Patient Satisfaction Survey Analysis

Name: **Kalpana M**

Batch: **DA1**

Domain: **Health Care**

1.Define Objectives

Aim:

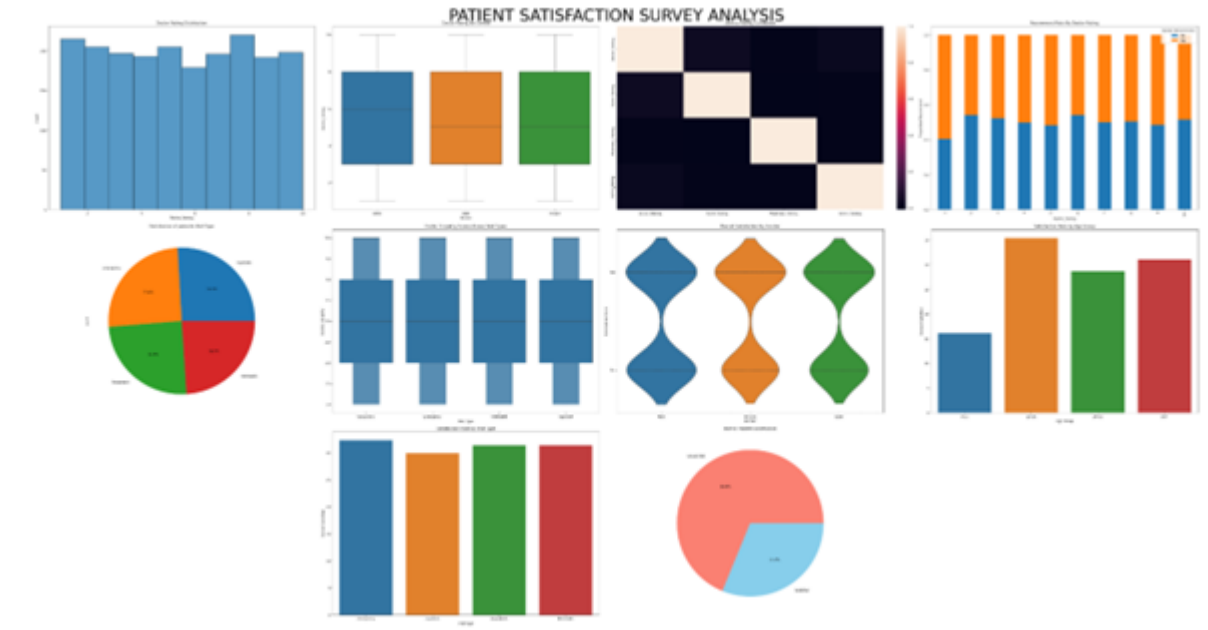
To analyse patient satisfaction survey data specifically doctor, nurse, pharmacy, and administration ratings and identify patterns, correlations, and key drivers affecting satisfaction levels.

Objectives

1. **Data Cleaning& Preparation:**
   Use SQL and Pandas to clean and validate survey data, removing invalid or missing ratings. Apply Seaborn's boxplots to doctor, nurse, pharmacy, and admin ratings to surface outliers and address them appropriately.

2. **Descriptive Analysis:**
   Compute mean, median, distributions, and visualize each rating domain with Matplotlib/Seaborn.

3. How does patient satisfaction differ among various visit types, and which visit type yields the highest satisfaction?
4. Which age groups report the highest satisfaction rates, and how does satisfaction vary across different age demographics?
5. What relationships exist between various service ratings, and do higher ratings in one area correlate with others?
6. What is the overall distribution of doctor ratings, and what does it indicate about perceived doctor quality?
7. What is the satisfaction rate (%) within each age group?
8. How do doctor ratings vary between different genders, and are there significant differences in patient perceptions?
9. What is the overall patient satisfaction rate, and what percentage of patients report being satisfied versus unsatisfied?



Set Goals:

The expected outcomes from this project are:

- **Actionable Improvement Insights**: Identify specific service areas needing targeted improvements based on feedback across domains.
- **Enhanced Patient Satisfaction & Loyalty:** Improve overall patient satisfaction scores and experience, leading to stronger retention, positive word-of-mouth, and potentially increased revenue or reimbursement under value-based care models.
- **Staff Performance Alignment:** Use satisfaction data to evaluate and recognize high-performing teams, while identifying areas for further training or process adjustments.
- **Data-Driven Operational Improvements**: Inform resource allocation, workflow redesigns (e.g. billing clarity, pharmacy triage), and service enhancements across organizational departments.

2.Data Collection

Step 1: Generate & Export to Excel/CSV

The initial dataset was created via a Python script, then saved as an Excel (.xlsx) file for manual review. For compatibility, the file was exported to CSV.
This provided a clean, portable snapshot of the survey responses.

Step 2: Import CSV to SQL

The CSV file was ingested into a SQL database using a bulk load operation (e.g., LOAD DATA INFILE for MySQL, COPY for PostgreSQL), ensuring structured, indexed storage for querying.

Step 3: Query SQL and Load Back into Python

Data was queried from SQL and extracted into Python using Pandas and SQLAlchemy:

Python:

```
import pandas as pd

from sqlalchemy import create_engine

# Create SQLAlchemy engine

engine = create_engine(

    "mysql+pymysql://root:kalpana223@localhost/patient_satisfaction_survey_analysis"

)

# Query into DataFrame once

query = "SELECT * FROM patient_survey"

df = pd.read_sql_query(query, con=engine)

# View the DataFrame

df.head()
```

This loads the cleaned and structured survey data into a Pandas DataFrame for analysis.

Why This Approach?

- CSV → SQL: Enables structured querying and data integrity via constraints/indexes.
- SQL → Python: Combines database efficiency with Pandas' analytical power.

Data Ingestion Process:

The pipeline begins with a Python-generated Excel file (.xlsx) containing raw survey data, which is exported to CSV for universal compatibility. This CSV is loaded into a PostgreSQL table using the COPY command. Finally, we reconnect via SQLAlchemy in Python and use pd.read_sql_query() to load the necessary rating columns (doctor_rating, etc.) and timestamp into a DataFrame for analysis.

Previewing the Initial Dataset:

Python:

Objective:

To obtain an initial overview of the dataset's structure, including the data types, column names, and sample values, facilitating an early assessment of data quality and consistency.

Implementation:

df.head

Explanation:

The head() method in pandas returns the first *n* rows of a DataFrame, with the default being 5 rows. By specifying df.head(10), we retrieve the first 10 rows of the DataFrame df. This approach is instrumental in quickly verifying the dataset's structure and content, ensuring that the data has been loaded correctly and is in the expected format. It serves as a preliminary check before proceeding with more in-depth data analysis or preprocessing steps.

OUTPUT:

	Patient_ID	Age	Gender	Visit_Type	Doctor_Rating	Doctor_Level	Doctor_Recommend	Doctor_Clarity	Doctor_Empathy	Nurse_Rating	..
0	1	75	Other	Outpatient	2	Very Dissatisfied	No	1	4	5	..
1	2	67	Other	Emergency	7	Satisfied	Yes	3	2	3	..
2	3	53	Male	Telehealth	6	Neutral	Yes	2	3	10	..
3	4	80	Male	Inpatient	7	Satisfied	No	1	4	3	..
4	5	22	Other	Emergency	3	Dissatisfied	No	1	3	9	..

5 rows × 29 columns

df.tail()

Explanation:

The df.tail() method in pandas is used to retrieve the last few rows of a DataFrame or Series. By default, it returns the last 5 rows, but you can specify a different number by passing an integer argument.

Key Points:

- df.tail() is useful for quickly inspecting the last few rows of your dataset, especially after operations like sorting or appending new data.
- If n is negative, df.tail(-n) returns all rows except the first n rows. For example, df.tail(-3) will return all rows except the first three.
- If n exceeds the number of rows in the DataFrame, all rows are returned.

This method is commonly used during data exploration to verify the integrity of the dataset's tail end.

OUTPUT:

	Patient_ID	Age	Gender	Visit_Type	Doctor_Rating	Doctor_Level	Doctor_Recommend	Doctor_Clarity	Doctor_Empathy	Nurse_Rating	...
1995	1996	83	Female	Inpatient	6	Neutral	No	1	2	5	...
1996	1997	23	Female	Inpatient	10	Very Satisfied	No	4	3	7	...
1997	1998	28	Male	Telehealth	1	Very Dissatisfied	Yes	1	1	2	...
1998	1999	83	Male	Emergency	10	Very Satisfied	No	3	2	10	...
1999	2000	48	Female	Emergency	10	Very Satisfied	No	5	2	5	...

5 rows × 29 columns

SQL:

Syntax:

#DATABASE CREATION

CREATE DATABASE patient_satisfaction_survey_Analysis;

#USING DATABASE

use patient_satisfaction_survey_Analysis;

#RETRIVING THE DATA

SELECT * FROM patient_survey;

OUTPUT:

Result Grid | Filter Rows: | Export: | Wrap Cell Content:

Patient_ID	Age	Gender	Visit_Type	Doctor_Rating	Doctor_Level	Doctor_Recommend	Doctor_Clarity	Doctor_Empathy	Nurse_Rating	Nurse_Level	Nurse_Recommend	Nurse_Clarity	Nurse_Empathy	Pharmacy_Rating
1	75	Other	Outpatient	2	Very Dissatisfied	No	1	4	5	Neutral	No	4	3	4
2	67	Other	Emergency	7	Satisfied	Yes	3	2	3	Dissatisfied	No	4	3	7
3	53	Male	Telehealth	6	Neutral	Yes	2	3	10	Very Satisfied	No	5	3	8
4	80	Male	Inpatient	7	Satisfied	No	1	4	3	Dissatisfied	No	2	2	9
5	22	Other	Emergency	3	Dissatisfied	No	1	3	9	Very Satisfied	Yes	4	2	5
6	38	Male	Outpatient	4	Dissatisfied	No	2	5	10	Very Satisfied	No	3	2	5
7	24	Female	Inpatient	3	Dissatisfied	No	5	3	7	Satisfied	Yes	3	1	2
8	48	Male	Telehealth	2	Very Dissatisfied	No	1	3	1	Very Dissatisfied	Yes	1	3	8
9	20	Female	Telehealth	10	Very Satisfied	No	1	2	9	Very Satisfied	Yes	3	5	6
10	32	Other	Outpatient	8	Satisfied	Yes	2	5	5	Neutral	No	5	4	10
11	48	Female	Telehealth	7	Satisfied	Yes	1	3	7	Satisfied	Yes	5	4	5
12	77	Other	Outpatient	3	Dissatisfied	Yes	5	2	8	Satisfied	Yes	5	1	8
13	46	Female	Telehealth	10	Very Satisfied	No	2	2	3	Dissatisfied	No	3	1	3
14	79	Male	Telehealth	5	Neutral	Yes	4	5	2	Very Dissatisfied	Yes	1	2	5
15	38	Male	Inpatient	4	Dissatisfied	Yes	4	1	9	Very Satisfied	No	1	5	3
16	71	Other	Inpatient	3	Dissatisfied	Yes	2	2	1	Very Dissatisfied	Yes	3	2	3
17	46	Female	Outpatient	9	Very Satisfied	Yes	2	4	8	Satisfied	No	4	2	7
18	88	Male	Inpatient	10	Very Satisfied	Yes	5	1	2	Very Dissatisfied	No	3	4	6
19	59	Other	Inpatient	2	Very Dissatisfied	Yes	2	3	2	Very Dissatisfied	No	4	2	8
20	58	Other	Emergency	2	Very Dissatisfied	No	5	5	8	Satisfied	No	1	2	10
21	20	Female	Inpatient	7	Satisfied	Yes	3	2	4	Dissatisfied	Yes	5	3	8
22	33	Female	Telehealth	9	Very Satisfied	Yes	1	4	8	Satisfied	No	4	1	6
23	76	Male	Emergency	3	Dissatisfied	No	2	2	9	Very Satisfied	No	5	3	4

3.Exploratory Data Analysis (EDA)

Display the Dimensions of the Loaded DataFrame:

Objective:

To inform stakeholders of the dataset's dimensions—specifically, the number of rows and columns—immediately after loading, ensuring the dataset aligns with expectations.

Implementation:

```
print("Loaded DataFrame shape:", df.shape)
```

Explanation:

The shape attribute of a pandas DataFrame returns a tuple (n_rows, n_columns), indicating the total number of records and features present in the dataset Immediately printing this tuple provides a concise summary of the dataset size, which is critical for:

- **Validation:** Confirming the dataset has loaded completely (e.g., no rows omitted or extra columns included).
- **Context**: Setting expectations for downstream analysis steps, such as data cleaning, preprocessing, or model training.
- **Debugging**: Rapidly identifying unexpectedly large or small datasets, which could indicate issues in data ingestion or merging logic.

OUTPUT:

```
Loaded DataFrame shape: (2000, 29)
```

Display a Concise Summary of the DataFrame Structure:

Objective:

To gain a high-level overview of the DataFrame's metadata—including entry count, column details, data types, missing values, and memory usage—thereby aiding in data validation, cleaning, and optimization.

Implementation:

```
: df.info()
```

Explanation:

The df.info() method in pandas prints a concise summary containing:

- The class type and index range (total number of rows).
- Column count, names, and an index column for reference.
- Non-null counts per column, essential for identifying missing data.
- Data types (dtype) for each column, useful for detecting mismatches or unexpected types.
- Memory usage of the DataFrame, which is crucial for performance tuning in large datasets. By default, this is an estimate, but can be made precise using memory_usage="deep"

Why It Matters:

- Validates that your dataset is complete and correctly loaded.
- Highlights missing data early, facilitating cleaning strategies.
- Reveals column data types to prevent type-related processing or modeling errors.

OUTPUT:

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 2000 entries, 0 to 1999
Data columns (total 29 columns):
 #   Column                   Non-Null Count  Dtype
---  ------                   --------------  -----
 0   Patient_ID               2000 non-null   int64
 1   Age                      2000 non-null   int64
 2   Gender                   2000 non-null   object
 3   Visit_Type               2000 non-null   object
 4   Doctor_Rating            2000 non-null   int64
 5   Doctor_Level             2000 non-null   object
 6   Doctor_Recommend         2000 non-null   object
 7   Doctor_Clarity           2000 non-null   int64
 8   Doctor_Empathy           2000 non-null   int64
 9   Nurse_Rating             2000 non-null   int64
 10  Nurse_Level              2000 non-null   object
 11  Nurse_Recommend          2000 non-null   object
 12  Nurse_Clarity            2000 non-null   int64
 13  Nurse_Empathy            2000 non-null   int64
 14  Pharmacy_Rating          2000 non-null   int64
 15  Pharmacy_Level           2000 non-null   object
 16  Pharmacy_Recommend       2000 non-null   object
 17  Pharmacy_Clarity         2000 non-null   int64
 18  Pharmacy_Empathy         2000 non-null   int64
 19  Admin_Rating             2000 non-null   int64
 20  Admin_Level              2000 non-null   object
 21  Admin_Recommend          2000 non-null   object
 22  Admin_Clarity            2000 non-null   int64
 23  Admin_Empathy            2000 non-null   int64
 24  Fee_Structure_Rating     2000 non-null   int64
 25  Fee_Structure_Level      2000 non-null   object
 26  Fee_Structure_Recommend  2000 non-null   object
 27  Fee_Structure_Clarity    2000 non-null   int64
 28  Fee_Structure_Empathy    2000 non-null   int64
dtypes: int64(17), object(12)
memory usage: 453.3+ KB
```

Generate a Statistical Summary of the Data Frame:

Objective:

To obtain a summary of key descriptive statistics for numerical (and optionally categorical) columns, aiding in quick insights into distribution, spread, and central tendencies.

Implementation:

```
: df.describe()
```

Explanation:

The df.describe() method produces descriptive statistics that summarize the central tendency, dispersion, and shape of a dataset's distribution, omitting NaN values.

For numeric columns, the output includes:

- count, mean, std (standard deviation), min, max
- Percentiles (25%, 50% (median), 75%) by default

For non-numeric columns, such as strings or timestamps, it may include:

- count, unique, top (most frequent), freq (frequency of top)
- Timestamps additionally show first and last values

OUTPUT:

	Patient_ID	Age	Doctor_Rating	Doctor_Clarity	Doctor_Empathy	Nurse_Rating	Nurse_Clarity	Nurse_Empathy	Pharmacy_Rating	Pharmacy_Clarity	Pha
count	2000.000000	2000.000000	2000.000000	2000.000000	2000.000000	2000.000000	2000.000000	2000.000000	2000.000000	2000.000000	
mean	1000.500000	53.761500	5.463500	2.977000	3.020000	5.463000	2.985000	3.009000	5.521500	3.015500	
std	577.494589	21.342994	2.899115	1.400876	1.414072	2.883711	1.410947	1.421594	2.881091	1.403298	
min	1.000000	18.000000	1.000000	1.000000	1.000000	1.000000	1.000000	1.000000	1.000000	1.000000	
25%	500.750000	35.000000	3.000000	2.000000	2.000000	3.000000	2.000000	2.000000	3.000000	2.000000	
50%	1000.500000	54.000000	5.000000	3.000000	3.000000	5.000000	3.000000	3.000000	6.000000	3.000000	
75%	1500.250000	73.000000	8.000000	4.000000	4.000000	8.000000	4.000000	4.000000	8.000000	4.000000	
max	2000.000000	90.000000	10.000000	5.000000	5.000000	10.000000	5.000000	5.000000	10.000000	5.000000	

Why It Matters:

- Helps validate data (e.g., no impossibly large or negative values)
- Provides insights into data distribution
- Identifies outliers and informs preprocessing steps

Column Data Type Inspection:

Objective:

To inspect and document the data type of each column, ensuring variables are correctly typed for subsequent analysis or transformations.

Implementation:

```
: df.dtypes
```

This returns a pandas Series where the index is the column name and the value is its data type.

OUTPUT:

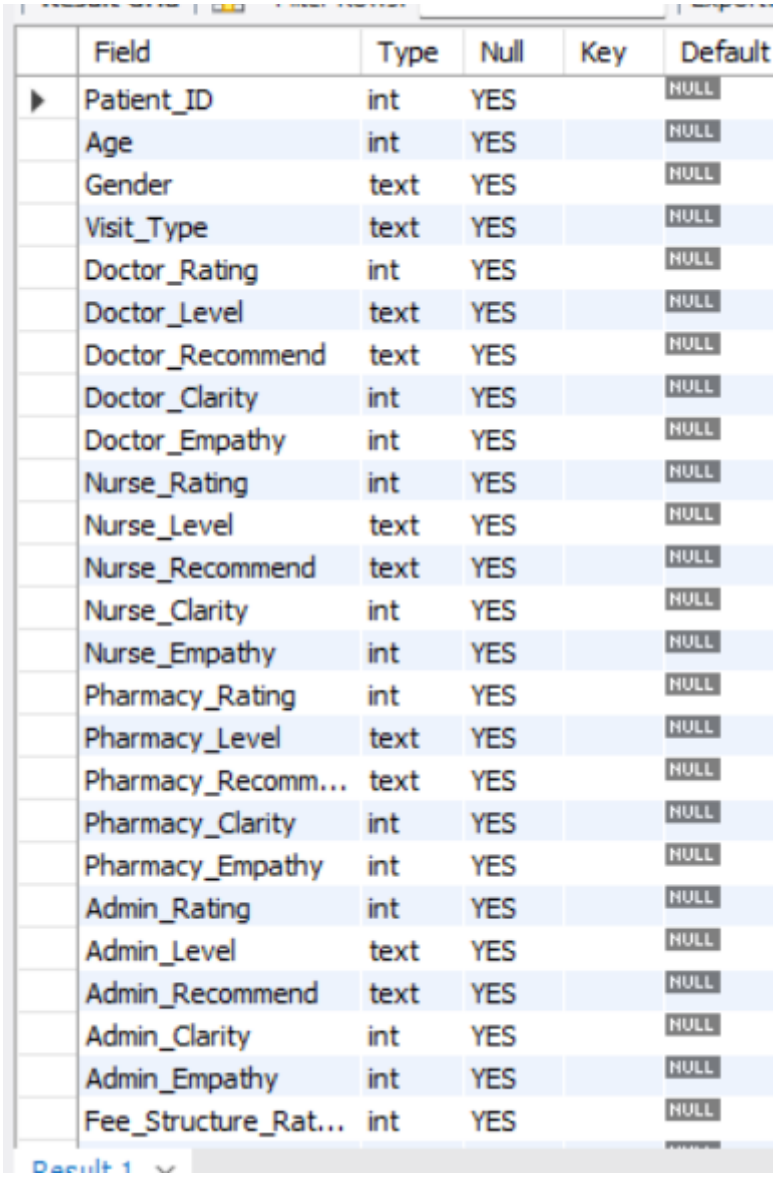
```
Patient_ID                int64
Age                       int64
Gender                   object
Visit_Type               object
Doctor_Rating             int64
Doctor_Level             object
Doctor_Recommend         object
Doctor_Clarity            int64
Doctor_Empathy            int64
Nurse_Rating              int64
Nurse_Level              object
Nurse_Recommend          object
Nurse_Clarity             int64
Nurse_Empathy             int64
Pharmacy_Rating           int64
Pharmacy_Level           object
Pharmacy_Recommend       object
Pharmacy_Clarity          int64
Pharmacy_Empathy          int64
Admin_Rating              int64
Admin_Level              object
Admin_Recommend          object
Admin_Clarity             int64
Admin_Empathy             int64
Fee_Structure_Rating      int64
Fee_Structure_Level      object
Fee_Structure_Recommend  object
Fee_Structure_Clarity     int64
Fee_Structure_Empathy     int64
dtype: object
```

SQL:

#to view the structure of data

DESCRIBE patient_survey;

OUTPUT:



Field	Type	Null	Key	Default
Patient_ID	int	YES		NULL
Age	int	YES		NULL
Gender	text	YES		NULL
Visit_Type	text	YES		NULL
Doctor_Rating	int	YES		NULL
Doctor_Level	text	YES		NULL
Doctor_Recommend	text	YES		NULL
Doctor_Clarity	int	YES		NULL
Doctor_Empathy	int	YES		NULL
Nurse_Rating	int	YES		NULL
Nurse_Level	text	YES		NULL
Nurse_Recommend	text	YES		NULL
Nurse_Clarity	int	YES		NULL
Nurse_Empathy	int	YES		NULL
Pharmacy_Rating	int	YES		NULL
Pharmacy_Level	text	YES		NULL
Pharmacy_Recomm...	text	YES		NULL
Pharmacy_Clarity	int	YES		NULL
Pharmacy_Empathy	int	YES		NULL
Admin_Rating	int	YES		NULL
Admin_Level	text	YES		NULL
Admin_Recommend	text	YES		NULL
Admin_Clarity	int	YES		NULL
Admin_Empathy	int	YES		NULL
Fee_Structure_Rat...	int	YES		NULL

Correct Data Representation

- int64 columns are numeric and ready for calculations or modeling.
- object columns often represent categories

Performance & Memory

- object dtype stores each value as a Python string inefficient in memory and slower in operations.
- Converting frequently repeated text categories to **category** dtype uses integer codes and a lookup table, saving RAM and speeding up grouping or comparisons

Accuracy in Analysis

- Converting types ensures downstream steps like encoding or validation correctly understand the data type

Column Names:

Objective:

When you use df.columns in pandas, you're accessing the column labels of the DataFrame as an Index object.

Implementation:

```
: df.columns
```

OUTPUT:

```
Index(['Patient_ID', 'Age', 'Gender', 'Visit_Type', 'Doctor_Rating',
       'Doctor_Level', 'Doctor_Recommend', 'Doctor_Clarity', 'Doctor_Empathy',
       'Nurse_Rating', 'Nurse_Level', 'Nurse_Recommend', 'Nurse_Clarity',
       'Nurse_Empathy', 'Pharmacy_Rating', 'Pharmacy_Level',
       'Pharmacy_Recommend', 'Pharmacy_Clarity', 'Pharmacy_Empathy',
       'Admin_Rating', 'Admin_Level', 'Admin_Recommend', 'Admin_Clarity',
       'Admin_Empathy', 'Fee_Structure_Rating', 'Fee_Structure_Level',
       'Fee_Structure_Recommend', 'Fee_Structure_Clarity',
       'Fee_Structure_Empathy'],
      dtype='object')
```

Explanation:

df.columns returns the column names (labels) of the DataFrame as an Index.

It takes **no parameters**, just df.columns on its own.

You can also **modify** column names by assigning a new list or index to it.

Why use it?

- **Inspect column names** easily — great for data exploration and validation.
- **Rename columns** via assignment.
- Works even with missing data or heterogeneous dtypes

4.Data Cleaning

Identify Missing Values:

Objective:

To systematically detect and count the number of missing entries (NaN, None, NaT) in each column of your DataFrame, enabling targeted cleaning operations.

Implementation:

```
[ ]: df.isnull().sum()
```

Explanation:

- df.isnull() generates a DataFrame of the same shape, replacing each cell with a Boolean: True if the value is missing and False otherwise
- Applying .sum() treats True as 1 and False as 0, producing the total count of missing values in each column by default (axis=0) .

Why It Matters:

- Pinpoints columns with missing data requiring action (technical fixes or domain-informed strategies).
- Helps prioritize cleaning efforts — large numbers of missing entries may require different handling than sporadic ones.
- Offers a first quantitative snapshot of data completeness, essential for transparency in reporting and reproducibility.

OUTPUT:

```
Patient_ID                0
Age                       0
Gender                    0
Visit_Type                0
Doctor_Rating             0
Doctor_Level              0
Doctor_Recommend          0
Doctor_Clarity            0
Doctor_Empathy            0
Nurse_Rating              0
Nurse_Level               0
Nurse_Recommend           0
Nurse_Clarity             0
Nurse_Empathy             0
Pharmacy_Rating           0
Pharmacy_Level            0
Pharmacy_Recommend        0
Pharmacy_Clarity          0
Pharmacy_Empathy          0
Admin_Rating              0
Admin_Level               0
Admin_Recommend           0
Admin_Clarity             0
Admin_Empathy             0
Fee_Structure_Rating      0
Fee_Structure_Level       0
Fee_Structure_Recommend   0
Fee_Structure_Clarity     0
Fee_Structure_Empathy     0
dtype: int64
```

Since **all columns have zero missing values**, the dataset is complete. No imputation or row/column removal is required.

Count Duplicate Rows:

Objective:

To detect and quantify any exact duplicate rows in your dataset, ensuring data quality and integrity before proceeding.

Implementation:

```
df.duplicated().sum()
```

Explanation:

- The df.duplicated() method returns a boolean Series of the same length as the DataFrame, where True indicates a duplicate row matching an earlier one, and False indicates either the first occurrence or a unique row
- By default, pandas uses keep='first', meaning it marks all but the first occurrence of each duplicate set as duplicates

OUTPUT:

```
]:  0
```

This indicates no duplicate rows were detected

Series.unique():

df['Gender'].unique() returns all distinct values that appear in the 'Gender' column (a pandas Series). It's based on a hash table algorithm and preserves the order of first appearance without sorting.

Implementation:

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

OUTPUT:

```
array(['Other', 'Male', 'Female'], dtype=object)
```

Implementation:

```
:  df['Visit_Type'].unique()
```

OUTPUT:

```
array(['Outpatient', 'Emergency', 'Telehealth', 'Inpatient'], dtype=object)
```

Explanation:

- This invokes the unique() method on the pandas **Series** df['Visit_Type'], returning **all distinct values** present in that column.
- The result is typically a **NumPy ndarray**, or an **ExtensionArray** if the Series uses a specialized dtype like categorical or datetime
- Unique values are preserved in the **order of their first appearance** no implicit sorting occurs

- **Missing values** (NaN, None, pd.NA) are included as unique entries if they appear in the Series

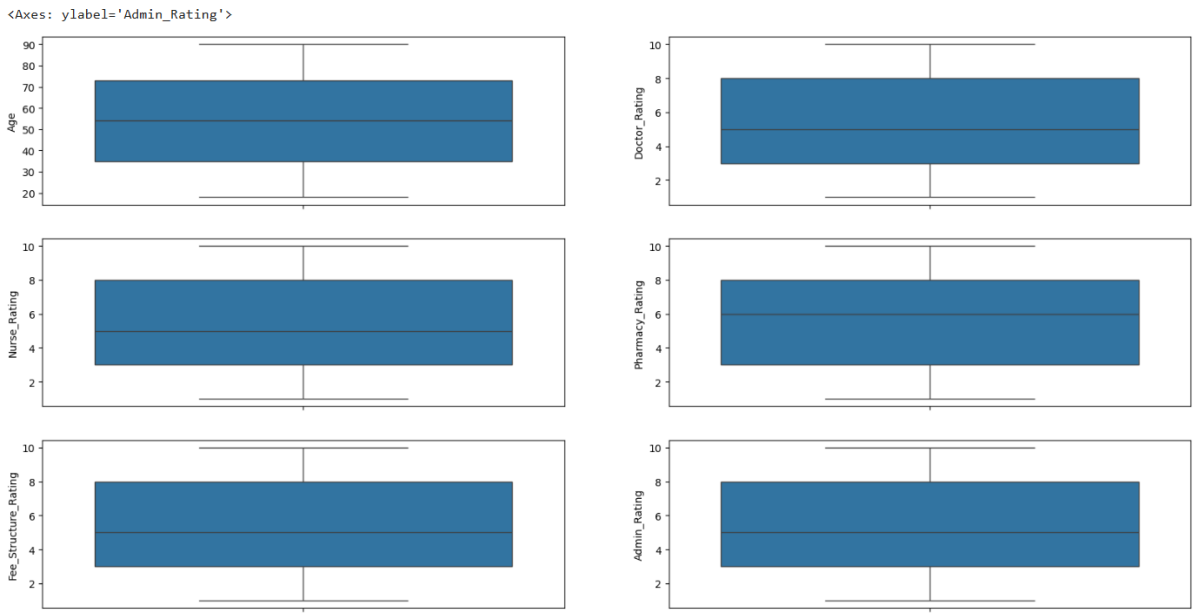
Identifying and Handling Outliers:

Implementation:

DETECT OUTLIERS

```
]:  import matplotlib.pyplot as plt
    import seaborn as sns
    plt.figure(figsize=(20,10))
    plt.subplot(3,2,1)
    sns.boxplot(df['Age'])
    plt.subplot(3,2,2)
    sns.boxplot(df['Doctor_Rating'])
    plt.subplot(3,2,3)
    sns.boxplot(df['Nurse_Rating'])
    plt.subplot(3,2,4)
    sns.boxplot(df['Pharmacy_Rating'])
    plt.subplot(3,2,5)
    sns.boxplot(df['Fee_Structure_Rating'])
    plt.subplot(3,2,6)
    sns.boxplot(df['Admin_Rating'])
```

OUTPUT:



Each boxplot displays these key components for the respective variable:

- Median (central line): the 50th percentile
- Box edges: first (Q1, 25th percentile) and third quartile (Q3, 75th percentile)
- Whiskers: data range up to 1.5×IQR above Q3 and below Q1
- Outliers: individual points outside the whiskers; calculated using Tukey's method (values $> Q3 + 1.5 \times IQR$ or $< Q1 - 1.5 \times IQR$)

Why This Matters

- **Compare variables**: Aligning the six variables side by side allows direct visual comparison of their distributions and variability.
- **Outlier detection**: Highlights data quality or unusual responses (for example, very low admin ratings).
- **Data insight**: Helps you decide if variables need transformation, if outliers should be investigated, or if data collection needs review.

Neither approach flagged any data points as outliers; all values fell within expected ranges. As a result:

- The **mean and median** are reliable estimates of central tendency .
- There is **no need to transform**, remove any observations.
- We can proceed under the assumption of data consistency and completeness .

Why "No Outliers" Is a Good Result?

- **Data quality confirmation**: The absence of outliers suggests no apparent data-entry or measurement errors.
- **Statistical validity**: Means, standard deviations, and other metrics are robust and representative.
- **Analysis simplicity**: No additional treatments (like trimming or robust modeling) are necessary, keeping the analysis straightforward.

5.DATA ANALYSIS

How many patients awarded both the maximum rating for Doctor_Rating and the maximum for Doctor_Clarity?

SQL(Stored Procedure):

#CREATING PROCEDURE TO GET DOCTOR INFORMATION

DELIMITER $$

CREATE PROCEDURE GetDoctorInfo(

IN Min_Doctor_Rating int,

IN Min_Doctor_Clarity int

)

BEGIN

SELECT * FROM patient_survey

WHERE Doctor_Rating>=Min_Doctor_Rating AND Doctor_Clarity>=Min_Doctor_Clarity;

END $$

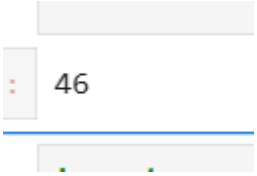
DELIMITER $$

Implementation:

Retrieving Stored Procedure Results into a Pandas DataFrame:

```
import pandas as pd
from sqlalchemy import create_engine, text

engine = create_engine("mysql+pymysql://root:kalpana223@localhost/patient_satisfaction_survey_analysis")

Min_Doctor_Rating = 10
Min_Doctor_Clarity = 5

# Use a text CALL statement and pass parameters
sql = text("CALL GetDoctorInfo(:min_rating, :min_clarity)")
df = pd.read_sql_query(sql, engine, params={
    "min_rating": Min_Doctor_Rating,
    "min_clarity": Min_Doctor_Clarity
})

len(df)
```

OUTPUT:



Description:

- engine = create_engine(...)
  Establishes a connection to the MySQL database using SQL Alchemy.

- sql = text("CALL GetDoctorInfo(:min_rating, :min_clarity)")
  Defines a stored procedure call using SQLAlchemy's text() function, which supports named parameters.

- pd.read_sql_query(sql, engine, params={...})
  Executes the stored procedure and reads the returned rows into a pandas DataFrame, binding parameters from the params dictionary.

Why This Approach?

- **Security**: Using bound parameters avoids SQL injection risks.
- **Flexibility**: Easily adjust thresholds or add filtering logic using Python variables.
- **Clarity**: Clearly documents which columns are relevant for reporting or analysis.

2.What is the average age of patients, grouped by gender?

Implementation:

```
import numpy as np
new_df=df.groupby('Gender')['Age'].mean()
new_df
```

OUTPUT:

```
Gender
Female    54.057751
Male      54.597579
Other     52.663730
Name: Age, dtype: float64
```

Verification:

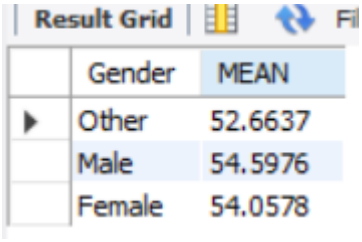
SQL:

SELECT Gender,AVG(Age) as MEAN

FROM patient_survey

GROUP BY Gender;

OUTPUT:



Result Grid

Gender	MEAN
Other	52.6637
Male	54.5976
Female	54.0578

Objective: Calculate the mean age of patients.

Stratification: Results are segmented by gender category (e.g. Male, Female, Other).

3.What are the ratings (Doctor, Nurse, Pharmacy, Admin, and Fee Structure) given by patients who visited as an Emergency?

Implementation:

```
Emergency_Patients_data=df.loc[df['Visit_Type']=='Emergency']
Emergency_Patients_data[['Gender','Visit_Type','Doctor_Rating','Nurse_Rating','Pharmacy_Rating','Admin_Rating','Fee_Structure_Rating']]
```

6 Female Emergency 10 8 4 9 3

OUTPUT:

	Gender	Visit_Type	Doctor_Rating	Nurse_Rating	Pharmacy_Rating	Admin_Rating	Fee_Structure_Rating
5	Male	Emergency	10	2	4	2	8
6	Female	Emergency	10	8	4	9	3
8	Male	Emergency	10	2	8	7	9
11	Other	Emergency	10	5	4	5	8
12	Other	Emergency	10	5	10	5	8
14	Male	Emergency	10	5	9	3	1
25	Female	Emergency	10	8	5	10	8
27	Male	Emergency	10	8	1	5	4
31	Other	Emergency	10	9	2	1	8

SQL:

```
SELECT
Visit_Type,Doctor_Rating,Nurse_Rating,Pharmacy_Rating,Admin_Rating,Fee_Structure_Rating
FROM patient_survey

WHERE Visit_Type='Emergency';
```

Data Filtering:

- **Purpose:** Selects only those survey records where patients visited under the Emergency category.
- **Mechanism**: Uses df.loc with the condition df['Visit_Type'] == 'Emergency' to create a filtered DataFrame named Emergency_Patients_data.

Column Selection

- **Purpose:** Focuses on the demographic and satisfaction metrics relevant to emergency visits.
- **Result:** A streamlined DataFrame showing demographic info and all rating dimensions for emergency-visit patients.

Targeted Analysis: Ideal when analyzing satisfaction patterns specifically among emergency department patients.

4.How many patients who visited as Emergency reported their doctor interaction as 'Very Satisfied' and what are their gender distributions?

Implementation:

```
Emergency_Satisfied_Patients=df.loc[(df['Visit_Type']=='Emergency') & (df['Doctor_Level']=='Very Satisfied')]
Emergency_Satisfied_Patients[['Gender','Visit_Type','Doctor_Level']]
```

OUTPUT:

	Gender	Visit_Type	Doctor_Level
5	Male	Emergency	Very Satisfied
6	Female	Emergency	Very Satisfied
8	Male	Emergency	Very Satisfied
11	Other	Emergency	Very Satisfied
12	Other	Emergency	Very Satisfied
14	Male	Emergency	Very Satisfied
25	Female	Emergency	Very Satisfied
27	Male	Emergency	Very Satisfied
31	Other	Emergency	Very Satisfied
43	Other	Emergency	Very Satisfied
45	Female	Emergency	Very Satisfied

SQL:

```
SELECT count(*) as Satisfied_Patients

FROM patient_survey

WHERE Visit_Type='Emergency' AND Doctor_Level='Very Satisfied';
```

Filtering via loc

Uses df.loc[...] to perform boolean indexing, filtering rows based on two criteria:

- Visit Type must be "Emergency"
- Doctor_Level must be "Very Satisfied"
  This method leverages label-based indexing combined with a boolean mask, as outlined in the pandas documentation

Selecting Specific Columns

After filtering, the code selects a subset of columns Gender, Visit_Type, and Doctor_Level to shape the resulting DataFrame. This makes the output concise and relevant for analysis.

Interpretation:

This yields a filtered dataset listing only those emergency-visit patients who expressed the highest satisfaction level with their doctor, including their gender for downstream demographic analysis.

5.How many patients are there in each gender category?

Implementation:

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

OUTPUT:

```
Gender
Other     681
Male      661
Female    658
Name: count, dtype: int64
```

SQL:

SELECT Gender,COUNT(*)

FROM patient_survey

GROUP BY Gender;

Purpose

- Computes the frequency of each unique value in the Gender column.
- Enables demographic breakdown by counting how many entries correspond to each gender (e.g., Male, Female, Other).

6.How many patients fall into each Visit_Type category?

Implementation:

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

OUTPUT:

```
Visit_Type
Inpatient     518
Emergency     508
Outpatient    495
Telehealth    479
Name: count, dtype: int64
```

SQL:

```
SELECT Visit_Type,COUNT(*)

FROM patient_survey

GROUP BY Visit_Type;
```

Explanation

- **Objective:** Determine the count of records for each unique value in the Visit_Type column.
- **Pandas Method:** The Series.value_counts() function returns a new Series indexed by the unique values in Visit_Type, with their corresponding frequencies in descending order .
- **Use Case:** Useful for quickly understanding the distribution of visit types (e.g., Emergency, Routine, Follow-Up) within the patient dataset .

7.What is the distribution of responses in the Doctor_Recommend field?

Python:

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

OUTPUT:

```
Doctor_Recommend
Yes    1003
No      997
Name: count, dtype: int64
```

SQL:

```
SELECT Doctor_Recommend,COUNT(*)

FROM patient_survey

GROUP BY Doctor_Recommend;
```

Purpose: Tally how many patients selected each unique response option in the Doctor_Recommend column

8.What is the count of each visit type for every gender category in the data?

SQL:

```
SELECT Gender,Visit_Type,COUNT(*) AS Total_Patients

FROM patient_survey

GROUP BY Gender,Visit_Type;
```

Python:

```
df.groupby(['Gender'])['Visit_Type'].value_counts().unstack()
```

OUTPUT:

Visit_Type	Emergency	Inpatient	Outpatient	Telehealth
Gender				
Female	176	155	170	157
Male	165	179	167	150
Other	167	184	158	172

df.groupby(['Gender'])

- Divides the DataFrame into separate groups based on the values in the Gender column.
- This is the "Split" step in the split-apply-combine paradigm used by pandas

['Visit_Type'].value_counts()

- For each gender group, counts occurrences of each distinct Visit_Type value.
- This is akin to applying value_counts() separately within each gender group, providing the frequency of different visit types per gender

unstack()

- Reshapes the resulting Series from a MultiIndex (gender × visit type) into a tabular DataFrame layout.
- Now each gender appears as a row, and each visit type becomes its own column, with frequency counts as cell values

This command groups patient visits by gender, counts how many visits fall into each visit-type category per gender, and then pivots the result so that each gender is a separate row and each visit type is a distinct column showing frequency counts.

How are different doctor levels distributed across genders?

SQL:

SELECT Gender,Doctor_Level,COUNT(*) AS PATIENT_COUNT

FROM patient_survey

GROUP BY Gender,Doctor_Level;

Python:

```
df.groupby(['Gender'])['Doctor_Level'].value_counts().unstack()
```

OUTPUT:

Doctor_Level	Dissatisfied	Neutral	Satisfied	Very Dissatisfied	Very Satisfied
Gender					
Female	135	119	136	134	134
Male	138	155	119	138	111
Other	117	110	161	148	145

Explanation:

Within each gender group, this counts the frequency of each distinct value in the Doctor Level column (e.g. Very Satisfied, Satisfied, etc.). It yields a hierarchical **Series** with a **Multi Index** (Gender, Doctor Level) paired with the count for each combination.

This command groups patient feedback by gender, counts the number of responses at each Doctor Level within each gender category, and then transforms the result into a pivoted table where each gender is a row and each doctor satisfaction level is a column showing the frequency of responses.

How are different Nurse levels distributed across genders?

SQL:

SELECT Gender,Nurse_Level,COUNT(*) AS PATIENT_COUNT

FROM patient_survey

GROUP BY Gender,Nurse_Level;

Python:

```
df.groupby(['Gender'])['Nurse_Level'].value_counts().unstack()
```

OUTPUT:

Nurse_Level	Dissatisfied	Neutral	Satisfied	Very Dissatisfied	Very Satisfied
Gender					
Female	131	141	139	136	111
Male	110	129	125	152	145
Other	144	151	128	127	131

How are different Pharmacy levels distributed across genders?

SQL:

```
SELECT Gender,Pharmacy_Level,COUNT(*) AS PATIENT_COUNT
FROM patient_survey
GROUP BY Gender,Pharmacy_Level;
```

Python:

```
df.groupby(['Gender'])['Pharmacy_Level'].value_counts().unstack()
```

OUTPUT:

Pharmacy_Level	Dissatisfied	Neutral	Satisfied	Very Dissatisfied	Very Satisfied
Gender					
Female	131	124	142	114	147
Male	136	133	128	130	134
Other	148	126	134	147	126

How are different Admin levels distributed across genders?

Python:

```
df.groupby(['Gender'])['Admin_Level'].value_counts().unstack()
```

OUTPUT:

Admin_Level	Dissatisfied	Neutral	Satisfied	Very Dissatisfied	Very Satisfied
Gender					
Female	143	122	134	123	136
Male	134	136	118	140	133
Other	143	121	142	143	132

SQL:

```
SELECT Gender,Admin_Level,COUNT(*) AS PATIENT_COUNT
FROM patient_survey
GROUP BY Gender,Admin_Level;
```

How many emergency visits involved patients who were 'Very Satisfied' with the doctor?

Stored procedure:

```
DELIMITER $$
CREATE PROCEDURE GetVisitTypeDoctorInfo (
    IN in_doctor_level VARCHAR(50),
    IN in_visit_type VARCHAR(50)
)
BEGIN
SELECT Patient_ID, Age,Gender,Visit_Type,Doctor_Level FROM patient_survey
WHERE Doctor_Level = in_doctor_level
AND Visit_Type = in_visit_type;
END $$
DELIMITER $$
```

Python:

Implementation:

```
import pandas as pd
from sqlalchemy import create_engine, text

engine = create_engine("mysql+pymysql://root:kalpana223@localhost/patient_satisfaction_survey_analysis")

in_doctor_level = 'Very Satisfied'
in_visit_type = 'Emergency'

# Use a text CALL statement and pass parameters
sql = text("CALL GetVisitTypeDoctorInfo(:in_doctor_level, :in_visit_type)")
df = pd.read_sql_query(sql, engine, params={
    "in_doctor_level": in_doctor_level,
    "in_visit_type": in_visit_type
})
df
```

OUTPUT:

	Patient_ID	Age	Gender	Visit_Type	Doctor_Level
0	27	32	Male	Emergency	Very Satisfied
1	106	60	Female	Emergency	Very Satisfied
2	159	59	Female	Emergency	Very Satisfied
3	184	27	Female	Emergency	Very Satisfied
4	262	86	Female	Emergency	Very Satisfied
...	...	...	...	...	...
101	1964	47	Other	Emergency	Very Satisfied
102	1980	44	Male	Emergency	Very Satisfied
103	1987	81	Female	Emergency	Very Satisfied
104	1999	83	Male	Emergency	Very Satisfied
105	2000	48	Female	Emergency	Very Satisfied

Explanation:

1. create_engine

Creates a SQLAlchemy engine that connects to your MySQL database via pymysql. This engine is used to execute SQL statements, including stored procedures.

2. text

The SQL query is wrapped in sqlalchemy.text() to represent a parameterized SQL statement.This allows named parameters (e.g. :in_doctor_level) to be bound at execution.
PandasPandas

3. pd.read_sql_query

Executes the SQL query and returns the result set as a pandas DataFrame.This function supports both raw SQL strings and SQLAlchemy text/select objects.
PandasPandas

4. parameters

Named parameters are passed as a dictionary mapping parameter placeholders to their Python variables.

5. df (returned DataFrame)

Contains the result rows returned by the GetVisitTypeDoctorInfo stored procedure.Each row becomes a DataFrame row, columns correspond to result set fields.

How many emergency visits involved patients who were 'Satisfied' with the doctor?

Implementation:

```
import pandas as pd
from sqlalchemy import create_engine, text

engine = create_engine("mysql+pymysql://root:kalpana223@localhost/patient_satisfaction_survey_analysis")

in_doctor_level = 'Satisfied'
in_visit_type = 'Emergency'

# Use a text CALL statement and pass parameters
sql = text("CALL GetVisitTypeDoctorInfo(:in_doctor_level, :in_visit_type)")
df = pd.read_sql_query(sql, engine, params={
    "in_doctor_level": in_doctor_level,
    "in_visit_type": in_visit_type
})
df
```

OUTPUT:

	Patient_ID	Age	Gender	Visit_Type	Doctor_Level
0	2	67	Other	Emergency	Satisfied
1	43	81	Other	Emergency	Satisfied
2	80	85	Female	Emergency	Satisfied
3	87	40	Other	Emergency	Satisfied
4	127	19	Male	Emergency	Satisfied
...	...	...	...	...	...
109	1880	57	Other	Emergency	Satisfied
110	1893	86	Female	Emergency	Satisfied
111	1958	51	Female	Emergency	Satisfied
112	1968	68	Female	Emergency	Satisfied
113	1991	78	Other	Emergency	Satisfied

114 rows × 5 columns

How many emergency visits involved patients who were 'Dissatisfied' with the doctor?

Implementation:

```
in_doctor_level = 'DisSatisfied'
in_visit_type = 'Emergency'

# Use a text CALL statement and pass parameters
sql = text("CALL GetVisitTypeDoctorInfo(:in_doctor_level, :in_visit_type)")
df = pd.read_sql_query(sql, engine, params={
    "in_doctor_level": in_doctor_level,
    "in_visit_type": in_visit_type
})
df
```

OUTPUT:

	Patient_ID	Age	Gender	Visit_Type	Doctor_Level
0	5	22	Other	Emergency	Dissatisfied
1	23	26	Male	Emergency	Dissatisfied
2	34	63	Male	Emergency	Dissatisfied
3	64	41	Male	Emergency	Dissatisfied
4	65	78	Female	Emergency	Dissatisfied
...	...	...	...	...	...
93	1950	21	Female	Emergency	Dissatisfied
94	1951	52	Male	Emergency	Dissatisfied
95	1960	59	Other	Emergency	Dissatisfied
96	1982	18	Male	Emergency	Dissatisfied
97	1994	53	Female	Emergency	Dissatisfied

98 rows × 5 columns

How many emergency visits involved patients who were 'Neutral' with the doctor?

Implementation:

```
in_doctor_level = 'Neutral'
in_visit_type = 'Emergency'
sql = text("CALL GetVisitTypeDoctorInfo(:in_doctor_level, :in_visit_type)")
df = pd.read_sql_query(sql, engine, params={
    "in_doctor_level": in_doctor_level,
    "in_visit_type": in_visit_type
})
df
```

OUTPUT:

	Patient_ID	Age	Gender	Visit_Type	Doctor_Level
0	28	29	Female	Emergency	Neutral
1	45	35	Female	Emergency	Neutral
2	61	80	Female	Emergency	Neutral
3	94	67	Female	Emergency	Neutral
4	115	73	Female	Emergency	Neutral
...	...	...	...	...	...
89	1937	37	Male	Emergency	Neutral
90	1961	29	Other	Emergency	Neutral
91	1988	30	Male	Emergency	Neutral
92	1989	74	Female	Emergency	Neutral
93	1992	82	Male	Emergency	Neutral

94 rows × 5 columns

How many Telehealth visits involved patients who were 'Neutral' with the doctor?

Implementation:

```
in_doctor_level = 'Neutral'
in_visit_type = 'Telehealth'
sql = text("CALL GetVisitTypeDoctorInfo(:in_doctor_level, :in_visit_type)")
df = pd.read_sql_query(sql, engine, params={
    "in_doctor_level": in_doctor_level,
    "in_visit_type": in_visit_type
})
df
```

OUTPUT:

	Patient_ID	Age	Gender	Visit_Type	Doctor_Level
0	3	53	Male	Telehealth	Neutral
1	14	79	Male	Telehealth	Neutral
2	29	24	Female	Telehealth	Neutral
3	48	22	Male	Telehealth	Neutral
4	51	51	Female	Telehealth	Neutral
...	...	...	...	...	...
79	1949	50	Female	Telehealth	Neutral
80	1952	76	Female	Telehealth	Neutral
81	1959	50	Other	Telehealth	Neutral
82	1981	46	Other	Telehealth	Neutral
83	1983	46	Female	Telehealth	Neutral

How many Telehealth visits involved patients who were 'Dissatisied' with the doctor?

Implementation:

```
in_doctor_level = 'Dissatisfied'
in_visit_type = 'Telehealth'

# Use a text CALL statement and pass parameters
sql = text("CALL GetVisitTypeDoctorInfo(:in_doctor_level, :in_visit_type)")
df = pd.read_sql_query(sql, engine, params={
    "in_doctor_level": in_doctor_level,
    "in_visit_type": in_visit_type
})
df
```

OUTPUT:

	Patient_ID	Age	Gender	Visit_Type	Doctor_Level
0	33	61	Female	Telehealth	Dissatisfied
1	84	86	Other	Telehealth	Dissatisfied
2	124	52	Male	Telehealth	Dissatisfied
3	145	37	Female	Telehealth	Dissatisfied
4	157	71	Female	Telehealth	Dissatisfied
...	...	...	...	...	...
79	1857	76	Male	Telehealth	Dissatisfied
80	1878	40	Male	Telehealth	Dissatisfied
81	1894	21	Other	Telehealth	Dissatisfied
82	1931	85	Male	Telehealth	Dissatisfied
83	1948	86	Male	Telehealth	Dissatisfied

84 rows × 5 columns

How many Telehealth visits involved patients who were 'Satisfied' with the doctor?

Implementation:

```
in_doctor_level = 'Satisfied'
in_visit_type = 'Telehealth'

# Use a text CALL statement and pass parameters
sql = text("CALL GetVisitTypeDoctorInfo(:in_doctor_level, :in_visit_type)")
df = pd.read_sql_query(sql, engine, params={
    "in_doctor_level": in_doctor_level,
    "in_visit_type": in_visit_type
})
df
```

OUTPUT:

	Patient_ID	Age	Gender	Visit_Type	Doctor_Level
0	11	48	Female	Telehealth	Satisfied
1	39	38	Other	Telehealth	Satisfied
2	163	20	Other	Telehealth	Satisfied
3	167	32	Male	Telehealth	Satisfied
4	168	40	Male	Telehealth	Satisfied
...	...	...	...	...	...
93	1876	37	Other	Telehealth	Satisfied
94	1879	78	Other	Telehealth	Satisfied
95	1965	24	Male	Telehealth	Satisfied
96	1974	52	Other	Telehealth	Satisfied
97	1993	23	Other	Telehealth	Satisfied

98 rows × 5 columns

How many Telehealth visits involved patients who were 'Very Satisied' with the doctor?

Implementation:

```
in_doctor_level = 'Very Satisfied'
in_visit_type = 'Telehealth'

# Use a text CALL statement and pass parameters
sql = text("CALL GetVisitTypeDoctorInfo(:in_doctor_level, :in_visit_type)")
df = pd.read_sql_query(sql, engine, params={
    "in_doctor_level": in_doctor_level,
    "in_visit_type": in_visit_type
})
df
```

OUTPUT:

	Patient_ID	Age	Gender	Visit_Type	Doctor_Level
0	9	20	Female	Telehealth	Very Satisfied
1	13	46	Female	Telehealth	Very Satisfied
2	22	33	Female	Telehealth	Very Satisfied
3	68	20	Other	Telehealth	Very Satisfied
4	104	49	Female	Telehealth	Very Satisfied
...	...	...	...	...	...
96	1824	58	Other	Telehealth	Very Satisfied
97	1847	22	Other	Telehealth	Very Satisfied
98	1870	88	Female	Telehealth	Very Satisfied
99	1891	52	Other	Telehealth	Very Satisfied
100	1910	25	Other	Telehealth	Very Satisfied

101 rows × 5 columns

How many Inpatient visits involved patients who were 'Very Satisied' with the doctor?

Implementation:

```
in_doctor_level = 'Very Satisfied'
in_visit_type = 'Inpatient'

# Use a text CALL statement and pass parameters
sql = text("CALL GetVisitTypeDoctorInfo(:in_doctor_level, :in_visit_type)")
df = pd.read_sql_query(sql, engine, params={
    "in_doctor_level": in_doctor_level,
    "in_visit_type": in_visit_type
})
df
```

OUTPUT:

	Patient_ID	Age	Gender	Visit_Type	Doctor_Level
0	18	88	Male	Inpatient	Very Satisfied
1	42	40	Female	Inpatient	Very Satisfied
2	75	80	Other	Inpatient	Very Satisfied
3	95	27	Other	Inpatient	Very Satisfied
4	96	35	Female	Inpatient	Very Satisfied
...	...	...	...	...	...
86	1877	21	Female	Inpatient	Very Satisfied
87	1890	88	Other	Inpatient	Very Satisfied
88	1938	53	Male	Inpatient	Very Satisfied
89	1973	23	Female	Inpatient	Very Satisfied
90	1997	23	Female	Inpatient	Very Satisfied

91 rows × 5 columns

How many Inpatient visits involved patients who were 'Dissatisied' with the doctor?

Implementation:

```
in_doctor_level = 'Dissatisfied'
in_visit_type = 'Inpatient'

# Use a text CALL statement and pass parameters
sql = text("CALL GetVisitTypeDoctorInfo(:in_doctor_level, :in_visit_type)")
df = pd.read_sql_query(sql, engine, params={
    "in_doctor_level": in_doctor_level,
    "in_visit_type": in_visit_type
})
df
```

OUTPUT:

	Patient_ID	Age	Gender	Visit_Type	Doctor_Level
0	7	24	Female	Inpatient	Dissatisfied
1	15	38	Male	Inpatient	Dissatisfied
2	16	71	Other	Inpatient	Dissatisfied
3	53	66	Other	Inpatient	Dissatisfied
4	58	44	Male	Inpatient	Dissatisfied
...	...	...	...	...	...
97	1884	27	Female	Inpatient	Dissatisfied
98	1912	42	Male	Inpatient	Dissatisfied
99	1946	49	Female	Inpatient	Dissatisfied
100	1956	24	Other	Inpatient	Dissatisfied
101	1962	90	Male	Inpatient	Dissatisfied

102 rows × 5 columns

How many Outpatient visits involved patients who were 'Very Satisied' with the doctor?

Implementation:

```
in_doctor_level = 'Very Satisfied'
in_visit_type = 'Outpatient'

# Use a text CALL statement and pass parameters
sql = text("CALL GetVisitTypeDoctorInfo(:in_doctor_level, :in_visit_type)")
df = pd.read_sql_query(sql, engine, params={
    "in_doctor_level": in_doctor_level,
    "in_visit_type": in_visit_type
})
df
```

OUTPUT:

	Patient_ID	Age	Gender	Visit_Type	Doctor_Level
0	17	46	Female	Outpatient	Very Satisfied
1	35	33	Other	Outpatient	Very Satisfied
2	50	38	Other	Outpatient	Very Satisfied
3	62	85	Female	Outpatient	Very Satisfied
4	73	66	Female	Outpatient	Very Satisfied
...	...	...	...	...	...
87	1887	26	Male	Outpatient	Very Satisfied
88	1909	24	Male	Outpatient	Very Satisfied
89	1915	81	Female	Outpatient	Very Satisfied
90	1955	69	Female	Outpatient	Very Satisfied
91	1963	50	Female	Outpatient	Very Satisfied

92 rows × 5 columns

How many Outpatient visits involved patients who were 'Dissatisied' with the doctor?

Implementation:

```
in_doctor_level = 'DisSatisfied'
in_visit_type = 'Outpatient'

# Use a text CALL statement and pass parameters
sql = text("CALL GetVisitTypeDoctorInfo(:in_doctor_level, :in_visit_type)")
df = pd.read_sql_query(sql, engine, params={
    "in_doctor_level": in_doctor_level,
    "in_visit_type": in_visit_type
})
df
```

OUTPUT:

	Patient_ID	Age	Gender	Visit_Type	Doctor_Level
0	6	38	Male	Outpatient	Dissatisfied
1	12	77	Other	Outpatient	Dissatisfied
2	24	80	Other	Outpatient	Dissatisfied
3	40	46	Other	Outpatient	Dissatisfied
4	67	31	Male	Outpatient	Dissatisfied
...	...	...	...	...	...
101	1881	85	Female	Outpatient	Dissatisfied
102	1900	64	Male	Outpatient	Dissatisfied
103	1920	64	Other	Outpatient	Dissatisfied
104	1922	77	Other	Outpatient	Dissatisfied
105	1985	74	Female	Outpatient	Dissatisfied

106 rows × 5 columns

How many Emergency visits involved patients who were 'Very Satisied' with the nurse?

SQL:

```
DELIMITER $$
CREATE PROCEDURE GetVisitTypeNurseInfo (
    IN in_nurse_level VARCHAR(50),
    IN in_visit_type VARCHAR(50)
)
BEGIN
SELECT Patient_ID, Age,Gender,Visit_Type,Nurse_Level FROM patient_survey
WHERE Nurse_Level = in_nurse_level
AND Visit_Type = in_visit_type;
END $$
```

DELIMITER $$

Python:

```
import pandas as pd
from sqlalchemy import create_engine, text

engine = create_engine("mysql+pymysql://root:kalpana223@localhost/patient_satisfaction_survey_analysis")
in_nurse_level = 'Very Satisfied'
in_visit_type = 'Emergency'

# Use a text CALL statement and pass parameters
sql = text("CALL GetVisitTypeNurseInfo(:in_nurse_level, :in_visit_type)")
df = pd.read_sql_query(sql, engine, params={
    "in_nurse_level": in_nurse_level,
    "in_visit_type": in_visit_type
})
df
```

OUTPUT:

	Patient_ID	Age	Gender	Visit_Type	Nurse_Level
0	5	22	Other	Emergency	Very Satisfied
1	23	26	Male	Emergency	Very Satisfied
2	32	22	Female	Emergency	Very Satisfied
3	106	60	Female	Emergency	Very Satisfied
4	115	73	Female	Emergency	Very Satisfied
...	...	...	...	...	...
88	1961	29	Other	Emergency	Very Satisfied
89	1979	23	Other	Emergency	Very Satisfied
90	1987	81	Female	Emergency	Very Satisfied
91	1992	82	Male	Emergency	Very Satisfied
92	1999	83	Male	Emergency	Very Satisfied

93 rows × 5 columns

How many Emergency visits involved patients who were 'Dissatisied' with the nurse?

Implementation:

```
in_nurse_level = 'DisSatisfied'
in_visit_type = 'Emergency'

# Use a text CALL statement and pass parameters
sql = text("CALL GetVisitTypeNurseInfo(:in_nurse_level, :in_visit_type)")
df = pd.read_sql_query(sql, engine, params={
    "in_nurse_level": in_nurse_level,
    "in_visit_type": in_visit_type
})
df
```

OUTPUT:

	Patient_ID	Age	Gender	Visit_Type	Nurse_Level
0	2	67	Other	Emergency	Dissatisfied
1	27	32	Male	Emergency	Dissatisfied
2	28	29	Female	Emergency	Dissatisfied
3	64	41	Male	Emergency	Dissatisfied
4	80	85	Female	Emergency	Dissatisfied
...	...	...	...	...	...
97	1914	49	Male	Emergency	Dissatisfied
98	1929	36	Male	Emergency	Dissatisfied
99	1958	51	Female	Emergency	Dissatisfied
100	1960	59	Other	Emergency	Dissatisfied
101	1967	33	Other	Emergency	Dissatisfied

102 rows × 5 columns

How many Outpatient visits involved patients who were 'Dissatisied' with the nurse?

Implementation:

```
in_nurse_level = 'DisSatisfied'
in_visit_type = 'Outpatient'

# Use a text CALL statement and pass parameters
sql = text("CALL GetVisitTypeNurseInfo(:in_nurse_level, :in_visit_type)")
df = pd.read_sql_query(sql, engine, params={
    "in_nurse_level": in_nurse_level,
    "in_visit_type": in_visit_type
})
df
```

OUTPUT:

	Patient_ID	Age	Gender	Visit_Type	Nurse_Level
0	46	23	Female	Outpatient	Dissatisfied
1	56	57	Female	Outpatient	Dissatisfied
2	69	26	Other	Outpatient	Dissatisfied
3	85	26	Other	Outpatient	Dissatisfied
4	89	21	Female	Outpatient	Dissatisfied
...	...	...	...	...	...
87	1897	49	Male	Outpatient	Dissatisfied
88	1906	66	Female	Outpatient	Dissatisfied
89	1966	87	Female	Outpatient	Dissatisfied
90	1969	53	Other	Outpatient	Dissatisfied
91	1970	51	Male	Outpatient	Dissatisfied

92 rows × 5 columns

How many Outpatient visits involved patients who were 'Very satisied' with the nurse?

Implementation:

```
in_nurse_level = 'Very Satisfied'
in_visit_type = 'Outpatient'

# Use a text CALL statement and pass parameters
sql = text("CALL GetVisitTypeNurseInfo(:in_nurse_level, :in_visit_type)")
df = pd.read_sql_query(sql, engine, params={
    "in_nurse_level": in_nurse_level,
    "in_visit_type": in_visit_type
})
df
```

OUTPUT:

	Patient_ID	Age	Gender	Visit_Type	Nurse_Level
0	6	38	Male	Outpatient	Very Satisfied
1	40	46	Other	Outpatient	Very Satisfied
2	57	77	Other	Outpatient	Very Satisfied
3	63	66	Other	Outpatient	Very Satisfied
4	67	31	Male	Outpatient	Very Satisfied
...	...	...	...	...	...
97	1861	78	Male	Outpatient	Very Satisfied
98	1909	24	Male	Outpatient	Very Satisfied
99	1916	79	Other	Outpatient	Very Satisfied
100	1926	33	Male	Outpatient	Very Satisfied
101	1957	51	Female	Outpatient	Very Satisfied

102 rows × 5 columns

How many Inpatient visits involved patients who were 'Very satisied' with the nurse?

Implementation:

```
in_nurse_level = 'Very Satisfied'
in_visit_type = 'Inpatient'

# Use a text CALL statement and pass parameters
sql = text("CALL GetVisitTypeNurseInfo(:in_nurse_level, :in_visit_type)")
df = pd.read_sql_query(sql, engine, params={
    "in_nurse_level": in_nurse_level,
    "in_visit_type": in_visit_type
})
df
```

OUTPUT:

	Patient_ID	Age	Gender	Visit_Type	Nurse_Level
0	15	38	Male	Inpatient	Very Satisfied
1	31	52	Female	Inpatient	Very Satisfied
2	66	63	Other	Inpatient	Very Satisfied
3	75	80	Other	Inpatient	Very Satisfied
4	81	22	Other	Inpatient	Very Satisfied
...	...	...	...	...	...
84	1907	19	Male	Inpatient	Very Satisfied
85	1939	34	Other	Inpatient	Very Satisfied
86	1943	40	Other	Inpatient	Very Satisfied
87	1986	44	Male	Inpatient	Very Satisfied
88	1990	27	Male	Inpatient	Very Satisfied

89 rows × 5 columns

How many Inpatient visits involved patients who were 'Dissatisied' with the nurse?

Implementation:

```
in_nurse_level = 'DisSatisfied'
in_visit_type = 'Inpatient'

# Use a text CALL statement and pass parameters
sql = text("CALL GetVisitTypeNurseInfo(:in_nurse_level, :in_visit_type)")
df = pd.read_sql_query(sql, engine, params={
    "in_nurse_level": in_nurse_level,
    "in_visit_type": in_visit_type
})
df
```

OUTPUT:

	Patient_ID	Age	Gender	Visit_Type	Nurse_Level
0	4	80	Male	Inpatient	Dissatisfied
1	21	20	Female	Inpatient	Dissatisfied
2	36	52	Other	Inpatient	Dissatisfied
3	42	40	Female	Inpatient	Dissatisfied
4	55	35	Female	Inpatient	Dissatisfied
...	...	...	...	...	...
89	1838	40	Male	Inpatient	Dissatisfied
90	1846	26	Female	Inpatient	Dissatisfied
91	1858	57	Other	Inpatient	Dissatisfied
92	1918	43	Female	Inpatient	Dissatisfied
93	1946	49	Female	Inpatient	Dissatisfied

94 rows × 5 columns

Total Howmany Patients are completely satisfied with Hospital?

Implementation:

SQL:

```
SELECT COUNT(*) AS very_satisfied_patients
FROM patient_survey
WHERE Doctor_Rating    >= 5
  AND Nurse_Rating     >= 5
  AND Pharmacy_Rating  >= 5
  AND Admin_Rating     >= 5
  AND Fee_Structure_Rating >= 5;
```

Python:

```
import pandas as pd
Very_satisfied_Patients=df[(df['Doctor_Rating']>=5) & (df['Nurse_Rating']>=5) & (df['Pharmacy_Rating']>=5) & (df['Admin_Rating']>=5) & (df['Fee_Structure
print("Overall Very Satisfied Patients:")
len(Very_satisfied_Patients)
```

OUTPUT:

```
Overall Very Satisfied Patients:
151
```

Explanation:

- Each condition like df['Doctor_Rating'] >= 5 checks if the patient's rating in that column is 5 or above.
- &: Logical AND — all conditions must be true for the row to be selected.
- The result: only those rows (patients) where all ratings (Doctor, Nurse, Pharmacy, Admin, Fee Structure) are 5 or higher.
- Calculates and returns the **total number of very satisfied patients**.

This code counts patients who gave a rating of **5 or more** to **all service aspects** — doctors, nurses, pharmacy, administration, and fee structure.

What percentage of all patient visits involved individuals who rated all five service categories?

Implementation:

SQL:

```
SELECT
  COUNT(*) AS very_satisfied_patients,
  ROUND(COUNT(*) * 100.0 / 2000, 2) AS very_satisfied_percentage
FROM patient_survey
WHERE Doctor_Rating >= 5
  AND Nurse_Rating  >= 5
  AND Pharmacy_Rating  >= 5
  AND Admin_Rating   >= 5
  AND Fee_Structure_Rating >= 5;
```

Python:

```
df1=len(Very_satisfied_Patients)/2000 *100
print("Overall_Very_Satisfied_Patients_percentage:",df1)
```

OUTPUT:

```
Overall_Very_Satisfied_Patients_percentage: 7.55
```

How many patients rated the doctor's treatment as 'satisfied' or better (ratings between 9 and 10)?

Implementation:

SQL:

```
SELECT COUNT(*) AS patients_Satisfied_with_Doctor_Treatment
FROM patient_survey
WHERE Doctor_Rating BETWEEN 9 AND 10;
```

Python:

```
df1=sum(df['Doctor_Rating'].between(9,10))
print("Total Number Of Patients Satisfied With Doctors Treatment:",df1)
```

OUTPUT:

```
Total Number Of Patients Satisfied With Doctors Treatment: 390
```

Explanation:

- The code identifies patients who rated doctors between 9 and 10.
- between(9, 10) checks if the rating falls within the specified range.
- sum(...) counts how many patients gave such high ratings.
- The result is stored in the variable df1.
- Finally, it prints the total number of highly satisfied patients with doctors.

How many patients rated the doctor's treatment as 'Dissatisfied'(ratings between 0 and 2)?

Implementation:

SQL:

SELECT COUNT(*) AS patient_Dissatisfied

FROM patient_survey

WHERE Doctor_Rating BETWEEN 0 AND 2;

Python:

```
df1=sum(df['Doctor_Rating'].between(0,2))
print("Total Number Of Patients Dissatisfied with Doctors Treatment:",df1)
```

OUTPUT:

```
Total Number Of Patients Dissatisfied with Doctors Treatment: 420
```

Explanation:

- The code checks for patients who rated their doctor between **0 and 2**.
- **df['Doctor_Rating'].between(0, 2)** returns True for those who rated in this range.
- **sum(...)** counts how many patients gave low ratings (0-2).
- The result is stored in the variable df1.
- It then prints the total number of **dissatisfied patients** with their doctor's treatment.

How many patients rated the Nurse treatment as 'Satisfied'(ratings between 9 and 10)?

Implementation:

Python:

```
df1=sum(df['Nurse_Rating'].between(9,10))
print("Total Number Of Patients Satisfied With Nurse Treatment:",df1)
```

OUTPUT:

```
Total Number Of Patients Satisfied With Nurse Treatment: 387
```

SQL:

```
SELECT COUNT(*) AS satisfied_with_nurse
FROM patient_survey
WHERE Nurse_Rating BETWEEN 9 AND 10;
```

How many patients rated the Nurse treatment as 'Dissatisfied'(ratings between 0 and 2)?

Implementation:

Python:

```
df1=sum(df['Nurse_Rating'].between(0,2))
print("Total Number Of Patients Dissatisfied with Nurse Treatment:",df1)
```

OUTPUT:

```
Total Number Of Patients Dissatisfied with Nurse Treatment: 415
```

SQL:

```
SELECT COUNT(*) AS patient_Dissatisfied
FROM patient_survey
WHERE Nurse_Rating BETWEEN 0 AND 2;
```

How many patients rated the Pharmacy treatment as 'Satisfied'(ratings between 9 and 10)?

Implementation:

Python:

```
df1=sum(df['Pharmacy_Rating'].between(9,10))
print("Total Number Of Patients Satisfied With Pharmacy:",df1)
```

OUTPUT:

```
Total Number Of Patients Satisfied With Pharmacy: 407
```

SQL:

SELECT COUNT(*) AS satisfied_with_Pharmacy

FROM patient_survey

WHERE Pharmacy_Rating BETWEEN 9 AND 10;

How many patients rated the Pharmacy treatment as 'Dissatisfied'(ratings between 0 and 2)?

Implementation:

Python:

```
df1=sum(df['Pharmacy_Rating'].between(0,2))
print("Total Number Of Patients Dissatisfied With Pharmacy:",df1)
```

OUTPUT:

```
Total Number Of Patients Dissatisfied With Pharmacy: 391
```

SQL:

SELECT COUNT(*) AS patient_Dissatisfied

FROM patient_survey

WHERE Pharmacy_Rating BETWEEN 0 AND 2;

How many patients rated the Admin treatment as 'Satisfied'(ratings between 9 and 10)?

Implementation:

Python:

```
df1=sum(df['Admin_Rating'].between(9,10))
print("Total Number Of Patients Satisfied With Admin:",df1)
```

OUTPUT:

```
Total Number Of Patients Satisfied With Admin: 401
```

SQL:

SELECT COUNT(*) AS satisfied_with_Admin

FROM patient_survey

WHERE Admin_Rating BETWEEN 9 AND 10;

How many patients rated the Admin treatment as 'Dissatisfied'(ratings between 0 and 2)?

Implementation:

Python:

```
df1=sum(df['Admin_Rating'].between(0,2))
print("Total Number Of Patients Dissatisfied With Admin Department:",df1)
```

OUTPUT:

```
Total Number Of Patients Dissatisfied With Admin Dpartment: 406
```

SQL:

SELECT COUNT(*) AS patient_Dissatisfied

FROM patient_survey

WHERE Admin_Rating BETWEEN 0 AND 2;

How can I calculate the overall satisfaction percentage across all these categories for each entry in the DataFrame?

Implementation:

```
import numpy as np
import pandas as pd
df['is_doctor_satisfied'] = np.where(df['Doctor_Rating'] >= 7, 'yes', 'no')
df['is_nurse_satisfied'] = np.where(df['Nurse_Rating'] >= 7, 'yes', 'no')
df['is_pharmacy_satisfied'] = np.where(df['Pharmacy_Rating'] >= 7, 'yes', 'no')
df['is_admin_satisfied'] = np.where(df['Admin_Rating'] >= 7, 'yes', 'no')
df['is_fee_satisfied'] = np.where(df['Fee_Structure_Rating'] >= 7, 'yes', 'no')
df
```

Explanation:

np.where(condition, 'yes', 'no'):

- If the rating is ≥ 7, mark it as 'yes' (satisfied).
- Otherwise, mark it as 'no' (not satisfied).

A patient is considered satisfied with a service if they gave a rating of 7 or above. This binary classification helps in identifying areas with low satisfaction and improving healthcare quality accordingly.

OUTPUT:

is_doctor_satisfied	is_nurse_satisfied	is_pharmacy_satisfied	is_admin_satisfied	is_fee_satisfied
no	no	no	no	no
yes	no	yes	no	no
no	yes	yes	yes	no
yes	no	yes	no	yes
no	yes	no	no	yes
...	...	...	...	...
no	no	yes	no	no
yes	yes	no	no	no
no	no	no	no	no
yes	yes	no	yes	no
yes	no	yes	no	yes

SQL:

SELECT

```
*,
 CASE WHEN Doctor_Rating >= 7 THEN 'yes' ELSE 'no' END AS is_doctor_satisfied,
 CASE WHEN Nurse_Rating >= 7 THEN 'yes' ELSE 'no' END AS is_nurse_satisfied,
 CASE WHEN Pharmacy_Rating >= 7 THEN 'yes' ELSE 'no' END AS is_pharmacy_satisfied,
 CASE WHEN Admin_Rating >= 7 THEN 'yes' ELSE 'no' END AS is_admin_satisfied,
 CASE WHEN Fee_Structure_Rating >= 7 THEN 'yes' ELSE 'no' END AS is_fee_satisfied
FROM patient_survey;
```

Thinking about your experience today, were you satisfied with at least three out of these five areas—doctor service, nurse service, pharmacy, admin support, and fees?

Implementation:

```
bool_cols = [
    'is_doctor_satisfied',
    'is_nurse_satisfied',
    'is_pharmacy_satisfied',
    'is_admin_satisfied',
    'is_fee_satisfied'
]
df[bool_cols] = df[bool_cols].map(lambda x: str(x).strip().lower() == 'yes')

# Count how many "Yes" per row
df['num_satisfied'] = df[bool_cols].sum(axis=1)

# Overall satisfaction flag
df['overall_satisfied'] = df['num_satisfied'] >= 3
df
```

Explanation:

Converts each column value to a boolean:

- If the value is 'yes' (case-insensitive, ignoring spaces), it becomes True.
- Otherwise, it becomes False.
- Adds a new column num_satisfied that counts how many True values (i.e., "Yes") each patient gave across the 5 services.
- Marks a patient as **overall satisfied (True)** if they were satisfied with **3 or more services**.

OUTPUT:

is_doctor_satisfied	is_nurse_satisfied	is_pharmacy_satisfied	is_admin_satisfied	is_fee_satisfied	num_satisfied	overall_satisfied
False	False	False	False	False	0	False
True	False	True	False	False	2	False
False	True	True	True	False	3	True
True	False	True	False	True	3	True
False	True	False	False	True	2	False
...	...	...	...	...	...	...
False	False	True	False	False	1	False
True	True	False	False	False	2	False
False	False	False	False	False	0	False
True	True	False	True	False	3	True
True	False	True	False	True	3	True

SQL:

```
SELECT *,
(
   CASE WHEN Doctor_Rating         >= 7 THEN 1 ELSE 0 END +
   CASE WHEN Nurse_Rating          >= 7 THEN 1 ELSE 0 END +
   CASE WHEN Pharmacy_Rating       >= 7 THEN 1 ELSE 0 END +
   CASE WHEN Admin_Rating          >= 7 THEN 1 ELSE 0 END +
   CASE WHEN Fee_Structure_Rating >= 7 THEN 1 ELSE 0 END
 ) AS num_satisfied,

 CASE
   WHEN (
     CASE WHEN Doctor_Rating         >= 7 THEN 1 ELSE 0 END +
     CASE WHEN Nurse_Rating          >= 7 THEN 1 ELSE 0 END +
     CASE WHEN Pharmacy_Rating       >= 7 THEN 1 ELSE 0 END +
     CASE WHEN Admin_Rating          >= 7 THEN 1 ELSE 0 END +
     CASE WHEN Fee_Structure_Rating >= 7 THEN 1 ELSE 0 END
   ) >= 3 THEN 1
   ELSE 0
 END AS overall_satisfied
```

FROM patient_survey;

Among patients who are overall satisfied?

Implementation:

```
New_df=df[df['overall_satisfied']==True]
New_df
```

Explanation:

df['overall_satisfied'] == True:

- Creates a Boolean mask where only rows with True in the overall_satisfied column are selected.
- Applies that mask to return only satisfied patient records.
- Contains only the rows where patients are satisfied.

OUTPUT:

is_doctor_satisfied	is_nurse_satisfied	is_pharmacy_satisfied	is_admin_satisfied	is_fee_satisfied	num_satisfied	overall_satisfied
False	True	True	True	False	3	True
True	False	True	False	True	3	True
True	True	False	False	True	3	True
True	True	True	False	True	4	True
True	False	True	True	False	3	True
...	...	...	...	...	...	...
True	True	True	True	True	5	True
True	True	False	False	True	3	True
True	True	True	False	False	3	True
True	True	False	True	False	3	True
True	False	True	False	True	3	True

How many patients were overall satisfied with their experience?

Implementation:

```
Total_Satisfied=df['overall_satisfied'].sum()
print("Total Satisfied Patients:",Total_Satisfied)
```

Explanation:

df['overall_satisfied'].sum():

- Sums up all values in the overall_satisfied column.
- Since True is treated as 1 and False as 0, this gives the total count of True values, i.e., number of satisfied patients.

OUTPUT:

```
Total Satisfied Patients: 624
```

SQL:

```
SELECT
  SUM(CASE
      WHEN (CASE WHEN Doctor_Rating >= 7 THEN 1 ELSE 0 END
        + CASE WHEN Nurse_Rating >= 7 THEN 1 ELSE 0 END
        + CASE WHEN Pharmacy_Rating >= 7 THEN 1 ELSE 0 END
        + CASE WHEN Admin_Rating >= 7 THEN 1 ELSE 0 END
        + CASE WHEN Fee_Structure_Rating >= 7 THEN 1 ELSE 0 END) >= 3
      THEN 1 ELSE 0
    END) AS Total_Satisfied_Patients
FROM patient_survey;
```

What percentage of patients are overall satisfied?

Implementation:

```
Satisfaction_Rate=df['overall_satisfied'].mean()*100
print(f"Satisfaction Rate: {Satisfaction_Rate:2f}%")
```

Explanation:

df.groupby('Gender'):

df['overall_satisfied'].mean():

- Calculates the average of the overall_satisfied column.
- Since the column contains Boolean values (True as 1, False as 0), the mean gives the proportion of satisfied patients.
- Converts the proportion to a percentage.

OUTPUT:

```
Satisfaction Rate: 31.200000%
```

SQL:

```
SELECT
  100 * AVG(
    (
      (Doctor_Rating       >= 7)
    + (Nurse_Rating        >= 7)
    + (Pharmacy_Rating     >= 7)
    + (Admin_Rating        >= 7)
    + (Fee_Structure_Rating>= 7)
    ) >= 3
  ) AS Satisfaction_Rate_Percent
FROM patient_survey;
```

What is the average satisfaction percentage among different genders?

Implementation:

```
Gender_satisfaction = df.groupby('Gender')['overall_satisfied'].mean() * 100
print(Gender_satisfaction)
```

Explanation:

df.groupby('Gender'):

- Groups the data based on gender (e.g., Male, Female, Other).

['overall_satisfied'].mean():

- Calculates the mean of the overall_satisfied column for each gender.
- Since True = 1 and False = 0, the mean gives the proportion of satisfied patients.
- Converts that proportion into a percentage.

OUTPUT:

```
Gender
Female    32.674772
Male      29.046899
Other     31.864905
Name: overall_satisfied, dtype: float64
```

What is the average satisfaction percentage among different Visit_Types?

Implementation:

```
Visit_Type_satisfaction = df.groupby('Visit_Type')['overall_satisfied'].mean() * 100
print(Visit_Type_satisfaction)
```

Explanation:

df.groupby('Visit_Type'):

- Groups the data based on different visit types (e.g., Emergency, Routine Checkup, Follow-up).

['overall_satisfied'].mean():

- Computes the mean of the overall_satisfied column (which should contain boolean values: True or False) for each visit type.
- Since True is treated as 1 and False as 0, the mean gives the percentage of satisfied patients in that group.

OUTPUT:

```
Visit_Type
Emergency     32.283465
Inpatient     29.922780
Outpatient    31.313131
Telehealth    31.315240
Name: overall_satisfied, dtype: float64
```

What is the satisfaction rate (%) within each age group?

Implementation:

```python
import pandas as pd
bins = [0, 18, 35, 55, 100]
labels = ['0-17', '18-35', '36-55', '56+']
df['Age_group'] = pd.cut(df['Age'], bins=bins, labels=labels)
result = (df
          .groupby('Age_group')['overall_satisfied']
          .agg(total='count', satisfied='sum')
          .assign(percent_satisfied=lambda x: x['satisfied'] / x['total'] * 100))
print(result)
```

Explanation:

Groups data by age group.

Calculates:

- total: total number of people in each group
- satisfied: how many are satisfied (overall_satisfied == True)

Adds a new column percent_satisfied that gives the **percentage of satisfied people** in each age group.

OUTPUT:

Age_group	total	satisfied	percent_satisfied
0-17	31	5	16.129032
18-35	481	170	35.343035
36-55	524	150	28.625954
56+	964	299	31.016598

What is the overall distribution of doctor ratings, and what does it indicate about perceived doctor quality?

Implementation:

```python
import seaborn as sns
import matplotlib.pyplot as plt
sns.histplot(df['Doctor_Rating'],bins=10)
plt.title('Doctor Rating Distribution')
plt.show()
```

Explanation:

- Plots a histogram using Seaborn.

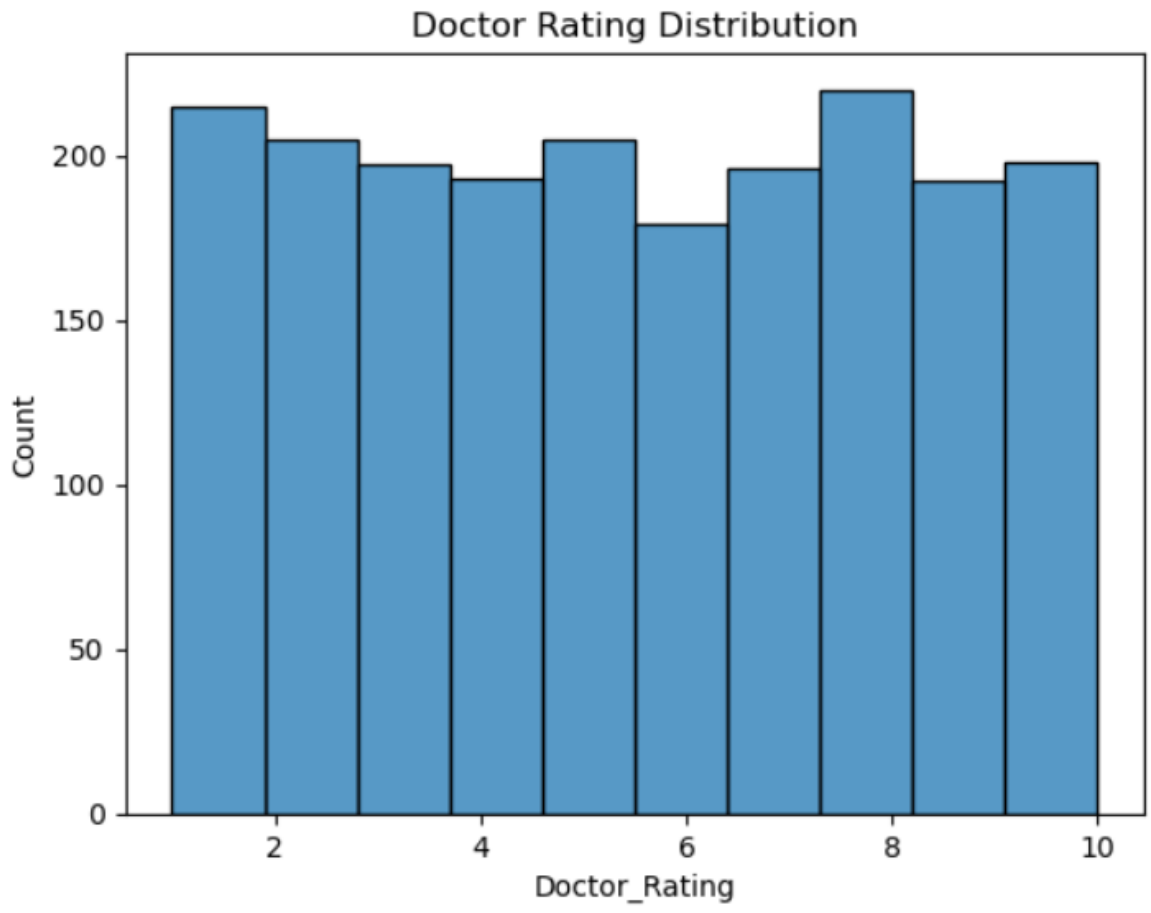
- df['Doctor_Rating']: The data being plotted is the Doctor_Rating column from your DataFrame.
- bins=10: The ratings are divided into 10 equal-width intervals (bins), and the number of ratings falling into each bin is counted and plotted.

Purpose of the Histogram:

A histogram is used to:

- Understand the distribution (e.g., normal, skewed) of numerical data.
- Identify common rating ranges (e.g., whether most ratings are 9–10 or spread out).
- Spot any data anomalies or outliers.

OUTPUT:



How do doctor ratings vary between different genders, and are there significant differences in patient perceptions?

Implementation:

```
sns.boxplot(x='Gender',y='Doctor_Rating',data=df,hue='Gender')
plt.title('Doctor Rating By Gender')
plt.show()
```

Explanation:

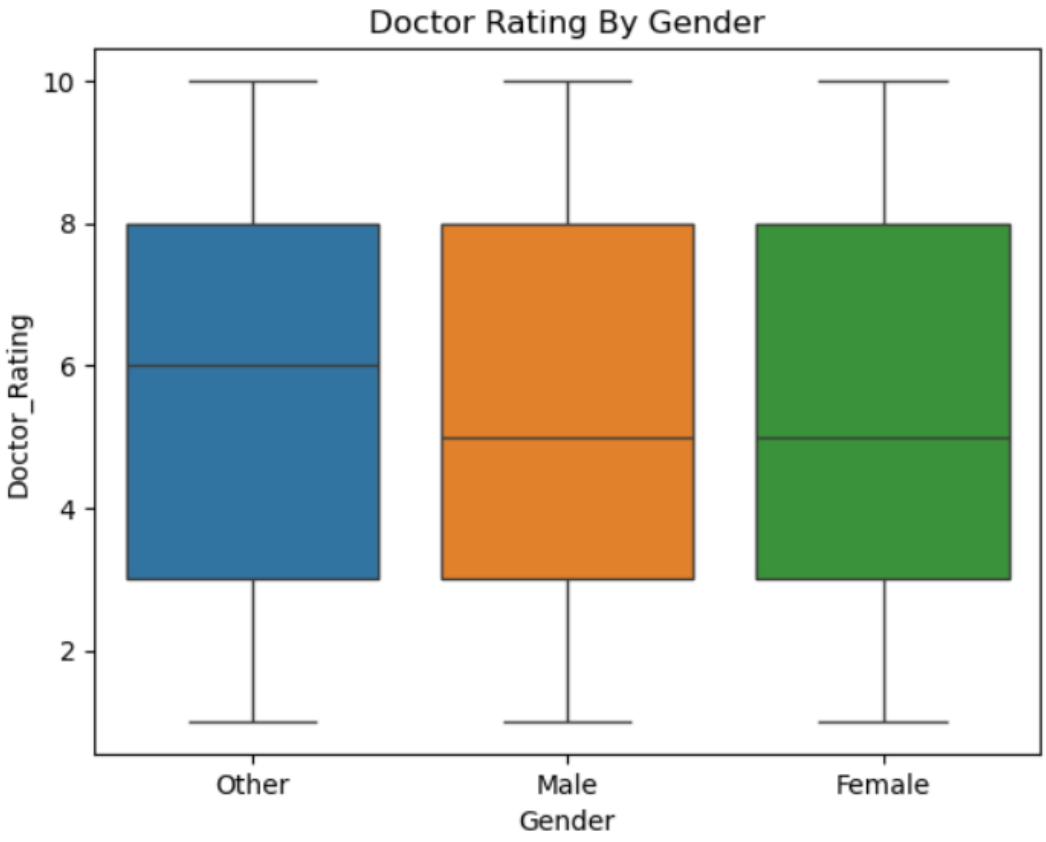
- sns.boxplot: Creates a box plot to show the distribution of Doctor_Rating grouped by Gender.•
- **Median:** (central line in the box)
- **Interquartile range (IQR)** :middle 50% of the data
- **Whiskers** –:range of data excluding outliers
- **Outliers** : individual points beyond whiskers

Why This Visualization Is Useful:

This plot helps compare how male and female patients rate doctors. You can observe:

- Which gender gives higher or lower ratings on average (check median lines).
- Which gender has more variation in ratings (box and whisker size).
- If there are more outliers in one gender group.

OUTPUT:



What relationships exist between various service ratings, and do higher ratings in one area correlate with others?

Implementation:

```
cols=['Doctor_Rating','Nurse_Rating','Pharmacy_Rating','Admin_Rating']
corr=df[cols].corr()
sns.heatmap(corr)
plt.title('Service Rating Correlation')
plt.show()
```

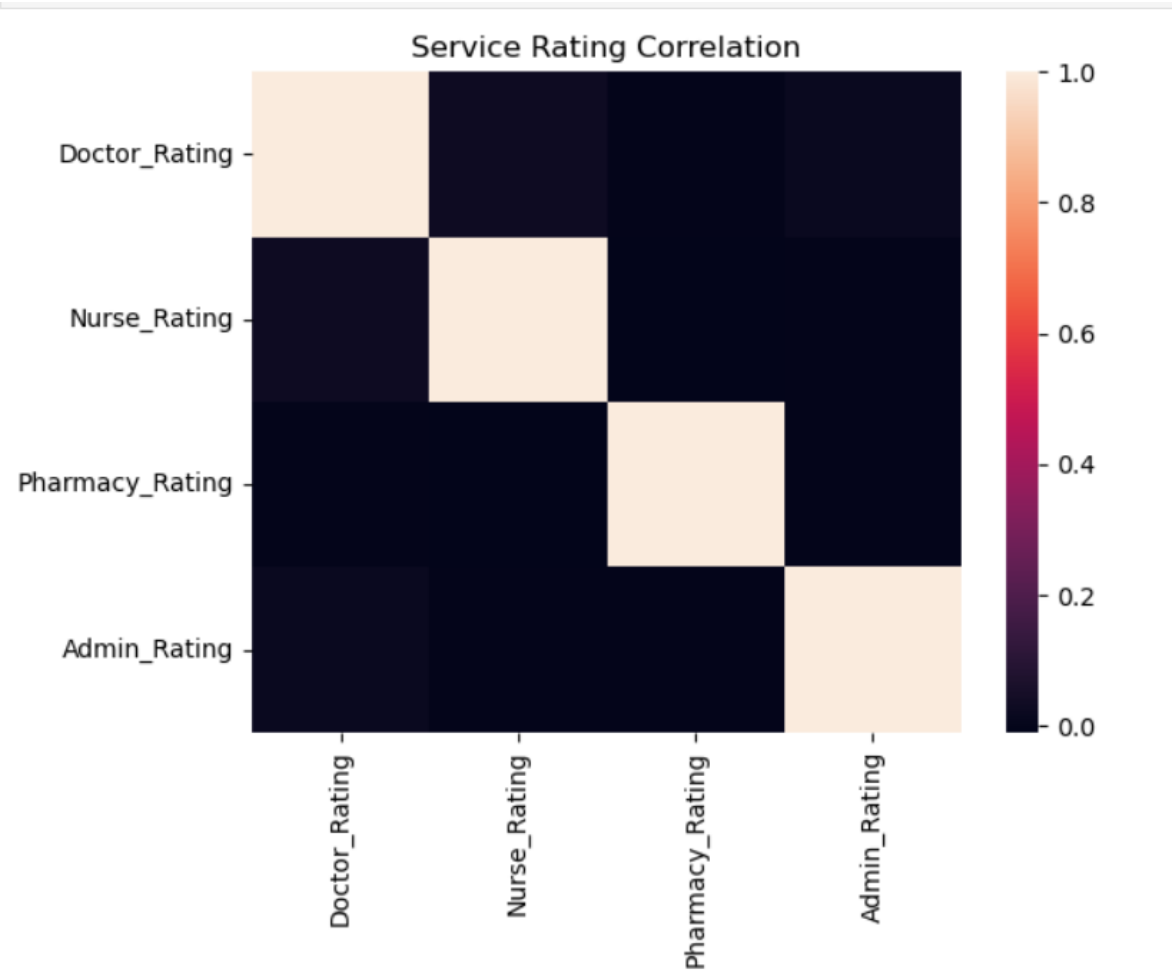
Explanation:

- sns.heatmap(corr) plots this correlation matrix as a heatmap, where each cell is color-coded according to correlation strength. Brighter or warmer colors highlight strong positive (or negative) relationships
- This matrix shows how strongly each pair of service ratings move together—values range from **+1** (perfect positive correlation) to **–1** (perfect negative), with **0** meaning no linear relationship

Why This Visualization Is Useful:

- **Identifies relationships**: You can instantly see if, for instance, satisfaction with nurses is strongly related to satisfaction with doctors or admin services.
- **Highlights multivariate patterns**: Clusters of high correlation indicate overlapping areas of patient perception—e.g., if pharmacy and admin ratings are highly correlated, improvements in one may impact perceptions of the other.
- **Pinpoints independent factors**: If some ratings show low correlation, those domains likely operate independently and merit separate improvement strategies.

OUTPUT:



How does the likelihood of patients recommending their doctor change across different doctor rating levels?

Implementation:

```
contigency=pd.crosstab(df['Doctor_Rating'],df['Doctor_Recommend'],normalize='index')
contigency.plot(kind='bar',stacked=True)
plt.ylabel('Proportion Recommend')
plt.title('Recommend Rate By Doctor Rating')
plt.show()
```

Explanation:

Build a contingency table:

- This produces a table where each row represents a rating category (e.g. 1–5).
- Each column corresponds to whether the patient would recommend the doctor (“Yes” or “No”).
- Using normalize='index' converts counts into row-wise proportions—so within each rating level, you get the share who said “Yes” vs. “No” to recommending

Plotting the stacked bar chart:

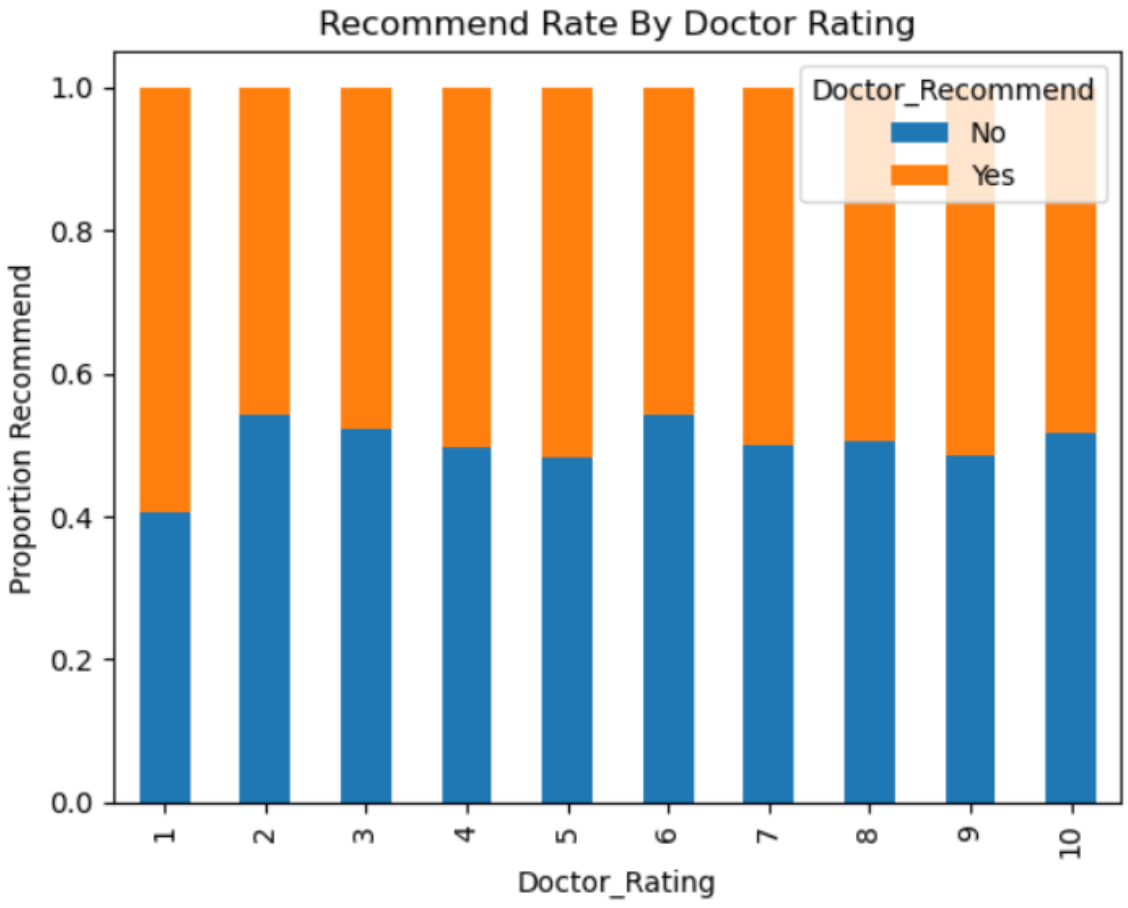
- Creates one bar per doctor rating level, divided into colored segments for "Recommend" or "Not Recommend".

- Since values are proportions (fractions of 1), they stack to fill each bar fully—showing relative shares per rating group

Why It Matters

- Allows you to compare **recommendation behavior** across different doctor rating levels.
- You can clearly see if **higher-rated doctors get more recommendations**, and how strong that trend is.
- The **100% stacked format** makes proportions easy to compare even if the number of patients per rating varies.
- Useful for identifying whether patients who give mid-range ratings are still likely to recommend or not.

OUTPUT:



What is the proportion of different visit types among patients, and which type is most common?

Implementation:

```
df['Visit_Type'].value_counts().plot(kind='pie',autopct='%1.1f%%',figsize=(10,5))
plt.title('Distribution of patients Visit Type')
plt.show()
```

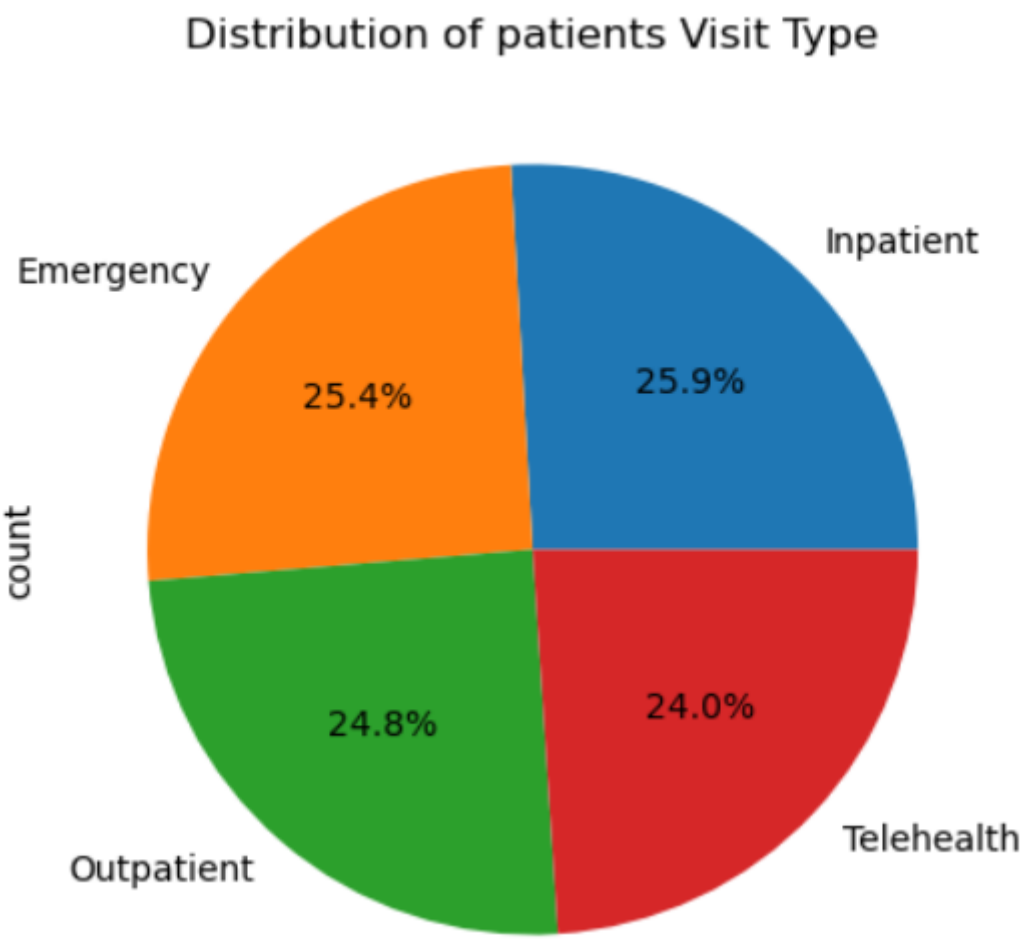
Explanation:

- Counts each visit type via df['Visit_Type'].value_counts(), producing a frequency count for categories like “inpatient” or “outpatient”.
- Generates a pie chart from those counts using .plot(kind='pie'), which wraps matplotlib.pyplot.pie() behind the scenes
- Displays percentages in each slice with autopct='%1.1f%%', formatting scores like “65.3%”.
- Sets figure size to 10×5 inches for clarity and layout balance using figsize=(10, 5).

Why Use This Pie Chart?

- It efficiently **visualizes the share of each visit type**—showing how the overall patient population is divided among types in a circular “part-to-whole” format
- Pie charts are particularly useful for **binary or limited categories**, such as a few distinct visit types = nominal data
- The slice proportions and labels make it easy to see which visit type predominates—especially helpful when comparing major categories.

OUTPUT:



How do doctor empathy scores differ across various visit types, and are certain visit types associated with higher empathy scores?

Implementation:

```
sns.boxenplot(data=df, x="Visit_Type", y="Doctor_Empathy")
plt.title("Doctor Empathy Scores Across Visit Types")
plt.show()
```

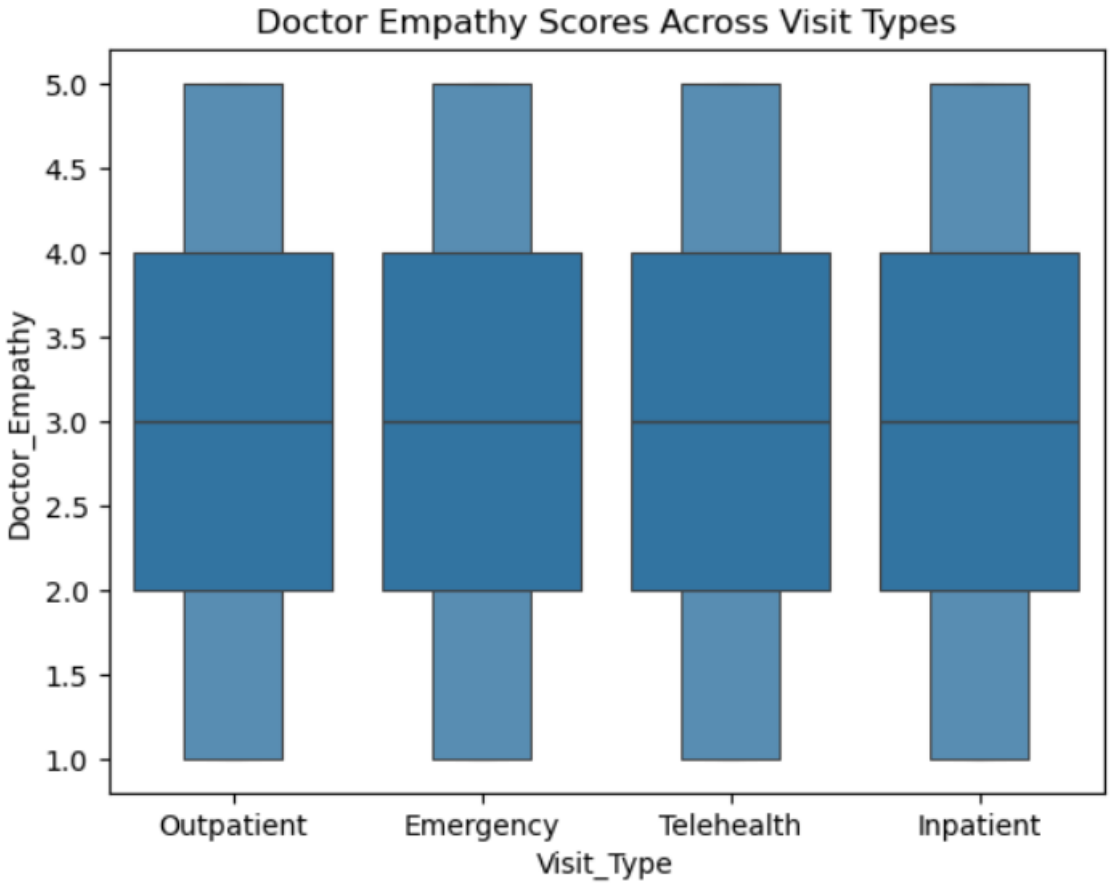
Explanation:

- x="Visit_Type" places visit categories (like “inpatient”, “outpatient”) along the horizontal axis.
- y="Doctor_Empathy" represents individual empathy scores on the vertical axis for each visit type.
- A box in the plot spans the interquartile range (IQR), which is from the 25th percentile (Q1) to the 75th percentile (Q3), capturing the middle 50% of responses.
- The horizontal line inside the box marks the median (50th percentile).
- Whiskers extend from the box to the smallest and largest values within 1.5 × IQR. Values beyond these are plotted individually as outliers

Why This Plot Is Helpful:

- It reveals the central tendency (median) and spread (IQR) of empathy scores across different visit types.
- Enables side-by-side comparison: You can easily see which visit type yields higher median empathy scores or more variability.
- Outliers highlight unusual responses, prompting further investigation.
- If one category's box is positioned higher and is narrower, it suggests consistently better empathy ratings for that visit type.

OUTPUT:



Are there notable differences in overall satisfaction scores between genders, and what might account for these differences?

Implementation:

```
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
sns.violinplot(x="Gender",y='Doctor_Recommend',data=New_df,hue='Gender', inner="quartile")

plt.title("Overall Satisfaction by Gender")
plt.xlabel("Gender")
plt.ylabel("Satisfaction Score")
plt.show()
```

Explanation:

Data structure

- Your DataFrame New_df contains records of patients with a numeric satisfaction score in the Doctor_Recommend column and a categorical grouping by Gender.

Main plot: violin plot

- Using sns.violinplot(), the code generates a separate violin for each gender category along the x-axis, showing the distribution of Doctor_Recommend scores.

Density curve and quartiles

- Each violin represents a kernel density estimate (KDE) of the data for that gender group—mapping the frequency of score values as a smooth, symmetrical "violin" shape. Wider parts mean more respondents at that value level.
- With inner="quartile", the plot overlays the first, second (median), and third quartile lines within the violin.

Why This Visualization Is Useful:

Distribution insight: Unlike simple bar charts showing means, violin plots reveal the full shape, spread, and density of satisfaction responses per gender.

Quartile information: Internal lines make it easy to see medians and IQR—the middle 50% range for each group—helping compare variability.

Comparative clarity: Side-by-side violins allow quick visual assessment of how response distributions differ between genders.

OUTPUT:

Which age groups report the highest satisfaction rates, and how does satisfaction vary across different age demographics?

Implementation:

```
import seaborn as sns
import matplotlib.pyplot as plt
sns.barplot(data=result, x='Age_group', y='percent_satisfied', hue='Age_group')
plt.ylabel('Percent Satisfied')
plt.xlabel('Age Group')
plt.title('Satisfaction Rate by Age Group')
plt.show()
```

Explanation:

Data structure

- result is a DataFrame where each row represents an age group and its corresponding percent_satisfied.

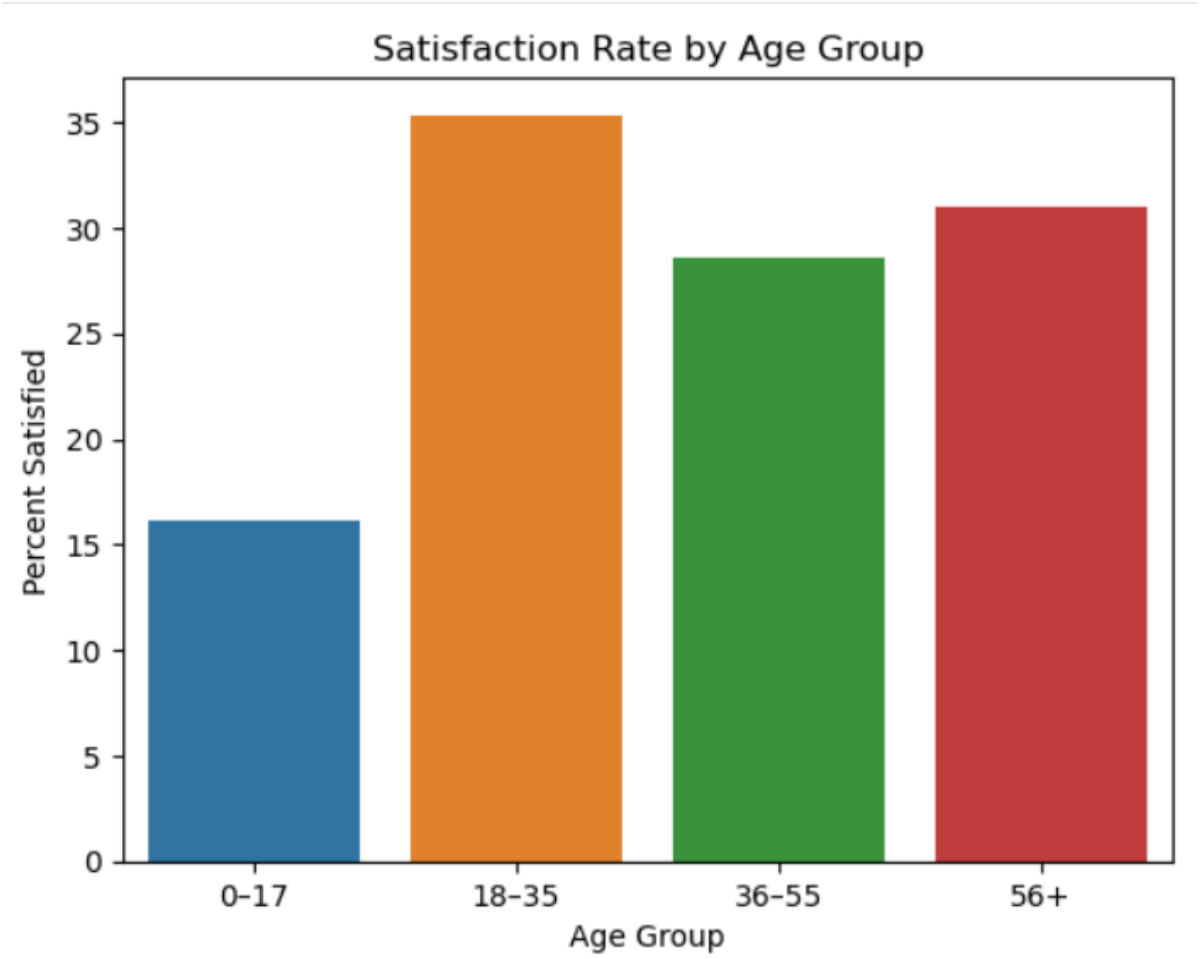
Plotting with sns.barplot()

- x='Age_group' puts age categories (e.g., 0–17, 18–35, etc.) on the x-axis.
- y='percent_satisfied' uses the satisfaction percentage as the bar height.
- hue='Age_group' applies distinct colors to each bar based on the age group, making them visually separable and easier to interpret.

What the bars represent

- By default, Seaborn calculates the mean of the percent_satisfied values per category (though here each category already has one value) and draws confidence-interval error bars using bootstrapping methods unless ci=None is specified

OUTPUT:



How does patient satisfaction differ among various visit types, and which visit type yields the highest satisfaction?

Implementation:

```
summary = Visit_Type_satisfaction.reset_index()
summary.columns = ['Visit_Type', 'percent_satisfied']
sns.barplot(
    data=summary,
    x='Visit_Type',
    y='percent_satisfied',
    hue='Visit_Type'
)
plt.ylabel('Percent Satisfied')
plt.xlabel('Visit Type')
plt.title('Satisfaction Rate by Visit Type')
```

Explanation:

1.Prepare the summary data

- Visit_Type_satisfaction.reset_index() converts the grouped summary into a DataFrame with index turned into a column.
- Then you rename columns to Visit_Type and percent_satisfied for clarity.

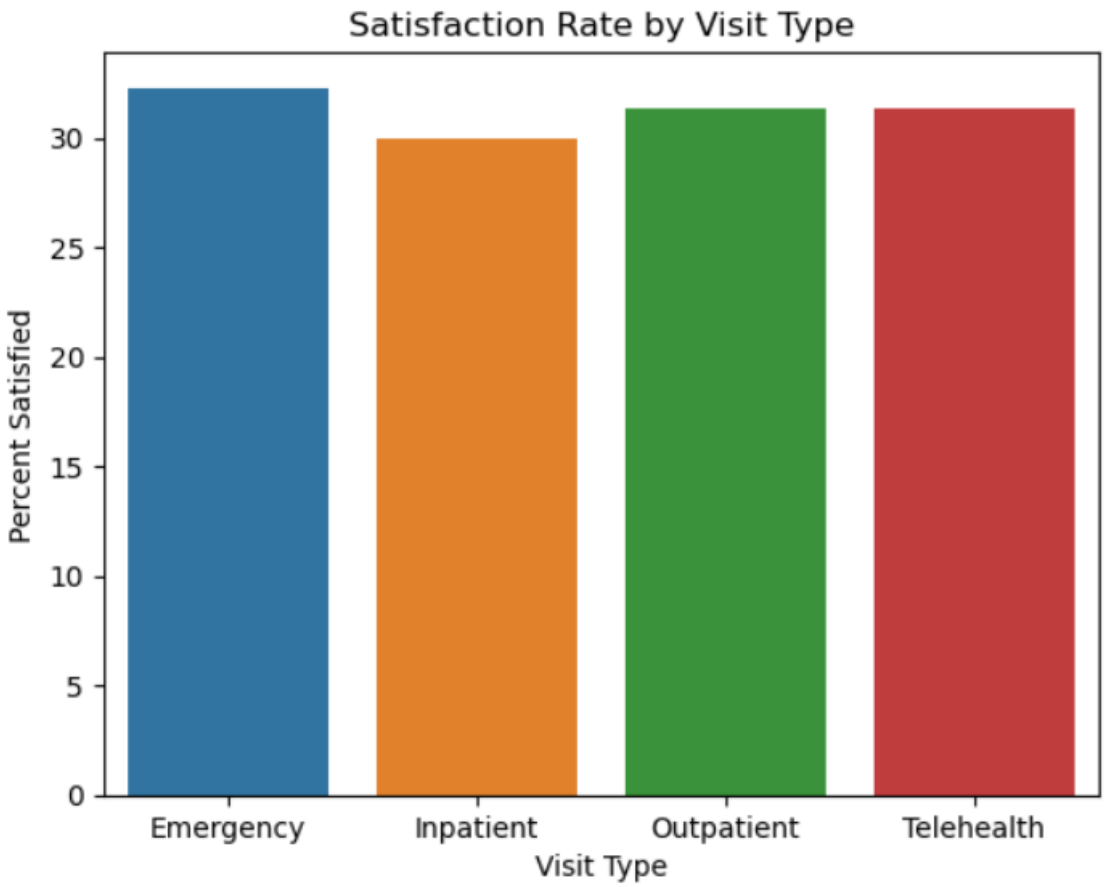
2.Plotting with Seaborn's barplot()

- x='Visit_Type': each unique visit type (e.g. 'inpatient', 'outpatient') is shown along the x-axis.
- y='percent_satisfied': the bar height represents the satisfaction rate for that visit type.
- hue='Visit_Type': adds color distinction to each category for emphasis and consistency.

Why this approach is helpful:

- Helps **compare satisfaction rates across visit types** visually and intuitively.
- Color-coded bars (using `hue`) enhance readability and highlight service types.

OUTPUT:



What is the overall patient satisfaction rate, and what percentage of patients report being satisfied versus unsatisfied?

Implementation:

```
counts = df['overall_satisfied'].value_counts().sort_index()
labels = ['Unsatisfied', 'Satisfied']
colors = ['salmon', 'skyblue']
plt.figure(figsize=(6, 6))
plt.pie(
    counts,
    labels=labels,
    colors=colors,
    autopct='%1.1f%%'
)
plt.title('Overall Patient Satisfaction')
plt.show()
```

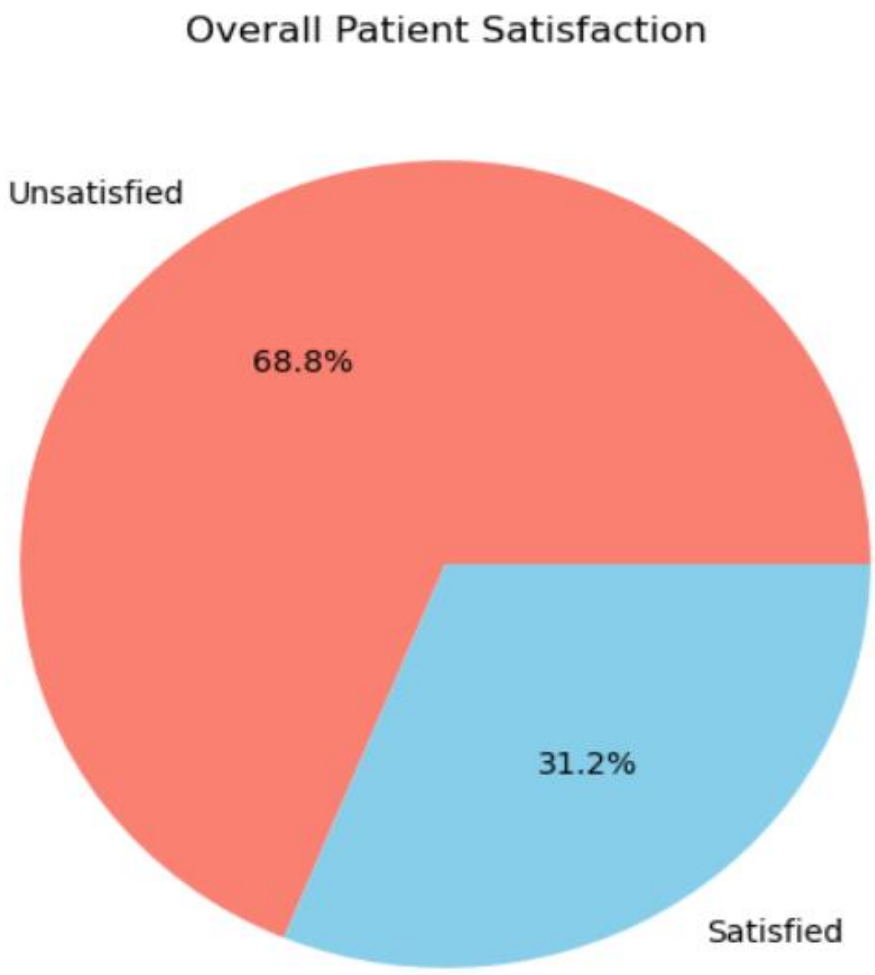
Explanation:

1. **Counting responses:**
   counts = df['overall_satisfied'].value_counts().sort_index() computes how many respondents marked each category (e.g. 0 for Unsatisfied, 1 for Satisfied), then orders them to match the labels.
2. **Setting labels & colors:**
   labels = ['Unsatisfied', 'Satisfied']: naming the two pie slices.colors = ['salmon', 'skyblue']: customizing the visual colour for each slice.
3. **Plotting the pie chart:**
   - plt.pie(counts, labels=labels, colors=colors, autopct='%1.1f%%') draws the chart.
   - The autopct='%1.1f%%' argument ensures that each slice shows its percentage of the whole, formatted with one decimal place—
4. **Title and display:**
   plt.title('Overall Patient Satisfaction') adds a title.plt.show() renders the chart.

Why Use This Visualization?

- A **pie chart provides a visual breakdown** of satisfied vs. unsatisfied respondents, making it easy to understand at a glance.
- The **percentage annotations** inside each slice (via autopct) allow you to quickly communicate the proportion of respondents in each satisfaction category without manual calculation.
- Using value_counts() ensures the data accurately reflects the categorical distribution in your survey.

OUTPUT:



Patient Satisfaction Survey Analysis Dashboard:

Implementation:

```
fig = plt.figure(figsize=(50, 26), constrained_layout=True)
ax1=fig.add_subplot(3,4,1)
sns.histplot(df['Doctor_Rating'],bins=10,ax=ax1)
ax1.set_title('Doctor Rating Distribution')
ax2=fig.add_subplot(3,4,2)
sns.boxplot(x='Gender',y='Doctor_Rating',data=df,hue='Gender',ax=ax2)
ax2.set_title('Doctor Rating By Gender')
ax3=fig.add_subplot(3,4,3)
cols=['Doctor_Rating','Nurse_Rating','Pharmacy_Rating','Admin_Rating']
```

```
corr=df[cols].corr()
sns.heatmap(corr,ax=ax3)
ax3.set_title('Service Rating Correlation')
ax4=fig.add_subplot(3,4,4)
contigency=pd.crosstab(df['Doctor_Rating'],df['Doctor_Recommend'],normalize='index')
contigency.plot(kind='bar',stacked=True,ax=ax4)
ax4.set_ylabel('Proportion Recommend')
ax4.set_title('Recommend Rate By Doctor Rating')
ax5=fig.add_subplot(3,4,5)
df['Visit_Type'].value_counts().plot(kind='pie',autopct='%1.1f%%',ax=ax5)
ax5.set_title('Distribution of patients Visit Type')
ax6=fig.add_subplot(3,4,6)
sns.boxenplot(data=df, x="Visit_Type", y="Doctor_Empathy",ax=ax6)
ax6.set_title("Doctor Empathy Scores Across Visit Types")
ax7=fig.add_subplot(3,4,7)
sns.violinplot(x="Gender",y='Doctor_Recommend',data=New_df,hue='Gender',
inner="quartile",ax=ax7)
ax7.set_title("Overall Satisfaction by Gender")
ax7.set_xlabel("Gender")
ax7.set_ylabel("Satisfaction Score")
ax8=fig.add_subplot(3,4,8)
sns.barplot(data=result, x='Age_group', y='percent_satisfied', hue='Age_group',ax=ax8)
ax8.set_ylabel('Percent Satisfied')
ax8.set_xlabel('Age Group')
ax8.set_title('Satisfaction Rate by Age Group')
ax9=fig.add_subplot(3,4,10)
sns.barplot(
    data=summary,
    x='Visit_Type',
    y='percent_satisfied',
    hue='Visit_Type',ax=ax9
```

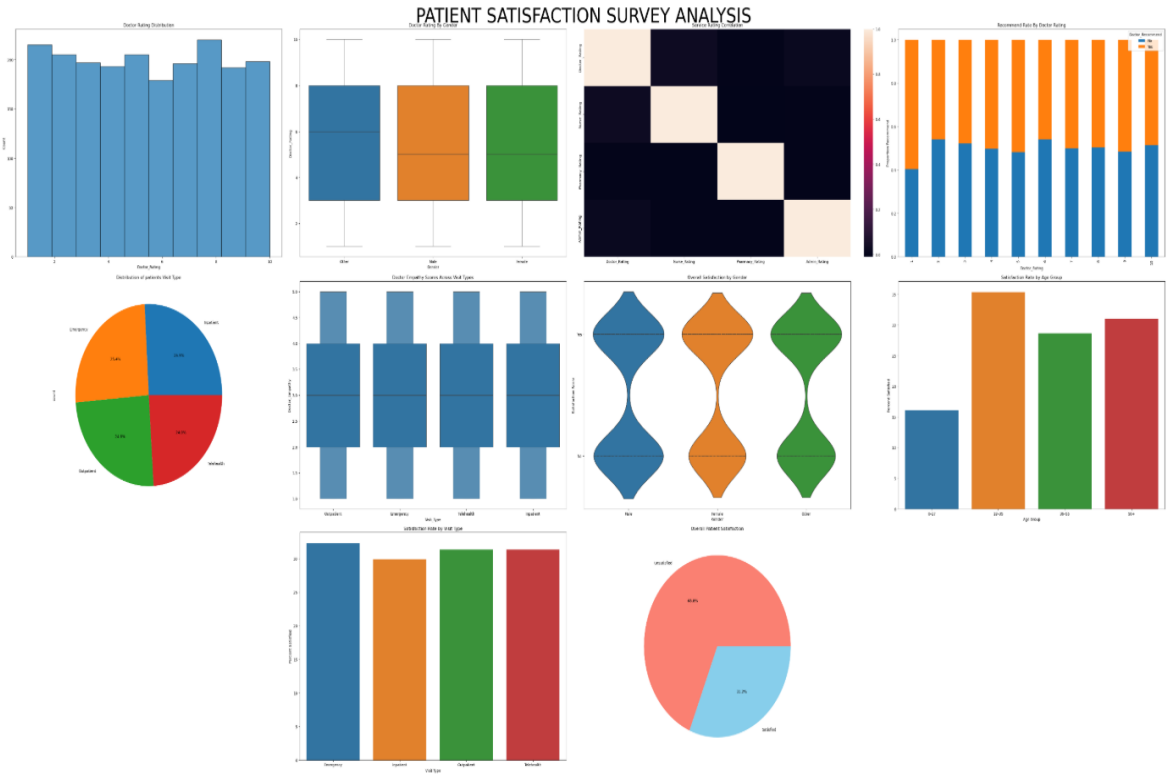
```
)
ax9.set_ylabel('Percent Satisfied')
ax9.set_xlabel('Visit Type')
ax9.set_title('Satisfaction Rate by Visit Type')
ax10=fig.add_subplot(3,4,11)
counts = df['overall_satisfied'].value_counts().sort_index()
labels = ['Unsatisfied', 'Satisfied']
colors = ['salmon', 'skyblue']
plt.pie(
    counts,
    labels=labels,
    colors=colors,
    autopct='%1.1f%%'
)
ax10.set_title('Overall Patient Satisfaction')
fig.suptitle("PATIENT SATISFACTION SURVEY ANALYSIS", fontsize=50, y=1.02)
plt.show()
```

Share Insights

Created Dashboard

This dashboard confirms that patient satisfaction in your facility is multi-dimensional—rooted in service quality, communication clarity, and operational efficiency. By focusing on strategic improvements across doctor communication, administrative transparency, pharmacy experience, and nursing engagement, your organization can meaningfully enhance patient experience, build trust, and strengthen loyalty.

PATIENT SATISFACTION SURVEY ANALYSIS DASHBOARD



Conclusion:

Our patient satisfaction survey across key service domains Doctors, nursing, pharmacy, fee transparency, and administrative support revealed that clear doctor-patient communication and efficient pharmacy service significantly drive positive patient sentiment, while opacity around fees and cumbersome administrative processes consistently correlate with dissatisfaction. Though nurses generally receive acceptable ratings, there is opportunity to elevate empathy and responsiveness. These insights emphasize that delivering transparent cost information, streamlining operations, providing effective medication counselling, and enhancing bedside manner can meaningfully uplift overall patient satisfaction, loyalty, and perceived quality of care.