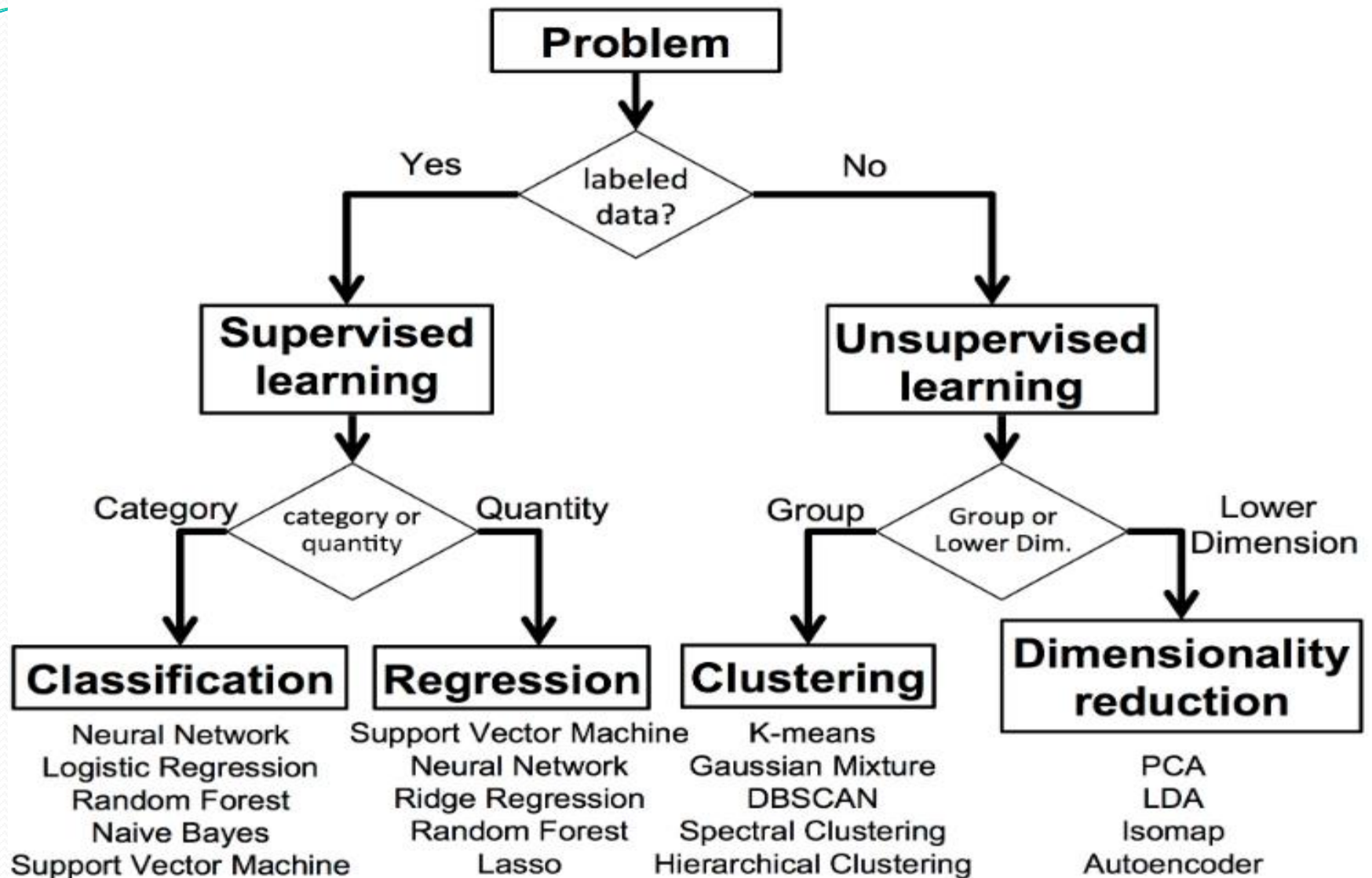



DATA PREPROCESSING

Let us know about pretreatment of Data before
modeling



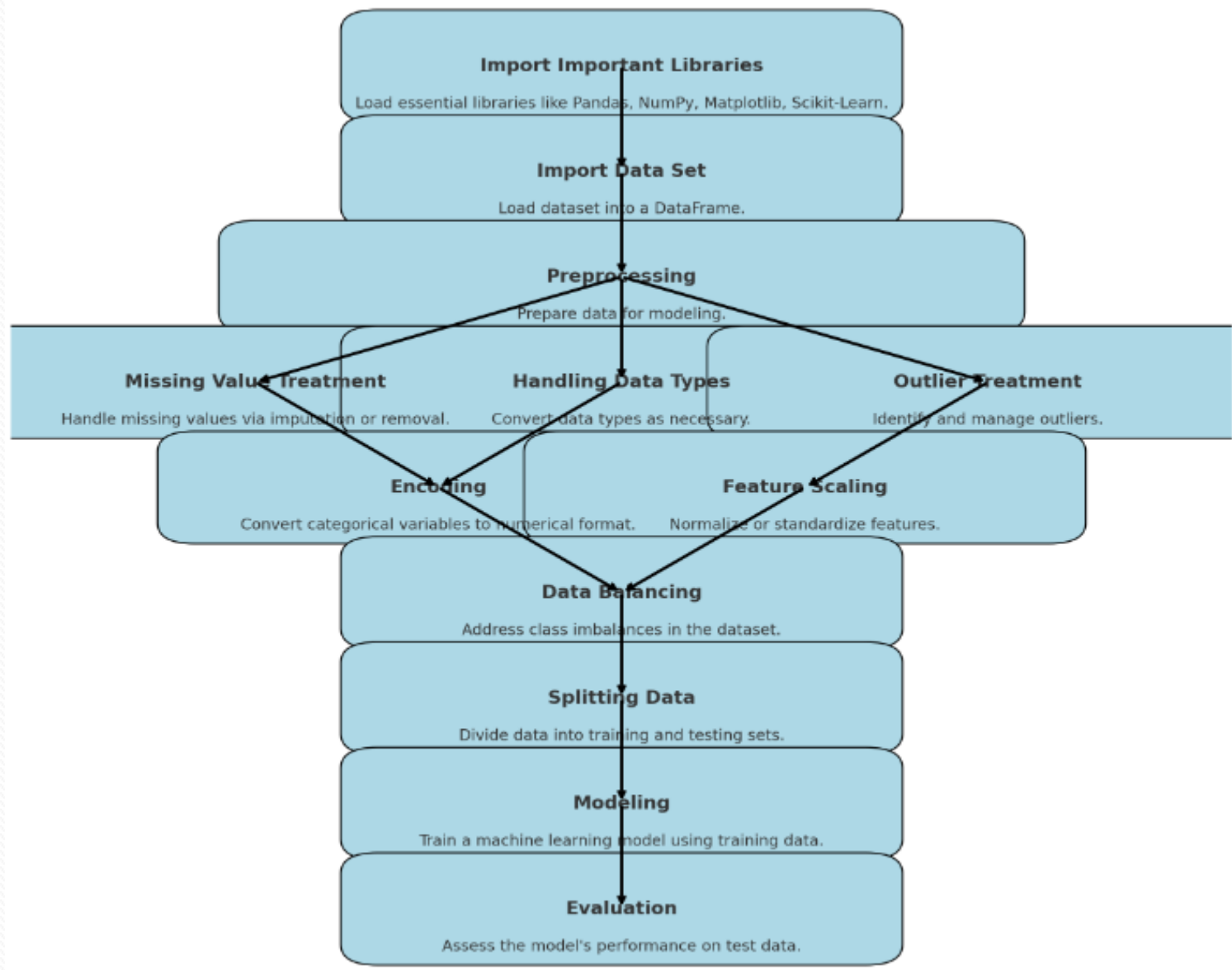
Difference between Stats and ML

- **Stats:** Draft the model for Problem and ask for Data
- **ML:** You have Data so give the Solution.
- **Statistical Way of thinking** said, you formulate a problem and then you get the Data to solve that problem.
- While **ML approach** says this is the Data tell me what the data is telling you.
- **Statistician** find problem in job search for Data scientist: Because when interviewer ask the question” Here is my data .What you can say?
- And **Statistician Answer** “ What do you want to know and business guys says that’s Why I want to hire you.
- And **Statistician** says What if you don’t tell me what you want to know, how do I know what to tell you.

- 
- And this goes round and round and no one happy with this process.
 - So the difference is the way these communities approach the things.
 - Moreover in the world **machine learning** '**data is cheaper**' but the '**Question is Expensive**' and you are paid for asking the right question.
 - While in the world of **Statistician**, "**question is cheap, but the data is expensive**". You paid for collecting the data.
 - But both have their own relevance.

Basic things to do with every data before modeling:

- Import important libraries
- Import Data set
- Preprocessing:
 - Missing value treatment
 - Handling data types
 - Outlier Treatment
 - Encoding
 - Feature Scaling
 - Data Balancing



Import important libraries

- import os
- import numpy as np
- import pandas as pd
- import matplotlib.pyplot as plt
- import seaborn as sns
- sns.set()
- %matplotlib inline
- import warnings
- warnings.filterwarnings('ignore')

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

import warnings
warnings.filterwarnings('ignore')
```

But Why: let us understand

- **import os:**
 - **Purpose:** Provides functions to interact with the operating system, such as file and directory manipulation.
 - **Example Usage:** `os.listdir()` lists files and directories in the specified path.
- **import numpy as np:**
 - **Purpose:** A fundamental package for scientific computing with Python. It provides support for arrays, matrices, and many mathematical functions.
 - **Example Usage:** `np.array([1, 2, 3])` creates a numpy array.

➤ **import pandas as pd:**

- **Purpose:** A powerful data manipulation and analysis library that provides data structures like DataFrames to work with structured data.

- **Example Usage:**

`pd.DataFrame({'A': [1, 2], 'B': [3, 4]})` creates a DataFrame.

➤ **import matplotlib.pyplot as plt:**

- **Purpose:** A plotting library used for creating static, animated, and interactive visualizations in Python.

- **Example Usage:**

`plt.plot([1, 2, 3], [4, 5, 6])` plots a simple line graph.

➤ **import seaborn as sns:**

- **Purpose:** A statistical data visualization library based on matplotlib. It provides a high-level interface for drawing attractive and informative statistical graphics.

- **Example Usage:**

`sns.lineplot(x='A', y='B', data=df)` creates a line plot.

➤ **sns.set():**

- **Purpose:** Sets the aesthetic parameters in one step for all the plots created using seaborn.

- **Example Usage:** Automatically sets the seaborn plot aesthetics to a default theme.

➤ **%matplotlib inline:**

- **Purpose:** A magic command used in Jupyter notebooks to display matplotlib plots inline within the notebook.

- **Example Usage:** Ensures that plots appear directly below the code cells that produce them.

➤ **import warnings:**

- **Purpose:** A built-in library to handle warnings. It allows the user to display, filter, or ignore warnings.

- **Example Usage:** `warnings.warn("This is a warning message")` would display a warning.

➤ **warnings.filterwarnings('ignore'):**

- **Purpose:** Suppresses the display of warnings.

- **Example Usage:** Prevents warnings from appearing in the output, which is useful for a cleaner output during demonstrations or presentations.

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()`

```
USAHousing = pd.read_csv(r'C:\Users\gurjinder\Desktop\Pankaj Sir\USA_Housing.csv')
USAHousing = pd.read_csv('C://Users//gurjinder//Desktop//Pankaj Sir//USA_Housing.csv')
USAHousing.head()
```

After loading data, before Data Preprocessing certain information gathered using following commands:

- **Step 1:** To get information about:

- Number of rows,
- Number of columns,
- Their name,
- Their type,
- not-null count

code: **data.info()**

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 891 entries, 0 to 890
Data columns (total 11 columns):
#   Column      Non-Null Count  Dtype
---  -
0   PassengerId  891 non-null    int64
1   Survived     891 non-null    int64
2   Pclass      891 non-null    int64
3   Name         891 non-null    object
4   Sex         891 non-null    object
5   Age         714 non-null    float64
6   SibSp       891 non-null    int64
7   Parch       891 non-null    int64
8   Ticket      891 non-null    object
9   Fare        891 non-null    float64
10  Embarked    891 non-null    object
dtypes: float64(2), int64(5), object(4)
memory usage: 76.7+ KB
```

- **Step 2:** To get the counts of rows and column can be find out using following command

```
[173]: df.shape
```

```
[173]: (10840, 13)
```

- **Step 3:** To get the counts of rows, mean, std, min, 25%, 50%, 75% and max can be find out using following command

```
dataset['Age'].describe()
```

```
[13]:
```

```
count      714.000000
```

```
mean       29.699118
```

```
std        14.526497
```

```
min         0.420000
```

```
25%        20.125000
```

```
50%        28.000000
```

```
75%        38.000000
```

```
max         80.000000
```

```
Name: Age, dtype: float64
```

- **Step 4:** To get the type of data of each independent variable (Idv), we use the following code.
 - It is required to know since for processing data should be in correct type.

Numeric vs Nonnumeric Data

```
[139]: df.dtypes
```

```
[139]: App                object
       Category          object
       Rating            float64
       Reviews           object
       Size              object
       Installs          object
       Type              object
       Price             object
       Content Rating    object
       Genres            object
       Last Updated      object
       Current Ver       object
       Android Ver       object
       dtype: object
```

```
[140]: # Reviews- Convert to Numeric
       # Size- Convert to Numeric
       # Installs- Convert to Numeric
       # Price- Convert to Numeric
       # Category or Geners either we can drop since both contains the same information
```

- **Step 5:** To know the unique values of a column use:

```
[56]:
```

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

```
[56]:
```

```
array(['5', '3', '6', 'A', '2', '1', '4'], dtype=object)
```

- **Step 6:** To know the unique values of a column use:

```
[57]:
```

```
df['number'].value_counts()
```

```
[57]:
```

```
A      139  
6      131  
1      129  
2      126  
4      126  
5      123  
3      117
```

```
Name: number, dtype: int64
```


- **Step 7:** To get the list of the name of the columns:

```
[138]: df.columns
```

```
[138]: Index(['App', 'Category', 'Rating', 'Reviews', 'Size', 'Installs', 'Type',  
            'Price', 'Content Rating', 'Genres', 'Last Updated', 'Current Ver',  
            'Android Ver'],  
          dtype='object')
```

DataPreprocessing

- Missing value treatment
- Handling data types
- Encoding
- Outlier Treatment
- Data Balancing
- Feature Scaling

Handling Missing Value

- Find the counts of missing value:
 - `data.isnull().sum()`
 - # This will give null values count in each variable.
 - To find the counts of missing value in terms of percentage:
 - `data.isnull().sum()/len(data)*100`
 - If any variable have missing value > 25% , drop it else impute the missing value.
- `data = data.drop(['name of column'], axis = 1)`**

```
PassengerId      0
Survived          0
Pclass           0
Name             0
Sex              0
Age             177
SibSp            0
Parch            0
Ticket           0
Fare             0
Cabin           687
Embarked         2
dtype: int64
```

Check whether variable is numerical or categorical

- If variable is categorical impute with 'MODE'.
- Example:

```
[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 categorical column  
       # - We can fill the null values with the mode value
```

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

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

```
[155]: 4.4
```

- In this example we check value counts before imputation and after imputation.
- Since variable is categorical will fill the missing value with mode.

```
dataset['Embarked'] = dataset['Embarked'].fillna('S')
```

```
dataset['Embarked'].value_counts()
```

[9]:

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

```
dataset['Embarked'].value_counts()
```

[11]:

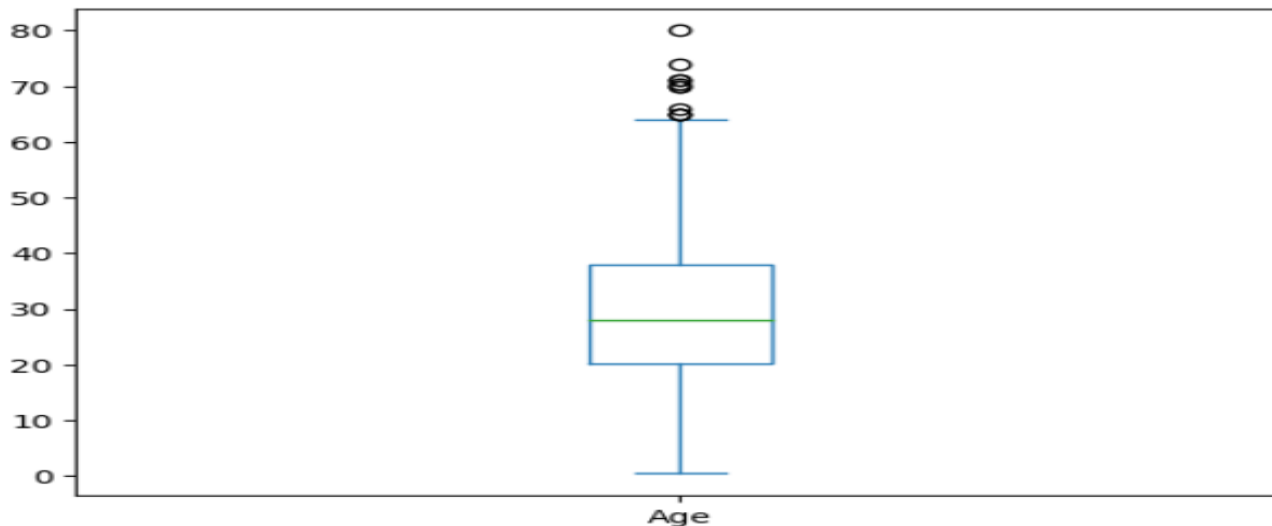
```
S      646  
C      168  
Q       77  
Name: Embarked, dtype: int64
```

- If variable is numerical than impute with 'Median' or 'Mean'.
 - When there is no outliers go with mean.
 - When there is outliers go with median.
 - **Example:** Since age is numerical variable and it carries outliers, so we will impute with median.

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

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

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



Handling Data types

- Handling data types correctly is crucial in data preprocessing.
- Because the type of data determines the operations that can be performed on it and the transformations it might require.
- Different machine learning algorithms and models have specific requirements for input data, and ensuring that the data types are appropriate.

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.

```
[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.

```
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
```

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

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

• Datetime Data:

- Handling: Convert datetime columns to datetime type and extract relevant features such as year, month, day, etc.

```
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:**

- Convert columns to appropriate types using `astype` or specific pandas functions.

```
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'])
```

Encoding

- Encoding categorical variables is a critical step in data preprocessing for machine learning models, as most models require numerical input.
- **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.

```
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.

```
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

- **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.

```
# One Hot Encoding
```

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

```
# Label encoder
```

```
dataset['Sex'] = dataset['Sex'].astype('category')
```

```
dataset['Sex'] = dataset['Sex'].cat.codes
```

```
# dummy variable
```

```
dataset = dataset.iloc[:, :-1]
```

```
[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

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

```
[63]: 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

```
df['num_categorical'] = 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**.

Handling Duplicates

- Find the duplicates :

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

```
[125]: 483
```

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

- Drop the duplicates:

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

```
[132]: df.shape
```

```
[132]: (10357, 13)
```

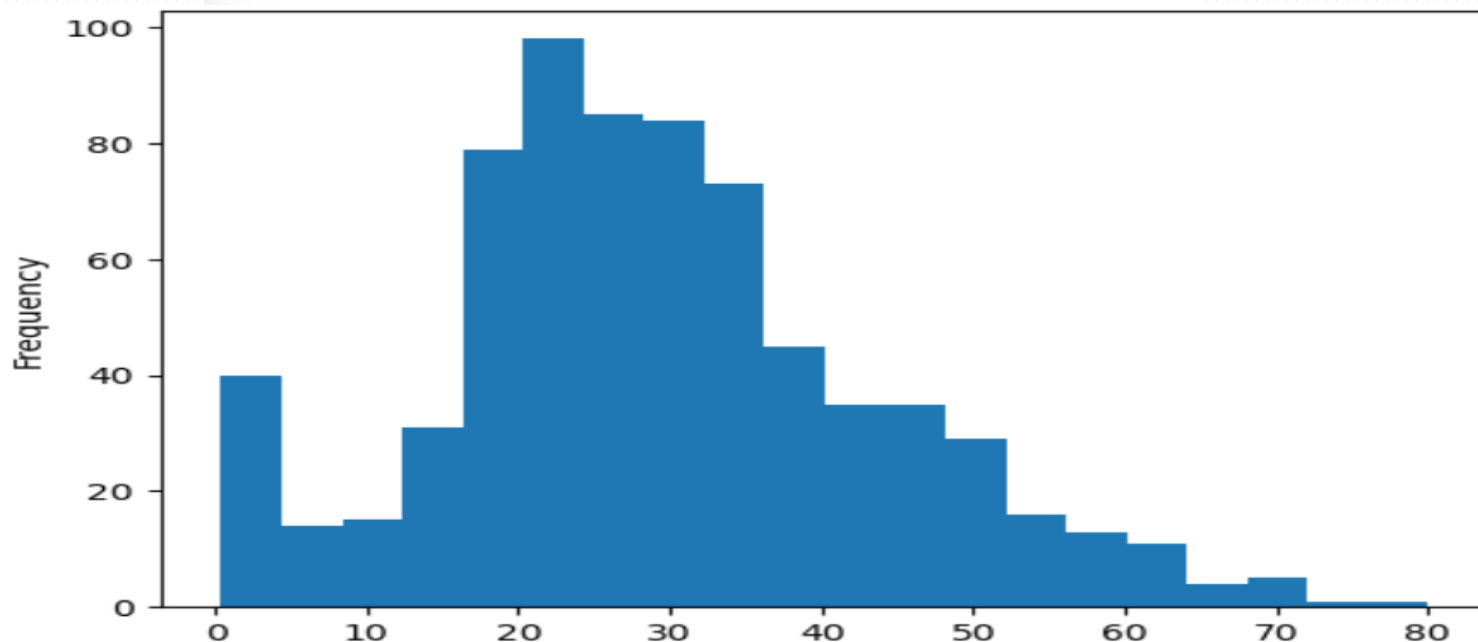
```
[134]: df.duplicated().sum()
```

```
[134]: 0
```

VISUALIZATION PLOTS

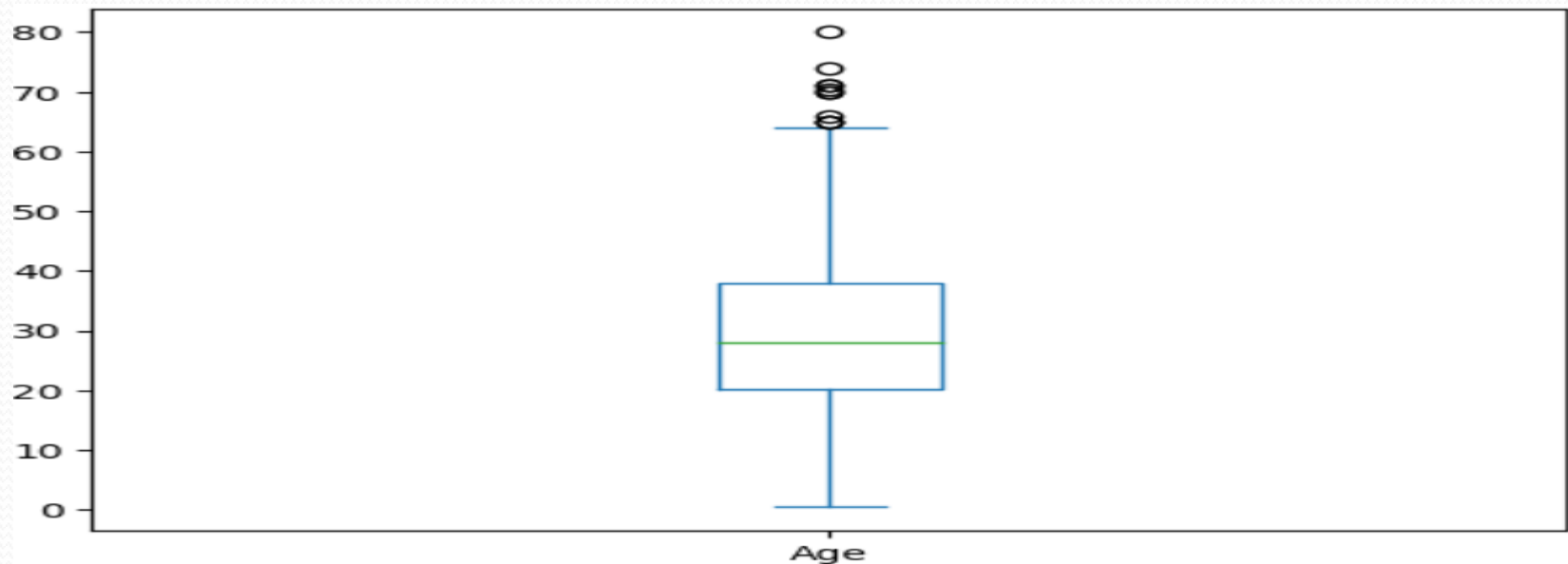
- **Histograms:** Histograms are useful for visualizing the distribution of a single numerical variable. They help identify the central tendency, spread, and skewness of the data.

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



- **Box plots (Box-and-Whisker plots):** Box plots provide a graphical summary of the distribution of numerical data through quartiles.
- They are particularly useful for detecting outliers and comparing distributions across different groups.

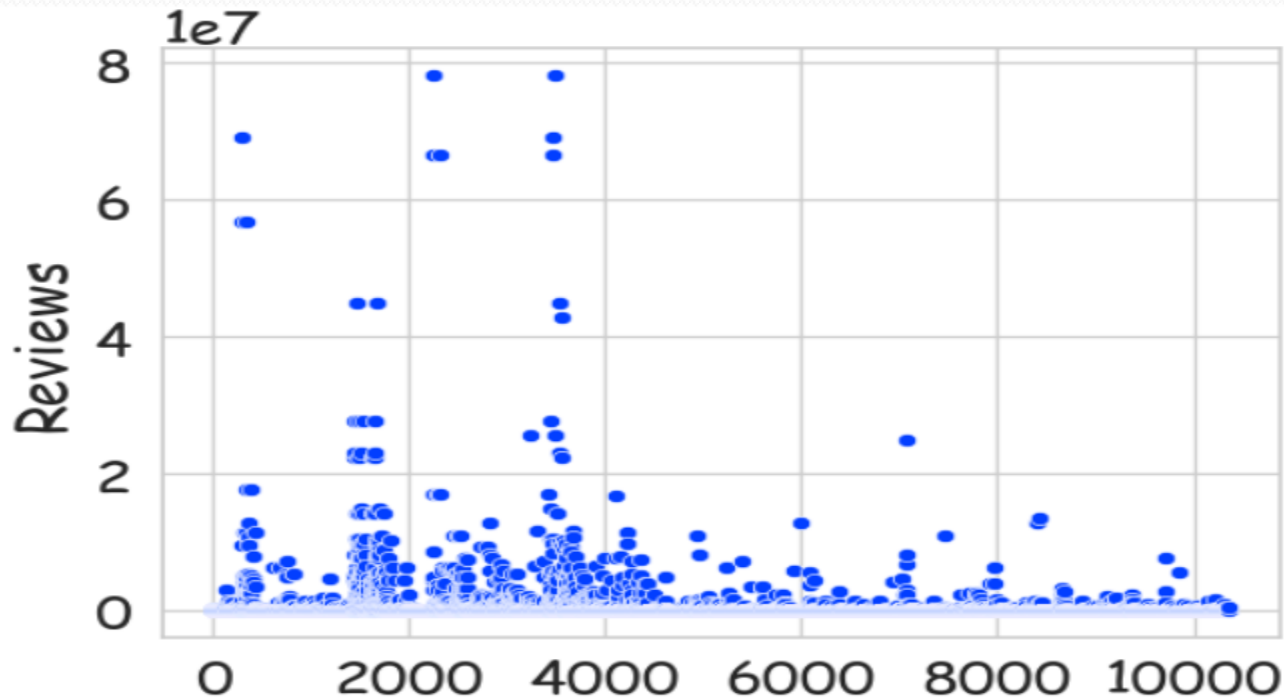
```
dataset['Age'].plot(kind='box')
```



- **Scatterplots:** Scatter plots are used to visualize the relationship between two numerical variables. They help identify patterns such as linear relationships, clusters, or correlations.

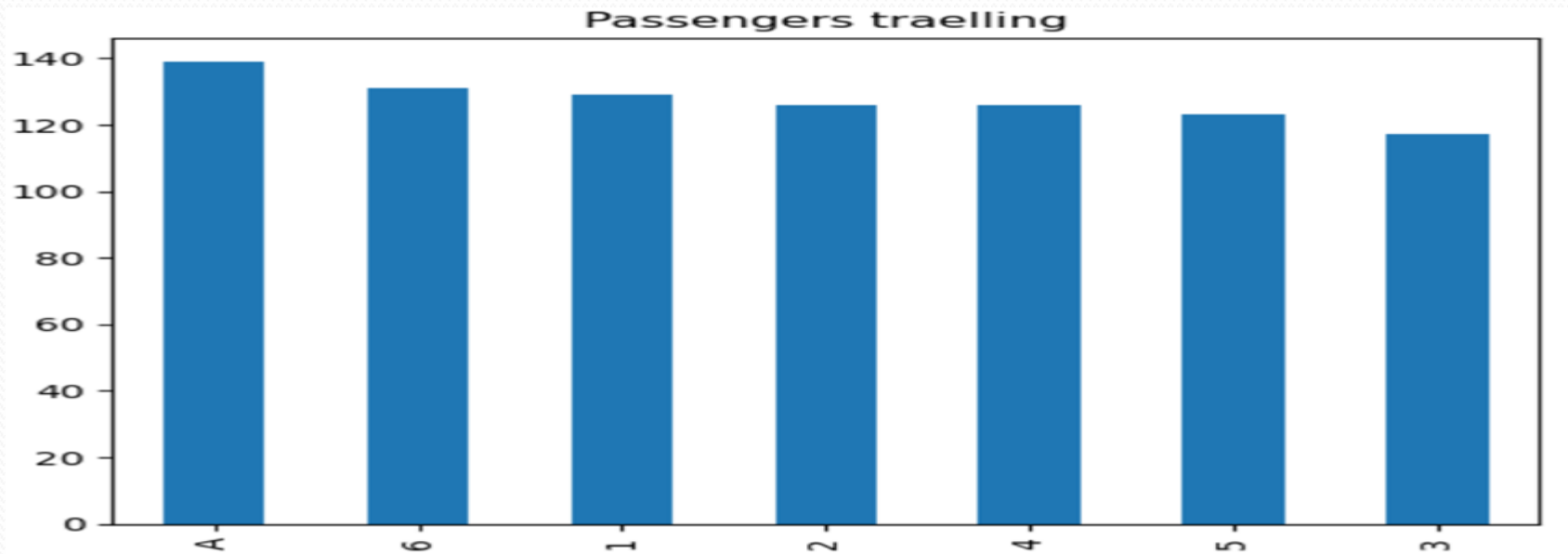
```
[142]: sns.set_theme(style='whitegrid',palette='bright',font='cursive',font_scale=1.8)
```

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



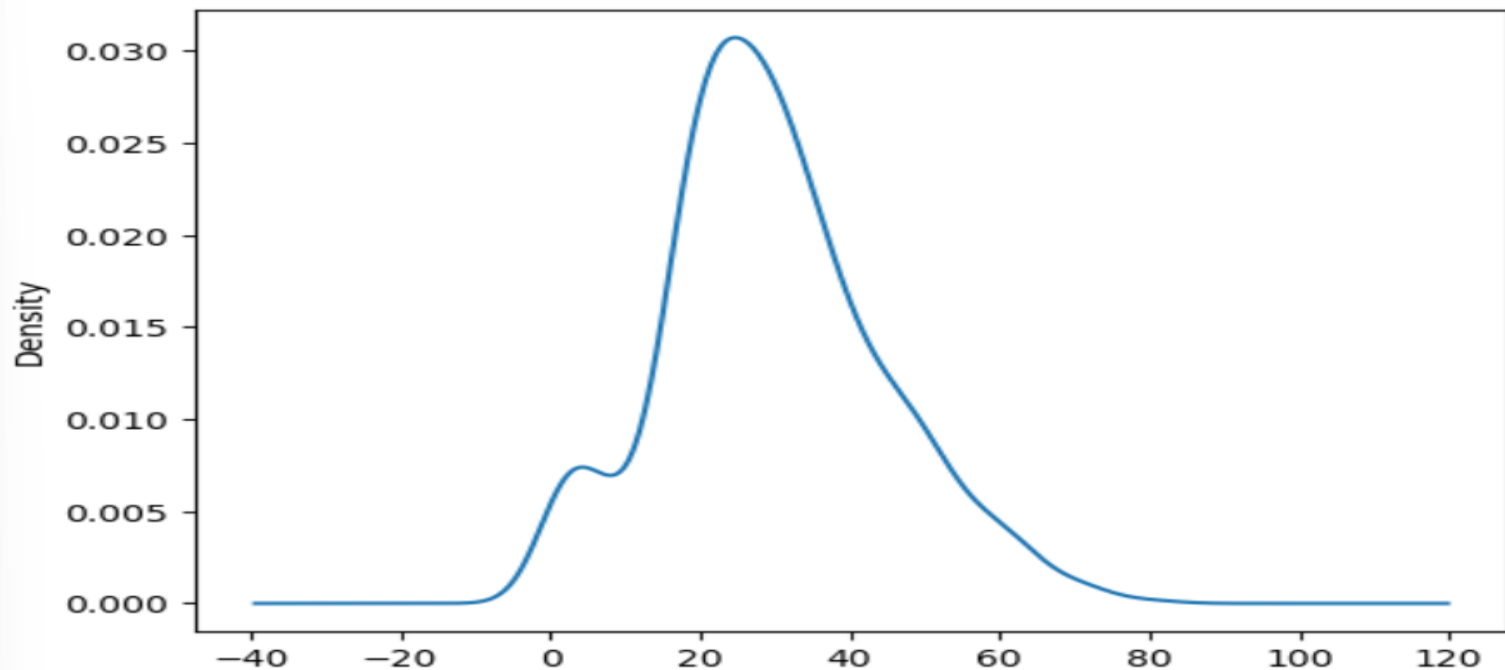
- **Bar plots:** Bar plots are effective for visualizing the distribution of categorical variables or comparing the frequencies of different categories.

```
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')
```

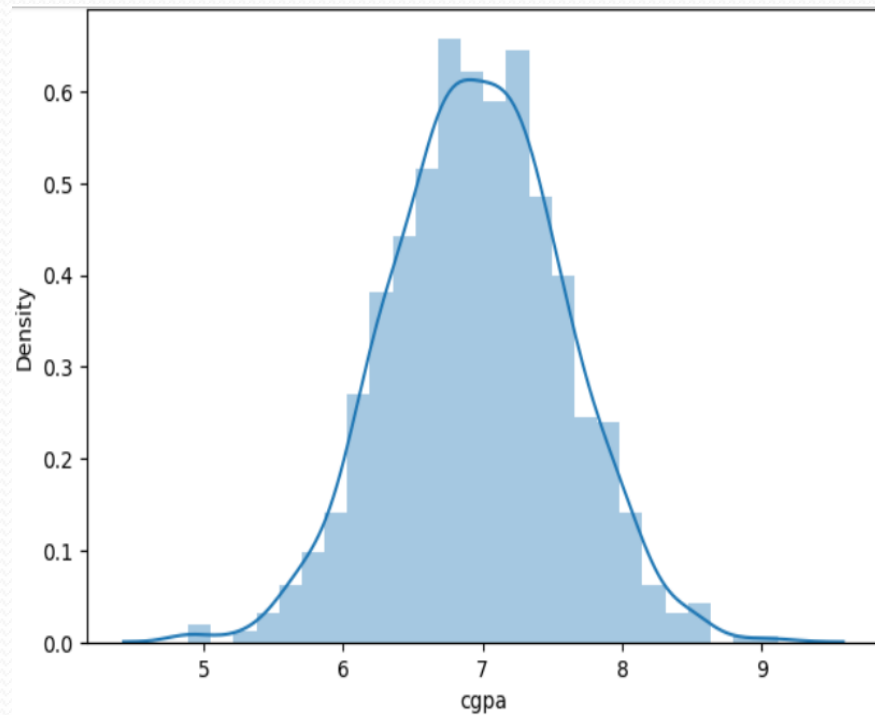


- A **distribution plot** is a visualization that combines aspects of a histogram and a kernel density plot to show the distribution of a continuous variable.
- It is useful for understanding the distribution of data points in a dataset and identifying patterns such as skewness, kurtosis, and the presence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

```
plt.figure(figsize=(16,5))
plt.subplot(1,2,1)
sns.distplot(df['cgpa'])

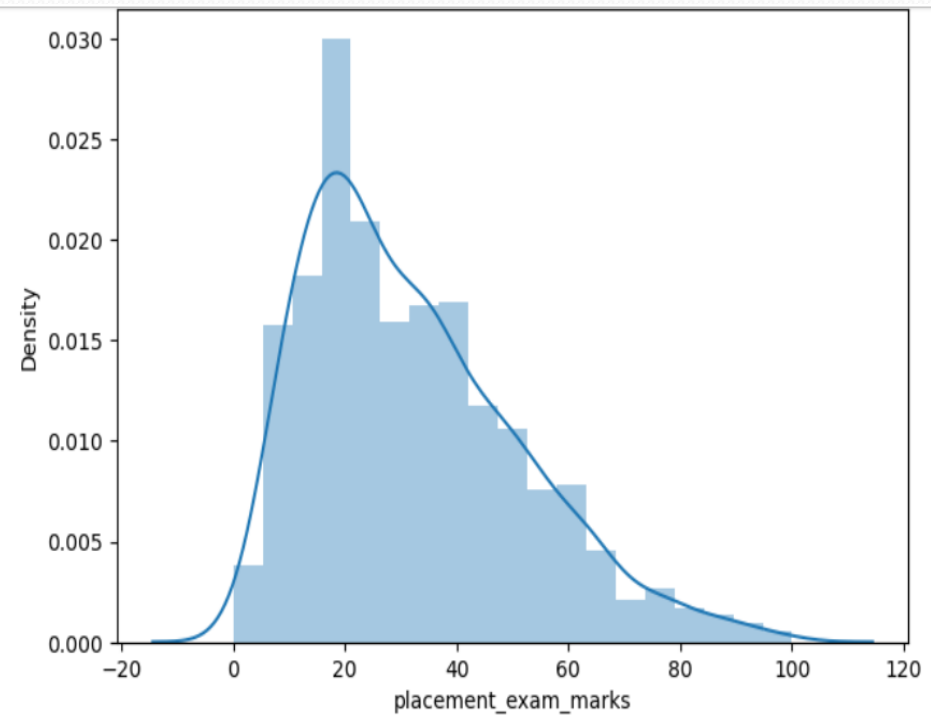
plt.subplot(1,2,2)
sns.distplot(df['placement_exam_marks'])

plt.show()
```



```
df['cgpa'].skew()
```

```
-0.014529938929314918
```



```
df['placement_exam_marks'].skew()
```

```
0.8356419499466834
```

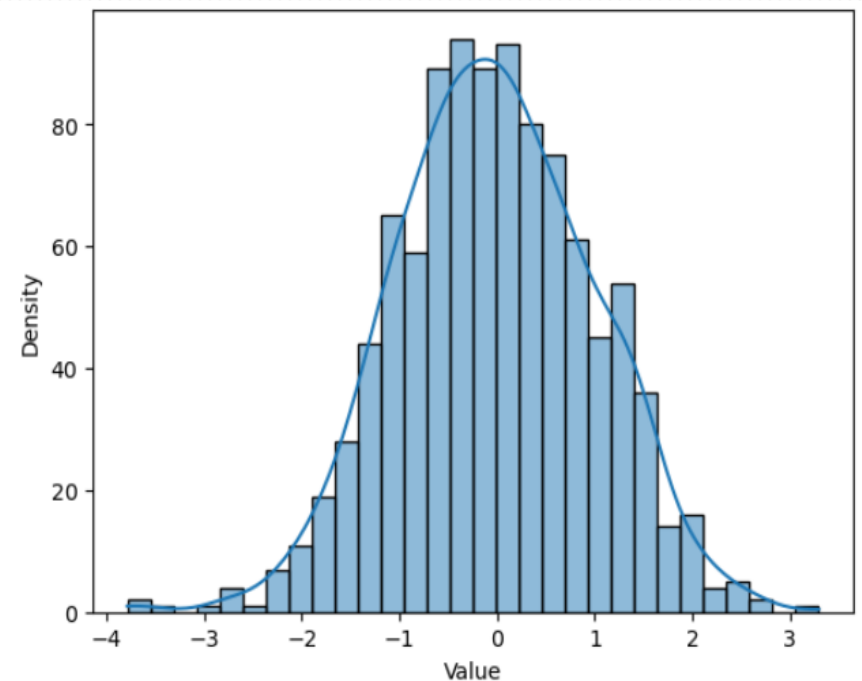
- In recent versions of Seaborn (0.11.0 and later), `sns.distplot` has been deprecated.
- Instead, you should use `sns.histplot` or `sns.kdeplot` for similar functionality. Here's how to create a similar plot using `sns.histplot`

```
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')
```



- **Outlier Treatment:**
 - Using 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 6.961240000000001
```

```
std value of cgpa 0.6158978751323894
```

```
min value of cgpa 4.89
```

```
max value of cgpa 9.12
```

- **Step 2:** Find Z- score to identify an outlier

```
print("mean + 3*std", df['cgpa'].mean() + 3*df['cgpa'].std())  
print("mean - 3*std", df['cgpa'].mean() - 3*df['cgpa'].std())
```

```
mean + 3*std 8.808933625397177
```

```
mean - 3*std 5.113546374602842
```

- **Step 3:** Finding outliers

```
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

- **Step 4: Calculating 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

- **Step 5:** Calculating outliers using 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

- **Step 6:** To remove the outliers use capping method:

```
[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
```

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

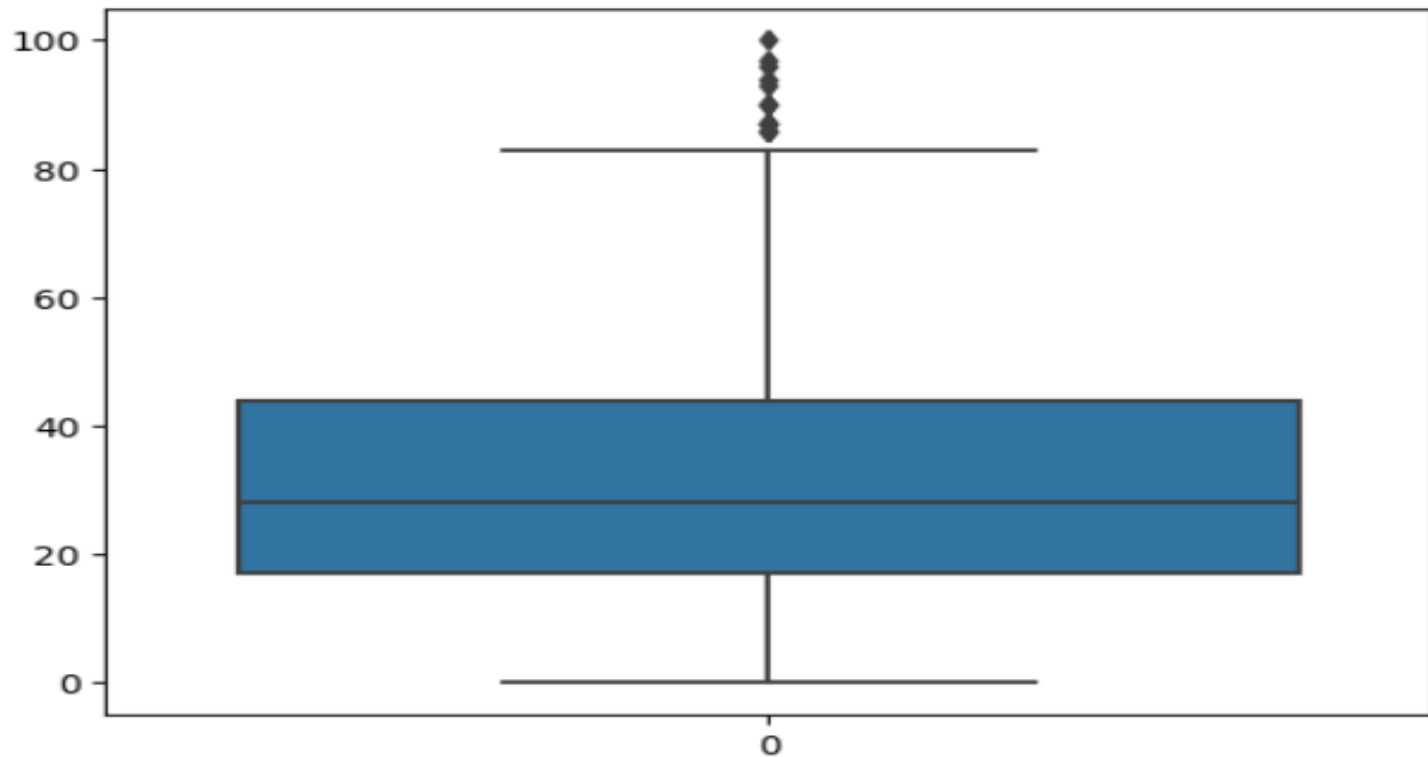
```
[20]: df['cgpa'].describe()
```

```
[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
```


- **Step 7:** To visualize the outliers use boxplot:

```
[22]: sns.boxplot(df['placement_exam_marks'])
```

```
[22]: <Axes: >
```



Step 8: IQR method to find the outliers :

```
[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
```

Step 8: IQR method to find the outliers :

```
[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
--	------	----------------------	--------	-------------

Step 9: Removing outlier using limits based IQR method by capping method:

```
[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]

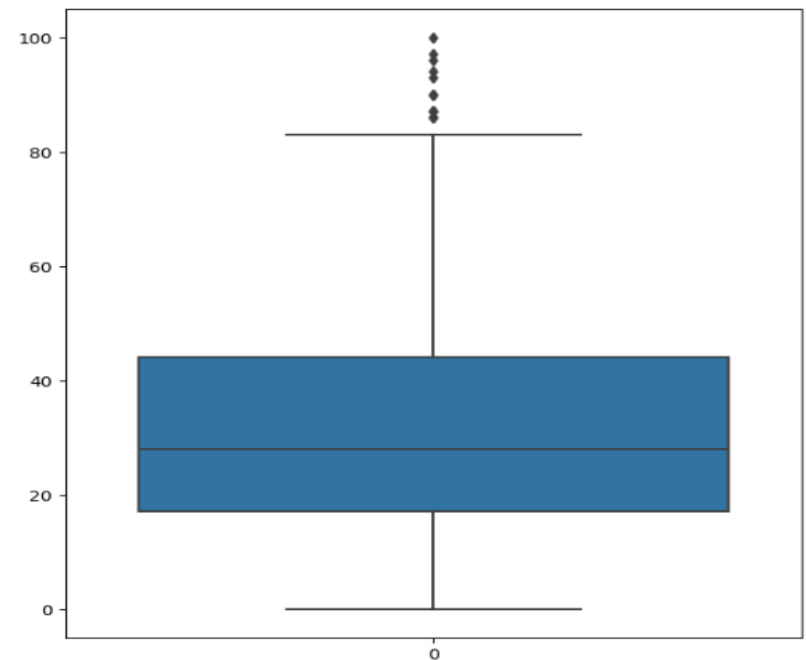
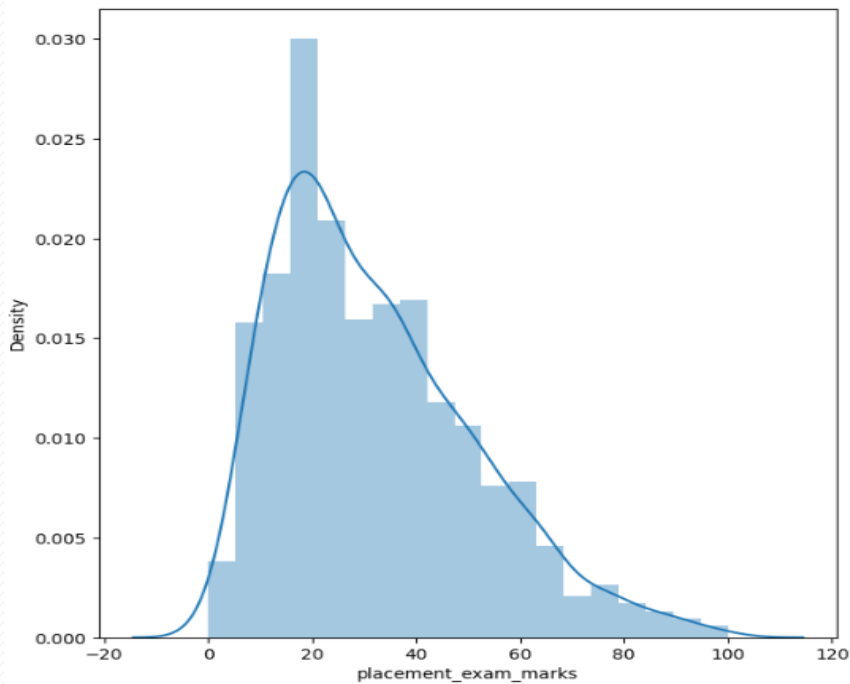
[34]:
```

cgpa	placement_exam_marks	placed	cgpa_zscore
------	----------------------	--------	-------------

Step 9: Visualization of outlier using distribution and box plot:

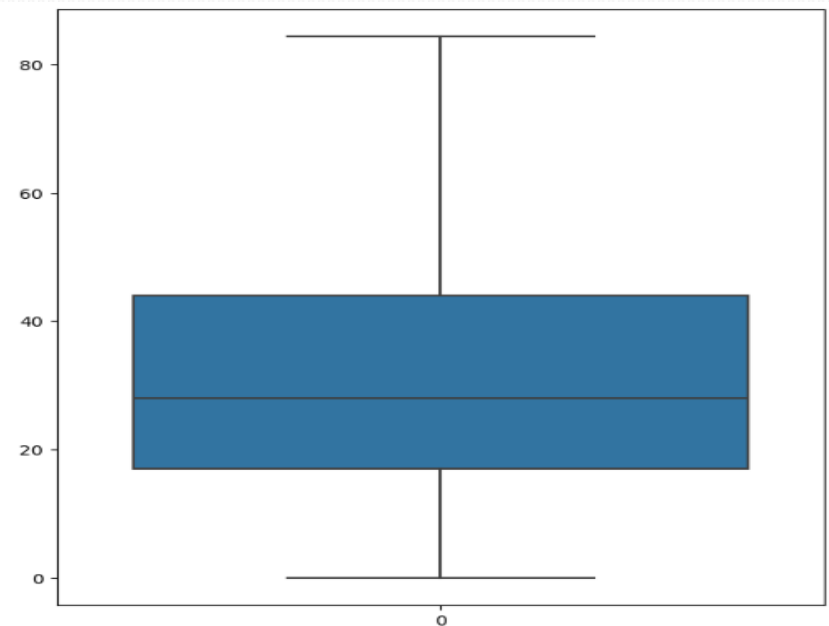
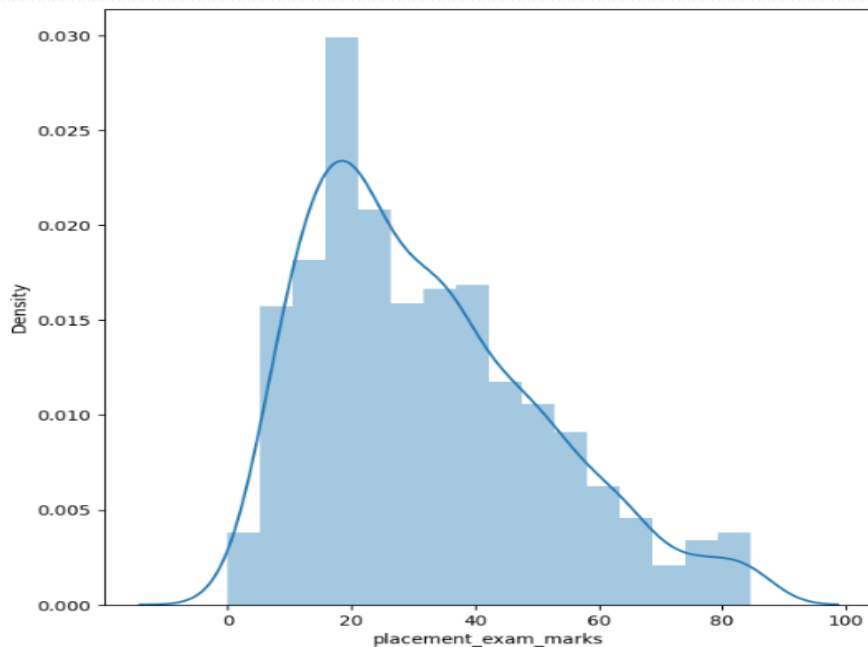
```
[35]: # comparing
plt.figure(figsize=(16,8))
plt.subplot(1,2,1)
sns.distplot(df['placement_exam_marks'])

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



Step 9: Visualization of outlier using 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()
```



Data balancing

- 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
```

Feature Scaling:

- 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']]
```

- Decide whether to use **StandardScaler** or **Normalizer**.

- 
- **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.
 - **Normalization:** Scaling features to a range between 0 and 1. Useful for algorithms like neural networks that benefit from bounded input values.

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.

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

- 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 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:**

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

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

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

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

- 
- **Examples:**
 - Text data represented as word counts or term frequencies, where cosine similarity is a common metric.
 - Algorithms k-nearest neighbors (KNN) that rely on distance metrics can benefit from data normalized to unit norm to ensure consistent scaling across all samples.
 - When dealing with sparse data (e.g., text data represented as TF-IDF), normalization helps to compare samples in a consistent manner without being affected by varying magnitudes.

Concept of `fit_transform` and `transform`:

When to Use

- **`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.

Exploratory Data Analysis (EDA):

- **Definition:** EDA is an approach to analyzing datasets to summarize their main characteristics, often using visual methods.
- **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.