

Code Explanation Request



give me in detail explanation line by line what does it meant and how does it work

Sure! Please upload or paste the code or file you'd like me to explain line by line. Once you do, I'll provide a **detailed explanation for each line**, covering:

- What it means
- How it works
- Why it's used

Go ahead and share the code or notebook file you'd like explained.



```
import pandas as pd
df=pd.read_csv("Iris.csv")
print(df.columns)
df.describe()
# Check for missing values
print("\nMissing values:")
print(df.isnull().sum())
# Get initial statistics
print("\nInitial statistics of 'df':")
print(df.describe(include='all')) #show all columns even the text included
# Variable descriptions and types
print("Information about the DataFrame 'df':")
df.info()
# Check the dimensions of the DataFrame 'df'
print("Dimensions of the DataFrame 'df' (rows, columns):", df.shape)
# Summarize variable types (already done using df.info())
print("\nData types of columns in 'df':")
print(df.dtypes)
#data normalization
from sklearn.preprocessing import MinMaxScaler

# Initialize the MinMaxScaler
scaler = MinMaxScaler()

# Select the numerical columns to normalize
numerical_cols = ['SepalLengthCm', 'SepalWidthCm', 'PetalLengthCm', 'PetalWidthCm']

# Fit the scaler to the numerical data and transform it
df[numerical_cols] = scaler.fit_transform(df[numerical_cols]) #All values in those 4 columns will be
between 0 and 1
```

```
print("\nFirst 5 rows of 'df' after Min-Max normalization:")
print(df.head())
# Turn categorical variables into quantitative variables in Python

# Identify the categorical column
categorical_col = 'Species'

# Perform One-Hot Encoding
df = pd.get_dummies(df, columns=[categorical_col], prefix=categorical_col)

print("\nDataFrame after One-Hot Encoding of 'Species' (first 5 rows):")
print(df.head())
print("\nInformation about the DataFrame after encoding:")
df.info()
print(df.dtypes)
```

Here's a **detailed line-by-line explanation** of your Python code using `pandas`, `scikit-learn`, and data preprocessing techniques:

1. Importing Required Library

```
python

import pandas as pd
```

- **Meaning:** This line imports the `pandas` library, commonly used for data manipulation and analysis.
 - `pd` is a common alias used for convenience.
-

2. Reading the CSV File

```
python

df = pd.read_csv("Iris.csv")
```

- **Meaning:** This reads a CSV file named `Iris.csv` and stores it as a DataFrame named `df`.
 - **How it works:** `read_csv()` parses the CSV and loads the data into a table format (rows and columns).
 - **Assumption:** The file must be present in the same directory or provide the full path.
-

3. Printing Column Names

```
python

print(df.columns)
```

- **Meaning:** Displays the column names in the DataFrame.
- **Why:** Helps understand the structure of the dataset.

4. Initial Summary Statistics (Not Printed)

```
python

df.describe()
```

- **Meaning:** Calculates basic statistics (mean, std, min, max, etc.) for numeric columns.
- **Note:** No `print()` used, so it won't display anything unless in an interactive environment like Jupyter.

5. Checking for Missing Values

```
python

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

- **Meaning:**
 - `df.isnull()` returns a DataFrame of the same shape with `True` for missing (null) values.
 - `.sum()` counts the number of missing values per column.
- **Why:** To check if any data cleaning is needed before analysis.

6. Summary Statistics Including All Columns

```
python

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

- **Meaning:** Shows descriptive stats for all columns, including non-numeric ones (like `Species`).
- `include='all'`: Ensures categorical data is included (e.g., count, unique values, top frequency).

7. DataFrame Information

```
python

print("Information about the DataFrame 'df':")
df.info()
```

- **Meaning:** Prints:
 - Column names

- Non-null counts
- Data types
- Memory usage
- **Why:** To get a quick overview of the dataset.

8. DataFrame Shape (Dimensions)

```
python

print("Dimensions of the DataFrame 'df' (rows, columns):", df.shape)
```

- **Meaning:** Displays the shape as a tuple (number of rows, number of columns).

9. Printing Data Types Again

```
python

print("\nData types of columns in 'df':")
print(df.dtypes)
```

- **Meaning:** Lists each column's data type (e.g., `float64`, `object`, etc.).
- **Why:** Useful for knowing which columns are numeric, categorical, etc.

Data Normalization (Min-Max Scaling)

10. Importing MinMaxScaler

```
python

from sklearn.preprocessing import MinMaxScaler
```

- **Meaning:** Imports a tool to scale numeric data between 0 and 1.

11. Creating Scaler Object

```
python

scaler = MinMaxScaler()
```

- **Meaning:** Creates an instance of the `MinMaxScaler` class.

12. Choosing Numerical Columns

```
python

numerical_cols = ['SepalLengthCm', 'SepalWidthCm', 'PetalLengthCm', 'PetalWidthCm']
```

- **Meaning:** Specifies which columns to normalize.
- **Why:** These columns have numeric values suitable for scaling.

13. Applying Normalization

```
python

df[numerical_cols] = scaler.fit_transform(df[numerical_cols])
```

- **Meaning:**
 - `fit_transform()` scales the values in the selected columns to range between 0 and 1.
 - It replaces the original values with normalized values.
- **Why:** Normalization is important before applying ML algorithms to ensure uniform scale.

14. Displaying First 5 Rows Post Normalization

```
python

print("\nFirst 5 rows of 'df' after Min-Max normalization:")
print(df.head())
```

- **Meaning:** Shows first 5 rows to verify scaling worked as expected.



Categorical Variable Encoding (One-Hot Encoding)

15. Identify the Categorical Column

```
python

categorical_col = 'Species'
```

- **Meaning:** Specifies that the column `'Species'` contains categorical text data.

16. Applying One-Hot Encoding

```
python

df = pd.get_dummies(df, columns=[categorical_col], prefix=categorical_col)
```

- **Meaning:**

- Converts the `Species` column into separate binary columns (like `Species_Iris-setosa`, etc.).
- `1` indicates the presence of a category, `0` means absence.
- **Why:** Machine learning models need numerical input, so this encodes the categories numerically.

17. Display DataFrame After Encoding

python

```
print("\nDataFrame after One-Hot Encoding of 'Species' (first 5 rows):")
print(df.head())
```

- **Meaning:** Shows how the data looks after categorical encoding.

18. Check Info Post-Encoding

python

```
print("\nInformation about the DataFrame after encoding:")
df.info()
print(df.dtypes)
```

- **Meaning:** Rechecks structure and data types after transformations.
- **Why:** Ensures the changes (like additional columns for species) were applied correctly.

✅ Summary of What's Done

- Loaded and explored the Iris dataset.
- Checked for missing values.
- Described the dataset.
- Normalized numerical data.
- Encoded categorical data using One-Hot Encoding.

Would you like me to visualize this data or prepare it for ML next (like splitting into train/test)?



```
import pandas as pd
import numpy as np

# Sample academic performance dataset
data = {
    'Student_ID': [1, 2, 3, 4, 5, 6],
    'Name': ['Alice', 'Bob', 'Charlie', 'David', 'Eva', 'Frank'],
    'Math_Score': [88, 92, np.nan, 45, 104, 56],
    'Science_Score': [72, 75, 80, np.nan, 110, 60],
    'English_Score': [85, 88, 90, 45, 98, 55],
```

```

}

df = pd.DataFrame(data)

print(df.isnull().sum())
df['Math_Score'].fillna(df['Math_Score'].mean(), inplace=True)
df['Science_Score'].fillna(df['Science_Score'].mean(), inplace=True)
df['Math_Score'] = df['Math_Score'].apply(lambda x: 100 if x > 100 else x)
df['Science_Score'] = df['Science_Score'].apply(lambda x: 100 if x > 100 else x)

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

lower = Q1 - 1.5 * IQR
upper = Q3 + 1.5 * IQR

# Detect outliers
outliers = df[(df['English_Score'] < lower) | (df['English_Score'] > upper)]
print(outliers)

df['English_Score'] = np.where(df['English_Score'] > upper, upper,
                             np.where(df['English_Score'] < lower, lower, df['English_Score']))

import seaborn as sns
import matplotlib.pyplot as plt

# Before transformation
sns.histplot(df['Science_Score'], kde=True)
plt.title("Before Log Transformation")
plt.show()

# Apply log transformation
df['Log_Science_Score'] = np.log1p(df['Science_Score'])

# After transformation
sns.histplot(df['Log_Science_Score'], kde=True)
plt.title("After Log Transformation")
plt.show()

```

Here's a **detailed, line-by-line explanation** of your code that demonstrates **data cleaning, outlier handling, and transformation** using Python:

Importing Required Libraries

```

python

import pandas as pd
import numpy as np

```

- `pandas` is used for handling tabular data (DataFrames).
- `numpy` provides mathematical functions and handling of NaN (Not a Number).



Creating a Sample Dataset

python

```
data = {
    'Student_ID': [1, 2, 3, 4, 5, 6],
    'Name': ['Alice', 'Bob', 'Charlie', 'David', 'Eva', 'Frank'],
    'Math_Score': [88, 92, np.nan, 45, 104, 56],
    'Science_Score': [72, 75, 80, np.nan, 110, 60],
    'English_Score': [85, 88, 90, 45, 98, 55],
}
```

- Creates a **dictionary** that simulates a real-world academic dataset.
- Note:
 - `np.nan` indicates **missing values**.
 - One math score and one science score are missing.
 - Some scores (like 104, 110) exceed 100 (invalid or outliers).

python

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

- Converts the dictionary to a **pandas DataFrame**, stored in `df`.



Checking for Missing Values

python

```
print(df.isnull().sum())
```

- `isnull()` returns `True` for NaNs.
- `sum()` counts the number of missing values per column.



Handling Missing Values (Imputation)

python

```
df['Math_Score'].fillna(df['Math_Score'].mean(), inplace=True)
```

- Replaces missing `Math_Score` with the column's mean (average).
- `inplace=True` applies the change directly to the DataFrame.

python

```
df['Science_Score'].fillna(df['Science_Score'].mean(), inplace=True)
```

- Same logic as above, but for `Science_Score`.

Fixing Invalid Data (>100) in Scores

python

```
df['Math_Score'] = df['Math_Score'].apply(lambda x: 100 if x > 100 else x)
```

- Any score above 100 is capped at 100 (maximum valid score).
- `apply()` applies the `lambda` function to each value.

python

```
df['Science_Score'] = df['Science_Score'].apply(lambda x: 100 if x > 100 else x)
```

- Same logic as above, for `Science_Score`.

Outlier Detection (English_Score)

python

```
Q1 = df['English_Score'].quantile(0.25)
Q3 = df['English_Score'].quantile(0.75)
```

- **Q1:** 25th percentile
- **Q3:** 75th percentile

python

```
IQR = Q3 - Q1
```

- **IQR (Interquartile Range)** = Q3 - Q1, used to detect outliers.

python

```
lower = Q1 - 1.5 * IQR
upper = Q3 + 1.5 * IQR
```

- Outlier boundaries: Anything outside `lower` and `upper` is considered an outlier.

Find Outliers

python

```
outliers = df[(df['English_Score'] < lower) | (df['English_Score'] > upper)]
print(outliers)
```

- Filters and prints rows where `English_Score` is an outlier.

Handling Outliers (Clipping)

python

```
df['English_Score'] = np.where(
    df['English_Score'] > upper, upper,
    np.where(df['English_Score'] < lower, lower, df['English_Score'])
)
```

- `np.where()` is a vectorized if-else:
 - If value > upper → set to upper.
 - If value < lower → set to lower.
 - Else → keep original.
- This **clamps** outliers to boundary values.

Visualizing Score Distribution (Before & After Log Transformation)

◆ Before Log Transform

python

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

- `seaborn` and `matplotlib.pyplot` are used for plotting.

python

```
sns.histplot(df['Science_Score'], kde=True)
plt.title("Before Log Transformation")
plt.show()
```

- Creates a **histogram** of `Science_Score` with a KDE (smoothed curve).
- Shows how data is spread (e.g., skewed to the right).

Applying Log Transformation

```
python
```

```
df['Log_Science_Score'] = np.log1p(df['Science_Score'])
```

- Applies **logarithmic transformation** using `log1p(x) = log(x + 1)` to avoid `log(0)`.
- Reduces skewness and makes the distribution closer to normal.

◆ After Log Transform

```
python
```

```
sns.histplot(df['Log_Science_Score'], kde=True)
plt.title("After Log Transformation")
plt.show()
```

- Compares distribution **before vs after** transformation to see improvements.

✓ Summary of Steps Performed:

| Step | Operation |
|------------------------|--|
| Missing value handling | Filled with column mean |
| Data correction | Capped scores >100 to 100 |
| Outlier detection | Used IQR method on English scores |
| Outlier handling | Clipped outliers to limits |
| Visualization | Shown histograms before and after log transformation |
| Feature transformation | Applied log transform to reduce skewness in Science scores |

Would you like help exporting this clean dataset to a CSV or using it for ML modeling?



```
import pandas as pd
import numpy as np
```

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

```
print(iris_df.columns)
petal_length_col = 'PetalLengthCm' # Store the column name in a variable
```

```
if petal_length_col in iris_df.columns: # Check if the column exists
    bins = [0, 3, 5, 7]
    labels = ['Short', 'Medium', 'Long']
    iris_df['petal length category'] = pd.cut(iris_df[petal_length_col], bins=bins, labels=labels,
right=False)
```

```
# Group by 'petal length category' and calculate summary statistics for 'SepalWidthCm'
```

```
grouped_stats = iris_df.groupby('petal length category')['SepalWidthCm'].agg(['mean', 'median',
'min', 'max', 'std'])

print("\nPart 1: Summary Statistics of Sepal Width by Petal Length Category:")
print(grouped_stats)

# Create a list containing the mean sepal width for each petal length category
mean_sepal_width_by_category = grouped_stats['mean'].tolist()
print("\nList of Mean Sepal Width for Each Petal Length Category:",
mean_sepal_width_by_category)
else:
    print(f"\nError: The column '{petal_length_col}' does not exist in the DataFrame. Please check the
column names.")

# --- Part 2: Basic Statistical Details for Iris Species ---
# Get unique species names
species_list = iris_df['Species'].unique() # Changed to 'Species'

# Iterate through each species and calculate statistics
print("\nPart 2: Basic Statistical Details for Each Species:")
for species in species_list:
    species_data = iris_df[iris_df['Species'] == species] # Changed to 'Species'
    print(f"\nStatistics for species: {species}")
    print(species_data.describe(percentiles=[.25, .5, .75]))
```

Here is a **detailed line-by-line explanation** of the code, which performs **categorization**, **grouping**, and **statistical analysis** on the Iris dataset using **Pandas** and **NumPy**:

Step 1: Importing Libraries

```
python

import pandas as pd
import numpy as np
```

- ``pandas``: For data manipulation and analysis.
- ``numpy``: For numerical operations (not directly used here but imported for potential use).

Step 2: Load Dataset

```
python

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

- Reads the ``Iris.csv`` file into a **DataFrame** named ``iris_df``.
- Make sure the CSV file is in the correct working directory or provide full path.

Step 3: Display Column Names

```
python

print(iris_df.columns)
```

- Lists all column names to help verify structure, like `SepalLengthCm`, `PetalLengthCm`, `Species`, etc.

Step 4: Define the Target Column for Binning

```
python

petal_length_col = 'PetalLengthCm' # Store the column name in a variable
```

- Assigns the name `PetalLengthCm` to a variable for easy reuse.

Step 5: Check if Column Exists

```
python

if petal_length_col in iris_df.columns:
```

- Verifies that the column `PetalLengthCm` exists in the DataFrame before proceeding.

Step 6: Create Categories (Binning)

```
python

bins = [0, 3, 5, 7]
labels = ['Short', 'Medium', 'Long']
```

- `bins` define the numeric ranges:
 - $0 \leq x < 3 \rightarrow$ Short
 - $3 \leq x < 5 \rightarrow$ Medium
 - $5 \leq x < 7 \rightarrow$ Long

```
python

iris_df['petal length category'] = pd.cut(
    iris_df[petal_length_col],
    bins=bins,
    labels=labels,
    right=False
)
```

- `pd.cut()` slices `PetalLengthCm` into bins.
- `right=False` makes intervals left-closed (includes left bound, excludes right).

- New column ``petal length category`` is created.

Step 7: Group & Summarize Sepal Width

```
python

grouped_stats = iris_df.groupby('petal length category')['SepalWidthCm'].agg(['mean', 'median', 'mi
```

- Groups data by the ``petal length category``.
- Calculates summary statistics on the ``SepalWidthCm`` column:
 - **mean, median, min, max, std (standard deviation).**

```
python

print("\nPart 1: Summary Statistics of Sepal Width by Petal Length Category:")
print(grouped_stats)
```

- Outputs the grouped statistical summary in a table.

Step 8: Create a List of Mean Sepal Widths

```
python

mean_sepal_width_by_category = grouped_stats['mean'].tolist()
```

- Extracts the mean values into a list (e.g., ``[3.4, 3.1, 2.9]``).

```
python

print("\nList of Mean Sepal Width for Each Petal Length Category:", mean_sepal_width_by_category)
```

- Prints the list of mean values for quick access or later use.

Part 2: Summary Statistics for Each Species

```
python

species_list = iris_df['Species'].unique()
```

- Gets all unique species names in the ``Species`` column (like ``Iris-setosa``, ``Iris-versicolor``, ``Iris-virginica``).

```
python

print("\nPart 2: Basic Statistical Details for Each Species:")
```

- Prints a heading to separate part 2 of the analysis.

🔄 Loop Through Species and Describe

python

```
for species in species_list:
    species_data = iris_df[iris_df['Species'] == species]
```

- Filters data for each species in a loop.

python

```
print(f"\nStatistics for species: {species}")
print(species_data.describe(percentiles=[.25, .5, .75]))
```

- Prints **descriptive statistics** for each species separately:
 - count, mean, std, min, 25%, 50% (median), 75%, max
- `percentiles=[.25, .5, .75]` ensures quartile statistics are included.

✅ Summary of Operations:

| Part | Operation |
|------|--|
| 1 | Binned `PetalLengthCm` into 3 categories |
| | Grouped by category and summarized `SepalWidthCm` statistics |
| | Extracted the mean values into a list |
| 2 | Printed descriptive statistics for each unique species |

Would you like a plot (bar chart or box plot) for visualizing the grouped stats or petal categories?



```
# Import necessary libraries
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error, r2_score
from sklearn.impute import SimpleImputer
import matplotlib.pyplot as plt
import seaborn as sns

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

# Display the first few rows of the dataframe to understand its structure
print("First 5 rows of the Boston Housing dataset:")
print(df.head())
# Display basic information about the dataset
print("\nInformation about the dataset:")
df.info()
```

```
# Display descriptive statistics of the dataset
print("\nDescriptive statistics of the dataset:")
print(df.describe())

# Print the names of all columns in your DataFrame to identify the target variable
print("Columns in your DataFrame:")
print(df.columns)
# Define features (X) and target (y) based on your column names
target_column = 'MEDV'
X = df.drop(target_column, axis=1)
y = df[target_column]
# Split the dataset into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Print the shapes of the training and testing sets to verify the split
print("\nShape of X_train:", X_train.shape)
print("Shape of X_test:", X_test.shape)
print("Shape of y_train:", y_train.shape)
print("Shape of y_test:", y_test.shape)

# Handle missing values using SimpleImputer (filling with the mean)
imputer = SimpleImputer(strategy='mean')
X_train_imputed = imputer.fit_transform(X_train)
X_test_imputed = imputer.transform(X_test)

# Create a Linear Regression model
model = LinearRegression()

# Train the model using the imputed training data
model.fit(X_train_imputed, y_train)

# Make predictions on the imputed test data
y_pred = model.predict(X_test_imputed)

# Evaluate the model
mse = mean_squared_error(y_test, y_pred)
print("\nMean Squared Error:", mse)
r2 = r2_score(y_test, y_pred)
print("R-squared:", r2)

# Print the coefficients of the linear regression model
print("\nCoefficients of the Linear Regression model:")
print(pd.DataFrame(model.coef_, X.columns, columns=['Coefficient']))

# Print the intercept of the Linear Regression model
print("\nIntercept of the Linear Regression model:", model.intercept_)
plt.figure(figsize=(8, 6))
sns.scatterplot(x=y_test, y=y_pred)
plt.xlabel("Actual House Prices (MEDV)")
plt.ylabel("Predicted House Prices (MEDV)")
```



```
plt.title("Actual vs. Predicted House Prices")
plt.plot([y_test.min(), y_test.max()], [y_test.min(), y_test.max()])
plt.grid(True)
plt.show()
```

Here is a **detailed line-by-line explanation** of your code, which implements **linear regression** on the Boston housing dataset, including steps for **data preprocessing**, **model training**, **evaluation**, and **visualization**.

Import Libraries

```
python

import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error, r2_score
from sklearn.impute import SimpleImputer
import matplotlib.pyplot as plt
import seaborn as sns
```

- ``pandas``: For data manipulation and analysis.
- ``train_test_split``: For splitting data into training and test sets.
- ``LinearRegression``: For performing linear regression.
- ``mean_squared_error``, ``r2_score``: For model evaluation.
- ``SimpleImputer``: For handling missing data by imputation.
- ``matplotlib.pyplot``, ``seaborn``: For plotting and visualizations.

Load the Dataset

```
python

df = pd.read_csv('housing.csv')
```

- Reads the housing data from a CSV file into a pandas **DataFrame** called ``df``.

Explore the Dataset

```
python

print("First 5 rows of the Boston Housing dataset:")
print(df.head())
```

- Displays the first 5 rows of the dataset to get an overview of the structure and data values.

python

```
print("\nInformation about the dataset:")
df.info()
```

- `df.info()`: Provides information about the dataset such as number of entries, columns, and non-null values.

python

```
print("\nDescriptive statistics of the dataset:")
print(df.describe())
```

- `df.describe()`: Displays basic statistics for numerical columns such as mean, standard deviation, min, max, and percentiles.

python

```
print("Columns in your DataFrame:")
print(df.columns)
```

- Prints the column names in the dataset to identify the target variable and features.

Define Target and Features

python

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

- `target_column = 'MEDV'`: Defines the target variable `'MEDV'` (Median value of owner-occupied homes) which we want to predict.
- `X = df.drop(target_column, axis=1)`: Drops the target column from `df` to create the feature set `X`.
- `y = df[target_column]`: Assigns the target variable (`'MEDV'`) to `y`.

Split the Data into Training and Testing Sets

python

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

- `train_test_split()` splits the data into:
 - 80% for training (`X_train`, `y_train`)
 - 20% for testing (`X_test`, `y_test`)
- `random_state=42` ensures reproducibility of results by fixing the seed for the random split.

python

```
print("\nShape of X_train:", X_train.shape)
print("Shape of X_test:", X_test.shape)
print("Shape of y_train:", y_train.shape)
print("Shape of y_test:", y_test.shape)
```

- Displays the shapes of the training and testing sets to verify the split.

Handle Missing Values (Imputation)

python

```
imputer = SimpleImputer(strategy='mean')
X_train_imputed = imputer.fit_transform(X_train)
X_test_imputed = imputer.transform(X_test)
```

- `SimpleImputer(strategy='mean')`: Imputes missing values by replacing them with the **mean** of the respective column.
- `fit_transform()`: Fits the imputer to the training data and transforms it.
- `transform()`: Applies the same transformation (imputation) to the test data.



Train the Model

python

```
model = LinearRegression()
```

- Creates a **Linear Regression** model.

python

```
model.fit(X_train_imputed, y_train)
```

- `model.fit()`: Trains the linear regression model using the imputed training data (`X_train_imputed`) and target variable (`y_train`).



Make Predictions

python

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

- `model.predict()`: Predicts the target variable (`y_pred`) using the imputed test data (`X_test_imputed`).



Evaluate the Model

```
python
```

```
mse = mean_squared_error(y_test, y_pred)
print("\nMean Squared Error:", mse)
```

- `mean_squared_error()`: Calculates the **Mean Squared Error (MSE)**, which measures the average squared difference between actual and predicted values.
- Lower MSE means better model performance.

```
python
```

```
r2 = r2_score(y_test, y_pred)
print("R-squared:", r2)
```

- `r2_score()`: Calculates the **R-squared** value, which indicates how well the model explains the variance in the data.
 - Values closer to 1 indicate better performance.



Display Model Coefficients

```
python
```

```
print("\nCoefficients of the Linear Regression model:")
print(pd.DataFrame(model.coef_, X.columns, columns=['Coefficient']))
```

- Prints the **coefficients** of the linear regression model.
- These represent the impact of each feature on the target variable (MEDV).



Display Model Intercept

```
python
```

```
print("\nIntercept of the Linear Regression model:", model.intercept_)
```

- Prints the **intercept** of the linear regression model, which is the predicted value when all features are 0.



Visualize Actual vs Predicted Values

```
python
```

```
plt.figure(figsize=(8, 6))
sns.scatterplot(x=y_test, y=y_pred)
plt.xlabel("Actual House Prices (MEDV)")
plt.ylabel("Predicted House Prices (MEDV)")
plt.title("Actual vs. Predicted House Prices")
```

- Creates a scatter plot comparing actual vs. predicted house prices.

- `sns.scatterplot()` creates the scatter plot with `y_test` (actual) on the x-axis and `y_pred` (predicted) on the y-axis.

python

```
plt.plot([y_test.min(), y_test.max()], [y_test.min(), y_test.max()])
```

- Adds a **line of perfect predictions** (diagonal line) to the plot, where the predicted values exactly match the actual values.

python

```
plt.grid(True)
plt.show()
```

- Adds a grid to the plot for better readability and displays it.

✓ Summary of Key Steps:

1. **Data Loading:** Loaded the housing dataset into a DataFrame.
2. **Preprocessing:** Split the data into training and testing sets, handled missing values using imputation.
3. **Model Training:** Trained a Linear Regression model on the training data.
4. **Model Evaluation:** Evaluated the model using MSE and R-squared.
5. **Model Coefficients:** Displayed the coefficients and intercept of the trained model.
6. **Visualization:** Created a scatter plot comparing actual vs. predicted values.

Would you like additional details or suggestions for improving the model's performance (e.g., feature scaling, feature selection, or regularization)?



```
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.model_selection import train_test_split
from sklearn.metrics import confusion_matrix
from sklearn.linear_model import LogisticRegression
# Load the dataset
df = pd.read_csv('Social_Network_Ads.csv')
df.head()
df.shape

df.info()
df.describe()
df.isna().sum()
X = df[['Age', 'EstimatedSalary']] # Features
```

```

y = df['Purchased']          # Target

# Split Dataset into Train and Test Sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=0)

# Initialize and train the Logistic Regression model
model = LogisticRegression()
model.fit(X_train, y_train)

# Make predictions on the test set
y_pred = model.predict(X_test)

# Compute the confusion matrix
cm = confusion_matrix(y_test, y_pred)
TN, FP, FN, TP = cm.ravel()

print("Confusion Matrix:")
print(cm)
print(f"TP: {TP}, FP: {FP}, TN: {TN}, FN: {FN}")

# Visualize the confusion matrix
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues',
            xticklabels=['Predicted 0', 'Predicted 1'],
            yticklabels=['Actual 0', 'Actual 1'])
plt.xlabel('Predicted')
plt.ylabel('Actual')
plt.title('Confusion Matrix')
plt.show()

# Extract TP, FP, TN, FN
accuracy = (TP + TN) / (TP + TN + FP + FN)
error_rate = 1 - accuracy
precision = TP / (TP + FP)
recall = TP / (TP + FN)

print(f"Accuracy: {accuracy:.2f}")
print(f"Error Rate: {error_rate:.2f}")
print(f"Precision: {precision:.2f}")
print(f"Recall: {recall:.2f}")

# Compute evaluation metrics
accuracy = (TP + TN) / (TP + TN + FP + FN)
error_rate = 1 - accuracy
precision = TP / (TP + FP)
recall = TP / (TP + FN)

print(f"Accuracy: {accuracy:.2f}")
print(f"Error Rate: {error_rate:.2f}")

```

```
print(f"Precision: {precision:.2f}")
print(f"Recall: {recall:.2f}")
```

Here's a **line-by-line breakdown** of your code that implements **Logistic Regression** on the **Social Network Ads dataset** to predict whether a user purchased a product based on their age and estimated salary. The code evaluates the model's performance using a **confusion matrix** and calculates various performance metrics.

Import Libraries

```
python

import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.model_selection import train_test_split
from sklearn.metrics import confusion_matrix
from sklearn.linear_model import LogisticRegression
```

- ``pandas``: For data manipulation and analysis.
- ``matplotlib.pyplot`` and ``seaborn``: For creating plots and visualizations.
- ``train_test_split``: For splitting the dataset into training and test sets.
- ``confusion_matrix``: For computing the confusion matrix to evaluate model performance.
- ``LogisticRegression``: For performing logistic regression (classification).

Load the Dataset

```
python

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

- Reads the **Social_Network_Ads.csv** file into a pandas DataFrame (``df``).
- ``df.head()``: Displays the first five rows of the dataset to give an overview.

```
python

df.shape
```

- ``df.shape``: Returns the dimensions of the dataset (rows, columns), helping to check the dataset's size.

```
python

df.info()
```

- ``df.info()``: Displays basic information about the DataFrame, such as the number of non-null entries and data types of the columns.

```
python

df.describe()
```

- `df.describe()`: Generates descriptive statistics of the numerical columns in the dataset (like mean, std, min, and max).

```
python

df.isna().sum()
```

- `df.isna().sum()`: Checks for missing values in each column. It returns the count of `NaN` values for each column.

Define Features and Target

```
python

X = df[['Age', 'EstimatedSalary']] # Features
y = df['Purchased']                # Target
```

- `X`: Contains the features or independent variables ('Age' and 'EstimatedSalary').
- `y`: Contains the target variable or dependent variable ('Purchased'), which indicates whether the user made a purchase (1 for yes, 0 for no).

Split the Data into Training and Testing Sets

```
python

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=0)
```

- `train_test_split()`: Splits the data into training (70%) and testing (30%) sets.
 - `test_size=0.3` means 30% of the data will be used for testing.
 - `random_state=0` ensures that the split is reproducible each time the code is run.

Initialize and Train the Logistic Regression Model

```
python

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

- `LogisticRegression()`: Initializes the logistic regression model.
- `model.fit(X_train, y_train)`: Trains the model using the training features (`X_train`) and the target variable (`y_train`).

Make Predictions

```
python

y_pred = model.predict(X_test)
```

- `model.predict(X_test)`: Makes predictions (`y_pred`) on the test dataset (`X_test`) based on the trained logistic regression model.

Compute the Confusion Matrix

```
python

cm = confusion_matrix(y_test, y_pred)
TN, FP, FN, TP = cm.ravel()
```

- `confusion_matrix(y_test, y_pred)`: Calculates the confusion matrix, which compares the actual values (`y_test`) with the predicted values (`y_pred`).
- `cm.ravel()`: Flattens the confusion matrix into a 1D array, which is then unpacked into:
 - **TN**: True negatives (actual 0, predicted 0)
 - **FP**: False positives (actual 0, predicted 1)
 - **FN**: False negatives (actual 1, predicted 0)
 - **TP**: True positives (actual 1, predicted 1)

```
python

print("Confusion Matrix:")
print(cm)
print(f"TP: {TP}, FP: {FP}, TN: {TN}, FN: {FN}")
```

- Prints the confusion matrix and the individual counts of TP, FP, TN, FN.

Visualize the Confusion Matrix

```
python

sns.heatmap(cm, annot=True, fmt='d', cmap='Blues',
            xticklabels=['Predicted 0', 'Predicted 1'],
            yticklabels=['Actual 0', 'Actual 1'])
plt.xlabel('Predicted')
plt.ylabel('Actual')
plt.title('Confusion Matrix')
plt.show()
```

- `sns.heatmap()`: Visualizes the confusion matrix as a heatmap with annotations (`annot=True`) to display the counts.
 - `fmt='d'` formats the annotations as integers.
 - `cmap='Blues'` sets the color map for the heatmap.

- Customizes x and y labels to show "Predicted 0" and "Predicted 1", as well as "Actual 0" and "Actual 1".
- `plt.show()`: Displays the plot.

Compute and Print Evaluation Metrics

python

```
accuracy = (TP + TN) / (TP + TN + FP + FN)
error_rate = 1 - accuracy
precision = TP / (TP + FP)
recall = TP / (TP + FN)

print(f"Accuracy: {accuracy:.2f}")
print(f"Error Rate: {error_rate:.2f}")
print(f"Precision: {precision:.2f}")
print(f"Recall: {recall:.2f}")
```

- `accuracy`: Proportion of correct predictions (both true positives and true negatives) out of all predictions.
 - **Formula:** $(TP + TN) / (TP + TN + FP + FN)$
- `error_rate`: Proportion of incorrect predictions, calculated as `1 - accuracy`.
- `precision`: Proportion of true positives out of all predicted positives.
 - **Formula:** $TP / (TP + FP)$
- `recall`: Proportion of true positives out of all actual positives.
 - **Formula:** $TP / (TP + FN)$
- Prints the calculated metrics: accuracy, error rate, precision, and recall.

Summary of Key Steps:

1. **Data Loading:** Loaded the dataset from the CSV file into a DataFrame.
2. **Preprocessing:** Defined features (`x`) and target (`y`), then split the dataset into training and testing sets.
3. **Model Training:** Trained a logistic regression model using the training data.
4. **Predictions:** Made predictions using the test data.
5. **Confusion Matrix:** Calculated and visualized the confusion matrix.
6. **Model Evaluation:** Computed accuracy, error rate, precision, and recall.

This is a solid implementation of **logistic regression** for binary classification. Would you like to explore hyperparameter tuning or try other models for better performance?



```
import pandas as pd
from sklearn.model_selection import train_test_split
```

```

from sklearn.naive_bayes import GaussianNB
from sklearn.metrics import confusion_matrix, accuracy_score, precision_score, recall_score
import matplotlib.pyplot as plt
import seaborn as sns

# Load the iris dataset
df = pd.read_csv('iris.csv')

df.keys()
df.head()
df.info()
# Separate features (X) and target (y)
X = data.drop('Species', axis=1)
y = data['Species']
# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)

# Initialize the Gaussian Naïve Bayes classifier
model = GaussianNB()

# Train the model
model.fit(X_train, y_train)

# Make predictions on the test set
y_pred = model.predict(X_test)

# Compute the confusion matrix
cm = confusion_matrix(y_test, y_pred)
print("Confusion Matrix:\n", cm)

# Extract TP, FP, TN, FN (for a multi-class problem, these are defined per class)
# Let's calculate them for each class:
classes = data['Species'].unique()
for i, cls in enumerate(classes):
    tp = cm[i, i]
    fp = cm[:, i].sum() - tp
    fn = cm[i, :].sum() - tp
    tn = cm.sum() - tp - fp - fn
    print(f"\nMetrics for class '{cls}':")
    print(f" TP: {tp}")
    print(f" FP: {fp}")
    print(f" TN: {tn}")
    print(f" FN: {fn}")
# Compute accuracy
accuracy = accuracy_score(y_test, y_pred)
print("\nAccuracy:", accuracy)

# Compute error rate
error_rate = 1 - accuracy
print("Error Rate:", error_rate)

```

```
# Compute precision
precision = precision_score(y_test, y_pred, average='weighted')
print("Precision:", precision)

# Compute recall
recall = recall_score(y_test, y_pred, average='weighted')
print("Recall:", recall)
# Visualize the confusion matrix for all three classes
plt.figure(figsize=(6, 5))
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues',
            xticklabels=data['Species'].unique(),
            yticklabels=data['Species'].unique())
plt.xlabel('Predicted')
plt.ylabel('Actual')
plt.title('Confusion Matrix (All Classes)')
plt.show()
```

Your code implements a **Gaussian Naïve Bayes** classification model using the **Iris dataset** to predict the species of Iris plants. It includes steps like splitting the data, training the model, making predictions, evaluating the model using metrics such as **accuracy**, **precision**, **recall**, and visualizing the **confusion matrix**. Here's a breakdown of the key steps:

Import Libraries

```
python

import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.naive_bayes import GaussianNB
from sklearn.metrics import confusion_matrix, accuracy_score, precision_score, recall_score
import matplotlib.pyplot as plt
import seaborn as sns
```

- ``pandas``: For data manipulation and reading the dataset.
- ``train_test_split``: For splitting the dataset into training and testing sets.
- ``GaussianNB``: The Gaussian Naïve Bayes classifier.
- ``confusion_matrix``, ``accuracy_score``, ``precision_score``, ``recall_score``: Evaluation metrics.
- ``matplotlib.pyplot`` and ``seaborn``: For visualizations, including the confusion matrix.

Load the Dataset

```
python

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

df.keys()
df.head()
df.info()
```

- `pd.read_csv('iris.csv')`: Loads the Iris dataset into a pandas DataFrame (`df`).
- `df.keys()`: Prints the column names of the dataset.
- `df.head()`: Displays the first five rows of the dataset.
- `df.info()`: Prints a summary of the DataFrame, such as data types and non-null counts.

Define Features and Target

python

```
X = data.drop('Species', axis=1)
y = data['Species']
```

- `X`: Features (all columns except the target `Species`).
- `y`: Target (the `Species` column).

Split the Data into Training and Testing Sets

python

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)
```

- `train_test_split()`: Splits the dataset into training (70%) and testing (30%) sets, with a `random_state=42` for reproducibility.

Initialize and Train the Model

python

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

- `GaussianNB()`: Initializes the Gaussian Naïve Bayes classifier.
- `model.fit(X_train, y_train)`: Trains the model on the training data (`X_train` and `y_train`).

Make Predictions

python

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

- `model.predict(X_test)`: Makes predictions on the test set (`X_test`).

Compute the Confusion Matrix

python

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

- `confusion_matrix(y_test, y_pred)`: Computes the confusion matrix, showing the comparison between the true and predicted values for each class.



Extract Metrics for Each Class

python

```
classes = data['Species'].unique()
for i, cls in enumerate(classes):
    tp = cm[i, i]
    fp = cm[:, i].sum() - tp
    fn = cm[i, :].sum() - tp
    tn = cm.sum() - tp - fp - fn
    print(f"\nMetrics for class '{cls}':")
    print(f"  TP: {tp}")
    print(f"  FP: {fp}")
    print(f"  TN: {tn}")
    print(f"  FN: {fn}")
```

- `classes = data['Species'].unique()`: Gets the unique class labels (species).
- `for i, cls in enumerate(classes):`: Loops through each class to compute the **True Positive (TP)**, **False Positive (FP)**, **True Negative (TN)**, and **False Negative (FN)** for each class.
- These metrics are printed for each class individually.



Compute and Print Overall Evaluation Metrics

python

```
accuracy = accuracy_score(y_test, y_pred)
print("\nAccuracy:", accuracy)

error_rate = 1 - accuracy
print("Error Rate:", error_rate)

precision = precision_score(y_test, y_pred, average='weighted')
print("Precision:", precision)

recall = recall_score(y_test, y_pred, average='weighted')
print("Recall:", recall)
```

- `accuracy_score()`: Calculates the overall accuracy of the model.
- `precision_score()`: Calculates the precision (weighted average across all classes).
- `recall_score()`: Calculates the recall (weighted average across all classes).
- `error_rate`: The complement of accuracy (`1 - accuracy`).



Visualize the Confusion Matrix

```
python

plt.figure(figsize=(6, 5))
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues',
            xticklabels=data['Species'].unique(),
            yticklabels=data['Species'].unique())
plt.xlabel('Predicted')
plt.ylabel('Actual')
plt.title('Confusion Matrix (All Classes)')
plt.show()
```

- `sns.heatmap()`: Plots the confusion matrix as a heatmap with annotations for better visualization.
 - `annot=True` displays the values in the cells.
 - `fmt='d'` formats the annotations as integers.
 - `cmap='Blues'` sets the color scheme of the heatmap.
- `plt.show()`: Displays the heatmap.

Summary of Key Steps:

1. **Data Loading**: Loaded the Iris dataset.
2. **Preprocessing**: Defined the features (`X`) and target (`y`), then split the dataset.
3. **Model Training**: Trained a Gaussian Naïve Bayes model.
4. **Predictions**: Made predictions on the test set.
5. **Confusion Matrix**: Calculated and visualized the confusion matrix.
6. **Model Evaluation**: Computed accuracy, error rate, precision, and recall for the model.

Potential Issue:

It seems like there is a minor mistake:

- `data` is used in the code instead of `df` for creating features (`X`) and target (`y`). Replace `data` with `df` for consistency. Here's the fix:

```
python

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

Would you like further clarification or improvements in the code?



```
import nltk
from nltk.tokenize import word_tokenize
from nltk.corpus import stopwords
from nltk.stem import PorterStemmer, WordNetLemmatizer
from collections import Counter
```

```
import math
# Download necessary NLTK data (run this once)
nltk.download('punkt')
nltk.download('averaged_perceptron_tagger')
nltk.download('stopwords')
nltk.download('wordnet')
nltk.download('omw-1.4')

# Sample Document
document = "The quick brown foxes are jumping over the lazy dogs. The dogs are barking loudly at the playful foxes."

# 1. Tokenization
tokens = word_tokenize(document)
print("Tokens:", tokens)
```

The code you've shared performs the **tokenization** of a given text document using the **Natural Language Toolkit (NLTK)** in Python. Tokenization is the process of splitting a text into individual words or tokens. Below is a breakdown of the code:

Breakdown of the Code:

1. Import Libraries and Download Necessary Data:

```
python

import nltk
from nltk.tokenize import word_tokenize
from nltk.corpus import stopwords
from nltk.stem import PorterStemmer, WordNetLemmatizer
from collections import Counter
import math
```

- ``nltk``: The library that provides tools for natural language processing.
- ``word_tokenize``: Tokenizes the input text into individual words.
- ``stopwords``: A collection of common words (e.g., "the", "is") that are usually removed in text processing.
- ``PorterStemmer`` and ``WordNetLemmatizer``: Used for stemming and lemmatization (not used in the given code, but imported for future tasks).
- ``Counter``: A dictionary subclass used to count occurrences of words.
- ``math``: Used for mathematical operations (though not used yet).

2. Download Required NLTK Data:

```
python

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



```
nlTK.download('wordnet')
nlTK.download('omw-1.4')
```

- ``punkt``: Tokenizer models needed for ``word_tokenize``.
- ``averaged_perceptron_tagger``: For part-of-speech tagging.
- ``stopwords``: Common stop words like "the", "a", etc.
- ``wordnet``: The WordNet lexical database used for lemmatization.
- ``omw-1.4``: WordNet Open Multilingual Wordnet, needed for lemmatization in different languages.

3. Sample Document:

```
python

document = "The quick brown foxes are jumping over the lazy dogs. The dogs are barking loudly at the"
```

- The document contains a simple sentence about foxes and dogs.

4. Tokenization:

```
python

tokens = word_tokenize(document)
print("Tokens:", tokens)
```

- ``word_tokenize`` splits the document into individual tokens (words and punctuation marks).
- This will output a list of tokens (words) in the document.

Output:

The output of the ``word_tokenize`` will look like this:

```
python

Tokens: ['The', 'quick', 'brown', 'foxes', 'are', 'jumping', 'over', 'the', 'lazy', 'dogs', '.', 'Ti']
```

These tokens can be further processed for tasks like stop word removal, stemming, or lemmatization.

Would you like to proceed with further text processing tasks like stop word removal or stemming/lemmatization?



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

# Load the Titanic dataset from the CSV file
df = pd.read_csv('titanic-dataset.csv')
```

```
df.head()
df.info()
df.describe()
print(df.isnull().sum())
# Handle missing values (Impute 'Age' with the median and drop others for simplicity)
df['Age'].fillna(df['Age'].median(), inplace=True)
df.dropna(subset=['Embarked'], inplace=True) # Remove rows where 'Embarked' is missing

# Check for missing values after handling them
print(df.isnull().sum())
df = df.drop("Cabin", axis=1)
print(df.isnull().sum())
# Check if there are any numerical columns
numerical_columns = df.select_dtypes(include=['number']).columns
print("Numerical Columns:", numerical_columns)
# Select numerical columns and calculate correlation
numerical_df = df.select_dtypes(include=['number'])
correlation_matrix = numerical_df.corr()
print("\nCorrelation Matrix:\n", correlation_matrix)

# Example: Count of survivors by gender
sns.countplot(x='Sex', hue='Survived', data=df)
plt.title("Survival Count by Gender")
plt.show()

sns.countplot(x='Pclass', hue='Survived', data=df)
plt.title("Survival Count by Passenger Class")
plt.xlabel("Passenger Class")
plt.ylabel("Count")
plt.show()

sns.histplot(data=df, x='Age', hue='Survived', multiple='stack')
plt.title("Age Distribution by Survival")
plt.xlabel("Age")
plt.ylabel("Count")
plt.show()

# Plot histogram for fare
sns.histplot(df['Fare'], kde=True)
plt.title("Fare Distribution")
plt.xlabel("Ticket Fare")
plt.ylabel("Number of Passengers")
plt.show()
```

Here's a detailed line-by-line explanation of the code provided for processing the Titanic dataset and visualizing key insights:

1. Import Necessary Libraries:

```
python
```

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

- `pandas`: Used for data manipulation, loading, and analysis.
- `numpy`: Provides support for numerical operations (though not directly used in this code).
- `seaborn`: A statistical data visualization library built on top of `matplotlib`, ideal for creating informative and attractive plots.
- `matplotlib.pyplot`: A plotting library used for creating static, animated, and interactive visualizations in Python.

2. Load the Titanic Dataset:

```
python
```

```
df = pd.read_csv('titanic-dataset.csv')
df.head()
df.info()
df.describe()
```

- `pd.read_csv('titanic-dataset.csv')`: Loads the Titanic dataset from a CSV file into a pandas DataFrame called `df`.
- `df.head()`: Displays the first 5 rows of the dataset to get a preview of the data.
- `df.info()`: Provides information about the dataset, such as the number of entries, column names, and data types.
- `df.describe()`: Displays basic statistics for numerical columns, like mean, standard deviation, min, max, etc.

3. Check for Missing Values:

```
python
```

```
print(df.isnull().sum())
```

- `df.isnull().sum()`: Checks for missing values (NaN) in the DataFrame and sums up the count of missing values per column.

4. Handle Missing Values:

```
python
```

```
df['Age'].fillna(df['Age'].median(), inplace=True)
df.dropna(subset=['Embarked'], inplace=True) # Remove rows where 'Embarked' is missing
```

- `df['Age'].fillna(df['Age'].median(), inplace=True)`: Fills missing values in the `Age` column with the median age of the dataset.

- `df.dropna(subset=['Embarked'], inplace=True)`: Drops rows where the `Embarked` column has missing values.

5. Recheck Missing Values After Handling:

```
python

print(df.isnull().sum())
```

- `df.isnull().sum()`: Rechecks if there are any missing values after the imputation and removal steps.

6. Drop the `Cabin` Column:

```
python

df = df.drop("Cabin", axis=1)
print(df.isnull().sum())
```

- `df.drop("Cabin", axis=1)`: Drops the `Cabin` column as it likely has too many missing values and would not be useful for analysis.
- `df.isnull().sum()`: Verifies that the `Cabin` column is dropped and checks for any remaining missing values.

7. Check Numerical Columns:

```
python

numerical_columns = df.select_dtypes(include=['number']).columns
print("Numerical Columns:", numerical_columns)
```

- `df.select_dtypes(include=['number']).columns`: Selects columns with numeric data types and stores them in `numerical_columns`. This is useful for performing statistical operations like correlation.
- `print("Numerical Columns:", numerical_columns)`: Displays the numerical columns in the dataset.

8. Correlation Matrix:

```
python

numerical_df = df.select_dtypes(include=['number'])
correlation_matrix = numerical_df.corr()
print("\nCorrelation Matrix:\n", correlation_matrix)
```

- `df.select_dtypes(include=['number'])`: Selects the numerical columns from the dataset.
- `correlation_matrix = numerical_df.corr()`: Computes the pairwise correlation of numerical features in the dataset.
- `print("\nCorrelation Matrix:\n", correlation_matrix)`: Displays the correlation matrix, showing how features are correlated with each other.

9. Countplot: Survival Count by Gender:

python

```
sns.countplot(x='Sex', hue='Survived', data=df)
plt.title("Survival Count by Gender")
plt.show()
```

- `sns.countplot(x='Sex', hue='Survived', data=df)`: Creates a count plot showing the number of survivors (`Survived`) based on gender (`Sex`), with different colors for survivors and non-survivors.
- `plt.title("Survival Count by Gender")`: Sets the title of the plot.
- `plt.show()`: Displays the plot.

10. Countplot: Survival Count by Passenger Class:

python

```
sns.countplot(x='Pclass', hue='Survived', data=df)
plt.title("Survival Count by Passenger Class")
plt.xlabel("Passenger Class")
plt.ylabel("Count")
plt.show()
```

- `sns.countplot(x='Pclass', hue='Survived', data=df)`: Creates a count plot showing the number of survivors based on the passenger class (`Pclass`).
- `plt.xlabel("Passenger Class")` and `plt.ylabel("Count")`: Label the x and y axes.
- `plt.show()`: Displays the plot.

11. Histogram: Age Distribution by Survival:

python

```
sns.histplot(data=df, x='Age', hue='Survived', multiple='stack')
plt.title("Age Distribution by Survival")
plt.xlabel("Age")
plt.ylabel("Count")
plt.show()
```

- `sns.histplot(data=df, x='Age', hue='Survived', multiple='stack')`: Creates a stacked histogram showing the age distribution of survivors and non-survivors. The `multiple='stack'` option stacks the bars for each survival status.
- `plt.title("Age Distribution by Survival")`: Sets the title of the plot.
- `plt.show()`: Displays the plot.

12. Histogram: Fare Distribution:

python

```
sns.histplot(df['Fare'], kde=True)
plt.title("Fare Distribution")
```

```
plt.xlabel("Ticket Fare")
plt.ylabel("Number of Passengers")
plt.show()
```

- `sns.histplot(df['Fare'], kde=True)`: Creates a histogram for the `Fare` column, with a kernel density estimate (KDE) overlaid to show the distribution of ticket fares.
- `plt.title("Fare Distribution")`: Sets the title of the plot.
- `plt.show()`: Displays the plot.

Summary:

The code above processes the Titanic dataset by handling missing values, performing some basic exploratory data analysis (EDA), and visualizing key insights:

- It fills missing values in the `Age` column, drops rows with missing `Embarked` values, and removes the `Cabin` column.
- It computes a correlation matrix for numerical columns.
- Several visualizations are generated:
 - Survival count by gender.
 - Survival count by passenger class.
 - Age distribution by survival status.
 - Fare distribution.

These steps provide valuable insights into the Titanic dataset and can serve as a foundation for further analysis and modeling. Let me know if you'd like to dive deeper into any specific aspect!



```
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
df = pd.read_csv('titanic-dataset.csv')
df.head()
df.info()
print(df.isnull().sum())
# Handle missing 'Age' values by filling with the median
df['Age'].fillna(df['Age'].median(), inplace=True)
df["Age"] = df["Age"].fillna(df["Age"].mean())
df.isna().sum()
# Plot the box 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.legend(title='Survived', labels=['No', 'Yes'])
plt.show()
```

```
# Observations:
print("\nObservations:")
print("- The box plot shows the distribution of age for males and females, further segmented by whether they survived or not.")
print("- By comparing the boxes and the median lines within each gender group, we can infer potential differences in age distribution between survivors and non-survivors for each sex.")
print("- The 'hue' parameter ('Survived') allows for a direct visual comparison of age distribution within each gender based on survival status.")
print("- Outliers in age for each group are also visible as individual points beyond the whiskers of the box plots.")
print("- Further statistical analysis (e.g., t-tests) could be performed to confirm if the observed differences in age distributions are statistically significant.")
```

The code provided reads and processes the Titanic dataset, handles missing values for the `Age` column, and visualizes the distribution of age by gender and survival status using a box plot. Here's a detailed explanation of each step:

1. Import Libraries:

```
python

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

- `pandas`: For data manipulation and loading the dataset.
- `seaborn`: For creating statistical plots.
- `matplotlib.pyplot`: For creating basic plots (like box plots).

2. Load Titanic Dataset:

```
python

df = pd.read_csv('titanic-dataset.csv')
df.head()
df.info()
```

- `pd.read_csv('titanic-dataset.csv')`: Loads the Titanic dataset into a pandas DataFrame `df`.
- `df.head()`: Displays the first five rows of the dataset to get an initial look at the data.
- `df.info()`: Provides information about the dataset, including the number of non-null values, column names, and data types.

3. Check for Missing Values:

```
python

print(df.isnull().sum())
```

- `df.isnull().sum()`: Checks for missing values in each column and prints the count of missing values per column.

4. Handle Missing Values in the `Age` Column:

python

```
df['Age'].fillna(df['Age'].median(), inplace=True)
df["Age"] = df["Age"].fillna(df["Age"].mean())
df.isna().sum()
```

- `df['Age'].fillna(df['Age'].median(), inplace=True)`: Fills any missing values in the `Age` column with the median value of the column.
- `df["Age"] = df["Age"].fillna(df["Age"].mean())`: This is redundant since the previous line already filled missing values with the median, but it attempts to fill any remaining missing `Age` values with the mean.
- `df.isna().sum()`: Verifies if there are any missing values left after the imputation steps.

5. Plotting the Boxplot:

python

```
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.legend(title='Survived', labels=['No', 'Yes'])
plt.show()
```

- `plt.figure(figsize=(8, 6))`: Sets the size of the plot.
- `sns.boxplot(x='Sex', y='Age', hue='Survived', data=df)`: Creates a boxplot that shows the distribution of `Age` by gender (`Sex`) and survival status (`Survived`). The `hue='Survived'` parameter segments the data by survival status and colors them accordingly.
- `plt.title('Age Distribution by Gender and Survival')`: Sets the title of the plot.
- `plt.xlabel('Gender')` and `plt.ylabel('Age')`: Labels the x and y axes.
- `plt.legend(title='Survived', labels=['No', 'Yes'])`: Adds a legend to the plot to indicate what the color coding for survival represents (`No` for non-survivors, `Yes` for survivors).
- `plt.show()`: Displays the plot.

6. Observations:

python

```
print("\nObservations:")
print("- The box plot shows the distribution of age for males and females, further segmented by whether they survived or not.")
print("- By comparing the boxes and the median lines within each gender group, we can infer potential differences in age distribution between survivors and non-survivors.")
print("- The 'hue' parameter ('Survived') allows for a direct visual comparison of age distribution between survivors and non-survivors.")
print("- Outliers in age for each group are also visible as individual points beyond the whiskers of the boxplot.")
print("- Further statistical analysis (e.g., t-tests) could be performed to confirm if the observed differences are statistically significant.")
```


• Box Plot Insights:

- The boxplot reveals the distribution of ages for males and females, segmented by whether they survived or not.
- It highlights the **median age** for each group, showing potential differences between survivors and non-survivors.
- **Outliers** (age values significantly different from the rest) are visible as individual points beyond the whiskers of the boxes.
- **Further Analysis:** To confirm if the differences in age distributions are statistically significant, tests like t-tests or ANOVA could be used.

Key Points:

- The `Age` column had missing values, which were imputed with the median and then the mean (although filling with the median was already sufficient).
- A **boxplot** was used to visualize how the distribution of age differs across genders and survival statuses.
- Outliers in the `Age` column were identified, and the observations suggest possible gender differences in age distribution among survivors.

Let me know if you need further clarification or additional analysis!



```
# Import necessary libraries
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
df = pd.read_csv('Iris.csv')
df.head()
df.describe()
print(df.columns)
df.info()
# --- 1. List down the features and their types ---
print("Features and their types:")
print("\nInference:")
print("- Id: Numeric (int64)")
print("- SepalLengthCm: Numeric (float64)")
print("- SepalWidthCm: Numeric (float64)")
print("- PetalLengthCm: Numeric (float64)")
print("- PetalWidthCm: Numeric (float64)")
print("- Species: Nominal (object)")
import seaborn as sns
import matplotlib.pyplot as plt

# Load dataset
df = sns.load_dataset('iris')
```

```
# Plot histograms
df.hist(figsize=(10, 6), edgecolor='black')
plt.suptitle("Histogram of Iris Dataset Features", fontsize=14)
plt.tight_layout()
plt.show()

plt.figure(figsize=(10, 6))
df.boxplot()
plt.title("Boxplot of Iris Dataset Features")
plt.ylabel("Value")
plt.show()

# Inference as code (printing strings)
print("\n--- Inference from Statistics ---")
print("\nObservations from Histograms:")
print(f"- SepalLengthCm: Distribution appears roughly normal, possibly with slight skew.")
print(f"- SepalWidthCm: Distribution seems more normal with a wider spread.")
print(f"- PetalLengthCm: Shows a likely multimodal distribution.")
print(f"- PetalWidthCm: Also indicates a likely multimodal distribution.")

print("\nObservations from Boxplots and Outliers:")
print(f"- SepalLengthCm: Potential outliers at higher and lower ends.")
print(f"- SepalWidthCm: Potential outliers, especially on the lower end.")
print(f"- PetalLengthCm: Relatively compact, no obvious significant outliers in this sample.")
print(f"- PetalWidthCm: Relatively compact, no obvious significant outliers in this sample.")

print("\nOverall Inference:")
print(f"- Multimodal distributions in PetalLengthCm and PetalWidthCm suggest these are key features for distinguishing species.")
print(f"- Potential outliers in SepalLengthCm and SepalWidthCm warrant further investigation.")
print(f"- Petal dimensions appear more discriminatory than sepal dimensions in this visualization.")
print(f"- 'Species' is categorical and requires different visualization for its distribution.")
```

The code provided loads the Iris dataset, performs basic analysis, and visualizes the distribution of the dataset using histograms and boxplots. Here's an explanation of each step:

1. Importing Libraries and Loading Dataset:

```
python

import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
df = pd.read_csv('Iris.csv')
df.head()
df.describe()
print(df.columns)
df.info()
```

- `pandas` is used for data manipulation and loading the dataset.
- `matplotlib.pyplot` and `seaborn` are used for creating visualizations (histograms and boxplots).

- `df = pd.read_csv('Iris.csv')` loads the Iris dataset from a CSV file.
- `df.head()` shows the first five rows of the dataset to inspect the data.
- `df.describe()` provides summary statistics (mean, standard deviation, etc.) for numerical columns.
- `df.columns` displays the column names to check which features are included.
- `df.info()` provides a summary of the dataset's structure, such as column types and non-null counts.

2. Listing Features and Their Types:

```
python

print("Features and their types:")
print("\nInference:")
print("- Id: Numeric (int64)")
print("- SepalLengthCm: Numeric (float64)")
print("- SepalWidthCm: Numeric (float64)")
print("- PetalLengthCm: Numeric (float64)")
print("- PetalWidthCm: Numeric (float64)")
print("- Species: Nominal (object)")
```

- This prints the feature names along with their data types (numeric or nominal).
- `Id` is numeric, while all other features related to plant measurements (sepal and petal lengths and widths) are numeric.
- `Species` is categorical (nominal), representing the type of Iris flower.

3. Loading the Iris Dataset (Seaborn):

```
python

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

- `sns.load_dataset('iris')` loads the Iris dataset from Seaborn, which is similar to the one you loaded manually, but it includes a 'species' column that categorizes the flower types (setosa, versicolor, virginica).

4. Plotting Histograms:

```
python

df.hist(figsize=(10, 6), edgecolor='black')
plt.suptitle("Histogram of Iris Dataset Features", fontsize=14)
plt.tight_layout()
plt.show()
```

- `df.hist()` creates histograms for all numerical features in the dataset. It displays how the values are distributed for features like sepal and petal lengths and widths.
- `plt.suptitle()` adds a title for the histogram plot.
- `plt.tight_layout()` adjusts the spacing to prevent overlapping elements in the plot.

- `plt.show()` displays the histogram.

5. Plotting Boxplots:

python

```
plt.figure(figsize=(10, 6))
df.boxplot()
plt.title("Boxplot of Iris Dataset Features")
plt.ylabel("Value")
plt.show()
```

- `df.boxplot()` creates boxplots for each numerical feature, showing the distribution and highlighting potential outliers.
- `plt.title()` and `plt.ylabel()` add titles and labels for clarity.
- `plt.show()` displays the boxplot.

6. Observations and Inferences:

python

```
print("\n--- Inference from Statistics ---")
print("\nObservations from Histograms:")
print(f"- SepalLengthCm: Distribution appears roughly normal, possibly with slight skew.")
print(f"- SepalWidthCm: Distribution seems more normal with a wider spread.")
print(f"- PetalLengthCm: Shows a likely multimodal distribution.")
print(f"- PetalWidthCm: Also indicates a likely multimodal distribution.")

print("\nObservations from Boxplots and Outliers:")
print(f"- SepalLengthCm: Potential outliers at higher and lower ends.")
print(f"- SepalWidthCm: Potential outliers, especially on the lower end.")
print(f"- PetalLengthCm: Relatively compact, no obvious significant outliers in this sample.")
print(f"- PetalWidthCm: Relatively compact, no obvious significant outliers in this sample.")

print("\nOverall Inference:")
print(f"- Multimodal distributions in PetalLengthCm and PetalWidthCm suggest these are key features")
print(f"- Potential outliers in SepalLengthCm and SepalWidthCm warrant further investigation.")
print(f"- Petal dimensions appear more discriminatory than sepal dimensions in this visualization.")
print(f"- 'Species' is categorical and requires different visualization for its distribution.")
```

- **Histograms:**

- `SepalLengthCm`: Shows a roughly normal distribution, possibly slightly skewed.
- `SepalWidthCm`: Appears normal with a wider spread.
- `PetalLengthCm` and `PetalWidthCm`: Likely multimodal distributions, suggesting different patterns for each species.

- **Boxplots:**

- `SepalLengthCm` and `SepalWidthCm`: Potential outliers at both ends of the range.
- `PetalLengthCm` and `PetalWidthCm`: No significant outliers, suggesting more compact distributions.

- **Overall Inference:**

- **Petal dimensions** (length and width) seem more useful for distinguishing between species.
- **Sepal dimensions** might need further exploration due to the potential presence of outliers.

- **Species** is categorical and needs different visualizations (such as a count plot or pairplot) to better understand its distribution.

Suggestions:

- **Pairplots** or **violin plots** for categorical feature (`Species`) can give better insights into the relationship between features and species.
- You might want to explore statistical tests (e.g., ANOVA or t-tests) to confirm the observations about the differences in petal and sepal dimensions across species.

Let me know if you need help with those further analyses!