplicated().sum()

[125]: 483

[128]: df[df.duplicated()]
```

```
[131]: df.drop_duplicates(ignore_index=True,inplace=True)

[132]: df.shape

[132]: (10357, 13)

[134]: df.duplicated().sum()

[134]: 0
```

Handling Missing Values

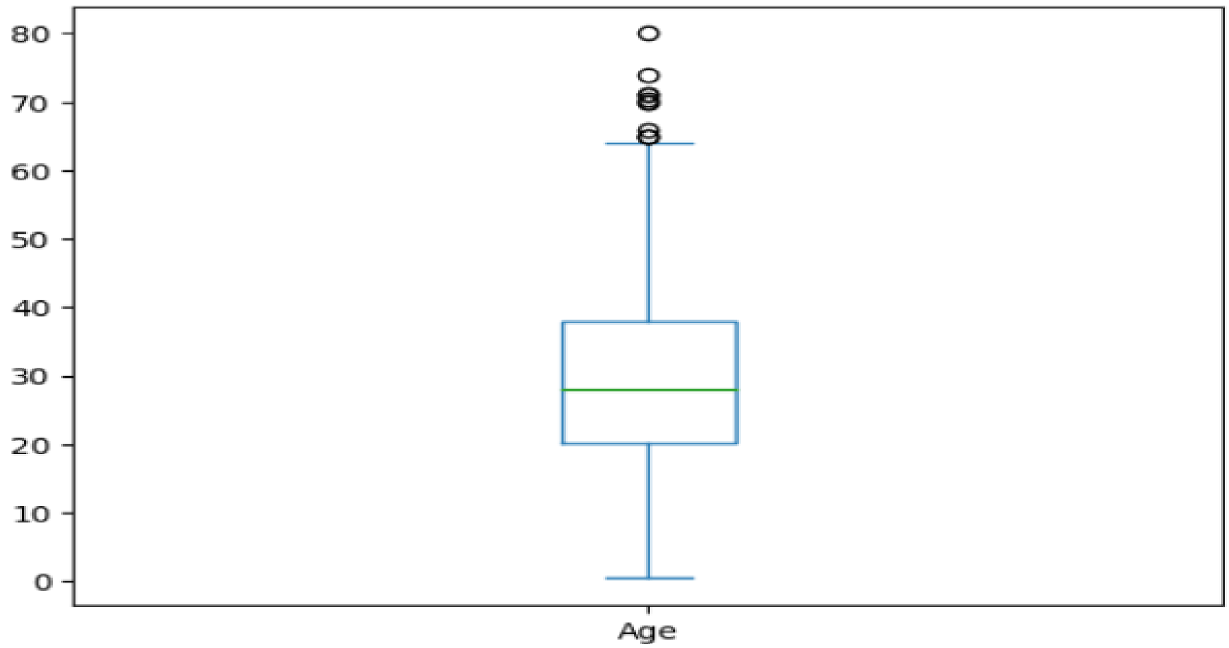
- Identifying missing values using `` `df.isnull().sum()` ``.
- **In percentage form: df.isnull().sum()/len((df)*100**
- Techniques to handle missing values:
- If any variable have missing value > 25% , drop it.
  - **data = data.drop([‘name of column’], axis = 1)**
- - Else Imputing missing values

Imputation Method

- **Various imputation Approaches are:**
- **Simple Statistical Imputation:**
  - **Mean: If no outliers.**
  - **Median:If data have outliers.**
  - **Mode : If variables are categorical type.**
- **KNN imputation:** Replaces missing values based on the mean or median of the nearest neighbors' values.

```
[17]: dataset['Age'].plot(kind='box')
```

```
[17]: <AxesSubplot:>
```




```
[19]: dataset['Age'] = dataset['Age'].fillna(dataset['Age'].median())
```

[111]:

```python
# Embarked : object
titanic['Embarked'].value_counts()
```

[111]:

```
Embarked
S    644
C    168
Q     77
Name: count, dtype: int64
```

[113]:

```python
# Filling missing data by "S"
titanic['Embarked'] = titanic['Embarked'].fillna('S')
```

```
[148]: df.Rating.dtype
```

```
[148]: dtype('float64')
```

```
[149]: df.Rating.unique()
```

```
[149]: array([ 4.1,  3.9,  4.7,  4.5,  4.3,  4.4,  3.8,  4.2,  4.6,  3.2,  4. ,
               nan,  4.8,  4.9,  3.6,  3.7,  3.3,  3.4,  3.5,  3.1,  5. ,  2.6,
               3. ,  1.9,  2.5,  2.8,  2.7,  1. ,  2.9,  2.3,  2.2,  1.7,  2. ,
               1.8,  2.4,  1.6,  2.1,  1.4,  1.5,  1.2, 19. ])
```

```
[150]: # - It's a cataegorical column
       # - We can fill the ull values with the mode value
```

```
[154]: df.Rating.fillna(df.Rating.mode()[0],inplace=True)
```

```
[155]: df.Rating.mode()[0]
```

```
[155]: 4.4
```

Encoding Categorical Variables

- Encoding categorical variables is a critical step in data preprocessing for machine learning models, as most models require numerical input.
- **There are two approach:**
- **Label encoding and one-hot encoding.**

Label Encoding

- It converts each category in a categorical variable to a unique integer.
- **When to Use:** When the categorical variable has an ordinal relationship (e.g., low, medium, high).
- When there are a limited number of categories.

[42]:

```
dataset2['Sex'] = dataset2['Sex'].astype('category')
dataset2['Sex'] = dataset2['Sex'].cat.codes
```

[46]:

```
dataset2.head(10)
```

[46]:

	Sex	Embarked
0	1	S
1	0	C
2	0	S
3	0	S
4	1	S

[48]:

```
dataset2['Embarked'] = dataset2['Embarked'].astype('category')
dataset2['Embarked'] = dataset2['Embarked'].cat.codes
```

[52]:

```
dataset2.head(10)
```

[52]:

	Sex	Embarked
0	1	2
1	0	0
2	0	2
3	0	2
4	1	2

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

# Sample data
data = {'color': ['red', 'green', 'blue', 'green', 'blue', 'red']}
df = pd.DataFrame(data)

# Initialize and apply LabelEncoder
label_encoder = LabelEncoder()
df['color_encoded'] = label_encoder.fit_transform(df['color'])

print(df)
```

```
   color  color_encoded
0    red              2
1  green              1
2   blue              0
3  green              1
4   blue              0
5    red              2
```

One-Hot Encoding

- One-hot encoding converts categorical variables into a series of binary columns, each representing a unique category.
- **When to Use**
- When the categorical variable is nominal (no intrinsic order).
- When you want to avoid introducing ordinal relationships.

[84]:

```
dataset1 = pd.get_dummies(dataset1, columns=['Sex', 'Embarked'])
```

```
# Dummi variable concept - (n-1)
dataset1 = dataset1.drop(['Sex_male','Embarked_C'], axis=1)
```

- **pd.get_dummies()** is a function in the pandas library in Python used for one-hot encoding categorical variables.
- In one hot encoding usually drop one column.

[86]:

	Sex_female	Sex_male	Embarked_C	Embarked_Q	Embarked_S
0	False	True	False	False	True
1	True	False	True	False	False
2	True	False	False	False	True
3	True	False	False	False	True
4	False	True	False	False	True

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

# Sample data
data = {'color': ['red', 'green', 'blue', 'green', 'blue', 'red']}
df = pd.DataFrame(data)

# Initialize and apply OneHotEncoder
one_hot_encoder = OneHotEncoder(sparse=False)
one_hot_encoded = one_hot_encoder.fit_transform(df[['color']])

# Convert to DataFrame for better readability
df_one_hot = pd.DataFrame(one_hot_encoded, columns=one_hot_encoder.categories_[0])

print(df_one_hot)
```

```
   blue  green  red
0   0.0    0.0  1.0
1   0.0    1.0  0.0
2   1.0    0.0  0.0
3   0.0    1.0  0.0
4   1.0    0.0  0.0
5   0.0    0.0  1.0
```

Handling Outlier

- **Identifying outliers:**
- Visualization-Based Detection:
  - Box plots
  - Histograms with normal distribution curve

- **Statistical Methods:**
  - Z-Score (standard deviation approach)
  - IQR (Interquartile Range):
  - Values below $Q1-1.5\times IQR$
  - Values above $Q3+1.5\times IQR$.

Approaches to Handle Outliers

- Capping method:
  - Using IQR or Z score
- Transformation Approach:
  - Log Transformation
  - Square root Transformation
  - Box- Cox Transformation
  - Winsorization Method

Finding Outliers Using IQR method

[23]:
```
# Finding the IQR

percentile25 = df['placement_exam_marks'].quantile(0.25)
percentile75 = df['placement_exam_marks'].quantile(0.75)
print(percentile25)
print(percentile75)
```

```
17.0
44.0
```

[24]:
```
iqr = percentile75 - percentile25
iqr
```

[24]:
```
27.0
```

[25]:
```
upper_limit = percentile75 + 1.5*iqr
lower_limit = percentile25 - 1.5*iqr

print(upper_limit)
print(lower_limit)
```

```
84.5
-23.5
```

[26]:
```
# finding outliers
df[df['placement_exam_marks'] > upper_limit]
```

[26]:

	cgpa	placement_exam_marks	placed	cgpa_zscore
9	7.75	94.0	1	1.280667
40	6.60	86.0	1	-0.586526
61	7.51	86.0	0	0.890992
134	6.33	93.0	0	-1.024910
162	7.80	90.0	0	1.361849
283	7.09	87.0	0	0.209061
290	8.38	87.0	0	2.303564

[27]:
```
df[df['placement_exam_marks'] < lower_limit]
```

[27]:

cgpa	placement_exam_marks	placed	cgpa_zscore

```
[13]: Q1 = np.percentile(df['cgpa'], 25)
      Q3 = np.percentile(df['cgpa'], 75)
      IQR = Q3 - Q1
```

```
[220]: pos_outlier = Q3 + 1.5 * IQR
       neg_outlier = Q1 - 1.5 * IQR
```

Handling outlier using capping by IQR

```
[30]: new_df_cap = df.copy()

[31]: new_df_cap.shape

[31]: (1000, 4)

[32]: new_df_cap['placement_exam_marks'] = np.where(new_df_cap['placement_exam_marks'] > upper_limit,
                                               upper_limit,
                                               np.where(new_df_cap['placement_exam_marks'] <lower_limit,
                                                        lower_limit,
                                                       new_df_cap['placement_exam_marks'] ))

[33]: new_df_cap.shape

[33]: (1000, 4)

[34]: new_df_cap[new_df_cap['placement_exam_marks'] > upper_limit]
```

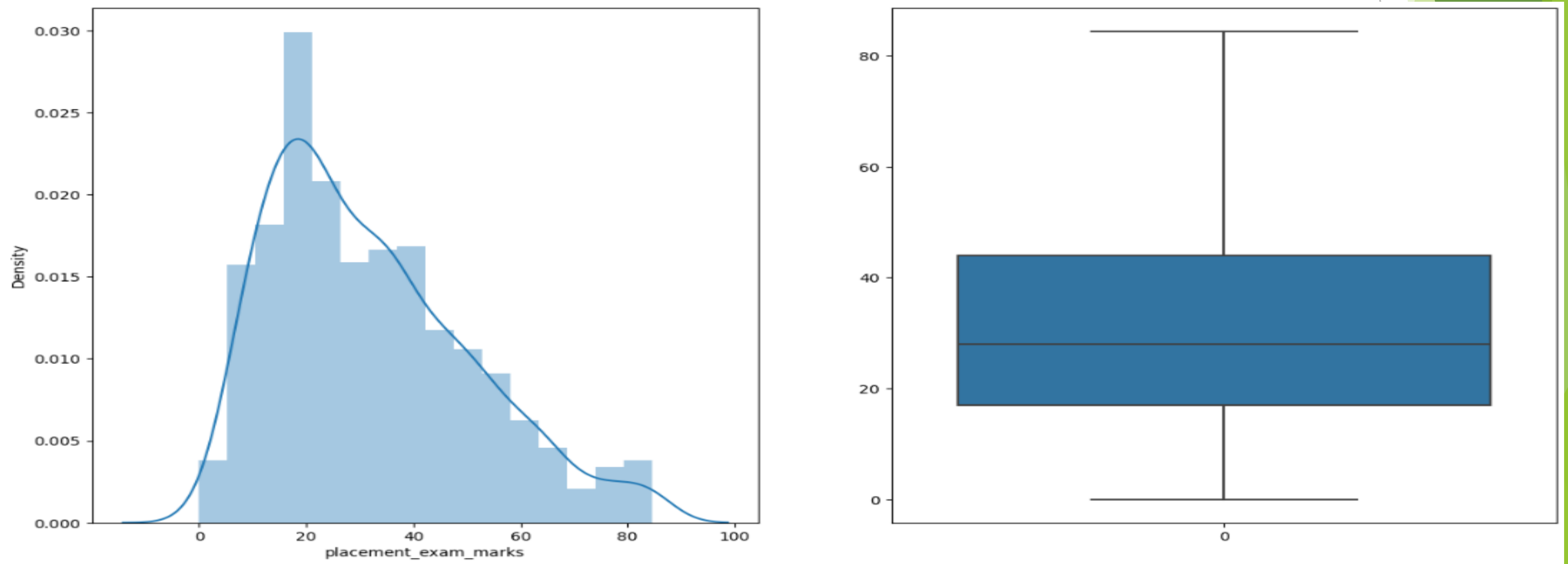
[34]:

cgpa	placement_exam_marks	placed	cgpa_zscore

Distribution and box plot

[36]:

```
plt.figure(figsize=(16,8))
plt.subplot(1,2,1)
sns.distplot(new_df_cap['placement_exam_marks'])

plt.subplot(1,2,2)
sns.boxplot(new_df_cap['placement_exam_marks'])

plt.show()
```



Handling outlier using capping by Z score:

- Step1: find mean and standard deviation

```
print("mean value of cgpa", df['cgpa'].mean())
print()
print("std value of cgpa", df['cgpa'].std())
print()
print("min value of cgpa", df['cgpa'].min())
print()
print("max value of cgpa", df['cgpa'].max())
```

```
mean value of cgpa 6.96124000000001

std value of cgpa 0.6158978751323894

min value of cgpa 4.89

max value of cgpa 9.12
```

- Step 2: Find minimum and maximum based on normal distribution parameters to identify an outlier

```
[17]: upper_limit = df['cgpa'].mean() + 3*df['cgpa'].std()
      lower_limit = df['cgpa'].mean() - 3*df['cgpa'].std()

[18]: print(upper_limit)
      print(lower_limit)

8.808933625397168
5.113546374602832
```

Step 3: Capping

```
[19]: df['cgpa'] = np.where(df['cgpa']>upper_limit, upper_limit,
                            np.where(df['cgpa']<lower_limit, lower_limit, df['cgpa']))

[20]: df['cgpa'].describe()

[20]: count    1000.000000
      mean        6.961499
      std         0.612688
      min         5.113546
      25%         6.550000
      50%         6.960000
      75%         7.370000
      max         8.808934
      Name: cgpa, dtype: float64
```

- To view the outliers based on mini or max

```
df[(df['cgpa'] > 8.80) | (df['cgpa'] < 5.11)]
```

	cgpa	placement_exam_marks	placed
485	4.92	44.0	1
995	8.87	44.0	1
996	9.12	65.0	1
997	4.89	34.0	0
999	4.90	10.0	1

To Calculate Z-score

```
df['cgpa_zscore'] = (df['cgpa'] - df['cgpa'].mean())/df['cgpa'].std()
```

```
df
```

	cgpa	placement_exam_marks	placed	cgpa_zscore
0	7.19	26.0	1	0.371425
1	7.46	38.0	1	0.809810
2	7.54	40.0	1	0.939701
3	6.42	8.0	1	-0.878782
4	7.23	17.0	0	0.436371

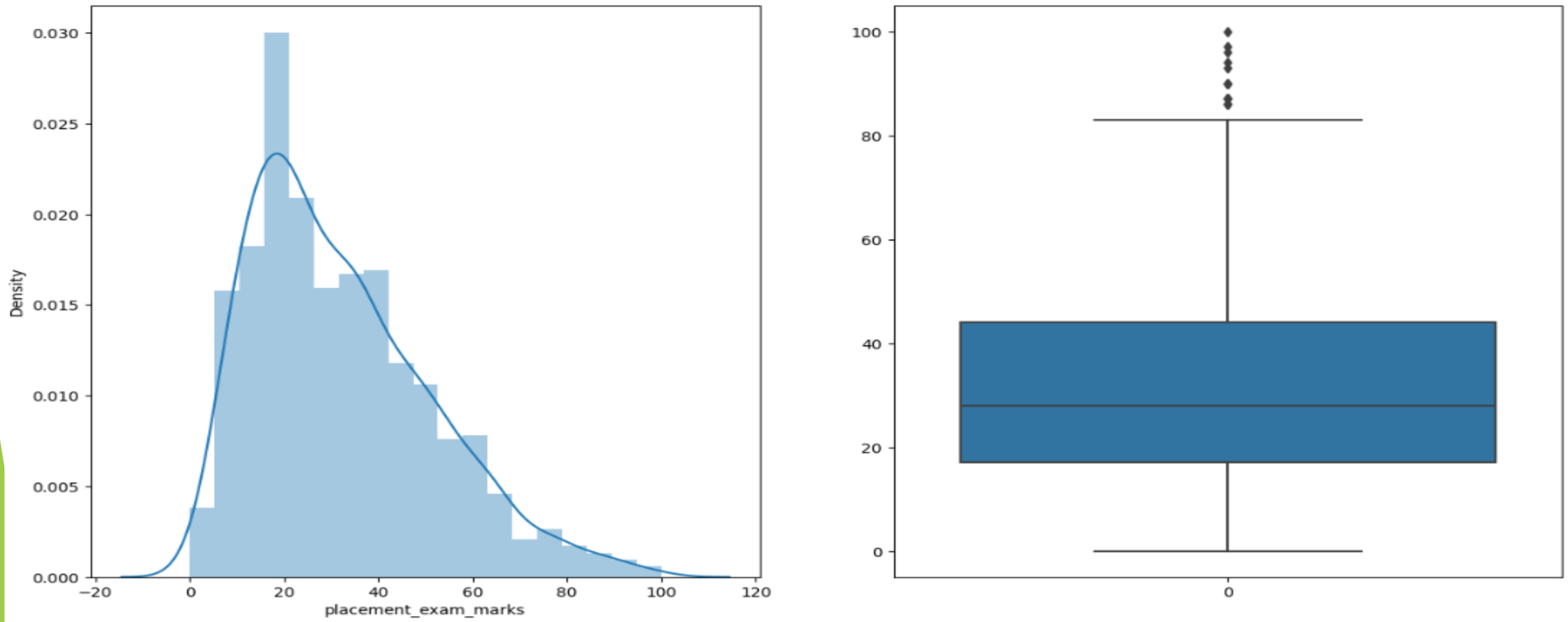
- To view the outliers based on Z-score:

```
df[(df['cgpa_zscore'] > 3) | (df['cgpa_zscore'] < -3)]
```

	cgpa	placement_exam_marks	placed	cgpa_zscore
485	4.92	44.0	1	-3.314251
995	8.87	44.0	1	3.099150
996	9.12	65.0	1	3.505062
997	4.89	34.0	0	-3.362960
999	4.90	10.0	1	-3.346724

To visualize the outliers by distribution plot and boxplot:

```
[35]: # comparing
plt.figure(figsize=(16,8))
plt.subplot(1,2,1)
sns.distplot(df['placement_exam_marks'])

plt.subplot(1,2,2)
sns.boxplot(df['placement_exam_marks'])
```



Log Transformation

- Logarithmic transformation is often used to reduce the effect of large outliers by compressing the range of data values.
- **Formula:**
- $y=\log(x)$
- Works best when all values are positive (since log of zero or negative values is undefined).
- Helps stabilize variance and normalize skewed distributions.

[10]:

```
df_log_transformation = df.applymap(lambda x: np.log(x) if x > 0 else x)
print(df_log_transformation.describe())
```

```
        Variable1   Variable2
count  103.000000  103.000000
mean     3.897492    3.450651
std      0.291443    0.262252
min      3.245329    2.988442
25%      3.727106    3.325204
50%      3.887418    3.405378
75%      4.027562    3.523882
max      5.135798    4.787492
```

Square Root Transform

- Square root transformation reduces the magnitude of large values but less aggressively than logarithmic transformation.
- It works with zero values but not negatives.
- **Formula:**
- y=sqrt{x}

[19]:

```
df_sqrt_transformation = df.applymap(lambda x: np.sqrt(x) if x > 0 else x)
print(df_sqrt_transformation.describe())
```

```
        Variable1   Variable2
count  103.000000  103.000000
mean     7.100766    5.669671
std      1.196587    0.931304
min      5.066573    4.455864
25%      6.446677    5.273014
50%      6.984609    5.488687
75%      7.491591    5.823731
max     13.038405   10.954451
```

Box-Cox Transformation

- The **Box-Cox transformation** is a family of power transformations that stabilize variance, make the data more normal-like, and reduce the impact of outliers.
- Unlike logarithmic or square root transformations, the Box-Cox method includes an adjustable parameter (λ that determines the transformation applied.
- **When to Use**
- When data is not normally distributed and transformations like log or square root are insufficient.
- To stabilize variance across different scales of data.

Mathematical Formula

The Box-Cox transformation is defined as:

$$y(\lambda) = \begin{cases} \frac{y^{\lambda}-1}{\lambda}, & \text{if } \lambda \neq 0 \\ \ln(y), & \text{if } \lambda = 0 \end{cases}$$

- λ: Determines the type of transformation. Common values include:
  - $\lambda = 1$: No transformation (identity).
  - $\lambda = 0$: Logarithmic transformation.
  - $\lambda = 0.5$: Square root transformation.
- Box-Cox requires all data values to be positive.

```
df_box_cox_transformation = df.copy()

for col in df.columns:
    df_box_cox_transformation[col], _ = boxcox(df[col]  + 1e-6)
    # adding small constant to handle zeros
print(df_box_cox_transformation.describe())
```

```
        Variable1   Variable2
count  103.000000  103.000000
mean     1.040327    0.595645
std      0.006870    0.000609
min      1.017788    0.593607
25%      1.036425    0.595341
50%      1.040986    0.595629
75%      1.044448    0.595989
max      1.060401    0.597434
```

Winsorization:

- Winsorization replaces extreme values with specified percentiles to limit the influence of outliers while retaining the dataset's size.
- **Steps:**
- Define lower and upper limits (e.g., 5th and 95th percentiles).
- Replace values below the lower limit with the 5th percentile and above the upper limit with the 95th percentile.

[59]:

```
df_winsorized = df.apply(lambda x: winsorize(x, limits=[0.05, 0.05]))
print(df_winsorized)
```

```
     Variable1  Variable2
0    64.458713  28.999760
1    56.530266  24.094480
2    33.324787  31.311350
3    33.324787  29.009641
4    60.190868  37.719077
..         ...        ...
98   57.291343  27.761510
99   58.190403  32.080371
100  69.629120  39.635437
101  69.629120  39.635437
102  69.629120  39.635437
```

Feature Scaling

- Importance of scaling features.
- **Methods:** Standard Scaler, Min-Max Scaler, Normalizer.
- We do not do feature scaling with dependent variables.
- So Ist separate the data into independent and dependent variable.

```
x = dataset.iloc[:,1:]
y = dataset[['Survived']]
```

Standardization:

- Scaling features to have zero mean and unit variance. This is particularly important for algorithms that rely on distance measures, such as SVM and k-NN.

```
from sklearn.preprocessing import StandardScaler
sc = StandardScaler()
x = sc.fit_transform(x)
pd.DataFrame(x)
```

When to Use Standard Scaler

- The **Standard Scaler** standardizes features by removing the mean and scaling to unit variance.
- Where "Remove the mean" means that for each feature in your dataset, you subtract the mean (average) value of that feature from all the values of that feature.
- This process centers the feature around zero, ensuring that the transformed feature has a mean of zero.
- This means each feature will have a mean of ‘0’ and a standard deviation of 1.
- This ensures that each feature contributes equally to the model.

- When your dataset contains features with different units (e.g., age in years, income in dollars, and height in centimeters), StandardScaler helps to bring all features to the same scale.
- You are using algorithms that assume normal distribution of features.

1. Original feature: `[1, 4, 7]`
2. Mean: $\mu = \frac{1+4+7}{3} = 4$
3. Subtract the mean: `[-3, 0, 3]`

```
X = np.array([[1, 2, 3],
              [4, 5, 6],
              [7, 8, 9]])
```

```
X_scaled = [[-1.22474487 -1.22474487 -1.22474487]
            [ 0. 0. 0. ]
            [ 1.22474487 1.22474487 1.22474487]]
```

When to Use Standard Scaler:

- **Algorithms that Assume Normal Distribution:**

1. **Linear Regression**: Assumes that the relationship between the input and output is linear.
2. **Logistic Regression**: Assumes a linear relationship between the input features and the log-odds of the target.
3. **Linear Discriminant Analysis (LDA):** Assumes data is normally distributed within each class.
4. **Support Vector Machines (SVM):** Assumes features have similar scales for optimal performance.
5. **Principal Component Analysis (PCA):** Assumes the data is centered around the origin for variance maximization.
6. **K-Means Clustering**: Assumes features are on similar scales for effective distance calculation.

Normalizer:

- Normalizer is suitable when the goal is to scale individual samples to have unit norm(length).
- This technique is useful when the **direction of the data points** is more important than the magnitude of their distance from the origin.
- Normalizing the data to unit norm ensures that the focus is on the direction of each sample.
  - **Sample data** X = [[3, 4], [1, 2], [4, 5]]
    - **Normalized data: means sum of data point(3,4 = 1).**

X = [[0.6 0.8] [0.4472136 0.89442719] [0.62469505 0.78086881]]

Each row vector in `X_normalized` has a length of 1. For example:

$$\sqrt{0.6^2 + 0.8^2} = \sqrt{0.36 + 0.64} = \sqrt{1} = 1$$

When to Use Normalizer:

1. **Feature Comparison**:
   - **Cosine Similarity**: When you want to measure the cosine similarity between samples. By normalizing, you ensure that the angle between vectors becomes the metric rather than their magnitude.
   - **Clustering**: When using clustering algorithms like K-Means, normalized data can improve the convergence speed and cluster quality, especially if the data has different scales.
   - **Nearest Neighbor**: When you want to perform k-nearest neighbors (KNN) classification or regression, normalizing ensures that all features contribute equally to the distance calculations.

4 Sparse Data:

- When working with sparse data (data with a lot of zeros), normalizing can make algorithms like Support Vector Machines (SVM) and Principal Component Analysis (PCA) perform better by ensuring that features with more non-zero values don't dominate.

5 Text Data:

- **TF-IDF**: In Natural Language Processing (NLP), TF-IDF vectors are often normalized to have unit norm to account for the difference in document lengths and to focus on the relative importance of terms.

```python
from sklearn.preprocessing import Normalizer
nor = Normalizer()
x1 = nor.fit_transform(x1)
pd.DataFrame(x1)
```

Concept of fit_transform and transform:

- **fit_transform**: Use this on your training data to compute the necessary parameters(mean, standard deviation) and apply the transformation in one step.
- **fit** the preprocessing transformers on the training data to learn the necessary parameters.
- **transform** the training data using the fitted transformers.

- **transform**: Use this on your test data (or any new data) to apply the transformation using the parameters computed from the training data.
- For test data we do not use ‘fit’ and we are using the same parameters as calculated for training.

- Converting data types using `**pd.to_numeric**`.

Common Data Types and Their Handling

- **Numerical Data:**
  - **Integers**: Whole numbers.
  - **Floats**: Decimal numbers.
  - Handling: Ensure numerical columns are in the correct format and handle missing values appropriately.

Numerical Data:

```
[11]: import pandas as pd

      df = pd.DataFrame({'integers': [1, 2, 3], 'floats': [1.1, 2.2, 3.3]})
      df['integers'] = df['integers'].astype(int)
      df['floats'] = df['floats'].astype(float)
```

```
[12]: df
```

[12]:

	integers	floats
0	1	1.1
1	2	2.2
2	3	3.3

```
[13]: df.dtypes
```

```
[13]: integers      int32
      floats      float64
      dtype: object
```

Categorical Data:

- **Nominal**: Categories without a specific order (e.g., color: red, green, blue).
- **Ordinal**: Categories with a specific order (e.g., rating: low, medium, high).
- Handling: Encode categorical variables using techniques such as label encoding or one-hot encoding.

- **Label encoding:**
- **Label encoding** is a technique used to convert ordinal type of categorical data into numerical format.
- It assigns a unique integer to each category in the categorical variable.
- This is particularly useful for machine learning algorithms that require numeric input.
- **dataset[‘Col_Name'] = dataset[' Col_Name '].astype('category')**
- **dataset[‘Col_Name'] = dataset[‘Col_Name'].cat.codes**
- convert the [‘Col_Name'] to categorical type using astype('category').
- then use cat.codes to assign numeric labels to each category in the [‘Col_Name'] and create a new column with the encoded values.

- **One-hot encoding:**
- **One-hot encoding** is a technique used to convert Nominal type categorical data into a binary format, where each category is represented as a binary vector.
- It creates new binary columns (also known as dummy variables) for each category, with a value of 1 indicating the presence of that category and 0 indicating absence.
- After OHE drop one variable. Here is python code:
- **dataset = pd.get_dummies(dataset, columns = [‘Col_Name'])**

Where:

“pd.get_dummies()” is a pandas function used for one-hot encoding categorical variables.

“columns=['Col_Name']” specifies the column(s) in the Data Frame that you want to encode.

Categorical Data:

```
from sklearn.preprocessing import LabelEncoder, OneHotEncoder

df = pd.DataFrame({'category': ['red', 'green', 'blue']})
label_encoder = LabelEncoder()
df['category_encoded'] = label_encoder.fit_transform(df['category'])

one_hot_encoder = OneHotEncoder(sparse=False)
df_one_hot = pd.DataFrame(one_hot_encoder.fit_transform(df[['category']]), columns=one_hot_encoder.categories_)
```

```
[15]: df_one_hot
```

[15]:

	blue	green	red
0	0.0	0.0	1.0
1	0.0	1.0	0.0
2	1.0	0.0	0.0

```
[16]: df_one_hot.dtypes

[16]: blue      float64
      green     float64
      red       float64
      dtype: object
```

dtype: object

```
[17]: df['category_encoded']

[17]: 0    2
      1    1
      2    0
      Name: category_encoded, dtype: int32
```

Datetime Data:

```
df = pd.DataFrame({'date': ['2020-01-01', '2021-01-01']})
df['date'] = pd.to_datetime(df['date'])
df['year'] = df['date'].dt.year
df['month'] = df['date'].dt.month
df['day'] = df['date'].dt.day
```

```
[22]: df['date']

[22]: 0   2020-01-01
      1   2021-01-01
      Name: date, dtype: datetime64[ns]

[23]: df['year']

[23]: 0    2020
      1    2021
      Name: year, dtype: int32
```

```
[24]: df['month']

[24]: 0    1
      1    1
      Name: month, dtype: int32

[25]: df['day']

[25]: 0    1
      1    1
      Name: day, dtype: int32
```

Convert Data Types:

```
df['integer'] = df['integer'].astype(int)
df['float'] = df['float'].astype(float)
df['category'] = df['category'].astype('category')
df['date'] = pd.to_datetime(df['date'])
```

Example

```
[59]: df['num_numerical']  = pd.to_numeric(df['number'], errors='coerce', downcast='integer')
```

```
[60]: df.head()
```

[60]:

	Cabin	Ticket	number	Survived	num_numerical
0	NaN	A/5 21171	5	0	5.0
1	C85	PC 17599	3	1	3.0
2	NaN	STON/O2. 3101282	6	1	6.0
3	C123	113803	3	1	3.0
4	NaN	373450	A	0	NaN

pd.to_numeric(df['number'], errors='coerce', downcast='integer')

- **pd.to_numeric:** This function is used to convert argument to a numeric type.
- **df['number']**: This specifies the column number from the DataFrame df that we want to convert.
- **errors='coerce':** This argument tells the function how to handle errors during the conversion process. Specifically:**'coerce':** This means that any values that cannot be converted to a numeric type will be set to NaN (Not a Number).
- **downcast='integer':** This argument attempts to downcast the numeric type to the smallest possible integer subtype, which helps in saving memory.
- For example, if all values can be represented by a smaller integer type (like int8), it will use that type instead of a larger one (like int64).

[61]:
```python
df['num_categorical'] = np.where(df['num_numerical'].isnull(), df['number'], np.nan)
```

[63]:
```python
df.head(20)
```

[63]:

	Cabin	Ticket	number	Survived	num_numerical	num_categorical
0	NaN	A/5 21171	5	0	5.0	NaN
1	C85	PC 17599	3	1	3.0	NaN
2	NaN	STON/O2. 3101282	6	1	6.0	NaN
3	C123	113803	3	1	3.0	NaN
4	NaN	373450	A	0	NaN	A
5	NaN	330877	2	0	2.0	NaN
6	E46	17463	2	0	2.0	NaN

np.where(df['num_numerical'].isnull(), df['number'], np.nan)

1. np.where(condition, x, y):

- This function from the NumPy library is used for element-wise selection from two arrays (x and y) based on a condition. It returns elements chosen from x or y depending on the condition.
- condition: This specifies the condition to be checked.
- x: The values to select where the condition is True.
- y: The values to select where the condition is False.

2. df['num_numerical'].isnull():

- This checks for NaN values in the num_numerical column of the DataFrame df. It returns a boolean Series where True indicates the presence of a NaN value and False indicates the absence of a NaN value.

3. df['number']:

- This specifies the original number column from the DataFrame df, which contains the original values before any numeric conversion.

4. np.nan:

- This specifies that NaN should be used where the condition is False.

Data Balancing

- Importance of balanced datasets.
- Data is said to be imbalance if twice of minority class is less than the majority class.
- To find it, check the count in dependent variables.
- Two popular approach to solve the problem:
  - **Oversampling**
  - **SMOTE**

```
[10]: y.value_counts()

[10]: 0    284315
      1       492
      Name: Class, dtype: int64
```

RandomOverSampler

- Simply duplicates random instances of the minority class to increase its representation in the dataset.
- This can be effective but may lead to overfitting as the same instances are repeated multiple times.

```
[ ]:  #!pip install imblearn
      import imblearn
      # split the data into feature variable and label/result/output variable
      x = dataset.iloc[:,:-1]
      y = dataset.iloc[:,-1]

[14]: from imblearn.over_sampling import RandomOverSampler
      over = RandomOverSampler()
      x_over, y_over = over.fit_resample(x,y)

[15]: y_over.value_counts()

[15]: 0    284315
      1    284315
      Name: Class, dtype: int64

[16]: y_over.shape

[16]: (568630,)
```

SMOTE
(Synthetic minority oversampling technique)

- Generates synthetic samples by interpolating between existing minority class instances.
- This technique creates more diverse samples compared to simple duplication, potentially reducing overfitting.

```
[17]: from imblearn.over_sampling import SMOTE
      smote = SMOTE()
      x_smote, y_smote = smote.fit_resample(x,y)
      y_smote.value_counts()

[17]: 0    284315
      1    284315
      Name: Class, dtype: int64
```

Cross Tabs for Feature Relationships

A cross tabulation presents the frequency distribution of variables in a matrix format. Each cell in the table represents the count or frequency of the occurrences of the specific combination of categories from the variables.

Why Use Cross Tabulation?

1. **Identify Relationships:** It helps in identifying relationships between categorical variables.
2. **Detect Patterns:** It can detect patterns and trends in the data.
3. **Summarize Data:** It provides a compact summary of data.
4. **Inform Decision-Making:** It aids in making informed decisions based on the observed relationships.

Example: Using Cross Tabulation in Python

Let's consider a dataset where we want to examine the relationship between two categorical variables: `Gender` and `Purchased`.

Sample Data:

```plaintext
| Gender | Purchased |
|--------|-----------|
| Male   | Yes       |
| Female | No        |
| Female | Yes       |
| Male   | No        |
| Female | Yes       |
| Male   | Yes       |
```

Creating a Cross Tabulation:

1. Load the Data:

```python
import pandas as pd

data = {
    'Gender': ['Male', 'Female', 'Female', 'Male', 'Female', 'Male'],
    'Purchased': ['Yes', 'No', 'Yes', 'No', 'Yes', 'Yes']
}

df = pd.DataFrame(data)
```

2. Create Cross Tab:

```python
cross_tab = pd.crosstab(df['Gender'], df['Purchased'])
print(cross_tab)
```

Output:

```plaintext
Purchased  No  Yes
Gender
Female      1    2
Male        1    2
```

3. **Add Margins** (to get totals):

```python
cross_tab_with_totals = pd.crosstab(df['Gender'], df['Purchased'], margins=True)
print(cross_tab_with_totals)
```

Output:

```plaintext
Purchased  No  Yes  All
Gender
Female      1    2    3
Male        1    2    3
All         2    4    6
```

Interpretation:

- The table shows that there are 3 females and 3 males in the dataset.
- Among the females, 2 have made a purchase, and 1 has not.
- Among the males, 2 have made a purchase, and 1 has not.
- The total number of purchases is 4, and the total number of non-purchases is 2.

Visualization:

To make the relationship more visually intuitive, you can plot the cross-tabulated data.

1. Bar Plot:

```python
cross_tab.plot(kind='bar', stacked=True)
import matplotlib.pyplot as plt
plt.title('Purchase Frequency by Gender')
plt.xlabel('Gender')
plt.ylabel('Frequency')
plt.show()
```

2. Heatmap:

```python
import seaborn as sns

sns.heatmap(cross_tab, annot=True, cmap="YlGnBu", cbar=False)
plt.title('Heatmap of Purchase Frequency by Gender')
plt.show()
```

Exploratory Data Analysis (EDA):

- **Definition:** Exploratory Data Analysis (EDA) is a critical step in the data analysis process. It involves examining and summarizing the main characteristics of a dataset, often using visual methods.
- Here are some key steps and techniques you can use during EDA
- **Techniques:**
  - **Data visualization**: Histograms, scatter plots, box plots.
  - **Summary statistics**: Mean, median, standard deviation.
  - **Outlier detection**: Identifying data points that deviate significantly from the rest of the dataset

What we can do in EDA

1. Understand the Data Structure

- **Load the Data:** Import the dataset and understand its structure.

python   Copy code

```python
import pandas as pd
df = pd.read_csv('your_dataset.csv')
df.head()
df.info()
df.describe()
```

2. Handling Missing Values

- **Identify Missing Values:** Check for missing values.

python   Copy code

```python
df.isnull().sum()
```

- **Handle Missing Values:** Depending on the context, you can drop or fill missing values.

python   Copy code

```python
df.dropna()  # Drop missing values
df.fillna(df.mean(), inplace=True)  # Fill missing values with mean
```

3. Data Cleaning

- **Remove Duplicates:** Identify and remove duplicate rows.

python Copy code

```python
df.drop_duplicates(inplace=True)
```

- **Handle Outliers:** Detect and treat outliers.

python Copy code

```python
df.boxplot(column='column_name')
```

4. Data Transformation

- **Feature Engineering:** Create new features from existing ones.

python Copy code

```python
df['new_feature'] = df['existing_feature1'] / df['existing_feature2']
```

- **Encoding Categorical Variables:** Convert categorical variables to numerical ones.

python Copy code

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

5. Univariate Analysis

- **Summary Statistics:** Get descriptive statistics for individual features.

python Copy code

```python
df['column_name'].describe()
```

- **Visualization:**
  - Histograms:

  python Copy code

  ```python
  df['column_name'].hist()
  ```

  - Box Plots:

  python Copy code

  ```python
  df.boxplot(column='column_name')
  ```

6. Bivariate Analysis

- **Correlation**: Check correlations between numerical features.

python Copy code

```python
df.corr()
```

- **Visualization**:
    - Scatter Plots:

    python Copy code

    ```python
    df.plot.scatter(x='feature1', y='feature2')
    ```

    - Pair Plots:

    python Copy code

    ```python
    import seaborn as sns
    sns.pairplot(df)
    ```

7. Multivariate Analysis

- **Heatmaps:** Visualize correlation matrices.

```python
sns.heatmap(df.corr(), annot=True)
```

- **Group By:** Group data and perform aggregations.

```python
df.groupby('categorical_column').mean()
```

8. Distribution Analysis

- **Density Plots:** Examine the distribution of numerical features.

```python
df['column_name'].plot(kind='density')
```

- **QQ Plots:** Assess if data follows a certain distribution.

```python
import statsmodels.api as sm
import matplotlib.pyplot as plt
sm.qqplot(df['column_name'], line ='45')
plt.show()
```

9. Feature Relationships

- **Cross Tabs:** Examine relationships between categorical features.

```python
pd.crosstab(df['categorical_feature1'], df['categorical_feature2'])
```

10. Dimensionality Reduction

- **PCA:** Apply Principal Component Analysis for high-dimensional data.

```python
from sklearn.decomposition import PCA
pca = PCA(n_components=2)
principalComponents = pca.fit_transform(df)
```

Visualization Libraries

- **Matplotlib**: Basic plotting.

```python
import matplotlib.pyplot as plt
plt.plot(df['column_name'])
```

- **Seaborn**: Advanced visualizations.

```python
sns.boxplot(x='categorical_feature', y='numerical_feature', data=df)
```

- **Plotly**: Interactive plots.

```python
import plotly.express as px
fig = px.scatter(df, x='feature1', y='feature2')
fig.show()
```

Visualization

- Using plots for making inferences in machine learning involves visualizing data to understand its structure, relationships, and patterns, which can guide feature selection, model choice, and evaluation.

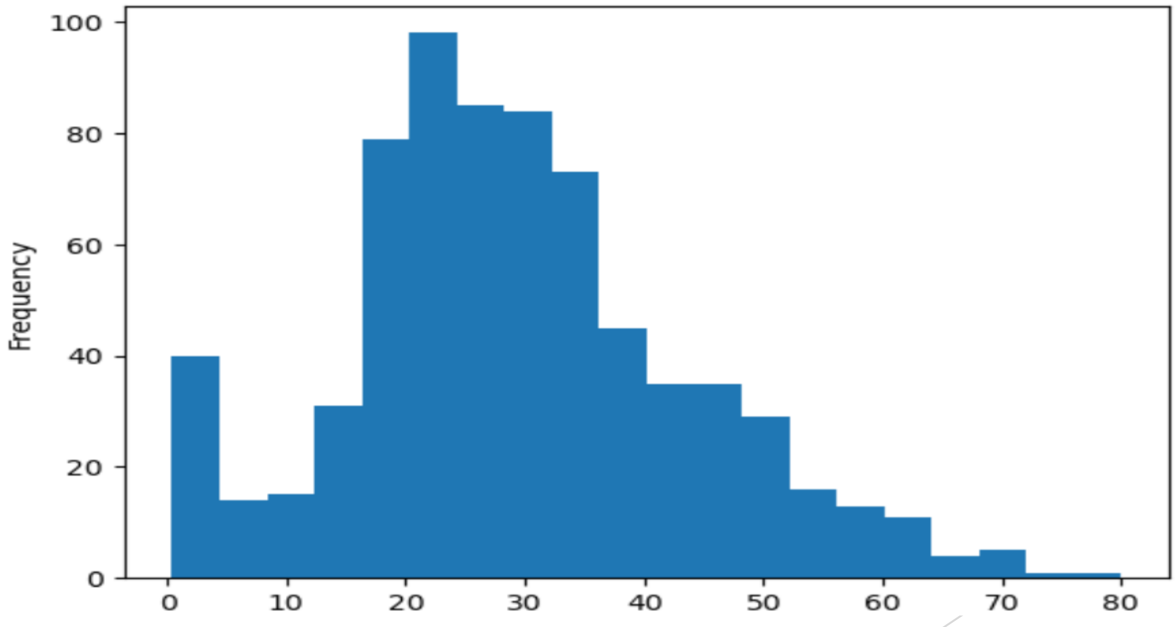
Visualization Techniques for EDA

- Histograms
- Box plots
- Scatter plots
- Pair plots
- Correlation matrix and heatmaps
- Bar plots
- Count plots
- Violin plots

Histograms

- **Purpose**: Understand the distribution of individual variables.
- **Inference**: Identify skewness, outliers, and the presence of multiple modes in the data.

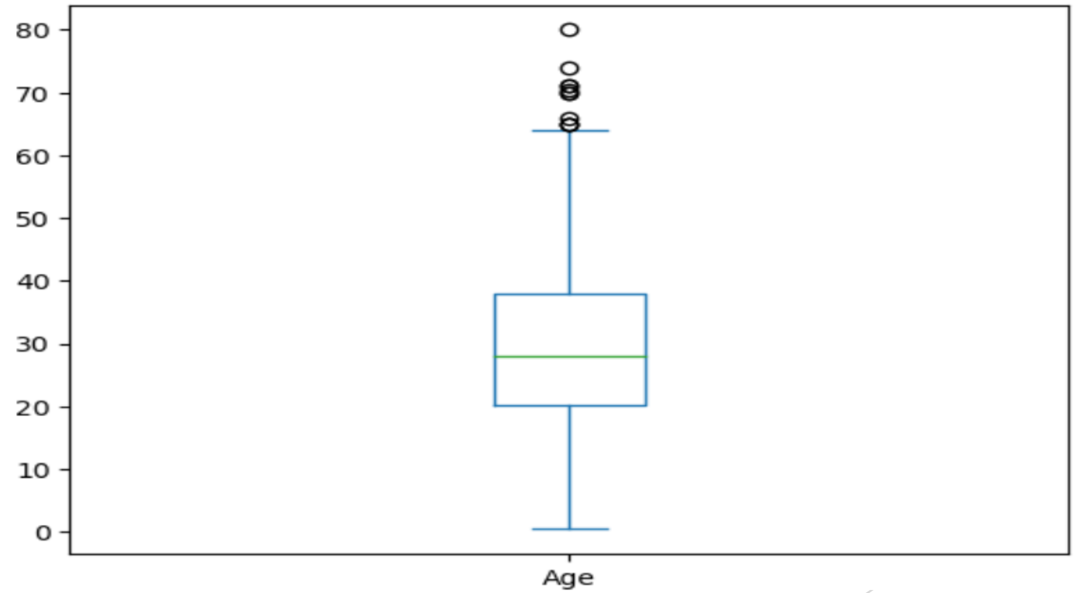
```
dataset['Age'].plot(kind='hist', bins=20)
```



Box Plots

- **Purpose:** Summarize the distribution of a dataset.
- **Inference:** Detect outliers, understand the spread and symmetry of the data.

```
dataset['Age'].plot(kind='box')
```

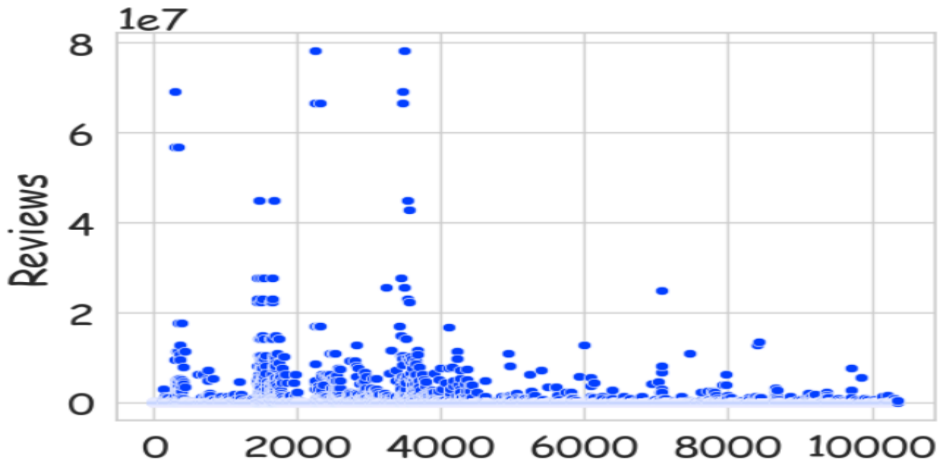


Scatter Plots

- **Purpose:** Visualize the relationship between two continuous variables.
- **Inference**: Identify correlations, clusters, and potential outliers

```
[142]: sns.set_theme(style='whitegrid',palette='bright',font='cursive',font_scale=1.8)
```

```
[148]: sns.scatterplot(y=df.Reviews,x=df.Reviews.index)
       plt.show()
```



Pair Plots (Scatterplot Matrix)

- **Purpose**: Visualize pairwise relationships between multiple variables.
- **Inference**: Detect relationships between pairs of features, spot trends, clusters, and outliers.

Correlation Matrix and Heatmaps

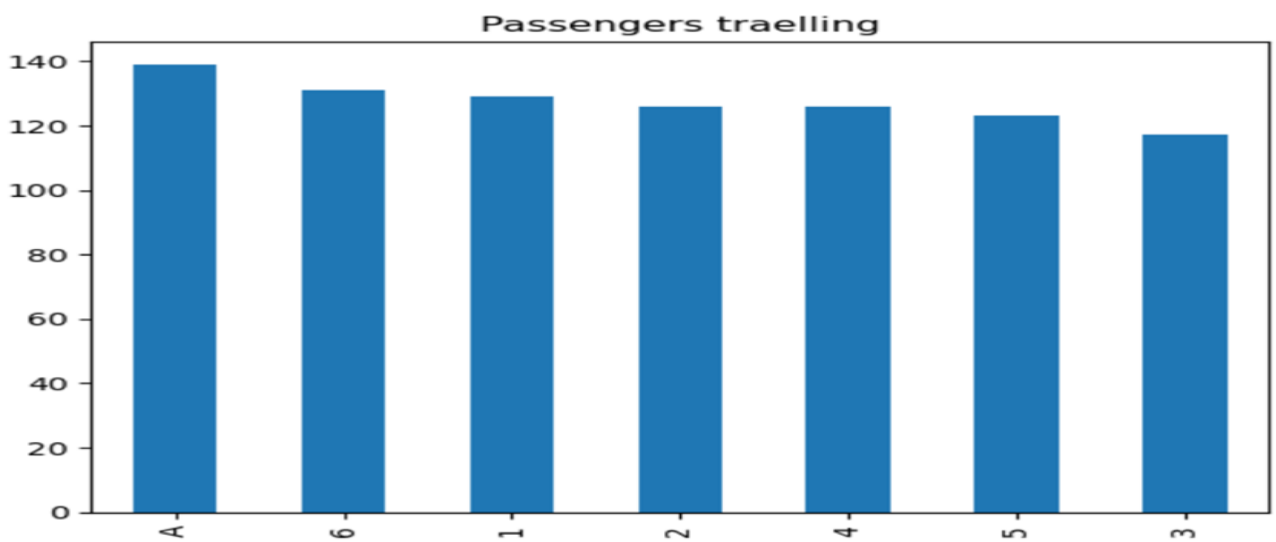
- **Purpose**: Show the correlation coefficients between variables.
- **Inference**: Identify highly correlated features that might be redundant.

```
plt.figure(figsize=(8, 6))
sns.heatmap(df.corr(), annot=True, cmap='coolwarm')
plt.title('Correlation Heatmap')
plt.show()
```

Bar Plots

- **Purpose**: Compare categorical data.
- **Inference**: Understand the frequency distribution of categorical features.

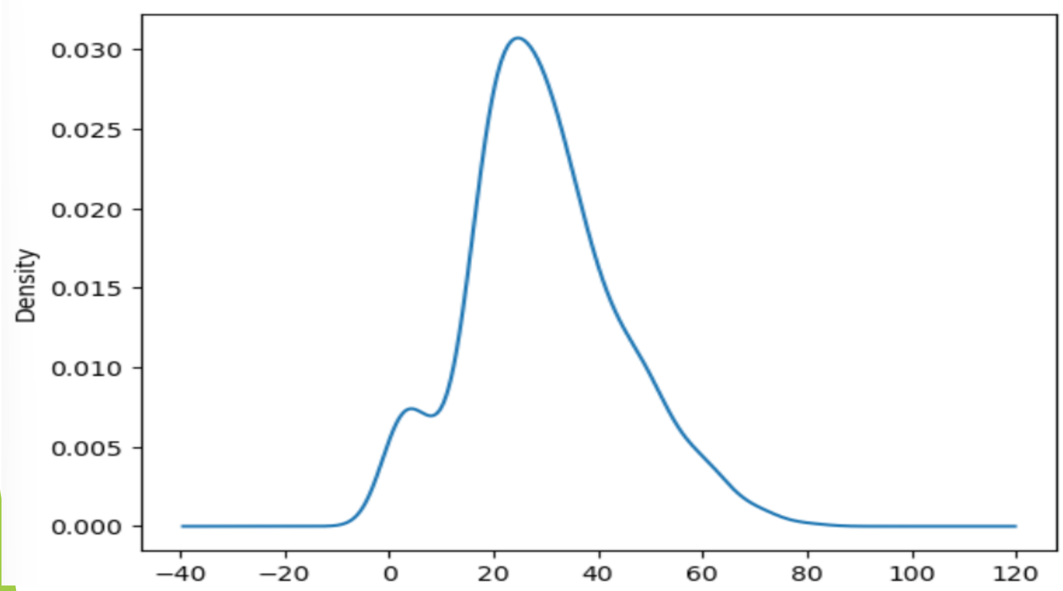
```
fig = df['number'].value_counts().plot.bar()
fig.set_title("Passengers traelling")
```



Density plots

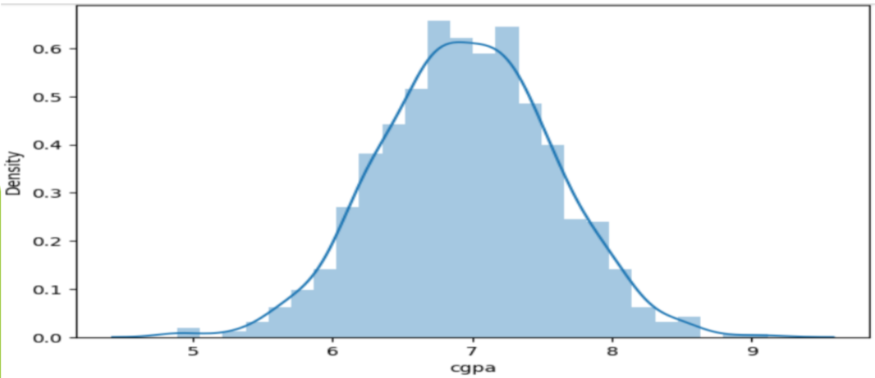
- Density plots display the probability density function of a continuous variable. They are useful for visualizing the overall shape of the distribution and comparing multiple distributions.

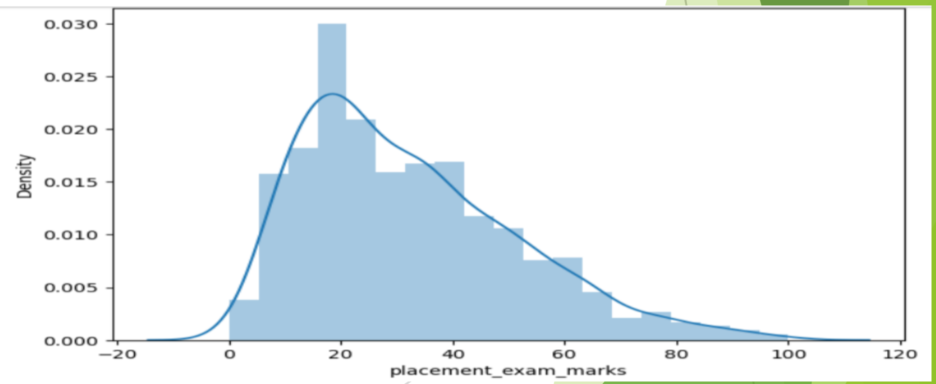
```
dataset['Age'].plot(kind='kde')
```



Distribution plot

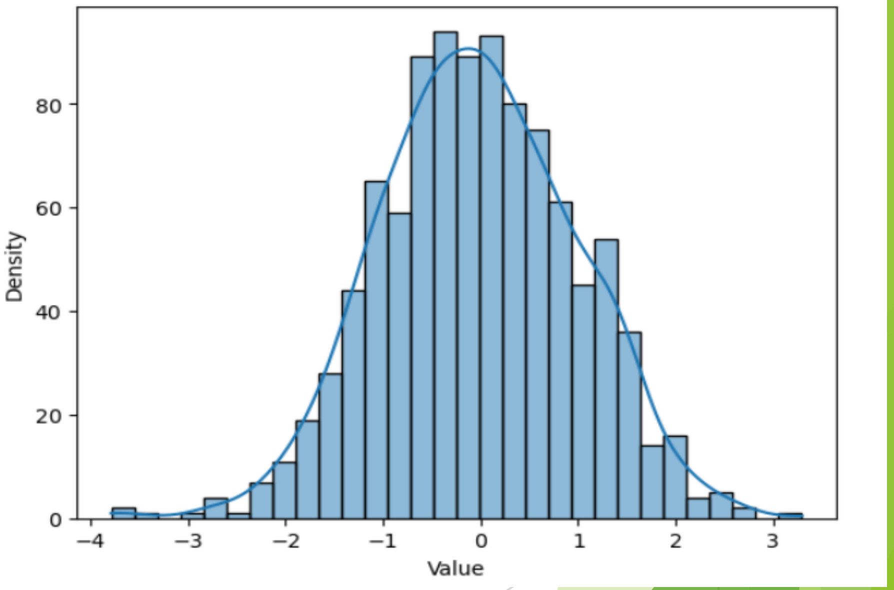
- A **distribution plot** is a visualization that combines aspects of a histogram and a kernel density plot to show the distribution of a continuous variable.
- It is useful for understanding the distribution of data points in a dataset and identifying patterns such as skewness, kurtosis, and the presence of outliers.
- sns.histplot is used instead of sns.distplot.
- The parameter kde=True adds the KDE line to the histogram





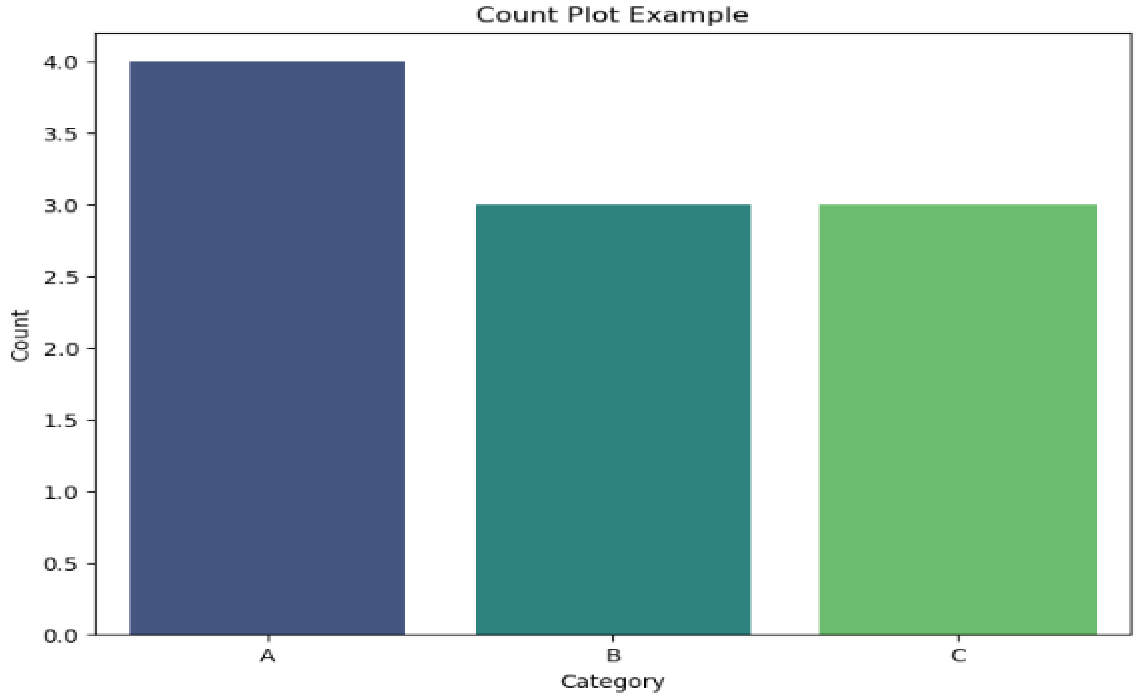
- In recent versions of Seaborn (0.11.0 and later), sns.distplot has been deprecated.
- Instead, you should use sns.histplot or sns.kdeplot for similar functionality. Here's how to create a similar plot using sns.histplot

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

# Example data
import numpy as np
data = np.random.randn(1000)  # Generating random data

# Creating a histogram with KDE
sns.histplot(data, kde=True, bins=30)

# Adding labels and title
plt.xlabel('Value')
plt.ylabel('Density')
```



Count Plots

- **Purpose:** Show the counts of observations in each categorical bin.
- **Inference**: Detect the distribution of categorical features.



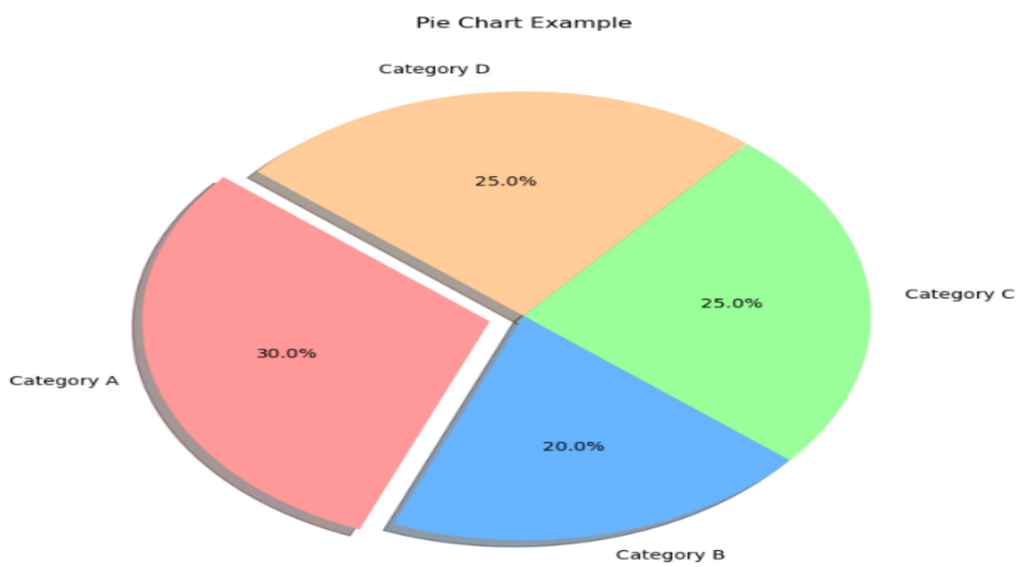
```python
import seaborn as sns
import matplotlib.pyplot as plt

# Sample data
import pandas as pd
data = pd.DataFrame({
    'Category': ['A', 'B', 'A', 'C', 'B', 'A', 'C', 'C', 'B', 'A']
})

# Create the count plot
plt.figure(figsize=(8, 6))  # Optional: Set figure size
sns.countplot(data=data, x='Category', palette='viridis')

# Add title and labels
plt.title('Count Plot Example')
plt.xlabel('Category')
plt.ylabel('Count')

# Show plot
plt.show()
```

Explanation:

- `data`: The DataFrame containing the data.
- `x`: The column name containing the categorical data you want to plot.
- `palette`: Specifies the color palette for the bars. You can use predefined palettes like 'viridis', 'coolwarm', etc., or define your own colors.

Pie Plot

- A pie plot (or pie chart) is a circular statistical graphic that is divided into slices to illustrate numerical proportions. Each slice represents a category's proportion to the whole dataset. Pie charts are useful for showing the relative sizes of parts to a whole, making it easy to compare the parts of a single categorical variable.



```python
import matplotlib.pyplot as plt

# Sample data
labels = ['Category A', 'Category B', 'Category C', 'Category D']
sizes = [30, 20, 25, 25]  # Corresponding sizes of each category
colors = ['#ff9999','#66b3ff','#99ff99','#ffcc99']  # Optional: colors for each slice
explode = (0.1, 0, 0, 0)  # Optional: explode the first slice for emphasis

# Create the pie chart
plt.figure(figsize=(8, 8))  # Optional: Set figure size
plt.pie(sizes, explode=explode, labels=labels, colors=colors, autopct='%1.1f%%', shadow=True, startangle=140)

# Add title
plt.title('Pie Chart Example')

# Show plot
plt.show()
```

Explanation:

- `labels`: The names of the categories to be shown in the pie chart.
- `sizes`: The values corresponding to each category.
- `colors`: The colors for each slice of the pie chart. This is optional.
- `explode`: A tuple to indicate which slice to "explode" (or pull out) for emphasis. The first value `0.1` means the first slice will be slightly pulled out.
- `autopct`: A string format for the percentage labels on each slice.
- `shadow`: Adds a shadow effect to the pie chart.
- `startangle`: Rotates the pie chart to start from a specified angle.

Feature Elimination Method

- If number of features are too large to handle than it is wise approach to remove insignificant features.
- There two popular approach.
- PCA :Principal Component Analysis
- RFE: Recursive Feature Elimination

Recursive Feature Elimination (RFE)

- **Purpose:**
- RFE is a feature selection method that iteratively removes less important features based on the model's performance, identifying the most influential features for predicting the target variable.

- Mathematical Basis:

1.Feature Importance:

- RFE uses an estimator (e.g., Logistic Regression, Random Forest, etc.) that provides feature importance, such as weights or coefficients.

Steps:

- Train the model on the dataset.
- Rank features based on their importance scores.
- Remove the least important feature(s).
- Repeat until the desired number of features remains.

- **Advantages:**
- Identifies the most critical features for the model.
- Helps improve model performance by eliminating redundant or irrelevant features.
- Works well with small to medium datasets.
- **Disadvantages:**
- Computationally expensive for large datasets.
- Performance depends on the chosen estimator.

Code Example:

```
from sklearn.feature_selection import RFE
from sklearn.linear_model import LogisticRegression
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split

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

# Train-test split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3,
random_state=42)

# Logistic Regression as estimator
logitR = LogisticRegression()

# Apply RFE to select top 2 features
selector = RFE(estimator=logitR, n_features_to_select=2, step=1)
selector.fit(X_train, y_train)

# Selected features
print("Selected Features:", selector.support_)
```

```python
from sklearn.feature_selection import RFE
from sklearn.linear_model import LogisticRegression
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split

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

# Train-test split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)

# Logistic Regression as estimator
logitR = LogisticRegression()

# Apply RFE to select top 2 features
selector = RFE(estimator=logitR, n_features_to_select=2, step=1)
selector.fit(X_train, y_train)

# Selected features
print("Selected Features:", selector.support_)
print("Feature Ranking:", selector.ranking_)
```

Principal Component Analysis (PCA)

- **Purpose:**
- PCA is a dimensionality reduction technique that transforms the dataset into a lower-dimensional space while preserving as much variance as possible.

Mathematical Basis:

1. **Covariance Matrix**:
   - Compute the covariance matrix of the dataset.

$$\text{Covariance Matrix: } \Sigma = \frac{1}{n-1} \sum_{i=1}^{n} (X_i - \mu)(X_i - \mu)^T$$

2. **Eigenvalues and Eigenvectors**:
   - Calculate eigenvalues and eigenvectors of the covariance matrix.
   - Eigenvalues represent the variance captured by each principal component.
   - Eigenvectors represent the directions of maximum variance.
3. **Principal Components**:
   - Transform the original dataset X into a new coordinate system using the eigenvectors.

$$Z = X \cdot W$$

Where W is the matrix of eigenvectors.

Advantages:

- Reduces dimensionality while retaining variance.
- Removes multicollinearity by creating uncorrelated components.
- Improves computational efficiency for large datasets.

Disadvantages:

- Components are linear combinations of original features, losing interpretability.
- Sensitive to scaling; features must be normalized.

Key Differences

Aspect	RFE	PCA
Purpose	Feature Selection	Dimensionality Reduction
Method	Eliminates less important features iteratively	Transforms data into new components
Model Dependence	Model-dependent	Model-independent
Interpretability	Retains original feature meaning	Loses interpretability
Computational Cost	Higher (iterative process)	Lower for fewer components