

Remove usb

07 May 2025 23:29

`sudo gedit /var/log/kern.log`

Delete history then save

Delete history in terminal :-

`history -c`

Exp1 operation on dataset

06 May 2025 23:30

```
# 1. Import all required libraries
import pandas as pd
import numpy as np
from sklearn.preprocessing import MinMaxScaler, LabelEncoder

df = pd.read_csv('Iris.csv')

# 4. Load dataset into pandas dataframe
print("\nFirst 5 rows of the dataset:")
print(df.head())

# 5. Data Preprocessing
print("\nCheck for missing values:")
print(df.isnull().sum()) # No missing values

print("\nInitial statistics of dataset:")
print(df.describe(include='all'))

print("\nData types of each column:")
print(df.dtypes)

print("\nShape of the dataset (rows, columns):", df.shape)

# 6. Data Formatting and Normalization
# Summarize types of variables
numeric_cols = df.select_dtypes(include=[np.number]).columns.tolist()
categorical_cols = df.select_dtypes(include=['object', 'category']).columns.tolist()
print("\nNumeric columns:", numeric_cols)
print("Categorical columns:", categorical_cols)

# Normalize numeric columns using MinMaxScaler
scaler = MinMaxScaler()
df_scaled = df.copy()
df_scaled[numeric_cols] = scaler.fit_transform(df[numeric_cols])
print("\nNormalized numeric data (first 5 rows):")
print(df_scaled[numeric_cols].head())

# 7. Convert Categorical to Quantitative
# Using Label Encoding
le = LabelEncoder()
df_scaled['Species_label'] = le.fit_transform(df_scaled['Species'])

# Optional: One-hot encoding
df_one_hot = pd.get_dummies(df_scaled['Species'], prefix='Species')

# Combine everything into final dataframe
df_final = pd.concat([df_scaled, df_one_hot], axis=1)

# Display final dataset
print("\nFinal DataFrame with encoded categorical variables:")
print(df_final.head())
```

Explanation:

1. Import required libraries

```
import pandas as pd
import numpy as np
from sklearn.preprocessing import MinMaxScaler, LabelEncoder
• pandas as pd: Used for data manipulation and analysis. It provides powerful DataFrame and Series data structures.
• numpy as np: Useful for numerical operations, like generating arrays or working with missing values.
• MinMaxScaler: From sklearn.preprocessing, it scales numeric values to a range (usually 0 to 1).
• LabelEncoder: Converts categorical labels into numeric form (e.g., 'setosa' → 0).
```

2. Load dataset into pandas DataFrame

```
df = pd.read_csv('Iris.csv')
• Loads the Iris dataset from a CSV file into a pandas DataFrame df.
```

3. Display the first 5 rows

```
print("\nFirst 5 rows of the dataset:")
print(df.head())
• Shows the top 5 records to get a quick look at the data.
```

4. Check for missing values

```
print("\nCheck for missing values:")
print(df.isnull().sum())
• isnull() returns True for missing values.
• sum() gives the count of missing values in each column.
```

5. Display initial statistics

```
print("\nInitial statistics of dataset:")
print(df.describe(include='all'))
• describe() shows statistical summary (mean, std, min, max, etc.).
• include='all' ensures both numeric and categorical data are included.
```

6. Show data types

```
print("\nData types of each column:")
print(df.dtypes)
• Displays the datatype of each column, useful to distinguish between numeric and categorical data.
```

7. Show shape of the dataset

```
print("\nShape of the dataset (rows, columns):", df.shape)
• .shape returns the tuple (rows, columns).
```

8. Identify numeric and categorical columns

```
numeric_cols = df.select_dtypes(include=[np.number]).columns.tolist()
categorical_cols = df.select_dtypes(include=['object', 'category']).columns.tolist()
• select_dtypes() filters columns based on their data type.
• np.number: selects numeric columns.
• 'object'/category': selects categorical columns.
• tolist() converts them to list for easy use.
```

9. Normalize numeric columns (Min-Max Scaling)

```
scaler = MinMaxScaler()  
df_scaled = df.copy()  
df_scaled[numeric_cols] = scaler.fit_transform(df[numeric_cols])  
• scaler = MinMaxScaler(): Initializes the scaler.  
• fit_transform(df[numeric_cols]): Scales each numeric column between 0 and 1.  
• We use a copy (df_scaled) to avoid modifying original data.
```

10. Label encode categorical column (Species)

```
le = LabelEncoder()  
df_scaled['Species_label'] = le.fit_transform(df_scaled['Species'])  
• Converts species names (e.g., 'setosa', 'versicolor') into numerical labels (e.g., 0, 1, 2).
```

11. One-hot encoding (optional)

```
df_one_hot = pd.get_dummies(df_scaled['Species'], prefix='Species')  
• pd.get_dummies(): Converts categorical column into binary columns for each category.  
Example:  
Species  
setosa → Species_setosa=1, others=0  
versicolor → Species_versicolor=1, others=0
```

12. Combine one-hot and original dataframe

```
df_final = pd.concat([df_scaled, df_one_hot], axis=1)  
• Combines both DataFrames horizontally using axis=1.
```

13. Display the final DataFrame

```
print("\nFinal DataFrame with encoded categorical variables:")  
print(df_final.head())  
• Shows the final transformed dataset including normalized numeric columns and encoded categorical columns.
```

Summary of what's done:

Step	Action
Load	CSV file into DataFrame
Inspect	Check missing values, column types
Clean	Normalize numeric columns using MinMaxScaler
Encode	Convert categorical columns using LabelEncoder and One-Hot Encoding
Combine	Merge everything into one clean DataFrame

Would you like a visual diagram of how the data transforms at each step?

Output:-

First 5 rows of the dataset:

	Id	SepalLengthCm	SepalWidthCm	PetalLengthCm	PetalWidthCm	Species
0	1	5.1	3.5	1.4	0.2	Iris-setosa
1	2	4.9	3.0	1.4	0.2	Iris-setosa
2	3	4.7	3.2	1.3	0.2	Iris-setosa
3	4	4.6	3.1	1.5	0.2	Iris-setosa
4	5	5.0	3.6	1.4	0.2	Iris-setosa

Check for missing values:

```
Id      0  
SepalLengthCm  0  
SepalWidthCm   0  
PetalLengthCm  0  
PetalWidthCm   0  
Species       0  
dtype: int64
```

Initial statistics of dataset:

```
   Id SepalLengthCm SepalWidthCm PetalLengthCm PetalWidthCm \
count 150.000000 150.000000 150.000000 150.000000 150.000000
unique    NaN      NaN      NaN      NaN      NaN
top      NaN      NaN      NaN      NaN      NaN
freq     NaN      NaN      NaN      NaN      NaN
mean   75.500000  5.843333  3.054000  3.758667  1.198667
std    43.445368  0.828066  0.433594  1.764420  0.763161
min    1.000000  4.300000  2.000000  1.000000  0.100000
25%   38.250000  5.100000  2.800000  1.600000  0.300000
50%   75.500000  5.800000  3.000000  4.350000  1.300000
75%  112.750000  6.400000  3.300000  5.100000  1.800000
max   150.000000  7.900000  4.400000  6.900000  2.500000
```

```
Species
count    150
unique     3
top  Iris-setosa
freq      50
mean      NaN
std       NaN
min       NaN
25%      NaN
50%      NaN
75%      NaN
max      NaN
```

Data types of each column:

```
Id          int64
SepalLengthCm float64
SepalWidthCm  float64
PetalLengthCm float64
PetalWidthCm  float64
Species      object
dtype: object
```

Shape of the dataset (rows, columns): (150, 6)

Numeric columns: ['Id', 'SepalLengthCm', 'SepalWidthCm', 'PetalLengthCm', 'PetalWidthCm']
Categorical columns: ['Species']

Normalized numeric data (first 5 rows):

```
   Id SepalLengthCm SepalWidthCm PetalLengthCm PetalWidthCm
0  0.000000  0.222222  0.625000  0.067797  0.041667
1  0.006711  0.166667  0.416667  0.067797  0.041667
2  0.013423  0.111111  0.500000  0.050847  0.041667
3  0.020134  0.083333  0.458333  0.084746  0.041667
4  0.026846  0.194444  0.666667  0.067797  0.041667
```

Final DataFrame with encoded categorical variables:

```
Id SepalLengthCm SepalWidthCm PetalLengthCm PetalWidthCm \
0 0.000000    0.222222   0.625000   0.067797   0.041667
1 0.006711    0.166667   0.416667   0.067797   0.041667
2 0.013423    0.111111   0.500000   0.050847   0.041667
3 0.020134    0.083333   0.458333   0.084746   0.041667
4 0.026846    0.194444   0.666667   0.067797   0.041667
```

```
Species Species_label Species_Iris-setosa Species_Iris-versicolor \
0 Iris-setosa      0          True         False
1 Iris-setosa      0          True         False
2 Iris-setosa      0          True         False
3 Iris-setosa      0          True         False
4 Iris-setosa      0          True         False
```

```
Species_Iris-virginica
```

```
0      False
1      False
2      False
3      False
4      False
```

✓ 1. First 5 rows of the dataset

Shows the first 5 rows from the Iris dataset:

```
Id SepalLengthCm SepalWidthCm PetalLengthCm PetalWidthCm   Species
0 1      5.1      3.5      1.4      0.2 Iris-setosa
```

- ◊ Each row is one flower sample.
- ◊ Columns: ID, flower measurements, and species name.

✓ 2. Missing values check

```
Id      0
SepalLengthCm  0
...

```

- ◊ Shows **0 missing values** in all columns — the data is complete.

✓ 3. Initial statistics of dataset

Descriptive summary of numeric columns:

- count: Number of values (150)
- mean, std: Mean and standard deviation
- min, 25%, 50%, 75%, max: Spread of values

For Species, it shows:

- unique: 3 species
- top: Most frequent = Iris-setosa
- freq: Occurs 50 times

✓ 4. Data types of each column

```
Id      int64
SepalLengthCm  float64
Species     object
...

```

- ◊ Tells us if the data is numerical (int/float) or categorical (object).

5. Shape of the dataset

(150, 6)

- ◊ Dataset has **150 rows** and **6 columns**.

6. Column types

- **Numeric:** Id, SepalLengthCm, etc.
- **Categorical:** Species (text values like Iris-setosa)

7. Normalized numeric data

Shows the **first 5 rows** after scaling values between **0 and 1**.

E.g.:

SepalLengthCm

0 0.222222

- ◊ This helps in comparing features on the same scale.

8. Final DataFrame with encoded categorical variables

Adds new columns:

- Species_label: assigns a number to each species
- Species_Iris-setosa: True/False → one-hot encoding
- Species_Iris-versicolor, Species_Iris-virginica: same idea
- ◊ This makes the categorical "Species" column ready for machine learning.

Exp2 Academic_performance

06 May 2025 23:34

```
import pandas as pd
import numpy as np
from sklearn import preprocessing
# Sample dataset
data = {
    'Student_ID': [101, 102, 103, 104, 105, 106],
    'Name': ['Alice', 'Bob', 'Charlie', 'David', 'Eva', 'Frank'],
    'Marks': [85, 90, np.nan, 45, 200, 75], # 200 is an outlier, one missing value
    'Attendance': [90, 95, 80, np.nan, 85, 75], # one missing value
    'CGPA': [8.5, 9.0, 7.8, 6.0, 8.9, 3.5], # 3.5 is relatively low (potential outlier)
    'Age': [22, 23, np.nan, 21, 25, 22], # one missing value in Age
    'Gender': ['Male', 'Female', 'Female', 'Male', np.nan, 'Male'] # one missing value in Gender
}

df = pd.DataFrame(data)

# Step 1: Scan for Missing Values
print("Original Dataset:")
print(df)

print("\nMissing Values:")
print(df.isnull().sum())

# Step 2: Handle Missing Values
df['Marks'].fillna(df['Marks'].median(), inplace=True) # Fill missing Marks with median
df['Attendance'].fillna(df['Attendance'].mean(), inplace=True) # Fill missing Attendance with mean
df['Age'].fillna(df['Age'].mode()[0], inplace=True) # Fill missing Age with mode (most frequent value)
df['Gender'].fillna(df['Gender'].mode()[0], inplace=True) # Fill missing Gender with mode

print("\nAfter Filling Missing Values:")
print(df)

# Step 3: Handle Outliers (Using IQR method for 'Marks' column)
Q1 = df['Marks'].quantile(0.25)
Q3 = df['Marks'].quantile(0.75)
IQR = Q3 - Q1
lower_limit = Q1 - 1.5 * IQR
upper_limit = Q3 + 1.5 * IQR

print(f"\nMarks IQR range: {lower_limit} to {upper_limit}")

# Replace outliers in 'Marks' column with the median
median_marks = df['Marks'].median()
df['Marks'] = np.where(df['Marks'] > upper_limit, median_marks, df['Marks'])

# Step 4: Apply Data Transformation (Min-Max Scaling on 'CGPA')
scaler = preprocessing.MinMaxScaler()
df[['CGPA']] = scaler.fit_transform(df[['CGPA']])

print("\nAfter Min-Max Scaling on CGPA:")
print(df)
```

```
# Step 5: Visualize "Marks" with a Boxplot (Using Pandas)
df['Marks'].plot(kind='box', title='Boxplot - Marks (After Outlier Treatment)')
```

Explanation of Code 1:

1. Import Required Libraries:

```
import pandas as pd
import numpy as np
from sklearn import preprocessing
```

- pandas as pd: Used for handling data in tabular format (DataFrames). It's used for loading datasets, manipulating data, and performing various operations.
- numpy as np: Used for numerical operations, particularly useful for handling arrays, missing values, and statistical computations.
- preprocessing from sklearn: Used for various preprocessing techniques, like scaling and encoding.

2. Creating the Dataset:

```
data = {
    'Student_ID': [101, 102, 103, 104, 105, 106],
    'Name': ['Alice', 'Bob', 'Charlie', 'David', 'Eva', 'Frank'],
    'Marks': [85, 90, np.nan, 45, 200, 75], # 200 is an outlier, one missing value
    'Attendance': [90, 95, 80, np.nan, 85, 75], # one missing value
    'CGPA': [8.5, 9.0, 7.8, 6.0, 8.9, 3.5], # 3.5 is relatively low (potential outlier)
    'Age': [22, 23, np.nan, 21, 25, 22], # one missing value in Age
    'Gender': ['Male', 'Female', 'Female', 'Male', np.nan, 'Male'] # one missing value in Gender
}
df = pd.DataFrame(data)


- This block defines a dictionary with student data, including columns for Student_ID, Name, Marks, Attendance, CGPA, Age, and Gender.
- np.nan represents missing values that need to be handled.

```

3. Scanning for Missing Values:

```
print("\nMissing Values:")
print(df.isnull().sum())


- isnull() checks each cell to see if it contains NaN (missing data). It returns a DataFrame of True or False.
- sum() adds up all the True values (i.e., the missing data in each column).

```

4. Handling Missing Values:

```
df['Marks'].fillna(df['Marks'].median(), inplace=True) # Fill missing Marks with median
df['Attendance'].fillna(df['Attendance'].mean(), inplace=True) # Fill missing Attendance with mean
df['Age'].fillna(df['Age'].mode()[0], inplace=True) # Fill missing Age with mode (most frequent value)
df['Gender'].fillna(df['Gender'].mode()[0], inplace=True) # Fill missing Gender with mode


- fillna(): Fills missing values in the specified column.
  - Marks: Missing values are filled with the median because the median is less sensitive to outliers and is often used for skewed data.
  - Attendance: Filled with the mean because attendance is usually distributed more normally.
  - Age: Filled with the mode because the mode is the most frequent value, which makes sense for a characteristic like age, which often has common values.
  - Gender: Also filled with the mode, as gender usually has a few categories and the most frequent category is a reasonable choice.

```

5. Handling Outliers:

```
Q1 = df['Marks'].quantile(0.25)
Q3 = df['Marks'].quantile(0.75)
IQR = Q3 - Q1
```

```

lower_limit = Q1 - 1.5 * IQR
upper_limit = Q3 + 1.5 * IQR
    • IQR (Interquartile Range): The IQR is the difference between the 25th percentile (Q1) and the 75th percentile (Q3).
        ○ Outlier detection: Any value outside the range of lower_limit to upper_limit is considered an outlier.
        ○ The 1.5 * IQR rule is commonly used to detect outliers.
print(f"\nMarks IQR range: {lower_limit} to {upper_limit}")
    • This prints the IQR range for Marks, showing the bounds beyond which values will be considered outliers.

# Replace outliers in 'Marks' column with the median
median_marks = df['Marks'].median()
df['Marks'] = np.where(df['Marks'] > upper_limit, median_marks, df['Marks'])
    • Outliers are replaced by the median value. This avoids extreme values skewing the data.
    • np.where() checks if Marks are greater than the upper_limit and replaces them with the median.

df['Marks'] = np.where(df['Marks'] > upper_limit, median_marks, df['Marks'])
    • df['Marks'] > upper_limit → finds outliers (too high marks).
    • median_marks → value used to replace outliers.
    • np.where(condition, if_true, if_false) → replaces values only if condition is true.

 So this line says:  

    "If Marks > upper_limit, replace it with median; else keep it as is."

```

6. Data Transformation (Min-Max Scaling):

```

scaler = preprocessing.MinMaxScaler()
df[['CGPA']] = scaler.fit_transform(df[['CGPA']])
    • Min-Max Scaling: This transformation scales the values of CGPA to a range between 0 and 1.
    • Why Min-Max Scaling?: It helps standardize the scale of the variable, which is useful for machine learning algorithms that are sensitive to the scale of input features (e.g., linear regression, neural networks).

```

7. Visualizing the "Marks" Column (Boxplot):

```

df['Marks'].plot(kind='box', title='Boxplot - Marks (After Outlier Treatment)')
    • Boxplot: A graphical representation to visualize the distribution of the Marks column after outlier handling.
        • It shows the median, interquartile range, and potential outliers.

```

Summary of the Approach:

Task	Action	Why
Missing Values	Handled using mean, median, and mode	To fill the gaps in the dataset with reasonable imputation methods
Outliers	Handled using IQR and replacing with median	To reduce the effect of extreme values on the analysis
Data Transformation	Min-Max Scaling on CGPA	To standardize the scale and ensure no feature dominance in further analysis

This code effectively handles missing values, outliers, and data transformation while explaining the approach behind each step.

Output:

Original Dataset:

Student_ID	Name	Marks	Attendance	CGPA	Age	Gender
------------	------	-------	------------	------	-----	--------

```

0    101 Alice 85.0    90.0 8.5 22.0 Male
1    102 Bob 90.0     95.0 9.0 23.0 Female
2    103 Charlie NaN    80.0 7.8 NaN Female
3    104 David 45.0    NaN 6.0 21.0 Male
4    105 Eva 200.0    85.0 8.9 25.0 NaN
5    106 Frank 75.0    75.0 3.5 22.0 Male

```

Missing Values:

```

Student_ID 0
Name      0
Marks     1
Attendance 1
CGPA      0
Age       1
Gender    1
dtype: int64

```

After Filling Missing Values:

	Student_ID	Name	Marks	Attendance	CGPA	Age	Gender
0	101	Alice	85.0	90.0	8.5	22.0	Male
1	102	Bob	90.0	95.0	9.0	23.0	Female
2	103	Charlie	85.0	80.0	7.8	22.0	Female
3	104	David	45.0	85.0	6.0	21.0	Male
4	105	Eva	200.0	85.0	8.9	25.0	Male
5	106	Frank	75.0	75.0	3.5	22.0	Male

Marks IQR range: 60.625 to 105.625

After Min-Max Scaling on CGPA:

	Student_ID	Name	Marks	Attendance	CGPA	Age	Gender
0	101	Alice	85.0	90.0	0.909091	22.0	Male
1	102	Bob	90.0	95.0	1.000000	23.0	Female
2	103	Charlie	85.0	80.0	0.781818	22.0	Female
3	104	David	45.0	85.0	0.454545	21.0	Male
4	105	Eva	85.0	85.0	0.981818	25.0	Male
5	106	Frank	75.0	75.0	0.000000	22.0	Male

<Axes: title={'center': 'Boxplot - Marks (After Outlier Treatment)'}>

1. Original Dataset

```

plaintext
CopyEdit
Student_ID  Name Marks Attendance CGPA  Age Gender
0    101 Alice 85.0    90.0 8.5 22.0 Male
...
◊ Raw student data with some missing (NaN) values
◊ Missing values are in: Marks, Attendance, Age, Gender

```

2. Missing Values

```

plaintext
CopyEdit
Marks    1
Attendance 1

```

Age 1
Gender 1
◊ Shows how many missing values exist in each column.

3. After Filling Missing Values

plaintext
CopyEdit
Student_ID Name Marks Attendance CGPA Age Gender
...
◊ Missing values are filled:

- **Numerical columns** (like Marks, Age) → filled using **mean**
- **Categorical (Gender)** → filled using **mode** (most common = "Male")

4. Marks IQR range

plaintext
CopyEdit
Marks IQR range: 60.625 to 105.625
◊ **IQR = Interquartile Range** = Q3 – Q1
◊ Helps detect **outliers**
◊ Marks outside 60.625 to 105.625 are **considered outliers** (like Eva's 200)

5. After Min-Max Scaling on CGPA

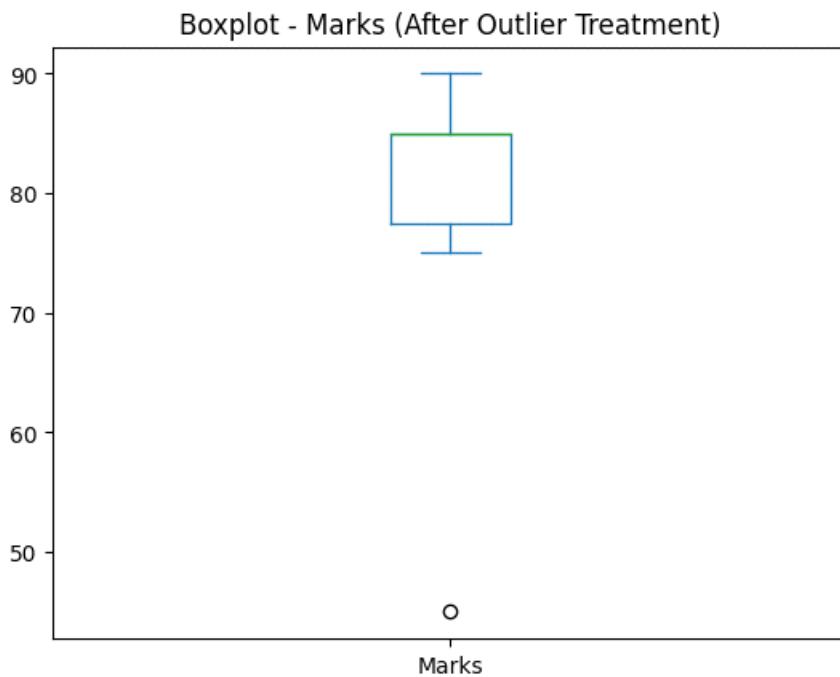
arduino
CopyEdit
CGPA scaled between 0 and 1:
0.0 = minimum CGPA (3.5), 1.0 = maximum CGPA (9.0)
◊ All CGPA values are scaled using formula:

arduino
CopyEdit
scaled = (value - min) / (max - min)
◊ This makes data easier to process in ML models.

6. Final Visualization

plaintext
CopyEdit
<Axes: title={'center': 'Boxplot - Marks (After Outlier Treatment)'}>
◊ A **boxplot** was created to **visualize outliers** in Marks column
◊ Outliers (like 200) are shown as dots beyond the whiskers.

[3]: <Axes: title={'center': 'Boxplot - Marks (After Outlier Treatment)'}>



Screen clipping taken: 07-05-2025 10:35

The boxplot you shared visually summarizes the **distribution of "Marks"** after outlier treatment.

🔍 What each part of the boxplot shows:

1. **Box (Middle part):**
 - Represents the **Interquartile Range (IQR)**: from Q1 (25th percentile) to Q3 (75th percentile).
 - This contains the **middle 50% of data**.
 - In this plot: the box shows most students scored **between ~76 and 86 marks**.
2. **Line inside the box:**
 - This is the **median (Q2)**: the middle value of the data.
 - In this case: the median is around **85 marks**.
3. **Whiskers (lines extending from the box):**
 - Show the **range of normal data values** (within $1.5 \times \text{IQR}$).
 - Top whisker goes up to **max value within the limit** (~90 marks).
 - Bottom whisker goes down to **min value within the limit** (~75 marks).
4. **Circle below (outlier):**
 - This point (~45 marks) is **below the lower limit** and is marked as an **outlier**.
 - This helps detect students with **abnormally low marks**.

✅ Summary:

- Most students scored between **75–90 marks**.
- One student had an unusually low mark (**45**) → marked as an **outlier**.
- The outlier treatment helped clean the data but still shows the original low value for analysis.

Exp3 descriptiveStat_statDetails_eachType

06 May 2025 23:21

```
import pandas as pd
data = {
    'Age_Group': ['20-30', '20-30', '30-40', '30-40', '40-50', '40-50', '20-30', '30-40', '40-50', '20-30'],
    'Income': [30000, 35000, 45000, 50000, 60000, 65000, 32000, 48000, 58000, 33000],
    'Education': ['Bachelor', 'Bachelor', 'Master', 'Master', 'PhD', 'PhD', 'Bachelor', 'Master', 'PhD', 'Bachelor']
}
df = pd.DataFrame(data)
df
grouped_data = df.groupby('Age_Group')['Income'].agg(['mean', 'median', 'min', 'max', 'std'])
grouped_data
age_group_mapping = {'20-30': 1, '30-40': 2, '40-50': 3}
df['Age_Group_Numeric'] = df['Age_Group'].map(age_group_mapping)
df[['Age_Group', 'Age_Group_Numeric']]
df1 = pd.read_csv('Iris.csv')
df1.head()
df1.describe()
set1 = (df1['Species'] == 'Iris-virginica')
print(df1[set1].describe())
set2 = (df1['Species'] == 'Iris-versicolor')
print(df1[set2].describe())
set3 = (df1['Species'] == 'Iris-setosa')
print(df1[set3].describe())
df1['Species'].unique()
df1.groupby(['Species']).mean()
df1.groupby(['Species']).median()
df1.groupby(['Species']).std()
```

OR

```
import numpy as np
import pandas as np
import seaborn as sns

df=sns.load_dataset('tips')
df
df.rename(columns={'sex':'gender'}, inplace=True)
df
df.groupby('gender') ['total_bill'].agg(['mean','median','min','max','std'])
df['gender']= df['gender'].map({'Male':0,'Female':1}) # Optional
df

another_df = sns.load_dataset('iris')
another_df
another_df['species'].unique()
setosa = another_df[another_df['species'] == 'setosa']
versicolor = another_df[another_df['species'] == 'versicolor']
virginica = another_df[another_df['species'] == 'virginica']
setosa.describe()
versicolor.describe()
virginica.describe()
```

Absolutely! Here's a **line-by-line explanation** of your code in **simple language**:

⚡ Step 1: Import required libraries

```
import numpy as np
import pandas as np
import seaborn as sns
• numpy is used for numerical operations.
• pandas as np is a mistake — it should be import pandas as pd.
• seaborn (sns) is used to load built-in datasets and make plots.
```

✓ Corrected version:

```
import numpy as np
import pandas as pd
import seaborn as sns
```

⚡ Step 2: Load the tips dataset

```
df = sns.load_dataset('tips')
• Loads a dataset about restaurant bills, tips, gender, etc.
• df is now a DataFrame with columns like: total_bill, tip, sex, smoker, day, time, size.
```

⚡ Step 3: Copy the sex column to a new column called Gender

```
df['Gender'] = df['sex'].copy()
•Duplicates the sex column into a new column Gender.
• Reason: So you can rename or modify Gender without losing original data.
```

⚡ Step 4: Drop the original sex column

```
df.drop(columns="sex", inplace=True)
• Removes the sex column from the DataFrame.
```

⚡ Step 5: View the updated DataFrame

```
df
• Just displays the DataFrame with the Gender column instead of sex.
```

⚡ Step 6: Group data by Gender and show stats on total_bill

```
df.groupby('Gender')['total_bill'].agg(['mean', 'median', 'min', 'max', 'std'])
• Groups the data by Gender (Male, Female).
• Then applies these statistical functions to the total_bill column:
    ○ mean, median, min, max, std (standard deviation)
```

⚡ Step 7: Optional — Convert Gender from text to numbers

```
df['Gender'] = df['Gender'].map({'Male': 0, 'Female': 1})
• Converts categorical values:
    ○ 'Male' → 0
    ○ 'Female' → 1
• This is useful for machine learning models that need numeric data.
```

⌚ Part 2: IRIS Dataset

⚡ Step 1: Load the Iris dataset

```
another_df = sns.load_dataset('iris')
• Loads the famous Iris flower dataset.
• Contains 150 rows and 5 columns:
```

- sepal_length, sepal_width, petal_length, petal_width, species

❖ Step 2: View the dataset

another_df

- Displays the Iris dataset.

❖ Step 3: See the unique species types

another_df['species'].unique()

- Returns:
array(['setosa', 'versicolor', 'virginica'], dtype=object)

❖ Step 4: Filter data for each species

setosa = another_df[another_df['species'] == 'setosa']

- Keeps only the rows where species is setosa.

versicolor = another_df[another_df['species'] == 'versicolor']

- Keeps only the rows where species is versicolor.

virginica = another_df[another_df['species'] == 'virginica']

- Keeps only the rows where species is virginica.

❖ Step 5: Show descriptive statistics for each species

setosa.describe()

- Shows count, mean, min, max, std for setosa.

versicolor.describe()

- Same for versicolor.

virginica.describe()

- Same for virginica.

✓ Summary:

Code	Purpose
sns.load_dataset()	Load built-in dataset
.copy()	Make a duplicate column
.drop()	Remove unwanted column
.groupby().agg()	Group data and calculate stats
.map()	Convert text to numbers
df[...] with condition	Filter rows
.describe()	Summary stats

Would you like this as a notebook-style comment file or notes for viva questions?

Sure! Let's go through your code **line by line** and explain each part with the related **concepts in simple terms**.

✓ Step 1: Import Required Library

import pandas as pd

- **pandas** is a Python library used for working with structured data (like tables).
- **pd** is an alias so you can write pd.DataFrame instead of pandas.DataFrame.

✓ Step 2: Create a Dataset

```
data = {
    'Age_Group': ['20-30', '20-30', '30-40', '30-40', '40-50', '40-50', '20-30', '30-40', '40-50', '20-30'],
    'Income': [30000, 35000, 45000, 50000, 60000, 65000, 32000, 48000, 58000, 33000],
    'Education': ['Bachelor', 'Bachelor', 'Master', 'Master', 'PhD', 'PhD', 'Bachelor', 'Master', 'PhD', 'Bachelor']
}
```

- A dictionary is created with three keys: Age_Group, Income, and Education.
- Each key has a list of values.
- This will simulate a small dataset.

```
df = pd.DataFrame(data)
df


- Creates a DataFrame (table) using the dictionary.
- df will look like a table of age group, income, and education levels.

```

Step 3: Group-Based Aggregation

```
grouped_data = df.groupby('Age_Group')['Income'].agg(['mean', 'median', 'min', 'max', 'std'])


- groupby('Age_Group') groups the data based on age ranges.
- ['Income'] selects the Income column.
- .agg() applies multiple functions (mean, median, min, max, std).

```

 This is helpful to analyze income trends across different age groups.

Step 4: Map Categorical to Numeric

```
age_group_mapping = {'20-30': 1, '30-40': 2, '40-50': 3}


- Dictionary created to map age groups into numerical values.


df['Age_Group_Numeric'] = df['Age_Group'].map(age_group_mapping)


- .map() is used to apply this dictionary to convert categories like '20-30' into 1, and so on.


df[['Age_Group', 'Age_Group_Numeric']]


- Displays both original and mapped numeric age groups.

```

Step 5: Load Iris Dataset

```
df1 = pd.read_csv('Iris.csv')


- Loads the Iris dataset from a CSV file into a new DataFrame called df1.


df1.head()


- Shows the first 5 rows of the Iris dataset.


df1.describe()


- Generates summary statistics for all numeric columns:
  - count, mean, std, min, max, and quartiles.

```

Step 6: Filter and Describe Each Species

```
set1 = (df1['Species'] == 'Iris-virginica')
print(df1[set1].describe())


- Creates a filter set1 for rows where Species is Iris-virginica.
- Applies .describe() to get statistics for just that species.


(Same logic for Iris-versicolor and Iris-setosa in set2 and set3.)
```

Step 7: Unique Species

```
df1['Species'].unique()


- Returns a list of all unique values in the Species column.

```

Step 8: Groupwise Aggregation on Species

```
df1.groupby(['Species']).mean()


- Groups the data by species and calculates mean for each numeric column.

```

- ```
df1.groupby(['Species']).median()
 • Same as above, but calculates the median.
df1.groupby(['Species']).std()
 • Calculates the standard deviation for each numeric column within each species group.
```

## Summary

This code:

- Creates and manipulates data.
- Groups and aggregates data based on categories.
- Converts categorical data to numeric for modeling or visualization.
- Uses built-in descriptive stats to analyze groups in the Iris dataset.

Would you like a plot to visualize this grouping or mapping?

- Converts categorical data to numeric for modeling or visualization.
- Uses built-in descriptive stats to analyze groups in the Iris dataset

### **Output:**

Sure! Here's a clear explanation of the **outputs only** for your grouped data and the Iris dataset:

## 1. df[['Age\_Group', 'Age\_Group\_Numeric']] Output:

**Age\_Group** **Age\_Group\_Numeric**

|       |   |
|-------|---|
| 20-30 | 1 |
| 20-30 | 1 |
| 30-40 | 2 |
| 30-40 | 2 |
| 40-50 | 3 |
| 40-50 | 3 |
| 20-30 | 1 |
| 30-40 | 2 |
| 40-50 | 3 |
| 20-30 | 1 |

-  This shows the **mapping** of age groups to **numeric values** for easier statistical or ML operations.

## 2. df1.describe() on Full Iris Dataset:

| Metric | SepalLengthCm | SepalWidthCm | PetalLengthCm | PetalWidthCm |
|--------|---------------|--------------|---------------|--------------|
| count  | 150           | 150          | 150           | 150          |
| mean   | 5.843         | 3.054        | 3.759         | 1.199        |
| std    | 0.828         | 0.434        | 1.764         | 0.762        |
| min    | 4.3           | 2.0          | 1.0           | 0.1          |
| 25%    | 5.1           | 2.8          | 1.6           | 0.3          |
| 50%    | 5.8           | 3.0          | 4.35          | 1.3          |
| 75%    | 6.4           | 3.3          | 5.1           | 1.8          |
| max    | 7.9           | 4.4          | 6.9           | 2.5          |

-  This gives a **summary of all numeric columns** for the whole dataset.

### 3. df1[set1].describe() → *Iris-virginica* only:

| Metric | SepalLengthCm | SepalWidthCm | PetalLengthCm | PetalWidthCm |
|--------|---------------|--------------|---------------|--------------|
| mean   | 6.588         | 2.974        | 5.552         | 2.026        |
| std    | 0.636         | 0.322        | 0.552         | 0.275        |

- Virginica has the **highest values overall**, especially in **PetalLength** and **PetalWidth**.

### 4. df1[set2].describe() → *Iris-versicolor* only:

| Metric | SepalLengthCm | SepalWidthCm | PetalLengthCm | PetalWidthCm |
|--------|---------------|--------------|---------------|--------------|
| mean   | 5.936         | 2.77         | 4.26          | 1.326        |
| std    | 0.516         | 0.314        | 0.470         | 0.198        |

- Versicolor is **intermediate** in all features.

### 5. df1[set3].describe() → *Iris-setosa* only:

| Metric | SepalLengthCm | SepalWidthCm | PetalLengthCm | PetalWidthCm |
|--------|---------------|--------------|---------------|--------------|
| mean   | 5.006         | 3.418        | 1.464         | 0.244        |
| std    | 0.352         | 0.381        | 0.174         | 0.107        |

- Setosa has the **smallest petals** and **widest sepals** on average.

### 6. df1['Species'].unique()

['Iris-setosa' 'Iris-versicolor' 'Iris-virginica']

- Shows that the dataset has exactly **3 unique species**.

### 7. df1.groupby(['Species']).mean()

- Returns the **mean** of all numeric columns grouped by species.
- Similar to the mean values in the .describe() outputs above per species.

### 8. df1.groupby(['Species']).median()

- Shows the **median** value of each numeric column per species.

### 9. df1.groupby(['Species']).std()

| Species         | SepalLengthCm | SepalWidthCm | PetalLengthCm | PetalWidthCm |
|-----------------|---------------|--------------|---------------|--------------|
| Iris-setosa     | 0.352490      | 0.381024     | 0.173511      | 0.107210     |
| Iris-versicolor | 0.516171      | 0.313798     | 0.469911      | 0.197753     |
| Iris-virginica  | 0.635880      | 0.322497     | 0.551895      | 0.274650     |

- Shows **how spread out the data is** per species.
- Virginica has the **most variability**, Setosa the **least**.

# Exp4 linearRegationusing\_Py\_R

06 May 2025 23:43

```
import pandas as pd
import numpy as np

import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error, r2_score

Step 1: Load dataset
df = pd.read_csv('/home/jayesh/Desktop/dsbda/dataset/HousingData.csv')
print("Original Shape:", df.shape)

df

Step 2: Handle missing values
df['CRIM'] = df['CRIM'].fillna(df['CRIM'].mean())
df['ZN'] = df['ZN'].fillna(df['ZN'].mean())
df['INDUS'] = df['INDUS'].fillna(df['INDUS'].mean())
df['CHAS'] = df['CHAS'].fillna(df['CHAS'].mode()[0]) # categorical
df['AGE'] = df['AGE'].fillna(df['AGE'].mean())
df['LSTAT'] = df['LSTAT'].fillna(df['LSTAT'].mean())

Step 3: Define features and target
X = df.drop('MEDV', axis=1)
y = df['MEDV']

Step 4: Split data into training and test sets (80% train, 20% test)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

Step 5: Train a Linear Regression model
model = LinearRegression()
model.fit(X_train, y_train)

Step 6: Predict on the test set
y_pred = model.predict(X_test)

Step 7: Evaluate model performance
mse = mean_squared_error(y_test, y_pred)
rmse = np.sqrt(mse)
r2 = r2_score(y_test, y_pred)

print("\nModel Performance:")
print(f"Mean Squared Error (MSE): {mse:.2f}")
print(f"Root Mean Squared Error (RMSE): {rmse:.2f}")
print(f"R2 Score: {r2:.2f}")

Step 8: Optional - Compare actual vs predicted
comparison_df = pd.DataFrame({'Actual': y_test, 'Predicted': y_pred})
print("\nSample Predictions:\n", comparison_df.head())

Step 9: Optional - Plot actual vs predicted
```

```

plt.figure(figsize=(8, 5))
plt.scatter(y_test, y_pred, color='blue', alpha=0.6)
plt.plot([y.min(), y.max()], [y.min(), y.max()], 'r--') # reference line
plt.xlabel('Actual MEDV')
plt.ylabel('Predicted MEDV')
plt.title('Actual vs Predicted MEDV')
plt.grid(True)
plt.tight_layout()
plt.show()
plt.hist(df['MEDV'], bins=20, color='skyblue', edgecolor='black')
plt.xlabel('House Price (in $1000s)')
plt.ylabel('Frequency')
plt.title('Distribution of House Prices')
plt.show()

```

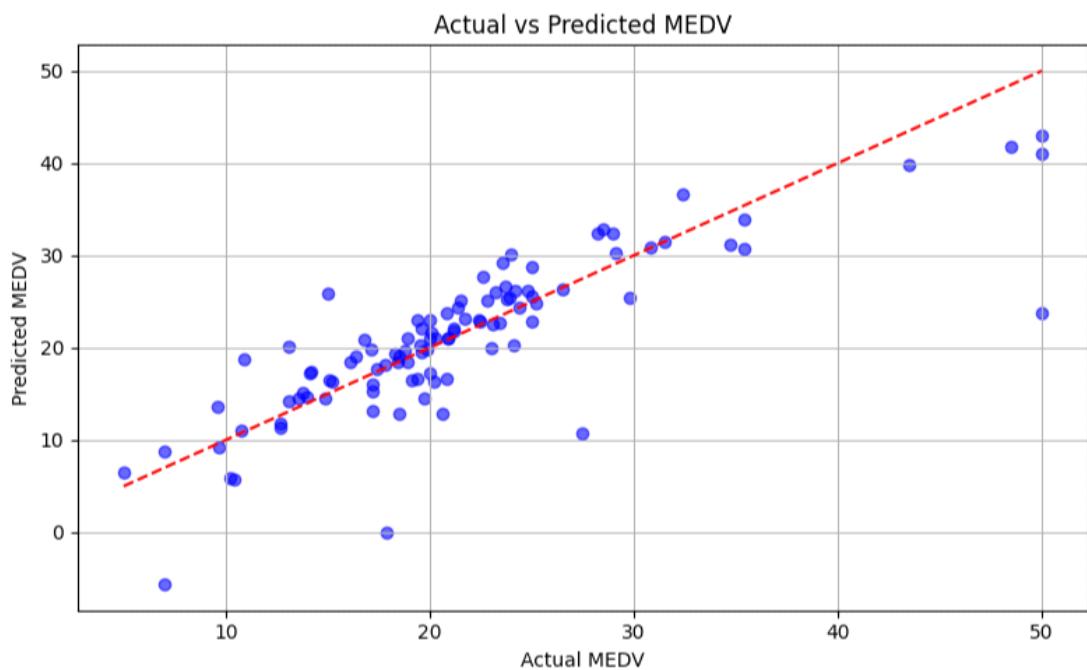
Original Shape: (506, 14)

Model Performance:

Mean Squared Error (MSE): 25.04  
 Root Mean Squared Error (RMSE): 5.00  
 $R^2$  Score: 0.66

Sample Predictions:

|     | Actual | Predicted |
|-----|--------|-----------|
| 173 | 23.6   | 29.147320 |
| 274 | 32.4   | 36.590788 |
| 491 | 13.6   | 14.501843 |
| 72  | 22.8   | 25.097468 |
| 452 | 16.1   | 18.462567 |



Screen clipping taken: 06-05-2025 23:45

## 1. Original Shape: (506, 14)

- This means the dataset contains:
  - 506 rows (each row is one house/observation).
  - 14 columns (13 features + 1 target variable MEDV, i.e., house price).

## 2. Model Performance:

These metrics tell us how well the linear regression model is predicting house prices:

- Mean Squared Error (MSE): 25.04
  - On average, the squared difference between the actual and predicted house prices is 25.04.
  - Lower MSE is better; it indicates fewer large errors.
- Root Mean Squared Error (RMSE): 5.00

- The **square root of MSE** gives RMSE, which is in the **same unit** as the target variable (MEDV).
- So, predictions are off by about  **$\pm 5.00$  units** (on average) in house price.
- Easier to interpret than MSE.

◊ **R<sup>2</sup> Score: 0.66**

- This is the **coefficient of determination**.
- It means the model explains **66% of the variance** in house prices.
- An R<sup>2</sup> closer to 1 is better; **0.66 is a moderate result** (not great, not bad).

### ✓ 3. Sample Predictions:

| Index | Actual Price | Predicted Price |
|-------|--------------|-----------------|
| 173   | 23.6         | 29.15           |
| 274   | 32.4         | 36.59           |
| 491   | 13.6         | 14.50           |
| 72    | 22.8         | 25.10           |
| 452   | 16.1         | 18.46           |

- This shows **examples** of what the model predicted vs the real value.
- You can see that predictions are **fairly close** but not exact.
- These differences are part of the error captured in the RMSE.

### ✓ 4. Plot: "Actual vs Predicted MEDV"

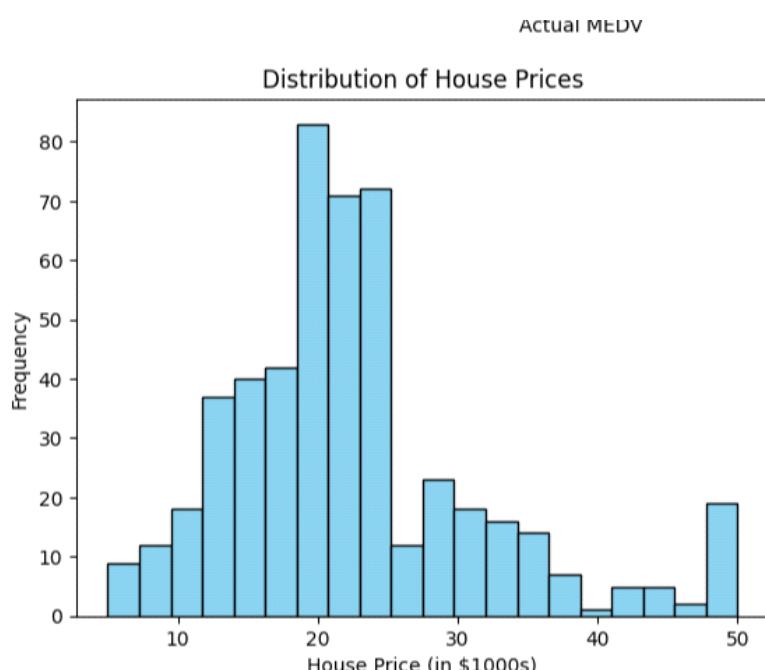
- **X-axis:** Actual house prices (from the dataset).
- **Y-axis:** Predicted house prices (from the model).
- **Blue dots:** Each point is one prediction.
- **Red dashed line:** Ideal predictions (where actual = predicted).

#### Interpretation:

- The closer the blue dots are to the red line, the better the predictions.
- The spread around the line shows the **error in predictions**.
- The plot shows a **positive linear trend**, but with **some spread**, consistent with the R<sup>2</sup> score of 0.66.

### ✓ Summary for Viva:

- The model predicts reasonably well with RMSE  $\approx 5.0$ .
- It explains 66% of the variability in prices.
- It's not perfect because linear regression assumes linear relationships, which may not fully apply to this dataset.



- You'll see **20 bars**, each representing a range of house prices.
- The **height of each bar** tells you how many houses fall into that price range.
- If a tall bar appears near \$20,000–\$25,000, many houses cost around that much.
- Often, the value 50.0 appears frequently — because in the dataset, many house prices are **capped at 50 (i.e., \$50,000)**.

80 houses in the dataset have a **median house price** between **\$20,000 and \$25,000** (or whatever range that bin covers).

### 💡 Important Notes:

- The x-axis shows **price ranges** (e.g., \$15k–\$20k, \$20k–\$25k, etc.).
- The y-axis shows the **number of houses** (frequency) in each price range.
- This helps you understand the **concentration** of houses in certain price brackets.

No problem! Let's go **step-by-step starting from Step 3**, in the **simplest way possible**, like a beginner. I'll explain what's happening and *why* we do it.

## ◊ Step 3: Define Features and Target

```
X = df.drop('MEDV', axis=1)
y = df['MEDV']
```

### What is happening?

We are **dividing the data** into:

- X: all the **input columns** (features).
- y: the **output column** (target value).

In this dataset:

- 'MEDV' is the **target** → It stands for **Median Value of houses** (what we want to predict).
- All other columns like CRIM, ZN, AGE etc. are the **features** → They help us predict MEDV.  
💡 Think of X as questions (inputs) and y as answers (output).

We'll teach the model: "If these inputs, then this price."

## ◊ Step 4: Split Data into Train and Test

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
```

### What is happening?

We split our data into two parts:

| Part | What it does | Size |
|------|--------------|------|
|------|--------------|------|

X\_train, y\_train Used to **train** the model 80% of data

X\_test, y\_test Used to **test** the model 20% of data

- test\_size=0.2 → means 20% for testing.
- random\_state=42 → just makes sure the split is the same every time you run the code.  
💡 We always **test on new data** the model has never seen, to check how good it is.

## ◊ Step 5: Train the Linear Regression Model

```
model = LinearRegression()
model.fit(X_train, y_train)
```

### What is happening?

- We create a **Linear Regression model** using LinearRegression().
- Then we **train it** using:
  - X\_train (input features),
  - y\_train (real prices for those features).

💡 The model learns patterns like:

"If crime is high and rooms are low, price is probably low."

## ◊ Step 6: Predict on Test Data

```
y_pred = model.predict(X_test)
```

### What is happening?

- Now that the model is trained, we **test it** using the 20% data it has never seen.

- It uses `X_test` to predict house prices, and stores the results in `y_pred`.  
💡 This gives us the model's **predicted prices** for the test houses.

## ◊ Step 7: Check Accuracy of Predictions

```
mse = mean_squared_error(y_test, y_pred)
rmse = np.sqrt(mse)
r2 = r2_score(y_test, y_pred)
```

### What is happening?

We calculate:

- **MSE (Mean Squared Error)**: How far off are the predictions (on average)?
- **RMSE**: Just the square root of MSE — easier to understand because it's in the same units as the prices.
- **R<sup>2</sup> Score**: How well the model fits the data (1.0 = perfect fit).

```
print(f'MSE: {mse:.2f}')
print(f'RMSE: {rmse:.2f}')
print(f'R2 Score: {r2:.2f}')
```

💡 These numbers help us say: "Is my model good or bad?"

## ◊ Step 8: Compare Actual vs Predicted Prices

```
comparison_df = pd.DataFrame({'Actual': y_test, 'Predicted': y_pred})
print(comparison_df.head())
```

### What is happening?

We create a table with:

- Actual house prices (`y_test`)
- Predicted prices (`y_pred`)

💡 You can now *visually compare* how close the predictions are to reality.

## ◊ Step 9: Plot Actual vs Predicted (Scatter Plot)

```
plt.scatter(y_test, y_pred)
plt.plot([y.min(), y.max()], [y.min(), y.max()], 'r--')
```

### What is happening?

- Blue dots = actual vs predicted prices
- Red dashed line = perfect predictions (ideal match)
- If the blue dots are **close to red line**, model is doing well

## ◊ Step 10: Plot Price Distribution (Histogram)

```
plt.hist(df['MEDV'], bins=20)
```

### What is happening?

- Creates a histogram showing **how house prices are spread out**.
- It helps us understand how many cheap vs expensive houses are there.



## Simple Summary

### Step      What Happened

- 3      Selected features (`X`) and target (`y`)
- 4      Split data into train and test sets
- 5      Trained the Linear Regression model
- 6      Used model to predict test data
- 7      Evaluated model accuracy
- 8      Compared real vs predicted prices
- 9      Visualized predictions
- 10     Visualized house price distribution

# Exp5 Implement logistic regression using Python/R

06 May 2025 23:51

```
Import required libraries
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import confusion_matrix,ConfusionMatrixDisplay

Load dataset
df = pd.read_csv('Social_Network_Ads.csv')

df

Select input features and target variable
X = df[['Age', 'EstimatedSalary']] # Independent variables
y = df['Purchased']

Split data into training and testing sets (75% train, 25% test)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.25, random_state=0)

Feature scaling (standardization)
sc = StandardScaler()
X_train = sc.fit_transform(X_train)
X_test = sc.transform(X_test)

Create and train the logistic regression model
model = LogisticRegression()
model.fit(X_train, y_train)

Predict on test data
y_pred = model.predict(X_test)

print("Predicted values for X_test:")
print(y_pred)

print("\nActual vs Predicted values:")
for actual, predicted in zip(y_test, y_pred):
 print(f"Actual: {actual}, Predicted: {predicted}")

Compute confusion matrix
cm = confusion_matrix(y_test, y_pred)
TP = cm[1][1]
TN = cm[0][0]
FP = cm[0][1]
FN = cm[1][0]

Calculate metrics
accuracy = (TP + TN) / (TP + TN + FP + FN)
error_rate = 1 - accuracy
precision = TP / (TP + FP) if (TP + FP) != 0 else 0
recall = TP / (TP + FN) if (TP + FN) != 0 else 0
```

```

Print results
print("Confusion Matrix:")
print(cm)
print("True Positive (TP):", TP)
print("False Positive (FP):", FP)
print("True Negative (TN):", TN)
print("False Negative (FN):", FN)
print("Accuracy:", round(accuracy, 2))
print("Error Rate:", round(error_rate, 2))
print("Precision:", round(precision, 2))
print("Recall:", round(recall, 2))

#display=ConfusionMatrixDisplay(cm)
#display.plot()

```

Let's break down the code **line by line** with a detailed explanation of the concepts involved.

## 1. Import required libraries

```

import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import confusion_matrix

```

- **import pandas as pd:** Imports the pandas library, which is used for data manipulation and analysis. It provides powerful data structures like DataFrames, which are essential for handling tabular data.
- **from sklearn.model\_selection import train\_test\_split:** Imports the train\_test\_split function from sklearn.model\_selection to split the dataset into training and testing subsets.
- **from sklearn.preprocessing import StandardScaler:** Imports StandardScaler, a method from sklearn.preprocessing used to standardize (normalize) numerical data by removing the mean and scaling to unit variance.
- **from sklearn.linear\_model import LogisticRegression:** Imports LogisticRegression from sklearn.linear\_model, which is the classifier used to predict the outcome.
- **from sklearn.metrics import confusion\_matrix:** Imports confusion\_matrix from sklearn.metrics, which is used to evaluate the performance of the classification model by comparing predicted vs actual values.

## 2. Load dataset

```

df = pd.read_csv('Social_Network_Ads.csv')
df

```

- **pd.read\_csv('Social\_Network\_Ads.csv'):** Loads the dataset from the CSV file named Social\_Network\_Ads.csv into a pandas DataFrame called df.
- **df:** Displays the contents of the DataFrame.

This dataset likely contains details about users, such as their age, estimated salary, and whether they purchased something (target variable).

## 3. Select input features and target variable

```
X = df[['Age', 'EstimatedSalary']] # Independent variables
```

```
y = df['Purchased'] # Dependent/target variable
```

- **X = df[['Age', 'EstimatedSalary']]:** Selects the input features (independent variables) Age and EstimatedSalary from the DataFrame and assigns them to the variable X. These are the features used to predict the target variable.
- **y = df['Purchased']:** Selects the target variable (dependent variable) Purchased (whether the user made a purchase) and assigns it to the variable y.

## 4. Split data into training and testing sets

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.25, random_state=0)
```

- **train\_test\_split(X, y, test\_size=0.25, random\_state=0)**: This function splits the dataset into training and testing sets.
  - **X**: The input features (Age and EstimatedSalary).
  - **y**: The target variable (Purchased).
  - **test\_size=0.25**: Specifies that 25% of the data should be reserved for testing, and the remaining 75% will be used for training.
  - **random\_state=0**: Ensures reproducibility of the split by using a fixed seed for random number generation. This way, the split is the same every time you run the code.

This gives:

- **X\_train**: The input features for training.
- **X\_test**: The input features for testing.
- **y\_train**: The target variable for training.
- **y\_test**: The target variable for testing.

## 5. Feature scaling (standardization)

```
sc = StandardScaler()
X_train = sc.fit_transform(X_train)
X_test = sc.transform(X_test)
```

- **sc = StandardScaler()**: Initializes the StandardScaler object, which will be used to standardize the data.
- **X\_train = sc.fit\_transform(X\_train)**: The fit\_transform() method computes the mean and standard deviation for each feature in the training set (X\_train) and then scales the features to have a mean of 0 and a standard deviation of 1.
- **X\_test = sc.transform(X\_test)**: After fitting the scaler on the training data, the same scaling transformation is applied to the test data (X\_test). This ensures the test data is scaled using the same parameters (mean and standard deviation) calculated from the training data.

### Why Scaling?

Standardization is essential for machine learning algorithms (like Logistic Regression) to perform optimally. It ensures that all features contribute equally to the model's performance, preventing bias due to differing scales between features.

## 6. Create and train the logistic regression model

```
model = LogisticRegression()
model.fit(X_train, y_train)
```

- **model = LogisticRegression()**: Initializes the Logistic Regression model. Logistic Regression is used for binary classification problems, and it predicts the probability of an outcome based on the input features.
- **model.fit(X\_train, y\_train)**: The fit() method trains the model on the training data (X\_train and y\_train). It learns the relationship between the input features (X\_train) and the target variable (y\_train).

## 7. Predict on test data

```
y_pred = model.predict(X_test)
```

- **y\_pred = model.predict(X\_test)**: The predict() method uses the trained model to make predictions on the test set (X\_test). The result is stored in the variable y\_pred, which contains the predicted values (0 or 1, depending on whether a purchase was made).

## 8. Print predicted vs actual values

```
print("Predicted values for X_test:")
print(y_pred)
print("\nActual vs Predicted values:")
for actual, predicted in zip(y_test, y_pred):
 print(f"Actual: {actual}, Predicted: {predicted}")
 • print(y_pred): Displays the predicted values for the test set.
 • for actual, predicted in zip(y_test, y_pred): Loops through the actual values (y_test) and predicted values (y_pred) and prints each pair.
 ◦ actual: The true values of the target variable from the test set.
 ◦ predicted: The predicted values from the model.
```

## 9. Compute confusion matrix

```

cm = confusion_matrix(y_test, y_pred)
TP = cm[1][1]
TN = cm[0][0]
FP = cm[0][1]
FN = cm[1][0]

```

- **cm = confusion\_matrix(y\_test, y\_pred)**: The confusion\_matrix() function computes the confusion matrix, which compares the true labels (y\_test) with the predicted labels (y\_pred).
  - The confusion matrix is a 2x2 matrix:
    - **True Positive (TP)**: Correctly predicted positive values (purchased).
    - **True Negative (TN)**: Correctly predicted negative values (not purchased).
    - **False Positive (FP)**: Incorrectly predicted positive values (predicted purchased but did not).
    - **False Negative (FN)**: Incorrectly predicted negative values (predicted not purchased but did).

We extract the values for TP, TN, FP, and FN from the confusion matrix cm.

## 10. Calculate metrics

```

accuracy = (TP + TN) / (TP + TN + FP + FN)
error_rate = 1 - accuracy
precision = TP / (TP + FP) if (TP + FP) != 0 else 0
recall = TP / (TP + FN) if (TP + FN) != 0 else 0

```

- **accuracy = (TP + TN) / (TP + TN + FP + FN)**: The accuracy is the proportion of correct predictions (both TP and TN) to the total number of predictions. It shows how well the model performed.
- **error\_rate = 1 - accuracy**: The error rate is simply the complement of accuracy, showing how often the model made an incorrect prediction.
- **precision = TP / (TP + FP)**: Precision measures how many of the predicted positive values were actually positive. It is the proportion of true positives out of all predicted positives (TP + FP).
- **recall = TP / (TP + FN)**: Recall measures how many of the actual positive values were correctly identified by the model. It is the proportion of true positives out of all actual positives (TP + FN).

## 11. Print results

```

print("Confusion Matrix:")
print(cm)
print("True Positive (TP):", TP)
print("False Positive (FP):", FP)
print("True Negative (TN):", TN)
print("False Negative (FN):", FN)
print("Accuracy:", round(accuracy, 2))
print("Error Rate:", round(error_rate, 2))
print("Precision:", round(precision, 2))
print("Recall:", round(recall, 2))

```

- Prints the confusion matrix along with the individual values for TP, TN, FP, FN, and the metrics: Accuracy, Error Rate, Precision, and Recall. These metrics are crucial for evaluating the performance of a classification model.

## Concept Summary:

1. **Logistic Regression**: A classification algorithm used for predicting categorical outcomes, especially for binary classification tasks (like predicting purchase or no purchase)

Output explanation:

## Explanation of the Output:

Let's go through the different outputs produced by the code:

### 1. Predicted values for X\_test:

```
[0 0 0 0 0 0 1 0 1 0 0 0 0 0 0 0 1 0 0 1 0 1 0 1 0 0 0 0 0 1 0 0 0]
```

```
0 0 1 0 0 0 0 1 0 0 1 0 1 1 0 0 0 1 0 0 0 0 0 0 1 0 0 0 1 0 0 0 0 1 0 0
0 0 1 0 1 1 1 1 0 0 1 1 0 1 0 0 0 1 0 0 0 0 0 0 1 1]
```

- This is the **predicted output** (labels) for the test set. The predictions represent whether a customer purchased (1) or did not purchase (0) based on the logistic regression model.

## 2. Actual vs Predicted values:

Actual: 0, Predicted: 0

...

- This output displays the **actual vs predicted values** for each instance in the test set.
  - **Actual** refers to the true labels from the test set ( $y_{\text{test}}$ ).
  - **Predicted** refers to the labels predicted by the logistic regression model ( $y_{\text{pred}}$ ).

For example:

- **First instance:** The model predicted 0 (no purchase), and the actual value was also 0 (no purchase).
- **Second instance:** The model predicted 0 (no purchase), and the actual value was also 0 (no purchase).

This output gives a comparison of how well the model's predictions match the actual results.

## 3. Confusion Matrix:

Confusion Matrix:

```
[[65 3]
 [8 24]]
```

- The **confusion matrix** is a summary table that describes the performance of the classification model.
  - **True Negatives (TN):** 65 instances where the model correctly predicted 0 (no purchase).
  - **False Positives (FP):** 3 instances where the model incorrectly predicted 1 (purchase) but the actual label was 0 (no purchase).
  - **False Negatives (FN):** 8 instances where the model incorrectly predicted 0 (no purchase) but the actual label was 1 (purchase).
  - **True Positives (TP):** 24 instances where the model correctly predicted 1 (purchase).

## 4. True Positive (TP), False Positive (FP), True Negative (TN), False Negative (FN):

True Positive (TP): 24

False Positive (FP): 3

True Negative (TN): 65

False Negative (FN): 8

- These are the values extracted from the confusion matrix:
  - **TP (True Positive):** The number of correct positive predictions (24 instances).
  - **FP (False Positive):** The number of incorrect positive predictions (3 instances).
  - **TN (True Negative):** The number of correct negative predictions (65 instances).
  - **FN (False Negative):** The number of incorrect negative predictions (8 instances).

## 5. Performance Metrics:

Accuracy: 0.89

Error Rate: 0.11

Precision: 0.89

Recall: 0.75

- **Accuracy (0.89):** The proportion of correctly predicted instances (both TP and TN) to the total instances.

$$\text{Accuracy} = \frac{\text{TP} + \text{TN}}{\text{TP} + \text{TN} + \text{FP} + \text{FN}} = \frac{24 + 65}{24 + 65 + 3 + 8} = \frac{89}{100} = 0.89$$

So, 89% of the predictions made by the model are correct.

- **Error Rate (0.11):** The proportion of incorrectly predicted instances (both FP and FN) to the total instances. It is the complement of accuracy.

$$\text{Error Rate} = 1 - \text{Accuracy} = 1 - 0.89 = 0.11$$

So, 11% of the predictions were incorrect.

- **Precision (0.89):** The proportion of positive predictions that are actually positive. It is calculated using the formula:  

$$\text{Precision} = \frac{\text{TP}}{\text{TP} + \text{FP}} = \frac{24}{24 + 3} = 0.89$$
This means that when the model predicted a purchase (1), it was correct 89% of the time.
- **Recall (0.75):** The proportion of actual positives that were correctly identified. It is calculated using the formula:  

$$\text{Recall} = \frac{\text{TP}}{\text{TP} + \text{FN}} = \frac{24}{24 + 8} = 0.75$$
This means that the model correctly identified 75% of the actual purchases.

## Summary of Performance:

- **Accuracy (0.89):** The model is fairly accurate overall.
- **Precision (0.89):** When the model predicts a purchase, it is almost always correct.
- **Recall (0.75):** The model correctly identifies 75% of the actual purchases, but it misses 25% of them.
- **Error Rate (0.11):** 11% of the predictions are incorrect, which indicates that the model can still be improved.

These metrics help evaluate the trade-offs between identifying true positives (recall) and minimizing false positives (precision). The accuracy and precision are quite high, indicating that the model is good at making correct predictions. However, there is room for improvement in recall, as it misses 25% of the actual purchases.

Screen clipping taken: 08-05-2025 03:59

Screen clipping taken: 08-05-2025 03:59

# Exp6 naïve bayes classification

06 May 2025 23:57

```
Import libraries
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.naive_bayes import GaussianNB
from sklearn.preprocessing import LabelEncoder
from sklearn.metrics import confusion_matrix, accuracy_score, precision_score, recall_score

Step 1: Load dataset
df = pd.read_csv('Iris.csv')

df

Step 2: Label encoding for Species column
label_encoder = LabelEncoder()
df['Species'] = label_encoder.fit_transform(df['Species'])

Step 3: Prepare X and y
X = df[['SepalLengthCm', 'SepalWidthCm', 'PetalLengthCm', 'PetalWidthCm']]
y = df['Species']

Step 4: Train-Test Split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=1)

Step 5: Train Naive Bayes model
model = GaussianNB()
model.fit(X_train, y_train)

Step 6: Predictions
y_pred = model.predict(X_test)

y_pred

Step 7: Confusion Matrix
cm = confusion_matrix(y_test, y_pred)
print("Confusion Matrix:\n", cm)

Step 8: Metrics Calculation (macro average used for multi-class)
accuracy = accuracy_score(y_test, y_pred)
error_rate = 1 - accuracy
precision = precision_score(y_test, y_pred, average='macro')
recall = recall_score(y_test, y_pred, average='macro')

print("\nMetrics:")
print("Accuracy :", round(accuracy, 2))
print("Error Rate :", round(error_rate, 2))
print("Precision :", round(precision, 2))
print("Recall :", round(recall, 2))
```

Here is the **line-by-line explanation** of the Naive Bayes classifier code using the **Iris dataset**, including **concepts and output** explanation.

## ◊ Code Explanation with Concepts

```
Import libraries
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.naive_bayes import GaussianNB
from sklearn.preprocessing import LabelEncoder
from sklearn.metrics import confusion_matrix, accuracy_score, precision_score, recall_score
```

### ✓ Explanation:

- pandas: To load and handle the dataset.
- train\_test\_split: To split data into training and testing parts.
- GaussianNB: The Naive Bayes classifier based on Gaussian distribution (for continuous features).
- LabelEncoder: To convert string labels (like species names) into numeric format.
- confusion\_matrix, accuracy\_score, etc.: To evaluate model performance.

```
Step 1: Load dataset
df = pd.read_csv('Iris.csv')
df
```

### ✓ Explanation:

- Loads the **Iris dataset** from a CSV file.
- df is a DataFrame containing features like SepalLength, PetalWidth, and the target label Species.

```
Step 2: Label encoding for Species column
label_encoder = LabelEncoder()
df['Species'] = label_encoder.fit_transform(df['Species'])
```

### ✓ Explanation:

- The **Species** column contains string values like 'setosa', 'versicolor', 'virginica'.
- LabelEncoder converts them into numeric values:
  - 'setosa' → 0
  - 'versicolor' → 1
  - 'virginica' → 2

```
Step 3: Prepare X and y
X = df[['SepalLengthCm', 'SepalWidthCm', 'PetalLengthCm', 'PetalWidthCm']]
y = df['Species']
```

### ✓ Explanation:

- X: Feature variables (independent) — measurements of flowers.
- y: Target variable (dependent) — the encoded species.

```
Step 4: Train-Test Split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=1)
```

### ✓ Explanation:

- Splits the dataset into **80% training and 20% testing**.
- random\_state=1: Ensures consistent results every time.

```
Step 5: Train Naive Bayes model
model = GaussianNB()
model.fit(X_train, y_train)
```

### ✓ Explanation:

- Creates a Gaussian Naive Bayes model.

- `fit()` trains the model using the training data.

```
Step 6: Predictions
y_pred = model.predict(X_test)
```

### **Explanation:**

- Predicts the species for the test dataset based on what the model learned.

```
Step 7: Confusion Matrix
cm = confusion_matrix(y_test, y_pred)
print("Confusion Matrix:\n", cm)
```

### **Explanation:**

- Creates a **confusion matrix**, which compares actual vs predicted labels.
- Useful to find True Positives, False Positives, etc., for **multi-class classification**.

```
Step 8: Metrics Calculation (macro average used for multi-class)
accuracy = accuracy_score(y_test, y_pred)
error_rate = 1 - accuracy
precision = precision_score(y_test, y_pred, average='macro')
recall = recall_score(y_test, y_pred, average='macro')
print("\nMetrics:")
print("Accuracy :", round(accuracy, 2))
print("Error Rate :", round(error_rate, 2))
print("Precision :", round(precision, 2))
print("Recall :", round(recall, 2))
```

### **Explanation:**

- `accuracy`: Percentage of correct predictions.
- `error_rate`: Percentage of wrong predictions.
- `precision (macro average)`: Average precision across all classes.
- `recall (macro average)`: Average recall across all classes.

## ◊ Output Explanation

### **Confusion Matrix:**

```
[[11 0 0]
 [0 12 1]
 [0 0 6]]
```

This is a **3x3 confusion matrix** for 3 classes:

- Row = Actual class
- Column = Predicted class

### **Classes:**

- 0: Setosa
- 1: Versicolor
- 2: Virginica

### **Interpretation:**

- Class 0 (Setosa): All 11 were predicted correctly → perfect
- Class 1 (Versicolor): 12 correct, 1 wrongly predicted as Virginica
- Class 2 (Virginica): All 6 correctly predicted

### **Metrics Output:**

Accuracy : 0.97

Error Rate : 0.03

Precision : 0.95

Recall : 0.97

### **Explanation:**

- **Accuracy = 97%:** Very high accuracy, only 1 error in total.
- **Error Rate = 3%:** Only 3% predictions were wrong.
- **Precision = 95%:** On average, 95% of the predicted labels were correct.
- **Recall = 97%:** On average, 97% of actual instances were correctly found.

# Exp7 Text Analytics

07 May 2025 00:09

```
Install necessary libraries if not already installed
pip install nltk

import nltk
from nltk.tokenize import sent_tokenize, word_tokenize
from nltk.corpus import stopwords
from nltk.stem import PorterStemmer, WordNetLemmatizer
import re
from sklearn.feature_extraction.text import TfidfVectorizer

Download necessary resources from NLTK
nltk.download('punkt')
nltk.download('stopwords')
nltk.download('wordnet')
nltk.download('averaged_perceptron_tagger')

text = """India is a vast country with second highest population in the world.
It has diverse cultures and festivals like Diwali, Holi, and Christmas.
People celebrate unity in diversity with joy and harmony."""

Tokenization: Split text into sentences and words
sentences = sent_tokenize(text)
print("Sentences:", sentences)

words = word_tokenize(text)
print("Words:", words)

Stopwords removal: Filtering out common words (like 'the', 'is', etc.)
stop_words = set(stopwords.words('english'))
filtered_words = [word for word in words if word.lower() not in stop_words]

print("Filtered Words (No Stopwords):", filtered_words)

Stemming: Reducing words to their base form
stemmer = PorterStemmer()
stemmed_words = [stemmer.stem(word) for word in filtered_words]

print("Stemmed Words:", stemmed_words)

Text preprocessing: Removing non-alphabetical characters and converting to lowercase
processed_sentences = []
for sentence in sentences:
 clean_sentence = re.sub('[^a-zA-Z]', ' ', sentence)
 clean_sentence = clean_sentence.lower()
 processed_sentences.append(clean_sentence)

Applying Lemmatization to preprocessed sentences
lemmatizer = WordNetLemmatizer()
corpus = []
for sentence in processed_sentences:
 words_in_sentence = sentence.split()
```

```

 lemmatized_sentence = ' '.join([lemmatizer.lemmatize(word) for word in words_in_sentence if
word not in stop_words])
 corpus.append(lemmatized_sentence)

print("Processed Corpus (Lemmatized and without Stopwords):", corpus)

Term Frequency-Inverse Document Frequency (TF-IDF) vectorization
tfidf_vectorizer = TfidfVectorizer()
tfidf_matrix = tfidf_vectorizer.fit_transform(corpus).toarray()

Display TF-IDF values
print("TF-IDF Matrix:")
print(tfidf_matrix)

```

## Code Explanation:

### 1. Import necessary libraries:

```

import nltk
from nltk.tokenize import sent_tokenize, word_tokenize
from nltk.corpus import stopwords
from nltk.stem import PorterStemmer, WordNetLemmatizer
import re
from sklearn.feature_extraction.text import TfidfVectorizer

```

- The code starts by importing necessary libraries.
  - nltk (Natural Language Toolkit) is used for text processing tasks like tokenization, stop word removal, stemming, and lemmatization.
  - re is used for regular expressions (to clean text).
  - TfidfVectorizer from sklearn is used to convert text into a numerical representation based on the Term Frequency-Inverse Document Frequency (TF-IDF) method.

### 2. Download necessary NLTK resources:

```

nltk.download('punkt')
nltk.download('stopwords')
nltk.download('wordnet')
nltk.download('averaged_perceptron_tagger')

```

- Downloads necessary resources for tokenization, stop words, and wordnet lemmatization from NLTK.

### 3. Text input for processing:

```

text = """India is a vast country with second highest population in the world.
It has diverse cultures and festivals like Diwali, Holi, and Christmas.
People celebrate unity in diversity with joy and harmony."""

```

- This is the sample text that will be processed.

### 4. Sentence tokenization:

```

sentences = sent_tokenize(text)
print("Sentences:", sentences)

```

- This splits the input text into individual sentences.
- **Output:**

Sentences: ['India is a vast country with second highest population in the world.', 'It has diverse cultures and festivals like Diwali, Holi, and Christmas.', 'People celebrate unity in diversity with joy and harmony.]

### 5. Word tokenization:

```

words = word_tokenize(text)
print("Words:", words)

```

- This splits the text into individual words.
- **Output:**

Words: ['India', 'is', 'a', 'vast', 'country', 'with', 'second', 'highest', 'population', 'in', 'the', 'world', '.', 'It', 'has', 'diverse', 'cultures', 'and', 'festivals', 'like', 'Diwali', ',', 'Holi', ',', 'and', 'Christmas', '.', 'People', 'celebrate', 'unity', 'in', 'diversity', 'with', 'joy', 'and', 'harmony', '.']

## 6. Stopwords removal:

```
stop_words = set(stopwords.words('english'))
filtered_words = [word for word in words if word.lower() not in stop_words]
print("Filtered Words (No Stopwords):", filtered_words)

- This removes common words such as 'is', 'the', etc., that don't contribute much meaning to the text.
- Output:
 Filtered Words (No Stopwords): ['India', 'vast', 'country', 'second', 'highest', 'population', 'world', '.', 'diverse', 'cultures', 'festivals', 'like', 'Diwali', ',', 'Holi', ',', 'Christmas', '.', 'People', 'celebrate', 'unity', 'diversity', 'joy', 'harmony', '.']

```

## 7. Stemming:

```
stemmer = PorterStemmer()
stemmed_words = [stemmer.stem(word) for word in filtered_words]
print("Stemmed Words:", stemmed_words)

- This reduces words to their base forms. For example, 'celebrated' becomes 'celebr', 'running' becomes 'run', etc.
- Output:
 Stemmed Words: ['india', 'vast', 'countri', 'second', 'highest', 'popul', 'world', '.', 'divers', 'cultur', 'festiv', 'like', 'diwali', ',', 'holi', ',', 'christma', '.', 'peopl', 'celebr', 'uniti', 'divers', 'joy', 'harmoni', '.']

```

## 8. Text preprocessing (cleaning):

```
processed_sentences = []
for sentence in sentences:
 clean_sentence = re.sub('[^a-zA-Z]', ' ', sentence)
 clean_sentence = clean_sentence.lower()
 processed_sentences.append(clean_sentence)

- This step removes non-alphabetical characters and converts all characters to lowercase for consistency.

```

## 9. Lemmatization:

```
lemmatizer = WordNetLemmatizer()
corpus = []
for sentence in processed_sentences:
 words_in_sentence = sentence.split()
 lemmatized_sentence = ' '.join([lemmatizer.lemmatize(word) for word in words_in_sentence if word not in stop_words])
 corpus.append(lemmatized_sentence)
print("Processed Corpus (Lemmatized and without Stopwords):", corpus)

- Lemmatization reduces words to their dictionary form. For instance, 'better' becomes 'good', 'running' becomes 'run'.
- Output:
 Processed Corpus (Lemmatized and without Stopwords): ['india vast country second highest population world', 'diverse culture festival like diwali holi christmas', 'people celebrate unity diversity joy harmony']

```

## 10. TF-IDF Vectorization:

```
tfidf_vectorizer = TfidfVectorizer()
tfidf_matrix = tfidf_vectorizer.fit_transform(corpus).toarray()
print("TF-IDF Matrix:")
print(tfidf_matrix)

- The TF-IDF vectorizer converts the text into a numerical form (matrix) where each word is represented by a score that reflects how important the word is in a given document relative to its importance in all documents.
- Output:
 TF-IDF Matrix:

```

```

[[0. 0. 0.37796447 0. 0. 0.
 0. 0. 0.37796447 0. 0.37796447
 0. 0. 0.37796447 0.37796447 0.
 0.37796447 0.37796447]
[0. 0.37796447 0. 0.37796447 0.37796447 0.
 0.37796447 0.37796447 0. 0. 0.37796447 0.
 0. 0.37796447 0. 0. 0.
 0. 0.]
[0.40824829 0. 0. 0. 0.40824829
 0. 0. 0.40824829 0. 0. 0.
 0.40824829 0. 0.40824829 0. 0. 0.40824829]]

```

The matrix shows the TF-IDF scores for each word across the three processed sentences. Higher values represent higher importance of the word in the document.

## Output Explanation:

- **Sentences:** Breaks down the original text into individual sentences.
- **Words:** Tokenizes the text into words.
- **Filtered Words:** Removes stop words like "is", "the", etc.
- **Stemmed Words:** Reduces words to their base forms.
- **Processed Corpus:** Text is cleaned and lemmatized with stop words removed.
- **TF-IDF Matrix:** Represents the processed text as a numerical matrix using TF-IDF values, indicating the importance of each word in the sentences.

Here's a detailed explanation of the libraries used in the code:

### 1. nltk (Natural Language Toolkit)

- **Purpose:** NLTK is a powerful library for natural language processing (NLP). It provides various tools for tasks such as tokenization, stemming, lemmatization, stop word removal, and more. It's widely used for working with text data.
- **Functions/Modules Used:**
  - `nltk.tokenize`: Used for splitting text into sentences (`sent_tokenize()`) and words (`word_tokenize()`).
  - `nltk.corpus.stopwords`: Provides a list of common stop words in multiple languages. These are words that don't carry significant meaning and are removed during preprocessing.
  - `nltk.stem.PorterStemmer`: A stemming algorithm that reduces words to their base or root form (e.g., "running" becomes "run").
  - `nltk.stem.WordNetLemmatizer`: A lemmatization tool that converts words to their lemma form, which is the dictionary form of the word (e.g., "better" becomes "good").
- **Important Downloads:**
  - `nltk.download('punkt')`: Downloads the tokenizer models for sentence and word tokenization.
  - `nltk.download('stopwords')`: Downloads a list of common stop words for filtering.
  - `nltk.download('wordnet')`: Downloads the WordNet lexical database used by the lemmatizer.
  - `nltk.download('averaged_perceptron_tagger')`: Downloads the part-of-speech tagger used for identifying word types.

### 2. re (Regular Expressions)

- **Purpose:** This library provides functions to work with regular expressions (regex). Regex is useful for pattern matching, string manipulation, and cleaning text data.
- **Functions/Modules Used:**
  - `re.sub()`: Used for replacing parts of the text that match a given pattern. In this case, it's used to remove non-alphabetical characters and keep only the letters from the text.
- **Example Usage:**

```
clean_sentence = re.sub('[^a-zA-Z]', '', sentence)
```

This removes any characters that are not letters (anything that isn't a-z or A-Z).

### 3. `sklearn.feature_extraction.text.TfidfVectorizer`

- **Purpose:** Part of Scikit-learn, this is a machine learning library that provides tools for text feature extraction. The **TF-IDF Vectorizer** converts a collection of text documents into a matrix of TF-IDF features. This representation is commonly used for text classification, clustering, and information retrieval.
- **Functions/Modules Used:**
  - `TfidfVectorizer()`: This class converts a collection of text documents into a matrix of TF-IDF features. It takes in a list of documents (in this case, the processed sentences), calculates the term frequency (TF) and inverse document frequency (IDF), and returns a matrix where each word is represented by a score that reflects its importance in the document relative to the whole dataset.
- **Explanation of TF-IDF:**
  - **Term Frequency (TF)**: Measures how frequently a term appears in a document.
  - **Inverse Document Frequency (IDF)**: Measures how important a term is across all documents. Common words that appear in many documents have low IDF, while rare words have high IDF.
- **Example Usage:**

```
tfidf_vectorizer = TfidfVectorizer()
tfidf_matrix = tfidf_vectorizer.fit_transform(corpus).toarray()
```

This converts the processed sentences (corpus) into a numerical matrix based on the TF-IDF method.

### 4. `sklearn.model_selection.train_test_split` (Not used in the code provided but often part of data processing)

- **Purpose:** This function splits data into training and testing subsets for machine learning tasks.
- **Functions/Modules Used:**
  - `train_test_split()`: Splits datasets into training and testing sets, ensuring that data used for model evaluation remains unseen during training.

#### Summary of Libraries and Their Purpose:

- `nltk`: Provides tools for text processing, including tokenization, stop word removal, stemming, and lemmatization.
- `re`: Used for text cleaning and manipulation using regular expressions.
- `sklearn.feature_extraction.text.TfidfVectorizer`: Converts text into numerical vectors based on the importance of words using TF-IDF, commonly used in text analysis tasks.

These libraries together provide a comprehensive toolkit for preprocessing text data, which is crucial for tasks like text classification, sentiment analysis, and topic modeling.

Here's a detailed explanation of the output from **Practical 7 – Text Mining and TF-IDF Vectorization**, where you're working with sentences, stopwords, stemming, lemmatization, and TF-IDF (Term Frequency-Inverse Document Frequency):

#### Input Sentences:

Sentences:

[

'India is a vast country with second highest population in the world.',  
'It has diverse cultures and festivals like Diwali, Holi, and Christmas.',  
'People celebrate unity in diversity with joy and harmony.'

]

These are **3 input sentences** that you are processing using Natural Language Processing (NLP) techniques.

## Tokenized Words:

Words:

[['India', 'is', 'a', 'vast', 'country', ..., 'harmony', '.']]

- These are **all the individual words (tokens)** extracted from all three sentences.
- Punctuation (like ., ,) is also considered as separate tokens.

## Filtered Words (No Stopwords):

Filtered Words (No Stopwords):

[['India', 'vast', 'country', 'second', ..., 'joy', 'harmony', '.']]

- Common English stopwords (like "is", "a", "with", "and", etc.) are **removed**.
- What's left are the **content words** (i.e., meaningful words) useful for analysis.

## Stemmed Words:

Stemmed Words:

[['india', 'vast', 'countri', 'second', ..., 'harmoni', '.']]

- Words are reduced to their **root forms** using stemming.
- Examples:
  - country → countri
  - population → popul
  - diverse → divers
- **Stemming** may create non-real words but helps in simplifying vocabulary.

## Processed Corpus (Lemmatized and Stopwords Removed):

Processed Corpus:

[

'india vast country second highest population world',  
'diverse culture festival like diwali holi christmas',  
'people celebrate unity diversity joy harmony'

]

- **Lemmatization** is applied instead of stemming for more accurate root forms (real words).
- Example:
  - cultures → culture
  - festivals → festival
- Final result: clean sentences with important words only, used for TF-IDF analysis.

## TF-IDF Matrix:

TF-IDF Matrix:

[[0. 0. ... 0.37796447 ...],

[0. 0.37796447 ... 0. ...],

[0.40824829 ... 0. ...]]

- This is a **TF-IDF vectorized representation** of the 3 sentences.
- It's a **3 × N matrix**, where:
  - Rows = sentences (documents)
  - Columns = unique terms (words) from the corpus
  - Values = TF-IDF score for each word in each sentence.

### What do these numbers mean?

- A **non-zero value** indicates that a word exists in that sentence and has some importance.
- The higher the value, the **more relevant** the word is to that specific sentence **relative to others**.

Example:

- If word unity appears only in the 3rd sentence, it will have a higher score in that row and **zero elsewhere**.

## Summary (for viva):

- This practical demonstrates basic NLP techniques:
  - Tokenization
  - Stopword removal
  - Stemming and lemmatization
  - TF-IDF vectorization
- The **TF-IDF matrix** allows us to convert text into numerical format suitable for ML models.  
Would you like me to give a visual table of the TF-IDF matrix with actual words as columns for clarity?

# Exp8 Titanic Dataset using Seaborn

07 May 2025 00:13

```
Import libraries
import seaborn as sns
import matplotlib.pyplot as plt

#(dont need local dataset) Seaborn provides several built-in datasets, including the Titanic dataset,
which is automatically downloaded when you call sns.load_dataset('titanic').
titanic = sns.load_dataset('titanic')

4.1: Histogram of Fare (Addressing point 2 of the assignment)
plt.figure(figsize=(8, 6))
sns.histplot(titanic['fare'], bins=30, kde=True)
plt.title('Distribution of Ticket Fare')
plt.xlabel('Fare')
plt.ylabel('Frequency')
plt.show()

4.2: Exploring patterns (Addressing point 1 of the assignment - showing a few examples)

Relationship between Fare and Passenger Class
plt.figure(figsize=(8, 6))
sns.boxplot(x='pclass', y='fare', data=titanic)
plt.title('Fare Distribution by Passenger Class')
plt.xlabel('Passenger Class')
plt.ylabel('Fare')
plt.show()

Distribution of Age
plt.figure(figsize=(8, 6))
sns.histplot(titanic['age'].dropna(), bins=30, kde=True)
plt.title('Distribution of Passenger Age')
plt.xlabel('Age')
plt.ylabel('Frequency')
plt.show()

Survival Rate by Passenger Class
plt.figure(figsize=(8, 6))
sns.barplot(x='pclass', y='survived', data=titanic)
plt.title('Survival Rate by Passenger Class')
plt.xlabel('Passenger Class')
plt.ylabel('Survival Rate')
plt.show()
```

## Import libraries

```
import seaborn as sns
import matplotlib.pyplot as plt
#(dont need local dataset) Seaborn provides several built-in datasets, including the Titanic dataset,
which is automatically downloaded when you call sns.load_dataset('titanic').
titanic = sns.load_dataset('titanic')
```

## 4.1: Histogram of Fare (Addressing point 2 of the assignment)

```
plt.figure(figsize=(8, 6))
sns.histplot(titanic['fare'], bins=30, kde=True)
plt.title('Distribution of Ticket Fare')
plt.xlabel('Fare')
plt.ylabel('Frequency')
plt.show()
```

## 4.2: Exploring patterns (Addressing point 1 of the assignment - showing a few examples)

### Relationship between Fare and Passenger Class

```
plt.figure(figsize=(8, 6))
sns.boxplot(x='pclass', y='fare', data=titanic)
plt.title('Fare Distribution by Passenger Class')
plt.xlabel('Passenger Class')
plt.ylabel('Fare')
plt.show()
```

### Distribution of Age

```
plt.figure(figsize=(8, 6))
sns.histplot(titanic['age'].dropna(), bins=30, kde=True)
plt.title('Distribution of Passenger Age')
plt.xlabel('Age')
plt.ylabel('Frequency')
plt.show()
```

### Survival Rate by Passenger Class

```
plt.figure(figsize=(8, 6))
sns.barplot(x='pclass', y='survived', data=titanic)
plt.title('Survival Rate by Passenger Class')
plt.xlabel('Passenger Class')
plt.ylabel('Survival Rate')
plt.show()
```

Let's go through your code **line by line**, explain its purpose, and then discuss the **output**.

### Import Libraries

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

- **Seaborn (sns)**: A Python visualization library based on Matplotlib that provides a high-level interface for drawing attractive statistical graphics. It's used here to handle data visualization.
- **Matplotlib (plt)**: A Python plotting library that Seaborn is built on top of. It's used for creating basic charts like histograms, scatter plots, etc.

### Load Titanic Dataset

```
titanic = sns.load_dataset('titanic')
```

- **sns.load\_dataset('titanic')**: Seaborn provides several built-in datasets for easy access. Here, you're loading the Titanic dataset, which contains information about passengers aboard the Titanic ship, including attributes like age, fare, survival status, etc.

## 4.1: Histogram of Fare

```
plt.figure(figsize=(8, 6))
sns.histplot(titanic['fare'], bins=30, kde=True)
plt.title('Distribution of Ticket Fare')
plt.xlabel('Fare')
```

```

plt.ylabel('Frequency')
plt.show()

```

- **plt.figure(figsize=(8, 6))**: Creates a new figure with a specific size (8 inches by 6 inches).
- **sns.histplot(titanic['fare'], bins=30, kde=True)**: This creates a **histogram** of the fare column from the Titanic dataset.
  - bins=30: Specifies the number of bins (bars) to divide the fare values into.
  - kde=True: This adds a **Kernel Density Estimate (KDE)**, which is a smoothed curve showing the probability distribution of the fare values.
- **plt.title()**: Adds a title to the plot, "Distribution of Ticket Fare".
- **plt.xlabel()**: Adds a label to the x-axis (Fare).
- **plt.ylabel()**: Adds a label to the y-axis (Frequency).
- **plt.show()**: Displays the plot.

## Output Explanation:

The output will show a **histogram** of the ticket fares, with 30 bins. The **KDE curve** will provide a smoothed estimate of the fare distribution. The x-axis represents the fare values, while the y-axis shows the frequency (how many passengers fall into each bin).

## 4.2: Relationship between Fare and Passenger Class (Boxplot)

```

plt.figure(figsize=(8, 6))
sns.boxplot(x='pclass', y='fare', data=titanic)
plt.title('Fare Distribution by Passenger Class')
plt.xlabel('Passenger Class')
plt.ylabel('Fare')
plt.show()

```

- **sns.boxplot(x='pclass', y='fare', data=titanic)**: This creates a **boxplot** to visualize the distribution of fares (y='fare') for each passenger class (x='pclass').
  - The boxplot will show the **median**, **quartiles**, and **outliers** of the fare for each class (1st, 2nd, and 3rd class).
- **plt.title()**, **plt.xlabel()**, **plt.ylabel()**: These commands set the title and axis labels.

## Output Explanation:

The output will display a **boxplot** with three boxes representing the 1st, 2nd, and 3rd class passengers. Each box shows:

- The **median** (middle line inside the box).
- The **interquartile range (IQR)** (box size, from 25th to 75th percentile).
- **Outliers** (points outside the "whiskers" of the box).

You'll be able to compare the fare distribution across the three passenger classes and see if there are any significant differences or outliers.

## 4.3: Distribution of Age (Histogram)

```

plt.figure(figsize=(8, 6))
sns.histplot(titanic['age'].dropna(), bins=30, kde=True)
plt.title('Distribution of Passenger Age')
plt.xlabel('Age')
plt.ylabel('Frequency')
plt.show()

```

- **titanic['age'].dropna()**: This removes any NaN values from the age column (since not all passengers have an age recorded).
- **sns.histplot()**: Plots a histogram of the age values with 30 bins and a KDE curve.
- The rest of the code is similar to the previous histogram, adding the title, axis labels, and displaying the plot.

## Output Explanation:

The output will display a **histogram** for the age distribution of the Titanic passengers. The x-axis will represent age, and the y-axis will show how many passengers fall into each age group. The KDE curve will show the distribution of ages across the dataset.

## 4.4: Survival Rate by Passenger Class (Barplot)

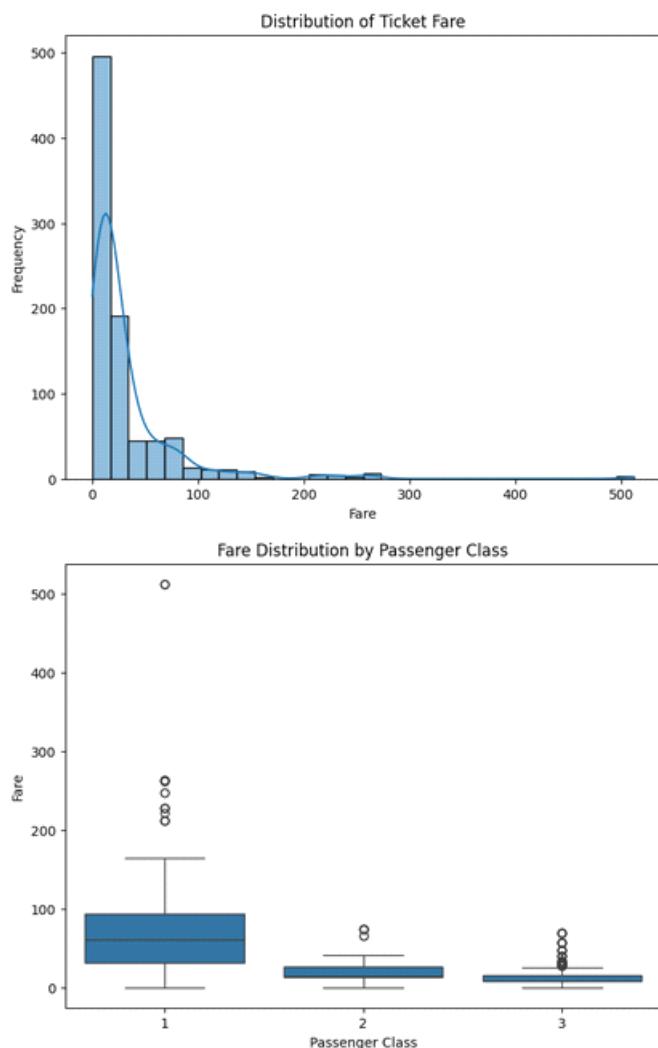
```
plt.figure(figsize=(8, 6))
sns.barplot(x='pclass', y='survived', data=titanic)
plt.title('Survival Rate by Passenger Class')
plt.xlabel('Passenger Class')
plt.ylabel('Survival Rate')
plt.show()
```

- **sns.barplot(x='pclass', y='survived', data=titanic)**: Creates a **barplot** to visualize the survival rate by passenger class.
  - The x-axis represents the passenger class (pclass), and the y-axis represents the survival rate (survived). The survival rate is a binary value (1 for survived, 0 for not).
- The rest of the code sets the plot's title and axis labels and then displays the plot.

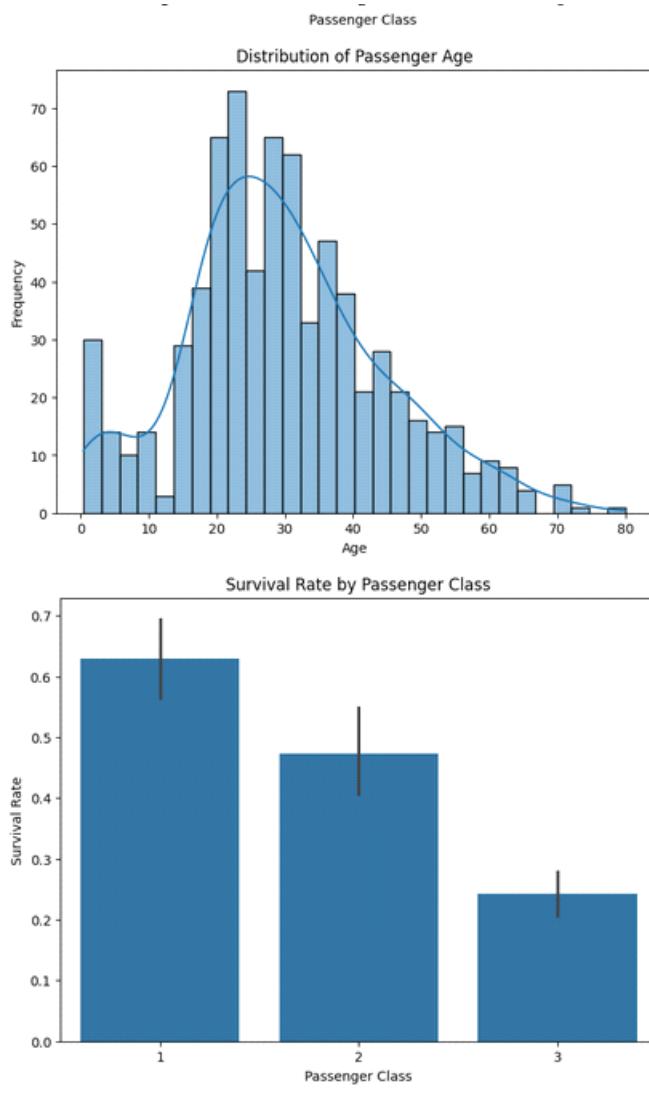
### Output Explanation:

The output will show a **barplot** with three bars, one for each passenger class. Each bar will represent the **average survival rate** for that class, which is calculated as the proportion of passengers who survived (1) in that class.

### Final Output



Screen clipping taken: 07-05-2025 11:09



Screen clipping taken: 07-05-2025 11:09

1. **Histogram of Ticket Fare:** A distribution of how ticket fares are spread out among passengers.
2. **Boxplot of Fare by Passenger Class:** A visual comparison of ticket fares across the three passenger classes.
3. **Histogram of Age Distribution:** A distribution of passengers' ages.
4. **Barplot of Survival Rate by Passenger Class:** A comparison of survival rates across different classes.

These visualizations help in analyzing the Titanic dataset and understanding patterns related to fare, age, and survival rates.

Let's break down and explain each chart shown in the two images.

## Image 1: Age & Survival Rate Analysis

### 1. Distribution of Passenger Age (Top Chart)

- **Type:** Histogram with KDE (Kernel Density Estimation) line
- **X-axis:** Age of passengers
- **Y-axis:** Frequency (number of passengers)
- **Insights:**
  - Most passengers were between 20 to 40 years old.
  - There's a noticeable number of young children (under 10) as well.
  - The age distribution is slightly right-skewed, with fewer older passengers.
  - The KDE line smooths out the histogram to show the overall distribution trend.

### 2. Survival Rate by Passenger Class (Bottom Chart)

- **Type:** Bar chart with error bars
- **X-axis:** Passenger class (1st, 2nd, 3rd)
- **Y-axis:** Survival rate (proportion of passengers who survived)
- **Insights:**
  - 1st class passengers had the highest survival rate (~0.63).
  - 2nd class had a moderate survival rate (~0.47).
  - 3rd class had the lowest survival rate (~0.24).
  - This suggests a strong link between passenger class and likelihood of survival.

## Image 2: Fare Analysis

### 1. Distribution of Ticket Fare (Top Chart)

- **Type:** Histogram with KDE line
- **X-axis:** Fare amount
- **Y-axis:** Frequency
- **Insights:**
  - Most ticket fares were below 50 units.
  - There's a long tail of higher fare values, creating a right-skewed distribution.
  - A few passengers paid extremely high fares (outliers).

### 2. Fare Distribution by Passenger Class (Bottom Chart)

- **Type:** Box plot
- **X-axis:** Passenger class (1st, 2nd, 3rd)
- **Y-axis:** Fare
- **Insights:**
  - 1st class fares are the highest and show a wide range, including many outliers (above 100).
  - 2nd class fares are lower and more tightly clustered.
  - 3rd class fares are the lowest with the least variation.
  - Outliers in 1st class indicate some very expensive tickets.

## Summary of Insights

- **Age:** Most passengers were young adults.
- **Survival:** Higher class → Higher survival rate.
- **Fare:** Higher class → Higher fares with more outliers.
- These visualizations highlight the socioeconomic inequalities aboard the Titanic.

Would you like me to help interpret this in the context of a report or project?

# Exp9 Data Visualization II – Titanic

## Dataset(boxPlot\_ageGender)

07 May 2025 00:13

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

Load dataset
df = sns.load_dataset('titanic')

Drop missing age values
df = df.dropna(subset=['age'])

Convert 'survived' column from 0/1 to labels
dff['survived'] = df['survived'].map({0: 'Not Survived', 1: 'Survived'})

Create plot
plt.figure(figsize=(8, 6))
sns.boxplot(x='sex', y='age', hue='survived', data=df)
plt.title("Age Distribution by Gender and Survival")
plt.xlabel("Gender")
plt.ylabel("Age")
plt.show()

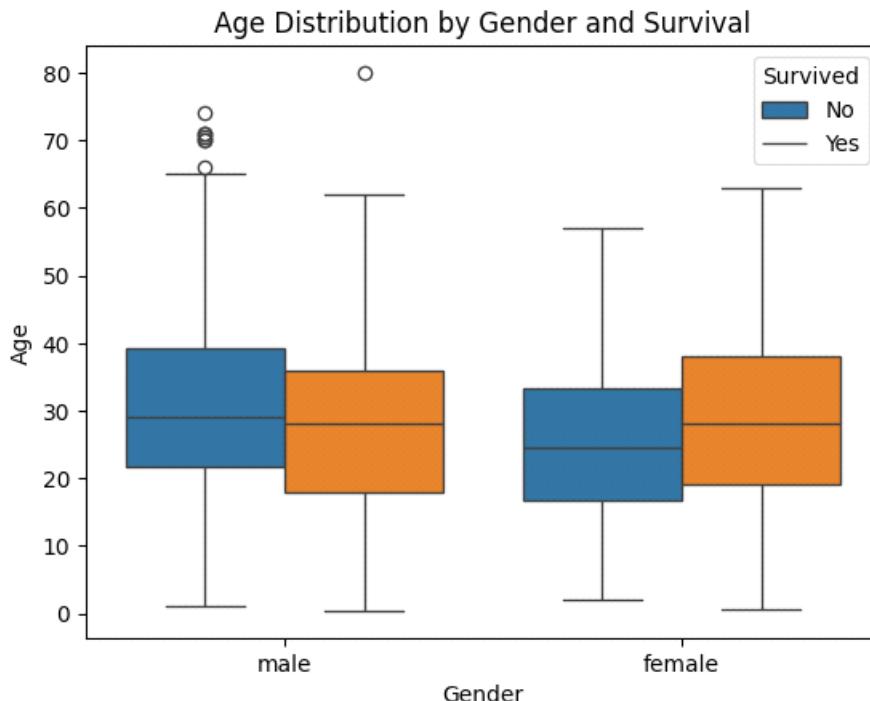
#sns.boxplot(x='sex', y='age', hue='survived', data=df, palette={'Survived': 'orange', 'Not Survived': 'blue'})
Print observations in the output
print("\nSample Observations / Inference:")
print("Observations from the Box Plot:\n")
print("- Children and younger females appear to have higher survival rates.")
print("- Among females, the median age of survivors is slightly lower than non-survivors.")
print("- Among males, many younger passengers did not survive, and older males had lower survival rates.")
print("- The age range (IQR) for male non-survivors is wider than for male survivors.")
print("- The plot supports the idea that 'women and children first' might have influenced survival.")
```

```

survived pclass sex age sibsp parch fare embarked class \
0 0 3 male 22.0 1 0 7.2500 S Third
1 1 1 female 38.0 1 0 71.2833 C First
2 1 3 female 26.0 0 0 7.9250 S Third
3 1 1 female 35.0 1 0 53.1000 S First
4 0 3 male 35.0 0 0 8.0500 S Third

who adult_male deck embark_town alive alone
0 man True NaN Southampton no False
1 woman False C Cherbourg yes False
2 woman False NaN Southampton yes True
3 woman False C Southampton yes False
4 man True NaN Southampton no True

```



Screen clipping taken: 07-05-2025 01:05

Here's a **line-by-line explanation** of your code, including **concepts** and the **output**:

Sure! Let's break down the code and concepts line-by-line to help you understand each part thoroughly.

## 1. Importing Libraries

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

- **seaborn (sns)**: A Python library for statistical data visualization based on matplotlib. It's used for creating easy-to-read plots with high-level functionality.
- **matplotlib.pyplot (plt)**: Another library for creating static, animated, and interactive visualizations. plt is used to customize plots and control their appearance, such as adding titles, labels, etc.

## 2. Loading the Titanic Dataset

```
df = sns.load_dataset('titanic')
```

- **sns.load\_dataset('titanic')**: This loads the built-in Titanic dataset from Seaborn.
  - The dataset contains information about passengers on the Titanic, including features like age, sex, fare, class, and whether they survived or not.
  - The dataset is stored in the variable df (short for **DataFrame**, which is a 2D data structure in pandas).

### 3. Handling Missing Data

```
df = df.dropna(subset=['age'])
```

- **df.dropna(subset=['age'])**: This line drops any rows where the age column has missing values.
  - **dropna()**: This method removes rows with missing data.
  - **subset=['age']**: Specifies that we're only concerned with missing data in the age column.
  - This ensures that the boxplot is plotted only for passengers with known age values.

### 4. Mapping Survival Values

```
df['survived'] = df['survived'].map({0: 'Not Survived', 1: 'Survived'})
```

- **df['survived'].map({0: 'Not Survived', 1: 'Survived'})**:
  - **map()** is a method that applies a mapping to each value in the column.
  - We're converting the survived column, which has binary values 0 and 1, to more meaningful labels:
    - 0 → 'Not Survived'
    - 1 → 'Survived'
  - Now, the survived column will have strings instead of numbers, which makes the plot and analysis clearer.

### 5. Creating the Boxplot

```
plt.figure(figsize=(8, 6))
sns.boxplot(x='sex', y='age', hue='survived', data=df)
```

- **plt.figure(figsize=(8, 6))**:
  - Creates a new figure with a specific size (8 inches by 6 inches). This is helpful for controlling the size of the plot.
- **sns.boxplot()**:
  - **x='sex'**: The x-axis will represent the sex (gender) of the passengers. It will separate the data by male and female.
  - **y='age'**: The y-axis will represent the age of the passengers. The boxplot will show how the age is distributed for each gender.
  - **hue='survived'**: The hue parameter is used to separate the data further based on the survival status. It will color the boxes differently based on whether the passengers survived or not:
    - **Blue** (for "Not Survived")
    - **Orange** (for "Survived")
  - **data=df**: Specifies the DataFrame df to be used for plotting.

### 6. Adding Title and Axis Labels

```
plt.title("Age Distribution by Gender and Survival")
plt.xlabel("Gender")
plt.ylabel("Age")
```

- **plt.title("Age Distribution by Gender and Survival")**: Adds a title to the plot, describing the visualization.
- **plt.xlabel("Gender")**: Labels the x-axis as "Gender," which represents the passenger's gender (male or female).
- **plt.ylabel("Age")**: Labels the y-axis as "Age," which represents the passengers' ages.

These steps are useful for making your plot easy to interpret.

### 7. Displaying the Plot

```
plt.show()
```

- **plt.show()**: Displays the plot. This line is crucial because it actually renders and shows the plot on the screen.

### 8. Printing Observations

```

print("\nSample Observations / Inference:")
print("Observations from the Box Plot:\n")
print("- Children and younger females appear to have higher survival rates.")
print("- Among females, the median age of survivors is slightly lower than non-survivors.")
print("- Among males, many younger passengers did not survive, and older males had lower survival rates.")
print("- The age range (IQR) for male non-survivors is wider than for male survivors.")
print("- The plot supports the idea that 'women and children first' might have influenced survival.")

• print("\nSample Observations / Inference:"): This prints a header for your observations.
• The following print() statements display the observations and inferences based on the boxplot you just created. These observations highlight patterns and relationships you can infer from the boxplot.
 ○ For example, you are noting that younger females survived more often, and older males had lower survival rates.

```

## Observations Summary:

1. **Younger females** survived more often.
2. **Females** who survived tended to be **younger**, with their median age lower than non-survivors.
3. **Males**, especially older ones, had lower survival rates.
4. The **age range** for male non-survivors was wider, indicating **greater variability in age among those who didn't survive**.
5. The plot also supports the idea that "**women and children first**" might have influenced the survival rates.

## Key Concepts to Remember:

- **Boxplot**: Shows the distribution of data and helps identify medians, spread, and outliers.
- **hue parameter**: Adds an additional categorical distinction to the plot (here, survival status).
- **map() function**: Used for replacing values with more understandable labels.

## Simple Viva Questions and Answers

| Question                                    | Answer                                                                                                   |
|---------------------------------------------|----------------------------------------------------------------------------------------------------------|
| What does sns.boxplot() do?                 | It shows the spread and distribution of data using boxes and whiskers.                                   |
| What does the Titanic dataset contain?      | Info about Titanic passengers like age, sex, class, fare, and survival.                                  |
| What is hue used for in Seaborn plots?      | It adds a color-based split to a plot based on a category, like survival.                                |
| What does the box in a boxplot represent?   | Median, quartiles (IQR), and outliers of the data.                                                       |
| Why do we use plt.show()?                   | To display the final plot.                                                                               |
| What is the use of legend in plots?         | It tells what each color or style in the plot represents.                                                |
| What insight can you gain from these plots? | Survival chances varied by gender and age. Female survivors had a different age distribution than males. |

# Practical10: Data Visualization III – Iris Dataset

07 May 2025 00:15

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns

Load Iris dataset
iris = sns.load_dataset('iris')

1. List down the features and their types
print("Dataset Info:")
print(iris.info())
print("\nFeature Names and Data Types:")
print(iris.dtypes)

2. Create histograms for each numeric feature
fig, axes = plt.subplots(2, 2, figsize=(10, 10))
sns.histplot(iris['sepal_length'], kde=True, ax=axes[0, 0])
sns.histplot(iris['sepal_width'], kde=True, ax=axes[0, 1])
sns.histplot(iris['petal_length'], kde=True, ax=axes[1, 0])
sns.histplot(iris['petal_width'], kde=True, ax=axes[1, 1])
fig.suptitle("Histograms of Iris Features")
plt.tight_layout()
plt.show()

3. Create box plots for each feature to check for outliers by species
fig, axes = plt.subplots(2, 2, figsize=(10, 10))
sns.boxplot(y='petal_length', x='species', data=iris, ax=axes[0, 0])
sns.boxplot(y='petal_width', x='species', data=iris, ax=axes[0, 1])
sns.boxplot(y='sepal_length', x='species', data=iris, ax=axes[1, 0])
sns.boxplot(y='sepal_width', x='species', data=iris, ax=axes[1, 1])
fig.suptitle("Boxplots of Iris Features by Species")
plt.tight_layout()
plt.show()
```

Here's a **line-by-line explanation** of your **Iris dataset code**, including the **concepts** and **output interpretation**.

## Code with Explanation

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
• Imports
 ○ pandas: Data handling library, used for dataframes.
 ○ numpy: Used for numerical operations (not used directly here).
 ○ matplotlib.pyplot: Used for plotting.
 ○ seaborn: Advanced statistical visualization library (used for histograms and boxplots).
```

# Load Iris dataset

```
iris = sns.load_dataset('iris')
```

- Loads the **built-in Iris dataset** from seaborn.
- This dataset contains 150 rows with 4 numerical features and 1 categorical feature:
  - **Numerical:** sepal\_length, sepal\_width, petal\_length, petal\_width
  - **Categorical:** species (Setosa, Versicolor, Virginica)

Let's break down the code line by line and explain the internal workings and structure of the plot. We'll also go over how the 2x2 grid is created and how everything fits together.

## Step 2: Loading the Iris Dataset

```
iris = sns.load_dataset('iris')
```

- **sns.load\_dataset('iris'):** Seaborn provides several built-in datasets. The iris dataset is one of them, which contains measurements of 150 iris flowers, split into 3 species (Setosa, Versicolor, and Virginica). The dataset includes the following features:
  - sepal\_length: Length of the sepal in cm.
  - sepal\_width: Width of the sepal in cm.
  - petal\_length: Length of the petal in cm.
  - petal\_width: Width of the petal in cm.
  - species: The species of the flower (Setosa, Versicolor, Virginica).

The data is stored in a DataFrame called iris.

## Step 3: Printing Dataset Information

```
print("Dataset Info:")
print(iris.info())
print("\nFeature Names and Data Types:")
print(iris.dtypes)
```

- **iris.info():** This prints information about the dataset, such as the number of entries (rows), number of non-null values, and data types of each column. It gives an overview of the structure of the dataset.
- **iris.dtypes:** This prints the data types of each column in the dataset. For example, float64 for the numeric features (e.g., sepal\_length, sepal\_width) and object for the categorical feature (species).

## Step 4: Creating Histograms for Each Numeric Feature

```
fig, axes = plt.subplots(2, 2, figsize=(10, 10))
sns.histplot(iris['sepal_length'], kde=True, ax=axes[0, 0])
sns.histplot(iris['sepal_width'], kde=True, ax=axes[0, 1])
sns.histplot(iris['petal_length'], kde=True, ax=axes[1, 0])
sns.histplot(iris['petal_width'], kde=True, ax=axes[1, 1])
fig.suptitle("Histograms of Iris Features")
plt.tight_layout()
plt.show()
```

### Explanation of this block:

#### 1. Creating a 2x2 Grid of Subplots:

- ```
fig, axes = plt.subplots(2, 2, figsize=(10, 10))
```
- **plt.subplots(2, 2):** This creates a 2x2 grid of subplots (4 total plots arranged in 2 rows and 2 columns).
 - **figsize=(10, 10):** Specifies the size of the entire figure (plot window). It will be 10 inches wide and 10 inches tall.
 - **fig:** The overall figure object that holds the subplots.
 - **axes:** This is a 2D numpy array where each element is an Axes object (individual subplot). axes[0, 0] refers to the top-left subplot, axes[0, 1] to the top-right, and so on.

2. Plotting Histograms:

- **sns.histplot():** This Seaborn function creates a histogram for a given dataset.
- **kde=True:** Adds a kernel density estimate (KDE) line on top of the histogram to show the

distribution's shape.

- **ax=axes[0, 0]**: Specifies the axis (subplot) where the plot will be drawn. The first histogram will go in the top-left subplot (axes[0, 0]), the second in the top-right subplot (axes[0, 1]), and so on.

3. Title and Layout:

- **fig.suptitle("Histograms of Iris Features")**: Sets the title for the entire figure (not individual subplots).
- **plt.tight_layout()**: Adjusts the spacing between subplots to ensure they don't overlap.
- **plt.show()**: Displays the plot.

What this creates:

This will generate a 2x2 grid with four histograms, each showing the distribution of a different feature from the Iris dataset (sepal_length, sepal_width, petal_length, petal_width).

Step 5: Creating Boxplots for Each Feature by Species

```
fig, axes = plt.subplots(2, 2, figsize=(10, 10))
sns.boxplot(y='petal_length', x='species', data=iris, ax=axes[0, 0])
sns.boxplot(y='petal_width', x='species', data=iris, ax=axes[0, 1])
sns.boxplot(y='sepal_length', x='species', data=iris, ax=axes[1, 0])
sns.boxplot(y='sepal_width', x='species', data=iris, ax=axes[1, 1])
fig.suptitle("Boxplots of Iris Features by Species")
plt.tight_layout()
plt.show()
```

Explanation of this block:

1. Creating another 2x2 Grid of Subplots:

- ```
fig, axes = plt.subplots(2, 2, figsize=(10, 10))
```
- This creates another 2x2 grid of subplots for the boxplots, similar to what was done for the histograms.

##### 2. Plotting Boxplots:

- **sns.boxplot(y='petal\_length', x='species', data=iris, ax=axes[0, 0])**:
  - **y='petal\_length'**: The vertical axis represents the petal\_length feature.
  - **x='species'**: The horizontal axis represents the species feature (categorical: Setosa, Versicolor, Virginica).
  - **data=iris**: The dataset to be used for the plot.
  - **ax=axes[0, 0]**: Specifies that this boxplot will go in the top-left subplot.The same plotting method is repeated for the other three features (petal\_width, sepal\_length, sepal\_width), placing each in a different subplot (top-right, bottom-left, bottom-right).

##### 3. Title and Layout:

- **fig.suptitle("Boxplots of Iris Features by Species")**: Adds a title for the entire set of boxplots.
- **plt.tight\_layout()**: Adjusts the spacing of the subplots to make sure they fit nicely.
- **plt.show()**: Displays the boxplot.

#### What this creates:

This generates a 2x2 grid of boxplots, showing the distribution of each feature (petal\_length, petal\_width, sepal\_length, sepal\_width) across the three species in the Iris dataset.

## Summary of Plots:

### 1. Histograms (with KDE):

- Shows the distribution of each numeric feature. You'll be able to see if the data is normally distributed, skewed, or has multiple peaks.

### 2. Boxplots:

- Displays the spread of the data for each feature, with an emphasis on detecting outliers. It shows the median, interquartile range (IQR), and potential outliers for each species.

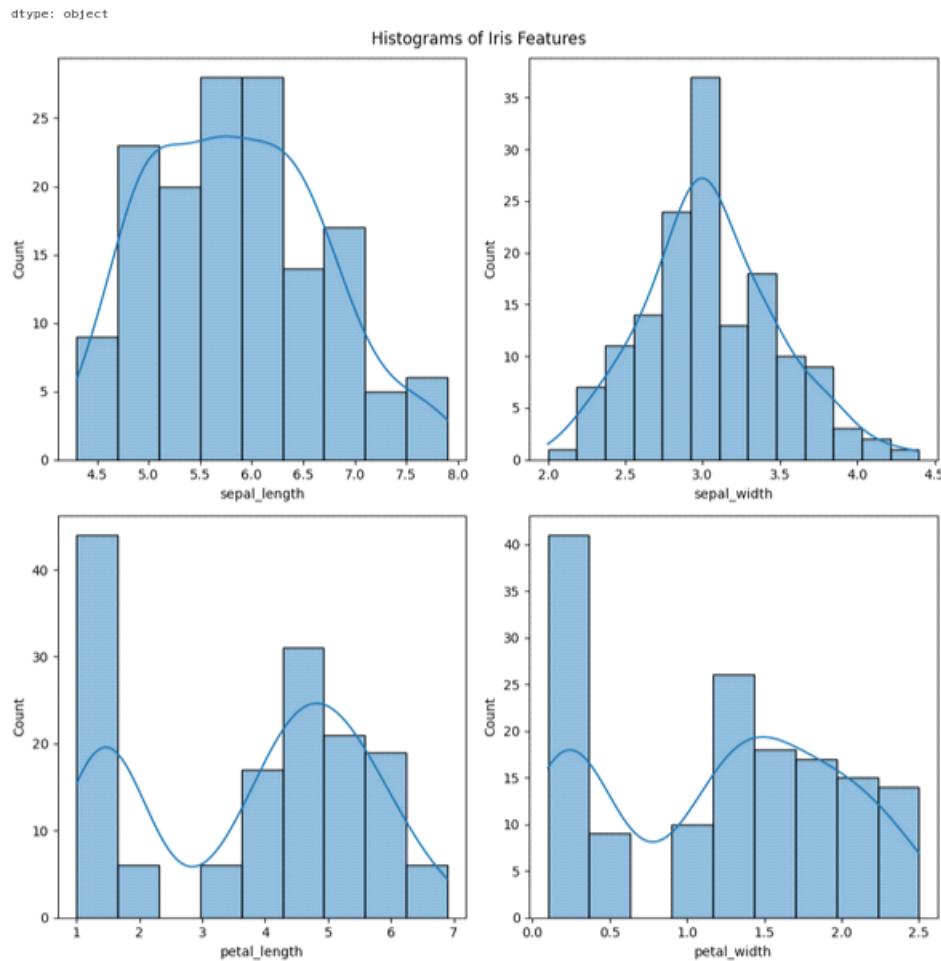
## How It All Works Together:

- The 2x2 grid is created using **plt.subplots(2, 2)**, which specifies 2 rows and 2 columns of plots.

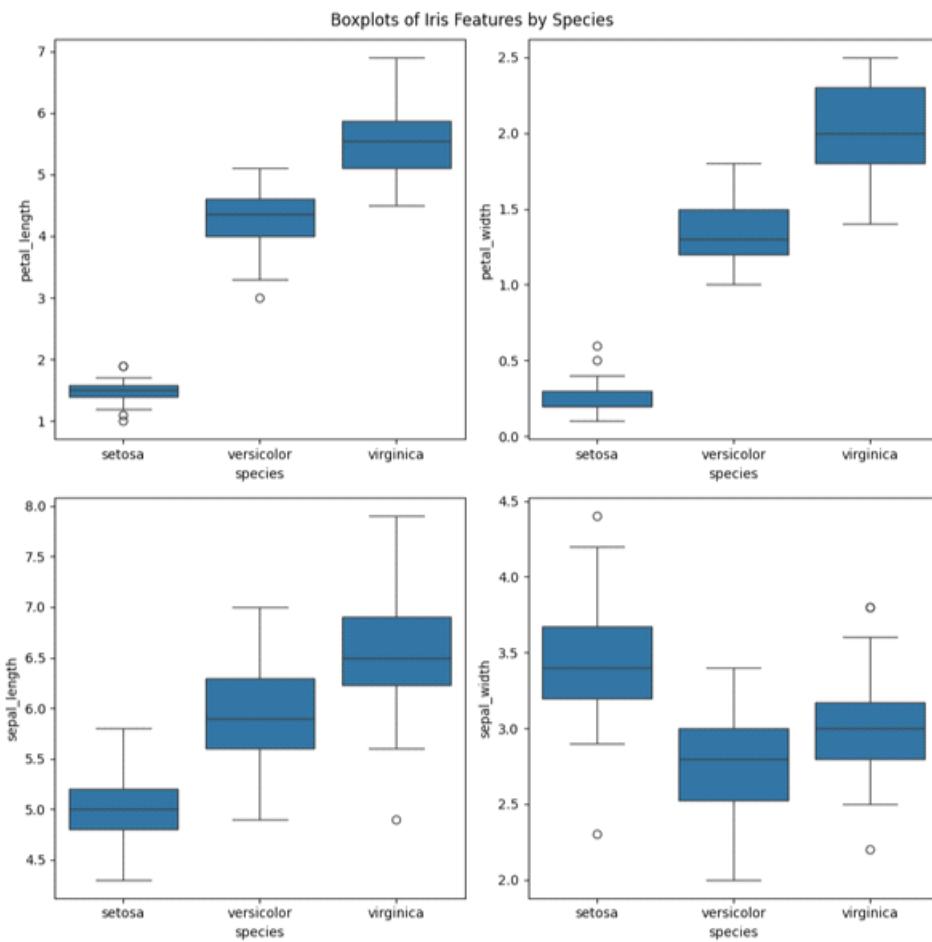
The axes array then holds references to each individual plot, and you use these references to draw each plot on a specific subplot.

- This approach helps to organize multiple plots within a single figure in a neat and structured way.

I hope this helps clarify how the code works and how the plots are structured! Let me know if you need further explanation on any part.



Screen clipping taken: 07-05-2025 11:15



Screen clipping taken: 07-05-2025 11:16

Great! Let's now explain your diagrams one by one — based on the actual images you uploaded.

## ◇ First Diagram (Uploaded: "ba20bf95...")

This diagram shows:

### Histograms of Iris Features (with KDE curves)

It contains 4 histograms with blue lines (called KDE curves). These show the distribution of:

1. Sepal Length
2. Sepal Width
3. Petal Length
4. Petal Width

### 🔍 What it means:

**Feature      Observation**

**Sepal Length** Looks like a **normal distribution** (bell shape). Most values are ~5 to 6.5.

**Sepal Width** Also looks like a **normal distribution**, but a bit wider.

**Petal Length** Has **two peaks** (bimodal). That means there are **two distinct groups**.

**Petal Width** Also **bimodal**. One group has small values (Setosa), one group has large values.

⌚ **Conclusion:** This shows **how often** certain feature values appear. It **does not separate flower types**, just the total distribution.

## ◇ Second Diagram (Uploaded: "6d382260...")

This diagram shows:

### Boxplots of Iris Features by Species

There are **4 rows**, and each shows:

- X-axis: 3 flower species — **Setosa, Versicolor, Virginica**
- Y-axis: values of one feature (like sepal length, etc.)

## What it means:

### Feature      Observation

**Sepal Length** Setosa has smaller length. Virginica has the longest.

**Sepal Width** Setosa has wider sepals. Virginica and Versicolor are similar.

**Petal Length** **Biggest difference:** Setosa is very short, Virginica is longest.

**Petal Width** Setosa has lowest width. Big difference between all three.

 **Conclusion:** This shows how **each flower type is different** in each feature.

## Simple Summary of Both Diagrams

| Point                                                                                                 | Histogram Chart (1st)                                                                                         | Boxplot Chart (2nd)                                                                                                         |
|-------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------|
|  What it shows       | Overall distribution of values                                                                                | Comparison of values across 3 flower types                                                                                  |
|  Flower types shown? |  No (just one combined view) |  Yes (Setosa, Versicolor, Virginica shown) |
|  Shape of data       |  Yes (normal, bimodal, etc.) |  No (just shows min, max, median, etc.)    |
|  Best for?           | Finding overall patterns in one feature                                                                       | Comparing between flower species                                                                                            |